Université Joseph Fourier de Grenoble (UJF)

Modèles d'habillage de surface
pour la synthèse d'images

Sylvain LEFEBVRE

Thèse présentée pour l'obtention du titre de
Docteur de l'Université Joseph Fourier
Spécialité Informatique
Arrêté Ministériel du 5 juillet 1984 et du 30 mars 1992
Préparée au sein du laboratoire
EVASION-GRAVIR/IMAG-INRIA. UMR CNRS C5527.

Composition du jury :

Fabrice	NEYRET	Directeur de thèse
Claude	PUECH	Président du Jury
Michael	MCCOOL	Rapporteur
Mathias	PAULIN	Rapporteur
Mark	HARRIS	Examinateur
Jean–Michel	DISCHLER	Examinateur
Przemek	PRUNSINKIEWICZ	Encadrant Eurodoc

Remerciements

Mes premiers remerciements vont à mes parents, Maryse et Jean-Claude Lefebvre, et à ma famille, pour leur soutien et leurs encouragements sans faille. Mais je n'aurais que difficilement traversé ces trois années sans ma compagne, Françoise, qui m'a apporté aide, soutien, et bonheur au quotidien, malgré les nombreuses séparations imposées par un emploi du temps trop chargé et de nombreux voyages. Je remercie aussi tout particulièrement mon frère, Stéphane Lefebvre, à qui je dois, entre–autre, ma passion pour les jeux vidéos, mes premiers pas de Basic et ma première lecture, plus tard, du Richie–Kernighan.

Je remercie également Fabrice, pour m'avoir initié au monde de la recherche, pour sa disponibilité, les nombreuses discussions et idées échangées, et la liberté qu'il m'a permise durant ces trois années de thèse. Merci aussi à Jean–Luc Koning, pour ses conseils précieux au travers de trois années de monitorat à l'ESISAR. Je souhaite également remercier Przemek Prunsinkiewicz, pour m'avoir accueilli à l'Université de Calgary dans d'excellentes conditions ; et Hugues Hoppe, pour m'avoir offert l'incroyable opportunité d'effectuer un stage, puis un postdoc à Microsoft Research.

Mille mercis à Jérôme Decroix et Thomas Bruel, grâce à qui j'ai découvert l'informatique graphique (POVray, les "coding party"), et qui m'ont beaucoup appris. C'est ensemble que nous avons mis les premières miettes de pizza dans nos claviers (tout un symbole !). Merci aussi à Alexis Angelidis, graphiste et codeur de talent, avec qui nous avons passé de nombreux bons moments à créer un raytracer, puis un jeu (Etees), puis à nous détendre autour de quelques Guiness bien méritées.

Merci à Samuel Hornus et à Jérome Darbon pour les nombreuses aventures de "deadline" inoubliables, et les recherches menées ensemble ! Merci également aux compagnons de thèse, Stéphane Grabli, Stéphane Guy, Sylvain Paris et à tous les thésards d'ARTIS et EVASION pour les bons moments et leur aide précieuse.

Et puis bien sûr, un grand merci aux équipes d'ARTIS et EVASION, pour quatre années de bonne humeur, pauses café, discussions improvisées et pour toutes les choses, indénombrables, que j'ai apprises à leurs côtés. Merci en particulier à Xavier Decoret, Lionel Reveret, Gilles Debunne, Phillipe Decaudin, Joelle Thollot, pour leurs conseils et les nombreux échanges d'idées.

Enfin, merci aux copains, Antoine, Philippe, Charly, Karine, Renaud, Lili, Eric, Vincent, Cindy pour les nombreuses soirées de détente. Un grand merci également à l'équipe du 17, Matthieu, Louis, Philippe, Cyril, Jean–Baptiste, pour les soirées autour du thème du jeu vidéo, les week–end "ShufflePuck Cafe" et les modèles 3D et textures que le lecteur retrouvera dans ces pages !

Table des matières

I Introduction et état de l'art **13**

1 Introduction **15**
1 Un problème de tapisserie . . . 17
2 Objectifs de la thèse . . . 18
3 Contributions . . . 19
4 Organisation du document . . . 20

2 Etat de l'art **23**
1 Rappel sur la création d'images de synthèse . . . 23
1.1 Représentation des formes . . . 23
1.2 Eclairage . . . 24
1.3 Algorithmes de rendu . . . 24
1.3.1 L'écran . . . 24
1.3.2 Lancer de rayon . . . 24
1.4 Rendu projectif . . . 26
1.5 Modèles d'illumination locale . . . 28
1.6 Moduler l'illumination locale avec un modèle d'habillage . . . 28
2 Modèles d'habillage pour les surfaces quelconques . . . 30
2.1 Placage de texture standard, par paramétrisation de surface . . . 30
2.1.1 Principe . . . 30
2.1.2 Paramétrisation planaire . . . 30
2.1.3 Stockage des données de texture . . . 35
2.1.4 Accès aux données de texture . . . 35
2.1.5 Interpolation . . . 35
2.1.6 Limitations du placage de texture . . . 37
2.1.7 Extensions du placage de texture . . . 40
2.2 Cartes d'environnement . . . 45
2.2.1 Principe . . . 45
2.2.2 Cartes d'environnement généralisées . . . 46
2.2.3 Conclusion sur les cartes d'environnement . . . 47
2.3 Particules de textures . . . 48
2.3.1 Principe . . . 48

2.3.2 Représentations utilisées et difficultés . . . 48
2.4 Habillage en volume . . . 49
2.4.1 Principe . . . 49
2.4.2 Données d'habillage en volume . . . 50
3 Modèles d'habillage pour les terrains . . . 52
3.1 Textures explicites . . . 52
3.2 Pavages périodiques du plan . . . 52
3.3 Pavages apériodiques du plan . . . 53
3.4 Pavages de surfaces . . . 56
3.5 Quelques techniques utilisées dans le jeu vidéo . . . 56
3.5.1 Mélange de différentes échelles . . . 57
3.5.2 Transitions entre matériaux . . . 57
3.5.3 Ajout de hautes fréquences . . . 58
4 Filtrage . . . 60
4.1 Origine du problème . . . 60
4.2 Sur–échantillonnage . . . 62
4.3 Filtrage et modèles d'habillage . . . 62
4.3.1 Filtrage pour le placage de texture . . . 62
4.3.2 Filtrage et combinaison de textures explicites . . . 66
4.3.3 Filtrage des textures procédurales . . . 66
4.3.4 Filtrage des modèles d'habillage en général . . . 66
4.4 Note sur la qualité de filtrage . . . 67
4.5 Limites du filtrage pour les modèles d'habillage . . . 67
4.5.1 Aliasing géométrique . . . 67
4.5.2 Interaction avec le modèle d'éclairage . . . 68
5 Techniques de génération d'aspect de surface . . . 69
5.1 Techniques de peinture . . . 69
5.2 Synthèse de textures . . . 69
5.2.1 Textures procédurales . . . 70
5.2.2 Synthèse à partir d'un échantillon . . . 71
5.2.3 Méthodes basées sur la simulation . . . 73
5.3 Méthodes respectant les contraintes des pavages . . . 73
5.4 Textures animées ou dynamiques . . . 74
6 Modèles d'habillage et représentations alternatives . . . 75
6.1 Textures Volumiques . . . 75
6.2 Imposteurs . . . 75
6.3 Rendu par point . . . 76
7 Concevoir des modèles d'habillage pour les applications interactives . . . 77
7.1 Cartes accélératrices . . . 77
7.2 Géométrie et applications interactives . . . 79
8 Résumé et conclusion . . . 80

II Habillage à base de motifs pour terrains et surfaces paramétrées 83

3 Textures procédurales à base de motifs 85
1 Notre approche : Textures procédurales à base de motifs . . . 86
1.1 Cartes et textures . . . 88
1.2 Combiner les composants . . . 88
1.3 Composants pour le choix et le positionnement des motifs . . . 88
1.4 Transitions . . . 96
1.5 Animation . . . 99
2 Notes d'implémentation . . . 102
2.1 Encodage des tables en 2D . . . 102
2.2 Encodage d'entiers dans des textures . . . 103
3 Connecter les composants pour générer des textures : étude de cas . . . 103
4 Filtrage des textures générées . . . 108
4.1 Solutions pour nos textures procédurales . . . 108
4.2 Discussion sur l'interpolation . . . 110
5 Résultats . . . 111
6 Conclusion . . . 112

4 Application : rendu de gouttes d'eau sur surfaces 115
1 Positionnement des gouttes d'eau . . . 115
2 Texture d'humidité . . . 116
3 Rendu . . . 116
4 Conclusion . . . 117

5 Application : ombres portées par un grand ensemble d'objets 121
1 Le contexte . . . 121
1.1 Génération de la forêt . . . 122
1.1.1 Principe . . . 122
1.1.2 Le carré de forêt . . . 122
1.1.3 Création des arbres à partir des graines . . . 123
2 Ombres d'objets sur un terrain . . . 123
3 Conclusion . . . 127

III Textures de haute résolution plaquées sur surfaces quelconques 129

6 Gestion de textures de haute résolution plaquées sur des surfaces quelconques 131
1 Travaux sur le chargement progressif . . . 132
2 Organisation des données de texture . . . 134
3 Vue d'ensemble de notre architecture . . . 136
4 Détermination des cases de texture utiles : carte de chargement de texture . . . 136
4.1 Calcul d'un niveau de la carte de chargement . . . 137

4.1.1 Elimination des faces cachées . . . 139
4.1.2 Obtenir un résultat conservatif . . . 140
4.1.3 Calcul du niveau de détail . . . 141
4.1.4 Prise en compte de l'occlusion . . . 143
4.2 Calcul de tous les niveaux d'une carte de chargement . . . 144
4.3 Génération des requêtes de chargement et déchargement . . . 145
5 Cache de Texture . . . 145
5.1 Saturation de la mémoire . . . 146
5.2 Utiliser le cache comme une texture . . . 146
6 Producteur de Texture . . . 147
7 Résultats . . . 148
7.1 Résultats de l'algorithme de carte de chargement . . . 150
7.2 Rendu sous forte contrainte mémoire . . . 151
7.3 Performances globales de notre architecture . . . 151
7.4 Points faibles et limitations . . . 151
8 Conclusion . . . 152

IV Modèles d'habillage sans paramétrisation planaire globale 153

7 Textures hiérarchiques en volume pour processeurs graphiques 155
1 Textures volumiques hiérarchiques pour les cartes graphiques . . . 156
1.1 Définition . . . 156
1.2 Implémentation . . . 156
1.2.1 Stockage . . . 157
1.2.2 Accès aux données . . . 158
1.3 Autres optimisations . . . 160
1.4 Note sur l'encodage des coordonnées . . . 161
2 Habillage avec une texture hiérarchique . . . 161
2.1 Habillage de la surface . . . 161
2.2 Qualité de rendu . . . 162
2.3 Conversion en une texture standard . . . 162
2.4 Résultats . . . 164
3 Conclusion et travaux futurs . . . 164

8 Texture dynamique sur surface 167
1 Automates cellulaires et habillage de surface . . . 167
2 Principe de notre approche . . . 168
3 Résultats . . . 169
4 Conclusion . . . 170

9 Textures composites **171**
1 Textures Composites . . . 174
2 Gestion des sprites dans la grille hiérarchique . . . 174
2.1 Stockage des attributs des sprites . . . 175
2.2 Ajout d'un sprite, construction de la grille hiérarchique . . . 175
2.3 Projection de la texture du sprite sur la surface . . . 177
2.4 Combiner les contributions des différents sprites . . . 179
3 Filtrage de la texture composite . . . 179
4 Nouveaux habillages et résultats . . . 181
4.1 Nouveaux habillages . . . 181
4.2 Performances . . . 183
5 Conclusion . . . 184

V Modèle d'habillage dédié : génération d'écorces d'arbre réalistes 185

10 Modèle d'habillage pour la génération d'écorces d'arbre réalistes **187**
1 Présentation de notre approche . . . 188
2 Travaux antérieurs . . . 191
2.1 Travaux en synthèse d'image . . . 191
2.1.1 Travaux sur les écorces . . . 191
2.1.2 Travaux sur la simulation de fractures . . . 192
2.2 Eléments de physique . . . 192
3 Simulation d'une plaque d'écorce . . . 193
3.1 Etude de cas et hypothèses . . . 193
3.2 Modèle d'écorce . . . 194
3.2.1 Modélisation du matériau . . . 195
3.2.2 Création d'une fracture . . . 197
3.2.3 Propagation des fractures . . . 198
3.2.4 Elargissement des déchirures . . . 199
3.3 Habillage de l'intérieur des déchirures . . . 199
4 Résultats de simulation d'une plaque d'écorce . . . 202
5 Habillage d'un arbre avec notre modèle d'écorce . . . 203
5.1 Attachement de l'écorce au substrat . . . 205
5.2 Croissance de l'arbre . . . 206
5.3 Simulation des fractures le long du tronc et des branches . . . 207
5.3.1 Adapter les propriétés mécaniques des bandelettes . . . 207
5.3.2 Adapter la propagation des fractures . . . 207
5.4 Habillage de la simulation . . . 209
6 Résultat de l'habillage d'un arbre par notre modèle d'écorce . . . 209
7 Conclusion . . . 210

VI Conclusions et perspectives 215

11 Conclusions et perspectives 217
1 Bilan . . . 217
1.1 Textures procédurales à base de motifs . . . 217
1.2 Textures composites . . . 218
1.3 Gestion de textures de haute résolution sur des surfaces quelconques . . . 219
1.4 Textures hiérarchiques en volume pour processeurs graphiques . . . 220
1.5 S'attaquer à la résolution de nombreux non–dits concernant la qualité . . . 220
1.6 Donner les outils pour réaliser des apparences de surface réalistes . . . 221
1.7 Unification et transparence d'utilisation . . . 222
1.8 Modèles d'habillage pour applications interactives . . . 222
2 Perspectives . . . 224
2.1 Un modèle d'habillage universel ? . . . 224
2.2 Aspects de surface réalistes : outils de création et variété . . . 225
2.3 L'aliasing : un problème d'actualité . . . 225

Table des figures

1.1 Les modèles d'habillage génèrent l'apparence des surfaces 16

2.1 Lancer de rayon 25
2.2 Algorithme de rasterisation 26
2.3 Modèle d'éclairage de Phong 29
2.4 Placage de texture 31
2.5 Les terrains sont représentés par des cartes de hauteur. 32
2.6 Mise à plat de maillage triangulaire avec une bordure unique. 33
2.7 Atlas de texture 34
2.8 Exemple d'interpolation linéaire 36
2.9 Paramétrisation d'un visage. 38
2.10 Interpolation linéaire erronée due aux discontinuités. 39
2.11 Compression des vides. 41
2.12 Carte de silhouettes 42
2.13 Texture dépendant du point de vue 43
2.14 Cartes d'environnement utilisée pour simuler des réflections 45
2.15 Cartes d'environnement utilisée comme une texture 46
2.16 Cartes d'environnement généralisées [THCM04] 47
2.17 Particules de textures 48
2.18 Textures procédurales définies dans un volume. 50
2.19 Texture volumique hiérarchique pour les surfaces (*octree textures*). 51
2.20 Pavage périodique du plan. 53
2.21 Pavages apériodiques 54
2.22 Pavés de Wang 54
2.23 Pavage de surface 56
2.24 Transition entre matériaux pour pavages apériodiques 58
2.25 Des effets d'aliasing apparaissent 60
2.26 Les pixels sont influencés par un morceau de la surface. 61
2.27 Empreinte du pixel dans la texture 62
2.28 Pyramide de MIP–mapping 64
2.29 Filtrage erroné du aux discontinuités. 65
2.30 Différents aspects générés par des textures procédurales. 71
2.31 Synthèse de texture à partir d'échantillon contrôlée par l'utilisateur. 72

2.32 Imposteurs : un enchevêtrement de plans texturés représente l'objet. 76
2.33 Chaîne des traitements (*pipeline*) d'un processeur graphique (version simplifiée). 77

3.1 Textures procédurales à base de motifs 87
3.2 (a) Pavage virtuel et (b) aliasing dans les cartes aléatoires virtuelles 89
3.3 Cartes d'indirections 90
3.4 Positionnement des motifs. 92
3.5 Gestion du débordements en dehors des cases. 92
3.6 Positionnement explicite d'un motif. 93
3.7 Carte de probabilité. 94
3.8 Opérateurs de *dithering* 1D et 2D 95
3.9 Interpolation d'une carte de matériaux. 96
3.10 Pavages apériodiques générés par notre méthode 97
3.11 Calcul du numéro du pavé de transition 99
3.12 Séquence d'animation. 100
3.13 Suppression des discontinuités temporelles. 101
3.14 Distribution aléatoire de motifs sur un terrain. 102
3.15 Pavages apériodiques générés par une combinaison des composants de base . . . 104
3.16 Schéma du pavage apériodique simple 105
3.17 Pavage apériodique avancé 106
3.18 Distribution aléatoire de motifs 106
3.19 Distribution aléatoire contrôlable de motifs 107
3.20 Grille de motifs animés 108
3.21 Motifs animés. 109

4.1 Texture d'humidité de la surface. 117
4.2 Effet de réfraction des gouttes. 118
4.3 Simulation de gouttes tombant le long d'une surface. 119

5.1 Génération de forêt. 123
5.2 Résultats de notre méthode de génération et rendu interactif de forêt. 124
5.3 Carte d'ombre. 125
5.4 Principe de projection d'ombres. 126
5.5 Ombres représentées par notre méthode. 127

6.1 Présentation d'ensemble de notre architecture de gestion de textures. 133
6.3 Structure pyramidale de la texture 134
6.4 Notre architecture gère uniquement les derniers niveaux 134
6.2 Organisation de notre architecture 135
6.5 Carte de chargement 137
6.6 Principe de l'algorithme de carte de chargement. 138
6.7 Carte de chargement pour un triangle. 139
6.8 Elimination des faces situées en dehors du rectangle de l'écran 140
6.9 La texture LOD pour $k = 3$ niveaux gérés par notre architecture. 142

6.10 Détermination du niveau de détail . . . 143
6.11 Stockage non continu des cases de texture. . . . 146
6.12 Mesures de performance. . . . 149
6.13 Prise en compte de l'occlusion . . . 150
6.14 Rendu sous forte contrainte mémoire. . . . 151

7.1 Un octree construit autour d'une surface . . . 156
7.2 Stockage de la grille hiérarchique . . . 157
7.3 Accès dans la grille hiérarchique . . . 159
7.4 Code Cg du *fragment program* permettant d'accéder au N^3–tree . . . 160
7.5 Conversion d'une texture hiérarchique en texture 2D standard. . . . 163
7.6 Extrapolation automatique des couleurs en dehors des bordures. . . . 163
7.7 Texture volumique accélérée par la carte graphique . . . 164

8.1 Organisation des données pour l'automate cellulaire. . . . 168
8.2 Liquide s'écoulant le long d'une surface. . . . 169

9.1 Exemple de modèles habillés par nos textures composites. . . . 172
9.2 Principe de nos textures composites . . . 173
9.3 Structure de données. . . . 176
9.4 Insertion d'un nouveau sprite . . . 177
9.5 Habillage de la surface avec des écailles rigides. . . . 181
9.6 Performances des exemples de la figure 9.1. . . . 183
9.7 Taille mémoire des exemples de la figure 9.1 . . . 184

10.1 Ecorces générées à la surface d'arbres par notre modèle d'habillage . . . 188
10.2 Ecorce réelles . . . 189
10.3 Coupe de troncs d'arbres . . . 189
10.4 Les déchirures tendent à s'entrelacer (images réelles). . . . 193
10.5 Principe de fonctionnement du simulateur . . . 195
10.6 Caractéristique d'un déchirure et bandelettes du modèle . . . 196
10.7 Raideur des éléments de l'épiderme de l'écorce. . . . 196
10.8 Algorithme de création et propagation de fractures. . . . 197
10.9 Evaluation du critère de fracture. . . . 197
10.10 Re–fracture de l'écorce. . . . 198
10.11 Résultat de simulation de déchirures avec différents paramètres. . . . 199
10.12 Habillage de la simulation. . . . 200
10.13 Reconstruction des silhouettes des déchirures. . . . 200
10.14 Différents modes d'ouverture des déchirures. . . . 201
10.15 Courbes de profil pour l'ajout de relief. . . . 202
10.16 Comparaison entre une écorce simulée (à droite) et une écorce réelle (à gauche). . 203
10.17 Différentes écorces de la même espèce . . . 204
10.18 Fractures fragiles. . . . 204
10.19 Propagation de fracture durant la croissance de l'arbre . . . 205

10.20Modèle d'attachement de l'écorce au substrat. 206
10.21Les bandelettes sont déformées par les courbes du tronc. 208
10.22Propagation de fractures sur l'arbre 208
10.23Ecorce simulée sur des troncs de forme complexe. 211
10.24Résultat avec embranchement. 212
10.25Embranchement complexe. 213

Première partie

Introduction et état de l’art

chapitre 1

Introduction

Des surfaces lisses aux couleurs uniformes, des lumières brillantes aux reflets éclatants. Voici peut être le principal défaut des images générées par ordinateur : elles sont trop parfaites. Combattre cette perfection pour se rapprocher du réel est devenu l'un des enjeux majeurs de la synthèse d'image. Le cinéma, l'industrie du jeu vidéo, les simulateurs, tous sont avides de réalisme. En fait, ce n'est pas tant le réalisme que l'immersion qui est recherchée. Il s'agit de plonger l'utilisateur dans un simulacre de réalité indiscernable du réel : un monde virtuel aussi riche, complexe et vivant que la réalité qui nous entoure.

Cependant nous sommes encore loin du but. Pour représenter les formes, il faut en créer des modèles mathématiques, qui seront utilisés pour calculer les images. Certaines formes sont intrinsèquement complexes : un arbre, un nuage, une chevelure. D'autres - la plupart - peuvent être simplement décrites à l'aide de formes simples. Un soigneux agencement de sphères, cylindres et triangles permet de représenter une grande variété d'objets, naturels ou manufacturés. Leur complexité ne réside pas dans les contours et les volumes, mais dans *l'apparence* de leur surface. Les détails, imperfections et motifs de surface ne sont pas nécessaires a la compréhension des formes. Ils sont néanmoins primordiaux pour enrichir l'aspect visuel, pour *habiller* la surface d'une apparence réaliste.

Cette distinction entre forme et aspect de surface permet d'introduire la notion de *modèle d'habillage* de surface. Celui–ci décrit la variation, le long de la surface géométrique, des propriétés du matériau (couleur, brillance, transparence, orientation, ...). La forme d'un mur de briques sera efficacement représentée par un simple rectangle, alors que les motifs colorés seront générés par un modèle d'habillage faisant varier la couleur.

Cette séparation entre forme et détails de surface présente de nombreux avantages. Les détails peuvent être ignorés par les algorithmes manipulant la géométrie des objets (détermination des parties visibles, animation, simulation physique, etc ...). Ils ne sont pris en compte que lors du calcul d'éclairage local, lorsque la lumière simulée interagit avec la surface pour engendrer une couleur. La figure 1.1 montre que les applications actuelles utilisent cette approche pour augmenter le réalisme des images sans sacrifier performance et qualité.

Cependant, alors que les exigences des utilisateurs et artistes ne cessent d'augmenter en matière de scènes très détaillées, de nombreuses difficultés perdurent. En premier lieu l'apparence de la surface doit être encodée et stockée en mémoire. La *quantité* de détails peut très vite

dépasser la capacité mémoire disponible. En second lieu, *attacher* convenablement les détails le long de la surface présente un grand nombre de difficultés et des défauts visuels tels que des effets de distorsions sont souvent introduits, brisant le réalisme des images produites. Enfin, la *création* des détails peut demander un temps considérable aux artistes, d'autant plus que certains aspects de surfaces sont extrêmement difficiles à reproduire (roc, écorce, surfaces abîmées, ...) Ces trois problèmes sont liés car la manière de représenter les détails et de les attacher sur la surface influe sur la façon dont les artistes vont pouvoir créer les aspects de surface.

Dans cette thèse nous proposons de nouveaux modèles d'habillage de surface qui permettent de générer des apparences très détaillées à faible coût mémoire, et de manière semi–automatique, tout en s'assurant que l'artiste puisse conserver un fort contrôle sur les apparences générées. Nous proposons également des extensions aux modèles d'habillage existants permettant d'en dépasser certaines limitations (occupation mémoire, qualité et performance d'affichage). La plupart de nos modèles sont conçus pour être implémentés sur les processeurs graphiques actuels.

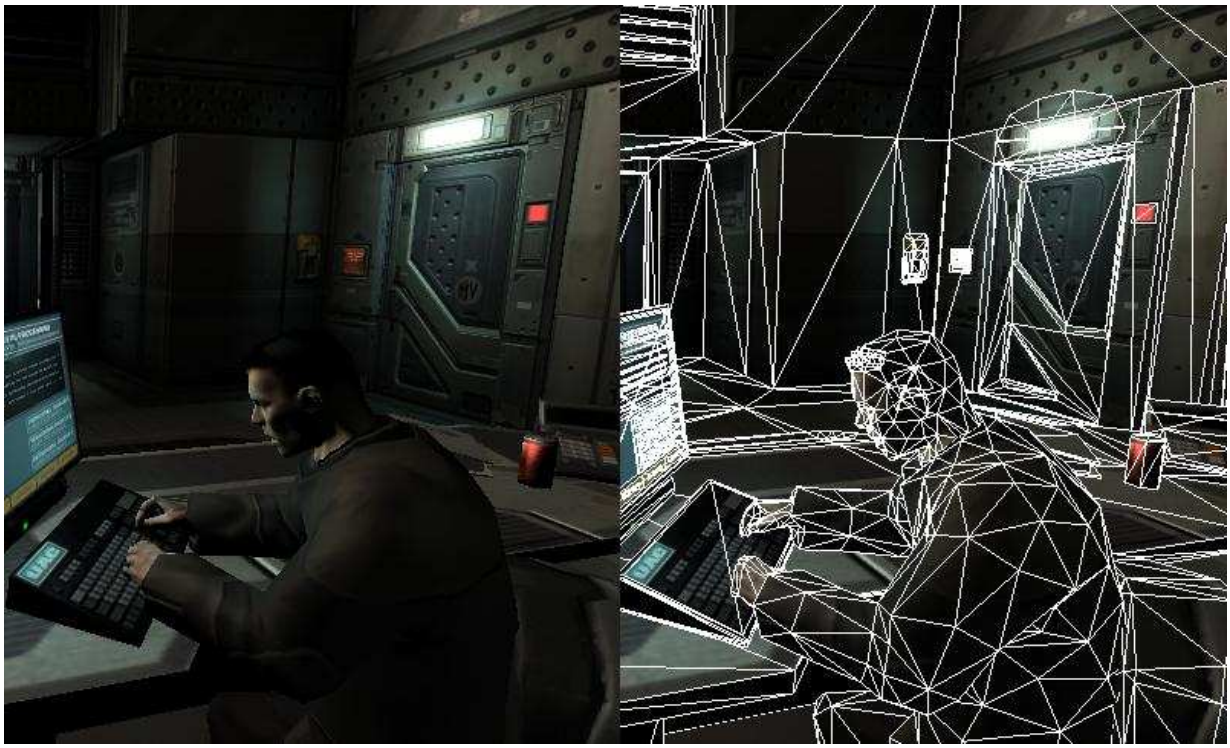


FIG. 1.1 – Les modèles d'habillage génèrent l'apparence des surfaces

Malgré l'utilisation d'un nombre relativement faible de primitives géométriques (à droite), de nombreux détails sont visibles dans l'image finale (à gauche). Ces détails sont créés par les modèles d'habillage qui font varier les paramètres de rendu (couleur, brillance, etc ...) le long des surfaces. Le modèle d'habillage utilisé ici est le placage de texture. Cette image est tirée du jeu Doom III, id software / Activision 2003).

1 Un problème de tapisserie

Afin de rendre plus intuitive la notion de modèle d'habillage , nous proposons une analogie avec le problème concret suivant : imaginons que nous souhaitions décorer un objet dont la surface est lisse, par exemple une statuette en métal, et que nous ayons à notre disposition du papier, des ciseaux, de la colle et des feutres. Il n'est pas possible d'utiliser les feutres directement sur le métal. Il nous faut donc coller du papier sur la surface, et dessiner sur le papier l'aspect que nous souhaitons donner à l'objet. Ceci correspond à ce qui se passe lorsque l'on doit appliquer un modèle d'habillage à la surface géométrique de l'objet. Généralement un espace de référence (ici, la feuille), dans lequel est dessinée l'apparence finale, est attachée sur le modèle géométrique de la surface. Supposons que nous souhaitions donner l'illusion que la statuette est en bois. On peut imaginer plusieurs approches :

- Dessiner un aspect de bois en trompe l'oeil sur la feuille de papier, puis coller la feuille sur la surface de la statuette. Le recours aux ciseaux va s'imposer si l'on veut que la forme de l'objet soit respectée. Or ces coupures vont se voir puisque les veines du bois, soigneusement peintes sur la feuille à plat, ne vont plus se correspondre sur la surface. Cependant la feuille peut être photocopiée et utilisée sur un autre objet, permettant un gain de temps considérable.
- Coller la feuille vierge sur la statuette, puis dessiner l'aspect de bois sur l'objet. Cette fois les discontinuités ne seront pas visibles. Si la feuille est dépliée on pourra constater que le dessin du bois, sur la feuille à plat, est distordu et discontinu à cause des coupures et des pliages. Ca n'est pas gênant pour cet objet mais cela interdit de réutiliser la feuille sur un autre objet.
- Découper un petit morceau de papier puis peindre dessus un aspect de bois. Ce petit morceau est dupliqué plusieurs fois, par exemple à l'aide d'une photocopieuse. Les petits morceaux seront ensuite collés sur la surface, tels des autocollants. Ce collage est plus facile car étant de petites tailles les autocollants seront moins sujets aux déformations dues aux courbes de la statuette. Ils peuvent être agencés de manière à ce que les bords se correspondent à peu près pour donner l'illusion d'une apparence continue. Cette méthode est plus rapide que de peindre une apparence de bois sur toute la surface. Par contre il est possible que des répétitions soient visibles sur le résultat final.

Chacune de ces méthodes constitue un modèle d'habillage : elles définissent comment des propriétés de matériau (ici la couleur) sont appliquées sur la géométrie de l'objet. On constate que le choix de la méthode influe sur le temps de réalisation, la qualité du résultat, et la manière de peindre (*i.e.* spécifier) l'apparence désirée sur la surface.

Par ailleurs, il semble extrêmement délicat de peindre à la main un aspect ressemblant à du bois ; il serait préférable d'utiliser une photographie. Mais, dans ce cas, des discontinuités risquent d'apparaître, comme avec la première méthode. Idéalement, il faudrait plutôt construire une machine spécialisée, capable de peindre automatiquement l'aspect de surface, pour un objet donné, en tenant compte de la méthode employée pour attacher l'apparence sur la surface.

Bien sûr, il ne s'agit là que d'esquisser les premières difficultés rencontrées. D'autres problèmes existent dans le cadre de la synthèse d'image. Nous les analyserons en détail dans l'état de l'art (chapitre 2). Nous verrons également que selon le type d'apparence choisie (détails uni-

formes ou non, aspect final peint par un artiste ou par un algorithme) et le type de surface (quasi–planaire ou quelconque) différentes méthodes s'appliquent. L'apparence de surface peut parfois également être animée (gouttes d'eau tombant le long de la surface, impacts, traces de pas, ...) ce qui demande des méthodes dédiées.

2 Objectifs de la thèse

Le but principal d'un modèle d'habillage est d'enrichir des surfaces simples en leur conférant une *apparence détaillée*, ceci afin de produire des *images de qualité*. L'esprit est de le faire à *moindre coût*. Le coût regroupe ici deux éléments : la *taille des données* mais également la *difficulté à créer* des aspects de surface intéressants pour l'utilisateur.

Ces éléments sont liés. Un modèle d'habillage efficace mais avec de fortes contraintes sur le contenu empêchera l'artiste de s'exprimer : l'apparence des surfaces peut être appauvrie par les contraintes imposées. Un modèle d'habillage utilisant une trop grande quantité de mémoire ne pourra être utilisé efficacement car il faudra charger progressivement les données (au risque de pénaliser les performances), ou bien avoir recours à des algorithmes de compression qui peuvent réduire la qualité des images. Il est important de réaliser que la quantité de détails dont il est question peut dépasser de plusieurs ordres de grandeur la capacité mémoire des systèmes d'affichage actuels. Au contraire, un modèle d'habillage capable de créer une grande variété d'aspects de manière semi–automatique à partir de peu d'informations consommera peu de mémoire tout en simplifiant le travail de l'artiste.

A ceci s'ajoute l'exigence de performances interactives (environ 30 images par seconde), importante à deux égards :

- En premier lieu, nous souhaitons pouvoir utiliser ces modèles dans le plus large contexte possible, et notamment dans les applications interactives telles que simulateurs et jeux vidéos. Si le modèle propose de bonnes performances tout en offrant des résultats de grande qualité visuelle, il pourra alors tout naturellement être également utilisé dans des applications de calcul d'image très réalistes. Un modèle autorisant des performances temps réel avec une quantité de données raisonnable sera d'autant plus à même de supporter les très grandes quantités de données nécessaires à la synthèse d'images très réalistes.
- En second lieu, l'interactivité est particulièrement importante du point de vue de la création. Dans l'industrie du cinéma, elle permet un gain de temps considérable aux artistes et réalisateurs qui peuvent rapidement tester et ajuster les effets visuels (par essai–erreur). Cette notion prend également tout son sens lorsque le rendu final exigera plusieurs semaines de calcul : il devient crucial de pouvoir très vite pré–visualiser et vérifier tout changement apporté à une scène.

Notre objectif est donc de proposer des modèles d'habillage permettant des apparences de surface **riches et détaillées**, une grande **liberté d'expression** à l'artiste en terme de **variété des apparences** possibles et de **facilité de création**, tout en permettant un **affichage rapide** avec une **occupation mémoire faible** et sans sacrifier la **qualité visuelle**. Nous nous attacherons à concevoir nos modèles pour qu'ils puissent tirer parti des dernières avancées technologiques en matière

de cartes d'accélération graphique (processeurs graphiques programmables), tant du point de vue des structures de données que de la forme très spécifique de parallélisme des algorithmes. Ceci est la condition sine qua none pour accéder à l'interactivité.

3 Contributions

Nos contributions consistent en de nouvelles approches pour l'habillage de surface, qui suivent les objectifs que nous nous sommes fixés en terme de richesse de détails, de variété des apparences, de qualité d'affichage, et de facilité d'utilisation :

- **Habillage de très haute résolution apparente, à faible coût mémoire :** Nous proposons des modèles d'habillages qui reposent sur la combinaison semi–automatique de motifs le long d'une surface pour créer des habillages variés et très détaillés sur de larges surfaces, avec une occupation mémoire très faible (chapitres 3 et 9). Nos modèles permettent en particulier de ne stocker les motifs qu'une seule fois en mémoire, et de les appliquer localement sur la surface en utilisant uniquement les informations de positionnement, lesquelles peuvent être explicitées par un artiste, ou bien générées à la volée, de manière automatique mais contrôlée (contrôle de la distribution spatiale des motifs, de variations d'aspect aléatoires, ...).
- **Ne charger en mémoire que le nécessaire, à la résolution requise :** Nous proposons une extension au placage de texture qui permet d'afficher des objets (possiblement animés), recouverts de textures de très haute résolution dont la taille peut dépasser de plusieurs ordres de grandeur la mémoire disponible, comme par exemple des personnages très détaillés, un paysage, ou un environnement urbain (chapitre 6).

  Les données de texture sont chargées progressivement depuis un média de masse ou un serveur distant, en fonction du point de vue, et au fur et à mesure que l'utilisateur évolue interactivement dans la scène. Notre architecture, conçue pour les processeurs graphiques actuels, permet d'unifier les solutions classiques de chargement progressif (cache de données, chargement asynchrone, priorité de chargement) dans un système *transparent* à l'utilisateur et applicable à tout type d'objet. Elle permet en outre d'utiliser des textures compressées ou procédurales, de façon souple et efficace.
- **Rendre disponibles dans les applications interactives des représentations complexes pour l'habillage de surface :** Au préalable, nous proposons une adaptation complète des *octree textures* [DGPR02, BD02] pour les processeurs graphiques, qui permet de rendre ces structures de données très génériques disponibles dans les applications graphiques interactives (chapitre 7). Ensuite, cette implémentation nous permet de créer des modèles d'habillage novateurs et de les rendre directement utilisables dans les applications interactives.
- **S'attaquer à la résolution de nombreux non–dits concernant la qualité visuelle :** (voir chapitre 2, section 2.1.6) Nous étudions et identifions, pour chacun de nos modèles d'habillage, les conditions dans lesquelles ils permettent de produire des images de qualité, sans défauts visuels tels que distorsions, couleurs parasites et aliasing (grouillement de

couleurs), souvent non mentionnés bien qu'inhérents à plusieurs méthodes très classiques. Nous proposons dans chaque cas des solutions, partielles sinon complètes, pour combattre ces effets indésirables.

- **Donner aux utilisateurs des outils pour réaliser des apparences de surfaces réalistes :** Nos modèles offrent différents niveaux de contrôle aux artistes sur l'apparence finale de la surface, allant du contrôle très fin, point par point, au contrôle très global, à grande échelle.

  Nous proposons également des exemples concrets, volontairement extrêmes, d'utilisation de nos modèles d'habillage génériques, pour obtenir des apparences de surface réalistes et jusqu'alors difficiles, voire impossibles, à réaliser :
  - au chapitre 4, la simulation temps réel de l'écoulement de gouttes d'eau sur une surface paramétrée ;
  - au chapitre 5, l'affichage des ombres détaillées d'un grand nombre d'objets (arbres d'une forêt) sur un terrain ;
  - au chapitre 8, la simulation temps réel (et interactive) de l'écoulement d'un liquide sur une surface quelconque, non paramétrée ;
  - au chapitre 9, une surface recouverte d'écailles qui coulissent les unes sur les autres en suivant les déformations d'un objet animé (un serpent).
- **Unification et transparence d'utilisation :** Nos modèles sont utilisables de manière transparente pour l'utilisateur, et visent à généraliser la séparation complète entre modèle d'habillage et représentation des formes. En particulier, il n'est jamais nécessaire de modifier ni d'introduire de nouvelle géométrie pour appliquer un habillage, contrairement à nombre de méthodes actuelles. (Notons que ceci permet en outre d'accroître performances et qualité visuelle). Nos modèles sont également tous rapidement convertibles en une texture 2D standard, ce qui permet de sélectionner à la volée la meilleure représentation, selon la situation.
- **Concevoir les modèles d'habillage pour les processeurs graphiques :** Nous avons pris soin de concevoir nos modèles (représentations et algorithmes) afin qu'ils puissent être implémentés efficacement sur les processeurs graphiques récents. Ceci permet de les utiliser directement dans les applications interactives avec de hautes performances. D'autre part, l'efficacité de nos représentations (structure de données, algorithmes), permet de garantir leur robustesse dans les situations extrêmes rencontrées lors du rendu d'images très réalistes, en particulier face à la gigantesque quantité de données utilisées.

4 Organisation du document

Nous commencerons, en première partie du document, par une présentation des modèles d'habillage existants et de leur rôle dans la création d'images de synthèse (chapitre 2). Le corps de l'ouvrage se compose ensuite de quatre autres parties, chacune dédiée à une approche spécifique du problème de l'habillage de surface.

Dans la seconde partie, nous présenterons notre premier modèle d'habillage générique à base de motifs. Ce modèle permet de créer une grande variété d'habillages de haute résolution pour les

terrains et les surfaces paramétrées, en combinant des motifs de base selon des règles spécifiées par l'artiste (chapitre 3). Nous verrons ensuite des applications concrètes de ce modèle d'habillage générique à deux problèmes spécifiques : un habillage animé simulant l'écoulement de gouttes d'eau le long d'une surface (chapitre 4) et un habillage permettant de représenter à faible coût les ombres de nombreux objets (les arbres d'une forêt) sur un terrain (chapitre 5)[1].

La troisième partie est consacrée à la présentation de notre architecture de chargement progressif de textures (chapitre 6). Il s'agit d'une extension du placage de texture standard, qui permet de ne charger que les parties utiles d'une texture, à la résolution requise, au fur et à mesure des déplacements de l'utilisateur, au lieu de saturer la mémoire et la bande passante avec des données qui ne sont pas utiles au rendu du point de vue courant. Notre architecture permet ainsi d'afficher des scènes utilisant des textures extrêmement détaillées, dont la taille dépasse largement la mémoire disponible et qui sont éventuellement stockées sur un média de masse, ou rapatriées via un réseau.

La quatrième partie introduit un ensemble de modèles qui ont tous pour point commun de rendre inutile le recours à une paramétrisation planaire et globale de l'objet, en s'appuyant sur les *octree textures* [DGPR02, BD02], introduits récemment. Un premier chapitre (chapitre 7) présentera une contribution technique : l'implémentation pour processeurs graphiques des *octree textures*, qui étaient jusqu'alors limités aux logiciels de rendu non interactif. Cette implémentation nous permet ensuite d'introduire notre modèle de simulation interactive d'un habillage dynamique sur une surface (chapitre 8). A titre d'exemple nous réalisons la simulation d'un liquide s'écoulant le long d'un objet. Le dernier chapitre (chapitre 9) présente notre second modèle d'habillage générique à base de motifs : les *textures composites*. Afin d'éviter les distorsions et pour créer des apparences variées, de haute résolution, à faible coût mémoire, les motifs sont appliqués localement sur la surface à la manière d'autocollants. Les motifs peuvent également être manipulés interactivement. Leurs contributions sont mélangées pour produire l'apparence finale de la surface.

La cinquième et dernière partie présente un modèle spécifiquement dédié à l'habillage de modèles géométriques d'arbres avec une apparence d'écorce réaliste (chapitre 10). Cette approche permet un gain de temps conséquent aux artistes (en prenant notamment en compte les problèmes d'embranchements et de croissance non–homogène), qui peuvent ainsi habiller automatiquement d'une apparence réaliste tous les arbres d'une forêt. Des paramètres de haut niveau permettent de contrôler aisément l'apparence finale des écorces.

[1] Les travaux présentés au chapitre 5 ont été effectués en collaboration avec le professeur Przemek Prusinkiewicz lors de mon séjour à l'université de Calgary (Canada), dans le cadre d'une bourse EURODOC de la région Rhônes–Alpes.

chapitre 2

Etat de l'art

L'*habillage*, au sens large, désigne l'opération qui permet d'enrichir la représentation d'un objet avec des détails de plus haute résolution. Ceci regroupe l'ensemble des méthodes permettant d'ajouter des détails à une forme plus simple : aspect de bois, d'écorce, mais également fourrure, cheveux, prairie, etc ... Dans cette thèse, nous nous concentrons sur les modèles d'habillage *de surfaces*, permettant de peindre des détails sur la surface de l'objet, sans introduire de géométrie supplémentaire.

Afin de mieux cerner le mécanisme qui permet aux modèles d'habillage de contrôler l'apparence des détails sur les surfaces, nous rappellerons, en section 1, les principaux algorithmes de rendu et modèles d'illumination locale. Nous présenterons ensuite, section 2, les modèles d'habillage utilisés en synthèse d'image sur des surfaces quelconques. Un certain nombre de surfaces, en particulier les terrains, présentent des caractéristiques qui permettent de développer des modèles d'habillage qui leur sont mieux adaptés. Ces modèles seront présentés en section 3. La section 4 introduira les mécanismes qui permettent d'empêcher des défauts visuels d'apparaître malgré la grande quantité de détails présents dans les images. Nous décrirons, en section 5, quelles méthodes et algorithmes permettent de créer des apparences de surface réalistes. Nous verrons ensuite, en section 6, quel est le lien entre les modèles d'habillage présentés ici et les représentations alternatives d'objets. Enfin, nous présenterons quelques éléments sur la conception de modèles d'habillage pour les applications interactives en section 7.

1 Rappel sur la création d'images de synthèse

Le but de cette section est de rappeler le principe des algorithmes de création d'images de synthèse. Nous verrons notamment où les modèles d'habillage interviennent dans le processus de génération des images.

1.1 Représentation des formes

Il existe une grande variété de formes et d'aspects dans la nature. On peut cependant distinguer deux grandes catégories d'objets : les objets ayant une surface tangible, pour lesquels il

est possible de discerner un intérieur et un extérieur, et les objets dont la forme n'est pas aussi précisément définie comme les fumées, la fourrure, les nuages.

Selon le type d'objet, différentes représentations sont utilisées. Les formes tangibles sont représentées à l'aide d'équations définissant leur surface, de maillages polygonaux, ou de fonctions décrivant leur intérieur [GAC+89, FvDFH90]. Les formes non tangibles sont par exemple décrites par des données volumiques explicites ou implicites [PH89, KK89] ou des systèmes de particules [Ree83, RB85]. Dans le cadre de cette thèse nous nous concentrons sur l'habillage de la surface de formes tangibles, même si certains de nos modèles sont compatibles avec d'autres représentations (voir section 6).

1.2 Eclairage

Lors de la formation d'une image, le calcul de la couleur et de l'intensité lumineuse s'effectue généralement en deux étapes. Une première étape consiste à estimer globalement les échanges lumineux dans la scène, afin de connaître la quantité de lumière (intensité et couleur) parvenant jusqu'à chaque point de chaque surface. Il s'agit de calculer *l'éclairage global* dans la scène. Si on veut prendre en compte les échanges lumineux globaux, une estimation peut être pré–calculée ou même déterminée lors du dessin.

Une seconde étape va calculer localement comment la surface réagit à la lumière reçue. Ce calcul est effectué par le *modèle d'illumination locale*. A cette fin, celui–ci utilise les propriétés locales du matériau (couleur, brillance, etc ...), qui peuvent varier le long de la surface : ce sont précisément les paramètres gérés par les modèles d'habillage .

1.3 Algorithmes de rendu

Le *rendu* désigne l'opération qui, à partir du modèle d'une scène et d'informations sur l'environnement lumineux, produit une image pour un point de vue donné. Il existe de nombreuses méthodes pour former des images de synthèse, mais la plupart sont basées sur deux approches : le *lancé de rayon* et le *rendu projectif*. Nous verrons tout d'abord comment les images sont affichées sur un écran, puis présenterons chacune de ces méthodes.

1.3.1 L'écran

Les images de synthèse sont destinées à être affichées sur un écran. Un écran classique est constitué d'une grille de petits éléments lumineux rouge, vert et bleu. Chaque élément de la grille est appelé un *pixel*. Le but d'un algorithme de rendu est donc de déterminer la couleur des pixels de manière à former l'image finale.

1.3.2 Lancer de rayon

L'approche du lancer de rayon [Whi80, GAC+89] consiste à simuler le parcours des rayons lumineux en remontant depuis l'oeil vers la scène et les lumières. Pour chaque pixel de l'écran un rayon est lancé, depuis l'oeil vers la scène. L'intersection avec les modèles géométriques des

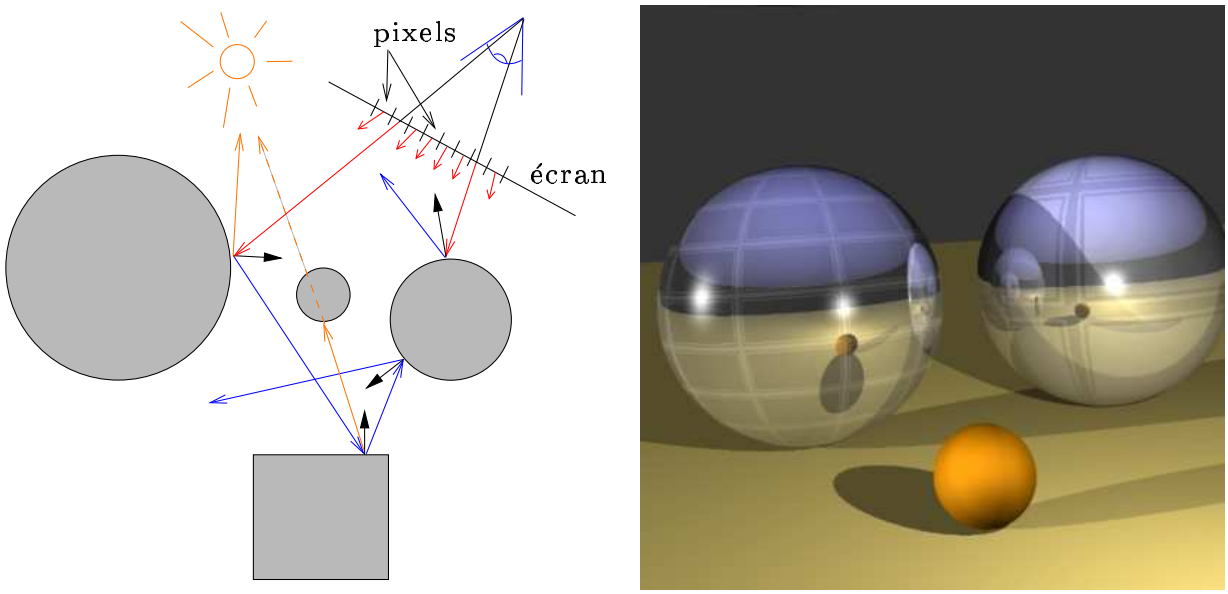


FIG. 2.1 – Lancer de rayon

A gauche : *L'image est formée en simulant le parcours de rayons lumineux depuis l'oeil vers la géométrie. Les intersections avec les objets sont recherchées. En chaque intersection un rayon est lancé vers la lumière et dans la direction de reflet. Le modèle d'illumination locale est alors utilisé pour calculer la couleur finale sur la surface des objets.* A droite : *Un modèle d'habillage forme les rainures des boules de pétanque en variant la couleur le long de la surface.*

objets (sphères, cylindres, triangles, ...) est alors calculée. Une fois l'intersection la plus proche trouvée, l'éclairage incident est estimé en lançant d'autres rayons en direction des lumières, afin de déterminer si elles influencent ou non le point de la surface. Des rayons sont également lancés dans les directions de reflet et de transmission par transparence, calculées grâce à la normale à la surface. Ceci permet de connaître les conditions d'éclairage au point d'intersection entre le rayon lancé depuis l'oeil et la scène.

Le modèle d'illumination locale est ensuite utilisé pour calculer la couleur du point de la surface, en fonction de l'éclairage provenant des différentes directions et des propriétés du matériau (voir section 1.5). Ces dernières sont déterminées par le modèle d'habillage . A partir de l'information géométrique (ici le point d'intersection), le modèle d'habillage va générer les propriétés du matériau utilisées par le modèle d'illumination locale. La couleur calculée déterminera la couleur du pixel correspondant au rayon initial.

L'avantage de cet algorithme est de permettre de prendre en compte toute l'optique géométrique, puisque l'on simule le parcours des rayons lumineux. Le principal défaut, en terme de réalisme, est d'ignorer l'éclairage indirect : les rayons sont lancés des surfaces directement vers les lumières. Or dans une scène réelle, les rayons émis par les lumières diffusent et rebondissent sur plusieurs surfaces avant d'arriver jusqu'à un point donné, créant ainsi des effets lumineux complexes. D'autres méthodes simulant l'éclairage global ont donc ensuite été développées, certaines basées sur le lancer de rayon [JC95], d'autres utilisant une approche basée sur les échanges

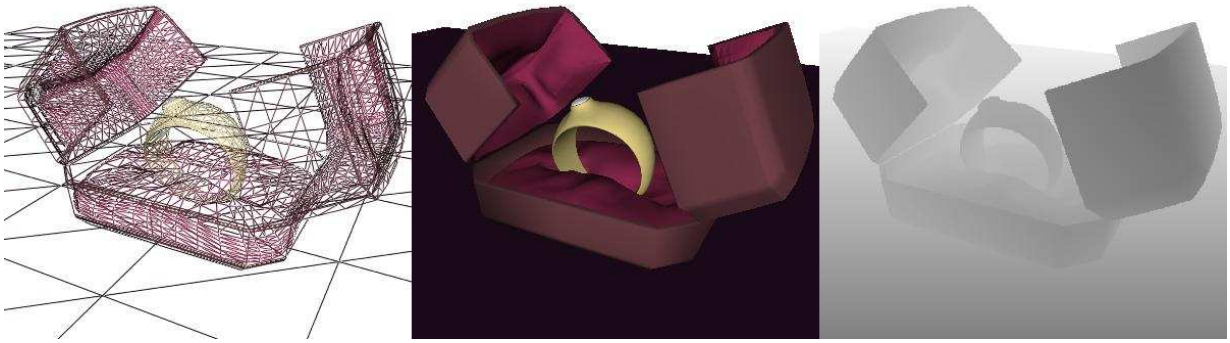


FIG. 2.2 – Algorithme de rasterisation

A gauche *: Les triangles projetés sur l'écran.* Au milieu *: Les triangles remplis par l'algorithme de rasterisation. Les couleurs et normales définies par sommet et interpolées lors du remplissage permettent d'effectuer le calcul d'illumination locale.* A droite *: le Z-buffer correspondant à ce rendu. La profondeur des pixels est également interpolée lors du remplissage des triangles.*

énergétiques entre surfaces (radiosité) [GTGB84, SAWG91]. L'algorithme de lancer de rayon est relativement coûteux car il met en oeuvre beaucoup de calculs d'intersection et n'exploite pas la cohérence spatiale. Grâce à de nombreuses optimisations et des processeurs dédiées, il commence à exister des versions expérimentales interactives [SWW+04].

1.4 Rendu projectif

Contrairement au lancer de rayon qui est une méthode "orientée écran", le rendu projectif est "orienté primitive" : la géométrie de chaque objet est projetée successivement sur l'écran. Cette approche implique moins de calculs géométriques et entraîne de meilleures performances. Cependant, l'éclairage des images formées est souvent moins réaliste. Comme cette approche met en oeuvre des calculs moins complexes, c'est celle qui a été retenue par les constructeurs de cartes graphiques et pour les applications interactives. Elle est également utilisée pour accélérer certains algorithmes de rendu réaliste.

Les formes sont représentées par des maillages triangulaires : un ensemble de sommets sur lesquels sont construits des triangles. Les triangles sont directement projetés sur le plan de l'écran. Ceux projetés en dehors du rectangle de l'écran sont éliminés (opération de *clipping*). L'approche est vectorielle : seuls les sommets des triangles sont projetés, entraînant relativement peu de calculs. Une fois les triangles sur l'écran, un algorithme de remplissage donne une couleur à chacun des pixels situés en leur intérieur. Cette conversion entre représentation vectorielle et pixels est appelée *rasterisation*.

Afin de dessiner les surfaces dans le bon ordre, deux algorithmes existent. Le premier, l'*algorithme du peintre*, affiche les triangles du plus loin au plus proche, les pixels de chaque nouveau triangle remplaçant les pixels déjà dessinés. Les triangles sont donc triés par ordre décroissant de profondeur avant dessin. Le second, l'*algorithme de Z–buffer* [Cat74], mémorise en chaque

pixel sa profondeur. Lorsqu'un nouveau triangle est dessiné, seuls les pixels plus proches que les pixels déjà en place sont acceptés et affichés. Ainsi, il n'est plus nécessaire de trier les triangles avant dessin, au prix de l'espace mémoire nécessaire au stockage de l'information de profondeur en chaque pixel.

Pour déterminer la profondeur d'un pixel, l'algorithme de rasterisation interpole les coordonnées des sommets dans le repère de la caméra, lors du remplissage des pixels à l'écran. En fait, cette notion s'étend à tous les attributs présents sur les sommets, tels que couleur, normale, coordonnées de texture, etc ... Les sommets peuvent être considérés comme des points de contrôle : les attributs des sommets sont *interpolés* pour former une fonction continue sur la surface des triangles. En général, l'interpolation utilisée est l'interpolation linéaire.

Lors du remplissage, un certain nombre d'informations géométriques sont donc connues pour les pixels candidats, appelés *fragments*. Afin de déterminer la couleur d'un fragment à l'aide du modèle d'illumination locale, le modèle d'habillage doit fournir les propriétés du matériau à partir des informations géométriques. Si le fragment est accepté par le *Z–buffer* sa couleur sera conservée pour le pixel correspondant sur l'écran. Le test avec le *Z–buffer* est souvent effectué avant de mobiliser le modèle d'habillage et le modèle d'illumination afin de ne pas gaspiller de temps à calculer l'apparence de fragments non visibles. D'autre part, le modèle d'habillage a également la possibilité d'éliminer le fragment : celui-ci sera ignoré et rien ne sera affiché (par exemple pour représenter des trous dans une surface).

Le rendu projectif et le *Z-buffer* sont notamment utilisés dans les cartes graphiques. L'algorithme permet de mieux tirer parti de la cohérence spatiale que le lancer de rayon, les primitives géométriques étant considérées les unes après les autres. Ceci permet de meilleures performances, mais également d'améliorer la qualité des images en réduisant *l'aliasing* (effets de fourmillement de couleurs ou d'escalier sur le contour des objets). Le principal défaut, en comparaison avec le lancer de rayon, est de ne pas pouvoir facilement tenir compte de l'environnement lumineux des objets. Cependant, il existe de nombreuses extensions pour simuler au moins en partie ces phénomènes (ombres [Wil78, Hei91, HHLH05], reflets [BN76, KG79, Gre86], etc...), d'autant plus que le récent gain en puissance et en souplesse de programmation des processeurs graphiques permet, de nos jours, des effets de rendu de plus en plus complexes [OHHM02].

1.5 Modèles d'illumination locale

Un modèle d'illumination locale prend en entrée l'éclairage incident et les propriétés du matériau en un point de la surface. Il modélise la réaction de la surface à la lumière et produit en sortie la lumière renvoyée par l'objet. Ici, le terme *lumière* regroupe à la fois la couleur et l'intensité lumineuse.

Il existe de nombreux modèles d'illumination locale. Nous allons présenter ici le modèle de Phong, qui est l'un des plus courants. Il s'agit d'un modèle empirique qui calcule la lumière renvoyée en fonction de la couleur du matériau, la normale à la surface, la direction de vue et la direction de la lumière incidente. Il étend le modèle diffus de Lambert, en ajoutant un reflet spéculaire qui représente les reflets brillants de la lumière sur la surface.
La lumière renvoyée est calculée comme :

$$I = I_{ambiant} + I_{diffus\ Lambert} + I_{speculaire\ Phong}$$

$$I = C_a k_a + C_d k_d (N.L)\, \mathbb{1}_{N.L>0} + k_s C_s (R.V)^n\, \mathbb{1}_{R.V>0}$$

où N est la normale à la surface, L la direction de la lumière incidente, R la direction du reflet spéculaire ($R = 2\ (L.N)\ N - L$) et V le vecteur de vue. C_a est la lumière ambiante, k_a la couleur ambiante du matériau. C_d est la lumière diffuse, k_d la couleur diffuse du matériau. C_s est la lumière spéculaire et k_s la couleur spéculaire du matériau. n contrôle l'exposant du reflet spéculaire (correspond à sa taille). Typiquement les plastiques ont un reflet spéculaire blanc ($k_s = 1$), alors que les métaux ont un reflet coloré ($k_s = k_d$). La figure 2.3 illustre l'effet obtenu avec le modèle de Phong pour différentes valeurs de paramètres.

Il existe bien sûr d'autres modèles d'illumination locale, comme le modèle de Torrance–Sparrow [TS66, TS67] issu de la physique et introduit en synthèse d'image par [CT82, Bli77]. D'autres modèles permettent de représenter des phénomènes tels que l'anisotropie [Kaj85, PF90] (la lumière possède une direction de reflet privilégiée). Encore plus générales, les fonctions de distribution de la réflectance bidirectionnelle [Rus97] (BRDF ou *bidirectionnal reflectance distribution function*) encodent la fonction d'illumination locale d'un matériau à partir de mesures effectuées sur des matériaux réels. Notons que, si le modèle de Phong est souvent préféré dans les applications interactives pour sa simplicité de mise en oeuvre, des travaux ont montré qu'il est possible d'utiliser, pour le rendu temps réel, des modèles d'illumination locale complexes [HS99,KM99,MAA01]. Le lecteur intéressé par les modèles d'illumination peut se référer à l'état de l'art de Schlick [Sch94].

1.6 Moduler l'illumination locale avec un modèle d'habillage

Ce qui crée véritablement l'apparence perçue, c'est l'interaction entre la lumière et le matériau de la surface de l'objet. Elle est donc complètement déterminée par le modèle d'habillage et le modèle d'illumination locale. **Modifier localement l'interaction entre la lumière et la surface, c'est modifier l'apparence perçue**.

Les premiers modèles d'habillage , en particulier le placage de texture présenté section 2.1, ont été très tôt utilisés pour faire varier la perception de l'aspect de la surface. Tout d'abord,

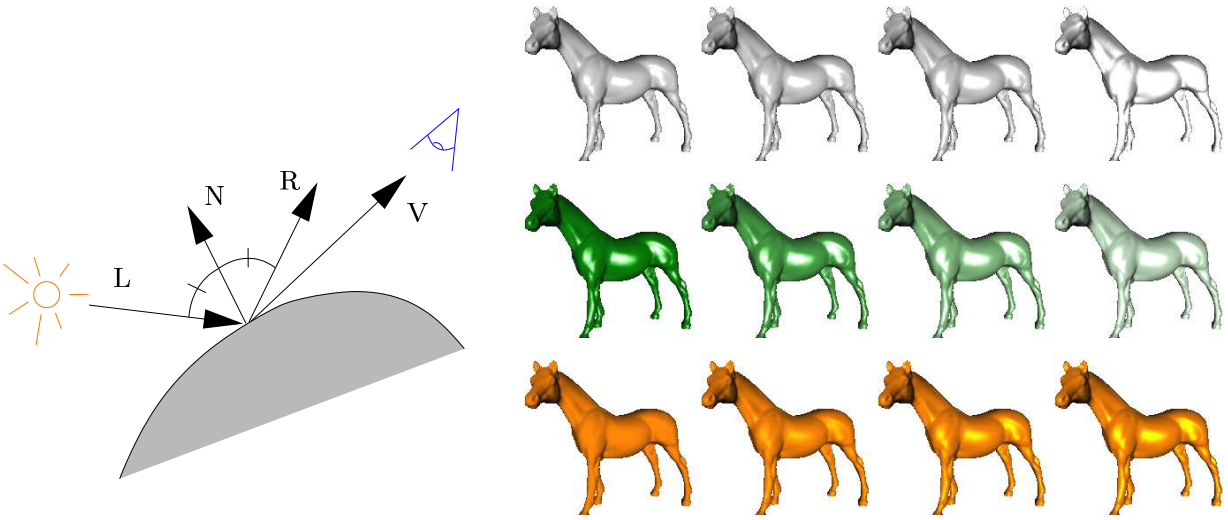


FIG. 2.3 – Modèle d'éclairage de Phong

A gauche : *normale à la surface N, direction de vue V, de lumière L et de reflet spéculaire R.* A droite : *Rendu d'une même géométrie avec différents paramètres de Phong. Les paramètres sont constants le long de la surface (aucun modèle d'habillage n'est utilisé). La première rangée illustre une variation de l'exposant spéculaire. La seconde rangée montre une variation de la couleur diffuse. La dernière rangée illustre une variation de la couleur de reflet spéculaire.*

seules des variations de couleur ont été introduites [Cat74]. Cependant, Blinn [Bli78] a très vite remarqué que l'on pouvait modifier la perception du relief de la surface en faisant localement varier la réaction de la surface à l'éclairage. A cette fin, il utilise un modèle d'habillage pour perturber la normale et créer l'illusion de motifs en relief. Cette technique, largement utilisée de nos jours, est connue sous le nom de *bump–mapping*. Kajiya [Kaj85] utilise lui aussi un modèle d'habillage pour contrôler, le long de la surface, les directions privilégiées de reflet d'un modèle d'illumination anisotropique. Les modèles d'habillage sont aussi utilisés pour décrire la contribution lumineuse de l'environnement sur la surface, que ce soit pour stocker l'intensité lumineuse pré–calculée en tout point (cartes d'éclairage ou *lightmaps*) [Hec90, ZIK98], l'auto–ombrage de la surface (*horizon map*) [Max88] ou pour simuler des reflets (voir section 2.2). En fait, tout paramètre du modèle d'illumination locale peut faire l'objet de variations le long de la surface, permettant ainsi de créer des détails, et peut donc bénéficier de l'utilisation d'un modèle d'habillage .

Un même modèle d'habillage peut donc être utilisé pour stocker différents types de paramètres. Cependant, certains seront mieux adaptés à une situation donnée : générer des motifs colorés très fins sur la surface ne pose pas les mêmes problèmes que stocker de l'éclairage diffus. De plus, tous les paramètres ne se manipulent pas de la même manière : des vecteurs unitaires représentant une normale ne se traitent pas comme des couleurs. Les modèles d'habillage doivent donc s'adapter à la fois aux types de surface auxquels ils sont appliqués, au type de paramètres et aux aspects de surface qu'ils sont destinés à représenter.

2 Modèles d'habillage pour les surfaces quelconques

Dans cette section, nous allons étudier les modèles d'habillage utilisés pour les surfaces quelconques (*i.e.* surface courbes, fermées, à trous, etc ...). Nous présenterons d'abord le modèle le plus répandu : le placage de texture (section 2.1). Ensuite, nous verrons des modèles d'habillage permettant d'associer aux surfaces des informations provenant de leur environnement, par exemple pour simuler des reflets (section 2.2). Ces structures plus complexes ont été étendues pour éviter certains défauts du placage de texture standard. Enfin, nous verrons des modèles d'habillage qui utilisent une représentation en volume du matériau constituant la surface (section 2.4). Nous allons identifier un certain nombre de limitations pour chacun de ces modèles. Nous verrons plus tard, dans la suite de l'état de l'art (section 3), que d'autres modèles d'habillage ont été développés pour des types de surface spécifiques, comme les terrains.

2.1 Placage de texture standard, par paramétrisation de surface

Le placage de texture est le modèle d'habillage le plus répandu. Il est utilisé dans les simulateurs, les jeux vidéos, mais également dans les logiciels de rendu réaliste. Il est aujourd'hui un composant de base de la synthèse d'image [HS93].

Il est donc important d'en détailler et d'en analyser les mécanismes et limitations : c'est précisément le but de cette section.

2.1.1 Principe

Le placage de texture a été introduit par Catmull [Cat74] en 1974. L'idée principale est d'associer une image à deux dimensions appelée *texture* à une surface géométrique. L'apparence de la texture est définie dans un espace de référence : l'*espace texture*. La correspondance entre l'objet et la texture est définie par la *paramétrisation* de la surface géométrique. On associe à chaque point de la surface à un point de l'espace texture (voir section 2.1.2), appelé *coordonnée de texture* (u,v). La paramétrisation permet de lire directement dans la texture la couleur correspondant à un point quelconque de la surface. La texture est ainsi appliquée à la surface de l'objet, comme illustré figure 2.4.

2.1.2 Paramétrisation planaire

Cette section discute des différentes approches utilisées pour attacher un espace texture 2D à une surface. On se place ici dans le contexte des surfaces représentées par des maillages triangulaires. Le problème est donc d'associer une coordonnée dans le plan (*coordonnée de texture*) à chacun des sommets du maillage. Cette coordonnée sera linéairement interpolée sur les triangles lors du rendu, ce qui permettra d'obtenir une paramétrisation continue de leur intérieur.

Au delà de quelques cas triviaux (voir section 2.1.2.1), la paramétrisation d'une surface est un problème difficile qui motive encore aujourd'hui de nombreuses recherches. Ce sujet trouve des applications dans d'autres domaines, tels que la simplification de maillages et la mise en correspondance de formes. Nous ne proposons pas ici une analyse détaillée, mais nous nous

FIG. 2.4 – Placage de texture

*La géométrie (*à gauche*) est associée à une texture (*au milieu*) via une paramétrisation qui définit une coordonnée dans la texture pour chaque sommet du maillage triangulaire. Il est dès lors possible de peindre la texture sur la géométrie (*à droite*). (Modèle et texture de Jean–Baptiste Rodriguez)*

attachons plutôt à décrire les approches les plus courantes et leurs liens avec le problème de l'habillage de surface. Le lecteur intéressé par les techniques de paramétrisation en général est invité à consulter l'état de l'art de Floater et Hormann sur le sujet [FH05].

2.1.2.1 Cas triviaux La paramétrisation planaire est facile sur quelques types de surfaces. Par exemple, les surfaces planes (sols, murs), cylindres et cônes peuvent être paramétrés sans distorsions. En fait, toute surface développable peut être mise à plat sans distorsion.

Il existe un autre cas largement répandu : les terrains, qui sont des surfaces quasi–planaires. La cartographie tire avantage de cette propriété en les représentant vus de dessus. Les coordonnées x, y fournissent alors une paramétrisation efficace du terrain. On considère généralement que la distorsion due à la pente est négligeable. Les terrains sont d'ailleurs souvent représentés par des cartes de hauteur : un maillage triangulaire régulier est construit à la surface du plan. Les sommets du maillage prennent leur troisième coordonnée en échantillonnant la carte de hauteur. Ceci est illustré figure 2.5. Il s'agit certes d'un cas particulier, mais correspondant à une application très largement répandue (simulateurs, jeux).

2.1.2.2 Paramétrisation de surfaces quelconques Afin d'associer une image 2D à une surface, il faut déplier la surface sur un plan, qui représente l'espace texture. Dans le cas général ceci ne peut se faire sans introduire de distorsions ou de replis. Un repli (*foldover*) désigne le cas ou deux parties distinctes de la surface sont associées à une même zone du plan et donc de l'espace texture. En conséquence, le même aspect se trouve répété sur les deux parties de la surface. Même si ceci est désirable dans certains cas très particuliers, il est souvent préférable d'obtenir une paramétrisation injective.

Il est connu que l'on peut éviter les replis si une surface est *homéomorphe* à un disque : il est possible de la déformer pour la transformer en un disque. Pour obtenir cette propriété, il faut souvent introduire une ou plusieurs coupures dans le maillage. Par exemple, une sphère ne possède pas de bordure et ne peut donc pas être mise à plat sur un plan sans introduire de

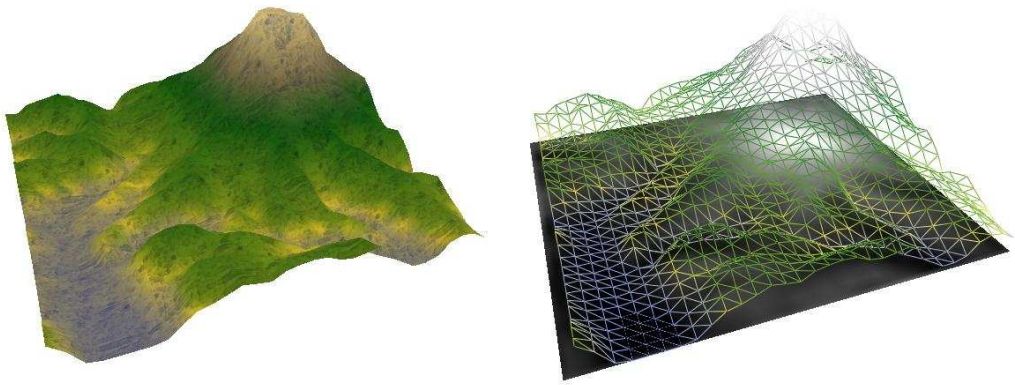


FIG. 2.5 – Les terrains sont représentés par des cartes de hauteur.
Le terrain est représenté par un maillage triangulaire régulier. La carte de hauteur est utilisée pour lire la coordonnée z des sommets. Les coordonnées des sommets dans le plan définissent une paramétrisation de la surface dont la distorsion est généralement ignorée, car faible.

replis. L'introduction d'une coupure rend la surface homéomorphe à un disque et une mise à plat sans replis devient possible. Il reste néanmoins à trouver une transformation minimisant les distorsions.

Deux types de distorsions peuvent être observées : les distorsions angulaires et les distorsions métriques. Les distorsions angulaires apparaissent lorsque les angles ne sont pas conservés par la paramétrisation : les angles entre les arêtes des triangles sur la surface ne seront plus les mêmes une fois le maillage mis à plat. Les distorsions métriques apparaissent lorsque les distances ne sont pas conservées : les longueurs des arêtes des triangles ne seront plus les mêmes une fois le maillage mis à plat. Un résultat connu [AL60] démontre qu'une paramétrisation planaire conservant les distances n'existe que si la surface est développable (e.g. cylindre, cône), ce qui n'est pas le cas de la plupart des surfaces manipulées en synthèse d'images.

Ces distorsions sont souvent indésirables dans le cas du placage de texture. La texture possède une résolution uniforme : la quantité d'information (résolution par unité d'aire) est constante. En conséquence les distorsions d'aire ou d'angle produisent des défauts visuels à la surface de l'objet : certaines parties de la surface possèdent plus de détails que d'autres.

Aplanir la surface de manière satisfaisante implique donc de pouvoir introduire des coupures et de minimiser les distorsions.

2.1.2.3 Mise à plat de surface à bordure unique De nombreux travaux [Sam86, ML88, Flo97, LM98, HG, HATK00, DMA02] se sont intéressés à minimiser les distorsions lorsque le maillage est homéomorphe à un disque. Si ce n'est pas le cas, le maillage doit être prédécoupé, par exemple par l'utilisateur à l'aide d'un outil d'édition. Sheffer et Hart [SH02] ont proposé une méthode permettant d'automatiser la coupure du maillage. La paramétrisation est ensuite

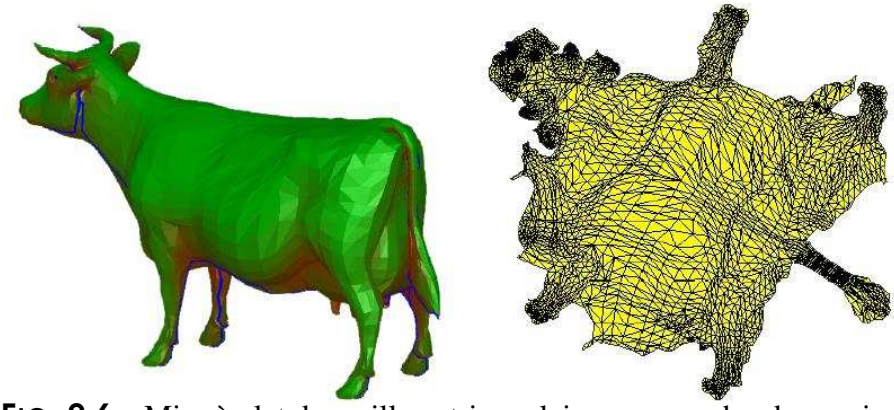


FIG. 2.6 – Mise à plat de maillage triangulaire avec une bordure unique.

Après avoir été découpé (coupure en bleu sur le modèle à gauche) le maillage est mis à plat en minimisant les distorsions [SH02].

créée avec l'une des méthodes précédentes. La figure 2.6 présente un maillage coupé puis paramétré avec cette approche. Leur algorithme dissimule les coupures dans des zones peu visibles du maillage triangulaire. Nous verrons en effet section 2.1.6 que les coupures produisent des défauts visuels. Notons en outre que l'introduction des coupures nécessite de modifier la géométrie puisque les sommets le long de la bordure doivent être dédoublés.

L'approche commune à ces travaux est d'exprimer la distorsion sous la forme d'une énergie à minimiser. Diverses méthodes de minimisation sont utilisées pour chercher une paramétrisation optimale (en terme de distorsions résiduelles). La bordure du maillage est parfois contrainte sur un rectangle ou un polygone convexe [Flo97, LM98], mais fait généralement également partie du processus d'optimisation, afin de réduire plus encore les distorsions. Par exemple, dans la figure 2.6 le contour du maillage dans la paramétrisation est quelconque. Toutes ces méthodes produisent une paramétrisation en un seul morceau, dont l'intérieur est continu.

2.1.2.4 Atlas de texture La notion d'*atlas de texture* a été introduite par Maillot et al. [MYV93]. Le principe est de découper le maillage en différentes parties distinctes, à la manière des patrons utilisés pour la confection de vêtements, puis de paramétrer chaque partie indépendamment. Les méthodes présentées ci-avant peuvent être utilisées à cet effet. Les différentes paramétrisations sont ensuite regroupées, en les disposant côte à côte dans le plan.

Cette approche permet de réduire les distorsions en relâchant les contraintes imposées par une paramétrisation globale [BVI91]. Les distorsions sont "échangées" contre l'introduction de discontinuités dans la paramétrisation (voir figure 2.7). Cette méthode est notamment utilisée par les artistes qui souvent effectuent cette opération manuellement : après avoir découpé le modèle en plusieurs parties, ils le déplient sur le plan à l'aide de logiciels dédiés à cette tâche.

Des travaux plus récents ont permis d'améliorer et d'automatiser l'algorithme [SSGH01, CMRS98, LPRM02]. Notons que Carr et Hart [CH02] proposent de pousser la notion d'atlas à l'extrême en paramétrant chaque triangle indépendamment.

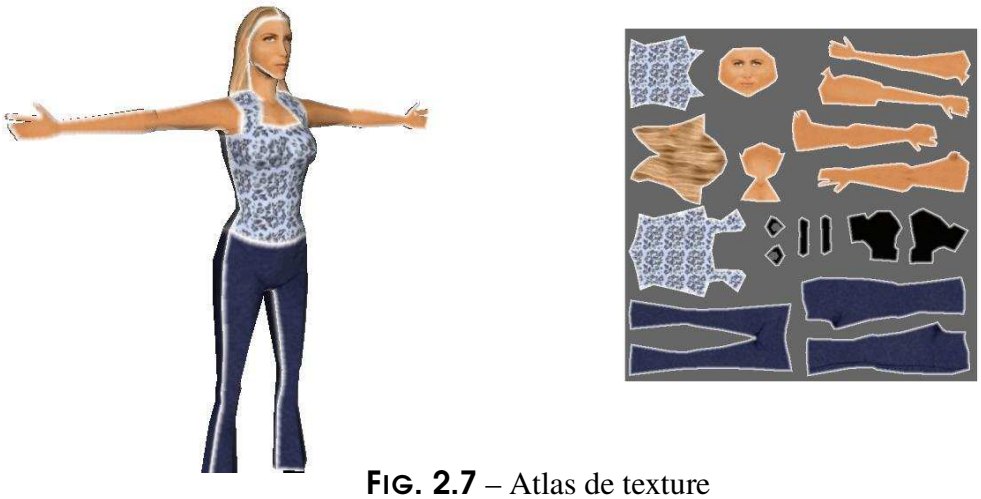


FIG. 2.7 – Atlas de texture

Différentes parties du modèle sont paramétrées séparément. Les sous–parties sont ensuite regroupées dans une même texture.

Un des points difficiles, avec les atlas, concerne l'empaquetage des différentes parties paramétrées dans un rectangle de taille minimale (problème équivalent à la minimisation des chutes de tissu dans la confection de vêtements). En effet, l'image utilisée pour le placage de texture est rectangulaire. Ce problème est NP–complet dans le cas général. Il est similaire à l'ordonnancement de tâches et connu sous le nom de *strip–packing*.

Nous verrons également en section 2.1.6 que les discontinuités introduites par les atlas de texture entraînent des problèmes importants du point de vue du placage de texture.

2.1.2.5 Paramétrisations contraintes Les travaux présentés précédemment s'attachent à paramétrer le maillage triangulaire en tenant compte uniquement de ses propriétés géométriques.

Néanmoins, il est parfois souhaitable d'effectuer la paramétrisation en fonction des données contenues dans la texture. Exemple classique : l'utilisateur dispose de la photographie d'un visage et souhaite la mettre en correspondance avec un maillage existant. Il faut alors faire correspondre les éléments géométriques (*e.g.* les yeux, la bouche, ...) avec le contenu de la texture (la photographie ou dessin). Des techniques de paramétrisation sous contraintes ont été développées en ce sens [GGW+98, Lév01, DMA02, KSG03]. Elles permettent à l'utilisateur de guider le système, en introduisant des contraintes de positionnement. Ces contraintes ne peuvent être respectées qu'au prix de l'introduction de distorsions supplémentaires.

2.1.3 Stockage des données de texture

La texture appliquée à la surface par le placage de texture est représentée sous la forme d'une image rectangulaire à deux dimensions stockée en mémoire. Les éléments de la texture sont appelés *texels* (pour *texture elements*). Lorsque la texture stocke des couleurs, chaque texel est un triplet (rouge,vert,bleu). Une quatrième valeur, nommée *alpha*, est souvent ajoutée ; elle permet de stocker un attribut supplémentaire, généralement un coefficient d'opacité. Il permet par exemple de mélanger plusieurs couches de textures appliquées sur une même surface. En outre, les textures servent aussi à stocker d'autres attributs : perturbations de normales (*bump maps*) [Bli78], repères locaux [Kaj85], éclairage (*lightmaps*), etc ... Par exemple, dans le cas du modèle d'illumination locale de Phong (voir section 1.5), il est possible de faire varier trois couleurs le long de la surface (ambiante, diffuse et spéculaire) ainsi que le coefficient spéculaire.

2.1.4 Accès aux données de texture

Durant le dessin à l'écran, l'algorithme de rendu connaît les coordonnées de texture des points de la surface. L'algorithme de rasterisation (voir section 1.3) interpole les coordonnées de texture définies en chaque sommet par la paramétrisation. L'interpolation n'est pas linéaire car il faut tenir compte de la projection perspective. Sans cela, des effets de glissement de la texture à la surface de l'objet apparaissent [HM91]. Les algorithmes basés sur du lancer de rayon [Whi80] calculent le point d'intersection entre le rayon lumineux et la surface de l'objet. Il est alors facile de retrouver la coordonnée de texture à partir du point sur la surface (par exemple en interpolant les coordonnées depuis les sommets du triangle intersecté).

Les texels sont considérés au centre des cases de la grille formée par l'image dans l'espace texture. L'accès le plus simple aux données consiste à renvoyer la couleur du texel le plus proche du point d'échantillonnage. Lorsque l'algorithme de dessin souhaite connaître la couleur au point (u,v), l'indice (i,j) du texel dans le tableau stocké en mémoire est calculé par[1] $(i,j) = (\lfloor u \rfloor, \lfloor v \rfloor)$.

Malheureusement, ceci produit un champ de couleur discontinu et, lorsque la surface est proche de l'écran, la grille due à l'utilisation du texel le plus proche devient visible. On parle alors de *pixelisation*. La section suivante décrit une méthode permettant de supprimer cet effet indésirable. Il existe un autre défaut visuel majeur dans la situation inverse (point de vue éloigné) : l'aliasing . Il s'agit d'un "grouillement" de couleurs qui se manifeste sur les faces obliques ou lorsque les objets s'éloignent (voir figure 2.25). Ce problème étant général à tous les modèles d'habillage , nous reportons son étude à la section 4.

2.1.5 Interpolation

Afin de supprimer les effets de la pixelisation (voir ci–avant), on considère les texels comme les points de contrôle d'une fonction continue. L'idée est de reconstruire localement une fonction

[1] L'espace texture est souvent normalisé de manière à ce que les coordonnées de texture soient comprises dans l'intervalle $[0,1[\times[0,1[$. Dans ce cas, pour une image de taille (l,h), $(i,j) = (\lfloor ul \rfloor, \lfloor vh \rfloor)$

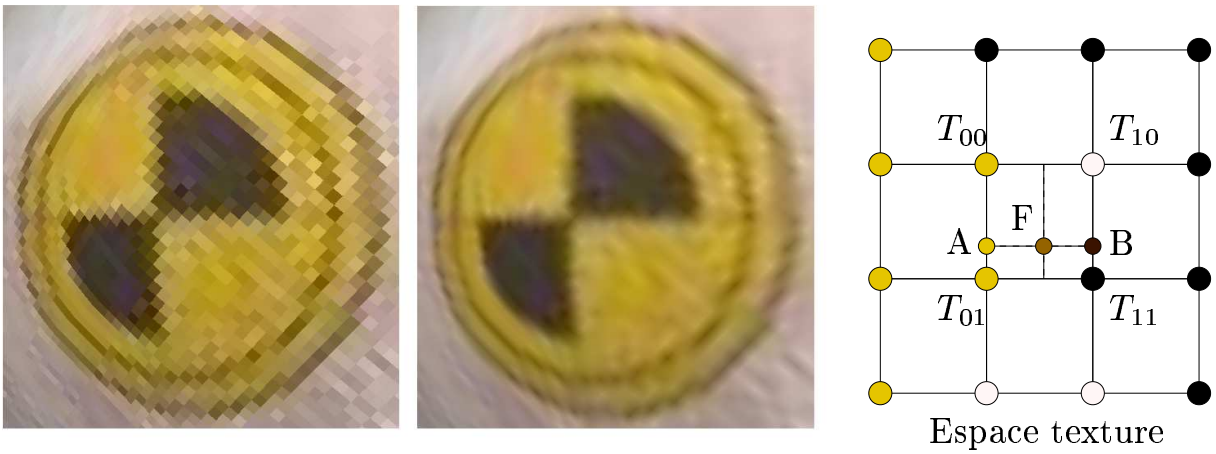


FIG. 2.8 – Exemple d'interpolation linéaire

A gauche *: Texture appliquée sans interpolation.* Au milieu *: Texture avec interpolation bi–linéaire.* A droite *: Le centre du pixel se projette en F dans la texture. La couleur du point A est linéairement interpolée entre les texels T_{00} et T_{01}. La couleur du point B est linéairement interpolée entre les texels T_{10} et T_{11}. La couleur au point F (point d'échantillonnage) est linéairement interpolée entre A et B. Sans interpolation, la couleur de F aurait été la couleur du texel le plus proche : T_{11}.*

lisse lorsqu'un point est échantillonné à une position arbitraire dans la texture. Ceci permettra de supprimer la grille et les défauts visuels qui lui sont associés.

La plupart des systèmes de rendu implémentent à cette fin un schéma d'interpolation, généralement l'interpolation *bi–linéaire*. L'idée est de considérer non pas un seul texel, mais les quatre texels encadrant le point d'échantillonnage. La couleur est déduite de la position du point dans le carré formé par ces quatre texels. Ce principe est illustré figure 2.8. La figure montre également l'amélioration de qualité obtenue en utilisant l'interpolation bi-linéaire. D'autres schémas d'interpolation sont bien sûr possibles, comme l'interpolation *bi–cubique*, mais elles mettent en jeu plus de texels et sont donc plus lentes. Nous présentons section 2.1.7.3 une méthode d'interpolation récente [Sen04] permettant de simuler des arêtes vives dans les textures.

2.1.6 Limitations du placage de texture

La plupart des limitations rencontrées avec le placage de texture proviennent à la fois de la difficulté à paramétrer les surfaces et de la structure de donnée très simple employée pour représenter le contenu (image rectangulaire de résolution uniforme).

Comme nous l'avons vu en section 2.1.2, les méthodes de paramétrisation doivent en effet découper les surfaces en différentes sous parties afin de réduire les distorsions (atlas de texture). Malheureusement, distorsions et discontinuités introduisent toutes deux différents types de problèmes que nous décrivons ci–après.

2.1.6.1 Gaspillage de mémoire

Bordures Un premier problème apparaît sur le contour des paramétrisations. Nous avons vu que le contour du maillage dans l'espace texture est souvent quelconque. Or la texture est stockée sous la forme d'une image rectangle. Ceci implique que l'ensemble des texels entre la bordure de la paramétrisation et le rectangle de la texture sont gaspillés (voir figure 2.6). Lorsque la texture est de haute résolution, le gaspillage peut devenir conséquent.

Distorsions Quelle que soit la qualité de l'algorithme de paramétrisation, il existe une distorsion résiduelle. Ceci est d'autant plus vrai que l'algorithme tente de ne pas découper la surface ou que des contraintes de positionnement sont imposées.

En conséquence, la densité de la texture à la surface n'est pas constante (par exemple, 1 cm^2 de surface correspond à une aire variable dans la texture). Certaines zones affichent moins d'informations que d'autres. Or, si l'utilisateur souhaite une résolution minimale garantie, par exemple lorsqu'un matériau homogène est appliqué à la surface, il devient obligatoire d'augmenter *globalement* la résolution de la texture afin de compenser la perte due à la distorsion. Ceci afin d'atteindre la résolution souhaitée dans les zones de distorsion maximale.

Prenons l'exemple concret d'un visage. Les fins détails de la peau peuvent être efficacement capturés par une texture. La peau présente souvent un grain homogène ; on souhaite donc appliquer la texture sur la géométrie du visage sans distorsions. Or, une paramétrisation typique du visage (en un seul morceau) va associer moins de résolution à la texture sur le nez qu'au reste du visage, comme le montre la figure 2.9. En conséquence, afin que l'aspect de la peau sur le nez reste cohérent, il va falloir augmenter *globalement* la résolution de la texture. Dans la texture, la zone correspondant au nez sera dessinée à la résolution maximale, alors que dans les autres parties (front, joues, etc ...) la texture sera dessinée avec moins de détails pour paraître homogène. Autrement dit, de la mémoire est gaspillée dans les zones correspondant aux autres parties du visage puisque la texture encode moins d'informations que sa résolution ne le lui permet. Encoder plus d'informations est possible mais non souhaitable car l'inhomogénéité deviendrait visible.

Notons que ce problème apparaît également lorsque l'utilisateur souhaite utiliser une texture non homogène (résolution variable). Reprenons l'exemple du visage : les yeux, et en particulier l'iris, contiennent souvent plus de détails que la peau. Il est donc prévisible que l'artiste souhaite accorder plus de résolution dans ces zones. Si la paramétrisation ne tient pas compte de ceci, il

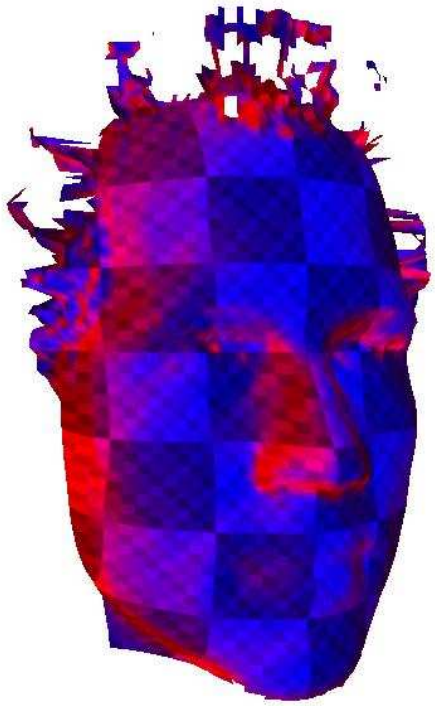


FIG. 2.9 – Paramétrisation d'un visage.
Une texture de damier est appliquée sur la géométrie. Les zones de distorsions sont en rouge (gris clair). Elles ont moins de résolution allouée dans la texture ce qui résulte en une apparence non–homogène.

devient donc encore une fois nécessaire d'augmenter globalement la résolution de la texture afin d'avoir suffisamment de résolution dans les zones les plus détaillées.

On peut résumer ainsi ce problème : à cause du stockage homogène d'une texture, la résolution doit *globalement* correspondre à la résolution permettant d'encoder les détails les plus fins sur la surface. Ceci peut impliquer un gaspillage de mémoire conséquent. Notons que des méthodes ont été développées afin d'optimiser la paramétrisation après ou pendant la création de la texture [SWB98, SGSH02, CH04]. La paramétrisation est déformée afin d'allouer plus de résolution (une plus grande zone de la texture) aux parties du maillage possédant les détails les plus fins, repoussant ainsi la limite à laquelle ce problème se produit.

Discontinuités Comme nous l'avons vu précédemment, les méthodes de paramétrisation minimisent les distorsions en découpant la surface, parfois en plusieurs parties (atlas de texture). Une fois la surface découpée, les différentes sous parties doivent être regroupées dans le rectangle correspondant à la texture stockée en mémoire, de manière à minimiser l'espace vide. La première mauvaise nouvelle est qu'il n'existe pas de regroupement éliminant totalement les espaces vides dans le cas général. La seconde mauvaise nouvelle est que ce type d'optimisation est connu pour être un problème NP–complet (problème de *strip–packing*). Il est donc résolu par des heuristiques [Rem96], qui permettent néanmoins de trouver de bonnes solutions.

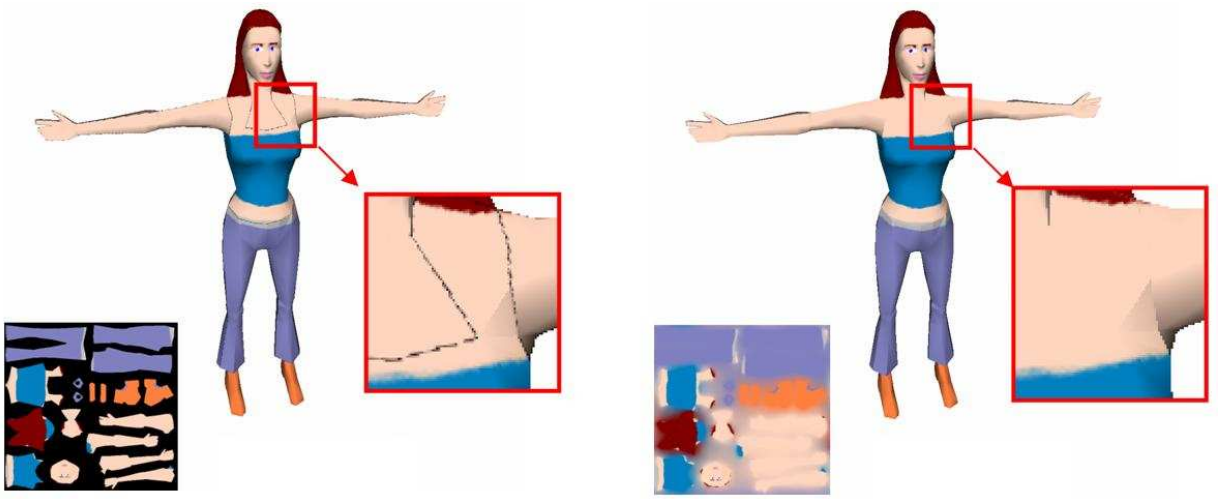


FIG. 2.10 – Interpolation linéaire erronée due aux discontinuités.
A gauche : *L'interpolation linéaire utilise des texels situés en dehors des différentes parties de l'atlas. Visuellement la couleur de fond s'étale dans la texture.* A droite : *L'artiste peint en dehors des différentes parties de l'atlas pour corriger l'interpolation.*

Ces espaces vides gaspillent eux aussi de la mémoire. Ce gaspillage peut devenir important, surtout lorsque la résolution globale de la texture augmente. De plus, comme nous allons le voir section 4.3.1.2, il est souvent souhaitable de laisser un espace entre les différentes sous parties de l'atlas pour éviter des défauts visuels (débordements de couleurs).

2.1.6.2 Interpolation et discontinuités Lors de l'interpolation linéaire, l'algorithme suppose que les texels voisins à l'écran sont également voisins dans la texture. En conséquence, si une discontinuité est introduite par la paramétrisation (atlas ou bordure), un à trois des texels voisins peuvent être dans la zone non définie de la texture. La couleur des espaces vides va alors déborder dans la texture lors de l'affichage. Ce défaut est illustré figure 2.10.

Une solution partielle consiste à colorer une bande d'au moins un pixel d'épaisseur dans la texture autour des bordures. Ceci permet d'ajouter le texel manquant lors des interpolations linéaires. Cependant, nous verrons en section 4.3.1.2 que cette bordure doit être élargie pour les algorithmes de filtrage.

2.1.7 Extensions du placage de texture

Plusieurs travaux apportent des extensions à l'algorithme standard de placage de texture, notamment en ce qui concerne l'encodage des données. Ils ont été principalement menés dans le contexte des applications interactives, pour lesquelles les limitations d'espace mémoire sont plus fortes. Ils portent sur la réduction de la taille occupée en mémoire par les textures, via des algorithmes de compression du contenu, mais également en adaptant localement la résolution de stockage aux données. D'autres travaux proposent des schémas d'interpolation et de stockage permettant de représenter plus fidèlement certains aspects de surface.

2.1.7.1 Compression de texture Les méthodes de compression pour les textures doivent permettre un accès aléatoire rapide. Ceci exclut les algorithmes de compression séquentiels tels que RLE (*Run Length Encoding*) ou *LZW*. La compression est donc généralement effectuée sur les texels ou sur de petits blocs de texels. Beers et al. [BAC96] proposent une méthode basée sur la quantisation des données. De petits blocs de pixels (typiquement 2×2) sont remplacés par un index dans un dictionnaire. Ce dictionnaire contient des blocs représentant au mieux les blocs d'origine. Il s'agit d'une compression avec perte. Des méthodes de compression destructives sont également disponibles dans les cartes accélératrices (algorithme ST3C). Même si des taux de compression non négligeables peuvent être atteints, ils ne permettent pas toujours de réduire suffisamment la taille des textures tout en préservant une qualité acceptable. Notons que les algorithmes de compression de texture prennent un nouvel intérêt avec l'émergence des plateformes graphiques mobiles [SAM] sur lesquelles la quantité de mémoire disponible est très limitée (assistant personnels, téléphones mobiles).

2.1.7.2 Textures Adaptatives Les textures adaptatives permettent de varier localement la résolution de stockage en fonction de l'utilisation de la texture. Heckbert [Hec90] propose un modèle de textures hiérarchiques pour stocker des informations lumineuses le long des surfaces. Les données sont stockées dans une grille hiérarchique 2D (*quadtree*). Fernando et al. [FFBG01] utilisent également une approche hiérarchique pour stocker, dans une texture, une carte d'ombres (*shadow map*) représentant la scène vue depuis une lumière.

Kraus et Ertl [KE02] proposent une structure de donnée différente pour permettre de limiter le stockage d'espace vide dans une texture 2D ou 3D (méthode également décrite par McCool [Coo02]). Leur approche présente l'avantage d'être bien adaptée aux cartes accélératrices actuelles et est donc très efficace. L'idée est de recouvrir l'espace texture d'une grille régulière, appelée *grille d'indirection*. Chaque case de la grille peut être vide, ou contenir un pointeur vers une imagette stockée dans une seconde texture, appelée *texture de référence* (voir figure 2.11). Celle–ci stocke de manière compacte de petites images carrées. Lorsque l'on souhaite accéder aux données, il faut tout d'abord déterminer dans quelle case de la grille d'indirection le point d'échantillonnage se trouve. Si la case est vide, la couleur choisie pour le fond est retournée. Si la case contient un pointeur, la position du point dans la texture de référence est calculée à partir du pointeur, et un accès dans la texture permet de retrouver la bonne couleur.

Le problème de cette méthode est qu'elle complique l'interpolation linéaire puisque les indirections modifient les voisinages. Les indirections ont en fait le même effet que des discontinuités

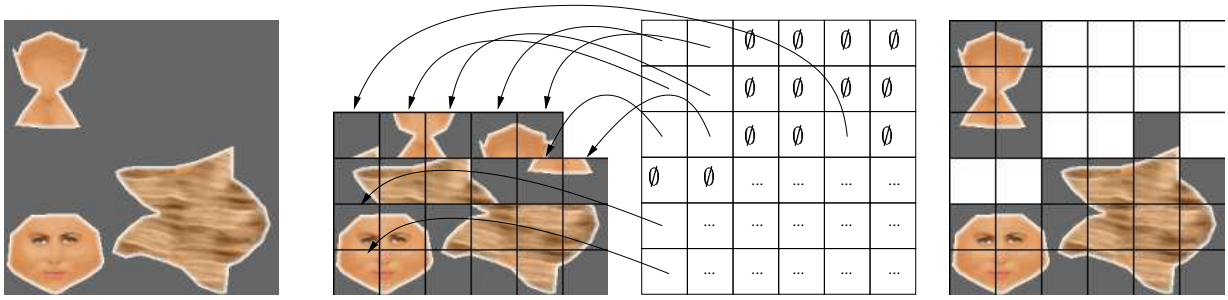


FIG. 2.11 – Compression des vides.

La texture initiale est découpée par une grille. La texture de référence contient uniquement les cases non vides. La grille d'indirection permet de recomposer la texture d'origine. Ses cases sont soit vides, soit contiennent un pointeur (pour des raisons de lisibilité les pointeurs des cases marquées de trois points ne sont pas représentés).

dans la paramétrisation, avec la différence notable qu'ici la géométrie n'est jamais modifiée. De plus, le filtrage de la texture devient également plus difficile (voir section 4). En conséquence, des défauts visuels apparaissent lors de l'affichage de l'objet. Ceci nous permet de constater que des structures de données à peine plus complexes qu'un stockage dans un tableau posent déjà de réels problèmes.

Notons que même simple, il s'agit ici d'un *algorithme* d'accès à une structure de donnée représentant la texture. De telles méthodes ne sont réalisables – et intéressantes – que depuis l'apparition de processeurs graphiques programmables. Auparavant une telle approche n'aurait pas été exploitable car impossible à mettre en oeuvre dans une application interactive. C'est ce potentiel que nous proposons d'exploiter au travers des modèles d'habillage exposés dans cette thèse. Cependant, trouver les algorithmes et structures de données permettant d'efficacement stocker une texture *tout en conservant un rendu de qualité* est un vrai défi dans le contexte des applications interactives.

2.1.7.3 Cartes de silhouettes Les cartes de silhouettes ont été introduites par Sen *et al.* [SCH03, Sen04]. La méthode s'attaque au problème de la représentation d'arêtes vives contenues dans les textures. Celles–ci sont en effet rendues floues par l'interpolation linéaire habituellement utilisée.

L'idée est donc de stocker dans la texture à la fois des informations de couleur et des informations de contour. Comme nous l'avons vu précédemment, les texels sont généralement considérés au centre des cases de la grille régulière ayant la résolution de la texture. Sen [Sen04] propose de stocker pour chaque texel une position arbitraire à l'intérieur de cette case. Les lignes reliant les texels voisins sont également annotées comme étant des arêtes vives ou non. Lorsque l'on souhaite accéder aux données, il faut tout d'abord déterminer dans quel quadrant (défini par les arêtes reliant les texels) de la case du texel le point d'échantillonnage se trouve (voir figure 2.12). Ensuite, selon le type des arêtes (vives ou non), différents schémas d'interpolation sont utilisés. Ceci permet d'améliorer grandement la qualité de texture contenant des motifs vectoriels, comme

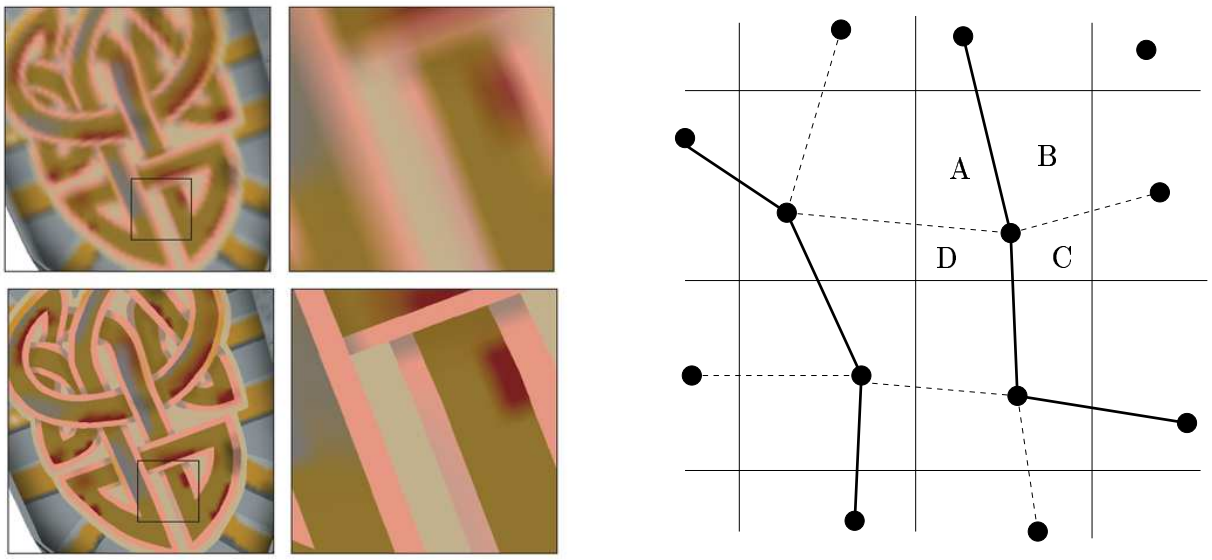


FIG. 2.12 – Carte de silhouettes

Les cartes de silhouettes [Sen04] permettent d'encoder des arêtes vives dans les textures. En haut à gauche *: interpolation standard.* En bas à gauche : *carte de silhouettes.* A droite : *Les texels sont à des positions arbitraires dans la grille formée par la texture. Les lignes entre texels sont déclarées comme arêtes vives ou non. Selon le quadrant dans lequel le point d'échantillonnage se situe (A,B,C ou D) et le type d'arête, différents schémas d'interpolation sont utilisés.*

le montre la figure 2.12, au prix d'un accès texture légèrement plus coûteux et d'une petite augmentation de la taille des textures.

2.1.7.4 Textures dépendant du point de vue Les textures dépendant du point de vue permettent d'encoder à la fois la variation des propriétés du matériau le long de la surface mais également leur variation selon l'angle de vue et (éventuellement) la position de la lumière (par exemple pour modéliser du métal brossé, des tissus, des matériaux granuleux, etc ...). La texture devient alors une table à plusieurs dimensions : par exemple les 2 dimensions usuelles, plus 2 dimensions pour la direction de vue. Lorsque la lumière est prise en compte, le modèle d'illumination locale devient inutile : l'interaction lumineuse est directement encodée dans la texture. Les données sont soit créées à partir d'une série de photographies dans lesquelles la direction lumineuse a été soigneusement contrôlée, soit générées par des modèles de rendu plus complexes. Les travaux de Dana et al. [DvGNK99] sur les BTF (*bidirectionnal texture functions*) ont notamment portés sur la capture de fonctions représentant un matériau sous tous points de vue et pour toutes les directions de lumière (fonction 6D).

Dans leurs travaux, Malzbender et al. [MGW01] encodent la fonction représentant la texture sous tous points de vue sous la forme de coefficients de polynômes stockés dans la texture. La fonction est localement reconstruite lors de l'accès à la texture. Dischler [Dis98] propose de recouvrir la surface d'une texture contenant, pour chaque texel, une table encodant la variation

FIG. 2.13 – Texture dépendant du point de vue

La technique de view dependent displacement mapping *[WWT+03] encode dans une texture à 5 dimensions la distance entre la surface de l'objet représentée par les polygones et la surface réelle. Ceci permet de sculpter la surface polygonale, sans avoir recours à de la fine géométrie. Les silhouettes et l'auto–ombrage sont correctement représentées.*

de la couleur selon la direction du rayon incident. Cette texture, à 4 dimensions, est pré–calculée. Lors du dessin, le rayon incident est transformé dans le repère local de la surface et la table 2D du texel est consultée pour trouver la bonne couleur. Des effets de relief et de parallaxe très convaincants apparaissent sur la surface, mais la silhouette de l'objet reste inchangée. Ce type d'approche permet ainsi de représenter la fine géométrie d'une surface sans avoir recours à des polygones.

Les récents travaux de Wang et al. [WWT+03] permettent de véritablement ajouter un effet de relief sur les surfaces. En particulier, les silhouettes et l'auto–ombrage sont capturés correctement. Pour chaque position dans l'espace texture, chaque courbure et chaque direction de vue (5 dimensions), la texture encode la distance entre le point sur la surface de référence (la géométrie habillée par la texture) et le véritable point d'intersection. Ceci permet de donner l'illusion que la surface est sculptée par la texture. Le résultat de cette approche est montré figure 2.13. La technique est appelée *view dependent displacement mapping* ou *VDM*. Notons que l'idée d'effectuer localement, depuis la surface polygonale de la géométrie, un lancer de rayon afin de simuler un effet de relief a été introduite par Patterson et al. [PHL91]. Cette approche, bien que coûteuse, devient réalisable dans les applications interactives [YJ04]. Les *VDM* sont similaires à cette approche, mais beaucoup plus rapides grâce aux pré–calcul des intersections.

Remarque : Les méthodes présentées ici constituent bien un habillage de surface : aucune géométrie supplémentaire n'est ajoutée. Il s'agit d'enrichir la surface géométrique, qui représente la forme à grande échelle de l'objet, avec des détails de petite échelle qui n'influent pas sur la perception globale de la forme. D'autres méthodes basées sur l'ajout de géométrie existent [BM93], mais celles–ci sortent du cadre de cette thèse et ne sont donc pas évoquées ici.

Le principal défaut de ces méthodes réside dans la taille des données à stocker. Les auteurs proposent cependant des solutions de compression. Du point de vue de la qualité de rendu, ces modèles sont particulièrement sujet à l'aliasing et plus difficiles à filtrer (voir section 4) et sont, de plus, coûteux à interpoler (à cause du grand nombre de dimensions).

Notons également que, récemment, une extension du *bump mapping* [Bli78], nommée *offset bump mapping* ou *parallax mapping* [KTI+01, Wel04] est apparue dans la communauté du jeu vidéo. L'idée de cette méthode est de stocker une carte de relief avec la texture de perturbations de normales. Lorsque la texture est accédée, la carte de relief permet de calculer un petit déplacement, dépendant du point de vue, qui déforme localement la texture et simule le relief. Cette méthode est très peu coûteuse et augmente l'impression de relief lorsque les motifs de la texture sont fins en comparaison du relief.

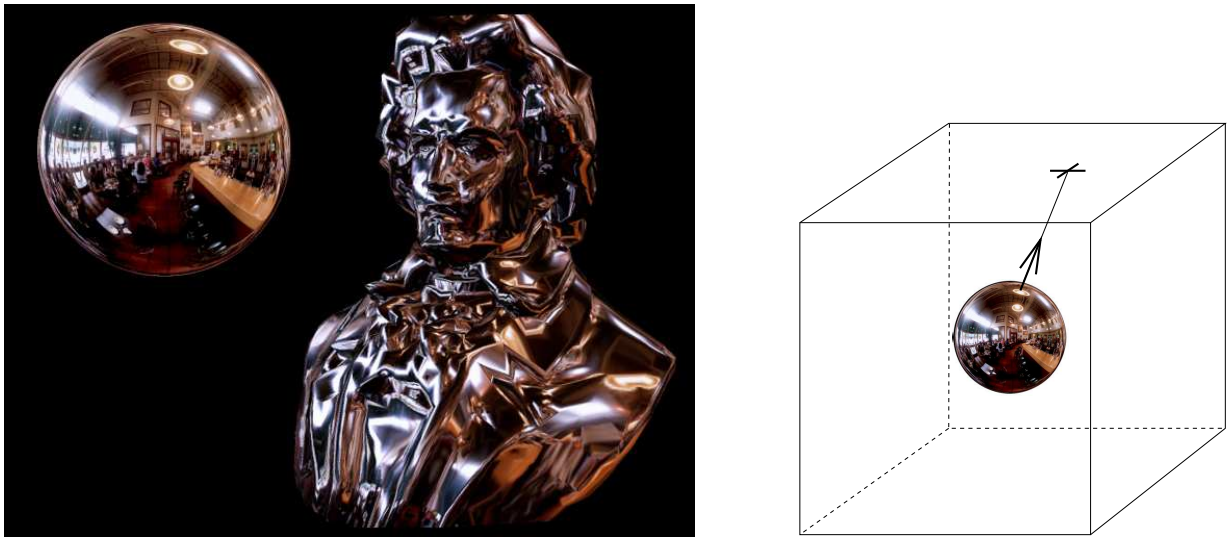


FIG. 2.14 – Cartes d'environnement utilisée pour simuler des réflections

La contribution de l'environnement est stockée dans des textures appliquées sur les six faces d'un cube. Pour trouver les données associées à une direction, un rayon est lancé depuis le centre du cube. L'intersection permet de lire les données depuis la texture de la face intersectée.

2.2 Cartes d'environnement

Les cartes d'environnement [Gre86] ont été développées pour associer à la surface de l'objet des informations provenant de son environnement. Elles ont été initialement utilisées pour simuler des réflexions [BN76, KG79] et réfractions [MH84].

L'hypothèse est que l'environnement est suffisamment éloigné de l'objet, donc considéré à l'infini. Dès lors, le point de départ du rayon réfléchi n'a plus d'importance, seule sa direction détermine la couleur du reflet. Les cartes d'environnement stockent donc des attributs de surfaces dans différentes directions, plutôt que spatialement. Depuis un point quelconque de la surface, la normale et la direction de vue suffisent pour calculer la direction de reflet (ou de réfraction) et accéder aux informations stockées dans la carte d'environnement. La figure 2.14 montre les résultats obtenus par cette technique.

Même si les cartes d'environnement ont été initialement utilisées pour simuler des réflexions, elles définissent néanmoins un attachement d'attribut à la surface d'un objet au même titre que le placage de texture, comme illustré par la figure 2.15. Nous allons voir ci-après (section 2.2.2) que cette idée a été étendue afin de définir un modèle d'habillage évitant certaines des difficultés dues aux paramétrisations 2D.

2.2.1 Principe

Le plus souvent, les cartes d'environnement sont représentées sous la forme de six images plaquées sur les faces d'un cube entourant l'objet. Cette structure est appelée une *cubemap*. La

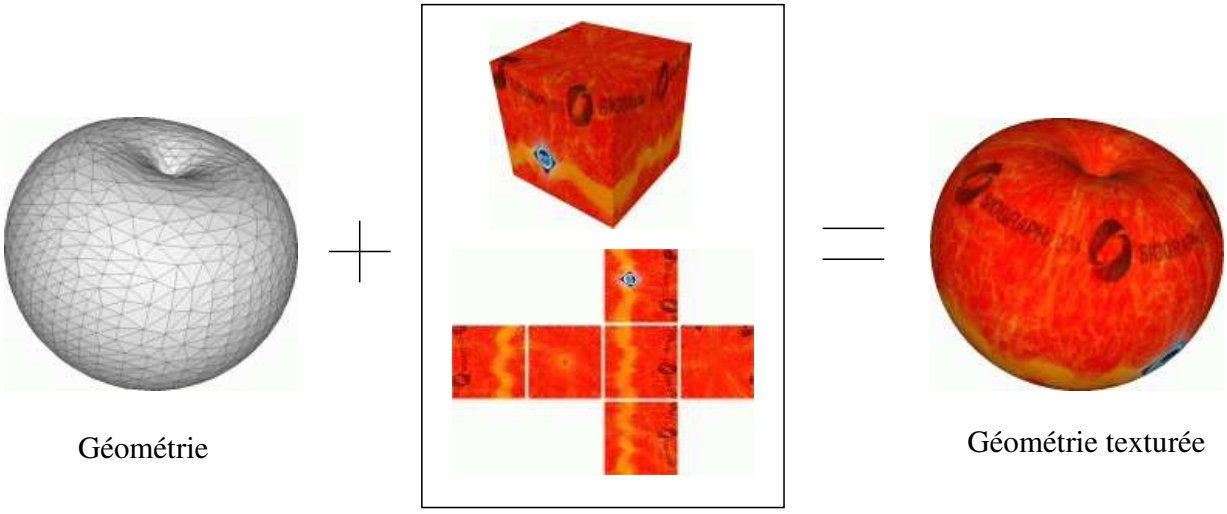


FIG. 2.15 – Cartes d'environnement utilisée comme une texture
Les cartes d'environnement permettent également d'attacher une texture le long d'une surface. (Image extraite de [THCM04])

texture n'est donc plus planaire, contrairement au placage de texture : c'est un ensemble de six images.

L'objet est considéré comme un point au centre du cube. La texture est accédée non pas par une coordonnée représentant un point dans l'espace, mais par un vecteur représentant une direction. Les données sont lues au point d'intersection entre le cube et le rayon partant du centre dans la direction d'échantillonnage (voir figure 2.14). L'accès aux données est efficace : la composante du vecteur ayant la plus grande valeur absolue, ainsi que son signe, permettent de déterminer la face du cube contenant l'information. Ensuite, l'image correspondante peut être utilisée comme dans le cas du placage de texture.

2.2.2 Cartes d'environnement généralisées

Récemment, Tarini et al. [THCM04] ont proposé une extension des cartes d'environnement dédiée à l'attachement de données le long d'une surface de géométrie arbitraire. L'idée est d'entourer la géométrie de l'objet d'un ensemble de cubes alignés sur une grille régulière (voir figure 2.16). Seules les faces extérieures des cubes sont conservées. Les sommets de la géométrie sont ensuite projetés sur cette structure, définissant ainsi une paramétrisation de l'objet sur les faces des cubes. Cette paramétrisation est optimisée afin de minimiser les distorsions. Chaque face est ensuite recouverte d'une image, de la même manière que les faces du cube d'une carte d'environnement.

Cette méthode peut être implémentée efficacement dans les cartes accélératrices récentes. Elle permet de définir des textures avec moins de distorsions que les paramétrisations 2D, et

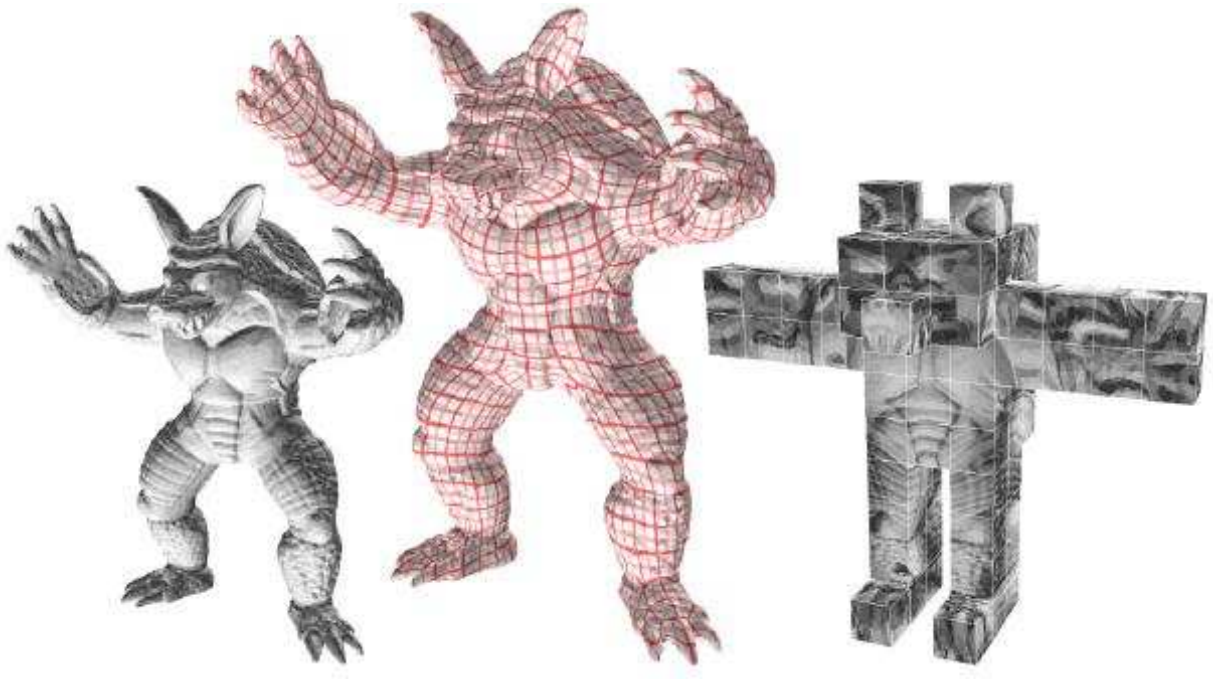


FIG. 2.16 – Cartes d'environnement généralisées [THCM04]

A gauche *: Modèle avec bump–mapping. Une carte d'environnement généralisée stocke les perturbations de normales.* Au milieu *: La paramétrisation est localement générée sur chaque face des cubes mais passe continûment d'une face à l'autre. Il y a relativement peu de distorsion sur la surface.* A droite *: Les cubes de la carte d'environnement généralisée.*

surtout sans introduire de coupures dans la géométrie. L'interpolation linéaire est possible mais un cas particulier intervient à la jonction entre les différentes faces des cubes.

2.2.3 Conclusion sur les cartes d'environnement

Les cartes d'environnement et leurs extensions nous permettent de constater que les modèles d'habillage peuvent mettre en jeu des *structures spatiales* plus complexes que le plan, qui simplifient la paramétrisation, tout en restant efficaces. Ce type de solution est largement facilité par la récente souplesse de programmation des cartes graphiques qui permet de les utiliser directement dans les applications interactives.

Néanmoins, interpolation linéaire et filtrage (voir section 4) sont plus difficiles à mettre en oeuvre, et des défauts visuels peuvent apparaître s'ils sont mal traités. Il devient également moins aisé de peindre les surfaces pour les artistes. L'édition directe des données de texture est difficile et il convient de proposer des outils de création adaptés (voir section 5.1).

FIG. 2.17 – Particules de textures
La surface est recouverte par des imagettes agencées de manière à obtenir l'apparence finale. De gauche à droite : Fleisher [FLCB95], Dischler et al. [DMLG02], Praun et al. [PFH00]

2.3 Particules de textures

2.3.1 Principe

Certains aspects de surface sont formés par l'agencement de petits éléments de texture (appelés motifs, ou *particules de texture*). C'est par exemple le cas des écailles de serpent, des taches du léopard ou des murs de briques (voir figure 2.17). Les travaux de Fleisher [FLCB95], Praun et al. [PFH00] et Dischler et al. [DMLG02] se sont attachés à générer automatiquement ce type de textures, très présentes dans la nature. De même, des aspects locaux tels que des impacts, gouttes, traces de pas, feuilles mortes sur un terrain, sont efficacement représentés par de petits motifs localement appliqués à la surface et répétés en plusieurs exemplaires avec une éventuelle modulation (couleur, taille, orientation, ...). Notons que ces effets peuvent être dynamiques (les traces de pas apparaissent alors qu'un personnage se déplace) ou animés (les écailles coulissent les unes sur les autres, les gouttes d'eau tombent le long de la surface).

2.3.2 Représentations utilisées et difficultés

Ces effets sont classiquement soit représentés à l'intérieur d'une texture globale appliquée à l'objet, soit par de petits éléments géométriques superposés à la surface de l'objet, appelés *decals*.

Texture globale L'approche la plus simple consiste à traiter ce type d'apparence comme une texture classique. Cependant cette représentation gaspille beaucoup de mémoire. Les mêmes motifs sont dupliqués plusieurs fois, l'espace vide entre les motifs est également stocké dans la texture. Peu d'information est donc représentée à l'aide de beaucoup de mémoire.

Lorsque les effets sont dynamiques, la texture doit de plus être mise à jour régulièrement [NHS02] : chaque pixel des motifs animés doivent être recopiés. Ceci peut pénaliser les performances, si de multiples motifs d'assez haute résolution sont présents. De plus, lorsque la surface est complexe et la paramétrisation distordue ou discontinue, la mise à jour peut devenir difficile.

Eléments géométriques superposés à la surface Les jeux vidéos représentent souvent des impacts ou des traces de pas sur les surfaces. L'approche généralement utilisée consiste à superposer à la scène de petits morceaux de géométrie texturée (un simple carré), appelés *decals*. Cette idée a été étendue par Praun et. al [PFH00] pour représenter des textures formées par la répétition, le long de la surface, d'un motif. En fait, il convient de dissocier, pour ce type de texture, les informations de couleur des informations de positionnement du motif. Au lieu d'utiliser une texture globale, les auteurs proposent ainsi de superposer de petits morceaux de géométrie, éventuellement courbes, sur la surface. Ces morceaux de géométrie utilisent tous la même texture (le motif), qui est, elle, stockée une seule fois en mémoire.

Néanmoins, si cette approche permet d'économiser la mémoire texture, le prix à payer en terme de géométrie est élevé : le nombre de particules peut être important et de nombreux triangles deviennent nécessaires pour les représenter. Au–delà de l'ajout conséquent de géométrie, ceci rend difficile l'utilisation d'optimisations telles que les bandelettes de triangles (*triangle strips*) ou les niveaux de détails progressifs. Enfin, plusieurs primitives géométriques étant superposées pour représenter une même surface, des défauts visuels peuvent apparaître à cause d'imprécisions numériques.

Les habillages à base de particules sont donc actuellement difficiles à représenter efficacement, que ce soit en terme de stockage en mémoire ou d'affichage. Nous proposons dans le cadre de cette thèse un modèle qui permet, en particulier, de représenter efficacement ce type d'habillage (voir chapitre 9).

2.4 Habillage en volume

Les modèles d'habillage que nous avons vus précédemment ont tous pour point commun de stocker les données sous la forme d'une ou plusieurs images à deux dimensions. Ces données sont ensuite associées à la surface via une paramétrisation et éventuellement une structure intermédiaire (voir section 2.2).

Nous présentons ici une approche différente, introduite par Perlin [Per85] et Peachey [Pea85] qui utilise une représentation en volume pour définir l'apparence de surface, au lieu d'une image planaire.

2.4.1 Principe

Le principe de l'habillage en volume est de considérer que l'objet est plongé dans un volume de matière. Le volume définit en tout point de l'espace les propriétés du matériau (par exemple sa couleur). En un point de la surface, il suffit de lire dans le volume les propriétés correspondantes. L'avantage de cette méthode est qu'il n'est pas nécessaire de recourir à une paramétrisation planaire : l'habillage ne souffre donc d'aucune distorsion, ni de problèmes de rendu dus à des discontinuités. Visuellement l'objet semble véritablement sculpté dans un bloc de matière, ce qui permet des apparences réalistes notamment avec des matériaux orientés comme le bois ou les veines du marbre (voir figure 2.18).

Il est possible d'animer les objets sans que la texture change. En fait on utilise rarement directement les coordonnées des points de la surface, mais on passe par l'intermédiaire de *co-*

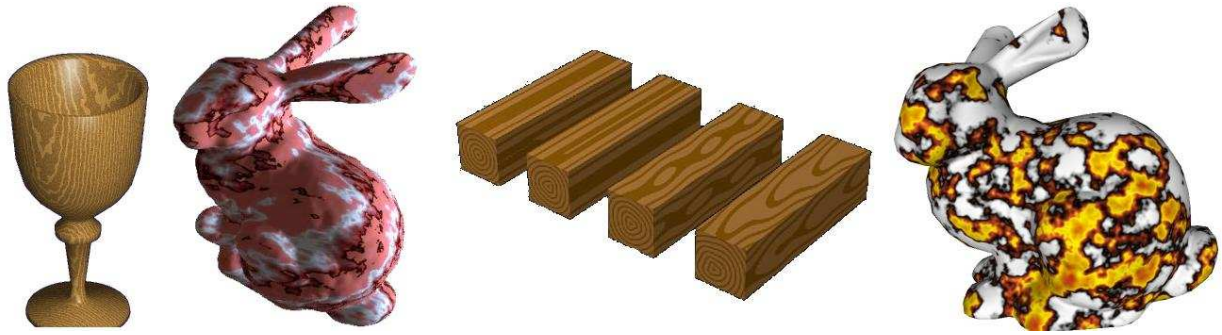


FIG. 2.18 – Textures procédurales définies dans un volume.
La couleur de la surface est calculée en chaque point de l'espace de manière procédurale, donnant l'illusion que la forme à été sculptée dans un volume de matière contenant la couleur.

ordonnées de texture 3D, qui correspondent par exemple aux coordonnées des sommets lorsque l'objet est dans un état de repos. En fait, ceci revient à définir une paramétrisation 3D. Cependant, contrairement au cas 2D, une paramétrisation triviale existe toujours : celle qui consiste à utiliser directement la position des points (l'espace texture correspond alors à l'espace 3D).

2.4.2 Données d'habillage en volume

Une approche simple consiste à stocker l'apparence de la surface dans une grille régulière en volume. Néanmoins les données ne sont utilisées que dans les cases intersectant la surface. Le gaspillage est énorme : seul un très faible pourcentage des cases de la grille est utilisé dans le cas général.

Deux approches permettent d'éviter ce problème : la génération à la volée des données (section 2.4.2.1), ou le stockage dans une grille hiérarchique (section 2.4.2.2).

2.4.2.1 Habillage procédural en volume L'idée des *textures procédurales* est de stocker non pas les données explicites de la texture, mais plutôt un algorithme permettant de les générer. Les données de textures sont alors générées à la volée, à partir des coordonnées d'un point dans l'espace texture.

Ce type de texture est donc particulièrement intéressant pour l'habillage en volume [Per85, Pea85] : le coût en espace mémoire est quasi–nul. Plusieurs types de matériaux volumiques ont été définis avec succès (marbre, bois, granit ...) [EMP+94, Wor96], mais néanmoins, il est difficile de généraliser la méthode à tous types de matériaux. Pour plus de détails sur la génération d'aspects de surface avec des textures procédurales, voir la section 5.2.1.

2.4.2.2 Habillage en volume avec une grille hiérarchique Comme nous l'avons évoqué plus haut, stocker l'apparence de la surface dans une grille en volume régulière résulte en un gaspillage de mémoire conséquent. Afin de dépasser cette limitation, Debry et al. [DGPR02] et Benson et al. [BD02] proposent de stocker l'information dans une grille hiérarchique (un *octree*),

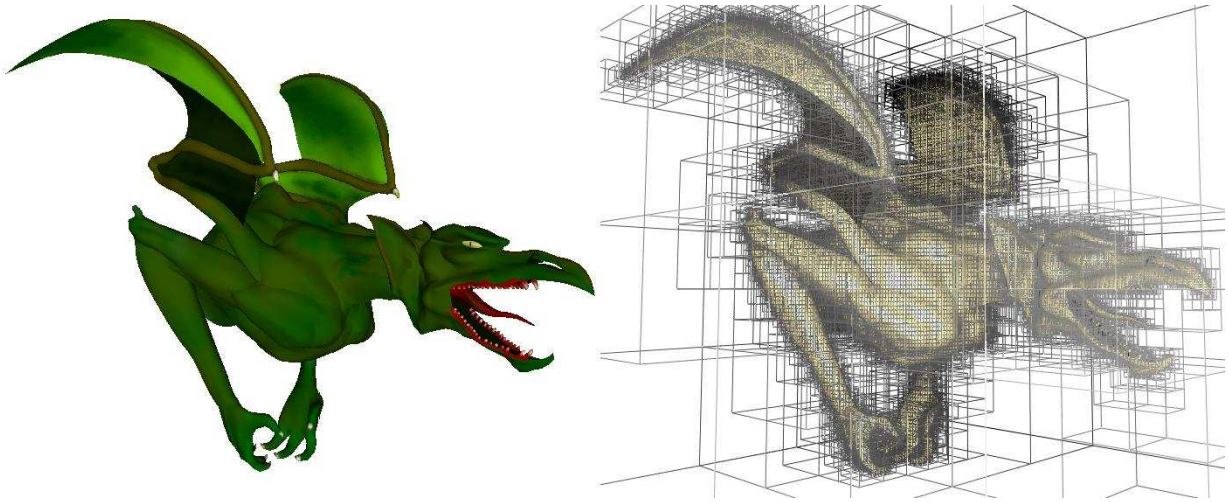


FIG. 2.19 – Texture volumique hiérarchique pour les surfaces (*octree textures*).

A gauche : Objet texturé par une texture volumique hiérarchique. A droite : Structure de la texture. Seules les feuilles de la hiérarchie stockent de l'information. Cette texture requiert 4.5 Mo, la profondeur maximale de la hiérarchie est de 12.

définissant ainsi une texture hiérarchique appelée *octree texture*. La figure 2.19 montre un objet texturé à l'aide d'une texture hiérarchique. Les auteurs définissent les opérations d'interpolation linéaire et de filtrage. Malheureusement, ces textures ne sont pas exploitables directement dans les applications interactives, car non supportées par les cartes accélératrices. Nous proposons dans le cadre de cette thèse (chapitre 7) une implémentation des octree texture sur les cartes graphiques programmables.

3 Modèles d'habillage pour les terrains

Les terrains sont très présents en synthèse d'images. Dès lors qu'une scène extérieure doit être représentée, un modèle de terrain est nécessaire. Les modèles d'habillage présentés en section 2 peuvent être utilisés sur les terrains. Cependant, la surface des terrains présente certaines caractéristiques particulières. En premier lieu, leur paramétrisation n'est pas aussi difficile que dans le cas général (voir section 2.1.2.1). En second lieu, les terrains nécessitent d'habiller des régions de très grande taille, lesquelles peuvent être vues de près comme de loin. En effet, ils représentent souvent des zones de plusieurs dizaines de kilomètres carrés (plus encore pour les simulateurs), mais la résolution de détail nécessaire peut être aussi précise que quelques centimètres si le point de vue est proche du sol (par exemple un marcheur ou un véhicule terrestre). Le terrain est de plus souvent vu à la fois de très près et de très loin (horizon). Notons cependant que seule une petite partie du terrain est visible à chaque instant dans le champ visuel.

La principale difficulté réside donc dans la quantité de données, tant du point de vue du stockage que de celui de la création.

3.1 Textures explicites

Une première approche consiste à utiliser le placage de texture sur la surface du terrain. La texture est alors une très grande image. Comme nous l'avons vu précédemment (section 2.1.2.1) la paramétrisation triviale est souvent considérée comme suffisante.

Cependant, la quantité d'informations à stocker dépasse rapidement les capacités mémoire des meilleurs ordinateurs, et ce de plusieurs ordres de grandeur. Au delà des techniques de compression présentées en section 2.1.7.1, diverses approches ont été proposées pour le cas spécifique des terrains [TMJ98, GY98, Hut98, CE98, DBH00]. Elles consistent à exploiter la cohérence du point de vue pour charger seulement la partie utile de la texture. Ce chargement peut être effectué de manière asynchrone afin de maintenir un temps d'affichage constant durant les déplacements de l'utilisateur : les données sont chargées progressivement, les zones non encore chargées en mémoire montrent une version basse résolution de la texture. Cette approche est justifiée par le fait qu'une animation saccadée est plus perceptible qu'une légère perte de résolution sur la texture pendant le mouvement.

Néanmoins, même lorsqu'il devient possible de les utiliser, créer le contenu de telles textures n'est pas facile. Dans le cas des simulateurs, l'imagerie satellite permet d'obtenir des données, mais pas toujours avec la résolution suffisante (en outre, si les données existent, elles ne sont pas toujours disponibles pour le public). Dans le cas des jeux vidéos, les mondes représentés sont souvent imaginaires et les artistes devraient peindre la texture du terrain pixel par pixel !

C'est pourquoi des modèles d'habillage capables de réduire stockage et temps de création en *générant* l'aspect du terrain ont été proposés.

3.2 Pavages périodiques du plan

Le pavage périodique est basé sur une idée simple : au lieu de stocker une très large texture, seul un motif est stocké et répété périodiquement sur la surface. Les oeuvres de l'artiste

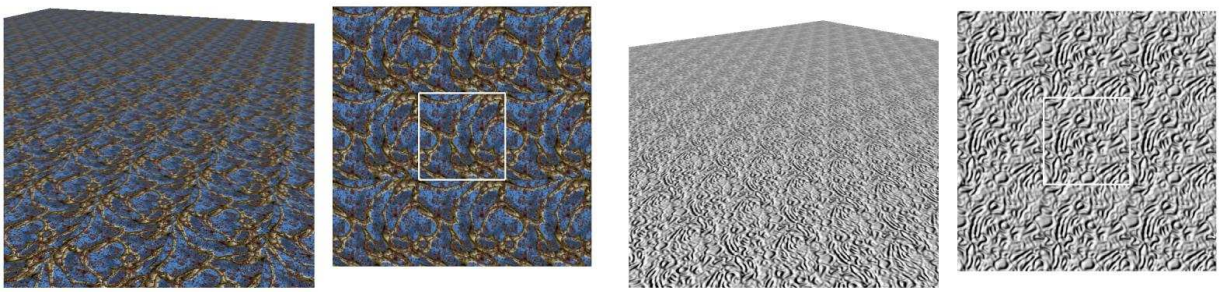


FIG. 2.20 – Pavage périodique du plan.

La texture est répétée plusieurs fois pour texturer le plan. Malheuresement à grande échelle les répétitions deviennent évidentes et des effets d'alignements apparaîssent dans l'image.

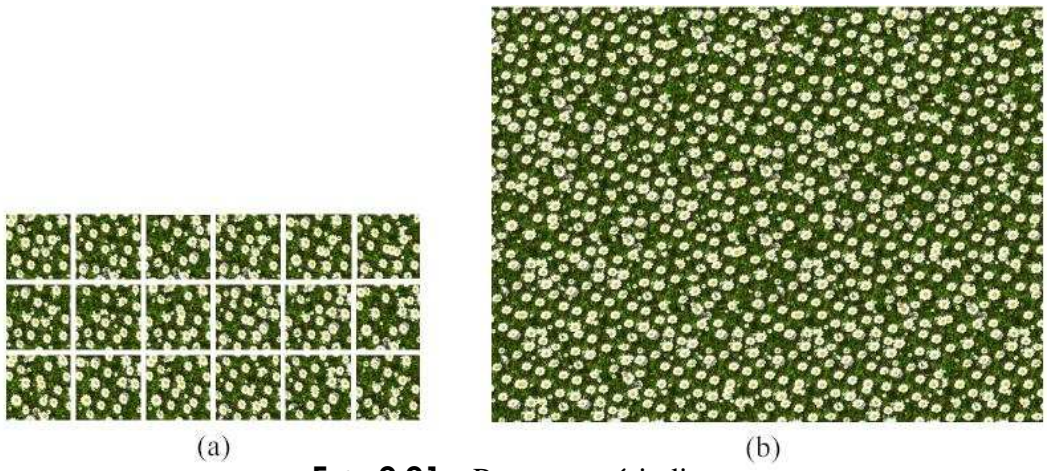
M.C. Escher illustrent très bien ce principe. Quand la surface est paramétrée, cette approche est implémentée comme une extension du placage de texture : le motif est répété régulièrement pour simuler une large texture. Cette "répétition" peut être effectuée très simplement en multipliant les coordonnées de texture puis en appliquant un modulo sur les coordonnées du point d'échantillonnage. Par exemple, si l'on considère les coordonnées de texture usuelles entre $[0,1[$, des coordonnées multipliées par 5 feront se répéter le motif 5 fois. Cela revient à considérer que l'espace texture est cyclique avec une topologie torique. L'interpolation linéaire et le filtrage peuvent être adaptés sans difficulté majeure. Créer un contenu de texture adapté (gardant un aspect continu lorsque la texture est répétée) n'est pas toujours facile : le contenu de la texture le long des arêtes horizontales (resp. verticales) doit en effet se correspondre pour éviter les discontinuités visuelles. Nous présentons quelques techniques de création, section 5.3.

Le principal défaut de cette méthode est que la répétition devient très visible à grande échelle. Des structures (alignements) apparaissent et dégradent la qualité des images. Ce problème est illustré figure 2.20.

3.3 Pavages apériodiques du plan

Afin de supprimer les défauts visuels de grande échelle dus à la cyclicité, l'idée du pavage apériodique est de réaliser un pavage non pas avec une seule, mais avec plusieurs textures, appelées pavés . Les contenus des pavés ont des arêtes compatibles : elles peuvent être mises côte à côte sans faire apparaître de discontinuités dans le résultat. Le pavage est créé de manière apériodique afin qu'à grande échelle il ne soit pas possible de repérer de larges structures similaires. Ce principe est illustré figure 2.21. Notons que les pavés ne sont pas nécessairement carrés.

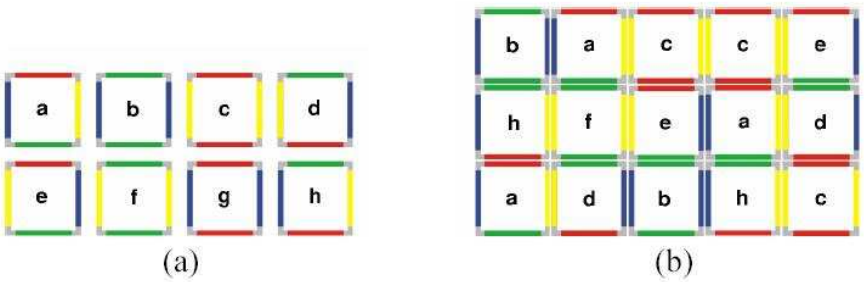
La combinaison de plusieurs textures pour en former une plus grande est une technique utilisée depuis longtemps par les jeux vidéos, notamment pour créer des décors à faible coût mémoire. Des jeux très populaires comme *Mario Bross* ou *Zelda* de *Nintendo* on notamment recours à cette technique ; mais il existe quantité d'autres exemples. Cette approche est toujours utilisée dans des jeux plus récents pour habiller des terrains : différentes petites textures



(a) (b)

FIG. 2.21 – Pavages apériodiques

(a) Différentes petites images sont combinées apériodiquement pour former une large texture qui ne montrera pas de répétitions à grande échelle (b). (Image extraite de [CSHD03])



(a) (b)

FIG. 2.22 – Pavés de Wang

(a) : 8 pavés de Wang utilisées par Cohen et al. [CSHD03] pour paver le plan aléatoirement. Les couleurs représentent les contraintes de bordure. (b) : Pavages apériodique du plan. Les nouveaux pavés sont ajoutés en fonction des pavés déjà présents, de gauche à droite et de haut en bas.

représentant divers types de surfaces (herbe, roc, sable, terre, ...) sont mises côte à côte pour donner l'illusion d'une plus grande texture. Néanmoins, la plupart du temps, le pavage est créé par l'artiste : la surface du terrain est découpée selon une grille régulière ; l'artiste choisit un pavé pour texturer la géométrie de chaque case.

Les travaux menés récemment en informatique graphique ont consisté à automatiser la création du pavage apériodique et des pavés . Notons que ces travaux s'inspirent souvent de l'étude des propriétés des pavages du plan effectuée en cristallographie et mathématiques (voir par exemple les pavages de Penrose [Gla98]). Il est important de souligner que l'apériodicité obtenue avec ces pavages n'est pas toujours visuellement intéressante car des structures régulières, facilement repérables, peuvent apparaître dans le choix des pavés .

Stam [Sta97] montre comment utiliser une méthode de pavage apériodique du plan pour créer des textures, qui plus est animées, à faible coût mémoire. La méthode utilise les pavés de Wang (*Wang Tiles*) [GS86]. Ces pavés carrés ont différentes contraintes de bordures (voir figure 2.22). Il

existe une méthode constructive permettant d'obtenir un pavage apériodique du plan, respectant les contraintes, avec seulement 16 pavés différents. Cohen et al. [CSHD03] utilisent également les pavés de Wang pour générer automatiquement de très larges textures. La méthode présentée crée des textures grâce à un choix aléatoire de pavés qui respecte les contraintes de bordure mais diffère des règles de construction des pavages de Wang utilisées par Stam. Ceci permet un résultat visuellement plus plaisant car moins de structures apparaissent dans le choix des pavés . De plus, les auteurs s'attachent à proposer des solutions pour générer des textures aux apparences variées et non homogènes. Ils proposent notamment d'utiliser plus de pavés et de contraintes au bord pour une meilleure variété dans le résultat.

Contraintes de bordure et alignements visibles Afin de garantir un résultat continu, le contenu des pavés doit respecter des contraintes de bordure. Cependant, ces contraintes peuvent créer des structures régulières visibles à grande échelle. Prenons le cas d'un pavage carré avec une seule contrainte horizontale : tous les pavés ont le même contenu sur leurs bords horizontaux. Si un élément très visible apparaît sur la bordure, celui-ci va donc être répété sur toutes les arêtes horizontales. Ceci provoque l'apparition de structures régulières à grande échelle, comme dans le cas du pavage périodique. Le problème est également présent dans les coins des pavés où, cette fois, deux contraintes (horizontales et verticales) sont en jeu.

Ce problème existe également si le nombre de pavés est petit et que la texture contient des éléments très visibles. Prenons l'exemple d'un jeu de pavés représentant du gazon : si un seul des pavés contient une fleur rouge très visible et que ce pavé est utilisé trop souvent, des alignements apparaîtront : en effet, la fleur étant située à une position constante à l'intérieur du pavé, elle va apparaître sur le pavage selon une grille régulière, révélant ainsi la structure sous–jacente du pavage. Il est possible d'utiliser plus de pavés et plus de contraintes sur les arêtes. Mais vue la combinatoire nécessaire pour satisfaire toutes les contraintes possibles (sur les quatre bords), le nombre de pavés peut augmenter rapidement. Cela requiert plus d'espace mémoire mais également plus de travail de création pour générer le contenu adéquat. Néanmoins ces défauts ne sont visibles qu'à large échelle et lorsque les pavés contiennent des éléments très visibles. Les pavages apériodiques sont généralement utilisés avec succès sur de plus grandes surfaces que les pavages périodiques, en particulier pour des matériaux relativement homogènes (terre, sable, herbe, ...).

Interaction avec la géométrie Comme nous l'avons évoqué, la géométrie de la surface habillée par un pavage doit être découpée selon une grille régulière pour permettre à l'artiste, ou à l'algorithme de pavage, de choisir un pavé pour texturer chaque case. Ceci a plusieurs conséquences néfastes. Tout d'abord, la géométrie est alourdie. Le pavage étant généralement assez fin (puisque utilisé pour représenter des détails de surface) l'ajout de géométrie peut être conséquent. Or la géométrie doit être gérée de manière globale par l'affichage, alors que la texture est seulement appliquée localement, sur les surfaces visibles. D'autre part, ces contraintes sur la géométrie rendent encore plus difficiles les optimisations géométriques comme les bandelettes de triangles (*triangle strips*) et les niveaux de détails, pourtant essentielles sur les terrains. Dernier

FIG. 2.23 – Pavage de surface

Des pavés triangulaires sont disposés sur la surface d'un objet. L'utilisateur peut contrôler la distribution spatiale des pavés .

point, les discontinuités étant introduites par de la géométrie, il devient impossible d'effectuer le filtrage de la texture résultante correctement (voir section 4.5).

Nous proposons au chapitre 3 un modèle d'habillage qui permet, en particulier, de générer des pavages apériodiques en répondant à ces limitations (alignements visibles, dépendance de la géométrie et filtrage).

3.4 Pavages de surfaces

Les pavages présentés ci–avant sont des pavages du plan. On les utilise sur les terrains en supposant les distorsions dues au relief négligeables. Néanmoins la notion de pavage peut s'étendre aux surfaces quelconques.

En particulier, Neyret et al. [NC99] ont montré comment un pavage triangulaire régulier peut être créé sur une surface de topologie arbitraire (notons que ce ne serait pas possible avec des carrés). Les pavés utilisés sont des triangles équilatéraux (voir figure 2.23). L'approche consiste à projeter sur la surface de l'objet un maillage triangulaire régulier. Chaque "face" de ce maillage devient alors support d'une texture appliquée localement (un pavé triangulaire). De plus, la texture résultante est continue : le contenu des textures, sur les arêtes de triangles voisins, se correspondent. L'avantage de la méthode est de ne jamais avoir à spécifier globalement une paramétrisation (les triangles sont paramétrés séparément) et donc d'éviter les problèmes qui y sont associés.

3.5 Quelques techniques utilisées dans le jeu vidéo

Les jeux vidéos doivent faire face au problème de l'affichage de terrains depuis plusieurs années. A cause des contraintes techniques parfois extrêmes (certaines consoles récentes, comme la *PlayStation 2* de Sony, possèdent uniquement 32 Mo de mémoire *au total*) et des exigences

de performance, les développeurs ont mis au point diverses techniques pour obtenir, malgré tout, des images visuellement riches.

Loin de se vouloir exhaustive, cette section présente certaines des techniques employées par les créateurs de jeu, afin d'illustrer les problèmes rencontrés en pratique.

3.5.1 Mélange de différentes échelles

Afin de rendre moins évidentes les répétitions à grande échelle des pavages périodiques, plusieurs textures sont souvent combinées sur le terrain. Par exemple, une première texture représente les détails très fins, une seconde introduit des variations de couleur de plus grande échelle, et enfin une troisième encode la lumière reçue par les différentes parties de la surface. Notons que les variations de couleur et lumière peuvent aussi être encodées en chaque sommet de la géométrie et interpolées sur les triangles, à condition qu'elles soient de suffisamment basse fréquence et que les éventuels niveaux de détails appliqués sur la géométrie n'introduisent pas d'effets de discontinuité temporelle (sommet apparaissant ou disparaissant brutalement).

La combinaison des différentes couches de texture, chacune apériodique à l'échelle où leur contribution est prépondérante, permet un résultat visuellement plus plaisant.

3.5.2 Transitions entre matériaux

Les terrains ont une apparence généralement très variée. Différents matériaux recouvrent la surface (sable, terre, roc, herbe, ...) le long de zones plus ou moins bien délimitées. Chacun de ces matériaux peut être représenté par un pavage (périodique ou apériodique). L'utilisation de différents matériaux permet également de masquer les répétitions à très grande échelle. Néanmoins, il faut définir les transitions entre zones de différents matériaux. Ce problème est crucial en pratique mais a, malheureusement, été relativement peu exploré en recherche où l'on se concentre souvent sur la représentation d'un unique matériau.

Transitions par interpolation Une des approches classiquement utilisée dans les jeux consiste à effectuer une interpolation entre les textures des matériaux dans les zones de transition : en passant la frontière entre deux zones, une texture devient transparente pendant que la seconde devient opaque. Si cette approche est simple, elle n'offre malheureusement pas des résultats très réalistes : par exemple l'herbe disparaît progressivement pour laisser apparaître de la roche, comme si elle était peinte à la surface d'une plaque de verre de moins en moins opaque.

Transitions explicites Une autre approche, offrant des résultats plus réalistes, consiste à définir des textures de transition prévues pour s'intercaler entre les zones de deux matériaux. Cette technique est généralement employée dans le cadre du pavage apériodique. Pour les transitions entre deux matériaux dans quatre cases voisines d'un pavage carré, on peut observer 16 configurations possibles (voir figure 2.24). L'artiste peut donc fournir les 14 pavés de transition (deux configurations correspondent au cas où les quatre cases voisines sont du même matériau). Ce principe est illustré figure 2.24.

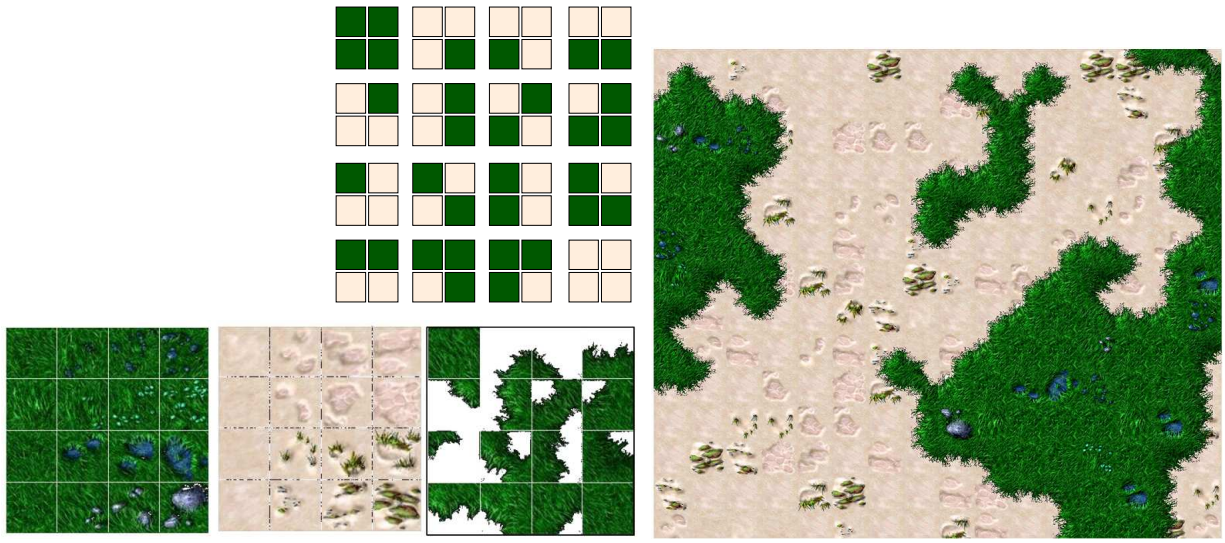


FIG. 2.24 – Transition entre matériaux pour pavages apériodiques

En haut au milieu : *16 configurations existent pour les transitions entre 4 cases d'un pavage avec 2 matériaux.* En bas à gauche : *Les pavés de chaque matériau et les pavés de transition.* A droite : *Pavage apériodique avec transitions.*
Pavés de textures extraits (avec autorisation) de Warcraft® III : Reign of Chaos™. *© 2002 Blizzard Entertainment ®. Tous droits réservés. Pavage réalisé avec notre méthode, présentée chapitre 3.*

Ces transitions peuvent être sélectionnées automatiquement lorsque l'artiste décide quels matériaux doivent apparaître dans les différentes cases du pavage régulier. Ce principe est employé dans de nombreux jeux de stratégie temps réel, comme par exemple *Warcraft III* de *Blizzard Entertainment*, dans lesquels l'utilisateur a un point de vue suffisamment éloigné du terrain pour percevoir d'éventuelles répétitions de large échelle, tout en continuant à distinguer de nombreux détails au sol.

L'un des défauts de cette technique est de devoir expliciter les transitions entre toutes les paires de matériaux. Afin de réduire la quantité de données, les transitions sont souvent spécifiées entre un matériau et un fond transparent. La texture de transition est ensuite surimposée, par transparence, à la texture du terrain.

3.5.3 Ajout de hautes fréquences

L'un des effets du manque de détails, notamment dans les simulateurs de vol, est d'empêcher l'utilisateur d'évaluer correctement les distances et les vitesses. Une technique consiste à ajouter suffisamment de hautes fréquences pour donner l'illusion que l'image contient des détails (*detail textures*). Ces détails ne sont pas vraiment cohérents (il s'agit souvent d'ajouter du bruit à l'image) mais le "grain'" ajouté suffit à redonner à l'utilisateur des repères de position et de vitesse.

Pour ce faire une texture contenant de hautes fréquences (par exemple du bruit blanc) est combinée à la texture des surfaces. Cette couche de détails n'est ajouté que lorsque l'utilisateur

est suffisamment proche des surfaces. Ainsi, il est difficile de repérer des répétitions de large échelle : les détails ne sont présents que lorsque le point de vue est proche et que seule une petite partie de la surface est visible.

Cette technique est également utilisée dans les jeux, avec un mode de vue à la première personne, notamment le jeu *Serious Sam* de *Croteam Ltd.*, où de fins détails apparaissent sur les murs lorsque le joueur s'en approche (craquelures, crépi, ...).

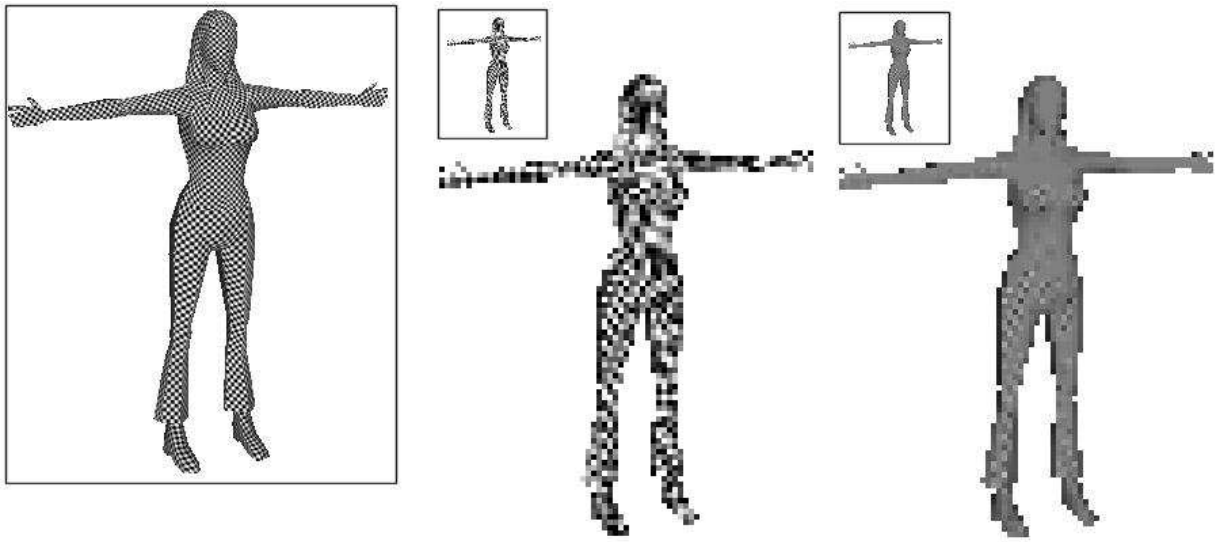


FIG. 2.25 – Des effets d'aliasing apparaissent

A gauche : *Un modèle de personnage est recouvert d'un damier. Cette image possède suffisamment de résolution et aucun aliasing n'est visible.* Au milieu : *Dans le cadre, on peut voir un rendu de faible résolution (reproduit en plus grand à côté). Les pixels de l'écran ne peuvent capturer correctement le motif du damier qui est de trop haute fréquence, et de l'aliasing apparaît.* A droite : *Dans le cadre le même rendu de faible résolution, mais en utilisant un algorithme de filtrage. Dans cette image, le filtrage élimine les trop hautes fréquences. L'effet d'aliasing n'est plus visible ; on ne voit plus que la couleur moyenne.*

4 Filtrage

Le filtrage permet d'adapter les données générées le long de la surface par le modèle d'habillage à la résolution de l'image. En effet, l'aliasing se produit quand les éléments de couleur générés par le modèle d'habillage sont plus denses que les pixels de l'écran. Chaque pixel de l'écran ira alors sélectionner un seul de ces éléments de couleur : de l'information est perdue. L'habillage généré est *sous–échantillonné* par les pixels de l'écran. Il s'agit exactement d'un problème de traitement du signal : la fréquence des motifs (le signal) est plus élevée que la fréquence d'échantillonnage de l'écran. On dépasse la limite de Nyquist et le signal est dégradé. Des effets de moiré et de grouillement de couleurs apparaissent. Ceci est illustré figure 2.25. Comme l'aliasing introduit des basses fréquences parasites et est un phénomène instable, il est très visible et dégrade la qualité des images et animations. Le filtrage consiste à pré–traiter le signal (la texture) pour qu'il puisse être correctement échantillonné par les pixels de l'écran.

4.1 Origine du problème

L'aliasing se produit car les pixels sont considérés comme des points sur l'écran. La scène n'est donc échantillonnée que sur ces points particuliers : on ignore ce qui se passe *entre* deux pixels. Dans le cas d'un appareil photographique (ou de l'oeil), plusieurs rayons lumineux par-

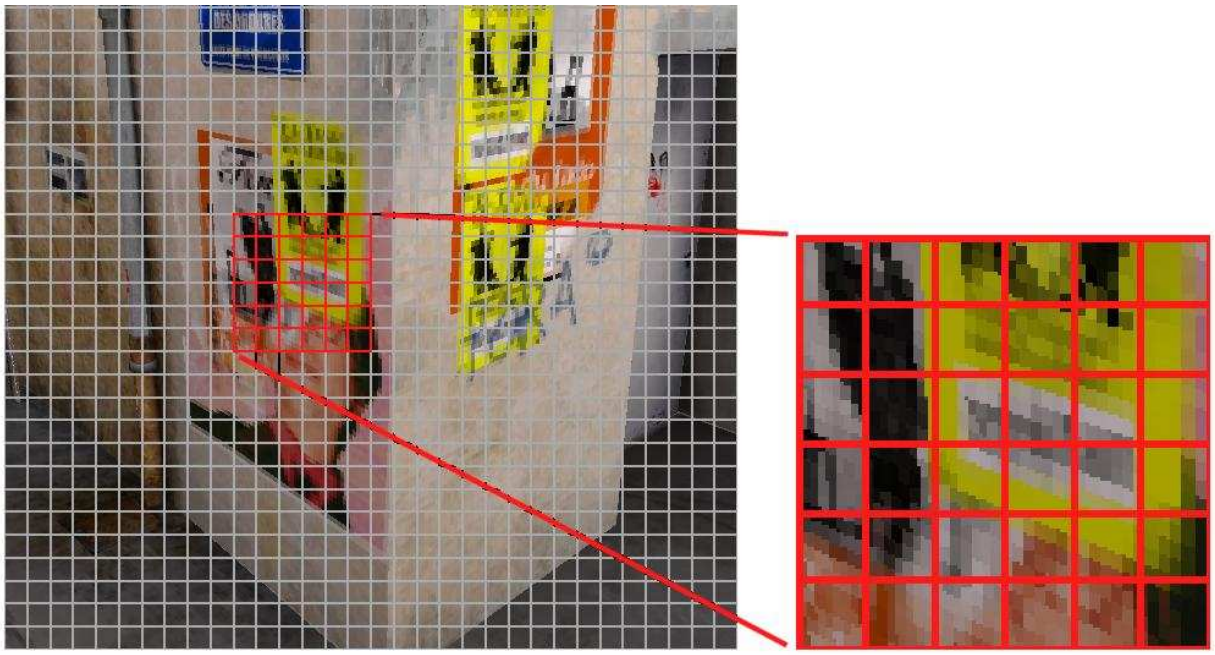


FIG. 2.26 – Les pixels sont influencés par un morceau de la surface.

A gauche : *La grille représente les pixels de l'écran. Chaque pixel contient un morceau de la surface dont la couleur est définie par le modèle d'habillage* . En bas à droite : *Gros plan sur le contenu de quelques pixels de l'écran.*

ticipent à la couleur des éléments photosensibles de la pellicule. Ceci provient du fait que les éléments photosensibles ne sont pas des points : ils ont une surface non nulle. Tout rayon lumineux est pris en compte par l'un ou l'autre des éléments sensibles (on ne perd pas d'information). En somme, les éléments sensibles *intègrent* sur un petit domaine spatial plus qu'ils n'échantillonnent. C'est pour cette raison que les photographies ne souffrent pas d'aliasing , contrairement aux images de synthèse. Une des approches pour éliminer l'aliasing consiste donc à traiter les pixels non pas comme des points, mais comme de petits éléments à la surface de l'écran.

Considérons maintenant une surface habillée (texturée), affichée à l'écran. Supposons que le modèle d'habillage forme une grille de couleur sur la surface (ce qui est le cas de la plupart des modèles générant des données à une résolution fixe). Lorsque la surface s'éloigne de l'observateur, plusieurs éléments de couleur sont inclus dans la zone couverte par un unique pixel de l'écran, comme illustré figure 2.26. Pour filtrer le résultat, la couleur du pixel devrait tenir compte de l'ensemble des éléments de couleur du morceau de surface l'influençant (il faut en faire la moyenne).

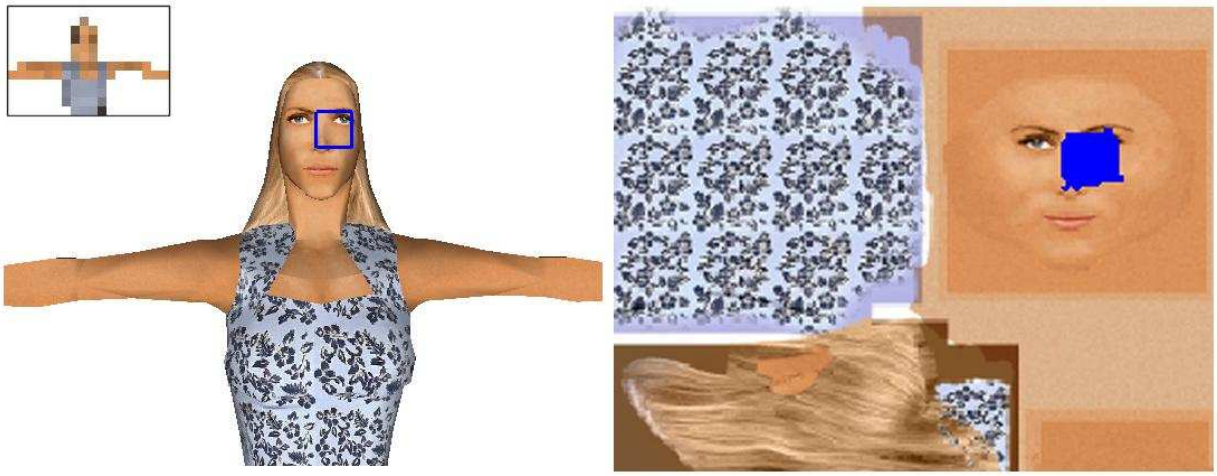


FIG. 2.27 – Empreinte du pixel dans la texture

A gauche : *Le carré bleu (en gras, sur le visage) délimite la zone correspondant à un des pixels de l'image de faible résolution présentée dans le cadre en haut à gauche. La couleur du pixel doit tenir compte du morceau de surface situé à l'intérieur du carré bleu.* A droite : *La zone bleue (gris foncé) représente l'empreinte, dans la texture, correspondant au pixel (carré bleu à gauche). L'empreinte permet de déterminer la partie de la texture devant contribuer à la couleur du pixel.*

4.2 Sur–échantillonnage

Une approche classique du filtrage, le *sur–échantillonnage*, consiste à générer des images de plus haute résolution que désiré, puis à réduire la résolution à l'aide d'un filtre. Ce filtre va permettre de simuler la contribution de plusieurs rayons lumineux : tout se passe comme si l'on avait considéré plusieurs échantillons par pixel au lieu d'un seul. Si ce type d'approche est générique, il n'élimine cependant pas complètement l'aliasing : il ne fait que repousser la limite à laquelle il se produit. Les surfaces obliques ou éloignées pouvant générer des fréquences arbitrairement élevées, de l'aliasing sera toujours visible. Dans le cas des modèles d'habillage , et en particulier du placage de texture, il est possible de traiter le problème à la source, en filtrant les trop hautes fréquences directement dans la texture ; opération qui de plus peut être accélérée par un pré–calcul comme nous allons le voir par la suite.

4.3 Filtrage et modèles d'habillage

4.3.1 Filtrage pour le placage de texture

4.3.1.1 Principe Plutôt que de considérer le problème sur l'écran, on cherche en général à résoudre le problème dual : Le pixel, considéré carré sur l'écran, est (anti–)projeté dans l'espace texture. L'empreinte du pixel dans l'espace texture permet de déterminer quels sont les texels contribuant à la couleur du pixel, comme illustré figure 2.27. La forme de cette empreinte dépend de la projection perspective, de la position de la surface, mais également de la paramétrisation.

Déterminer l'empreinte du pixel et intégrer les contributions des différents texels étant très coûteux, on souhaite ne pas avoir à le faire au moment du rendu. Les chercheurs se sont donc attachés à trouver des approximations raisonnables, qui permettent autant que possible d'effectuer les traitements coûteux en pré–calcul. L'état de l'art de Heckbert [Hec86] sur le placage de texture détaille plusieurs de ces techniques. La plupart des algorithmes de filtrage font l'hypothèse que le contenu de la texture est continu (pas de discontinuité comme dans les atlas de texture). Ils ignorent également le fait que certaines parties de la texture peuvent devenir invisibles selon le point de vue et supposent généralement que l'empreinte du pixel à une forme prédéterminée (carré, rectangle, ellipse, ...).

L'algorithme de *MIP-mapping* Différentes méthodes ont été proposées pour approximer l'empreinte du pixel et effectuer rapidement l'intégrale sur les texels concernés. Nous présentons ici l'algorithme le plus couramment utilisé : l'algorithme de MIP-mapping introduit par Williams en 1983 [Wil83]. Connaître cet algorithme nous permettra par la suite de comprendre certains défauts visuels apparaissant avec les paramétrisations discontinues (voir section 4.3.1.2), mais également de mieux comprendre comment filtrer correctement les données générées par nos modèles d'habillage .

"MIP" signifie *multum in parvo*, c'est à dire "multiples dans un même espace". Ceci fait référence aux multiples éléments de couleurs (ici des texels) pouvant être contenus dans un même pixel de l'écran. Dans cette méthode l'empreinte du pixel dans la texture est approximée dans l'espace texture par un carré aligné sur les axes. L'idée est de pré–calculer l'intégrale pour des empreintes carrées de différentes tailles alignées sur une grille. Lors du rendu, une empreinte carrée est estimée pour le pixel, et l'intégrale sur cette empreinte est reconstruite à partir des intégrales pré–calculées.

Afin de permettre un filtrage rapide, la texture est supposée avoir une taille en puissance de deux et est stockée sous la forme d'une pyramide de Gauss (voir Figure 2.28). Chaque niveau de la pyramide est deux fois moins large que le précédent et correspond à une version filtrée de l'image précédente. Le filtre utilisé est souvent une simple moyenne, mais des filtres plus évolués peuvent être choisis. Le premier niveau de la pyramide est la texture originale, le dernier niveau a une résolution de 1×1 texel. Cette organisation demande environ 33 % d'espace mémoire supplémentaire, ce qui est généralement acceptable au vu du gain de qualité.

Pour une empreinte de pixel d'une taille donnée, on reconstruit l'intégrale comme suit. La taille de l'empreinte est utilisée pour déterminer à quelle résolution, ou niveau de la pyramide, l'information doit être lue. La taille idéale se situe entre deux niveaux. Une interpolation linéaire est alors effectuée entre les contributions des deux niveaux. Chaque niveau est lui traité comme une texture ordinaire (leur résolution est adaptée au pixel, puisqu'ils sont sélectionnés par l'empreinte). La couleur dans chaque niveau est donc donnée par l'interpolation bilinéaire du placage de texture. Il faut donc effectuer trois interpolations : une dans chaque niveau de la pyramide, puis une entre les deux niveaux (d'où l'appellation, parfois, d'interpolation *tri–linéaire*).

La taille de l'empreinte du pixel peut être déterminée en estimant les dérivées partielles de la paramétrisation $(u(x,y), v(x,y))$ en fonction du repère écran (x,y). Les deux dérivées partielles recherchées sont $(\frac{\partial u}{\partial x}, \frac{\partial v}{\partial x})$ et $(\frac{\partial u}{\partial y}, \frac{\partial v}{\partial y})$.

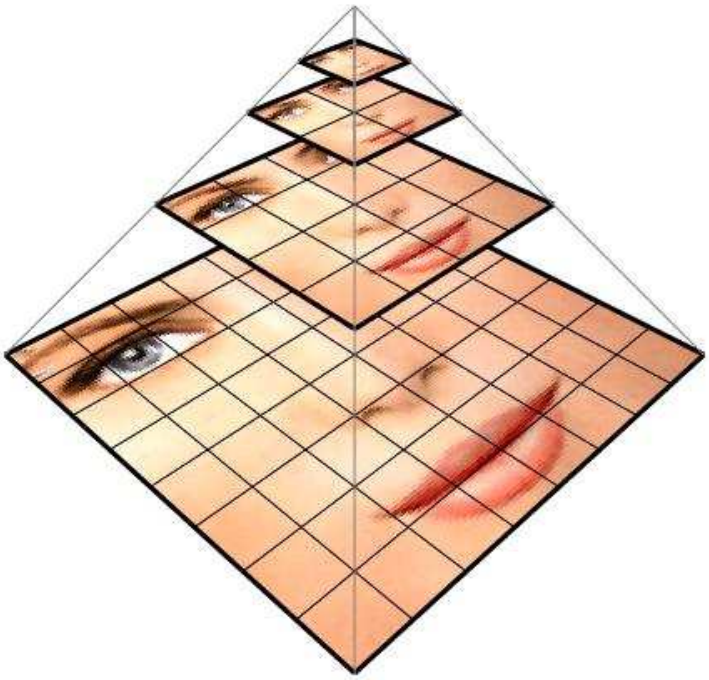


FIG. 2.28 – Pyramide de MIP–mapping

La pyramide contient la texture à plusieurs résolutions. Chaque niveau est deux fois plus petit que le précédent. Un pixel d'un niveau contient la couleur moyenne des quatre pixels correspondant au niveau précédent.

Cette estimation peut être calculée de différentes manières :

- En examinant la coordonnée de texture utilisée par les pixels voisins dans l'écran, il est possible de déduire le nombre de texels utilisés entre deux pixels. Il s'agit en fait d'une estimation numérique des dérivées partielles.
- Selon l'algorithme utilisé, cette information peut être directement disponible, analytiquement. Par exemple, un algorithme de rasterisation effectue l'interpolation des coordonnées de texture. Il calcule donc directement leur variation entre les pixels.

Dans les deux cas, on obtient une information de longueur selon deux dimensions : l'axe des abscisses et l'axe des ordonnées de l'écran. Les deux longueurs concurrentes sont $\sqrt{\frac{\partial u}{\partial x}^2 + \frac{\partial v}{\partial x}^2}$ et $\sqrt{\frac{\partial u}{\partial y}^2 + \frac{\partial v}{\partial y}^2}$. L'algorithme de MIP-mapping considérant l'empreinte du pixel comme étant carrée et alignée sur les axes, il faut utiliser ces deux informations pour estimer la taille du carré. Si l'on choisit la plus grande variation, la texture sera trop filtrée dans une direction. En conséquence on ne verra pas d'aliasing mais l'image perdra en netteté. Si, au contraire, on choisit la plus faible variation, l'image gagnera en netteté, mais il restera de l'aliasing . Ce réglage peut être laissé au programmeur via un paramètre de biais. Cependant, l'isotropie de l'empreinte entraîne des cas pathologiques, malheureusement fréquents, quand les empreintes réelles sont très étirées (silhouettes, plans inclinés). Afin d'atténuer la perte de netteté parfois produite par

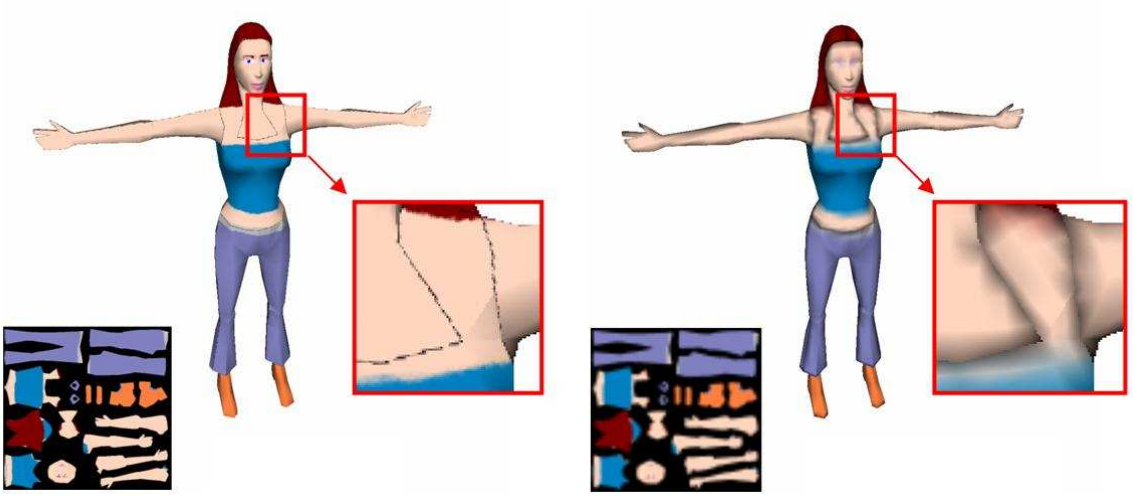


FIG. 2.29 – Filtrage erroné du aux discontinuités.
A gauche : *L'interpolation linéaire utilise des texels situés en dehors des différentes parties de l'atlas. Visuellement la couleur de fond s'étale dans la texture.* A droite : *Le problème s'aggrave encore avec le filtrage : la couleur de fond est utilisée dans la couleur moyenne.*

l'algorithme, les artistes trichent parfois légèrement en appliquant des traitements sur les niveaux de la pyramide qui ne correspondent alors plus à des versions filtrées de la texture d'origine.

D'autres approches permettent de prendre en compte des empreintes plus complexes. Le filtrage *anisotropique* aujourd'hui présent dans nombre de cartes graphiques considère des empreintes rectangulaires quelconques. Il effectue une petite intégration durant le rendu le long de l'axe majeur du rectangle estimant l'empreinte, à l'intérieur du niveau de la pyramide sélectionné par la plus petite dimension. Les tables de Crow [Cro84] (également appelées *Summed Area Tables*) permettent de considérer des empreintes rectangulaires alignées sur les axes. Tous les calculs d'intégrales sont effectués en pré–traitement.

4.3.1.2 Filtrage incorrect dû aux discontinuités

Toutes les méthodes de filtrage supposent qu'on peut estimer la zone de texture à intégrer à partir de valeurs différentielles calculées au centre des pixels, et donc que la zone couverte par l'empreinte est continue dans la texture. Or, les discontinuités souvent introduites dans la paramétrisation (voir section 2.1.2) ne respectent pas l'hypothèse de continuité des algorithmes de filtrage. Si cela ce produit, différentes parties voisines dans la texture mais non voisines à la surface de l'objet sont utilisée par les méthodes de filtrage (comme le MIP–mapping) effectuées dans l'espace texture. Ce problème est illustré figure 2.29.

Le résultat est un effet de débordement de couleur lors du rendu. Ce phénomène est la combinaison de deux problèmes : la prise en compte dans le filtrage de la couleur utilisée pour remplir les vides (par exemple dans un atlas de texture) et l'interpolation linéaire effectuée dans les niveaux de MIP-mapping (voir également la section 2.1.6.2).

Deux "astuces" (qui ne sont que des solutions partielles) sont utilisées par les artistes dans les applications interactives : en premier lieu, la couleur de fond est choisie de manière à minimiser son impact visuel en cas de débordement (en général la couleur moyenne de la texture) ; en second lieu, la texture est créée de manière à ce que chaque partie distincte puisse déborder d'au moins un pixel, comme expliqué section 2.1.6.2. Ceci permet d'obtenir une interpolation linéaire sans biais de couleur (voir figure 2.10). Néanmoins dans le cas du filtrage, ce débordement d'un pixel devrait être effectué dans l'ensemble de la pyramide de MIP-mapping. Ceci implique de garder un large espace vide entre les différentes sous–parties, gaspillant d'autant plus de mémoire. Cette approche est rarement choisie, au détriment de la qualité visuelle puisque la couleur filtrée devient fausse. Notons que l'algorithme de *push–pull* [GGSC96] peut-être utilisé pour automatiser l'extrapolation de la couleur en dehors des différentes parties de l'atlas, comme cela à été fait par Sander et al. [SSGH01].

4.3.2 Filtrage et combinaison de textures explicites

Certains modèles d'habillage utilisent le placage de texture ordinaire comme élément de base et peuvent donc s'appuyer sur l'algorithme de MIP–mapping. Par exemple, la combinaison linéaire de différentes textures filtrées indépendamment produit un résultat filtré : tous les modèles reposant sur une combinaison linéaire de textures planaires peuvent donc être filtrés correctement, à condition de pouvoir estimer la taille de l'empreinte des pixels dans les textures.

4.3.3 Filtrage des textures procédurales

L'un des avantages des textures procédurales est que l'information peut théoriquement être toujours générée à la résolution de l'affichage puisque l'on dispose de formules analytiques. Il faut cependant veiller à ne pas introduire de trop hautes fréquences sinon de l'aliasing peut toujours se produire. Le filtrage n'est possible que si l'algorithme de génération permet un contrôle sur les fréquences du contenu. Certains de ces algorithmes, notamment ceux utilisant le bruit de Perlin [Per85, Per02], génèrent la texture en combinant différentes fréquences. Un filtrage possible consiste à empêcher la génération de fréquences dépassant les capacités d'affichage. Néanmoins, le filtrage analytique doit être étudié pour chaque nouvelle texture procédurale.

4.3.4 Filtrage des modèles d'habillage en général

Quelque soit le modèle d'habillage , il est essentiel que le filtrage soit correctement défini pour les apparences générées. En effet, l'aliasing est essentiellement traité, dans les applications interactives comme dans le rendu très réaliste, grâce au filtrage appliqué sur les textures (au point que le sur–échantillonnage est généralement concentré sur les arêtes des polygones [AMN03]). Ne pas traiter l'aliasing entraîne une perte de qualité et de réalisme importante, notre système visuel y étant très sensible.

Malheureusement le filtrage est parfois négligé : certaines méthodes fonctionnent bien pour produire une image fixe d'un objet relativement proche, mais ne pourraient être utilisées pour tous points de vue à cause de l'aliasing . Il faut donc proposer des solutions pour le filtrage des

modèles d'habillage (notons que Olano *et al.* [OKS03] ont récemment proposé des travaux en ce sens) et, le cas échéant, identifier les situations dans lesquelles il va s'avérer difficile ou incorrect.

4.4 Note sur la qualité de filtrage

Il est important de souligner que le filtrage est effectué avant tout pour éliminer l'aliasing , en particulier lorsque celui–ci est instable et introduit des variations dans l'image lors des déplacements du point de vue. Cependant, aucun des algorithmes de filtrage utilisés dans les applications interactives ne produit un résultat strictement correct, car aucune des hypothèses faites n'est strictement applicable partout. Par exemple, considérons le cas d'une bosse sur un terrain ayant une face rouge et une face bleue, la face bleue étant du côté visible. L'algorithme de filtrage choisi est l'algorithme de MIP-mapping. Lorsque le point de vue s'éloigne de la bosse, tout en continuant à regarder la face bleue, la couleur des pixels va glisser vers le violet. Ceci résulte du fait que la visibilité n'est pas prise en compte lors du filtrage. Cependant, bien que la couleur soit fausse, aucun aliasing temporel n'est introduit : la couleur change continûment. Il est donc relativement difficile pour l'utilisateur de repérer ces erreurs, sauf dans quelques cas pathologiques.

4.5 Limites du filtrage pour les modèles d'habillage

Filtrer les données d'habillage ne résout pas tous les problèmes ; la qualité est aussi limitée par deux autres facteurs : l'aliasing géométrique et l'interaction entre l'algorithme d'illumination locale et les données d'habillage.

4.5.1 Aliasing géométrique

L'aliasing géométrique intervient lorsque plusieurs primitives géométriques sont contenues dans un même pixel. Ce cas se produit lorsque les primitives sont petites à l'écran, en comparaison des pixels [2] ou lorsque l'arête entre deux primitives traverse un pixel. Le filtrage effectué sur les textures ne traite pas l'aliasing sur la géométrie, les primitives géométriques étant généralement prises en compte séparément. Généralement, seul le sur–échantillonnage permet de lutter contre l'aliasing géométrique, mais comme nous l'avons vu précédemment (voir section 4.2), il ne s'agit que d'une solution partielle. Notons qu'avec le rendu projectif (voir section 1.4) il est aussi possible de dessiner les arêtes des polygones dans un mode spécial afin de réduire l'aliasing des bords de polygones (mais cela ne résout pas tous les problèmes d'aliasing géométrique).

Lorsque de l'aliasing géométrique se produit entre des primitives utilisant différents modèles d'habillage, le filtrage ne peut s'effectuer correctement puisqu'il est seulement appliqué sur l'unique primitive géométrique sélectionnée dans le pixel. Pour pouvoir filtrer, il faudrait pouvoir considérer l'ensemble des primitives géométriques participant au pixel, filtrer leur habillage indépendamment, puis recombiner le résultat pour obtenir la couleur finale du pixel. Notons

[2] Situation devenue fréquente de nos jours avec l'accroissement de la puissance des processeurs graphiques.

qu'une extension du rendu projectif, l'algorithme de *A–Buffer* [Car84] permet ce filtrage. Ce n'est malheureusement pas le cas de la plupart des systèmes de rendu actuels.

Cependant, lorsque de l'aliasing géométrique se produit entre des primitives qui utilisent un habillage spatialement cohérent (par exemple une surface courbe constituée de milliers de petits triangles utilisant une même texture), le filtrage du modèle d'habillage va éliminer l'aliasing géométrique : considérer les différentes primitives géométriques revient dans ce cas à intégrer sur la zone correspondante dans l'habillage.

Il est donc primordial de ne pas découper finement la géométrie, en particulier si cela s'accompagne d'une perte de cohérence spatiale de l'habillage (par exemple dans le cas des pavages apériodiques, section 3.3). En effet, de l'aliasing géométrique risque de se produire, empêchant tout filtrage correct de l'habillage.

4.5.2 Interaction avec le modèle d'éclairage

Nous avons vu précédemment que les modèles d'habillage sont utilisés pour stocker différents attributs (couleur, normale, ...), utilisés par l'algorithme d'illumination locale pour calculer l'apparence finale de la surface (voir section 1.6). Le filtrage des données d'habillage ne peut être correctement défini que s'il est équivalent au filtrage effectué après application de l'algorithme d'illumination locale. Autrement dit, il doit y avoir équivalence entre filtrer les données puis appliquer l'algorithme d'illumination ou bien appliquer l'algorithme d'illumination *puis* filtrer les données. Si cela est souvent vrai en ce qui concerne les couleurs, cette propriété ne s'étend pas à tous les attributs. En particulier, les normales utilisées par l'algorithme de *bump mapping* [Bli78] ne peuvent pas être simplement moyennées : ceci ne produit pas le bon résultat car l'illumination résultante n'est pas une fonction linéaire de la normale (par exemple, les reflets spéculaires). Il faut donc appliquer des filtres spécifiques [Fou92]. De manière générale, tous les attributs entraînant un changement dans l'interaction entre la lumière et la surface sont difficiles à filtrer et nécessitent des algorithmes dédiés. Il reste de nombreuses études à effectuer sur ce sujet car peu de travaux se sont intéressés à cette problématique pourtant importante. Une conséquence de cela est que dans la plupart des jeux qui utilisent la technique de *bump mapping*, l'effet de relief est rapidement "aplati" par le filtrage (incorrect) lorsque les surfaces s'éloignent du point de vue.

5 Techniques de génération d'aspect de surface

Les modèles d'habillage permettent de faire varier les propriétés du matériau le long de la surface. Dans les sections précédentes, nous avons vu comment *encoder* les données. Nous allons maintenant voir comment les *générer* pour obtenir des apparences de surface réalistes.

Les méthodes de génération permettent de créer du contenu pour les modèles d'habillage, afin de donner une apparence réaliste à la surface (bois, marbre, écorce, paysage, ...). La *représentation* utilisée par le modèle d'habillage pour encoder le contenu, et la *génération* du contenu ne sont pas des problèmes distincts. Il est en effet possible d'économiser de l'espace mémoire en ne stockant pas les textures mais plutôt leur méthode de création (voir par exemple les *textures procédurales*, section 5.2.1). Le modèle d'habillage contient alors un algorithme permettant de créer l'apparence de la surface à la volée : celle–ci n'est plus explicitement créée par l'artiste. Il faut, dans ce cas, veiller à fournir aux artistes les outils leur permettant de visualiser l'aspect final de la surface et de contrôler la génération de la texture. En fait, quelque soit la représentation interne utilisée par le modèle d'habillage, l'artiste doit avoir suffisamment de contrôle et de liberté pour pouvoir générer une grande variété d'aspects de surface.

Nous présentons tout d'abord en section 5.1 les approches proposées aux artistes pour explicitement peindre une apparence sur une surface. Nous verrons ensuite des méthodes de génération automatique d'aspect de surface en section 5.2. Nous évoquerons en section 5.3 comment certaines de ces approches ont été adaptées pour générer des images utilisables dans les pavages (voir section 3.2 et section 3.3). Enfin, nous discuterons de la création d'aspects de surface animés en section 5.4.

5.1 Techniques de peinture

Une des premières approches est de permettre à l'artiste de contrôler complètement l'habillage : l'apparence est peinte point par point. On parle alors de texture *explicite*. Peindre dans une texture en compensant la distorsion et les discontinuités des paramétrisations étant difficile, des outils adaptés ont été développés pour permettre de peindre directement sur la surface.

La peinture sur surface a été introduite par Hanrahan et al. [HH90]. Dans ces travaux, la couleur est stockée par sommet et le maillage triangulaire est raffiné au fur et à mesure que l'utilisateur peint. De manière à éviter le sur–échantillonnage de la géométrie, la plupart des outils actuels utilisent une paramétrisation 2D de la surface [Dee,ZBr,Bod,Tex] et peignent directement dans la texture. Des techniques ont été développées pour permettre de dynamiquement mettre à jour la paramétrisation durant l'édition [IC01, CH04].

Si cette approche est bien adaptée à la peinture d'objets uniques de taille limitée, elle requiert très vite un temps considérable lorsqu'un grand nombre d'objets doivent être texturés, ou que les surfaces texturées deviennent grandes ou détaillées.

5.2 Synthèse de textures

Le but de la synthèse de texture est d'automatiser l'étape de création de l'aspect de surface. Un algorithme va prendre en charge la génération de l'apparence à partir de paramètres fournis

par l'utilisateur. L'apparence de la surface est alors entièrement définie par l'algorithme et ses paramètres.

Il existe deux principales modalités pour la synthèse de texture : à la volée durant le rendu, ou comme outil de création d'une image. Dans la première approche, l'apparence de la surface est générée à la volée par l'algorithme qui calcule les propriétés du matériau en un point donné. Il s'agit dans ce cas d'une forme de compression ultime : l'algorithme de génération, partie intégrante du modèle d'habillage , permet d'échanger du temps de calcul contre du stockage mémoire. L'aspect de la surface n'est plus entièrement explicité mais généré à la volée à partir de données dont la taille est petite en comparaison de la taille des surfaces à habiller. Les *textures procédurales*, présentées en section 5.2.1, sont un bon exemple de cette approche. Dans la seconde approche, l'algorithme de génération se substitue à l'artiste : il génère une texture qui est ensuite stockée explicitement, comme si elle avait été peinte par l'artiste. Dans ce cas l'algorithme de génération ne fait pas partie intégrante du modèle d'habillage . La synthèse de texture à partir d'échantillon présentée en section 5.2 et la génération d'aspect de surface par simulation d'un phénomène physique présentée section 5.2.3 suivent généralement cette seconde approche. La distinction fondamentale entre les deux approches tient au fait de pouvoir calculer l'apparence d'un point quelconque de la surface très rapidement, indépendamment des points voisins : la génération à la volée ne peut être performante que si l'algorithme de génération supporte des requêtes aléatoires (l'ordre de génération doit pouvoir être quelconque) et nécessite seulement des calculs locaux. Pour finir, notons que tout algorithme de génération à la volée peut bien entendu être utilisé pour générer, en pré–calcul, une large texture explicite.

Les enjeux de la synthèse de textures sont d'offrir une grande variété de matériaux (généricité), une bonne variation d'aspect au sein d'un même matériau (richesse) et un fort contrôle de l'utilisateur sur l'aspect final de la surface (contrôlabilité), que ce soit à grande échelle (contrôle global) ou localement sur les détails (contrôle local).

5.2.1 Textures procédurales

Les textures procédurales ont été introduite par Perlin [Per85] et Peachey [Pea85]. Elles sont parfois définies dans un volume (voir section 2.4.2.1), mais peuvent également être calculées dans le plan. Une texture procédurale permet de calculer la couleur (ou d'autres attributs) d'un matériau en un point de l'espace texture (2D ou 3D). Les textures procédurales sont maintenant utilisables dans les applications interactives, notamment grâce aux progrès techniques effectués en terme de performance et de souplesse de programmation des cartes graphiques (voir section 7).

L'approche classique est de perturber des fonctions périodiques (par exemple un sinus) par un bruit dont la fréquence est contrôlée (le bruit est généré en combinant différentes fréquences). Des opérateurs sont ensuite appliqués pour changer l'aspect résultat, comme par exemple des seuillages ou des valeurs absolues. Les nombres ainsi générés sont utilisés pour accéder une palette de couleur et générer l'apparence finale.

Les motifs réguliers (par exemple un damier) sont également très bien représentés par les textures procédurales (une simple fonction analytique dans ce cas). D'autres approches ont été développées pour permettre de représenter différentes apparences. Par exemple les textures de Worley [Wor96], basées sur les diagrammes de Voronoï, permettent de représenter les textures

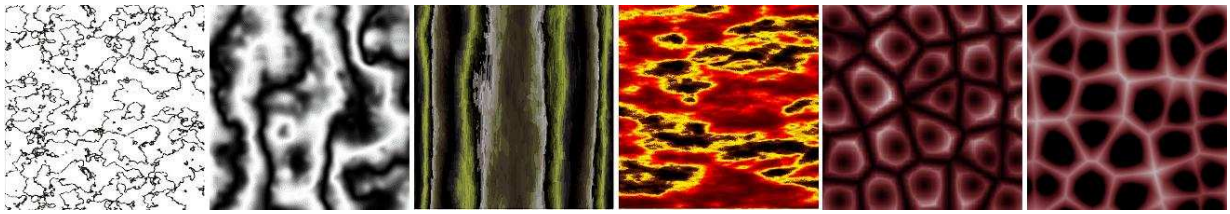


FIG. 2.30 – Différents aspects générés par des textures procédurales.

Les quatre premières textures ont été générées à partir de bruit de Perlin [Per85]. Les deux dernières textures ont été générées avec la méthode de Worley [Wor96].

basées sur des agglomérats (minéraux, ...). La figure 2.30 montre quelques aspects générés par diverses textures procédurales. Enfin, différentes méthodes peuvent facilement être combinées pour créer de nouvelles apparences [AW90].

Les principales limitations des textures procédurales sont leur manque de généricité et leur manque de contrôle local : tous les matériaux ne peuvent être représentés. Même lorsqu'un modèle de matériau existe, par exemple le marbre, il peut être délicat de trouver les paramètres définissant l'aspect final désiré (notons que des méthodes d'exploration du domaine des paramètres existent [MAB+97]). D'autre part le contrôle local est difficile : si l'artiste souhaite que certaines veines du marbre passent par des points précis, trouver l'ensemble des paramètres globaux permettant de respecter ces contraintes locales est un problème complexe.

5.2.2 Synthèse à partir d'un échantillon

La modélisation à partir d'exemples est un paradigme très plaisant pour les utilisateurs. C'est sur cette idée qu'est basée la synthèse de texture à partir d'échantillon. L'utilisateur fourni une image de petite taille, représentant la texture. L'algorithme de synthèse va dans une première phase analyser cet échantillon et construire un modèle interne de l'information qu'il contient. Par la suite l'algorithme sera capable de générer de plus larges images ayant la même apparence que l'échantillon de départ.

Les premiers travaux [GdM85, HB95, DB97, PS99] se basent sur une analyse statistique des données. Comme remarqué par DeBonnet [DB97], les données statistiques ne suffisent pas toujours à capturer l'apparence d'une texture. Il faut également en apprendre la structure, les relations entre les différentes fréquences présentes dans la texture. Des travaux plus récents [WL00, HJO+01, EF01] permettent de générer fidèlement de larges textures à partir d'échantillons pour une assez grande variété de textures. Les textures encore mal représentées sont celles contenant des structures globales et des alignements (par exemple un mur de briques). La limitation vient du fait que les textures sont supposées stationnaires et locales : la couleur d'un point ne doit dépendre que d'un petit voisinage alentour – ce qui n'est pas le cas d'une texture ayant des alignements globaux. Notons que des travaux commencent à s'intéresser à ce type de textures [LLH04]. Les textures non homogènes (donc non stationnaires) sont également difficiles pour la plupart des algorithmes.

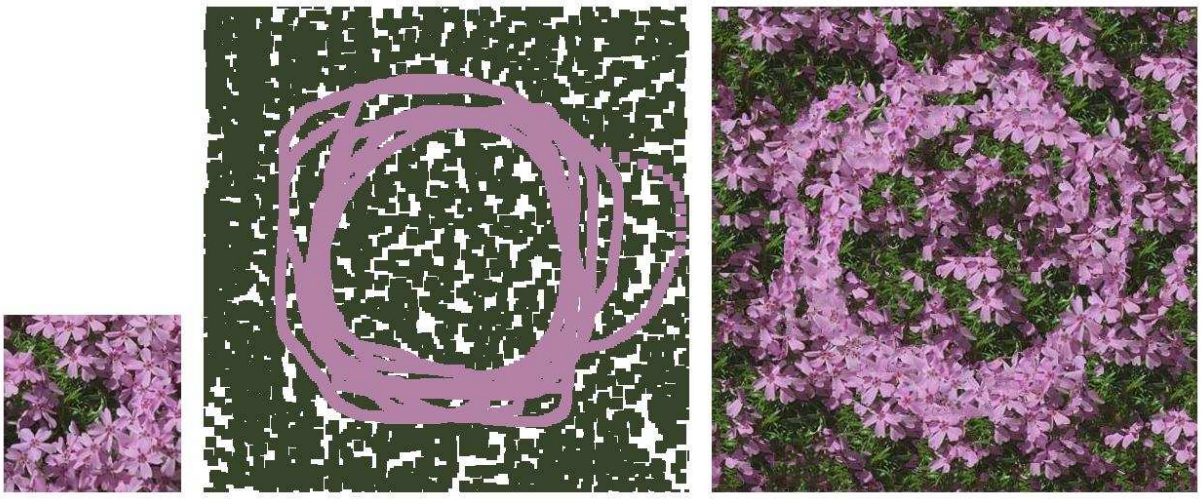


FIG. 2.31 – Synthèse de texture à partir d'échantillon contrôlée par l'utilisateur.
A gauche : échantillon utilisé. Au milieu : Texture de contrôle peinte par l'utilisateur. A droite : Texture produite par l'algorithme de Ashikhmin [Ash01].

Il existe deux principales approches pour la synthèse à partir d'échantillons : la génération séquentielle pixel après pixel [WL00] et le collage côte à côte de petits morceaux de l'échantillon [EF01]. Dans les deux cas, l'approche est globale et tout changement de paramètres nécessite de régénérer l'ensemble de la texture. Certains algorithmes permettent un contrôle de l'aspect final de la texture (répartition entre différentes zones de textures non homogènes) [Ash01, HJO+01, ZZV+03], mais, dans tous les cas, la texture doit être globalement reconstruite, même si le changement est local. La figure 2.31 présente un résultat obtenu avec la méthode de Ashikhmin [Ash01].

La synthèse de texture à partir d'échantillon est un processus trop coûteux pour permettre une génération à la volée dans les applications interactives. Certains algorithmes sont rapides [GSX00, ZG02] mais la qualité est alors très dégradée. D'autre part, les algorithmes restent souvent séquentiels ce qui empêche l'évaluation à coût constant (et faible) en un point donné de la surface. Les textures doivent donc être pré–calculées et stockées comme de larges images, ce qui gaspille beaucoup de mémoire, la texture étant uniquement constituée de petits morceaux de l'échantillon.

La plupart des algorithmes fonctionnant dans le plan ont été étendus aux surfaces courbes [Tur01, WL01]. Soit ces algorithmes génèrent un ensemble de petits éléments géométriques collés sur la surface et recouvert d'un morceau de texture [PFH00, DMLG02] (voir également section 2.3), soit ils génèrent une nouvelle géométrie finement échantillonnée et encodent la couleur en chaque sommet [Tur01, WL01]. Cette dernière approche étant peu efficace du point de vue représentation, le résultat est convertit en une géométrie simplifiée recouverte d'une texture, ce qui ré–introduit tous les problèmes des paramétrisations planaires.

5.2.3 Méthodes basées sur la simulation

Pour créer des apparences de surface réalistes, certains travaux simulent le phénomène produisant (ou altérant) l'aspect de la surface. La réaction–diffusion [WK91, Tur91] simule l'interaction entre deux réactifs sur la surface d'un objet. Ceci permet de reproduire la formation de motifs biologiques tels que les zébrures du tigre ou les taches du léopard. De plus, la simulation étant surfacique elle ne souffre pas de distorsions. D'autres travaux enrichissent l'aspect d'une surface en simulant leur vieillissement (salissure, poussière, impacts, corrosion, ...) [cHtW95, DPH96,PPD01,MDG01b,MDG01a,PPD02]. Il existe en fait un grand nombre de modèles dédiés à des apparences spécifiques : écailles [FLCB95], plumages [CXGS02], céramiques [Gob01], ... Ces modèles présentent l'avantage d'obtenir des résultats très réalistes au prix d'une perte de généricité.

Il est toutefois souvent difficile d'offrir à l'artiste un fort contrôle sur le résultat de la simulation. Idéalement, il doit lui être possible de spécifier des contraintes sur le résultat (par exemple des zones recevant plus d'impacts ou de poussière). Malheureusement il est souvent difficile de trouver les conditions initiales correspondant au résultat désiré après simulation. La simulation est d'autant plus difficile à utiliser et contrôler qu'il faut longtemps pour générer et visualiser un résultat : l'approche par essai erreur devient impossible pour l'artiste. Les récents travaux de McNamara et al. [MTPS04] sur le contrôle utilisateur d'une simulation complexe (fumée, liquide), montrent cependant qu'un fort contrôle est possible même dans le cas de simulations coûteuses. Les auteurs injectent des forces durant la simulation afin de modifier le comportement de la fumée simulée et lui faire prendre les formes désirées (définies par *keyframes*).

Finalement, la principale limitation de cette approche est que tous les matériaux ne peuvent être simulés. Les processus de formation des aspects de surface sont loin d'être tous connus, ou sont si complexes qu'il devient presque impossible de les simuler en un temps raisonnable. Les approches phénoménologiques, qui s'attachent à reproduire l'aspect visuel d'un phénomène à partir de son observation, plutôt que de simuler son processus de formation physique, permettent de représenter ces matériaux complexes en synthèse d'image. Nous présentons au chapitre 10 un modèle de création de texture pour habiller une géométrie d'arbre d'une apparence d'écorce réaliste – un cas typique de matériau difficile à simuler car mal connu et très complexe.

5.3 Méthodes respectant les contraintes des pavages

Le contenu des pavés utilisés dans les pavages, périodiques ou non, doit respecter des contraintes de bordure : les arêtes du contenu de différents pavés doivent se correspondre pour obtenir une apparence continue sur le pavage final. Certaines des techniques de génération automatique vues précédemment peuvent être adaptées pour cet objectif.

Cohen et al. [CSHD03] présentent une technique basée sur la synthèse d'image à partir d'échantillon pour générer des pavés respectant les contraintes au bord. Neyret et al. [NC99] et Stam [Sta97] montrent comment créer des pavés avec des textures procédurales. Pour les pavages périodiques, les textures sont souvent créées à partir de photographies. Un certain nombre de logiciels (Adobe Photoshop, Gimp, ...) permettent de modifier l'image pour la rendre cyclique.

Généralement, la méthode employée est basée sur des symétries et des masques de transparence (une telle technique est décrite dans [MDG00] pour générer des textures d'écorce cycliques).

5.4 Textures animées ou dynamiques

Les aspects de surface animés ou dynamiques (simulant un aspect de surface évoluant avec le temps ou réagissant aux interactions avec l'utilisateur) sont particulièrement difficiles à réaliser dans les applications interactives. Dans les deux cas, l'aspect de la surface est contrôlé par un algorithme d'animation ou de simulation à chaque pas de temps. Ceci correspond aux textures procédurales en ce sens que l'aspect est généré lors de l'affichage, mais en plus cet aspect est régénéré à chaque pas de temps, éventuellement en fonction du pas de temps précédent (dynamique).

Différents types d'animations existent :

- Un aspect de surface globalement en évolution (surface d'eau, écoulements de liquide, fissures, ...). Des modèles de simulation ont été proposés pour créer des aspects de surface dynamiques [HCSL02], mais ils fonctionnent dans un volume ou un espace 2D non distordu et ne sont pas compatibles avec la paramétrisation d'une surface quelconque. Ce type de textures animées commence à apparaître dans les jeux vidéos, par exemple pour simuler la surface de l'eau dans les jeux *Morrowind* de *Bethesda Softworks* (2002), et *Virtual Skipper 3* de *Nadéo* (2003). Cependant leur utilisation reste limitée à quelques cas simples à cause de la difficulté à gérer correctement la simulation sur les surfaces.
- De petits éléments apparaissant et se déplaçant sur la surface (gouttes d'eau, feuilles mortes, traces de pas, impacts, ...), ou réagissant aux déformations géométriques (écailles coulissantes, côte de maille, ...). Ces aspects peuvent être représentés par des techniques à base de motifs, comme les particules de textures (présentées section 2.3). Cependant nous avons vu qu'il n'y a pas de modèle d'habillage les gérant de manière satisfaisante.

Les apparences animées sont aussi difficiles à créer du point de vue de l'artiste puisqu'il ne visualise pas directement l'aspect final de la surface et que des défauts visuels non maîtrisés peuvent apparaître (distorsions, géométrie additionnelle mal positionnée, ...). Nous proposons dans le cadre de cette thèse différents modèles d'habillage capables de générer efficacement des apparences de surface animées (voir chapitres 3, 4, 7 et 9), que ce soit en terme de facilité de création, d'efficacité de stockage ou de vitesse d'affichage.

6 Modèles d'habillage et représentations alternatives

Il existe d'autres approches que les modèles polygonaux pour représenter les formes. Ces *représentations alternatives* permettent de mieux capturer l'apparence de certains types d'objets, en particulier les objets à la géométrie fine et complexe (arbres, chevelures) ou les objets non tangibles tels que la fumée ou les nuages. Il nous faut alors revoir la façon d'associer l'habillage à la forme, mais on notera que, dans bien des cas, l'habillage est souvent lui–même une partie de la définition de la forme de l'objet.

6.1 Textures Volumiques

Les textures volumiques [KK89, Ney96] encodent les formes à l'aide de grilles de densité tridimensionnelles. Le volume est affiché soit par lancer de rayon, soit en superposant des polygones transparents texturés sur l'écran [LL94, WE98, MN98a]. Ces méthodes permettent de représenter efficacement des couches de matière complexes mais peu denses recouvrant une surface, telles que fourrures et forêts [DN04]. Les effets de parallaxe sont très bien capturés. Elles sont également très efficaces pour représenter des formes non tangibles telles que les nuages et la fumée. Cependant, le principal défaut de la méthode est de difficilement représenter les surfaces dures : les contours des silhouettes d'objets solides manquent parfois de netteté. Notons que la plupart des modèles d'habillage que nous proposons dans le cadre de cette thèse peuvent être étendus pour définir des textures volumiques (les polygones transparents forment la surface support) et ne sont donc pas incompatibles avec cette approche.

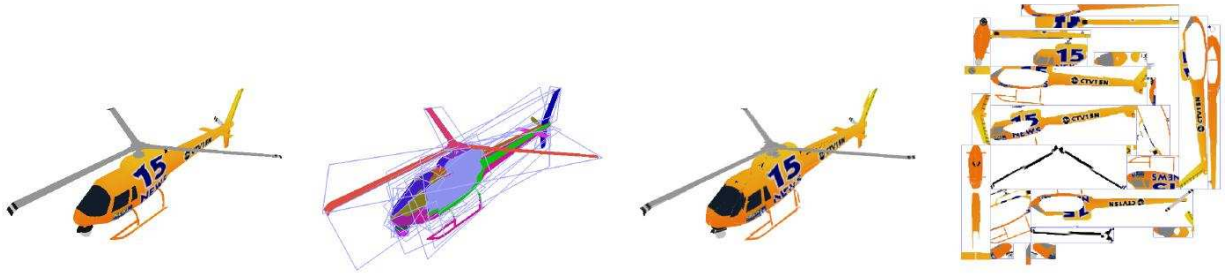
6.2 Imposteurs

Les techniques d'imposteur représentent des objets complexes par des formes plus simples supportant un modèle d'habillage . L'approche est de donner l'illusion de forme sans essayer de représenter les objets, mais plutôt en reproduisant la perception que l'on en a.

La première technique, très utilisée dans les simulateurs, notamment pour représenter les arbres, est connue sous le nom de *billboards*. Il s'agit de simples rectangles supportant une texture. Si l'illusion est bonne pour des points de vue éloignés, le manque de parallaxe devient évident dès que l'on se rapproche. Cette méthode a été étendue par les travaux de Décoret [DDSD03] qui représente les formes complexes par un enchevêtrement de billboards (voir figure 2.32). La parallaxe est bien mieux capturée et le ré-éclairage est possible grâce à la technique de *bump mapping* [Bli78].

Les textures de relief (*relief textures*) introduites par Oliveira et al. [OBM00] représentent les objets à l'aide de cartes de profondeurs appliquées sur les faces d'un cube englobant la forme. Les points des faces sont re–projetés en fonction du point de vue, de manière à former l'objet.

Ces méthodes ne sont pas des modèles d'habillage de surface : la surface sur laquelle le modèle est appliqué (le ou les rectangles, les faces du cube) ne correspond pas à la géométrie que l'on souhaite visualiser. Il ne s'agit pas d'un ajout de détail, mais véritablement d'une représentation alternative de la forme d'un objet.



(a) (b) (c) (d)

FIG. 2.32 – Imposteurs : un enchevêtrement de plans texturés représente l'objet.

Les plans sont positionnés pour représenter au mieux la forme de l'objet [DDSD03]. (a) Objet original (b) Enchevêtrement de plans (c) Objet représenté l'enchevêtrement de plans (d) Texture appliquée sur les plans.

6.3 Rendu par point

Le rendu par point, plutôt que d'encoder la densité dans des grilles volumiques régulières, représente les formes à l'aide de nuages de "points" [LW85, PZvBG00, RL00]. Cette approche permet d'afficher efficacement de gros volumes de données et est particulièrement efficace sur de grands ensembles d'objets complexes tels que les forêts. En outre, le filtrage "géométrique" est bien défini sur ces représentations. La méthode n'est pas incompatible avec les modèles d'habillage puisque les points (ou disques) peuvent être texturés [GP03]. Le rendu par point représente relativement mal les larges surfaces texturées, et des modèles hybrides, mélangeant point et polygones sont logiquement apparus [ZSBP02]. Les modèles d'habillage , potentiellement compatibles avec les deux approches, permettent d'utiliser une même représentation pour les aspects de surface.

7 Concevoir des modèles d'habillage pour les applications interactives

7.1 Cartes accélératrices

Les processeurs graphiques (GPU) des cartes accélératrices récentes offrent une grande souplesse de programmation. Ils permettent de directement implémenter sur la carte graphique des modèles d'habillage complexes et de les utiliser dans les applications interactives. La figure 2.33 présente une version simplifiée de la chaîne de traitement (*pipeline*) d'une carte graphique programmable, ainsi que le flot de données et les parties programmables d'un processeur graphique.

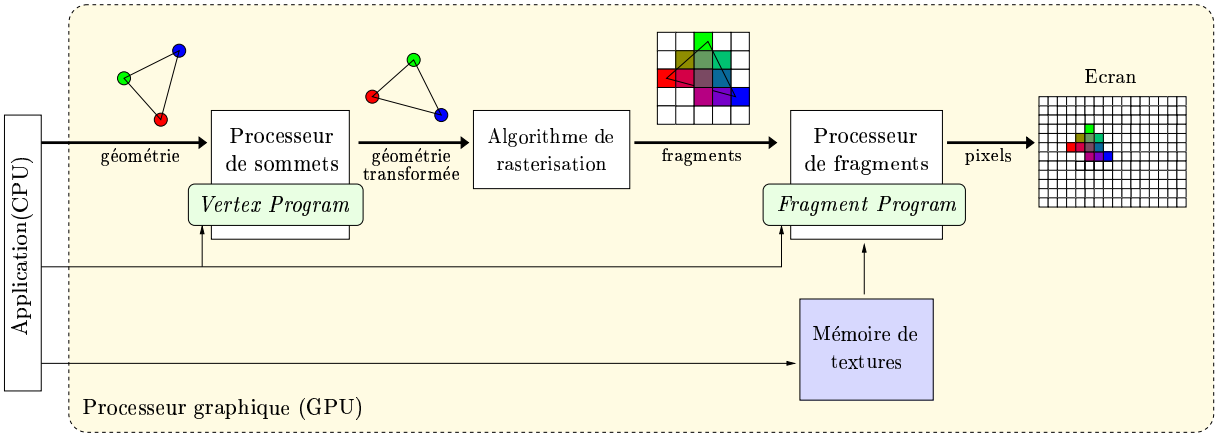


FIG. 2.33 – Chaîne des traitements (*pipeline*) d'un processeur graphique (version simplifiée).

Le processeur graphique reçoit les données envoyées par l'application exécutée sur le CPU (processeur standard), via le driver *(interface logicielle entre l'API graphique – OpenGL ou Direct3D – et la carte graphique). Lorsque l'application envoie de la géométrie, celle–ci est tout d'abord traitée par le* processeur de sommets*. Ce processeur est programmable via un* vertex program*. Il permet d'effectuer des opérations sur les sommets de la géométrie et est notamment en charge de la projection des points à l'écran. Ensuite la géométrie est envoyée à l'algorithme de rasterisation (voir section 1.4). La géométrie vectorielle est alors convertie en un ensemble de fragments. Les données disponibles en chaque sommet (couleur, normales, coordonnées de texture, ...) sont automatiquement interpolées. Les fragments sont ensuite traités par le* processeur de fragments*. Ce processeur est programmable via un* fragment program*. Il permet de calculer la couleur finale du fragment à partir des informations interpolées depuis les sommets et des informations stockées dans la mémoire texture. Si le fragment est accepté (test de profondeur –* Z–buffer*) il sera finalement affiché sur le pixel de l'écran correspondant.*

Si les processeurs graphiques sont programmables, il s'agit cependant d'un matériel spécifique, au comportement très différent des processeurs auxquels nous sommes habitués. Lors de la conception de modèles d'habillage il faut prendre soin de tenir compte de ce contexte. Ceci est d'autant plus vrai que des langages de programmation de plus en plus haut niveau apparaissent (langages de haut niveau *Sh* [MTP+04], *Brooks* [BFH+04], *Cg* [MGAK03], *RTSL* [PMTH01], *Ashli* (ATI), et langages d'API *GLSL* pour OpenGL et *HLSL* pour DirectX). Si ceci est vital

pour permettre un niveau d'abstraction suffisant et donc s'affranchir des spécificités de chaque constructeur et API (OpenGL / DirectX), il faut cependant tenir compte des particularités de l'architecture des processeurs graphiques si l'on veut les exploiter correctement :

- La précision des données stockées doit être minimale, et ce pour deux raisons. En premier lieu, augmenter la précision nécessite d'augmenter aussi le stockage. Les textures étant souvent de taille conséquente, il est vite possible de saturer la mémoire disponible. En second lieu, accéder aux données coûte cher. En fait, sur les dernières cartes disponibles l'accès est plus coûteux que les calculs [FSH04]. Minimiser la précision réduit la bande passante et les défauts de cache, et donc les temps d'accès. Réduire la précision ne peut se faire trivialement sans introduire de nombreux défauts visuels. Il s'agit donc de concevoir les algorithmes afin qu'ils puissent produire des images de qualité malgré une faible précision de stockage (généralement 8 bits). La précision de calcul, quant à elle, est généralement de 32 bits (nombre à virgule flottante).
- L'accès aux données de texture entraîne un temps de latence. Il est cependant possible d'entrelacer calculs arithmétiques et accès texture afin que les calculs soient effectués en parallèle (à condition, bien entendu, que les calculs n'utilisent pas les données lues dans la texture). Ce type d'approche peut impliquer un gain de performance conséquent, mais ici aussi, il faut concevoir l'algorithme et les structures de données de manière à la rendre possible.
- L'architecture des processeurs graphiques est conçue pour traiter de larges flots de données unidirectionnels. Les informations de géométrie, texture et éclairage sont envoyées vers la carte qui doit alors produire une image sur l'écran. Récupérer l'image produite pour traitement par le CPU est malheureusement inefficace. Il ne s'agit pas uniquement d'une question de bande passante, mais véritablement d'un problème d'architecture : relire les données impose à la carte de finir les opérations en cours. Ceci a un peu le même effet qu'un saut de pointeur d'instruction sur un processeur : le pipeline de données est rompu. Il faut donc tenter de minimiser, ou mieux d'éviter, ce type d'opération. C'est pour cette raison que la plupart des travaux tentent de déporter l'ensemble d'un algorithme vers le GPU, d'autant que les transfert CPU vers GPU sont également assez lents.
- Les modèles d'habillage sont généralement implémentés avec les *fragment program*, exécutés en chaque pixel (voir figure 2.33). Si les programmes pouvant être écrits peuvent sembler génériques (sauts conditionnels et boucles sont disponibles), il est important de se rappeler que les cartes accélératrices traitent les pixels en parallèle. Ceci impose de fortes contraintes sur l'exécution et les performances des programmes écrits. Par exemple, un saut conditionnel ne sera efficace que si un large bloc de pixels (de l'ordre de 16×16) effectuent le même branchement. Dans le cas contraire, le traitement parallèle devra se synchroniser sur le pixel ayant effectué le plus long traitement.

Ne pas tenir compte de ces éléments résultera en de mauvaises performances. L'intérêt d'utiliser des cartes graphiques réside dans l'exploitation de leur architecture massivement parallèle et de leur capacité à traiter de grandes quantités de données. Par exemple, une carte graphique est très efficace pour effectuer un traitement simple sur l'ensemble des cases d'un tableau ; opération généralement assez lente sur un processeur standard. L'accélération obtenue peut être

conséquente ; de plusieurs ordres de grandeur. Il s'agit donc de concevoir des algorithmes adaptés à cette architecture en tenant compte de ses spécificités.

7.2 Géométrie et applications interactives

La géométrie est généralement stockée sous une forme statique directement dans la mémoire accessible par le processeur graphique. Ceci permet de meilleures performances car la géométrie n'a plus besoin d'être transférée du CPU vers le GPU à chaque image. Accéder ou modifier la géométrie stockée sur le GPU depuis le CPU est une opération lente. De plus, la géométrie est généralement optimisée pour tirer profit des caches accélérant l'accès mémoire. Ceci permet des gains de performance considérables. Il reste néanmoins possible d'animer cette géométrie en utilisant des *vertex program*, qui sont exécutés par le GPU en chaque sommet et permettent de modifier leurs positions et leurs attributs à la volée.

Cependant, il n'est pas actuellement possible de subdiviser ou de créer de nouvelle géométrie durant le rendu. Toute modification de topologie nécessite de renvoyer les informations du CPU vers le GPU, opération relativement coûteuse. Il faut donc tenter de ne jamais avoir à créer dynamiquement de nouvelle géométrie afin d'obtenir des performances optimales.

8 Résumé et conclusion

Nous avons vu, tout au long de ce chapitre, que les modèles d'habillage actuels présentent un certain nombre de limitations :

- **Gaspillage de mémoire**. Qu'il soit dû à la difficulté à paramétrer des géométries complexes (voir la section 2.1.6), ou à une représentation mal adaptée d'un aspect de surface donné (voir la section 2.3), de nombreuses situations provoquent un gaspillage de mémoire. Ceci se combinant à l'exigence toujours croissante de haute résolution, l'espace mémoire disponible devient rapidement un facteur limitant lorsque l'on souhaite afficher des scènes riches en détails.
- **Défauts visuels**. Afin d'éviter les distorsions lors de la paramétrisation planaire des surfaces, la plupart des algorithmes font le choix d'introduire des discontinuités dans les textures et la géométrie. Malheureusement, ce choix contredit certaines des hypothèses utilisées par les algorithmes de filtrage et provoque des défauts visuels dans les images finales.
- **Manque de variété, aspects irréalistes**. Les aspects de surface réalistes sont difficiles à créer. Reproduire la grande variété d'apparences observables dans la nature, la richesse et la complexité des détails, et ceci sur de larges surfaces, exige beaucoup de temps aux artistes. Ceci, combiné aux limitations de mémoire évoquées plus haut, implique d'utiliser des raccourcis, par exemple en ayant recours aux pavages (voir sections 3.2 à 3.4) qui répètent une même apparence le long des surfaces. Ces méthodes souffrent de problèmes d'alignements visibles et de manque de variété. De plus, les outils générant automatiquement des apparences pour les surfaces ne sont pas toujours adaptés aux modèles d'habillage efficaces et, lors de la conversion, de nombreux défauts visuels peuvent apparaître (voir section 5.2.2). En outre, il n'existe pas d'algorithme de génération pour tous les aspects de surface.
- **Dépendance entre modèles de forme et modèles d'habillage** Dans certaines situations, il est actuellement nécessaire de modifier la représentation de la forme en fonction du modèle d'habillage (en ajoutant ou modifiant les polygones). Ceci a de nombreuses conséquences néfastes : que ce soit pour séparer un modèle en différentes parties pour le paramétrer ou, pire, pour pouvoir introduire des variations d'aspect à relativement petite échelle comme dans le cas des pavages (voir section 3.3) ou des particules de textures (voir section 2.3), modifier la forme en fonction de l'apparence de surface brise l'indépendance entre ces deux problèmes distincts. Les détails alourdissent ou empêchent les algorithmes manipulant les formes (animation, simulation d'objet déformables, visibilité, niveaux de détails) sans aucune raison d'ordre géométrique. Modifier la géométrie peut ralentir considérablement l'affichage (voir section 7.2) et provoquer de l'aliasing (voir section 4.5) ce qui nuit considérablement aux applications interactives.

Nous avons présenté des travaux récents qui permettent d'alléger certaines limitations des modèles d'habillage actuels : les techniques de compression limitent le gaspillage mémoire. Cependant, la compression est souvent destructive et se fait au détriment de la qualité, tout en ne permettant pas toujours de gagner d'ordre de grandeur sur la taille des données. Des structures de données plus complexes, comme les cartes d'environnement généralisées (voir section 2.2.2),

les textures adaptatives (section 2.1.7.2) ou les *octree textures* (voir section 2.4) apportent des solutions. Toutefois, elles ne sont pas toujours efficaces (vitesse de rendu) ni exemptes de défauts visuels (en particulier filtrage et interpolation sont souvent incorrects ou inefficaces).

Nous nous concentrerons donc sur un triple objectif :

- En premier lieu, nous allons créer des modèles d'habillage permettant de dépasser ces limitations dans les situations favorables, en particulier pour la large classe des textures à base de motifs, particules de texture et pavages apériodiques (voir chapitres 3 et 9).
- En second lieu, nous allons améliorer les modèles existants, notamment en réduisant le gaspillage mémoire dans le cas général du placage de texture de haute résolution sur des surfaces quelconques (voir chapitre 6), et en rendant disponibles dans les applications interactives des modèles reposant sur des structures de données plus complexes, en particulier les textures volumiques hiérarchiques (voir chapitre 7).
- Enfin, nous proposerons des méthodes pour représenter des aspects de surface difficiles à créer pour les artistes et à simuler car mal connus, avec notamment la création de textures représentant des phénomènes animés (goutte d'eau, voir chapitre 4 ; liquide à la surface d'un objet, voir chapitre 8) et un modèle pour la création de textures réalistes d'écorces (voir chapitre 10).

Pour chacun des modèles d'habillage proposés, nous garderons à l'esprit les grands principes que sont le contrôle de l'artiste sur le résultat final, la légèreté (en coût mémoire) et l'efficacité (en coût calcul) des représentations qui, outre leur utilisation dans des applications interactives, leur permettra de supporter les quantités de données extrêmes utilisées pour le rendu réaliste.

Deuxième partie

Habillage à base de motifs pour terrains et surfaces paramétrées

chapitre 3

Textures procédurales à base de motifs

Habiller les terrains avec des apparences réalistes et riches en détails, qui peuvent être vues de très près comme de loin, est une tâche difficile. En effet, même si les terrains peuvent être facilement paramétrés (voir chapitre 2, section 2.1.2.1), le placage de texture est difficile à utiliser directement sur de si grandes surfaces (voir chapitre 2, section 3) : leur taille rend longue et difficile la création des textures par les artistes, l'énorme quantité de détails entraîne un coût mémoire conséquent. En outre, un terrain comporte à la fois des matériaux relativement homogènes (sable, roc, terre, herbe, ...) mais également de nombreux objets similaires répartis sur la surface (fleurs, feuilles, pierres, ...). Encoder ensemble ces détails de nature très différente dans une même texture entraîne un gaspillage de mémoire conséquent (voir chapitre 2, section 2.3).

Afin de réduire le coût mémoire, les jeux vidéos et les simulateurs reposent souvent sur la répétition d'un ensemble de motifs le long du terrain (voir chapitre 2, section 3.3). L'idée sous–jacente est que chaque motif est à haute résolution et stocké une seule fois en mémoire. Les motifs sont alors positionnés par les artistes dans une grille régulière, sur le terrain, à l'aide de géométrie supplémentaire. Cependant, cette approche entraîne un surcoût géométrique important, et des alignements peuvent toujours apparaître à grande échelle (voir chapitre 2, section 3.3), notamment à cause d'objets très visibles dessinés dans les motifs (une fleur rouge sur fond d'herbe).

Nous proposons dans ce chapitre d'étendre l'idée de textures à base de motifs à un cadre plus général. Nous souhaitons bénéficier des avantages, en terme de coût mémoire et de haute résolution, de la répétition d'un ensemble de motifs ; mais sans souffrir des problèmes qui y sont généralement associés : dépendance à la géométrie, positionnement individuel de chaque motif par un artiste, et alignements visibles à grande échelle. Notre méthode est très générale et permet de définir une grande variété de textures à base de motifs, allant de pavages apériodiques pour les matériaux relativement homogènes (herbe, sable), à une distribution aléatoire de motifs sur la surface pour les objets distinguables (fleurs, feuilles mortes, pierres), en passant par des textures composées de motifs animés.

Au delà de sa généricité, une contribution importante de notre méthode est que l'agencement et le choix des motifs peut être laissé à un algorithme procédural. L'artiste garde cependant un fort contrôle sur la distribution spatiale à grande échelle des motifs (contrôle global). Ceci permet par exemple de décider quelle zone est recouverte d'herbe, quelle autre est recouverte

de sable, et de laisser l'algorithme procédural s'occuper de générer des variations d'apparences pour chacun des deux matériaux. Si l'utilisateur le souhaite, il lui reste néanmoins possible de contrôler explicitement la position précise de certains motifs (contrôle local).

Il n'est donc plus nécessaire ni de consacrer du temps à l'agencement des motifs, ni de stocker cette information, celle–ci étant générée à la volée par notre algorithme procédural lors de l'accès à la texture. La mémoire requise est donc principalement déterminée par les motifs de base utilisés, qui peuvent être à très haute résolution. Notre méthode, qui fonctionne dans l'espace texture, permet de ne pas introduire de contraintes sur le maillage triangulaire de la surface, qui reste inchangé. Notre approche est conçue pour les cartes graphiques récentes, mais les outils de rendu logiciel peuvent également en bénéficier pour générer de larges textures sans répétitions de grande échelle et à très faible coût mémoire.

Ce chapitre est structuré de la manière suivante : dans la section 1 nous décrivons le principe de notre approche et ses composants de base. Nous donnons des détails d'implémentation en section 2. Nous montrons en section 3 comment ces blocs de base peuvent être combinés pour créer différents types de textures. Nous présentons les résultats en section 5.

Remarque : Les travaux présentés dans ce chapitre ont fait l'objet d'une publication à la conférence I3D (*ACM SIGGRAPH Symposium on Interactive 3D Graphics*) en 2003, sous le titre *Pattern Based Procedural Textures* [LN03].

1 Notre approche : Textures procédurales à base de motifs

Avec notre système, un utilisateur compose de très larges textures en positionnant des instances de motifs de base. Ce positionnement est soit explicite, soit procédural. Les motifs sont stockés sous forme de petites images. L'utilisateur fournit donc un ensemble de motifs de base, choisi la taille de la texture virtuelle (car jamais stockée) à générer, décrit le type de combinaisons de motifs souhaitées et fournit les différentes cartes et paramètres nécessaires pour contrôler ces combinaisons.

Le principe de notre algorithme est de choisir un motif correspondant à une coordonnée de texture u, v à la volée, durant l'exécution. Nous divisons l'espace texture en une grille virtuelle. A partir d'une coordonnée de texture u, v, nous trouvons la case correspondante dans cette grille. Nous choisissons alors un motif pour cette case selon divers paramètres (voir figure 3.1). Une fois le motif sélectionné, nous pouvons utiliser les coordonnées du point u, v à l'intérieur de la case pour trouver la couleur finale.

Les motifs n'ont pas à être alignés sur la grille : en transformant les coordonnées u, v à l'intérieur d'une case, il est possible d'appliquer une translation, une rotation ou une mise à l'échelle des motifs. Tous ces calculs peuvent être effectués dans la carte graphique en utilisant un *fragment program* qui calcule la couleur finale des pixels à partir des coordonnées de texture (voir chapitre 2, section 7).

Selon l'application, différents paramètres vont influencer le choix et le positionnement des motifs : type de matériau, distribution spatiale, temps, distance à un point, etc ... Les très larges

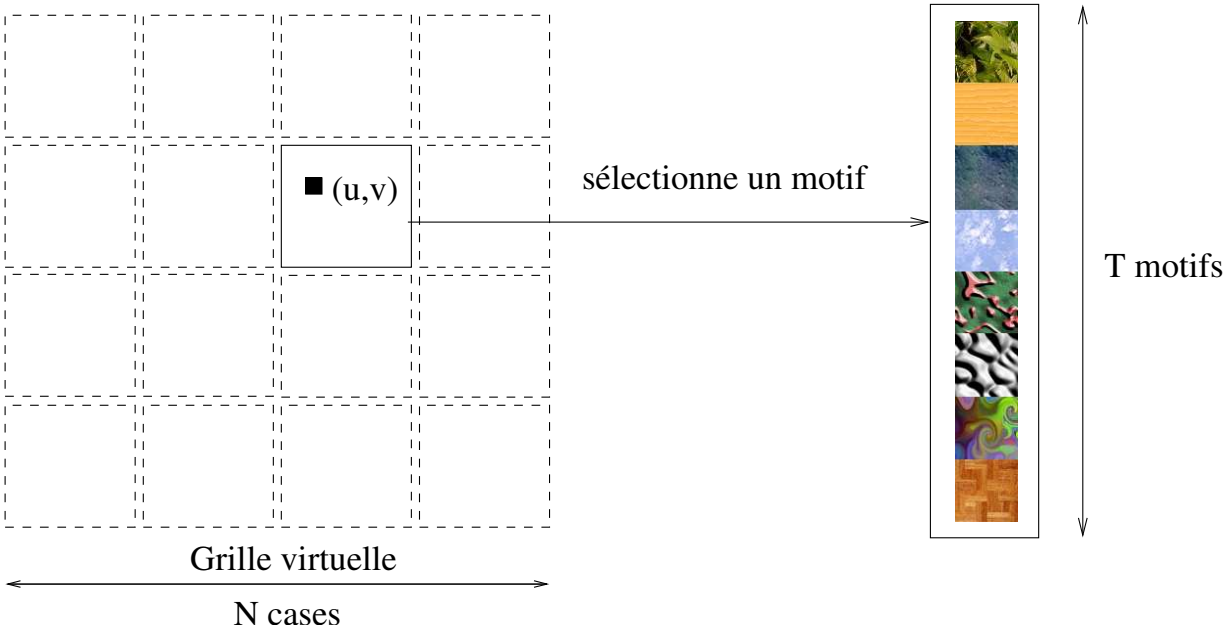


FIG. 3.1 – Textures procédurales à base de motifs

Un motif est choisi à la volée pour la case contenant les coordonnées de texture u, v.

textures ainsi générées ne sont donc jamais stockées : elles sont uniquement évaluées en chaque pixels, lors de l'affichage. De plus, tous les calculs étant effectués dans l'espace texture, la géométrie n'a pas besoin d'être modifiée.
Les opérations nécessaires pour créer une large variété de textures à l'aide d'une combinaison de motifs peuvent être classées en trois catégories :

- **Choix et positionnement des motifs** (section 1.3) : Le choix des motifs doit être effectué soit explicitement (par l'artiste), soit de manière procédurale sans montrer de périodicité. Un contrôle local sur la distribution spatiale des différents motifs est nécessaire pour créer des textures avec un aspect riche à partir d'un petit nombre d'éléments. Une autre caractéristique importante est d'avoir la possibilité de déplacer, tourner ou redimensionner les motifs à l'intérieur d'une case de la grille virtuelle, et ce afin de briser la régularité introduite par la structure de la grille.
- **Transitions** (section 1.4) : Les pavages de motifs carrés (ou *pavés*) sont largement utilisés dans les jeux vidéos pour texturer de larges zones à faible coût mémoire. Afin de couvrir différentes zones avec différents matériaux (herbe, sable, etc ...), il est nécessaire de pouvoir spécifier des transitions (voir chapitre 2, section 3.5.2).
- **Animation** (section 1.5) : Les motifs peuvent être utilisés pour créer des textures animées. Chaque motif est alors remplacé par une séquence animée (une fleur s'ouvrant, une lampe clignotante, etc ...). Le but est ici de pouvoir animer chaque motif de manière apériodique et asynchrone par rapport à ses voisins (par exemple un champ de fleurs oscillant sous le vent).

Afin de ne pas enfermer l'utilisateur dans un nombre restreint d'opérations sur les motifs, nous proposons une méthode générique basée sur un ensemble de composants de base (de manière similaire à [AW90]). Chaque composant est un petit algorithme indépendant en charge d'une fonctionnalité particulière (apériodicité, transitions, animation, etc ...). Afin de concevoir une nouvelle texture procédurale à base de motifs, l'utilisateur connecte ces composants ensemble de manière à obtenir une couleur à partir d'une coordonnée de texture u, v. Nous montrons section 3 comment ces différents composants peuvent être combinés pour créer des *fragment programs* capables, par exemple, de générer des pavages apériodiques ou de positionner des motifs (par exemples des feuilles) sur le terrain tout en respectant une distribution spatiale fournie par l'utilisateur.

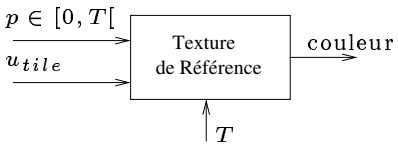
1.1 Cartes et textures

Afin de mieux différencier les textures stockant différentes informations, nous appellerons *textures* les textures qui encodent des couleurs et *cartes* les textures qui encodent d'autres paramètres. Une carte (resp. texture) *explicite* est une carte (resp. texture) qui est stockée en mémoire. Une carte (resp. texture) *virtuelle* est une carte (resp. texture) qui est générée à la volée. Une carte de taille N est une fonction de $[0,N[$ dans $[0,M[$, où M est la valeur maximale qui peut être stockée dans la carte. Le choix de M est déterminé par l'utilisation de la carte. L'accès aux cartes est cyclique : pour une carte de taille N, $carte(x+N) = carte(x)$.

1.2 Combiner les composants

Nos composants de base peuvent être facilement combinés : les paramètres d'entrée peuvent utiliser les sorties d'autres composants ; toute entrée peut utiliser une carte virtuelle ou explicite si un contrôle utilisateur supplémentaire est nécessaire. Les sections suivantes détaillent les composants de base. A côté du nom de chaque composant un petit schéma indique ses entrées et sorties. Nous proposons également quelques composants utilitaires formés à partir de composants de base. Pour des raisons de clarté, nous expliquons notre modèle dans le cas 1D, sauf mention contraire. L'extension aux cas 2D et 3D est triviale : il suffit d'étendre les opérations scalaires utilisées dans le cas 1D aux vecteurs.

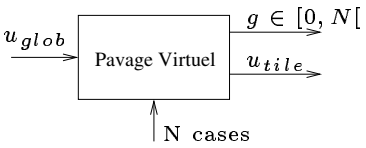
1.3 Composants pour le choix et le positionnement des motifs



Texture de référence

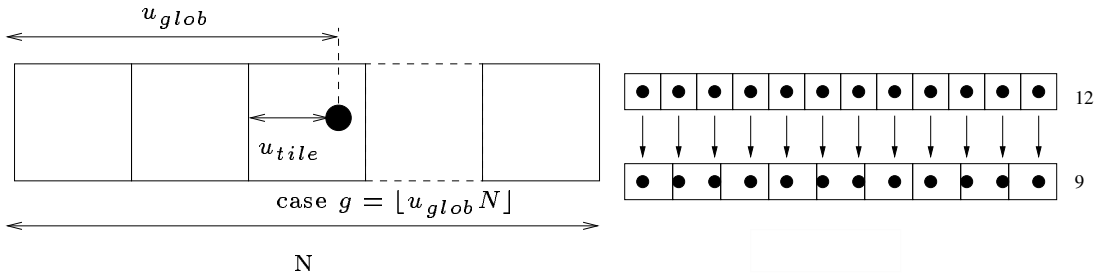
Le but de ce composant est de simplifier la gestion des différents motifs. Au lieu de stocker chaque motif indépendamment dans des textures séparées, l'utilisateur fournit une seule texture dans laquelle les motifs sont stockés côte à côte. Le nombre de motifs stockés est noté T. Nous

supposons les motifs de même résolution (en pixels). Pour un index de motif p et une coordonnée de texture relative $u_{tile} \in [0,1]$ définie dans la case d'un motif, ce composant retourne simplement la couleur du pixel désigné par u_{tile} dans le motif désigné par p.



Pavage virtuel (see Figure 3.2(a))

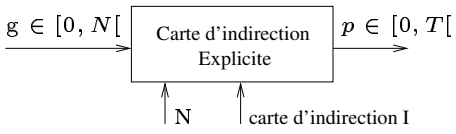
Un *Pavage virtuel* de taille N est une grille virtuelle de N cases (N^2 cases en 2D) qui recouvre entièrement l'espace texture (voir figure 3.1). u_{glob} est la coordonnée de texture qui provient directement du maillage triangulaire. Ce composant calcule quelle case de la grille virtuelle contient la coordonnée u_{glob}. Il calcule également la coordonnée relative u_{tile} du point à l'intérieur de la case. Si la grille virtuelle à une taille de N, alors le point est dans la case $g = \lfloor u_{glob} N \rfloor$ et ses coordonnées relatives sont $u_{tile} = frac(u_{glob} N)$ où $frac(x)$ désigne la partie fractionnaire de x.



(a) Le composant *Pavage virtuel* calcule la position du point aux coordonnées u_{glob} dans la grille virtuelle de taille N.

(b) Avec T=12 et T'=9 de l'aliasing apparaît produisant un biais dans la distribution.

FIG. 3.2 – (a) Pavage virtuel et (b) aliasing dans les cartes aléatoires virtuelles



Carte d'indirection explicite

Ce composant permet à l'utilisateur de choisir explicitement quel motif doit apparaître dans chaque case de la grille du pavage virtuel. Soit I une *carte d'indirection* de taille N définie explicitement par l'utilisateur. Une valeur dans I encode l'indice p du motif qui doit être utilisé dans une case de la grille. Le motif p choisi dans une case g est donc simplement $p = I(g)$.

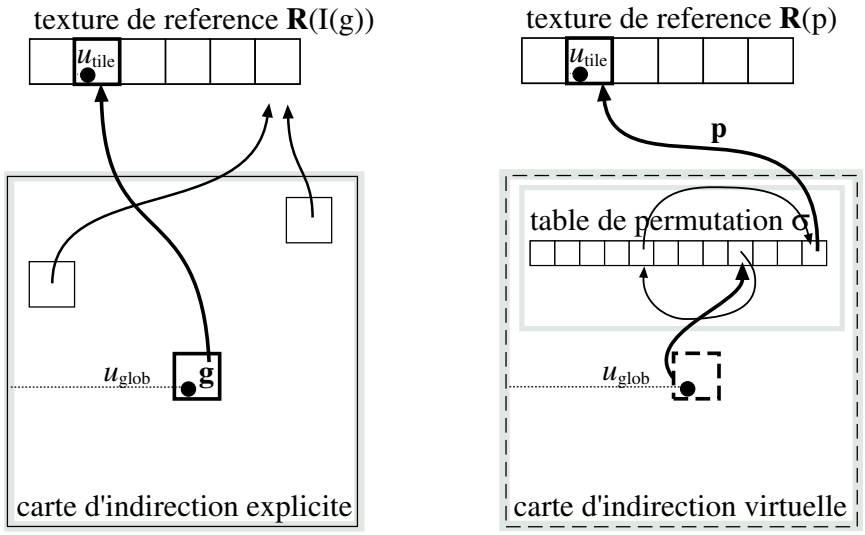
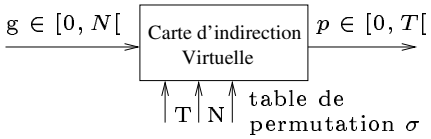


FIG. 3.3 – Cartes d'indirections
A gauche : Carte d'indirection explicite *indexant la* Texture de Reférence. A droite : Carte d'indirection virtuelle *les numéros des motifs sont calculées avec une fonction de hashage appliquée aux indices des cases. La permutation* σ *est utilisée pour générer des nombres aléatoires.*



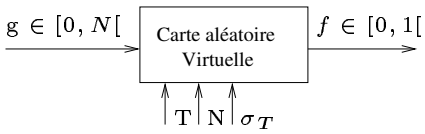
Carte d'indirection virtuelle

Afin de créer de larges textures à partir de motifs sans faire apparaître de répétitions évidentes, nous devons pouvoir choisir les motifs de manière apériodique (voir également chapitre 2, section 3.3). C'est l'objectif de ce composant. Un numéro de motif p est aléatoirement généré à partir d'un indice de case g. Ce numéro de motif peut ensuite être utilisé avec la texture de référence présentée plus haut. Un pavage apériodique est ainsi crée puisqu'un motif est aléatoirement associé à chaque case de la grille du pavage virtuel : le choix des motifs ne montre pas de répétitions (voir figure 3.10).

La taille de la grille du pavage virtuel N et le nombre de motifs T sont donnés en paramètre. Afin de calculer un numéro de motif aléatoire p (constant dans une même case g) nous utilisons un générateur de nombres pseudo–aléatoires obtenu à partir d'une fonction de hashage σ. La fonction de hashage est appliquée à l'indice de case g. Nous utilisons une table σ de taille T qui encode une permutation d'indices entre 0 et $T-1$. Le nouvel indice de i est donné par $\sigma(i)$.

Afin de générer des nombres aléatoires compris entre 0 et T à partir d'indices plus grands que T, nous utilisons une approche similaire à celle utilisée par Perlin [Per85]. Dans ses travaux, un nombre aléatoire est généré à partir de coordonnées 2D (x,y) en appliquant $\sigma(x+\sigma(y))$. Ceci permet d'éviter les corrélations entre x et y. Dans notre cas nous évaluons la série $s_0 = \sigma(g)$, $s_i = \sigma(\frac{g}{T^i} + s_{i-1})$ jusqu'à ce que $T^{i+1} \geq N$. En pratique nous utilisons seulement 2 ou

3 étapes, en utilisant par exemple $\sigma\left(\frac{g}{T^2}+\sigma\left(\frac{g}{T}+\sigma(g)\right)\right)$. Ceci fourni un pavage apériodique de T motifs dans la grille du pavage virtuel de taille N.



Carte aléatoire virtuelle

Les nombres pseudo–aléatoires générés par la carte d'indirection virtuelle peuvent être également utilisés pour contrôler tout autre paramètre, tel qu'une translation ou une rotation.

Pour obtenir des nombres à virgule flottante dans $[0,1[$, nous calculons $\frac{r}{T}$ où $r \in [0,T[$ est le résultat des permutations σ successives. Ceci produit des valeurs quantifiées puisqu'il n'y a que T valeurs différentes produites par σ. Si une meilleure résolution est nécessaire, une table de permutation de plus grande taille peut être utilisée. La nouvelle taille remplace T dans les formules précédentes.

Si au contraire les valeurs aléatoires désirées sont des nombres entiers, par exemple un indice dans une autre carte de taille T', il est alors préférable d'utiliser une table σ adaptée. En effet, si un nombre entier T' était calculé par la formule $T'\frac{r}{T}$ où $r \in [0,T[$ est le résultat des permutations σ successives, un effet similaire à l'aliasing apparaîtrait. A cause de la résolution limitée de $\frac{r}{T}$ les distributions de nombres produites ne seraient pas uniformes dans $[0,T'[$ (voir figure 3.2(b)). Notons que si T est un multiple de T', ce problème n'apparaît pas.

Des nombres aléatoires non corrélés peuvent être obtenus en utilisant différentes tables de permutation σ, ou en ajoutant un offset à g avant de calculer les permutations (des endroits distants ne sont pas corrélés puisque la permutation produit un nombre aléatoire). De plus, ces opérations peuvent être effectuées de manière vectorielle dans les cartes graphiques , ce qui permet de générer jusqu'à quatre nombres aléatoires en parallèle.

Transformation de Motif

Ce composant permet de redimensionner, déplacer ou tourner un motif à l'intérieur d'une case de la grille du pavage virtuel. L'objectif n'est pas ici de créer des textures continues, mais des textures montrant des objets sur un fond transparent : une couche de détails. Ceci est particulièrement utile pour représenter des objets distribués aléatoirement : pierres, feuilles mortes, etc ... (voir figure 3.14). Les transformations appliquées aux motifs peuvent être soit explicites (contrôlées par l'utilisateur), soit procédurales (générées par le composant *Carte aléatoire virtuelle*).

Le composant applique mise à l'échelle, rotation et translation (2D ou 3D) sur les coordonnées locales u_{tile}. Si les coordonnées transformées sortent du motif, la couleur de fond est affichée. Pour une translation d et un facteur d'échelle s dans une case, les nouvelles coordonnées peuvent être calculées comme $u'_{tile} = (\frac{u_{tile}}{s} + d)$. Cependant, dans ce cas le motif peut sortir de la case tout ou en partie, et être coupé. Il y a plusieurs manières de résoudre ce problème (voir figure 3.4) :

- Réduire la taille du motif d'une échelle s et autoriser seulement les déplacements qui ne permettent pas au motif de sortir de la case : $u'_{tile} = \frac{1}{s}(u_{tile} + (1-s)d)$.
- Gérer le débordement des motifs à l'aide de plusieurs passes, en tenant compte, dans chaque case, des débordements des motifs voisins. L'idée est de dessiner les deux parties du motif (resp. quatre parties en 2D) en dessinant la grille deux fois (resp. quatre en 2D). La première passe est habituelle. La seconde passe dessine la grille décalée d'une case vers la droite et applique une translation de $1-d$ à tous les motifs (voir figure 3.5). La couleur finale est la somme des contributions de chaque passe (une équation de mélange peut aussi être utilisée pour permettre aux motifs se recouvrant de se masquer).

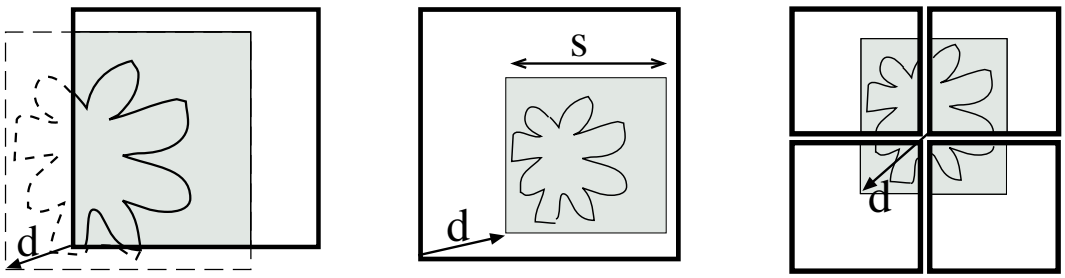


FIG. 3.4 – Positionnement des motifs.
A gauche : *Une translation d peut faire sortir le motif de la case.* Au milieu : *Mise à l'échelle d'un facteur s et translation contrainte.* A droite : *Gestion correcte du débordement des motifs voisins.*

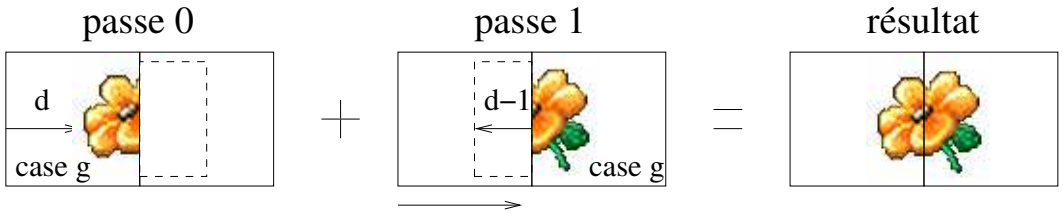


FIG. 3.5 – Gestion du débordements en dehors des cases.
La seconde passe dessine la grille décalée d'une case et applique la transformation complémentaire de la première passe. Le motif est ainsi complètement dessiné.

Carte de positionnement explicite

Le positionnement explicite de motifs utilise une *carte de positionnement* définie par l'utilisateur pour placer les motifs à des positions arbitraires dans la texture virtuelle. Ceci ne doit pas être confondu avec les cartes d'indirection où les motifs restent alignés avec la grille du pavage virtuel. Les motifs peuvent ici être positionnés n'importe où dans la texture.

L'idée est d'encoder le choix d'un motif comme dans une carte d'indirection, mais d'ajouter une translation (ou rotation, ou mise à l'échelle) au motif. Dans chaque case, une carte de positionnement contient donc :

- le numéro du motif choisi
- la transformation à appliquer au motif

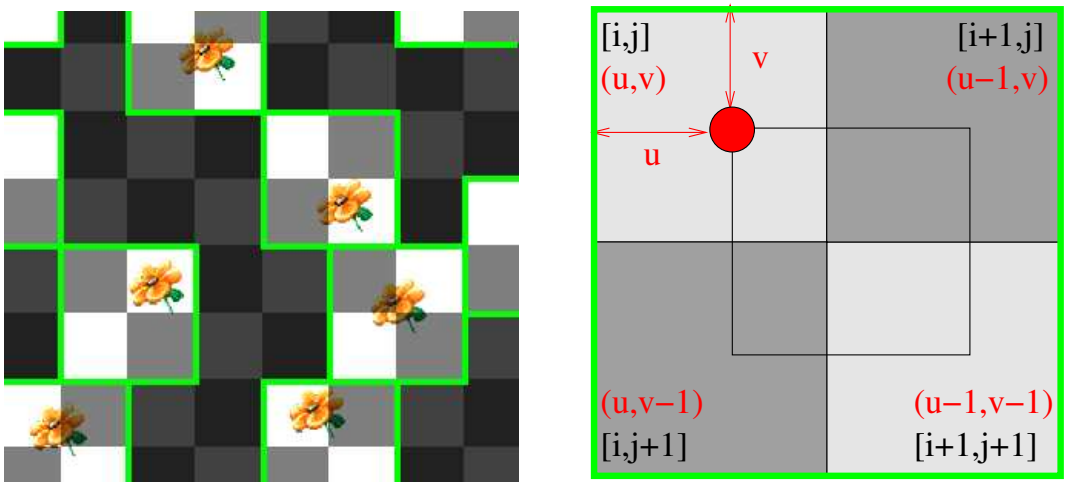


FIG. 3.6 – Positionnement explicite d'un motif.

Quatre cases doivent être mises à jour afin de positionner un motif dans la texture.

Si aucun motif ne doit être affiché, un numéro spécial est utilisé (-1). Lorsque les motifs représentent des objets distincts, une couleur de fond transparente est utilisée pour permettre d'appliquer la texture comme une couche de détails supplémentaires.

Afin de positionner le coin en haut à gauche d'un motif en une coordonnée u,v dans la texture, nous devons mettre à jour quatre cases de la carte de positionnement (voir figure 3.6). Pour permettre à n motifs d'être présents dans une même case de la grille virtuelle, il faut avoir recours à n cartes de positionnement. Une fois la position du motif encodée, celui–ci peut encore être mis à l'échelle ou tourné en utilisant le composant de transformation de motifs après le composant de carte de positionnement.

Notons que ce mécanisme de positionnement explicite correspond au concept de *particules de texture* — c'est à dire des éléments de textures indépendants vivant dans l'espace texture (voir chapitre 2, section 2.3). Grâce à notre approche, nous pouvons instancier et déplacer continûment les motifs dans la texture, sans avoir à modifier la géométrie et à faible coût mémoire puisque seules les informations de positionnement sont stockées. La carte de positionnement peut être mise à jour dynamiquement de manière efficace par l'utilisateur ou par un programme simulant le déplacement des motifs, par exemple pour simuler des impacts ou des gouttes d'eau (voir chapitre 4).

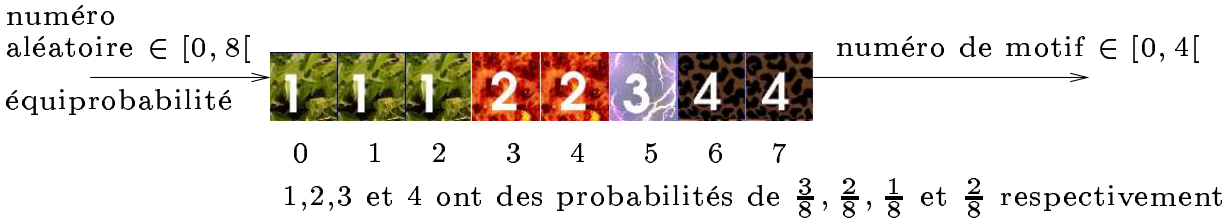
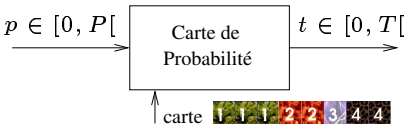


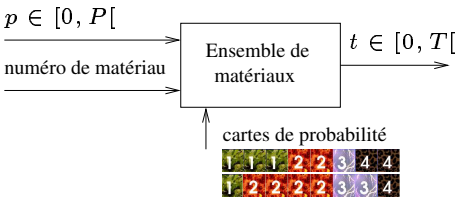
FIG. 3.7 – Carte de probabilité.

La carte de probabilité *représentée ici contient une distribution de $T = 4$ motifs, indexée par un nombre aléatoire dans $[0,8[$ ($P = 8$). Ceci permet de contrôler la probabilité d'apparition de chaque motif.*



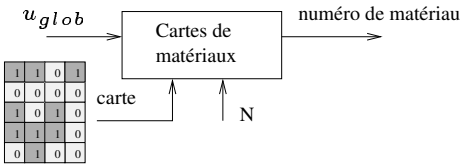
Carte de probabilité

Lorsqu'une carte d'indirection virtuelle est utilisée pour choisir les motifs, leur apparition est équiprobable : le générateur de nombres pseudo–aléatoires génère une distribution uniforme de valeurs. Le but de ce composant est de contrôler la probabilité d'apparition des motifs. Ceci est particulièrement utile si l'on veut, par exemple, distribuer deux fois plus de fleurs bleues que de fleurs rouges sur un terrain. Pour ce faire, nous intercalons le composant *Carte de probabilité* juste avant la carte de référence. La carte de probabilité contient P entrées. Les valeurs stockées sont les numéros des motifs entre $[0,T[$. Le choix aléatoire d'un numéro entre $[0,P[$ permettra d'accéder à la carte de probabilité pour trouver un numéro de motif entre $[0,T[$ (voir Figure 3.7). Si la distribution des nombres entre $[0,P[$ est uniforme, la distribution des nombres entre $[0,T[$ en sortie de la carte de probabilité est arbitraire et définie par les numéros de motifs stockés dans la carte. Par exemple, si un seul numéro de motif est stocké dans la carte, ce motif sera toujours sélectionné. Si deux numéros de motifs sont stockés, et que l'un est présent dans deux fois plus de cases de la carte de probabilité, alors ce motif sera sélectionné deux fois plus souvent. La taille P doit être choisie de manière à équilibrer la taille mémoire utilisée avec la résolution des probabilités (la plus petite probabilité possible est $\frac{1}{P}$).



Ensemble de matériaux

Nous pouvons considérer qu'une carte de probabilité (voir ci-avant) définit un *matériau*. Ce matériau est constitué par la probabilité d'apparition des motifs (par exemple de l'herbe avec quelques fleurs bleues). Le composant *Ensemble de matériaux* stocke simplement plusieurs matériaux dans une même structure, de la même manière que la texture de référence contient plusieurs motifs. Le composant ensemble de matériaux permet donc de sélectionner une carte de probabilité à partir d'un numéro de matériau.



Carte de matériaux

Afin de permettre à l'utilisateur de contrôler spatialement la distribution de différents matériaux sur le terrain, nous introduisons le composant *Carte de matériaux*. Ceci est particulièrement utile lorsque des motifs sont aléatoirement distribués sur le terrain : on souhaite par exemple qu'une zone donnée soit recouverte de fleurs bleues alors qu'une autre sera recouverte de fleurs rouges. Ce type de contrôle est essentiel pour briser les répétitions d'aspect à grande échelle (voir chapitre 2, sections 3.3 et 3.5.2). Une carte de matériaux contient simplement les numéros des matériaux utilisés le long du terrain. Cette carte peut être soit définie par l'utilisateur (explicite) soit procédurale (virtuelle).

La résolution d'une carte de matériaux est généralement basse en comparaison de la texture générée. Le contrôle se veut global. Chaque case de la carte recouvre plusieurs cases de la grille du pavage virtuel. Si la carte des matériaux a une taille de A, alors la taille N de la grille du pavage virtuel doit être un multiple de A.

Carte de matériaux interpolées

Les cartes de matériaux sont de faible résolution. Malheureusement, ceci peut devenir évident car les transitions entre différents matériaux sont abruptes : d'une case à l'autre de la carte, la probabilité d'apparition des motifs change brusquement. Pour éviter ce défaut visuel nous définissons comment interpoler entre deux matériaux.

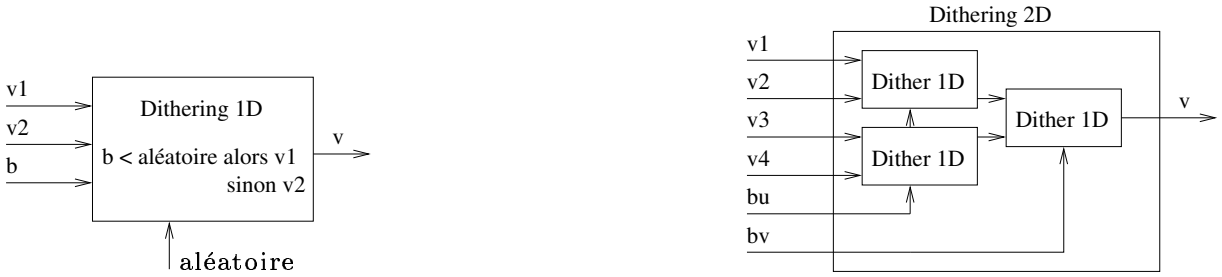


FIG. 3.8 – Opérateurs de *dithering* 1D et 2D

A gauche : *Opérateur de dithering 1D.* A droite : *Opérateur de dithering 2D construit à partir de l'opérateur 1D.*

Quel type d'interpolation utiliser ?

- L'interpolation linéaire n'est pas définie dans ce cas puisque nous travaillons sur des numéros de matériaux, indexant des cartes de probabilités.
- Nous pourrions mélanger par transparence les motifs sélectionnés par les deux matériaux, mais cela produirait de mauvais résultats visuels (les motifs deviendraient progressivement transparents dans les zones de transition).

Pour interpoler entre les deux matériaux nous utilisons en fait un opérateur de *dithering*. Cet opérateur sélectionne l'un ou l'autre des matériaux en fonction d'un paramètre d'interpolation *b* (voir figure 3.8). L'idée est de générer un nombre pseudo–aléatoire avec la table de permutation σ et de le comparer à *b*. Si *b* est inférieur au nombre aléatoire, le premier matériau est choisi, dans le cas contraire le second matériau est choisi. Notons que, dans le cas 2D, il faut choisir entre quatre matériaux : l'opérateur de *dithering* est appliqué deux fois. Pour contrôler l'interpolation entre deux cases nous calculons *b* comme la coordonnée barycentrique du pixel dans le voisinage g',g'+1 (voir figure 3.9).

Ceci nous permet de définir des *Cartes de matériaux interpolées* qui permettent des transitions continues (du point de vue des motifs choisis) entre les matériaux sélectionnés par une carte de matériaux. Le résultat est illustré figure 3.10d.

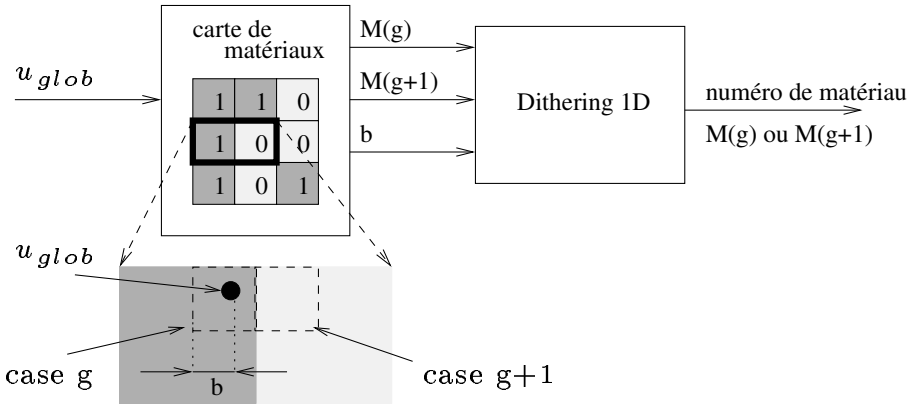


FIG. 3.9 – Interpolation d'une carte de matériaux.

Les cartes de matériaux de faible résolution peuvent être interpolées par un opérateur de dithering qui choisit l'un ou l'autre des matériaux avec une probabilité dépendant de la position. Pour plus de clarté, l'interpolation est ici effectuée sur une seule dimension.

1.4 Transitions

Afin de créer des textures continues avec un pavage apériodique (rappelons que dans ce cas les motifs sont carrés et appelés pavés), le contenu des différents pavés doit être le même le long des arêtes (voir chapitre 2, section 3.3) : les arêtes doivent être *compatibles*. Cependant ceci implique que la texture générée est relativement homogène. Or les terrains sont composés de nombreux aspects (herbe, sable, roche, ...). Même si chacun des ces aspects est relativement homogène sur une même zone, il faut pouvoir spécifier des transitions entre les différents matériaux (voir chapitre 2, section 3.5.2).

Nous introduisons donc la notion de *Familles de matériaux*. Une famille correspond à un jeu de pavés dont les arêtes sont compatibles : ils peuvent former un pavage dont l'aspect sera continu. Les composants présentés ci–après permettent de définir des transitions entre différentes familles de matériaux. Notons que les pavés d'une même famille peuvent être simplement sélectionnés à l'aide d'une carte de probabilité.

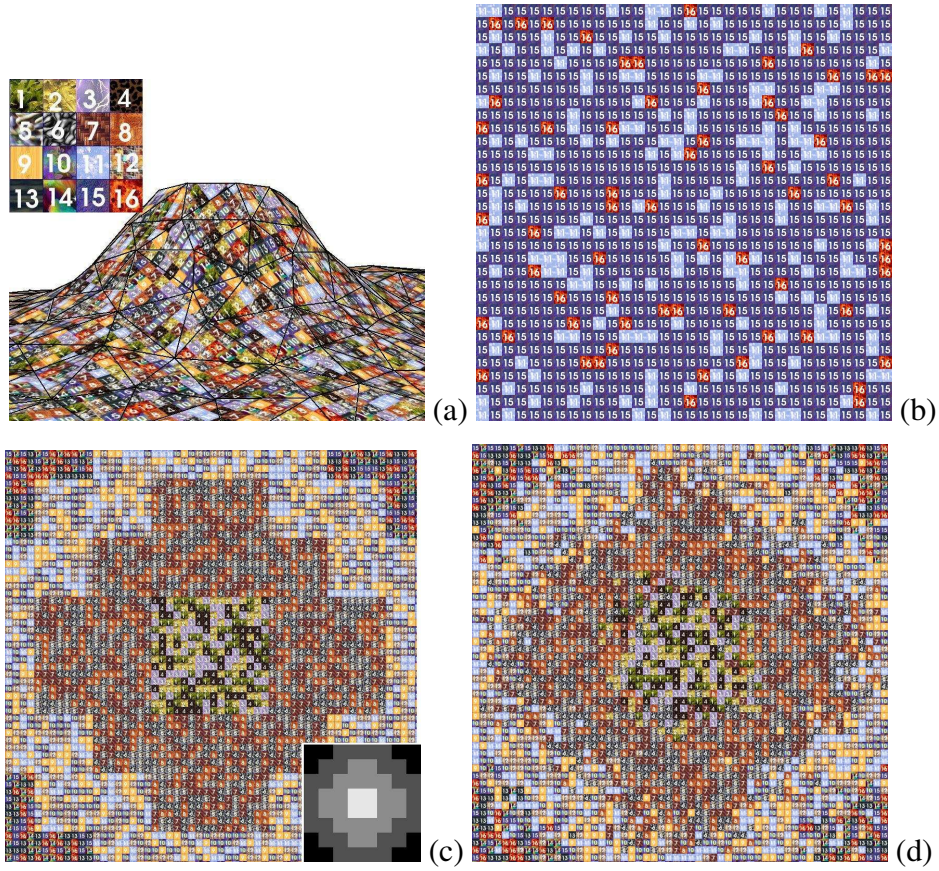
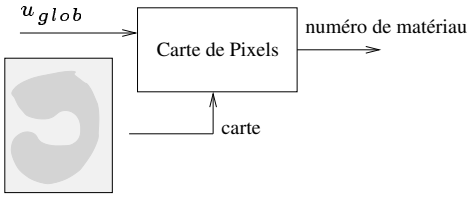


FIG. 3.10 – Pavages apériodiques générés par notre méthode

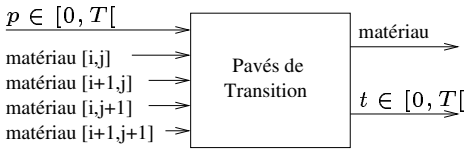
De gauche à droite *:* (a) : *Pavage apériodique de 16 motifs sur un terrain. Tout étant calculé dans l'espace texture, le pavage est indépendant du maillage triangulaire.* (b) : *Distribution non–uniforme de motifs avec une carte de probabilité.* (c) : *Distribution non–uniforme contrôlée spatialement avec une Carte de matériaux de résolution* 8×8 *(montrée en bas à droite).* (d) : *Même résultat en utilisant une carte de matériaux interpolées. La texture générée a une résolution virtuelle de* 4096×4096*, et peut atteindre des résolutions bien plus élevées en augmentant la valeur de N (taille de la grille du pavage virtuel).*

Nous introduisons deux nouveaux composants dédiés aux zones de transition. Le premier est basé sur une *carte de pixels* qui indique quelle famille utiliser pour chacun des pixels de la texture générée. Le second composant est basé sur la notion de *pavés de transition* (voir chapitre 2, section 3.5.2). Les pavés spécifient la transition entre deux familles de matériaux pour toutes les configurations possibles (16 en 2D, voir chapitre 2, figure 2.24).



Cartes de pixels

Les cartes de pixels sont utilisées pour déterminer précisément la forme des bordures des zones recouvertes par différentes familles de matériaux. Elles sont de haute résolution, possiblement la même que celle de la texture générée. Cependant, elle peuvent être compressées fortement par exemple par la méthode de Kraus et Ertl [KE02]. Ces cartes se distinguent des pavés de transition en cela qu'elles spécifient les transitions au niveau du pixel alors que les pavés de transition spécifient les transitions au niveau des cases du pavage virtuel.



Pavés de transition

Afin d'utiliser les pavés de transition, l'utilisateur doit fournir un ensemble de pavés qui décrit toutes les transitions possibles entre les différentes familles de matériaux. (voir figure 3.11). Ces pavés sont stockés comme les autres. Ils sont choisis par le composant *Pavés de transition* qui sélectionne le pavé de transition en fonction des familles de matériaux présentes dans les quatre cases voisines (en 2D) de la grille du pavage virtuel.

Une transition entre m familles de matériaux nécessite m^4 pavés de transition. Pour 2 familles, il y a 16 configurations possibles pour un voisinage en 2D (2×2), en incluant les 2 cas correspondant à "toutes les cases sont de la même famille". Le numéro des pavés de transition sélectionnés est généré en combinant les numéros des familles de matériaux (voir figure 3.11) Si aucune transition n'est nécessaire, alors le composant n'a aucune action ; sinon le pavé de transition est utilisé dans la texture générée.

Notons qu'en pratique les transitions sont souvent définies entre une famille de matériau et un fond transparent. Ceci permet d'obtenir toutes les transitions entre cette famille et les autres. Le figure 3.15a, en haut à droite, montre un exemple de pavés de transition entre de l'herbe et un fond transparent (montré en noir).

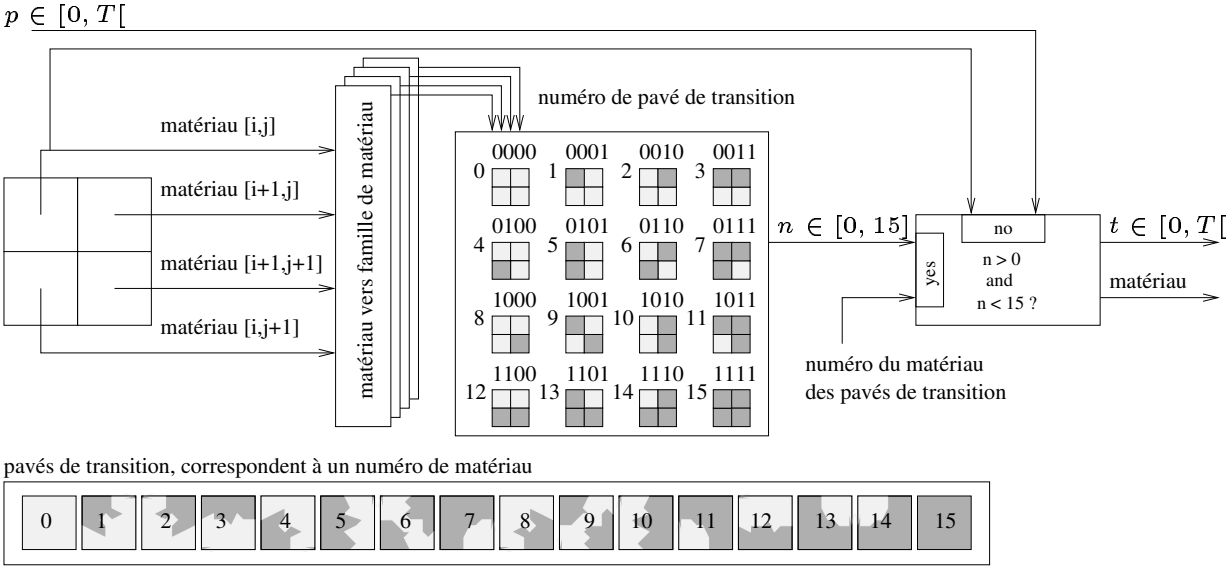


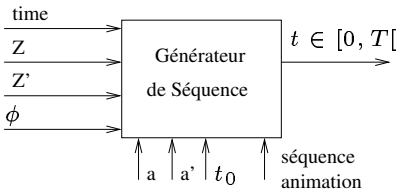
FIG. 3.11 – Calcul du numéro du pavé de transition

Le numéro du pavé de transition est calculé à partir du voisinage. Si toutes les cases voisines utilisent la même famille de matériau alors le composant ne prend aucune action : il recopie le paramètre d'entrée p sur sa sortie t et retourne le numéro de matériau de la case i, j.

1.5 Animation

Notre système permet de définir des motifs dont l'apparence dépend de paramètres tels que le temps, la direction de vue, la distance à un point, etc ... Ceci est réalisé en remplaçant un unique motif par une séquence de motifs qui encode sa variation selon le paramètre. Par exemple une fleur s'ouvrant et se refermant avec le temps. Le premier motif de la séquence est considéré comme l'état au repos (par exemple la fleur fermée).

Le motif animé peut être distribué sur le terrain comme auparavant. Si l'animation est jouée telle quelle, tous les motifs seront animés exactement en même temps. Par exemple toutes les fleurs s'ouvriront et se refermeront de manière synchrone sur le terrain. Cet effet n'a rien de naturel et doit donc être supprimé afin que des motifs voisins aient des animations légèrement différentes. C'est le rôle du composant suivant.



Générateur de séquence

Lorsque des séquences d'animation sont utilisées, l'utilisateur souhaite généralement éviter la synchronisation entre les animations des motifs et en contrôler le déclenchement. Le rôle du composant *Générateur de séquence* est de modifier la séquence d'animation fournie par l'utilisateur pour chaque instance du motif animé. Deux types de contrôle sont effectués : le contrôle du

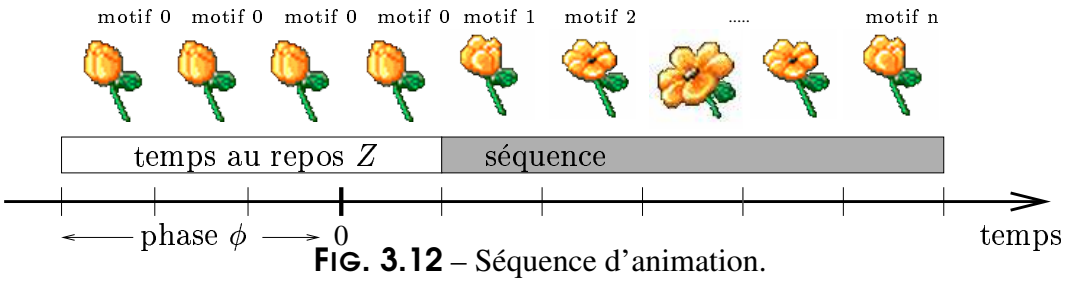


FIG. 3.12 – Séquence d'animation.

Séquence d'animation avec $\phi = 3$ *et* $Z = 2$.

temps Z pendant lequel l'animation reste au repos et le contrôle de la phase ϕ de l'animation qui permet de décaler les animations de deux motifs voisins. (voir figure 3.12). Les deux paramètres peuvent être contrôlés explicitement ou procéduralement.

Pour obtenir des animations asynchrones, un paramètre ϕ différent est utilisé en divers endroits. Ceci est réalisé à l'aide d'une carte aléatoire virtuelle (générateur de nombre pseudo–aléatoires). Le paramètre Z varie également spatialement afin que les animations ne soient pas déclenchées avec la même fréquence (différents motifs resteront plus ou moins longtemps au repos). Afin de faciliter le contrôle global sur les animations, nous multiplions le temps au repos Z par un paramètre global a qui peut être changé par l'application. Ceci permet de contrôler avec quelle fréquence les animations sont globalement déclenchées.

Le numéro du motif de la séquence d'animation à utiliser au temps t est ainsi calculé par

$$p = MAX(\ 0,\ (t+\phi) mod (n+aZ) - aZ\ )$$

où *mod* est l'opérateur modulo et n la longueur de la séquence d'origine du motif.

Transitions temporelles Nous avons maintenant un mécanisme permettant d'animer les motifs avec des séquences d'animations variant spatialement. Cependant, il est souhaitable de pouvoir également faire varier les séquences d'animation (paramètres a et Z) dans le temps. Le principal problème est de pouvoir modifier ces paramètres sans introduire de discontinuité temporelle (*popping*) visible dans la texture générée. Il nous faut donc définir des transitions temporelles : comment passer d'une séquence animée définie par des paramètres a_0, Z_0 à une autre séquence animée définie par de nouveaux paramètres a_1, Z_1. La difficulté majeure est que nous ne devons pas avoir recours à des variables d'état : cela empêcherait la texture d'être procédurale puisqu'il faudrait stocker et maintenir à jour un état par case de la grille de pavage virtuel. Nous devons donc choisir le bon moment pour passer de l'ancienne séquence à la nouvelle séquence (voir la figure 3.13).

Soit a_0, Z_0 les anciens paramètres et a_1, Z_1 les nouveaux paramètres après la transition souhaitée au temps t_0. Le motif de l'animation choisi par l'ancienne séquence est :

$$p_0(t) = MAX(\ 0,\ (t+\phi_0) mod (n+a_0 Z_0) - a_0 Z_0\ )$$

Le motif choisi par la nouvelle séquence est :

$$p_1(t) = MAX(\ 0,\ (t+\phi_1) mod (n+a_1 Z_1) - a_1 Z_1\ )$$

L'idée est de choisir, à chaque pas de temps, entre le motif de l'ancienne séquence, le motif de la nouvelle séquence ou le motif correspondant à l'état au repos. La fin de la nouvelle séquence est attendue, puis l'animation est maintenue au repos (si nécessaire) jusqu'à ce que la nouvelle séquence recommence (en passant par l'état au repos). Ceci assure la continuité de l'animation.

Notons t_f le temps auquel l'ancienne séquence retourne au motif de repos. Pour une transition déclanchée au temps t_0, examinons la valeur de $p_0(t_0)$: soit $p_0(t_0) = 0$ (au temps t_0 l'ancienne séquence montre le motif au repos), et, dans ce cas, nous choisissons $t_f = t_0 - 1$, soit $p_0(t_0)$ est l'un des n motifs de la séquence et, dans ce cas, la séquence se terminera à $t_f = t_0 + n - p_0(t_0)$. Aussi longtemps que $t \leq t_f$, nous continuons l'ancienne séquence. Dès que t_f est atteint, soit $p_1(t_f) = 0$ et nous pouvons commencer la nouvelle séquence, soit nous devons attendre la fin (passage du motif au repos) de la nouvelle séquence à $t_{end} = t_f + n - p_1(t_f)$. En attendant le motif au repos est affiché. Notons que le choix de la séquence peut être entièrement déterminé à partir des anciens et nouveaux paramètres, du temps t et de l'instant de transition souhaité t_0. Il n'est donc pas nécessaire de stocker de variable d'état. Rappelons que les paramètres a et Z sont soit stockés dans des cartes de faible résolution, soit générés procéduralement.

Lorsque l'utilisateur veut changer la carte définissant Z ou le paramètre a, il doit conserver les anciennes valeurs et l'instant de transition t_0. La transition sera effectuée en au plus $2n$ pas de temps. Une fois ce temps écoulé une autre transition peut être effectuée avec la garantie qu'aucun effet de discontinuité ne va apparaître.

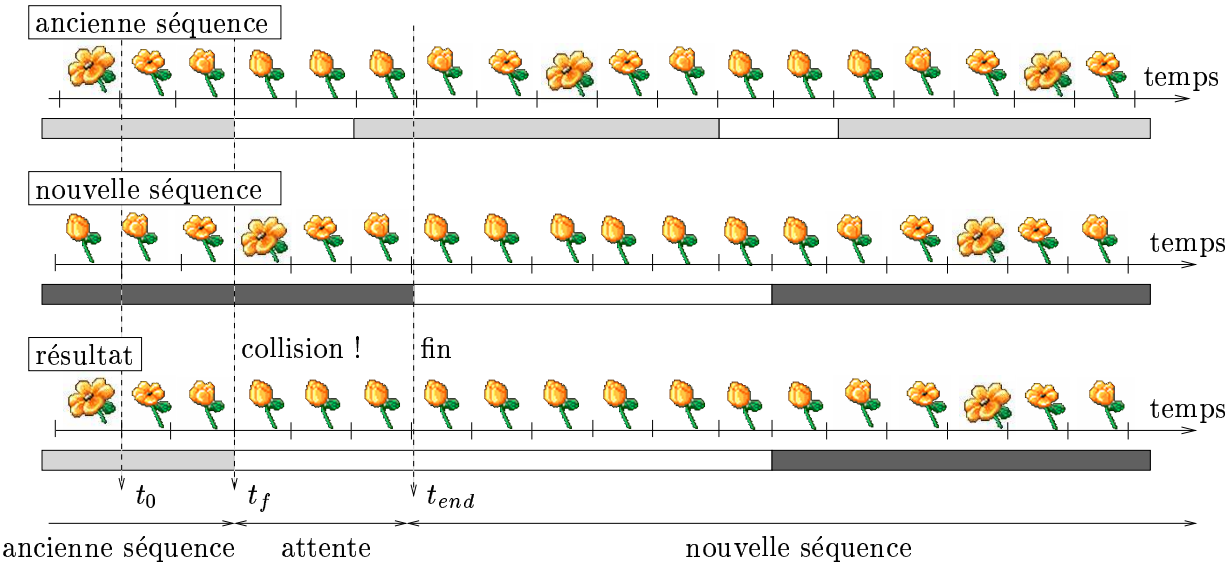


FIG. 3.13 – Suppression des discontinuités temporelles.

Transitions entre anciennes et nouvelles séquences définies par différents paramètres Z et a.

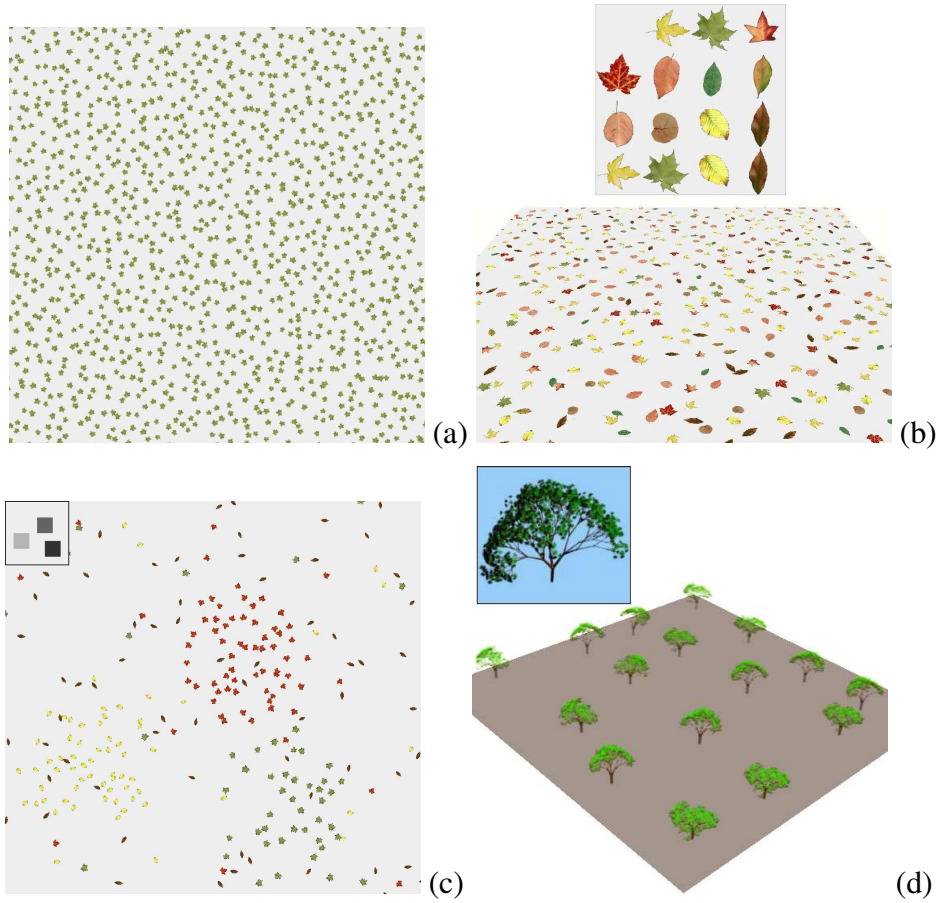


FIG. 3.14 – Distribution aléatoire de motifs sur un terrain.
De gauche à droite *:* (a) : *Distribution aléatoire d'un motif de feuille (avec rotations aléatoires).* (b) : *Distribution aléatoire de motifs de feuilles avec carte de probabilité.* (c) : *Distribution aléatoire de motifs de feuilles avec carte de matériaux interpolée (*8×8*, en haut à gauche).* (d) : *Même type de texture utilisé pour positionner des textures volumiques (l'arbre est une texture* $256 \times 256 \times 256$*, la texture volumique générée est affichée avec 256 couches de polygones).*

2 Notes d'implémentation

2.1 Encodage des tables en 2D

Bien que nous manipulions des tables qui pourraient être 1D (la texture de référence pourrait stocker les motifs sur une ligne, la table de permutation σ peut être définie sur une seule dimension, etc ...), nous choisissons la plupart du temps d'utiliser des tables à deux dimensions.

Ceci est principalement dû à l'implémentation pour les processeurs graphiques. En effet, utiliser des tables 2D permet de tirer meilleur profit de la dynamique limitée des coordonnées de texture utilisées. De plus, les processeurs graphiques effectuent des opérations vectorielles et de nombreux calculs peuvent être parallélisés.

Par exemple la texture de référence contient une liste de T motifs. Nous la stockons sous la forme d'une table 2D dans laquelle $T_x \times T_y$ motifs sont empaquetés. Au lieu d'utiliser un numéro de motif p, nous utilisons la position de la case du motif dans la texture de référence (p_x, p_y). Ceci peut être considéré comme un numéro "vectoriel". Nous appliquons la même approche à toutes les tables. Les cartes d'indirections encodent un numéro de motif sous forme vectorielle. Une carte d'indirection I devient donc une fonction de de $[0,N_x[\times[0,N_y[$ dans $[0,T_x[\times[0,T_y[$ (où $N_x \times N_y$ est la taille de la grille du pavage virtuel et $T_x \times T_y$ le nombre de motifs). De la même manière, les tables de permutation σ sont vectorielles : le premier composant σ_x est une permutation des indices horizontaux (entre 0 et $T_x - 1$) alors que la seconde composante σ_y est une permutation des indices verticaux (entre 0 et $T_y - 1$). La permutation est définie par $\overrightarrow{\sigma}(\overrightarrow{p}) = (\sigma_x(\overrightarrow{p}), \sigma_y(\overrightarrow{p}))$.

2.2 Encodage d'entiers dans des textures

Pour plus de simplicité, nous avons supposé jusqu'ici que des nombres entiers étaient stockés sans difficulté dans des textures. Cependant, les textures contiennent généralement des valeurs entre $[0,1]$ sur les cartes graphiques , et en particulier les textures 8 bits. De plus l'espace texture est normalisé et accédé à l'aide de coordonnées comprises dans l'intervalle $[0,1]$. Il nous faut donc simuler une carte de taille N qui encode une fonction de $[0,N[$ dans $[0,M[$ (où M est la valeur maximale qui peut être stockée) à l'aide d'une texture qui représente une fonction de $[0,1]$ dans $[0,1]$. Si la carte doit être accédée par un indice i, alors nous utilisons $\frac{i}{N}$ pour accéder la texture correspondante. L'entier correspondant a la valeur $r \in [0,1]$ lue dans la texture est $\lfloor rM \rfloor$. Il faut apporter un soin particulier au choix des valeurs M car si la texture utilisée a une précision de 8 bits alors la valeur r à une résolution limitée.

3 Connecter les composants pour générer des textures : étude de cas

Dans les sections précédentes, nous avons défini un ensemble de composants de base. Chacun effectue un traitement simple : génération de coordonnées dans une grille, génération de nombres pseudo–aléatoires, transformation des motifs, etc ...

En fait, chacun de ces composants correspond à un petit programme écrit en langage *Cg* [Nvi02] et qui peut être exécuté sur une carte graphique. En combinant les composants nous pouvons donc écrire des programmes qui génèrent des textures.

Nous allons voir maintenant comment connecter ensemble les composants pour générer deux types de textures très importants : pavages apériodiques et distribution aléatoire d'objets sur un terrain. Dans chaque cas, nous proposons une version simple et une version permettant un fort contrôle utilisateur. Nous verrons ensuite comment créer des textures animées.

Pavage apériodique simple (voir figure 3.3 à droite, 3.10a)
L'utilisateur fournit une texture de référence qui contient $T \times T$ pavés. Le but est de réaliser un

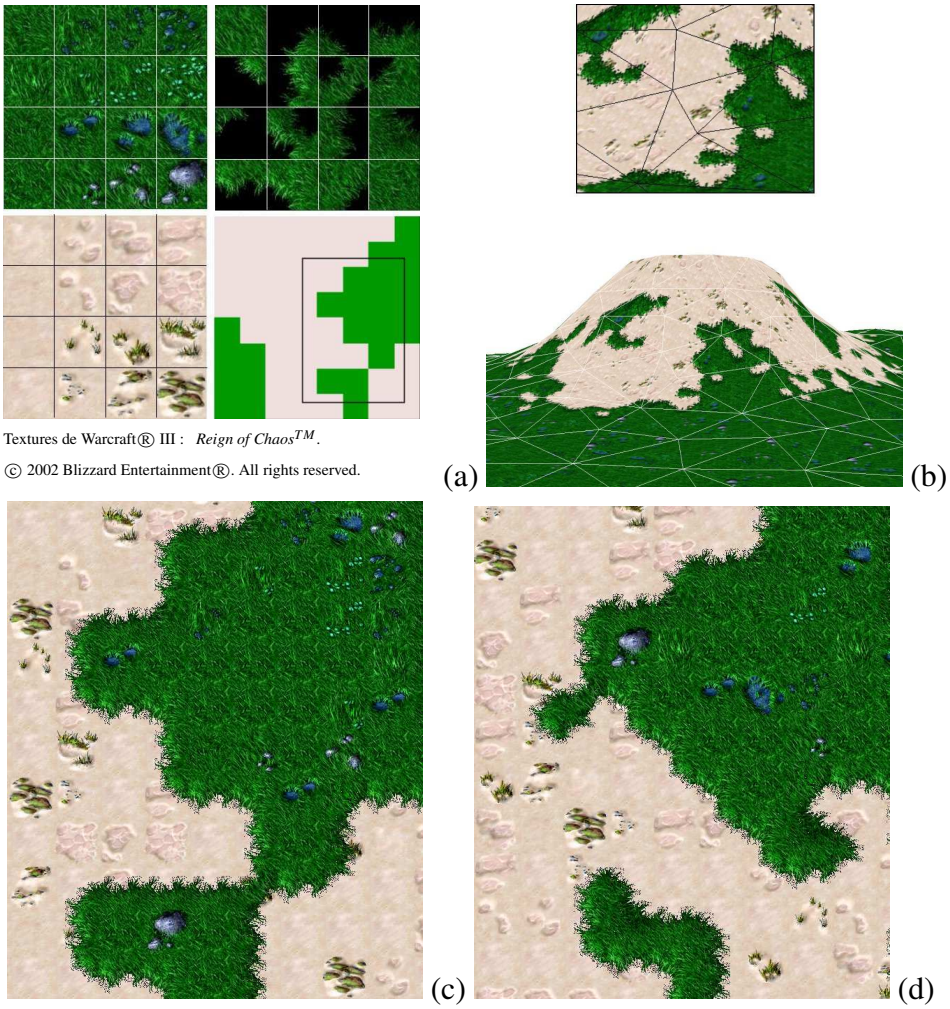


FIG. 3.15 – Pavages apériodiques générés par une combinaison des composants de base

De gauche à droite *:* (a) : *Pavés standard et pavés de transition (provenant du jeu Warcraft 3) et une carte de matériaux* 8×8. (b) : *Zoom sur le détail d'une texture générée (la géométrie est un simple rectangle).* (c) : *Interpolation de la carte.* (d) : *La texture générée (*4096×4096*, générée à la volée) est appliquée sur un terrain.*

pavage apériodique dans une grille virtuelle de taille $N \times N$ avec une distribution uniforme des pavés. Les composants nécessaires sont :

- un composant *Pavage virtuel* qui calcule les coordonnées de la case g et les coordonnées relatives u_{tile} à partir des coordonnées de texture u_{glob}.
- un composant *Carte d'indirection virtuelle* qui calcule un nombre pseudo–aléatoire p à partir de l'indice de case g.
- un composant *Texture de référence* utilisé pour calculer la couleur finale à partir du numéro de pavé et des coordonnées relative u_{tile} Le schéma de connexion est donné figure 3.16.

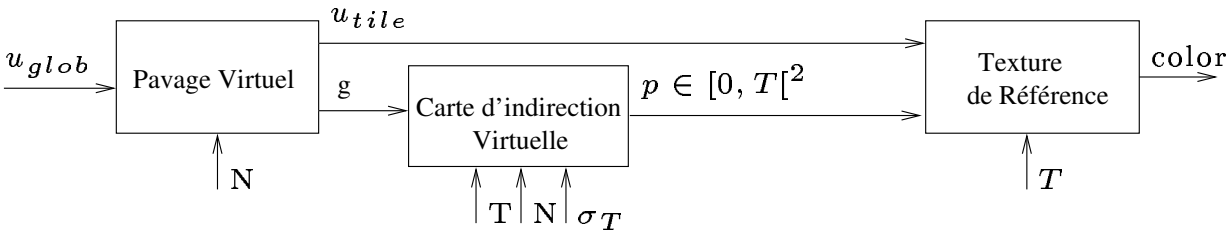


FIG. 3.16 – Schéma du pavage apériodique simple

Les composants Pavage virtuel, Carte d'indirection virtuelle *et* Texture de référence *sont connectés pour générer une texture contenant un pavage apériodique avec distribution uniforme des pavés.*

Le *fragment program* peut être écrit très simplement à partir du schéma :

```
void fragment_program(float2 u_glob,out:float4 color) {
   float2 g,u_tile,p;
   pavage_virtuel(u_glob,out:g,out:u_tile);
   carte_indirection_virtuelle(T,N,sigma,g,out:p);
   texture_reference(T,p,u_tile,out:color);
}
```

Pavage apériodique étendu (voir figure 3.15)
L'utilisateur définit 2 familles de matériaux (par exemple herbe et sable) et 2^4 pavés de transition. Il définit également différents matériaux à l'aide de cartes de probabilité (voir figure 3.15a gauche). Rappelons qu'un matériau est défini par une probabilité d'apparition des motifs et qu'une famille de matériaux est un ensemble de pavés aux arêtes compatibles. Enfin, il fournit également une carte de matériaux M de faible résolution (voir figure 3.15a en bas à droite).

L'objectif est de générer un pavage aléatoire avec le matériau "herbe" dans les zones sombres de la carte de matériaux et le matériau "sable" dans les zones claires. Les pavés de transition devront être automatiquement sélectionnés (voir figure 3.15 b et c).

Pour ce faire, l'idée est de d'abord vérifier si une transition est nécessaire avec un composant *Pavés de transition*. Ses entrées sont les numéros des matériaux de la case courante g et de ses trois cases voisines, ainsi qu'un numéro aléatoire p généré par un composant *Carte d'indirection virtuelle*. Les numéros de matériaux sont fournis par quatre cartes de matériaux interpolées. Lorsqu'une transition est nécessaire le composant *Pavés de transition* renvoie le numéro du pavé de transition approprié. Dans le cas contraire, il renvoie le numéro aléatoire p ainsi que le matériau m utilisé par la case g (tous deux fournis en entrée). Ces informations sont utilisées par un composant *Ensemble de matériaux* qui sélectionne la carte de probabilité du matériau m et utilise le numéro p pour trouver un numéro de motif t. Un composant *Texture de référence* est alors utilisé pour trouver la couleur finale. Le schéma de connexion est donné figure 3.17.

Distribution aléatoire de motifs (voir figure 3.14a)
Un seul motif est utilisé et stocké dans la texture de référence ($T = 1$). L'objectif est de distribuer aléatoirement le motif sur le terrain en approximant une distribution de Poisson de rayon p. Ceci

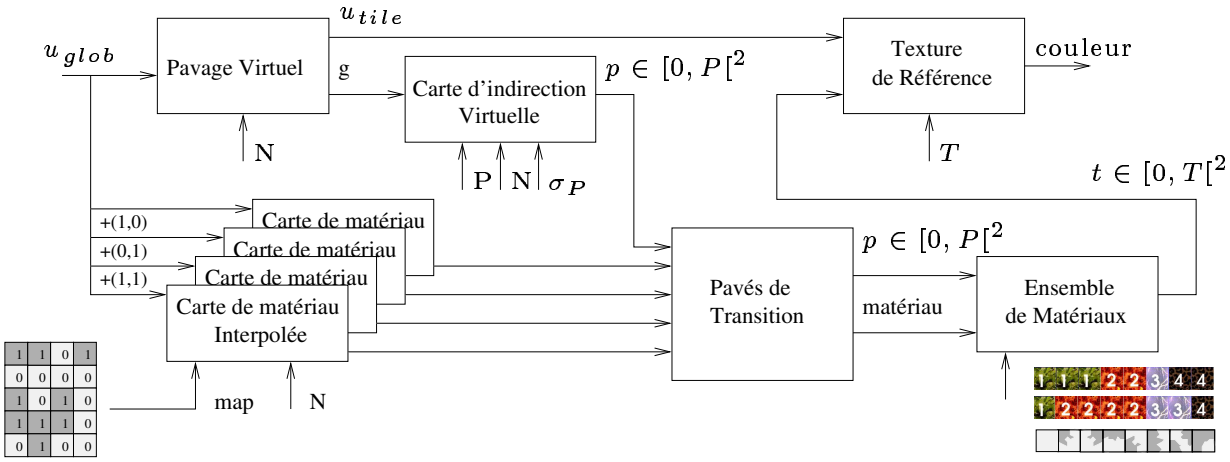


FIG. 3.17 – Pavage apériodique avancé

se fait en choisissant une position aléatoire pour le motif dans chaque case de la grille du pavage virtuel. La taille de la grille est $N \times N$, avec $N = \frac{1}{p}$ (comme cela est généralement fait [Coo85] pour approcher une distribution de Poisson). Le schéma de connexion est donné figure 3.18.

Le composant *Carte aléatoire virtuelle* est utilisé pour calculer un vecteur de nombres flottants aléatoires. Ils permettent de spécifier une translation (2 nombres), un facteur d'échelle et une rotation (1 nombre pour chaque). En pratique nous utilisons une permutation σ de taille 256×256. Ceci permet une résolution de $\frac{1}{256}$ pour les nombres flottants générés. Le composant *Transformation de motif* est ensuite utilisé pour appliquer la transformation aux coordonnées relatives. Ce composant peut décider d'arrêter la génération de la texture et d'afficher la couleur de fond lorsque la transformation sort des bords du motif.

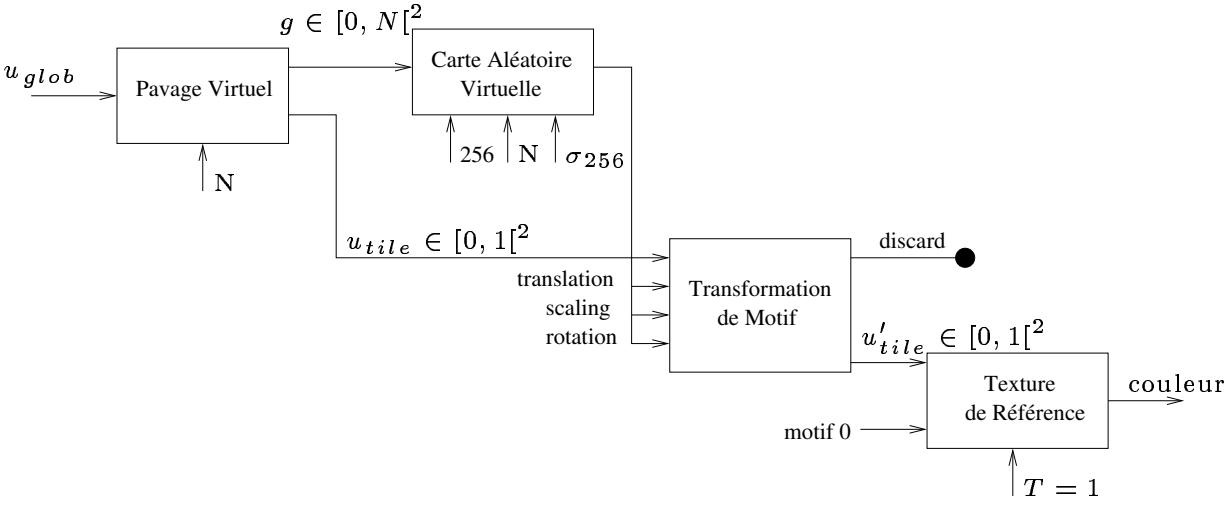


FIG. 3.18 – Distribution aléatoire de motifs

Distribution aléatoire contrôlable (voir figure 3.14c)
Cette extension du schéma précédent permet à l'utilisateur de contrôler la densité et la distribution spatiale des motifs. A cette fin, l'utilisateur définit des matériaux correspondant aux proportions souhaitées de chaque motif. Un motif "vide" est utilisé pour permettre de faire varier la densité. L'utilisateur fournit également une carte de matériaux *M* de faible résolution. Ceci va lui permettre de contrôler les distributions de motifs apparaissant le long du terrain. On souhaite par exemple voir plus de feuilles mortes sous un arbre que sur le reste du décor. Par rapport au schéma de distribution aléatoire les composants suivants sont ajoutés :

- un composant *Carte d'indirection virtuelle* pour générer un numéro aléatoire dans chaque case g.
- un composant *Carte de matériaux interpolée* pour choisir les matériaux.

Le schéma de connexion est donné figure 3.19.

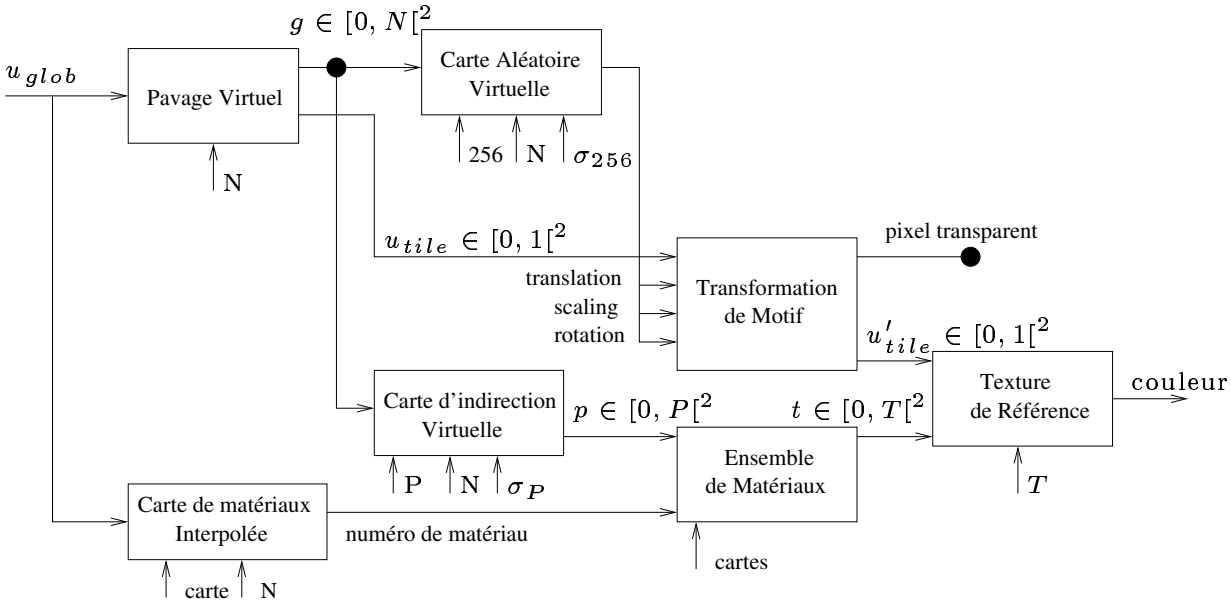


FIG. 3.19 – Distribution aléatoire contrôlable de motifs

Animation (voir figures 3.21d et 3.21a-c)
Nous présentons ici deux types d'animations :

- déplacement continu d'un motif le long d'un chemin sur le terrain (voir figure 3.21d).
- motifs avec séquence d'animation (voir figure 3.21a-c).

Le premier cas correspond à la mise à jour dynamique d'une carte de positionnement explicite. Seul un petit nombre de paramètres doivent être mis à jour à chaque pas de temps : les quatre cases (en 2D) permettant de positionner un motif. Aucune géométrie supplémentaire n'est nécessaire pour véritablement *instancier* les motifs : l'image n'est stockée qu'une seule fois en mémoire. Le chapitre 4 présente également un habillage animé complexe créé autour de cette approche.

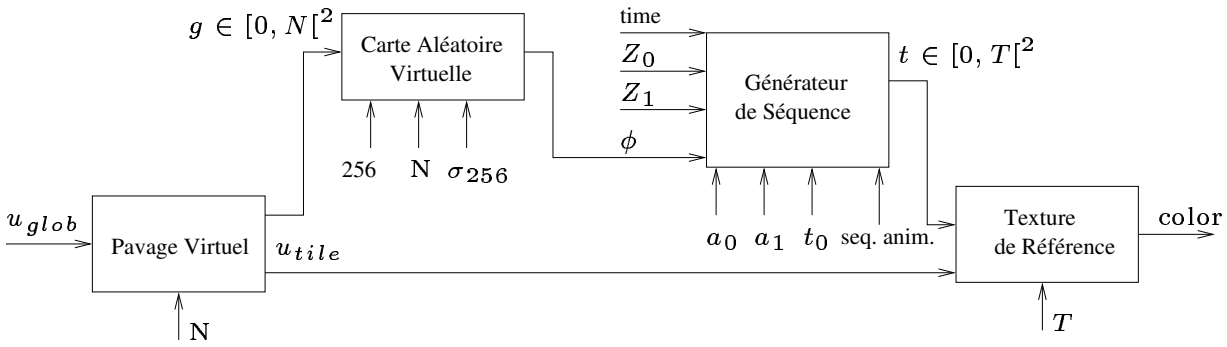


FIG. 3.20 – Grille de motifs animés

Le paramètre ϕ est aléatoire. Le paramètre Z est contrôlé par l'application.

Le second cas correspond à animer l'apparence des motifs en fonction d'un paramètre tel que le temps ou une distance. Ceci peut être le contrôle de la couleur, de la rotation mais également le contrôle de séquences d'animation, telles que celles présentées en section 1.5. Nous nous concentrons ici sur l'animation de motifs à l'aide de séquences d'animation. Le schéma de connexion complet est donné figure 3.20. Les paramètres *temps*, Z_0, Z_1, a_0, a_1 et t_0 sont gérés par l'application (il peuvent changer dans le temps, voir section 1.5).

4 Filtrage des textures générées

Le filtrage des textures effectué par les cartes graphiques ne peut pas fonctionner correctement lorsque des discontinuités de voisinage sont introduites (voir chapitre 2, section 4). Or les cartes d'indirection que nous utilisons introduisent ce type de discontinuités. Nous discutons, dans cette section, les différentes approches possibles pour obtenir un meilleur filtrage, dans le contexte des applications interactives et des processeurs graphiques.

Si nous laissons la carte graphique filtrer comme d'habitude, le problème est que les voisins utilisés pour l'interpolation linéaire au bord des motifs sont lus dans la texture de référence au lieu d'être considérés dans la texture générée. Du coup les couleurs interpolées au bord sont fausses ce qui peut produire des discontinuités visibles. Ce problème s'étend aussi à l'interpolation linéaire effectuée par le MIP–mapping (voir chapitre 2, section 4).

4.1 Solutions pour nos textures procédurales

Le filtrage à appliquer dépend du nombre de texels qui sont projetés dans un pixel de l'écran. Dans le cas idéal, un texel correspond à un pixel. Nous pouvons distinguer les trois cas suivants :

Moins d'un texel par pixel : Pour correctement effectuer une interpolation linéaire, il nous faudrait calculer, à l'aide de notre texture procédurale, non plus un, mais quatre pixels. Le coût devient élevé, même si ceci reste possible.

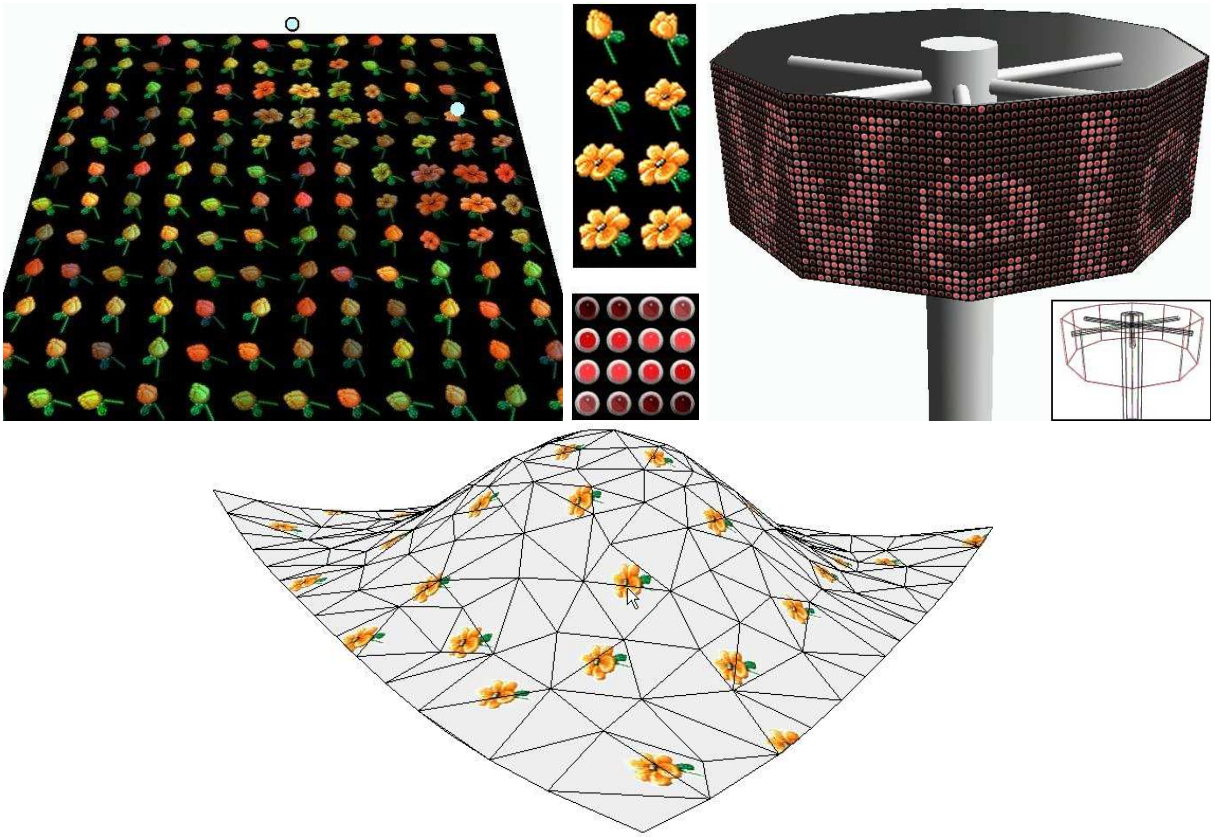


FIG. 3.21 – Motifs animés.
De gauche à droite *:* (a) *La séquence d'animation est contrôlée par la distance à deux points.* (b) *Séquences d'animations des motifs utilisés dans les images a et c.* (c) *Un tableau d'affichage. Les LED clignotent à une fréquence qui est contrôlée par une texture défilant (paramètre Z). La texture générée a une résolution de* 12288×1536. (d) *Positionnement interactif de motifs.*

En fait, nous utilisons en pratique des pavés dont les arêtes sont compatibles, ou des motifs sur fond transparent. Dans un cas comme dans l'autre, l'interpolation avec les voisins dans la texture de référence va donner un résultat similaire à l'interpolation dans la texture générée. En effet, les arêtes étant compatibles, les voisinages, du point de vue des couleurs, sont les mêmes.

Notons que ceci ne reste malheureusement pas vrai si l'interpolation linéaire est effectuée dans différents niveaux de la pyramide de MIP-mapping. A partir d'un certain niveau, les bordures cessent de se correspondre. Nous discutons en section 4.2 comment une extension des cartes graphiques actuelles pourrait résoudre ce problème.

Plusieurs texels dans un pixel : Les motifs sont tous stockés dans la texture de référence. Supposons que les motifs aient une taille en puissance de deux. Il existe donc un niveau de MIP-mapping de la texture de référence pour lequel un texel correspond à la moyenne faite sur l'ensemble des texels d'un motif.

Le niveau de MIP–mapping suivant, de plus faible résolution, n'est plus exploitable : ses texels correspondent à un filtre appliqué sur *plusieurs* motifs. Il faut donc empêcher l'utilisation des niveaux de MIP-mapping faux dans la texture de référence, ce qui est heureusement possible sur les cartes graphiques [1]. Ceci permet d'éviter l'apparition de fausses couleurs mais ne permet pas un filtrage strictement correct à cause du problème d'interpolation linéaire évoqué plus haut. La qualité du filtrage va dépendre des données employées (voir section 4.2 ci–après).

Plusieurs motifs dans un pixel : Si le point de vue est très éloigné, plus d'un motif peut être projeté dans un même pixel. Ce cas est difficile car le filtrage dépend maintenant du choix procédural des motifs. Plusieurs approches sont envisageables :

- Sur–échantillonner les texels. Si l'échelle du filtrage reste relativement petite, ceci peut être fait pour un coût raisonnable. Notons que ce sur–échantillonnage serait fait dans l'espace texture et non dans l'espace écran.
- Pré–calculer des versions filtrées de la texture procédurale. Si le point de vue est éloigné, la taille de ces versions filtrées devrait être raisonnable.
- Dans le cas des cartes de matériaux, tenter d'estimer, grâce aux distributions de motifs définies par les cartes de probabilités, la couleur moyenne de chaque matériau.

Ces solutions ne sont pas simples à mettre en oeuvre sur un processeur graphique, mais sont cependant valides, en particulier pour les logiciels de rendu. En outre, dans le cas de matériaux ou de motifs de couleurs homogènes, les défauts visuels produits sont masqués par la similarité des couleurs moyennes des motifs.

Notons que la solution habituelle pour les pavages ou les particules de texture, qui consiste à modifier la géométrie, entraîne un aliasing géométrique : il devient impossible de filtrer les textures correctement (voir chapitre 2, section 4). Dans notre cas, le filtrage *est possible*, même si le filtrage correct devient complexe dans les cas où le point de vue est très éloigné de la surface.

4.2 Discussion sur l'interpolation

Afin de simplifier la prise en charge correcte de l'interpolation linéaire, notamment dans les niveaux de la pyramide de MIP–mapping, nous proposons un nouveau type de texture, qui pourrait être pris en charge par une extension des processeurs graphiques.

Pour faciliter l'interpolation linéaire, il faudrait que les motifs soient dans des textures séparées, chacune avec une couleur de bordure. Les cartes graphiques savent gérer correctement le filtrage dans cette situation. Avec une bordure de couleur noire, il nous serait alors possible de combiner les contributions des quatre motifs voisins (dans la texture générée), chacun filtré séparément. Ce qui nous empêche de mettre en oeuvre ce mécanisme, c'est l'impossibilité de sélectionner efficacement des textures distinctes depuis un *fragment program*. Or, lorsque les motifs sont stockés côte à côte, il n'est plus possible de définir une couleur de bordure pour chaque motif ; à moins de gaspiller une grande quantité de mémoire.

[1] Avec OpenGL, ceci correspond au paramètre `GL_TEXTURE_MAX_LOD` de `glTexParameter`. Un mécanisme similaire existe sous Direct3D

L'extension que nous proposons est de pouvoir stocker les motifs dans une pile : une texture 3D dont une des dimensions permettrait de choisir le motif. La texture serait filtrée (MIP–mapping) uniquement sur les deux autres dimensions. Grâce à la couleur de bordure il nous serait alors possible d'effectuer correctement le filtrage, tant que plusieurs motifs ne se projettent pas dans un même pixel. Notons que de nombreux travaux basés sur les indirections peuvent bénéficier de cette approche, comme par exemple [KE02, Wei04].

5 Résultats

Nous avons implémenté l'ensemble des composants et diverses applications de démonstration à l'aide d'OpenGL et Cg pour carte graphique NVidia. Nos tests ont été effectués sur un prototype de carte GeForceFX NV30. Notons que depuis (nous sommes fin 2004) des cartes largement plus puissantes (en terme de puissance de calcul par pixel) sont apparues sur le marché. Les performances sont donc données ici à titre de référence mais ne sont absolument plus les meilleures performances possibles. D'autre part, nous n'avons pas cherché à optimiser le code assembleur produit par le compilateur. En outre, les cartes graphiques récentes permettent d'éliminer rapidement les pixels transparents. Nos textures correspondant à une couche de détails (distribution aléatoire de motifs) contiennent de 40% à 70% de pixels transparents.

Les temps sont mesurés en recouvrant l'écran par la texture. Le temps reste stable quelque soit la taille de la grille de pavage virtuel utilisée. De faibles variations dues au cache des données de texture sont cependant observables. Nous utilisons une précision de calcul de 32 bits dans les *fragment programs*. La table ci–dessous montre que les performances sont entre 0.4 et 0.8 Giga–instructions par secondes. Le coût est directement proportionnel au nombres de pixels dessinés.

	Pavage apériodique simple	Pavage apériodique avec carte de matériaux	Pavage apériodique avec carte de matériaux interpolée	Pavage apériodique avec carte de matériaux interpolée et transition
longueur du code	56 instr.	65 instr.	117 instr.	512 instr.
nombre d'accès aux textures	5	7	10	39
320x200	113 fps	73 fps	36 fps	5 fps
640x480	24.5 fps	15.5 fps	8.5 fps	1.1 fps

Toutes les textures ont des tailles en puissance de deux et sont stockées avec une précision de 8 bits. Les motifs ont tous une forme carrée (éventuellement avec un fond transparent) et sont de même résolution.

La figure 3.10 présente un pavage apériodique de 16 pavés de résolution 64×64 avec un contrôle utilisateur de plus en plus important. La texture générée a une résolution équivalente à une texture 4096×4096, l'espace mémoire nécessaire est très faible : les 16 motifs et la table σ. Comme montré figure 3.10a, la géométrie est complètement indépendante du pavage.

La figure 3.14 montre une distribution aléatoire de motifs. Le positionnement et la rotation aléatoire des feuilles permettent de générer des textures non répétitives. De plus cette apparence peut être contrôlée par l'utilisateur qui peut choisir la densité de chaque motif le long du terrain.

Sur la droite de la même figure, nous montrons comment notre méthode peut être utilisée pour produire des textures volumiques à faible coût. Ici, le motif est un arbre stocké dans une texture volumique de $256 \times 256 \times 256$. Le rendu est effectué par couches [MN98b].

La figure 3.15 montre l'utilisation de pavés de transition. Deux familles de 16 pavés de résolution 64×64 sont utilisées pour l'herbe et le sable. 16 pavés de transition sont définis. La texture résultante n'est bien sûr jamais stockée : elle est générée à la volée, lors du dessin du terrain à l'écran. De très larges aires de jeu peuvent donc être définies à l'aide de notre approche, sans jamais avoir à modifier la géométrie du terrain. La distribution des deux familles de matériaux est ici contrôlée par l'utilisateur mais aurait pu être générée procéduralement.

La figure 3.21 présente les animations réalisées à l'aide de notre méthode[2]. La première animation (figure 3.21a) présente des motifs dont la séquence animée est déclenchée par la distance à des cibles : les fleurs s'ouvrent lorsque les petites sphères sont proches. Les seconde et troisième animations montrent comment les paramètres d'animation peuvent être contrôlés. Pour l'animation de la figure 3.21c, nous utilisons un motif de LED animé (clignotement). Au repos, la LED est éteinte. La phrase que l'on souhaite afficher est dessinée dans une texture contrôlant le paramètre Z de la séquence (temps au repos). Les zones blanches déclenchent l'animation moins souvent que les zones noires. C'est pour cela que l'on peut voir le texte apparaître. Comme de l'aléatoire est également introduit dans les séquences d'animation, les LED forment des motifs non périodiques. Le défilement du texte est obtenu en déplaçant la texture contenant le texte. Rappelons que seule la partie visible de la texture générée est calculée (la texture complète à une résolution de 12288×1536)

La dernière animation, figure 3.21d, montre le positionnement interactif de motifs sur une surface. L'utilisateur peut déplacer les fleurs sur la surface du terrain avec la souris. Une carte de positionnement explicite est utilisée à cette fin. Le chapitre 4 détaille une simulation d'écoulement de gouttes d'eau le long d'une surface, où le positionnement explicite est contrôlé par programme plutôt que par l'utilisateur.

6 Conclusion

Nous avons proposé une méthode générique pour créer une grande variété de textures de très haute résolution, à faible coût mémoire, en combinant un ensemble de motifs fournis par l'utilisateur. La méthode n'impose aucune contrainte sur la géométrie, ce qui permet de gérer les niveaux de détails et optimisations du maillage indépendamment de la texture (point particulièrement important pour les terrains).

Notre méthode, bien que basée sur les pavages réguliers carrés, permet cependant de positionner arbitrairement des motifs sur la surface d'un terrain, sans souffrir d'alignements à grande échelle. En outre, les pavages apériodiques générés peuvent contenir différents types de

[2]Les vidéos peuvent être téléchargées sur le site http://www-evasion.imag.fr/Membres/Sylvain.Lefebvre/these

matériaux afin de briser l'homogénéité. La distribution des motifs est contrôlable spatialement. Les motifs peuvent également être animés le long de la surface et peuvent avoir une apparence animée, selon le temps, la distance à un point, ou tout autre paramètre.

Notre approche, basée sur un ensemble de composants élémentaires connectés pour former un générateur de textures procédurales, permet une très grande souplesse et un fort contrôle utilisateur. Les paramètres peuvent être soit définis à l'aide de cartes explicites, soit générés aléatoirement (mais sous contrôle). L'ensemble est implémenté sur les dernières cartes graphiques et permet des performances interactives.

Il reste cependant de nombreux points à résoudre :

- **Filtrage :** Bien que nous ayons proposé diverses approches, le filtrage des textures basées sur la composition de différents motifs reste délicat.
- **Coût de génération :** La texture est régénérée à chaque image. Or, les points de vue sont souvent cohérents d'une image à l'autre. Un grand nombre de calculs est donc sans cesse répété. Il serait souhaitable de pouvoir utiliser un cache dans lequel la texture serait générée et réutilisée par les points de vue suivants. Cependant, il faut pour cela déterminer les parties de la texture utiles à un point de vue donné.
- **Limité aux terrains et aux surfaces facilement paramétrables :** Ce dernier point constitue la principale limitation. Nous avons vu que notre méthode représente efficacement les *particules de texture* (voir chapitre 2, section 2.3), en particulier grâce au *positionnement explicite* présenté section 1.3. Cependant, ceci n'est vrai que dans le cas ou la paramétrisation est triviale. En effet, à cause des distorsions et des discontinuités, le pavage de l'espace texture ne permet pas de bons résultats sur une surface quelconque.

Le premier et dernier point (filtrage et extension aux surfaces quelconques) seront le sujet du chapitre 9. Le second point sera en partie l'objet du chapitre 6 qui présente une méthode de génération progressive de textures pour des surfaces quelconques et possiblement animées.

Nous avons également utilisé la technique présentée dans ce chapitre dans divers contextes, comme la simulation de gouttes de liquide s'écoulant sur une surface (chapitre 4) et le dessin des ombres portées par des arbres sur un terrain (chapitre 5).

chapitre 4

Application : rendu de gouttes d'eau sur surfaces

Ce chapitre présente un modèle d'habillage qui génère un effet de gouttes d'eau tombant le long d'une surface. Ce type de texture dynamique est souvent présent dans les jeux, mais des primitives géométriques (*decals*) sont habituellement utilisées pour représenter les motifs animés (voir chapitre 2, section 2.3). Ce travail est basé sur le modèle d'habillage présenté au chapitre 3 ; la surface habillée doit donc être développable si l'on veut éviter les distorsions. Le but est ici de montrer comment différents éléments peuvent être combinés pour former des effets visuels complexes et animés, gérés entièrement dans l'espace texture.

Notre approche utilise trois composants : le positionnement de gouttes d'eau sur la surface, présenté en section 1, la génération d'une texture d'humidité, présenté en section 2 et le rendu réaliste des gouttes et de la surface, présenté en section 3

Remarque : Ces travaux ont fait l'objet d'un chapitre du livre *ShaderX2 : Shader Programming Tips and Tricks* aux éditions Wordware Publishing (ISBN 1-556-229887), sous le titre *Drops of Water and Texture Sprites* [Lef03].

1 Positionnement des gouttes d'eau

Notre objectif est de simuler de petites gouttes de liquide tombant le long de la surface. Nous ne réalisons pas une simulation physique mais tentons simplement de reproduire de manière plausible le comportement observable des gouttes – notre but, ici, est de travailler sur l'habillage plus que sur la simulation.

Le mouvement des gouttes est géré de manière simple par un programme (sur CPU) qui met à jour leur position sur la surface à chaque pas de temps. Les gouttes ont différentes tailles : lorsqu'elles glissent, elles déposent de l'eau sur la surface et leur taille diminue. Lorsque deux gouttes se rencontrent, elles fusionnent en une goutte plus grosse. La direction des gouttes est globalement la même, mais, comme cela peut être observé dans la réalité, les gouttes subissent de petites variations d'orientation lorsqu'elles tombent, dues à la rugosité de la surface. L'algorithme générant le mouvement tient compte de ces phénomènes.

Afin de positionner les gouttes sur la surface sans avoir à modifier ou à introduire de géométrie, nous nous appuyons sur la méthode de *positionnement explicite* présentée au chapitre 3. Les gouttes d'eau sont représentées par une petite image qui est instanciée dans la texture aux positions simulées. Il est ainsi possible de recouvrir la surface d'un grand nombre de gouttes sans avoir à ajouter de géométrie, et à faible coût mémoire. D'un pas de temps à l'autre, la carte de positionnement explicite est mise à jour en fonction du mouvement des gouttes. Cette carte étant de faible résolution, la mise à jour est très rapide. Pour que les gouttes puissent se rencontrer, nous utilisons en fait plusieurs couches de texture (rappelons que le recouvrement de motifs n'est pas possible avec la méthode présentée chapitre 3). En pratique, nous utilisons environ 6 couches de texture (il y a donc 6 cartes de positionnement explicite).

2 Texture d'humidité

Lorsque les gouttes tombent le long de la surface elles déposent du liquide. Les parties humides de la surface s'assombrissent et deviennent plus brillantes, ce qui correspond à assombrir leur couleur diffuse et à augmenter leur coefficient spéculaire. Ensuite, la surface sèche et retrouve progressivement son aspect d'origine.

Afin de représenter cet effet, nous utilisons une *texture d'humidité*, qui encode en tout point de la surface l'humidité présente. La texture d'humidité contrôle les paramètres de rendu de la surface (voir section 3). Pour générer cette texture, nous dessinons dedans, à chaque pas de temps, les gouttes présentes sur la surface. Elles sont affichées sous la forme de points blancs. Afin que des traînées humides apparaissent, on procède à une relaxation : la texture de l'étape précédente est utilisée comme fond pour la nouvelle étape, légèrement assombrie pour simuler un effet de sèchement. La résolution de la texture d'humidité doit être suffisante pour capturer la présence des gouttes les plus petites. La figure 4.1 présente une texture d'humidité obtenue après quelques secondes de simulation.

3 Rendu

Afin d'obtenir un effet convaincant, il nous faut effectuer un rendu réaliste combinant la texture de fond (celle sur laquelle les gouttes évoluent) et les gouttes d'eau.

Pour afficher la surface, nous utilisons simplement du *bump mapping* [Bli78]. Le coefficient spéculaire et la couleur de la texture de fond sont modulés par la texture d'humidité (voir section 2). Les gouttes d'eau individuelles sont dessinées avec un effet de rendu plus complexe, afin de simuler transparence et réfraction. Le motif de goutte d'eau que nous stockons est en fait une carte de hauteur, représentant la hauteur de liquide. La carte de positionnement explicite (voir chapitre 3) permet de déterminer si une goutte est présente en un point de la surface. Si une goutte d'eau est présente, on obtient donc la hauteur de liquide en ce point. Nous appliquons alors une formule ad–hoc qui permet de simuler un effet de réfraction. L'idée est similaire au *parallax mapping* présenté chapitre 2, section 2.1.7.4 : un petit déplacement est appliqué aux coordonnées de texture utilisées pour la texture de la surface. Ce petit déplacement est calculé en

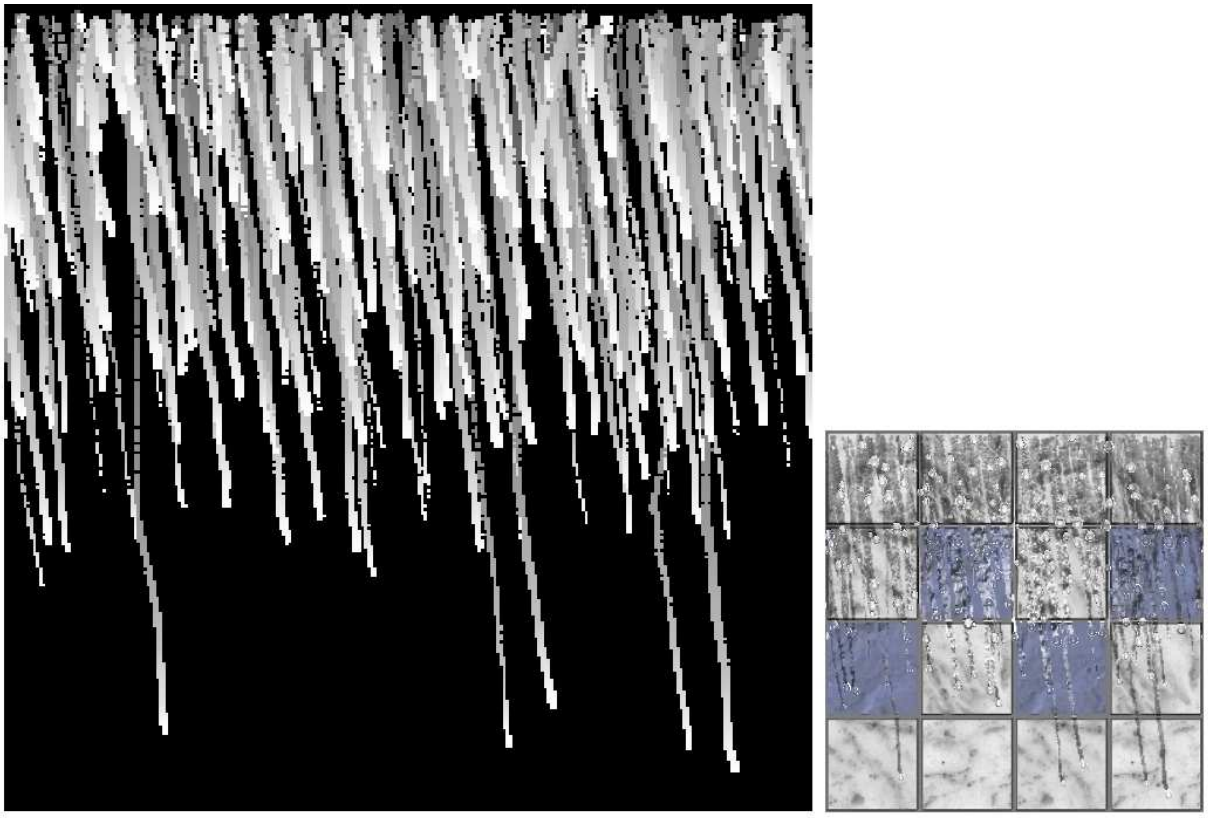


FIG. 4.1 – Texture d'humidité de la surface.
Cette texture est mise à jour durant la simulation. Les gouttes laissent des traînées humides en se déplaçant. Le niveau de gris représente le niveau d'humidité. Le séchage est obtenu en modulant la contribution de l'étape précédente lors de la mise à jour. Cette texture est utilisée pour contrôler la couleur et le coefficient spéculaire lors du rendu final de la surface.

fonction de la hauteur de liquide, du vecteur de vue, et d'un paramètre permettant de simuler l'indice de réfraction. Cet effet, physiquement très approximatif, permet de simuler à moindre coût un effet de réfraction convaincant sur *chacune* des gouttes présentes sur la surface. La figure 4.2 présente l'effet obtenu pour une goutte avec différentes valeurs du paramètre contrôlant l'indice de réfraction. Le résultat final de notre modèle d'habillage est montré figure 4.3. La vidéo et le programme de démonstration peuvent être téléchargés sur le site web de cette thèse[1].

4 Conclusion

Cet effet montre qu'il est possible de générer des habillages animés complexes pour les applications interactives. La complexité est entièrement produite par le modèle d'habillage , la géométrie restant très simple. Ceci permet d'avoir un effet de haute qualité visuelle avec de très

[1] http://www-evasion.imag.fr/Membres/Sylvain.Lefebvre/these

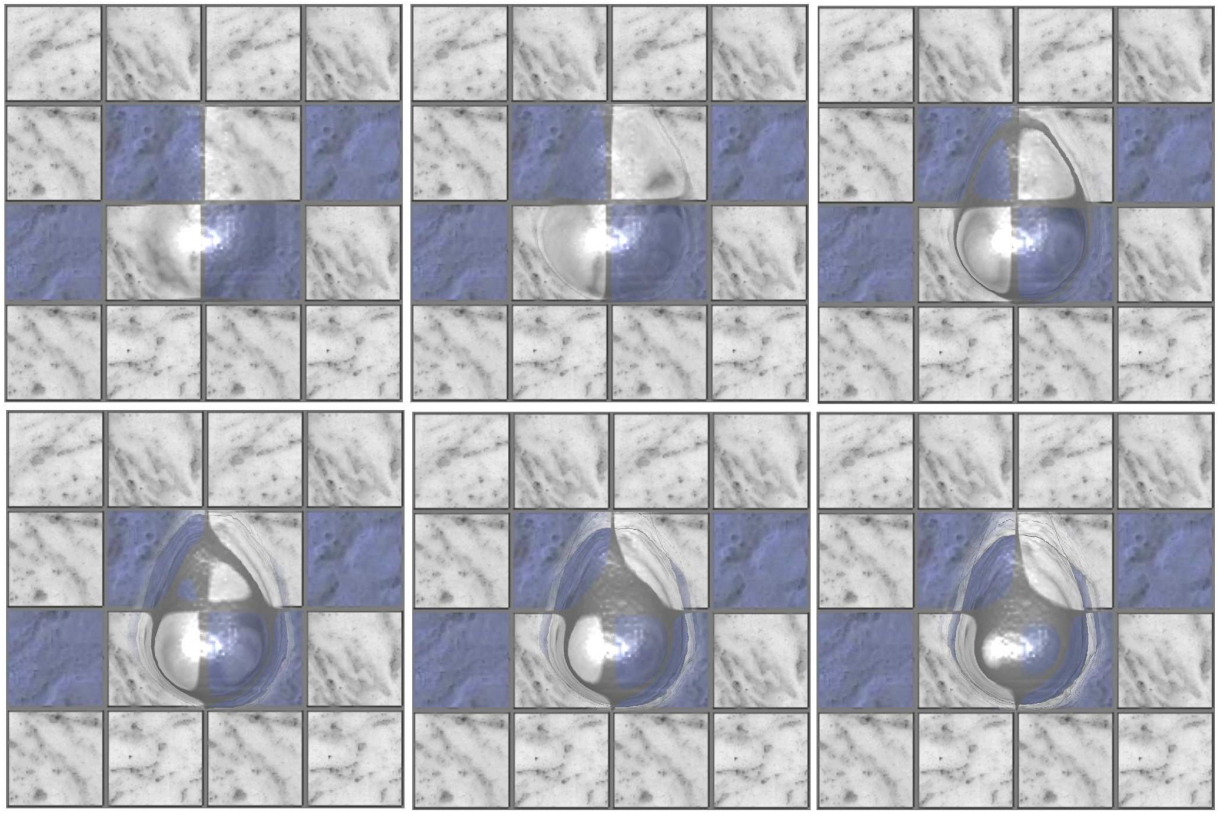


FIG. 4.2 – Effet de réfraction des gouttes.

Chaque goutte est rendue avec un effet de réfraction. L'effet dépend du point de vue et d'un paramètre simulant l'indice de réfraction. Cet effet est obtenu en appliquant un petit déplacement sur les coordonnées de texture utilisées pour la texture du fond. Il est approximatif, mais permet un effet visuel convaincant à faible coût, ce qui est le but recherché. Le paramètre contrôlant l'indice de réfraction augmente de gauche à droite et de haut en bas.

bonnes performances, de façon transparente pour l'application sur CPU (on préfère généralement consacrer la puissance CPU aux tâches de plus haut niveau). Notons cependant que tout comme la méthode du chapitre 3, la texture animée générée souffre d'un problème de filtrage pour des points de vue très éloignés.

Nous présenterons au chapitre 8 une autre simulation d'écoulement de liquide, fonctionnant cette fois sur une surface non paramétrée et entièrement gérée par le processeur graphique.

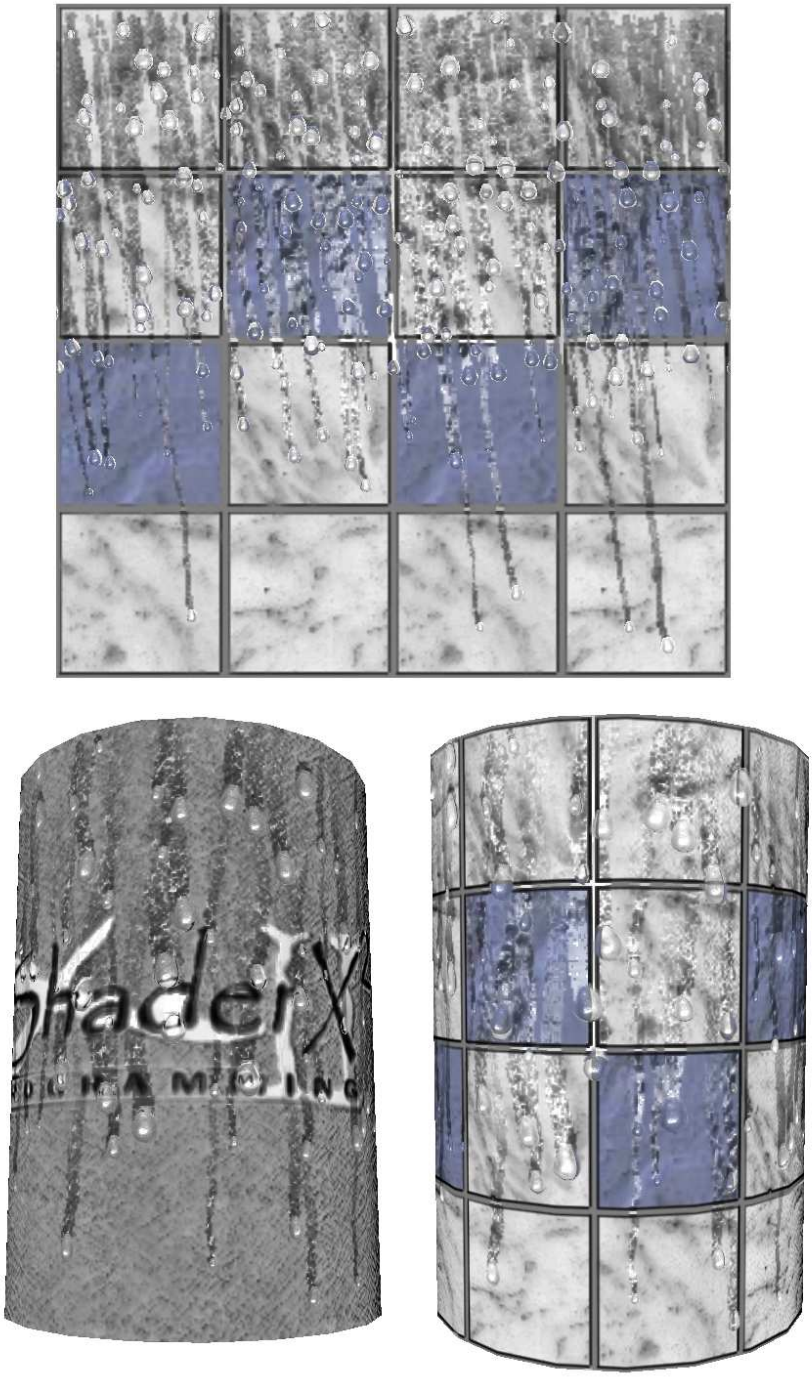


FIG. 4.3 – Simulation de gouttes tombant le long d'une surface.

Les gouttes sont positionnées grâce à notre méthode de positionnement explicite (voir chapitre 3). Le coût mémoire est très faible : seul un motif de goutte est stocké en mémoire, ainsi que les cartes de positionnement qui sont de faible résolution. La résolution du motif de goutte est arbitraire. La géométrie n'est pas modifiée. Dans l'image de gauche seuls deux triangles sont nécessaires. Cependant la méthode ne fonctionne sans distorsion que sur les surfaces développables ou quasi–planaires. Les performances obtenues sont d'environ 85 images par secondes sur une GeForce 6800 GT.

chapitre 5

Application : ombres portées par un grand ensemble d'objets

Ce chapitre, tout comme le précédent, présente des travaux effectués à partir de la méthode du chapitre 3. Il s'agit ici de représenter l'ombre projetée par un grand ensemble d'objets sur un terrain. Ces travaux, menés dans le contexte du rendu interactif de forêt, sont encore assez préliminaires et les problèmes posés n'ont pas tous été traités. Cependant, les premiers résultats sont encourageants et l'approche devrait bénéficier des développements techniques récents[1].

Nous présenterons tout d'abord, en section 1, le contexte dans lequel cette recherche a été menée et notamment l'approche que nous utilisons pour générer efficacement une forêt à la surface d'un terrain. Nous verrons ensuite, en section 2, le modèle d'habillage que nous utilisons pour créer les ombres des arbres sur le terrain.

Remarque : Ces travaux ont été effectués en collaboration avec le professeur Przemek Prusinkiewicz, dans le cadre d'un séjour à l'université de Calgary (Canada) financé par une bourse EURODOC de la région Rhônes–Alpes.

1 Le contexte

Avant de pouvoir s'intéresser à la représentation des ombres des arbres, il faut tout d'abord pouvoir afficher efficacement une forêt. Or, l'une des principales difficultés pour le rendu interactif de forêt concerne la grande quantité de données à transférer entre le processeur central (CPU) et le processeur graphique (GPU). On considère qu'il faut environ 30000 arbres par kilomètre carré pour constituer une forêt dense. Si l'on doit afficher dix kilomètres carrés de forêt, envoyer un ordre de dessin par arbre génère déjà un transfert de données très conséquent (sans même parler des arbres qui sont eux–mêmes très complexes). En outre, positionner les arbres un par un constitue un travail long et pénible pour les artistes.

La méthode que nous avons développée pour générer et afficher la forêt repose sur les idées présentées au chapitre 3 : l'instanciation et la génération procédurale à la volée sont combinées

[1] En particulier, le support de l'instanciation géométrique par les cartes graphiques .

pour réduire les coûts à la fois en terme d'occupation mémoire et en terme de temps de création. Seul un petit nombre d'objets sont stockés en mémoire et positionnés procéduralement sur le terrain. Des variations d'apparence des arbres (taille, couleur, etc ...) sont générées automatiquement (mais le contrôle reste possible).

La génération de la forêt à la surface du terrain n'est pas un modèle d'habillage de surface puisque la géométrie des arbres est ajoutée (voir section 1.1.3). Notons cependant qu'il s'agit bien du concept d'habillage : la forme simple du terrain est enrichie d'un aspect complexe (la forêt). Le modèle d'habillage de surface qui fait l'objet de ce chapitre concerne le dessin de l'ombre des arbres sur la surface du terrain. Cependant, avant de le décrire, il nous faut introduire la méthode employée pour générer la forêt et positionner les arbres.

1.1 Génération de la forêt

1.1.1 Principe

Seul un petit carré de forêt est stocké en mémoire et va être répété le long du terrain pour générer la forêt complète. Le terrain est tout d'abord recouvert d'une grille régulière (voir figure 5.1). Lors du rendu, on détermine (à l'aide de la carte graphique) les cases visibles pour le point de vue courant. Ce mécanisme permet d'éviter de dessiner les cases de forêt situées en dehors de l'écran ou masquées par des collines. Ensuite, nous demandons au processeur graphique de dessiner le carré de forêt sur chacune des cases visibles. Nous le programmons pour qu'il introduise des variations dans l'apparence des arbres (taille, couleur, orientation), et qu'il supprime certains arbres pour changer la densité de la forêt (voir section 1.1.2).

Un seul ordre de dessin est donc envoyé pour chaque case de forêt présente sur le terrain (et visible). Chaque ordre de dessin provoquant l'affichage de multiples arbres, la quantité de données envoyée au processeur graphique est largement réduite. L'introduction de variations dans l'apparence des arbres permet d'éviter l'apparition de répétitions dans leur agencement. Le contrôle en densité permet de sculpter facilement des carrières ou des routes dans une forêt épaisse, et de changer la densité le long des pentes et selon l'altitude. Ainsi de grandes variations d'apparence sont obtenues, tout en utilisant très peu de mémoire.

1.1.2 Le carré de forêt

Le carré de forêt contient, en fait, un ensemble de points, appelés *graines*. Chaque graine représente une position à laquelle un arbre *peut* apparaître ainsi que d'autres paramètres tels que la taille, l'orientation et la couleur de l'arbre. En pratique, le carré de forêt et les graines sont un tableau de points (*vertex buffer*) stocké dans la mémoire du processeur graphique. Nous utilisons un *vertex program* pour modifier les paramètres des graines lors du rendu. En particulier, nous programmons le processeur graphique pour obtenir une variation aléatoire des paramètres de couleur, de taille et d'orientation. Afin de contrôler la desnité de la forêt, certaines graines sont supprimées (elles sont simplement envoyées hors du champ de vision).

1.1.3 Création des arbres à partir des graines

Chacune des graines du carré de forêt est ensuite remplacée par un arbre (on *instancie* les arbres). Les arbres distants sont dessinés à l'aide de *billboards* (voir chapitre 2, section 6.2) alors que les arbres proches sont dessinés à l'aide de géométrie (ce choix est effectué sur chaque case de forêt). Pour les billboards, il suffit de positionner un polygone texturé sur chaque graine. Ceci peut être effectué directement par la carte graphique[2]. Pour la géométrie, les récentes extensions d'instantiation géométrique vont permettre d'effectuer cette opération directement en sortie du *vertex program*. Ceci n'était pas possible à l'époque où nous avons mené nos travaux, et nous devions donc effectuer cette opération sur le CPU, ce qui explique en partie les performances relativement basses que nous obtenons (environ 5 images par secondes avec les ombres, voir figure 5.2). Cependant les récents progrès techniques des processeurs graphiques devraient nous permettre une implémentation beaucoup plus efficace. Quelques résultats de notre méthode sont montrés figure 5.2.

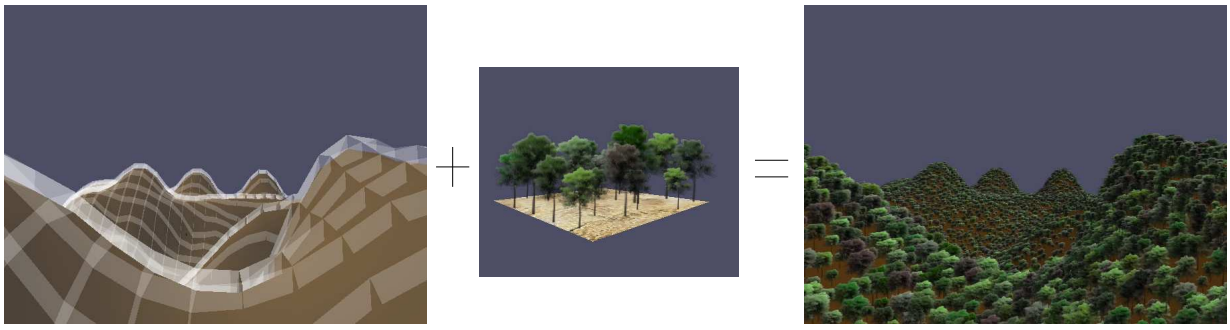


FIG. 5.1 – Génération de forêt.
A partir d'une grille couvrant le terrain (à gauche) et d'un carré de forêt (au milieu), une couverture forestière est générée (à droite). Seul le carré de forêt est stocké en mémoire. Lorsque la grille est remplie, des variations aléatoires d'orientation, taille et couleur des arbres permettent de masquer les répétitions.

2 Ombres d'objets sur un terrain

Dans ce contexte il nous fallait élaborer une méthode permettant de représenter les ombres des arbres sur le terrain. Notons que ce ne sont pas les seules ombres à prendre en compte, il en existe cinq types :

- ombre des arbres sur le terrain (objet de ce chapitre),
- ombre de chaque arbre sur ses voisins,
- ombre du terrain sur les arbres,
- auto–ombrage des arbres,
- auto–ombrage du terrain.

[2]Par exemple grâce à l'extension OpenGL GL_NV_point_sprite.

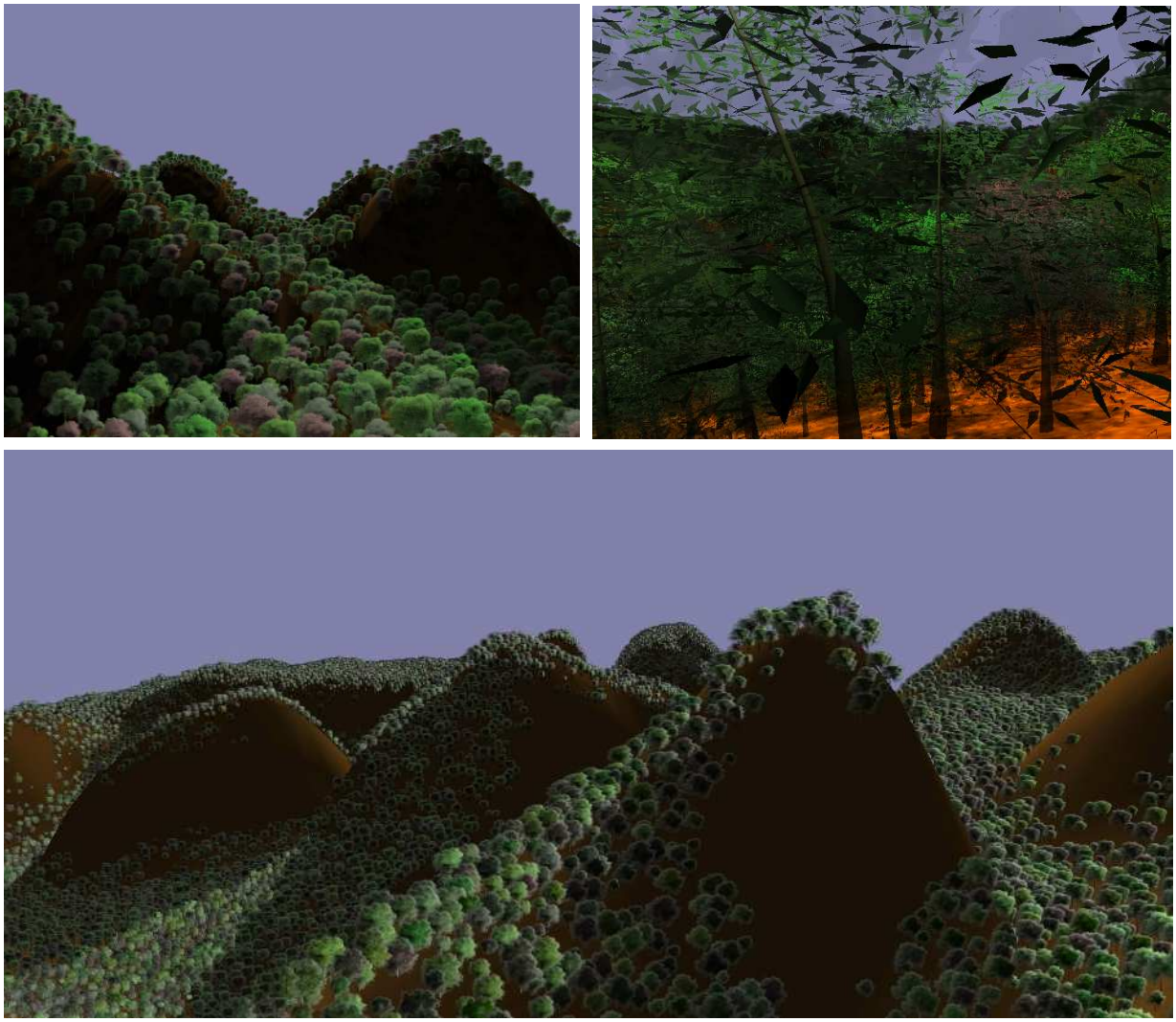


FIG. 5.2 – Résultats de notre méthode de génération et rendu interactif de forêt.

Résultats obtenus avec notre méthode de rendu de forêt. Le rendu est effectué à environ 5 images par seconde sur un prototype de carte NVidia GeForce FX 5900.

L'ombre des arbres sur le terrain est cependant plus difficile à représenter étant donné le grand nombre d'objets et la génération procédurale de la forêt. Notons que le problème de prendre en compte l'ombre d'un arbre sur ses voisins est relativement proche : il s'agit dans les deux cas de déterminer si l'un des arbres bloque la lumière en un point donné. Notre objectif est d'obtenir des motifs d'ombre très détaillés, permettant notamment de distinguer des enchevêtrements de branches.

Nous travaillons sous l'hypothèse que les ombres d'un grand ensemble d'objets n'ont pas besoin d'être exactes pour tromper l'oeil humain et sembler réalistes : dans une forêt relativement dense, il est généralement difficile d'identifier les branches qui projettent telle ou telle

ombre. Le domaine de validité de cette hypothèse reste cependant à étudier, probablement au travers d'évaluations avec des utilisateurs. Néanmoins, elle nous paraît raisonnable dans le cadre d'applications interactives, et les premiers résultats le confirment.

Notre approche est basée sur la technique de cartes d'ombres (*shadow map*) [Wil78]. Cette méthode commence par dessiner l'objet produisant une ombre, en noir sur fond blanc, depuis le point de vue de la lumière. L'image obtenue est ensuite projetée sur la surface du terrain, comme illustré figure 5.3. Utilisée directement sur une forêt la méthode ne permet pas de représenter des ombres avec une haute résolution, car l'ensemble des ombres des arbres de la forêt devrait être dessiné dans la même image (il faudrait une résolution gigantesque pour capturer les détails de toutes les ombres).

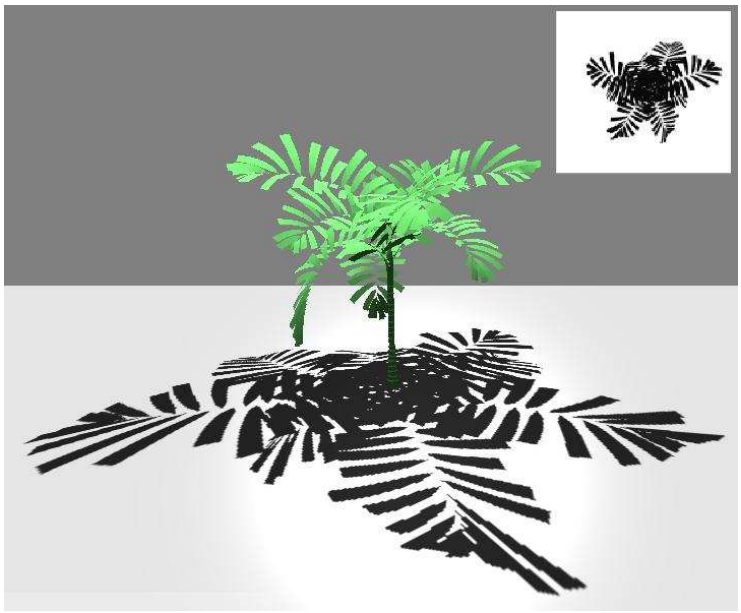


FIG. 5.3 – Carte d'ombre.
En haut à droite : *L'arbre est dessiné en noir sur blanc depuis le point de vue de la lumière (située au dessus) dans la carte d'ombre.* Au centre : *La carte d'ombre est projetée sur le sol pour dessiner l'ombre de l'arbre.*

Notre approche est d'instancier dans une large texture, pour chaque arbre, une image représentant l'ombre (détaillée) d'un arbre, selon la méthode décrite au chapitre 3. Cette large texture (mais de faible coût mémoire) sera ensuite projetée sur le terrain pour former les ombres des arbres, comme une carte d'ombre (voir figure 5.4). Les ombres sont positionnées avec la même procédure que les arbres : elles seront donc instanciées aux bonnes positions. Afin de permettre aux ombres de différents arbres de se recouvrir, nous utilisons plusieurs couches de textures ainsi générées (voir chapitre 3, positionnement explicite). Si trop d'ombres se recouvrent en un même point, certaines ombres seront ignorées. Cependant, il est fort probable que la zone soit fortement assombrie par les ombres déjà présentes, et l'absence de l'ombre ignorée ne sera pas repérable facilement. Il faut également éviter qu'une ombre apparaisse sur les deux flancs d'une montagne, lorsque la lumière est rasante. Nous stockons, à cette fin, une profondeur avec chaque instance

d'image d'ombre. Cette profondeur est la distance entre la base de l'arbre et la lumière. Elle est comparée, lors du dessin, avec la distance entre la lumière et le terrain afin de n'afficher l'ombre que si la surface est exposée à la lumière.

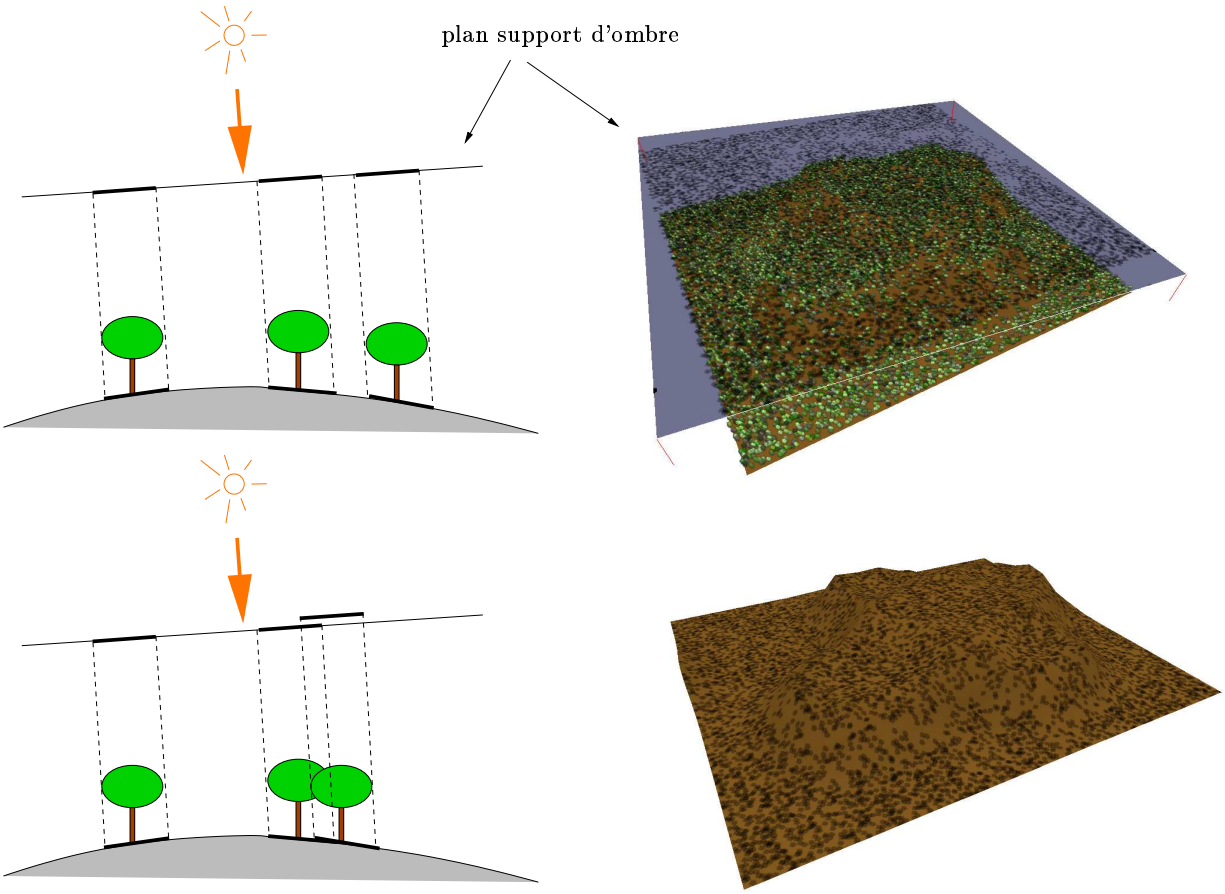


FIG. 5.4 – Principe de projection d'ombres.

En haut à gauche : *L'image représentant l'ombre des arbres est répétée dans une texture appliquée au plan support d'ombre. Cette texture est ensuite projetée sur le terrain, dans la direction de lumière, pour représenter les ombres des arbres.* En haut à droite : *Le plan support d'ombre est situé au dessus du terrain et recouvre toute la forêt.* En bas à gauche : *Les ombres de différents arbres peuvent se recouvrir. Plusieurs cartes d'ombre sont utilisées dans ce cas.* En bas à droite : *Les ombres projetées sur le terrain, sans les arbres.*

L'utilisation de la méthode décrite au chapitre 3 permet de stocker les images des ombres avec une très haute résolution. Ces images étant instanciées, le coût mémoire reste faible. Les fins détails des ombres sont ainsi bien capturés.

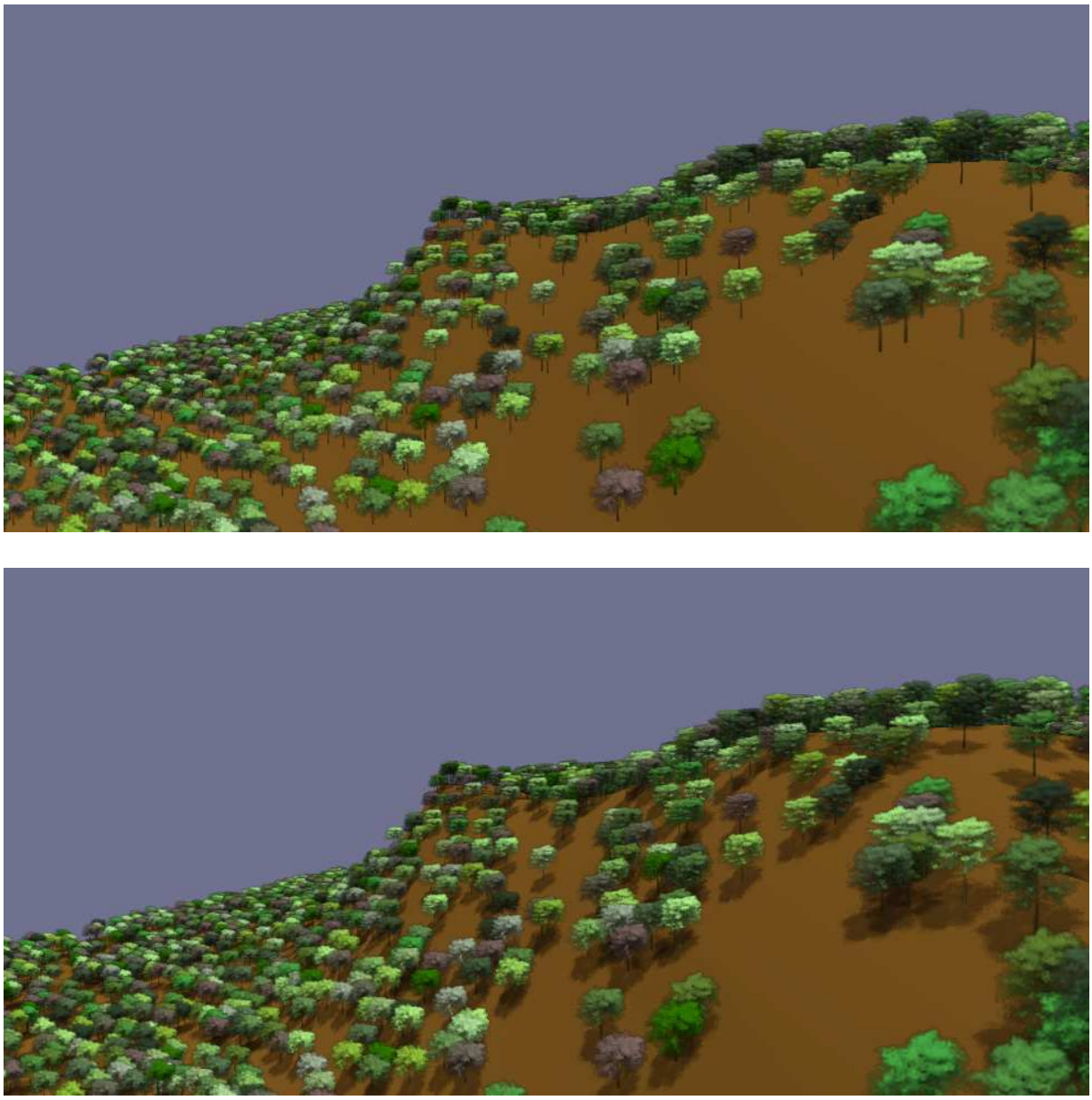


FIG. 5.5 – Ombres représentées par notre méthode.

La même scène affichée avec et sans ombres. Remarquez comme les ombres augmentent le réalisme de l'image. Les ombres ne sont pas exactes, mais perceptuellement convaincantes car elles sont placées à la bonne position et ont globalement la bonne forme.

3 Conclusion

Le rendu réaliste de forêts est un domaine vaste, posant de nombreux défis. Nous avons ici proposé quelques directions pour mieux représenter les forêts et les ombres projetées par un grand ensemble d'objets sur un terrain. Cependant, il reste un gros travail à effectuer sur

l'éclairage (autres types d'ombres, auto–ombrage des arbres) avant d'obtenir des forêts très réalistes. Cette méthode a été développée dans le contexte du rendu de forêt mais est applicable à d'autres cas, comme par exemple l'ombre d'une foule sur le sol et les bâtiments.

Du point de vue des ombres, la méthode fournit de bons résultats : elles sont très détaillées et le coût mémoire est faible. Cependant, les lumières rasantes posent des problèmes de robustesse : trop de recouvrement apparaît et la quantité d'instances en un même point devient trop importante. Une direction intéressante pourrait être alors de remplacer un ensemble d'ombres se recouvrant, par une seule image, ayant une densité d'ombre équivalente.

Ces travaux ne sont pas terminés et doivent être explorés plus avant afin d'obtenir des résultats robustes, mais l'approche est validée par nos premiers résultats et nous espérons que de futures collaborations nous permettront de conclure cette recherche.

Troisième partie

Textures de haute résolution plaquées sur surfaces quelconques

chapitre 6

Gestion de textures de haute résolution plaquées sur des surfaces quelconques

Les textures utilisées par les applications modernes sont de plus en plus détaillées. Avec l'accroissement des capacités d'affichage et l'apparition de techniques de rendu mélangeant plusieurs couches de textures, les limites de capacité mémoire sont très vite atteintes.

La plupart des systèmes de rendu imposent en effet de charger l'ensemble des données nécessaires à l'affichage d'une scène depuis tous les points de vue possibles. Néanmoins, le *chargement progressif* propose une alternative : dans cette approche, les données sont chargées uniquement au fur et à mesure des besoins, en fonction du point de vue courant et lorsque l'utilisateur se déplace. En effet, un point de vue donné n'utilise pas l'ensemble des textures (certains objets sortent tout ou en partie de l'écran, se cachent les uns les autres). Quant aux textures utilisées, elles sont rarement entièrement visibles, et peuvent être nécessaires seulement à basse résolution si l'objet est loin (voir chapitre 2, section 4). Afin que le chargement progressif ne pénalise pas les performances et n'introduise pas de ralentissements, il faut veiller à minimiser les transferts en chargeant uniquement les données utiles, en appliquant des stratégies de cache pour exploiter la cohérence temporelle et spatiale, et en ayant éventuellement recours à des algorithmes de compression.

Ces diverses approches ont déjà été explorées. Cependant la plupart des recherches se concentrent sur le cas des terrains. Or, les textures détaillées sont désormais également utilisées sur des objets aux géométries complexes (par exemple, des personnages). Le problème crucial de savoir quelles parties d'une texture sont utiles, et à quelle résolution, n'est pas résolu efficacement dans le cas général (géométrie complexe, éventuellement animée, paramétrisation arbitraire). D'autre part, ces diverses approches ne sont pas regroupées dans un ensemble cohérent.

Nous proposons tout d'abord, dans ce chapitre, un nouvel algorithme de détermination des parties visibles d'une texture. Cet algorithme fonctionne sur des géométries arbitraires, avec des paramétrisations quelconques. Aucune modification de la géométrie n'est nécessaire – en particulier, la géométrie peut être optimisée pour le rendu indépendamment de la texture. Nous construisons ensuite, autour de ce nouvel algorithme, une architecture de chargement progressif de texture permettant d'unifier les solutions usuelles (notamment cache de texture et compression). Notre architecture permet également de prendre en charge les textures procédurales : lors-

qu'elles ne sont pas pré–calculées, le temps nécessaire à leur génération provoque en effet un délai analogue au temps d'accès à des données de texture stockées en mémoire. Dans notre cas, une portion de texture n'est chargée (ou générée) que si elle apparaît à l'écran, et à la résolution utile. Elle sera, de plus, conservée dans un cache en mémoire, pour exploiter la cohérence temporelle. Notre architecture, bien que générique et applicable à tout système de rendu, est conçue pour les cartes graphiques récentes.

Nous commençons par présenter, en section 1, les travaux existant sur le chargement progressif de texture. La section 2 explique comment nous organisons les données de texture afin de les manipuler avec notre architecture de chargement progressif, présentée en section 3. La section 4 introduit notre nouvel algorithme de détermination des parties utiles de la texture. La section 5 explique comment nous réalisons notre cache de texture, et la section 6 comment les données de texture sont produites. Nous présentons nos résultats en section 7.

Remarque : Ces travaux ont fait l'objet d'un rapport de recherche INRIA (rapport n°5210 [LDN04]) et ont été développés en collaboration avec Jérome Darbon, doctorant à l'ENST.

1 Travaux sur le chargement progressif

Nous discutons ici les travaux portant sur le chargement progressif de textures. Le problème qui nous intéresse principalement est la détermination des parties de la texture utiles au point de vue courant. Ceci va nous permettre de mettre en évidence les limitations des méthodes existantes dans le contexte des géométries quelconques.

L'architecture de *Clipmaps* introduite par Tanner *et al.* [TMJ98] utilise un centre d'intérêt dans l'espace texture et un rayon d'intérêt fournit par l'utilisateur pour déterminer les parties utiles de la texture. Les données sont chargées autour du centre d'intérêt avec une résolution plus faible au fur et à mesure que la distance au centre s'accroît. Au–delà du rayon d'intérêt, la texture n'est plus chargée. Malheureusement, déterminer le centre et le rayon d'intérêt de manière à englober le point de vue n'est pas toujours aisé. De plus, cette méthode ne peut fonctionner efficacement sur des paramétrisations arbitraires : si deux petites zones distantes dans la texture sont utilisées pour un même point de vue, un très large sous ensemble doit être chargé (il faut inclure les deux zones dans un cercle). Cependant, la méthode est efficace sur les terrains et ne nécessite pas de modification de la géométrie.

Les MP-grids de Hüttner [Hut98] n'utilisent pas d'entrée utilisateur spécifique. La texture est ici découpée en une grille régulière dont les cases sont chargées progressivement. Pour déterminer les parties utiles pour le point de vue courant, la géométrie associée à chaque case de texture est calculée. Un test géométrique (visibilité sur la boite englobante) est alors effectué pour déterminer si la case est utilisée, et à quelle résolution. Cline *et al.* [CE98] déterminent également les parties utiles de la texture à l'aide d'un calcul géométrique sur chaque polygone. Les polygones doivent être découpés en fonction des cases de texture pour le rendu. Les deux méthodes requièrent de nombreux calculs géométriques durant le rendu. Au–delà des considérations de performance, ceci rend également difficile l'animation de la géométrie des objets. Ces approches,

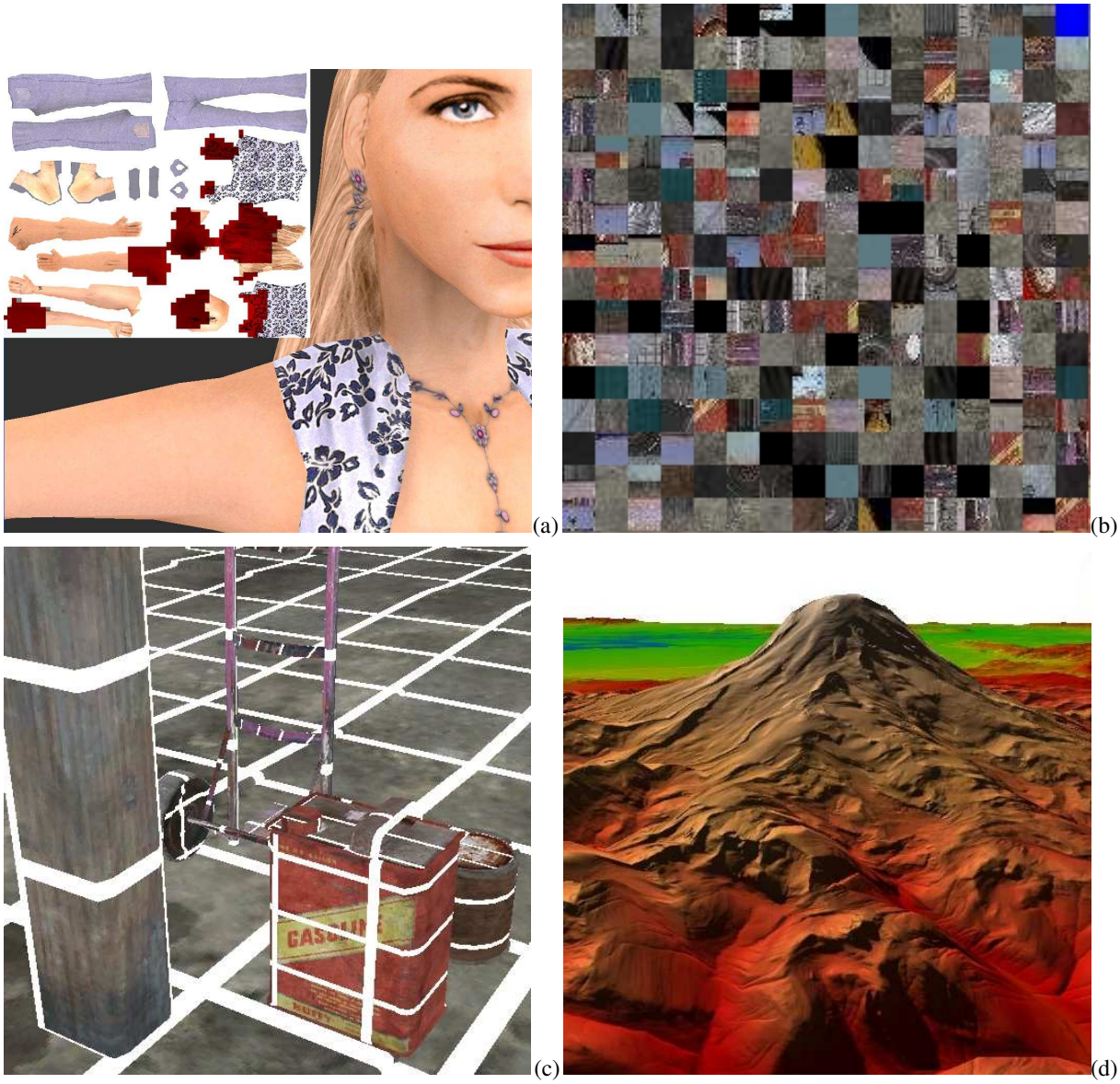


FIG. 6.1 – Présentation d'ensemble de notre architecture de gestion de textures.

(a) *En rouge (sombre) : Parties de la texture utiles pour le point de vue courant.* ***(b)*** *La texture est découpée en cases stockées dans la mémoire texture sur la carte graphique pour le rendu (8 Mo)* ***(c)*** *Les cases de textures sont ensuite utilisées pour recomposer uniquement la partie visible de la texture (la texture entière pèse 128 Mo). La géométrie n'est pas modifiée : notre architecture fonctionne dans l'espace texture.* ***(d)*** *Survol temps réel d'un terrain avec une texture de 1 Go.*

conçues pour le cas spécifique des terrains, ne sont pas aisément extensibles aux modèles arbitraires.

Dollner *et al.* [DBH00] maintiennent une hiérarchie contenant à la fois le modèle géométrique du terrain et la texture. Cette méthode exploite également des propriétés spécifiques des terrains et ne peut pas être facilement étendue aux modèles arbitraires.

Goss *et al.* [GY98] proposent une modification des cartes graphiques afin de trouver les cases de texture devant être chargées pour chaque image. Chaque fois que le processeur graphique accède à la mémoire texture, il marque la case correspondante comme nécessaire. Les cases utilisées par le plus de pixels sont chargées en premier lieu. Cependant cette méthode fonctionne uniquement sur des textures de taille modérée puisqu'une variable d'état doit être stockée en mémoire pour chaque case de texture. De plus la modification nécessaire est relativement importante (elle ne peut pas être réalisée par simple programmation de la carte graphique) et n'a pas – a notre connaissance – été intégrée dans un processeur graphique.

Enfin, notons que les cartes graphiques actuelles intègrent un système permettant d'échanger les textures entre la mémoire de l'ordinateur et la mémoire texture. Lorsque la mémoire texture n'est pas assez grande, les textures vont être échangée via le bus entre les deux mémoires. Une texture déplacée l'est entièrement. Ce cas est détecté en comparant les identifiants des textures déjà en mémoire aux identifiants des textures utilisées par la géométrie : le point de vue n'est pas pris en compte. De plus, le chargement n'est pas progressif et n'est pas contrôlable par l'utilisateur.

2 Organisation des données de texture

Notre architecture de chargement progressif fonctionne pour le placage de texture standard. La texture est stockée comme une pyramide de Gauss, similaire à la pyramide utilisée par l'algorithme de MIP–mapping (voir chapitre 2, section 4.3.1.1). Chaque niveau de la pyramide est découpé par une grille. Les cases des différents niveaux ont la même résolution (voir figure 6.3). Les *cases filles* d'une case de texture T sont les quatre cases correspondant à la même portion de texture au niveau suivant, plus détaillé, de la pyramide. T est la *case parente*.

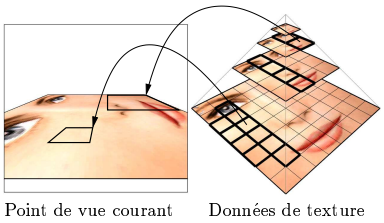


FIG. 6.3 – Structure pyramidale de la texture

Le problème va donc être de déterminer les cases utiles au point de vue courant, de les charger progressivement, d'appliquer un cache sur les données et de pouvoir utiliser cette structure comme une texture standard.

Bien que notre architecture puisse gérer l'ensemble de la pyramide, le placage de texture standard est extrêmement efficace sur des textures de taille raisonnable. En pratique nous utilisons notre système seulement sur les derniers niveaux de la pyramide, là où la résolution devient trop importante (typiquement supérieure à 2048×2048). Les premiers niveaux sont utilisés comme une texture habituelle, notre architecture s'active lorsque les derniers niveaux sont utiles au point de vue courant (voir figure 6.4).

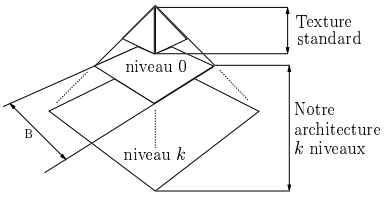


FIG. 6.4 – Notre architecture gère uniquement les derniers niveaux

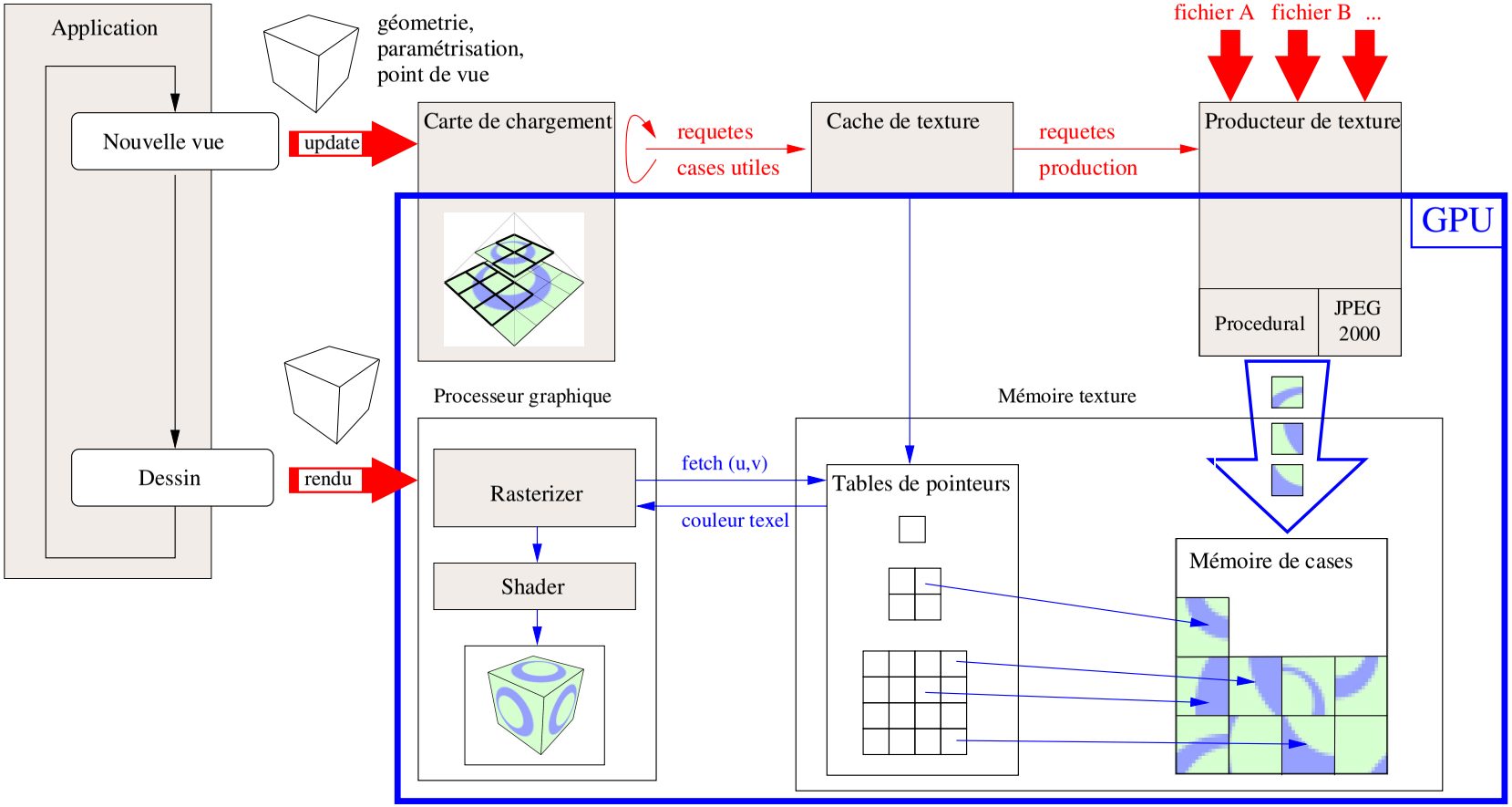


FIG. 6.2 – Organisation de notre architecture

Le module carte de chargement calcule les cases de texture utiles à partir du point de vue courant et de la géométrie. Le cache de texture gère le stockage des cases de texture. Le producteur de texture produit les données de texture de manière asynchrone. Il lit ses données depuis un média de masse et les décode en mémoire texture. Les traitements effectués par le processeur graphique apparaissent en bleu.

3 Vue d'ensemble de notre architecture

Notre architecture fonctionne de la manière suivante : à chaque nouvelle image, les cases de texture utiles (devant être chargées) sont déterminées grâce au nouveau point de vue. Elles sont ensuite chargées de manière asynchrone. Notre architecture peut être utilisée à tout moment comme une texture standard. Les cases de texture non encore chargées sont simplement remplacées par une version basse résolution de la texture.

L'architecture est organisée en trois modules (voir figure 6.2). Le premier module, la *carte de chargement de texture* détermine à partir du point de vue courant et de la géométrie quelles cases de la texture sont utiles (à tous les niveaux de la pyramide). Cette information est envoyée au second module, le *cache de texture*. Si la case est déjà présente dans la mémoire du cache (appelée *mémoire de cases*), elle est directement utilisée. Sinon, le cache lui réserve un emplacement et envoie une requête de chargement au dernier module, le *producteur de texture*. Celui–ci charge, ou *produit*, de manière asynchrone les données. Il peut simplement les charger depuis le disque, les décompresser directement en mémoire texture ou bien générer une texture procédurale.

Notre architecture présente plusieurs avantages. Tout d'abord elle est **transparente à l'utilisateur**. Celui–ci envoie seulement le point de vue courant et la géométrie utilisée pour ce point de vue. Les calculs géométriques effectués pour la détermination des cases utiles sont tous faits par le processeur graphique, là où ils sont le plus efficaces. D'autre part la notion de **producteur de texture** permet d'englober les textures standard, compressées et procédurales dans une même architecture. L'étape de production de texture peut être effectuée directement par le processeur graphique en mémoire texture : le transfert mémoire concerne alors uniquement des données compressées ou les paramètres d'une texture procédurale, minimisant ainsi les transferts coûteux entre mémoire centrale (CPU) et mémoire texture (GPU).

4 Détermination des cases de texture utiles : carte de chargement de texture

La carte de chargement détermine quelles cases de la texture sont utiles pour un point de vue donné. C'est une pyramide de tableaux 2D, chaque niveau correspondant à un niveau de la pyramide de texture. Une case d'un tableau correspond à une case de texture. Elle contient les informations sur l'utilisation de cette dernière par le point de vue courant. La figure 6.5 montre deux niveaux d'une carte de chargement dans un cas simple. Chaque case contient 3 valeurs :

- Un marqueur d'utilisation. Si ce marqueur est présent, la partie de la texture couverte par la case est utilisée par le point de vue courant. Ceci ne signifie par pour autant que cette case de texture doit être chargée : cela va dépendre du niveau de détail auquel elle est utilisée.
- La valeur de niveau de détail v_{min} (voir ci–après).
- La valeur de niveau de détail v_{max} (voir ci–après).

Les valeurs de niveau de détail (ou *valeurs LOD*) v_{min} et v_{max} correspondent aux niveaux de détail minimal et maximal, auxquels les pixels affichés à l'écran utilisent une case de texture. Une valeur LOD est comprise dans l'intervalle $[0,k]$ où k est le nombre de niveaux gérés par notre méthode (voir figure 6.4). Une case de texture représente elle–même un certain niveau de

détail, déterminé par son niveau dans la pyramide. Si les valeurs v_{min} et v_{max} encadrent le niveau n de la case dans la pyramide, ($v_{min} \leq n \leq v_{max}$) alors la case est utile et doit être chargée.

Notre algorithme de calcul de la carte de chargement est conçu pour être implémenté sur les processeurs graphiques actuels. Il fonctionne sur des géométries arbitraires, possiblement animées et des paramétrisations quelconques. L'algorithme a une approche hiérarchique et produit une estimation conservative des cases de texture utiles. Il est facile à implémenter et suffisamment rapide pour des applications temps réel. Nous décrivons d'abord l'algorithme pour calculer un niveau donné de la carte de chargement en section 4.1, puis nous verrons comment calculer toute la pyramide à partir de seulement quelques niveaux en section 4.2.

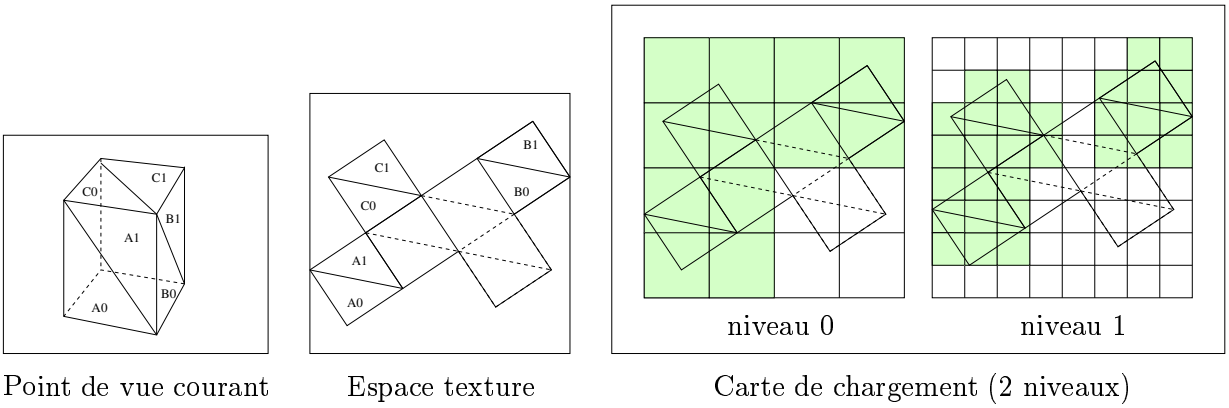


FIG. 6.5 – Carte de chargement

Notre algorithme calcule l'ensemble des cases de texture utiles, marquées en vert (gris) dans les niveaux de la carte de chargement.

4.1 Calcul d'un niveau de la carte de chargement

L'idée clé de notre algorithme est de dessiner la géométrie utilisée pour le point de vue courant dans l'espace texture, via la paramétrisation. Les coordonnées de texture des sommets sont utilisées pour le dessin, à la place de leurs coordonnées 3D. Le dessin est directement effectué dans le niveau de la carte de chargement, par le processeur graphique. Le niveau de la carte de chargement est représenté par une texture, et une opération de "dessin dans texture" (*render to texture*) est réalisée pour que le dessin soit fait dans la carte de chargement plutôt qu'à l'écran. La résolution d'affichage est choisie de manière à ce qu'un pixel affiché corresponde à une case du niveau de la carte de chargement. Les couleurs générées seront directement les valeurs stockées dans les cases d'un niveau de la carte de chargement (marqueur d'utilisation, v_{min}, v_{max}). Ce principe est illustré figure 6.6.

Les triangles sont donc dessinés dans le niveau de la carte de chargement en utilisant les coordonnées de texture des sommets. Lorsqu'un pixel est affiché aux coordonnées (i, j), cela implique que le triangle en train d'être dessiné est texturé par une partie de la case de texture (i, j) (voir figure 6.7). La case doit alors être marquée comme utilisée. La détermination des

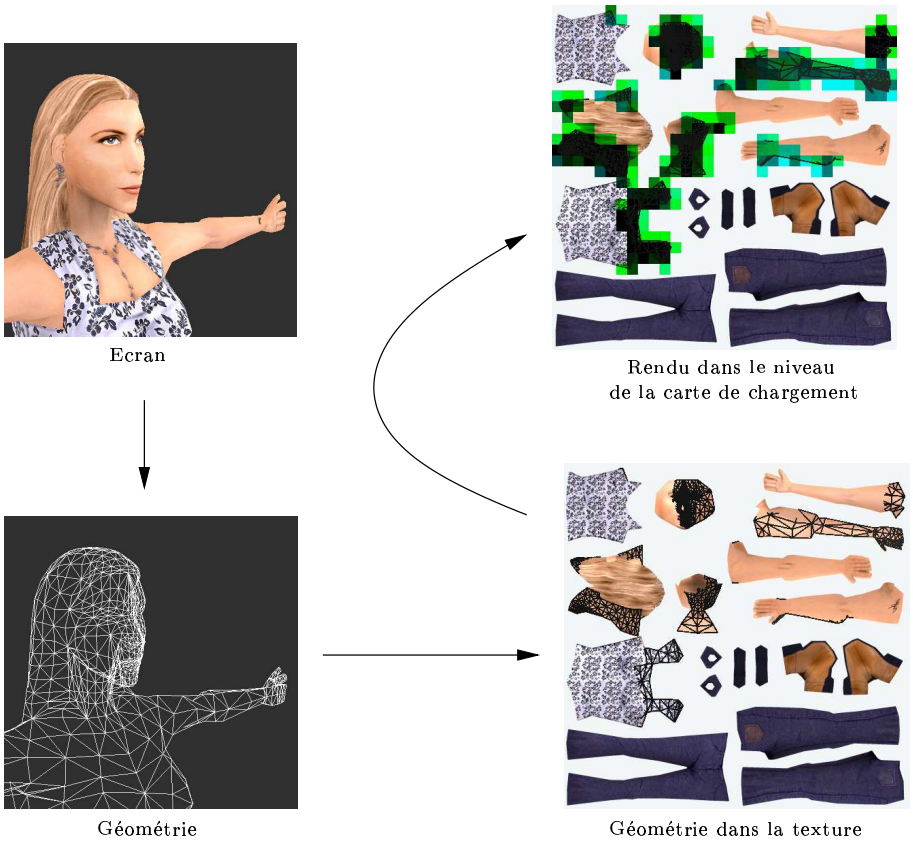


FIG. 6.6 – Principe de l'algorithme de carte de chargement.

La géométrie du point de vue courant (en bas à gauche) est dessinée dans l'espace texture (en bas à droite). Le rendu est directement effectué dans le niveau de la carte de chargement (en haut à droite). La couleur des pixels détermine les valeurs stockées dans les cases.

valeurs LOD (voir section 4.1.3) permettra de décider si la case est *utile* (si elle doit être chargée ou non).

Pendant le dessin, nous calculons également, en chaque sommet, les coordonnées e auxquelles ils sont projetés à l'écran pour le point de vue courant. Ce calcul est effectué dans un *vertex program* à l'aide de la matrice de projection du point de vue. Les coordonnées e sont calculées en chaque sommet et interpolées en chaque pixel (voir chapitre 2, section 1.4). Nous avons donc accès, en chaque case du niveau de la carte de chargement, aux coordonnées à l'écran d'un point du triangle utilisant la case de texture correspondante. Les coordonnées des sommets à l'écran vont nous permettre d'éliminer les triangles non visibles depuis le point de vue courant (voir section 4.1.1), de calculer le niveau de détail auquel les cases de texture sont utilisées (voir section 4.1.3) et de prendre en compte l'occlusion (voir section 4.1.4).

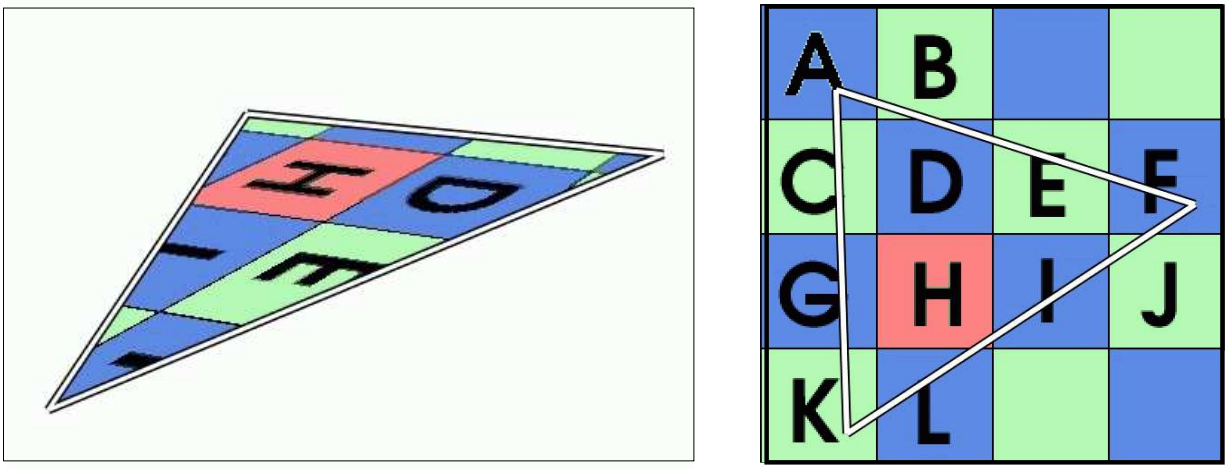


FIG. 6.7 – Carte de chargement pour un triangle.

A gauche : *Triangle texturé affiché à l'écran.* A droite : *Même triangle dessiné dans le niveau de la carte de chargement de résolution* 4×4. *Les coordonnées de texture sont utilisées pour le rendu. Les pixels dessinés correspondent chacun à une case de texture. Les cases utilisées sont marquées d'une lettre.*

4.1.1 Elimination des faces cachées

Lors du calcul d'un niveau de la carte de chargement, tous les triangles de la géométrie sont potentiellement pris en compte. Or, nous souhaiterions éliminer les triangles non visibles dans le point de vue courant, soit parce qu'ils sont vus de dos, soit parce qu'ils sortent des bordures de l'écran (la prise en compte de l'occlusion – objet en cachant un autre – sera discutée en section 4.1.4).

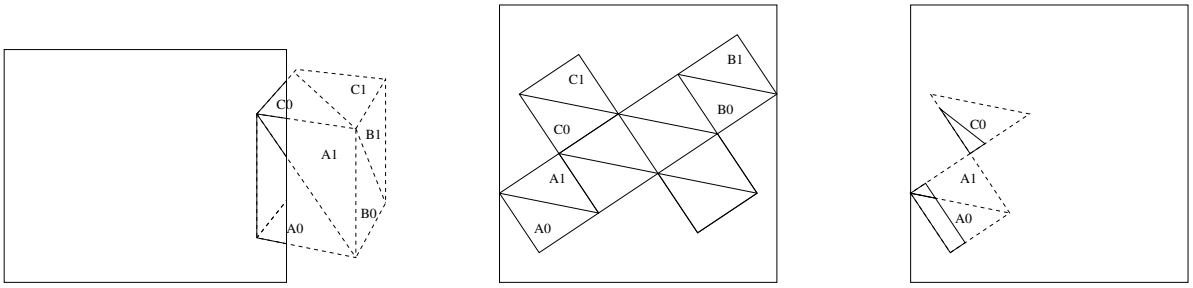
Pour éliminer les triangles en dehors du bord de l'écran, nous effectuons une opération de découpe (*clipping*) lors du dessin dans le niveau de la carte de chargement. La découpe des triangles est effectuée *en fonction des coordonnées à l'écran des sommets*, et non pas en fonction des coordonnées de texture utilisées pour le dessin. Cette opération peut paraître coûteuse, mais elle peut être effectuée directement par les processeurs graphiques, en sortie du *vertex program*[1]. Grâce à cela, seule la géométrie située dans le cadre de l'écran est effectivement dessinée dans la carte de chargement (voir figure 6.8).

L'élimination des faces vues de dos est un peu plus difficile. Les cartes graphiques effectuent automatiquement cette opération, mais en considérant l'orientation des triangles dans l'affichage courant. Or, nous voulons tenir compte de l'orientation des triangles depuis le point de vue courant (à l'écran), alors que nous dessinons dans l'espace texture. Nous pourrions utiliser la normale en chaque triangle. Cependant, la plupart des maillages sont spécifiés avec une normale en chaque sommet. Obtenir la normale en chaque triangle peut paraître simple, mais cette opération n'est pour le moment pas disponible sur les processeurs graphiques. Spécifier une normale par triangle demanderait de modifier complètement le maillage, ce qu'il faut éviter à tout prix, surtout que le maillage deviendrait très inefficace (il faut déconnecter les triangles). En attendant

[1] Par exemple à l'aide des registres de découpe (*clipping registers*) des cartes NVidia

que cette possibilité soit offerte sur les cartes graphiques , nous proposons d'utiliser les plans de découpe de triangles (plans de *clipping*) automatiquement gérés par les processeurs graphiques. Le plan qui nous intéresse est le plan de découpe *loin* (*far plane*). L'espace texture est 2D. Ce plan est donc inutile durant le dessin du niveau de la carte de chargement. Nous utilisons la normale au sommet pour simuler une coordonnée $z = dot(nrm, vv) - 1.0$ où *nrm* est la normale et *vv* le vecteur de vue normalisé, calculé en chaque sommet. Le plan de coupure *loin* est positionné en $z = -1$. Les triangles sont donc découpés sur la ligne $dot(nrm, vv) = 0.0$.

La géométrie utilisée pour calculer un niveau de la carte de chargement est la même que pour dessiner le point de vue courant. Le même traitement géométrique est effectué dans les deux cas (et toujours par le processeur graphique). Le coût géométrique est donc le même pour calculer le niveau de la carte de chargement que pour afficher la scène. Cependant, la résolution de dessin est beaucoup plus faible et moins de pixels sont dessinés.



Ecran - point de vue courant | Géométrie complète dans l'espace texture | Géométrie découpée dans l'espace texture

FIG. 6.8 – Elimination des faces situées en dehors du rectangle de l'écran

4.1.2 Obtenir un résultat conservatif

Nous avons vu comment remplir les cases d'un niveau de la carte de chargement avec une opération de rendu effectuée par la carte graphique . Cependant il nous faut garantir que l'ensemble des cases nécessaires va être détecté, tout en essayant d'obtenir la meilleure estimation possible.

A cette fin, il nous faut disposer d'un algorithme de rasterisation (voir chapitre 2, section 1.4) conservatif : tout pixel intersecté par la géométrie doit être dessiné. Sans cette propriété des cases de texture utiles pourraient être manquées – et les données correspondantes ne seraient pas chargées. La plupart des algorithmes de rasterisation ne sont pas conservatifs par défaut (ils suivent la règle du diamant, voir la norme OpenGL [Ope03]). Pour obtenir cette propriété, nous utilisons un mode spécial qui permet de dessiner des arêtes épaisses[2]. L'algorithme de rasterisation dessine alors tous les pixels intersectés par la géométrie [Ope03].

[2]Le but premier de ce mode de rendu est de supprimer l'aliasing des silhouettes. Ce mode est disponible en OpenGL sous le nom *GL_POLYGON_SMOOTH*

D'autre part nous verrons en section 4.1.3 comment calculer le niveau de détail auquel la case est utilisée par un triangle. Comme plusieurs triangles peuvent utiliser une même case de texture, il faut également combiner ces informations pour calculer les valeurs v_{min} et v_{max}. Nous utilisons à cette fin un opérateur de mélange (*blend*) qui maintient le maximum entre les valeurs déjà présentes dans un pixel et les nouvelles valeurs produites par le dessin (le minimum peut être obtenu de la même manière).

4.1.3 Calcul du niveau de détail

Nous devons calculer le niveau de détail auquel les cases de texture sont utilisées. Ce calcul est effectué dans un *fragment program* exécuté en chaque pixel lors du dessin du niveau de la carte de chargement. Rappelons que les valeurs provenant de plusieurs triangles sont ensuite combinées pour calculer les valeurs LOD minimales et maximales v_{min} et v_{max} (voir section 4.1.2).

Le niveau de détail peut être déduit en examinant comment une case de texture est utilisée par les pixels de l'écran. Ce problème est très proche de celui que l'on tente de résoudre pour le filtrage, en cherchant l'empreinte des pixels de l'écran dans la texture (voir chapitre 2, section 4). Ici, il s'agit non plus de déterminer l'empreinte du pixel de l'écran dans la texture, mais du problème dual : il faut déterminer l'empreinte de la case de texture à l'écran. Comme expliqué en section 4.1, nous avons accès en chaque case du niveau de la carte de chargement aux coordonnées *à l'écran* d'un point du triangle dessiné. Ce point est situé sur le triangle, dans la zone utilisant la case de texture. En calculant la différence finie de ces coordonnées entre les cases du niveau de la carte de chargement, il est possible de déterminer combien de pixels à l'écran sont utilisés pour afficher chaque case de texture. Ceci donne une bonne approximation du niveau de détail. Les opérateurs de différence finie, disponibles sur certains processeurs graphiques[3], permettent ce calcul.

Cependant des problèmes de précision peuvent apparaître avec cette approche. Nous proposons donc une autre méthode utilisant uniquement le placage de texture standard et l'algorithme de MIP–mapping.

Calcul du niveau de détail avec le placage de texture standard Ce paragraphe, relativement technique, explique notre méthode pour éviter le recours aux opérateurs de différence finie.

Nous créons tout d'abord une texture utilitaire 2D et sa pyramide de MIP–mapping. Cette texture, appelée texture LOD, a une résolution de $2^{k-1} \times 2^{k-1}$ pixels où k est le nombre de niveaux de texture gérés par notre architecture.

La texture LOD a donc k niveaux de MIP-mapping. La couleur de tous les pixels d'un niveau de MIP-mapping est choisie de manière à correspondre au numéro d'un niveau de notre système : les pixels du niveau i ont la couleur $R = G = B = i$. Un exemple de texture LOD est montré figure 6.9.

[3]Cartes graphiques NVidia à partir de la GeForce FX (NV30)

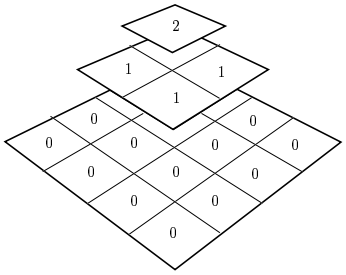


FIG. 6.9 – La texture LOD pour $k = 3$ niveaux gérés par notre architecture.

Lors du dessin, nous appliquons la texture LOD sur les triangles. Nous accédons à la texture en utilisant les coordonnées à l'écran des sommets. Comme la texture LOD est filtrée par l'algorithme de MIP–mapping, les couleurs produites correspondront aux niveaux de la pyramide sélectionnés. Cette couleur *est* la valeur du niveau de détail à laquelle la case de texture est utilisée par le triangle : puisqu'on utilise les coordonnées écran pour accéder à la texture LOD et que l'on dessine dans l'espace texture, l'algorithme de MIP–mapping va estimer l'empreinte des cases du niveau de la carte de chargement *sur l'écran* (au lieu d'estimer l'empreinte des pixels de l'écran dans l'espace texture).

Cependant, le filtrage automatique du processeur graphique sélectionne le niveau de MIP–mapping en fonction de la résolution de la texture LOD et de la résolution du niveau de la carte de chargement dessiné. Or, nous souhaitons calculer le niveau de détail auquel les cases de la texture gérée par notre architecture sont utilisées lors du dessin à l'écran. Nous devons donc décaler la sélection du niveau de MIP–mapping : il faut "remplacer" la résolution de la texture LOD par la résolution de l'écran, et la résolution du niveau de la carte de chargement par la résolution du premier niveau de la texture géré par notre architecture.

Pour plus de clarté, nous effectuons le raisonnement suivant en 1D. Pour le cas 2D, nous appliquons le raisonnement sur les deux dimensions indépendamment et gardons la plus grande valeur de niveau de détail (ceci afin d'être certain de charger toutes les données nécessaires). Soit M la résolution du niveau de la carte de chargement dessiné. Soit B la résolution du premier niveau de texture géré par notre système (voir figure 6.4). L est la résolution de la texture LOD. Soit $l \in [0,1]$ la coordonnée de texture utilisée pour accéder la texture LOD, $x \in [0,1]$ les coordonnées de texture utilisées lors du dessin et $s \in [0,1]$ les coordonnées à l'écran. Pendant le dessin dans la carte de chargement, la texture LOD est vue à pleine résolution si $\frac{dl}{dx} = \frac{M}{L}$. Nous voudrions que ceci corresponde à la sélection du premier niveau géré par notre système. Celui–ci doit être sélectionné lorsque $\frac{ds}{dx} = \frac{B}{S}$. Il vient $\frac{dl}{ds} = \frac{S}{B} \times \frac{M}{L}$. Ainsi, les coordonnées écran s doivent être multipliées par $\frac{S}{B} \times \frac{M}{L}$ avant d'être utilisées pour accéder à la texture LOD. Rappelons que chaque niveau de la texture LOD a une couleur uniforme : la couleur est uniquement donnée par le niveau de MIP–mapping sélectionné. La multiplication a donc pour seul effet de modifier la sélection du niveau de détail. La figure 6.10 illustre ce principe.

Le défaut de cette méthode est que la texture LOD a une résolution de $2^{k-1} \times 2^{k-1}$, ce qui peut gaspiller de la mémoire. Cependant cette limitation pourrait être dépassée si les cartes graphiques donnaient accès au mécanisme de sélection du niveau de MIP–mapping sans avoir à réellement stocker une texture en mémoire. En pratique, nous utilisons rarement plus de $k = 6$ niveaux : le premier niveau géré par notre architecture a généralement une résolution de 2048×2048 et la taille de la texture double à chaque niveau.

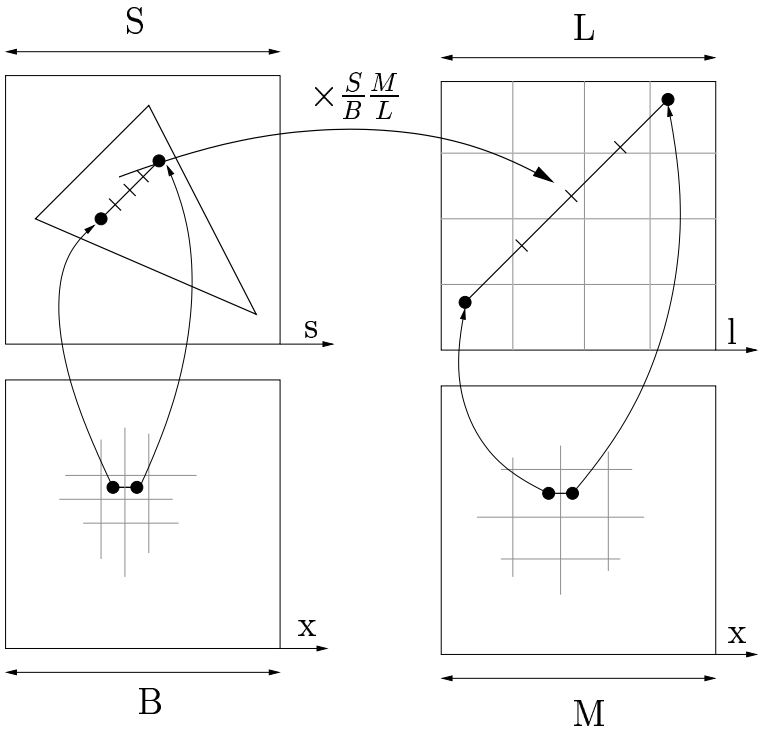


FIG. 6.10 – Détermination du niveau de détail

En haut à gauche : *La ligne sur le triangle fait quatre pixels de longueur sur l'écran.* En bas : *La ligne est texturée avec deux texels du premier niveau géré par notre architecture. Nous avons donc* $\frac{ds}{dx} = 4\frac{B}{S}$. *Dans ce cas, le troisième niveau géré par notre architecture devrait être sélectionné pour un rendu à pleine résolution.* En bas à droite : *La même ligne est dessinée dans le niveau de la carte de chargement de résolution M.* En haut : *La ligne est texturée par la texture LOD. Les coordonnées écran sont multipliées par* $\frac{S}{B} \times \frac{M}{L}$ *avant d'être utilisées pour accéder la texture LOD. Comme* $\frac{dl}{dx} = 4\frac{M}{L}$, *l'algorithme de MIP–mapping sélectionne le troisième niveau de la pyramide de la texture LOD. Ceci sélectionne également le troisième niveau de notre architecture.*

4.1.4 Prise en compte de l'occlusion

Notre algorithme, tel que présenté ci–dessus, surestime l'ensemble des cases de texture utiles. Cette estimation est bonne, puisque qu'une case est marquée comme utilisée, uniquement si un triangle faisant face à la caméra *et* situé dans le rectangle de l'écran utilise cette case de texture.

Cependant, l'occlusion n'est pas prise en compte : si un objet masque une partie de la texture, celle-ci sera tout de même déclarée visible. Gérer ce cas est important, notamment dans les scènes de type environnement urbain. Pour éviter des calculs géométriques complexes et coûteux (le problème de visibilité est très difficile à résoudre de manière exacte et efficace), nous proposons une solution certes approximative, mais simple à mettre en oeuvre. Il s'agit d'une approche similaire à l'algorithme de *shadow buffer* [Wil78]. Nous calculons, à chaque image, une carte de profondeur de la vue courante dans une texture de faible résolution. Rappelons que, lors du dessin dans le niveau de la carte de chargement, chaque case est associée à un point sur l'écran

(voir section 4.1). Nous utilisons cette coordonnée pour déterminer si les points sont visibles : leur profondeur à l'écran (coordonnée z) est alors égale à la profondeur lue dans la texture.

Cependant le résultat n'est pas conservatif : il s'agit d'une approximation. En effet, tout comme l'algorithme de *shadow buffer*, cette méthode souffre d'erreurs numériques. D'autre part seul un point est testé, alors que tout une partie du triangle peut être concernée : si le point est rejeté, toute la partie du triangle utilisant la case de texture est considérée comme non visible, ce qui peut être faux. Néanmoins, dans les applications interactives où la qualité peut être réduite au profit de la fluidité de rendu, ce choix est raisonnable. Dans le cas contraire, cette information de visibilité reste intéressante pour calculer une priorité de chargement.

4.2 Calcul de tous les niveaux d'une carte de chargement

Nous avons vu précédemment comment calculer un unique niveau d'une carte de chargement. Nous allons maintenant présenter comment caculer efficacemment l'ensemble des niveaux.

Une carte de chargement présente les propriétés suivantes :

- un niveau de la carte peut être déduit de tout autre niveau de plus haute résolution. La valeur v_{max} (resp. v_{min}) d'une case est alors calculée comme le maximum (resp. minimum) des valeurs v_{max} (resp. v_{min}) stockées dans les cases filles. Le résultat est conservatif par construction.
- dans les niveaux de haute résolution, seule une petite partie de la texture est utilisée par un point de vue donné : la résolution de l'écran limite la quantité de texture visible. Cependant, la zone visible n'est pas nécessairement continue. Nous appelons la zone marquée comme utilisée lors du dessin dans le niveau de la carte de chargement la *zone active*. Grâce à la conservativité, les zones actives d'un niveau de haute résolution de la carte de chargement sont toujours incluses dans les zones actives des niveaux de plus faible résolution.

Ces deux propriétés nous permettent de définir l'algorithme de calcul suivant :

```
Choisir la zone active de manière à couvrir toute la texture
Calculer le premier niveau (plus faible résolution)
Restreindre le dessin à la zone active
Pour (i=premier niveau+1; i<=dernier niveau; i+=N)
  Calculer le niveau (i+N)
  Restreindre le dessin à la zone active
  for (j=i+N-1; j>=i; j--)
      Calculer le niveau (j) à partir du niveau (j+1)
```

Cet algorithme calcule un niveau complet uniquement tous les N niveaux. Le rendu est restreint aux zones actives des niveaux précédents. Ceci requiert donc moins de calculs géométriques et de traitements par pixels que de dessiner l'ensemble des niveaux un par un. Le paramètre N permet de choisir la qualité de l'estimation. En pratique le nombre de niveaux gérés par notre méthode est souvent faible (chaque niveau étant deux fois plus grand que le précédent la résolution croît rapidement). Nous effectuons généralement uniquement le calcul du dernier niveau (le plus grand) et en déduisons les niveaux de plus faible résolution.

Enfin, après que le calcul de tous les niveaux ait été effectué, la carte de chargement est stockée sous la forme de textures sur la carte graphique (une texture par niveau) Il faut récupérer

cette information pour traitement sur le CPU. Le transfert à effectuer peut être lent[4]. C'est ici que le fait d'avoir une carte hiérarchique est essentiel : nous rapatrions d'abord le plus petit niveau, puis seulement les zones actives des niveaux de plus haute résolution.

4.3 Génération des requêtes de chargement et déchargement

Une fois la carte de chargement calculée et rapatriée, il faut générer les requêtes vers le cache de texture. Deux événements doivent être détectés : les cases devenant utiles et les cases qui ne sont plus utiles. Pour éviter un parcours complet de la carte de chargement, opération coûteuse, nous utilisons, une fois de plus, ses propriétés hiérarchiques.

Rappelons que chaque case de la carte de chargement contient les valeurs LOD minimales et maximales (v_{min} et v_{max}) auxquelles la case est utilisée pour le point de vue courant. Une case est utile lorsque le niveau n auquel est situé la case dans la pyramide, est compris entre les valeurs v_{min} et v_{max} calculées pour la case, dans la carte de chargement. Notons que, si une case à un niveau donné n'est pas utile, alors aucune de ses cases filles ne peut l'être. En effet, une case n'est pas utile si la valeur LOD maximale calculée pour la case est en dessous du niveau de la case dans la pyramide. Autrement dit, une case parente aura été utilisée à sa place, ou bien la case n'est pas visible. Comme toutes les cases filles sont à des niveaux plus grands, elles ne peuvent être utiles.

Pour générer les requêtes, nous commençons donc par visiter entièrement le premier niveau de la carte de chargement (celui de plus faible résolution). Nous créons alors une liste des cases utiles. Seules les cases filles des cases présentes dans la liste sont visitées au second niveau. Une nouvelle liste est créée pour visiter le niveau suivant. Ceci continue jusqu'à ce que le dernier niveau soit atteint. Le résultat est la liste des cases utiles pour le point de vue courant.

Les requêtes sont générées en comparant les cases utiles au point de vue précédent et les cases utiles au nouveau point de vue. Les cases qui ne sont plus utiles sont marquées comme libres dans le cache : elles pourront être effacées. Les cases nouvellement utiles sont soit déjà présentes dans le cache, soit seront générées par le producteur de texture. Notons que les cases de texture sont stockées dans le cache sous la forme de petites pyramides de MIP–mapping, afin de simplifier le rendu (voir section 5). Il n'est donc pas utile de générer de requête de chargement pour une case parente utile si toutes ses cases filles sont également utiles, ce qui peut être facilement déterminé en vérifiant leurs valeurs LOD.

5 Cache de Texture

Le rôle du cache de texture est de gérer le stockage des cases de texture utiles. Les cases sont effacées avec une approche LRU (*Least Recently Used*) : si le cache est plein et qu'une nouvelle case doit être chargée, la case qui n'a pas été utilisée depuis le plus longtemps est remplacée. Si toutes les cases sont utiles au point de vue courant, la mémoire est saturée. Nous verrons ci–après comment ce cas est géré.

[4]Notons que l'arrivée des bus PCI Express devrait réduire ce coût.

Ce cache doit pouvoir être utilisé comme une texture standard à tout moment. Il doit donc résider dans la mémoire de texture. Nous utilisons l'approche de Kraus et Ertl [KE02] pour effectuer un stockage discontinu de la pyramide de texture (voir chapitre 2, section 2.1.7.2). Chaque niveau est recouvert d'une grille de pointeurs. Chaque pointeur pointe soit vers le vide, soit vers une petite texture correspondant à la case et stockée dans la mémoire de cases. Un pointeur vide n'est pas visuellement un trou dans la texture : une version basse résolution de la texture est utilisée à la place. Ce principe est illustré figure 6.11.

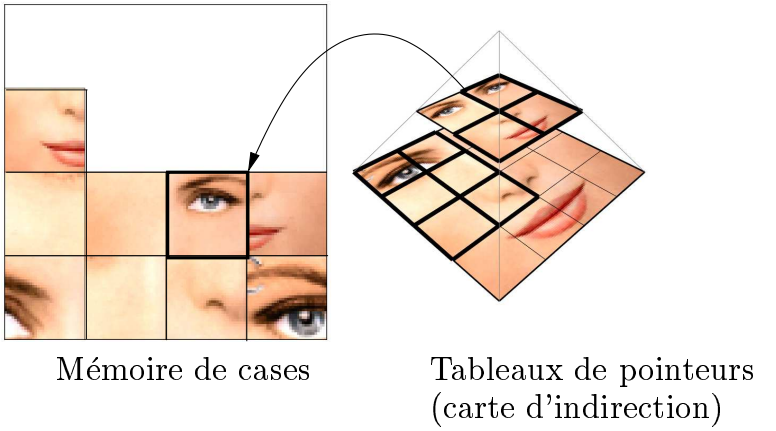


FIG. 6.11 – Stockage non continu des cases de texture.

Un tableau de pointeurs recouvre chaque niveau de la pyramide de texture. Les pointeurs indiquent une case stockée dans la mémoire de cases, ou indiquent un emplacement vide.

Lorsque le cache reçoit une requête pour une case qui devient visible mais n'est pas présente dans la mémoire de cases, il effectue les opérations suivantes :

- un emplacement est réservé dans la mémoire de cases ;
- le pointeur correspondant à la case est marqué comme vide ;
- une requête est envoyée au producteur de texture.

Dès que les données de la case sont générées par le producteur de texture dans la mémoire de cases, le pointeur est mis à jour.

5.1 Saturation de la mémoire

Dans certains cas, le cache peut ne pas être assez grand pour contenir l'ensemble des données. Quand cela se produit, une priorité permet de déterminer les cases devant être éliminées en premier (voir section 6).

5.2 Utiliser le cache comme une texture

Les données du cache sont utilisables comme une texture standard par la carte graphique . Afin d'obtenir un rendu de haute qualité, équivalent au placage de texture standard, nous devons

effectuer correctement le filtrage (l'algorithme de MIP–mapping est utilisé). A cette fin nous stockons les cases de texture sous forme de petites pyramides de MIP–mapping. Ceci permet de laisser la carte graphique effectuer le filtrage automatiquement dans une case. Le calcul de cette petite pyramide est effectué soit explicitement par le producteur de texture, soit par la carte graphique (cette opération est disponible sur la plupart des cartes[5]). Les discontinuités produites par le stockage non continu sont gérées en programmant explicitement l'interpolation linéaire, y compris entre les niveaux de MIP–mapping. Grâce au *stencil buffer* nous limitons cette correction, relativement coûteuse, aux pixels situés aux bordures des cases.

6 Producteur de Texture

L'objectif du producteur de texture est de générer la texture des cases déclarées nouvellement utiles. Il reçoit les requêtes de génération du cache de texture. Ces requêtes sont ajoutées à une file de priorité et gérées de manière asynchrone, selon un paramètre de bande passante fourni par l'utilisateur.

Les données nécessaires sont lues depuis les médias de masse (réseau, disque dur, ...). Ces données sont ensuite envoyées vers la mémoire texture, dans une zone temporaire. A partir de ces données, le producteur de texture dessine directement la case de texture dans l'espace réservé en mémoire de cases. En pratique, ceci est effectué avec une opération de rendu dans une texture (*render to texture*). Un *fragment program* est utilisé pour calculer la texture à partir des données chargées. Ce programme prend en paramètre les coordonnées de la case dans la pyramide de texture et les données chargées.

Le producteur de texture le plus simple copie simplement les données. Un producteur de texture plus complexe permet d'envoyer les données en mémoire texture sous forme compressée. Celles–ci sont alors décompressées par le *fragment program* exécuté par la carte graphique . L'intérêt est que les transferts de données coûteux (mémoire CPU vers mémoire texture) ne concernent plus que les données sous leur forme compressée. Pour illustrer ce principe nous avons implémenté deux décompresseurs basés sur des ondelettes : ondelettes de Haar et JPEG 2000 (noyau sur les entiers). Ceux–ci sont particulièrement adaptés au chargement progressif : seuls les détails doivent être envoyés pour augmenter la résolution des données déjà présentes dans le cache. Les détails peuvent également être envoyés par ordre d'importance. Implémenter ces décompresseurs dans des *fragment program* est un exercice relativement technique, mais ne présente pas d'obstacle particulier (le point le plus difficile concerne la gestion de la précision limitée des entiers manipulés par les processeurs graphiques). Enfin, un producteur de texture peut générer une texture procédurale. Dans ce cas aucune donnée n'est chargée : la texture est entièrement calculée par le *fragment program*, directement en mémoire texture.

Priorités de chargement Lors d'un chargement progressif, deux événements désagréables peuvent survenir : la mémoire allouée peut ne pas être suffisante pour un des points de vue ; la bande passante (ou la capacité de génération d'une texture procédurale) peut être très lente.

[5] Par exemple sous OpenGL avec l'extension `GL_SGIS_generate_mipmap`

L'utilisation d'une règle de priorité peut réduire ces problèmes : les parties les plus "importantes" sont chargées en premier. Lors du dessin de la carte de chargement, une priorité est calculée à partir de plusieurs critères : distance entre le triangle utilisant la case de texture et le centre de l'écran ; distance entre le triangle et le point de vue. Le producteur de texture propose également un critère basé sur les données, par exemple à partir des coefficients d'ondelettes ou d'une carte de priorités peinte par l'utilisateur. Les requêtes de production de texture sont triées en fonction d'une somme pondérée de ces critères.

7 Résultats

Nous avons testé notre architecture dans différentes situations : rendu de terrain, rendu de personnages et scènes architecturales. Dans tous les exemples présentés, nous laissons la carte graphique gérer les premiers niveaux de la pyramide de texture (dans ces exemples ceci concerne les niveaux plus petits que 1024^2). Notre architecture s'occupe des niveaux de plus haute résolution. Nous mesurons les performances sur trois scènes (terrain, personnage et scène architecturale) avec des chemins de caméra prédéfinis. Les mesures sont effectuées sur un Pentium 3GHz avec une carte NVidia GeForce FX 5950.

Terrain Notre système étant capable de gérer des maillages arbitraires, il fonctionne également sur de larges terrains. Nous utilisons les données du terrain *Puget Sound* (forunies par l'USGS et l'Université de Washington). La résolution de la texture est 16384×16384 RGB (1Go avec la pyramide de MIP-mapping). Les quatre derniers niveaux sont gérés par notre système. La résolution des cases de texture est 64×64.

Personnages Les personnages sont notre premier exemple de maillage arbitraire. Ils sont classiquement texturés avec des atlas de texture (voir chapitre 2, section 2.1.2.4) et animés. Notre algorithme de carte de chargement fonctionne très bien sur ce cas. En particulier, les zones inutiles de la texture ne sont jamais chargées. La texture utilisée a une résolution de 2048×2048 RGB. Les deux derniers niveaux sont gérés par notre architecture. La résolution des cases de texture est de 32×32 pixels.

Scènes architecturales Ce type de scène est généralement composé de multiples textures appliquées sur différents objets. Pour gérer ce cas, nous créons une unique texture en empaquetant toutes les textures dans une large image. Les coordonnées de texture des objets sont modifiées en conséquence. Notons que ceci n'est pas une modification de la géométrie : il suffit d'appliquer une translation et une mise à l'échelle pour calculer les nouvelles coordonnées. En outre, empaqueter les textures peut se faire avec une approche naïve : notre algorithme de carte de chargement ignorera les vides. Nous illustrons ce principe sur les données de la démonstration NVidia *Gas Station*[6]. La scène contient 26000 triangles et 128 Mo de texture. Les trois derniers niveaux sont gérés par notre architecture. La résolution des cases de texture est de 32×32 pixels.

[6]Géométrie et textures utilisées avec l'accord de NVidia

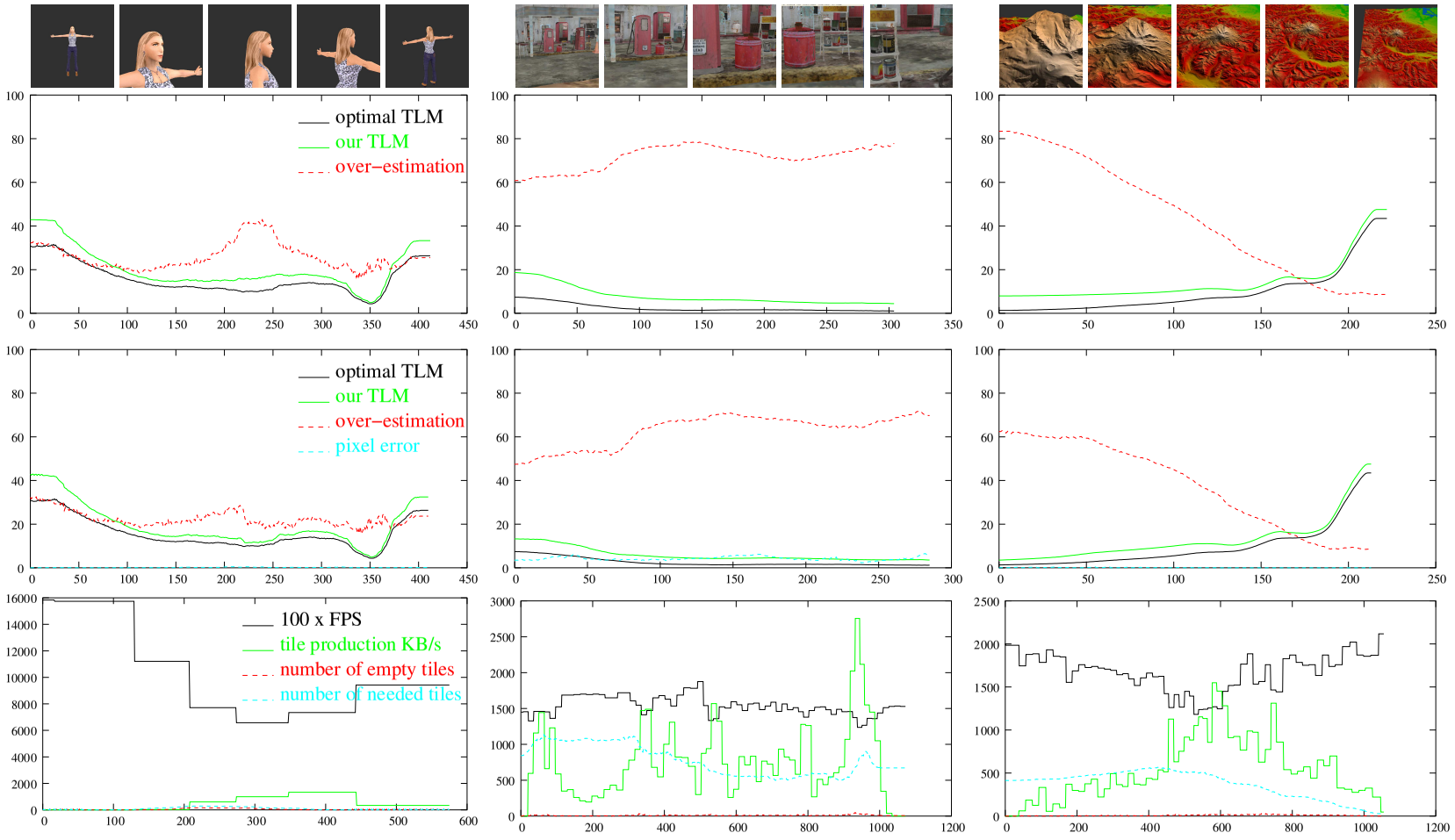


FIG. 6.12 – Mesures de performance.

De gauche à droite : *Modèle de personnage, scène architecturale (Gas Station) et modèle de terrain (Puget Sound).* De haut en bas : *Mesure de la qualité d'estimation des cartes de chargement, même mesure avec l'occlusion, performance du calcul de carte de chargement, performance de l'architecture complète. Les animations sont montrées dans la vidéo disponible sur* `http://www-evasion.imag.fr/Membres/Sylvain.Lefebvre/these` .

7.1 Résultats de l'algorithme de carte de chargement

Performance L'algorithme est suffisamment efficace pour les applications temps réel. Nous mesurons le nombre d'images par seconde avec seulement le calcul de la carte de chargement et la génération des requêtes activées (aucune donnée de texture n'est produite). Ceci inclut le rapatriement de la carte de chargement pour traitement par le CPU. Pour le personnage, le nombre moyen d'images par seconde est de 135 FPS et le coût de traitement sur le CPU est compris entre 0.02 ms et 0.6 ms. Pour la scène *Gas Station*, le nombre moyen d'images par seconde est de 36 FPS et le coût de traitement sur le CPU est compris entre 2.7 ms et 6.0 ms. Pour le terrain, le nombre moyen d'images par seconde est de 90 FPS et le coût de traitement sur le CPU est inférieur à 1 ms.

Qualité d'estimation Nous estimons la qualité de notre algorithme en comparant l'ensemble de cases de texture sélectionnées avec l'ensemble optimal. L'ensemble optimal est obtenu en dessinant le point de vue courant, en affichant les coordonnées de texture comme couleur. L'image affichée à l'écran est ensuite rapatriée pour traitement par le CPU. Elle est parcourue pixel par pixel pour déterminer les cases de texture utilisées. Ce processus est évidemment très lent.

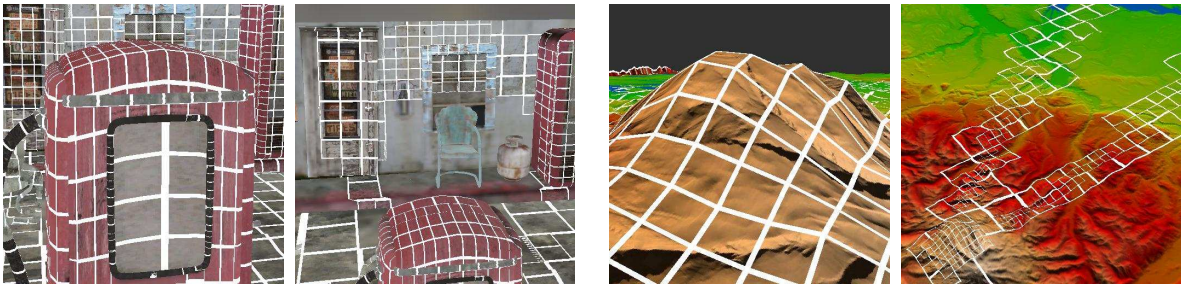


FIG. 6.13 – Prise en compte de l'occlusion

Gros plan sur le modèle Gas Station *et le terrain. La première image est le point de vue courant, la seconde image un point de vue externe sur la même scène. Dans les deux cas, la prise en compte de l'occlusion permet de ne pas charger les parties cachées par d'autres objets.*

La figure 6.12 résume les résultats obtenus avec et sans prise en compte de l'occlusion (première et seconde lignes). Les deux premières courbes indiquent le pourcentage de la texture complète sélectionné par notre algorithme et par l'optimal. La sur–estimation est donnée en pourcentage de cases faussement sélectionnées comme utiles par notre algorithme *par rapport au nombre total de cases sélectionnées*. L'erreur est le pourcentage de pixels, à l'écran, qui ne sont pas texturés correctement, suite à une imprécision dans le calcul d'occlusion, lorsque celui–ci est activé.

L'estimation de notre algorithme suit l'optimum de près. La sur–estimation est plus importante dans les scènes présentant une forte occlusion (*Gas Station* et terrain). Notez comme la sur–estimation diminue lorsque le point de vue s'éloigne du terrain : grâce à l'altitude, il n'y a presque plus d'occlusion et l'estimation est proche de l'optimum. Dans le cas du personnage, le

pic de sur–estimation (vers l'image 230) est supprimé par l'algorithme d'occlusion : cela correspond au moment où un bras du personnage est masqué par sa tête.

7.2 Rendu sous forte contrainte mémoire

Sous de très fortes contraintes mémoire, notre architecture permet de concentrer la résolution vers le centre de l'écran. Ceci grâce au chargement avec priorité (voir section 6). La figure 6.14 montre une vue de la *Gas Station* obtenue avec une contrainte de taille mémoire de 11 Mo (la texture pèse 128 Mo). Ceci montre également, qu'en cas de bande passante limitée, les zones les plus significatives sont chargées en priorité.

7.3 Performances globales de notre architecture

La figure 6.12 (troisième ligne) montre les performances de notre architecture pour les trois scènes de test. Dans les trois cas le taux de rafraîchissement est suffisant pour des applications interactives. Remarquez l'impact de la production de texture sur le temps de rendu (ceci inclut le temps de transfert des données de texture). Le nombre de cases vides est maintenu bas, ce qui permet de garantir une bonne qualité visuelle.

FIG. 6.14 – Rendu sous forte contrainte mémoire.
Deux points de vue rendus avec une contrainte mémoire de 11 Mo sur les textures. L'image de gauche est obtenue avec notre architecture. L'image de droite est obtenue en redimensionnant la texture originale pour que sa taille soit de 11 Mo. Les détails sont préservés par notre système.

7.4 Points faibles et limitations

L'algorithme de calcul de carte de chargement est effectué sur la carte graphique . Ceci implique que le résultat doit être rapatrié pour traitement sur le CPU. En pratique, la latence due à cette opération est difficile à évaluer. Cependant, les performances ne sont pas dangereusement dégradées, comme le montrent les résultats obtenus. Les progrès techniques, en matière de bus de

données, vont également permettre de bien meilleures performances dans le futur. Cependant, le fait que notre carte de chargement soit hiérarchique permet déjà d'équilibrer qualité d'estimation et transferts de données.

En fait, le principal point faible de notre architecture concerne le rendu final. En premier lieu, nous devons programmer explicitement l'interpolation linéaire, y compris entre les niveaux de MIP–mapping, ce qui est relativement coûteux. Ceci pourrait être résolu par le support du stockage discontinu de textures sur les cartes graphiques . En second lieu, nous devons afficher les différents niveaux de la pyramide avec plusieurs passes de rendu. Ceci signifie que la géométrie est envoyée plusieurs fois (une fois par niveau) pour le dessin à l'écran, ce qui dégrade les performances. La contribution des différents niveaux est mélangée à l'écran avec un opérateur de *blend*. Cette limitation va pouvoir être dépassée en tirant profit des branchements dynamiques maintenant disponibles sur les cartes graphiques pour sélectionner le niveau de la pyramide, en chaque pixel, durant le dessin à l'écran. Finalement la principale limitation concerne la taille maximale des textures que nous pouvons gérer. Notre implémentation du cache de texture nécessite de maintenir des grilles de pointeurs. La taille des ces grilles est la taille de la texture divisée par la résolution des cases de texture. Malheureusement, cette taille peut devenir elle–même trop importante pour que les grilles soient stockées en mémoire texture. En pratique cela se produit pour des grilles de pointeurs plus large que 4096^2. En supposant des cases de texture ayant une résolution de 64^2 pixels, cela donne une taille maximale de texture de 262144^2. Une solution à ce problème serait d'utiliser des grilles de pointeurs hiérarchiques. Nous verrons au chapitre 7 que de telles structures sont possibles (et efficaces) sur les cartes graphiques modernes.

8 Conclusion

Nous avons présenté une méthode unifiée pour gérer de très larges textures plaquées sur des maillages triangulaires quelconques. La paramétrisation peut être quelconque et la géométrie animée. L'utilisateur ne doit fournir que la position de la caméra et la géométrie utilisée pour dessiner le point de vue courant. Les deux contributions principales sont l'algorithme de calcul des parties utiles de la texture, bien adapté aux cartes graphiques actuelles et qui permet une réponse suffisamment rapide pour des applications temps réel, et l'architecture permettant d'unifier les solutions de chargement progressif (cache de texture, compression, chargement asynchrone avec priorité).

La texture est chargée uniquement à la résolution à laquelle elle est utilisée et les espaces vides présents dans les atlas de texture sont ignorés. Ceci permet de ne pas souffrir du gaspillage mémoire induit par le placage de texture avec les paramétrisations planaires de surfaces complexes (voir chapitre 2, section 2.1.6.1). D'autre part notre architecture propose un support naturel des textures procédurales telles que celles présentées au chapitre 3. En particulier, une stratégie de cache est appliquée aux données générées.

Cette approche s'inscrit parfaitement dans la tendance actuelle de mise en place de systèmes de mémoire virtuelle dans les cartes graphiques : pouvoir déterminer efficacement quelles parties des textures doivent être chargées (ou déchargées) de la mémoire, en fonction du point de vue, est un élément indispensable pour éviter les longs échanges (phénomènes de *swap*) entre mémoire texture et mémoire principale.

Quatrième partie

Modèles d'habillage sans paramétrisation planaire globale

chapitre 7

Textures hiérarchiques en volume pour processeurs graphiques

L'état de l'art (chapitre 2) nous a permis de mettre en évidence les difficultés à associer une image 2D à une surface complexe. Ceci pose de nombreux problèmes tant du point de vue de l'efficacité du stockage (voir chapitre 2, section 2.1.6.1), de la qualité du rendu (voir chapitre 2, section 2.1.6.2) que du point de vue de la création des textures pour les artistes ou pour les algorithmes qui doivent tous deux tenir compte des discontinuités et distorsions (voir chapitre 2, section 5).

Les approches en volume (voir chapitre 2, section 2.4) permettent d'associer une apparence à une surface sans avoir recours à une paramétrisation planaire. En particulier, les *octree textures* [DGPR02,BD02] permettent d'encoder l'information de couleur dans l'espace et seulement autour de la surface. Le principal problème de cette approche est qu'elle n'est pas disponible sur les cartes graphiques pour les applications interactives.

Dans la première partie de ce chapitre, nous allons présenter notre implémentation pour les cartes graphiques d'une structure hiérarchique similaire aux *octree textures*. Il s'agit d'une contribution technique qui s'attache à réaliser une implémentation efficace en fonction des caractéristiques des cartes graphiques actuelles (section 1). Nous présenterons ensuite comment utiliser notre implémentation pour habiller des modèles dans les applications interactives (section 2). Ce sera l'occasion d'aborder la question du filtrage. Ensuite, nous introduirons deux approches nouvelles autorisées par notre implémentation : la simulation d'une texture dynamique et interactive sur la surface de la géométrie (chapitre 8) et la création de *textures composites* (chapitre 9).

Remarque : Les travaux présentés ici ont fait l'objet d'un chapitre du livre *GPU Gems 2* (à paraître en 2005 aux éditions *Addison–Wesley*) sous le titre *Octree Textures on the GPU* [LHN05a]

1 Textures volumiques hiérarchiques pour les cartes graphiques

Après avoir présenté la structure de données que nous souhaitons réaliser, nous verrons comment celle–ci est efficacement stockée en mémoire texture et comment nous pouvons y accéder depuis un *fragment program* (voir chapitre 2, section 7).

1.1 Définition

Un *octree* est une grille hiérarchique régulière, formant une structure d'arbre. Le premier noeud de l'arbre est appelé la *racine*, c'est un cube unitaire. Chaque noeud a soit 8 fils, soit aucun. Les 8 fils forment une subdivision régulière $2 \times 2 \times 2$ du cube du noeud parent. Un noeud avec fils est appelé *noeud interne*, un noeud sans fils une *feuille*. La figure 7.1 montre un octree construit autour d'une surface : les noeuds contenant un morceau de surface ont été subdivisés, les autres sont laissés vides.

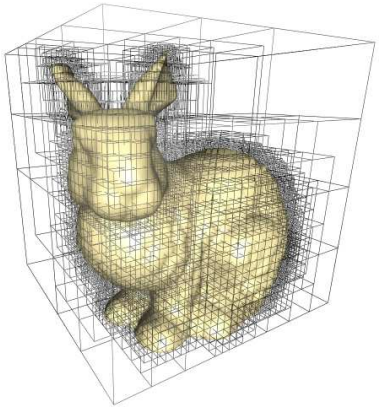


FIG. 7.1 – Un octree construit autour d'une surface

Dans un octree, la résolution selon chaque dimension est multipliée par 2 à chaque niveau de subdivision, ou *profondeur*. Ainsi pour atteindre une résolution de $256 \times 256 \times 256$ il faut 8 niveaux ($2^8 = 256$). Selon l'application, il peut être préférable de subdiviser les noeuds selon chaque dimension par un nombre N arbitraire, plutôt que par 2. Nous définissons donc une structure un peu plus générale que l'octree appelée N^3–tree . L'octree est obtenu pour $N = 2$. Une valeur plus grande de N réduit la profondeur de l'arbre nécessaire pour atteindre une certaine résolution, mais tend à gaspiller de la mémoire car la structure ne suit plus la surface d'aussi près.

1.2 Implémentation

Pour implémenter un N^3–tree sur une carte graphique , il nous faut tout d'abord définir comment stocker la structure efficacement en mémoire texture, puis définir comment y accéder depuis un *fragment program*.

Une approche classique pour implémenter un octree sur un CPU est d'utiliser des pointeurs pour relier les noeuds de l'arbre entre eux. Chaque noeud interne contient un tableau de pointeurs vers ses fils. Un fils peut être un autre noeud interne ou bien une feuille. Une feuille contient seulement un champ de données.

Notre implémentation suit une approche similaire. Les pointeurs deviennent des coordonnées dans une texture. Ils sont encodés comme des valeurs RGB. Le contenu des feuilles est directement stocké comme une valeur RGB dans le tableau de pointeurs du noeud parent. Nous utilisons la valeur d'alpha (la texture est donc au format RGBA) pour distinguer un pointeur du contenu

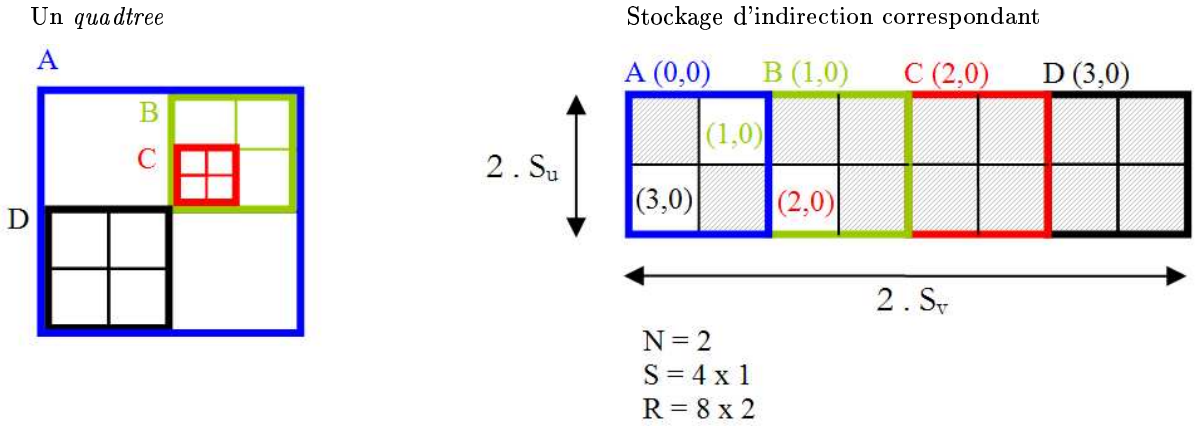


FIG. 7.2 – Stockage de la grille hiérarchique

Le stockage d'indirections encode l'arbre. Les grilles d'indirections des noeuds sont sur–lignées avec différentes couleurs. Les cases grises contiennent des données.

d'une feuille. Notre approche utilise des accès texture dépendants (ou *textures d'indirections*) Les processeurs graphiques utilisés doivent donc supporter ce type d'opération, ce qui est heureusement le cas de la plupart d'entre eux.

Les sections suivantes détaillent notre implémentation. Pour plus de clarté, les figures illustrent l'équivalent 2D d'un octree (appelé *quadtree*).

1.2.1 Stockage

Nous stockons l'arbre dans une texture 3D au format RGBA 8 bits appelée *stockage d'indirections*. Chaque texel de cette texture est appelé une *case*. Le stockage d'indirections est divisé en *grilles d'indirections*. Une grille d'indirections est un cube de $N \times N \times N$ cases (donc une grille $2 \times 2 \times 2$ pour un octree). Chaque noeud du N^3–tree est représenté par une grille d'indirections. Ceci correspond au tableau de pointeurs de l'implémentation pour CPU évoquée plus haut.

Une case d'une grille d'indirections peut être vide ou contenir :

- une donnée si le fils correspondant est une feuille
- la coordonnée d'une autre grille d'indirections si le fils correspondant est un autre noeud interne.

La figure 7.2 montre notre stockage. Nous notons $S = S_u \times S_v \times S_w$ le nombre de grilles d'indirections stockées dans le stockage d'indirections et $R = NS_u \times NS_v \times NS_w$ la résolution (en nombre de cases) du stockage d'indirectionss.

Les données et les coordonnées des noeuds fils sont toutes deux stockées comme des valeurs RGB. La valeur alpha est utilisée pour déterminer le contenu d'une case (alpha = 1 indique une donnée, alpha = 0.5 une coordonnée, alpha = 0 une case vide). La racine de l'arbre est toujours stockée aux coordonnées $(0,0,0)$ dans le stockage d'indirections.

1.2.2 Accès aux données

Maintenant que le N^3–tree est stocké en mémoire texture, nous devons définir comment y accéder depuis un *fragment program*. Comme pour une texture 3D standard, le N^3–tree définit une texture dans le cube unitaire. Nous souhaitons donc retrouver une donnée correspondant à un point de l'espace $M \in [0,1]^3$. La descente de l'arbre commence par la racine et se poursuit jusqu'à ce qu'une feuille soit rencontrée. Les noeuds contenant le point M sont tous visités.

Soit I_D la coordonnée de la grille d'indirections du noeud visité à la profondeur D. La descente est initialisée avec $I_0 = (0,0,0)$, ce qui correspond à la racine de l'arbre. Lorsque nous sommes à une profondeur D nous connaissons donc la coordonnée I_D de la grille d'indirections du noeud courant. Nous expliquons maintenant comment retrouver la coordonnée I_{D+1} du noeud fils à partir de I_D.

Le point M est à l'intérieur du noeud visité à la profondeur D. Pour décider de la suite, nous devons tout d'abord lire dans la grille d'indirections désignée par I_D la valeur stockée à la position correspondant au point M. A cette fin, nous devons calculer les coordonnées de M *à l'intérieur* du noeud.

A une profondeur D, un arbre complet produit une grille régulière de résolution $N^D \times N^D \times N^D$ à l'intérieur du cube unitaire. Nous appelons cette grille *la grille de profondeur D*. Chaque noeud de profondeur D correspond à une case de cette grille. En particulier, le point M se trouve dans une case qui correspond au noeud visité à la profondeur D. Les coordonnées de M à l'intérieur de cette case sont $frac(M\,N^D)$. Nous utilisons ces coordonnées pour lire la valeur dans la grille d'indirections I_D. Les coordonnées d'accès à la texture de stockage d'indirections sont donc :

$$P = \frac{I_D + frac(M\,N^D)}{S}$$

Ceci nous permet de lire la valeur RGBA stockée au point M dans la grille d'indirections I_D du noeud courant. Selon la valeur d'alpha (A), nous allons soit renvoyer la valeur *RGB* (feuille) où interpréter RGB comme la coordonnée du noeud fils (I_{D+1}) et continuer à la profondeur suivante. La figure 7.3 résume l'ensemble de ce processus dans le cas 2D.

La descente se termine lorsqu'une feuille est atteinte. En pratique notre *fragment program* s'arrête également après un nombre fixe d'accès texture : la plupart des processeurs graphiques ne permettent pas d'implémenter des boucles dépendantes des données (cependant il est possible d'interrompre l'exécution sur les plus récents). L'application est chargée de limiter la profondeur de l'arbre. Le *fragment program* complet est donné figure 7.4.

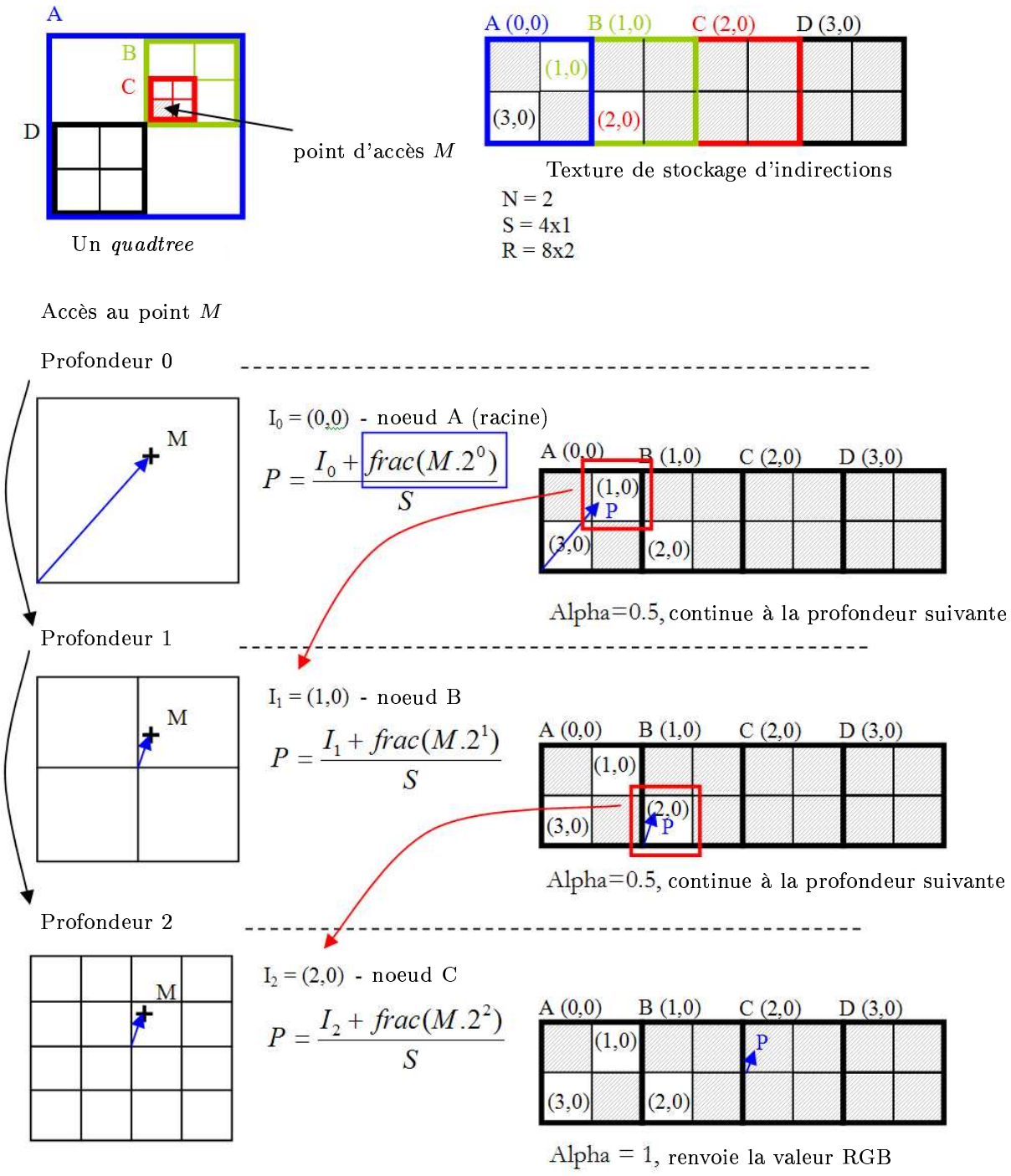


FIG. 7.3 – Accès dans la grille hiérarchique

Exemple d'accès dans une grille hiérarchique. A chaque étape, la valeur stockée dans la grille d'indirections du noeud courant est lue. Si cette valeur encode la coordonnée du noeud fils la descente continue, sinon la valeur RGB est renvoyée.

```
float4 tree_lookup(uniform sampler3D IndirPool, // Indirection Pool
                   uniform float3 invS, // 1 / S
                   uniform float N,
                   float3 M) // coordonnées d'accès
{
  float4 I = float4(0.0,0.0,0.0,0.0);
  float3 MND = M;

  for (float i=0;i<HRDWTREE_MAX_DEPTH;i++) // nombre fixe d'itérations
  {
    float3 P;
    // calcule les coordonnées dans le noeud courant
    P = (MND + floor(0.5+ I.xyz*255.0)) * invS;
    // accès au stockage d'indirections
    if (I.w < 0.9) // déjà dans une feuille?
      I =(float4)tex3D(IndirPool,P); // non, continuer

#ifdef DYN_BRANCHING // sortie si branchement dynamiques disponibles
    if (I.w > 0.9) // une feuille est atteinte
      break;
#endif

    if (I.w < 0.1) // feuille vide
      discard;
    // calcule la position dans la grille de profondeur suivante
    MND = MND * N;
  }
  return (I);
}
```

FIG. 7.4 – Code Cg du *fragment program* permettant d'accéder au N^3–tree

1.3 Autres optimisations

Il est possible de supprimer l'instruction *frac* du programme en tirant profit du comportement cyclique des textures. Nous n'allons pas décrire ici cette optimisation, mais le lecteur intéressé pourra se référer au chapitre du livre *GPU Gems 2* ou au code source disponible en ligne[1] pour plus d'informations.

[1] http://www-evasion.imag.fr/Membres/Sylvain.Lefebvre/these

1.4 Note sur l'encodage des coordonnées

Les coordonnées des grilles d'indirections sont des entiers. La texture utilisée pour le stockage d'indirections est une texture 8 bits. Nous disposons donc d'un espace d'adressage de 24 bits, ce qui est suffisant pour la plupart des applications, d'autant plus que la taille maximale des textures volumiques est à ce jour limitée à 256^3.

Dans un *fragment program*, un accès à une texture 8 bits retourne une valeur dans l'intervalle $[0,1]$. Cependant nous désirons stocker des entiers. Utiliser une texture de nombres flottants pourrait convenir mais cela gaspillerait beaucoup de mémoire et réduirait les performances (voir chapitre 2, section 7). A la place, nous stockons les entiers comme des nombres entre $[0,1]$ avec une précision fixe de $1/255$ et nous multiplions la valeur flottante lue depuis la texture par 255 pour obtenir un entier. Notons que, sur les cartes graphiques ne disposant pas d'arithmétique en virgule fixe (c'est le cas de la dernière génération !), nous devons calculer $floor(0.5 + 255 * v)$ pour éviter les erreurs dues aux imprécisions numériques.

2 Habillage avec une texture hiérarchique

Nous décrivons, dans cette section, comment utiliser notre implémentation pour habiller des modèles géométriques avec une texture hiérarchique. En particulier, nous discutons de la qualité de rendu et de la conversion entre une texture hiérarchique et une texture planaire standard.

2.1 Habillage de la surface

Afin d'habiller une surface avec une texture hiérarchique, il faut tout d'abord construire le N^3–tree autour de l'objet, puis remplir les feuilles de couleur.

La première étape de construction du N^3–tree consiste à définir en chaque sommet de l'objet une coordonnée dans le cube unitaire représentant l'espace texture en volume (voir chapitre 2, section 2.4). Cette coordonnée peut être trivialement définie à partir de la boîte englobante de l'objet. Ensuite, le N^3–tree est construit dans le cube unitaire, en subdivisant uniquement les feuilles qui intersectent la surface, jusqu'à ce que la profondeur de dessin soit atteinte (voir figure 7.1) . La profondeur de dessin fixe la résolution à laquelle la surface pourra être peinte. Notons que la subdivision n'a pas à être uniforme : il est possible d'allouer plus de résolution à certaines parties de la surface (par exemple pour dessiner les fins détails des yeux d'un personnage).

Une fois le N^3–tree construit, il est très facile de définir un outil de peinture analogue à l'outil de peinture des logiciels de dessin 2D. L'outil est une sphère déplacée par l'utilisateur sur la surface, et qui dépose de la couleur dans les feuilles de l'arbre qu'elle englobe. Les artistes peuvent ainsi peindre directement sur la surface, dans une application interactive : le résultat est directement visualisé. La résolution de peinture peut être adaptée dynamiquement (en subdivisant l'arbre). L'habillage produit ne souffre d'aucune distorsion, ni de défauts d'affichage. L'application de peinture est disponible sur le site internet de la thèse[2].

[2] http://www-evasion.imag.fr/Membres/Sylvain.Lefebvre/these

2.2 Qualité de rendu

Afin d'obtenir un rendu de haute qualité, comparable à celui obtenu avec la placage de texture standard, il nous faut réaliser deux opérations : l'interpolation linéaire et le filtrage.

Nous implémentons l'interpolation linéaire en étendant la méthode 2D au cas 3D : en 2D l'interpolation est effectuée dans un carré entre quatre voisins : en 3D, il faut l'effectuer dans un cube entre huit voisins. Huit accès sont donc nécessaires dans la texture hiérarchique. La texture doit également être créée de manière à ce que tous les échantillons nécessaires soient présents.

Le filtrage (voir chapitre 2, section 4) est plus difficile. Même s'il est possible d'étendre l'algorithme de *MIP–mapping* 2D à la structure hiérarchique 3D en pré–calculant les couleurs moyennes de tous les noeuds, le *fragment program* utilisé devient assez long et l'affichage s'en trouve ralenti. Nous proposons une solution alternative. Il est en effet possible de convertir dynamiquement, et efficacement, une texture hiérarchique en une texture standard – et ce en évitant les défauts visuels dus aux discontinuités (voir section 2.3). Notre idée est donc d'utiliser une texture standard pour le filtrage. Lorsque la surface est vue à pleine résolution, la texture hiérarchique est utilisée. Dans les zones devant être filtrées, la texture standard est utilisée. Même si la paramétrisation est de mauvaise qualité, peu de défauts seront visibles puisque la texture hiérarchique est utilisée pour tous les points de vue détaillés. Bien sûr, cette solution est valable seulement si les performances sont cruciales. Dans le cas contraire, l'algorithme de MIP–mapping appliqué à la texture hiérarchique 3D, bien que plus lent, offrira la meilleure qualité.

2.3 Conversion en une texture standard

Même si les performances des textures hiérarchiques sur cartes graphiques sont déjà très encourageantes, les textures standard restent beaucoup plus rapides car plus simples et directement supportées par les cartes graphiques . Il est donc intéressant de pouvoir sélectionner dynamiquement la meilleure représentation selon les cas. Par exemple la texture hiérarchique peut être utilisée pour les points de vue proches, alors que les objets plus lointains seraient texturés par une texture standard, les éventuels défauts visuels étant moins visibles.

Il est donc intéressant de pouvoir convertir rapidement une texture hiérarchique en une texture standard. Dans la suite, nous supposons que le maillage triangulaire est paramétré par une méthode usuelle de paramétrisation (voir chapitre 2, section 2.1.2).

Pour créer une texture 2D, il suffit de dessiner les triangles de la surface dans la texture, en utilisant les coordonnées de texture planaire des sommets (générées par la paramétrisation). La texture hiérarchique en volume est alors appliquée aux triangles en utilisant, comme avant, les coordonnées 3D interpolées depuis les sommets pour y accéder. Ce principe est illustré figure 7.5. Cependant, la texture ainsi générée produit des défauts d'interpolation et de filtrage à cause des discontinuités présentes dans l'atlas de texture (voir chapitre 2, section 2.1.6.2). Afin de résoudre ce problème, nous implémentons une version simplifiée de l'algorithme de *push–pull* [GGSC96] exécutée par la carte graphique. Ceci permet d'extrapoler les couleurs de la texture comme illustré figure 7.6. Cette méthode a précédemment été utilisée par Sander et al. [SSGH01] avec le même objectif. Cette conversion, très rapide, peut être effectuée à chaque image.

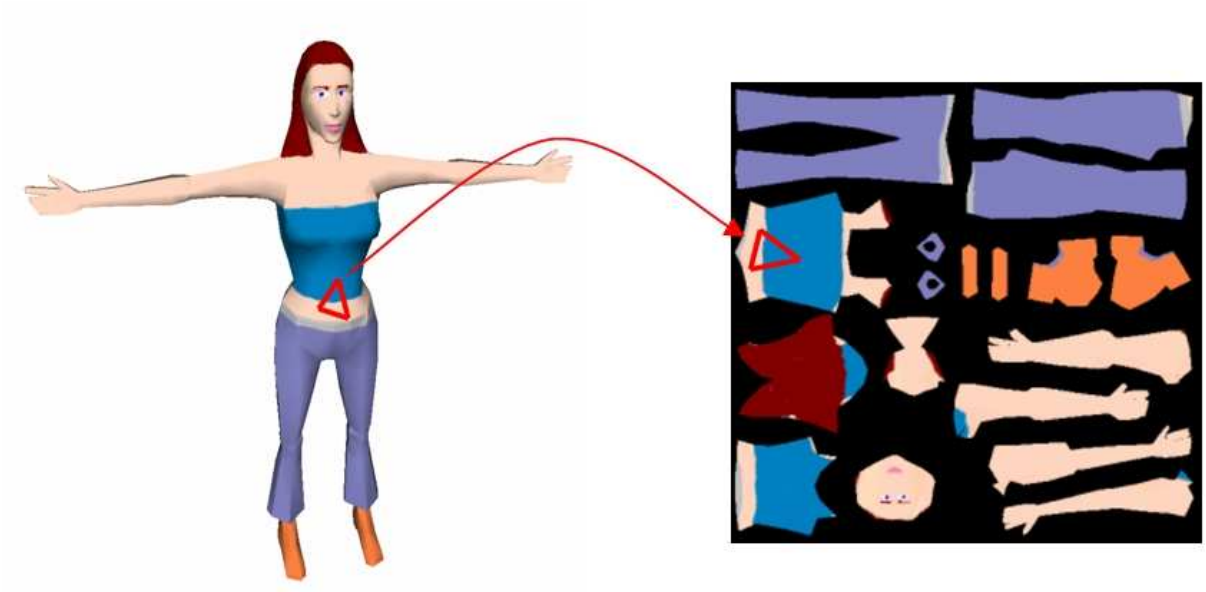


FIG. 7.5 – Conversion d'une texture hiérarchique en texture 2D standard.

FIG. 7.6 – Extrapolation automatique des couleurs en dehors des bordures.

2.4 Résultats

La figure 7.7 présente un modèle texturé avec notre implémentation des textures hiérarchiques. L'arbre est construit de manière à entourer la surface. La résolution varie le long de la surface.

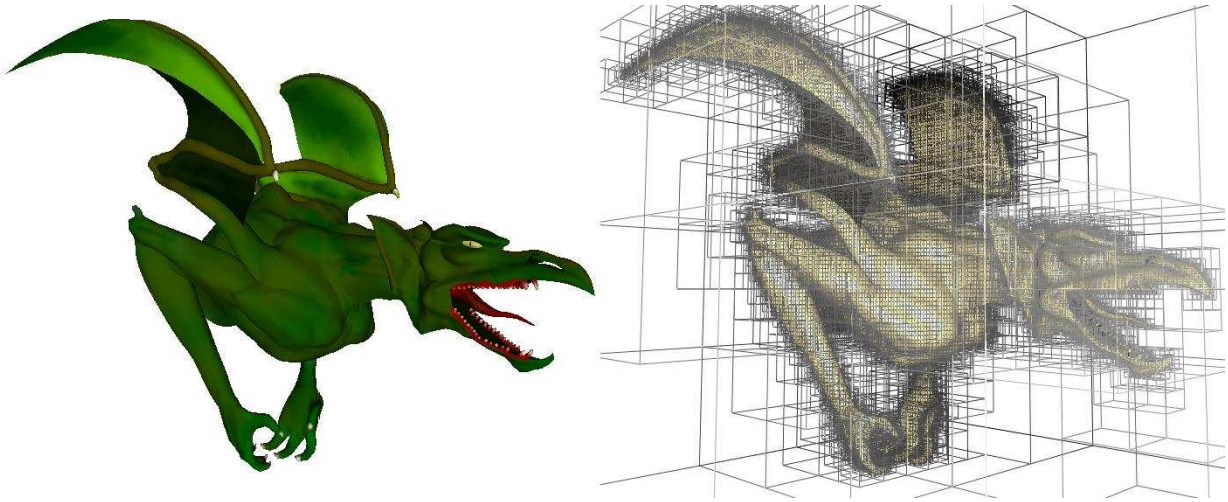


FIG. 7.7 – Texture volumique accélérée par la carte graphique .

A gauche : *Objet texturé par une texture volumique hiérarchique.* A droite : *Structure de la texture. Cette texture requiert 4.5 Mo, la profondeur maximale de la hiérarchie est de 12. Le nombre d'images par seconde est en moyenne de 70 pour une résolution de rendu de* 1024×768 *avec interpolation linéaire.*

3 Conclusion et travaux futurs

Nous avons présenté une implémentation pour les cartes graphiques de textures hiérarchiques volumiques, similaires aux *octree textures*. Cette implémentation permet d'habiller des modèles géométriques sans avoir recours à une paramétrisation planaire et de les afficher dans des applications interactives. Nous avons proposé des solutions pour le filtrage et un algorithme de conversion vers une texture standard.

Pouvoir afficher des textures hiérarchiques avec de hautes performances, les rend disponibles en tant que primitives de rendu dans les applications interactives, au même titre que le placage de texture. Elles ne sont plus seulement cantonnées aux logiciels de rendu très réalistes. Cependant, les performances, même si elles sont satisfaisantes, sont inférieures aux performances du placage de texture standard. Ceci est principalement dû aux indirections nécessaires pour la descente de l'arbre. Le coût provient notamment de la faible cohérence des accès lors de la descente : le cache est probablement sous–utilisé. Ceci pourrait être fortement amélioré en étudiant l'agencement optimal des grilles d'indirections dans la texture 3D. En attendant, notre structure de N^3–tree , un peu plus générale qu'un octree, permet d'adapter la taille des noeuds pour choisir entre la profondeur de l'arbre et l'espace gaspillé autour de l'objet.

Nous pensons que les textures hiérarchiques vont être de plus en plus utilisées dans le cadre des applications interactives. Elles permettent une solution élégante à des problèmes auxquels les artistes sont confrontés en permanence. Cette impression est renforcée par le succès des logiciels de peinture sur objet, pour lesquels les textures hiérarchiques sont parfaitement adaptées. De nombreux algorithmes restent à mettre au point pour ces textures : compression, chargement progressif, etc ...

chapitre 8

Texture dynamique sur surface

L'un des habillages les plus difficiles à réaliser est l'habillage dynamique : l'apparence de la surface évolue et réagit aux interactions extérieures (voir chapitre 2, section 5.4). Nous présentons dans ce chapitre une méthode permettant de définir un automate cellulaire à la surface d'un objet, via une texture hiérarchique (voir chapitre 7). Afin d'illustrer notre méthode, nous simulons, grâce à l'automate cellulaire, l'écoulement d'un liquide le long de la surface. L'automate est mis à jour par la carte graphique, ce qui nous permet d'atteindre des performances temps réel et de complètement décharger le CPU de la simulation.

Nous présenterons brièvement, en section 1, comment les automates cellulaires sont habituellement utilisés pour simuler des textures dynamiques. Nous décrirons, en section 2, le principe de notre approche et montrerons les résultats en section 3 avant de conclure en section 4.

1 Automates cellulaires et habillage de surface

Les automates cellulaires sont classiquement utilisés pour créer des apparences dynamiques et simuler des phénomènes physiques. Cependant, lorsqu'il s'agit de les utiliser pour générer des habillages de surface dynamiques, ils sont généralement limités à des surfaces planes ou développables [HCSL02]. En effet, un automate cellulaire est généralement représenté dans une grille régulière. Le contenu de la grille évolue, en mettant à jour l'état d'une cellule en fonction de l'état, à l'étape de simulation précédente, des cellules voisines. Si la grille de l'automate est attachée à la surface via une paramétrisation, les problèmes de distorsion et discontinuité vont apparaître.

Cependant, pour réaliser la simulation efficacement à l'aide d'une carte graphique , l'espace de simulation doit être à deux dimensions. Il est en effet possible, dans ce cas, de mettre à jour la grille de l'automate avec une opération de rendu et un *fragment program* approprié. C'est pour cela que les automates cellulaires sont souvent utilisés dans les applications interactives : le traitement en parallèle des pixels par le processeur graphique permet de les mettre à jour très efficacement.

Notons que, si l'automate est simulé via une paramétrisation planaire, les espaces vides présents dans un atlas de texture ou autour du contour d'une paramétrisation en un seul morceau vont entraîner, non seulement un gaspillage de mémoire, mais également un gaspillage de

puissance de calcul : les cases des espaces vides seront quand même traitées par le processeur graphique (même si leur résultat est ignoré).

2 Principe de notre approche

Notre approche est d'entourer la surface d'une texture hiérarchique. Les feuilles de la texture hiérarchique ne vont pas contenir une couleur, mais la coordonnée d'une case (i, j) dans une texture 2D, appelée *texture de simulation*. La texture de simulation correspond à la grille d'un automate cellulaire standard. Chaque case contient l'information associée à une feuille de l'arbre.

Cependant, les cases voisines dans la texture de simulation ne correspondent *pas* à des feuilles voisines dans la texture hiérarchique 3D. Or, l'automate a également besoin des informations de voisinage entre feuilles de l'arbre. Il nous faut donc stocker séparément ces relations de voisinage 3D. Pour ce faire nous utilisons un ensemble de textures, appelées *texture de voisins* (voir figure 8.1). Une feuille f est associé à une case (i, j) de la texture de simulation. Soit f_v une feuille voisine de f associée à la case (i_v, j_v) dans la texture de simulation. Dans une des textures de voisins, la case (i, j) va alors contenir les coordonnées (i_v, j_v) de la case de la texture de simulation associée à la feuille voisine f_v de f dans l'arbre.

En 3D, chaque feuille a potentiellement 26 voisins. Il faudrait donc utiliser 26 textures de voisins. En pratique, comme la simulation est effectuée sur une surface 2D, le nombre de voisins est plus proche de 9. Selon le phénomène simulé, il est donc possible d'utiliser moins de voisins. Lorsqu'une feuille à plus de voisines que le nombre de texture de voisins disponibles, celles ayant la plus petite influence (du point de vue du phénomène simulé) sont ignorées.

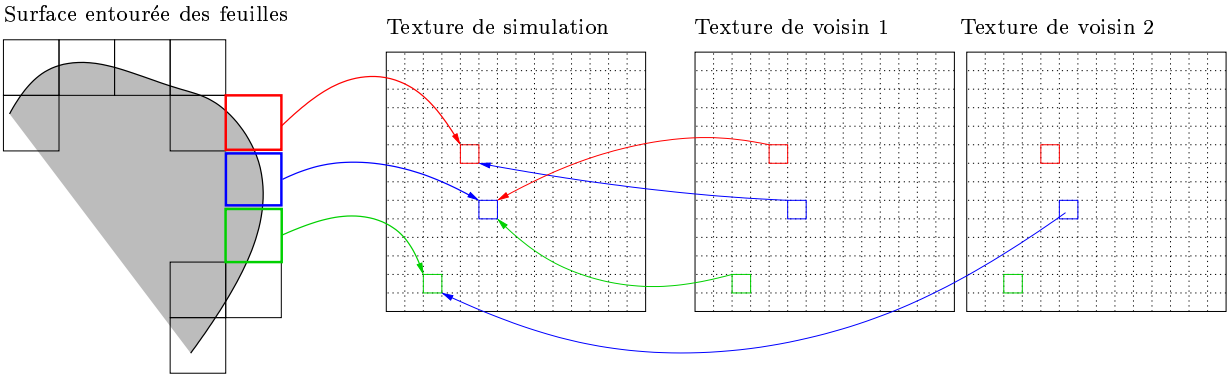


FIG. 8.1 – Organisation des données pour l'automate cellulaire.

La surface est entourée par les feuilles de la texture hiérarchique. Trois feuilles voisines sont surlignées. Chacune est associée à une case de la texture de simulation. Les textures de voisins indiquent les cases occupées par les feuilles voisines dans la texture de simulation.

3 Résultats

Nous avons implémenté cette approche afin de simuler l'écoulement d'un liquide le long d'une surface. Notre objectif est ici d'illustrer le fonctionnement de notre automate cellulaire plutôt que d'effectuer une simulation très réaliste. La simulation n'est donc pas basée sur la physique mais sur une brève étude du comportement d'un liquide sur une surface. Le liquide tend à former des gouttes qui ne tombent qu'à partir d'une certaine taille. Une fois que la goutte a commencé à s'écouler, elle ne s'arrête à nouveau que lorsque sa taille est passée en dessous d'un autre seuil, inférieur au premier. En outre, ces seuils sont plus faibles si la surface est déjà humide.

Notre automate simule la quantité de liquide à la surface et les échanges de liquide entre les feuilles de la texture hiérarchique. Afin de ne pas souffrir de distorsions, nous tenons compte dans la simulation de l'aire de la surface intersectée par chaque feuille (cette information est stockée dans les cases de la texture de simulation, avec la quantité de liquide simulée et l'humidité de la surface). La simulation est effectuée dans la texture de simulation par le processeur graphique. Le rendu final utilise la texture hiérarchique pour déterminer, le long de la surface, une couleur et une normale à partir de la quantité de liquide. La figure 8.2 montre un résultat de cette simulation. Nous recommandons de visualiser la vidéo et le programme de démonstration disponibles en ligne[1] afin de se rendre compte de la qualité de l'effet obtenu.

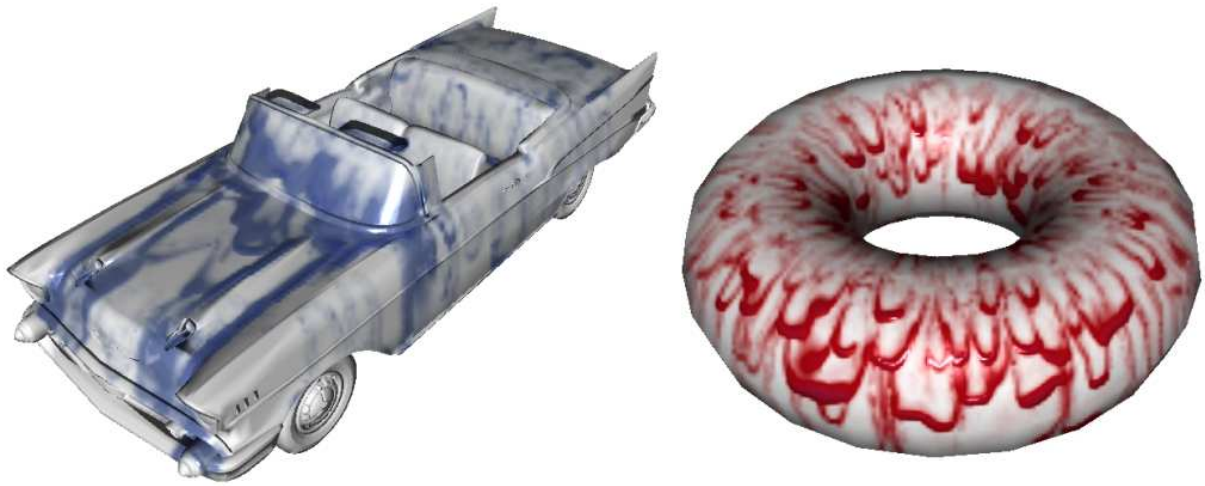


FIG. 8.2 – Liquide s'écoulant le long d'une surface.

L'utilisateur peut interactivement ajouter du liquide sur la surface à l'aide de la souris. Cette simulation est exécutée à environ 30 images par seconde sur une GeForce 6800 GT.

[1] http://www-evasion.imag.fr/Membres/Sylvain.Lefebvre/these

4 Conclusion

Nous avons montré comment simuler, sur la surface d'un objet, un habillage dynamique via un automate cellulaire. La simulation reproduit l'écoulement d'un liquide le long d'une surface. L'utilisateur peut influer sur la simulation en ajoutant interactivement du liquide. L'automate est entièrement mis à jour par la carte graphique, qui est très efficace pour ce type de calculs (opérations similaires effectuées dans toutes les cases d'un tableau) (voir chapitre 2, section 7). Ce type d'approche semble très prometteur car il permet déjà de réaliser, à des performances interactives, des effets visuels jusqu'alors très difficiles à obtenir. De plus, le coût mémoire est faible grâce à l'approche hiérarchique.

La texture hiérarchique nous permet, ici, d'associer de petites parties de la surface à une cellule d'un automate cellulaire planaire. Ces petites parties de surface correspondent à l'intersection entre la grille 3D et la surface. En fait, on peut considérer que la texture hiérarchique nous permet de paramétrer une à une les petites zones de la surface découpées par la grille, en les associant à des cases de l'automate. Nous allons voir au chapitre suivant (chapitre 9) une approche qui exploite plus avant l'idée de stocker localement, autour d'une partie de la surface, une paramétrisation.

chapitre 9

Textures composites

Nous introduisons, dans ce chapitre, la notion de *textures composites*. Ce nouveau modèle d'habillage *générique*, permet d'habiller avec une très haute résolution apparente des *surfaces non paramétrées*, à faible coût mémoire, en combinant les contributions d'un ensemble de motifs positionnés le long d'un objet. Il est possible de créer une grande variété d'apparences, y compris des aspects de surface animés ou dynamiques. Ce modèle d'habillage est conçu et implémenté pour les processeurs graphiques actuels.

Les recours à des motifs permet efficacité de stockage, qualité d'affichage et souplesse de création :

- Les motifs sont *localement* appliqués à la surface, à la manière d'autocollants. Ceci permet de réduire les distorsions introduites par les courbures de la surface.
- Les motifs ne sont stockés qu'une seule fois en mémoire. Seule l'information de positionnement est nécessaire pour les appliquer plusieurs fois sur la surface. Ceci permet d'obtenir une très grande résolution apparente à faible coût mémoire.
- Les motifs peuvent être manipulés indépendamment, de manière interactive. Ceci permet l'édition interactive de l'habillage par les artistes, mais permet également de créer des habillages dynamiques, pour lesquels le positionnement et l'animation des motifs sont gérés par programme. Il est ainsi possible de créer des habillages dont l'apparence s'adapte aux déformations du modèle (voir figure 9.5).
- La texture résultant de la combinaison des motifs est correctement filtrée et interpolée, grâce au filtrage et à l'interpolation effectués indépendamment sur la texture de chaque motif par le processeur graphique.

S'il suit l'idée d'utiliser des motifs développée au chapitre 3, ce nouveau modèle d'habillage est destiné à habiller des objets aux formes complexes, plutôt qu'à l'habillage de larges surfaces quasi–planaires, comme les terrains. Sur ce type de surfaces, les enjeux sont un peu différents. La taille des surfaces est généralement limitée, même si l'on souhaite tout de même atteindre des résolutions de détail qui peuvent vite saturer la mémoire disponible. Le problème vient principalement de la complexité de la forme de la surface, qui entraîne, avec le placage de texture standard, gaspillage et défauts visuels (voir chapitre 2, section 2.1.6), et rend difficile la création d'un pavage ; mais également de l'hétérogénéité des besoins en détails (souvent très localisés).

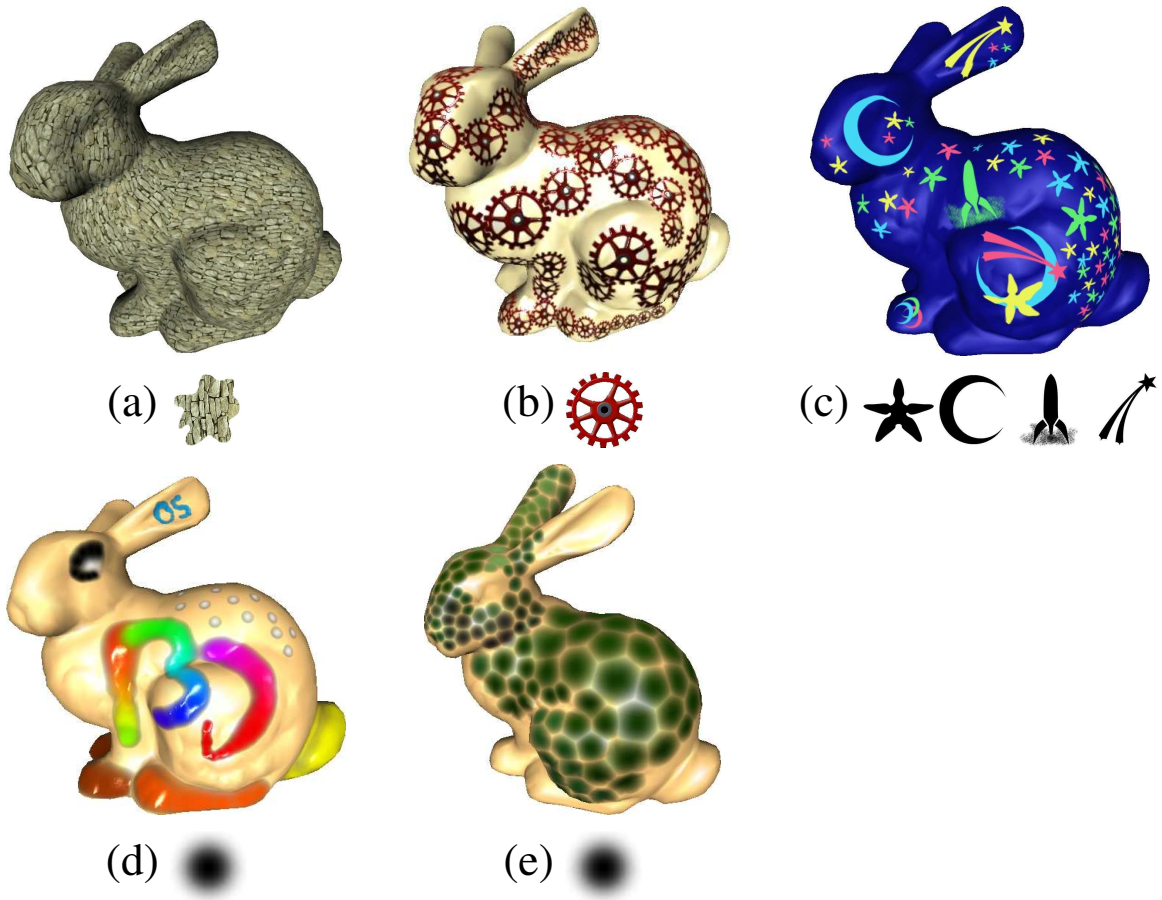


FIG. 9.1 – Exemple de modèles habillés par nos textures composites.

De gauche à droite *: (a) Lapped textures créée par Praun et. al [PFH00] et affichées par notre méthode. (b-c) Particules de texture animées. (d) Blobby–painting (e) Pavage de Voronoi entre particules de texture. La ligne du bas montre les particules de textures utilisées. Celles–ci ne sont stockées qu'une seule fois en mémoire. Tous ces lapins sont rendus avec une seule passe, en temps réel, et habillés par textures composites. Les particules composant la texture (ou sprites) peuvent être dynamiquement déplacées. La géométrie d'origine n'est pas modifiée.*

Notons que les *textures composites*, bien que basées sur les motifs, permettent de représenter une grande variété d'aspects de surface (voir figure 9.1). Tout d'abord, nombre de textures naturelles peuvent être obtenues en agençant des motifs sur une surface, des algorithmes de création dédiés ayant d'ailleurs été conçus à partir de cette observation [FLCB95, PFH00, DMLG02]. Nous avons déjà évoqué le fait que ce type d'habillage, basé sur des *particules de texture*, est mal représenté par les approches actuelles (du point de vue stockage et affichage, voir chapitre 2, section 2.3). Les textures composites capturent particulièrement bien ce type d'apparence et peuvent être directement utilisées à cet effet. Cependant, il est également possible de créer des habillages pour lesquels les motifs ne sont plus apparents, et ceci, grâce à la possibilité de combiner arbitrairement leurs contributions (voir figure 9.1). De plus, les motifs peuvent être efficacement animés,

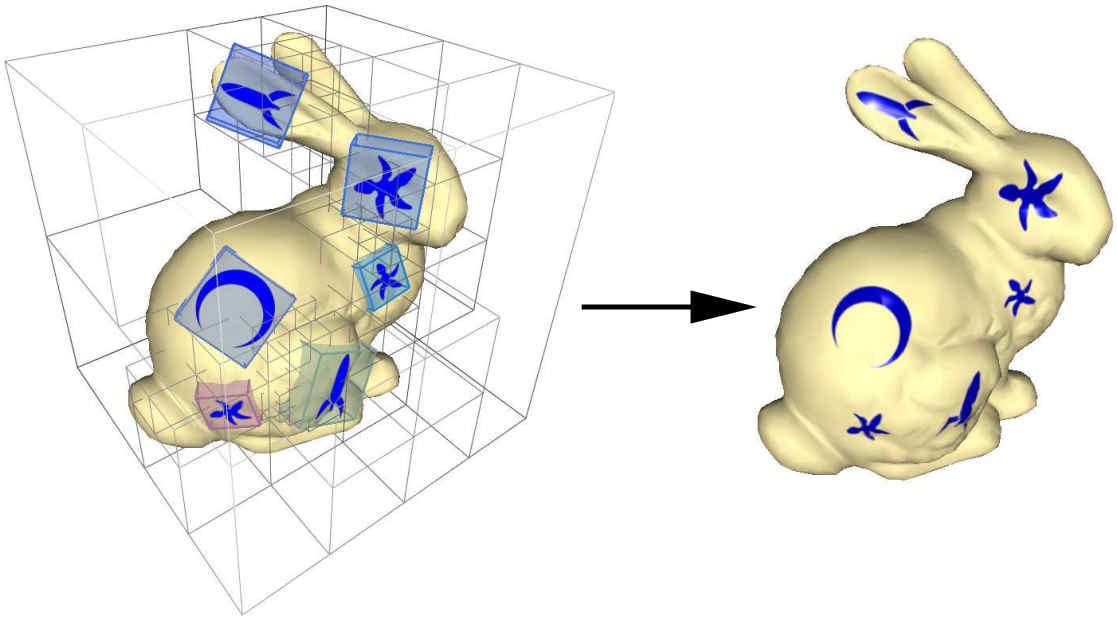


FIG. 9.2 – Principe de nos textures composites

Les attributs des sprites sont stockés dans une grille hiérarchique similaire à un octree.

permettant ainsi de créer des habillages qu'il serait très difficile – et inefficace – d'obtenir avec d'autres méthodes (par exemple, les écailles coulissantes du serpent).

Afin d'obtenir un stockage efficace des motifs, notre idée est de stocker dans un octree construit autour de la surface, non pas des couleurs, mais les paramètres permettant d'appliquer localement un motif à la surface. Chaque motif est donc représenté par un identifiant de texture et une paramétrisation locale (par exemple une projection planaire). Il est possible de rapidement connaître, en un point de la surface, les contributions des différents motifs, et donc de calculer l'apparence de la surface en ce point. Grâce à notre implémentation des *octree textures* sur cartes graphiques présentée au chapitre 7, nous pouvons fournir une implémentation complète de nos textures composites sur carte graphique . Ceci permet d'atteindre des performances autorisant leur utilisation dans les applications interactives.

Remarque : Ces travaux ont fait l'objet d'un rapport de recherche INRIA (rapport n°5209 [LHN04]) et d'une publication à la conférence I3D (*ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*) en 2005, sous le titre *Texture Sprites : Texture Elements Splatted on Surfaces* [LHN05b]. Ils ont été développés en collaboration avec Samuel Hornus, doctorant dans l'équipe ARTIS du laboratoire GRAVIR.

1 Textures Composites

Cette section décrit plus en détail notre approche. Les textures composites sont formées par le mélange de motifs appliqués localement à la surface. Puisqu'ils peuvent être animés, nous appelons ces motifs *sprites de texture* par analogie avec les petits objets animés des premières applications graphiques. Nous stockons les attributs décrivant un sprite (position, orientation, etc ...) dans une structure hiérarchique entourant l'objet. Cette structure correspond au N^3*-tree* présenté au chapitre 7. Plus d'informations sur les structures en volume utilisées pour habiller les surfaces sont données en section 2.4 du chapitre 2. La structure de données est très compacte car l'information est stockée uniquement autour de la surface et les textures sont partagées entre les sprites. En outre, nous pouvons entièrement encoder notre structure de données dans la mémoire texture et y accéder depuis un *fragment program*. A partir des coordonnées du point de la surface, les paramètres des sprites y ayant une influence sont retrouvés et l'apparence finale est calculée à partir de leurs contributions. Notre structure peut donc être utilisée dans les applications interactives. Comme elle définit un habillage à partir d'informations stockées en volume, le recours à une paramétrisation planaire globale est évité. De plus, la géométrie texturée n'a pas à être modifiée. La texture composite résultant de la combinaison des contributions des sprites est interpolée et filtrée correctement.

Ce type de texture est facile à éditer : L'utilisateur peut simplement cliquer sur la surface et déplacer les sprites à l'aide de la souris. Il peut changer leur taille et leur orientation. Ces contrôles peuvent également être gérés par une simulation afin de créer des textures dynamiques (de la même manière que la technique présentée au chapitre 4). Nos textures composites sont très flexibles : l'ordre des sprites est contrôlable et les contributions des sprites se recouvrant peuvent être mélangées pour générer des effets d'habillage inédits (voir figure 9.1). Les sprites peuvent également réagir à des déformations de surface, comme illustré figure 9.5.

Au delà des applications interactives, nos textures composites sont également intéressantes pour les applications de rendu logiciel, pour réduire la taille nécessaire au stockage de ce type de textures tout en évitant les défauts visuels. Comme nous disposons d'une implémentation sur carte graphique efficace, nous nous concentrons ici sur sa description. Cette implémentation est très facile à transposer dans un logiciel de rendu.

Nous supposons, dans la suite de ce chapitre, que le lecteur est familiarisé avec la structure de données présentée au chapitre 7. La section 2 présente comment utiliser un N^3-tree pour stocker les attributs des sprites et y accéder. La section 3 explique comment nous filtrons la texture composite et comment les contributions de différents sprites sont combinées. Enfin, la section 4 présente plusieurs effets d'habillage de surface rendus possibles par nos textures composites, ainsi que leurs performances.

2 Gestion des sprites dans la grille hiérarchique

Chaque sprite est décrit par un ensemble de paramètres. Ceci inclut les paramètres de positionnement (section 2.3) et une boîte englobante qui définit les limites de la contribution du sprite sur la surface. Cette boîte englobante est utilisée pour déterminer quelles cases de la grille

hiérarchique doivent contenir les informations concernant le sprite (voir figure 9.2). Elle est également utilisée pour détecter les collisions entre sprites (section 2.1).

2.1 Stockage des attributs des sprites

Les paramètres d'un sprite doivent être stockés dans les feuilles du N^3-tree qui intersectent la boîte englobante du sprite. Ces paramètres sont un ensemble de p nombres flottants. Comme un sprite va probablement intersecter plusieurs feuilles du N^3-tree, stocker directement les paramètres dans les feuilles ne semble pas une bonne idée. En effet ceci gaspillerait de la mémoire (le nombre de paramètres est relativement élevé), mais surtout compliquerait leur mise à jour dynamique. A la place, nous introduisons une indirection. La *table des paramètres* est une texture qui contient, dans chacune de ses cases, les p paramètres d'un sprite. Le sprite est associé à l'indice de la case contenant ses paramètres. Nous stockons cet indice dans les feuilles du N^3-tree. La valeur *RGB* est utilisée pour encoder la coordonnée de la case contenant les paramètres du sprite, dans la table des paramètres.

La table des paramètres doit être allouée en mémoire texture avec une taille qui permette de contenir les paramètres de suffisamment de sprites. Ce nombre maximal de sprites est noté M. Si d'autres sont ajoutés, la table doit être ré–allouée et les anciennes valeurs recopiées.

Sprites se recouvrant Deux sprites, ou plus, peuvent se recouvrir : leurs boites englobantes sont en collision. Dans ce cas, une même feuille du N^3-tree doit pouvoir stocker les informations de tous les sprites.

A cette fin, nous intercalons une seconde indirection entre les feuilles et la table des paramètres (voir figure 9.3). La *table des vecteurs* est une texture 3D qui stocke les indices de tous les sprites insérés dans une feuille. Chaque feuille possède un indice 2D dans la table des vecteurs. Cet indice permet de retrouver le vecteur qui stocke les indices des sprites dans la table des paramètres. La troisième dimension permet de parcourir ce vecteur et donc l'ensemble des sprites présents dans la feuille. Nous utilisons une valeur spéciale pour désigner les cases vides. Cependant, le nombre maximal de sprites pouvant être présents dans une même feuille est fixe. Il s'agit bien d'un vecteur et non pas d'une liste. Le nombre maximal de sprites pouvant être insérés est noté O_{max}. Ce nombre correspond au nombre maximal de sprites contribuant à un même point de la surface [1]. Désormais, au lieu de stocker dans les feuilles de l'arbre l'indice d'un sprite dans la table des paramètres, nous stockons l'indice du vecteur de la feuille dans la table des vecteurs.

La structure complète est illustrée figure 9.3.

2.2 Ajout d'un sprite, construction de la grille hiérarchique

A chaque insertion, il faut ajouter le sprite aux feuilles du N^3-tree intersectant sa boîte englobante. De plus, si la limite maximale O_{max} de sprites par feuille est atteinte, il faut subdiviser la grille. Notre algorithme d'insertion (effectué sur le CPU) est le suivant :

[1] En pratique nous utilisons une valeur $O_{max} = 8$.

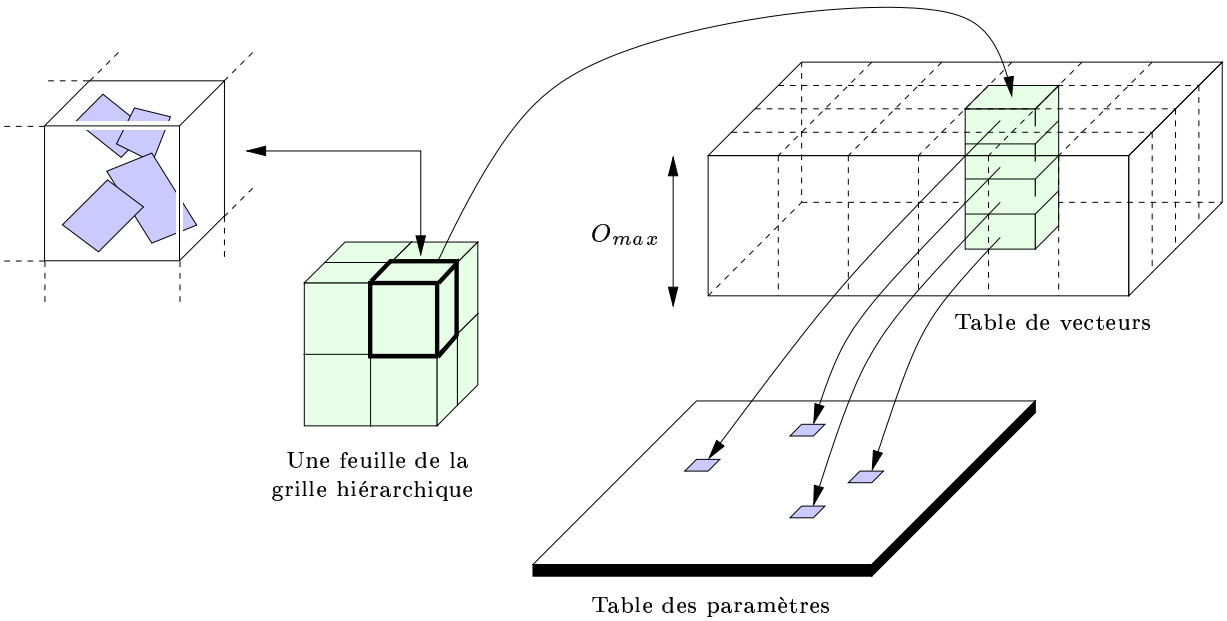


FIG. 9.3 – Structure de données.
Chaque feuille stocke l'indice d'un vecteur. Chaque case d'un vecteur stocke l'indice des paramètres d'un sprite dans la table de paramètres.

```
ajoutSprite(node n, sprite s) :
 si (n est une feuille)
   si (nombre de sprites dans n < Omax) insère s dans n
   sinon
    si (s en collision avec tous les sprites dans n)
     erreur(Omax trop petit !)
    sinon
    {
     si (profondeur maximale atteinte)
      erreur(Profondeur max !)
     subdivise n
     ajoutSprite(n,s)
    }
 sinon
    pour tous les fils c de n intersectant s
     ajoutSprite(c,s)
```

Les feuilles ne sont pas subdivisées tant que la limite de O_{max} sprites n'est pas atteinte. Ceci permet de minimiser la profondeur de l'arbre et de remplir les feuilles au maximum. Maintenir une faible profondeur de l'arbre permet d'avoir de larges feuilles. Ceci améliore les performances sur la carte graphique : de larges ensembles de pixels sur la surface effectuent le même traitement (ce qui augmente la cohérence pour le cache de texture) et moins d'indirections sont nécessaires pour descendre la hiérarchie.

Généralement lorsqu'une feuille contient O_{max} sprites, le nombre de sprites en intersections C_{max} est en fait inférieur : $C_{max} \leq O_{max}$. Si un nouveau sprite doit être inséré, il est donc possible de subdiviser la feuille afin que le nombre de sprites dans les fils soit inférieur à O_{max} (voir figure 9.4). Ceci peut parfois générer un ensemble de petits noeuds dans les zones où deux sprites sont difficiles à séparer. Cependant, ce cas est rarement observé en pratique.

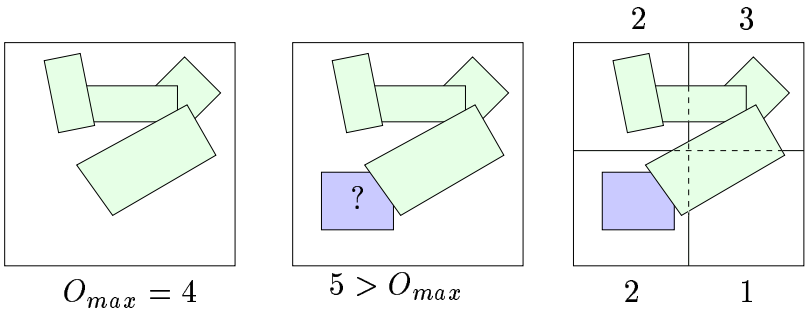


FIG. 9.4 – Insertion d'un nouveau sprite

Un nouveau sprite (bleu / sombre) est inséré dans une feuille déjà pleine. Comme le sprite n'est pas en collision avec tous les autres (vert / clair), il existe une subdivision permettant d'insérer le nouveau sprite. Subdiviser permet ici de conserver le nombre des sprites par feuille inférieur ou égal à O_{max}.

Echec d'insertion Si O_{max} n'est pas suffisamment grand, ou que la profondeur maximale autorisée de l'arbre est atteinte (voir chapitre 7), l'insertion échoue. Une application doit alors choisir de supprimer les sprites dans les zones de fort recouvrement.

Ces échecs se produisent uniquement dans les implémentations sur cartes graphiques à cause de limitations techniques (nombre d'instructions exécutées par pixel limité, branchements dynamiques non supportés, etc ...). Dans une application de rendu logiciel, O_{max} peut être changé dans chaque feuille en utilisant des listes à la place de vecteurs.

Ordre des sprites Dans certains cas, l'ordre dans lequel les sprites sont pris en compte lors du calcul de l'apparence finale de la surface peut être important (par exemple des écailles). Nous pouvons facilement contrôler l'ordre de parcours des sprites en échangeant leur rang dans le vecteur de chaque feuille.

2.3 Projection de la texture du sprite sur la surface

Jusqu'à maintenant, nous avons vu comment construire et stocker notre structure de données dans la mémoire texture. Il nous est donc maintenant possible, depuis un *fragment program*, de récupérer les paramètres de tous les sprites influençant un point donné de la surface, en accédant à l'octree, puis aux sprites contenus dans les feuilles. Nous décrivons, dans cette section, comment la texture associée au sprite est alors projetée sur la surface. En fait, il nous faut calculer des coordonnées de texture à partir des paramètres du sprite et du point de la surface.

Parmi les paramètres d'un sprite, nous stockons un repère définissant un plan, une orientation et une mise à l'échelle. Nous calculons les coordonnées de texture en projetant le point de la surface sur le plan support du sprite (la taille des sprites est supposée petite en comparaison de la courbure : la surface est localement planaire). Les paramètres de positionnement sont (il s'agit des paramètres stockés dans la table de paramètres, voir section 2.1) :

- un centre c.
- deux vecteurs l, h définissant un plan, une orientation et une échelle.
- une normale n définissant une direction de projection.

Soit M la matrice (l,h,n). Une fois les paramètres d'un sprite récupérés pour un point P de la surface, nous calculons les coordonnées de texture $U = (u,v,w)$ avec $U = M^{-1} \cdot (P - c)$. La coordonnée w peut être utilisée pour définir des textures volumiques, mais sert aussi dans le cas 2D : lorsque deux faces de la surface sont très proches (replis) nous pouvons utiliser w et n pour limiter l'influence du sprite dans une couche autour de la surface. Ceci permet de résoudre l'assignement de couleur ambigu décrit par Debry et al. et Benson et al. [DGPR02, BD02].

La projection de la texture sur la surface est équivalent à définir une paramétrisation locale planaire. Lorsque les sprites sont suffisamment petits par rapport à la courbure, peu de distorsions apparaissent. Dans le cas contraire, l'application est en charge de subdiviser un sprite en plusieurs sous–sprites, chacun supportant une petite partie de la texture du sprite initial. Notre approche permet ce type de décomposition, mais le calcul est à la charge de l'application utilisant la texture composite. En pratique, pour éviter l'apparition de défauts visuels à cause de distorsions dans la projection, nous atténuons la contribution des sprites avec le produit scalaire entre la normale à la surface et la direction de projection n.

Notons que nous envisageons d'étudier d'autres types de projection (cylindriques, morceaux d'ellipsoïdes, ...) permettant d'éviter les distorsions locales tout en restant rapides à évaluer. Cette étude est laissée en travaux futurs.

Habillage suivant les déformations Lorsque l'objet sur lequel est appliqué une texture en volume est animé ou déformé, l'approche usuelle consiste à utiliser les coordonnées de texture 3D définies sur l'objet au repos (voir chapitre 2, section 2.4). Dans ce cas, l'apparence de la texture est déformée en même temps que la surface : la texture se ressert dans les zones compressées, et est étalée dans les zones étirées. Si cela convient dans certains cas (peau), nous verrons comment réaliser des textures dont l'apparence s'*adapte* à la déformation (voir figure 9.5).

Notons également que plusieurs maillages, représentant une même géométrie, peuvent partager une même texture composite. Dans le cas des surfaces de subdivisions, ou des maillages progressifs [Hop96], il suffit que la texture soit appliquée sur le maillage le plus grossier. Durant le raffinement, les coordonnées 3D de texture des nouveaux points doivent être simplement interpolées linéairement, depuis les coordonnées des points du maillage de base (les nouveaux points, dans l'espace texture, restent sur les faces du maillage grossier).

En fait, le recours à des triangles n'est même pas vraiment nécessaire : avec une texture volumique, un nuage de points représentant une surface peut également être texturé.

2.4 Combiner les contributions des différents sprites

Lorsque plusieurs sprites se recouvrent, la couleur résultat est calculée à partir d'un mélange des contributions de chacun. De nombreuses manières de combiner les sprites peuvent être définies. Contrairement aux approches géométriques pour les particules de texture (voir chapitre 2, section 2.3), nous ne sommes en rien limités aux opérations de mélange standard (addition, soustraction, etc ...).

Nous avons créé de nouveaux effets d'habillage en combinant les sprites à la manière d'une surface implicite (*blobby-painting*, figure 9.1(e)) ou de pavages de Voronoi (figure 9.1(f)). Dans chaque cas, la texture représente une fonction de distance par rapport au centre du sprite. Dans le cas du *blobby-painting*, la texture est définie par une iso–distance sur la somme des distances au centre des sprites. Dans le cas du pavage de Voronoi, le sprite le plus proche est choisi comme seul contribuant.

3 Filtrage de la texture composite

Comme nous l'avons vu dans l'état de l'art (chapitre 2, section 4) le filtrage est essentiel pour éviter l'aliasing . Créer des modèles d'habillage complexes est une chose, mais il reste à les filtrer correctement pour obtenir une qualité de rendu satisfaisante.

Nous pouvons distinguer trois cas : interpolation linéaire, filtrage (MIP–mapping) des sprites et filtrage de la structure hiérarchique (N^3-tree).

Interpolation linéaire L'interpolation linéaire de la texture de chaque sprite est gérée automatiquement par la carte graphique . Lorsque la texture du sprite est lue à partir des coordonnées calculées (voir section 2.3), la carte graphique peut en effet interpoler la texture comme d'habitude. Si la combinaison des contributions de différents sprites est linéaire, alors la texture composite est correctement interpolée linéairement.

Filtrage des sprites Le filtrage de la texture de chaque sprite peut être géré automatiquement par la carte graphique . Du moment que l'équation de combinaison des contributions des sprites reste linéaire, la texture résultante est correctement filtrée (voir chapitre 2, section 4.3.2). Cependant, comme nous calculons explicitement les coordonnées de texture (voir section 2.3), la carte graphique ne peut plus connaître automatiquement les dérivées partielles nécessaires à l'évaluation de la taille de l'empreinte du pixel (voir chapitre 2, section 4.3.1.1). Nous devons donc calculer explicitement ces dérivées partielles (nous utilisons les intructions `ddx` et `ddy` des langages HLSL et Cg).

Filtrage du N^3-tree Si le point de vue est très loin de l'objet, de multiples feuilles du N^3-tree vont se projeter dans un même pixel. De l'aliasing peut alors se produire puisque seule une des feuilles se projetant dans le pixel déterminera la couleur finale. Résoudre cette situation n'est pas simple. Nous allons d'abord voir quelles sont les solutions possibles. Nous expliquerons ensuite pourquoi ce filtrage n'est pas toujours nécessaire en pratique.

Il est possible d'éliminer complètement l'aliasing en filtrant la texture composite. La difficulté réside dans le fait que la texture est implicitement créée par la combinaison des sprites lors du rendu. Deux approches sont possibles :

- Première approche : effectuer le filtrage sur la grille hiérarchique (voir également chapitre 7, section 2.2). Il faut alors stocker dans chaque noeud de l'arbre la couleur moyenne de ses noeuds fils. La couleur moyenne d'une feuille doit être calculée à partir des sprites qu'elle contient. Il faut considérer uniquement la partie des sprites incluse dans la feuille et, bien entendu, appliquer la même fonction de mélange que celle utilisée pour le rendu avant de calculer la couleur moyenne. Cette méthode augmente le stockage (il faut stocker une couleur par noeud de la grille), mais elle est également plus lente. Il faut en effet déterminer, pour chaque noeud rencontré lors de la descente de l'arbre, s'il faut s'arrêter (et utiliser la couleur moyenne du noeud) ou bien continuer la descente. Celle–ci doit être stoppée avant que la taille du noeud projeté à l'écran soit inférieure à la taille d'un pixel. Si une feuille est atteinte, la texture est visible à pleine résolution et le filtrage de l'arbre n'est pas nécessaire. Pour une qualité optimale, il faut également effectuer une interpolation linéaire entre noeuds voisins (ceci correspond à l'interpolation linéaire effectuée dans les niveaux de la pyramide de MIP–mapping d'une texture standard). Cette interpolation est coûteuse puisqu'il faut accéder aux noeuds voisins de même résolution et aux noeuds parents (voir également [DGPR02, BD02]). Enfin, toute mise à jour de l'arbre implique de recalculer les couleurs moyennes des feuilles vers la racine. Il est donc recommandé de réserver cette approche pour les applications de rendu non–interactif où la qualité prime sur les performances.
- Seconde approche : calculer une paramétrisation planaire de l'objet et convertir la texture composite en une texture 2D standard (voir chapitre 7, section 2.3). Cette texture ne sera utilisée que pour les points de vue éloignés. Elle peut donc être de faible résolution, et la paramétrisation de faible qualité, puisque l'important sera uniquement de capturer correctement la couleur moyenne pour les points de vue éloignés. Cependant, la résolution de la texture 2D doit être suffisamment grande pour que les feuilles de la grille hiérarchique ne produisent pas de l'aliasing lors de la conversion (il est toujours possible, ensuite, de réduire la résolution de la texture pour diminuer le stockage). Comme précédemment, il faut utiliser la texture 2D filtrée lorsqu'un noeud devient plus petit qu'un pixel. Cette solution, puisqu'elle s'appuie sur des textures 2D standards, offre de meilleures performances. Cependant, la qualité de rendu peut être inférieure, à cause des défauts possiblement introduits par la paramétrisation.

Heureusement, ce type d'aliasing n'apparaît que rarement en pratique. Tout d'abord, notre algorithme d'insertion (voir section 2.2) tente de minimiser la profondeur de l'arbre, créant ainsi de larges feuilles. Ensuite, de petites feuilles voisines encodent souvent des ensembles de sprites similaires et vont donc avoir une couleur moyenne proche : l'aliasing ne sera pas visible.

L'un des cas difficiles concerne les petites feuilles qui peuvent être créées pour séparer deux sprites proches mais qui ne se recouvrent pas (un cas rare en pratique). Supposons que l'un des sprites est blanc et le second noir. Certaines des petites feuilles qui séparent les sprites seront uniquement noires, les autres uniquement blanches, provoquant de l'aliasing sur leur frontière. Pour éviter ce problème, il est recommandé de conserver un bord transparent autour des images

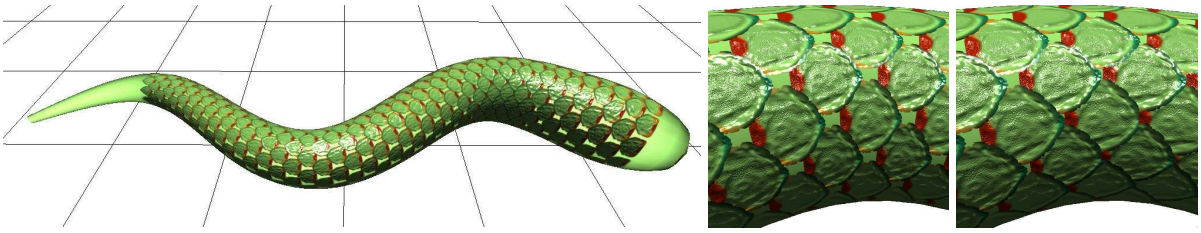


FIG. 9.5 – Habillage de la surface avec des écailles rigides.

Un serpent ondulant couvert de 600 écailles se recouvrant (chaque écaille est un sprite). La texture du motif d'écaille a une résolution de 512×512*. La résolution de la texture composite est équivalente à une texture de résolution* 30720×5120*. Il est possible d'observer le filtrage correct dans les zones inclinées et aux bordures de sprites. Cette figure montre que chaque écaille est indépendante et s'adapte à la déformation : les écailles coulissent les unes sur les autres.* Au milieu : *Sans compensation de déformation la texture est écrasée dans les zones de compression.* A droite : *Avec compensation de déformation, les écailles deviennent rigides et coulissent (voir également la vidéo).*

des sprites. Ceci augmente le recouvrement, mais deux sprites ainsi séparés n'auront pas d'influence véritable dans les petites feuilles sur leur pourtour (ils sont transparents), évitant ainsi l'aliasing pour les points de vue proches. Bien entendu, ceci repousse le problème plutôt que de le résoudre : au– delà d'une certaine distance de vue l'aliasing pourra redevenir visible (et peut–être traité avec une des deux approches présentées ci–dessus). Cette distance dépend de la taille de la bordure transparente introduite autour du sprite.

Notons finalement que, dans le cas des surfaces complètement recouvertes de sprites de couleurs moyennes proches (voir figure 9.1 *(a)*), le filtrage de la grille hiérarchique n'est pas nécessaire. Quelle que soit la feuille qui apparaît dans le pixel, sa couleur est proche de la couleur des autres feuilles présentes dans ce même pixel, masquant ainsi l'aliasing .

Pour l'ensemble de ces raisons, nous n'avons pas implémenté, en pratique, ce dernier cas de filtrage. Comme démontré par la vidéo, aucun défaut visuel n'apparaît pour des points de vue raisonnablement proches. Notons que les exemples de la vidéo restent correctement filtrés jusqu'à ce que l'objet ne fasse que quelques dizaines de pixels sur l'écran - l'aliasing géométrique devient prépondérant bien avant que la texture composite ne crée des défauts visuels.

4 Nouveaux habillages et résultats

4.1 Nouveaux habillages

Nous avons créé de nouveaux habillages de surface à partir de nos textures composites, comme illustré figure 9.1, figure 9.5 et sur la vidéo [2].

Création interactive de textures (figure 9.1(c) et vidéo) Dans cette application, l'utilisateur peut interactivement coller et déplacer des sprites sur la surface de l'objet. Après avoir

[2]Disponible à l'adresse `http://www-evasion.imag.fr/Membres/Sylvain.Lefebvre/these`

défini un ensemble de textures, un simple “clic” suffit pour ajouter un sprite. Ceux-ci peuvent être tournés ou déplacés. Leur ordre d'affichage peut être changé. Le lapin ”ciel nocturne” a été réalisé en quelques minutes.

Ceci illustre bien comment nos textures composites peuvent être utilisées. Ici, l'application implémente l'interface utilisateur et s'occupe de positionner les sprites (un simple *picking*). Les sprites sont alors insérés dans notre structure qui prend en charge stockage efficace et affichage de qualité en temps réel. Les informations de positionnement peuvent être mises à jour dynamiquement.

Remarquez que les sprites peuvent se recouvrir, mais que de larges zones de la surfaces peuvent rester non couvertes (cet espace vide *n'est pas* stocké). Ceci permet d'utiliser une texture standard sous la couche de sprites. En particulier, cela permet d'utiliser de multiples couches de sprites pour dépasser le nombre maximal O_{max} de recouvrement.

Approximation de *Lapped Textures* (figure 9.1(a)) Cet exemple a été créé à partir d'une sortie de l'algorithme de *lapped texture*[3] [PFH00]. Notre représentation de textures composites correspond bien à l'approche de cet algorithme de synthèse de texture basé sur le recouvrement d'une surface par un ensemble de petits éléments supportant la même texture (analogues à des autocollants).

Les *lapped textures* sont efficacement stockées par notre représentation : le motif de base est stocké une seule fois à pleine résolution et le N^3-tree minimise la mémoire requise pour les informations de positionnement. De plus, l'affichage ne souffre d'aucun des défauts visuels crées par un atlas ou par le recours à de petits éléments géométriques. Le maillage géométrique n'est pas modifié. Aucune géométrie supplémentaire n'est nécessaire.

Eléments de texture animés (figure 9.1(b,c) et vidéo) Les sprites appliqués sur un modèle 3D peuvent être animés de deux manières. En premier lieu, l'application peut modifier dynamiquement leurs paramètres de positionnement (position, orientation, échelle) à chaque image. Cette opération est efficace grâce au mécanisme d'indirection (seule la case du sprite dans la table de paramètres doit être mise à jour). Des animations de particules comme des gouttes d'eau peuvent être ainsi réalisées. La figure 9.1(b) montre une texture composée d'engrenages positionnés par l'utilisateur. Chaque engrenage tourne indépendamment. L'animation est dynamiquement générée par la mise à jour d'un seul paramètre : le temps. En second lieu, la texture d'un sprite peut elle même être une séquence animée, simulant une animation à la façon des dessins animés. Les motifs de la figure 9.1.c sont animés de cette façon.

Ecailles de serpent (figure 9.5 et vidéo) Comme expliqué ci–dessus, chaque sprite peut être animé dynamiquement, et ce sans avoir à changer le N^3-tree, du moment que la boite englobante du sprite n'est pas modifiée.

Ceci permet de créer des textures dont l'apparence s'adapte aux déformations de la géométrie. Afin d'illustrer cet effet, nous avons créé une texture d'écailles de serpent. Lorsque la surface se

[3]Merci à Emil Praun et Hugues Hoppe pour nous avoir fourni les données.

déforme, les écailles coulissent les unes sur les autres (voir figure 9.5, au milieu et la vidéo). Avec le placage de texture standard, les écailles auraient été déformées sans coulisser. Nous estimons la déformation géométrique en chaque sommet et changeons la taille des sprites dans la texture pour compenser la déformation géométrique. Rappelons qu'aucune géométrie n'est ajoutée. Notre représentation définit réellement une texture qui peut être évaluée en tout point de la surface.

4.2 Performances

Temps de rendu Les performances des exemples présentés dans le chapitre sont résumées table 9.6. Les mesures ont été effectuées sur une GeForceFX 6800 GT, sans avoir recours au branchement dynamique. L'implémentation est réalisée en NVidia Cg. Les performances permettent d'utiliser nos textures composites dans les applications interactives. Le coût provient principalement du nombre de sprites se recouvrant. Nous n'avons pas tenté d'optimiser le code produit par Cg (version 1.3).

Sur un processeur graphique ne permettant pas les branchements dynamiques, un accès à notre texture composite requiert la même quantité de calculs que si O_{max} sprites étaient stockés à la plus grande profondeur. (Une conséquence positive est que le coût est constant quel que soit le nombre de sprites présents dans la texture).

Sur un processeur graphique avec branchement dynamique, nous sommes dans un cas favorable. En effet, les feuilles de la grille hiérarchique étant généralement larges, elles contiennent de grandes portions de la surface ce qui permet un branchement dynamique efficace (voir chapitre2, section 7).

Notons que le coût de rendu est complètement déterminé par le nombre de pixels à texturer. La bonne nouvelle est que l'on ne paie que les pixels visibles, à condition d'utiliser correctement le test de profondeur. D'autre part, les approches comme le *deferred shading* sont particulièrement intéressantes dans notre cas, pour ne pas dessiner un seul pixel en trop.

	nombre de sprites	taille d'un noeud	profondeur d'arbre	O_{max}	FPS $_{800\times600}$
Lapped	536	4	4	16	27
Gears	50	4	2	8	80
Stars	69	4	2	8	26
Blobby	125	4	3	10	70
Voronoi	132	4	3	12	56

FIG. 9.6 – Performances des exemples de la figure 9.1.

Taille mémoire Nos textures utilisent peu de mémoire en comparaison des textures standard qui seraient nécessaires pour obtenir un résultat équivalent. Nos tests montrent qu'un modèle de lapin texturé à l'aide d'un atlas de texture automatiquement généré par un logiciel de modélisation (voir vidéo) demanderait une résolution de 2048×2048 (16 Mo) pour un résultat équivalent

au nôtre. Les tailles des textures de nos exemples sont indiquées table 9.7. Les textures devant avoir des résolution en puissance de deux[4], nous devons allouer plus de mémoire que nécessaire. Nous indiquons donc, à la fois, la vraie taille de notre structure, et la taille qui doit être allouée. La dernière colonne de la table 9.7 indique la somme de la taille de la structure et de la taille des textures des sprites.

	taille de la structure	mémoire allouée	mémoire totale
Lapped	1 Mo	1.6 Mo	1.9 Mo
Gears	0.012 Mo	0.278 Mo	0.442 Mo
Stars	0.007 Mo	0.285 Mo	5.7 Mo
Blobby paint	0.141 Mo	0.418 Mo	0.434 Mo
Voronoi	0.180 Mo	0.538 Mo	0.554 Mo

FIG. 9.7 – Taille mémoire des exemples de la figure 9.1

5 Conclusion

Nous avons introduit une nouvelle représentation permettant d'habiller des surfaces arbitraires avec des textures composites : des motifs sont appliqués le long de la surface et combinés pour former une apparence complexe, éventuellement animée. Les textures produites ont de très hautes résolutions mais ne requièrent que peu d'espace mémoire, grâce au recours à une structure hiérarchique et au mécanisme de positionnement (instanciation). Elles peuvent être utilisées dans des applications interactives et sont affichées sans défauts visuels (grâce aux paramétrisations indépendantes des motifs qui réduisent les distorsions, et permettent un filtrage correct). La représentation est très flexible : les motifs, ou *sprites*, peuvent être animés et dynamiquement déplacés, leurs contributions sont arbitrairement mélangées, créant ainsi des aspects continus. Nous avons proposé de nouveaux habillages de surface, rendus possibles par notre approche (textures animées, effets de mélange complexes, écailles coulissantes).

Parmi les futures directions de recherche, nous souhaiterions développer de meilleures paramétrisations locales. Il serait également intéressant d'étudier comment utiliser cette approche pour générer des textures volumiques, éventuellement animées, à moindre coût.

Ce modèle d'habillage permet de représenter efficacement la très large classe des textures à base de particules (voir chapitre 2, section 2.3), à la fois en terme d'occupation mémoire, mais également en terme de qualité de rendu. La notion plus générale de *textures composites* permet de représenter des habillages pour lesquels les différents motifs ne sont plus apparents : leur combinaison permet de créer un aspect de surface plus complexe. Il s'agit donc d'un modèle d'habillage générique et applicable à tous types d'objets sans paramétrisation globale et sans dégradation de qualité. Il atteint en cela la plupart des objectifs que nous nous sommes fixés au début de cette thèse.

[4]Ce n'est plus le cas depuis peu ...

Cinquième partie

Modèle d’habillage dédié : génération d’écorces d’arbre réalistes

chapitre 10

Modèle d'habillage pour la génération d'écorces d'arbre réalistes

Les modèles d'habillage que nous avons vu jusqu'à maintenant sont génériques : ils permettent une représentation efficace d'une grande variété d'aspects de surface (chapitres 3, 7 et 9). Cependant, l'apparence finale de la surface doit créée par un artiste, via les outils du modèle d'habillage (certains modèles peuvent prendre en charge une partie de la création, voir chapitre 3). Malheureusement, il existe des aspects de surface qui sont, de part leur complexité, particulièrement difficiles à reproduire pour un artiste. Même s'il lui est possible d'habiller une seule géométrie, la tâche peut devenir impossible si de très nombreux objets du même type sont présents dans la scène.

Or, cela est typiquement le cas avec les écorces d'arbre. D'une part l'apparence des écorces est difficile à reproduire fidèlement (il faut tenir compte des embranchements et des variations de rayon le long du tronc et des branches), mais de plus une forêt contient des dizaines de milliers d'arbres, tous aux géométries différentes.

Nous proposons dans ce chapitre un modèle d'habillage capable de générer des apparences d'écorce réalistes pour des géométries d'arbres. Le modèle n'est pas destiné à l'affichage, mais à la génération de l'apparence de l'écorce sur la surface de l'arbre. Le modèle pré–calcule une apparence d'écorce, qui peut être convertie sous la forme de textures, ensuite appliquées au modèle géométrique de l'arbre. Notre algorithme simule la croissance des écorces en tenant compte, entre autre, des variations de rayon le long du tronc et des branches, et des embranchements. Parmi l'infinie variété d'écorces que l'on peut observer dans la nature, nous nous sommes plus particulièrement intéressés aux écorces présentant de larges déchirures. En effet, ce phénomène est très présent dans la nature et assez peu étudié jusqu'à aujourd'hui. La figure 10.2 présente quelques écorces de ce type.

De nombreux outils, basés sur des approches empiriques ou sur la biologie, permettent de créer des modèles géométriques d'arbres très réalistes [Bio, dREF$^+$88, Xfr, PLH88, LD98]. Des travaux étudient également la simulation d'écosystèmes [DHL$^+$98] ou l'influence des conditions d'éclairage sur la croissance des plantes [SSBD01]. Cependant, les points de vue rapprochés ne sont pas gérés du tout. Certains outils utilisent des pavages périodiques de textures

FIG. 10.1 – Ecorces générées à la surface d'arbres par notre modèle d'habillage .

(voir chapitre 2, section 3.2) pour habiller l'arbre d'écorce. Néanmoins, cela donne des résultats peu réalistes (pas de continuité aux embranchements, pas de contrôle de positionnement sur les éléments de la texture, et un étirement de l'apparence selon la taille des branches). Pour obtenir de bons résultats, les artistes doivent souvent peindre la surface entière ou doivent combiner de multiples textures [Stu99] ce qui est peu pratique et prend beaucoup de temps.

Les arbres ainsi générés peuvent être adaptés à des points de vue éloignés, mais la faible qualité de leur surface devient apparente dès que le point de vue se rapproche (moins de 10 mètres). Il y a un besoin réel, dans les applications actuelles (simulateurs, jeux vidéos, effets spéciaux et études d'impact sur l'environnement) pour un modèle d'habillage capable de représenter fidèlement les écorces avec des points de vue rapprochés – d'autant plus que les arbres sont un élément indissociable des scènes d'extérieur.

Ce chapitre débute par une présentation générale sur les écorces et notre approche en section 1. Nous verrons les travaux précédents portant sur les écorces et les phénomènes de déchirures, caractéristiques de la surface des arbres, en section 2, Nous présenterons ensuite notre algorithme de simulation d'écorce en section 3. Nous verrons en section 5 comment nous habillons le modèle d'arbre avec notre simulation. Nous verrons ensuite les résultats en section 6.

Remarque : Ces travaux ont fait l'objet d'une publication au *Eurographics Workshop on Rendering* en 2002 sous le titre *Synthesizing Bark* [LN02]

1 Présentation de notre approche

L'écorce est le résultat de la croissance de l'arbre. Le tissu du bois qui grandit en se divisant (le *phellogen*) est une couche de quelques millimètres à quelques centimètres située sous l'écorce. Cette couche produit du nouveau bois vers l'intérieur et de l'écorce nouvelle vers l'extérieur. La couche externe (l'écorce visible) est donc constituée de bois plus vieux que les

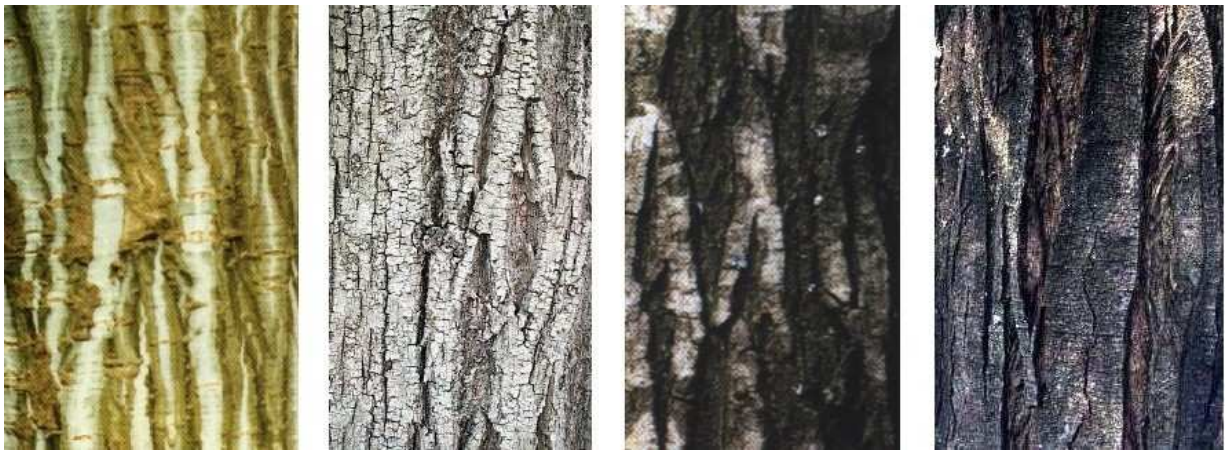


FIG. 10.2 – Ecorce réelles

Ecorce fracturées provenant de diverses espèces d'arbres (images du livre de Vaucher [Vau93] sur les écorces).

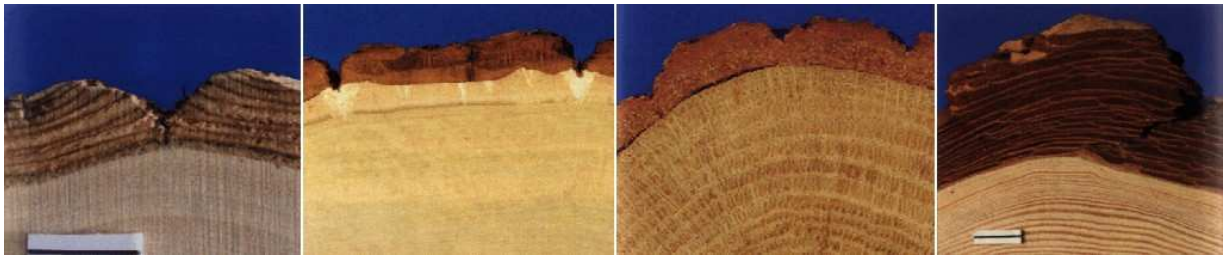


FIG. 10.3 – Coupe de troncs d'arbres

Coupe du tronc de diverses espèces, montrant la couche d'écorce et le profil des déchirures (images du livre de Vaucher [Vau93]).

couches internes de l'écorce. Ceci résulte en une forte tension tangentielle, qui produit des fractures, déchirements et autres effets résultant de contraintes mécaniques. La figure 10.3 présente ces effets sur de vraies écorces.

Pour être utilisable, un modèle de simulation d'écorce doit être suffisamment automatique pour gérer le travail de création pénible, mais doit impérativement laisser à l'artiste le contrôle sur le résultat final. Il ne serait pas raisonnable de tenter de simuler physiquement et de manière précise le matériau constituant une écorce. Le nombre de paramètres des phénomènes biologiques et physiques mis en oeuvre est énorme. De plus, un tel modèle ne permettrait pas un contrôle aisé sur le résultat. Une simulation coûteuse, par exemple basée sur des éléments finis, prendrait trop de temps pour permettre une édition interactive, pourtant nécessaire pour permettre à l'utilisateur de régler les paramètres.

Notre approche est à la fois basée sur la physique et empirique : nous simulons les déchirures de l'écorce sur un matériau simple, en tirant avantage des connaissances à priori du comportement de l'écorce. Notre simulation génère la structure des déchirures (position, forme, taille) ;

nous utilisons des textures pour habiller le résultat avec des détails et produire les textures d'écorce pour l'arbre.

Notre méthode est basée sur deux structures évoluant simultanément : des bandelettes circulaires d'écorce (entourant la section du tronc) et des déchirures orthogonales aux bandelettes (parallèles au tronc ou aux branches). Nous simulons, en 1D, la croissance des bandelettes et l'apparition des fractures causées par la tension accumulée. Nous simulons, dans la direction orthogonale, la propagation des fractures. Cette propagation est le seul couplage entre les différentes bandelettes, qui sont mécaniquement indépendantes. Cette approche est basée sur l'hypothèse que la croissance est orientée et que l'écorce n'a pas d'influence sur la croissance de l'arbre : elle subit la déformation et doit s'y adapter (hypothèse valide sur les arbres).

Lors de la simulation et de l'élargissement des déchirures, nous maintenons l'aspect de l'épiderme d'origine (avant apparition des fractures). L'intérieur des déchirures, lui, est habillé par une texture située entre les deux bords. Ce modèle est continu dans le temps (par construction) et permet de visualiser l'évolution de l'écorce. Au final, une texture globale contenant épiderme et fractures peut être générée et utilisée pour habiller le modèle d'arbre. Notons qu'il est également possible, si besoin, de générer une géométrie supplémentaire à la surface du tronc, représentant le relief de l'écorce.

Il est important de souligner que les bandelettes d'écorce simulées à la surface de l'arbre sont géométriques. Ceci permet d'éviter les distorsions et discontinuités dans l'apparence des déchirures. Elles sont dessinées, après simulation, dans une texture globale en tenant compte des éventuelles distorsions de la paramétrisation du tronc (et des branches) pour que l'aspect final, après placage de la texture, soit non distordu. Les fractures sont également continues entre le tronc et les branches. Néanmoins, il arrive parfois, contrairement à l'intuition, que les écorces réelles ne montrent pas de continuité entre les fractures du tronc et les fractures des branches. Ceci est notamment observable sur le dessus des branches. Nos contraintes de continuité sont donc seulement partielles.

2 Travaux antérieurs

Cette section discute des travaux effectués sur la simulation d'écorces et de fractures. La section 2.1 décrit principalement les travaux effectués en synthèse d'image avant 2002, date de publication de notre article sur le sujet. D'autres travaux ont été effectués par la suite, en particulier la méthode de Wang et. al [WWHG03], basée sur la synthèse de texture à partir d'échantillons, qui fourni de très bons résultats.

Nous verrons également en section 2.2 quelques éléments de physique, qui nous permettront de mieux comprendre les phénomènes de rupture et d'apparition de fractures dans un matériau.

2.1 Travaux en synthèse d'image

2.1.1 Travaux sur les écorces

Deux types d'approche existent dans ce domaine : l'utilisation du placage de texture pour un habillage statique ou la simulation.

Avec le placage de texture, le problème principal est, comme souvent, la paramétrisation (voir chapitre 2, section 2.1.2). Dans le cas des arbres, les problèmes apparaissent principalement à cause des variations de diamètre entre tronc et branches, et aux embranchements. Une simple texture répétée le long de l'arbre va présenter sur les branches la même apparence que sur le tronc mais à une échelle différente. Or, ce n'est pas ce qui peut être observé dans la nature. En particulier, sur les écorces présentant des déchirures, on observe une plus faible densité de fractures sur les branches que sur le tronc. Bloomenthal [Blo85] utilise une image rayon X d'un échantillon d'écorce d'érable pour créer une texture de bosselage (*bump mapping*). Il définit une paramétrisation des branches et embranchements à l'aide de splines, qui permet de bons résultats pour cette écorce particulière, même si les problèmes de distorsions et discontinuités perdurent. Maritaud *et al.* [MDG00] utilisent une carte de déplacement qui crée de la géométrie fine sur la surface, et une paramétrisation plus simple des embranchements : chaque tronc est paramétré indépendamment ; pour éviter les discontinuités visuelles les différentes textures sont mélangées avec un effet de transparence. Hart et Baker [HB96] s'appuient sur la paramétrisation générée par les surfaces implicites représentant l'arbre pour y appliquer une texture.

La seconde famille d'approches, comme celle proposée par Federl et Prusinkiewicz [PP96], utilise des réseaux masses–ressorts attachés à la surface de l'arbre. Les ressorts se brisent au–delà d'un certain effort, produisant les fractures. Hirota *et al.* [HTK98] utilisent plusieurs couches de masses–ressorts, étendant aux écorces leur travaux sur les fractures [HKK98]. Les deux méthodes créent de fines fractures, mais ne peuvent produire les larges déchirures observables sur les écorces. De plus, il faut utiliser un très grand nombre de ressorts pour produire des résultats convaincants : la taille et le nombre de fractures dépend directement de la résolution du réseau masses–ressorts.

2.1.2 Travaux sur la simulation de fractures

Puisque l'on s'intéresse à la formation de déchirures à la surface d'un matériau subissant une contrainte de déformation (l'écorce), les travaux sur la simulation des fractures nous intéressent directement.

Terzopoulos *et al.* [TF88] ont introduit une première représentation des fractures ainsi qu'un modèle physique de matériau élastique, plastique et visco–élastique. Les fractures sont représentées comme des discontinuités dans le modèle de matériau. Norton *et al.* [NTB+91] proposent une approche pour représenter les objets solides comme des cubes reliés par des ressorts, qui peuvent se briser pour fracturer les objets. Dans le même esprit, Smith *et al.* [SWB00] proposent une méthode basée sur la représentation des objets solides sous forme de tétraèdres. Dans ces deux approches les fractures doivent être situées le long des jonctions entre éléments (cubes ou tétraèdres). Ceci impose une fine discrétisation pour éviter de voir les éléments. Muller *et al.* [MMDJ01] étendent ce principe à la simulation temps–réel en construisant les fractures lors des impacts et en simulant l'objet comme un solide rigide le reste du temps. Comme dans les méthodes précédentes, les fractures doivent suivre les bords de tétraèdres. Dans ce cas, augmenter la résolution interdit les performances temps–réel et le biais est donc très visible dans les résultats. O'Brien et Hodgins [OH99] utilisent des éléments finis pour déterminer l'origine et l'orientation des fractures. Ceci donne des résultats très réalistes, au prix d'un temps de calcul considérable. Neff *et al.* [NF99] utilisent un modèle procédural de motifs de fractures pour simuler des fenêtres se brisant sous l'effet d'une onde de choc. Le modèle procédural évite le coût important d'une simulation. Cependant, la méthode n'est proposée que sur un objet plat.

Notons que ces travaux sont intéressants concernant la simulation des fractures, mais ils ne prennent pas en compte certains éléments spécifiques aux écorces, comme l'interaction entre la couche d'écorce et le tronc, ou l'aspect de l'intérieur des déchirures.

2.2 Eléments de physique

Une rupture apparaît dans un matériau lorsque les contraintes internes dépassent un seuil de résistance. Le lien entre atomes est alors brisé et la rupture s'initie. Celle-ci se propage tant que le système n'est pas revenu dans un état énergétique stable, à l'équilibre. Les fractures se propagent orthogonalement à la direction d'effort maximal. Les imperfections du matériau ont une forte influence sur son comportement en rupture.
Il faut distinguer deux modes de rupture :

- **rupture fragile :** ce type de rupture nécessite peu d'énergie et se propage donc rapidement dans le matériau. Elle résulte généralement en une brisure brutale et rapide de l'objet.
- **rupture ductile :** la rupture ductile se distingue de la rupture fragile par la présence d'une zone de déformation plastique aux points de propagation. L'énergie est dissipée et la fracture est donc moins brutale.

Il existe un lien entre les ruptures fragiles et ductiles. En effet, un objet qui présente habituellement des fractures ductiles peut néanmoins former des fractures fragiles lorsqu'il est soumis à des efforts brefs ou intenses. La rupture fragile est modélisée par la mécanique des fractures

FIG. 10.4 – Les déchirures tendent à s'entrelacer (images réelles).

élastiques ([Bro91, And95, TF88]). Il existe deux approches distinctes au problème de la rupture des matériaux élastiques. L'approche d'Inglis [Ing13], basée sur les efforts, permet d'évaluer l'intensité des contraintes aux extrémités d'une fracture. L'effort dépend directement du rapport entre la longueur et la largeur de la fracture. L'approche de Griffith [Gri20] définit la condition limite de fracture comme l'instant où l'agrandissement de la fissure a lieu sous des conditions d'équilibre, sans changement brutal de l'énergie totale du système. Ainsi, pour qu'une fissure dans une plaque soumise à une contrainte s'agrandisse, il doit y avoir assez d'énergie potentielle emmagasinée pour compenser la création des nouvelles surfaces.

Ces éléments de physique vont nous permettre de définir un modèle de matériau et de fracture bien adapté au cas des écorces, que nous pourrons ensuite enrichir avec les comportements observés dans la nature.

3 Simulation d'une plaque d'écorce

Dans cette section nous décrivons notre simulation dans un cas simple : la génération à plat d'une plaque d'écorce. Nous verrons, en section 5, comment ce modèle est étendu à l'habillage d'un arbre.

Comme nous l'avons expliqué, la simulation physique du matériau d'une écorce est hors de portée : la complexité et la méconnaissance du matériau nous en empêche. Notre approche est basée sur une étude de cas, qui nous permet d'extraire des lois de comportement des déchirures. Nous simulons ensuite les fractures sur un matériau plus simple et contrôlons leur comportement afin qu'elles obéissent aux lois observées.

Nous étudions les caractéristiques du comportement des fractures des écorces en section 3.1. Ceci nous permet également de définir les hypothèses de notre modèle de simulation. Nous expliquons le modèle en sections 3.2 et 3.3.

3.1 Etude de cas et hypothèses

Lors de l'étude des écorces, nous avons fait les observations suivantes :

– la surface d'origine (épiderme) est conservée ;

- les fractures sont largement ouvertes (déchirures) et les aspects des bordures varient selon l'espèce.
- les déchirures ont une forme similaire sur le tronc et les branches et apparaissent lors de la croissance de l'arbre ;
- les déchirures s'influencent (voir figure 10.4) ;
- des lamelles apparaissent entre déchirures proches (voir figure 10.4) ;
- les déchirures peuvent exister à différentes échelles (re–fracture à l'intérieur des déchirures déjà ouvertes) ;
- l'écorce subit la croissance mais n'a pas d'effet sur celle–ci ;
- la croissance subie par l'écorce est principalement radiale, les fractures s'allongent de haut en bas ;
- le matériau a un comportement instantané élastique (ce qui explique l'apparition des fractures), mais plastique à long terme (aucune tension résiduelle n'existe dans l'écorce) ;

A partir de ces observations, nous avons formulé les hypothèses suivantes :

- l'écorce est représentée par des bandelettes longitudinales (radiales sur le tronc) qui sont traversées orthogonalement par les déchirures ;
- nous supposons que l'épiderme est quasi–rigide ;
- nous supposons que le mode de rupture est fragile ;
- nous simulons un état quasi–statique : l'écorce subit une série d'élongations instantanées, entre lesquelles le matériau est à l'équilibre.

Notons que le comportement instantané élastique du matériau correspond à ce qui se passe dans la nature : l'écorce subit en effet chaque année une croissance relativement rapide, pendant seulement quelques mois consécutifs. Elle s'effectue donc à une échelle de temps suffisamment petite pour que le matériau soit considéré comme élastique pendant cette phase, alors qu'il est plastique à long terme (voir section 2.2).

3.2 Modèle d'écorce

Trois évènements se produisent sur les écorces présentant des déchirures et doivent être simulés sur notre matériau :

- l'apparition de nouvelles fractures ;
- la propagation de fractures déjà existantes ;
- l'ouverture de fractures (déchirures) déjà existantes.

Notre objectif est de simuler ces trois phénomènes à partir de lois physiques ou à partir des comportements observés, tout en maintenant le contrôle utilisateur. La figure 10.5 montre le synopsis de notre simulation de fractures, que nous détaillons dans les paragraphes suivants. L'algorithme de simulation est donné figure 10.8 et les résultats montrés en figure 10.11.

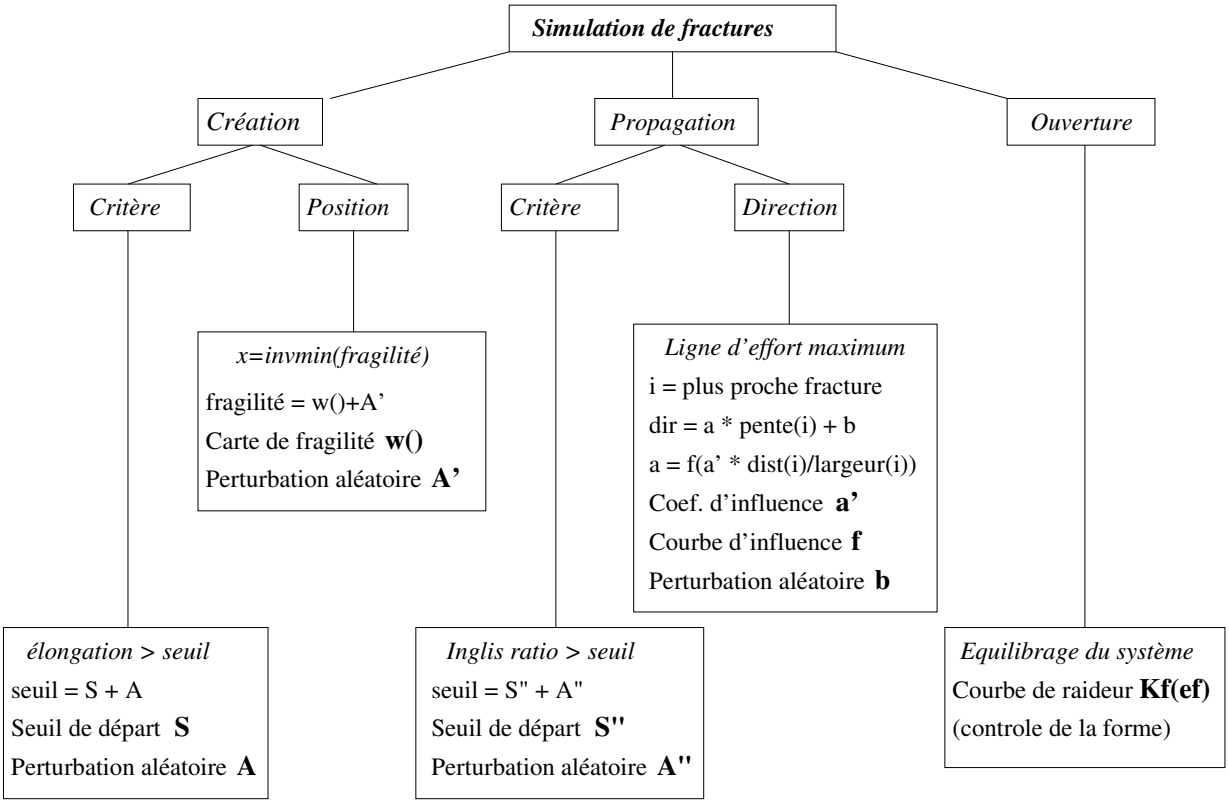


FIG. 10.5 – Principe de fonctionnement du simulateur

Les éléments écrits en gras sont contrôlables par l'utilisateur.

3.2.1 Modélisation du matériau

Comme expliqué ci–avant, nous représentons le matériau par une série de bandelettes 1D parallèles à la direction d'élongation et capables de se fracturer. Chaque bandelette est constituée d'une chaîne d'éléments qui peuvent être soit de l'épiderme (surface présente à l'origine), soit l'intérieur d'une déchirure (surface qui apparaît). Lors de la fracturation, des éléments plus souples que l'épiderme sont introduits entre les lèvres de la fracture. Ceux-ci permettent de simuler le comportement en ouverture des déchirures. Les éléments représentant l'épiderme sont quasi–rigides alors que les éléments à l'intérieur des déchirures sont souples. Les étapes de croissance successives vont ainsi élargir les déchirures représentées par les éléments souples.

Comme le relâchement de la tension est seulement partiel et local (une extension du modèle présentée section 5 permettant de représenter l'attachement au substrat), l'écorce peut se fracturer à nouveau sur une bandelette adjacente. Même si le comportement global des éléments est plastique, c'est leur comportement élastique instantané qui nous intéresse pour la fracturation, c'est pourquoi nous les appelons *éléments élastiques*.

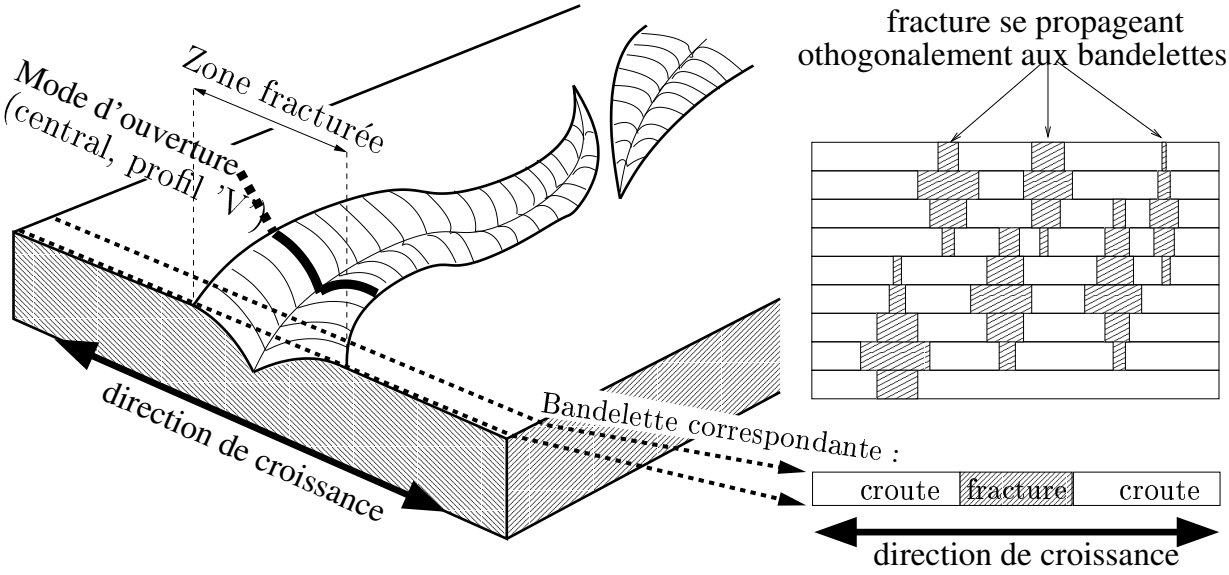


FIG. 10.6 – Caractéristique d'un déchirure et bandelettes du modèle

Raideur de l'écorce : Selon la loi de Hooke, un élément élastique linéaire 1D de raideur K, subissant une élongation e produit un effort $F = Ke$. Néanmoins il faut bien distinguer *raideur du matériau* et *raideur de l'élément*. En effet, un élément de longueur à vide l_0 constitué d'une matière de raideur R présente une raideur propre $K = \frac{R}{l_0}$. Ceci est très important car lorsqu'un élément de l'épiderme se rompt, il doit être scindé en deux éléments équivalents. Leur raideur doit donc être ajustée en conséquence (voir figure 10.7).

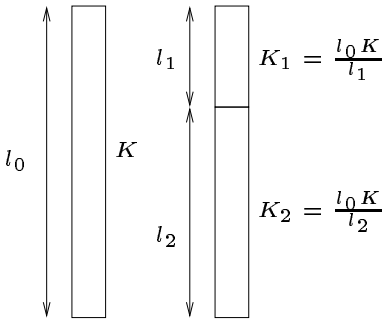


FIG. 10.7 – Raideur des éléments de l'épiderme de l'écorce.

Raideur des déchirures : Le comportement précédent est vrai pour les éléments de l'épiderme de l'écorce, mais les éléments représentant les déchirures ne sont pas un matériau réel et n'obéissent pas à cette loi. Leur raideur est liée à leur attachement au substrat et à la forme des fractures (voir section 3.2.4).

Nous supposons que l'évolution de l'écorce est quasi–statique, donc chaque bandelette doit être à l'équilibre après chaque élongation. A chaque pas de simulation, le système d'équation suivant doit être résolu pour chaque bandelette :

$$\left\{ \forall i \in [0;N], K_i e_i = K_{i+1} e_{i+1}, \sum_i e_i = L^{tot} - l_0^{tot} \right\}$$

K_i est la raideur de l'élément i, e_i son élongation. L^{tot} est la longueur totale de la bandelette, l_0^{tot} la somme des longueurs à vide des éléments. Le système étant linéaire, une méthode de résolution classique peut être utilisée. En pratique, nous utilisons la méthode du gradient bi–conjugué. L'algorithme de la simulation est donné figure 10.8.

```
pour chaque bandelette
  appliquer l'élongation
  résoudre l'équilibre
  tant qu'un élément vérifie le critère de fracture
    choisir le lieu de fracture
    fracturer l'élément
    résoudre l'équilibre
  fin tant que
  propager les fractures
fin pour
```

FIG. 10.8 – Algorithme de création et propagation de fractures.

3.2.2 Création d'une fracture

Le critère de fracture est basé sur l'élongation relative des éléments élastiques (rapport entre la longueur sous contrainte de l'élément et sa longueur à vide). L'élongation relative reflète directement l'énergie dépensée par l'élément, ce qui rejoint l'approche énergétique de Griffith.

Le choix du seuil S d'élongation relative produisant une fracture doit être effectué dans des bornes raisonnables. En effet, à cause de la raideur de la matière apparaissant dans la déchirure, il est possible de se trouver dans un schéma itératif infini : les fractures qui apparaissent n'absorbent pas assez de déformation et de nouvelles fractures continuent d'être insérées. L'étude ci-après à pour but de déterminer dans quelles bornes doit être choisi le seuil de fracture.

La figure 10.9 présente un élément élastique de raideur K_s, longueur L, longueur à vide l_0 avant et après fracture. Soit S le seuil de fracture, la bande se fracture lorsque l'élongation relative de l'élément dépasse ce seuil, et donc $\frac{L}{l_0} > S$. L'insertion d'un élément de fracture fait changer l'élongation de l'élément d'épiderme de e à e'. La différence $e_f = e - e'$ correspond à la largeur de la fracture. Nous ne voulons pas que l'élément se fracture à nouveau, il faut donc $\frac{l_0+e'}{l_0} < S$. Soit K_f la raideur de l'élément de fracture. Puisque le système est à l'équilibre, nous avons $K_s e' = K_f e_f$ et donc $e' = e\frac{K_f}{K_s+K_f}$. Il nous faut donc choisir S tel que $S > 1 + \frac{e}{l_0}\frac{K_f}{K_s+K_f}$.

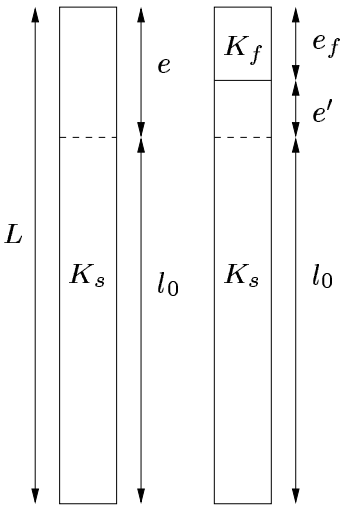


FIG. 10.9 – Evaluation du critère de fracture.

Cependant, si l'on choisit un seuil de fracture trop important,
aucune fracture n'apparaîtra. Une fracture apparaît forcément lorsque $S < \mathcal{G}$, où $\mathcal{G}$ est le pas d'élongation relative de la circonférence de l'arbre.

Dans la nature, les fractures n'apparaissent pas simultanément, et ceci est d'autant plus vrai que les matériaux sont non homogènes. Nous modélisons cette non homogénéité en bruitant le seuil de fracture. En pratique, nous choisissons un seuil égal au pas d'élongation $\mathcal{G}$ et faisons uniquement varier le paramètre de bruit. Nous obtenons ainsi un bon indicateur de la densité de fracture, contrôlable par l'utilisateur.

Lieu de fracture : L'écorce se rompt en son endroit le plus faible. Nous modélisons l'hétérogénéité de la résistance de l'écorce à l'aide d'une carte de fragilité et d'un bruit aléatoire. Le bruit permet de représenter l'inhomogénéité alors que la carte de fragilité permet à l'utilisateur de choisir en quels endroits l'écorce va probablement se rompre. Notre modèle peut également appliquer des fractures récursivement pour simuler le comportement de certaines espèces où l'intérieur des déchirures devient fragile, tout comme l'épiderme. La carte de fragilité est utilisée, de manière automatique, pour que les nouvelles fractures tendent à apparaître dans les anciennes déchirures.

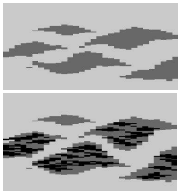


FIG. 10.10 – Re–fracture de l'écorce.

3.2.3 Propagation des fractures

La propagation des fractures est effectuée en décidant si un élément de déchirure d'une bandelette doit ou non fracturer la bandelette voisine. Le critère de propagation est basé sur les constatations d'Inglis [Ing13] :

- L'effort en bout de fracture est proportionnel au rapport longueur/largeur des déchirures (*Inglis ratio*) ;
- Une fracture suit la ligne d'effort maximal ;
- Les courbes d'iso–effort autour d'une fracture sont parallèles aux bordures (ainsi les fractures tendent à s'entrelacer au lieu de se rejoindre) ;
- Les imperfections du matériau provoquent des changements dans l'angle de propagation des fractures. Ceux–ci peuvent être modélisés par une déviation angulaire aléatoire [And95, Rah].

Critère de propagation : Nous comparons le rapport longueur/largeur des fractures à un seuil. Les liens entre les éléments qui se propagent d'une bandelette à l'autre sont conservés pour former une structure arborescente que nous appelons *squelette des déchirures*. Cette structure permet d'identifier les éléments de chaque déchirure.

Direction de propagation : Nous recherchons tout d'abord les déchirures dans le voisinage d'un élément devant se propager. Ceci permet de déterminer la direction locale d'effort maximum. Nous supposons que l'influence d'une déchirure est proportionnelle à sa largeur – ceci correspond au terme $f(a'\frac{dist}{largeur})$ de la figure 10.5, où f est une fonction de poids. Dans notre

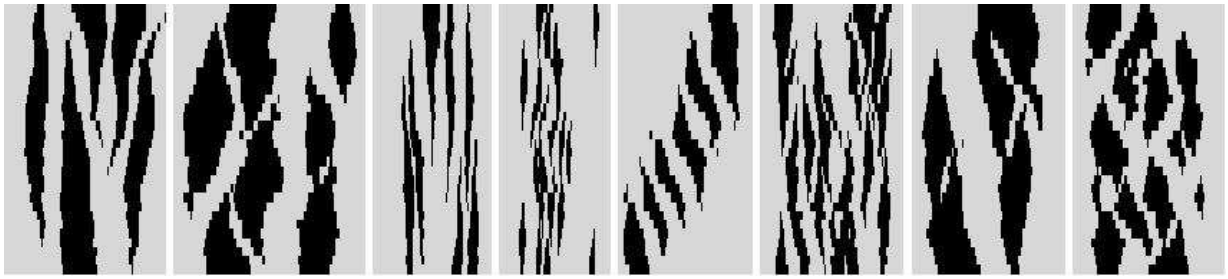


FIG. 10.11 – Résultat de simulation de déchirures avec différents paramètres.

implémentation, nous utilisons la largeur moyenne d'une déchirure, calculée sur l'ensemble de ses éléments. Nous utilisons une fonction de seuil pour f.

Ceci nous donne une direction de base (allant de la verticale à la pente de la plus proche déchirure). Nous bruitons cette direction pour simuler les défauts du matériau. L'amplitude du bruit est contrôlable par l'utilisateur, ce qui permet d'influencer l'aspect anguleux des déchirures.

3.2.4 Elargissement des déchirures

Le comportement des déchirures en ouverture dépend directement de la raideur des éléments qui les composent. La loi de variation de la raideur des éléments permet donc d'influencer la forme des déchirures. Dans la nature, la résistance à l'ouverture provient à la fois de l'attachement au substrat, et de la rigidité du matériau autour des extrémités de la déchirure. Comme ceci a un fort impact sur la forme des déchirures, nous représentons ces phénomènes par une fonction $K_f(l_0)$ qui peut être éditée par l'utilisateur.

3.3 Habillage de l'intérieur des déchirures

Une fois la position et la largeur des déchirures calculées par le modèle, il est nécessaire d'habiller le résultat afin d'obtenir une image visuellement intéressante. Il s'agit d'appliquer une texture sur l'épiderme et à l'intérieur des déchirures, comme montré figure 10.12. Les différentes étapes à réaliser sont illustrées figure 10.13. Nous utilisons des textures fournies par l'utilisateur pour l'épiderme et l'intérieur des textures. Bien entendu, ces textures ne doivent pas contenir les déchirures puisqu'elles sont simulées par notre algorithme. Cependant, elles peuvent contenir tous les détails fins, souhaités pour enrichir l'apparence finale de l'écorce.

L'habillage de la simulation nécessite les étapes suivantes :

- reconstruction des silhouettes des déchirures à partir du squelette de fracture ;
- triangulation de l'intérieur de silhouette et de l'épiderme ;
- application des textures ;
- éventuellement, créer une information de relief (utile pour le bump–mapping par exemple).

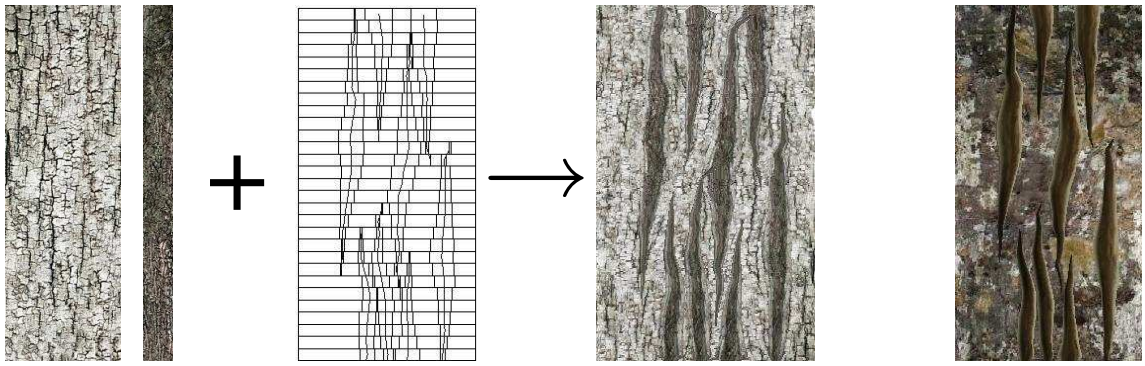


FIG. 10.12 – Habillage de la simulation.

De gauche à droite : *textures d'épiderme et de déchirure fournies par l'utilisateur ; les données brutes produites par la simulation ; l'écorce finale.* A droite : *Même simulation habillée avec différentes textures.*

Notons qu'il est possible de générer de la géométrie à partir des données de simulation. Cependant, on préférera souvent dessiner le résultat de la simulation texturée dans une image 2D et l'utiliser comme texture sur l'arbre.

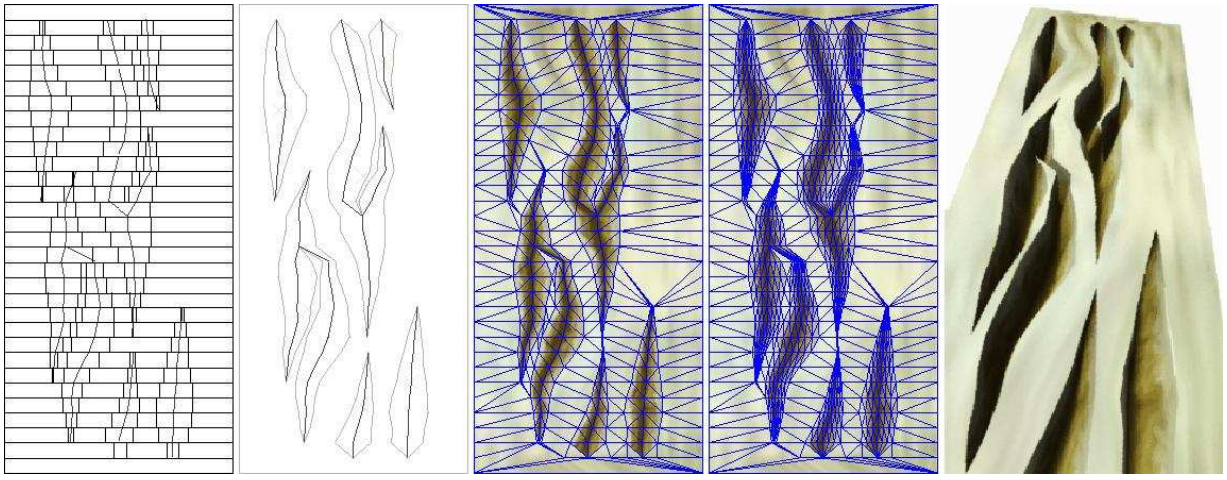


FIG. 10.13 – Reconstruction des silhouettes des déchirures.

De gauche à droite : *données de simulation ; reconstruction des silhouettes ; triangulation ; triangulation avec relief (optionnel) ; géométrie équivalente (par exemple utilisée pour créer une texture de bump).*

Construction des silhouettes : Il est facile de produire un contour à partir du squelette des fractures (graphe de segments avec largeurs). Le seul point délicat est de gérer correctement les fractures se rejoignant.

Triangulation de la surface de l'écorce : Nous souhaitons ici créer un maillage triangulaire de la surface de l'écorce, qui suit les silhouettes des fractures. Dans notre implémentation, nous

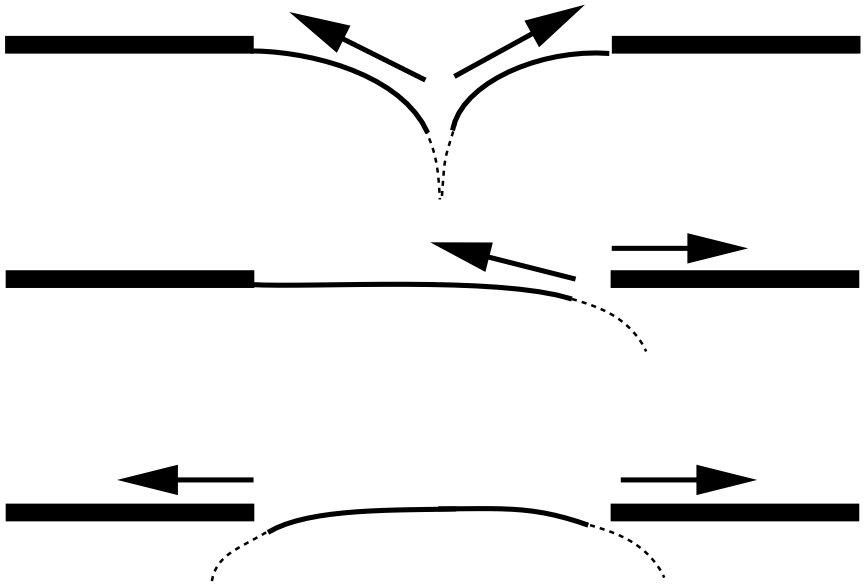


FIG. 10.14 – Différents modes d'ouverture des déchirures.

utilisons à cette fin la libraire *Triangle* [She96] qui construit efficacement une triangulation de Delaunay avec des arêtes contraintes.

Application des textures : Nous devons calculer les coordonnées de texture associées à l'épiderme et aux déchirures durant la croissance. Chaque élément de déchirure possède une coordonnée qui correspond au lieu où la fracture est apparue sur l'épiderme d'origine. Cette coordonnée nous permet d'appliquer la texture de l'épiderme sur la surface de l'écorce. L'intérieur d'une déchirure est habillé en fonction de son mode d'ouverture. Ainsi, la matière peut rester collée à une bordure ou bien fixe autour de la ligne de fracture, tandis que les lèvres découvrent progressivement le substrat (voir figure 10.14).

Information de relief : Au–delà des informations de relief qui peuvent être présentes dans les textures fournies par l'utilisateur, nous devons pouvoir ajouter une information de relief créée par les déchirures de la surface. La mise en relief est effectuée en ajoutant des courbes de niveau entre les bords et le centre des déchirures avant l'étape de triangulation. Nous utilisons une fonction de profil, contrôlée par l'utilisateur, qui sculpte le relief de la déchirure entre le bord et le centre. Les courbes de niveau sont paramétrées entre $[0,1]$, 0 correspondant au bord et 1 au centre de la déchirure. La hauteur de chaque courbe de niveau est ajustée par les courbes de profil, dont quelques exemples sont présentés figure 10.15. Le relief ainsi créé permet, par exemple, de générer des textures pour le *bump–mapping* [Bli78], mais peut aussi être utilisé pour modifier la géométrie de l'arbre.

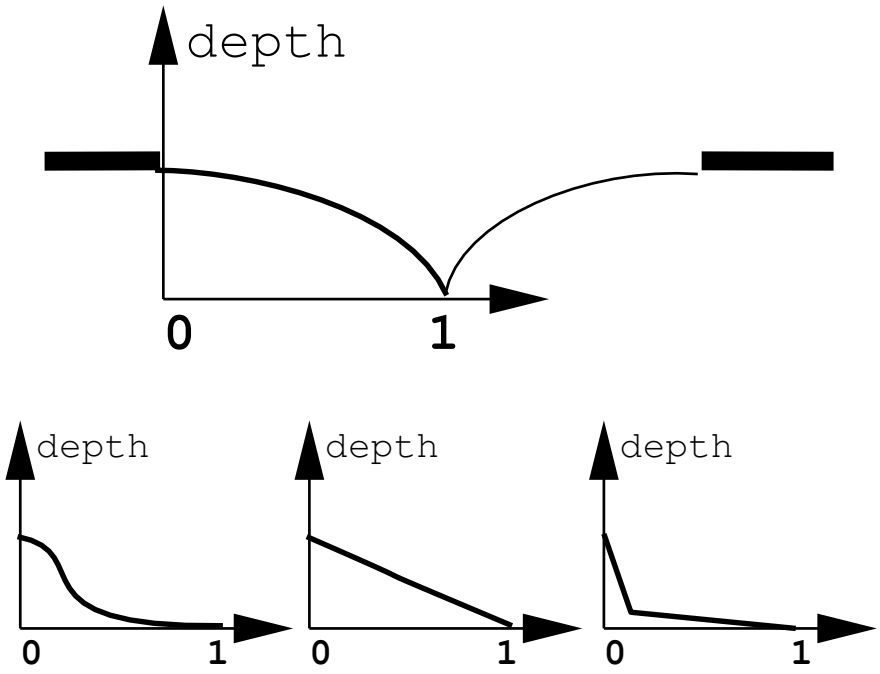


FIG. 10.15 – Courbes de profil pour l'ajout de relief.

4 Résultats de simulation d'une plaque d'écorce

Tous les résultats présentés, à part la figure 10.10 et la cinquième image de la figure 10.11 ont été produits sans l'utilisation d'une carte de fragilité et sans intervention de l'utilisateur durant la simulation. Après le choix initial des paramètres, la simulation s'est déroulée automatiquement.

Bien que notre modèle permette d'ajuster la courbe de raideur $K_f(l_0)$ des éléments de déchirure, nous utilisons en pratique une valeur K_f constante pour tous les éléments de fracture. La figure 10.10 a été produite en utilisant la simulation précédente comme carte de fragilité. La cinquième image de la figure 10.11 a été créée en laissant l'utilisateur interactivement couper l'épiderme pour introduire des fractures durant la simulation.

Des résultats de simulation sont montrés figure 10.11 avec différents paramètres de raideur et de seuil de fracture. La figure 10.18 montre le résultat d'une écorce avec rupture fragile (le seuil de fracture est proche de 1). La figure 10.16 compare une écorce simulée à une écorce réelle. Le point important est que la forme et la distribution des fractures correspondent à l'écorce originale.

La figure 10.17 montre différentes écorces d'une même espèce produites avec différentes initialisations du générateur de nombres pseudo–aléatoires. Ceci montre qu'il est possible d'habiller avec une écorce tous les arbres d'une forêt sans que deux ne se ressemblent. Il est également possible de générer des textures cycliques pour les pavages périodiques (voir chapitre 2, section 3.2) en considérant les bandelettes et éléments des extrémités comme étant voisins.

FIG. 10.16 – Comparaison entre une écorce simulée (à droite) et une écorce réelle (à gauche).

La figure 10.19 présente une animation de la propagation d'une déchirure durant la croissance d'un arbre.

Les simulations présentées ici sont interactives (quelques étapes par seconde). En pratique, le temps de calcul varie approximativement linéairement avec le nombre total d'éléments simulés (ce nombre est proportionnel au nombre de bandelettes multiplié par le nombre moyen de fractures par bandelettes). Le temps de calcul pour l'étape d'habillage est linéaire pour la construction des silhouettes et de $O(n\log(n))$ pour la triangulation de Delaunay (n étant le nombre de points). Calculer une écorce constituée de centaines d'éléments prend moins d'une seconde (simulation complète). Une fois l'écorce calculée et dessinée sous forme de texture globale, le rendu de l'arbre peut être effectué avec le placage de texture standard.

Notre modèle, tel que nous l'avons présenté dans cette section, peut générer des écorces à plat. Cependant, pour habiller une géométrie d'arbre, cette méthode souffrirait des mêmes problèmes que les approches basées sur le placage de texture [Blo85, MDG00] (distorsions et discontinuités). La section suivante explique comment étendre notre modèle à un arbre entier.

5 Habillage d'un arbre avec notre modèle d'écorce

Nous avons présenté, en section 3, notre modèle de simulation, dans le cas d'une plaque d'écorce. Les éléments de base sont : les bandelettes, la simulation des déchirures et la génération de l'apparence finale à l'aide de textures. Afin de pouvoir étendre ce modèle à l'habillage d'un arbre complet, nous devons tenir compte des éléments suivants :

- les bandelettes deviennent cycliques (on peut les considérer comme des élastiques positionnés autour du tronc) ;
- il faut attacher l'écorce au substrat (pour éliminer le degré de liberté en rotation des bandelettes autour du tronc) ;

FIG. 10.17 – Différentes écorces de la même espèce

FIG. 10.18 – Fractures fragiles.

- les bandelettes ont différentes largeurs et longueurs (le rayon et la courbure varient le long du tronc) ;
- il faut gérer les embranchements entre tronc et branches.

Gérer des bandelettes déformées est nécessaire car le tronc et les branches ont des formes courbes. Dans notre implémentation, nous représentons les branches de l'arbre par des cylindres généralisés. Leur axe est une courbe et leur rayon varie sur la longueur et selon l'angle (le profil d'une section n'est pas un simple cercle). De plus, nous souhaitons permettre une croissance non homogène de l'arbre. La croissance est définie par un *champ de dilatation* qui varie le long de l'axe et de la circonférence. Enfin, l'ajout d'une force d'attachement entre l'écorce (les bandelettes) et le substrat (le bois de l'intérieur du tronc) est nécessaire pour supprimer le degré de liberté introduit par les bandelettes cycliques : il faut fixer leur position radiale autour du tronc. Cette force d'attachement permet également un contrôle local sur le lieu d'apparition des frac-

FIG. 10.19 – Propagation de fracture durant la croissance de l'arbre

tures. Si une zone croît plus rapidement que son voisinage, les fractures vont plus probablement apparaître à cet endroit plutôt que tout le tour du tronc.

Nous décrivons l'attachement entre les bandelettes et le tronc en section 5.1. Nous expliquons comment gérer la croissance de l'arbre en section 5.2 et nous présentons comment modifier la simulation des fractures en section 5.3. L'habillage final (application de textures sur l'épiderme et les fractures) est décrit en section 5.4. Les résultats sont montrés en section 6.

5.1 Attachement de l'écorce au substrat

Les éléments représentant l'épiderme sur les bandelettes sont quasi–rigides. Ainsi, lorsque l'arbre croît, la déformation est absorbée par l'écorce au travers des fractures. L'écorce étant attachée au tronc, il ne suffit pas de simplement relâcher la tension sur tous les éléments de la bandelette : chaque élément d'épiderme est 'collé' sur un morceau du substrat qui a grandi. La tension n'est donc pas répartie uniformément sur tous les éléments. Cette force d'attachement est représentée dans notre modèle, par des ressorts de longueur nulle au repos. Ceux–ci attachent les extrémités des éléments à leur position d'origine sur le substrat (voir figure 10.20). Les ressorts modélisent une force d'attachement instantanée : ils sont réinitialisés au début de chaque pas de temps. Rappelons en effet que le matériau est plastique à long terme et absorbe les déformations. Ils produisent une force d'attachement $K_a e_a$ qui s'oppose au déplacement produit par la croissance e_a du substrat. K_a caractérise la force d'attachement et contrôle la localité des fractures (en particulier lorsque la croissance n'est pas homogène).
Le système à résoudre pour chaque bandelette devient :

$$\left\{ \ \forall i, \ -K_i e_i + K_{i+1} e_{i+1} + K_a e_a^i = 0 \ \right\}$$

que nous pouvons ré–écrire comme :

$$-K_i(x_i - x_{i-1} - l_0^i) + K_{i+1}(x_{i+1} - x_i - l_0^{i+1}) + K_a(x_i - a_i) = 0$$

où x_{i-1} et x_i sont les extrémités (que l'on cherche à déterminer) en coordonnés curvilignes le long de la bandelette du $i^{\text{ème}}$ élément, et a_i est la position curviligne où le point x_i est attaché au

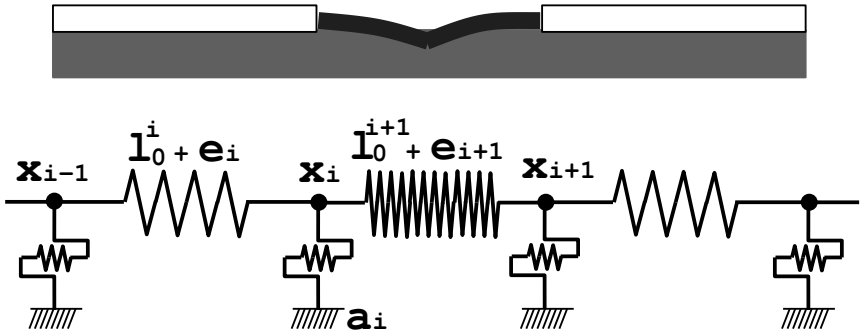


FIG. 10.20 – Modèle d'attachement de l'écorce au substrat.

substrat (voir section 5.2 pour le calcul des a_i). Une fois le pas de simulation terminé, les valeurs de a_i sont mises à jour avec les nouvelles valeurs de x_i pour absorber les déformations.

5.2 Croissance de l'arbre

La géométrie de l'arbre est supposée fournie par l'utilisateur, sous la forme d'un ensemble de cylindres généralisés (cette géométrie peut avoir été générée avec des outils spécialisés [Bio, Xfr]). Il est possible de calculer une paramétrisation curviligne le long des cylindres généralisés. Nous supposons cette paramétrisation disponible pour les cylindres du tronc et des branches. Il faut maintenant faire grandir l'arbre pour que la simulation de l'écorce puisse se dérouler. Au cours de la simulation, il faut pouvoir mettre en correspondance les paramétrisations d'un même cylindre à différentes étapes de simulation. Cette mise en correspondance détermine le champ de déformation de l'écorce, et permet de suivre où sont transférés les points d'attachement des éléments sur le substrat, au pas de simulation suivant (coordonnées a_i de l'équation section 5.1). Il existe différentes possibilités :

- **La croissance de l'arbre est gérée par la simulation.** La nouvelle forme de l'arbre est obtenue à partir de la précédente au travers d'une loi de croissance contrôlée par l'utilisateur. Ceci permet de mettre en correspondance les paramétrisations des cylindres à chaque étape (l'ancienne et la nouvelle position des points de la surface sont connues).
- **La géométrie de l'arbre est fournie à diverses étapes de croissance.** Chaque étape peut être paramétrée indépendamment avec les coordonnées curvilignes. Nous pouvons alors mettre les paramétrisations en correspondance en déterminant où à été transféré, sur la nouvelle surface, un point de l'ancienne surface. A cette fin, nous projetons les points de l'ancienne surface vers la nouvelle dans la direction de la normale, ou dans la direction radiale. En pratique, nous utilisons la direction radiale car les calculs sont plus rapides. Cependant, la direction de la normale correspond mieux au phénomène de croissance observé dans la nature.
- **L'utilisateur ne souhaite pas animer ou modifier la géométrie de l'arbre.** Dans ce cas, l'utilisateur ne veut pas que l'arbre fourni soit modifié. Il souhaite uniquement que le

système génère une texture. Ce mode est très important car il est probable que les artistes ne veuillent pas que la simulation modifie une géométrie soigneusement créée. Dans ce cas, nous couvrons l'arbre d'un épiderme non fracturé, et au lieu de faire croître l'arbre, nous simulons le rétrécissement de l'épiderme. La loi de rétrécissement utilisée est l'inverse de la loi de croissance utilisée précédemment.

Il nous est donc possible de faire croître l'arbre (ou rétrécir l'écorce) et de mettre en correspondance les points de la surface durant la croissance. Ceci permet de simuler les bandelettes indépendamment, comme décrit précédemment, tout en tenant compte de la croissance globale du tronc. Nous expliquons dans la section suivante comment propager les fractures le long du tronc et des branches.

5.3 Simulation des fractures le long du tronc et des branches

Deux aspects de la simulation de fractures sont affectés par le fait de construire les bandelettes sur des cylindres généralisés :

- la déformation des bandelettes change les propriétés mécaniques des éléments,
- la propagation des fractures doit tenir compte des variations de diamètre et des embranchements.

5.3.1 Adapter les propriétés mécaniques des bandelettes

Comme expliqué en section 5, les bandelettes correspondent à des tranches de tronc ou de branches, qui sont eux représentés par des cylindres généralisés. La largeur et la longueur des bandelettes peut donc varier : la largeur est réduite dans les zones concaves lorsque le tronc est courbé ; la longueur des bandelettes varie avec le rayon du tronc.

La variation de longueur est prise en compte par la simulation, en utilisant les coordonnées curvilignes des éléments le long de la bandelette. L'équilibre du système formé par les éléments de la bandelette est ainsi obtenu pour les vraies longueurs, autour du tronc. La variation de largeur sur une même bandelette est gérée en adaptant le coefficient de raideur K_i des éléments. Nous étendons l'expression utilisée en section 3.2.1 : un élément de longueur au repos l_0 et de largeur w constitué d'un matériau de raideur R aura une raideur de $K = \frac{Rw}{l_0}$. Ceci n'est bien entendu qu'une approximation, la variation de largeur étant uniquement prise en compte sur chaque élément (or, la largeur peut varier *sur* un élément). Cependant, la courbure du tronc n'est généralement pas suffisamment importante pour introduire un biais visible dans la simulation.

5.3.2 Adapter la propagation des fractures

Nous considérons que chaque bandelette est indépendante : les éléments d'une bandelette sont simulés dans son espace curviligne 1D. Pour passer d'une bandelette à une autre, et ainsi propager les fractures, nous utilisons cependant les coordonnées 3D des points : la position de l'élément à l'extrémité d'une fracture est convertie de l'espace curviligne, sur la bandelette, à l'espace 3D. Ce point est alors reconverti vers l'espace curviligne de la bandelette voisine pour obtenir le lieu où propager la fracture. (voir figure 10.22).

L'avantage de considérer la propagation dans l'espace 3D est de pouvoir ainsi définir une propagation universelle. Dans les zones d'embranchement, cela permet notamment de propager la fracture d'une bandelette du tronc à une bandelette d'une branche : nous testons si le point de propagation touche, dans l'espace 3D, une bandelette d'une branche. Dans ce cas, la fracture est propagée sur la branche plutôt que sur la bandelette voisine sur le tronc. On obtient ainsi une propagation continue des fractures (voir figure 10.22, à droite).

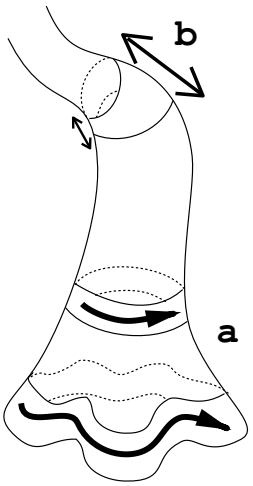


FIG. 10.21 – Les bandelettes sont déformées par les courbes du tronc.

*La longueur (*a*) et largeur (*b*) des bandelettes varient.*

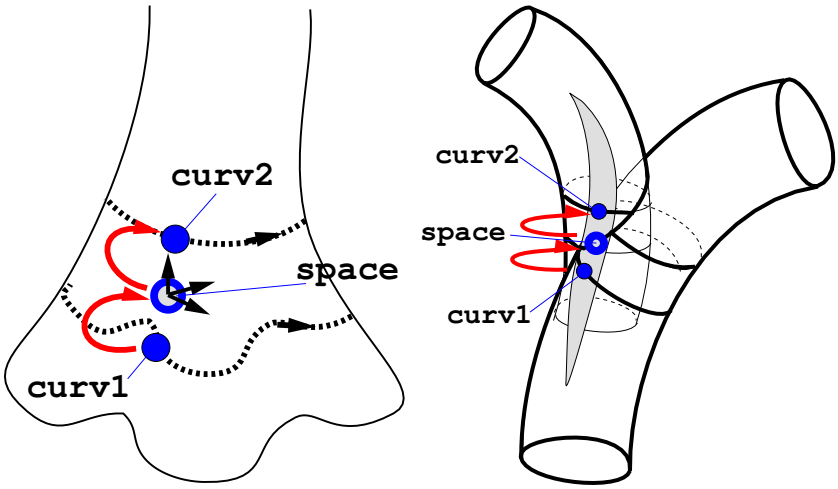


FIG. 10.22 – Propagation de fractures sur l'arbre

Propagation entre les bandelettes d'une même branche et entre les bandelettes du tronc et d'une branche.

5.4 Habillage de la simulation

L'habillage des données de simulation (structure des fractures et déchirures) est réalisé de manière similaire à celle décrite en section 3.3.

Nous supposons ici qu'une paramétrisation UV est fournie par l'utilisateur, pour chaque cylindre généralisé. Cette paramétrisation est celle utilisée pour le placage de texture durant le rendu. Si une telle paramétrisation n'est pas fournie, une nouvelle peut être créée à partir de la paramétrisation curviligne. Le but est de créer une texture d'écorce pour chaque cylindre. La texture d'écorce doit être dessinée à partir des données de simulation de manière à ce que le résultat soit non distordu et continu lorsque la paramétrisation UV est utilisée pour plaquer la texture sur le cylindre.

Les textures sont créées séparément pour chaque cylindre (une texture par cylindre). Nous commençons par convertir les coordonnées des éléments des bandelettes en coordonnées dans l'espace UV défini par l'utilisateur. Nous pouvons alors reconstruire l'apparence de l'écorce comme dans le cas planaire. Grâce à la conversion des coordonnées entre espace curviligne (bandelettes) et espace UV, les déchirures seront peintes dans la texture avec une distorsion qui compense l'éventuelle distorsion introduite par la paramétrisation UV lors du placage de la texture. Une fois les textures appliquées sur l'épiderme et l'intérieur des déchirures, l'image résultat est la texture d'écorce pour le cylindre généralisé.

Note sur la continuité
Comme précédemment (voir section 3.3), l'utilisateur fournit une texture d'épiderme et d'intérieur de déchirure. La texture d'écorce de chaque cylindre est créée séparément. Dans les embranchements, les lieux de propagation de fractures vont néanmoins se correspondre par construction, et l'on verra des fractures continues entre le tronc et les branches. Cependant, il n'en est pas de même pour la texture de l'épiderme : les paramétrisations UV du tronc et des branches n'ont aucune raison de se correspondre. L'aspect de l'épiderme est donc discontinu aux embranchements. Cependant, la texture d'épiderme est d'aspect relativement homogène. Ainsi les discontinuités de cette texture entre tronc et branches seront peu visibles. Notre hypothèse est que les éléments visuellement dominants sont les fractures : celles–ci *doivent* être continues. La discontinuité de la texture de l'épiderme, qui contient des détails fins, est masquée par la continuité des fractures.

6 Résultat de l'habillage d'un arbre par notre modèle d'écorce

Notre objectif était ici de pouvoir générer des écorces fracturées qui se comportent de manière réaliste lors des changements de rayon du tronc et des embranchements. La figure 10.23 montre différentes espèces d'écorce appliquées sur des troncs de formes variées. On peut constater que la densité de fracture varie pour maintenir un aspect constant. La figure 10.24 montre des embranchements. Les fractures se propagent entre le tronc et les branches. Comme illustré figure 10.25, notre modèle gère raisonnablement bien les cas d'embranchements complexes.

Il est cependant possible de créer des cas d'intersection mal conditionnés, où la nature elle–même ne produit pas de fractures continues (par exemple sur l'angle aigu d'un embranchement). Notons que nos fractures continues seraient visuellement plus plaisantes si la géométrie support était, elle aussi, continue à l'embranchement.

Pour toutes ces images, nous générons une texture qui contient couleur et perturbation de normales. Nous utilisons l'approche de rétrécissement de l'épiderme pour la simulation. Les textures d'épiderme et d'intérieur de fracture sont les mêmes que précédemment. Pour la figure 10.24 nous avons utilisé 133 bandelettes. Le temps de calcul est de 49 s sur un PentiumIII 900Mhz pour simuler la structure des fractures et 11 s pour générer les deux textures d'écorces (tronc et branche). Pour la figure 10.25 nous avons utilisé 225 bandelettes. Le temps de calcul est de 341 s pour simuler structure des fractures et 37 s pour générer les 5 textures d'écorce.

7 Conclusion

Nous avons présenté un modèle d'habillage qui permet de générer des écorces réalistes pour les arbres. Contrairement aux modèles des chapitres précédents, celui–ci génère des textures pour le placage de texture et n'est pas destiné à être utilisé pour l'affichage final. Il suit cependant nos grands principes que sont la non modification de la géométrie initiale (ici grâce à l'approche par rétrécissement de l'épiderme), un fort contrôle de l'artiste sur le résultat (carte de fragilité, insertion de fractures durant la simulation, forme des déchirures) et la variété d'apparences qui peuvent être facilement créées grâce à l'approche structurelle (la simulation calcule lieu et formes des déchirures avant de leur appliquer un habillage).

Du point de vue des écorces générées, notre modèle permet de produire des écorces réalistes et avec un aspect homogène malgré les variations de rayon des branches. Le modèle permet de gérer correctement la plupart des embranchements. Dans certaines situations difficiles, des discontinuités peuvent apparaître, mais ces situations correspondent également à l'apparition de discontinuités sur les arbres réels. Il faudrait cependant modéliser plus fidèlement ces zones qui ont des aspects rugueux et boursouflés sur les vrais arbres.

Les apparences produites permettent des plans rapprochés sur les écorces sans briser le réalisme, ce qui correspond à notre objectif initial. Il reste bien sûr de nombreux points à explorer, en particulier dans les zones d'embranchement, mais également du point de vue des différentes échelles de fractures. Certaines écorces génèrent notamment des aspects tressés, qui semblent résulter de la rupture successive des différentes couches composant l'écorce.

Ce type de modèle permet un gain de temps précieux aux artistes, en prenant en charge tout ou partie de la création de l'apparence de surface (l'artiste peut, s'il le souhaite, retoucher les textures produites). Néanmoins, un des problèmes de ce type d'approche est sa spécificité. Elle est ici justifiée par le fait que les arbres sont des objets très courants et difficiles à habiller. Cependant, la conception d'algorithmes capables de générer une très grande variété d'apparences réalistes sur des objets aux formes quelconques est un problème toujours ouvert, qu'il convient de continuer d'explorer (voir la discussion au chapitre 11, section 2.2).

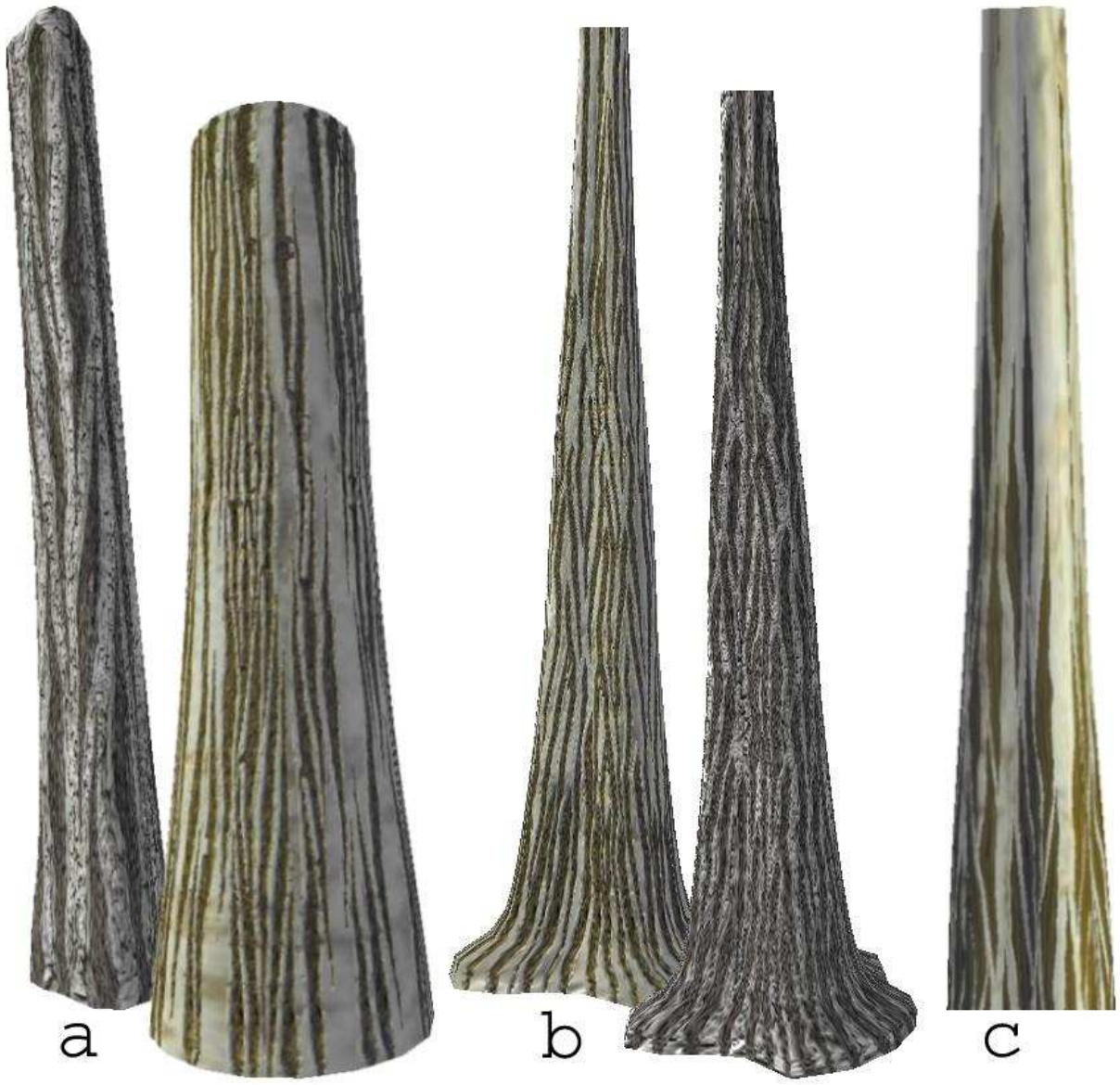


FIG. 10.23 – Ecorce simulée sur des troncs de forme complexe.
a) *Différentes formes et densités de fractures.* b) *Densité de fracture constante le long du tronc (malgré la variation de rayon et la section irrégulière).* c) *Longueur (radiale) d'épiderme constante.*

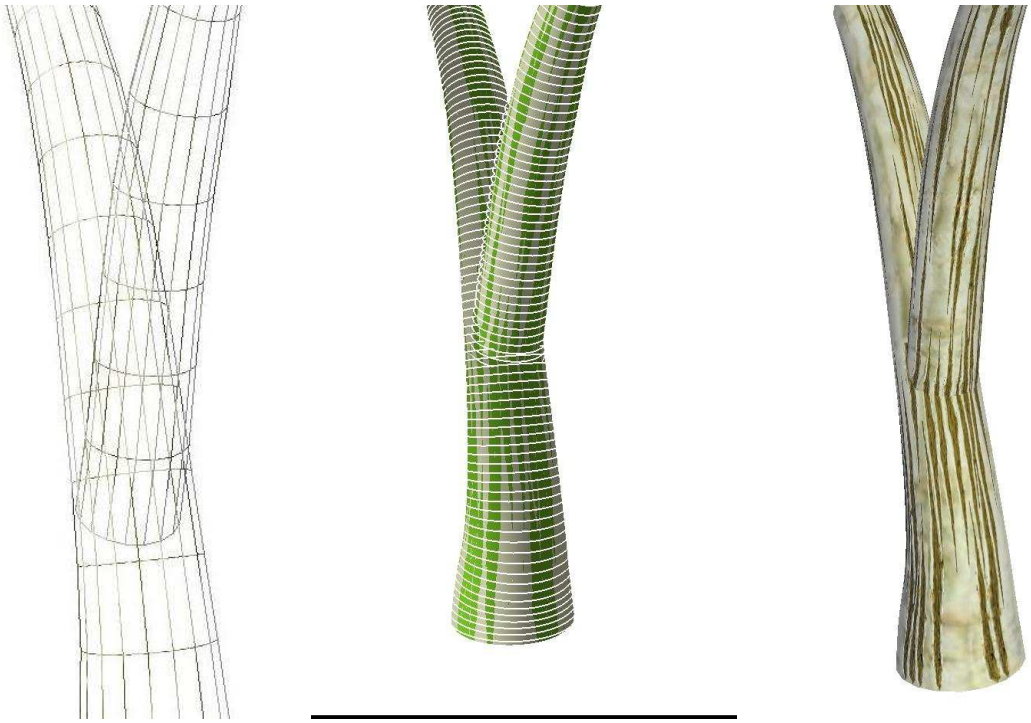


FIG. 10.24 – Résultat avec embranchement.
Les fractures se propagent d'un cylindre à l'autre. Les textures de chaque cylindre se correspondent ainsi et montrent des fractures continues. A gauche : *Le maillage du modèle d'arbre, correspondant à un ensemble de cylindres généralisés.* Au milieu : *Les bandelettes et éléments élastiques utilisés.* A droite : *Ecorce générée.*

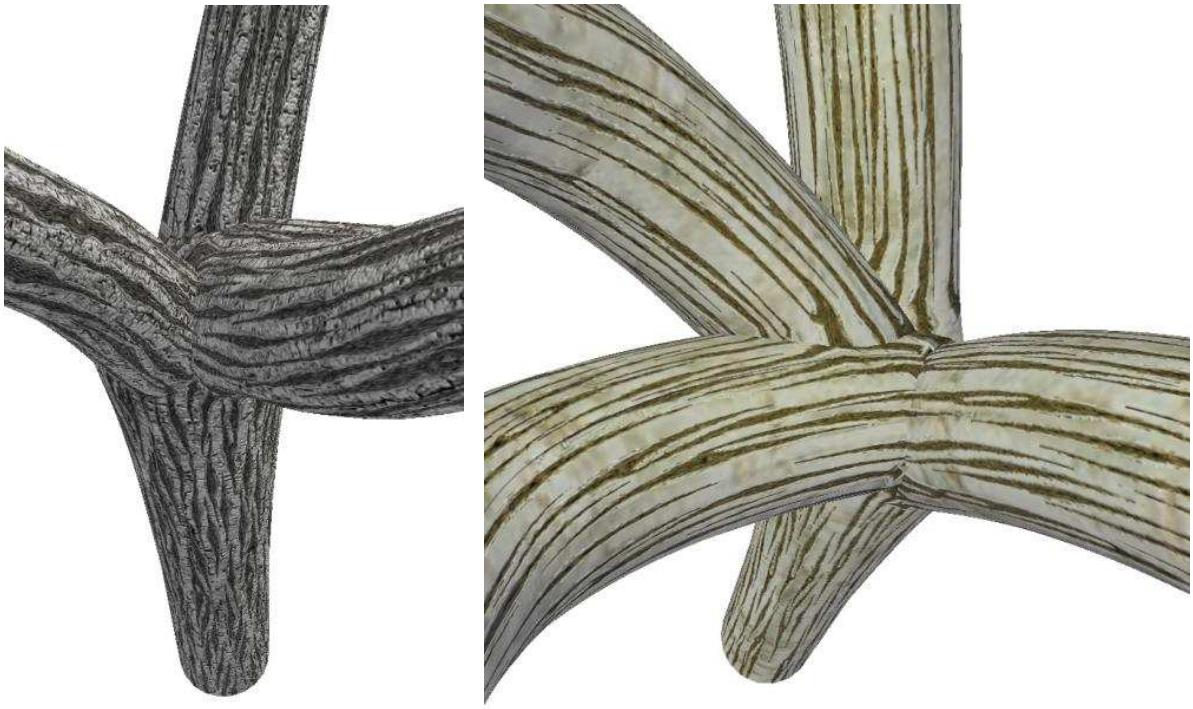


FIG. 10.25 – Embranchement complexe.

La propagation aux embranchements fonctionne relativement bien, même en présence d'angles aigus et de multiples intersections (notons qu'une telle continuité n'existe généralement pas dans la nature).

Sixième partie

Conclusions et perspectives

chapitre 11

Conclusions et perspectives

1 Bilan

Tout au long de cette thèse, nous avons proposé diverses solutions pour répondre au besoin croissant de représenter des apparences de surface réalistes et détaillées dans les applications de synthèse d'image. Nous avons, à cette fin, conçu des modèles d'habillage de surface qui permettent, en faisant varier localement les propriétés du matériau de la surface, d'y faire apparaître des détails, enrichissant ainsi les images produites.

Nous nous sommes concentrés sur plusieurs aspects :

- **l'efficacité** des modèles proposés, en terme d'*occupation mémoire* et de *rapidité d'affichage* (en veillant notamment à ce que les algorithmes et structures de données soient utilisables sur les processeurs graphiques),
- la **qualité de rendu**, en particulier en ce qui concerne le *traitement de l'aliasing* et la *résolution apparente des détails*,
- la **souplesse** de création pour les artistes utilisant nos modèles, en particulier en proposant un *fort contrôle* sur les apparences produites, à *un niveau d'abstraction modulable* : du plus global au plus détaillé, en passant par le semi–automatique.

Les sections suivantes résument nos contributions.

1.1 Textures procédurales à base de motifs

L'idée : Assembler des motifs à la volée, pour constituer automatiquement mais de manière contrôlée, une texture très large et variée (possiblement animée) sur une surface paramétrée, avec une faible occupation mémoire.

Nous avons introduit au chapitre 3 les *textures procédurales à base de motifs* [LN03], modèle d'habillage destiné aux terrains et surfaces facilement paramétrables. Il permet de représenter une très grande variété de textures à base de motifs, sans souffrir des défauts qui y sont habituellement associés : ajout de géométrie supplémentaire, alignements visibles à grande échelle, positionnement un à un des motifs par l'artiste.

En particulier, notre modèle peut prendre en charge de manière procédurale le positionnement aléatoire (sous contrôle) des motifs sur la surface. Il peut également introduire des variations d'aspect sur les motifs (couleur, taille, orientation). Ceci nous permet d'atteindre de très hautes résolutions sur de vastes domaines et à faible coût mémoire : les motifs ne sont stockés qu'une seule fois en mémoire, l'information de positionnement est implicite. L'artiste conserve un fort contrôle spatial à grande échelle sur la distribution des motifs, mais n'a plus à les positionner un par un.

Nos *textures procédurales à base de motifs* sont définies par un ensemble de composants de base, combinés pour créer un algorithme qui génère à la volée la texture, lors du dessin de la surface. Ceci permet à la fois de gagner du temps lors de la création de l'apparence de surface (grâce au contrôle global de l'apparence, l'algorithme s'occupe des variations), mais également de générer d'immenses textures de très haute résolution à faible coût mémoire (grâce à la génération procédurale à la volée). A noter que comme tous les modèles d'habillage de cette thèse (sauf le modèle de génération d'écorce du chapitre 10), notre méthode est implémentée sur processeur graphique ce qui permet d'obtenir une grande vitesse d'affichage.

Notons que nous avons également utilisé cette approche pour aborder des problèmes plus spécifiques comme la représentation de gouttes d'eau s'écoulant sur une surface au chapitre 4 [Lef03], et le dessin de l'ombre d'une forêt sur un terrain au chapitre 5. Cette approche a été poursuivie par les travaux de Wei [Wei04], qui intègrent l'idée des pavages aléatoires développés par Cohen et. al [CSHD03] dans un schéma de génération de texture à la volée proche du nôtre.

Nous avons également étendu l'idée des textures à base de motifs dans la suite de nos travaux, avec la notion de *textures composites* (chapitre 9).

1.2 Textures composites

L'idée : Attacher des motifs sur des surfaces quelconques en se passant de paramétrisation planaire globale. Définir l'apparence finale de la surface comme la combinaison arbitraire des contributions des motifs.

Nous avons étendu l'idée des textures à base de motifs, auparavant limitées aux surfaces facilement paramétrables, à des surfaces quelconques (statues, véhicules, objets manufacturés, arbres, ...). Sur ce type de surfaces, les enjeux sont un peu différents. La taille des surfaces est généralement limitée, même si l'on souhaite tout de même atteindre des résolutions de détail qui peuvent vite saturer la mémoire disponible. Le problème vient principalement de la complexité de la forme de la surface, qui entraîne, avec le placage de texture standard, gaspillage et défauts visuels (voir chapitre 2, section 2.1.6), et rend difficile la création d'un pavage ; mais également de l'hétérogénéité des besoins en détails (souvent très localisés).

Nous introduisons au chapitre 9 la notion de *textures composites* [LHN05b], qui créent l'apparence finale de la surface en combinant les contributions de motifs répartis sur la surface. Utiliser des motifs a de multiples avantages : en premier lieu, il est possible, comme précédemment, de ne stocker le motif qu'une seule fois en mémoire et de le répéter à faible coût (instanciation). En second lieu, les motifs ont souvent une petite taille en comparaison des courbes de

l'objet, ce qui permet de les appliquer *localement* sur la surface (à la manière d'autocollants), et ainsi d'éviter nombre de difficultés dues aux paramétrisations planaires globales (voir chapitre 2, section 2.1.2). Enfin, les motifs peuvent être ajoutés et modifiés dynamiquement, ce qui autorise l'édition interactive du contenu. Utilisé dynamiquement pendant le rendu, ceci permet de créer des *apparences dynamiques* (impacts, gouttes, ...) mais également d'adapter l'habillage de la surface aux éventuelles déformations de l'objet en ajustant taille et position des motifs (voir l'exemple du serpent, chapitre 9).

Notre structure de données, de faible coût mémoire puisque basée sur une structure hiérarchique en volume entourant la surface (voir chapitre 2, section 2.4), est encodée en mémoire texture et peut être directement affichée par le processeur graphique en temps réel. Il n'est pas nécessaire de modifier ni d'introduire de nouvelle géométrie. Il n'est pas nécessaire de paramétrer la surface. Enfin, l'habillage produit est correctement filtré et interpolé.
Les principales propriétés sont donc :

- une *grande résolution apparente*,
- une *faible consommation mémoire*,
- une *grande qualité visuelle* (peu de distorsion, filtrage).

Nous avons construit autour de cette représentation, à titre d'exemple, différents types d'habillage inédits, rendus possibles uniquement par notre approche.

1.3 Gestion de textures de haute résolution sur des surfaces quelconques

L'idée : Ne garder en mémoire (ne *générer*) que les parties de la texture utiles à un moment donné, à la résolution requise.

Nous avons présenté au chapitre 6 [LDN04] un modèle d'habillage dédié à la gestion de textures de haute résolution plaquées sur des géométries quelconques, possiblement animées. Nous avons montré que la plupart des travaux concernant la gestion de textures de haute résolution (en particulier le chargement progressif des données), dont la taille dépasse aisément la mémoire disponible, s'étaient jusqu'alors concentrés spécifiquement sur le rendu de terrains ; un cas particulier car la surface quasi–planaire permet une paramétrisation simple et facilite la détermination des parties de la texture visibles depuis un point de vue donné.

Cependant, les textures très détaillées sont désormais utilisées sur des géométries quelconques, éventuellement animées (par exemple un personnage), et via des paramétrisations planaires complexes (atlas de texture). Dans ce chapitre, nous introduisons une architecture de chargement progressif, conçue et implémentée sur les processeurs graphiques actuels, qui permet de gérer cette situation. Les données de texture sont chargées progressivement, en fonction du point de vue, et au fur et à mesure que l'utilisateur évolue dans la scène. Le coeur de notre architecture est un nouvel algorithme de détermination des parties utiles de la texture, très performant car il repose sur le processeur graphique pour effectuer les calculs géométriques. L'algorithme fournit également une bonne estimation (petit sur–ensemble) des parties utiles de la texture.

Notre architecture permet d'unifier les solutions classiques de chargement progressif (cache de données, chargement asynchrone, priorité de chargement), pour toutes les classes

d'objets, dans un système *transparent* à l'utilisateur : celui–ci doit simplement fournir le point de vue courant et la géométrie utilisée (qui n'est pas modifiée). Notre architecture permet également de gérer les textures compressées et procédurales. Celles–ci peuvent alors être directement générées par la carte graphique dans le cache de texture (résidant en mémoire texture), minimisant ainsi les coûteux transferts de données entre CPU et GPU. Cette méthode suit l'évolution actuelle des architectures graphiques qui s'orientent vers la mise en place de systèmes de gestion de mémoire virtuelle. Notre approche permet de minimiser les phénomènes de *swap* (échanges de données intempestifs) dans ce contexte.

Il devient possible de naviguer dans des mondes immenses, avec des paysages aux apparences variées, des personnages différents les uns des autres, sans souffrir des limitations de mémoire (puisque l'on peut efficacement transférer les données, sous forme compressée, depuis un serveur distant ou un média de masse).

1.4 Textures hiérarchiques en volume pour processeurs graphiques

L'idée : Proposer une implémentation efficace des *octree textures* pour processeurs graphiques, afin de rendre ces structures très génériques disponibles dans les applications graphiques interactives.

Nous avons proposé au chapitre 7 [LHN05a] une implémentation complète pour processeurs graphiques des *octree textures*, textures hiérarchiques en volume encodant de l'information uniquement au voisinage de la surface d'un objet ; donc sans paramétrisation planaire, et à faible coût mémoire (en particulier si la résolution des détails est hétérogène). Nous avons aussi proposé un algorithme de conversion rapide vers une texture 2D standard, permettant de choisir à la volée la représentation la mieux adaptée.

Cette contribution est technique, mais proposer cette implémentation sur processeur graphique permet de rendre les *octree textures* disponibles dans les applications interactives, au même titre que le placage de texture. En outre, nous avons montré au chapitre 8 comment de telles structures pouvaient être utilisées pour simuler un automate cellulaire à la surface d'un objet, sans souffrir des difficultés introduites par les paramétrisations planaires (distorsions, discontinuités, gaspillages dus aux espaces vides). Nous avons également vu au chapitre 9 que les *octree textures* permettent de stocker efficacement tout type d'information autour d'une surface sans paramétrisation.

1.5 S'attaquer à la résolution de nombreux non–dits concernant la qualité

L'idée : Développer des modèles d'habillage respectant les hypothèses permettant un rendu de qualité.

Les solutions développées pour produire des images de qualité, en particulier l'interpolation et le filtrage, fonctionnent uniquement sous certaines hypothèses. Par exemple, le filtrage utilisé

pour le placage de texture (généralement l'algorithme de *MIP–mapping*, voir chapitre 2, section 4) suppose une continuité de la paramétrisation planaire et une continuité du contenu de la texture.

Malheureusement ces hypothèses sont souvent oubliées, et les méthodes d'habillage classiques (pavages, atlas de texture) entraînent des défauts visuels qui ne peuvent être que difficilement et partiellement masqués (voir chapitre 2, section 2.1.6).

Nous avons pris soin de concevoir nos modèles d'habillage de manière à pouvoir filtrer et interpoler l'habillage résultant soit de manière exacte, soit de manière partielle mais en identifiant les situations délicates et en proposant, lorsque possible, des solutions alternatives.

1.6 Donner les outils pour réaliser des apparences de surface réalistes

L'idée : Donner aux artistes des outils performants pour créer des apparences de surface réalistes, y compris dans des situations difficiles (apparences dynamiques, surfaces complexes d'aspect difficile à reproduire, ...) ; et ceci au travers de modèles *génériques* mais aisément *contrôlables*, au travers d'*exemples concrets* réalisés avec nos modèles d'habillage, ou encore grâce à la création d'algorithmes dédiés à la *simulation d'une apparence* de surface spécifique.

Les modèles que nous avons évoqués tout au long de cette thèse proposent divers degrés de généricité, du point de vue des apparences qui peuvent être produites. Une grande généricité implique souvent une plus grande difficulté de création d'aspects réalistes pour les artistes. Nous pouvons classer nos modèles suivant le niveau d'intervention requis par l'utilisateur (du plus fort au plus faible), en mentionnant à chaque fois l'avantage apporté par rapport à l'existant :

- Création explicite de textures (pixel par pixel), mais sans limite de mémoire (chapitre 6), grâce à une gestion appropriée des données lors du rendu.
- Peinture interactive directe sur surface, soit point par point (grâce à l'implémentation des *octree textures* sur GPU, chapitre 7), soit motif par motif avec notre modèle de *textures composites* (chapitre 9).
- Spécification de règles s'appuyant sur des paramètres plus ou moins globaux (distribution de motifs, de matériaux, transitions, ...) pour faire générer automatiquement de très larges textures à partir de motifs (chapitre 3), à très faible coût mémoire.
- Contrôles spécifiques au sein d'un modèle dédié à une classe d'apparences (écorces du chapitre 10), permettant de créer une apparence réaliste difficile à reproduire "à la main" et un gain de temps considérable lors de l'habillage d'un grand nombre d'objets.
- Simulation pseudo–physique et entièrement automatique d'un aspect de surface dynamique jusqu'alors très difficile, voire impossible, à produire dans des applications interactives (gouttes d'eau au chapitre 4, écoulement de liquide au chapitre 8).

Nous avons également proposé tout au long de la thèse, à titre d'exemple de nos modèles d'habillage, différentes applications dédiées à la création d'apparences réalistes :

- au chapitre 4, la simulation temps réel de l'écoulement de gouttes d'eau sur une surface quasi–planaire ; chaque goutte est affichée à haute résolution, avec un effet de réfraction ;

– au chapitre 5, l'affichage des ombres détaillées d'un grand nombre d'objets (les arbres d'une forêt) sur un terrain ;
– au chapitre 8, la simulation temps réel (et interactive) de l'écoulement d'un liquide sur une surface quelconque ;
– au chapitre 9, une surface recouverte d'écailles qui coulissent les unes sur les autres en suivant les déformations de l'objet (un serpent).

Tous ces exemples sont des modèles d'habillage qui s'occupent à la fois de la génération de l'aspect de surface, de son encodage (en mémoire texture) et de son affichage.

1.7 Unification et transparence d'utilisation

L'idée : Concevoir des modèles d'habillage dont l'utilisation est transparente pour l'utilisateur, et en particulier n'implique aucune modification ni aucun ajout de géométrie pour définir l'apparence de surface. Concevoir des ponts entre représentations pour permettre de choisir à la volée celle qui est la mieux adaptée à une situation.

Les modèles que nous proposons sont accessibles via une API correspondant à l'utilisation classique d'une texture. Du point de vue de l'utilisateur il n'y a aucune différence entre habiller une géométrie avec une texture gérée par notre architecture de chargement progressif (chapitre 6), habiller un terrain par une texture procédurale à base de motifs (chapitre 3) ou une texture composite (chapitre 9). Nos modèles sont donc simples et accessibles. Les fonctions de chacun (contrôle des paramètres, ajout de motifs, ...) sont exposées via une API facilement exploitable, directement ou via un logiciel d'édition spécialisé.

En outre, tous nos modèles ont été conçus de manière à ne pas avoir à modifier ou introduire de géométrie pour des raisons d'habillage, sans lien avec la représentation des formes. Ce point est particulièrement important pour les terrains, sur lesquels on utilise très souvent une approche multi–résolution pour la géométrie, mais également pour les personnages dont la géométrie est animée.

Enfin, nos modèles sont tous facilement convertibles vers une texture standard 2D (de manière triviale ou via les algorithmes de conversion que nous fournissons), afin de pouvoir sélectionner, durant l'affichage, la représentation la mieux adaptée à une situation donnée.

1.8 Modèles d'habillage pour applications interactives

L'idée : Concevoir des modèles d'habillage adaptés aux évolutions récentes des processeurs graphiques, afin de permettre l'utilisation de nos modèles dans les applications graphiques à des performances interactives.

Nous avons pris soin de concevoir et implémenter nos modèles d'habillage sur processeurs graphiques. Il nous a fallu, à cette fin, vaincre de nombreux obstacles techniques (drivers expérimentaux, compilateurs en version béta), surmonter quelques disfonctionnements de fonctionnalités peu utilisées, donc moins robustes, et parfois faire face à des impossibilités techniques

qui nous imposaient de modifier notre algorithme. Les processeurs graphiques sont plus souples qu'avant, mais restent un matériel spécialisé. On doit donc veiller à exprimer les algorithmes en un ensemble d'opérations élémentaires directement supportées par les processeurs.

Notons qu'au travers de ces difficultés, nous avons eu la chance de développer un contact privilégié avec les constructeurs de cartes graphiques (ATI et NVIDIA), qui nous ont conseillés, fourni des prototypes, et à qui nous avons pu signaler un certain nombre de comportements inattendus ou mal spécifiés (test de stencil effectué après le *fragment program*, filtrage incorrect sur les textures en puissance de deux mais rectangulaires, problèmes de compilation, ...) et effectuer des suggestions (support natif du stockage de pavés , mode de rasterization conservatif, accès au niveau de MIP–mapping sélectionné par la carte, ...).

Nous avons notamment pu exposer à la communauté et aux constructeurs ces éléments lors d'une présentation, intitulée *Questions on the GPU*, effectuée durant la réunion finale de l'AS temps–réel, à Bordeaux, en 2003 (la présentation est disponible sur le site de l'AS temps–réel).

2 Perspectives

Avant de conclure ce manuscrit, il me semble important d'ouvrir la discussion sur certains des thèmes qui gravitent autour de l'habillage de surface.

2.1 Un modèle d'habillage universel ?

Nous avons vu, tout au long du manuscrit, que la création d'une apparence réaliste ne peut se faire que dans le cadre des contraintes imposées par le modèle d'habillage ; et inversement : les modèles d'habillage sont plus ou moins adaptés à certaines apparences de surface.

Par exemple, une image utilisée comme texture est parfaite pour représenter un unique tableau accroché sur un mur, alors qu'elle est très inefficace pour représenter un parterre de feuilles mortes sur le sol, ce que fait au contraire très bien une texture à base de motifs. A l'extrême, un modèle d'habillage peut être entièrement dédié à une apparence très spécifique (les écorces du chapitre 10), ou bien générique au point que l'artiste doive entièrement spécifier l'apparence de la surface en peignant un à un les éléments de couleur (les textures 2D classiques, ou les *octree textures*).

On peut, dès lors, se poser la question du "juste milieu". En fait, il me semble important de toujours conserver une certaine généricité. Les textures à base de motifs sont moins générales que les textures standard, mais plus efficaces sur une *très* large classe d'apparences. Elles permettent toujours aux artistes de créer une grande variété d'aspects. Un modèle très spécifique, tel que notre modèle d'écorce, même s'il offre un contrôle important sur le résultat et permet de représenter un nombre non négligeable d'écorces, pose le problème suivant : une multiplication des modèles d'habillage dédiés à des apparences spécifiques, chacun contrôlé par des paramètres propres (parfois en grand nombre), met les éventuels utilisateurs face à une complexité croissante. Cette complexité n'est pas due à un modèle en particulier, mais à la prolifération des modèles spécifiques.

Ceci est gérable, dans le cas où la scène est créée en tout ou partie par un moteur semi–automatique : typiquement, dans des applications dédiées à certains types de scènes (environnements urbains, paysages pour simulateurs de vol, etc...), où l'univers est gigantesque et très détaillé. Aucun humain ne peut prendre en charge la création des détails individuels et il faut avoir recours à un générateur automatique, qui lui–même peut faire appel à une variété de modèles d'habillage. Cependant, bien des scènes, dans l'industrie des jeux vidéos et des effets spéciaux, sont en fait créées de toutes pièces par des artistes. Dès lors, l'utilisateur va–t–il choisir un modèle pour les arbres, un pour le sol, un pour simuler des impacts sur la porte de garage, un pour les fines craquelures de la tasse en céramique posée sur la table, etc ... ? S'il s'agit d'un objet central, important pour l'application, alors probablement oui. Mais, dans le cas général, probablement pas : l'utilisateur aura recours à une texture peinte à la main, peut–être moins réaliste, peut–être moins efficace (stockage, affichage) mais plus *accessible*.

Il est donc important de s'orienter *aussi* vers des modèles d'habillage génériques, se concentrant sur de larges classes de textures plutôt que sur des cas très particuliers. Ceci s'applique aussi bien à l'encodage de l'apparence de la surface, qu'aux algorithmes de création d'aspect réalistes.

2.2 Aspects de surface réalistes : outils de création et variété

Créer des aspects de surface réalistes est une tâche difficile, qui peut être effectuée sur de petits objets à la main, mais devient impraticable si l'on doit créer une grande variété d'objets aux aspects *similaires* mais *distincts* (les personnages d'une foule, l'écorce des arbres d'une forêt). Nous avons montré, dans cette thèse, que des approches dédiées, combinées à des modèles d'habillage représentant efficacement le contenu, pouvaient réaliser des aspects de surface complexes et réalistes, jusqu'alors extrêmement difficiles à obtenir (gouttes d'eau, écoulement de liquide, écorces réalistes).

Même si ceci dépasse du cadre de cette thèse, il nous faut également nous poser la question de la prolifération de méthodes dédiées à la création d'apparences très spécifiques. Pourrait–on imaginer une méthode de création générique, qui permette de *reproduire* l'habillage réaliste d'un objet sur des modèles géométriques représentant des individus de la même catégorie ? (par exemple, habiller dix nouvelles tasses de formes variées à partir d'une seule tasse déjà habillée).

Il me semble que cette idée d'*habillage par l'exemple* est une piste intéressante, d'ailleurs déjà entrouverte par la synthèse de texture à base d'échantillons (attention cependant, la plupart des algorithmes de synthèse de texture à partir d'échantillons sont eux–même limités à la classe des textures homogènes et stationnaires, voir chapitre 2, section 5.2.2). L'objectif serait le suivant : l'utilisateur fournit un modèle géométrique de taille raisonnable, habillé d'une apparence qu'il a peinte lui–même (en s'aidant éventuellement de photographies, etc ...). Le point important est que la taille de la surface reste petite, et la tâche est donc réalisable par un artiste (notons que ce modèle pourrait également avoir été capturé par d'autres moyens, par exemple avec un algorithme de reconstruction). L'algorithme d'*habillage par l'exemple* pourrait alors, à partir de l'exemple de ce modèle géométrique habillé, générer un nouvel habillage sur un second modèle géométrique, nouveau et éventuellement plus grand. Par exemple, si l'on souhaite peindre une armée de dragons (tous les dragons étant de formes différentes), on pourrait se contenter d'en peindre un seul (voire la moitié d'un seul), et l'habillage serait automatiquement répercuté sur les autres, par *analogie*[1]. Une des difficultés, par rapport à la synthèse de texture à partir d'échantillons (problème déjà difficile en soit), est de faire *apprendre* à l'algorithme comment l'habillage est corrélé (s'il l'est) avec la forme, la géométrie de l'objet. Cependant, cet objectif ambitieux, une fois atteint, offrirait un système très souple est très *accessible* aux artistes (voire même aux utilisateurs dénués de tout sens artistique).

2.3 L'aliasing : un problème d'actualité

Il suffit de regarder quelques jeux vidéos récents pour s'en convaincre : malgré la présence d'algorithmes de sur–échantillonnage[2] dans les cartes graphiques, l'aliasing est de retour. Le trop grand nombre de triangles, combiné à l'utilisation de plus en plus fréquente de transparence (notamment sur des *billboards*) dans des scènes toujours plus complexes et détaillées (arbres, herbes,

[1] Cette idée n'est d'ailleurs pas sans rapport avec les travaux de Hertzmann et al. [HJO+01].

[2] Le sur–échantillonnage ne fait que repousser le problème, voir chapitre 2, section 4.2, et à de plus un impact important sur les performances.

fils électriques) font que le travail impressionnant des artistes est dégradé par un grouillement de couleurs pénible et insistant.

La seule chose qui limite encore l'aliasing , c'est le mécanisme de filtrage propre aux textures (voir chapitre 2, section 4). Cependant, cet ilôt de qualité risque de disparaître quand les jeux vidéos utiliseront toutes les dernières technologies des processeurs graphiques. En effet, celles–ci permettent désormais le recours à des modèles d'habillage complexes, pour lesquels le filtrage est souvent très difficile, et parfois négligé. Il est important que les concepteurs de modèles d'habillage (chercheurs en première ligne), et de tout effet graphique à base de calculs par pixel (illumination locale incluse) prennent conscience de ce fait : sans un filtrage correct, le recours à des modèles d'habillage complexes risque bien d'être un pas vers le réalisme ... au prix de la dégradation de la qualité des images, l'aliasing étant extrêmement bien perçu par notre système de vision.

D'autre part, il ne s'agit pas du seul problème : l'aliasing géométrique, contre lequel le filtrage des modèles d'habillage ne peut rien (voir chapitre 2, section 4.5), est également de plus en plus présent à cause d'un plus grand nombre de primitives géométriques représentant des formes fines (cables, tuyaux, grilles, petits objets décoratifs, etc ...). Cette course aux micro–polygones, surtout si elle s'accompagne d'une perte de cohérence spatiale de l'habillage de surface, risque également de conduire à une explosion des effets d'aliasing .

Poussée par la surenchère en matière d'effets graphiques sophisitiqués et par la course aux performances, la communauté graphique risque donc de briser un des piliers soutenant la qualité visuelle des images produites en synthèse d'image. De ce point de vue, nous espérons, avec cette thèse, offrir quelques pistes, tant pour préserver la cohérence spatiale de l'habillage (en le rendant indépendant de la géométrie), que pour limiter le recours à de la géométrie pour représenter les détails, et s'attaquer au filtrage des représentations complexes, dans le contexte de leur implémentation sur processeurs graphiques. Nous espérons avoir au moins donné matière à prendre conscience des enjeux et des dangers cachés derrière le problème de l'habillage de surface, ainsi que du grand nombre de degrés de liberté offerts par les fonctionnalités étonnantes des nouvelles (et futures) générations de processeurs graphiques.

Bibliographie

[AL60] L.V. Ahlfors and L.Sario. *Riemann Surfaces*. Princeton University Press, 1960. Princeton Mathematics Series, No. 26. 32

[AMN03] Timo Aila, Ville Miettinen, and Petri Nordlund. Delay streams for graphics hardware. *ACM Transactions on Graphics*, 22(3) :792–800, 2003. 66

[And95] T.L. Anderson. *Fracture Mechanics, Fundamental and Applications*. CRC Press, 1995. ISBN 0-8493-4260-0. 193, 198

[Ash01] Michael Ashikhmin. Synthesizing natural textures. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics*, pages 217–226. ACM Press, 2001. 72

[AW90] Gregory D. Abram and Turner Whitted. Building block shaders. In *Proceedings of ACM SIGGRAPH*, pages 283–288. ACM Press, 1990. 71, 88

[BAC96] Andrew C. Beers, Maneesh Agrawala, and Navin Chaddha. Rendering from compressed textures. In *Proceedings of ACM SIGGRAPH*, Computer Graphics Proceedings, Annual Conference Series, pages 373–378, August 1996. 40

[BD02] David Benson and Joel Davis. Octree textures. In *Proceedings of ACM SIGGRAPH*, pages 785–790. ACM SIGGRAPH, ACM Press, 2002. 19, 21, 50, 155, 178, 180

[BFH+04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus : stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3) :777–786, 2004. 77

[Bio] Bionatics. AMAP : a plant and scenery modeling tool. `http://www.bionatics.com/`. 187, 206

[Bli77] J. F. Blinn. Models of light reflection for computer synthesized pictures. In *Proceedings of ACM SIGGRAPH*, pages 192–198, July 1977. 28

[Bli78] James F. Blinn. Simulation of wrinkled surfaces. In *Proceedings of ACM SIGGRAPH*, volume 12(3), pages 286–292, August 1978. 29, 35, 44, 68, 75, 116, 201

[Blo85] Jules Bloomenthal. Modeling the mighty maple. In B. A. Barsky, editor, *Computer Graphics (Proceedings of SIGGRAPH)*, volume 19, pages 305–311, 1985. 191, 203

[BM93] Barry G. Becker and Nelson L. Max. Smooth transitions between bump rendering algorithms. In James T. Kajiya, editor, *Computer Graphics (Proceedings of SIGGRAPH)*, volume 27, pages 183–190, August 1993. 43

[BN76] James F. Blinn and Martin E. Newell. Texture and reflection in computer generated images. *Proceedings of ACM SIGGRAPH*, 10(2) :266–266, 1976. 27, 45

[Bod] Bodypaint3D. Texturing software. `http://www.maxoncomputer.com/`. 69

[Bro91] D. Broek. *Elementary Engineering Fracture Mechanics*. Kluwer Academic, 1991. ISBN 9-0247-2580-1. 193

[BVI91] Chakib Bennis, Jean-Marc Vézien, and Gérard Iglésias. Piecewise surface flattening for non-distorted texture mapping. In Thomas W. Sederberg, editor, *ACM Transactions on Graphics*, volume 25, pages 237–246, July 1991. Proceedings of ACM SIGGRAPH. 33

[Car84] Loren Carpenter. The a-buffer, an antialiased hidden surface method. In *Computer Graphics (Proceedings of SIGGRAPH)*, volume 18, pages 103–108, 1984. 68

[Cat74] Edwin E. Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. Ph.d. thesis, University of Utah, December 1974. 26, 29, 30

[CE98] David Cline and Parris K. Egbert. Interactive display of very large textures. In *Proceedings of VIS*, pages 343–350, 1998. 52, 132

[CH02] Nathan A. Carr and John C. Hart. Meshed atlases for real-time procedural solid texturing. *ACM Transactions on Graphics*, 21(2) :106–131, 2002. 33

[CH04] Nathan A. Carr and John C. Hart. Painting detail. *ACM Transactions on Graphics*, 23(3) :845–852, 2004. 38, 69

[cHtW95] Siu chi Hsu and Tien tsin Wong. Simulating dust accumulation. *IEEE Computer Graphics Applications*, 15(1) :18–22, 1995. 73

[CMRS98] P. Cigogni, C. Montani, C. Rocchini, and R. Scopino. A general method for recovering attributes values on simplified meshes. pages 59–66. ACM Press, 1998. 33

[Coo85] Robert L. Cook. Antialiasing by Stochastic Sampling. In *ACM SIGGRAPH, State of the Art in Image Synthesis seminar notes*. July 1985. 106

[Coo02] Michael Mc Cool. Sparse Texture Storage for Graphics Accelerators, 2002. Technical Talk. `http://www.cgl.uwaterloo.ca/Projects/rendering/Talks/sparse/slides.pdf`. 40

[Cro84] Franklin C. Crow. Summed-area tables for texture mapping. In *Computer Graphics (Proceedings of SIGGRAPH)*, pages 207–212. ACM Press, 1984. 65

[CSHD03] Michael F. Cohen, Jonathan Shade, Stefan Hiller, and Oliver Deussen. Wang tiles for image and texture generation. *ACM Transactions on Graphics*, 22(3) :287–294, July 2003. Proceedings of ACM SIGGRAPH. 54, 55, 73, 218

[CT82] R. L. Cook and K. E. Torrance. A reflectance model for computer graphics. *ACM Transactions on Graphics*, pages 7–24, January 1982. 28

[CXGS02] Yanyun Chen, Yingqing Xu, Baining Guo, and Heung-Yeung Shum. Modeling and rendering of realistic feathers. In *Proceedings of ACM SIGGRAPH*, pages 630–636. ACM Press, 2002. 73

[DB97] Jeremy S. De Bonet. Multiresolution sampling procedure for analysis and synthesis of texture images. In Turner Whitted, editor, *Proceedings of ACM SIGGRAPH*, pages 361–368. ACM SIGGRAPH, Addison Wesley, August 1997. 71

[DBH00] Jurgen Dollner, Konstantin Baumman, and Klaus Hinrichs. Texturing techniques for terrain visualization. In *Proceedings of VIS*, pages 227–234, 2000. 52, 134

[DDSD03] Xavier Décoret, Frédo Durand, François Sillion, and Julie Dorsey. Billboard clouds for extreme model simplification. In *Proceedings of ACM SIGGRAPH*. ACM Press, 2003. 75, 76

[Dee] Deep paint3D. Texturing software. `http://www.righthemisphere.com/products/dp3d/`. 69

[DGPR02] David DeBry, Jonathan Gibbs, Devorah DeLeon Petty, and Nate Robins. Painting and rendering textures on unparameterized models. In *Proceedings of ACM SIGGRAPH*, pages 763–768. ACM SIGGRAPH, ACM Press, 2002. 19, 21, 50, 155, 178, 180

[DHL+98] Oliver Deussen, Patrick Hanrahan, Bernd Lintermann, Radomír Mech, Matt Pharr, and Przemyslaw Prusinkiewicz. Realistic modeling and rendering of plant ecosystems. *Proceedings of ACM SIGGRAPH*, pages 275–286, July 1998. 187

[Dis98] J.-M. Dischler. Efficiently rendering macrogeometric surface structures using bi-directional texture functions. In *Rendering Techniques '98*, Eurographics, pages 169–180, 1998. 42

[DMA02] M. Desbrun, M. Meyer, and P. Alliez. Intrinsic parameterizations of surface meshes. *Computer Graphics Forum*, 21(3) :209–218, 2002. 32, 34

[DMLG02] J.M. Dischler, K. Maritaud, B. Lévy, and D. Ghazanfarpour. Texture particles. 21(3) :401–410, 2002. 48, 72, 172

[DN04] Philippe Decaudin and Fabrice Neyret. Rendering forest scenes in real-time. pages 93–102, june 2004. 75

[DPH96] Julie Dorsey, Hans Køhling Pedersen, and Pat Hanrahan. Flow and changes in appearance. In Holly Rushmeier, editor, *Proceedings of ACM SIGGRAPH*, Annual Conference Series, pages 411–420. ACM SIGGRAPH, Addison Wesley, August 1996. 73

[dREF+88] Phillippe de Reffye, Claude Edelin, Jean Francon, Marc Jaeger, and Claude Puech. Plant models faithful to botanical structure and development. In *Computer Graphics (Proceedings of SIGGRAPH)*, volume 22, pages 151–158, August 1988. 187

[DvGNK99] Kristin J. Dana, Bram van Ginneken, Shree K. Nayar, and Jan J. Koenderink. Reflectance and texture of real-world surfaces. *ACM Transactions on Graphics*, 18(1) :1–34, 1999. 42

[EF01] Alexei A. Efros and William T. Freeman. Image quilting for texture synthesis and transfer. *Proceedings of ACM SIGGRAPH*, pages 341–346, August 2001. 71, 72

[EMP+94] David Ebert, Kent Musgrave, Darwyn Peachey, Ken Perlin, and Worley. *Texturing and Modeling : A Procedural Approach*. Academic Press, October 1994. ISBN 0-12-228760-6. 50

[FFBG01] Randima Fernando, Sebastian Fernandez, Kavita Bala, and Donald P. Greenberg. Adaptive shadow maps. In *Proceedings of ACM SIGGRAPH*, pages 387–390. ACM Press, 2001. 40

[FH05] M. S. Floater and K. Hormann. Surface parameterization : a tutorial and survey. In ISBN 3-540-21462-3 Springer, editor, *Advances on Multiresolution in Geometric Modelling*, 2005. 31

[FLCB95] Kurt W. Fleischer, David H. Laidlaw, Bena L. Currin, and Alan H. Barr. Cellular texture generation. *Computer Graphics (Proceedings of SIGGRAPH)*, 29 :239–248, 1995. 48, 73, 172

[Flo97] Michael S. Floater. Parametrization and smooth approximation of surface triangulations. *Computer Aided Geometric Design*, 14(4) :231–250, 1997. 32, 33

[Fou92] Alain Fournier. Normal distribution functions and multiple surfaces. In *Graphics Interface, Workshop on Local Illumination*, pages 45–52, May 1992. 68

[FSH04] Kayvon Fatahalian, Jeremy Sugerman, and Pat Hanrahan. Understanding the efficiency of gpu algorithms for matrix-matrix multiplication. In *Proceedings of the ACM SIGGRAPH / Eurographics Conference on Graphics Hardware*. Eurographics Association, August 2004. 78

[FvDFH90] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics : Principles and Practices (2nd Edition)*. Addison Wesley, 1990. 24

[GAC+89] A. S. Glassner, J. Arvo, R. L. Cook, E. Haines, P. Hanrahan, P. Heckbert, and D. B. Kirk. *Introduction to Ray Tracing*. Academic Press, 1989. 24

[GdM85] Andre Gagalowicz and Song de Ma. Model driven synthesis of natural textures for 3-D scenes. pages 91–108, 1985. 71

[GGSC96] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. In *Proceedings of ACM SIGGRAPH*, pages 43–54. ACM Press, 1996. 66, 162

[GGW+98] B. Guenter, C. Grimm, D. Wood, H. Malvar, and F. Pighin. Making faces. In *Proceedings of ACM SIGGRAPH*, pages 55–66. ACM Press, 1998. 34

[Gla98] Andrew Glassner. Penrose tiling. *IEEE Compututer Graphics and Applications*, 18(4) :78–86, 1998. 54

[Gob01] Stephane Gobron. Three-dimensionnal surface cellular automata for visual simulation of natural phenomena. Master's thesis, Iwate University, 2001. 73

[GP03] G. Guennebaud and M. Paulin. Efficient screen space approach for Hardware Accelerated Surfel Rendering. In *Vision, Modeling and Visualization, Munich*, pages 1–10. IEEE Signal Processing Society, 19-21 novembre 2003. 76

[Gre86] Ned Greene. Environment mapping and other applications of world projections. *IEEE Computer Graphics Applications*, 6(11) :21–29, 1986. 27, 45

[Gri20] A.A. Griffith. The phenomena of rupture and flow in solids. *Philosophical Transactions*, 221 :163–198, 1920. 193

[GS86] B. Grünbaum and G.C. Shepard. *Tilings and Patterns*. 1986. 54

[GSX00] Baining Guoa, Harry Shum, and Ying-Qing Xu. Chaos mosaic : Fast and memory efficient texture synthesis, 2000. Microsoft Research technical report MSR-TR-2000-32. 72

[GTGB84] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modelling the interaction of light between diffuse surfaces. In *Computer Graphics (Proceedings of SIGGRAPH)*, volume 18, pages 212–22, July 1984. 26

[GY98] Michael E. Goss and Kei Yuasa. Texture tile visibility determination for dynamic texture loading. In *Proceedings of the ACM SIGGRAPH / Eurographics Conference on Graphics Hardware*, 1998. 52, 134

[HATK00] S. Haker, S. Angenent, A. Tannenbaum, and R. Kikinis. Conformal surface parameterization for texture mapping. *IEEE Transaction on Visualization and Computer Graphics*, 6 :181–187, 2000. 32

[HB95] David J. Heeger and James R. Bergen. Pyramid-Based texture analysis/synthesis. In Robert Cook, editor, *Proceedings of ACM SIGGRAPH*, pages 229–238. ACM SIGGRAPH, Addison Wesley, August 1995. 71

[HB96] J. Hart and B. Baker. Implicit modeling of tree surfaces, 1996. In Implicit Surfaces '96. 191

[HCSL02] Mark J. Harris, Greg Coombe, Thorsten Scheuermann, and Anselmo Lastra. Physically-based visual simulation on graphics hardware. In *Proceedings of the ACM SIGGRAPH / Eurographics Conference on Graphics Hardware*, 2002. 74, 167

[Hec86] Paul S. Heckbert. Survey of texture mapping. *IEEE Computer Graphics & Applications*, 6 :56–67, November 1986. 63

[Hec90] Paul S. Heckbert. Adaptive radiosity textures for bidirectional ray tracing. In *Proceedings of ACM SIGGRAPH*, pages 145–154. ACM Press, 1990. 29, 40

[Hei91] Tim Heidmann. Real shadows, real time. In *IRIS Universe*, volume 18, pages 23–31. Silicon Graphics, Inc, 1991. 27

[HG] K. Hormann and G. Greiner. Mips : An efficient global parametrization method. in Proceedings of Curve and Surface Design, 1999. 32

[HH90] Pat Hanrahan and Paul E. Haeberli. Direct WYSIWYG painting and texturing on 3D shapes. In Forest Baskett, editor, *Proceedings of ACM SIGGRAPH*, volume 24, pages 215–223, August 1990. 69

[HHLH05] Samuel Hornus, Jared Hoberock, Sylvain Lefebvre, and John C. Hart. Zp+ : correct z-pass stencil shadows. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. ACM, ACM Press, April 2005. 27

[HJO+01] Aaron Hertzmann, Charles E. Jacobs, Nuria Oliver, Brian Curless, and David H. Salesin. Image analogies. In *Proceedings of ACM SIGGRAPH*, pages 327–340. ACM Press, 2001. 71, 72, 225

[HKK98] K. Hirota, H. Kato, and T. Kaneko. A physically-based simulation model of growing tree barks. *IPSJ Journal*, 39(11), 1998. in Japanese. 191

[HM91] Paul Heckbert and Henry Moreton. Interpolation for polygon texture mapping and shading. In *State of the Art in Computer Graphics : Visualization and Modeling*, pages 101–111. Springer-Verlag, 1991. 35

[Hop96] Hugues Hoppe. Progressive meshes. In *Proceedings of ACM SIGGRAPH*, Computer Graphics Proceedings, Annual Conference Series, pages 99–108, August 1996. 178

[HS93] Paul Haeberli and Mark Segal. Texture mapping as a fundamental drawing primitive. In *Proceedings of the Eurographics Workshop on Rendering*, pages 259–266. Eurographics, June 1993. 30

[HS99] Wolfgang Heidrich and Hans-Peter Seidel. Realistic, hardware-accelerated shading and lighting. In *Proceedings of ACM SIGGRAPH*, Computer Graphics Proceedings, Annual Conference Series, pages 171–178, August 1999. 28

[HTK98] K. Hirota, Y. Tanoue, and T. Kaneko. Generation of crack patterns with a physical model. *The Visual Computer*, 1998. 191

[Hut98] Tobias Huttner. High resolution textures. In *Proceddings of VisSym*, 1998. 52, 132

[IC01] Takeo Igarashi and Dennis Cosgrove. Adaptive unwrapping for interactive texture painting. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics*, pages 209–216, 2001. 69

[Ing13] C.E. Inglis. Stresses in a plate due to the presence of cracks and sharp corners. *Transactions of the Institute of Naval Architects*, 55 :219–241, 1913. 193, 198

[JC95] Henrik Wann Jensen and Niels Jørgen Christensen. Photon maps in bidirectional monte carlo ray tracing of complex objects. *Computers & Graphics*, 19(2) :215–224, March 1995. 25

[Kaj85] James T. Kajiya. Anisotropic reflection models. In B. A. Barsky, editor, *Computer Graphics (Proceedings of SIGGRAPH)*, volume 19(3), pages 15–21, July 1985. 28, 29, 35

[KE02] M. Kraus and T. Ertl. Adaptive Texture Maps. In *Proceedings of the ACM SIGGRAPH / Eurographics Conference on Graphics Hardware*, pages 7–15. ACM SIGGRAPH, 2002. 40, 98, 111, 146

[KG79] Douglas Scott Kay and Donald Greenberg. Transparency for computer synthesized images. In *Proceedings of ACM SIGGRAPH*, pages 158–164. ACM Press, 1979. 27, 45

[KK89] James T. Kajiya and Timothy L. Kay. Rendering fur with three dimensional textures. In Jeffrey Lane, editor, *Computer Graphics (Proceedings of SIGGRAPH)*, volume 23(3), pages 271–280, July 1989. 24, 75

[KM99] Jan Kautz and Michael D. McCool. Interactive rendering with arbitrary brdfs using separable approximations. In *Proceedings of the Eurographics Workshop on Rendering*, June 1999. 28

[KSG03] V. Kraevoy, A. Sheffer, and C. Gotsman. Matchmaker : constructing constrained texture maps. *ACM Transactions on Graphics*, 22(3) :326–333, 2003. Proceedings of ACM SIGGRAPH. 34

[KTI+01] Tomomichi Kaneko, Toshiyuki Takahei, Masahiko Inami, Naoki Kawakami, Yasuyuki Yanagida, Taro Maeda, and Susumu Tachi. Detailed shape representation with parallax mapping. In *Proceedings of the ICAT 2001 (The 11th International Conference on Artificial Reality and Telexistence)*, pages 205–208, December 2001. 44

[LD98] Bernd Lintermann and Oliver Deussen. A modelling method and user interface for creating plants. *Computer Graphics Forum*, 17(1) :73–82, 1998. 187

[LDN04] Sylvain Lefebvre, Jérome Darbon, and Fabrice Neyret. Unified texture management on arbitrary meshes. Technical Report 5210, INRIA, May 2004. 132, 219

[Lef03] Sylvain Lefebvre. *Shaderx2 : Shader Programming Tips & Tricks*, chapter Drops of water texture sprites, pages 190–206. Wordware Publishing, 2003. ISBN 1-55622-988-7. 115, 218

[Lév01] Bruno Lévy. Constrained texture mapping for polygonal meshes. In *Proceedings of ACM SIGGRAPH*, Computer Graphics Proceedings, Annual Conference Series, pages 417–424, August 2001. 34

[LHN04] Sylvain Lefebvre, Samuel Hornus, and Fabrice Neyret. All-purpose texture sprites. Technical Report 5209, INRIA, May 2004. 173

[LHN05a] Sylvain Lefebvre, Samuel Hornus, and Fabrice Neyret. *Octree Textures on the GPU*. Addison–Wesley, 2005. 155, 220

[LHN05b] Sylvain Lefebvre, Samuel Hornus, and Fabrice Neyret. Texture sprites : Texture elements splatted on surfaces. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. ACM, ACM Press, April 2005. 173, 218

[LL94] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear–warp factorization of the viewing transformation. In Andrew Glassner, editor, *Proceedings*

of ACM SIGGRAPH, Computer Graphics Proceedings, Annual Conference Series, pages 451–458. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0. 75

[LLH04] Yanxi Liu, Wen-Chieh Lin, and James Hays. Near-regular texture analysis and manipulation. *ACM Transactions on Graphics*, 23(3) :368–376, 2004. Proceedings of ACM SIGGRAPH. 71

[LM98] Bruno Lévy and Jean-Laurent Mallet. Non-distorted texture mapping for sheared triangulated meshes. In *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, pages 343–352, July 1998. 32, 33

[LN02] Sylvain Lefebvre and Fabrice Neyret. Synthesizing bark. In *Proceedings of the Eurographics Workshop on Rendering*, 2002. 188

[LN03] Sylvain Lefebvre and Fabrice Neyret. Pattern based procedural textures. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics*. ACM, ACM Press, 2003. 86, 217

[LPRM02] Bruno Lévy, Sylvain Petitjean, Nicolas Ray, and Jérome Maillot. Least squares conformal maps for automatic texture atlas generation. *ACM Transactions on Graphics*, 21(3) :362–371, July 2002. Proceedings of ACM SIGGRAPH. 33

[LW85] M. Levoy and T. Whitted. The use of points as a display primitive. Technical report, Computer Science Department, University of North Carolina at Chapel Hill, 1985. 76

[MAA01] Michael D. McCool, Jason Ang, and Anis Ahmad. Homomorphic factorization of brdfs for high-performance rendering. In *Proceedings of ACM SIGGRAPH*, Computer Graphics Proceedings, Annual Conference Series, pages 171–178, August 2001. 28

[MAB+97] J. Marks, B. Andalman, P. A. Beardsley, W. Freeman, S. Gibson, J. Hodgins, T. Kang, B. Mirtich, H. Pfister, W. Ruml, K. Ryall, J. Seims, and S. Shieber. Design galleries : a general approach to setting parameters for computer graphics and animation. In *Proceedings of ACM SIGGRAPH*, pages 389–400. ACM Press/Addison-Wesley Publishing Co., 1997. 71

[Max88] Nelson L. Max. Horizon mapping : shadows for bump-mapped surfaces. *The Visual Computer*, 4(2) :109–117, July 1988. 29

[MDG00] K. Maritaud, J.M. Dischler, and D. Ghazanfarpour. Rendu réaliste d'arbres à courte distance. In *AFIG, (Actes des 13èmes journées de l'AFIG)*, 2000. 74, 191, 203

[MDG01a] S. Merillou, Jean-Michel Dischler, and D. Ghazanfarpour. Surface scratches : measuring, modeling and rendering. *The Visual Computer*, 17(1) :30–45, 2001. 73

[MDG01b] Stephane Merillou, Jean-Michel Dischler, and Djamchid Ghazanfarpour. Corrosion : Simulating and rendering. In *Graphics Interface 2001*, pages 167–174, June 2001. 73

[MGAK03] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg : a system for programming graphics hardware in a c-like language. *ACM Transactions on Graphics*, 22(3) :896–907, 2003. 77

[MGW01] Tom Malzbender, Dan Gelb, and Hans Wolters. Polynomial texture maps. In *Proceedings of ACM SIGGRAPH*, Computer Graphics Proceedings, Annual Conference Series, pages 519–528. ACM Press, August 2001. 42

[MH84] Gene S. Miller and C. Robert Hoffman. Illumination and reflection maps : Simulated objects in simulated and real environments, 1984. Course Notes for Advanced Computer Graphics Animation, ACM SIGGRAPH. 45

[ML88] Song De Ma and Hong Lin. Optimal texture mapping. pages 421–428, September 1988. 32

[MMDJ01] M. Muller, L. McMillian, J. Dorsey, and R. Jagnow. Real-time deformation and fracture of stiff materials. In *Eurographics Workshop on Animation*, pages 113–124. Springer Wien NewYork, 2001. 192

[MN98a] Alexandre Meyer and Fabrice Neyret. Interactive volumetric textures. pages 157–168, New York City, NY, Jul 1998. Eurographics, Springer Wein. ISBN. 75

[MN98b] Alexandre Meyer and Fabrice Neyret. Interactive Volumetric Textures. In G. Drettakis and N. Max, editors, *Rendering Techniques : Eurographics Rendering Workshop*, pages 157–168. Eurographics, Springer Wein, July 1998. 112

[MTP+04] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. *ACM Transactions on Graphics*, 23(3) :787–795, 2004. 77

[MTPS04] Antoine McNamara, Adrien Treuille, Zoran Popovic, and Jos Stam. Fluid control using the adjoint method. *ACM Transactions on Graphics*, 23(3), August 2004. Proceedings of ACM SIGGRAPH. 73

[MYV93] Jérôme Maillot, Hussein Yahia, and Anne Verroust. Interactive texture mapping. In *Proceedings of ACM SIGGRAPH*, Computer Graphics Proceedings, Annual Conference Series, pages 27–34, August 1993. 33

[NC99] Fabrice Neyret and Marie-Paule Cani. Pattern-based texturing revisited. In *Proceedings of ACM SIGGRAPH*, pages 235–242. ACM SIGGRAPH, Addison Wesley, August 1999. `http://www-imagis.imag.fr/Membres/Fabrice.Neyret/publis/SIG99/`. 56, 73

[Ney96] Fabrice Neyret. *Textures Volumiques pour la Synthèse d'images*. PhD thesis, Université Paris-XI - INRIA, 1996. `http://www-imagis.imag.fr/Membres/Fabrice.Neyret/publis/thesefabrice-fra.html`. 75

[NF99] M. Neff and E. Fiume. A visual model for blast waves and fracture. In *Graphics Interface*, pages 193–202. Communications Society, 1999. `http://www.graphicsinterface.org/proceedings/1999/117/`. 192

[NHS02] Fabrice Neyret, Raphael Heiss, and Franck Senegas. Realistic Rendering of an Organ Surface in Real-Time for Laparoscopic Surgery Simulation. *the Visual*

Computer, 18(3) :135–149, may 2002. http://www-imagis.imag.fr/Publications/2002/NHS02. 48

[NTB+91] A. Norton, G. Turk, B. Bacon, J. Gerth, and P. Sweeney. Animation of fracture by physical modeling. *Visual Computer*, 7 :210–219, 1991. 192

[Nvi02] Nvidia. CG Toolkit Reference Manual v1.5 http://www.cgshaders.org, 2002. 103

[OBM00] Manuel M. Oliveira, Gary Bishop, and David McAllister. Relief texture mapping. In *Proceedings of ACM SIGGRAPH*, Computer Graphics Proceedings, Annual Conference Series, pages 359–368, July 2000. 75

[OH99] J.F. O'Brien and J.K. Hodgins. Graphical modeling and animation of brittle fracture. In *Proceedings of ACM SIGGRAPH*, pages 137–146. ACM SIGGRAPH, 1999. 192

[OHHM02] Marc Olano, John C. Hart, Wolfgang Heidrich, and Michael McCool. *Real-Time Shading*. AK Peters, Ltd., July 2002. ISBN 1-56881-180-2. 27

[OKS03] Marc Olano, Bob Kuehne, and Maryann Simmons. Automatic shader level of detail. In *Proceedings of the ACM SIGGRAPH / Eurographics Conference on Graphics Hardware*, pages 7–14, July 2003. 67

[Ope03] OpenGL SGI. *The OpenGL Graphics System : A Specification (version 1.5). Mark Segal and Kurt Akeley*, chapter 3 - Rasterization. Sillicon Graphics, Inc., 2003. 140

[Pea85] Darwyn R. Peachey. Solid texturing of complex surfaces. In *Computer Graphics (Proceedings of SIGGRAPH)*, volume 19, pages 279–286. ACM SIGGRAPH, 1985. Proceedings of ACM SIGGRAPH. 49, 50, 70

[Per85] Ken Perlin. An image synthesizer. In B. A. Barsky, editor, *Proceedings of ACM SIGGRAPH*, volume 19(3), pages 287–296, July 1985. See also http://www.noisemachine.com/. 49, 50, 66, 70, 71, 90

[Per02] Ken Perlin. Improving noise. In *Proceedings of ACM SIGGRAPH*, pages 681–682. ACM Press, 2002. 66

[PF90] Pierre Poulin and Alain Fournier. A model for anisotropic reflection. In Forest Baskett, editor, *Computer Graphics (Proceedings of SIGGRAPH)*, volume 24(4), pages 273–282, August 1990. 28

[PFH00] Emil Praun, Adam Finkelstein, and Hugues Hoppe. Lapped textures. In *Proceedings of ACM SIGGRAPH*, Computer Graphics Proceedings, Annual Conference Series, pages 465–470, July 2000. 48, 49, 72, 172, 182

[PH89] Ken Perlin and Eric M. Hoffert. Hypertexture. In Jeffrey Lane, editor, *Computer Graphics (Proceedings of SIGGRAPH)*, volume 23(3), pages 253–262, July 1989. 24

[PHL91] J. W. Patterson, S. G. Hoggar, and J. R. Logie. Inverse displacement mapping. *Computer Graphics Forum*, 10(2) :129–139, June 1991. 43

[PLH88] Przemyslaw Prusinkiewicz, Aristid Lindenmayer, and James Hanan. Developmental models of herbaceous plants for computer imagery purposes. In John Dill, editor, *Computer Graphics (Proceedings of SIGGRAPH)*, volume 22, pages 141–150, August 1988. 187

[PMTH01] Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *Proceedings of ACM SIGGRAPH*, Computer Graphics Proceedings, Annual Conference Series, pages 159–170, August 2001. 77

[PP96] Federl P. and Prusinkiewicz P. A texture model for cracked surfaces, with an application to tree bark. In *Proceedings of Western Computer Graphics Symposium*, pages 23–29, March 1996. 191

[PPD01] Eric Paquette, Pierre Poulin, and George Drettakis. Surface aging by impacts. In *Graphics Interface*, pages 175–182, 2001. 73

[PPD02] Eric Paquette, Pierre Poulin, and George Drettakis. The simulation of paint cracking and peeling. In *Graphics Interface*, pages 59–68, 2002. 73

[PS99] J. Portilla and E. Simoncelli. Texture modeling and synthesis using joint statistics of complex wavelet coefficients. In *IEEE Workshop on Statistical and Computational Theories of Vision*, Fort Collins, CO, 1999. 71

[PZvBG00] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels : Surface elements as rendering primitives. In *Proceedings of ACM SIGGRAPH*, pages 335–342, 2000. 76

[Rah] Sharif Rahman. Probabilistic simulation of fracture by meshless methods. `http://www.ccad.uiowa.edu/projects/solidmech/probsim.html`. 198

[RB85] William T. Reeves and Ricki Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. In B. A. Barsky, editor, *Computer Graphics (Proceedings of SIGGRAPH)*, volume 19(3), pages 313–322, July 1985. 24

[Ree83] W. T. Reeves. Particle systems – a technique for modeling a class of fuzzy objects. *ACM Transactions on Graphics*, 2 :91–108, April 1983. Proceedings of ACM SIGGRAPH. 24

[Rem96] E. Remila. Approximate strip packing. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science*, page 31. IEEE Computer Society, 1996. 38

[RL00] S. Rusinkiewicz and M. Levoy. QSplat : A multiresolution point rendering system for large meshes. In *Proceedings of ACM SIGGRAPH*, pages 343–352, 2000. 76

[Rus97] S. Rusinkiewicz. A survey of brdf representation for computer graphics, 1997. 28

[SAM] Jacob Strom and Tomas Akenine-Moller. Packman : Texture compression for mobile phones. ACM SIGGRAPH Technical Sketch, 2004. 40

[Sam86] M. Samek. Texture mapping and distortion in digital graphics. *The Visual Computer*, 2(5) :313–320, 1986. 32

[SAWG91] Francois X. Sillion, James R. Arvo, Stephen H. Westin, and Donald P. Greenberg. A global illumination solution for general reflectance distributions. In *Computer Graphics (Proceedings of SIGGRAPH)*, volume 25, pages 187–196, July 1991. 26

[Sch94] Christophe Schlick. A survey of shading and reflectance models. *Computer Graphics Forum*, 13(2) :121–131, June 1994. 28

[SCH03] Pradeep Sen, Mike Cammarano, and Pat Hanrahan. Shadow silhouette maps. *ACM Transactions on Graphics*, 22(3) :521–526, July 2003. Proceedings of ACM SIGGRAPH. 41

[Sen04] Pradeep Sen. Silhouette maps for improved texture magnification. In *Proceedings of the ACM SIGGRAPH / Eurographics Conference on Graphics Hardware*. Eurographics Association, August 2004. 36, 41, 42

[SGSH02] Pedro V. Sander, Steven J. Gortler, John Snyder, and Hugues Hoppe. Signal-specialized parameterization. In *Proceedings of the Eurographics Workshop on Rendering*, pages 87–100, 2002. 38

[SH02] A. Sheffer and J. Hart. Seamster : Inconspicuous low-distortion texture seam layout, 2002. 32, 33

[She96] Jonathan Richard Shewchuk. Triangle : Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. In *Applied Computational Geometry : Towards Geometric Engineering*. Springer-Verlag, 1996. From the First ACM Workshop on Applied Computational Geometry. `http://www-2.cs.cmu.edu/~quake/triangle.html`. 201

[SSBD01] Cyril Soler, François Sillion, Frédéric Blaise, and Philippe Dereffye. A physiological plant growth simulation engine based on accurate radiant energy transfer. Technical Report 4116, INRIA, February 2001. `http://www-imagis.imag.fr/Publications/2001/SSBD01`. 187

[SSGH01] Pedro V. Sander, John Snyder, Steven J. Gortler, and Hugues Hoppe. Texture mapping progressive meshes. In Eugene Fiume, editor, *Proceedings of ACM SIGGRAPH*, pages 409–416. ACM Press, 2001. 33, 66, 162

[Sta97] Jos Stam. Aperiodic Texture Mapping. Technical Report R046, European Research Consortium for Informatics and Mathematics (ERCIM), January 1997. `http://www.ercim.org/publication/technical_reports/046-abstract.html`. 54, 73

[Stu99] Pixar Animation Studios. Flagrant abuses of a perfectly nice texture system. In *ACM SIGGRAPH Course Notes 24*, 1999. 188

[SWB98] Peter-Pike J. Sloan, David M. Weinstein, and J. Dean Brederson. Importance driven texture coordinate optimization. *Computer Graphics Forum*, 17(3) :97–104, 1998. 38

[SWB00] Jeffrey Smith, Andrew Witkin, and David Baraff. Fast and controllable simulation of the shattering of brittle objects. In *Graphics Interface*, May 2000. http://www.graphicsinterface.org/proceedings/2000/145/. 192

[SWW+04] Jörg Schmittler, Sven Woop, Daniel Wagner, Wolfgang J. Paul, and Philipp Slusallek. Realtime ray tracing of dynamic scenes on an fpga chip. In *Proceedings of the ACM SIGGRAPH / Eurographics Conference on Graphics Hardware*. Eurographics Association, August 2004. 26

[Tex] Texture paint. Texturing software. http://home.t-online.de/home/uwe_maurer/texpaint.htm. 69

[TF88] D. Terzopoulos and K. Fleisher. Modeling inelastic deformation : Viscoelasticity, plasticity, fracture. In *Proceedings of ACM SIGGRAPH*, pages 269–278, 1988. 192, 193

[THCM04] M. Tarini, K. Hormann, P. Cignoni, and C. Montani. Polycubemaps. *ACM Transactions on Graphics*, 23(3) :853–860, August 2004. Proceedings of ACM SIGGRAPH. 9, 46, 47

[TMJ98] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. The Clipmap : A Virtual Mipmap. In *Proceedings of ACM SIGGRAPH*, pages 151–158. ACM SIGGRAPH, July 1998. 52, 132

[TS66] K. E. Torrance and E. M. Sparrow. Polarization, directional distribution, and off-specular peak phenomena in light reflected from roughened surfaces. *Journal of Optical Society of America*, 1966. 28

[TS67] K. E. Torrance and E. M. Sparrow. Theory for off-specular reflection from roughened surfaces. *Journal of Optical Society of America*, 1967. 28

[Tur91] Greg Turk. Generating textures for arbitrary surfaces using reaction-diffusion. In Thomas W. Sederberg, editor, *Proceedings of ACM SIGGRAPH*, volume 25, pages 289–298, July 1991. 73

[Tur01] Greg Turk. Texture synthesis on surfaces. *Proceedings of ACM SIGGRAPH*, pages 347–354, August 2001. 72

[Vau93] H. Vaucher. *Guide des écorces*. DELACHAUX et NIESTLE, 1993. ISBN 2-603-00850-1. 189

[WE98] Rüdiger Westermann and Thomas Ertl. Efficiently using graphics hardware in volume rendering applications. *Proceedings of ACM SIGGRAPH*, pages 169–178, July 1998. ISBN 0-89791-999-8. 75

[Wei04] Li-Yi Wei. Tile-based texture mapping on graphics hardware. In *Proceedings of the ACM SIGGRAPH / Eurographics Conference on Graphics Hardware*. Eurographics Association, August 2004. 111, 218

[Wel04] Terry Welsh, 2004. Parallax Mapping with Offset Limiting : A Per–Pixel Approximation of Uneven Surfaces, *Infiscape Corporation*, http://www.infiscape.com/doc/parallax_mapping.pdf. 44

[Whi80] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6) :343–349, June 1980. 24, 35

[Wil78] L. Williams. Casting curved shadows on curved surfaces. pages 270–274. ACM, ACM Press, 1978. 27, 125, 143

[Wil83] Lance Williams. Pyramidal parametrics. In *Computer Graphics (Proceedings of SIGGRAPH)*, volume 17, pages 1–11, July 1983. 63

[WK91] Andrew Witkin and Michael Kass. Reaction-diffusion textures. In Thomas W. Sederberg, editor, *Proceedings of ACM SIGGRAPH*, volume 25, pages 299–308, July 1991. 73

[WL00] Li-Yi Wei and Marc Levoy. Fast texture synthesis using tree-structured vector quantization. In Kurt Akeley, editor, *Proceedings of ACM SIGGRAPH*, pages 479–488. ACM SIGGRAPH, ACM Press, 2000. 71, 72

[WL01] Li-Yi Wei and Marc Levoy. Texture synthesis over arbitrary manifold surfaces. *Proceedings of ACM SIGGRAPH*, pages 355–360, August 2001. 72

[Wor96] Steven P. Worley. A cellular texturing basis function. In Holly Rushmeier, editor, *Proceedings of ACM SIGGRAPH*, pages 291–294. ACM SIGGRAPH, Addison Wesley, August 1996. 50, 70, 71

[WWHG03] Xi Wang, Lifeng Wang, Ligang Liu Shimin Hu, and Baining Guo. Interactive modeling of tree bark. In *Proceedings of Pacific Graphics 2003*, 2003. 191

[WWT+03] Lifeng Wang, Xi Wang, Xin Tong, Stephen Lin, Shimin Hu, Baining Guo, and Heung-Yeung Shum. View-dependent displacement mapping. *ACM Transactions on Graphics*, 22(3) :334–339, 2003. 43

[Xfr] Xfrog. `http://www.xfrogdownloads.com/`. 187, 206

[YJ04] K. Yerex and M. Jagersand. Displacement mapping with ray-casting in hardware. In *ACM SIGGRAPH Sketch*, 2004. 43

[ZBr] ZBrush. Texturing software. `http://pixologic.com/home/home.shtml`. 69

[ZG02] Steve Zelinka and Michael Garland. Towards real-time texture synthesis with the jump map. In *Proceedings of the Eurographics Workshop on Rendering*, pages 99–104. Eurographics Association, 2002. 72

[ZIK98] S. Zhukov, A. Iones, and G. Kronin. Using light maps to create realistic lighting in real-time applications. In *In Proceedings of WSCG*, pages 464–471, 1998. 29

[ZSBP02] Wenting Zheng, Hanqiu Sun, Hujun Bao, and Qunsheng Peng. Rendering of virtual environments based on polygonal & point-based models. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 25–32. ACM Press, 2002. 76

[ZZV+03] Jingdan Zhang, Kun Zhou, Luiz Velho, Baining Guo, and Heung-Yeung Shum. Synthesis of progressively-variant textures on arbitrary surfaces. *ACM Transactions on Graphics*, 22(3) :295–302, 2003. Proceedings of ACM SIGGRAPH. 72

Titre : Modèles d'habillage de surface pour la synthèse d'images.

Résumé : La complexité visuelle des objets ne réside pas uniquement dans leurs formes, mais également dans l'apparence de leurs surfaces. Les détails de surface ne sont pas nécessaires à la compréhension des formes. Ils sont cependant primordiaux pour enrichir l'aspect visuel des images produites, et répondre aux besoins croissants de réalisme des applications graphiques modernes (jeux vidéos, effets spéciaux, simulateurs).

En synthèse d'image, les modèles d'habillage de surface, tels que le placage de texture, sont utilisés conjointement à la représentation des formes pour enrichir l'aspect des objets. Ils permettent de représenter les variations des propriétés du matériau le long de la surface, et ainsi de créer de nombreux détails, allant de fins motifs colorés à des aspects rugueux ou abîmés.

Cependant, la demande croissante de l'industrie, en terme de richesse, de qualité et de finesse de détails, implique une utilisation des ressources toujours plus grande : quantité de données à stocker, temps et difficulté de création pour les artistes, temps de calcul des images. Les modèles d'habillage de surface actuels, en particulier le placage de texture, ne permettent plus de répondre efficacement à toutes les situations.

Nous proposons dans cette thèse de nouveaux modèles d'habillage, qui permettent d'atteindre de très hautes résolutions de détails sur les surfaces, avec peu de mémoire, un temps de création réduit et avec des performances interactives : nous les avons conçus pour les processeurs graphiques programmables récents. Nos approches sont multiples : combinaison semi–automatique de motifs sur la surface, gestion de texture dépendante du point de vue, méthodes basées sur des textures hiérarchiques pour éviter le recours à une paramétrisation planaire globale. Nous proposons également, à titre d'exemple, des applications concrètes de nos modèles d'habillage génériques à des cas difficiles, voire impossibles, à réaliser auparavant.

Title : Texturing Methods for Computer Graphics

Abstract : The visual complexity of objects does not only come from their shape, but also from their surface appearence. Surface details are not essential to understand shape. However they are crucial to the realism of the produced images, required by modern graphics applications such as video games, special effects and simulators.

In Computer Graphics, texturing methods such as texture mapping are commonly used to enrich the apearence of an object's surface. They allow to introduce variations of the object's material properties along the surface, and thus to create details, from fine colored patterns to bumped or damaged aspects. However, the increasing needs in terms of quality, richness and detail resolution implies an increasing demand on ressources : memory consumption, time and difficulty of creation for artists, time required to compute an image. Existing texturing methods, in particular texture mapping, no longer answer efficiently to all the texturing situations.

We propose in this thesis new texturing methods, able to reach extremly high resolutions of details on surfaces while using little memory, reducing creation time for artists, and performing at interactive frame rates. Our methods are designed to run on recent programmable graphics hardware. We rely on multiple approches : semi–automatic compositing of patterns along the surface, view-dependant texture data management, methods based on hierarchical textures to avoid using a global planar parameterization. We also propose practical applications of our generic texturing methods to create effects that were difficult - if not impossible - to achieve previously.