Rapport de stage de fin de 1ère année de Magistère
effectué au laboratoire iMAGIS sous la direction de Fabrice Neyret

Textures procédurales en temps réel avec OpenGL®

Soutenu en septembre 1998

Table des matières

Chapitre 1

Introduction

1.1 Présentation du laboratoire *i*MAGIS

Le laboratoire *i*MAGIS, situé à Grenoble, s'occupe d'informatique graphique et en particulier de *synthèse d'images*. *i*MAGIS est un projet commun de l'INRIA, du CNRS, de l'Université Joseph Fourier et de l'Institut national polytechnique de Grenoble dirigé par Claude Puech.

Parmi les thèmes de recherche figurent la visualisation d'environnements complexes, le rendu réaliste de l'éclairage, la modélisation et l'animation par modèles physiques et basés sur les surfaces implicites, la réalité augmentée et la géométrie algorithmique.

Le laboratoire *i*MAGIS développe ses propres outils de modélisation, d'animation et de rendu. Les recherches effectuées ont des applications pratiques très diverses comme la simulation d'éboulement, la simulation chirurgicale et la réalisation d'effets spéciaux pour le cinéma. Le laboratoire participe également à des projets européens (projets ARCADE et SIMULGEN) et travaille avec des partenaires industriels (tels que Hewlett Packard et le Bureau des Recherches Géologiques et Minière).

Fabrice Neyret, chercheur au CNRS dans le laboratoire *i*MAGIS , s'est particulièrement intéressé aux *textures*, et à la manière d'augmenter le réalisme des applications tridimensionnelles temps réel. Ses recherches actuelles portent sur les *représentations alternatives* et l'amélioration des méthodes de texture.

1.2 Les textures en synthèse d'image

Après une brève présentation de la synthèse d'image, cette section donne un aperçu des deux principaux types d'algorithmes utilisés en synthèse d'images et introduit le concept de texture. Pour une approche plus complète de la synthèse d'images, un livre tel que [FvDF90] est recommandé.

1.2.1 Génération des images de synthèse

En imagerie de synthèse, nous voulons faire dessiner par un ordinateur des images réalistes d'un univers virtuel.

La première étape est le *modelage* : l'utilisateur définit, grâce à un logiciel appelé *modeleur*, des objets tridimensionnels, et place des sources de lumière qui éclairent ces objets. L'assemblage de tous ces objets et ces sources lumineuses forme une *scène*.

Nous voulons dessiner sur l'écran de l'ordinateur ce qu'un observateur verrait si l'écran était une fenêtre ouvrant sur la scène. La position de l'observateur et de l'écran dans la scène définissent le *point de vue*.

Pour créer un film, l'utilisateur définit également le mouvement des objets et du point de vue dans la scène grâce à un logiciel d'*animation*. L'ordinateur doit alors dessiner des images de la scène à intervalles de temps régulier.

La scène est ensuite *rendue*, c'est à dire dessinée sur l'écran. L'écran est une grille fine dont chaque maille, appelée *pixel*[1], possède une couleur indépendante des autres. Chaque pixel est stocké en mémoire par l'intensité de ses trois composantes, rouge, verte et bleue, qui définissent sa couleur. Rendre la scène revient donc à déterminer la couleur de chaque pixel de l'écran.

Il existe plusieurs algorithmes de rendu.

Certains privilégient la qualité et d'autres la vitesse. Parmi ces derniers, les algorithmes *temps réel* permettent à l'utilisateur de bouger le point de vue et les objets tout en visualisant immédiatement le rendu grâce à une vitesse de calcul de l'ordre de quelques dizaines d'images par seconde. Ceci est possible grâce à des circuits électroniques spécialisés qui prennent en charge une partie des calculs à une vitesse beaucoup plus élevée que ce qui est possible par logiciel.

L'algorithme de tracé de rayons

L'algorithme de *tracé de rayons* calcule le trajet des rayons lumineux qui sont émis par les sources lumineuse, diffusés et réfléchis par les objets de la scène et viennent frapper l'oeil de l'observateur.

En fait le trajet est suivi à l'envers : un rayon est envoyé depuis le point de vue à travers le centre de chaque pixel de l'écran et vient toucher un objet de la scène ; la couleur du pixel considéré est la couleur du premier objet intersecté par le rayon, au point d'intersection P. Cette couleur dépend non seulement de la couleur de l'objet lui-même, mais aussi de l'éclairage de P. D'autres rayons sont donc envoyés du point d'intersection P vers les sources lumineuses pour savoir comment la lumière émise est atténuée par d'éventuels objets entre P et la source. Si la surface de l'objet est réfléchissante, la couleur en P dépend également de la couleur du point qu'il réfléchi. Pour trouver ce point, un nouveau rayon est envoyé de P dans la direction réfléchie (symétrique du rayon incident par rapport à la normale de l'objet en P).

Une équation *d'illumination locale* indique comment combiner les couleurs de tous ces rayons pour obtenir celle de P. Cette équation fait intervenir différents paramètres physiques de l'objet (couleur, transparence, réflectivité, etc.). Elle fait aussi intervenir le vecteur normal en P à la surface, qui est un paramètre géométrique et non physique de l'objet.

L'algorithme de tracé de rayons se résume donc à des calculs de rayons réfléchis, d'intersections entre rayons et objets et à l'évaluation de l'équation d'illumination locale.

Cet algorithme offre une grande qualité, mais il faut compter de l'orde de la dizaine de minutes pour rendre une scène simple sur les ordinateurs actuels. Il n'est donc par envisageable pour l'instant de l'utiliser pour les applications temps réel.

Les algorithmes projectifs

Ces algorithmes décomposent chaque objet en facettes polygonales planes qui sont *projetées* sur l'écran de l'ordinateur.

Chaque facette est définie par les coordonnées de ses sommets. La projection de ces sommets sur l'écran donne un polygone plan qu'il est facile de dessiner. L'opération consistant à calculer l'ensemble des pixels intérieurs à un polygone est appelée *rasterisation*. La couleur de chaque pixel généré par rasterisation est déterminée par l'équation d'illumination locale évoquée plus haut.

Cependant, en chaque pixel de l'écran se projettent des points issus de plusieurs facettes différentes et il ne faut dessiner que celui le plus proche de l'observateur ; c'est le problème de l'*occlusion*.

Ce problème peut être résolu de différentes façons et nous présentons ici l'algorithme du *z-buffer* qui est utilisé dans la bibliothèque OpenGL®. Le z-buffer est un tableau qui contient la profondeur de chaque pixel de l'écran, c'est à dire sa distance à l'observateur. Dès qu'un pixel est généré par rasterisation, sa profondeur est comparée à celle stockée dans le z-buffer. Si le nouveau pixel a une profondeur plus faible, il remplace le précédent sur l'écran et le z-buffer est mis à jour, sinon le nouveau pixel est simplement ignoré.

Parfois, pour gagner du temps, l'équation d'illumination locale n'est calculée rigoureusement qu'aux sommets de chaque facette et un mécanisme d'interpolation permet d'obtenir la couleur en tout pixel de

[1] Pixel est abréviation de *picture element*. Toutes les images que nous considérerons, l'écran de l'ordinateur compris, sont stockés dans des tableaux de pixels.

la facette. De plus, les objets courbes doivent être décomposés en facettes, ce qui leur donne un aspect polygonal. Les algorithmes projectifs offrent donc une qualité de rendu inférieure à celle du tracé de rayons.

D'un autre côté, l'algorithme de projection de facettes avec z-buffer est implémenté efficacement en matériel. Il est donc très employé dans les applications temps réel. La bibliothèque OpenGL® que nous utiliserons par la suite est basée sur cet algorithme.

1.2.2 Les textures

Définition

Chaque algorithme de rendu ne sait dessiner que quelques *primitives* de base simples ; les objets complexes doivent alors être décomposés en de telles primitives. La seule primitive des algorithmes projectifs est la facette plane tandis que l'algorithme de tracé de rayons offre un choix plus vaste de primitives, mais ce ne sont encore que des objets très simples (sphère, cône, etc.) dont l'intersection avec un rayon se calcule facilement.

L'aspect d'une primitive est contrôlée d'une part par sa géométrie et d'autre part par des propriétés physiques telles que la couleur, la reflectance, la transparence, etc. qui interviennent dans l'équation d'illumination locale. Si l'aspect géométrique d'une primitive est global, sa couleur peut varier à sa surface. Il n'y a plus alors *une* couleur de la primitive mais une *texture* qui définit la couleur en chaque point de la primitive.

Il est possible d'imaginer également des textures pour d'autres paramètres physiques locaux comme la reflectance, la transparence, etc.

Utilisation

Dans les applications temps réel, la vitesse de calcul d'une scène est un élément crucial. Il est donc nécessaire de trouver un compromis entre réalisme et nombre de primitives dessinées. L'utilisation de primitives texturées permet de réduire considérablement le nombre de primitives nécessaire pour dessiner un objet convaincant. Ainsi un mur de briques peut être rendu par un seul pavé dont les faces possèdent une texture donnant l'aspect des briques au lieu d'être rendu comme un ensemble de pavés unicolores représentant individuellement chaque brique. Une texture ajoute à une primitive des petits *détails* : géométrie et texture décrivent la primitive à deux échelles différentes et complémentaires.

Une primitive texturée est souvent plus coûteuse à rendre qu'une primitive non texturée. Cependant, le surcoût dû au texturage est largement compensé par les calculs géométriques (intersection avec un rayon, projection et rasterisation) économisés du fait de la réduction du nombre de primitives.

Le concept de texture est également une aide précieuse au modelage. Il permet de distinguer clairement la *forme* d'un objet (l'ensemble des primitives géométriques qui le constituent) de sa *matière* (des textures qui définissent ses paramètres physiques locaux tels que sa couleur, sa reflectance, etc). Il existe ainsi des bibliothèques de textures simulant l'aspect de différents bois, métaux, roches, etc. directement applicables sur n'importe quel objet.

Dans ce qui suit, nous allons nous intéresser plus particulièrement aux textures définissant la couleur d'un objet.

Chapitre 2

Différents types de textures

Il existe plusieurs manières de définir une texture. Nous nous intéresserons aux textures plaquées et aux textures procédurales. Le *plaquage* de texture consiste à plaquer une image bidimensionnelle sur la primitive. L'approche *procédurale* consiste à calculer en chaque point de la surface de l'objet une fonction mathématique définissant son aspect ; nous nous intéresserons au modèle de Perlin basé sur l'utilisation d'une fonction *bruit*. Chaque méthode a ses avantages et ses inconvénients que nous évoquerons plus loin. Les différents algorithmes de calcul d'image de synthèse sont eux-mêmes adaptés à l'une ou l'autre des méthodes.

2.1 Textures plaquées

Si nous reprenons l'exemple du mur de briques, l'idée la plus naturelle est de plaquer sur un pavé une image ou une photographie d'un vrai mur de briques[1]. L'idée de plaquer une image sur un objet est très ancienne : il s'agit de la première méthode implémentée en logiciel et reste encore pratiquement la seule, à ce jour, à être implémentée en matériel.

2.1.1 Principe

De manière générale, plaquer une texture sur une primitive consiste à associer à chaque point de sa surface, des coordonnées (u, v) dans une image bidimensionnelle. (u, v) sont appelées les *coordonnées de texture*. La couleur de la primitive en un point est alors la couleur de l'image en (u, v).

L'image est stockée en mémoire comme un tableau de pixels, appelé souvent lui-même *texture*. Toutefois les coordonnées (u, v) prennent, à priori, leur valeur dans $\mathbb{R}^2$, il faut donc trouver un moyen de se ramener aux valeurs stockées dans le tableau.

Si le tableau est de taille $n \times m$, une manière simple est d'associer à $(u, v) \in [0; 1]^2$ la couleur du pixel de position $(\lfloor u \cdot n \rfloor, \lfloor v \cdot m \rfloor)$ dans le tableau. Il est facile, ensuite, de prolonger virtuellement la texture par périodicité en associant à $(u, v) \in \mathbb{R}^2$ la couleur du pixel $(\lfloor u \cdot n \rfloor \textit{ mod } n, \lfloor v \cdot m \rfloor \textit{ mod } m)$ dans le tableau (*mod* représente la fonction *modulo*).

Pour améliorer la qualité du rendu, sans augmenter la taille de la texture, on recourt à divers traitements complémentaires ; il est notamment possible d'interpoler les couleurs des pixels de l'image voisins de $(u \cdot n, v \cdot m)$.

2.1.2 Utilisation

Cette méthode est particulièrement adaptée aux surfaces paramétrables car une paramétrisation associe à tout point de la surface des coordonnées de texture (u, v).

Le cas de la primitive *facette* plane est particulièrement intéressant car la correspondance entre les coordonnées (x, y, z) et les coordonnées de texture (u, v) d'un point est une simple projection, peu coûteuse

[1] Il faut en réalité trois photographies différentes car le mur de briques n'a pas le même aspect vu de face, de haut ou sur la tranche !

en temps de calcul et ne provocant pas de déformation : il suffit de définir les coordonnées de texture (u, v) aux sommets de trois points de la facette ; la valeur en tout point de la facette s'obtient alors par interpolation linéaire. C'est pour cela que la plupart des matériels graphiques 3D proposent le plaquage de texture sur des triangles.

Pour les primitives non planes, le plaquage de texture peut amener des déformations de l'image initiale ; il est ainsi impossible de plaquer correctement un dessin plan sur une sphère ([BVI91] propose une méthode pour réduire les distorsions lors du plaquage de textures sur les surfaces courbes). Le placage de texture est également difficile sur les surfaces sans paramétrisation intrinsèque, comme les surfaces définies par une fonction implicite (Pedersen a récemment proposé dans [Ped95] une méthode de plaquage de textures sur les surfaces implicites).

Le même problème se pose pour plaquer une texture sur un objet décomposé en facettes (soit obtenues par *triangulation* d'une surface définie mathématiquement, soit acquises par un *scanner* tridimensionnel) car il faut déterminer les coordonnées de texture en chaque sommet des facettes.

2.1.3 Inconvénients

Un inconvénient de cette méthode est la place occupée par les textures en mémoire, en particulier pour une application temps réel qui doit accéder à de nombreuses textures simultanément. Les matériels graphiques 3D possèdent actuellement quelques méga-octets (pour les cartes de jeux) à plusieurs centaines de méga-octets (pour les stations graphiques hautes performances) dédiés au stockage des textures, ce qui constitue encore de sérieuses limites.

D'autre part, les images utilisées pour le plaquage de texture sont généralement des dessins ou des photographies digitalisées. Outre la résolution limitée par la mémoire disponible, ces photographies manquent souvent de flexibilité : bien que l'on puisse modifier l'image point par point, il est impossible de modifier facilement des critères macroscopiques comme le grain d'un bois ou la position d'une veine dans un marbre. De plus, toujours pour des raisons d'occupation mémoire, le grandes surfaces texturées sont généralement conçues à partir d'une petite image répétée périodiquement et cette répétition est difficile à camoufler.

2.1.4 Extension aux textures volumiques

Il est imaginable d'utiliser un tableau tridimensionnel au lieu de bidimensionnel ; un élément du tableau ne s'appelle plus alors *pixel*, mais *voxel*[2]. Il faut alors utiliser trois coordonnées de texture (u, v, w).

Un premier avantage est qu'il existe une correspondance simple entre les coordonnées d'un point et ses coordonnées de texture : il suffit de poser $u = x$, $v = y$ et $w = z$. Les objets n'apparaissent plus alors *enveloppés* par une image qui épouse leur surface, mais *taillés* dans un bloc solide. Ainsi, la figure 2.1 montre un cube *plein* d'une texture de marbre. La surface complexe de la figure 2.2, pour laquelle il serait difficile de définir des coordonnées de texture bidimensionnelles en chaque point, est habillée par la même texture volumique que le cube précédent.

Malheureusement, si on évite la recherche difficile d'une paramétrisation limitant les déformations, on perd le contrôle sur la portion exacte de la texture qui affleure à la surface de l'objet et donc sur son aspect une fois habillé. De plus il est très difficile d'obtenir des photographies volumiques d'un matériau réel[3]. Enfin, les textures tridimensionnelles sont beaucoup moins disponibles sur les matériels graphiques que les textures bidimensionnelles et elles occupent encore plus de place mémoire.

En pratique, les images volumiques sont plus utilisées pour le rendu volumique médical[4] que pour donner un aspect réaliste à une primitive. Nous les utiliserons également dans le cadre de ce stage pour construire des textures procédurales.

[2] Voxel est l'abréviation de *volume element*. Une image volumique est stockée dans tableau tridimensionnel de voxels.

[3] En fait, la texture volumique utilisée pour les figures 2.1 et 2.2 n'est pas une photographie mais une texture procédurale calculée en chaque point de l'espace (voir section suivante). Elle illustre cependant bien le concept de texture *volumique* dont il est question dans ce paragraphe.

[4] Dans le rendu volumique direct, l'objet à afficher est une densité de matière stockée dans la mémoire de texture comme un tableau tridimensionnel et dessinée en *tranches* bidimensionnelles. La texture est issue d'un appareil d'acquisition tridimensionnel (un scanner, par exemple).

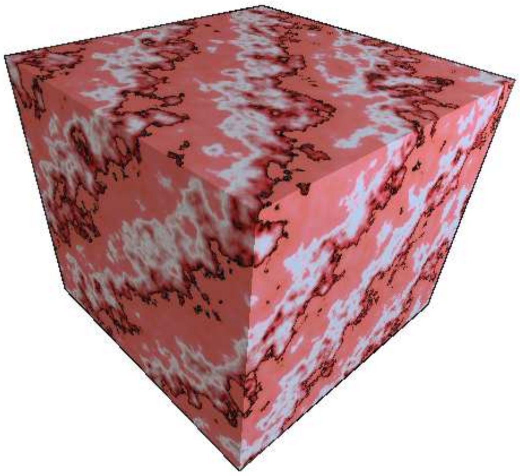

FIG. 2.1 – Cube habillé par une texture volumique procédurale de marbre.

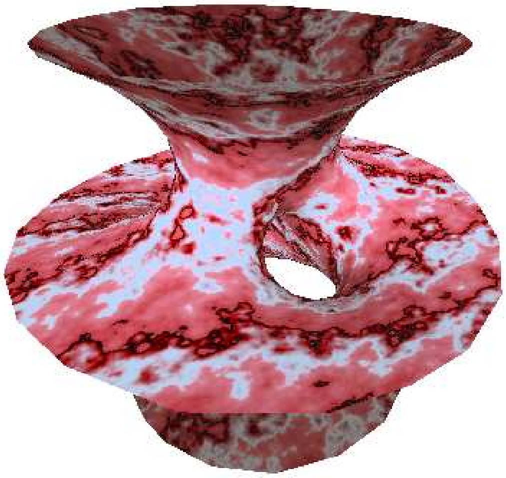

FIG. 2.2 – Surface complexe *taillée* dans la même texture volumique.

Un modèle utilisant des volumes de texture pour les plaquer sur des surface a été introduit par Kajiya dans [KK89] puis perfectionné par F.Neyret dans [Ney98]. Il permet de créer de la fourrure sur la peau d'un animal.

2.2 Textures procédurales de Perlin

Les textures procédurales ont été proposées par Ken Perlin en 1985 dans [Per85]. Le modèle de Perlin est volumique : il s'agit d'une fonction qui associe une couleur à tout point de l'espace. L'effet dans ce cas est donc encore un objet taillé dans un bloc solide. De plus, contrairement aux textures plaquées, les textures procédurales sont *modélisées* par une équation mathématique et ne nécessitent pas d'acquisition et de stockage d'image.

2.2.1 Principe

Les textures procédurales de Perlin sont basées sur une fonction de *bruit* $B : \mathbb{R}^3 \rightarrow [0;1]$ appelée *bruit turbulent*. Cette fonction a des variations aléatoires mais douces (fig. 2.13) : deux points proches ont des valeurs proches tandis que deux points éloignés sont décorrélés.

Simulation de l'agate

Une texture d'*agate* est obtenue en appliquant une *colormap* C au bruit précédent. La colormap est une fonction qui associe une couleur à chaque valeur prise par B.

La couleur finale d'un point est donc :

$$couleur(x, y, z) = C(B(x, y, z))$$

La figure 2.3 donne un exemple d'agate.

La colormap utilisée (fig. 2.14) *seuille* le bruit en colorant en noir les valeurs faibles et en blanc les îlos de haute valeur, ce qui forme des *tâches* dans la texture.

Simulation du marbre et du bois

Une texture de *marbre* ou de *bois* est obtenue en utilisant des bruits turbulents $B_x,\ B_y,\ B_z : \mathbb{R}^3 \to [-1; 1]$ pour perturber une fonction $f : \mathbb{R}^3 \to \mathbb{R}$, caractéristique du matériau *pur*.

La fonction caractéristique du *marbre* est (fig. 2.4) :

$$f(x, y, z) = x$$

et celle du *bois* est (fig. 2.8) :

$$f(x, y, z) = \sqrt{x^2 + y^2}$$

La colormap, appliquée ensuite, a pour effet de colorer chaque ligne de niveau de f. Une colormap avec quelques pics de couleur sur un fond noir (fig. 2.15) fait apparaître des veines colorées verticales dans un marbre pur (fig. 2.5) et des anneaux circulaires dans un bois pur (fig. 2.9).

Pour donner leur réalisme à ces textures, on ne calcule plus la fonction caractéristique au point (x, y, z), mais au point $(x + \alpha B_x(x, y, z), y + \alpha B_y(x, y, z), z + \alpha B_z(x, y, z))$ perturbé par le bruit turbulent.

La texture est donc définie par :

$$couleur(x, y, z) = C(f(x + \alpha B_x(x, y, z), y + \alpha B_y(x, y, z), z + \alpha B_z(x, y, z)))$$

L'effet obtenu est une déformation des veines du marbre (fig. 2.7) ou des anneaux du bois (fig. 2.10) qui suivent les lignes de niveau de la fonction f perturbée par le bruit turbulent.

Le paramètre α et le *grain* s du bruit (voir le paragraphe suivant) contrôlent l'amplitude et la période de la perturbation.

Autres utilisations de l'approche procédurale de Perlin

De manière générale, l'approche procédurale de Perlin consiste à perturber une fonction grâce à un bruit turbulent B. Dans ce stage, nous nous intéresserons uniquement à la génération du marbre, du bois et de l'agate bien qu'il existe beaucoup d'autres applications de cette approche .

En voici quelques exemples :

- Il est possible d'utiliser le bruit B pour perturber une image bidimensionnelle I :

  $$couleur(x, y) = I(x + \alpha B_x(x, y), y + \alpha B_y(x, y))$$

  Ceci permet, par exemple, de simuler la déformation d'une image par des vagues à la surface de l'eau.
- Il est également possible d'utiliser le bruit B pour engendrer du *bump mapping*. Il s'agit de perturber, en chaque point de la primitive, la normale à la surface. Celle-ci est responsable de l'effet de *relief* dans l'équation d'illumination locale. On donne ainsi un aspect *bosselé* à une surface sans avoir à la décomposer en primitives géométriques plus petites.

  Ceci permet, par exemple, de créer des vagues à la surface de l'eau ([FR86]) ou du crépi sur un mur.

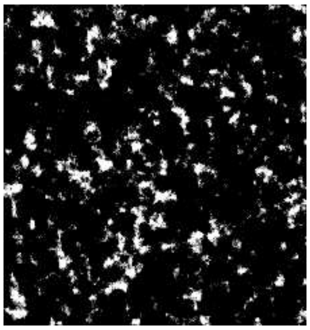

FIG. 2.3 – Texture procédurale d'agate.

- Gardner utilise un modèle procédural de Perlin sur des ellipses pour modéliser et animer des nuages ([Gar85]).

De nombreux autres exemples d'utilisation de l'approche procédurales de Perlin sont donnés dans [EMP$^+$94].

2.2.2 Construction du bruit turbulent

B, B_x, B_y et B_z sont des fonctions *bruit turbulent* (fig. 2.13) obtenues par la superposition, à des fréquences croissantes, d'un bruit *pseudo-périodique* (fig. 2.12) $P : \mathbb{R}^3 \to [-1;1]^3$:

$$B_c(x,y,z) = \frac{\sum_{i=0}^{n-1} \frac{1}{2^i} P_c(2^i s\ x, 2^i s\ y, 2^i s\ z)}{\sum_{i=0}^{n-1} \frac{1}{2^i}}$$

s contrôle la *pseudo fréquence* du bruit turbulent, c'est à dire son *grain*. $\frac{1}{s}$ correspond à la distance moyenne entre deux taches de l'agate et à la période d'oscillation des veines du marbre ou des anneaux du bois.

n est le nombre d'*octaves* du bruit, idéalement infinie pour une fonction fractale. De grandes valeurs, tout en augmentant la qualité, augmentent le temps de calcul. Une valeur de 4 est généralement acceptable. Des valeurs plus petites donnent un aspect *lissé* au bruit.

Pour construire une fonction P de pseudo-période 1, il suffit d'interpoler des valeurs aléatoires de $[-1;1]$ placées sur une grille tridimensionnelle infinie de maille unité (fig. 2.11 et 2.12). La grille infinie est simulée par un tableau tridimensionnel fini de taille $N \times N \times N$ qui contient des valeurs aléatoires ; un processus de réindiçage permet ensuite de ramener un entier relatif quelconque à un entier de l'intervalle $[0 \dots N-1]$, sans périodicité apparente.

L'utilisation d'une interpolation cubique garantit que B est $\mathcal{C}^1$, ce qui se traduit visuellement par une fonction douce. Toutefois, une interpolation linéaire donne des résultats assez satisfaisant pour un coût beaucoup plus faible . Il s'agit aussi de l'interpolation la plus fréquemment implémentées en matériel.

Notons également qu'il est possible d'*animer* une texture procédurale en considérant une fonction P non plus tridimensionnelle mais quadridimensionnelle, fabriquée à partir d'un tableau quadridimensionnel de valeurs aléatoires. Un marbre ou un bois animé n'a pas beaucoup de sens, mais l'animation est particulièrement importante pour des vagues ou les nuages procéduraux.

Grâce à l'interpolation des valeurs en quatre dimension, P est continue par rapport au temps (et même $\mathcal{C}^1$ pour une interpolation cubique). Ceci garantit que les motifs de la texture évoluent progressivement.

2.2.3 Application d'une colormap

Dans la définition d'une texture procédurale de Perlin, la colormap est aussi importante que la fonction caractéristique f. La fonction caractéristique définit la forme des lignes de niveaux de la texture tandis que la colormap définit leur couleur.

FIG. 2.4 – Fonction *marbre* $f(x, y) = x$ pure.

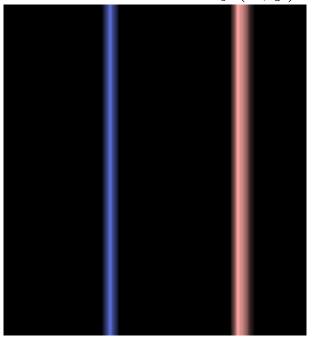

FIG. 2.5 – Une colormap composée de deux *pics* de couleur créé des *veines* dans le marbre pur.

FIG. 2.6 – Fonction marbre perturbée un bruit turbulent B : $f(x, y) = x + B(x)$.

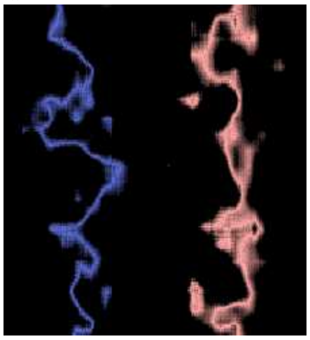

FIG. 2.7 – La colormap est appliquée cette fois à la fonction marbre perturbée.

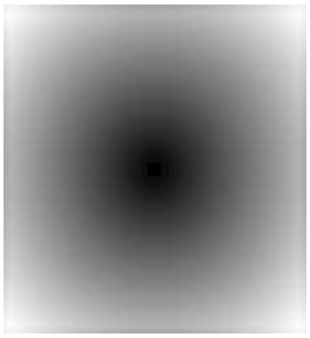

FIG. 2.8 – Fonction *bois* $f(x,y) = \sqrt{x^2+y^2}$ pure.

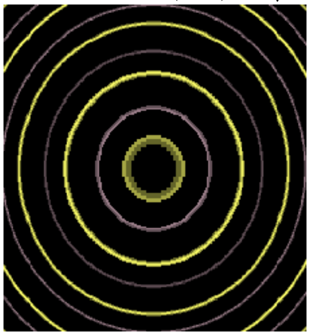

FIG. 2.9 – Une colormap fait apparaître des *anneaux* dans le bois pur.

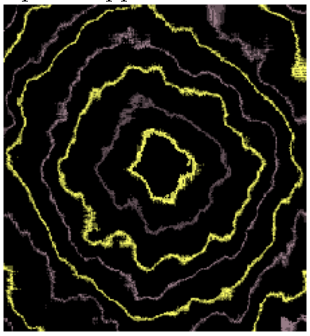

FIG. 2.10 – Les anneaux du bois sont perturbés par un bruit turbulent B.

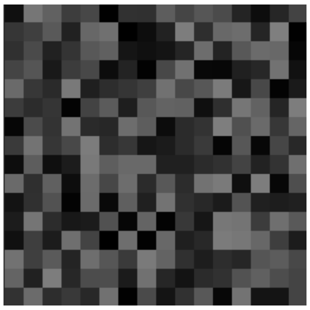

FIG. 2.11 – Grille bidimensionnelle de valeurs aléatoires.

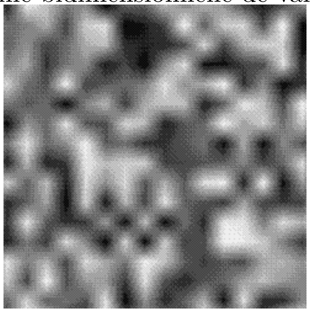

FIG. 2.12 – L'interpolation linéaire de la fonction de la figure 2.11 donne le bruit pseudo-périodique P.

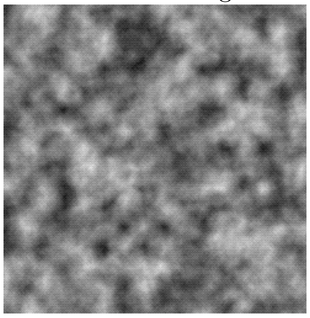

FIG. 2.13 – Quatre octaves du bruit pseudo-périodique P de la figure 2.12 donnent le bruit turbulent B.

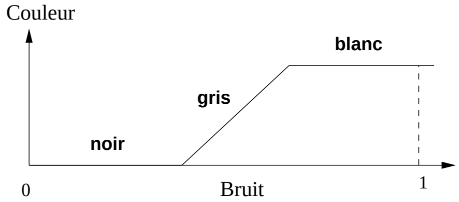

FIG. 2.14 – Colormap utilisée pour l'agate de la figure 2.3.

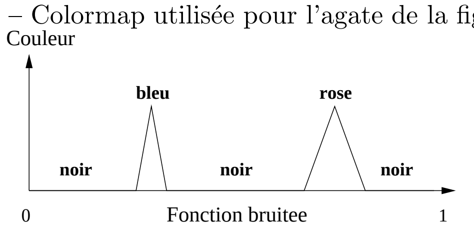

FIG. 2.15 – Colormap utilisée pour le marbre des figures 2.5 et 2.7.

C'est la colormap qui fait apparaître les veines du marbre, les anneaux du bois et les grains de l'agate en privilégiant certaines lignes de niveau par rapport à un *fond* (fig. 2.14 et fig. 2.15).

Comme l'ensemble des couleurs, souvent assimilé à $[0;1]^3$, est borné, il est pratique de définir la colormap comme une fonction sur $[0;1]$ prolongée par périodicité sur $\mathbb{R}$.

En ce qui concerne la modélisation, la colormap est souvent définie par l'interpolation de points de contrôle judicieusement choisis. Quelques points interpolés linéairement suffisent à donner de bons résultats.

2.2.4 Avantages

Le résultat obtenu est une texture volumique très réaliste simulant le bois, le marbre ou l'agate.

Un des avantages des textures procédurales est d'être défini en tout point de l'espace sans artifice d'interpolation ou de périodicité et sans consommation de mémoire. Comme le bruit turbulent calculé est homogène et isotrope, la texture a un aspect semblable en tout point sans pour autant posséder de répétition apparente.

Les textures procédurales sont particulièrement utilisées dans le rendu par tracé de rayons. Contrairement aux algorithmes basés sur le rendu de facettes, le tracé de rayons permet le rendu direct de nombreuses primitives (sphères, cônes, surfaces implicites, etc.) pour lesquelles un plaquage de texture n'est pas forcément possible ou induit des déformations inacceptables.

Les paramètres qui définissent une texture procédurale sont peu nombreux ; les équations précédentes ne font intervenir qu'une pseudo-période s et une amplitude α du bruit turbulent ainsi que le choix d'une colormap. Alors que les paramètres définissant une texture plaquée (le couleur de chaque pixel de la texture étant un paramètre indépendant) sont locaux, les paramètres définissant une texture procédurale sont macroscopiques. On perd le contrôle sur chaque détail du matériau mais il devient possible de modifier son aspect général (comme la taille, la turbulence ou la couleur d'une veine).

2.2.5 Inconvénients

Les calculs nécessaires à générer une texture procédurale sont plus compliqués que ceux nécessaires à un simple plaquage de texture sur une facette. Les textures procédurales sont donc implémentées en logiciel et utilisées fréquemment pour le rendu de scènes de haute qualité avec un algorithme de tracé de rayons. Par contre, aucune implémentation en matériel existe, ce qui interdit tout utilisation dans les applications temps réel.

L'absence de rendu en temps réel de textures procédurales nuit également au rendu de haute qualité. Il est en effet impossible de changer interactivement les paramètres de la texture lors du modelage d'une scène. Ceci est très gênant car la modification d'un paramètre a souvent un effet global pas toujours prévisible. La mise au point de l'aspect d'un objet demande donc du temps.

Enfin, les textures procédurales ne sont aptes à décrire que quelques familles particulières de matériaux. Nous avons donné l'exemple de l'agate, du marbre et du bois ; ce sont les plus utilisés. L'approche procédurale de Perlin ne permet pas, par exemple, de simuler des motifs organiques (texture du foie, peau d'une girafe, etc.). De même, les artefacts humains (pierres taillés, planches de bois découpées en rectangle, etc.) sont difficilement intégrables à un matériau procédural de type Perlin.

Il existe cependant d'autres modèles procéduraux mieux adaptés à ces cas. Par exemple, les *textures cellulaires* proposées par Worley ([Wor96]) permettent de décrire des minéraux en cristaux et des peaux à écaille. Un autre exemple est celui des textures par *réaction-diffusion* de Witkim et Kass ([WK91]), permettant de générer les rayures d'un chat ou les tâches d'un léopard .

Chapitre 3

Réalisation de textures procédurales en temps réel[1]

L'objet du stage est de concevoir une méthode d'application de textures procédurales sur des scènes en temps réel. Pour cela nous utiliserons la bibliothèque de rendu OpenGL® 1.1 de Silicon Graphics. OpenGL® exploite au maximum le matériel graphique présent sur l'ordinateur. OpenGL® a aussi l'avantage d'être très portable : cette bibliothèque est implémentée sur la plus part des systèmes d'exploitation (Irix, Linux, Windows, etc.) et supporte un large choix de matériel graphique dédié à la 3D temps réel (des stations graphiques hautes performances Silicon Graphics aux cartes accélératrices 3D pour les jeux sur les PC).

Par respect pour les spécification des matériels graphiques actuellement disponibles, OpenGL® supporte les textures plaquées, mais pas les textures procédurales. Pourtant OpenGL® permet d'effectuer de nombreuses opérations arithmétiques en parallèle sur tous les pixels de l'écran. Nous utiliserons des *extensions* d'OpenGL® disponibles sur toutes les stations Silicon Graphics qui étendent encore le nombre de ces opérations. Il est alors possible de combiner astucieusement ces opérations pour calculer en temps réel une texture procédurale de Perlin[2].

Le stage aboutit à la programmation d'une petite bibliothèque de fonctions nommée GLP permettant d'utiliser des textures procédurales dans le rendu de scènes avec OpenGL® (voir annexe B). Le stage aboutit également à la programmation d'une application utilisant cette bibliothèque pour visualiser un modèle habillé d'une texture procédurale et changer interactivement les paramètres de la texture.

L'intérêt est donc double : d'une part améliorer la qualité visuelle des applications temps réel, d'autre part autoriser la modification interactive des paramètres d'une texture dans un modeleur, avant un rendu de plus haute qualité.

3.1 Rendu avec OpenGL®

3.1.1 Rendu de triangles

La primitive de base d'OpenGL® est le *triangle* : tous les objets doivent être décomposés en triangles, qui sont passés un par un à la bibliothèque. Les triangles sont *projetés* sur l'écran puis *rasterisés*. Le tableau de pixels contenant l'image rendue sur l'écran est appelé *framebuffer* ; un autre tableau contient les pixels de la texture à plaquer, c'est la *mémoire de texture*. Tous les pixels ont quatre composantes : rouge, vert, bleu et alpha ; chacune prend des valeurs dans $[0; 1]$. Rouge, vert et bleu déterminent la couleur du pixel ; la composante alpha n'est pas visible mais sert à différents effets (en particulier la transparence). Un dernier tableau contient un *z-buffer* pour les tests d'occlusion.

Chacun des trois sommets d'un triangle est déterminés par ses coordonnées (x, y, z) dans l'espace, mais aussi par une couleur (r, g, b, a) et par des coordonnées de texture (s, t, r, q). Bien que quatre coordonnées

[1]Voir l'annexe A pour le détail de l'implémentation en OpenGL®.

[2]Il est même imaginable de combiner ces opérations pour effectuer toutes sortes de calculs en parallèle sur un grand nombre de données. Le matériel graphique est alors utilisé comme super-calculateur !

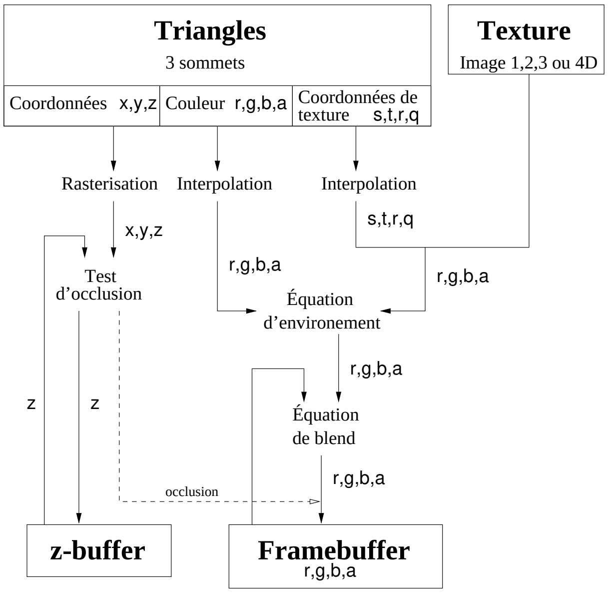

FIG. 3.1 – Schéma simplifié du rendu des triangles sous OpenGL®.

de texture soient disponibles, la plupart des implémentations d'OpenGL® ne supportent que les textures uni- ou bidimensionnelles qui suffisent à la plupart des applications ; les coordonnées de texture superflues sont alors simplement ignorées. Nous utiliserons par la suite des textures tridimensionnelles qui sont une extension d'OpenGL® disponible sur toutes les stations Silicon Graphics.

Le rendu d'un triangle sous OpenGL® se fait en plusieurs passes très paramétrables ; nous simplifierons ici au maximum l'exposé des fonctionnalités (fig. 3.1). Après quelques transformations géométriques du triangle (rotation du point de vue, projection sur l'écran, découpage des parties du triangle qui sortent de l'écran), le triangle est *rasterisé* pour générer des pixels. On obtient ainsi la position que doit occuper dans le framebuffer chaque pixel du triangle, on obtient aussi sa profondeur pour le test d'occlusion. La couleur (r, g, b, a) et les coordonnées de texture (s, t, r, q) d'un pixel du triangle sont déterminées par interpolation des valeurs spécifiées aux sommets.

Les coordonnées de texture du pixel permettent d'obtenir une couleur (r, g, b, a) à partir d'un tableau uni-, bi- ou tridimensionnel de pixels stocké dans la mémoire de texture. Cette opération étant très paramétrables, nous ne rentrerons pas dans les détails. Il suffit de dire que la texture est automatiquement prolongée par périodicité et qu'une interpolation linéaire des couleurs du tableau est possible.

Deux équations paramétrables, *l'équation d'environnement* et *l'équation de blend* combinent composante par composante la couleur obtenue par interpolation des couleurs des sommets et la couleur issue de la texture, puis cette couleur avec la couleur du pixel framebuffer que l'on doit remplacer[3]. Si le test d'occlusion du pixel réussi, le pixel du framebuffer est effectivement remplacé par la couleur calculée précédemment, sinon le pixel est ignoré.

3.1.2 Opérations dans le framebuffer

OpenGL® est aussi capable d'exécuter des opérations directement sur les pixels du framebuffer[4]. Chaque pixel impliqué est lu, un certain nombre de transformations sont appliquées à ses composantes, puis le pixel est réécrit dans le framebuffer. Les opérations disponibles sont variées et certaines dépendent

[3]L'équation d'environnement sert généralement à moduler la texture de l'objet par la couleur et l'intensité de l'éclairage : l'éclairage est calculé rigoureusement en chaque sommet d'un triangle et passé à OpenGL® qui l'interpole pour obtenir l'éclairage en chaque point du triangle. L'équation de blend sert à simuler des objets transparents. C'est dans ces deux équations que la composante alpha sert.

[4]Ces opérations peuvent aussi être effectuées dans d'autres tableaux de pixels gérés par OpenGL®, en particulier dans la mémoire de texture et dans les *auxbuffers* (semblables au framebuffer mais n'apparaissant pas à l'écran). Il est même possible de choisir, pour source et pour destination des opérations, des tableaux de pixels différents.

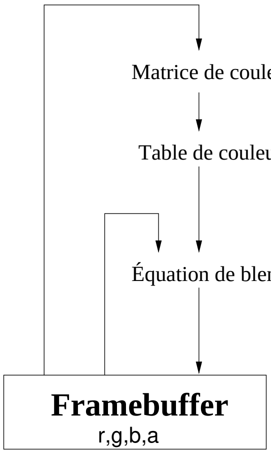

FIG. 3.2 – Schéma simplifié des opérations OpenGL® sur les pixels.

des extensions OpenGL® disponibles sur une machine particulière, c'est pourquoi nous nous limiterons à celles qui nous seront utiles par la suite et qui sont disponibles sur toutes les stations Silicon Graphics.

Les opérations successives qui composent une seule passe sont, dans l'ordre d'exécution (fig. 3.2) :

- $\begin{pmatrix} r \\ g \\ b \\ a \end{pmatrix} = M \begin{pmatrix} r \\ g \\ b \\ a \end{pmatrix}$ où M est une matrice 4×4 appelée *matrice de couleurs* ;
- $c = C_c(c)$ où c est r, g, b ou a et $C_c : [0;1] \to [0;1]$ sont quatre applications définies par des tableaux unidimensionnels à valeurs dans $[0;1]$ passés à OpenGL® ; l'ensemble de ces quatre applications forme une *table de couleurs* (ou, en anglais, *color looku-up table*, d'abréviation *color LUT*) ;
- les composantes (r, g, b, a) sont combinées avec celles du pixel déjà présent dans framebuffer qui va être écrasé par une équations de blend ; les mêmes équations que dans le cas des triangles sont disponibles ;

A chaque étape, les composantes (r, g, b, a) sont ramenées dans l'intervalle $[0;1]$ par la fonction g suivante, appelée *fonction de clamping* :

$$g(c) = \begin{cases} 0 & \text{si } c < 0 \\ c & \text{si } 0 \le c \le 1 \\ 1 & \text{si } c > 1 \end{cases}$$

La matrice de couleurs et la table de couleurs qui la suit sont des extensions d'OpenGL® présentes sur toutes les stations Silicon Graphics. L'utilisation de l'extension *texture tridimensionnelle* n'est pas nécessaire dans l'algorithme de textures procédurales proposé dans ce stage (voir l'annexe A). Par contre la présence de des extensions *matrice de couleurs* et *table de couleurs* sont indispensables.

3.1.3 Rendu multi-passes

Le rendu de triangles sous OpenGL® offre beaucoup de possibilités de modulation : il est par exemple possible de dessiner en une seule fois un triangle texturé et illuminé grâce à l'équation d'environnement. Toutefois, on peut vouloir utiliser une image codant l'intensité lumineuse en chaque point du triangle pour décrire plus finement l'illumination. Il faut alors plaquer sur le triangle deux textures indépendantes : une texture d'illumination, variant avec l'éclairage, qui module une texture décrivant la couleur de l'objet. Il n'est possible de définir deux textures par triangle que sur certains matériels ; il s'agit d'une extension d'OpenGL® très peu supportée (elle n'est, en particulier, pas supportée sur les stations Silicon Graphics, utilisées pendant le stage).

La solution consiste à dessiner plusieurs fois le même triangle avec des textures et des paramètres différents, et à utiliser l'équation de blend pour définir comment se combinent les composantes issues des diverses passes. Dans l'exemple précédent, il suffira de rendre la scène avec la texture de matériau sur les triangles, puis de rendre une seconde fois la scène avec la texture d'illumination en utilisant une équation de blend multiplicative. Il est également possible de combiner des passes de rendu de triangles et l'application de matrices ou de tables de couleurs.

Nous appellerons *passe tridimensionnelle* le rendu des triangles qui composent une scène (fig. 3.1) et *passe bidimensionnelle* l'application à tous les pixels du framebuffer les opérations décrites dans la section précédente (fig. 3.2). Le côut d'une passe tridimensionnelle dépend de la *complexité* de la scène (c'est à dire du nombre de triangles qui la constituent) pour les calculs géométriques ainsi que de la *résolution* du framebuffer (c'est à dire du nombre de pixels qui le constituent) pour l'étape de rasterisation. Le côut d'une passe bidimensionnelle dépend uniquement de la résolution du framebuffer.

La méthode de calcul de textures procédurales de Perlin en temps réel que je présente ici fait intensivement appel au rendu multi-passes, alternant passes tridimensionnelles et bidimensionnelles.

3.2 Génération du bruit

Dans toute la suite, la scène à texturer consiste en un ensemble de triangles ne contenant pas d'information de couleur ni de texture à leurs sommets.

Rappelons que le bruit turbulent B est généré par la superposition de n bruits P pseudo-périodiques obtenus par interpolation de valeurs aléatoires placées sur une grille. OpenGL® possède un filtre de grossissement : si un pixel de la texture se projette sur plusieurs pixels du le framebuffer, il est possible d'obtenir une interpolation des pixels de la texture (fig. 2.12) au lieu d'une couleur unique pour tous ces pixels (fig. 2.11), ce qui donnerait un effet de *mosaïque*. Pour créer P, la mémoire de texture est chargée avec un tableau tridimensionnel de valeurs aléatoires de $[0;1]$. Cette texture est ensuite appliquée avec un filtre de grossissement sur la scène. Les coordonnées de texture (s,t,r) en un sommet sont simplement les coordonnées (x,y,z) de ce sommet.

Pour construire B, n passes sont effectuées de cette manière. A la passe i, les coordonnées de texture sont multipliées par 2^i pour obtenir la croissance en fréquence des octaves. En utilisant une couleur unique $(\frac{\alpha}{2^i}, \frac{\alpha}{2^i}, \frac{\alpha}{2^i}, 0)$ pour tous les sommets ainsi qu'une équation d'environnement multiplicative, on obtient la décroissance en amplitude des octaves. Enfin, une équation de blend additive est utilisée pour combiner les composantes de chaque passe.

Pseudo-code

Voici, résumé en pseudo-code, l'algorithme que je propose :

- ◇ `Charger la mémoire de texture avec un tableau tridimensionnel de valeurs aléatoires`
- ◇ `Choisir un filtre de grossissement linéaire`
- ◇ `Choisir comme coordonnées de texture de chaque sommet de la scène ses coordonnées` (x,y,z)
- ◇ `Faire de` $(\alpha,\alpha,\alpha,0)$ `la couleur courante`
- ◇ `Rendre la scène`
- ◇ `Choisir une équation de blend additive`
- ◇ `Pour i de 1 à n-1`
  - ◇ `Faire de` $(\frac{\alpha}{2^i}, \frac{\alpha}{2^i}, \frac{\alpha}{2^i}, 0)$ `la couleur courante`
  - ◇ `Multiplier par 2 les coordonnées de texture`
  - ◇ `Rendre la scène`

n passes tridimensionnelles sont effectuées. n est de l'ordre de 4.

Résultat dans le framebuffer

Le résultat obtenu est qu'en chaque pixel du framebuffer qui correspond à point de la scène de coordonnées (x, y, z), la couleur est :

$$\begin{cases} r = \alpha B_r(x, y, z) \\ g = \alpha B_g(x, y, z) \\ b = \alpha B_b(x, y, z) \\ a = 0 \end{cases}$$

La figure 2.13 est obtenue par une variante bidimensionnelle et en luminance de cet algorithme (c'est à dire avec $r = g = b = a = \alpha B_r(x, y, z) = \alpha B_g(x, y, z) = \alpha B_b(x, y, z) = \alpha B(x, y)$).

L'utilisation d'OpenGL® impose $0 \leq \alpha \leq 1$[5]. Le bruit turbulent a lui même des valeurs comprises dans $[0; 2\alpha] \cap [0; 1]$ et est périodique. Il est possible de briser cette périodicité en appliquant à chaque passe i une rotation du bruit P : la superposition B de ces fonctions apparaît non périodique bien que chacune de ses fréquences le soit.

3.3 Calcul de la fonction caractéristique

Le bruit turbulent en luminance $r = g = b = a = \alpha B(x, y, z)$ obtenu précédemment permet, avec une colormap appropriée, d'obtenir un matériau qui a l'aspect de l'agate. Pour obtenir du marbre ou du bois, il est nécessaire de calculer la fonction caractéristique f correspondante en $(x + \alpha B_x(x, y, z), y + \alpha B_y(x, y, z), z + \alpha B_z(x, y, z))$.

Puisque nous avons déjà $(\alpha B_r(x, y, z), \alpha B_g(x, y, z), \alpha B_b(x, y, z), 0)$ dans le framebuffer, il suffit d'ajouter à chaque pixel correspondant à un point (x, y, z) de la scène la couleur $(x, y, z, 0)$. Il est ensuite nécessaire de traduire l'application de la fonction f en passes OpenGL® bidimensionnelles pour obtenir en chaque pixel du framebuffer :

$$r = g = b = a = f(x + \alpha B_r(x, y, z), y + \alpha B_g(x, y, z), z + \alpha B_b(x, y, z))$$

3.3.1 Fonction identité

Pour ajouter à chaque pixel du framebuffer, de coordonnées (x, y, z) dans la scène, la couleur $(x, y, z, 0)$, nous chargeons dans la mémoire de texture la texture volumique définie par $(r, g, b, a) = (Id(x), Id(y), Id(z), 0)$, où Id est la fonction identité. Puis nous effectuons une passe de rendu tridimensionnel avec une équation de blend additive et des coordonnées de textures $(u, v, w) = (x, y, z)$ en chaque sommet.

Malheureusement, une texture sous OpenGL® ne peut être définie que comme une fonction périodique, de période 1, à valeurs dans $[0; 1]^4$. De même, le framebuffer ne peut contenir que des couleurs dont les composantes prennent leurs valeurs dans $[0; 1]$.

Une solution est d'utiliser, à la place de la fonction identité Id, la restriction I_r de Id sur $[0; 1]$, automatiquement prolongée par périodicité sur $\mathbb{R}$ par OpenGL® (fig. 3.3).

Le résultat dans le framebuffer est :

$$\begin{cases} r = \alpha B_r(x, y, z) + \beta(x \; mod \; 1) \\ g = \alpha B_g(x, y, z) + \beta(y \; mod \; 1) \\ b = \alpha B_b(x, y, z) + \beta(z \; mod \; 1) \\ a = 0 \end{cases}$$

Un coefficient β permet de rester dans l'intervalle $[0; 1]$ lors de l'addition de I_r. Il faut, pour cela, avoir $\alpha + \beta \leq 1$.

Une autre fonction utilisable est la fonction triangulaire I_t (fig. 3.4). Celle-ci a, en particulier, l'avantage de correspondre à la fonction $|2x|$ sur $[-\frac{1}{2}; \frac{1}{2}]$. Cette propriété sera particulièrement importante pour construire un texture de bois.

[5] α est utilisée comme couleur au sommet des triangles et OpenGL® applique à chaque composante la fonction de clamp pour la ramener dans $[0; 1]$.

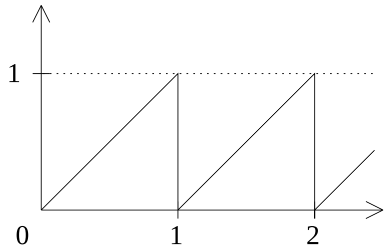

FIG. 3.3 – Prolongement périodique I_r de la fonction identité restreinte à $[0; 1]$.

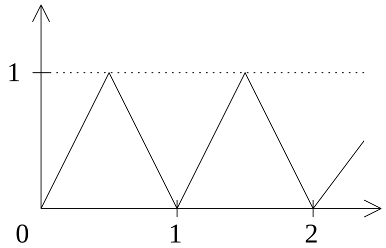

FIG. 3.4 – Fonction en *triangulaire* I_t qui a l'avantage d'être continue et de correspondre à $|2x|$ sur $[-\frac{1}{2}; \frac{1}{2}]$.

Résultat dans le framebuffer

La fonction I choisie étant I_r ou I_t, le résultat dans le framebuffer est alors :

$$\begin{cases} r = \alpha B_r(x, y, z) + \beta I(x) \\ g = \alpha B_g(x, y, z) + \beta I(y) \\ b = \alpha B_b(x, y, z) + \beta I(z) \\ a = 0 \end{cases}$$

Comme pour α, il est nécessaire d'avoir $0 \leq \beta \leq 1$.

3.3.2 Texture de marbre

Pour obtenir un marbre, il nous faut dans le framebuffer :

$$r = g = b = a = \alpha B(x, y, z) + \beta I(x)$$

Après le calcul du bruit et l'ajout de la fonction identité, la composante rouge de chaque pixel du framebuffer contient $\alpha B_r(x, y, z) + \beta I(x)$. Il suffit donc de recopier dans les composantes verte, bleue et alpha de chaque pixel, la valeur de la composante rouge de ce pixel. Cette étape se fait avec une passe bidimensionnelle spécifiant une matrice de couleur :

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Toutefois, il y a plus simple et plus efficace.

Un bruit turbulent en *luminance* remplit le framebuffer avec $r = g = b = a = \alpha B(x, y, z)$. Il suffit d'ajouter à chaque pixel, grâce à une texture unidimensionnelle $r = g = b = a = I(x)$, la couleur $(r, g, b, a) = (I(x), I(x), I(x), I(x))$ modulée par un facteur β.

On évite ainsi l'application de la matrice de couleurs, et on remplace une texture tridimensionnelle par une texture unidimensionnelle (qui prend moins de place mémoire et est parfois plus rapide).

Pseudo-code

Voici le résumé, en pseudo-code, du deuxième algorithme :

```
◇ Calculer un bruit turbulent en luminance dans le framebuffer
  ◇ Choisir une fonction de blend additive
  ◇ Charger la mémoire de texture avec la fonction I
  ◇ Faire de (β,β,β,0) la couleur courante
◇ Rendre la scène
  ◇ Charger la table de couleurs avec la colormap C
◇ Effectuer une passe bidimensionnelle
```

En plus des n passes tridimensionnelles effectuées pour calculer le bruit turbulent, une passe tridimensionnelle et une passe bidimensionnelle sont nécessaire.

Choix de la fonction I

Il est important d'avoir une texture d'aspect continu.

La solution la plus simple est d'utiliser la fonction I_t qui est continue sur $\mathbb{R}$. Cependant, comme I_t croît et décroît, le résultat est un marbre dont les veines apparaissent inversées une fois sur deux, ce qui n'est pas forcément souhaitable.

Il est aussi possible d'utiliser la fonction I_r si on prend garde à se placer dans une partie de l'espace où $(I_r(x), I_r(y), I_r(z))$ est continue (le cube $[0;1]^3$ par exemple). En multipliant les coordonnées de texture (s,t,r) par un facteur adéquat, l'objet apparaît, quelle que soit sa taille, taillé dans un bloc où la texture est continue.

Toutefois ceci n'est pas praticable pour des gros objets vus de près. A cause de la précision limitée des calculs dans le framebuffer, des artefacts apparaissent quand on agrandit trop à l'intérieur d'une période de I_t ou I_r. Il est alors nécessaire d'utiliser plusieurs périodes de I_r ou de I_t.

En prenant quelques précautions, nous pouvons utiliser la fonction I_r sur plusieurs périodes sans discontinuité apparente.

Il faut s'assurer que $C_c(\alpha B(x,y,z) + \beta I_t(1^-)) = C_(\alpha B(x,y,z) + \beta I_t(1^+))$, c'est à dire $C_c(x+\beta) = C_c(x) \quad \forall x \in [0;\alpha]$. Ceci est possible avec, par exemple, $\alpha = \frac{1}{3}$, $\beta = \frac{2}{3}$ et une colormap C qui se répète trois fois dans l'intervalle $[0;1]$.

3.3.3 Texture de bois

Pour fabriquer du bois, nous devons obtenir dans le framebuffer :

$$r = g = b = a = \sqrt{\frac{1}{2}(\alpha B_x(x,y,z) + \beta I(x))^2 + \frac{1}{2}(\alpha B_y(x,y,z) + \beta I(y))^2}$$

Les coefficients $\frac{1}{2}$ garantissent que l'on reste toujours dans l'intervalle $[0;1]$.

Nous partons d'un bruit turbulent ajouté à une fonction identité. Une passe bidimensionnelle avec une équation de blend multiplicative permet de remplacer chaque pixel du framebuffer (r,g,b,a) par (r^2, g^2, b^2, a^2). Une deuxième passe bidimensionnelle avec une matrice de couleurs :

$$\frac{1}{2}\begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

permet de sommer les composantes rouges et vertes contenant respectivement $(\alpha B_r(x,y,z) + \beta I(x))^2$ et $(\alpha B_g(x,y,z) + \beta I(y))^2$:

$$r = g = b = a = \frac{1}{2}(\alpha B_x(x,y,z) + \beta I(x))^2 + \frac{1}{2}(\alpha B_y(x,y,z) + \beta I(y))^2$$

Pour calculer la racine carrée, la solutiom la plus commode consiste à utiliser une table de couleurs précalculée S_c (c valant r, g, b ou a) :

$$S_c(c) = \sqrt{c}$$

L'application d'une table de couleurs nécessite une passe bidimensionnelle, mais la multiplication par la matrice de couleurs a lieu *avant* la recherche dans la table de couleurs, nous pouvons factoriser ces deux opérations en une seule passe bidimensionnelle.

Nous pouvons même aller plus loin et condenser le calcul de la racine carrée et l'application d'une colormap C en une seule colormap C' en posant :

$$C'(i) = C(\sqrt{i})$$

Ainsi la multiplication par la matrice de couleurs, le calcul de la racine carrée et l'application de la colormap ne nécessitent qu'une unique passe bidimensionnelle.

Pseudo-code

Voici un résumé en pseudo-code de l'algorithme de génération du bois :

- ♢ `Calculer un bruit turbulent dans le framebuffer`
  - ⋄ `Choisir une fonction de blend additive`
  - ⋄ `Charger la mémoire de texture avec la fonction` I
  - ⋄ `Faire de` $(\beta, \beta, \beta, 0)$ `la couleur courante`
- ♢ `Rendre la scène`
  - ⋄ `Choisir une fonction de blend multiplicative`
- ♢ `Effectuer une passe bidimensionnelle`
  - ⋄ `Choisir une fonction de blend qui remplace simplement l'ancien pixel par le nouveau`
  - ⋄ `Choisir une matrice de couleurs :`

$$\frac{1}{2}\begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

  - ⋄ `Charger la table de couleurs avec la colormap corrigée` C'
- ♢ `Effectuer une passe bidimensionnelle`

En plus des n passes tridimensionnelles dues au calcul du bruit turbulent, une passe tridimensionnelle et deux passes bidimensionnelles sont effectuées.

Choix de la fonction I

Pour obtenir des lignes de niveau circulaires au voisinage de $(0, 0)$, il faut avoir $I(x)^2 + I(y)^2 = x^2 + y^2$.

Comme $|I_r(-x)| = 1 - x \neq x = |I_r(x)|$ sur $[0; 1]$, I_r ne permet pas d'obtenir des lignes de niveau circulaires, mais seulement des quarts de cercles (fig. 3.5). Il est donc impossible d'utiliser I_r pour construire une texture de bois.

Par contre, on a bien $|I_t(-x)| = 2x = |I_t(x)|$ sur $[0; \frac{1}{2}]$. Les lignes de niveaux obtenues sont des cercles centrés en tous les points de coordonnées entières (fig. 3.6).

3.4 Application d'une colormap

Nous avons maintenant dans le framebuffer :

$$r = g = b = a = f(\alpha B_r(x, y, z) + \beta I(x), \alpha B_g(x, y, z) + \beta I(y), \alpha B_b(x, y, z) + \beta I(z))$$

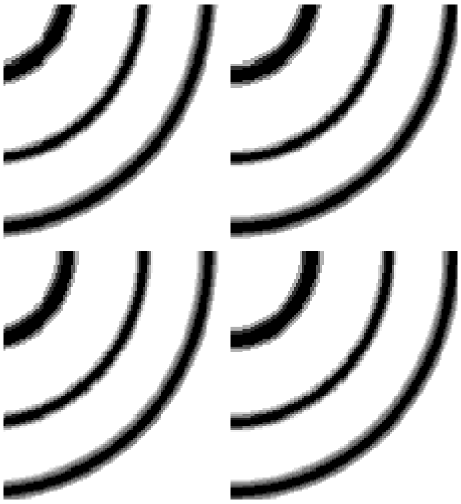

FIG. 3.5 – Lignes de niveau d'une fonction caractéristique de bois utilisant I_r.

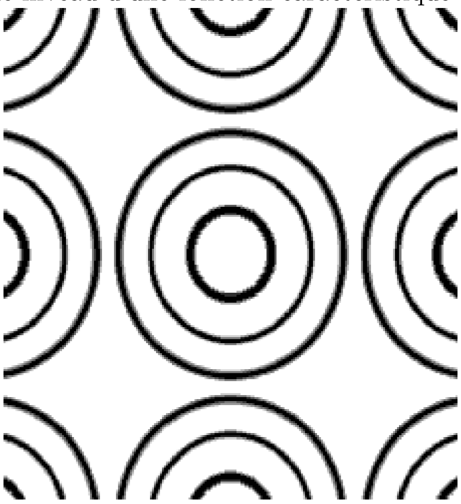

FIG. 3.6 – Lignes de niveau d'une fonction caractéristique de bois utilisant I_t.

ou bien, dans le cas de l'agate :

$$r = g = b = a = \alpha B(x, y, z)$$

Il reste à appliquer la colormap qui associe une couleur à chacune de ces valeurs. Nous utilisons une passe bidimensionnelle avec une table de couleur.

La colormap C de $[0;1]$ à valeurs dans l'ensemble de couleurs $[0;1]^4$ est décomposé en quatre fonctions, une pour chaque composante, qui, discrétisées dans quatre tableaux, définissent la table de couleurs (C_r, C_g, C_b, C_a). Chaque composante est normalement modifiée indépendamment par une table de couleurs mais, comme on a dans le framebuffer $r = g = b = a$, l'effet est le même qu'une fonction associant une couleur à une valeur de $[0;1]$.

Par rapport à la définition précédente d'une colormap, une dimension a été ajoutée pour tenir compte de la composante alpha. La variation d'alpha sur l'objet peut être utilisée pour générer des effets intéressants lors de passes ultérieures, mais la composante alpha n'est pas utilisée pendant le processus même de calcul de la texture procédurale.

Nous avons vu également dans le cas du marbre qu'une colormap doit parfois avoir une période plus petite que 1 (dans ce cas précis, il s'agissait d'une période de $\frac{1}{3}$). Il est donc pratique de définir la colormap par quelques points de contrôle à interpoler dans $[0; \frac{1}{n}]$ et un facteur de répétition n. La colormap est ensuite prolongée automatiquement par périodicité sur $[0;1]$.

Chapitre 4

Application et résultat

J'ai effectivement implémentée, sur les stations Silicon Graphics du laboratoire *i*MAGIS, les techniques de texture procédurales sous OpenGL® décrites précédemment.

Le résultat du stage est composé d'une bibliothèque, nommée GLP (voir annexe B), et d'un programme d'application. La bibliothèque GLP ajoute à OpenGL® les textures procédurales de Perlin de type agate, bois et marbre. Le programme d'application permet de générer une texture procédurale et de changer interactivement ses paramètres.

4.1 Programme d'application

4.1.1 Présentation

Il s'agit d'un programme écrit en C++ et utilisant les bibliothèques OpenGL® et GLP pour le rendu, ainsi que la bibliothèque Motif pour l'interface graphique.
La version a été développée sous Silicon Graphics mais devrait être portable sur n'importe quel système Unix comportant Motif et la bibliothèque OpenGL®.

L'application lit des *modèles* au format standard très simple *.m*. Un modèle est une scène simple composée d'un seul objet. Ces modèles sont composés exclusivement de triangles sans aucune information de couleur ou de texture.
L'application utilise ensuite la bibliothèque GLP pour habiller l'objet d'une texture procédurale. De nombreux paramètres sont modifiables interactivement. Les paramètres définissant la texture résultante peuvent ensuite être sauvés dans un fichier, dans un format texte simple et lisible par l'utilisateur.
La séparation entre objet et texture est donc clairement marquée par la présence de deux types de fichiers, l'un contenant le modèle sans information de texture et l'autre contenant les paramètres de la texture procédurale.

4.1.2 Paramètres

Les paramètres définissant une texture procédurale reflètent ceux proposés par la bibliothèque GLP (voir annexe B).

Voici la liste de ces paramètres :

- le bruit turbulent B peut être bidimensionnel au lieu de tridimensionnel si les textures volumiques ne sont pas supportées ;
- le bruit turbulent B peut consister en trois bruits décorrélés, un pour chaque coordonnée x, y et z, ou être en luminance ;
- le facteur de croissance en fréquence des différents bruits pseudo-périodiques composant le bruit turbulent peut être réglé (les équations précédentes utilisaient un facteur 2) ;
- le facteur de décroissance en amplitude des différents bruits pseudo-périodiques composant le bruit turbulent peut être réglé (les équations précédentes utilisaient un facteur 2) ;

- la fréquence de base du bruit turbulent est réglable sur les trois axes, de façon à générer une texture non isotrope ;
- les amplitudes α et β du bruit turbulent et de la fonction I sont paramétrables ;
- plusieurs fonctions I sont utilisables :
  - fonction identité discontinue *ramp* I_r,
  - fonction *triangulaire continue* I_t,
  - fonction *sinusoïdale*[1] $I_s(c) = \frac{1}{2}(1 - \cos 2\pi c)$;
- le nombre de bruits pseudo-périodiques composant le bruit turbulent (appelé aussi *nombre d'octaves*) est réglable ;
- une colormap est spécifiée par quelques points de contrôles qui sont interpolés linéairement, et un facteur de répétition ;
- l'application de la colormap fait également intervenir un facteur γ généralisant l'application d'une racine carrée dans le cas d'une fonction de type bois : la colormap réellement utilisée est $C'(c) = C(c^\gamma)$ où C est la colormap spécifiée par les points de contrôles et le facteur de répétition ;
- plusieurs fonctions caractéristiques sont utilisables :
  - fonction *gradient* $f(x, y, z) = x$ simulant le marbre,
  - fonction *cylindre* $f(x, y, z) = x^2 + y^2$ simulant le bois (en posant également $\gamma = \frac{1}{2}$),
  - fonction *sphère* $f(x, y, z) = x^2 + y^2 + z^2$ qui peut être utilisée pour générer un motif en *léopard* (avec toujours la correction apportée par $\gamma = \frac{1}{2}$).

Les tailles du tableau de valeurs aléatoires et de ceux définissant la fonction I ainsi que la colormap valent respectivement, par défaut : $16 \times 16 \times 16$, 256 et 256 mais peuvent être changés dans la bibliothèque GLP. L'application utilise les valeurs par défaut.

4.1.3 Interface

Édition des paramètres

L'interface *Motif* (fig. 4.1) est mise à profit pour créer des boutons et des curseurs qui simplifient l'utilisation de l'application. Dès qu'un paramètre est changé, le modèle texturé et illuminé est mis à jour.

Cette interface Motif a été créée en C++ avec l'aide du logiciel *RapidApp* de Silicon Graphics.

La colormap est éditée grâce à la souris : l'utilisateur peut créer, déplacer, supprimer des points de contrôle. La colormap, obtenue par interpolation linéaire de ces points de contrôle, apparaît directement dans cette fenêtre (fig. 4.2, les points de contrôle apparaîssent sous forme de petits carrés blancs).

Interface tridimensionnelle

En plus de tous les paramètres précédents, il est possible de déplacer et de tourner le modèle avec six degrés de liberté (position, orientation, taille) en utilisant la souris.

Il est également possible de changer la position, l'orientation et l'échelle de la texture procédurale par rapport au modèle. Il est, en effet, très important de pouvoir placer une veine d'un marbre à un endroit précis du modèle ou d'orienter le motif d'un bois. Ainsi on reprend le contrôle sur la partie de la texture procédurale qui affleure à la surface de l'objet, ce qui était un problème avec les textures volumiques.

La matrice 4×4 définissant la position relative du modèle et de la texture fait partie intégrante de la définition de la texture, elle est sauvegardée dans le fichier de texture.

Fichiers de texture

Les paramètres définissant une texture (y compris la matrice 4×4 et la colormap) sont sauvegardés et lus dans un format texte, qu'il est facile de lire et même d'écrire directement.

Voici, par exemple, le fichier définissant le marbre rose utilisés dans plusieurs figures (par exemple, dans la figure 4.4) :

[1] La fonction I_s a l'avantage d'être $\mathcal{C}^\infty$ mais n'est pas très utilisée en pratique.

```
colormap {
   pts 6
       6 points de contrôle définissent la colormap
       le format est  p r g b a  où (r, g, b, a) est la couleur en p
      0.000000   0.000000 0.000000 0.000000 1.000000
      0.053704   0.800000 0.200000 0.200000 1.000000
      0.475926   1.000000 0.500000 0.500000 1.000000
      0.600000   1.000000 1.000000 1.000000 1.000000
      0.914815   0.600000 0.000000 0.000000 1.000000
      1.000000   0.000000 0.000000 0.000000 1.000000
  repeat 4          facteur de répétition
  gamma 1.000000    facteur de correction γ
}

pattern {
  type gradient     fonction caractéristique du marbre
  func ramp         fonction I_r
  amplitude 0.740000    β

  matrix   matrice
      0.213925 -0.002062 -0.000517  0.000000
     -0.002717  0.204226 -0.007168  0.000000
     -0.001914 -0.020202  0.241150  0.000000
     0 .000000  0.000000  0.000000  1.000000
}

noise {
  octave 4
  freq_mult 2.000000
  amp_div 2.000000
  amplitude 0.200000    α
  dimension 3    bruit tridimensionnel
  component l    bruit en luminance
  frequency  1.000000 1.000000 1.000000
}
```

4.2 Vitesse et qualité atteintes

4.2.1 Images

Les figures 4.3 à 4.8 sont des copies d'écran de quelques objets texturés par le programme d'application. Les modèles *.m* proviennent de Marc Levoy de l'Université de Stanford[2] (pour le lapin) et de Greg Turk de l'Université de Caroline du Nord (pour la surface minimale). Toutes les textures procédurales ont été construites interactivement sur les modèles grâce au programme d'application.

Remarquez comme il est possible, à partir seulement de trois modèles procéduraux (agate, marbre et bois), de générer des textures variées en modifiant quelques paramètres et en choisissant une colormap appropriée.

4.2.2 Vitesse

L'implémentation des textures procédurales en temps réel a été testée sur deux types de stations Silicon Graphics : une station Onyx2 trés haut de gamme, équipée d'une carte graphique InfiniteReality 2, et une station O2 d'entrée de gamme.

[2]http ://www-graphics.stanford.edu

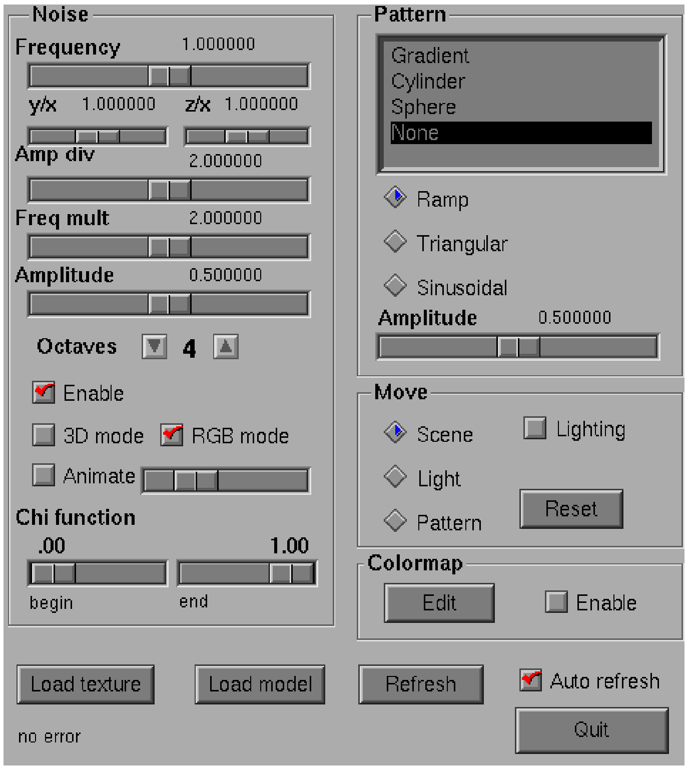

FIG. 4.1 – Interface de l'application.

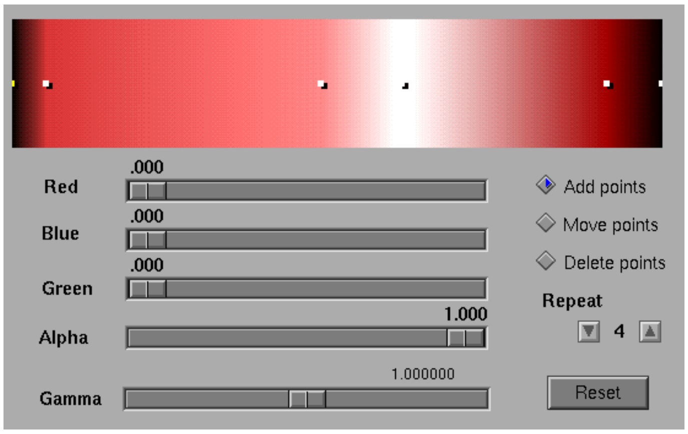

FIG. 4.2 – Exemple de colormap définie dans l'application.

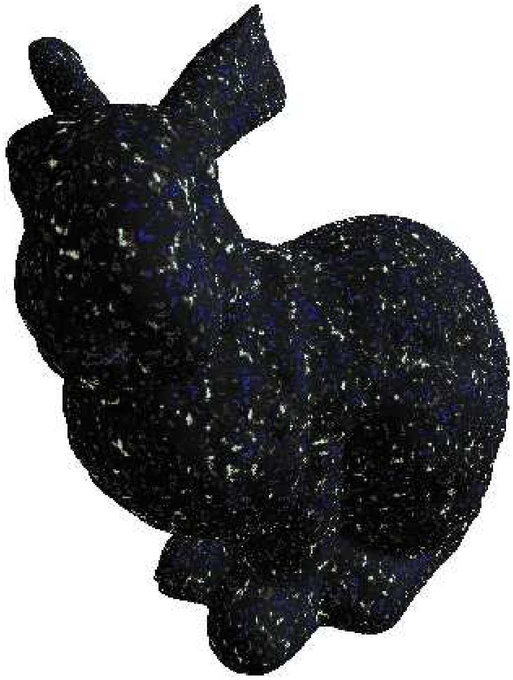

FIG. 4.3 – Lapin habillé d'une texture d'*agate*.

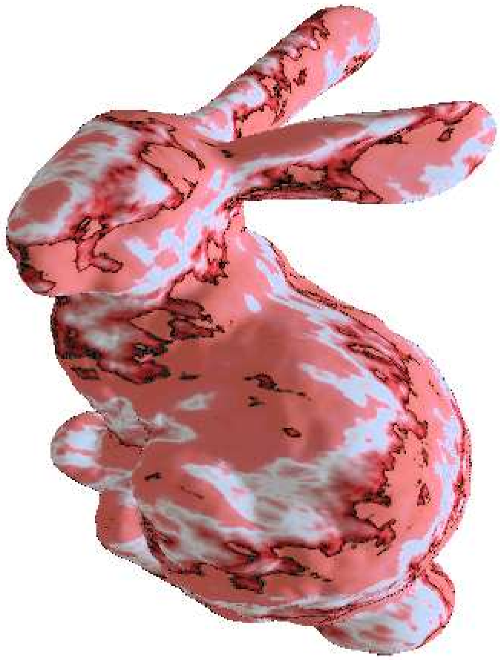

FIG. 4.4 – Lapin habillé d'une texture de *marbre*.

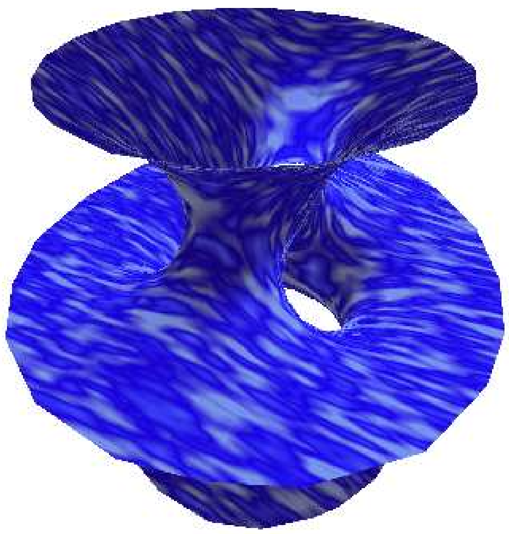

FIG. 4.5 – Une texture *agate* étirée dans une direction donne de l'*eau* sur ce modèle de surface minimale.

FIG. 4.6 – Lapin habillé d'une texture de *bois non perturbé*.

FIG. 4.7 – Lapin habillé d'une texture de *bois* plus réaliste.

FIG. 4.8 – Le *pyjama* du lapin est crée à partir d'une fonction *marbre*.

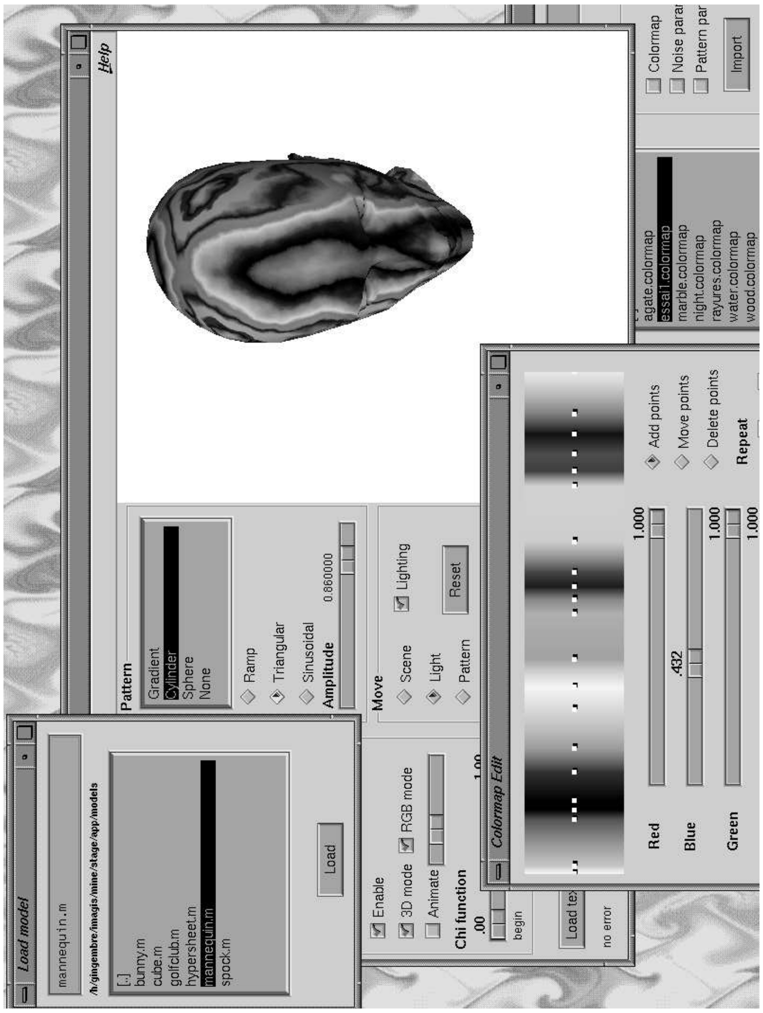

FIG. 4.9 – Capture d'écran lors de l'utilisation de l'application.

Résultats sur une Onyx2 InfiniteReality 2

Grâce à sa carte graphique InfiniteReality 2, l'Onyx 2 supporte, en matériel, les extensions OpenGL® texture tridimensionnelle, table de couleurs et matrice de couleurs.

Pour un modèle constitué de quelques milliers de triangles (la surface minimale de la figure 4.5, par exemple), le taux d'images est de 30 à 40 images par secondes sur une station Onyx2.
Pour le modèle du lapin (figure 4.3, par exemple) contenant 70 000 triangles, l'animation est saccadée mais encore interactive (quelques images par seconde). Un rendu en tracé de rayons d'une telle scène nécessite une dizaine de minutes sur cette même station.

Résultats sur une O2

Les stations O2 sont équipées d'un matériel graphique plus *standard* qui n'implémente pas en matériel ces extensions OpenGL®. La bibliothèque OpenGL® effectue tous les calculs relatifs à ces extensions en logiciel, donc beaucoup plus lentement qu'une Onyx2. A cause de cela, ce n'est pas possible d'animer, sur une O2, un modèle, même simple, si on utilise toute les fonctionalités mentionnées.

Une solution consiste à ne pas utiliser les textures tridimensionnelles. L'annexe A tient compte de ce problème en proposant, à tous les endroits où une texture tridimensionnelle est utilisée, une implémentation ne l'utilisant pas. Il s'en suit que le bruit turbulent calculé n'est plus tridimensionnel, mais bidimensionnel, c'est à dire volumique et constant le long d'un axe.
Il devient alors possible d'animer en temps réel un modèle composé de quelques milliers de triangles sur une O2 (environ 10 images par seconde). Des modèles plus compliqués (comme le lapin) ne sont, par contre, pas envisageables sur ce type de station.

Autres plateformes

Les algorithmes développés au cours du stage nécessitent la présence des extensions OpenGL® table de couleurs et matrice de couleurs, qu'elles soient présentes en matériel (comme sur une Onyx2) ou programmées en logiciel (comme sur une O2)
Ces extensions étant propres à Silicon Graphics, peu d'autres plateformes les reconnaissent actuellement. Cependant, il y a beaucoup de chances pour que ces extensions soient incluses dans le standard de la prochaine version d'OpenGL® en cours de normalisation, ce qui encouragerait les constructeurs de matériel graphique à les supporter.
Il y a bon espoir que ces algorithmes soient utilisables sur de nombreuses plateformes dans un avenir proche.

4.2.3 Qualité

Précision des calculs

Tous les calculs sont limités pas la résolution du framebuffer. Généralement, chaque composante de couleur d'un pixel peut prendre 256 valeurs différentes dans $[0; 1]$. Certains calculs internes sont effectués avec une résolution meilleure tandis que sur certaines cartes accélératrices 3D pour PC, ces calculs sont tronqués pour réduire le nombre de circuits de la carte. Ceci explique, en partie, les différences de prix entre cartes graphiques : de quelques centaines de dollars US pour une carte de basse qualité (prévue pour les jeux) à plusieurs milliers de dollars US pour les cartes graphiques professionnelles.

A cause de cela, il est difficile de quantifier la précision des calculs dans le framebuffer et nous nous appuyons ici sur des critères visuels.
Après un fort grossissement, des artefacts apparaissent dans l'image (fig. 4.10) : le nombre limité des valeurs que peuvent prendre les composantes dans le framebuffer introduit un effet en *marches d'escalier*.

Problèmes de domaine

Une des différences les plus importantes entre une implémentation logicielle des textures procédurales et l'implémentation sous OpenGL® proposée ici tient à la nature des nombres utilisés en interne pour les calculs. Une implémentation logicielle peut utiliser des nombres dits *à virgule flottante* qui sont capables

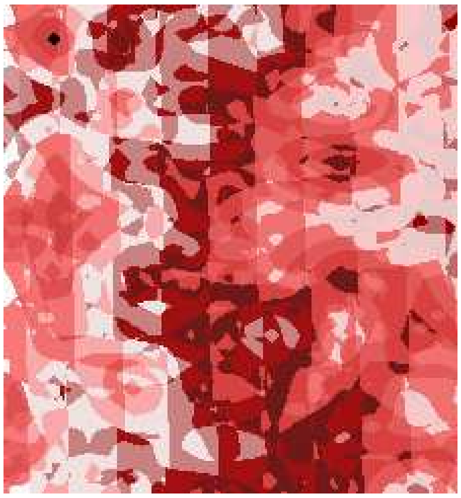

FIG. 4.10 – Artefacts apparaîssant sur un marbre après un fort grossissement.

de représenter avec une bonne précision un grand intervalle de réels. Les calculs dans le framebuffer d'OpenGL® font, par contre, intervenir des nombres *à virgule fixe* qui sont capables de reprśenter avec une précision limitée l'intervalle $[0;1]$. De plus, les textures ne permettent de calculer dans le framebuffer que des fonctions *périodiques*.

Ceci introduit des problèmes de continuité et de périodicité lors du calcul de I. La conséquence est qu'il faut faire très attention pour obtenir une texture continue.

Une des extensions possibles de mon stage est le calcul de *bump mapping* procédural donnant une illusion de relief grâce à une texture perturbant l'illumination en un point.

Ici encore se pose un problème de domaine. Les calculs portent en effet sur des *vecteurs* dont les coordonnées ont pour domaine $[-1;1]$. Il faut donc trouver un moyen pour effectuer ces calculs dans le domaine $[0;1]$ du framebuffer, au prix de passes supplémentaires.

4.2.4 Futur des textures procédurales en temps réel

Toute la philosophie du stage repose sur l'utilisation de commandes OpenGL® génériques pour obtenir des effets particuliers qui ne sont pas directement proposés par OpenGL®.

L'objectif final, dans lequel s'inscrit ce stage, est de faire calculer par OpenGL® une fonction complexe de texture et d'illumination en chaque pixel du framebuffer. Le but est d'obtenir un rendu dont la qualité se rapproche de celle du tracé de rayons, tout en restant temps réel, ce qui représente une accélération d'un facteur environ 5000 !

Il est vraissemblable que dans l'avenir, les matériels graphiques implémenteront directement des textures procédurales, du *bump mapping* et des fonctions d'illuminations locales complexes calculées en chaque pixel. Certains matériels très spécialisés (et très chers) offrent déjà de telles possibilitées, cependant il faudra sûrement plusieurs années avant que cette pratique soit répandue.

Annexe A

Détail de l'implémentation sous OpenGL®

Cette annexe reprend la processus de calcul d'une texture procédurale sous OpenGL® en rentrant dans le détail des appels aux fonctions OpenGL®.

A.1 Rendu multi-passes

A.1.1 Passes tridimensionnelles

Le principe du rendu multi-passes tridimensionnel est de rendre une même scène complètement plusieurs fois de suite. Un problème d'occlusion se pose : il ne faut redessiner à partir de la deuxième passe que les pixels visibles après le test d'occlusion ; les pixels cachés ne doivent pas contribuer à l'équation de blend.

Heureusement le z-buffer permet d'obtenir cet effet. Lors de la première passe, le test d'occlusion classique est employé : l'équation de blend étant désactivée, un pixel de profondeur plus faible que l'actuel pixel du framebuffer le remplace. Les valeurs du z-buffer calculées lors de la première passe ont pour effet de ne laisser passer, lors des passes suivantes, que les pixels de profondeur la plus faible, c'est à dire ceux qui sont visible.

Malheureusement, la manière dont la profondeur des pixels est calculée peut varier lorsque certains paramètres d'OpenGL® changent. Ces erreurs de calcul font qu'un pixel visible peut avoir une profondeur légèrement supérieur lors de la deuxième passe et être ainsi éliminé. Pour résoudre ce problème, la fonction `glPolygonOffsetEXT` permet de *biaiser* la profondeur des pixels. Nous allons donc augmenter légèrement la profondeur des pixels lors de la première passe et interdire l'écriture dans le z-buffer lors des passes suivantes grâce à `glDepthMask` pour ne comparer la profondeur des pixels qu'à la profondeur biaisée.

Voici donc le schéma à suivre :

```
glEnable(DEPTH_TEST);
glDepthFunc(GL_LESS);
glDepthMask(GL_TRUE);
glEnable(GL_POLYGON_OFFSET_EXT);
glPolygonOffsetEXT(1.0,0.0001);
glDisable(GL_BLEND);
```

Dessiner la 1ère passe

```
glDepthMask(GL_FALSE);
glDisable(GL_POLYGON_OFFSET_EXT);
glEnable(GL_BLEND);
```

Spécifier une fonction de blend
Dessiner la $2^{\grave{e}me}$ *passe*

Spécifier une fonction de blend
Dessiner la $3^{\grave{e}me}$ *passe*
...

A.1.2 Passes bidimensionnelles

Nous avons besoin d'appliquer des matrices et des tables de couleurs sur les pixels du framebuffer. Comme ces opérations ne sont possibles que lors d'un transfert de pixels (entre framebuffer, auxbuffers, mémoire de texture et mémoire principale), nous allons copier en bloc tous les pixels du framebuffer sur eux-mêmes[1].

Il faut pour cela mettre à identité la matrice de projection et la matrice *modelview*. Cela a pour effet d'associer au coin inférieur gauche de l'écran les coordonnées $(-1, -1, 0)$ et $(1, 1, 0)$ au coin supérieur droit.

Le test d'occlusion n'ayant pas de signification dans une passe bidimensionnelle nous le supprimons avec `glDisable(GL_DEPTH_TEST)`. De même l'écriture dans le *stencil buffer* n'a aucune signification car elle touche tous les pixels du framebuffer, nous la désactivons donc momentanément grâce à `glStencilOp(GL_KEEP,GL_KEEP,GL_KEEP)`. Tous les paramètres changés sont sauvegardés puis restitués par `glPush/PopAttrib` et `glPush/PopMatrix`.

Il en résulte le programme suivant :

```
GLint vp[4];
glGetIntegerv(GL_VIEWPORT,vp);

glPushAttrib(GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
glDisable(GL_DEPTH_TEST);
glMatrixMode(GL_PROJECTION);
glPushMatrix();
glLoadIdentity();
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glLoadIdentity();
glStencilOp(GL_KEEP,GL_KEEP,GL_KEEP);
glRasterPos2f(-1,-1);

     Spécifier une matrice, des tables de couleurs
     et une fonction de blend
glCopyPixels(0,0,vp[2],vp[3],GL_COLOR);
     Désactiver la matrice et les tables de couleurs

glPopAttrib();
glMatrixMode(GL_PROJECTION);
glPopMatrix();
glMatrixMode(GL_MODELVIEW);
glPopMatrix();
```

A.2 Génération du bruit

La scène à rendre est placée dans une *display list* notée *scene*. Elle ne doit contenir aucune information de couleur ou de coordonnées de texture. Elle peut par contre contenir des informations de normale et de

[1] Il serait imaginable de ne copier qu'un bloc réctangulaire correspondant à une boîte englobante de l'objet pour gagner du temps. Malheureusement, ce n'est pas facile de déterminer cette boîte englobante à partir des triangles composant l'objet et du point de vue et nécessite, dans le cas général, plus de calculs que l'économie.

matériau pour une éventuelle illumination, mais l'illumination est désactivée durant toutes les passes de calcul de la texture procédurale.

Les coordonnées de texture sont générées automatiquement par OpenGL® ; on pose $s = x, t = y, r = z$:

```
glEnable(GL_TEXTURE_GEN_R);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);

glTexGeni(GL_S,GL_TEXTURE_GEN_MODE,GL_OBJECT_LINEAR);
glTexGeni(GL_T,GL_TEXTURE_GEN_MODE,GL_OBJECT_LINEAR);
glTexGeni(GL_R,GL_TEXTURE_GEN_MODE,GL_OBJECT_LINEAR);

GLfloat s[4] =  1.0, 0.0, 0.0, 0.0;
GLfloat t[4] =  0.0, 1.0, 0.0, 0.0;
GLfloat r[4] =  0.0, 0.0, 1.0, 0.0;

glTexGenfv(GL_S,GL_OBJECT_PLANE,s);
glTexGenfv(GL_T,GL_OBJECT_PLANE,t);
glTexGenfv(GL_R,GL_OBJECT_PLANE,r);
```

Il ne faut pas oublier de désactiver la génération automatique de coordonnées de texture dès qu'on en a plus besoin (en particulier si on veut à nouveau spécifier manuellement des coordonnées de texture en chaque sommet) :

```
glDisable(GL_TEXTURE_GEN_R);
glDisable(GL_TEXTURE_GEN_S);
glDisable(GL_TEXTURE_GEN_T);
```

Si une texture bidimensionnelle ou unidimensionnelle est utilisée avec cette génération de coordonnées de texture, l'effet sera exactement le même que si texture était volumique mais constante le long d'un ou de deux axes.

La mémoire de texture est initialisée par un tableau tridimensionnel de valeurs aléatoires :

```
#define max 64
GLfloat tex[max][max][max][3];
for (i=0;i<max;i++)
for (j=0;j<max;j++)
  for (k=0;k<max;k++)
    for (l=0;l<3;l++)
      tex[i][j][k][l] = drand48();

      Les éléments du tableau sont alignés en mémoire sans trou
glPixelStorei(GL_UNPACK_ALIGNMENT,1);

      Interpolation linéaire
glTexParameteri(GL_TEXTURE_3D_EXT, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

      Texture périodique
glTexParameteri(GL_TEXTURE_3D_EXT, GL_TEXTURE_WRAP_S    , GL_REPEAT);
glTexParameteri(GL_TEXTURE_3D_EXT, GL_TEXTURE_WRAP_T    , GL_REPEAT);
glTexParameteri(GL_TEXTURE_3D_EXT, GL_TEXTURE_WRAP_R_EXT, GL_REPEAT);

glTexImage3DEXT(GL_TEXTURE_3D_EXT,0,GL_RGB,
                max,max,max, 0, GL_RGB, GL_FLOAT, tex);
```

Il est aussi possible de définir un bruit en *luminance* avec $r = g = b = a = B(x, y, z)$ en changeant la dernière ligne en :

```
glTexImage3DEXT(GL_TEXTURE_3D_EXT,0,GL_INTENSITY_EXT,
                max,max,max, 0, GL_LUMINANCE, GL_FLOAT, tex);
```

Si les textures tridimensionnelles ne sont pas supportées, on peut toujours définir un bruit bidimensionnel. L'effet obtenu sera une texture volumique constante le long d'un de ses axes :

```
glTexImage2D(GL_TEXTURE_2D,0,GL_RGB,
             max,max, 0, GL_RGB, GL_FLOAT, tex);
```

La matrice de texture est utilisée pour faire croître la fréquence du bruit pseudo-périodique.

Il est également possible d'utiliser la matrice de texture pour changer la fréquence du bruit (grâce à `glScalef`) et déplacer le bruit par rapport à la scène (grâce à `glRotatef` et `glTranslatef`).

La scène est finalement rendue avec une texture volumique de bruit par :

```
glEnable(GL_TEXTURE_3D_EXT);

    Équation d'environnement multiplicative
glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_MODULATE);

glMatrixMode(GL_TEXTURE);
glPushMatrix();
GLfloat f = 1.0;
GLfloat a = noise_amp;
for (int i=0;i<octave;i++,f*=2,a/=2) {
   glScalef(f,f,f);        fréquence
   glColor3f(a,a,a);       amplitude
   glCallList(scene);
   glMatrixMode(GL_TEXTURE);
   glPopMatrix();
   glPushMatrix();
   if (!i) {
         À faire après la 1ère passe
     glDepthMask(GL_FALSE);
     glDisable(GL_POLYGON_OFFSET_EXT);
     glBlendFunc(GL_ONE,GL_ONE);
     glEnable(GL_BLEND);
   }
}

glPopMatrix();
glDisable(GL_BLEND);
glDisable(GL_TEXTURE_3D_EXT);
```

noise_amp est l'amplitude du bruit, notée aussi α plus haut.

A.3 Calcul de la fonction caractéristique

A.3.1 Fonction identité

On commence par initialiser la mémoire de texture avec l'approximation I de la fonction identité choisie. Une texture unidimensionnelle est utilisée.

Pour la fonction identité *discontinue*, on a :

```
#define maxfun 256
GLfloat id[maxfun];

for (i=0;i<maxfun;i++)
```

```
   id[i] = 1.0/(maxfun-1)*i;

glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S    , GL_REPEAT);

glTexImage1D(GL_TEXTURE_1D,0,GL_INTENSITY_EXT,
             maxfun, 0, GL_LUMINANCE, GL_FLOAT, id);
```

L'interpolation des pixels de la texture a été déactivée (option `GL_NEAREST`) pour éviter d'avoir une pente non nulle lors du retour brusque de la valeur 1.0 à la valeur 0.0. La taille élevée de la texture permet d'éviter, malgré l'absence d'interpolation, l'effet en *marches d'escalier*. Pour cela, il suffit que les composantes de deux pixels consécutifs de la texture ne soient pas distincts de plus d'une unité, une fois discrétisées par OpenGL®. La valeur 256 a été choisie car les matériels graphiques actuels n'offrent, gère plus de 256 valeurs distinctes pour chaque composante.

Pour la fonction *triangulaire*, on a :

```
#define maxfun 256
GLfloat id[maxfun];

for (i=0;i<maxfun/2;i++) {
   id[i] = 2.0/maxfun*i;
   id[i+maxfun/2] = 1.0-2.0/maxfun*i;
}

glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S    , GL_REPEAT);

glTexImage1D(GL_TEXTURE_1D,0,GL_INTENSITY_EXT,
             maxfun, 0, GL_LUMINANCE, GL_FLOAT, id);
```

L'interpolation des couleurs est activée ici grâce à l'option `GL_LINEAR`. Le problème précédent ne se pose pas car nous avons ici une fonction continue. Il est en théorie possible de n'utiliser qu'une texture de taille 2 pixels contenant $\{0.0, 1.0\}$ (l'interpolation linéaire entre ces deux valeurs créant la fonction continue voulue), malheureusement, la précision limitée du matériel induit alors un effet de marches d'escalier. C'est pour cela que la texture a, ici aussi, une taille de 256 pixels.

La génération automatique de coordonnées de texture $(u, v, w) = (x, y, z)$ doit être initialisée exactement comme dans le cas du calcul du bruit. Il est possible d'utiliser, comme pour calculer le bruit, une texture tridimensionnelle où $(r, g, b, a) = (I(x), I(y), I(z), 0)$, mais nous donnons ici une autre approche en trois passes[2] de texture unidimensionnelle $r = g = b = a = I(x)$ qui calculent indépendamment la fonction identité sur chaque composante grâce à `glColorMask`. Chaque passe n'affecte qu'une composante r, g ou b ; la texture unidimensionnelle (qui se comporte comme une texture volumique ne variant que sur un axe et constante sur les deux autres) est tournée de $\frac{\pi}{2}$ pour simuler $r = g = b = a = I(y)$ ou $r = g = b = a = I(z)$ lors des deux dernière passes :

```
glEnable(GL_TEXTURE_1D);
glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_MODULATE);
glColor3f(fun_amp,fun_amp,fun_amp);

   Passe r = I(x)
glColorMask(GL_TRUE,GL_FALSE,GL_FALSE,GL_FALSE);
glCallList(scene);
```

[2]Cette méthode, bien que nécessitant trois passes au lieu d'une seule, a l'avantage de ne pas utiliser les textures tridimensionnelles qui ne sont pas disponibles sur toutes les plateformes. Sur certaines machines où les textures volumique sont très lentes (car implémentées en logiciel et non en matériel), la méthode en trois passes est en fait plus rapide !

```
glDepthMask(GL_FALSE);
glDisable(GL_POLYGON_OFFSET_EXT);
```

Passe $g = I(y)$

```
glColorMask(GL_FALSE,GL_TRUE,GL_FALSE,GL_FALSE);
glMatrixMode(GL_TEXTURE);
glPushMatrix();
glRotatef(90.0,0.0,0.0,1.0);
glCallList(scene);
glMatrixMode(GL_TEXTURE);
glPopMatrix();
```

Passe $b = I(z)$

```
glColorMask(GL_FALSE,GL_FALSE,GL_TRUE,GL_FALSE);
glPushMatrix();
glRotatef(90.0,0.0,1.0,0.0);
glCallList(scene);
glMatrixMode(GL_TEXTURE);
glPopMatrix();

glColorMask(GL_TRUE,GL_TRUE,GL_TRUE,GL_TRUE);
glDisable(GL_TEXTURE_1D);
```

fun_amp est l'amplitude de la fonction I, notée aussi β plus haut.

A.3.2 Fonction marbre

Une des textures *identité* définies ci-dessus est utilisée, mais nous allons calculer ici $(r, g, b, a) = (I(x), I(x), I(x), I(x))$ au lieu de $(r, g, b, a) = (I(x), I(y), I(z), 0)$ et l'ajouter grâce une équation de blend au bruit précédemment calculé dans le framebuffer. Comme la texture est définie en luminance par $r = g = b = a = I(x)$, il suffit d'une passe tridimensionnelle de texture qui affecte toutes les composantes à la fois. On utilise toujours la même génération automatique de coordonnées de texture qui donne, dans ce cas unidimensionnel, $u = x$:

```
glBlendFunc(GL_ONE,GL_ONE);
glEnable(GL_BLEND);
glEnable(GL_TEXTURE_1D);
glColor3f(fun_amp,fun_amp,fun_amp);
glCallList(scene);
glDisable(GL_TEXTURE_1D);
```

En changeant la matrice de texture avant de plaquer la fonction identité, il est possible de déplacer et de changer l'orientation et la densité des veines du marbre.

A.3.3 Fonction bois

Nous ajoutons au bruit calculé dans le framebuffer une fonction $(I(x), I(y), 0, 0)$ la troisième composante n'étant pas utile. Pour cela nous procédons comme au paragraphe B.3.1 en nous limitant aux deux premières passes. Nous élevons ensuite les composantes de tous les pixels du framebuffer au carré par une équation de blend multiplicative puis nous appliquons une matrice de couleur. Le calcul de la racine carré est supposé contenu dans celui de la table de couleur représentant la colormap. On obtient :

Ajout de $(I(x), I(y), 0, 0)$

```
glBlendFunc(GL_ONE,GL_ONE);
glEnable(GL_BLEND);
```

```
glEnable(GL_TEXTURE_1D);
glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_MODULATE);
glColor3f(fun_amp,fun_amp,fun_amp);

glColorMask(GL_TRUE,GL_FALSE,GL_FALSE,GL_FALSE);
glCallList(scene);

glDepthMask(GL_FALSE);
glDisable(GL_POLYGON_OFFSET_EXT);

glColorMask(GL_FALSE,GL_TRUE,GL_FALSE,GL_FALSE);
glMatrixMode(GL_TEXTURE);
glPushMatrix();
glRotatef(90.0,0.0,0.0,1.0);
glCallList(scene);
glMatrixMode(GL_TEXTURE);
glPopMatrix();

glColorMask(GL_TRUE,GL_TRUE,GL_TRUE,GL_TRUE);
```

Élévation au carré

```
GLint vp[4];
glPushAttrib(GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
glGetIntegerv(GL_VIEWPORT,vp);
glDisable(GL_DEPTH_TEST);
glStencilOp(GL_KEEP,GL_KEEP,GL_KEEP);
glMatrixMode(GL_PROJECTION);
glPushMatrix();
glLoadIdentity();
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glLoadIdentity();

glRasterPos2f(-1,-1);
glBlendFunc(GL_ZERO,GL_SRC_COLOR);
glReadBuffer(GL_BACK);
glCopyPixels(0,0,vp[2],vp[3],GL_COLOR);
glDisable(GL_BLEND);
```

Multiplication par une matrice de couleur

```
GLfloat mat[16] = {
  0.5, 0.5, 0.5, 0.5,
  0.5, 0.5, 0.5, 0.5,
  0.0, 0.0, 0.0, 0.0,
  0.0, 0.0, 0.0, 0.0 };
glMatrixMode(GL_COLOR);
glPushMatrix();
glLoadMatrixf(mat);

glRasterPos2f(-1,-1);
glCopyPixels(0,0,vp[2],vp[3],GL_COLOR);
glMatrixMode(GL_COLOR);
glPopMatrix();
glMatrixMode(GL_PROJECTION);
```

```
glPopMatrix();
glMatrixMode(GL_MODELVIEW);
glPopMatrix();
glPopAttrib();
```

De même que pour le marbre, on peut utiliser la matrice de texture pour modifier l'orientation et la position des cylindres concentriques formant le motif du bois.

A.4 Application de la colormap

Une table de couleurs est définie comme un tableau T de quadruplets de flottants, un flottant pour chaque composante. Nous supposons donc que la colormap est spécifiée sous cette forme. La taille t du tableau qui doit être une puissance de 2. La table de couleurs utilisée ici est `GL_POST_COLOR_MATRIX_COLOR_TABLE_SGI` (il s'agit d'une extension d'OpenGL®). Cette table de couleurs est appliquée *après* l'éventuelle matrice de couleurs.

La portion de programme suivante change chaque composante $r = g = b = a = c \in [0; 1]$ des pixels du framebuffer par la composant correspondante de l'élément $T[\lfloor c \times t \rfloor]$:

```
GLint vp[4];
glPushAttrib(GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
glGetIntegerv(GL_VIEWPORT,vp);
glDisable(GL_DEPTH_TEST);
glStencilOp(GL_KEEP,GL_KEEP,GL_KEEP);
glMatrixMode(GL_PROJECTION);
glPushMatrix();
glLoadIdentity();
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glLoadIdentity();

glColorTableSGI(GL_POST_COLOR_MATRIX_COLOR_TABLE_SGI,GL_RGBA,
                t,GL_RGBA,GL_FLOAT,T);
glRasterPos2f(-1,-1);
glReadBuffer(GL_BACK);
glEnable(GL_POST_COLOR_MATRIX_COLOR_TABLE_SGI);
glCopyPixels(0,0,vp[2],vp[3],GL_COLOR);
glDisable(GL_POST_COLOR_MATRIX_COLOR_TABLE_SGI);

glMatrixMode(GL_PROJECTION);
glPopMatrix();
glMatrixMode(GL_MODELVIEW);
glPopMatrix();
glPopAttrib();
```

Le bout de programme précédent applique la colormap à *tous* les pixels du framebuffer alors que seuls les pixels provenant de la scène doivent être affectés laissant le fond intact.

Nous utilisons le *stencil buffer* pour fabriquer un *masque* indiquant quels pixels doivent être affectée. Le stencil buffer est initialisé à 0, puis mis à 1 pour les pixels affectés par une passe tridimensionnelle. Lors des passes bidimensionnelles, un *stencil test* permet d'ignorer les pixels dont la valeur dans le stencil buffer est 0.

Voici le schéma :

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
        GL_STENCIL_BUFFER_BIT);
glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
```

```
glStencilFunc(GL_ALWAYS, 1, 0xffffffff);
glEnable(GL_STENCIL_TEST);
```

Première passe tridimensionnelle

```
glDisable(GL_STENCIL_TEST);
```

Autres passes tridimensionnelles

```
glStencilOp(GL_KEEP,GL_KEEP,GL_KEEP);
glStencilFunc(GL_EQUAL, 1, 0xffffffff);
glEnable(GL_STENCIL_TEST);
```

Passes bidimensionnelles

```
glDisable(GL_STENCIL_TEST);
```

Passes tridimensionnelles

```
glEnable(GL_STENCIL_TEST);
```

Passes bidimensionnelles

...

Cette technique est valable pour toutes les passes bidimensionnelles (multiplication par une matrice de couleurs, élévation au carré, application d'une table de couleurs,etc.). Pourtant si le fond de l'écran est noir, on peut se passer du stencil buffer dans le cas autres que l'application d'une colormap ; c'est ce que nous avons supposé implicitement précédemment. En effet, la matrice de couleurs et l'élévation au carré ne changent pas la couleur d'un pixel noir. De même, si le fond est noir et que la colormap associe à 0 la couleur noire, les pixels du fond seront inchangés même sans l'utilisation du stencil buffer.

Si le fond n'est pas noir (si on veut, par exemple, ajouter un objet avec une texture procédurale à une scène déjà rendue dans le framebuffer), l'utilisation du stencil buffer est nécessaire.

Annexe B

Utilisation de la bibliothèque GLP

Mon travail de stage débouche en particulier sur la programmation en C de la bibliothèque GLP permettant d'automatiser les tâches décrites dans l'annexe A. Elle permet d'ajouter facilement des textures procédurales à n'importe quelle scène rendue avec OpenGL® ; elle même utilise OpenGL® et certaines de ses extensions. Il est également possible de *mixer* le rendu classique avec OpenGL® et le rendu avec la bibliothèque GLP.

Cette annexe présente quelques fonctions de la bibliothèque et donne quelques exemples d'utilisation mais n'est pas exhaustif.

B.1 Comment utiliser GLP

GLP est basée sur les bibliothèques OpenGL® et OpenGL® Utility, il faut donc inclure dans le programme :

```
#include <GL/gl.h>
#include <GL/glu.h>
#include "glp.h"
```

L'initialisation de GLP se fait par un appel à `glpInit()` après avoir initialisé OpenGL®.

Les fonctions de rendu utilisent une scène définie par une *display list.* Ces scènes ne doivent par comporter d'information de couleur ou de coordonnées de texture.

Chaque fonction de rendu effectue plusieurs passes tridimensionnelles et bidimensionnelles. A la fin d'une fonction de rendu GLP, une passe bidimensionnelle peut rester en suspens pour être factorisée avec une autre passe lors de l'appel à la fonction GLP de rendu suivante. Ceci permet de minimiser le nombre de passes.

La fonction `glpFinalize()` achève toutes les passes en suspens. Il est nécessaire d'appeler cette fonction avant d'effectuer du rendu directement avec OpenGL® ou quand le rendu est fini, mais ce n'est pas la peine de le faire entre deux appels à des fonctions GLP.

Avant de rendre une scène, il faut initialiser OpenGL® (comme décrit dans l'annexe A) par :

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
        GL_STENCIL_BUFFER_BIT);
glPolygonOffsetEXT(1.0,0.0001);
glDepthMask(GL_TRUE);
glEnable(GL_POLYGON_OFFSET_EXT);
glEnable(GL_STENCIL_TEST);
glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
glStencilFunc(GL_ALWAYS, 1, 0xffffffff);
```

Après une passe tridimensionnelle, les fonctions de rendu GLP appellent automatiquement :

```
glDisable(GL_POLYGON_OFFSET_EXT);
glDepthMask(GL_FALSE);
glStencilOp(GL_KEEP,GL_KEEP,GL_KEEP);
```

Ce n'est donc pas la peine de le faire manuellement, sauf si on effectue la première passe tridimensionnelle soi-même.

Lors d'une passe bidimensionnelle, le stencil buffer est momentanément désactivé par `glStencilOp(GL_KEEP,GL_KEEP,GL_KEEP)` et la matrice de projection et de modèle sont changées. Tous ces paramètres sont restaurés ensuite à leur valeur lors de l'appel à la fonction.

De manière générale, les paramètres OpenGL® et les matrices changées sont restaurés.

Les paramètres changés et non restaurés sont :

- l'équation de blend ;
- l'équation d'environnement ;
- la couleur courante ;
- la génération automatique des coordonnées de texture ;
- la mémoire de texture.

A la sortie d'une fonction de rendu, on a de plus :

- l'équation de blend désactivée ;
- l'application de texture désactivée ;
- l'écriture dans le z-buffer interdit ;
- le biais pour la profondeur des pixels désactivée ;
- la génération automatique des coordonnées de texture desactivée.

Beaucoup de paramètres modifient la manière dont les fonctions GLP rendent une scène. La valeur d'un paramètre est changée par `glpSetParameteri/f` et lue par `glpGetParameteri/f`.

Certains appels à des fonctions GLP peuvent provoquer des erreurs. La fonction `glpGetError()` renvoie le code numérique de la première erreur qui a eu lieu depuis le dernier appel à `glpGetError` (une seule erreur est stockée). `glpErrorString` permet ensuite d'obtenir une chaine de caractères décrivant une erreur à partir de son code. Les fonctions `glpGetError` et `glpErrorString` remplacent en fait les fonctions `glGetError` et `gluErrorString` : elles gèrent les erreurs générées par les bibliothèques OpenGL® et OpenGL Utility en plus des erreurs spécifiques à GLP.

B.2 Liste des fonctions

- `glpDrawNoise(GLuint scene)`
  Rend la scène contenue dans la display list *scene* avec une texture de bruit turbulent.
- `glpDrawPattern(GLuint scene, GLenum func)`
  Rend la scène *scene* avec une texture identité, puis applique la fonction *func* sur les pixels du framebuffer. En un pixel correspondant au point de coordonnées (x, y, z) de la scène, voici le résultat en fonction de *func* :
  –`GLP_FUNC_GRADIENT` donne $r = g = b = a = x$;
  –`GLP_FUNC_CYLINDER` donne $r = g = b = a = \frac{1}{2}(x^2 + y^2)$;
  –`GLP_FUNC_SPHERE` donne $r = g = b = a = \frac{1}{3}(x^2 + y^2 + z^2)$.
  Une équation de blend additive permet de calculer la fonction, non pas en (x, y, z), mais en $(x + r', y + g', z + b')$ si (r', g', b', a') sont les composantes initiales des pixels du framebuffer.
- `glpSetColormap(GLuint sizetab, GLuint ntab, GLenum mode,GLuint size,float* tab)`
  Défini une colormap qui sera utilisée par `glpDoColormap`.

  La colormap est définie par un tableau `tab` contenant `sizetab` groupes de 5 flottants consécutifs nommés (p, r, g, b, a). Chaque groupe donne la valeur de la colormap (r, g, b, a) en un point $p \in [0; 1]$. La colormap est ensuite obtenue par interpolation linéaire de ces points de contrôle ; cette fonction sur $[0; 1]$ est ramenée sur $[0; \frac{1}{ntab}]$ et répétée *nbtab* fois pour obtenir à nouveau une fonction sur $[0; 1]$.
  `size` indique la taille de la table de couleur qui, en interne, discrétise la colormap obtenue. Ce doit être une puissance de deux. Le maximum de qualité est obtenu quand `size` vaut le nombre de valeurs que peut prendre chaque composante du framebuffer, c'est à dire 256 dans la plupart des cas.

- `glpDoColormap(void)`
  Applique sur les pixels du framebuffer la colormap définie par `glpSetColormap`. En fait, la colormap utilisée est $C'(c) = C(c^{\texttt{GLP\_GAMMA}})$ si C est la colormap définie par `glpSetColormap`.
  Il est possible d'utiliser le `stencil buffer` pour limiter l'application de la colormap à une partie seulement du framebuffer (et, par exemple, épargner les pixels n'appartenant pas à la scène).

B.3 Liste des paramètres

Les paramètres sont de deux types : ceux à valeur entière et ceux à valeur flottante. Les premiers utilisent les fonctions `glpSetParameteri(GLuint param, GLuint value)` et `glpGetParameteri(GLuint param, GLuint* var)` tandis que les autres utilisent `glpSetParameterf(GLuint param, GLfloat value)` et `glpGetParameterf(GLuint param, GLfloat* var)`. Certains paramètres à valeur entière ne peuvent, de plus, prendre que quelques valeur définies par des constantes symboliques.

- `GLP_DIMENSION : GLuint`
  Indique si le bruit turbulent est tridimensionnel (`GLP_MODE_3D`) ou bidimensionnel (`GLP_MODE_2D`, par défaut). Un bruit tridimensionnel offre une meilleur qualité mais nécessite l'extension OpenGL® autorisant les textures volumiques.
- `GLP_COMPONENT : GLuint`
  Indique si le bruit est en luminance (`GLP_MODE_L`) ou composé de trois bruits décorrélés (`GLP_MODE_RGB`, par défaut).
- `GLP_FREQ_MULT : GLfloat`
  Le bruit turbulent est composé d'une superposition de bruits pseudo-périodiques de fréquence croissante : chaque couche voit sa fréquence multipliée par ce paramètre (valeur de 2 par défaut).
- `GLP_AMP_DIV : GLfloat`
  Le bruit turbulent est composé d'une superposition de bruits pseudo-périodiques d'amplitude décroissante : chaque couche voit son amplitude divisée par ce paramètre (valeur de 2 par défaut).
- `GLP_NOISE_AMP : GLfloat`
  Amplitude du bruit pseudo-périodique, aussi notée α ($\frac{1}{2}$ par défaut, doit se situer dans $[0;1]$).
  L'amplitude totale du bruit turbulent est :

$$\texttt{GLP\_NOISE\_AMP}(1 + \texttt{GLP\_AMP\_DIV}^{-1} + \texttt{GLP\_AMP\_DIV}^{-2} + \cdots + \texttt{GLP\_AMP\_DIV}^{1-\texttt{GLP\_OCTAVE}})$$

- `GLP_OCTAVE : GLuint`
  Nombre total de couches de bruit pseudo-périodique utilisées pour fabriquer le bruit turbulent (4 par défaut).
- `GLP_NOISE_SIZE : GLuint`
  Taille du tableau tridimensionnel de valeurs aléatoires servant à définir le bruit pseudo-périodique. Ce doit être une puissance de deux (16 par défaut, ce qui correspond à un tableau de taille $16 \times 16 \times 16$).
- `GLP_FUN_MODE : GLuint`
  Indique la fonction I utilisée : `GLP_MODE_RAMP` correspond à I_r (valeur par défaut), `GLP_MODE_TRIANGULAR` correspond à I_t et `GLP_MODE_SINUSOIDAL` correspond à $I_s(c) = \frac{1}{2}(1 - \cos 2\pi c)$ (fonction sinusoïdale peu utilisée en pratique).
- `GLP_FUN_SIZE : GLuint`
  Taille du tableau unidimensionnel qui discrétise I (256 par défaut).
- `GLP_FUN_AMP : GLfloat`
  Amplitude de I, aussi notée β ($\frac{1}{2}$ par défaut, doit être dans $[0;1]$).
- `GLP_GAMMA : GLfloat`
  Facteur de correction exponentiel de la colormap, noté aussi γ (1 par défaut).

B.4 Exemples

B.4.1 Rendu d'un bruit turbulent B en luminance

D'abord, il faut initialiser la bibliothèque et placer l'objet à dessiner (un tore dans cet exemple) dans une display list :

```
glpInit();

GLuint scene = glGenLists(1);
glNewList(scene, GL_COMPILE);
glutSolidTorus(1.0,2.0,20,20);
glEndList();
glpSetParameterf(GLP_NOISE_AMP,1.0);
glpSetParameteri(GLP_COMPONENT,GLP_MODE_L);

glDepthTest(GL_LESS);
```

Ensuite nous dessinons la scène :

```
glDepthMask(GL_TRUE);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glEnable(GL_DEPTH_TEST);
glPolygonOffsetEXT(1.0,0.0001);
glEnable(GL_POLYGON_OFFSET_EXT);
glpDrawNoise(scene);
glpFinalize();
```

B.4.2 Rendu d'un marbre rose

Cette fois nous initialisons de plus une colormap :

```
glpInit();
glDepthTest(GL_LESS);

    Définition d'une display list nommée list
    comme dans le cas précédent

float marble[6][5] = {
     {0.00, 0.0,0.0,0.0,1.0},
     {0.05, 0.8,0.2,0.2,1.0},
     {0.50, 1.0,0.5,0.5,1.0},
     {0.60, 1.0,1.0,1.0,1.0},
     {0.95, 0.6,0.0,0.0,1.0},
     {1.00, 0.0,0.0,0.0,1.0}};
glpSetColormap(256,4,GLP_NORM,6,(float*)marble);

    Bruit en luminance :
glpSetParameteri(GLP_COMPONENT,GLP_MODE_L);
    Amplitude du bruit :
glpSetParameterf(GLP_NOISE_AMP,0.25);
    Amplitude de I :
glpSetParameterf(GLP_FUN_AMP,0.75);
```

Puis nous dessinons un bruit en luminance et appliquons une fonction marbre ainsi qu'une colormap :

```
glDepthMask(GL_TRUE);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    Choisir les matrices de projection, de modèle,...

    Dessine un bruit de période 0.2
```

```
glMatrixMode(GL_TEXTURE);
glLoadIdentity();
glScalef(0.2,0.2,0.2);
glEnable(GL_DEPTH_TEST);
glPolygonOffsetEXT(1.0,0.0001);
glEnable(GL_POLYGON_OFFSET_EXT);
glpDrawNoise(scene);
```

Applique la fonction gradient

```
glMatrixMode(GL_TEXTURE);
glLoadIdentity();
glScalef(0.2,0.2,0.2);
glTranslatef(0.5,0.5,0.5);
glEnable(GL_BLEND);
glBlendFunc(GL_ONE,GL_ONE);
glpDrawPattern(scene,GLP_FUNC_GRADIENT);
```

Applique la colormap

```
glDisable(GL_BLEND);
glpDoColormap();
glpFinalize();
```

La colormap affecte tous les points du framebuffer, même le fond de l'écran mais ce n'est pas visible car la valeur 0 correspond au noir dans la colormap.

B.4.3 Rendu d'un bois

Procédure d'initialisation :

```
glpInit();
glDepthTest(GL_LESS);
```

Définition d'une display list nommée list
comme dans le cas précédent

```
float wood[][4] = {
     {0.000, 1.00, 1.00, 1.00 },
     {0.120, 0.70, 0.41, 0.11 },
     {0.231, 0.70, 0.46, 0.11 },
     {0.496, 1.00, 1.00, 1.00 },
     {0.701, 1.00, 1.00, 1.00 },
     {0.829, 0.70, 0.46, 0.11 },
     {1.000, 1.00, 1.00, 1.00 }};
glpSetColormap(256,4,GLP_NORM,7,(float*)wood);
```

Trois bruits décorrélés :

```
glpSetParameteri(GLP_COMPONENT,GLP_MODE_RGB);
```

Amplitude du bruit :

```
glpSetParameterf(GLP_NOISE_AMP,0.25);
```

Amplitude de I :

```
glpSetParameterf(GLP_FUN_AMP,0.75);
```

Fonction identité triangulaire I_t :

```
glpSetParameterf(GLP_FUN_MODE,GLP_TRIANGULAR);
```

Cette fois ci, nous utilisons le stencil-buffer pour que la colormap ne modifie que les pixels du tore (ainsi les pixels du fond restent noir bien que la colormap n'associe pas le noir à la valeur 0) :

```
glDepthMask(GL_TRUE);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
        GL_STENCIL_BUFFER_BIT);
```

Choisir les matrices de projection, de modèle,...

Met à 1 dans le stencil buffer les pixels du framebuffer qui sont dessinés

```
glEnable(GL_STENCIL_TEST);
glStencilOp(GL_REPLACE, GL_REPLACE, GL_REPLACE);
glStencilFunc(GL_ALWAYS, 1, 0xffffffff);
```

Dessine un bruit

```
glMatrixMode(GL_TEXTURE);
glLoadIdentity();
glScalef(0.25,0.25,0.25);
glEnable(GL_DEPTH_TEST);
glPolygonOffsetEXT(1.0,0.0001);
glEnable(GL_POLYGON_OFFSET_EXT);
glpDrawNoise(scene);
```

Applique la fonction cylindre

```
glMatrixMode(GL_TEXTURE);
glLoadIdentity();
glScalef(0.3,0.3,0.3);
glTranslatef(0.5,0.5,0.5);
glEnable(GL_BLEND);
glBlendFunc(GL_ONE,GL_ONE);
glpDrawPattern(scene,GLP_FUNC_CYLINDER);
```

Applique la colormap sur les pixels ayant leur valeur de stencil a 1

```
glStencilFunc(GL_EQUAL,1,0xffffffff);
glStencilOp(GL_KEEP,GL_KEEP,GL_KEEP);
glDisable(GL_BLEND);
glpDoColormap();
glpFinalize();
```

Bibliographie

[BVI91] Chakib Bennis, Jean-Marc Vézien, and Gérard Iglésias. Piecewise surface flattening for non-distored texture mapping. In *SIGGRAPH'91*, volume 25, pages 237–246. Thomas W. Sederberg eds., 1991. `http://www-syntim.inria.fr/syntim/textes/siggraphjmv-eng.html`.

[EMP+94] D. Ebert, K. Musgrave, P. Peachey, K. Perlin, and S. Worley. *Texturing and Modeling : A Procedural Approach*. Academic Press Professional, 1994. `http://www.cs.umbc.edu/~ebert/`.

[FR86] A. Fournier and W.T. Reeves. A simple model of ocean waves. *SIGGRAPH'86*, 20 :75–84, 1986. `http://www.acm.org/pubs/citations/proceedings/graph/15922/p75-fournier/%`.

[FvDF90] J. D. Foley, A. van Dam, and J. F. Feiner, S. K. amd Hughes. *Computer Graphics : Principles and Practices (2nd Edition)*. Addison-Wesley, 1990.

[Gar85] Geoffrey Y. Gardner. Visual simulation of clouds. In *SIGGRAPH'85*, volume 19, pages 297–303. B. A. Barsky, 1985.

[KK89] J. T. Kahiya and T. L. Kay. Rendering fur with three dimensional textures. In *SIGGRAPH'89*, volume 23(3), pages 271–280. Jeffrey Lane Editor, 1989. `http://www.acm.org/pubs/citations/proceedings/graph/74333/p271-kahiya/`.

[Ney98] Fabrice Neyret. Modeling animating and rendering complex scenes using volumetric textures. In *IEEE Transactions on Visualization and Computer Graphics*, volume 4(1), 1998. ISSN 1077-2626.

[Ped95] Hand Kohling Pedersen. Decorating implicit surfaces. In *SIGGRAPH'95*, pages 291–300. Robert Cook, 1995. `http://implicit.eecs.wsu.edu/course14/decorate.ps.gz`.

[Per85] Ken Perlin. An image synthetizer. In *SIGGRAPH'85*, volume 19(3), pages 287–296. B. A. Barsky, 1985.

[WK91] Andrew Witkin and Michael Kass. Reaction-diffusion textures. In *SIGGRAPH'91*, volume 25, pages 299–308. W. Sederberg, 1991.

[Wor96] Steven P. Worley. A cellular texturing basis function. In *SIGGRAPH'96*, pages 291–294. Holly Rushmeier, 1996. `http://www.acm.org/pubs/citations/proceedings/graph/237170/p291-worley`.