Université de Montréal

Modélisation procédurale par composants

par
Luc Leblanc

Département d'informatique et de recherche opérationnelle
Faculté des arts et des sciences

Thèse présentée à la Faculté des études supérieures et postdoctorales
en vue de l'obtention du grade de Philosophiæ Doctor (Ph.D.)
en informatique

Août 2011

Université de Montréal

Faculté des études supérieures

Cette thèse intitulée :

Modélisation procédurale par composants

présentée par :

Luc Leblanc

a été évaluée par un jury composé des personnes suivantes :

Max Mignotte

président-rapporteur

Pierre Poulin

directeur de recherche

Neil F. Stewart

membre du jury

Éric Galin

examinateur externe

Max Mignotte

représentant du doyen de la FAS

Résumé

Le réalisme des images en infographie exige de créer des objets (ou des scènes) de plus en plus complexes, ce qui entraîne des coûts considérables. La modélisation procédurale peut aider à automatiser le processus de création, à simplifier le processus de modification ou à générer de multiples variantes d'une instance d'objet. Cependant même si plusieurs méthodes procédurales existent, aucune méthode unique permet de créer tous les types d'objets complexes, dont en particulier un édifice complet. Les travaux réalisés dans le cadre de cette thèse proposent deux solutions au problème de la modélisation procédurale : une solution au niveau de la géométrie de base, et l'autre sous forme d'un système général adapté à la modélisation des objets complexes.

Premièrement, nous présentons le *bloc*, une nouvelle primitive de modélisation simple et générale, basée sur une forme cubique généralisée. Les blocs sont disposés et connectés entre eux pour constituer la forme de base des objets, à partir de laquelle est extrait un maillage de contrôle pouvant produire des arêtes lisses et vives. La nature volumétrique des blocs permet une spécification simple de la topologie, ainsi que le support des opérations de CSG entre les blocs. La paramétrisation de la surface, héritée des faces des blocs, fournit un soutien pour les textures et les fonctions de déplacements afin d'appliquer des détails de surface. Une variété d'exemples illustrent la généralité des blocs dans des contextes de modélisation à la fois interactive et procédurale.

Deuxièmement, nous présentons un nouveau système de modélisation procédurale qui unifie diverses techniques dans un cadre commun. Notre système repose sur le concept de composants pour définir spatialement et sémantiquement divers éléments. À travers une série de déclarations successives exécutées sur un sous-ensemble de composants obtenus à l'aide de requêtes, nous créons un arbre de composants définissant ultimement un objet dont la géométrie est générée à l'aide des blocs.

Nous avons appliqué notre concept de modélisation par composants à la génération d'édifices complets, avec intérieurs et extérieurs cohérents. Ce nouveau système s'avère général et bien adapté pour le partionnement des espaces, l'insertion d'ouvertures (portes et fenêtres), l'intégration d'escaliers, la décoration de façades et de murs, l'agencement de meubles, et diverses autres opérations nécessaires lors de la construction d'un édifice complet.

Mots clefs :

Modélisation procédurale, composant, architecture, édifice, grammaire de formes, opération booléenne, CSG, bloc, représentation géométrique, surface de subdivision.

Abstract

The realism of computer graphics images requires the creation of objects (or scenes) of increasing complexity, which leads to considerable costs. Procedural modeling can help to automate the creation process, to simplify the modification process or to generate multiple variations of an object instance. However although several procedural methods exist, no single method allows the creation of all types of complex objects, including in particular a complete building. This thesis proposes two solutions to the problem of procedural modeling : one solution addressing the geometry level, and the other introducing a general system suitable for complex object modeling.

First, we present a simple and general modeling primitive, called a *block*, based on a generalized cuboid shape. Blocks are laid out and connected together to constitute the base shape of complex objects, from which is extracted a control mesh that can contain both smooth and sharp edges. The volumetric nature of the blocks allows for easy topology specification, as well as CSG operations between blocks. The surface parameterization inherited from the block faces provides support for texturing and displacement functions to apply surface details. A variety of examples illustrate the generality of our blocks in both interactive and procedural modeling contexts.

Second, we present a novel procedural modeling system which unifies some techniques into a common framework. Our system relies on the concept of components to spatially and semantically define various elements. Through a series of successive statements executed on a subset of queried components, we grow a tree of components ultimately defining an object whose geometry is made from blocks.

We applied our concept and representation of components to the generation of complete buildings, with coherent interiors and exteriors. It proves general and well adapted to support partitioning of spaces, insertion of openings (doors and windows), embedding of staircases, decoration of façades and walls, layout of furniture, and various other operations required when constructing a complete building.

Keywords :

Procedural modeling, component, architecture, building, shape grammar, boolean operation,

CSG, block, geometric representation, subdivision surface.

Table des matières

Table des figures

Table des listages

Remerciements

Les travaux de cette thèse ont été rendus possibles grâce aux subventions du FQRNT (personnelle et équipe), du CRSNG et de GRAND PROMO.

Une recherche d'une telle durée ne peut se faire sans l'aide et le support de bien des gens croisés au fil des ans. Il y a en premier lieu tous les membres du laboratoire d'infographie (LIGUM), anciens et courants (au moment d'écrire ces lignes), dont particulièrement, Simon Bouvier-Zappa, Marie-Élise Cordeau et Jean-François Dufort pour leur support pendant, mais aussi après leur séjour passé au laboratoire.

D'autres ont aussi apporté leur aide en participant au processus de soumissions d'articles. J'aimerais donc remercier Philippe Beaudoin, Romain Pacanowski, Neil Stewart, Victor Ostromoukhov et George Drettakis pour leurs commentaires éclairés, ainsi que Eric Blanchard pour son aide lors de la création de vidéos. Il y a aussi eu des collaborations effectuées durant cette longue période, entre autres avec Fabrice Rousselle, Victor Ostromoukhov et Petrik Clarberg, et une autre avec Jean-François Dufort.

Et finalement deux personnes nécessitent des remerciements particuliers pour leur engagement constant tout au long de mes études. Tout d'abord, mon directeur Pierre Poulin pour son support, son aide, sa patience et avec qui c'est toujours un plaisir de travailler ensemble, et ensuite Jocelyn Houle mon ami et partenaire de travail pour son support, son aide constante et particulièrement sa patience pour ne pas m'avoir lâché et sans qui je n'aurais pas réussi à terminer.

Chapitre 1

Introduction

1.1 Description du problème et motivations

FIGURE 1.1 – Ville de New York entièrement digitale, tirée du film *King Kong* de 2005.

L'importance de la modélisation est omniprésente dans presque tous les domaines de l'infographie. Tous les algorithmes tri-dimensionnels s'effectuent à la base sur un modèle géométrique ou sur une scène (ensemble d'objets), et de plus en plus, une complexité croissante est exigée de ces modèles afin d'en augmenter leur réalisme. Les domaines de la simulation (vol, conduite, etc.), des effets spéciaux pour le cinéma et la télévision, et des jeux vidéo sont particulièrement avides en ce sens. Ils tentent toujours de repousser la limite du réalisme et exigent par conséquent un niveau très élevé de complexité de la part des modèles utilisés.

Pour ne nommer que quelques exemples de cette complexité croissante, on peut citer le cas de films tels *King Kong* en 2005 (voir la figure 1.1) et *Superman Returns* en 2006. Plus de deux

ans de recherche et développement ont été nécessaires afin de recréer la ville de New York des années 1930, telle que vue dans la version 2005 du film *King Kong*, alors que le film *Superman Returns* a exigé l'équivalent d'un an d'efforts pour 15 personnes afin de modéliser la ville fictive de Métropolis.

Cette complexité de modélisation croissante entraîne d'énormes coûts de développement, tant en argent qu'en temps. Les difficultés de création de ces scènes et modèles résident surtout en quelques points principaux. La quantité de modèles à créer est souvent énorme. On n'a qu'à imaginer les deux cas mentionnés plus haut, où l'on a recréé une ville avec ses nombreux détails (des milliers d'édifices, de routes, de véhicules, de panneaux de signalisation, de végétation, de personnages, etc.). En plus de la quantité, il y a la qualité visuelle. Chacun de ces modèles commande un niveau de détail très élevé pour pouvoir hériter de l'épithète "réel". Un seul de ces objets peut être composé d'un maillage de plusieurs millions de polygones. En plus de l'information géométrique, les objets modélisés doivent être complémentés d'attributs tels une paramétrisation de surface, plusieurs textures, une description de ses matériaux, des caractéristiques physiques pour fin de simulation, d'informations squelettiques et même musculaires pour fin d'animation, etc.

En plus d'occasionner d'énormes difficultés de modélisation, l'immense taille des scènes entraîne un ensemble de problèmes connexes. L'entreposage et le transfert, sur disque et plus particulièrement en mémoire vive de l'ordinateur ou de la carte graphique, devient un problème sérieux à gérer. Lorsque la mémoire ne peut contenir en entier la totalité de la scène, on se doit de gérer le chargement partiel de cette dernière. De plus, la transmission des données sur réseaux devient aussi difficile. Dans tous ces cas, il est important de pouvoir compresser et décompresser les scènes efficacement tout en tenant compte de la localité des objets, ce qui est particulièrement critique dans le cas des transmissions et des chargements progressifs. Il y a aussi des problèmes d'efficacité lors de l'affichage et pour y remédier, l'utilisation de niveaux de détail est souvent employée selon la taille dans l'image finale, mais vue la complexité des scènes, la gestion des niveaux de détail doit être automatique.

Pour corriger ces problèmes dus à la complexité de modélisation, toute une gamme d'outils ont été développés. Les modeleurs proposent de nombreuses fonctionnalités pouvant être effectuées sur un modèle de base (maillage triangulaire, de subdivision, par points, etc.). On peut, entre autres, créer des triangles, déplacer des sommets, couper, coller ou dupliquer des polygones, extruder des faces, etc. Bien que ces outils aident à la création de modèles complexes,

des problèmes importants persistent : la quantité de choix et de variantes d'outils et de primitives disponibles rendent complexe la tâche des usagers, et malgré leur nombre, la manipulation des objets se fait toujours à un bas niveau (on doit modifier la topologie par exemple) et demeure plusieurs ordres de magnitude trop lente pour assister à la production de scènes ayant la complexité voulue.

Dans une autre avenue de solutions, des techniques procédurales ont été inventées pour produire automatiquement, avec un minimum d'interaction d'un usager, des objets représentés par un modèle géométrique de base. On retrouve parmi elles, les modèles fractals utilisés pour créer des terrains ou des nuages, les systèmes-L et les grammaires, plus génériques, pour créer de la végétation, des édifices, des routes, des villes, et encore bien d'autres types d'objets. Cependant ces techniques ne corrigent pas tous les problèmes.

Certaines apportent une solution trop spécifique pour être employées à d'autres fins, ou encore sont trop complexes à utiliser et à contrôler pour permettre à un usager non initié (et même dans certains cas pour des spécialistes) de raffiner la solution efficacement afin d'obtenir le résultat escompté. La qualité des objets produits par ces techniques dépend, entre autres, du modèle géométrique de base employé. Différents problèmes (craques, mauvaise paramétrisation de surface, maillage non lisse, etc.) sont directement liés au fait qu'aucun des modèles de base ne possède toutes les caractéristiques importantes pour faciliter la modélisation procédurale.

Les caractéristiques principales recherchées pour des modèles géométriques sont : avoir un volume bien défini ; posséder une bonne paramétrisation de surface ; offrir une facilité d'utilisation (e.g. pas de topologie à spécifier) et de contrôle (contrôle de la courbure, arêtes vives, etc.) ; permettre l'ajout d'informations sémantiques pour des fins d'animation, de simulation physique, etc. ; produire un maillage de qualité, bien formé, fermé, sans interpénétration, et avec une résolution adaptative pour des fins d'affichage rapide.

1.2 Contributions

La recherche effectuée dans ce document tente de répondre à ce besoin de faciliter la construction de scènes complexes. Pour y parvenir deux solutions nouvelles sont proposées. Dans un premier temps, on introduit un nouveau modèle géométrique de représentation de base remédiant à plusieurs lacunes des modèles précédents, en répondant à tous les critères mentionnés ci-haut. Ce nouveau modèle est accompagné d'un nouvel algorithme de CSG (*Constructive Solid Geometry*) efficace et assez robuste pour être utilisé dans un contexte procédural. Deuxièmement, un

nouveau paradigme de modélisation procédurale par composants, générique, souple et simple à utiliser est proposé. Ce système est ensuite mis à contribution en construisant procéduralement des édifices complexes et complets comprenant intérieurs et extérieurs. Ces deux solutions ont été publiées dans les conférences internationales *Computer Graphics International* [LHP11b] et *Graphics Interface* [LHP11a] respectivement, et dans le premier cas est aussi paru dans le revue *The Visual Computer*.

1.3 Plan de la thèse

Pour introduire notre recherche, le prochain chapitre (2) présente les travaux reliés antérieurs. Cette description divise la modélisation en deux catégories principales, soient les représentations géométriques de base et la modélisation procédurale. Les travaux représentant les recherches plus étroitement reliées à nos contributions sont présentés dans les chapitres correspondant directement à leur sujet. Le chapitre 3 décrit la nouvelle primitive de modélisation, le bloc, ainsi que tous les algorithmes pour son élaboration, dont entre autres un nouvel algorithme de CSG. Suit le chapitre 4, introduisant le nouveau système de modélisation procédurale par composants. Ce chapitre se concentre sur sa définition et ses principes. Le chapitre 5 présente un cas complexe d'utilisation du système par composants : la modélisation procédurale d'édifices complets avec intérieurs et extérieurs. Ce chapitre débute par une introduction présentant l'état de l'art quant à la modélisation d'édifices avant d'expliquer la nouvelle procédure. Les travaux futurs suivent au chapitre 6, pour indiquer des directions intéressantes ouvertes par cette recherche. Finalement, la conclusion au chapitre 7 fait le point sur les contributions obtenues dans ces travaux.

Chapitre 2

Travaux antérieurs

Puisque notre sujet de recherche traite de la modélisation à deux niveaux, soient au niveau de base descriptif et au niveau procédural, il est par conséquent de mise d'étudier l'ensemble du domaine. Toutefois, l'étendue du sujet est telle qu'un résumé exhaustif de l'état de l'art serait à lui seul un travail de considérable envergure, allant bien au-delà d'une thèse de doctorat. Alors, par souci de simplicité et de concision, seul un très bref survol des techniques courantes de modélisation est présenté dans ce chapitre. Les travaux les plus pertinents sont abordés aux chapitres respectifs portant sur la recherche effectuée.

2.1 Modélisation géométrique

Le terme *représentation de base* est employé ici pour décrire une représentation explicite d'un modèle géométrique par opposition à une *représentation procédurale* définissant ce dernier par un ensemble de règles et de procédures. Une représentation de base contient donc une description directe du modèle à l'aide par exemple de points, de polygones, de surfaces courbes, etc., alors qu'une représentation procédurale décrit un procédé pour construire un modèle, soit à l'aide d'un langage, d'une grammaire ou d'un ensemble d'opérations à appliquer. La représentation de base, généralement produite par un usager à l'aide d'un modeleur, peut aussi être le résultat d'une numérisation ou encore l'aboutissement d'un processus de modélisation procédurale. Dans ce dernier cas, la représentation de base est utilisée afin de permettre la manipulation des objets pour fin de visualisation, d'animation, de modification, etc.

Il existe de nombreuses représentations de base pour exprimer un objet tri-dimensionnel.

On peut les séparer en deux grandes classes, soient les représentations surfaciques et celles volumiques.

2.1.1 Représentation surfacique

Les représentations surfaciques, aussi appelées représentations frontalières, définissent un objet exclusivement par la description de sa frontière. Bien que ces méthodes de description soient dans bien des cas incomplètes et insuffisantes pour définir sans ambiguïté un objet, ce sont aujourd'hui les représentations les plus employées, puisqu'elles permettent, entre autres, un affichage simple et rapide.

Maillage polygonal

La plus simple des représentations frontalières consiste à approximer, dans le cas d'une surface courbe, la frontière d'un objet par un maillage polygonal tel qu'illustré à la figure 2.1.

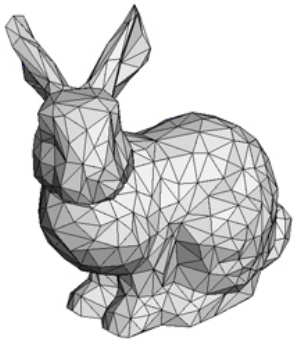


FIGURE 2.1 – Modèle d'un lapin, exprimé à l'aide d'un maillage polygonal.

Ce maillage peut être formé à partir de polygones quelconques, mais est généralement restreint à un ensemble de quadrilatères ou de triangles. Cette restriction permet de simplifier les opérations applicables à un tel maillage, et aussi d'éliminer l'ambiguïté de définition de la surface qui survient lorsque les sommets d'un polygone ne sont pas tous dans un même plan. En effet, un polygone formé de trois sommets repose dans un seul plan bien défini, alors qu'un polygone formé de quatre sommets peut être considéré comme une surface bilinéaire, encore une fois bien définie. Cependant, la surface d'un polygone ayant au moins cinq sommets ne partageant pas le même plan n'a pas de définition unique et simple.

Afin de définir un maillage polygonal, trois entités doivent être énumérées ainsi que leurs interrelations. Il y a d'abord les sommets, puis les arêtes reliant les sommets, et finalement

les faces des polygones délimitées par les arêtes. Ces entités et ces relations peuvent être exprimées sous différents formats de structures de données. Les deux formats les plus connus et les plus utilisés sont le *winged-edge* [Bau72] et les demi-arêtes (*half-edge*) [Man88]. Un schéma explicatif de ces deux formats est présenté à la figure 2.2.

Dans les structures de données du format *winged-edge*, un sommet contient sa position en 3D ainsi qu'un lien vers une des arêtes liées à celui-ci. Une face possède, quant à elle, un seul lien vers une de ses arêtes. Une arête est liée à ses deux sommets, ses deux faces ainsi que quatre de ses arêtes voisines.

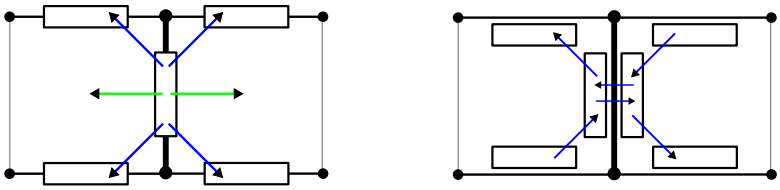


FIGURE 2.2 – Représentation des relations entre les sommets, arêtes et faces pour le format *winged-edge* à gauche et demi-arêtes à droite. Une flèche bleue représente un lien entre arêtes (à gauche) ou demi-arêtes (à droite), alors qu'une flèche verte représente un lien entre une arête et une face.

Afin de déterminer l'orientation des faces, le format demi-arête exprime les faces non pas par un ensemble d'arêtes mais plutôt de demi-arêtes orientées. Dans ce schéma, une face contient un lien vers une de ses demi-arêtes, alors que chaque demi-arête est liée à ses deux sommets, sa demi-arête suivante dans la formation de sa face, ainsi que sa demi-arête voisine formant la même arête.

Le *directed-edge* [CKS98] apporte une optimisation côté mémoire à cette dernière représentation (demi-arête) pour les maillages à base de triangles. Il est à noter que cette variante peut aussi être appliquée à une représentation à base de quadrilatères. Cette représentation échange de la vitesse d'exécution pour de la mémoire en gardant le minimum d'informations possible et en recalculant au besoin les informations manquantes. De plus certaines informations peuvent être inférées par les structures de données. Par exemple, le fait de garder l'information de connectivité dans un tableau permet de dériver automatiquement la relation entre les demi-arêtes et leurs faces, et inversement.

Surface de subdivision

Les surfaces de subdivision peuvent être considérées comme une extension du maillage polygonal, ajoutant à ce dernier la définition de surfaces courbes. Le maillage polygonal sert de maillage de contrôle dans un processus de subdivision qui définit la surface courbe comme étant la surface limite (si elle existe) de ce processus appliqué à l'infini. Un exemple de modèle géométrique entièrement créé à l'aide de surfaces de subdivision est illustré à la figure 2.3. Ce modèle, provenant du film de court métrage *Geri's Game* de *Pixar*, est l'un des premiers exemples issus du milieu industriel.

FIGURE 2.3 – Modèle entièrement créé par surfaces de subdivision. Image tirée de [DKT98].

L'application du processus de subdivision s'effectue généralement en deux étapes. La première étape, le raffinement du maillage polygonal, consiste à subdiviser les polygones courants de la surface augmentant ainsi sa densité. Lors de la seconde étape, la position des sommets est calculée ou réajustée selon qu'il s'agisse de nouveaux ou d'anciens sommets, pour s'approcher de la surface limite. Une technique de subdivision modifiant la position de ses anciens sommets est dite approximante, ou interpolante dans le cas contraire. La figure 2.4 illustre l'application de trois étapes successives de raffinement.

Il existe une multitude de schémas de subdivision se distinguant selon leurs règles de raffinement du maillage, et du calcul de la position des sommets. Le premier schéma pour une courbe remonte à la fin des années 1940 avec De Rham [DR47], alors que parmis les premiers

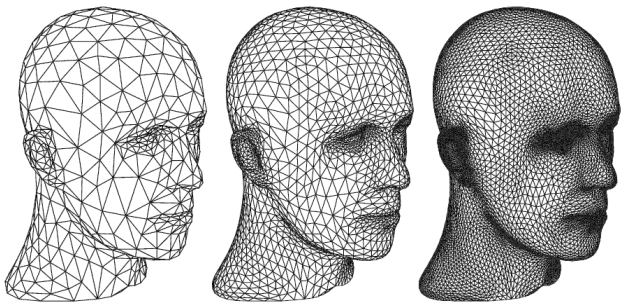


FIGURE 2.4 – Exemple de trois niveaux de subdivision successifs appliqués à un modèle de tête. Image tirée de [ZS00].

schémas de subdivision de surface, Catmull-Clark [CC78] et Doo-Sabin [DS78], apparurent à la fin des années 1970. Encore aujourd'hui, la subdivision de Catmull-Clark est très populaire et est probablement la variante la plus utilisée. Elle se caractérise par une subdivision, appliquée à des quadrilatères, de type *séparation de face*, par opposition à une *séparation de sommet*. Une face du maillage appartenant à un niveau donné est remplacée par quatre faces au niveau suivant. Pour calculer la position des sommets, trois différents noyaux de lissage sont appliqués selon leurs types. Il y en a un pour les nouveaux sommets situés au centre des faces, un pour les nouveaux sommets situés sur les arêtes des faces et finalement un pour les anciens sommets.

Le schéma de subdivision de Loop [Loo87] s'effectue sur un maillage triangulaire. Il est aussi de type *séparation de face*, mais à la différence du schéma de Catmull-Clark, il procède strictement en séparant les arêtes et donc, il n'insère pas de nouveaux sommets au centre des faces.

La figure 2.5 illustre la comparaison entre quatre schémas de subdivision appliqués à un maillage issu d'un cube.

D'autres variantes de schémas, le 4-8 [VZ01] et le $\sqrt{3}$ [Kob00], apportent une augmentation plus progressive du maillage. Contrairement aux schémas de Catmull-Clark et Loop où le nombre de faces quadruple à chaque niveau de raffinement, le $\sqrt{3}$ le triple (ou un facteur de $\sqrt{3}$ dans le cas de l'application d'une demi-étape), alors que le 4-8 le double. En plus d'imposer un nombre moins important de faces, ces deux techniques offrent la possibilité d'utiliser un maillage adaptatif, ce qui a pour effet d'augmenter la qualité du maillage pour un nombre déterminé de polygones. Aussi, la surface obtenue par la subdivision de type 4-8 possède une continuité plus élevée que ses congénères, soit C^4 versus C^2 et même C^1.

Pour contrôler le niveau de courbure de la surface, DeRose *et al.* [DKT98] proposent de limiter l'application des règles de lissage pour un certain nombre d'itérations, alors que Biermann

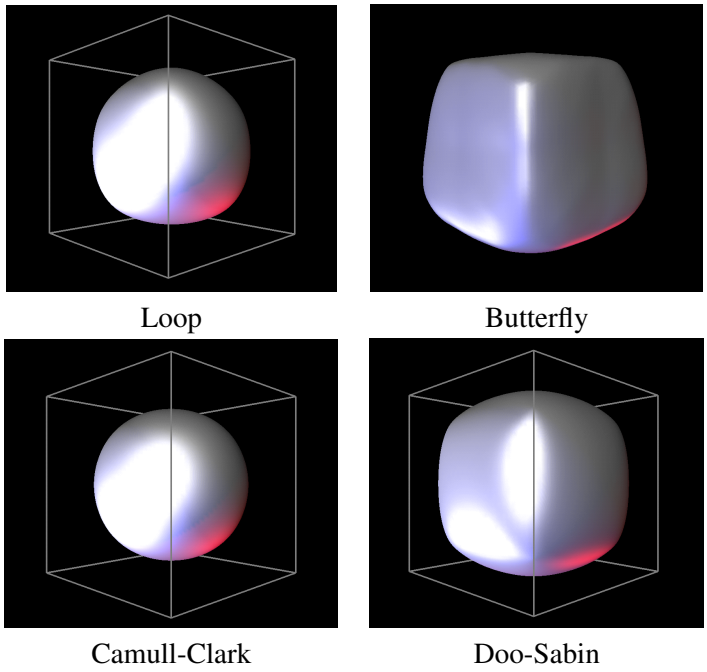


FIGURE 2.5 – Comparaison entre différentes méthodes de surface de subdivision appliquées au cube en fil de fer. Images tirées de [ZS00].

et al. [BLZ00] modifient les schémas de Catmull-Clark et de Loop pour que la subdivision tienne compte de normales assignées aux sommets du maillage original.

Les surfaces de subdivision sont généralement restreintes aux variétés 2D avec et sans frontières. Ying et Zorin [YZ01] présentent une variante du schéma de Loop éliminant cette restriction.

Comparativement aux surfaces courbes paramétriques, définies à la prochaine section, les surfaces de subdivision possèdent de nombreux avantages faisant en sorte qu'elles soient en gain de popularité :

- Traitement aisé pour des topologies complexes
- Numériquement stables
- Tessellation simple et comprise dans le processus
- Niveau de détail simple.

Pour plus d'information, un résumé complet des différentes techniques de subdivision peut être trouvé dans le cours offert par SIGGRAPH [ZS00] et le livre de Andersson et Stewart [AS10].

Surface courbe paramétrique

Une surface paramétrique est définie par une équation paramétrique possédant un domaine dans $\mathbb{R}^2$ et une image dans l'espace $\mathbb{R}^3$. Un exemple simple d'une telle surface est un plan x-y défini par l'équation 2.1.

$$S : \mathbb{R}^2 \to \mathbb{R}^3, S : (s,t) \to (s,t,0). \tag{2.1}$$

Un second exemple, cette fois-ci un tore de rayon intérieur b (le petit rayon) et de rayon extérieur a, est décrit par l'équation 2.2.

$$S : \mathbb{R}^2 \to \mathbb{R}^3, S : (s,t) \to ((a + b\cos t)\cos s, (a + b\cos t)\sin s, b\sin t) \tag{2.2}$$

où

$$s,t \in [0, 2\pi]$$
$$a,b \in \mathbb{R}.$$

La *spline* est une des principales catégories de surfaces paramétriques et majoritairement employée dans le domaine de la *conception assistée par ordinateur* (CAO). Elle est une surface paramétrique polynomiale par morceaux, passant près ou par un ensemble de points appelés les points de contrôle. Ce type de surface possède généralement la propriété de l'enveloppe convexe, c'est-à-dire que la surface est entièrement contenue dans l'enveloppe convexe formée par ses points de contrôle. Il existe de nombreuses variantes de ces surfaces courbes. Parmi elles, les plus employées sont probablement les Béziers pour leur simplicité et les NURBS pour leur flexibilité.

Introduites au début des années 1970 par Pierre Bézier [Bé70], les surfaces de Bézier peuvent être de différents degrés, mais plus souvent qu'autrement elles prennent la forme bicubique. Dans ce dernier cas, 16 points de contrôle définissent la forme de la surface. Un exemple d'une telle Bézier est illustré à la figure 2.6. L'équation 2.3 définit la surface de Bézier de degrés (m,n), égale à $(3,3)$ dans le cas d'une bicubique, où k_{ij} représente un point de contrôle.

$$S(s,t) = \sum_{i=0}^{n}\sum_{j=0}^{m} B_i^n(s)B_j^m(t)k_{ij} \tag{2.3}$$

où

$$B_i^n(u) = \binom{n}{i}u^i(1-u)^{n-i}$$
$$(s,t) \in [0,1]^2.$$

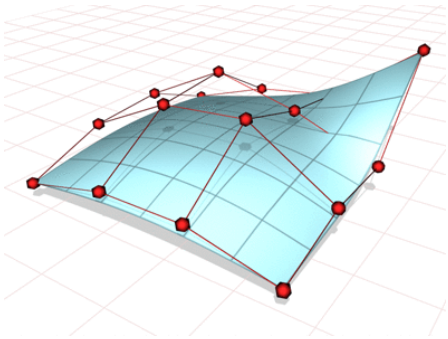
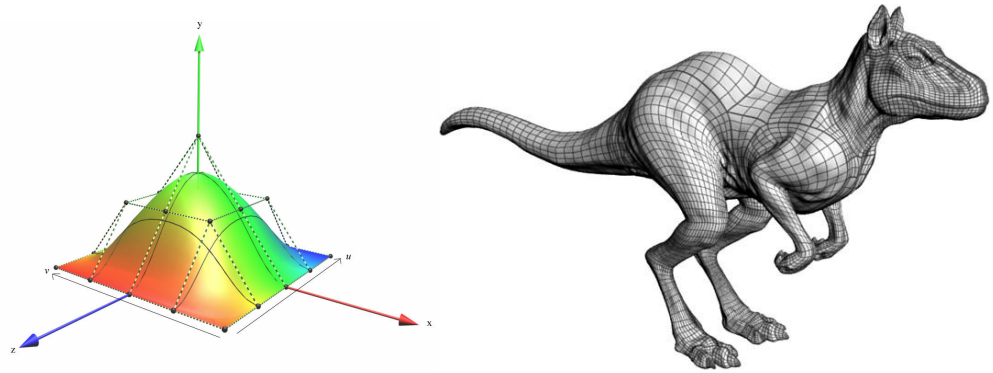


FIGURE 2.6 – Surface de Bézier.

Les surfaces NURBS (*non-uniform rational B-spline*) [Pie91, PT95] sont une généralisation des Béziers et des *B-splines*. Elles possèdent l'invariance par rapport aux transformations affines et perspectives, et comme la plupart des surfaces courbes avec points de contrôle, elles ont la propriété de l'enveloppe convexe. La figure 2.7a montre une surface NURBS, alors que la figure 2.7b montre un modèle complet créé avec des NURBS.



(a) (b)

FIGURE 2.7 – À gauche, une surface NURBS. À droite, un modèle formé de 90 surfaces NURBS (tiré du site *headus.com.au*).

Les T-splines [SZBN03] et ses variantes [LLWQ10] supportent l'ajout de sommets en T dans la définition de la surface. Ceci permet de diminuer le nombre de points de contrôle nécessaires. Une comparaison entre un modèle de NURBS et de T-splines est illustrée à la figure 2.8.

Il existe encore bien d'autres variantes de surfaces courbes paramétriques telles les surfaces de Coons [Coo67], les Hermites, les *B-splines*, les bilinéaires, etc.

Réseau de courbes

Une autre technique commune de description frontalière consiste à définir une surface à l'aide d'un ensemble de courbes 3D formant un réseau [CK83, SH90, SWZ04]. Le réseau de courbes

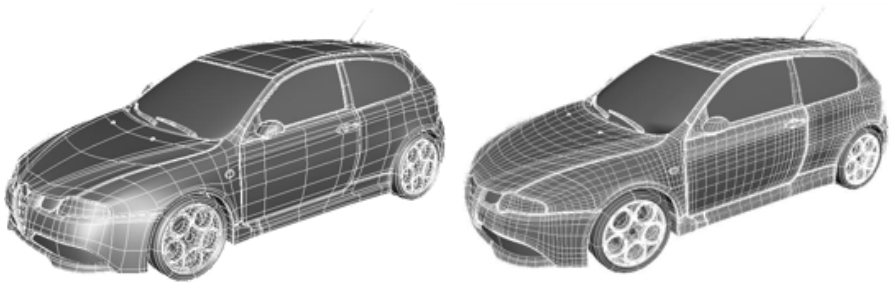


FIGURE 2.8 – À gauche, un modèle par T-splines. À droite, un modèle par NURBS. Image tirée du site *tsplines.com*.

est interpolé automatiquement pour produire une surface. Le réseau de courbes peut aussi servir à définir un volume si, par exemple, toutes les courbes de ce dernier sont fermées tel qu'illustré à la figure 2.9. Différents types de courbes 3D peuvent être employés, mais généralement les splines cubiques telles les NURBS sont les plus utilisées.

Schaefer *et al.*[SWZ04] utilisent les surfaces de subdivision pour reconstruire une surface formée d'un réseau de courbes. Ils présentent une extension au schéma de subdivision de Catmull-Clark afin de lui permettre de créer une surface interpolant le réseau de courbes cubiques *B-splines*.

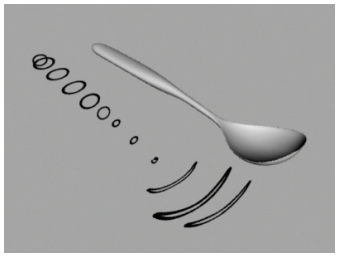


FIGURE 2.9 – Surface formée par un réseau de courbes.

Problèmes

Différents problèmes résultent de l'utilisation d'un modèle de représentation surfacique tel que décrit précédemment. L'intérieur d'un objet est généralement mal défini à moins que des restrictions soient imposées. Par exemple, chaque arête doit avoir un nombre pair de faces incidentes, comme dans le cas d'un objet dont la surface est une variété 2D sans frontière qui divise l'intérieur de son extérieur sans ambiguïté.

Bien que les modèles de représentations frontalières soient faciles à manipuler, ils demeurent des moyens de bas niveau pour exprimer des formes. On doit tenir compte de la topologie lors de modifications, ce qui rend ardue l'utilisation de ces modèles avec des techniques procédurales

de plus haut niveau. Même pour l'utilisation directe par un usager, la tâche n'est pas facile. Généralement pour tenter de remédier à ce problème, les différents logiciels de modélisation fournissent une panoplie d'opérations surfaciques. Mais le grand nombre d'opérations ne fait qu'augmenter la confusion des usagers.

Les surfaces courbes paramétriques telles les NURBS souffrent aussi d'un problème de robustesse lié à l'apparition de craques entre les différentes surfaces dues à la représentation numérique en point flottant. S'assurer de l'étanchéité de ces surfaces n'est pas un problème simple et beaucoup de recherche y est reliée.

2.1.2 Représentation par points

Une technique surfacique qui est en gain de popularité depuis la dernière décennie, est la représentation par points [RL00, PvBZG00, PKKG03]. Dans cette représentation, un objet est décomposé en points, parfois aussi appelés *surfels* (*surface elements*). Un point est, dans cette représentation, généralement de forme elliptique planaire possédant une normale, une couleur et possiblement d'autres paramètres comme des coordonnées de texture. La figure 2.10 illustre un exemple d'objet représenté par des points, ainsi qu'une vue rapprochée montrant les points formant la surface.

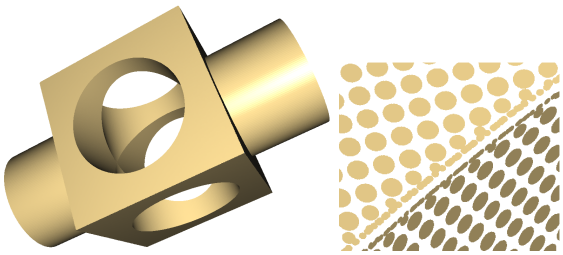


FIGURE 2.10 – À gauche, un objet représenté par un ensemble de points, ainsi qu'une vue rapprochée à droite. Images tirées de [AD03].

L'avantage de cette technique repose dans la simplicité de sa représentation. Aucune information topologique n'est contenue dans cette dernière, ce qui simplifie beaucoup les différentes opérations surfaciques [ZPKG02]. Même les opérations de CSG (*constructive solid geometry*), difficiles à appliquer sur un maillage, deviennent presque triviales [AD03]. Un exemple d'opérations booléennes exécutées sur un modèle par points est montré à la figure 2.11.

La gestion des niveaux de détail est aussi simple et efficace [PGK02], puisqu'elle n'a pas à se soucier, encore une fois, de la topologie de la surface. Cette caractéristique permet de simplifier des surfaces complexes non reliées entre elles comme les branches d'arbres, ce que

peu d'algorithmes de simplification peuvent se vanter de bien réussir.

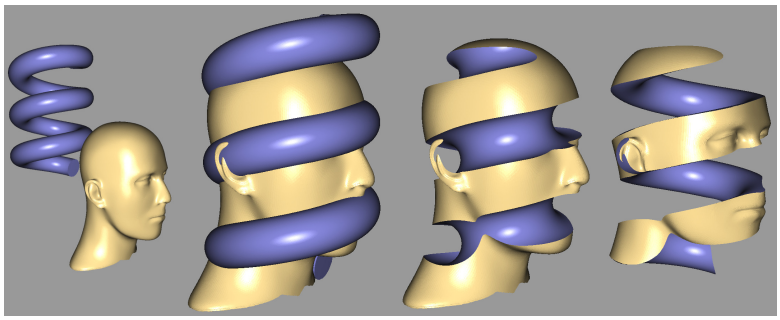


FIGURE 2.11 – Opérations de CSG (union, différence et intersection) appliquées à un modèle de points. Image tirée de [AD03].

La représentation par points est aussi efficace afin d'afficher des modèles complexes. Dachsbacher *et al.* [DVS03] présentent un algorithme permettant d'effectuer le calcul d'affichage complètement sur carte graphique. De plus, la transition à un maillage triangulaire y est aussi présentée pour fin d'optimisation.

Ce type de représentation bénéficie de beaucoup d'avantages comparativement à bien d'autres techniques. Toutefois, certains de ses désavantages sont non négligeables. La simplicité de cette représentation se paie en mémoire. En effet, afin de bien représenter une surface, une densité très élevée de points doit être utilisée, et ce même si la surface est simple comme dans le cas d'un cube. De plus, bien que le fait de ne pas avoir de topologie soit avantageux pour beaucoup d'opérations, l'information topologique absente dans cette représentation peut être utile dans plusieurs cas (e.g. calculer une paramétrisation de surface par propagation).

2.1.3 Représentation volumique

Contrairement aux représentations surfaciques, les représentations volumiques définissent un objet non pas par sa surface, mais par le volume qu'il occupe. Ces représentations ont l'avantage d'être plus réalistes au point de vue physique et permettent de leur attacher des caractéristiques telles que la masse, la densité, des propriétés de matériaux, etc.

Quadriques naturelles et superquadriques

Habituellement utilisées en combinaison avec d'autres techniques (CSG, volume implicite, etc.) puisque l'ensemble des formes représentables est limitée, les quadriques et superquadriques sont des surfaces implicites qui séparent l'espace en deux sections, intérieure et extérieure.

Les quadriques sont des surfaces implicites définies par une équation du deuxième degré et comprennent entre autres les formes suivantes : les ellipsoïdes, les hyperboloïdes à une et deux feuilles, les paraboloïdes elliptiques et les paraboloïdes hyperboliques. La forme normalisée et centrée à l'origine d'une quadrique est donnée par l'équation 2.4.

$$f(x,y,z) = \pm\frac{x^2}{a^2} \pm \frac{y^2}{b^2} \pm \frac{z^2}{c^2} = 1. \tag{2.4}$$

Toute quadrique peut être représentée par un produit sphérique selon deux paramètres angulaires. Par exemple, l'équation 2.5 représente le produit sphérique d'un ellipsoïde.

$$f(\theta,\phi) = \begin{bmatrix} a\cos\theta\cos\phi \\ b\cos\theta\sin\phi \\ c\sin\theta \end{bmatrix}, \quad \begin{array}{c} \theta \in [-\pi/2, \pi/2] \\ \phi \in [-\pi, \pi[. \end{array} \tag{2.5}$$

Les superquadriques [Bar81] (voir la figure 2.12) sont dérivées des quadriques. Elles sont obtenues en ajoutant des exposants aux termes trigonométriques, ce qui permet de contrôler l'arrondissement de leurs formes. L'équation 2.6 est la superquadrique construite à partir d'un ellipsoïde.

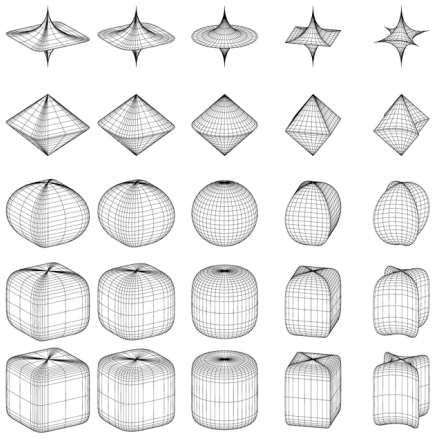


FIGURE 2.12 – Variation de formes de superquadriques.

$$f(\theta,\phi) = \begin{bmatrix} a\cos^{e1}\theta\cos^{e2}\phi \\ b\cos^{e1}\theta\sin^{e2}\phi \\ c\sin^{e1}\theta \end{bmatrix}, \quad \begin{array}{c} \theta \in [-\pi/2, \pi/2] \\ \phi \in [-\pi, \pi[. \end{array} \tag{2.6}$$

Volumes implicites

Introduits par Blinn [Bli82], les volumes implicites définissent une fonction de densité dans l'espace 3D. La surface de ces volumes, appelée iso-surface, est l'iso-contour pour lequel on a choisi une valeur de densité.

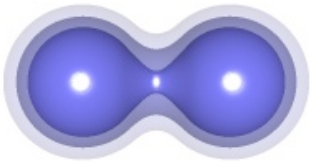


FIGURE 2.13 – Volumes implicites pour différentes valeurs de densité.

Généralement la fonction de densité est construite à partir de la sommation de fonctions simples comme la distance à un point. La figure 2.13 est un exemple d'une telle fonction de densité construite comme la sommation de deux fonctions de distance à un point. Pour avoir plus de contrôle sur l'expression de la fonction de densité, la distance à un squelette [CH01, BGA04] peut être utilisée. Wyvill *et al.* [WGG99] introduisent le *blobtree* définissant la fonction de densité à l'aide d'un arbre d'opérations.

L'affichage d'un volume implicite peut être directement effectué par lancé de rayons. Toutefois, pour fin d'affichage rapide, il est nécessaire de construire un maillage polygonal approximant la surface. De nombreux algorithmes de polygonisation existent. Le premier de ces algorithmes et le plus célèbre est le *marching cube* [LC87, Blo88]. Il consiste à subdiviser en une grille régulière un volume de l'espace que l'on veut polygoniser et à y évaluer la fonction de densité à chacun de ses sommets. La surface est ensuite reconstruite dans toutes les cellules dont les valeurs de densité aux sommets sont d'une part et d'autre de la valeur d'iso-surface. La position exacte de la surface à l'intérieur d'une cellule dépend des densités à ses sommets.

Beaucoup de variantes ont vu le jour pour tenter de parer aux lacunes du *marching cube*. Certaines des techniques s'attaquent au problème d'ambiguïté [MSS94, LLVT03] de la surface à l'intérieur d'une cellule, d'autres à la résolution nécessaire pour obtenir une reconstruction de qualité en effectuant l'échantillonnage de façon hiérarchique [SZK95, JLSW02]. Finalement, certains algorithmes [KBSS01, JLSW02] permettent l'évaluation de traits surfaciques vifs (*sharp features*). Pour y arriver, le *dual contouring* [JLSW02, SW03, ZHK04] augmente l'information contenue dans chaque cellule de la grille avec les points d'intersection précis de la surface avec les arêtes des cellules, ainsi que leurs normales à ces endroits.

Avec le *cubical marching square* [HWC+05], Ho *et al.* proposent de régler tous les problèmes précédents (ambiguïtés, échantillonnage adaptatif, traits vifs) en transférant le problème d'extraction de surface en 2D. Pour ce faire, la polygonisation d'une cellule est reconstruite en dépliant cette dernière et en travaillant sur les six faces obtenues.

Une variante des surfaces implicites définies par une fonction de densité est de représenter les volumes comme un champ de distance à la surface la plus près [GPRJ00, PF01]. Un exemple 2D, utilisant un champ de distance défini hiérarchiquement, peut être observé à la figure 2.14. Tout comme dans le cas d'utilisation d'une fonction de densité, le rendu peut être directement effectué par lancé de rayons, ou par l'affichage du maillage obtenu par la polygonisation de la surface [Gib98, dBVP+00].

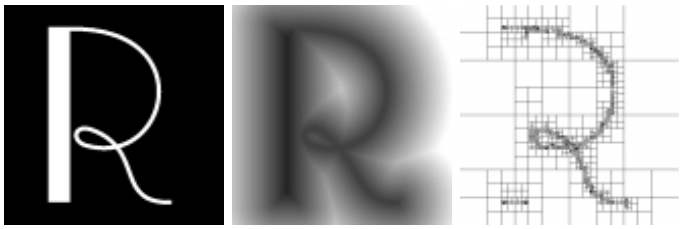


FIGURE 2.14 – Représentation par un champ de distance 2D hiérarchique tirée de [GPRJ00]. De gauche à droite, le modèle, le champ de distance et la subdivision hiérarchique associée.

En plus de la difficulté d'obtenir un maillage de qualité, d'autres problèmes sont reliés à la représentation implicite. Il peut être difficile de bien contrôler la forme de la surface. Entre autres, il y a risque de formation de renflements à la jonction de deux fonctions de densité. Dans ce dernier cas, Bernhardt *et al.* [BBCW10] proposent de limiter la région de fusion. Calculer une bonne paramétrisation de la surface n'est pas un problème trivial. Une solution partielle à ce problème est présentée par Schmidt *et al.* [SGW06] sous la forme d'une paramétrisation locale.

Voxels

Principalement utilisés dans le domaine de l'imagerie médicale, les *voxels* (*volumetric elements*) [KS86, Kau86] constituent la représentation volumique la plus simple. Par analogie aux pixels, les voxels divisent un volume en éléments réguliers de taille identique, alignés et distribués selon une grille régulière. La figure 2.15 illustre l'exemple d'un modèle courbe représenté par des voxels cubiques. Puisqu'un voxel ne peut être partiellement plein, bien qu'une densité peut lui être attribuée, la qualité de la description dépend directement de la résolution de celui-ci. Le problème est que la croissance mémoire en voxels augmente de façon cubique (n^3).

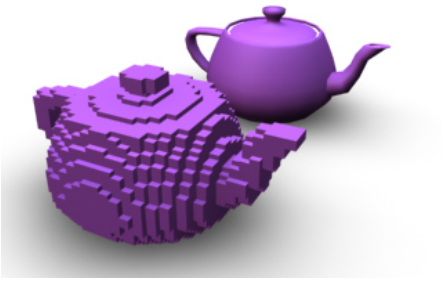


FIGURE 2.15 – Représentation par voxels de la théière de Newell.

L'affichage d'un modèle sous forme de voxels peut s'effectuer par lancé de rayons, par projection ou encore en polygonisant la surface. Crassin *et al.* [CNLE09] ainsi que Laine et Karras [LK10] proposent une version accélérée sur GPU.

Volume par balayage

Le déplacement d'une surface selon une trajectoire dans l'espace 3D définit ce qu'on appelle un volume par balayage. Lorsque la trajectoire est restreinte à un déplacement linéaire, on nomme *extrusion* cette classe de volumes, et dans le cas où la trajectoire consiste en une rotation autour d'un axe, il s'agit d'une *surface de révolution*.

Les cylindres généralisés [Bin71] se définissent par une trajectoire quelconque et aussi par la flexibilité de changer la forme déplacée au cours du déplacement. Ce changement peut être de l'ordre d'un simple changement d'échelle ou encore plus complexe, comme une modification des points de contrôle de la surface balayée dans le cas d'une surface courbe. Un exemple de volume par balayage est présenté à la figure 2.16.

Snyder et Kajiya [SK92] démontrent la puissance expressive de cette représentation en l'incluant dans ce qu'ils appellent la modélisation générative. Ce mode de représentation consiste à décrire les objets par un ensemble d'opérations, et non seulement par une description du résultat final comme par exemple dans le cas des voxels ou d'un maillage. Avec cette technique, ils décrivent des scènes complètes ayant un haut niveau de complexité pour l'époque.

Le maillage de la surface peut être obtenu en échantillonnant la position de la surface de balayage au cours de la trajectoire et en la joignant à la position précédente. Bien que simple, cette technique peut générer une surface s'intersectant elle-même. L'utilisation d'une surface implicite [SLL94] peut être employée afin de corriger ce problème. L'utilisation d'une surface implicite permet aussi de retrouver la surface dans des cas beaucoup plus complexes lorsque la surface de balayage est remplacée par un volume [KVLM03].

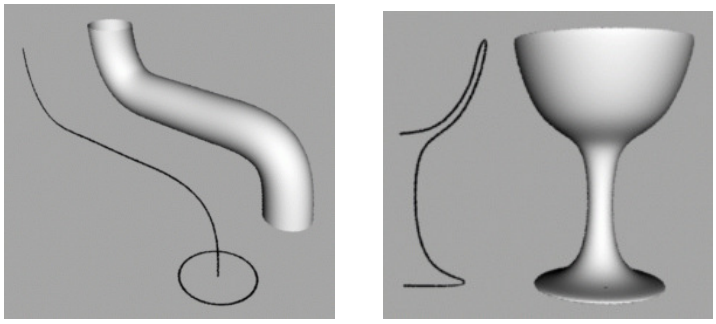


FIGURE 2.16 – Volume par balayage. À gauche, un cylindre généralisé. À droite, une surface de révolution.

Winter et Chen [WC02] proposent de remplacer la surface ou le volume de balayage par une ou plusieurs images. Ceci leur permet de modéliser des volumes hétérogènes complexes.

Opérations de CSG

Combiner des objets pour en former de nouveaux est un procédé intuitif de modélisation et c'est justement en quoi consistent les opérations de CSG (*constructive solid geometry*) [RV77]. Les opérations possibles sur les volumes sont les opérations booléennes d'ensemble, soient l'union, la différence et l'intersection. Ces opérations sont effectuées à la base sur des primitives généralement simples telles des cubes, des sphères, des cylindres, des cônes, etc. Ces primitives peuvent être exprimées sous différents formats (surface implicite, maillage fermé, points, etc.). Pour être valide, il suffit que l'espace intérieur et extérieur soit défini sans ambiguïté.

Une illustration des trois opérations booléennes effectuées sur deux primitives simples est fournie à la figure 2.17.

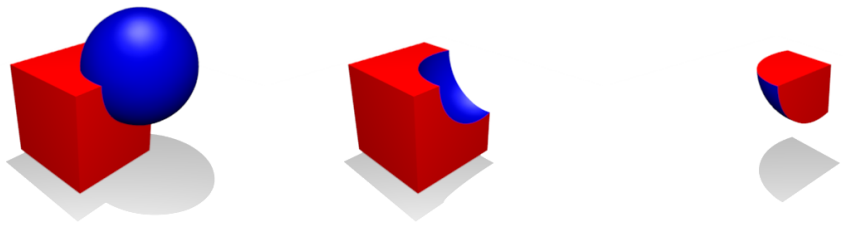


FIGURE 2.17 – Opérations de CSG. De gauche à droite, union, différence et intersection.

Un objet complexe peut être exprimé à l'aide d'une représentation arborescente d'opérations comprenant les opérations booléennes d'ensemble ainsi que des opérations de transformations géométriques telles que la translation, la rotation et le changement d'échelle (voir la figure 2.18 pour un exemple).

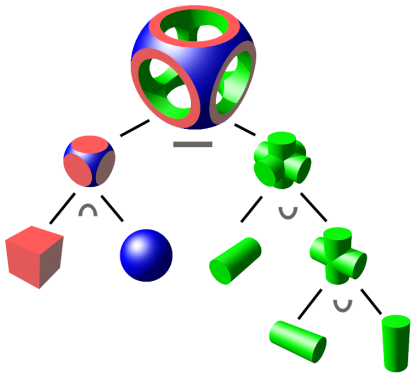


FIGURE 2.18 – Arbre d'opérations de CSG.

Comme toutes les représentations volumiques, l'affichage peut être effectué directement par lancé de rayons ou encore par le rendu du maillage polygonal approximant le volume. L'algorithme de polygonisation diffère selon le type de représentation utilisé à la base comme primitive. Dans le cas de primitives polygonales, un arbre BSP (*binary space partitioning*) [TN87] peut être employé. Dans une première étape, le maillage polygonal des primitives est converti en arbre BSP. Cet arbre divise les volumes en sous-volumes convexes selon des plans de séparation. Ces plans subdivisent, dans un même temps, le maillage des primitives en polygones convexes qui sont assignés aux noeuds de l'arbre. Dans une seconde passe, l'algorithme décide quels sont les polygones que l'on garde et quels sont ceux que l'on doit éliminer. Cette décision est basée selon le fait que les polygones appartiennent ou non à une région contenue à l'intérieur du nouveau volume formé selon le type d'opération effectuée. Un désavantage marqué de cette technique est le nombre élevé de polygones créés lors de la séparation en zones convexes. Pour remédier à cette situation, on peut procéder à une passe de fusion des polygones appartenant à un même plan.

Un autre algorithme de polygonisation se rapportant à un maillage polygonal proposé par Laidlaw *et al.* [LTH86] consiste en deux étapes majeures. La première étape effectue le découpage des polygones s'intersectant entre eux pour éliminer toute intersection. La deuxième consiste simplement à éliminer les faces contenues à l'intérieur du volume.

Dans le cas où le volume des primitives est délimité par des surfaces courbes, la difficulté de l'algorithme est d'autant plus élevée qu'il soit complexe de calculer l'intersection entre deux surfaces. Krishnan *et al.* [KMG+01] présentent une méthode rapide de maillage pour un arbre CSG construit à partir de surfaces de Bézier.

Toutes ces techniques souffrent d'importants problèmes de robustesse dus à la représentation et aux calculs effectués par point flottant. Il est extrêmement difficile de garantir la validité

du modèle obtenu. Pour remédier à ce problème, Keysar *et al.* [KCF+02] proposent d'utiliser le calcul exact avec marges d'erreurs connues. Il y a toutefois un prix à payer en plus de la complexité d'implémentation. Le nouvel algorithme est de deux ordres de magnitude plus lent que les autres. D'autres détails sur les algorithmes de CSG sont donnés à la section 3.2.1.

2.2 Modélisation procédurale

Définie par un ensemble de règles et de procédures, la modélisation procédurale a pour but d'automatiser la génération d'objets. Cette automatisation permet de produire des scènes d'une complexité extrême non-envisageable par tout autre procédé, et offre aussi la possibilité de simuler de façon réaliste différents phénomènes et types d'objets. Plusieurs grandes catégories existent et sont couvertes dans la prochaine section dont les grammaires (systèmes-L et grammaires de formes), les fractales, les tuilages et les langages. Beaucoup d'autres techniques plus spécifiques existent ou peuvent être combinées ensemble (par exemple la simulation et les fractales).

2.2.1 Systèmes-L

Principale méthode de modélisation procédurale, les systèmes-L ont été élaborés par Lindenmayer [Lin68] afin de représenter le développement des plantes. Un système-L est une grammaire formelle[1] basée sur les grammaires de Chomsky. Il possède donc un ensemble de symboles, de règles de réécriture et un axiome de départ (un symbole). Cependant, contrairement aux grammaires de Chomsky, les règles de réécriture ne sont pas séquentielles, mais plutôt parallèles, c'est-à-dire que toutes les règles applicables à une chaîne sont exécutées en même temps, et non pas une après l'autre. Un exemple simple de grammaire peut être défini comme suit :

$$\begin{aligned} symboles: &\quad AB \\ axiome: &\quad A \\ règles: &\quad A \rightarrow B \\ &\quad\quad\; B \rightarrow AB \end{aligned}$$

Le résultat de l'exécution de quelques itérations de la grammaire précédente est comme suit :

[1] Plus d'information sur la définition des grammaires est donnée à la section 5.1.1 du chapitre 5.

$$
\begin{aligned}
0 &: \; A \\
1 &: \; B \\
2 &: \; AB \\
3 &: \; BAB \\
4 &: \; ABBAB \\
5 &: \; BABABBAB \\
&\vdots
\end{aligned}
$$

Il est intéressant de noter que la longueur de la chaîne produite par cette grammaire génère la séquence de Fibonacci.

Afin de donner une représentation géométrique à un système-L, les symboles d'une chaîne peuvent être interprétés comme des commandes contrôlant une tortue à la LOGO. Par exemple, dans le cas d'une interprétation 2D, le symbole F peut indiquer le déplacement de la tortue vers l'avant d'un pas, alors que les symboles $+$ et $-$ indiquent respectivement une rotation à gauche et à droite d'un nombre préfixé de degrés.

Il existe différentes variantes et extensions possibles aux systèmes-L. Parmi celles-ci on retrouve les variantes suivantes :

parenthésée : Ajoute deux symboles spéciaux [et] pour empiler et dépiler l'état courant dans une pile.

stochastique : Introduit de la variation dans l'application des règles en leur assignant une probabilité. En somme, un symbole peut être remplacé par plusieurs règles et chacune d'elles possède une probabilité d'exécution.

sensible au contexte : L'application d'une règle peut dépendre des symboles dans son contexte, c'est-à-dire les symboles situés avant et après dans la chaîne.

paramétrique : Associe des paramètres numériques aux symboles, ce qui permet entre autres de produire des résultats continus (longueur de pas).

Végétation

Le domaine principal d'application des systèmes-L est la création de végétation. Beaucoup de travaux ont été effectués sur ce sujet. Certains portent sur la création de plantes et de fleurs

[FHP89, PL90, FJP04], d'autres sur leur croissance [PHM93], ou encore sur la génération de feuilles [HPW92]. Il y a aussi la production d'arbres [AK84, PL90, PHHM97, Bot11] (voir la figure 2.19), en passant même par l'art topiaire [PJM94] modifiant les systèmes-L pour tenir compte de la réaction à l'élagage.

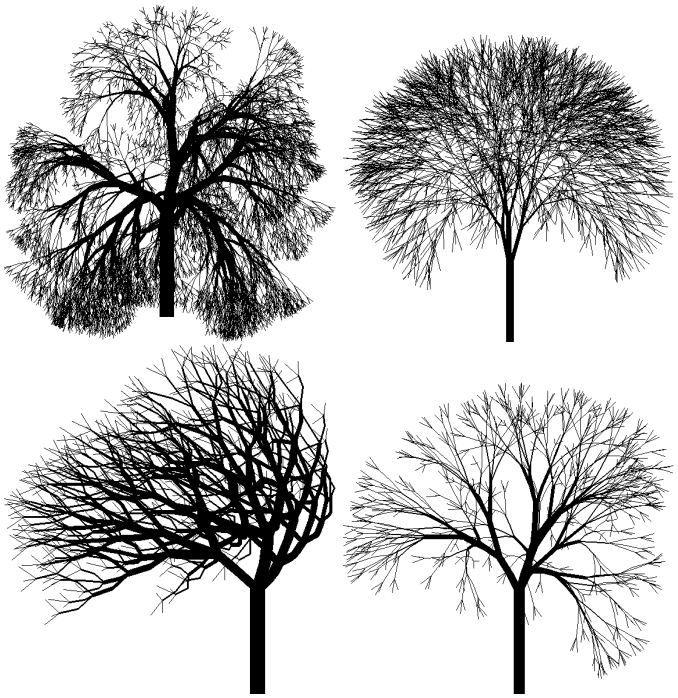


FIGURE 2.19 – Arbres ternaires procéduraux obtenus par système-L. Images tirées de [PL90].

Mech et Prusinkiewicz [MP96] introduisent une nouvelle variante appelée *open L-system* permettant l'échange d'informations entre l'environnement et la végétation simulée. Ils approximent différents phénomènes tels la limitation de la pousse de branches due aux collisions externes, la croissance de plantes compétitionnant pour un espace favorable, l'interaction des racines compétitionnant pour l'eau et les nutriments, et finalement la compétition inter et intra arbres pour l'éclairage. Un exemple de résultat obtenu de cette compétition pour la ressource d'éclairage est illustré à la figure 2.20.

Avec les systèmes-L, il est aussi possible de simuler des phénomènes à plus grande échelle, tels la distribution de la végétation dans une forêt pour former un écosystème. Deussen *et al.* [DHL+98] présentent une version de grammaire et un ensemble d'outils produisant une distribution uniforme de la végétation tout en tenant compte des désirs d'un usager. La figure 2.21 illustre un exemple de résultat obtenu par cette technique. Lane et Prusinkiewicz [LP02], quant à eux, apportent une solution au phénomène de groupement d'espèces.

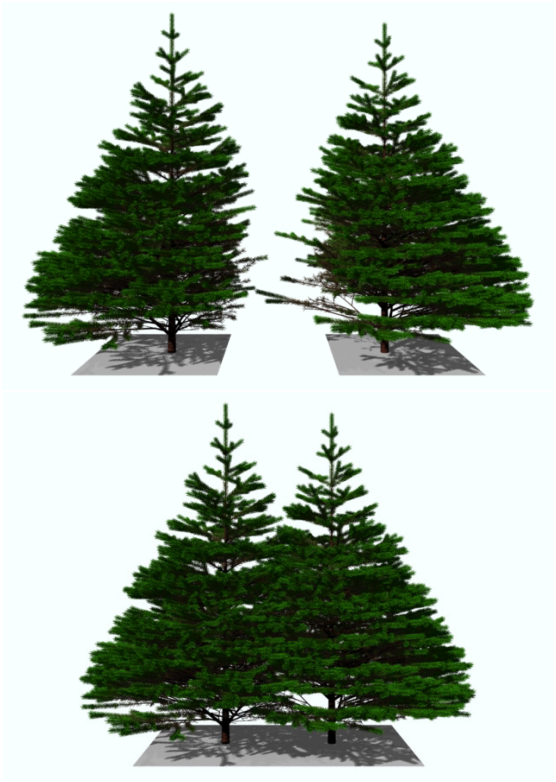


FIGURE 2.20 – Compétition pour la ressource d'éclairage entre deux conifères. La position de compétition est présentée en bas, alors que le haut présente la position déplacée. Images tirées de [MP96].

FIGURE 2.21 – Écosystème par système-L tiré de [DHL+98].

Villes

Bien que la majorité des travaux portant sur les systèmes-L se rapporte à la production de végétation, des recherches exploitent la puissance descriptive de ces grammaires pour produire d'autres types de modèles comme des routes et des édifices.

Parish et Müller [PM01] ont présenté des extensions aux systèmes-L pour pouvoir créer des villes comprenant leur réseau routier, leurs lotissements et leurs édifices. Différents types de cartes fournissent des informations de départ pour orienter l'apparence finale recherchée. Ces cartes d'entrée décrivent entre autres la frontière entre la terre et l'eau, l'élévation du terrain, la position de la végétation, la densité de population, le zonage (commercial, résidentiel, mixte), etc.

La création des routes utilise une variante de systèmes-L pouvant considérer les buts globaux (densité de population, positionnement des étendues d'eau, etc.) tout en tenant compte des contraintes locales (interdiction de croiser certaines zones, connexion des routes avoisinantes, etc.). La géométrie simplifiée des édifices est aussi obtenue par grammaire en utilisant l'union de volumes et l'extrusion de contours. La figure 2.22 illustre un exemple de routes ainsi que d'édifices formés par ce système.

FIGURE 2.22 – Ville procédurale créée par système-L. Images tirées de [PM01].

Coelho *et al.* [CdSF05] utilisent un système-L paramétrique et stochastique afin de produire

une description arborescente d'une ville. Les villes ainsi formées sont cependant très simples et peu détaillées.

Édifices

Le système-FL est introduit par Marvie *et al.* [MPB05] pour faciliter la création d'édifices ainsi que de scènes complexes. Dans cette modification du système-L, les symboles ne sont pas seulement de simples caractères interprétés à partir de la chaîne finale, mais sont plutôt des fonctions pouvant être exécutées à chaque itération du processus de réécriture des symboles. En plus, le système-FL offre la possibilité d'exécuter des règles de réécriture de façon séquentielle, et non seulement parallèle, en ajoutant un symbole de synchronisation qui lorsque rencontré, exige la terminaison des règles de réécriture se rapportant aux symboles le précédant.

Avec ce système, ils réussissent à instancier des objets créés séparément (porte, fenêtre, balcon, etc.) et à les positionner les uns par rapport aux autres. En combinant ainsi ces objets, ils arrivent à créer des extérieurs d'édifices simples.

Grammaires de formes

Similaires aux systèmes-L, à quelques différences près, les grammaires de formes[2] [SG71, Sti75, Gip75] sont majoritairement utilisées dans le domaine de l'architecture afin d'étudier les différents styles et aussi comme outil de création de plans schématiques. Leur transition au domaine de l'infographie ne s'est faite que plus récemment [WWSR03].

Les différences principales entre les systèmes-L et les grammaires de formes se résument en deux points. Les règles de réécriture sont appliquées séquentiellement, et non parallèlement comme dans une grammaire de Chomsky, et les symboles de la grammaire sont des formes et donc, possèdent un volume, une orientation et possiblement d'autres caractéristiques.

Wonka *et al.*, se basant sur les grammaires de formes, ont introduit les grammaires de partitionnement [WWSR03] pour aider à la modélisation des façades d'édifices. Cette grammaire permet de partir d'un bloc, de le diviser en étages, puis de les diviser à leur tour verticalement pour leur assigner une identité (porte, mur, fenêtre) qui seront encore selon le cas divisés, le tout pour former la façade d'un édifice.

[2]Plus d'information au sujet des grammaires de formes est donnée à la section 5.1.1 du chapitre 5.

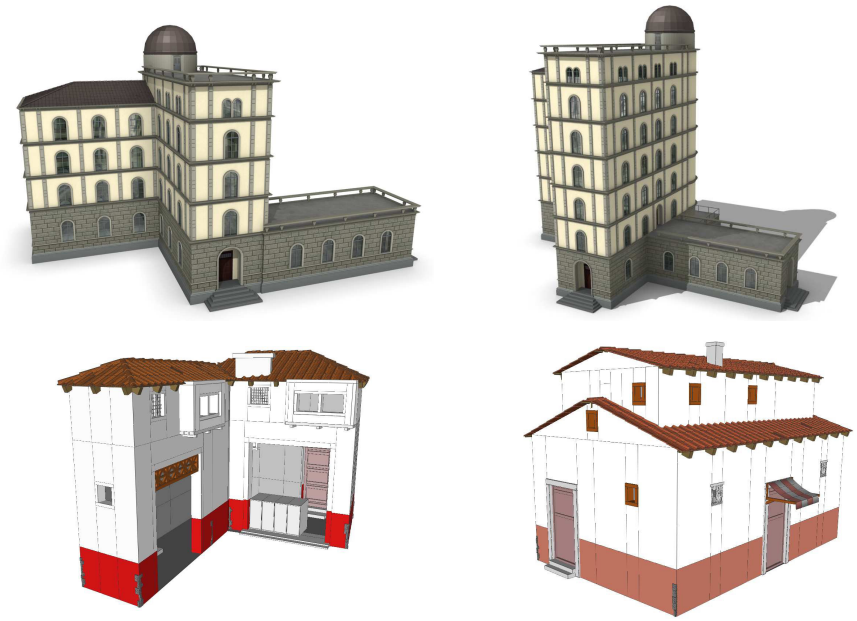


FIGURE 2.23 – Édifices procéduraux créés par grammaire de formes CGA [MWH+06].

Müller *et al.* [MWH+06] reprennent leurs travaux précédents sur la génération de villes (Müller) et de façades (Wonka) pour introduire la grammaire de formes CGA (*computer graphic architecture*). En plus de posséder toutes les caractéristiques des grammaires de formes standards, les règles de partitionnement [WWSR03] y sont ajoutées et améliorées. Aussi, les règles de réécriture ont accès aux requêtes d'occultation d'une forme en relation avec son environnement et peuvent dépendre de ces requêtes. Ces requêtes retournent comme résultat une occultation complète, partielle ou nulle. Ils s'en servent entre autres pour déterminer si une forme terminale doit être remplacée par une fenêtre dans le cas d'occultation nulle, un mur dans le cas d'occultation partielle, ou rien dans le cas d'occultation complète.

À l'aide de cette nouvelle grammaire, les façade d'édifices complexes sont créées pour différents styles architecturaux. Deux variantes de deux différents styles architecturaux sont présentés à la figure 2.23. Un logiciel complet [Pro11] basé sur ces travaux a vu le jour dernièrement.

Problèmes

Les techniques de modélisation utilisant les grammaires à leur base sont généralement très puissantes et très efficaces. À partir de quelques règles (généralement moins d'une centaine, et souvent moins d'une dizaine) et de quelques paramètres, on obtient une complexité très élevée. Cependant, il y a un prix à payer. Il est difficile de trouver de bonnes règles pour obtenir les

résultats escomptés et c'est encore plus vrai pour les artistes qui ne sont pas des programmeurs.

Il y a un second problème, mais cette fois relié indirectement aux grammaires. Ces techniques ne génèrent pas directement de la géométrie ; il faut interpréter leurs résultats pour l'obtenir. Et il est généralement difficile de créer dans ces situations de la géométrie possédant un maillage de qualité sans interpénétrations et ni craques.

2.2.2 Fractale

Une forme fractale est construite récursivement par autosimilitude, généralement infinie. Ces formes présentent donc une apparence similaire à des niveaux d'échelle différents.

Certains objets dans la nature peuvent être représentés sous une forme fractale. On compte, entre autres, les terrains, certaines plantes comme la fougère, les nuages, les flocons de neige, les réseaux de rivières et aussi jusqu'à un certain point les arbres.

Terrain

La notion de fractale a été rapidement employée dans le domaine de la modélisation, et principalement pour la construction de terrains procéduraux (voir la figure 2.24). Introduite sous la forme du mouvement brownien fractionnaire [Man78], elle est encore aujourd'hui utilisée sous diverses variantes.

Le mouvement brownien fractionnaire (fBm) a été utilisé la première fois par Mandlebrot après qu'il eut observé la similitude entre le tracé 2D du mouvement dans le temps et le profil d'une montagne. Le fBm est composé d'une somme infinie de fonctions sinus de fréquence discrète, de phase aléatoire et d'amplitude reliée à sa fréquence par un facteur de $f^{-\beta}$, où f est la fréquence moyenne de base et $\beta \in [1, 3]$. Bien entendu, pour leur utilisation, seul un ensemble discret contraint de sommes de fonctions est utilisé, généralement de trois à huit suffisent.

On peut noter au moins cinq façons différentes de création de terrains utilisant le mouvement brownien. Il y les cassures de poisson (*poisson faulting*) [Vos85], le filtrage de Fourier, le point milieu [Lew87, PH93], les additions successives aléatoires et la sommation limitée de bruits [EMP+02].

La plupart des techniques fractales utilisent à la base le mouvement brownien fractionnaire. Cependant, plusieurs problèmes lui sont attribués. Les formes et les détails caractéristiques produits par les phénomènes d'érosion ne sont pas simulés. L'érosion a un apport important dans l'apparence des terrains. Elle est responsable du fait que les montagnes sont plus affûtées

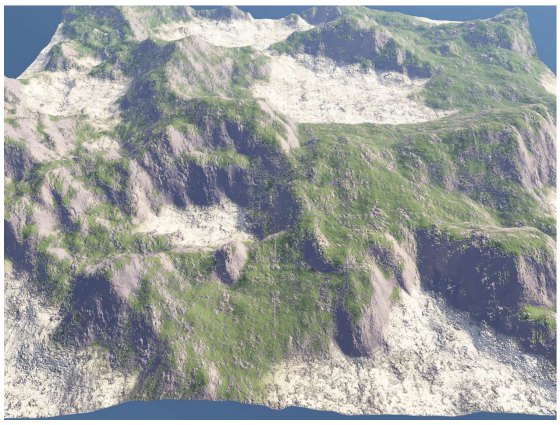


FIGURE 2.24 – Terrain procédural par fractale tiré de [Dac06].

alors que les plaines sont arrondies. Un autre problème du fBm est que la géométrie produite est statistiquement symétrique à l'horizontal, c'est-à-dire qu'il y a autant de pointes que de creux, ce qui n'est généralement pas le cas dans la réalité.

Plusieurs variantes ont été introduites afin d'améliorer cette situation. Une comparaison de quelques variantes est illustrée à la figure 2.25.

Il y a d'abord le fBm de base défini par l'équation 2.7 où n est le nombre d'octaves, N est la fonction de bruit, L est la distance entre deux fréquences successives et H est l'incrément fractal.

$$fBm(x) = \sum_{i=0}^{n-1} N(x \cdot L^i) L^{-iH} \tag{2.7}$$

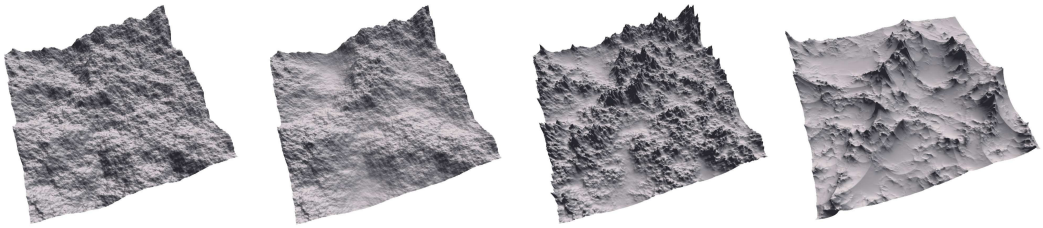
Il y a aussi le multi-fractal représenté par l'équation 2.8 où O est une valeur de décalage. Contrairement au fBm qui est une sommation de fonctions, le multi-fractal est une multiplication de fonctions.

$$M(x) = \prod_{i=0}^{n-1} (N(x \cdot L^i) + O) L^{-iH} \tag{2.8}$$

Encore beaucoup d'autres variantes ont été produites telles le multi-fractal hybride et le multi-fractal strié qui sont des variantes des deux précédentes.

Pour ajouter des détails, Prusinkiewicz et Hammel [PH93] incluent la création de rivières dans la formation du terrain, alors que Gamito *et al.* [GIM03] appliquent des déformations sur la surface du terrain à l'aide d'un champ de vecteurs pour créer des surplombs. La figure 2.26 illustre le résultat obtenu dans ce dernier cas.

Plusieurs travaux sur la génération de terrains se sont tournés vers la simulation des



(a) fBm (b) Multi-fractal (c) Multi-fractal hybride (d) Multi-fractal strié

FIGURE 2.25 – Terrains procéduraux tirés de [Dac06].

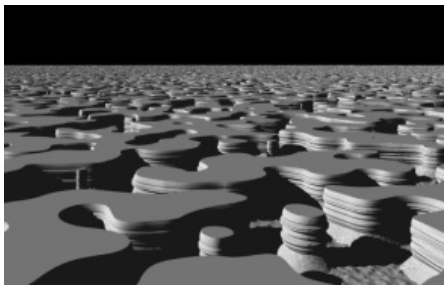


FIGURE 2.26 – Terrain procédural avec surplombs. Image tirée de [GIM03].

phénomènes d'érosion comme solution au problème de réalisme. Musgrave *et al.* [MKM89] simulent l'érosion hydraulique et thermale pour obtenir des terrains plausibles, alors que Benes et Arriaga [BA05b] les utilisent pour produire des montagnes à sommets plats.

Pour mieux simuler l'érosion du terrain par le passage de l'eau, le transport et le dépôt des matériaux, Benes *et al.* [BF02, BTHB06] se basent sur les équations de Navier-Stokes. Le résultat d'une de ces simulations est illustré à la figure 2.27. Dans un autre cas de simulation, Peytavie *et al.* [PGMG09a] utilisent un modèle hybride pour encoder leur terrain composé d'un modèle volumique par couches (supportant air, eau, sable et roches) et d'un modèle implicite. Le terrain peut être édité interactivement et aussi recevoir des simulations locales d'érosion et de dispersion de matériaux pour créer des détachements de roches. La figure 2.28 illustre des exemples de terrains complexes avec arches possibles avec ce système.

Dachsbacher [Dac06] propose, quant à lui, une nouvelle méthode basée sur les techniques de synthèse de texture [NA03] afin de créer des terrains possédant des caractéristiques similaires à des terrains existants.

Il propose aussi l'ajout de rochers au terrain pour augmenter la qualité du modèle. Ces rochers sont formés en utilisant l'enveloppe convexe d'un ensemble de points qu'il subdivise et déplace récursivement. Quelques rochers ainsi créés apparaissent à la figure 2.29.

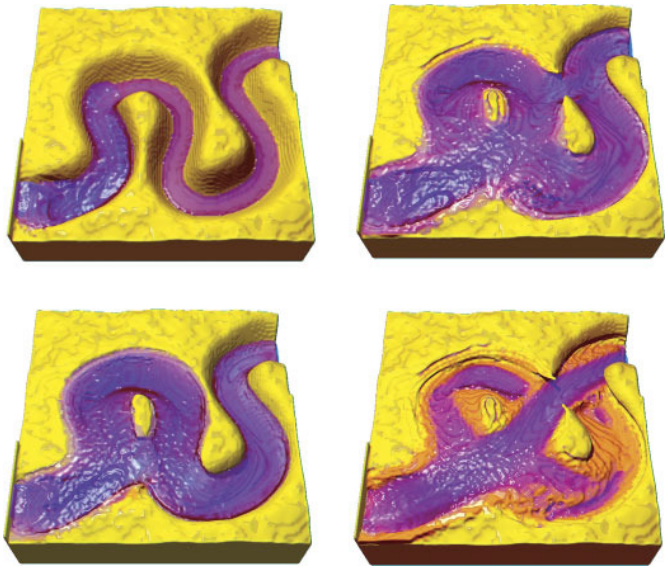


FIGURE 2.27 – Terrain modifié par érosion hydraulique avec les équations de Navier-Stokes. Images tirées de [BTHB06].

FIGURE 2.28 – Terrains procéduraux. Images tirée de [PGMG09a].

FIGURE 2.29 – Rochers créés procéduralement [Dac06].

2.2.3 Modélisation par tuiles

Tuiles de Wang

Introduites en 1960 par Hao Wang, les tuiles de Wang [Wan60] sont un ensemble de carrés de taille identique ayant une couleur associée à chacune de leurs arêtes. Chacune de ces tuiles peut se connecter à une autre tuile seulement si leurs arêtes communes possèdent la même couleur. Le problème standard, relié à ces tuiles, consiste à trouver un ensemble restreint de tuiles pour lesquelles un pavage apériodique du plan est possible. La figure 2.30 illustre un exemple de pavage d'une zone restreinte à partir d'un ensemble de huit tuiles.

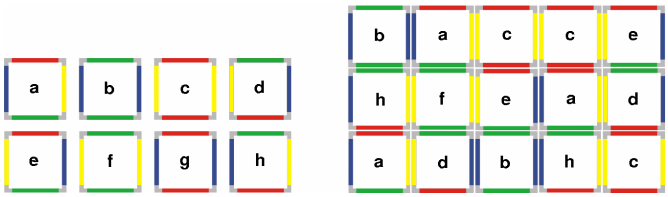


FIGURE 2.30 – Tuiles de Wang et pavage [CSHD03].

L'utilisation des tuiles de Wang a été reprise en infographie comme moyen de génération, premièrement dans le domaine de la génération de textures [CSHD03], puis pour produire un échantillonnage de fonctions 2D sur un plan [CSHD03, KCODL06].

Un ensemble de huit tuiles de Wang (voir la figure 2.31) est utilisé pour former une texture infinie et apériodique. La texture résultante ne démontre aucun problème de couture puisque chaque tuile de la texture peut se joindre sans artefacts à une autre tuile voisine partageant une arête possédant le même code de couleur.

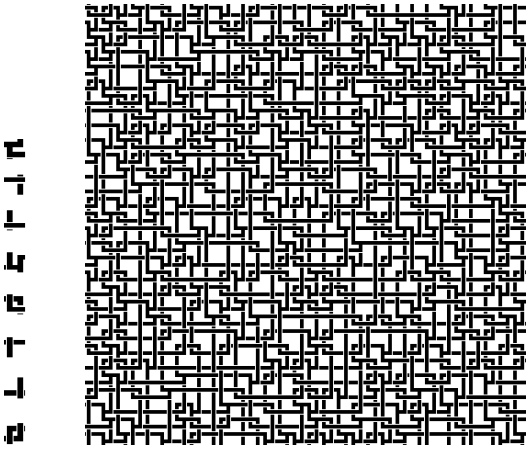


FIGURE 2.31 – Tuiles de Wang et texture générée à partir de ces tuiles. Image tirée de [CSHD03].

Comme illustré à la figure 2.32, la même technique décrite précédemment peut être employée

pour générer un échantillonnage uniforme possédant la propriété de bruit bleu. Une variante hiérarchique [KCODL06] peut aussi être utilisée afin de produire un échantillonnage sur une fonction de densité non-uniforme.

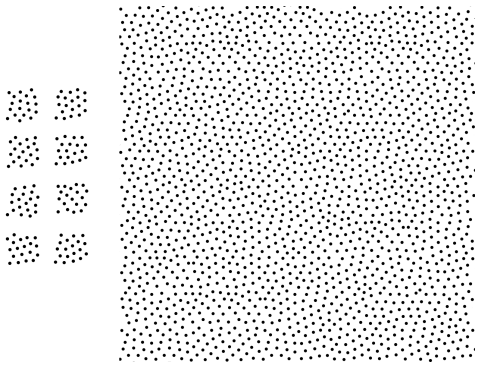


FIGURE 2.32 – Tuiles de Wang et échantillons générés à partir de ces tuiles. Image tirée de [CSHD03].

Tuiles 3D

Peytavie *et al.* [PGMG09b] se servent de pavage apériodique de cubes [LD06] pour générer des piles de roches complexes réalistes de façon efficace. Le pavage par cubes de coin est un système de tuilage où la couleur est associée aux coins des cubes et ceux-ci peuvent être voisins seulement si leurs coins avoisinants partagent les mêmes couleurs. Une illustration de ces piles de roches est présentée à la figure 2.33.

Édifices

Pour la création du film *King Kong* [Whi06], le studio d'effets spéciaux *Weta* a créé une version très simplifiée d'un système de tuiles afin de reproduire procéduralement la ville de New York (voir la figure 2.34) pour l'époque des années 1930.

FIGURE 2.33 – Piles de roches générées par tuilage apériodique. Images tirées de [PGMG09b].

FIGURE 2.34 – New York des années 1930 dans le film *King Kong*.

Les formes simplifiées des extérieurs des édifices ont été construites automatiquement par extrusion à partir des plans de la ville. L'information des plans de la ville contenait la hauteur et la forme des différents étages des édifices. Chaque maillage des édifices a ensuite été séparé automatiquement en cellules de tailles uniformes selon la façade. Ensuite, à chaque cellule ont été connectées une ou plusieurs tuiles aléatoirement choisies parmi une banque préfabriquée et ordonnée selon le style d'architecture et son type (fenêtre, balcon, porte). Quelques règles procédurales ont été ajoutées afin d'éviter les cas non-souhaitables, tels qu'avoir une porte qui ne soit pas au rez-de-chaussée et sans balcon.

Malgré la simplicité de leur technique, les résultats obtenus sont déjà très intéressants, bien que limités, et démontrent le potentiel que la modélisation par tuiles peut atteindre.

Leur technique souffre de plusieurs problèmes et limitations. Il n'y a pas d'intérieurs et la forme des édifices est élaborée semi-automatiquement, ce qui nécessite un apport extérieur et limite les formes possibles. L'apparence des façades est limitée par son tuilage uniforme, aligné sur les axes et sans contraintes, ce qui diminue le positionnement des fenêtres, portes, balcons, etc.

2.2.4 Langages

Une autre approche à la modélisation procédurale est d'utiliser un langage dans lequel on y introduit des opérations spécifiques à un ou plusieurs types de modèles de base.

Cutler *et al.* [CDM+02] proposent une telle approche procédurale pour la création de solides par couches. Un champ de distance signé est utilisé comme modèle descriptif de base, lequel peut être combiné à un ensemble d'opérations de simulation et de sculpture. Dans un premier temps, le modèle intérieur ainsi que ses couches externes sont spécifiés à partir soient de maillages ou

de fonctions implicites qui sont aussitôt convertis en champ de distance signé. Par la suite, la formation de nouvelles couches externes et la combinaison de volumes peuvent être effectuées à l'aide d'opérations de base telles l'union, la différence ou le déplacement du champ de distance. Finalement, l'ensemble du modèle peut être modifié par l'application d'un ensemble d'opérations de simulation. Un système de simulation d'éléments finis y est, entre autres, implémenté. Le solide peut donc, sous la réaction de forces, subir des déformations et même des bris.

FIGURE 2.35 – Modèle d'un bonbon et de différentes variantes de couches décrites procéduralement. Images tirées de [CDM+02].

La figure 2.35 montre un exemple de modèle ainsi que diverses variations de couches pouvant être décrites par ce système. Un exemple plus complexe montrant le résultat d'une séance de simulation de sculpture est illustré à la figure 2.36.

Pour des fins d'affichage de simulation interactive, par exemple pour sculpter, le modèle est converti en un modèle tétraédral.

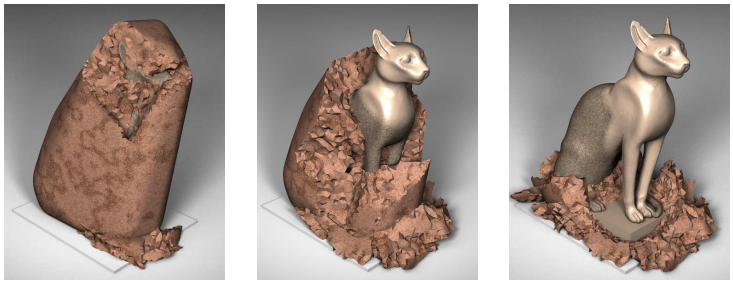


FIGURE 2.36 – Modèle volumique par couches subissant une opération de sculpture. Images tirées de [CDM+02].

Un autre exemple de langage de modélisation procédurale est GML (*generative modeling language*) [HF04, Hav05]. Inspiré par le langage PostScript, GML utilise comme modèle de base

un maillage polygonal avec surfaces de subdivision de type Catmull-Clark. Dans ce langage, une panoplie d'opérations sont disponibles allant de simples opérations mathématiques (sinus, cosinus, etc.) à de plus complexes telles le calcul d'intersection entre primitives. Mais le coeur du système repose sur les opérations d'Euler et d'extrusions. Ce langage est donc de très bas niveau puisque l'usager doit définir la topologie des objets créés. Un exemple de modèle procédural créé par ce langage est montré à la figure 2.37. Le langage GML peut servir à construire des objets de tous niveaux de complexité comme le démontrent de récents travaux [HHKF10] pour générer des édifices.

FIGURE 2.37 – Fenêtre procédurale créée en GML. Images tirées de [HF04].

2.2.5 Autres

Contrairement aux techniques de modélisation basées sur les grammaires ou les langages, qui sont très générales et extensibles, il existe beaucoup de méthodes procédurales ad hoc. Pour n'en nommer que quelques-unes, il y a la modélisation de coquillages (voir la figure 2.38) présentée par Fowler *et al.* [FMP92], la génération de façades [FBS05], ou encore la construction de modèles simples d'édifices à partir d'empreintes réelles [LD03]. Une autre technique [GPSL03] permet de produire une ville simple en créant des édifices à partir d'extrusions ou par simulation [LWWF03]. Chen *et al.* [CEW+08] (voir figure 2.39) utilisent des champs tensoriels passés par images pour contrôler la génération de réseaux routiers, alors que Galin *et al.* [GPMG10] créent des routes en se servant d'une minimisation de coûts basée sur un algorithme de chemin le plus court (figure 2.40).

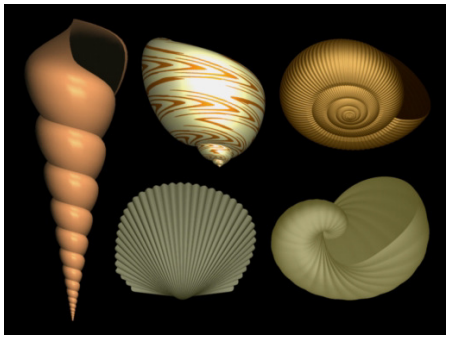


FIGURE 2.38 – Coquillages créés procéduralement. Image tirée de [FMP92].

FIGURE 2.39 – Modélisation procédurale et interactive de routes. Images tirées de [CEW+08].

FIGURE 2.40 – Modélisation procédurale de routes. Image tirée de [GPMG10].

Chapitre 3

Modélisation par blocs

La modélisation d'objets synthétiques doit satisfaire à des exigences différentes, en fonction des propriétés surfaciques des objets, des contraintes topologiques et du processus de modélisation lui-même. Au fil des années, comme il a été démontré au chapitre précédent, plusieurs représentations de surface et de volume ont été introduites, comprenant entre autres les maillages polygonaux, les maillages de subdivision, les surfaces polynomiales, les surfaces implicites, etc. Quant aux outils utilisés pour manipuler ces représentations, ils ont évolué pour répondre à un large spectre de besoins de modélisation, de la conception de haute précision (en CAD), au prototypage rapide.

Dans ce chapitre est introduit une nouvelle primitive de modélisation permettant de produire efficacement et intuitivement des objets possédant de bonnes propriétés de surface et de topologie. Nous sommes intéressés par une primitive de modélisation qui peut être utilisée principalement dans deux contextes :

- Modélisation interactive : Construction rapide et intuitive de l'approximation d'un objet, qui peut par la suite être sculpté et modifié aisément par un usager.
- Modélisation procédurale : Spécification topologique aisée, avec une définition volumique (pour supporter entre autres les opérations booléennes), incluant une paramétrisation de surface générale et finalement un bon contrôle surfacique.

Lors de la modélisation d'objets à l'aide de polygones, de maillages de subdivision [AS10], et de surfaces polynomiales [PT95], les artistes doivent être très prudents pour éviter les auto-intersections, les fissures, les sommets dupliqués, la définition incohérente du volume (intérieur et extérieur), la discontinuité des surfaces et de la paramétrisation, et encore plusieurs autres

problèmes potentiels. Ces problèmes sont exacerbés lorsque les objets résultants doivent avoir une surface et un volume cohérents. Bien entendu, cette liste de problèmes ne s'applique pas à toutes les représentations.

Les surfaces implicites [Blo97] et les *F-Rep* [PASS95] offrent une définition de surface continue et des propriétés volumiques valides. Malheureusement, leurs surfaces limites peuvent être complexes à extraire, et souvent la qualité du maillage obtenu n'est pas très élevée. Ce dernier point est particulièrement vrai dans le cas des surfaces contenant des arêtes vives et aussi des surfaces contenant de grandes régions minces comme dans le cas d'un mur. De plus, une bonne paramétrisation surfacique peut être difficile à obtenir en présence, entre autres, de changements topologiques.

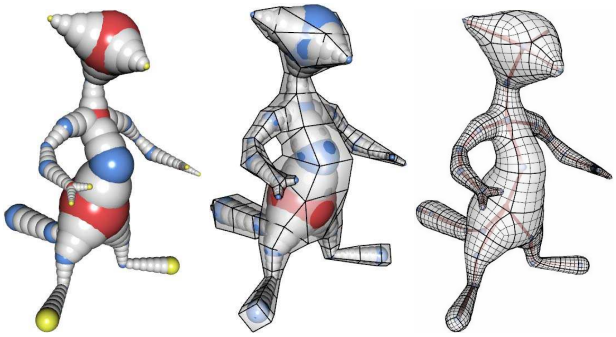


FIGURE 3.1 – Exemple de *B-Mesh* où le modèle est créé à partir de sphères connectées entre elles. Images tirées de [JLW10].

Les *ZSpheres*/*B-Mesh* [PIX11, JLW10], dont un exemple peut être vu à la figure 3.1, sont des primitives de modélisation flexibles basées sur les maillages de subdivision générés en enveloppant un arbre de sphères. Elles offrent un ensemble de bonnes propriétés surfaciques et volumiques ainsi qu'une paramétrisation permettant de sculpter des détails de surface grâce aux cartes de déplacement. Ces primitives sont bien adaptées pour la modélisation d'objets de type organique (personnage, animal, végétation, etc.), mais beaucoup moins pour les objets de type industriel possédant des arêtes vives (édifices, meubles, etc.).

Les "*polycube maps*" [THCM04, XGH+11] ont été utilisés comme moyen de représentation efficace d'objets de diverses topologies, mais non comme primitives de modélisation. Chaque face d'un *polycube* encode dans une carte le déplacement afin de générer le maillage polygonal final (voir figure 3.2).

Inspirés par ces deux dernières représentations et plus généralement par les surfaces implicites, nous avons combiné un certain nombre de leurs concepts clés pour dériver une primitive

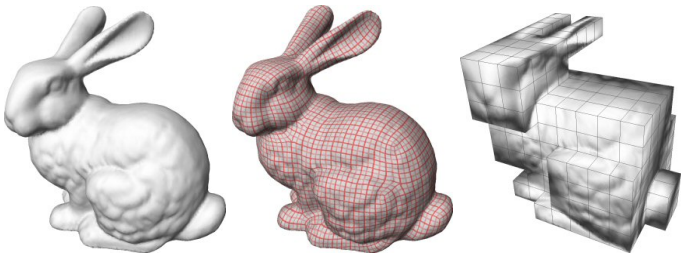


FIGURE 3.2 – *Polycube map*. De gauche à droite : la géométrie initiale, la paramétrisation de la surface et les polycubes avec chaque texture de déplacement. Images tirées de [THCM04].

à base de blocs. Chaque bloc peut être interprété comme un cube dans un *polycube map*, avec sa paramétrisation établie par face. Un groupe de blocs sont assemblés en reliant certaines de leurs faces, produisant un résultat similaire aux surfaces implicites et aux *ZSpheres*, mais avec un plus grand contrôle sur la résolution et la courbure locale de chaque connexion. Les blocs reliés résultants définissent la forme de base (ainsi que sa topologie) de l'objet modélisé, alors que la paramétrisation des faces externes sert à encoder les détails de surface qui sont juxtaposés sur la surface de subdivision augmentée d'arêtes lisses et vives.

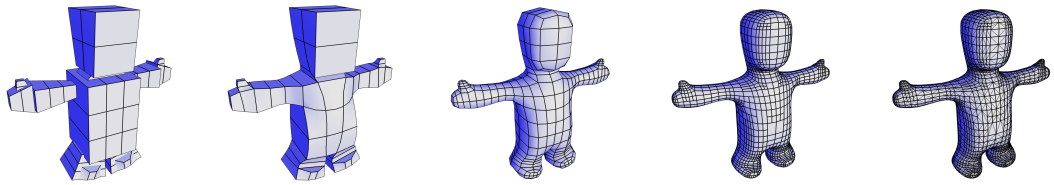
Cette combinaison de représentations conduit à une primitive de modélisation intuitive, facile à contrôler, et générale, qui peut générer une grande variété d'objets avec de bonnes propriétés surfaciques et volumiques, et de topologies différentes.

Pour le reste de ce chapitre, une description détaillée de notre système de modélisation par blocs suit et s'attarde sur la définition de la primitive, la création de connexions inter-blocs, la création du maillage de contrôle et finalement, la tessellation de la surface. Une extension de notre système est décrite pour utiliser des opérations booléennes, avec un emphase particulier sur l'efficacité et la robustesse afin d'être pratique. Ensuite, un langage simple est fourni pour son utilisation dans un contexte procédural, puis sont discutés ses limitations, quelques résultats, des comparaisons avec d'autres modèles, ainsi que des directions pour des améliorations futures.

3.1 Système de base

Le concept derrière la modélisation par blocs consiste premièrement à construire une forme approximative de l'objet voulu et possédant la bonne topologie, sans avoir à se soucier avec les détails de descriptions et d'opérations de topologie de bas niveau, telles les opérations d'Euler. Dans un second temps, cette forme approximative est raffinée avec une fonction de déplacement fournie procéduralement ou construite à la main (par exemple, par sculpture). Cette façon de procéder est à la fois bénéfique dans le cas de la modélisation interactive et de celle procédurale.

Du point de vue usager, un objet construit par blocs est défini par trois éléments. Tout d'abord, un ensemble de blocs (notre primitive de base) est utilisé pour décrire les parties principales de l'objet. Ensuite, des liens spécifient la connectivité inter-blocs. Ensemble, ils définissent un maillage de contrôle possédant la topologie voulue. Finalement, un déplacement de surface est optionnellement appliqué en se servant de la paramétrisation simple héritée des blocs pour générer le résultat final.



(a) Blocs (b) Connexions (c) Maillage de contrôle (d) Maillage final et tessellation

FIGURE 3.3 – Les quatre étapes de la chaîne de modélisation par blocs.

Du point de vue implémentation, le pipeline pour générer un objet est divisé en quatre étapes : la définition des blocs, la connexion entre les blocs, la création du maillage de contrôle et la génération du maillage. Une illustration de ce pipeline apparaît à la figure 3.3.

3.1.1 Blocs

Un *bloc* est une primitive volumique s'apparentant à un cuboïde défini par huit sommets et six faces. Il n'y a aucune restriction sur la position des sommets du bloc, si ce n'est qu'ils doivent générer un intérieur valide composé d'un ensemble 3D ouvert connexe. Ceci n'est, cependant, pas imposé ni vérifié dans notre système. Un exemple de configuration invalide est illustré à la figure 3.6.

Chaque face d'un bloc est défini comme une surface bilinéaire qui peut être subdivisée indépendamment des faces adjacentes, sous le format d'une grille régulière de sous-faces de résolution quelconque. Les faces et les sous-faces produites sont strictement des quadrilatères, cependant, sans être contraintes à la planarité.

Les sous-faces sont les éléments définissant la géométrie. Elles peuvent être connectées ensemble (voir prochaine section 3.1.2) pour former la surface finale en utilisant une subdivision de type Catmull-Clark [DKT98], héritant ainsi de ses propriétés : une continuité C^2 partout, sauf C^1 pour les régions entourant les sommets de valence $\neq 4$.

Pour contrôler la continuité, chaque arête d'un bloc peut être étiquetée comme vive. Les

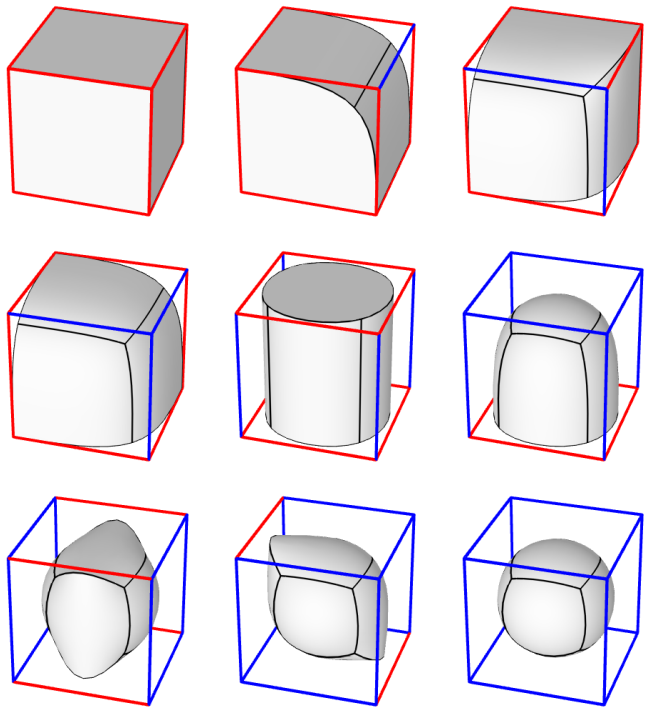


FIGURE 3.4 – Différentes configurations d'arêtes lisses et vives pour un bloc. Les arêtes rouges d'un bloc produisent des arêtes vives sur le maillage, alors que les arêtes bleues produisent une surface lisse.

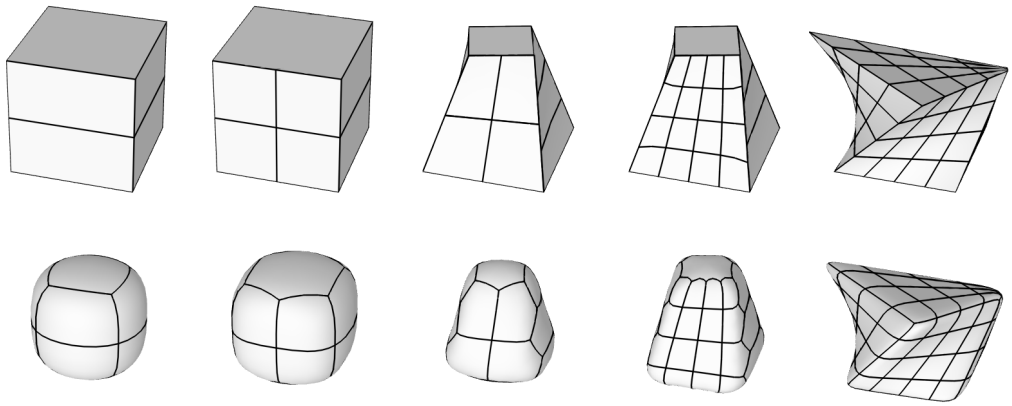


FIGURE 3.5 – Rangée du haut : augmentation du nombre de sous-faces et déplacement des sommets avec des arêtes vives. Rangée du bas : mêmes configurations, mais avec des arêtes lisses.

figures 3.4 et 3.5 montrent différentes définitions de blocs possédant des variations de types d'arêtes avec la surface résultante. La figure 3.7 montre la relation entre les éléments composant une géométrie par blocs, soit : les faces, les sous-faces, les *patchs* et les *sous-patchs*.

Maintenant que la structure d'un simple bloc a été définie, la prochaine section s'attardera à expliquer comment former des groupes de blocs à l'aide de connexions.

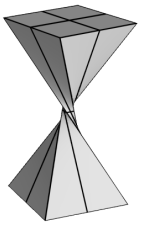
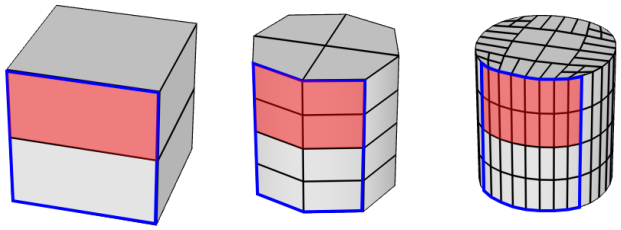


FIGURE 3.6 – Exemple de configuration invalide de sommets pour un bloc.



(a) Faces et sous-faces (b) *Patchs* (c) *Sous-patchs*

FIGURE 3.7 – Les quatre éléments de la primitive bloc. Dans l'exemple spécifique de cette figure : (a) Une face est subdivisée en deux sous-faces. (b) Une sous-face est subdivisée en quatre *patchs*. (c) Une *patch* est subdivisée en quatre *sous-patchs*. Le contour bleu indique la région associée à une des faces originales du bloc, et le remplissage rouge indique la région associée à une des sous-faces.

3.1.2 Connexions

Une fois que les blocs sont définis (subdivision de faces et vivacité des arêtes) et positionnés pour établir la forme approximative de l'objet désiré, les connexions peuvent être calculées entre leurs sous-faces. Ces connexions, paires de faces ou de sous-faces appartenant à des blocs voisins, vont créer la topologie de l'objet automatiquement. Ce processus exige seulement un petit nombre d'attributs : un identificateur de groupe pour chaque bloc, une liste globale de paires d'identificateurs de groupe, et un seuil de distance exprimé sous forme scalaire. Les paires d'identificateurs de groupe définissent quelles parties d'un modèle peuvent se connecter ensemble (voir la figure 3.8). La valeur du seuil détermine la distance maximale de connexion permise, au-delà de laquelle les blocs ne se fusionnent pas. Ce seuil est spécifié pour être invariant au changement d'échelle. Il est ainsi calculé en considérant les périmètres de chaque sous-face :

$$\text{distance} < \text{seuil} \times (\text{périmètre}A + \text{périmètre}B).$$

La formation automatique des connexions s'effectue de la façon suivante. Pour chaque

sous-face[1] de chaque bloc, un rayon est lancé du centre de la sous-face vers l'extérieur du bloc en direction de sa normale (voir figure 3.9). Une connexion entre une sous-face A (lanceur) et une sous-face B (la plus proche atteinte) existe et est établie si et seulement si :

1. Les sous-faces A et B appartiennent à des blocs différents.
2. L'identificateur de groupe de A permet la connexion à l'identificateur de groupe de B (de la liste globale de paires).
3. La distance entre A et B est comprise dans les limites du seuil relatif spécifié.
4. La sous-face B est la plus proche atteinte de A, et vice-versa.
5. Aucune arête dégénérée ne doit être créée (voir figure 3.10).

Ici, le lancer de rayons est utilisé pour détecter les connexions car il est rapide, facile d'implémentation et non ambigu : les connexions peuvent seulement être établies si le rayon partant du centre d'une sous-face atteint une autre sous-face. Si cela est jugé trop restrictif, le lancer peut être remplacé par n'importe quelle technique de calcul de couverture, comme par exemple le lancer de rayons multiples, ou le tracer de volume (*shaft*).

Pour prévenir la formation de topologies inconsistantes (condition 5 précédemment), une seule connexion entre deux blocs peut être établie. Ainsi, seule la connexion possédant la plus courte distance sera conservée. Une liste de connexions forcées, indépendantes de leurs distances actuelles, peut aussi être spécifiée manuellement pour une flexibilité accrue, en spécifiant les deux sous-faces à joindre pour chacune des connexions.

(a) (b) (c)

FIGURE 3.8 – Trois variations d'une main où les teintes représentent trois identificateurs de groupe : (a) aucune connexion, même pas dans le même groupe ; (b) tous les groupes se connectent ensemble ; et (c) les identificateurs de groupe des doigts définissent une connexion à la paume, mais pas entre eux. Pour se connecter à la paume, la face du dessus de la paume est subdivisée en 4×1 sous-faces et en 3×1 pour le côté gauche (le pouce se connecte à la sous-face du milieu).

[1] ou la face entière, si elle n'est pas subdivisée.

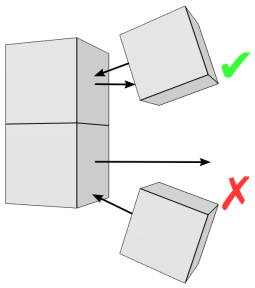


FIGURE 3.9 – Connexion de sous-faces. Les sous-faces supérieures se connectent ensemble puisqu'elles sont mutuellement leur sous-face la plus proche, contrairement à la paire inférieure.

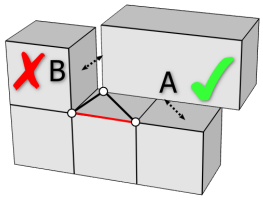


FIGURE 3.10 – Exemple d'une connexion invalide. Si la connexion B est exécutée après que A soit connectée, les trois sommets (blanc) vont fusionner et former une arête dégénérée (rouge).

Après que toutes les connexions soient établies, les positions des sommets de toutes les sous-faces connectées sont modifiées pour être jointes ensemble. Pour ce faire, nous déterminons, premièrement, une relation entre les paires de sommets provenant de chaque sous-face. Puisque nous travaillons avec des quadrilatères, il y a quatre façons possibles de connecter les deux sous-faces. Pour trouver la meilleure configuration, on commence par choisir une paire arbitraire de sommets, un sommet pour chaque sous-face, et ensuite on crée une correspondance entre le reste des sommets en sens anti-horaire pour la sous-face A et horaire pour la sous-face B. La figure 3.11 illustre ce processus. Le coût associé à cette configuration est calculé comme la somme totale des distances entre chaque paire de sommets. Les trois autres configurations sont testées en changeant un des sommets de la paire de départ par chacun (tour à tour) des autres sommets de la même sous-face. Après avoir évalué toutes ces configurations de connexion, celle possédant le coût minimum est choisie et exécutée.

La nouvelle position des sommets joints est calculée comme étant leurs moyennes. Rien n'empêche de calculer une sommation pondérée des sommets au lieu de la moyenne. Il n'est cependant pas très clair de quelle façon les pondérations peuvent être automatiquement ou intuitivement attribuées, spécialement dans le cas de connexions multiples tel qu'illustré à la figure 3.12. Il est important de noter que plus de deux sommets peuvent être joints entre eux, notamment pour les sommets situés sur les bords d'un bloc ayant de multiples voisins (voir

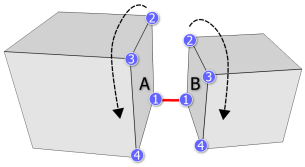


FIGURE 3.11 – Connexion entre deux sous-faces. Les sommets numérotés 1 forment la paire de départ de la connexion. Pour les trois autres configurations à tester, la paire de départ est définie par les sommets des sous-faces A-B numérotés suivant : 2-1, 3-1, et 4-1.

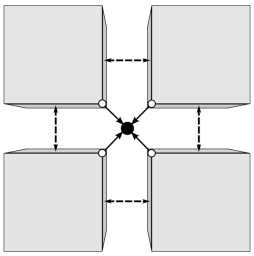


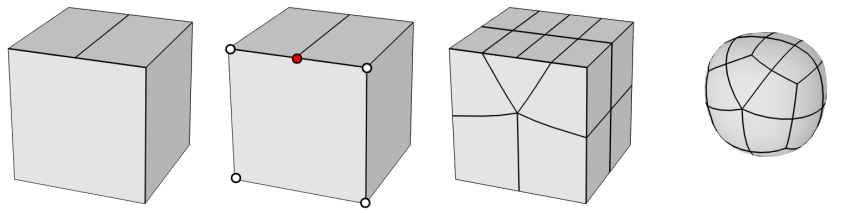
FIGURE 3.12 – Calcul de la position d'un sommet connecté de valence 4. Les segments en pointillés représentent les connexions entre blocs, et le point noir central est la nouvelle position calculée comme étant la moyenne des quatre sommets.

figure 3.12). Dans ce cas, le sommet résultant peut présenter une valence quelconque plus grande que 2, dépendamment du nombre de blocs connectés.

3.1.3 Maillage de contrôle

L'étape suivante crée un maillage de contrôle pour générer une surface de subdivision. Ce maillage de contrôle est créé en assemblant toutes les sous-faces externes (i.e. des sous-faces non connectées) pour former un maillage étanche. Les sous-faces internes sont simplement ignorées et ne participent pas à la géométrie finale. Une arête du maillage de contrôle est étiquetée comme vive (i.e. formant un pli) lorsqu'au moins une des arêtes originales provenant des blocs formant cette arête est vive. La priorité est donnée aux arêtes vives à l'opposé des arêtes lisses, puisque la vivacité est considérée comme une propriété ajoutée dans notre système. Après l'assemblage des sous-faces externes, nous nous retrouvons avec un maillage de quadrilatères, contenant possiblement des sommets en T en raison des différentes résolutions des grilles régulières de sous-faces (voir figure 3.13a). Ce problème est corrigé en deux étapes. Tout d'abord, des sommets sont ajoutés aux quadrilatères contenant des sommets en T, les transformant ainsi en n-gones (figure 3.13b). Ensuite, une passe de subdivision de Catmull-Clark [DKT98] est appliquée pour

obtenir un maillage de contrôle composé exclusivement de quadrilatères (voir figures 3.13c et 3.13d).



(a) Sous-faces avec sommet en T (b) Sous-faces (c) *Patchs* (d) *Patchs* lisses

FIGURE 3.13 – (a) La sous-face (face divisée en 1×1) de devant contient un sommet en T puisque la face du dessus contient deux sous-faces. (b) Élimination du sommet en T par l'insertion d'un nouveau sommet (en rouge). La sous-face de devant, initialement un quadrilatère, devient un pentagone. (c) Des *patchs* quadrilatères sont formées après l'exécution d'une passe de subdivision de Catmull-Clark. Les arêtes ici sont vives. (d) Géométrie finale lorsque toutes les arêtes initiales sont lisses.

3.1.4 Génération du maillage

La dernière étape du processus utilise les surfaces de subdivision pour générer le maillage final. Le maillage de contrôle est subdivisé adaptativement en regard de la courbure locale et du déplacement appliqué au modèle géométrique. Les opérations booléennes sont aussi supportées en insérant des arêtes et des sommets aux points d'interpénétration et en déterminant les régions intérieures et extérieures par l'évaluation de l'arbre de CSG. Ce dernier point sera expliqué en détail à la section 3.2. Puis, une dernière opération de tessellation est appliquée pour produire un maillage triangulaire fermé (étanche).

Approximation par *patch*

Pour obtenir un maillage de haute qualité avec moins de subdivisions, on approxime la surface de subdivision à l'aide de *patchs* paramétriques. Ceci donne deux avantages par rapport aux surfaces de Catmull-Clark conventionnelles : pour n'importe quel niveau de subdivision, chaque sommet est positionné exactement sur la surface limite au lieu de converger vers celle-ci, et le motif de subdivision est découplé de son évaluation. Ce dernier point donne beaucoup de flexibilité sur la tessellation et simplifie la mise en place d'un maillage adaptatif.

Au cours des dernières années, quelques techniques pour approximer les surfaces de subdivision à l'aide de *patchs* paramétriques ont été élaborées. Notre choix s'est arrêté sur celle

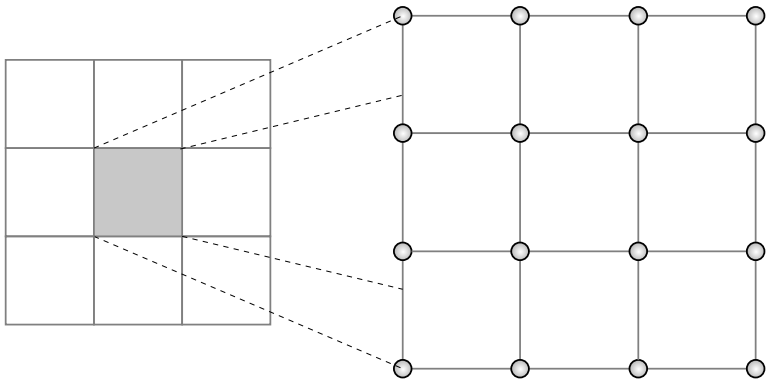


FIGURE 3.14 – Un quadrilatère (en gris) du maillage de contrôle ne contenant aucun sommet extraordinaire (valence $\neq 4$) est transformé en *patch* bicubique (une *patch* de Bézier) de degré 3×3 ayant 4×4 points de contrôle.

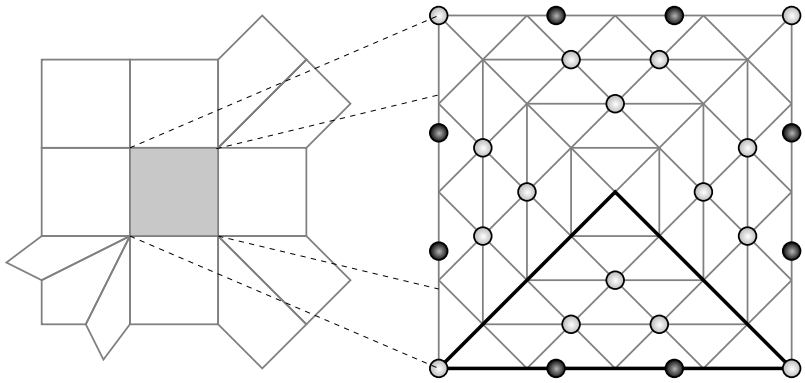


FIGURE 3.15 – Un quadrilatère (en gris) du maillage de contrôle contenant au moins un sommet extraordinaire est transformé en une *patch* composite (c-*patch*). Une des quatre *patchs* triangulaires de degré 4 formant la c-*patch* est illustrée par un contour noir. La c-*patch* est formée de 24 points de contrôle. Les points de contrôle en gris correspondent directement à des points de contrôle des *patchs* triangulaires, alors que les points de contrôle noirs positionnés de façon à assurer la continuité (C^1) entre les *patchs* bicubiques voisines servent à interpoler certains points de contrôle des *patchs* triangulaires.

```
1  void ParametricPatch::parameters( float u, float v, Vec3& pos, Vec3& normale )
2  {
3     if( régulière )
4     {
5        evalCubic( u, v, pos, normale );
6     }
7     else
8     {
9        // Trouver et évaluer la bonne patch triangulaire.
10       if( v ≤ u )
11       {
12          if( u+v ≤ 1.0f )
13             evalTriangle0( 1.0f-u-v, u-v, 2.0f*v, pos, normale );
14          else
15             evalTriangle1( u-v, u+v-1.0f, 2.0f*(1.0f-u), pos, normale );
16       }
17       else
18       {
19          if( u+v ≤ 1.0f )
20             evalTriangle3( v-u, 1.0f-v-u, 2.0f*u, pos, normale );
21          else
22             evalTriangle2( u+v-1.0f, v-u, 2.0f*(1.0f-v), pos, normale );
23       }
24    }
25 }
```

LISTAGE 3.1 – Pseudocode pour l'évaluation d'une *patch* (position et normale). Dans un premier temps, la sélection du type de *patch* bicubique ou triangulaire est effectuée. Dans le cas d'une c-*patch*, la bonne *patch* triangulaire est choisie et les coordonnées paramétriques sont converties en coordonnées barycentriques de la *patch*.

développée par Ni *et al.* [NYM+08]. Dans cette approximation, les quadrilatères ordinaires du maillage de contrôle, soient les quadrilatères dont les quatre sommets ont une valence de 4, sont convertis en *patchs* bicubiques (illustré à la figure 3.14) qui entre elles possèdent une continuité C^2. Tous les autres quadrilatères du maillage de contrôle, soient ceux possédant au moins un sommet extraordinaire (valence $\neq 4$), sont convertis en quatre *patchs* triangulaires possédant une continuité C^1 et sont regroupés en une seule c-*patch* définie par 24 points de contrôle. Les points de contrôle externes sur la bordure de la c-*patch* correspondent aux points de contrôle d'une *patch* bicubique, alors que les points de contrôle internes correspondent à des points de contrôle des *patchs* triangulaires. Un exemple est illustré à la figure 3.15. Lors de l'évaluation d'une c-*patch*, les coordonnées paramétriques sont transformées en coordonnées barycentriques de la *patch* triangulaire contenant les coordonnées initiales. Un exemple de code pour effectuer cette transformation est donné au listage 3.1. Pour le calcul des points de contrôle, tous les détails sont fournis dans l'article de Ni *et al.* [NYM+08].

Parmi les autres techniques d'approximation, on retrouve celle de Loop et Schaefer [LS08] où chaque quadrilatère initial du maillage de contrôle est converti en trois *patchs*. Une première approximant la géométrie, et deux autres définissant les tangentes de la surface, le tout pour un total de 25 points de contrôle. Une limitation importante de cette technique, réduit son utilisation :

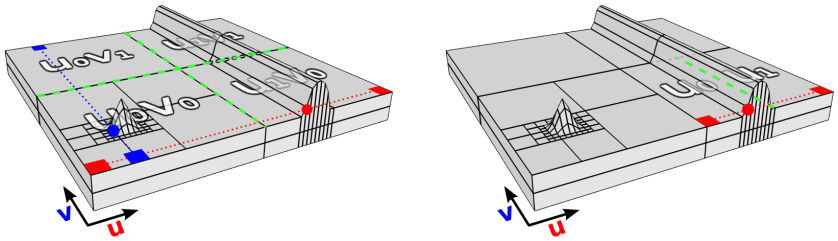
elle ne supporte pas les arêtes vives. Cette restriction a été enlevée par les travaux de Kovacs *et al.* [KMDZ09] avec cependant quelques artefacts dans des cas limites. Finalement, les travaux de Loop *et al.* [LSNC09] utilisant des *patchs* de Gregory pour l'approximation, semblent être une bonne solution alternative[2]. Chaque *patch* rectangulaire exige 20 points de contrôle pour être définie et les *patchs* triangulaires sont supportées contrairement à toutes les autres techniques mentionnées précédemment.

Subdivision de *patch*

Après avoir converti en *patch* paramétrique chaque quadrilatère du maillage de contrôle, on subdivise chaque *patch* en un sous-ensemble de *sous-patchs* en suivant une décomposition de type *kd-tree* [LC05]. Un exemple de cette subdivision peut être aperçu aux figures 3.3d et 3.20. Chaque sommet de la *sous-patch* est positionné selon l'évaluation de sa *patch* paramétrique, et ajusté par un déplacement enregistré sous la forme d'une fonction ou d'une texture. La paramétrisation utilisée pour évaluer le déplacement est similaire à celle décrite par Burley et Lacewell [BL08], où chaque *patch* a sa propre paramétrisation reliée à sa taille en espace monde ou selon son niveau de détail.

L'algorithme et la métrique utilisés pour évaluer si la subdivision d'une *sous-patch* doit être effectuée sont décrits comme suit. En premier lieu, si le déplacement est exprimé sous la forme d'une fonction et non d'une texture, on construit une image géométrique [GGH02] de la résolution recherchée. Ceci a pour but de se retrouver avec seulement des déplacements sous la forme de texture, élément nécessaire pour la suite de l'algorithme. Il faut noter cependant que cette étape n'est seulement utile que lorsqu'il y a une fonction de déplacement. Cette conversion sous forme de texture exige une quantité de mémoire limitée puisque seulement la *patch* courante peut être transformée et gardée à la fois. Il y a une exception cependant. Lorsqu'une *patch* contient une arête vive, il y a discontinuité de normales de chaque côté de celle-ci. Puisque généralement la fonction de déplacement est relative à la normale, il pourrait y avoir formation d'une craque le long de l'arête. Pour corriger ce problème, il faut évaluer la position du sommet voisin (de la *patch* voisine) le long de l'arête ou des sommets multiples dans le cas des coins de la *patch*. La position corrigée et partagée par les *patchs* voisines peut être calculée de plusieurs façons. Une première technique consiste tout simplement à moyenner les positions. Une autre façon plus coûteuse mais donnant de meilleurs résultats dans les cas

[2]Cette technique n'a pas été employée puisqu'elle est apparue après notre implémentation de cette section de l'algorithme.



(a) (b)

FIGURE 3.16 – Une *sous-patch* est subdivisée en u lorsqu'un point 3D provenant d'un texel d'une rangé de la texture est plus éloigné, d'un certain seuil, d'un segment formé en joignant le premier texel et le dernier texel de la rangée. L'évaluation est similaire en v. À gauche : subdivision d'une *sous-patch* en uv. À droite : subdivision d'une *patch* en u seulement.

extrêmes (grandes différences entre les normales et un déplacement élevé) calcule le point le plus proche sur l'intersection des plans sous-tendants ou utilise la solution d'une quadrique formée par tous les plans sous-tendants dans le cas des coins. Une dernière solution, proposée par Castaño *et al.* [Cas08, LSNC09] consiste à donner la propriété unique des arêtes et des sommets à une seule *patch*. Ainsi, lors de l'évaluation des positions le long d'une arête, la position choisie sera celle provenant de la *patch* propriétaire de l'arête. Cette technique donne de moins bons résultats que les deux autres précédentes, mais est en contrepartie la plus efficace, puisqu'elle n'exige qu'une seule évaluation de position.

Après avoir converti en texture de positions la fonction de déplacement, on peut finalement procéder à la subdivision de la *patch* en *sous-patchs*. On commence par la création d'une *sous-patch* identique à la *patch* de départ, puis récursivement on évalue la planarité et on procède à la subdivision si nécessaire. Une *sous-patch* est soit terminale, soit subdivisée en u, en v, ou dans les deux sens uv. Dans chacun de ces cas, la *sous-patch* sera subdivisée en son milieu. L'évaluation de la texture de déplacement s'effectue en deux passes, une en direction u et la seconde en v. Lors de l'évaluation en u, chaque texel (un point 3D déplacé) est comparé, rangée par rangée, à un segment 3D défini en reliant le premier texel et le dernier texel de la rangée courante. Aussitôt qu'un texel est plus éloigné qu'un seuil spécifié, la *sous-patch* est marquée pour subdivision en u. L'évaluation en v est effectuée de façon similaire, mais en testant les texels colonne par colonne. Le processus est illustré à la figure 3.16. Dans la figure de gauche, un cercle rouge indique qu'une subdivision est demandée en u lors de la traversée et de la vérification des rangées de texels, et un cercle bleu l'indique en v. La *sous-patch* est par conséquent subdivisée en quatre *sous-patchs* étiquetées en gris $u_i v_j$. Dans la figure de droite, pour une *sous-patch*, un

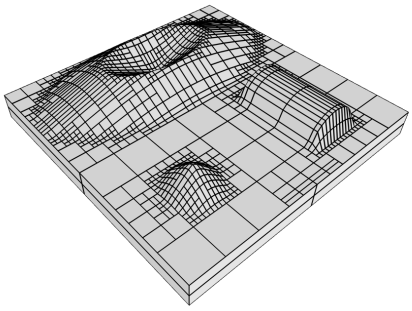
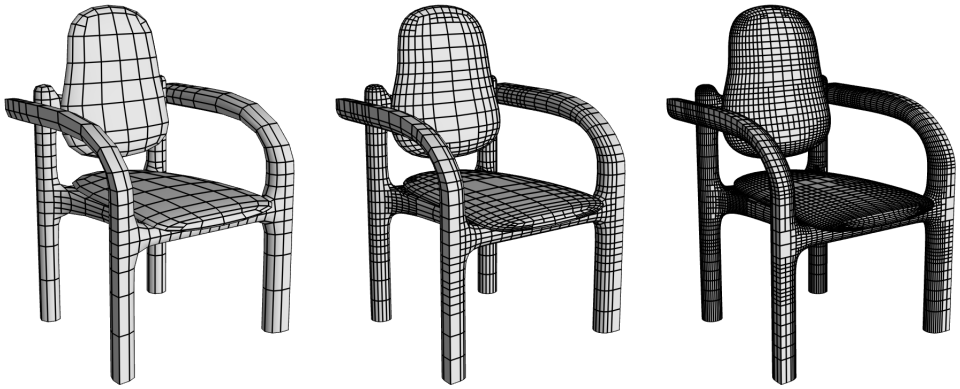


FIGURE 3.17 – Subdivision hiérarchique et anisotropique d'une *patch* en *sous-patchs*.



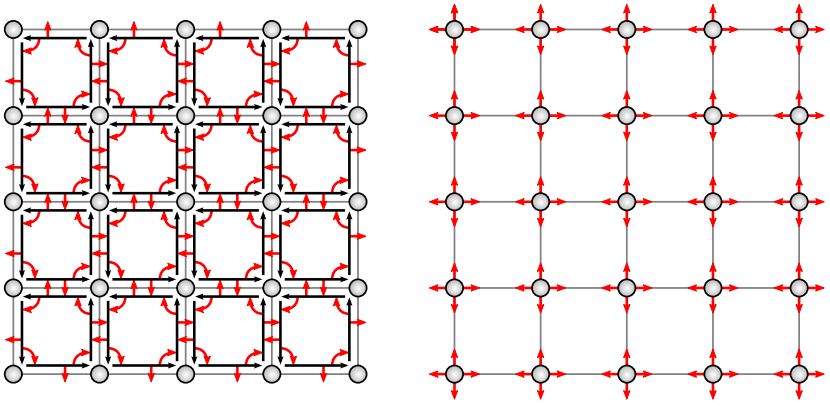
(a) (b) (c)

FIGURE 3.18 – Trois niveaux de subdivision adaptative d'un même modèle de chaise. Le niveau le plus bas (a) contient 1336 *sous-patchs*, alors que le niveau intermédiaire (b) en contient 4282 et 14989 pour le niveau le plus élevé (c).

cercle rouge indique qu'une subdivision est demandée en u, mais aucune telle subdivision n'est demandée en v. Cette *sous-patch* est donc subdivisée exclusivement en u en deux *sous-patchs*. Cette métrique fournit une bonne qualité de subdivision anisotropique comme on peut l'observer à la figure 3.17. Elle permet aussi d'ajuster le niveau de détail selon la qualité recherchée, comme en fait foi la figure 3.18.

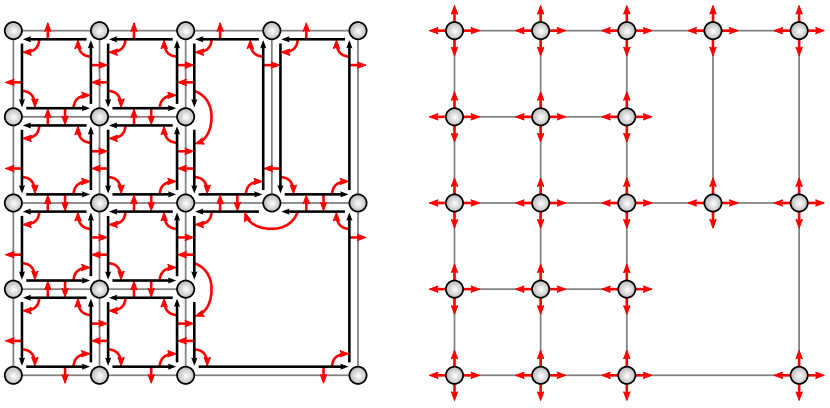
Structure de données

Différentes structures de données peuvent être utilisées pour exprimer la surface formée par l'ensemble des *sous-patchs*. La plus connue et utilisée est sans doute la structure par demi-arêtes encodant chaque face par un ensemble de demi-arêtes, chacune pointant sur la demi-arête suivante, voisine, sur le sommet de départ et optionnellement sur la demi-arête précédente. Cette structure est souple, mais peut être assez coûteuse en espace mémoire et en performance. Puisque la structure interne d'une *patch* est restreinte à un ensemble de quadrilatères alignés, nous avons



(a) Maillage de demi-arêtes (b) Maillage en croix

FIGURE 3.19 – (a) Chaque demi-arête (en noir) possède trois pointeurs. Un pointant sur la demi-arête suivante, un sur la demi-arête adjacente (les deux en rouge), et un troisième (non visible pour fin de clareté) sur le sommet associé. Au total, pour représenter l'information de connectivité de cette *patch* subdivisée en 16 *sous-patchs*, 192 pointeurs sont nécessaires. (b) Dans un maillage en croix, chaque sommet possède quatre pointeurs sur les sommets avoisinants. Au total, 100 pointeurs sont requis pour décrire la connectivité de la *patch*.



(a) Maillage de demi-arêtes (b) Maillage en croix

FIGURE 3.20 – (a) Dans cette *patch* subdivisée adaptativement en 11 *sous-patchs*, 47 demi-arêtes définissent la connectivité, résultant en 141 pointeurs. (b) La *patch* contient 20 sommets et 80 pointeurs décrivant la connectivité dans la structure de maillage en croix.

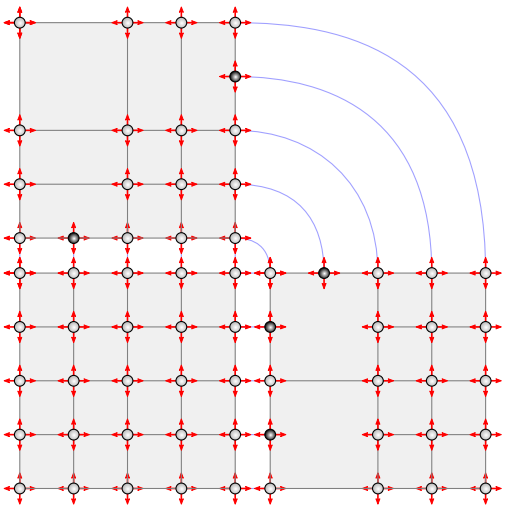
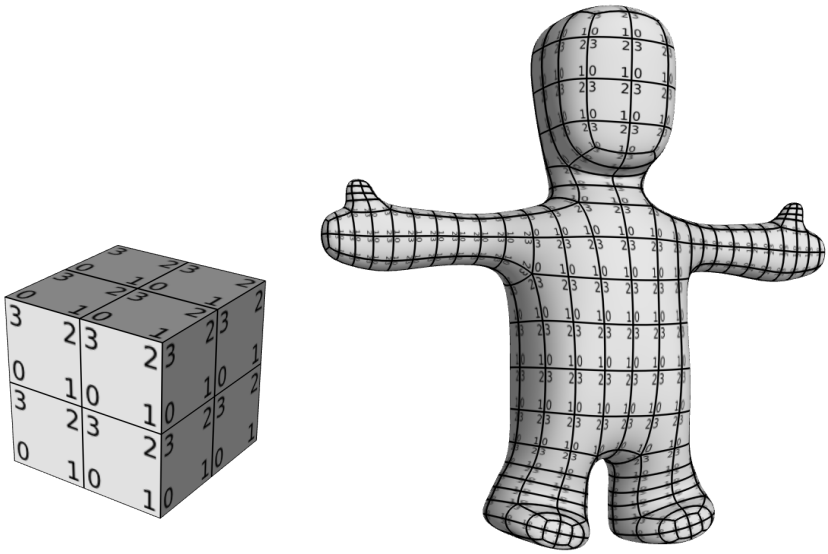


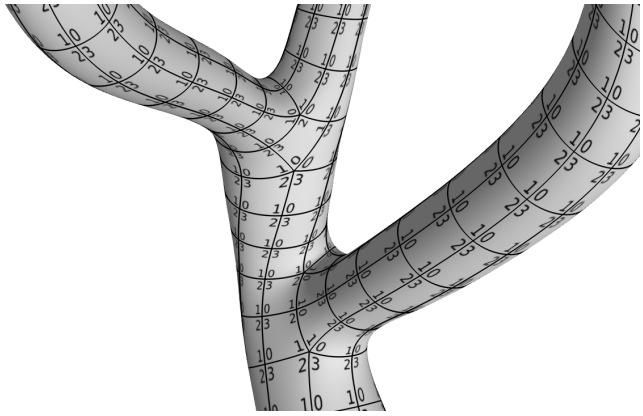
FIGURE 3.21 – Structure de maillage en croix pour définir trois *patchs*. Les sommets noirs sont ajoutés en bordure des *patchs* pour maintenir la connectivité inter-*patchs*. Les segments bleus représentent les pointeurs liant des sommets inter-*patchs*. Ces sommets ont des positions identiques.

développé une structure moins coûteuse et adaptée à la topologie spécifique obtenue lors de la subdivision des *sous-patchs* (voir figure 3.19). Chaque *patch* est définie par un ensemble de sommets reliés entre eux par quatre pointeurs contenus pour chacun des sommets. Les pointeurs d'un sommet relient les sommets voisins possédant les uv en espace paramétrique suivant : $(u, v - v0)$, $(u + u0, v)$, $(u, v + v1)$ et $(u - u1, v)$ où (u, v) est la coordonnée paramétrique du sommet. Les valeurs $u0$, $u1$, $v0$ et $v1$ sont définies comme les valeurs les plus petites possibles. Si pour une direction, aucun sommet ne correspond à la définition, le pointeur associé sera nul comme dans l'illustration de la figure 3.20. Comme chaque *patch* possède sa structure en croix, les sommets aux frontières entre deux *patchs* sont dupliqués et ont leurs propres uv et normales. La figure 3.21 montre la structure en croix associée pour la définition de *patchs* multiples.

Comparativement à la structure par demi-arêtes, notre structure utilise environ la moitié de l'espace mémoire pour définir la topologie du maillage. Au niveau de la performance, lors de changements de la structure, comme dans le cas de la subdivision des *patchs*, la structure en croix est deux fois plus performante. Par exemple, lors d'une opération de base importante, soit la subdivision d'une arête, avec la structure par demi-arêtes les opérations suivantes sont effectuées : la création d'un sommet et de deux demi-arêtes, et 10 pointeurs à assigner. Avec la structure par croix seulement un sommet est créé et 4 pointeurs doivent être assignés.



(a) Simple bloc (b) Personnage



(c) Tronc

FIGURE 3.22 – Affichage de la paramétrisation de surface pour différents maillages créés par le système de blocs. Chaque *patch* possède sa propre paramétrisation qui est illustrée ici par les chiffres 0 à 3 indiquant les quatre coins (0,0), (1,0), (1,1) et (0,1).

Paramétrisation

Pour permettre la définition d'une fonction de déplacement ainsi que l'application de textures associées aux propriétés de matériaux telles que la réflectance, la spécularité et la réfraction, une paramétrisation de la surface est requise. Toutes les techniques conventionnelles comme la création d'un atlas à l'aide de la triangulation [CMR$^+$99, CH02, LPRM02] ou encore les techniques d'aplanissement de surface [PB00, SH02] peuvent être utilisées pour créer cette paramétrisation. Cependant la structure par blocs permet de définir aisément une paramétrisation de base comportant plusieurs avantages. Cette paramétrisation est tout simplement définie en

octroyant à chaque *patch* sa propre plage de coordonnées rectangulaire comprise entre $[0, 1]$, comme on peut le voir à la figure 3.22a. L'orientation de la paramétrisation est liée à la définition des blocs, alors que la taille en texels dépend de deux facteurs, soient la taille de la *patch* en espace objet ainsi que le niveau de détail de cette région.

Lorsque notre système de modélisation est utilisé dans un contexte procédural, une information supplémentaire identifiant le bloc ainsi que la face (l'une des six) et la sous-face du bloc est fournie pour rendre unique tout point de la surface. Dans le cas où l'information de surface n'est nécessaire que du côté logiciel utilisée par un CPU, les texels peuvent être gardés dans une liste de textures multiples (une par *patch*) exactement comme décrit par Burley et Lacewell [BL08]. Toutefois, pour l'utilisation par un GPU, toutes les textures sont regroupées en une seule pour former un atlas. Pour ce faire, un algorithme de compactage de rectangles [NSS$^+$00, LC06b, CH07] doit être exécuté.

Parmi les avantages de cette paramétrisation, outre sa simplicité, on retrouve sa définition unique de la surface (comme les techniques mentionnées précédemment) contrairement à des paramétrisations par projection (planaire, cubique, cylindrique, etc.). On dénote aussi que la paramétrisation suit bien les lignes de l'objet (comme illustré à la figure 3.22) puisqu'elle est directement liée à la position et la forme des blocs approximant l'apparence de l'objet. Elle supporte aussi aisément les modifications apportées à l'objet lors de son édition, pour cause de sa localité et de son attachement aux blocs. Par exemple, on peut changer la topologie de l'objet en déplaçant des blocs et la paramétrisation suivra. Finalement, cette paramétrisation est facile d'utilisation lors de la construction d'objets procéduraux, contrairement aux techniques conventionnelles par atlas, encore une fois puisqu'elle est rattachée aux blocs et non à la triangulation.

Toutefois, cette paramétrisation n'est pas sans inconvénients. Moins une *patch* est rectangulaire (i.e. possède des angles de $90°$), plus la distorsion de la paramétrisation est élevée. De plus, chaque *patch* cause une discontinuité de la paramétrisation et exige par conséquent un dédoublement de tous les sommets de son contour. Ce dernier fait cause aussi des complications au niveau du filtrage GPU puisque les techniques de base matérielles, comme le *mip-mapping* et le filtrage anisotropique par échantillonnage multiple le long d'une ligne, ne peuvent s'appliquer sans artefacts (fuite de couleurs). Ces problèmes peuvent être solutionnés par la création d'un programme (*shader*) spécifique, mais au détriment de la performance. Ces deux derniers inconvénients peuvent être grandement diminués en regroupant dans l'atlas les *patchs* voisines.

Dès lors, les sommets des *patchs* voisines, autant sur le maillage que dans l'atlas, peuvent utiliser les mêmes coordonnées uv et ainsi éviter le dédoublement et la fuite des couleurs lors du filtrage. Pour ce qui est des contours où les discontinuités se trouvent, une copie de plusieurs texels de largeur des valeurs voisines peut suffire pour permettre le filtrage matériel sans artefacts. Cette technique est utilisée, entre autres, dans certains travaux [PB00, LPRM02]. Cette solution peut aussi être utilisée sans avoir recours à l'agrégation des *patchs*, mais au détriment de l'espace mémoire puisqu'il y aura un plus grand nombre de discontinuités.

Tessellation

La dernière étape pour l'obtention d'un maillage triangulaire est la tessellation des *sous-patchs*. Cette tessellation est effectuée en itérant sur toutes les *sous-patchs* de façon indépendante et ce, sans créer de craques malgré les niveaux disparates de subdivision entre voisins. Ceci est possible puisque lors de la subdivision adaptative des *patchs*, des sommets sont insérés (les sommets noirs de la figure 3.21) dans toutes les *sous-patchs* partageant une arête de la *patch*. Cette insertion de sommets en bordure des *patchs* a pour conséquence d'éviter la formation de sommets en T.

La tessellation d'une *sous-patch* s'effectue en deux temps. Dans un premier temps, tous les sommets du contour sont rassemblés pour former une polyligne en forme de boucle. Lors de cette étape, chaque sommet unique est inséré dans une table et y est assigné un index. La clef de recherche d'un sommet est formée des valeurs le constituant, soient sa position, sa normale et ses coordonnées de paramétrisation. De cette façon, si deux sommets voisins partagent les mêmes valeurs, un seul index sera créé dans la table, réduisant ainsi la taille du maillage final. Dans un second temps, la tessellation de la polyligne est effectuée. Un algorithme de triangulation simple tel que celui de Cignoni *et al.* [CMS96] peut être employé dans la majorité des cas. Cet algorithme consiste à créer une bande de triangles en reliant les sommets opposés en alternance, puis à créer à la toute fin, si nécessaire, un éventail de triangles pour les sommets restants ne pouvant pas faire partie de la bande. Dans les rares cas de polyligne concave complexe, nous utilisons un algorithme dit de "découpage d'oreilles" (*ear-clipping*) [Hel01]. Ce que nous entendons par complexe ici, c'est que bien que l'algorithme présenté plus haut soit défini pour une polyligne convexe, il peut fonctionner dans bien des cas concaves "simples", c'est-à-dire n'ayant pas trop de régions concaves trop prononcées. Pour détecter les cas complexes, on exécute toujours l'algorithme de base dans un premier temps en vérifiant l'orientation des triangles générés.

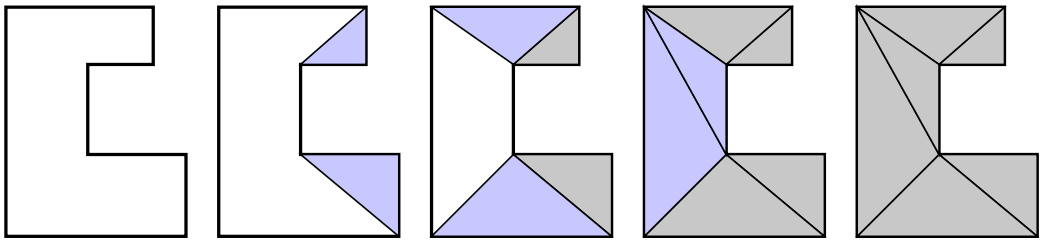


FIGURE 3.23 – Découpage par oreilles d'un polygone en blanc. Les triangles bleus sont les oreilles du polygone, alors que les triangles gris représentent la section tessellée.

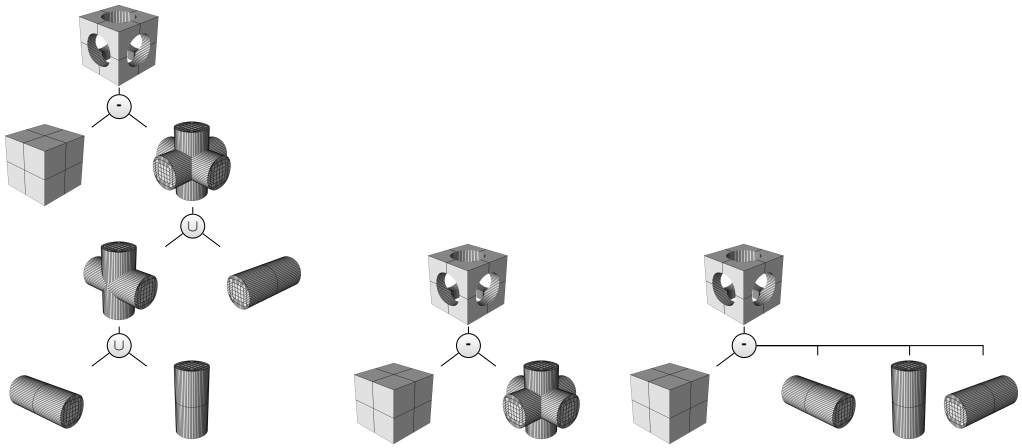
Dans le cas où un triangle est inversé en relation avec d'autres triangles de la *sous-patch*, la triangulation est considérée invalide et on utilise la méthode par découpage d'oreilles.

L'algorithme du découpage d'oreilles tire son nom du fait qu'il tesselle un polygone simple, triangle par triangle, où chaque triangle éliminé forme une "oreille". Une oreille est définie comme un triangle ayant deux de ses arêtes appartenant au polygone, et la troisième arête étant interne et ne croisant aucune autre arête. Puisque chaque polygone simple de plus de trois sommets possède au moins deux oreilles, il suffit d'en choisir une et de la découper pour se retrouver avec un polygone possédant une arête de moins. Récursivement, une oreille à la fois, le polygone est découpé jusqu'au dernier triangle. La figure 3.23 illustre ce procédé en accéléré (retrait de deux triangles à la fois). Différentes métriques peuvent être utilisées pour choisir l'oreille à découper. Par exemple, on peut choisir le triangle selon son aire, son périmètre, ses angles internes, aléatoirement, etc. L'algorithme peut s'effectuer de façon vorace ou par optimisation pour tenter de maximiser la qualité de la tessellation. Dans notre implémentation, nous avons opté pour le procédé vorace avec un coût basé sur l'aire des triangles, ce qui nous donne un bon compromis entre la qualité et le temps d'exécution.

3.2 CSG

3.2.1 Description

Puisqu'un modèle par blocs possède une définition volumique, les opérations booléennes peuvent être employées pour accroître l'expressivité de l'élaboration d'un objet. Les opérations booléennes sont généralement exprimées sous la forme d'un arbre CSG, supportant les opérations standards (union, soustraction, intersection), et où chaque feuille est une primitive de base. Dans notre cas, la primitive de base est un groupe de blocs pour lesquels les connexions ont été exécutées. Un exemple simple d'un tel arbre est montré à la figure 3.24. Afin de mieux supporter



(a) (b) (c)

FIGURE 3.24 – Différentes variations d'arbres de CSG produisant la même géométrie finale. (a) Opérations standards de soustraction et d'union. Les groupes de blocs (les feuilles de l'arbre) possèdent tous un seul bloc. (b) Le groupe de blocs soustrayant (feuille de droite) possède plusieurs blocs formant une géométrie complexe. Aucune fusion entre blocs n'est cependant effectuée. (c) Utilisation de l'opération de soustraction multi-paramètres.

la modélisation par blocs, ainsi que pour lui donner plus de souplesse, différents ajouts ont été apportés à la description de l'arbre.

Dans un premier temps, le support d'opérations multi-paramètres permet de diminuer la complexité d'un arbre sans toutefois en changer son résultat, comme on peut le constater en comparant les figures 3.24a et 3.24c. Puis, deux nouvelles opérations de composition ont été introduites, soient la composition de blocs et la composition d'arbres. L'opération de composition de blocs regroupe tous les blocs en un seul groupe et permet ainsi la connexion entre tous les blocs. Toutefois, cette opération n'est valide que si elle vient avant tout autre type d'opération. La raison est simple : toute autre opération générera des sous-faces partielles aux intersections entre blocs et empêcherait donc la connexion dans cette région, puisque la connexion de blocs s'effectue entre deux sous-faces (complètes). La figure 3.25 montre l'utilisation de cette opération pour fin de création d'un personnage où l'on compose chacun des membres au tronc du personnage. Finalement, l'opération de composition d'arbres permet de fusionner plusieurs arbres de CSG en un seul. Chaque arbre contenu dans un noeud de composition se connecte à l'arbre suivant en s'insérant dans son noeud d'entrée s'il existe, sinon ils sont réunis par une opération d'union. Les figures 3.26 et 3.27 illustrent ce concept avec un exemple, ainsi que le résultat de sa transformation en arbre ne contenant que des opérations standards. Dans ce dernier exemple, l'opération de composition d'arbre permet d'ajouter facilement des éléments architecturaux (porte et fenêtres) à un mur. Chaque porte et chaque fenêtre contient un trou ainsi que la géométrie

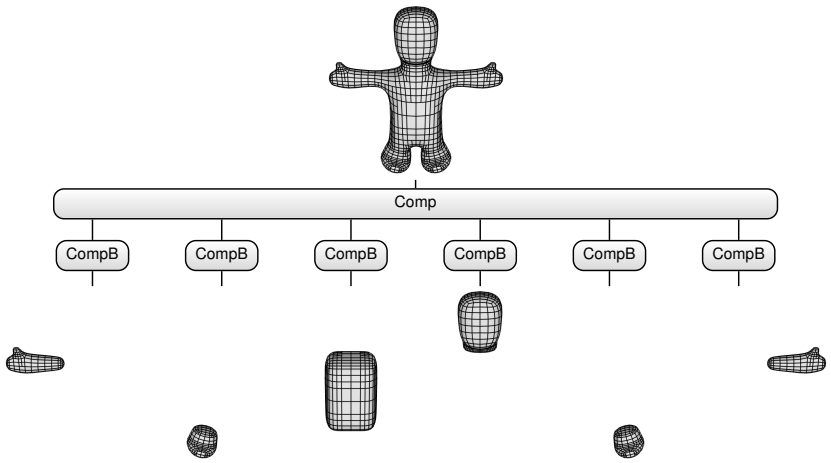


FIGURE 3.25 – Arbre de CSG avec opération de composition de blocs. L'opération de composition regroupe et positionne tous les groupes blocs ensemble et permet la fusion entre ces derniers pour former dans ce cas un personnage.

de l'élément à ajouter. Pour simplifier son utilisation dans un contexte procédural où l'on pourrait vouloir donner la priorité de composition à certains éléments, chaque noeud de composition peut être affublé d'une valeur. Les noeuds contenant les arbres sont par la suite triés selon cette valeur de priorité avant d'être connectés entre eux. Cette fonctionnalité peut être utile, par exemple, si on veut s'assurer que toutes les portes soient posées après que tous les trous et leurs encadrements soient exécutés. La transformation en arbre de CSG standard crée un arbre fortement linéaire (voir figure 3.27), ce qui a comme conséquence de rendre son évaluation plutôt coûteuse. Toutefois, dans bien des cas, l'arbre obtenu peut être facilement transformé en un arbre plus équilibré et de moindre profondeur comme à la figure 3.28. Dans ce dernier cas, puisque les primitives de base ne s'intersectent pas, il est possible de réordonner les opérations sans en affecter les résultats. Après le réordonnancement, il suffit de regrouper les opérations successives de même type en une seule opération.

Les nouvelles opérations sont particulièrement utiles dans le cadre de la modélisation par composants, qui sera introduite au prochain chapitre. Dans ce cas, chaque noeud de composition pourra être attaché à un composant pour ajuster son positionnement.

Différentes techniques existent pour effectuer l'exécution d'opérations booléennes sur une géométrie selon, entre autres, le type de représentation utilisée, la quantité de détails et si la robustesse du résultat est recherchée. Par exemple, pour la géométrie représentée sous forme d'un maillage polygonal, un algorithme de BSP (*Binary Space Partitioning*) [TN87, NAT90, LDS08] peut être utilisé, et même de façon robuste [BF09]. Cependant, lorsque la géométrie en question possède un niveau élevé de détails, les techniques par découpage [LTH86, Hub90, SD07, LBD10]

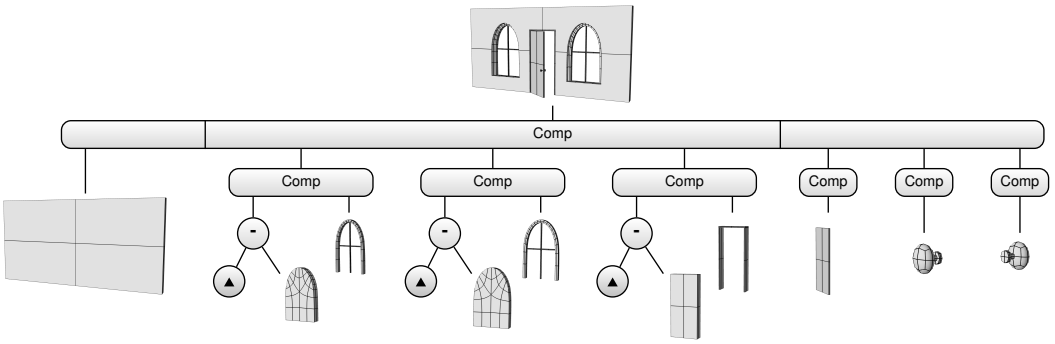


FIGURE 3.26 – Arbre de CSG avec opération de composition. Les noeuds d'entrée (noeud avec un triangle) connectent entre eux le noeud de composition précédent. Lorsqu'un noeud de composition ne contient pas de noeud d'entrée dans son sous-arbre, ce noeud est connecté au précédent par une opération d'union.

s'avèrent nettement plus efficaces. Toutefois, pour les rendre robustes, il est souvent nécessaire d'utiliser de l'arithmétique exacte. Récemment, des techniques mixes ont vu le jour, comme celle de Campen et Kobbelt [CK10] mélangeant découpage et BSP, et celle de Pavic *et al.* [PCK10] alliant le découpage au contourage dual [JLSW02, BPK05]. Toutes deux se servent d'un *octree* pour diviser l'espace de la géométrie en régions affectées par les opérations booléennes de celles intactes. D'autres algorithmes existent lorsque la géométrie prend une autre forme ou est convertie sous forme de voxels [Lai07], de points [AD03] ou encore de demi-espaces [BR96].

Dans notre cas, nous avons élaboré notre propre technique, adaptée à notre représentation par blocs, et basée partiellement sur les travaux de Hubbart [Hub90] et de Smith *et al.* [SD07]. Puisque nous voulons permettre la création d'objets complexes, entre autres, par l'utilisation de fonctions de déplacement et aussi que nous cherchons à garder intacte, le plus possible, la tessellation obtenue par le système de base, l'utilisation d'une technique par découpage nous semble le plus approprié. On recherche aussi un maximum de robustesse, c'est-à-dire un algorithme qui nous retournera le plus souvent possible une géométrie bien formée, sans toutefois exiger que cette dernière soit exacte. En fait, dans bien des cas, une solution approximative est préférable à la solution exacte. Ceci est, entre autres, une des raisons pour laquelle l'utilisation de l'arithmétique exacte n'a pas été envisagée. En plus de cette raison, pour être plus précis, il y a aussi la vitesse et la mémoire.

À moins d'avoir tout le système de modélisation exprimé en arithmétique exacte, il est difficile de garantir que la géométrie de base représente précisément la forme voulue. Dans certains types de modélisation comme dans le cas de la modélisation d'édifices, beaucoup d'éléments sont déposés les uns sur les autres, comme par exemple, les moulures sur les murs

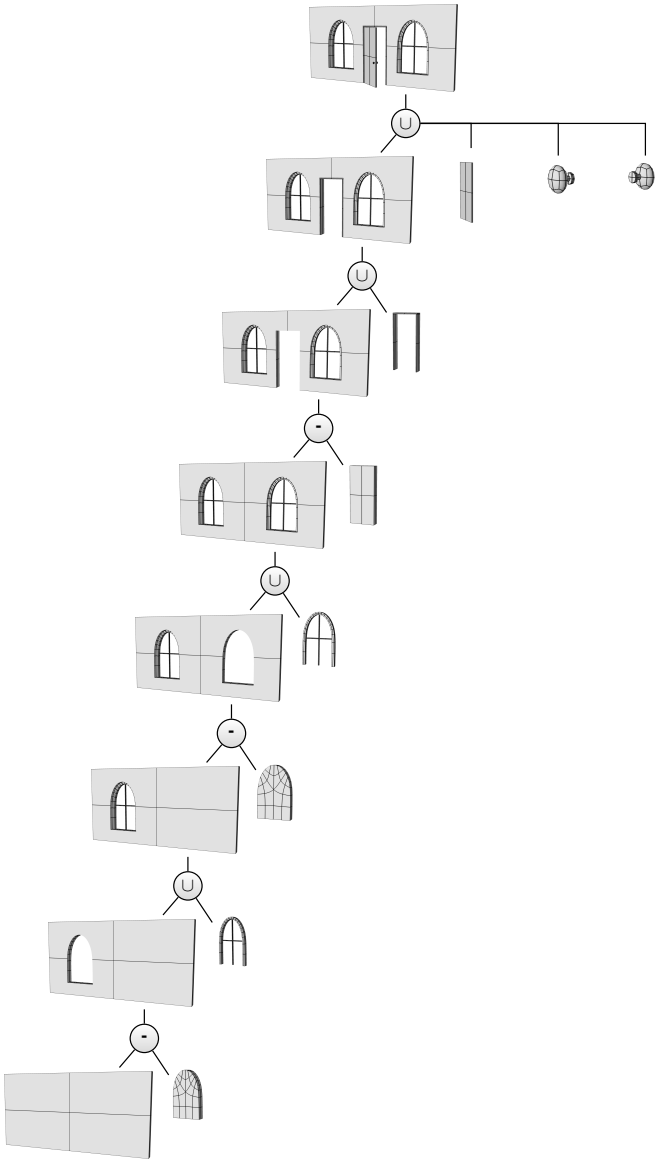


FIGURE 3.27 – Transformation de l'arbre de CSG de la figure 3.26 en arbre ne contenant que des opérations conventionnelles d'union et de soustraction.

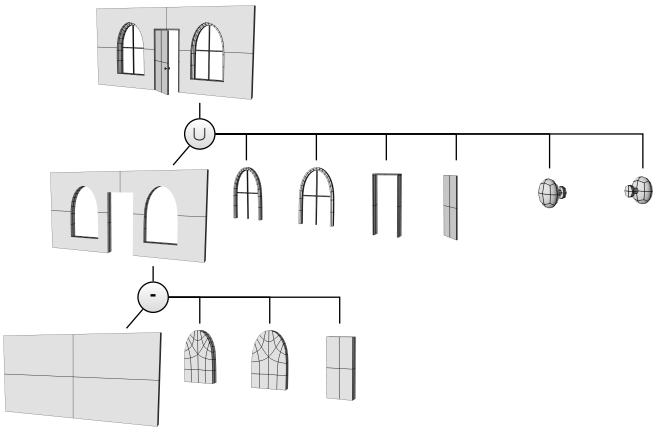


FIGURE 3.28 – Réordonnancement des opérations et optimisation de l'arbre de la figure 3.27. Le réordonnancement est possible puisque les différentes opérations ne s'intersectent pas.

ou les murs entre eux. Ces cas entraînent des problèmes de précision lorsque ces éléments sont non alignés sur les axes, puisqu'ils ne peuvent pas être exprimés précisément. Notre technique va permettre de solutionner ce problème, contrairement aux techniques conventionnelles, en projetant les composants de base (sommets, arêtes et faces) entre eux selon une distance seuil.

Les modifications nécessaires pour supporter les opérations booléennes dans le pipeline de modélisation par blocs décrit à la section 3.1, sont introduites après la subdivision adaptative des *sous-patchs*. La technique comporte les trois étapes principales suivantes :

- Préparation : Transformation de l'arbre de CSG, ajustement du maillage et création de structures de recherche spatiales.
- Découpage : Création de boucles de polylignes formées de segments et de sommets d'intersection.
- Évaluation : Détermination des différentes sections de la surface appartenant au modèle final selon l'arbre de CSG.

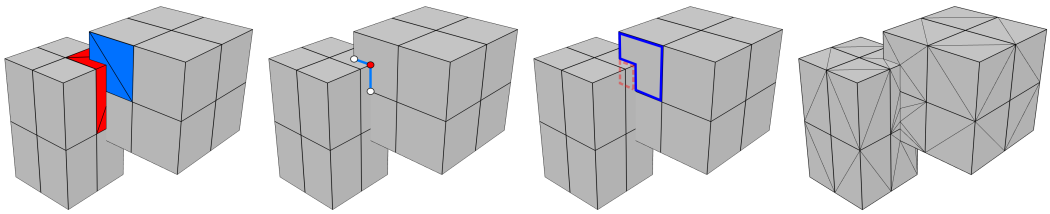
Par la suite, l'étape de tessellation est adaptée pour supporter le découpage occasionné par les intersections de géométries.

3.2.2 Préparation

Avant de procéder au calcul d'intersection entre tous les groupes de blocs, chacun formant les feuilles de l'arbre de CSG, il est nécessaire d'effectuer quelques préparations. Premièrement, on procède à la simplification de l'arbre de CSG comme expliqué précédemment, et on en profite

pour déplacer tous les blocs dans un espace global et pour calculer la boîte englobante de chaque noeud de l'arbre. Puis, après la subdivision adaptative des *sous-patchs*, une linéarisation de toutes les arêtes des *sous-patchs* est effectuée dans le but de simplifier le calcul d'intersection (une *sous-patch* peut maintenant être représentée par deux triangles sans créer de craques entre ses voisins). Cette linéarisation du maillage peut être obtenue, par exemple, en linéarisant chacun des quatre côtés d'une *sous-patch*, pour toutes les *sous-patchs*, de la plus profonde (selon le niveau de subdivision) à la plus simple. La linéarisation d'un des côtés consiste à déplacer chacun des sommets compris entre les sommets extrêmes pour former une droite.

Après avoir linéarisé les arêtes du maillage, différentes structures spatiales sont construites afin d'accélérer les requêtes d'intersection. Puisque l'étape de découpage consiste à découper entre eux tous les triangles s'intersectant, il est impératif de réduire le nombre de paires de triangles testées au minimum. De nombreuse structures spatiales existent, comme les grilles (hiérarchiques ou régulières) et les arbres (alignés sur les axes (AABB), orientés (OBB), *kd-tree*, *octree*, *BSP*, *BIH*), chacune possédant ses caractéristiques propres en termes de mémoire et de performance selon le type de géométrie et de requête. Dans notre cas, on n'a pas affaire à une soupe de triangles ou un maillage sans aucune information, mais plutôt à une géométrie possédant déjà une hiérarchie inhérente (blocs, faces, *patchs*, *sous-patchs* et triangles). En regard à ce dernier point, on a donc opté pour l'utilisation de deux structures imbriquées. La première consiste en une grille hiérarchique [Eri04] contenant toutes les *patchs*, et la seconde, une hiérarchie de boîtes englobantes alignées sur les axes (AABB) et compressée contenant toutes les *sous-patchs* d'une *patch*. Il y aura donc autant, ou presque, d'arbres AABB que de *patchs*. Presque, car il n'est pas nécessaire de construire un arbre pour les *patchs* constituées d'une seule *sous-patch*. La première structure, la grille hiérarchique, permet la modification efficace de la géométrie (ajout, retrait et déplacement de blocs) tout en supportant un niveau de détail très disparate contrairement à la grille régulière. La structure d'arbre AABB compressée, quant à elle, offre aussi de bonnes performances de requêtes, mais permet d'encoder plus efficacement la répartition spatiale de la géométrie, là où elle est plus dense, c'est-à-dire à la surface, surtout en présence de fonctions ou de textures de déplacement. Lors de la modification de la surface d'une région de la géométrie, on a le choix de reconstruire l'arbre au complet, ce qui n'est pas très coûteux puisque la région est restreinte, ou de réajuster l'arbre (*refit*) dans les cas de changements mineurs. La combinaison de ces deux structures adaptées à la géométrie par blocs offre de bonnes performances tout en étant flexible et compacte.



(a) *Sous-patchs* (b) Arêtes et sommets (c) Boucles (d) Tessellation

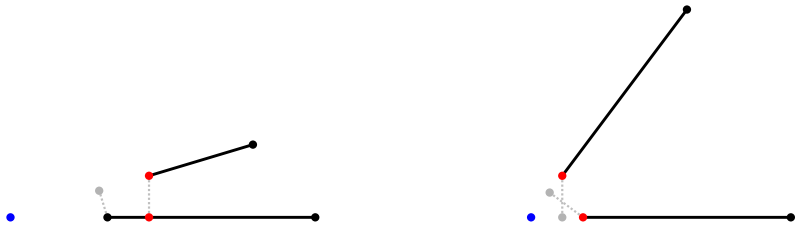
FIGURE 3.29 – Les différentes étapes de l'évaluation d'une opération booléenne d'union. (a) Test d'intersection de la *sous-patch* bleue contre les *sous-patchs* rouges. (b) Insertion de sommets et d'arêtes intersectants. (c) Intersection des arêtes d'une *sous-patch*, raccordement en boucles fermées et vérification contre l'arbre de CSG. (d) Tessellation complète de l'union de deux blocs, sans la géométrie interne.

3.2.3 Découpage

Le processus de découpage débute en trouvant quelles *patchs* s'intersectent grâce à la structure de grille hiérarchique. Pour chaque paire de *patchs* s'intersectant potentiellement, un second test détermine toutes les paires de *sous-patchs* s'intersectant, s'il y a lieu. Ce test consiste à traverser simultanément chacun des arbres AABB des deux *patchs* formant la paire en élaguant les branches des régions disjointes. Finalement, dans le cas d'intersection entre *sous-patchs*, pour chaque *sous-patch* de la paire, des segments et des sommets représentant l'intersection sont créés et assignés à chacune d'entre elles (voir figure 3.29b). Ces segments et sommets sont générés en approximant chaque *sous-patch* par une paire de triangles et en exécutant les quatre tests d'intersection triangle-triangle (figure 3.29a). Après que toutes les intersections aient été calculées, tous les segments associés à une *sous-patch* sont intersectés entre eux. Chaque segment ainsi intersecté est divisé en deux sous-segments au point d'intersection et un nouveau sommet est créé. Par la suite, les segments résultants sont traversés pour former des boucles polygonales non chevauchantes (figure 3.29c). Chacune de ces boucles correspond à une section de la surface (de la *sous-patch*) appartenant à une même région de l'espace selon l'arbre de CSG. Une seule évaluation par boucle est donc nécessaire pour déterminer l'appartenance à la surface finale ou son exclusion.

Création des segments

L'étape clef et la plus difficile à rendre robuste lors du processus de découpage (et même du processus complet de CSG) est la création de segments obtenus par l'intersection de deux triangles. Si l'arithmétique exacte est utilisée pour effectuer les calculs et aussi pour représenter les



(a) Sommet-segment (b) Sommet-sommet

FIGURE 3.30 – Intersection entre deux segments proches. Le sommet bleu représente le point d'intersection entre les deux droites définies par les deux segments, alors que les sommets rouges représentent les points les plus proches appartenant aux segments. En gris (sommets et lignes pointillées) est illustrée la projection des sommets d'un segment sur l'autre segment.

positions des sommets, n'importe quel algorithme d'intersection triangle-triangle peut convenir, tel que celui élaboré par Möller [Mö97]. Toutefois, si l'on veut supporter un certain seuil de distance à l'intérieur duquel on considère qu'une intersection a lieu, on se doit de construire un algorithme dans cette optique. L'ajout d'epsilons bien placés ne suffit pas. Comme expliqué précédemment, c'est exactement ce que l'on recherche pour permettre d'effectuer des intersections approximatives dans le cas de géométries se touchant presque. Puisque l'on doit supporter les cas de "presque intersection", on ne peut pas se contenter de simples calculs d'intersection. Comme on peut le constater à la figure 3.30 dans une simplification 2D du problème où l'on cherche l'intersection approximative entre deux segments, on peut s'apercevoir que le point d'intersection recherché n'est pas le point d'intersection des deux droites, mais bien la projection d'un des sommets sur l'autre segment ou encore sur un des sommets (le sommet de couleur rouge).

Basé sur cette observation, nous avons élaboré un algorithme d'intersection triangle-triangle utilisant principalement le calcul des distances entre leurs différents éléments (face, arête, sommet). L'algorithme consiste à calculer les distances entre chaque paire d'éléments provenant de chaque triangle pour toutes les combinaisons d'éléments. Une paire est considérée comme s'intersectant si la distance entre ses éléments est en-dessous d'un seuil spécifié, et générera un point d'intersection. Lorsque deux points d'intersection sont trouvés, un segment d'intersection est ainsi créé entre eux. Le cas face à face est un cas particulier qui sera discuté un peu plus loin.

Pour le bon fonctionnement de l'algorithme, le calcul de distance doit être effectué selon un ordonnancement bien précis. Ainsi, on débute par les paires reliant les sommets entre eux, puis dans l'ordre, les sommets et les arêtes, les sommets et les faces, les arêtes entre elles et finalement les arêtes et les faces. Au total, un maximum de 48 tests doivent être exécutés comme le démontre la figure 3.31. Aussitôt que deux points d'intersection approximative sont découverts,

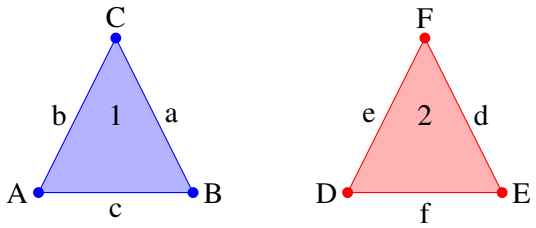


FIGURE 3.31 – Soit la définition ci-haut de deux triangles, l'ensemble des paires d'intersection devant être analysées sont les suivantes :
sommet-sommet (9) : AD, AE, AF, BD, BE, BF, CD, CE, CF
sommet-arête (18) : Af, Ad, Ae, Bf, Bd, Be, Cf, Cd, Ce, Dc, Da, Db, Ec, Ea, Eb, Fc, Fa, Fb
sommet-face (6) : A2, B2, C2, D1, E1, F1
arête-arête (9) : cf, cd, ce, af, ad, ae, bf, bd, be
arête-face (6) : c2, a2, b2, f1, d1, e1.

l'algorithme termine et retourne le segment d'intersection. En fait, deux segments sont retournés, un pour chaque triangle, puisque l'intersection est approximative. Chaque paire d'éléments génère les deux sommets les plus proches, un pour chaque élément. Ces sommets coïncident seulement lorsque l'intersection a réellement lieu et n'est pas approximative. Par conséquent, lors d'intersection approximative, les triangles sont découpés selon un segment légèrement différent (où chaque sommet peut dévier d'une distance inférieure au seuil spécifié) et occasionne par le fait même une craque dans le maillage de la surface finale. Pour remédier à ce problème, il suffit d'effectuer une étape de fusion de sommets après avoir complètement terminé le découpage. Il est intéressant de noter que bien que l'intersection conventionnelle entre deux triangles ne peut générer qu'un seul segment, l'algorithme d'intersection approximative peut en générer plus d'un s'il n'est pas arrêté après l'obtention du premier. C'est entre autres pourquoi il est important de respecter l'ordre des paires mentionné plus haut, et d'arrêter après avoir découvert un segment. Dans le cas où un seul sommet d'intersection est obtenu, il peut être omis sans conséquence. Un autre point important est qu'un élément ne doit participer qu'à la création d'un seul point d'intersection. En d'autres termes, aussitôt qu'un élément est identifié comme intersectant à l'intérieur d'une paire d'éléments, il ne doit plus être inclus dans les autres paires d'éléments testées.

Afin d'accélérer le calcul d'intersection et d'éviter d'avoir à analyser chaque combinaison d'éléments, on peut commencer par déterminer la distance de chaque sommet d'un triangle au plan sous-tendant de l'autre triangle de la paire (ci-après appelé "opposé"). Tous les sommets compris à une distance inférieure au seuil spécifié sont considérés comme étant situés à la surface du plan. Si tous les sommets d'au moins un des deux triangles se retrouvent du même côté (et

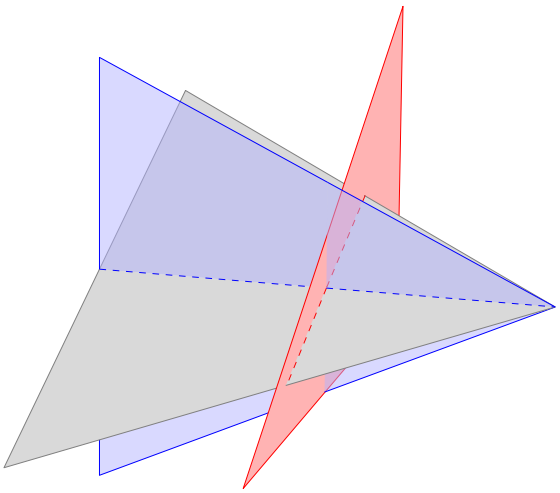


FIGURE 3.32 – Intersection entre trois triangles. Les lignes pointillées bleue et rouge représentent les segments d'intersection calculés et accumulés pour le triangle gris. On peut noter que ces deux segments se croisent et doivent aussi être découpés entre eux.

aucun sommet sur le plan), il n'y a alors pas d'intersection. De plus, si tous les sommets d'un triangle se retrouvent sur le plan opposé, il s'agira d'une intersection face à face qui sera rejetée puisque l'intersection potentielle sera retrouvée plus simplement à l'aide d'un de ses voisins adjacents. La distance ainsi calculée d'un sommet au plan opposé peut aussi servir à éliminer rapidement des paires d'éléments. Par exemple, si un sommet est à une distance supérieure au seuil, les trois paires d'éléments sommet-sommet, ainsi que les trois paires sommet-arêtes et la paire sommet-face n'ont pas besoin d'être calculées. Finalement, les paires arête-arête et arête-face sont évaluées seulement si les sommets d'une des arêtes se retrouvent de chaque côté du plan (signes opposés pour ces distances).

Création des boucles

La seconde et dernière étape du découpage, exécutée après avoir trouvé toutes les intersections entre *sous-patchs*, consiste à former des boucles de segments pour chaque *sous-patch*. Dans le cas d'une *sous-patch* n'ayant pas participé à des intersections, la boucle est simplement constituée de son contour. Elle sera tessellée dans une étape subséquente comme expliqué à la section 3.1.4 si elle fait partie de la surface finale. Dans le cas contraire, lorsqu'une *sous-patch* possède un ensemble de segments d'intersection, un algorithme divisé en trois étapes est exécuté pour extraire une liste de boucles :

- Découpage des segments : Lors du calcul d'intersection entre deux triangles, il est avantageux de ne pas effectuer le découpage des triangles immédiatement, mais plutôt d'ac-

cumuler les segments d'intersection. Il en résulte un algorithme plus robuste, plus rapide et produisant un nombre total de triangles moins élevé. Cependant, il y a un désavantage concernant les cas traités. Lorsqu'au moins trois triangles de plans différents s'intersectent en un point, deux segments sont créés et accumulés par triangle, mais ces deux segments se croisent et doivent aussi être découpés entre eux, ce qui est le but de cette étape (voir figures 3.32 et 3.33b).

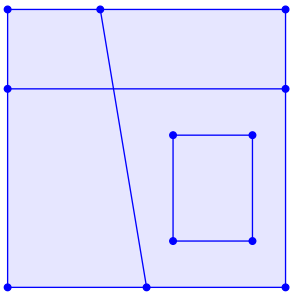
- Connexion des segments : Pour simplifier la tessellation, on recherche la formation de boucles simples et non imbriquées définissant ainsi des polygones sans trou. Lorsqu'il y a imbrication, pour corriger ce cas, on ajoute un segment reliant les boucles entre elles. Bien qu'à ce stade les boucles ne sont pas encore formées, il est tout à fait possible d'effectuer ce travail en reliant les amas de segments disparates (figure 3.33c).

- Création des boucles : Un à un les segments contenus dans une *sous-patch* sont traversés et reliés ensemble en suivant leurs connexions entre sommets (voir figure 3.33d).

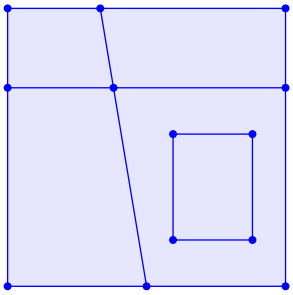
Pour simplifier les différents algorithmes, la plupart des calculs sont effectués en 2D. Pour ce faire, les segments sont projetés sur le plan formé par les deux axes mineurs de la *sous-patch*. L'axe majeur correspond à l'axe canonique en espace monde dont la coordonnée de la normale moyenne de la *sous-patch* est la plus grande en valeur absolue, alors que les axes mineurs sont les deux axes canoniques restants. La projection consiste simplement à éliminer la coordonnée majeure (X, Y ou Z) des sommets.

Pour détecter efficacement les segments s'intersectant, on débute par les trier selon un de leurs deux axes. Ensuite, en traversant les segments en ordre croissant de leur axe de tri, on compare un segment à la fois, aux segments suivants pour lesquels leurs intervalles chevauchent celui du segment courant. Les paires de segments se chevauchant sont ainsi testées plus précisément pour détecter s'il y a intersection et calculer en quel point, si c'est le cas. Le nouveau point d'intersection, lorsqu'il y en a un, est ajouté à la *sous-patch*, alors que les deux segments sont séparés en quatre segments et réinsérés dans la liste triée. Pour augmenter la robustesse de l'algorithme, de la même façon que lors du test triangle-triangle, l'intersection est calculée en vérifiant dans l'ordre toutes les combinaisons d'éléments (sommet-sommet, sommet-arête et arête-arête).

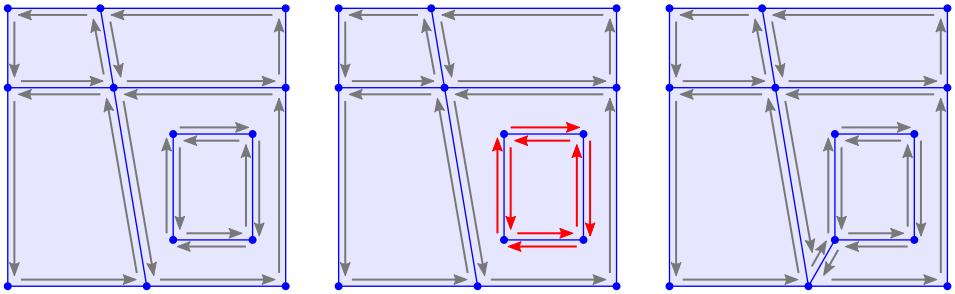
Avant d'entamer les deux prochaines étapes, une structure de voisinage est formée. Dans cette structure, chaque segment appartenant au contour de la *sous-patch* est transformé en demi-arête



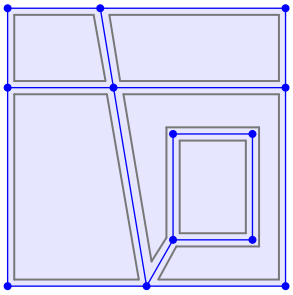
(a) La *sous-patch* et ses segments



(b) Découpage des segments



(c) Connexion des segments



(d) Création des boucles

FIGURE 3.33 – Les trois étapes de la création de boucles pour une *sous-patch*. L'étape de connexion des segments est subdivisée en trois sous-étapes, soient (de gauche à droite) la création de demi-arêtes, la séparation des segments en groupes (ici deux groupes sont trouvés) et la connexion des groupes. La connexion s'effectue à partir du sommet minimum du groupe en rouge au sommet le plus près, inférieur, et sans intersection avec les autres demi-arêtes.

orientée en sens anti-horaire, alors que pour les autres segments internes, deux demi-arêtes de sens opposés sont créées. Finalement, une liste de toutes les demi-arêtes sortantes est assignée pour chaque sommet.

Afin de connecter les boucles imbriquées entre elles, on doit premièrement déceler les amas de segments (sous forme de demi-arêtes) disjoints. Un algorithme de remplissage par diffusion est utilisé à cette fin. Une demi-arête est choisie, puis partant de son sommet terminal, chaque demi-arête sortante est ajoutée au groupe en formation et récursivement le processus est réitéré pour ces demi-arêtes. Une demi-arête n'est traitée qu'une seule fois. Lorsque toutes les demi-arêtes joignables ont été insérées dans le groupe, un nouveau groupe est débuté avec une demi-arête non choisie préalablement, s'il en reste une, et ainsi de suite jusqu'à l'obtention de tous les groupes. Pour chaque groupe, le sommet minimum, selon l'axe de tri initial (dans le cas où plusieurs sommets partagent la même position selon le premier axe, un tri selon le second axe sert à les départager), est conservé. Il servira de point d'ancrage pour le relier à un autre groupe. Après avoir trié tous les sommets, chaque sommet minimum de chaque groupe est relié au premier sommet inférieur (selon l'axe de tri) pour lequel le segment résultant n'intersecte aucun autre segment. Ce segment est converti en deux demi-arêtes opposées et la structure de voisinage est mise à jour en conséquence.

Finalement, après tout ce traitement accompli sur les segments accumulés pour la *sous-patch*, les boucles peuvent être formées. Une première demi-arête de départ est choisie arbitrairement, puis à partir de son sommet terminal, la demi-arête suivante est sélectionnée. Cette demi-arête, choisie parmi toutes les demi-arêtes sortantes du sommet terminal, est celle formant le plus petit angle entre la demi-arête précédente et elle, en sens anti-horaire. Le procédé est par la suite réitéré jusqu'à la fermeture de la boucle, c'est-à-dire, jusqu'à ce qu'on rejoigne la demi-arête de départ. Pour obtenir toutes les boucles, on répète la construction à partir d'une demi-arête non choisie, s'il y en reste, jusqu'à épuisement.

3.2.4 Évaluation

Avant d'effectuer la tessellation de la surface, il faut définir quelle partie (*patchs*, *sous-patchs* et boucles) appartient à la surface en tenant compte de la sémantique de l'arbre de CSG. Afin de déterminer cette appartenance, pour chacune des boucles de chaque *sous-patch*, un rayon est lancé d'un point contenu à l'intérieur de la boucle, en direction de sa normale. Ce rayon est testé contre toutes les autres *sous-patchs* et l'ensemble des intersections obtenues, lorsque

évalué avec l'arbre de CSG, permet de déterminer si le point d'origine du rayon (et aussi toute la surface délimitée par la boucle) appartient au maillage final (après le CSG). Dans le cas où la boucle appartient à la surface, elle est étiquetée de la sorte, ainsi que la *sous-patch* et la *patch* la contenant. De cette façon, lors de la tessellation de l'objet, des *patchs* et des *sous-patchs* entières peuvent être omises si elles ne sont pas étiquetées. Dans certains cas, en plus d'être étiquetées comme appartenant à la surface, les normales ainsi que la direction de la triangulation doivent être inversées. Ces cas surviennent pour les régions provenant d'une opération de soustraction.

Pour que l'évaluation d'un rayon et de ses intersections retourne la bonne information d'appartenance à la surface, lors de la comparaison avec l'arbre de CSG, il est fondamental que l'ensemble des intersections soit distinct. Un tel problème peut survenir lorsque le rayon croise la surface exactement sur une arête. Dans certains cas l'intersection sera nulle, et dans d'autres, simple ou encore double. Pour remédier à ce problème, on peut éliminer les intersections considérées doubles selon un seuil et leur orientation, mais il est difficile de garantir la robustesse du résultat. Nous avons opté, avec succès, pour un calcul d'intersection basé sur les travaux de Dachsbacher *et al.* [DSD$^+$09]. La façon dont l'intersection est calculée garantit son unicité dans le cas où le rayon passe par une arête ou un sommet.

Puisque bien des *patchs* voisines appartiennent à une même région de l'espace vis-à-vis de l'arbre de CSG, il n'est pas nécessaire de lancer un rayon pour chaque *patch* et chaque *sous-patch*. À partir d'une *sous-patch* dont l'appartenance a été déterminée par lancer de rayon, l'information de visibilité peut être transmise à ses *sous-patchs* et *patchs* voisines par un algorithme de remplissage par diffusion, si elles n'ont pas été intersectées par d'autres *sous-patchs*. En effet, la région de visibilité à la surface change uniquement lors de la rencontre de segments d'intersection.

3.2.5 Tessellation

Peu de changements sont nécessaires à l'étape de tessellation pour supporter l'ajout d'opérations booléennes. Puisque la forme des boucles créées à l'étape de découpage peut être complexe, l'utilisation exclusive de la tessellation par découpage d'oreilles [Hel01] est de rigueur pour ces boucles. Autre petit changement, lorsqu'indiqué, la tessellation est accomplie dans le sens contraire et les normales doivent être inversées. Cette information est transmise à partir de l'évaluation de l'arbre de CSG. Finalement, la tessellation doit être restreinte aux régions de la surface finale. À cette fin, les *patchs* et les *sous-patchs* contenant au moins une boucle visible sont étiquetées comme tel, alors que chaque boucle est étiquetée comme visible ou non. Cette

propagation d'information permet de retrouver efficacement les surfaces à tesseller.

3.3 Langage

Jusqu'à maintenant, nous avons décrit en détail le fonctionnement des différents algorithmes nécessaires pour mettre en place un système de modélisation par blocs. Toutefois, aucune information spécifique quant à la façon de décrire la géométrique par blocs ainsi que son inclusion dans des arbres de CSG, n'ont encore été données. Bien que le système de modélisation par blocs est un bon candidat pour être intégré à l'intérieur d'un modeleur sous la forme d'un système de sculpture, nous nous sommes attardés dans notre recherche sur son utilisation surtout dans un contexte procédural. Nous en avons fait la pierre angulaire de la partie géométrique de notre système de modélisation de haut niveau par composants, qui est présenté au prochain chapitre.

Nous proposons donc pour des fins procédurales un langage de description géométrique construit à partir du langage de programmation *Lua* [IdFF96, IdFF05], en y ajoutant des extensions spécifiques. La description qui suit peut tout aussi bien être effectuée à l'aide d'un autre langage de script que *Lua*, comme *Python* ou *Ruby*. *Lua* a été choisi principalement pour sa simplicité d'utilisation et d'extension, ainsi que pour ses performances.

L'arbre de CSG est décrit de façon immédiate à l'aide de fonctions dénotant le début et la fin de chaque noeud, et est présenté dans un ordre dit de traversée en profondeur. Les commandes permises pour d'écrire un arbre peuvent être classifiées en trois groupes présentés dans la table 3.1.

La création de blocs à l'aide de la commande *block* doit toujours se faire à l'intérieur d'un groupe de blocs (*blocksBegin* et *blocksEnd*). Plusieurs variations de la commande existent, de la plus simple en spécifiant un seul sommet, à la plus complète en spécifiant les huit sommets. Lorsqu'un seul sommet est spécifié, un bloc de rayon unitaire canonique est créé, centré autour du sommet. Lorsque deux sommets sont passés en paramètres, ils définissent les coins opposés d'un prisme rectangulaire orienté sur les axes. Outre la spécification des sommets, le groupe d'attraction (g=), la subdivision des faces (s=), la saillance des arêtes (c=) et un identificateur de matériau (*mat*=) peuvent être passés en paramètres. Finalement, la commande *attraction* permet de définir une paire d'identificateurs de groupe pour lesquels les blocs peuvent fusionner ensemble. Le champ d'action est limité au groupe de blocs courant.

Pour permettre un positionnement plus souple des blocs, autre que le noeud de transformation

Type	Commande	
	Ouverture	Fermeture
CSG	*differenceBegin* *intersectionBegin* *transformBegin* *unionBegin*	*differenceEnd* *intersectionEnd* *transformEnd* *unionEnd*
Composition	*compositeBegin* *compositeBlocksBegin* *inputNode*	*compositeEnd* *compositeBlocksEnd*
Géométrique	*blocksBegin* *block* *attraction*	*blocksEnd*
	scopeBegin *rotate* *scale* *translate*	*scopeEnd*
	displacement *mapping*	

TABLE 3.1 – Commandes de traitement géométrique.

(*transformBegin* et *transformEnd*), une pile de transformations matricielles est accessible à l'aide des commandes *translate*, *rotate*, *scale*, *scopeBegin* et *scopeEnd*. Les commandes *translate*, *rotate*, *scale* modifient la matrice courante, alors que la commande *scopeBegin* empile une nouvelle matrice et *scopeEnd* dépile la courante. De plus, toutes les commandes de création de noeud ont pour effet d'empiler une nouvelle matrice, alors que les commandes de terminaison de noeud la dépile. Lors de la création d'un bloc, la position des sommets sera transformée par la pile matricielle complète.

Il est aussi possible de décrire l'apparence de la surface des blocs. Pour ce faire, deux commandes sont disponibles. La première commande (*mapping*) permet de définir la paramétrisation de surface utilisée pour l'application de textures. Cette commande prend en paramètre une fonction qui est exécutée pour chaque sommet de chaque *patch* de la surface obtenue après la fusion des blocs, et doit retourner ses coordonnées u et v. La deuxième commande (*displacement*) prend en paramètre une fonction de déplacement pour décrire la micro-géométrie de la surface. Ces deux commandes ont pour champ d'action tous les blocs créés après elles. Ainsi, il est possible de définir plusieurs fonctions de paramétrisation et de déplacement à l'intérieur d'un arbre de CSG.

```
displacement(
   function( IN )
      local p = IN.pos
      return p + IN.n * (perlin1( p*2 )*0.2)
   end
)

blocksBegin()
   block{ {-1,-1,-1}, {1,1,1} }
blocksEnd()
```

LISTAGE 3.2 – Exemple de définition d'une fonction de déplacement. Le résultat est une sphère déformée par un bruit de Perlin.

```
differenceBegin()
   blocksBegin()
      block{ {-1.5,-1.5,-1.5}, {1.5,1.5,1.5}, c=0xfff }
   blocksEnd()
   unionBegin()
      unionBegin()
         blocksBegin()
            block{ {-2,-1,-1}, {2,1,1}, c=0xff0 }
         blocksEnd()
         blocksBegin()
            block{ {-1,-2,-1}, {1,2,1}, c=0xf0f }
         blocksEnd()
      unionEnd()
      blocksBegin()
         block{ {-1,-1,-2}, {1,1,2}, c=0x0ff }
      blocksEnd()
   unionEnd()
differenceEnd()
```

LISTAGE 3.3 – Pseudocode pour générer l'exemple de CSG illustré à la figure 3.24.

Un programme peut faire appel à un autre programme externe par l'entremise de la commande *execute*. Cette commande prend comme paramètres le nom du fichier (programme) à exécuter sur place, ainsi que les paramètres qui lui seront passés. Ses paramètres pourront être récupérés à l'intérieur du programme comme illustré aux lignes 1 à 7 du listage 3.4.

Un exemple simple de programme décrivant le cube troué de trois cylindres vu à la figure 3.24 est présenté par le listage 3.3, alors qu'un exemple d'utilisation de fonction de déplacement est montré au listage 3.2. Finalement, le listage 3.4 présente un exemple plus complexe, construisant une fenêtre paramétrable à l'aide d'opérations de composition.

```
local params = ... or {}
local h  = params.h or 1
local w  = params.w or 1
local d  = params.d or 0.1
local d2 = params.d2 or 0.1
local t  = params.t or 0.1
local f  = params.f or 0.04

local n    = 4
local sa   = 0.5
local inc  = 0.5/n
local hinc = inc/2

```

```
14 local c = component{ id="window", size={w,h,d}, connectorPosition={w/2,h/2,0} }
15 compositeBegin(c,100)
16    differenceBegin()
17       inputNode()
18       -- Hole.
19       blocksBegin()
20          attraction(0,0)
21          block{ {0,0,0}, {w/2,h/2,d}, c=0x139, g=0, mat=0 }
22          block{ {w/2,0,0}, {w,h/2,d}, c=0x2c9, g=0, mat=0 }
23          local a = sa
24          for i=0,n-1 do
25             local sina = sin( vec3(a,a-hinc,a-inc) )
26             local cosa = cos( vec3(a,a-hinc,a-inc) )
27             sina = sina*(h/2) + (h/2)
28             cosa = cosa*(w/2) + (w/2)
29             block{
30                {cosa.x,sina.x,0}, {w/2,h/2,0},
31                {cosa.y,sina.y,0}, {cosa.z,sina.z,0},
32                {cosa.x,sina.x,d}, {w/2,h/2,d},
33                {cosa.y,sina.y,d}, {cosa.z,sina.z,d},
34                c=0x036,g=0,mat=0
35             }
36             a = a - inc
37          end
38       blocksEnd()
39    differenceEnd()
40    -- Frame
41    if params.frame then
42    blocksBegin()
43       attraction(0,0)
44       local a = sa
45       for i=0,n-1 do
46          local sina  = sin( vec3(a,a-hinc,a-inc) )
47          local cosa  = cos( vec3(a,a-hinc,a-inc) )
48          local sina0 = sina*(h/2) + (h/2)
49          local sina1 = sina*(h/2-t) + (h/2)
50          local cosa0 = cosa*(w/2) + (w/2)
51          local cosa1 = cosa*(w/2-t) + (w/2)
52          block{
53             {cosa0.x,sina0.x,0},  {cosa1.x,sina1.x,0},
54             {cosa0.y,sina0.y,0},  {cosa1.y,sina1.y,0},
55             {cosa0.x,sina0.x,d2}, {cosa1.x,sina1.x,d2},
56             {cosa0.y,sina0.y,d2}, {cosa1.y,sina1.y,d2},
57             c=0x0f0,g=0,mat=1
58          }
59          block{
60             {cosa0.y,sina0.y,0},  {cosa1.y,sina1.y,0},
61             {cosa0.z,sina0.z,0},  {cosa1.z,sina1.z,0},
62             {cosa0.y,sina0.y,d2}, {cosa1.y,sina1.y,d2},
63             {cosa0.z,sina0.z,d2}, {cosa1.z,sina1.z,d2},
64             c=0x0f0,g=0,mat=1
65          }
66          a = a - inc
67       end
68       block{ {0,0,0},   {t,h/2,d2}, c=0x3f9, g=0, mat=1 }
69       block{ {w-t,0,0}, {w,h/2,d2}, c=0x3f9, g=0, mat=1 }
70       -- inside cross.
71       block{ {(w-f)/2,0,f},       {(w+f)/2,h-t,f*2}, c=0xfff, g=1, mat=2 }
72       block{ {t,(h-f)/2,f},   {(w-f)/2,(h+f)/2,f*2}, c=0xfff, g=1, mat=2 }
73       block{ {(w+f)/2,(h-f)/2,f}, {w-t,(h+f)/2,f*2}, c=0xfff, g=1, mat=2 }
74    blocksEnd()
75    end
76 compositeEnd()
77 return c
```

LISTAGE 3.4 – Pseudocode pour générer une fenêtre ainsi que son trou que l'on retrouve à la figure 3.26.

3.4 Utilisation et limites

Notre expérience nous a démontré que ce processus de description d'objets à l'aide de blocs est rapide et assez facile d'utilisation. Il n'y a pas de restriction sur le type de topologie qu'il est possible de reproduire lorsque l'on procède dans un premier temps par la construction d'une forme approximative, et par l'ajout de détails par texture de déplacement par la suite. Bien que nous n'ayons pas rencontré de problèmes jusqu'à maintenant, nous n'avons pas essayé de construire beaucoup de maillages de contrôle précis utilisant seulement des blocs. Cependant, nous pouvons spéculer que certains cas pourraient s'avérer plus difficiles.

Par exemple, dans un cas où l'on doit faire face à un nombre élevé de blocs de différentes échelles, le fait que chaque face soit limitée à une grille fixe et régulière de sous-faces pourrait être restrictif dans la forme de connexion. Ce cas pourrait nécessiter de diviser un bloc en plusieurs petits blocs. Permettre la subdivision de sous-faces en une grille multi-niveaux pourrait aider dans ces cas.

Aussi, dans certains cas avec beaucoup de connexions, il peut s'avérer un peu pénible d'assigner correctement les bons identificateurs de groupe pour assurer que les connexions résultantes soient celles désirées. Toutefois, cette tâche devrait s'avérer beaucoup plus facile que de traiter la formation de la topologie manuellement.

3.5 Résultats

Les figures 3.34 à 3.42 montrent des objets modélisés exclusivement avec des blocs. Ils illustrent la flexibilité des blocs comme primitives alors qu'ils sont utilisés pour modéliser des objets architecturaux, tels que des édifices et des escaliers, ainsi des formes plus organiques, telles que des arbres et des personnages. Les objets contenant à la fois un mélange d'arêtes vives et lisses, comme des chaises, sont également bien adaptés à être modélisés avec des blocs.

Tous les édifices des figures 3.40 à 3.42 et les escaliers de la figure 3.39 sont modélisés procéduralement à l'aide du système de composants [LHP11a] décrit au prochain chapitre. Ce système procédural utilise les blocs comme base pour générer la géométrie, et par conséquent, gagne plusieurs caractéristiques intéressantes : chaque entité a un volume (i.e. les murs ne sont pas minces comme du papier), et la tessellation est étanche et adaptative. De plus, le CSG permet l'insertion facile de portes et fenêtres sur des murs existants en soustrayant d'abord des blocs du mur pour créer un trou, et ensuite en ajoutant (par union) la porte ou fenêtre. Pour

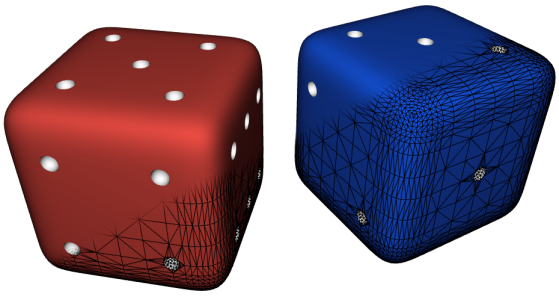


FIGURE 3.34 – La forme générale de chaque dé est modélisée avec un bloc ayant une subdivision de face de 4×4, et toutes les arêtes sont lisses. Chaque point d'un dé est modélisé comme un petit bloc ayant une seule sous-face par face et toutes les arêtes sont lisses. Chaque point est soustrait par CSG de la forme générale du dé. Les lignes noires montrent la tessellation finale.

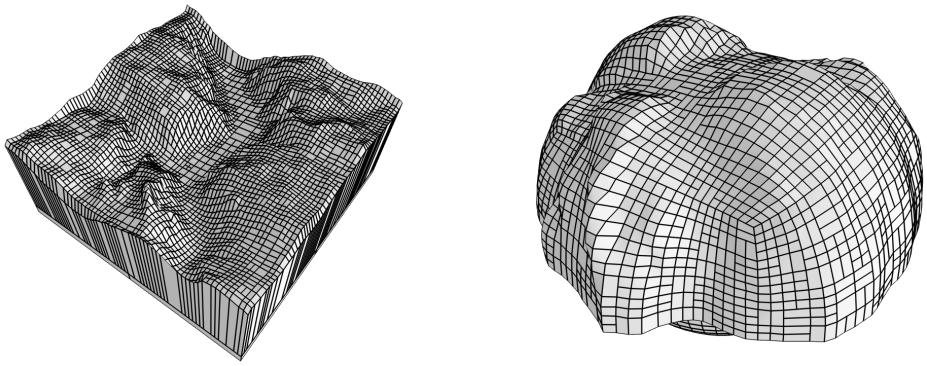


FIGURE 3.35 – Un terrain (sous la forme d'une champ d'élévation) est déplacé (gauche) sur la face planaire vive d'un bloc, et (droite) sur une face lisse et arrondie. Les sous-faces sont dessinées à l'aide de contours noirs.

mieux supporter des simulations physiques sur des objets, nous pourrions aisément étendre la description des blocs pour inclure des propriétés de matériaux.

3.6 Comparaison avec d'autres systèmes de modélisation

De nombreuses techniques et primitives de modélisation existent, chacune ayant son propre ensemble d'avantages et d'inconvénients. Parmi elles, les surfaces implicites comme le *blobtree* [WGG99] et les *ZSpheres* [PIX11] sont les plus près de notre primitive de bloc.

3.6.1 Surfaces implicites

Les surfaces implicites, aussi appelées *metaballs* ou *blobbies*, ont été introduites en infographie par Blinn [Bli82] comme l'isosurface d'une fonction de champ définie par la somme de fonctions simples telle que la distance gaussienne à un point. Dans le *blobtree* [WGG99], le champ de densité est décrit par un arbre d'opérations supportant la fusion, la déformation et les

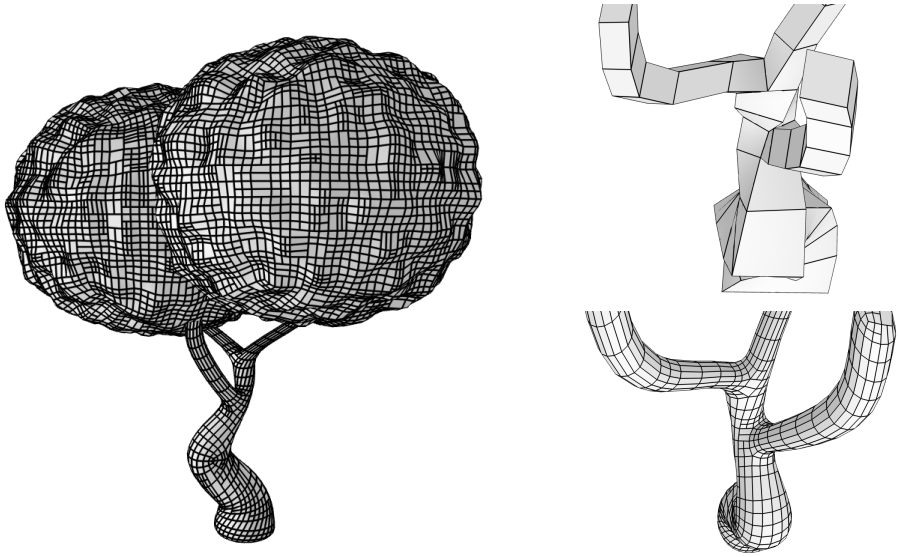


FIGURE 3.36 – Arbre stylisé. Le tronc tordu est modélisé par dix blocs, chaque branche par cinq à sept blocs, et le feuillage par de larges blocs. Toutes les arêtes sont lisses. Le bloc où se connectent les deux branches les plus hautes formant une intersection en Y, a sa face du dessus divisée en 2×1. Les images de droite montrent une vue rapprochée de la connexion des branches, où celle du haut montre les blocs et celle du bas les *sous-patchs* résultantes. L'arbre se compose de 30 blocs, 502 *patchs*, 13 498 *sous-patchs* et 25 778 triangles. Les *sous-patchs* sont dessinées à l'aide de contours noirs.

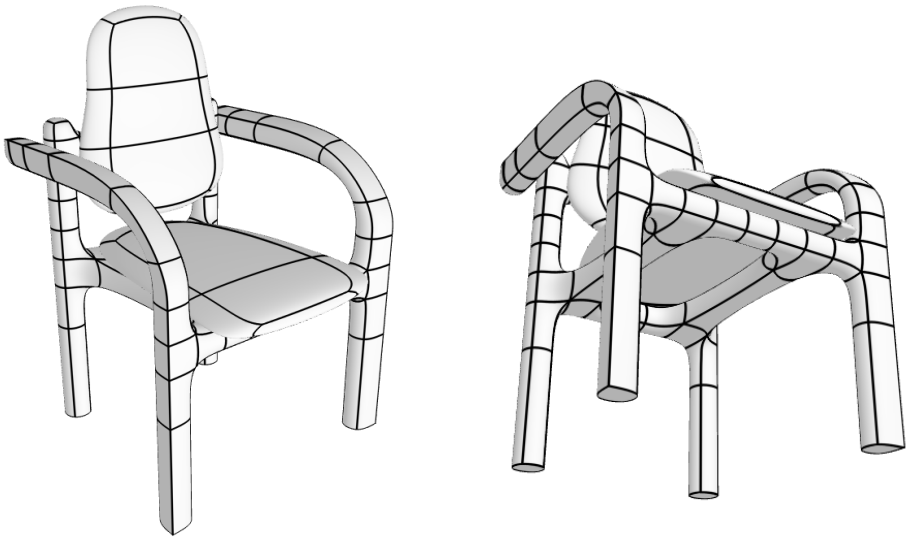


FIGURE 3.37 – Deux vues d'une chaise de bureau. Le bas de chaque patte est modélisé à l'aide d'arêtes vives, ainsi qu'une arête s'élevant jusqu'à former un bras. Les sous-faces sont dessinées par des lignes noires.

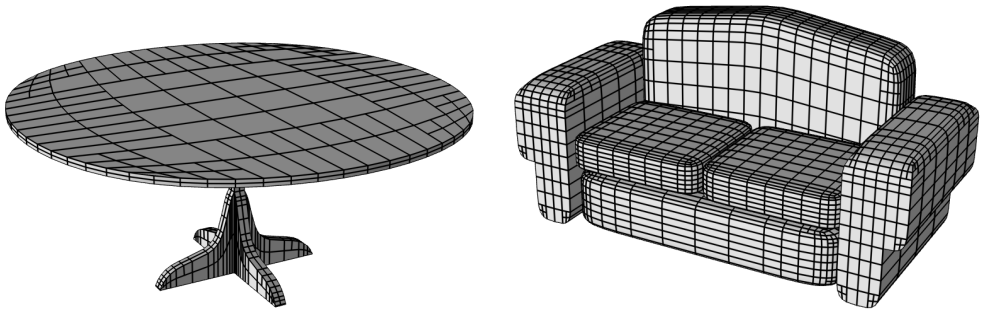


FIGURE 3.38 – Deux meubles de salon, fabriqués par un mélange de connexions et d'opérations booléennes d'union. La table ronde est composée de 392 *patchs* et de 888 *sous-patchs*, alors que le divan est composé de 2416 *patchs* et de 4068 *sous-patchs*.

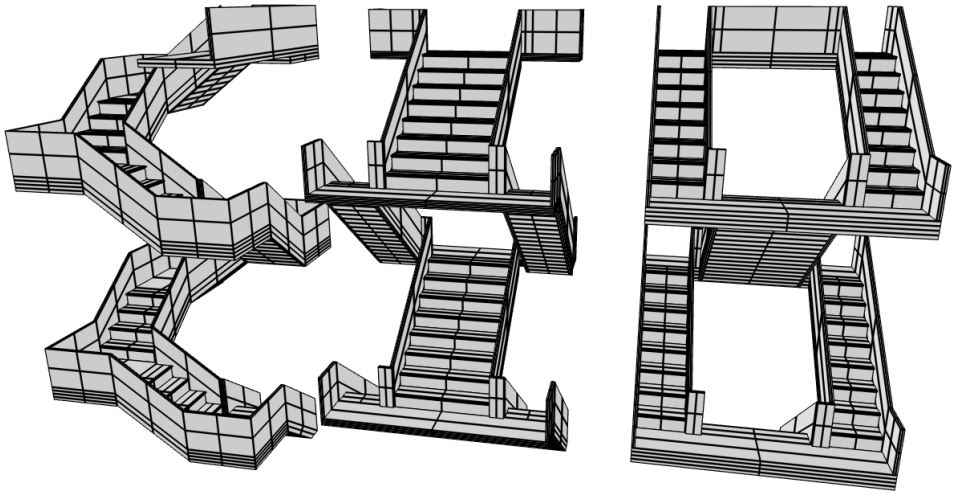


FIGURE 3.39 – Escalier procédural modélisé avec des blocs aux arêtes vives. Les *patchs* sont démarquées par les contours noirs.

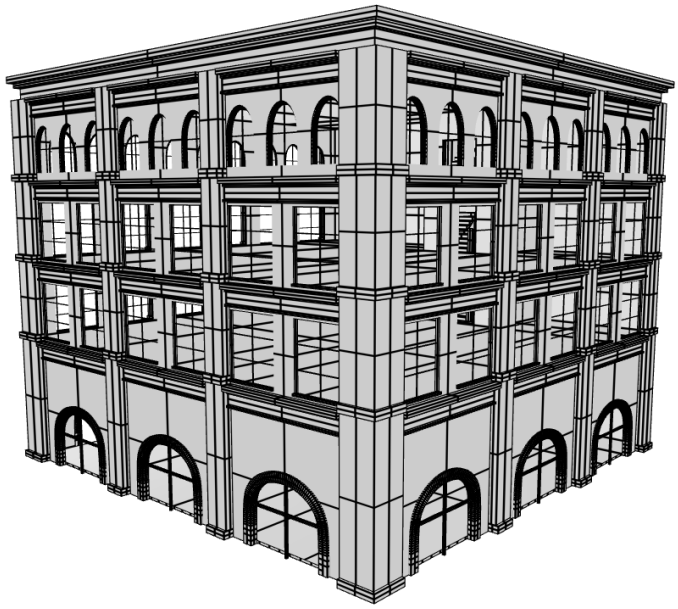


FIGURE 3.40 – Un immeuble à bureaux de quatre étages, principalement composé d'espaces vides, sauf pour un escalier. Il est formé de 1 694 blocs, 34 864 *patchs*, 62 836 *sous-patchs* et de 105 264 triangles.

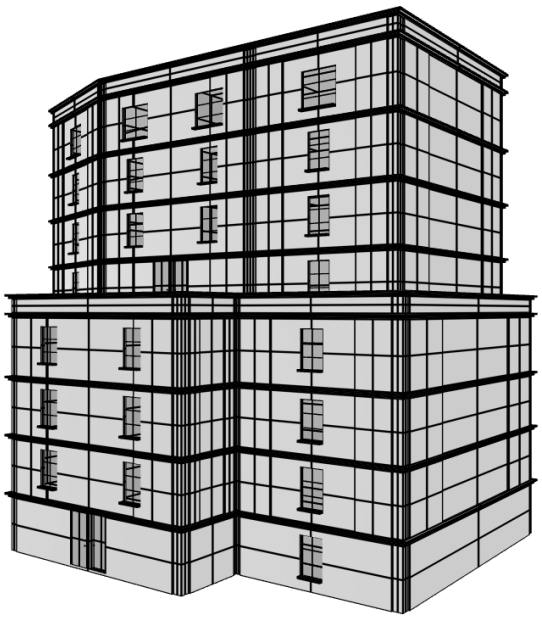


FIGURE 3.41 – Le bâtiment hôtel dispose d'une terrasse à mi-niveau, et de chambres non meublées. Il compte 986 blocs, 22 832 *patchs*, 23 408 *sous-patchs* et 36 658 triangles.

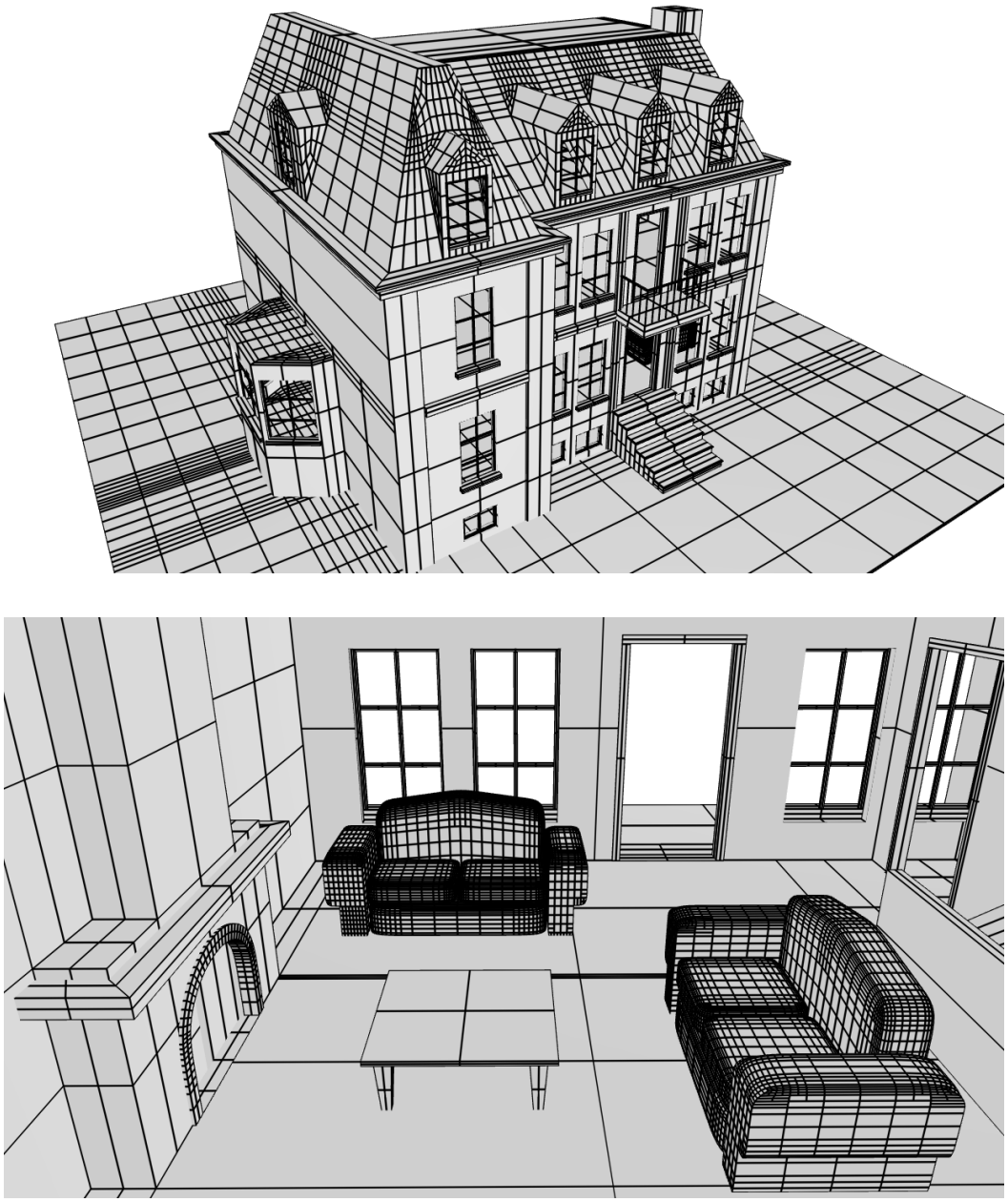


FIGURE 3.42 – L'image du bas est une vue intérieure de la maison de trois étages figurant à l'image du haut. Elle est construite à partir de 1 407 blocs, 28 456 *patchs*, 33 036 *sous-patchs* et 58 632 triangles. Pour tous les édifices, les fenêtres et les portes sont le résultat d'opérations de CSG.

opérations booléennes (voir figure 3.43). Pour mieux contrôler la localité de la fusion, Bernhardt *et al.* [BBCW10] ont présenté une solution limitant sa région à l'intersection de ses primitives.

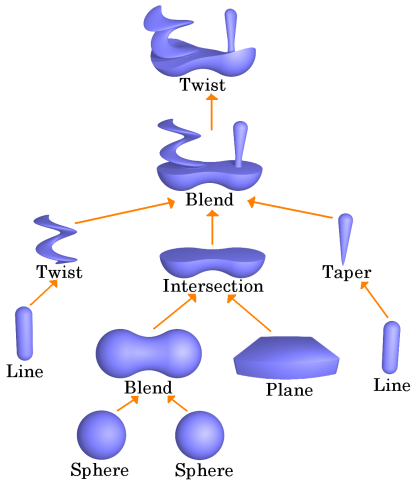


FIGURE 3.43 – Le *blobtree*. Tiré de [WGG99].

On peut considérer les connexions des blocs comme une forme discrète de *blobbies*. Par rapport à ceux-ci, notre système de connexion est moins versatile et souple parce que les *blobbies* peuvent se connecter n'importe où, peu importe la position des primitives de base (sphère, squelette, etc.), contrairement aux blocs, où une connexion est établie seulement de sous-face à sous-face. Cependant, avec les blocs, nous avons beaucoup de contrôle sur l'aspect final de la surface et on peut aisément ajouter des arêtes vives. Notre subdivision de surface conduit à une meilleure tessellation et la surface possède une bonne paramétrisation, un aspect manquant cruellement aux *blobbies*.

3.6.2 *ZSpheres*

Les *ZSpheres* [PIX11] et *B-Mesh* [JLW10] (une variante du précédent) décrivent un objet comme une hiérarchie de sphères. Un maillage de contrôle est construit en englobant les sphères, produisant la même topologie que l'arbre. Contrairement aux blocs, les *ZSpheres* ne semblent pas supporter de topologie complexe (genre > 0), mais pourraient sûrement être modifiées pour le faire.

Elles ne conviennent seulement que pour les objets lisses de type organique, et ne peuvent pas modéliser aisément des objets de type architectural et mécanique. Aussi, elles ne peuvent pas modéliser des surfaces complexes contenant des plis, des vallées et des crêtes sans fonction de déplacement. Puisque leur paramétrisation est effectuée automatiquement, basée sur le maillage

de contrôle, il pourrait s'avérer difficile de les utiliser dans un contexte procédural.

Il est intéressant de noter que l'approche des blocs peut émuler les résultats produits par les *ZSpheres* en utilisant une description d'arbre analogue.

3.7 Conclusion

Nous avons présenté une primitive de bloc simple pour modéliser aisément des objets dans des contextes procédural et interactif. La modélisation par blocs possède des caractéristiques importantes et souhaitables :

- spécification simple de la topologie avec des connexions,
- une définition volumique valide,
- un bon contrôle de la surface à l'aide des sommets des blocs et de fonctions (ou textures) de déplacement,
- un maillage de surface adaptatif.

La représentation par blocs est relativement compacte, vu le nombre final de triangles pouvant être générés. Dans le cas des édifices illustrés aux figures 3.40 à 3.42, principalement constitués de surfaces planes, un bloc pour engendrer entre 37 et 62 triangles. Pour l'arbre de la figure 3.36 possédant une surface lisse, un bloc génère en moyenne près de 860 triangles. Bien que cela soit clairement lié à la métrique de subdivision, nous avons été prudents en ce qui concerne la qualité visuelle obtenue.

On peut considérer que notre système de modélisation représente un bon pas dans la direction pour la définition d'une primitive simple et pourtant puissante pour la modélisation.

3.8 Travaux futurs

Plusieurs avenues intéressantes pourraient et devraient être explorées. Pour les connexions, des poids pourraient être ajoutés, la subdivision des faces en sous-faces pourrait être effectuée automatiquement, et un autre type de connexion pourrait permettre de remplir l'espace entre deux blocs à la place de fusionner leurs sommets. Dans ce dernier cas, cette solution pourrait être généralisée pour permettre de connecter plus de deux blocs à la fois. Les T-Splines [SZBN03]

pourraient possiblement remplacer le maillage de contrôle de Catmull-Clark et ainsi réduire la tessellation et la distorsion de la paramétrisation dans quelques cas.

Comparativement aux *polycube maps* [THCM04, XGH+11], un modèle par blocs avec ses configurations plus flexibles de sommets, devrait pouvoir représenter un objet avec moins de cubes (blocs), tout en se rapprochant davantage de sa forme. Ceci offrirait donc une meilleure compression de la texture de déplacement associée.

Dans ce chapitre, un algorithme de tessellation des blocs a été présenté, mais puisque les blocs sont des primitives de plus haut niveau, ils peuvent être convertis en différents formats. Les convertir en voxels avec un algorithme basé sur les travaux de Lai et Chang [LC06a] permettrait des opérations de CSG plus rapides et robustes.

Chapitre 4

Modélisation par composants

Au chapitre précédent, une méthode de modélisation géométrique et procédurale a été introduite en détail. Contrairement à certaines descriptions exclusivement surfaciques ou volumiques, elle permet une création simplifiée. Toutefois, une telle technique demeure encore insuffisante pour notre objectif initial : la définition efficace d'objets et de scènes complexes tels un édifice, un arbre, une forêt, voire même une ville.

Pour atteindre ce niveau de complexité, il nous faut une méthode de plus haut niveau qu'une simple description de surface ou de volume, bien que cette dernière soit accompagnée d'un langage. On entend ici par plus haut niveau, une méthode qui n'exige pas d'avoir à spécifier tous les moindres détails de la géométrie, mais qui au contraire permet d'automatiser une bonne partie du travail.

Il existe quelques techniques pouvant servir à cette fin. Il y a tout d'abord des techniques de modélisation par optimisation, mais celles-ci sont plutôt rares et restreintes à des cas spécifiques tels la modélisation de plans d'étage et le placement de meubles. Il y a aussi des langages tels que GML [Hav05], l'instanciation géométrique procédurale de Hart ([EMP+02], chapitre 11), le langage de description par couche de Cutler *et al.* [CDM+02], ou encore MEL (*Maya Embedded Language*). Cependant, de tels langages demeurent de trop bas niveau et ne simplifient pas suffisamment la tâche de création. Ils ne font, en fait, que donner accès à des primitives de modélisation de bas niveau dans le contexte d'un langage. La technique de modélisation de haut niveau la plus répandue est l'utilisation de grammaires. Bien qu'elle soit utilisée avec succès pour la génération d'arbres et de forêts, d'édifices (spécialement les extérieurs), de réseaux routiers, et encore de plusieurs autres types de géométries, elle demeure néanmoins limitée

dans son utilisation, et ce, non pas par son manque de flexibilité descriptive, mais plutôt par sa difficulté d'expression. Il peut être en effet très difficile de décrire un modèle spécifique à l'aide d'une grammaire. Une grammaire excelle lorsqu'on désire construire des objets possédant des motifs répétitifs et récursifs offrant des variations. On y retrouve par exemple les arbres avec leurs branches, et les façades d'édifices avec leurs fenêtres, portes et balcons. Mais lorsqu'il n'y a aucun motif évident et simple à décrire, les grammaires deviennent dès lors beaucoup moins effectives. C'est entre autres pour cette raison que, bien que les grammaires aient été introduites en infographie dans le contexte de la modélisation d'extérieurs d'édifices il y a plus de 10 ans, elles n'ont pas encore remporté de succès pour la description complète d'intérieurs et d'extérieurs d'édifices. En effet, on commence à peine à en voir des exemples [HHKF10]. Et dans ce dernier cas, il ne s'agit pas de grammaires au sens strict puisqu'elles sont exprimées dans un langage impératif (GML) et de nombreuses limitations s'y rattachent. Entre autres, en effectuant strictement la subdivision spatiale à l'aide d'opérations de tranchage et de partitionnement sur des espaces rectangulaires, les résultats sont difficilement généralisables et extensibles.

Inspirée des techniques de modélisation de textures, la modélisation géométrique par exemples semble très prometteuse. Entre autres, les travaux réalisés par Merrell et Manocha [Mer07, MM09] conduisent à des résultats intéressants. Dans un même ordre d'idées, il y a la modélisation inverse. Ainsi, Bokeloh *et al.* [BWS10] génèrent une grammaire à partir d'un modèle de base. Ces deux techniques similaires, ont toutefois quelques limitations importantes. Elles dépendent d'un ou de plusieurs modèles de base (et ayant possiblement certaines restrictions) et il peut être difficile de contrôler les résultats obtenus.

En résumé, il existe quelques techniques de modélisation de haut niveau, mais aucune d'entre elles ne fournit la flexibilité désirée tant sur le type de modèles pouvant être générés que sur l'apparence finale du modèle. Prenons l'exemple de la modélisation complète d'un édifice (extérieur, intérieur, ameublement, décoration, etc.). Il existe très peu, pour ne pas dire aucune solution à ce problème. La forme de l'édifice ainsi que ses façades peuvent être décrites adéquatement par grammaires, alors que des techniques d'optimisations semblent être mieux adaptées pour générer l'intérieur. Toutefois dans ce dernier cas, le contrôle est très restreint.

Ce que l'on recherche donc, est une technique de modélisation efficace et flexible, efficace en terme de productivité, qui nous permet de construire des modèles complexes sans avoir à spécifier tous les moindres détails du modèle, et de produire plusieurs variations. La technique doit être flexible, en ne limitant pas le type de géométrie que l'on peut produire et surtout en

nous offrant le contrôle là où c'est nécessaire pour générer exactement le modèle désiré.

La prochaine section décrira sommairement la technique proposée à l'aide d'un exemple simple. Les sections subséquentes entreront dans les détails, allant des structures de données jusqu'à la création de la géométrie. Finalement, une discussion sur la provenance de notre technique et une comparaison avec les grammaires viendra clore le chapitre.

4.1 Sommaire

Dans notre système, un objet procédural (par exemple un édifice) est construit en exécutant un programme composé d'une série de déclarations (voir le listage 4.1). Chaque déclaration, exécutée en séquence, opère sur un ensemble de composants pour modifier leurs attributs ou pour créer de nouveaux composants. Un composant représente une forme positionnée dans l'espace et à laquelle est associée un nombre arbitraire d'attributs (système ou usager). Nous utilisons des requêtes pour restreindre l'ensemble des composants affectés par chaque déclaration. Ces requêtes retrouvent les composants partageant un ensemble commun d'attributs. Comme les composants ne sont jamais remplacés, nous avons accès à tous les composants créés précédemment (contrairement aux grammaires, lesquelles ne travaillent exclusivement que sur les symboles courants).

Les opérations contenues dans les déclarations s'appliquent soit à chaque composant retourné par la requête, soit à l'ensemble des composants. Ceci ouvre la porte à l'utilisation d'opérations binaires et multiples telles que les opérations booléennes. Il est intéressant de noter que ce type d'opérations est difficile à inclure à l'intérieur d'une grammaire puisque cette dernière ne peut combiner plusieurs symboles pour les remplacer par un nouveau symbole. Gervautz et Traxler [GT96] utilisent une grammaire pour créer un arbre de CSG, toutefois sans remplacer de multiples symboles, limitant ainsi la portée des résultats. Lors de la création de nouveaux composants, un arbre est formé, reliant les composants source à leurs enfants (voir figure 4.1).

Non seulement des composants peuvent être créés ou modifiés, ils peuvent aussi être connectés ensemble à l'aide de régions. Une région est un type d'attribut particulier d'un composant définissant des contraintes de positionnement et d'orientation sur la façon dont deux composants peuvent se connecter. Ceci peut être conçu comme une technique analogue de la modélisation par assemblage [SR93] utilisée en CAD.

Notre système de modélisation procédurale unifie dans le même système, l'équivalent des grammaires, la modélisation par assemblage avec les opérations de connexions, des opérations

génériques de modélisation (CSG, extrusion, etc.), et des techniques d'optimisation. Cette combinaison de techniques permet de choisir les meilleures opérations pour différentes parties du modèle généré. Par exemple, pour la construction d'édifices, nous utilisons des opérations de séparation pour les façades, de connexion pour placer les décorations (portes, fenêtres, meubles, luminaires, etc.), et des opérations de CSG pour partitionner l'espace intérieur. Des opérations d'optimisation pourraient aussi être utilisées pour un design d'intérieur ou un ameublement plus automatisé.

Le listage 4.1 donne un exemple simple d'un programme générant quelques pièces d'un édifice montré à la figure 4.1, où est aussi présenté l'ensemble (sous la forme d'un arbre) de ses composants. Ce programme est formé d'une dizaine de déclarations dont certaines n'utilisent pas de requêtes (lignes 2 et 17) et une est imbriquée (ligne 32). Des opérations de séparation (*split*), d'extrusion (*extrude*), booléennes (*subtract*) et de connexion (*connect*) créent de nouveaux composants insérés dans une structure arborescente, structure qui est transformée en géométrie à l'aide d'une opération de conversion de géométrie (ligne 49) et d'une instanciation d'un modèle prédéfini (ligne 44). Dans le cas de l'instanciation, le composant vient avec toute sa géométrie attachée à lui. Cette géométrie utilise une opération de composition (voir section 3.2.1) pour pouvoir soustraire un trou, dans un premier temps, et insérer un contour et sa porte, par la suite. Un exemple semblable au fichier *porte01* est donné par le listage 3.4, mais dans le cas similaire d'une fenêtre.

4.2 Structures

Quatre structures principales sont nécessaires à l'exécution de notre système procédural. Les composants et les contraintes sont définis au niveau d'un programme, alors que les frontières et les régions peuvent être considérées comme des attributs d'un composant.

4.2.1 Composant

Le composant est la structure clef du système. Il est formé d'un ensemble d'attributs géométriques (montrés à la figure 4.2) tels une frontière, une boîte englobante, une liste de régions ainsi que d'attributs intangibles tels une liste d'étiquettes et des attributs usager. Le listage 4.2 présente sa définition en pseudocode.

Un composant définit une zone de l'espace à l'aide d'une boîte englobante orientée. Cette zone sert d'espace local lors de l'application de certaines opérations. Par exemple, une opération

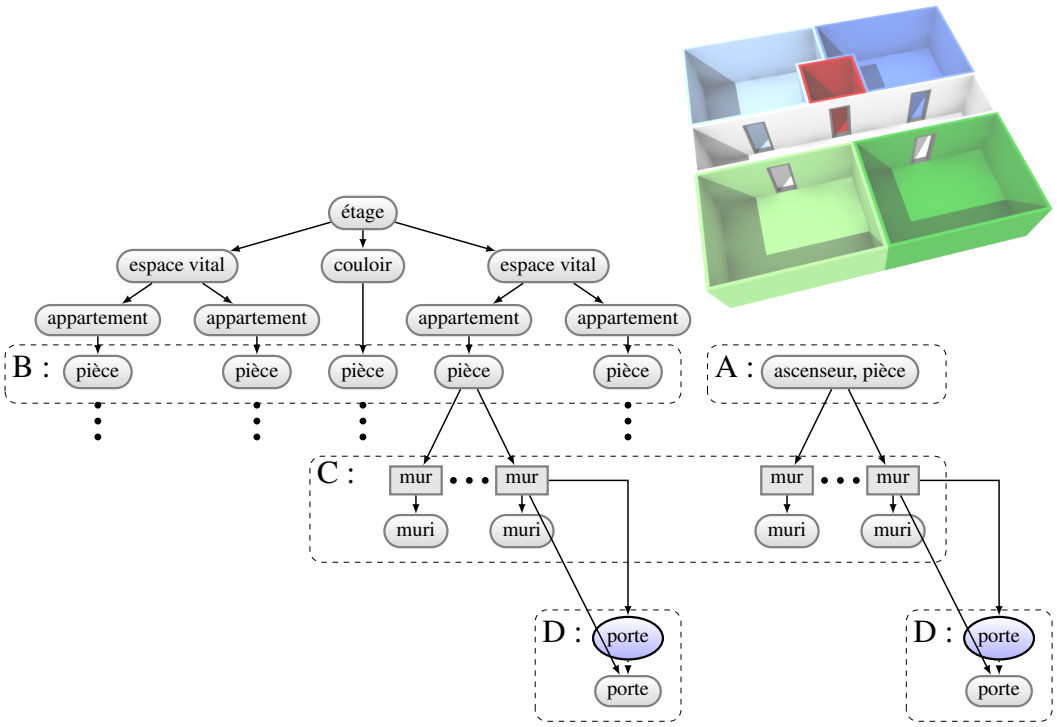


FIGURE 4.1 – Arbre de composants associé au plan simple (coin supérieur droit) de quatre pièces, un couloir et une cage d'ascenseur. Une boîte rectangulaire représente un composant 2D, une boîte aux coins arrondis représente un composant 3D, et une ellipse bleuâtre représente une région. Les points de suspension indiquent la répétition de la structure. Les boîtes étiquetées (A à D) et en pointillé réfèrent aux modifications de l'arbre de composants après avoir exécuté une portion du code du listage 4.1.

de partitionnement ou de tranchage (voir section 4.3.3) s'exécute le long d'un des trois axes de cet espace. L'espace occupé par le composant peut être précisé à l'aide d'une frontière 2D ou 3D consistant dans le cas 2D en la définition d'un polygone et dans le cas 3D, d'un polyèdre.

Un composant possède un nombre arbitraire de composants enfant. Cette organisation de composants forme une arborescence que l'on peut voir dans un exemple à la figure 4.1. Cette hiérarchie est créée lors de l'application d'opérations sur un composant. Par exemple, l'opération de partitionnement (*split*) à la ligne 6 du listage 4.1 génère trois composants enfant. Cet arbre subdivise l'espace hiérarchiquement. Il est important de noter que cette partition de l'espace n'est pas unique, c'est-à-dire que les différentes zones peuvent se chevaucher. Aussi, plusieurs arbres peuvent coexister dans un même programme.

Un ensemble arbitraire d'attributs peut enrichir la définition d'un composant, permettant à l'usager de spécifier diverses méta-informations reliées à ce composant. Par exemple, lors de la construction d'un édifice, des attributs spécifiant le numéro de l'étage, l'épaisseur des murs ou des planchers, le type de matériau, etc., peuvent être assignés aux composants. Ces attributs

```
// Composant principal.
component( label="étage", size={10, 2.5, 10} )

// Création des appartements.
for c in query( "étage" ) do
   split( c, "Z", { label="espace de vie", rel=1 },
                  { label="couloir",       abs=2 },
                  { label="espace de vie", rel=1 } )
end

for c in query( "espace de vie" ) do
   split( c, "X", { label="appartement", rel=1 },
                  { label="appartement", rel=1 } )
end

// Création de la cage d'ascenseur (A).
component(
   label   ={"ascenseur", "pièce"},
   size    ={2, 2.5, 2},
   position={4, 0, 2}
)

// Création des pièces découpées par la cage d'ascenseur (B).
for c in query( "appartement" or "couloir" ) do
   subtract( c, query( "ascenseur" ), { label="pièce" } )
end

// Extrusion des murs appartenant aux pièces, avec un attribut de couleur (C).
var i = 0
for c in query( "pièces" ) do
   i = i + 1
   for f in fquery( c, "SIDE" or "BOTTOM" ) do
      component( c, label="mur", boundary=f )
   end
   extrude( query( c, "mur" ), -0.05, { label="muri", couleur=i } )
end

// Création des portes à l'aide des régions (D).
for c in query( "mur" and not parent("couloir") and occlusion("couloir") > 0 ) do
   region( c, label="porte" )
end

for r in rquery( "porte" ) do
   connect( componentFromFile( "porte01" ), r, {0.5,0,0}, {0,0.05,-0.05} )
end

// Création de la géométrie.
for c in query( "muri" ) do
   solidGeometry( c, c.couleur )
end
```

LISTAGE 4.1 – Pseudocode pour générer l'exemple simple de la figure 4.1.

```
classe Composant
{
    // Définition de l'espace.
    Référentiel          systèmeDeCoodonnées;
    Vec3                 taille;
    Frontière            frontière;

    // Définition de la hiérarchie.
    Composant*           parent;
    Liste<Composant*>    enfants;

    // Définition des connexions.
    Référentiel          connecteur;
    Liste<Région*>       région;

    // Autres attributs.
    Liste<Texte>         étiquettes;
    Table                attributs;
}
```

LISTAGE 4.2 – Pseudocode définissant la structure d'un composant.

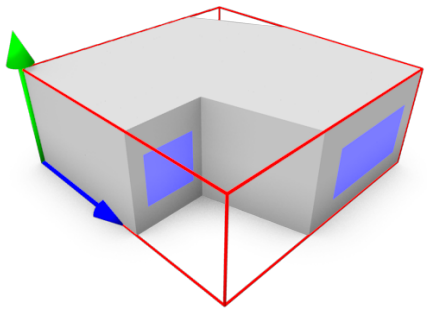


FIGURE 4.2 – Attributs géométriques d'un composant. La frontière du composant est représentée par le polyèdre de couleur grise alors que ses régions sont en bleu et sa boîte englobante en fil de fer rouge. On aperçoit aussi à la gauche deux des axes (Y et Z) du référentiel du composant.

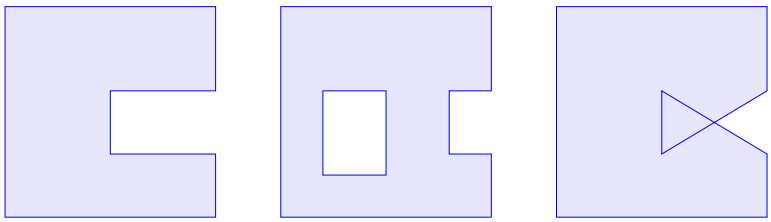
pourront servir soit lors de la spécification d'une requête pour sélectionner certains composants, soit pour être passés comme paramètres à des opérations subséquentes. Les composants enfant héritent des attributs de leur parent. Ainsi, à la figure 4.1 (et dans le listage 4.1), si on assigne un numéro d'étage au composant affublé de l'étiquette "étage", alors tous ses composants enfant comme par exemple les composants "pièce", posséderaient cet attribut.

Une liste d'étiquettes qualifie le type de composant. Ces étiquettes servent exclusivement lors des requêtes pour sélectionner des composants spécifiques. Contrairement aux attributs usager, les étiquettes ne sont pas héritables puisqu'il serait inconvénient de sélectionner un sous-arbre complet au lieu d'un composant précis lors d'une requête. Néanmoins, on utilise souvent de multiples étiquettes pour raffiner sa définition. C'est ainsi qu'un composant peut être une "pièce", une "chambre" et une "chambre des maîtres". Les étiquettes et la frontière sont définies à la création du composant et ne peuvent pas être altérées, alors que les attributs usager peuvent changer en tout temps. La raison est premièrement pour éviter les confusions et deuxièmement pour des fins d'optimisation. En effet, la définition de la frontière des composants enfant est dans bien des cas (selon l'opération de création utilisée) encodée comme une sous-partie de la frontière parente.

Pour permettre la connexion de composants entre eux, des régions et un connecteur entrent dans la définition d'un composant. Un composant possède un seul connecteur et un nombre arbitraire de régions. Un connecteur est défini par un référentiel (un système de coordonnées positionné dans l'espace) et indique le point de connexion et son orientation, alors qu'une région spécifie les endroits où des composants peuvent se connecter (voir les sections 4.2.3 et 4.3.3 pour plus d'informations).

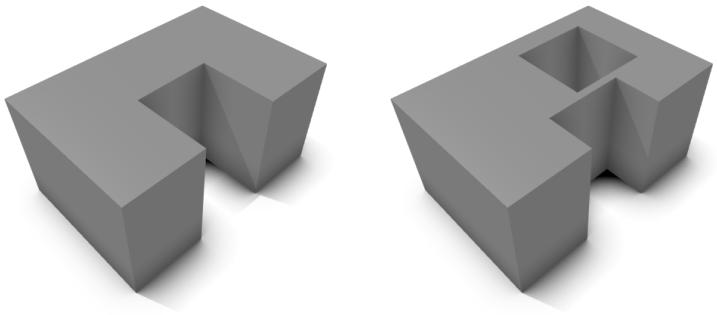
4.2.2 Frontière

La frontière est la définition géométrique du composant. Elle peut être 2D, définie par un polygone, ou elle peut être 3D, définie par un polyèdre. En 2D, le polygone doit être simple, c'est-à-dire formé d'un ensemble d'arêtes qui ne se croisent pas. Il peut toutefois contenir des trous. Dans ce dernier cas le polygone est dit faiblement simple (voir figures 4.3 et 4.4). En 3D, les seules restrictions sont que les faces du polyèdre ne s'intersectent pas et que chacune de ses faces soit un polygone valide (simple ou faiblement simple). Ces restrictions permettent de définir clairement l'intérieur de la frontière et simplifient l'exécution des opérations pouvant l'affecter.



(a) (b) (c)

FIGURE 4.3 – Frontières 2D : (a) simple, (b) faiblement simple et (c) non valide.



(a) (b)

FIGURE 4.4 – Frontières 3D : (a) simple et (b) faiblement simple.

Une étiquette peut être assignée à chacune des faces d'une frontière. Cette étiquette sert principalement à identifier une ou plusieurs faces précises lors d'une requête. Elle permet aussi de contrôler le résultat de certaines opérations. Entre autres, lors des opérations booléennes, deux faces coplanaires sont fusionnées en une seule, seulement si les deux faces possèdent la même étiquette, tel que montré à la figure 4.5.

Lors de l'application d'une opération sur un ou plusieurs composants, de nouveaux composants sont créés ainsi que de nouvelles frontières les définissant. Les étiquettes assignées aux faces des frontières des composants source se voient propagées aux faces des nouvelles frontières (voir figure 4.6).

4.2.3 Région

Une région est une forme sémantique significative (polygone ou polyèdre) appartenant à un composant, positionnée relativement à ce dernier et définissant un ensemble de positions et d'orientations valides (voir figure 4.8). Dans notre implémentation, l'ensemble des positions

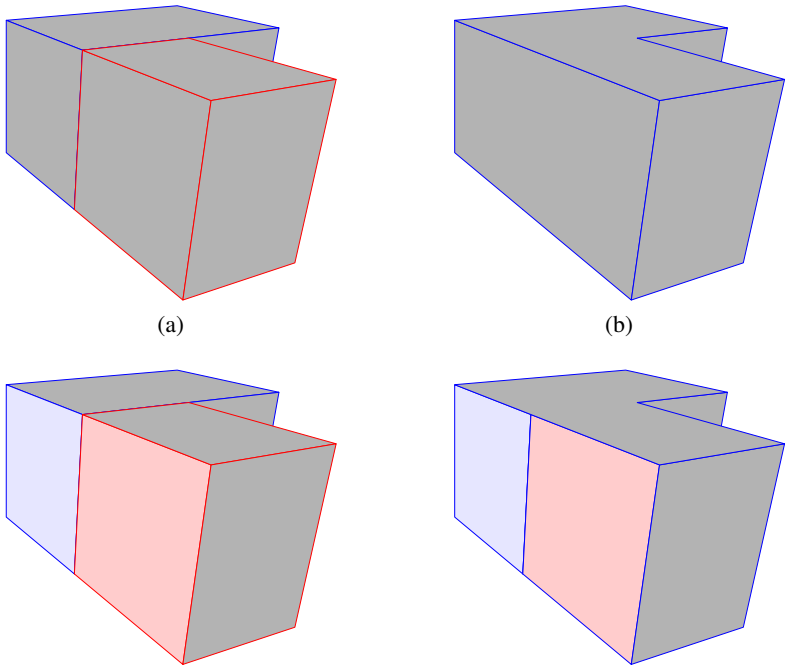


FIGURE 4.5 – Fusion de faces selon leur étiquette lors d'une opération d'union. À gauche (a,c), deux composants, avec contours en bleu et rouge, avant une opération d'union. À droite (b,d), le résultat de l'union. Les différentes couleurs des faces représentent les différentes étiquettes.

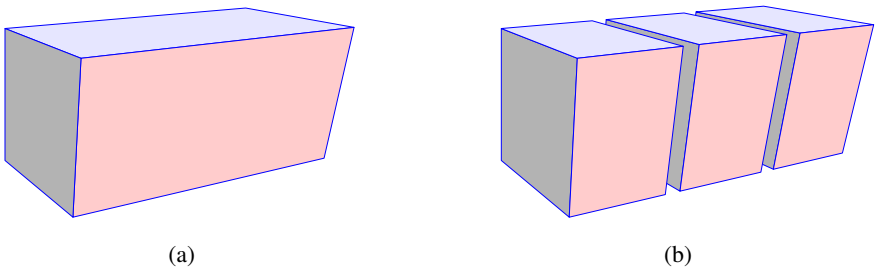


FIGURE 4.6 – Propagation des étiquettes lors d'une opération de tranchage. (a) Composant avant l'opération de tranchage, et (b) composants enfant après l'opération. Les différentes couleurs des faces représentent les différentes étiquettes. À droite (b), l'espace entre les composants est seulement inclus pour montrer les nouvelles faces créées sur le plan de l'axe de tranchage. Une étiquette nulle (de couleur grise) est assignée par défaut à ces nouvelles faces internes.

```
classe Région
{
    // Attributs.
    Texte         étiquette;
    Composant*    parent;

    // Définition de l'espace.
    Référentiel   systèmeDeCoodonnées;
    AABBox        boîte;
    Frontière     domaine;

    // Définition des orientations.
    Quaternion    orientation;
    Vec2          limiteX;
    Vec2          limiteY;
    Vec2          limiteZ;
}
```

LISTAGE 4.3 – Pseudocode définissant la structure d'une région.

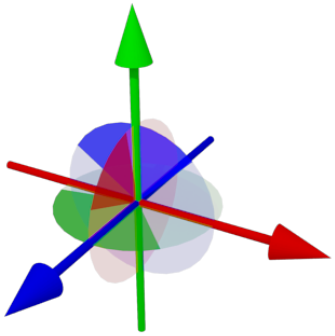


FIGURE 4.7 – Attributs géométriques d'une région définissant les contraintes d'orientation. La rotation permise selon chaque axe est spécifiée par un angle minimum et maximum, et est exprimée ici par un arc de cercle de couleur plus foncée.

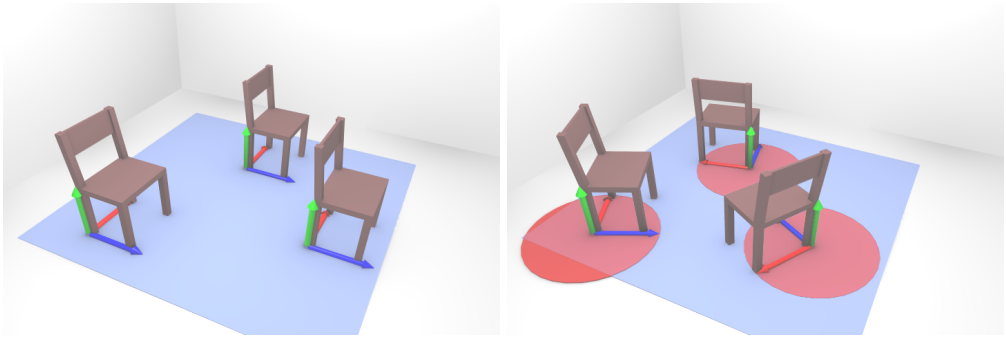


FIGURE 4.8 – À gauche : Une région rectangulaire située sur le composant plancher (en bleu clair) avec trois composants contenant des chaises ; chaque connecteur est dessiné comme un système d'axes. À droite : Une variation du positionnement original à l'aide de rotations (les orientations valides sont représentées par les disques rouges).

valides est défini par une frontière 2D ou 3D, alors que l'ensemble des orientations valides est spécifié à l'aide d'un angle minimum et maximum autour de chacun des axes d'un système de coordonnées. Le listage 4.3 ainsi que la figure 4.7 montrent avec plus de détails cette définition. Bien que nous nous sommes restreints à cette version, on peut aisément imaginer d'autres types de région se servant de courbes ou de surfaces pour définir les restrictions de positionnement. Ceci pourrait être approprié, par exemple, pour des régions localisant les surfaces de connexion de branches sur un tronc d'arbre ou sur une autre branche.

La région sert comme élément de contrainte lors d'opérations de connexion. Dans ce cas, un composant est attaché à un autre composant par une région de ce dernier en alignant son connecteur (système de coordonnées) à une position et une orientation valides définies par la région. Un composant peut contenir un nombre arbitraire de régions, mais possède un seul connecteur. La surface de la région 2D ou le volume (dans le cas 3D non dégénéré) définissent un nombre continu et infini de positions et d'orientations de connecteur. Une étiquette permet de retrouver la région voulue pour effectuer une connexion. Bien que souvent une région est identifiée de la même étiquette que le composant qui y sera connecté, ceci est en aucun cas une restriction et ainsi des composants de différents types peuvent se connecter sur une même région, tels des fenêtres, portes et peintures sur un mur.

4.2.4 Contraintes

Les contraintes, créées à partir de la frontière d'un composant, servent à modifier le résultat de certaines opérations comme on pourra le voir plus en détail dans la prochaine section. Lors de ces opérations, une contrainte peut jouer le rôle d'attracteur ou de répulseur, c'est-à-dire que la géométrie créée s'alignera selon la contrainte ou au contraire tentera de l'éviter. Par exemple, lors de la création de murs, on imposera une contrainte de répulsion à partir des fenêtres pour que les murs ne les intersectent pas. Ces contraintes peuvent s'exprimer sous trois formes, soient planaire, polygonale ou volumique.

Une contrainte planaire est formée à partir d'une face de frontière et est définie par un plan. Elle est principalement utilisée lors des opérations de tranchage pour définir les points de coupe des composants (voir section 4.3.3).

Aussi formée à partir d'une face de frontière, la contrainte polygonale est cependant restreinte à la superficie occupée par cette dernière. Un *offset* peut être spécifié pour élargir ou diminuer d'une taille fixe, le contour du polygone formé.

Une contrainte volumique, quant à elle, est définie par une frontière 3D entière. Comme dans le cas de la contrainte polygonale, un *offset* peut être spécifié pour élargir ou diminuer le volume voulu.

4.3 Exécution

Dans notre système, la construction d'un modèle procédural suit l'exécution d'un programme, qui applique une séquence d'opérations sur un arbre de composants, modifiant ainsi l'arbre. Les opérations sont appliquées sur l'ensemble complet ou un sous-ensemble des composants, sélectionnés avec un mécanisme de requêtes.

4.3.1 Programme

Un programme est formé d'instructions pouvant être regroupées sémantiquement en déclarations, l'équivalent des règles dans une grammaire. Une déclaration a pour but d'appliquer une action précise sur l'arbre des composants et est généralement constituée d'une requête suivie d'une ou plusieurs opérations. Optionnellement, une condition peut contrôler son application. Différents schémas de déclarations sont possibles et les trois principaux sont illustrés dans le listage 4.4. Par exemple, une séquence d'opérations peut être appliquée sur chaque composant retourné par une requête, ou dans un autre cas, une seule opération peut être appliquée sur l'ensemble des composants retournés par une requête. Dans un autre exemple (requêtes imbriquées), une opération peut être appliquée sur chaque composant retourné par une première requête, et aussi, sur un deuxième ensemble de composants retournés par une deuxième requête différente. On peut voir dans ces exemples que l'exécution d'une ou plusieurs opérations sur les composants d'une requête prend la forme d'une boucle utilisant un itérateur.

Bien que le concept de déclaration, réunissant une requête et une ou des opérations en un bloc, est intéressant au point de vue structurel et théorique, en pratique il est quelquefois difficile de bien séparer les différentes instructions en déclarations. C'est le cas lorsque l'on imbrique plusieurs requêtes ensemble pour des raisons de performance et parfois de lisibilité.

Pour notre implémentation, nous utilisons le langage *Lua* tel qu'expliqué à la section 3.3. Puisqu'un programme est écrit sous la forme d'un script, en plus des instructions de requêtes et d'opérations sur les composants, il bénéficie de l'addition de structures de contrôle conditionnelles et itératives. Ces structures sont très utiles pour créer des familles de modèles géométriques. Par exemple, une probabilité d'exécution (à l'aide d'une instruction *if* et d'une valeur aléatoire)

```
1  // Opérations exécutées sur chaque composant d'une requête.
2  for c in query( "étiquette" ) do
3     operation1( c, ... )
4     operation2( c, ... )
5     ...
6     operationN( c, ... )
7  end
8
9  // Opération appliquée sur l'ensemble des composants d'une requête.
10 operation( query( "étiquette" ), ... )
11
12 // Requêtes imbriquées.
13 for c in query( "étiquette1" ) do
14    operation( c, query( "étiquette2" ), ... )
15 end
```

LISTAGE 4.4 – Principaux schémas de déclarations.

peut servir à déterminer si une séquence de déclarations (ou une déclaration unique) est exécutée. Cet accès aux conditions et aux générateurs de nombres aléatoires (valeurs aléatoires, graines aléatoires ou contrôlées) est au coeur du processus de création de variations. Ce concept sera illustré en détail dans le prochain chapitre sur la création procédurale d'édifices. D'ailleurs, un exemple de variation peut être vu à la figure 5.30.

4.3.2 Requêtes

Afin de limiter le champ d'action des opérations, un mécanisme de requêtes passe au travers et analyse tous les composants pour en sélectionner un sous-ensemble respectant des critères arbitrairement complexes. Si des composants source sont passés comme arguments à une requête, au lieu d'effectuer la recherche sur tous les composants, l'analyse sera limitée à ces composants ainsi qu'à leurs enfants. Trois types de critères sont disponibles pour filtrer les requêtes, soient les attributs des composants (i.e., les étiquettes, les identificateurs de face, les attributs usager), la proximité des composants entre eux, et le pourcentage d'occultation. Ces critères peuvent être combinés à l'aide des opérateurs logiques conventionnels (*and*, *or* et *not*) pour former le prédicat de la requête.

Une requête basée sur le critère de proximité trouve tous les composants situés à l'intérieur d'une certaine distance d'un composant donné. Bien entendu, d'autres critères peuvent être combinés à cette requête pour en restreindre la portée. Le facteur d'occultation entre un composant et un groupe de composants est calculé en projetant les frontières du groupe sur le composant, et en évaluant la fraction de la surface du composant couverte par cette projection. Pour obtenir cette fraction, chacune des faces des frontières est convertie sous la forme d'un BSP, puis on applique une opération de soustraction pour chaque face du groupe sur toutes les faces du composant de

base. L'aire des faces restantes est ensuite comparée à l'aire originale pour former le facteur d'occultation. L'occultation peut ainsi être calculée pour une seule face précise de la frontière d'un composant ou pour l'ensemble complet. Dans le listage 4.1 à la ligne 39, l'occultation est utilisée pour identifier les murs (des composants 2D) adjacents à un couloir, dans le but d'y placer des portes.

Jusqu'à maintenant les requêtes ont été expliquées en terme de recherche sur des composants. En fait, trois types d'éléments peuvent être recherchés, le principal étant bien entendu les composants. Outre ce dernier, des requêtes spécialisées permettent de retrouver des régions par leur étiquette (pour les connexions), ainsi que les faces d'un composant (i.e. pour créer des composants 2D pouvant être utilisés par des opérations d'extrusion). Lors de la recherche de faces, en plus des étiquettes assignées par un usager et propagées par les différentes opérations, des étiquettes prédéfinies sont fournies. On retrouve les étiquettes *-X*, *X*, *-Y*, *Y*, *-Z*, *Z* et *SIDE*, cette dernière correspondant à l'ensemble des quatre premières étiquettes (*-X* à *Y*). Chaque face d'une frontière correspond à une seule des étiquettes simples, soit celle dont la normale est la plus similaire à l'axe spécifié par l'étiquette. Les axes peuvent être définis dans différents espaces, soit l'espace global, l'espace local du composant ou un espace quelconque fourni en paramètre.

En résumé, voici la liste de toutes les variations des instructions de requête ainsi que leurs paramètres :

```
query( [composant(s),] prédicat )                ; pour composant
nquery( composant, [distance,] prédicat )        ; par voisinage
rquery( [composant(s),] prédicat )               ; pour région
fquery( [composant(s),] prédicat )               ; pour face
occlusion( [distance,] prédicat )                ; par occultation
```

4.3.3 Opérations

Les opérations décrites dans cette section ne sont que quelques-unes parmi toutes les opérations possibles qu'on pourrait imaginer. Elles ont été cependant suffisantes pour produire tous les résultats présentés dans cette recherche. Une opération travaille sur un ou des éléments, principalement un composant, pour le modifier (altérations, connexions) ou pour en créer de nouveaux (instanciation, partitionnement, tranchage, alternance, etc.).

Altération

```
composant.attribut = valeur
```

L'opération la plus simple consiste à ajouter, modifier ou supprimer un attribut à un composant. Les composants peuvent contenir un nombre arbitraire d'attributs usager génériques. Les types de base supportés par notre système sont les nombres, les chaînes de caractères, les valeurs booléennes, les vecteurs (2D, 3D et 4D) et les tableaux. Pour supprimer un attribut il suffit de lui assigner la valeur *nil*.

Connexion

```
connect( composant, région, [position,] [offset,] [orientation] )
```

Connecter un composant à une région d'un second composant consiste à aligner le connecteur de l'élément source (un système de coordonnées) sur la région de l'autre composant. La transformation rigide résultante appliquée au composant est ensuite propagée aux enfants de la source, aux enfants de ses enfants et ainsi de suite.

Étant donné qu'une région définit généralement un ensemble continu de systèmes de coordonnées possibles, nous permettons à l'usager de spécifier la position et l'orientation exactes au moment de la connexion. Quand aucune position et/ou orientation n'est spécifiée, des valeurs aléatoires sont générées dans l'intervalle défini par la région. Ceci est illustré dans la figure 4.8. La position est spécifiée en espace relatif ($(0, 0, 0)$ pour le coin inférieur et $(1, 1, 1)$ pour le coin supérieur) de la boîte englobante de la région et en y ajoutant optionnellement un *offset* en valeur absolue dans l'espace de la région. L'orientation est spécifiée à l'aide d'un quaternion exprimé en espace local de la région. La position ainsi que l'orientation sont tronquées pour être contenues à l'intérieur des limites de la définition de la région.

Bien que cette opération est beaucoup utilisée dans cette forme simple, elle a un potentiel encore plus grand qui sera exploré dans de futurs travaux. Cette opération pourrait prendre des contraintes en paramètre et utiliser un système d'optimisation pour décider simultanément de l'emplacement de connexions multiples, ce qui pourrait être idéal, par exemple, pour placer des meubles dans une pièce.

```
component{
   [composant,]
   label=,
   [position=,] [orientation=,]
   [connectorPosition=,] [connectorOrientation=,]
   [size=,] [boundary=,] [clip=,]
   [attribut1=,] [attributn=,]
}
```

Instanciation de composant

Puisque notre technique repose sur la génération d'une hiérarchie de composants, de nombreuses façons de les construire sont fournies. La plus directe, l'instanciation, crée un composant explicitement, soit comme composant racine, soit comme enfant d'un parent. Dans ce dernier cas, les composants enfant sont définis dans l'espace du parent et, par défaut, ces composants enfant ont leur frontière découpée selon la frontière de leur parent, pour ne pas s'étendre en dehors de ces derniers. Ce découpage est facultatif mais très utile pour créer un partitionnement de l'espace complexe, par exemple dans le cas d'édifices. Si aucune définition de frontière (*boundary*) n'est passée en paramètre, le composant aura une frontière correspondant à sa boîte englobante (*size*). Dans le cas contraire, la frontière peut être explicitement décrite.

```
component{ composant, boundary=face, label=, ... }
```

Cette définition peut prendre la forme d'une face fournie par une requête de face (*fquery*) à partir de la frontière d'un composant existant. Ceci permet de convertir une frontière 2D en composant 2D pour donner la flexibilité d'appliquer d'autres opérations par la suite à ce nouveau composant.

Des attributs usager quelconques peuvent aussi être définis lors de la création du composant.

Instanciation de région

```
region{
   composant, [face,]
   label=,
   [position=,] [orientation=,]
   [rel=,] [abs=,]
   [limiteX=,] [limiteY=,] [limiteZ=]
}
```

Lors de sa création, la région est assignée à un composant et est positionnée et orientée selon

celui-ci. Alors que les contraintes d'orientation sont exprimées explicitement par les paramètres *limiteX, limiteY et limiteZ*, les contraintes de positionnement sont exprimées relativement au volume ou à une des faces (paramètre *face*) de la frontière du composant. Dans ce cas, l'élément est découpé selon une boîte englobante alignée dans l'espace local du composant dans le cas 3D, et selon l'espace local de la face dans le cas 2D. Les limites de la boîte englobante sont définies par les paramètres *rel* et *abs* selon l'équation suivante :

$$boîte\_min = e\_min + (e\_max - e\_min) \times rel\_min + abs\_min$$

$$boîte\_max = e\_min + (e\_max - e\_min) \times rel\_max + abs\_max$$

où *e_min* et *e_max* représentent les limites de l'élément de base (frontière ou une de ses faces).

Instanciation de contrainte

```
planeConstraint{ composant, [face,] [repulse=] }
faceConstraint{ composant, [face,] [offset=,] [repulse=] }
volumeConstraint{ composant, [offset=,] [repulse=] }
```

Toutes les contraintes sont construites à partir d'un composant, soit dans son entièreté (dans le cas de volume ou de composant 2D pour les contraintes de face ou de plan) ou en spécifiant une face particulière (pour les contraintes 2D). Un *offset* permet d'agrandir ou de rapetisser la taille des contraintes de face et volumiques. Cette option est particulièrement utile lorsque combinée au paramètre de répulsion pour augmenter la distance de déplacement. Par exemple, lorsqu'un mur est déplacé pour ne pas intersecter une fenêtre (le composant de la fenêtre sert à créer la contrainte), on peut contrôler la distance minimale entre le mur et la fenêtre en modifiant le paramètre *offset* de la contrainte.

Une contrainte peut soit attirer (par défaut) ou repousser (*repulse*) des composants lorsqu'elle est utilisée dans une opération (tranchage, partitionnement, alternance).

Tranchage

```
slice( composant, axe, { label=, abs=, [attributs] }, [surplus,] [contraintes] )
```

Le tranchage génère des composants (avec pour étiquette la valeur du paramètre *label*) en découpant un parent avec une taille spécifique unique (*abs*) sur l'un des trois axes principaux de son système de coordonnées de référence. Cette opération est similaire à la fonction de

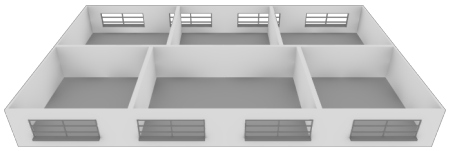
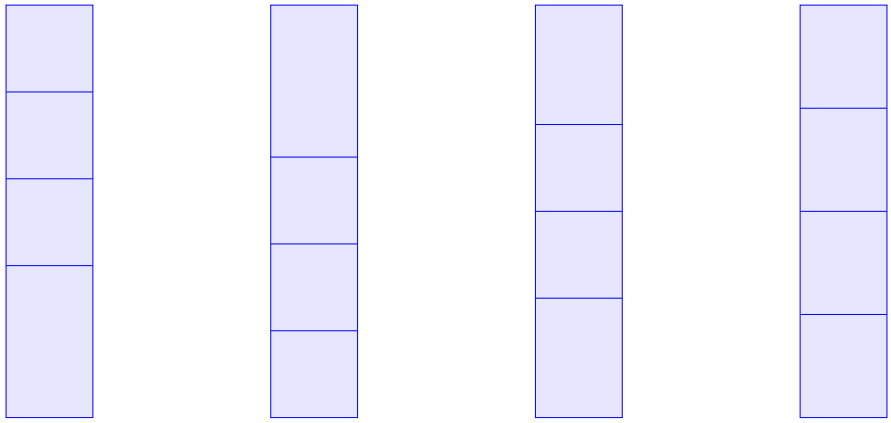


FIGURE 4.9 – Exemple d'utilisation de contraintes lors d'une opération de tranchage avec pour résultat sans (pièces à l'arrière) ou avec (pièces à l'avant) leur application.



(a) Donner au premier (b) Donner au dernier (c) Réparti aux extrémités (d) Réparti à tous

FIGURE 4.10 – Différentes politiques de distribution de la taille restante lors d'une opération de tranchage.

répétition présentée dans les travaux de Müller, Wonka et collègues [WWSR03, MWH+06], avec la distinction que nous avons plusieurs stratégies pour distribuer l'espace restant (donner au premier, donner au dernier, réparti entre les deux, ou réparti entre tous) comme illustré à la figure 4.10.

Une liste de contraintes peut être passée en paramètre pour influencer le positionnement des plans de découpage. Pour calculer l'effet des contraintes, on doit d'abord les projeter sur l'axe de découpage et former des intervalles 1D pour les contraintes de face et de volume, et des points pour les contraintes de plan. L'intervalle d'une contrainte correspond à la valeur minimale et maximale de la projection de tous ses sommets. Puisqu'un plan ne possède pas une taille finie, au lieu de calculer une projection, on calcule plutôt son intersection avec l'axe passant par le centre du composant à découper. Dans ce dernier cas, de nouveaux plans de découpage passant par ces points sont ajoutés à la liste initiale des plans créés automatiquement par l'opération. Par la suite, tous les plans de découpage sont ajustés pour tenir compte des intervalles de contrainte. Les plans contenus dans un intervalle de répulsion sont déplacés vers son extrémité la plus proche, alors que le plan situé le plus près d'une extrémité d'un intervalle d'attraction est positionné dessus. Un exemple utilisant des contraintes de répulsion est illustré à la figure 4.9.

Partitionnement

```
split(
   composant, axe,
   { label=, [attributs,] [rel=,] [abs=,] [mode=] },
   ...,
   { label=, [attributs,] [rel=,] [abs=,] [mode=] },
   [contraintes]
)
```

Le partitionnement (également présent dans les travaux de Müller, Wonka et collègues [WWSR03, MWH+06]) crée des composants enfant selon l'un des trois axes principaux à l'aide d'une liste de tailles (relative, absolue, ou les deux). Les tailles absolues sont d'abord soustraites des dimensions du composant parent, et l'espace restant est divisé entre toutes les tailles relatives pondérées en fonction de leur importance spécifiée. De façon similaire au tranchage, une liste de contraintes peut aussi être donnée en paramètre. Une nouveauté est cependant ajoutée. Chaque composant enfant peut être assigné un mode fixe ou flexible pour indiquer comment il se comporte lors de l'évaluation de contrainte. Lorsqu'un composant est désigné comme fixe et qu'un de ses plans de découpage se déplace pour satisfaire une contrainte, le plan définissant l'extrémité opposée sera déplacé de façon similaire pour garder la taille du composant fixe. Aussi, les contraintes planaires sont ignorées puisque contrairement à l'opération de tranchage, le nombre de composants créés est fixe.

Alternance

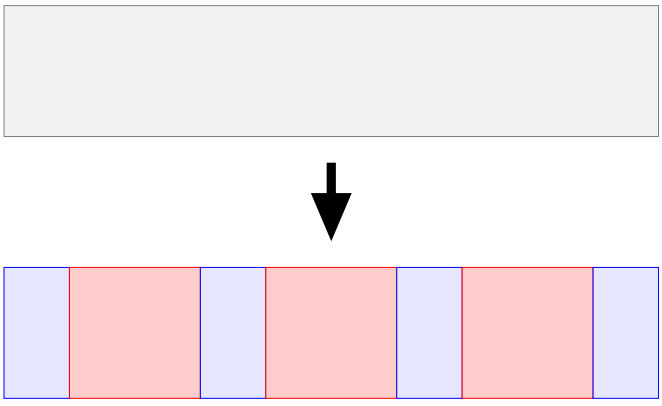
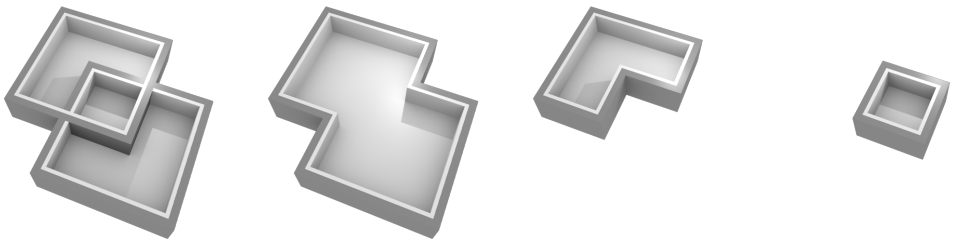


FIGURE 4.11 – Opération d'alternance résultant de l'application de la commande suivante : alternate (parent, "X", { label="A", abs=1 }, { label="B", abs=2 }). Le composant initial parent est en gris, alors que le premier composant (A) est en bleu et que le deuxième (B) est en rouge.

```
alternate(
   composant, axe,
   { label=, abs=, [attributs] },
   { label=, abs=, [attributs] },
   [surplus,] [contraintes]
)
```

L'opération d'alternance est une variante des deux opérations précédentes de partitionnement et de tranchage, où la définition de deux composants est donnée en paramètre. Ces deux types de composant sont alternés sur un axe spécifié jusqu'à atteindre la taille du composant parent (voir figure 4.11). Le dernier composant est toujours du même type que le premier. La taille restante est distribuée également entre tous les composants, tous les composants du premier type, ou tous les composants du second type, selon la spécification fournie (*surplus*).

Opérations booléennes



(a) Sans opération (b) Union (c) Soustraction (d) Intersection

FIGURE 4.12 – Opérations booléennes effectuées entre deux composants se chevauchant. La géométrie présentée ici est créée à partir de l'extrusion interne et externe des faces du composant résultant de l'opération booléenne, sauf dans le premier cas (a) où la géométrie est créée à partir des deux composants de base. On peut ainsi voir que l'utilisation d'opérations booléennes combinées à des opérations d'extrusion permet de bien gérer la géométrie interne et externe, ce qui est la fondation de la création des murs et façades lors de la construction d'édifices. Plus de détails sur ce sujet sont présentés au prochain chapitre.

```
intersect( { composant_1, ..., composant_n }, { label=, [attributs] } )
merge( { composant_1, ..., composant_n }, { label=, [attributs] } )
subtract( composant, { composant_1, ..., composant_n }, { label=, [attributs] } )
subtract( composant, clef, { label=, [attributs] } )
```

Les opérations booléennes (union, intersection et soustraction) peuvent être appliquées à des composants en 2D ou 3D. L'union et l'intersection s'appliquent sur un ensemble de composants, et le résultat est emmagasiné comme un composant racine pour éviter des conflits avec les

attributs de ses parents multiples. La soustraction découpe un composant source à l'aide d'un ensemble de composants, et enregistre le résultat comme un enfant du composant source subissant la soustraction. Une deuxième variante de l'opération de soustraction permet de spécifier la liste de composants à soustraire à l'aide d'une clef (cette clef doit avoir une valeur numérique). Tous les composants intersectant le composant source ayant un attribut du nom de la clef et dont la valeur est supérieure à la valeur du composant source seront enjoints à la liste des composants à soustraire. Cette variation de soustraction par priorité permet de diviser l'espace occupé par des composants en espaces disjoints. Par exemple, pour créer une subdivision de l'espace complexe, on crée un ensemble de composants nommés "espace" se chevauchant selon nos désirs et en assignant à chacun d'eux un attribut usager "priorité" (le nom de l'attribut n'est pas important) indiquant quel composant a la priorité sur l'espace se chevauchant (le composant ayant la valeur la plus élevée l'emporte). Par la suite, l'exécution de la déclaration suivante :

```
for c in query( "espace" ) do
   subtract( c, "priorité", { label="pièce" } )
end
```

va créer un ensemble de composants "pièce" qui ne se chevauchent pas. Ces composants pourront dès lors être repris pour construire des murs et obtenir un partitionnement de l'espace logique.

Puisque la forme des composants est polygonale et généralement simple, un algorithme de BSP [TN87] est appliqué pour obtenir la frontière résultant de l'opération effectuée.

Extrusion

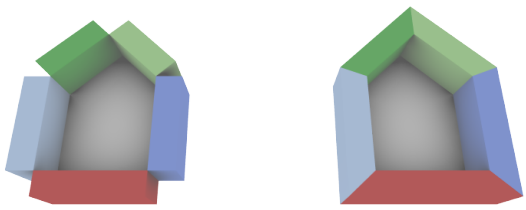


FIGURE 4.13 – Comparaison des deux modes d'extrusion. À gauche, tous les composants sont extrudés séparément. À droite, tous les composants sont extrudés ensemble.

```
extrude( component, direction, { label=, [attributs] } )
extrude( {composant_1, ..., composant_n }, distance, { label=, [attributs] } )
```

L'extrusion d'un seul composant 2D le long d'un vecteur arbitraire génère un composant 3D. Lorsqu'elle est appliquée à un ensemble de composants 2D, chaque arête partagée par deux composants voisins 2D est extrudée dans une direction unique calculée à partir de ces composants 2D, générant ainsi deux nouveaux composants 3D partageant une face, comme illustré dans la figure 4.13. L'arête partagée par deux composants est en fait deux arêtes différentes contenues dans chacune des deux frontières voisines, mais elles se chevauchent. Dans l'extrusion individuelle, chacune de ces arêtes peut être extrudée dans une direction différente, alors que dans l'extrusion d'ensemble, ces deux arêtes sont extrudées dans une même direction unique. Cette opération est hautement utilisée pour créer des murs et des façades cohérents ne se chevauchant pas. Lors de l'extrusion de multiples composants, si un attribut distance (*depth*) est présent dans un composant, il sera utilisé pour son extrusion.

Toiture

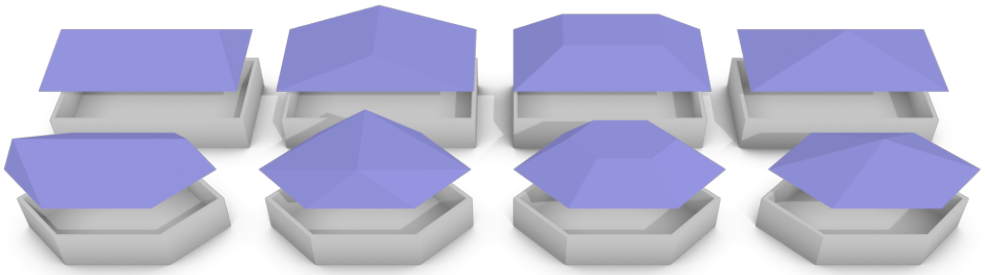


FIGURE 4.14 – Différentes variations de toitures créées par l'opération *roof*.

```
roof( composant, échelle, { label=, [attributs] } )
```

L'opération de toiture est une opération d'extrusion spécialisée pour créer des toits simples (voir figure 4.14). Le polygone de base du composant 2D passé en paramètre est prolongé vers une version modifiée et possiblement dégénérée de ce même polygone, de telle façon à obtenir un pignon. Le polygone extrême est calculé avec un changement d'échelle en relation à son centroïde, suivi d'une translation en direction de sa normale.

Quelques-unes de ces opérations ont été démontrées dans un contexte pratique, entre autres dans l'exemple de la figure 4.1 et du listage 4.1. Pour ceux-là ainsi que toutes les autres, leurs

utilisations seront discutées dans des exemples concrets au prochain chapitre qui s'attardera à la création d'édifices.

4.4 Géométrie

La hiérarchie de composants créée par notre technique est purement abstraite, et représente uniquement le partitionnement spatial d'un objet. Cette hiérarchie sert de contenant pour la véritable géométrie qui peut y être attachée. Tout composant (noeud de l'arbre) peut porter de la géométrie, et pas uniquement les composants feuille de la hiérarchie, mais seulement les composants sélectionnés par l'usager seront convertis en géométrie.

La géométrie peut provenir de la conversion de la frontière d'un composant (par l'opération *solidGeometry* du listage 4.1), ou de toute autre source externe (comme par exemple un logiciel commercial de modélisation). Lors de l'utilisation de l'opération *solidGeometry*, une paramétrisation de surface est générée automatiquement pour la géométrie afin de permettre le placement d'une texture (image ou procédurale).

La géométrie utilisée pour définir la structure d'un édifice (i.e. planchers, murs, plafonds, portes et fenêtres) nécessite le support de la modélisation de solide, puisque les portes et fenêtres creusent des trous dans les murs pour s'insérer (voir la prochain chapitre pour plus de détails). Il est important de noter que cette modélisation de solide n'est pas la même que les opérateurs de CSG décrits précédemment à la section 4.3.3, puisque dans ce cas, les opérations s'appliquent sur la géométrie et non sur les composants.

Par contraste, la géométrie utilisée pour la décoration (meubles, appareils électroménagers, luminaires, etc.) n'implique pas d'opérations de CSG, et peut donc prendre avantage de l'instanciation. On utilise dans notre cas le système de modélisation par blocs présenté au chapitre précédent et dans la publication suivante [LHP11b].

4.5 Provenance

Pour mieux comprendre le système de modélisation par composants et ce qui le distingue des autres systèmes comme les grammaires, il peut être intéressant d'expliquer sa provenance. Dès le début de nos travaux, nous recherchions une façon de décrire procéduralement des édifices. Les grammaires de formes (ou plutôt les grammaires d'ensemble) étaient et sont encore la technique la plus utilisée pour cette fin. Toutefois, à part des cas très restreints, les grammaires n'étaient

pas utilisées pour décrire l'intérieur d'un édifice, mais seulement sa forme et ses façades pour ne créer qu'une enveloppe vide, ce qui est bien pour créer l'apparence d'une ville complexe de l'extérieur, mais problématique lorsque l'on a besoin d'interagir avec les édifices (i.e. les visiter).

On s'est rapidement rendu compte qu'une partie très importante de la modélisation d'un édifice (avec intérieurs) est la subdivision de ses espaces internes, c'est-à-dire sa séparation en pièces, couloirs, cages d'escalier ou d'ascenceur, et autres. Cette séparation est difficile à obtenir et à contrôler avec les grammaires. Il y a bien des opérations de partitionnement (*split*) et de répétition (*repeat*) dans la grammaire de partitionnement (*split grammar*) [WWSR03] qui peuvent servir à cette fin, mais rapidement on en vient à voir leurs limitations. Les vraies grammaires de formes [SG71] (qu'on verra avec plus de détails au prochain chapitre) permettent une plus grande variabilité de description d'espace. Cependant, elles sont difficiles d'implémentation et de contrôle puisqu'elles sont basées sur la reconnaissance de formes émergentes. Les opérations booléennes se sont rapidement dévoilées comme un moyen flexible, lorsque combinées avec des opérations de partitionnement, et simple à contrôler pour subdiviser l'espace. Mais contrairement aux opérations de partitionnement qui sont unaires, les opérations booléennes sont binaires et même parfois variadiques, ce qui cause un problème pour leur inclusion dans une grammaire. En effet, si on prend la définition d'une règle de grammaire :

$$(\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$$

où N est un ensemble fini de symboles non terminaux et Σ est un ensemble fini de symboles terminaux disjoints de $N (\Sigma \cap N = \emptyset)$, on peut voir qu'il peut être difficile de choisir les bons symboles multiples sur lesquels on veut effectuer une opération puisqu'une règle limite la sélection à un symbole et possiblement quelques-uns de ses voisins pour former une chaîne de symboles. Ce problème est encore plus évident pour les grammaires de partitionnement (utilisées pour construire des façades) où un seul symbole à la fois peut être transformé. Ce qu'on aurait besoin plutôt, c'est une règle avec aucune restriction sur le sous-ensemble de symboles que l'on choisi. Alors pourquoi se restreindre à cette définition de règles juste pour appartenir à la description d'une grammaire ?

En généralisant le concept de règles (en fait, la partie gauche d'une règle) on obtient le concept de requêtes, mais cependant, les requêtes ne se combinent pas très bien avec un système récursif de remplacement comme les grammaires. La raison étant qu'une requête ne forme pas vraiment un simple motif de symboles pouvant être retrouvé et remplacé puisque le sous-

ensemble de symboles peut être très complexe. En fait, le système de requête s'intègre beaucoup mieux dans un système itératif comme nous l'avons introduit dans ce chapitre. En permettant d'appliquer individuellement une opération à chaque symbole de la requête, on peut simuler le résultat d'une grammaire. De plus, en permettant d'appliquer une seule opération à l'ensemble de tous les symboles retournés par une requête, on peut maintenant aisément effectuer des opérations quelconques à paramètres multiples et même variadiques comme c'est le cas pour les opérations booléennes.

Une autre limitation des grammaires que nous pouvons lever et qui a pour effet de donner encore plus de flexibilité est la définition de l'ensemble de symboles sur lequel une règle, ou une requête dans notre cas, peut travailler. Une grammaire est un système de remplacement où chaque symbole choisi est remplacé par un ou plusieurs autres symboles. Après cette substitution, les anciens symboles ne sont plus accessibles. Müller *et al.* [MWH$^+$06] avaient noté que le remplacement de symboles à l'aide de règles peut produire un arbre de symboles où chaque symbole remplacé devient le parent de ses remplaçants. Dans leurs travaux, cet arbre de symboles servait à répondre à des opérations d'occultation, mais ne changeait rien quant à l'exécution des règles. Cette persistance des symboles offre toutefois encore plus de souplesse pour la formation de règles (ou de requêtes) qui n'a pas été explorée dans leurs travaux et ceux subséquents.

Dans notre système, nous gardons cet arbre, et l'utilisons lors de la formation de requêtes. De plus, nous nous servons de cette hiérarchie afin de transmettre par héritage des attributs assignés aux symboles (appelés composants dans notre système).

En regardant l'ensemble des changements apportés, on peut considérer le système de composants comme un système près des grammaires et ayant une philosophie similaire, mais en plus flexible. Cette nouvelle souplesse acquise sera mise à bon escient dans le prochain chapitre pour obtenir la création d'édifices complets avec intérieurs et extérieurs.

Chapitre 5

Modélisation procédurale d'édifices

L'objectif qu'on s'était fixé au début de cette recherche était de développer un système permettant la génération procédurale d'objets complexes afin de réduire leur temps de création. Ce gain de temps devrait venir de l'opportunité qu'apporte la modélisation procédurale d'effectuer rapidement des changements importants et principalement de la possibilité de générer une multitude de variantes.

Le chapitre 3 a établi les bases géométriques du système en présentant une nouvelle primitive, le bloc, aidant à créer des modèles géométriques complexes sans avoir à gérer tous les détails topologiques. Cette primitive supporte les opérations booléennes simplement avec une définition volumique de la géométrie et avec un nouvel algorithme assez robuste pour rendre son utilisation pratique dans le contexte procédural.

Enfin, au chapitre 4, un nouveau système de modélisation procédurale de haut niveau a été introduit en détail. Toutefois, l'explication s'est restreinte principalement aux aspects théoriques et descriptifs sans donner beaucoup d'exemples concrets et pratiques. Le but de ce chapitre est exactement de passer de la théorie à la pratique en utilisant ce nouveau système pour créer des édifices. Le choix de se concentrer sur la modélisation d'édifices contrairement à d'autres types de modèles n'est pas sans fondement. À l'inverse de la végétation ou des terrains, par exemple, aucune solution complète n'existe pour les édifices. Dans le cas de la végétation et des terrains, malgré que les recherches se poursuivent, il est aujourd'hui possible de générer complètement des arbres, des forêts et des terrains de tout type procéduralement. La génération d'édifices est un problème plus complexe pour lequel il existe de bonnes solutions pour certains aspects, principalement de création de façades ou de certains types de plans d'étage 2D.

De plus, des gains importants peuvent être obtenus par la "procéduralisation" des édifices puisque leur conception manuelle est complexe et longue. Ce problème est décuplé lorsqu'il est question de modéliser un quartier, ou encore pire, une ville. La possibilité de générer automatiquement des variations d'un édifice modèle prend alors toute sa signification.

La prochaine section fera un survol des techniques courantes pour la modélisation procédurale d'édifices en commençant par une introduction aux grammaires (de formes). Ce sommaire sera suivi d'une description générale des différentes étapes possibles utilisant la modélisation par composants pour créer un édifice complet. Ensuite, une série d'exemples sera présentée pour donner plus de détails sur des variations possibles des différentes étapes. Finalement, un ensemble de résultats complexes expose les capacités de la technique.

5.1 Travaux antérieurs

5.1.1 Grammaire de formes

La modélisation procédurale à l'aide de grammaires est une méthode efficace pour générer des scènes avec de riches détails géométriques. De manière générale, ses règles de production modifient une structure en ajoutant des détails à chaque itération. Plus qu'une simple automatisation du processus de conception, elles permettent l'exploration de designs alternatifs, et même possiblement peuvent suggérer éventuellement des designs innovateurs. Un grand nombre de grammaires de conception ont été introduites, notamment les systèmes-L [PL90], les grammaires de Chomsky [Sip96], les grammaires de graphes [EEKR99], les grammaires de formes [SG71], et les grammaires avec attributs [Knu68].

Les grammaires de formes sont à la base de plusieurs grammaires servant à la création d'édifices, entre autres les grammaires d'ensembles, les grammaires de partitionnement (*split*), les grammaires CGA et les grammaires de murs. Avant d'étudier ces grammaires, une petite introduction de celles-ci est de mise.

Grammaire formelle

Une grammaire formelle permet de définir un langage à l'aide d'un ensemble strict de règles. Il existe deux catégories de grammaires : analytiques et génératives. Les grammaires génératives sont formées d'un ensemble fini de symboles terminaux, d'un ensemble fini de symboles non-terminaux, d'un ou plusieurs symboles de départ et d'un ensemble de règles de production

séparées en membres gauche et droit formés de mots (un ensemble de symboles). L'application d'une règle se fait sur un mot et consiste à remplacer une sous-chaîne du mot contenant le mot du membre gauche de la règle par le mot du membre droit. L'ensemble de tous les mots, ne contenant que des symboles terminaux, et obtenu par dérivation d'une séquence de règles définit le langage associé à une grammaire.

Grammaire de formes

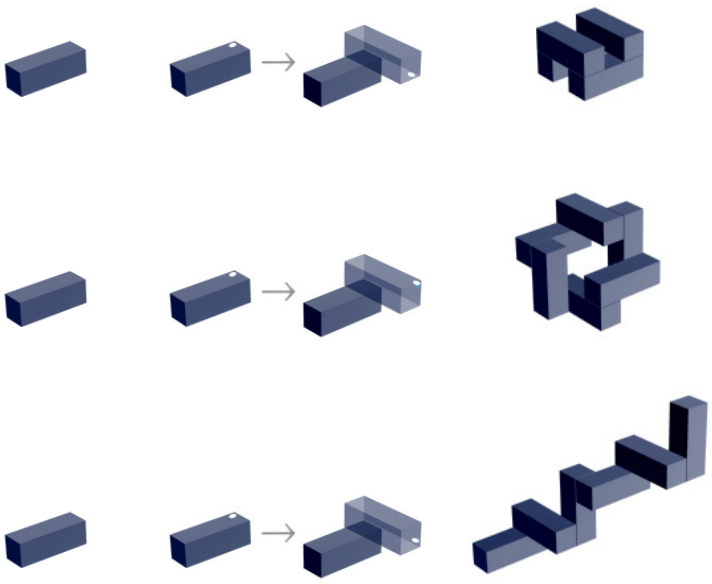
Dérivée des grammaires formelles et introduite dans les années 1970, la grammaire de formes [SG71, Sti75, Gip75] fut et est encore aujourd'hui surtout utilisée dans le domaine architectural pour étudier le style (design). L'idée générale de ce type de grammaire est d'associer aux symboles une représentation géométrique, la forme, qui est définie par un ensemble limité de segments de droites dans l'espace euclidien tri-dimensionnel. Cette grammaire aura donc une ou plusieurs formes initiales, un ensemble de formes et un ensemble de règles qui remplacera, déplacera ou modifiera ces formes géométriques de façon récursive. Cette grammaire définit implicitement une famille de modèles géométriques formés par un enchaînement de règles légales.

Plus formellement, une grammaire de formes est définie comme étant un tuple $G = (N, T, R, I)$ où $N \subseteq U$ est le sous-ensemble des symboles non-terminaux, $T \subseteq U$ est le sous-ensemble des symboles terminaux, $I \subseteq N$ est un sous-ensemble de symboles initiaux (contenant généralement un seul élément) et $R \subseteq U \times U$ est l'ensemble des règles. U est l'ensemble des symboles aussi appelés dans cette grammaire, la forme.

Un exemple de ce type de grammaire introduite par Stiny est la grammaire de Kindergarten [Sti80], qui peut être observée à la figure 5.1. Dans cette figure on peut voir un sous-ensemble des règles proposées, ainsi que les formes obtenues après quelques étapes de dérivations. Les points blancs situés sur la surface des blocs dans les règles définissent l'orientation de la règle. Un ensemble plus complet des formes possibles est illustré à la figure 5.2.

Grammaire d'ensembles

Beaucoup de recherche et d'applications de la grammaire de formes [SW78, Fle87, Cag96, AC98, Dua05] ont été développées au cours des années. La plupart d'entre elles se restreignent à l'application manuelle de leurs règles ou utilisent un sous-ensemble de la grammaire. Le problème est qu'implémenter une grammaire de formes n'est pas une tâche simple. La difficulté



(a) (b) (c)

FIGURE 5.1 – Grammaire de formes de Kindergarten. (a) Forme initiale. (b) Règles. (c) Résultats après quelques itérations de la règle correspondante en (b).

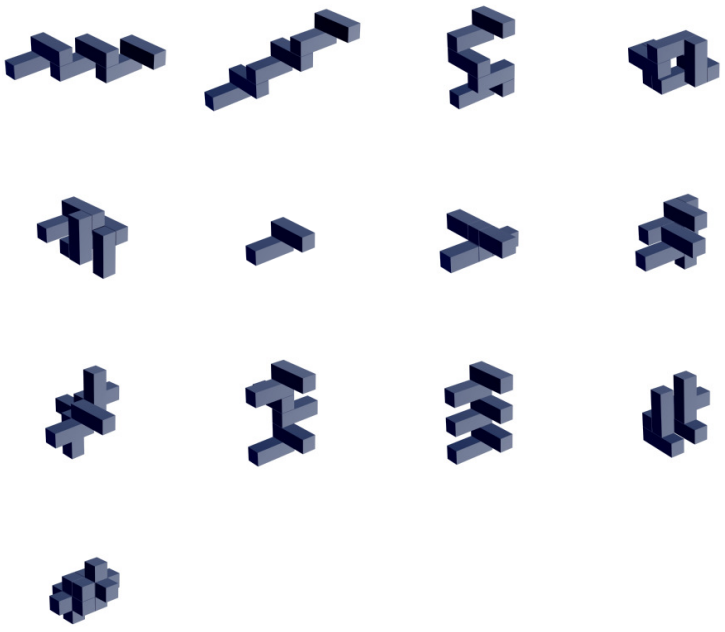


FIGURE 5.2 – Autres résultats de variantes de la grammaire de Kindergarten de la figure 5.1 en appliquant différentes règles.

principale réside dans l'apparition de formes émergentes qui demandent à implémenter la reconnaissance de formes. Un exemple classique de ce problème est illustré à la figure 5.3. Cet exemple composé de deux formes initiales et de deux règles, produit rapidement, après la dérivation d'une règle, une nouvelle forme, un carré, pouvant être à son tour transformée par une règle. En plus d'être difficile à produire, ce genre de grammaire est difficile à prévoir et comprendre, et donc à utiliser.

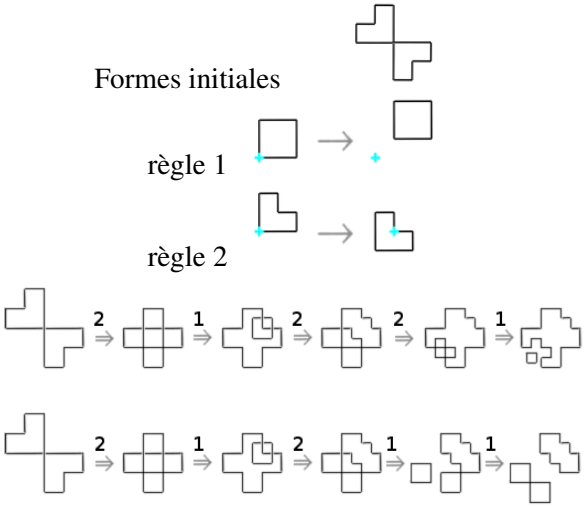


FIGURE 5.3 – Exemple d'une grammaire de formes démontrant l'apparition de formes émergentes.

Pour éviter le problème de formes émergentes, un sous-ensemble de la grammaire de formes peut être employé. Cette grammaire, appelée grammaire d'ensembles [Sti82], considère les formes comme des objets symboliques et ne nécessite par conséquent aucune reconnaissance de formes.

Extensions

De nombreuses extensions ont été apportées aux règles des grammaires afin de faciliter la création de formes complexes et d'augmenter le pouvoir expressif des règles. Parmi les principales extensions conçues on retrouve les suivantes :

conditions : Une condition permet l'application conditionnelle des règles.

paramètres : Des paramètres peuvent être assignés aux symboles, et en collaboration avec des conditions, ils permettent une meilleure diversité dans l'application des règles.

probabilités : Les règles peuvent avoir plusieurs successeurs (membre droit), chacun ayant une probabilité d'application, le tout sommant à un.

sensibilité au contexte : L'application des règles peut être dépendante du contexte spatial et géométrique des formes. Entre autres, la visibilité à partir d'une forme, sa position, ses voisins, etc., peuvent servir à modifier l'application d'une règle.

division : Une règle de division [WWSR03, MWH$^+$06] décompose une forme en plusieurs formes contenues à l'intérieur du volume de la première. Un exemple d'application de ce type de règle est illustré à la figure 5.4. Ce type de règle se révèle particulièrement utile pour créer des façades d'édifices.

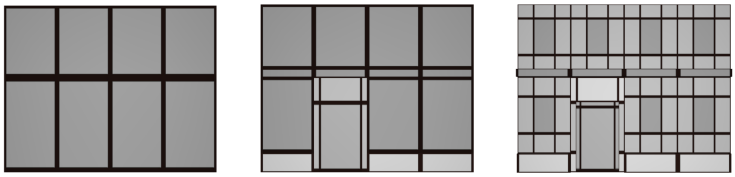


FIGURE 5.4 – Trois étapes de division d'une grammaire de formes. Images tirées de [WWSR03].

Édifice

Comme mentionné précédemment, bien que les grammaires de formes ont servi de base à ce qui allait suivre en infographie, leur complexité et la nécessité, dans bien des cas, de la participation d'un usager pour résoudre la grammaire, entrave le type de modélisation générative efficace requis pour leur application à grande échelle. Néanmoins, un certain nombre de grammaires de formes [KE81, Fle87] ont été dédiés à un type bien particulier d'architectures, bien que très peu sont accompagnées par une implémentation informatique. Dans une exception à cette remarque, Heisserman [Hei94] a développé un formalisme de grammaire de frontières pour solides, ainsi que son implémentation montrant des variations de maisons de style reine Anne, ce formalisme étant basé sur les grammaires de Flemming [Fle87]. Un autre exemple est celui de Duarte [Dua05], qui a implémenté une grammaire pour reproduire des maisons de Siza.

Parish et Müller [PM01] ont initié en grande partie la tendance actuelle sur la génération procédurale d'édifices pour leur application en infographie. Alors que leurs édifices (vus à la figure 5.5) ont des formes simples et leurs façades utilisent seulement un ensemble de textures de base, ils ont réussi à créer des villes complètes et crédibles composées d'édifices divers, construits par grammaires d'ensembles et répartis le long d'un réseau routier généré par système-L.

FIGURE 5.5 – Différentes étapes de la création d'un édifice par grammaire d'ensembles. Tiré des travaux de Parish et Müller [PM01].

FIGURE 5.6 – Quelques façades d'édifices originant de Wonka *et al.* [WWSR03], générés par une grammaire de partitionnement.

L'étape suivante, par Wonka *et al.* [WWSR03], a nettement amélioré les façades pouvant être générées par la technique de Parish et Müller en introduisant les grammaires de partitionnement (*split grammars*). Ils ont créé une base de données d'éléments architecturaux pour les façades, qui peuvent être combinés pour produire des décorations et des styles variés. Quelques exemples de façades sont illustrés à la figure 5.6. Larive et Gaildrat [LG06] (voir figure 5.7) ont introduit une grammaire de murs pour leurs façades, qui s'intègre aux contours de plans d'édifices au niveau du sol, extraits de photos, pour extruder des formes générales d'édifices.

Les formes complexes d'édifices ont été intégrées aux façades lorsque Müller *et al.* [MWH$^+$06] ont introduit les grammaires CGA, une grammaire de formes (ou d'ensemble pour être correct) pour l'architecture, permettant de créer des édifices de haute qualité avec beaucoup de détails. Ils utilisent les grammaires de partitionnement de Wonka *et al.* [WWSR03], mais parmi plusieurs nouvelles contributions, ils ont développé des règles contextuelles aux formes (des segments d'alignement et des requêtes d'occultation) pour adapter les éléments de façades en présence de formes de base de l'édifice s'intersectant. La figure 5.8 montre justement un exemple de ceci. Finalement, Krecklau *et al.* [KPK10] ont par la suite généralisé la définition des symboles non-terminaux de façon orientée-objet, permettant l'ajout facile de nouveaux opérateurs comme les déformations de formes libres.

Pour simplifier la création de règles, Müller *et al.* [MZWG07] ont présenté un système pour

FIGURE 5.7 – Un exemple d'édifice construit par grammaire de murs [LG06].

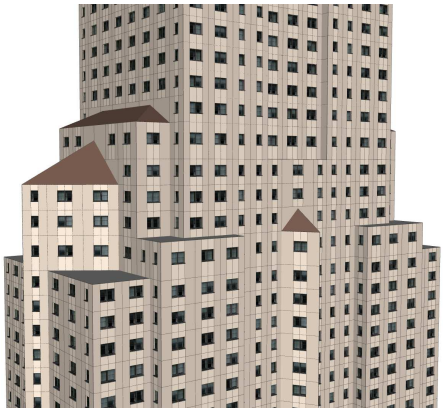


FIGURE 5.8 – Édifice créé à l'aide d'une grammaire CGA [MWH+06] avec requêtes d'occultation pour positionner les fenêtres dans des régions non obstruées. Voir aussi la figure 2.23 pour plus d'exemples.

segmenter automatiquement des photos et les interpréter afin de recréer la géométrie associée à l'aide de règles de partitionnement. Des améliorations supplémentaires sous la forme d'un éditeur visuel interactif de grammaires ont été proposées par Lipp *et al.* [LWW08]. Cet éditeur offre une modélisation plus intuitive et empêche une explosion combinatoire des règles en permettant d'appliquer et d'enregistrer des changements locaux et persistants.

5.1.2 Modélisation à partir de photos

Une toute autre branche de la modélisation procédurale d'édifices consiste en la reconstruction à partir de photos. Dans ce contexte, pour créer un édifice, un modèle réel doit exister, ce qui impose des contraintes importantes. Toutefois, certaines techniques permettent de générer des variations. Ceci est le cas de la modélisation par nombres [BA05a], alliant le traitement d'images à la modélisation procédurale. De nouveaux modèles d'édifices sont produits à partir de photos en appliquant une duplication de blocs de textures, provenant des images source, selon un identificateur (nombre) associé à la nouvelle géométrie d'une région.

Dans un premier temps, un modèle géométrique 3D (figure 5.9a) est reconstruit à partir

des images source. Ce modèle est ensuite subdivisé en blocs selon ses caractéristiques (portes, fenêtres, murs, etc.) et on leur associe des nombres (figure 5.9b). Ceci définit une transformation entre l'espace des images source et l'objet 3D. Un nouveau modèle 3D simplifié est ensuite construit soit manuellement, soit procéduralement (figure 5.9c). Finalement, les blocs de textures sont transférés sur la nouvelle géométrie selon leurs nombres (figure 5.9d).

Bien que l'idée de construction par nombres ait beaucoup de potentiel, cette technique est limitée par plusieurs contraintes. Il y a d'abord toutes les tâches manuelles : la création du modèle 3D de base, et l'extraction et la séparation du modèle en cellules numérotées. Les cellules sont texturées à partir de l'extraction manuelle et donc aucune géométrie complexe n'est créée. Le positionnement des cellules est simpliste, en ce sens qu'il n'y a pas de superpositions et elles sont alignées sur les axes. Finalement, comme toutes techniques à base d'images, un modèle réel doit exister.

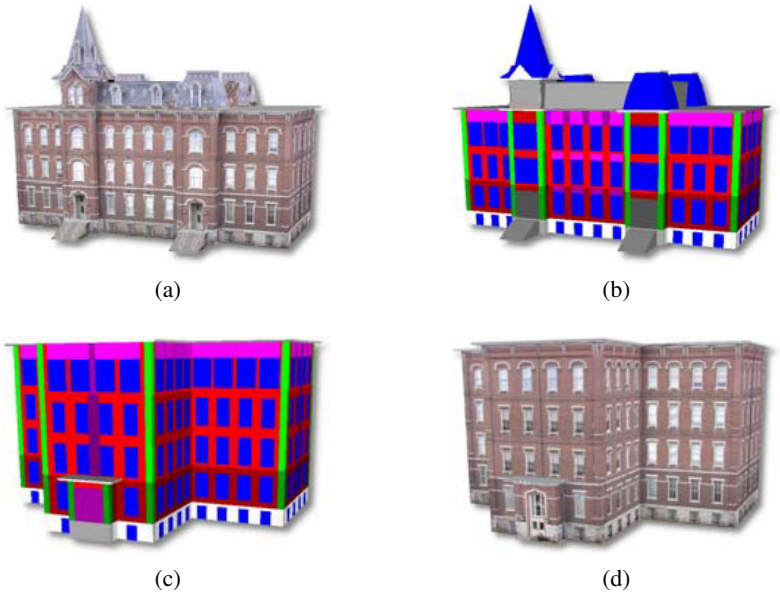


FIGURE 5.9 – Construction par nombres. (a) Modèle initial. (b) Numérotation (couleur) du modèle initial. (c) Numérotation d'un nouveau modèle. (d) Nouveau modèle construit. Images tirées de [BA05a].

Dans des travaux plus récents, Xiao *et al.* [XFT+08] réussissent à extraire un modèle géométrique plus détaillé à partir d'un grand nombre de photos prisent de la rue. Leur approche semi-automatique crée une texture de profondeur pour la façade à partir d'un nuage de points extrait des photos et calculé automatiquement en récupérant la structure par le mouvement. Afin de diminuer le bruit sur les profondeurs, la façade est décomposée en partitions rectangulaires, chacune ayant sa profondeur. Ces partitions sont calculées automatiquement à partir de lignes

retrouvées dans les photos, mais ces partitions peuvent être également ajustées par un usager. Une illustration des résultats obtenus est montrée à la figure 5.10.

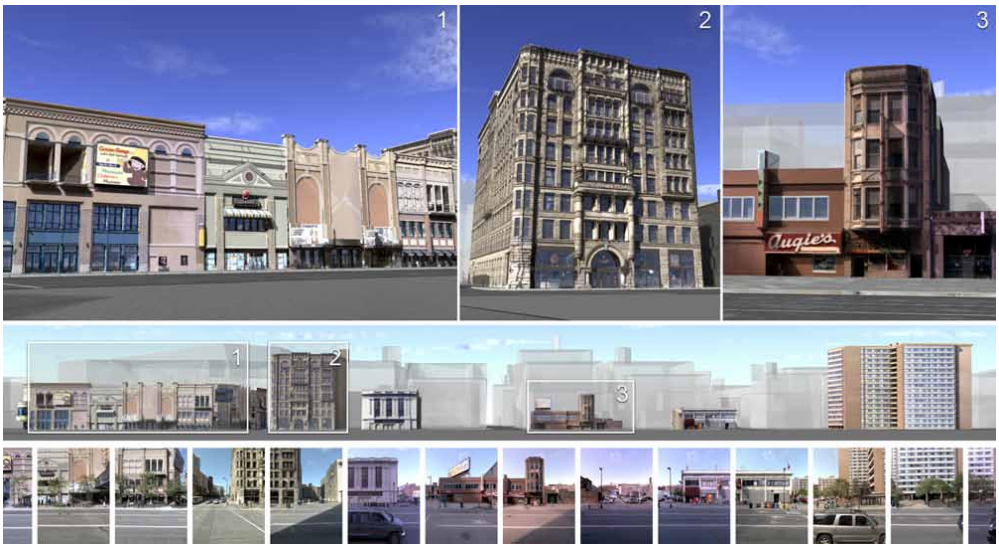


FIGURE 5.10 – Modélisation de quelques édifices à partir de 281 photos [XFT+08] dont quelques-unes sont montrées à la ligne du bas. Le résultat est visible dans la ligne du milieu et de plus près au haut de la figure.

Nan *et al.* [NSZ+10], quant à eux, démontrent qu'il est possible de reconstruire un modèle géométrique malgré l'acquisition (entre autres par technologie LiDAR) d'un nuage de points épars et bruité. Ils accomplissent la tâche à l'aide d'un outil interactif composé d'une primitive de type boîte qui ajuste sa position et sa taille selon les points locaux et les boîtes voisines. Ces boîtes, lorsque bien positionnées, permettent de récupérer une géométrie adaptée.

5.1.3 Optimisation

D'autres techniques reposent sur la résolution de contraintes par optimisation. Pour la plupart, elles peuvent être catégorisées soit en création de plans d'étage, soit en distribution d'ameublement.

Plan d'étage

Harada *et al.* [HWB95] ont présenté une méthode pour la manipulation interactive et sensible au contexte de plans d'étage (voir figure 5.11). Ils n'ont pas travaillé sur un espace réel 3D, ni sur la forme de l'édifice. Cette constatation s'applique en fait à la plupart des techniques d'aménagement de plans d'étage étudiées en architecture, en ingénierie et en infographie. C'est le cas, entre autres, des techniques de Jo et Gero [JG98] qui utilisent un algorithme génétique

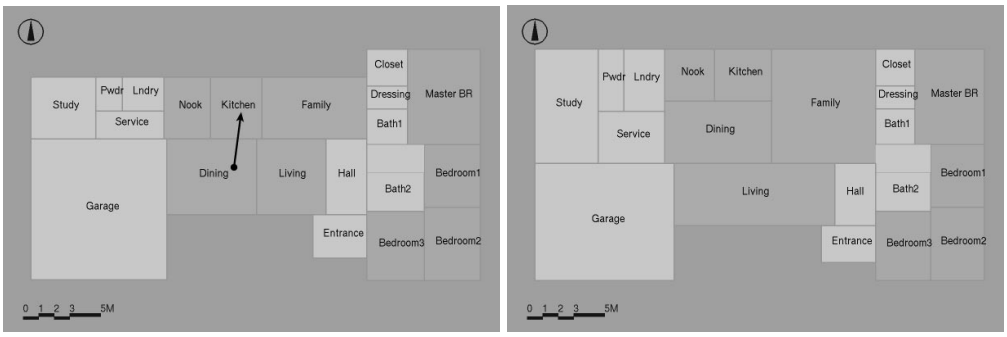


FIGURE 5.11 – Création interactive de plans d'étage [HWB95]. Dans la figure de gauche, une pièce est déplacée et le système reconfigure automatiquement (à droite) l'agencement des pièces en positionnant le salon (*living*) à l'avant.

et de Tutenel *et al.* [TBSdK09] dont leur système par résolution de contraintes démontre son utilisation aussi bien pour l'ameublement que pour la séparation en pièces d'un coutour d'édifice prédéfini.

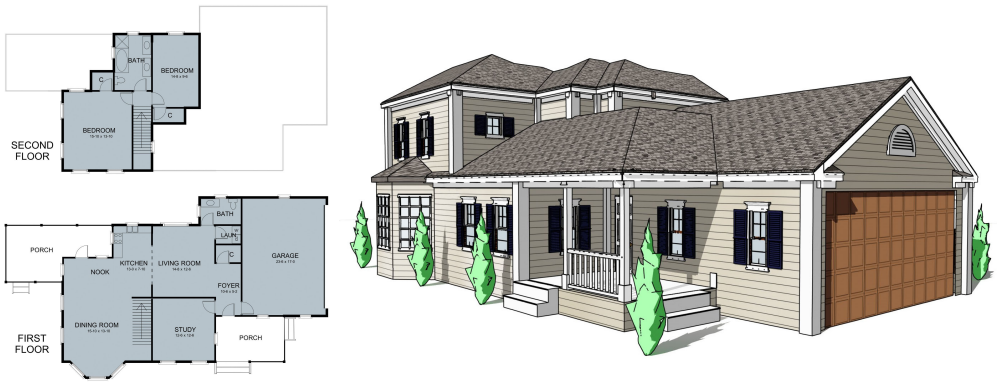


FIGURE 5.12 – À gauche, les plans des premier et second étages générés par optimisation stochastique [MSK10]. À droite, la maison générée automatiquement à partir des plans d'étage.

Récemment, Merrell *et al.* [MSK10] ont décrit une technique à base d'optimisation stochastique, entraînée avec un réseau bayésien sur des données réelles de plans d'étage provenants d'architectes. Une session d'optimisation pour un plan d'étage débute en disposant toutes les pièces requises dans un alignement de grille et en leur donnant toutes la même taille. Puis, à chaque étape du processus, soit une partie d'un mur est déplacée, soit deux pièces sont échangées (en inversant leurs étiquettes). La nouvelle variation est acceptée avec une probabilité qui dépend de son coût déterminé en relation avec des données obtenues par apprentissage. Bien qu'ils génèrent en entier des édifices de type maison de ville de deux étages, leurs travaux se concentrent sur la génération automatique de plans d'étage 2D et très peu, pour ne pas dire aucun détail n'est donné quant à la façon dont le modèle géométrique 3D est créé. Un exemple de plan d'étage

créé par cette technique ainsi que l'édifice correspondant est illustré à la figure 5.12.

Ameublement

FIGURE 5.13 – Disposition de meubles semi-automatique répondant à des directives de design intérieur. Image provenant des travaux de Merrell *et al.* [MSL+11].

FIGURE 5.14 – Deux variations d'ameublement d'une même pièce obtenues par optimisation, tirées de [YYT+11].

En utilisant un algorithme génétique pour résoudre des contraintes de positionnement, Sanchez *et al.* [SLRLG03] créent des ameublements complexes et réalistes.

Merrell *et al.* [MSL+11] ont présenté un système de positionnement interactif, lequel peut donner des suggestions basées sur des contraintes. Entre autres, un usager peut indiquer la position précise de certains meubles ainsi que des contraintes de dégagement autour de ces meubles et le système suggèrera un ensemble de configurations respectant ces contraintes ainsi que des directives de design intérieur. Ces directives sont basées sur des critères séparés en deux catégories : fonctionnelles (dégagement, circulation, relation inter-meuble et conversation) et visuelles (balance, alignement et emphase).

Contrairement au système précédent, celui de Yu *et al.* [YYT+11] est complètement auto-

matique, mais demande une période d'apprentissage pour extraire la relation hiérarchique entre les différents meubles et objets. L'optimisation est calculée par recuit simulé avec un pas de recherche de *Metropolis Hasting*. Voir la figure 5.14 pour un exemple.

5.1.4 Autres

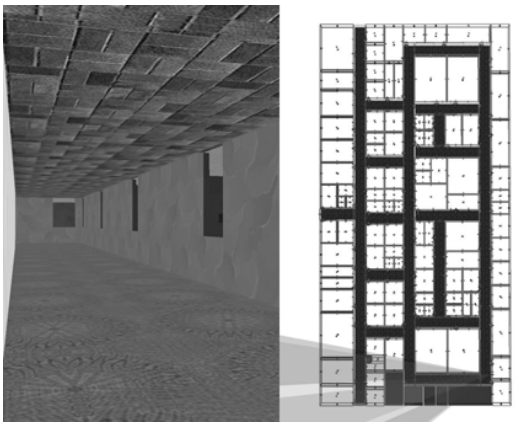


FIGURE 5.15 – Plan d'étage simple généré par la technique de Hahn *et al.* [HBW06], ainsi qu'un point de vue intérieur à gauche.

Hahn *et al.* [HBW06] ont créé des intérieurs simples d'édifices multi-étages pour lesquels la génération peut être faite sur demande dépendamment du point de vue. Ils proposent des étapes récursives de génération par remplacement (similaire à une grammaire) pour créer des cages d'escalier, des couloirs et des pièces. Cependant, l'organisation d'un plan d'étage est restreinte à une subdivision simple alignée sur les axes d'un édifice de forme rectangulaire. Leur technique ne gère pas les détails dans les pièces, comme les fenêtres, les décorations, l'ameublement, etc., sauf pour l'ajout de portes et de textures simples. Un exemple de plan d'étage généré par cette méthode est illustré à la figure 5.15.

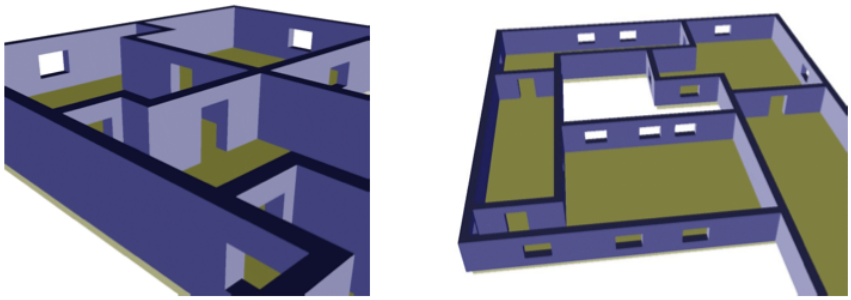


FIGURE 5.16 – Division d'un plan d'étage polygonal aligné sur les axes. Images tirées de [Bra05].

Bradley [Bra05] quant à lui présente un algorithme de division d'intérieurs permettant de

travailler sur un édifice de forme polygonale un peu plus complexe, mais toujours alignée sur les axes. Il ajoute des fenêtres simples (un trou rectangulaire) dans les murs extérieurs (voir figure 5.16).

Finalement, d'autres travaux [HDMB07, YWR09] se concentrent sur la génération d'un modèle géométrique d'édifices à partir de plans d'étage existants.

5.2 Modélisation par composants

Il y a plusieurs façons de construire un édifice, et puisque les opérations nécessaires sont nombreuses et variées, cette section ne doit pas être considérée comme un ensemble rigide de recettes à suivre à la lettre. Nous voulons plutôt donner au lecteur une certaine intuition, et montrer comment notre méthode de modélisation par composants peut être utilisée pour modéliser divers éléments détaillés particuliers aux édifices.

La création d'édifices peut en général être divisée en quatre étapes :

(1) *Le partitionnement de l'espace* divise un édifice en sections, principalement en couloirs, cages d'escalier, pièces et placards.

(2) *La création de la géométrie de base* ajoute les murs, les planchers, les plafonds et les toitures.

(3) *Les éléments architecturaux*, tels les portes, les fenêtres et les escaliers, sont ajoutés à la géométrie de base, généralement à l'aide d'opérations de connexion.

(4) *La décoration* ajoute la touche finale avec le positionnement de meubles, luminaires, cadres, etc.

Les trois premières étapes peuvent et même doivent alterner dans certains cas. Par exemple, lorsque le positionnement d'une fenêtre devrait influencer le placement d'un mur, des étapes de partitionnement de l'espace et de création de géométrie de base devraient aussi se produire après l'application des éléments architecturaux reliés.

5.2.1 Division de l'espace

Afin d'obtenir les composants 3D requis pour la création de la géométrie de base, on partitionne successivement l'espace de l'édifice à l'aide d'opérations diverses. En général, on procède comme suit.

On représente d'abord un édifice à l'aide d'un groupe de composants 3D définissant l'aspect général de sa frontière. Pour chacun des composants nécessitant un toit, sa face supérieure est convertie en un composant 2D, qui devient ensuite la base polygonale utilisée par une opération de création de toiture. Les différentes parties de cette forme de base sont par la suite combinées en un seul composant avec une opération booléenne d'union, résultant en une forme avec un intérieur bien défini, sans recouvrements. Un toit pourrait à la place être généré sur l'union de ces composants selon la forme désirée. Puis, on tranche l'édifice complet en étages, assignant automatiquement un numéro d'étage à chaque composant créé.

Pour chacun des composants d'un étage, on le segmente en couloirs et en appartements, en utilisant une combinaison d'opérations tels que le partitionnement, le tranchage et la création explicite de composants. Le numéro d'étage est automatiquement hérité comme attribut pour tous les composants nouvellement créés. De façon similaire aux numéros d'étage, un numéro d'appartement peut être assigné et hérité par ses pièces internes.

Les appartements sont alors subdivisés en pièces, aussi avec une combinaison de partitionnement, tranchage et de création explicite de composants. Les pièces créées sont catégorisées (salon, cuisine, chambre à coucher, salle de bain, etc.) en utilisant une ou plusieurs étiquettes du composant.

Certains composants (cage d'escalier et d'ascenseur, auditorium) créés après la génération des étages, peuvent s'étendent sur plusieurs étages et chevaucher plusieurs autres composants. Puisque la formation de la géométrie de base nécessite une subdivision de l'espace disjointe, les espaces s'intersectant doivent être assignés à un seul composant. Ceci est effectué en découpant judicieusement cet espace du volume des composants, soit explicitement, soit par un système automatique. Dans le cas automatique, l'usager assigne des priorités aux composants tel qu'expliqué à la section 4.3.3, et chaque composant se chevauchant sera découpé par les composants ayant une priorité plus élevée. Puisque la frontière d'un composant est fixe à la création, le résultat du découpage (à l'aide d'une opération booléenne de soustraction) est obtenu sous la forme de nouveaux composants enfant. Ceci est illustré à la figure 5.21d (et à la ligne 21 du listage 5.1) où un composant "ascenseur" chevauche des composants "appartement". Une opération de soustraction crée de nouveaux composants disjoints "pièce".

Les cages d'escalier traversant plusieurs étages nécessitent d'être séparées par étage afin de créer des groupes de marches alignés correctement avec les planchers. Bien que l'on puisse réutiliser les mêmes opérations de découpage (tranchage, partitionnement et alternance) utilisées

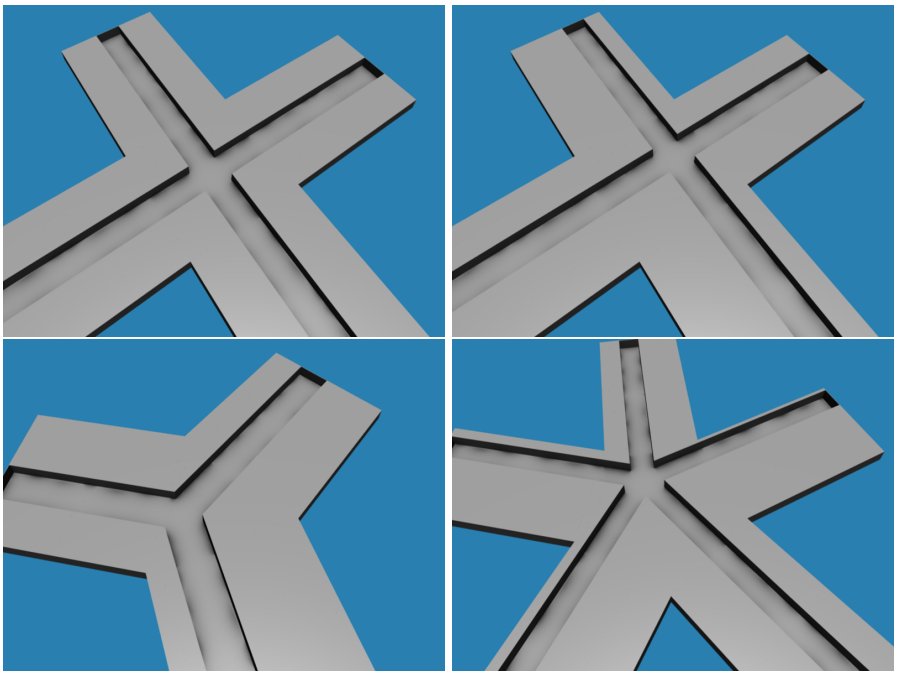


FIGURE 5.17 – Exemple de connexion de couloirs principaux entre plusieurs ailes rectangulaires. Des contraintes permettent d'aligner correctement la position des couloirs entre eux.

précédemment pour créer les étages, on peut aussi utiliser les contraintes à cette fin. En retrouvant les composants voisins des cages d'escalier à l'aide de requêtes, on peut extraire les composants 2D formant les planchers, et les convertir en contraintes planaires. Les cages d'escalier peuvent ensuite être tranchées avec ces contraintes pour récupérer des composants séparés exactement selon les étages.

Les contraintes sont aussi pratiques dans d'autres situations. Pour éviter d'avoir des murs perpendiculaires traversant une fenêtre, on spécifie des contraintes de répulsion sur l'espace occupé par les fenêtres, récupérées par une requête de voisinage. Ces contraintes vont ensuite guider le placement des murs générés par des opérations de découpage. La figure 4.9 (du précédent chapitre) illustre un exemple du résultat obtenu dans ce cas. D'autre part, des contraintes de déplacement peuvent être utilisées pour aligner des couloirs entre des ailes voisines d'un édifice (voir figure 5.17).

Bien que cette façon de partitionner l'espace soit flexible et puissante, il peut être difficile de produire l'ensemble des déclarations nécessaires. Ceci est particulièrement vrai lorsque l'on tente de créer un édifice pouvant supporter plusieurs variations distinctes dans sa forme et dans ses plans d'étage. Une façon plus simple (qui n'a pas été explorée, voir travaux futurs au chapitre 6) serait d'ajouter une nouvelle opération capable de partitionner un composant à l'aide d'une technique d'optimisation de plan d'étage, telle que celle proposée par Merrell *et al.* [MSK10].

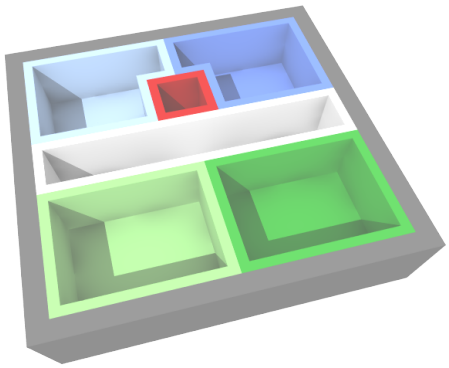


FIGURE 5.18 – Géométrie de base comprenant murs, planchers et façades (en gris) pour un édifice très simple.

Une telle opération cadrerait bien à l'intérieur de notre système de composants, en fournissant un contrôle précis (avec des opérations de plus bas niveau), et en laissant le système optimiser le partitionnement dans d'autres cas plus complexes.

5.2.2 Géométrie de base

On donne le nom de *géométrie de base* d'un édifice à la géométrie associée à la subdivision spatiale de ce dernier. Elle est composée des murs, planchers et plafonds de chaque pièce, couloirs et cages d'escalier (et d'ascenseur), ainsi que les murs extérieurs formant les façades. Ceci exclut les ouvertures, telles que les portes et fenêtres, ainsi que toutes les décorations (luminaires, peintures, meubles, etc.).

La géométrie de base est formée en commençant par extraire un sous-ensemble des composants 3D de l'arbre de tous les composants. Par la suite, les faces des frontières des composants 3D sont converties en composants 2D qui ensemble sont extrudés pour former les composants murs, planchers et plafonds. Ceci est illustré à la figure 5.22 où les composants étiquetés "pièce" sont choisis pour créer les composants 2D "mur" qui sont extrudés ensemble vers l'intérieur formant les composants "muri". Ces composants sont convertis en géométrie à la fin du programme.

Les composants internes (pièces, couloirs, cages d'escalier, etc.) sont extrudés ensemble en direction de leur intérieur, alors que les composants externes (les éléments de la façade) sont extrudés ensemble vers l'extérieur de l'édifice. La figure 5.18 montre les extrusions internes et externes d'un édifice très simple.

5.2.3 Façade

Les façades sont fabriquées à partir des murs externes (composants 2D) d'un édifice. Elles sont créées à l'aide d'une série d'opérations de partitionnement et de tranchage, d'une manière très similaire aux techniques précédentes de génération de façades telles que celle présentée dans les travaux de Wonka *et al.* [WWSR03]. L'extrusion des composants 2D crée des corniches, des murs de briques, etc., alors que des régions sont définies pour le placement des fenêtres, balcons et portes (voir prochaine section 5.2.4).

Puisqu'il est possible que l'ensemble des composants définissant la forme de base de l'édifice s'intersecte, on ne peut utiliser directement leurs faces pour former les murs extérieurs car certaines portions de ces faces seront contenues à l'intérieur de l'édifice. On obtient les murs extérieurs en fusionnant (opération d'union) tous les composants définissant la forme générale de l'édifice. Ces composants fusionnés forment l'enveloppe externe de l'édifice. On peut combiner ces composants avant ou après qu'ils soient découpés en étages, si l'on veut ou non que les éléments des façades correspondent au positionnement des étages. Lorsque le positionnement des étages est respecté, l'attribut indiquant le numéro de l'étage (attribut hérité) peut contrôler des variations de motifs (voir section 5.3.2). Les étiquettes assignées aux faces des composants de base peuvent aussi être utilisées pour identifier et varier le type de façade (e.g. en identifiant la zone de l'entrée principale à l'avant de l'aile principale de l'édifice).

5.2.4 Éléments architecturaux

Jusqu'à ce point, toute la géométrie créée forme un ensemble fermé hermétiquement. Pour accéder aux pièces, on doit découper de la géométrie entre elles. Ceci est fait avec des opérations booléennes (en particulier des soustractions) sur la géométrie contenue dans les composants et non pas sur leurs frontières.

Les portes et fenêtres sont donc créées avec une combinaison d'opérations booléennes qui vont dans un premier temps percer un trou dans le mur avant d'ajouter le cadre, puis la porte ou la fenêtre. Cette opération booléenne composite (soustraction suivie d'une union) est enregistrée dans la géométrie du composant et son positionnement est contrôlé par le mécanisme de connexion au sein d'une région. Le placement de portes avoisinant des couloirs est fait en calculant le facteur d'occultation occasionnée par le couloir sur les murs des pièces.

D'autres éléments architecturaux tels que des balcons (voir figures 5.33 et 5.35) et des escaliers comptent aussi sur les connexions pour leur positionnement, mais ne requièrent pas de

découper des trous.

Encore une fois, la variation est aisément possible en interpétrant des attributs usager assignés aux composants, par les étiquettes des frontières ou encore par la composition des composants voisins (e.g. varier le type de portes selon qu'elle mène à un couloir, une autre pièce ou un placard).

5.2.5 Décoration

Le mobilier est positionné à l'aide de régions (généralement 3D) placées sur le plancher des pièces. Certains éléments, tels que des étagères, armoires, appareils électroménagers jouxtent généralement les murs, alors que d'autres éléments tels que des tables, tapis, divans ont généralement tendance à être plus centrés dans une pièce. Les différentes régions d'un plancher sont définies et utilisées en conséquence.

Les luminaires sont placés sur le plafond d'une pièce, pas trop près des murs ou sur les murs, alors que les lampes sont placées près de ou sur d'autres meubles. Des précautions doivent être prises pour les éléments montés sur les murs (cadres, étagères, luminaires, etc.) pour ne pas chevaucher une fenêtre ou une porte. Dans ce dernier cas il faut aussi inclure son espace d'ouverture. Pour y arriver, on utilise une combinaison de contraintes, de requêtes de voisinage et de facteurs d'occultation.

Bien que l'on a généré procéduralement tous les éléments dans notre système, un usager est libre d'attacher à n'importe quel composant de la géométrie de provenance externe. Par exemple, des modèles créés dans des logiciels commerciaux de modélisation peuvent être connectés n'importe où dans notre forêt de composants.

Il est important de noter que l'on n'a pas essayé d'optimiser le placement des éléments (comme discuté à la section 5.1.3), même si les résultats seraient meilleurs. Ceci est laissé pour des travaux futurs. Dans notre cas, on a simplement utilisé des politiques manuelles ou de placements simples en plus de les varier aléatoirement pour augmenter la diversité des résultats.

5.2.6 Matériaux

Les matériaux pour les différentes parties d'un édifice sont assignés lors de la création de la géométrie (généralement à l'appel de *solidGeometry*). Ils sont définis par indice et associés par la suite (après la formation de la géométrie) à une liste décrivant concrètement les spécifications des matériaux. Ceci donne la possibilité de réutiliser un même édifice avec différents matériaux

sans avoir à reconstruire une nouvelle géométrie.

Les indices sont généralement passés par les attributs usager, et hérités d'un composant parent, ce qui est un moyen efficace pour propager cette information. Dans les différents exemples de ce chapitre, un attribut "couleur" est utilisé à cette fin. Malgré le nom évocateur de l'attribut, des matériaux complexes avec textures pourraient y être assignés. Une paramétrisation automatique est assignée à la géométrie pour permettre l'utilisation de textures et dans le cas où cette paramétrisation ne convient pas, une nouvelle paramétrisation peut être définie procéduralement comme expliqué au chapitre 3, ection 3.1.4.

5.3 Exemples

Maintenant que toutes les bases sont posées, on va pouvoir s'attarder à des cas réels et précis contenant entre autres des extraits de code, des graphes de composants et des figures montrant les résultats des étapes intermédiaires. On va d'abord commencer par introduire un exemple simple et complet contenant la plupart des étapes présentées à la section précédente (sauf l'ameublement). Ensuite, on regardera différentes variantes de deux sous-étapes importantes soient la formation de façades et le partitionnement intérieur.

5.3.1 Édifice simple

Le listage 5.1 (toutes références dans cette section à des lignes de code y sont reliées sauf avis contraire) montre un exemple simple contenant les principales étapes de construction d'un édifice, alors que la figure 5.19 correspond à une partie du graphe de composants créé après avoir exécuté ce programme.

La première étape (figure 5.21a), toujours présente, consiste à créer la forme de base de l'édifice (ligne 2). Cette étape est l'une des rares ne nécessitant pas de requêtes puisqu'elle créer un ou des composants racine. Dans ce cas, il s'agit d'un composant en forme de boîte auquel on attribut un indice de couleur (le matériau) qui sera hérité par tous ses enfants sauf ceux qui le redéfiniront. Ce composant est ensuite séparé en étages à l'aide d'une opération de tranchage (lignes 4 à 6 et figure 5.21b). Le nombre de composants "étage" résultant de cette opération dépend de la taille du composant parent "base" qui est découpé selon son axe Y en tranches de 2.5 unités. Dans le cas présent, deux étages sont créés et chacun se voit assigné un numéro dans un attribut "niveau" en utilisant un compteur. Le compteur est un objet retourné par le constructeur *counter()*, qui s'incrémente à chaque appel. Lorsqu'utilisé dans une opération

créant plusieurs composants (e.g. *slice, split, alternate*), il est appelé pour chaque assignation à un attribut de chacun des composants nouvellement créés. Après avoir séparé l'édifice en étages, chacun est partitionné en couloir et espaces (lignes 7 à 11) et chaque espace en appartements (lignes 12 à 15 et figure 5.21c) avec des opérations de partitionnement. Un composant multi-étage est ensuite ajouté (ligne 17) et soustrait (lignes 20 à 22 et figure 5.21d) pour répartir l'édifice en pièces disjointes. Cet ensemble de déclarations complète le partitionnement spatial de l'édifice.

La prochaine étape consiste à générer la géométrie de base et ses composants reliés. On débute par l'intérieur (lignes 25 à 33, figure 5.22a), suivi de l'extérieur (lignes 35 à 40, figure 5.22b). Dans chacun de ces deux cas, on commence par créer des composants 2D à partir des faces de la frontière des composants 3D retrouvées par une requête de face (*fquery*). Ces nouveaux composants 2D sont séparés en murs et en planchers à l'aide des étiquettes de frontières prédéfinies *SIDE* et *BOTTOM* respectivement. Puis ensemble, les composants 2D sont extrudés vers l'intérieur (facteur de -0.1 à la ligne 32) ou vers l'extérieur (facteur de 0.2, ligne 40) pour former les composants 3D représentant la géométrie de base. La géométrie actuelle est en fait créée à la fin (lignes 63 à 65). Pour que l'extrusion d'ensemble donne le résultat escompté, elle est exécutée pièce par pièce. Pour ce faire, l'extrusion est imbriquée à l'intérieur de la requête itérant sur chacune des pièces et le composant représentant la pièce courante est passé en paramètre à la requête utilisée pour l'extrusion. Ceci a pour effet de limiter les composants retournés "mur" et "plancher" aux enfants du composant courant.

Maintenant que les murs et les planchers sont formés et séparent l'édifice en pièces hermétiques, il faut ajouter les portes pour relier les pièces entre elles. Des régions 2D sont assignées aux composants 2D "mur" reliant les pièces aux couloirs (lignes 43 à 46) et la cage d'escalier aux couloirs (lignes 50 à 52). Afin de sélectionner les bons composants, un facteur d'occultation vis-à-vis du couloir est calculé et doit être non nul, ce qui est la cas seulement pour les composants touchant à un couloir. Avant de créer les régions pour les murs de la cage d'escalier, on doit d'abord séparer ses murs par étage (lignes 47 à 49) puisqu'il s'agit d'un composant multi-étage. Enfin, des portes sont connectées au centre (0.5, en horizontal) des régions (lignes 55 à 60 et figure 5.23). Différents types de porte sont choisis selon que la région appartienne à une pièce ou à la cage d'escalier, ce qui est déterminé en vérifiant les étiquettes des composants parent.

En regardant le graphe de composants à la figure 5.19 construit en exécutant ce programme, on peut voir que la façade a été construite à partir du composant initial. Ce n'est pas mauvais,

mais puisqu'en général les façades sont alignées avec les étages intérieurs, il serait plus aisé de construire les façades à partir des composants étage. Cette modification est donc apportée aux lignes 35 à 39 du listage 5.2 (à partir de maintenant les références au code seront associées à ce listage) et peut être vue à la figure 5.20. Pour complexifier l'apparence de la façade, on la sépare en deux sections (lignes 40 à 42), la corniche et le reste du mur (appelé ici arbitrairement "f1") avant de les extruder séparément avec des distances différentes (lignes 43 et 44, et figure 5.24). Afin d'ajouter des fenêtres dont le nombre dépend de la largeur de la façade de l'édifice, on utilise une opération de tranchage en spécifiant la taille occupée par sa région (lignes 58 à 60). De la même façon dont les portes ont été créées, on crée pour les fenêtres des régions (lignes 61 à 63) puis on connecte sur celles-ci des composants extérieurs contenant la géométrie de trou et de la fenêtre (lignes 73 à 78). À la seule différence, cette fois-ci, la connexion s'effectue centrée selon les deux axes (0.5 vertical et 0.5 horizontal). Finalement, pour donner un peu de diversité, on change le type de fenêtre selon le numéro d'étage qui peut être obtenu par l'attribut hérité "niveau". On voit donc ici un exemple de l'utilité et de la flexibilité obtenues par les attributs usager, ainsi que par le concept d'héritage.

```
// Partitionnement principal.
component{ label="base", size={10,5,10}, couleur=1 }

for c in query( "base" ) do
   slice( c, "Y", { label="étage", abs=2.5, niveau=counter() } )
end
for c in query( "étage" ) do
   split( c, "Z", { label="espace vital", rel=1 },
                  { label="couloir"     , abs=2 },
                  { label="espace vital", rel=1 } )
end
for c in query( "espace vital" ) do
   split( c, "X", { label="appartement", rel=1 },
                  { label="appartement", rel=1 } )
end

component{ label={"ascenseur", "pièce"}, size={2,5,2}, position={4,0,2}, couleur=3 }

// Priorités.
for c in query( "appartement" or "couloir" ) do
   subtract( c, query( "ascenseur" ), { label="pièce" } )
end

// Charpente
for c in query( "pièce" ) do
   for f in fquery( c, "SIDE" ) do
      component{ c, label="mur", boundary=f }
   end
   for f in fquery( c, "BOTTOM" ) do
      component{ c, label="plancher", boundary=f }
   end
   extrude( query( c, "mur" or "plancher" ), -0.1, { label="muri" } )
end

for c in query( "base" ) do
   for f in fquery( c, "SIDE" ) do
      component{ c, label="facade", boundary=f }
   end
end
extrude( query( c, "facade" ), 0.2, { label="mure", couleur=0 } )

// Régions
for c in query( "mur" and parent("appartement")
                      and occlusion("couloir") > 0 ) do
   region{ c, label="porte" }
end
for c in query( "mur" and parent("ascenseur") ) do
   slice( c, "Y", { label="mura", abs=2.5 } )
end
for c in query( "mura" and occlusion("couloir") > 0 ) do
   region{ c, label="porte" }
end

// Éléments architecturaux
for r in rquery( "porte" and parent("mur") ) do
   connect( componentFromFile("porte"), r, {0.5,0,0}, {0,0.1,-0.1} )
end
for r in rquery( "porte" and parent("mura") ) do
   connect( componentFromFile("porte2"), r, {0.5,0,0}, {0,0.1,-0.1} )
end

// Création de la géométrie
for c in query( "muri", "mure" ) do
   solidGeometry( c, c.couleur )
end
```

LISTAGE 5.1 – Pseudocode pour générer l'exemple illustré par le graphe de composants de la figure 5.19 et des figures 5.21 à 5.23.

```
24 // Charpente
25 for c in query( "pièce" ) do
26    for f in fquery( c, "SIDE" ) do
27       component{ c, label="mur", boundary=f }
28    end
29    for f in fquery( c, "BOTTOM" ) do
30       component{ c, label="plancher", boundary=f }
31    end
32    extrude( query( c, "mur" or "plancher" ), -0.1, { label="muri" } )
33 end
34
35 for c in query( "étage" ) do
36    for f in fquery( c, "SIDE" ) do
37       component{ c, label="facade", boundary=f }
38    end
39 end
40 for c in query( "facade" ) do
41    split( c, "Y", { label="f1", rel=1 }, { label="corniche", abs=0.2 } )
42 end
43 extrude( query( c, "f1" ), 0.2, { label="mure", couleur=0 } )
44 extrude( query( c, "corniche" ), 0.3, { label="mure", couleur=0 } )
45
46 // Régions
47 for c in query( "mur" and parent("appartement")
48                       and occlusion("couloir") > 0 ) do
49    region{ c, label="porte" }
50 end
51 for c in query( "mur" and parent("ascenseur") ) do
52    slice( c, "Y", { label="mura", abs=2.5 } )
53 end
54 for c in query( "mura" and occlusion("couloir") > 0 ) do
55    region{ c, label="porte" }
56 end
57
58 for c in query( "f1" ) do
59    slice( c, "X", { label="f2", abs=4 } )
60 end
61 for c in query( "f2" ) do
62    region{ c, label="fenêtre" }
63 end
64
65 // Éléments architecturaux
66 for r in rquery( "porte" and parent("mur") ) do
67    connect( componentFromFile("porte"), r, {0.5,0,0}, {0,0.1,-0.1} )
68 end
69 for r in rquery( "porte" and parent("mura") ) do
70    connect( componentFromFile("porte2"), r, {0.5,0,0}, {0,0.1,-0.1} )
71 end
72
73 for r in rquery( "fenêtre" and niveau==1 ) do
74    connect( componentFromFile("fenêtre"), r, {0.5,0.5,0}, {0,0.1,-0.1} )
75 end
76 for r in rquery( "fenêtre" and niveau>1 ) do
77    connect( componentFromFile("fenêtre2"), r, {0.5,0.5,0}, {0,0.1,-0.1} )
78 end
79
80 // Création de la géométrie
81 for c in query( "muri", "mure" ) do
82    solidGeometry( c, c.couleur )
83 end
```

LISTAGE 5.2 – Modification du programme du listage 5.1 générant le graphe de composants de la figure 5.20. Les différentes modifications apportées peuvent être vues à la figure 5.24.

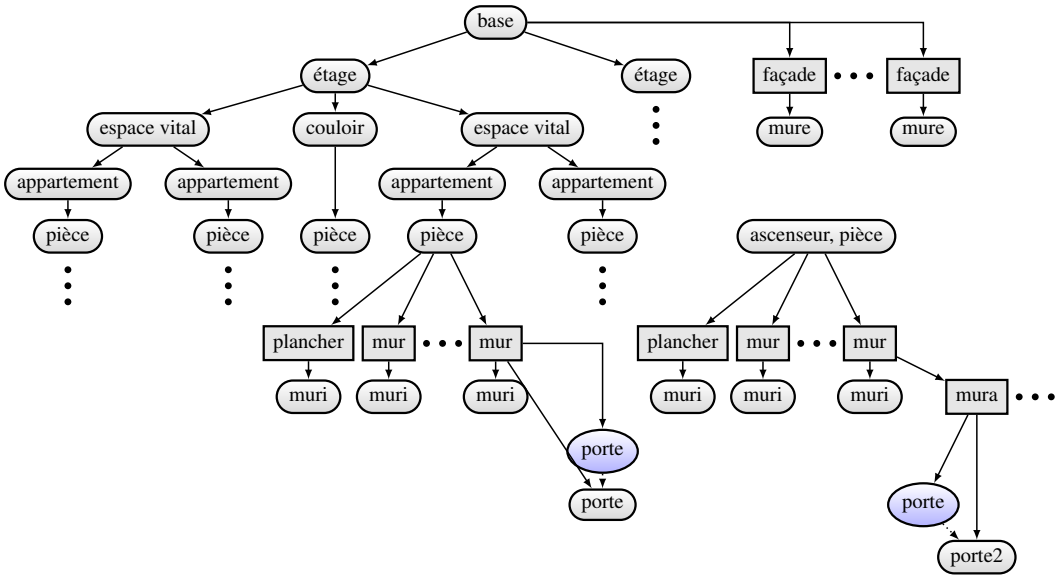


FIGURE 5.19 – Graphe de composants dont le résultat se trouve à la figure 5.23b.

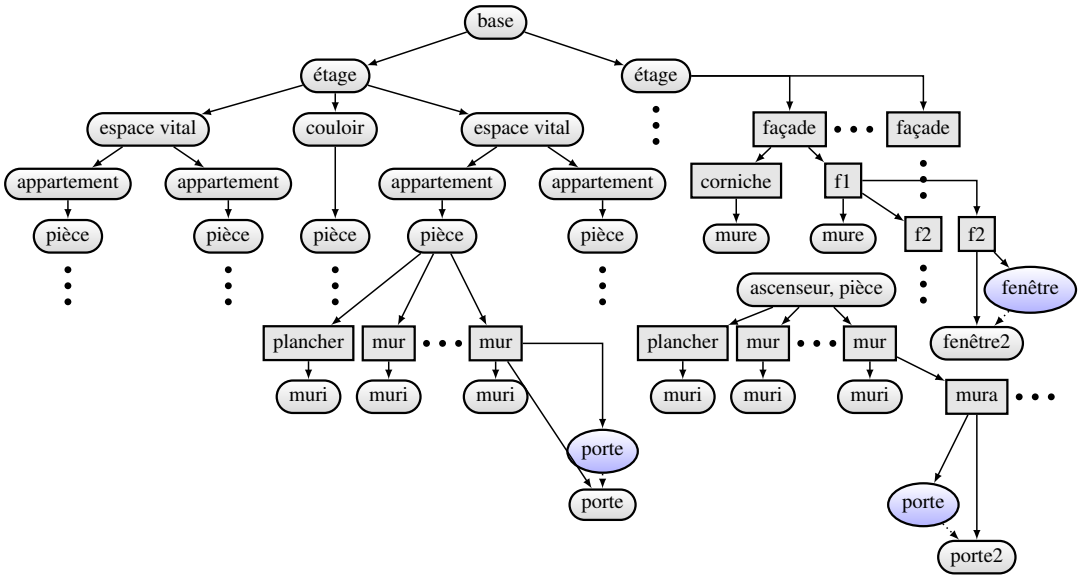
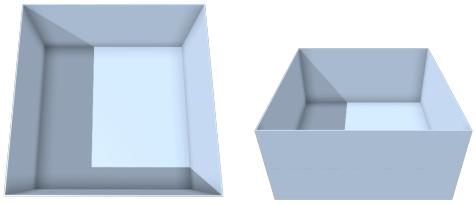
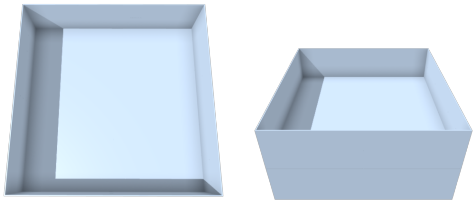


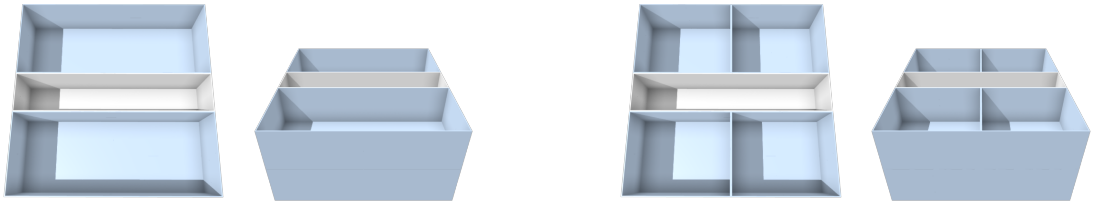
FIGURE 5.20 – Graphe de composants dont le résultat se trouve à la figure 5.24c.



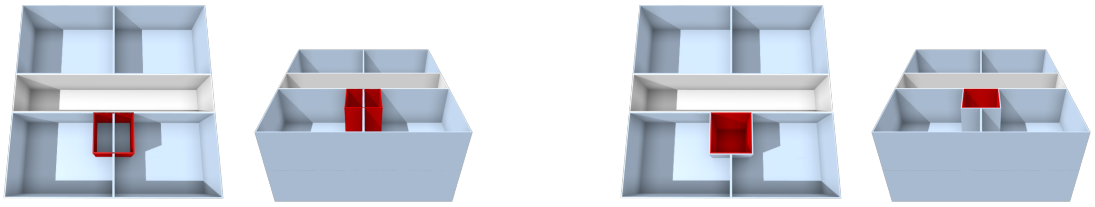
(a) Composant de base.



(b) Séparation en étages à l'aide d'une opération de tranchage (*split*).

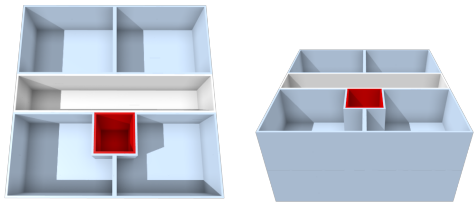


(c) Partitionnement de l'espace interne de l'édifice en utilisant des opérations de partitionnement.

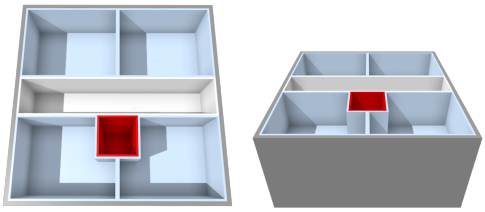


(d) Ajout d'un composant multi-étage (en rouge) définissant l'espace d'une cage d'ascenseur (à gauche). L'espace occupé chevauche les espaces des appartements. À droite, une opération booléenne de soustraction entre les appartements et la cage d'escalier est utilisée pour produire des composants pièce dont les espaces ne se recouvrent pas.

FIGURE 5.21 – Division de l'espace pour un édifice simple. Chaque étape est représentée par deux points de vue différents.

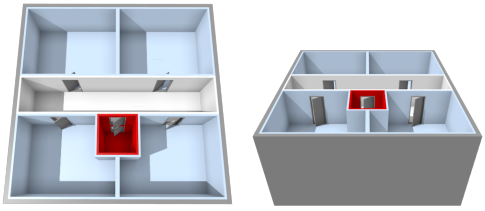


(a) Extrusion interne des composants 2D formant les faces des pièces.

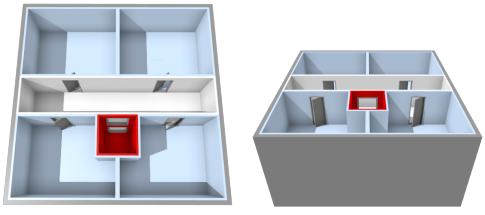


(b) Extrusion externe des composants 2D formés à partir des faces de côté du composant principal.

FIGURE 5.22 – Construction des composants représentant les murs internes et externes.

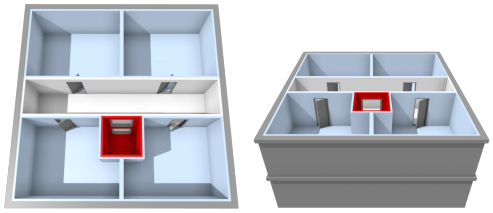


(a) Ajout de portes par connexions.

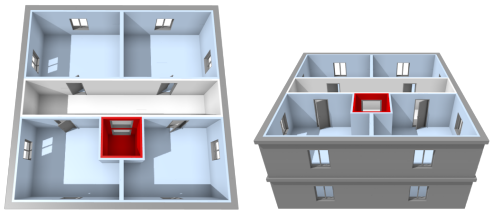


(b) Changement du type de porte pour l'ascenseur.

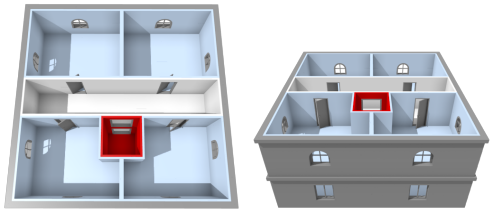
FIGURE 5.23 – Ajout des éléments architecturaux.



(a) Complexification de la façade en ajoutant des corniches. Les façades sont maintenant construites à partir des composants étage et non du composant principal.



(b) Ajout de fenêtres par connexions.



(c) Utilisation d'attributs usager pour ajuster le type de fenêtre selon le niveau de l'étage.

FIGURE 5.24 – Quelques modifications apportées à l'édifice.

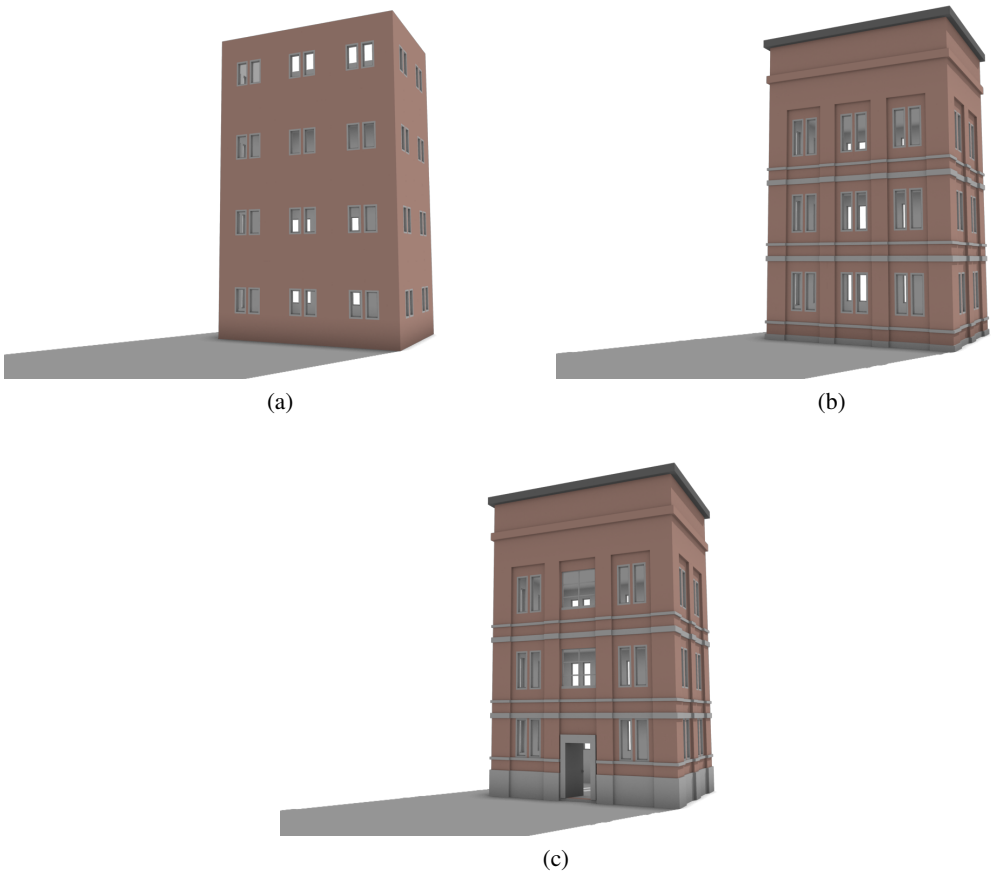
5.3.2 Façade

Les façades sont une des parties importantes d'un édifice, qui lui confèrent son style et sa personnalité. Cette section s'attardera à expliquer comment générer une façade relativement complexe et quels sont les moyens pour y parvenir, entre autres en faisant bon escient des compteurs, des étiquettes apposable sur les frontières et des opérations d'extrusion et d'alternance.

Le pseudocode du listage 5.3 constitue un squelette conventionnel pour une façade simple. Après avoir créé la forme de l'édifice et de l'avoir séparée en étages (lignes 2 à 6), chacune de ses faces est extirpée pour produire un composant 2D (lignes 18 à 20) servant à former la base de la façade. La construction d'une façade est principalement un processus de partitionnement 2D fait à partir de ce composant. Ce partitionnement a pour but de séparer la façade selon les matériaux et le relief désiré, ainsi que de positionner les régions pour connecter les éléments architecturaux, principalement des fenêtres et des portes, mais possiblement des balcons, escaliers ou autres. Après ce partitionnement, les différents composants sont extrudés (ligne 24) pour former le volume de la façade, puis les éléments architecturaux sont apposés (lignes 30 à 33) et finalement, la géométrie est ajoutée (lignes 36 à 38).

Dans le cas du squelette de façade résultant à la figure 5.25a, chaque étage exécute les mêmes opérations sans distinction de son positionnement, qu'il soit le premier ou même le dernier. Le motif simple de partitionement sans relief contenant une région de fenêtre par subdivision est généré à l'aide d'une opération de tranchage (*slice*) permettant d'adapter automatiquement le nombre de ces subdivisions (et donc de fenêtres) selon la taille de l'édifice. Un point à noter dans cet exemple, deux fenêtres étroites sont connectées par région. Aucune restriction nous oblige à connecter un seul élément par région, ce qui laisse la place à bien des variations, même avec un seul motif de partitionnement.

Plusieurs modifications sont apportées à cet exemple de base pour obtenir la façade présentée à la figure 5.25b. Ces modifications sont exprimées dans le listage 5.4 remplaçant complètement le code de la section "Façades". Le premier changement consiste à appliquer un motif de partitionnement différent selon le niveau de l'étage (lignes 22 à 31) obtenu par un attribut usager. Dans le cas présent, le dernier étage est différencié des autres. Ensuite le motif de partitionnement est complexifié pour contenir plusieurs matériaux différents (attribut "couleur") et un relief diversifié. Cette fois le partitionnement horizontal est effectué par l'opération d'alternance (ligne 23) au lieu de celui de tranchage. Contrairement à cette dernière, l'opération d'alternance permet de bien répartir l'espace lorsque l'on utilise un motif à deux composants principaux s'alternant



(a) (b) (c)

FIGURE 5.25 – Différents niveaux de complexité d'une façade construite à partir du pseudocode des listages 5.3 à 5.5.

comme c'est le cas ici et aussi dans beacoup d'autres cas en général.

Dans la prochaine série de modifications (figure 5.25c et listage 5.5), le premier étage est dissocié des autres pour changer son apparence et principalement pour lui ajouter une porte, alors que l'apparence de la section verticale centrale contenant la porte est changée. Pour y parvenir l'attribut usager est encore employé (lignes 24 à 29) pour changer la suite d'opérations appliquées (lignes 25 et 53 à 68) au premier étage. Pour identifier la section verticale centrale, un compteur est utilisé de la même façon que pour distinguer les étages, mais cette fois-ci, il est spécifiquement attribué aux deuxièmes composants des opérations d'alternance (lignes 25 et 27). Puisqu'on veut ajouter une porte strictement sur la devanture de l'édifice et non sur les côtés ou l'arrière, il nous faut un moyen de reconnaître à quel côté un composant appartient. Pour ce faire, on se sert des étiquettes associées aux faces des frontières des composants. Ces étiquettes peuvent être assignées lors de la création explicite de la frontière (le paramètre *boundary* de la ligne 3 avec ses étiquettes "id"). Ces étiquettes peuvent par la suite faire partie d'une requête

```
1  -- Volumes principaux.
2  component{ label="base", size={7,12,5} }
3
4  for c in query( "base" ) do
5     slice( c, "Y", { label="étage", abs=3, niveau=counter() } )
6  end
7
8  -- Intérieur.
9  for c in query( "étage" ) do
10    for f in fquery( c, "SIDE", "BOTTOM" ) do
11       component{ c, label="mur", boundary=f }
12    end
13    extrude( query( c, "mur" ), -0.1, { label="muri", couleur=2 } )
14 end
15
16 -- Facades.
17 for c in query( "étage" ) do
18    for f in fquery( c, "SIDE" ) do
19       component{ c, label="facade", boundary=f }
20    end
21    for c2 in query( c, "facade" ) do
22       slice( c2, "X", { label="f1", abs=2 } )
23    end
24    extrude( query( c, "f1" ), 0.1, { label="mure", couleur=1 } )
25 end
26
27 -- Éléments architecturaux.
28 for c in query( "f1" ) do region{ c, label="fenêtre" } end
29
30 for r in rquery( "fenêtre" ) do
31    connect( componentFromFile( "fenêtre04", {h=1,w=0.5,d=0.2} ), r, {0.5,0.5,0}, {-0.3,0,-0.1} )
32    connect( componentFromFile( "fenêtre04", {h=1,w=0.5,d=0.2} ), r, {0.5,0.5,0}, {0.3,0,-0.1} )
33 end
34
35 -- Création de la géométrie.
36 for c in query( "muri", "mure" ) do
37    solidGeometry( c, c.couleur )
38 end
```

LISTAGE 5.3 – Pseudocode pour générer l'exemple illustré à la figure 5.25a.

(*faceID*, lignes 78, 82 et 84) pour identifier précisément certains composants dérivant d'une face particulière. Contrairement aux fenêtres, lors de la connexion, la porte est centrée au sein de sa région seulement horizontalement et positionnée verticalement pour être à la hauteur du plancher.

Finalement, ces façades peuvent être appliquées à un édifice de forme plus complexe. Le listage 5.6 crée trois ailes de différentes tailles s'intersectant, et les combine par une opération d'union (ligne 11). Le résultat peut être apprécié à la figure 5.26.

```
16 -- Facades.
17 for c in query( "étage" ) do
18    for f in fquery( c, "SIDE" ) do
19       component{ c, id="facade", boundary=f }
20    end
21    for c2 in query( c, "facade" ) do
22       if c2.niveau < 3 then
23          alternate( c2, "X", { label="f4", abs=0.5 }, { label="f3", abs=1.5 }, 3 )
24       else
25          split( c2, "Y",
26             { label="f2", abs=0.1, rel=1 },
27             { label="f5", abs=0.3 },
28             { label="f2", abs=0.1, rel=1 },
29             { label="f6", abs=0.3 }
30          )
31       end
32    end
33    for c2 in query( c, "f3" ) do
34       split( c2, "Y",
35          { label="f7", abs=0.2 },
36          { label="f9", abs=0.4 },
37          { label="f7", abs=0.1 },
38          { label="f1", abs=0.1, rel=1 }
39       )
40    end
41    for c2 in query( c, "f4" ) do
42       split( c2, "Y",
43          { label="f8", abs=0.2 },
44          { label="f2", abs=0.4 },
45          { label="f8", abs=0.1 },
46          { label="f2", abs=0.1, rel=1 }
47       )
48    end
49    extrude( query( c, "f1" or "f9" ), 0.1, { label="mure", couleur=1 } )
50    extrude( query( c, "f2" ), 0.20, { label="mure", couleur=1 } )
51    extrude( query( c, "f5" ), 0.30, { label="mure", couleur=1 } )
52    extrude( query( c, "f6" ), 0.40, { label="mure", couleur=4 } )
53    extrude( query( c, "f7" ), 0.15, { label="mure", couleur=2 } )
54    extrude( query( c, "f8" ), 0.25, { label="mure", couleur=2 } )
55 end
```

LISTAGE 5.4 – Modification du pseudocode précédent pour générer l'exemple illustré à la figure 5.25b.

```
1  -- Volumes principaux.
2  component{ label="base", size={7,12,5},
3     boundary={{0,0,0},{0,0,1},{1,0,1},{1,0,0}, id={"n","n","n","e","n","n"}}
4  }

   ⋮

18 -- Facades.
19 for c in query( "étage" ) do
20    for f in fquery( c, "SIDE" ) do
21       component{ c, id="facade", boundary=f }
22    end
23    for c2 in query( c, "facade" ) do
24       if c2.niveau == 0 then
25          alternate( c2, "X", { label="b2", abs=0.5 }, { label="b1", abs=1.5, col=counter() }, 3 )
26       elseif c2.niveau < 3 then
27          alternate( c2, "X", { label="f4", abs=0.5 }, { label="f3", abs=1.5, col=counter() }, 3 )
28       else
29          split( c2, "Y",

   ⋮

53    for c2 in query( c, "b1" ) do
54       split( c2, "Y",
55          { label="f7", abs=1.0 },
56          { label="f9", abs=0.4 },
57          { label="f7", abs=0.1 },
58          { label="f1", abs=0.1, rel=1 }
59       )
60    end
61    for c2 in query( c, "b2" ) do
62       split( c2, "Y",
63          { label="f8", abs=1.0 },
64          { label="f2", abs=0.4 },
65          { label="f8", abs=0.1 },
66          { label="f2", abs=0.1, rel=1 }
67       )
68    end

   ⋮

77 -- Éléments architecturaux.
78 for c in query( "f1" and col == 1 and niveau > 0 and faceID( "e" ) ) do
79    region{ c, label="fenêtre2" }
80 end
81 for c in query( "f1" and ( col != 1  or not faceID( "e" ) ) ) do
82    region{ c, label="fenêtre" }
83 end
84 for c in query( "b1" and col == 1 and faceID( "e" ) ) do
85    region{ c, label="porte" }
86 end
87
88 for r in rquery( "fenêtre" ) do
89    connect( componentFromFile( "fenêtre04", {h=1.5,w=0.5,d=0.2} ), r, {0.5,0.5,0}, {-0.3,0,-0.1} )
90    connect( componentFromFile( "fenêtre04", {h=1.5,w=0.5,d=0.2} ), r, {0.5,0.5,0}, {0.3,0,-0.1} )
91 end
92 for r in rquery( "fenêtre2" ) do
93    connect( componentFromFile( "fenêtre01" ), r, {0.5,0.5,0}, {0,0,-0.1} )
94 end
95 for r in rquery( "porte" ) do
96    connect( componentFromFile( "porte01" ), r, {0.5,0,0}, {0,0.1,-0.1} )
97 end
```

LISTAGE 5.5 – Modifications du pseudocode précédent pour générer l'exemple illustré à la figure 5.25c.

```
-- Volumes principaux.
component{ label="aile", size={7,12,5}, position={-10,0,2},
    boundary={{0,0,0},{0,0,1},{1,0,1},{1,0,0}, id={"n","n","n","e","n","n"}}
}
component{ label="aile", size={7,12,5}, position={10,0,2},
    boundary={{0,0,0},{0,0,1},{1,0,1},{1,0,0}, id={"n","n","n","e","n","n"}}
}
component{ label="aile", size={25,9,15}, position={-9,0,-10},
    boundary={{0,0,0},{0,0,1},{1,0,1},{1,0,0}, id={"n","n","n","n","n","n"}}
}
merge( query( "aile" ), { label="base" } )
```

LISTAGE 5.6 – Modification des composants de base pour générer la façade complexe illustrée à la figure 5.26, avec les mêmes déclarations de design de façade qu'aux listages 5.5 à 5.5.

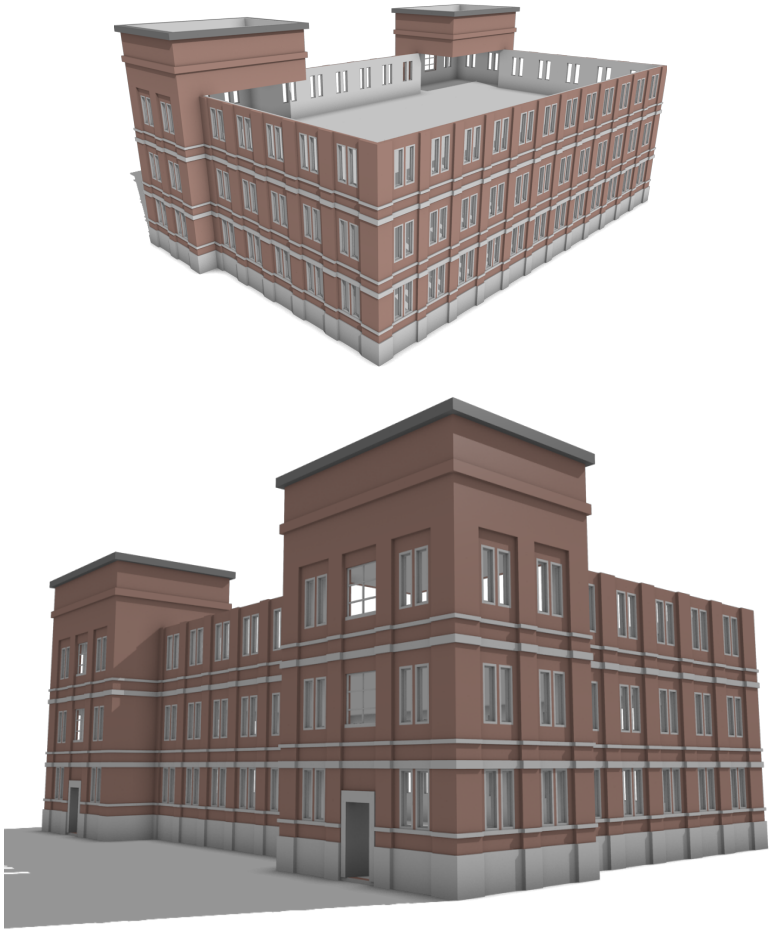


FIGURE 5.26 – Deux points de vue d'une façade complexe obtenue par l'union de multiple ailes (voir listage 5.6).

5.3.3 Intérieur

Un des avantages de la modélisation par composants, le dissociant des autres techniques et principalement des grammaires, est sa souplesse pour sélectionner ses éléments (composants dans le présent contexte) et sa flexibilité pour les combiner au sein d'opérations. Cet avantage nous permet d'ajouter des intérieurs complexes aux édifices, qui ne sont que des façades dans la plupart des autres systèmes. Cependant, créer un ensemble de déclarations pour générer des intérieurs est un processus plus difficile que celui pour les façades. Dans cette section trois approches différentes pouvant être combinées sont présentées pour donner une meilleure perspective sur ce processus.

Tranchage et partionnement

La première technique de modélisation d'intérieur présentée est celle utilisée dans l'exemple simple de la section 5.3.1. Elle consiste principalement à partitionner indépendamment les différentes ailes d'un édifice à l'aide d'opérations de tranchage (*slice*) et de partitionnement (*split*), et d'ajouter quelques pièces comme une cage d'escalier par opérations booléennes. Ce dernier point n'est pas montré dans cet exemple puisqu'il a déjà été présenté dans l'exemple simple précédent (figure 5.21d et listage 5.1), et aussi dans le but de ne pas surcharger le code.

Le listage 5.7 présente la base du pseudocode résultant en l'intérieur d'édifice illustré à la figure 5.27a. Toutes les ailes sont unies ensemble (ligne 4) pour former la base de l'édifice servant à produire la façade (lignes 46 à 51). Ici, puisque l'emphase est sur la création de l'intérieur, la façade consiste simplement en un mur uniforme sans autres éléments. Chaque aile est divisée en un couloir principal (lignes 6 à 19) et en appartements (lignes 28 à 33). Ces appartements pourraient être par la suite partitionnés en pièces dans un cas concret. Avant de partitionner l'espace en appartements, l'aile secondaire ("aile2") est pourvue d'un couloir secondaire (lignes 20 à 26) permettant de lier son couloir principal à celui de l'aile principale. Sans prendre les précautions nécessaires, les deux couloirs (principal et secondaire) ne se joindront pas correctement comme illustré à la figure 5.27a. Ce problème peut être corrigé en ajustant les paramètres de l'opération de partitionnement. Toutefois, pour être robuste aux déplacements de l'aile et par le fait-même produire un programme flexible et paramétrable, une contrainte peut être utilisée pour forcer le positionnement correct. Le listage 5.8 montre comment créer cette contrainte à partir du couloir principal (lignes 21 à 24) et son utilisation (ligne 29), alors que la figure 5.27b affiche le résultat. Pour bien permettre aux deux plans formant le couloir lors de

```
1  -- Volumes principaux.
2  component{ label="aile1", size={18,2,18}, position={-9,0,-9},   couleur=2 }
3  component{ label="aile2", size={10,2,25}, position={-19,0,-12}, couleur=2 }
4  merge( query( "aile1", "aile2" ), { label="base" } )
5
6  for c in query( "aile1" ) do
7     split( c, "Z",
8        { label="appartement_z", rel=1 },
9        { label="div_b", abs=2 },
10       { label="appartement_z", rel=1 }
11    )
12 end
13 for c in query( "aile2" ) do
14    split( c, "X",
15       { label="appartement_x", rel=1 },
16       { label="div_c", abs=2 },
17       { label="div_a", rel=1 }
18    )
19 end
20 for c in query( "div_a" ) do
21    split( c, "Z",
22       { label="appartement_x", rel=1 },
23       { label="div_c", abs=2 },
24       { label="appartement_x", rel=1 }
25    )
26 end
27
28 for c in query( "appartement_x" ) do
29    slice( c, "Z", { label="pièce", 4 }, 2 )
30 end
31 for c in query( "appartement_z" ) do
32    slice( c, "X", { label="pièce", 4 }, 2 )
33 end
34
35 merge( query( "div_c", "div_b" ), { label="couloir", couleur=3 } )
36
37 -- Intérieur.
38 for c in query( "pièce", "couloir" ) do
39    for f in fquery( c, "SIDE", "BOTTOM" ) do
40       component{ c, label="mur", boundary=f }
41    end
42    extrude( query( c, "mur" ), -0.1, { label="muri" } )
43 end
44
45 -- Facades.
46 for c in query( "base" ) do
47    for f in fquery( c, "SIDE" ) do
48       component{ c, label="facade", boundary=f }
49    end
50    extrude( query( c, "facade" ), 0.3, { label="mure", couleur=1 } )
51 end
52
53 -- Création de la géométrie.
54 for c in query( "muri", "mure" ) do
55    solidGeometry( c, c.couleur )
56 end
```

LISTAGE 5.7 – Pseudocode pour générer l'exemple d'intérieur d'édifice illustré à la figure 5.27a.

l'opération de tranchage (lignes 26 à 30) de se positionner selon la contrainte, on donne une taille nulle aux composants précédent et suivant ("appartement_x"). Ceci donne toute la liberté de mouvement à ces plans qui en se déplaçant redistribuent l'espace restant à ces deux composants.

La capacité d'adaptation du nouveau pseudocode est démontrée lors du déplacement de l'aile

```
21 local cs={}
22 for c in query( "div_b" ) do
23    for f in fquery( c, "SIDE" ) do cs[#cs+1] = faceConstraint{ c, f } end
24 end
25 for c in query( "div_a" ) do
26    split( c, "Z",
27       { label="appartement_x", abs=0 },
28       { label="div_c", rel=1 },
29       { label="appartement_x", abs=0 }, cs
30    )
31 end
```

LISTAGE 5.8 – Modification du pseudocode précédent pour ajouter une contrainte sur la création du couloir secondaire. Le résultat est illustré à la figure 5.27b.

```
1 -- Volumes principaux.
2 component{ label="aile1", size={18,2,18}, position={-9,0,-9}, couleur=2 }
3 component{ label="aile2", size={10,2,25}, position={-19,0,-6}, couleur=2 }
```

LISTAGE 5.9 – Déplacement de l'aide secondaire ("aile2") pour démontrer le positionnement adaptative provoqué par la contrainte introduite dans le listages 5.8. L'intérieur obtenu est illustré à la figure 5.27c.

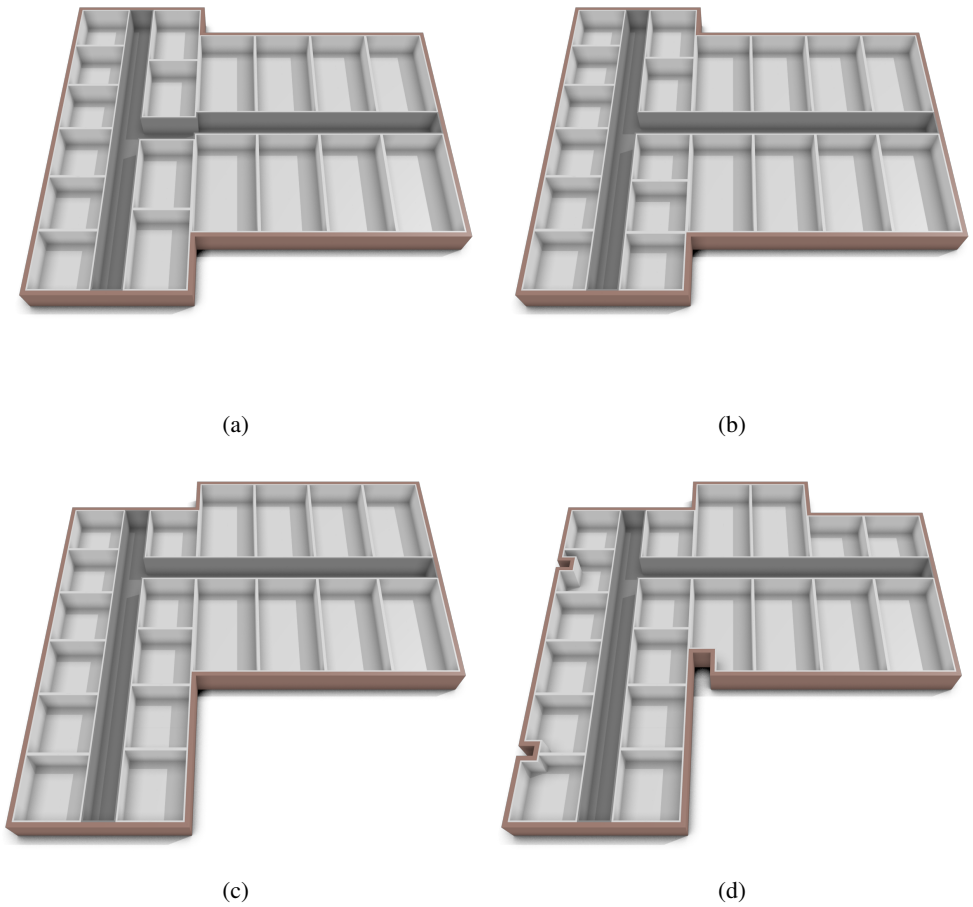
secondaire (listage 5.9 et figure 5.27c). Enfin, pour donner plus de complexité à la forme de l'intérieur, la frontière des ailes est changée sans affecter le reste des opérations. Le listage 5.10 donne la nouvelle définition de ces frontières, alors que la figure 5.27d illustre le résultat. Il est intéressant de noter ici que les opérations de partitionnement et de tranchage s'exécutent sur la frontière complexe et non sur sa boîte englobante.

```
1 -- Volumes principaux.
2 component{ label="aile1", size={18,2,18}, position={-9,0,-9}, couleur=2,
3    boundary={
4       {0,0,0},{0,0,0.9},{0.1,0,0.9},{0.1,0,1},{1,0,1},
5       {1,0,0.2},{0.5,0,0.2},{0.5,0,0}
6    },
7 }
8 component{ label="aile2", size={10,2,25}, position={-19,0,-6}, couleur=2,
9    boundary={
10       {0,0,0},{0,0,0.2},{0.1,0,0.2},{0.1,0,0.25},{0,0,0.25},{0,0,0.8},
11       {0.1,0,0.8},{0.1,0,0.85},{0,0,0.85},{0,0,1},{1,0,1},{1,0,0}
12    }
13 }
```

LISTAGE 5.10 – Changement des frontières des ailes formant l'édifice illustré à la figure 5.27d.

Soustraction

Pour créer correctement les murs des couloirs dans l'exemple présenté à la section précédente, on doit obtenir la forme complète des couloirs en les unissant par une opération booléenne (listage 5.7, ligne 35). Il peut aussi être difficile de créer des configurations complexes en ce restreignant à partitionner les espaces définissant l'édifice. Une autre façon de procéder que le



(a) (b)

(c) (d)

FIGURE 5.27 – Quatre variations d'intérieurs dont le pseudocode est accessible aux listages 5.7 à 5.10. (a) Sans contraintes, (b) avec contraintes pour le couloir secondaire, (c) déplacement de l'aile secondaire (de gauche) démontrant l'ajustement des couloirs, et (d) un contour un peu plus complexe.

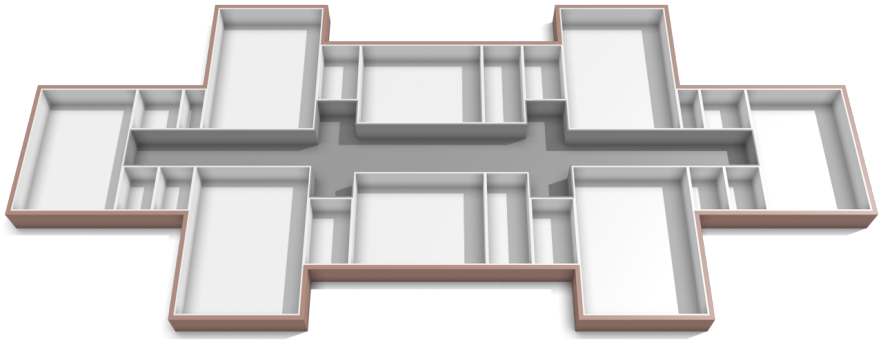


FIGURE 5.28 – Intérieur dont les couloirs sont obtenus par soustraction. Le pseudocode pour générer ce modèle d'édifice est donné au listage 5.11.

partitionnement récursif et présentée dans cette section est de générer un volume par soustraction.

Au listage 5.11, l'espace habitable de chaque aile, les pièces, est créé dans un premier temps. Ici, des opérations de partitionnement (lignes 9 à 44) servent à cette fin, mais tout autre moyen serait valide. Ensuite, après que tout l'espace habitable soit défini, le réseau de couloirs est extrait par une opération de soustraction (lignes 46 à 48) en soustrayant l'espace habitable à la forme complète de l'édifice (ligne 7). L'intérieur obtenu est illustré à la figure 5.28.

Bien qu'appliqué pour la génération de couloirs, ce principe de création d'espace par soustraction peut être appliqué pour n'importe quel type de pièce ou d'espace, et donne plus de flexibilité que la technique de la section précédente par tranchage.

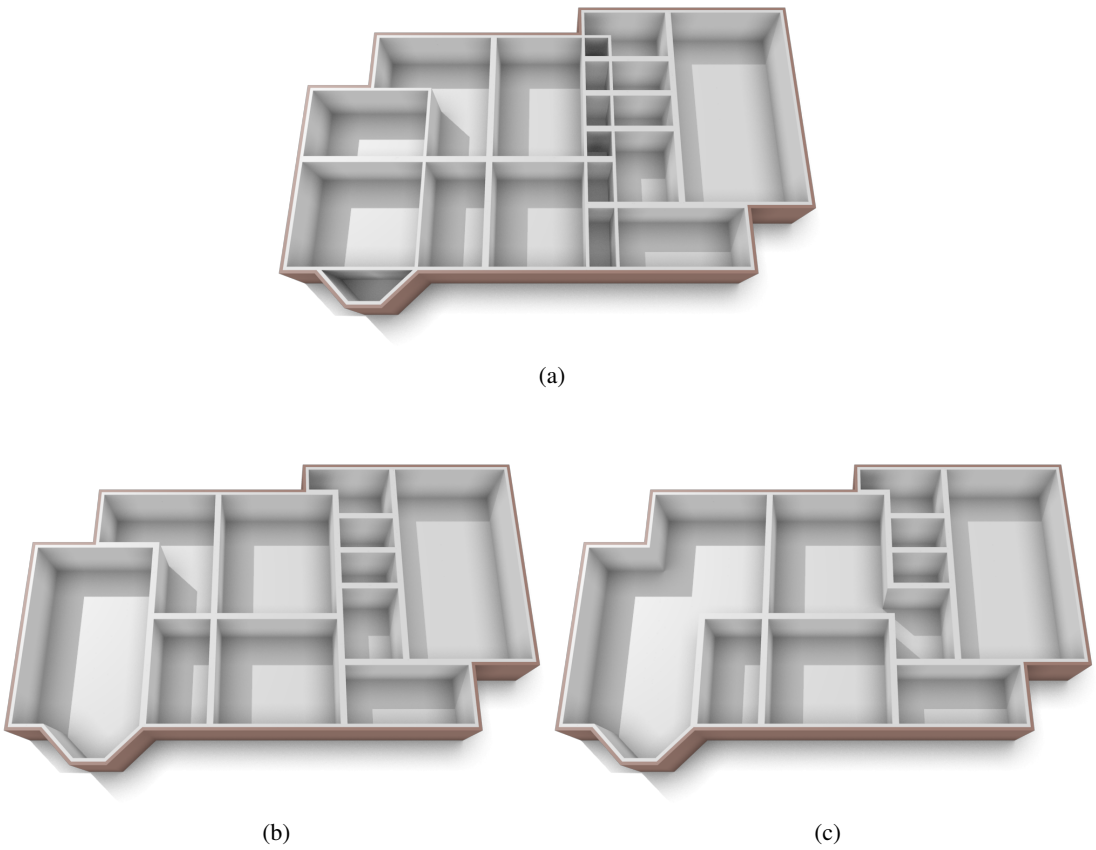
Priorités

Une autre façon de concevoir une séparation de l'espace complexe consiste à assigner l'espace par priorité. Le principe est de créer les volumes de chaque pièce sans se préoccuper du chevauchement de l'espace entre elles. C'est exactement ce que fait le listage 5.12 et dont le résultat est affiché à la figure 5.29a.

Maintenant pour distribuer l'espace de façon unique et éviter les chevauchements, il faut que les composants pièce soient disjoints. En assignant une priorité à chacun des composants (attribut "prio"), il est possible de créer en une opération (lignes 45 à 47 du listage 5.13) de soustraction les composants disjoints. Pour chaque composant "pièce_a" l'opération de soustraction découpe toutes les frontières des composants le chevauchant ayant une priorité supérieure définie par l'attribut passé en paramètre. Une illustration de ce résultat est montrée à la figure 5.29b.

```
-- Volumes principaux.
component{ label="aile1", size={18,2,18}, position={-9,0,-9},      couleur=2 }
component{ label="aile2", size={8,2,25},  position={-17,0,-12.5}, couleur=2 }
component{ label="aile3", size={8,2,25},  position={9,0,-12.5},   couleur=2 }
component{ label="aile4", size={12,2,10}, position={-29,0,-5},    couleur=2 }
component{ label="aile5", size={12,2,10}, position={17,0,-5},     couleur=2 }
merge( query( "aile1", "aile2", "aile3", "aile4", "aile5" ), { label="base" } )

for c in query( "aile2", "aile3" ) do
   split( c, "Z", { label="pièce", rel=1 }, { label="vide", abs=3 }, { label="pièce", rel=1 } )
end
for c in query( "aile4" ) do
   split( c, "X", { label="pièce", rel=1 }, { label="sub1", abs=5 } )
end
for c in query( "aile5" ) do
   split( c, "X", { label="sub2", abs=5 }, { label="pièce", rel=1 } )
end
for c in query( "sub1" ) do
   split( c, "Z", { label="sub3", rel=1 }, { label="vide", abs=3 }, { label="sub4", rel=1 } )
end
for c in query( "sub2" ) do
   split( c, "Z", { label="sub5", rel=1 }, { label="vide", abs=3 }, { label="sub4", rel=1 } )
end
for c in query( "sub3" ) do
   split( c, "X", { label="pièce", rel=2 }, { label="pièce", rel=1 } )
end
for c in query( "sub4" ) do
   split( c, "X", { label="pièce", rel=1 }, { label="pièce", rel=1 } )
end
for c in query( "sub5" ) do
   split( c, "X", { label="pièce", rel=1 }, { label="pièce", rel=2 } )
end
for c in query( "aile1" ) do
   split( c, "X", { label="sub6", abs=3 }, { label="sub7", rel=1 }, { label="sub6", abs=3 } )
end
for c in query( "sub6" ) do
   split( c, "Z", { label="pièce", abs=5 }, { label="vide", rel=1 }, { label="pièce", abs=5 } )
end
for c in query( "sub7" ) do
   split( c, "Z", { label="sub8", rel=1 }, { label="vide", abs=4 }, { label="sub8", rel=1 } )
end
for c in query( "sub8" ) do
   split( c, "X", { label="pièce", rel=1 }, { label="pièce", abs=3 } )
end

for c in query( "base" ) do
   subtract( c, query( "pièce" ), { label="couloir", couleur=3 } )
end

-- Intérieur.
for c in query( "pièce", "couloir" ) do
   for f in fquery( c, "SIDE", "BOTTOM" ) do
      component{ c, label="mur", boundary=f }
   end
   extrude( query( c, "mur" ), -0.1, { label="muri" } )
end
-- Facades.
for c in query( "base" ) do
   for f in fquery( c, "SIDE" ) do
      component{ c, label="facade", boundary=f }
   end
   extrude( query( c, "facade" ), 0.3, { label="mure", couleur=1 } )
end

-- Création de la géométrie.
for c in query( "muri", "mure" ) do
   solidGeometry( c, c.couleur )
end
```

LISTAGE 5.11 – Pseudocode pour générer l'exemple d'intérieur d'édifice illustré à la figure 5.28.



(a)

(b) (c)

FIGURE 5.29 – Trois variations d'intérieurs dont le pseudocode est accessible aux listages 5.12 à 5.14. (a) Sans priorité, (b) avec priorités, et (c) en fusionnant quelques pièces avant d'effectuer les priorités.

Une autre fonctionnalité entrant dans la confection de l'intérieur de cet exemple est le découpage de la frontière d'un enfant par son parent. Par défaut lorsqu'un composant est créé comme étant enfant d'un autre composant (lignes 9 à 29 du listage 5.12), sa frontière (ici une simple boîte) est découpée par le parent (ici la frontière complexe est définie aux lignes 3 à 6). Les opérations d'union peuvent aussi être combinées aux priorités sans aucun problème. Dans le listage 5.14 quatre pièces sont regroupées en deux (lignes 45 et 46) avant d'appliquer la soustraction par priorité. Le résultat obtenu à la suite de ces modifications est illustré à la figure 5.29c.

Toutes ces techniques présentées (tranchage avec contraintes, soustraction de volumes, priorités, union de volumes et découpage du parent) peuvent être utilisées ensemble sans aucun problème. En fait, c'est exactement ce qui est fait pour les exemples plus complexes présentés dans la prochaine section.

```
-- Volumes principaux.
component{ label="base",
    boundary={
        {0,0,3},{0,0,9},{1,0,9},{2,0,10},{3,0,10},{4,0,9},{14,0,9},{14,0,7},{16,0,7},
        {16,0,0},{9,0,0},{9,0,1},{2,0,1},{2,0,3}
    }
}

for c in query( "base" ) do
    component{ c, label="espace1", size={7,2,7}, position={9,0,0} }
    component{ c, label={"chambre", "pièce_a"},
        size={7,2,2},    position={9,0,7},    prio=1
    }
    component{ c, label={"salon",    "pièce_a"},
        size={4,2,4.5}, position={6,0,1},    prio=2
    }
    component{ c, label={"boudoir", "pièce_a"},
        size={4,2,3.5}, position={6,0,5.5}, prio=3
    }
    component{ c, label={"cuisine", "pièce_a"},
        size={6,2,4.5}, position={0,0,1},    prio=4
    }
    component{ c, label={"entree",   "pièce_a"},
        size={6,2,3.5}, position={0,0,5.5}, prio=5
    }
    component{ c, label={"salle",    "pièce_a"},
        size={4,2,7},    position={0,0,3},    prio=6
    }
end
for c in query( "espace1" ) do
    split( c, "X",
        { label="espace2", abs=3 },
        { label={"garage", "pièce_a"}, rel=1, prio=0.5 }
    )
end
for c in query( "espace2" ) do
    split( c, "Z",
        { label={"bain",     "pièce_a"}, rel=1.5, prio=0.1 },
        { label={"lavage",   "pièce_a"}, rel=1,    prio=0.2 },
        { label={"placard", "pièce_a"}, rel=1,    prio=0.3 },
        { label={"foyer",    "pièce_a"}, rel=2,    prio=0.4 }
    )
end

-- Intérieur.
for c in query( "pièce_a" ) do
    for f in fquery( c, "SIDE", "BOTTOM" ) do
        component{ c, label="mur", boundary=f }
    end
    extrude( query( c, "mur" ), -0.1, { label="muri", couleur=2 } )
end

-- Facades.
for c in query( "base" ) do
    for f in fquery( c, "SIDE" ) do
        component{ c, label="facade", boundary=f }
    end
    extrude( query( c, "facade", 0.1, { label="mure", couleur=1 } )
end

-- Création de la géométrie.
for c in query( "muri", "mure" ) do
    solidGeometry( c, c.couleur )
end
```

LISTAGE 5.12 – Pseudocode pour générer l'exemple illustré à la figure 5.29a. où les pièces se chevauchent.

```
44
45 for c in query( "pièce_a" ) do
46    subtract( c, "prio", { label="pièce" } )
47 end
48
49 -- Intérieur.
50 for c in query( "pièce" ) do
51    for f in fquery( c, "SIDE", "BOTTOM" ) do
52       component{ c, label="mur", boundary=f }
53    end
54    extrude( query( c, "mur" ), -0.1, { label="muri", couleur=2 } )
55 end
```

LISTAGE 5.13 – Modifications apportées au pseudocode précédent pour générer l'intérieur illustré à la figure 5.29b en appliquant des priorités.

```
9  for c in query( "base" ) do
10    component{ c, label="espace1", size={7,2,7}, position={9,0,0} }
11    component{ c, label={"chambre", "pièce_a"},
12       size={7,2,2},    position={9,0,7},    prio=1
13    }
14    component{ c, label={"salon",    "pièce_c"},
15       size={4,2,4.5}, position={6,0,1}
16    }
17    component{ c, label={"boudoir", "pièce_a"},
18       size={4,2,3.5}, position={6,0,5.5}, prio=3
19    }
20    component{ c, label={"cuisine", "pièce_b"},
21       size={6,2,4.5}, position={0,0,1}
22    }
23    component{ c, label={"entree",   "pièce_a"},
24       size={6,2,3.5}, position={0,0,5.5}, prio=5
25    }
26    component{ c, label={"salle",    "pièce_b"},
27       size={4,2,7},    position={0,0,3}
28    }
29 end
30 for c in query( "espace1" ) do
31    split( c, "X",
32       { label="espace2", abs=3 },
33       { label={"garage", "pièce_a"}, rel=1, prio=0.5 }
34    )
35 end
36 for c in query( "espace2" ) do
37    split( c, "Z",
38       { label={"bain",     "pièce_a"}, rel=1.5, prio=0.1 },
39       { label={"lavage",   "pièce_a"}, rel=1,   prio=0.2 },
40       { label={"placard", "pièce_a"}, rel=1,   prio=0.3 },
41       { label={"foyer",    "pièce_c"}, rel=2 }
42    )
43 end
44
45 merge( query( "pièce_b" ), { label="pièce_a", prio=6 } )
46 merge( query( "pièce_c" ), { label="pièce_a", prio=2 } )
```

LISTAGE 5.14 – Fusion de quelques pièces avant de procéder à la soustraction par priorité pour générer l'intérieur illustré à la figure 5.29c.

5.4 Résultats

Les figures 5.30 à 5.39 montrent divers modèles d'édifices créés avec notre système procédural par composants et en utilisant toutes les techniques présentées au cours de ce chapitre. La figure 5.30 illustre des variations d'un édifice à appartements. Toutes ces variations sont créées à l'aide d'un même programme en changeant les valeurs de deux graines (*seeds*) d'un générateur aléatoire, l'une contrôlant la disposition des ailes principales de l'édifice lui donnant sa forme, alors que l'autre contrôlant le positionnement des couloirs principaux ainsi que la distribution des pièces aux alentours. Des valeurs aléatoires servent aussi à décider de la configuration de l'ameublement simple.

La figure 5.31 présente un différent type d'édifice à appartements de style maison en rangée. On peut observer dans cet exemple la présence de quelques murs non-alignés sur les axes, textures de briques à l'extérieur, planchers en bois et carrelage, et quelques meubles. Une vue extérieure de différents types d'édifices (hôtel, loft industriel, Habitat 67) est illustrée à la figure 5.32. Il est intéressant de noter que dans le cas du dernier édifice (ressemblant à Habitat 67), la procédure standard de séparation de la forme générale de l'édifice en étages n'a pas été suivie, mais plutôt remplacée par le positionnement direct des unités d'appartements.

Notre système n'est pas limité à la création d'édifices. Il peut aisément servir à la construction de maisons avec toiture et pignons (voir figure 5.33). On peut noter la présence d'un balcon placé par une opération de connexion, d'une cheminée créée entièrement par composants et d'une opération de connexion pour l'entrée de son foyer, d'un grenier et de lucarnes.

Dans l'exemple d'une école (illustrée à la figure 5.34), on démontre un cas où le partitionnement intérieur d'une aile s'adapte à son emplacement. En effet, la position de l'aile influencera la position relative, à l'intérieur de celle-ci, de son couloir principal connectant au couloir de l'aile adjacente. Le déplacement du couloir a une répercussion sur l'espace disponible de chaque côté de ce dernier et le nombre de salles s'ajuste en conséquence.

Dans un dernier exemple d'édifice à appartements vu de l'extérieur à la figure 5.35 et de l'intérieur aux figures 5.36 à 5.38, on peut noter l'emplacement de décorations diverses et de l'ameublement. Des requêtes d'occultation servent à délimiter les zones n'obstruant pas la vue des fenêtres et l'ouverture des portes pour permettre le positionnement sans encombre. Finalement deux autres variations de partitionnement de l'espace pour un appartement sont montrées à la figure 5.39.

Dans tous les exemples présentés dans ce chapitre, la taille des programmes varie de quelques

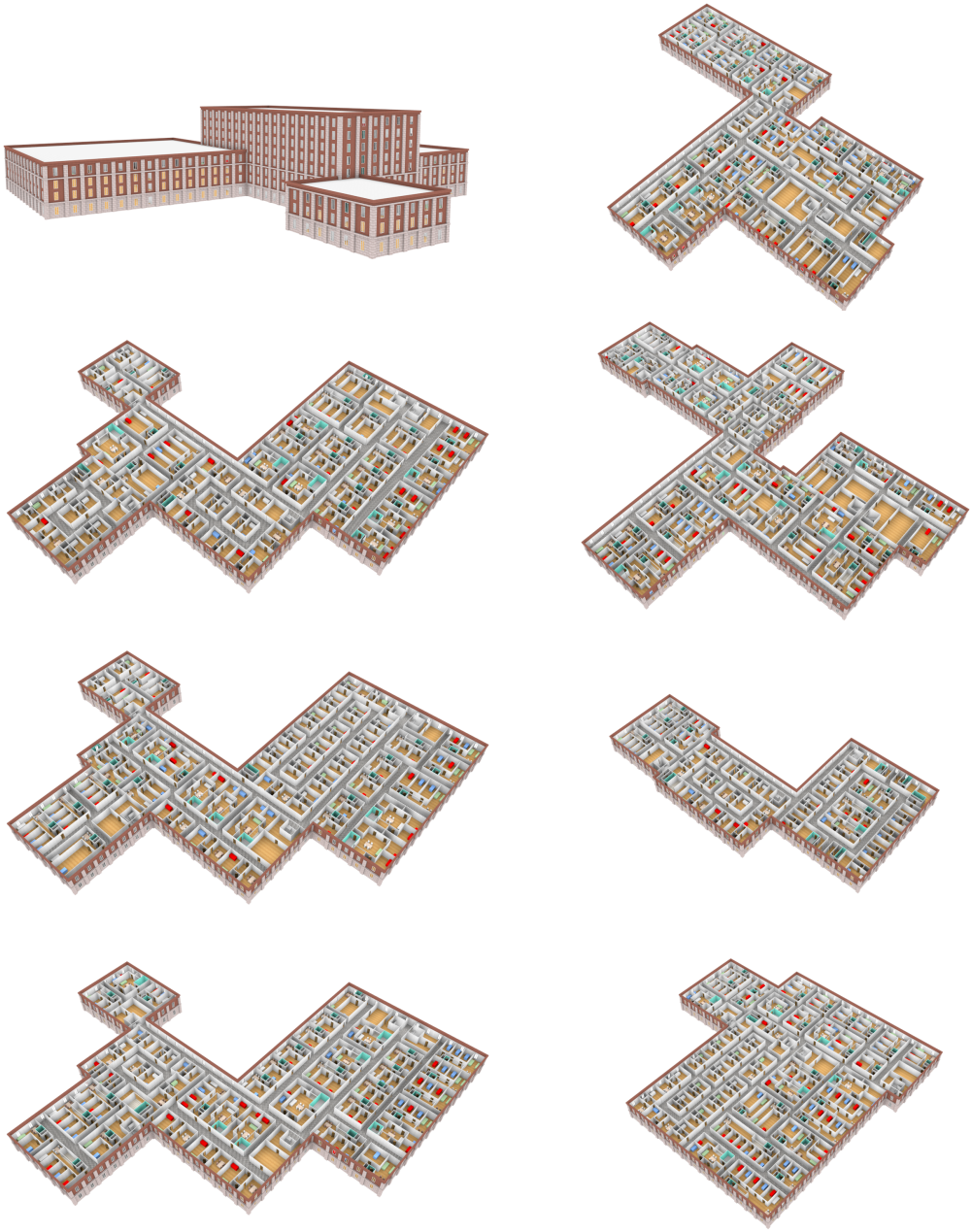


FIGURE 5.30 – Variations sur un édifice. À gauche : variations aléatoires sur la distribution des appartements, des corridors secondaires, des pièces et de l'ameublement pour une configuration d'ailes générées aléatoirement dans un édifice multi-étages. À droite : variations aléatoires de la forme des ailes et leurs contenus.

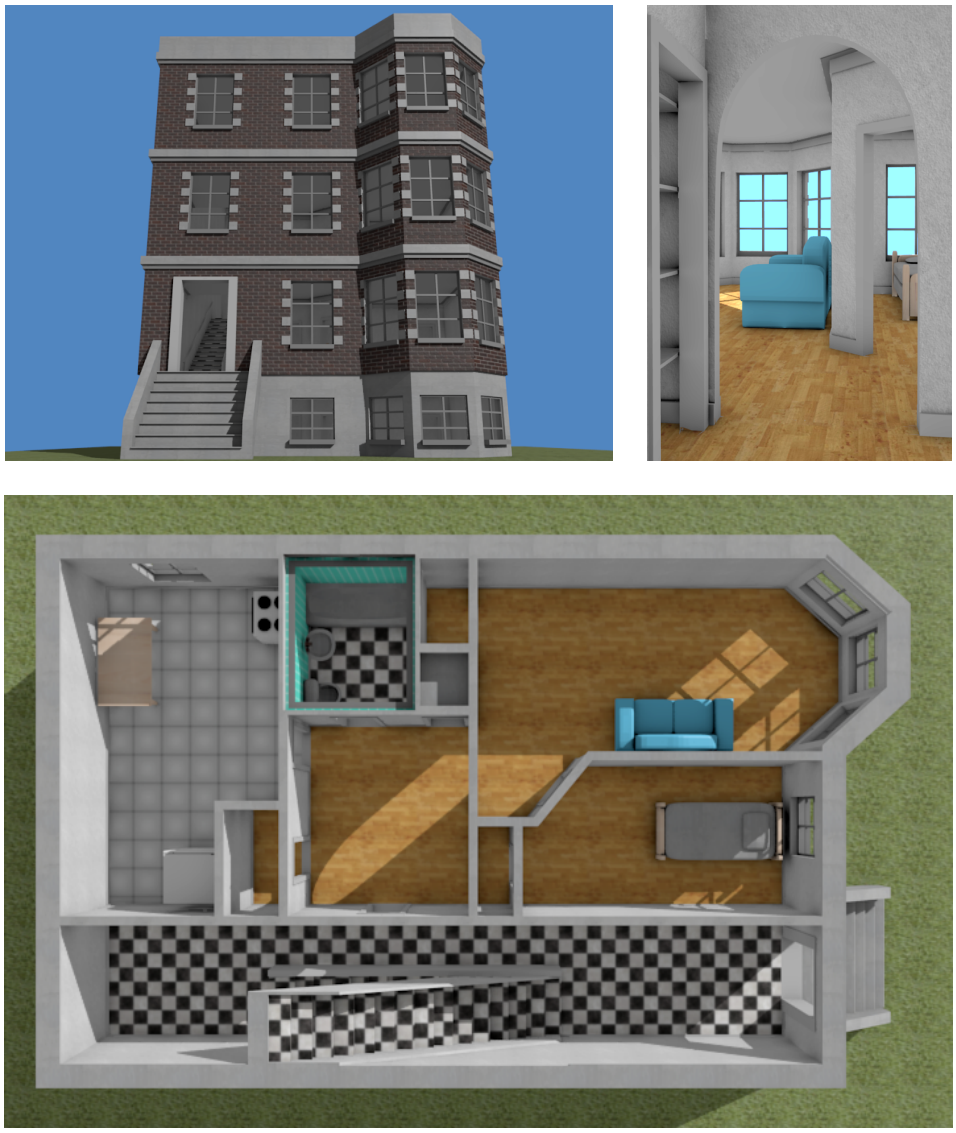


FIGURE 5.31 – Maison en rangée, vue de l'intérieur, et vue du premier étage.

FIGURE 5.32 – Un petit hôtel, immeuble à bureaux de style loft, et un complexe d'appartements de style blocs (inspiré de Habitat 67).

dizaines de lignes de code à environ 500 pour les édifices illustrés à la figure 5.30. C'est le cas le plus complexe, et c'est celui qui a pris le plus de temps à concevoir, soit environ une journée. Pour le reste des édifices, quelques heures tout au plus ont été nécessaires pour les mettre au

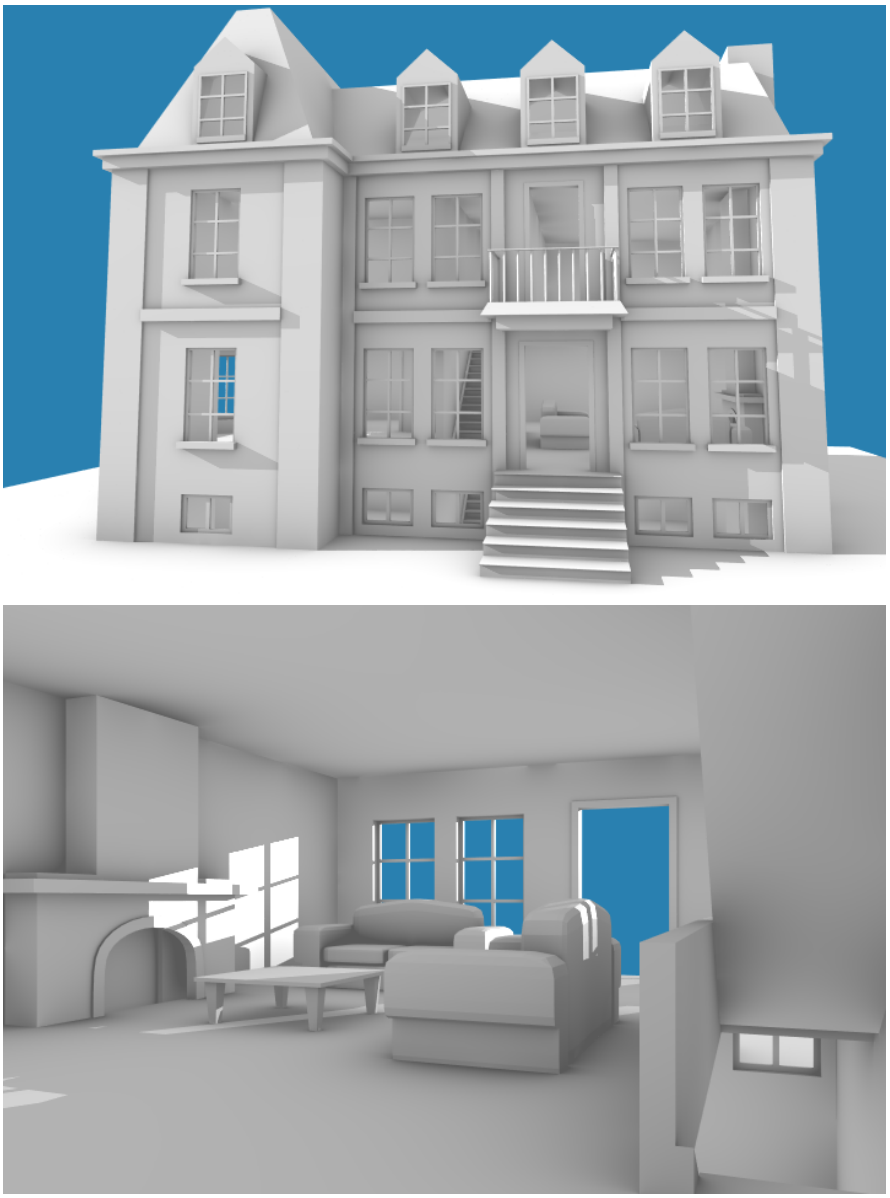
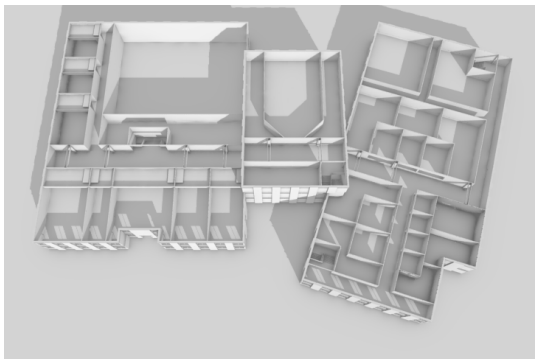


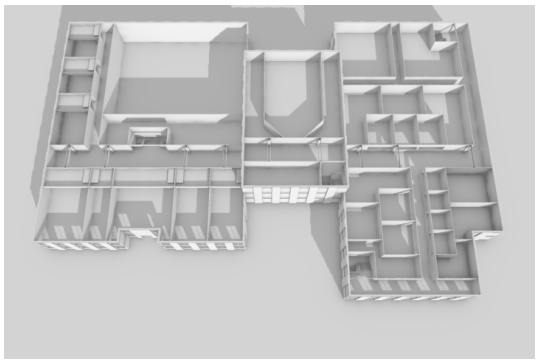
FIGURE 5.33 – Maison et vue de son intérieur.

point. Seulement quelques secondes à quelques minutes sont toutefois nécessaires pour apporter des modifications (e.g. changer le nombre d'étages, le type de portes ou de fenêtres, modifier la façade, etc.) tant et aussi longtemps que les modifications n'entraînent pas un changement majeur de la configuration du code.

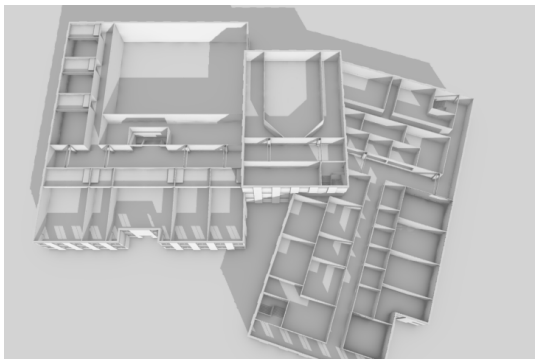
(a) Vue du sol



(b) Vue aérienne de (a) avec le toit coupé



(c) Même vue aérienne avec rotation de l'aile droite



(d) Même vue aérienne avec une translation de l'aile droite

FIGURE 5.34 – Vues et variations sur une école. L'aile droite est transformé par rotation et / ou par translation. Le nombre et la position des pièces s'adapte automatiquement à leurs nouvelles configurations.

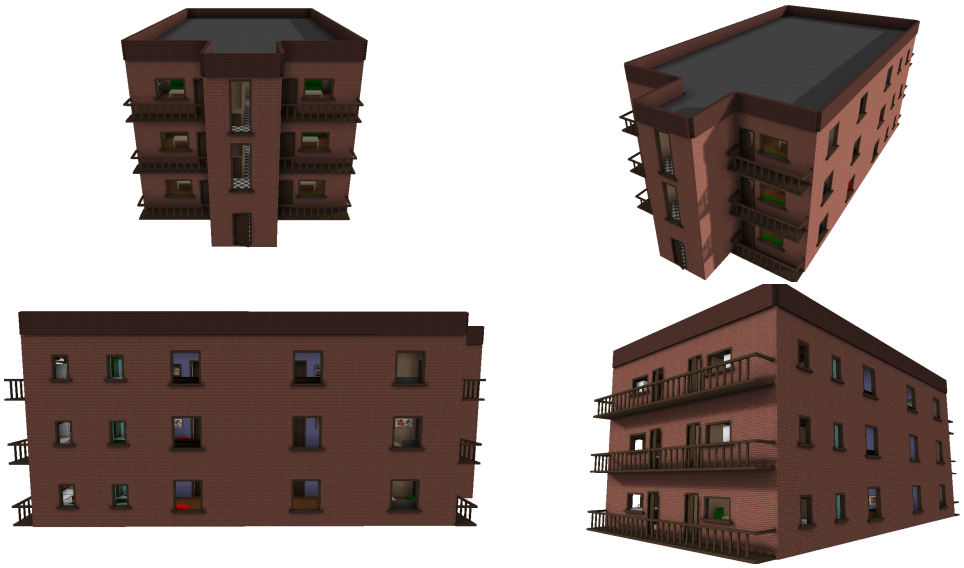


FIGURE 5.35 – Vues extérieures d'un petit immeuble d'appartements.

FIGURE 5.36 – Deux vues du deuxième étage.

FIGURE 5.37 – Vues de l'intérieur.

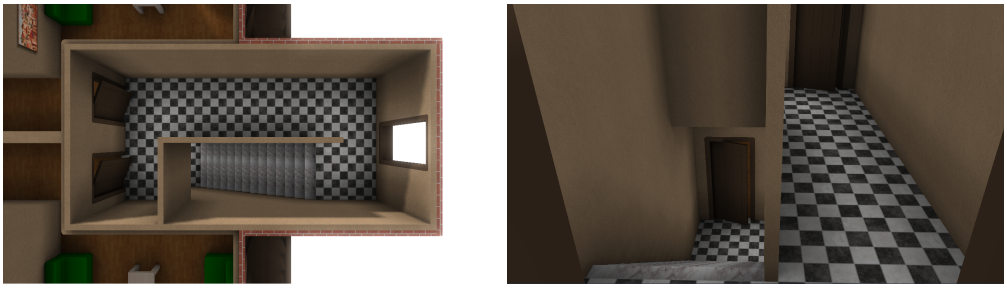


FIGURE 5.38 – Cage d'escalier vue du dessus et de l'intérieur.

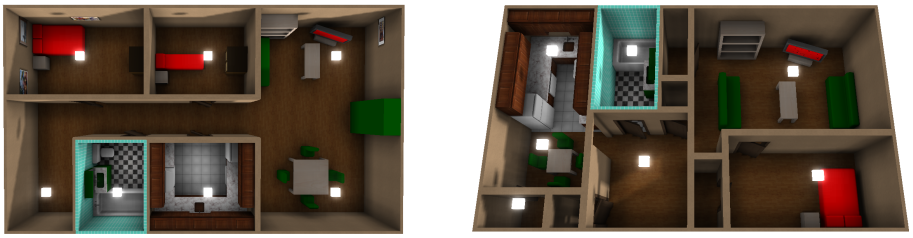


FIGURE 5.39 – Deux autres petits appartements.

Chapitre 6

Travaux futurs

Malgré tout le temps, les efforts et les bonnes intentions mis dans cette recherche, les résultats de ces travaux ne parviennent pas, bien sûr, à solutionner tous les problèmes de la modélisation procédurale. Des améliorations peuvent encore être apportées aussi bien au système géométrique par blocs qu'au système de modélisation par composants. Des extensions peuvent aussi être ajoutées pour augmenter leurs fonctionnalités. Finalement, l'introduction des nouveaux systèmes présentés dans ces pages ouvre la voie à de nouvelles possibilités à explorer. Les prochaines sections donneront une intuition sur les pistes à suivre.

6.1 Améliorations

Au chapitre 3, la section 3.8 fait mention de quelques améliorations possibles pour le système de blocs. La limitation principale de cette primitive est le type de connexion possible. L'idée de travailler avec une primitive volumique simple possédant une bonne paramétrisation de surface et pouvant se connecter simplement entre elles est très intéressante, cependant il faudrait éliminer l'alignement des connexions causé par la subdivision en grille des faces en sous-faces. Le système pourrait effectuer cette subdivision automatiquement selon le voisinage avec un motif adapté à chaque situation.

Une autre avenue à explorer est l'incorporation de blocs généralisés à des formes autres que des cuboïdes offrant un support pour des faces triangulaires. Les impacts sur la simplicité du système actuel versus les gains possibles en expressivité devraient être investigués.

L'utilisation de T-splines au lieu d'une surface de subdivision de Catmull-Clark pour

représenter la surface obtenue après fusion des blocs pourrait possiblement apporter certains gains. Entre autres, certains cas de distorsions de la paramétrisation pourraient être éliminés lorsque l'on doit effectuer une étape de subdivision pour se retrouver avec uniquement des quadrilatères et ainsi éliminer les sommets en T. Cette étape serait inutile avec les T-splines, mais toutefois, les conséquences d'avoir un maillage en T au lieu d'un maillage de quadrilatères ne sont pas très claires. Il est possible aussi que les T-splines offrent un meilleur contrôle de la surface, mais encore une fois, cela reste à déterminer.

Le système de modélisation par composants peut aussi bénéficier de quelques améliorations. C'est un système puissant qui peut simuler les grammaires et décrire bien des objets procéduralement, mais une ombre demeure : il peut être difficile d'approche pour les non-programmeurs (et même pour les programmeurs), et le code nécessaire reste un peu plus verbeux que nous l'aurions espéré. Une approche plus visuelle et concise pourrait apporter des gains notables au niveau de sa simplicité sans toutefois affecter son expressivité. C'est pourquoi une représentation d'un programme sous forme de graphe semble être une bonne idée à explorer. Aussi, la possibilité de générer des parties de code (ou de graphes) automatiquement soit par librairies ou par extraction de motifs à partir de photos simplifierait et accélérerait la création procédurale.

6.2 Extensions

Un langage (section 3.3) a été conçu pour générer des objets procéduralement par blocs, mais son utilisation nous a démontré qu'il peut parfois être long de produire le code voulu pour obtenir le résultat escompté. Nous pensons qu'il serait intéressant d'introduire un niveau d'abstraction entre les blocs et le système de modélisation par composants. Un système par chaînage, où chacun des noeuds représenteraient un bloc ou un groupe de blocs, offrirait une solution intéressante, similaire à la modélisation par *B-Mesh* [JLW10] qui utilise une description par arbre. Dans notre cas, on supporterait une définition par graphe, encore plus souple.

Au chapitre 3, il est aussi mentionné que puisque les blocs sont des primitives de haut niveau, il est possible de générer différentes représentations de plus bas niveau automatiquement selon notre choix. Dans ces travaux, seule la conversion en maillage triangulaire a été explorée. La voxélisation semble être une alternative intéressante. Elle permettrait l'exécution rapide d'opérations booléennes robustes avec une implémentation parallèle sur GPU assez aisée. Il est

envisageable que la conversion pourrait être assez rapide pour permettre sa transformation en temps réel selon le point de vue.

Du côté des composants, deux extensions semblent très pertinentes, soient ajouter de nouveaux types de connecteurs plus souples et introduire des opérations basées sur des techniques d'optimisation pour le placement de composants lors des connexions (utile entre autres pour l'ameublement) et pour le partitionnement complexe d'intérieurs. Dans ce dernier cas, un algorithme inspiré des travaux de Merrell *et al.* [MSK10] devrait être adéquat.

6.3 Travaux connexes

Des édifices, balcons, cages d'escalier, meubles et personnages ont été créés avec le système par composants (même si seulement les édifices ont été détaillés dans ce document), montrant sa polyvalence. Toutefois, deux types de modèles omniprésents n'ont pas été étudiés lors de ces travaux. La végétation serait particulièrement un bon candidat pour modéliser par connexions. En effet, le tronc et chaque branche pourraient être créés procéduralement ou à la main, et associés à un composant. Des régions spécifieraient les emplacements probables où des branches pourraient être connectées ainsi que leurs contraintes d'orientation. Ensuite, quelques déclarations contenant des opérations de connexion suffiraient pour assembler le tout en un arbre. La paramétrisation des blocs serait toute indiquée pour produire des écorces réalistes avec un bon facteur de compression (en utilisant du tuilage par exemple).

Dans le second cas, il n'est pas clair comment le système de modélisation procédurale par composants s'adapterait à la modélisation de terrains. Toutefois, l'utilisation de blocs comme éléments de base au lieu d'utiliser un champ de hauteur ouvrirait la voie à des terrains complexes avec surplombs. Mais pour donner de la souplesse de modélisation, il serait préférable que la subdivision d'une face en sous-faces puisse être hiérarchique.

La structure hiérarchique de composants obtenue par le processus de création couplée à la possibilité qu'offrent les attributs usager d'ajouter de la méta-information permet de créer des objets "intelligents", contenant plus que leur définition géométrique ou physique. Cette information pourrait être récupérée et utilisée à plusieurs fins. Par exemple, lors de la construction d'un édifice, une sémantique peut être assignée à ses différentes parties. Chacun des espaces intérieurs peut être caractérisé comme une pièce avec son type (chambre, salon, ...) ou comme lieu de transition entre pièces (couloir) ou étages (escalier, ascenseur). Cette information peut

ensuite être utilisée par des algorithmes de chemins (*path finding*) utilisés lors de simulations autant de déplacements de foules que de placements de câbles ou de tuyaux.

Des algorithmes de visibilité pourraient tirer profit de la structure hiérarchique des composants et des propriétés de construction pour accélérer le calcul de portails et de PVS (*potentially visible sets*). En effet, en principe avant l'ajout des éléments architecturaux (portes et fenêtres), chaque pièce est fermée et inaccessible des autres. Seules les portes, fenêtres et autres éléments servent de portails en ouvrant la voie.

Ce découpage de l'espace pourrait aussi servir à accélérer la construction et diminuer l'espace mémoire en ne construisant que les pièces visibles du point de vue d'un observateur. Il peut cependant être difficile d'évaluer quelles déclarations (requêtes et opérations) auraient des répercussions sur les éléments visibles.

Chapitre 7

Conclusion

Nous avons entrepris ces travaux pour rechercher et trouver des solutions aux coûts, principalement en temps, de la conception d'objets géométriques complexes. Tel qu'énoncé dans l'introduction, plusieurs domaines ont besoin de créer des scènes complexes et le temps consacré à cette fin est de plus en plus long vus les besoins grandissants pour un niveau de détail élevé et une augmentation de la qualité requise des modèles produits. Il n'est pas rare pour des groupes assez larges (une centaine de personnes) de dédier plusieurs mois à la conception de scènes complexes (e.g. des villes). C'est entre autres le cas pour l'industrie du jeu vidéo où les coûts et le temps de développement se situent principalement de ce côté.

Nous avons présenté le domaine de la modélisation géométrique selon deux catégories principales, soient la modélisation de bas niveau comprenant les différentes primitives de description géométrique et la modélisation de haut niveau décrivant les objets mais sans s'attarder à sa description géométrique. Cette dernière catégorie englobe toutes les méthodes procédurales telles les grammaires, la simulation et les procédés d'optimisation.

Au cours de ces travaux, nous avons proposé des solutions pour chacune de ces catégories. Au chapitre 3, nous avons introduit une nouvelle primitive de base, le bloc, comportant plusieurs caractéristiques importantes facilitant la modélisation procédurale. Cette primitive volumique possède implicitement une paramétrisation de surface simple adaptée à un contexte procédural. De plus, la définition de sa topologie se fait sans avoir à gérer d'opérations de bas niveau (comme les opérations d'Euler), ce qui encore une fois aide lors de la conception d'algorithmes de plus haut niveau. Contrairement, aux surfaces implicites (les *blobbies*), notre primitive garde une bonne maîtrise de sa surface soit par la modification de son maillage de contrôle en déplaçant les

sommets des blocs, soit par l'ajout d'une fonction de déplacements. Finalement, cette primitive peut aisément être convertie en maillage triangulaire ou tout autre représentation (e.g. voxels, points, etc.) pour des fins de rendu ou de simulation, et contrairement à ces représentations, elle est beaucoup plus compacte en espace mémoire. Nous avons aussi complémenté cette primitive d'un nouvel algorithme de CSG assez robuste en pratique pour être utilisé procéduralement sans intervention extérieure. Ce nouvel algorithme repose sur une intersection approximative entre triangles calculée à partir des distances relatives de chacun de leurs éléments (sommets, arêtes et faces).

Au chapitre 4, nous nous sommes attaqués aux problèmes de modélisation procédurale en proposant un nouveau système par composants inspiré par les grammaires, mais dont plusieurs limitations furent levées en changeant les règles strictes dans leur forme en déclarations composées de requêtes et d'opérations. L'accessibilité constante de tous les composants ainsi que leurs relations arborescentes offrent une plus grande flexibilité. Il est maintenant possible de réutiliser à maintes reprises un même composant pour effectuer différentes tâches. L'héritage des différents attributs usager ou système, est intuitif et permet la propagation des informations. L'ajout de régions avec contraintes ouvre la possibilité de combiner la modélisation de remplacement (provenant principalement des grammaires) à une modélisation par assemblage.

Le chapitre 5, bâtissant sur les deux chapitres précédents, valide la nouvelle expressivité offerte par notre système de modélisation par composants en présentant comment générer des édifices complets et cohérents avec intérieurs et extérieurs, un problème jusqu'à maintenant très peu abordé dû à sa complexité et à la limitation des techniques existantes. Les divers exemples présentés ainsi que les résultats obtenus démontrent le potentiel apporté par notre approche.

Pour conclure, nos travaux ont abouti en une nouvelle primitive géométrique et un nouveau système de modélisation procédurale qui ont respectivement été publiés sous forme d'articles [LHP11b, LHP11a]. Nous espérons maintenant que ces travaux seront repris et inspireront d'autres chercheurs afin de continuer de proposer des solutions aux problèmes de modélisation géométrique à grande échelle.

Bibliographie

[AC98] M. Agarwal et J. Cagan. ≪ A Blend of Different Tastes : The Language of Coffee Makers ≫. *Environment and Planning B : Planning and Design*, volume 25, pages 205–226, 1998.

[AD03] Bart Adams et Philip Dutré. ≪ Interactive boolean operations on surfel-bounded solids ≫. *ACM Transactions on Graphics*, volume 22, numéro 3, pages 651–656, juillet 2003.

[AK84] M. Aono et T. L. Kunii. ≪ Botanical Tree Image Generation ≫. *IEEE Computer Graphics and Applications*, volume 4, numéro 5, pages 10–34, mai 1984.

[AS10] Lars-Erik Andersson et Neil F. Stewart. *Introduction to the Mathematics of Subdivision Surfaces*. SIAM, 2010.

[BA05a] Daniel Bekins et Daniel G. Aliaga. ≪ Build-by-Number : Rearranging the Real World to Visualize Novel Architectural Spaces ≫. Dans *IEEE Visualization*, pages 143–150, 2005.

[BA05b] B. Beneš et X. Arriaga. ≪ Table Mountains by Virtual Erosion ≫. Dans *Natural Phenomena*, pages 33–39, 2005.

[Bar81] A. H. Barr. ≪ Superquadrics and angle-preserving transformations ≫. *IEEE Computer Graphics & Applications*, volume 1, numéro 1, pages 11–23, 1981.

[Bau72] Bruce G. Baumgart. ≪ Winged edge polyhedron representation. ≫. Rapport technique CS-TR-72-320, Stanford University, Department of Computer Science, octobre 1972.

[BBCW10] Adrien Bernhardt, Loïc Barthe, Marie-Paule Cani et Brian Wyvill. ≪ Implicit Blending Revisited ≫. *Computer Graphics Forum*, volume 29, numéro 2, pages 367–375, mai 2010.

[BF02] Bedrich Benes et Rafael Forsbach. ≪ Visual Simulation of Hydraulic Erosion ≫. Dans *WSCG*, pages 79–94, 2002.

[BF09] Gilbert Bernstein et Don Fussell. ≪ Fast, Exact, Linear Booleans ≫. *Computer Graphics Forum*, volume 28, numéro 5, pages 1269–1278, 2009.

[BGA04] Aurélien Barbier, Eric Galin et Samir Akkouche. ≪ Complex Skeletal Implicit Surfaces with Levels of Detail ≫. Dans *WSCG*, pages 35–42, 2004.

[Bin71] T. Binford. ≪ Visual Perception by Computer ≫. Dans *Proceedings, IEEE Conference on Systems and Control*, 1971.

[BL08] Brent Burley et Dylan Lacewell. « Ptex : Per-Face Texture Mapping for Production Rendering ». Dans *Eurographics Symposium on Rendering '08*, pages 1155–1164, 2008.

[Bli82] James F. Blinn. « A Generalization of Algebraic Surface Drawing ». *ACM Trans. Graph*, volume 1, numéro 3, pages 235–256, 1982.

[Blo88] Jules Bloomenthal. « Polygonization of implicit surfaces ». *Computer Aided Geometric Design*, volume 5, numéro 4, pages 341–355, novembre 1988.

[Blo97] Jules Bloomenthal, éditeur. *Introduction to Implicit Surfaces*. Morgan Kaufmann, 1997.

[BLZ00] Henning Biermann, Adi Levin et Denis Zorin. « Piecewise Smooth Subdivision Surfaces with Normal Control ». Dans *SIGGRAPH '00*, pages 113–120, 2000.

[Bot11] Algorithmic Botany. « TreePad ». algorithmicbotany.org/TreePadForums, 2011.

[BPK05] Stephan Bischoff, Darko Pavic et Leif Kobbelt. « Automatic restoration of polygon models ». *ACM Trans. Graph.*, volume 24, pages 1332–1352, octobre 2005.

[BR96] Raja (P.K.) Banerjee et Jarek R. Rossignac. « Topologically exact evaluation of polyhedra defined in CSG with loose primitives ». *Computer Graphics Forum*, volume 15, numéro 4, pages 205–217, 1996.

[Bra05] Benjamin Bradley. « Towards the Procedural Generation of Urban Building Interiors ». Mémoire de maîtrise, University of Hull, 2005.

[BTHB06] B. Benes, V. Tesinsky, J. Hornys et S. K. Bathia. « Hydraulic erosion ». *Computer Animation and Virtual Worlds*, volume 17, numéro 2, pages 99–108, mai 2006.

[BWS10] Martin Bokeloh, Michael Wand et Hans-Peter Seidel. « A connection between partial symmetry and inverse procedural modeling ». *ACM Trans. Graph.*, volume 29, numéro 4, pages 104 :1–104 :10, juillet 2010.

[Bé70] Pierre E. Bézier. *Emploi des machines a commande numerique*. Masson et Cie, 1970.

[Cag96] G Cagdas. « A shape grammar model for designing row-houses ». *Design Studies*, volume 17, numéro 1, pages 35–51, janvier 1996.

[Cas08] Ignacio Castaño. « Next-Generation Rendering of Subdivision Surfaces ». Dans *SIGGRAPH*, 2008.

[CC78] E. Catmull et J. Clark. « Recursively generated B-spline surfaces on arbitrary topological meshes ». *Computer-Aided Design*, volume 10, numéro 6, pages 350–355, septembre 1978.

[CDM+02] Barbara Cutler, Julie Dorsey, Leonard McMillan, Matthias Müller et Robert Jagnow. « A Procedural Approach to Authoring Solid Models ». Dans *SIGGRAPH '02*, ACM Transactions on Graphics, pages 302–311, juillet 2002.

[CdSF05] António Fernando Coelho, António Augusto de Sousa et Fernando Nunes Ferreira. « Modelling urban scenes for LBMS ». Dans *Proceeding of the Tenth International Conference on 3D Web Technology, Web3D 2005*, pages 37–46, 2005.

[CEW+08] Guoning Chen, Gregory Esch, Peter Wonka, Pascal Mueller et Eugene Zhang. « Interactive Procedural Street Modeling ». *ACM Trans. Graph.*, volume 27, numéro 3, pages Article 103 : 1–10, 2008.

[CH01] Marie-Paule Cani et Samuel Hornus. « Subdivision Curve Primitives : a New Solution for Interactive Implicit Modeling ». Dans *Shape Modeling International*, mai 2001.

[CH02] Nathan A. Carr et John C. Hart. « Meshed atlases for real-time procedural solid texturing ». *ACM Trans. Graph.*, volume 21, numéro 2, pages 106–131, avril 2002.

[CH07] Mao Chen et Wenqi Huang. « A two-level search algorithm for 2D rectangular packing problem ». *Comput. Ind. Eng.*, volume 53, numéro 1, pages 123–136, août 2007.

[CK83] B. Chiyokura et F. Kimura. « Design of Solids with Free-Form Surfaces ». *Computer Graphics*, volume 17, numéro 3, pages 289–293, 1983.

[CK10] Marcel Campen et Leif Kobbelt. « Exact and Robust (Self-)Intersections for Polygonal Meshes ». *Computer Graphics Forum*, volume 29, numéro 2, pages 397–406, 2010.

[CKS98] Swen Campagna, Leif Kobbelt et Hans-Peter Seidel. « Directed Edges — A Scalable Representation for Triangle Meshes ». *Journal of Graphics Tools*, volume 3, numéro 4, pages 1–12, 1998.

[CMR+99] Paolo Cignoni, Claudio Montani, Claudio Rocchini, Roberto Scopigno et Marco Tarini. « Preserving attribute values on simplified meshes by resampling detail textures ». *The Visual Computer*, volume 15, numéro 10, pages 519–539, 1999.

[CMS96] P. Cignoni, C. Montani et R. Scopigno. « Triangulating Convex Polygons Having T-Vertices ». *Journal of Graphics, GPU, and Game Tools*, volume 1, numéro 2, pages 1–4, 1996.

[CNLE09] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre et Elmar Eisemann. « GigaVoxels : Ray-Guided Streaming for Efficient and Detailed Voxel Rendering ». Dans *Proceedings of the symposium on Interactive 3D graphics and games (I3D '09)*, pages 15–22, Boston, MA, Etats-Unis, février 2009. ACM, ACM Press.

[Coo67] S. A. Coons. « Surfaces For Computer-Aided Design of Space Forms ». Rapport technique MIT/LCS/TR-41, Massachusetts Institute of Technology, juin 1967.

[CSHD03] Michael F. Cohen, Jonathan Shade, Stefan Hiller et Oliver Deussen. « Wang Tiles for image and texture generation ». Dans *SIGGRAPH '03*, pages 287–294, New York, NY, USA, 2003. ACM.

[Dac06] Carsten Dachsbacher. *Interactive Terrain Rendering : Towards Realism with Procedural Models and Graphics Hardware*. Thèse de doctorat, Universität Erlangen-Nürnberg, Technische Fakultät, 2006.

[dBVP+00] Paul W. de Bruin, Frans Vos, Frits H. Post, Sarah F. Frisken Gibson et Albert M. Vossepoel. « Improving Triangle Mesh Quality with SurfaceNets ». Dans *Medical*

Image Computing and Computer-Assisted Intervention - MICCAI 2000, volume 1935 de *Lecture Notes in Computer Science*, pages 804–813, 2000.

[DHL+98] Oliver Deussen, Pat Hanrahan, Bernd Lintermann, Radomír Mech, Matt Pharr et Przemyslaw Prusinkiewicz. « Realistic Modeling and Rendering of Plant Ecosystems ». Dans *SIGGRAPH '98*, pages 275–286, juillet 1998.

[DKT98] Tony DeRose, Michael Kass et Tien Truong. « Subdivision surfaces in character animation ». Dans *SIGGRAPH '98*, pages 85–94, 1998.

[DR47] Georges De Rham. « Un peu de mathématiques à propos d'une courbe plane ». *Elemente der Mathematik*, volume 2, numéro 5, pages 89–104, 1947.

[DS78] D. Doo et M. Sabin. « Behaviour of Recursive Division Surfaces Near Extraordinary Points ». *Computer-Aided Design*, volume 10, pages 356–360, septembre 1978.

[DSD+09] Carsten Dachsbacher, Philipp Slusallek, Tomas Davidovic, Thomas Engelhardt, Mike Phillips et Iliyan Georgiev. « 3D Rasterization – Unifying Rasterization and Ray Casting ». Rapport technique, VISUS/University Stuttgart and Saarland University, 2009.

[Dua05] José Pinto Duarte. « Towards the mass customization of housing : the grammar of Siza's houses at Malagueira ». *Environment and Planning B : Planning and Design*, volume 32, numéro 3, pages 347–380, mai 2005.

[DVS03] Carsten Dachsbacher, Christian Vogelgsang et Marc Stamminger. « Sequential point trees ». *ACM Trans. Graph*, volume 22, numéro 3, pages 657–662, 2003.

[EEKR99] H. Ehrig, G. Engels, H.-J. Kreowski et G. Rozenberg. *Handbook of graph grammars and computing by graph transformation : applications, languages, and tools*, volume 2. World Scientific Publishing, 1999.

[EMP+02] David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin et Steven Worley. *Texturing and Modeling : A Procedural Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[Eri04] Christer Ericson. *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology) (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[FBS05] D. Finkenzeller, J. Bender et A. Schmitt. « Feature-based decomposition of facades ». Dans *Proceedings of Virtual Concept 2005*, novembre 2005.

[FHP89] Deborah R. Fowler, James Hanan et Przemyslaw Prusinkiewicz. « Modelling spiral phyllotaxis ». *Computers and Graphics*, volume 13, numéro 3, pages 291–296, 1989.

[FJP04] Martin Fuhrer, Henrik Wann Jensen et Przemyslaw Prusinkiewicz. « Modeling Hairy Plants ». Dans *Pacific Conference on Computer Graphics and Applications*, pages 217–226, 2004.

[Fle87] U Flemming. « More than the sum of parts : the grammar of Queen Anne houses ». *Environment and Planning B : Planning and Design*, volume 14, numéro 3, pages 323–350, 1987.

[FMP92] Deborah R. Fowler, Hans Meinhardt et Pizemyslaw Prusinkiewicz. « Modeling Seashells ». Dans *SIGGRAPH '92*, pages 379–388, juillet 1992.

[GGH02] Xianfeng Gu, Steven J. Gortler et Hugues Hoppe. « Geometry Images ». Dans *SIGGRAPH '02*, pages 355–361, 2002.

[Gib98] Sarah F. Frisken Gibson. « Constrained Elastic Surface Nets : Generating Smooth Surfaces from Binary Segmented Data ». Dans *MICCAI '98*, pages 888–898, London, UK, 1998. Springer-Verlag.

[GIM03] Manuel N. Gamito, Edifcio Iscte et F. Kenton Musgrave. « Procedural Landscapes with Overhangs », novembre 2003.

[Gip75] J. Gips. *Shape Grammars and their Uses : Artificial Perception, Shape Generation, and Computer Aesthetics*. Birkhauser Verlag, 1975.

[GPMG10] Eric Galin, Adrien Peytavie, Nicolas Maréchal et Eric Guérin. « Procedural Generation of Roads ». *Computer Graphics Forum (Proceedings of Eurographics)*, volume 29, numéro 2, pages 429–438, 2010.

[GPRJ00] Sarah F. Frisken Gibson, Ronald N. Perry, Alyn P. Rockwood et Thouis R. Jones. « Adaptively sampled distance fields : a general representation of shape for computer graphics ». Dans *SIGGRAPH '00*, pages 249–254, 2000.

[GPSL03] Stefan Greuter, Jeremy Parker, Nigel Stewart et Geoff Leach. « Real-time procedural generation of 'pseudo infinite' cities ». Dans *GRAPHITE '03*, pages 87–94, 2003.

[GT96] Michael Gervautz et Christoph Traxler. « Representation and realistic rendering of natural phenomena with cyclic CSG graphs. ». *The Visual Computer*, volume 12, numéro 2, pages 62–74, 1996.

[Hav05] Sven Havemann. *Generative mesh modeling*. Thèse de doctorat, TU Braunschweig, 2005.

[HBW06] Evan Hahn, Prosenjit Bose et Anthony Whitehead. « Persistent realtime building interior generation ». Dans *SANDBOX '06 : Proc. ACM SIGGRAPH symposium on Videogames*, pages 179–186, 2006.

[HDMB07] Sebastien Horna, Guillaume Damiand, Daniel Meneveaux et Yves Bertrand. « Building 3D Indoor Scenes Topology from 2D Architectural Plans ». Dans *Conference on Computer Graphics Theory and Applications*, March 2007.

[Hei94] Jeff Heisserman. « Generative Geometric Design ». *IEEE Computer Graphics and Applications*, volume 14, numéro 2, pages 37–45, 1994.

[Hel01] Martin Held. « FIST : Fast Industrial-Strength Triangulation of Polygons ». *Algorithmica*, volume 30, numéro 4, pages 563–596, 2001.

[HF04] Sven Havemann et Dieter W. Fellner. « Generative Parametric Design of Gothic Window Tracery ». Dans *VAST*, pages 193–201, 2004.

[HHKF10] Bernhard Hohmann, Sven Havemann, Ulrich Krispel et Dieter Fellner. « A GML shape grammar for semantically enriched 3D building models ». *Computers and Graphics*, volume 34, numéro 4, pages 322 – 334, 2010.

[HPW92] Mark S. Hammel, Przemyslaw Prusinkiewicz et Brian Wyvill. « Modelling compound leaves using implicit contours ». Dans *Proceedings of the 1992 Western Computer Graphics Symposium*, pages 27–34, avril 1992.

[Hub90] Philip M. Hubbard. « Constructive Solid Geometry for Triangulated Polyhedra ». Rapport technique, Brown University, Providence, RI, USA, 1990.

[HWB95] Mikako Harada, Andrew Witkin et David Baraff. « Interactive physically-based manipulation of discrete/continuous models ». Dans *SIGGRAPH '95*, pages 199–208, 1995.

[HWC+05] Chien-Chang Ho, Fu-Che Wu, Bing-Yu Chen, Yung-Yu Chuangs et Ming Ouhyoungs. « Cubical Marching Squares : Adaptive Feature Preserving Surface Extraction from Volume Data ». *Computer Graphics Forum*, volume 24, numéro 3, pages 537–545, 2005.

[IdFF96] Roberto Ierusalimschy, Luiz Henrique de Figueiredo et Waldemar Celes Filho. « Lua — an Extensible Extension Language ». *Software : Practice and Experience*, volume 26, numéro 6, pages 635–652, 1996.

[IdFF05] Roberto Ierusalimschy, Luiz Henrique de Figueiredo et Waldemar Celes Filho. « The Implementation of Lua 5.0 ». *J. UCS*, volume 11, numéro 7, pages 1159–1176, 2005.

[JG98] J.H. Jo et J.S. Gero. « Space Layout Planning using an Evolutionary Approach ». *Artificial Intelligence in Engineering*, volume 12, pages 149–162, 1998.

[JLSW02] Tao Ju, Frank Losasso, Scott Schaefer et Joe D. Warren. « Dual contouring of hermite data ». Dans *SIGGRAPH '02*, pages 339–346, 2002.

[JLW10] Zhongping Ji, Ligang Liu et Yigang Wang. « B-Mesh : A Modeling System for Base Meshes of 3D Articulated Shapes ». *Computer Graphics Forum (Proc. Pacific Graphics)*, volume 29, numéro 7, pages 2169–2178, 2010.

[Kau86] Arie E. Kaufman. « Voxel-Based Architecture for Three-Dimensional Graphics ». Dans *IFIP Congress*, pages 361–366, 1986.

[KBSS01] Leif P. Kobbelt, Mario Botsch, Ulrich Schwanecke et Hans-Peter Seidel. « Feature sensitive surface extraction from volume data ». Dans *SIGGRAPH '01*, pages 57–66, 2001.

[KCF+02] J. Keysar, T. Culver, M. Foskey, S. Krishnan et D. Manocha. « ESOLID : A System for Exact Boundary Evaluation ». Dans *Proceedings of the Symposium on Modeling and Application (SMA '02)*, pages 23–34, juin 2002.

[KCODL06] Johannes Kopf, Daniel Cohen-Or, Oliver Deussen et Dani Lischinski. « Recursive Wang tiles for real-time blue noise ». *ACM Trans. Graph.*, volume 25, numéro 3, pages 509–518, 2006.

[KE81] H Koning et J Eizenberg. « The language of the prairie : Frank Lloyd Wright's prairie houses ». *Environment and Planning B : Planning and Design*, volume 8, pages 295–323, 1981.

[KMDZ09] Denis Kovacs, Jason Mitchell, Shanon Drone et Denis Zorin. « Real-time creased approximate subdivision surfaces ». Dans *Proceedings of the symposium on Interactive 3D graphics and games (I3D '09)*, pages 155–160, New York, NY, USA, 2009. ACM.

[KMG+01] Shankar Krishnan, Dinesh Manocha, M. Gopi, Tim Culver et John Keyser. « BOOLE : A Boundary Evaluation System for Boolean Combinations of Sculptured Solids ». *Int. J. Comput. Geometry Appl*, volume 11, numéro 1, pages 105–144, 2001.

[Knu68] D. Knuth. « Semantics of context-free languages ». *Mathematical Systems Theory*, volume 2, numéro 2, pages 127–145, 1968.

[Kob00] Leif Kobbelt. « Sqrt(3)-Subdivision ». Dans *SIGGRAPH '00*, pages 103–112, juillet 2000.

[KPK10] Lars Krecklau, Darko Pavic et Leif Kobbelt. « Generalized Use of Non-Terminal Symbols for Procedural Modeling ». *Computer Graphics Forum*, volume 29, numéro 8, pages 2291—2303, 2010.

[KS86] Arie Kaufman et Eyal Shimony. « 3D scan-conversion algorithms for voxel-based graphics ». Dans *Proceedings of 1986 Workshop on Interactive 3D Graphics*, pages 45–75, octobre 1986.

[KVLM03] Young J. Kim, Gokul Varadhan, Ming C. Lin et Dinesh Manocha. « Fast swept volume approximation of complex polyhedral models ». Dans *Symposium on Solid Modeling and Applications*, pages 11–22, 2003.

[Lai07] Shuhua Lai. « Robust and Error Controllable Boolean Operations on Free-Form Solids Represented by Catmull-Clark Subdivision Surfaces ». *Computer-aided Design*, pages 487–496, 2007.

[LBD10] Cyril Leconte, Hichem Barki et Florent Dupont. « Exact and Efficient Booleans for Polyhedra ». Rapport technique RR-LIRIS-2010-018, LIRIS UMR 5205 CNRS/INSA de Lyon/Université Claude Bernard Lyon 1/Université Lumière Lyon 2/École Centrale de Lyon, octobre 2010.

[LC87] William E. Lorensen et Harvey E. Cline. « Marching cubes : A high resolution 3D surface construction algorithm ». Dans *SIGGRAPH '87*, pages 163–169, 1987.

[LC05] Shuhua Lai et Fuhua Cheng. « Adaptive Rendering of Catmull-Clark Subdivision Surfaces ». Dans *CAD-CG '05 : Proc. Intl. Conf. Computer Aided Design and Computer Graphics*, pages 125–132, 2005.

[LC06a] Shuhua Lai et Fuhua Cheng. « Voxelization of Free-Form Solids using Catmull-Clark Subdivision Surfaces ». Dans *GMP'06 : Lecture Notes in Computer Science*, pages 595–601. Springer, 2006.

[LC06b] Yanbing Liu et Duanbing Chen. « A Novel Greedy Computing Algorithm for Rectangle Packing Problems ». *International Journal of Computer Science and Network Security*, volume 6, numéro 4, pages 78–81, 2006.

[LD03] R. G. Laycock et A. M. Day. « Automatically generating large urban environments based on the footprint data of buildings ». Dans *Symposium on Solid Modeling and Applications*, pages 346–351, 2003.

[LD06] Ares Lagae et Philip Dutré. « Poisson Sphere Distributions ». Dans L. Kobbelt, T. Kuhlen, T. Aach et R. Westermann, éditeurs. *Vision, Modeling, and Visualization 2006*, pages 373–379, Berlin, novembre 2006. Akademische Verlagsgesellschaft Aka GmbH.

[LDS08] Mikola Lysenko, Roshan D'souza et Ching-kuan Shene. « Improved Binary Space Partition merging ». *Computer-aided Design*, volume 40, pages 1113–1120, 2008.

[Lew87] J. P. Lewis. « Generalized stochastic subdivision ». *ACM Transactions on Graphics*, volume 6, numéro 3, pages 167–190, juillet 1987.

[LG06] Mathieu Larive et Veronique Gaildrat. « Wall grammar for building generation ». Dans *GRAPHITE '06*, pages 429–437, 2006.

[LHP11a] Luc Leblanc, Jocelyn Houle et Pierre Poulin. « Component-based Modeling of Complete Buildings ». Dans *Graphics Interface 2011*, pages 87–94, mai 2011.

[LHP11b] Luc Leblanc, Jocelyn Houle et Pierre Poulin. « Modeling with blocks ». *The Visual Computer (Proc. Computer Graphics International 2011)*, volume 27, pages 555–563, juin 2011.

[Lin68] A. Lindenmayer. « Mathematical Models for Cellular Interactions in Development, I & II ». *Journal of Theoretical Biology*, volume 18, pages 280–315, 1968.

[LK10] Samuli Laine et Tero Karras. « Efficient Sparse Voxel Octrees ». Dans *Proceedings of the symposium on Interactive 3D Graphics and Games (I3D '10)*, pages 55–63. ACM Press, 2010.

[LLVT03] Thomas Lewiner, Hélio Lopes, Antônio Wilson Vieira et Geovan Tavares. « Efficient Implementation of Marching Cubes' Cases with Topological Guarantees ». *Journal of Graphics Tools*, volume 8, numéro 2, pages 1–15, 2003.

[LLWQ10] Bo Li, Xin Li, Kexiang Wang et Hong Qin. « Generalized PolyCube Trivariate Splines ». Dans *Shape Modeling International*, pages 261–265, 2010.

[Loo87] C.T. Loop. « Smooth subdivision surfaces based on triangles ». Mémoire de maîtrise, Department of Mathematics, University of Utah, 1987.

[LP02] Brendan Lane et Przemyslaw Prusinkiewicz. « Generating Spatial Distributions for Multilevel Models of Plant Communities ». Dans *Proc. Graphics Interface*, pages 69–80, mai 2002.

[LPRM02] Bruno Lévy, Sylvain Petitjean, Nicolas Ray et Jérôme Maillot. « Least squares conformal maps for automatic texture atlas generation ». *ACM Trans. Graph*, volume 21, numéro 3, pages 362–371, 2002.

[LS08] Charles Loop et Scott Schaefer. « Approximating Catmull-Clark subdivision surfaces with bicubic patches ». *ACM Trans. Graph.*, volume 27, pages 8 :1–8 :11, mars 2008.

[LSNC09] Charles Loop, Scott Schaefer, Tianyun Ni et Ignacio Castano. « Approximating subdivision surfaces with Gregory patches for hardware tessellation ». *ACM Trans. Graph.*, volume 28, pages 151 :1–151 :9, décembre 2009.

[LTH86] D. H. Laidlaw, W. B. Trumbore et J. F. Hughes. « Constructive Solid Geometry for Polyhedral Objects ». Dans *SIGGRAPH '86*, pages 161–170, août 1986.

[LWW08] Markus Lipp, Peter Wonka et Michael Wimmer. « Interactive visual editing of grammars for procedural architecture ». Dans *SIGGRAPH '08*, pages 102 :1–10, 2008.

[LWWF03] Thomas Lechner, Ben Watson, Uri Wilensky et Martin Felsen. « Procedural city modeling ». Dans *1st Midwestern Graphics Conference*, 2003.

[Man78] B. Mandelbrot. *Fractals : Form, chance, and dimension*. W.H. Freeman and Company, 1978.

[Man88] Martti Mantyla. *An Introduction to Solid Modeling*. Computer Science Press, Inc., New York, NY, USA, 1988.

[Mer07] Paul Merrell. « Example-based model synthesis ». Dans *Proceedings of the symposium on Interactive 3D graphics and games (I3D '07)*, pages 105–112, New York, NY, USA, 2007. ACM.

[MKM89] F. Kenton Musgrave, Craig E. Kolb et Robert S. Mace. « The Synthesis and Rendering of Eroded Fractal Terrains ». Dans *SIGGRAPH '89*, pages 41–50, juillet 1989.

[MM09] Paul Merrell et Dinesh Manocha. « Constraint-based model synthesis ». Dans *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*, SPM '09, pages 101–111, New York, NY, USA, 2009. ACM.

[MP96] Radomí Měch et Przemyslaw Prusinkiewicz. « Visual Models of Plants Interacting With Their Environment ». Dans *SIGGRAPH '96*, pages 397–410, août 1996.

[MPB05] Jean-Eudes Marvie, Julien Perret et Kadi Bouatouch. « The FL-system : a functional L-system for procedural geometric modeling ». *The Visual Computer*, volume 21, numéro 5, pages 329–339, 2005.

[MSK10] Paul Merrell, Eric Schkufza et Vladlen Koltun. « Computer-generated residential building layouts ». *ACM Trans. Graph.*, volume 29, pages 181 :1–181 :12, décembre 2010.

[MSL+11] Paul Merrell, Eric Schkufza, Zeyang Li, Maneesh Agrawala et Vladlen Koltun. « Interactive Furniture Layout Using Interior Design Guidelines ». *SIGGRAPH '11*, août 2011. To appear.

[MSS94] Claudio Montani, Riccardo Scateni et Roberto Scopigno. « A modified look-up table for implicit disambiguation of Marching Cubes ». *The Visual Computer*, volume 10, numéro 6, pages 353–355, 1994.

[MWH+06] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer et Luc Van Gool. « Procedural modeling of buildings ». Dans *SIGGRAPH '06*, pages 614–623, juillet 2006.

[MZWG07] Pascal Müller, Gang Zeng, Peter Wonka et Luc Van Gool. « Image-based procedural modeling of facades ». Dans *SIGGRAPH '07*, pages 85 :1–9, 2007.

[Mö97] Tomas Möller. « A Fast Triangle-Triangle Intersection Test ». *journal of graphics, gpu, and game tools*, volume 2, numéro 2, pages 25–30, 1997.

[NA03] Andy Nealen et Marc Alexa. « Hybrid Texture Synthesis ». Dans *Proceedings of the 14th Eurographics workshop on Rendering*, pages 097–105, 2003.

[NAT90] Bruce Naylor, John Amanatides et William Thibault. « Merging BSP trees yields polyhedral set operations ». Dans *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '90, pages 115–124, New York, NY, USA, 1990. ACM.

[NSS+00] Akira Nagao, Takashi Sawa, Yuji Shigehiro, Isao Shirakawa et Takashi Kambe. « A new approach to rectangle-packing ». *Electronics and Communications in Japan (Part III : Fundamental Electronic Science)*, volume 83, numéro 12, pages 94–104, 2000.

[NSZ+10] Liangliang Nan, Andrei Sharf, Hao Zhang, Daniel Cohen-Or et Baoquan Chen. « SmartBoxes for interactive urban reconstruction ». Dans *SIGGRAPH '10*, pages 93 :1–10, 2010.

[NYM+08] T. Ni, Y. Yeo, A. Myles, V. Goel et J. Peters. « GPU smoothing of quad meshes ». Dans *SMI'08 : IEEE Intl. Conf. on Shape Modeling and Applications*, pages 3–9, juin 2008.

[PASS95] A. Pasko, V. Adzhiev, A. Sourin et V. Savchenko. « Function representation in geometric modeling : concepts, implementation and applications ». *The Visual Computer*, volume 11, pages 429–446, 1995.

[PB00] Dan Piponi et George D. Borshukov. « Seamless Texture Mapping of Subdivision Surfaces by Model Pelting and Texture Blending ». Dans *SIGGRAPH '00*, pages 471–478, juillet 2000.

[PCK10] Darko Pavic, Marcel Campen et Leif Kobbelt. « Hybrid Booleans ». *Computer Graphics Forum*, volume 29, numéro 1, pages 75–87, 2010.

[PF01] Ronald N. Perry et Sarah F. Frisken. « Kizamu : a system for sculpting digital characters ». Dans *SIGGRAPH '01*, pages 47–56, 2001.

[PGK02] Mark Pauly, Markus Gross et Leif P. Kobbelt. « Efficient simplification of point-sampled surfaces ». Dans *VIS '02 : Proceedings of the conference on Visualization '02*, pages 163–170, 2002.

[PGMG09a] Adrien Peytavie, Eric Galin, Stephane Merillou et Jerome Grosjean. « Arches : a Framework for Modeling Complex Terrains ». *Computer Graphics Forum (Proceedings of Eurographics)*, volume 28, numéro 2, pages 457–467, 2009.

[PGMG09b] Adrien Peytavie, Eric Galin, Stephane Merillou et Jerome Grosjean. « Procedural Generation of Rock Piles Using Aperiodic Tiling ». *Computer Graphics Forum (Proceedings of Pacific Graphics)*, volume 28, numéro 7, pages 1801–1810, 2009.

[PH93] Przemyslaw Prusinkiewicz et Mark Hammel. « A fractal model of mountains and rivers ». Dans *Graphics Interface '93*, pages 174–180, mai 1993.

[PHHM97] Przemyslaw Prusinkiewicz, Mark Hammel, Jim Hanan et Radomir Měch. « L-Systems : From the Theory to Visual Models of Plants ». Dans *Plants to Ecosystems*, volume 1 de *Advances in Computational Life Sciences*, chapitre 1, pages 1–27. CSIRO Publishing, février 1997.

[PHM93] Przemyslaw Prusinkiewicz, Mark Hammel et Eric Mjolsness. « Animation of plant development ». Dans *SIGGRAPH '93*, pages 351–360, 1993.

[Pie91] Les Piegl. « On NURBS : A Survey ». *IEEE Computer Graphics and Applications*, volume 11, numéro 1, pages 55–71, janvier 1991.

[PIX11] PIXOLOGIC. « ZBrush ». www.pixologic.com, 2011.

[PJM94] Przemyslaw Prusinkiewicz, Mark James et Radomiŕ Měch. « Synthetic Topiary ». Dans *SIGGRAPH '94*, pages 351–358, juillet 1994.

[PKKG03] Mark Pauly, Richard Keiser, Leif P. Kobbelt et Markus Gross. « Shape modeling with point-sampled geometry ». *ACM Transactions on Graphics*, volume 22, numéro 3, pages 641–650, juillet 2003.

[PL90] Przemyslaw Prusinkiewicz et Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[PM01] Yoav I. H. Parish et Pascal Müller. « Procedural modeling of cities ». Dans *SIGGRAPH '01*, pages 301–308, 2001.

[Pro11] Procedural. « CityEngine ». www.procedural.com, 2011.

[PT95] Les Piegl et Wayne Tiller. *The NURBS book*. Springer-Verlag, 1995.

[PvBZG00] Hanspeter Pfister, Jeroen van Baar, Matthias Zwicker et Markus Gross. « Surfels : Surface Elements as Rendering Primitives ». Dans *SIGGRAPH '00*, pages 335–342, juillet 2000.

[RL00] Szymon Rusinkiewicz et Marc Levoy. « QSplat : A Multiresolution Point Rendering System for Large Meshes ». Dans *SIGGRAPH '00*, pages 343–352, juillet 2000.

[RV77] Aristides A. G. Requicha et H. B. Voelcker. « Constructive solid geometry ». Rapport technique 25, University of Rochester, novembre 1977.

[SD07] J. M. Smith et N. A. Dodgson. « A topologically robust algorithm for Boolean operations on polyhedral shapes using approximate arithmetic ». *Comput. Aided Des.*, volume 39, pages 149–163, février 2007.

[SG71] George Stiny et James Gips. « Shape Grammars and the Generative Specification of Painting and Sculpture ». Dans *IFIP Congress (2)*, pages 1460–1465, 1971.

[SGW06] Ryan Schmidt, Cindy Grimm et Brian Wyvill. « Interactive decal compositing with discrete exponential maps ». *ACM Trans. Graph.*, volume 25, numéro 3, pages 605–613, 2006.

[SH90] Tsuyoshi Saitoh et Mamoru Hosaka. « Interpolating Curve Networks with New Blending Patches ». Dans *Eurographics '90*, pages 137–146, septembre 1990.

[SH02] Alla Sheffer et John C. Hart. « Seamster : Inconspicuous Low-Distortion Texture Seam Layout ». Dans *Proceedings of the IEEE Visualization 2002 Conference (VIS '02)*, pages 291–298, octobre 2002.

[Sip96] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1996.

[SK92] John M. Snyder et James T. Kajiya. « Generative modeling : A symbolic system for geometric modeling ». Dans *SIGGRAPH '92*, pages 369–378, juillet 1992.

[SLL94] W. J. Schroeder, W. E. Lorensen et S. Linthicum. « Implicit Modeling of Swept Surfaces and Volumes ». Dans *Proceedings of the Conference on Visualization*, pages 40–45, octobre 1994.

[SLRLG03] Stephane Sanchez, Olivier Le Roux, Hervé Luga et Véronique Gaildrat. « Constraint-Based 3D-Object Layout using a Genetic Algorithm ». Dans *International Conference on Computer Graphics and Artificial Intelligence (3IA)*. Laboratoire XLIM - Université de Limoges, mai 2003.

[SR93] Jami J. Shah et Mary T. Rogers. « Assembly modeling as an extension of feature-based design ». *Research in Engineering Design*, volume 5, pages 218–237, 1993.

[Sti75] G. Stiny. *Pictorial and Formal Aspects of Shape and Shape Grammars*. Birkhauser, Basel, 1975.

[Sti80] G Stiny. « Kindergarten grammars : designing with Froebel's building gifts ». *Environment and Planning B : Planning and Design*, volume 7, pages 409–462, 1980.

[Sti82] G Stiny. « Spatial relations and grammars ». *Environment and Planning B*, volume 9, pages 113–114, 1982.

[SW78] G Stiny et Mitchell WJ. « The Palladian grammar ». *Environment and Planning B*, volume 5, pages 5–18, 1978.

[SW03] Scott Schaefer et Joe Warren. « Dual Contouring : The Secret Sauce ». Rapport technique, Rice University, février 2003.

[SWZ04] Scott Schaefer, Joe D. Warren et Denis Zorin. « Lofting Curve Networks using Subdivision Surfaces ». Dans *Symposium on Geometry Processing*, pages 105–116, 2004.

[SZBN03] Thomas W. Sederberg, Jianmin Zheng, Almaz Bakenov et Ahmad Nasri. « T-splines and T-NURCCs ». *ACM Trans. Graph.*, volume 22, pages 477–484, juillet 2003.

[SZK95] Renben Shu, Chen Zhou et Mohan S. Kankanhalli. « Adaptive marching cubes ». *The Visual Computer*, volume 11, numéro 4, pages 202–217, 1995.

[TBSdK09] Tim Tutenel, Rafael Bidarra, Ruben M. Smelik et Klaas Jan de Kraker. « Rule-based layout solving and its application to procedural interior generation ». Dans *3AMIGAS : Proc. CASA workshop on 3D Advanced Media in Gaming and Simulation*, pages 15–24, 2009.

[THCM04] M. Tarini, K. Hormann, P. Cignoni et C. Montani. « PolyCube-Maps ». *ACM Trans. Graph.*, volume 23, numéro 3, pages 853–860, août 2004.

[TN87] William C. Thibault et Bruce F. Naylor. « Set operations on polyhedra using binary space partitioning trees ». Dans *SIGGRAPH '87*, pages 153–162, New York, NY, USA, 1987. ACM.

[Vos85] R. P. Voss. « Random Fractal Forgeries ». Dans *Fundamental Algorithms for Computer Graphics*. Springer-Verlag New York, Inc., 1985.

[VZ01] Luiz Velho et Denis Zorin. « 4-8 Subdivision ». *Computer Aided Geometric Design*, volume 18, numéro 5, pages 397–427, 2001.

[Wan60] Hao Wang. « Proving Theorems by Pattern Recognition I ». *Commun. ACM*, volume 3, numéro 4, pages 220–234, 1960.

[WC02] Andrew S. Winter et Min Chen. « Image-Swept Volumes ». *Comput. Graph. Forum*, volume 21, numéro 3, pages 441–456, 2002.

[WGG99] Brian Wyvill, Andrew Guy et Eric Galin. « Extending the CSG Tree - Warping, Blending and Boolean Operations in an Implicit Surface Modeling System ». *Comput. Graph. Forum*, volume 18, numéro 2, pages 149–158, 1999.

[Whi06] R. Christopher White. « Effects Omelette, "King Kong" - The Building of 1933 New York City ». Dans *SIGGRAPH 2006 Sketches*. ACM, août 2006.

[WWSR03] Peter Wonka, Michael Wimmer, François X. Sillion et William Ribarsky. « Instant architecture ». *ACM Trans. Graph*, volume 22, numéro 3, pages 669–677, 2003.

[XFT+08] Jianxiong Xiao, Tian Fang, Ping Tan, Peng Zhao, Eyal Ofek et Long Quan. « Image-based façade modeling ». Dans *SIGGRAPH Asia '08*, pages 161 :1–10, 2008.

[XGH+11] Jiazhi Xia, Ismael Garcia, Ying He, Shi-Qing Xin et Gustavo Patow. « Editable Polycube Map for GPU-based Subdivision Surfaces ». Dans *I3D '11 : ACM Symposium on Interactive 3D Graphics and Games*, pages 151–158, 2011.

[YWR09] Xuetao Yin, Peter Wonka et Anshuman Razdan. « Generating 3D Building Models from Architectural Drawings : A Survey ». *IEEE Computer Graphics and Applications*, volume 29, numéro 1, pages 20–30, 2009.

[YYT+11] Lap-Fai Yu, Sai-Kit Yeung, Chi-Keung Tang, Demetri Terzopoulos, Tony F. Chan et Stanley Osher. « Make it Home : Automatic Optimization of Furniture Arrangement ». *SIGGRAPH '11*, août 2011. To appear.

[YZ01] Lexing Ying et Denis Zorin. « Nonmanifold Subdivision ». Dans *Proceedings Visualization 2001*, pages 325–331. IEEE Computer Society Technical Committee on Visualization and Graphics Executive Committee, 2001.

[ZHK04] Nan Zhang, Wei Hong et Arie E. Kaufman. « Dual Contouring with Topology-Preserving Simplification Using Enhanced Cell Representation ». Dans *IEEE Visualization*, pages 505–512, 2004.

[ZPKG02] Matthias Zwicker, Mark Pauly, Oliver Knoll et Markus Gross. « Pointshop 3D : an interactive system for point-based surface editing ». *ACM Trans. Graph.*, volume 21, numéro 3, pages 322–329, 2002.

[ZS00] Denis Zorin et Peter Schröder. « Subdivision for Modeling and Animation ». Dans *SIGGRAPH '00 Course*, juillet 2000.