Rapport de POM : Modélisation de terrains 3D avec surplombs

Claire BEGUIN, Charly BOLLINGER

Mai 2021

Résumé : Ce document présente l'étude de modélisation d'objets 3D par surface implicite. L'objectif de ce projet sera de modéliser des terrains réalistes avec caractéristiques 3D, tout en veillant à utiliser le moins d'espace mémoire.

Mots-clés : modélisation 3D, rendu, surface implicite, terrain.

1. Introduction

Ce travail a été réalisé dans le cadre d'une UE sous la tutelle d'Eric Galin, enseignant-chercheur du laboratoire de recherche LIRIS : Laboratoire InfoRmatique en Image et Système d'information. Compte tenu la situation actuelle, son soutien aura été entièrement dispensé à distance par l'intermédiaire de régulière conférence en ligne.

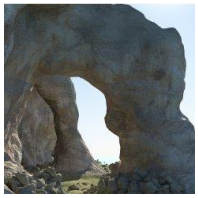

Figure 1 : Surplomb dans un terrain

Ce projet s'inscrit dans les projets de recherche sur la représentation de terrain par surface implicite. L'objectif étant de rendre des terrains réalistes, c'est-à-dire avec des surplombs, des marques d'érosion du sol ou encore des gravats, non-représentables via une fonction 2D $z = f(x, y)$. Tout cela sans passer par des structures de données trop volumineuses rendant impossible la génération à grande échelle. Un bon exemple de recherche réalisée par une équipe du LIRIS s'incluant dans ce projet est [PG19], cité dans notre cahier des charges.

Dans ce projet, nous nous intéresserons à l'élaboration de modèles compacts afin de définir des structures 3D pour réaliser des falaises, des arches ou encore des grottes. Pour cela, nous nous pencherons sur la construction d'un modèle de terrain par assemblage de primitives sous la forme d'un arbre. Aussi, nous implémenterons des algorithmes de synthèses pour la gestion des surfaces implicites.

2. Recherches

Novices en informatique graphique, notre démarche fut de passer par différentes étapes de recherches et d'applications de ces nouvelles connaissances afin de nous rapprocher petit à petit de notre objectif.

2.1. Terrains 2D

2.1.1. Génération procédurale

Avant de s'attaquer à l'aspect tridimensionnel du projet, nous nous sommes familiarisés avec une technique de génération procédurale de terrain : les champs de hauteur par somme de bruit. En effet, dans l'optique d'ajouter des éléments à un terrain, nous devions d'abord nous renseigner sur la création d'un terrain représentable uniquement de manière bidimensionnelle via une définition surfacique de type $z = f(x, y)$.

Figure 2 : Bruit de Perlin en deux dimensions

Pour ce faire, nous avons utilisé le bruit, noté $N : R^2 \to R$, fonction pseudo-aléatoire dont les détails sont pourtant de même taille (voir Figure 2). Grâce à une telle fonction, on peut associer une hauteur à un point $p \in R^2$, nous donnant ainsi ce qu'on appelle une carte de hauteurs.

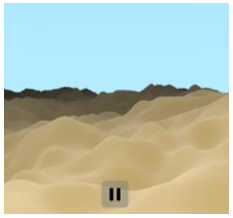

Figure 3 : Terrain généré par somme de bruit

Une simple addition de celles-ci aboutirait probablement à un terrain haut et relativement plat. Pour les champs de hauteurs par somme de bruit, on procède donc un peu différemment. Cette somme va servir à l'amélioration de la précision du terrain en ajoutant des détails à différentes échelles. On peut également faire varier la fréquence de ces détails de la même manière pour encore plus de réalisme. En ajoutant leur facteur respectif, f pour la fonction de fréquence et a pour la fonction d'échelle dite d'amplification, qui dépendent du niveau de précision actuel k dans la somme, on obtient alors $z(p) = \sum_{k=1}^{k=n} a(k) \cdot N(\frac{p}{f(k)})$. L'amplitude $a(k) = \frac{1}{\alpha^k}$ et la fréquence $f(k) = \varphi^k$ sont des fonctions décroissantes de k, où α et φ sont des hyperparamètres.

2.1.2. Primitives 2D

Pour continuer de mieux appréhender ce terrain, nous avons cherché à réaliser des premières primitives bidimensionnelles afin de transformer le terrain par leur assemblage. À ce moment-là, nous avions deux types de primitives en tête : des primitives modifiant le terrain autour d'un point, et d'autres qui le modifieraient dans sa globalité, et qui pourraient donc servir à d'autres objets.

Finalement, nous avons réalisé une fonction permettant d'aplanir le terrain dans un rayon donné, en fonction d'un premier rayon où le terrain serait plat, d'un second rayon pour interpoler la hauteur du terrain entre le plan et le reste, de la hauteur du plan, et du point au centre de la modification. Concernant la fonction de déformation, nous avons simplement fait des essais d'augmentation globale de la hauteur via des fonctions mathématiques comme $y = \cos(x)$ ou $y = x^2$ afin d'ajouter un effet de vague, par-dessus les irrégularités du terrain générées par le bruit.

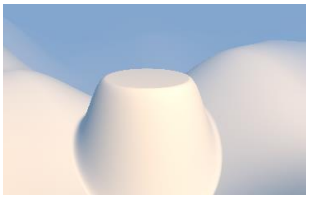

Figure 4 : Primitive de plan circulaire sur un terrain

2.2. SDF

Nous avons ensuite poussé les recherches cette fois-ci orientées 3D, le but étant d'abord de nous renseigner sur les techniques de rendu que nous pourrions utiliser. En effet, il existe deux types de méthodes pour afficher des surfaces implicites. Tout d'abord, nous présentons ce qu'est une surface implicite et l'intérêt de cette représentation des données.

Une surface implicite est une surface dans l'espace euclidien définie par une équation de la forme $f(x, y, z) = 0$ avec $f : R^3 \to R$. La surface d'un objet étant définie par une fonction, sa précision peut alors être infinie et l'objet peut ainsi paraître net. L'avantage d'une telle représentation d'objet est qu'elle est de par sa nature beaucoup plus légère à stocker qu'un maillage par exemple. Pour la visualiser, on a donc le choix entre des algorithmes de triangulation de surface ou bien de lancer de rayon. Dans les deux cas, on va chercher à calculer des points d'intersection avec la surface via l'utilisation de fonction de distance signée, *SDF*. Une telle fonction nous renverra un nombre positif si le point se trouve en-dehors de l'objet, respectivement négatif à l'intérieur. Une sphère de rayon r peut alors être définie comme $f(p) = |p| - r$, où $p \in R^3$.

2.2.1. Lancer de rayon

Nous nous sommes d'abord penchés sur le *ray marching,* et plus précisément le *sphere tracing,* pour son réalisme. En effet, cette méthode a pour avantage d'avoir une bonne restitution de la scène car elle ne décompose pas les objets en triangles ou surfaces élémentaires planes, ce qui permet d'avoir de réelles courbes.

Le lancer de rayon, ou *ray casting*, consiste littéralement à lancer des rayons depuis la caméra vers la scène. Ces rayons sont des lignes qui existent au même nombre que la résolution de l'image calculée. On peut représenter cela comme une grille placée devant la caméra de la taille de l'image (voir Figure 5). Ainsi, pour chaque pixel, une ligne va avancer jusqu'à rencontrer un objet. Lorsqu'elle en rencontre un, on sait alors qu'il est visible depuis la caméra et qu'il faut l'afficher. Sinon, on s'arrête évidemment à une certaine distance de vision. C'est ce qu'on appelle le *ray marching*. La scène étant entièrement définie par des *SDF*, il suffit d'évaluer en chaque point du rayon s'il a ou non rencontré un objet. Pour être sûr de ne pas passer à travers un objet et donc de voir au travers, il conviendrait d'avancer sur la ligne par petit pas, mais cela serait extrêmement coûteux en calcul. C'est ici que des améliorations interviennent comme l'algorithme de *sphere tracing* que nous avons utilisé.

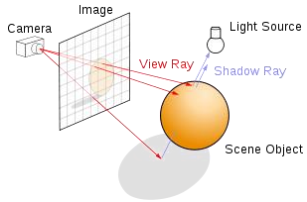

Figure 5 : Schéma du lancer de rayon

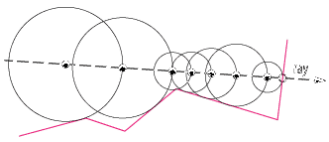

Figure 6 : Schéma du sphere tracing

Pour faire des pas plus grands et ainsi grandement accélérer la vitesse de rendu, le *sphere tracing* va pleinement tirer parti des *SDF*. Au lieu de faire un petit pas, l'algorithme va faire le plus grand pas où il sera sûr de ne pas croiser la surface. Pour le trouver, nous passons simplement par la distance du point à la surface, c'est-à-dire exactement ce que la *SDF* nous fournit. On obtient alors une distance de pas adaptative.

2.2.2. Cube marching

Pour la suite du projet, nous avons dû nous rabattre sur la seconde technique de visualisation de surface implicite, à savoir la triangulation de surface. En effet, le lancer de rayon étant assez coûteux en performance malgré l'utilisation du *sphere tracing*, nous avons cherché à implémenter un algorithme de *cube marching* pour que le rendu puisse être plus interactif sans trop en demander à nos machines.

La triangulation de surface permet le maillage de surface implicite dans un espace donné. Dans le cas du *cube marching*, l'espace doit respecter des contraintes afin de convenir à un découpage en cubes, c'est-à-dire que l'espace de maillage doit être un parallélépipède rectangle dont les côtés sont des multiples de la taille d'un cube. Effectivement, l'algorithme agit en itérant sur des cubes, que l'on peut aussi appeler *voxels*, superposés dans une région de la fonction implicite.

Si les 8 points du cube sont positifs via la *SDF*, ou négatifs, le cube est entièrement au-dessus, ou respectivement au-dessous, de la surface et aucun triangle n'est créé. Sinon, le cube chevauche la fonction et certains triangles sont générés. Puisque chaque point, ou vertex, peut être positif ou négatif, il y a techniquement 2^8 configurations possibles, mais la plupart d'entre elles étant équivalentes, il ne reste que 15 cas uniques (voir Figure 7). Ces différentes configurations précalculées permettent ainsi de déterminer les triangles à générer dans ce voxel.

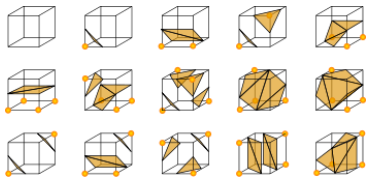

Figure 7 : Configurations possibles d'intersection entre un cube et la surface

Après itération sur tous les cubes de l'espace défini, le maillage peut être créé par l'union des différents triangles ajoutés à une liste durant l'algorithme. Pour augmenter la précision du maillage, il suffit ainsi d'augmenter le nombre de cube dans la région. Des techniques pour améliorer l'algorithme existent afin que la précision ne soit pas au détriment de la performance, dû au nombre élevé de triangles générés. Ainsi, on peut complexifier l'algorithme afin de rendre la résolution d'un cube adaptative en fonction de la complexité locale de la surface à rendre. Nous ne nous intéresserons pas plus à ces méthodes car elles ne permettent pas de concurrencer le lancer de rayon de plus nous souhaitions simplement avoir un rendu rapide.

2.3. Modèle par arbres de construction

Enfin, nous avons terminé nos recherches avec le modèle de notre terrain 3D. En d'autres termes, le but était de combiner les connaissances acquises jusqu'ici pour intégrer des objets tridimensionnels au terrain sans

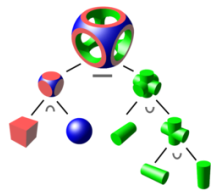

Figure 8 : Schéma d'arbre de construction

surplomb. Nous avons décidé de reprendre le concept d'arbre de construction de [PG19]. L'idée est d'assembler des primitives via des opérateurs, et de fusionner ces résultats avec le terrain.

Tout d'abord, il nous faut définir des primitives de base qui apparaitront aux feuilles de l'arbres. Elles peuvent prendre la forme d'objets simples, comme des sphères ou des cylindres, ou d'objets plus complexes qui sont la combinaison d'objets simples. C'est là qu'interviennent nos surfaces implicites définies avec des *SDF*, donc sous la forme compacte d'équation. Pour les formes complexes, le même principe de combinaison entre en jeu qu'avec le terrain. Les nœuds intermédiaires sont des opérateurs. On distinguera deux types d'opérateurs : les opérateurs unaires et binaires. Les binaires permettent l'association de plusieurs primitives en faisant leur union, la différence de l'un dans l'autre, ou encore l'intersection des deux (voir Figure 9). Les opérateurs unaires peuvent quant à eux avoir différentes utilités. On retrouve aussi bien les transformations spatiales, permettant de déplacer un objet ou le faire pivoter, que des transformations sur l'objet en lui-même pour l'agrandir ou le « vieillir » par exemple. Forts de tous ces nœuds, on peut finalement les fusionner avec le terrain de base, via un opérateur binaire, et obtenir en racine le terrain modifié avec des éléments 3D.

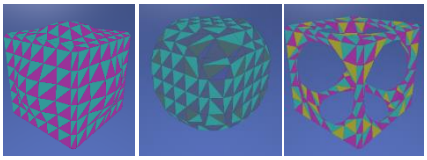

Figure 9 : Union, intersection et différence d'un cube avec une sphère

3. Réalisations

Pour chaque partie recherche que nous avons réalisé, une mise en pratique a été faite pour mieux visualiser et comprendre ces concepts. Nos travaux ont cependant été réalisé en deux grandes phases : une partie découverte avec le langage GLSL et le site *Shadertoy*, puis une partie plus approfondie en C++.

3.1. Lancer de rayon en GLSL

Pour cette partie, nous avons travaillé à partir d'un code de base entièrement en GLSL, à la fois sur *Shadertoy* et nos propres machines. Ce code nous a permis de visualiser un premier terrain via lancer de rayon. De ce fait, la résolution du terrain parait très nette car il n'est pas maillé en triangles. Le terrain généré est notre première implémentation d'une génération procédurale de champ de hauteur par somme de bruit.

Figure 10 : Terrain généré par somme de bruit

Ensuite, nous avons réalisé différentes fonctions afin de modifier le terrain, le but étant de comprendre comment modifier notre terrain 2D sans modifier directement la fonction de champ de hauteur. En d'autres termes, nous cherchions comment créer nos premières primitives.

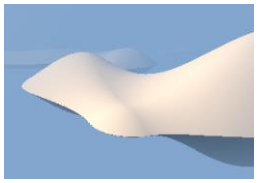

Figure 11 : Îles flottantes

Du côté des opérateurs, nous sommes restés sur des opérations simples pour commencer. Dans le but de réaliser des sortes d'îles flottantes, nous avons fait deux opérateurs unaires, l'un pour augmenter la hauteur du terrain de manière hyperbolique, et l'autre pour inverser un terrain. Avec ceux-là, il ne nous manquait plus qu'un opérateur binaire pour faire l'intersection de deux terrains hyperboliques, dont un à l'envers.

3.2. Marching cube en C++

Une fois ces concepts adoptés, nous souhaitions développer avec le langage C++ pour plus de flexibilité. Cependant, le lancer de rayon restant une méthode coûteuse en calcul, son adoption dans ce langage nous aurait freiné pour faire nos différents tests de primitives rapidement. Souhaitant tout de même continuer avec les *SDF* pour utiliser le modèle par arbre de construction qui est très léger en mémoire, il nous fallait opter pour la seconde solution de rendu de surface implicite : la triangulation de surface.

Pour débuter, un code fourni par notre encadrant nous permettait déjà d'afficher des maillages avec OpenGL. Nous avons alors implémenté un algorithme de *cube marching* afin de convertir nos fonctions de distance en maillage. Cet algorithme est l'adaptation d'une librairie par Magnus2 (voir [MM18]) dont nous avons modifié certains éléments. Il prend en compte la *SDF* dans une région du monde donnée, dont on fait en sorte qu'elle se redimensionne pour respecter la taille d'un cube, que l'on peut également choisir.

3.3. Modèle d'arbre en C++

3.3.1. La structure

Le plus gros du travail était ensuite d'ajouter notre arbre de construction avec *SDF*. Pour cette partie, nous avons décidé de travailler en orienté-objet. Chacun des nœuds de l'arbre peut être évalué par sa fonction de distance signée qui lui est propre. Partant de là, tous les types de nœuds dérivent de cette classe abstraite. D'un côté on retrouve les primitives qui ne sont que de simples nœuds implémentant à leur manière la *SDF*. De l'autre côté on a les opérateurs unaires et binaires qui héritent tous deux de classes abstraites respectives leur permettant de pointer vers un ou deux autres nœuds afin de relier les feuilles (primitives) et ainsi créer un arbre. Les opérateurs unaires nous permettent de transformer une *SDF* pour la déplacer et la tourner. Quant aux binaires, on s'en sert pour l'union, l'intersection ou la différence de deux nœuds. Comme les fonctions renvoient des distances aux objets, les opérateurs se traduisent naturellement par les équations $union(a,b) = \min(a,b)$, $intersection(a,b) = \max(a,b)$ et $difference(a,b) = \max(a,-b)$. Nous avons également créé une sous-classe aux opérateurs binaires afin de réaliser les mêmes opérations mais de manière plus douce avec un certain coefficient.

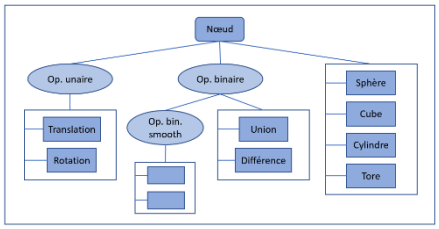

Figure 13 : Diagramme de classe des nœuds

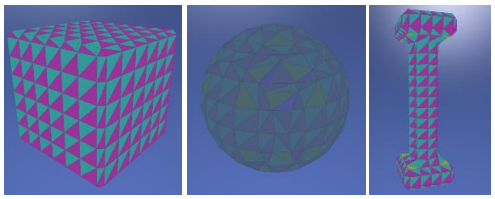

Figure 12 : Cube, sphère, et forme complexe

3.3.2. L'interface

Finalement, il ne manquait plus qu'à implémenter des primitives pour pouvoir sculpter notre terrain. La première chose à ajouter était évidemment un terrain généré à partir d'un champ de hauteur par somme de bruit comme en GLSL, mais cette fois sous la forme d'une primitive. Ensuite, nous avons implémenté des formes géométriques de base comme la sphère, le cube ou encore le cylindre. Bien sûr, ces formes n'apparaissent pas naturellement mais elles serviraient de base pour des formes plus complexes permettant de modeler le terrain.

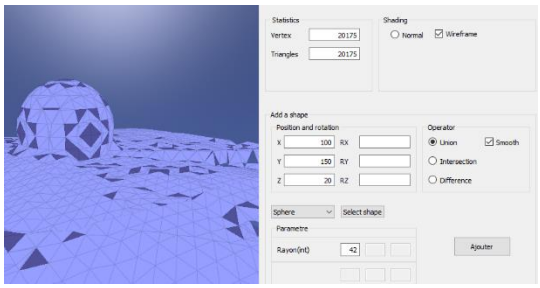

Figure 14 : Partie de l'interface du programme

Pris par le temps que nous a demandé l'assimilation des différents concepts et leur mise en application, nous n'avons malheureusement pas pu transposer ce travail sur un réel habillage de terrain avec des primitives complexes, ni ajouter ces primitives au terrain de manière procédurale. Cependant, nous avons réalisé une interface permettant de générer toutes les primitives et les ajouter au terrain. On peut alors régler leurs différents paramètres, comme le rayon d'une sphère par exemple, et les placer où l'on veut avec translation et rotations. Bien entendu, l'utilisateur peut décider de l'opérateur avec lequel il souhaite intégrer sa forme au terrain en sélectionnant l'opérateur parmi union, différence, intersection, smooth union, smooth différence et smooth intersection.

4. Conclusion

Ce sujet de recherche avait pour ambition de traiter la modélisation de terrains avec caractéristiques 3D en utilisant la modélisation d'objets 3D par surface implicite. Il nous a fallu dans un premier temps étudier le sujet afin de s'accoutumer aux différents concepts et algorithmes utilisés dans ce domaine de recherche. Nous avons commencé par réaliser un terrain à l'aide de la génération procédurale de terrain avec champs de hauteur par somme de bruit. Ensuite, nous nous sommes dirigés vers le fonctionnement d'une surface implicite et avons travaillé sur la réalisation de terrain par lancer de rayon puis par triangulation de surface. Enfin, pour ajouter des caractéristiques 3D complexes à notre terrain, nous avons utilisé un modèle d'arbre de construction. Celui-ci nous a permis d'ajouter différents types d'opération entre des primitives ou autres objets complexe, et notre terrain. Notre travail permet à présent de construire nos propres terrains avec toutes sortes de caractéristiques 3D complexes. Cependant, il se limite à un placement manuel de ces structures, et ces dernières ne sont pas forcément pratique à utiliser pour un placement à la main. On pourrait donc envisager d'ajouter ces primitives de manière procédurale, en détectant les endroits propices à certaines structures et en adaptant celle-ci au terrain. Cela pourrait être réalisé en conservant le champ de hauteur généré initialement et en cherchant les normales les plus abruptes, calculées itérativement en amont, pour y ajouter par exemple une arche ou creuser les falaises pour créer des surplombs.

Cette première expérience de recherche nous a permis de nous former dans un domaine qui nous était inconnu jusqu'alors. Nous avons également appris à allier recherche et développement d'algorithmes. Cela nous a permis de nous rendre compte que la première piste n'est pas toujours la bonne et qu'il est important de se remettre en question tout au long du projet pour parfois même recommencer en partant avec des algorithmes plus adaptés au problème donné.

Notre lien GitHub : https://github.com/charlyb01/AppMeshPOM

5. Références

[Gal15] Génération procédurale de mondes virtuels. Eric Galin, 2015.

[Mig04] Cours de lancer de rayons. Pascal Mignot, 2004.

[MM18] Mesh Reconstruction. Magnus2, MarsMero, 2018

[PG15] Terrain Modeling from Feature Primitives J-D. Génevaux, E. Galin, A. Peytavie, E. Guérin, C. Briquet, F. Grosbellet, B. Benes, 2015.

[PG19] Terrain Amplification with Implicit 3D Features. A. Paris, E. Galin, A. Peytavie, E. Guérin, J. Gain. *ACM Transactions on Graphics*, **38**(5), 2019.

[PG20] Segment Tracing Using Local Lipschitz Bounds. E. Galin, E. Guérin, A. Paris, A. Peytavie. *Segment Tracing Using Local Lipschitz Bounds.Computer Graphics Forum,* Wiley, press. 2020.

[Qui] Inigo Quilez. Inigo Quilez.

[Won16] Ray marching and signed distance functions. Jamie Wong, 2016.

6. Annexe

6.1. Diagramme de classe

Diagramme de classe dont la classe Node est le parent pour schématiser la structure choisie pour le modèle d'arbre de construction *(généré automatiquement avec Doxygen)*.