INF8702 - Infographie Avancée

Rapport Final

Rendu réaliste d'une surface océanique non bornée d'eau en temps réel

Par
Huyen Trang Dinh (1846776)
Jonathan Laroche (1924839)

Travail présenté à
Tarik Boukhalfi

École Polytechnique Montréal
décembre 2020

1. Introduction **3**

2. LOD **4**

2.1. Première implémentation: Implémentation naïve 4

2.1.1. Génération de la surface planaire 4

2.1.2. Tessellation 4

2.2. Deuxième implémentation: Quadtrees **5**

2.2.1. Génération de la surface planaire 5

2.2.1.1. Construction du quadtree 5

2.2.1.2. Génération des sommets 7

3. Bruit fractal et heightmap **8**

4. Défis rencontrés **9**

4.1. Craques 9

4.2. Distortions des normales 11

4.3. Difficultés techniques 12

5. Résultats finaux **13**

6. Références **15**

Annexe A **17**

Annexe B **19**

1. Introduction

La simulation d'eau en temps réel est un sujet intéressant en infographie vu qu'il démontre les limites des engins de rendu en temps réel. Avec la popularité croissante des jeux vidéo et de la réalité virtuelle, il est important pour les programmeurs de graphiques de trouver un compromis optimal entre des rendus réalistes et les contraintes de performance. De ce fait, une simulation de fluide basée sur les équations de Navier-Stokes sur une surface non bornée n'est pas optimale en termes de performances [7].

Un papier publié à la neuvième conférence CAD/CG en 2005 [14] présente un *framework* pour une simulation en temps réel d'une surface océanique à grande échelle.

Dans un premier temps, pour assurer une performance adéquate, la recherche propose un *framework* unique qui divise une seule grande surface en plusieurs petits blocs carrés à différentes résolutions. Le concept englobant cette idée est le LOD, ou *level of detail*, une forme de gestion du détail qui est également à la base du mipmapping. La technique de LOD employée dans [14] comprend l'utilisation de *quadtrees* en combinaison avec d'autres techniques d'optimisation GPU. Nous répliquons cette idée dans ce travail en créant également un maillage multirésolution afin de ne générer que les triangles nécessaires selon la distance de la caméra jusqu'aux sommets. Nous tirons avantage non seulement de l'utilisation de *quadtrees*, mais également des nuanceurs de tessellation du pipeline OpenGL.

Dans un deuxième temps, la recherche utilise une surface fractale enveloppée (*wrapped fractal surface)* afin de simuler un océan tranquille. Dans notre travail, nous reprenons cette idée en appliquant un bruit fractal OpenSimplex 3D sur un placage de hauteurs. Cette technique produit un effet chaotique sur une surface de manière continue, simulant le mouvement de la mer de façon suffisamment réaliste.

Dans un dernier temps, nous avons connu des difficultés au niveau d'artefacts visuels. Premièrement, nous avons observé des discontinuités (*cracks*) d'arêtes entre deux *patchs* de résolutions différentes. C'est un problème adressé par [14], mais sans solution fournie. Deuxièmement, lorsque nous illuminons notre scène, nous observons des distorsions avec les normales encore une fois entre les *patchs* de différentes résolutions. Nous détaillerons également dans l'avant-dernière section du rapport les diverses difficultés techniques auxquelles nous avons fait face. Enfin, nous relevons les résultats et les limites de notre solution dans la dernière section de ce rapport. La contribution de chacun des membres est également présentée à l'annexe B.

2. LOD

Le LOD (*level of detail*) réfère à la complexité géométrique d'un modèle 3D, qui se retrouve ici à être notre surface d'eau. Il est nécessaire d'implémenter plusieurs niveaux de LOD afin de produire un rendu réaliste sans gaspiller les ressources du GPU. Éviter ce gaspillage est primordial pour une application de rendu en temps réel. Un exemple de gaspillage se produit lorsque le modèle est surdétaillé alors qu'il est éloigné de la caméra, dans ce genre de cas des fragments rendus sont trop petits et n'amènent aucun changement visuel. À l'inverse, sous-détailler le modèle enlève l'impression de réalisme du rendu en laissant distinguer les différents fragments.

Nous voulons dynamiser le LOD selon la distance d'un *patch* par rapport à la caméra, de sorte à accorder moins de détails aux sections de surface océanique plus lointaine, et plus de détails aux sections les plus proches de la caméra.

Deux approches ont été considérées pour créer une surface avec un LOD dynamique. La première approche consiste à envoyer au GPU des *patchs* uniformes et d'implémenter entièrement le LOD dynamique dans le pipeline programmable de tessellation d'OpenGL 4. La seconde approche consiste à envoyer au GPU des *patchs* non uniformes dont les sommets sont calculés avec le CPU en utilisant un arbre quaternaire (*quadtree*). Cette seconde option utilise également le pipeline programmable de tessellation, mais le LOD n'en dépend pas entièrement.
Nous verrons en détail ces deux méthodes dans les prochaines sections.

2.1. Première implémentation: Implémentation naïve

Premièrement, notre recherche initiale a abouti à une interprétation naïve de ce que l'article de recherche de base [14] cherche à expliquer. Nous avions immédiatement pensé à utiliser nos connaissances de base en infographie pour implémenter les LODs dynamiques à l'aide de ce qu'offrent les nuanceurs de tessellation. La tessellation raffine les primitives en ajoutant des sommets et arêtes selon un niveau de subdivision spécifié pour des groupes de sommets que nous appelons des *patches*.

2.1.1. Génération de la surface planaire

Comme mentionné plus tôt, nous devions générer des *patchs* uniformes de 4 sommets devant être envoyés aux nuanceurs de tessellation. Nous avions donc créé une fonction permettant de *générer* des données de sommets et d'indices de connectivité de sorte à former un VAO contenant l'information requise pour rendre une grille de quads uniformes.

2.1.2. Tessellation

Nous passions au nuanceur de contrôle de tessellation une variable uniforme contenant la position de la caméra dans les coordonnées du monde. À l'aide de cette variable, nous pouvions

calculer la distance séparant la caméra et les sommets. Ensuite, nous affections les valeurs de niveau de tessellation (*outer* et *inner*) selon ces distances de façon catégorique. Nous arrivions effectivement à un rendu où les quads les plus proches de la caméra avaient des niveaux de tessellation plus élevés. Pour aider à la visualisation, nous avons représenté les quads en grillage afin de visualiser le niveau de détail aux différents endroits de la surface.Voici le résultat qui était obtenu:

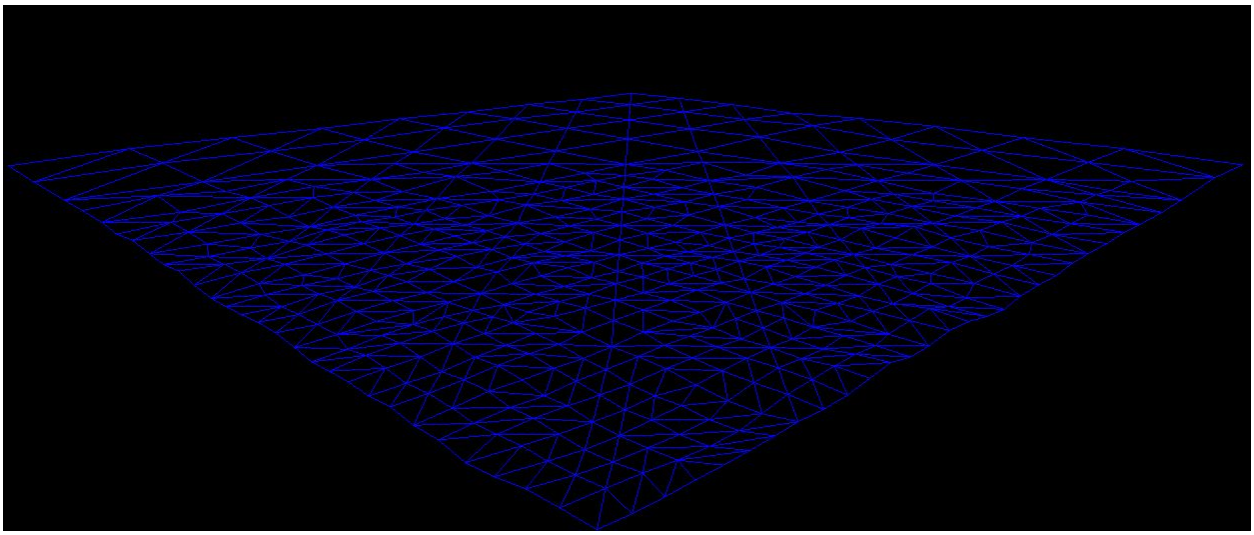


Figure 1. Résultat implémentation LOD naïve

2.2. Deuxième implémentation: Quadtrees

Deuxièmement, c'est lors d'une lecture de l'article de Victor Bush [1] que nous nous sommes rendus compte que nous n'étions pas sur la bonne voie. En effet, Bush énonce dans son article que notre approche naïve connaît ses limites dans la mesure où il y a une limite de tessellation par le GPU de 64 niveaux. Encore plus important, notre approche n'est pas extensible et optimale. En effet, si nous voulons achever l'océan non borné tel que décrit dans [14], il faut ajouter une plus grande quantité de *patchs* uniformes et donc, une plus grande quantité de sommets. Les auteurs de [14] ont choisi d'adopter un algorithme de LOD basé sur les avantages des *quadtrees*. L'idée est d'abord de représenter la surface océanique en entier par un seul quad et de subdiviser celle-ci en 4 autres primitives récursivement jusqu'à obtenir un niveau de détail satisfaisant. L'algorithme de subdivision décide selon la distance entre nœud étudié et la caméra s' il doit arrêter la subdivision.

2.2.1. Génération de la surface planaire

2.2.1.1. Construction du *quadtree*

La construction du *quadtree* s'effectue, comme suggéré par Victor Bush dans son article de LOD dynamique pour terrain [1], comme suit :

1. Définir le nœud racine de l'arbre comme étant le *patch* de base.
2. Comparer la position de la caméra dans le monde à l'origine du nœud. Si cette distance est assez petite, subdiviser le nœud en 4 enfants.
3. Répéter pour chaque nœud enfant jusqu'à ce que le niveau de subdivision maximal soit atteint, ou que la distance soit assez grande qu'un niveau de détail acceptable soit atteint.

L'arbre étant construit, nous pouvons le traverser et les feuilles de l'arbre peuvent être rendues selon leur niveau dans l'arbre. Nous avons utilisé l'implémentation Victor Bush lui-même, que nous pouvons retrouver [2]. Nous avons adapté son code pour l'utiliser dans le nôtre. La structure qui compose un noeud de l'arbre est la suivante:

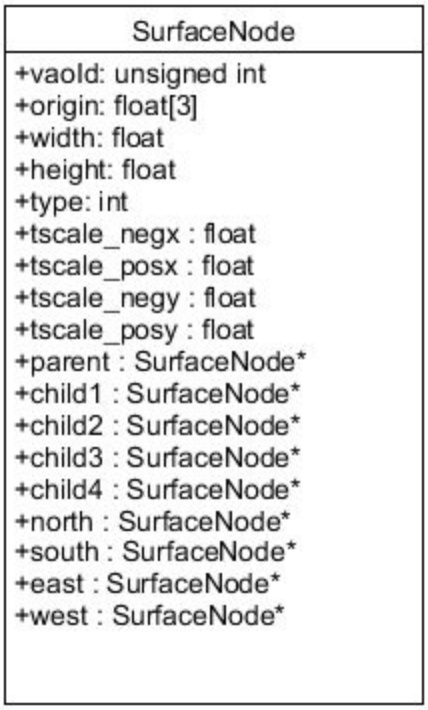


Figure 2. Représentation d'un noeud de la surface

Pour clarifier, la position et la taille du nœud est connu par les membres *origin*, *width* et *height*. Le *vaoId*, permet d'identifier le VAO permettant de rendre ce nœud. Le *type* représente qu'il est le n-ième enfant de son parent et il aura une valeur de 0 s'il est la racine. Les membres *tscale_x* représentent l'échelle de tessellation dans les 4 directions. *Parent* est un lien vers le parent du nœud et les membres *childX* sont des liens vers ses enfants, si ceux-ci sont à null cela signifie que le nœud est une feuille. Les membres *north*, *south*, *east* et *west* représentent les voisins directs du nœud. Effectivement, dans le domaine de l'infographie, les *quadtrees* s'avèrent être des structures très efficaces pour traverser une grande quantité de sommets. Ils représentent un système organisationnel de choix pour représenter une géométrie 2D comme la nôtre.

Des fonctions de bases permettent la recherche d'un nœud, l'ajout de nœuds et la création et le nettoyage d'arbre quaternaire. Le programme principal limite ces interactions aux fonctions surfaceInit(), createTree() et renderSea(). La fonction surfaceInit() génère un tampon pour les nœuds et les VAOs. Puis, un appel à la fonction createTree() permet de créer l'arbre selon une position de caméra. Finalement, renderSea() permet de traverser l'arbre afin de rendre les feuilles comme il sera expliqué dans la section suivante.

2.2.1.2. Génération des sommets

Pour en revenir au rendu sur le GPU, il faut générer des attributs de sommets et envoyer des VAO au GPU pour chacune des feuilles de manière cohérente. Ce processus est appelé la triangulation [6]. À l'aide de la structure que nous avons composée du code de Bush, nous suivons le processus suivant.

La fonction createTree() prend en paramètre l'origine de la surface océanique, la largeur et la longueur désirée de la surface ainsi que la position de la caméra actuelle de la caméra. Elle génère le tampon pointant au début d'un array de VAO de la taille de nœuds maximale. Ensuite, nous transmettons les positions initiales encadrant l'océan à l'aide des paramètres d'entrée. Puis, nous divisons de façon récursive ce nœud jusqu'à ce que les conditions de distance de caméra et de subdivision maximale soient respectées. C'est lors de la subdivision que nous créons un nœud et que nous lions les positions nouvellement subdivisées des sommets et des indices aux VBOs.

La fonction renderSea() effectue le dessin à partir de l'arbre généré, dans le nuanceur spécifié. Il va récursivement, en préordre, traverser l'arbre et rendre les feuilles de l'arbre uniquement. Cette étape consiste à indiquer au GPU l'information de quelle VAO il doit rendre tout en mettant à jour les variables uniformes pertinentes.

Ainsi, les changements qui ont été apportés au code de Bush consistent à l'ajout des VAOs, de quelques variables uniformes et l'ajout de fonctionnalité tel que l'affichage en grille de la surface. Voici un simple rendu des sommets résultant du *quadtree*:

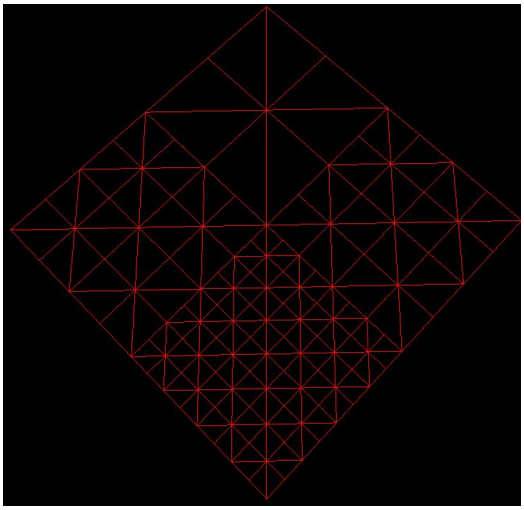


Figure 3. Rendu du *quadtree*

3. Bruit fractal et *heightmap*

Le bruit fractal est une composition de plusieurs couches de bruit de Perlin à différentes octaves [9]. Plus ce nombre de couches est grand, plus il y a du détail dans le bruit. Conformément à [14], nous allons ainsi utiliser le bruit fractal comme *heightmap*. Le *heightmap* est simplement un placage de déplacement unidirectionnel. Chaque sommet sera ainsi déplacé selon sa normale. Puisque notre surface est plate, nous allons simplement faire varier la position *y* (le *up vector* de notre monde) de chaque sommet avant ses transformations de visualisation et de projection. Plutôt que d'utiliser un bruit de Perlin comme prescrit dans [14], nous utilisons la version *open source* de son bruit successeur Simplex, le OpenSimplex. Cette amélioration réduit les artefacts visuels [13] résultant en un effet *bouncy*, tout en améliorant la performance du bruit de Perlin. Utiliser le bruit OpenSimplex assure une continuité sur les coordonnées adjacentes, ce qui permet d'obtenir des courbes lisses vu qu'il est généré à partir d'une carte de gradients évalués et interpolés. Le bruit fractal est dans notre cas la sommation de plusieurs bruits OpenSimplex, celui-ci est utilisé directement comme *heightmap* afin de représenter de façon réaliste un mouvement chaotique d'océans calmes [14].

En ce qui concerne l'implémentation, nous utilisons celle de Inigo Quilez en 3D qu'il partage sur ShaderToy [8]. Les trois dimensions correspondent aux trois paramètres de la fonction de bruit: nous utilisons la position *x* du sommet, sa position *y*, puis le temps écoulé depuis le début de l'application. Ce dernier permet une animation de la surface océanique.

Nous avions pensé à deux manières de plaquer les hauteurs à l'aide de la carte de bruit. La première était de passer par le nuanceur d'évaluation de tessellation et d'évaluer directement le bruit à chaque position de sommet. La deuxième consistait à générer 3 cartes de bruit à 3 intervalles de temps successifs sur un quad enveloppant la surface de l'eau dans un nuanceur et

de les rendre dans trois FBOs lors de l'initialisation du programme principal. Une fois rendus dans les textures des FBOs, il aurait été possible d'échantillonner comme toute autre texture et d'interpoler entre ces trois textures selon le temps depuis le début de programme. La première idée s'est révélée suffisante, nous l'avons donc conservée.

4. Défis rencontrés

4.1. Craques

Le premier défi que nous avons rencontré était la présence de discontinuités dans le maillage de la surface que nous allons appeler les craques. Cela cause un artefact visuel qui ne peut pas être ignoré si nous voulons produire un rendu réaliste. Le problème des craques surgissant de la subdivision en *quadtree* est fréquemment relevé dans la littérature. Il est possible d'apercevoir sur la capture d'écran ci-dessous ce phénomène qui produit du vide visuel entre certains *patch*.

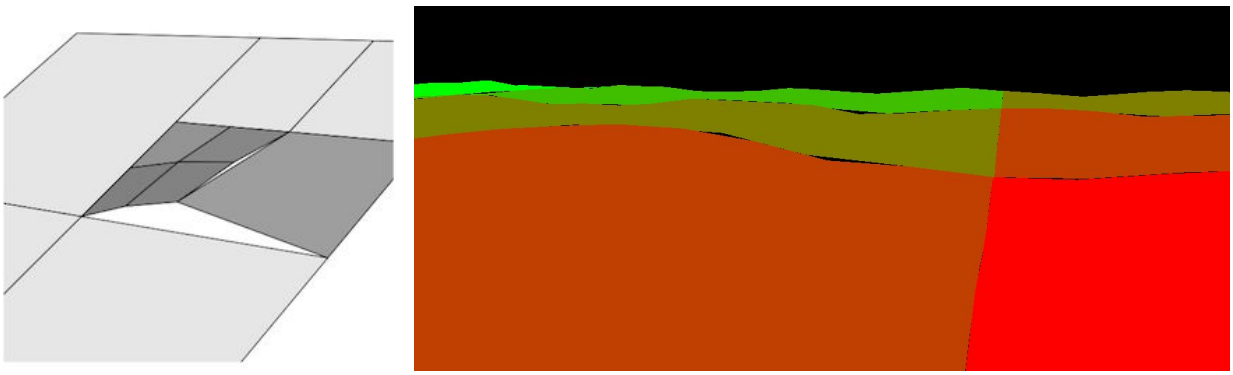


Figure 4. Craque dans le maillage [16]

La raison de cet effet est que dans un maillage à multiples résolutions comme la nôtre, il peut apparaître des *T-junctions* [10], créant des discontinuités dans les attributs interpolés. En d'autres mots, puisque tous les sommets de tous les *patch* sont générés une fois et que la hauteur est calculée pour chaque sommet, il y aura des problèmes de continuité dans l'interpolation entre deux *patch* à ce *T-junction*. Voici quelques endroits ou l'on retrouve des *T-junctions* dans le rendu de *quadtree* qui a été présenté précédemment:

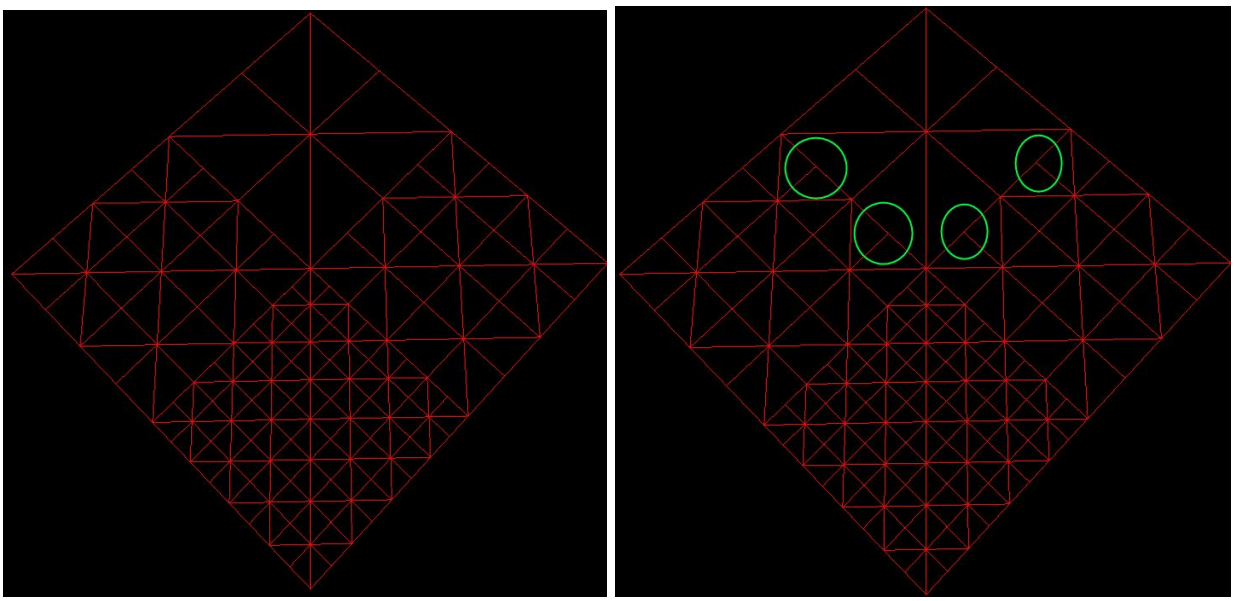


Figure 5. Position des jonctions en T

Pour régler ce problème, il faut trouver une stratégie pour décider s'il faut ou non générer un sommet supplémentaire. Il est souhaitable de ne pas générer un sommet d'un *patch* plus petit s'il avoisine un *patch* plus grand. L'idée proposée par Victor Bush est de prime abord garantir que la taille des voisins n'auront que des différences d'un facteur de 2 exactement. Ensuite, il faut évaluer dans le nuanceur de contrôle de tessellation, par le biais de propriétés du *patch* passées en variable uniforme, si une arête du *patch* avoisine un *patch* plus grande. Dans ce cas, nous modifions la tessellation de cette arête de sorte qu'elle soit mise à l'échelle d'un facteur de 2. L'image suivante illustre les résultats de cette modification au niveau de la tessellation.

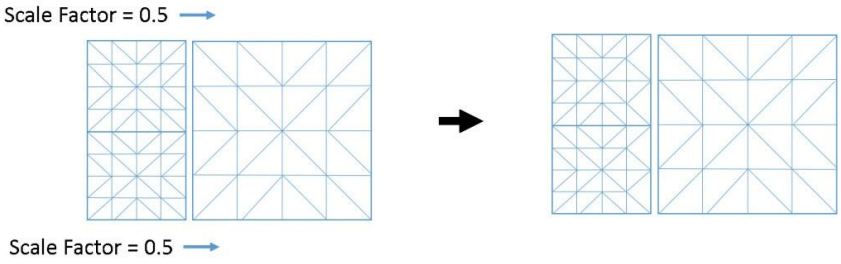


Figure 6. Correction par tessellation [1]

Le résultat est la disparition des jonctions en T ce qui fait en sorte que l'ensemble des arêtes des *patches* correspondront à celles de leurs voisins. Voici le résultat de l'application de la carte de hauteur sur une surface générée avec le *quadtree* qui applique la correction dans le nuanceur de tessellation pour éviter l'apparition de craques:

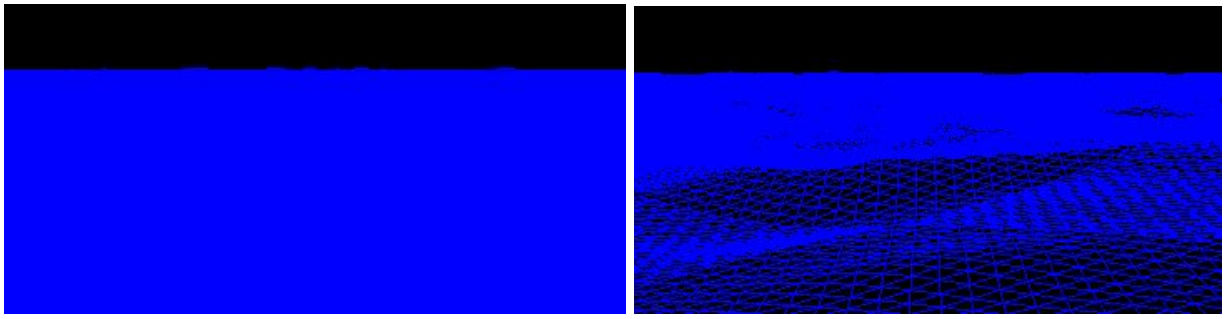


Figure 7. Résultat sans craque

L'absence d'illumination empêche de percevoir les différences de hauteur sans l'affichage en grille, mais l'absence de craque est bien visible.

4.2. Distortions des normales

Le deuxième défi que nous avons rencontré est au niveau de l'interpolation des normales. L'implémentation de normales est une preuve de concept qui permettrait dans un futur projet d'implémenter des coordonnées de textures de sorte à étendre les limites notre solution. En vue de produire un rendu réaliste, il est nécessaire d'illuminer notre scène. Nous avons illuminé la scène avec les méthodes traditionnelles de rastérisation. Le modèle d'illumination que nous implémentons est le modèle de Phong, qui consiste notamment à interpoler les normales au nuanceur de fragments. Puisque notre pipeline implémente les nuanceurs de tessellation et que les hauteurs sont calculées dans son nuanceur d'évaluation, nous devons générer les normales au même endroit.

Premièrement, notre nuanceur d'évaluation de tessellation travaille sur un *patch* de 4 sommets. Nous n'avons besoin que de 3 sommets pour établir la normale d'un sommet en réalisant le produit vectoriel. Pour chaque sommet nous trouvons sa position déplacée en hauteur ainsi que celles de deux points infiniment proches. Comme notre surface est un plan il suffit de poser l'un des points comme étant légèrement décalé dans l'axe des x et l'autre dans l'axe des z. Nous trouvons les vecteurs partant du sommet vers ces deux points infiniment proches pour ensuite en faire le produit vectoriel. Le produit entre ces deux vecteurs doit être fait dans l'ordre permettant que la normale soit dans la direction *y* positive. Voici une image représentant l'idée:

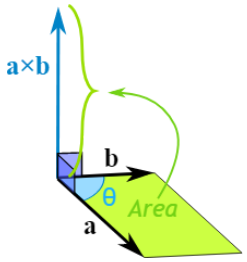


Figure 8. Calcul de la normale d'un plan [17]

Le résultat est généralement bien visuellement, cependant quelques distorsions peuvent être perçues sur l'intersection entre deux *patchs.* En voici un exemple ci-dessous:

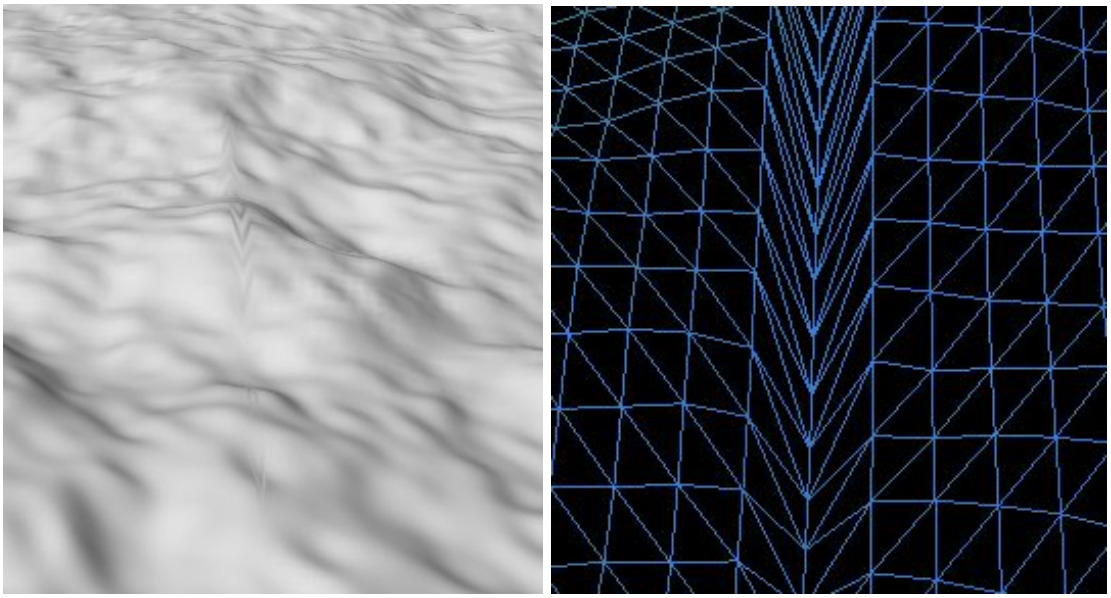


Figure 9. Distorsion dans les vagues

Comme nous pouvons le voir, cette distorsion est causée par une section non uniforme du maillage. Une solution qui à été essayée consiste à reporter le calcul de la normale afin dans le nuanceur de fragments afin d'éviter qu'elle soit distorsionné par l'interpolation. Cette solution est lourde en calculs vu qu'il y a un nombre de fragments nettement supérieur au nombre de sommets. De plus, la distorsion restait en partie visible. En effet, l'illumination semblait visuellement correcte, mais l'animation était tout de même différente pour ces sections vu que les mouvements des fragments ne concordaient pas avec leurs voisins. La solution idéale ferait en sorte que le maillage soit uniforme même dans ces sections. La solution n'a pu être trouvée au cours de ce projet.

4.3. Difficultés techniques

Notre rapport présente premièrement une implémentation naïve du LOD à l'aide de la tessellation. Cet effort initial témoigne d'un manque de compréhension de notre part par rapport à au papier technique [14]. Notamment, la différence entre la tessellation et la subdivision nous échappait. C'était à travers de l'étude supplémentaire que nous avons retracé le papier de Lindstrom [6] qui est à la base de plusieurs autres recherches sur les LOD sur de larges surfaces polygonales. Celui-ci nous a permis de démystifier plusieurs termes techniques présents dans [14] qui étaient traduits maladroitement. Il nous a aussi donné un élan dans notre recherche en l'orientant vers le rendu de larges terrains procéduraux. Bien sûr, la différence entre notre méthode de génération d'eau et celle de la génération de terrain diffère en ce qui concerne le bruit. Puisque l'eau est en mouvement, nous avons une 3e dimension qui représente le temps.

Enfin, sommes parvenus à rassembler les connaissances techniques nécessaires pour mieux diriger notre recherche et mener à bien le projet.

5. Résultats finaux

Nous incluons en Annexe des images montrant les résultats de notre projet. Le tableau suivant présente les commandes que les utilisateurs peuvent utiliser pour explorer la scène.

Tableau des commandes fournies par l'application

Touche	**Effet**
G	Affichage du maillage en *wireframe*
T	Fige l'état du *quadtree*. Le *quadtree* ne se mettra plus à jour selon la distance par rapport à la caméra.
Y	Augmente le multiplicateur de la hauteur des vagues
U	Diminue le multiplicateur de la hauteur des vagues
W, A, S, D, Souris	Déplacement dans la scène
1	Activation de la lumière directionnelle
2	Activation de la lumière positionnelle
3	Activation de la lumière spotlight

Pour conclure, nous souhaitons éventuellement étendre notre implémentation de sorte à supporter des textures afin de produire un rendu plus réaliste. Comme mentionné dans la section 4.2., les coordonnées de textures devraient être passées comme les normales. Nous devons toutefois résoudre l'interpolation des normales avant d'implémenter les textures. Également, il aurait été intéressant d'explorer et de comparer une implémentation du bruit à l'aide de FBO, plutôt que directement faire des calculs dans le nuanceur d'évaluation de tessellation. De cette façon, nous aurions pu nous rapprocher davantage de l'implémentation décrite en [14]. De plus, notre méthode consomme une charge de traitement GPU élevée. Si nous faisions un *bake* de nos résultats hors-ligne comme il est proposé, cette charge de calculs est réduite à de simples accès en mémoire. Enfin, notre implémentation, avec un quadrilatère de départ de 1000x1000, produit des résultats suffisamment performants pour un projet d'école à 348 fps en moyenne. Le système utilisé pour cette mesure utilisait une carte graphique GTX 1070 avec 8Go de mémoire vive, un processeur intel i7-7740X et 32 Go de mémoire vive. Par contre, pour un projet plus ambitieux où d'autres modèles 3D se retrouvent dans la scène et/ou d'autres techniques avancées en

infographie peuvent rapidement détériorer les performances. Aujourd'hui, les jeux vidéo s'attendent à rouler à un constant 60 fps. Avec l'avènement des cartes graphiques supportant du lancer de rayon en temps réel, notre solution paraît encore moins bien optimisée. Les contributions de chaque personne dans le projet est présentée à l'annexe B.

6. Références

[1] Bush, V. (2015). Tessellated Terrain Rendering with Dynamic LOD. https://victorbush.com/2015/01/tessellated-terrain/

[2] Bush, V. (2015). Tessellated terrain with dynamic LOD. https://bitbucket.org/victorbush/ufl.cap5705.terrain/src

[3] Cantlay, I., Tatarinov, A. (2014). From Terrain To Godrays: Better Use of DX11, Developer Technology Group NVIDIA, GDC 2014. https://developer.nvidia.com/sites/default/files/akamai/gameworks/events/gdc14/GDC_14_From%20Terrain%20to%20Godrays%20-%20Better%20Use%20of%20DirectX11CantlayTatarinov.pdf

[4] Kass, M., Miller, G. (1990). Rapid, stable fluid dynamics for computer graphics. SIGGRAPH Comput. Graph. 24, 4 (Aug. 1990), 49–57. https://doi.org/10.1145/97880.97884

[5] Khronos. (2020). Tessellation, OpenGL Wiki. https://www.khronos.org/opengl/wiki/Tessellation

[6] Lindstrom, P., Koller, D., Ribarsky, W., Hodges, L. F., Faust, N., & Turner, G. A. (1996) Real-Time, Continuous Level of Detail Rendering of Height Fields, SIGGRAPH 96 Conference Proceedings, pp. 109-118, Aug 1996.

[7] Müller, M. (2007). Real Time Fluids in Games, University of British Columbia. https://www.cs.ubc.ca/~rbridson/fluidsimulation/GameFluids2007.pdf

[8] Quilez, I. (2013). Noise - simplex - 2D, ShaderToy. https://www.shadertoy.com/view/Msf3WH

[9] Red Giant. (2020). Fractal Noise, Red Giant by Maxon. https://www.redgiant.com/user-guide/trapcode-shine/fractal-noise/#:~:text=Fractal%20noise%20is%20created%20by,reduce%20detail%20in%20the%20noise.

[10] Reddy, M. (2002). Terrain Level Of Detail, SIGGRAPH 2002. https://graphics.pixar.com/library/LOD2002/4-terrain.pdf

[11] Starn, J. (2001). A Simple Fluid Solver Based on the FFT, Journal of Graphics Tools, 6:2, 43-52, DOI: 10.1080/10867651.2001.10487540

[12] Wagner, D. (2004). Terrain Geomorphing in the Vertex Shader. TU Wien.
https://www.ims.tuwien.ac.at/publications/tuw-138077.pdf

[13] Wikipédia. (2020). OpenSimplex noise, Wikipedia, the free encyclopedia.
https://en.wikipedia.org/wiki/OpenSimplex_noise

[14] Yang, X., Pi, X., Zeng, L., & Li, S. (2005). GPU-Based Real-time simulation and Rendering of Unbounded Ocean Surface. School of Computer, National University of Defense Technology.
http://www-evasion.inrialpes.fr/Membres/Fabrice.Neyret/images/fluids-nuages/waves/Jonathan/articlesCG/GPUBasedRealTimeSimulationAndRendering2005.pdf

[15] Zhao, Y., Ji, Z., Shi, J., & Pan, Z. (2001). A Fast Algorithm For Large Scale Terrain Walkthrough. CAD/GRAPHICS '2001, vol 1.
https://www.researchgate.net/publication/299394692_A_Fast_Algorithm_For_Large_Scale_Terrain_Walkthrough

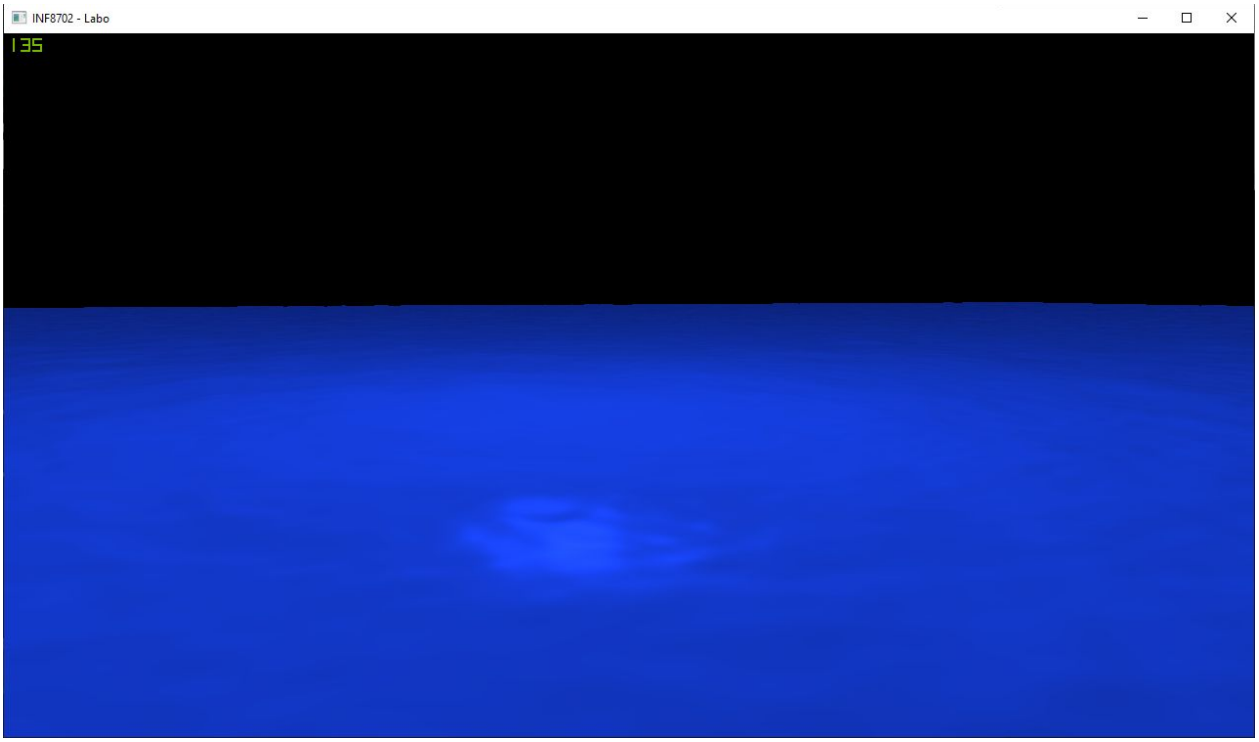
[16] lhf (2015) Why do T-junctions in meshes result in cracks? [En ligne]
Disponible:
https://computergraphics.stackexchange.com/questions/1461/why-do-t-junctions-in-meshes-result-in-cracks

[17] MathsIsFun.com (2017) Cross Product [En ligne]
Disponible: https://www.mathsisfun.com/algebra/vectors-cross-product.html

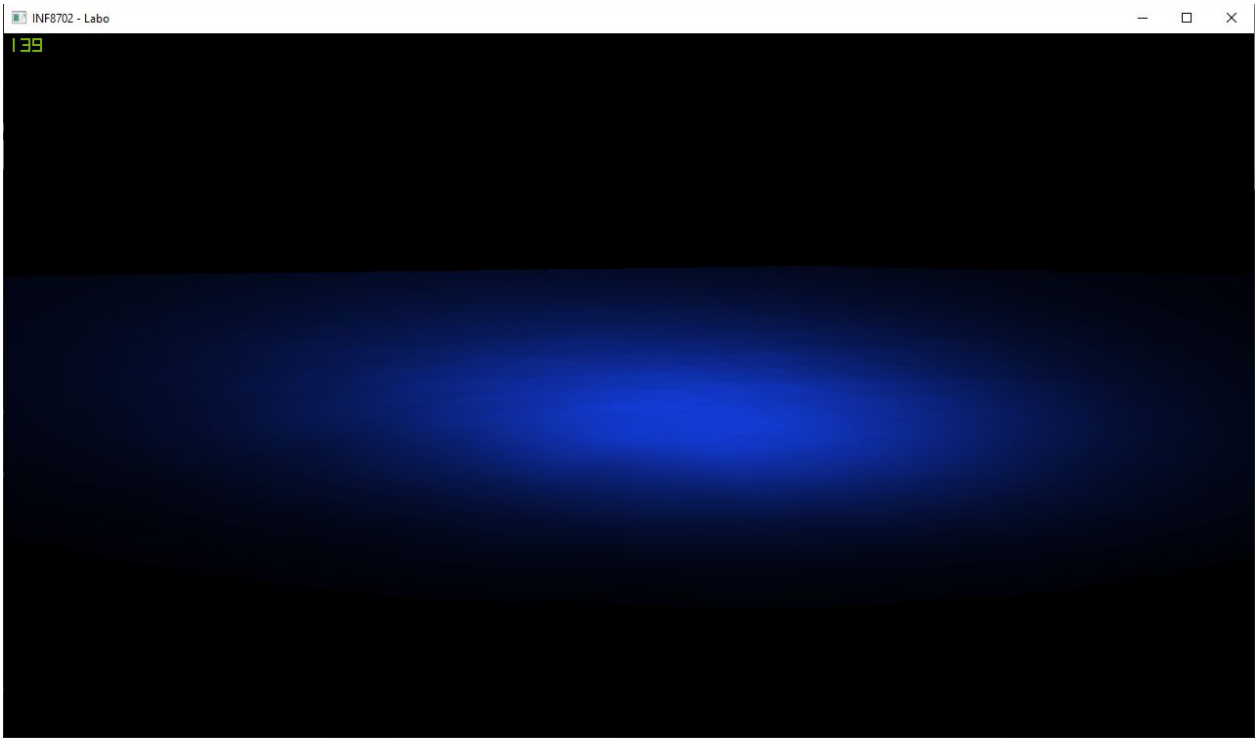
Annexe A

Directionelle:

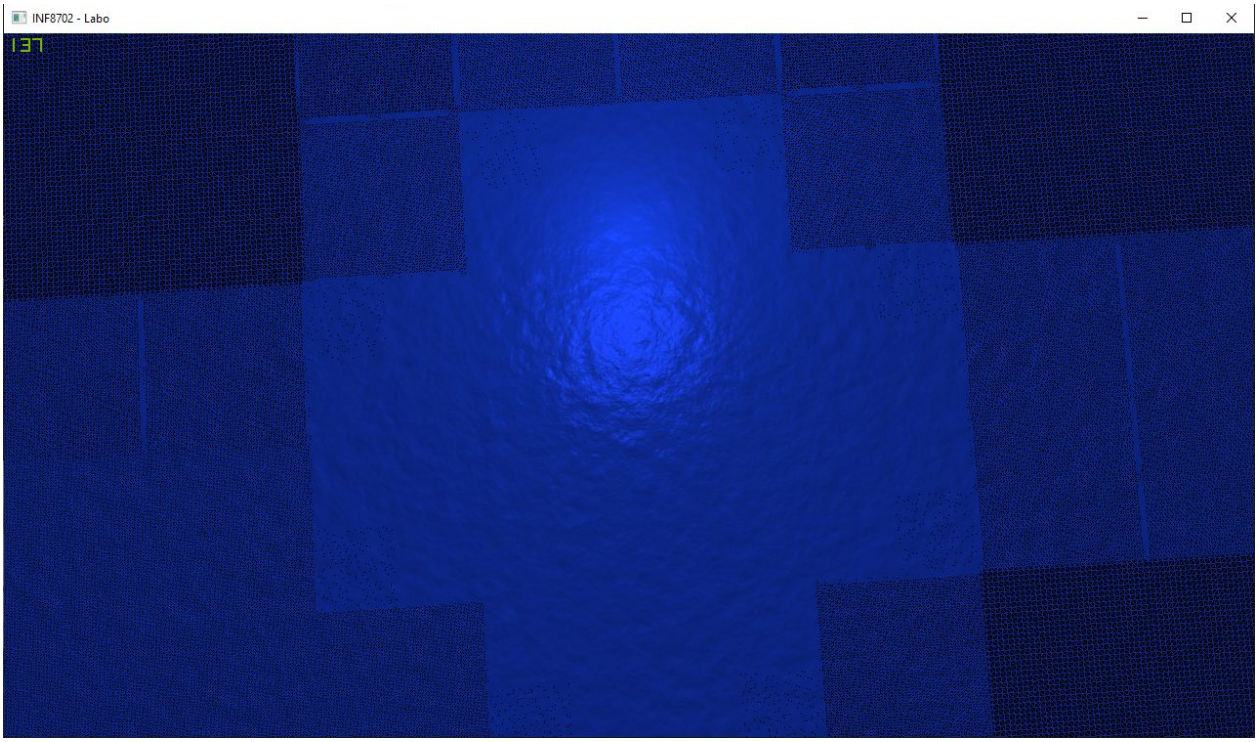
Positionnelle:



Spotlight:



Quadtree:



Annexe B

Contribution de Jonathan:

- Organisation et nettoyage du code du logiciel de base provenant des travaux pratique de ce cours
- Ajout des nuanceurs de tessellation
- Implémentation de l'envoi des VBO, VAO dans la structure du quad-tree
- Adaptation au programme du code gérant le quad-tree
- Interpolation des normales et illumination
- Nuanceur de contrôle de tessellation (tessellation et LOD, gestion des artéfacts)
- Nuanceur d'évaluation de tessellation (débogage, gestion des artéfacts)
- Rédaction du rapport et de la présentation Google Slides
- Écriture des nouvelles commandes de touches clavier

Contribution de Huyen Trang:

- Recherche sur les quad-trees, le bruit, les exemples d'implémentation et le retraçage du contexte.
- Adaptation de la structure quad-tree dans notre logiciel
- Interpolation des normales et illumination
- Nuanceur de contrôle de tessellation (gestion des artéfacts)
- Nuanceur d'évaluation de tessellation (bruit, déplacement des sommets)
- Peaufinement des valeurs de lumières et de matériel
- Rédaction du rapport et de la présentation Google Slides