



FACULTÉ DES SCIENCES DE MONTPELLIER

MASTER 1 IMAGINE
ANNÉE 2021-2022

RAPPORT DE TER

Stylisation d'images par triangulation

Groupe SALAGG :
Alexandre SPATOLA
Alexandre LANVIN
Gauthier GERMAIN

Encadrants :
Noura FARAJ
William PUECH

Remerciements

Nous adressons nos remerciements tout particuliers à nos deux encadrants Noura FARAJ et William PUECH qui nous ont accompagnés et ont su répondre à nos questions lors de ce projet.

Table des matières

1	Introduction	4
2	Présentation du programme et des fonctionnalités	5
1.	Page d'accueil	5
2.	Modes de rendu	7
3.	Génération d'une triangulation initiale	7
4.	Passes d'optimisation	9
5.	Suppression manuelle de sommets	10
6.	Export de l'image finale	10
7.	Quelques résultats de triangulation	11
3	Rapport technique	12
1.	Architecture Logicielle	12
a.	Technologies utilisées	12
b.	Classes	13
2.	Interface Utilisateur	13
3.	Génération de grille initiale	15
a.	Régulière	15
b.	Par Split and Merge	15
c.	Par extraction de contour	15
d.	Triangulation de Delaunay	15
4.	Optimisation de la triangulation	16
a.	Calcul de l'énergie	16
b.	Descente de gradients	17
c.	Topologie dynamique	17
d.	Stabilité numérique	18
e.	Énergies alternatives	19
5.	Rendu	19
a.	Couleur constante	19
b.	Couleur en gradients linéaires	21
6.	Études documentaires	23
a.	Image Smoothing via L_0 Gradient Minimization	23
b.	A Remeshing Approach to Multiresolution Modeling	24
c.	Image Signature : Highlighting Sparse Salient Regions	25

4 Rapport d'activité	26
1. Rôles dans l'équipe	26
2. Organisation	26
3. Difficultés rencontrées	27
5 Conclusion	28
1. Bilan	28
2. Perspectives d'améliorations	28

Chapitre 1

Introduction

La triangulation d'images est un art permettant d'obtenir des images abstraites et dont les formes sont simplifiées tout en conservant certaines caractéristiques, comme certains contours ou encore une forme spécifique. Cet art est en effet relativement chronophage pour l'artiste en fonction du niveau de détails auquel il souhaite travailler. De plus, il s'agit d'un travail fastidieux qui requiert des compétences artistiques solides.

L'objectif de ce projet est donc de mettre à disposition à l'utilisateur une grille triangulée automatiquement, plusieurs modes de rendu ainsi que la possibilité de modifier la triangulation de multiples façons. Ainsi la triangulation d'images deviendrait moins chronophage et serait plus accessible puisqu'il ne serait plus nécessaire d'être artiste pour la réaliser.

Nous nous sommes largement inspirés de l'[article](#) principal fourni par nos encadrants (*Stylized Image Triangulation* par Kai LAWONN et Tobias GÜNTHER).



FIGURE 1.1 – Exemples d'images triangulées présentées dans l'article.

Chapitre 2

Présentation du programme et des fonctionnalités

1. Page d'accueil

L'utilisateur est accueilli par une interface minimale. Il a seulement la possibilité d'ouvrir une image dont il souhaite réaliser une triangulation.

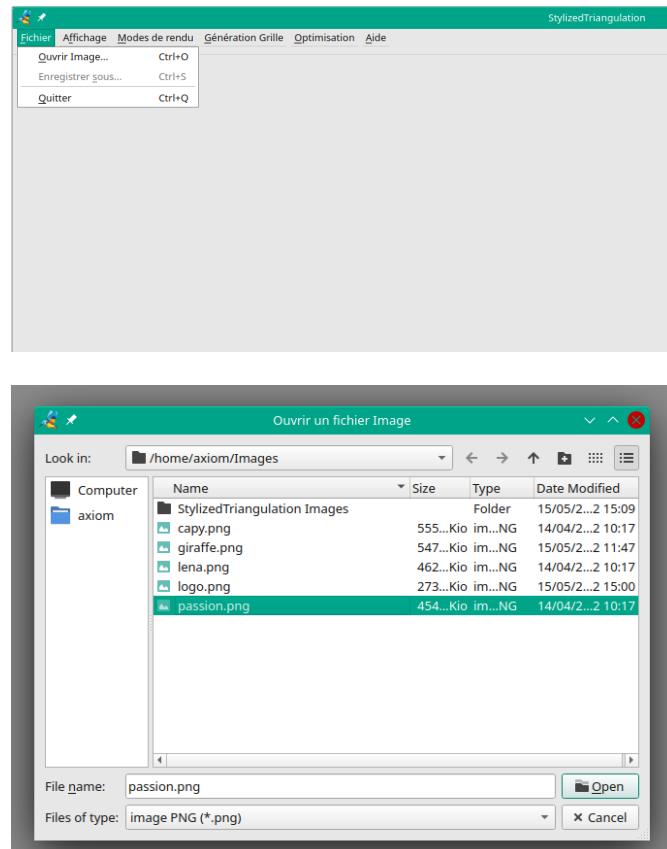


FIGURE 2.1 – Ouverture de l'image à trianguler

Considérons qu'il choisisse l'image suivante :



FIGURE 2.2 – *passion.png*

Il lui sera alors présenté une première triangulation régulière de son image :

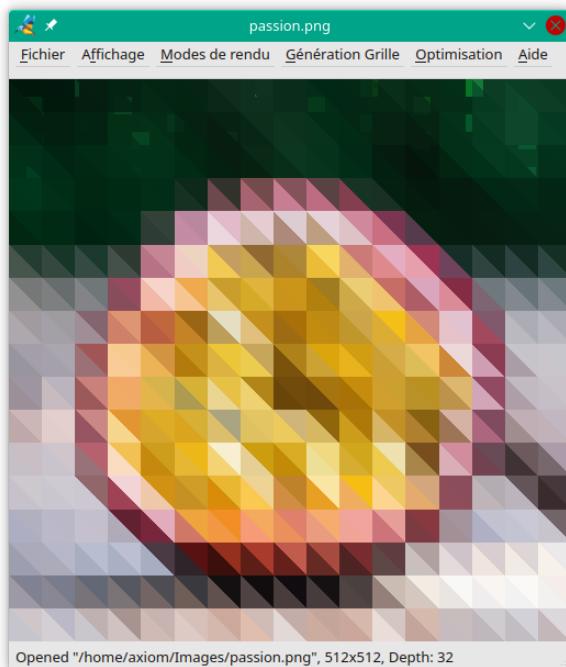


FIGURE 2.3 – Triangulation initiale lors de l'ouverture

A partir de là, l'utilisateur a plusieurs choix.

2. Modes de rendu

L'utilisateur a la possibilité d'alterner à n'importe quel moment entre les différents modes de rendu.

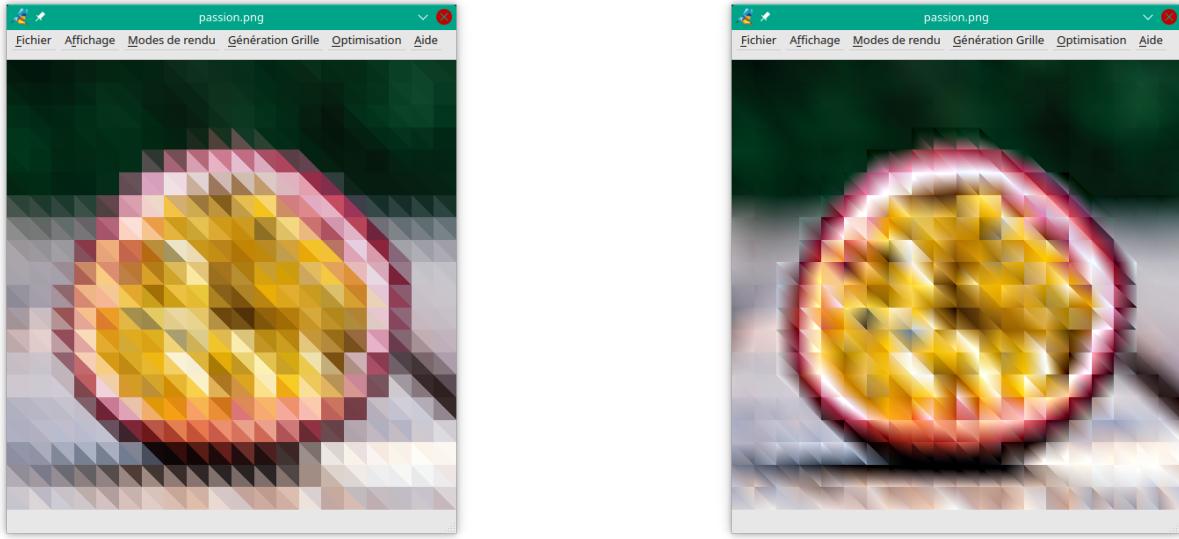


FIGURE 2.4 – Modes de rendu

3. Génération d'une triangulation initiale

L'utilisateur a également l'option de régénérer une grille régulière avec la résolution de son choix :

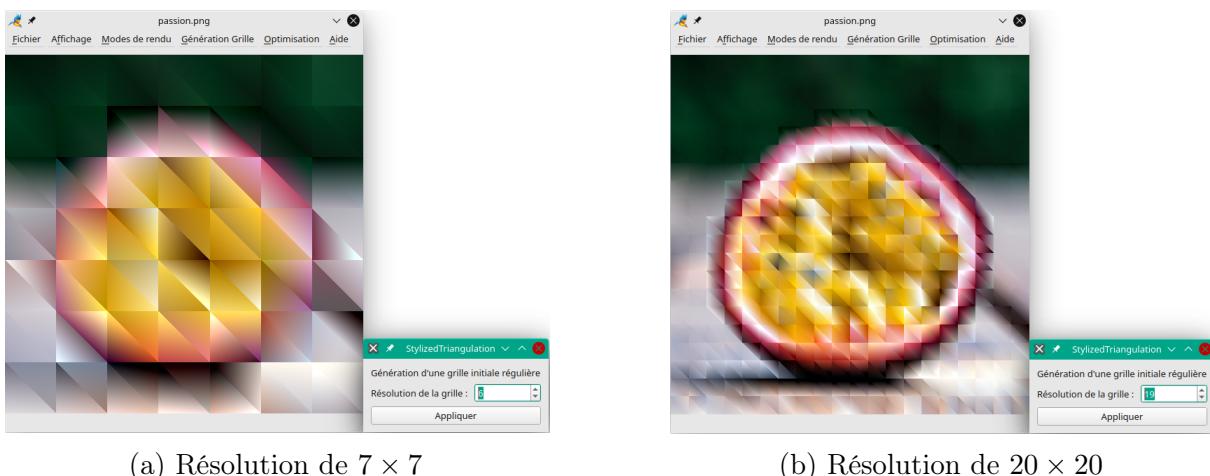


FIGURE 2.5 – Génération d'une grille initiale régulière

Il peut en outre générer une ensemble de points initial en utilisant les coordonnées des régions obtenues après segmentation par une méthode de *Split and Merge*. Du contrôle est laissé à l'utilisateur sur cette étape de segmentation en lui permettant d'agir sur les paramètres de génération (Variance maximale, distance maximale dans le graphe d'adjacence).

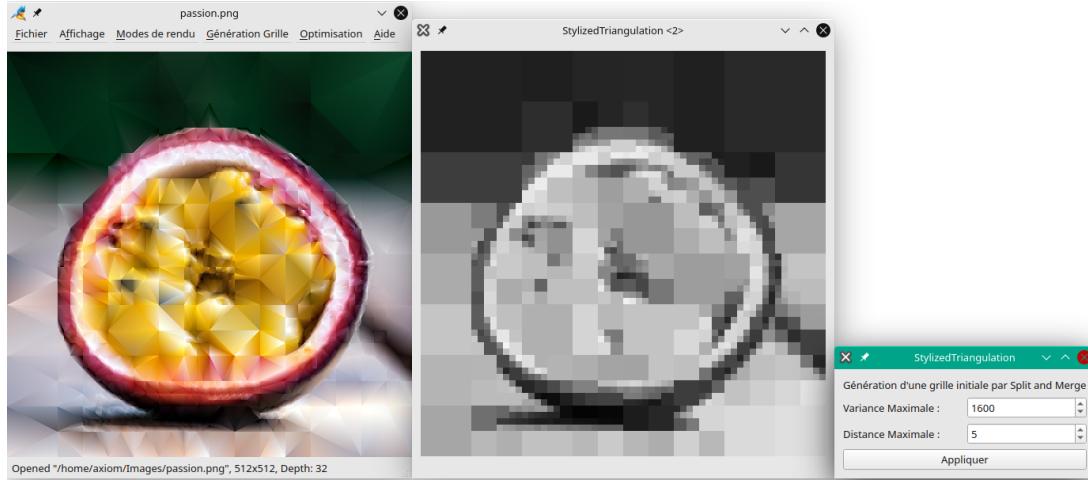


FIGURE 2.6 – Génération d'une triangulation : Approche *Split and Merge*

De plus, il a le choix de générer une première triangulation basée sur une approche contour de l'image. Nous avons choisi pour cela l'application d'un filtre de Sobel. Un échantillonage des points va ensuite disposer préférentiellement les points sur les parties importantes de l'objet : ses bords. Encore une fois, nous donnons la possibilité à l'utilisateur de changer les paramètres de génération de la carte de contour et de changer les paramètres d'échantillonnage.

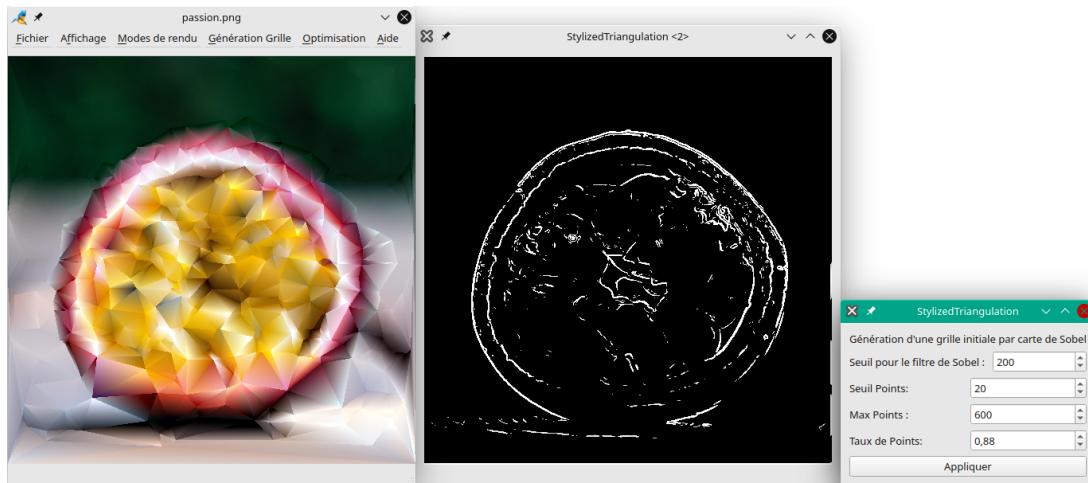


FIGURE 2.7 – Génération d'une triangulation : Approche contour (Sobel)

4. Passes d'optimisation

Ce que l'on appelle l'optimisation est la partie du logiciel qui va modifier progressivement la triangulation afin d'approcher une approximation de l'image originale satisfaisante pour l'image stylisée.



FIGURE 2.8 – Exemple d'optimisation

L'utilisateur a l'option d'exécuter des passes d'optimisation une par une ou en continu, à une vitesse paramétrable. Il peut également choisir si les triangles peuvent se subdiviser pendant l'optimisation ou non.

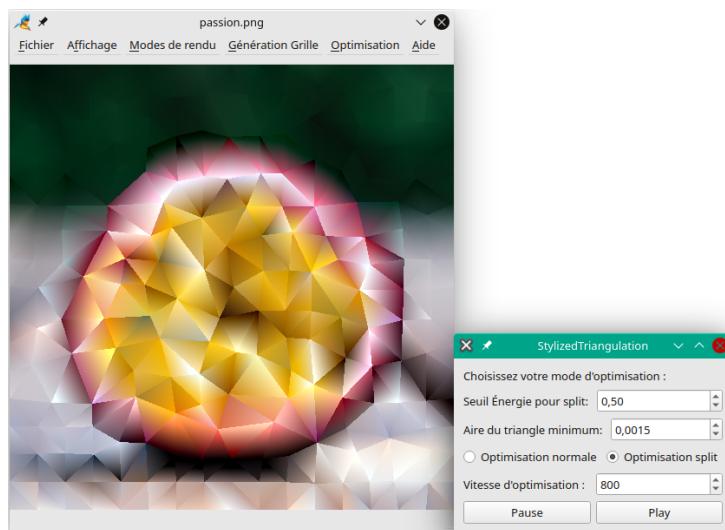


FIGURE 2.9 – Optimisation en continu

5. Supression manuelle de sommets

L'utilisateur a la possibilité de retirer manuellement des sommets en cliquant sur ces derniers. Une retriangulation locale est alors effectuée.



FIGURE 2.10 – Supression manuelle de sommets

6. Export de l'image finale

Lorsque l'utilisateur est satisfait il peut sauvegarder son image triangulée dans le format de son choix.

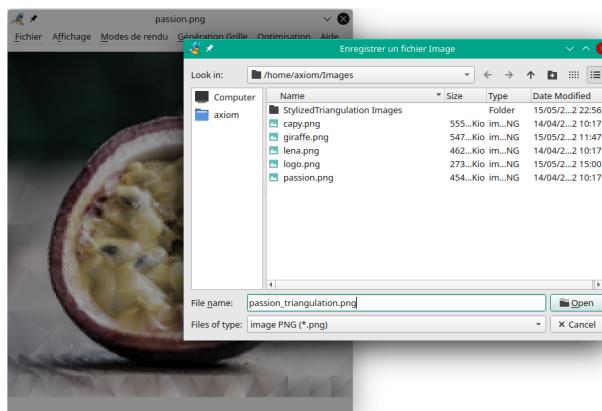
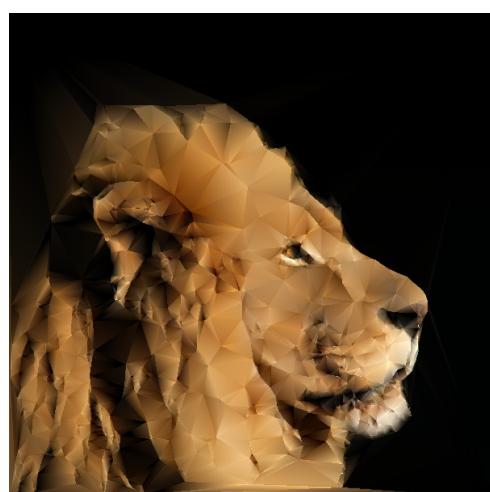


FIGURE 2.11 – Enregistrement de l'image finale

7. Quelques résultats de triangulation



Chapitre 3

Rapport technique

1. Architecture Logicielle

a. Technologies utilisées

Nous avons utilisé [Qt](#) qui est une bibliothèque logicielle multiplateforme permettant de développer des interfaces graphiques et des applications portables grâce à un système de build intégré. Ainsi nous avons pu créer une application avec laquelle l'utilisateur peut interagir et modifier les différents paramètres liés à la triangulation de l'image lors de l'exécution. Enfin, notre application a été développée dans le langage C++.



Inclusion de Eigen

Nous nous sommes servi de la librairie [Eigen](#) qui est une bibliothèque d'analyse numérique qui nous met à disposition des outils d'algèbre linéaire. Nous nous sommes particulièrement servis de la décomposition de Cholesky ainsi que du solveur lors du calcul de la couleur gradients linéaires.



FIGURE 3.2 – Logo d'Eigen

b. Classes

Nous avons séparé notre architecture en trois parties principales : **le rendu, l'optimisation et l'interface utilisateur.**

Nous avons créé une classe *Triangulation*, responsable de contenir une triangulation ainsi que de maintenir les données topologiques associées, afin de partager ces données de manière unifiée entre les différentes parties. De même, nous avons choisi de transmettre l'image originale en tant que texture OpenGL, laissant la responsabilité des détails de son obtention au code UI.

L'optimisation de la triangulation est intégralement implémentée dans la classe *TriangulationOptimizer*. Similairement, le rendu est géré par la classe *Renderer*, c'est celle ci qui calcule les différents modes de rendu proposés par l'application.

La classe *MainWindow* est la classe définissant la fenêtre principale sur laquelle nous greffons un widget central de rendu défini dans *glWidget* qui permet d'encapsuler le contexte openGL utilisé par les parties de rendu et d'optimisation.

2. Interface Utilisateur

Nous avons choisi une interface minimale composé d'une barre de menu, d'un Widget central de rendu et d'une barre de statut permettant de donner des informations lors de l'ouverture d'image par exemple.

Nous nous sommes parti d'un [Exemple d'Image Viewer](#) que nous avons dû adapter selon les features ajoutés. Toutes les fonctionnalités sont accessibles dans la barre de menu ou directement via des raccourcis clavier. Lors de la sélection de ces fonctionnalités, un signal est émis au *slot* auquel il est connecté permettant d'ouvrir une fenêtre de dialog, de quitter l'application...

Le Widget central de la fenêtre est une classe personnalisée héritant de *QOpenGLWidget*. C'est là qu'est initialisé le contexte de rendu et que l'on redéfinit les fonctions permettant l'initialisation, le redimensionnement et la mise à jour du rendu.

L'objet QImage de Qt nous permet simplement d'ouvrir et d'enregistrer notre image sous un grand éventail de formats. Pour la sélection de points, nous récupérons les coordonnées fenêtres du clic, avant de les convertir en coordonnées textures. Il suffit alors d'itérer sur tous les sommets et de sélectionner celui ayant la distance euclidienne au curseur la plus courte.

Les Fonctionnalités des Menus pour la génération de grille ouvrent des fenêtres de dialog contenant des *QSpinBox* permettant de modifier les paramètres de génération.

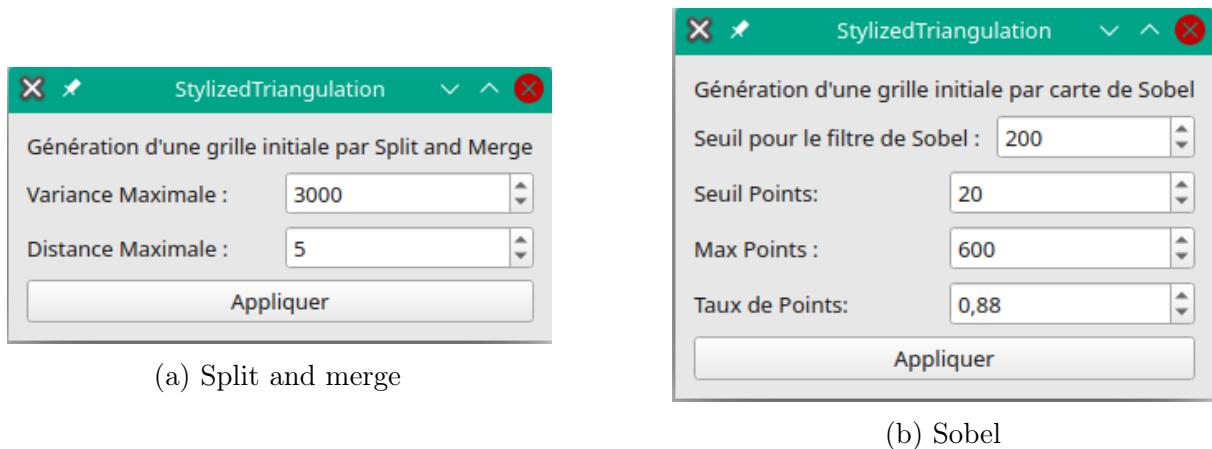


FIGURE 3.3 – Exemples des fenêtres permettant de modifier les paramètres de génération de grille via des SpinBox

Pour l'optimisation continue, nous donnons le choix entre le mode d'optimisation avec ou sans split des triangles, et permettons de configurer la vitesse à laquelle ces passes sont effectuées (nous avons utilisé pour cela un *QTimer*).

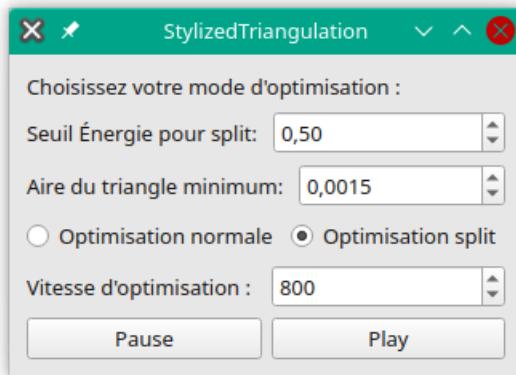


FIGURE 3.4 – Fenêtre pour l'optimisation en continu

3. Génération de grille initiale

a. Régulière

Nous plaçons des points répartis de manière régulière en fonction d'un paramètre de résolution et nous relierons ces points par carré et deux triangles par carré (sens antihoraire).

b. Par Split and Merge

Pour rappel la méthode de segmentation *Split and merge* est une technique qui consiste à effectuer des division récursives de l'image en blocs selon un critère d'homogénéité comme la variance à l'intérieur du bloc. Après vient l'étape de fusion qui consiste à fusionner des régions voisines ayant des valeurs moyennes assez proches (inférieur à un seuil). Pour cela nous nous servons d'un graphe d'adjacence (RAG) généré lors de l'étape de division.

Nous avons donc récupéré et adapté l'algorithme implémenté en Travaux Pratiques d'*Analyse et Traitement d'Images* de Monsieur Puech. Il nous a suffit de renvoyer les points correspondants aux coins des blocs ainsi générées.

Remarque : cette méthode ne marche que pour les images carrées.

c. Par extraction de contour

Le but de cette partie est de déterminer une carte en niveau de gris avec des intensités lumineuses dépendantes des variations de couleur locales.

Génération de cartes de Sobel

Nous avons effectué une convolution des Noyaux de Sobel en x et en y sur notre image d'origine A :

$$\mathbf{G}_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * \mathbf{A} \quad \text{et} \quad \mathbf{G}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * \mathbf{A}$$

Finalement, nous renvoyons la carte correspondant à la norme du gradient : $\mathbf{G} = \sqrt{\mathbf{G}_x^2 + \mathbf{G}_y^2}$

Il faut ensuite effectuer un seuillage puis un échantillonage en 2D des points selon la carte de contours ainsi obtenue.

d. Triangulation de Delaunay

Les méthodes de Split and Merge et les Cartes de Gradient et de Sobel (après échantillonage) décrites ci-dessus ne nous renvoient que des listes de points non ordonnées.

Nous nous sommes donc servis de la librairie [delaunator-cpp](#) qui permet de récupérer, après adaptation de nos structures de données, une liste de triangles indexés.

4. Optimisation de la triangulation

Le but de l'optimisation est de modifier la géométrie de la triangulation afin d'épouser au mieux certaines caractéristiques de l'image, l'exemple le plus simple étant la couleur.

Pour ce faire, l'optimiseur définit pour chaque sommet une énergie, qu'il cherche ensuite à minimiser en modifiant la triangulation. La triangulation est optimale lorsque l'énergie est minimale pour chaque sommet.

a. Calcul de l'énergie

Avant de la définir pour un sommet, il est plus ais  et intuitif de définir l'énergie d'un triangle. Celle-ci est une grandeur abstraite, pour laquelle une grande valeur exprime que le triangle n'est pas id al pour la stylisation souhait e.

Dans la majorit  des cas, comme par exemple si l'on cherche   rapprocher au maximum la couleur de l'image stylis e de celle de l'originale, l'énergie peut  tre d finie comme l'accumulation en chaque point du triangle d'une diff rence entre les points correspondants de l'image originale et stylis e. Formellement, l'énergie ϵ d'un triangle Δ s'exprimerait :

$$\epsilon(\Delta) = \int_{\Delta} d(O(p), S(p)) dp$$

o  $O(p)$ et $S(p)$ correspondent au point de coordonn es p de respectivement l'image originale et stylis e, et d est une fonction distance entre ces deux points. Dans l'exemple de l'optimisation de la couleur, d serait la somme des diff rences carr es de chacune des composantes R, G et B.

 tant donn  que le support informatique impose des images discr tes, l'int grale pr c dente devient une somme o  p repr sente les coordonn es d'un pixel de Δ .

Cela implique  g alement une m thode directement adapt e au probl me : effectuer un traitement pour chaque pixel d'un triangle, et ce sur un ensemble de triangles, est exactement ce pourquoi les GPUs sont con us. La triangulation est donc envoy e directement   OpenGL (nous avons utilis  OpenGL, mais n'importe quelle API graphique aurait pu l'affaire), et apr s la rasterization, le fragment shader est appell  pour chaque pixel de chaque triangle. Il suffit alors de programmer ce fragment shader pour calculer le terme correspondant de la somme. Ces termes sont ainsi accumul s dans un compteur propre   chaque triangle   l'aide d'op rations atomiques, qui permettent de conserver le parall lisme malgr  l'acc s   une donn e partag e entre les threads.

En pratique, nous utilisons deux passes : une premi re passe permet de calculer la couleur moyenne des pixels de l'image originale dans chaque triangle, et la seconde utilise cette couleur moyenne comme $S(p)$ afin de calculer l'énergie   proprement parler.

Ces donn es calcul es par le GPU sur la VRAM sont alors r import es sur la RAM pour  tre trait es par le CPU.

Une fois que l'énergie d'un triangle est calcul e, l'énergie d'un sommet est simplement d finie comme l'accumulation de l'énergie de tous les triangles dont il fait partie (divis e par 3 pour partager l'énergie des triangles entre leur sommets).

b. Descente de gradients

Une fois que l'énergie est définie aux sommets, le but est de la minimiser. Nous utilisons pour cela une technique de descente de gradients. Le concept est d'estimer le gradient de l'énergie aux sommets, qui décrit la pente de cette fonction énergie, afin de savoir quel déplacement du sommet la diminue le plus. Il suffit alors de modifier la géométrie en fonction. Le gradient $\nabla\epsilon(v)$ est calculé avec la formule :

$$(\nabla\epsilon)_i(v) = \frac{\epsilon(v + e_i\varepsilon) - \epsilon(v - e_i\varepsilon)}{2\varepsilon}$$

où $i \in \{0, 1\}$ est la dimension.

Le gradient étant une notion continue basée sur des écarts infinitésimaux, il est nécessaire de considérer de très courtes distances sur notre calcul discrétré. Nous choisissons donc ε comme étant la taille d'un pixel. De même, le déplacement engendré par la descente de gradients doit être limité à de très courtes distances afin de conserver une cohérence vis-à-vis du gradient qui est une information purement ponctuelle. Nous introduisons donc h qui représente une telle distance minime, et le sommet modifié par descente de gradients devient :

$$v' = v - h\nabla\epsilon(v)$$

En pratique, nous ajoutons en plus de h une opération pour limiter le déplacement à $\frac{\varepsilon}{5}$ sur chaque axe.

Pour calculer les gradients, il apparaît qu'il est nécessaire de connaître, en plus des énergies aux sommets, les énergies des sommets décalés dans les quatre directions (haut bas gauche droite). Pour ce faire, un geometry shader est ajouté à l'étape de calcul des énergies. Il s'agit d'un shader facultatif qui est capable de modifier les primitives fournies au reste de la pipeline graphique. Dans notre cas, nous programmons le geometry shader pour générer, à partir de chaque triangle, 13 nouveaux triangles : l'original, suivi de toutes les variantes où l'un des 3 sommets a été déplacé dans une des 4 directions d'une distance ε .

Une fois ces données obtenues, les gradients peuvent être calculés pour chaque sommet et la géométrie peut être mise à jour. En répétant ces traitements, les sommets vont converger vers des minimums d'énergie locaux, qui correspondront donc à une triangulation optimisée pour l'image.

c. Topologie dynamique

Jusqu'à maintenant, nous n'avons parlé que de modifications géométriques de la triangulation. Cependant, dans certains cas, il peut être intéressant que l'optimiseur corrigé également la topologie.

La première situation où cela peut s'avérer nécessaire est lorsqu'un triangle converge vers une forme trop allongée. Le programme détecte cette situation en vérifiant, pour chaque arête, la somme des deux angles opposés à cette arête. Si cet angle est supérieur à π , un edge flip est effectué. Les triangles gardent ainsi des formes plus régulières.

Il y a cependant des cas où cette opération n'est pas suffisante, et certains triangles

vont tout de même se retrouver écrasés. Pour palier à cela, tout triangle ayant une aire inférieure à un seuil fixé est supprimé. Cette suppression s'effectue en fusionnant deux sommets du triangle, en favorisant les deux sommets les plus proches, afin de modifier le moins possible l'apparence de la triangulation.

Enfin, l'optimiseur donne l'option de raffiner dynamiquement la triangulation. Si cette option est utilisée, tous les triangles dont l'énergie calculée est supérieure à un seuil donné seront split. Ce split consiste simplement à introduire un nouveau sommet au milieu du triangle, puis à remplacer ce dernier par les trois triangles qui relient le nouveau sommet à deux des anciens. Cela permet d'avoir une triangulation dont la précision s'adapte à la complexité de l'image, même en partant d'une grille régulière de basse résolution.

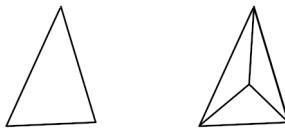


FIGURE 3.5 – Un triangle avant et après split

d. Stabilité numérique

Même en minimisant le mouvement des sommets à chaque itération, des sécurités supplémentaires sont nécessaires afin de garder une triangulation cohérente. En effet, le programme dans son entièreté suppose que la triangulation forme une partition de l'image, et il est de la responsabilité de l'optimiseur de conserver cette propriété.

La première étape est d'interdire les mouvements des sommets situés en bord d'image. Cette interdiction pourrait être assouplie ultérieurement en autorisant le mouvement le long du bord de l'image sur lequel se trouve le sommet.

Il faut de plus empêcher les triangles de se recouvrir mutuellement. En supposant que la triangulation de départ forme bien une partition et que tous les triangles sont connectés entre eux, cela revient à s'assurer que le triangle ne change pas d'orientation. Pour ce faire, l'orientation du triangle est vérifiée avant et après le déplacement de chaque sommet. Si un changement est détecté, le mouvement est annulé pour cette passe.

Pour minimiser ces situations, un coefficient de régularisation est également introduit. Il s'agit d'une valeur ajoutée à l'énergie du sommet lorsqu'elle est calculée, et qui a la propriété d'être minimisée lorsque le sommet est au centre de son voisinage. Ainsi, le sommet est attiré par ce centre lors de la descente de gradients. La valeur de ce coefficient pour le sommet v est :

$$\frac{1}{2|V(v)|} \sum_{p \in V(v)} d(v, p)^2$$

où $V(v)$ est le voisinage de v , et d la distance euclidienne. Ce coefficient est cependant ajouté avec un facteur faible afin de rendre l'effet subtil, et de ne pas prendre le pas sur les caractéristiques de l'image.

e. Énergies alternatives

Dans l'état actuel du programme, l'énergie des triangles est calculée d'une seule manière : à partir de la différence entre les couleurs de l'image et la couleur moyenne du triangle. Bien que cela soit adapté pour avoir le rendu en couleur moyenne le plus adapté, d'autres manières de calculer l'énergie sont possibles.

Une première piste serait de réutiliser le code issu du mode de rendu actuel pour calculer la couleur des triangles puis l'erreur sur cette couleur. La triangulation convergerait alors vers la forme idéale pour le mode de rendu choisi par l'utilisateur.

Il serait également intéressant d'étudier les changements apportés par une utilisation d'autres espaces de couleur que le RGB pour le calcul d'erreur.

Enfin, il y a également la possibilité de calculer l'énergie des sommets sur d'autres critères que la couleur des triangles. Nous pourrions par exemple prendre en compte la saillance ou les formes, en se basant cette fois ci sur les arêtes pour ces calculs. Cela entraînerait cependant un changement majeur dans la logique actuelle à la fois du calcul d'énergie, mais aussi des opérations sur la topologie qui prennent en compte l'énergie du triangle.

5. Rendu

Après avoir obtenu une triangulation il faut appliquer une couleur à notre rendu, le but étant que cette couleur approche au mieux l'image de base. Pour cela, nous avons choisi de stocker l'image dans une texture OpenGL et d'interpréter les coordonnées des sommets de la triangulation comme des UV. Ainsi plusieurs modes de rendu sont décrits dans [le principal article scientifique](#) fourni pour réaliser ce TER tels que la couleur constante et la couleur en gradients linéaires.

a. Couleur constante

La première approximation de la couleur de la triangulation est la couleur constante. Cette méthode de rendu consiste à appliquer une couleur unie sur chacun des triangles, cette couleur étant la moyenne des couleurs des pixels contenus dans le triangle. Ainsi, la couleur pour chaque triangle peut être exprimée de cette manière :

$$f(x, y) = c = \frac{1}{q} \sum_{i=1}^q I(x_i, y_i)$$

avec q : le nombre de pixels couverts par le triangle, $I(x, y)$: l'intensité lumineuse de l'image au point $[x, y]$ et $f(x, y)$: la couleur du pixel aux coordonnées $[x, y]$.

Cette méthode de rendu se fait en 2 passes :

1. La triangulation ainsi que la texture contenant l'image de base sont envoyées au vertex shader puis les primitives (triangles) sont rastérisés. Dans le fragment shader, il faut appliquer une somme atomique sur les composantes RGB de chaque pixel

d'un même triangle (`gl_PrimitiveID`) et mettre à jour le nombre de pixels contenus dans le triangle.

2. Pour chaque pixels dans le fragment shader, affecter la couleur accumulée lors de la première passe puis la diviser par le nombre de pixels contenus dans le triangle. Ainsi chaque pixels d'un même triangle a la même couleur.

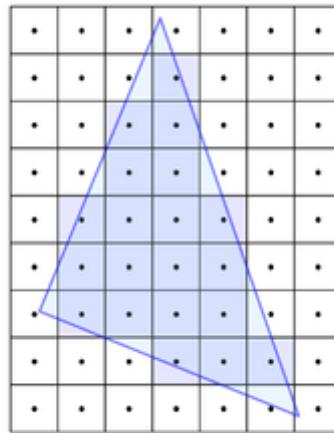


FIGURE 3.6 – Rastérisation

Il faut aussi penser à discréteriser la couleur puisque celle ci est exprimée comme un vecteurs de flottants dans les shaders mais l'opérateur de somme atomique d'OpenGL ne peut utiliser que des entiers.

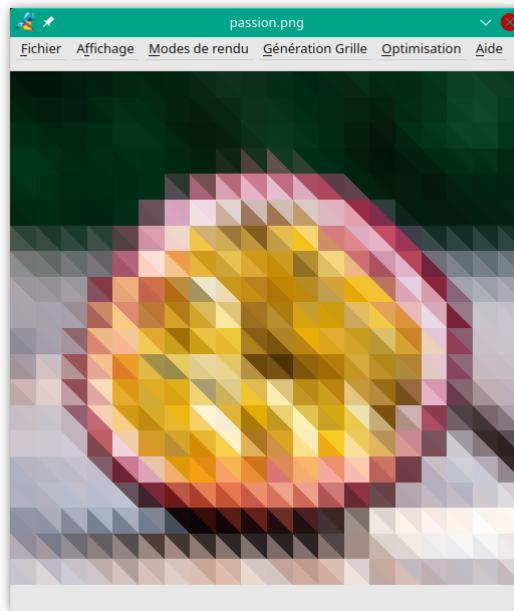


FIGURE 3.7 – Rendu couleur constante

De plus, cette méthode de coloration peut introduire des arêtes qui n'étaient pas présentes dans l'image de base :



(a) Rendu constant



(b) Image de base

FIGURE 3.8 – Exemples des d'arêtes introduites avec le mode de rendu couleur constante.

b. Couleur en gradients linéaires

Le rendu couleur constante donne un certain style abstrait à notre triangulation mais celle-ci est assez loin d'être une approximation satisfaisante de l'image originelle. Cela nous a motivé à implémenter le mode de rendu couleur gradients.

Tout comme pour la couleur constante, il s'effectue en 2 passes mais il nécessite d'effectuer des calculs sur le CPU, en C++, à partir d'informations accumulées dans le fragment shader. Il nous a donc fallu recourir aux Shader Storage Buffer Object ([SSBO](#)).

La couleur finale en un point de l'image triangulée sera alors donnée par :

$$f(x, y) = ax + by + c$$

pour déterminer les coefficients a , b et c il faut alors résoudre le système linéaire suivant :

$$\underbrace{\left[\sum_{i=1}^q \begin{pmatrix} x_i^2 & x_i \cdot y_i & x_i \\ x_i \cdot y_i & y_i^2 & y_i \\ x_i & y_i & 1 \end{pmatrix} \right]}_A \underbrace{\begin{pmatrix} a \\ b \\ c \end{pmatrix}}_x = \underbrace{\left[\sum_{i=1}^q \begin{pmatrix} x_i \cdot I(x_i, y_i) \\ y_i \cdot I(x_i, y_i) \\ I(x_i, y_i) \end{pmatrix} \right]}_b$$

Avec x_i, y_i : les coordonnées 2D du i -ème fragment, q : le nombre de fragment contenus dans le triangle courant et $I(x, y)$: l'intensité lumineuse de l'image au point $[x, y]$.

Si $\det(A) \neq 0$ alors nous pouvons utiliser une [décomposition de Cholesky](#) pour résoudre ce système linéaire. Si A n'est pas singulière, nous appliquons à la place la couleur moyenne au triangle.

Pour effectuer cette décomposition, nous avons utilisé la librairie [Eigen](#).

Ainsi, nous pouvons décomposer notre implémentation en 3 parties :

1. Calcul de la matrice A et des vecteurs b (*un pour chaque composante*) dans le fragment shader à l'aide des fonctions de somme atomique d'*OpenGL*.
2. Récupération sur le CPU, pour chaque triangle de la matrice A et des vecteurs b calculés dans la première passe grâce à un *SSBO*. Nous appliquons ensuite une

décomposition de Cholesky sur cette matrice avant de résoudre le système linéaire pour chacun des b (*un par composante*) avec la même décomposition de A .

Cela nous donne les coefficients a, b et c pour les composantes R, G et B . Ces coefficients sont par la suite envoyés au fragment shader pour effectuer la seconde passe.

3. Nous avons désormais pour chaque triangle les coefficients a, b et c . La couleur d'un fragment est alors donnée par : $f(x, y) = c = ax + by + c$.

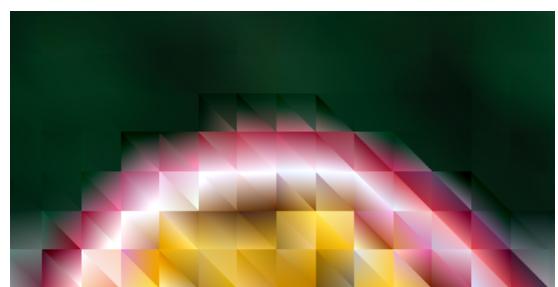


FIGURE 3.9 – Rendu couleur gradients linéaires

Nous pouvons observer une meilleure approximation de l'image de base avec ce mode de rendu puisque la couleur d'un pixel varie en fonction de sa position au sein d'un même triangle à la différence de la couleur moyenne :



(a) Rendu constant



(b) Rendu gradients linéaires

FIGURE 3.10 – Différences entre la couleur moyenne et la couleur gradients linéaires.

6. Études documentaires

a. Image Smoothing via L_0 Gradient Minimization

Nous aurions pu améliorer notre triangulation en proposant d'appliquer au préalable le traitement décrit dans cet [article](#) sur notre image.

La méthode de traitement d'image présentée dans l'article a pour but de rendre certains contours de l'image plus nets, abruptes tout en simplifiant les zones avec de faibles variations d'intensité lumineuse. Pour ce faire les auteurs utilisent une technique " L_0 gradient minimization" qui consiste à contrôler le nombre de gradients non nuls et donc des contours ou bien des arêtes lorsqu'elle est appliquée à une image. Ainsi, cette méthode a pour but d'amincir les contours et les bords saillants de l'image pour les rendre plus visuels, cela aide l'utilisateur à distinguer les principales formes contenues dans une image.



(a) Image en entrée



(b) Image en sortie

FIGURE 3.11 – Exemple d'application de la technique décrite dans l'article.

L'image résultante est alors plus abstraite, comme elle contient moins de variations d'intensités lumineuses elle pourrait permettre d'avoir une triangulation plus précise.

b. A Remeshing Approach to Multiresolution Modeling

Cette section étudie l'article *A Remeshing Approach to Multiresolution Modeling*, par Mario Botsch et Leif Kobbelt. Cet article propose une nouvelle manière d'effectuer un remaillage d'un modèle 3D, et explique comment le maillage généré par cette technique permet d'effectuer de la modélisation multirésolution d'une manière à la fois plus simple et plus efficace.

Le remaillage proposé est un processus itératif basé sur des arêtes. Celles trop longues sont partagées en 2, et celles trop courtes sont supprimées en fusionnant leurs sommets. À ceci sont ajoutés des edge flips afin de rapprocher au plus la valence des sommets de 6, et les sommets sont enfin déplacés pour obtenir une topologie plus régulière.

Cette technique seule génère un maillage simplifié, à la forme plus douce que celle de l'original. L'approximation n'est pas exacte et l'article ne mentionne pas de calcul des normales, mais ce n'est pas important dans leur cas d'utilisation. Le but est simplement d'avoir une forme simplifiée du maillage, qui sert de proxy en interne aux modifications de zone désirées par l'utilisateur, et ce maillage répond parfaitement à ce critère, ayant de plus des propriétés géométriques et topologiques permettant une grande stabilité numérique des transformations. Le maillage original est vu comme une différence par rapport au simplifié, et cette différence est réappliquée sur le maillage simplifié après modification afin d'obtenir le maillage final.

Cependant, dans le cas de notre projet, cette technique n'est pas nécessaire. L'effectuer seule reviendrait à supprimer les transformations effectuées par notre optimiseur sur l'ensemble de l'image. De plus, bien qu'offrir des outils de remodélisation à l'utilisateur soit en effet une optique pertinente, il n'est pas nécessaire de générer un maillage simplifié pour effectuer des opérations de zone sur une image purement 2D. Le travail peut s'effectuer directement au niveau des coordonnées qui forment déjà un espace régulier. Et si le besoin d'un maillage support se fait tout de même sentir, nous avons la possibilité de générer facilement un triangulation régulière ou de Delaunay.

Nous pouvons toutefois considérer l'utilité de cette technique dans le cas où nous généraliserions notre projet à des images 3D. Nous entrerions alors dans le cas d'utilisation exact de cet article, et pourrions en effet implémenter cette technique pour permettre des modifications avancées à l'utilisateur. Cependant, étant donné que ces modifications seraient effectuées après l'optimisation (puisque l'optimisation aurait tendance à reconverger vers la forme pré-modification), il est pertinent de se demander si il ne serait pas plus pratique pour l'utilisateur d'exporter directement le maillage 3D généré vers un logiciel spécialisé tel que Blender pour les retouches finales.

c. Image Signature : Highlighting Sparse Salient Regions

Comme nous l'avons vu, pour réaliser une triangulation réussie il est préférable de partir d'une grille basée sur les caractéristiques de l'objet en premier plan. Plus qu'un simple masque basé sur l'extraction de contour, il serait intéressant de réaliser un échantillonage préférentiel à partir d'une carte de saillance qui pourrait donner de meilleures informations sur les objets importants dans l'image par rapport au système visuel humain.

L'[article](#) présenté ici décrit justement une méthode de génération de carte de saillance basée sur un descripteur de l'image appelé *signature de l'image*.

Ce descripteur binaire est défini comme la fonction signe de la Transformée en cosinus discrète de l'image (DCT).

L'image reconstruite permet d'isoler les objets de premier plan et est définie comme la transformation inverse de la signature de l'image dans le domaine spatial. Il reste alors à convoluer un filtre gaussien sur notre image reconstruite au carré (Produit matriciel de Hadamard) pour obtenir notre carte de saillance. Le filtre gaussien est utilisé pour contrebalancer le bruit introduit avec la quantification par la fonction signe.

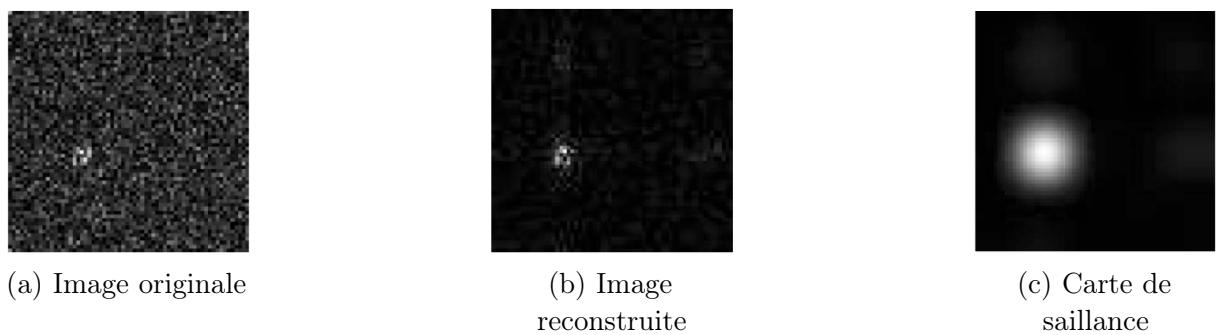


FIGURE 3.12 – Génération de la carte de saillance sur une image en Niveaux de Gris

La génération de la carte de saillance se fait bien dans le domaine RGB en générant une carte par canal.

Finalement les auteurs montrent que leur algorithme surpassé d'autres algorithmes de génération de carte de saillance sur un certain benchmark en faisant de meilleures prédictions sur les points de fixations de l'oeil humain tout en étant bien plus efficace en temps de calcul.

Chapitre 4

Rapport d'activité

1. Rôles dans l'équipe

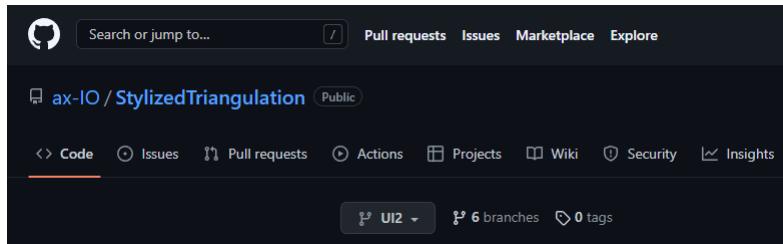
1. Alexandre SPATOLA : Optimisation de la triangulation
2. Alexandre LANVIN : Rendu
3. Gauthier GERMAIN : UI et Génération grille/Triangulation initiale



FIGURE 4.1 – Diagramme de Gantt original

2. Organisation

Au cours de ce projet nous avons communiqué principalement via discord. Ce logiciel nous a permis de nous réunir en vocal ou par écrit pour partager l'avancement de chacun. Nous avons aussi utilisé git pour gérer le versioning du projet, centralisé sur un dépôt [github](#). Chaque membre avait une branche dans laquelle il développait ses fonctionnalités indépendamment de l'avancement des autres.



(a) Notre projet git



(b) Discord

3. Difficultés rencontrées

Nous avons d’abord persisté pour utiliser la librairie [GSL](#) comme on nous l’a conseillé mais nous ne sommes pas arrivés à faire en sorte que notre projet Qt soit portable sur différents systèmes d’exploitation avec celle-ci. Nous avons donc opté pour la librairie [Eigen](#) qui, quant à elle est très légère et permet de conserver la portabilité du projet.

Bien que les tâches fondamentales des parties optimisation et rendu aient été complétées dans les temps, elles se sont révélées plus délicates à finaliser que ne le laissait entendre l’article sur lequel nous avons basé nos travaux. Des cas particuliers introduisaient de l’instabilité numérique qui a dû être gérée. Ces parties se basant sur l’utilisation d’une API graphique, elles étaient également plus difficile à tester et déboguer.

Cette charge de travail supplémentaire, ajoutée à celle induite par le reste du Master, s’est traduite par une prise de retard sur ces parties. Nous avions cependant prévu du temps supplémentaire sur nos tâches afin de prendre en compte de telles possibilités. Comme Gauthier a de plus fini ses tâches plus tôt que prévu, nous avons pu lui redistribuer certaines tâches, notamment celles sur la génération de grille initiale non régulière.

Chapitre 5

Conclusion

1. Bilan

Malgré les difficultés rencontrées, nous sommes arrivés à une version satisfaisante du projet comparé à ce que nous voulions avoir : Proposer une application permettant une triangulation automatique d'images importées tout en laissant du contrôle à l'utilisateur sur les paramètres de génération de cette triangulation ainsi que sur les modes de rendus.

Nous en retirons que pour obtenir une triangulation satisfaisante à l'oeil, une triangulation automatique telle que décrite dans l'article ne semble pas être suffisante et a besoin d'être raffinée par l'utilisateur.

Nous sommes aussi satisfait de ce projet dans sa globalité qui a été une application directe des notions vues en cours avec M. PUECH (Analyse et traitement d'images) et Madame FARAJ (Modélisation, Rendu 3D) tout en nous donnant la possibilité d'aller plus loin.

2. Perspectives d'améliorations

Avec plus de temps ou moins de projets en parallèle nous aurions pu améliorer notre logiciel en corrigeant des bugs produisant des artefacts lors de l'optimisation sur un rendu gradients linéaires. Aussi, d'autres modes de rendu pourraient être ajoutés comme les [tonal art maps](#) qui ajoutent un caractère encore plus abstrait à l'image.

L'UI peut aussi être perfectionnée en ajoutant des fonctionnalités telles que le zoom ou bien la capacité de naviguer dans l'image pour sélectionner plus facilement des points. Il peut être envisagé de pouvoir donner la possibilité à l'utilisateur de pouvoir exporter son image dans un format vectoriel comme le svg.

Le programme pourrait également être étendu pour gérer des cas plus larges, tels que des vidéos et des images 3D.