université de BORDEAUX

Projets de fin d'études

Optimisation du 4D Gaussian Splatting pour son utilisation en VR

Kevin JOURDAIN
Louis DELTENRE
Sabry CHASSERAY

Mars 2025

Table des matières

1 - Introduction

Introduite en 2023, le *Gaussian Splatting* est une technique qui permet la création et le rendu de scènes 3D de haute qualité et en temps réel.

Metalograms [1] est une entreprise française spécialisée dans la vidéo volumétrique, utilisant les techniques récentes de *Gaussian Splatting* afin de capturer et visualiser des scènes 3D photoréalistes. L'entreprise travaille sur un prototype permettant de visualiser ces scènes sur des casques VR autonomes, comme le Meta Quest 3. Cependant, la puissance de calcul limitée et la résolution importante de ces appareils rendent difficile l'obtention d'un taux de rafraîchissement suffisant pour permettre une expérience immersive de qualité.

Le *Gaussian Splatting* s'appuie sur une méthode de discrétisation de l'espace à partir de photographies pour produire un nuage de « tâches » plus ou moins grandes et plus ou moins nombreuses (généralement nombreuses, magnitude de 10^5 - 10^6). C'est ce grand nombre d'éléments individuels avec des caractéristiques propres à chacun qui pousse le matériel à ses limites, en particulier lorsqu'il est question de mouvement fluide et de hautes résolutions de rendu comme c'est le cas en réalité virtuelle.

L'objectif du projet est donc d'optimiser le *viewer* Meta Quest 3 afin de permettre une visualisation de scènes 3DGS et 4DGS d'un million de *splats* (ou qualité équivalente si réduction du nombre de *splats*) avec un nombre d'images par secondes suffisant. Meta [2] fixe le nombre minimum d'IPS à 72 afin de publier une application sur leur *store*, c'est donc l'objectif que nous nous sommes fixés.

2 - État de l'art et Existant

2.1 - Le *3D Gaussian Splatting*

Le *3D Gaussian Splatting* est une technique récente, présentée pour la première fois en 2023 dans l'article *3D Gaussian Splatting for Real-Time Radiance Field Rendering* [3]. Cette technique permet de créer des scènes 3D de haute qualité et adaptées à du rendu en temps réel à partir d'un ensemble de photographies. Contrairement à la modélisation polygonale classique, cette technique représente les scènes 3D comme un nuage de « taches » ellipsoïdales disposées dans l'espace. Le terme « *Gaussian* » vient du fait que l'opacité de ces tâches suit une fonction gaussienne.

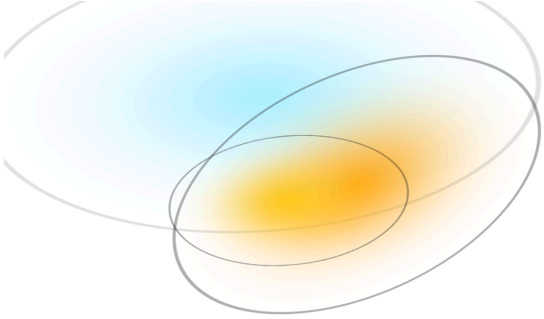


Fig. 1. – Un groupe de trois *splats* gaussiens [4].

Fig. 2. – Une scène constituée de 7 millions de gaussiens, dont l'opacité a été rendue constante dans l'image inférieure afin de mieux les visualiser [4].

La création de ces scènes se fait selon une méthode inspirée des techniques de *machine learning* : Un nuage de points est d'abord généré à partir d'un ensemble de photos, en utilisant un algorithme *SfM* (Structure from Motion) classique. Ce nuage de point est ensuite utilisé pour initialiser les *splats* gaussiens, et certaines de leurs propriétés (position, couleur). La scène est ensuite dessinée à l'aide d'un algorithme de rendu différentiable, puis comparée aux photographies sources. L'erreur obtenue est ensuite utilisée dans un algorithme de descente de gradient, afin d'optimiser les différents paramètres des *splats* (position, taille, orientation, opacité, couleur, etc..).

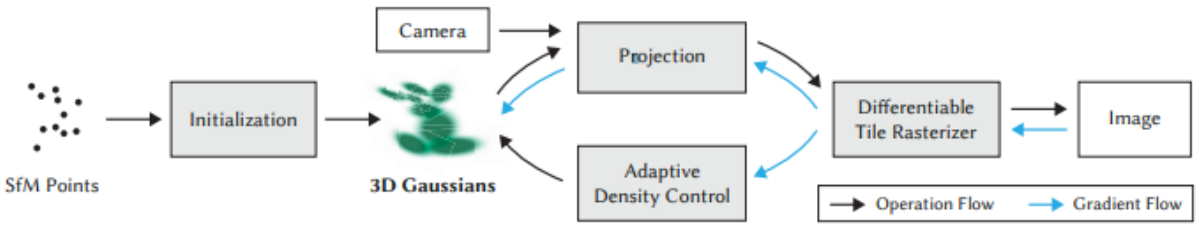


Fig. 3. – Le processus de création de scène de l'article original [3] présentant la technique.

Différentes techniques peuvent être utilisés pour améliorer le résultat de cet entraînement : par exemple, l'article original effectue périodiquement une fusion des *splats* qui convergent vers des positions similaires et une division des *splats* qui deviennent trop importants.

Metalograms utilise également les techniques de *Spacetime Gaussian* [5] afin de proposer des vidéos volumétriques en plus de simples scènes statiques. Avec cette technique, un ensemble de

coefficients polynomiaux sont ajoutés à chaque *splat* afin de représenter l'évolution de certaines de ses propriétés (position, opacité) dans le temps.

2.2 - Visualiseurs existants

Malgré le fait que le *Gaussian Splatting* soit une technique récente, il existe de nombreuses implémentations de visualiseurs de scènes proposant des performances temps-réel, comme par exemple l'application web Antimatter [6]. Cependant, ces applications ont été développées pour des ordinateurs de bureau équipés d'une carte graphique moderne, et sont donc loin des contraintes d'une application VR autonome, comme la faible performance des processeurs graphiques ou le nombre d'image par secondes requises pour éviter un inconfort pour l'utilisateur.

Il existe différents prototypes de visualiseurs disponibles pour la VR [7], [8], principalement en tant qu'extensions pour le moteur de jeu Unity. Ces applications sont cependant limitées. Par exemple, par le besoin d'utiliser un ordinateur externe connecté au casque pour faire le rendu, ou par le faible nombre de *splats* avec lesquels les performances obtenues sont suffisantes. Ces applications ne sont donc pas comparables avec les besoins de Metalograms, qui souhaite proposer des scènes complexes rendues directement par le processeur embarqué du Meta Quest 3.

2.3 - Le prototype de Metalograms

Le logiciel sur lequel nous avons travaillé nous a été fourni par Metalograms. Il s'agit d'un prototype permettant le chargement, la visualisation et la navigation de scènes 3D à l'intérieur du Meta Quest 3. De part la nécessité d'utiliser les processus de développement Android, le logiciel est composé de plusieurs langages, mais la partie du logiciel s'occupant du rendu 3D, sur laquelle nous avons travaillé, est écrite en C++.

Le logiciel proposait également un mode *desktop* permettant de le lancer comme une application classique sur un PC, afin de faciliter le prototypage et le test de nouvelles fonctionnalités et amélioration. Cette version appelle exactement le même code de rendu que la version tournant sur le casque de réalité virtuelle.

Fig. 4. – Capture d'écran de la version de base du logiciel. L'outil de *metrics* au centre de l'écran indique 13 FPS.

La scène fournie en exemple avait été crée à partir du *dataset* « Garden » de Mipnerf360 [9] et grâce à l'outil en ligne de Luma AI dédié à la création de scènes 3DGS [10], depuis abandonné. Tout au long du projet, c'est cette scène qui a été utilisée comme point de repère afin de mesurer les performances.

Avec cette scène comportant 1 119 682 *splats*, le visualiseur atteint 15 FPS, ce qui est très insuffisant si l'on veut proposer une expérience agréable en réalité virtuelle.

3 - User Stories

Le déroulement de ce projet d'optimisation a suivi 3 grandes pistes dont la majorité des tâches découlent :

- **En tant qu'utilisateur, je souhaite que le viewer tourne à $>$ 72 FPS afin de pouvoir interagir confortablement avec les scènes VR fournies.**

Le logiciel de visualisation en lui même présentant une structure ressemblant à celle d'un moteur de rendu 3D classique, les tâches dérivant de cette User Story prennent donc l'allure de tâches d'optimisation d'une pipeline de rendu moderne. Ici il est donc question de paramétrages des différentes étapes de la pipeline OpenGL, d'ajustements/corrections/optimisations des *shaders* et de modification/transformation des méthodes de paquetage et d'envoi des données à la carte graphique.

- **En tant que développeur, je souhaite intégrer des techniques alternative pour optimiser les scènes (en taille et en performance).**

La création de scènes représente l'autre côté de la pièce qu'est le *Gaussian Splatting*, où les optimisations sont moins nombreuses et plus difficiles à implémenter, demandant potentiellement une adaptation de la part du viewer si le format des scènes vient à changer. Optimiser les scènes revient à produire une représentation plus compacte en terme de nombre de *splats*, avec le moins d'informations géométrique et visuelle perdue. De manière générale cette User

Story demande la création de tâches impliquant la création, l'utilisation et/ou la modification de réseaux de neurones.

- **En tant que développeur, je souhaite explorer les méthodes d'upscaling.**

La création et la visualisation de scènes représentent le coeur du *Gaussian Splatting* mais des méthodes de traitement de l'image peuvent venir s'intégrer à la pipeline et l'étendre pour obtenir des gains en qualité et/ou en performance. L'objectif est donc de proposer des extensions, impactant le moins possible l'architecture originale, afin de permettre au matériel graphique de moins travailler ou de compenser le manque de qualité par des méthodes de traitement de l'image tout en corrigeant d'éventuelles erreurs dans des parties de l'image.

4 - Architecture

Le logiciel fourni par Metalograms étant un simple prototype dont le rôle est de permettre d'évaluer les performances du *Gaussian Splatting* en VR, son architecture est très simple : Une partie du code effectue l'initialisation de l'application et API graphiques en fonction de la version (Desktop / Meta Quest 3), le moteur de rendu est initialisé avec les données de la scène et l'application rentre dans une boucle qui consiste à lire les différents *inputs* (clavier, manettes du casque, mouvements de la tête) et à générer une nouvelle image en conséquence.

Le rendu d'une nouvelle image suit les étapes suivantes :

- Si nécessaire, les *splats* sont re-triés en fonction de leur distance à la caméra. Il est nécessaire que les splats soient triés en fonction de la distance à la caméra car le rendu d'une scène *Gaussian Splatting* utilise des opérations de mélange (*blending*) de millions de polygones transparents. Ces opérations n'étant pas commutatives, il est important que les *splats* soient dessinés dans l'ordre. Un nouveau tri des splats est nécessaire lorsque la caméra s'est trop éloignée de sa position par rapport à celle qu'elle occupait au moment du dernier tri. Cette opération entraîne une saccade lors du rendu, mais son optimisation ne faisait pas partie du projet car l'accent avait été mis sur le nombre de FPS, que ces saccades ne se produisaient que lors de mouvements dans l'espace et que des implémentations de tri rapides (par exemple accéléré sur GPU) existaient déjà dans la littérature.
- Les informations de la caméra virtuelle sont mises à jour sur la carte graphique.
- Un appel à l'API graphique pour dessiner les *splats* est lancé. Les *splats* sont dessinés en envoyant leurs propriétés (position, orientation, couleur, etc..) à un *geometry shader* sous forme d'attributs de points. Le *geometry shader* utilise ces attributs pour calculer un polygone correspondant à la forme du *splat* projetée en espace écran. Ce polygone est ensuite récupéré par un *fragment shader*, qui calcule la couleur et la transparence du splat en tout pixel de ce polygone et dessine celui-ci dans le *framebuffer*, en effectuant le mélange avec la couleur déjà présente en fonction de la transparence.

Ces étapes sont répétés un nombre maximal de fois par secondes et constituent le cœur de l'application.

5 - Réalisation

De part la nature du projet, deux pistes d'optimisation se sont naturellement présentés :

- Une optimisation du code de rendu, afin de dessiner plus rapidement la scène et augmenter le nombre d'images produites par secondes.
- Une optimisation des scènes : Étant une technique récente, la recherche en *Gaussian Splatting* progresse rapidement et de nouvelles méthodes de création et d'entraînement de scènes dont

développées. Ces méthodes pourraient permettre de représenter les scènes plus efficacement, avec un nombre réduits de *splats*, tout en conservant la même qualité.

5.1 - Optimisation du rendu

5.1.1 - Remplacement du *Geometry Shader*

Après avoir examiné le code fourni, notre attention s'est portée sur l'utilisation d'un *geometry shader*. Ce type de *shader* (programme GPU) permettant de générer des polygones à dessiner directement depuis la carte graphique est notoirement inefficace, notamment sur des cartes graphiques de type mobiles comme celle qui équipe le Meta Quest 3. Le guide développeur de ARM concernant ses puces graphiques [11] indique par exemple « Before using geometry shading, keep in mind that tile-based GPU architecture is sensitive to geometry bandwidth levels. » et met même en garde contre l'utilisation exacte qui est faite du *geometry shader* dans notre cas : « Arm recommends that you do not use [..] Geometry shaders to expand points to quads. Instance a quad instead. »

Une modification a donc été mise en place pour remplacer le *geometry shader* par une technique d'instanciation de polygone, comme indiqué dans le guide du fabriquant ARM.

Malheureusement, les performances de cette version se sont avérés moindres par rapport à celle utilisant le *geometry shader*. Il semblerait que l'utilisation d'un *geometry shader* dans le cas du *gaussian splatting* soit préférable à des techniques alternatives, et ce pour plusieurs raisons :

- Le *geometry shader* émet directement des polygones entiers, et permet donc facilement de choisir de ne pas en émettre dans certains cas, comme par exemple si la position du splat se situe en dehors du champ de vision. Les *vertex shaders*, utilisés dans notre implémentation alternative, agissent individuellement sur les points du polygone et ne permettent donc pas de facilement le supprimer dans ces cas, encombrant les étapes suivantes de la pipeline graphique avec des points superflus.
- Le *geometry shader* est exécuté une fois par *splat* et produit un polygone constitué de 4 points. Ainsi, les calculs et transformations qui permettent de passer de la position d'un splat en espace monde aux différentes positions de points du polygone en espace écran sont effectués une unique fois par *splat* : le *shader* calcule les vecteurs correspondants aux deux axes majeurs et mineurs du *splat*, et crée 4 points placés sur ces axes. Remplacer le *geometry shader* par un *vertex shader* fait basculer ces transformations sur chaque point du polygone, et multiplie donc par 4 la charge totale de la carte graphique, chaque point effectuant donc séparément et de manière redondante le calcul des axes.

Le rendu de scènes de *gaussian splatting*, avec ses millions de polygones transparents dont la position en espace écran doit être recalculée à chaque image, constitue donc une opération très différente des techniques de rendu polygonaux classiques. L'utilisation d'un *geometry shader* est donc spécialement adaptée à ce type de rendu, malgré les désavantages de ce type de *shader* lors d'une utilisation plus « classique ».

5.1.2 - Forme de polygone utilisée

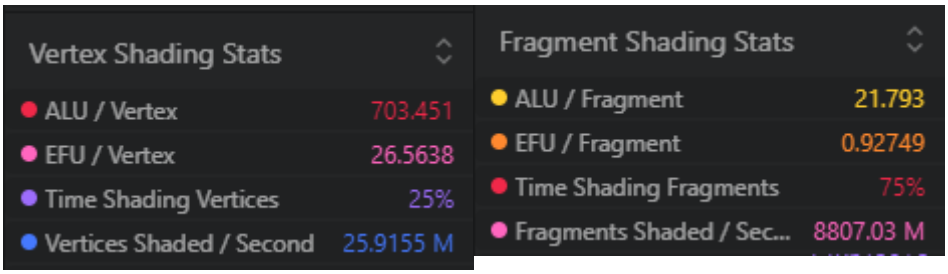


Fig. 5. – Extrait des statistiques d'utilisation GPU récolté durant l'exécution du programme sur le casque VR. Il est à noter que le *geometry shader* est considéré par le profileur comme appartenant à la même catégorie que les *vertex shaders*.

Une analyse avec le profileur intégré au *Meta Quest Developer Hub* lors de l'éxécution du visualiseur révèle que les transformations et calculs décrits précédemment et exécutées durant le *geometry shader* ne constituent que 25% du temps d'utilisation du processeur graphique. Le reste du temps passé à attendre les résultats de la carte graphique correspond à du temps passé dans le *fragment shader*, dont le rôle est de calculer et d'écrire la couleur des pixels recouverts par chaque *splat*.

Le *fragment shader* utilisé comporte cependant très peu d'opérations :

```
in vec4 gColor;
in vec2 gPosition;
out vec4 fragColor;

void main() {
  float A = -dot(gPosition, gPosition);
  float B = exp(A) * gColor.a;
  fragColor = vec4(B * gColor.rgb, B);
}
```

Celui-ci se contente de calculer la transparence en fonction de la distance par rapport au centre du *splat*, puis d'écrire la couleur obtenue. On peut donc sans risque supposer que le la complexité du *fragment shader* n'est pas la source de tout ce temps passé dans cette étape, mais que ce temps provient de la nécessité de mélanger la couleur obtenue avec celle déjà présente dans l'image afin d'obtenir l'effet de transparence des *splats*, étape du pipeline graphique qui n'est pas programmable et pour laquelle les options de configuration à disposition du développeur sont limitées. De plus, ce *fragment shader* est très sollicité : comme discuté dans le Chapitre 5.2, il n'est pas rare pour la scène fournie de comporter plusieurs milliers de *splats* contribuant au même pixel, ce qui entraîne un travail considérable pour le fragment *shader*, exécuté plusieurs milliers de fois pour de tels pixels.

Afin d'améliorer les performances, il semble donc indispensable de diminuer le nombre d'appels au *fragment shader*.

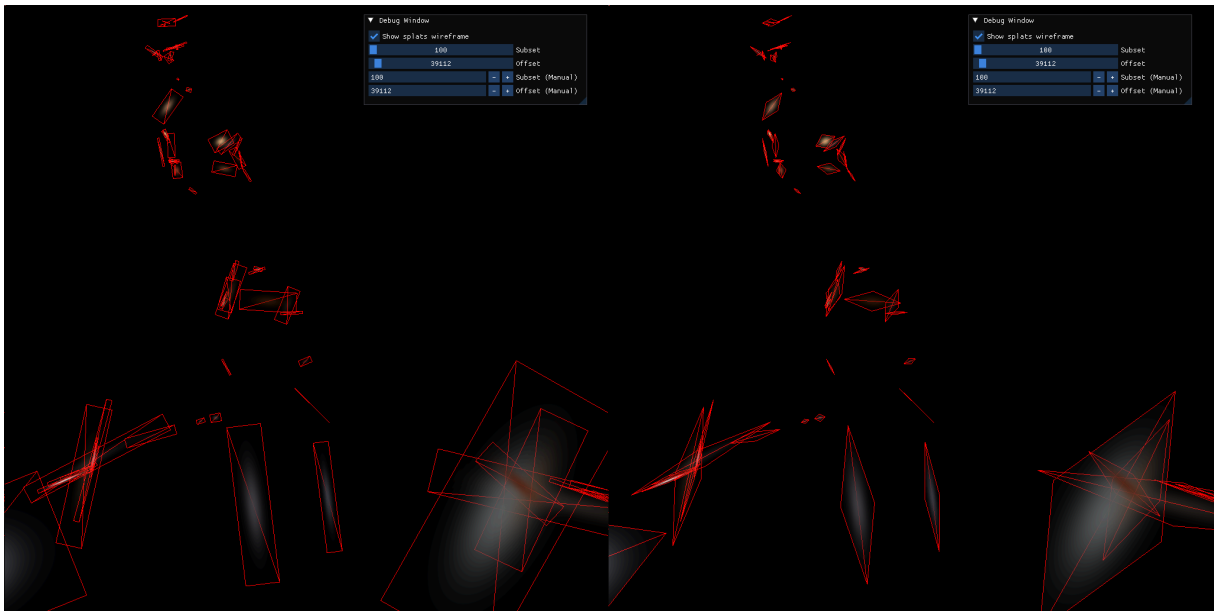


Fig. 6. – Visualisation des *quads* (polygones) utilisés lors du rendu d'un sous-ensemble de 500 points de la scène. À gauche, le *quad* original, et à droite le nouveau *quad*.

En visualisant les *quads* utilisés pour le rendu des *splats*, on observe que une partie importante de la surface est transparente, en particulier dans les coins. Chacun des pixels appartenant à cette surface cause une invocation du *fragment shader* inutile, qui calculera la couleur et effectuera le *blending* sans avoir aucune influence sur l'image finale.

Afin de réduire considérablement le nombre d'invocation du *fragment shader*, nous avons modifié la façon dont le quadrilatère était crée dans le *geometry shader* : au lieu d'être les médianes du *quad*, les axes du *splat* sont maintenant ses diagonales.

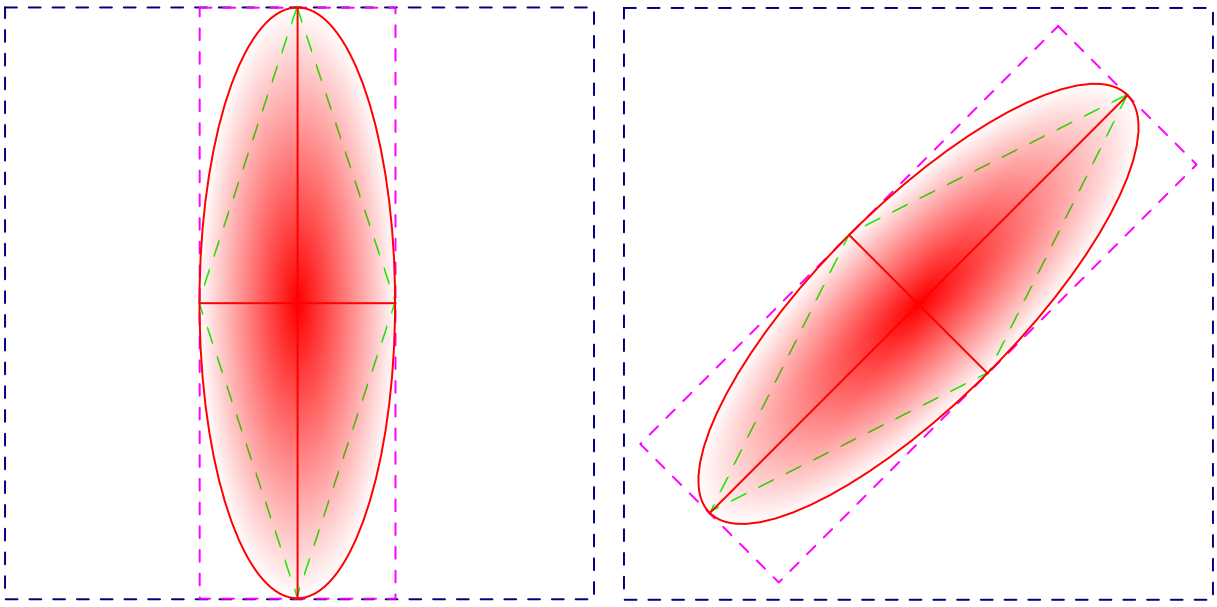


Fig. 7. – Visualisation des formes de *quad* : en violet la forme originale, dont les coins entraînent des invocations inutiles du *fragment shader*, et en vert la nouvelle forme, entièrement contenue à l'intérieur du *splat*.

Cette forme de *quad* entraîne une réduction de 50% de la surface dessinée, et cette réduction se fait quasi-exclusivement dans les zones qui étaient à l'origine entièrement transparentes.

Cette réduction du nombre d'invocations du *fragment shader* se traduit par une augmentation des FPS de 55%, passant de 13 à 21 FPS sur le casque. Il est important de noter que cette augmentation semble se maintenir et reste proportionnelle aux performances « de base » : par exemple, en réduisant la résolution de rendu sur le casque afin d'augmenter artificiellement le nombre de FPS, on passe de 24 à 37 FPS ($\approx +55\%$).

Malgré cette augmentation significative des performances, cette optimisation n'est pas sans compromis : la réduction agressive de la surface de *quad* élimine une partie du rebord du *splat*, ce qui entraîne une altération minime du rendu.

Fig. 8. – Comparaison d'un détail d'une *frame*. À droite la forme originale, et à gauche la forme optimisée. On observe un affinement des formes et un *aliasing* plus prononcé.

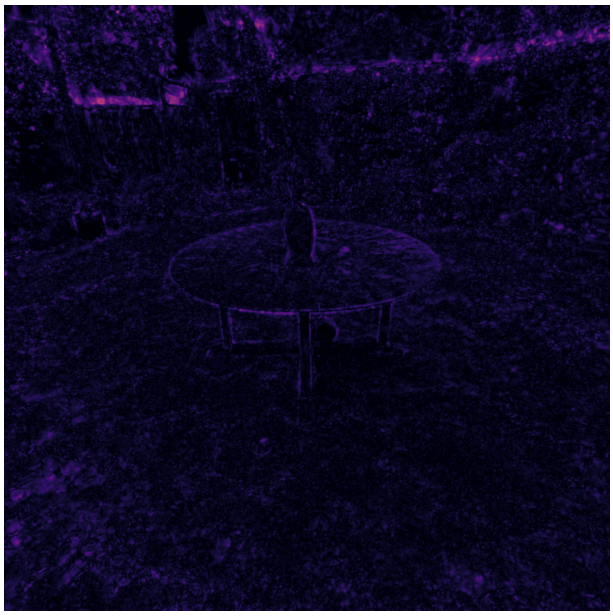


Fig. 9. – Visualisation de l'erreur due à la forme de *splat* optimisée, obtenue par la méthode FLIP [12]. On remarque une perte de « matière » sur les bords des objets, comme sur le vase ou la table, ainsi que dans la végétation. L'arrière plan semble comporter une zone importante d'erreur, mais celle-ci est à ignorer, car constituée de larges *splats* grossiers qui ne sont pas représentatifs du niveau de détail général de la scène.

Cette dégradation visuelle est cependant à relativiser, car très difficile à percevoir sans comparer directement des images superposées, et impossible à deviner sans comparaison. De plus, l'affinement des silhouettes produit permet de contrer l'aspect « flou » que les scènes de *Gaussian Splatting* peuvent parfois produire, et pourrait donc être considérée comme positif pour certains utilisateurs. Une analyse poussée et quantifiable des différences visuelles au niveau du rendu dues aux formes de *splats*, par exemple à l'aide d'un questionnaire utilisateur, est cependant hors de portée pour ce projet.

5.1.3 - Utilisation de *interleaved rendering*

Ici, on a cherché a optimiser le nombre d'images par seconde en utilisant une technique d'*upscaling* : l'*interleaved sampling* [13].

L'*upscaling* est le fait de dessiner une image dans une résolution plus faible afin de limiter le coût en performances, puis d'appliquer une transformation en image plus haute résolution. La façon la plus simple d'implémenter cette technique est d'utiliser un *framebuffer* qui sera de taille X% de la résolution finale visée et de passer a la résolution visée en étirant simplement avec un filtrage bilinéaire. Cette optimisation avait été implémentée dans le code fourni pour le Meta Quest 3, dans lequel la taille du *framebuffer* était initialisée à 75% de sa résolution maximale. Cette optimisation permettait de facilement faire passer les FPS de 15 à 25, mais possède un impact trop important sur la qualité de l'image.

L'*interleaved sampling* est à la base une technique d'anti-aliasing dont l'idée principale est d'entrelacer des échantillons de plusieurs grilles régulières pour créer un motif d'échantillonnage localement irrégulier.

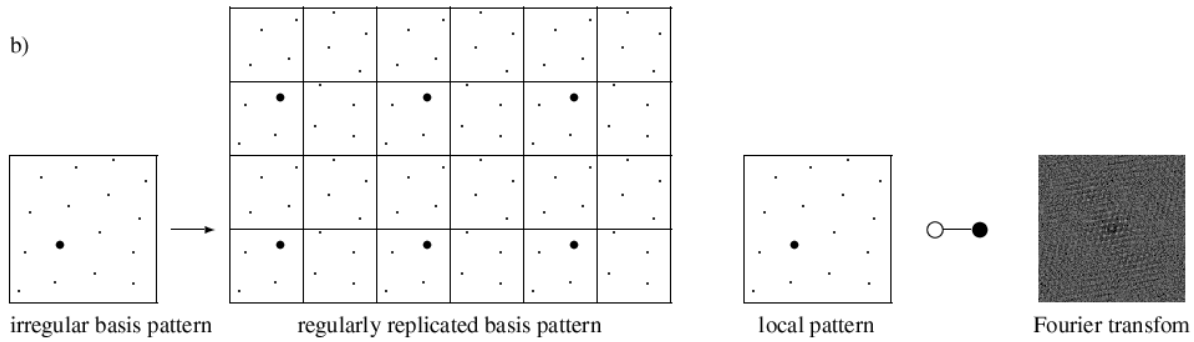


Fig. 10. – [13] Figure1. « La partie b) montre l'entrelacement 2 x 2. Les transformés de Fourier (à droite) démontrent la probabilité significativement réduite d'aliasing pour l'échantillonnage entrelacé. »

Mais ici, on veut utiliser l'*interleaved sampling* pour de la *upscaling*, on adapte donc la technique de cette façon : on crée un *framebuffer* pour résolution réduite, on applique un échantillonnage irrégulier, on effectue ensuite le rendu des *splats* et enfin le *framebuffer* est mis à la résolution visée avec un filtrage bilinéaire.

On peut ici observer et comparer l'impact sur les performance de *interleaved sampling* et d'un filtrage bilinéaire simple.

Interleaved

Resolution Scale	Augmentation des IPS
100%	0%
90%	+1%
80%	+12%
70%	+20%
60%	+26%
50%	+35%

Bilinéaire

Resolution Scale	Augmentation des IPS
100%	0%
90%	+ 3%
80%	+ 9%
70%	+ 23%
60%	+ 27%
50%	+ 41%

Tableau 1. – Comparaison des performances de mise à l'échelle de résolution

Test effectué sur la version desktop du programme

Les performances des deux méthodes sont donc très similaires avec un léger avantage pour la méthode bilinéaire simple, ce qui s'explique par le fait que la méthode *interleaved* possède une étape d'échantillonnage supplémentaire. Si l'on regarde les gains d'IPS obtenus, on remarque que le premier gros gain est obtenu avec une résolution de 70 %, c'est donc la résolution avec laquelle nous avons décidé de faire les tests de qualité. En dessous de 70 % la qualité d'image se dégrade tellement qu'il n'est plus du tout intéressant d'utiliser l'*upscaling*.

On va maintenant comparer la qualité d'une frame avec le *framebuffer* et *interleaved* à 70 %.

Fig. 11. – Comparaison de deux rendus à 70% de résolution : à gauche un filtrage simple, à droite *interleaved*.

On peut observer que la majorité des erreurs vient des hautes fréquences notamment au niveau de la porte ou du feuillage, ces défauts étant communs aux deux méthodes. Pour mieux comprendre la différence entre les deux méthodes, observons une version zoomée.

Fig. 12. – À gauche on peut observer la version *interleaved* 70% qui se caractérise par sa grille, et à droite la version upscalée bilinéairement.

On observe donc que l'image issue d'*interleaved* est beaucoup plus nette sa concurrente, on peut aussi observer facilement les erreur avec *NVIDIA FLIP* [12]

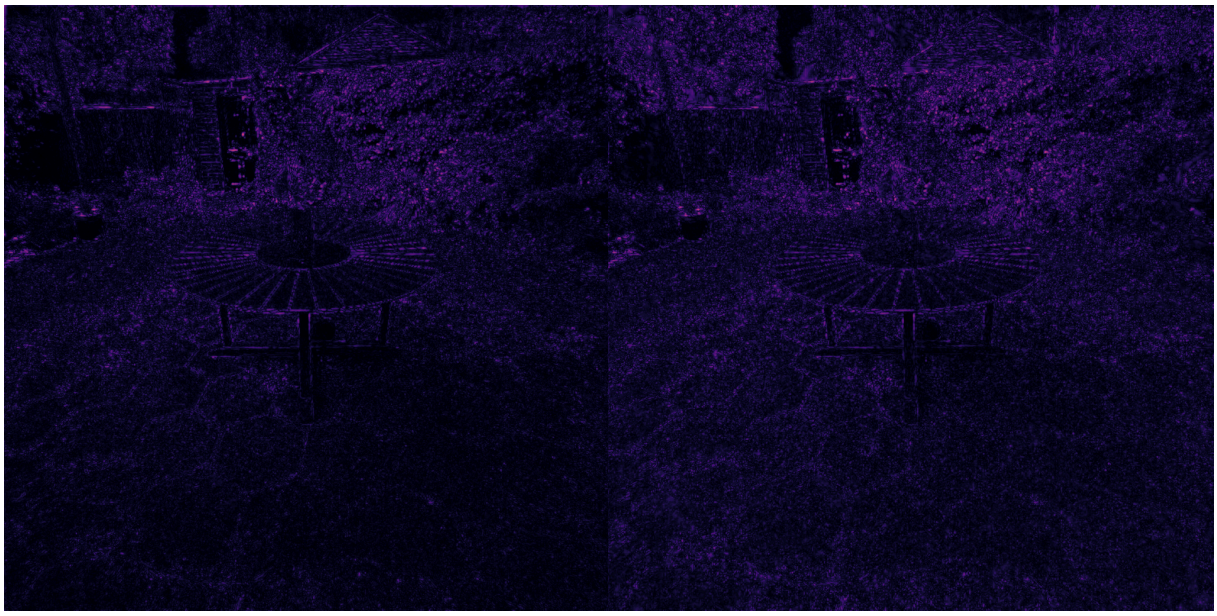


Fig. 13. – À droite une visualisation de l'erreur obtenue lors d'un upscaling bilinéaire simple à 70%, et à gauche celle obtenue par *interleaved*. Les deux images sont comparés à un rendu obtenu à la résolution maximale.

Résultat du FLIP bilinéaire	
Comparison	Résolution 100% vs *framebuffer* 70%
Mean	0.065450
Weighted Median	0.080566
1st Weighted Quartile	0.052254
3rd Weighted Quartile	0.125611
Min	0.000448
Max	0.613058
FLIP Error Map	Figure 13

Résultat du FLIP *interleaved*	
Comparison	Résolution 100% vs. *interleaved* 70%
Mean	0.084911
Weighted Median	0.099933
1st Weighted Quartile	0.069110
3rd Weighted Quartile	0.145663
Min	0.000474
Max	0.616788
FLIP Error Map	Figure 13

Tableau 2. – Comparaison des résultats FLIP entre différentes images

On peut observer que les résultats sont très similaires, les caractéristiques les plus intéressantes à regarder sont *mean* min et max. Les min et max ne présentent pas de différence remarquable, mais les moyennes sont assez différentes ce qui s'explique par l'échantillonnage irrégulier d'*interleaved*. Les scores des deux méthodes peuvent être considérés comme bon, car la moyenne est inférieure à 0.10.

Au niveau des performances et de la qualité graphique les méthodes restent similaires, l'avantage va tout de même à l'*interleaved*, car il maintient une netteté que n'a pas le filtrage bilinéaire simple.

5.1.4 - Calculs avec moindre précision (half floats)

Le gain en performance d'une approche à précision réduite étant en général non négligeable, une piste d'optimisation utilisant des half-floats a été sérieusement considéré. La simple modification de la précision au sein des shaders (avec `precision mediump float` pour des floats en 16-bits) semble suffir pour obtenir un gain de performance sur desktop mais ce n'est pas le cas pour le Meta Quest 3. Le halo n'est pas présent et les performances sont identiques, aucun changement n'est observé sur le casque avec cette modification, c'est ce pour quoi il a fallu considérer à convertir une partie de la pipeline de rendu en floats, et plus en particulier le code qui envoie les données relatives aux splats à la carte graphique.

Deux pistes d'implémentations sont envisageables:

- Utiliser une implémentation existante des floats à moindre précision [14]
- Développer une structure 16-bits agissant comme un half-float de l'extérieur, avec les outils de conversion nécessaires (double, float -> half-float)

La première piste semble prometteuse puisque utiliser une partie du standard C++ pour implémenter cette optimisation serait simple et direct. Cependant cette feature n'est implémenté qu'en C++23 et seulement supporté par GCC 13 et EDG eccp 6.4 [15].

Requirements for optional extended floating-point types	P1467R9	13		N/A		6.4				

Fig. 14. – Compiler support pour les nouveaux types flottants dont le float16_t ou encore half-float. Seuls GCC 13 et EDG eccp 6.4 l'ont implémenté

Par soucis de compatibilité et de simplicité, cette piste n'est pas retenu et, avant de considérer la deuxième, d'autres implémentations sont explorées. GLM, faisant déjà parti du projet, est retenu avec son implémentation des half-float à travers `glm::detail::hdata` [16] accompagné des outils nécessaires à la conversion.

Les propriétés relatives aux splats (position, taille, orientation et couleur) et les buffers les contenant sont donc convertis en half-floats avant d'être utilisées par le geometry shader. Cette partie de la pipeline est aussi adaptée puisque OpenGL doit maintenant intéragir avec des half-floats (data buffer, stride, …). Sur desktop, les performances augmentent de presque 30% en zone dense et un peu plus de 40% en zone usuel.

Sur le casque, la scène présente maintenant les artéfacts présents sur la version desktop mais les performances restent malheureusement les mêmes.

Fig. 15. – Scène garden avec l'optimisation complète

Par manque de temps et de connaissance du monde du rendu VR sur mobile, un fix n'est pas proposé.

Initialement l'objectif était d'implémenter un arrêt prématuré des calculs en fonction de la transparence de chaque pixel (une fois une certaine opacité atteinte, interrompre les calculs qui peuvent être très lourds et contribuer à quelques pourcents du rendu final).

Dans une scène typique de Gaussian Splatting, nombreux sont les splats en tout point et chaque splat se présentent différemment, avec une couleur, une taille, une orientation et une opacité propre à lui. Pour tout pixel, un grand nombre de splats est considéré, plus ce nombre est conséquent moins les performances sont bonnes. Cette optimisation a donc pour objectif de réduire le nombre de splats considéré sans trop affecter la qualité du rendu, en interrompant les calculs une fois une certain seuil de transparence atteint.

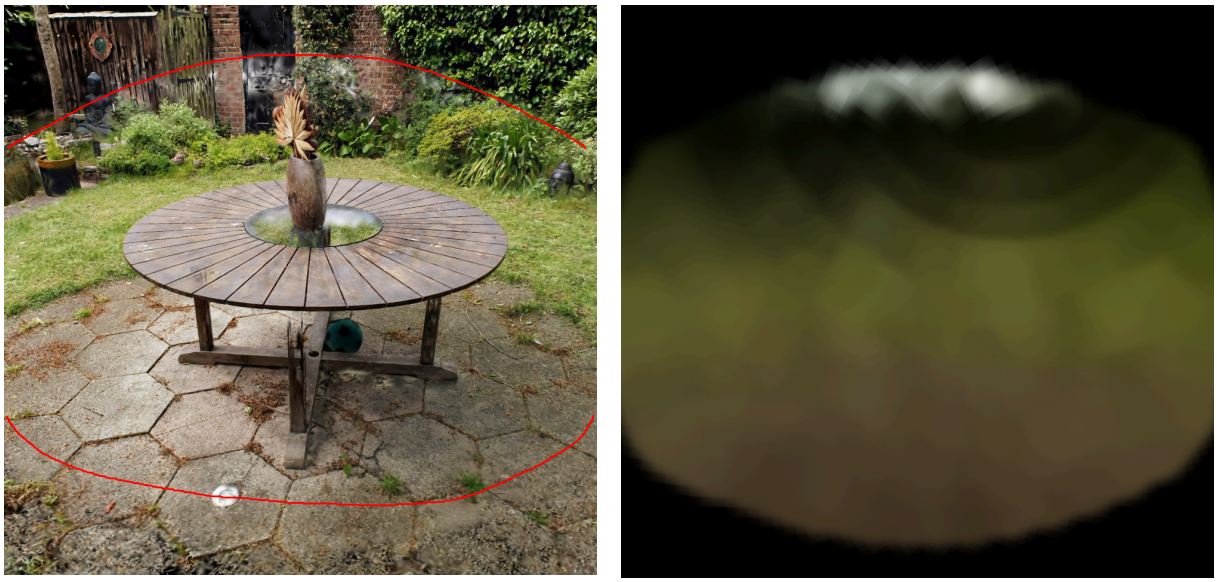
L'implémentation serait la suivante:

- Créer un tableau qui représente l'ensemble des valeurs de transparence de l'image, un élément du tableau = la transparence d'un pixel
- Réaliser l'alpha blending des splats sans modification et stocker à chaque itération la nouvelle valeur de transparence dans le tableau pour le pixel considéré
- Interrompe l'alpha blending et conserver le résultat calculé jusqu'ici comme résultat final pour ce pixel en fonction de la valeur de transparence.

Avant d'implémenter la structure et l'algorithme qui permettraient l'arrêt prématuré des calculs dans les shaders, quelques « sanity checks » ont été réalisés (sur desktop) pour savoir si les efforts d'implémentation seraient récompensés:

- Précision des floats du fragment shader(precision lowp/mediump/highp float) mais sans répercussions remarquables sur les performances
- Discard, ou encore suppression/défaussement, prématuré du fragment en fonction de la taille du splat, avec légère augmentation des FPS, 7%, en zones denses (les plus nombreuses en splats). L'idée étant d'interrompre les calculs en jetant le fragment en considération, s'il est trop petit, pour passer directement au traitement d'un fragment suivant.
- Full discard du fragment shader augmente les FPS d'environ 14% en zones denses

Les résultats n'étant pas tellement fructueux, c'est pourquoi notre intérêt s'est porté sur le geometry shader qui lui est plus complexe: Modifier la précision des floats en lowp ou mediump a permis une augmentation des FPS de 26% au coût de quelques artéfacts visuels (certaines couleurs se blanchissent dans une zone importante de la vision centrale et la qualité du rendu à la frontière de ce halo est réduite).



Halo avec scène Halo sans scène

Fig. 16. – Scène garden avec l'optimisation naïve `precision mediump float` dans les shaders, le halo couvre portion importante de la vision

C'est à travers ces « sanity checks » que les travaux sur une pipeline à moindre précision ont commencé.

5.2 - Optimisation des scènes

Fig. 17. – Rendu de la scène fournie, en ne dessinant que les *splats* dont l'opacité dépasse 10%.

En modifiant légèrement le code afin de ne pas dessiner les *splats* dont l'opacité (α) est inférieure à un certain seuil, par exemple 10%, on observe sur la scène « garden » fournie avec le projet une importante altération du rendu et même certaines zones sans *splats*, en particulier au centre de la scène. On peut donc logiquement supposer que l'agencement et les paramètres des *splats* constituant la scène est loin d'être optimal : la scène est produite en dessinant de nombreux *splats* à la contribution minime, au lieu d'un nombre réduit de *splats* possédant une forte opacité.

De plus, la création de la scène a été effectué grâce à l'outil en ligne de Luma AI, depuis abandonné par cette entreprise. De part le peu d'information disponible à propos de cet outil, il semblerait que l'algorithme d'entraînement de la scène utilisé par Luma AI soit *GSplat* [17], une version accélérée de l'algorithme de la méthode originale. La création de cette scène n'a donc vraisemblablement pas bénéficié des deux dernières années de recherche en *Gaussian Splatting*.

5.2.1 - Mini-Splatting2

Apres quelques recherches, notamment grâce à l'étude de comparaison de techniques *3DGS.zip: A survey on 3D Gaussian Splatting Compression Methods* [18], il s'est avéré que la technique de création de scènes *Mini-Splatting* [19] semblait être la meilleure en terme de réduction du nombre de *splats* utilisés pour représenter une scène (colonne « k Gauss »). Depuis la publication de la meta-étude, une version améliorée (*Mini-Splatting2*) a été publiée, c'est donc celle-ci que nous avons décidé d'utiliser.

Après avoir effectué l'entraînement sur les images *Garden* du dataset utilisé pour créer la scène originale fournie avec le projet, nous avons obtenue la scène équivalente entraîné avec *Mini-Splattting2*. Il est important de noter que les images utilisées sont celles du dossier `images_04`, c'est à dire *downscale* 4 fois afin de réduire la consommation en VRAM durant la création et la taille de la scène finale.

Fig. 18. – Visualisation des *splats* indésirable à l'intérieur de l'éditeur *SuperSplat*.

5.2.1.1 - Défauts

Après examen de la scène ainsi générée, il s'est avéré qu'un important *splat* parasite était présent au centre de la scène et venait dégrader la qualité de rendu. Il semblerait que ce défaut ne soit pas unique aux images utilisées pour la scène *Garden*, et des défauts similaires ont également étés observés sur la scènes *Bicycle*, faisant elle aussi partie du *dataset* de MipNerf360. Une fois ces *splats* supprimés manuellement, le test des performance a pu débuter.

5.2.1.2 - Performances

Sur le Meta Quest 3, cette scène entraînée par *Mini-Splatting2* permet maintenant d'atteindre entre 37 et 60 FPS en fonction de l'endroit de la scène sur le regard de l'utilisateur se porte, contre entre 13 et 15 FPS pour la scène entraînée sur Luma AI.

Caractéristique	Luma AI	Mini-Splatting2	Évolution
FPS Minimales	13	37	× 2.8
FPS Maximales	15	58	× 4
Nb. de *Splats*	1 119 682	730 272	× 0.65

Avec l'optimisation de *splat* décrite dans le Chapitre 5.1.2, on atteint même l'objectif désiré de 72 FPS, et sans jamais passer en dessous de la barre des 60 FPS dans les zones denses.

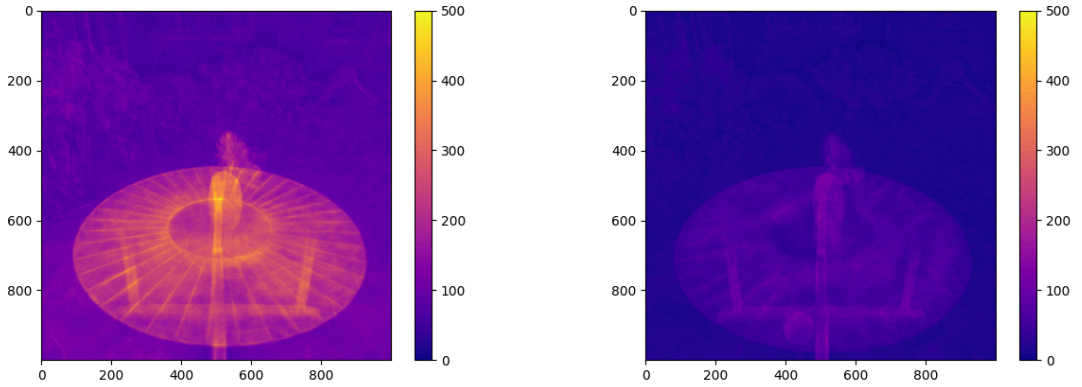
5.2.1.3 - Qualité

Fig. 19. – Comparaison du rendu visuel obtenu sur le Meta Quest 3. La scène originale produite avec Luma AI est à gauche, celle obtenue grâce à Mini-Splatting2 à droite.

Malgré leur nombre réduit de *splats* et leur important gain en performances, les scènes ne semblent pas sacrifier la qualité de rendu : en comparant avec la nouvelle scène *Garden* avec celle fournie avec le projet, on observe un bien meilleur rendu des couleurs avec plus de contrastes et de variété dans les *splats*, ainsi que des formes plus détaillées, en particulier dans les zones géométriquement complexes comme l'herbe au sol ou la végétation. En revanche, l'arrière plan lointain est bien moins détaillé, comme on peut l'observer sur la première paire d'images.

5.2.1.4 - Analyse

On observe que les gains de performances obtenues ne sont pas simplement proportionnelles à la réduction du nombre de *splats* des nouvelles scènes : sur la scène *Garden*, une réduction du nombre de *splats* de 35% a conduit à une augmentation des FPS de 300%. On peut donc logiquement supposer que les scènes entraînées par Mini-Splatting2 ne contiennent pas simplement moins de *splats*, mais que ces *splats* sont également mieux positionnées et paramétrer afin de réduire le nombre de *splats* inutiles ou peu optimisés qui ralentissent le rendu pour une contribution minime à l'image finale.



LumaAI Minisplat2

Fig. 20. – Carte de chaleur comparant le nombre de *splats* superposés contribuant à chaque pixel.

Afin de confirmer cette hypothèse, on peut modifier le moteur de rendu purement logiciel en python fourni par Metalograms afin de produire des *heatmaps* représentant le nombre de *splats* ayant contribué à chaque pixel lors du rendu d'une image. Ces graphiques permettent de se rendre compte que les scènes obtenues grâce à *Mini-Splatting2* sont bien mieux optimisées, nécessitant un nombre bien moins important de couches de *splats* afin de produire un rendu visuel pourtant supérieur.

6 - Conclusion : Bilan et Perspectives

Le *Gaussian Splatting* est une jeune technologie prometteuse déjà très capable avec des applications nombreuses et variées où le photoréalisme est désiré:

- Visualisation architecturale (visite d'appartement ou *streetview* interactive)
- Reconstruction de modèles 3D photoréalistes
- Photoréalisme VR/AR

Le domaine est très prolifique puisque l'on recense plus de 6600 brevets et articles scientifiques depuis 2023 [20].

Cependant cette technique reste immature en particulier lorsqu'il est question de temps-réel, les optimisations sont nombreuses mais pas toujours compatibles entre elle; les formats de scène peuvent varier, d'autant plus quand il est nécessaire d'obtenir une certaine fluidité de visualisation ou taille de fichier; l'état de l'art évolue sans cesse laissant place jour après jour à une nouvelle méthode qui vient améliorer la technique.

Les efforts ont donc été concentrés sur 3 pistes d'optimisation:

- La création de scènes (optimisation du format, de la taille et de la performance des scènes)
- Le traitement d'image (adapter le rendu et transformer l'image finale pour gagner en qualité et performance)
- Le rendu du visualiseur (simplifier les calculs et rendre plus compact le transfert de données)

La piste de l'optimisation de scène avec Minisplat2 s'est avéré être fructueuse, la diminution/densification du nombre *splats* et donc de la taille de la scène s'est répercutée sur les performances. C'est cette piste qui pousse les performances jusqu'aux tant convoitées 72 images par seconde. De part son aspect de densification, des scènes plus lourdes profiteraient éventuellement de plus grands gains, en diminution de taille comme en performance. Cette méthode de création de scènes n'est cependant pas sans inconvénients, comme le besoin d'éditer manuellement ses

résultats, et appelle donc à rester attentif sur la recherche en ce qui concerne les nouvelles techniques qui pourraient corriger ces défauts tout en maintenant l'imporante qualité obtenue.

L'*interleaved rendering*, porte drapeau de la piste d'optimisation traitement d'image, offre des performances similaires à un simple downscale de résolution du framebuffer, avec une qualité légèrement meilleure, malgré la toute nouvelle étape de traitement d'image ajoutée à la pipeline. Une optimisation et/ou un paramétrage plus fins permettrait éventuellement d'obtenir des performances et une qualité encore meilleures.

Les différentes branches de la piste d'optimisation du rendu du visualiseur viennent bousculer la structure et les acquis de la base de code déjà établie pour y introduire des nouvelles perspectives et sentiers d'optimisations. Le remplacement du *geometry shader*, la transformation de la forme des quad, les recherches menées sur l'arrêt prématuré en fonction de l'alpha et la modification de la pipeline pour lui permettre de stocker et d'utiliser des *half-floats* contribuent tous à une meilleure compréhension des chemins chauds (*hotpaths*), des embûches et du futur du *Gaussian Splatting* en temps réel.

La jumelle dynamique du Gaussian Splatting en 3D, le 4DGS ou en d'autres termes la vidéo volumétrique photoréaliste, ne s'est pas vu porter la même attention. Ceci étant dit, la majorité des optimisations apportées au 3DGS peuvent être directement implémentées ou portées. Le 4DGS présente une méthode plus impliquée mais elle peut aussi jouir d'optimisations de rendu et création de scène puisqu'elle présente des chemins chauds similaires.

Le monde du Gaussian Splatting est encore immature et les avancées sont déjà conséquentes, les travaux présentés ici solidifient l'édifice érigé par les chercheurs qui élaborent les techniques de demain.

Bibliographie

[1] « Metalograms ». [En ligne]. Disponible sur: https://www.metalograms.com/

[2] « Meta Quest Virtual Reality Check (VRC) guidelines ». [En ligne]. Disponible sur: https://developers.meta.com/horizon/resources/publish-quest-req

[3] B. Kerbl, G. Kopanas, T. Leimkühler, et G. Drettakis, « 3D Gaussian Splatting for Real-Time Radiance Field Rendering », *ACM Transactions on Graphics (SIGGRAPH Conference Proceedings)*, vol. 42, nº 4, juill. 2023, [En ligne]. Disponible sur: http://www-sop.inria.fr/reves/Basilic/2023/KKLD23

[4] D. Ebert, « Introduction to 3D Gaussian Splatting ». [En ligne]. Disponible sur: https://huggingface.co/blog/gaussian-splatting

[5] Z. Li, Z. Chen, Z. Li, et Y. Xu, « Spacetime Gaussian Feature Splatting for Real-Time Dynamic View Synthesis », *arXiv preprint arXiv:2312.16812*, 2023.

[6] K. Kwok, « antimatter15/splat ». [En ligne]. Disponible sur: https://github.com/antimatter15/splat

[7] CLARTE-LAB, « GaussianSplattingVRViewerUnity ». [En ligne]. Disponible sur: https://github.com/clarte53/GaussianSplattingVRViewerUnity

[8] C. Kleinbeck, « Unity-VR-Gaussian-Splatting ». [En ligne]. Disponible sur: https://github.com/ninjamode/Unity-VR-Gaussian-Splatting

[9] J. T. Barron, B. Mildenhall, D. Verbin, P. P. Srinivasan, et P. Hedman, « Mip-NeRF 360: Unbounded Anti-Aliased Neural Radiance Fields », *CVPR*, 2022.

[10] « Luma AI - Interactive Scenes ». [En ligne]. Disponible sur: https://lumalabs.ai/interactive-scenes

[11] « Arm® GPU Best Practices Developer Guide ». [En ligne]. Disponible sur: https://developer.arm.com/documentation/101897/0304

[12] P. Andersson, J. Nilsson, T. Akenine-Möller, M. Oskarsson, K. Åström, et M. D. Fairchild, « FLIP: A Difference Evaluator for Alternating Images », *Proc. ACM Comput. Graph. Interact. Tech.*, vol. 3, nº 2, août 2020, doi: 10.1145/3406183.

[13] A. Keller et W. Heidrich, « Interleaved Sampling », in *Eurographics Workshop on Rendering*, S. J. Gortle et K. Myszkowski, Éd., The Eurographics Association, 2001. doi: /10.2312/EGWR/EGWR01/269-276.

[14] cppreference, « Fixed with floating-point types ». [En ligne]. Disponible sur: https://en.cppreference.com/w/cpp/types/floating-point

[15] cppreference, « C++23 Compiler support ». [En ligne]. Disponible sur: https://en.cppreference.com/w/cpp/compiler_support#C.2B.2B23_features

[16] glm, « glm type_half ». [En ligne]. Disponible sur: https://github.com/g-truc/glm/blob/master/glm/detail/type_half.inl

[17] V. Ye *et al.*, « gsplat: An Open-Source Library for Gaussian Splatting », *arXiv preprint arXiv:2409.06765*, 2024, [En ligne]. Disponible sur: https://arxiv.org/abs/2409.06765

[18] M. T. Bagdasarian *et al.*, « 3DGS.zip: A survey on 3D Gaussian Splatting Compression Methods », *arXiv preprint arXiv:2407.09510*, 2024.

[19] G. Fang et B. Wang, « Mini-Splatting2: Building 360 Scenes within Minutes via Aggressive Gaussian Densification », 2024, [En ligne]. Disponible sur: https://arxiv.org/abs/2411.12788

[20] « Gaussian Splatting publications since 2023 ». [En ligne]. Disponible sur: https://scholar.google.com/scholar?as_vis=0&q=Gaussian+Splatting&hl=en&as_sdt=2007&as_ylo=2023