REFEYTON
Mattias

Rapport de projet IG3DA

Laplacian Kernel Splatting for Efficient Depth-of-field and Motion Blur Synthesis or Reconstruction

0 Introduction

Dans ce projet, nous nous attachons à implanter deux parties de l'article titre : le calcul et la sparsification du laplacien des PSF, dont les échantillons ont été fournis par les auteurs ; ainsi que l'étape de *splatting* utilisant le schéma pyramidal de convolution par Farbman et. al [1]. Les deux étapes ont été implantées en deux programmes différents : l'un prenant en argument une PSF sous forme d'image et écrivant dans un fichier (par concaténation si existant) les données binaires représentant les points du *spreadlet* correspondant à la PSF ; l'autre lisant le fichier résultant de l'exécution du premier programme sur un ensemble de PSF et l'utilisant pour appliquer l'effet sur une scène 3D.

Les deux programmes ont été écrits en C++ et utilisent Eigen de manière à profiter de la vectorisation des instructions quand cela est possible, en particulier dans le code de convolution et du schéma de convolution pyramidal. L'étape de pré-calcul utilise la bibliothèque *jc_voronoi* pour les calculs liés aux diagrammes de Voronoi. Le rendu utilise OpenGL 4 et les *compute shaders*, à l'initiative des auteurs.

I Pré-calcul (dossier *precompute*)

Pour lancer l'application : make precompute [IN=files] [OUT=prefix] [CX=psf center X] [CY=psf center Y]. Toutes les variables ont des valeurs par défaut, de sorte à ce que 'make precompute' fonctionne.

Il est aussi possible de lancer l'application sur une seule image avec make run [IN1=image] [OUT=prefix] [CX=…] [CY=…]. Ceci est utile pour générer des données de test pour le dossier test, cf partie II.

La première étape de ce pré-calcul consiste à échantillonner le laplacien de la PSF d'entrée selon un disque de Poisson à rayon variable, avec le rayon dépendant de l'inverse de la norme du laplacien filtré. À cette fin, on calcule le laplacien par convolution avec un noyau 3x3, puis le résultat est convolué avec un noyau gaussien 5x5. On tire ensuite un point au hasard sur l'image, qui est rejeté si la distance au point accepté le plus proche est plus grande qu'un *remapping* empirique de l'inverse de la norme du laplacien filtré. On répète cette procédure jusqu'à avoir rejeté 10 000 candidats d'affilée.

Cette première étape sert d'initialisation à la procédure complète. Elle a cependant le désavantage de très mal approximer les PSF avec peu d'extension spatiale ; ainsi, on teste le nombre de pixels non nuls du laplacien originel, et s'il est en-dessous d'une valeur empirique, on sauvegarde chaque pixel comme point du *spreadlet* et la procédure s'arrête ainsi, en une alternative du *fast-track* évoqué dans l'article.

La seconde étape consiste à effectuer 50 relaxations de Lloyd sur les points acceptés, en utilisant de nouveau la norme du laplacien filtré comme pondération, puis à intégrer la valeur du laplacien originel sur les cellules de Voronoi résultantes. À ces fins, et à défaut de ressources en ligne, j'ai écrit deux routines de calcul exact (à la précision de la représentation des nombres flottants près) : l'une intégrant une fonction d'une position entière ρ sur une cellule de Voronoi, et l'autre calculant le centroïde d'une cellule pondéré par une telle fonction ρ – ce qui revient à calculer $\frac{\int_C \vec{x}\,\rho(\vec{x})\,d\vec{x}}{\int_C \rho(\vec{x})\,d\vec{x}}$ sur la cellule C. L'exactitude du calcul est rendue possible par le fait que bien que les valeurs de position soient représentées par des nombres réels, la fonction de pondération ne varie qu'avec la partie entière de la position. Ces deux routines et les sous-routines qu'elles utilisent sont écrites dans *mathutil.hpp*.

La troisième étape consiste en 400 itérations de recuit simulé, où l'état voisin est déterminé par un décalage d'au plus un pixel d'1 % des positions des points du *spreadlet*, et où le coût d'un état est déterminé par intégration du *spreadlet* et comparaison au sens des moindres carrés avec la PSF originelle. Bien que l'intérêt du recuit simulé n'est pas tant la compression que l'optimisation, pour une raison que je n'ai pas encore su déterminer, je n'ai jamais réussi à faire en sorte que cette étape améliore le résultat de l'étape précédente seule sur l'ensemble des PSF. Ci-dessous, une comparaison entre (de haut en bas) la référence, l'image synthétisée avec recuit simulé en pré-calcul, et l'image synthétisée sans recuit simulé.

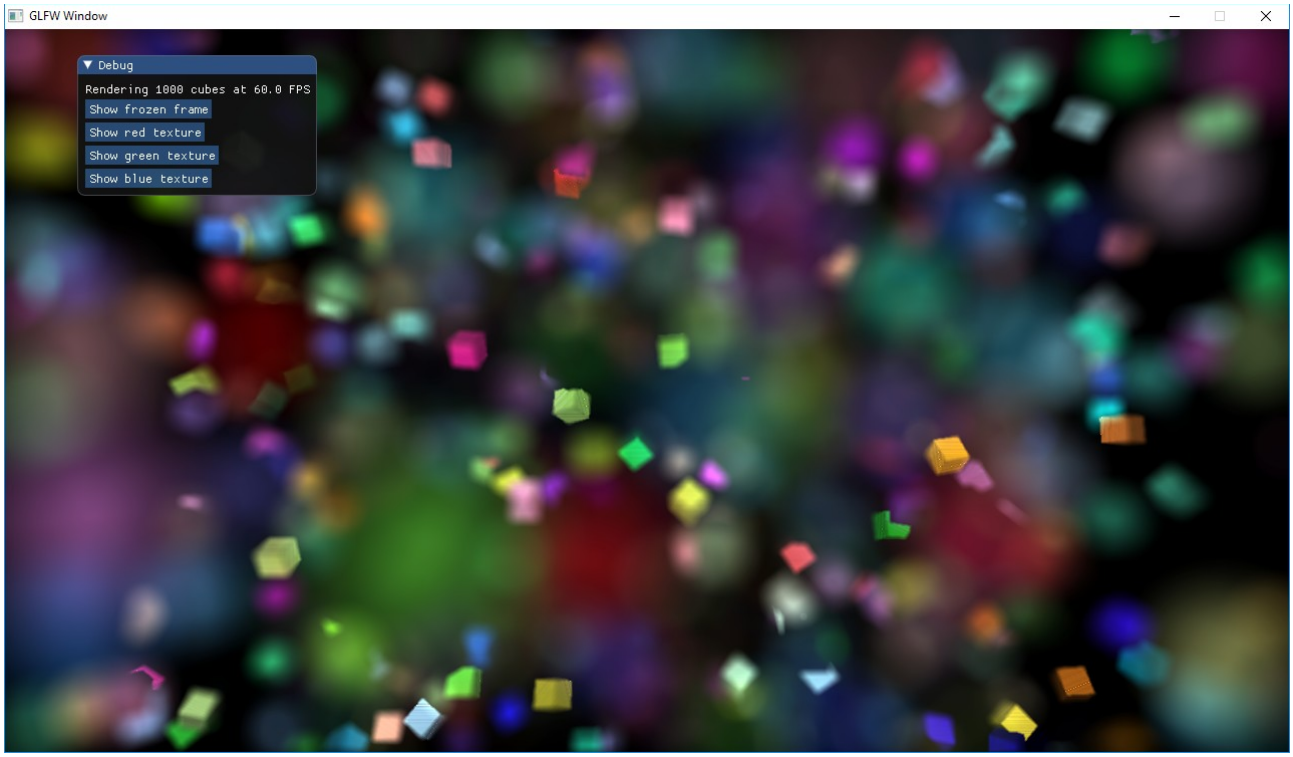


Illustration 1: Référence (pas de sparsification du laplacien)

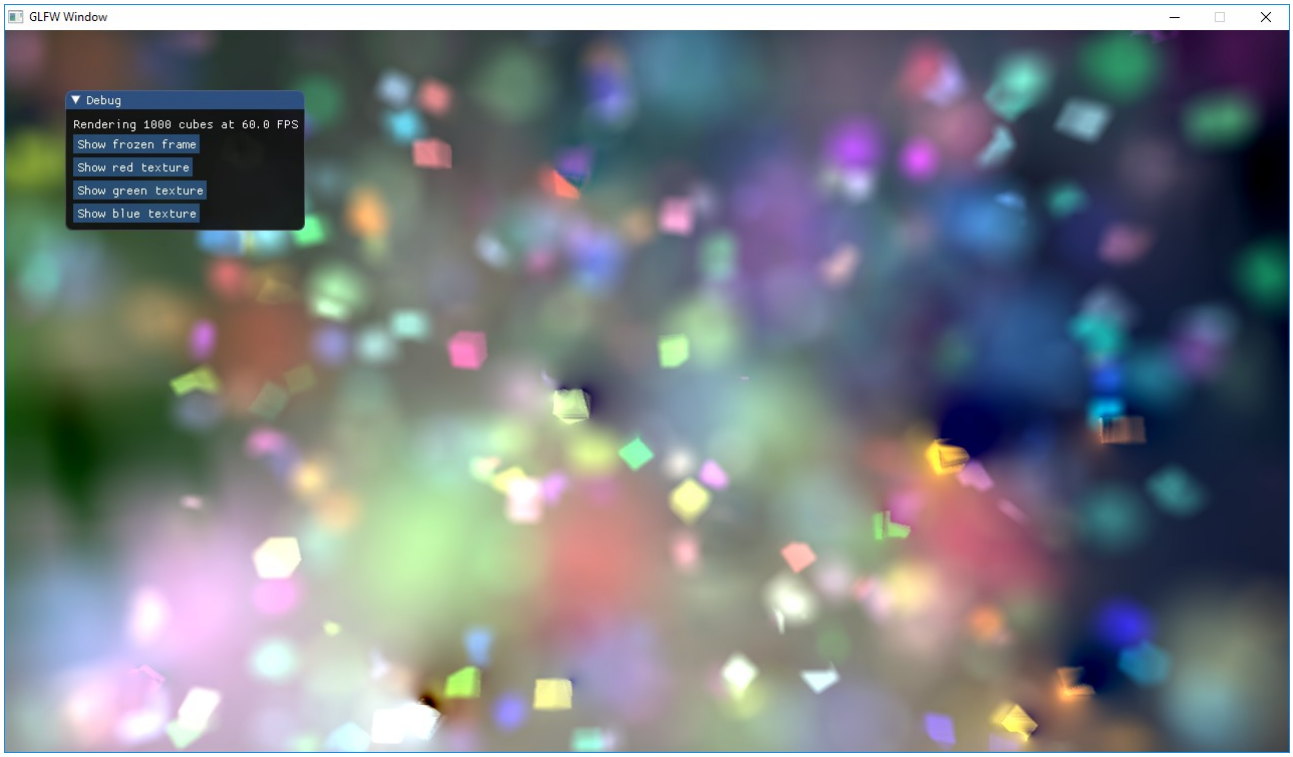


Illustration 2: Sparsification du laplacien avec recuit simulé (68,5% de compression)

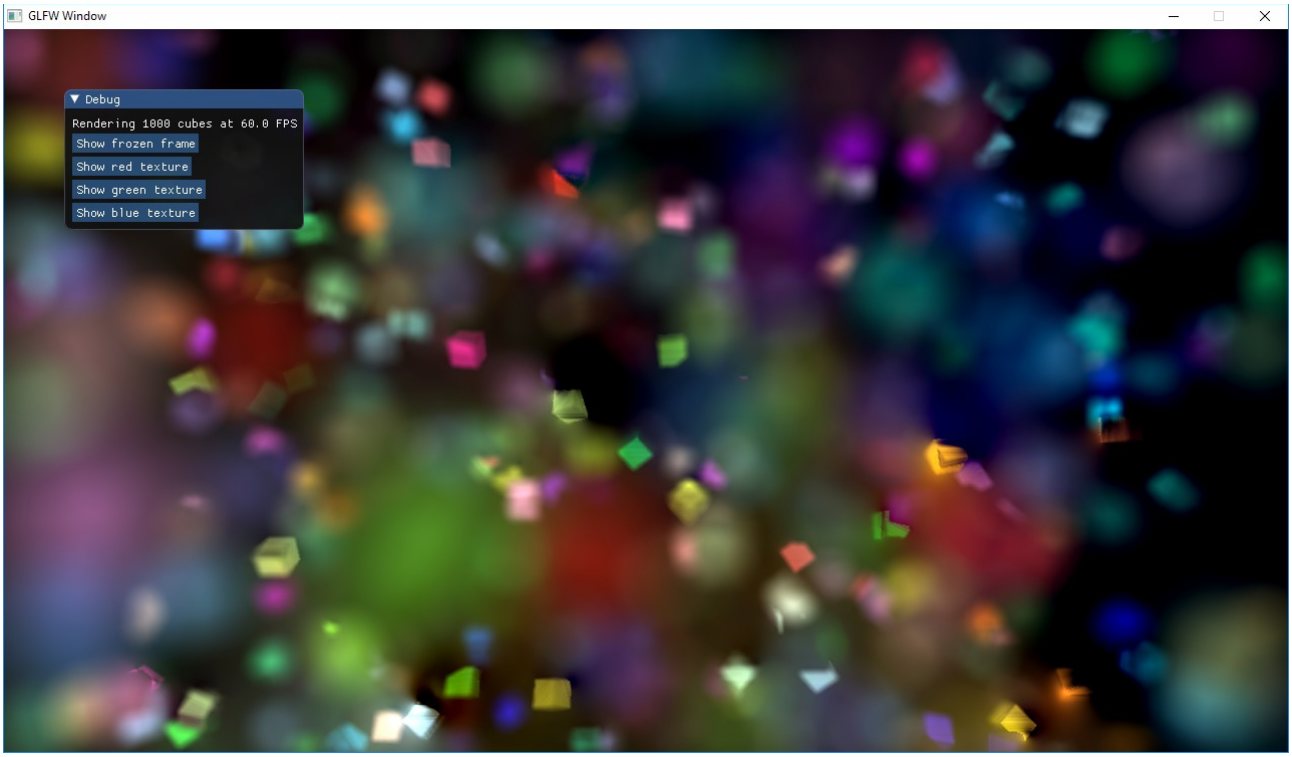
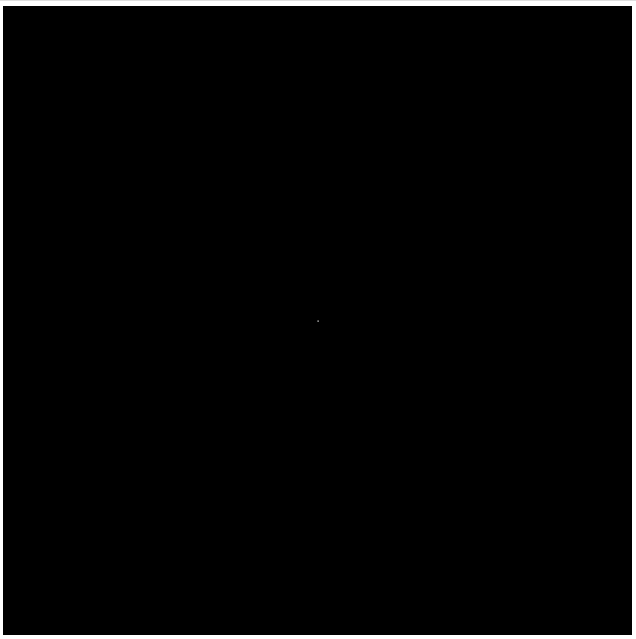


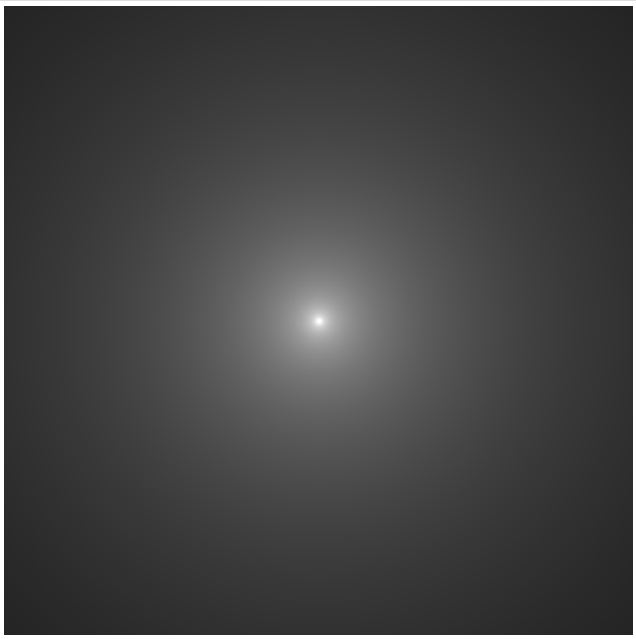
Illustration 3: Sparsification du laplacien sans recuit simulé (67% de compression)

II Vérification du pré-calcul (dossier *test*)

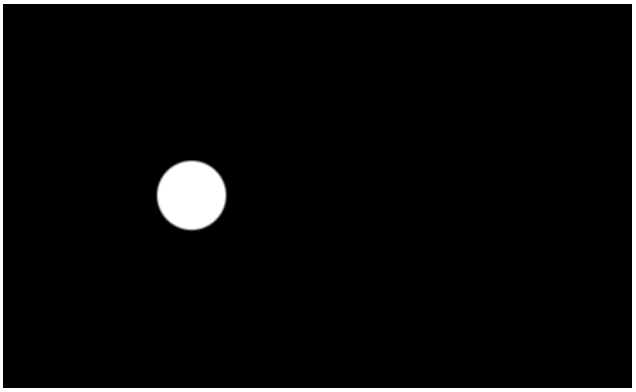
Pour lancer l'application : ./conv_test <prefix> . Pour les fichiers de spreadlets inclus test_data.bin et test_sizes.bin, l'appel correspondant est './conv_test test'. L'application produit une image 'integrate.png' contenant le résultat. L'originale correspondant à l'exemple inclu est kernels_depth_only_png/100.raw.png .

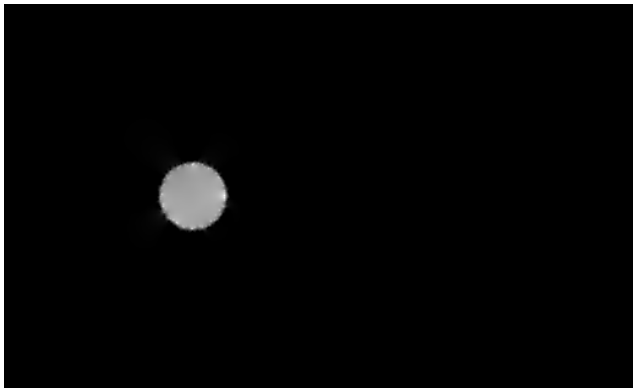
Afin de vérifier l'exactitude des fonctions écrites, et plutôt que de comparer avec la référence – ce qui m'aurait difficilement aidé à différencier d'éventuels bugs de simples imprécisions numériques, j'ai écrit un programme de test qui prend en argument un *spreadlet*, l'intègre, puis le compare avec l'image de la PSF originelle. Cela sert à la fois de tester la compression en laplacien éparse du dossier *precompute*, mais aussi le code d'intégration dudit laplacien, venant de l'article *Convolution Pyramids* par Farbs et. al, comme évoqué par l'article – n'ayant pas trouvé de code C++ en ligne, j'ai en effet décidé de l'implanter moi-même à l'aide d'Eigen pour bénéficier de la vectorisation d'instructions.





Test des pyramides convolutionnelles sur un dirac (gauche). On reconnaît la forme logarithmique de la fonction de Green 2D en espace libre sur l'image intégrée (droite - l'échelle a été manipulée volontairement).





Test du processus de sparsification. À gauche, l'image originelle et à droite l'image reconstruite (l'échelle n'est pas respectée). L'image de droite est obtenue par intégration d'un spreadlet de 194 points, par opposition aux 2800 pixels non nuls de l'image originelle. Elle est également plus sombre que l'original à cause d'une mise à l'échelle faisant en sorte que le pixel le plus clair ait une valeur de 255 – ceci n'impacte pas le résultat final.

III Application des *spreadlets* (dossier *compute*)

Pour lancer l'application : make run [IN=prefix]. IN a une valeur par défaut de 'spreadlets', de sorte à ce que 'make run' fonctionne si le dossier res contient les fichiers spreadlets_data.bin et spreadlets_sizes.bin produit par 'precompute'.

Une fois les *spreadlets* calculés et vérifiés, ils forment deux fichiers *spreadlets_data.bin* et *spreadlets_sizes.bin*, qui sont respectivement les données *x y valeur* de chaque *spreadlet* à la suite (*float* 4 octets par valeur) et le vecteur des tailles

cumulées (*int 4 octets par valeur).* Ce format permet l'interchangeabilité de plusieurs *datasets* de *spreadlets* afin de tester différentes valeurs sur différentes scènes.

Là où l'article mentionne l'usage de CUDA, j'ai opté pour les *compute shaders* OpenGL pour une meilleure portabilité du code. Ceci n'apporte aucune différence au résultat. L'algorithme est le même, où pour chaque fragment, le *spreadlet* correspondant est calculé via la topologie de grille présentée dans l'article, puis multiplié par la couleur du pixel et réécrit dans 3 nouvelles textures (un pour chaque canal).

Ma scène de test consiste en un nuage de cubes, chacun avec une position et une rotation (axe et vitesse) aléatoires. Il a été mentionné par l'un des auteurs lors d'une correspondance par mail qu'une telle scène possède un fort degré de complexité en profondeur, et qu'il serait ainsi difficile d'obtenir des résultats visuellement probants. J'ai malgré tout décidé de garder cette scène afin de voir les limites de cette approche.

Les résultats actuels sont ceux représentés dans l'Illustration 3. Cette image est générée en un peu moins d'une seconde (0.908s en moyenne), en comptant tous les tranferts CPU-GPU, les écritures disques (facteur omis par les *timings* de l'article des pyramides convolutionnelles) et les trois intégrations nécessaires pour les trois canaux de couleur.

IV Conclusion

Avec une compression de 67 %, le résultat est visuellement proche mais bien distinct. Je suppute que la clé pour améliorer ce résultat réside dans le recuit simulé que je n'ai pas réussi à faire fonctionner. Selon l'article, cette étape apporte une très faible amélioration en termes de compression, mais une nette amélioration en terme de visuels.

Je n'ai également pas eu le temps de pleinement explorer la topologie de grille évoquée dans l'article, mes efforts s'étant concentrés sur l'utilisation de *kernels* « *depth-only* », bien que j'aie à ma disposition des *kernels depth and speed*. Le *dataset* fourni par les auteurs étant déjà trié selon cette grille, l'étape de *precompute* ne change en rien. La différence réside dans le *compute shader* de l'étape *compute*, qui devrait appliquer le *backwards mapping* donné dans l'article pour trouver le *spreadlet* correspondant (ligne 39 de *res/splattingCompute.glsl*).