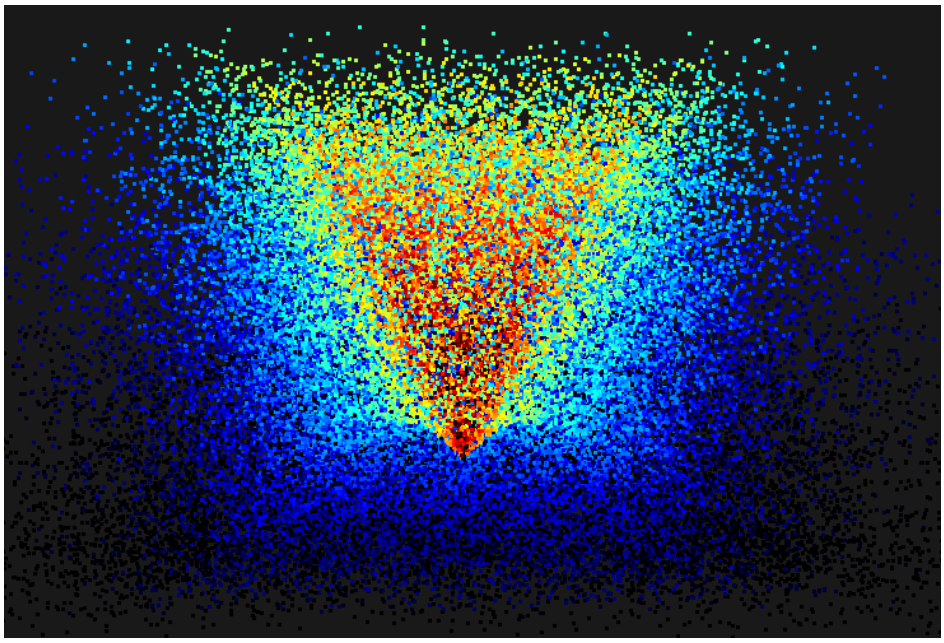
Rapport du TP4 du module GPU

Transform Feedback et Compute Shader pour le déplacement de particules

Antoine BRALET et Guillaume DURET



06 Novembre 2020

1 Introduction

Ce rapport vise à décrire, expliquer et relater les notions apprises et les résultats obtenus lors du TP n°4 du module GPU. Ce dernier visait à implémenter une méthode permettant de simuler le déplacement de particules indépendantes dans un espace à trois dimensions à l'aide du GPU de notre machine. La raison pour laquelle l'on cherche à implémenter cette simulation sur le GPU est simple : la grande rapidité d'exécution de ce dernier, en particulier pour un nombre si important de particules indépendantes. Pour ce faire deux méthodes ont été explorées : les *Transform Feedback* ainsi que les *Compute Shader*. Ces deux dernières sont décrites et comparées dans la première section de ce rapport. La seconde section permet de visualiser les résultats obtenus pour une simulation de 100.000 particules dont les contraintes sont décrites dans la section en question.

2 *Transform Feedback* vs *Compute Shader*

Cette section vise à décrire le fonctionnement et l'intérêt des deux méthodes : *Transform Feedback* ainsi que *Compute Shader*. Elles sont tout d'abord décrites séparément, avec leur fonctionnement et les subtilités d'implémentations. Le dernier paragraphe permet de comparer les méthodes en récapitulant leur fonctionnement.

L'utilisation de *Transform Feedback* fait partie intégrante du pipeline graphique. En d'autres termes, celui-ci ne peut être implémenté que si au minimum un vertex shader est présent dans le pipeline graphique. Cette méthode consiste à récupérer les données du buffer de sortie du vertex shader (ou au plus loin du geometry shader) et de les stocker dans un buffer différent. Les shaders utilisés ici visent uniquement à stocker ou calculer des données sur le GPU mais n'ont aucun intérêt graphique, en effet rien ne sera affiché à l'écran. À la fin de cette opération, seul un (ou plusieurs) buffer sera renvoyé sur le CPU. Ainsi il est possible d'utiliser les données récupérées pour modifier les buffers qui seront utilisés dans un deuxième temps depuis le CPU pour le pipeline graphique "classique". Cette fois ci l'ensemble des shaders de ce nouveau programme permettent non plus de modifier le comportement des données mais bien de les faire apparaître sur l'écran. Du point de vue implémentation la plus grosse différence réside dans la nécessité d'utiliser des buffers supplémentaires pour pouvoir stocker le résultat obtenu. De plus il est nécessaire dans l'initialisation de vertex shader utilisé dans le *Transform Feedback* de spécifier quelles données de sortie on veut stocker. Il faut enfin ajouter le fait qu'il faille bien actualiser les données présentes dans le VAO (en l'occurrence celles relatives aux buffers modifiés avant d'appeler le pipeline graphique).

Les *Compute Shader* fonctionnent d'une manière différente mais permettent de récupérer le même résultat. Ceux-ci sont complètement indépendants du pipeline graphique et peuvent être utilisés avant, pendant ou après celui-ci, sans que ça n'ai aucune incidence sur son fonctionnement. Ce type de shader vise à modifier directement les données présentes dans le buffer d'entrée et dans la mémoire du GPU. Ainsi il suffit alors à partir du CPU de récupérer les données présentes en mémoire du GPU pour obtenir dans le même buffer que précédemment les données modifiées par le GPU. Enfin il ne reste plus qu'à appeler le pipeline graphique classique permettant d'afficher les particules à l'écran. Du point de vue implémentation, il n'y a pas de grosses différences exceptés l'adaptation du nom du shader utilisé (GL_COMPUTE_SHADER) et le fait qu'il faille bien actualiser les données présentes dans le VAO (en l'occurrence celles relatives aux buffers modifiés par le *Compute Shader*) avant d'appeler le pipeline graphique. La plus grosse différence réside dans l'écriture du *Compute Shader* car la syntaxe diffère légèrement des précédents shaders (vertex, geometry et fragment) en ce sens que nous travaillons directement avec les buffers et non pas avec un sommet, primitive ou fragment en particulier. Mais une fois cet obstacle surmonté, l'implémentation reste classique.

En conclusion, les deux méthodes précédemment décrites permettent bien de modifier les données directement sur le GPU et de traiter les particules indépendamment les unes des autres en fonction de leurs caractéristiques propres. Néanmoins elles diffèrent selon leur fonctionnement. La première doit être directement intégrée au pipeline graphique et nécessite la création de deux buffers distincts sur le GPU : un pour les données d'entrée et un pour les données de sortie, et il est nécessaire de bien actualiser les données du buffer d'entrée à partir de celles présentes dans le buffer de sortie à chaque boucle d'affichage. Cette

duplication de buffer peut être particulièrement intéressante quand il s'agit de comparer un état passé des données à un état présent. Cependant dans un problème de modification de données comme c'est le cas ici, cette méthode en plus d'utiliser plus de mémoire nécessite un travail du CPU afin d'échanger les données des buffers. L'utilisation des *Transform Feedback* serait à première vue mieux adapter à la vérification du bon fonctionnement du pipeline graphique que pour modifier les buffers. La seconde, les *Compute Shaders*, au contraire modifie directement les données présentes dans le buffer d'entrée dans la mémoire GPU et est complètement indépendante du pipeline graphique. Il faut néanmoins veiller à actualiser les données du buffer d'entrée dans le VAO à chaque affichage afin de ne pas écraser les données en mémoire GPU à partir d'anciennes données en mémoire CPU. Cette méthode est d'autant plus intéressante qu'elle économise de l'espace en mémoire et est donc particulièrement utile lorsque l'on a un très grand nombre de données en entrée.

3 Résultats expérimentaux : Chute libre et visualisation

Les résultats présentés dans cette section sont obtenus en compilant puis exécutant les codes présents dans les dossiers "/tp_gpu_tf" et "tp_gpu_cs". Sachant que le résultat visuel est identique d'une méthode à l'autre, nous ne présenterons ici qu'une seule figure, le but étant non pas de comparer les méthodes mais de voir leur efficacité sur la simulation en question.

Cette simulation vise donc à modifier le comportement de 100.000 particules dont nous fixons la masse à 0.02g dans l'espace 3D et se divise en quatre grandes parties :

- Le déplacement des particules en fonction de leur vitesse (fixe)
- Appliquer l'effet de la gravité sur les particules et adapter la vitesse en fonction du temps
- Limiter le déplacement à une zone de l'espace puis revenir à une position par défaut et une vitesse aléatoire
- Colorer les particules en fonction de leur vitesse

Chacun de ces points est décrit de façon séparée dans ce rapport, néanmoins, pour les observer à partir du code fourni, il sera nécessaire de commenter/décommenter certaines lignes de codes, en particulier dans les shaders. En exécutant le code en tant que tel, le résultat obtenu est celui tenant compte de l'ensemble des points précédemment évoqués.

3.1 Déplacement des particules à partir des vitesses

Les modifications implémentées ici visaient particulièrement à mettre en place d'un côté le *Transform Feedback* et de l'autre le *Compute Shader*. Ceci passait dans les deux cas à la création d'un nouveau programme contenant un nouveau shader (resp. tf.vs et cs.vs) dans lequel les positions seraient modifiées en fonction de la vitesse propre d'une particule. De plus toutes les particularités mentionnées dans la section 2 ont dû être implémentées. Ceci permet alors d'obtenir des particules qui se déplacent en formant un parallélépipède comme visible sur la Figure 1. On peut bien y observer que ce parallélépipède s'élargit mais aussi se déplace légèrement vers le haut et vers la caméra.

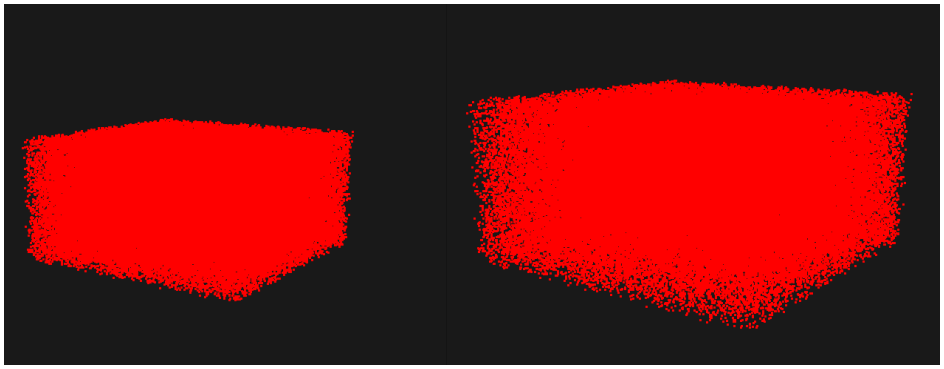


Figure 1: Déplacement des particules formant un parallélépipède, la figure de gauche est ici antérieure à la figure de droite

3.2 Effet de la gravité et évolution temporelle

De nouveau, très peu de modifications ont eu lieu afin de pouvoir mettre en œuvre l'effet de la gravité sur les particules et la dépendance temporelle du déplacement des particules. La majeure différence résidait dans le fait que désormais, le buffer de positions des particules n'était plus le seul à être modifié puisque le buffer de vitesse l'était également suivant la composante "y" puisque la gravité s'y appliquait, ce qui nous a permis d'expérimenter le bon fonctionnement relatif à la modification de plusieurs buffers à l'aide du *Transform Feedback* et du *Compute Shader*. Concernant l'évolution temporelle des particules, il nous suffisait de récupérer le temps écoulé entre deux rafraichissements d'écran et passer ce temps en variable *uniform* aux shader pour qu'il soit pris en compte. Le résultat de la Figure 2 permet de visualiser le bon fonctionnement de l'algorithme car on y voit les particules tomber tout en continuant à s'éloigner les unes des autres (dû aux vitesses suivant les autres composantes).

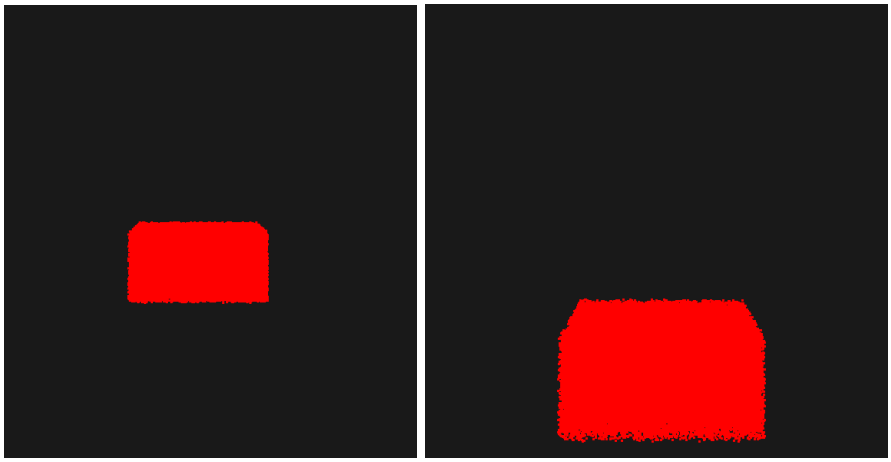


Figure 2: Effet de la gravité sur les particules, de nouveau l'image de gauche est antérieure à l'image de droite, on peut toujours y voir le parallélépipède s'agrandir à cause des vitesses suivant les autres composantes "x" et "z"

3.3 Limitation de l'espace

Cette limitation de l'espace est également accompagnée d'une mise à jour des positions et vitesses qui sont remises à des valeurs par défaut, soit le centre de l'image pour les positions (de coordonnées $(0, 0, 0)$) et

des vitesses aléatoires. La limitation de l'espace se fait simplement à l'aide d'un test sur les positions de la particule dans l'espace, il est important de noter que seule la limitation basse suivant la composante "y" est réellement utile ici car la gravité est la force la plus importante et attire les particules vers le bas. Enfin concernant les valeurs aléatoires par défaut attribuées dans le shader de la méthode utilisée, nous avons pris exemple sur le code implémenté dans http://www.science-and-fiction.org/rendering/noise.html qui se base sur les valeurs d'un sinus avec une fréquence très élevée au cours du temps et que nous avons passé en une dimension pour avoir une vitesse différente suivant chacune des composantes. La Figure 3 permet de visualiser le comportement des particules arrivant à la limite puis dont les valeurs sont mises à jour avec de nouvelles vitesses. De plus on peut remarquer que ce changement permet de stabiliser peu à peu le comportement des particules après le temps que l'on appellera *transitoire* de l'image de gauche.

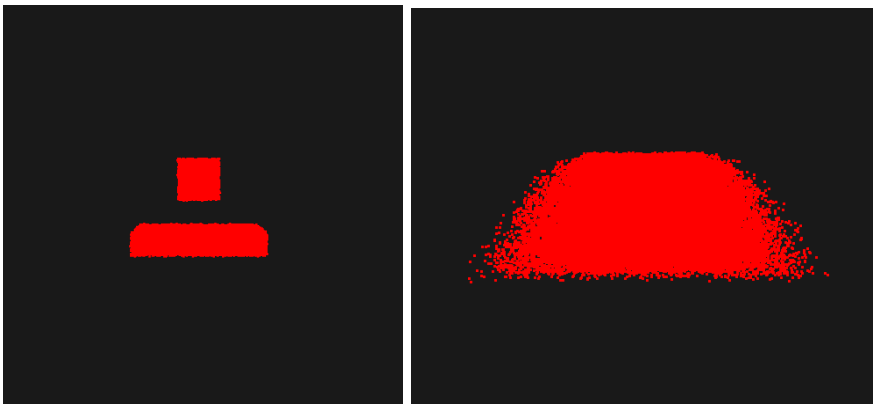


Figure 3: L'image de gauche, de nouveau antérieure à l'image de droite, illustre la mise à jour de la position et de la vitesse des particules lorsque celles-ci atteignent une limite basse fixée suivant la composante "y". Notons que les coordonnées $(0, 0, 0)$ correspondent au centre de l'image. L'image de droite correspond au comportement que l'on considérera comme stable des particules. En effet, une fois le comportement transitoire terminé (image de gauche) les particules "s'ordonnent" de cette façon malgré des vitesses toujours aléatoires lors de la mise à jour.

3.4 Coloration des particules

Enfin afin de visualiser plus aisément le comportement des particules, il est possible de colorer celles-ci. Nous avons délibérément choisi de colorer les particules uniquement en fonction de leur vitesses suivant la composante "y" en ce sens que les autres composantes restent constantes au cours du temps. Ainsi étudier leur vitesse permettrait uniquement de voir la bonne distribution aléatoire de vitesses. Dans notre cas, nous allons pouvoir constater l'action de la gravité et sa force au cours du temps. Concernant la coloration des particules, nous avons repris des *colormap* implémentées pour les shaders présents dans https://github.com/kbinani/colormap-shaders/. Ainsi il est possible de choisir plusieurs *colormap* en modifiant le nom de la fonction présente dans le fragment shader. Cinq d'entre elles sont implémentées dans le fragment shader : Summer, Spring, Jet, Hot et YlOrBr. L'image suivante a été obtenue à l'aide de la *colormap* Jet. On peut alors y constater que plus la couleur est rouge, plus elle est positive et forte, et plus elle est bleue (voire noire) plus elle est fortement négative. Comme attendu, la couleur est distribuée homogènement autour du centre : bleu lorsque les particules partent vers le bas, rouge quand elles partent vers le haut et vertes quand elles restent à hauteur. Puis petit à petit à cause de la gravité et de la masse de la particule, la vitesse décroit jusqu'à devenir négative et finalement retomber.

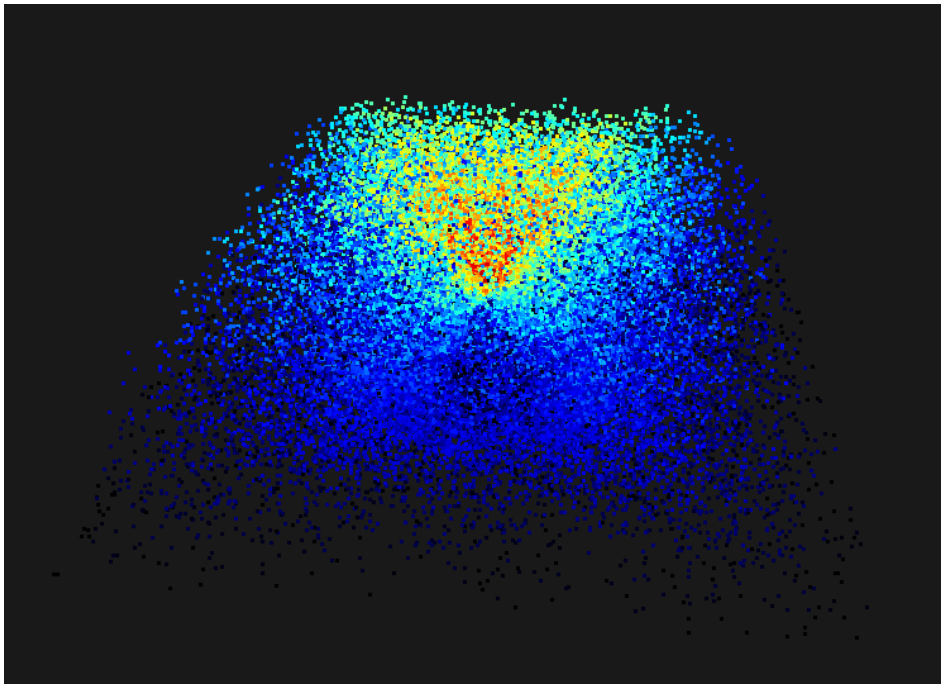


Figure 4: Effet de la gravité sur la vitesse des particules suivant la composante "y", plus la particule est rouge, plus celle-ci a une vitesse positive et grande et plus elle devient bleue plus la vitesse est fortement négative. On peut ici constater l'inversion de signe due à la gravité lorsque les particules deviennent vertes ainsi que la limite haute que les particules ne peuvent pas dépasser due à la limitation de leur vitesse initiale et à leur masse.

Il est bien sûr possible de modifier de nouveau la mise à jour des vitesses pour obtenir des effets différents, par exemple, l'image illustrant ce rapport sur la page de garde a été obtenue simplement en forçant la mise à jour des vitesses à être positive au départ lors de la mise à jour des valeurs. Ainsi ce type de simulation peut être particulièrement utile pour simuler des objets s'échappant d'une zone limitée de l'espace : un volcan, un canon à confettis, un feu d'artifice, une piñata relâchant des friandises... En adaptant le poids de la particule, la vitesse et l'angle de projection, l'ensemble de ces éléments peuvent être simulés.

4 Conclusion

Il a été vu durant ce TP l'utilisation des *Transform Feedback* ainsi que les *Compute shader* afin de pouvoir réaliser du GPGPU et ainsi pouvoir animer des très nombreuses particules et pouvoir simuler physiquement celles-ci avec la gestion des vitesses et de la gravité. Il est tout de fois notable que le *compute shader* est plus adapté à la réalisation des calculs avec sa possibilité de paramétrer les processeurs de la carte graphique alors que le *transform feedback* serait plutôt adapté pour avoir un feedback et vérifier le bon fonctionnement du pipeline graphique. Les deux méthodes sont configurées et appelés sur le CPU mais les calculs réalisés pour appliquer l'effet des vitesses et de la gravité sont réalisés sur le GPU pour ainsi bénéficier de la puissance de calcul en parallèle de celui-ci. Cela est particulièrement adapté ici car les calculs des 100 000 particules sont parfaitement parallélisable, les simulations sont restées totalement fluides malgré un nombre de particules très élevé.