Rapport de GPGPU

L'objectif de ce projet était de simuler le comportement de fourmis, qui partent d'une fourmilière à la recherche de nourriture et d'eau. Sur leur chemin, elles peuvent rencontrer des cailloux qui les obligent à dévier de leur trajectoire. Elles déposent des phéromones sur la carte durant leurs déplacements afin de retrouver ensuite le chemin de la fourmilière.

Fonctionnement général de l'application

L'application créée s'exécute ainsi :

- Initialisation d'OpenGL
- Ouverture de la fenêtre avec OpenGL
- Paramétrage d'OpenGL
- 4 Surfaces sont créées (Carte, Ressources, Phéromones, Fourmis)
- Initialisation de la simulation
    - Création des structures de données correspondants aux objets de la simulation (Rock, Anthill…)
    - Copie des structures sur les surfaces appropriées
- Association à la fenêtre d'une fonction callback appelée lors d'un appui sur une touche
- Affichage des informations sur le GPU
- Chargement en mémoire des textures depuis les fichiers situés dans le répertoire data
- Boucle principale :
    - On affiche des statistiques de la simulation
    - On récupère un pointeur sur la surface qui s'affichera
    - On fait avancer la simulation de 1 frame (fonction GenereImage)
    - On vérifie quelle surface on doit afficher (Tous, Carte, Ressources, Phéromone, Fourmis)
    - On applique les textures si elles ont été chargées
- On libère la mémoire prise par la simulation
- On libère les 4 surfaces utilisées par la simulation
- On détruit la fenêtre
- On arrête OpenGL

Mise à jour de la simulation

La fonction GenereImage du fichier gpgpu.cu est le cœur de notre application. Elle est appelée à chaque mise à jour de la fenêtre (frame) et se charge de recalculer les paramètres de la simulation. Elle effectue les opérations suivantes :

- Fusion de la surface de la carte et la surface des ressources dans la surface qui s'affichera
- Mise à jour la surface des phéromones en les dispersant/diluant

- Vérifie et, au besoin, régénère les ressources de la surface des ressources
- Lance la simulation pour chaque fourmi et les affiche sur la surface de sortie
- Génère une chaine de caractères à partir de propriétés de la fourmilière, qui sera ensuite affichée en titre de la fenêtre

Parallélisation des traitements

L'objectif premier de ce TP était d'accélérer la simulation en déportant les calculs depuis le CPU vers les 1024 threads disponibles sur le GPU. Pour cela, nous avons donc écrit des kernels pour chaque traitement parallélisable, c'est-à-dire les traitements qui nécessitent de parcourir soit une surface, soit un buffer (et pour lesquels chaque traitement sur un élément peut se faire de manière indépendante des autre éléments).

Nous avons utilisé des surfaces pour représenter en mémoire les informations qui sont destinées à être affichées à l'écran, à savoir :

- Le fond de carte (cailloux, bâtons, herbe)
- Les ressources (eau, nourriture)
- Les fourmis
- Les phéromones

Toutes ces surfaces sont ensuite superposées dans une surface de sortie qui sera ensuite affichée à l'aide d'OpenGL.

Les surfaces sont bien adaptées pour représenter graphiquement les éléments de la simulation, mais ne sont pas appropriées pour stocker les informations qui y sont liées car la création d'une surface engendre l'allocation d'un espace mémoire de 1024*1024 éléments de cette surface. Cela aurait été du gaspillage en comparaison du nombre d'éléments sur la carte (par exemple, nous avons 4*1024 fourmis générées). Nous avons donc également eu besoin de créer des buffers pour stocker les informations relatives aux éléments suivants :

- Les cailloux
- Les bâtons
- La nourriture
- L'eau
- Les fourmis
- La fourmilière

Le kernel le plus conséquent de notre application est celui responsable du déplacement des fourmis, nommé kernel_draw_ant. Il applique le traitement suivant en parallèle sur chaque fourmi :

- Calcul de la direction à prendre en fonction des pixels placés devant la fourmi (phéromones, obstacles et ressources)
- Déplacement de la fourmi (en avant ou demi-tour) : changement de ses coordonnées dans le buffer
- Vérification que les nouvelles coordonnées ne correspondent pas à l'objectif recherché (fourmilière ou ressources)

    - Si on a atteint la fourmilière, on vide la fourmi et on incrémente, dans la fourmilière, le compteur associé à la ressource trouvée. La fourmi fait demi-tour.
    - Si on a atteint une ressource, on modifie les informations de la fourmi dans le buffer, puis elle fait demi-tour pour chercher la fourmilière
- Efface le pixel de la fourmi dans la surface
- Synchronisation des threads (on attend que tous les threads aient terminé leur traitement)
- Coloration du nouveau pixel de la fourmi

Gestion des phéromones

A chaque appel de kernel_draw_ant, les fourmis déposent des phéromones. Cela correspond à des pixels sur la surface des phéromones, dont la couleur dépend du type de fourmi, et dont l'intensité dépend de la distance parcourue depuis la fourmilière.

Les différents types de fourmis sont :

- En recherche de ressources
- En recherche de fourmilière (avec de l'eau)
- En recherche de fourmilière (avec de la nourriture)

Lors de l'étape du calcul de direction, on prend en compte le type de fourmi et les phéromones devant elle :

- Lorsqu'une fourmi part de la fourmilière, elle se déplace de façon aléatoire, et, si elle trouve des phéromones correspondant à une fourmi portant une ressource, elle va suivre ces phéromones.
- A l'inverse, les fourmis portant des ressources cherchent à suivre les phéromones des fourmis en recherche de ressources afin de regagner la fourmilière.

Afin d'éviter que, lorsqu'une fourmi se perd, les autres fourmis la suivent, les phéromones sont mises à jour à chaque frame par un kernel chargé de leur dispersion, qui diminue l'intensité de toutes les phéromones de la surface qui leur est associée.

Pour permettre aux fourmis en recherche de trouver plus facilement leurs congénères, nous avons également ajouté un phénomène de dilution des phéromones. Un kernel est appliqué à la surface des phéromones, et, pour chaque pixel de cette surface, lui affecte une nouvelle valeur, qui est la moyenne des pixels voisins. Chacun des trois types de phéromones est représenté par une des trois composantes de couleur rouge (nourriture), vert (recherche fourmilière) ou bleu (eau). La figure 1 représente uniquement la surface des phéromones, et permet bien de voir le phénomène de dilution (élargissement des « chemins ») et de dispersion (perte d'intensité).

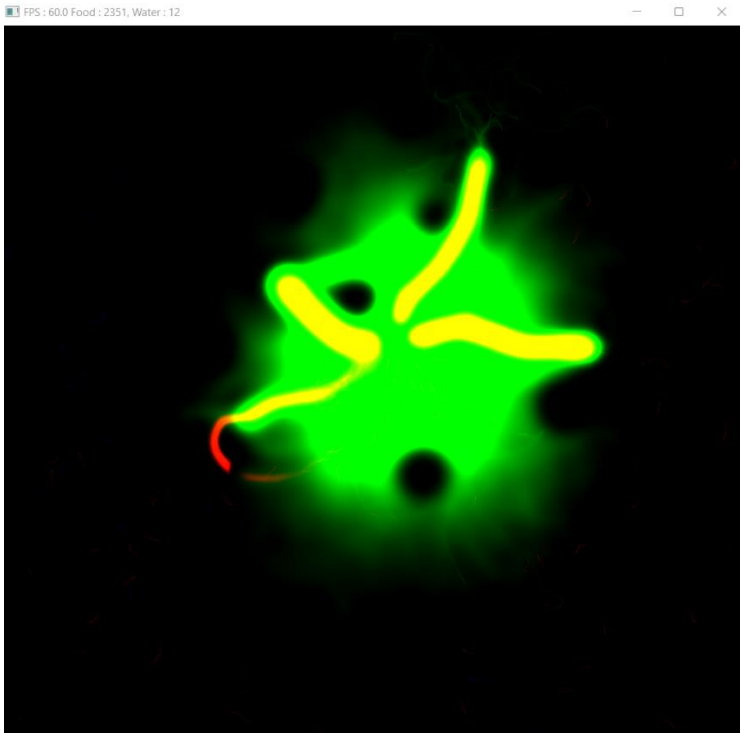

Figure 1 : Surface des phéromones

Détection des appuis clavier

La fonction glfwSetKeyCallback fournie par OpenGL permet de définir une fonction callback qui sera appelée lors de la détection d'un frappe clavier sur une fenêtre donnée.

Cela nous a permis de définir un comportement particulier pour certaines touches :

- ESCAPE : entraine la fin du programme
- 'M' : affiche seulement la carte
- 'R' : affiche seulement les ressources
- 'P' : affiche seulement les phéromones
- 'A' : affiche seulement les fourmis
- 'U' : enlève/remet la limite de 60 FPS
- 'T' : enlève/affiche les textures
- 'L' : active/désactive l'affichage des textures en mode linéaire
- '+' : change d'écran dans le sens Toutes les surfaces -> Carte -> Ressources -> Phéromones -> Fourmis -> Toutes les surfaces -> etc…
- '-' : change d'écran dans le sens opposé a +
- '0' : réinitialise la simulation, génération d'une nouvelle carte

On peut donc, à l'aide du clavier, obtenir l'affichage de chaque surface indépendamment. Par exemple, ci-dessous, l'image de gauche montre uniquement les fourmis, l'image du milieu uniquement les ressources, et l'image de droite, le fond de carte statique. La figure 1, présentée plus haut, affiche quant à elle seulement les phéromones.

Figure 2 : Différentes surfaces affichées séparément

Application de textures externes

Afin de rendre l'application plus visuelle, nous avons ajouté la possibilité d'importer des textures à partir d'images Bitmap (non compressées), stockées, dans notre cas, dans un dossier Data. Pour cela nous avons écrit un parseur de fichiers bitmap (voir fonction loadBMP). Cette fonction commence par lire les en-têtes du fichier, puis on lit les 768 octets (16*16*3 pour trois composantes dans un carré de 16 pixels de côté) représentant la texture. Cette texture est ensuite stockée dans un objet texture (fournit par CUDA) à l'aide de la fonction fromArrayToTex. Une fois toutes les textures chargées en mémoire (dans des variables globales), elles sont appliquées par la fonction applyTex2D (qui permet de choisir le mode d'affichage de la texture : avec ou sans interpolation linéaire des pixels), qui elle-même fait appel au kernel applyTexture afin de remplacer, dans la surface finale, les couleurs des différents éléments par la texture associée.

Les images ci-dessous permettent de voir, pour une même simulation, le résultat avec l'application de textures (à gauche) ou simplement avec des couleurs (à droite).

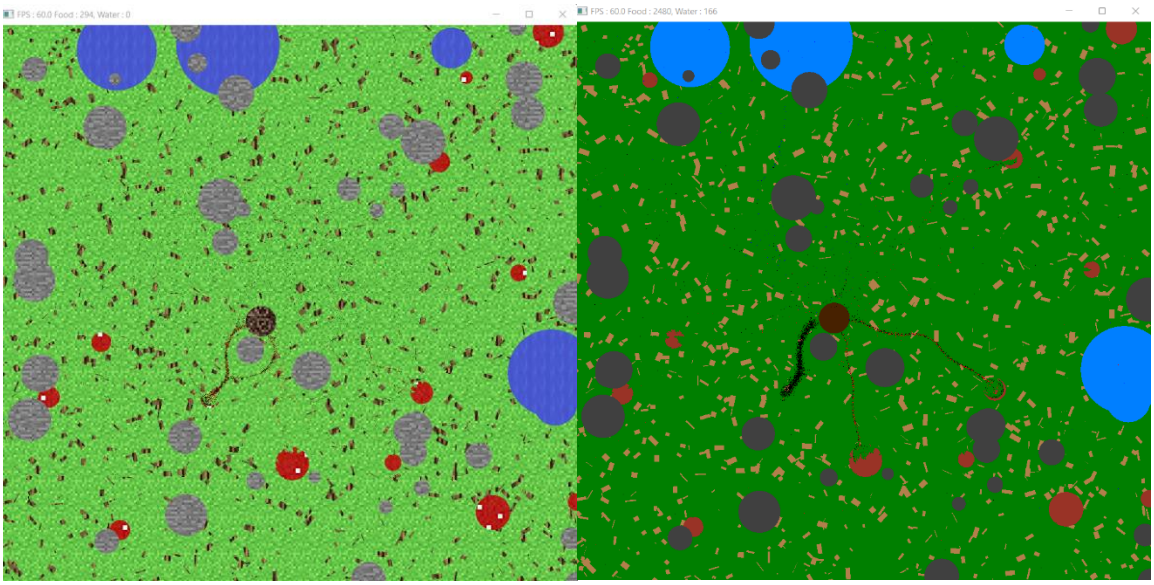

Figure 3 : Exemple de simulation avec et sans textures

Conclusion

L’objectif de ce TP, qui était de simuler le comportement de fourmis recherchant des ressources en exploitant les possibilités du GPU, est atteint. Nous disposons d’une simulation fonctionnelle, qui utilise différentes fonctionnalités mise à notre disposition par CUDA. Il resterait encore à régler quelques cas limites dans le fonctionnement de simulation introduits par la parallélisation. En effet, il faudrait sécuriser l’accès concurrent à certains espaces mémoire, notamment au compteur de ressources de la fourmilière ou aux pixels associés à ces ressources. Nous avons également pu exploiter quelques fonctionnalités supplémentaires d’OpenGL et de CUDA, afin de détecter les appuis clavier pour afficher différents écrans, ou bien d’importer des textures pour embellir l’application.