Systèmes d'Exploitation && Programmation Concurrente
Pipeline graphique

Travaux Pratiques Séance 13 — Jeudi 12 décembre 2013

1 Présentation

Les cartes graphiques modernes utilisées dans nos ordinateurs, tablettes ou smart-phones sont organisées en *pipeline* graphique.

Un pipeline est une suite d'étages permettant de réaliser une tâche complexe en la découpant en sous-tâches plus ou moins indépendantes.

Dans le cas d'un pipeline graphique, l'idée est de confier les différentes étapes de transformation des données graphiques à différentes unités de calcul.

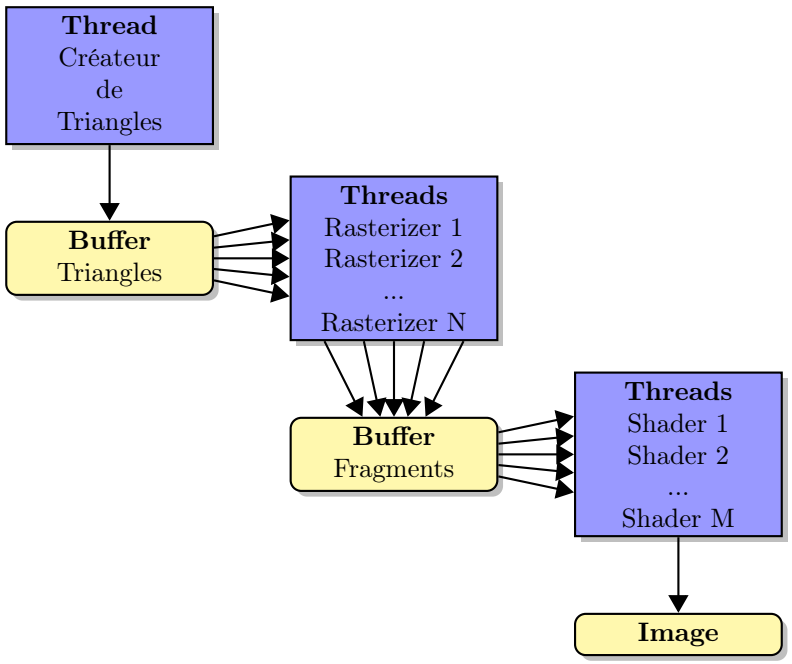

Figure 1 – Notre pipeline simplifié

Pour le TP d'aujourd'hui, nous allons simuler le pipeline simplifié présenté sur la figure 1.

Ce pipeline est composé de trois étages et de trois zones de mémoires. Chaque étage sera composé d'un ou plusieurs *threads*.

Le premier étage (`triangle_stage`), composé d'un seul thread, aura pour rôle de produire des triangles et de les stocker dans un buffer de `Triangles`. Ce thread unique se comportera donc comme un *producteur*. Les triangles seront récupérés un à un à l'aide de la fonction triangle_creator_get_next_triangle() présentée ci-après.

Le deuxième étage (`raster_stage`) sera composé de plusieurs threads qui viendront consommer des triangles depuis le buffer de triangles précédents. Chacun de ces triangles sera ensuite découpé en fragments d'images comportant une position dans l'image ainsi que d'autres informations. Ces fragments seront obtenus un à un à l'aide de la fonction raster_get_next_fragment() présentée ci-après. Les fragments devront être déposés dans une zone de mémoire partagée (un buffer tournant). Le deuxième étage est donc également producteur de fragments.

Le troisième étage (`shader_stage`) sera composé de plusieurs threads qui viendront consommer des fragments pour les stocker dans l'image finale.

L'image est créée et sauvegardée par le premier étage.

2 Les trois étages

Bien entendu, il ne vous est pas demandé de créer toutes les fonctions du pipeline, mais uniquement de les utiliser dans des threads qui devront implémenter des mécanismes de synchronisations via des mutex, des sémaphores, des conditions ou des barrières.

Les fonctions manipulant les données graphiques sont déclarées dans le fichier `gpu.h` et implémentées dans le fichier `gpu.c`. **Ne modifiez pas ces fichiers !** Cependant, vous devrez utiliser ces fonctions.

Le but du programme est de créer une image (`image.ppm`) à partir d'une scène 3D. Cette scène est partiellement stockée dans le fichier `elephant.obj`.

2.1 Le créateur de triangles

Le rôle du thread (il n'y en a qu'un dans notre modèle) « créateur de triangles » est de récupérer des triangles et de les produire dans le tableau `triangles`. Il est implémenté dans la fonction `triangle_stage` que vous devrez compléter.

La fonction suivante :

```
void *triangle_creator_init();
```

permet d'initialiser le créateur de triangles, de charger la scène et d'effacer l'image. La valeur renvoyée permet d'identifier le créateur et doit être passée comme premier paramètre aux deux fonctions suivantes.

Une fois l'initialisation faite, la fonction suivante permet de récupérer chacun des triangles qui composent la scène.

```
int triangle_creator_get_next_triangle(void *creator, struct Triangle *triangle);
```

Le premier paramètre doit être ce qui est renvoyé par `triangle_creator_init()`, le second contient l'adresse d'une variable de type `struct Triangle` qui sera remplie par la fonction. La valeur renvoyée par cette fonction est 1 si le triangle a été correctement créé et 0 sinon. La valeur zéro signifie donc que la scène est terminée et que le thread créateur de triangles peut se terminer.

Toutefois la fonction suivante doit être appelée pour libérer la mémoire et sauvegarder l'image sur le disque :

```
void triangle_creator_destroy(void *creator);
```

Enfin, la fonction suivante permet de savoir si un triangle est réellement un triangle ou une demande de synchronisation.

```
int triangle_is_sync(struct Triangle *triangle);
```

En effet, de temps en temps (lors d'un changement de couleur dans notre cas), il est nécessaire de synchroniser tous les étages, et donc de vider (consommer) les buffers. Cela permet d'éviter que les triangles et fragments qui restent lors d'un changement de couleurs ne soient dessinés avec la nouvelle couleur (voir figure 2).

2.2 Les rasteriseurs

L'étage des rasteriseurs a pour rôle de transformer les triangles (en les consommant) en fragments (en les produisant). Il est implémenté dans la fonction `raster_stage` que vous devrez compléter.

La fonction suivante permet d'initialiser la rasterisation d'un triangle (passé en paramètre) :

```
void *raster_init(const struct Triangle *tri);
```

La valeur renvoyée doit être utilisée comme premier paramètre de la fonction suivante (dont le fonctionnement rappelle celui de la fonction `triangle_creator_get_next_triangle()`

```
int   raster_get_next_fragment(void *raster_state, struct Fragment *fragment);
```

Une fois un triangle complétement traité, on doit libérer la mémoire qu'il a utilisé en appelant la fonction suivante :

```
void  raster_destroy(void *raster_state);
```

Lors de la synchronisation, nous allons devoir créer des fragments de synchronisation à l'aide de la fonction suivante :

```
struct Fragment fragment_sync(void);
```

Enfin, la fonction suivante permet de savoir si un fragment est un réellement un fragment ou une demande de synchronisation.

```
int fragment_is_sync(struct Fragment *fragment);
```

2.3 Les shaders

Le dernier étage de notre pipeline est implémenté par la fonction `shader_stage`. Son rôle est de consommer des fragments pour les mettre dans l'image.

Pour cela, on appelera la fonction suivante qui placera le fragment au bon endroit, calculera sa couleur exacte, etc. :

```
void gpu_shader(struct Fragment fragment);
```

3 Travail demandé

Vous ne devrez modifier que le fichier `pipeline-graphic.c`. Le travail demandé est en trois temps :
— comprendre le programme (il n'est pas nécessaire de comprendre ce qu'il y a dans `gpu.c`)
— implémenter les producteurs/consommateurs de triangles et fragments dans les différents étage
— implémenter les synchronisations à l'aide de barrières (facile) ou de conditions (complexe)

3.1 producteurs / consommateurs

Pour cette partie, il vous faudra compléter le code dans les trois fonctions des threads. Essayez de bien comprendre qui produit quoi et qui consomme quoi.

Une fois ceci fait, vous devez obtenir une image du genre de celle de la figure 2.

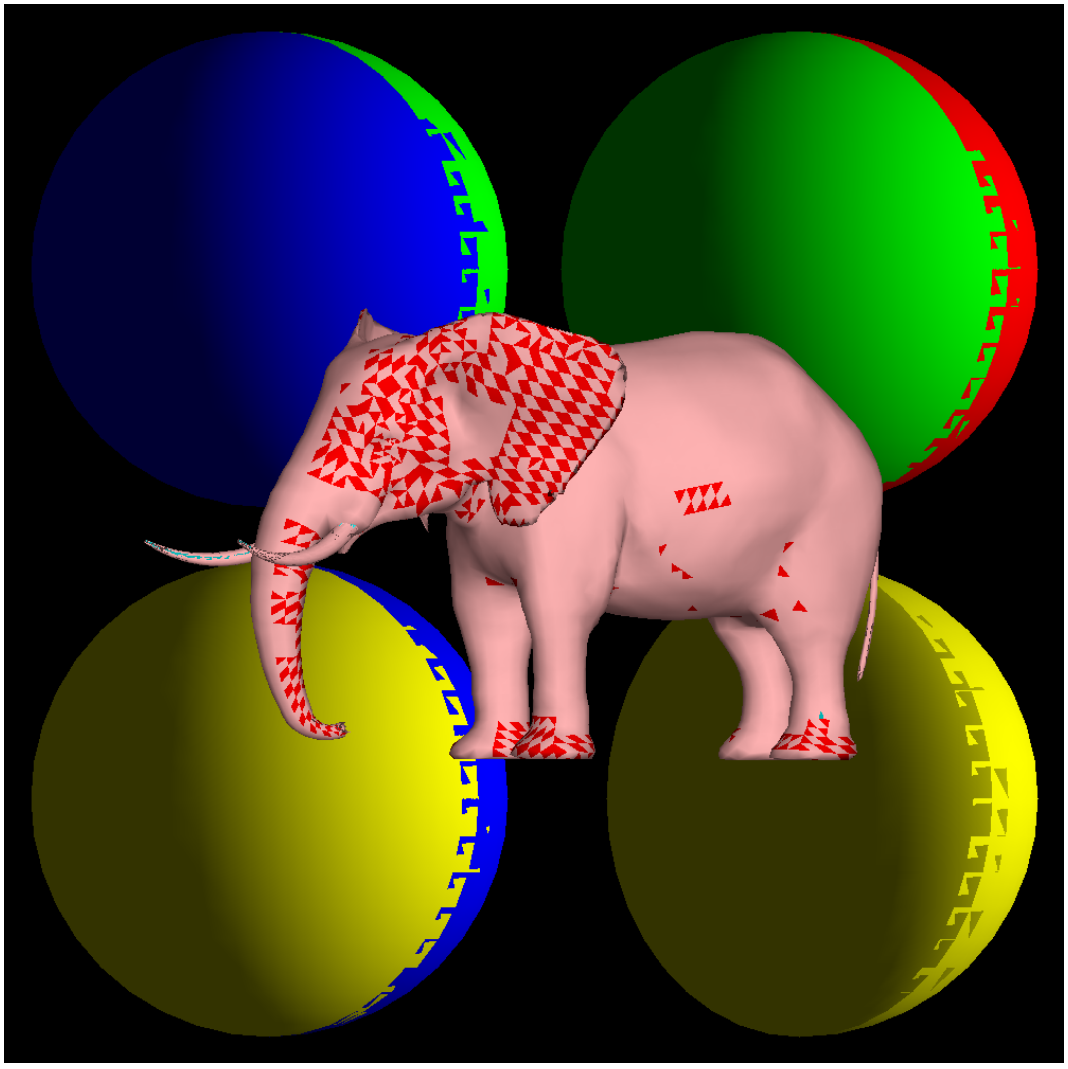

FIGURE 2 – Notre scène avec des problèmes de synchroniation des étages.

3.2 Barrières

Afin d'éviter les problèmes de la figure 2, il faut implémenter des synchronisations. En effet, lorsqu'un changement de couleur intervient, un triangle particulier, dit "de synchronisation" est créé. Cela permet au thread créateur de triangles de détecter qu'il doit attendre que les autres threads finissent leur travail.

Pour cela, je vous conseille fortement l'utilisation des fonctions suivantes :

— `pthread_barrier_init`
— `pthread_barrier_wait`
— `pthread_barrier_destroy`

Vous devriez alors obtenir une image identique à celle de la figure 3.

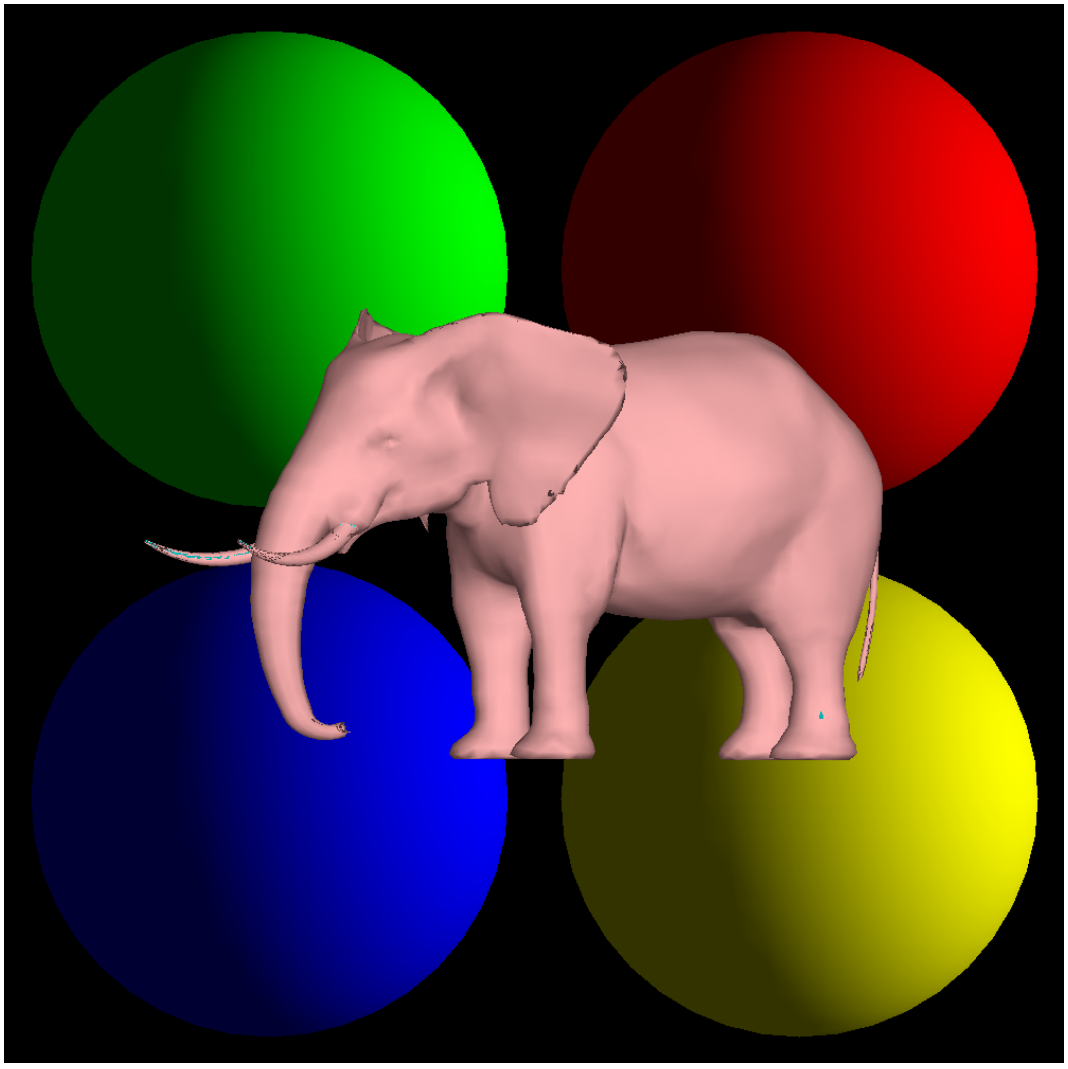

FIGURE 3 – Notre scène finale

4 Bonne fin d'année