Rapport GPGPU

L'objectif de ce TP était de créer une simulation de proies/prédateurs. Nous devions utiliser des lapins et des renards. Les lapins se reproduisent de temps en temps pour faire grandir le nombre de lapins. Les renards traquent et mangent les lapins, si un renard ne mange pas pendant un certain temps, il meurt.

La première étape fut de créer une structure d'animaux. Nous aurions pu créer une structure Renard et un autre Lapin. Cependant nous avons fait le choix de créer une structure commune.
On a choisi cette option pour permettre un passage plus simple en paramètre des fonctions communes aux deux espèces.

```
/**
    @struct Animaux
    @brief Contient des informations sur un animal dans une simulation.
    @param float u : X position of the animal.
    @param float v : Y position of the animal.
    @param float radius :  Size of the animal.
    @param float norm : Magnitude of the animal's velocity.
    @param float angle : Direction of the animal's movement.
    @param bool alive : Boolean value indicating whether the animal is alive or not.
    @param Espece type : Enumeration representing the species of the animal.
    @param float4 color :  Float4 value representing the color of the animal.
    @param int diet : time depuis dernier repas (pour fox)
*/
struct Animaux {
    float u;
    float v;
    float radius;
    float norm;
    float angle;
    bool alive;
    Espece type;
    float4 color;
    int diet;
};
```

Ensuite nous avons essayé d'afficher un lapin, représenté par un cercle dans notre window. Première solution, nous faisions un parcours de l'ensemble des pixels, si le pixel était compris dans le radius d'un lapin, on coloriait ce pixel. C'était loin d'être la meilleure façon, car cela nécessite de faire un parcours de notre tableau de lapin pour chaque pixel, on a donc amélioré cela.

```
/** @brief Dessine le cercle représentant les animaux
    @param cudaSurfaceObject_t surface : structure de l'image
    @param Animaux * animal : Tableau des animaux
    @return void*/
__global__ void Draw_Crcl(cudaSurfaceObject_t surface, Animaux * animal)
    float4 color;
    int ind =  threadIdx.x;
    if (animal[ind].alive) {
        float x = animal[ind].u - animal[ind].radius;
        float y = animal[ind].v - animal[ind].radius;
        float x2 = animal[ind].u + animal[ind].radius;
        float y2 = animal[ind].v + animal[ind].radius;

        float r2 = animal[ind].radius * animal[ind].radius;
        color = animal[ind].color;

        for (int i = max(0, (int)x); i <= min(kWidth-1, (int)x2); i++) {
            for (int j = max(0, (int)y); j <= min(kHeight-1, (int)y2); j++)
                float dx = animal[ind].u - (float)i;
                float dy = animal[ind].v - (float)j;
                if (dx*dx + dy*dy < r2) {
                    surf2Dwrite(color, surface, i * sizeof(float4), j);
                }
            }
        }
    }
}
```

La deuxième et dernière solution a été de parcourir un tableau de structure où tous les lapins étaient stockés. Lors du parcours, une fonction est appelée afin de colorier les pixels autour de la position du lapin.

La particularité de notre code est de créer un seul tableau avec au début les lapins, puis les renards à la suite. Cela permet d'avoir qu'un seul paramètre dans la fonction qui 'draw'. (Spoiler alert : on a fait deux tableaux différents par la suite)

Le tableau initial étant déclaré dans la mémoire cpu, il faut créer une copie sur la mémoire gpu, pour permettre d'y accéder.

```
// copie de donner CPU vers memoir GPU
Animaux* fox_d;
cudaMalloc(&fox_d, sizeof(Animaux) * NB_FOX);
cudaMemcpy(fox_d, v_renard, sizeof(Animaux) * NB_FOX, cudaMemcpyHostToDevice);

Animaux* rabbits_d;
cudaMalloc(&rabbits_d, sizeof(Animaux) * NB_LAPINS);
cudaMemcpy(rabbits_d, v_lapin, sizeof(Animaux) * NB_LAPINS, cudaMemcpyHostToDevice);
```

Par la suite, on a rempli le tableau de lapins avec l'initialisation de chaque animal. Les positions en x et y sont initialisées aléatoirement. Pour cela, on a créé un fonction qui renvoie une valeur aléatoire entre deux bornes.

```
/** @brief Permert de retourner un nombre aléatoire dans un intervalle
    @param int a : borne inférieure de l'intervalle
    @param int b : borne supérieure de l'intervalle
    @return float : nombre aléatoire dans cet intervalle */
float Rand(int a, int b)
{
    int nRand;
    nRand = a + (int)((float)rand() * (b - a + 1) / (RAND_MAX - 1));
    return (float)nRand;
}
```

La fonction pour le déplacement a été faite en générant un nombre pseudo aléatoire entre 0 et 1. Ce nombre est multiplié par une 'vitesse' afin d'avoir une norme plus ou moins grande. La direction de la trajectoire est l'addiction entre la direction à l'instant d'avant et un nombre compris entre $\frac{-\pi}{4}$ et $\frac{\pi}{4}$. Ce nombre est généré avec un nombre pseudo aléatoire auquel on applique une translation et une multiplication.

```
/** @brief Crée un trajectoire pseudo random
   @param Animaux* Animaux : tableau des Animaux en cours de traitement
   @param float time : time pour avoir des seeds random pour les déplacements
   @return void*/
__global__ void Deplacment(Animaux* Animaux, float time) {
    int ind = threadIdx.x;

    if (Animaux[ind].alive == true) {
        //lapin[ind].norm = ((random(lapin[ind].u * time, lapin[ind].v) - .5));
        if (Animaux[ind].type == LAPIN) {
            Animaux[ind].angle += (((random(Animaux[ind].v * time, Animaux[ind].u) - .5) * 2) * 3.14 / 5);
        }
        Animaux[ind].u += cos(Animaux[ind].angle) * (Animaux[ind].norm);
        (Animaux[ind].u > 1024) ? Animaux[ind].u = 0 : ((Animaux[ind].u < 0) ? Animaux[ind].u = 1024 : Animaux[ind].u);
        //lapin[ind].u = float(int(lapin[ind].u) % 1024) + fracf(lapin[ind].u);
        Animaux[ind].v += sin(Animaux[ind].angle) * (Animaux[ind].norm);
        (Animaux[ind].v > 1024) ? Animaux[ind].v = 0 : ((Animaux[ind].v < 0) ? Animaux[ind].v = 1024 : Animaux[ind].v);
        //lapin[ind].v = float(int(lapin[ind].v) % 1024) + fracf(lapin[ind].v);
    }
}
```

On voulait initialement affecter un déplacement totalement aléatoire. C'est-à-dire, une norme, une direction et un sens aléatoires. Cependant, les déplacements étaient pas assez naturels/fluides.

Un détail qui a son importance est la gestion des animaux qui s'approchent des bords de la window. Nous avions pensé à faire rebondir les animaux sur les bords. Nous avons pourtant choisi de laisser les animaux traverser les bords pour apparaître sur le bord opposé.

A ce moment-là, on a souhaité mettre en œuvre la reproduction des lapins. On a donc séparé les lapins et les renards en deux tableaux.

Pour la reproduction, on a donc besoin d'avoir un tableau dynamique. Pour plus de simplicité au début, on a fait le choix de créer un tableau avec N lapins mais juste un pourcentage d'entre eux seront vivants à l'initialisation. Pour les renards, ils seront tous vivants à l'initialisation du tableau mais pourront mourir.

```
// creation de la situation init
for (int i = 0; i < NB_LAPINS; i++) {
    v_lapin[i] = {Rand(0,1023), Rand(0,1023),8,3,0.0, (i <= NB_LAPINS / 5) , LAPIN, RABBIT_COLOR,0};
}

for (int i = 0 ; i < NB_FOX; i++) {
    v_renrad[i] = { Rand(0,1024), Rand(0,1024),10,2,0.0, (i <= NB_FOX), RENARD, FOX_COLOR,0};
}
```

Les naissances sont faites quand deux lapins sont proches. Pour chaque lapin, le gpu calcul si il y a un autre lapin à proximité. Pendant le premier parcours du tableau, on récupère l'indice du premier lapin mort, on se sert de cet emplacement pour faire naître le nouveau lapin. Il naitra à la position de ses parents.

On utilise le même procédé pour qu'un renard mange un lapin. Pour finir, si un renard voit un lapin à proximité, il le traque jusqu'à le manger. Si un nouveau lapin rentre dans sa ligne de mire, alors il traquera le plus proche des deux.