|  |
| --- |
| Université de Reims Champagne-Ardenne |
| Moteur de particules en CS4.0 |
| Projet de programmation GPU |

|  |
| --- |
| Foulon Matthieu  19/04/2010 |

Sommaire

[Introduction 4](#_Toc259627129)

[But du projet 4](#_Toc259627130)

[Difficulté 4](#_Toc259627131)

[Architecture 5](#_Toc259627132)

[Diagramme de classe de SPARK 5](#_Toc259627133)

[Diagramme de classe pour ce projet 6](#_Toc259627134)

[« Traduction » de la hiérarchie SPARK pour le projet 7](#_Toc259627135)

[Implémentation 8](#_Toc259627136)

[Fonctions implémentées 8](#_Toc259627137)

[Initialisation du moteur sur le CPU 9](#_Toc259627138)

[Génération de nombres aléatoires sur le GPU 11](#_Toc259627139)

[Gestion des collisions sur le GPU 11](#_Toc259627140)

[Conclusions 13](#_Toc259627141)

[Capture d’écran 13](#_Toc259627142)

# Introduction

## But du projet

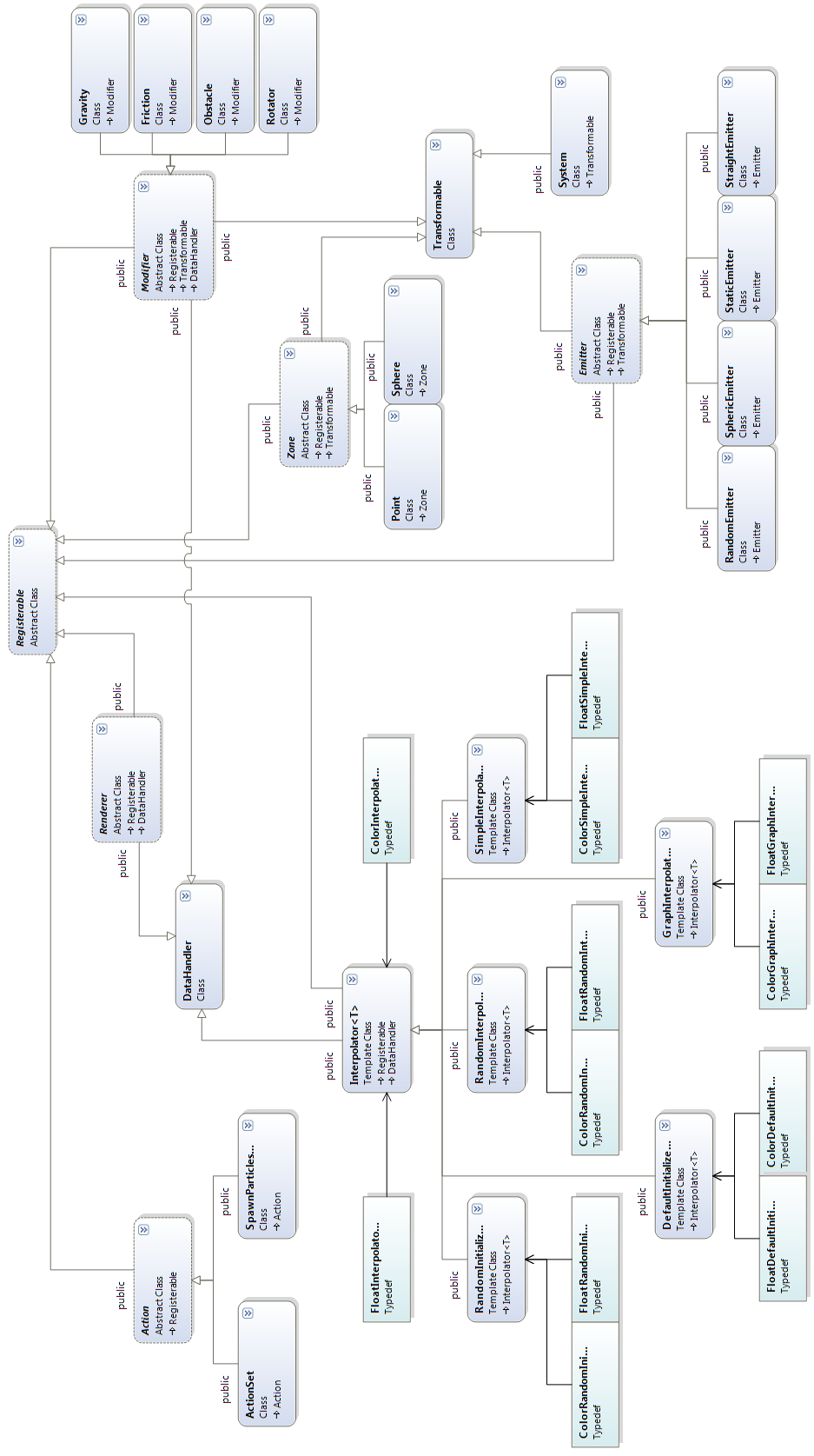
* Porter le moteur de particules SPARK (<http://spark.developpez.com/?page=home&lang=fr>) sur GPU
* Vérifier le gain de performance et la faisabilité de porter toutes les fonctionnalités

## Difficulté

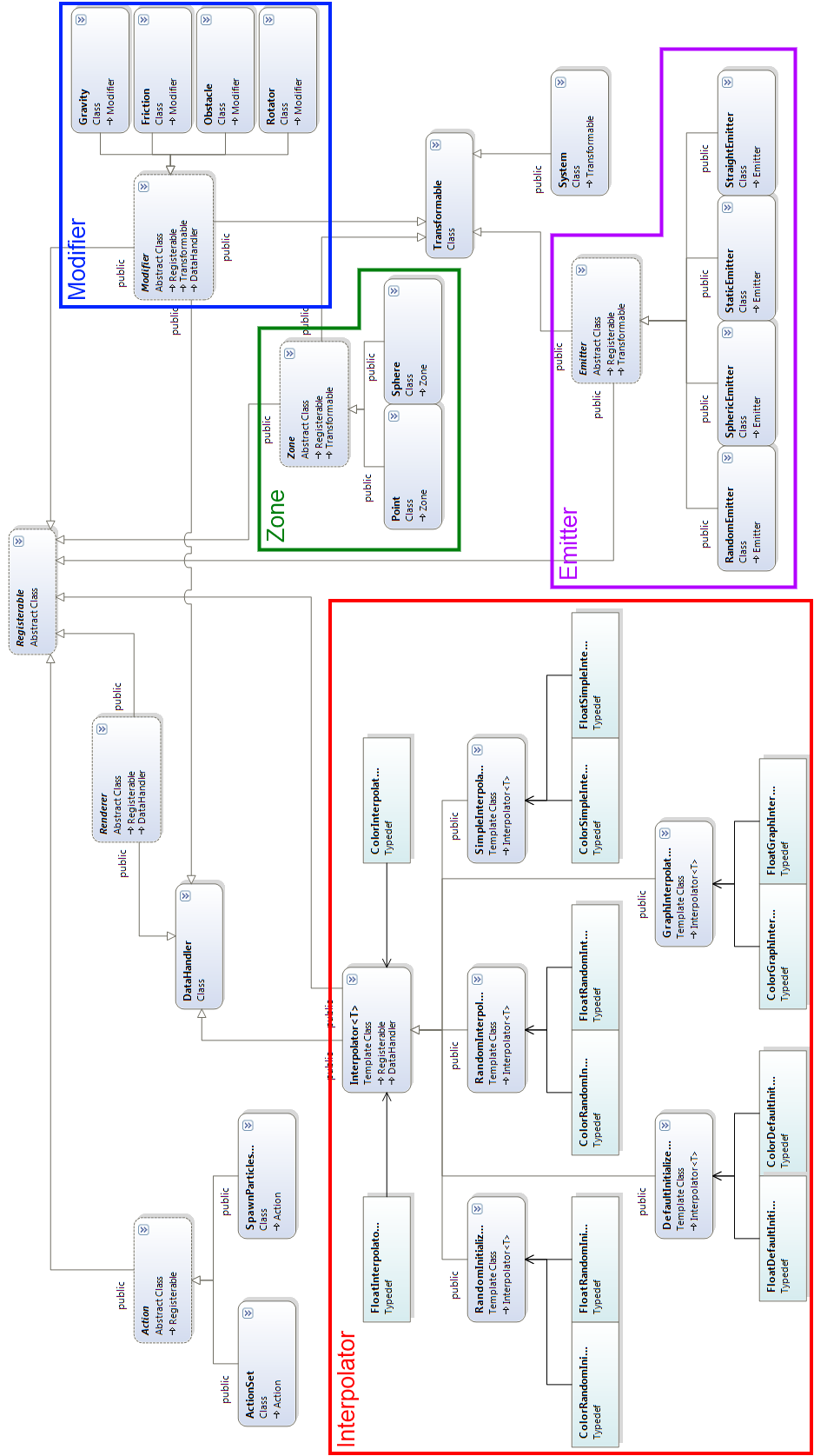
* Un seul buffer en écriture en cs4.0
* Pas de pointeur
  + Remplacement de toutes les références par un index dans une table
* Pas de classe (au sens c++) donc pas d’héritage
  + Obligation sur la taille des structures de la fixer au maximum de l’ensemble des classes que chacune des structures doit représenter
* Taille des tableaux fixes
* Pas de nombres aléatoires sur le GPU (en compute shader tout du moins)
* Pas de possibilité de débugger son code facilement

# Architecture

## Diagramme de classe de SPARK



## Diagramme de classe pour ce projet



## « Traduction » de la hiérarchie SPARK pour le projet

Les structures encadrées doivent être assez grandes pour contenir toutes les informations disponibles dans ces mêmes cadres :

|  |
| --- |
| struct X  {  float4 data1;  float4 data2;  …  }; |

Toutes les structures ont une taille multiple d’un float4, pour l’alignement.

Exemple avec la structure Zone :

|  |
| --- |
| struct Zone  {  D3DXVECTOR4 data1; // position, radius  D3DXVECTOR4 data2; // tPosition, type  D3DXVECTOR4 data3; // normal  D3DXVECTOR4 data4; // tNormal  D3DXVECTOR3& Position() {return \*((D3DXVECTOR3\*)&data1);}  float& Radius() {return data1.w;}  D3DXVECTOR3& TPosition() {return \*((D3DXVECTOR3\*)&data2);}  float& Type() {return data2.w;}  D3DXVECTOR3& Normal() {return \*((D3DXVECTOR3\*)&data3);}  D3DXVECTOR3& TNormal() {return \*((D3DXVECTOR3\*)&data4);}  void generatePosition(D3DXVECTOR3& v, bool full, float radius = 0.0f);  }; |

Une zone pouvant représenter un point, une sphère, un plan, etc, il est nécessaire d’avoir les informations suivantes :

* Position -> pour tous les types de zone
* Normale -> pour les plans
* Rayon -> pour les sphères
* etc

# Implémentation

## Fonctions implémentées

Zone, permet de représenter une partie de l’espace :

* Méthodes sur le CPU
  + createPlaneZone ; une position et une normale, créé un plan infini
  + createSphereZone ; une position et un rayon, créé une sphère
  + createPointZone ; une position, créé une zone ponctuelle
* Méthodes sur le GPU
  + generatePosition ; retourne un point appartenant à la zone
  + computeNormal ; retourne la normale à un point de la zone
  + intersects ; teste si le rayon donné en argument intersecte la zone

Interpolator, permet de faire des interpolations sur des données et de les initialiser :

* Méthodes sur le CPU
  + createDefaultInitializer ; donne une valeur de départ à une variable
  + createSimpleInterpolator ; permet de faire varier une variable au cours du temps
* Méthodes sur le GPU (X, 1..4, la taille du vecteur en entrée, 1 et 4 seulement de programmé)
  + interpolateParamX ; réalise une interpolation linéaire
  + interpolateX ; fait varier la variable au cours du temps si nécessaire
  + initX ; initialise la variable

Modifier, modélise les contraintes agissant sur les particules :

* Méthodes sur le CPU
  + createGravityModifier ; créé un effet de gravité
  + createObstacleModifier ; créé un obstacle
* Méthodes sur le GPU
  + modify ; applique le modificateur

Emitter, différentes formes d’émission des particules (pas intégré dans le code GPU malgré que l’implémentation ait été faite) :

* Méthodes sur le CPU
  + createRandomEmitter ; créé un émetteur qui envoie dans toutes les directions
* Méthodes sur le GPU
  + generateVelocity ; créé un nouveau vecteur de vitesse

## Initialisation du moteur sur le CPU

|  |
| --- |
| void InitEngine()  {  Group group = createGroup(MAX\_PARTICLES);  group.MinLifeTime() = 2.0f;  group.MaxLifeTime() = 5.0f;  // indice de l’interpolator dans g\_vecInterpolators  group.ColorInterpolator() = 1.0f;  // possibilité de stocker plusieurs modificateurs  // ici dans x et y, correspondant comme précédement à l’indice  // mais ici dans g\_vecModifiers  group.Modifiers().x = 0.0f;  group.Modifiers().y = 1.0f;  // indice dans g\_vecInterpolators  group.ParamAngleInterpolator() = 2.0f;  group.ParamSizeInterpolator() = 3.0f;  // la possibilité d’avoir plusieurs groupes n’est pas exploitée  // pour l’instant  g\_vecGroups.push\_back(group);  // ajoute la gravité selon l’axe y  g\_vecModifiers.push\_back(  createGravityModifier(D3DXVECTOR3(0.0f, -10.f, 0.0f)));  // ajoute un obstacle à la scène, il utilisera la zone d’indice 2  g\_vecModifiers.push\_back(createObstacleModifier(2, 1.0f, 1.0f));  g\_vecZones.push\_back(createPointZone(D3DXVECTOR3(0.0f, 0.0f, 0.0f)));  g\_vecZones.push\_back(  createSphereZone(D3DXVECTOR3(3.0f, -20.0f, 3.0f), 15.0f));  // le plan qui servira d’obstacle  g\_vecZones.push\_back(createPlaneZone(  D3DXVECTOR3(0.0f, -50.0f, 0.0f), D3DXVECTOR3(0.0f, 1.0f, 0.0f)));  // créé un émetteur dans toutes les directions  g\_vecEmitters.push\_back(  createRandomEmitter(-1, false, 500, 100.0f, 10.0f, 20.0f));  g\_vecInterpolators.push\_back(createDefaultInitializer(  D3DXVECTOR4(0.0f, 1.0f, 1.0f, 1.0f)));  g\_vecInterpolators.push\_back(createSimpleInterpolator(  D3DXVECTOR4(1.0f, 0.1f, 0.1f, 1.0f),  D3DXVECTOR4(0.0f, 0.0f, 1.0f, 0.0f))); |

|  |
| --- |
| // dans ces 2 interpolateurs, on se sert seulement de la coordonnée x  g\_vecInterpolators.push\_back(createSimpleInterpolator(  D3DXVECTOR4(0.0f, 0.0f, 0.0f, 0.0f),  D3DXVECTOR4(2.0f\*D3DX\_PI, 0.0f, 0.0f, 0.0f)));  g\_vecInterpolators.push\_back(createSimpleInterpolator(  D3DXVECTOR4(1.0f, 0.0f, 0.0f, 0.0f),  D3DXVECTOR4(4.0f, 0.0f, 0.0f, 0.0f)));  // initialise une particule  ParticleData particle;  particle.Position() = D3DXVECTOR3(0.0f, 0.0f, 0.0f);  particle.Age() = 0.0f;  particle.Velocity() = D3DXVECTOR3(  float(rand()) / float(RAND\_MAX) - 0.5f,  float(rand()) / float(RAND\_MAX) - 0.5f,  float(rand()) / float(RAND\_MAX) - 0.5f);  particle.Velocity() \*= 40.0f;  particle.Energy() = 1.0f;  particle.OldPosition() = D3DXVECTOR3(0.0f, 0.0f, 0.0f);  particle.LifeTime() = 5.0f;  particle.Color() = D3DXVECTOR4(1.0f, 0.5f, 0.2f, 0.5);  particle.ParamSize() = 1.0f;  particle.ParamMass() = 1.0f;  particle.ParamAngle() = 0.0f;  particle.ParamTexIndex() = 0.0f;  particle.ParamRotSpeed() = 0.1f;  particle.SqrDist() = 0.0f;  particle.Group() = 0.0f;  unsigned int t = GetTickCount();  particle.Random() = \*((float\*)&t);  // créé la liste de particules  for( int i = 0; i < MAX\_PARTICLES; ++i )  {  g\_vecParticles.push\_back(particle);  particle.Velocity() = D3DXVECTOR3(  float(rand()) / float(RAND\_MAX) - 0.5f,  float(rand()) / float(RAND\_MAX) - 0.5f,  float(rand()) / float(RAND\_MAX) - 0.5f);  D3DXVec3Normalize(&particle.Velocity(), &particle.Velocity());  particle.Velocity() \*= 40.0f;  }  } |

Fonctionnement :

* Remplissage de buffers de type ID3D11ShaderResourceView
* Remplissage de 2 buffers de particules de type ID3D11UnorderedAccessView
* Simulation sur le GPU
* Echange des 2 buffers de particules

## Génération de nombres aléatoires sur le GPU

|  |
| --- |
| float randomP(int dec)  {  // optimized standard minimal  uint randomSeed = asuint(newParticle[0].paRandom) + dec;  int tmp0 = 16807L \* (randomSeed & 0xFFFFL);  int tmp1 = 16807L \* (randomSeed >> 16);  int tmp2 = (tmp0 >> 16) + tmp1;  tmp0 = ((tmp0 & 0xFFFF)|((tmp2 & 0x7FFF) << 16)) + (tmp2 >> 15);  // correction of the error  if( (tmp0 & 0x80000000L) != 0 )  tmp0 = (tmp0 + 1) & 0x7FFFFFFFL;  newParticle[0].paRandom = asfloat(tmp0);  // find a random number in the interval  return (float(tmp0) - 1.0f) / 2147483646.0f;  } |

Inconvénient : ça surcharge newParticle[0].paRandom par contre ça génère des aléatoires comme sur le CPU, on ne voit pas de zone de vide.

Autre technique testée :

* Générer un tableau de nombre aléatoire sur le CPU et l’envoyer au GPU -> génère des zones vides visibles
* Créer un générateur aléatoire par groupe -> impossibilité d’avoir des générateurs totalement indépendant (malgré le fait qu’ils sont totalement indépendants en mémoire), des motifs apparaissent dans les générations

## Gestion des collisions sur le GPU

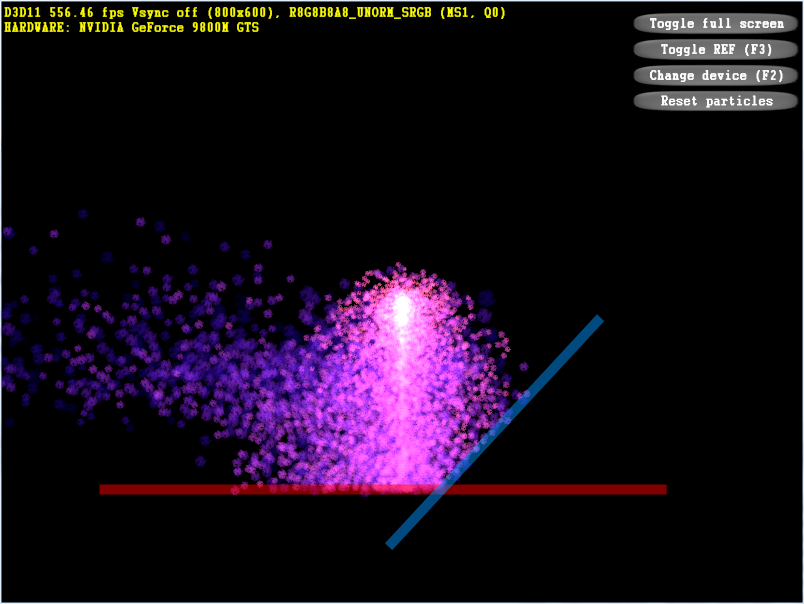
|  |
| --- |
| // test de collision entre le rayon formé par l’ancienne et  // la nouvelle position de la particule et la zone  if( intersects(modifier.moZone, particle.paOldPosition,  particle.paPosition, particle.paParamSize \*  groups[particle.paGroup].grRadius) )  {  // pas de subtilité, si il y a collision, on se base sur  // l’ancienne position de la particule  particle.paPosition = particle.paOldPosition;  // calcul de la normale à la zone à la position de la particule  float3 normal = computeNormal(modifier.moZone, particle.paPosition);  float dist = dot(particle.paVelocity, normal);  normal \*= dist;  particle.paVelocity -= normal; // tangent component  particle.paVelocity \*= modifier.moFriction;  normal \*= modifier.moBouncingRatio; // normal component  particle.paVelocity -= normal;  } |

# Conclusions

L’utilisation des compute shaders 4.0 limite beaucoup les possibilités, la plus grande limitation étant qu’il est possible d’attacher qu’un seul UAV au shader ; celui-ci étant destiné à la mise à jour des particules.

Il est également possible de s’en servir pour créer un générateur aléatoire, mais avec surement un stress assez important.

# Capture d’écran



Exemple avec 2 plans servant d’obstacle, symbolisé pour les lignes rouge et bleu ;

* Interpolation des couleurs, du rose pour les particules naissantes vers le bleu violet en fin de vie
* Non visible sur cette capture, les particules tournent sur elles mêmes
* Particules lancées dans toutes les directions
* Gravité réglée pour être dirigé vers le bas
* Le plan « bleu » restitue plus d’énergie d’où le flot de particules sur la gauche de l’image