UNIVERSITÉ TOULOUSE III PAUL SABATIER — Université de Toulouse

IRIT — Institut de Recherche en Informatique de Toulouse

Dynamic Sampling and Rendering of Algebraic Point Set Surfaces

Conception

16 Janvier 2017

William Caisson
Xavier Chalut
Christophe Claustre
Thibault Lejemble

A destination de :
Nicolas Mellado
David Vanderhaeghe
Adrian Basarab
Mathias Paulin

Table des matières

1 Introduction

Avec l'arrivée des méthodes de numérisation 3D telles que le scanner laser, la représentation par nuage de points devient bien plus courante pour représenter les surfaces d'objets en 3 dimensions. Cette représentation de plus en plus facile à obtenir, est assez complexe à visualiser. Les problèmes majeurs étant le bruit généré lors de l'acquisition des points et la quantité de données à manipuler si l'on souhaite représenter avec une grande précision la scène d'origine.

L'article de monsieur Guennebaud et ses collègues publié en 2008 propose une méthode réglant le problème des points bruités et permettant une visualisation pertinente de la scène même avec une plus faible quantité de données pour la représenter.

L'objectif de ce chef d'œuvre est de parvenir à une implémentation de cette méthode de visualisation de nuage de points au sein d'un plugin du moteur de rendu *Radium-Engine* . Dans un second temps, l'objectif sera de rendre l'implémentation la plus efficace possible en se basant sur l'utilisation de structures de données et d'algorithmes optimisés pour GPU par l'utilisation de la technologie CUDA.

Ce document à pour but d'aborder le problème d'un point de vue de l'architecture logiciel. Le système est d'abord rappelé dans sa forme globale avec les différents modules qui le compose. Les choix de structures de données et de classes sont ensuite décrits en détail. Finalement, un ensemble de tests unitaires sont définis et la dernière version du planning prévisionnel est présentée.

2 Vue d'ensemble du système

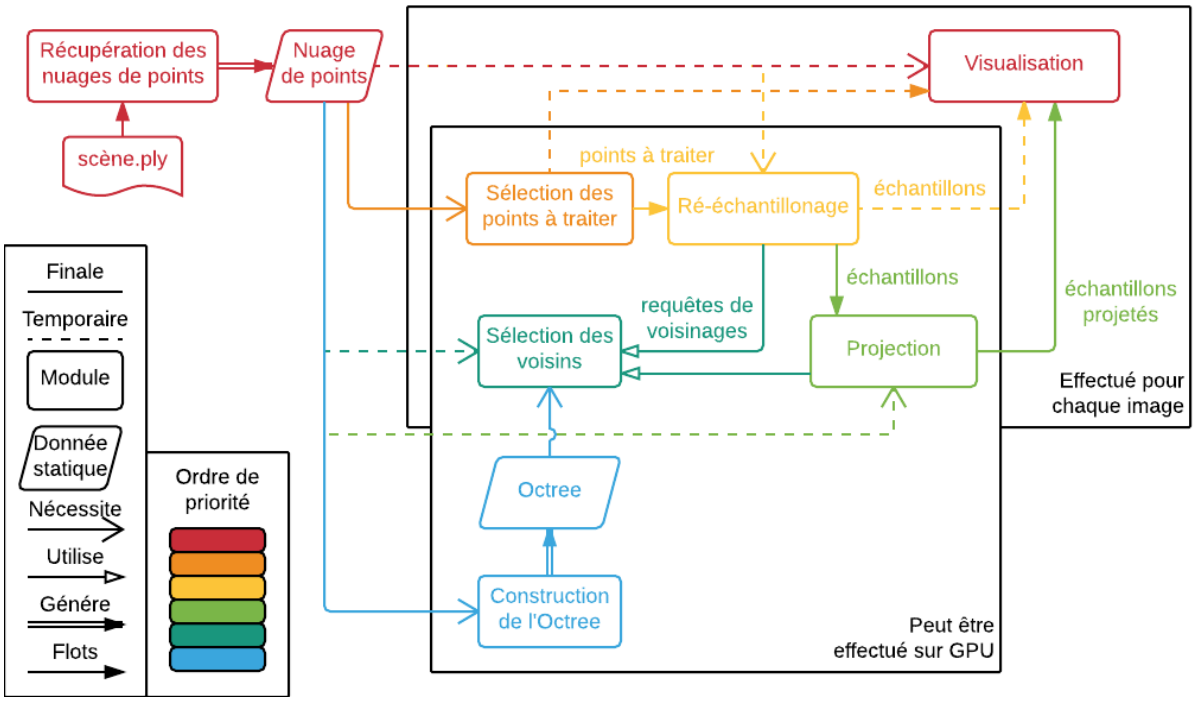

FIGURE 1 – Vue d'ensemble des différents modules

2.1 Module : Récupération des nuages de points

2.1.1 Description

Ce module doit permettre de lire et d'écrire un fichier au format .PLY et d'en extraire un nuage de points. Ce module doit pour cela définir une structure de représentation du nuage de points. Structure qu'il utilisera pour retourner le nuage de points aux autre modules. La lecture et l'écriture de fichiers au format .PLY se feront en utilisant la bibliothèque ASSIMP, déjà utilisée dans *Radium-Engine* .

Correction La lecture des *fichiers .PLY* ne s'effectue plus dans le cadre de ce projet. En effet *Radium-Engine* intègre déjà la lecture de différent type de fichier et est capable de lire des nuages de points. De fait, les tests de cette partie du module sont sous la responsabilité des développeurs de *Radium-Engine* et non sous la notre.

Entrée	Un *fichier .PLY*
Sortie	Un nuage de points
Objectif	Extraire d'un *fichier .PLY* un nuage de points

2.2 Module : Visualisation

2.2.1 Description

Ce module doit permettre de gérer l'affichage d'un nuage de points dans le visualiseur de *Radium-Engine* . Les points devront être correctement placés dans l'espace 3D de la scène. Afin de bien les visualiser, l'affichage des points se fera par l'affichage de disque correctement orienté suivant leur normale. Ce module doit également gérer correctement l'occultation possible entre les disques.

Entrée	Des points
Sortie	Visualisation correcte de ces points
Objectif	Afficher correctement des points dans un espace 3D du point de vue de la caméra

2.3 Module : Sélection des points à traiter

2.3.1 Description

Afin de réduire les temps de calculs, ce module doit permettre de sélectionner les points qui seront réellement visibles lors de la visualisation. Ceci dépend de la normale des points ainsi que de la position et de l'orientation de la caméra.

Entrée	Un nuage de points
Sortie	Des points à traiter
Objectif	Sélectionner uniquement les points qui ont besoin d'être traités/affichés

2.4 Module : Ré-échantillonnage

2.4.1 Description

Ce module doit permettre le ré-échantillonnage des points du nuage de points. Ce module fait varier le nombre d'échantillons en fonction de la courbure de la surface décrite par le nuage

de points en ce point, et en fonction de l'angle de vue de la caméra en ce point et de la position et de l'orientation actuelle de la caméra.

Entrée	Un point, ses voisins
Sortie	Plusieurs échantillons
Objectif	Générer un nombre correcte d'échantillon correspondant au point reçu en entrée en fonction de ses voisins

2.5 Module : Projection

2.5.1 Description

Via la *bibliothèque Patate* , ce module doit permettre la projection d'un ensemble de points sur l'approximation par des sphères algébriques d'une surface calculée à partir d'un nuage de points. La *bibliothèque Patate* permet déjà d'effectuer la projection d'un point sur une surface décrite par des sphères algébriques, le module doit donc permettre d'étendre cette utilisation à tout un ensemble de points. Aussi la *bibliothèque Patate* n'implémente actuellement que la méthode de projection orthogonale pour projeter les points sur la surface, ce module devra donc également permettre de projeter les points en suivant une projection presque orthogonale.

L'API Patate étant déjà utilisable sur GPU, il faudra s'assurer que ce module n'altère pas cette propriété afin de pouvoir facilement l'adapter pour la projection sur GPU.

Entrée	Un point, ses voisins
Sortie	Un point projeté
Objectif	Projeter correctement le point reçu en entrée sur la surface décrite par ses voisins

2.6 Module : Sélection des voisins

2.6.1 Description

Afin d'accélérer le calcul de la projection, ce module s'occupera de l'optimisation des requêtes de voisinage nécessaires au module *Projection* et au module *Ré-échantillonnage* .

Entrée	Un point
Sortie	Ses voisins
Objectif	Sélectionner efficacement les voisins du point reçu en entrée

2.7 Module : Construction de l'Octree

2.7.1 Description

Ce module doit permettre la construction d'une structure de donnée nommée Octree à partir d'un nuage de points. L'Octree permettra d'accélérer les requêtes de voisinage (voir Section 2.6). **L'Octree doit être reconstruit à chaque modification du nuage de points.**

Entrée	Un nuage de points
Sortie	Un Octree
Objectif	Générer efficacement l'Octree décrivant le nuage de points en entrée

3 Structures de données et classes

Cette partie présente les différents choix d'implémentation faits en terme de structures de données et de classes afin de répondre aux exigences de notre client. La façon dont sont représentés les nuages de points en mémoire, puis les classes liées à la mise en place d'un plugin Radium sont en premier lieu décrites. L'ensemble des classes directement liées à l'algorithme de l'APSS est finalement détaillé.

3.1 Représentation du nuage de points

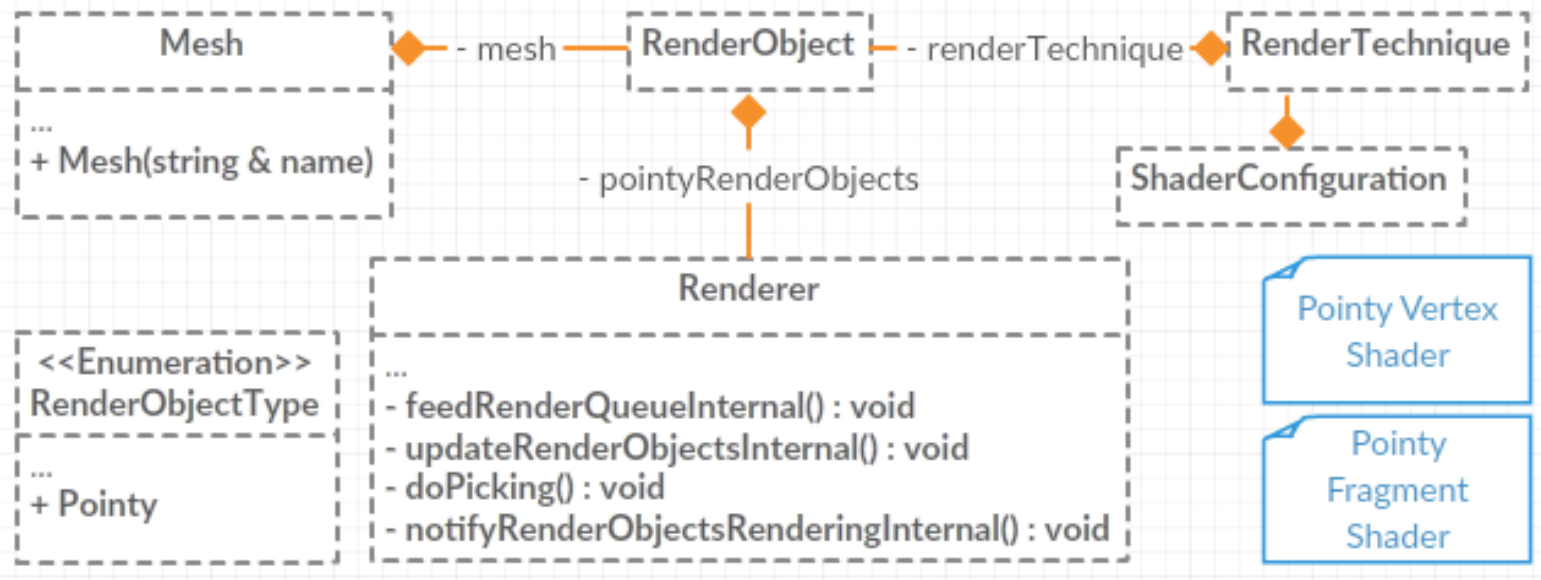

FIGURE 2 – Diagramme des classes de représentation du nuage de points et interactions avec *Radium-Engine*

Afin de représenter les points sur *Radium-Engine* nous aurons besoin de reprendre les structures actuellement utilisées et d'y ajouter un autre attribut *obligatoire* en plus des normales et positions, la couleur ! En effet, puisque on ne peut réellement appliquer des textures sur un nuage de points il nous faudra obligatoirement avoir une information de couleur sur nos données (si elles manquent lors de la lecture, une couleur par défaut sera donnée).

Nous aurons donc besoin de traiter nos données sur notre nuages de points directement depuis la classe **Mesh** au lieu de passer par une classe gérant uniquement la géométrie comme **TriangleMesh**. Aussi nous modifierons légèrement cette classe afin qu'elle puissent accepter un rendu de point (uniquement les lignes et les triangles sont possibles uniquement).

Grâce à cela nous pourrons en utilisant la classe **RenderObject** avoir des renderObject de **Mesh** de points sans problème. Il nous faudra pour cela ajouter le type **Pointy** dans les types disponibles pour un **RenderObject** et ajouter dans la classe **Renderer** une liste des **RenderObjects** de type **Pointy**.

Nous allons également écrire deux *shaders* à utiliser par défaut pour afficher des Points.

Modules

- La partie représentation des nuages de points du module de Récupération des nuages de points est implémentée par l'utilisation de la classe **Mesh**, la partie lecture des *fichiers .PLY* étant déjà implémentée finalement par Radium.
- Le module Visualisation est en partie implémenté par les deux *shaders*.

3.2 Le plugin Radium

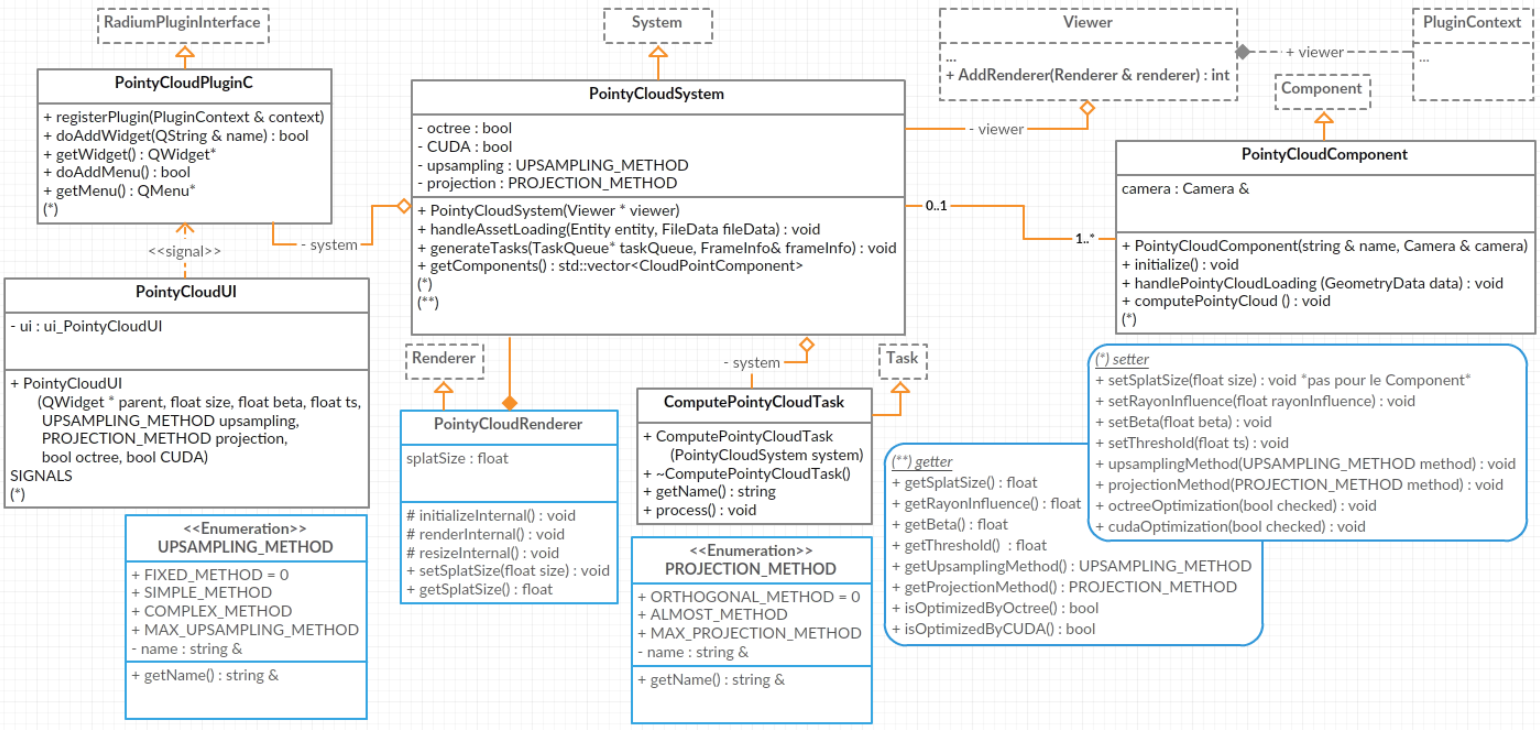

FIGURE 3 – Diagramme des classes de notre plugin Radium

Notre projet devant être en grande partie implémenté sous la forme d'un Plugin dans Radium, certaines classes sont obligatoires. Ce sont les classes **PointyCloudPluginC**, **PointyCloudSystem** et **PointyCloudComponent**.

PointyCloudUI est une classe qui nous servira à avoir une simple interface graphique pour la saisie des paramètres de l'APSS par exemple.

ComputePointyCloudTask est une classe nécessaire au mécanisme de tâches dans Radium, servant à définir une tâche (ou liste de tâches) que l'on souhaite exécuter entre 2 images générées par Radium. Dans notre cas, nous n'aurons qu'une seule tâche (**ComputePointyCloudTask**) qui appelle la fonction **computePointyCloud** de chaque **PointyCloudComponent** de notre **PointyCloudSystem**. Pour cela notre tâche devra avoir accès au Système, et celui-ci devra pouvoir lui donner ses components.

Nous ajouterons également une classe **PointyCloudRenderer**, héritant de **Renderer**, qui effectuera un affichage simple des **PointyCloud**. Dans le but de pouvoir activer notre Renderer, nous devrons pouvoir le donner au **Viewer** via une nouvelle méthode publique *AddRenderer()* et nous ajouterons le viewer aux données nécessaires à la création d'un Plugin (en ajoutant le viewer à la structure **PluginContext** qui est passée en paramètre au constructeur de l'interface **RadiumPluginInterface** qu'implémente **PointyCloudPluginC**).

Nous ajouterons le même mécanisme que celui implémenté dans le plugin ***FancyMesh*** pour la création à la volée de component contenant un Mesh dès l'ouverture d'un fichier (méthode *handleAssetLoading()* appelant *handlePointyCloudLoading()*). Ceci afin de bien créer des **PointyMesh** quand un fichier contenant un nuage de points à été ouvert (rien si ce n'était pas le cas).

Enfin, dans le but de pouvoir paramétrer notre affichage et notre reprojection, nous avons mis en place plusieurs paramètres configurables par l'interface (**PointyCloudUI**) et nous avons mis également la possibilité de choisir parmi plusieurs méthodes celle que l'utilisateur souhaite utiliser (via les énumérations UPSAMPLING_METHOD et PROJECTION_METHOD et les booléens CUDA et octree). Pour cela un chaîne de plusieurs setter et getter sera disponible dans l'architecture du Plugin.

Module Aucun module ici n'est implémenté directement, car il s'agit essentiellement de la structure nécessaire au fonctionnement de notre Plugin dans *Radium-Engine* . Cependant, la classe **PointCloudRenderer** aidera en partie l'implémentation du module Visualisation puisque qu'elle appellera avec les bon paramètres les *shaders* vus en section 3.1.

3.3 L'APSS

Cette partie s'occupera de la sélection des points, du sur-échantillonnage, de la reprojection et de la sélection des voisins.

Ici, les PointyCloudComponent se verront équipés de classes implémentant chacune des méthodes nécessaires pour effectuer notre APSS. Aussi, via le Design-pattern stratégie, les components auront accès à différente version de chaque méthodes.

Étant en temps réel, la rapidité d'exécution est très importante. L'utilisation de ce Design-pattern nous permet de limiter le nombre d'instruction conditionnelle (suite à la vérification de la méthode à utiliser par exemple).

Modules

- Le module Sélection des points à traiter est implémenté par la classe **UsefulPointsSelection**.
- Le module Ré-échantillonnage est implémenté par la classe **UpSampler** (et ses classes filles **UpSamplerSimple** et **UpSamplerAdvanced**.
- Le module Projection est implémenté par la classe **OrthogonalProjection** (et sa classe fille **AlmostOrthogonalProjection**).
- Le module Sélection des voisins est implémenté par les classes **NeigborsSelection**, **NeighborsSelectionWithOctree**, **Octree** et **OctreeLeaf**.

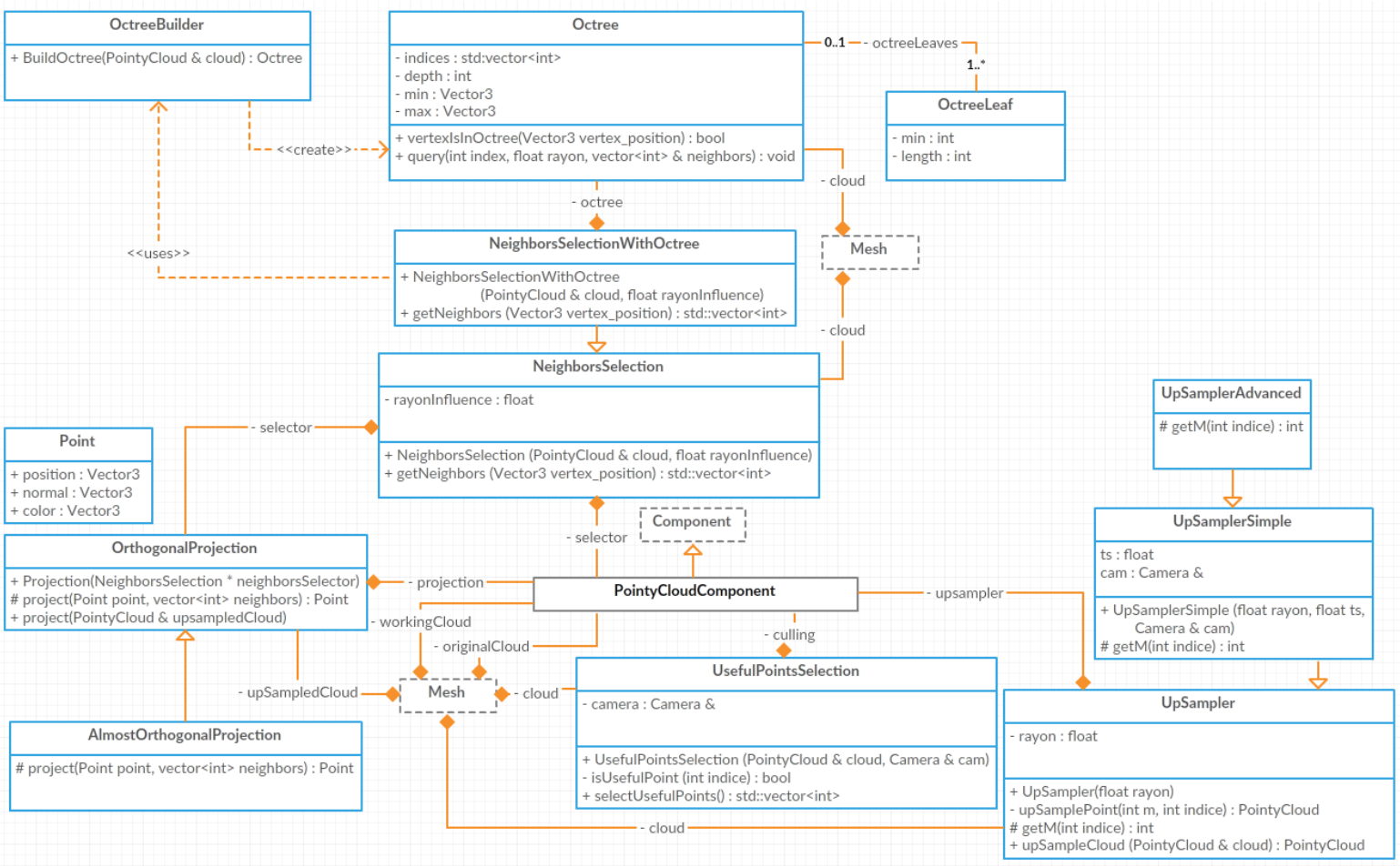

FIGURE 4 – Diagramme des classes effectuant l'APSS et interactions avec *Radium-Engine*

- Le module Construction de l'Octree par la classe **OctreeBuilder**.

4 Tests unitaires

Des test unitaires sont prévus afin de s'assurer du bon fonctionnement des différentes classes et méthodes. En voici quelques exemples clés, des tests unitaires additionnels pourront effectivement être ajoutés lors de l'implémentation des différentes classes. Un environnement de tests unitaires déjà existant dans *Radium* aidera à leur mise en place.

4.1 Test : Récupération des nuages de points

Description De base, Radium nous permet déjà de lire des fichiers *.PLY* grâce à la librairie *Assimp*. Il faut donc vérifier que les objets de type **PointyCloud** soient correctement créés.

Id	Classe/Méthode	Données d'entrée	Résultat attendus	Protocole de vérification
1	`PointyMeshSystem::handleAssetLoading()`, `PointyCloudComponent::handlePointyCloudLoading()`	Un fichier contenant un nuage de points avec peu d'éléments	Le nuage de points créé doit contenir les même données que le fichier d'entrée	Affichage de la structure de données et comparaisons avec les données du fichier

4.2 Test : Visualisation

Description Le test de visualisation est difficilement vérifiable. Nous ferons donc une simple validation visuelle en constatant qu'il n'y a pas d'aberrations dans les modèles chargés. Des comparaisons peuvent aussi être effectuées avec des visualisations de nuages de points dans des logiciels tierces tels que *MeshLab*.

Id	Classe/Méthode	Données d'entrée	Résultat attendus	Protocole de vérification
2	`PointyCloudRenderer` et les différents *shaders* utilisés	Un nuage de points	Visualisation correcte du nuage	Vérification visuelle ou comparaison avec un logiciel tierce

4.3 Test : Sélection des points à traiter

Description L'étape de la sélection des points à traiter est effectuée par la classe **UsefulPointsSelection**. La méthode *isUsefulPoint()* peut être facilement testée sur un nuage de points contenant un seul élément dont on connaît la direction de sa normale. La méthode est ainsi testée pour plusieurs orientations de caméra définies manuellement. Le méthode *selectUsefulPoints()* peut être testée sur des nuages de points simples et en contrôlant le vecteur d'indices retourné.

Id	Classe/Méthode	Données d'entrée	Résultat attendus	Protocole de vérification
3	`UsefulPointsSelection:: isUsefulPoint()`	Un point et une caméra orientée vers l'arrière du point et alignée sur sa normale	Retourne *false*	Un fichier *.PLY* est rempli manuellement. La caméra est orientée grâce à sa méthode *setDirection()*.
4	`UsefulPointsSelection:: isUsefulPoint()`	Un point et une caméra orientée vers l'avant du point et alignée sur sa normale	Retourne *true*	Un fichier *.PLY* est rempli manuellement. La caméra est orientée grâce à sa méthode *setDirection()*.
5	`UsefulPointsSelection:: isUsefulPoint()`	Un point et une caméra orientée vers l'arrière du point et décalée de 45° par rapport à sa normale	Retourne *false*	Un fichier *.PLY* est rempli manuellement. La caméra est orientée grâce à sa méthode *setDirection()*.
6	`UsefulPointsSelection:: isUsefulPoint()`	Un point et une caméra orientée vers l'avant du point et décalée de 45° par rapport à sa normale	Retourne *false*	Un fichier *.PLY* est rempli manuellement. La caméra est orientée grâce à sa méthode *setDirection()*.
7	`UsefulPointsSelection:: selectUsefulPoints()`	Un nuage de points dont les 10 premiers points sont alignés selon l'axe `+X` et les 10 derniers selon l'axe `-X`, et une caméra orientée dans la direction `-X`	Un vecteur contenant des indices allant de 1 à 10.	Un fichier *.PLY* est rempli manuellement. La caméra est orientée grâce à sa méthode *setDirection()*. Le vecteur retourné est finalement affiché

4.4 Test : Ré-échantillonnage

Description Inspiré de l'article, nous afficherons les points de différentes couleurs suivant les différents niveaux de ré-échantillonnage. Puis nous allons nous assurer visuellement que l'échantillonnage reste correcte. Les zones de forte courbure se distingueront ainsi des portions du nuage plus homogènes.

Id	Classe/Méthode	Données d'entrée	Résultat attendus	Protocole de vérification
8	`UpSampler:: upSampleCloud()`	Un nuages de points de courbure variable	Les zones courbées se distinguent par leur haut niveau d'échantillonnage	Vérification visuelle grâce à un système de couleur

4.5 Test : Projection

Description Étant donnée la difficulté de vérifier le calcul de l'APSS et parce qu'il fait notamment appel à la librairie *Patate*, ce module ne contient qu'un test canonique : un nuage de points contenus dans un plan projettera tout échantillon dans ce même plan.

Id	Classe/Méthode	Données d'entrée	Résultat attendus	Protocole de vérification
9	`OrthogonalProjection:: project()`	Un nuage de points contenus dans un plan de plus 1000 points légèrement bruitées	Toute les projections se feront dans le même plan (non bruitées) décrit par le nuage initial	Affichage des projections

4.6 Test : Sélection des voisins

Description Le module de sélection des voisins, implémenté par la classe **NeighborsSelection** et sa classe dérivée **NeighborsSelectionWithOctree**, est testé en fournissant un nuage de points connus pour lequel le résultat d'une requête de voisinage d'un points est parfaitement connu. Les tests unitaires de la construction de l'Octree (voir Section 4.7) doivent d'abord être validés.

Id	Classe/Méthode	Données d'entrée	Résultat attendus	Protocole de vérification
10	`NeighborsSelection:: getNeighbors`	Un nuage de points dont tous les points sont espacés de plus de 2 unités de longueurs, avec un rayon d'influence de 1 unité de longueur	Un vecteur vide pour toutes requêtes	Le fichier *.PLY* est créé manuellement ou de manière procédurale puis des requêtes de voisinages sont effectuées pour tous les points du nuage
11	`NeighborsSelection:: getNeighbors`	Un nuage de point contenant notamment une sphère échantillonnée de rayon de 1, et un rayon d'influence de 2 unités de longueur	La requête du voisinage du centre de la sphère retourne tous les indices des points de la sphère	Création procédurale du fichier *.PLY* puis comparaison des indices retournés avec ceux des points de la sphère

4.7 Test : Construction de l'Octree

Description La classe **OctreeBuilder** est responsable de la construction de l'Octree et peut être testée grâce à des exemples canoniques de nuages de points. Le calcul de la boite englobante doit aussi être testé.

Id	Classe/Méthode	Données d'entrée	Résultat attendus	Protocole de vérification
12	`OctreeBuilder:: buildOctree()`	Un nuage vide	Un Octree contenant la racine vide et aucune autre feuille	Chargement d' un fichier *.PLY* vide, et affichage des feuilles de l'Octree créé
13	`OctreeBuilder:: buildOctree()`	Un nuage de plusieurs points dont on connaît les points correspondant aux coins minimal et maximal	Les *min* et *max* de l'Octree doivent correspondre à ceux attendus	Création manuelle d' un fichier *.PLY*, puis contrôle manuel des *min* et *max*
14	`OctreeBuilder:: buildOctree()`	Un nuage de points répartis uniformément dans un cube	Un Octree contenant un point dans chaque feuille	Création procédurale d'un tel fichier d'entrée, puis comptage du nombre de points par feuille de l'Octree

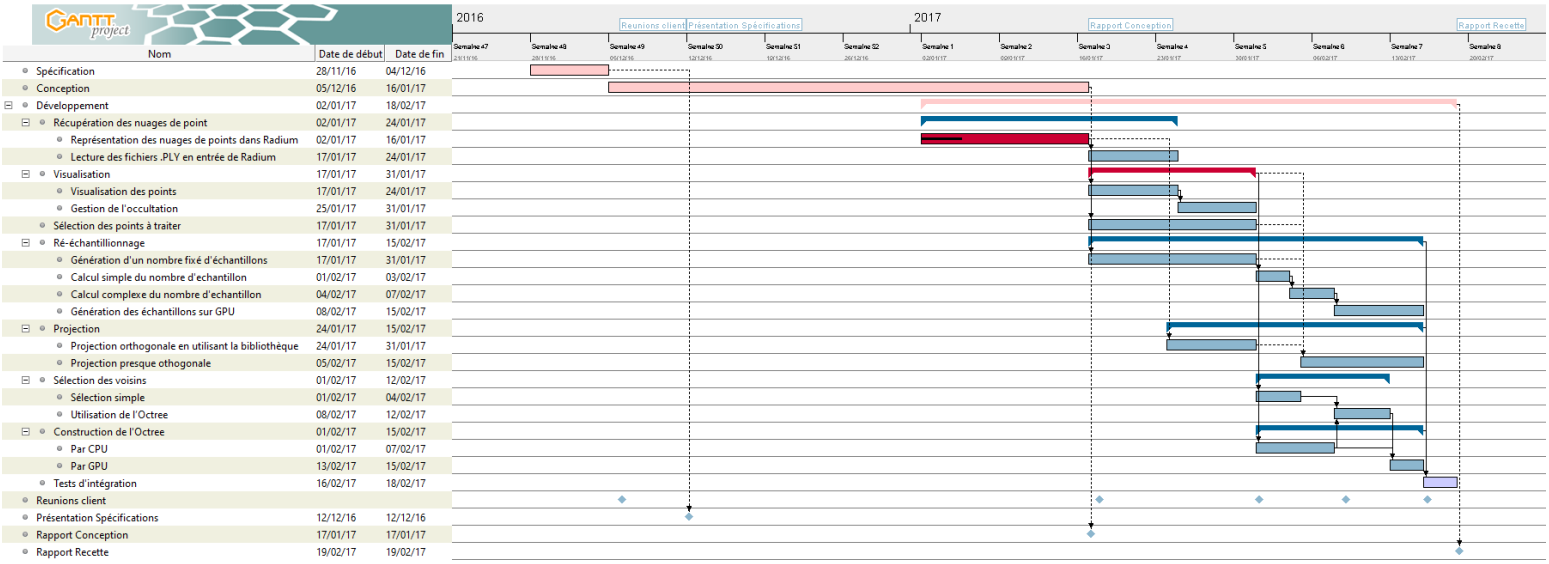

FIGURE 5 – Planning prévisionnel Mise à Jour. En *rose* sont représentées les phases du projet, en *bleu* les tâches du développement, en *rouge* les tâches/modules critiques du développement et en *bleu clair* les tests d'intégration (servant aussi de marge au projet)

5 Planning prévisionnel & Analyse de risque

5.1 Planning

Modification du planning Le planning provisionnel a subit des modifications en raison d'un retard sur l'élaboration de la conception du projet. En effet, la conception de l'architecture du plugin a nécessité un temps supérieur au temps prévus. Ce retard impacte donc tout le projet et produit un décalage d'environ 3 jours sur la plupart des tâches imputées sur le temps des tests d'intégrations. Certaines taches ont également été raccourcies notamment celles sur la sélection des points et les premières versions du ré-échantillonnage et de la projection. Néanmoins, bien que l'étape de développement ait pris du retard, il est à noter que ce temps a été principalement utilisé pour l'élaboration d'une architecture solide qui pourrait avoir des impactes bénéfiques sur la suite du développement.

5.2 Gestion des risques

Pour rappel, voici ci-dessous la liste des taches et modules principaux avec l'analyse des risques associés :

Modules/Tâches clés :

- Représentation des nuages de points dans Radium : **Tâche critique**. Tant que cette tâche n'est pas effectuée, nous ne pouvons stocker les points et nuage de points.
- Visualisation des points : **Module critique**. C'est ce module qui nous permettra de visualiser la surface lorsque tout les autres modules seront terminés.
- Sélection des points à traiter : **Module très important**. L'application est en temps réel, ce pré-traitement sera à priori celui qui allégera le plus les temps de calculs.
- Génération d'un nombre fixé d'échantillons : **Tâche importante**. Sans cette fonctionnalité implémentée, aucune raison d'effectuer de la reprojection.

- Projection en utilisant l'API : **Tâche importante**. La visualisation d'une surface lisse calculée à partir du nuage de points ne pourra se faire sans cette étape.
- Lecture des fichiers .PLY en entrée de Radium : **Tâche assez importante**, cette tâche est d'une importance plus faible mais permettra de visualiser n'importe quelle scène et permettra de tester plus en profondeur nos modules.

Les risques précédemment détaillés dans le document de Spécification restent inchangés dans l'état actuel du projet. On notera néanmoins que le retard de planning introduit dans la sous-section 5.1, entraîne une réduction du temps alloué à des tâches critiques qui pourrait se répercuter sur les tâches moins critiques notamment d'optimisation GPU.