Conception détaillée du PBRadium

Hugo Rens, Matthias Roves et Jehan-Antoine Vayssade,
Projet proposé par le Pr. M. Paulin, de l'équipe IRIT/STORM.

17 janvier 2018

Table des matières

Introduction

Dans ce rapport, il sera détaillé notre vision de la conception du chef-d'œuvre. Une vue aussi détaillée que possible sera présentée, ainsi que les différentes classes et structures de données impliquées. Il sera aussi présenté les protocoles de tests, et un raffinement du planning prévisionnel et des risques.

Pour rappel, le contexte de ce projet est le chef-d'œuvre de Master 2 ainsi que le développement sur le long-terme du Radium Engine, le moteur interne utilisé par l'équipe STORM, de l'IRIT.

Le chef d'œuvre consiste en l'implémentation en tant que **plugin du Radium Engine**, d'un renderer supportant les **matériaux de type Disney**, un éclairage direct à base de **sources polygonales** et un éclairage indirect à base de **sources ponctuelles nombreuses** (many-lights). *Le rapport n°1 sur les méthodes et algorithmes explicite plus largement ces notions et fait une liste exhaustive des techniques, et le rapport n°2 fournit les spécifications du projet.*

1 Vue d'ensemble détaillée du système

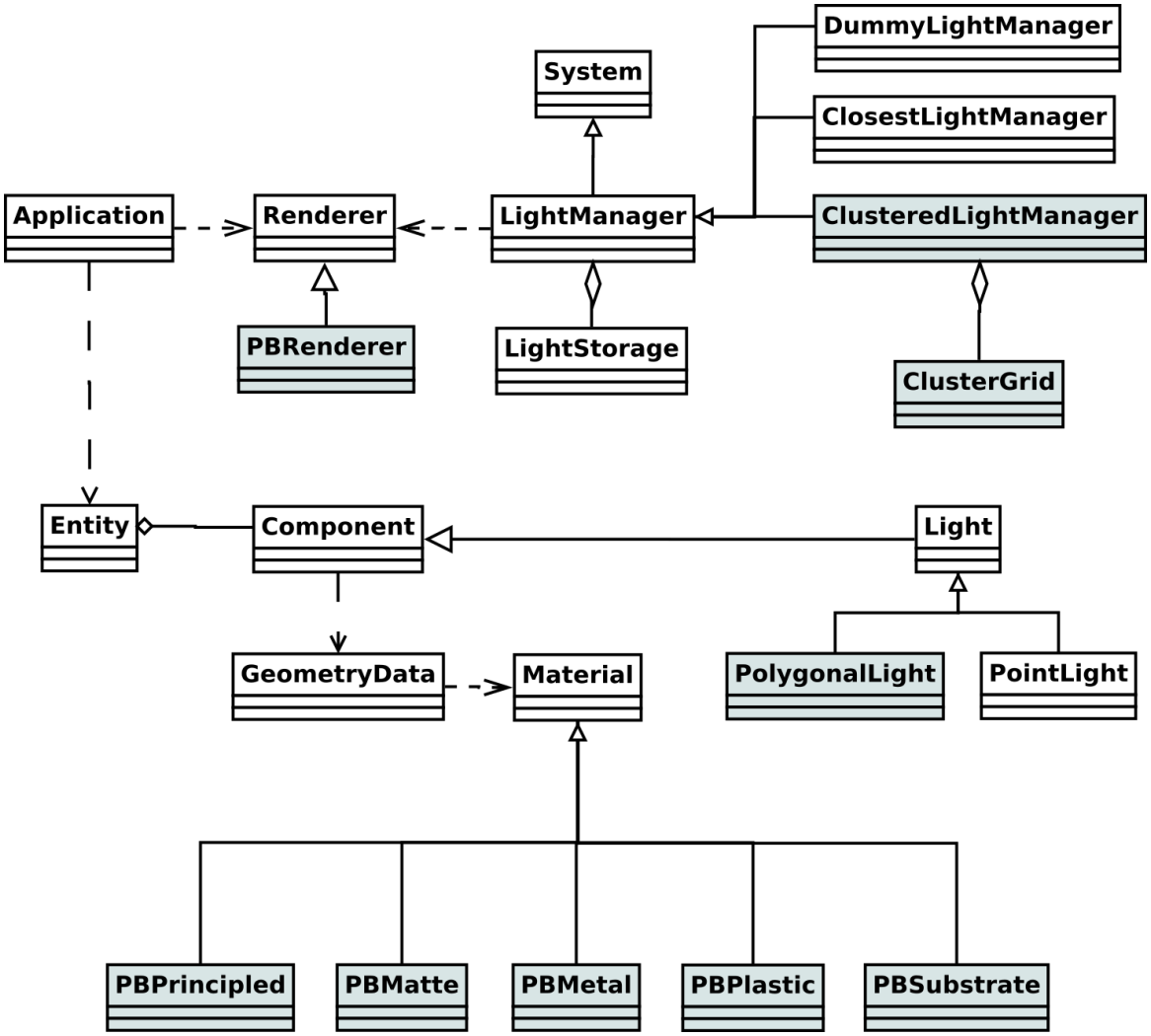


FIGURE 1 – Vue générale du système. Les cases grises sont les composants appartenant au plug-in.

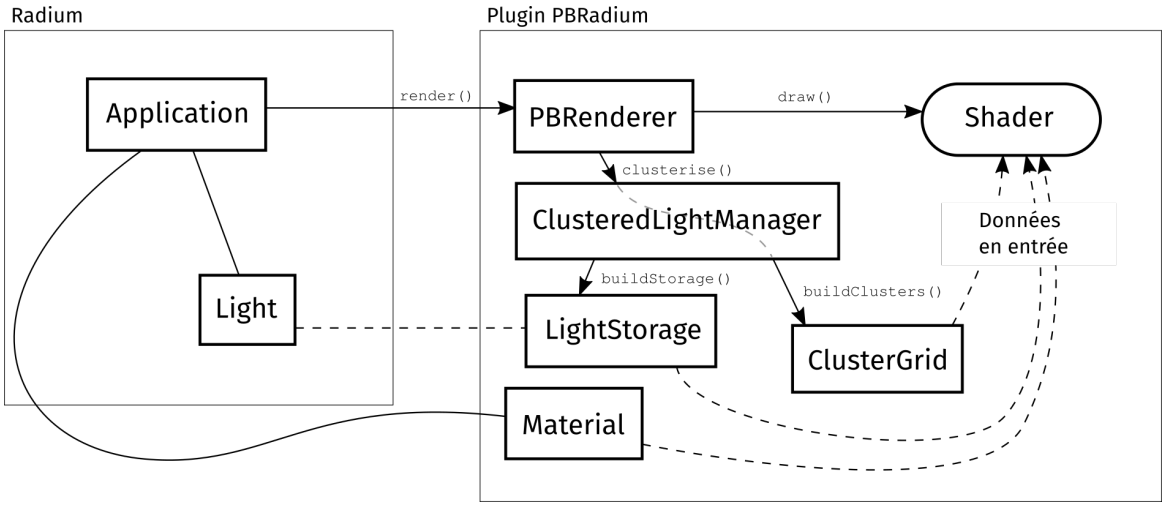


FIGURE 2 – Collaboration des composants.

2 Classes et structures de données

2.1 Dans le Radium

`Light` Définit l'interface pour une source lumineuse. Les interfaces sont déjà disponibles, mais nous avons pensé à une autre architecture où les `Light`s seraient des `Component`s. L'utilité serait de pouvoir attacher les lumières aux entités [1] et de pouvoir les transformer de manière plus unifiée.

Actuellement, les lumières ont leur propre système dissocié en terme de transformation et d'information. Elles sont données de façon brute au renderer par une référence `std::vector<Ligth*>`. Cette interface existante définit les propriétés basiques des lumières de type Phong ; ceci ne correspond pas nécessairement à la représentation que nous souhaitons avoir ici. Nous avons donc décidé d'apporter quelques modifications au Radium afin de proposer une interface plus générique. De plus il n'existait pas de possibilité pour lier une `Entity` à une `Light`, ce qui posait souci notamment pour le suivi et l'utilisation de certaines données dans le cadre des `PolygonalLight`s et des sources pour le many-lights.

`PolygonalLight` Les lumières polygonales héritent de l'interface des sources lumineuses définies précédemment. Notamment, elles tirent partie de leurs liens avec une entité pour avoir un mesh permettant 1) de connaître les points du polygone et 2) de les afficher. Elles gèrent, comme les matériaux, le code glsl correspondant, et la matrice précalculée à envoyer.

`LightManager` L'interface `LightManager` a pour objectif de gérer un groupe de lumières. Cette construction a plusieurs avantages. Le premier est de permettre de grouper les lumières appliquées d'une même manière. Par exemple, on peut appliquer un lot de lumières comme on le fait dans le pipeline forward standard, ou alors n'utiliser que les N plus proches lumières. Cela permet aussi de gérer efficacement le transfert des lumières à la GPU, en lui déléguant l'implémentation. Dans le cas des sources ponctuelles pour le many-lights, on imagine une implémentation de cette interface (nommément `ClusteredLightManager`) envoyer les sources dans un format organisé en clusters (voir 2.4.2).

1. Et par la même d'associer des `Mesh`s pour les sources polygonales.

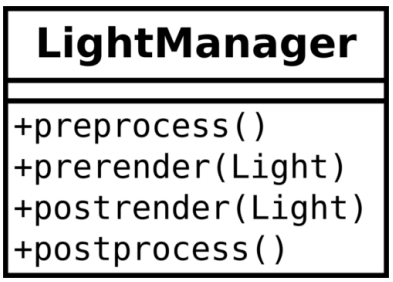


FIGURE 3 – Classe **LightManager**.

Voici un exemple typique d'utilisation du LightManager par le renderer (code non-contractuel) :

```
LightManager& lm;
lm.preprocess();
for (const Light& light : _lights)
{
    lm.prerender(light);
    // Render...
    lm.postrender(light);
}
lm.postprocess();
```

Dans le cadre de l'implementation du ClusteredLightManager la méthode preprocess a pour optique de calculer la clusterisation de la scène en utilisant le z-prepass, puis nous itérerons sur les `Light`s pour effectuer l'association cluster-light. Cela consiste en un premier filtrage des lumières, les méthodes postrender et postprocess permettrons de nettoyer certaines données ou transformations si nécessaire.

`LightStorage` L'interface `LightStorage` propose l'interface standard d'un conteneur, à savoir la création et l'accès. Elle permettra de stocker les lumières au sein du plugin.

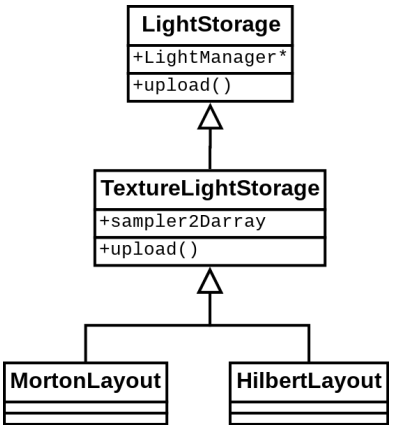


FIGURE 4 – Interface `LightStorage` et implémentations possibles.

La méthode `upload` permettra d'envoyer au GPU les informations structurées et mises à jour. Les `LightStorage`s sont utilisés par les `LightManager`s. *Le détail de la structure employée pour stocker les lumières ponctuelles nombreuses est fourni en 2.4.2.*

Il est important de noter que les sources polygonales et les sources ponctuelles sont stockées de manière indépendantes, mais elles le sont toutes les deux à l'aide d'un `LightStorage`.

`Material` Cette interface propose les signatures des fonctions de création et de modification d'un matériau. Il est sous-classé par tous les matériaux.

2.2 Dans le plugin

ClusterStorage Une grille de clusters, qui représente la liste des clusters. Cette partie détermine les méthodes de stockage des clusters et permettra notamment de définir la structure d'indirections utilisée pour représenter l'association entre les clusters et lumières.

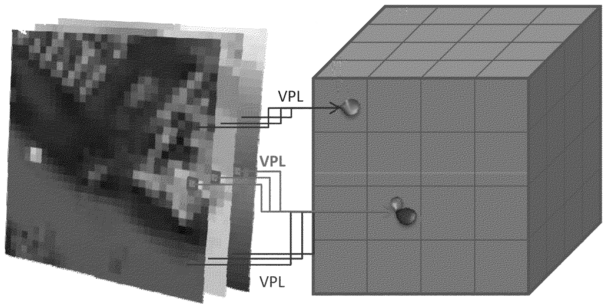


FIGURE 5 – Schéma d'un cluster contenant des sources de lumières.

Dans notre cas d'utilisation, par exemple avec des VPL chargées depuis l'extérieur, chaque cluster contiendra une liste des indices des lumières qui impactent le-dit cluster en terme de lumières, en fonction de la vue, permettant l'optimisation telle que décrite dans le documents **Méthodes et Algorithmes**.

Le schéma ci-dessus représente l'étape d'association lumière-cluster, nous avons à gauche la scène contenant les lumières vues depuis la caméra, et à gauche les clusters. La seconde étape consiste à venir calculer la position du pixel afin de trouver le sous-ensemble de lumières impactant la zone du pixel.

Cette structure d'indirection utilisera quant à elle la structure `LightStorage` dont la représentation choisie pour les lumières est détaillée dans la partie 2.4.2.

PBRenderer L'objet du plugin [2] qui effectuera le rendu. Il n'hérite pas du `ForwardRenderer` du Radium, car son organisation actuelle est trop monolithique ; toutefois nous utiliserons le code présent pour faire notre Z-Prepass. Il fera appel au LightManager à chaque frame pour lui laisser la possibilité de faire du précalcul qui pourrait lui être nécessaire (nous pensons notamment au clustering).

2.2.1 Materials

Les différents matériaux implémenté dans le plugin sont des dépendances direct de la classe `Material` définis dans le Radium engine. Ce liens avec le moteur nous permettant de garantir le fonctionnement de ces techniques de rendu même dans d'autre type de renderer.

Chaque matériaux ainsi définis va amener avec lui un ensemble de paramètres basée sur ceux que l'on peut retrouver dans PBRT. Chacun de ses paramètres (voir table 1) seront définis lors du chargement des objets dans le moteur ou encore par l'utilisateur.

Afin de faire le lien avec les fonctions des BSDF sur la carte graphique, l'enregistrement des matériaux et des fonctions de rendu se fera à l'aide d'une fonction interne à la classe.

2.3 Shaders

Pour fournir les programmes GLSL répondant à la spécification de chaque type de matériaux et de chaque type de lumières, il faudrait écrire un shader program pour chaque combinaison. Pour limiter la redondance au maximum, nous prévoyons de générer toutes les combinaisons de types de matériaux et de lumières à partir d'un programme glsl template, et, en mettant à profit la capacité du Radium à gérer les `#include` de glsl, générer automatiquement tous ces shaders.

2. Le code appartient au plugin, mais l'instance effective est stockée dans la liste des Renderers du Radium.

Table 1 – Ensemble des paramètre de chaque matériaux

BSDF Disney	BSDF Plastique	BSDF Métallique	BSDF Matte	BSDF Substrate
Couleur diffuse	Couleur diffuse	Couleur diffuse	Couleur diffuse	Couleur diffuse
Couleur spéculaire	Couleur spéculaire	Coefficient d'absorption		Couleur Spéculaire
Rugosité	Rugosité	Rugosité	Rugosité	
Anisotropie		Rugosité selon $\vec{u}$		Rugosité selon $\vec{u}$
Indice de réflexion		Rugosité selon $\vec{v}$		Rugosité selon $\vec{v}$
Paramètre métallique				
Paramètre d'effet soie				
Couleur de l'effet soie				
Paramètre de Clearcoat				
Brillance du Clearcoat				

L'implication majeure de ce choix est le respect strict d'une interface/convention de l'écriture des segments de shaders.

Un avantage transversal de ce choix est de rendre facile la validation des shaders pour des versions d'OpenGL données. Nous pouvons donc vérifier simplement et systématiquement le support de nos shaders pour la version 4.1 dans l'hypothèse d'un retour sur macOS.

Cela permettra éventuellement d'amorcer ou faciliter une transition dans l'utilisation des shaders au sein du Radium vers un modèle de shader séparables et des shader subroutine.

Nous pensons enfin que ce ne sera que bénéfique par rapport à une solution à base d'un seul gros shader composite, qui impliquerait des branchements sur GPU.

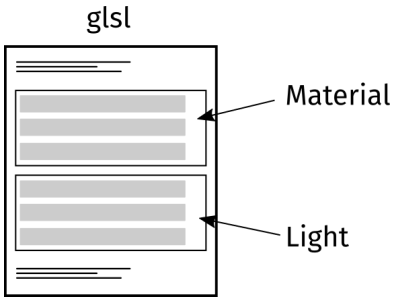


Figure 6 – Schéma d'un shader "template". Les zones grises représentent des fonctions.

2.4 Détails notables

2.4.1 Modifications du Radium

Malgré notre implémentation en plugin, certaines parts du *Radium Core* doivent évoluer.

Le `LightManager` (présenté en 2.1) implique de changer l'implémentation courante des lumières au sein des Renderers déjà implémentés. Il en va de même pour l'association des lumières aux `Entity`. Par exemple, elles utiliseront directement les propriétés des entités, notamment en terme de gestion des transformations.

2.4.2 Structure de stockage

Les lumières sont stockées côté CPU dans une structure telle que présentée en figure 7. Nous les transférons de manière contiguës au GPU.

Sur ce schéma, il est représenté le stockage des sources. Nous choisissons de l'implémenter sous forme d'un `sampler2DArray` (aggrégat de textures 2D en layers). Ainsi, une couche représente un même paramètre pour

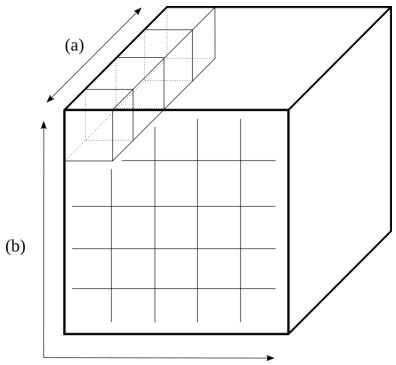


FIGURE 7 – Layout des sources.

chaque lumière, et chaque couche est caractérisée par un paramètre. De cette manière tous les texels de même coordonnées (x, y) sont les paramètres d'une source. Nous bénéficions donc de la taille conséquente d'une telle structure sur la GPU. De plus, on notera que l'on évite des problèmes liés à l'interpolation des valeurs, ce qui n'est pas souhaitable.

Qui plus est, la hiérarchie de cache de la GPU étant très évoluée, nous pouvons, avec des efforts supplémentaires, faire en sorte que deux lumières proches dans l'espace le soient dans la texture ; ceci permettra de réduire le coût des accès mémoire. Les efforts mentionnés sont l'organisation sur (x, y) des sources en fonction d'une courbe de Hilbert.

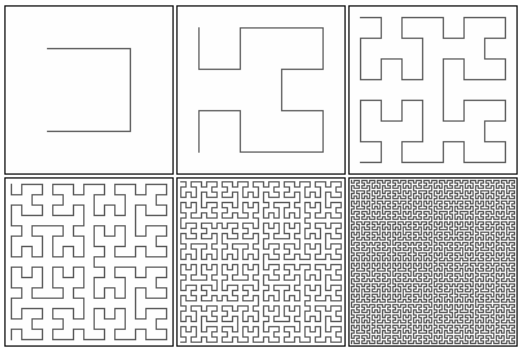


FIGURE 8 – Courbe de Hilbert, qui maximise la localité.

Dans le cadre du many-lights, nous enverrons ces données côte à côte avec une structure d'indirections qui représente le clustering.

2.5 Résumé

Sur **CPU**, le Radium charge des lumières. Le plugin, toujours sur CPU, se sert de ces sources (de différents types) pour créer plusieurs `LightManager`s (un par type). À chaque frame, le plugin commence par une étape de **clustering sur GPU** lors de laquelle il associe les sources ponctuelles à des clusters, qui sont dépendants de la vue. Autrement dit, il segmente la scène et créée des groupes de sources. Une fois que ceci est fait, le Renderer demande à chaque objet de se rendre, ce qu'il fait **sur GPU à l'aide des shaders composites** créés par le plugin.

Beaucoup de choses se jouent dans ces shaders composites. Les accès aux lumières, les évaluations des fonctions et les paramètres doivent être suivant bien organisés pour être indépendants. Un shader typique pourra donc récupérer la liste des lumières à traiter, pourra évaluer une lumière, et pourra évaluer un matériau.

3 Tests

3.1 Clustering

Il est relativement simple de tester le clustering. Pour une configuration de caméra donnée et une liste de lumières donnée, l'accès à chaque cluster donnera un sous-ensemble de lumières déterminé. Deux informations sont indispenssable a tester, d'une part la clusterisation de la scene en fonction du Z-prepass, on pourras derterminer quelques scenes simple pour vérifier la correspondance dont l'association est prédictible telque des boites ou spheres. D'autre part, tester la bonne association entre les clusters et les lumières.

3.2 Éclairage et matériaux

L'éclairage, ainsi que les matériaux, peuvent être testées d'une manière plus détournée. La seule manière fiable serait de faire un dump des données acquises par un intégrateur PBRT lors du calcul des dits matériaux et textures, et de les comparer avec les données que l'on obtient via le plugin. Ces données, calculées sur GPU, devront donc être récupérées d'une manière ou d'une autre.

Nous mettrons donc en place la base de tests permettant de créer ces dumps, en effectuant un rendu sur GPU (ce qui implique un contexte OpenGL), puis en les enregistrant dans un format d'image simple. Une fois que ce sera fait, une comparaison avec la même image effectuée par PBRT fournira l'indice de ressemblance. Si il y a des différences notables, le test échoue.

Ces tests sont substantiellement longs à établir. Nous nous concentrerons, dans le temps imparti au projet, au développement du test côté Radium, qui réalise le rendu en OpenGL et créée la texture contenant les résultats. Les textures en sorties seront constituées différemment pour les matériaux et les lumières.

Pour les lumières, les paramètres caractérisant une valeur unique du test sont la rugosité α du matériau et l'angle d'incidence Θ, pour un polygone donné. Les deux dimensions de la texture sont donc la rugosité et l'angle d'incidence (à l'instar de la texture précalculée, d'ailleurs), et chaque élément contient l'énergie arrivant au point intégrée sur un polygone donné, qui caractérisera la texture. Ainsi donc nous obtiendrons un dossier d'images, chacune caractérisée par un polygone et de la forme :

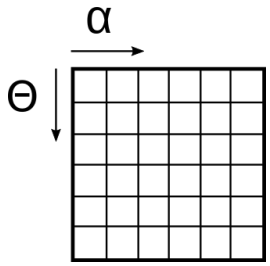


FIGURE 9 – Texture contenant les valeurs d'éclairement calculée pour le polygone caractérisant l'image. α représente la rugosité d'un matériau.

Pour les matériaux de type Disney, les paramètres caractérisant une valeur unique du test sont principalement l'angle d'éclairage Φ et l'angle de vue Θ, qui représenteront les deux dimensions de la texture. Pour chaque matériau, des paramètres spécifiques leurs seront ajoutés, donnant lieu à un dossier d'images de la forme :

FIGURE 10 – Dump d'un matériau. Le Θ représente l'angle de vue par rapport au matériau. Le Φ représente l'angle de la lumière incidente. À droite, une image similaire qui est la représentation MERL-100 d'une BRDF. Ces images représente l'évaluation de la BRDF pour des angles de vue et de lumière donnés.

4 Planning prévisionnel

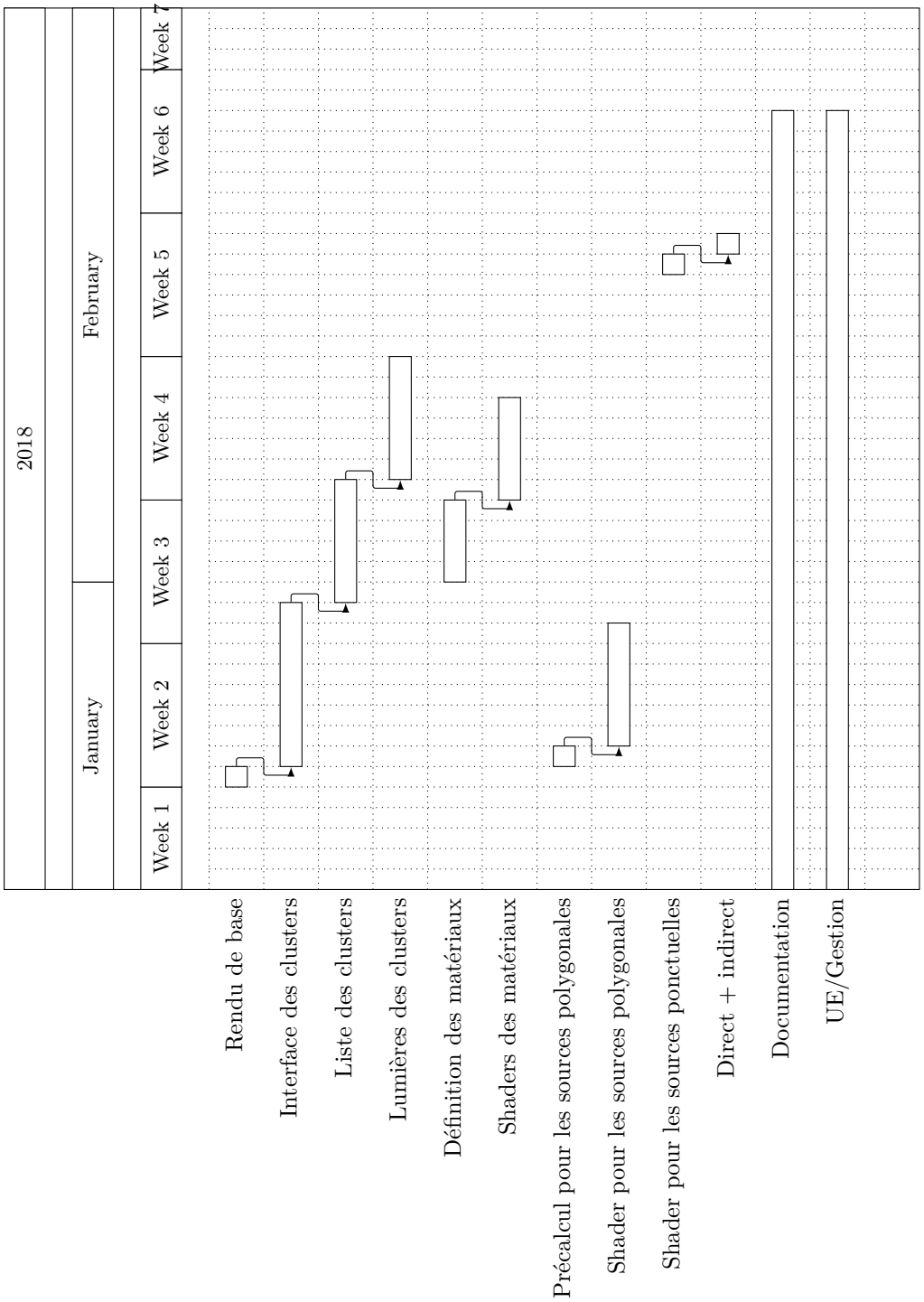
Dans cette section se trouve une rétrospective du planning présenté dans le rapport de spécifications, et un planning mis à jour. Les durées des tâches ont été compressées au mieux pour pouvoir les réaliser avec les effectifs prévus initialement. Il est à noter que pour l'instant, nos estimations étaient pessimistes. Une fois compressées, elles deviennent un peu plus optimistes à réaliser, **mais ces estimations sont toujours réalisables**. Nous ne pourrons pas (à priori) recourir à cette échappatoire une seconde fois.

De plus, la tâche "Précalcul pour les sources polygonales" est passée de 4 à 1 jour estimé suite à la permission d'utiliser les données de précalcul fournies par les auteurs de la publication. Ce qui permet donc de raccourcir la durée où le développeur assigné était occupé, lui permettant de rejoindre les autres sur les tâches qui ont dues être comprimées.

Le tableau qui suit présente donc les tâches, réévaluées.

Tâche	Jours /Homme	Nombre de développeurs
Rendu de base	1	1
Interface des clusters	8	2
Liste des clusters	6	2
Lumières des clusters	6	2
Définition des matériaux	4	1
Shaders des matériaux	6	2
Précalcul pour les sources polygonales	1	1
Shader pour les sources polygonales	6	3
Shader pour les sources indirectes ponctuelles	1	3
Direct + indirect	1	3

Ici, le GANTT mis à jour en fonction de modifications ci-dessus.



5 Analyse des risques

L'analyse des risques mise en avant ici est une évolution de l'analyse initiale présentée dans le rapport de spécifications. En particulier, le dépassement de délais s'est classé comme risque numéro 1 et disposera donc d'une analyse précise. Certains risques ont aussi été éludés, car trop insignifiants, ou trop généraux.

5.1 Dépassements de délais

Le dépassement de délai concerne tout débordement sur le GANTT initial. La probabilité d'occurrence est élevée, et l'impact, cumulé au bout d'un certain temps, est simplement critique. Pour l'instant, notre prévention de risques, qui se fait en prévoyant un temps de retard, fonctionne. Cette situation est à prendre au sérieux.

Notre première mesure est de réévaluer nos estimations de temps (voir 4). Elles ne sont pas lourdement changées (le plus notable est l'interface des clusters, qui passe de 10 jours pour 2 personnes à 8 jours pour 2 personnes). Nous avons aussi réorganisé notre planning pour ne laisser aucun "trou".

5.2 Autres risques

Globalement, les autres risques ainsi que les plans de secours associés n'ont pas grandement évolué. Il est donc possible de se référer au rapport des spécifications. Voici donc tout de même un rappel informel de ces risques.

Temps de rendu non conforme Dans la cas où les techniques que nous implémentons se révèlent peu efficaces ou si nous manquons d'optimisation, le temps de rendu peut être défaillant et entraîner des chutes de framerate occasionnelles ou constantes. Une revue des algorithmes à des fins d'optimisation ou des retraits de parties "lourdes" peut permettre d'amortir ce genre de problème.

Rendu de la scène non-conforme Si le rendu obtenu est, malgré notre étude des Méthodes et Algorithmes et nos discussions, non-conforme à ce qui était attendu, cela impliquera que les méthodes ne sont pas implémentées correctement. Nos seules options seraient alors de corriger les méthodes en place, ou d'annuler la fonctionnalité défaillante (ce qui signifierait un échec partiel du projet).

Modification soudaine du Radium engine Notre plug-in repose sur des structures déjà existantes du Radium, la suppression ou forte modification de celles-ci aura donc un impact fort. Sur le court terme, un suivi régulier des avancées du Radium est suffisant pour corriger les éventuelles incompatibilités qui apparaissent. *Un support sur le long terme du plug-in restera nécessaire (mais non inclus dans le cadre temporel du projet) pour pouvoir le maintenir compatible avec le Radium.*

Intégration sous forme de plug-in L'attente de notre projet est de former un plug-in au radium engine contenant des techniques physiquement réalistes. Une incapacité a produire les fonctionnalité dans une structure externe obligerait donc a inclure celle-ci dans la structure interne du moteur. Une discussion avec le client pour trouver une solution serait donc nécessaire si ce problème devait arriver.