Spécifications du PBRadium

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

3 janvier 2018

Table des matières

1 Cahier des charges **2**

1.1 Exigences fonctionnelles . . . 3

1.1.1 Renderer . . . 3

1.1.2 Éclairage . . . 3

1.1.3 Shader . . . 4

1.2 Exigences non fonctionnelles . . . 4

1.3 Contraintes . . . 5

1.4 Niveaux conceptuels . . . 5

2 Vue générale du système **6**

3 Modules et fonctionnalités **7**

3.1 Dépendances des modules . . . 7

3.2 Tests de validation des modules . . . 7

4 Planning prévisionnel et délais **8**

4.1 Tâches . . . 8

4.2 Planning prévisionnel (GANTT) . . . 9

4.3 Délais . . . 10

5 Analyse de risques **10**

Introduction

Suite à la demande expresse (et non planifiée) d'un nom de projet, nous suggérons de l'appeler le Radium/PBRadium, mais il pourra aussi être désigné par Radium-pbrtPlugin.

Il y sera présenté le projet de manière générale et le cahier des charges, un découpage en modules couplé à une analyse fonctionnelle répartie selon ces modules, et enfin un découpage en tâches cette fois, avec le planning associé.

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.*

1 Cahier des charges

Cette section présente le cahier des charges, qui liste les fonctionnalités attendues pour ce chef d'œuvre. Les charges (résumées dans le tableau 1), sont décrites dans les sous-sections ci-dessous.

Renderer 1.1.1	Éclairage 1.1.2		BSDF 1.1.3
	Direct	Indirect	
Rendu (draw)	Définition de polygones	Clustering	Définition desmatériaux Implémentation des BSDFs
	Fonction d'éclairage Stockage des sources Modification des sources		

FIGURE 1 – Tableau résumé des différentes charges, le numéro préfixé correspond au numéro de section ci-dessous

1.1 Exigences fonctionnelles

1.1.1 Renderer

Rendu ....

Objectif Permettre au Radium Engine de rendre en temps réel la scène courante avec les fonctionnalités du chef-d'œuvre, sous l'hypothèse que la scène (sans parler des sources de lumières) peut être rendue en temps réel initialement.

Description Le rendu est effectué à chaque `update()` du Radium Engine. Il dessine les objet de la scène en fonction de leurs matériaux respectifs et des lumières présentes.

Contraintes Cette fonction représente l'une des interfaces principales entre les autres fonctionnalités et le code existant, et doit donc être adaptée à ce dernier. Elle ne doit pas coûter trop cher en temps – ce qui implique une contrainte sur toutes les autres fonctions – de façon à avoir un affichage en temps réel.

Priorité Critique. Elle est la base de toutes les autres, qui ne serviraient à rien sans moteur pour afficher leurs résultats.

....

1.1.2 Éclairage

Fonction d'éclairage ....

Objectif Définir, pour une surface en un point, la contribution d'éclairage apporté par la source en question.

Description Cette fonctionnalité dépend du type de la source et de ses paramètres. Elle sera réalisée, pour chaque source, comme un shader. Le calcul de l'éclairage indirect (many-lights) est destiné, à terme, à n'être calculé que lorsque la scène est stable (e.g. lorsque la vue ne bouge pas).

Contraintes

Priorité Normale.

....

Stockage des sources ....

Objectif Stocker les sources.

Description Doit permettre de stocker efficacement les sources, suivant le type de sources.

Contraintes Les différentes implémentations doivent faire en sorte d'être – le plus simplement possible – réorganisable, dans l'optique optionnelle de maximiser la localité cache.

Priorité Normale.

....

Modification des sources *(direction privilégiée)* ....

Objectif Disposer des lumières dynamiques.

Description Consiste à laisser la possibilité de changer dynamiquement les caractéristiques des lumières (en terme de code) pour permettre à des lumières (ou du moins des groupes de lumières) d'être mobiles. Cet **objectif est optionnel**, dans le sens ou il n'est pas à proprement parler inclus dans le cahier des charges du client, cependant il est clair qu'une éventuelle évolution future pourra devoir en disposer. Autant prévoir son intégration.

Contraintes Doit pouvoir modifier les sources simplement, ce qui implique de prévoir la mise à jour rapide des structures qui en dépendent.

Priorité Faible.

....

Définition de polygones ....

Objectif Définir des sources de type polygonale pour l'éclairage direct.

Description Cette fonctionnalité assure la création de sources polygonales, paramétrées par un ensemble de points et des caractéristiques lumineuses.

Contraintes Les points doivent être coplanaires.

Priorité Normale.

....

Clustering ....

Objectif Permet de segmenter l'espace scène en fonction de la vue.

Description Consiste à répartir l'information des sources lumineuses indirectes au sein de clusters 3D (en espace caméra). Cela permettra de diminuer le nombre de lumières impactant chaque fragment (et donc le temps de calcul). Cette fonction créée une structure de données qui sera transmise au processeur graphique, lui permettant d'utiliser les lumières proches.

Contraintes Nécessite d'avoir la profondeur (Z-depth). Elle doit pouvoir être appelée à chaque frame, ce qui implique une grande réactivité.

Priorité Critique.

....

1.1.3 Shader

Définition des matériaux ....

Objectif Permettre de définir un matériau réaliste, par objet.

Description Définir un ensemble de matériaux réalistes et paramétrables. Chaque objet ne pourra avoir qu'un seul type de matériau pour le rendu. Chaque type de matériau est associé à un ensemble de paramètres différents, et définis pour l'objet ; à cette fin, il faudra implémenter une interface performante pour accéder au différents paramètres depuis les shaders.

Contraintes Les différents paramètres seront définis dans la scène, et ne seront pas voués à évoluer fréquemment durant le rendu.

Priorité Moyenne.

....

Implémentation des BSDFs ....

Objectif Permettre au moteur de représenter les différents matériaux

Description Implémenter les différentes fonctions modélisant la BSDF et pouvoir accéder à celle-ci en fonction du matériau de l'objet. Ces fonctions devront donc être liée avec la partie Rendu du Renderer.

Contraintes L'affichage de ces matériaux ne peut être fait que dans le Renderer et rester le plus modulaire possible afin de pouvoir être interchangé.

Priorité Moyenne.

....

1.2 Exigences non fonctionnelles

Performance Le système doit rester temps-réel, c'est à dire s'approcher de 60 fps idéalement, et de ne pas descendre de manière régulière sous les 30 fps. Dans cette optimique il faudras vérifier les goulots détranglement des ressources cpu, gpu, mémoire ou de bande-passante sur les transactions des données. Les différentes techniques seront donc paramétrables pour diminuer la complexité, mais impacteras alors la qualiter visuel.

Documentation utilisateur Court tutoriel sur l'utilisation du PBRadium.

Documentation développeur Le code doit être documenté en Doxygen.

1.3 Contraintes

Le projet est soumis aux contraintes suivantes :

— Il doit s'exécuter sur un système linux actuel [1].

— Il doit pouvoir compiler avec Clang ≥ 3.8 et GCC ≥ 6.3 [2].

— **Il doit pouvoir être réintegré relativement simplement sur la branche master du dépôt Radium Engine upstream.**

— Le Radium est déjà en C++ et doit donc être étendu en C++.

— Le Radium dispose également de conventions de code [3] qui seront utilisées pour le chef d'œuvre.

— Le programme doit être sous forme de plugin pour le Radium Engine [4].

1.4 Niveaux conceptuels

Plusieurs niveaux du programme sont impactées par ces exigences, en effet :

Niveau bas (OpenGL) Concerne les transferts de données vers la GPU tels que le passage des sources, les appels `draw()`.

Niveau moyen-bas (GLSL) Concerne les opérations faites sur la GPU par le truchement des shaders GLSL, autrement dit 1° les accès aux lumières (clusters ou non) et 2° les calculs des éclairages.

Niveau moyen-haut (C++) Architecture du plugin et interfaces.

1. Les systèmes testés sont Slackware, Archlinux, Ubuntu et Debian, supposées à jour et disposant des drivers standards.
2. Les versions sont étalonnées sur celles de Debian Stretch, supposée la plus "en retard" (la plus stable).
3. Le style LLVM Allman en discussion. Les conventions spécifiées sur `https://github.com/STORM-IRIT/Radium-Engine/blob/master/conventions.md` sont à priori obsolètes, se référer à l'issue n°178 pour plus d'informations `https://github.com/STORM-IRIT/Radium-Engine/issues/178`.
4. L'issue n°265 contient des informations importantes à cet égard `https://github.com/STORM-IRIT/Radium-Engine/issues/265`.

2 Vue générale du système

Nous présentons en figure 2 une vue plus générale du système. Hormis un certain nombre de classes utilitaires et non-métier, toutes les classes qui seront amenées à être créées sont sur ce schéma, qui contient également des classes existantes du Radium Engine. Le lien entre ces classes est également représenté.

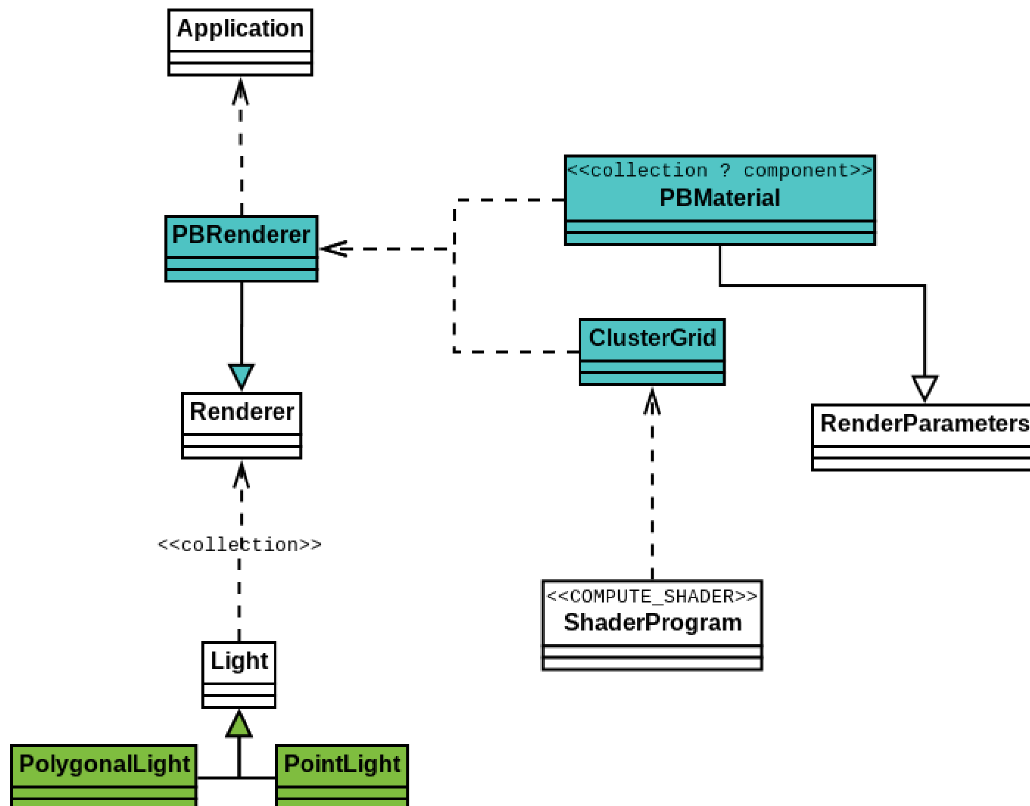


Figure 2 – Vue générale, les éléments blanc font partie de l'existant, tandis que les éléments bleus font parti du plugin PBRadium, en revanche les éléments verts auront leurs interfaces incluses au Radium-Engine et implémentées par chaque renderer souhaitant ces techniques de rendu.

La classe centrale ici est `PBRenderer`, qui sera la classe représentant le moteur en lui-même. Il hérite du `Renderer` de base du Radium Engine.

3 Modules et fonctionnalités

Le projet sera divisé en modules, subdivision déduite et mise en évidence par le cahier des charges. Cette approche nous donne au moins la subdivision suivante :

1. Renderer
2. Clustering
3. Matériaux BSDF
4. Éclairage (direct+indirect)

Nous avons choisi de séparé les différentes méthodes correctement du Renderer. Cela permet entre autres d'utiliser les matériaux et l'éclairage (indépendamment l'un de l'autre) sans utiliser le Renderer (en perdant les optimisations comme le clustering).

3.1 Dépendances des modules

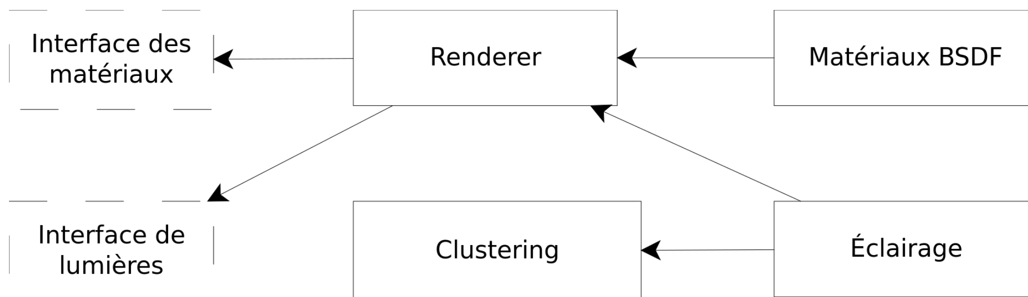


FIGURE 3 – Dépendances des modules. Les composants en pointillés sont les interfaces inclues (mais pas forcément existantes) dans le Radium.

3.2 Tests de validation des modules

Tester une application graphique 3D, surtout lorsque la configuration de la caméra dépend de l'utilisateur, est dépendante de trop de facteurs pour être bien efficaces. Nous établissons des "protocoles" de tests permettant de valider humainement (visuellement) que les différents modules fonctionnent.

Renderer Le simple constat d'un rendu fourni par le plugin sera suffisant, car il prouvera qu'on a bien accès aux caractéristiques de la scène depuis ce dernier.

Clustering En fixant une lumière à un endroit voulu à partir de la caméra, on peut vérifier en accédant au cluster que la source de lumière est bien rattachée au cluster attendu, et répeter cette procédure pour chaque cellule du cluster.

Matériaux BSDF La comparaison visuelle entre un rendu PBRT et un rendu PBRadium de la même scène (ou du moins du même objet, disons une sphère pour l'exemple) sera notre seule option. Un rendu PBRT étant bien plus précis et prenant en compte des paramètres plus étendus, il est peu fiable de faire une comparaison d'image automatique. La comparaison de scène dépend du chargement de ces scènes depuis les fichiers .pbrt, fonctionnalité dépendante d'un autre projet en cours, il risque donc de ne pas être disponible en partie ou intégralement *(voir les risques en section 5).*

Éclairage Le protocole sera similaire à celui des matériaux de type Disney (ci-dessus).

En plus de ces "protocoles" de tests, nous prévoyons des procédures de test de non régression rudimentaires, comparant des images du Radium que nous considérons correctes avec les images futures pour détecter d'éventuelles régressions.

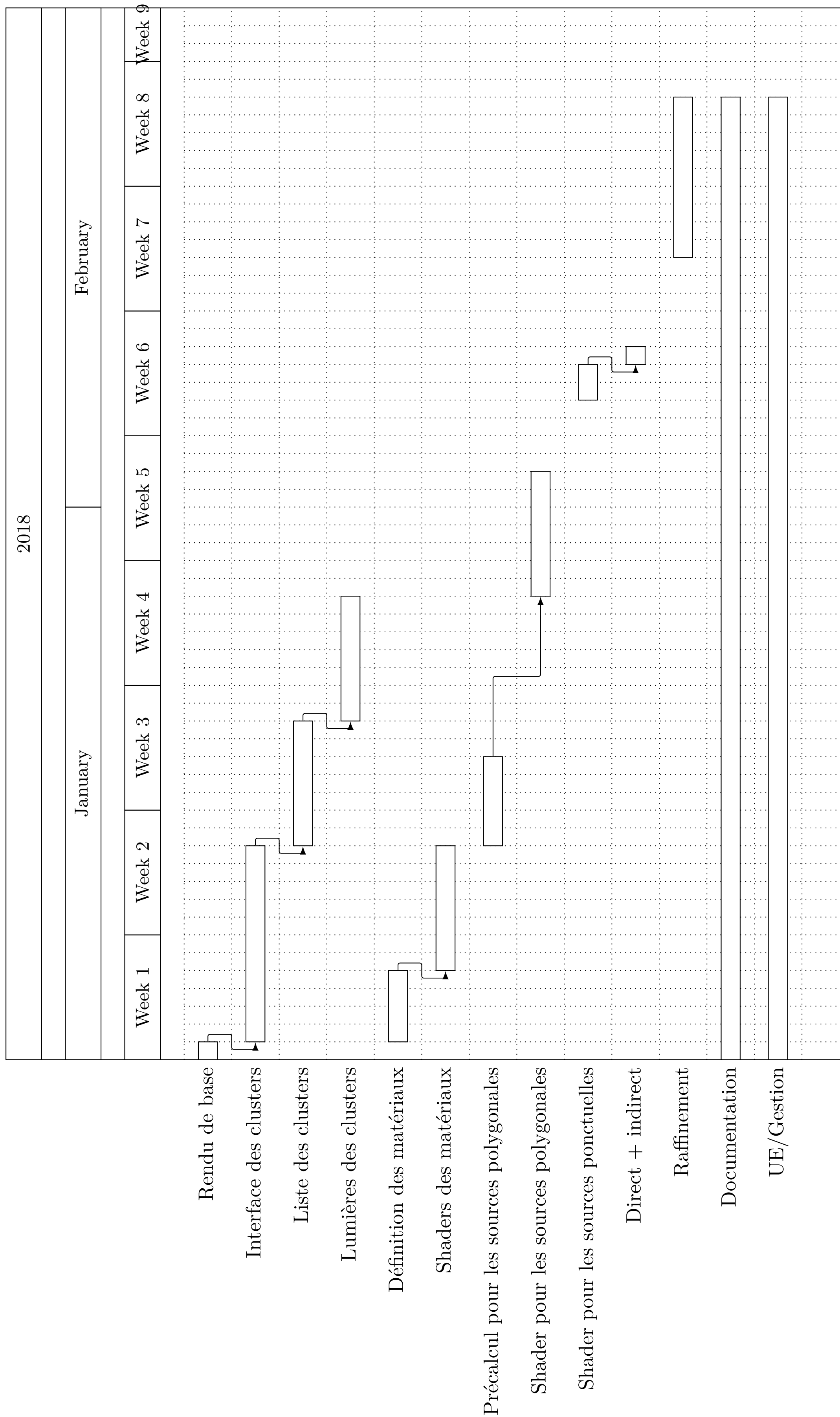
4 Planning prévisionnel et délais

4.1 Tâches

Module	Tâche	Description	Jours /Homme	Nombre de développeurs
Renderer	Rendu de base •	Mettre en place une base isolée de moteur de rendu à partir des composants du Radium. Cette base est prévue pour être en forward shading avec Z prepass.	1	1
Clustering	Interface des clusters ••••	Implémenter l'interface des clusters, c'est à dire avoir à dispositions tous les moyens pour utiliser ces clusters, le plus notable étant l'accès paramétré par des coordonées.	10	2
Clustering	Liste des clusters •••	Créer effectivement la liste des clusters associés à chaque fragment, et ce dans une structure creuse.	6	2
Clustersing	Lumières des clusters •••	Créer l'association entre un cluster et la liste des sources l'impactant, et définir ces dernières.	6	2
BSDF	Définition des matériaux ••	Définir les différents matériaux, organiser leurs paramètres.	4	1
BSDF	Shaders des matériaux •••	Écrire les parties de programme GLSL correspondant au calcul de l'effet des BSDFs.	6	2
Éclairage	Précalcul pour les sources polygonales •••	Créer la partie permettant de précalculer la table des valeurs permettant de simuler la distribution GGX ou intégrer les résultats existants du papier, fournis sous forme d'un header C.	4	1
Éclairage	Shader pour les sources polygonales ••••	Mettre en œuvre la partie effective de l'éclairage par sources polygonales, impliquant de trouver les vecteurs de direction de la source et de se referrer à la table précalculée.	6	3
Éclairage	Shader pour les sources indirectes ponctuelles ••	Gérer le calcul de l'éclairage par les sources indirectes une fois clusterisées.	1	3
Renderer	Direct + indirect ••	Créer la boucle permettant de calculer les 2 types d'éclairage et de les combiner.	1	3
	Raffinement ••	Revenir sur les choses douteuses.	∞	∞
	Documentation •	Créer la documentation de code/utilisateur.	2	3
	UE/Gestion ••	Assurer la gestion du projet, faire les rapports et revues.	8	3

FIGURE 4 – Tâches. Les • indiquent la difficulté estimée d'une tâche ($1 = facile$, $5 = difficile$).

4.2 Planning prévisionnel (GANTT)



4.3 Délais

Nous avons jugé utile de présenter ici les délais concernant les livrables.

- Le rapport sur les méthodes et algorithmes était dû pour le 7 novembre 2017.
- Le rapport ici-présent, sur les spécifications, est à rendre le 3 janvier 2018.
- Le rapport de conception est à rendre le 17 janvier 2018.
- La recette, qui s'accompagne d'un rapport personnel ainsi que du code, et d'une démonstration (ou vidéo), aura lieu le 23 février 2018.

5 Analyse de risques

Risques	Probabilité	Impact	Prévention	Correction
Indisponibilité d'un membre	Faible	Fort	Prioriser les fonctionnalités	Certaines fonctions ne seront pas développées
Mauvaise conception	Moyenne	Fort	utiliser des modèles connus et flexible	Éviter les modifications du Radium-Engine
Mauvais choix d'implémentation	Faible	Fort	Établir des testes durant la phase d'inception	Dialoguer avec les professeurs
Copyright	Faible	Faible	Discussion avec le client	Signer la cession des droits
Perte des données	Faible	Critique	Utiliser un dépôt privé	Utiliser des sauvegarde hors ligne
Dépassement des délais	Fort	Fort	Estimer le temps des différentes tâches	Faire des rushs
Clustering trop lent	Faible	Fort		Utiliser des clusters creux
Difficulté théorique	Moyenne	Faible	Préparation des articles	Discussion
Intégration en plugin	Faible	Faible	Suivi des évolutions du Radium	Faire une version non-plugin
Divergence entre l'API prévue et l'API effective	Forte	Faible		Faire un erratum

Il est à noter que le risque sur les choix d'implémentation ne concerne, pour nous, que les choix principaux, qui sont couverts par des exemples et l'étude et les discussions menées lors de la revue Méthodes et Algorithmes. Des problèmes plus locaux sont peut-être plus susceptibles d'apparaître mais sont la plupart du temps bien plus faciles à résoudre et ne sont donc pas comptés ici comme un risque.