Projet collectif de réalité virtuelle
Short Story

Rapport

Cao Hongyu
Ducceschi Rémi
Havard Thibault
Maugeais Mélanie
Noguer Thomas
Puret Mickaël
Vandenhove Pierre
Vernou Cedric

Introduction

Ce projet s'inscrit dans le cadre de l'option de réalité virtuelle de cinquième année. À la suite des cours et des travaux pratiques, il nous est demandé de réaliser un projet afin de montrer nos acquis dans un projet libre avec néanmoins quelques contraintes. Le projet a pour source une description simple afin de pouvoir se concentrer sur l'essentiel et ne pas surcharger la masse de travail. L'idée originelle sera développée dans la première partie de ce rapport avec les différentes contraintes imposées et les choix techniques. La deuxième partie présente le découpage des tâches ainsi qu'un diagramme de Gantt pour illustrer le déroulement du projet. Le reste du rapport est découpé en sous-parties dédiées chacune à une partie du développement expliquant le travail effectué, les problèmes rencontrés et les solutions apportées.

Cao Hongyu Havard Thibault Noguer Thomas Vandenhove Pierre
Ducceschi Rémi Maugeais Mélanie Puret Mickaël Vernou Cedric

Plan

Cao Hongyu Havard Thibault Noguer Thomas Vandenhove Pierre
Ducceschi Rémi Maugeais Mélanie Puret Mickaël Vernou Cedric

Cao Hongyu Havard Thibault Noguer Thomas Vandenhove Pierre
Ducceschi Rémi Maugeais Mélanie Puret Mickaël Vernou Cedric

Présentation du projet

Le projet initial tournait autour d'une expérience vidéo ludique jouant sur l'ambiance principalement. Le personnage principal devait se retrouver dans une ambiance sombre et étrange, la scène se trouvant dans un immeuble avec des pièces plus ou moins remplies. Un chat devait accentuer l'ambiance en suivant le personnage un peu partout dans l'immeuble. La scène finale devait prendre place au dernier étage de l'immeuble ou une scène automatique se déclenchait pour que le joueur puisse observer son avatar se défenestrer et chuter jusqu'au bas de l'immeuble.

Ce scénario permettait d'inclure tous les prérequis imposés par notre encadrant :

- Picking sur la porte ou la fenêtre destructible

Il nous faut avoir la possibilité d'interagir avec un objet de la scène. Il faut donc utiliser les lancers de rayons.

- Animation automatique avec le chat

  Il nous faut mettre en place un système d'animation automatique, il faut donc modéliser des animations si nécessaire puis les utiliser dans la scène.

- Animation manuelle avec le personnage

  Il nous faut mettre en place une animation manuelle déclenchée par l'utilisateur. Il faut donc gérer un système d'input pour pouvoir utiliser le périphérique clavier.

- Scène automatique avec la vidéo de fin

  Il nous est demandé de mettre en place une scène automatique. Cela veut donc dire utiliser des courbes pour déterminer la trajectoire d'objets et de la caméra.

- Particules avec l'explosion de la fenêtre

  Il nous est demandé de mettre en place un système de particules.

- Niveau de détail avec les objets en dehors de l'immeuble lorsque l'on chute

  Il nous est demandé de mettre en place du niveau de détail afin de simplifier un mesh.

- Scène manuelle

  Il nous est demandé de mettre en place une scène manuelle. Cela requiert alors une gestion des déplacements, des collisions et de la caméra.

Déroulement du projet

Découpage des tâches

Le découpage des tâches s'est initialement fait ainsi :

Pierre Vandenhove
Modélisation de l'immeuble

Thibault Havard
Modélisation du personnage
Animation et intégration du chat

Mickael Puret
Mise en place des particules
Mise en place du terrain

Mélanie Maugeais
Mise en place du son et des lumières

Hongyu Cao
Mise en place de la scène automatique

Cédric Vernou
Mise en place du niveau de détail
Gestion de la caméra
Gestion du picking

Thomas Noguer
Intégration des meshs dans Ogre3D
Lancer de rayons pour la gestion des collisions
Coordination des tâches et personnes

Rémi Ducceschi
Gestion des collisions
Mise en place de l'architecture globale

Cette liste est évidemment présente comme indication et n'est pas représentative de la réalité. En effet, il est régulièrement arrivé qu'une personne intervienne dans une tâche qui ne lui était pas attribuée

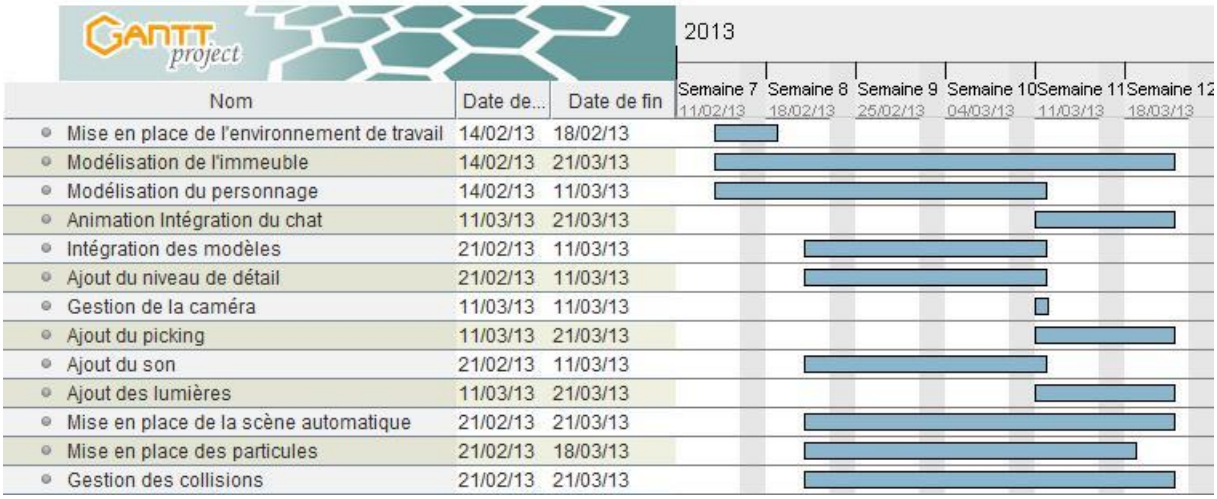
Contraintes de développement

Nous avons choisi d'utiliser un système de versionnèrent de fichiers afin de rendre la lourde de tâche de travailler à 8 sur un même projet possible. Notre choix s'est tourné vers git par affinité. Nous avons choisi GitHub comme dépôt puisque déposer un projet sur GitHub permet d'en distribuer les sources par ailleurs si une personne est intéressée par le projet.

Notre choix de moteur 3D s'est tourné vers Ogre3D puisqu'il était prévu d'avoir des TPs sur ce moteur en particulier. De plus, le développement sous Ogre3D est en C++ qui est un langage très répandu et très puissant. Une communauté est également très active ce qui permet la présence de nombreux tutoriels et documents pour aider au développement *via* ce moteur.

De plus, nous avons choisi de travailler avec la dernière version d'Ogre3D afin de disposer de toutes les dernières fonctionnalités et de ne pas être confrontés à des problèmes dépréciés. Ce choix impose d'utiliser la dernière version du compilateur gcc.

Déroulement des tâches



Nom	Date de...	Date de fin
Mise en place de l'environnement de travail	14/02/13	18/02/13
Modélisation de l'immeuble	14/02/13	21/03/13
Modélisation du personnage	14/02/13	11/03/13
Animation Intégration du chat	11/03/13	21/03/13
Intégration des modèles	21/02/13	11/03/13
Ajout du niveau de détail	21/02/13	11/03/13
Gestion de la caméra	11/03/13	11/03/13
Ajout du picking	11/03/13	21/03/13
Ajout du son	21/02/13	11/03/13
Ajout des lumières	11/03/13	21/03/13
Mise en place de la scène automatique	21/02/13	21/03/13
Mise en place des particules	21/02/13	18/03/13
Gestion des collisions	21/02/13	21/03/13

La configuration de l'environnement de développement a pris plus de temps que prévu pour plusieurs raisons.

La première étant de configurer le projet avec toutes les librairies et les dossiers d'include nécessaires. Cela n'est pas particulièrement compliqué, mais si mal fait résulte en de nombreux problèmes.

Deuxièmement, la dernière version de gcc est un prérequis lors de l'utilisation de la dernière version d'Ogre3D, ce point lorsqu'il n'est pas pris en compte, pose problème lors de la phase de configuration du projet. Effectivement, on pense avoir fait une erreur dans la configuration de l'IDE alors que le problème vient de la version du compilateur.

Afin d'intégrer le moteur physique Bullet à notre projet, il est indispensable d'utiliser OgreBullet qui permet de faire l'interfaçage entre le moteur physique et le moteur 3D. Cependant,

OgreBullet a besoin d'être recompilé avant d'être intégré dans le projet. Il faut ainsi recompiler directement la librairie avec CMake. Cette solution a été pénalisante dans le début du projet et a ainsi été abandonnée rapidement afin de se limiter à une gestion des collisions avec de simples lancers de rayons.

La modélisation s'est avérée plus compliquée que prévu. La charge de travail de la modélisation de l'immeuble a été mal évaluée et mal gérée. Elle a été bloquante pour de nombreuses tâches, notamment la gestion des collisions, ce qui a fortement ralenti la progression initiale du projet.

Modélisation et importation dans la scène

Présentation de Blender

Blender est un logiciel libre, sous licence GNU GPL, de modélisation, d'animation et de rendu 3D. Il est extrêmement complet et reconnu dans le monde entier grâce à de nombreux projets basé uniquement dessus (http://mango.blender.org/production/tears-of-steel-download-watch/, http://www.sintel.org/ .. .)

La prise en main de Blender est délicate dû à ses nombreuses fonctionnalités et à son interface basée sur les nombreux raccourcis clavier. Afin de pallier cette difficulté, il existe une communauté énorme et de très nombreux tutoriaux sont disponibles sur Internet (Site du Zero, Blender Clan…).

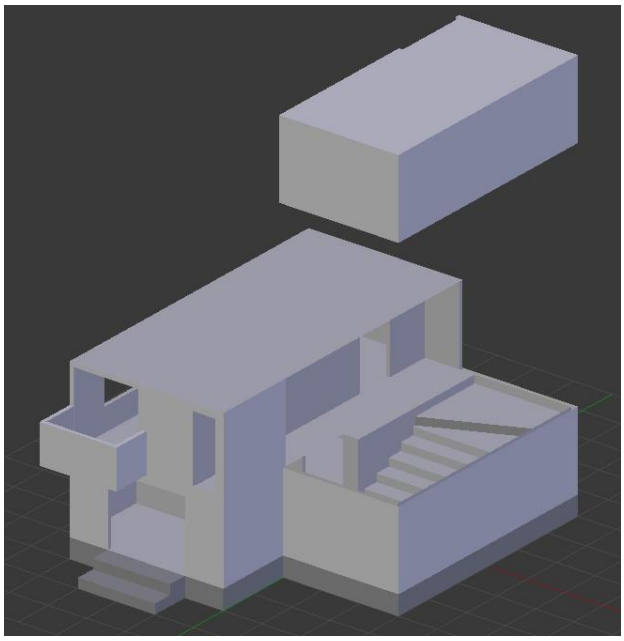
Lors de notre première réunion, il a été décidé que toute la partie modélisation (immeuble, voiture et chat) et certaines animations (celles du chat) seraient faites grâce à ce logiciel.

Immeuble

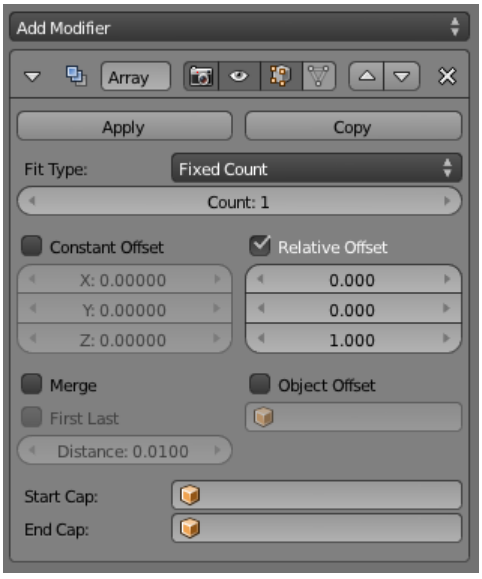
Modélisation

Pour modélise un immeuble nous avons commencé par chercher un modèle tout fait sur internet (http://www.blenderguru.com/videos/high-rise-building/). Le modèle présent sur ce lien est complet, bien trop, et il possède un trop grand nombre de faces pour être intéressant pour notre projet. Nous avons donc décidé de suivre le tutoriel explicatif pour pouvoir adapter un maximum leur immeuble à notre projet.

Cela s'est déroulé correctement pendant la première partie du projet, mais un nouveau problème est apparu : l'immeuble une fois intégré à notre scène paraissait « plat », pas assez élevé. Cela ne cadrait pas avec l'histoire que nous voulions mettre en place. Il a donc été décidé de repartir d'un plan papier et de créer un immeuble de toutes pièces, grâce aux techniques apprises lors du tutoriel.



Sur cette illustration, on peut voir la modélisation des parties fixes de l'immeuble. En effet, pour éviter une perte de temps énorme, il n'est pas nécessaire de modéliser tous les étages, par exemple. Il suffit d'en modéliser un et d'appliquer au mesh obtenu un modificateur *Array* qui permet de dupliquer le mesh selon un Offset relatif (ici nous choisissons 1 unité selon Z pour placer les étages les uns au-dessus des autres). Pour le peu de temps que nous avions, nous avons préféré faire des étages tous identiques plutôt que de différencier chacun d'eux.



La modélisation des escaliers implique exactement les mêmes techniques.

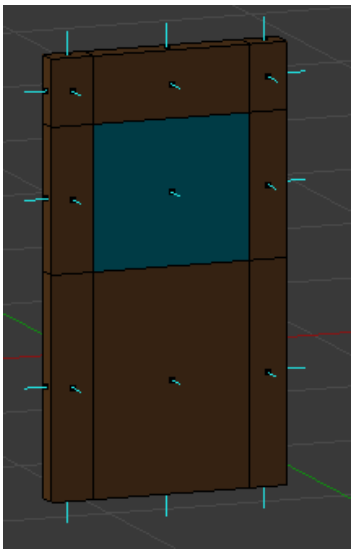
Une fois la base de l'immeuble créée, il a fallu placer des éléments qui vont nous servir de vitres cela se fait tout simplement grâce à des mesh rectangulaire placés dans les espaces prévus à cette effet. Cela donne :

Un matériel spécial est créé pour faire les vitres transparentes.

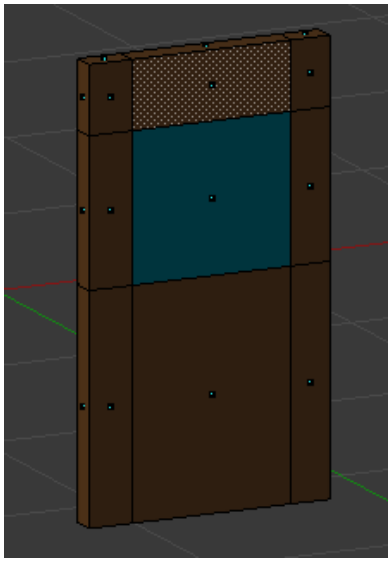
Une fois toutes ces étapes validées par l'équipe, il suffit de choisir le nombre d'étages (i.e. le nombre d'*Array* à mettre) et on obtient :

Problèmes rencontrés

Le principal problème rencontré a été une disparition des faces lors de l'exportation vers Ogre. Cela était dû à une mauvaise orientation des normales des faces dans Blender.



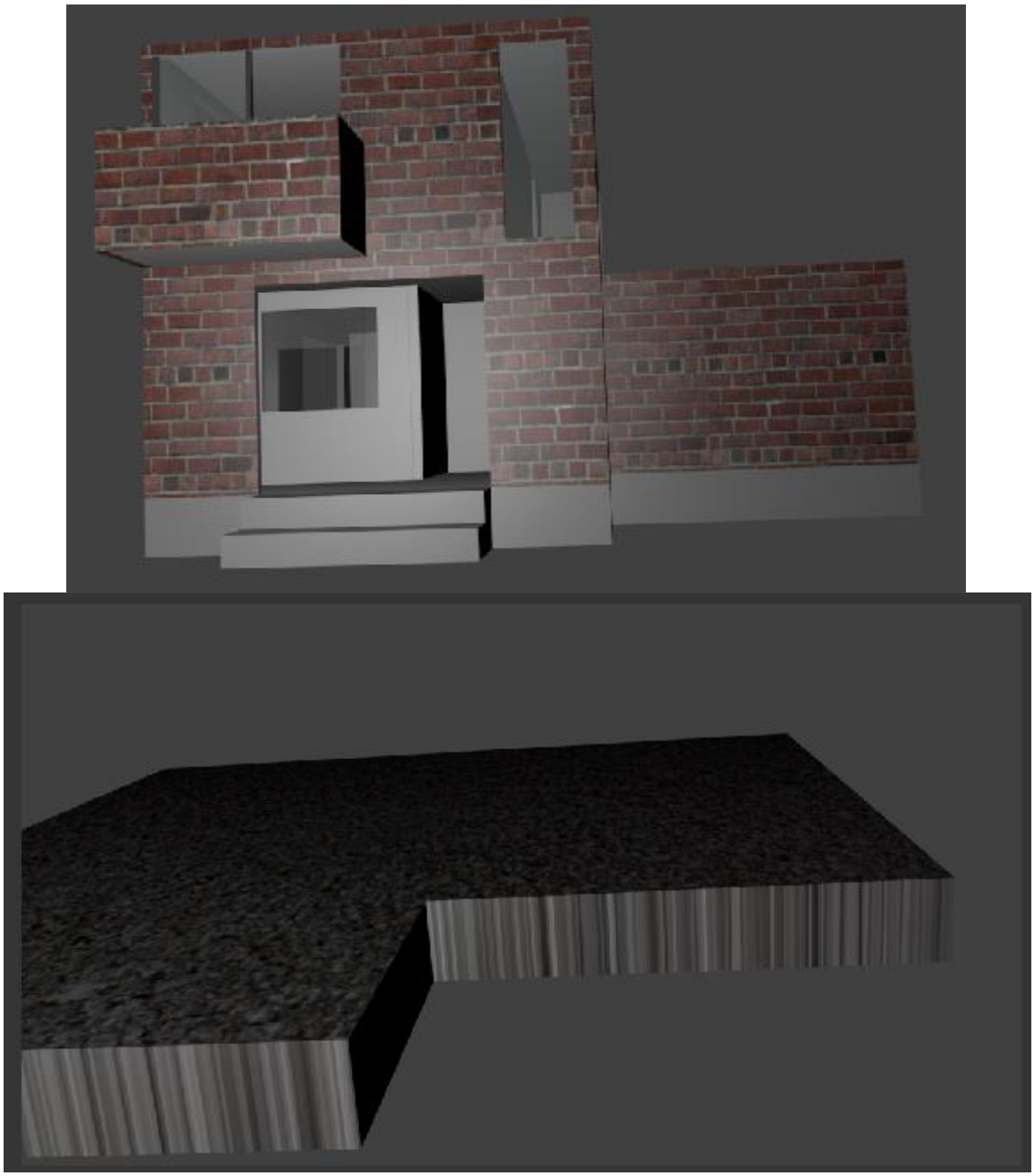
Sur cette image les normales ont toutes orientées vers l'extérieur donc l'exportation se passera bien et la porte sera visible une fois affiché dans Ogre.



Sur cette image les normales sont toutes orientées vers l'extérieur donc l'exportation se passera bien et la porte sera visible une fois affiché dans Ogre.

Ici en revanche les normales sont situées à l'intérieur des faces (cf les points bleus qui indique les normales de chaque face) donc sous Ogre les faces ne seront pas affichées correctement.

Le deuxième problème a été de texturer un minimum l'immeuble pour éviter d'avoir un ensemble tout gris tout moche. Le problème n'est pas réglé, car la texture n'a fonctionné que sur un seul matériel. Les textures sur les murs intérieurs sont très compliquées à mettre en place, car il faudrait prendre du temps pour placer correctement chaque texture sur le mur.

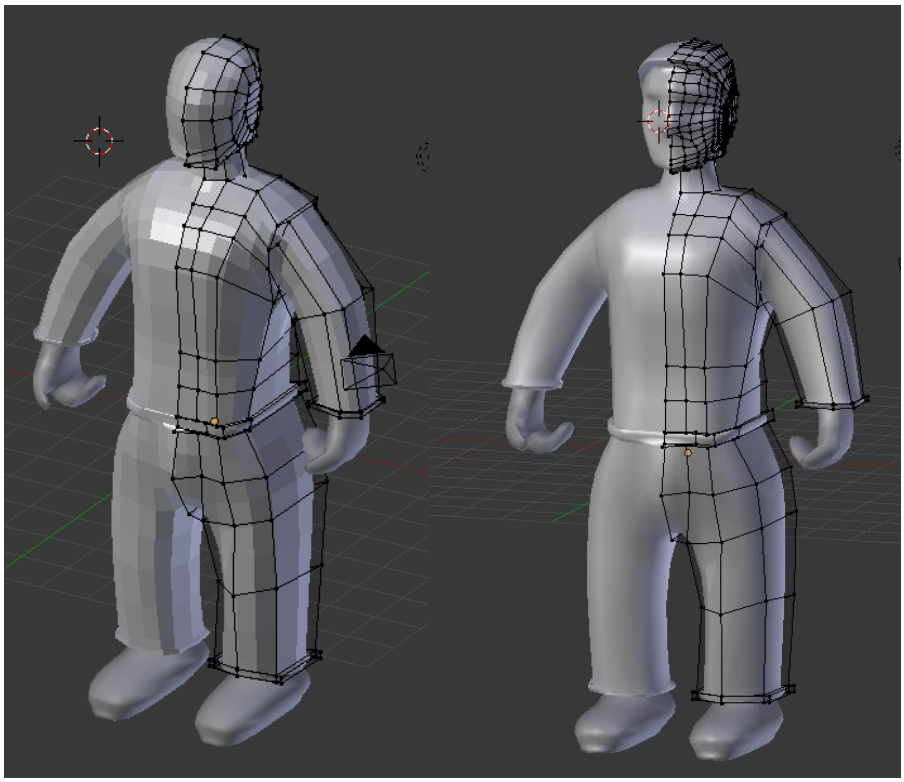


Personnage

Modélisation du personnage

Nous avons créé un personnage, celui-ci n'a pas été gardé par la suite, car l'implémentation du personnage du ninja était déjà terminée et plus complète que notre modèle. De plus, il fallait mettre en place l'animation du chat.

Cependant, ce travail m'a permis de voir comment créer un personnage. Pour cela, on formera dans un premier temps la structure de notre personnage de manière grossière :



Puis on ajoutera au fur et à mesure des séparations afin d'augmenter le niveau de détails (notamment pour la tête du personnage). On fera le tout en appliquant des modificateurs dans Blender, celui de la symétrie pour avoir à faire seulement une partie du corps et le modificateur *sub-division surface* afin que notre modèle soit plus lisse en ajoutant un nombre de surfaces supplémentaires (tout en gardant un niveau bas pour éviter d'avoir trop de face). Le corps et la tête du personnage sont en un seul morceau et sont fabriqués grâce à plusieurs extrusions. Seuls les bras et les pieds sont séparés, on les ajoutera par la suite.

Cette partie de la modélisation est très longue, il faut ajouter les parties petit à petit tout en essayant de garder un aspect au personnage.

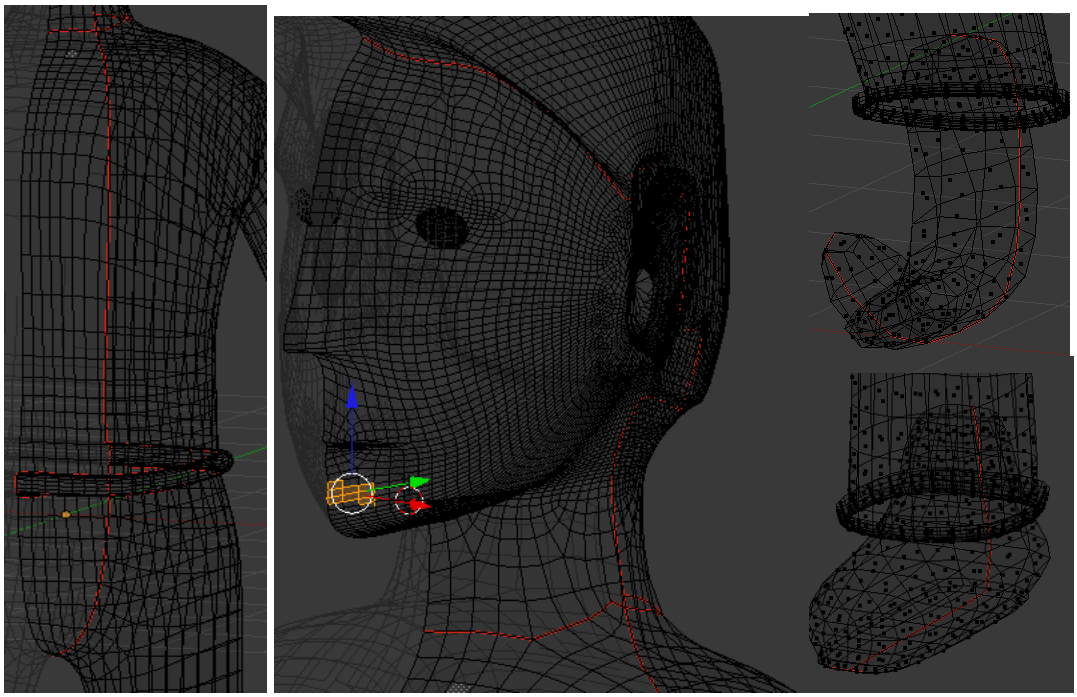
Une fois que notre personnage possède des formes de base, on appliquera le modificateur de subdivision afin d'obtenir les vertex réels de notre personnage. On pourra alors passer en mode sculpture, ce mode permet de pouvoir sculpter notre objet avec différents outils pour lui donner la

forme que l'on souhaite, cet outil est intéressant, mais nécessite beaucoup de face pour faire du travail précis ce qui n'est pas bénéfique pour le rendu dans ogre. Cependant, on pourra s'en servir afin de marquer le nez, la bouche, les yeux.

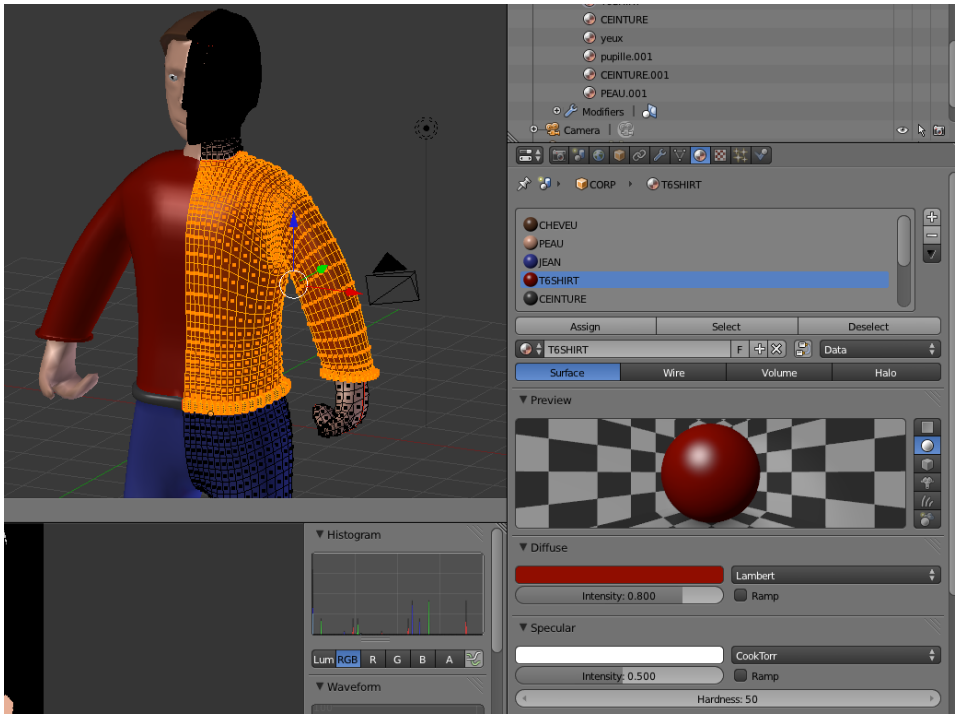
Pour la création des yeux la solution la plus simple est de créer une sphère à laquelle on peut enlève la moitié. Pour mettre en place les textures, il va falloir maintenant ajouter nos mains, les pieds et les yeux en joignant le tout avec *ctrl+j*. A ce stade, nous avons plus qu'un seul *mesh* pour notre personnage.

Texture du personnage

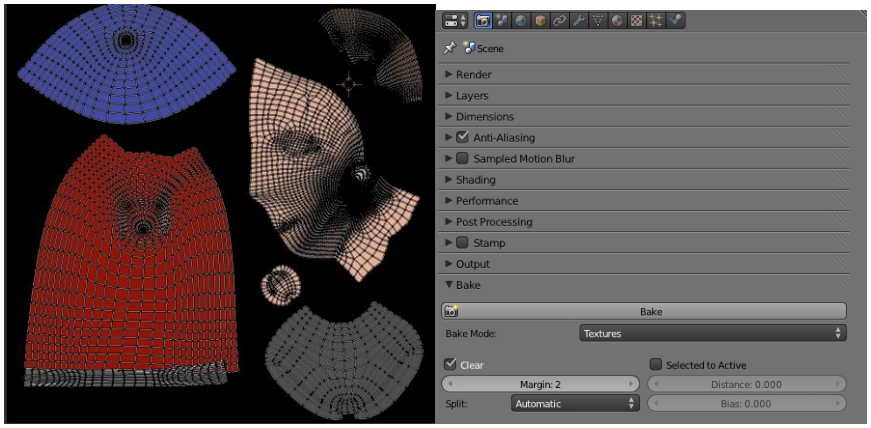
Pour la mise en place de la texture, on va définir des zones de découpages, celle-ci sert lors du dépliage de notre texture sur ce que l'on appelle l'*UV map*. Pour cela, on se met en *edit mode* et on choisira les vertex de séparation. Dans notre cas, ces vertex se situent au niveau du cou (séparation tête et T-shirt), la chevelure, la ceinture et le long du dos, du cou, des mains et des pieds pour séparer la texture.



Une fois les séparations effectuées, on peut lui appliquer des matériaux. Cela simplifiera la mise en place de la texture en attribuant une première couleur aux différentes parties. Pour cela, on se met en I et avec le mode de sélection des faces puis on va appuyer sur la touche L afin de sélectionner les différentes parties prédécoupées. Enfin, en allant dans le menu des matériaux on ajoute le matériau ou on en choisit un existant à appliquer à notre modèle.



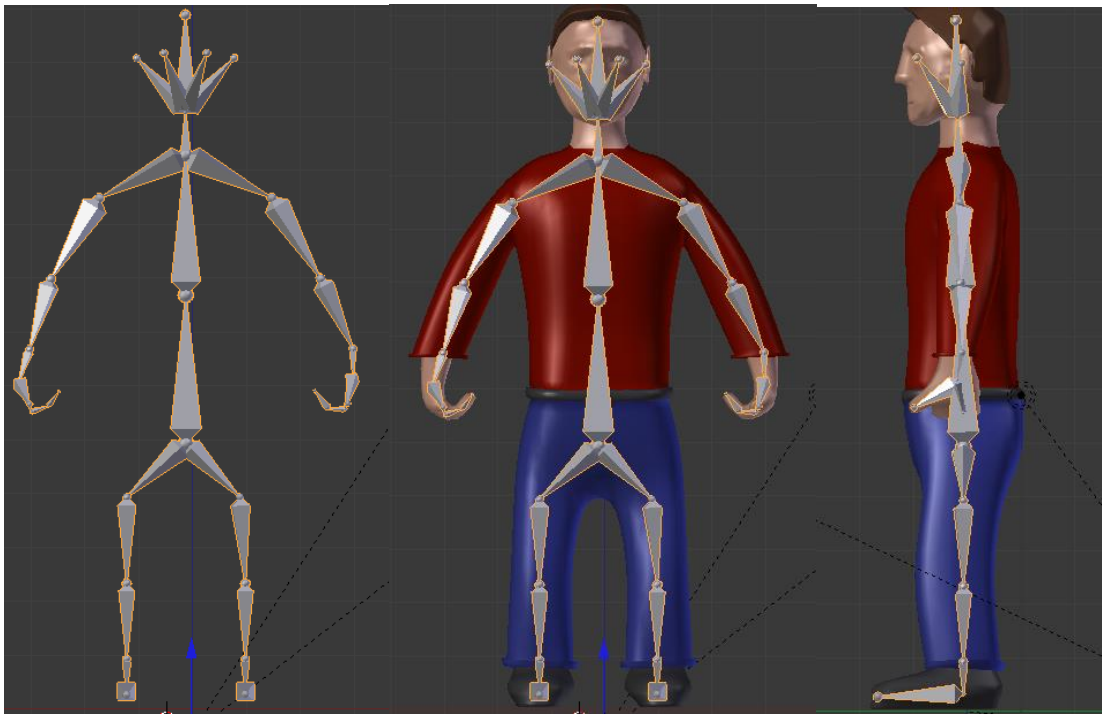
Ensuite, il faudra déplier et appliquer nos matériaux sur la texture. Pour cela, on appuie sur *U* et on choisit *unwrap*. Cela va afficher dans l'*UV editor* nos formes dépliées, il faudra les replacer correctement. Par la suite, on applique les matériaux définis auparavant en cliquant sur *bake* avec l'option texture afin d'appliquer les couleurs des matériaux sur la texture :



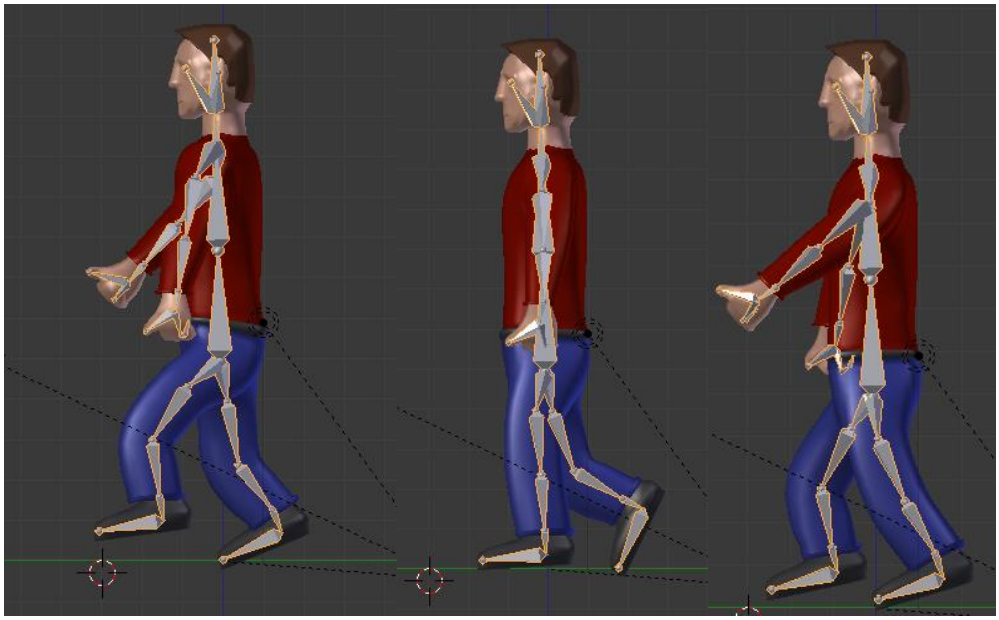
On pourra alors par la suite redessiner sur la texture comme on le désire.

Animations du personnage

Pour mettre en place les animations, il faut définir un squelette. Pour cela, on va placer dans notre modèle un os en bas du dos pour au travers de différentes extrusions agrandir notre squelette sur chaque extrémité. On obtient alors un squelette qui correspond à la forme de notre personnage :



Une fois notre squelette correctement positionné on peut après avoir fait *ctrl+p* et choisi le mode de poids automatique passé en mode *pose*. Dans ce mode on peut faire bouger les articulations de notre modèle et grâce à la touche *i* marquer la positionner actuel comme position clé *KeyFrame*. On peut alors définir notre animation en déplacent les os sur plusieurs frames.



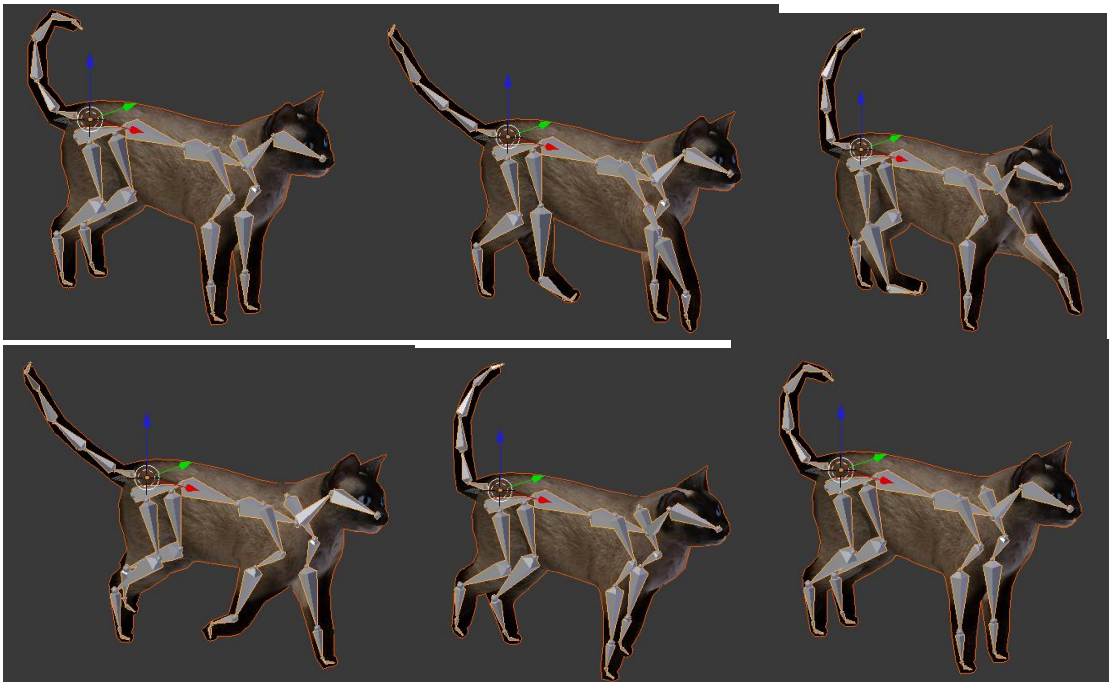
Cao Hongyu Havard Thibault Noguer Thomas Vandenhove Pierre
Ducceschi Rémi Maugeais Mélanie Puret Mickaël Vernou Cedric

Chat

Animation du chat

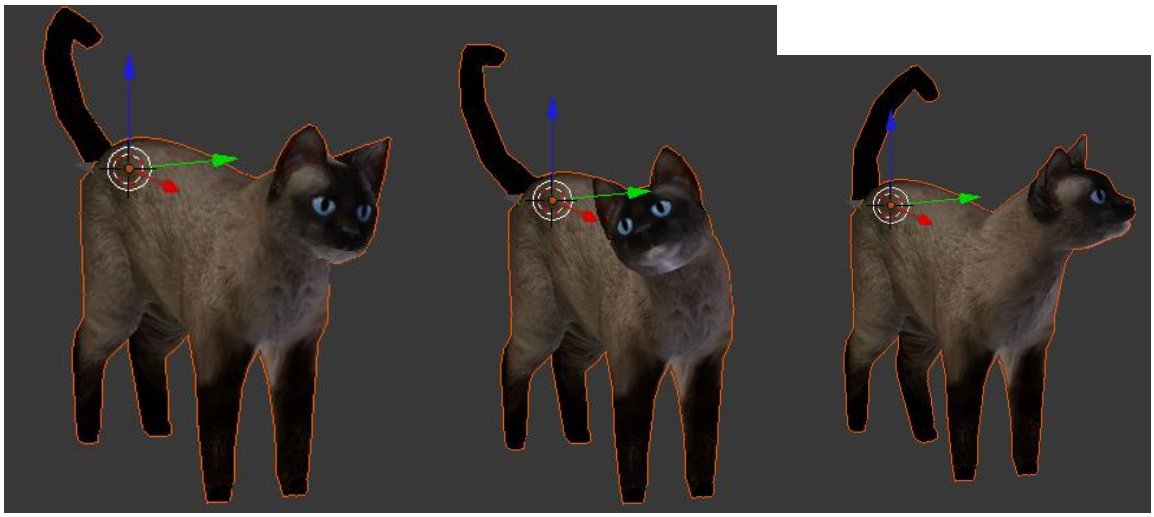
Nous possédions un modèle de chat, celui-ci avait déjà le squelette et les textures de définie. Il ne rester plus qu'à mettre en place des animations. De la même manière que pour le personnage on en pose mode définir nos *keyframes* afin de définir notre animation. J'ai mis en place 4 animations : attendre, marcher, assoir, debout. Dans le programme seulement 2 seront mis en place.

Pour l'animation de marche, il y a une astuce, un chat quand il se déplace bouge de manière coordonnée la patte avant gauche avec l'arrière droit et inversement. Du coup, on définira des frames avec les extrêmes et les positions intermédiaires (patte alignée) c'est-à-dire la patte le plus avancée possible – position intermédiaire – patte le plus reculé possible – position intermédiaire :

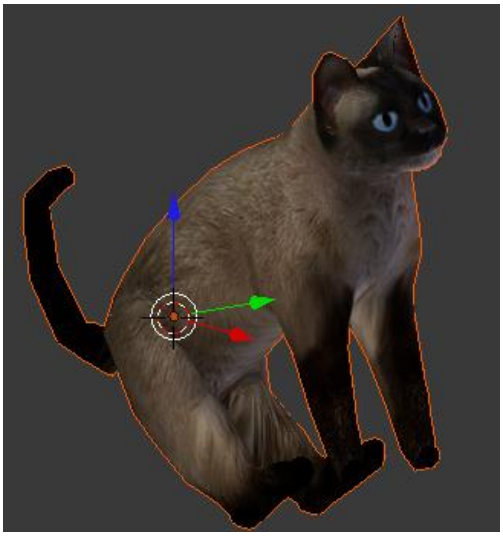


Une fois les *Keyframe* mis en place, on aura plus qu'à ajuster la longueur de l'animation en déplaçant comme désiré les frames. De plus, pour une animation plus réaliste il est intéressant de multiplier la longueur de l'animation en rajoutant un cycle de marche supplémentaire. Enfin, pour rendre le tout encore plus réaliste on animera la tête et la queue du chat.

L'attente du chat consiste juste à bouger la tête et la queue, il suffit donc de quelque frame avec la tête et la queue dans différentes positions.



De même pour s'assoir et se lever, l'animation est la même dans le sens inverse, il suffit alors de faire assoir le chat sur ses pattes arrière :



Enfin, il faut exporter le tout en format ogre en vérifiant (très important !) que le *mesh* et le squelette de notre animation ont bien le même point d'origine.

Implémentation du chat dans ogre

Pour implémenter notre chat dans ogre, on l'ajoutera dans la scène. On créé donc une fonction *setChat()* qui placera le chat dans un nœud du graphe de scène, on vérifiera les proportions de notre chat et son emplacement.

On ira ajouter dans la classe des animations, les animations correspondant aux 4 animations définies auparavant avec les fonctions suivantes : *walkCat(), waitCat (), sitCat , getupCat().*

Chacune de ces fonctions appelant l'animation correspondante exemple :

```
Entity* chat = mShortStory->getSceneManager()->getEntity("Chat");
chat->getAnimationState("marche")->setEnabled(true);
chat->getAnimationState("marche")->addTime(evt.timeSinceLastFrame);
```

Enfin, il faut pouvoir faire en sorte que le chat suive le personnage du regard et qu'il le suive. Pour cela, dans la fonction d'animation *InputListener ::frameRenderingQueued* on appliquera les différents mouvements ainsi que l'appel aux fonctions déterminant l'animation à exécuter.

Pour le déplacement, il suffit de calculer le vecteur directeur dans le plan X,Z pour cela on effectue un simple calcul *vecteur_déplacement = personnage_position – chat_position* puis on normalise ce vecteur il suffit alors de translater le *mesh* dans cette direction en multiplier par un facteur de vitesse. De plus, on vérifiera que le *mesh* garde une certaine distance pour ne pas coller le personnage. Pour ajouter les animations, il suffira d'appeler notre fonction d'animation correspondante à l'état du chat selon la condition de distance.

```
if((fabs(pers_pos.x - chat_pos.x) > 200)||(fabs(pers_pos.z- chat_pos.z) > 200)){
   mScene->getCatNode()->translate(move_cat*4);
   mAnimation->walkCat(evt) ;
} else{
   mAnimation->waitCat(evt);
}
```

Pour finir, il faut que le chat regarde vers le personnage, car il se déplace vers lui. Pour cela il faut récupérer l'orientation du *mesh* selon l'axe X puis de définir le quaternion correspondant à l'angle de rotation pour que le chat reste en face du personnage et enfin appliquer cette rotation à notre *mesh*.

```
Ogre ::Vector3 orientation = mScene->getCatNode()->getOrientation() *
Ogre ::Vector3::UNIT_X;
Ogre ::Quaternion quat = orientation.getRotationTo(move_cat);
mScene->getCatNode()->rotate(quat);
```

Pour conclure, la modélisation et l'animation prennent beaucoup de temps, notamment lorsque les modèles sont complexes. De plus, ce personnage étant mon premier il a fallu du temps pour obtenir une forme à peu près réelle et surtout la mise en place de la texture qui est loin d'être simple.

Gestion des collisions

La gestion de collisions a été un problème majeur dans notre programme. C'est en effet un des points essentiels de notre jeu en 3D, puisqu'elle permet d'éviter au personnage de traverser les murs et de garder les pieds sur terre.

Le moteur de collisions Bullet

Il existe un moteur libre gérant la physique appelé Bullet. C'est un moteur très puissant dont de nombreux *wrapper* existent, autant pour des moteurs écrits en C++ comme Ogre que pour des moteurs Java comme Jmonkey.

Une manière simple et efficace d'intégrer Bullet dans un projet Ogre est d'utiliser le plug-in pour Ogre : OgreBullet.

Son utilisation aurait permis de gérer les collisions de manière quasi automatique, nous aurions ainsi pu nous concentrer davantage sur l'aspect de notre jeu plutôt que sur ce problème relativement complexe.

Malheureusement, ni OgreBullet, ni Bullet ne sont distribués avec les binaires d'Ogre. Les binaires pour le moteur Bullet peuvent être trouvés facilement, mais nous ne savions pas avec quelle compilateur et quelle version ils ont été compilés. Les binaires d'OgreBullet n'existent pas pour Windows (ou alors pour de vieilles versions).

L'installation d'OgreBullet nécessitait donc la compilation manuelle de Bullet, puis d'OgreBullet afin d'être certain d'avoir une complète compatibilité avec les binaires d'Ogre. Ayant déjà perdu beaucoup de temps à l'installation du moteur Ogre et d'un compilateur correctement exploitable, nous avons jugé que l'installation d'OgreBullet était trop laborieuse. Cela nous aurait fait perdre plus de temps que de trouver une parade simpliste au problème de la gestion de collisions.

De plus, travaillant à huit sur le projet, avec au moins quatre systèmes différents (Linux, Windows, en version 32 ou 64 bits), la compilation de Bullet et OgreBullet aurait dû être faite par chacun des collaborateurs, ce qui aurait été une perte de temps inacceptable.

Le lancer de rayons

Étant arrivés à la conclusion que nous ne pourrions pas nous servir d'un moteur physique pour notre programme, nous avons décidé de gérer le problème des collisions manuellement.

Pour cela, d'après notre cours de Réalité Virtuelle, deux solutions s'offraient à nous :

- le lancer de rayons ;
- l'utilisation de boîtes englobantes alignées sur les axes.

La deuxième solution est bien trop complexe à mettre en œuvre en quelques semaines : cela serait revenu à recoder un vrai moteur physique. Nous nous sommes donc penchés sur le lancer de rayons, et avons fait face à plusieurs problèmes. Heureusement, Ogre permet de lancer des rayons de manière intuitive et relativement aisée.

Collision avec les volumes englobants

Par défaut, le lancer de rayon dans Ogre est très primitif : il ne travaille qu'avec les volumes englobants des objets. Lors d'un lancer de rayon, on ne peut que récupérer un booléen qui nous dit si oui ou non le rayon a touché un volume englobant (donc potentiellement touché un objet), ainsi que la liste des objets potentiellement touchés.

Si cela peut suffire pour du picking primaire, il nous fallait un système bien plus avancé pour gérer les collisions de notre personnage contre les murs et les objets de l'immeuble.

Atteindre les objets et obtenir une distance

Le but de notre lancer de rayon est de définir une distance minimale entre notre personnage et les objets qui l'entourent, afin d'éviter qu'il ne rentre dedans. L'idée était donc de lancer un rayon dans la direction souhaitée et de vérifier que le premier objet touché était à une distance supérieure à la taille du personnage.

Il fallait donc pouvoir récupérer le premier objet touché, ainsi que la distance par rapport à cet objet. Pour cela, nous nous sommes servis de la liste des objets potentiellement touchés retournée par Ogre lors d'un lancer de rayon. Nous parcourons cette liste en extrayant tous les triangles de chaque objet potentiellement touché.

Pour chaque triangle de chaque objet, nous essayons de voir si le rayon intersecte le triangle ou non. Si oui, à quelle distance.

À la fin de la méthode, nous sommes capables de déterminer quel triangle de quel objet est le premier à avoir rencontré le rayon, et à quelle distance il se trouve. Le désavantage de cette méthode est que si nous n'avons pas de chance, nous sommes obligés de parcourir tous les triangles de tous les objets, ce qui peut s'avérer assez long.

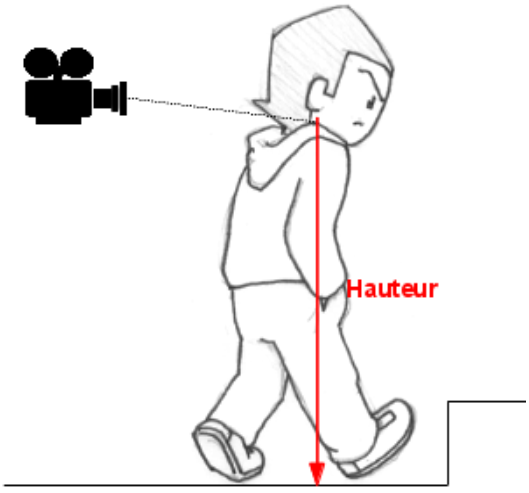
Plusieurs rayons

Dans la version actuelle, nous lançons cinq rayons lors du déplacement du personnage : un pour déterminer les collisions avec le sol (hauteur du personnage), et quatre pour les collisions avec les murs.

Gestion de la hauteur

La gestion de la hauteur permet au personnage de garder les pieds sur terre, sans se mettre à flotter au-dessus du sol... Il doit ainsi être capable de monter les escaliers de l'immeuble correctement.

Cela se fait particulièrement facilement à l'aide d'un simple lancer de rayon. Nous avons placé l'origine du personnage au niveau de ses épaules, la caméra en mode troisième personne est pointée sur ce point particulier. C'est aussi de là que nous lançons les neuf rayons de détection de collisions.



Nous lançons un rayon à partir de ce point, verticalement vers le bas, et imposons le premier objet touché par le rayon soit à une distance fixe. Si la distance est plus forte, nous baissons le personnage de la différence, si elle est plus faible, nous l'élevons.

Gestion des murs

La gestion des murs est beaucoup plus délicate puisqu'elle nécessite de détecter les objets aux alentours qui ne sont pas à une distance fixe, contrairement au sol. De même, alors que le sol ne peut se trouver uniquement sous le bonhomme, les murs peuvent apparaître n'importe où sur le côté. Il est donc nécessaire de lancer plusieurs rayons pour les détecter dans toutes les directions.

Nous lançons 4 rayons (devant, derrière, à gauche et à droite). Pour chaque rayon, on regarde l'objet le plus proche touché. Si cet objet est inférieur à une distance donnée, on interdit au personnage d'avancer dans cette direction. Autrement, le déplacement est libre.

Effet de rebond

Le traitement des collisions est fait dans la fonction *frameRenderingQueued()* d'Ogre, donc à chaque nouvelle image calculée. Cependant, selon la vitesse de déplacement du joueur et la puissance de calcul de l'ordinateur, le personnage peut avoir un déplacement important entre deux images. Il pourrait potentiellement entrer en collision avec un objet durant ce déplacement.

C'est pourquoi nous utilisons un système de rebond, qui fait rebondir le personnage contre le mur afin de le déplacer à la distance minimale autorisée de tout objet. Ainsi, si une collision apparaît entre deux frames, le personnage sera automatiquement repoussé, et cette collision n'apparaîtra pas à l'écran (ou très rapidement).

En général, la technique de lancer de rayon n'est pas mauvaise si elle est bien appliquée et si le nombre d'objets dans la scène n'est pas trop important (tester tous les triangles de tous les objets d'une scène peut vite devenir très lourd).

Gestion des inputs et de la caméra

Gestion des inputs

Lors de la création d'un projet sous Ogre, il est possible de créer un projet simple dont la majeur partie de la configuration de base est déjà faite, notamment celle des gestions des périphériques de manipulation comme le clavier ou la souris, et la gestion de la caméra (par défaut, une caméra à la première personne).

Néanmoins, il est vivement recommandé de ne pas utiliser ce système simpliste au profit d'une architecture personnelle, généralement plus performante car optimisée au projet.

Nous avons choisi cette option car cela nous donnait entre autres la souplesse nécessaire à la personnalisation du moteur Ogre pour notre application. Nous avons séparé le code en six classes dont les trois principales sont :

- *ShortStory* qui contient la boucle de rendu et la paramétrisation du moteur Ogre
- *InputListener* qui gère tous les inputs (clavier et souris)
- Scene dans lequel sont créées toutes les entités apparaissant dans la scène

Cette séparation a dû être effectuée très rapidement afin que chacun sache dans quelle partie du projet travailler et éviter au maximum les conflits lors de la mise en commun du travail effectué.

La gestion des inputs se fait donc dans *InputListener*. Il est possible de réaliser certaines actions comme le déplacement d'un objet à plusieurs instants clefs dans la boucle de rendu d'Ogre. Nous réalisons tous les traitements des inputs et les actions qui en découlent lorsqu'Ogre vient d'envoyer l'image à afficher au GPU et que le processeur est donc théoriquement en repos.

La première chose que nous avons implémentée est le déplacement de la caméra (puis plus tard du personnage en 3° personne) avec les touches ZQSD. Une spécificité d'Ogre fait que par défaut, les claviers sont tous en QWERTY sous Windows (il faut donc définir les touches WASD dans Ogre pour utiliser en réalité les touches ZQSD), ce qui n'est pas le cas dans Linux (le clavier AZERTY est bien reconnu). Le déplacement de la caméra est aussi venu très rapidement, mais il a été perfectionné avec le temps.

Après l'intégration du ninja dans la scène, le déplacement du personnage est animé avec une des animations intégrées à ce modèle.

Au final, nous avons peu à peu ajouté des fonctionnalités rendant l'application manipulable, et facilitant le déplacement dans le jeu :

- augmenter la vitesse de déplacement du personnage en maintenant la touche shift (gauche) enfoncée

- possibilité d'enlever la détection de collisions avec la touche Fin. Dans ce mode, il est possible de monter à la verticale en appuyant sur Espace, ou de descendre avec Contrôle (gauche)
- passer d'une caméra en première ou en 3° personne avec la touche P (utilisable seulement en mode sans détection de collisions à l'intérieur de l'immeuble)
- connaître la position exacte du joueur avec K
- casser la vitre à n'importe quel moment avec B
- casser la vitre lorsque l'on est assez proche avec L

Gestion de la caméra

La caméra est gérée à l'aide de la souris. Lorsque le jeu commence, la caméra est en mode 3° personne (derrière le personnage) et passe automatiquement en mode première personne quand on rentre dans l'immeuble (cela évite au joueur d'avoir une meilleure vision dans l'immeuble). Lorsque l'on tourne la caméra, le personnage tourne en même temps.

Afin d'éviter au joueur de se perdre avec la caméra, nous avons interdit la possibilité de faire plus d'un demi-tour de rotation selon l'axe X. cela a été possible très facilement car l'origine de la rotation de la caméra est le nœud du personnage, qui a toujours la même orientation par rapport au repère de la caméra. Il a donc suffi de limiter la rotation selon l'axe X. Cela n'aurait pas été aussi facile si la caméra pouvait tourner librement autour du personnage.

Terrain et environnement

Le décor est tout aussi important que la scène elle-même, il fixe l'ambiance du jeu. Dans notre scénario, il fait noir, c'est la nuit, la pluie est abondante (voir la section sur les particules), le seul endroit pour s'abriter est l'immeuble.

Terrain

Le terrain crée le sol et l'horizon. Pour que le joueur ne se déplace pas sur un plateau aux bords visibles, nous avons généré notre terrain à partir d'une *heightmap*. C'est donc un terrain généré par une image en niveau de gris. L'immeuble est placé au centre d'une cuvette, pour la créer, nous avons trouvé la *heightmap* d'une île, grâce à une inversion de couleur, on obtient un plateau avec des bords.

Il y a quelques paramètres à ajuster pour créer le terrain, il faut définir la taille du monde, la hauteur et la gestion du niveau de détail (lod) sur le terrain. Une hauteur est d'environ un dixième de la taille du monde, cela permet d'avoir des collines de taille suffisante. Pour la gestion du lod, deux paramètres sont à prendre en compte ; il indique la taille minimale et maximale des *batchs* divisant le terrain. Il est préférable que l'ordre de grandeur de ces deux valeurs soit de 2^n + 1, le maximum ne peut dépasser 65. Comme il fait sombre et que le joueur ne marche pas directement sur ce terrain, nous avons choisi d'avoir une valeur minimale proche de celle maximale, ainsi, la variation du lod sera rapide et le calcul rapidement simplifié sur la distance.

Obtenir l'image qui gère les variations de hauteur du terrain ne suffit pas, il faut aussi les textures à plaquer en fonction des variations de niveau. Nous utilisons celles mises à notre disposition par Nvidia dans les medias d'Ogre. Il y a une texture terre, herbe et cailloux.

Le choix de la texture à appliquer est dépendant de la hauteur du terrain. Sur une faible partie, nous avons mis de l'herbe, cela correspond à la partie plate. Cette première section donne la main de manière diffuse à la texture de cailloux puis de terre car on arrive sur la partie abrupte de la colline.

L'éclairage du terrain est indépendant de l'éclairage ambiant. Cette lumière est directionnelle, comme il fait nuit sa couleur diffuse est noire et celle spectrale est gris moyen afin de conserver des reflets. Sa direction est identique à la lumière ambiante.

Ciel

Pour le ciel, nous avons utilisé un *sky plane*, comme nous avons des collines qui définissent le contour de notre terrain, c'est ce qui est le plus adapté. L'image est animée, c'est un ciel étoilé avec des nuages qui se déplacent. La transition entre le sol est le ciel est adouci par un brouillard.

Brouillard

Normalement, le brouillard est utilisé pour éviter les apparitions brutales d'objets dans le frustom. Dans notre cas, comme le terrain est petit (il est visible dans tout le champ de la caméra) et que tous les objets sont en son centre ; nous utilisons le brouillard pour augmenter l'effet de nuit et adoucir la transition entre le sol et le ciel.

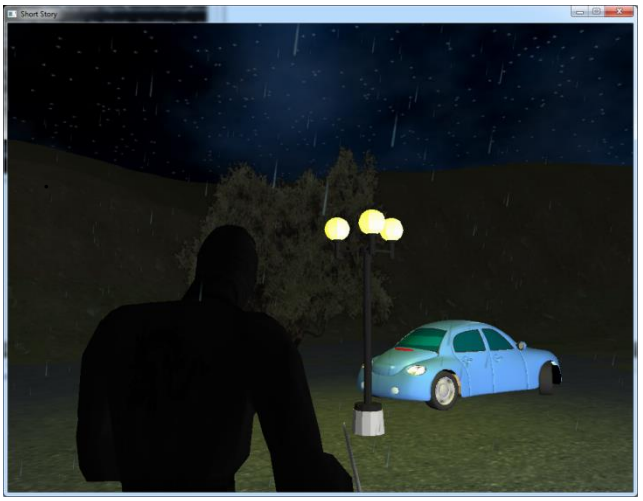
Dans Ogre, il existe trois types de brouillard : linéaire, exponentiel et exponentiel2.

Le type linéaire est un simple brouillard compris entre deux distances partant de la caméra. Le résultat est trop opaque pour l'effet recherché.

L'exponentiel et exponentiel2 sont des brouillards dont la densité est croissante. Le seul paramètre qui les concerne permet de gérer cette densité. La différence entre ces deux fonctions tient dans la fonction de densité utilisée, la première est exponentielle et la seconde est élevée au carré.

Comme nous cherchons à augmenter l'impression de nuit, nous avons utilisé un brouillard exponentiel2 de couleur noir.

Voici le résultat avant et après l'application du brouillard :



Eléments du décor

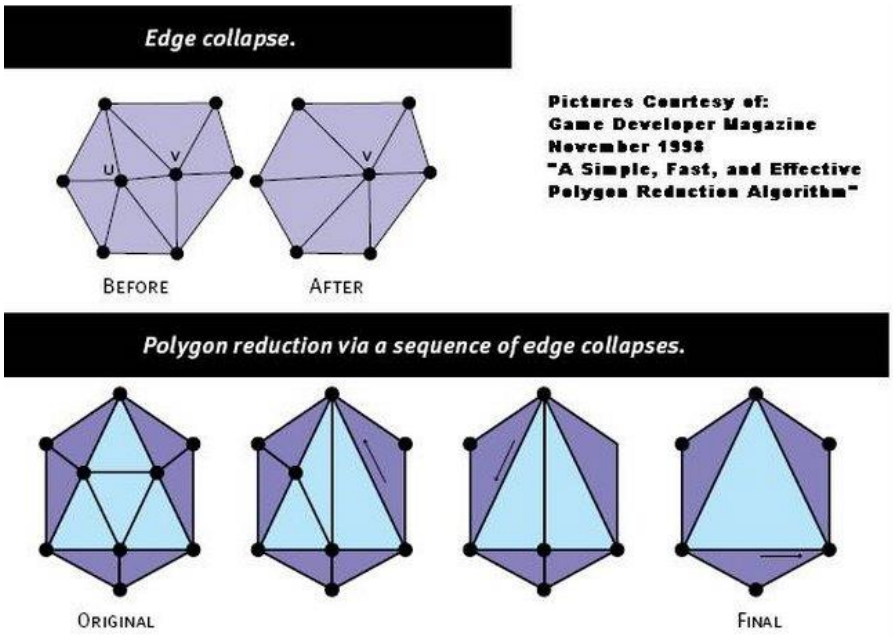
Comme il fait nuit noire, il est nécessaire de rajouter de la lumière sur la scène. Avec les lampadaires à trois boules, on augmente localement la luminosité au centre de la scène.

L'ajout de l'arbre qui est un imposteur donne du naturel au décor.

Niveau de détail

Ogre3D offre la possibilité d'utiliser un LOD discret. L'une des possibilités est de créer manuellement des meshs simplifiés depuis son outil de modélisation et d'indiquer quel mesh doit être utilisé pour tel niveau de détail. Mais nous avons retenu l'autre possibilité. Lors du démarrage de l'application, des sous-meshs sont précalculés depuis le mesh principal. L'intérêt est que ces sous-meshs disposent d'un rendu moins lourd, mais on a une perte de détail sur l'objet.

Ogre3D intègre un algorithme de réduction du nombre de sommets : *Progressive Mesh type Polygon Reduction Algorithm* de Stan Melax. L'objectif de la réduction de polygone, est d'à partir d'un mesh détaillé de créer un mesh contenant moins de détail, mais qui s'affichera plus rapidement (le temps d'affichage du mesh dépend du nombre de polygones).

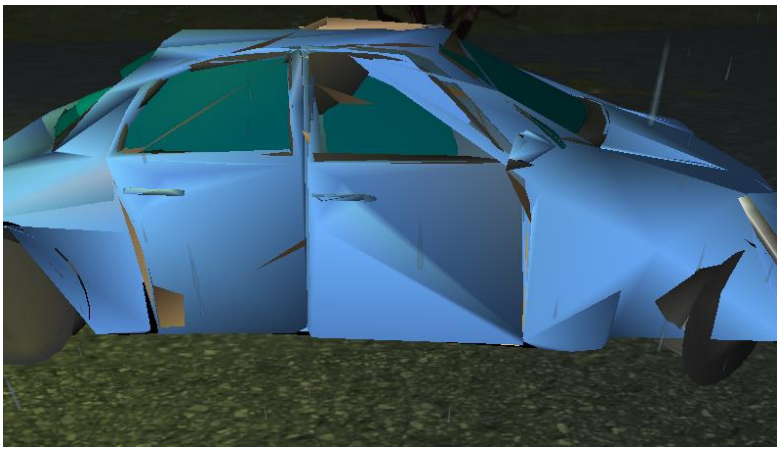


Ogre3D propose d'indiquer si on désire enlever un nombre constant de vertex ou une proportion de vertex à chaque réduction :

- VRQ_CONSTANT
- VRQ_PROPORTIONAL
- VRQ_ERROR_COST

La dernière valeur est particulière. On n'indique pas une quantité de vertex, le coût maximal d'erreur autorisé pour un vertex lors de la réduction. Si le vertex a un coût d'erreur de réduction inférieur au coût maximal d'erreur de réduction, on le supprime.

L'un des problèmes que l'on peut rencontrer est le débordement des textures. En effet, malgré la modification du mesh, la texture quant à elle ne subit pas de modification. De ce fait, en appliquant la texture sur le nouveau mesh, on peut obtenir des effets assez moches. Heureusement, dans notre cas (LOD discret), il suffit de préalablement tester le résultat et modifier les paramètres pour éviter ces débordements de texture.



Lors de l'affichage, Ogre3D détermine quel sous-mesh de l'objet il doit utiliser en se basant sur la distance entre la caméra et le nœud feuille de l'objet. Voici le résultat présent dans notre projet.

Picking

Le picking consiste à lancer un rayon dans la scène pour détecter des objets. Ogre3D offre la possibilité d'interroger la scène pour effectuer du picking. On indique la position de départ et la direction du rayon et Ogre3D nous retourne la liste des objets qui ont collisionnés avec le rayon. Lorsque le rayon touche un terrain, on peut aussi obtenir le point sur le terrain touché par le rayon.

Nous utilisons ce système dans notre système de picking. Le système de picking doit faire le tri dans la liste retourner par Ogre3D et vérifier que l'objet n'est pas trop éloigné. De cette manière, on vérifie que notre personnage est bien en face de la vitre ou de la porte avant de lancer l'une des animations.

Particules

Le système de particules sous Ogre3d

Les particules sont des éléments générés par un émetteur et représentés par un imposteur.

La gestion des particules dans Ogre se fait à base de scripts dont l'extension est *particle*. Ces scripts sont rédigés en suivant le même concept que les matériaux. Le format de ces scripts est du pseudo-C++. Les sections sont délimitées par des accolades et le commentaire se fait avec un double slash.

La première section est spécifique au type de script. Il s'agit de *particle_system* pour les particules. Les paramètres les plus fréquents sont :

- *matrial* : utilisé pour indiquer quel matériau est utilisé pour le rendu de chaque particule.
- *particle_width* et *particle_height* : utilisées pour préciser les tailles de chaque entité.
- *quota* : indique le nombre maximum de particules à un instant donné. Pour éviter une interruption involontaire de l'effet, car il y a trop de particules, il convient d'adapter le quota au nombre de particules émises au taux d'émission et à leur durée de vie.

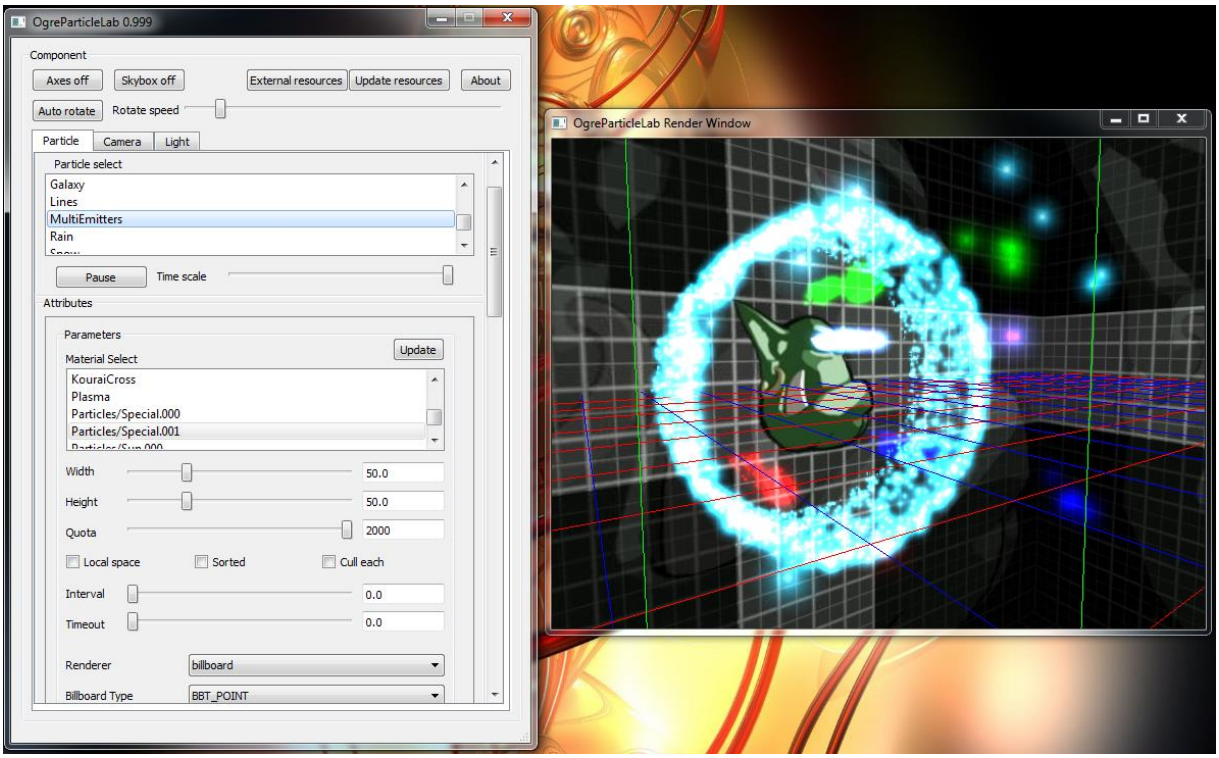
Pour la génération des particules, Ogre met à notre disposition différents types d'émetteur tels que le point, la boite, le cylindre, l'ellipsoïde, l'anneau et l'ellipsoïde creuse. Tout système de particules peut avoir plusieurs émetteurs distincts et indépendants. Pour chacun d'entre eux, il convient d'indiquer le nombre de particules par seconde, leur durée de vie… Pour que le rendu ne soit pas monotone, certains paramètres comme la durée de vie ou la vitesse de déplacement peuvent être mentionnés sous la forme d'une plage.

Ce n'est pas parce que la particule est émise qu'elle n'est plus modifiable. Il existe des sections *affector* de différents types comme *rotator* ou *scaler* qui permettent respectivement de programmer un comportement de rotation et de changement d'échelle dans le temps.

Comme mentionné au début, un système de particules se voit affecter un matériau. Également paramétrable sous la forme de script, il est possible d'appliquer aux matériaux d'autres programmes exécutés cette fois par le moteur de rendus, ce sont les *shaders*. Parmi les programmes existants, on retrouve le glsl pour l'OpenGL, l'hlsl de DiectX et le CG de Nvidia.

Comme on peut le voir, créer un bel effet de particules nécessite d'ajuster de nombreux paramètres. Comme il est extrêmement agaçant d'exécuter son application pour constater la modification apportée, il existe une application pour visualiser et paramétrer ses effets, il s'agit de Ogre Particle Lab (voir ci-dessous). Cependant attention, cette application utilise OpenGl comme

moteur de rendu, de ce fait, certaines particules présentent dans sa bibliothèque ne sont donc pas utilisables partout.



Les effets de particules mis en place

Notre application utilise trois systèmes de particules différents, il y a la pluie, le verre brisé et les flammes sous le robot.

La pluie

Notre pluie se veut abondante, dense et elle doit rester à l'extérieur. Cette brève description montre bien les problèmes à régler. Pour avoir une pluie abondante et dense, il faut augmenter le nombre de particules émises en même temps et une vitesse élevée. Le but est que le joueur perçoive la pluie en permanence sans qu'il puisse compter les gouttes.

Problème du nombre de particules

Mettre un émetteur au-dessus de toute la surface du terrain et suffisamment haut pour dépasser l'immeuble est l'approche naïve. Le nombre de particules à émettre est bien trop important pour que cela soit supportable par l'application, le *lag* sur les ordi-centre est perceptible. De plus, cela pose un autre problème, il pleut dans le bâtiment.

En effet, comme tout objet de la scène, sans précision, les particules traversent tout. Sous Ogre, il n'y a pas de boite d'exclusion et l'idée de faire une détection de collision pour chaque élément est à ranger au côté des contes et légendes. Bref, cette solution est mauvaise.

Nous avons donc attaché notre système de particules au nœud de la caméra et du personnage, ainsi l'émetteur peut être beaucoup plus petit et les nombres de particules à générer pour le même résultat sont tout à fait convenables. Il y a tout de même quelques précautions à prendre.

Une fois que les particules sont émises, elles deviennent indépendantes de leur émetteur. Déplacer ce dernier ne change donc pas la position des particules existantes. Pour éviter que le joueur ne fasse la course avec la pluie, c'est-à-dire qu'il sort de la zone de retombée des gouttes, il faut que l'émetteur soit une boite qui englobe la caméra sur une distance suffisante.

Pour limiter le nombre de particules, on peut encore jouer sur la hauteur de la boite émettrice. Inutile qu'elle fasse la taille de l'immeuble, mais si elle est trop petite une caméra qui regarde vers le ciel verra les goutes se créer.

Afin d'améliorer l'effet de désastre naturel, on ajoute un angle à l'émetteur pour incliner les gouttes. On déforme l'image pour que celle-ci soit effilée afin que cela corresponde à la vitesse de chute, et on l'oriente avec le même angle que l'émetteur.

Problème de l'immeuble

Nous avons réglé le problème du nombre, reste celui de l'immeuble. À moins qu'il ne soit délabré, il faut éviter qu'il pleuve à l'intérieur. Pour cela rien de plus simple, le lancer de rayon qui gère la hauteur entre le sol et la caméra, sait sur quel objet il est projeté. Si c'est sur l'immeuble, alors on coupe la pluie.

Afin que le joueur constate que la météo ne s'arrange pas quand il regarde par une fenêtre, on affiche une bande de pluie sur les faces de l'immeuble qui possèdent des vitres.

Si cette technique devenait trop gourmande, il est possible d'ajuster la hauteur des émetteurs en fonction de la hauteur de la caméra dans l'immeuble. Mais cela n'a pas été nécessaire.

Le verre brisé

Une fenêtre qui casse, ça fait des morceaux de taille variable et certains ne tombent pas. Pour avoir un émetteur de *meshs*, il faut le créer. Ce qui revient à patcher le plug-in ParticleFX. Le seul lien encore accessible traitant de ce sujet se trouve sur le forum d'Ogre3d (http://www.ogre3d.org/forums/viewtopic.php?f=2&t=19606). En revanche, il est écrit pour la version 1.0.6 de OgreSDK, et on utilise la 1.8.1, il y a plus de six ans d'écart et de nombreuses modifications. Nous avons donc changé de principe… Nous allons casser une vitre faite avec du verre sécurité.

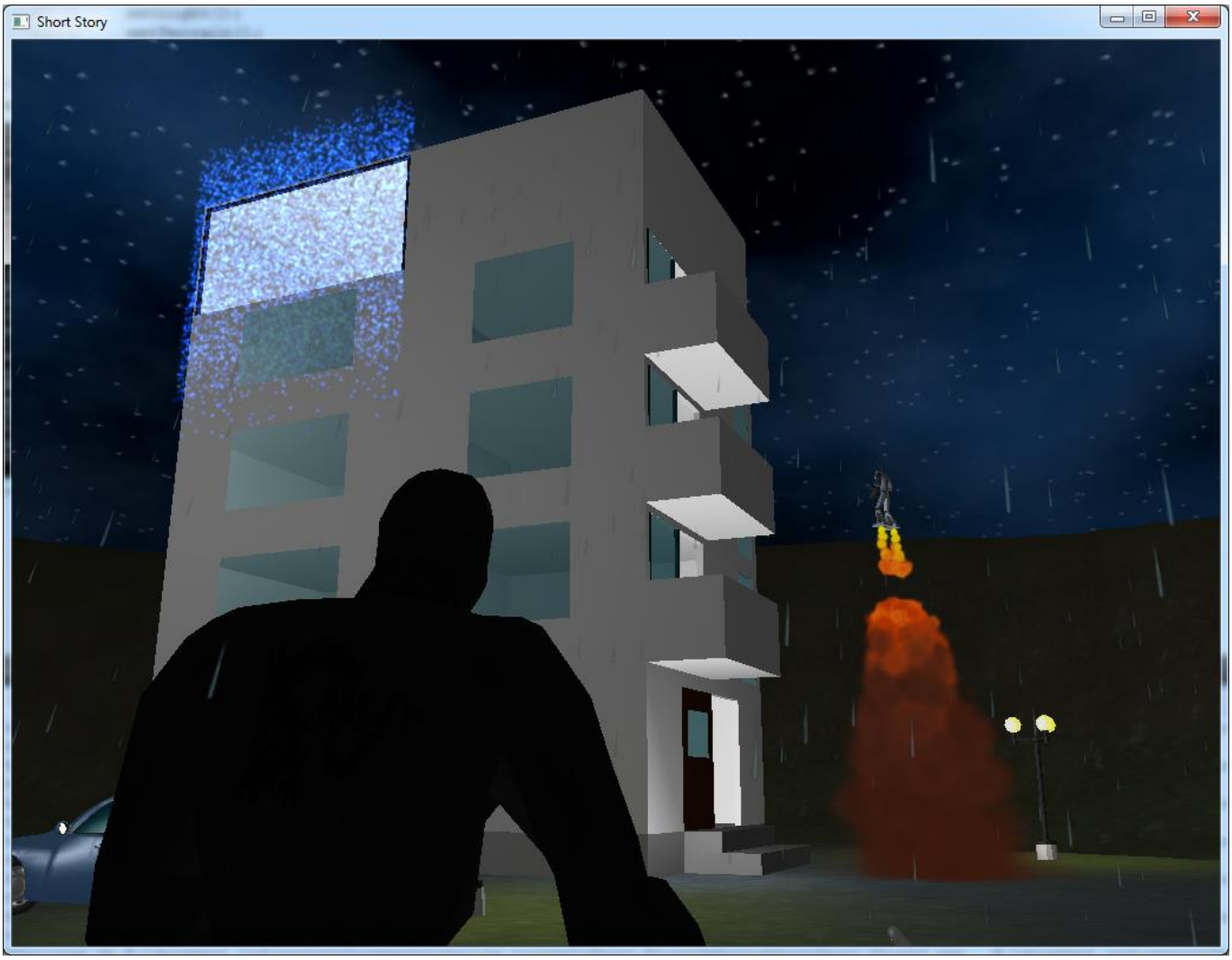
Le gros avantage de ce type de verre, c'est qu'il se casse en plein de petits morceaux de taille homogène et qu'ils tombent tous en même temps. L'inconvénient réside dans le nombre de particules à générer subitement.

Comme les morceaux sont d'aspect irrégulier et que leurs côtés reflètent la lumière, il faut un matériau qui utilise une image au contour flou, car avec la vitesse de chute des débits, les facettes n'apparaissent pas distinctement.

Au niveau de l'émetteur, les couleurs de particules doivent changer dans le temps, car les reflets évoluent et l'impact de la lumière aussi. C'est pourquoi les couleurs sont des variations de bleu marine, bleu bris et un peu de violet.

L'émetteur plan est placé au même niveau que la fenêtre, et quand celle-ci est masquée, l'effet de particule est affiché.

Voici le résultat final avec toutes les particules :



Son et lumière

Lumière et ombre

La gestion des lumières et des ombres est très importante pour créer l'ambiance dans une scène. Ogre met à disposition toute une panoplie de lumières et d'ombres pour créer l'ambiance que l'on souhaite.

Lumières

Ogre permet de mettre en place différents types de lumières :

- Lumière ambiante (*ambient light*) : c'est une lumière dont l'intensité est la même en tout point de l'espace.

  Pour définir une lumière ambiante sous Ogre, on utilise la fonction *setAmbientLight()* du *scene manager*, qui prend en paramètre une couleur. Par exemple, dans notre application, nous avons défini la lumière ambiante de la manière suivante :

```
mShortStory->getSceneManager()->setAmbientLight(ColourValue(0.3f, 0.3f, 0.3f));
```

  NB : Il ne peut y avoir qu'une seule lumière ambiante pour une scène. Toute autre déclaration viendra remplacer la première.

Pour les lumières suivantes, la création se fait en appelant la fonction *createLight()* du *scene manager*, prenant en paramètre le nom donné à la lumière créée, et retournant un objet de type Ogre ::Light*. Par exemple :

```
Ogre ::Light* streetLamp2B1 = mShortStory->getSceneManager()->createLight("streetLamp2B1");
```

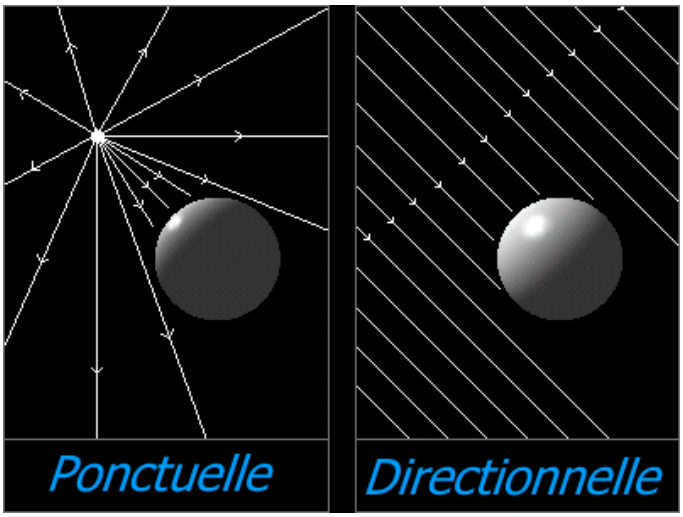
Ensuite, une fois la lumière créée, on lui donne un type, grâce à la fonction *setType()* parmi les autres types de lumières.

- Lumière ponctuelle (*point light*) : elle émet dans toutes les directions depuis sa position.
  Le type Ogre associé à une lumière ponctuelle est Light ::LT_POINT. Par exemple :

```
streetLamp2B1->setType(Ogre::Light ::LT_SPOTLIGHT);
```

- Lumière directionnelle (*directional light*) : elle émet depuis l'infini des rayons allant dans une seule direction (les rayons sont tous parallèles à cette direction). Son type est *Light::LT_DIRECTIONAL* ; on l'utilise pour imiter le soleil dans une scène 3D. Dans notre application, nous l'avons donc appliquée sur notre terrain :

terrainLight->setType(Ogre ::Light ::LT_DIRECTIONAL);



- Le projecteur (*spot light*) : lumière qui émet des rayons à l'intérieur d'un cône, à la manière d'une lampe torche. C'est le type *Light ::LT_SPOTLIGHT*, et nous l'avons utilisé pour toutes nos lampes extérieures, mais aussi à l'intérieur de l'immeuble :

streetLamp2B1->setType(Ogre::Light ::LT_SPOTLIGHT);

Enfin, il faut positionner et paramétrer les lumières créées. La position est affectée à une lumière par la fonction *setPosition()*, prenant le vecteur 3D correspondant aux coordonnées où l'on désire placer la source de lumière.

Voici un exemple issu de notre code, permettant de positionner une lumière en fonction de la position du nœud correspondant au *mesh* d'un lampadaire extérieur :

streetLamp1B1->setPosition(streetLamp1Node->getPosition() + Ogre ::Vector3(30, 500, 15));

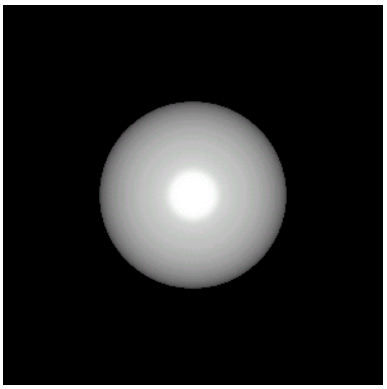
NB : 1) Il n'est pas utile de créer la lumière ambiante, et encore moins de la positionner.

De même, une lumière directionnelle émettant depuis l'infini, il est inutile de préciser sa position.

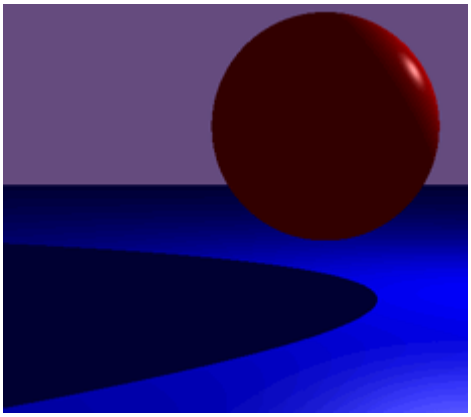
À chaque type de lumière on peut spécifier certaines propriétés telles que la couleur émise (diffuse, spéculaire), l'atténuation (pour une lumière ponctuelle), la direction (pour une lumière directionnelle) et l'angle d'ouverture (pour un spot).

La couleur diffuse : Il faut savoir que la lumière incidente est réfléchie dans toutes les directions lorsqu'elle atteint un objet. La composante diffuse indique l'intensité qui repart, en tenant compte de l'inclinaison avec laquelle la lumière incidente arrive sur la surface, en supposant que l'intensité est la même quelle que soit la direction que prend le rayon réfléchi. On spécifie sous Ogre qu'une source lumineuse a une influence diffuse sur la couleur d'un objet de la manière suivante :

streetLamp1B1->setDiffuseColour(1., 1, 0);



NB : Une lumière ponctuelle a une influence diffuse et spéculaire sur la couleur d'un objet :



La direction, l'atténuation et l'angle d'ouverture sont des propriétés spécifiques respectivement à un seul type de lumière (lumière directionnelle, lumière ponctuelle, le spot).

- La direction : la méthode *setDirection()* permet de définir le vecteur directeur des rayons lumineux d'une lumière directionnelle.

```
terrainLight->setDirection(0.55f, -0.3f, 0.75f);
```

- L'atténuation : la méthode *setAttenuation()* permet de limiter la portée de l'éclairage d'une lumière ponctuelle. Chacun de ses paramètres doit être choisi avec soin. Le premier paramètre est la distance caractéristique d'atténuation, c'est-à-dire la distance à partir de laquelle la luminosité diminue ; le deuxième paramètre est une constante d'atténuation comprise entre 0 et 1 (plus elle est proche de 0 et plus le passage de la lumière à l'ombre est brutal) ; enfin, *linear* et Q*uadratic* sont les paramètres de la courbe d'atténuation (ils doivent être assez faibles, sinon la lumière s'atténue trop rapidement).

```
Real range = 2000;
Real attenuation = 1.0;
Real linear = 4.5 / range;
Real Quadratic = 75.0 / (range * range);
streetLamp1B1->setAttenuation(range, attenuation, linear, Quadratic);
```

- L'angle d'ouverture : Le spot propose une lumière directionnelle définie dans un cône central et un cône extérieur, avec 2 intensités différentes : la lumière est plus intense dans le cône central et est atténuée dans le cône extérieur. Ces deux cônes sont définis par leur angle d'ouverture et un coefficient indiquant si la transition d'intensité entre les deux cônes doit être plus ou moins rapide (on l'appelle le *falloff*). La valeur du *falloff* vaut par défaut 1.0, et si on spécifie une valeur inférieure c'est l'atténuation est plus lente, sinon si la valeur donnée est supérieure à 1.0, l'atténuation est plus rapide.

Dans notre application, nous avons laissé le *falloff* à sa valeur par défaut :

```
streetLamp3B3->setSpotlightRange(Ogre ::Degree(30), Ogre::Degree(90));
```

NB : le spot étant une lumière directionnelle émettant dans un cône, il nécessite lui aussi qu'on spécifie sa direction.

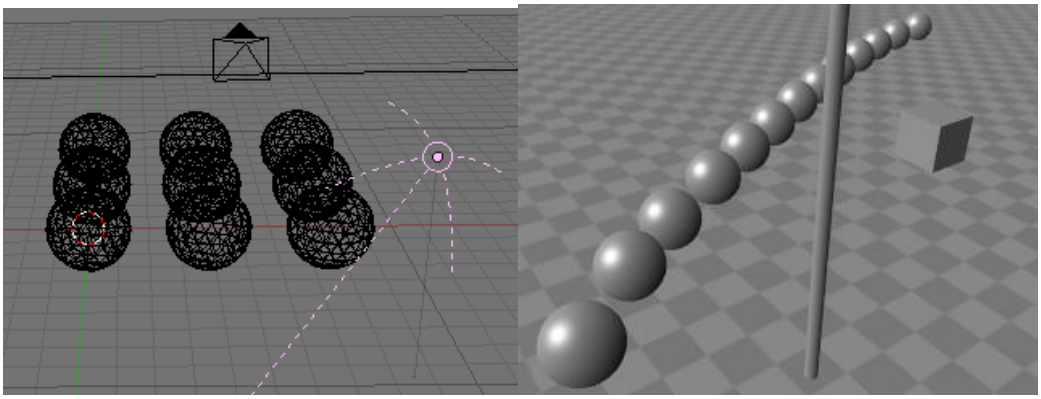
Afin de simplifier le positionnement des sources lumineuses dans Ogre, pour qu'elles soient placées de manière adéquate sur les *meshs*, nous avons étudié la création de sources lumineuses sous Blender, et leur exportation vers Ogre.

Sous Blender, il est possible d'ajouter des lampes (sources lumineuses) de différents types. Pour cela, il faut se placer en mode *Object* et appuyer sur la barre d'espace du clavier avec notre pointeur de souris dans la vue 3D. Un menu s'ouvre alors, et pour ajouter une lampe, on clique sur *Add->Lamp*. Les principaux contrôles sous Blender sur les lampes sont les suivants :

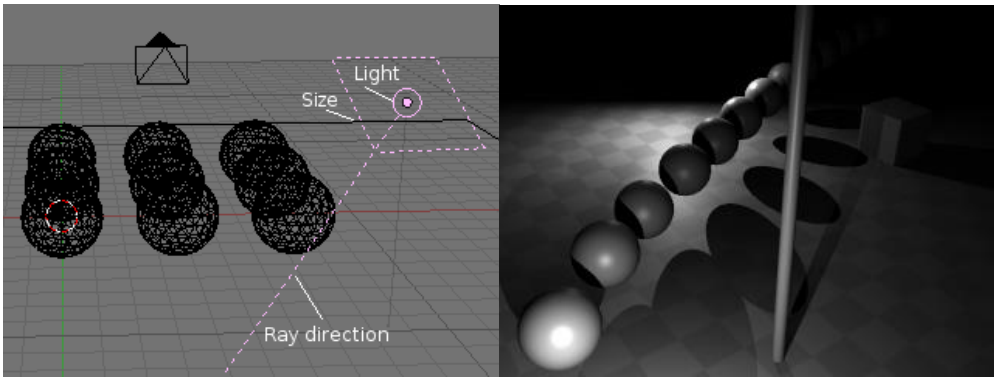
- Sélection du type de lampe parmi Point, Sun, Spot, Hemi, Area.
- La couleur de la lampe
- Les positions et direction de la lampe
- Les réglages de la lampe, dont l'énergie et la décroissance de l'intensité lumineuse.

Les types Point, Sun et Spot correspondent respectivement aux types Point Light, Directional Light et Spot Light de Ogre. Les sources lumineuses de type Hemi et Area sont des lampes un peu plus spécifiques :

La lampe Hemi émet de la lumière depuis un hémisphère, afin de simuler la lumière venant d'un ciel uniforme (par exemple nuageux). En d'autres termes, c'est une lumière émise uniformément par un dôme lumineux englobant la scène. Tout comme une lampe de type Sun, il n'est pas nécessaire de préciser sa position. Son orientation par contre est importante : la lampe Hemi est représentée par 4 arcs de cercle représentant l'orientation du dôme, et une ligne brisée indiquant la direction dans laquelle est émis le maximum de lumière, vers l'intérieur de l'hémisphère.



La lampe Area simule la lumière venant d'une surface émettrice (par exemple un écran TV, une fenêtre, …).



Nous ne détaillerons pas plus la gestion des lampes sous Blender, car nous ne les avons pas utilisées.

Il faut toutefois savoir également qu'il est possible d'exporter une scène Blender contenant des lampes dans un format exploitable par Ogre : .scene. Ce format de fichier contient toutes les caractéristiques de la scène complète : positions-orientations, sources lumineuses, couleurs, brouillard et autres.

Il est ensuite possible d'utiliser directement sous Ogre le fichier *dotScene* ainsi exporté, mais il faut pour cela implémenter un parseur de ce type de fichier, basé sur du XML.

N'ayant malheureusement pas eu le temps de mettre en œuvre ce parseur sous Ogre, cette solution de création des sources lumineuses sous Blender et de les exporter pour Ogre a été mise de côté et nous avons placé les lumières « à tâtons » à la bonne position sur les *meshs* des lampes directement sous Ogre.

Ombres

Pour mettre en place des ombres dans une scène il faut :

- Une ou plusieurs sources lumineuses
- Un sol
- Spécifier que les lumières et objets de notre scène projettent des ombres ; il est tout à fait inutile que notre sol émette lui-même des ombres. Pour cela, nous utilisons la propriété setCastShadows(), qui est par défaut à *FALSE*, et nous la passons donc à *TRUE* pour les objets qui doivent avoir une ombre :

```
Trees->setCastShadows(true);
streetLamp1->setCastShadows(true);
entVoiture->setCastShadows(true);
```

Une fois les ombres activées sur les objets voulu, il faut choisir un type d'ombre et une technique de rendu parmi :

- Type *Stencil*
- Type *Texture*

Et pour chacun de ces deux types :

- Technique *Modulative*
- Technique *Additive*

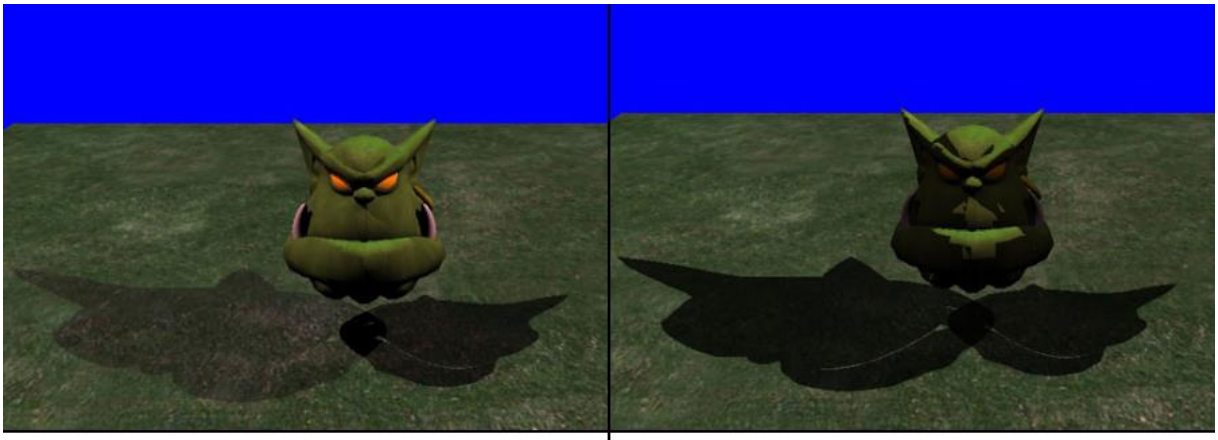
On obtient ainsi 4 techniques d'application des ombres sous Ogre :

- SHADOWTYPE_TEXTURE_MODULATIVE
- SHADOWTYPE_TEXTURE_ADDITIVE
- SHADOWTYPE_STENCIL_MODULATIVE
- SHADOWTYPE_STENCIL_ADDITIVE

Les ombres de type *stencil* sont très précises dans les contours et permettent une très bonne projection d'ombre lorsqu'on y regarde de près. Elles sont toutefois assez coûteuses en ressources. De plus, elles ne prennent pas en compte la transparence des objets : un cube transparent projettera donc une ombre, comme s'il était opaque.

Les ombres de type *texture* sont moins précises et donc moins coûteuses en ressources, mais elles permettent de gérer la transparence des textures.

Enfin, dans chacun des cas, la technique *additive* donne un résultat plus réaliste que la technique *modulative*. La différence est peu flagrante pour un type *texture*, mais elle est très visible pour un type *stencil* :



Pour projeter des ombres, il faut les activer dans le *scene manager*, en spécifiant quelle technique on souhaite appliquer pour le rendu :

```
mSceneMgr->setShadowTechnique(Ogre::SHADOWTYPE_STENCIL_ADDITIVE);
```

Son

Ogre est uniquement un moteur 3D, il n'a donc pas de moteur de son intégré. Pour jouer des sons sous Ogre, il faut donc utiliser une bibliothèque sonore compatible, à ajouter dans les paramètres du projet.

Dans le cadre de notre projet, nous avons trouvé plusieurs moteurs de son tels que OpenAL, DirectSound, Irrklang, EAX, I3DL2, A3D, FMOD, Audiere et d'autres. Mais un seul nous a vraiment attirés : il s'agit de Irrklang.

Irrklang

Irrklang est un moteur de son et une librairie audio de haut niveau, développée par Ambiera, permettant de jouer des sons 2D et 3D. Irrklang est multiplateforme, et fonctionne donc sur Windows, Unix ou Mac OS.

Irrklang est utilisable dans les langages .NET et C++, ce qui nous convient parfaitement pour notre application, codée en C++, d'autant plus que Irrklang est gratuit pour un usage non commercial, ce qui est notre cas.

Il est possible de jouer des sons de tous formats : WAV, MP3, OGG, FLAC, MOG, XM, IT, S3M et bien d'autres encore.

L'installation d'Irrklang est très simple : il suffit de télécharger la bibliothèque, de copier les DLL dans le dossier de notre projet, et de renseigner qu'on l'utilise dans notre projet. Ensuite, on inclut les headers et on utilise les fonctions de la bibliothèque de manière tout à fait usuelle.

Son utilisation est elle aussi des plus aisées : une poignée de fonctions permettent d'accéder à toutes les fonctionnalités :

- Créer un moteur de sons
- Jouer un son en 2D
- Jouer un son en 3D
- Sélectionner l'emplacement de l'utilisateur dans l'univers 3D
- Spécifier des options pour les sons

Pour jouer un son avec Irrklang, il faut commencer par créer une instance de *ISoundEngine* :

```
irrklang::ISoundEngine* engine = irrklang::createIrrKlangDevice() ;
```

Ensuite, on peut jouer un son 2D ou 3D en appelant la fonction play2D() ou play3D(), prenant en paramètre le chemin du fichier sonore à lire et un booléen indiquant si on le joue en boucle ou non (la valeur par défaut est *FALSE*), ainsi qu'une position (vecteur 3D) pour un son 3D :

```
engine->play2D(« someSound.wav ») ;

irrklang ::vec3df position(20,75,90);

engine->play3D(« some3Dsound.mp3 », position) ;
```

Notez bien que la fonction play3D() prend bien d'autres paramètres optionnels, permettant de paramétrer le son de manière plus précise.

Enfin, il faut éteindre le service en appelant la fonction IRefCounted ::drop() :

```
engine->drop() ;
```

NB : En plus de libérer notre variable engine, il faut penser à libérer tous les autres pointeurs qui ont été alloués, tels que des sons (type ISound*) etc… Avec la même méthode drop().

Dans le cadre de notre projet, l'utilisation de Irrklang s'est avérée impossible, car, bien que nous ayons correctement installé Irrklang pour accéder à ses fonctionnalités sous Ogre, nous n'avons pas pu générer de son !

Pourquoi ? Tout simplement, car Irrklang fonctionne avec un compilateur plus ancien que celui que nous utilisons pour notre application ! Nous avons notamment pu constater ce fait grâce à des exemples fournis dans le dossier téléchargé d'Irrklang : en lançant l'exécutable déjà disponible, nous avons un son qui se joue correctement, mais en recompilant les projets exemples les exécutables ne jouaient plus de son… Nous avions également une fenêtre préventive s'affichant sous Code ::Blocks pour informer de la différence de versions entre le compilateur utilisé pour recompiler le projet et celui utilisé lors de sa création.

Notre choix d'utilisation de la dernière version de gcc a été imposé par notre choix d'utilisation de la dernière version d'Ogre3D. C'est ainsi qu'il nous était impossible de revenir sur une version antérieure de gcc.

Nous avons alors décidé de revenir à un standard : OpenAL.

OpenAL

OpenAL (Open Audio Library) est une API audio 3D multiplateforme, développée par Loki Software et Creative Labs. OpenAL est un standard et est distribuée selon les termes de la licence publique générale limitée GNU (hormis quelques drivers propriétaires tels que ceux pour la Xbox…).

OpenAL permet de modéliser un ensemble de sources sonores se déplaçant dans un espace en trois dimensions, et un auditeur dans ce même espace.

En bref, OpenAL est une bibliothèque des plus complètes. Alors, pourquoi ne l'avons-nous pas privilégiée dans notre premier choix ? Tout simplement parce que OgreAL, l'interfaçage de OpenAL avec Ogre est un projet abandonné. De ce fait, nous avons d'abord cherché d'autres moteurs de sons avant de revenir sur cette solution.

En fin de compte, nous ne sommes pas parvenus à installer OpenAL pour l'utiliser sous Ogre : l'installation avec *oalinst* affichait « OpenAL installé », mais nous n'avons jamais pu trouver le moindre dossier exploitable, contenant les « include » et autres fichiers nécessaires pour l'incorporer dans notre projet et l'utiliser sous Ogre.

C'est pourquoi, au vu du caractère optionnel de la gestion du son dans notre application, nous n'avons pas poursuivi nos recherches et avons abandonné l'idée d'inclure du son dans notre projet.

Scène automatique

La partie du rapport du sur la scène automatique n'a pas été rédigée par la personne concernée après maintes demandes. Il n'y aura donc pas d'explications du travail effectué sur cette partie.

Bilan

Cette expérience fut avant tout une expérience de travail en groupe qui prouve que plus on est nombreux sur un projet et une période réduite plus il est important d'avoir un système de gestion de versions et une bonne organisation de travail. La présence d'un chef de projet afin de coordonner le travail de chacun sans avoir besoin de faire des réunions quotidiennes avec toute l'équipe semble indispensable dans un tel contexte.

Ce projet a été une riche expérience des différentes contraintes liées aux environnements et architecturation. C'est un type d'application très spécifique qui nécessite une expérience et une approche différente de la programmation classique.