LEDRU Mathieu
3ème année de cycle ingénieur

MAISON DE L'INGENIEUR
Plateau de Moulon - Bat 620
Université de Paris Sud
91405 ORSAY

@ http://www.ifips.u-psud.fr/

Responsable école :
Aurélien MAX
☎ 01 69 33 86 14

18-20, rue Soleillet
75020 PARIS

@ http://www.kylotonn.com/

Responsable entreprise :
Hubert SARRET
☎ 01 43 15 08 17

RAPPORT DE STAGE DÉVELOPPEMENT PLAYALL

Du **24 mars** au **29 août 2008**

REMERCIEMENTS

Je tiens à remercier Monsieur Yann TAMBELLINI, directeur artistique et cofondateur de Kylotonn, d'avoir accepté ma candidature.

Je remercie Monsieur Hubert SARRET, mon responsable de stage et Benoît JACQUIER, de m'avoir encadré tout au long de mon stage. Ainsi que tous les membres de PlayAll, participants à un projet d'envergure réunissant plusieurs entreprises dont Kylotonn.

Toutes ces personnes ont su me soutenir et m'ont souvent assisté en m'informant et en me guidant dans les différentes tâches que j'ai été amené à réaliser. Ils m'ont aussi autorisé une certaine autonomie, ce qui est révélateur de la confiance qu'ils m'ont attribuée tout comme au personnel.

Je voulais aussi mettre en avant la confiance dont a su faire preuve les membres de PlayAll à mon égard. Durant tout mon stage, je n'ai pas eu l'impression d'avoir le statut de stagiaire mais celui d'un intervenant comme un autre.

De manière plus générale, je tiens à remercier l'ensemble de l'équipe de PlayAll pour son accueil chaleureux, pour toutes les connaissances et l'aide au développement qu'elle m'a apportés, sa gentillesse son soutien et sa disponibilité.

Enfin, je voudrais remercier les responsables de l'IFIPS encadrant ce stage en entreprise, Monsieur Aurélien MAX, Madame Brigitte Journiac et Madame Martine Rousseau.

ABSTRACT

Kylotonn is a French studio Created in 2002 based in Paris.

Kylotonn has a solid experience of high level action game based on past developments of 'IronStorm', 'Bet on Soldier', 'Blood of Sahara' and 'Black-out Saigon'. All of the developments follow a solid production scheme, mainly supported by in house proprietary tools and technology.

Kylotonn is today involved in a Research and Development platform PlayAll.

PlayAll is a structuring project and collaborative for the sector of the French Video game. PlayAll has for object the fusion of the technologies of several studios and the development of a common engine of game creation, as well as a technological standard shared with a certain number of companies of middleware and innovative research laboratories.

This federation of additional companies elaborated a strategy of collaborative Research and Development, allowing it to take place in a winning economic position for the next 5 years and the future technological evolutions.

I had the opportunity to work into PlayAll locals. I developed and integrated several components into the main PlayAll engine. There, I learn and reinforce my knowledge on C++ and shader usages and tricks. There is also all the background of the company to discover, as well on the human point of view as on its organization.

SOMMAIRE

1ERE PARTIE

INTRODUCTION

Ce rapport présente les travaux que j'ai pu effectuer au sein de la société Kylotonn Entertainement, société de développement et production de jeux vidéo située à Paris, dans le cadre de mon stage de fin d'étude d'ingénieur spécialisé en informatique.

Dû à l'amélioration constante des performances des cartes graphiques, que ce soit pour les PC ou pour les consoles de nouvelle génération « Next-Gen », les possibilités au niveau du rendu de scènes 3D ont grandement évoluées, et nombre d'effets graphique sont maintenant réalisables en temps réels. Mon étude a donc consisté à approfondir le développement au sein du moteur 3D PlayAll, afin d'y ajouter les fonctionnalités souhaitées.

Le contenu de ce rapport commence tout d'abord par une présentation de l'entreprise Kylotonn, son historique, ses activités, suivi par une présentation plus précise du stage en lui-même, le cadre et les objectifs. S'en suivra le développement sur le travail effectué durant ce stage, les solutions utilisées et les résultats obtenus. Finalement, je conclurai en faisant un bilan de mon étude et de mon stage au sein de Kylotonn.

2EME PARTIE

PRÉSENTATION DE L'ENTREPRISE

1 KYLOTONN ENTERTAINMENT

www.kylotonn.com

Kylotonn Entertainement est une SARL dont le siège est situe au 18-20 rue du Soleillet à Paris XXème. Kylotonn a été crée officiellement en 2006 et est issue de la société fondée par Roman Vincent, RVP (Roman Vincent Production) en 2002.

Après le développement « d'Ironstorm », de « Bet on Soldier » et, plus récemment, de « Blood of Sahara » ainsi que « Black-out Saigon », Kylotonn possède une solide expérience en matière de jeux d'action en *vue à la première personne* (ou First Person Shooter). Basés sur une technologie propriétaire, le KT Engine, tous les jeux sont réalisés sur une chaîne de production efficace et éprouvée.

La stratégie de Kylotonn repose sur deux fondamentaux :

- Développer des jeux dont Kylotonn conserve et exploite les droits, en collaborant avec des distributeurs voulant s'investir dans ces jeux.

- Travailler avec des éditeurs désireux d'externaliser le développement d'une licence ou d'un concept de jeu dont ils sont propriétaires.

Kylotonn compte aujourd'hui une trentaine de collaborateurs.

2 TECHNOLOGIE 3D KT ENGINE

La chaîne de production a été entièrement développée en interne. Elle est constituée d'un moteur 3D Next Gen, le KT Engine, ainsi que d'un éditeur de niveau « in-game », le KT Toolkit. Cependant, le projet PlayAll détaillé plus loin va lui permettre d'être utilisé par de nombreux studios, il devrait donc évoluer en conséquence.

Eprouvé sur la trilogie « BoS », le KT Engine tirait déjà parti de toutes les fonctionnalités offertes par les cartes 3D. Mise à jour dès la fin 2005, Kylotonn utilise désormais l'évolution 2.0 Next Gen de son moteur KT Egine pour ses productions actuelles et futures.

Les atouts majeurs de cette nouvelle version sont une gestion avancée des *shaders* et des *rendus pré-calculés*, couplée à différents systèmes d'optimisation d'affichage et de transfert d'informations :

- Moteur 3D complet : rendu, animations, sons, collisions

- Technologie de pointe en terme de rendu visuel et de performance : *placage de relief*, lightmap, *rendu pré-calculé* ...

- Gestion avancée des éclairages : *cartes d'ombres*, *volumes d'ombres*...

- Utilisation simplifiée pour les graphistes

Couplé au KT Toolkit, Kylotonn dispose d'un outil de travail efficace et éprouvé. Cet éditeur de niveau, entièrement intégré au jeu, permet de scripter en un temps record et de tester en temps réel les modifications apportées.

Enfin, Kylotonn travaille en étroite collaboration depuis plusieurs années avec des sociétés spécialisées dans *l'intergiciel* telles AGEIA (Carte PhysX) ou nVidia (Carte video 3D GeForce) ce qui leur a permis de s'offrir un support pour leur carte accélératrice d'effets physiques.

Le moteur de jeu puissant, permet un rendu de qualité et de bonnes performances. Il se situe au niveau de la plupart des moteurs actuels. Le rendu est de type *DirectX* 9, le passage à *DirectX* 10 n'est pas encore planifié. La majorité des fonctionnalités et des effets présents dans les jeux concurrents sont supportés.

3 JEUX DÉVELOPPÉS

- Ironstorm

En 2002, Roman Vincent crée la société de production RVP. L'équipe de développement du jeu Ironstorm, intègre la société RVP et crée le label de jeu d'action « Kylotonn ».

Première expérience « *vue à la première personne* », IronStorm (sous le nom 4x Studio) sortit en 2002, est un jeu d'action ayant lieu en 1964 alors que la première guerre mondiale n'a jamais pris fin.
Il a été vendu à 300 000 exemplaires dans le monde entier.

- Bet on Soldier

Durant trois ans, cette équipe va développer le jeu Bet On Soldier ainsi que les outils constituant la chaîne de production. Bet on Soldier, sorti en septembre 2005, est distribué dans plus de 30 pays C'est ce jeu qui a initié le modèle économique mis en place chez Kylotonn que nous verrons par la suite.

Bet On Soldier, suite spirituelle de IronStorm et reprenant le principe de ce dernier, B.O.S. se situe entre le jeu de guerre et la *vue à la première personne* d'arène (type Quake 3 Arena ou Unreal Tournament). Il décrit une guerre spectacle mise en scène par de puissantes industries où les meilleurs combattants s'affrontent en duel sur les champs de batailles. Introduisant un système original de paris et de gestion de l'équipement, B.O.S. à été bien accueilli par la presse spécialisée.

- Blood of Sahara et Blackout on Saigon

Enfin, début 2006, Roman Vincent et Yann Tambellini cofondent Kylotonn Entertainment qui devient une société à part entière, spécialisée dans l'*artwork* et la création de jeux vidéo d'action. Dans le courant de l'année 2006, deux suites au jeu

Bet On Soldier seront éditées, Bet on Soldier : Blood of Sahara et Bet On Soldier : Black-out On Saigon.

« Blood of Sahara » est la prèmiere suite de « Bet on Soldier ». L'action prend place en Afrique du Nord, 5 ans avant la mise en place du systeme BoS.

Troisième volet de la collection « Bet on Soldier », « Blackout on Saigon » a pour terrain d'opération le Vietnam.

- Speedball2

Speedball2 (www.speedball2.com) est produit en collaboration avec Frogster Interactive. Les développeurs ont travaillé d'arrache-pied pour en faire un jeu digne de son prédécesseur.

Speed Ball 2 est un remake du jeu des Bitmap Brothers, « Speedball 2 : Brutal Deluxe ». Originellement sorti sur Amiga en 1990, il a poursuivi son succès sur de nombreuses plate-formes : SEGA Master System et MegaDrive, Nintendo NES, gameBoy, GameBoy Advance, Pocket PC... Le Speedball est un jeu de sport futuriste reprenant plus ou moins les règles du handball ou quasiment tous les coups sont permis : les joueurs sont équipés d'armures et les contacts physiques sont nombreux et violents. Kylotonn a donc complètement retransmis l'esprit d'un match de Speedball en 3D dans un jeu très brutal et rapide, et dont le gameplay arcade se mêle à un rendu futuriste plutôt sympathique comportant un grand nombre d'effets. En plus de la refonte graphique du jeu, Speedball comporte, bien sur un mode multi-joueur en réseau local, mais aussi sur Internet. Un site multi-joueur est spécialement développé pour accueillir les joueurs et permettre l'organisation de tournois avec un classement mondial et une communauté active. Le but étant par la suite de faire de Speedball le PES du PC (Pro Evolution Soccer est un jeu de football ayant beaucoup de succès sur console notamment).

- En développement : Crusaders, Invasion of Constantinople

Crusaders, Invasion of Constantinople est un projet en développement. C'est un jeu d'action *vue à la troisième personne* se passant aux temps des croisades et reproduisant la prise de Constantinople.

Invasion of Constantinople est un jeu original entièrement crée par Kylotonn. Il propose aux joueurs de s'immerger dans des décors reproduits le plus fidèlement possible a partir des documents disponibles pour participer a la prise de Constantinople par les croises en 1204 lors de la quatrième croisade. Le principe de jeu sera un compromis entre le jeu de tir en vue subjective, un jeu plus type RPG (Jeu de rôles, avec un personnage dont les compétences évoluent au cours du jeu) et un jeu d'action *vue à la troisième personne*.

4 ORGANISATION

4-1 ORGANIGRAMME

Kylotonn Entertainment emploie 16 collaborateurs permanents et environ 30 travailleur indépendant. Comme nous l'avons vu précédemment, la société a été co-fondée par Roman Vincent et Yann Tambellini. Ils sont toujours à la tête de l'entreprise actuellement. L'organisation est séparée en 4 fonctions.

Une partie s'occupe de la programmation des jeux et des outils internes de la chaîne de production.

La partie Game Design assure la conception du fonctionnement du jeu et la conception des maps, c'est-à-dire les endroits visités par le joueur.

Une troisième partie est en charge de la conception graphique des décors, des modèles et des animations d'un jeu.

Enfin, une dernière partie gère la communication entre l'entreprise, les clients et les partenaires.

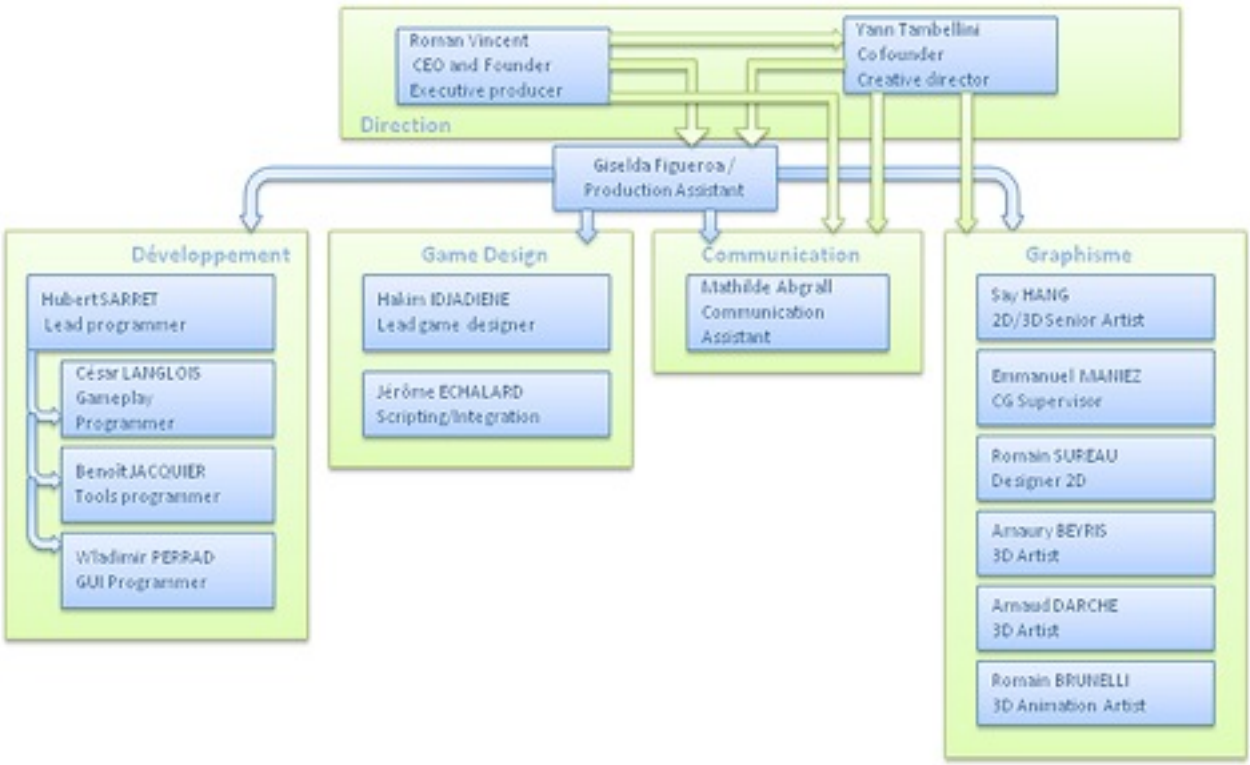


Diagramme de l'organisation hiérachique de Kylotonn

Sur le schéma ci-dessus, j'appartiens à la direction développement. Mon maître de stage est Hubert SARRET et Benoît JACQUIER me fournit le travail à effectuer.

4-2 STRATÉGIE

Traditionnellement, les producteurs de jeux vidéo sont relégués au rôle de simples prestataires de services selon un modèle très simple : L'Editeur fait développer le produit par une structure de développement externe, rémunérée pour cette prestation. Le financement de cette production est en fait une avance sur royalties de l'Editeur au Développeur. Ces royalties sont généralement calculées selon des méthodes alambiquées de telles sortes que le développeur doit réellement considérer les sommes allouées au développement comme forfaitaires et ce, quelque soit le succès du produit.

C'est en partant de ce constat, et aidé par le succès d'Ironstorm, puis de Bet on Soldier que Kylotonn a décidé de créer une structure de développement capable de mettre en oeuvre une nouvelle stratégie en se positionnant en aval de la chaîne comme éditeur-producteur. L'ambition est de bénéficier des retombées financières générées par les ventes des jeux développés par Kylotonn durant les années qui suivent la sortie d'un titre.

Kylotonn s'adresse donc a des distributeurs, territoire par territoire. Les interlocuteurs et clients comptent parmi les acteurs majeurs de la distribution en Europe et dans le monde. Ils ont a charge la fabrication, la logistique, le marketing et la vente des produits développés. Kylotonn conserve toute la propriété intellectuelle de ses titres et ne licencie que les droits de distribution. Le prix de vente des droits de distribution n'est pas calculé en fonction des coûts de production mais est fonction d'un minimum garanti.

Ce modèle présente l'avantage de faire remonter les recettes directement vers Kylotonn. De plus, ce modèle n'implique pas de « cross latéralisation » des coûts et des recettes ; les royalties proviendront des marches a succès sans être diluées par les marches sur lesquels le produit ne fonctionnerait pas. Bien sur, ce modèle n'empêche pas la société de continuer une activité plus typée prestataire de services moins contraignante.

4-3 PARTENAIRES

Pour améliorer ses capacités de production, Kylotonn travaille en relation avec plusieurs partenaires qui apportent leur expertise. Trois de ces partenaires sont particulièrement proches : Roman Vincent Productions, AGEIA et nVIDIA.

RVP étant la société au sein de laquelle l'équipe de Kylotonn a développe ses premiers titres avant de gagner son indépendance, beaucoup des outils utilisés pour les développements de Kylotonn lui appartiennent, comme le moteur 3D (KTEngine) ainsi que l'éditeur de niveaux et le moteur de jeu. Roman Vincent Productions octroie à Kylotonn une licence illimitée pour ces technologies propriétaires.

AGEIA, inventeur du premier processeur physique (PPU) a accorde a Kylotonn une licence illimitée et gratuite de sa technologie (matériel et logiciel).

La société est également partenaire officiel de nVIDIA Corporation, premier constructeur mondial de cartes vidéo. nVIDIA met a disposition des moyens matériels et humains conséquents.

5 PLATE-FORME PLAYALL

Kylotonn est membre du Pole de compétitivité Ile de France CAP DIGITAL et participe ainsi avec de nombreux partenaires à des projets d'envergure comme PlayAll. Ce projet regroupe neuf PME et cinq établissements publics, chacun pour leur spécialité. Le but de cette coopération est de créer et proposer sur le marché. Le produit final sera un moteur de jeu modulable aisément entre plusieurs types de jeux et compatible avec de nombreux supports de jeu (PC, PS3, XBOX 360, Wii, DS, PSP, ...). Puisque Kylotonn apporte a ce projet entre autre l'expertise de son moteur 3D, j'ai donc participé à PlayAll au sein de l'équipe de développement du moteur de jeux PlayAll.

5-1 PROJET PLAYALL

PlayAll est un projet structurant et collaboratif pour le secteur du Jeu Vidéo français.

Il a pour objet la fusion des technologies de plusieurs studios et le développement d'un moteur de création de jeu commun, ainsi qu'un standard technologique partagé avec un certain nombre de sociétés d'*intergiciel* et de laboratoires de recherche innovants. Le but est de mutualiser les connaissances des différents acteurs afin de « s'ouvrir à la collaboration et à l'entraide entre studios français pour franchir ensemble les ruptures technologiques de demain ».

Cette fédération d'entreprises complémentaires a élaborée une stratégie de Recherche & Développement collaborative, lui permettant de se placer dans une position économique gagnante pour les 5 prochaines années et les évolutions technologiques futures.

PlayAll permettra de concentrer au maximum les budgets de développement sur la création de contenu, coeur de métier et domaine d'excellence de Kylotonn.

Au delà des participants au projet, PlayAll s'adresse également à l'ensemble de la communauté du jeu vidéo française, qui pourra en exploiter le résultat à des conditions d'accès privilégiés, notamment la gratuité de la licence pour le premier titre réalisé.

Le projet PlayAll a débuté depuis septembre 2007 et fait travailler 40 ingénieurs pendant 2 ans. Il est labellisé par le pôle de compétitivité parisien Cap Digital, et financé par la Direction Générale des Entreprises, la Région Ile de France, les départements de Paris, des Hauts de Seine et du Val de Marne.

Les membres PlayAll sont listés en annexe de ce document.

3EME PARTIE

LE STAGE

1 INTÉGRATION À L'ENTREPRISE

1-1 POSITION DANS L'ENTREPRISE

L'équipe de développement de Kylotonn travaille donc à la fois sur les jeux en cours de réalisation mais aussi sur l'évolution du moteur 3D. C'est pourquoi, comme nous l'avons vu dans la partie organisation, l'équipe se divise en deux parties : les programmeurs moteurs et les programmeurs *gameplay*. Lors de mon arrivé à Kylotonn, on m'a donc proposé de travailler au choix sur le développement du *gameplay* de Crusaders, ou bien sur l'implémentation de nouvelles fonctionnalités du moteur PlayAll, c'est-à-dire en *DirectX* 9, et en *HLSL* pour la programmation des *shaders*.

C'est donc dans ce dernier domaine de l'industrie du jeu que va porter mon étude : l'intégration des *shaders* et donc d'effets, au sein du moteur PlayAll, dans le but bien sur d'apporter, fonction des besoins, de nouveaux éléments au rendu d'un jeu, mais aussi d'améliorer celui-ci de sorte que les graphismes soient les plus complets et réalistes possibles. Durant ce stage, j'ai donc rejoins l'équipe des programmeurs moteur de Kylotonn, et travaillé en tant que développeur graphique et R&D (pour l'apport de nouvelles technologies) avec toute l'équipe PlayAll.

Dans mon travail, j'ai souvent travaillé seul à la conception et à la réalisation de mes différents projets, en étant toutefois aidé et orienté par le responsable de développement du moteur 3D, Benoît JACQUIER. J'ai aussi pu, à plusieurs reprises, travailler avec lui à la conception de différents points du moteur. J'ai cependant eu tout au long du stage, différents objectifs secondaires.

J'ai pu travailler aux côtés de professionnels mais aussi d'autres stagiaires avec qui j'ai échangé des connaissances et des conseils, l'esprit d'entraide et de co-apprentissage étant très présent dans l'entreprise.

1-2 DÉCOUVERTE DE L'ENVIRONNEMENT DE PROGRAMMATION

La première semaine a consisté en une découverte de l'application existante par la lecture de la documentation et l'exploration du code source et me familiariser avec les différents outils et méthodes de programmation, d'organisation et de travail mises en place par la société. Le développement s'effectue en C++ sous Microsoft Visual Studio, comme la quasi-totalité des jeux commerciaux pour machine PC. Je ne connaissais qu'à peine l'IDE de Microsoft, cependant son appropriation n'a pas été très difficile.

Le code source est relativement clair et bien structuré, ce qui rend sa compréhension assez aisée. Toutefois, il est évident que sur une application de cette importance, son évolution par rapport à sa première version et le nombre de programmeurs différents ayant contribué à son développement, certaines parties du code et de sa structure sont difficile à appréhender. Il m'a donc fallu plusieurs semaines avant de bien comprendre le fonctionnement des différents composants et certains me sont encore inconnus.

De plus la structure du code est très évoluée, utilisant énormément de classes, d'héritage, de polymorphisme... Il est facile de comprendre ce que doit faire chaque composant et où insérer les nouvelles fonctionnalités. Cette structure idéale est toutefois altérée par les modifications successives des classes qui les dégénèrent en leur attribuant des rôles qui ne sont pas les leurs. Le développement est donc une succession d'ajouts et de modifications permettant de garder une structure claire et évolutive.

Il est découpé en de nombreux composants et couches permettant la séparation des rôles et l'abstraction des implémentations. Le futur du moteur PlayAll étant tourné vers le multiplate-forme, il est important d'avoir un tronc commun complet et puissant pouvant tourner sur tous supports et permettant la généralisation des comportements et donc la factorisation du code. Notamment, l'utilisation de *DirectX* 9 et de Windows sont encapsulés dans des composants dédiés. Il serait cependant difficile de remplacer directement ces composants par d'autres, l'implémentation du coeur de l'application s'inspirant fortement de *DirectX*.

La structure est donc vouée à beaucoup évoluer pour s'adapter aux nouvelles technologies ainsi qu'aux nouvelles utilisations qui vont en être faites.

1-3 ENVIRONNEMENT ET OUTILS UTILISÉS

Durant mon stage, j'ai eu l'occasion d'utiliser des logiciels dont je m'étais plus ou moins déjà servi et bien sûr beaucoup d'autres nouveaux, souvent utilisés dans le monde professionnel du jeu vidéo.

- Le moteur de jeux PlayAll

Le moteur de jeux PlayAll permet de gérer en particulier tout le rendu graphique ainsi que les périphériques d'entrée et les collisions. Cette couche de bas niveau peut être réutilisée dans la conception de tous types de jeux, voire même dans d'autres applications de rendu graphique. Il est basé sur *DirectX* pour la plate-forme PC et est suffisamment puissant pour gérer l'utilisation de technologies récentes comme les *shaders*. Celui-ci comporte de nombreux effets, tels que l'utilisation des techniques de *rendus pré-calculés*, la gestion du *rendu différé* ou encore la génération d'ombres dynamiques. Le moteur comporte aussi un ensemble d'outils (viewer, exporter, toolkits,...) qui sont autant de fonctionnalités permettant entre autre de manipuler et visualiser des objets de type *Mesh* contenant des scènes 3D, celle-ci représentée par un arbre dans lequel chaque entité est un ensemble de nœuds. En tant que développeur moteur, c'est donc directement dans ce moteur que j'ai eu à réaliser l'implémentation des différentes techniques dont j'ai fait l'étude durant ce stage.

- Microsoft Visual Studio .NET 2005

Kylotonn dispose de la dernière version de l'environnement de développement Visual Studio de Microsoft. J'avais déjà pu utiliser ce logiciel dans le cadre de mon stage précédent en VB.NET. Celui-ci est très complet et m'a permis dans le cadre du développement, de programmer sous le langage C++.

- Tortoise SVN

La SVN est un outil permettant de gérer les différentes versions de fichiers. Nous nous en servons essentiellement pour gérer l'ensemble des fichiers source du moteur et des autres applications en développement. Il permet lui aussi d'en maintenir constamment des versions correctes, d'y revenir dans le cas d'une erreur, ainsi que de comparer les différences avec les copies en cours de modification.

- 3D Studio Max 8

Logiciel de modélisation et d'animation 3D, c'est une référence en infographie. Les graphistes travaillent dessus pour créer les scènes et tous les modèles 3D utilisés par la suite dans les jeux, utilisant l'exporter du moteur pour en permettre la manipulation.

- Langages de *Shaders* utilisés

Le moteur de jeux PlayAll et l'ensemble de ses fonctionnalités sont implémentées en C++. Dans le cadre de l'étude et du développement de *vertex shader* et *pixel shader*, le langage assembleur ainsi que le langage *HLSL* ont aussi été utilisés.

- Communication

La bonne communication est un des atouts de Kylotonn. Il n'y a pas de formalisme particulier entre les membres du personnel, la communication se fait donc de manière simple et naturelle. De ce fait mon intégration au sein de l'équipe fut plus aisée et je pense qu'il m'a fallu assez peu de temps pour me familiariser avec ses membres. En confrontant nos expériences avec d'autres stagiaires de ma formation, il semble que ce type de relation simple et franche entre les membres, se retrouve assez souvent et c'est une chose que j'apprécie énormément dans ce milieu.

- Gestion et partage des sources et des ressources

La synchronisation des versions et l'intégrité des sources et des données des différents projets sont gérées à l'aide du logiciel Tortoise SVN. Son utilisation, s'avère essentielle pour permettre à plusieurs membres de l'équipe de travailler avec des versions à jour. Le principe est de récupérer en local, sur sa machine, les fichiers présents sur le serveur en en faisant une copie permettant ainsi la modification du projet, sans pénaliser les autres à cause de parties de code non approuvé. En cas d'erreur, le SVN permet de récupérer à tout moment des versions antérieures de chacun des fichiers de la base de donnée. Je n'avais pas encore utilisé de tel outil durant ma formation mais sa prise en main, grâce à une interface du type Explorateur Windows, se fait très aisément.

- Méthodes de programmation

Avant de se plonger directement dans l'écriture de code, et afin de trouver la méthode la plus simple et la plus efficace en terme de performance, il s'est avéré judicieux de bien analyser les problèmes donnés, lors de la phase initiale de chaque projet. Cette première partie d'analyse est une phase importante dans une étude, assurant le fait d'arriver le plus vite au résultat souhaité et attendu, car elle permet un véritable débat en amont du travail.

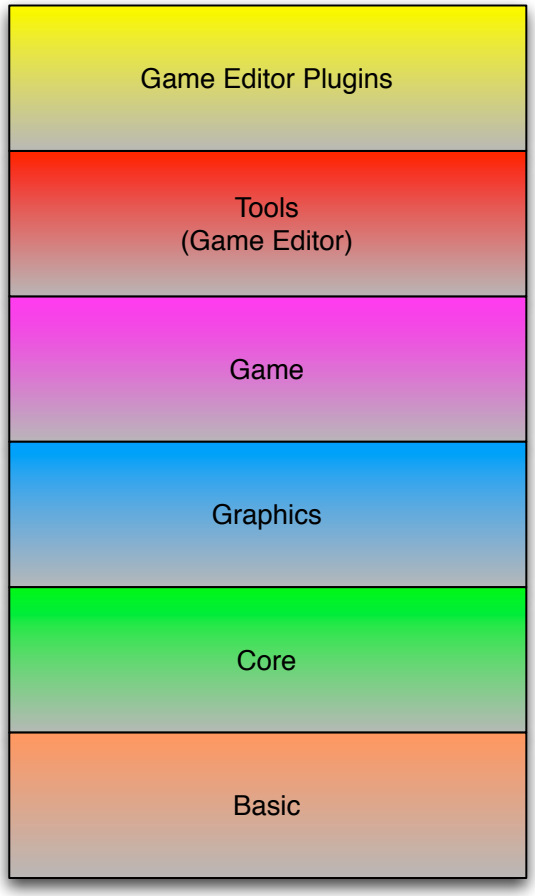
Afin de produire du code utilisable que l'on puisse intégrer à la production, il a fallu que je m'adapte à quelques règles et conventions d'écriture, nécessaires à la production d'un code clair et lisible. Il faut éviter par exemple les commentaires en abondance, ceux-ci n'aidant pas toujours à la lisibilité, sont plutôt utilisés pour expliquer un détail technique particulier ou une astuce de programmation.

Les autres règles et conventions permettent de garder une cohérence dans l'ensemble du projet. On retrouve ainsi, au niveau C++, les classes sont spécifiées dans des espaces de nomages relatifs à l'architecture du moteur PlayAll. Les variables on leur propre syntaxe d'utilisation, commune à tous les developpeurs travaillant à PlayAll. Toujours de façon à mettre en avant la compréhension du code, les variables auront des noms explicites en anglais, langue universelle de programmation. Enfin l'indentation et la présentation du code ont aussi toute leur importance.

Après une courte période d'adaptation, j'ai facilement intégré ces méthodes au sein de ma démarche de programmation, puisqu'on se rend vite compte que c'est le meilleur moyen d'écrire du code compréhensible et réutilisable.

2 MOTEUR PLAYALL

2-1 ARCHITECTURE DU MOTEUR PLAYALL



La plate-forme PlayAll est composée de différents couches (ou Layers). Ces couches sont numérotes de façon ascendante, 0 pour le plus bas niveau (le niveau Basic). Des sources d'une couche inférieure ne doivent jamais inclure des *fichiers d'en-têtes* de couches supérieurs, par exemple, les *fichiers sources* du layer 0 ne doivent en AUCUN CAS inclure des *fichiers d'en-têtes* de couches supérieurs.

On retrouve alors dans ces couches la couche Basic regroupant notamment la déclaration de tous les types de base, ainsi que des macros de code couramment utilisées. Ensuite, nous avons la couche Core qui implémente toute la gestion utilitaire d'utilisation des pointeurs managés, des erreurs, des logs, des fichiers. La couche Graphics ajoute toutes les fonctionnalités liés à l'affichage et les effets graphiques. Puis la couche Game implémente les composants du jeu, tels que les déplacements, la gestion des collisions et de la physique du jeu, ou encore les animations. La couche Tools et Game Editor Plugins permettent d'aider à créer plus facilement un jeu au plus haut niveau de réalisation.

J'ai travaillé au niveau de la couche Graphics qui gère tous les affichages et rendu 3D du moteur.

2-2 GESTION DES SHADERS PLAYALL

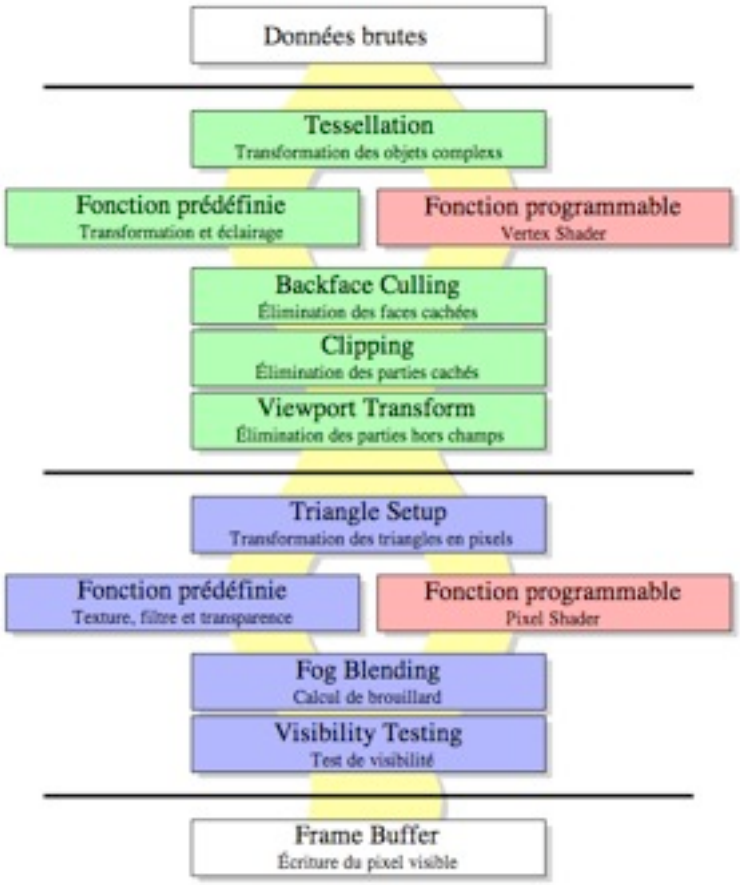
Au cours de mon stage, il m'est couramment arrivé d'avoir à travailler et modifier l'affichage graphique. Pour cela, il m'a fallu comprendre ce qu'est un *shader* et comment on l'utilise. Ce point que nous allons détailler dans cette partie permettra de mieux introduire mon travail par la suite.

Un *shader* est un programme qui permet de modifer le traitement appliqué aux vertices ou aux fragments lors du processus de rendu. Les shaders peuvent se diviser en deux groupes : les *vertex shaders* permettent de modifer les informations relatives à chaque vertex et les *fragments (ou pixel) shaders* permettent de modifier celles de chaque fragment.

L'intérêt tiré des *shaders* est tout d'abord la palette d'effets graphiques pouvant être crées (notamment grâce aux *pixel shaders*). Le second interêt, non négligeable, est le fait que le programme soit exécuté non pas par le *CPU* mais par le *GPU*, qui est lui spécialisé dans le traitement de données en 3D (vecteurs et matrices). Le *CPU* est donc libre de faire autre chose pendant que le programme s'exécute grâce au *GPU* de manière plus rapide.

La programmation de *shaders* se fait par le biais de 3 langages au choix :
- Le Cg (C for graphics), développé par nVidia, est le plus ancien et le plus utilisé : c'est celui que nous avons jugé le plus abordable. Il est compatible avec DirectX et OpenGL.
- Le *HLSL*, développé par Microsoft, est spécifque à Direct.
- Le GLSL, spécifque à OpenGL.

Voici le système du processus de rendu couramment définit.



Processus d'affichage de la carte graphique

Le langage des *shader* est compilé non pas à l'avance mais à l'exécution du programme. Ceci permet une compilation optimale en fonction du profile de la carte graphique disponible.

Un *vertex shader* est exécuté une fois pour chaque vertex rendu. Un *fragment shader* est exécuté une fois pour chaque fragment de chaque polygone rendu. Il devient donc évident que ces programmes doivent être le plus léger possible (surtout les *fragments shaders*) car ils s'exécutent plusieurs centaines, milliers, voire millions de fois par polygone et par *frame*. Néanmoins, l'avantage du traitement par la carte graphique et la palette d'effets disponibles ne sont pas non plus négligeables. Un compromis est à faire.

3 TRAVAIL EFFECTUÉ

Dans cette partie, je vais présenter la majorité de mon travail effectué tout au long de mon stage chez Kylotonn pour le moteur de jeu PlayAll. Chaque étude sera présentée de façon à comprendre quel en était le cadre, les objectifs, suivi de la présentation des solutions mises en places. Enfin et pour chaque partie des exemples des résultats obtenus.

3-1 ANIMATED MATERIAL

Introduction :

Ma toute première tâche au sein de l'entreprise a été de créer et rajouter un projet au sein de la solution du moteur PlayAll. Ce projet nommé “Animated Material” qui montre comment créer, utiliser et appliquer simplement des matériaux sur des objets qui sont :

- de type rectangles (ou quads) à deux dimensions affichés sur l'écran.

- de type *Mesh*. Kylotonn a développé un plugin sur 3DS Max permettant l'export d'une scène vers un fichier d'extension KT3 et contenant des informations sur les *Meshs*. Ces fichiers spécialement crées seront alors compatible avec le moteur 3D pour être chargé dans la scène du programme. Les *Meshs* contiennent des objets de type polygonaux (texturés, skinnés, bump mappés ..), les animations, cameras, lampes, boites englobantes. Le tout est structuré sous la forme de scène 3D.

Ce premier travail m'a permis en même temps d'étudier une partie des classes du moteur PlayAll. Ainsi, j'ai pu regarder à cette occasion :

- comment générer un projet pour l'inclure dans la solution du moteur PlayAll.

Afin de réduire au minimum les problèmes de gestion des différents projets PlayAll liés à la diversité des plate-formes ciblées et des IDE utilisés, certains outils vont devoir être obligatoirement utilisés.

La problématique principale est la suivante :

Chaque plate-forme va nécessiter pour un module donné des fichiers de projets distincts, et ce même pour les modules SP2 multiplate-formes. Par exemple, la compilation sur PS3 et X360 s'opèrent toutes les deux sous Visual Studio mais utilisent des plugins différents et la compilation sur Wii se fera sous CodeWarrior. Le problème intervient lorsqu'il s'agit de maintenir tous ces projets dans un même état synchronisé, que ce soit au niveau de la configuration mais surtout de la liste des fichiers.

Par exemple, si une personne travaille sur la solution PlayAll pour Win32 rajoute un fichier dans le projet Core, il faut que le même fichier soit rajouté dans les projets PS3, X360, etc.

La solution retenue est basée sur l'utilitaire premake qui nous permet de générer des fichiers de projets pour les différents IDE à partir d'un fichier de description générique.

- comment est structuré le moteur de jeu et regarder des classes de l'architecture du moteur. Par exemple, la gestion des pointeurs intelligents (StrongPtr et WeakPtr), des erreurs (classes ErrorCode), des fichiers et des systèmes de log.

- comment initialiser une nouvelle fenêtre graphique contenant une scène vide, ajouter des lumières et spécifier la position de la caméra dans la scène, ajouter des *Meshs*, voir que la scène gère les objets selon une arborescence de noeuds.

- comment sont gérés et utilisés les matériaux. Le MaterialManager s'occupe de garder et créer les instances uniques des matériaux et de leur description. Ces matériaux sont eux mêmes crées soit à la main dans le code, ou bien à partir de fichiers enregistrés sur le disque. Les matériaux et descripteur de matériaux possèdent un certain format et seront lus par l'intermédiaire des classes WrapperLua qui se chargeront de récupérer les propriétés tels que les Macros que l'on va appliquer au fichier de *Shader* fx spécifié dans le champ "Shader" d'une description du SchaderDesc. Le fichier *Shader* fx contient le programme *shader* écrit en *HLSL* et décrit quels seront les transformations appliquées au *vertex shader* puis au *pixel shader*. Des variables sont spécifies et utilisées par le *vertex shader* et le *pixel shader*, il est alors possible de les rendre dynamique en les ajoutant dans le champ "ShaderParams" du SchaderDesc décrivant le type de la variable (Float, Vector, Matrix, Texture2D, Texture3D, TextureCubeMap, VectorArray) et initialisées dans le champ "ParamDatas" du fichier du matériel. Les variables sont dynamiques dans le sens où l'on peut initialiser la variable par une fonction qui sera alors exécutée à chaque *frame* lors du rendu et mettra la variable à jour tel que la fonction le décrit.

- comment fonctionne le système de rendu. En modifiant les fichiers *shader* de type *HLSL*, j'ai découvert les différents tampons mis en place pour améliorer les performances le l'illumination d'une scène 3D.

Implémentation :

Voici ci dessous la réalisation finale

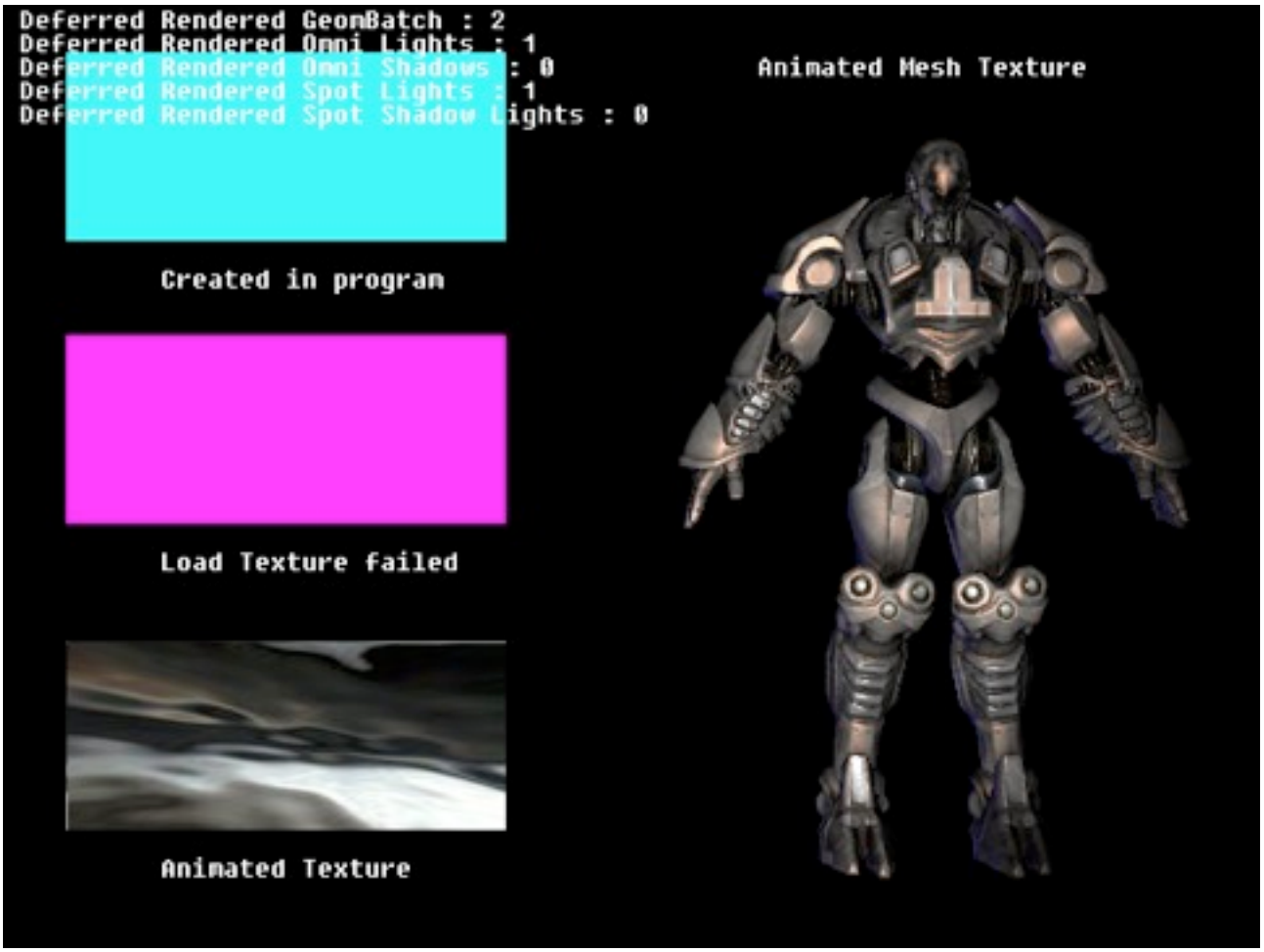


Photo d'écran du "Sample" animé

Dans ce "Sample", nous montrons à l'utilisateur comment sont chargées et utilisés les matériaux et leurs textures. En haut à gauche, un matériel a été crée à partir d'un code écrit en dur dans le programme. En bas à gauche, un autre matériel animé, mais cette fois-ci crée à partir d'un fichier lua. Au milieu à gauche, le matériel a été crée, mais la texture est manquante et n'a pas été trouvé volontairement dans les fichiers, une couleur rose est alors affichée à la place de la texture. A droite, nous plaquons une texture sur un *Mesh*.

L'animation n'est pas visible sur ce simple exemple, mais il faut savoir que tous les matériels sont animés en continue, c'est-à-dire que ce "Sample" montre que le moteur PlayAll offre la possibilité de pouvoir changer à chaque *frame* du rendu des propriétés sur les matériaux.

3-2 MODIFICATION DES TESTS GRAPHIQUES

Introduction :

Le moteur dispose de nombreux tests unitaires qui sont automatiquement lancés lors de la recompilation du projet. Ces tests sont très utiles afin de s'assurer que des modules fonctionnent correctement dans les circonstances décrites par les procédures de test.

Certains tests portent sur la couche graphique s'occupent de vérifier des propriétés lors du chargement de fichiers KT3 tels que :

- les positions données dans 3DS Max sont effectivement les mêmes sur la scène.

- lors du changement de position d'un noeud, alors les fils de ce noeud changent aussi de position conformément à leur parent.

- les propriétés sur les *vertex buffer* et *index buffer* sont respectées.

- les positions des animations de *Mesh* coïncident avec les mêmes positions dans le temps sur 3DS Max.

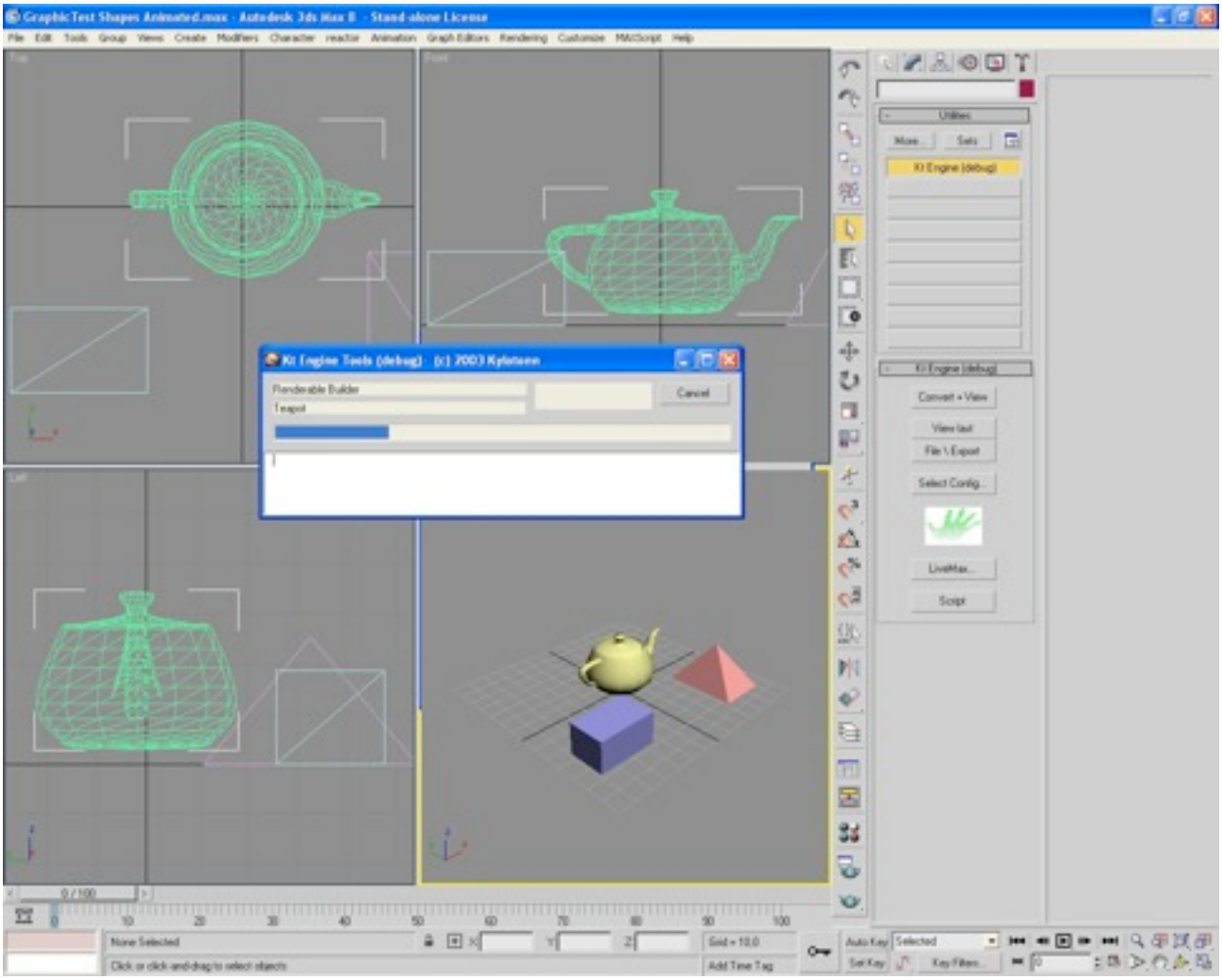
Parallèlement à cela, les *Mesh* qui sont chargés dans les procédures de test proviennent d'une base commune de fichiers KT3.

Développement :

Mon travail ici à été de créer de nouvelles scènes sur 3DS Max très légères (ne comportant pas beaucoup de polygones) qui seront exportés en fichiers KT3 et utilisés dans les procédures de tests.

Cela permettra alors d'avoir définitivement des *Meshs* KT3 qui seront juste utilisés spécifiquement pour les tests et resterons inchangés jusqu'à ce que l'on modifie ou ajoute de nouveaux tests. En effet, les anciens fichiers KT3 risqueraient d'évoluer, car il n'ont en fait pas vocation à être utilisés dans les procédures de tests, mais plutôt à être utilisés et modifiés voir supprimés dans le moteur de PlayAll, ce qui à long terme pourraient rendre les tests défectueux non pas parce que le code est faux, mais parce que les Meshs ont changés.

Ainsi, ces modifications que j'ai apportés m'ont permises de découvrir comment sont gérés les procédures de tests, mais aussi de voir que PlayAll dispose d'une bibliothèque d'objets 3D enregistrés sous 3DS Max dans un répertoire SVN différent du répertoire des sources du moteur. J'ai, à cette occasion, j'ai modifié les scripts qui permettent automatiquement d'exporter des scènes au format 3DS Max en fichiers KT3 ainsi qu'un autre script permettant directement de recopier les fichiers KT3 vers les bons emplacements du SVN du moteur. Ce travail m'aura aussi permis de découvrir l'environnement de 3DS Max, car j'ai du moi-même refaire de nouvelles scènes 3D comportant des animations et une hiérarchie d'objet pour être conforme aux anciennes procédures de tests. C'est d'ailleurs l'utilisation de 3DS Max qui m'a pris le plus de temps, étant donné que je n'ai pratiquement aucune connaissance en modélisation.



Processus d'export des Mesh sur 3DS Max à l'aide du plugin de Kylotonn

Ici, nous pouvons voir une scène composée d'objets simple est en train d'être exportée par le plugin vers un fichier KT3.

Au passage, j'en ai profité pour modifier le plugin d'export de 3DS Max afin de prendre en compte l'intensité lumineuse d'une lumière lors de l'export d'une scène dans un fichier KT3.

J'ai ensuite pris en compte ces changements au chargement d'une scène dans le moteur de jeu PlayAll.

3-3 GESTION DE L'AFFICHAGE DU TEXTE À L'ÉCRAN

Introduction :

L'intérêt de ce travail a été de rendre l'affichage du texte à l'écran compatible avec toutes les plate-formes de jeux.

En effet, l'affichage des fontes à l'écran n'était jusqu'alors géré que par des primitives *DirectX* compatibles qu'avec les plate-formes PC et XBox360. Les développeurs qui travaillent sur les autres plate-formes ne peuvent pas afficher du texte, celui-ci utilisé en particulier pour pouvoir écrire des informations utiles relatives à une scène.

Le fait de modifier l'affichage des fontes était aussi l'occasion de revoir son implémentation et modifier son utilisation au sein du moteur de jeux. Au lieu d'utiliser les primitives *DirectX* pour effectuer un simple affichage sur l'écran, voici les nouvelles fonctionnalités apportées :

- Les Fontes : stokage des données

Les fontes sont crées à partir d'un fichier au format XML qui fait correspondre pour chaque caractère son espacement, sa taille et sa représentation dans la texture. La primitive *DirectX* n'affichait alors qu'un seul type de fonte par défaut. Il est alors maintenant possible de créer différents types de fontes personnalisées.

- FontRenderer : la gestion des Fontes

Pour rendre une fonte à l'écran, il faut instancier un objet FontRenderer. Un objet FontRenderer contient une structure de données permettant de mettre en tampon les Fontes et le texte qu'il devra rendre à l'écran.

Principe :

Le principe se décrit en 3 étapes successives :

- On dit à la fonte de se dessiner en appliquant la fonction DrawString() dans un FontRenderer à une certaine position, couleur et taille de police à l'écran en précisant le texte à afficher.

- L'instance de FontRenderer va alors mémoriser le texte en ajoutant les *vextex buffer* et *index buffer* et de la texture associée au texte correspondant.

- A l'appel de la méthode FlushRenderer() de l'instance de FontRenderer, celui-ci va alors afficher à l'écran tout le texte qu'il a conservé en mémoire selon la fonte utilisée, la couleur et les *vertex buffer* et *index buffer* associés pour représenter le texte.

L'intérêt de passer par une gestion des fontes est de pouvoir mettre en tampon le texte et de l'afficher en une seule fois. De plus, il est possible de créer plusieurs instances de FontRenderer et de pouvoir accumuler du texte dans plusieurs tampons différents. Cela permet de différer le rendu du texte sur des canaux différents. Par exemple, on peut avec se système rendre du texte une fois à chaque *frame* et pourquoi pas un autre texte de la même fonte ou d'une autre fonte une fois toutes les 10 *frames*.

Exemple :

On crée une instance de fonte a partir du fichier descriptif DefaultFont.xml. La lecture du fichier chargera d'abord la texture correspondante pour tous les caractères.

Texture de Glyph générée et correspondante aux données de DefaultFont.xml

Ensuite, toutes les informations relatives pour chaque caractère sur les espacements, ses dimensions et ses coordonnées relatives dans la texture sont lues et sauvegardées en mémoire dans l'instance de la Fonte.

Lors du rendu de la Fonte à l'écran, des *index buffer* et *vertex buffer* seront accumulés décrivant tous les rectangles à afficher sur l'écran, relativement au matériel utilisé et à la cordonnée de la texture de la fonte.

Rendu des Fontes à l'écran :
En Haut : les fontes sont rendues sans la texture associée au glyph
En Bas : les fontes sont rendues avec la texture associée au glyph

Optimisation :

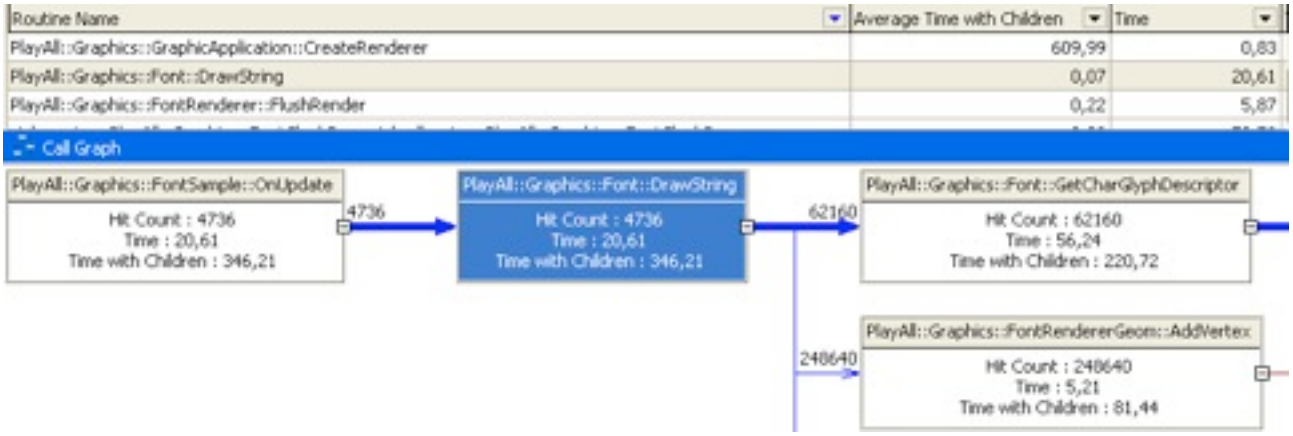
Lorsque l'affichage et la gestion des fontes était opérationnel, j'ai cherché à optimiser le temps d'affichage. En effet, l'affichage des fontes sera toujours utilisé pour afficher la *rapidité d'affichage* ou d'autres indications à l'écran. Il est alors nécessaire de réduire au mieux le temps.

Un moyen simple et rapide pour comparer les performances est de regarder la *rapidité d'affichage* affiché à l'écran et de comparer le chiffre avec d'autres exécutions. Cependant, cette méthode a plusieurs désavantages. En effet, la *rapidité d'affichage* change constamment et il faut estimer une *rapidité d'affichage* moyenne à vue d'oeil. De plus, la *rapidité d'affichage* ne varie qu'a chaque *frame* et il n'est pas commode de savoir quelle est la partie de code qui ralentit le processus, surtout que de petites optimisations améliorent parfois de l'ordre de la micro-seconde les exécutions et la *rapidité d'affichage* est un chiffre bien trop élevé pour y voir une différence. Mais aussi, il faut prendre en compte le fait que si l'on arrive à optimiser d'une *frame* le code à 100 *frames* par secondes par exemple, cela n'aura pas le même impact que d'améliorer une *frame* à 200 *frames* par secondes. La *rapidité d'affichage* ne donne pas d'indication linéaire sur ce que l'on optimise.

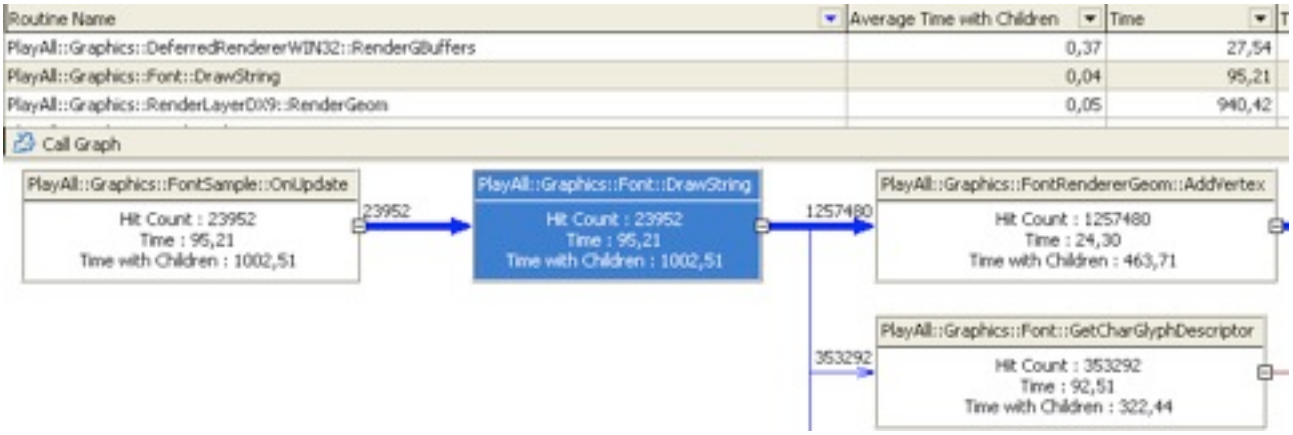
Toutes ces raisons mènent à chercher un autre moyen pour pouvoir étudier les performances d'affichages. Ainsi, je me suis servit du logiciel AQTime. On lance l'application sur AQTime pendant une période donnée. AQTime se charge ensuite de collecter les données sur l'exécution et nous donne très précisément (de l'ordre de la micro-seconde) dans un tableau le temps passé dans chaque fonction du code ainsi que le temps passé en additionnant tous les appels fils d'une fonction. Une vue nous permet de voir visuellement quels sont les fonctions qui prennent le plus de ressources en temps.

Lors de l'affichage des résultats, il n'était pas possible d'optimiser les fonctions d'affichages, car ce sont les fonctions propres à *DirectX*. Par contre, il m'était tout à fait possible de modifier ma structure de données et appels de fonctions. Une Fonte contenait à l'origine un tableau de hash (c'est-à-dire une map de la bibliothèque standard stl) reliant un caractère à sa structure de glyphe correspondante. A l'aide de AQTime, j'ai remarqué que la fonction la plus appelée et la plus lente était la recherche d'un caractère dans la table de hachage afin d'obtenir les informations sur sa structure de glyphe. De ce fait, j'ai changé la structure de données en remplaçant la table de hachage par un simple tableau (c'est-à-dire un vecteur de la bibliothèque stl) que je trie par caractère à l'initialisation de la fonte. J'ai crée un algorithme de recherche dichotomique pour récupérer en temps logarithmique la structure de glyphe correspondant à un caractère.

Utilisation de AQtime :



Gestion par table de hachage



Gestion par tableau

En exécutant un test de performance par AQTime, j'ai constaté que la fonction Drawstring ne mettait en moyenne plus que 0,04 mili-secondes à terminer avec la gestion par tableau contre 0,07 avec la gestion par table de hachage. Cela permet donc de gagner le double de mili-secondes à chaque appel à la fonction DrawString pour chaque affichage de texte.

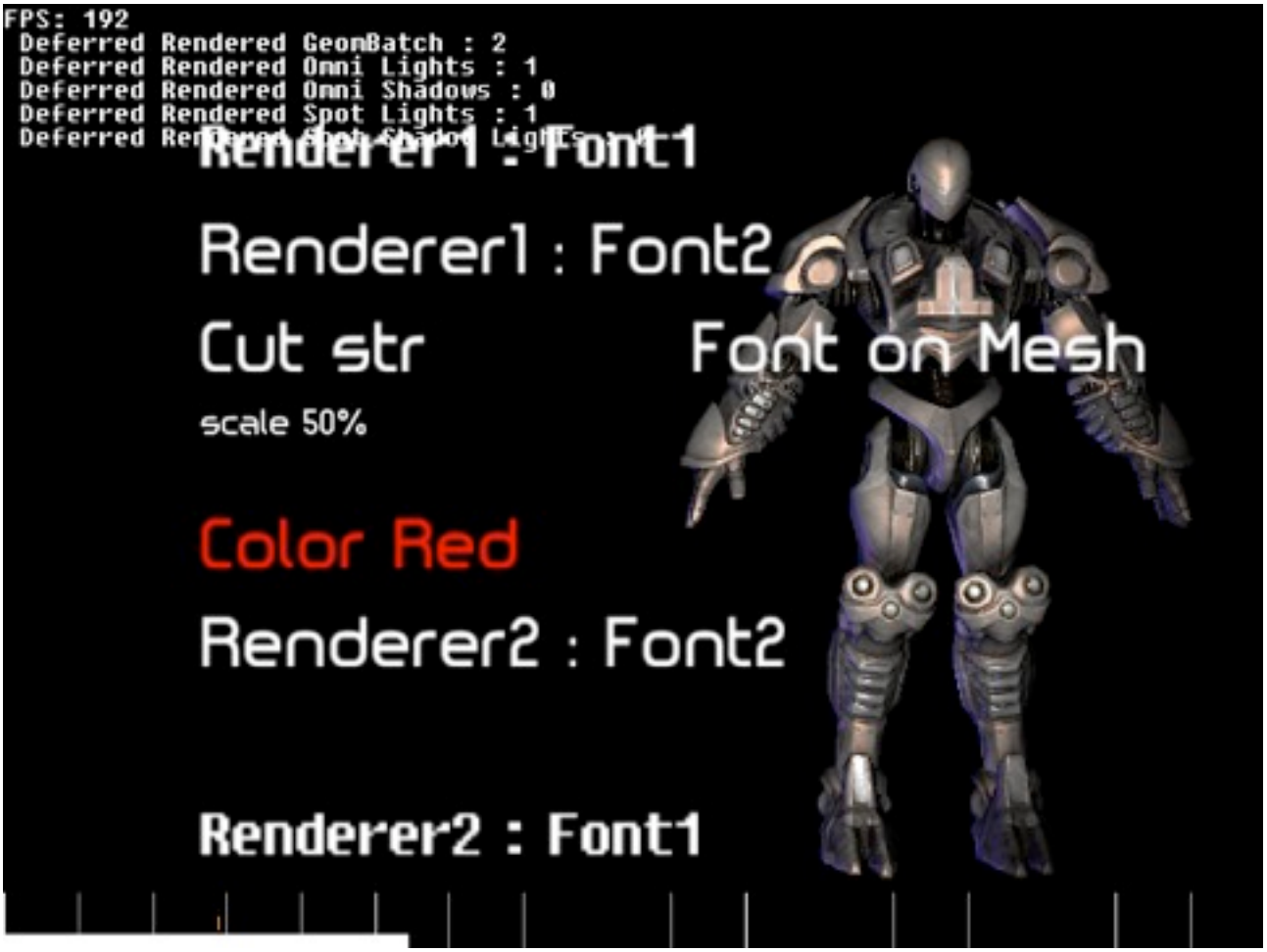
D'autres optimisations ont été apportées en nettoyant le code inutile ou mal utilisé et en rendant des fonctions souvent appelées en *inline*, c'est à que code est inséré directement dans le flux de code de la fonction appelante, se qui permet d'éviter d'empiler les arguments et codes de retour. Cette amélioration aura permis de gagner une micro-seconde.

Implémentation :

En parallèle au développement de la nouvelle gestion de Fontes, j'ai crée un exemple dans la solution du moteur de jeu qui montre comment se servir des Fontes et des FontRenderer.

Mais aussi, j'ai mis en place une série de tests unitaires qui permettent de s'assurer de leur bon fonctionnement actuel et futur lors d'éventuelles modifications. A la suite de cela, j'ai crée une page sur le Wiki de l'entreprise qui décrit les spécifications des classes et de leur utilisation.

Enfin, j'ai changé le système d'affichage du texte effectué via *DirectX* par le système des Fontes et rendu des Fontes. Ainsi, le texte peut être affiché sur toutes les plate-formes de jeu.



Capture d'écran du Sample qui manipule plusieurs types d'affichage de Fontes différentes

Voici une capture d'écran montrant différentes façons d'afficher une fonte. Cet exemple que j'ai conçut m'a été bien utile pour réaliser tous mes tests d'affichage. En haut à gauche, la police de caractère utilisée est gérée par l'ancien système utilisant les primitives d'affichage de *DirectX*. Au milieu de la scène, on peut voir différentes manières d'utiliser le nouveau système. Il est possible d'afficher une même Fonte sur plusieurs FontRenderer différents. Ainsi qu'un FontRenderer peut gérer l'affichage de plusieurs Fontes. Il est possible de redimensionner l'affichage d'une Fonte, de couper une le texte à une position et de changer de couleur en précisant ces données dans la fonction DrawString() de la Fonte.

3-4 ENREGISTREMENT DES FONTES PAR LE CONTENTPIPELINE

Introduction :

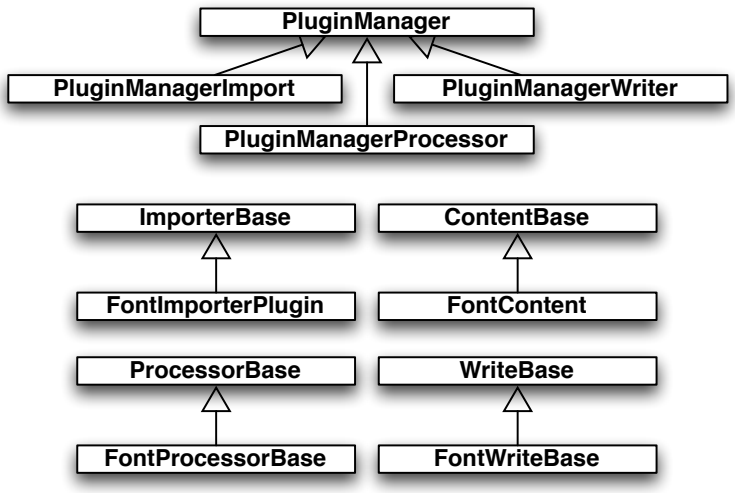
Dans le cas du chargement d'une fonte, le programme va tout d'abord lire le fichier XML ainsi que la texture correspondante au format d'image d'extension TGA. Cependant, seule la plate-forme Windows est capable de lire un fichier au format TGA. La seule manière pour pouvoir charger une fonte dans le moteur PlayAll pour une autre plate-forme est de convertir le fichier de format TGA en format DDS.

Le problème dans ce cas, c'est que ce travail est répétitif et il faudrait aussi changer toutes les descriptions de chaque fichier de fonte si on venait à modifier la lecture du fichier XML lors du chargement. C'est la raison pour laquelle PlayAll dispose d'un processus d'exportation générique de tous les fichiers données de la base vers la plate-forme.

L'avantage d'utiliser un tel processus est de pouvoir créer automatiquement les ressources à partir d'un fichier descriptif et de régler les problèmes énoncés précédemment.

Il y a aussi d'autres avantages à passer par un pipeline d'exportation, par exemple, il est possible de gérer différentes versions de fichier. De même que PlayAll réduit sensiblement le temps chargement, puisque les données sont alors directement lues et entièrement contrôlées par le moteur de jeu PlayAll.

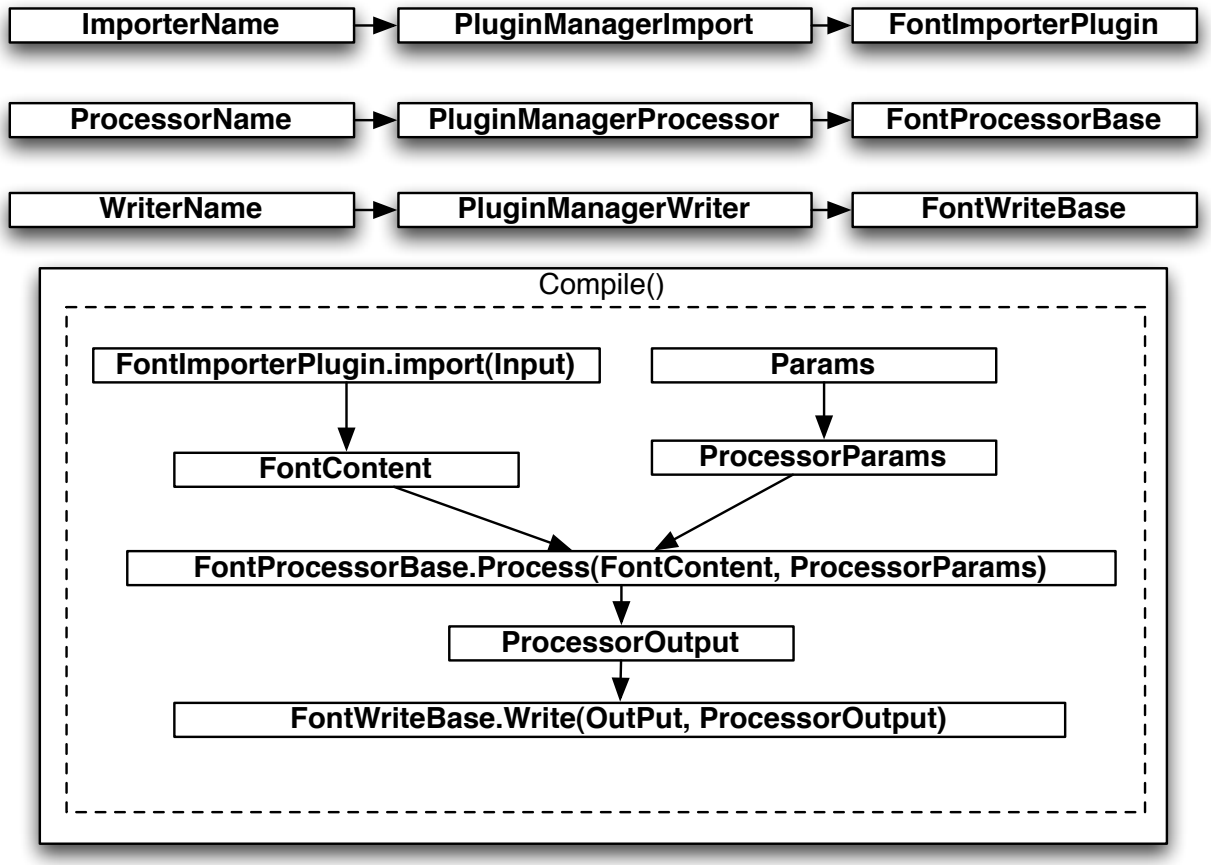
Le schéma du processus est le suivant :



Structure de donnée utilisée pour le ContentPipeline

Pour chaque type de ressources à exporter, il faut rajouter des plugins qui rentrent dans le processus d'exportation :
- Importer : ce composant va se charger de lire le fichier d'entrée pour le convertir en format intermédiaire.
- Format Intermédiaire : le format intermédiaire stocke en mémoire les informations nécessaires qui seront transmises au writter.
- Processeur : le processeur se charge de faire un traitement particulier sur les données.
- Writer : le writter va écrire dans un fichier binaire les données.

Processus d'export des fichiers :



Shéma explicatif pour l'export des données

Ensuite, la fonction Compile() va lancer le processus d'export des fichiers comme vue sur le schéma ci-dessus.

Ma tache à donc été de créer les 3 plugins permettant l'export des fontes à partir d'un fichier XML descriptif.

Pour cela, j'ai repris le code développé par les équipes de Kylotonn qui générait l'ancien format de fichier XML des fontes. J'ai fait ce long travail de retranscription du code dans la solution PlayAll du ContentPipeline.

Cependant, la politique de l'export de la fonte par le ContentPipeline n'est pas la même que celle utilisée par Kylotonn. En effet, auparavant, le fichier XML généré était complètement dépendant de la langue choisie et des données à générer. Maintenant, l'utilisateur passe au Content Pipeline le fichier d'entré XML requière les informations sur la fonte utilisée ainsi qu'une liste de caractère à enregistrer.

3-5 PROFONDEUR DE CHAMP

Introduction :

J'ai ajouté une nouvelle technique en temps réel de rendu au sein du moteur PlayAll : la profondeur de champ (ou Depth Of Field)

Cette technique est un effet visuel que l'on retrouve en photographie. La profondeur de champ désigne l'espace qui sera net lors de la prise de vue.

Par exemple : lorsque l'on fait la mise au point sur un sujet, un espace devant et derrière ce plan de mise au point détermine la profondeur de champ. Une profondeur de champ réduite (sujet net et arrière-plan flou) mettra le sujet en évidence. Une profondeur de champ maximale (avant-plan net et arrière-plan net) rendra une scène avec tous les détails, le réalisme et la profondeur. La profondeur de champ d'un plan de mise au point est plus grande derrière celui-ci, que devant.

Ainsi, on constate un effet de flou apparent sur la scène, selon la mise au point que l'on aura attribué. Bien que cet effet de flou peu être perçu comme une imperfection qui dégrade la qualité de l'image rendue à l'écran, il peut être aussi utilisé comme un outil qui permet de mettre en valeur des objets de la scène au détriment d'autres.

L'usage de la profondeur de champ améliore alors effectivement l'effet photoréaliste et ajoute une touche artistique au rendu de l'image.

Description de la technique :

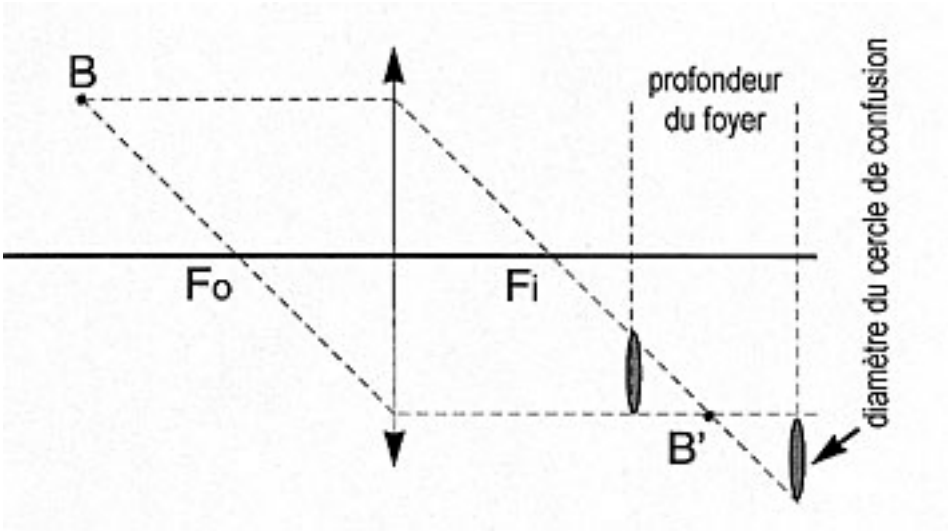
Nous allons ici décrire de façon la plus simple l'effet de la profondeur de champ observé en photographie.

- cercle de confusion

Il nous faut d'abord rappeler la définition d'un cercle de confusion avant de définir la profondeur de champ.

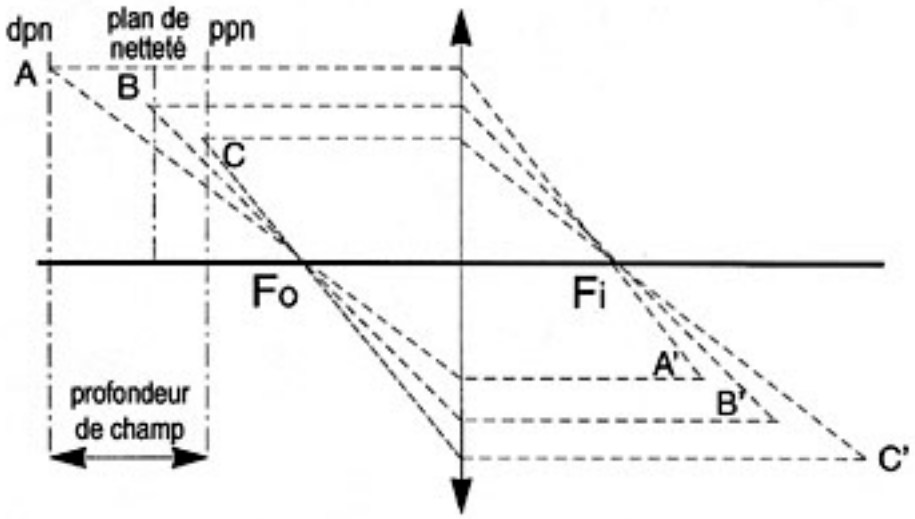
Les cercles de confusion sont les plus petits points placés l'un à côté de l'autre qu'il est possible de distinguer à l'oeil sur un négatif, ou plus généralement sur le support d'un appareil photographique. Le diamètre de ces cercles est variable en fonction de la taille d'un négatif et de l'observateur. Suivant les individus, on voit plus ou moins bien et la notion de netteté est légèrement différente pour chaque humain. En règle générale, on accepte un diamètre de 0,02 mm pour un négatif 24x36 pouces, un diamètre de 0,05 mm pour un négatif 6x6 pouces et un diamètre de 0,1 mm pour un négatif 4x5 pouces.

- Profondeur du foyer



Un objet (B) vu au travers de la lentille a son image inversée (B') de l'autre coté de cette lentille selon la distance focale f. (B') est le point à l'endroit précis ou se forme l'image mais on a en plus devant et derrière ce point une image moins précise mais toujours acceptée comme nette. Cette image est alors non plus un point mais un cercle légèrement flou. S'il ne dépasse pas le diamètre d'un cercle de confusion, il est considéré comme net. On obtient ainsi une zone entre le cercle devant le plan de netteté et celui derrière le plan. Cette zone où le photographe considère comme nette toute image s'y trouvant est appelée profondeur du foyer.

- Profondeur de champ



A l'inverse de la profondeur du foyer, on a une profondeur de champ dans laquelle on positionne l'objet à photographier. On peut considérer la position du négatif comme stable et précise dans le boîtier de l'appareil. Ainsi la profondeur du foyer n'est pas nécessaire puisque l'on considère nettes les images se trouvant entre les plans contenant les cercles de confusion, on peut alors retrouver les objets de ces images. Pour la facilité du dessin on remplace les cercles de confusion par des points (A' et C') et on recherche les objets de ces points.

Tous les points contenus entre A et C sont considérés comme nets sur la pellicule. On constate que le plan où se trouve le point B s'appelle plan de netteté.

Que le plan ou se trouve le point A s'appelle DPN (dernier plan net) et que le plan où se trouve le point C s'appelle PPN (premier plan net).

L'espace entre les points A et C s'appelle "profondeur de champ". La profondeur de champ c'est une zone de netteté qui s'étend en deçà et au delà du plan de netteté. Elle se repartit 1/3 devant et 2/3 derrière. Les objets situés en dehors de la profondeur de champ seront flous sur l'image.

- Le modèle des lentilles minces convergentes

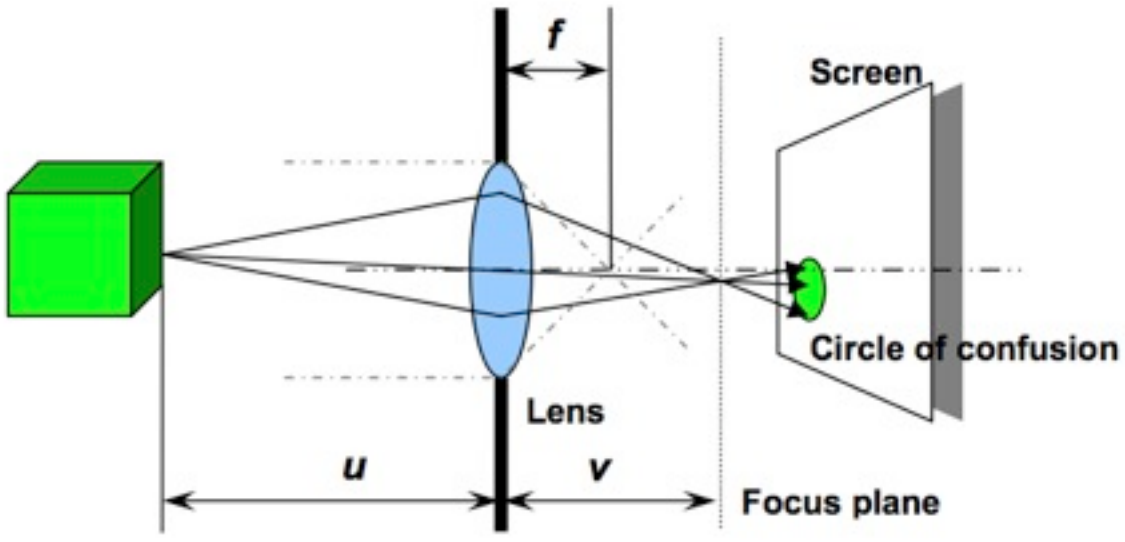
Dans le monde réel, les lentilles minces sont de dimension finie. Cela a comme conséquence qu'une partie d'une scène est plus ou moins nette si les objets sont situés dans les environs ou au-delà du plan focal. C'est ce que l'on a vu précédemment avec la profondeur de champ.

On considère donc que notre objectif s'assimile à une simple lentille mince convergente et possède une distance focale f finie. Une image dans le plan focal de distance v de la lentille est nette si l'objet observé se situe à une distance u de la lentille et que les variables f, u et v vérifient l'équation suivante :

$$\frac{1}{u} + \frac{1}{v} = \frac{1}{f}$$

La distance entre le plan focal et l'objet peut être exprimée de la manière suivante :

$$z_{focus} = u + v$$



De multiples rayons partant d'un point donné d'un objet vont passer à travers la lentille mince et forment un cône de lumière. Si l'objet observé est dans le plan foyer, tous les rayons vont converger en un seul et unique point sur l'image du plan focal. Par contre, si un point donné de l'objet de la scène n'est pas à côté de la distance focale, le cône de lumière va intercepter l'image du plan focal dans une zone en forme de section conique. Typiquement, la section conique forme un cercle et appelé cercle de confusion que l'on a vu précédemment.

Le cercle de confusion de diamètre b dépend de la distance du plan focal et de l'angle d'ouverture a de la lentille. Pour une focale f et de diamètre D d'une lentille mince donnée, la taille du cercle de confusion peu être calculée de la façon suivante :

$$b = |\frac{D * f * (z_{focus} - z)}{z_{focus} * (z - f)}|$$

avec

$$D = \frac{f}{a}$$

Tout cercle de confusion plus grand que le plus petit point que l'oeil humain peut discerner contribue alors à rendre flou l'image, ce que l'on appelle la profondeur de champ.

Mise en oeuvre :

i) Choix de la technique :

Il existe de multiples techniques qui sont utilisée pour simuler la profondeur de champ dans le rendu d'une scène. Il a donc tout d'abord fallut choisir la technique la plus appropriée.

Une des technique utilise le lancer de rayons distribués. Pour chaque point de l'image, de multiples rayons sont tirés à travers la lentille. En provenant d'un seul et unique point de l'image, ces rayons se rencontrent au seul et unique point de l'objet observé, si celui-ci est placé sur la focale de la lentille. Dans le cas contraire, les rayons sont alors éparpillés dans l'environnement, ce qui contribue à rendre le point flou. Cette technique est alors la plus réaliste, mais la plus coûteuse en terme de calculs. Ce ne sera donc pas la technique que nous utiliserons pour simuler la profondeur de champ en temps réel.

Une autre méthode consiste à accumuler des tampons d'images. Chaque image est rendue autour d'une position et direction légèrement différente que celles de base en composant avec l'ouverture d'angle de la lentille. Cette technique est moins complexe que le lancer de rayon, mais celle-ci est toute de fois coûteuse puisque qu'il faut rendre plusieurs images à la fois afin d'obtenir un effet visuel acceptable.

Une technique moins coûteuse et plus raisonnable en temps qu'alternative pour l'implémentation en temps réel est d'utiliser la méthode de *rendu pré-calculé*. Usuellement, cette méthode met en jeu deux passes de rendu. A la première passe, la scène est rendue tel quel avec des indications complémentaires tels que la profondeur de la scène (ou couramment appelé *tampon de profondeur*). A la seconde passe, un filtre est appliqué sur le résultat de la première passe pour rendre l'image floue et contribuer au rendu final.

Nous utiliserons clairement la troisième méthode, la moins onéreuse en puissance de calcul, et compatible avec le rendu en temps réel. Nous verrons plus tard que nous en utiliserons 3 passes de rendu au lieu des 2 nécessaires pour arriver finalement à 7 passes que nous justifions par la suite.

ii) Technique de rendu utilisant le *rendu pré-calculé* à deux passes

Avant de décrire la technique, il faut savoir que le *GPU* a la possibilité d'écrire des valeurs de pixel distinctes dans plusieurs tampons d'images simultanément, c'est ce que l'on appelle couramment le *MRT*. Il nous est alors possible de créer des images en mémoire tampon puis de les utiliser par la suite pour réaliser le rendu final.

La technique de *rendu pré-calculé* de la profondeur de champ utilise au minimum deux passes, c'est-à-dire qu'une image sera mise en tampon pour calculer les informations de la première passe sans être rendue directement à l'écran. A la deuxième passe, nous allons utiliser des informations de la première passe rajoutée avec de nouvelles pour faire le rendu final qui sera affiché à l'écran.

Voici les étapes du rendu de la profondeur de champ :

- On rend la scène telle quelle dans la première passe.

- Lors de la deuxième passe, on rend la scène entièrement floue. Avec les informations de la profondeur de la scène, nous calculons un coefficient compris entre 0 et 1 qui fera office d'interpoler la scène nette de la première passe avec la scène floue de la deuxième passe.

Pour rendre la scène floue, on effectue un flou gaussien. Cela consiste pour un pixel donné a prendre ces coordonnées et d'appliquer un poids gaussien entre la position d'un pixel voisin et la position du pixel de base. Typiquement, cet effet est crée de la manière suivante :

$$F = \frac{\sum_{i=1}^{n}\sum_{j=1}^{n} P_{ij}C_{ij}}{S},$$

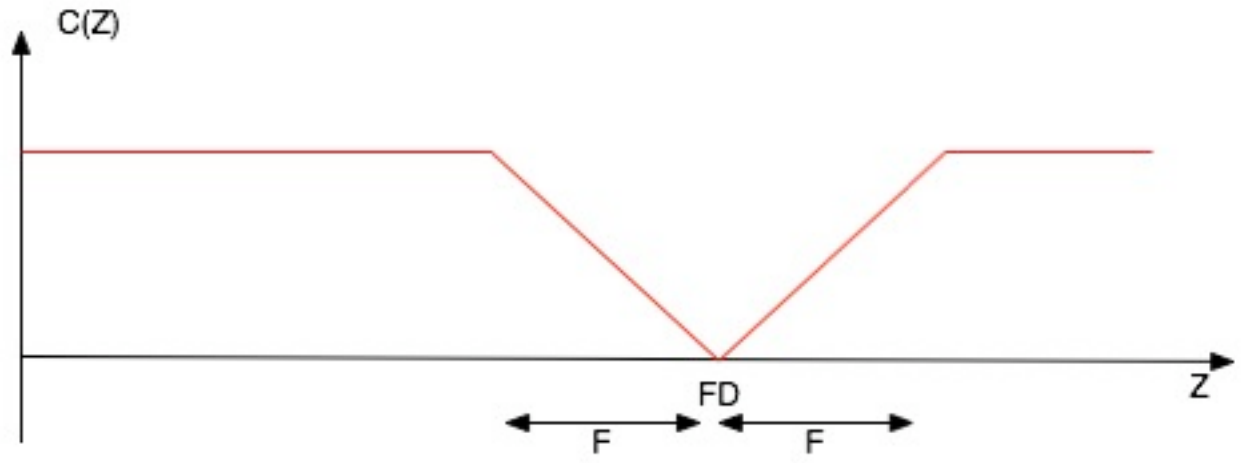
Où F est la valeur finale du pixel rendu flou, P la valeur d'un pixel voisin de du pixel de base, C, le coefficient de la matrice gaussienne et n la dimension horizontale et verticale de la matrice. S est la somme de toutes les valeurs de la matrice gaussienne.

Nous venons de voir très simplement comment rendre la profondeur de champ en deux passes. La technique est donc très simple, c'est une interpolation entre une image nette et une image floue. Deux passes sont bien nécessaires puisque qu'il faut conserver en mémoire la scène originale pour pouvoir l'interpoler avec la scène floue. Le rendu final sera donc entièrement basé sur le calcul du coefficient qui va interpoler ces deux images.

Le calcul du coefficient d'interpolation est basé sur la profondeur de la scène. En effet, reprenons l'exemple de la lentille mince convergente. Lorsque l'objet observé est placé sur le plan focal, alors celui-ci est net. Et plus l'objet sort du plan focal, plus celui-ci devient flou. Pour rendre cet effet, nous allons calculer le coefficient d'interpolation de telle sorte que la valeur 0 signifie que le rendu sera net et flou pour la valeur 1. Nous allons simplement appliquer un coefficient linéaire qui simulera le passage du net au flou selon la distance de l'objet à la focale. La fonction mathématique du coefficient d'interpolation noté C en fonction de la distance Z pour une telle lentille de focale F doté d'un plan focal situé à la distance FD sera :

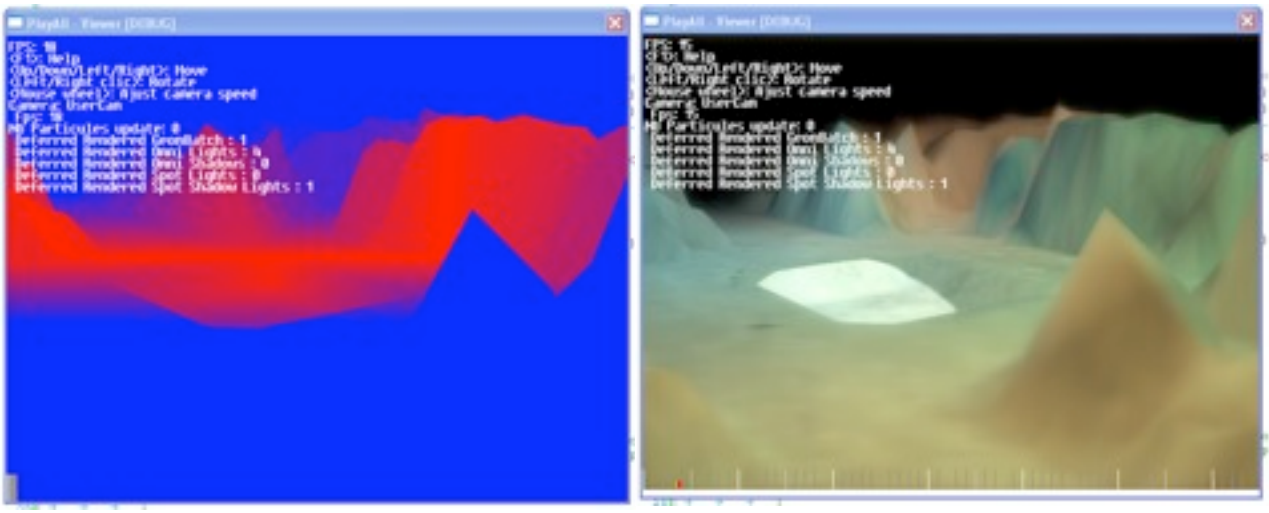
$$C(Z) = max(1, \frac{|Z - FD|}{F})$$

Graphiquement, voici le résultat du coefficient en fonction de la distance :



Amplitude du Coefficient d'interpolation en fonction de la distance et du foyer

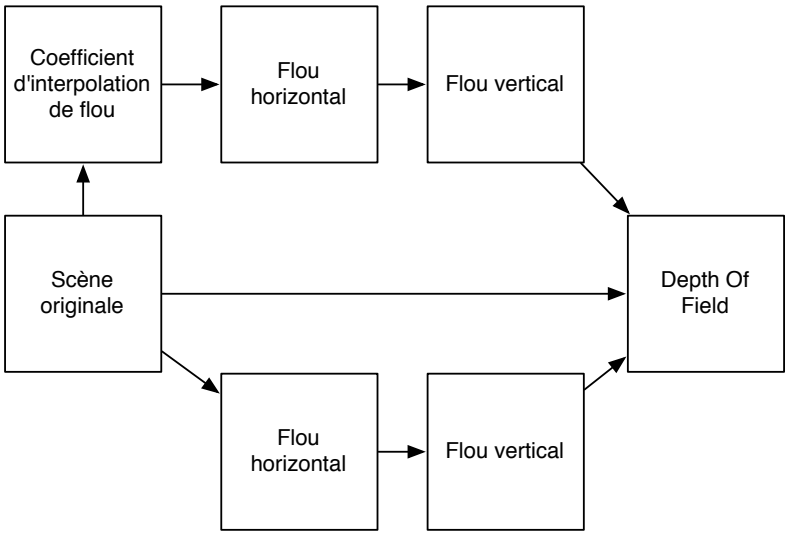
Et voici son implémentation sur une scène :



A gauche, on peut voir le calcul du coefficient d'interpolation. Plus la zone est rouge, et plus la scène sera nette. A l'inverse, plus la zone est bleue, et plus la zone sera floue. A droite, l'image finale rendue a l'écran, on remarque bien que le milieu de la scène est net et l'arrière et l'avant plan de la scène est flou.

Implémentation :

Voici le processus de rendu de la profondeur de champ



Processus de rendu de la profondeur de champ

On part de la scène nette originale. On la rend floue en parallèle dans un *render target* différent en appliquant un flou horizontal puis vertical. Dans un autre *render target*, on calcule le coefficient d'interpolation de flou jouant le rôle du cercle de confusion. On applique un flou sur le *render target* du coefficient d'interpolation afin de corriger des artefacts apparents sur les bords des objets, cette fonctionnalité sera détaillée dans la partie amélioration du processus de la profondeur de champ.

Au final, les trois *render target* obtenus (scène originale, scène floue et coefficient d'interpolation) permettent de rendre la scène finale du champ de profondeur.

Limitations :

La technique est limitée dans le sens où l'on effectue un flou gaussien discret. En effet, le flou gaussien parfait consisterait à faire l'intégrale sur toutes les valeurs réelles de la fonction gaussienne. Dans ce cas, nous effectuerons un nombre trop important de calcul au niveau du *pixel shader*, comme somme infinie de valeurs. Calculer l'intégrale n'est donc pas possible, puisque cette méthode est beaucoup trop coûteuse en calculs. Dans notre cas, nous avons discrétisé les valeurs pour faire au finale une somme sur 13 valeurs à intervalle régulier de la fonction de gauss sur l'axe horizontal et vertical. Cela qui limite dans ce sens la technique, car s'il l'on augmente le flou en allant étalant la fonction de gauss, nous commencerons alors à discerner des paliers de flou puisque l'on se base que sur un nombre de valeur discret limité. Ainsi, il existe une limite de flou maximal, c'est-à-dire le flou le plus intense que l'on arrive à produire tant que l'on ne voie pas à l'oeil les paliers discrétisés.

Voici les performances obtenues lorsque l'on applique un flou gaussien pour une même scène (ici : FPS est l'acronyme de *Frame* Par Seconde):

scène originale sans flou : 31 FPS équivaut à 32.2ms par frame

à gauche : flou sur 13 samples : 28 FPS équivaut à 35.7ms par frame
à droite : flou sur 25 samples : 26 FPS équivaut à 38.4ms par frame

Il y a deux remarques à faire suite à ces résultats :

D'une part, il est nécessaire de connaître les performances en milli-secondes par *frames* plutôt qu'en *frames* par seconde, étant donné que la première formule est linéaire et la deuxième est une fonction non linéaire en 1/x. Par exemple, s'il l'on gagne une *frame* par seconde à 10 FPS, alors cela ne revient pas au même de dire que l'on gagne une *frame* par seconde à 100 FPS. Le calcul en seconde par *frame* est proportionnel est alors choisit pour faire les tests de performances.

D'autre part, on remarque que l'on ne distingue à l'oeil pas forcément une image floue de meilleure qualité à 25 samples (ou échantillonnages) plutôt qu'une image floue à 13 samples. Nous avons alors choisi d'implémenter le processus de flou à 13 samples.

Améliorations :

i) *Arctefacts* visibles sur les bords des objets

Nous appliquons un flou en plein écran qui calcule le cercle de confusion pour chaque pixel dans le *render target*. Les pixels qui sont dans la focale utilisent un cercle de confusion de 0 (c'est-à-dire qu'ils sont net). Nous appliquons ensuite un flou gaussien ce qui donne des résultats satisfaisants. Nous divisons l'image par quatre afin d'utiliser l'*anti-aliasing* qui va faire la moyenne des pixels, l'avantage en plus de cela est de mettre moins de mémoire dans les textures des *renders targets*.

Mais cela nous donne pas un résultat satisfaisant. Il reste un problème pour les formes qui se situent au premier plan et ne sont pas dans la focale. En effet : les contours de l'objet de premier plan est alors flou à 50 pourcent (à cause des continuités du flou gaussien), alors que nous voulons qu'elle soit flou à 100 pourcent.

Pour fixer ceci, nous considérons deux objets nous recalculons le cercle de confusion de certains pixels : ceux qui sont situés à la frontière des objets qui ont un cercle de confusion de taille différentes, nous utilisons alors le cercle de confusion de plus grande taille.

Puisque les pixels ont plusieurs voisins, nous allons éviter de faire des recherche parmi tous les voisins pour des raisons de performances. A la place de cela, nous calculons le cercle de confusion, puis le cercle de confusion rendu flou par le filtrage de gauss pour en déduite le cercle de confusion final qui est le plus grand des deux. C'est donc à ce niveau que l'on justifie l'usage du flou du cercle de confusion dans le processus de rendu.

Considérons deux objets tel que l'un soit flou au premier plan et l'autre dans la focale à l'arrière-plan. Ils ont tous les deux des cercles de confusion bien différents à la frontière des deux objets.

Alors, dans ce cas, le diamètre du cercle de confusion Db est donné par :

$$Db = \frac{1}{2} * (D0 + D1)$$

Cette équation peut donner précisément le diamètre de flou lorsque le gradient du diamètre est le même sur les deux bords.

Nous pouvons avoir le diamètre original du pixel $D0$ et le diamètre rendu flou Db dans deux *render target* différents.

On peut alors estimer $D1$ en nous appuyant l'équation précédente :

$$D1 = 2 * Db - D0$$

Nous définissons finalement le diamètre du cercle de confusion D par

$$D = max(D0, 2 * Db - D0) = 2 * max(D0, Db) - D0$$

l'équation suivante :

Soit $D1$ le plus grand cercle de confusion. Cette équation va faire transiter depuis un diamètre $D1$ de région frontière vers un diamètre $D0$. La région de flou sera alors par dessus la région nette. C'est bien le résultat que l'on attendait.

Artefact visible à gauche et rectifié à droite

A gauche on peut voir l'*artefact* sur les piliers en avant-plan, car les contours des piliers sont nets alors qu'il devraient être flou. Le fait d'ajouter une passe de flou sur le coefficient d'interpolation, ainsi que la formule finale qui va calculer le cercle de confusion, permettent de supprimer cet *artefact* et l'on obtient la scène sur la droite.

ii) Script permettant de personnaliser la profondeur de champ

Afin de faciliter la tâche aux graphistes mettant en place le profondeur de champ final, un script a été crée pour faciliter la paramétrisation de l'effet.

Il est possible de donner la distance focale, la position ainsi que le taux de flou maximal à la lentille d'avant et d'arrière plan.

La lecture des paramètres de la profondeur de champ directement dans un fichier. La profondeur de champ dépend des caméras et est interpolé linéairement en fonction de la *frame* et des paramètres de la profondeur de champ du début et de fin d'une séquence.

iii) Nouveau calcul du Z

Pour calculer le Z, nous le lisons directement la valeur noté $Z(1)$ sur la texture qui contient les informations sur la profondeur, pour chaque pixel.

Cependant, le $Z(1)$ que l'on lit est celui qui correspond au $Z(1)$ non linéaire calculé en espace vue et est compris entre 0 et 1.

C'est-à-dire que $Z(1)$ est précis pour les objets proches de la vue, et tend rapidement vers 1 pour des objets lointains.

Ainsi, on utilise la matrice de projection inverse de l'espace vue vers l'espace caméra pour récupérer la valeur notée $Z(2)$ correspondant après transformation à une distance.

Ce $Z(2)$ est alors linéaire et il est facile de comparer la profondeur du $Z(2)$ d'un pixel avec la profondeur à laquelle est affiché le pixel correspondant.

Voici le calcul du $Z(2)$ relatif au $Z(1)$ par la matrice inverse de projection, on multiplie le vecteur $\begin{bmatrix}0 & 0 & Z(1) & 1\end{bmatrix}$ par la matrice de projection inverse.

Le résultat du calcul est un vecteur de la forme $\begin{bmatrix}a\\b\\c\\d\end{bmatrix}$. On calcul alors $Z(2)$ de la façon suivante : $Z(2) = \frac{c}{d}$.

Cependant, on remarque que l'on passe en paramètre la matrice de projection inverse, cela pour chaque *frame*.

En reprenant le calcul précédent, pour une matrice de projection inverse quelconque de la forme :

$$\begin{bmatrix}a11 & a12 & a13 & a14\\a21 & a22 & a23 & a24\\a31 & a32 & a33 & a34\\a41 & a42 & a43 & a44\end{bmatrix}$$

On a :

$$\begin{bmatrix}a\\b\\c\\d\end{bmatrix} = \begin{bmatrix}0 & 0 & Z(1) & 1\end{bmatrix} * \begin{bmatrix}a11 & a12 & a13 & a14\\a21 & a22 & a23 & a24\\a31 & a32 & a33 & a34\\a41 & a42 & a43 & a44\end{bmatrix} = \begin{bmatrix}Z(1) * a31 + a41\\Z(1) * a32 + a42\\Z(1) * a33 + a43\\Z(1) * a34 + a44\end{bmatrix}$$

On peut en déduire $Z(2)$ directement via 4 paramètres :

$$Z(2) = \frac{c}{d} = \frac{Z(1) * a33 + a43}{Z(1) * a34 + a44}$$

Cela se résume à passer un vecteur de paramètres à chaque *frame* au lieu d'une matrice, ce qui fait gagner du temps de transfert de données.

De plus, on peut comparer le nombre d'instructions que l'on utilise par l'une et l'autre méthode :

- Matrice : 10 instructions utilisées :1 texture et 9 arithmétiques

- Vecteur : 5 instructions utilisées : 1 texture et 4 arithmétiques

On remarque clairement que le calcul du Z en passant un vecteur est plus efficace qu'en passant une matrice.

3-5 LUEUR DIFFUSE

Introduction :

Après avoir implémenté la technique de la profondeur de champ, J'ai ajouté une nouvelle technique en temps réel de rendu au sein du moteur PlayAll : l'effet de lueur diffuse (ou Glow)

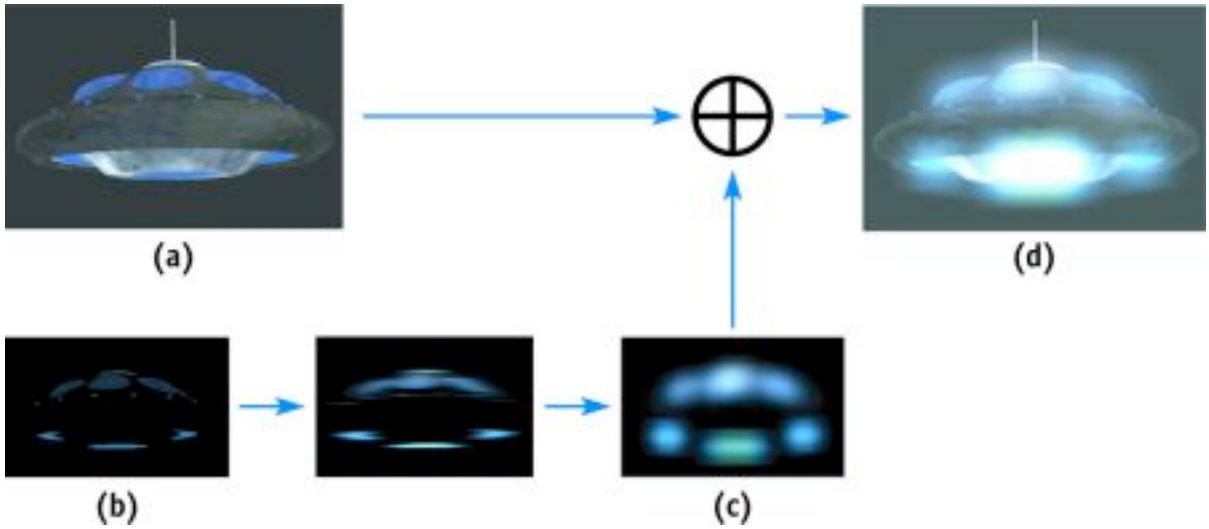
On retrouve cet effet, dans des titres, qui amène une touche particulière sur des textes, des effets créatifs variés, des transitions entre images, ou encore pour des trucages comme les fameux sabres laser de la saga de Georges Lucas.

L'effet consiste donc à mettre en valeur les couleurs les plus vives de la scène. Ainsi, on donne plus d'intensité à la lumière et l'observateur perçoit des sources lumineuses très brillantes.

Description et mis en oeuvre de la technique

Cette technique est très comparable à celle utilisée pour la profondeur de champ. Elle consiste à pré-calculer des tampons d'image puis de les combiner ensemble pour en déduire l'image finale rendue à l'écran.

Voici les étapes du procédé sur un exemple :



Processus de rendu de la profondeur de la lueur diffuse

La première chose à faire est d'isoler des parties de la scène où l'on appliquera la lueur diffuse de celles qui ne seront pas affectées par la lueur diffuse. Pour ce faire, nous rendons la scène normalement. Nous créons en parallèle un tampon d'image a partir du rendu de la scène. Cette image est noire partout, excepté aux endroits clairs et lumineux que l'on conserve. Ensuite, on rend l'image ainsi produite floue, ce qui permet de faire déborder la luminosité intense de sa position d'origine. La dernière étape consiste à additionner l'image nette à l'image lumineuse et floue. Le résultat final nous donne une image de la scène où les l'intensité lumineuse est bien mise en valeur.

En pratique, on peut voir les résultats appliqués à une scène dans le moteur de jeux PlayAll :

A gauche, la scène normale, à droite, la scène avec la lueur diffuse.

On remarque bien que cet effet fait ressortir les couleurs et les rend plus vives.

Applications :

Les effets de rendu de la profondeur de champ et de la lueur diffuse ont été présentés, en plus d'autres fonctionnalités du moteur PlayAll, à l'occasion de la conférence de GDC 2008 (Game Developers Conference). Cette conférence rassemble les meilleurs développeurs de la communauté européenne et internationale autour du partage d'expériences, des échanges d'idées et des dernières innovations dans le développement de jeux vidéo de la génération actuelle et future.

3-5 LES MATÉRIAUX PHYSIQUE

Introduction :

Un matériel physique (ou physical materials) est une propriété qui est ajoutée à chaque face d'un *mesh*. L'intérêt d'ajouter cette information supplémentaire est de pouvoir déterminer la nature d'une collision entre plusieurs *meshs*.

En pratique, un matériel physique est en particulier composé d'un *flag* ou plus exactement un entier représentant un champ de bits que nous noterons 0bx...x, où x est un bit prenant pour valeur 0 ou 1.

Le champ de bits définit alors un vecteur de bits pour lequel le matériel physique pourra activer ou désactiver des propriétés définissant le matériel. Cela consiste à assigner les bits x du vecteur à la valeur 0 (désactivé) ou 1 (activé).

Prenons le cas d'un exemple simple. Nous définissons quatre matériaux physiques :
- Le matériel nul (NULL) avec un *flag* binaire 0b00.
- Le matériel eau (WATER) avec un *flag* binaire 0b01.
- Le matériel terre (EARTH) avec un *flag* binaire 0b10.
- Le matériel boue (MUD) est composé d'eau et de terre. Le *flag* binaire est alors crée à partir d'un OU logique entre le matériel WATER et le matériel EARTH, c'est-à-dire 0b11.

Un joueur qui se déplace sur une scène rentrera en interaction avec les *meshs* de l'environnement. L'algorithme de collision spécifique au déplacement du joueur consistera à lancer une bulle ou boite englobante à l'instant dt devant lui. La boite englobante va récupérer tous les noeuds contenant les *meshs* qui sont contenu à l'intérieur de cet espace.

Cependant, en même temps que l'on lance la boite englobante, nous précisons en plus un *flag* binaire noté FlagBoite qui va déterminer quels seront les *meshs* que nous devons retenir dans l'espace de la boite.

En effet, dans le cas du joueur, nous allons permettre le fait que celui-ci puisse rentrer en collision avec les *meshs* composés d'un matériel physique de type EARTH. De même que le joueur peut se déplacer dans l'eau et pour simplifier ne rentrera pas en collision avec les *meshs* composés d'un matériel physique de type WATER.

Ainsi, nous assignons le *flag* FlagBoite à la valeur 0b10.

Un des principes de l'algorithme de collision dans le cas des boites englobantes sera d'effectuer un ET logique sur le matériel physique des *meshs* contenus dans la boite englobantes avec le *flag* FlagBoite. Cela aura pour effet de filtrer tous les *meshs* ne correspondant pas à cette condition.

En particulier, on remarque que les *meshs* possédant un matériel de type NULL ou WATER ne seront pas considérés comme *mesh* de collision dans la boite englobante, mais les *meshs* de type EARTH et MUD le seront. Le filtrage proposé est bien celui attendu, le joueur ne rentrera en collision qu'avec la terre et la boue.

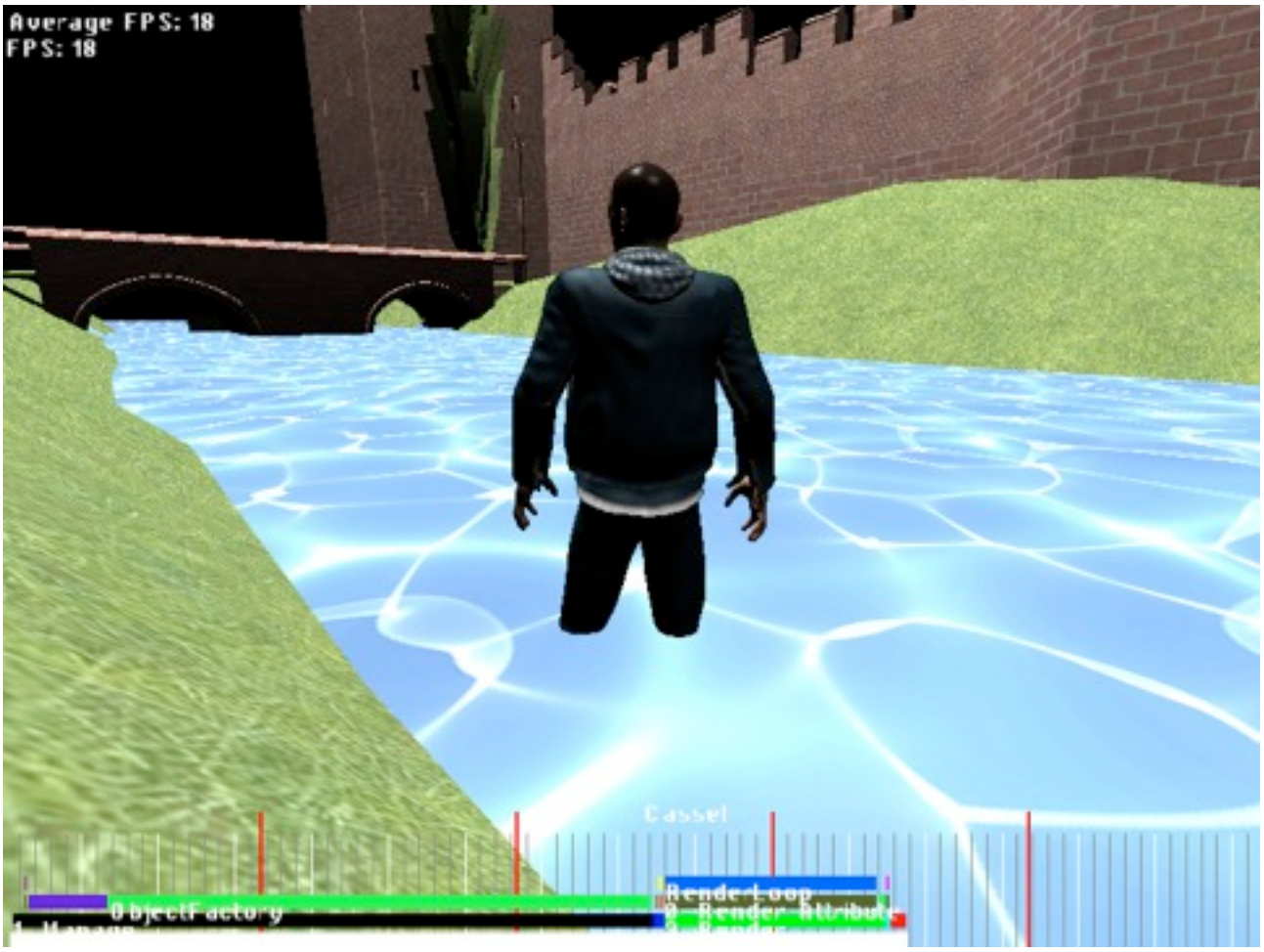
De plus, à l'issue de la collision, le matériel physique pourra contenir des informations supplémentaires qui déboucherons sur des actions spécifiques selon matériel.

Par exemple, on peut considérer que la terre est un sol dur, mais que la boue, de matière plus visqueuse, engendre plus de frottements que la terre et donc rajouter ces informations dans le matériel physique. De même que l'on pourrait aussi rajouter un son spécifique à la collision entre deux *meshs*, selon leurs matériaux physique.

Développement :

J'ai ajouté dans un premier temps le système d'export des matériaux physique dans le ContentPipeline.

A partir d'un fichier XML, on précise quels sont les propriétés spécifiques au matériel physique qui vont composer le *flag* final de celui-ci. Puis, j'ai changé le processus d'export des collisions pour que les géométries des *meshs* de la scène prennent en compte le matériel physique que l'on leur aura attribué. Enfin, j'ai fait prendre en compte ces nouveaux changements dans le moteur de jeu PlayAll, lors du chargement des *meshs* et de leurs collisions. Pour vérifier l'intégrité des matériaux physique, j'ai effectué des tests sur une scène de jeu regrouppant plusieurs objets de matériaux différents.



Capture d'écran révélant un exemple d'utilisation des matériaux physique

Dans le cas suivant, l'image montre le personnage en collision avec le matériel physique EARTH du sol, mais ne rentre pas en collision avec le matériel physique WATER de l'eau.

4EME PARTIE

CONCLUSION

Durant toute la période de mon stage, j'ai pu apporter des solutions aux différents besoins qui m'ont été posés, répondant ainsi à une demande de la société pour améliorer la production. Les résultats de l'étude de chacune des techniques sont à ce jour intégrés et utilisés au sein du KTEngine ainsi que ses outils. Il est assez plaisant de se rendre compte que le travail que l'on a fourni est utile pour la production du jeu et utilisé par les autres membres de l'équipe.

Ce stage de 6 mois chez Kyloton Entertainement a bien sûr été pour moi une expérience très enrichissante sous dans nombreux aspects. Tout d'abord j'ai pu avoir une première vision de la façon de créer et produire des jeux vidéo, ainsi que des méthodes de travail qui le permettent. Prendre le temps de poser tous les problèmes à plat, et de rechercher toutes les solutions possibles pour ensuite choisir la plus adaptée, est ainsi bien plus efficace et professionnel, que de plonger tête baissée dans du code directement. Je pense aussi avoir gagné en rigueur en prenant conscience de la responsabilité qui repose sur les programmeurs au sein d'une équipe de développement. Le code produit doit être propre, sans bug et optimisé. Je pense, grâce aux conseils et remarques de Benoit Jacquier, j'ai réellement amélioré mon niveau en programmation C++ et mes méthodes de programmation de manière générale. J'ai pu constater qu'un stage de longue durée au sein d'une société est une expérience unique pour mieux comprendre comment fonctionne la mise place d'un projet en travail d'équipe, mais aussi en termes de contacts humains. Grâce à ce stage, je pense aussi avoir fait un premier grand pas dans le monde du jeu vidéo, et plus précisément dans le développement de jeux vidéo.

Pour conclure, je dirais que je pense avoir eu beaucoup de chance de pouvoir effectuer ce stage chez Kylotonn Entertainement. Grâce aux enseignements fournis par mes professeurs lors de ma formation en ingénieur Informatique, associé à un sujet de stage très motivant, correspondant exactement à mes attentes, et des conditions de travail plus qu'appréciables, j'ai pu passer de très bons moments durant cette période.

5EME PARTIE

GLOSSAIRE ET BIBLIOGRAPHIE

GLOSSAIRE

vue à la première personne (*ou FPS : First Person Shooter*) : type de jeu vidéo de tir en 3D dans lequel l'angle de vue proposé simule le champ visuel du personnage incarné.

placage de relief (*ou bump mapping*) : technique de rendu qui sert à donner du relief aux objets, textures ou à toutes autres choses en 2D ou en 3D, dans ce dernier cas par placage d'une image (texture) sur l'objet.

carte de lumière (*ou lightmap*) : texture contenant une information de lumière permettant en particulier d'améliorer les calculs en stockant.

cartes d'ombres (*ou Shadows Map*) : une technique d'affichage d'une ombre.

volumes d'ombres (*ou Shadows Volume*) : une technique d'affichage d'une ombre.

intergiciel (*ou Middleware*) : logiciel servant d'intermédiaire de communication entre plusieurs applications.

travail artistique (*ou artwork*) : désigne l'ensemble des travaux de création artistique destinés à mettre en valeur un produit tel qu'un album de musique, un jeu vidéo ou encore un film sur support vidéo.

jeu de rôle (*ou RPG : Role Playing Game*) : jeu de société dans lequel plusieurs participants créent ou vivent ensemble une histoire par le biais de dialogues, chacun incarnant un personnage.

Game Design (mot français et anglais) : processus de conception préalable des mécaniques d'un jeu avant son élaboration.

Gameplay (mot français et anglais) : terme caractérisant des éléments d'une expérience vidéoludique. le gameplay serait les règles du jeu, la manière dont le joueur est censé y jouer, la fluidité de ces règles une fois appliquées à l'environnement du jeu, et également la manière dont le joueur peut jouer, les possibilités offertes par l'environnement.

DirectX : Microsoft DirectX est une collection de bibliothèques (ou API) destinées à la programmation d'applications multimédia. Plus particulièrement de jeux ou de programmes faisant intervenir de la vidéo, sur les plates-formes Microsoft.

HLSL ou **High Level Shader Language** : langage de programmation des pipelines des cartes graphiques 3D intégré à l'API DirectX.

shaders (mot français et anglais) : programme utilisé en image de synthèse pour paramétrer une partie du processus de rendu réalisé par une carte graphique.

rendu pré-calculé ou **rendu différé** (*ou Deferred Shading*) : processus de rendu consistant à différer le rendu de l'image finale en stockant au préalable des informations dans plusieurs tampon d'images afin d'optimiser les calculs sur certaines techniques de rendu.

vertex shader (mot français et anglais) : dans le processus de rendu graphique, l'utilisation des vertex shader permettent de calculer la projection de coordonnées des sommets des primitives à partir de l'espace 3D dans l'espace écran.

fragment shaders ou **pixel shader** (mot français et anglais) : dans le processus de rendu graphique, l'utilisation des pixel shader permettent de déterminer la couleur du pixel.

fichier d'en-tête (ou *header file*) : fichier dont le nom se termine traditionnellement en .h et permettent de partager des déclarations entre plusieurs fichier source composant un même programme.

fichiers sources (*ou source file*) : fichier dont le nom se termine traditionnellement en .cpp et contient les définitions de des fonctions décrites dans les fichiers d'en-tête.

CPU ou **Central Processing Unit** : unité centrale de traitement d'un ordinateur.

GPU ou **Graphics Processing Unit** : microprocesseur présent sur les cartes graphiques au sein d'un ordinateur ou d'une console de jeux vidéo.

Mesh (mot français et anglais) : objet tridimensionnel constitué de polygones sous forme de fil de fer.

vertex buffer (mot français et anglais) : tampon composé de vertex, ou vecteurs de l'espace.

index buffer (mot français et anglais) : tampon composé d'indice, permettant de relier les vertex entre eux et de pouvoir afficher les primitives graphiques (triangle et rectangle).

rapidité d'affichage (*ou framerate*) : La rapidité d'affichage est généralement exprimé en « frames per second » (fps), soit « images par seconde » (i/s) ou aussi en hertz (Hz).

frame : image rendue à l'écran. Le nombre de d'images rendues à l'écran par secondes indique la rapidité d'affichage.

fonction inline (mot français et anglais) : fonction dont le code est inséré directement dans le flux de code de la fonction appelante.

tampon de profondeur (*ou Z-Buffer*) : dans le cadre de l'affichage d'une scène 3D, le tampon de profondeur, de précision de 32 bits, permet de gérer le problème de la visibilité qui consiste à déterminer quels éléments de la scène doivent être rendus, lesquels sont cachés par d'autres et dans quel ordre l'affichage des primitives doit se faire.

MRT ou **Multiple Render Target** (mot français et anglais) : possibilité donnée au GPU d'écrire des valeurs de pixel distinctes dans plusieurs tampons d'images simultanément.

render target : cible de rendu contenant un tampon d'image.

artefacts : élément indésirable et/ou défectueux non attendu rendu à l'écran.

anticrénelage (*ou anti-aliasing*) : méthode permettant d'éviter le crénelage, un phénomène qui survient lorsqu'on visualise certaines images numériques dans certaines résolutions.

flag (mot français et anglais) : valeur binaire (de type vrai ou faux) indiquant le résultat d'une opération ou le statut d'un objet

BIBLIOGRAPHIE

- Amélioration du blur :

NGUYEN Hubert. GPU Gems3. Copyright NVIDIA Corporation 2008

- Techniques pré-calculées, la profondeur de champ et la lumière diffuse :

Shader x^5 advanced rendering techniques. Copyright 2007. Edited by Wolfgang Engel. Published by Charles River Media. Printed by Thomson Lerning Inc :
p163 Real-Time Depth-of-Field Implemented with a Postprocessing only Technique (by David Gillham)

ShaderX. Shader Programming Tips and Tricks with DirectX 9 :
Real-Time Depth of Field Simulation

A Survey of Techniques, GPU Gems3 Programming Techniques, Tips, and Tricks for Real-Time Graphics. Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks. Edited by Randima Fernando. Copyright NVIDIA Corporation 2004. Woldgang Engel. Paperback 2002 :
p343 Real Time Glow
p375 Depth of Field

- Programmation C++ :

STROUSTRUP Bjarne. Le Langage C++. CampusPress. 2001

GÉRON Aurélien, TAWBI Fatmé. Pour mieux développer avec C++ Design patterns, STL, RTTI et smart pointers. EnterEditions

6EME PARTIE

ANNEXE

MEMBRES PLAYALL

Les membres de PlayAll sont :

- Studios de développement de jeux

Kylotonn Studio de développement FPS PC 75 Paris

Darkworks Studio de développement Action PC Xbox360 PS3 75 Paris (Alone in the Dark, Cold Fear)

Load Inc Studio de développement Action XboxLive Arcade 75 Paris (Mad Tracks)

Whitebirds Studio de développement Aventure PC 94 Joinville le pont

Wizarbox Studio de développement Action PC PS2 Xbox 92 Sèvres (Kirikou, Azur et Asmar ...)

- Studios de Middlewares spécialisés

Atonce Middleware moteur graphique PS2, PSP. 33 Bordeaux

Bionatics Middleware terrain procédural 34 Montpellier

SpirOps Middleware intelligence Artificielle et moteur d'analyse topologique 75 Paris

Voxler Middleware son 75 Paris

- Laboratoires de recherche

CNAM / Cedric Moteur son 75 Paris

ENST Réseau 75 Paris

LIP6 Réseau 75 Paris

LIRIS Animation procédurale / motion capture faciale / moteur graphique 69 Lyon

ENJMIN Ecole Nationale du Jeu et des Médias Interactifs Numériques 16 Angoulême