Sommaire

INTRODUCTION 1

APPROCHE GENERALE..2

A. Présentation du contexte 3
- Projecteur panoramique.. 3
- Avancement de la recherche 3
- Applications envisageables 4

B. Présentation du jeu 4
- Choix de l'application 4
- Choix du type du jeu 4
- Editeur 5

C. Outils utilisés 5
- Bibliothèque graphique – DirectX 5
- Modélisation 3D – 3D Studio Max 7

D. Organisation du groupe 7

APPROCHE TECHNIQUE..9

A. Le jeu 10
- Le jeu..10
- L'éditeur...11

B. Projection 11
- Principe du Cubic Environment 11
- Principe de la projection 12
- Implementation de la projection...14
- Performances : 17

C. Optimisation – Pixel Shader..18
- Explication...18
- HLSL et Assembleur 19

CONCLUSION 21

BIBLIOGRAPHIE 22

GLOSSAIRE 23

ANNEXES..25

Annexe 1 : Manuel d'utilisation de l'éditeur 26

Annexe 2 : glossaire des fonctions DirectX utilisées...................................37

Annexe 3 : Calcul de recombinaison avec angle...40

Annexe 4 : Splines cubiques...41

Annexe 5 : Image du jeu 43

Introduction

Lors de la 2ème année du DUT imagerie numérique, il est proposé aux étudiants de travailler sur un projet tuteuré afin d'appliquer les notions théoriques et pratiques étudiées en cours, et d'en découvrir d'autres.

Notre choix s'est porté sur un sujet proposé par M. Sarry, professeur de mathématiques et d'algorithmique pour le traitement d'image du département informatique de l'IUT du Puy en Velay et membre de l'Equipe de Recherche en Imagerie Médicale (ERIM) de l'université Clermont 1. Ce projet est la continuité d'un travail de l'année précédente et s'inscrit dans une logique de recherche sur un projecteur panoramique.

Dans le cadre d'un programme de recherche de l'ERIM et du Laboratoire de Logique Algorithmique et Informatique de Clermont Ferrand (LLAIC), le développement d'un projecteur panoramique permettant l'immersion de l'utilisateur dans un environnement virtuel sur 4π Stéradians* est en cours.

Lors du projet précédent, la faisabilité du système a été démontrée. Il s'agit, cette année, de créer une application, en l'occurrence un jeu vidéo, permettant de démontrer l'aspect immersif du système et la possibilité d'interagir avec la scène.

L'année dernière, les tests sur le projecteur ont été effectués à l'aide d'applications réalisées sous OpenGL*. Cette année, il a été décidé d'utiliser l'API* DirectX* développée par Microsoft, qui est l'autre bibliothèque graphique de référence dans le monde de la 3D temps réel*. Ce choix comportait une réelle difficulté car DirectX n'est pas étudié en cours, contrairement à OpenGL.

Dans un premier temps, la partie générale du projet sera expliquée, ainsi que ses principales étapes. Dans un second temps, les parties techniques seront abordées.

Approche générale

A. Présentation du contexte

- Projecteur panoramique

Le but du projecteur panoramique est d'afficher une scène 3D sur les murs d'une pièce à l'aide d'un vidéo projecteur classique et d'un miroir hémisphérique. L'utilisateur se trouve ainsi plongé dans un monde virtuel sur 4π stéradians. Pour cela, l'image est projetée sur le miroir placé à la verticale du projecteur et est réfléchie sur les murs de la pièce.

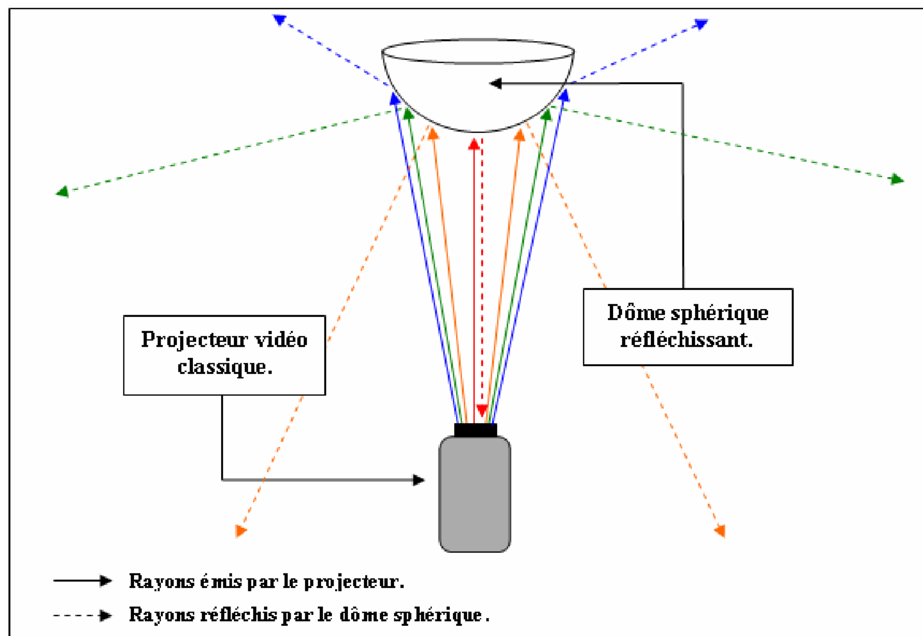


Figure 1 : Principe de conception du projecteur panoramique.

La recherche sur ce projecteur s'inscrit dans un programme mené conjointement par l'ERIM et le LLAIC. A terme, le projecteur possèdera un système d'auto calibrage permettant de s'adapter à la pièce, en tenant compte de sa forme, des volumes présents et même des couleurs en appliquant à l'image projetée un filtre chromatique calculé lors de la phase de calibrage.

- **Avancement de la recherche**

Différents stages et projets ont déjà été effectués sur le système.

L'année dernière deux applications ont été réalisées. Elles permettent de projeter des images :

- en temps réel avec la bibliothèque graphique OpenGL, dans un jeu vidéo mettant en scène une course de vaisseaux.
- pré calculées* d'une animation faite sous 3DS Max* et exportée grâce à un plug-in* conçu pour le projet.

Ces deux applications utilisent des projections sphériques et perspectives dont toutes les transformations avaient dû être calculées et implémentées lors du projet.

Lors d'un stage de l'année 2005, une application permettant l'auto calibrage du projecteur a été commencée. Elle est toujours en cours de développement, par un étudiant de doctorat.

- Applications envisageables

Le système, une fois abouti, possède plusieurs avantages :
- un déplacement aisé et un faible encombrement
- un réglage facile et rapide grâce à l'auto calibrage
- ne nécessite pas de salle spécifique car la projection s'adapte à la pièce (couleur des murs, volumes dans la pièce…)

De ce fait, il est possible d'imaginer une infinité d'applications:
- des jeux vidéo adaptés au projecteur, permettant une immersion totale du joueur, avec possibilité de rajouter des accessoires tels que des fusils ou autres, fonctionnant à l'aide d'un capteur de position et de visée…
- des outils de présentation comme par exemple, la projection d'une cuisine virtuelle dans une maison en construction pour présenter aux clients les différents aménagements possibles.
- des performances artistiques, par exemple en projetant un univers qui fait perdre la notion de l'espace.
- des salles de cinéma dédiées à la diffusion de films spécifiques au projecteur.

B. Présentation du jeu

- Choix de l'application

Le but du projet de cette année est de créer une application de démonstration pour le projecteur. Le jeu vidéo, de part son coté immersif et interactif est probablement l'une des applications qui impressionne le plus. C'est également l'application qui parait la plus apte, au vu de l'avancement des travaux de recherche, à montrer le principe essentiel du système.

Au début du projet, la possibilité d'obtenir un capteur 6 dimensions*, qui permet de connaître la position du capteur et de la direction visée, apportait une dimension encore plus interactive au jeu. Cela permettait de s'affranchir totalement des périphériques habituellement reliés à un ordinateur, comme clavier et souris, et qui limitent les possibilités de mouvements, surtout dans un environnement à 4π stéradians. De plus, ce capteur permet de donner un exemple des accessoires qu'il serait possible de rajouter au projecteur. C'est également la raison du choix de concevoir un jeu de tir.

- Choix du type du jeu

Le jeu a été réalisé sur le modèle d'un FPS (First Person Shooter) avec vue à la première personne, c'est-à-dire à travers les yeux du personnage que l'on incarne. Ce choix, garant d'une réelle interactivité, est imposé par la configuration du système car voir le héros, censé se trouver au milieu de la pièce, projeté sur les murs, serait une aberration.

L'obtention d'un capteur 6 positons aurait permis de détecter le point visé par l'utilisateur et donc de simuler l'utilisation d'une arme à feu ou autre. Il aurait pu être envisagé de fixer ce capteur sur un support afin de simuler un fusil et d'obtenir une immersion encore plus totale. Le résultat obtenu aurait été comparable à une borne d'arcade mais dans un univers englobant l'utilisateur.

L'espace choisi pour le jeu est le corps humain. Le héros, un médicament, se déplace dans le corps à travers le réseau de vaisseaux sanguins et doit éliminer tout ce qui ne devrait pas se trouver dans le sang, comme par exemple des virus ou des bactéries. Par contre il doit faire attention à ne pas détruire les éléments vitaux du sang tels que les globules blancs et rouges.

Ce thème a été choisi en premier lieu, en référence au projet proposé par le Genethon. De plus, cet aspect gentil/méchant associé au corps humain permet de proposer un jeu qui pourrait éventuellement être ludoéducatif, par exemple, en expliquant en parallèle, pourquoi les globules rouges sont nécessaires, et pourquoi les virus sont nocifs.

Le jeu a été baptisé Organic, en référence à l'univers évoqué et aussi pour la sonorité dynamique du nom.

Utiliser le thème du corps humain, et surtout des vaisseaux sanguins, imposait l'utilisation de couleurs vives, notamment le rouge. Cela représente un gros avantage pour la projection, car il est nécessaire d'avoir un fort contraste de couleur pour pourvoir distinguer clairement les objets une fois projetés. Cette nécessité a été remarquée lors de la visualisation de vidéos réalisées l'an passé. La palette de couleurs utilisée pour les personnages, décors, et autres objets d'Organic, a été choisie en gardant à l'esprit cette nécessité d'avoir des couleurs bien visibles.

Une autre des spécificités du jeu, est la prédéfinition des trajectoires. En effet tous les déplacements sont calculés à l'avance et définis à l'aide d'une spline 3D. Ainsi le joueur n'a pas besoin de gérer ses déplacements, ce qui simplifie l'utilisation du jeu. En effet, devoir gérer à la fois les déplacements au clavier, et les tirs à la souris ou avec le détecteur rendent les choses plus difficiles, surtout lorsqu'il faut regarder dans toutes les directions. De plus cette méthode permet de s'affranchir des tests de collisions et donc de gagner du temps de calcul.

Le design des personnages à été choisi de manière à avoir un minimum de polygones, le but étant de minimiser le temps de calcul pour l'affichage de ceux-ci. De ce fait, tous les personnages sont de style cartoon. Ce style permet d'avoir des personnages au design simple, et permet aussi d'avoir des couleurs vives, donc d'être visibles à la projection. Cela donne aussi un coté plus ludique au jeu.

- Editeur

Pour créer les niveaux, un éditeur a été intégré au jeu. Cet éditeur permet d'importer le décor et les personnages. Il permet aussi de définir la trajectoire du héros et des autres entités.

La présence d'un éditeur permet de rendre le jeu tout à fait évolutif, puisque l'utilisateur peut créer son propre niveau, à condition qu'il possède différents modèles de décors et personnages. Il a aussi été imaginé de façon à ce que les nouveaux objets soient importés facilement, s'ils sont au bon format. Il est bien sûr possible de créer des décors et personnages appartenant à un thème complètement différent.

C. Outils utilisés

- Bibliothèque graphique – DirectX

Actuellement, il existe deux bibliothèques graphiques* faisant référence, DirectX et OpenGL. L'an passé il a été démontré qu'OpenGL pouvait être utilisé pour créer des applications destinées au projecteur. En décidant d'utiliser DirectX pour développer

l’application cette année, il a été possible de montrer qu’il permettait également d’effectuer les opérations pour la projection. Et par la même occasion, qu’il permettait au groupe d’utiliser une API non vue en cours, alors que l’IUT offre dans sa formation des cours d’OpenGL.

La grande nuance entre OpenGL et DirectX vient du fait que ce dernier regroupe sous son nom plusieurs modules :

- Direct3D : gère l'affichage 3D
- DirectDraw : gère l'affichage 2D
- DirectInput : gère les périphériques d'entrée (joystick, clavier, etc.)
- DirectPlay : gère les échanges de données par réseau
- DirectSound3D : gère les sons
- DirectMusic : API de DirectSound
- DirectVoice : gère les échanges vocaux en direct
- DirectShow : gère l'affichage et la capture de vidéo

Outre cette vision différente entre les deux API, une des difficultés inhérentes au choix de la bibliothèque DirectX est l'absence de documentation facile d'accès. OpenGL, étant Open Source, bénéficie d'une communauté très active qui fournit tutoriaux, cours, aide sur les forums, etc. DirectX est la bibliothèque propriétaire de Microsoft. De ce fait il existe très peu de documentations libres d'accès autre que la msdn*, complète et technique, fournie par Microsoft.

De plus, de part la nature du projet, il a été nécessaire d'étudier des notions avancées. L'approche de DirectX a donc été radicalement différente de celle faite pour OpenGL. Ainsi, contrairement à ce qui a été pensé au début du projet, l'étude d'une bibliothèque graphique en cours n'a pas apporté une réelle aide à la création du jeu.

Présentation du SDK* DirectX

Pour l’affichage, le SDK DirectX contient un module appelé DirectX Graphics, celui-ci est indépendant du matériel. Il s’appuie sur une abstraction du fonctionnement interne des cartes graphiques, le HAL (Hardware Abstraction Layer) plus communément appelé le « device *» ou « 3D device ». Le jeu Organic, utilise l’API 3D DirectX Graphics. Cette dernière envoie des commandes au HAL qui par la suite pilote la carte graphique par l’intermédiaire des drivers.

Au niveau de l’API se trouve la bibliothèque D3DX. C’est une LIB* (glossaire : extension de fichier de bibliothèque de données pour langage de programmation) donc les routines D3DX font partie intégrante de l’application. Un mécanisme de Microsoft appelé COM permet d’apporter d’autres fonctionnalités à DirectX par l’intermédiaire de DLL* (glossaire : Extension attribuée aux modules exécutables appelés par plusieurs applications en environnement Windows). Tous les objets manipulés dans un programme disposent d’une interface COM (Component Object Model).

Un objet COM est créé à l’aide de fonctions dont la nomenclature est le plus souvent Create suivi du nom de l’objet. Par exemple CreateDevice pour créer un objet Device. Le mécanisme COM se charge lui-même de détruire l’objet lorsque la durée de vie de celui-ci arrive à zéro. Il s’occupe également de charger les DLL ou autres modules nécessaires à l’objet.

Pour finir, le SDK DirectX de Microsoft fournit le code source d’un jeu de classes utilitaires appelé framework pouvant servir de base solide à la construction d’applications complètes.

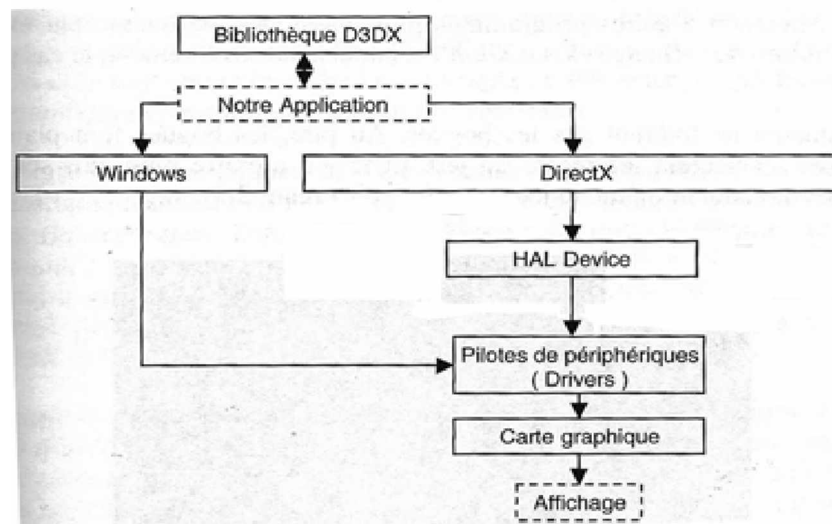


Figure 2 : Les composants de DirectX Graphics, interface avec le matériel

- Modélisation 3D – 3D Studio Max

Tous les personnages, décors et autres bonus ont été modélisés sous 3DS Max. Ce logiciel étant utilisé en cours, il était déjà connu et donc son utilisation a facilité la réalisation de tous les modèles 3D.

De plus, 3DS Max permet, grâce à un plug-in, d'exporter facilement les objets créés au format « .x », qui est le format d'importation sous DirectX. Cette fonctionnalité permet donc, de créer facilement de nouveaux objets, personnages et décors et de les exporter de manière à pouvoir les inclure dans un nouveau niveau du jeu.

Habituellement, ce logiciel est utilisé pour des modélisations ou des animations de grandes qualités. Cependant, les modèles du jeu sont d'aspect géométrique et font plus penser aux personnages des jeux vidéos d'il y a une dizaine d'années, qu'aux personnages actuels. Ceci est dû à une des contraintes principales du projet, le temps de calcul. Afin d'écarter tout risque de ralentissement lors de l'affichage des personnages, ces derniers ont été réalisés en low poly*. En effet le traitement nécessaire à l'affichage étant proportionnel au nombre de polygones, diminuer ce nombre a pour effet d'accélérer le rendu.

D. Organisation du groupe

Le projet à été décomposer en un ensemble de tâches. Celles si sont présente dans le GANT si après, ainsi que la durée de chacune d'elles et la répartition du groupe.

Au fur et à mesure de l'année, la durée allouée au début du projet à certaines tâches a été réévaluée avant de les commencer (indiquée en vert sur le GANT). D'autres ont nécessitées plus de travail que prévus (le débordement est indiqué en orange sur le GANT). Les périodes rentrant dans les prévisions sont indiquées en bleu sur le GANT.

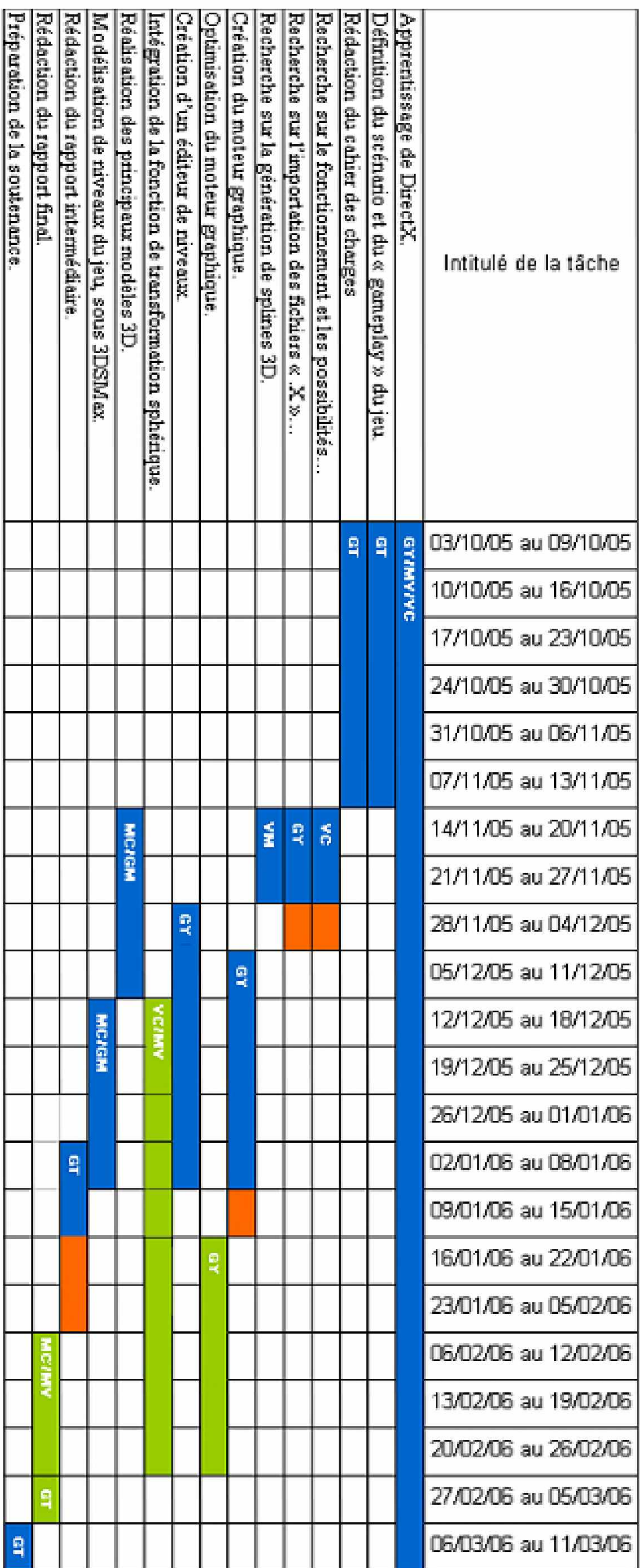


Figure 3 : GANT

Approche technique

A. Le jeu

- Le jeu

Pour programmer le jeu, le concept de la POO (Programmation Orientée Objet) a été utilisé. Ainsi l'application est régie par une classe *CMyD3DApplication* qui possède en données membres les informations globales au programme, comme par exemple, la liste des éléments à afficher, et en méthodes, les événements.

Tout d'abord, une classe *OrganicMesh* a été réalisée permettant l'importation et l'affichage de fichier au format « .X ». Sa création était indispensable, tous les éléments du monde (décor du niveau, entités, particules) étant affichés de cette façon.

A l'exception du moteur de particules, les entités du jeu dérivent toutes d'une classe de base *CEntity.* Une classe de base fille *CEntity3D* (elle-même dérivée en sous-classes) permet d'afficher des entités représentées par un modèle 3d (ennemis, bonus…). Une autre classe fille *CEntity2D* propose le rendu d'éléments 2d (c'est à dire qui restent toujours face au joueur) tels que les effets de flash provoqués par un tir d'arme.

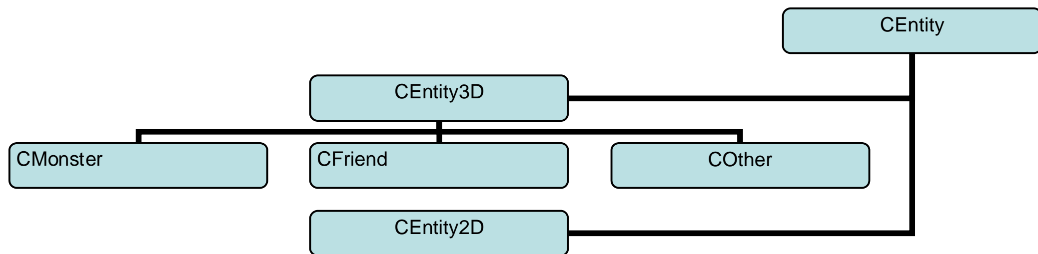


Figure 4 : Hiérarchie des classes

En tant que donnée principale, une *CEntity* possède une trajectoire, qui permet de définir sa position au cours du temps.

Chaque entité dérivée de *CEntity3D* possède ses particularités. En effet, un objet de type *CFriend* se déplaçant sur sa trajectoire, ajuste sa direction tout seul. Une entité *CMonster*, se positionnera toujours face au joueur pour lui tirer dessus, et possède une arme, objet de la classe *CWeapon*. Enfin, un *COther* a la particularité de gérer son orientation par rapport à des paramètres fixés par le créateur du niveau.

La classe *CWeapon* possédant de nombreux paramètres tels que la cadence de tir, les sons émis ou les dommages provoqués par les balles, permet de simuler les armes utilisées par les ennemis de l'environnement, et par le joueur.

Enfin, une classe *CParticuleManager* permet d'utiliser le moteur de particules créé pour le jeu, générant des objets du type *CParticule*. Ce système offre des effets visuels agréables, lors de tirs d'armes sur les murs, ou à la mort d'entités.

- L'éditeur

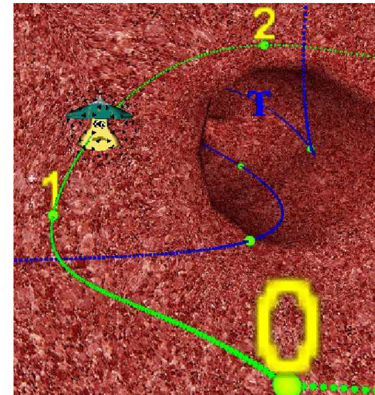


Figure 5 : Aperçue du mode éditeur

L'éditeur fonctionne vraiment de la même façon que le jeu, cela permettant une communication plus aisée entre ces 2 modes. Bien entendu, il n'exploite pas les effets graphiques de flash et de particules, qui encombreraient l'affichage et diminuerait alors les performances : en effet, l'affichage en mode édition est plus chargé que dans le mode jeu, car en plus des entités affichées, différentes propriétés de ces dernières sont visualisables pour pouvoir les paramétrer plus efficacement.

Ainsi, les trajectoires splines de toutes les entités sont affichées avec leurs points de contrôles numérotés. La sphère de collision de chacune d'entre elles est également représentée pour pouvoir les ajuster plus aisément au modèle 3d de l'entité. La navigation dans le niveau se fait alors librement, permettant au créateur de prendre tous les points de vue possibles, tout en contrôlant l'écoulement du temps, indispensable à la bonne coordination des entités dans le décor.

B. Projection

- Principe du Cubic Environment

Le but est de représenter la scène qui entoure un objet. Le cubic environment consiste à considérer l'objet comme le centre d'un cube. Chaque face du cube couvre un champ de vue de 90 degrés en horizontal et vertical. L'ensemble des six faces du cube permet de couvrir la totalité de la pièce.

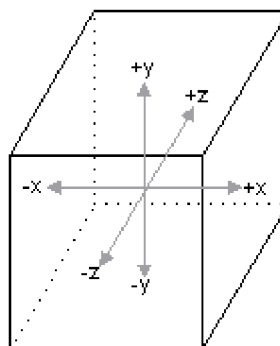


Figure 6 : Cube avec les axes centraux perpendiculaires aux faces

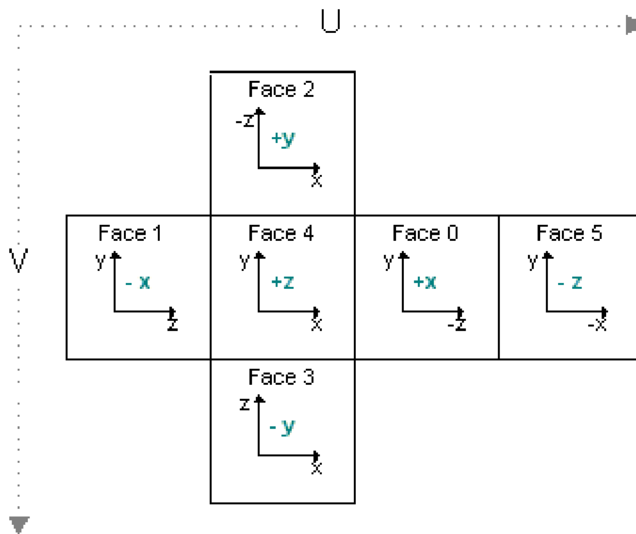


Figure 7 : Face du cube avec les coordonnées de chaque face

- <u>Principe de la projection</u>

Il suffit de récupérer les rendus correspondant à chaque face du cube puis de les recombiner pour former une seule image, un peu comme si le cube était déplié pour le mettre à plat. C'est cette image finale qui est projetée sur le miroir hémisphérique.

La fonction utilisée pour récupérer les images des faces du cube, travaille principalement avec les angles. Grâce à un calcul, elle détermine pour chaque pixel, la face à laquelle il appartient. (cf annexe X) De ce fait, cette méthode est adaptée à une pièce d'aspect cubique où il est facile de déterminer les angles limites de chaque mur.

Afin de mieux visualiser les étapes il est possible d'appliquer une couleur par face, ainsi la projection effectuée est celle d'une pièce ou chaque mur serait coloré de façon différente et uniforme.

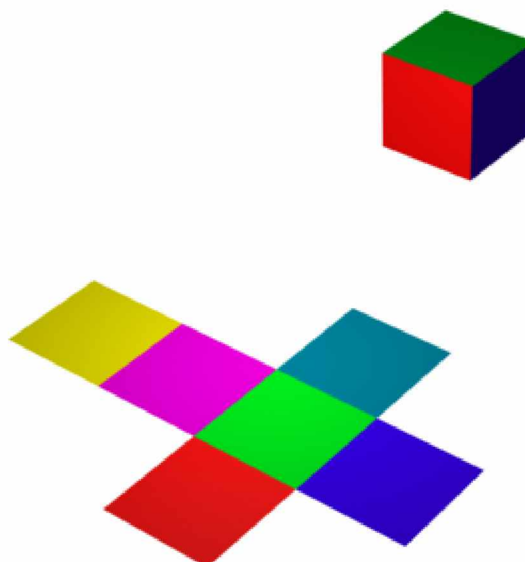


Figure 8 : représentation du cube et de son patron en couleur pour repérer les faces

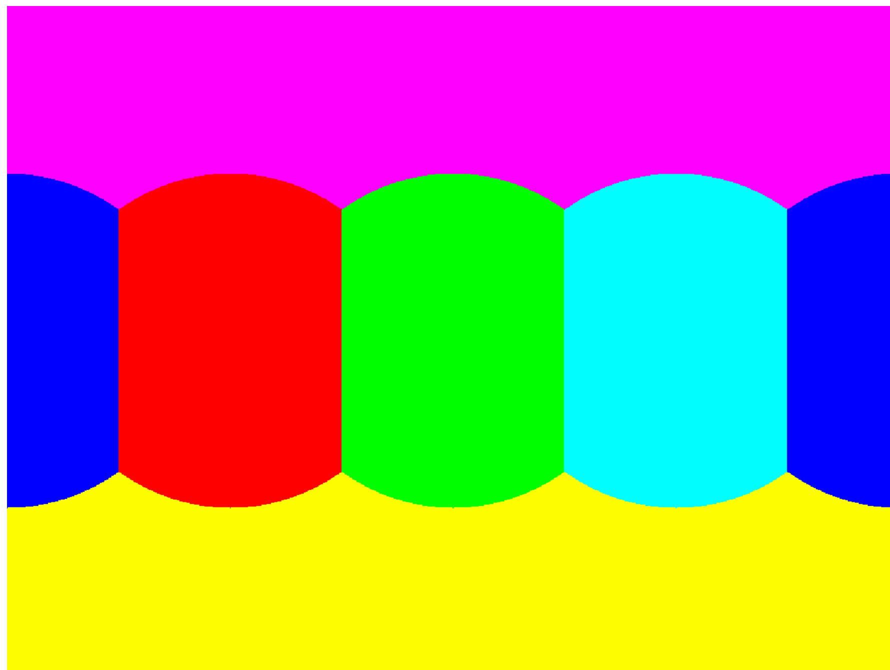


Figure 9 : Recombinaison des 6 faces

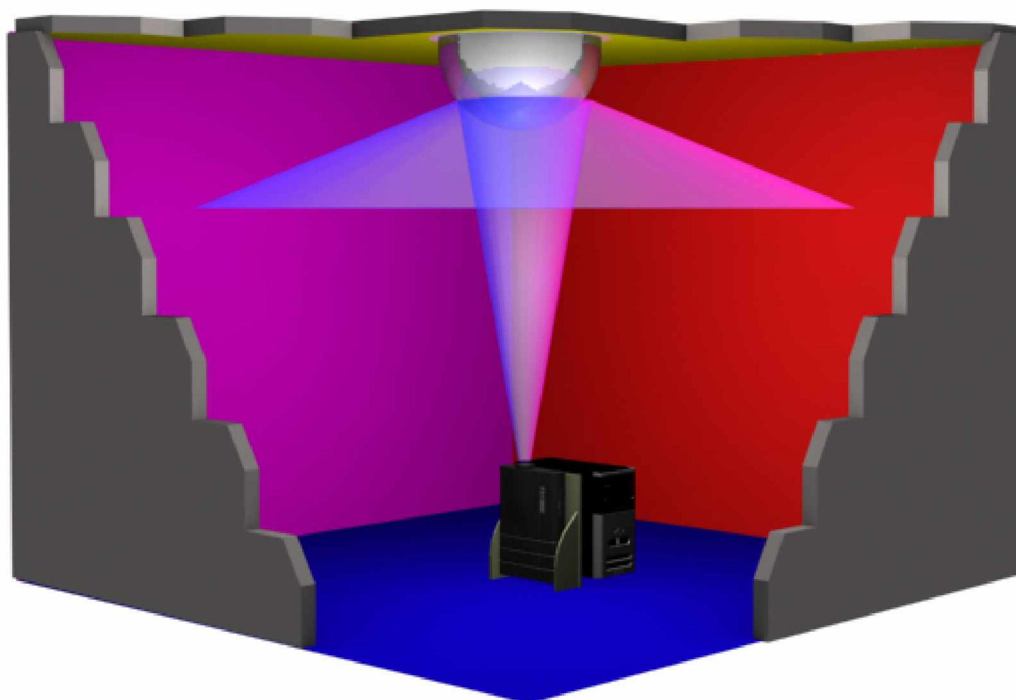


Figure 10 : Projection de l'image recombinée

Pour une pièce où les murs ne se coupent pas en angle droit, cette méthode montre vite ses limites et il vaut mieux travailler avec une fonction qui calcule les intersections des murs par un lancer de rayon par exemple.

- Implementation de la projection

L'algorithme de la projection peut se découper en deux fonctions principales, appelées à la suite pour toutes les frames* :

- La première fonction effectue les six rendus à partir de la position de la caméra.
- La deuxième récupère ces six rendus, pour les recombiner en un seul qui va être affiché à l'écran.

Explication rapide des fonctionnalités directX utilisées:

DirectX travaille avec un objet appelé Surface (**LPDIRECTSURFACE9**), cette surface est en fait un tableau de fragments* avec diverses propriétés. Il existe par défaut plusieurs surfaces rattachées au device. Elles servent de zones mémoires pour stocker les rendus. Les deux surfaces les plus importantes sont le FrontBuffer* et le BackBuffer*. Le premier étant la surface affichée à l'écran, le deuxième servant de zone de rendu avant d'être passé au FrontBuffer.

Lors de la création d'une surface, des flags* lui sont passés en paramètre. Ces flags sont des constantes qui définissent les propriétés de la surface, son format, etc.
Ces flags bien définis destinent la surface à une application spécifique et bloquent certaines fonctionnalités.

DirectX utilise aussi un outil, le CubeMap (**LPDIRECT3DCUBETEXTURE9**), un ensemble de six surfaces qui est rempli avec les rendus respectifs. Le CubeMapping est très utilisé pour les réflexions, par exemple en texturant un objet réfléchissant avec une texture tirée du CubeMap.
Chaque face est référencée par une constante DirectX.

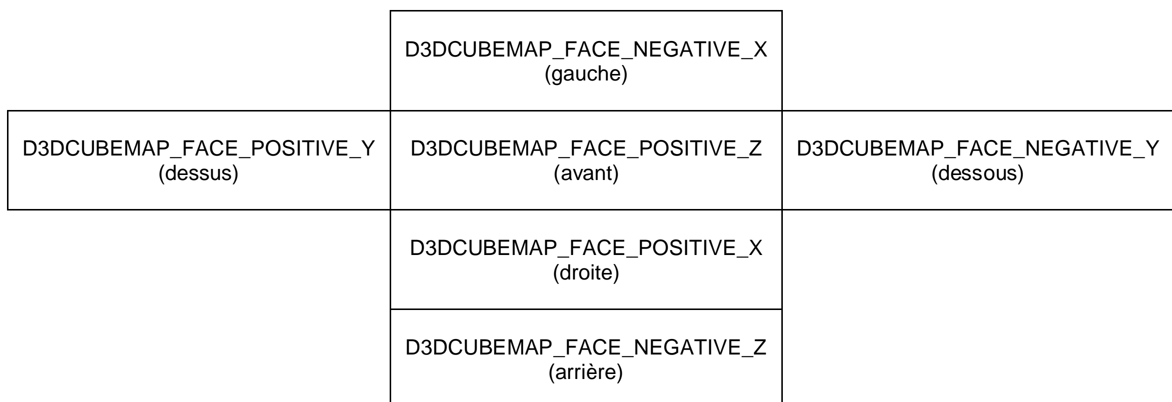


Figure 11 : Face du cube avec les constantes associées à chaque face

Détails des étapes :

- **Création des surfaces :**

Pour réussir la projection il est, en théorie, seulement nécessaire d'un CubeMap et d'une surface finale.

A l'initialisation il est demandé un flag de format : le flag **D3DFMT_R8G8B8** a été choisi. Comme son nom l'indique, les éléments de la surface seront codés sur 24 bits, 8 pour chaque composante (rouge, vert et bleu). D'autres formats existent comme le **D3DFMT_A8R8G8B8**, qui utilise aussi le coefficient alpha de transparence.

Un flag d'usage est aussi demandé, ce type de flag définit et donc limite les utilisations de la surface, ce qui pose problème dans le cas présent.

Pour pouvoir effectuer les 6 rendus, le flag doit être **D3DUSAGE_RENDERTARGET**, il indique la surface comme étant une zone de rendu possible. Cependant l'utilisation de ce flag implique que la surface soit stockée dans une zone mémoire dédiée à l'écriture et donc que toute lecture directe des données est impossible. Il existe d'autres flags équivalents qui bloquent l'écriture (**D3DUSAGE_READONLY** par exemple).

Lors de l'initialisation du CubeMap, 6 faces sont créées avec le flag défini pour le CubeMap. Pour pouvoir les exploiter, il faut créer un jeu de six pointeurs sur Surface qui pointeront sur les faces du cube.

```
Récapitulatif des variables obtenues

LPDIRECT3DCUBETEXTURE9  m_pCubeRenderMap;
//Un cubemap ( composé de six surfaces) crée en RGB qui sert de zone de rendu.

LPDIRECT3DSURFACE9 pCubeMapRenderFace[6];
//Les six surfaces associées au cubemap et qui ont les mêmes attributs.

LPDIRECT3DSURFACE9 pMapFinalRendu;
//Six surfaces en RGB qui sont des copies des faces du cube et qui servent pour l'accession aux
//données.
```

- **Récupérations des rendus :**

La récupération des rendus est effectuée dans la fonction *RenderSceneIntoCubeMap()*, appelée à chaque frame.

```
HRESULT RenderSceneIntoCubeMap()
{
      // Sauvegarde du contexte, c'est-à-dire les matrice de vue et de projection qui vont
      // être modifiées.

      // Récupération de  la matrice de vue associée à la camera
      D3DXMATRIXA16 matView=PLAYER->_pCamera->GetmatView();

      //Replissage des  faces du CubeMap
       for( UINT i = 0; i < 6; i++ )
       {
```

```
// Récupération  dans pCubeMapRenderFace[i]  de la surface associée à la face  numéro i.
m_pCubeRenderMap->GetCubeMapSurface( (D3DCUBEMAP_FACES)
      i,0,&pCubeMapRenderFace[i]);

//définition de la surface où doit s'appliquer le rendu
m_pd3dDevice->SetRenderTarget(0,pCubeMapRenderFace[i]);

//récupération de la nouvelle matrice de vue correspondant à la la face voulue
matView =D3DUtilGetCubeMapViewMatrixLH( (D3DCUBEMAP_FACES) i );

//rendu de la scène avec la bonne matrice de vue
RenderScene(&matView,&matProj,(int)i);
	}
	// Restauration du contexte
}
```

Résultat : Les 6 pCubeMapRenderFace de remplies avec les six rendus.

Ø **Recombinaison des six rendus :**

La fonction *Spherical_Recombinaison()* est utilisée pour recombiner les six rendus obtenus précédemment. Cette fonction reprend la méthode de calcul utilisée l'an dernier.

La fonction importante utilisée ici est *LockRect().* Elle permet à l'aide d'une variable **D3DLOCKEDRECT** d'accéder au tableau de bits qui compose une surface et ainsi de la modifier ou de la lire.

```
VOID Spherical_Recombinaison (double TetaMin, double TetaMax, double PhiMin,double PhiMax)
{
	// tableau de bit de l'image finale
	byte * imgout= new byte[WX*WY*3];

	// tableaux de bits des 6 surfaces du cubeMap
	byte **tabimg;
	tabimg= new byte*[6];

	char msg[256];

	HRESULT hr;

	RECT sourceRect = { 0,0, 255 , 255};

	//Recopiage des 6 pCubeMapRenderFace en mode de rendu dans les 6
	// pCubeMapFinalFace en mode lecture.
	// Cette fonction est longue et ralentit //beaucoup l'exécution du programme.

	for(int k=0;k<6;k++)
	{
		D3DXLoadSurfaceFromSurface(pCubeMapFinalFace[k],NULL,NULL,
			pCubeMapRenderFace[k],NULL,&sourceRect,D3DX_FILTER_NONE,0);

	}

	//Récupération de la taille des six images de rendus à l'aide d'un D3DSURFACE_DESC

	D3DSURFACE_DESC pDesc;
	pCubeMapFinalFace[0]->GetDesc(&pDesc);
	wx=pDesc.Width;
	wy=pDesc.Height;

	D3DLOCKED_RECT pLockedRect[6];
```

```
	for(int i=0;i<6;i++)
	{
		// initialisation des D3DLOCKED_RECT
		ZeroMemory( &pLockedRect[i], sizeof(D3DLOCKED_RECT));

		//"Lockage" des surfaces
		hr=pCubeMapFinalFace[i]->LockRect(&pLockedRect[i],NULL,0);

		if(hr==D3D_OK)
		{
			// Remplissage des tabimg avec le contenu des surfaces.
			tabimg[i]=new byte[wx*wy*3];
			memcpy(tabimg[i],pLockedRect[i].pBits, wx*wy*3*sizeof(byte));
		}

	}

	// Recombinaison des rendus, la structure du code est la même que celle de l'an dernier
	[...] Le code étant le même, il n'est pas réécrit ici.

	// "Unlockage" des 6 surfaces
	for(i=0;i<6;i++)
	{
		pCubeMapFinalFace[i]->UnlockRect();
	}

	RECT sourceRect2 = { 0,0, WX-1 ,WY-1};

	// Création de pMapFinalRendu avec le contenu du tableau imgout qui contient les images
	// recombinées.
	hr=D3DXLoadSurfaceFromMemory(pMapFinalRendu,NULL,NULL,imgout,D3DFMT_R8G8B8,0WX*3
		,NULL,&sourceRect2,D3DX_FILTER_NONE,0);

	// Récupération du backBuffer
	LPDIRECT3DSURFACE9 pBackBuffer;m_pd3dDevice>GetBackBuffer(0,0,
		DBACKBUFFER_TYPE_MONO,&pBackBuffer);

	// mise à jour à l'aide de la nouvelle surface recombinée
	hr=D3DXLoadSurfaceFromSurface(pBackBuffer,NULL,NULL,pMapFinalRendu,NULL,&sourceRect2,
		D3DX_FILTER_NONE,0);
}
```

- Performances :

Du fait des nombreux calculs, il s'ensuit que le jeu « rame » beaucoup. Cela est principalement du au remplissage du CubeMap, et surtout à la recopie des faces de rendu dans les faces de traitement, dans la fonction Spherical_Recombinaison.

Pour augmenter la fluidité il suffit de réduire la taille du CubeMap. Mais cela influe sur la qualité de l'image finale, et sur les « bordures » noires qui se trouvent à la délimitation correspondant aux changements de face. Plus le CubeMap est petit, plus la taille de ses bordures augmente. En effet il n'y a pas assez de pixels dans les six faces pour remplir entièrement l'image finale, les « trous » sont donc complétés avec du noir.

Taille du CubeMap	Vitesse du jeu	Bordures	Qualité d'image
512	Très lente (0à3 FPS)	Aucunes	Très bonne
256	Lente (6 à 10 FPS)	Légères	bonne
128	Moyenne (15 a 20 FPS)	moyennes	moyenne
64	Rapide (20 à 40 FPS)	Très marquées	médiocre

Le fait que le CubeMap doive avoir une taille en puissance de deux limite beaucoup les possibilités de réglages sur ce point, car les pas sont très élevés, et le changement d'une valeur affecte beaucoup le jeu.

Pour un jeu « jouable », une taille de 128 devra être prise, mais pour une bonne qualité 256 serait préférable si la puissance de l'ordinateur le permet.

C. Optimisation – Pixel Shader

- Explication

Le circuit intégré à une carte graphique repose sur un processeur dédié à la 3D, le GPU (Graphics Processing Unit). Celui-ci est capable de traiter suivant le modèle 128 ou 256 bits en parallèle et donc d'effectuer les calculs quatre ou huit fois plus rapidement qu'un processeur central classique 32 bits. Sur les cartes graphiques récentes il est possible d'envoyer au GPU des instructions proches de l'assembleur. Celles-ci permettent de court-circuiter les traitements standards. Ces instructions sont appelées vertex shaders, traitement sur les sommets, et pixel shaders, traitement sur les pixels et sont effectués très rapidement.

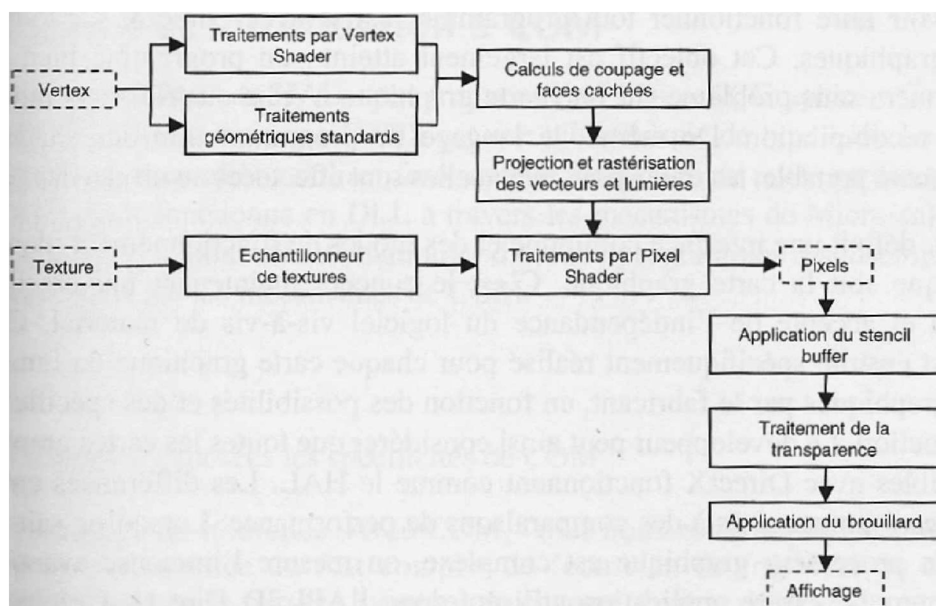


Figure 12 : Pipeline des traitements graphiques

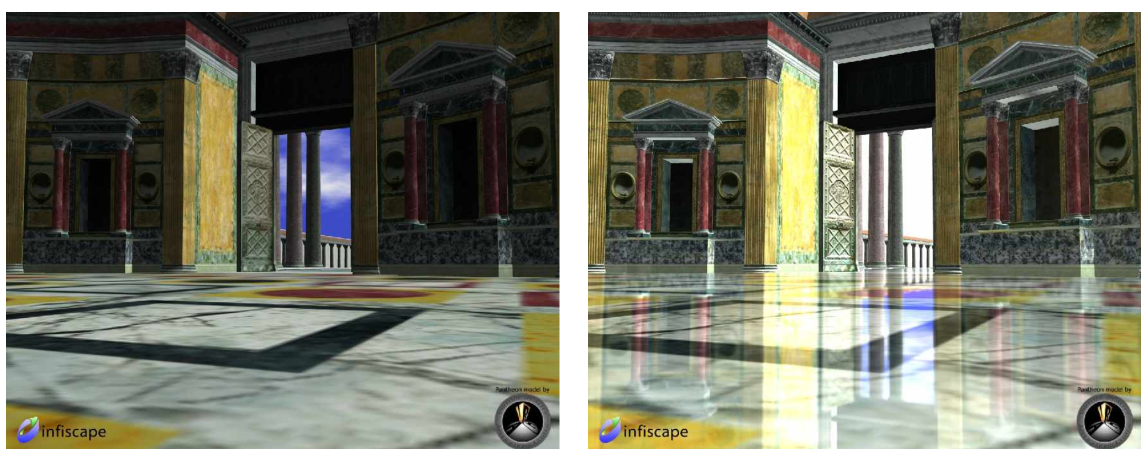


Figure 13 : Exemple d'utilisation des shaders pour faire de la réflexion

Les shaders sont très à la mode dans les jeux vidéo, ils permettent d'obtenir des effets visuels agréables avec un temps de calcul très réduit. Le Toon Shading est un exemple d'utilisation d'effet visuel possible.

- <u>HLSL et Assembleur</u>

L'utilisation des shaders nécessite l'apprentissage d'un langage proche de l'assembleur, difficile à mettre au point, complexe et peu documenté. Cependant l'apparition de langage de haut niveau comme le HLSL de Microsoft (High Level Shader Language) permet en théorie de s'affranchir du codage des shaders en assembleur. Malgré tout, ces langages de haut niveau font perdre de vue le fonctionnement interne des shaders, et rendent impossible certaines optimisations. Il semble donc impossible de coder du HLSL solide, sans de bonnes bases en assembleur.

<u>Exemple d'utilisation des vertex Shaders</u>

Les vertex shaders permettent de remplacer le traitement effectué par la carte graphique sur les vertices par une application personnalisée. Voici quelques exemples d'utilisations envisageables :

- Tweening, animation de personnages par interpolation de frames.
- Morphing, déformation et transformation d'objets ou de personnages.
- Modification dynamique des matrices de projection. Effet de grand angle, effet sous-marin ou de chaleur sur la route, loupe et effets optiques comme la pluie sur un pare-brise.
- Animation locale des objets. Feuilles d'arbre, herbe, cheveux ou vêtements.
- Systèmes de particules.
- Effet « aquatique », réflexion et diffraction à la surface de l'eau.
- Illuminations complexes.

<u>Exemple d'utilisation des pixels Shaders</u>

Les pixels Shaders permettent de prendre le contrôle de tous les traitements relatifs aux pixels, ce qui donne des possibilités illimitées. Il pourrait, même être envisagé de reprogrammer totalement la logique de fonctionnement de la carte graphique.
Voici quelques applications courantes :

- passage en noir et blanc ou sépia.
- effet de profondeur.
- brouillard volumétrique.
- ombres locales.
- convolutions et filtres, détection de contours.
- effets de bande dessinée.
- effets de peinture, crayon, hachures.
- calculs de textures en temps réel.

Il pourrait aussi être envisagé d'utiliser les pixels shaders pour effectuer les opérations nécessaires à la projection. Le processeur serait ainsi déchargé de ces traitements lourds et le résultat sur les performances de l'application serait nettement visible. Cependant, les instructions des pixels shaders sont soumises à certaines limitations :

- Limitation du nombre d'instructions d'échantillonnages de textures (6 au maximum en version 1.4)
- Limitation du nombre d'instructions arithmétiques (8 au maximum en version 1.4)

Donc pas plus de 14 lignes en version 1.4. C'est la limitation majeure des pixels shaders

- Pas plus d'une constante par instruction
- Pas plus de trois registres temporaires par instruction
- Impossibilité de modifier les entrées
- Impossibilité d'utiliser des sous programmes

A noter que le même type de contraintes existe pour les Vertex Shaders. Malgré tout, il serait bon d'envisager une étude plus approfondie des shaders dans l'optique d'un essai d'implémentation pour la projection. Une telle tentative se devrait d'utiliser le HLSL. Comme vu plus haut, c'est un langage qui aide à la réalisation de shaders. Il fait abstraction du niveau de shaders et prend en compte les contraintes du nombre d'instructions et autres. De plus la lisibilité du code est bien meilleure et il est réutilisable.

Conclusion

Le but du projet, créer un jeu vidéo sous DirectX, adapté au projecteur panoramique, est atteint. Le jeu créé se veut amusant et fonctionnel, prouvant que les calculs de projection sont possibles en DirectX. Cependant il est nécessaire d'émettre un bémol sur les performances.

En effet, l'algorithme de projection réalisé prend beaucoup de temps. Le jeu est plus lent que celui réalisé sous OpenGL. L'absence de connaissance sur DirectX au début du projet et la complexité des fonctions utilisées sont probablement la cause de cet inconvénient. Pour arriver à obtenir une projection de meilleure qualité, il faudrait soit utiliser d'autres fonctions de DirectX, soit essayer d'utiliser les Shaders.

Cette dernière méthode semble la meilleure, sachant qu'aujourd'hui, une grande majorité des jeux vidéo les utilisent. Savoir en programmer serait donc un réel plus pour un étudiant.

Voir le résultat du jeu vidéo une fois projeté, son aspect à la fois immersif et interactif a sûrement de quoi combler un très grand nombre de joueurs. C'est du moins notre avis. Et il nous est très facile d'imaginer qu'une fois finalisé, nous pourrions voir ce système à la fois au cinéma, chez des commerçants ou dans des salles d'arcades, voir même chez nous.

En tant que joueurs réguliers ce système nous paraît porteur d'une réelle évolution dans le domaine des jeux vidéo. De plus, étant donné que le calcul de la projection reste une fonction à part, une fois cette brique créée, développer de tels jeux ne poserait pas plus de difficultés que le développement d'un jeu "normal".

Bibliographie

Ouvrage:

TESTUD, Laurent, DirectX 9 Programmation de jeux 3D, CampusPress, 2005, 611

Site Internet

- Microsoft, MSDN en ligne, http://msdn.microsoft.com/directx

- C/C++ Codes Sources, http://www.cppfrance.com

- Club des développeurs, http://www.developpez.net/forums

Glossaire

3D pré calculée : toutes les images sont calculées avant la diffusion. Cela nécessite un temps de calcul très long et ne permet aucune interaction, mais donne des images ou vidéos très réalistes.

3D temps réel : toutes les images sont calculées dynamiquement en fonction des changements effectués dans l'environnement 3D (navigation, interaction…)

3D Studio Max : logiciel développé par Discreet Logic, société d'Autodesk. Programme de modélisation et d'animation 3D.

API (Application and Programming Interface) : Interface pour la programmation d'applications. C'est un ensemble de librairies permettant une programmation plus aisée car les fonctions deviennent indépendantes du matériel.

Bibliothèque graphique : composant utilisé pour la conception de logiciels qui regroupe différentes fonctions permettant la conception d'interfaces graphiques. Les bibliothèques graphiques communiquent directement avec le matériel et servent d'interface entre la carte vidéo, le clavier, la souris, etc et le programmeur.

Back buffer : surface de calcul en prévision de l'affichage dans le Front buffer.

Capteur 6 dimensions/1 position : Capteur de mouvement permettant de connaître la position et la direction visée.

Device : périphérique d'abstraction de la carte graphique

DirectX : bibliothèque propriétaire de Microsoft. Elle permet la programmation 3D depuis sa version 7 (la version utilisé dans Organic est la 9). Le terme DirectX regroupe Direct3D pour l'affichage 3D, DirectDraw pour l'affichage 2D, DirectInput pour les périphériques d'entrée, DirectPlay pour les échanges de données par réseau, DirectSound3D pour le son.

DLL : extension attribuée aux modules exécutables appelés par plusieurs applications en environnement Windows.

Flag : octet système spécial qui contient des indicateurs sur l'état du processeur.

FPS (First Personn Shoot) : style de jeu vidéo de tir à la première personne.

Fragment : carré de la grille de l'écran regroupant les valeurs chromatiques, de profondeurs et de textures d'un pixel.

Frame : image dont est composé un film ou une animation.

Front Buffer : Surface affichée à l'écran

LIB : extension de fichier de bibliothèque de données pour langage de programmation.

Low poly : expression utilsée à la place de low polygone qui exprime une méthode de modélisation utilisant un minimum de polygones.

MSDN : site Web et programme d'abonnement Microsoft, qui fournit aux développeurs des outils de développement Microsoft, des informations, des exemples de code et du matériel de formation les plus à jour.

OpenGL (Open Graphic Library) : bibliothèque d'outils de traitement graphique permettant de créer des programmes interactifs aussi bien 2D que 3D. C'est une bibliothèque libre multi-plateforme.

POO (programmation orientée objet) : méthode de programmation qui consiste à combiner au sein d'une même structure de données, appelée classe, les opérations et données.

Plug-in : programme informatique qui interagit avec un autre logiciel en lui fournissant de nouvelles fonctionnalités.

Rendu : processus de création d'une image en 2 dimensions à partir d'une scène en 3 dimensions.

SDK (Software Developpment Kit) : ensemble d'outils permettant aux développeurs de créer des applications de type défini.

Stéradians : unité du système international de l'angle solide. L'angle solide étant l'analogue tridimensionnel d'un angle dans le plan. 4π stéradians équivaux à une sphère complète.

Toon shading : effet visuel donnant l'impression que les images sont dessinées.

Vertices : (pluriel de vertex) En géométrie, coin d'un polygone ou d'un polyhedron.

Annexes

Annexe 1 : Manuel d'utilisation de l'éditeur

Introduction :

L'éditeur de *Organic* est compris dans le même programme que le jeu. C'est-à-dire que l'on peut passer du mode « *Jeu* » conventionnel au mode « *Edition* », simplement, et à n'importe quel moment, en appuyant sur la touche « M » du clavier. Votre niveau en cours de création peut donc être testé immédiatement dans le jeu, et ainsi les ajustements de déplacements ou de timing deviennent très rapides à réaliser par le créateur.

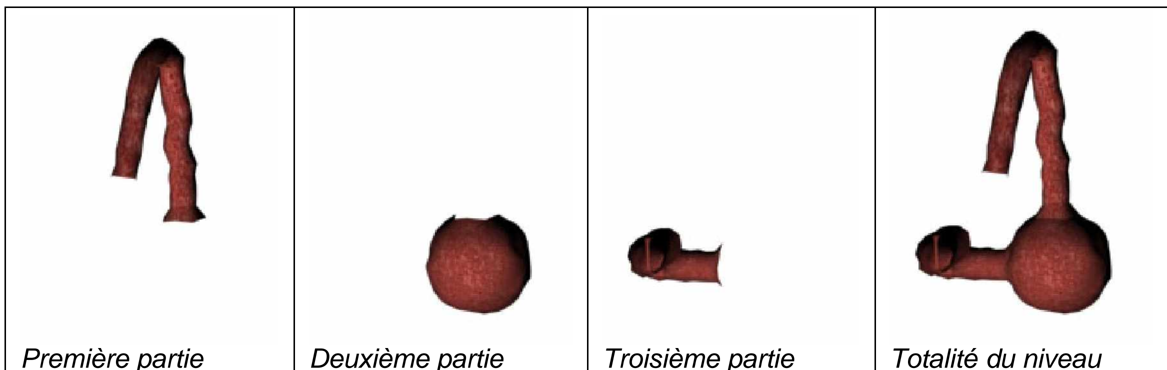
Avant de commencer :

Tous les fichiers spécifiques aux entités, c'est-à-dire tous les éléments autres que les décors (monstres, amis, bonus, autres objets en déplacement) se trouvent dans le dossier « *…/data/entities/* » du jeu. Nous verrons plus loin dans ce tutorial comment modifier ou même créer des entités comme bon vous semble.

Tous les fichiers spécifiques à votre niveau se trouvent dans le dossier « *…/data/maps/nomdevotremap/* ». Il y a donc un sous-dossier par niveau dans le dossier « *…data/maps/* ».

Création du décor :

Commencez donc par créer un sous-dossier dans le dossier « *…data/maps/* » qui porte le nom de votre niveau. A l'intérieur, il va falloir stocker les fichiers 3d du décor. Avant de commencer à créer le décor, il faut savoir que *Organic* travaille en « morceaux » de décor, et que seules les parties adjacentes à la partie où se trouvera le joueur seront calculées. En d'autres termes, si le joueur se situe dans le morceau n°3, seuls les morceaux n° 2, 3 et 4 seront calculés.



Première partie | *Deuxième partie* | *Troisième partie* | *Totalité du niveau*

Organic exige que cette décomposition en morceaux soit réalisée dans les fichiers 3d. C'est-à-dire qu'un fichier 3d correspondra à un morceau. Les fichiers 3d sont du format « .X », le format de DirectX. Ils doivent être modélisés à l'aide d'un logiciel 3d, comme *3ds Max* par exemple, et exportés au format « .X » (voir l'exporteur « *Panda Exporter* » pour *3ds Max*). Les fichiers doivent porter le nom « *numdumorceau.X* ». Ainsi le premier morceau du décor sera dans le fichier « *01.X* » (pensez à bien mettre le « 0 » devant pour la première dizaine), et le 13eme morceau dans le fichier « *13.X* ».

Une fois le décor réalisé et stocké correctement dans les fichiers « .X » dans le sous-dossier correspondant à votre niveau, vous pouvez déjà l'exécuter dans le jeu.

Création de la trajectoire du joueur :

Dans l'état actuel, votre nouveau niveau n'est pas très jouable. En effet, vous remarquerez que le joueur reste à l'origine du monde sans bouger. C'est normal puisque qu'il n'a pas encore de trajectoire prédéfinie… Qu'à cela ne tienne nous allons lui en créer une afin qu'il puisse se dégourdir les jambes.

Passez du mode « *Jeu* » au mode « *Edition* » en appuyant sur la touche « M » de votre clavier. Vous pouvez à présent naviguer comme bon vous semble dans le niveau à l'aide des touches directionnelles et de votre souris pour vous diriger.

A gauche de votre écran sont indiquées toutes les informations utiles à l'élaboration de votre niveau, entre autres des informations sur l'entité en elle-même (que l'on verra plus loin dans ce tutorial), et sur sa trajectoire…

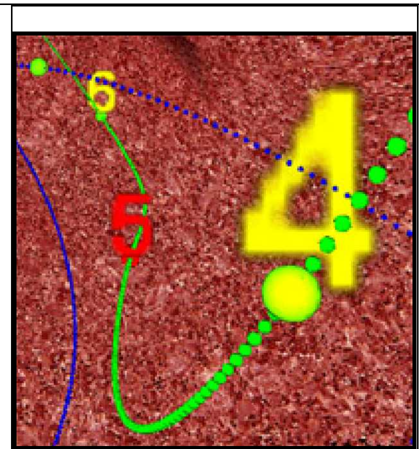
Les « *Infos Trajectoire* » correspondent à la trajectoire de l'entité qui est sélectionnée.
Par défaut, comme il n'y a pas d'entités autres que le joueur dans le niveau, vous visualisez donc les informations concernant la trajectoire du joueur.
Chaque trajectoire possède n points de contrôles (par défaut $n = 0$).

Infos Trajectoire :
- 9 Points de controles
- Point selectionne : ID 3 (Switch W/X)
- Type : Spline ouverte (Switch K)
- Temps actuel : 0.853124
Ajouter un point avant (A)
Ajouter un point apres (Z)
Supprimer le point selectionne (C)
Deplacer le point selectionne (T)

« Infos Trajectoire »

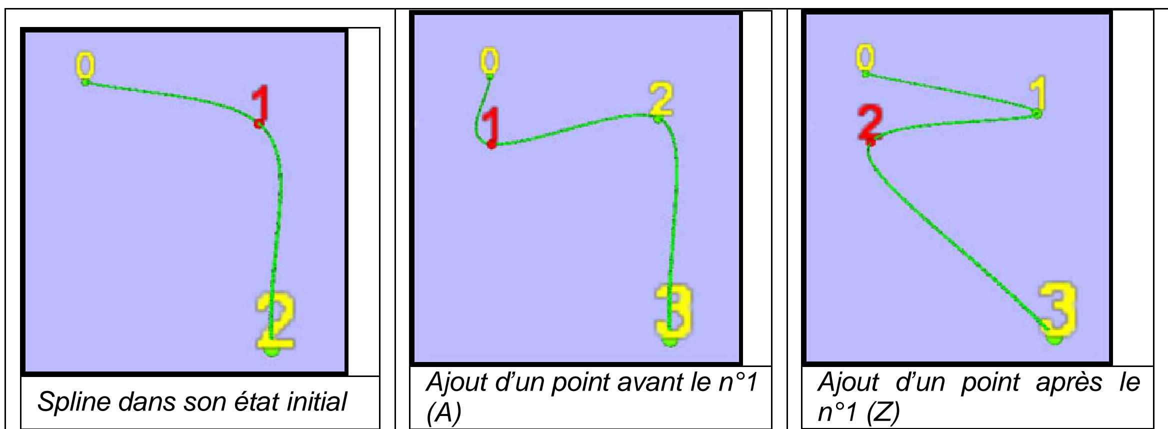
Pour ajouter un point de contrôle à la position où se situe la caméra, vous pouvez appuyer sur la touche « A » ou « Z ». « A » ajoute un point de contrôle avant le point de contrôle couramment sélectionné, et « Z » l'ajoute après.

La sélection des points de contrôles quand à elle, se fait aussi très facilement : « W » pour sélectionner le point de contrôle précédent, et « X » pour sélectionner le suivant. Pour plus de clarté, l'éditeur affiche en permanence le numéro du point de contrôle sélectionné à gauche, dans les « *Infos Trajectoire* », et sur la spline : le numéro sélectionné apparaît en rouge surbrillant.



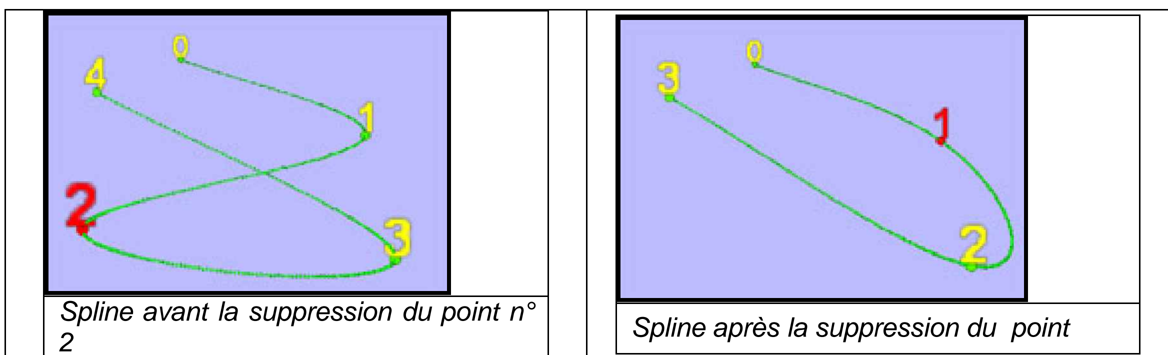
Le point n° 5 est sélectionné

Ainsi, si le point de contrôle sélectionné est le n° 1, appuyer sur « A » ajoutera un point de contrôle qui prendra place entre le n° 0 et le n° 1, et appuyer sur « Z » en ajoutera un entre le n° 1 et le n° 2.



Spline dans son état initial	*Ajout d'un point avant le n°1 (A)*	*Ajout d'un point après le n°1 (Z)*

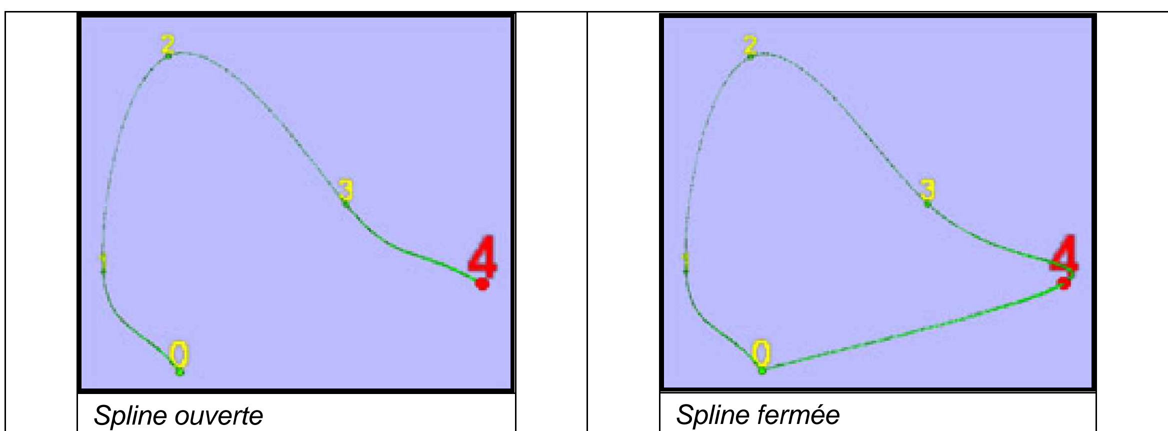
Pour supprimer le point de contrôle qui est sélectionné, la touche est le « C ». Bien entendu ici encore, le raccordement de la spline avec les points de contrôles restant s'effectue automatiquement, même en supprimant un point de contrôle qui se trouve en milieu de spline.



Spline avant la suppression du point n° 2	*Spline après la suppression du point*

Chaque point de contrôle a une position spécifique dans l'espace. Mais leurs positions peuvent être modifiées. En appuyant sur la touche « T », vous pouvez entrer en mode de « *Déplacement de point de contrôle* ». Les touches « Z/Q/S/D/-/+ » servent alors à déplacer le point de contrôle sur les différents axes. La touche « T » permet alors de quitter le mode de déplacement. A l'aide de cet outil, les ajustements de trajectoire deviennent très aisés.

Enfin, les splines peuvent être soit ouvertes, soit fermées. Pour modifier ce paramètre, il suffit d'appuyer sur la touche « K ». Dans le cas d'une spline fermée, l'entité bouclera sa trajectoire. Dans le cas d'une spline ouverte, l'entité recommencera au début si elle arrive au bout. Il n'y a par contre aucun intérêt à fermer la spline de trajectoire du joueur…



Spline ouverte	*Spline fermée*

Gestion des entités :

En ce qui concerne les entités, le type de l'entité sélectionnée est indiqué. Vous pouvez changer votre sélection et parcourir toutes les autres entités à l'aide de « Q » et « S ». Comme votre niveau est nouveau, il n'y a que le joueur comme entité présente actuellement.

Entite Selectionnee (Switch Q/S) :
- Type : "Joueur"
- Nom : "spline du joueur" (N pour renommer)
- Vitesse : 0.700000 (V pour la modifier)

Ajouter une entite (E)
Supprimer l'entite selectionnee (D)

Gestion des entités

Vous pouvez aussi choisir de supprimer une entité avec la touche « D », qui efface l'entité sélectionnée. L'entité « Joueur » est la seule à ne pas pouvoir être supprimée.

Chaque entité possède un nom qui sert juste d'information pour le créateur. Vous pouvez le changer (le nom, pas le créateur :p) en appuyant sur « N », en tapant son nouveau nom et en appuyant sur « Entrée » pour achever la saisie.

Renommage de l'Entite :

Ancien nom : "spline du joueur"
Entrez le nouveau nom :

Modification du nom de l'entité

Chaque entité a une vitesse, qui correspond à la vitesse de déplacement sur la trajectoire que l'on définit. Pour la modifier, il suffit d'appuyer sur la touche « V », de taper la nouvelle valeur et d'appuyer sur « Entrée » pour achever la saisie.

Vitesse de l'Entite :

Ancienne vitesse : 0.700000
Entrez la nouvelle vitesse :

Définition d'une nouvelle valeur de vitesse

La touche « E » est très importante : elle sert à ajouter des entités. Vous pouvez ajouter tout types d'entités, sauf l'entité « Joueur » qui doit rester unique dans le niveau. Presser « E » vous conduit à un menu qui vous permet de choisir l'entité à ajouter. Faites votre choix en tapant le numéro souhaité, puis « Entrée » pour valider. Votre nouvelle entité est sélectionnée par défaut, et apparaît à l'origine du monde : en effet, elle ne possède pas encore de trajectoire associée. Définissez lui une spline, comme nous l'avons fait pour le joueur, et votre entité se placera au bon endroit automatiquement.

Choix d'une Entite :

0 - Gentil_Anticorps
1 - Mechant_Baton
2 - Mechant_Champignon
3 - Bonus_+_100_de_vie
4 - Autre_Bloc_Rond_01
5 - Arme_ERRADICATOR

Votre choix :

Choix de l'entité à ajouter

A noter aussi que en plus de l'affichage de l'entité, le mode Edition affiche la Sphère de collisions, que l'on verra plus tard.

Contrôle du temps :

Pouvoir gérer le temps est un outil qui vous facilitera bien la vie dans l'élaboration de niveaux de *Organic*, vous permettant de synchroniser les entités en les ajustant petit à petit.

Chaque entité possède sa spline, et sa position sur cette trajectoire est dépendante d'une variable de temps, que vous pouvez consulter à tout moment pour l'entité sélectionnée, dans les « *Infos Trajectoire* ». Cet indice varie entre 0.0 et 1.0.

Vous pouvez donc agir sur le temps comme avec un magnétoscope. Ce contrôle est global, c'est-à-dire qu'il influe sur toutes les entités du niveau. Il n'est pas possible de ne faire bouger qu'une seule entité en laissant les autres en « pause » : ceci serait contraire à notre convention de synchronisation des entités…

Les contrôles sont très simples : « I » pour remettre le temps à zéro (position initiale), la touche « O » pour retour rapide, et « P » pour avance rapide. Bien entendu, vous pouvez aussi à tout moment lancer le mode jeu avec « M » pour voir ce que donne réellement votre dernière modification.

Théoriquement donc, le jeu ne peut pas se désynchroniser. Cependant, dans certaines situations, des erreurs de synchronisation peuvent apparaître, nécessitant de remettre le temps à sa position initiale pour que tout soit bien recalé.

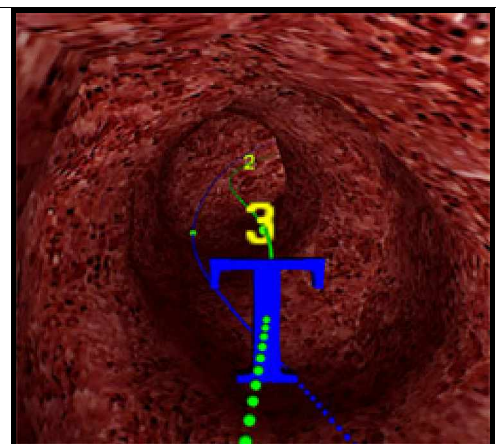
Ces erreurs sont inévitables. En effet, imaginez tout simplement la situation où vous ajoutez une entité alors que le temps global n'est pas à zéro. Une fois après avoir défini la trajectoire de cette nouvelle entité, vous faites avancer le temps afin de pouvoir la synchroniser avec les autres éléments du monde… Erreur ! Dans ce cas ci, votre entité aura commencé à se déplacer à partir du moment où vous avez repris l'avancée du temps. Or, l'entité aurait du commencer à bouger dès le début du jeu, en même temps que le joueur…

Afin d'éviter des erreurs de ce type, pensez à remettre le temps à zéro lors d'ajouts d'entités ou de modifications considérables de trajectoires.

<u>Chargement en multi-parties</u> :

Comme on l'a vu précédemment, les niveaux de *Organic* fonctionnent en « morceaux ». Pour savoir dans quelle portion du niveau le joueur est situé, la trajectoire du joueur doit être « découpée ». Pour cela, on utilise un système de marquages appelés « *Tmarks* » que l'on place sur sa spline. A chaque fois que le joueur franchit une *Tmark*, il passe dans le morceau suivant (ou précédent, ça dépend s'il avance ou s'il recule...).

Pour placer une *Tmark*, commencez par positionner le joueur au bon endroit (soit en stoppant le jeu à l'endroit voulu avec « M », soit en avançant/reculant le temps en mode « Edition » en visualisant le positionnement de la caméra symbolisant le joueur). Ensuite, il suffit d'appuyer sur la touche « H » pour placer une *Tmark*. Un « T » s'affiche à la position retenue, sur la spline du joueur : à partir de cette endroit, le joueur passera dans la zone suivante. Vous pouvez vérifier en faisant varier le temps, et en scrutant l'indice de « Zone en cours » affiché en bas à gauche de l'écran, en mode « Edition ».



« Tmark » sur la spline du joueur

Faites de même pour les autres jonctions de parties, et la segmentation de la spline joueur sera réalisée. Si une *Tmark* est mal placée selon votre goût, vous pouvez choisir de supprimer la *Tmark* en cours (par rapport à la position actuelle du joueur symbolisé par le logo de caméra) avec la touche « J ».

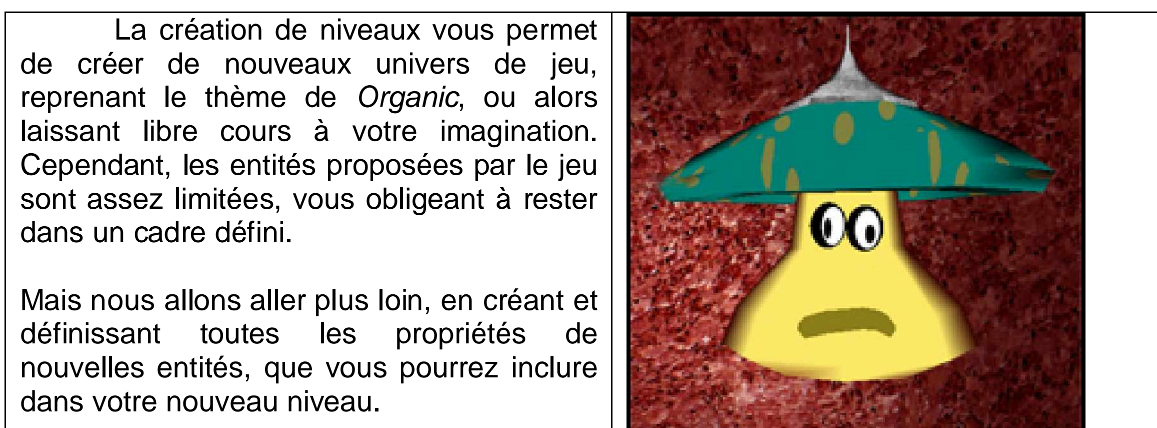
Attention ! Les *Tmarks* sont des informations de temps, et non de position dans l'espace. C'est-à-dire que faire varier la spline du joueur peut faire varier la position des *Tmarks*. Pensez donc à positionner vos *Tmarks* correctement une fois que la spline du joueur sera achevée, et que vous n'aurez plus besoin d'y retoucher.

Hormis le joueur, chaque entité appartient à un morceau du niveau, et c'est au créateur de le définir. Même si vous créez un nouveau monstre dans la partie finale de votre niveau, ce dernier appartiendra à la partie n° 0 par défaut. Ainsi votre entité jouera seulement lorsque le joueur se trouvera dans les parties n° 0 et 1, et non dans la partie souhaitée.

Vous pouvez consulter, pour chaque entité (sauf pour le joueur) le champ « Partie » dans les informations sur l'entité sélectionnée, sous sa vitesse. Par défaut étant à 0, vous pouvez faire varier ce numéro de partie à l'aide de la touche « L ».	Entite Selectionnee (Switch Q/S) : - Type : "Gentil_Anticorps" - Nom : "anticorps premiere salle" (N pour renommer) - Vitesse : 0.200000 (V pour la modifier) - Partie : 2 (L pour la modifier) Ajouter une entite (E) Supprimer l'entite selectionnee (D) *Cette entité appartient à la partie n° 2*

Ainsi, grâce à ce système, vos entités pourront commencer à vivre à partir du moment où le joueur arrivera à proximité d'elles, et elles pourront disparaître lorsqu'elles ne seront plus dans le champ d'action.

Créez vos propres entités :



La création de niveaux vous permet de créer de nouveaux univers de jeu, reprenant le thème de *Organic*, ou alors laissant libre cours à votre imagination. Cependant, les entités proposées par le jeu sont assez limitées, vous obligeant à rester dans un cadre défini.

Mais nous allons aller plus loin, en créant et définissant toutes les propriétés de nouvelles entités, que vous pourrez inclure dans votre nouveau niveau.

Tout les fichiers spécifiques aux entités se trouvent dans le dossier « *…/data/entities/* ». Une entité est composée au minimum d'un fichier 3d au format « .X » et d'un fichier texte indiquant ses propriétés, portant l'extension « *.ent* ». D'autres fichiers média peuvent être rattachés ensuite à l'entité, tels que des fichiers sons, effets ou particules, mais ceci n'est pas imposé.

Pour créer le fichier 3d, il n'y a pas de problèmes, le principe est le même que pour la création du niveau. Créez votre monstre ou votre personnage avec votre modeleur favori, et exportez-le au format « .X » dans le dossier des entités.

En ce qui concerne le fichier de propriétés, s'agissant d'un fichier texte, vous pourrez le créer et l'éditer avec n'importe quel éditeur de texte, (vive le Bloc-Notes !!).

Voici la liste des propriétés pouvant être spécifiées à chaque entité :

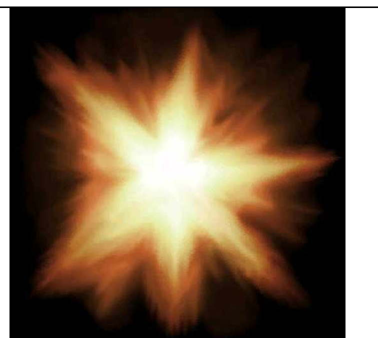
Nom de variable	***Description***	***Affecté aux entités***	***Valeur attendue***
NAME	Nom de l'entité	tous	chaîne de caractères sans espaces
DESC	Description de l'entité	tous	chaîne de caractères sans espaces
TYPE	Type de l'entité	tous	MONSTER, FRIEND ou OTHER
MODEL	Fichier du mesh à afficher	tous	chaîne de caractères sans espaces
SCALE	Echelle	tous	flottant
BOUNDINGSPHERE	Rayon de la bounding-sphere	Tous	flottant
LIFE	Nombre de points de vie	Tous	entier
MONSTERDAMAGE	Points de vie retirés au joueur touché	MONSTER seulement	entier
MONSTERCADENCY	Cadence de tir	MONSTER seulement	flottant
MONSTERFIREEFFECT	Effet graphique lors du tir du monstre	MONSTER seulement	Chaîne, flottant, flottant *(Nom de l'image, scale, alpha)*
MONSTERFIRESOUND	Son de tir du monstre	MONSTER seulement	chaîne de caractères sans espaces
PAINSOUND	Son émit quand elle est touchée	Tous	chaîne de caractères sans espaces
DEATHSOUND	Son émit quand elle meurt	Tous	chaîne de caractères sans espaces
ROTSPEEDX	Vitesse de rotation sur l'axe X	OTHER seulement	Flottant
ROTSPEEDY	Vitesse de rotation sur l'axe Y	OTHER seulement	Flottant
ROTSPEEDZ	Vitesse de rotation sur l'axe Z	OTHER seulement	Flottant
BONUSTYPE	Type de Bonus	Tous	SCORE, LIFE, AMMO, WEAPON ou SHIELD
BONUSVALUE	Valeur du Bonus	Tous	Flottant
BONUSWEAPONPROPS	Propriétés de l'arme Bonus	Seulement avec WEAPON en BONUSTYPE	chaîne, entier, flottant, flottant, entier, entier, chaîne *(Nom, Dommages, Précision, Cadence, Munitions, Auto, Fichier son)*
BONUSWEAPONFIREEFFECT	Effet graphique de l'arme Bonus	Seulement avec WEAPON en BONUSTYPE	Chaîne, flottant, flottant *(Nom de l'image, scale, alpha)*
PARTICULEMODEL	Propriétés des particules émises par l'entité	tous	Chaîne, flottant, flottant *(Nom du modèle 3d, scale, alpha)*

Quelques commentaires complémentant ce tableau seraient bien venus :

- Tout d'abord, que signifie le « *nom* » d'une entité ? Le nom est la propriété la PLUS IMPORTANTE de l'entité. En effet, c'est grâce à ce nom que le système va savoir quelle entité il devra ajouter au lancement du jeu.
Par exemple, vous créer une nouvelle entité portant le nom « *super_alien* » et vous l'ajoutez dans un ou plusieurs niveaux. Vous avez alors tout à fait le droit de modifier les propriétés du votre entité après (sauf le nom bien entendu), telles que son échelle (*SCALE*) ou son type (*TYPE*), mais aussi le nom du fichier « .X » à charger (*MODEL*). Vous avez même le droit de modifier le nom du fichier « .ent » en le renommant comme vous le souhaitez. Vous pouvez modifier absolument toutes les propriétés de votre entité, du moment que son nom reste constant, et que le moteur pourra trouver une entité portant le nom « *super_alien* » au démarrage du niveau.

- Le paramètre « *BOUNDINGSPHERE* » définit le rayon de la sphère de collision de l'entité. Toute entité a une sphère de collision (sauf le joueur), qui lui permet de se faire « toucher » par les tirs ennemis. Normalement, la *Bounding Sphere* doit coller au mieux la géométrie 3d de l'entité, afin d'éviter l'agacement du joueur qui a beau vider un chargement sur son ennemi, ne parvient pas à le tuer car sa *Bounding Sphere* est trop petite...

- Dans les paramètres de fichiers audio, tels que « *MONSTERFIRESOUND* », « *PAINSOUND* », « *DEATHSOUND* » ou le dernier paramètre d'une arme, il faut rentrer le nom du fichier son, avec son extension (« *tir.wav* » par exemple) que vous souhaiterez exécuter pour l'action définie. Ce fichier son doit se trouver dans le dosser « *data/sounds/* » du jeu.

- Les « effets » sont les éclairs lumineux qu'émettent les ennemis et le joueur lorsqu'ils tirent. Les paramètres tels que « *MONSTERFIREEFFECT* » ou « *BONUSWEAPONFIREEFFECT* » proposent de modifier respectivement l'effet provoqué par le monstre défini, ou par l'arme en bonus récoltée par le joueur. Il faut tout d'abord rentrer le nom du fichier image, avec son extension (« *halo.tga* » par exemple) qui sera affiché pour l'action définie.	

Cette texture peut posséder des zones transparentes (voir la couche alpha d'un TGA en 32 bits) pour un meilleur rendu. Ce fichier image doit impérativement se trouver dans le dossier « *data/effects/* » du jeu. Ensuite le paramètre « *scale* » correspond à la taille d'affichage de l'effet, et « *alpha* » définit la valeur de transparence (entre 0.0 et 1.0) de l'effet à sa naissance. En effet, les effets disposent d'un système de disparition progressive, faisant ainsi varier leur opacité de « *alpha* » à 0.0.

- Le moteur de particules intégré dans le jeu permet de générer un grand nombre de débris en temps réel, disposant chacun de ses propriétés (taille, vitesse, accélération...). Le paramètre « *PARTICULEMODEL* » permet de définir les propriétés des particules associées à l'entité, c'est à dire les débris rejetés par l'entité lorsque qu'elle se fait toucher par une balle, et lorsqu'elle meurt. Tout comme pour les effets, il faut d'abord rentrer le nom du fichier graphique, qui est un fichier 3d au format « *.X* » pour les particules (« *metal_debris.X* » par exemple) qui doit se situer dans le dossier « *data/particules/* » du jeu. Ensuite le paramètre « *scale* » correspond à la taille moyenne des particules, et « *alpha* » définit la valeur de transparence (entre 0.0 et 1.0) lors de la naissance des particules, car ici aussi, les particules disposent d'un effet de disparition progressive lors de leur mort.

Voici quelques exemples d'entités type :

Entité ennemie :

Elle vous tire dessus, ce qui réduit votre niveau de vie. Quand vous lui tirez dessus, elle éjecte des particules de sang. Si vous la tuez, vous engrangez des points supplémentaires au niveau du score.

Code :	***Commentaires :***
`NAME ennemi_alienvolant`	Nom (ne doit pas être changé une fois sur l'autre).
`DESC Alien_Volant`	Description affichée dans l'éditeur.
`TYPE MONSTER`	C'est un ennemi : *MONSTER*.
`MODEL alienvolant.X`	Fichier 3d à charger dans le dossier « *data/entities/* ».
`SCALE 0.2`	Rapport d'échelle.
`BOUNDINGSPHERE 0.3`	Taille de la sphère de collision.
`LIFE 100`	Niveau de vie à sa naissance.
`BONUSTYPE SCORE`	Si le joueur la tue, il engrange un bonus affecté au score.
`BONUSVALUE 500`	Le joueur engrange 500 points.
`MONSTERDAMAGE 2`	Pour chaque tir réussi, la cible de l'entité perd 2 de vie.
`MONSTERCADENCY 0.15`	Tire une balle toutes les 0.15 secondes.
`MONSTERFIREEFFECT muz.tga 1.5 1.0`	Affiche « muz.tga » de taille 1.5, opaque (1.0 en alpha), lorsqu'elle tire.
`MONSTERFIRESOUND tir.wav`	Son joué lorsqu'elle tire.
`DEATHSOUND mort.wav`	Son joué lorsqu'elle meurt.
`PAINSOUND touche.wav`	Son joué lorsqu'elle se fait toucher par une balle.
`PARTICULEMODEL blood.X 0.05 1.0`	Perd du sang opaque (1.0 en alpha) de taille 0.05.

Entité amie :

Elle ne s'occupe de personne, elle trace sa route en regardant devant elle. Son nom : Alien de Secours. C'est un gentil alien qui possède une très grande résistance aux balles. Cette bestiole est muette, nous ne lui spécifierons donc aucun son de douleur ou de mort. Mais si vous la tuez, cela ne vous enlève pas de points au score : elle vous redonne quelque points de vie !

Code :	***Commentaires :***
`NAME ami_aliensecours`	Nom .
`DESC Alien_de_Secours`	Description affichée dans l'éditeur.
`TYPE FRIEND`	C'est un ami : *FRIEND.*
`MODEL aliensecours.X`	Fichier 3d.
`SCALE 2.2`	Rapport d'échelle.
`BOUNDINGSPHERE 0.25`	Taille de la sphère de collision.
`LIFE 400`	Très grande résistance aux balles.
`BONUSTYPE LIFE`	Si le joueur la tue, il gagne de la vie.
`BONUSVALUE 50`	50 points de vie accordés.

Arme :

Une arme placée dans le niveau, qui tourne sur elle même. Si le joueur tire une balle dessus, l'arme lui revient dans les mains. C'est un « Super Laser », avec pleins de munitions, qui cause beaucoup de dommages, mais qui n'est pas automatique et qui ne possède pas une cadence de tir très rapide

Code :	Commentaires :
NAME arme_superlaser	Nom.
DESC Super_Laser	Description affichée dans l'éditeur.
TYPE OTHER	C'est ni un ennemi, ni un ami : *OTHER*.
MODEL superlaser.X	Fichier 3d.
SCALE 0.04	Rapport d'échelle.
BOUNDINGSPHERE 0.3	Taille de la sphère de collision.
LIFE 1	On a besoin que d'une balle pour la décrocher.
ROTSPEEDY 0.07	L'arme tourne sur elle-même, sur l'axe Y.
BONUSTYPE WEAPON	Type de bonus : une arme (WEAPON).
BONUSWEAPONPROPS Super_Laser 55 0.5 1.2 99 0 tir.wav	Définition des propriétés de l'arme : - Nom : Super_Laser (pas d'espaces !!). - 55 points de dommages à chaque balle ! - Précision de tir moyenne (0.5). - Cadence de tir très faible (1.2s entre chaque tir). - 99 munitions dans la poche. - Arme non automatique (non = 0). - Son joué à chaque tir de l'arme.
BONUSWEAPONFIREEFFECT FX.tga 0.6 0.5	Définition de l'effet provoqué par l'arme à feu : - Nom de l'image à charger. - Echelle lors de l'affichage (0.6). - Taux de transparence à sa naissance (0.5).

Une fois votre nouvelle entité créée, il vous suffit de la déclarer pour que le moteur puisse la prendre en compte lors du prochain démarrage. La déclaration des entités se trouve dans le fichier « *entitylist.txt* » du dossier « *…/data/entities/* ». Pour déclarer une nouvelle entité, il vous suffit d'ajouter le nom du fichier « *.ent* » à charger (Attention ! Ne pas écrire l'extension du fichier). L'ajout d'une déclaration peut se faire entre deux déclarations précédentes, il n'y a pas de problème. L'ordre dans le fichier « *entitylist.txt* » influence juste l'ordre d'affichage dans le menu de choix d'ajout d'entités.

Ainsi, si par exemple, le fichier « *entitylist.txt* » était déjà composé des déclarations :	*ennemi_alienvolant* *ami_aliensecours*

Si vous chercher à déclarer votre nouvelle entité définie dans le fichier « *arme_superlaser.ent* », vous pouvez écrire dans le fichier « *entitylist.txt* »:	*ennemi_alienvolant* *arme_superlaser* *ami_aliensecours*

Valeurs initiales du niveau :

Vous l'aurez sans doute remarqué, lors du démarrage du jeu, certaines valeurs par défaut sont attribuées au joueur, tels que son niveau de vie, son score, sa première arme, ou encore le temps de bouclier lui étant attribué initialement. Bien entendu, ces valeurs sont modifiables, vous permettant par exemple de faire débuter le joueur dans votre nouveau niveau avec un seul et unique misérable point de vie... ;)

Comme tous les fichiers spécifiques au niveau, le fichier d'initialisation du joueur dans votre niveau se trouve dans le dossier « *…/data/maps/nomdevotremap/* ». Le fichier s'appelle « *init.txt* ». Voici les propriétés que vous pourrez rentrer :

Nom de variable	***Description***	***Valeur attendue***
LIFE	Nombre de points de vie initial	entier
SHIELD	Temps de bouclier accordé au début	flottant
SCORE	Score initial	entier
WEAPON	Propriétés de la première arme	chaîne, entier, flottant, flottant, entier, entier, chaîne *(Nom, Dommages, Précision, Cadence, Munitions, Auto, Fichier son)*
WEAPONFIREEFFECT	Effet provoqué par l'arme (à définir après un « WEAPON »)	Chaîne, flottant, flottant *(Nom de l'image, scale, alpha)*
PARTICULEMODEL	Propriétés des particules émises par le décor	Chaîne, flottant, flottant *(Nom du modèle 3d, scale, alpha)*

Astuces :

- Comment définir une entité immobile, par exemple un bonus qui ne reste qu'à une position précise ?

Pour définir une entité immobile, il suffit que sa trajectoire soit composée d'un seul point de contrôle. Ainsi, lors de l'édition de sa trajectoire, placez un point de contrôle une seule fois à la position que vous souhaitez, et votre entité ne bougera pas au cours du temps.

- Comment faire varier la vitesse d'une entité, pour par exemple lui indiquer des pointes d'accélérations, ou des portions plus lentes ?

La vitesse d'une entité définit le temps que met l'entité pour parcourir la distance entre 2 points de contrôles consécutifs. Donc l'entité mettra autant de temps pour parcourir une distance d'un ou dix mètres si elle est comprise entre 2 points de contrôles. Ainsi, si vous voulez produire une accélération, il suffit d'écarter les points de contrôles, et au contraire, pour produire une décélération, les resserrer.

Annexe 2 : glossaire des fonctions DirectX utilisées

Il est présenté ici les fonctions DirectX utilisées pour la projection, dans leur ordre d'utilisation.

- Fonction : D3DXCreateCubeTexture :

Déclaration :

```
- HRESULT D3DXCreateCubeTexture(

   LPDIRECT3DDEVICE9 pDevice,          // le device dans lequel charger le CubeMap
   UINT Size,                          // la taille d'une face ( doit être une puissance de 2 )
   UINT MipLevels,                     // niveau de mipmap.
   DWORD Usage,                        // flag d'usage ( RENDER_TARGET pour un CubeMap de
rendu,
                                       // NULL sinon)
   D3DFORMAT Format,                   // format des données de la surface ( D3DFMT_R8G8B8
                        pour du
                                       // RGB
   D3DPOOL Pool,                       // le pool dans lequel mettre la surface
(D3DPOOL_DEFAULT pour
                                       // une surface de rendu, D3DPOOL_SCRATCH sinon)
   LPDIRECT3DCUBETEXTURE9 *ppCubeTexture // le CubeMap à initialiser.
);
```

Utilité : Elle initialise un objet COM CubeMap.

- Fonction : CreateOffscreenPlainSurface:

Déclaration :

```
- HRESULT IDirect3DDevice9 ::CreateOffscreenPlainSurface(

   UINT Width,                         // longueur de la surface en pixel
   UINT Height,                        // largeur de la surface en pixel
   D3DFORMAT Format,                   // format des données de la surface ( D3DFMT_R8G8B8
                        pour du
                                       // RGB

   DWORD Pool,                         // le pool dans lequel mettre la surface
(D3DPOOL_DEFAULT pour
                                       // une surface de rendu, D3DPOOL_SCRATCH sinon)

   IDirect3DSurface9** ppSurface,      // la surface à initialiser
   HANDLE* pHandle                     // toujours NULL .
);
```

Utilité : Elle initialise un objet COM surface.

- Fonction : GetCubeMapSurface:

Déclaration :

```
- HRESULT IDirect3DCubeTexture9::GetCubeMapSurface (

   D3DCUBEMAP_FACES FaceType,          // le numéro de la face à récupérer
   UINT Level,                         // l'indice du niveau de la face dans le cas des mipmaps
   IDirect3DSurface9 **ppCubeMapSurface // la surface dans laquelle va être recopiée la face
);
```

Utilité : Elle renvoie la face du CubeMap ayant pour numéro FaceType.

- Fonction : SetRenderTarget :

  Déclaration :

```
- HRESULT  IDirect3DDevice9::SetRenderTarget (

   DWORD RenderTargetIndex, // l'indice du niveau de mipmap
   IDirect3DSurface9 *pRenderTarget // la surface de rendu.
);
```

Utilité : Elle indique la surface en paramètre comme étant la surface de rendu courante.

- Fonction : D3DXLoadSurfaceFromSurface :

  Déclaration :

```
- HRESULT D3DXLoadSurfaceFromSurface(

   LPDIRECT3DSURFACE9 pDestSurface,     // la surface qui reçoit la copie.
   CONST PALETTEENTRY *pDestPalette,    // la palette de la surface (NULL).
   CONST RECT *pDestRect,               // rectangle qui définit la zone dans la quelle on copie (NULL
                                        // pour toute la surface)
   LPDIRECT3DSURFACE9 pSrcSurface,      // surface à copier.
   CONST PALETTEENTRY *pSrcPalette,     // palette de la surface à copier (NULL).
   CONST RECT *pSrcRect,                // rectangle qui indique la zone à copier (NULL pour toute la
                                        // surface.
   DWORD Filter,                        // filtre pour certains effets
   D3DCOLOR ColorKey                    //couleur qui remplace la couleur de transparence
);
```

Utilité : Elle recopie la surface source dans la surface destination.

- Fonction : LockRect:

  Déclaration :

```
- HRESULT  IDirect3DSurface9::LockRect (

   D3DLOCKED_RECT *pLockedRect,         // Objet qui a pour donnée le tableau de bits de la surface.
   const RECT *pRect,                   // rectangle qui définit la partie de la surface à « locker »
(NULL
                                        // pour toute la surface ).
   DWORD Flags                          // divers paramètres.
);

- HRESULT  IDirect3DSurface9::UnLockRect (     )
```

Utilité : Elles bloquent/débloquent la surface pour pouvoir accéder au tableau de bit contenu dans le D3DLOCKEDRECT. Entre le « lock » et le « unlock » toute modification du tableau de bit entraîne une modification de la surface.

- Fonction : GetBackBuffer :

  Déclaration :

```
- HRESULT IDirect3DDevice9 ::GetBackBuffer(

   UINT iSwapChain,                     // indice de changement de buffer
   UINT BackBuffer,                     // numero du backbuffer
   D3DBACKBUFFER_TYPE Type,             // type (stéréo ou mono)
   IDirect3DSurface9 **ppBackBuffer     // surface qui reçoit le backbuffer
);
```

Utilité : Elle retourne le backbuffer courant du device, comme elle retourne un pointeur sur le buffer, toute modification de celui-ci modifie l'affichage a l'écran, c'est comme cela qu'est mis a jour l'écran lors de la projection grace à la fonction suivante.

- Fonction : D3DXLoadSurfaceFromMemory :

Déclaration :

```
- HRESULT D3DXLoadSurfaceFromMemory(

   LPDIRECT3DSURFACE9 pDestSurface,     // la surface qui reçoit la copie.
   CONST PALETTEENTRY *pDestPalette,    // la palette de la surface (NULL).
   CONST RECT *pDestRect,               // rectangle qui définit la zone dans la quelle on copie (NULL
                                        // pour toute la surface)

   LPCVOID pSrcMemory,                  // tableau des données
   D3DFORMAT SrcFormat,                 // format des données ( dans l'application
D3DFMT_R8G8B8).
   UINT SrcPitch,                       // taille d'une colonne en octets.
   CONST PALETTEENTRY *pSrcPalette,     // palette de la surface à copier (NULL).
   CONST RECT *pSrcRect,                // rectangle qui indique la zone à copier (NULL pour toute la
                                        // surface.

   DWORD Filter,                        // filtre pour certains effets
   D3DCOLOR ColorKey                    //couleur qui remplace la couleur de transparence
);
```

Utilité : Elle remplit une surface avec un tableau de données.

Annexe 3 : Calcul de recombinaison avec angle

Les opérations mathématiques utilisant les angles pour faire la recombinaison sont détaillées dans le rapport de projet tuteuré, « Génération d'images panoramiques en temps réel ou pré calculées ». S'y référer pour plus de précisions. Cette annexe ne présente que le principe général de ces calculs.

L'image envoyée au projecteur est une image de type sphérique.

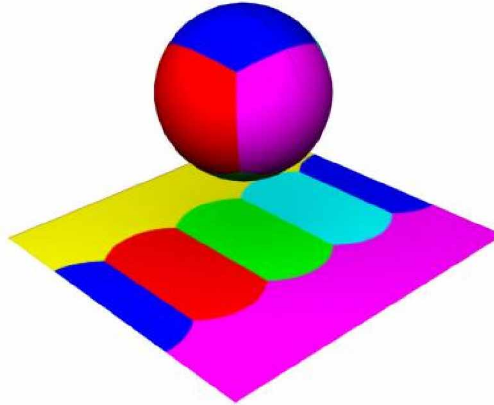


Figure 14 : Image plane envoyée au projecteur plaquée sur une sphère

Dans l'espace, une direction est caractérisée par deux angles (θ,φ). Chaque pixel de l'image plane est un point de coordonnées (θ,φ). Pour trouver la couleur de ce point il faut trouver le pixel intersection entre le cube et la droite ayant pour direction (θ,φ). La couleur recherchée est la couleur de ce point.

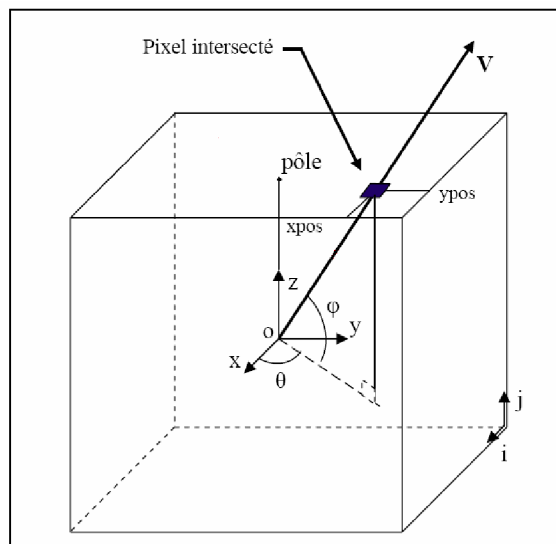


Figure 15 : Recherche d'un pixel pour un couple d'angle donné

Annexe 4 : Splines cubiques

Les splines cubiques sont issues d'une interpolation de points deux par deux, ainsi lorsqu'un point est modifié seul la partie autour de ce point change. Ce qui facilite grandement le travail de création des trajectoires. De plus par leur statut de courbe C^2 - dont la dérivée et la dérivée seconde sont définies et continues - les Splines cubiques sont des courbes qui se prêtent très bien aux trajectoires.

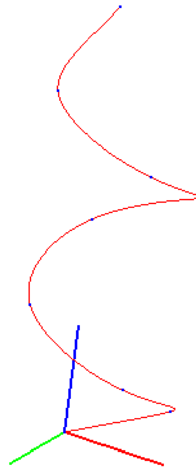


Figure 16 : exemple d'une spline 3D telle qu'elle est obtenue avec l'algorithme implémenté pour Organic

Afin d'obtenir une telle courbe la méthode utilisée a été le calcul de deux courbes splines 2D non fonctionnelles, c'est à dire dont les points ne sont pas ordonnés. Chacune de ces deux splines étant le simple projeté en z = 0 et en x = 0 obtenu à partir des points de contrôles 3D.
Pour un t donné on a ainsi deux points, un en z=0 et l'autre en x=0. Pour obtenir le point de temps t sur la spline 3d, on trace deux droites normales aux deux plans de projections, et passant chacune par le point obtenu sur la spline 2D projetée. On définit le point 3d que l'on recherche comme étant l'intersection de ces deux droites.

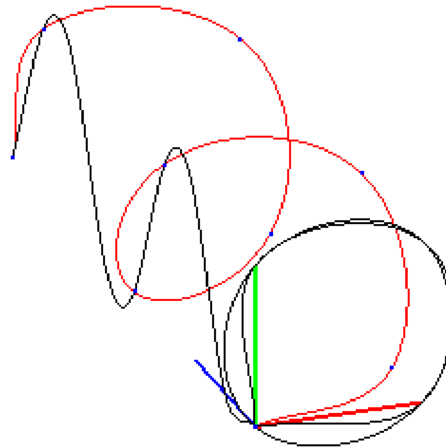
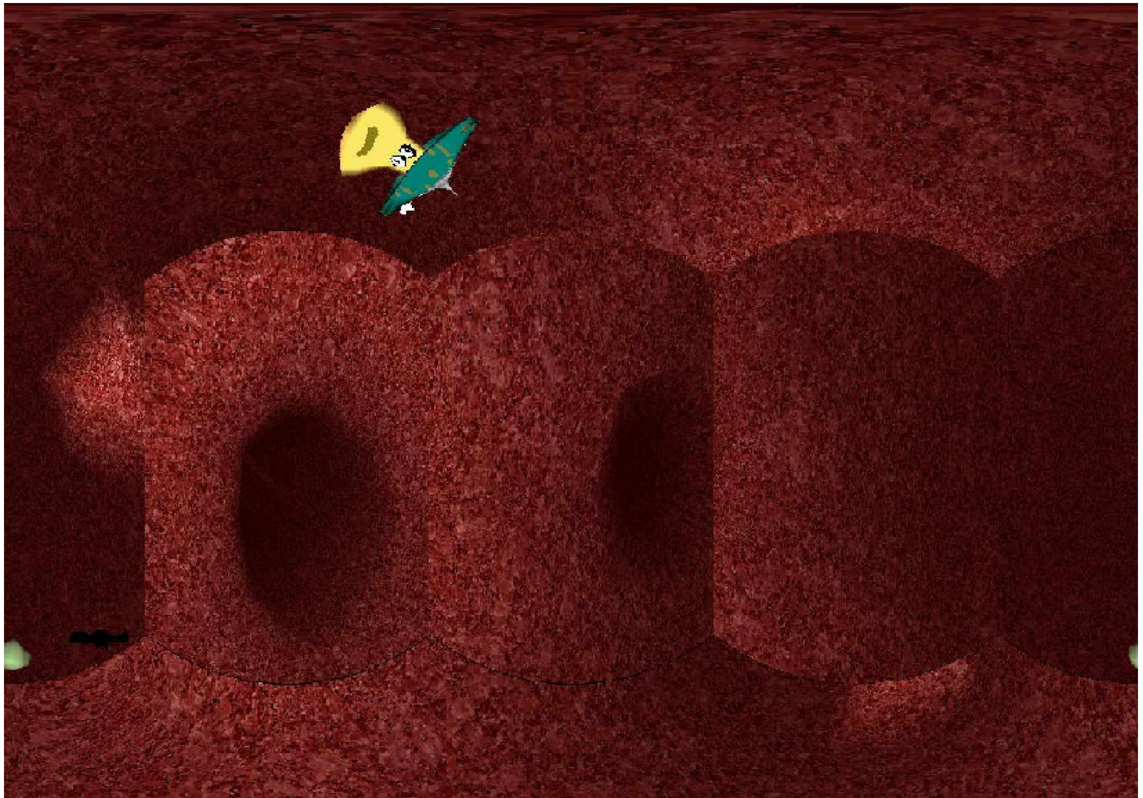


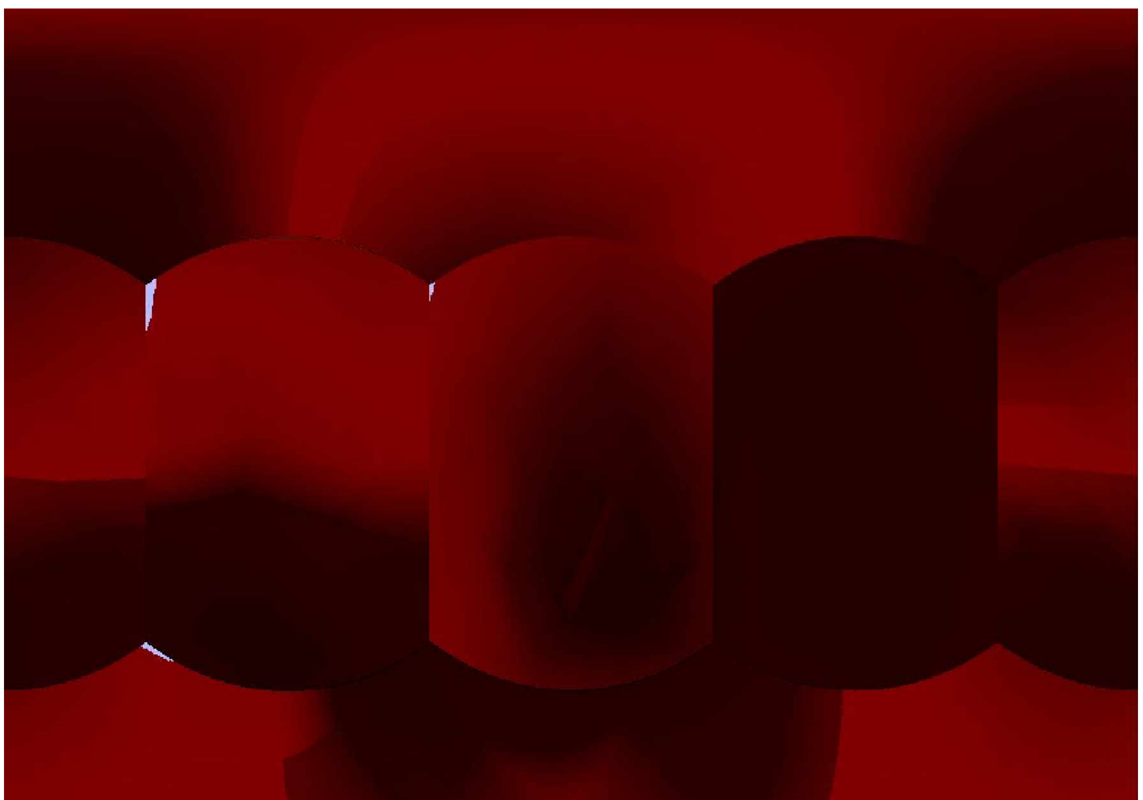
Figure 17 : En noir, les deux projetés en x=0 et z=0.
En rouge, la courbe obtenue par intersection de ses deux projetés

Cette méthode donne un résultat tout à fait convenable dans le cadre d'Organic. Par contre elle ne peut pas être réellement définie comme mathématiquement juste. En effet on utilise un même t pour calculer le point de chacune des deux courbes projetées. Or la position exacte du point défini par t dépend de la distance séparant les points de contrôles. En projetant les points de contrôles 3D on perd une information de position, ainsi rien ne nous assure que les distances séparant les points de contrôles des courbes projetées soient les mêmes. Afin de s'affranchir de cette incertitude une méthode consisterait à extruder les splines 2D le long de la normale au plan de projection. Et à effectuer une opération booléenne sur les deux objets obtenus, l'intersection de ceux-ci donne une spline 3D exacte. Cependant en plus d'être lourde à l'exécution cette méthode ne peut être utilisée pour des splines non fonctionnelles – dont les points ne sont pas triés en ordre croissant de leurs coordonnées -.

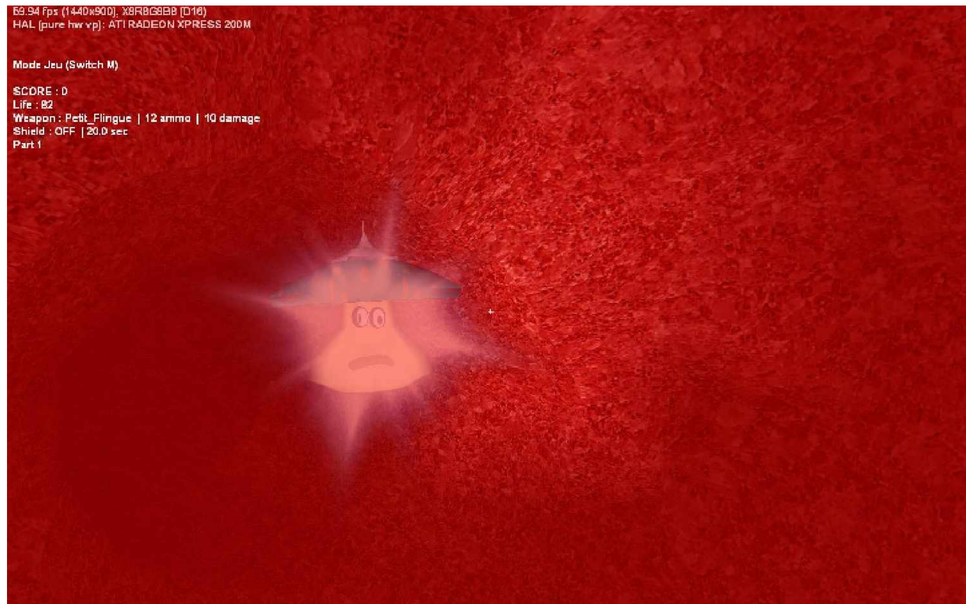
Annexe 5 : Image du jeu

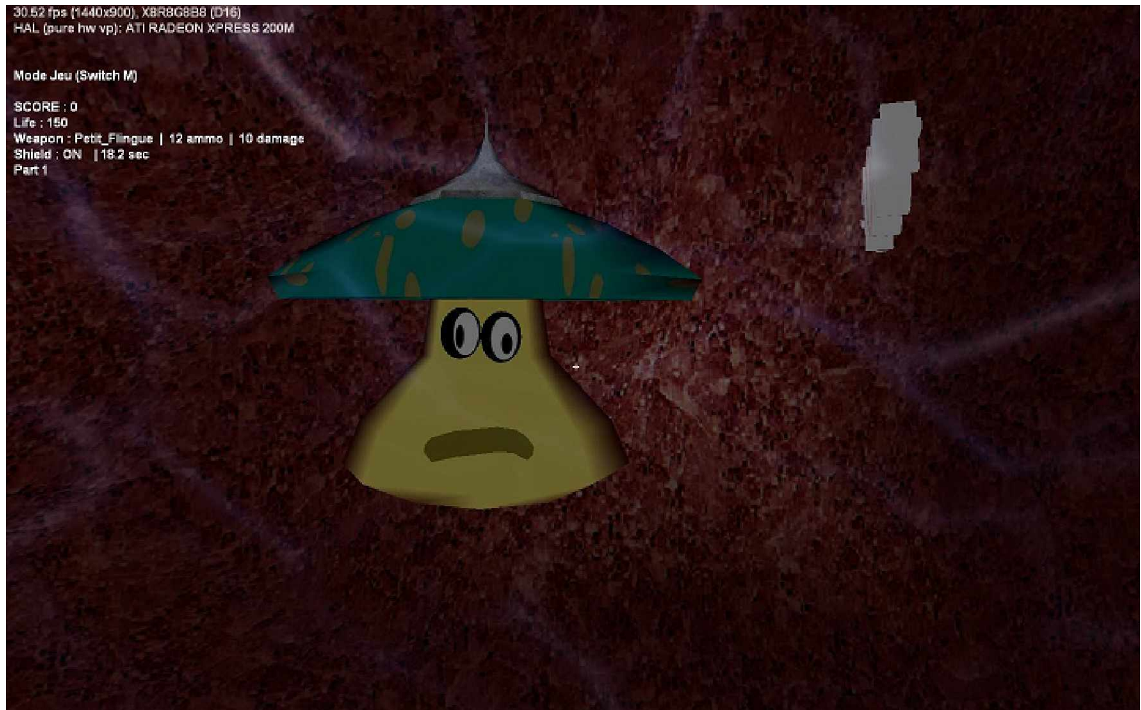
Images en mode projection :





Images en mode joueurs :





Images en mode éditeur :