Zeste de savoir

Création d'un objet de jeu vidéo

12 août 2019

Table des matières

La création d'objets 3D pour les jeux vidéo est un processus qui varie énormément d'un jeu à l'autre, et qui évolue très vite au fil de la technologie. Les joueurs passionnés entendent souvent parler de *textures*, de *shaders*, ou de termes abstraits comme *physically based* ; mais les développeurs professionnels prennent rarement le temps d'expliquer comment ils travaillent.

Cet article vise à montrer un exemple de processus créatif moderne pour réaliser des éléments de jeux vidéo. L'exemple choisi est le projet de jeu vidéo Helium Rain , un projet qui ne représente pas forcément un cas typique, mais qui reste proche des tendances de l'industrie. Il s'appuie sur deux outils gratuits - le modeleur 3D **Blender**, et le moteur de jeu **Unreal Engine 4**. Les techniques évoquées sont généralement transposables à d'autres technologies.

FIGURE 0. – La corvette Dragon, un vaisseau de Helium Rain

1. La 3D pour le jeu vidéo

Le jeu vidéo est le domaine le plus exigeant de la 3D. En presque vingt ans de jeux vidéo 3D, la course technologique n'a jamais ralenti et chaque nouveau titre tente à son tour de repousser les limites de la qualité visuelle, tout en restant accessible au plus grand nombre. Trois chiffres suffisent à visualiser à quel point la 3D temps réel est impressionnante :

— chaque image de jeu vidéo doit être rendue en quelques millièmes de seconde pour produire un résultat fluide ;
— les jeux vidéo modernes affichent parfois plus de dix millions de triangles à l'écran ;
— la PlayStation 4 est deux fois plus puissante que le meilleur super-calculateur disponible en 1997 - une machine conçue pour simuler des essais nucléaires.

Et malgré ces chiffres fous, il n'a jamais été aussi simple de créer des éléments de jeux vidéo. La raison à ce fait est très simple : pour un ingénieur qui travaille sur un jeu vidéo, il y a en moyenne dix artistes. Il est donc crucial pour les développeurs que les artistes travaillent efficacement, sans perdre de temps, sans s'adapter systématiquement aux moindres contraintes techniques. Voyons ensembles le résultat !

2. Concept et modélisation

La première étape de la production d'un élément de jeu est le *concept art*, la conception avec papier et crayon. Le concept permet d'essayer rapidement différentes atmosphères, différents styles, pour prendre des décisions avant de lancer la véritable production.

FIGURE 2. – Concept art et rendu final dans Destiny - © Bungie/Activision

Dans notre cas, l'étape du concept art est ignorée - essentiellement parce que la compétence n'existe pas dans l'équipe. C'est donc l'étape suivante qui est directement attaquée : la modélisation, et plus précisément ce qu'on appelle le **high poly**. C'est une modélisation 3D à très haut niveau de détail, sur laquelle tout est possible. Il y a typiquement plusieurs millions de triangles sur un objet de ce type, et les outils pour le produire sont de plus en plus éloignés de la technique.

On parle aujourd'hui beaucoup de *sculpture 3D* ; c'est-à-dire que l'outil de modélisation utilisé cherche à se rapprocher des méthodes de la sculpture classique. De tels outils sont devenus prépondérants pour les personnages ou les éléments vivants des jeux, par exemple. Dans notre cas, le *high poly* est réalisé intégralement dans Blender, par modélisation classique, en manipulant des faces. L'objet utilisé comme exemple dans cet article est un moteur de vaisseau.

FIGURE 2. – High poly

C'est l'étape la plus longue et la plus importante pour le résultat final, et paradoxalement le *high poly* ne sera jamais inclus dans le jeu. Il est en effet bien trop complexe, nécessite trop de ressources matérielles, et se prête mal aux étapes suivantes du rendu. Dans le jeu vidéo lui-même on utilisera donc un autre objet, qu'on appelle le *low poly* parce qu'il sera constitué de quelques milliers de triangles, et plus quelques millions. Toute la difficulté du temps réel est de

produire un rendu équivalent à ce *high poly*, mais avec un *low poly*. Nous allons voir tout de suite comment ces deux versions de l'objet cohabitent.

Pour Helium Rain, nous produisons les *low poly* en éliminant tous les détails du *high poly*, là où d'autres développeurs vont plutôt créer un nouvel objet à partir de zéro, ou encore utiliser des outils automatisés. Chacun sa méthode !

FIGURE 2. – Low poly

L'étape finale de ce processus de modélisation 3D est de réaliser ce qu'on appelle une **UV map** - l'association de chaque surface 3D à une surface 2D, pour que le jeu vidéo puisse faire correspondre des images 2D à chaque face de l'objet 3D. Vous pouvez en apercevoir un échantillon en bas de l'image précédente : chaque morceau de la surface 3D a été découpé et aplati en 2D, et l'objet garde en mémoire l'association entre 3D et 2D. C'est ainsi que au moment du rendu, pour chaque élément de surface, on peut retrouver la portion de textures concernée.

3. Le normal mapping, le détail à moindres frais

Revenons à notre *high poly*. Ses formes, ses détails sont très importants pour la qualité du rendu, mais il est impossible de les garder dans la géométrie 3D de l'objet pour des raisons de performance. Il faut donc tricher et la technique de triche fondamentale s'appelle le **normal mapping**. Le *normal mapping* consiste à encoder tous ces détails de géométrie dans une image - une **texture** ou **map** dans le jargon du rendu 3D. Chaque pixel de la texture correspond à la direction d'une face, *via* un encodage rudimentaire. L'algorithme d'éclairage du jeu vidéo utilise ces données pour modifier légèrement le comportement de la surface, afin de simuler du relief.

FIGURE 3. – High poly, normal map, et le résultat sur un simple plan - © Wikimedia

Cette texture est produite en réalisant, en quelque sorte, la *différence* entre le high et le low poly. Nous utilisons Blender pour générer cette texture, en quelques secondes de travail, à partir des deux objets. Ce processus est connu sous le nom de **baking** et c'est une étape universelle aujourd'hui pour réaliser un jeu vidéo. Passé ce stade, le *high poly* est totalement inutile, on se concentre désormais sur le *low poly* uniquement.

Le *baking* est généralement l'occasion de produire une seconde texture, l'**ambient occlusion map** ou **AO map**. Cette texture en noir et blanc permet d'assombrir les recoins des objets, ce

qui leur donne généralement un aspect plus réaliste. Elle est un peu moins connue et n'est pas aussi systématique que la première, mais nous en utilisons toujours une dans Helium Rain.

4. Shaders, matériaux et textures

Les **shaders**, c'est comme cela que l'on nomme les programmes qui réalisent le rendu des jeux vidéo. Ces programmes sont exécutés sur la carte graphique et produisent, à partir d'objets 3D, de textures et de paramètres divers, chaque pixel à l'écran. C'est un *shader* qui simule l'effet de la lumière sur un objet, évalue les reflets, les couleurs, les reliefs et produit une couleur pour chaque pixel.

Autrefois partie intégrantes des cartes graphiques, ils sont devenus programmables, spécifiques à chaque jeu vidéo, et sont aujourd'hui générés automatiquement *via* des outils graphiques. On parle alors de **material** : un *material* est une représentation abstraite utilisée pour générer des shaders, des programmes pour votre GPU. Voyons ensemble à quoi ressemble la production de *materials* dans Unreal Engine 4.

FIGURE 4. – Un exemple de material

Le graphe à droite de l'image permet de produire le rendu visualisé à gauche. Cette description graphique permet à un artiste de générer des *shaders* sans avoir de compétence particulière en programmation, et offre un contrôle incroyable sur le rendu. Le cas visualisé ici est le *material* qui contrôle le rendu de chaque objet de Helium Rain. Nous utilisons le même partout, avec des paramètres différents, et des textures différentes. C'est ici, par exemple, que la *normal map* est prise en compte.

Dans Unreal Engine 4, on procède typiquement en assemblant des *materials* génériques, que l'on réalise suivant le modèle du **Physically Based Rendering** ou **PBR**. Conçu pour respecter les lois de la physique et initialement introduit dans le monde de l'animation, ce modèle s'est immiscé en quelques années dans les principaux moteurs de jeu. Le but de cet article n'est pas d'introduire cette technologie complexe, nous dirons simplement qu'elle propose aux artistes une vision intuitive des *materials*.

FIGURE 4. – Quelques exemples en jouant avec les propriétés d'un matériau PBR

Nous disposons donc d'une banque de matériaux *PBR*, et notre *material* se borne à assembler les différents composants suivant les consignes d'un **masque**, une texture qui n'est jamais affichée dans le jeu mais contrôle le *material*. Peinture ? Métal brut ? Lumière ? Éraflure de la surface ? Tout ça est codé par une couleur dans la texture, à gauche sur l'image ci-dessous.

FIGURE 4. – Masque, normal map, AO map

Dans notre processus, par exemple, les couleurs ont l'association suivante :

— le noir représente du métal brut ;
— le bleu représente la peinture principale ;
— le rouge, le rose indiquent la peinture secondaire ;
— le vert indique du métal brillant.

Cette correspondance indirecte a un avantage supplémentaire : les couleurs elles-mêmes ne sont qu'un paramètre ! La peinture principale peut être blanche comme dans notre objet d'exemple, mais aussi bleue, noire, rouge, elle peut avoir un motif ou des détails supplémentaires... Cette approche est donc extrêmement flexible.

Il ne reste plus qu'à importer le *low poly* dans le jeu, lui assigner le *material* configuré et effectuer quelques menu réglages. On réalise aussi, à ce moment-là, une deuxième version du low poly, qui est *encore moins détaillée* et qui sera utilisée à grande distance pour alléger la consommation en ressources.

Voilà à quoi ressemble le résultat final dans l'éditeur d'Unreal Engine, avec les collisions en violet :

FIGURE 4. – Objet final

C'est fini pour la partie 3D, mais il reste encore du travail !

5. Intégration

Maintenant que l'on dispose d'un objet 3D complet, il faut que le jeu lui-même puisse s'en servir. Pour ce faire, nous avons développé un mécanisme de **catalogue**, une énorme liste de tous les composants de vaisseau disponibles.

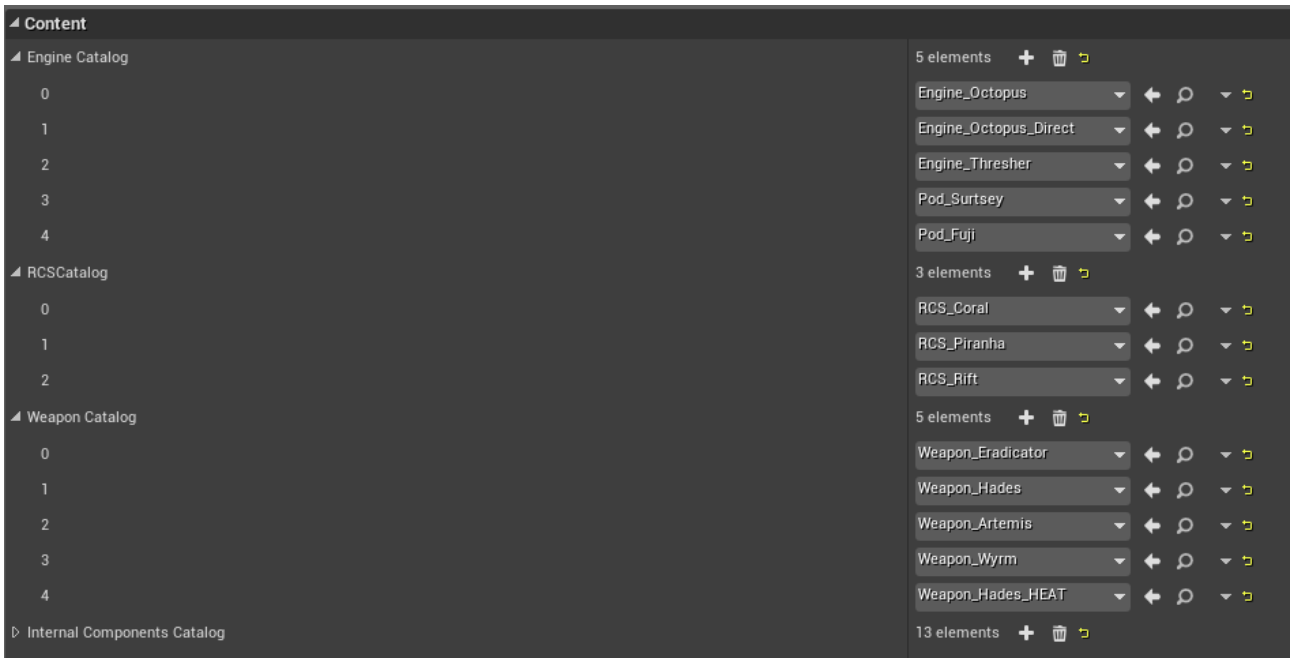

FIGURE 5. – Le catalogue des pièces

Rien de tout cela n'est stocké dans le code du jeu - ajouter ce moteur, le rendre visible dans les menus, utilisable par un vaisseau, *tout ça est fait sans jamais changer une seule ligne de code* puisque nous utilisons le mécanisme de gestion du contenu du moteur. Le catalogue est ici un *asset* comme le sont des textures ou des objets 3D, ce qui permet d'ajouter du contenu sans intervention en C++.

Le code associé au catalogue est relativement simple et lisible, même pour quelqu'un qui n'a pas l'habitude de l'API associée au moteur. On génère un formulaire graphique de ce type en quelques lignes de C++ avec Unreal Engine 4.

```
1  UCLASS()
2  class UFlareSpacecraftComponentsCatalog : public UDataAsset
3  {
4          GENERATED_UCLASS_BODY()
5
6  public:
7
8          /** Orbital engines */
9          UPROPERTY(EditAnywhere, Category = Content)
10         TArray<UFlareSpacecraftComponentsCatalogEntry*>
               EngineCatalog;
11
12         /** RCS engines */
13         UPROPERTY(EditAnywhere, Category = Content)
14         TArray<UFlareSpacecraftComponentsCatalogEntry*> RCSCatalog;
15
16         /** Weapons */
17         UPROPERTY(EditAnywhere, Category = Content)
```

```
18          TArray<UFlareSpacecraftComponentsCatalogEntry*>
                WeaponCatalog;
19
20          /** Internal modules */
21          UPROPERTY(EditAnywhere, Category = Content)
22          TArray<UFlareSpacecraftComponentsCatalogEntry*>
                InternalComponentsCatalog;
23
24  };
```

Chacune des entrées que vous avez pu voir dans le catalogue correspond à un formulaire graphique dans lequel nous entrons des données et du contenu. L'objet 3D lui-même y est lié, ainsi que le prix de la pièce, l'icône ou l'accélération du moteur : c'est un bric à brac de toutes les informations utiles ! Voici l'entrée de notre objet :

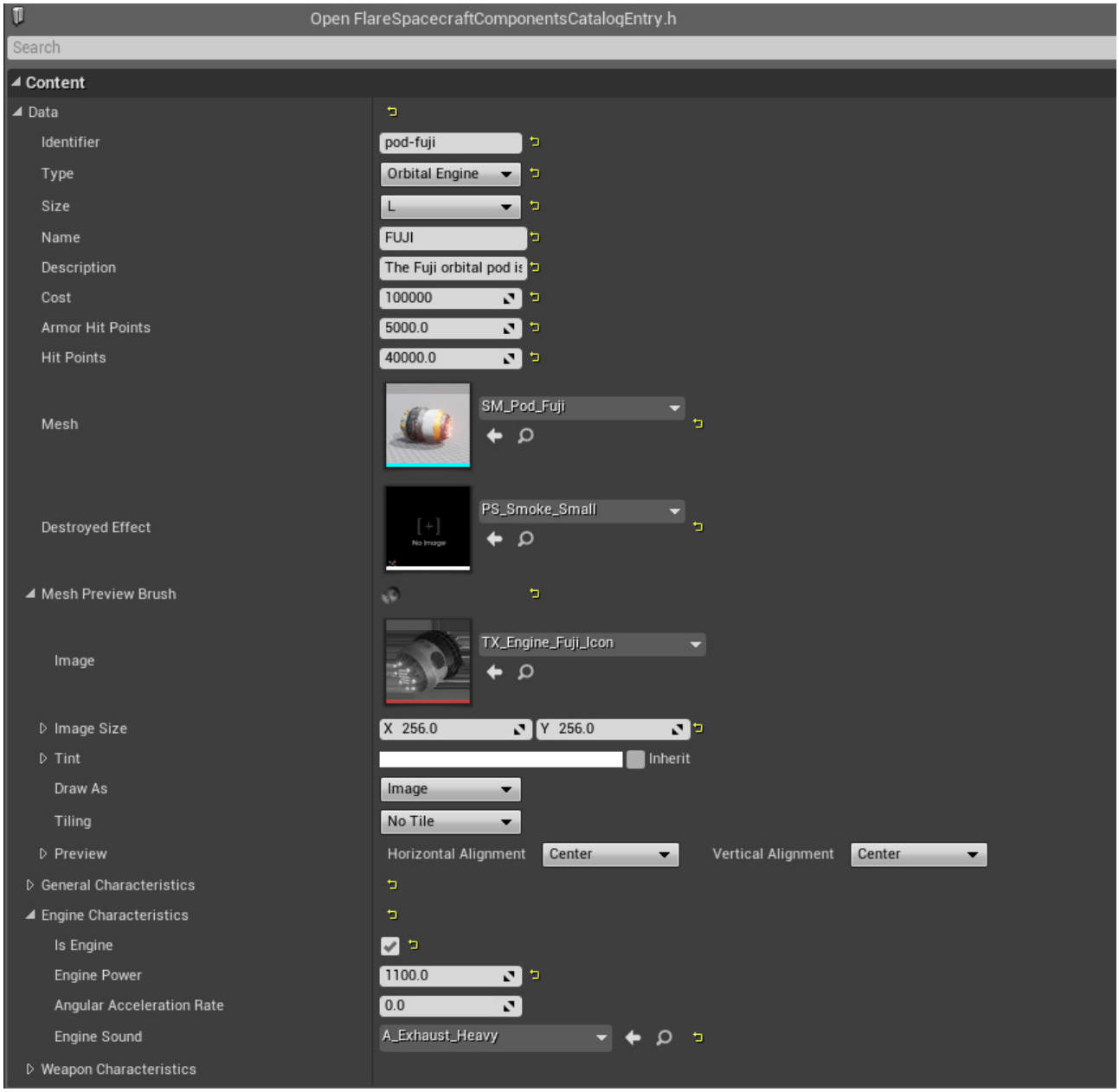

FIGURE 5. – Une entrée de catalogue

On reprend ici le même procédé, mais avec un code C++ un peu plus massif . Le principe reste le même : associer, par introspection, une structure C++ à un formulaire graphique. Disposer d'un tel outil dans le moteur de jeu est extrêmement pratique.

6. Pour en savoir plus

J'espère que ce rapide exposé vous a appris des choses. Gardez en tête qu'il s'agit d'un exemple issu d'un jeu particulier et que chaque équipe a ses propres méthodes. Si vous voulez approfondir cet article, voici quelques ressources utiles :

— le guide création de jeux vidéo par **Linko**, qui approfondit les notions de 3D évoquées ici ;
— pour ceux qui font déjà de la 3D, un exemple d'apprentissage du high poly avec **Elvian** ;
— un guide de la technologie PBR ;
— la présentation du projet Helium Rain sur Zeste de Savoir et le site officiel .

Le forum **Multimédia & Jeux Vidéo** de Zeste de Savoir est toujours prêt à guider de nouveaux débutants, alors n'hésitez pas à vous lancer si la 3D vous intéresse. De même, n'hésitez pas à poser ici toutes les questions qui vous passent par la tête !

Merci de votre lecture.