

BORDERLANDS

HMIN201 - Travaux d'études et de Recherches

Florian Lacombe, Axel Veber, Yoann Ongaro

Equipe STIMPAC



Contents

1	Introduction	4
1.1	Ligne de produits logicielle	4
1.2	Présentation de la licence Borderlands	6
2	Cahier des charges	7
2.1	Personnages	7
2.1.1	Base	7
2.1.2	Objets	7
2.2	Arsenal	7
2.2.1	Armes	7
2.2.2	Types de dégâts	8
2.3	Monstres	8
3	Gestion du projet	9
3.1	Ingénierie logicielle et Jeux vidéo	9
3.2	Gestion du code et des ressources	10
3.3	Estimation du temps de travail	11
4	Conception de la ligne de produit	12
4.1	Analyse du domaine	12
4.2	Modélisation	13
4.3	Le configurateur	14
4.4	Les composants	15
4.5	Application sur le projet	15
4.6	La configuration des composants	16
5	Génération des armes et personnages	19
5.1	Bases de la programmation 3D	19
5.1.1	Mesh	19
5.1.2	Matériaux	19
5.1.3	Animations	20
5.2	Gestion procédurale	21
5.3	Extraction de ressources	21
5.4	Traitements	22
5.4.1	Conversion	22
5.4.2	Coloration	23
5.4.3	Découpage	25
5.4.4	Assemblage	26
5.5	Animation	30
5.6	Comparaison de solutions	31
5.6.1	3ds Max	31
5.6.2	Blender	32
5.6.3	MilkShape	32
6	Perspectives	33
6.1	Apprentissage	33
6.2	Portage sur système Android	33
6.3	Mise en place d'un système multijoueurs en ligne	33
6.4	Autres	33
7	Remerciements	34

8	Annexe	35
References		36

1. Introduction

1.1. Ligne de produits logicielle

Les lignes de produit existent dans de nombreux domaines et principalement dans le domaine de l'automobile. Le constructeur automobile Dacia® utilise le concept de ligne de produit afin de réduire ses coûts et réduire ses marges.

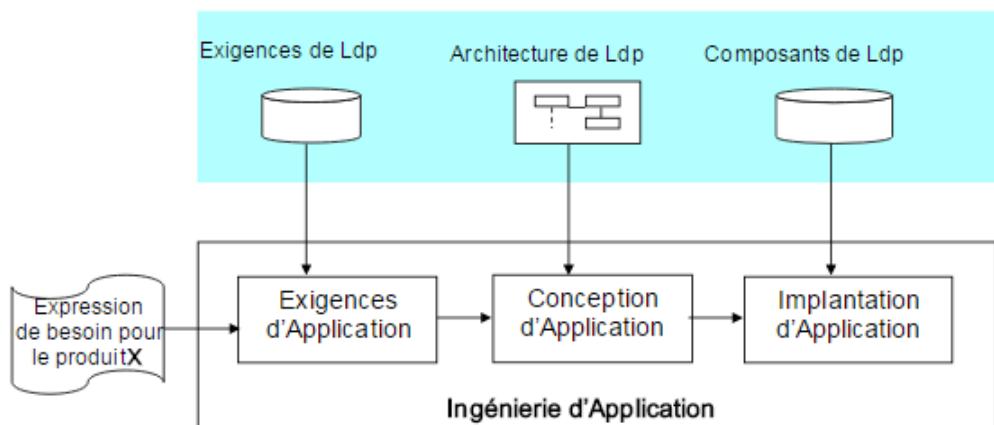
Ses véhicules sont construits à partir d'une banque de pièces déjà existantes et ayant fait preuve de leur fiabilité.

Les pièces qu'ils ont eux même développé (principalement, esthétiques) sont réutilisées dans une grande partie de leur gamme afin d'amortir les coûts de conception (phares, tableau de bord). Environ la moitié des pièces du Dacia Duster® proviennent de la Dacia Sandero®.

Les ingénieurs logiciels ont tenté d'adapter ce système au développement logiciel. Une ligne de produits logiciels est un ensemble de logiciels qui partagent des éléments communs et des variabilités afin de satisfaire un besoin précis.

Le but principal d'une ligne de produits logiciels est de réduire le plus possible les coûts et les délais de conception d'un logiciel.

Tous les logiciels (produits) issus d'une ligne de produit sont développés à partir d'une même architecture, dite de référence, à laquelle sont assemblés divers composants déjà existants ou non via des points de variabilité.



De cette manière la création d'un nouveau produit ne nécessite pas de recommencer la conception “from scratch”, mais seulement d'assembler des fonctionnalités préexistantes.

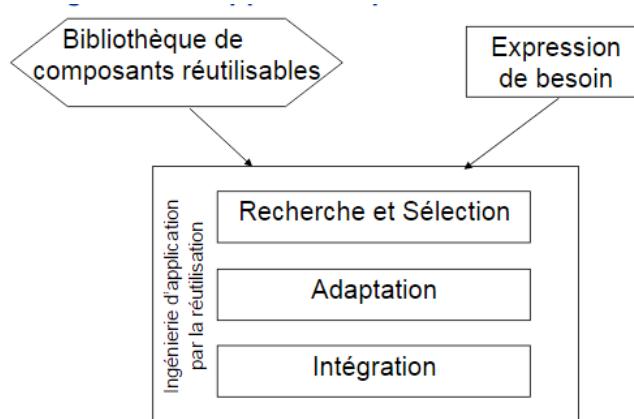


Fig. 1. Réutilisation des composants

Dans le cas où aucune fonctionnalité ne correspond à nos besoins, il est tout de même possible de modifier les composants déjà existants (via paramétrage, héritage) ou d'en concevoir de nouveaux. Ces nouveaux composants viendront alors enrichir notre banque, et pourront être réutilisés dans d'autres produits.

Les lignes de produits disposent également d'autres avantages :

- Gain de temps lors des tests : les composants que l'on réutilise ont déjà été testés séparément et les procédures de test sont déjà en place sur la ligne de produit.
- L'estimation des coûts et des temps de développement est simplifiée et améliorée : on dispose d'un retour sur expérience grâce aux produits précédemment développés, les processus de développement sont fiables et déjà mis en place.
- Amélioration de la qualité à chaque nouveau produit, les éventuels défauts des produits précédents ne seront plus reproduits.
- Via les anciens produits, il est plus simple d'un point de vue marketing, de promouvoir un nouveau produit ou de donner au client un aperçu de ce à quoi il peut s'attendre.

L'efficacité d'une ligne de produit est étroitement liée à la finesse de conception de son architecture de référence, ainsi qu'à la pertinence de ses points de variabilité.

La conception d'une ligne de produit est donc un exercice complexe qui nécessite de penser au-delà d'un unique produit.

1.2. Présentation de la licence *Borderlands*

Borderlands est un mélange de jeu de tir à la première personne et de jeu de rôle multi-joueurs. Le jeu permet d'incarner un chasseur de trésors confronté à la nature hostile du monde fictif dans lequel il se trouve.



Chaque chasseur définit une classe qui dispose d'une capacité spéciale et de compétences qui lui sont propres. Un joueur peut également coopérer avec plusieurs amis qui amèneront leur personnage dans la partie du joueur hôte.

Cette licence tient son succès de son énorme part d'aléatoire.

Lors de son apparition chaque objet utilisable par le joueur est généré à partir d'une combinaison de plusieurs sous ensembles d'objets, ce qui permet au jeu de se renouveler constamment et de rendre chaque partie unique.

2. Cahier des charges

2.1. Personnages

2.1.1. Base

Un personnage dispose de trois valeurs lui permettant d'encaisser les dégâts causés par ses adversaires:

- **Santé**

Quand cette valeur atteint 0 le personnage meurt.

- **Armure**

Cette valeur absorbe complètement les dégâts en priorité, lorsque l'armure atteint 0, la santé subit les dégâts.

- **Bouclier**

Cette valeur absorbe les dégâts avant l'armure ou la santé. Elle se remplit automatiquement lorsqu'aucun dégât n'a été subit pendant un certain temps.

2.1.2. Objets

Cette partie concerne la gestion des objets équipements du personnage:

- **Inventaire**

L'inventaire est une collection d'objets (armes, boucliers,...) de taille fixe.

- **Equipements**

Un personnage dispose de 5 emplacements dans lesquels il peut placer des objets.

- 4 emplacements d'armes qui correspondent à des raccourcis clavier.
- 1 emplacement pour un bouclier

- **Munitions**

les munitions sont représentées par une collection contenant pour chaque type d'arme (pistolet,fusil d'assaut, sniper, fusil à pompe,lance-roquettes) le nombre de munitions transportées par le personnage ainsi que le nombre maximum de munitions de ce type qu'il peut avoir sur lui.

2.2. Arsenal

2.2.1. Armes

Les armes seront composées de plusieurs parties qui pourront être assemblées de différentes manières, par le joueur ou par le jeu. Les objets qui apparaîtront au cours du jeu seront une combinaison aléatoire de ces parties.

- Une partie principale déterminera le type de l'arme (fusil de précision, fusil à pompe....) , le type de dommage qu'elle infligera ainsi que ses statistiques telles que par exemple la vitesse de chargement, la dispersion des projectiles etc.
- D'autres parties viendront s'y greffer (crosse,canon,viseur) modifiant ainsi les statistiques de l'arme.

2.2.2. Types de dégâts

Les projectiles émis par les armes infligeront aux personnages des dégâts normaux ou de différents types, ainsi que des statuts. On distinguerá cinq types de dommages:

- **Feu**
Bonus de dégâts sur la santé.
- **Corrosif**
Bonus de dégâts contre l'armure.
- **Électrique**
Bonus de dégâts contre les boucliers.
- **Slag**
Infligera l'état "slag" à l'ennemi. Un ennemi sous cet état verra ses dégâts reçus (de tous types hormis slag) doublés.
- **Explosif**
Inflige des dégâts dans une zone.

Un des challenges du jeu consistera donc à combiner habilement les différents types afin de maximiser les dégâts infligés.

2.3. Monstres

Les joueurs seront emmenés à rencontrer plusieurs types d'ennemis aux stratégies différentes. Nous nous concentrerons lors de ce TER sur 2 comportements différents:

- **Dégâts à distance**
Un ennemi mobile qui tire à vue sur les personnages et tente d'éviter les tirs en se mettant à couvert.
- **Dégâts au corps à corps**
Un adversaire qui se rapprochera rapidement du joueur tout en évitant les tirs par des déplacements agiles et infliger d'importants dégâts une fois à portée.

3. Gestion du projet

3.1. Ingénierie logicielle et Jeux vidéo

Créer une ligne de produit impose une toute nouvelle manière de concevoir que nous n'avions pas abordé jusqu'à présent.

En effet jusqu'à maintenant nous nous concentrions sur la conception et le développement de logiciels uniques n'ayant aucun lien.

Mais les lignes de produits demandent une architecture d'entités constituées de composants qui définissent la réponse de l'entité lors de certaines demandes ce qui constitue un premier verrou pour l'avancement du projet.

La conception de jeux vidéos comporte également son lot de difficultés, en particulier un jeu Borderlands.

La part d'aléatoire de la série impose une modularité dans des domaines que nous n'avons pas eu l'occasion d'étudier avant ce projet.

Par exemple créer le modèle 3D d'une arme aléatoirement à partir d'autres modèles 3D, non conçus pour cet usage, constitue plusieurs verrous techniques importants :

- Comment créer un ensemble de pièces à partir d'une arme créée en un seul morceau ?
- Comment attacher dynamiquement les pièces obtenues lors de l'exécution du jeu ?
- Comment animer l'ensemble pour que l'arme soit convaincante et non inerte ?

Une des difficultés de ce projet fut donc d'adopter cette nouvelle méthode de conception tout l'appliquant au développement de jeux vidéos.

Afin de développer notre jeu nous nous sommes appuyés sur le moteur de jeu Unreal Engine que nous n'avions jamais utilisés au-paravant. Ce moteur de jeu fut notre plus grand frein au cours de ce TER. Le manque de documentation sur l'utilisation de ce moteur couplé au C++ et ses spécificités techniques (notamment sa politique d'allocation mémoire) a énormément compliqué notre phase de développement.

3.2. Gestion du code et des ressources

Nous avons utilisé GIT afin partager le code et gérer les versions.

Cependant au vu de la lourdeur des ressources 3D (environ 1.7Go), nous les avons séparé du code afin d'atténuer les conséquences d'un problème sur le dépôt.

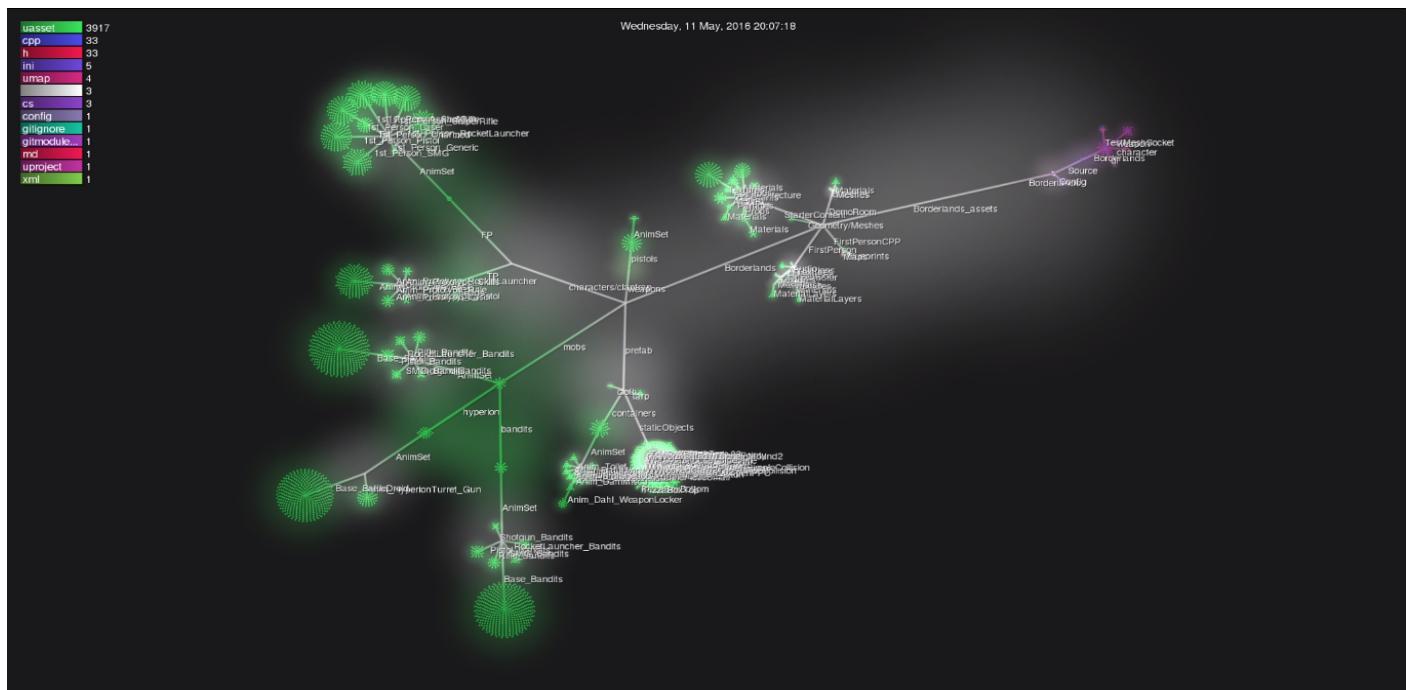


Fig. 2. Visualisation source des dépôts GIT (ressources 3D en vert)

En effet dans le cas d'un dépôt unique, une erreur de manipulation pourrait obliger à remettre en ligne des ressources 3D lourdes alors qu'elles ne sont que très rarement modifiées.

Un problème de gestion du code d'une copie locale du dépôt obligerait également à télécharger à nouveau les assets graphiques inutilement alors que le contenu demandé ne pèse que quelques Mo.

Les ressources 3D sont donc gérées par un sous-module, facilement récupérable et modifiable et n'alourdissent pas le dépôt principal.

3.3. Estimation du temps de travail

Nous nous sommes basé sur le nombre de crédits ECTS attribués à l'UE afin d'estimer le temps de travail que chaque personne allait consacrer au projet chaque semaine.

On sait que 5 ECTS ont été attribué au TER ce qui dans UE équivalente vaut 5h de travail par semaine et environ $\frac{1}{3}$ de travail personnel supplémentaire, soit 2h par semaine.

En plus de ce temps alloué au projet, nous avons organisé des réunions chaque semaine avec notre encadrant M. Seriai.

Chaque réunion durait 1h et nous passions 1h supplémentaire à préparer ces réunions (démonstrations, constructions de schémas et diagrammes, ...).

Nous estimons donc notre charge de travail à 9h par semaine par personne.

Comme chacun de nous trois a travaillé sans absence sur les 16 semaines du projet, on peut estimer le temps passé sur le projet à $16 * 3 * 9 = 432$ heures.

4. Conception de la ligne de produit

L'ingénierie du domaine est la première grande phase de l'élaboration d'une ligne de produit. Elle a pour but de définir l'architecture de référence d'une ligne de produit et de concevoir l'ensemble des artefacts, répondants aux problématiques du domaine. Ces artefacts doivent être pensés de façon à pouvoir être utilisés dans l'ensemble des produits issus de la ligne de produit.

4.1. Analyse du domaine

Il faut pour cela procéder à une analyse du domaine, dans notre cas le jeu vidéo. Cette étape consiste à collecter et d'organiser l'ensemble des informations que l'on peut recueillir sur le domaine. Ces informations peuvent provenir de sources différentes : littérature, analyse d'implémentations existantes, expérience de développement....

Nous avons donc réalisé une étude de cas du jeu Borderlands mais également d'autres jeux de la même catégorie. L'analyse du domaine est donc une étape cruciale permettant de définir ce qui est commun et ce qui diffère dans les différents produits appartenant au domaine.

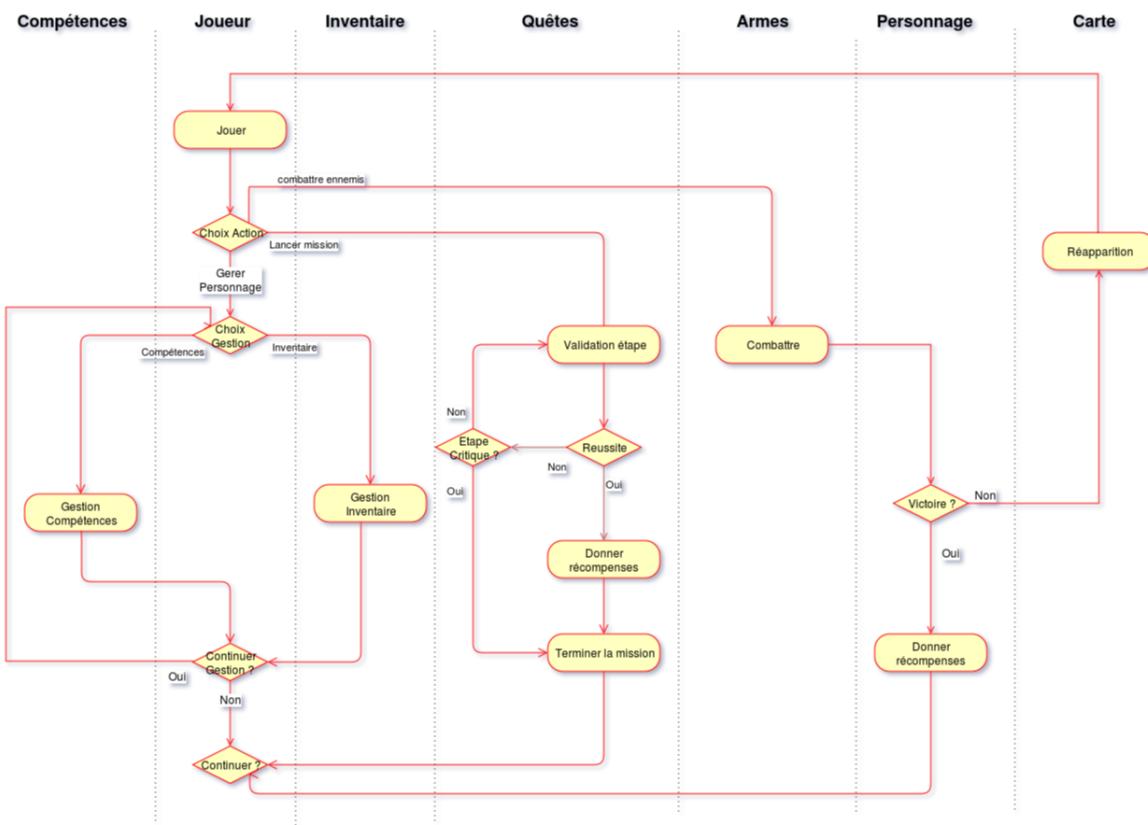


Fig. 3. Diagramme d'activité d'un produit

4.2. Modélisation

Les données recueillies lors de l'étape précédentes doivent ensuite être représentées, formalisées afin d'avoir une vision claire. On réalise pour cela un "feature model". Le feature model permet de représenter toutes les fonctionnalités/features de la ligne de produit et de les hiérarchiser sous forme d'arborescence.

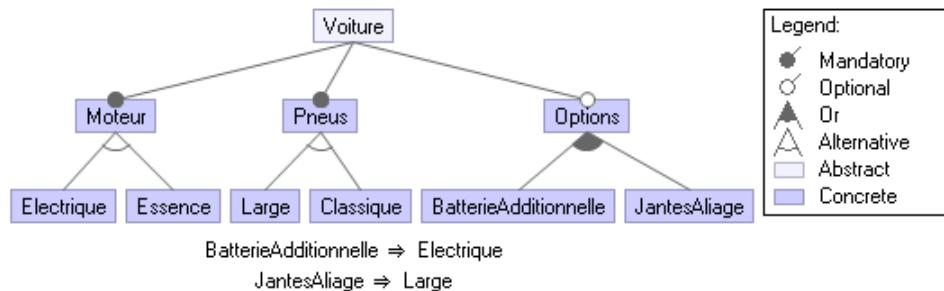


Fig. 4. Un exemple de feature model

Chaque noeud représente une feature et chaque arc ou groupe d'arcs représente la variabilité de la feature qui peut prendre 4 formes :

- **mandatory** : si le parent existe alors les enfants "mandatory" doivent également être présents. Dans l'exemple une voiture a forcément un moteur et des pneus.
- **optional** : si le noeud parent est présent, le noeud enfant peut ou non être présent. Une voiture peut ne pas avoir d'option.
- **alternative** : quand le parent est présent un seul de ses fils peut être sélectionné. Une moteur est soit électrique soit essence, il ne peut pas être les deux à la fois.
- **or** : si le parent est sélectionné un ou plusieurs de ses fils doit l'être également. On peut donc dans l'exemple cumuler les options BatterieAdditionnelle et JantesAliages.

On peut ainsi pour chaque feature modéliser l'ensemble de ses variantes et établir un ensemble d'interdépendances pour empêcher la création d'un produit non conforme.

Dans l'exemple deux contraintes de type implication sont présentes : la sélection de l'option BatterieAdditionnelle implique d'avoir un Moteur de type Electrique. On parle dans ce cas de contrainte forte (il existe également un conflit A si A est sélectionné alors B est désactivé).

```

<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<featureModel chosenLayoutAlgorithm="1">
  <struct>
    <and abstract="true" mandatory="true" name="Voiture">
      <alt mandatory="true" name="Moteur">
        <feature mandatory="true" name="Electrique"/>
        <feature mandatory="true" name="Essence"/>
      </alt>
      <alt mandatory="true" name="Pneus">
        <feature mandatory="true" name="Large"/>
        <feature mandatory="true" name="Classique"/>
      </alt>
      <or name="Options">
        <feature mandatory="true" name="BatterieAdditionnelle"/>
        <feature mandatory="true" name="JantesAliage"/>
      </or>
    </and>
  </struct>
  <constraints>
    <rule>
      <imp>
        <var>BatterieAdditionnelle</var>
        <var>Electrique</var>
      </imp>
    </rule>
    <rule>
      <imp>
        <var>JantesAliage</var>
        <var>Large</var>
      </imp>
    </rule>
  </constraints>
  <calculations Auto="true" Constraints="true" Features="true" Redundant="true" Tautology="true"/>
  <comments/>
  <featureOrder userDefined="false"/>
</featureModel>

```

Fig. 5. XML décrivant le feature model et les contraintes associées

Depuis leur apparition en 1990, les features models, de plus en plus utilisés dans le domaine des lignes de produit, ont été améliorés et des versions dites étendues ont fait leur apparition.

4.3. Le configurateur

Par explosion combinatoire le nombre de produits réalisables par une ligne de produit est généralement élevé. Gérer cette variabilité au niveau de chaque artefact lors de l'exécution n'est pas envisageable. La majorité de ses variabilités doivent être résolues avant l'exécution du programme. On utilise pour cela un fichier de configuration décrivant l'ensemble des features qui seront utilisées. Ce fichier est alors lu à chaque exécution et les composants instanciés en fonction.

```

if (HasFeature("Electrique"))
{
    /*Initialisation du composant*/
}

```

Le plugin Eclipse FeatureIDE que nous avons utilisé pour créer notre feature model génère un formulaire aussi appelé configurateur. C'est ce même configurateur qui est présenté aux clients par les product managers afin de les aider à paramétriser leur produit en fonction de leurs exigences.

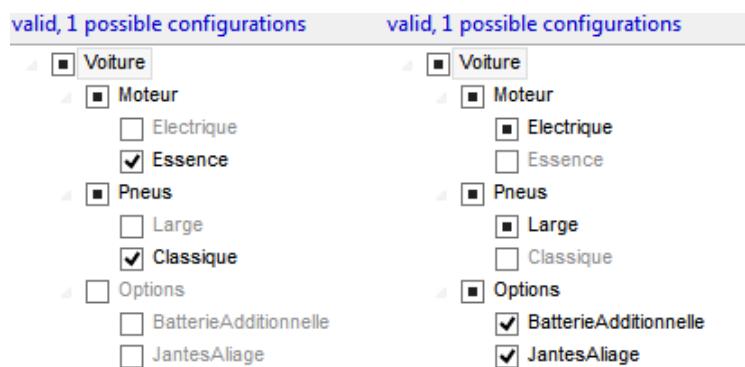


Fig. 6. Un exemple de formulaire de configuration

4.4. Les composants

Les artefacts de notre ligne de produits sont implémentés sous forme de composant logiciels. Les composants logiciels comme leur nom l'indique ont été créés dans le but d'être composés, c'est à dire assemblés entre eux sans que leur fonctionnement interne soit connu. Par conséquence les composants ont une reutilisabilité accrue et peuvent même être développés indépendamment d'une application.

Chaque composant fournit un service et communique avec d'autres composants par des points d'accès appelés interfaces. Certaines interfaces sont requises (le composant en a besoin pour fonctionner) d'autres sont offertes (le service rendu par le composant). Ces interfaces sont la seule contrainte pour remplacer un composant par un autre.

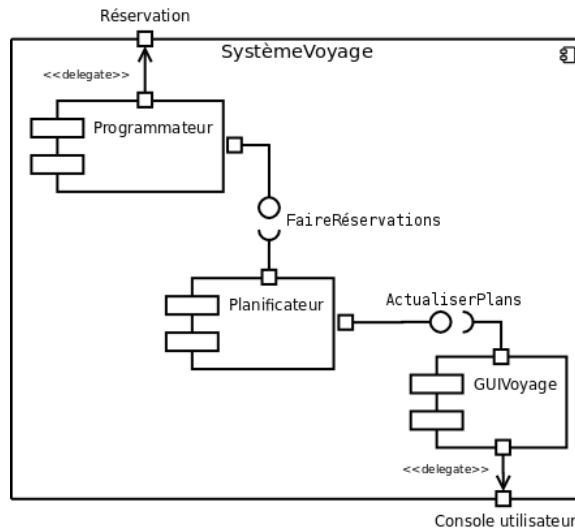


Fig. 7. Modélisation d'un système de composants

Les interfaces requises sont modélisées sous forme de demi-cercle, les interfaces fournies sous forme d'un cercle complet. Le planificateur fournit l'interface *ActualiserPlan* et requiert l'interface *FaireReservation*.

4.5. Application sur le projet

Afin d'incorporer le principe de ligne de produit dans le projet, plusieurs documents ont été créés.

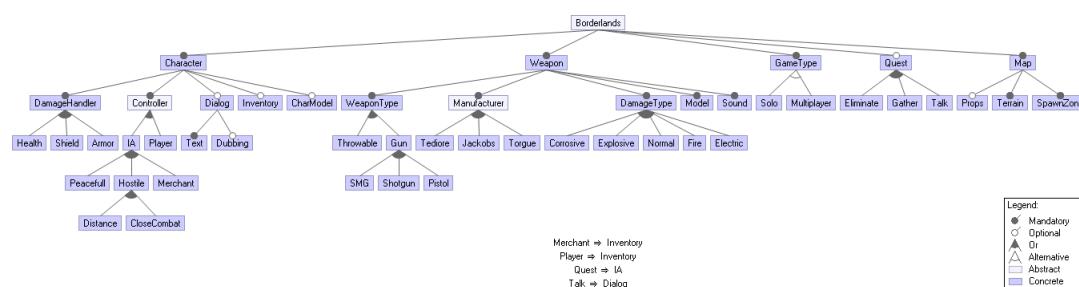


Fig. 8. Le feature model initial

La figure 8 est le feature model initialement créé. Il a été conçu volontairement large afin de considérer le plus de points de variabilité possible. On peut retenir néanmoins qu'un produit reposera sur trois grandes entités, les features *Character*, *Weapon* et *GameType*.

A l'origine du projet, lorsque les architectures à base de composants ont été évoquées, nous avons envisagé d'implémenter une architecture ECS (pour Entity-Component-System), architecture à base de composants qui a récemment trouvé sa popularité dans le développement de jeux vidéos. Le principe est de créer des entités, qui vont peupler le jeu, et leur donner des fonctionnalités à travers les composants. Unreal Engine, le moteur utilisé pour le projet, ne supporte pas entièrement cette architecture, mais nous nous en sommes néanmoins inspiré pour créer notre architecture.

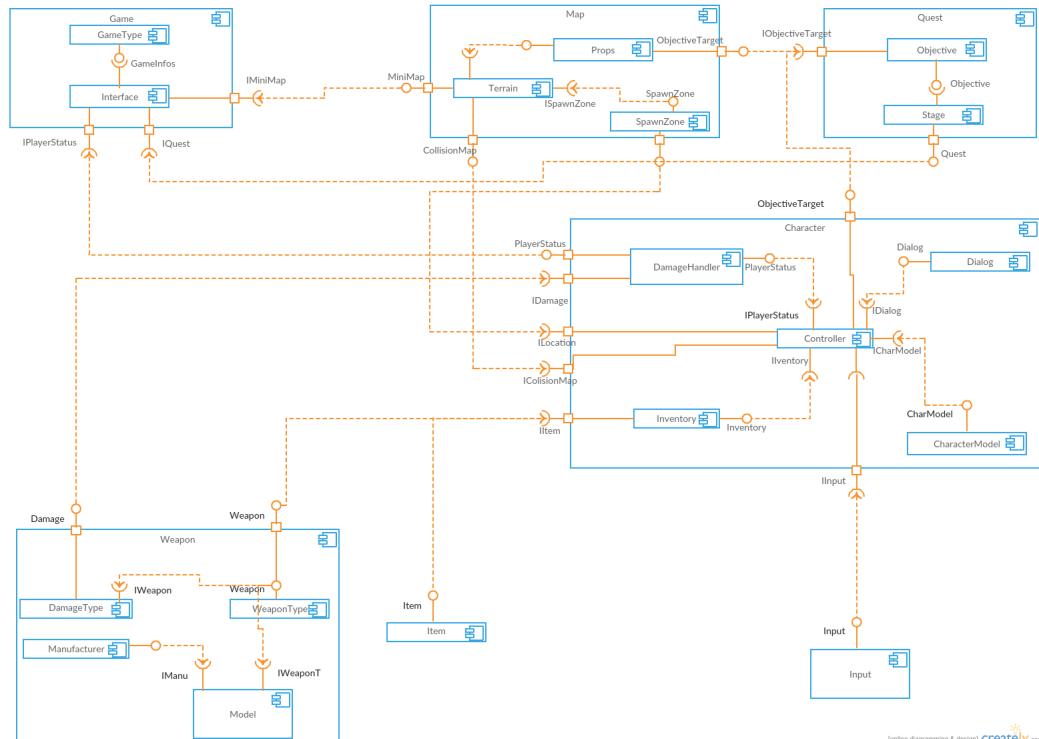


Fig. 9. Le diagramme de composant initial

Le feature model nous a donc conduit à créer le diagramme de composant de la figure 9. On y retrouve les grandes entités précédemment citées, ainsi que leurs composants. Pour ce projet, nous avons supposé qu'une fonctionnalité correspond à un composant.

Certains composants, comme les composants visuels par exemple, sont déjà implémentés dans Unreal Engine, nous avons donc implémentés les composants métiers. Sur ce diagramme, nous pouvons voir que les composants peuvent contenir d'autres composants, ce qui nous a dirigé vers un modèle hiérarchique. Ainsi, un composant possède des références vers ses sous-composants, et chaque sous-composant a une référence vers ses "propriétaires". Il est à noter que même Unreal Engine organise ses composants sous la forme d'un arbre.

Tous les composants ont donc été implémentés avec des traitements qui leurs sont propres et des interactions avec les composants père et fils, en fonction de leur position dans l'arborescence.

4.6. La configuration des composants

Une fois les composants implémentés, il faut donc les assembler entre eux. Nous nous servons donc du plugin Eclipse FeatureIDE, mentionné ci-dessus, afin d'en extraire un fichier de configuration, qui va contenir les features voulues pour créer un produit.

```
Character
DamageHandler
Health
Shield
Armor
...
```

Le fichier de configuration n'a pas de format spécial, il est sous la forme d'un fichier texte avec une fonctionnalité par ligne.

```
bool UBorderlandsGameConfigurator :: LoadFeaturesFromFile()
{
    FString GameDir = FPaths::GameDir();
    FString FileData = "";
    FString CompleteFilePath = GameDir + "default.config";

    FFileHelper :: LoadFileToString(FileData, *CompleteFilePath);
    if (FileData == "")
    {
        return false;
    }

    FileData.ParseIntoArrayLines(features, true);

    return true;
}
```

Le listing ci-dessus présente la manière dont le jeu récupère les fonctionnalités envoyées par le configurateur.

L'assemblage des composants se fait à partir des grandes entités décrites par le diagramme de composant, sur lequel on connecte les sous-composants. Les composants sont choisis en fonction des fonctionnalités extraites du fichier de configuration, tâche grandement simplifié grâce à la supposition "une feature correspond à un composant" faite précédemment.

```
if (HasFeature("Health"))
{
    UHealthAbsorber* healthAbsorber = NewObject<UHealthAbsorber>(damageHandler,
        GetComponentFromFeature(TEXT("Health")));
    damageHandler->addAbsorber(healthAbsorber);
}
```

Cette supposition facilite également la mise en place de la traçabilité, car si on sait qu'une fonctionnalité correspond à un composant, on peut facilement mémoriser des couples fonctionnalité-classe, grâce au framework de Unreal Engine qui propose des opérations sur les types.

Pour connecter les composants entre eux et vérifier le respects des interfaces nous utilisons des méthodes qui nous sont fournies par le framework Unreal Engine. Voici un exemple de la connexion des composants *ABCharacter* et *DamageHandler*

```
void ABCharacter :: Connect(UActorComponent * comp)
{
    //DamageHandler ?
    auto damageHandler = Cast<UDamageHandler>(comp);
    if (damageHandler != nullptr)
    {
        damageHandler->OuterActor = this;
        if (!damageHandler->IsRegistered())
        {
            damageHandler->RegisterComponent();
        }
        if (DamageHandler != nullptr)
        {
            DamageHandler->DestroyComponent();
        }
        DamageHandler = damageHandler;
    }
}
```

Dans cet exemple ci-dessus on vérifie si le composant que l'on souhaite connecter est du bon type. On vérifie ensuite si ce composant a déjà été enregistré. Si ce n'est pas le cas on peut alors l'enregistrer et s'y relier.

Dans l'exemple ci-dessous le composant *ATrigger* permettant de créer des zones d'interaction dans le jeu nécessite que l'objet auquel il est connecté fournit l'interface *Trigerrable*. Le composant utilise pour cela la méthode *bool ImplementsInterface*.

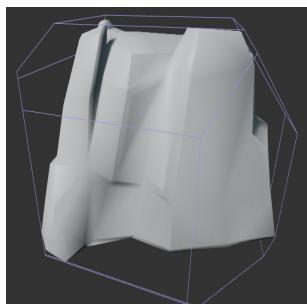
```
bool ATrigger::addObjectToTrigger(UObject* obj) {
    if (obj->GetClass()->ImplementsInterface(UTriggableInterface::StaticClass())) {
        triggerActions.Add(obj);
        return true;
    }
    return false;
}
```

5. Génération des armes et personnages

5.1. Bases de la programmation 3D

L'affichage de ressources 3D nécessite l'utilisation de plusieurs structures de données.

5.1.1. Mesh



La première structure est le maillage (ou mesh) qui contient la forme de l'objet à afficher.

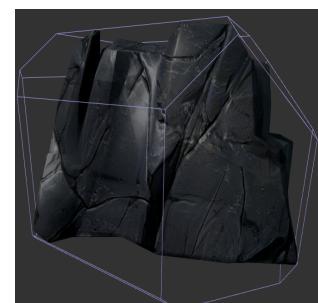
Un maillage contient l'ensemble des points de l'objet ainsi que les informations permettant de tracer les faces.

La plupart du temps ces faces sont triangulaires mais il est commun de trouver des maillages comportant des faces polygonales.

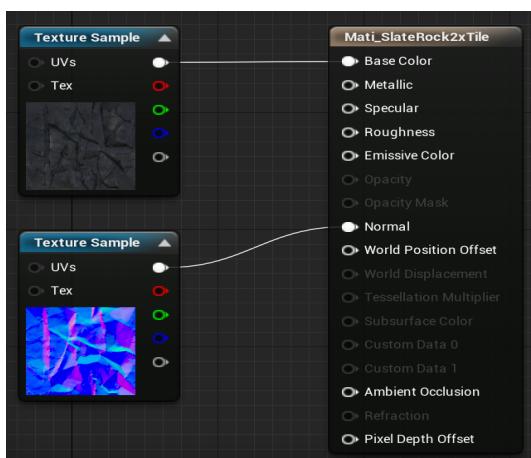
Un maillage comporte également un ensemble de vecteurs (un par point). Ces vecteurs appelés normales, sont utilisés pour le calcul des différents effets de lumières ainsi que pour des problèmes d'orientation de l'objet.

Chaque point comporte enfin 2 coordonnées u et v chacune comprises entre 0 et 1.

Ces coordonnées désignent un point sur une image texture qui sera plaquée sur l'objet selon l'ensemble des $[u, v]$.



5.1.2. Matériaux

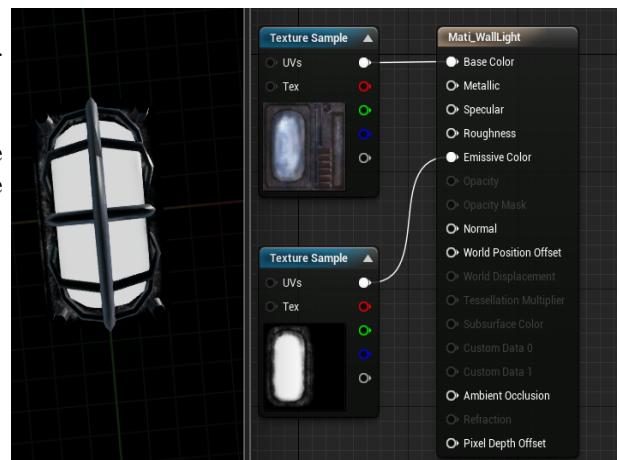


Un matériau est un ensemble d'images et de variables qui vont finir le rendu de l'objet affiché.

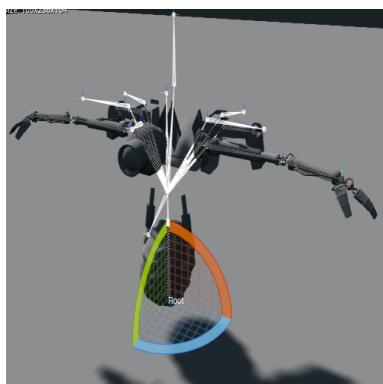
En plus de la texture plaquée sur l'objet, certaines images peuvent être utilisées pour remplacer les normales de l'objet lors de certains calculs de lumières et obtenir certains effets indépendamment de la forme de l'objet.

On peut également utiliser certaines textures pour définir des zones lumineuses sur l'objet.

Ces zones prendront alors la couleur de la carte de couleurs émissives, mais ont cependant besoin d'une vraie source de lumière afin que l'effet soit complet.



5.1.3. Animations



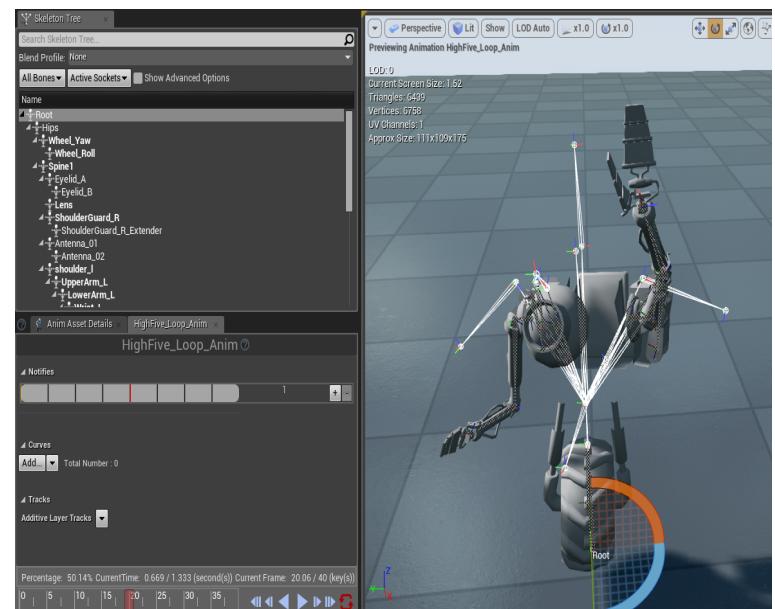
Afin d'animer cet ensemble, une autre structure de données est utilisée pour ne pas avoir à bouger individuellement chaque point du maillage.

Le squelette est donc un ensemble de points qui sera déplacé lors d'une animation.

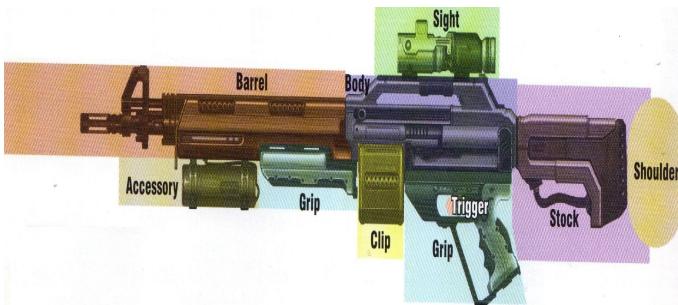
Chaque point du maillage est relié (rigged) à un point du squelette, ce qui permet d'appliquer facilement aux zones concernées un même ensemble de transformations linéaires (Translation, rotation, ...).

Une animation est découpée en plusieurs images (frames) qui représentent une suite de transformations pour chaque point du squelette.

Le moteur décide quant à lui de la vitesse à laquelle chaque image de l'animation sera affichée.



5.2. Gestion procédurale



Dans Borderlands chaque arme est une composition de plusieurs type pièces.

Lors de la création d'une nouvelle arme, une pièce est tirée au hasard dans chaque catégorie puis emboîtée sur l'ensemble.

L'animation de l'arme est elle définie par le corps de l'arme et tout l'ensemble utilise cette même animation.

Il en va de même pour les personnages qui sont composés d'un corps et d'une tête qui peut être changée pour pouvoir différencier plusieurs joueurs utilisant le même personnage.

Il est également possible de changer la texture de la tête et du corps sur certains modèles pour les personnaliser d'avantage.

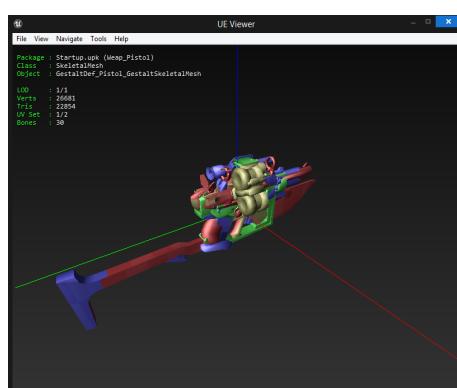


5.3. Extraction de ressources

Toutes les ressources du jeu proviennent de deux Borderlands différents :

- **Borderlands 2 : Game Of The Year Edition** a fourni les armes du jeu ainsi que les ennemis et les objets décoratifs.
- **Borderlands : The Pre-Sequel** a servi pour extraire le personnage jouable et ses animations.

Ces ressources sont cependant empaquetées dans des archives upk, format utilisé par les moteurs unreal afin de redistribuer les jeux aux utilisateurs.



Afin de pouvoir extraire des ressources de ces jeux, nous avons utilisé UE Viewer, un outils qui permet de lire le contenu de ces archives.

UE Viewer permet ensuite d'extraire le contenu au format PSK pour les maillages et squelettes, tga pour les textures et psa pour les animations.

5.4. Traitements

Malheureusement Unreal Engine 4 n'utilise que les formats FBX et OBJ pour importer maillages, squelettes et animations, il est donc impossible d'utiliser directement les ressources précédemment extraites.

De plus certains modèles comme ceux des armes nécessitent un lourd traitement avant de pouvoir remplir leur rôle.

5.4.1. Conversion

La première étape de traitement consiste donc à importer les modèles et animations dans un logiciel de CAO (Conception Assistée par Ordinateur) afin de pouvoir modifier les ressources puis de les exporter aux formats compatibles avec le moteur.

Pour cela nous avons étudié plusieurs possibilités :

- Un ensemble de scripts d'imports PSK/psa et d'export FBX depuis Blender.
- Les scripts ActorX pour 3ds max.
- Milkshape un logiciel d'import et export spécialisé dans les moteurs de jeux.



Fig. 10. Blender

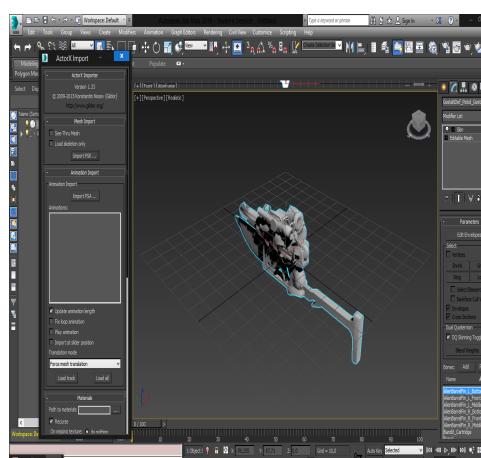


Fig. 11. 3ds Max

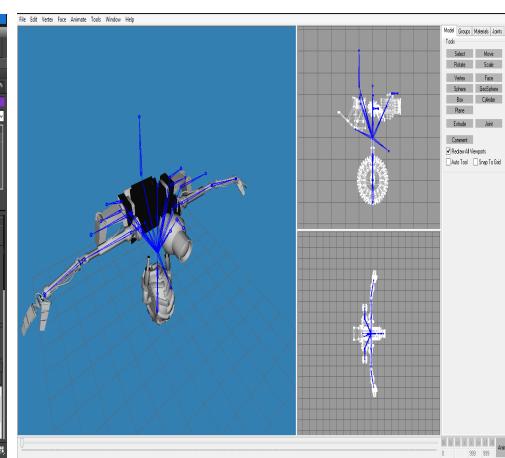
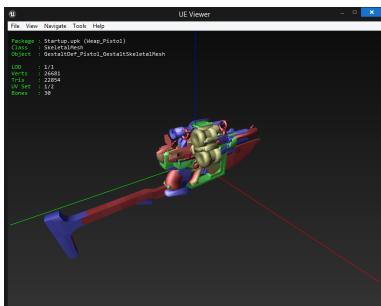


Fig. 12. Milkshape

Cette étape consiste donc simplement à exporter le maillage chargé dans un de ces logiciels, ainsi que ses animations associées, au format FBX qui est le seul format reconnu par Unreal Engine 4 pour l'importation d'animations.

5.4.2. Coloration



Comme on a pu le voir précédemment, toutes les pièces d'un même type d'arme sont rassemblées dans un même fichier, et comme ceux-ci utilisent le même squelette pour garantir leur assemblage et l'animation de l'ensemble, le fichier de base ressemble à un tas désordonné et inexploitable.

De plus la coloration obtenue sur UE Viewer qui présente une piste intéressante n'est pas sauvegardée lors de l'extraction de la ressource depuis le jeu original.

Une première méthode consiste donc à reproduire cette carte des couleurs, et attribuer de manière plus spécifique une couleur ou une nuance d'une couleur unique à chaque pièce, et non à un groupe de pièces comme sur UE Viewer.

La création d'une carte de couleurs se fait facilement avec l'API python de Blender. Pour cela il suffit de parcourir toutes les faces du maillage et d'attribuer à la face courante une couleur liée à son indice dans le parcours.

Si l'on prend par exemple le maillage des pièces de pistolets qui comporte 68562 faces. On commence par calculer un pas $step = \frac{1.0}{68562.0}$ permettant de ramener l'indice de la face à une nuance de couleur comprise dans $[0; 1]$ par $col = (ind * step)$ et $rgb = [col/2.0, col, col/4.0]$ pour obtenir des nuances de vert.

```
import bpy
import bmesh

#Récupération de l'objet
bpy.ops.object.mode_set(mode='OBJECT')
my_object = bpy.data.objects['GestaltDef_Pistol_GestaltSkeletalMesh'].data

print('Début de coloration')
#Initialisation de la carte de couleurs
color_map_collection = my_object.vertex_colors
if len(color_map_collection) == 0:
    color_map_collection.new()

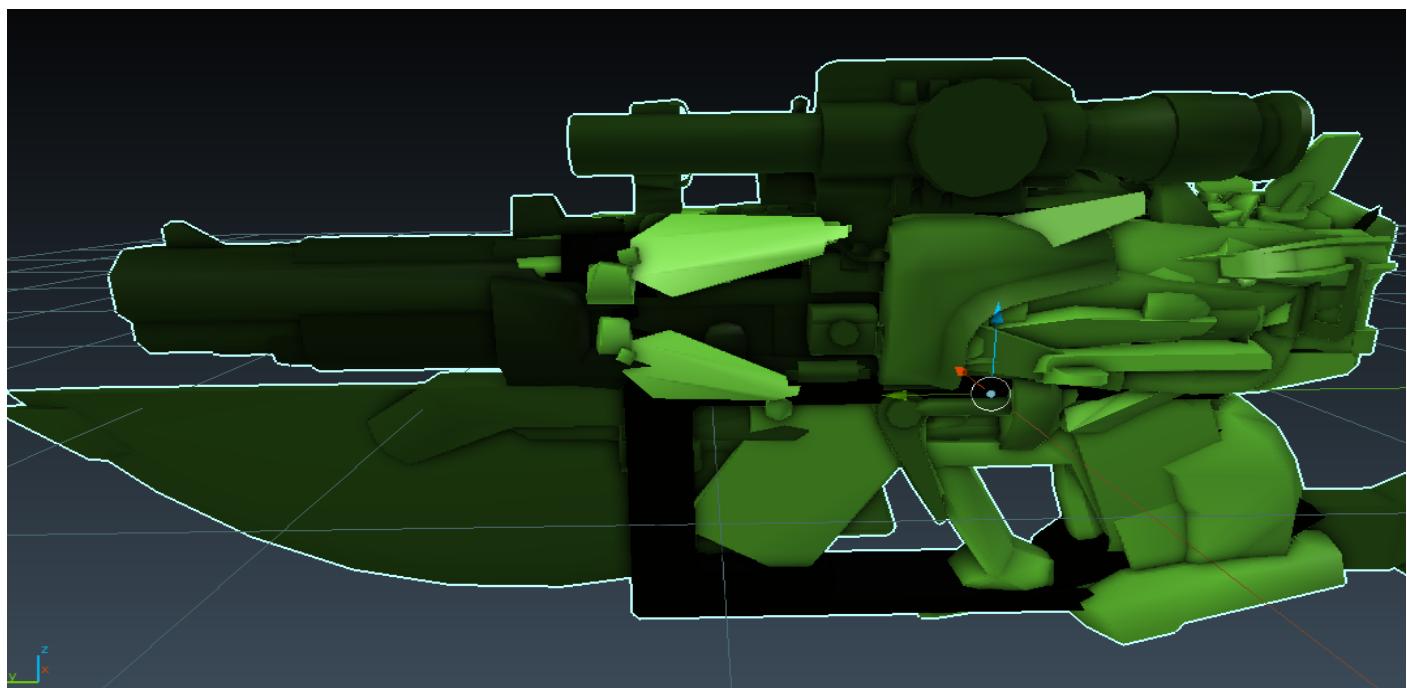
color_map = color_map_collection['Col']

rgbStep = 1.0 / 68562.0
i = 0
rgb = [0,0,0]
ratio = 0.0

#Parcours des faces et coloration
for poly in my_object.polygons:
    for idx in poly.loop_indices:
        ratio = (i * rgbStep)
        rgb = [ratio/2.0,ratio,ratio/4.0]
        color_map.data[i].color = rgb
        i += 1
print('Coloration du modèle terminée')

#Passage en mode coloration pour afficher le résultat
bpy.ops.object.mode_set(mode='VERTEX_PAINT')
```

On obtient la coloration suivante :



On peut remarquer qu'une segmentation plus fine s'est mise en place avec une couleur par pièce. Cependant les parties qui se succèdent ont une coloration trop peu nuancée pour pouvoir être séparées correctement.

Cette coloration montre en revanche que le modèle n'a pas été obfuscué d'avantage (en distribuant les faces au hasard dans le modèle par exemple) et qu'un autre traitement est possible.

5.4.3. Découpage

Une autre méthode de segmentation consiste à ne garder du maillage que les faces dont l'indice appartient à un intervalle donné.

Voici une implémentation de cette méthode pour l'API python de Blender.

```
import bpy
import bmesh

#Récupération de l'objet sélectionné
bpy.ops.object.mode_set(mode='OBJECT')
current_OBJ = bpy.context.active_object

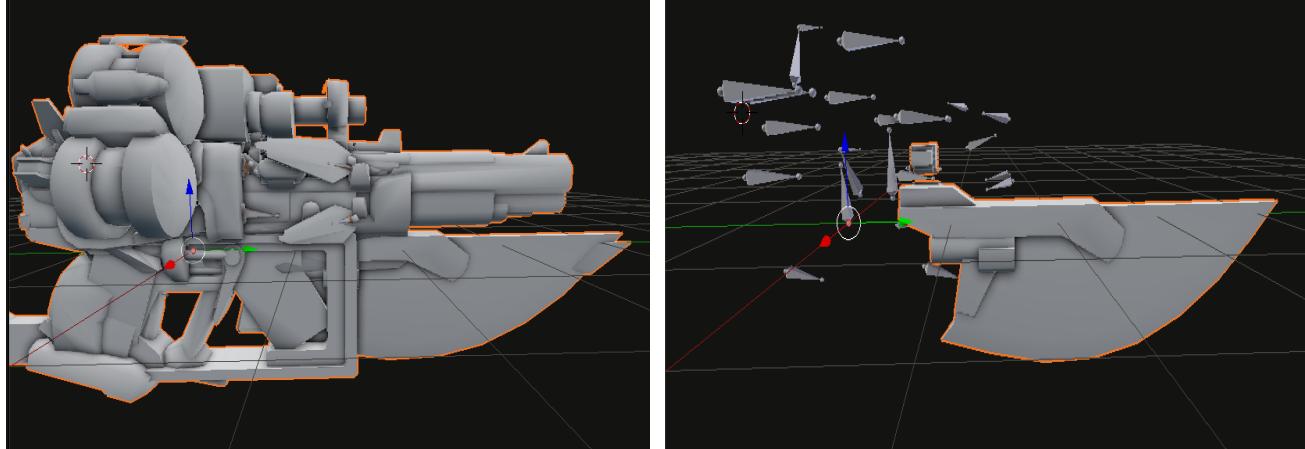
#Définition d'une zone de faces à conserver (Ici extraction d'une baïonnette)
borneInf = 7868
borneSup = 8350

#Réinitialisation de la sélection courante
bpy.ops.object.mode_set( mode = 'EDIT' )
bpy.ops.mesh.select_all( action = 'DESELECT' )
bpy.ops.mesh.select_mode( type = 'FACE' )

#Parcours complet du maillage et sélection des faces à effacer
bm = bmesh.from_edit_mesh( current_OBJ.data )
i = 0
for f in bm.faces:
    f.select = not ( i < borneSup and i >= borneInf )
    i += 1

#Suppression des faces
bm.select_flush(True)
bpy.ops.mesh.delete(type='FACE')
bpy.ops.object.mode_set(mode='OBJECT')
```

L'exécution de ce script abouti au résultat suivant :



Cette méthode fastidieuse permet d'extraire toutes les pièces nécessaires du maillage de base, tout en conservant le squelette ainsi que le rig de la pièce obtenue.

Toute les pièces extraites par ce procédé pourront donc utiliser les mêmes animations que le reste de l'ensemble.

5.4.4. Assemblage

L'assemblage des différentes parties du modèle se fait du côté du moteur.

Le principe de cette opération consiste à prendre deux objets 3D, ne possédant pas nécessairement le même squelette, et d'appliquer sur l'objet fixé les transformations linéaires d'un point d'attachement (socket) présent sur l'autre objet.

Dans Unreal Engine 4 les maillages sont importés sous forme de USkeletalMeshComponent. La méthode la plus simple pour charger et attacher deux objets ensemble consiste à créer deux USkeletalMeshComponent, récupérer les maillages souhaités grâce à des ConstructorHelpers, attribuer les maillages aux composants et enfin les attacher.

Si on prend comme exemple le personnage jouable on obtient :

```
//Corps du personnage
MeshComp = CreateDefaultSubobject<USkeletalMeshComponent>(TEXT( "MyPawnMesh" )) ;

const ConstructorHelpers :: FObjectFinder<USkeletalMesh>
MeshObj(TEXT( "SkeletalMesh '/Game/ [ . . . ] /Skel_FragTrap.Skel_FragTrap'" )) ;

MeshComp->SetSkeletalMesh(MeshObj. Object ) ;

//Tete du personnage
HeadMesh = CreateDefaultSubobject<USkeletalMeshComponent>(TEXT( "MyPawnHeadMesh" )) ;

const ConstructorHelpers :: FObjectFinder<USkeletalMesh>
HeadMeshObj(TEXT( "SkeletalMesh '/Game/ [ . . . ] /Che_GuevaTrap.Che_GuevaTrap'" )) ;

HeadMesh->SetSkeletalMesh(HeadMeshObj. Object ) ;

//Attachement du corps sur le socket Head de la tête .
MeshComp->AttachTo(HeadMesh, TEXT( "Head" ), EAttachLocation :: SnapToTargetIncludingScale, true) ;
```



Pour l'assemblage d'une arme, les modèles de chaque pièces ont été répartis dans plusieurs catégories.

Une arme sera donc constituée d'un corps, d'une poignée, d'un canon et éventuellement d'un accessoire.

Afin de gérer les collections de pièces pour les différents types d'armes, nous avons répertorié les localisations des pièces extraites ainsi que leurs types, catégories et manufacturier dans un fichier xml.

```
<Root>
  <WeaponType value="pistols">
    <category value="body">
      <Part manufacturer="Bandit">
        <description>
          <desc name="mesh" value="SkeletalMesh '/Game/ [...] / body_bandit.body_bandit'"></desc>
          <desc name="fireAnim" value="AnimSequence '/Game/ [...] / fire_auto.fire_auto'"></desc>
          <desc name="reloadAnim" value="AnimSequence '/Game/ [...] / reload_bandit.reload_bandit'"></desc>
          <desc name="damage" value="10" />
          <desc name="ammo" value="20" />
          <desc name="fireRate" value="10" />
        </descriptions>
      </Part>
    </category>
    <category value="canon">
      <Part manufacturer="Bandit">
        <descriptions>
          <desc name="mesh" value="SkeletalMesh '/Game/ [...] / canon_bandit.canon_bandit'"></desc>
          <desc name="fireRate" value="+0.10" />
          <desc name="damage" value="-0.10" />
        </descriptions>
      </Part>
    </category>
  </WeaponType>
</Root>
```

Ce fichier contient également les chemins vers les différentes animations nécessaires pour lancer les animations de recharge sur l'arme ainsi que sur le personnage qui la tient.

Nous n'avons cependant pas trouvé les textures des armes dans les ressources des deux jeux, les armes sont donc importés avec un matériaux par défaut pour chaque catégorie de pièce.

Du coté du moteur, le contenu du fichier xml est stocké, via le parser en annexe, dans les structures suivantes :

```
//Contient toutes les informations d'une pièce d'arme et son manufacturier
USTRUCT()
struct FPart{
  GENERATED_USTRUCT_BODY()
  FString manufacturer;
  TMap<FString, FString> desc;
  FString getValue(FString key){ return desc[key]; }
};

//Définit l'ensemble des pièces pour un type d'arme
//Le but est de demander une partie de chaque collection pour constituer une arme de manière procédurale
USTRUCT()
struct FWaponsParts{
  GENERATED_USTRUCT_BODY()
  FString wtype;
  TArray<FPart*> bodies;
  TArray<FPart*> canons;
  TArray<FPart*> handles;
  TArray<FPart*> others;
  //Donne une FPart aléatoire en fonction de la catégorie passée en paramètre
  FPart* getRandom(FString);

  //Ajoute la FPart à la collection de la catégorie passée en paramètre
  void Add(FPart*, FString);
};
```

Chaque type d'arme est donc représenté par une FWeaponsParts et une collection globale de ces structures représente l'ensemble des pièces de toutes les armes.

```
UCLASS()
class BORDERLANDS_API AWeaponGraphic : public AActor{
GENERATED_BODY()

static TArray<FWeaponsParts*>* weaponsParts;
TArray< class USkeletalMeshComponent*> meshes;
}
```

L'assemblage de toutes les parties d'une arme se déroule de manière similaire à l'assemblage d'un personnage.

L'unique différence consiste à attacher chaque partie, peu importe sa catégorie, à un USceneComponent qui sera la racine de l'arme.

```
void AWeaponGraphic::RandomPistol(){
    FWeaponsParts* pistols = AWeaponGraphic::getWtype(FString("pistols"));
    FPart* gunBodyP = pistols->getRandom(FString("body"));
    FPart* gunCanonP = pistols->getRandom(FString("canon"));
    FPart* gunHandleP = pistols->getRandom(FString("handle"));

    //Recherche des skeletal mesh pour les différentes parties
    const ConstructorHelpers::FObjectFinder<USkeletalMesh>
    gunBMeshObj( *(gunBodyP->getValue(FString("mesh"))));
    const ConstructorHelpers::FObjectFinder<USkeletalMesh>
    gunCMeshObj( *(gunCanonP->getValue(FString("mesh"))));
    const ConstructorHelpers::FObjectFinder<USkeletalMesh>
    gunHMeshObj( *(gunHandleP->getValue(FString("mesh"))));

    //Creation des composants
    RootComponent = CreateDefaultSubobject<USceneComponent>(TEXT("gunRoot"));
    meshes.Add(CreateDefaultSubobject<USkeletalMeshComponent>(TEXT("gunBMesh")));
    meshes.Last()->SetSkeletalMesh(gunBMeshObj.Object);

    meshes.Add(CreateDefaultSubobject<USkeletalMeshComponent>(TEXT("gunCMesh")));
    meshes.Last()->SetSkeletalMesh(gunCMeshObj.Object);

    meshes.Add(CreateDefaultSubobject<USkeletalMeshComponent>(TEXT("gunHMesh")));
    meshes.Last()->SetSkeletalMesh(gunHMeshObj.Object);
}

//Attachement des pièces
void AWeaponGraphic::bind(){
    if( meshes.Num() > 2 ){
        for(size_t i = 0; i != meshes.Num(); ++i){
            meshes[i]->AttachTo( RootComponent, TEXT("Root"),
                EAttachLocation::SnapToTargetIncludingScale, true );
        }
    }
}

//Constructeur
AWeaponGraphic::AWeaponGraphic(){
    // Set this actor to call Tick() every frame.
    PrimaryActorTick.bCanEverTick = true;
    RandomPistol();
    bind();
}
```

Voici un exemple de génération aléatoire de pistolet suivant cette méthode :



5.5. Animation

L'animation de l'ensemble consiste simplement à jouer une même UAnimSequence à toutes les parties de l'objet.

La récupération d'une animation se fait exactement de la même manière qu'un maillage ou un matériau.

```
UCLASS()
class BORDERLANDS_API AWeaponGraphic : public AActor{
GENERATED_BODY()

static TArray<FWeaponsParts*>* weaponsParts;
TArray< class USkeletalMeshComponent* > meshes;
TArray< FPart* > parts;
class UAnimSequence* reloadAnim;
class UAnimSequence* fireAnim;

public:
    virtual void defaultWeap();
    virtual void bind();
    virtual void fire();
    virtual void reload();
};

//Lecture de l'animation de tir
void AWeaponGraphic::fire(){
    if( meshes.Num() > 2 ){
        for(size_t i = 0; i != meshes.Num(); ++i){
            meshes[ i ]->PlayAnimation(fireAnim, false);
        }
    }
}

//Lecture de l'animation de recharge
void AWeaponGraphic::reload(){
    if( meshes.Num() > 2 ){
        for(size_t i = 0; i != meshes.Num(); ++i){
            meshes[ i ]->PlayAnimation(reloadAnim, false);
        }
    }
}
```

5.6. Comparaison de solutions

Au cours de ce TER nous avons utilisé Milkshape pour convertir les ressources extraites par UE Viewer au format FBX puis Blender pour les modifier.

Voici une rapide étude des solutions que nous avons considéré, présentant les points forts de chaque solution ainsi que les raisons pour lesquelles certaines n'ont pas été retenues.

5.6.1. 3ds Max

Le créateur d'UE Viewer ayant codé deux maxscript pour le chargement de ressources dans 3ds max, la solution d'Autodesk semblait la plus adaptée pour le traitement des ressources.

Cependant, bien qu'il soit possible de découper les maillages comme sur blender, 3ds max force une conversion de la ressource dans un de ses formats (editable Mesh ou editable Poly).

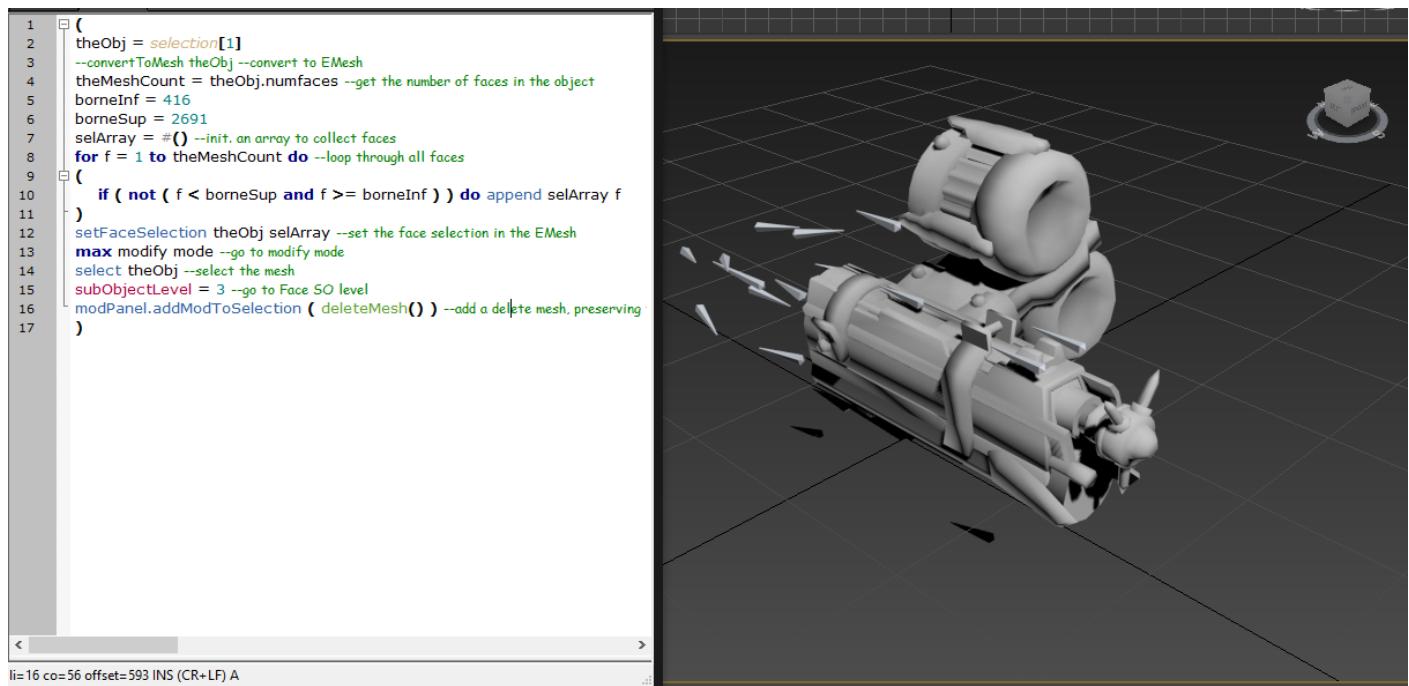


Fig. 13. Interface 3ds max lors de l'exécution d'un script

Cette conversion du maillage touche également le squelette qui se retrouve à faire partie du maillage dans la nouvelle structure.

Même en cachant le squelette dans l'éditeur, la conversion finit toujours par détruire le rig du maillage ce qui rend le résultat du traitement impossible à animer, à moins de refaire entièrement le lien entre chaque point du maillage et du squelette manuellement.

Pour cette raison nous n'avons donc pas retenu 3ds max comme solution pour le traitement des ressources.

5.6.2. Blender

L'avantage de Blender sur 3ds max est de cloisonner le maillage et le squelette durant chacune des opérations.

Il n'y a donc aucune conversion avec perte d'information durant le traitement d'une ressource. Cependant Blender présente deux soucis majeurs d'import et d'export :

- l'import d'animations au format PSA ne permet de charger qu'une seule animation par fichier (uniquement la première s'il y en a plusieurs). Le format stockant intégralement les données en binaire, il est impossible de séparer les animations en plusieurs fichiers sans créer son propre script d'importation
- L'export au format FBX peu importe la version rajoute systématiquement un noeud racine supplémentaire ce qui rend le travail effectué incompatible avec les animations.

Heureusement le problème d'export a pu être corrigé par une modification du script d'export FBX ASCII de Blender.

5.6.3. MilkShape

Milkshape est un outils spécialisé dans l'import de ressources en provenance de jeux vidéos.

Le but de ce logiciel est de proposer plusieurs profils d'importations pour un même format correspondants aux différentes variantes mises en place pour chaque jeu.

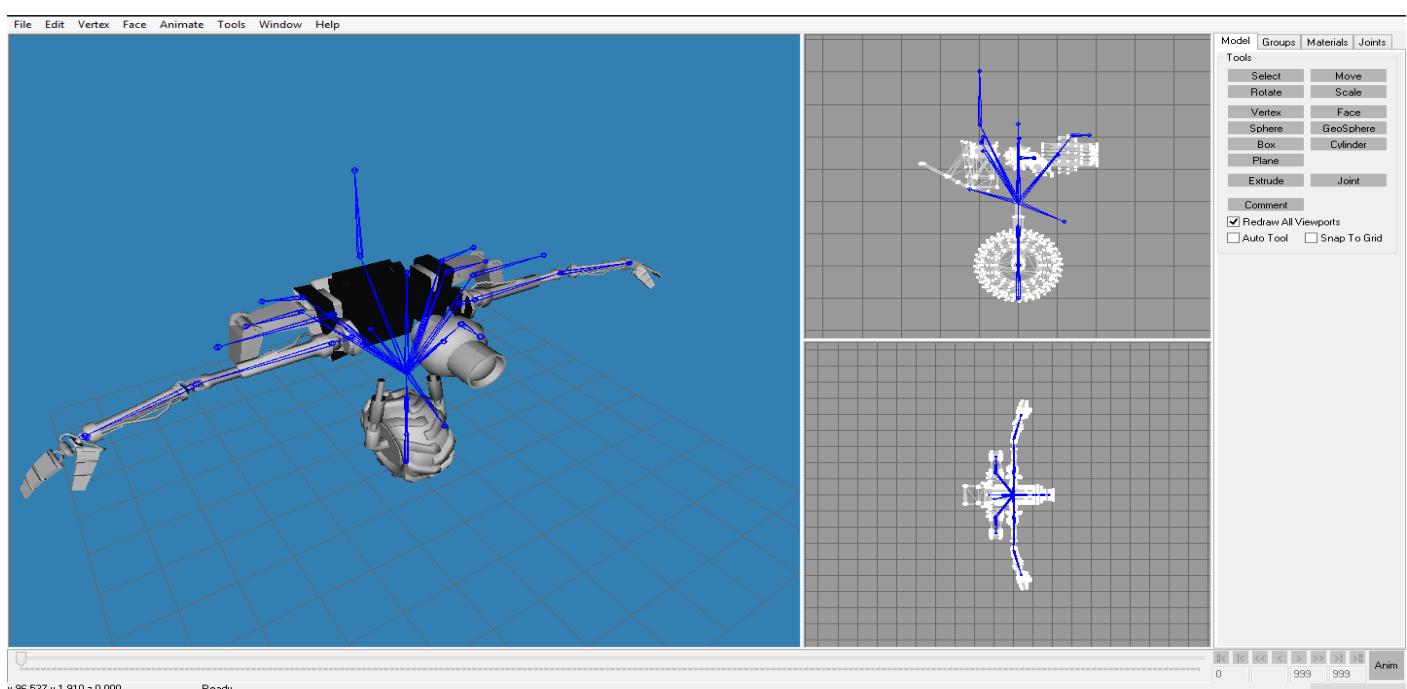


Fig. 14. Interface de Milkshape

Les fonctions d'import et d'export de ressources extraites de jeux sont les meilleures que l'on puisse trouver et remplissent leur rôle sans soucis.

Cependant Milkshape ne permet aucune édition de contenu via un langage de script, il n'est donc pas utilisable seul et une autre solution comme Blender est donc nécessaire pour terminer le traitement.

6. Perspectives

6.1. Apprentissage

Ce projet nous a permis d'approfondir nos connaissances dans la programmation par composant, ainsi que l'utilisation d'un framework aussi complexe que celui de Unreal Engine.

L'implémentation de systèmes complexes, comme les implémentations de composants nous a rendu soucieux de la qualité des solutions que nous avions à créer, tant sur le plan de la complexité des algorithmes que sur le plan de l'espace utilisé.

L'architecture à base de composants nous a fait réfléchir sur la modularité et la réutilisation de notre projet, en totalité comme en parties.

Enfin l'utilisation d'un framework nous a permis d'utiliser de nouvelles ressources, comme les textures ou la musiques, et de les exploiter pleinement en créant des animations ou des effets de particules.

6.2. Portage sur système Android

La portabilité sur plusieurs Systèmes d'exploitations est assurée par Unreal Engine. Cependant, les limites matérielles des smartphones nécessitent l'utilisation d'assets graphique allégés.

6.3. Mise en place d'un système multijoueurs en ligne

Le mode multijoueur que nous n'avons pu développer reste d'actualité. Cela nous permettra d'aborder le problème de la réPLICATION des variables entre serveur et clients

6.4. Autres

Un dernier ajout, visant purement à améliorer le confort de jeu, serait l'ajout d'un support manette. En effet, Unreal Engine permet de récupérer les entrées sur manette, mais certains comportement des entités aurait besoin d'une refonte complète pour prendre en compte les entrées de la manette.

7. Remerciements

Nous remercions tout d'abord M. Abdelhak-Djamel SERIAI, notre encadrant, pour avoir accepté de soutenir notre projet ainsi que de nous avoir initié au concept de ligne de produit.

Nous remercions Microsoft de fournir gracieusement une version de son EDI Visual Studio 2015.

Nous remercions également Github pour avoir hébergé notre dépôt GIT.

Nous remercions Epic Games pour avoir fourni un framework adapté à nos besoins.

8. Annexe

Code complet du parser xml

```

void AWeaponGraphic::loadXML(){
    if( AWeaponGraphic::weaponsParts == NULL ){
        AWeaponGraphic::weaponsParts = new TArray<FWaponsParts*>();
        /*Lecture d'un fichier XML*/
        FString pathToFile = FPaths::GameContentDir(); pathToFile+="Borderlands/weapons/weapons.xml";
        FXmlFile *file = new FXmlFile(pathToFile);

        FWaponsParts* collection = NULL;
        FString cat;
        FPart* cur = NULL;

        if ( file->IsValid() ){
            FXmlNode* root = file->GetRootNode();
            if( root != nullptr ){
                TArray<FXmlNode*> weaponTypes = root->GetChildrenNodes();
                for( FXmlNode* weaponType : weaponTypes ){
                    collection = AWeaponGraphic::getWtype( weaponType->GetAttribute(FString("value")));
                    TArray<FXmlNode*> categories = weaponType->GetChildrenNodes();
                    for( FXmlNode* category : categories ){
                        cat = category->GetAttribute(FString("value"));
                        TArray<FXmlNode*> parts = category->GetChildrenNodes();
                        for( FXmlNode* part : parts ){
                            cur = new FPart();
                            cur->manufacturer = part->GetAttribute(FString("manufacturer"));
                            TArray<FXmlNode*> descs = part->GetChildrenNodes()[0]->GetChildrenNodes();
                            for( FXmlNode* d : descs){
                                cur->desc.Add( d->GetAttribute(FString("name")),
                                d->GetAttribute(FString("value")) );
                            }
                            collection->Add(cur,cat);
                        }
                    }
                }
            }
            else{ UE_LOG(LogTemp, Warning, TEXT("No root")); }
        }
        else{ UE_LOG(LogTemp, Warning,TEXT("Fichier introuvable : %s"), *pathToFile); }
    }
}

```

References

- [1] Dépot Github du projet <https://github.com/yongaro/Borderlands>
Sous module des ressources 3D du projet https://github.com/yongaro/Borderlands_assets
- [2] Software Engineering Institute <http://www.sei.cmu.edu/productlines/>
IBM Research https://www.research.ibm.com/haifa/projects/services/product_lines/index.shtml
Software Product Line Engineering with Feature Models <http://www.pure-systems.com/fileadmin/downloads/pure-variants/tutorials/SPLWithFeatureModelling.pdf>
Composants agiles et réutilisation du logiciel : vers les lignes de produits logiciels http://www.exibri.com/published_docs/AgileTour2010-exibri_Composants-agiles-et-reutilisation-du-logiciel_2010-10-07.pdf
Domain Engineering https://www4.cs.fau.de/Lehre/SS03/V_OSE/Skript/03ose-A5.pdf
- [3] Extraction de ressources des jeux utilisant Unreal Engine <http://www.gildor.org/en/projects/umodel>
- [4] Import de ressources psk et psa dans 3ds max <http://www.gildor.org/projects/unactorx>
- [5] Import de ressources psk et psa dans Blender (inclus de base à partir de Blender 2.7)
https://wiki.blender.org/index.php/Extensions:2.6/Py/Scripts/Import-Export/Unreal_psks