Arthur Chateauneuf **M1 Informatique, Imagine**

Compte Rendu : Projet Moteur de jeu

Sanctia : Proof Of Concept

Lien github du projet : https://github.com/MonsieurBleu/Sanctia

Lien des vidéos de présentation :

- Système de dialogues : https://www.youtube.com/watch?v=I2dEp3ZyjQs
- Système de combat : https://www.youtube.com/watch?v=BDIPG-vZeGk
- Bataille de PNJ : https://www.youtube.com/watch?v=NQf-IKH4k7s

Contrôles :

Mouvements et combats :

- ZQSD : déplacements
- Shift : courir
- Espace : saut
- F : coup de pied
- Clic droit : défense
- Clic gauche : attaque
- Clic molette : attaque lourde
- Coup de souris vers la droite/gauche : changer de côté d'attaque
- Flèches directionnelles : changer de côté d'attaque

Utilitaires :

- F2 : activer/désactivé le suivit de la souris de la caméra
- F11 : pause
- F12 : mode spectateur
- E : lancer un dialogue
- P : Ajouter un ennemi qui attaquera le joueur
- Backspace : tuer le joueur
- Right Shift : stun le joueur
- 0 et 9 : active les helpers de physique ou d'entités

Vulpine Engine

Présentation du moteur

Le moteur de jeu que j'ai développé s'appelle Vulpine. Ce projet est en cours depuis juillet 2023 et est également travaillé dans le cadre de mon Projet TER. Voici une liste succincte des fonctionnalités présentes :

- Rendu de scènes 3D et 2D à l'aide de OpenGL
- Foward rendering et Clustered Rendering
- Post-processing et early depth testing
- Ombres douces
- Moteur audio avec OpenAL
- Rendu de police d'écriture MSDF avec markdown simple
- Moteur Physique basique
- Interface rapide de debug.
- Entity Component System
- Animations squelettiques et graphes d'animations

Ce rapport ne concernera que les éléments sur lesquels j'ai travaillés dans le cadre de cette matière.

Chargement d'assets

Afin de simplifier le développement de ce projet j'ai créé un système de chargement des ressources automatique. Ce dernier s'aide d'un format textuel simple, inspiré du json, que j'ai créé. Dans ce dernier, un « : » correspond à l'entrée d'une nouvelle donnée, suivi de son type ou de son nom. Si la ressource est définie à cet endroit, les informations relatives à son chargement son données et finissent par un « ; ». Sinon, un « | » suivit du nom de la ressource signifie que l'on référence la ressource sans la créer.

Voici un résumé du fonctionnement de ce système :

- Lors de la compilation : pour chaque structure/classe dont une fonction de chargement a été créé, une série de hashmap vont être crées :
    - Map associant le nom de la ressource avec les informations de sa création
    - Map associant le nom de la ressource avec ses données chargée

- Lors du lancement de l'application : le fichier de ressource définie (ici « data ») est scanné de façon récursive. Toutes les ressources dont l'extension est reconnue sont chargé dans la map d'informations. Le code de chargement des formats simple comme KTX ou VulpineSkeleton est généré automatiquement.

- Lorsqu'une ressource est demandée, le moteur va vérifier si elle est chargée en mémoire. Si ce n'est pas le cas, les informations sont retrouvées et utilisées pour charger la ressource. Si aucune information est disponible, ou si un problème est survenu, un objet vide est renvoyé et un message d'erreur détaillé est affiché dans le flux d'erreur.

Le processus de chargement est récursif. C'est-à-dire que si le chargement d'un modèle nécessite le chargement d'une texture, le même processus est suivi jusqu'à ce que tout soit disponible. De façon similaire, si deux modèles différents utilisent la même mesh, cette dernière ne sera chargée qu'une seule fois. Voici comment demandé une ressource dans Vulpine :

```
scene.add(
    Loader<MeshModel3D>::get("barrel01").copy()
);
```

Le code en lien avec ce système peut être retrouvé dans les fichiers sources et header AssetManager du dossier Engine.

Entity Component System

J'ai également implémenté un ECS dans le cadre de ce projet. Ce dernier permet de créer à la compilation une liste de tous les composants qui sont répartis en plusieurs catégories : Entity List, Data, Graphic, Physic, Sound et AI. Une entité contient ainsi son ID dans chacune de ces catégories. Cela permet, par exemple, d'avoir un nombre d'entité maximal différent selon les catégories. Voici comment créer une entité dans ce système (src/EntityBlueprint.cpp) :

```
EntityRef zweihander(new Entity("ZweiHander"
    , ItemInfos{100, 10, DamageType::Slash, B_Collider().setCapsule(0.1, vec3(0, 0, 0), vec3(1.23, 0, 0))}
    , Effect()
    , EntityModel{Loader<ObjectGroup>::get("Zweihander").copy()}
    , ItemTransform{}
    , PhysicsHelpers{}
));
```

Les composants peuvent être définis dans le constructeur grâce à des paramètres variadiques ou être manipulés avec les méthodes « set », « removeComp », « hasComp » ou « comp ».

Voici désormais comment créer un composant (include/SanctiaEntrity.hpp) :

```
/*************** GRAPHICS ***************/
    const int MAX_GRAPHIC_COMP_USAGE = MAX_ENTITY;

    struct EntityModel : public ObjectGroupRef{};

    COMPONENT(EntityModel, GRAPHIC, MAX_GRAPHIC_COMP_USAGE);
    template<>void Component<EntityModel>::ComponentElem::init();
    template<>void Component<EntityModel>::ComponentElem::clean();

    COMPONENT(SkeletonAnimationState, GRAPHIC, MAX_GRAPHIC_COMP_USAGE);
    template<>void Component<SkeletonAnimationState>::ComponentElem::init();

    COMPONENT(AnimationControllerRef, GRAPHIC, MAX_GRAPHIC_COMP_USAGE);
```

Les systèmes sont définis comme de simples fonctions à template, prenant en paramètre une fonction qui serra appliquer à toutes les entités correspondant à la template. Afin de rapidement trier les entités, un bitset de masque sera généré lors de la première exécution. Voici un exemple de création de système (src/Game__physicsloop.cpp) :

```
/***** ALL STUNED OR BLOCKING ENTITY ARE IMMOBILE *****/
    System<B_DynamicBodyRef, ActionState>([](Entity &entity){
        auto &s = entity.comp<ActionState>();

        if(s.stun || s.blocking)
            entity.comp<B_DynamicBodyRef>()->v *= vec3(0, 1, 0);
    });
```

Le code en lien avec ce système peut être retrouvé dans les fichiers sources et header Entity du dossier Engine.

Sanctia

Présentation du projet

Sanctia est un projet de jeu de rôle à la première personne sur laquelle je me suis engagé avec un artiste. Nous sommes actuellement à l'étape de Pré-alpha de ce dernier. Mon rendu pour le projet de cette matière, quant à lui, est une version proof-of-concept contenant l'implémentation d'un système de dialogue et une première version de ce que nous imaginons pour les déplacements et les combats du jeu. Un agent de combat a également été développé en prenant en compte les différents éléments du système de combat.

Système de dialogue et de Scripting

Dans le cadre de ce projet, j'ai créé un système de dialogue pouvant être chargé à partir de fichiers markdown. L'objectif de ce format et de permettre de créer des embranchements et des pants de quêtes sans avoir à écrire du code redondant ou à recompiler le jeu. Ce dernier permet, par exemple, de créer un système d'affinité simplement et d'adapter n'importe quel dialogue au niveau d'affinités.

Ce système est compatible avec l'utilisation de plusieurs langues et permet pour chaque ligne de dialogue dites par le pj ou le pnj :

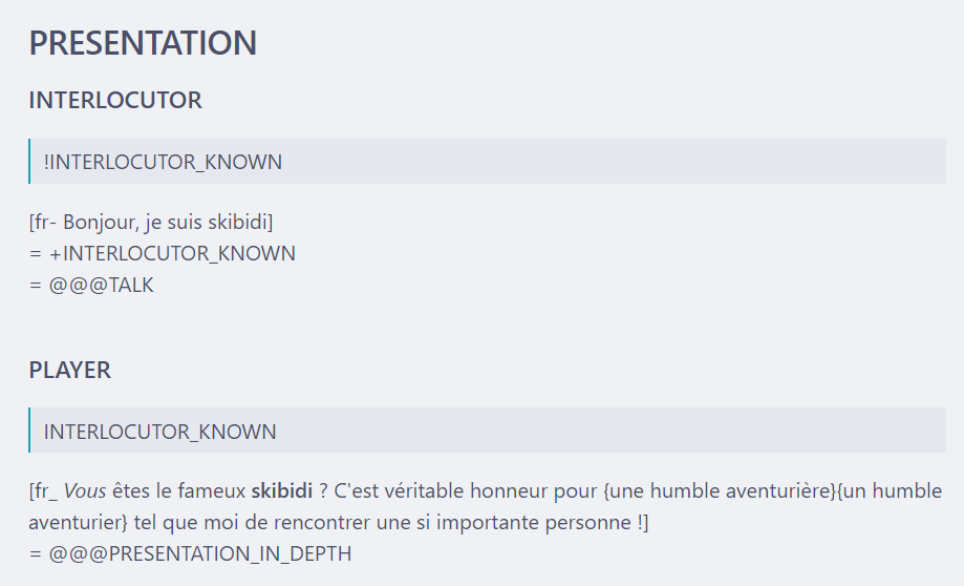
- Définir des conditions d'affichages du dialogue/choix
- Définir les embranchements et leur type
- Définir les conséquences : conditions qui seront misent à vrai ou faux
- Définir les évènements (exemple : le pj gagne +1 d'affinité avec le pnj)

Voici les instructions de misent en forme d'un fichier de dialogue :

Règles & mise en forme

Signe	Utilisation
#	Début de l'entrée d'un personnage
##	Début de l'entrée d'un dialogue du personnage
###...	Sous titres, seront ignorés par le parseur mais restent utiles pour l'organisation
>	Prérequis, toujours suivis d'une condition, COND_ALL si aucun prérequis
>!	Prérequis inversé
[]	Contient le texte qui sera affiché
fr_	Toujours après l'ouverture des [], détermine qu'il s'agit d'une réponse du PJ en français
fr-	Toujours après l'ouverture des [], détermine qu'il s'agit d'une réponse du PNJ en français
{F}{H}	F sera affiché si le joueur a un corps féminin, H sinon
=	tout le temps après la liste des textes entre [], nouvelle conséquence
= ~	conséquence vide
= +	toujours suivi d'une condition, met la condition à vrai
= -	toujours suivi d'une condition, met la condition à faux
= @	toujours suivi d'un dialogue, change le dialogue prononcé par l'interlocuteur
= @@	toujours suivi d'un dialogue, change la liste des réponses du joueur
= @@@	toujours suivi d'un dialogue, change la liste des réponses du joueur ET le dialogue prononcé par l'interlocuteur

Pour finir, voici un exemple de dialogue écrit avec ces règles :



Ici, le pnj dira sa présentation et le joueur aura à nouveau les options de dialogues de base (« TALK », lorsque le joueur initie la conversation). Cependant, un nouveau choix sera présenté au joueur, ce qui entrainera le pnj à se présenter en profondeur si ce dernier est choisis.

Le code en lien avec ce système peut être retrouvé dans les fichiers source et header Dialogue et DialogueController.

Système de mouvement

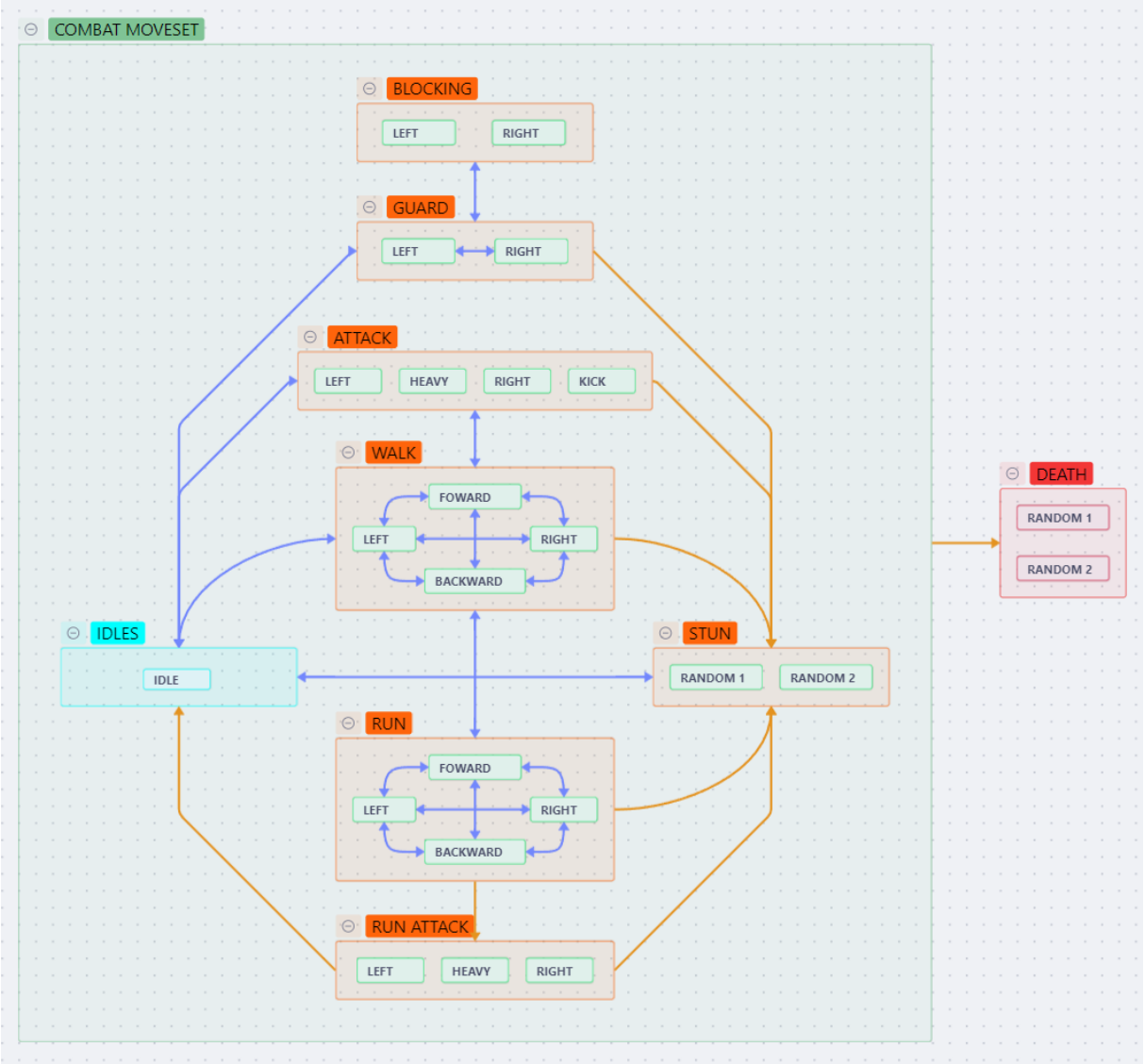
Pour les mouvements, je passe par mon moteur physique. Ce dernier a été créé par moi-même pour la démo, je n'utilise ainsi pas le moteur physique disponible dans le moteur, car ce dernier a été fait dans le cadre de mon projet TER.

Les contrôles du joueur utilisent l'accélération et non la vélocité afin de déplacer l'entité attribué à ce dernier. Cela a pour but de rendre plus réalistes les déplacements du joueur et donné plus de poids à ces derniers. Dans cette même optique, j'ai fait le choix d'utiliser des animations à la troisième personne pour toutes les entités, y compris le joueur. De ce fait, le joueur est capable de voir son corps et son équipement. La caméra suit l'os de la tête du modèle squelettique, ce qui permet d'avoir de façon native tous les effets de mouvements liés aux actions du joueur. Cependant, afin d'améliorer le confort de cette approche, la caméra suit uniquement la position de l'os, et non sa rotation.

Les entités non joueuses, quant à elles, utilisent la vélocité afin de simplifier le développement. Je projette dans le futur d'établir un seul et même système pour le déplacement réaliste de toutes les entités, mais je n'ai pas eu le temps car limité par mon moteur physique qui manque de fonctionnalités.

Système de combat et agent

Pour créer un système de combat, j'utilise le système de graphe d'animations de Vulpine. Ce dernier permet de définir une liste de transitions configurable, ainsi que des fonctions lambda de routine pour le début, la fin et le déroulement d'une animation. J'ai ainsi créé un système de création de moveset. Ce dernier charge une liste d'animations possédant un préfix donné en paramètre et implémente le graphe si dessous (note : ceci est une représentation simplifiée, les conditions de transition et les callbacks n'y sont pas mentionnés) :



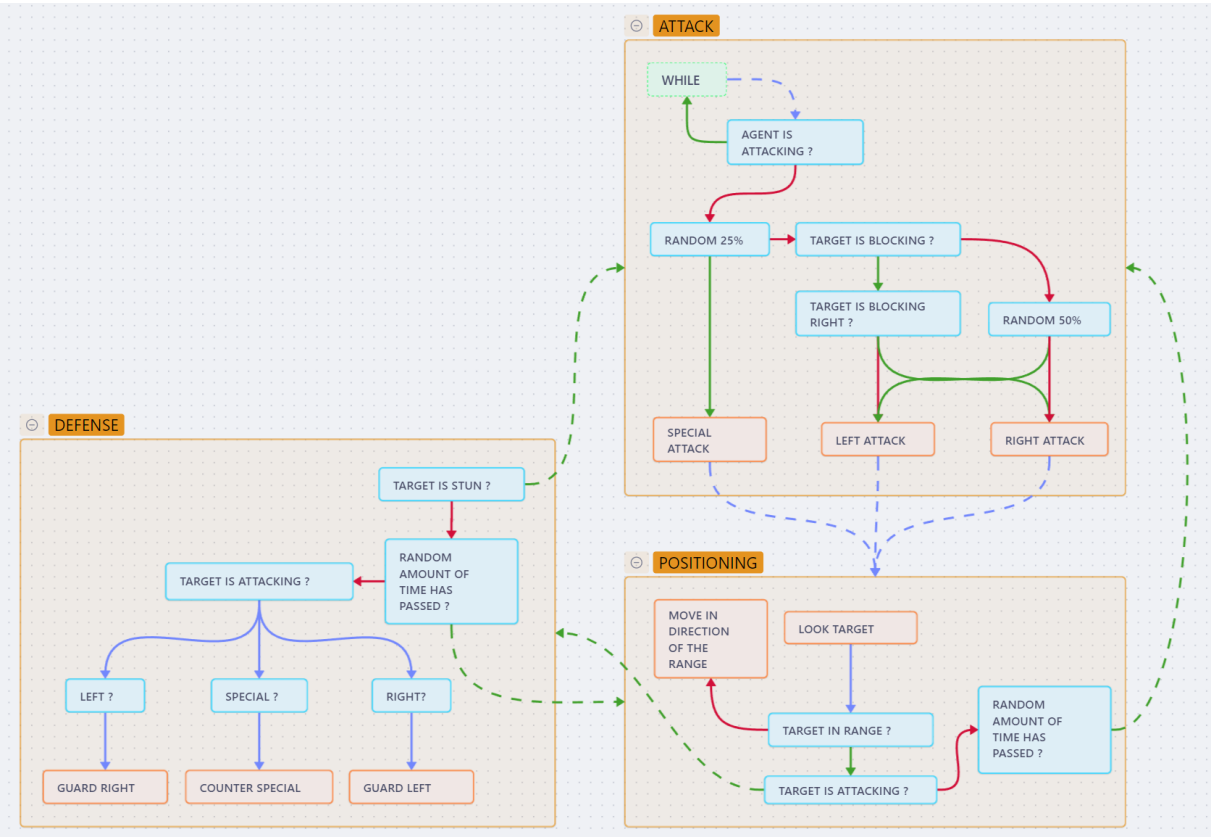
Notre système de combat contient ainsi un système de direction d'attaque : gauche, droite et « special ». Le spécial représente les actions lourdes rendant vulnérable l'attaquant. Afin qu'une attaque soit correctement bloquée, il faut que la direction de la garde corresponde à la direction de l'attaque. Les entités peuvent également être sonnée oo mourir à n'importe quelle étape.

Voici le code permettant d'équiper une entité d'une arme et de lui donner un moveset :

```
GG::playerEntity->comp<Items>().equipped[WEAPON_SLOT] = {24, Blueprint::Zweihander()};
GG::playerEntity->comp<Items>().equipped[LEFT_FOOT_SLOT] = {7, Blueprint::Foot()};

GG::playerEntity->set<AnimationControllerRef>(
    AnimBlueprint::bipedMoveset("65_2HSword", GG::playerEntity.get())
);
```

Pour finir, j'ai créé un agent simple de combat utilisant les aspects de ce système. Voici une représentation graphique de son comportement :



Voici le code permettant de faire apparaitre une entité de combat :

```
auto e = Blueprint::TestManequin();
e->set<DeplacementBehaviour>(FOLLOW_WANTED_DIR);
e->set<AgentState>({AgentState::COMBAT_POSITIONING});
e->set<Target>(Target{GG::playerEntity});
```

Le code en lien avec le graphe d'animation peut être trouvé dans src/AnimationBlueprint.cpp. Le code en lien avec le code de l'agent peut être trouvé dans src/game__physicsloop.cpp dans le système approprié.

Conclusion

La démo que j'ai rendue contient les bases d'une pré-alpha. L'ECS est énormément utilisé afin de rendre l'implémentation plus fluide. Je n'ai malheureusement pas eu le temps de faire l'intégralité de choses que j'aurais voulue. Cependant cette démo montre un système de combat complet axé animation ainsi qu'un outil de dialogue avancé.