Emma Vodable & Thomas Zorroché présentent

Projet de Synthèse d'Image, Programmation C++
et Architecture Logicielle

Introduction & Compilation

Oniris a été réalisé en binôme par Emma Vodable et Thomas Zorroché. Globalement, la répartition du travail s'est fait telle quelle :

- Thomas Zorroché : Programmation Graphic Engine
- Emma Vodable : Programmation Gameplay et UI

Nous vous invitons à regarder les commits du projet github pour voir plus en détail la répartition du travail.

Librairies et Logiciels

Nous avons développé ce jeu avec **C++ (14)** et **OpenGL 3.3**.
Voici la liste des autres librairies utilisées dans Oniris :

- **GLFW** : création d'un contexte OpenGL, définit une fenêtre et gère les entrées utilisateurs.
- **Glad** : donne accès à l'ensemble des fonctions OpenGL.
- **IrrKlang** (1.6) : gestion du son.
- **Assimp** : importe des modèles 3D

Pour développer Oniris, nous avons utilisé l'IDE **Visual Studio** sur **Windows**. Dans un autre registre, les modélisations 3D ont été réalisées sur **Blender**, les interfaces graphiques sur **Photoshop**. La gestion de projet à été réalisé sur **Notion**.

Compilation

Pour faire fonctionner le jeu sur **Linux** :

- Soyez sûr d'avoir g++ et cmake d'installer
- installez les packages nécessaires :

```
apt-get install libsoil-dev libglm-dev libassimp-dev libglfw3-dev
libxinerama-dev libxcursor-dev libxi-dev
```

- créez et installez vous dans un directory 'build/' puis lancez les commandes de make

```
mkdir build
cd build
cmake ..
make
```

- Vous pouvez lancer le programme à l'aide des flags custom de résolution (`-hd -fhd`) et de warning (`-none -info -warning -error`) tel que :

```
../bin/Oniris -fhd -none // window size in full HD and no warning
../bin/Oniris -hd -error // window size in HD and all warning
```

Et sur **Windows** :

- avoir cmake et Visual Studio d'installer sur sa machine
- cloner le repo
- dans le dossier du repo exécuter les commandes suivantes :

```
mkdir build
cd build
```

```
cmake "Visual Studio 16 2019" ..
```

Jouer

Objectifs du jeu et Univers d'Oniris

Ce jeu nous plonge sur une île mystérieuse à l'ambiance inquiétante. Au fil du jeu, le joueur découvre l'histoire de cet endroit et de l'Opération Oniris. L'objectif est l'exploration. Le naufragé observe sur sa route des bouts d'histoire pour l'aider à comprendre le lieu et ses habitants disparus. Il pourra ramasser quatre cristaux, le cœur de l'île, qui, au complet, permettent d'ouvrir un portail vers un autre monde.

**** **Attention Spoiler : À lire après avoir découvert un peu le jeu** ****
Une base militaire à été construite sur cette île pour découvrir la source d'une étrange créature qui apporte une obscurité destructrice partout où il passe. La base est bientôt abandonnée par manque de moyen et d'avancement. Et la créature reprend ces droits. L'opération Oniris, créée par un riche homme d'affaires, reprend les recherches. Les techniciens et scientifiques sont logés sur place avec leurs familles. Un jour, une erreur de manipulation est commise et la créature s'échappe, ramenant au néant tous les habitants de l'île.
Le joueur cherche à guérir l'île de sa malédiction. Il récolte les fragments de cristaux pour les remettre dans l'autre monde ouvert dans le portail, apaisant et détruisant la créature des ténèbres de notre réalité.
**** **Fin Spoiler** ****

Programmation gameplay

Commandes du jeu :

- Déplacement : **Z, Q, S, D**
- Interagir : **E**
- Afficher la carte : **A**
- Courir : **Shift**
- Afficher les boîtes de collision : **C** *(Debug)*

La commande Interagir comprend les actions suivantes :

- **Observer** des objets ramassés
- **Ranger** des objets dans un inventaire
- **Collecter** des cristaux
- **Interagir** avec des objets (ouvrir une porte, activer le courant)

Ces inputs sont traités dans la méthode `InputHandler::ProcessInput()` ou `Object::OnOverlap()`. Ces méthodes mettent à jour l'état de **Hud** et de **Game**.
En effet à chaque frame, l'appel de `Hud::Update()` remet à jour l'interface utilisateur.

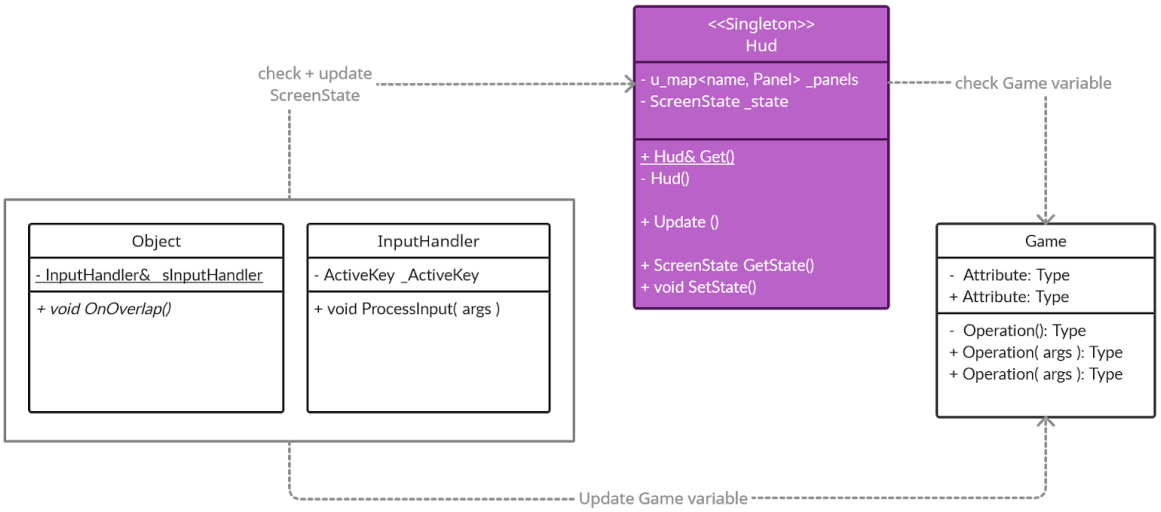
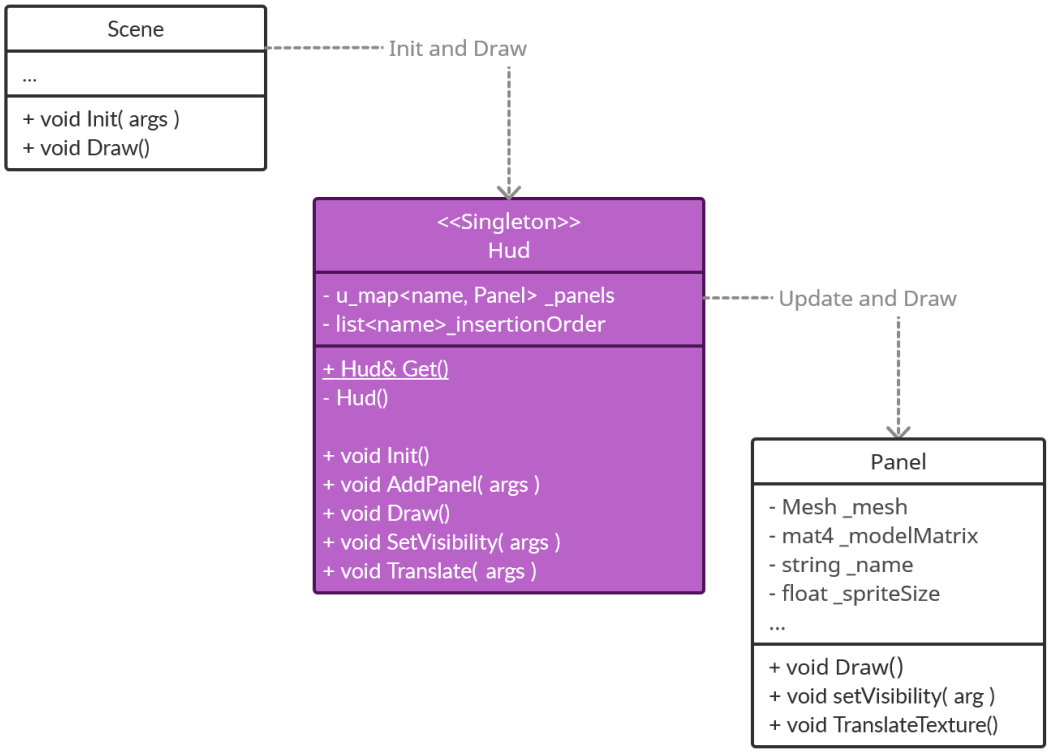


Diagramme de Classe des objets Gameplay

Interface Utilisateur

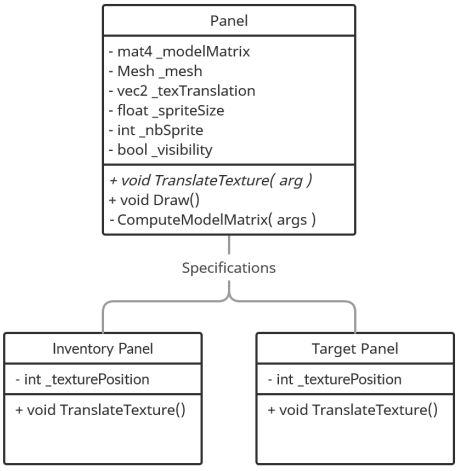
Hud

L'interface est gérée par le **singleton Hud** qui contient tous les panels. Les panels sont des plans placés sur le "near plane" de la caméra. Cela revient à créer mesh (avec x et y compris entre -1 et 1) auquel on n'applique pas de matrice de projection ni de view matrice.
Pour animer ces interfaces, nous avons utilisé des sprites. Nous les manipulons en translatant les textures coordinates du panel à l'aide d'un uniforme (c'est le vec2 `_texTranslation` dans **Panel**).



Panel

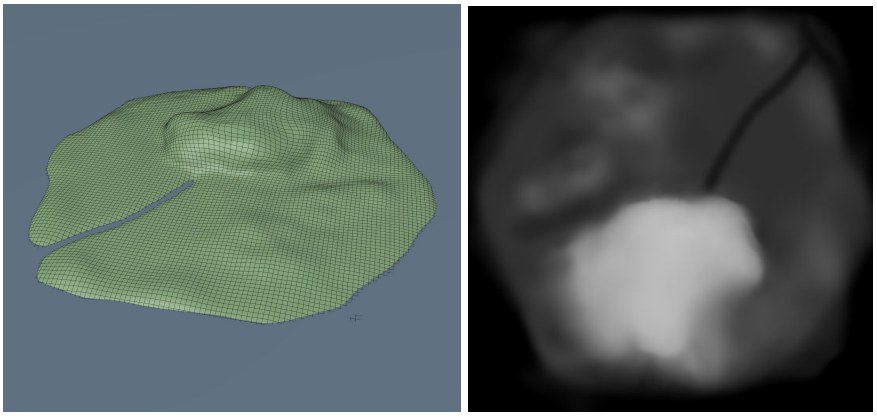
Nous utilisons un **héritage par spécification** afin que les panels puissent avoir des comportements différents pour une même input (les **InventoryPanel** changent lorsqu'on ramasse un panel ou lorsqu'on scroll, et le **Target Panel,** lorsqu'on est en contact d'une collision Box). Ici la fonction `Panel::TranslateTexture()` est override par les filles.



Le monde

Terrain

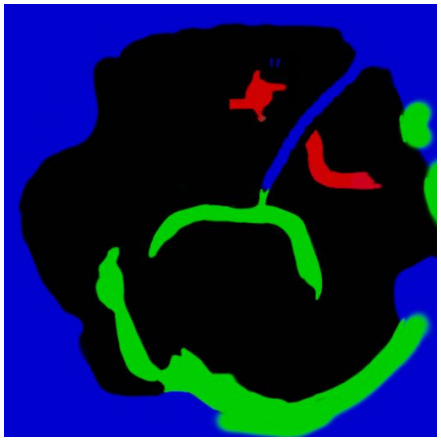
Nous utilisons une **heightmap** pour générer le terrain. Nous avons dans un premier temps sculpté le terrain dans Blender, puis on a exporté une heightmap du mesh.



Nous avons rencontré de nombreux problèmes concernant la génération du terrain, notamment le fait qu'il ne soit pas vraiment "lisse". Pour palier à ce défaut, nous avons opté pour deux solutions :

- Exporter la heightmap en **Grayscale 16-bit**.
- Faire une **interpolation linéaire** sur la coordonnée Y (hauteur) de la caméra.

Multi - Texturing



Afin d'appliquer plusieurs textures à notre terrain, nous avons importer plusieurs images que l'on mélange dans le fragment shaders à l'aide d'une **blendmap**. Ainsi, on utilise le canal rouge pour les pavées, le vert pour les rochers, le bleu pour le sable, ainsi que le noir pour l'herbe.

Océan

L'océan est constitué d'un unique plan que l'on agrandit de 120% de la taille du terrain. Dans le **fragment shader**, nous avons utilisé un **noise 2D** pour animer les textures coordinates du plan et obtenir un aspect de mouvement sur l'écume. Le résultat est parfois un peu étrange, nous avons juste expérimenté avec des fonctions de noise très basiques, c'est un aspect qui pourrait être amélioré.

Brouillard

Le brouillard a nécessité plusieurs étapes avant d'arriver dans son état final :

- Dans un premier temps, on a appliqué un fog **selon la distance** entre le fragment et la caméra (utilisation d'une exponentielle pour lisser le rendu)
- Ensuite, on a décidé d'appliquer un brouillard d'une couleur différente selon **l'orientation de la caméra** avec la direction du soleil.
- Le brouillard est moins **dense** selon l'altitude
- On a ensuite ajouté deux variables (**extinction** et **inscaterring**) pour faire varier plus précisément la transition entre les deux couleurs de brouillard.
- Enfin on entoure le joueur d'un brouillard épais bleu qui suit son déplacement.

Entity Importer

Nous avons utilisé des fichiers .txt afin d'importer les entités graphiques dans notre code. Ces fichiers sont rangés dans Oniris/res/scene. Il y a trois types de fichier :

Particule System

[name] Trees
[obj] res/path/model.obj
[alpha] 0
[count] 150
[size] 1.8 0.5
[cp] 136 608 1 40
[cp] 200 400 1 40
[end]
. . .

Objects

[name] carnet
[type] N
[obj] res/path/model.obj
[tex] res/path/panel/png
[pos] 100 400
[end]
. . .

Static Meshes

[obj] res/path/model.obj
[loc] 594 450
[rot] 0 270 0
[scl] 1.5
[end]
. . .

<u>Détails des paramètres :</u>

- **name** : nom du système de particules
- **obj** : chemin du modèle 3D
- **alpha** : si la texture utilisée par le modèle contient un canal alpha (billboard)
- **count** : nombre d'instances
- **size** : taille d'une instance = 1er nombre +/- 2ème nombre
- **cp** : point de contrôle
    - x position
    - z position

    - densité (de 1 à 3)
    - rayon
- **type** : type d'object
- **tex** : texture du panel pour les objets
- **pos** : position (x,y) sur l'écran pour les panels
- **loc** : position (x,z) sur le terrain pour les static meshes
- **rot** : rotation du static mesh
- **scl** : échelle du static mesh

L'importation de ces trois fichiers est gérée par la classe **EntityImporter**.

Modèles 3D

Nous avons modélisé ces modèles 3D sous **Blender**.


Les objets

Principe

La classe Object représente les meshs qui ont besoin de traiter les événements de collisions avec le joueur. Dans l'ensemble ce sont les objets de gameplay qui sont :

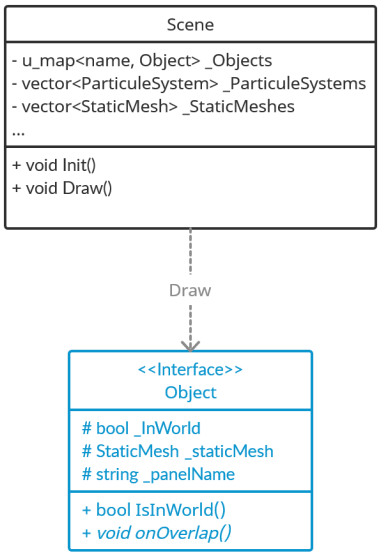
- **Narrative :** Lorsque le joueur trouve et observe l'objet (en appuyant sur E), un menu s'ouvre et décrit l'objet. Les objets narratifs restent à leur place. Ils sont optionnels au jeu. Il y en a 13.
- **Usable :** Ces objets sont ceux que l'on peut ajouter à notre inventaire: Soit la clef, la carte et les cristaux. A proximité de ces objets, le joueur peut les ramasser et ils s'ajoutent à son inventaire sur le HUD.
  La clef est l'ancêtre d'un système d'énigme que nous n'avons pas eu le temps d'implémenter.
  Le joueur peut ouvrir la carte à partir du moment où il l'a ramassée et qu'il appuie sur A.
  Lorsque le joueur réunit les quatre cristaux répartis aléatoirement dans le monde, le portail qui surplombe l'île s'ouvre afin de lui permettre l'accès à l'autre monde.

- **Interactive** : Ces objets comprennent les portes (IODoor) et l'interrupteur (IOLight). Ils ne peuvent pas être ni ramassés ni observés. Cependant, manipuler l'objet produit un résultat dans le monde, soit allumer les lumières, soit ouvrir une porte.

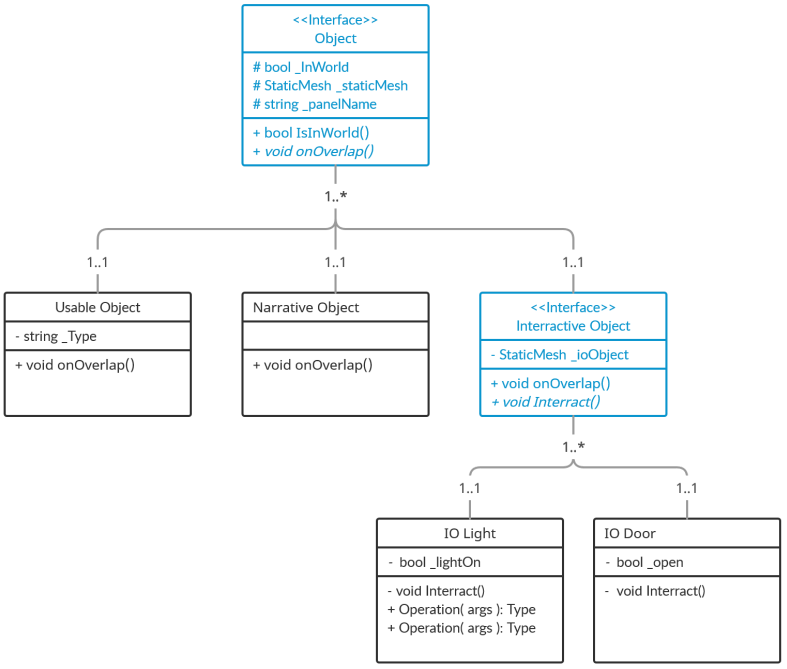
On peut facilement ajouter des objets dans le fichier 'res/scene/objects.txt'.

POO

Pour ces classes, nous avons implémenté un système d'héritage. Ce design pattern a pour avantage de déclarer sous le même nom des objets avec des comportements différents. Ainsi on peut stocker et dessiner dans la classe Scene indépendamment tous types d'objets.



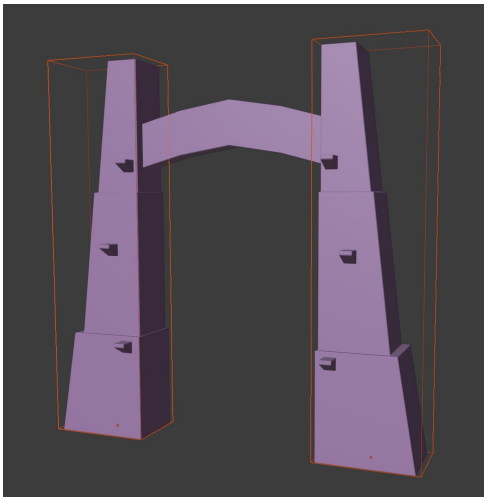
Les objets sont créés avec tous leurs attributs dans la méthode `EntityManager::Objects()` à partir d'un fichier texte.



La classe **CboxPortal**, héritant de Object, a été ajoutée dernièrement. C'est une classe particulière qui fait fonctionner la traversée du portail. C'est un objet dans le sens ou il doit traiter les événements de collisions.

Collisions

Dans cette partie, cBox = boîte de collision



Les cBoxes sont principalement utilisés pour :

- **Arrêter le mouvement** du joueur et ainsi éviter de traverser des objets.
- **Déclencher des événements** lors du passage du joueur dans une zone.

Nous avons opté pour le modèle de collision **AABB** (axis aligned bounding box). Ainsi chaque mesh est modélisé avec une cBox

dans Blender. Toutes les cBoxes doivent alors être nommées “name_of_mesh_cBox” .
Lors de la création d’une entitée, la class **Model** reconnaît alors dans le nom de l’objet “cBox” et crée une **CollisionBox** associée à un **StaticMesh**.

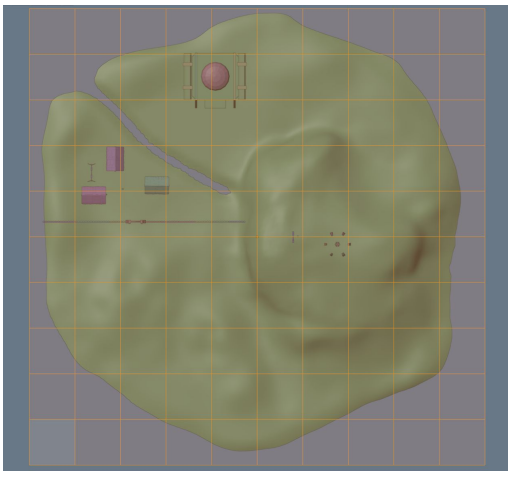
Gestion des événements lors des collisions

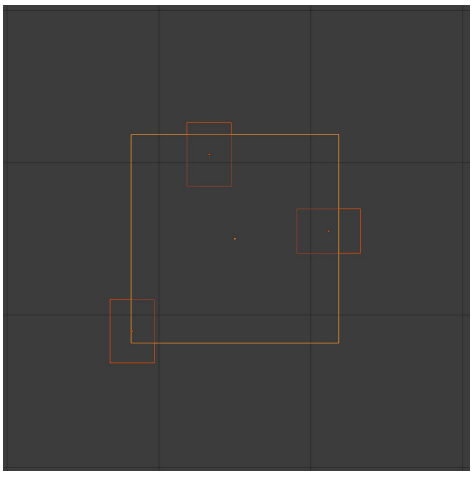
Nous avons ici essayé d'implémenter un design pattern **Observer**. Lorsque la cBox de la caméra entre en contact avec une autre cBox, celle-ci appelle la méthode virtuelle pure Object::OnOverlap() de l’objet associé à cette cBox. Cette fonctionnalité nous à été inspirée par le fonctionnement des collisions de Unreal Engine.
CollisionLayout définit le comportement lors d’une collision :

- si la cBox stoppe le mouvement
- si elle peut changer durant le jeu (rotation, déplacement etc…)
- et si elle est liée à un objet

Grille

Afin de gagner en performance, nous avons décidé de créer une grille sur notre carte, et d’associer chaque cBox à une case de la grille. De ce fait, on teste alors chaque collisions entre le joueur et les objets qui se trouvent dans la même case que le joueur.
Pour





fluidifier le nombre de cBox “actives” (celles testées pour une frame), on autorise une case à accepter les cBox dépassant légèrement ses frontières, afin que le passage d’une case à une autre soit plus lisse.

Gestion de la Rotation

Le modèle AABB a certaines limites, dont celle de ne pas pouvoir appliquer de rotation (les axes doivent être alignés). On autorise alors dans notre jeu seulement les rotations de 90°C sur les objets ayant une cBox. Toutes les autres transformations sont autorisées.

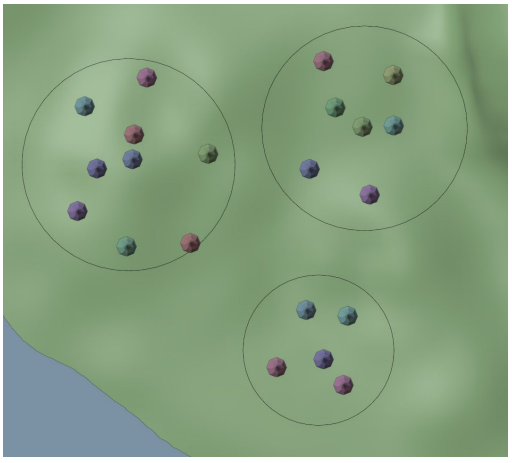
Système de Particules

Instancing

Les systèmes de particules permettent de donner de la vie à notre monde en lui fournissant des milliers de mesh. Pour cela, on a utilisé les fonctions d'**instance** d'opengl, comme la fonction de tracé suivante :

```
glDrawElementsInstanced(GL_TRIANGLES, _indices.size(), GL_UNSIGNED_INT, 0,
countParticules);
```

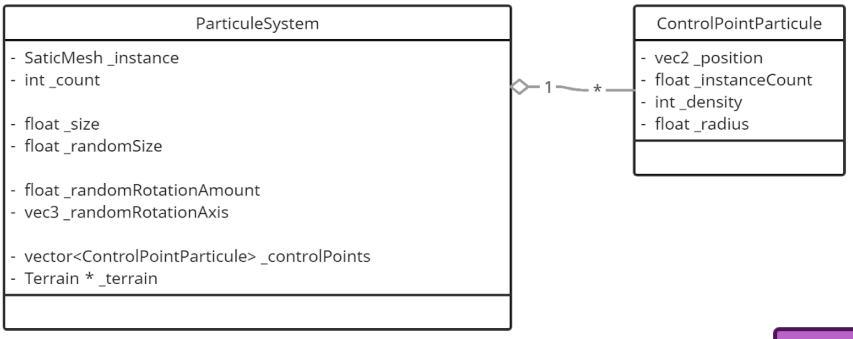
Point de contrôle



Grâce à la classe **ControlPointParticule**, on peut émettre des objets autour d'un point, avec un nombre d'instances et un rayon variables. Ensuite, on fournit une instance de **ControlPointParticule** à **ParticuleSystem** qui se charge d'émettre chaque objet de façon aléatoire dans l'espace restreint dicté par les points de contrôle.
La génération d'objets est alors faite de façon aléatoire sur les tailles des particules ainsi que leur orientation.

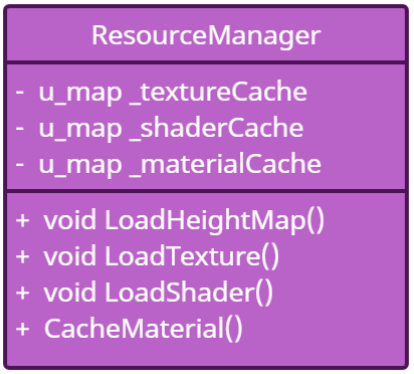
Gestion de l'aléatoire

Afin d'avoir des coordonnées et des valeurs d'échelles aléatoires sur les particules, nous avons utilisé les classes de la STL, notamment `std::default_random_engine` pour le générateur. On a également utilisé une **distribution normale** pour les coordonnées des particules, ainsi qu'une **distribution uniforme** pour les échelles.



Gestion des Ressources

La gestion des ressources est gérée par un **Singleton** : **ResourceManager**. Il permet d'importer des textures, de stocker des shaders et des matériaux. Implémenté en cours de projets, il nous a permis d'éviter certaines fuites de mémoire dues au total contrôle des ressources passant par une unique instance de classe.



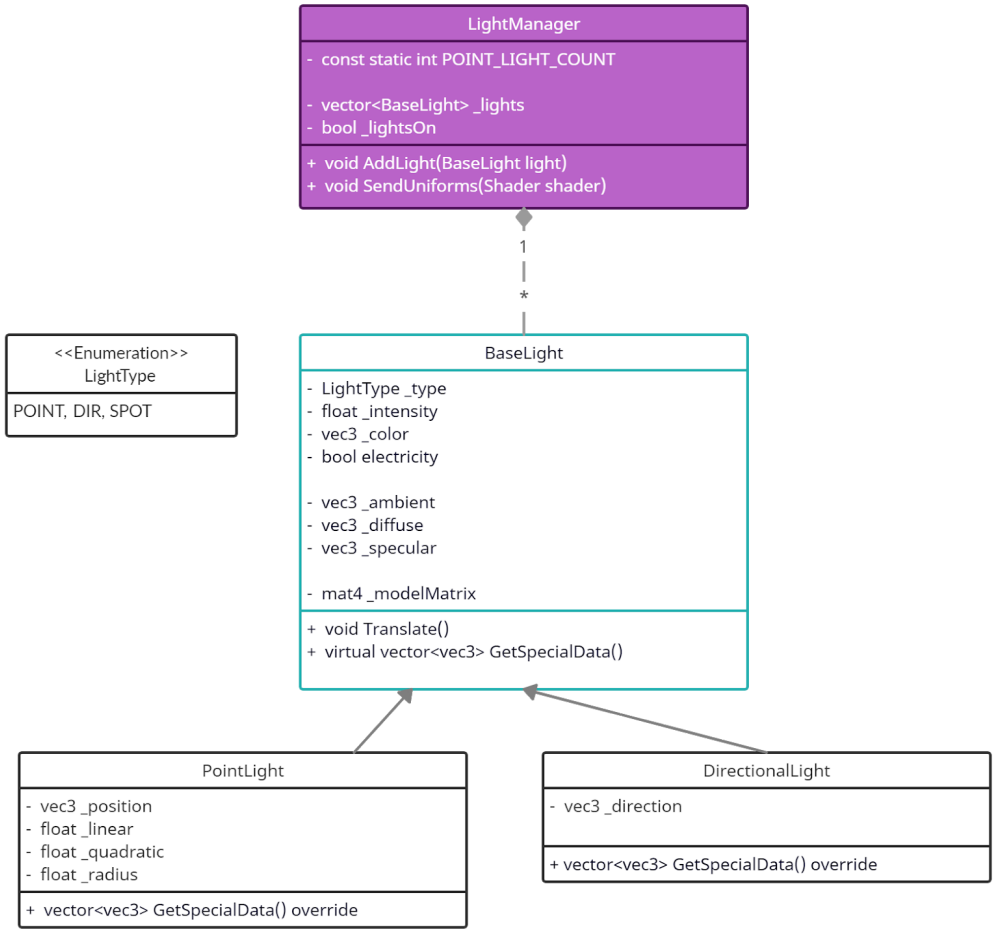
Il permet aussi d'éviter d'importer plusieurs fois des ressources grâce à un système de cache (unordered_map).

Gestion des Lumières

Les lumières sont gérées par un **Singleton** : **LightManager**. Nous disposons de deux types de lumières : lumière ponctuelle et lumière directionnelle. Toutes deux dérivent d'une classe mère abstraite : **BaseLight**. On utilise alors le polymorphisme pour appeler les redéfinitions des méthodes des classes filles afin de pouvoir gérer toutes types de lumières.

Afin d'envoyer les uniforms aux shaders, on boucle sur toutes les lumières afin d'envoyer les valeurs aux structures des shaders.

Dans les fragments shaders, on additionne alors les lumières, puis on multiplie à la couleur que nous donne le modèle Blinn-Phong.



C++ Features

Technique	Fichier/Ligne	Description
Classe Template + surcharge d'opérateur	*collision/CollisionGrid.hpp* ligne 25	Nous utilisons une unordered_map avec comme clé une classe custom : CollisionGridCase. On a donc dû **surcharger** la méthode **operator()** de la class **hash** de la std, puis créer une fonction de comparaison **operator==()** pour que le compilateur puisse comparer deux CollisionGridCase.
Fonction Template	*common.hpp* ligne 8	**template<typename T> T Lerp(T stat, T end, T t);** Permet de faire une interpolation linéaire pour chaque type ayant accès aux opérateurs + et *
Héritage	*Hud/Panel.hpp*	cf Panel (p.5)
Polymorphisme	*lighting/BaseLight.hpp*	cf Gestion des Lumières (p.10)
Polymorphisme	*gameplay/Object.hpp*	cf Les objets (p.7)
STL : structure de données	*engine/Scene.hpp* ligne 39:41 *hud/Hud.hpp* ligne 49:50	**… std::unordered_map<std::string, std::shared_ptr<Object>>** : Structures de données (unordered_map + vector + list) sauvegardant des shared-ptr (ou autres). Dans Hud.hpp, **std::list<std::string> _insertionOrder** permet de sauvegarder l'ordre d'ajout des Panels et ainsi les dessinés dans le bon ordre, évitant des erreurs d'affichages.
STL : aléatoire	*engine/ParticuleSystem.cpp* ligne 26	**std::default_random_engine** Gestion de l'aléatoire dans la génération de particules : position **std::normal_distribution** échelles **std::uniform_real_distribution**
STL : pointeurs dynamiques	*engine/Scene.hpp* ligne 42:46	Nous utilisons souvent des pointeurs intelligents partagés (**shared_ptr**) afin d'avoir une gestion automatique de la mémoire sur des objets dont les ressources ont besoin d'être partagées par plusieurs objets.
STL : Iterator	*collision/CollisionManager.cpp* ligne 101	Boucle for à l'aide d'un **itérateur** sur une **unordered_map** avec comme clé CollisionGridCase.
Exception	*lighting/PointLight.cpp* ligne 18	**throw std::string("Wrong radius size …");** Les lumières directionnelles doivent avoir des rayons prédéfinis. Si ce n'est pas le cas, on avertit l'utilisateur et on met des valeurs par défaut.
Exception	*engine/Material.cpp* ligne 67	**try catch** Toutes les importations des textures sont comprises dans des structures try, catch. Ainsi, si l'utilisateur donne un mauvais chemin, on lui envoie un message d'erreur mais le programme continue.
Assert	*engine/Terrain.cpp* ligne 186	**assert("Math error: Attempted to divide by Zero")** Lorsqu'on génère le terrain, on ne peut pas avoir une case plus petite que 1 car cela engendre une division par 0;

Références

Nos principales **inspirations artistiques** pour Oniris ont été :

- Among Trees par FJRD interactive : C'est un jeu indépendant de survie et de craft dans une forêt. Nous avons repris le minimalisme de son Hud et l'ambiance brumeuse et colorée.
- Lost (série TV): Cette série nous a inspiré pour la narration.
- Divers jeux en low poly ont inspiré les assets graphiques, tel que Zelda : Wind Waker pour l'eau...

Nos **références techniques** sont :

- Les playlists OpenGL et C++ de The Cherno
- La playlist 3D Game in Java de Thin Matrix
- learnopengl

Conclusion

Emma

Je suis très fière de ce projet. Nous avons été très vite passionné par le projet et donc nous avons eu pleins d'idées qui ne sont pas forcément présentes au terme du projet. Il atteint mes attentes techniques, artistiques et narratives. Je pense avoir appris beaucoup de choses sur la répartition des tâches dans un projet. En effet, même si je n'ai pas coder le système de rendu 3D, il a fallu que je comprenne et utilise le code de Thomas, ce qui nous a demandé de communiquer. Cependant, je pense que j'aurais pu être plus appliqué lors de la réflexion du système de gameplay sur le papier. Bien que nous ayons dessiné des schéma UML au prémisse du projet, il a été difficile de les mettre en œuvre et cela nous a coûté l'ajout de fonctionnalités de dernière minute. Globalement, je pense avoir amélioré mes habitudes de développement, surtout grâce aux notes de Thomas sur les conventions de nommages et autres joyeusetés. J'espère que vous avez aimé jouer à Oniris.

Thomas

Pour ce projet, nous sommes partis avec d'assez grandes ambitions. Finalement, très peu ont pu être réalisés, cependant je reste tout de même très fier du résultat obtenu. Oniris a été mon premier jeu 3D codé en dehors d'un moteur de jeu grand public, et on se rend vite compte de la quantité de concepts à coder pour avoir des aspects de bases attendus dans un jeu vidéo (collision, hud, particules ...). C'était la première fois que je traitais des sujets comme l'instancing, les collisions, la gestion d'un brouillard. J'ai beaucoup aimé me plonger dans de nombreux articles et thèses de synthèses d'images passionnantes, parfois même en oubliant le projet... Du côté du C++, j'ai pu acquérir de nombreux réflexes notamment sur l'utilisation de classes ou encore sur les outils de la STL (map, itérator, random engine, ...). Au final, je pense que mon seul regret réside sur la structure globale du code. Nous avons démarré très vite dans le code, certes avec des schémas UML pour nous guider, mais l'abstraction d'une librairie comme OpenGL n'a pas été si simple. Globalement, c'est l'abstraction à la fois d'un jeu vidéo (gameplay, hud, joueur) et d'un moteur de jeu (gestion des textures, lumières, shaders) qui m'a vraiment donné beaucoup de mal. Je pense que si je devais refaire le projet, je m'y prendrais d'une façon complètement différente, et cela me motive d'ailleurs à le refaire le plus vite possible. Pour conclure, je pense que ma plus grande fierté est l'ambiance qu'on a réussi à donner au jeu. Lorsqu'on a fait tester le jeu à plusieurs personnes, l'esthétique et le

style sont toujours ressortis comme points forts. Je pense que sur cet aspect du jeu contemplatif, nous avons réussi cette mission.