

ECOLE POLYTECHNIQUE DE L'UNIVERSITÉ FRANÇOIS RABELAIS DE TOURS

Département Informatique

64 avenue Jean Portalis

37200 Tours, France

Tél. +33 (0)2 47 36 14 14

polytech.univ-tours.fr

Projet Recherche & Développement

2018-2019

Réalisation d'un jeu vidéo : Don't get caught !

Entreprise

Ubisoft

Tuteur entreprise

Romain CORREIA

Étudiant

Alexandre DHALENNE (DI5)

Tuteur académique

Mathieu DELALANDRE



Liste des intervenants

Entreprise

Ubisoft

Montreuil

Nom	Email	Qualité
Alexandre DHALENNE	alexandre.dhalenne@etu.univ-tours.fr	Étudiant DI5
Mathieu DELALANDRE	mathieu.delalandre@univ-tours.fr	Tuteur académique, Département Informatique
Romain CORREIA	romain.correia@ubisoft.com	Tuteur entreprise



Avertissement

Ce document a été rédigé par Alexandre DHALENNE susnommé l'auteur.

L'entreprise Ubisoft est représentée par Romain CORREIA susnommé le tuteur entreprise.

L'Ecole Polytechnique de l'Université François Rabelais de Tours est représentée par Mathieu DELALANDRE susnommé le tuteur académique.

Par l'utilisation de ce modèle de document, l'ensemble des intervenants du projet acceptent les conditions définies ci-après.

L'auteur reconnaît assumer l'entière responsabilité du contenu du document ainsi que toutes suites judiciaires qui pourraient en découler du fait du non respect des lois ou des droits d'auteur.

L'auteur atteste que les propos du document sont sincères et assument l'entière responsabilité de la véracité des propos.

L'auteur atteste ne pas s'approprier le travail d'autrui et que le document ne contient aucun plagiat.

L'auteur atteste que le document ne contient aucun propos diffamatoire ou condamnable devant la loi.

L'auteur reconnaît qu'il ne peut diffuser ce document en partie ou en intégralité sous quelque forme que ce soit sans l'accord préalable du tuteur académique et de l'entreprise.

L'auteur autorise l'école polytechnique de l'université François Rabelais de Tours à diffuser tout ou partie de ce document, sous quelque forme que ce soit, y compris après transformation en citant la source. Cette diffusion devra se faire gracieusement et être accompagnée du présent avertissement.



Pour citer ce document

Alexandre DHALENNE, *Réalisation d'un jeu vidéo : Don't get caught !*, Projet Recherche & Développement, Ecole Polytechnique de l'Université François Rabelais de Tours, Tours, France, 2018-2019.

```
@mastersthesis{
  author={DHALENNE, Alexandre},
  title={Réalisation d'un jeu vidéo : Don't get caught !},
  type={Projet Recherche \& Développement},
  school={Ecole Polytechnique de l'Université François Rabelais de Tours},
  address={Tours, France},
  year={2018-2019}
}
```

Table des matières

Liste des intervenants	a
Avertissement	b
Pour citer ce document	c
Table des matières	i
Table des figures	v
1 Introduction	1
1 Acteurs, enjeux et contexte	1
2 Objectifs	2
2.1 Etude d'un moteur de jeu : Unity	2
2.2 Etude d'un Design Pattern : Entity Component System	2
2.3 Réalisation d'un jeu vidéo	3
3 Base méthodologiques	3
2 Description générale	5
1 Environnement du projet	5
2 Caractéristiques des utilisateurs	5
3 Fonctionnalités du système	6
4 Contraintes de Développements, d'exploitation, de maintenance	7
4.1 Contraintes de développements	7
4.2 Contraintes d'exploitation	7
4.3 Maintenance et évolution du système	7

3	Etat de l'art / veille technologique	8
1	Unity	8
1.1	Fonctionnement de Unity	9
1.1.1	La scène	9
1.1.2	La fenêtre de jeu	9
1.1.3	L'inspecteur	10
1.2	GameObject, Prefab, Scripts, Axes	10
1.2.1	Un GameObject	10
1.2.2	Un Prefab	10
1.2.3	Les scripts	11
1.2.4	Les Axes	11
1.2.5	UNet : Le framework Réseau de Unity	12
2	Entitas	13
2.1	Les contextes	13
2.2	Les composants	13
2.3	Les entités	14
2.4	Les systèmes	15
4	Analyse et conception	18
1	Éléments principaux de l'analyse	18
1.1	Liste des fonctionnalités à implémenter	18
1.2	Decoupage du projet selon Entitas	19
1.2.1	Les Composants	19
1.2.2	Les Entités	19
1.2.3	Les systèmes	20
1.3	Autres scripts	20
1.4	UML	21
2	Sprints	21
2.1	Début du projet	21
2.2	Sprint 01	22
2.3	Sprint 02	22
2.4	Sprint 03	23
5	Mise en oeuvre	24
1	Outils et bibliothèques	24
1.1	Unity	24
1.2	Visual Studio 2017	24
1.2.1	Entitas	24
2	Implémentation	24
2.1	Les composants	24

2.1.1	Player	24
2.1.2	GameComponents	25
2.1.3	InputComponents.....	25
2.2	Les entités	25
2.2.1	Le voleur	25
2.2.2	Le Gardien	25
2.2.3	La capsule	26
2.2.4	L'InputManager	26
2.3	Les systèmes.....	26
2.3.1	InputSystem.....	26
2.3.2	SelectedSystem	26
2.3.3	GroundedSystem	26
2.3.4	GuardianMoveSystem	26
2.3.5	ThiefMoveSystem	26
2.3.6	ThiefUseSystem	27
2.4	Les prefabs	27
2.4.1	Le voleur	27
2.4.2	Le Gardien	27
2.4.3	La capsule	27
2.4.4	Le LobbyManager	27
6	Bilan et conclusion	28
1	Ce qui est fait	28
2	Retards et reste à faire	28
3	Bilan qualité.....	28
4	Capitalisation.....	29
Annexes		30
1	Manuel d'installation	32
1.1	Pour Windows.....	32
1.2	Pour Android	32
2	Manuel Utilisateur.....	32
2.1	L'hôte Le voleur	32
2.2	Le client Le Gardien	33
3	Manuel Développeur	33
3.1	Logiciel et Framework nécessaire.....	33
3.2	Architecture des fichiers Unity avec Entitas.....	34
3.3	Développer avec Unity / Entitas	34
4	Tests	35

4.1	Exemple de scénario : le déplacement Voleur.....	35
4.2	Exemple de scénario : Gardien : mettre le piège Cage.....	35
4.3	Liste des fonctionnalités testées	36
4.3.1	Pour le Voleur	36
4.3.2	Pour le Gardien.....	36

Table des figures

1 Introduction

1	Logo Unity	2
2	Logo Unreal Engine	3

2 Description générale

1	Logo PEGI 12	5
---	--------------------	---

3 Etat de l'art / veille technologique

1	Unity Logo	8
2	Image de l'éditeur Unity.....	9
3	Exemple de GameObject	10
4	Description d'un Axe	11
5	Schéma du Peer To Peer	12
6	Logo Entitas	13
7	Classe en CSharp d'un composant.....	14
8	Entites	14
9	Un GameController en CSharp	15
10	Architecture d'un projet Entitas	16

4 Analyse et conception

1	Tableau des fonctionnalités	18
2	Exemple de GameController simple	21

Bilan et conclusion

1	Diagramme de classes	31
2	Le Voleur	33
3	Vue du Gardien.....	33
4	Architecture	34
5	Trap UI	35

1

Introduction

1 Acteurs, enjeux et contexte

Le sujet de mon projet de Recherche et Développement consiste à la réalisation d'un jeu vidéo.

La réalisation d'un jeu vidéo suit une démarche un peu différente des projets informatiques classiques, même si les grandes lignes sont les mêmes. Le sujet que j'ai proposé était la réalisation d'un jeu vidéo en utilisant le moteur Unity et le framework Entitas. J'ai été mis en relation avec Romain CORREIA, programmeur Gameplay chez Ubisoft Paris afin qu'il m'encadre lors de la réalisation du projet.

Afin de mieux définir le sujet, 3 contraintes m'ont été imposées : Multijoueur asymétrique, Survie, et IA Systémique. J'ai dû imaginer 3 jeux respectant chacun une contrainte, puis nous avons choisi celui qui semblait le plus intéressant et réalisable durant l'année : l'idée du jeu multijoueur asymétrique.

Un jeu multijoueur asymétrique est un jeu qui doit être jouable à plusieurs, et où il y a au minimum 2 types de joueurs différents, qui ont une caméra / un contrôle / un personnage différent. Par exemple, imaginons Mario Kart avec du multijoueur asymétrique. Il y aurait les joueurs qui jouent de manière classique, et d'autres joueurs qui verraient la carte du dessus et pourraient rajouter des pièges, des bonus, des murs...

Une fois le jeu défini, il faut définir les différentes technologies que je vais utiliser afin de mener à bien le projet. J'ai proposé d'utiliser Unity, un moteur de jeu, avec le langage de programmation C#, et le framework Entitas, qui est un framework implémentant le design pattern Entité-Composant-Système pour Unity en C#.

2 Objectifs

2.1 Etude d'un moteur de jeu : Unity

Le développement d'un jeu vidéo comporte de multiples facettes. La base de tous les jeux est son moteur. Le moteur de jeu, c'est celui qui va gérer tout ce qui concerne la physique (gravité, collisions), mais aussi le rendu graphique, le son, les entrées utilisateurs, le réseau... Le moteur, c'est tous les composants logiciels qui permettent l'évolution et l'affichage de notre jeu.

Le développeur peut choisir de développer son propre moteur pour son jeu, ou utiliser un moteur de jeu existant. Il existe plusieurs moteurs de jeu, plus ou moins populaires. En voici quelques-uns :

1. Unreal Engine : Moteur de jeu développé par Epic Games. Orienté 3D. Langages : C++, Assembleur. Multiplateformes. Ce moteur de jeu a été utilisé dans la réalisation de beaucoup de jeux différents, comme Fortnite.
2. Unity : Moteur de jeu développé par Unity Technologies. Langages : C#. Multiplateformes. Ce moteur est populaire chez les développeurs indépendants, mais aussi chez les grands du jeu vidéo. Il a notamment été utilisé par Blizzard afin de réaliser Hearthstone.
3. Source 2 : Moteur de jeu développé par Valve. Langages : C++. Mutlplateformes. Ce moteur, développé par un grand du jeu vidéo, est utilisé par beaucoup de jeux populaires (Portal, CounterStrike : Global Offensive...).

Dans le cadre de mon projet, mon choix s'est porté sur Unity. Sa popularité et le multiplateformes très bien pris en charge par ce moteur sont deux points positifs qui ont porté mon choix vers ce moteur.



Figure 1 – Logo Unity

2.2 Etude d'un Design Pattern : Entity Component System

Afin d'architecturer le code de son jeu, il est nécessaire de respecter un certain design pattern. Cependant, il existe différentes manières d'organiser son code.

Pour ce projet, j'ai décidé d'utiliser le Design Pattern Entity Component System (Entités Composants Systèmes). Contrairement à d'autres qui sont basés sur la programmation orientée objets, ECS est basé lui sur la programmation orientée données. Ce sont les données qui vont influencer sur la logique du programme et non l'inverse.

Les données sont représentées par des composants. Un composant, il faut le voir comme une "pièce", assemblée à d'autres composants, vont former une Entité. Et l'Entité aura un comportement défini par ses différents composants. Ce qui va agir sur les composants, ce sont les systèmes. Les systèmes vont appliquer une logique aux composants, qui, peu importe l'entité et les autres composants qui la composent, agiront de la même manière.

Ce design pattern apporte deux gros avantages : il évite la répétition de code et permet de faciliter l'ajout de fonctionnalités.

Comme dit précédemment, les systèmes agissent sur les composants, non les entités. Imaginons le composant "Roue", et que nous créons deux entités "Vélo" et "Voiture". Ces deux entités auront certains composants, dont le composant "Roue". Et le système qui est en charge des composants "Roue" est unique. Les composants auront des données différentes selon l'entité, mais seront quand même des roues et la logique reste la même peu importe la "Roue".

Aussi, si nous souhaitons ajouter des fonctionnalités, il suffit de rajouter un composant et le(s) système(s) qui lui correspond. Je définis le composant et ses systèmes, et je n'ai plus qu'à ajouter ce composant aux entités qui vont bénéficier de cette nouvelle fonctionnalités.

Afin d'implémenter ce Design Pattern dans mon projet, j'ai utilisé le Framework Entitas C# pour Unity. Ce framework permet de simplifier la mise en place d'un projet respectant cette architecture et fournis des outils très intéressants lors du développement.

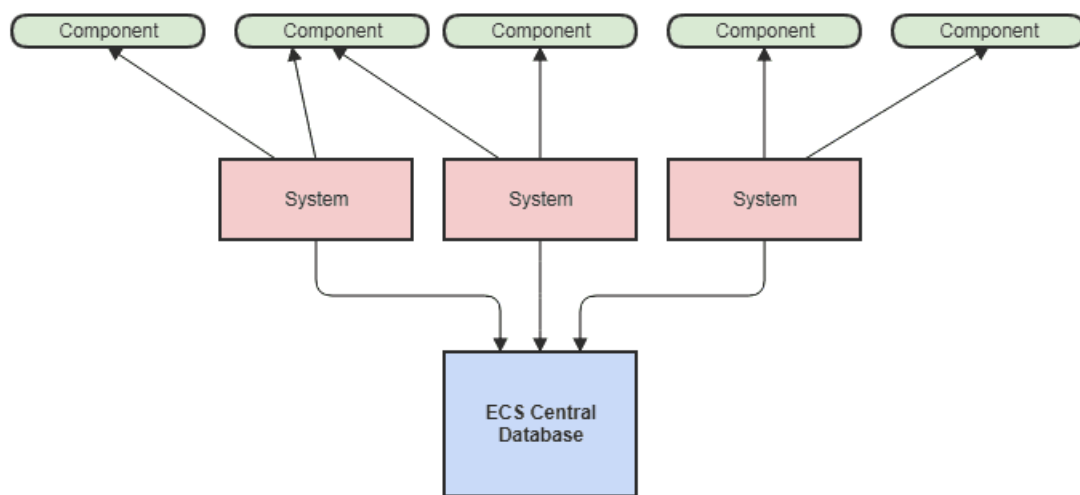


Figure 2 – Logo Unreal Engine

2.3 Réalisation d'un jeu vidéo

Le Titre : Don't get caught !

Pitch : Un joueur incarne un voleur, un autre un gardien. Le joueur incarnant le voleur est en vue FPS et doit s'introduire dans un bâtiment, voler un / des objets, et s'enfuir. Le Gardien quant à lui voit depuis une vue du dessus et avec des caméras, peut activer des pièges/ lumière, déplacer un / des gardes en point'n click.

Univers : Monde réel, Nuit ou thème sombre. Soit "voleur" soit "gardien".

Éléments Narratifs : Voleur renommé, vous parcourez différents musées et maisons afin de vous y introduire et de prendre des objets de valeurs. Seulement, avec les progrès de la technologie, voler devient plus difficile et la police fera tout pour vous mettre des bâtons dans les roues afin de vous arrêter.

3 Base méthodologiques

Le développement d'un jeu vidéo passe par des phases de prototypage. Mon tuteur m'a proposé de suivre une méthode agile avec des sprints à rendre environ toutes les 3 semaines. Chaque sprint est un prototype du jeu, qui ajoute une / des fonctionnalités.

Avant de commencer le développement, nous avons tout d'abord lister les différentes fonctionnalités à implémenter pour chaque joueur (Voleur / Gardien) et mis une priorité dessus afin de savoir par où commencer.

Une fois les priorités décidées, nous avons défini le contenu des premiers sprints.

Pour la modélisation du système, je me suis servi du langage UML (Unified Modeling Language). Il permet d'avoir une modélisation du projet compréhensible par l'ensemble des intervenants sur le projet.

Les phases de tests d'un jeu vidéo sont un peu plus complexes. Certains tests unitaires peuvent être mis en place, cependant ils ne couvriront pas tous les bugs possibles dont le développeur ne dépend pas forcément (comportement particulier du framework, du moteur...). Pour tester, il faut donc jouer au jeu, vérifier que dans les conditions normales de jeu tout se passe bien, et que le jeu ne plante pas s'il y a des situations particulières, tout en vérifiant qu'on ne peut pas trop s'écarter du comportement initialement prévu. Par exemple, vérifier que l'on ne peut pas tomber du monde de jeu dans le vide, mais que si jamais cela arrive, il y a une solution de prévu.

2

Description générale

1 Environnement du projet

Lors de la réalisation de ce projet, aucun existant n'est présent.

Le projet étant un jeu vidéo multijoueurs Peer-To-Peer, il y aura donc 2 utilisateurs qui vont interagir. Un d'eux sera le "Gardien", l'autre sera le "Voleur". Le Gardien aura la possibilité de jouer sur PC (Windows) ou sur un périphérique mobile Android, tandis que le Voleur sera obligatoirement sur PC (Windows). De plus, seul le joueur "Voleur" sera hôte de la partie afin d'assurer une connexion plus stable que si le jeu pouvait être hébergé mobile.

2 Caractéristiques des utilisateurs

Le jeu respectera le classement PEGI 12, c'est à dire :

Le jeu est interdit aux moins de 12 ans :

Il peut comporter des scènes pouvant choquer les plus jeunes (nudité, violence),

il peut aussi comporter un langage légèrement grossier (mais pas d'insultes à caractère sexuel).

En effet, le jeu mettant en scène un voleur, qui sera armé tout comme les pièges, le classement PEGI 12 est le classement le plus bas que le projet puisse respecter.

Les utilisateurs pourront être toute personne âgée de plus de 12 ans.

De plus, le public visé est un public constitué de joueurs régulier de jeux vidéos. Le côté asymétrique du jeu ainsi que le multijoueur intéressera une tranche d'âge aux alentours de 14 ans minimum.



Figure 1 – Logo PEGI 12

3 Fonctionnalités du système

But du jeu :

Voleur : Voler l'objet objectif dans un temps limité sans se faire attraper.

Gardien : Empêcher le vol.

Les 3Cs Voleur :

1. Character : Un être humain, agile, voleur expérimenté, intelligent, connaissances technique
2. Camera : FPS
3. Controller : Clavier/Souris

Les 3Cs Gardien :

1. Character : Les éléments de sécurité du lieu, et s'il y en a un / des gardes
2. Camera : Topview, Gestion
3. Controller : Clavier/Souris

Plateformes : PC-Android

Infos complémentaires : Après quelques recherches, un jeu un peu similaire existe (Of guards and thieves). Le principe est un peu similaire, sauf que mes 3Cs sont différents.

Capacités d'un voleur :

1. Déplacement furtif (sans bruit)
2. Escalader certains endroits
3. Crocheter une serrure
4. Interagir avec des objets
5. Utiliser une arme

Capacités d'un "Policier"(joueur) :

1. Ordonner à un ou des gardes d'aller vérifier une position
2. Allumer / éteindre une lumière
3. Activer / désactiver un piège
4. Activer / désactiver une caméra
5. Capturer un voleur (via piège ou garde)
6. Mettre un piège contre les déplacements (peinture sur mur à escalader, choses au sol pour entendre les pas des voleurs)

Principe d'une partie :

1 voleur

1 Gardien

Le voleur a (à déterminer) XX min pour voler un / des objets. Pendant qu'il tente de s'introduire dans le bâtiment, le policier organise les IA, allume ou non certaines lumières, place des pièges, active certains, etc... Contraintes pour le policier pour pas que ça soit trop facile : il y a un nombre limité de pièges simultanés, les gardes se déplacent à une vitesse raisonnable. Tous les pièges ne sont pas "gagnant", un piège peut seulement révéler la position d'un voleur, un autre sert à la capturer.

4 Contraintes de Développements, d'exploitation, de maintenance

4.1 Contraintes de développements

Le développement du projet utilisera Unity 2018.2.0f2 en tant que moteur. Le langage de programmation utilisé sera le C# avec le framework .Net 3.5 afin de créer les différents scripts liés au jeu. Aussi, il utilisera Entitas 1.11.0.

Le développement s'effectuera sous Windows, et avec Visual Studio 2017. C'est pour cela que le jeu ne sera compatible que sous Windows et Android. En effet, afin de déployer une application pour Mac/iOs, il faut utiliser MacOS et cela rajouterait aussi un certains nombres de contraintes à gérer afin d'avoir un jeu complètement multiplateforme.

Le jeu sera testé sur machine Windows et sur un smartphone Android physique et non un émulateur.

Le délai de réalisation étant assez court, l'ensemble des fonctionnalités ne seront pas développées. Le maximum sera fait dans les temps.

4.2 Contraintes d'exploitation

Le périphérique utilisé pour jouer devra avoir accès à un réseau. Si les deux utilisateurs sont situés sur le même réseau, un accès à Internet n'est pas requis, sinon, il est nécessaire.

4.3 Maintenance et évolution du système

Une fois les fonctionnalités développées, il faudra les adapter afin que le jeu soit équilibré. De plus, il se peut que d'autres fonctionnalités voient le jour afin d'agrandir l'expérience de jeu, de l'améliorer.

3

Etat de l'art / veille technologique

1 Unity



Figure 1 – Unity Logo

Unity est un moteur de jeu multi-plateforme (PC, Console, Mobile, Web) développé par Unity Technologies. Il permet de réaliser des jeux vidéos en 2D et en 3D, à l'aide de deux langages possibles : le JavaScript ou le CSharp. Il existe plusieurs licence, dont une gratuite.

Ce moteur de jeu est beaucoup utilisé dans le monde du jeu vidéo, parcequ'il est assez facile d'accès et complet, ce qui n'empêche pas de réaliser des jeux assez complexes.

Aussi, de par sa popularité, il possède un "Asset Store", une boutique où des modèles 3D sont disponibles et que l'on peut intégrer d'un simple clic à notre projet. Certains sont gratuits, d'autres payants.

Le moteur permet au développeur de ne pas se préoccuper de toute une partie du code. Le rendu graphique et la physique sont gérés nativement par le moteur avec des comportements par défaut que l'on peut bien évidemment ajuster afin qu'ils correspondent au résultat attendu.

Unity propose différentes manières de gérer le rendu graphique, du rendu basique à un rendu orienté haute définition. Plusieurs APIs sont accessibles afin de pouvoir faire du rendu multi-plateforme de manière simple et efficace.

Il intègre un moteur physique qui va permettre de gérer les collisions entre objets, les différentes forces... de manière simple, et surtout modulable par le développeur afin qu'il puisse adapter le moteur à ses besoins.

Unity est un moteur "What you see is what you get", c'est à dire que lorsqu'on construit notre scène dans l'éditeur, c'est ce que l'on verra en jeu.



Figure 2 – Image de l'éditeur Unity

L'éditeur est simple à prendre en main, et les différentes fenêtres sont ajustables comme l'on veut. Sur la gauche, on retrouve la hiérarchie du projet, avec tous les dossiers et fichiers. Ensuite viens la fenêtre principale : l'éditeur de scène. On y retrouve la scène d'un menu principal. La fenêtre suivante est le jeu, c'est ce que la caméra qu'on a fixé dans la scène voit. Enfin, sur la droite, on retrouve l'inspecteur d'élément, et la console.

Unity a des mises à jour régulièrement qui corrigent des bugs et ajoutent du contenu au moteur, et modifient certains comportements de scripts. J'utilise la version 2018.2.0f2 actuellement, mais il n'est pas impossible que je migre vers une nouvelle version si celle-ci propose des ajouts intéressants.

1.1 Fonctionnement de Unity

1.1.1 La scène

La scène est l'endroit où l'on va construire notre monde, notre niveau. C'est dans celle-ci que nous ajouterons, placerons, nos objets. C'est via cette fenêtre que nous créons un niveau de jeu.

Un jeu peut être constitué de plusieurs scènes. Par exemple, plusieurs niveaux de jeux peuvent être représentés par différentes scènes.

1.1.2 La fenêtre de jeu

C'est ici que l'on voit le rendu du jeu tel qu'il le sera pour le joueur. On peut le lancer via le bouton play en haut de au centre de l'interface. Le jeu se lance et on peut y jouer comme s'il était terminé.

C'est utile de le lancer ici afin d'avoir accès à la console et aux différents messages d'erreurs qui peuvent apparaître, afin de corriger les différents bugs.

1.1.3 L'Inspector

L'Inspector est une zone très importante de l'éditeur. C'est via cette fenêtre qu'on peut obtenir toutes les informations sur l'objet sélectionné. On a accès à tous ses composants et les propriétés publiques, que l'on peut modifier.

1.2 GameObject, Prefab, Scripts, Axes

1.2.1 Un GameObject

Avec Unity, chaque objet que nous allons mettre dans notre scène est basé sur un GameObject. C'est le parent de tous les objets que nous allons placé.

C'est à ce GameObject que nous ajoutons tous les différents composant Unity qui existent : Transform (gère la position dans l'espace), Mesh Renderer (permet à notre objet d'avoir un rendu graphique)...

Il existe une hierarchie entre les objets afin de pouvoir regrouper nos GameObject présent sur une scène de manière logique. En effet, un GameObject peut avoir des GameObject enfants, qui peuvent eux même avoir des GameObject enfants... C'est très utile afin de regrouper nos objets en fonction de leur rôle dans la scène.

Les GameObject sont le coeur du développement avec Unity.

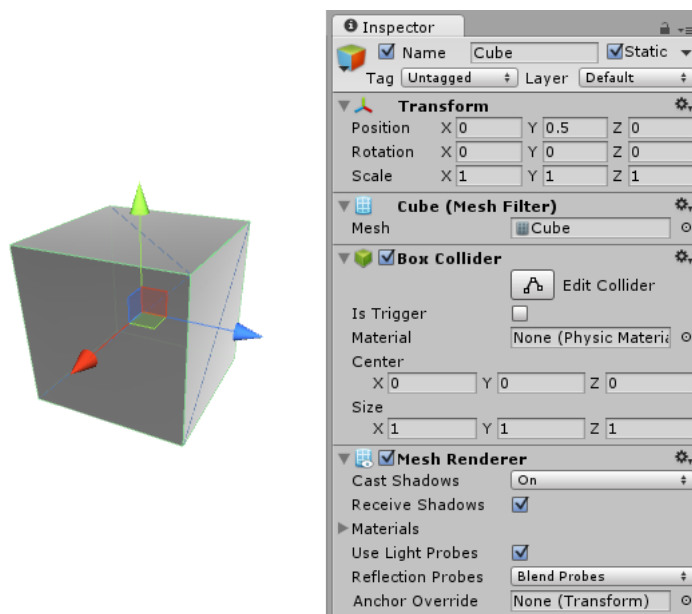


Figure 3 – Exemple de GameObject

1.2.2 Un Prefab

Le Prefab, c'est un GameObject que nous avons paramétré, et sauvegardé en l'état.

Prenons l'exemple d'une balle. Par défaut, elle sera blanche, n'aura pas de nom, sera positionné à un certain endroit, ne sera pas affecté par la gravité...

Le but du prefab est de pouvoir créer nos propres GameObject personnalisés.

Je vais créer ma balle, la déplacer, lui assigner la couleur bleue, faire en sorte qu'elle soit influencé par la gravité... Et sauvegardé dans mes ressources ce GameObject, qui une fois sauvegardé sera un Prefab.

Une fois le Prefab sauvegardé, si je décide de l'ajouter à ma scène, il créera bien ma balle bleue avec les différents paramètres que j'ai sauvegardé auparavant.

1.2.3 Les scripts

Les scripts, ce sont des composants attaché à nos GameObject. En général, le script interagit avec le GameObject afin d'effectuer des actions dessus. Si nous voulons déplacer un objet, je vais créer un script qui permet de récupérer l'entrée utilisateur, et effectuer les modifications nécessaire aux composants de mon GameObject afin d'effectuer l'action demandée. C'est par le biais de ces scripts que nous codons le comportement de nos GameObject dans nos scènes.

Chaque script hérite de MonoBehaviour, c'est à dire que chaque script représente un certain comportement. Il faut voir les scripts Unity comme un bout de code qui va définir le comportement de l'objet dans le jeu.

1.2.4 Les Axes

Dans Unity, afin de pouvoir laisser paramétrables les différentes touches du jeu, Unity nous fourni des axes. Un axe est représenté comme suit : un nom, une description, une touche pour incrémenter, une touche pour décrémenter, une sensibilité, un type (Touche ou bouton de souris, un mouvement, ou un axe joystick), et enfin l'axe concerné (X,Y). Par défaut, la valeur de tous les axes est 0.

Voici un exemple pour l'axe Jump :

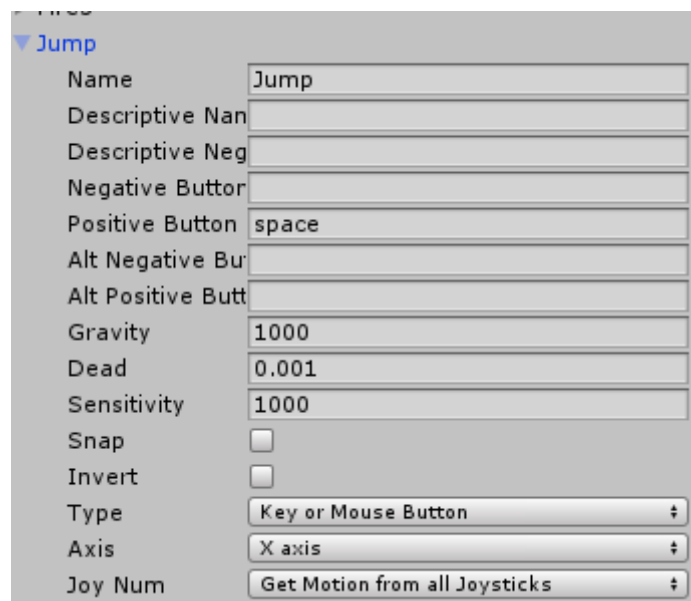


Figure 4 – Description d'un Axe

Lorsque le joueur va appuyer sur Espace, la valeur de l'axe Jump sera de 1. Lorsque le joueur relachera la touche, la valeur retombera à 0. Ce qui permet de savoir très facilement quel est l'action demandé par l'utilisateur, sans se soucier de la touche sur laquelle il appuie.

1.2.5 UNet : Le framework Réseau de Unity

Unity propose son architecture afin de faire communiquer un client avec un serveur : Unity Networking (UNet).

UNet permet de mettre en place un jeu multijoueur Peer-To-Peer. Un jeu Peer-To-Peer, c'est un jeu où l'un des clients sera aussi l'hôte de la partie, c'est à dire que les autres joueurs vont se connecter à lui et c'est l'hôte qui est en charge de la synchronisation entre les clients.

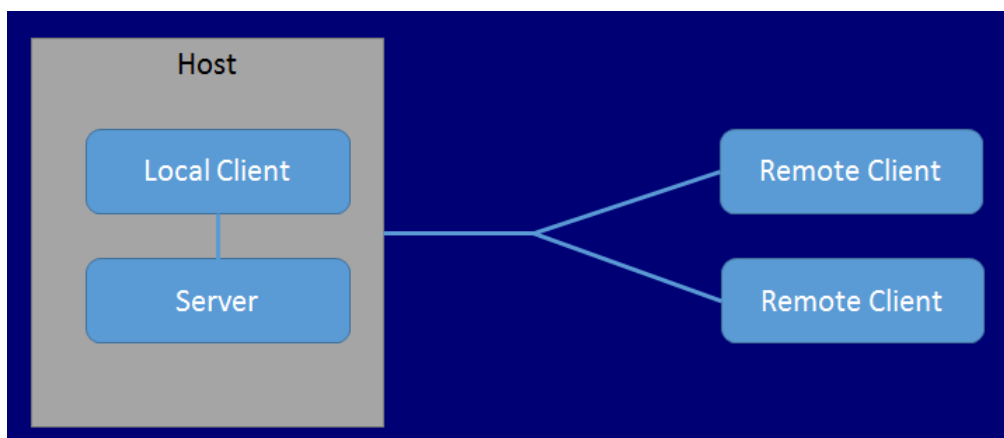


Figure 5 – Schéma du Peer To Peer

Le coeur de UNet est le NetworkManager. C'est une classe qui va contrôler tout ce qui concerne la partie réseau de notre jeu (la connexion d'un joueur, les objets synchronisés, gère les différents événements liés au réseau...). Il a un comportement par défaut mais nous pouvons créer notre propre classe afin de redéfinir, personnaliser, certains comportements.

Tous nos scripts ne sont plus des "MonoBehaviour" comme dit précédemment, mais des "NetworkBehaviour". Cette classe permet d'avoir accès à différents éléments de la communication réseau et de faire savoir à Unity que ce comportement est lié au réseau.

Chaque objet présent dans une scène, afin qu'il puisse communiquer avec les autres clients, a obligatoirement un composant nommé "NetworkIdentity". C'est son identifiant.

Il existe différents composants pré-configurés par UNet afin d'éviter d'avoir à recoder tout nous-même. Par exemple, il existe un NetworkTransform, qui s'occupe de la synchronisation de la position de l'objet entre les clients.

UNet est complet et permet d'avoir un accès "haut niveau" et plus clair dans les communications réseau que l'on souhaite effectuer. De plus, comme la plupart des composants d'Unity, on peut redéfinir son comportement sur certains aspects afin qu'il agisse exactement comme le développeur veut.

2 Entitas



Figure 6 – Logo Entitas

Entitas est un framework CSharp pour Unity implémentant le design pattern Entity Component System (ECS, Entité Composants Systems en français).

Entity Component System repose sur le principe que chaque objet de notre jeu est en fait une Entité, qui est constituée de différents Composants, et le comportement que cette entité aura dépendra de ses composants et des Systèmes qui vont agir sur les composants. Il existe différentes sortes de systèmes, mais le principe général est qu'un système récupère toutes les entités qui possèdent tel composant, et effectue une action sur cet entité.

2.1 Les contextes

Afin de garder une architecture lisible dans le projet, il est important de définir les contextes et ce qu'ils vont contenir.

Un contexte, c'est le nom d'un regroupement de plusieurs objets, là où ces objets vont "vivre", évoluer. Chaque objet appartenant à un contexte peut avoir connaissance d'un objet appartenant à ce même contexte.

En règle général, nous avons deux contextes principaux : Game, et Input.

Le contexte Game va contenir tous les objets liés directement au jeu : l'objet représentant le joueur, les ennemis, les objets interactifs... Ce contexte contient aussi tout ce qui concerne les objets en charge de l'application des règles du jeu : comptage de points de score, de vie, objectifs atteints... En quelque sorte, tout ce que vous voyez dans le jeu.

Le contexte Input quant à lui va contenir tous les objets qui vont faire le lien entre les interactions utilisateurs et le jeu. C'est dans ce contexte qu'on va retrouver des objets qui vont écouter les entrées utilisateurs, comme l'appuie d'une touche sur le clavier, un clic de souris... Et qui va transmettre les informations à notre contexte Game afin que l'action demandé par l'utilisateur soit effectuée.

2.2 Les composants

Un composant est un objet avec certains attributs. C'est plusieurs composants réunis qui vont constituer une entité. Sa classe va implémenter l'interface IComponent d'Entitas. Il faut indiquer le contexte dans lequel le composant se trouve. Son nom doit respecter la syntaxe XXXComponent, pour le générateur de code. Il peut avoir ou non des propriétés. Voici à quoi ressemble un composant :

```

1 using Entitas;
2
3 [Game]
4 public sealed class LifeComponent : IComponent {
5     public int amount;
6 }
7 |

```

Figure 7 – Classe en CSharp d'un composant

Ce composant est nommé Life, se trouve dans le contexte Game, et a une propriété amount qui représente le nombre de vies. Nous verrons dans la partie sur les entités comment accéder aux propriétés d'un composant.

Une fois notre composant créé, on utilise l'outil de génération de code d'Entitas afin de générer nos composants. Cet outil va créer les différents contextes, et générer le code de base d'Entitas (de quoi créer les entités et y ajouter nos différents composants). L'ajout d'un composant est simple, il suffit de re-générer le code. Cependant, retirer un composant ou une propriété d'un composant est plus compliqué. Il faut supprimer le code généré et toutes les utilisations du composant dans notre code avant de pouvoir re-générer le code.

Les composants sont la première chose à créer. En effet, c'est un regroupement de composants dans un objet qui va être intéressant dans ce design pattern.

2.3 Les entités

Une entité est un objet situé dans un contexte, constitué de plusieurs composants. Pour créer une entité, c'est simple :

```
var monEntite = Contexts.sharedInstance.game.CreateEntity();
```

Ici, j'ai créé une entité dans le contexte "game".

Une fois notre entité créée, il faut lui ajouter nos composants. Imaginons que nous souhaitons créer un joueur, il faut donc créer une entité, et lui ajouter nos différents composants qui composent un joueur, comme par exemple de la vie.

```
monEntite.AddLifeComponent(100);
```

Pour ajouter le composant Life, le code généré nous fournit une méthode AddLifeComponent() qui prend en paramètre le nombre de points de vies (notre propriété amount). Maintenant que notre entité a le composant Life, on peut récupérer la propriété tout simplement :

```
monEntite.life.amount;
```

Cependant, un composant peut ne pas avoir de propriétés. Dans ce cas, il agira comme un booléen sur l'entité. Le code généré pour ajouter le composant sera alors

```
monEntite.isPlayer = true;
```

Ce qui aura pour effet d'ajouter le composant Player a monEntite. Et lorsqu'on passe cette valeur à faux, le composant est enlevé de l'entité.

Voici un exemple d'une entité avec différents composants :

```

Entity_25(*6)(Asset, Position, View, ViewParent)
Entity_26(*7)(Asset, Interactable, Position, View, ViewParent)
Entity_27(*6)(Asset, Position, View, ViewParent)

```

Figure 8 – Entites

La création d'une entité va résulter d'un point de vue Unity à la création d'un GameObject vide, dans le bon contexte, c'est à dire en enfant du GameObject représentant le contexte, avec en tant que composants, nos "components" personnalisé (par exemple life). Il est important de différencier les composants Unity des composants créés par nos scripts.

Maintenant que l'on sait créer des composants et les ajouter à nos entités, il faut créer les différents systèmes qui vont nous permettre de créer des comportements.

2.4 Les systèmes

Il existe différentes sortes de systèmes : Initialisation, Execution, Nettoyage, Reactif. Chaque système est instancié dans le GameController, qui va donner l'ordre d'exécution des différents système à chaque frame*.

```

24 void Start()
25 {
26     GameRandom.core = new Rand();
27     var contexts = Contexts.sharedInstance;
28
29     _systems = createSystems(contexts);
30     contexts.game.isInInitialization = true;
31     _systems.Initialize();
32     contexts.game.CreateEntity().AddAmbiance(Resources.Load<AudioClip>(Res.sfxBirdAmbiance1));
33 }
34
35 void Update()
36 {
37     _systems.Execute();
38     _systems.Cleanup();
39 }
40
41 void OnDestroy(){
42     _systems.TearDown();
43 }
44
45 Systems createSystems(Contexts contexts)
46 {
47     return new Feature("Systems")
48         .Add(new CreateGameStateSystem(contexts))
49         .Add(new CreateChallengeStateSystem(contexts))
50         .Add(new CreateRankingStateSystem(contexts))
51         .Add(new CreateFacebookDataSystem(contexts))
52         .Add(new PlayerStateSystem(contexts))
53         .Add(new GainLifeSystem(contexts))
54         .Add(new ManageLivesSystem(contexts))
55         .Add(new PlayGameSystem(contexts))
56         .Add(new GoldSystems(contexts))
57
58         .Add(new NetworkSystems(contexts))
59         .Add(new TranslationSystem(contexts))
60
61         .Add(new SceneSystems(contexts))
62         .Add(new GameBoardSystem(contexts))
63         .Add(new AddViewSystem(contexts))
64
65         .Add(new AddBonusSystem(contexts))
66         .Add(new RemoveInputsSystems(contexts))
67         .Add(new FirstInputSystem(contexts))
68
69         .Add(new ResetSystem(contexts))
70         .Add(new FirstStepSolutionSystem(contexts))
71         .Add(new ProcessInputSystem(contexts))
72
73         .Add(new SaveMoveSystem(contexts))
74         .Add(new UndoMoveSystem(contexts))
75         .Add(new PointsSystem(contexts))
76         .Add(new EndOfGameSystem(contexts))
77         .Add(new RemoveBonusSystem(contexts))
78
79         .Add(new FBLoginSystems(contexts))
80         .Add(new AudioSystems(contexts))
81         .Add(new DestroySystems(contexts))
82     ;
83 }

```

Figure 9 – Un GameController en CSharp

Un système peut implémenter plusieurs interfaces (IInitializeSystem, IExecuteSystem, ICleanupSystem) et étendre la classe abstraite ReactiveSystem.

Un système d'initialisation implémente la méthode Initialize qui est appelée lors du démarrage du jeu. Tous les systèmes qui l'implémentent, dans l'ordre du GameController, vont exécuter leur méthode Initialize(). En général, c'est dans ces méthodes que l'on crée les entités qui vont rester tout le long du jeu, qu'on récupère les différentes données sauvegardées localement ou sur un serveur.

Un ExecuteSystem aura une méthode Execute(). Cette méthode s'exécutera à chaque frame*. En général, un ExecuteSystem est utilisé pour créer des entités ou réaliser une tâche de fond

qui durera toute la durée de vie du jeu. Il est pratique de s'en servir pour attendre des input physique, interactions humaines (clic de souris, appuie d'une touche...) et de réagir en fonction.

Cependant, exécuter du code à chaque frame pour chaque entité n'est peut être pas ce que l'on cherche. Afin d'avoir plus de flexibilité dans notre code, au lieu d'exécuter du code à chaque frame, on peut détecter l'ajout ou la suppression d'un composant grâce aux ReactiveSystem.

Les ReactiveSystem fonctionnent de la même manière qu'un ExecuteSystem, à la différence que leur méthode Execute n'est appelé que lorsqu'un composant est ajouté ou supprimé d'une entité.

Voici les méthodes qu'il faut implémenter :

1. Un constructeur, qui permet de récupérer le contexte dans lequel le système récupère les entités.
2. Une méthode GetTrigger(), qui collecte toutes les entités qui ont le composant souhaité.
3. une méthode Filter(), qui permet d'appliquer un filtre sur les entité récolté.
4. Une méthode Execute(), corps du système. Traite les entités collectées et filtrées.

La méthode Filter est importante car elle permet d'agir sur différentes entités constitués des mêmes composants, ou d'agir sur certains composants en fonction des valeurs de leur propriétés. Imaginons un jouer contre une intelligence artificielle, les deux entités auront les mêmes composants. Pour les différencier, on peut utiliser la méthode Filter().

Une fois qu'une entité n'est plus utile ou on souhaite la détruire, il faut utiliser un Cleanup-System. Ce système implémente une méthode Cleanup(), qui s'exécutera après que toutes les méthodes execute des ExecuteSystem et ReactiveSystem ont été effectué. Il reste aussi les entités qui sont présente toute la durée de vie du jeu, que nous ne voulons pas détruire dans ce système mais seulement quand on quitte le jeu. C'est ce que font les méthode TearDown. Elles sont appelées quand l'utilisateur quitte le jeu, et servent à réaliser toutes les actions souhaitées avant de fermer le jeu, comme par exemple sauvegarder l'avancer d'un joueur et détruire toutes les entités restantes.

Voilà un schéma représentant l'architecture d'un projet utilisant Entitas :

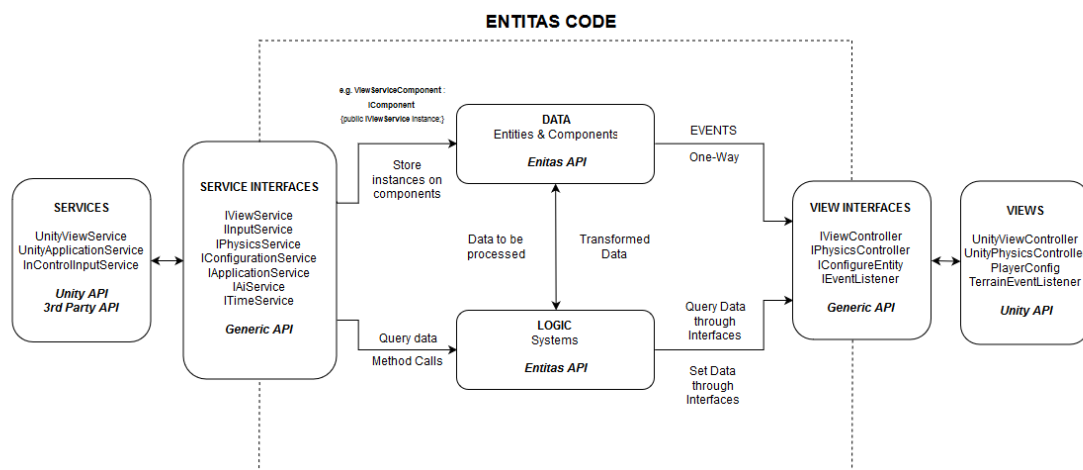


Figure 10 – Architecture d'un projet Entitas

C'est un pattern très populaire dans le jeu vidéo. En effet, il évite la répétition de code et de comportement pour différents objets, il suffit de construire notre entité avec les composants que l'on souhaite pour avoir un comportement identique pour toutes les entités qui ont un composant commun (exemple : des points de vie). De plus, n'utiliser que les scripts Unity appliqué aux objets du jeu, peut altérer les performances si on décide d'en créer un grand nombre. A l'aide d'Entitas, on évite ce problème en ayant un code beaucoup plus léger à charger pour le moteur de jeu.

Entitas est continuellement mis à jour pour être disponible pour les dernières version d'Unity. Cependant, Unity a sorti récemment leur propre framework implémentant EntityComponent-System, mais n'est pour le moment pas aussi complet et efficace qu'Entitas.

4

Analyse et conception

1 Eléments principaux de l'analyse

1.1 Liste des fonctionnalités à implémenter

Fonctionnalités	Détail	Priorités
Partie Voleur		
Déplacement	Permettre au joueur de se déplacer dans le monde	0
Saut	Permettre au joueur de sauter	8
Accroupir	Permettre au joueur de s'accroupir	8
S'allonger	Permettre au joueur de s'allonger	8
Tirer	Permettre d'utiliser une arme	5
Scans de proximité	Permettre de détecter les pièges proches	4
Ramasser un objet	Permettre de ramasser un objet dans le niveau	2
Vie	Vie du joueur	4
Etat	Etourdi, endormi, paralyse	6
Bonus	Améliorations du joueur (Déplacement plus rapide, plus discrets...)	8
FPS	Vue du jeu en première personne	0
Inventaire (simple)	Inventaire avec quelques items	6
Vanish	Disparaître	4
Partie Gardien		
Pièges	Laser, bloquer la vision, verrouiller des portes, pièges lethal, paralysant	3
TopView	Voit l'intérieur des bâtiments via des caméras / une Topview	0
Contrôle d'IA	Pouvoir ordonner via une interface simple des actions à faire à une IA	3
IA	IA de Pathfinding, attaque quand proche, visée pas 100% précise	3
Compétences	Compétences pour bloquer le voleur, avec délai de récupération	6
Partie Monde		
LevelDesign	Créations des niveaux	5
Technique		
Connectivité Mobile / PC	Peer-To-Peer	2
Réalisation des shaders	Shader : Petit programme qui se lance pour chacun des pixels de mon écran. Permet de changer le rendu en connaissance de cause	4

Figure 1 – Tableau des fonctionnalités

1.2 Decoupage du projet selon Entitas

1.2.1 Les Composants

GameContext :

Nom	Attributs	Détails
Player	/	Nous permet de savoir si l'entité est un joueur ou
Name	value : string	Indique le nom de l'entité.
CharacterClassType	type : CharacterClassType	Représente le type de joueur (Gardien ou Voleur)
GameObject	value : GameObject	Un GameObject.
Grounded	/	Permet d'indiquer si une entité est contre un sol ou
Life	value : int	Représente le nombre de points de vie de l'entité.
Trap	/	Permet d'indiquer si l'entité est un piège ou non.
AI	/	Permet d'indiquer si l'entité est une intelligence artificielle.
TrapType	type : TrapType	Indique le type du piège.
Damages	value : float	Représente une quantité de dégâts.
Weapon	/	Permet d'indiquer si l'entité est une arme ou non.
WeaponType	type : WeaponType	Représente le type d'une arme.
Usable	/	Permet de savoir si l'entité est utilisable ou non.

Voici les entités qui appartiendront à l'InputContext :

Nom	Attributs	Détails
HorizontalInput	value : float	Permet d'avoir la valeur du déplacement sur l'axe horizontal.
VerticalInput	value : float	Permet d'avoir la valeur du déplacement sur l'axe vertical.
JumpInput	value : float	Permet d'avoir la valeur d'un saut.
Action1Input	value : float	Permet de savoir si le bouton relié à Action1 est pressé.
Action2Input	value : float	Permet de savoir si le bouton relié à Action2 est pressé.
UseInput	value : float	Permet de savoir si le bouton relié à Use est pressé.

1.2.2 Les Entités

Lors du déroulement du jeu, nous aurons besoin d'avoir les entités suivantes : GameContext :

Nom	Composants	Détails
PlayerEntity	Player,Name,CharacterClassType, GameObject,Grounded,Life	Entité représentant un joueur.
AIEntity	AI,Name,GameObject,Life	Entité représentant une intelligence artificielle
TrapEntity	Trap,Name,GameObject,Life, TrapType,Damages	Entité représentant un piège.
WeaponEntity	Weapon,Name,GameObject, WeaponType,Damages	Entité représentant une arme.

Il y aura aussi toutes les entités représentant des objets avec lesquels le joueur pourra interagir.

InputContext :

Dans ce contexte, nous aurons une unique entité : InputManager. Elle sera unique et aura tous les composants qui sont dans le contexte Input. Le jeu étant en temps réel, il est plus optimal de travailler en fonction du changement d'une valeur plutôt que d'ajouter et supprimer des entités à chaque fois que l'utilisateur va effectuer une action.

1.2.3 Les systèmes

Ce projet sera constitué des systèmes suivants :

1. InputSystem
2. MoveSystem
3. Action1System
4. Action2System
5. UseSystem

- InputSystem : S'occupe de tout ce qui va concerner les composants dans le contexte Input (tout ce qui concerne entrées utilisateurs). A chaque frame va écouter la valeur des différents axes afin de modifier la valeurs des composants correspondants.
- MoveSystem : Permet de déplacer le joueur en fonction des différents Input utilisateurs. Ce système récupère les différentes valeurs des composants de l'entité InputManager, et va appliquer ou non certaines forces au GameObject représentant le joueur.
- Action1System : Permet d'effectuer l'action principale, comme par exemple tirer avec une arme.
- Action2System : Permet d'effectuer l'action secondaire, comme par exemple viser avec une arme.
- UseSystem : Permet d'effectuer l'action "Utiliser", comme par exemple interagir avec un interrupteur. Ce système vérifiera que l'action Utiliser est disponible via le composant "Usable".

1.3 Autres scripts

Même si l'essentiel de notre code sera des composants, des entités, et des systèmes, nous avons besoin d'autres scripts afin de pouvoir interagir pleinement avec Unity, et donc d'objets.

- Objet Player : Attaché au prefab de chaque joueur, ce script permet de créer l'entité qui correspond au joueur et de la lier au GameObject représentant le joueur.

- CameraController : Attaché à la camera du joueur, ce script permet de gérer les différents déplacements de la caméra en fonction du joueur et de la souris.
- GameController : Script principale. Ce script est le coeur du jeu. C'est lui qui contient ce qu'il se passe à chaque boucle de jeu. Il se charge d'instancier nos différents systèmes et de les appeler un à un, dans l'ordre que l'on souhaite. Il est attaché à un objet chargé au démarrage de la scène.
- CustomNetworkManager : Afin de personnaliser différents aspects du network manager, il faut créer son propre script. Il permet de modifier la plupart des comportements du network manager, afin d'avoir le fonctionnement que l'on attend.

```
public class GameController : NetworkBehaviour {
    private Contexts _contexts;
    private Systems _systems;

    void Start () {
        _contexts = Contexts.sharedInstance;
        _systems = CreateSystems(_contexts);
        _systems.Initialize();
    }

    void Update () {
        _systems.Execute();
        _systems.Cleanup();
    }

    private static Systems CreateSystems(Contexts contexts)
    {
        return new Feature("Systems")
            .Add(new InputSystem(new UnityInputService()))
            .Add(new MoveSystem());
    }
}
```

Figure 2 – Exemple de GameController simple

1.4 UML

2 Sprints

2.1 Début du projet

Avant de démarrer les sprints, avec mon tuteur, nous avons commencé à spécifier un minimum le jeu et les choses à faire. Il en est ressorti une liste de fonctionnalités, que nous avons priorisé.

Ensuite, il m'a indiqué ce qu'il attendait pour le premier sprint, et dans quels direction

J'ai configuré mon ordinateur afin de pouvoir m'en servir pour tout le développement (installation de Unity, organisations des dossiers, création d'un repository Gitlab, création d'un projet vide, installation d'Entitas).

2.2 Sprint 01

Pré-requis	Objectifs	Résultats fin de sprint
/	Veille Technologique sur Unity	En cours.
	Étude de la mise en place du framework Entitas	En cours.
	Déplacements d'un joueur	Fait.

Ce sprint a pour but de commencer à étudier les technologies que je vais utiliser et de préparer une "base" pour le jeu afin de me concentrer sur le développement des fonctionnalités au prochain semestre.

La veille technologique consiste à se renseigner sur le moteur de jeu, son fonctionnement, l'état dans lequel il est aujourd'hui...

Le framework Entitas est un framework simple d'utilisation mais très complet. Sa documentation est bien faite en ce qui concerne les bases et quelques exemples existent en open source, ce qui permet de mieux comprendre son utilisation.

Afin de mettre en place les déplacement d'un joueur, il faut tout d'abord créer un projet Unity, créer une scène, un objet représentant notre joueur, et enfin créer les scripts nécessaires à la récupération d'entrées utilisateurs et appliquer un mouvement à l'objet.

2.3 Sprint 02

Pré-requis	Objectifs	Résultats
Sprint 01	Veille technologique sur Unity	En cours.
	Etude de l'architecture d'un projet Entitas	En cours.
	Commencer rédaction du rapport	Fait.
	Faire un diagramme de classes	Fait
	Déplacement multijoueurs	Fait.

Continuation de la veille technologique sur Unity. Lecture de documentation sur la partie UNet et ce qui concerne la mise en place d'un jeu en réseau et sur ce que ça implique. Etude du fonctionnement basique de UNet.

Étudier l'architecture d'un projet Entitas complexe, afin d'avoir plus d'informations. Commencer à rechercher quel est l'architecture d'un projet Entitas multijoueur et comment le framework gère les communications réseau avec Unity et UNet.

Commencer à faire le diagramme de classes du jeu avec les éléments basiques.

Intégrer UNet à notre projet afin que le projet soit multijoueur et qu'un joueur puisse se déplacer et que la synchronisation soit faite avec un autre client.

2.4 Sprint 03

Pré-requis	Objectifs	Résultats
Sprint 02	Fin de la veille technologique Unity	Fait.
	Améliorer le diagramme de classes	Fait.
	Recherche fonctionnement Entitas / Unity	Fait.
	Rédaction du rapport	Fait.

Terminer de se documenter sur Unity, afin d'avoir toutes les informations nécessaire concernant le développement du projet.

Etudier le fonctionnement de Entitas au travers de Unity, comment le framewok est utilisé au travers du moteur. Etudier aussi l'architecture d'un jeu multijoueur avec Entitas et Unity.

Détailler le diagramme de classe de manière à faire apparaître la connexion entre Unity et Entitas et l'interaction entre les deux, ainsi que l'interaction que l'on a avec UNet.

Rédiger le rapport de mi-semestre avec toutes les informations requises.

5

Mise en oeuvre

1 Outils et librairies

1.1 Unity

La version utilisée 2018.3.0f2. C'est une des dernières version supportant le framework réseau UNet.

1.2 Visual Studio 2017

Je me suis servi de Visual Studio 2017 comme IDE. Unity et cette IDE fonctionne plutôt bien ensemble, il permet notamment de lancer le code via Visual Studio pour débbug avec des breakpoints et des espions.

1.2.1 Entitas

La version utilisée est la 1.8.2.

2 Implémentation

2.1 Les composants

Voici les différents composants que j'ai implémenté :

2.1.1 Player

Concernant les composants pour un joueur :

1. PlayerComponent : représente le joueur avec ses caractéristiques.

2. Grounded : représente si le joueur est au sol ou non.
3. Jumping : représente un saut avec une force et une durée.
4. CharacterClassComponent : permet de savoir de quel classe le joueur est (Voleur ou Gardien)
5. GameObjectComponent : sert de stockage du GameObject (Graphismes) du joueur
6. NameComponent : représente un nom

Tous ces composants sont dans le GameContext, et sont rassemblées dans le fichier PlayerComponents.cs.

2.1.2 GameComponents

Voici les composants que j'ai implémenté pour le fonctionnement du jeu :

1. Interactable : Composant pour savoir si le joueur peut interagir ou non avec cette entité.
2. Capsule : représente une capsule, ici, un objet à voler.
3. Looked : composant pour savoir si le joueur regarde actuellement l'entité.
4. Selected : Composant pour savoir si le joueur a sélectionné l'entité.
5. Trapped : Composant pour savoir si l'entité est actuellement piégée ou non

Tous ces composants sont dans le GameContext, et sont rassemblées dans le fichier GameComponents.cs.

2.1.3 InputComponents

Voici les composants que j'ai implémenté gérer tout ce qui concerne l'entrée utilisateur :

1. InputManager : Composant pour indiqué que l'entité est celle qui va gérer les input.
2. HorizontalInput : Permet d'avoir la valeur de l'axe Horizontal
3. VerticalInput : Permet d'avoir la valeur de l'axe Vertical
4. JumpInput : Permet d'avoir la valeur de l'axe Jump
5. Fire1Input : Permet d'avoir la valeur de l'axe Fire1
6. UseInput : Permet d'avoir la valeur de l'axe Use

Tous ces composants sont dans le InputContext, et sont rassemblées dans le fichier InputComponents.cs.

2.2 Les entités

Maintenant que j'ai tous mes composants, je peux créer mes entités avec les composants nécessaires.

2.2.1 Le voleur

Le voleur est un joueur, il est composé de : PlayerComponent, CharacterClassComponent, NameComponent. Les composants Grounded et Jump seront ajouté à cette entité si le joueur est au sol ou en train de sauter.

2.2.2 Le Gardien

Le voleur est un joueur, il est composé de : PlayerComponent, CharacterClassComponent, NameComponent et GameObjectComponent. Il ne peut pas sauter et il n'y a pas d'intérêt à savoir si le gardien est au sol ou non puisqu'il a une vision du dessus de la carte.

2.2.3 La capsule

La capsule est un objet que le voleur doit voler. Une entité représentant une capsule a donc : Interactable, Capsule. Les autres composants comme Selected ou Trapped sont ajoutés par le biais de systèmes.

2.2.4 L'InputManager

Je ne créer qu'une entité pour gérer toutes les inputs utilisateur. Cette entité a donc tous les composants présent dans l'InputContext. Chaque fois que l'utilisateur modifie la valeur d'un axe en pressant une touche, la valeur du composant correspondant est mis à jour.

2.3 Les systèmes

2.3.1 InputSystem

Ce système est exécuté à chaque frame du jeu. Il remplace la valeur des composants concernant les axes à chaque frame, ce qui permet de ne pas avoir de délai entre l'interaction de l'utilisateur et la modification de la valeur.

2.3.2 SelectedSystem

Ce système est appelé lorsque le composant "Selected" est ajouté à un objet. On vérifie quel est le type de l'entité, et on agit en conséquence. Dans le cas de nos capsules par exemple, il faut mettre la capsule en verte et afficher au gardien l'interface de sélection de pièges.

2.3.3 GroundedSystem

Ce système est exécuté à chaque frame. Il vérifie si le voleur est sur le sol ou non.

2.3.4 GuardianMoveSystem

Ce système permet au gardien de se déplacer. Le gardien étant sur une surface tactile, il ne se sert pas de l'input manager pour le déplacement. Tout ce qui concerne le déplacement est donc dans ce système (dont le zoom).

2.3.5 ThiefMoveSystem

Ce système, en fonction de la valeur des axes, va déplacer le voleur dans la direction souhaitée par l'utilisateur.

2.3.6 ThiefUseSystem

Ce système, en fonction de la valeur de l'axe Use, va vérifier que le voleur souhaite utiliser un objet avec lequel il peut interagir, puis ensuite effectuer l'action liée à l'interaction de cet objet. Par exemple, pour voler une balle, la balle doit disparaître.

2.4 Les prefabs

2.4.1 Le voleur

Le prefab "AutoCreatedSoldier" est le GameObject représentant le voleur. Il contient les graphiques et son script associé. Son script se charge de créer l'entité correspondant au voleur avec les bons composants. C'est ce prefab qui est instancié lorsque le voleur se connecte à la partie.

2.4.2 Le Gardien

Le prefab "AutoCreatedGuardian" est le GameObject représentant le gardien. Son script se charge de créer l'entité correspondant au gardien avec les bons composants. Ce prefab est instancié lorsque le gardien se connecte à la partie.

2.4.3 La capsule

Le prefab "Capsule" est le prefab qui représente les capsules du jeu (les balles rouges). On y retrouve leur script qui permet de créer l'entité correspondante, et le comportement à avoir lorsque le joueur souhaite interagir avec.

2.4.4 Le LobbyManager

Le LobbyManager est l'objet qui va gérer le multijoueur. Il a un script personnalisé afin d'avoir le comportement spécifique à mon jeu. Il s'occupe d'instancier les joueurs comme il faut, gère la connectivité (sur quel port on lance l'host, quelle ip on se connecte en tant que client...). C'est via lui aussi que l'on peut récupérer les informations concernant les joueurs dans le salon avant la partie, qu'ils puissent indiquer s'ils sont prêts à jouer par exemple, et de lancer la partie.

6

Bilan et conclusion

1 Ce qui est fait

Durant ce premier semestre, j'ai pu effectuer des recherches sur le framework Entitas et sur Unity, le moteur de jeu.

Comprendre le fonctionnement entre le moteur de jeu et le framework est indispensable afin de bien architecturer son projet, et utilisé les bons outils que l'on a à disposition.

De plus, j'ai mis en place l'environnement préparatoire au développement du jeu. Ce sont les bases du développement du jeu. Tout est configuré, il ne reste qu'à développer les différentes fonctionnalités et règles.

Aussi, la mise en place de cet environnement m'a permis de mettre en application ce que j'apprenais via la veille technologique et l'étude du framework, ce qui m'a permis de confirmer que je comprenais bien le fonctionnement.

2 Retards et reste à faire

Je n'ai pas pris spécialement de retard durant le premier semestre.

L'environnement de développement étant prêt, et les fonctionnalités définies et priorisées, il faut maintenant s'attaquer à leur développement. Aussi, il ne faut pas oublier de prendre en compte les différentes phases de tests afin d'éviter tout bug du jeu.

3 Bilan qualité

L'ensemble du projet est documenté. Les classes et méthodes sont commentées. Les noms de fonctions et variables sont logiques. Aussi, en annexe, il y a le cahier du développeur, de l'utilisateur et aussi un manuel d'installation pour que la reprise du projet soit simple et faisable en ne lisant que ces documents.

4 Capitalisation

Toutes les 3 semaines, j'avais un rendez vous avec mon encadrant Ubisoft. Le fonctionnement en méthode agile avec des sprints de cette durée me paraît très adapté à ce type de projet. De plus, même si l'encadrant n'était pas présent physiquement, il était très disponible par mail et téléphone.

Prévoir plus précisément les sprints et réévaluer la longueur des tâches sont des erreurs que j'ai commise qui m'aurait évitée d'avoir un des objectifs trop utopiste.

Annexes

1 Manuel d'installation

Don't Get Caught est un jeu pour 2 joueurs avec un joueur sur ordinateur sous Windows, et un joueur sur mobile sous Android.

1.1 Pour Windows

Le joueur PC sera le voleur et l'hôte de la partie. Afin de pouvoir héberger la partie et permettre aux autres joueurs de se connecter, il est nécessaire de connaître son adresse IP et d'ouvrir un port.

Afin de connaître son adresse IP, il suffit d'aller sur un site tel que <http://www.mon-ip.com/>. Une fois qu'elle est connue, il faut maintenant ouvrir le port qu'utilisera l'autre joueur pour se connecter.

Pour ouvrir un port, la manipulation dépend de votre fournisseur d'accès internet, mais la démarche reste la même : il suffit d'aller sur l'interface de sa box, trouver la partie concernant les ports, en choisir un, et ajouter une règle afin qu'il accepte les connexions et donc que l'autre joueur puisse se connecter à l'ordinateur.

Pour plus de précisions, il existe plusieurs tutoriels sur internet concernant votre fournisseur d'accès internet et l'ouverture d'un port.

1.2 Pour Android

Pour jouer à Don't Get Caught sur Android, il faut tout d'abord déplacer l'APK sur l'appareil et ensuite depuis le gestionnaire de fichiers, cliquer dessus afin de lancer l'installation. Ensuite, le jeu devrait fonctionner.

2 Manuel Utilisateur

2.1 L'hôte | Le voleur

L'hôte doit choisir le port sur lequel il va héberger le serveur, puis cliquer sur "Host". Une fois cette action effectuée, il attend la connexion de l'autre joueur.

Une fois les deux joueurs connectés, l'hôte peut démarrer la partie. Il incarne le voleur. Il démarre dans une salle, son objectif est de parcourir toutes les salles et de voler un maximum d'objets (balles rouges) afin la fin du temps imparti.

Les touches du voleurs sont :

1. ZQSD Pour le déplacement
2. E pour voler un objet



Figure 2 – Le Voleur

2.2 Le client | Le Gardien

Le client doit renseigner l'IP à laquelle il souhaite se connecter, à savoir l'adresse IP de l'hôte. Aussi, il faut qu'il respecte le format adresseIP :Port, le port étant celui que l'hôte a ouvert précédemment.

Une fois le champ renseigné, il ne reste plus qu'à cliquer sur "Connect". Si la connexion a fonctionné, les deux joueurs seront visibles dans le menu.

Une fois la partie lancée par l'hôte, le gardien apparaît au dessus du voleur. Il peut se déplacer en faisant glisser son doigt sur l'écran. Il peut interagir avec toutes les balles rouges afin de les piéger pour en empêcher leur vol. Le gardien doit toucher une balle rouge, la balle sélectionnée devient verte, et un menu apparaît. Il peut choisir le piège. Pour piéger la balle, il faut cliquer sur le bouton représentant le piège.

Pièges disponibles :

1. Cage : entoure d'une cage la balle pendant 10 secondes.

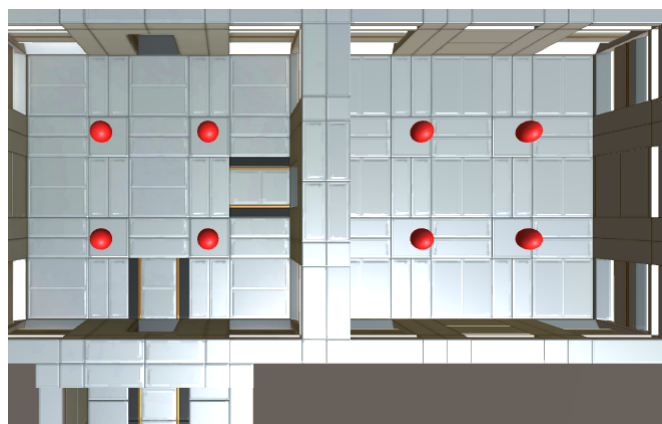


Figure 3 – Vue du Gardien

3 Manuel Développeur

3.1 Logiciel et Framework nécessaire

Don't Get Caught est réalisé avec Unity 2018.3.0f2 et utilisant le framework Entitas 1.8.2 et UNet, en C#, et avec l'IDE Visual Studio 2017.

3.2 Architecture des fichiers Unity avec Entitas

Ce projet respecte l'architecture d'un projet Unity utilisant le framework Entitas. Voici l'architecture des fichiers :

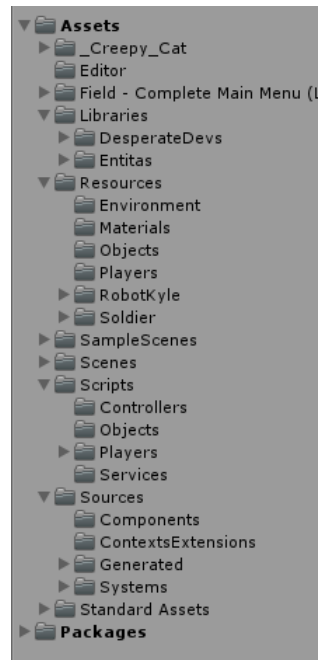


Figure 4 – Architecture

Le dossier principal est le dossier Assets. A l'intérieur, on y retrouve plusieurs sous dossiers.

1. CreepyCat : Asset gratuit récupéré sur l'Asset Store. Contient tous les éléments pour construire un niveau (murs, salles...)
2. Field Complete Main Menu (Lite) : Asset gratuit. Utilisé pour faire le menu principal de Don't Get Caught.
3. Libraries : Contient les bibliothèques utilisées, ici Entitas.
4. Resources : Contient tous les prefabs du projet
5. SampleScenes : Asset gratuit. Contient différents éléments basiques, comme des matériaux, réutilisable dans tout le projet.
6. Scenes : Contient les scènes du projet
7. Scripts : Contient tous les scripts liés aux objets Unity et autres.
8. Sources : Contient tous les scripts utilisés par Entitas. Les composants, les systèmes, et le code généré par le framework.
9. Standard Assets : Asset gratuit. Contient différents éléments basiques.

Il est important de respecter cette architecture, notamment celle du dossier Sources, pour qu'Entitas n'ait pas de problèmes à générer le code souhaité.

3.3 Développer avec Unity / Entitas

Afin de pouvoir utiliser Entitas dans un script, il faut l'inclure en utilisant

```
using Entitas;
```

Il faut premièrement créer nos composants. Chaque composant est sous la forme

```
[Context]
public class MonComposant : IComponent
```

Une fois le composant créer, il faut retourner sur Unity, tools, Entitas, et cliquer sur "Generate". Le composant est ajouté au contexte et peut être ajouté aux entités depuis le code, comme par exemple :

```
_gameContext.CreateEntity().AddMonComposant();
```

Pour plus d'informations, consultez la documentation officielle [Entitas](#), [Unity](#) et UNet (incluse dans Unity).

4 Tests

Pour effectuer un test dans un jeu vidéo, il faut suivre des scénarios. Le testeur, en général différent du développeur, connaît la fonctionnalité à tester, les pré-requis, l'action à effectuer, et le résultat attendu.

4.1 Exemple de scénario : le déplacement Voleur

1. Pré-requis : Aucun.
2. Fonctionnalité : utiliser les touches ZQSD pour se déplacer dans le niveau
3. Résultat attendu : Parcourir le niveau sans comportement inattendu (collisions non détectées, ralentissement à certains endroits...)

4.2 Exemple de scénario : Gardien : mettre le piège Cage

1. Pré-requis : Aucun.
2. Fonctionnalité : Sélectionner un objet et le piéger avec une Cage
3. Résultat attendu : L'interface de sélection de piège apparaît. Le clique sur le bouton Cage fait apparaître la cage autour de l'objet.

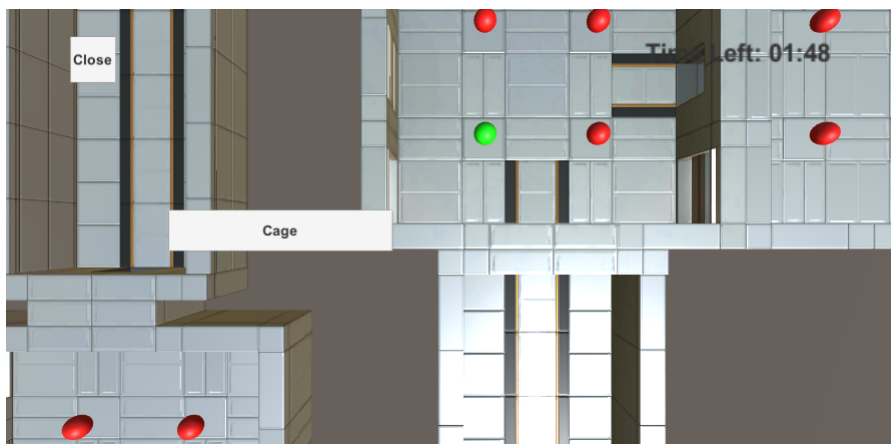


Figure 5 – Trap UI

Le jeu étant multijoueur, il faut tester toutes les fonctionnalités "Hôte" avec un client qui observe que l'action effectuée est bien répliquée, et inversement, mais aussi que les actions qui ne doivent être visible que par un joueur localement n'est pas répliquée.

4.3 Liste des fonctionnalités testées

4.3.1 Pour le Voleur

Hosting en réseau local Déplacement Voler un objet

4.3.2 Pour le Gardien

Connexion à un hôte Déplacement Piège : Cage

Réalisation d'un jeu vidéo : Don't get caught !

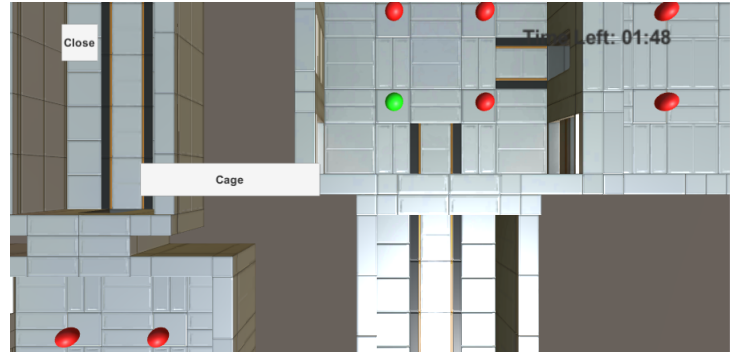
Alexandre DHALENNE

Encadrement : Mathieu DELALANDRE

En collaboration avec Ubisoft

Constat

Jeu fonctionnel en l'état. Principe du jeu globalement respecté. L'architecture Entité-Composant-Système permet une évolution simple.



Interface du Gardien

Solution

Afin de terminer le jeu, on peut rajouter plusieurs fonctionnalités au voleur et au gardien pour augmenter le gameplay. Le jeu évoluera en fonction des idées et envies du développeur.

Conclusion

Ce projet est un prototype de jeu vidéo crossplatform multijoueur. Le gameplay est encore pauvre mais le jeu est jouable en l'état et poursuivre son développement permettra d'améliorer son gameplay.



Le Voleur

Réalisation d'un jeu vidéo : Don't get caught !

Alexandre DHALENNE

Encadrement : Mathieu DELALANDRE

En collaboration avec Ubisoft

Constat

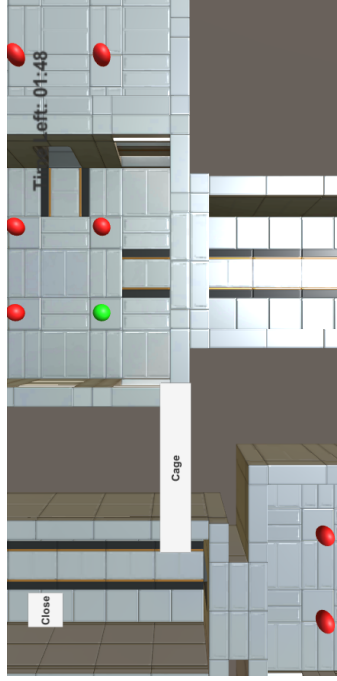
Jeu fonctionnel en l'état. Principe du jeu globalement respecté. L'architecture Entité-Composant-Système permet une évolution simple.

Solution

Afin de terminer le jeu, on peut rajouter plusieurs fonctionnalités au voleur et au gardien pour augmenter le gameplay. Le jeu évoluera en fonction des idées et envies du développeur.

Conclusion

Ce projet est un prototype de jeu vidéo crossplatform multijoueur. Le gameplay est encore pauvre mais le jeu est jouable en l'état et poursuivra son développement permettra d'améliorer son gameplay.



Interface du Gardien



Le Voleur

Réalisation d'un jeu vidéo : Don't get caught !

Résumé

Projet de Recherche et Développement sur la réalisation d'un jeu vidéo utilisant le moteur de jeu Unity, le framework Entitas, en C#. Le jeu est un jeu multijoueur asymétrique, cross-plateformes Windows / Android.

Mots-clés

Unity,CSharp,Jeux vidéos

Abstract

Research and development project about videogame development, using Unity, Entitas Framework, C#.It's an asymmetric multiplayer game, crossplatform Windows / Android.

Keywords

Unity,CSharp,VideoGames

Entreprise

Ubisoft

Tuteur entreprise

Romain CORREIA

Étudiant

Alexandre DHALENNE (DI5)

Tuteur académique

Mathieu DELALANDRE