****

**Integrated Project 1 (Games)**

[Group 6]

[Project Title]

**Software Design Document**

Nestor Medina, Riccardo Testa, Adam Hood, Nadine Hazel, Courtney Healy, Lauren Findlay

April/2019

# Purpose (5)

*State briefly the purpose of this document and its intended audience. The description given should complement the game design document and give an overview of what is being developed.*

The purpose of this document is to give a brief explanation of all the code used in the different stages of development of the game. This game is meant to be played in a library, so the intended audience encompasses people of all ages, although the cyberpunk theme of the game may attract younger people the most. Since it is a requested game most of the constraints were initially given by the client and will be described in the scope section below.

# Scope (15)

*This section will serve as a brief description of your product. Describe the in-game goals and objectives, and how the game will be deployed (target platforms). List and briefly describe the features that the game incorporated. The features should coincide with what is described in the game design document.*

As explained before, this game is requested and meant to be played in a library to encourage its sci-fi and fantasy sections, so the target platform is the same the client asked for: windows PC, is the easiest platform to get since other kind of computers are too expensive and a console requires too much space (more cables, controllers, etc.).

The game is easy in mechanics but needs a bit of background to be fully understood.   
The player is an artificial intelligence trapped in a big company’s intranet. This I.A. begins to acquire self-thinking and understands its situation, the urge to escape comes when it realizes what it is going to be used for: testing and, if it doesn’t work, be destroyed.

Now, the mechanics, are simple: shoot to defeat all enemies and escape to the next layer of the intranet moving through rooms and finding different kinds of enemies, with a boss at the end of each layer.   
Some of the features this game includes are the following:

* Random generated levels (a new level generates each time the player starts the game or advances to the next level)
* Shooting
* Health system
* Enemies
* Player ability (triggered with a different button, it has a different effect on enemies)
* Travelling through rooms (each room has a portal that, when activated, allows the player to travel to the next room with an animation, simulating travelling through a wire)

# Data Design (60)

## DungeonGenerator

This class is in charge of the random level generation. It creates a certain amount of random rooms and makes a path connecting some of these rooms with another room, declaring a start and finish room. Then, the rooms that were included in the path are filled with walls, floor, enemies and the rest of the objects, and the ones that were not included remain empty. Some of the methods included in this script are:

* void RoomGenerator\_OnRoomsGenerated()

Keeps track of the generated rooms and calls other methods to generate walls and other stuff inside the rooms.

* void ConnectDoors()

Checks which rooms are filled and connects each room with another one so each room has a enter an exit door.

* void AddWalls()

Checks the intended dimension of each room and generates walls in a square shape according to it.

These variables are included in the script:

* A public array of GameObject containing everything needed to make the rooms: floor tiles, wall tiles, void tiles.
* int RoomCount : contains the number of filled rooms.
* float MainRoomFrequency : contains the frequency in which filled rooms spawn.
* float RoomConnectionFrequency : contains the frequency in which filled rooms are connected.
* Lists of integers type of room and number of rooms.

## PlayerMovement

This class contains everything related to the player: movement, shooting and health. Since it holds different features it contains quite more variables such as:

* GameObject healthbarPlayer : assigns the health bar prefab.
* GameObject loosePanel : assigns the panel that shows when the player is defeated.
* bool cd, blocked : holds the cooldown between shots and whether the player can or cannot shoot.
* float health, maxhealth : variables for the current health and the max health of the player.
* float speed : the speed at which the player moves.
* Vector2 direction, rot, shootingDir : these vector2 variables hold the direction the player is aiming at and uses them to instantiate a bullet in that direction.
* Rigidbody2D : is the rigidbody2D unity component used to inform when an object (a wall or an enemy bullet) collides with the player.
* SpriteRenderer : unity component to render a sprite for the player.
* Animator : makes the animations of the player run correctly (idle, moving in each direction)

Some of the methods utilised in this script are:

* void Start : assigns values to some of the variables and assigns components of the player to variables so that these can be used later much easier.
* private void OnTriggerEnter2D(Collider2D collision) : checks if the rigidbody2D component of the player collides with something and calls a function depending on the object it collided with.
* void Damage() : checks if the player gets hit by an enemy and if so, reduces health.
* void Loose() : destroys the player when killed and shows a defeat panel on screen.

## Chaser

This class is a certain type of enemy. These one chases the player around and deals damage in contact. Checks if the player is nearby and if so, follows it with a slightly slower speed than the player’s.

These are some of the variables used:

* GameObject healthBar : holds the enemy’s health bar game object.
* bool active, chasing : checks if the enemy is or is not chasing the player.
* float speed, minRadius, minPlayerRadius : holds the speed of the enemy and the distance the player needs to be at in order to start chasing it.
* Rigidbody2D rb : holds the rigidbody2D component of the enemy.
* float health, maxhealth : holds the current health and max health of the enemy.

Methods used in this class:

* void Start () : assign components to variables.
* void Follow() : this method is in charge of the movement of the enemy when the player is nearby. Follow the player when is close enough and stand still when is out of range.
* private void OnTriggerEnter2D(Collider2D other) : checks if the enemy gets hit by a player’s bullet.
* void Damage(float amount) : takes the amount of damage dealt by the player and deducts it from the enemy’s health or destroys the enemy if it gets killed.

## Splitter

This class is another type of enemy. These one moves around bouncing on walls and when killed splits into smaller copies of itself until a certain point.

Lots of variables and methods of this class are the same used in the chaser class so only the new ones will be explained:

Variables:

* GameObject childrens : variable for the copies of itself that spawn when the big one is killed.

Methods:

* void Damage(float amount) : these method remains the same except for one thing. When the health is < 1 instead of just destroying the enemy game object, it instantiates two children right in the spot where the it was killed.

## RedCode

These class holds everything related to the ability of the player. This ability makes the player shoot a different kind of bullet that, if it hits an enemy, it transforms this enemy into a clone of the player that shoots normal bullets but is not able to shoot the ability bullet. It has a considerable cooldown and the bullet travels slower than the normal ones.

Variables used:

* List<GameObject> clones : this is a list of game objects that contains the spawned clones of the player.
* GameObject bullet, clone : game objects for the ability bullet and the clone prefab.
* float coolDown : holds the cooldown time for the ability to be ready again.
* float time : this variable is assigned to a timer such as Time.deltaTime in order to keep track in real life seconds of the cooldown.

Methods included in this class:

* void Start () : initial settings such as set the timer to 0 or create the list of clones.
* void Update () : checks if the cooldown is on or off and if the player is pressing the ability key. Also holds the timer.
* void ShootSpecial() : shoots the special bullet if the cooldown is off and the player hits the ability key.
* void AddClone(Vector3 pos) : if the bullet hits an enemy, this function instantiates a clone of the player right where the enemy was hit.
* void RemoveClone(GameObject t) : removes the clone from the game when it is hit by an enemy.

# Problems and Solutions

1. In early stages of the game, there was a different system to travel between rooms that consisted of handmade alleys connecting the rooms. These alleys had doors that only opened when the player defeated all the enemies in the room but instead of working like this, the enemies will just disappear when the player entered the room (when the player collided with the hitbox of the door). This was solved by changing the way the player moved between rooms to what it ended up being in the game: wires with just an animation.
2. We had problems with the random generation of levels until the last moment. We even thought about just removing it and creating manually structured levels. At the beginning, the random generation was just an object moving around generating tiles of floor and walls and the level ended up being just a large room with some walls and enemies in it so there was no start nor end of the level. The second stage of this random generation script was about generating a few squared rooms connected by alleys so there could be a starting room and an ending room for the level. It ended up not working at all because some tiles were layering on top of other tiles and sometimes the level was so broken the player couldn’t even move. It wasn’t until the very end that we could make it work and it could be implemented into the game after tries and tries of different methods.   
   The last algorithm basically spawns a fixed number of rooms with random distribution in the two dimensions inside a circle with a fixed radius, and choose the biggest ones, then generate a minimum spanning tree for connections and pick the start and the end room. The algorithm is taken from another game (tiny keeps), which code has been implemented in unity under the MIT licence on GitHub. Here is the repository <https://github.com/jongallant/DungeonGenerator>
3. Since the procedural generation didn’t implement a good system for the rooms, and avoiding repetitions was one thing to consider, rooms population algorithm takes an image from the assets checking the dimensions and using the colour of the pixels to spawn different prefabs in different position. This enabled designers to make some rooms that were designed to be fun and unique. This method was implemented following the tutorial at the following link: <https://www.youtube.com/watch?v=B_Xp9pt8nRY>