Written Report - Ultru

Pablo Sarmiento and Martin Thomsen

May 22, 2016

1 Project Concept

ULTRU is a First Person Shooter game that takes place in randomly generated levels of a space station. The player needs to explore each level collecting items and information, hack the main computer, and find and escape in the elevator.

1.1 Random generation of space station

Two different random map generation approaches were analyzed in the design stage of the project. Tile based and Prefab based.

1.1.1 Tile based

Traditionally used to generate 2D "roguelike" dungeons. It is basically a 2-dimensional matrix where rooms and corridors are carved in.

After generating the 2d matrix, the actual map will have to be built in Unity by spawning "tile" game objects that need to be created. The following tiles needed are:

- Corner tile
- Wall tile
- Empty tile (just floor and ceiling)
- Door tile

Pros	Cons
Room overlapping is easily prevented	Low performance
Square based layout so it feels like a Space station	Single level map (no stairways)
Robust/compact layout	Repetitive. Always squared rooms

Table 1: Pros and Cons of the Tile based approach .

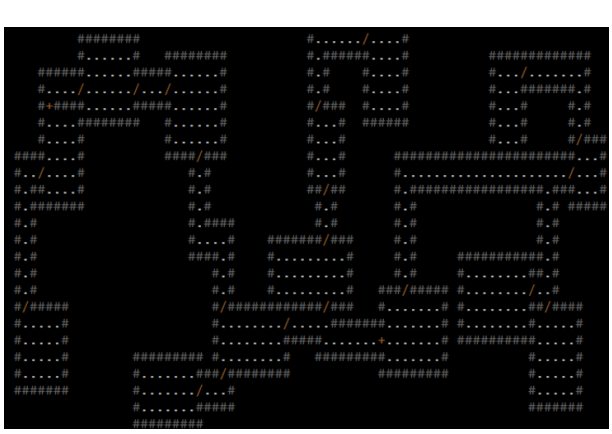
References http://theantranch.com/blog/the-cauldron-a-random-dungeon-generator/

1.1.2 Prefab based

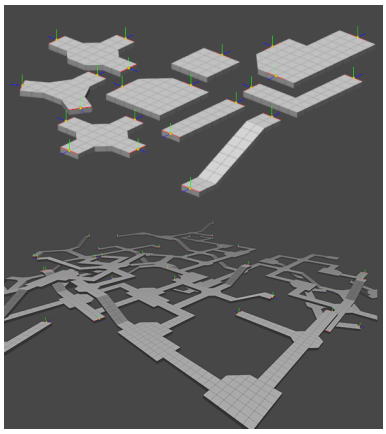
Instead of using a 2d matrix to place tile game objects. A set of room and hallways prefabs are directly placed on the scene. Each prefab has "possible exits" that are used to connect to other prefabs.

Pros	Cons
Fast performance	Final layout looks branch based
Can lead to more interesting levels	Issues with overlapping
Easier to decorate rooms and walls	Time has to be spent in room modelling
Allows starways (multi-level map)	

Table 2: Pros and Cons of the Prefab based approach .



(a) Tile based generation



(b) Prefab based generation

Figure 1: Comparison of the two approaches to level generation

References http://gamedevelopment.tutsplus.com/tutorials/bake-your-own-3d-dungeons-with-procedural-recipes–gamedev-14360

After this research, the 2nd approach (prefab based) was the one chosen for ULTRU. Since the game feature levels with up to 50 rooms, the performance issues with the first approach was a decisive factor. The algorithm works in the following way:

It starts placing a fixed Spawn Room (where the player character starts), then it randomly selects one of the room prefabs and connects it with its exit. It checks its bounds to see if its colliding with any of the existent rooms. If it collides, it discards that room prefab and takes another one. If after 10 times a suitable room has not been found, the algorithm blocks that exit (a blocked door prefab will be spawned there) and looks for another available exit to start the process again. This goes on until a certain number of rooms are successfully placed. If all doors are blocked before reaching the desired number of rooms, the algorithm destroys all created objects and starts again.

1.2 Random items/terminals - Issues with placement positions

ULTRU features rooms with random objects in them. Those objects can be enemies, terminals and weapon pickups. The basic approach is to pick a random point in the floor plane of the room and spawn an item on it. When spawning several objects in the same room using this approach, two problems have to be taking into account.

Items can overlap with each other or with the walls of the room. Items are spread randomly around, which is not the optimal aesthetics for a Space Station game.

To solve both problems at once, a grid based approach was used. If there is only one object in the room, this would be in the exact center. If there is more, they will be placed along the same axis of the first one, but displaced a fixed distance.

1.3 Truly random enemies

Making truly random enemies can be difficult, because of how you need constraint the randomness. If everything is completely random, then it is impossible to balance it for a game. Even random games needs a progression curve to be interesting to the player, and not be frustrating or boring.

To fix this dilemma a specific approach to the problem was taken. By very specifically constraining in which areas the enemies were random, it was possible to make them progressively more difficult. The plan was to create 3 different enemy types, based on 3 different kinds of movement; a walking enemy, a floating enemy and a static enemy. Each of the 3 types had a base model that fit their movement type. (Legs for the walker, thrusters for the floater and a base for the static one). The randomness would here be made in form of prefabs, built upon this base. Each of these prefabs would consist of attributes, connection points, a model and a cost. The cost was then used to progressively scale the enemies, as the ones in the beginning of the game wouldn't have as many points to spend on modules, as the ones later on.

Attributes could be any of the following, but always defined by the modular piece: Armor, health, shield, weapons, ammo capacity, movement speed, attack damage, and extra abilities. By following this approach, each enemy would look slightly different, but have the same silhouette and basic behavior, which means that the player could develop strategies to defeat each kind, and in that way give the player a feeling of getting better at the game, while still leaving variation to each single enemy.

1.4 Balanced Levels

As with the enemies, the levels need a difficulty curve as well. You can balance a level by changing of of many factors, like: Level size, amount of enemies, which kinds of enemies, amount of health/weapon pickups, which rooms, to name a few. In order to make progression, these things needs to get more difficult to surpass, the later on in the game you are.
Two ways of balancing can be weight and cost.

Weight is a chance value, that is given to each individual factor, eg. if an enemy will spawn, and which kind, etc. This weight can be changed throughout the playthrough to change the difficulty curve.

Cost is a value that is given to each individual thing you want to spawn, so each pickup, enemy, room has a cost. In order to spawn one of these things, the level generator needs to pay the cost from a pool of credits. The pool of credits is the balancing factor, and can be increased/decreased to make the game easier or more difficult. On top of this system, you then have a probability system, to further balance it. This is to make it less likely, that all the points will be used on one very difficult enemy, and an otherwise empty level. It is more interesting for the player, if the whole level is populated with objects, because it gives the player a reason to explore.

2 Final results

2.1 Finished game

The final result of the project is a finished game, free of major bugs and with a strong degree of replayability. It features a final boss level which is not randomly generated.

2.1.1 Boss level

To spice up the game and let the player experience level that is not a closed space station, a boss level was added. This level takes place in an open area created with the Terrain object of Unity.

The boss is a new enemy with custom AI and the player will need to find the strategy to beat it. If the player manages it, an ending cinematic will play.

2.1.2 Replayable

The map generator creates interesting map layouts. The room manager places items and enemies in a random fashion but inside the balance constraints. The player mechanics are solidly defined. All together makes ULTRU a replayable game.

2.1.3 3rd Party assets

Since the main focus of the game was the map generation and having a finished game. The following 3rd party assets are used in the game:

- Textures
- Hacking Terminal model
- Door model
- FPS controller
- Pistol model
- Laser shot shader
- Music and Sounds

2.2 Player mechanics

To make the game experience more interesting, the game needs additional game mechanics to walking and shooting. These game mechanics add flavor to the game, and makes the experience more special. It was decided to add these additional mechanics:

2.2.1 Hacking

In order to finish a level, the player needs to find the exit portal, but this portal will not be active, and therefore cannot be used. To activate the portal, the player needs to find a unique terminal in each level, and interact with it. This triggers all the lights in the level to turn off, leaving the level darker than before, to enforce a scary atmosphere. To further enforce this, the player is given a flashlight, with a limited cone of light, that automatically turns on after a few seconds of leaving the player in complete darkness. For an added effect, each of the laser shots have point lights, that light up the dark corridors and rooms.

These decisions were made, to force the player to look around in the level. Since both the exit portal and terminal are randomly placed, there's a big chance of one of them spawning right next to the player spawn location. Therefore the decision of having both, forces the player to explore each level.

2.2.2 Stamina

The player has a limit to how much they can run. Every time the player runs, their stamina gauge drains. If it reaches zero, they cannot run anymore. Usually in games, the stamina regenerates over time, but it was decided to not include this. Therefore stamina works as a valuable resource, that can get the player out of a sticky situation. The only way to regain stamina is by either completing the level, or by finding a health terminal, which will regain all of your stamina.

This decision was made due to the fact, that the player would otherwise be running non-stop, and because it's more interesting when the player has to plan their next move carefully. It was quickly discovered, that the player would be completely helpless without any stamina, which was not the intention. To balance this, the player's normal run speed was increased, as well as the run speed, to make it more worthwhile.

2.3 Challenges with Unity

At first Unity was chosen to be the game engine of choice for the game, based on the amount of experience in the group. As the purpose was to make something of a certain technical quality, it was decided to take a well known engine, so the project period wouldn't be spent on learning a new engine, but to research and create the project.

Unity however ended up not really supporting the mechanics of runtime level generation. Many things in Unity depends on baking, like lighting, navigation, physics, ambient sounds, occlusion culling, etc. These functions are there to increase performance in levels, and give you additional tools at your disposal. However, none of these functions can be baked at runtime. As our level is generated randomly at runtime, it was decided to either use alternatives or not use them at all.

One thing that was desperately needed, was navigation. Coding and implementing a working home-made navigational system for enemy AI, is a time consuming process, that would take a lot of the project's time. By researching already existing projects, an alternative method was found. Each level contains a big plane, that encapsulates the entire level and works as its floor. This plane has a baked navmesh into it. To then modify the navmesh, all the walls in the game has the component "Navmesh Obstacle" attached, with the parameter "Carve" set. By doing this, the area in which the enemies could move, was limited to the level.

This way however had limitations. Navmeshes cannot be moved runtime, so if more height levels were wanted, several planes needed to be added and baked, which could work, but it would be impossible to connect these navmeshes to each other, which could break the enemies.

Lighting also was a big performance hit on the game. As it was not possible to bake lights or use light probes, only realtime dynamic lights could be used. This meant that in order to make the levels look passable, the limit for dynamic lights were increased. As realistic lights were not needed a lot of places, but still used, it meant that there was a lot of unnecessary draw calls being made, that could have been avoided with baked lights.

3 Week-by-week progress

Week 44

- Pablo
  - Basic Level Generation
  - Researching different approaches to random dungeon generation
  - Spawning in random rooms
  - Room prefabs
  - Isolated rotation and placement system to align rooms
- Martin
  - Basic Level Generation
  - Logic of connecting rooms with connections
  - Failure handling of illegal dungeon creation
  - Doors and blocked doors

Week 45

- Pablo
  - Finish Map Generation
  - Interactable item system
- Martin
  - Navigation
  - Basic enemy AI
  - Gamecontroller to rule game rules
  - Framework for weapons
  - Weapon: Raygun

Week 46

- Pablo
  - Room texture placement
  - Decorations, Lights
  - Big room props
  - Terminal Shut down effect
  - Main game loop closed
- Martin
  - Player prefab
  - Weapon: Blaster
  - Adding random weapons to enemies
  - Equipping weapons to player automatically
  - Modelling turret enemy

Week 47

- Pablo
  - UI

  - Basic Hit points system
  - Sound Effects
  - Music
  - Flashlight
- Martin
  - Enemy turret behaviour AI
  - Weapon camera
  - Particles
  - Health system

Week 48

- Pablo
  - Minimap
  - Health and map terminal
- Martin
  - Explosion effects to enemies
  - Ragdolls for enemies
  - Weapon pickups

Week 49

- Pablo
  - Item spawner (Room population)
  - Title screen/Main menu
  - Boss level
  - Balance
- Martin
  - Weapon: Grenade launcher
  - Weapons and enemy balancing
  - Weapon pickup terminals
  - Weapon: Pistol
  - Stamina system to player
  - Boss enemy and AI
  - Ending cinematics
  - Player death animation