

Team Smol

蒸
発
し
ま
す



SMOL - Project Chiron

Interim Demo


23.05.2021

Mehmet Dereli

Felix Kosian

Julius Krüger

Louis Hoetzl



For the interim demo we have managed to implement a basic version of our game loop. The player can move through a procedurally generated level and shoot enemies, which shoot back. If the player dies or kills all enemies then a new level is generated. We have completed most of our goals for the functional minimum except for a few UI elements and although we do have a handmade level it would make for a poor tutorial. We hit most of our low targets as well. Our player character is still lacking a few abilities, however, and we do not have a boss enemy yet. We did implement a solid foundation for our AI that allows us to add new enemies with relative ease once we are set on their functionality.

Game Loop

Our game loop consists of the player and enemies spawning, fighting and when either all enemies or the player die a new level is generated. We handle this with a GameManager that organizes the game loop by resetting and starting a level. We use the manager to call the reset and start functions of the other managers to make sure everything is executed in the correct order. The current start order is level generation, navmesh building, AI and UI starting. Then player and enemies are placed in the level. Additionally it links independent subsystems together, for example the spawned enemies and their corresponding health bars.

Level Generation

The procedural generation of levels and their playability, is tightly linked to the player experience and therefore plays a vital role. In order to generate levels that are not only visually, but also structurally appealing, a lot of different aspects, such as complexity, density or size, come together and influence each other. None of those aspects are allowed to outweigh each other, because it would lead to a frustrating or boring player experience. If levels for example, are too big and complex, the player might find it frustrating to play those levels, but if they were too small and plain, the player might get bored. In order to balance those aspects, it is necessary to create a level generator with a lot of easily tweakable parameters, so that the generator can be adjusted rapidly during playtesting. Because this is not the first game with procedural level generation, there are a lot of different approaches to this task, such as Cellular Automata, Cave Generation or Chess Maze generation for example. These approaches however, did not appear to produce the results that we wanted, which is why we decided to take a different approach.

The levels in our game are based upon a 3D-CubeTileMap and consist of 3 different elements: The layout, the obstacles and the decoration. As of this moment, our interim demo includes 1.5 of these elements, which are the generation of the layout and a partial generation of the obstacles.

In order to reduce the computational cost of the algorithm, we are using a 2D-Array to store the information of each Grid-Cell and place the Tiles into the Grid afterwards. The level layout is generated by randomly generating different sized conjunct clusters of tiles and placing them randomly into our grid until a predefined threshold, which is either a minimum coverage of the level or a minimum number of clusters, is reached. The cluster sizes are randomly chosen from within a predefined range. Tiles that stick out of the level, can be either cutoff or kept, to allow for more choices during the playtesting. This method of generating levels however has a big flaw, and that is the possibility of disjunct spaces. This can be eliminated by choosing a small, high number range for the cluster sizes and a higher level coverage, but this will result in a lot of similar open field levels which might be boring. We decided to check if there are any such disjunct spaces by using a Flood Fill Algorithm and generate a new level if there are any such spaces. This might seem a little wasteful in regards to computational resources, but if the level generation parameters are set correctly, the chances of having disjunct spaces can be minimized.

After generating the layout, the walls are placed into the array. Because our algorithm only takes care of disjunct space and not of holes in the layout, the wall generation might place walls into the layout and thereby create obstacles. These walls will be included within the total number of obstacles that will be generated, so that the level generator does not have the possibility of creating too many obstacles.

Additional things that are generated include the PlayerSpawnPosition within the grid, as well as the EnemySpawnPoints. While the PlayerSpawnPosition is in the top-left corner of the map to ensure a certain level of consistency for the player, the EnemySpawnPoints can be anywhere on the map as long as they have predefined distances to the PlayerSpawnPosition. In order to avoid a single cluster of EnemySpawnPoints, poisson disc sampling is used. This method of generating points will also be used to create decorations and further obstacles.

One problem that occurred during the implementation process was the use of 3D Objects in combination with Unity's TileMap System. Because every object placed into the TileMap has to be a ruletile and those ruletiles do not support 3D objects, but 2D Sprites, a lot of features were not accessible for us in this project. One Problem is the use of objects that are supposed to be bigger than 1 Cube, because every object is automatically scaled to fit into that cube. If we, for example, want to use a statue as an obstacle within the level and its size is bigger than 1x1x1, we have to split the object into cube sized parts and rearrange them in the TileMap to fit them into the level. There might be other more suited solutions that we have not found yet, but the standard method of putting them into the grid does not seem to work, which is why a roundabout way of solving this problem will be needed either way.

Character

We have implemented the character's basic functionality that is required for our game to be playable. As such the character can move, aim, and shoot based on the player's input. The character also has a health bar that is reduced when the character is hit by enemy projectiles, and dies when it is depleted. We have also implemented a dash that allows the player to quickly reposition in order to dodge projectiles or flank enemies. Initially, the dash was planned as a single dash after which the ability would be unavailable for a time. This didn't feel great and now we allow the player to make a second dash directly after the first one if they feel like it. Otherwise the ability is on cooldown.

Initially, we had planned to implement control schemes for mouse and keyboard as well as for a gamepad. This turned out to be more difficult than expected. The initial assumption was that Unity's Standalone Input System would offer us an easy integration of both input methods. The first problem we encountered was that for some reason Unity's system does not recognize the Right Stick of a gamepad. There is a workaround, but we decided to de-prioritize gamepad support for now as there is another problem with it. It would be relatively straightforward to add gamepad support for the basic elements (moving, shooting, aiming) of our controller. Some of our abilities are a bit more challenging, though. In particular, the teleport ability requires the player to select a location on the map to which they want to teleport. With the mouse this is a simple point-and-click action, but with the gamepad this action might feel a lot clunkier and will probably need some work to feel acceptable. We consider other parts of the game to be more important at the moment and as such will only integrate gamepad support if we have time for it.

AI

For an intelligent enemy behaviour we need 3 parts: 1. What can the enemy do, 2. How can the enemy interact with the environment, 3. How does the enemy know what to do and when. Furthermore there is an Ai Director which manages the macro actions like spawning enemies.

1. One type of enemy has a class which contains all actions the enemy can do. Currently those actions are: Check target visibility, shoot bullet towards target, take damage, die. One extra action not in this class is: select a random position to walk towards.
2. The main interaction with the environment is movement. This is handled by generating a navmesh after the level is generated. The enemy movement is handled by a Navmesh Agent Component which can navigate the enemy as close as possible to a target destination.
3. The intelligent decision making for entities is handled by a Behaviour Tree (BT). This is a hierarchical structure which will run through it's nodes in a logical way defined by the game developers. It will be called every frame. Every node can return one of three results:

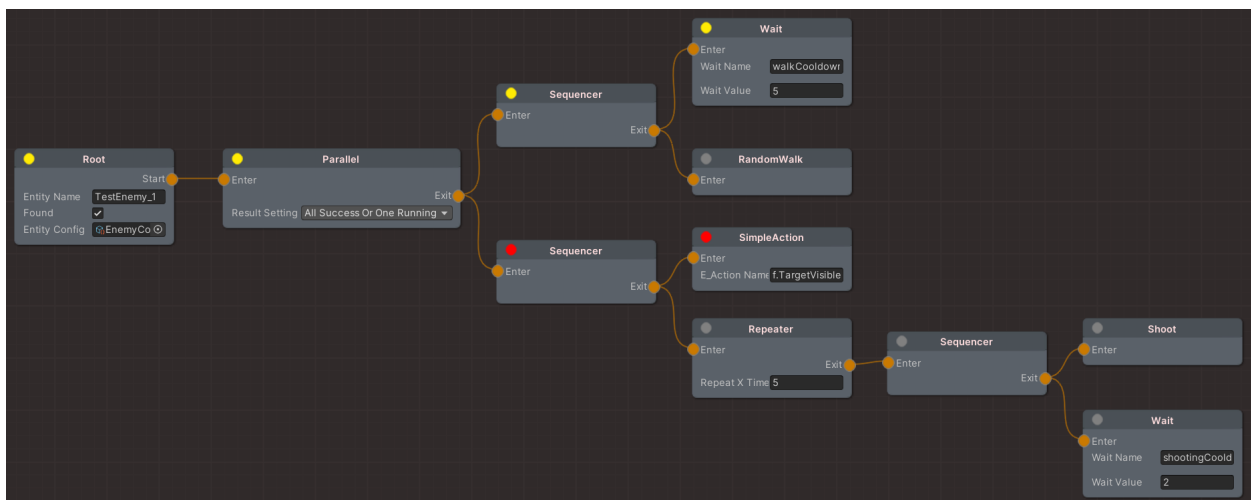
success, running, failure. Based on those results, the tree can decide which node to invoke next. Nodes for the structure of the tree are Composite Nodes or Decorator Nodes. Composite Nodes can have multiple children and decide what to do with multiple results, for example invoke the children as a sequence or in parallel. Decorator Nodes can have one child and modify it's result, for example negate it or return "running" until the child returns multiple successes. Leaf nodes are actions which correlate to the enemy action with addition of general functions like "wait for x seconds".

Each BT can have multiple entities so the invocation of the tree is context related to reduce the amount of BTs. Information necessary for the entities is saved in a "Blackboard" which is a dictionary containing parameters and actions. Parameters are pre-defined in a serializable class and loaded into the blackboard at the start of the game.

At the start of the game a blackboard gets created and pre-defined parameters from a serializable class linked to the enemy and functions defined in the enemy behaviour are loaded. Afterwards the BT is created. Every frame the BT is invoked for each enemy.

To design and manage larger BTs a visual interface is needed. This is done with XNode (XN) which provides a simple node based visual framework. This XN Tree is read at the start of the game to create the BT. Each BTNode (which handles the logic) needs a corresponding XNNode (which handles the configuration).

For live debugging each BTNode saves its result for each entity in a separate dictionary so the XNodes can read them. Additionally an extra state "inactive" was added to the BTNodes to be able to see which BTNodes were active last frame.



Example of the XN Tree in the demo

In the example image the entity "TestEnemy_1" with a specific "EnemyConfig" is loaded. In this frame the BT is still waiting before it will move again and in parallel could not view the target and therefore didn't shoot.

Another feature of this implementation is live editing of entity configuration values (like shooting cooldown) directly in the XN Graph which is not fully functional yet.

Implementation difficulties arose while changing one BT to handle multiple entities as well as having multiple entity configurations in the same BT. An additional challenge was managing editor and runtime instances. While the XN Tree is present in the Editor as well as in Runtime (with a complete reload caused by the XN framework), the BT tree is only created at Runtime.

User Interface

Currently the UI consists of the player's health bar, three ability slots and the enemy health bars and is managed by the Ui Manager. The player's health bar is linked to the player's game object in the Game Manager over the Ui Manager. The enemy health bars are created in the UI Manager after the invocation from the GameManger.

To display the correct health information, we use UniRx with Reactive Properties which follow the observer pattern. As a result the entity health is directly linked to the Ui and updated live.

Design Revisions

We made the following design revisions:

- Based on the feedback we received after the last milestone we decided to add a progression system to our game design.
- We also decided to reduce the area of the level accessible to the player over time by having level tiles break away into nothingness while the level is played. This should put an emphasis on speed. We also think this fits well into our setting. The more time the enemies spend in the level the more they corrupt it. It also acts as an interpretation of chaos, if chaos is to mean 'void'.
- The dash ability now allows for double dashes.
- We deprioritized gamepad support