Technical Design

Project: Neon Nations
Supervisor: Tijn Witsenburg (WGM06)

Authors:
Chiel Smit s1110788
Benji Riegman s1101306
Stella Grandvoinet s1139688
Nicander Mohrmann s1103064
David Ceccarelli s1139670

University of Applied Sciences Windesheim
ICT Software Engineering – Games Programming

Version: 4.0
Date: 11-06-2019

Table of Content

High-Level Overview

Looking from a bird's eye perspective, there are five major components. The most important one would of course be Unity, since the game is made in Unity and uses a lot of methods provided by Unity. Next, there would be code written by the team. The code is written in C#, because this works the best with the Unity engine. To add multiplayer to the game, the project uses Mirror. This is a variation on Unity's multiplayer service, but it works faster and uses less lines of code. It's still uses the foundation of the Unity multiplayer service, so that's why it uses Unity as well. And of course, since the game works with local multiplayer, it also uses a network to which the players can connect. Finally, there is the Post Processing component. This isn't a major part of the game, but it gives a nice effect to the lights in Unity. With all these components combined, Neon Nations is created.

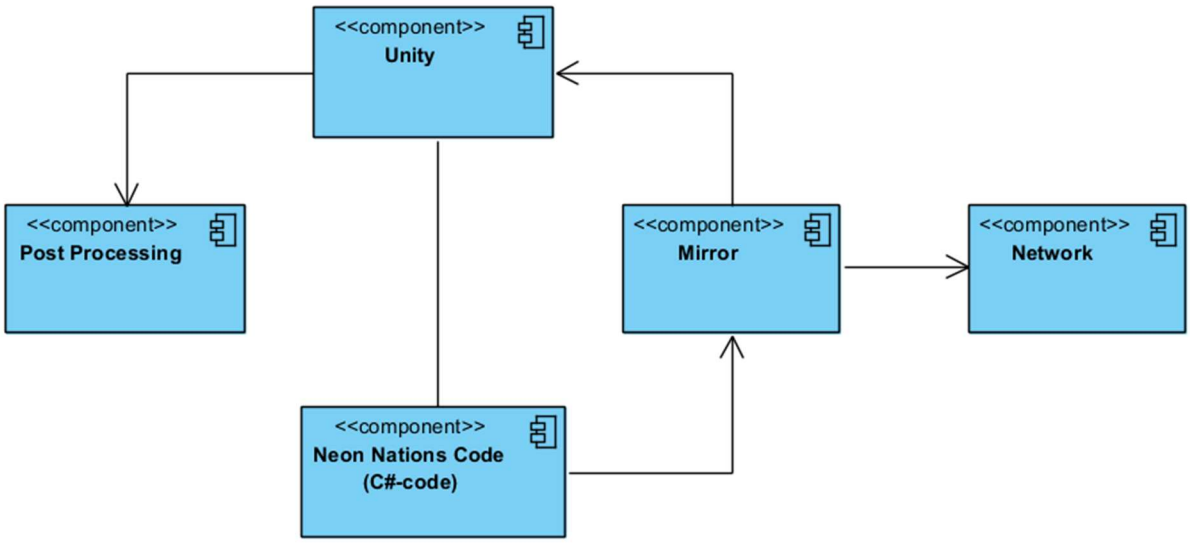
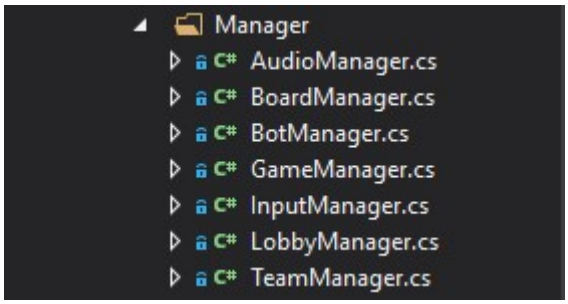


Figure 1, The high-level overview of the game.

Managing the game

To manage the game, certain managers are used. These managers are all singletons, since only one of each is supposed to exist. The GameManager is the most import one to run the game. It mainly concerns itself with loading the game. This includes loading a map, loading the players and loading the teams. It also handles the final win or lose screen for the game.



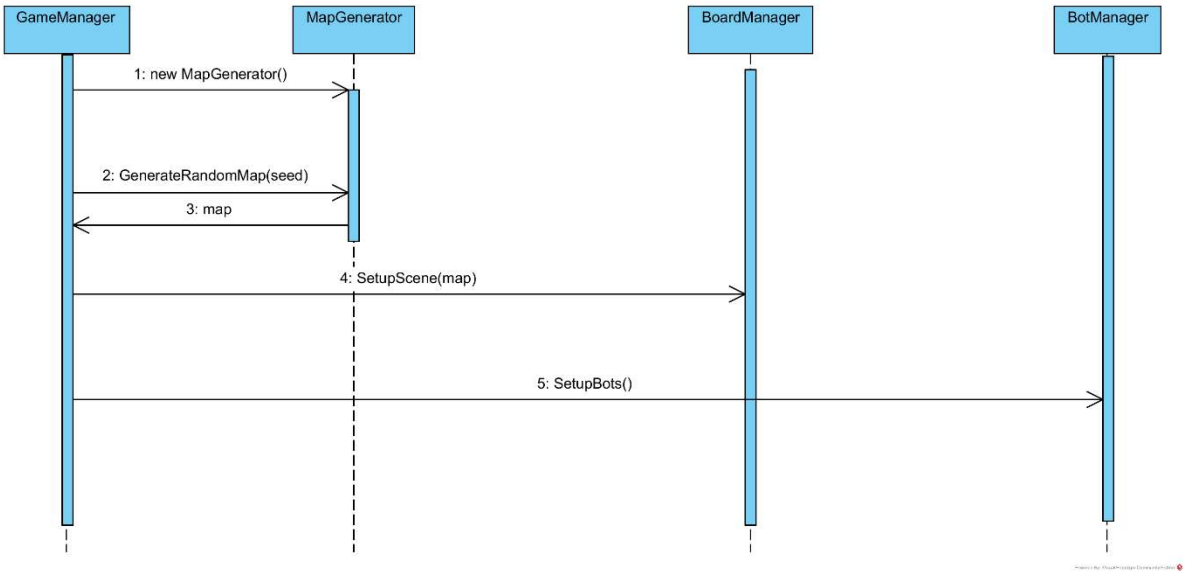


Figure 2, InitGame() of GameManager

The BoardManager is used to load the map into the game, more on that later. BotManager is only used to create the bots at the beginning of the game. After that, the bots manage themselves. LobbyManager is used to control the lobby before the game starts. It handles the players joining and readying. TeamManager adds and removes players from the team. It also makes sure the teams are synced nicely. InputManager is used to edit and save the key bindings. AudioManager is used for all the sounds in the game.

Soldier

The entire game is centered around finding and killing your opponents in the dark, so having an easy to work with main character is a must. To make players the super class Soldier is used that can either be real players or bots. They both use the same code for functionalities such as shooting and using the sonar, but use different code to determine when this should happen. The player gets controlled from the PlayerController, this looks at the keystrokes and inputs from hardware and handles them by calling the action script. The bot works mostly the same by calling the action script, but this is done by the various behaviours that are contained in the BotController. The super class Soldier handles general attributes like health, energy and score. Things that should only happen in the update for bot or player are overridden in the respective implementations.

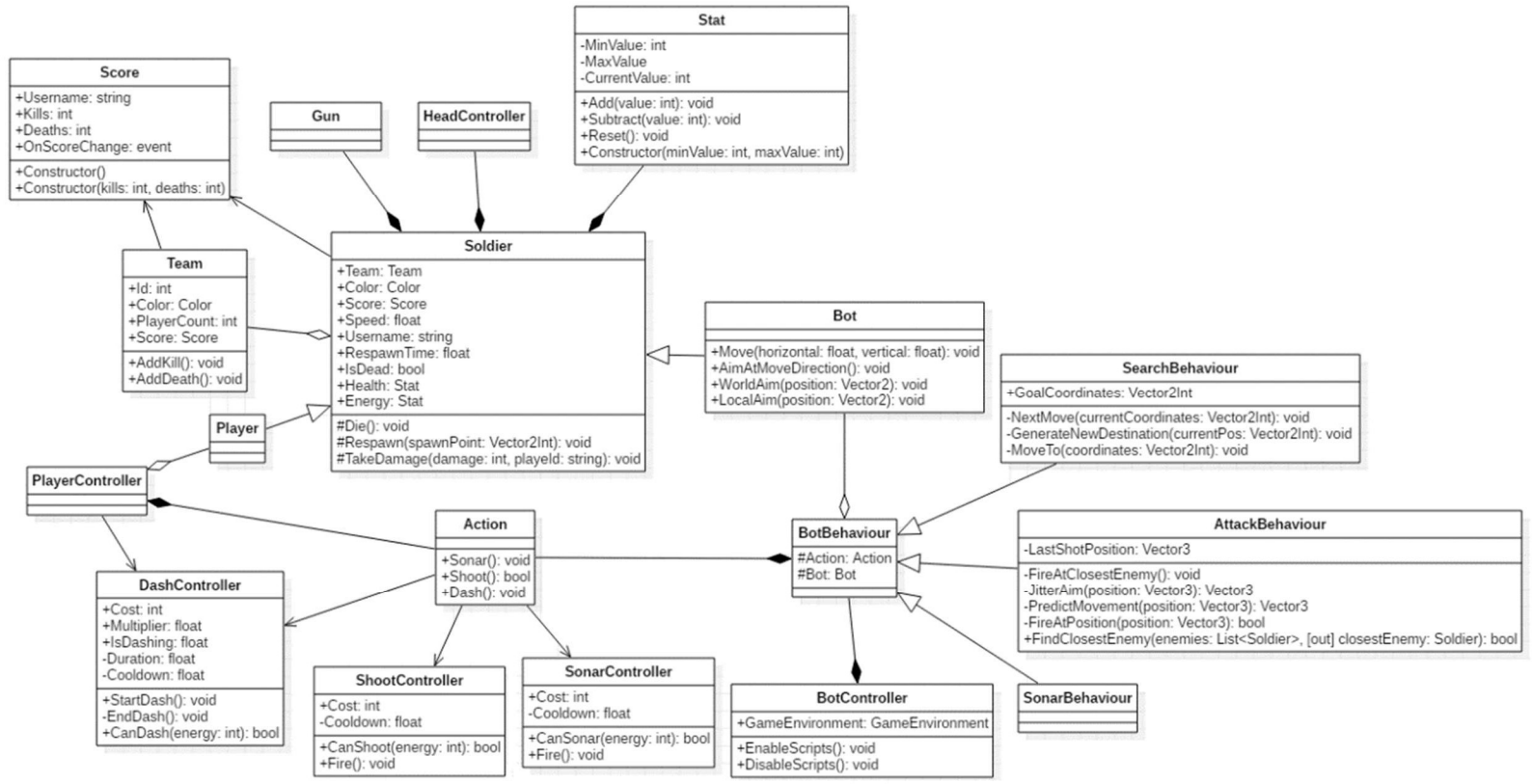


Figure 3, Class diagram soldier

Random map generation

The algorithm that is used is based on the level generation technique used in the game Brogue. The algorithm doesn't have an official name, but on the internet it's usually called broguelike or room addition algorithm, because of how the algorithm works. The map generation is being done in MapGenerator.cs.

Generate Random Map

```
public Tile[][] GenerateRandomMap(string seed)
    {
1       // Check if settings can be used
        CheckSettings();

2       // Create level with only walls
        m_map = Map.GenerateEmptyMap(Tile.Wall, m_mapWidth, m_mapHeight,
m_spawnAreaMinSize);

3       // Change seed of randomizer
        UnityEngine.Random.InitState(seed.GetHashCode());

4       // Generate first room (in the middle) and add it to the map
        Room room = GenerateRandomRoom();
        room.Pos = new Vector2Int(m_mapWidth / 2 - room.TileMap.Length / 2,
m_mapHeight / 2 - room.TileMap[0].Length / 2);
        AddRoom(room);

5       int currentRoomAmount = 1;
        int currentBuildAttempt = 0;
        while (currentBuildAttempt < MAX_BUILD_ATTEMPTS && currentRoomAmount <
m_maxRoomAmount)
        {
            // Generate a room
            room = GenerateRandomRoom();
            // Try to place the room
            bool placed = PlaceRoom(room);
            // If succeeded
            if (placed)
            {
                currentRoomAmount++;
            }
            currentBuildAttempt++;
        }

6       // Add shortcuts
        AddShortcuts();

7       // Add reflectors
        AddReflectors();

8       // Set seed to random again
        UnityEngine.Random.InitState((int)DateTime.Now.Ticks);

        return m_tileMap;
    }
```

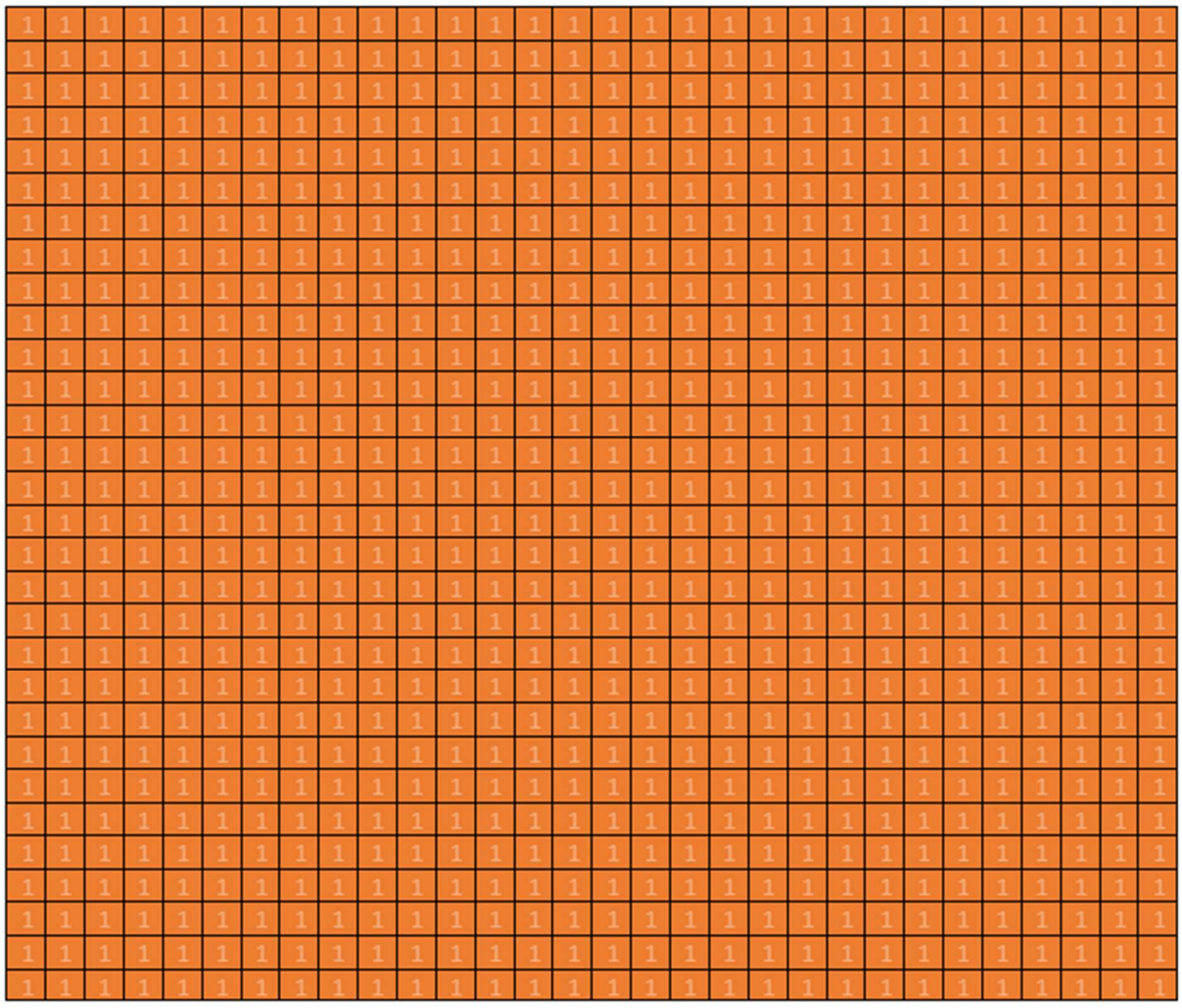
Code Fragment 1

GenerateRandomMap is the main method of the generation of a random map. It consists of eight steps:

1. Check the settings of the generation
2. Generate a map consisting of only walls
3. Set the randomizer to use the seed given
4. Generate the first room and put it in the middle of the map
5. Keep generating new rooms and add them to the map until either the max amount of rooms has been reached, or the max number of tries has been reached.
6. Add shortcuts to the map
7. Add reflectors to the map
8. Set the randomizer to use the tick value again

Step 1 is to check if all the settings are correct. This means that minimum values can't be larger than maximum values, some minimum values can't be smaller than 0, map size can't be too small or too large, and so forth. When a setting like that is encountered, it changes the value and logs the change.

GenerateEmptyMap(1) sets the m_map within the MapGenerator class to a Map object with a jagged array with all 1's. The 1's are walls. An example of such a map would be:



Tilemap 1, map after step 2 in GenerateRandomMap()

The tile in the lower left corner is (0,0) to conform to how Unity does it's coordinate system.

After this, the randomizer is set to use the seed. Seeds are being used for 2 reasons: debugging and saving fun maps. This seed is used throughout all of the map generation. After the generation of the map, the randomizer gets set to its normal randomized seed.

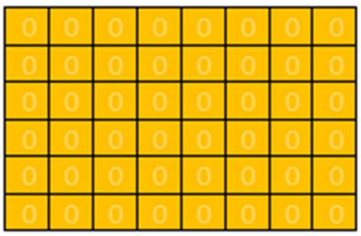
After the instantiation of the tilemap, the first room will be generated, and added to the map. To generate a room the method GenerateRandomRoom is used.

```
private Room GenerateRandomRoom()
    {
        int width = UnityEngine.Random.Range(m_minRoomLength, m_maxRoomSize / m_min-
RoomLength);
        int height = UnityEngine.Random.Range(m_minRoomLength, m_maxRoomSize / width);

        Tile[][] roomMap = new Tile[width][];
        for (int x = 0; x < width; x++)
        {
            roomMap[x] = new Tile[height];
        }
        return new Room(roomMap);
    }
```

Code Fragment 2

It generates a random width and height based on the settings, and creates a new tilemap with it. It returns a Map object with the tilemap in it. The Map object is an object to primarily save the tilemap and the position of the map. It also includes lots of useful methods like pasting a map inside another map, checking out of bounds, and generating a spawn point. At this point the position is unknown, so it's not set. The tilemap of this room could look something like this:



Tilemap 2

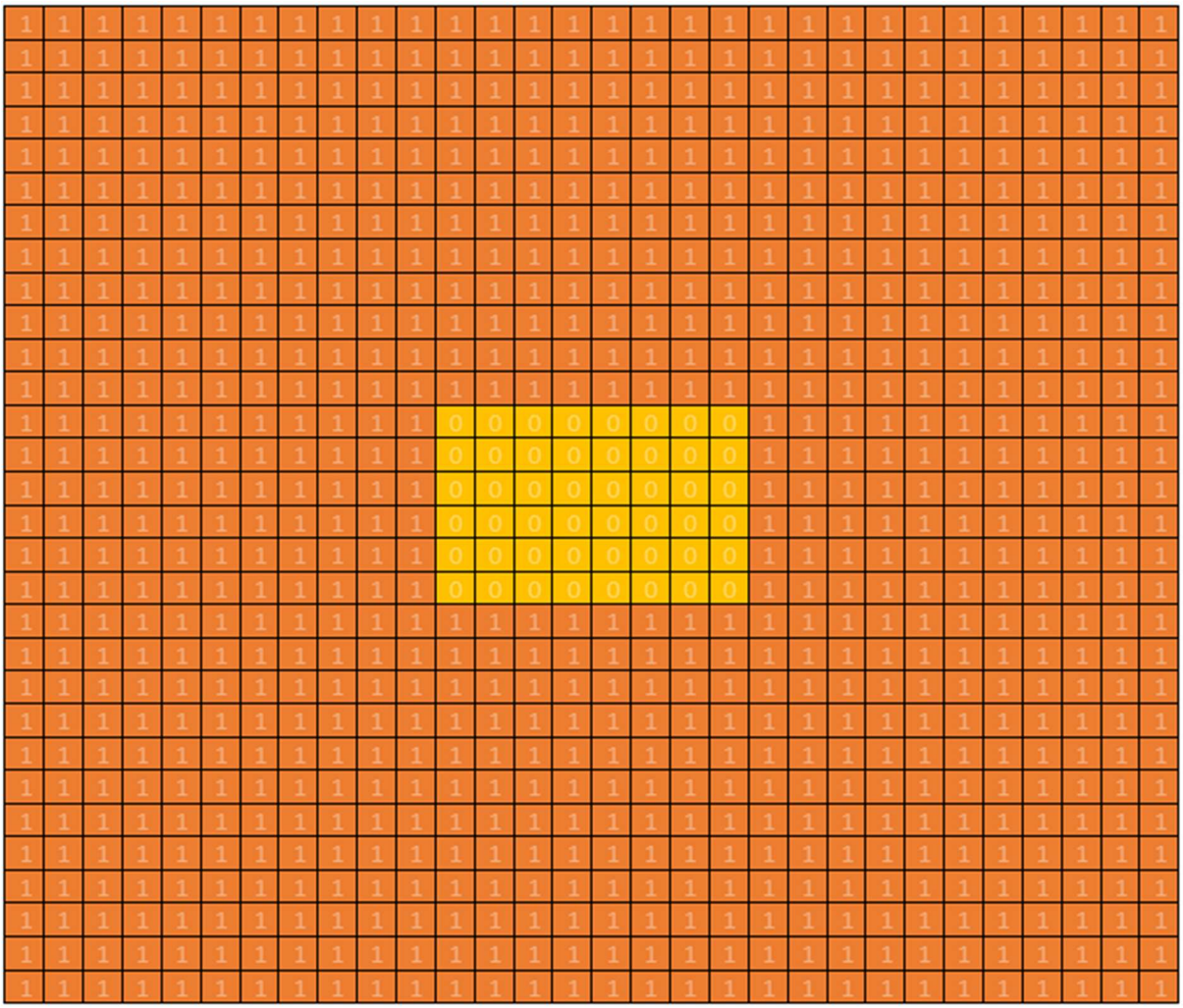
After this, the position of this first room will be calculated. The first room will always be in the middle. In this example, the position of this room within the 30 by 30 map would be (11,12). When this is done, the room is added to the map by the AddRoom method.

```
private void AddRoom(Room room)
    {
        m_map.PasteTileMap(room.RoomMap, m_tileMap, room.Pos);
    }

public void PasteTileMap(Vector2Int pos, Tile[][] smallMap)
    {
        // for every tile in the small map
        for (int x = 0; x < smallMap.Length; x++)
        {
            for (int y = 0; y < smallMap[0].Length; y++)
            {
                // change the according tile on the big map
                if (smallMap[x][y] != Tile.Wall)
                {
                    TileMap[x + pos.x][y + pos.y] = smallMap[x][y];
                }
            }
        }
    }
```

Code Fragment 3

AddRoom is just used to simplify adding a Room to m_map. PasteTileMap is used to paste the contents of a small map onto the map the method is being called on, at the given position. After this has been done in the example, the map looks like this:



Tilemap 3, map after step 4 in GenerateRandomMap()

After this, GenerateRandomRoom goes into a while-loop, which it only exits if the amount of rooms is sufficient, or the max number of tries has been reached. Within this loop it generates a room and tries to place the room. To place the room, PlaceRoom is used.

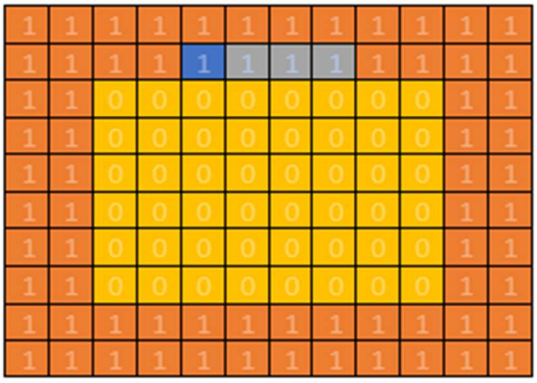
```
private bool PlaceRoom(Map room)
    {
        for (int i = 1; i <= MAX_PLACE_ATTEMPTS; i++)
        {
            Map tryMap = TryPlacement(room);
            if (tryMap != null)
            {
                // add room to the map
                AddRoom(tryMap);
                return true;
            }
        }
        return false;
    }
```

Code Fragment 4

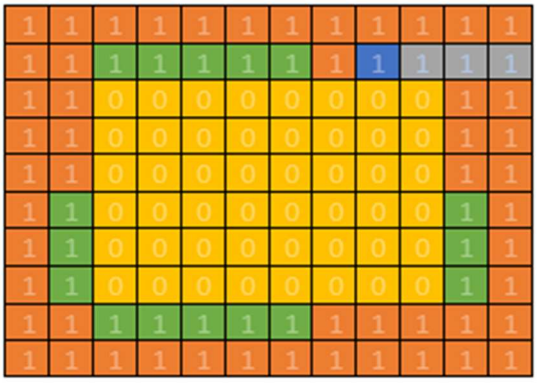
This method tries to place the room for a certain amount of tries. If this succeeds within the limit, the room gets added to the map. If not, it goes back to the while loop in GenerateRandomMap to generate another sized room.

Try Placement

This method is used to try and place a room in the map. It starts by generating a random direction for the corridor to attach to the map. It then selects a random wall tile in that direction. This takes into account the width of the corridor. When the corridor is placed, the width of the tunnel always goes to the right or up. In the example below to the left, if the blue tile has been selected as wall tile, the width of the corridor expends to the right until it's the correct width. While selecting a wall tile, it takes this into consideration, so that the example on the right does not happen. The green tiles indicate the selectable wall tiles for every direction (if the corridor width is 4).

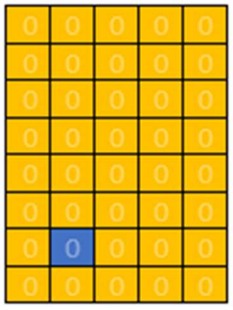


Tilemap 4

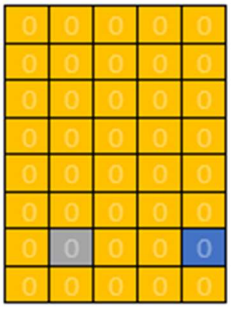


Tilemap 5, not possible

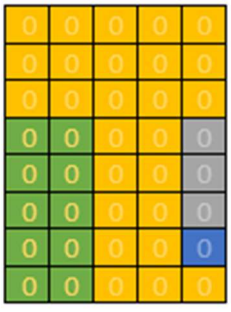
After that is done, an entrance tile is selected within the room that needs to be added to the map. It does this by selecting a random tile in the room and then moving the tile to the correct wall. In this case the width also expands to the right or up, which is being considered when selecting a random tile. In the example below, the direction of the corridor is left, seen from the map. This means the corridor will enter the room at the right side. The green tiles indicate the selectable tiles for all directions (if the corridor width is 4).



Tilemap 6

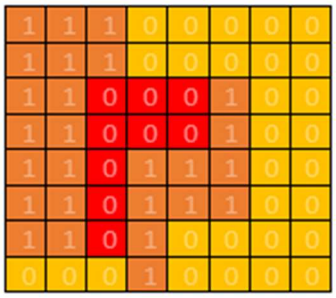


Tilemap 7

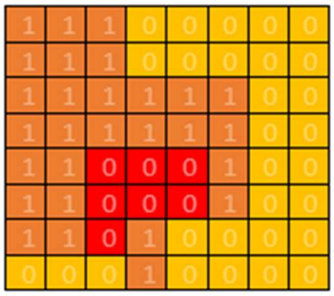


Tilemap 8

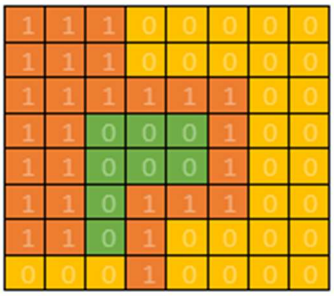
When this is done, the position of the room is determined within the map. It uses the previously generated wall tile and entrance tile. This position assumes that the room is placed directly next to the room it's attached to. If the position is (5,0), then it indicates that the tile in the lower left corner of the room will be at position (5,0) within the big map. To place the room in the map, a corridor is needed to connect the rooms. The corridors have a maximum length and a minimum length. When the room has to be placed, it goes through every possible tunnel length between the lengths until it finds a good spot.



Tilemap 9, doesn't fit

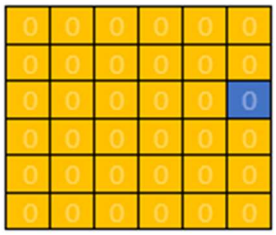


Tilemap 10, doesn't fit

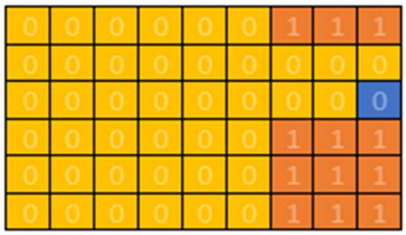


Tilemap 11, fits!

To make sure it doesn't always take the shortest possible corridor, the list of possible lengths gets randomized, and then a for loop goes through them. Before the example above can be done however, there are still a few steps that need to be done. Step 1 is to update the tilemap to add a tunnel. Step 2 is to update the position on the map and the entrances of the room. This will only be necessary if the direction is left or down.

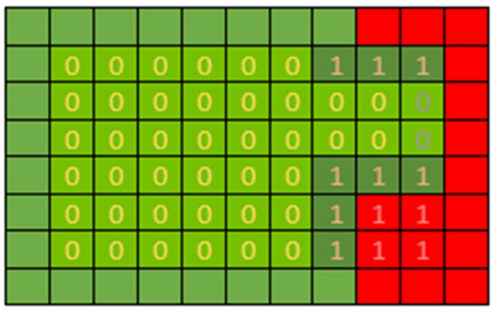


Tilemap 12, position on map is (5,0)



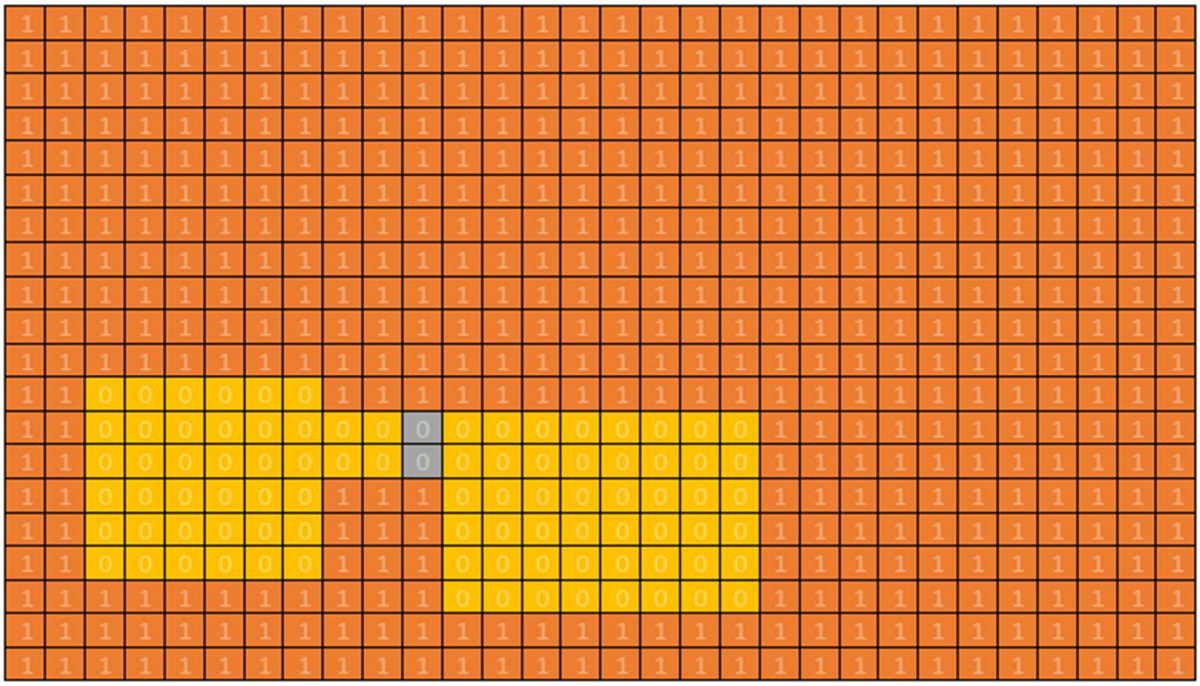
Tilemap 13, position on map now is (2,0)

After this has been done, the rest of the entrance tiles will be calculated. This is necessary for the algorithm to determine if a room can be placed. CanPlace is used for this. It first checks if the room isn't out of bounds. After that it checks if the room fits in the map. Every floortile within the map is checked. It checks if the same spot on the map is a wall, and the tiles around it are walls. The exception to this are the before mentioned entrance tiles. These only check the tiles perpendicular to the direction of the corridor. Given the tilemap of the last example, these will be the tiles checked in m_map.

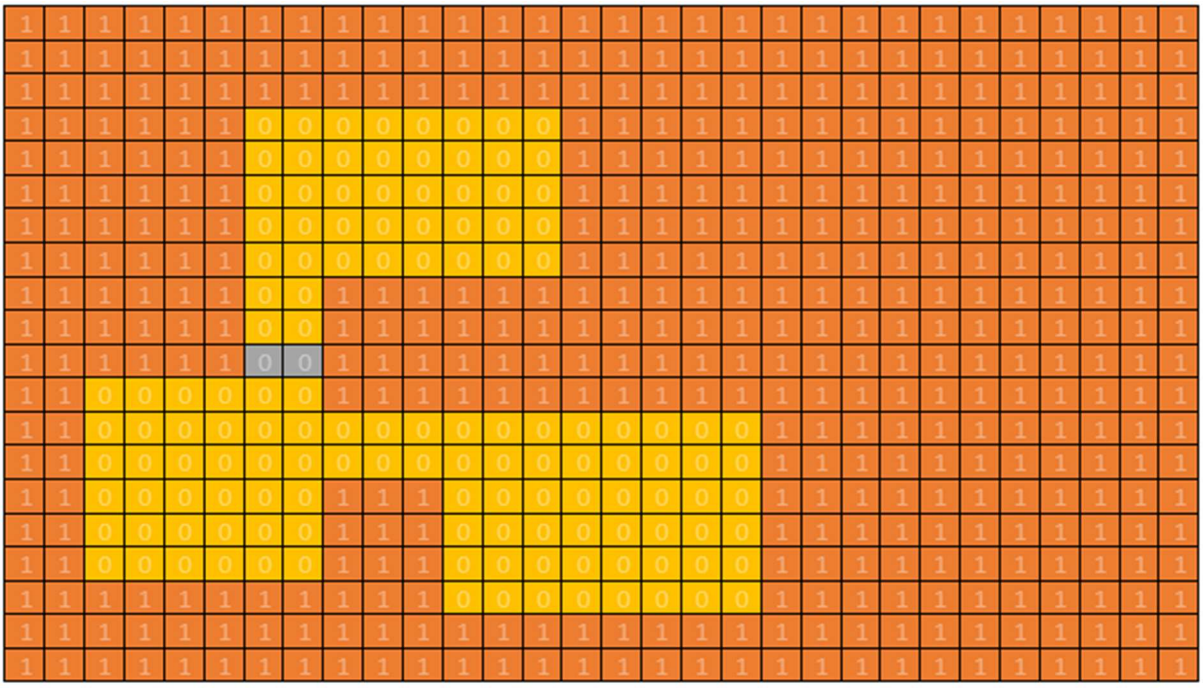


Tilemap 14

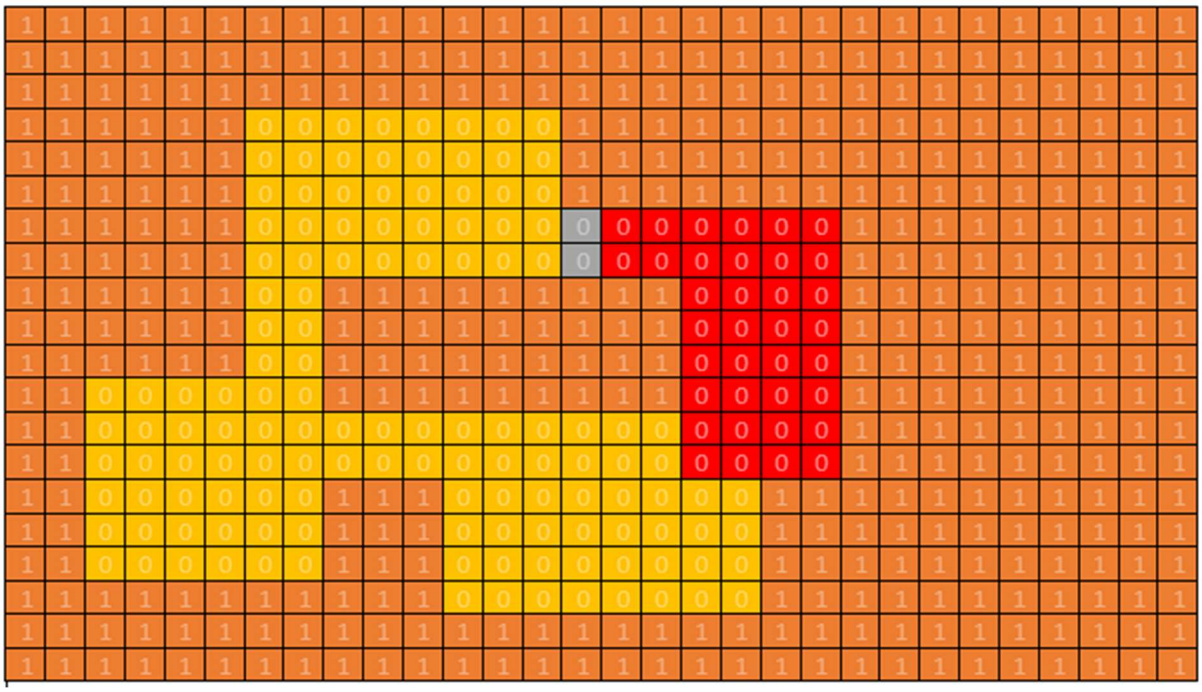
When one of those tiles isn't a wall in m_map, the method tries another corridor length. If the room fits, the method returns the room with the corridor attached and goes back to PlaceRoom, where it will add the room to the map, and return true. This then goes back to GenerateRandomRoom where it adds 1 to the room counter, and it repeats itself again. This is what it could look like (to save space, the lower 10 rows won't be shown).



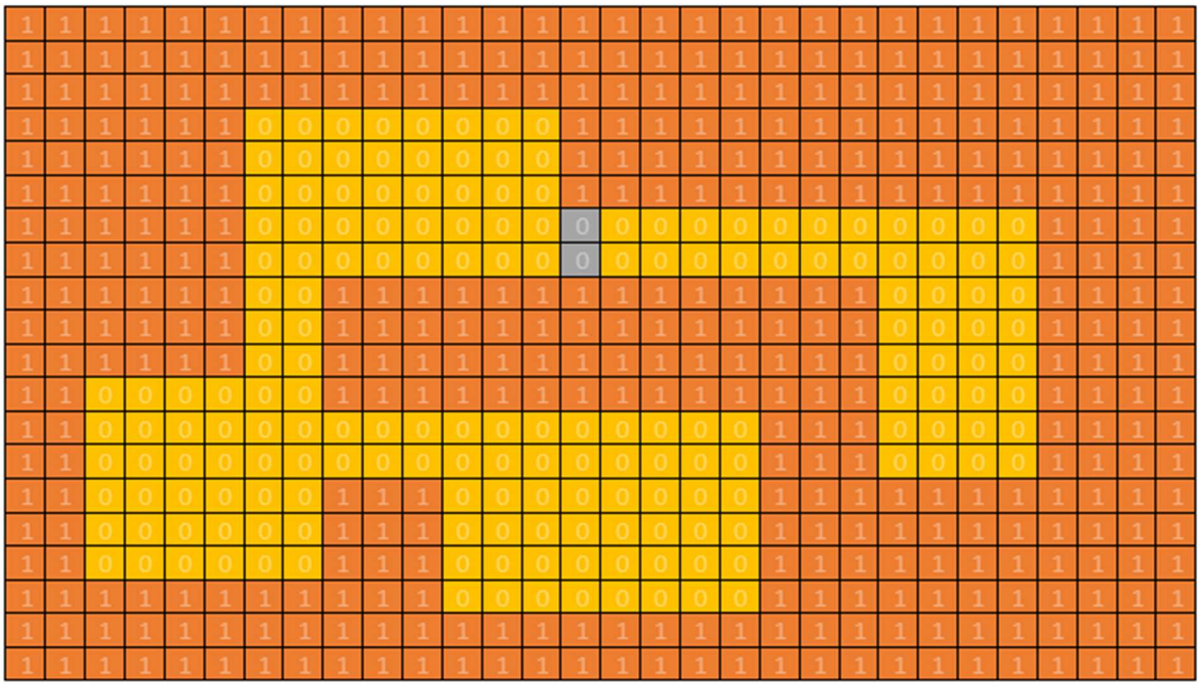
Tilemap 15, CanPlace() returns true



Tilemap 16, CanPlace() returns true

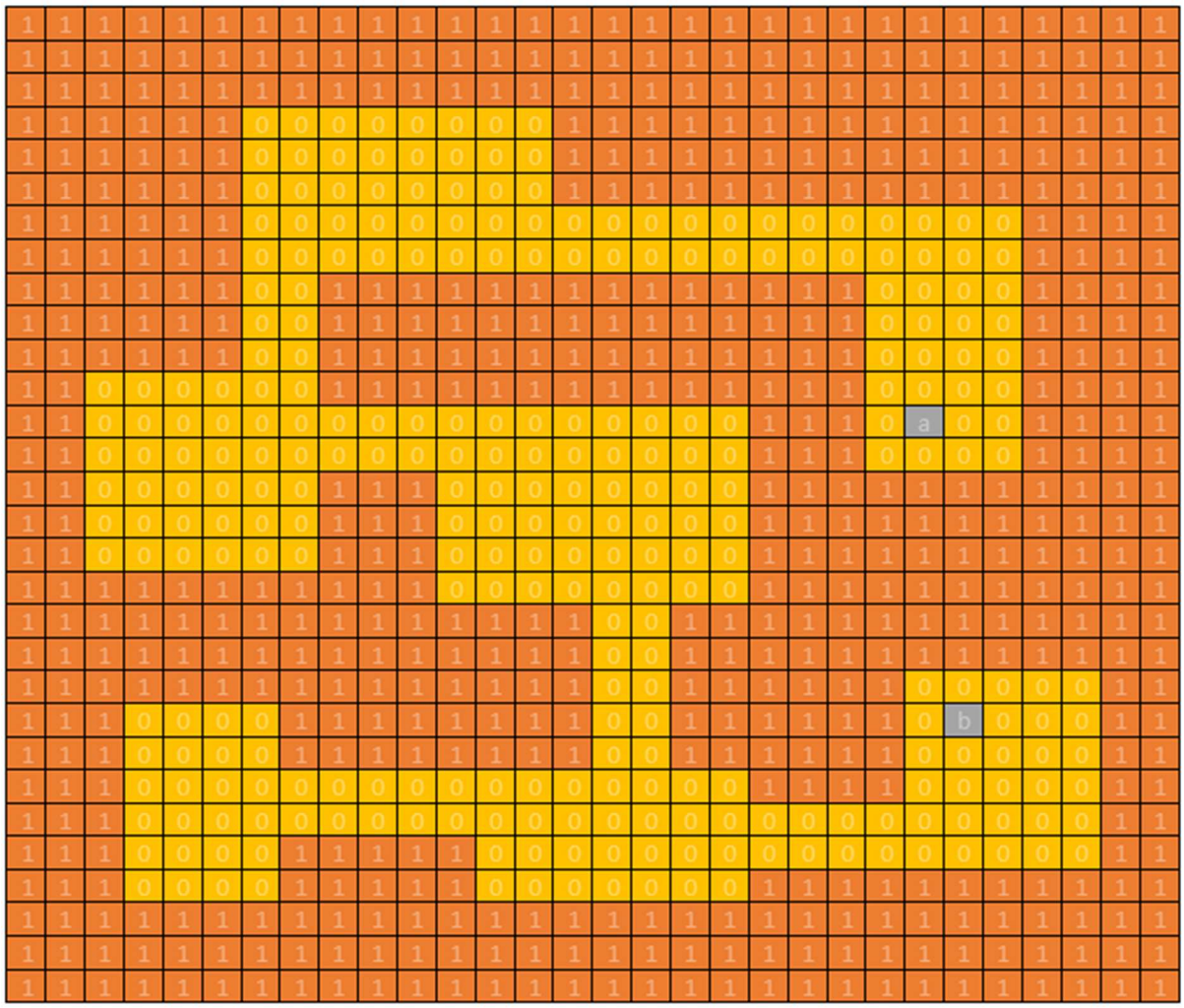


Tilemap 17, CanPlace() returns false



Tilemap 18, CanPlace() returns true

After this while loop in GenerateRandomMap(), the map is built like a tree, where the center room is the root node. This makes it so that there may need to be a lot of backtracking if you want to go from a to b (see below).

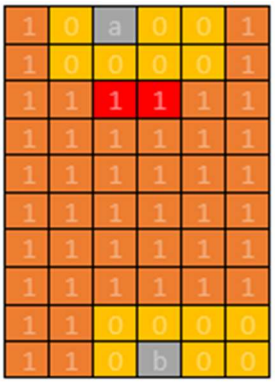


Tilemap 19, map after step 4 in GenerateRandomMap()

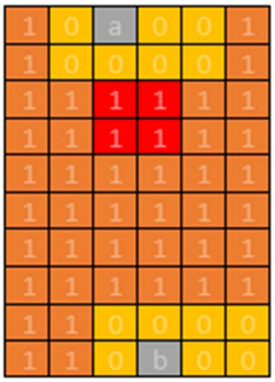
Add Shortcuts

This is where step 6 comes in, AddShortcuts(). It's used to punch corridors through walls, to make the map more open, and to add the ability to flank. It immediately enters a while loop, which it only exits if the number of shortcuts has been reached, or no shortcut can be placed anymore.

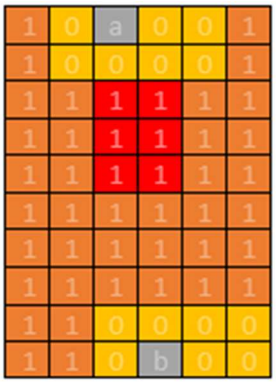
In the while loop, it first gets all the wall tiles and their direction. After that, it enters a for loop which goes through all the wall tiles, and then a for loop is entered where it checks for every possible corridor length if it's possible to place a corridor there. If not, it goes to the next wall tile.



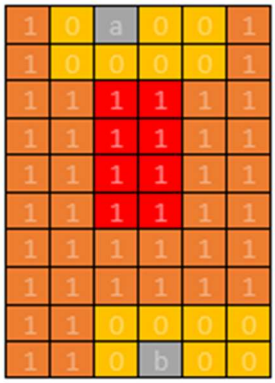
Tilemap 20



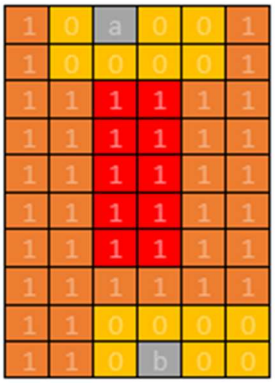
Tilemap 21



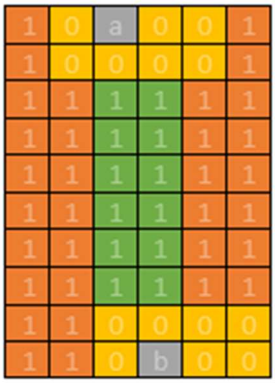
Tilemap 22



Tilemap 23

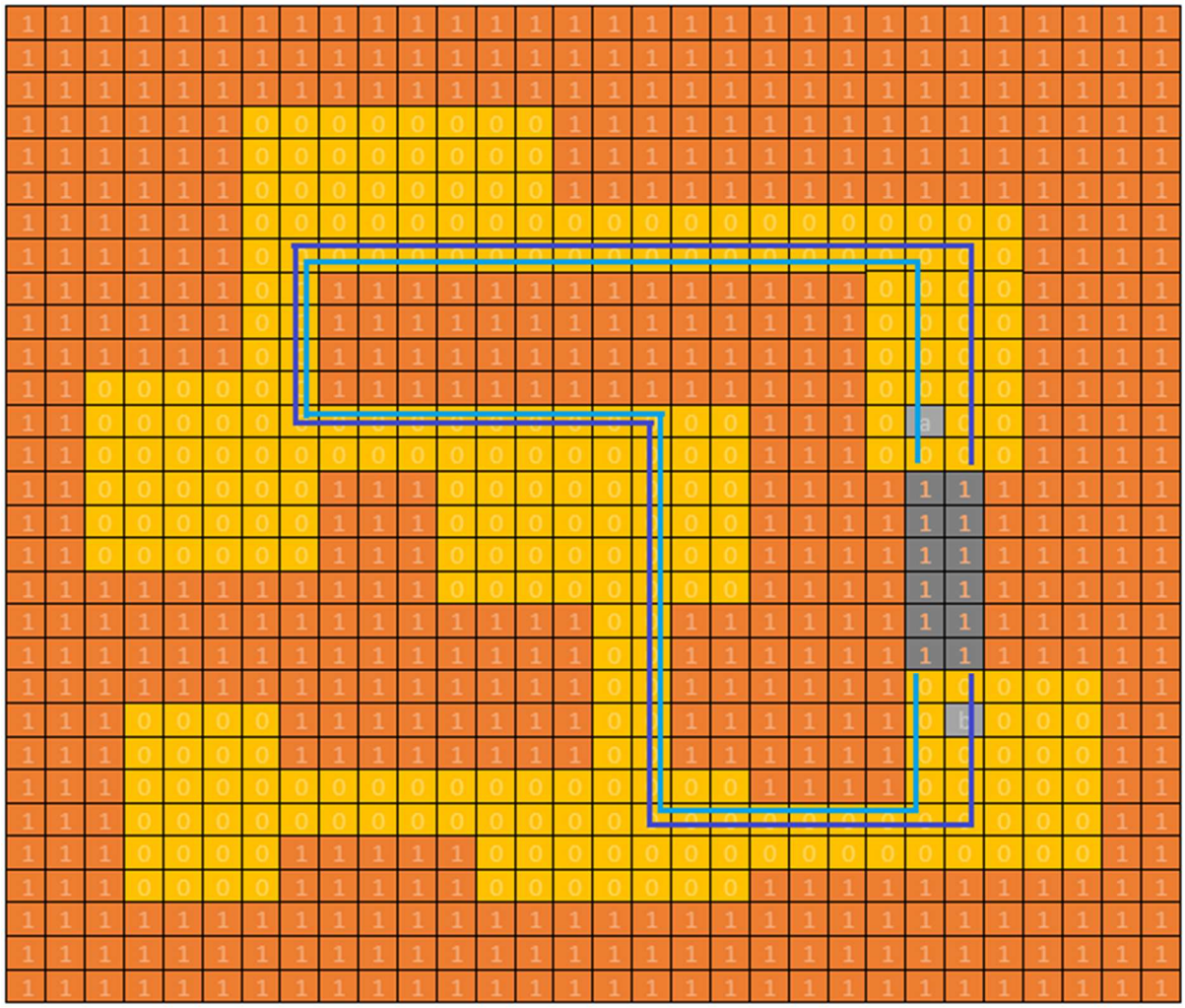


Tilemap 24



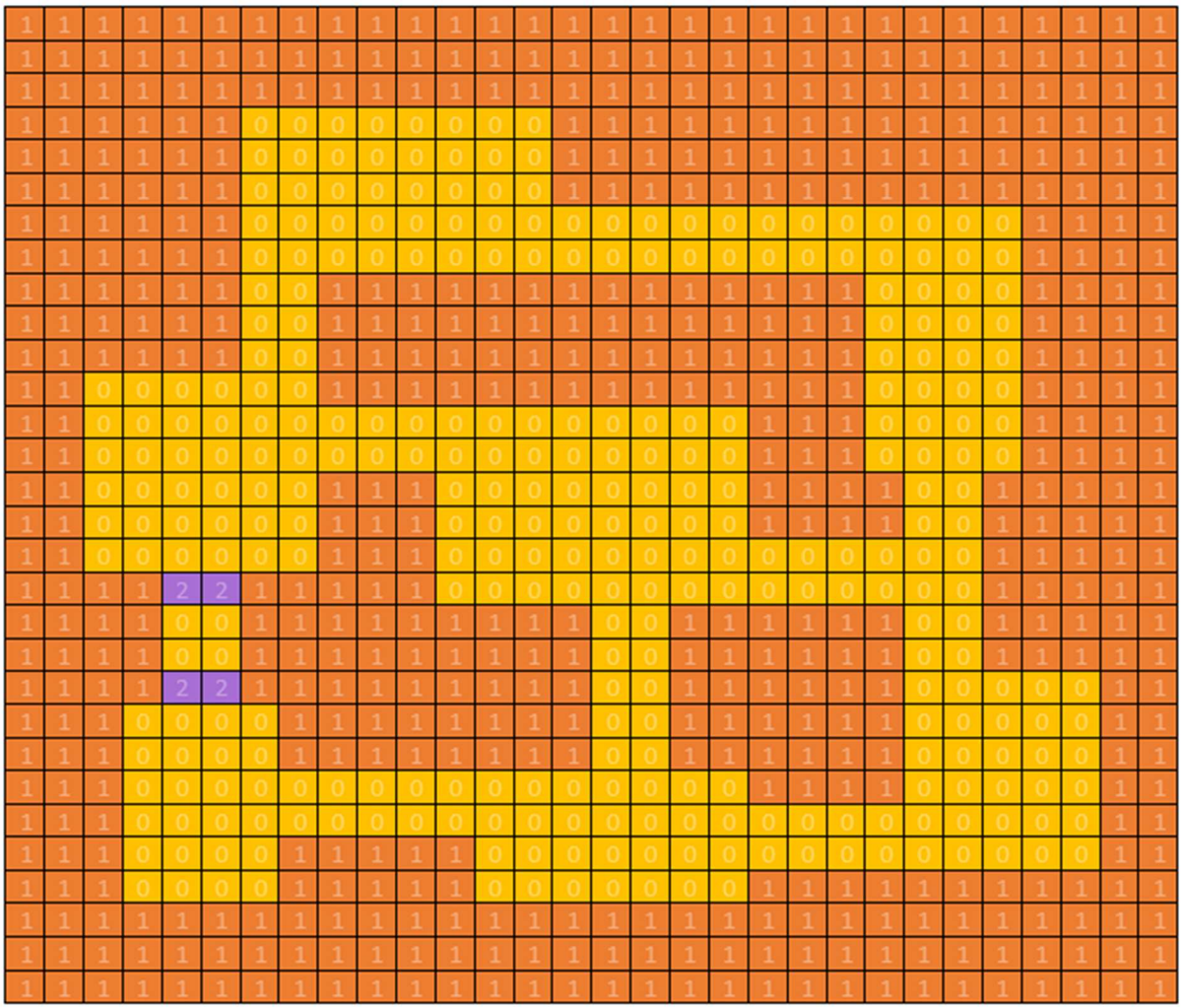
Tilemap 25

If it is possible, the tilemap for the tunnel is created and the position of said tunnel within the map is calculated. It then calculates the entrance tiles and determines if it can be placed, just like with adding the rooms. If it can be placed, it determines if it is useful to place a shortcut there. To do this, DStarLite is used from one side of the potential shortcut to the other side. If the shortest distance is above a certain value, a shortcut will be placed. The distance is calculated twice, from the position of the tunnel and from the position of the tunnel + tunnel width.



Tilemap 26, distance calculated from two sides

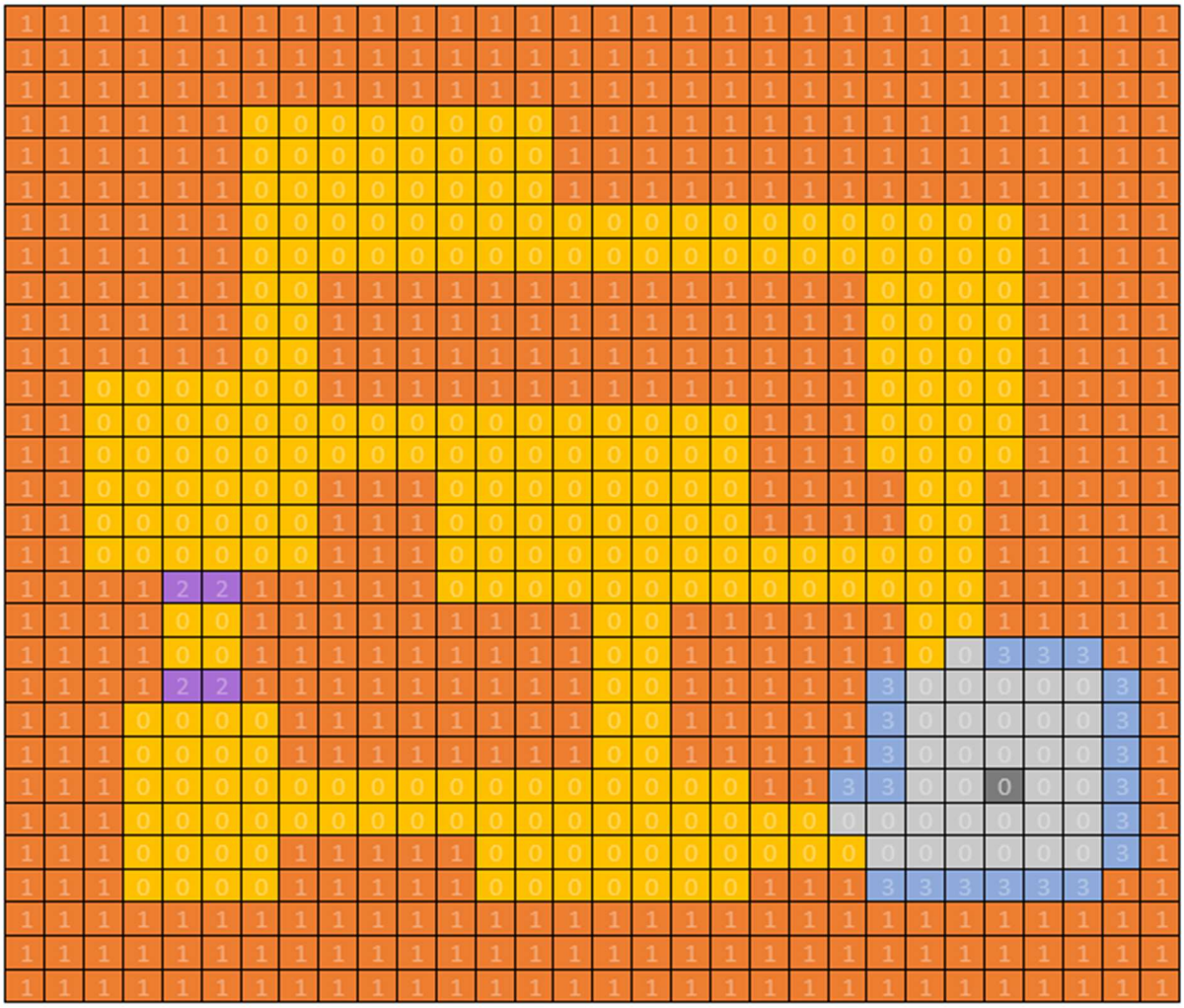
After this it determines whether this corridor is open or secret. If it's a secret corridor, the entrance tiles will be converted to 2's which stand for breakable walls. It then pastes the tunnel in the map, after which it repeats. The reason the wall tiles are gotten in the while loop, is because the algorithm can then add shortcuts to the wall created by the previous shortcuts. If no shortcut has been placed after going through all the wall tiles, it is impossible to add any more shortcuts, after which the method returns. The example map now looks like this:



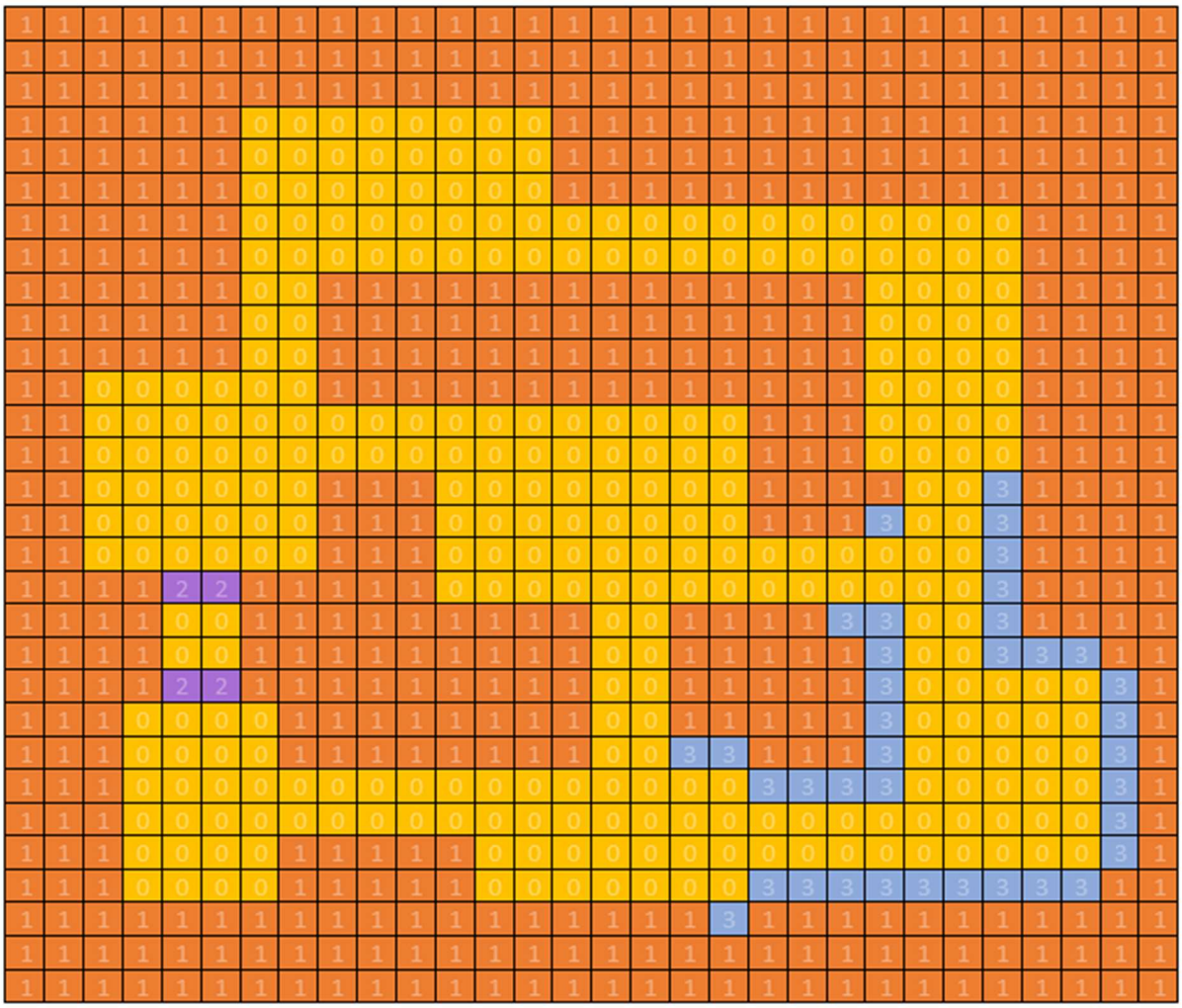
Tilemap 27, map after shortcuts have been placed

Add reflectors

Now the reflectors are added to the map. Reflectors are special wall tiles, that reflect a bullet that hits it, instead of destroying the bullet like a normal wall. To add the reflectors to the map, a floodfill algorithm is used. A random floor tile on the map is picked, and from there the algorithm floods. Whenever a wall is encountered, it is converted to a reflector. The filling is stopped after a certain amount of floor tiles have been filled.



Tilemap 28, filling the map with reflectors



Tilemap 29, final map

Map loading

Map loading happens within the boardmanager, once the gamemanager calls for it. The public method SetupScene() handles this.

```
public void SetupScene(Tile[][] tileMap)
    {
        m_tileMap = tileMap;
        m_map = new GameObject();
        m_map.name = "Map";

        LoadFloor();
        LoadMap();
        CreateOuterWalls();
        CombineAllMeshes();
    }
```

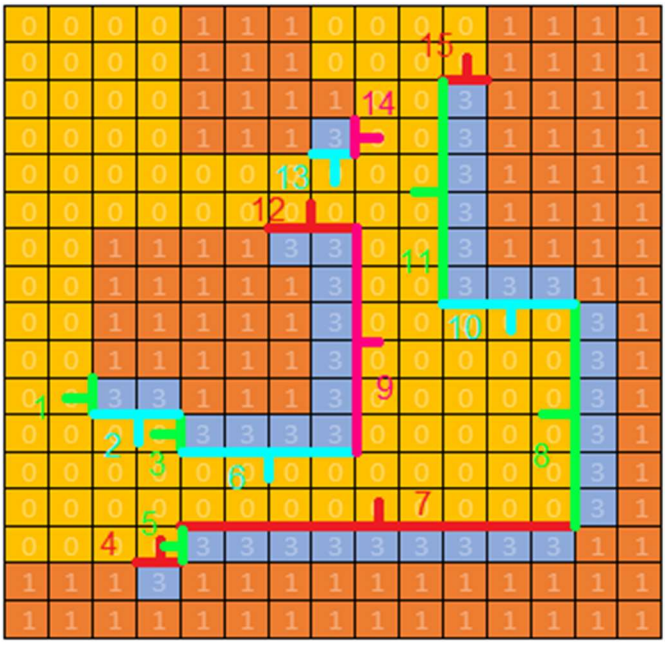
Code Fragment 5

It sets the local tilemap to the tilemap given. It then starts by loading the floor. The floor is just a simple plane, taking into account the size of the tilemap and the width of the outer wall.

After this is done, LoadMap() is called. This method is used to convert the content of the tilemap into a unity map. It goes through every tile, and handles differently depending on the type of tile. If the tile is Tile.Floor, it does nothing. If the tile is Tile.Wall, it creates a wall. A wall is a primitive cube, with an altered mesh. The reason the mesh gets altered is because most of the times, only 1 side of the cube is needed (the top one). Only if it's next to a floortile, other sides are needed. To generate the new mesh, GenerateNewMesh is used. It first determines what vertices and triangles are needed to correctly show the wall. It then creates a new mesh with those vertices and triangles, and then recalculates the normal. When it's done with recalculating the mesh, the position within Unity is calculated, and the wall is then added to the map. If the tile is Tile.BreakableWall and the one loading the map is the server, it spawns a Breakable wall and adds it to the network. If the client is loading the map, it does nothing on Tile.BreakableWall, it just receives these walls from the server.

If the tile is Tile.Reflector, an algorithm is used to make sure there are only a few reflector objects. It checks if the tiles next to the reflector are also reflectors, and it increases the size of the reflector object based on the number of reflectors next to it. This makes it so that the example map generated only has 15 reflectors, instead of 45.

Next is CreateOuterWalls. This simply adds a thick layer of walls to the map, so that a player can't see into the abyss next to the map. These walls only need to have the top side of the mesh.



Tilemap 30, all reflectors made

This all results in a really large list of objects. To reduce the amount of objects, CombineAllMeshes is used. It combines all the meshes into a few objects. It takes all the MeshFilters within the map, except for the breakable walls and reflectors, and adds them to a list. The reason the breakable walls and reflectors aren't added, is because they should have separate meshes to check if bullets hit. For every MeshFilter, add to a combiner list and increase the amount of vertices currently in the list. The reason the amount of vertices is tracked, is because a mesh can't contain more than 65535 vertices. If there are more than that, the shader will fail. This algorithm uses a limit of 30000 vertices per mesh, since meshes too large will affect performance.

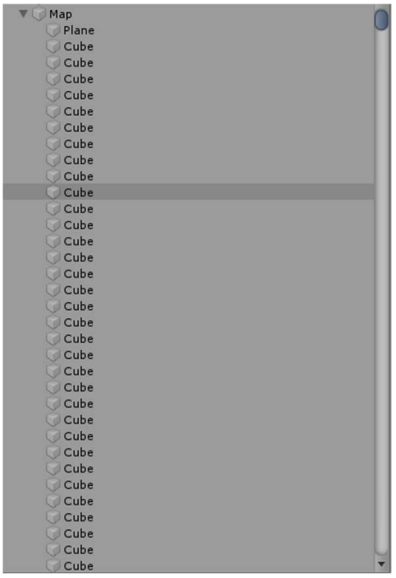


Figure 4, list of map elements without combining them

Whenever the vertex amount reaches 30000, or when the last MeshFilter has been combined, CreateCombinedMesh is called. It takes every mesh currently in the combiner list, and creates one mesh from them. It then puts the mesh in a new object called MapPart. It also updates the meshcollider of the map part to accommodate the new mesh. The map part then gets added to a list. When it's done with all the meshes, it deletes every object from the map except the breakable walls and the reflectors, and adds the map parts to the map. This makes sure that only the map parts and the interactable parts are in the map.

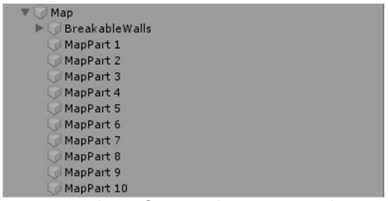


Figure 5, list of map elements with CombineAllMeshes

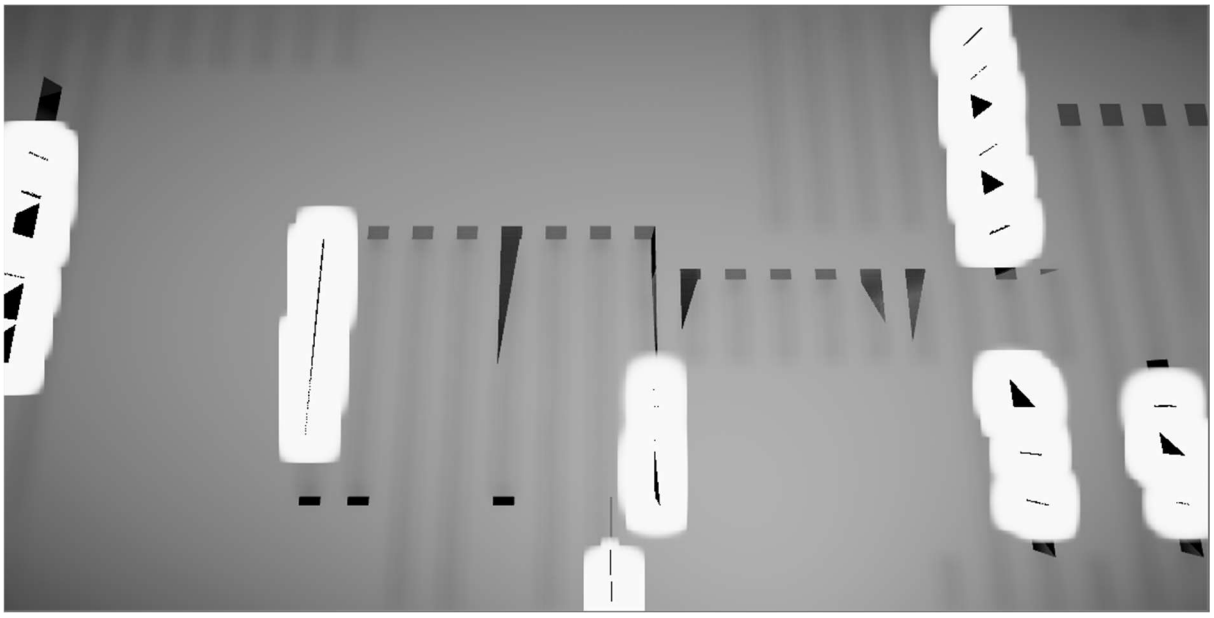


Figure 6, result of too many vertices in a mesh

Multiplayer

First choice

The Unity Engine uses UNet to implement online multiplayer, a network layer made by the Unity team specifically for Unity. This means that it has been optimized for the Unity Engine and should be easy to work with. Unfortunately, this functionality is deprecated and is removed in Unity 2019. However, we decided to start working with UNet and migrate to another layer later because there was no clear replacement for it in Unity 2019.

Replacement

We had different options to replace the deprecated UNet layer.

New Unity networking layer

Link: https://support.unity3d.com/hc/en-us/articles/360001252086-UNet-Deprecation-FAQ?_ga=2.103879683.2035367031.1558000420-61829319.1557688008

"*The HLAPI (High Level API) and LLAPI (Low Level API) will be replaced with a new networking layer, with development focused first on a fast and minimal transport layer. The P2P Relay service will be replaced with a Game Server Hosting service, enabling server-authoritative games. A new matchmaking service will replace the legacy Matchmaker and work seamlessly with the Game Server Hosting service.*"

Unity will no longer support P2P Relay service and will provide a paid service to host games on dedicated servers. However, we want to keep the P2P functionality to allow players to easily create and join server without having to create or buy dedicated server.

Photon

Link: https://www.photonengine.com/en/PUN

"Photon Realtime is a fully managed service (SaaS) of Photon on-premises servers running in regions worldwide."

This multiplayer engine is not only designed for Unity but also for PlayStation 4, Android, Unreal Engine and more. This solution also does not include P2P technology, but it provides free servers. The disadvantage with Photon is that the API is completely different from the one we currently use, and the game is too advanced to entirely change the Network API.

Mirror

Link: https://github.com/vis2k/Mirror

"A community replacement for Unity's abandoned UNet Networking System"

Mirror has the same API and works the same as the one we were previously using, it is free, open-source, frequently updated, has an excellent supporting developers that can be reached through their forums or Mirror's Discord server and is used and tested for Massive Multiplayer Online (MMO) scale networking. Therefore, we chose Mirror to replace our current system.

Server and client

Definitions

- **Server**
  A server is an instance of the game which all other players connect to when they want to play together. A server often manages various aspects of the game, such as keeping score, and transmit that data back to the client.

- **Clients**
  Clients are instances of the game that usually connect from different computers to the server. Clients can connect over a local network, or online.

- **Host server**
  When there is no dedicated server, one of the clients also plays the role of the server. This client is the "host server". The host server creates a single instance of the game (called the host), which acts as both server and client.

Variables

- **isServer**
  True if the GameObject is on a server (or host) and has been spawned.
- **isClient**
  True if the GameObject is on a client and was created by the server.
- **isLocalPlayer**
  True if the GameObject is a player GameObject for this client.
- **netId**
  Unique identifier for this network session, assigned when spawned.

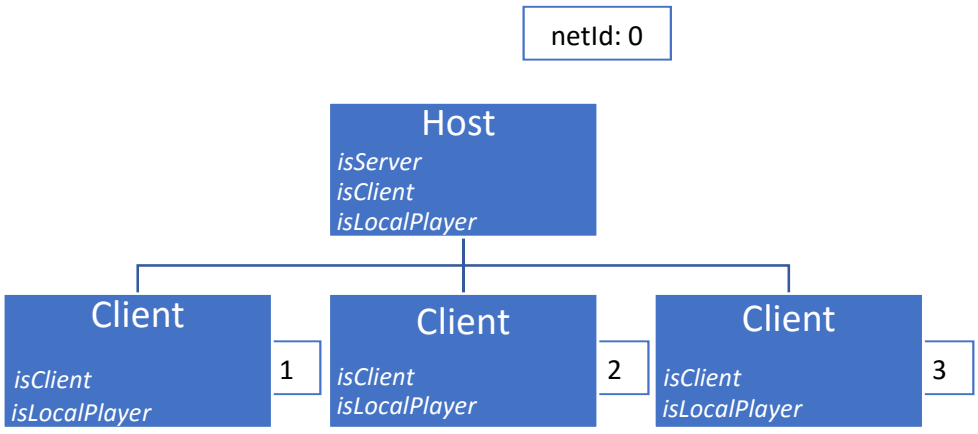


Figure 7, illustration of variables across network.

Remote Actions

- **Command**
  Commands are sent from player objects on the client to player objects on the server.

```
[Command]
void CmdMethod() {
      DoSomethingServerSide();
}
```

- **ClientRpc Calls**
  ClientRpc calls are sent from objects on the server to objects on clients. They can be sent from any server object with a NetworkIdentity that has been spawned.

```
[ClientRpc]
void RpcMethod() {
      DoSomethingClientSide();
}
```

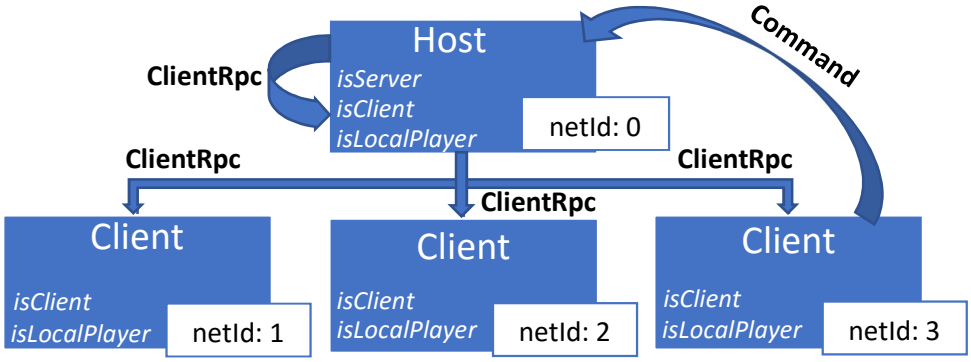


Figure 7, network-methods

Scenario

1. Client 3 calls *CmdMethod().*
2. Host receives a command and executes *CmdMethod()* on its side then calls *RpcMethod().*
3. Client 0, 1, 2 and 3 receive a command and execute *RpcMethod()* on their sides.

This scenario is used to synchronize players' position and rotation, bullet, etc. between all clients.

Transport

Mirror is a high-level Networking Library that can use several different low-level transports. There are seven transport libraries that are compatible with Mirror but only two of them were interesting for our use case:

- TCP - Telepathy
  Simple, message based, MMO Scale TCP networking in C#.
- UDP - Ignorance
  Ignorance implements a reliable and unreliable sequenced UDP transport based on ENet.

Quick summary of differences between UDP and TCP (source):

- TCP is suited for applications that require high reliability, and transmission time is relatively less critical.
- UDP is suitable for applications that need fast, efficient transmission, such as games. UDP's stateless nature is also useful for servers that answer small queries from huge numbers of clients.

Unity uses UDP while Mirror is using TPC (Telepathy) by default. Their choice is explained here. However, our game needs to quickly send data and does not particularly need to receive them ordered or to be sure that it correctly received them. We therefore chose Ignorance.

Network Discovery

Like described in the Unity Manual:

“The Network Discovery component allows Unity multiplayer games to find each other on a local area network (a LAN). This means your players don’t have to find out the IP address of the host to connect to a game on a LAN. Network Discovery doesn’t work over the internet, only on local networks.”

The Network Discovery has two modes:

- When in **client mode**, the component listens for broadcast messages on the specified port. When a message is received, this means that a game is available to join on the local network.
- When in **server mode**, the Network Discovery component sends broadcast messages over the network on the port specified in the inspector. This mode is used if a game is being hosted on that machine.

Search Behaviour and D* Lite

In the SearchBehaviour the bot will search for a specific target in the map. This target could either be an enemy that it wants to attack, or a random position so it can explore the map some more. To check if there is an enemy nearby, the behaviour will call the GetIlluminatedEnemies method from GameEnvironment. This method will return a List of enemies the bot is able to see. If this List is not empty, the target will be set to the closest enemy in this List.

In the case where the List does not contain any enemies, the target will not change until the bot has reached the target. This target could either be the last position where the enemy has been seen, or the random position the bot wants to explorer. Once the bot has reached this position, it will try to fire it's sonar. This will give the bot more information about its current room and when chasing an enemy, it might be able to find that enemy again using the sonar.

The navigation of the bot towards this target is controlled in the NextMove method in SearchBehaviour. This method will start by asking the D* Lite algorithm for the shortest path. After it received this shortest path, it will send this path to the path-smoothing algorithm. This algorithm will return the farthest location on the path where the bot could move to in a straight line. At last, the search behaviour will normalize this position and will call the Move method of the bot with the normalized value.

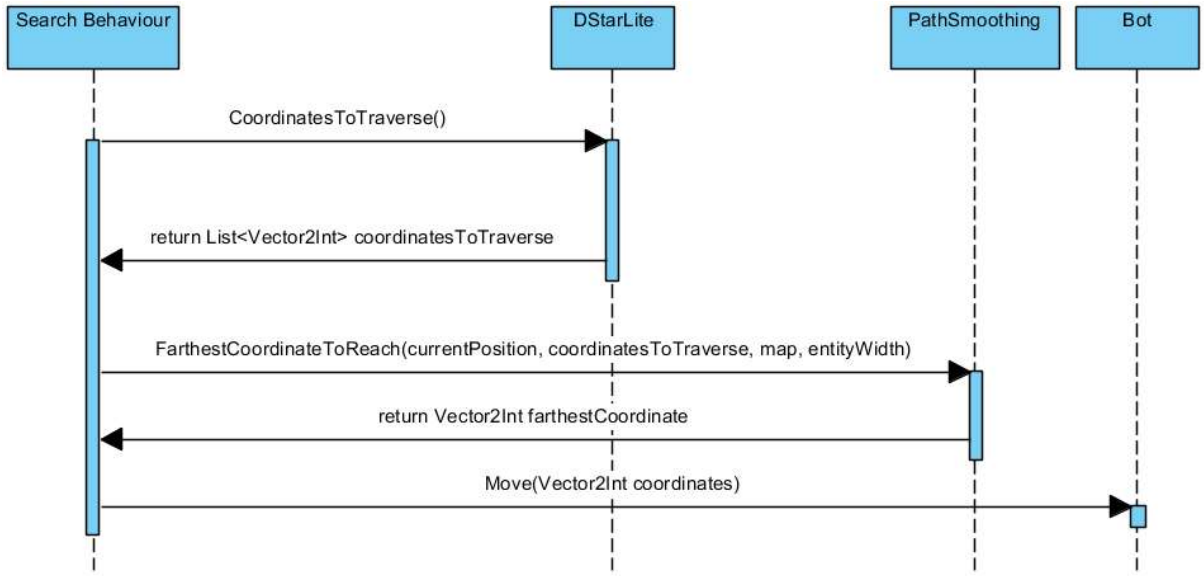


Figure 8, Sequence diagram for Search Behaviour

D* Lite

Because exploration is a big part in the game, the bots should explore the map as well. So instead of already knowing the whole layout and simply using an A*-algorithm to navigate through the map, the bots will use a D*-algorithm. A D*-algorithm works a bit like A*, but for unknown areas. This means that the algorithm will try to follow the shortest path and once it encounters an obstacle, it will update its map and recalculates the shortest path. This way, the bots in the game will start with zero knowledge of their surroundings making it fairer for the players.

In general, there are three types of D*-algorithms. Original D*, Focused D* and D* Lite. Like the name suggests, Original D* was the first of its kind. It is an informed incremental algorithm, but because the newer versions like Focused D* and D* Lite are faster, it isn't used that often anymore.

Focused D*, is the first improvement to the original D*. There aren't that many changes to it, but it uses the heuristics of an A* algorithm to filter which nodes should be recalculated after a map-change has occurred. Because it will only recalculate the nodes that matter for reaching the goal, it is much faster than the original D*-algorithm.

The algorithm used in the game is D* Lite. D* Lite isn't based on both Focused D* or the original D*, but it does implement the same behaviour. It can be implemented with fewer lines of code and works faster. Therefore, the name D* Lite. The big difference between D* Lite and the other two, is that D* Lite is based on the Lifelong Planning A* algorithm (LPA*). LPA* is an improvement of A*. The main difference between LPA* and A* is that LPA* uses a lookahead value named rhs. This value is based on the g-values of the surrounding nodes and adds the cost for traveling from that node to the current node to this value. The lowest of these calculated values will most likely be the g-value of the current node. This way the value of the node can be calculated without having to traverse the whole path again. D* Lite is used over the original D* and the Focused D* since it is faster and therefore enhancing the performance of the game.

To begin the algorithm, the SearchBehaviour will reset the DStarLite with the new start- and goal-positions of the bot. Here it will reset its g- and rhs-values and calculate the first shortest path from the bot till the goal.

After this, the SearchBehaviour will call the CoordinatesToTraverse method from DStarLite. This will return a list of coordinates the bot will have to traverse. The algorithm will first check if anything has changed in the map. To do this, it will start by asking the GameEnvironment to return the illuminated coordinates in range of the bot. It does this by checking the positions of the point-lights in the game and tag the surrounding coordinates. Next, it will compare the values of those coordinates to its own map. If those values are identical, it will just return the previous shortest path since nothing has changed. If one of these values did change, it will update its map and reevaluate the affected nodes. After the nodes are re-evaluated, it will compute the new shortest path and return this to the SearchBehaviour.

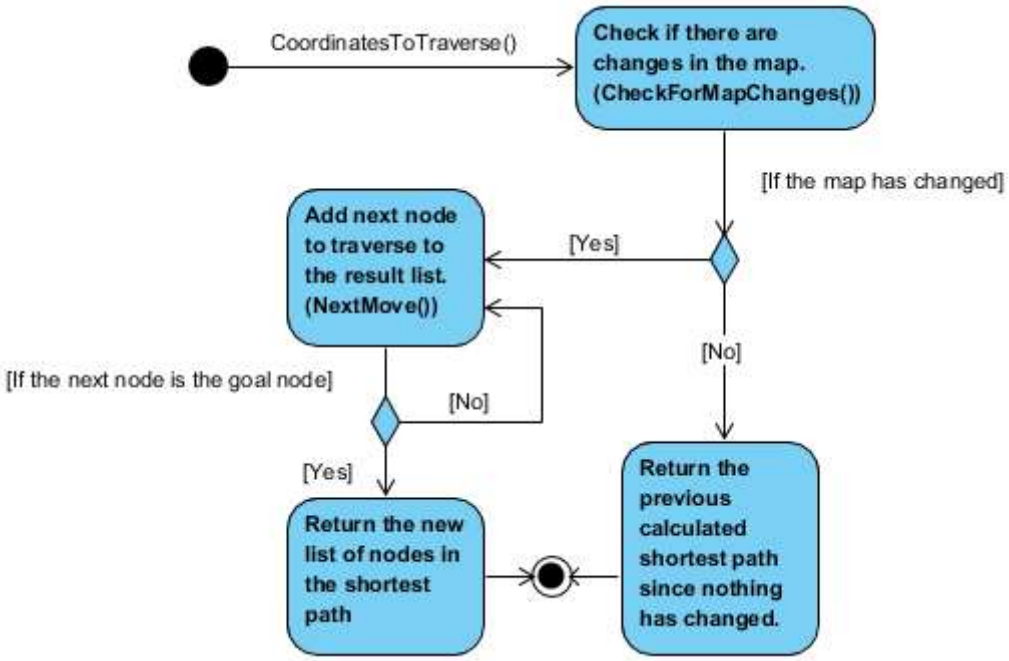


Figure 9, Activity diagram for D Lite*

Path Smoothing

After the SearchBehaviour has received the shortest path, it will call the FarthestCoordinateToReach method from PathSmoothing. This method will return the farthest coordinate the bot can reach in a straight line from the list of coordinates to traverse. The algorithm will loop through the coordinates to traverse and check if the nodes are reachable from the bot's current position. To improve the performance, the algorithm will first skip all the nodes that are on the same x- or the same y-position as the bot. Because those nodes are either already calculated by the D* Lite-algorithm, or the loop will not reach these nodes because it will find an unreachable node before these nodes. (As seen in figure 8.)

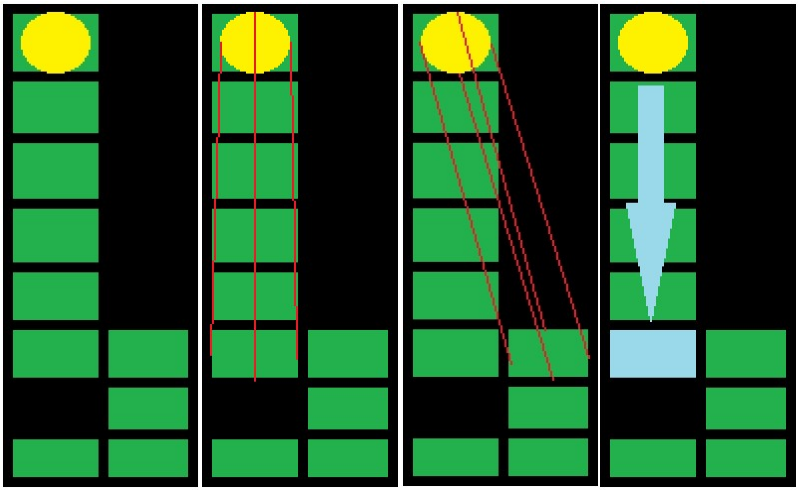


Figure 10, Illustration of the path smoothing algorithm.

After skipping those nodes, the algorithm will check if it is still possible to reach the current node or if the current node is the goal node. If one of those statements is true, it will exit the loop. If the reason was because the next node was the goalNode, it will return the goalNode. If the reason was because the next node wasn't reachable, it will take the previous node that was still reachable and return that node.

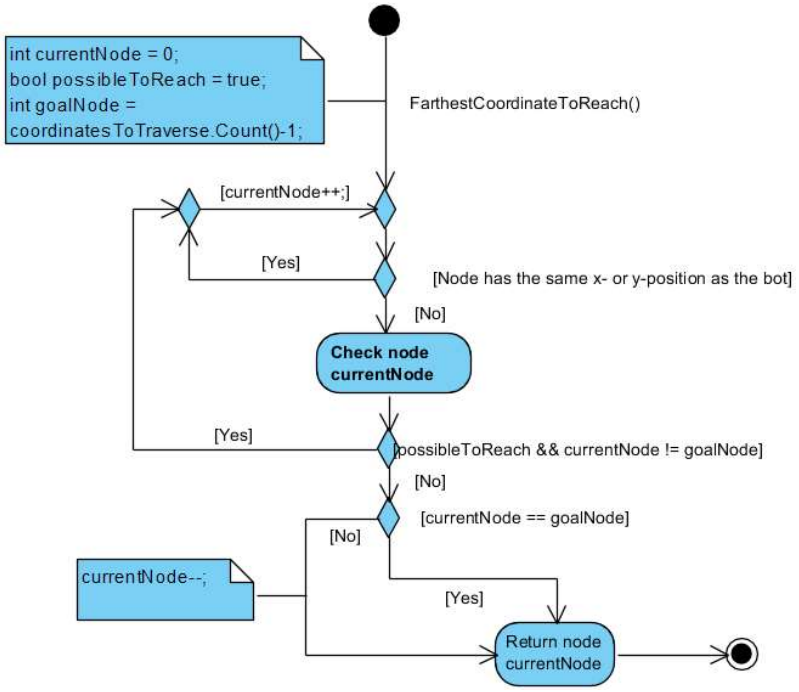


Figure 11, Activity diagram for path smoothing algorithm.

To check if a node is reachable, the algorithm will raycast lines from the top, bottom, left and right-side of the bot to the corresponding sides of the node it tries to reach. Next, it will calculate which coordinates are covered by those lines and these coordinates will be added to an array. If one of those coordinates contains an obstacle, it means that this node can't be reached in a straight line. In this case, the method will return false and break out of the loop. If it returns true, it will continue to the next node.

To calculate which coordinates are covered by a line, the algorithm will first calculate how many horizontal and how many vertical tile-crossings there will be between the start and end point of the line. Next it will calculate if the next intersection will be with a horizontal or with a vertical crossing. If it is with a horizontal line, it will tag the next vertical line and vice versa. This process will then be continued until the end point has been reached.

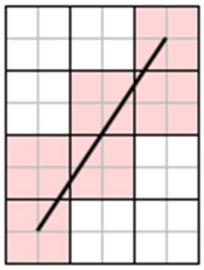


Figure 12, Example for calculating which coordinates are intersected by a line.

Attack behaviour

When the bot can see the enemy through any means of light and they are in range, they should shoot the enemy. To make it fair for the player the bots don't have perfect aim, this is achieved by using an offset the first time the bot aims at the player. This offset is based on the distance between de bot and the target, the further away the target is, the less accurate the bot shoots. If the player moves in both world space and local space (relative to the bot) the bot has to re-aim and will have the offset again. The offset is random so it could randomly shoot 100% accurate. After the first shot the bot with lerp between the last shot position and the actual position of the target based on a constant accuracy value, the higher the value, the more accurate the bot shoots. Finally if the player moved, the bot will take into account his movement and aim a bit in front of its target, this is done by taking the last shot position (red) relative to the target (green) and adding it to the lerped vector (yellow) resulting in the final aiming position (light purple) as seen in figure 11. This isn't 100% fool proof, but it doesn't have to be, since this makes it fairer to the (human) players playing the game.

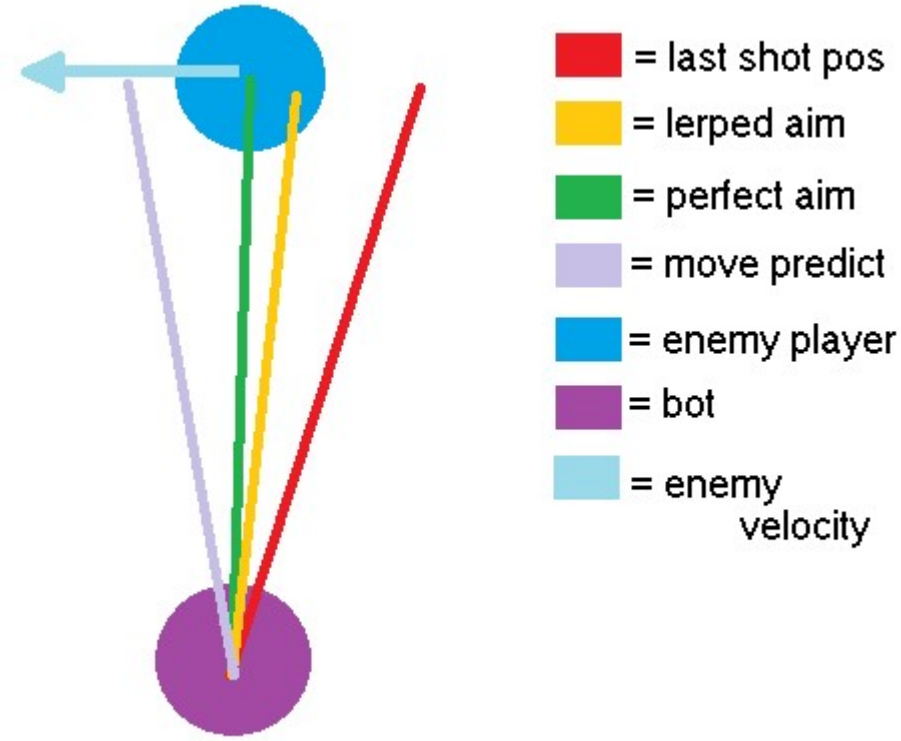


Figure 13, bot aim on a moving target

To check which enemy to shoot, an algorithm is used to find the closest enemy in vision. This is done through the FindClosestEnemy method. This method takes a list of possible candidates and outputs the closest one. This gets called with a list gotten through the game environment which has a list of enemies that the bot can see based on light. The distance between the bot and the enemy gets calculated and if this is less than the lowest distance so far, a raycast is done between the two. If the raycast returns that it has hit an enemy, that enemy will become the closest. If no player is found at all the method returns false, otherwise it returns true. The raycasting only happens if the target is closer than the closest so that there is no wasted performance. Calculating the distance is less taxing than performing a raycast. As seen in figure 12, even though the top enemy is closer, the bot will shoot the left enemy because the top raycast hit the wall and not the enemy.

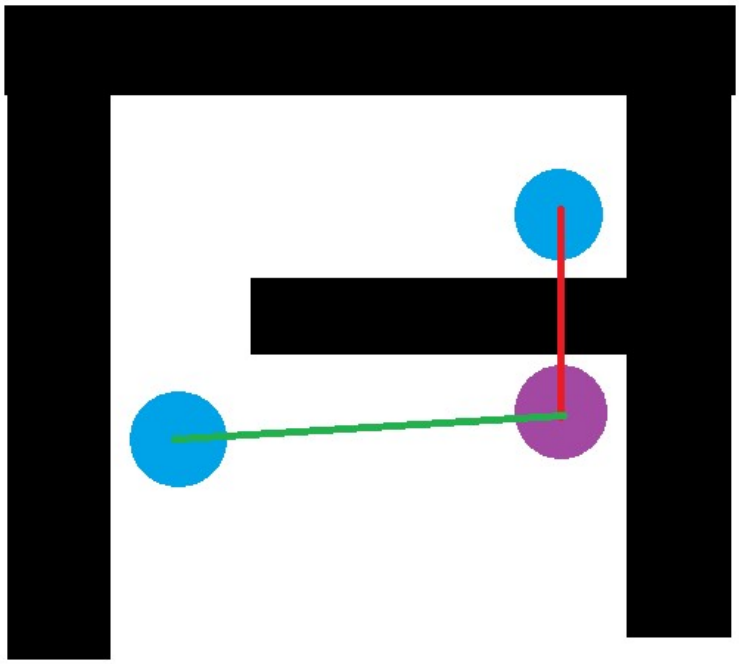


Figure 14, raycasting

Aiming

Because the camera is at a slight angle of 80 degrees instead of a right angle of 90 above the player and the height difference between a floor and a wall the player can't just aim at the mouse position. To get the correct location to aim at we draw an invisible plane across the floor on a height of 0 and raycast from the mouse position on the screen to the invisible plane. The player will then aim at the raycasthit.point, this works on any camera angle and ignores any height differences because the plane is always at 0 height and the raycast only interacts with this invisible plane.

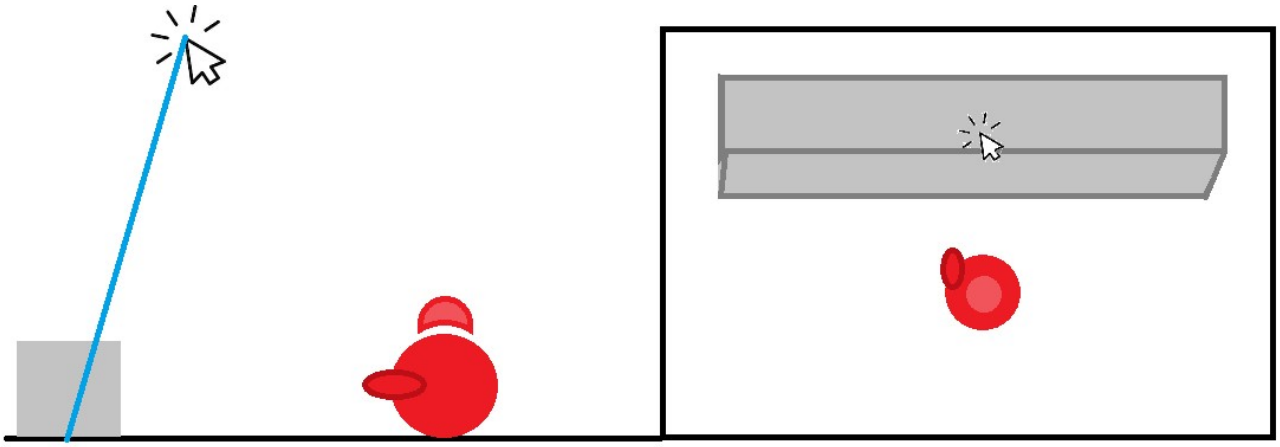


Figure 15, Aiming in side view (left) and camera view (right)

Appendix A: Coding conventions

Our coding conventions are a modified version of the conventions described at https://unionassets.com/blog/stan-s-assets-unity-c-coding-guidelines-465.

• Classes & Interfaces

Public, *Private* and *Abstract* classes in **PascalCase**.
For example, `Button` Interfaces in **PascalCase** + **I**prefix: `IButton`

• Methods

Methods are written in **PascalCase**.
`DoSomething()`

• Fields

Public and *Public static* fields in **PascalCase**.
Static fields have a **s_** prefix + **camelCase**.
Private, *Protected* and *Internal* fields have a **m_** prefix + **camelCase**.
Const are written in **UPPER_CASE**.

```
public class ExampleClass : MonoBehaviour
{
    public  const int MY_CONST_VALUE = 0;
    private const int MY_CONST_VALUE_2 = 0;

    public int PublicField;
    public static int MyStaticField;


    int m_packagePrivate;
    private int m_myPrivate;
    protected int m_myProtected;

    private static int s_myPrivate;
}
```

- ## Parameters

Parameters are written in **camelCase**.

```
public void SetPosition(Vector3 position)
{
    Vector3 newPosition = position;
}
```

Motivations: With such structure, we can always tell:

- Is this field only for private use or publicly available.
- Is this a static field or not.
- Is this a constant.
- Is this field was declared inside a call or it's only a part of the method you are looking at.

- ## Names

Prefix event methods with the prefix **On**.

```
public event Action OnClose;
```

If a method is used to handle delegate callback add the suffix **Handler**.

```
void CloseHandler();
void ResetButtonClickHandler();
void PlayerDisconnectedHandler();
```

Motivations: Make the code be more readable for others by explaining what our methods are doing.

- Brace Style

Braces get their own line.

```
namespace Example
{
    public class ExampleClass : MonoBehaviour
    {
        private int m_property;

        public void DoSomethingMethod()
        {
            if (someTest)
            {
                DoSomething();
            }
            else
            {
                DoSomethingElse();
            }

            switch (variable)
            {
                case 1:
                    break;
                case 2:
                    break;
                default:
                    break;
            }
        }

        public int Property
        {
            get
            {
                return m_property;
            }

            set
            {
                m_property = value;
            }
        }
    }
}
```

- Code Structure Rules

Editor menu item are added in a separate class, that is inside an Editor folder and has *EditorMenu* postfix.

Motivations: This will save a lot of time when we will need to understand what this or that menu item is doing, and which script is adding it.

- Reserved Names

When you have a number of plugin or modules in your project, most of the classes are meant to do similar tasks, so it's very helpful when those classes also have similar names. Classes which add new items to a Unity Editor Menu should have *EditorMenu** suffix Custom script Inspectors should have the same name as the target class + *Inspector** suffix.