UNIVERSITÀ DEGLI STUDI DI MILANO
FACOLTÀ DI SCIENZE E TECNOLOGIE

ARTIFICIAL INTELLIGENCE FOR VIDEO GAMES
M.Sc. in Computer Science - A.A. 2023-2024

BSP Cellular Automata Dungeon

Andrea Fedeli - 10339A

Contents

1 Introduction

This document provides an in-depth exploration of the design and implementation of a Unity project focused on generating cave-like dungeons. The project employs a combination of Binary Space Partitioning (*BSP*) trees for space partitioning and Cellular Automata (*CA*) for detailed dungeon generation. The primary objective is to offer a thorough overview of the project's functionalities, techniques, and implementation methods.

1.1 Project objectives

The technique starts by creating the rooms and corridors using a BSP tree technique for space partitioning. Then a CA is applied to each leaf of the BSP tree, in order to give organic look to the walls of the rooms and corridors.

The main objectives are:

- Design and implement a BSP tree and a way to generate the dungeon outline by defining rooms and corridors exploiting the BSP tree structure.
- Design and implement a CA able to give an organic look to the dungeon.
- Design the number and nature of the states of the CA's cells and the rules that allow to maintain the structure created by the space partitioning technique.

1.2 Constraints

In order to have a good result the project has some constraints:

- The space partitioning technique should be able to create a number of rooms between 20 and 50, over a 200 by 200 meters square area.
- The CA should not create new corridors, nor close the corridors created by the space partitioning technique.
- It is possible that the CA will create small unreachable walking areas: it is not required to check and delete them.

2 Usage

Once the project is running, it's possible to randomly generate cave-like dungeon by selecting in the hierarchy of the scene the DungeonGenerator object and pressing the "Generate Dungeon" button on its inspector.

Before generating a dungeon is possible to set in the DungeonGenerator inspector different parameters that influence the technique results like:

- **Random seed**: the specified value is used as seed of the random generation of different values in the technique. In order to use a random seed is sufficient to set it to 0.
- **Number of rooms**: the specified value determines the number of rooms inside the dungeon. As specified in the constraints the value can be between 20 and 50.
- **Room fill rate**: this value determines the probability that a cell of the CA is set to a room tile.
- **Corridor fill rate**: this value determines the probability that a cell of the CA is set to a corridor tile.

In addition to those parameters, the DungeonGenerator inspector allows to set different flags useful for debugging purposes and visible only in the Unity's "Scene view":

- **Draw BSP**: if set to true, areas generated using the BSP tree are displayed as gizmos lines.
- **Draw rooms**: if set to true, the dungeon's rooms are displayed as gizmos lines.
- **Draw corridors**: if set to true, the corridors that connect different rooms in the dungeon are displayed as gizmos lines.

3 Project structure

Inside the "*Assets*" project folder there are 3 subfolders:

- *Scripts*, which contains the scripts used for the dungeon generation
- *Scenes*, which contains the project scene
- *Materials*, which contains the materials used for the dungeon.

Being a project with no entities, except for the generated dungeon, the default scene provided by Unity is almost perfect; in fact just the camera has changed position and orientation in order to show the produced dungeon.

In addition to the objects provided by the Unity's default scene, an empty object called "*DungeonGenerator*" was added. This element specifies the position where to generate the dungeon and has attached on it two C# scripts, called "*DungeonGenerator*" and "*MeshGenerator*", that are respectively in charge to generate the dungeon and the mesh that describes it.

3.1 Classes and code structure

To achieve a cave-like dungeon the technique is divided into two parts: the first one exploits a BSP tree in order to place the base rooms and corridors, the second one uses a CA to give a more organic look to the previously generated dungeon. These two steps are contained within the DungeonGenerator script, which provides a class of the same name, responsible for implementing both steps. To support the algorithm some utility classes were created:

- A class `TreeNode` implementing a generic node of a binary tree data structure. The class provides different utility methods and is generic to maximize the decoupling between the data structure and the values that it contains.
- A class `RectSpaceArea` that represents a single rectangular area inside the BSP. The class stores each area using four variables that represent the minimum and maximum x and y coordinates of two opposite corners of the rectangle and provides methods to get different information of the space that represents, such as the total area of the space or the height and width of it.
- A class `BSP` providing a way to build a binary space partition. The class inherits from `TreeNode` because the space partitioning exploits a binary tree to build and store the different partitions, also it uses as specific type of each node the class `RectSpaceArea` in order to model each space with a rectangular shape.
- A class `MeshGenerator` that provides a method to build a mesh based on the generated dungeon structure, stored as a 2d matrix, using a square marching algorithm. Inside this class are also defined different utility classes specific for the mesh generation.

Figure 1 shows the class diagram describing the relationships between those classes.

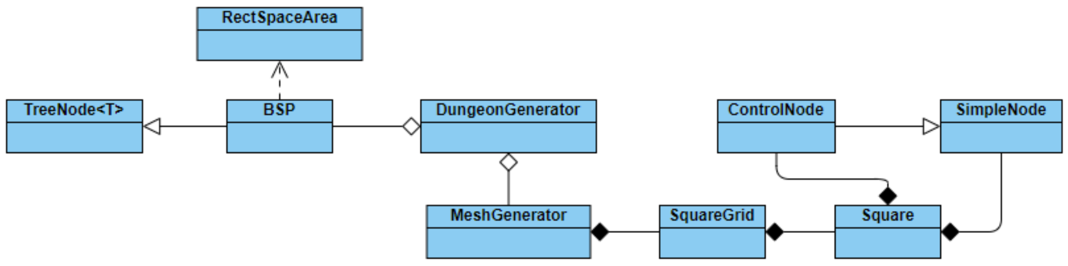


Figure 1: Class diagram of the project.

4 Technique implementation

As described in chapter 3.1 the technique is implemented by the `DungeonGenerator` class, in fact once the inspector button is pressed a public method of this class called `GenerateDungeon` is executed. The general outline of this method is:

1. Create an instance of the `BSP` class providing the area that needs to be subdivided.
2. Call the method `BuildBSP` to build the BSP tree, specifying the number of rooms that needs to be generated and the minimum and maximum percentages that can be used for splitting each area.
3. Initialize the 2d matrix that will store the generated dungeon.
4. Call the methods `CreateRooms` and `CreateCorridors` that places rooms and corridors based on the areas of the generated BSP tree and stores them in the matrix.
5. Call the method `ApplyAutomata` for each leaf area of the generated dungeon to give an organic look to each room's space area, specifying the number of iterations to apply.
6. Retrieve the `MeshGenerator` component and call on it the `GenerateMesh` method to create the mesh for the dungeon.

In the following sections are provided more detailed explanations of each phase.

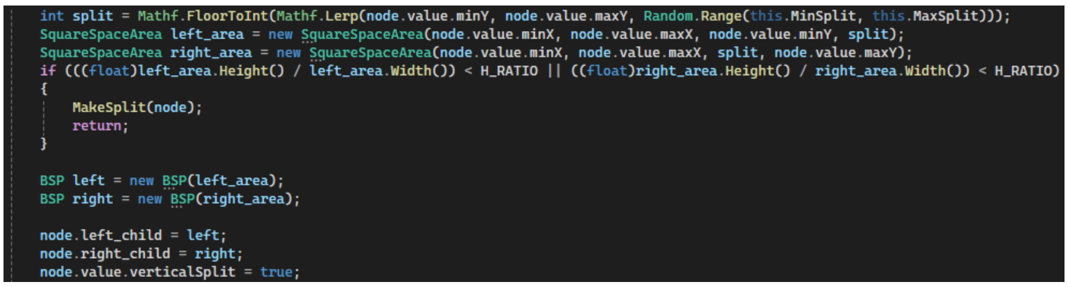
4.1 BSP tree generation

The first step consists of generating a BSP by recursively divide in different cells a 200 by 200 meters area modeled by an instance of the `RectSpaceArea` class, while creating a binary tree during the process. This is done by the `BuildBSP` method of the `BSP` class. Being a specialization of a generic node of a tree, the final result is stored inside the `BSP`'s field called `value`, which will represent the root of the tree.

The method takes as parameters the number of splits that needs to be applied to the space, which will contain the dungeon's rooms, and the minimum and maximum percentage where a cell can be split, in order to obtain sufficiently large subareas. For example, using as minimum percentage 10% and as maximum percentage 90%, a 100 meters width cell can be split starting from 10 meters to a maximum of 90 meters. Then the method cycles for the specified number of splits and for each iteration:

1. Makes a split randomly choosing its orientation and position.
2. Selects the next area to split by searching inside a list of tree's leaves the one with the biggest area.

The orientation of the split is randomly selected between horizontal and vertical directions, by randomly generating the value 0 (horizontal) or 1 (vertical), then the split is done simply by computing a linear interpolation between the minimum and maximum *x* or *y* coordinates stored inside the RectSpaceArea value of the considered BSP node, and using as weight a randomly generated value between the minimum and maximum split percentages. Then, to ensure that the obtained subareas are sufficiently large, the ratio between their width and height is compared with a fixed value, which for this project is empirically set to 0.45; if the ratio is smaller then this value then the split needs to be remade all over again, because the obtained cells are too small to place rooms inside them. Once the split is successfully done, two BSP instances are created and assigned to the left_child and right_child fields of the currently considered cell.



```
int split = Mathf.FloorToInt(Mathf.Lerp(node.value.minY, node.value.maxY, Random.Range(this.MinSplit, this.MaxSplit)));
SquareSpaceArea left_area = new SquareSpaceArea(node.value.minX, node.value.maxX, node.value.minY, split);
SquareSpaceArea right_area = new SquareSpaceArea(node.value.minX, node.value.maxX, split, node.value.maxY);
if (((float)left_area.Height() / left_area.Width()) < H_RATIO || ((float)right_area.Height() / right_area.Width()) < H_RATIO)
{
    MakeSplit(node);
    return;
}

BSP left = new BSP(left_area);
BSP right = new BSP(right_area);

node.left_child = left;
node.right_child = right;
node.value.verticalSplit = true;
```

Figure 2: Code snippet for vertical split.

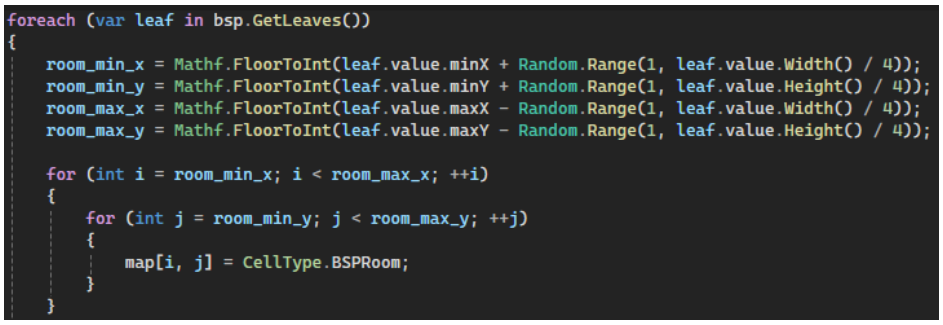
4.2 Rooms and corridors creation

Once the space partitions are created, the technique proceeds calling the CreateRooms and CreateCorridors methods of the DungeonGenerator class to generate rooms inside each partition and corridors to connect them. When rooms and corridors are generated, the cells of a matrix corresponding to their coordinates are set to a specific value to be used in the CA step.

The CreateRooms method cycles on the leaves of the BSP tree, which are obtained using the GetLeaves method of the TreeNode class, each leaf will contain a cell where a room needs to be placed. The room is placed simply by computing the minimum and maximum *x* and *y* coordinates of the two room's edges using the following equations:

$$
\begin{aligned}
room.min_x &= cell.min_x + randomShift \\
room.max_x &= cell.max_x - randomShift \\
room.min_y &= cell.min_y + randomShift \\
room.max_y &= cell.max_y - randomShift
\end{aligned}
$$

The idea is to displace the minimum or maximum *x* and *y* coordinates using a random value generated between 1 and $\frac{1}{4}$ of the cell height or width, in order to avoid rooms that belongs to different cells to be adjacent and to generate a sufficiently large room. With these simple equations is guaranteed that the rooms will always be inside each cell area and that two rooms belonging to two different cells won't be adjacent.



```
foreach (var leaf in bsp.GetLeaves())
{
    room_min_x = Mathf.FloorToInt(leaf.value.minX + Random.Range(1, leaf.value.Width() / 4));
    room_min_y = Mathf.FloorToInt(leaf.value.minY + Random.Range(1, leaf.value.Height() / 4));
    room_max_x = Mathf.FloorToInt(leaf.value.maxX - Random.Range(1, leaf.value.Width() / 4));
    room_max_y = Mathf.FloorToInt(leaf.value.maxY - Random.Range(1, leaf.value.Height() / 4));

    for (int i = room_min_x; i < room_max_x; ++i)
    {
        for (int j = room_min_y; j < room_max_y; ++j)
        {
            map[i, j] = CellType.BSPRoom;
        }
    }
```

Figure 3: Code snippet for room generation.

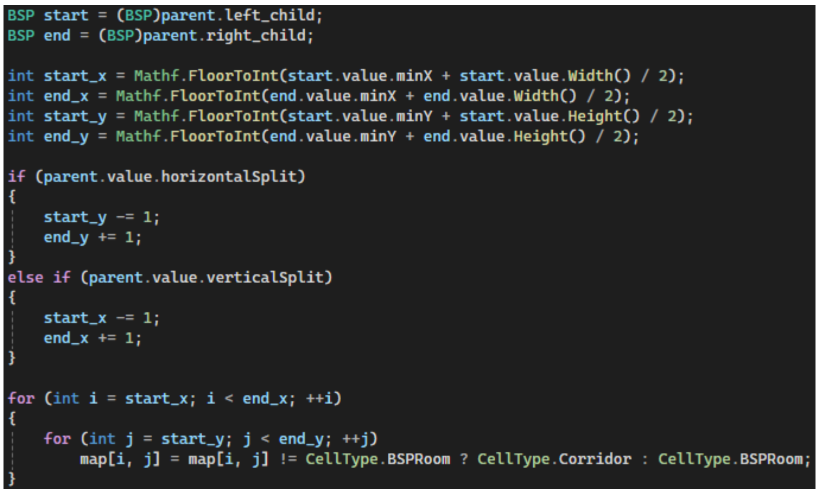
The `CreateCorridors` method exploit the BSP tree structure by visiting it using a preorder visit and creating a corridor, starting from the center of the left child's cell and reaching the center of the right child's cell, which are computed by these equations:

$$start_x = left.min_x + left.width/2$$
$$end_x = right.min_x + right.widht/2$$
$$start_y = left.min_y + left.height/2$$
$$end_y = right_min_y + right.height/2$$

Since the rooms are generated by simply reducing the size of the cells, it's guaranteed that the center of them is always occupied by a room or by a corridor that connects two internal nodes of the tree, so no further verifications are needed. Finally, the method needs to shift these computed coordinates to make the corridors large enough. To do so, it verifies if the split made during the BSP tree generation is horizontal or vertical, by checking the `verticalSplit` and `horizontalSplit` fields of the `BSP` instance (two separated fields was specified for readability reasons):

- If `verticalSplit` is set to true, then starting and ending *y* coordinates are respectively decremented and incremented by 1.
- If `horizontalSplit` is set to true, then starting and ending *x* coordinates are respectively decremented and incremented by 1.

In order to allow the preorder visit of the tree structure using a simple `foreach` statement, the `TreeNode` class implements the C# `IEnumerable<T>` interface providing a preorder visit iterator. Although the tree is visited using a `foreach` statement, it is important to note that the internal implementation is still a recursive approach, which can be used because the number of nodes within the tree is limited. If the tree contained a much larger number of nodes, it would be necessary to use an iterative approach.



```
BSP start = (BSP)parent.left_child;
BSP end = (BSP)parent.right_child;

int start_x = Mathf.FloorToInt(start.value.minX + start.value.Width() / 2);
int end_x = Mathf.FloorToInt(end.value.minX + end.value.Width() / 2);
int start_y = Mathf.FloorToInt(start.value.minY + start.value.Height() / 2);
int end_y = Mathf.FloorToInt(end.value.minY + end.value.Height() / 2);

if (parent.value.horizontalSplit)
{
    start_y -= 1;
    end_y += 1;
}
else if (parent.value.verticalSplit)
{
    start_x -= 1;
    end_x += 1;
}

for (int i = start_x; i < end_x; ++i)
{
    for (int j = start_y; j < end_y; ++j)
        map[i, j] = map[i, j] != CellType.BSPRoom ? CellType.Corridor : CellType.BSPRoom;
}
```

Figure 4: Code snippet for a single corridor generation.

4.3 Application of the cellular automata

The idea of a CA is to apply to each cell of a matrix a set of rules based on their surroundings a certain amount of time. In this case the matrix is stored in a `DungeonGenerator`'s field called `map`, the matrix has the same width and height of the area we previously subdivided with the BSP and its cells contain values from an enumerator called `CellType`, which describes all the possible types that a cell in the matrix can take. This values are:

- `BSPRoom`: it is used to represent a cell whose dungeon's area is inside a room generated during the BSP phase.
- `Room`: it is used to represent a cell whose dungeon's area is inside a room generated during the CA phase.
- `Wall`: it is used to represent a cell whose dungeon's area isn't inside a room or a corridor.
- `Corridor`: it is used to represent a cell whose dungeon's area is inside a corridor generated during the BSP phase.
- `CorridorBorder`: it is used to represent a cell whose dungeon's area is adjacent to a corridor.
- `CorridorBorderWall`: it is used to represent a cell whose dungeon's area is adjacent to a corridor and that isn't inside a room during the CA phase.
- `CorridorBorderRoom`: it is used to represent a cell whose dungeon's area is adjacent to a corridor and that is inside a room during the CA phase.

The definition of these values is done in this way mainly for two reasons: first, they allow to easily understand if a cell representing a dungeon's area is inside a room or corridor, whose value was assigned during the BSP phase, then it can't be modified; second, they allow to easily check if a cell is near a corridor, in order to apply some specific rules to them.

As shown in figures 3 and 4, the matrix cells were initialized inside the `CreateRooms` and `CreateCorridors` methods, by setting each cell corresponding to the coordinates of the created rooms and corridors with a specific value of the enumerator. For example, if a room gets generated from $(min_x = 0,\ min_y = 25)$ to $(max_x = 0,\ max_y = 15)$, then the `CreateRooms` method initializes the `map`'s cells inside those indexes with the value `BSPRoom`.

Once the BSP phase is finished, `map` is set up by assigning the `CorridorBorder` value to all of its cells that are adjacent to a corridor; this is done by considering the whole matrix and not just the portion corresponding to a leaf of the BSP tree to handle situations in which a corridor is generated on the edge of a BSP cell, making it impossible to identify cells in the matrix that are adjacent to that corridor, but belong to an adjacent BSP tree's leaf.

After that, the CA is applied to the portions of matrix corresponding to the area of each leaf in the BSP tree. The CA is composed by two steps:

1. It applies an initialization step by cycling the cells of the considered area matrix and checking their value:
   - If it is `Wall`, then it randomly sets the considered cell to `Room` or `Wall` using as probability the fill rate specified in the inspector.
   - If it is `CorridorBorder`, then it randomly sets the considered cell to `CorridorBorderRoom` or `CorridorBorderWall` using as probability the fill rate specified in the inspector.
2. It applies the rules to the cells of the considered area matrix.

Rules are designed to follow these goals:

1. Let rooms and corridors expand when they don't generate new connections with a pre-existing corridor or room
2. Maintain rooms, corridors and their connections if they are generated during the BSP phase
3. Keep track of corridors' borders

The defined rules are:

1. If the value of the current cell is `BSPRoom` or `Corridor`, it remains the same to avoid deleting rooms or corridors created during the BSP phase.
2. If the value of the current cell is `Room` or `Wall`, then:
   (a) It counts the corridors cells adjacent to the current cell. If there is none of them, it isn't possible to generate new connections with a pre-existing corridor, so CA can add `Room` cells using the following rules:
      - If the number of adjacent `Wall` cells is greater then 4, the current cell is set to `Wall`. In this way, all cells that are in areas where the majority of the cells are walls also become walls.

UNIVERSITÀ DEGLI STUDI DI MILANO

- If the number of adjacent `Wall` cells is lower then 4, the current cell is set to `Room`. In this way, all cells that are in areas where the majority of the cells are rooms also become rooms.
- If the number of adjacent `Wall` cells is equal to 4, the current cell keeps its value.

(b) If there is at least one corridor cell adjacent to the current cell, it's possible to generate new connections with a pre-existing corridor, so CA avoid adding `Room` cells using the following rules:

- If the current cell value isn't `Room`, it remains the same because it will be a `Wall`
- Otherwise, it is set to `CorridorBorderWall` because it is adjacent to a corridor and must be a wall to avoid generating new connections with it.

3. If the value of the current cell is `CorridorBorderWall` or `CorridorBorderRoom`, we want to add a `CorridorBorderRoom` only if the adjacent cells aren't set to `Room` or `BSPRoom` values, otherwise the CA will generate a new connection with a pre-existing corridor. It does so with the following rules:

(a) If there are more than one `Room` or `BSPRoom` adjacent to the current cell, its value is set to `CorridorBorderWall` to keep the corridor and the room separeted

(b) Otherwise, it is set to `CorridorBorderRoom` to let the corridor expand.

Since the rules are applied considering the values of each cell at the previous step, they update a local copy of the portion of the matrix, that will be copied into the original one at the end of the iterations.

At the end of this phase the `map` matrix describes the structure of the dungeon, that, when rendered, should look something like the figure 5.

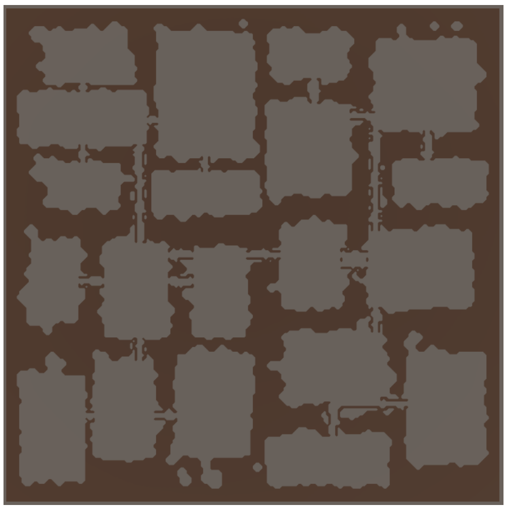


Figure 5: Example of a dungeon with 20 rooms generated using 0.4% *and* 0.15% *of fill rate for rooms and corridors respectively.*

4.4 Mesh generation

Although mesh generation is not directly related to the dungeon generation technique, a brief explanation of the algorithm is given in the next few lines.

As mentioned earlier, the square marching algorithm is used to create the mesh. This algorithm is suitable for this case because it is based on a binary image, which can be easily obtained using the 2d matrix inside the `DungeonGenerator` class, considering all of its walkable cells (`Room`, `Corridor`, `BSPRoom` and `CorridorBorderRoom`) as the first binary value, and all its nonwalkable cells (`Wall`, `CorridorBorderWall`) as the second binary value.

Once we have done that, the algorithm builds on top of the image a grid of squares, where each square is characterized by 4 control nodes placed on its corners and 4 simple nodes placed on its edges, halfway between two corners. These nodes represent the vertices of the mesh.

A control node can be in one of two statuses:

- Enabled, in our case if the node corresponds to a walkable cell in the image
- Disabled, in our case if the node corresponds to a nonwalkable cell in the image

All the possible combinations of the 4 control nodes of a square (16 in total) determine the connections between the corresponding vertices, forming the triangles of the mesh following the shapes showed in figure 6. To determine which combination a square represents, a single bit value is assigned for each of the 4 control nodes, forming a 4 bit binary number following a clockwise order and considering the top left control node as the most significant bit. Each bit is set to 1 if the corresponding control node is enabled, for example, the following are some of the possible combinations:

$$All\ control\ nodes\ active \implies (1111)_2 = (15)_{10}$$
$$Top\ left\ and\ bottom\ right\ active \implies (1010)_2 = (10)_{10}$$
$$None\ active \implies (0000)_2 = (0)_{10}$$

Based on these rules, it is possible to build the mesh that describes the generated dungeon.

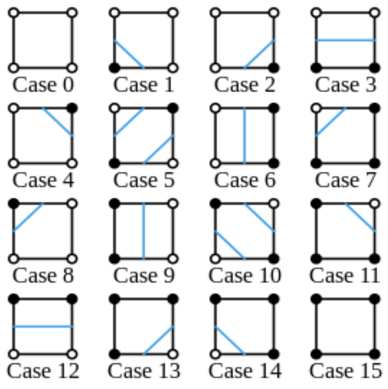


Figure 6: Possible shapes for a square based on the combination of its control nodes (picture from Wikipedia)