Home Creator:
Procedural house generator

Margherita Orlando
970894
Artificial Intelligence for Videogames
Dept. Of Computer Science
Università di Milano

Index

Introduction

*Home Creator i*s a procedural house generator, it allows you to create rooms that will then be randomly positioned within a given space, through the use of a Binary Space Partitioning algorithm that takes care of dividing the space. The user interfaces with an editor called *HomeCreatorEditor* which allows you to randomly generate a house using the *CreateNewHome* button after entering all the required data.

The name “Home Creator” was chosen to emphasize the fact that the algorithm allows you to create your own home.

Home Creator Editor

The user interfaces with the *HomeCreatorEditor* to define all the characteristics that the house he wants to generate must have, in particular:

- **Length and width of the base of the house:** the house is built on a space (a rectangle) whose dimensions are decided by the user. All the rooms will then be inserted in this rectangle. The house can not be smaller than 6, so that the objects are created correctly.
- **Average length and width of the rooms:** the dimensions of the rooms will vary around a range based on the measurements given, so that each room is similar but not equal in size. A room can not be smaller than 3, so that the objects are created correctly.
- **Max Iterations:** number of maximum iterations that the program must adhere to.
- **Number of rooms**
- **Number of windows**
- **Room Top / Bottom Corner Modifier:** these are modifiers used to adjust the distance between one room and another so that the user can decide whether to want spaced or contiguous rooms.
- **Room offset:** value used to define the distance between the walls of one room and another, including the walls of the perimeter of the house. Avegare room length/width is modified in base on the offset value.
- **Floor material**

- **Horizontal and vertical walls, windows and doors:** depending on the position in which they are to be inserted, one object or the other will be instantiated.

Entities

Node

It is an abstract class containing the corners and the list of child nodes of the current node, which can be added or removed.

Line

Class representing the "line" that will be used for the partitioning of the space. The class has orientation and coordinates as attributes.

RoomNode

This class inherits the Node class and takes care of the creation of a rectangle, that is the space occupied by the individual rooms, associating each room with the area of the individual corners and its index in the tree.

Room

This class was not fully used during the development of the project, but has been included for future implementations. In fact, in this class we find the "widthMax", the "lengthMax" and the "typeOfRoom" associated with each room, in such a way that will allow to create rooms of a certain type, with certain dimensions.

Binary Space Partitioner

A Binary Space Partitioning (BSP) tree is a data structure that represents a recursive, heirarchical subdivisions of n dimensional space into convex subspaces. Starting off with the root node, all subsequent insertions are partitioned by the hyperplane of the current node. In this case the hyperplane is a line.
BSP trees are very useful for real time interaction with displays of static images. Before the static scene is rendered, the BSP tree is calculated.

The algorithm:

- The constructor of the *BinarySpacePartitioner* class creates a *RoomNode* which will be our root node by assigning homeWidth and homeLength as values.

- The *PrepareNodesCollection* method is then called, which takes care of returning the list containing the spaces in which the rooms will be placed. This method calls within a *while* (which continues until the maximum number of iterations is reached) the *SplitTheSpace* method which is called only if at least one of the dimensions of the current node (length or width) is greater than or equal to the user-defined dimensions in the editor.

- *SplitTheSpace* is used to divide the space vertically or horizontally based on the orientation of the "line" thus creating 2 nodes each time that will be added to the list to be returned through the *AddNewNodeToCollections* method.

- The orientation of the Line is given by the *GetLineDivingSpace* method: it choose the orientation based on the size of the space and return a new *Line* that stores all the values needed for splitting the space (orientation and coordinates).

- The coordinates of the Line are given by the *GetCoordinatesForOrientation* method: it create a point between the value of the minimum and the maximum ensuring that the room will not be splitted to a smaller section of the room length / width.

Home Generator

This class takes the size of the house in its constructor (homeWidth / homeLength) and is called to start the space partitioning algorithm (BinarySpacePartitioner), go up the tree using the *TraverseGraphToExtractLowerLeaves* method of the *StructureHelper* class, then generate the list of rooms using the *RoomGenerator* class and then return it.

Structure Helper

It is a static class that contains 3 methods:

- **TraverseGraphToExtractLowestLeaves:** it is used to return a list containing the leaves of the tree, the nodes that have no children.
- **GenerateBottomLeftCornerBetween:** returns a randomly generated Vector2Int as follows:

```
int minX = boundaryLeftPoint.x + offset;
int maxX = boundaryRightPoint.x - offset;
int minY = boundaryLeftPoint.y + offset;
int maxY = boundaryRightPoint.y - offset;

return new Vector2Int(
Random.Range(minX, (int)(minX + (maxX - minX) * pointModifier)),
Random.Range(minY, (int)(minY + (minY - minY) * pointModifier)));
```

  The returned vector will be the new bottom left point in the *RoomGenerator* class, used to create the corner area of the room to be inserted.

- **GenerateTopRightCornerBetween:** returns a randomly generated Vector2Int as follows:

```
int minX = boundaryLeftPoint.x + offset;
int maxX = boundaryRightPoint.x - offset;
int minY = boundaryLeftPoint.y + offset;
int maxY = boundaryRightPoint.y - offset;

return new Vector2Int(
Random.Range((int)(minX+(maxX-minX)*pointModifier),maxX),
Random.Range((int)(minY+(maxY-minY)*pointModifier),maxY)
);
```

  The returned vector will be the new top right point in the *RoomGenerator* class, used to create the corner area of the room to be inserted.

Room Generator

This class uses the *GenerateRoomsInGivenSpace* method to create the rooms: for each space previously created, the bottom left and top right corners are randomly generated between a minimum value and a maximum value, using the *StructureHelper* methods, taking into account the offset values and any modifiers entered by the user at the beginning in the editor.

Home Creator

This class is a MonoBehavior. Inside we find the *Start* method, called before the first frame update, which calls the *CreateHome* method.

CreateHome: The first function to be called by this method is *DestroyAllChildren* which takes care of destroying all the gameObjects that were created in a previous execution of the program. Then a check is made on the number and size of the rooms entered: if the size of the rooms exceeds those of the house, a warning is reported and a return is performed. If, on the other hand, the total area of the rooms to be entered (taking into account all the modifiers) is greater than or equal to the air of the house, a message of warning is reported to the user saying that there is no possible configuration.
The *CalculateHome* method of the *HomeGenerator* class is then called and for each room, which is in the list returned by the method, the *CreateMesh* method is called.
After doing this, the walls of the rooms are created first via *CreateWalls* and then the *CreateMesh* method is invoked again to create the mesh of the house base which will act as a “corridor" between the rooms. At the end, the *WindowsGenerator* and *CreateWalls* methods are called, to generate the windows, the main door, and base walls of the house.

Create Mesh: creates a mesh starting from the position of its vertices (bottomLeft, bottomRight, topLeft, topRight). In this method *DoorsGenerator* is called for each room mesh created so that the doors are placed once for room. The *AddWallPositionToList* method is then called for each point of the sides that compose the rooms, so that a vertical or horizontal wall is associated with each point using the *possibleWallHorizontalPosition* and *possibleWallVerticalPosition* lists.

Create Walls: it scrolls all the walls in *possibleWallHorizontalPosition* and *possibleWallVerticalPosition* in such a way that invoke the *CreateWall* method passing it the correct prefab to instantiate.

Create Wall/Door/Window: They create the instance of a gameObject which can be a wall, a door or a window.

Doors Generator: It inserts a door per room by choosing one of the sides on which to place it randomly. After choosing a side, calculate the midpoint between two vertices using the *CalculateMiddlePoint* method and create the door in that position with the *CreateDoor* method, only if the chosen side does not match the perimeter of the house and if there is not already another door. If the chosen side does not fit, the next side on which to insert a door is chosen using *RandomIntExcept* so that the same side is not chosen again.

RandomIntExcept: It executes the random function on a range of values from which some, previously identified as exceptions, have been removed.

Main Door: It inserts the main door of the house by placing it randomly on one of the sides of the house checking that its position does not correspond to one of the positions of the walls of the rooms and that there is at least some space to allow passage, to do this the *RoomPosition* method is used.

Room Position: It returns a list of Vector3 containing the positions of all the walls of all the rooms.

Windows Generator: Windows are first of all inserted into the rooms whose sides coincide with the perimeter of the house, creating a window on each side. After that, if the number of windows to insert is still greater than 0, the *MoreWindows* method is called.

Max Windows: returns the maximum number of extra windows, to be inserted in the "corridors" and not in the rooms, taking into account the offset between one window and another which varies according to the size of the house.

More Windows: takes care of inserting the number of windows left in a random way. First of all, the offset between one window and another is calculated based on the size of the house and then a *for* that reaches the value returned by *MaxWindows is started*. Inside the *for* one of the 4 edges (top, bottom, right or left), on which the window will be inserted, is chosen at random: if the position in which to insert the window does not correspond to one position of of the rooms (always calling *RoomPosition*), does not corresponds to the position of the other windows already inserted and does not correspond to the position of the main door of the house, the *CreateWindow* method is called and the next edge on which to insert the window is chosen using *RandomIntExcept* so that the same side is not chosen several times in a row. The windows are positioned first from the middle of one side of the house to the end and then from the beginning of one side of the house to the middle minus the offset.

Conclusions, future implementations and corrections

The program may sometimes have some bugs because not all the configurations created are acceptable: it is not always possible to cross the whole house without encountering obstacles, especially when the modifiers are set to maximum. However, this problem can be overlooked by the user regenerating a new house configuration until he is satisfied with the result. Some future implementations, in addition to the corrections of the current errors, will be the introduction of the floors inside the house and the replacement of the current rectangular base with one that can be adapted according to the positioning of the rooms.
Although the functionality of inserting multiple floors has not been implemented yet, it is possible to create a building by saving, after pressing *"CreateNewHome"*, the preferred configuration as a prefab and then combining the prefabs by simply changing their position in y. Another future implementation consists in exploiting the *Room* class to create rooms of different types (bathrooms, bedrooms, kitchens, etc.) and associating each type with a set of objects to be inserted, always in a pseudo-random way, inside of the rooms of the same type, in such a way that the algorithm will create a real randomly furnished house.