PRIFYSGOL
BANGOR
UNIVERSITY

Bangor University Technical Report CS-TR-004-2014

Randomised Multiconnected Environment Generator

Christopher J. Headleand, Gareth Henshall, Llyr Ap Cenydd, William Teahan

20th October 2014

Abstract

This technical report presents a novel method for the generation of randomised, multiconnected environments. This is released with the intention of providing the AI community with a standard method of generating sandbox environments for the testing and benchmarking of situated agent algorithms. We will describe the implementation, and conclude by providing a code library which can be utilised in other projects under a BSD licence.

1 Introduction

When testing and evaluating various situated, embodied agent algorithms it is often important to place them in a randomised environment, to evaluate their effectiveness. These sanitised environments allow us to demonstrate that the agent is able to operate in an independent manor, in a dynamic setting. It also removes one possible confounding variable, notably, the possibility that the algorithm is only effective in a single environment configuration.

Multiconnected environments (also known as conjunct environments, non-perfect mazes or labyrinths) are good candidates for use as a testing environment. This is because much of the real-world is a multiconnected environment, including buildings and road networks. In contrast there are relatively few, real-world examples of linear environments or perfect mazes.

However, few environment generators in the public domain generate a truly stochastic environment. In fact, most public domain algorithms produce a relatively standardised configuration of rooms connected by corridors, conforming to a style which could, in principle be learnt. While the possibility of learning an environment is unlikely in most cases, we must always consider external factors, and clearly with embodied agents, the environmental factor cannot be ignored.

There is also an argument that by conforming to an standard method of generating test environments, benchmarking exercises could be better undertaken. We present our algorithm as a possible candidate.

2 Background

The computer games industry has arguably put the most research effort into procedurally generated environments. This is for several reasons, including providing the player with a unique experience every time they play the game. However, procedurally generated environments also have a financial benefit. Once the generator has been developed, large environments or "open worlds" can be generated without any additional cost implication (no additional developers or artists are required). In contrast, if the environment is manually designed, the cost implications scale with the size of the environment being developed.

In our introduction, we discussed a multi-connected environment comprised of a variety of open spaces and connections. The games industry typically refers to this type of environment as a "dungeon". The Future Data Lab website [1] provides a list of procedural various dungeon generation algorithms used in the games development industry, for which we will provide a brief overview.

Random Room Placement Described as one of the most common dungeon generator algorithms. This places room of random size randomly on a grid ensuring there are no overlaps. The algorithm then loops over the rooms creating connections using a variant of the A* algorithm.

Cellular Automata This method uses a cellular automata to create a natural looking cave system. The main difference between this approach and the others described is that it avoids the room and corridors archetype, and instead grows a single, connected space.

BSP Tree The approach begins with a rectangular, blank dungeon template, which is then subdivided into two spaces of non-equal size, then these new spaces are also subdivided into two, with this process continuing for a set number of iterations. Within each of the newly created sub-spaces, a room is

randomly placed, and connections are made between each of the split rectangles. While this approach makes a good use of the initial space, the number of rooms and connections are constrained by the initial parameters, meaning that the environment is not truly stochastic.

Procedurally Built This approach tries to model the way a man-made dungeon may actually be built. First an initial room is created, from this room a random number of walls are selected and a door placed along their edge. On the other side of this door, a feature is placed, either a room or a corridor. This grows the environment until a terminal condition is reached (such as the generation of a desired number of rooms). One criticism of this approach is that it makes poor use of space, and leads to a very linear environment with few interconnections.

3 The New Algorithm

Our new algorithm begins by generating a number of voxels in a 2-dimensional grid, the number of voxels generated being the product of the width and length of the environment the user specifies. This block of voxels provides us with a blank environment that we can develop from.

The next phase involves stamping random room outlines into the blank environment. Each room is generated at a random initial x, y coordinate, with a size generated as a random sample between a minimum and maximum room size (that is user defined). All the voxels within the room are given a label of "Room" and given an ID which represents the order in which the rooms were generated. The immediate border voxels around the room are labelled as "Wall" and given a null id. In figure 1 the voxels labelled as "Room" are displayed in grey, whereas the voxels labelled as "Wall" have been left black.

Figure 1: The blank environment with a single room, voxels labelled as Wall are highlighted in Grey.

The algorithm now loops through generating random rooms. Each new randomly generated room overwrites any voxel data previously defined. In figure 2 we can see a second generated room which has overwritten the voxel data in the bottom left of the first room. We can consider this to be overlaying new rooms to generate a patchwork configuration.

This process continues generating rooms randomly within the blank environment. This provides us with a floor plan which resembles figure 3. The method provides us with something which typically fills the majority of the available space organically, without the need for complex space filling algorithms, or the resultant predictable layouts generated by the BSP tree approach.

Figure 2: A second room added to the example environment. Notice how the bottom left of the first room has been overwritten.

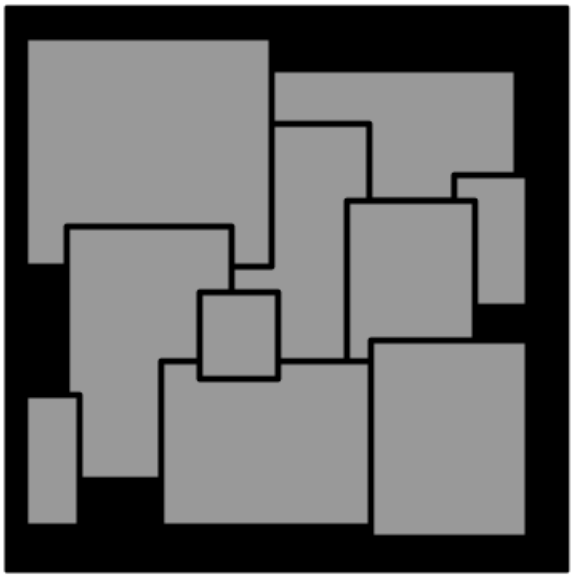

Figure 3: The result of the random room generation.

Once the room generation algorithm has concluded, the next phase is the door placement, ensuring that all

rooms within the final environment are accessible to the agent.

This is achieved by first selecting a random voxel with the label "Room" and accessing its ID. In figure 4, this initial voxel has been identified with a black circle, and we will refer to this as the "focal point". Then all voxels which share the same ID as the focal point are instructed to change their label to "Accessible".

Four paths are then generated by stepping through the voxel array in four directions, left, right, up and down. For each step, the current voxel is sampled, and if its ID is the same as the ID of the focal point, or if the it's label is"Wall", then the steps in that direction continue. If the current voxel is unlabelled, then that path is destroyed, in figure 4, this is represented by the dashed red lines.

Alternatively, if the label of the current sample voxel is "Room", then the path has found a new room, and all voxels with the same room ID as the sampled voxel, have their label changed to "Accessible". To determine where a door should be placed, we simply back track along the path until we find a voxel or voxels with the label "Wall" and change their label to "Accessible". In the figures, all rooms which have had their label changed to "Accessible" are coloured blue, and the paths which have created this route are coloured black.

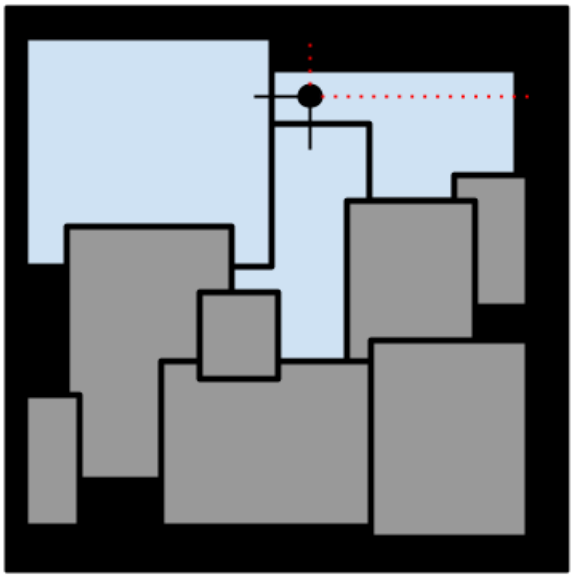

Figure 4: The first randomly placed focal point, showing the exploration in four directions. Notice the failed search, up and right, and the successful searches down and left.

In each new room discovered, a new focal point is spawned within that room, and the process is repeated, as can be seen in figures 5 and 6. Eventually, a focal point will be spawned and no new rooms will be found. This has been highlighted in figure 6 with two white focal points which have four red, dashed lines from each. This does not necessarily mean that there isn't a connected room (as with the bottom left focal point), but could simply mean that the spawned focal point is just not in a position to discover it.

There are a few approaches which can be taken to solve this issue. However, the focus of this algorithm is that the environment is randomised and non-uniform, and a few missed rooms can add to this effect. However, too many missed locations could be detrimental to the environment generation. A compromise solution we implemented was as follows If a new focal point found no new rooms, it was re-spawned at a different voxel in the environment with an "Accessible" label. If both the first and second attempts failed then that branch of the tree was destroyed.

The search continues until either all the branches from each generated focal point have been killed or, alternatively the user can set a search depth to limit the size of the environment generated.

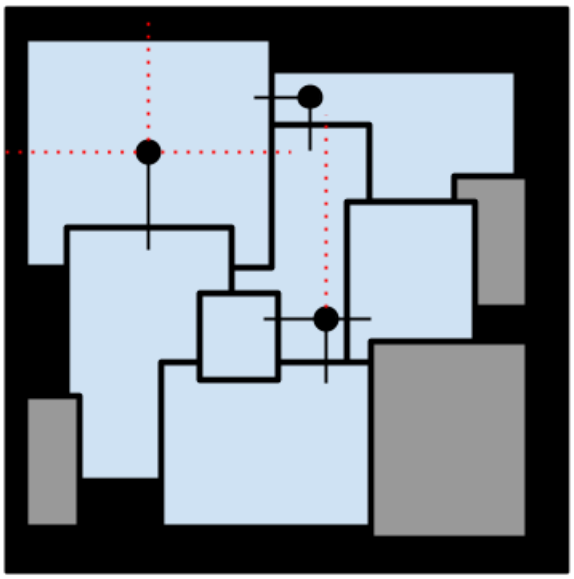

Figure 5: A second set of focal points are created from the successful branches of the first focal points.

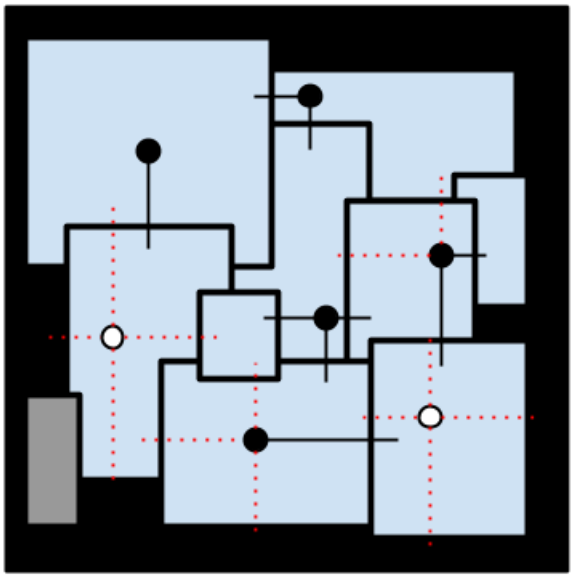

Figure 6: A second room added to the example environment, notice how the bottom left of the first room has been overwritten.

The final process involves iterating through each voxel in the environment. If the voxel has a label of "Accessible", then it is deleted. This removes all the space created by

the rooms and connections. The final result is a multi-connected environment, constructed out of rooms with non-uniform layouts and a varying number of connections such as the example in figure 7. The full process can be seen in the flow diagram in figure 8.

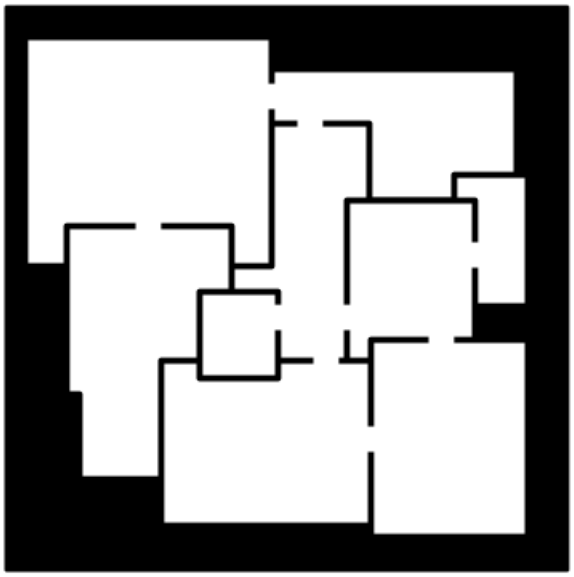

Figure 7: The final generated environment

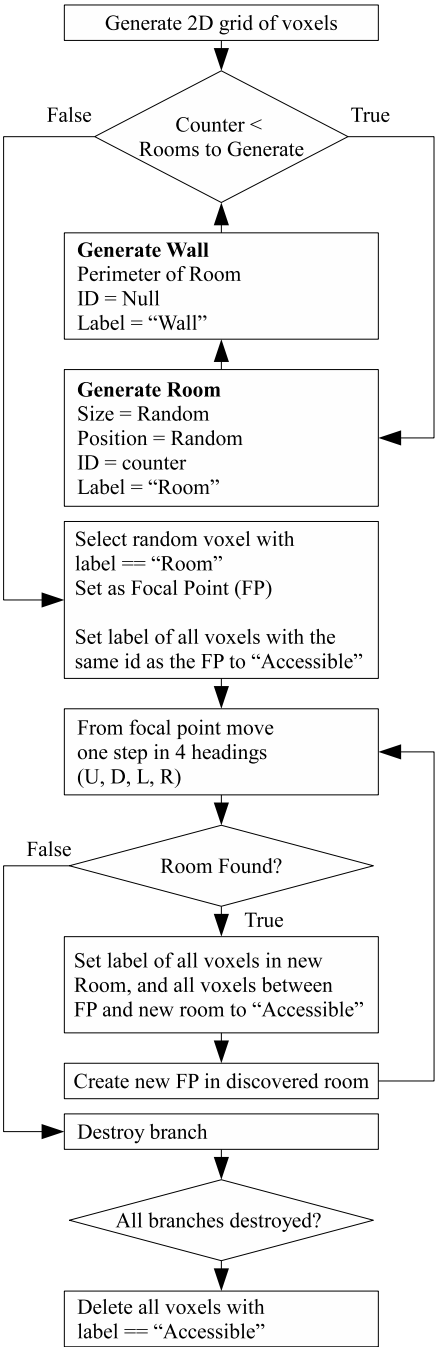

Figure 8: The algorithm as a flow diagram

4 Conclusion

Our approach generates an environment which is clearly stochastic, with a large number of possible rooms and connection configurations. By generating the rooms in a patchwork manner, we remove the possibility that all rooms will be rectangular with a standard number of connecting features. This ensures that all generated environments are truly random, and eliminates the possibility that an agent could simply learn a standard configuration.

4.1 Software

An implementation of the patchwork environment generator can be downloaded from *http://www.project-amber.co.uk/software/random-environment-generator/*.

This page contains a web-playable example of the environment generator which can be explored in first person. It also contains a download link for a Unity3D [2] project which includes the scripts to generate the environment. The script is released under a standard BSD licence for use by other projects with attribution. It is expected that we will later include the Random environment generator within a larger benchmarking software, so this release should be considered an community beta release (V0.1). Updates to the software will indexed on this page and maintained for legacy purposes.

An example generated environment and a first person rendering can be seen in figures 9 and 10.

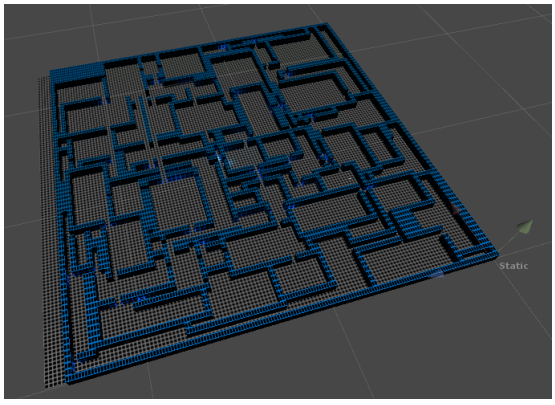

Figure 9: An example environment generated with the new algorithm

4.2 3D Rendering

As with any voxel-based algorithm draw calls can be high, especially in environments which utilise dynamic lighting. There are solutions to this issue, such as implementing occlusion culling, but this may not be suitable for all applications. Another option is to merge the voxels to create a single geometry, or replace large areas of connected voxels prefabricated units.

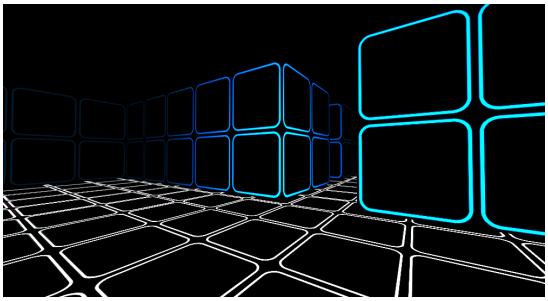

Figure 10: A first person rendering from inside the environment

However, it should be noted that most modern game engines have built-in optimisation procedures which solve most of these issues without the need for a further computational step. In large environments that we experimented with (22500 initial voxels), the Unity engine was able to render to the player at between 9 and 15 draw calls per frame (without dynamic lighting).

5 Acknowledgements

Chris Headleand and Gareth Henshall would like to thank HPC Wales for the ongoing support of their research activities.

References

[1] F. D. Lab. (Oct. 2014). Procedural dungeon generation, [Online]. Available: `http://www.futuredatalab.com/proceduraldungeon/`.

[2] David Helgason (CEO). (2004). Unity website, [Online]. Available: `http://unity3d.com/`.