Procedural Level Generation Algorithms

Tommy Bacher

August 29, 2018

Contents

1 Abstract

Procedural content generation, abbreviated PCG, is the use of algorithms and functions to generate media content for various mediums including games, movies, art, and more. In video games PCG is used to create what appears to the user as an endless number of levels, items, characters, enemies, power-ups, and abilities. This paper explores procedural content generation with regards to level design in video games by first examining Rogue, one of the first games to use PCG.

Rogue is a top-down text-based adventure game designed in 1980 by Michael Toy and Glenn Wichman. Rogue, implemented in C, allowed users to explore a seemingly endless number of dungeons by procedurally generating levels as the user played. Rogue has inspired many similar games, and has created a whole genre of games deemed Roguelikes. This paper examines Rogue's level generation algorithm, and then explores other level generation algorithms and compares them with Rogue.

This paper also discusses software developed to test the Rogue and Binary Space Partitioning Algorithm. The software allows users to generate levels based on the two level design algorithms, and highlights the similarities and differences between the two algorithms. The software is implemented in C++ utilizing GLUT, GLUI, and OpenGL.

2 Introduction

2.1 Procedural Content Generation

Procedural content generation is the use of algorithms to generate objects for video games, movies and other forms of media. The goal of procedural content generation in video games is to reduce the amount of time it takes to create various entities while also expanding upon the number of possible variations of the game that can exist. Procedurally generated content increases variety while decreasing the time needed to reach that variety [10]. Many algorithms for procedural level generation utilize concepts that are native to graph theory including vertexes, edges and graphs [1]. Unless otherwise specified, consider rooms to be vertexes and passages to be the edges that connect those vertexes.

2.2 Aims and Methods

This paper focuses on using procedural content generation to create dungeon-like levels for video games. Procedurally generated levels for games, and specifically for games involving dungeons, began with a 1980 game named Rogue [2]. This paper first examines procedural level generation in Rogue. Other methods for generating levels are explored that improve upon Rogue's system. Finally, software is explored that implements the Rogue and BSP algorithms.

Procedurally generated levels provide clear benefits over handcrafted levels, which can take weeks to create, while procedurally generated levels are generated in seconds [10]. When analyzing procedural level generation algorithms, the more potential levels an algorithm can generate the better the algorithm. Also, algorithms that produce diverse levels, where generated levels have differing structure, are better as well. In addition, since procedural level generation algorithms use random values, the lower an algorithm's average case running time, the better. Finally, the algorithm's ability to generate aesthetically pleasing levels is important, but not as important as the other factors presented.

3 Rogue: An Introduction to Level Design

3.1 Rogue History

Rogue is a top-down text-based adventure game designed in 1980 by Michael Toy and Glenn Wichman. Rogue, implemented in C, allowed users to explore a seemingly endless number

of dungeons by procedurally generating levels as the user played [3]. Rogue's system for generating levels at runtime led to the creation of other algorithms and an entire genre of games known as Roguelikes [2]. An example of what a Rogue level might look like during play can be seen in figure 1.

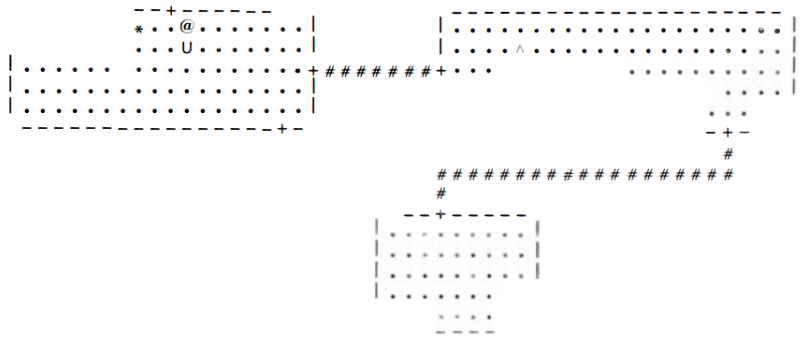


Figure 1: Rogue Level During Play [3].

3.2 Level Generation

The Rogue level generation algorithm comes from the source code for the 1980 version of Rogue [9]. Rogue generates levels through two function calls. The first function call is to doRooms() and the second call is to doPassages(). In the abstract, these two functions generate the rooms and passages respectively that make up the current level. These functions achieve this by dividing up the playable area into cells based on the maximum number of rooms to place. The doRooms() function allows up to one room to be placed within each cell with a maximum size such that the room cannot overflow into another cell. The doPassages() function creates a graph out of the existing rooms and then connects them to create a connected graph. By using this method, the Rogue algorithm guarantees that rooms do not overlap and also that there is a path from each room to every other room. Perhaps the most pressing issue with this form of level generation in terms of room creation and placement is the reduction in variety. Since each room is constrained to its own cell, each level maintains a degree of similarity which is opposed to the purpose of procedural content generation; increased variety.

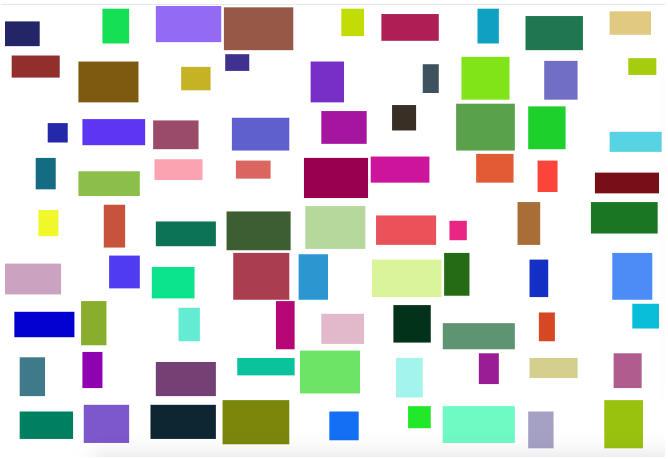


Figure 2: Rooms generated with the Rogue Algorithm.

3.3 doRooms()

Broadly, the doRooms() function divides the play area into a grid of cells, then it randomly places a room into each cell, then each room is drawn to the screen. Each room has two coordinates that represent the four total relevant properties. These coordinates are maximum and position, each having an x-value and y-value. The position represents, in relation to the origin of the play area, the room's upper left corner. The maximum parameter is the value that must be added to position to find the bottom right corner of the room. To create the cells within the grid, the function determines the number of necessary cells by taking the square root of the maximum number of rooms to be generated and rounding up. Therefore, the function guarantees that the number of cells is greater than or equal to the maximum number of rooms to be generated, but if the maximum number of rooms is not a perfect square, the cells at the end of the grid are not be populated with rooms. The function then iterates through each room in an array of rooms and finds the corresponding cell such that cell number one corresponds to room one and cell two corresponds to room two. Then, the function finds a random location in the cell which is defined to be the position of the room, and generates a random value which is less than the space remaining in the cell to define as the maximum. The final step is to draw each room into the play space. The code that implements this process is included in the appendix.

3.4 doPassages()

The second function relevant to Rogue's level generation is the doPassages() function that connects the rooms generated with the doRooms() function. doPassages() utilizes simple

graph theory to connect the rooms, first by mapping out the rooms as vertexes in a graph, and then by making sure that each room is connected by at least one passage, which is an edge, to another room such that the graph is connected. A connected graph is a graph where, starting from any vertex, you are able to move along edges to reach every other vertex [1]. Once a connected graph has been generated, for the sake of providing variety, the function attempts to generate a certain number of random connections. It is important to remember that passages are considered to be bidirectional edges, therefore the function is working on undirected graphs. doPassages() utilizes a few additional functions to create and connect the graph.

The first function is the mapPotentialConnections() function that looks at each cell in the play area and determines which other cells it is connected to. A cell should have a potential connection to another cell if they are either in the same row or in the same column of the array. This is achieved by iterating through the list of cells and creating a potential connection between the cell to the current cell's east and the cell to the current cell's south. Since this function iterates through all the cells, it ignores the final cell which has no east and south neighbors. Also, middle cells, which are cells that have neighbors on more than two sides, obtain their north and west connections from when their north and west neighbors connect to their south and east neighbors respectively.

The second function is the connect() function which draws the connection between two rooms. Once an abstract connection has been created between two rooms, connect() determines where in the place space to play the passage. First, the function determines which walls of each room being connected are facing each other, and then it determines a point along the starting room's wall to be the initial point and a point along the ending room's wall to be the end point. The function then determines a location between the two points to be the turning point. The function then creates passages from the initial point to the turning point, and then it changes the direction of the passages it is placing. It continues placing passages until the current point is lined up with the end point. Finally, it connect the current point to the end point with passages.

3.5 Analysis

The Rogue level generation algorithm is efficient, simple, and displays the potential of procedurally generated levels, but lacks variety. In terms of potential number of levels, the Rogue algorithm has only small variety in the rooms based on size and placement within their cell because rooms are always perfectly rectangular and there is always exactly as many rooms

as the maximum number of rooms. Also, since connections between rooms can only occur if the cells within which the rooms reside are adjacent, the connections between rooms is limited. While together the small amount of variety that is added by the randomness in rooms and passages is valuable, this algorithm consistently produces similar levels which is not useful in terms of procedural content generation as the purpose of procedural content generation is variety.

The average running time of the Rogue algorithm consists of the average running time of doRooms() and doPassages() combined. Since doRooms() repeats the room generation process for n times where n is the maximum number of rooms, doRooms() runs in average case linear time. doPassages() runs in average case dn time because it must make n connections and each connection occurs d time where d is the average distance between two rooms. Therefore, the average running time for the Rogue algorithm is dn^2, but d and n are inversely related in the sense that as n increases d will decrease because the screen will fill with more rooms. The result is that the more rooms the algorithm has to generate the worse the runtime because more connections need to be drawn.

4 Procedural Level Design Methods

4.1 Binary Space Partitioning

4.1.1 Algorithm

The Binary Space Partitioning, BSP, algorithm utilizes a similar grid and cell system as the Rogue algorithm, but seeks to increase variety in the placement and size of rooms [10]. The BSP algorithm creates an abstract representation for the play space through a tree called a BSP tree. The root node in the tree is the entire play space, which is the initial cell. The algorithm then divides up the root cell along a randomly chosen vertical or horizontal line such that neither of the two new cells are too small. These new cells are added to the tree as children of the root nodes since they were created by dividing up the root node. One of the new cells is chosen at random and a check is run to make sure that dividing up the cell does not produce a cell below minimum acceptable cell size, then the process is repeated. This process can be seen in figure 3 and an implementation is included in the appendix. Once partitioning is finished, all leaf nodes are assigned a room of random size within the bounds of the cell. At this point in the algorithm the rooms have been randomly generated to make up the dungeon. From here, the rooms are connected such that if two rooms share a parent

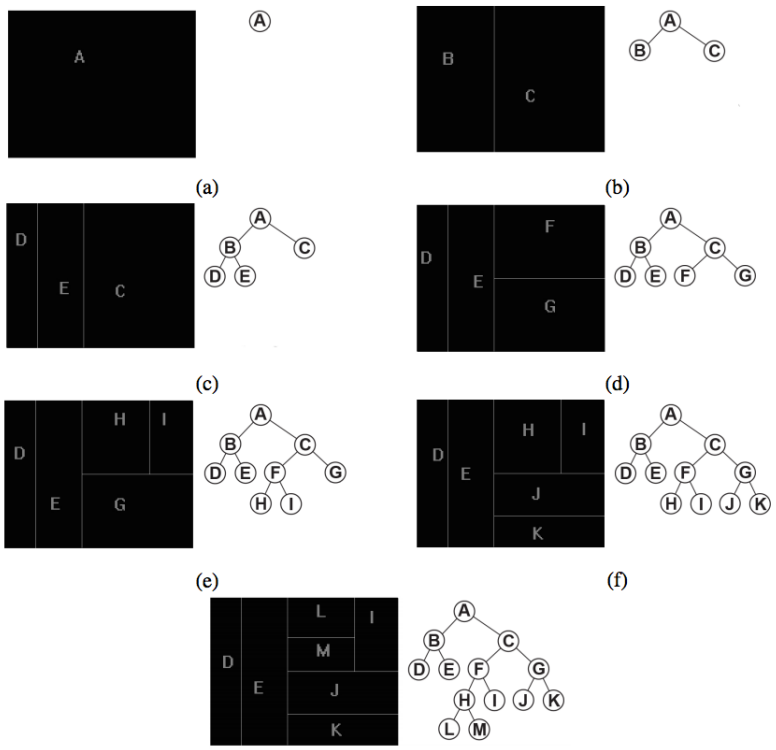


Figure 3: The following represents the division of space that occurs when running the BSP algorithm [10].

node they are connected, and then tracing this up the tree as seen in figure 4.

4.1.2 Analysis

The BSP algorithm improves on the Rogue algorithm. Unlike the Rogue Algorithm, there is no opportunity for unused space where a cell doesn't have a room within it. Also, the BSP algorithm allows for a much greater variety in room placement because the location of cells is randomized and the location of the rooms within those cells are also randomized. The BSP algorithm also constructs a BSP tree which can be used to structure level progression by starting the player in the far left or right leaf node's cell, and placing the exit in the opposite leaf node's cell, and placing increasingly difficult challenges in the leaf nodes' cells in between [10].

The BSP algorithm has a large amount of overhead because it utilizes tail-end recursion to construct the paths between rooms. In worst case scenarios, the stack of recursive calls will reach the height of the BSP before any calls are removed from the stack. The BSP algorithm runs in average case ldn^2 time where l is the number of leaf nodes, d is the distance between

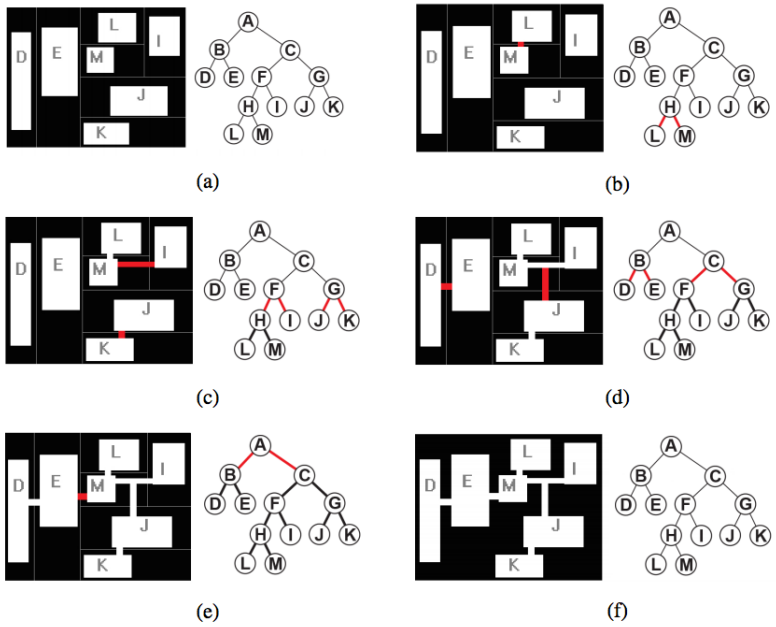


Figure 4: Here you can see the placement of rooms and connection that happens after the space has been partitioned [10].

rooms when connecting them with passages, and n is the total number of nodes. Since n nodes need to be generated, and all the leaf nodes need to have rooms generated in them it takes nl time to generate the BSP structure. The passages are constructed in dn time, but unlike Rogue we have no guarantee about a relationship between d and n. Obviously, the average running time of the Rogue algorithm is, for large n, faster by a magnitude of l which is less than n, but the BSP algorithm not only has more variety it also creates the level structure and those two factors outweigh average runtime since n is usually small.

4.2 Imperfect Graphs

4.2.1 Nystrom Algorithm

The next algorithm is the Nystrom algorithm named after the discoverer of the algorithm [7]. The Nystrom algorithm starts with similar goals as the other algorithms. It seeks to create a layer of rooms, and then create passages that connect those rooms, but Nystrom also wants to ensure that there are cycles in the graph of the level because cycles create more variety in level generation [7]. It is important to note that walls of rooms are a crucial part of this algorithm and therefore the representation of this algorithm differs slightly from previous algorithms, but the overall concepts remains the same.

The Nystrom Algorithm considers the play space to be either empty space or filled in space, and initially the entire area is filled in space. Then it starts by trying to add room-sized empty space a certain number of times. Anytime that the newly added empty space would overlap already existing empty space, the new empty space is not added, so rooms do not overlap. Therefore, it is impossible to guarantee that a certain number of rooms exist once the algorithm finishes. Next, the Nystrom algorithm uses mazes to create the passages between rooms. Mazes, for the purpose of the Nystrom Algorithm, are connected empty space. A maze is imperfect if there exist cycles in the maze. The Nystrom Algorithm completely empties the space between rooms with imperfect mazes. Then, it maps all potential connections between empty spaces where potential connections exist if there is only one spot of filled in space separating two empty spaces. For each adjacent empty spaces, if the empty spaces are not already connected, a random filled in space is made empty to connect the two empty spaces. Finally, to reduce the amount of empty space that exists, the empty space of mazes are filled in if three of the four surrounding spaces are filled in, and this process is repeated until there are no more empty spaces which have three adjacent filled in spaces. The result of the Nystrom algorithm can be seen in figure 5.

4.2.2 Analysis

The Nystrom algorithm is the largest departure from the Rogue algorithm. It does not use the concept of cells, and it purposely tries to create imperfect levels with cycles if the level were graphed. Therefore, the Nystrom algorithm has a very large potential number of levels that it can create with a great deal of variety in room placement and location. Furthermore, if the algorithm is augmented to be able to create rooms that are not rectangles, the variety in rooms could increase as well. In terms of variety Nystrom severely outpaces Rogue, but, unlike Rogue, Nystrom cannot guarantee a certain number of rooms are created. Also, Nystrom

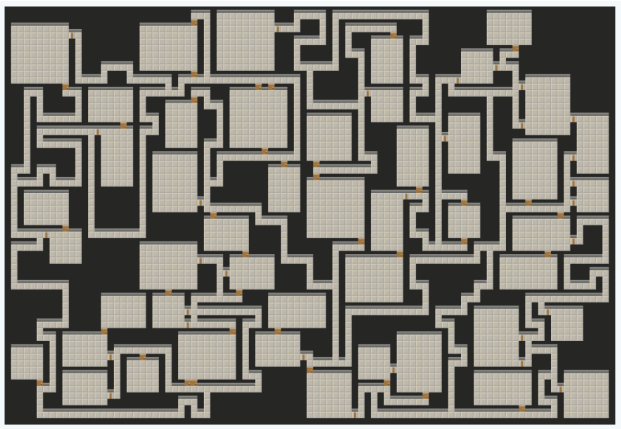


Figure 5: Finished Nystrom Algorithm Level [7]

runs slowly in comparison to both Rogue and BSP. The process of generating passages takes longer the larger the play area is because each square of the play area must be traversed to carve out the maze that makes up the passages. Also, the algorithm must undo some of the work done in creating the passages to avoid having a play area of only passages and rooms, and the result is repetition that increase runtime. Also, in worst case scenarios, the algorithm could generate a dungeon with only a few rooms. Overall, the Nystrom algorithm is a departure from the Rogue algorithm both in the levels it generates, which are detailed and complex, but also in terms of runtime.

4.3 Random Walks

4.3.1 The Digger Algorithm

The final algorithm is an agent-based approach to level generation through the use of a digger or walker. In the field of graph theory there is the concept of a random walk which is where a vertex in a graph is selected as the starting point and then for a certain number of times a random neighbor of the current vertex is selected and the walker moves to that vertex [6]. In procedural level generation, a digger or walker is an agent who empties out the level by walking and randomly placing empty space [10]. First, the algorithm selects a random location as the starting point for the digger and the location where the digger starts is set as empty space. The algorithm selects a random direction and the digger moves a predetermined or random distance in that direction while leaving empty space wherever he walks. Every time the digger finishes a move, the algorithm generates a random number to determine if a room, which is simply a larger area of empty space, is placed. Every time the algorithm does not place a room the chance to place a room on the next iteration is increased. This process is repeated until a certain limit, either time or space, is reached.

4.3.2 Analysis

The digger algorithm functions as a micro based approach in comparison to the macro based approach of Rogue [10]. Unlike the Rogue algorithm, the digger has no concept of what a level should look like, and therefore the result is often more chaotic in nature with rooms and passages intersecting each other as seen in figure 6. While it is possible to include checks in the digger algorithm to reduce the chance of overlapping rooms and passages as implemented in figure 7, the advantage of the digger algorithm is that the levels end up appearing more natural. Also, since the digger must traverse the dungeon to generate it,

Figure 6: Natural Digger Algorithm Result[10].

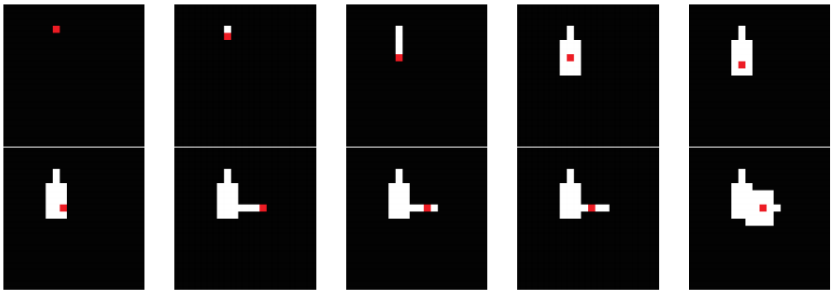
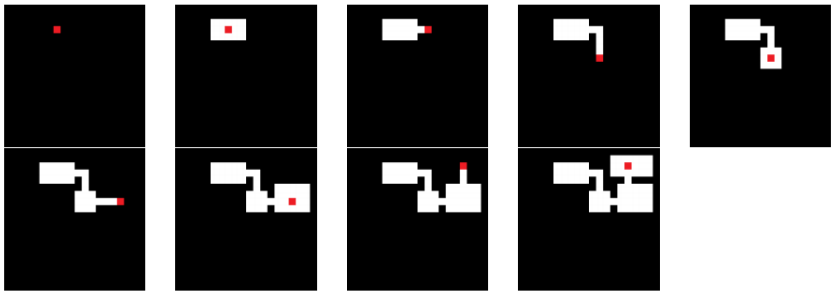


Figure 7: Smart Digger Algorithm Result[10].



there is no need to check or guarantee for connectedness between rooms since the digger had to navigate to them. In terms of runtime, the digger algorithm produces a wide variety or runtimes depending on the result of the random number rolls associated with moving the digger. In general, the larger the play area, the more likely the digger is to retrace it's steps and therefore it will take longer to create a full level, but to avoid this a time limit is put on the digger. The levels that result from digger algorithms are chaotic, and very different than any of the other algorithms examined.

5 Level Design Visualized

The goal of the software developed alongside this paper is to explore in detail two of the algorithms that this paper examines. Therefore, the Rogue algorithm and the BSP algorithm were chosen because of how the Rogue algorithm's use of cells influenced the BSP algorithm. The Rogue algorithm implemented in this software is similar to the original algorithm used in the Rogue video game, but was expanded to function for varying maximum number of rooms.

The software is implemented in C++ and uses GLUT, OpenGL, and GLUI to handle the graphical aspects of the software. The GLUT library handles the window creation and

management, while GLUI, an extension of GLUT, handles the creation of user interface elements in GLUT windows [4] [8]. OpenGL is used to draw in the GLUT window to render the rooms and passages necessary to explore the implemented algorithms [5].

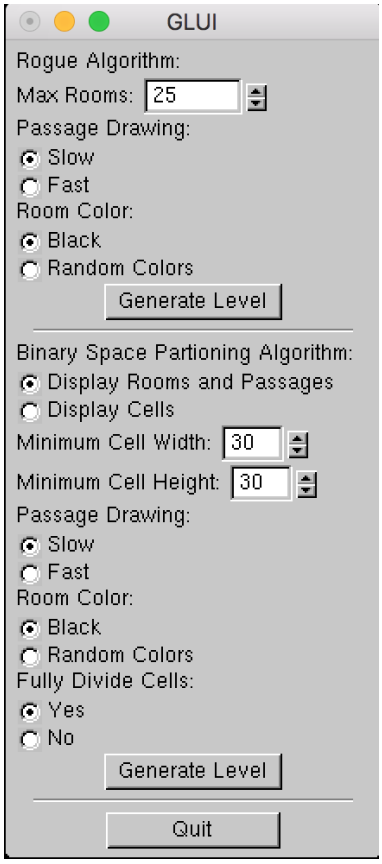


Figure 8: Software Options

The options window seen in figure 8 is created entirely using GLUI. GLUI makes use of live variable which attach to UI elements, such as the spinners or radio buttons, so that when the UI element is changed by a user the live variable associated with it is updated [8]. GLUI also handles all UI formatting by positioning each element in the window relative to when it was instantiated such that the earlier an element was instantiated the more to the top of the window it will appear [8]. Through the use of columns and rollout menus, it is possible to have greater control over the placement of GLUI elements. The options are divided such that those that are within the Rogue algorithm box only affect the running of the Rogue algorithm, and those that are within the BSP box only affect the running of the BSP algorithm. The code for setting up the options window using GLUI is in the appendix.

The BSP algorithm implemented seeks to exhibit its advantages over the Rogue algorithm. Unlike Rogue, with BSP you are guaranteed to avoid overlapping passages as a result of the way passages can only be drawn between cells which share a parent node. Even though the BSP algorithm utilizes perfect passage drawing in the sense that you are guaranteed to go from the center of one room to the center of another room, the variety in cell location, size, and placement creates much greater variety than in the Rogue algorithm. Furthermore, it is clear that the BSP algorithm creates levels with an obvi-

ous progression, and that very little space is left unused both of which are crucial advantages of the BSP algorithm that become apparent after various tests with the software.

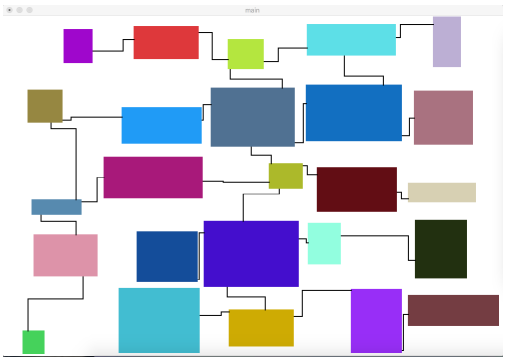


Figure 9: Rogue Example

When either of the generate level buttons are clicked, the algorithm to which that button is connected runs. For both Rogue and BSP, the rooms are generated as OpenGL quads with large sizes and are also stored in an array for use later. The passages are constructed using OpenGL quads that are placed next to each other. Also, each point on the screen where a passage quad is placed is designated as being part of a passage in an array that can be used later. If slow passage drawing is selected, then each quad of a passage shows up one at a time showing how the algorithm creates the passage, but if fast is selected then all the passages appear at once allowing the algorithms to run many times in quick succession. Room color can either be chosen to be black, the same color as passages, or each room can be assigned a random color. Also, it is possible to run the BSP algorithm and show only the cells generated by the algorithm which shows how division of the play area occurs.

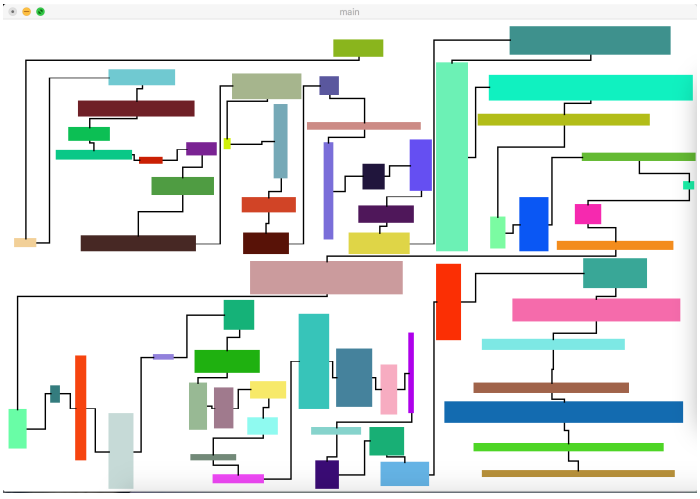


Figure 10: BSP Example

6 Conclusion

The result of the analysis of level generation algorithms through procedural content generation shows that, depending on what sort of level you want to generate and what sort of metric you want to use to evaluate the algorithms, many of the algorithms already discussed are strong candidates for use in larger systems. The Rogue algorithm, although simple, is consistent and fast and therefore has value when consistency is desired. If there is a need for organization in such a way that the user is familiar with each level upon seeing it, then the BSP algorithm or the Nystrom algorithm provide clear advantages over the digger, but if there is a desire for a naturally generated level then the digger approach may be best. Each algorithm takes an aspect of the Rogue algorithm and builds on it to generate a more diverse and powerful algorithm. The resulting algorithms are good at filling specialized roles while producing more variety in potential levels than the Rogue algorithm.

Appendix

```
/*
 *  Creates the cells and rooms for rogue algorithm
 */
void doRooms(){
    for (int j = 0; j < maxRooms; j++) {
        rooms[j].pos.x = 0;
        rooms[j].pos.y = 0;
        rooms[j].max.x = 0;
        rooms[j].max.y = 0;
    }

    //Generate the divisor number based on maxRooms, rounding up to the nearest integer
    int divisor = ceil(sqrt(maxRooms));

    int topX, topY;
    int bszeX = COLS / divisor;
    int bszeY = ENDLINE / divisor;
    int maxX, maxY, posX, posY;

    for (int i = 0; i < maxRooms; i++) {
        /*
         * Find upper left corner of box that this room goes in
         */
        topX = (i % divisor) * bszeX + 1;
        topY = i / divisor * bszeY;


        /*
         * Find a place and size for a random room
         */
        do {
            maxX = (rand() % (bszeX - int(21 - sqrt(maxRooms)))) + int(21 - sqrt(maxRooms));
            maxY = (rand() % (bszeY - int(21 - sqrt(maxRooms)))) + int(21 - sqrt(maxRooms));
            posX = topX + rand() % (bszeX - maxX);
            posY = topY + rand() % (bszeY - maxY);

```

```
37
38          } while (posY == 0);
39
40          rooms[i].pos.x = posX;
41          rooms[i].pos.y = posY;
42          rooms[i].max.x = maxX;
43          rooms[i].max.y = maxY;
44
45      }
46
47      displayRooms(bszeX, bszeY, divisor);
48
49  }
```

```
1  //Driver for BSP algorithm
2  void bspAlgo(int dispMode){
3
4      bsp = new cellNode[2000];
5      bspRooms = new room[2000];
6
7      bsp[ROOT].c.me.max.x = COLS;
8      bsp[ROOT].c.me.max.y = ENDLINE;
9      bsp[ROOT].c.me.pos.x = 0;
10     bsp[ROOT].c.me.pos.y = 0;
11     bsp[ROOT].parent = -1;
12     bsp[ROOT].leftChild = -1;
13     bsp[ROOT].rightChild = -1;
14
15     nextEmpty = ROOT + 1;
16
17     bspRecursive(ROOT);
18
19
20     //Print to screen rooms
21     if (dispMode == 0){
22         displayBSP();
23         bspSimpleConnRooms();
24         if(slowbsp == 1)
25             glutSwapBuffers();
26     }
```

```
27      //print to screen the nodes
28      else{
29          displayNodes();
30      }
31
32      delete[] bspRooms;
33      delete[] bsp;
34      currentRooms = 0;
35      // Deconstruct the bspTree, could copy out if not just drawing
36  }
37
38  //Recursive Function to generate BSP
39  int bspRecursive(int curr){
40
41      //check = 0, then either can be divided
42      // check = 1, then only y can be divided
43      // check = 2, only x can be divided
44      // check = 3 none can be divided
45      int check, randVal;
46
47      if (cellWidth(curr) < (minCellWidth + 5) && cellHeight(curr) < (↩
            minCellHeight + 5)) {
48          check = 3;
49      }
50      else if(cellWidth(curr) < (minCellWidth + 5)){
51          if(bspFullDivideMode == 0)
52              check = 1;
53          else
54              check = 3;
55      }
56      else if(cellHeight(curr) < (minCellHeight + 5)){
57          if(bspFullDivideMode == 0)
58              check = 2;
59          else
60              check = 3;
61      }
62      else
63          check = 0;
64
65      if(check == 3){
66          return bspGenerateRoom(curr);
```

```
67      }
68      if (check == 1) {
69          //Exclude due to similarity to check == 0
70      }
71      else if (check == 2)
72      {
73          //Excluded due to similarity to check == 0
74      }
75      else{
76          // rand works on either width or height, so pick one
77          // we will consider 0 to be horizantol
78          // and 1 to be vertical
79          randVal = rand() % 2; // 0 -> 1
80          if (randVal == 0) {
81              //horiz
82              int xLine;
83              xLine = rand() % (bsp[curr].c.me.max.y);
84
85              if (xLine > (bsp[curr].c.me.max.y - minCellHeight)) {
86                  xLine = bsp[curr].c.me.max.y / 2;
87              }
88              else if(xLine <= (minCellHeight)){
89                  if (bsp[curr].c.me.max.y >= (minCellHeight * 2)) {
90                      xLine = minCellHeight;
91                  }
92                  else
93                      xLine = bsp[curr].c.me.max.y / 2;
94              }
95              //Left Child
96              bsp[nextEmpty].c.me.pos.x = bsp[curr].c.me.pos.x;
97              bsp[nextEmpty].c.me.pos.y = bsp[curr].c.me.pos.y;
98              bsp[nextEmpty].c.me.max.x = bsp[curr].c.me.max.x;
99              bsp[nextEmpty].c.me.max.y = xLine;
100             bsp[nextEmpty].parent = curr;
101             bsp[curr].leftChild = nextEmpty;
102             bsp[nextEmpty].leftChild = -1;
103             bsp[nextEmpty].rightChild = -1;
104
105             nextEmpty++;
106
107             //Right Child
```

```
108             bsp[nextEmpty].c.me.pos.x = bsp[curr].c.me.pos.x;
109             bsp[nextEmpty].c.me.pos.y = xLine + bsp[bsp[curr].leftChild].c.↩
                    me.pos.y;
110             bsp[nextEmpty].c.me.max.x = bsp[curr].c.me.max.x;
111             bsp[nextEmpty].c.me.max.y = bsp[curr].c.me.max.y - xLine;
112             bsp[nextEmpty].parent = curr;
113             bsp[curr].rightChild = nextEmpty;
114             bsp[nextEmpty].leftChild = -1;
115             bsp[nextEmpty].rightChild = -1;
116 
117             nextEmpty++;
118 
119             randVal = rand() % 2; // 0 -> 1
120             if (randVal == 0){
121                 bspRecursive(bsp[curr].leftChild);
122                 bspRecursive(bsp[curr].rightChild);
123                 return 1;
124             }
125             else{
126                 bspRecursive(bsp[curr].rightChild);
127                 bspRecursive(bsp[curr].leftChild);
128                 return 1;
129             }
130 
131         }
132         else if(randVal == 1){
133             //vert
134             int yLine;
135             yLine = rand() % (bsp[curr].c.me.max.x);
136 
137             if (yLine > (bsp[curr].c.me.max.x - minCellWidth)) {
138                 yLine = bsp[curr].c.me.max.x / 2;
139             }
140             else if(yLine <= (minCellWidth)){
141                 if (bsp[curr].c.me.max.x >= (minCellWidth * 2)) {
142                     yLine = minCellWidth;
143                 }
144                 else
145                     yLine = bsp[curr].c.me.max.x / 2;
146             }
147 
```

```
148             //Left Child
149             bsp[nextEmpty].c.me.pos.x = bsp[curr].c.me.pos.x;
150             bsp[nextEmpty].c.me.pos.y = bsp[curr].c.me.pos.y;
151             bsp[nextEmpty].c.me.max.x = yLine;
152             bsp[nextEmpty].c.me.max.y = bsp[curr].c.me.max.y;
153             bsp[nextEmpty].parent = curr;
154             bsp[curr].leftChild = nextEmpty;
155             bsp[nextEmpty].leftChild = -1;
156             bsp[nextEmpty].rightChild = -1;
157 
158             nextEmpty++;
159 
160             //Right Child
161             bsp[nextEmpty].c.me.pos.x = yLine + bsp[bsp[curr].leftChild].c.↩
                    me.pos.x;
162             bsp[nextEmpty].c.me.pos.y = bsp[curr].c.me.pos.y;
163             bsp[nextEmpty].c.me.max.x = bsp[curr].c.me.max.x - yLine;
164             bsp[nextEmpty].c.me.max.y = bsp[curr].c.me.max.y;
165             bsp[nextEmpty].parent = curr;
166             bsp[curr].rightChild = nextEmpty;
167             bsp[nextEmpty].leftChild = -1;
168             bsp[nextEmpty].rightChild = -1;
169 
170             nextEmpty++;
171 
172             randVal = rand() % 2; // 0 -> 1
173             if (randVal == 0){
174                 bspRecursive(bsp[curr].leftChild);
175                 bspRecursive(bsp[curr].rightChild);
176                 return 1;
177             }
178             else{
179                 bspRecursive(bsp[curr].rightChild);
180                 bspRecursive(bsp[curr].leftChild);
181                 return 1;
182             }
183         }
184     }
185 
186     //Should never reach this case
187     return -1;
```

```
188 }
```

```
1     /* UI Options Code */
2     GLUI *glui = GLUI_Master.create_glui("GLUI");
3     glui->add_statictext("Rogue Algorithm: ");
4     GLUI_Spinner *maxRoomsRogue_spinner = glui->add_spinner("Max Rooms: ", ↩
          GLUI_SPINNER_INT, &maxRooms);
5     maxRoomsRogue_spinner->set_int_limits(4, 324);
6     maxRoomsRogue_spinner->set_speed(.05);
7
8     glui->add_statictext("Passage Drawing: ");
9     // the radio buttons
10    rogueRadioGroup = glui->add_radiogroup(&slow);
11    glui->add_radiobutton_to_group(rogueRadioGroup, "Slow");
12    glui->add_radiobutton_to_group(rogueRadioGroup, "Fast");
13
14    glui->add_statictext("Room Color: ");
15    rogueRoomColorGroup = glui->add_radiogroup(&rogueRoomColor);
16    glui->add_radiobutton_to_group(rogueRoomColorGroup, "Black");
17    glui->add_radiobutton_to_group(rogueRoomColorGroup, "Random Colors");
18
19    glui->add_button("Generate Level", genRoomButton, glui_cb);
20
21    glui->add_separator();
22
23    glui->add_statictext("Binary Space Partioning Algorithm: ");
24    bspDispMode = glui->add_radiogroup(&bspDispModeVar);
25    glui->add_radiobutton_to_group(bspDispMode, "Display Rooms and Passages"↩
          );
26    glui->add_radiobutton_to_group(bspDispMode, "Display Cells");
27
28    GLUI_Spinner *minCellWidth_Spinner = glui->add_spinner("Minimum Cell ↩
          Width: ", GLUI_SPINNER_INT, &minCellWidth);
29    minCellWidth_Spinner->set_int_limits(30, 125);
30    minCellWidth_Spinner->set_speed(.05);
31
32    GLUI_Spinner *minCellHeight_Spinner = glui->add_spinner("Minimum Cell ↩
          Height: ", GLUI_SPINNER_INT, &minCellWidth);
33    minCellHeight_Spinner->set_int_limits(30, 125);
34    minCellHeight_Spinner->set_speed(.05);
```

```
35
36      glui->add_statictext("Passage Drawing: ");
37      // the radio buttons
38      bspSpeed = glui->add_radiogroup(&slowbsp);
39      glui->add_radiobutton_to_group(bspSpeed, "Slow");
40      glui->add_radiobutton_to_group(bspSpeed, "Fast");
41
42      glui->add_statictext("Room Color: ");
43      bspRoomColorGroup = glui->add_radiogroup(&bspRoomColor);
44      glui->add_radiobutton_to_group(bspRoomColorGroup, "Black");
45      glui->add_radiobutton_to_group(bspRoomColorGroup, "Random Colors");
46
47      glui->add_statictext("Fully Divide Cells: ");
48      bspDivideMode = glui->add_radiogroup(&bspFullDivideMode);
49      glui->add_radiobutton_to_group(bspDivideMode, "Yes");
50      glui->add_radiobutton_to_group(bspDivideMode, "No");
51
52      glui->add_button("Generate Level", genBSPButton, glui_cb);
53
54      glui->add_separator();
55      // quit button
56      glui->add_button("Quit", 0, (GLUI_Update_CB)exit);
```

References

[1] Bondy, J. A., and Murty, U. S. R. *Graph theory.* No. 244 in Graduate texts in mathematics. Springer, New York, 2008.

[2] Daivd L. Craddock. *Dungeon Hacks: How NetHack, Angband, and Other Roguelikes Changed the Course of Video Games.* Press Start Press, Canton, Ohio, 2015.

[3] Dewdney, A. Computer recreations. *Scientific American* (Feb. 1985), 18–21.

[4] Khronos Group. GLUT - The OpenGL Utility Toolkit. `https://www.opengl.org/resources/libraries/glut`, 2016. Accessed: 2017-04-30.

[5] Khronos Group. OpenGL API. `https://www.opengl.org/documentation`, 2016. Accessed: 2017-04-30.

[6] Losasz, L. Random walks on graphs: A survey. *Combinatorics, Paul Erdös is eighty 2* (1993), 1–46.

[7] Nystrom, B. Rooms and Mazes: A Procedural Dungeon Generator. `http://journal.stuffwithstuff.com/2014/12/21/rooms-and-mazes/`, Dec. 2014. Accessed: 2017-02-16.

[8] Rademacher, P. GLUI User Interface Library. `http://glui.sourceforge.net`, 2006. Accessed: 2017-04-30.

[9] Ritzl, B. The Rogue Archive. `https://britzl.github.io/roguearchive/`. Accessed: 2017-02-16.

[10] Shaker, N., Togelius, J., and Mark J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research.* Springer, Switzerland, 2016.