2D Procedural Dungeon Generation

Jessica Baron and Aurora Silverwood

University of North Carolina - Wilmington
CSC 380 Fall 2015
Abstract

The purpose of this study is to recreate and examine multiple algorithms for procedurally generating 2D dungeon maps.

Maps consist of traversable rooms and corridors, represented by connected floors and bounded by untraversable walls.

The algorithms are divided as room-generating algorithms and corridor-generating algorithms. Each algorithm pair is written in Java using custom graphs, implemented with two-dimensional arrays.

The efficiency of each algorithm pair is measured in Theta notation, which is reflected by a recorded time measurement taken in milliseconds to create each map.

Each algorithm, has its strengths and weaknesses and can be used in combination for different desired results.
Formal Problem Statement

Procedurally generate a 2D video game dungeon map with minimal computational iterations measured in Theta Notation, which is composed of rectangular "rooms" that are connected by narrower "corridors" as a graph of \( w \times h \) vertices, with a width of \( w \) vertices and a height of \( h \) vertices.

Each vertex is assigned a state that is either "on" or "off" ("on" forming the traversable rooms and corridors, and "off" forming untraversable boundaries, or "walls"). One benchmark is that the "on" vertices of the graph are connected, meaning each “on” vertex is adjacent to at least one other “on” vertex, this guarantees all traversable vertices are connected. The other benchmark is that the sparsity ratio, being defined approximately as 40\% of the graph vertices being "on" and 60\% being "off," within a 10\% deviation from that ratio.

A "room" is a subset of the set of graph vertices, consisting of "on" vertices that are adjacent to one another, forming a subgraph of their own of a width and a height; width and height each constrained by certain minimum and maximum values. A "corridor" is a subgraph of a width and a height, where either the width or the height has to be equal to one. Corridors are generated after rooms have been generated connecting two rooms at a time. A room cannot be connected to another room, but it must be connected to at least one corridor. A corridor must be connected to at least one other corridor or room, and if it is connected to a corridor, the corridor it is connected to must be connected to a room.
Context

*Rogue* (1980), developed by Michael Toy and Glenn Wichman is one of the first games to use procedural dungeon generation in game. Many games following it have been considered “Rogue-like.”

Due to the severely limited memory and disk space of early computers, storing large maps and artwork simply was not feasible.

Today, computer memory and processing power aren’t so limited.

Procedurally generated dungeon maps allows developers to devote more time to creating complex story and gameplay. The early games used ASCII representation, similar to our implementation. However, increased computer processing power and memory have allowed more modern games to combine procedural generation with original artwork to create more appealing and organic maps such as: Blizzard’s *Diablo*, and *Minecraft*. 
<table>
<thead>
<tr>
<th>5 Main Algorithms</th>
<th>5 Algorithm Pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>2 Room-Generating:</strong></td>
<td>Random rooms, Random-corporders</td>
</tr>
<tr>
<td>Random room placement</td>
<td>Random rooms, Drunkard’s Walk</td>
</tr>
<tr>
<td>BSP rooms</td>
<td>BSP-rooms, Random-corporders</td>
</tr>
<tr>
<td><strong>3 Corridor-Generating:</strong></td>
<td>BSP-rooms, Drunkard’s Walk</td>
</tr>
<tr>
<td>Random points connect</td>
<td>BSP-rooms, Drunkard’s Walk</td>
</tr>
<tr>
<td>Drunkard’s Walk</td>
<td>BSP-rooms, BSP-corporders</td>
</tr>
<tr>
<td>BSP corridors</td>
<td></td>
</tr>
</tbody>
</table>
Room-Generating: Random Rooms

Random rooms is a brute force room-generating algorithm. It is implemented by randomly generating a room width and height (bounded by min and max values), then picking a point \((x, y)\) on the graph. This potential room is then checked for overlapping other “on”-vertices of the graph. If not overlapping, place room. Otherwise, try same room dimensions with different coordinates (max tries = 100). We have it set to not overlap other rooms or the borders of the graph plus a space of 3 vertices horizontally and vertically.

Once these rooms are created, they are added to an array-list.
BSP works by partitioning the graph into “leafs,” starting with the entire graph as the “root”, until a split leaf has children of a defined minimum size (which is directly related to the graph’s dimensions). A bottom leaf has no children as it does not get split.

Those bottom childless leaves are assigned rooms (the max size of which is smaller than the min leaf size), which are added to an array-list in the order of which the leaves were created.
Random Rooms and Random Points Connect

This corridor-generating algorithm accesses the array-list created by the Random Rooms algorithm and connects two rooms at a time (which are two consecutive elements in the room list).

The algorithm connects rooms of this array list in a brute force manner, selecting a random vertex each bordering the two rooms and connecting them horizontally then vertically (or vice versa). Overlapping other corridors and rooms with these new corridors is ignored.
Random Rooms and Random Corridors

Examples
Random Rooms and Drunkard’s Walk

The drunkard’s walk corridor-generating algorithm accesses the array-list created by the Random Rooms algorithm.

It connects a pair of rooms, by a random side and vertex. It starts in the appropriate x and y directions towards the second point, incrementing n vertices \(n \in \mathbb{N} = \{2,3,4,5,6\}\) horizontally or vertically with each iteration. Direction is negated if path reaches too closely to the graph’s borders.

Drunkard’s will keep iterating until it reaches the other point, or when too many steps have been taken.

This one gives the most random results, from a sparse map, to a cave like map.
Random Rooms and Drunkard’s Walk

Examples
BSP Rooms and BSP Corridors

BSP corridors connects room pairs from BSP-generated rooms’ array-list.

This looks very similar to Random Room Connect, however, since the rooms in the array-list are in order, the result often looks much more orderly and linear.
BSP Rooms and BSP Corridors examples
BSP Rooms and Random Corridors

Random corridors connects room pairs from the BSP array-list.

This often looks similar to BSP rooms and corridor connect, but with more variety since the rooms are not ordered in respect to map placement in the array-list.
BSP Rooms and Random Corridors Examples
Drunken corridors connects room pairs from the BSP array-list.

Like the max room percentage parameter for the Random Rooms algorithm, the BSP Rooms algorithm has minimum Leaf dimension modifiers, which are 1/8 the max room dimensions when paired with Drunkard’s Walk (1/15 with the other two corridor algorithms).
BSP Rooms and Drunkard’s Walk Examples
Random Room Placement Θ notation

- Our specific case with Graph dimension restraints: \( \Theta(1*(1 + (n * 1) + 1 + 1) = \Theta(n) \)
- In general with no dimension caps: \( \Theta(100*(1 + (100 * (w+3)*(h+3)) + 1 + (w*h)) = \Theta(w*h) \)

- Main while loop (letting \( r \) = number of rooms): \( \Theta(1) + \Theta(r) = \Theta(r) = \Theta(100) \) in general, \( \Theta(1) \) in our case
  - generateRandomRoomSize(): \( \Theta(1) \) (handful of \( \Theta(k) \) operations)
  - while loop calling overlaps() (letting \( n \) = number of iterations of this loop, \( w \) = room width, \( h \) = room height): \( \Theta(n) + \Theta((w+3)*(h+3)) = \Theta(n*(w+3)*(h+3)) = \Theta(100*(w+3)*(h+3)) \) in general, \( \Theta(w*h) \) here
    - overlaps(): \( \Theta((w+3)*(h+3)) \) in general, \( \Theta(1) \) here
  - if not overlapping, add item to an ArrayList: \( \Theta(1) \)
  - drawRoom() (letting \( w \) = room width, \( h \) = room height): \( \Theta(w*h) \) in general, here \( \Theta(1) \)
• Our specific case with Graph dimension restraints: $\Theta(1 \text{ (tree)} + 1 \text{ (children)} + 1 \text{ (drawing)}) = \Theta(1)$

• In general: $\Theta(n \text{ (tree)} + c \text{ (children)} + w \ast h \text{ (drawing)}) = \Theta(n+c+(w\ast h))$

- BSPTree instantiation (letting $n =$ number of splits, $L =$ total number of leafs): $\Theta(n+L) = \Theta(n+2n) = \Theta(3n) = \Theta(n) \text{ in general}.$
  $\Theta(k+2k) = \Theta(3k) = \Theta(k) = \Theta(1) \text{ in our case} \ (\text{splitting the Graph down to a min Leaf size, which is at smallest graphW/15 graphH/15})$

  - splitGraph(): $\Theta(n) + \Theta(1) = \Theta(n) \text{ normally, } \Theta(1) \text{ here. (graph is split a constant number of times)}$

  - getLeafLists(): $\Theta((L \text{ or } 1)+1+1) = \Theta(L) = \Theta(2n) \text{ normally, } \Theta(1) \text{ here. (iterating through each BSPLleaf, which is twice the number of times splitGraph() ran)}$
    - adding to bottomChildren list: $\Theta(1) \ (\text{adding to ArrayList})$
    - adding to parentsList: $\Theta(1)$

- iterating through children to assign Rooms (letting $c =$ number of children): $\Theta(c) + \Theta(1) + \Theta(1) = \Theta(c) \text{ normally, } \Theta(1) \text{ here}$

  - generateRandomRoomSize(): $\Theta(1)$ operations
  - adding to lists: $\Theta(2) \text{ worst case (in our case there are never more than 20 rooms)}$

- drawRoom() for each Room (letting $w =$ room width, $h =$ room height): $\Theta(w\ast h)$
Binary Space Partition Corridors &\ notation

In our case: $\Theta(1 \ ((\text{creating each corridor part per parent})) + 1 \ ((\text{for each child})) + 1 \ ((\text{drawing rooms}))) = \Theta(1)$

General: $\Theta(1 \ ((\text{creating each corridor part per parent})) + 1 \ ((\text{for each child})) + 3c^*w^*h \ ((\text{drawing rooms}))) = \Theta(3c^*w^*h)$

- for each parent: $\Theta(n(1+n)) = \Theta(n^2)$
  $\Theta(c/2(1+4^*(c/2))$
  $= \Theta(c/2(1+2^*(c))$
  $= \Theta(c/2+c)$
  $= \Theta(c*(3/2))$
  $= \Theta(20 \ ((\text{max c value})) *3/2)$
  $= \Theta(1) \ (\text{letting parents p = c / 2, and c = # of children})$

- createACorridor(): $\Theta(1)$ (many machine operations and 1 or 2 addings to a list: $\Theta(2)$ (worst, in our case))

- for each corridor part (letting i = # of corridor parts): $\Theta(i)$ ($i_{\text{max}} = 4$ (max possible corr parts per room pair) * p)

- for each child: $\Theta(c)$

- for each room: $\Theta(3c)$ in general, $\Theta(60) = \Theta(1)$ in our case
  Updated room list size = c + i (which is 4*p = 4*(c/2)) = c + 2c = 3c = 3 * 20 ((c_{\text{max}})) = 60

- drawRoom(): $\Theta(w^*h)$
Our specific case with dimension caps: $\Theta(1(n ((picking valid side)) + 1*1 ((going along distances))) = \Theta(n+1) = \Theta(n)$

General: $\Theta(1(n ((picking valid side)) + w*h ((going along distances))) = \Theta(n+w*h) = \Theta(w*h)$

- For each room (letting $r = \text{number of rooms}$): $\Theta(r)$
  - $\Theta(64)$ (64 max is the possible rooms) = $\Theta(1)$
  - pickRandomSidePoint twice (per each room in a pair) (letting $n = \# \text{ of loops until valid side found}$): $\Theta(n+n) = \Theta(2n) = \Theta(n)$
    - isValidSide(): $\Theta(1)$
    - loop ‘til valid side: $\Theta(n)$

  - for each Vertex along horizontal/vertical distance (letting $d = \text{room } w \text{ or } h$): $\Theta(d)$
    - for each Vertex along horizontal/vertical distance (letting $d = \text{room } w \text{ or } h$): $\Theta(d)$
      - turn vertex “on”

Max horizontal/vertical distances are directly related to the Graph $w$ and $h$. Therefore, if the Graph $w$ and $h$ are capped, those distances are capped, and drawing along those distances would be constant. $O(1)$
Drunkard’s Walk & notation

Our specific case with dimension caps: \( \Theta(1 ((\text{max 64 rooms }))* (n + (1 * 1 * 1)) = \Theta(n) \)

General: \( \Theta(1* (n + (m * w*h))) = \Theta(m*w*h) \)

- for each room (letting \( r \) = number of rooms): \( \Theta(r) = \Theta(64) \) (64 max possible rooms) = \( \Theta(1) \)
- pickRandomSidePoint twice (letting \( n \) = # of loops until valid side found): \( \Theta(n+n) = \Theta(2n) = \Theta(n) \)
- while loop til hit 2\(^{nd}\) room or cap (which is graphArea/20) (letting # of loops = \( m \)): \( \Theta(m) \) in general
  \( \Theta(m_{\text{max}}) = (((\text{max height } x \text{ max width}) / 20) \)
  \( \Theta(m_{\text{max}}) = ((500*500) / 20) = \Theta(12500) = \Theta(1) \) in our case
- draw a horizontal or vertical line: \( \Theta(n) \)

  ○ max horizontal/vertical distances are directly related to the Graph w and h.

  Therefore, if the Graph w and h are capped, those distances are capped, and drawing along those distances would be constant. \( \Theta(1) \) )
**Algorithm Pairs & notation**

Random Rooms and Random Point Connect:
- In general: \( \Theta(w*h) + \Theta(w*h) = \Theta(2(w*h)) = \Theta(w*h) \)
- In our case: \( \Theta(n) + \Theta(n) = \Theta(2n) = \Theta(n) \)

Random Rooms and Drunkard’s Walk:
- In general: \( \Theta(w*h) + \Theta(m*w*h) = \Theta(m*2(w*h)) = \Theta(m*w*h) \)
  \( m = \text{iterations to find 2nd room} \)
- In our case: \( \Theta(2n) = \Theta(n) \)
  \( n = \# \text{ of loops until valid side found} \)

BSP Rooms and Random Point Connect:
- In general: \( \Theta(n+c+(w*h)) + \Theta(w*h) = \Theta(n+c+(w*h)) = \Theta(n+c+(w*h)) \)
- In our case: \( \Theta(1) + \Theta(n) = \Theta(n) \)
  \( n = \text{number of splits} \)

BSP Rooms and Drunkard’s Walk:
- In general: \( \Theta(n+c+(w*h)) + \Theta(m*w*h) = \Theta(n+c+(w*h)(1+m)) \)
- In our case: \( \Theta(1) + \Theta(n) = \Theta(n) \)

BSP Rooms and BSP Corridors:
- In general: \( \Theta(n+c+(w*h)) + \Theta(3c*w*h) = \Theta(n+c+(w*h)(1+3c)) \)
- In our case: \( \Theta(1) + \Theta(1) = \Theta(1) \)
Algorithm Run Times

![Algorithm Runtimes Graph]

- **Algorithm Pair**: R-R, R-D, BSP-R, BSP-D, BSP-Full
- **Runtime (in milliseconds)**
- **Average**, **Std. Dev.**, **Min**, **Max**

The graph compares the runtimes of different algorithm pairs, illustrating variations in performance.
Algorithm Run Times Continued

Of the rooms, BSP took the longest time to generate on average. Of the corridors, Drunkard’s took the longest to generate.
On average, we got very close to our benchmark sparsity 40 : 60 ratio
Some algorithm pairs are heavier in vertices than rooms and vice-versa. These differences are expected and is one of the elements that measures the difference in graphs.
Questions for you

1) Which algorithm pair implements a brute force approach?

2) Informally, how is overlapping checked differently in random-room placement and BSP-room placement algorithms?

3) Which corridor-generating algorithm produces the greatest variation in the “on”-vertex : “off”-vertex ratio? Which trend does this algorithm produce for the room-vertex : corridor-vertex ratio when paired with either room-generating algorithm?

4) What defines a “good” dungeon?

5) Why do the five algorithm pairs have more efficient runtimes when considering that the dimensions of the graph are limited to be within a certain range (which is from 30x30 vertices to 500x500 vertices) versus the algorithm pair runtimes when disregarding any restraints on the graph dimensions?
 Answers

1) Random room placement, random points connect

2) Random-rooms checks for overlapping by checking the location of the room and its area plus three-vertex-wide border around the room, and makes sure the room will not touch any other existing room. The BSP-rooms algorithm handles overlapping prevention by partitioning space of the graph and fitting rooms into that space. The max room size is smaller than that partitioned space, so there would never be touching or overlapping of rooms.

3) Drunkard’s walk. This algorithm can “walk” corridors sparsely through the dungeon, or it can take a long time finding destination rooms and wander all over. When paired with either random-rooms or BSP-rooms, this algorithm also tends to produce more corridors in respect to rooms.

4) An average sparsity ratio of 40-60% (on-off), connectedness, non-linear, but ultimately subjective.

5) With the graph dimensions limited to a range of certain values, many computations (such as drawing rooms and corridors) actually run in constant time at both worst and best cases rather than at, for example, width x height time.
Questions For Us?
Sources

General:
https://en.wikipedia.org/wiki/Procedural_generation

Random:
http://journal.stuffwithstuff.com/2014/12/21/rooms-and-mazes/

BSP:

Drunkard’s:

Images:
Slide 4: Michael Toy & Glenn Wichman’s Rogue & Blizzard’s Diablo I Dungeon maps
Slides 40, 41: Nintendo® - Legend of Zelda: Link between worlds concept art & Ocarina of Time, Navi

Examples of 2D dungeons:
(play) http://munificent.github.io/hauberk/
(generate) http://www.myth-weavers.com/generate_dungeon.php