Dynamically Adaptive Procedural Generation of Dungeons

Senior Project
Computer Engineering
California Polytechnic State University, San Luis Obispo

By
Cameron Thibodeaux
May 2014

Advisor
Foaad Khosmood

Table of Contents

Introduction

Procedural content generation (PCG) is a technique utilized in many productions that allows content to be created algorithmically in real time. PCG gives games an element of probabilistic variability and replayability, allowing content such as level layout and enemy spawning to vary between each playthrough. It is used in many game genres, including roguelikes like Dungeons of Dredmor (Gaslamp Games, 2011) and sandbox games like Minecraft (Mojang, 2011). PCG can also save time for designers who would otherwise have to build all levels and content from scratch. Another popular game mechanic is dynamic difficulty adjustment, which scales a game's difficulty depending on the player's actions and the desired user experience. This makes a game consistently challenging, but not impossible. One example of this is Left 4 Dead (Valve, 2008), which changes the frequency and location of enemy spawn points based on the player's performance.

However, when most games dynamically alter content, they only modify a few simple parameters such as the number of health pickups or strength of enemies. This makes the game generically easier or more difficult. One underutilized aspect of games that can greatly increase the degree of customization is level design. Jennings-Teats et al. (2010) created a 2D platformer game called Polymorph that utilized structural adjustment instead of numerical adjustment to generate levels. They found that dynamic levels generated in real time resulted in a strong correlation between difficulty and the player's skill while creating a unique gameplay experience. Similar to Polymorph, this project focuses on the dynamically adaptive generation of dungeon levels in a roguelike game named *Pearl of the World*, where the geometry and layout of individual rooms are

influenced by the player's actions throughout the game and are generated in session, as seen in Figure 1. Specifically, the game adds elements the player finds difficult, adding a challenge specific to them. By doing this, the entire dungeon is personalized for each player, creating a new world and a unique experience each playthrough.

Figure 1: A procedurally generated room in *Pearl of the World*

Background and Related Work

Procedural level generation has been the focus of many studies over the past few years. Valtchanov and Brown (2012) have developed a method of dynamic dungeon generation using an evolutionary approach, where each map segment, or "chromosome," is individually manipulated by crossover and mutation operators and is selected to be added to the final map using a fitness score. The map is represented by a tree, where each node contains a preset room shape and a door that connects it to its parent. This method of level generation can create large, intricate 2D dungeon layouts with the fitness score for some variation, but the individual rooms remain unpopulated.

While the orientation of rooms is the first essential part of creating levels, it does not include the extra personalization that the contents of a room add to the game, nor does it take the player's actions into account when determining room arrangement.

In addition to procedural level generation, using metrics to determine the player's preferences can help personalize a game. Katavić (2013) explains an experiment involving using a set of metrics to alter the layout of a first-person shooter level. These metrics included enemies killed, checkpoints reached, items collected, and number of deaths, and the combination of these metrics establishes a persona (soldier, athlete, or puzzle solver) to determine how the next level should be arranged. The experiment shows that the players reacted positively to the customized level design, demonstrating that levels created from properly recorded metrics can strengthen the game and its enjoyment. Roguelike games in particular could benefit because they have a wider variety of metrics to collect and level features to change, creating many opportunities for customization.

Design

Gameplay

Pearl of the World is a roguelike game whose objective is to escape a dungeon. The game is divided into multiple levels, each with a certain number of interconnected rooms. In each level, the player fights fast enemies and strong enemies, collects items, and explores the dungeon. Partway through the level, the player will encounter a pearl, which is needed to get to the next level but will gradually increase the chance of unpleasant events the longer the player holds it. This is meant to highlight the

personalized level design and difficulty adjustment. After the player completes several levels, the player is placed in a room with a pit and an exit and is given the option to throw the pearl in the pit before leaving. The game will end if the player destroys the pearl, but the player will be put back into the dungeon for another level if they try to leave with the pearl.

User Experience

The desired user experience is to make the player realize that holding the pearl brings misfortune and hardship without telling them explicitly or being too obvious. This is implied in each level with a gradual difficulty increase and the music slowly becoming more frantic as the player is holding the pearl. The opposite effect will occur if the player drops the pearl. On each new level, the amount of time holding the pearl in the previous level will affect how much the level geometry changes in response to the player's weaknesses. Ideally, at the end of the game, the player should want to throw the pearl in the pit and leave the dungeon without it. Otherwise, the player will have additional levels to understand the pearl's effect.

The Misfortune Parameter

In order to achieve the desired user experience, a misfortune parameter is needed to alter the difficulty of the current level. Misfortune does not affect the level's geometry, but it provides immediate difficulty adjustment to help the player more easily understand the pearl's effects.

When the player picks up the pearl, misfortune slightly increases every second until it reaches its maximum value. Similarly, when the player drops the pearl,

misfortune decreases every second until it reaches its minimum value. A sample graph of misfortune is shown in Figure 2. Misfortune is proportional to the magnitude of changes, which include enemy sight radius, enemy and player accuracy, player attack speed, the chance that a sword or shield will break on hit, and the chance that a killed enemy will drop an item.

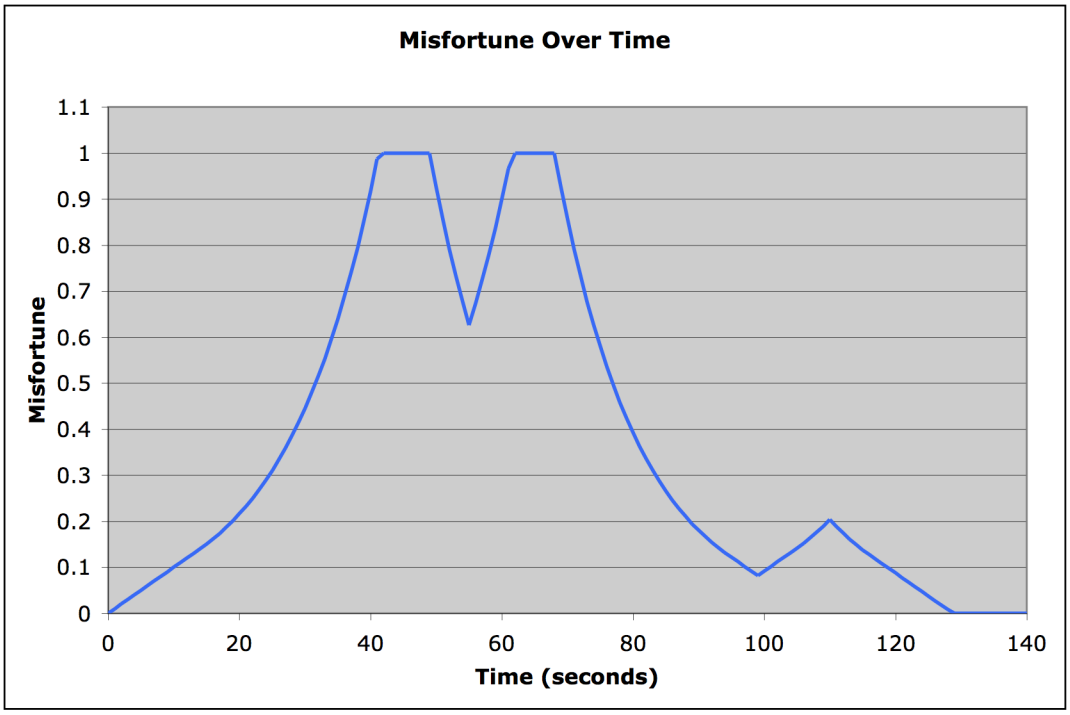

Figure 2: An example of misfortune over time

Level Layout

Unlike misfortune, level layout is affected by the total time the pearl was held in the previous level. It affects the minimum number of rooms in a level and the general orientation of the rooms. These two factors influence difficulty. For instance, a higher amount of rooms will make the player fight through more enemies while having several adjacent rooms gives the player more opportunities to escape a group of enemies. More

time holding the pearl results in more rooms and less doors per room, creating a more linear dungeon and forcing the player to fight enemies in the way. The rooms are configured in a 3D grid with adjacent rooms connected with doors or staircases, making the dungeon cohesive and consistent. In addition, the percentage of staircases taken in the previous level will inversely change the number of staircases in the current level, causing the player to change the way they explore the dungeon.

Room Generation

Room generation is the primary method of personalizing the dungeon for the player. The dimensions, number of doors, wall density, items, and enemies in each room vary depending on several metrics collected from the previous level. All room attributes are affected by the amount of time the player held the pearl, but vary otherwise. A flowchart of the room generation process is shown in Figure 3.

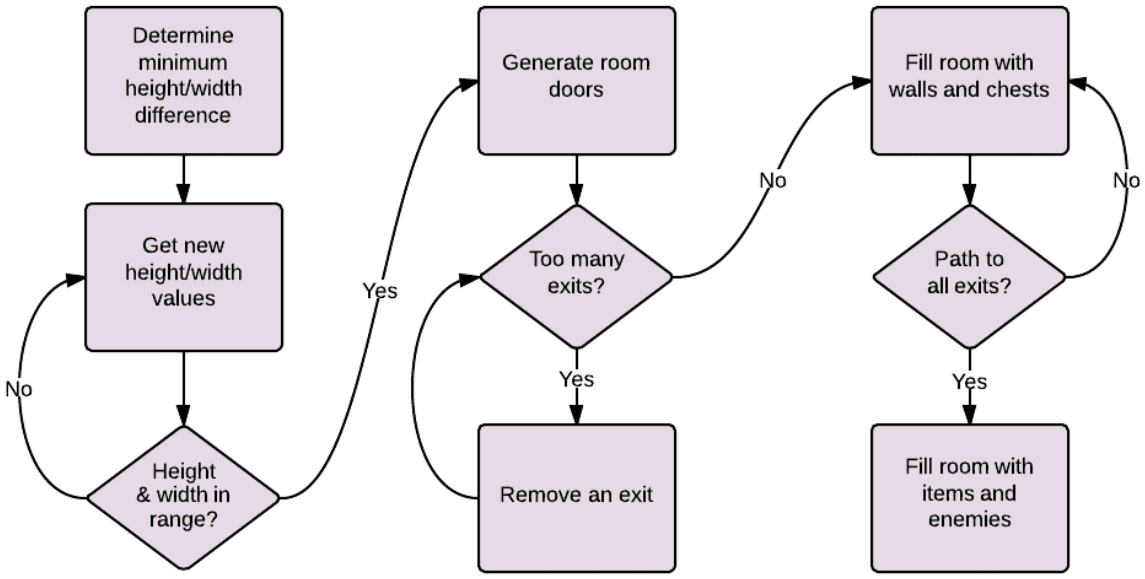

Figure 3: Room generation process

The room's dimensions establish the flow of movement through the room. The player has the freedom to walk around a large square room, whereas the player is forced to go from end to end of a long narrow room. In general, bigger rooms with similar height and width are easier than smaller rooms with a large height and width difference. This adjustment is done by using the number of fast enemies killed in the previous level to determine a minimum difference between height and width. Less fast enemies killed results in a higher difference and a narrower room, making enemies harder to avoid. Height and width values are repeatedly generated until their difference is at least the minimum, enforcing a specific room shape.

The number of doors in a room influences the amount of routes through the dungeon and the ability to escape a challenging fight, so fewer doors results in a higher difficulty. A room can have up to four doors (north, south, east, and west), which are probabilistically determined. If the current number of doors exceeds the calculated number of doors the room should have, doors are removed one by one until there are the right amount. Extra tests ensure that there are not too many dead ends in the dungeon.

The wall density of a room determines how much space the player has to navigate, which is influenced by the ratio of fast and strong enemies killed. A high density creates more chokepoints and gives the player less room to move around. The process of creating the walls starts with a temporary 2D array, which is populated by walls and chests using the calculated density. To make sure that all exits are accessible, the A* search algorithm is used to find paths from one exit to each of the others. The room can use this configuration of walls and chests if there is a path to all

exits. If a path is blocked, the array is reset and populated again, with the density decreasing every five attempts to prevent infinite loops.

After the geometry is established, the room is filled with items and enemies. Item type and placement vary consistently, but the frequency, type, and strength of the enemies are affected by the ratio of enemies killed in previous levels. If the player did not kill many fast enemies, the current level will generate more fast enemies to make them harder to avoid. If the player did not kill many strong enemies, the strong enemies will have higher attack, defense, and health. The maximum number of enemies per room is affected by both types of enemies killed.

Tools

The game engine was created using the enchant.js framework, which is a HTML5 and JavaScript based library. It is responsible for creating the game window, loading assets, processing keyboard input, and more. It also features many helpful classes for creating 2D game elements such as sprites, labels, maps, and scenes.

The music is created by AUD.js, a JavaScript based procedural music generator (Adam, 2014). The singleton object takes a stress and an energy value as inputs to generate a chiptune loop with a specific mood. AUD.js fits very well with this project because it can seamlessly change moods in the music without pausing, coinciding with the level's gradual difficulty change.

Pathfinding was implemented using Easystar.js, a JavaScript library for an asynchronous A* search algorithm. The A* search is used for enemy movement and for ensuring paths between doors while populating rooms with walls and chests.

Conclusion and Future Work

This project was an excellent introduction to game development. While I had previously worked on smaller games, creating *Pearl of the World* caused me to problem solve frequently and pay great attention to detail, which are beneficial for all programming projects. I feel like I have created a game that introduced a new feature to the roguelike genre and I hope that it can influence other games as well.

However, *Pearl of the World* still needs more work to be considered a successful game. In the future, more metrics could be utilized in the probability calculations in order to provide an even more personalized experience. Additionally, a story needs to be implemented in order to convey the context of the game and to better create the desired user experience. Finally, the game should be played in a user study to determine if the game successfully fulfilled its purpose of dynamically adjusting the dungeon to the player's actions.

References

Adam, T. 2014. AUD.js. http://timotheyadam.com/AUD/

Gaslamp Games 2011. Dungeons of Dredmor.

Jennings-Teats, M., Smith, G., and Wardrip-Fruin, N. 2010. Polymorph: A Model For Dynamic Level Generation. Proceedings of the 2010 Workshop on Procedural Content Generation in Games (2010). http://sokath.com/main/files/jenningsteats-aiide10.pdf

Katavić, T. 2013. Using Metrics to Create a Dynamic Level. University of Abertay Dundee. http://www.tinkatavic.com/uploads/3/3/2/0/3320510/dissertation.doc

Mojang 2011. Minecraft.

Valtchanov, V. and Brown, J. A. 2012. Evolving Dungeon Crawler Levels With Relative Placement. Proceedings of the Fifth International C* Conference on Computer Science and Software Engineering (2012). http://www.uoguelph.ca/~jbrown16/Vvaltchanov_c3s2e12.pdf

Valve 2008. Left 4 Dead.

Appendix: Room Population Code

```
/* Fills the room with walls, chests, and items and sets collision accordingly */
populateRoom: function() {
   var countRow, countCol;
   var isPath, startX, startY, endX, endY;
   var exitCoords = Array();
   var pathFinder = new EasyStar.js();
   var retry = {Value: false, Attempts: 0};
   var tempTiles = Array(ROOM_HIG_MAX);
   var placeholder = 3; // Arbitrary placeholder tile not used in rooms for chest placement

   var obstacleChance = metrics.getObstacleChance();
   var chestChance = metrics.getRoomChestChance();

   pathFinder.setAcceptableTiles([0, NEXT_LEVEL, NORTH, SOUTH, EAST, WEST, UP, DOWN]);
   do {
      retry.Value = false;

      /* Populate the map with walls and chests */
      for (countRow = 0; countRow < ROOM_HIG_MAX; countRow++) {
         tempTiles[countRow] = Array(ROOM_WID_MAX);
         for (countCol = 0; countCol < ROOM_WID_MAX; countCol++) {
            tempTiles[countRow][countCol] = this.tiles[countRow][countCol];
            if (tempTiles[countRow][countCol] == 0 && Math.random() < obstacleChance)
               tempTiles[countRow][countCol] = 2;
            else if (tempTiles[countRow][countCol] == 0 && Math.random() < chestChance)
               tempTiles[countRow][countCol] = placeholder;
            if (countRow > 0 && tempTiles[countRow-1][countCol] == 2 &&
                (tempTiles[countRow][countCol] == 1 || tempTiles[countRow][countCol] == 2))
               tempTiles[countRow-1][countCol] = 1;
         }
```

```
        }

        pathFinder.setGrid(tempTiles);

        /* Establishing the tiles that can't be blocked*/
        if (this.North != false) {
           exitCoords.push((ROOM_WID_MAX-1)/2);
           exitCoords.push(this.wallN);
        }
        if (this.South != false) {
           exitCoords.push((ROOM_WID_MAX-1)/2);
           exitCoords.push(this.wallS);
        }
        if (this.East != false) {
           exitCoords.push(this.wallE);
           exitCoords.push((ROOM_HIG_MAX-1)/2);
        }
        if (this.West != false) {
           exitCoords.push(this.wallW);
           exitCoords.push((ROOM_HIG_MAX-1)/2);
        }
        if (this.Up != false) {
           exitCoords.push((ROOM_WID_MAX-1)/2);
           exitCoords.push((ROOM_HIG_MAX-1)/2);
           exitCoords.push(this.wallE - 1);
           exitCoords.push((ROOM_HIG_MAX-1)/2);
        }
        if (this.Down != false) {
           exitCoords.push((ROOM_WID_MAX-1)/2);
           exitCoords.push((ROOM_HIG_MAX-1)/2);
           exitCoords.push(this.wallW + 1);
           exitCoords.push((ROOM_HIG_MAX-1)/2);
        }
        if (exitCoords.length <= 2 || sceneList.length == minRooms || sceneList.length == 0) {
           exitCoords.push((ROOM_WID_MAX-1)/2);
           exitCoords.push((ROOM_HIG_MAX-1)/2);
        }

        /* Make sure there is a path to each exit */
        startX = exitCoords.shift();
        startY = exitCoords.shift();

        while (exitCoords.length > 0) {
           endX = exitCoords.shift();
           endY = exitCoords.shift();
           /* The callback makes the loop run again if the open paths weren't found */
```

```
            pathFinder.findPath(startX, startY, endX, endY, function(path) {
               if (path == null)
                  retry.Value = true;
            });
            pathFinder.calculate();
         }
         exitCoords.length = 0;

         if (++retry.Attempts % 5 == 0)
            obstacleChance *= 0.9;
      } while (retry.Value && retry.Attempts < 50);

      /* Only set the walls and chests if it didn't time out */
      if (retry.Attempts < 50)
         this.tiles = tempTiles;

      /* Put chests, items, and collision in the room */
      for (countRow = this.wallN; countRow <= this.wallS; countRow++) {
         for (countCol = this.wallW; countCol <= this.wallE; countCol++) {
            this.tiles[countRow][countCol] = tempTiles[countRow][countCol];
            this.collision[countRow][countCol] = 0;
            if (this.tiles[countRow][countCol] == 0 &&
                Math.random() < metrics.getRoomItemChance(player.seenOrb, player.numKeys)) {
               if (exitPlaced && !player.seenOrb && metrics.needEmergencyOrb())
                  this.items.tiles[countRow][countCol] = game.getRandomItem(true);
               else
                  this.items.tiles[countRow][countCol] = game.getRandomItem(false);
            }
            else if (this.tiles[countRow][countCol] == placeholder) {
               this.tiles[countRow][countCol] = 0;
               this.chests.tiles[countRow][countCol] = CHEST_CLOSED;
               this.collision[countRow][countCol] = 1;
               metrics.totalChests++;
            }
            else if (this.tiles[countRow][countCol] == 1 || this.tiles[countRow][countCol] == 2)
               this.collision[countRow][countCol] = 1;
         }
      }

      this.loadData(this.tiles);
      this.items.loadData(this.items.tiles);
      this.chests.loadData(this.chests.tiles);
      this.collisionData = this.collision;
   }
```