Procedural Level Generation for a 2D Platformer

Brian Egana

California Polytechnic State University, San Luis Obispo
Computer Science Department

June 2018

Introduction

Procedural Content Generation (PCG), as defined in the book “Procedural Content Generation in Games”, is “the algorithmic creation of game content with limited or indirect user input”[1]. Whether it creates the levels by itself, or together with game designers, PCG is a method used to reduce the cost of designing games, and to produce an endless amount of game content. One of the main challenges that come with PCG is the desire to use PCG algorithms to create entire levels that are feasible, unique, and fun. This is especially challenging for games of progression within the 2D platformer genre; insufficient consideration for developing the level-generating algorithms for this genre can easily lead to boring, repetitive levels. Therefore, the goal of this project was to implement a game mode that procedurally generates levels to determine the effectiveness of PCG algorithms for producing 2D platformer levels that are playable and interesting. The game used to implement PCG is the 2D platformer, Darkour.

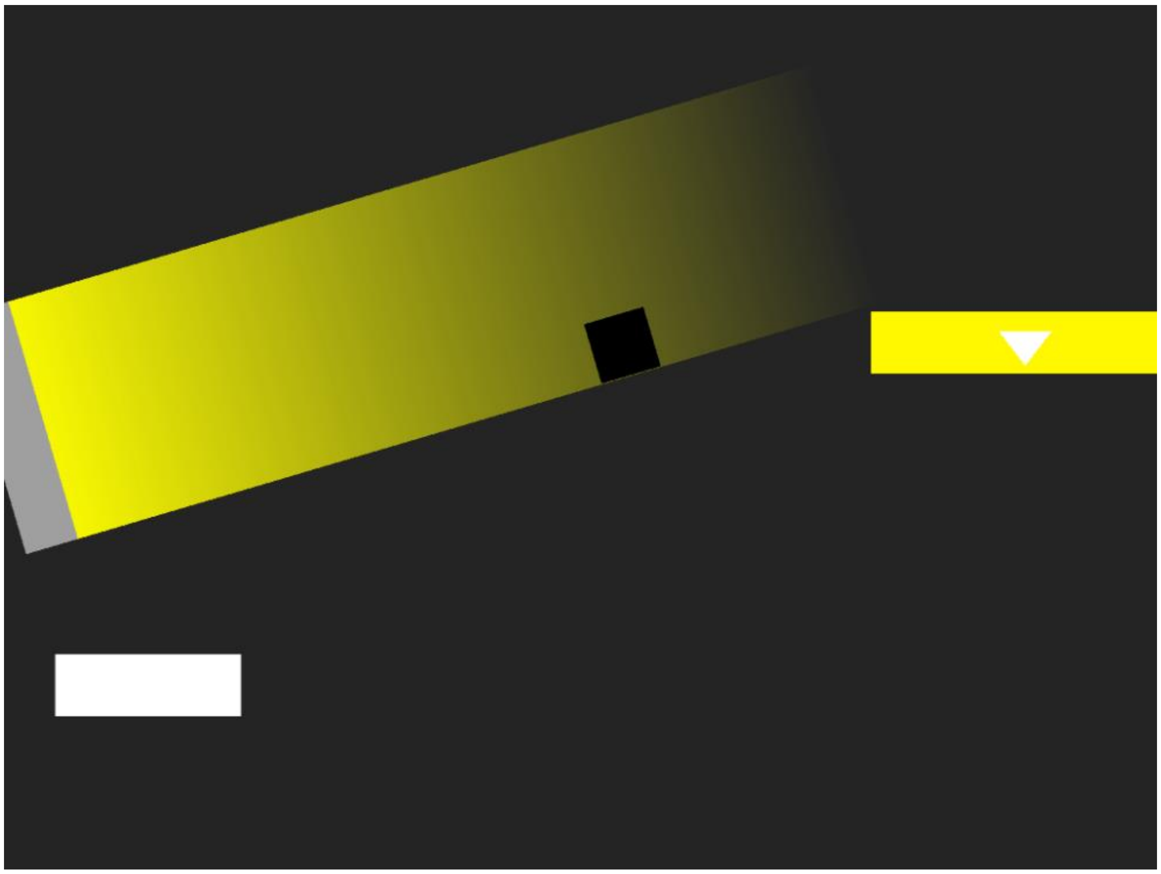
Figure 1: Darkour

Application

Darkour is a 2D platformer where players control a square that can only change state when exposed to light. Normally, while the player is a white square, the player can move, jump, wall jump, and pass through light emitted by a lamp. When the player transforms into a black square, the player can collide with the sides of any emitted light and, depending on the angle of the lights, can interact with their sides as though they were regular platforms or regular walls. The lights themselves become weaker the farther they are from their respective lamps, so it is possible that the player can fall out of the light and revert into their normal form. The lamps may have various states and patterns, including light flicker, rotation, dynamic scaling, and movement. Specifically for light flicker, when the light turns off while the player is a black square, the player will be forced back to normal.

Default Controls

- A / Left Arrow: Move Left
- D / Right Arrow: Move Right
- W / Up Arrow / Space: Jump
- S / Down Arrow: Move to Next Level (When Standing on End Platform)
- Shift: Change State (Toggle Light Collision)
- Escape / P: Pause Game

Player Goals

In each level, the player must use platforms and lights to get to a yellow end platform while avoiding gaps and pits that would lead to their death. Once the player is physically touching the end platform, the platform will turn green, indicating that the player can press the assigned button to continue to the next level. For the normal game mode, the player's goal is to complete all nineteen, handmade levels. There is also a timed mode where the player attempts to clear the same nineteen levels with the fastest time possible. For the endless game mode, the goal is to complete as many levels as possible.

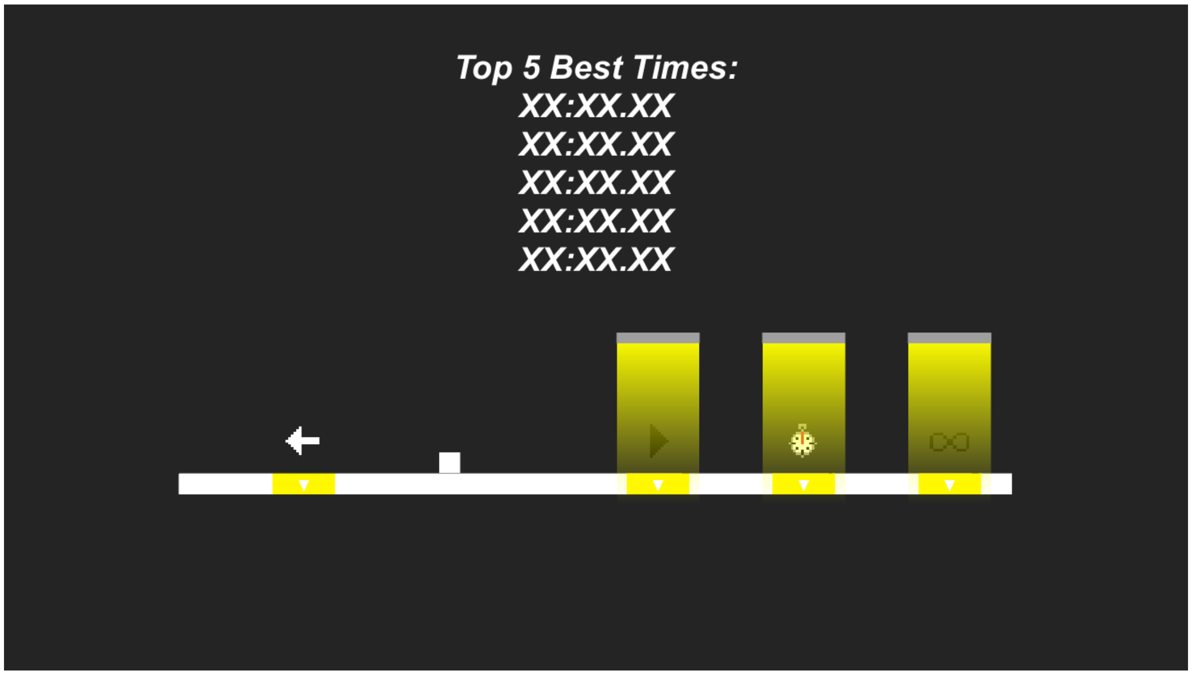

Figure 2: Game Mode Select Screen

Background

To better understand the implementation of PCG for platformers, we must first describe one of the core mechanics of a platformer, and how the physics of this mechanic applies to generating a level. Clearly, the core mechanic for platformers is movement from one point of a level to a defined end point. Movement may include jumping over pits that would restart a level or restart the entire game, should the player fall into one. This implies that a player's given jump distance must be accounted for in order to have feasible platform placement. To place these platforms in appropriate locations, we define the player as a projectile and use the projectile displacement equations to measure the distance between the player's initial and final locations, given the player's initial speed and amount of air time. These equations, shown in Figure 3, are crucial pieces for developing a PCG algorithm to generate feasible platformer levels. Depending on the value of t, the next platform location, shown by the red dot in Figure 4, could lie anywhere on the parabola produced by the equations.

$$x = x_0 + v_{x0}t \quad y = y_0 + v_{y0}t - \frac{1}{2}gt^2$$

Figure 3: Projectile Displacement Equations

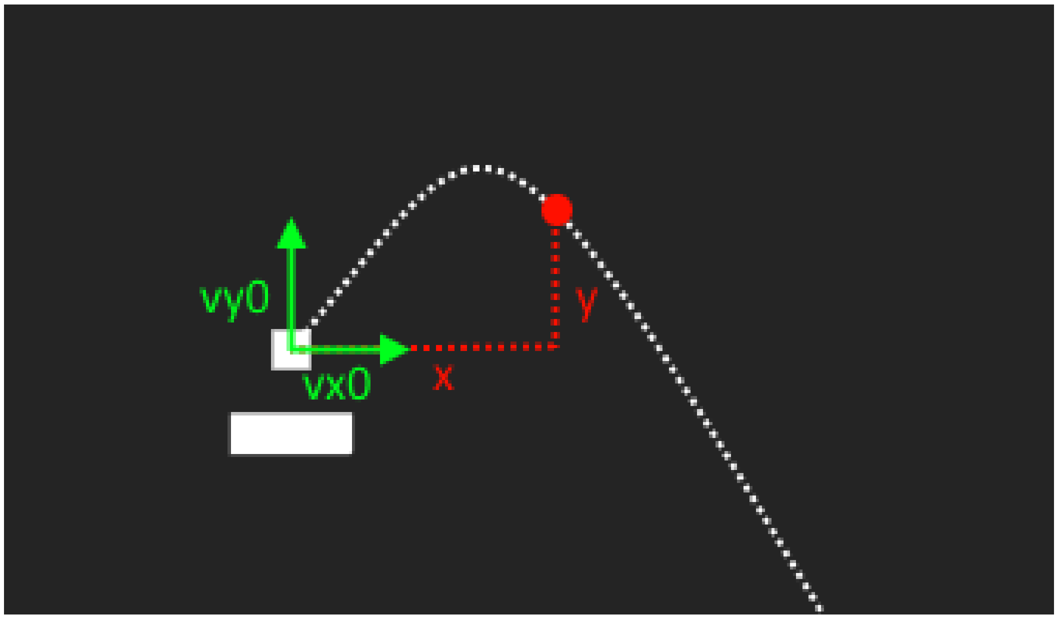

Figure 4: Possible Platform Placement along a Parabola

For implementation, the Unity Game Engine was used to make the game. This is done for three reasons. First, Unity, as well as most game engines, offloads low-level tasks like graphics rendering, memory management, and cross-platforming to different operating systems. Offloading these tasks allows creators to quickly develop prototypes of their games. Second, Unity is a relatively easy tool for game programmers and artists to learn how to use because of its thorough manual, its extensive API documentation, and its tutorials. Third, Unity represents assets within a game as objects, so scripts that are made to interact with these game assets are primarily made in an object-oriented style of programming. This style matches the introductory Computer Science coursework taught at Cal Poly SLO. For this game engine, scripts are made in C#, a programming language similar to Java.

Design

Darkour has three game modes. The first mode has 19 hand-crafted levels. The second game mode is the same as the first, with the exception of an added timer used to measure how fast a player can clear all of the levels, and the five fastest times will be displayed in the main menu. The third game mode will use procedurally generated levels in order to have an endless game mode. Specifically, the level will spawn platforms and lights in positions that allow the player to traverse from the beginning platform to the end platform.

For all game modes, the difficulty increases as the player progresses. Each successive level in both modes is mostly unique, with different platform and lamp placements, as well as different lamp behaviors that are initially introduced one by one, and then put together in different combinations to add challenge and complexity. In later levels, there can be lamps that each have one different behavior, or lamps that each have several behaviors at once.

To design Darkour so that it can use PCG, each game asset, including the square that the player controls, the regular platforms, the lamps, and the end platform, is preconfigured in a way that simplifies the process of designing a level. This is where using a game engine becomes useful, as there are heavy use of prefabs, or game objects with preset properties. Once the game assets were made into prefabs, the PCG algorithm made use of these prefabs to create feasible levels.

At the code organization level, each script for movement, rotation, and other patterns is made into modular components so that if there is a desired behavior for a platform or a lamp, the script for that behavior can be easily added to the game object at runtime. Most of the other scripts were also made so that each did one task. There were scripts that did the following:

- The Player script implemented the player's movement and state changes.
- The Scoreboard script logged the player's top five best completion times.
- The Timer script kept track of how much time the player has taken to play the game so far.
- The Rhythm Generator script generated a rhythm, or a set of random intervals at which the player would be in the air after a jump.
- The Lamp Property Generator script generated each individual lamp's properties.
- The Level Generator script took a list of rhythms and translated them into actual platform and lamp positions based on the game's physics and the aforementioned projectile equation.

The Game Control script was responsible for operations that needed to persist across different scenes, one of which is passing information from the Rhythm Generator and the Lamp Property Generator to the Level Generator. It was also responsible for containing the thresholds for increasing difficulty. The specifics of these scripts are described in the next section.

Implementation

Wall Jumping

One of the core mechanics for Darkour is wall jumping. After several iterations, wall jumping was implemented so that when the player is sticking to a wall (i.e. holding down the key respective to the direction of the wall) and the player presses the jump button at the same time, the player jumps up and opposite the direction of the wall. To apply the wall jump force, the normal vector of the wall is added along with the usual upward vector, and both are scaled by a set jump speed. For instance, if a wall is to the player's right, the normal vector of that wall would be to the left, so the player would wall jump up and to the left. After the player wall jumps, the wall jump speed is reduced by a set value every frame until it reaches zero.

Rhythm Generator

As defined in the previous section, a rhythm is a set of intervals at which the player would be in the air after a jump. This concept has been adapted from the research paper, "Rhythm-Based Level Generation for 2D Platformers" [2]. To generate a rhythm, the Rhythm Generator script sets a value representing each interval length to be between initial jumping time and the time needed to fall down at a certain velocity. For this game, it is assumed that the first rhythm corresponds to a jump from the starting platform to a lamp, the last rhythm corresponds to a jump from the last lamp to the end platform, and every rhythm in between is for jumping between lamps. The Rhythm Generator also initializes the properties that each lamp has in a rhythm. These properties can be rotation, translation, or flickering. This information will go to the Lamp Property Generator, as seen in the Procedural Generation Pipeline in Figure 5.

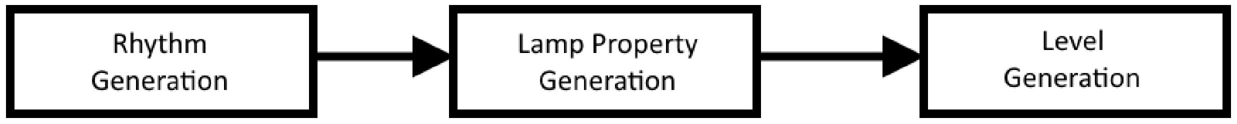

Figure 5: Procedural Generation Pipeline

Lamp Property Generator

As the name implies, the Lamp Property Generator script generates the actual properties that each lamp will have. The main function is shown as pseudocode in Figure 6. This is the most important function in the procedural generation pipeline, since the oscillation rate of each lamp property needs to be synchronized and the movement directions must be set accordingly in order for a generated level to be feasible. For example, assume that a lamp moves left and right within a given distance. If the next lamp that is to the right rotates in such a way that it is only reachable every five seconds, the lamp moving left and right must be at the rightmost position every five seconds so that the player can reach the rotating lamp. To implement this, a rotation speed is generated for a supposed lamp rotation property. The time it takes for an object to make a revolution is 360 divided by the rotation speed. The oscillation rates for the other lamp properties are based on this calculation. A lamp with a translation property moves left and right twice for a given rotation, and lamp with a flicker property is off at half of a rotation, and on at the full rotation time. After each lamp property is generated with the appropriate oscillation rates, these properties and their associated rhythms are passed to the Level Generator.

```
GenerateLampProperties(rhythms) {
    rotationSpeed = number between 20 and 30
    translationSpeed = number between 2 and 4
    quarterCircleRotationTime = (360 / rotationSpeed) / 4
    flickerTimeStart = quarterCircleRotationTime
    flickerRate = quarterCircleRotationTime * 2

    for i = 0 to rhythms.length {
        apply speeds and rates to respective lamp properties
    }
}
```

Figure 6: Pseudocode for Lamp Property Generation

Level Generator

The level generator is where the rhythms are used to initialize prefabs of the player's starting location, the end platform, and all of the lamps in between. The positions of the starting location and the end platform are also used to set the camera size and restart collider positions. To determine the positions of each lamp, the generator first calculates the player's position after following a generated rhythm. In other words, the generator uses the projectile equation with the player's current position as the initial position, the player's maximum speed as the initial velocity, and the rhythm interval as the time input. After using the equation to get the player's final position, the level generator instantiates a lamp above this projected position. Then, the level generator applies the lamp property associated with the rhythm, adjusting the angle and movement speed of the lamp accordingly to ensure feasibility. Finally, to generate the next lamp, the player's current position is set to the rightmost lamp position according to the lamp's given properties. These steps are repeated until the final rhythm leads to the end platform. An example is seen in Figure 7, with the first lamp having rotation, the second having translation, and the third having flicker.

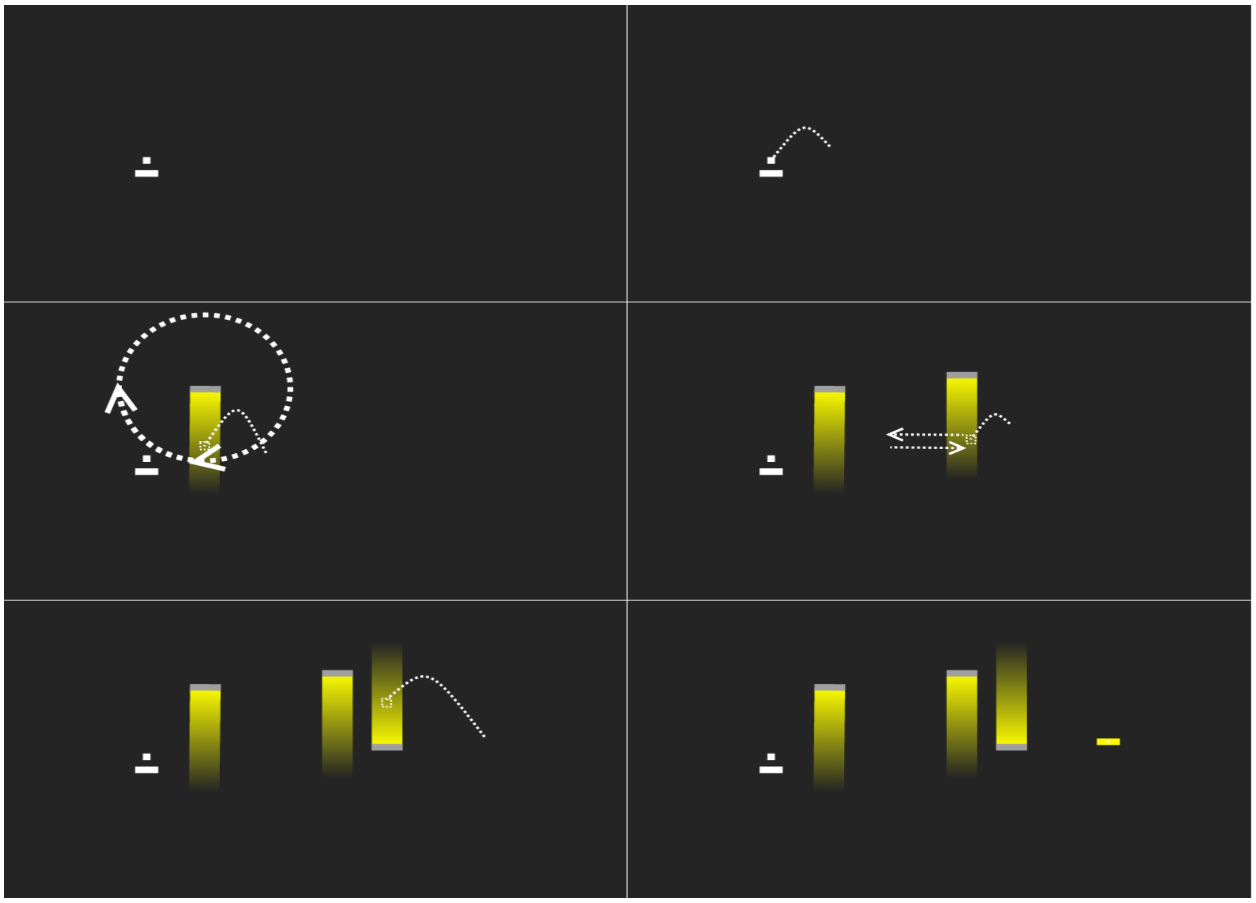
Figure 7: Example Process of Level Generation

Analysis

In order to measure the effectiveness of the PCG algorithm, a survey was sent out and gathered data from 130 playtesters. The survey contained questions about the core gameplay, as well as the endless game mode.

The questions and results about core gameplay were as follows:

- In the main menu and pause menu, there is a button that shows the controls for the game. Did you use this to learn the controls?
- The controls were responsive enough to let me perform the actions that I expect to do and finish each level.
- If you found the controls unsatisfactory, what specific flaws did you notice about the controls, and do you have any suggestions for improving them?

- For the flickering lights, do you want an indicator that shows when the lights turn on and off?

In the main menu and the pause menu, there is a button that shows the controls for the game. Did you use this to learn the controls?

130 responses

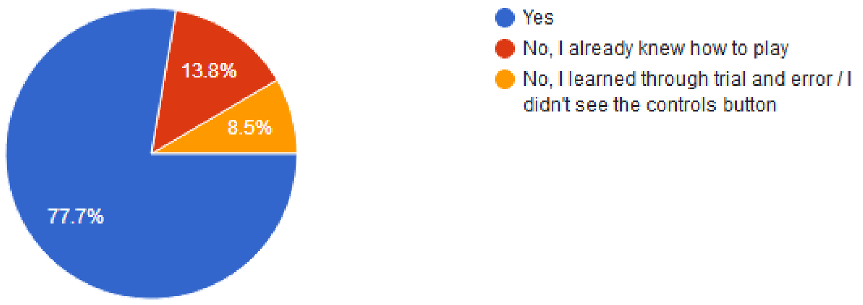

Figure 8: Core Gameplay Question 1 Responses

For the first question regarding whether or not playtesters learned how to play using the controls menu, 77.7% said that they used the controls menu, 13.8% already knew how to play, and 8.5% learned without using the menu. Knowing how players learn the controls is important because adapting to how they learn will help first-time players focus on the actual game, rather than spending too much time learning the controls. For this game, the fastest way of learning the controls was through the controls menu, which was used by a majority of the respondents.

The controls were responsive enough to let me perform the actions that I expect to do and finish each level.

130 responses

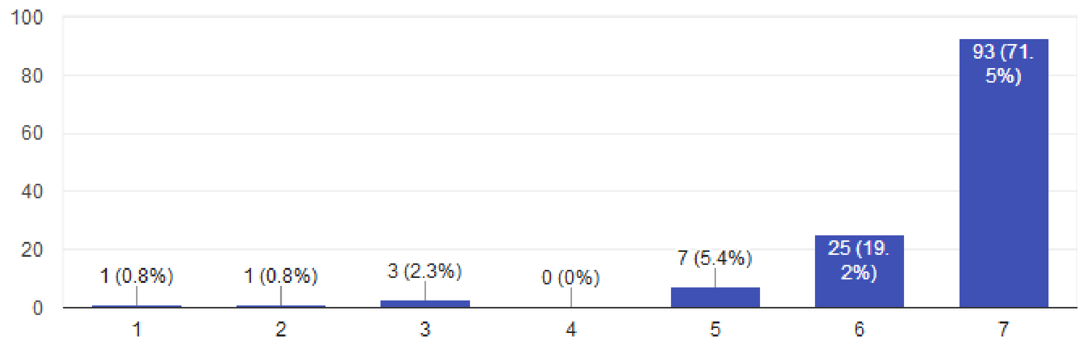

Figure 9: Core Gameplay Question 2 Responses

For the second question rating control responsiveness from 1 (not responsive at all) to 7 (very responsive), 96% of the respondents rated the controls to be 5 and above, while the rest rated it 3 or lower. The responsiveness of controls are the most important aspect of a platformer, so seeing that almost all of the respondents reacted positively is a sure sign of success. However, the existence of complaints, however small in number they are, clearly show that there is still room for improvement. For those who experienced that the controls were unresponsive, the third question prompted for specific details about the unresponsiveness and asked for potential improvements. Out of the 3 responses for this question, 2 said that there was an input delay, while the third said that "jumping while simultaneously launching was difficult".

For the last question concerning the flickering lights, 88.5% said that they wanted an indicator showing when the light would turn on and off. This purpose of this question was to show whether or not having no indicator was a reasonable way of challenging the player. According to the responses, the lack of an indicator implied that players were frustrated, rather than challenged, by the lack of an indicator.

The questions about the endless game mode were as follows:

- In the endless game mode, did you play a level that wasn't feasible?
- Do you want the option to generate a new level on demand?
- The endless game mode's difficulty scaled at a pace that frequently challenged me.
- The levels in the endless game mode were repetitive.
- How long did you play the endless mode?
- Overall, I enjoyed playing the endless game mode.
- Is there anything in this game mode that you would like to see added?

In response to the first question, 34.6% of playtesters said that they played an unfeasible level. Of course, it is a critical failure if the PCG algorithm produces a level that cannot be finished. However, determining whether or not a level is feasible by directly observing the players' skill and the generated level itself was beyond the scope of this project. Therefore, the less costly but less certain way to determine success was to ask the players and see if the majority say that they did not play an unfeasible level.

For the second question, 85.3% said that they want the option to generate a new level on demand.

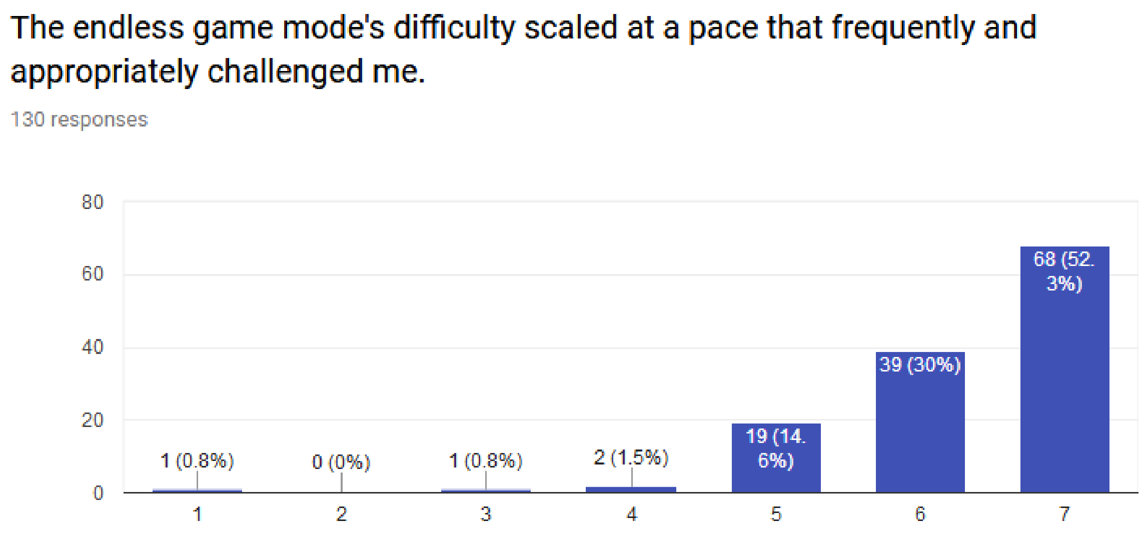

Figure 10: Endless Game Mode Question 3 Responses

As seen in Figure 10, 97% of playtesters said that the difficulty scaled appropriately. Since difficulty scaling is an important part in making a game fun, the responses to this question suggest that the PCG algorithm succeeded in this aspect.

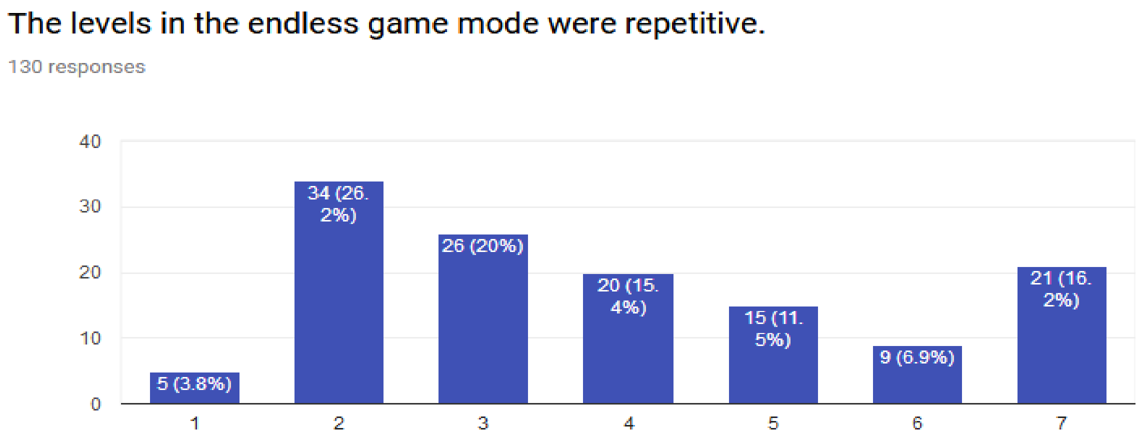

Figure 11: Endless Game Mode Question 4 Responses

It is important to note that even though levels may have appropriate difficulty scaling, this does not necessarily mean that subsequent are entirely unique. This is why it was important to ask playtesters if subsequent levels were repetitive. Compared to the previous question, the responses to the fourth question had much more variation, as seen in Figure 11. With 1 being strongly disagree, 4 being neutral, and 7 being strongly agree, 50% of playtesters responded with a 3 or lower, 34.6% responded with a 5 or above, and 15.4% responded with a 4. While a majority of playtesters say that the levels were not repetitive, there is clearly room for improvement.

For the fifth question, 40 respondents have said that they played the endless mode for at least half an hour, with 23 of those respondents playing for at least an hour. The amount of time playing may explain how some players would perceive the game as repetitive; as the player becomes more skilled, the player would eventually see every possible combination of lamp properties, thereby seeing levels at much later times as more of the same. Nevertheless, the other implication of the responses for this question is that this game mode is able to garner a very positive level of player engagement.

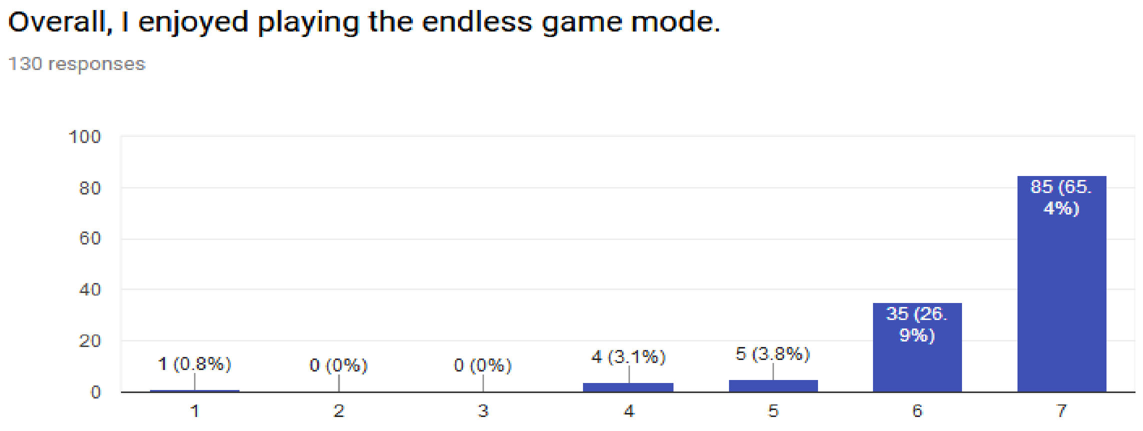

Figure 12: Endless Game Mode Question 6 Responses

For overall satisfaction, the responses are similar to the one for the difficulty scaling question, except a higher percentage strongly agreed with the presented statement. This is shown in Figure 12.

Related Work

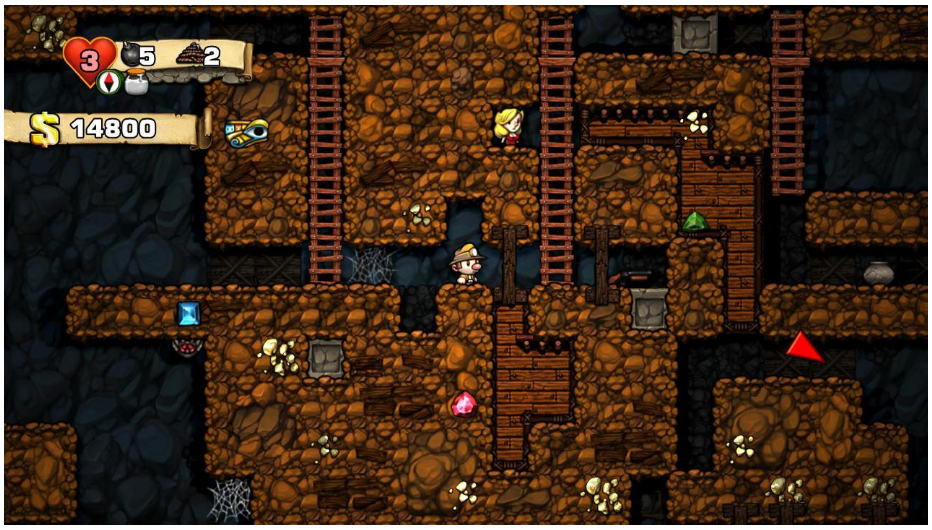

Figure 13: Spelunky

Spelunky

Spelunky is one of the earlier examples of a platformer that uses procedural level generation to make each run through the game unique. The difference between Darkour and Spelunky is that while the player is avoiding death traps, outsmarting enemies, and collecting treasure in Spelunky, in Darkour players only need to reach the end platform. Another difference is that in Spelunky, losing all health means that the player has to start from the beginning of the game. In Darkour, the player can attempt a level any number of times, but any mistake will reset the level.

Figure 14: Cloudberry Kingdom

Cloudberry Kingdom

Darkour will have the most similarities with the 2013 platformer, Cloudberry Kingdom. Both games will have players traversing from a starting location to the end, avoiding hazards along the way. While playing through a level for both games are very similar, one difference from Cloudberry Kingdom will be that Darkour will have core game mechanics that are mostly unique for the genre. What is also different is that it will not support multiplayer, as its implementation would be too large for the scope of this project.

Future Work

Included in the survey was a question prompting playtesters to suggest additions to the game. These suggestions were:

- Different difficulty scaling rates
- Permanent death to highlight the goal of lasting as long as possible

The first suggestion may have come from how the endless game mode would seem repetitive after playing for a considerable amount of time. Therefore, adding more variables to the PCG algorithm, whether it be different difficulty scales, more lamp properties, or an additional mechanic, would lessen its repetitiveness. As for the second suggestion, this is because the endless game mode currently does not provide any incentive to play as much as possible; all the mode does is generate levels. So, this could be improved by simply having a goal to work toward.

Other additions that were not specific to the mode included:

- A level counter
- A more challenging route that can be traversed without turning dark
- More colors and shapes

Conclusion

The endless game mode of Darkour garnered mostly positive reception, providing interesting and varied play. While there are several improvements that have been revealed through feedback, the goal to implement a game mode that procedurally generates levels and determine the effectiveness of PCG algorithms for producing feasible and interesting 2D platformer levels has been met.

References

[1] http://pcgbook.com/

[2] Smith, G., Treanor, M., Whitehead, J., Mateas, M.: Rhythm-based level generation for 2D platformers. In: Proceedings of the 4th International Conference on Foundations of Digital Games, FDG 2009, pp. 175–182. ACM (2009)