Senior Thesis

Procedural Level Generation for *Monument Valley* Styled Puzzle Games

Faith Meacham
Advisor: Aline Normoyle

Submitted in Partial Fulfillment of the Requirements of the BA in Computer Science

Bryn Mawr College

Spring 2021
Abstract

This project explored how to best generate puzzle games that require path planning to solve. We looked to a specific path-puzzle game, called Monument Valley, for a basis to begin experimentation. The goal of this project was to develop an algorithm which can automatically generate levels that contain features characteristic of Monument Valley style puzzle games. Foundational research into different types of procedural content generation helped us select a guided random approach for our level generation, similar to that of a test-and-reject based model. Specifically, the work for this project required Unity to construct Monument Valley styled game objects and C# to program the gameplay features and conduct the level generation. The work reported for this project concluded with the development of a partially procedural level generator which lays the groundwork for the development of a fully automated system to be created in the future. We analyzed two categories of levels: short generated levels consisting of one or two switches; and replications of levels from Monument Valley, which were more complex and contained subgoals in addition to the final goal. The results from this analysis highlighted the distinct potential that incorporating elements of surprise within the generated levels has in remedying their lack of comparable difficulty. Further work focused on transitioning the final manual steps in our system to procedural steps will result in the fully automated system initially envisioned.
Acknowledgments

Thank you to my thesis advisor, Professor Aline Normoyle for all the help and guidance given over the course of this semester. This project would not have been successful without your insight, direction, and work. Additional acknowledgement and gratitude to Professor Dianna Xu and my fellow senior conference classmates for providing helpful peer reviews and suggestions along the way. And a special thanks to my friends and family for offering support and advice when needed.
Contents

1 Introduction .............................................. 7

2 Related Work ........................................... 8
   2.1 Grammar Based PCG .................................... 8
   2.2 Answer Set Programming Based PCG ...................... 10
   2.3 Wave Function Collapse Based PCG ...................... 10
   2.4 Evolutionary Search Based PCG ......................... 12
   2.5 Test-and-Reject Based PCG ....................... 13
   2.6 Influences on Our Implementation ...................... 14

3 Methodology ........................................... 15

4 Results and Analysis ................................ 21

5 Future Work ........................................... 25

6 Summary ............................................... 26

References ............................................. 27

Appendix ............................................... 29
1 Introduction

Procedural content generation (PCG) is the use of algorithms to produce game content automatically [TSN16]. This content can range from the game rules, characters, goals, and scenery needed for the game design. A particularly fruitful application of PCG is the generation of game levels. Such generation enables games to have the potential for an infinite number of levels, preventing users from ever running out of levels to complete. Because of this appeal, a lot of research and work have gone into this topic, resulting in a variety of algorithms aimed to produce successful game levels efficiently [TSN16]. These algorithms include grammars [TSD16], answer set programming [NS16], wave-function collapse [KS17], evolutionary search [TS16], and test-and-reject [DB17]. Each of these derives its basis from different fields and backgrounds but has been proven to function well for procedural level generation.

In addition to various types of PCG algorithms, there are also a variety of game types that look to benefit from procedural level generation. From First Person Shooter games [TS16] to dungeon set games [SLT+16] to puzzle games [KSG16], each application contains its own special set of problems when attempting to generate playable levels. The work for this project analyzes how procedural level generation can best be implemented for Monument Valley styled puzzle games, see Figure 1 for a visual reference of what these levels look like in the commercial version of Monument Valley. Note that in Figure 1 the goal is to find a path from the orange character in the bottom, point A, to the black square-shaped target in the top, point B. Monument Valley was awarded the Apple Design Award in 2014 and has earned high praise and reviews for its compelling puzzles and stunning visuals [Tac14]. It is, however, rather short in number of playable levels [Rie14]. PCG’s potential to infinitely expand on this number of playable levels makes Monument Valley a great candidate for the work of this project. Focusing on the Monument Valley style of puzzle games involves a new set of problems for the level generation beyond those normally involved in level generation for other puzzle games. This is due to Monument Valley’s unique use of “switches” to edit the game space.

In this style of puzzle games, the path from point A to point B may not be readily available to the player and must be found by activating a series of “switches.” These “switches” modify the game environment and, consequently, the paths available to navigate. Monument Valley portrays the game environment from an isometric perspective. Its use of an isometric perspective lets the “switches” occur in unexpected ways.

Figure 1: Monument Valley Reference Image.
The player can unlock paths through Escher-like illusions to reach the level’s goal position. The inclusion of such “switches” within the game impacts how the level generation can occur. Special consideration must now be given to how to abstractly represent the environment, including the “switches”, and which style of PCG algorithm will produce playable, fun levels of this kind efficiently. The result of this project is a partially automated rule based generation algorithm. Specifically, this system generates a random path within a 3D voxel volume. This path is then manually adjusted by adding “switches” identified from a library of potential “switch” styles. This process produces playable puzzle game levels of the Monument Valley style. To the best of our knowledge, this is the first system created to procedurally generate game levels of this kind. This paper discusses existing implementations of various PCG algorithms for level generation with puzzle games, then goes on to outline the approaches and results of our implementation of PCG for producing Monument Valley style game levels.

2 Related Work

The field of Procedural Content Generation has garnered a large influx of research in recent years [TSN16]. Many algorithms that originated in fields like linguistics [TSD16] and artificial intelligence [ART16] have found applications within PCG, and specifically for procedural level generation. The primary aspects of PCG for level generation involve content representation and actual content creation [TSN16]. Each algorithm has a particular way of achieving these two aspects, as well as focusing on other specific aspects which they attempt to optimize or control. We see this in the following analyses of some of the major types of PCG algorithms for level generation.

2.1 Grammar Based PCG

Grammar based PCG is heavily influenced by linguistic and natural language based work. Initially, grammars were simply sets of rules for rewriting strings [TSD16]. At the base level, this is essentially what they do within PCG as well. In the PCG implementation of grammars, sets of rules are used to expand strings of symbols that represent commands like ‘F’ for forward, ‘+’ for right, ‘-’ for left. Additionally, grammars can be bracketed. This allows for a grammar to return to a previously saved position after performing a series of actions on a later position in the string [TSD16].

Grammars can also be used to produce graphs, tile maps, shapes, etc. as well as the string expansion explained above [TSD16]. Such graph generation makes them useful for generating game spaces, especially if the game space allows for multiple paths to the level’s goal position. Similar to the way they expand strings, grammars use sets of rules to construct their graphs. These rules can be thought of as steps within the design process for the level and its graph representation. Additionally, grammar based PCG for levels often breaks up the generation into two sub-tasks: game space generation and level “mission” generation. Game space generation involves defining the shape/layout of the level and what is included in the game environment. Level “mission” generation involves determining the goal for the player solving that level. Generating these separately allows for different styles of representation
and rule sets to be used. Linking them back together, in the end, produces a completely generated level \cite{TSD16}.

Figure 2: Abstract representation converted to actual game representation for a generated level of May’s Journey \cite{JICEN20}

Grammars have a wide range of applications within PCG. Jemmali et al. used it to procedurally generate levels for an education focused puzzle game, May’s Journey \cite{JICEN20}. The authors let the grammar based method handle the content generation aspects within the level generation while implementing a “work backward” approach to ensure the generated levels are solvable without needing post-processing or filtering. Specifically, the authors converted the input, a random solution, to an abstract syntax tree. From the tree, the game object that should be involved in the level is easily extracted, as well as the actions and attributes ascribed to it. This is passed through the grammar that used a series of rules to determine what shape the game environment should take to satisfy all the requirements. The rules within the set were weighted with probabilities to favor certain rules over others. Additionally, penalties were implemented when rules were selected so that no one rule was disproportionately chosen. Each object specific map shape was combined to a final game space by minimizing the number of modifications needed for such merging. Figure 2 shows two object specific map shapes combined to a full map in both the abstract representation, left, and the actual game representation, right. Finally, either a reward path algorithm or a maze path algorithm was used to determine the path through the level. Either algorithm maintained entrance-to-exit navigability \cite{ART16}.

The results of the grammar based PCG approach efficiently produced solvable levels with a high amount of variation. The paper identified, however, that the method required a fair amount of familiarity with the game’s reliance on coding conventions. A similar application of this method for our Monument Valley styled game levels presented the opportunity to make use of the highly successful model. We could avoid this identified dependence on coding conventions as the gameplay within Monument Valley styled games consists of fairly simple point-and-click environment navigation.

However, from a large-scale perspective, grammars can be difficult to implement as they require a deep understanding of the domain being modeled. Intricacies in the Monument Valley style levels may make this task especially complex. Due to such constraints, a less design-reliant approach may be more applicable for our generator.
2.2 Answer Set Programming Based PCG

Answer set programming (ASP) is a logic programming approach for PCG [NS16]. ASP is used to specify what the generated content should be like and then an ASP solver produces the content from this program. The first step in this process is defining the game logic i.e. the game structure and mechanics. Additionally, the constraints for how the content should adhere to certain properties must be defined. Once these are outlined for the ASP, the ASP solver can produce and find content that matches such input. ASP is a well-defined and frequently used method for PCG. This is largely due to the existing reliable tools like AnsProlog and Clingo that take the input required for ASP and efficiently produce the valid output [NS16].

One specific use of ASP was outlined by Lindeman to generate levels for a self-created puzzle game called Swappy [Lin18]. See Figure 3 for an example of the Swappy game interface. In this application of ASP, the method of “core-sim-style” [Lin18] was implemented in which three AnsProlog programs were developed. One, core, was used to generate the game pieces and explain the relationship between them. The second, sim, acted as a simulation of gameplay to ensure the rules of gameplay were followed. The final, style, was used to generate artifacts with a mind towards some aesthetic criteria. These three programs were linked by a Python command-line tool that took in parameters defining the level width and the number of character tokens to be used in the level. The AnsProlog files were run by the Clingo ASP solver. Of the many output levels, one was randomly selected once each ASP program had run. The Python tool then parsed and rendered the facts of the answer set chosen into ASCII art in the terminal. This output was finally provided to the Swappy game client [Lin18].

The results of this implementation of ASP were mixed. Lindeman noted that the generator occasionally still produced unsolvable levels. This was largely due to the bugs surrounding how the win condition of the game was represented in the ASP input. Additionally, the implementation was much slower than desired. Because ASP based systems have been found to handle large-scale generation problems well, this issue was likely also a result of the intricacies surrounding how the rules were denoted for the ASP [Lin18]. Such results highlight the importance of defining clear game logic in the start to avoid such undesirable features for our level generator.

2.3 Wave Function Collapse Based PCG

The wave function collapse algorithm was originally an example-driven image generation algorithm. It was later expanded into the world of PCG and used for level generation. It uses a non-backtracking, greedy approach that looks to match each “window” present in the input directly to a “window” present in the output [KS17]. The algorithm can be understood in a

Figure 3: Swappy Game Interface. [Lin18]
constraint solving light. In this way, it is essentially weighing certain heuristics and selecting the best or minimum values. It relies heavily on the heuristic of the minimum remaining value. The algorithm has been found to work best with abstract chunks of input rather than more literal input. Various versions of the algorithm have been formulated since its release including work to add in backtracking capabilities and building out a complete Unity toolset asset for it [KS17]. Figure 4 shows an example of how the wave function collapse algorithm uses pattern matching for content generation. Here we can see the parsing of tiles from the input image. It takes these tiles and the rules for what they can be adjacent to and produces a new image for which the rules are satisfied.

![Figure 4: Example of Wave Function Collapse pattern matching. [KS17]](image)

Kim et al. introduced a use of the wave function collapse algorithm that could work with graph based representations of content rather than simply grid based content [KLL+19]. This research successfully altered the algorithm to accurately produce content as specified. To test this graph based version of the wave function collapse algorithm they applied it to generate game content for various game styles including Sudoku and 3D prototype game levels. In the generation of 3D prototype game levels, they made use of an open source path-finding algorithm and the 3D software Blender.

Success with this style of level generation indicated that implementing the graph based version of the algorithm may offer a great reference for our level generator as it will need to produce 3D based Monument Valley style game levels. The results of the experiments with both Sudoku and the 3D prototype levels indicated that this new version of the wave function collapse algorithm allows for PCG to take advantage of wave function collapse’s excellent content control [KLL+19]. However, a potential drawback for this implementation was the growth in computation time once it was paired with graph based representation. To improve such computation time, work with optimization must be done since the graph based representation requires many more constraints to be specified at the start of the algorithm [KLL+19]. Additionally, the wave function collapse algorithm required a sufficiently large example from which to extract the tiles or “windows” on which to frame its matching based generation. Our inventory of Monument Valley styled game levels may not satisfy this requirement.
2.4 Evolutionary Search Based PCG

Evolutionary search based methods for PCG take inspiration loosely from the concept of biological evolution and the concept of survival of the fittest [TS16]. Generally, it starts with a random “population” of levels. Each level is judged by some evaluation function and ranked by their resulting values. The top-ranked levels remain in the “population” while the rest are substituted for mutated or reproduced versions of the highly ranked levels. This process continues until some level in the “population” reaches a predetermined goal score from the evaluation function [TS16].

The key features of this method are the search algorithm, the content representation, and the evaluation function [TS16]. Tweaking each of these features can improve or harm the efficiency and success of this method. Typically the game space is initially represented abstractly in a tree or graph and is evolved to a more concrete representation after each iteration [TS16]. In addition to impacting the performance of the algorithm (i.e. the search speed, game object creation ability, etc.), the style of representation can also impact the final appearance of the level [ART16].

Evolutionary search based content generation has been used to produce levels within a variety of puzzle games. Specifically, Kar-tal et al. applied it to procedurally generate Sokoban styled puzzle game levels [KSG16]. Sokoban is a puzzle game that requires the player to move boxes within the game environment to certain goal positions. See Figure 5 for an example of the game interface. The green arrows indicate the goal path for the generated level. The paper’s particular implementation of this form of PCG depended on the Monte Carlo Tree Search algorithm. The authors structured their game environment as a tree and defined a set of actions by which to evolve the tree to a playable game level. When a terminal action from the ruleset was performed the game level was evaluated by a data-driven evaluation function. The authors developed this evaluation function based on the data gathered from human input. This human input ranked the difficultly of existing Sokoban levels. Comparing their generated levels against this input the authors could determine if their levels had reached the desired level of difficultly [KSG16].

Results have shown that playable levels were successfully generated with increasing difficulty. However, the authors noted that generating larger puzzle levels caused the time required for the level generation to grow exponentially. From these results, we learn that using tree-based content representation and data-driven evaluation functions are successful aspects for search based PCG methods when puzzle levels have a relatively small game environment. Our Monument Valley styled levels range in size and complexity. This approach will likely demonstrate similar difficulties when attempting to produce such levels.

A similar search based method was performed by Baghdadi et al. with a Spelunky styled game [BEAO+15]. Spelunky, while not directly a puzzle game, requires puzzle-like path planning to “solve” its levels. Their implementation of search based PCG relied on a genetic algorithm and node based graphs to perform level generation. Figure 6 shows...
how this method evolved its generated levels from an abstract representation to actually indicate what the game level should include, as noted by the legend to the right. Unlike the Sokoban based level generation, this approach did not use a data-driven evaluation function. The authors used a function based on comparing a starting difficulty score to all the evolved difficulty scores. The results indicated that this approach could also successfully create playable levels, yet the paper identified that there was little variation between the procedurally generated levels. The paper cited the potential of distributing “enemies and items” within the generated game space post-generation to “spice up the level” \cite{BEAO+15}. This idea of adding post-generation “spice” offers potential in how we can use our own Monument Valley style “enemies” and game features (e.g. crows, building windows, and rooftop flags as seen in Figure 1) to ensure that our generated levels do not reflect the same lack of variation that this implementation identified.

2.5 Test-and-Reject Based PCG

Test-and-reject based PCG is an approach to content generation that does as it says. Essentially, it relies on an “expert” or designer provided template that the system, then, uses to generate levels \cite{DB17}. Each generated level is tested by a constraint checker to ensure it meets the requirements of a sound game level. If the levels fail any number of the tests in this check, the level is rejected and the generation step begins again. This process loops until a level is found to pass all the tests from the constraint checker. This approach is a slightly more rudimentary version of the evolutionary search based method. Both approaches require a step for testing or evaluating the generated content. The primary difference is that, with test-and-reject, the levels that do not satisfy the test are simply rejected while with evolutionary search based PCG the levels that do not rank highly by the evaluation function are evolved into levels reflecting the properties of the levels which the evaluation function ranks higher \cite{TS16}.

Dong and Barnes implemented a test-and-reject based method for procedurally generating educational puzzle game levels for the game BOTS, see Figure 7 \cite{DB17}. The red circled sections indicate the key aspects of the BOTS game interface including the game environment and the player movement options. In this study, the authors looked to test-and-reject PCG to allow for puzzle levels to be generated faster than relying on human “experts” to produce puzzles on an ad hoc basis. Specifically, this approach was careful to ensure the PCG system still allowed for the generated levels to maintain the game’s particular educational intent. With this in mind, the authors selected a test-and-reject based PCG method such that the experts who typically produced the game levels could instead simply input a
template with encoded goal requirements. The system first parsed the provided template, checking the template’s validity, and then created a template object. This object was passed to the program generator which converted the object into a valid solution program. This program was tested by the constraint checker to ensure that the requirements for a “good” game level had been met. If not, the program was rejected and the program generator had to run again to produce a new solution program. This process was repeated until a solution program was generated that passed the tests of the constraint checker. Finally, the puzzle file formatter converted the program into a puzzle file that was compatible with the BOTS game.

The results of this method successfully produced valid puzzle levels for the encoded goal requirements. Dong and Barnes additionally identified that the generated levels were produced much faster than the expert produced levels were for the same goals. Further, the generated levels had a higher level of variation. However, the generated levels had less pattern apparentness than the expert levels. In the realm of educational games, where specific learning goals are the primary focus, this could lead to the intended learning goal not coming across clearly. Similarly, this test-and-reject based PCG system did not allow for difficulty constraints to be considered. For Monument Valley styled puzzle levels such issues would be important to remedy. Despite not being an educational game, Monument Valley does have clear goals for the player to “learn” from each level and these would ideally be fairly apparent during gameplay. In addition, the difficulty of the generated levels for Monument Valley styled levels is important for realistic and interesting progression between each level.

2.6 Influences on Our Implementation

Each of these PCG methods offers pros and cons. The specific use of each method in the work and papers outlined offer additional insight into how one might best approach level generation for puzzle games similar to Monument Valley. Assessing these conclusions, as well as our inventory size and timeline, we proposed a method for generation most similar to that of test-and-reject. Like Dong and Barnes’ BOTS levels [DB17], Monument Valley styled levels have clear goals to achieve during gameplay and direct rules on how the game environment can be structured. We used these goals and rules as a template for which the system can base its level construction. Once levels were produced they were assessed on whether they satisfied the identified constraints and were adjusted, accepted, or rejected accordingly.
3 Methodology

The focus of this project was determining how to develop an abstract representation of the *Monument Valley* styled puzzle levels and, then, how to best generate these abstractly represented levels. The content representation must account for the architecture present in the game environment, the “switches” used to alter the game environment, as well as, the level’s goal/intended path points. The preliminary work in this endeavor was recreating several levels that currently exist in the actual *Monument Valley* commercial game. Such work required Unity to build out the game architecture and C# to program the “switches” present within the level. The process for recreating these aspects for each level followed a similar workflow outlined below.

First, a level was chosen from the game based on the architecture present and the “switching” style used. Such architecture features like levels with multiple tiers or exciting Escher-like illusion forming object placement were selected to ensure that each *Monument Valley* specific game environment element was represented in these levels. Additionally, because we required a variety of “switches” to generate levels with variation, we had to ensure that the hand-built levels exhibited this variation as well. Once an existing reference level had been selected, the architecture within the level was modeled in Unity. The root shape within the game architecture was a cube. Long thin cylinders were also used to create supports within double-decked levels. As a placeholder, the game character was represented by a red sphere. Building out levels required determining the placement of a series of cubes such that they represented the desired shape within the isometric view that *Monument Valley* makes use of. See Figure 8 for a side-by-side comparison of the scene view in Unity and the isometric game view. The scene view depicted the game environment as the level designer.
saw it while the isometric game view depicted the level as the player would. Ensuring that the two views created the desired shape/environment posed some challenges as the isometric view can make objects appear to be in a physical space they were not actually in from the designer’s perspective in the scene view. Mapping the placement from the existing levels into the recreated levels required an understanding of how the placement will lend itself to the level’s use of Escher-like “switching.” Additional consideration was given to the object materials in the built game environment. A shader was created to include a distinguishing outline along each edge and apply color evenly across each cube and cylinder. For particular objects, materials to denote the presence of ladders, doors, and goal positions were also included.

When the architecture of the built game environment matched that of the existing level, the work shifted focus to programming the “switches” necessary for the level. The “switches” were encoded in C# and allowed the game objects to move in such a way that new path potentials were revealed and/or advanced the game level. These “switch” movements made use of simultaneous rotations and translations. The player could click designated handles and buttons to activate or perform these “switch” movements.

Additionally, the player could move their game character throughout these hand-built game environments in a point-and-click manner. When a cube position was clicked by the player and this position was reachable via a path of connected game objects, the game character would follow this path to reach the clicked-object’s position. This traversal was based on the A* algorithm. Essentially, in this algorithm, the cost of choosing to advance onto one of the game character’s positional neighbors over its other neighbors is minimized until a path from the game character’s current position to the goal position was optimally found. The “switches” present in the built game levels affected the neighbors available to consider for this path planning. If a cube position was clicked that was not reachable the game character remained in their current position. The way a “switch” oriented or presented a particular game object made the same path between two positions navigable in some cases and not in others. A graph representing the navigation allowed for the level was automatically computed based on the level’s current geometry and the player character’s current orientation. This graph determined how the point-and-click style navigation could occur within the level as it was currently configured.

The game level could have multiple “switches” required to complete the intended path and reach the goal position. For the level in Figure 9, there were a total of six “switch” based movements involved. The series of “switch” based movements and their impact on the configuration of the game environment is shown in Figure 9. Using the “switches” in the order from Figure 9a to Figure 9g allowed the game character to reach the upper target position. The first “switch” based movement (Figure 9b) rotated the green highlighted section to form a ramp. The second “switch” based movement (Figure 9c) rotated the central object to meet the left middle platform. The third “switch” based movement (Figure 9d) rotated the central object to meet the far right platform on which a button appears. The fourth “switch” based movement (Figure 9e) pressed the button on the far right platform down, which triggered the green highlighted stack of three cubes to fall to the left. The fifth “switch” based movement (Figure 9f) rotated the green highlighted central object to meet this back platform. This “switch” based movement demonstrated how the Escher-like illusions are used in gameplay. Now the game character could reach the upper deck of the
green highlighted central object from the lower deck. The final “switch” based movement rotated the green highlighted central object one last time such that its upper deck met the target goal position. In combination, these “switch” based movements allowed the player to move their game character, the red sphere, from the start position it is seen in now to the goal position marked by the square target. Without activating and implementing these “switch” based movements, occupying this goal position would have been impossible.

![Figure 9: How a series of “switches” can alter the game environment.](image)

After all the “switches” for the existing level were present in the hand-built level, all the necessary pieces were represented and this recreation process could begin for the next level. This workflow was repeated until all the desired architecture components and “switches” had been recreated in the hand-built levels. We approximated this requiring five hand-built levels, each with unique game components.

With these constructed game levels, an inventory of the “switches” and game objects could be compiled. From the five hand-built levels, seven total “switch” styles were identified, see Table 1 for a detailed taxonomy of each style of “switch.” Each of these “switch” styles included its own requirements and could only be placed on specifically shaped game objects.
<table>
<thead>
<tr>
<th>“Switch” Style</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ladders</td>
<td>Vertical travel. Does not change orientation of the player. See Figure 10 for an example of how these “switches” appear in the game environment.</td>
</tr>
<tr>
<td>Doors</td>
<td>Teleportation between two cells in the level. Can change player orientation. See Figure 11 for an example of how these “switches” appear in the game environment.</td>
</tr>
<tr>
<td>Translation</td>
<td>Straight-line movement of one or more blocks. Can be linked to the movement (translation or rotation) of other blocks in the level. See Figure 12 for an example of how these “switches” can change the game environment.</td>
</tr>
<tr>
<td>Rotation</td>
<td>Rotation of one or more blocks. May change player orientation. Can be linked to the movement (translation or rotation) of other blocks in the level. This “switch” style was highlighted in Figure 9b-d and 9f-g.</td>
</tr>
<tr>
<td>45° Cube Divisions</td>
<td>One cube object is divided along the 45° diagonal to reveal a ramp. See Figure 13 for an example of how these “switches” appear in both their open and closed states.</td>
</tr>
<tr>
<td>Buttons</td>
<td>Once “hit” by the game character, the button is “pressed” down and a switch action is activated. This “switch” style was demonstrated in Figure 9e.</td>
</tr>
<tr>
<td>Button Reveals</td>
<td>A new button object slides up once another button object is slid down. See Figure 14 for an example of how these “switches” can be used within the game environment.</td>
</tr>
</tbody>
</table>

Table 1: Taxonomy of “switches.”

Figure 10: Ladder cube game object.  
Figure 11: Door cube game object.
To aid in the inclusion of these “switch” styles for the eventual level generation algorithm the “switch” movements for each style were reduced to clearly defined positional states. These states identified the key positions the “switch” afforded the object on which it was applied. For example, consider a rotational “switch” style. The primary states this style of “switch” highlighted were the four rotational degrees divisible by ninety. Defining such states for each “switch” style zeroed in on how that “switch” would affect the paths available. This simplified the representation of each “switch” style and enabled the eventual level generator to easily track the “switches” present in the environment.

Additionally, from the five hand-built levels, a variety of game objects were compiled. These game objects were groups of cubes and cylinders parsed from the larger built game environments that could be packaged separately and included within other levels. Each of these game objects had identified “switch” styles for which they were compatible. Particular game objects featured specific “switch” styles, like game cubes with the ladder texture. Other game objects contained multiple “switch” styles, like a rotatable platform which also contained a door. A small number of game objects in this inventory exhibited no “switch” style and were included as pure building blocks for the levels.

These two inventories formed the root for our level generator. From here we specified rules for how and where the compiled game objects could connect. The most important rule for generating valid levels was that the level contained a path from the game character’s start position to the goal position. This rule was checked and ensured by the A* based path
planning previously described, taking into account the various positional states that the “switches” offered. Other rules included requiring a start position, requiring a goal position, and making sure each game object’s connector piece had one or more matching connector piece(s) on some other game object present in the level. “Connector piece” refers to the piece of the game object where another game object could connect to it. The connector piece(s) for each game object in the inventory were defined manually. These rules acted similarly to the template used within the test-and-reject methods described above.

Our system to produce game levels performed a series of automated steps, followed by a few manual steps. This enabled us to partially procedurally generate *Monument Valley* style game levels that adhered to the production rules specified above. Our system started by satisfying the requirements for a start and a goal position. These positions were randomly selected from within the defined game space. This game space was represented as a 3D voxel grid of cells. From these two positions, our system computed a path using a breadth-first search technique. The positions selected for this path were then filled with a cube, the basic game object for *Monument Valley* style game levels. Using a breadth-first search technique ensured that the goal position could be reached by the game character through the adjacent cubes. However, these generated levels often had sections that were not navigable without implementing some style of “switch.” This process of adding “switches” to achieve full navigability was done by hand. One or more of the seven “switch” styles were identified as useful and then placed strategically into the level. In doing this step manually, the final rule, that all connector pieces have at least one other corresponding connector piece, was satisfied.

Our system of partial procedural level generation produced many levels. Certain levels were rejected if the level was deemed unable to be made navigable by adding any of the “switch” styles. Often this was because either the start or the goal position had been obstructed by a cube’s placement during the automated breadth-first search based steps. See Figures 15 and 16 for examples of such rejected levels. If the generated level was not rejected it was either manually adjusted and accepted or simply accepted if no adjustment was needed. The generated levels varied in length, size, and style of “switch” needed.

![Figure 15: Rejected because the start position is obstructed.](image1)

![Figure 16: Rejected because the goal position is obstructed.](image2)
4 Results and Analysis

The automated initial steps of our partial procedural level generator produced an extensive amount of paths. These paths, then, were either polished with manual “switch” additions or rejected for reasons mentioned above. In this manner, we obtained ten different new levels. Each “switch” style was represented within these ten new levels. Figures 17 through 26 show these ten new levels in their starting configuration.

Figure 17: Level Features: Button activated Escher-like illusion based translation “switch.”

Figure 18: Level Features: Linked translation “switch.”

Figure 19: Level Features: Button activated translation “switch.”

Figure 20: Level Features: No “switch” style needed, quite simple.
Figure 21: Level Features: Rotation “switch” and button activated 45° division “switch.”

Figure 22: Level Features: Rotation “switch” and button activated Escher-like translation “switch.”

Figure 23: Level Features: Ladder “switch” style.

Figure 24: Level Features: Door “switch” style.

Figure 25: Level Features: Linked rotation “switch.”

Figure 26: Level Features: Rotation “switch” and door “switch” style.
In comparison to the hand-built levels, like the one shown in Figure 9, these generated levels were much smaller and typically featured fewer “switch” styles. However, the time required to create these levels was much less. For reference, these ten generated levels were produced within one day while the five hand-built levels were produced over the course of two weeks.

To determine how these generated levels compared when in gameplay, we compiled all ten into a Mac OS X compatible application. At the start of the application, one of the generated levels was displayed at random. The user had to, then, find the path from the game character’s start position to the indicated goal position. Once the goal position was achieved, another of these generated levels was displayed at random. This process continued until all generated levels were played. During gameplay, the level name, game character position, and time since application startup were recorded to a white-space separated file every couple of seconds. An application with the same process and recording utilities was also compiled for three of the hand-built levels created at the start of this project.

Using these applications, we conducted a small pilot study with eighteen participants. The participants were asked to play the generated level application and the hand-built level application. At the conclusion of their gameplay, the participants’ white-space separated files were collected. From these collected files we computed the participant’s solve time for each level and, then, the average time required for the user to solve each generated and hand-built level. As indicated previously, the hand-built levels were larger and implemented more “switch” styles. Because of this, the time required to solve them was expected to be considerably longer. To account for this anticipated difference, we additionally obtained the amount of time taken by myself, the level designer/“expert”, to solve each of the generated and hand-built levels. With these values, we could determine the difference between the average solve time and the “expert” solve time for each level. Analyzing these values, seen in Figure 27, enabled us to draw conclusions about the difficulty of the generated levels in comparison to the hand-built levels. Note that generated level 1 through generated level 10 (the values labeled in purple in Figure 27) correspond respectively to Figure 17 through Figure 26.

Additionally, histograms displaying the distribution of the solve times for each level are located in the Appendix. These histograms also display the “expert” solve time in relation to the other participants’ solve times. It is important to include these histograms to better understand what the average solve time for each level communicated. Super-imposing the “expert” solve time onto these histograms also showed how many of the participants were close to this ideal measure and how many stray far from it. As expected, the “expert” solve time typically appeared in the left-most position. Occasionally, as in Figure 31 with generated level 3 and Figure 36 with generated level 8, the “expert” solve time did fall in line with the peak of the histogram distribution. These cases corresponded to lower reported values in Figure 27.

Figure 27 highlights how much more difficult the hand-built levels were to solve than the levels produced by our partial PCG generator. On average, the amount of additional time needed to solve each level compared to the “expert” solve time increased by about 67 seconds when the participant played the hand-built levels versus when they played the generated levels. This result was expected as the hand-built levels made use of more “switch” styles in their intended paths. This helped to disguise the intended path from the user.
These results offered some other interesting conclusions. We see that level 2, 6, and 9 of the generated levels had higher time differences than the other generated levels. Generated level 9 even surpassed hand-built levels 1 and 3 in the amount of additional time needed to solve it. These generated levels that had higher difficulty (Figures 18, 22, and 25 respectively) used a combination of “switch” styles and Escher-like illusions to form their paths. Their difficulty was emphasized even more when compared to generated levels 1, 4, and 8 (Figures 17, 20, and 24 respectively). These levels used up to one “switch” in their path and navigating the level was very straightforward. This difference suggested that what helped make the generated levels 6, 9, and 10 stand out from the other generated levels was the element of surprise. When attempting to navigate any of the levels, the player always knew where they wanted to finish. If a direct line could be drawn from where the player began to where this goal point was, there was no surprise in the navigation of the level. But, as with level 6, 9, and 10, the player could not easily draw this straight line and thus had to navigate through the environment to reveal the surprising way in which they could reach the final goal position. Implementing this element of surprise better into our partial PCG level generator would increase the difficulty of levels it can generate.

Hand-built level 2 stood out amongst all the other levels as being the most difficult to solve. Its starting and ending configurations can be seen in Figure 28. It was the only hand-built level that required a “switch” to change the plane on which the player character could travel. From this connection, we concluded that this plane-change requirement offered the most amount of surprise in level navigation. We see a similar plane-change requirement, while on a much smaller scale, in generated level 9 (Figure 23). And, much like how hand-
built level 2 stood out amongst the hand-built levels, generated level 9 stood out amongst the generated levels in terms of difficulty. These results argued that, if surprise within navigation was what contributed most to the difficulty of that level, then “switches” that changed the plane of travel and navigation were the best style of “switches” for increasing a level’s difficulty.

5 Future Work

The goal of this project was to develop an algorithm which could automatically generate levels containing features characteristic of Monument Valley. The work for this project concluded with the development of a partially automatic level generation algorithm. The next stages for this work will extend the current generation algorithm to a fully automated level generator. This system will need to handle the large number of potential graph states made possible by each “switch” style. Each of these “switch” styles has up to four positional states that it could be oriented in with additional variability in connection points and directional movement. Allowing for these things to be automatically implemented into a generated level will require accounting for every combination of these states and degrees of variability.

From where the work leaves off now, the idea of including subgoals within the generated levels offers a promising direction in which to direct future work. To achieve this the generation algorithm would choose one or two subgoal positions that the player character would have to navigate through before obtaining the final goal position. This would increase the difficulty of the generated levels as more “switches” would be required, as well as, increasing the overall size of the game level. Further, with these subgoals, there would be additional opportunities to introduce elements of surprise into the level navigation.
Beyond this point, an in-depth study of the manual aspects of our current generation system must be done. This study and analysis should identify common choices and ideas that can be reduced to more rules for the generator to follow. These new rules should guide the new random choices to be made by the generation system. These rules, in combination with a clever abstract representation of the “switches,” and all their variability, could lead to the fully automated generation system this project originally set out to produce.

Once this fully automated system is produced, additional work should be done to better understand what makes certain levels harder to solve. This work and research will enable us to identify ways in which the generator can produce levels with varied complexity. Such insight will assist in producing and displaying generated levels with increasing difficulty.

6 Summary

Procedural content generation for game levels offers access to an inexhaustible source of gameplay. For games that have received criticism for having very few levels, like Monument Valley, this potential is especially enticing and beneficial. Starting from scratch, this project worked towards a fully automated system that might tap into such potential for Monument Valley style games. Analyzing the requirements needed to produce these levels by hand, we identified various rules and procedures followed throughout the level building process. These identified aspects helped form a partially procedural generation technique similar to the PCG test-and-reject method. This generation system worked to satisfy these rules and perform these procedures through a series of automatic and manual steps. Our pilot study showed that even with our very simple generated levels, some puzzles were harder to solve than others. Future work will focus on understanding what exact features lead to this increased difficulty. We can, then, attempt to leverage this in our generation system and produce levels with tailored degrees of difficulty. Additionally, steps to convert our partial procedural system to a fully automated one, would allow for the full potential of PCG to be reached for these Monument Valley style games.
References


Appendix

Figure 29: Generated Level 1 Solve Time Histogram.
Figure 30: Generated Level 2 Solve Time Histogram.

Figure 31: Generated Level 3 Solve Time Histogram.
Figure 32: Generated Level 4 Solve Time Histogram.

Figure 33: Generated Level 5 Solve Time Histogram.
Figure 34: Generated Level 6 Solve Time Histogram.

Figure 35: Generated Level 7 Solve Time Histogram.
Figure 36: Generated Level 8 Solve Time Histogram.

Figure 37: Generated Level 9 Solve Time Histogram.
Figure 38: Generated Level 10 Solve Time Histogram.

Figure 39: Hand-Built Level 1 Solve Time Histogram.
Figure 40: Hand-Built Level 2 Solve Time Histogram.

Figure 41: Hand-Built Level 3 Solve Time Histogram.