Procedural Level Design for Platform Games

Kate Compton and Michael Mateas

Literature, Communication & Culture and College of Computing, Georgia Institute of Technology
kate@gatech.edu michaelm@cc.gatech.edu

Abstract

Although other genres have used procedural level generation to extend gameplay and replayability, platformer games have not yet seen successful level generation. This paper proposes a new four-layer hierarchy to represent platform game levels, with a focus on representing repetition, rhythm, and connectivity. It also proposes a way to use this model to procedurally generate new levels.

Introduction

Procedural level generation has been successfully implemented in several genres of games. Games have used level generation since 1980, when *Rogue* was written. *Rogue* is an ASCII graphic role-playing game, but one of its major innovations was its ability to generate an unlimited number of unique dungeon levels. *Rogue* levels are generated by partitioning dungeons into solid rock and empty space and filling the empty space with monsters and loot. Though the generated levels are not as complex as a hand-designed level could be, the innovation was not in the design of the dungeons, but in the ability to generate infinite numbers of them without human supervision.

More recently, games such as *Diablo II* and *Civilization* include similar level generation to that seen in *Rogue*. The never-ending supply of dungeons to plunder or continents to colonize gives the player many more rewarding hours of gameplay than could be available with even a large set of human-designed levels.

Platformer games would benefit from the replayability of generated levels, but as yet, there has been no commercially distributed platformer game that uses level generation. Platformer level generation is a more difficult problem than level generation in either RPG or strategy games, since very small changes, such as slightly changing the width of a chasm, can change a whole level from challenging to physically impossible. Rogue-like level generation can make heavy use of relatively unconstrained, random decisions. The playability of the level is ensured by the constraints implicit in the human-designed atomic units employed by the generator (e.g. rooms, hallway connectors, terrain tiles). In contrast, the playability of a platformer level is strongly determined by the relationships *between* units, requiring that these relationships be explicitly modeled and manipulated during level generation. The loosely constrained, random placement of elements that works in dungeon level and terrain generation, can easily lead to accidentally unwinnable platformer levels.

This paper proposes a model for the hierarchical elements and relationships between platformer levels, and an algorithm for level generation that is organized around the ontology. We have a working prototype of the level generator, and are currently implementing the complete architecture.

Model of Platformer Levels

The structure of this hierarchy is inspired by "A Novel Representation for Rhythmic Structure," a paper describing a model for representing complex rhythmic patterns in African and African-American music (Iyer 1997). Iyer describes a hierarchical representation that captures rhythmic repetition and the combination of short rhythmic sequences into longer, more complex passages. Though relating musical composition to the design of platformer levels may seem like a stretch, level design in platform games relies heavily on rhythm. Rhythmic actions help the player reach a "flow" state in games, a state of heightened concentration (Csikszentmihalyi 1990). When a player is "in the flow" or "in the rhythm" of a game, making jumps requires not only distance calculation, but also timing. Rhythmic placement of obstacles creates a rhythmic sequence of player movements, making each individual jump easier to time. Using a rhythmic but varied repetition of elements also serves an economic function; it gives the player a longer level to play through with only a few game objects. For example, by repeating and reshuffling a few basic elements such as pipes, blocks and platforms, it is possible for designers to construct long and interesting levels in the *Mario Brothers* games.

Components

Components, such as vines, platforms, little hills, and spikes, are the basic units out of which platform games are constructed. Often a component will have both an obstacle and a resting spot, which may be as simple as a stretch of empty air that must be jumped over, along with a platform to land on.

It is represented to the physics engine as a set of edges with certain properties (such as bounciness, slipperiness, potential for injury). It also has a visual representation on the screen, which signals the properties of the edge to the player, so that he can plan his moves accordingly.

A specific component's construction is determined in two ways. The dimensions are determined by the space they have to fill in the world. The rest of the construction is determined by two "tweaking" values, though how these affect the component is determined by what kind of component it is. As shown in the illustration, a vine's tweaks determine its length and offset, but a platform's determine the width of the gap and the angle of the platform. This allows a single type of component to vary widely when constructed with different values.

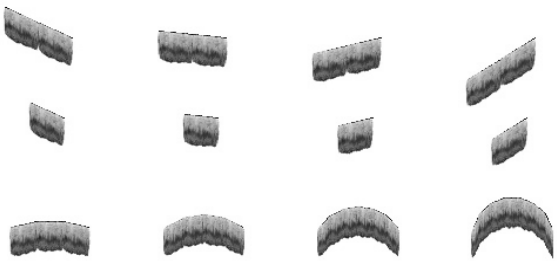

Figure 1: Components are constructed from parameters, allowing wide variety

Patterns

Of course, a single jump wouldn't make much of a game. Patterns provide the mechanism for grouping individual components into a longer sequence, while still maintaining rhythmic movement for the player. Currently the model defines 4 pattern types: basic, complex, compound, and composite.

Basic Patterns. A basic pattern consists of a component, either by itself, or repeated several times with no variation.

Complex Patterns. A complex pattern is a repetition of the same component, but with the tweaks changed according to some set sequence, such as a series of horizontal jumps of increasing width.

Compound Patterns. A compound pattern alternates between basic patterns made of two different types of components. An example of this would be a series of three horizontal jumps, followed by three spiky hurdles, followed by three horizontal jumps again. Note that compound patterns introduce a rhythmic pattern at a higher level of abstraction, in the form of changing the rhythm in a rhythmic manner.

Composite Patterns. A composite pattern consists of two components placed so close to each other that they require a different kind of action, or a coordinated action, that would not be required for each one individually. This requires the player to take the knowledge of the two problems and synthesize it into a solution. A timed spike requires the player to judge the right time to pass, and a gap requires a judgment of distance, but placed together, they require the player to coordinate the timing with the run-up to the jump.

With patterns one can construct strictly linear sequences. While such sequences may be challenging and fun to play through, non-linear elements like branching paths, setbacks, loops, and hidden levels are necessary to give players the ability to choose their own paths through the game. Since patterns, even the very elaborate ones, are strictly linear, the model needs another layer to deal with non-linear structure.

Cells and Cell Structures

Cells, the building blocks of non-linear level design, are constructed from linear patterns. A cell is an encapsulation of some pattern. At this level, the characteristics of the specific pattern are ignored; it is only important that it is possible for the player to get from one end of the pattern to the other.

A cell structure describes how cells may be connected in such a way that it affects gameplay. By studying the structure of both classic and recently released platform games, we are building a library of possible cell structures. A few of the most common cell structures found in platform games are illustrated in Figure 4.

A branch cell structure is the most basic way to add non-linearity to a level by giving players a choice of two paths to take. The player chooses not only between the two paths, but the two possible destinations, which might lead a player to choose a harder-looking path with the expectation of a greater reward at the end. Similar to a regular branch, a parallel branch creates two paths, both of which end at the same location.

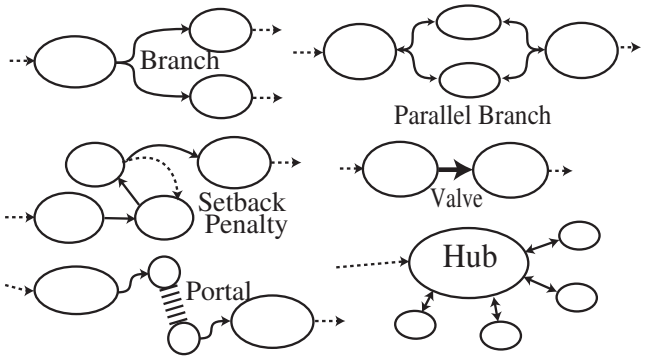

Figure 2: Cell structures create non-linear levels

Level Design Algorithm

This model can be partially represented as a context-free grammar, with a level as a start symbol, patterns and cells as non-terminal symbols, cells structures and pattern combinations as productions, and components as terminal symbols. Since generating strings from a grammar is very simple, the process of constructing a new valid level is equally straightforward. Each level is started with a cell

structure, which can be broken down into more cell structures several times as space permits. Those cells are each given a pattern, which, like the cells, can be recursively expanded into more patterns. Finally, each bottom level pattern is translated into a series of components.

The Physics Model and Difficulty Calculator

The physics model calculates accurately how the avatar will move in a world built of two-dimensional edges. Familiar platformer components, like vines, ladders, and platforms are represented as sets of these edges. The avatar either runs along connected edges, or jumps off at a normal to the edge. Like Super Mario Brothers, which does not conform to real-world physics, this model also allows the avatar to change direction in mid-air. Given two edges, basic ballistics calculations can determine where, if at all, the avatar would hit the second edge after jumping off of some point on the first edge.

The system calculates the spatial window the avatar can start from in order to successfully jump from one component (here represented as a set of edges) to another. In addition, it can estimate the temporal window the player has to change direction mid-air in order to change an unsuccessful jump into a successful one. With knowledge of the player's aiming and timing abilities, this allows the system to approximate the difficulty of a jump. If there are a series of components, like in a pattern, it can calculate the difficulty of getting through the entire pattern.

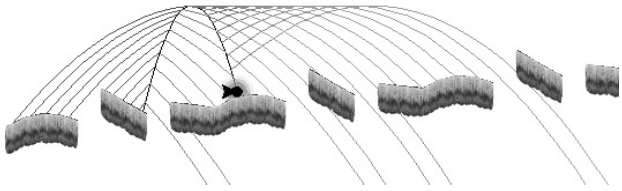

Figure 3: The possible trajectories, including mid-air jumps, available from this starting position

Pattern-Building

To build a pattern, the system starts with a short list of possible component types and two bracketing components already present in the level that mark the beginning and the end of the desired pattern. For each possible set of component types and number of components, the optimal pattern is built, and then the best of these is selected as the final pattern.

The optimal pattern for any given set and number of components is determined using a simple hill-climbing algorithm, with the system hill-climbing towards the target difficulty. Each component type has a set of parameters defining the component. For example, for platforms, the two component parameters are the length and angle of the platform. The component parameters define the difficulty surface on which the system hillclimbs.

To start the hill-climbing search, the system creates an initial pattern by first selecting random values for the component parameters (e.g. random angles and platform lengths), and then evenly sub-dividing the space between the start and end locations of the pattern with these components. To then find a pattern close to the desired difficulty, the system adjusts the component parameters in the direction of steepest assent.

The fitness graphs for simple and complex pattern are relatively smooth and not riddled with local maxima, so this hill-climbing algorithm has a fair chance of returning the strictly optimal solution. Even non-optimal solutions are usually not far enough off from the desired mark to be noticeable to a player. Catastrophic results, like a pattern of either impossible or trivial difficulty, are overwritten by testing multiple sets and numbers of components.

Cell Structure Decisions

Patterns are generated for an initial cell. As the player reaches the outer limits of that cell, a new cell, or set of cells, is connected to it with one of the cell structures mentioned above. This new cell is populated with patterns through the previously described algorithm. Which cell structure to use is based on pre-determined values for the level, such as the level of desired branching. The direction and dimensions of the cell are based on physical constraints of the already developed space, as well as some degree of randomness.

Though not implemented at this time, this could be used for dynamic difficulty adjustment (Hunicke 2004). If the system monitors the player's progress through the level, perceived preferences (through which path the player takes) and the success and failure rate, this could be used to create a player profile and inform the generation of successive cells.

Current and Future Work

The pattern builder algorithm works successfully. It is capable of constructing and optimizing single patterns. At this time, an algorithm for building cell structures has been designed, but is not yet implemented in the prototype.

References

Csikszentmihalyi, M. 1990. Flow: the psychology of optimal experience. New York, NY, HarperCollins.

Hunicke, R., Chapman, V. 2004. "AI for Dynamic Difficult Adjustment in Games." Proceedings of the Challenges in Game AI Workshop, Nineteenth National Conference on Artificial Intelligence.

Iyer, V., Bilmes, J., Wessel, D. and Wright, M. 1997. A Novel Representation for Rhythmic Structure. In Proceedings of the 23rd International Computer Music Conference, (Thessaloniki, Hellas, 1997), International Computer Music Association, 97-100.