Rhythm-Based Level Generation for 2D Platformers

Gillian Smith, Mike Treanor, Jim Whitehead, Michael Mateas
Expressive Intelligence Studio
University of California, Santa Cruz
Santa Cruz, CA, USA
{gsmith, mtreanor, ejw, michaelm}@soe.ucsc.edu

ABSTRACT

We present a rhythm-based method for the automatic generation of levels for 2D platformers, where the rhythm is that which the player feels with his hands while playing. Levels are created using a grammar-based method: first generating rhythms, then generating geometry based on those rhythms. Generation is constrained by a set of style parameters tweakable by a human designer. The approach also minimizes the amount of content that must be manually authored, instead relying on geometry components that are included in the level designer's tileset and a set of jump types. Our results show that this method produces an impressive variety of levels, all of which are fully playable.

Categories and Subject Descriptors

K.8.0 [**Personal Computing**]: General – *Games.* I.2.4 [**Artificial Intelligence**]: Knowledge Representation Formalism and Methods – *Representations (procedural and rule-based).*

General Terms

Design, Human Factors.

Keywords

Games, levels, procedural generation, 2D platformers.

1. INTRODUCTION

Despite hundreds of successful games with interesting levels, the science behind level design is still imperfectly understood. As a result, existing methods for procedural level generation tend towards either terrain generation or fitting together large, hand-authored chunks of a level using heuristics to determine their ordering. While these methods can produce pleasing results, they can also lead to repetitive levels with a high authorial burden. This is especially true for 2D platformers, where the entirety of the player's experience is heavily influenced by the layout of levels.

The main elements of 2D platformers are well known: platforms, enemies, collectibles, and hidden areas. However, the ways in which these elements can be combined to make an interesting level is highly variable. Surprisingly little has been written on how to create these levels; our recent work [14] provides a first step towards defining a vocabulary and analytical framework for examining these levels.

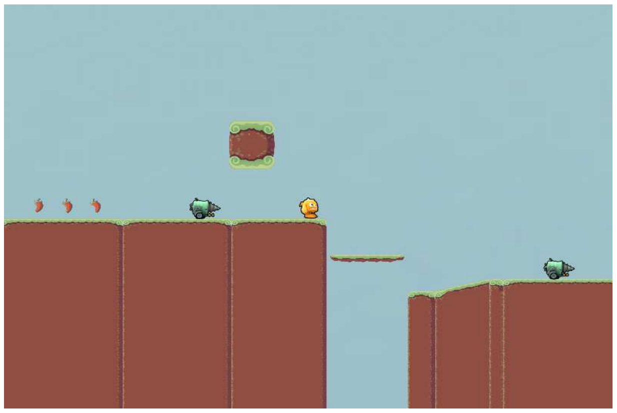


Figure 1. Part of one of the many 2D platformer levels that can be created by the rhythm-based level generator.

We believe a key underlying idea behind 2D platformer level design is the notion of rhythm, and the timing and repetition of distinct user actions [4][7]. Players strive to navigate complicated playfields full of obstacles and collectible items; manually designed levels frequently contain a series of challenging jumps that must be timed perfectly. This paper presents a realization of this theory of level design in the form of an algorithm for automated level generation. An example of the kinds of levels it can produce is shown in Figure 1. In order to capture the importance of rhythm, the level generator is designed with a two-tier, grammar-based approach, where the first tier is a rhythm generator, and the second tier creates geometry based on that rhythm. The separation of tiers ensures that the intended rhythm is always present, regardless of geometric representation.

This work addresses several questions:

1. Is rhythm-based level generation a feasible approach?
2. Is the rhythm-based method for generating levels a significant improvement over existing approaches?
3. How can human control be exerted over the generated levels?

This work is similar to that of Nelson & Mateas in that its goal is not entirely to replace a human designer but rather to further understanding of deeper issues in level design and provide an "intelligent … design tool to support human … designers." [6] The deeper issue of level design for platformers that is explored in this paper is the rhythm the player experiences during the course of a level.

Although the level generator itself is fully automated, it can take input from a human designer to restrict the kinds of levels that it will produce. The designer is provided a set of style "knobs"; these knobs dictate a general path through the level, the kinds of rhythms that can be generated, the types and frequencies of geometry components, and how collectible items (coins) are interspersed throughout the level. Adjusting these style parameters can drastically alter the generated levels.

This paper presents the details of the rhythm-based generator, and examples of the kinds of levels it can produce. The primary contribution of this work is its use of a two-layer grammar-based geometry generation system using rhythms that are made up of verbs, and the use of these verbs as a driver for geometry generation. The level generator can be guided by a human designer who tweaks the style of levels. The paper also presents a set of algorithms that operate on the level as a whole, including critics that determine how good a level is according to style parameters, and a method for decorating a level with coins.

2. RELATED WORK

Ever since Rogue [16] became popular in the mid-1980s there has been interest in automated or procedural level generation. Even as recently as 2007, Hellgate: London [5] uses some of the same techniques to generate its levels. Rogue-like level generation operates on the principle of encoding level design knowledge into an algorithm that randomly generates levels. Dwarf Fortress [1] makes perhaps the most extensive use of procedural generation; everything in the game is automatically generated, including terrain, underground spaces, and even a world history.

Within the genre of 2D platformers, Markus Persson's *Infinite Mario Bros!* [11] generates levels by probabilistically choosing a few idiomatic pieces of platformer levels and fitting them together. While this technique produces levels that, on the surface at least, look and feel a great deal like Mario, over time it becomes apparent that there is very little variety between levels due to the repetition of idioms. Our work attempts to solve this problem by using a much finer granularity for geometry—the pieces that we fit together are much the same as those that would be found in a level designer's tileset. The designer need only classify those pieces by a set of rules for when they should show up. One consequence is a lower authoring burden as compared to existing level generation techniques, while producing a larger variety of levels.

Spelunky is another 2D platformer with procedurally generated levels [18]. This game uses Rogue-like level generation techniques to fill rooms with passages, treasure, and enemies. *Spelunky* belongs to a sub-genre of 2D platformers that is not well-suited to our approach of rhythm based level generation, as the challenges in the game are not dexterity-based, but rather exploration-based. Examples of the dexterity-based platformers that we target are Super Mario World [9] or Sonic the Hedgehog [12].

Other work that specifically addresses procedural level generation for 2D platformers is that of Compton & Mateas [3]. They also split up levels into smaller sections and generate those sections from individual platform tilesets; they call their sections "patterns" where we call ours "rhythm groups". Our work is quite different, though, in that our two-tiered approach to rhythm group generation, by explicitly separating rhythm generation and geometry generation, means we can represent more variety in levels than their pattern-building approach, in which rhythm is implicitly determined from geometry decisions. The use of style parameters also provides authorial control over our generated levels, missing in Compton & Mateas's work.

Recent work on procedural level generation in other game genres includes Togelius et al. [15], who take a difficulty-based approach for evolving levels for racing games. Their heuristics are based on creating more "fun" levels for players, whereas our critics operate given the designer's goals for the level.

Charbitat is an exploration into procedurally generated levels built by Michael Nietsche et al. [10]. Their work focuses on an evolving world that is based entirely on player actions; the components Charbitat uses to create a potentially infinite world are procedurally generated terrain tiles with randomly placed components within them. Although different from our work in terms of both genre and approach, a common idea is shared: a "procedurally determined context is necessary to structure and make sense of this [procedurally generated] content" [2]. While their context comes in the form of automatically generated quests, ours comes in the form of a rhythm for the geometry to follow and the style parameters we make available to a human designer.

Although not in the realm of content generation for games, other grammar-based approaches are related. The Instant Architecture [17] project uses grammars for both architecture generation and ensuring the correct distribution of attributes. Tableau Machine [13], an art generator, uses a grammar for generating art, and their method for solving the problem of under-constrained results by generate-and-test was the inspiration for our own implementation of critics.

3. SYSTEM OVERVIEW

The rhythm-based approach for generating levels is described in Figure 2. It begins with a two-layered grammar-based approach for generating small chunks of a level, called rhythm groups [14]. The first stage creates a set of actions that the player will take,

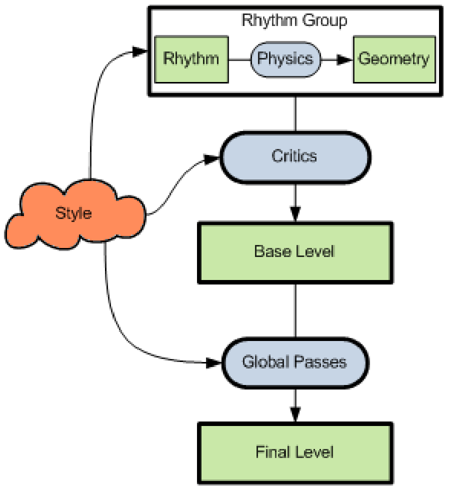


Figure 2. Level generation algorithm. Green squares indicate generated entities, while blue circles indicate constraints. Style parameters influence many aspects of level generation.

constrained to form a rhythm. The second stage uses a grammar to convert this set of actions into corresponding geometry according to a set of physical constraints.

To form a complete level, rhythm groups are fit together side by side, bridging them with a small platform that acts as a rest area for the player. Many different levels are generated, forming a pool of candidate levels which are then tested against a set of critics to determine the best level. This level then is referred to as a "base level", which can be improved through decorating it with coins and tying its platforms to a common ground point.

At all stages of level creation, style plays an important role. Style is represented as a set of parameters that a human designer can tweak. Parameters include the frequency of jumps per rhythm group, and how often a spring should be generated for a jump.

4. RHYTHM GROUP GENERATION

Rhythm groups are small, non-overlapping sections of a level that encapsulate a sense of rhythm for the player. For example, a rhythm group could be three short hops with minimal movement in between, or a repeated section of long runs with small jumps in between. To capture this sense of rhythm, rhythm and geometry are separated through a two-tiered generation system. All rhythm groups built off of the same rhythm may have a different geometric representation, but ultimately "feel" the same to the player. Rhythms maintain a length, density, beat pattern, and the ability to reflect or repeat that pattern. This section discusses the generation of rhythms and how they drive geometry generation through a set of physics constraints.

4.1 Rhythms

The rhythm generator chooses a set of verbs corresponding to player actions; currently, the verbs it can choose from are "move" and "jump". It also chooses the times that these verbs should begin and end. This produces rhythms such as the following:

```
move 0 5
jump 2 2.25
jump 4 4.25
move 6 10
jump 6 6.5
jump 8 8.5
```

In this example, the player starts moving at 0 seconds, and continues moving until 5 seconds. While moving, the player jumps once at the 2 second mark and again at the 4 second mark, each jump lasting .25 seconds. The length associated with the "jump" verb corresponds to the amount of time the player will hold down the "jump" button. Since different hold times influence the height of the avatar's jump, it is important to keep these jump types distinct. For example, the player may hold the button down for only .25 seconds, resulting in a very short hop, or may hold it down much longer for a long jump. The example rhythm above has the player moving almost all the time, except for a 1 second "wait" at time 5 due to the lack of a movement or jump command occurring at that time.

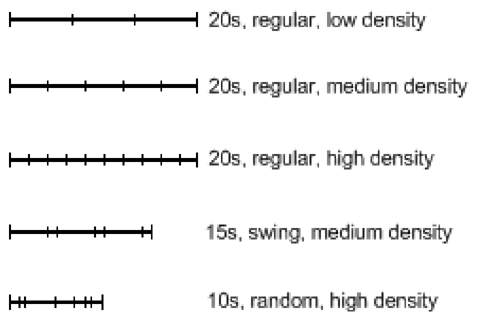

20s, regular, low density
20s, regular, medium density
20s, regular, high density
15s, swing, medium density
10s, random, high density

Figure 3. Examples showing the effects of varying the length, type, and density of a rhythm. Lines indicate the length of the rhythm, and hatch marks indicate the times at which an action will begin.

To introduce variety, the beat type, length, and density of the rhythms can be modified. The beat type refers to the way the start times of actions are organized to create rhythm. The current options are "regular", the beats spaced evenly, "random", the beats placed at random intervals, and "swing", the beats placed in a swung beat as understood in music theory. The length can be 5, 10, 15, or 20 seconds, and the density can be "low", "medium", or "high". Density refers to the number of actions that should be performed by the player during the rhythm group. Figure 3 shows examples of the types of rhythms that can be generated.

4.2 Geometry

The geometry generator is responsible for taking rhythms from the rhythm generator and creating a possible interpretation for them. These interpretations can be wildly varied due to the number of options for each verb and the way jumps must be handled due to physics constraints.

Received verbs are first converted into a list of movement states and a queue of jump commands. The movement states ("moving" or "waiting") have a length associated with them, and the jump queue maintains the start time and length of each jump. For example, the rhythm:

```
move 0 4
jump 2 2.25
move 6 10
jump 6 6.25
jump 8 8.75
```

forms the following set of states and jump queue. Figure 4 shows a graphical representation of this rhythm, with a solid line denoting movement, a dashed line denoting waiting, and hatch marks showing jumps.

States: `[4, moving], [2, waiting], [4, moving]`

Jumps: `[2, short], [6, short], [8, long]`

However, the jump length merely tells how long the player holds the "jump" button; the actual time in the air varies based on which jump type is chosen; a jump across a flat gap takes considerably less time than a jump onto a spring. Therefore, jumps consume some amount of the movement state list at the time the jump occurs. For example, assuming there are only two jump options

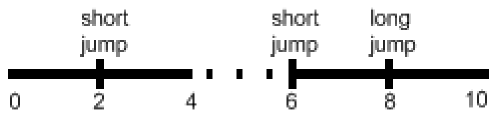
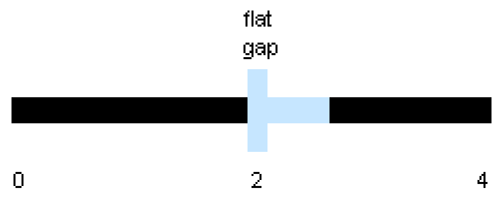
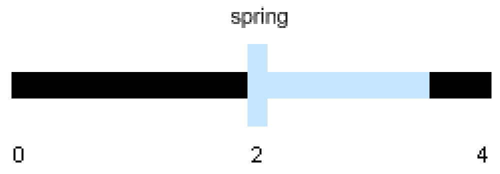


Figure 4. An example rhythm.



```
Moving  2.0 seconds
Jumping 0.5 seconds
Moving  1.5 seconds
```



```
Moving  2.0 seconds
Jumping 1.5 seconds
Moving  0.5 seconds
```

Figure 5. Two different types of jump can contribute to different movement and jump state lengths. The blue area is the amount of time consumed by the jump being in the air.

(flat gap and spring), and a jump across a flat gap takes 0.5 seconds and a jump onto a spring takes 1.5 seconds, processing the first jump in our example could result in one of the two configurations shown in Figure 5.

The states that the jump consumes are used to determine how large the gap is, using the physics model; the rhythm of stopping and starting motion is still felt by the player while in mid-air. Note also that the type of jump that the geometry generator chooses at any time must be constrained by the start time for the next jump, ensuring that one jump finishes before the next jump begins. This keeps complete control over rhythms in the hands of the rhythm generator.

4.3 Geometry Grammar

The movement states that are not consumed by jumping and queued jumps form the non-terminals in the geometry generation grammar (Figure 7). The "waiting" state is meaningless on its own, as there must be something to wait for. Generating geometry for a wait state involves looking ahead in the state list.

Figure 6 shows an example of an initial rhythm (top of Figure 6) and four different geometries that can be generated from it. Figures 6(b) and 6(d) show how moving platforms consume a wait-move-wait; other interpretations of two wait-moves in a row are shown in Figures 6(a) and 6(c). The dotted lines show how the first three jumps correspond to geometry; note that waits introduce variation to the physical length of the rhythm group and jumps that occur after waits no longer "line up" with the sample rhythm. This figure also show many different interpretations for each jump.

4.4 Physics Constraints

The physics model maintains information about the avatar's capabilities and the different types of jumps available. The model includes the avatar's size, maximum movement speed, initial jumping velocity, and the height the avatar can jump given a short, medium, or long jump button press. For jumps, the model includes the in-air time for each jump type, as well as the relative height difference for the two platforms on either side of a jump and the velocity imparted by a spring. Available slopes for platforms are also recorded. This model is ballistics based, extended to allow variable jump heights; the longer the player holds down the jump button, the longer the avatar will be in the air. This style is common for Mario-style platformers. More advanced player physics, such as double-jumping or wall-jumping, are not supported.

The physics model is responsible for providing the rhythm generator with information about the amount of time the avatar will be in the air for a given jump hold time. Its primary use, however, is to provide constraints for the geometry generator, thus ensuring that all generated levels are playable.

5. CRITICS

Complete levels are generated by piecing together rhythm groups and connecting them with a "rest area" platform. Rhythm groups can optionally be repeated before this rest area, which can provide additional challenge [7] and more visual consistency.

```
move  0.00  8.00
jump  2.00  2.25
jump  4.00  4.25
jump  6.00  6.50
move 10.00 12.00
move 14.00 20.00
jump 16.00 16.25
jump 18.00 18.25
```

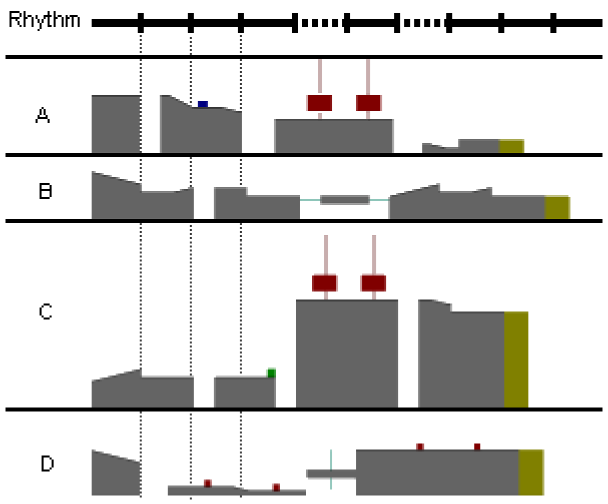


Figure 6. Four possible geometry interpretations of the provided rhythm. Small red boxes denote enemies to kill, small green boxes denote springs, blue boxes are enemies to avoid, large red boxes are stompers that follow the associated line, and platforms on green lines are moving platforms. The large green platform is the joiner from this rhythm group to the next.

```
Moving      →  Sloped | flat_platform
Sloped      →  Steep | Gradual
Steep       →  steep_slope_up | steep_slope_down
Gradual     →  gradual_slope_up | gradual_slope_down

Jumping     →  flat_gap
               | (gap | no_gap) (jump_up | Down | spring | fall)
               | enemy_kill
               | enemy_avoid
Down        →  jump_down_short | jump_down_medium | jump_down_long

Waiting-Moving → stomper

Waiting-Moving-Waiting -> moving_platform_vert
              | moving_platform_horiz
```

Figure 7. Geometry generation grammar. Player states derived from the generated rhythms are non-terminals in this grammar.

Unfortunately, a common problem with using design grammars is over-generation; even with constraints on rhythms and the types of jump, the level design space specified by the grammar is simply too large and results in many levels that are undesirable. Attempts to tighten the grammar to eliminate undesirable levels generally results in eliminating some number of desirable levels. The basic problem is that design grammars are good at capturing local constraints, but some design constraints are global. This problem is addressed by generating many levels and testing them against a few different critics to determine which is best. Given a generated level, a critic can perform tests over the entire level to determine how well a given global constraint is met. Our level generator has two critics: fitting to a designer-specified line, and ensuring the correct distribution of level components according to the designer's style specification. The importance of each critic can be adjusted to alter the kinds of levels produced.

5.1 Line Distance Critic

One input a human designer provides to the level generator is a path that the level should follow, providing control over where the level should start and end and the general direction it should follow in between. This critic serves to rein in the large space of levels that can be generated and allow the human designer to assert additional control; existing, human-designed levels in Super Mario World [9] and Sonic the Hedgehog [12] tend to follow regular paths, rather than meander aimlessly through space. The path is specified as a set of line segments which are not necessarily connected end-to-end. In order to fit levels to this line, the distance between platform endpoints and the line are computed. Good levels are those with minimal overall distance.

5.2 Component Style Critic

There are many different parameters available for a human designer to alter, affecting all stages of the generation process; changing these parameters produces drastically different levels. Rhythm style parameters provide control over rhythm length, density, beat type, and beat pattern, as well as the frequency of a beat being a jump or a wait. In the geometry stage, there are parameters for the frequency of different jump types, and the probability of sloped platforms. These parameters contribute towards visual variation between levels, and are used in the component frequency critic. The path that a level should follow and the relative importance of following that path vs. having the correct frequency of components are style parameters for the critic stage. And finally, there is control for the numbers of coins in a group, the amount of space between those coins, and the probability for a coin to appear over a gap.

Style parameters are used in geometry generation as a weight for how often each component should be chosen; however, these weights only guarantee that over a large number of rhythm groups the frequency of each component will asymptotically approach the probability of it appearing. Each rhythm group is created by pulling a relatively small number of geometric components from a pool of generate-able components, and then levels are created by choosing a small number of rhythm groups from a large number of randomly generated rhythm groups. Therefore, there is no guarantee that the observed frequency of components will match the expected frequency, just as in a series of 10 coin flips there is no guarantee that 5 will come up heads and 5 will come up tails. The style critic is designed to ensure that the number of each type of component in the level best matches the probability distribution formed by the style parameters.

This critic works by applying the chi-square goodness-of-fit test to each potential level, and choosing the level with the smallest test statistic, representing the level that has the closest component distribution to the desired style.

5.3 Combining Critics

There are many cases in which these two critics will contradict each other as to which level is best. For example, a level that is heavily weighted towards having springs will not fit a line that slopes only downward. To resolve this contradiction, the level with the lowest weighted average of the two critics is selected, where the weight on each critic determines its importance.

6. GLOBAL PASSES

Although most of the level can be created with local generation techniques, it is important to be able to reason over these levels. The rhythm-based generator has two "global pass" algorithms: tying platforms to a common ground plane, and decorating levels with coins. These algorithms reason over both the player states and the geometry associated with them. Tying platforms to the ground provides some visual consistency for levels and removes the possibility of the player unintentionally falling off platforms that are spaced apart from each other.

Collectible items are treated as decoration over a level; this is stylistically consistent with Mario and Sonic-style platformers. However, games that treat collectible items as a primary goal, such as Donkey Kong Country [8], would perhaps be better served by placing coins during geometry generation.

Two rules determine collectible item placement:

1. Place a group of coins on a long platform of predetermined length that has no action other than move associated with it.

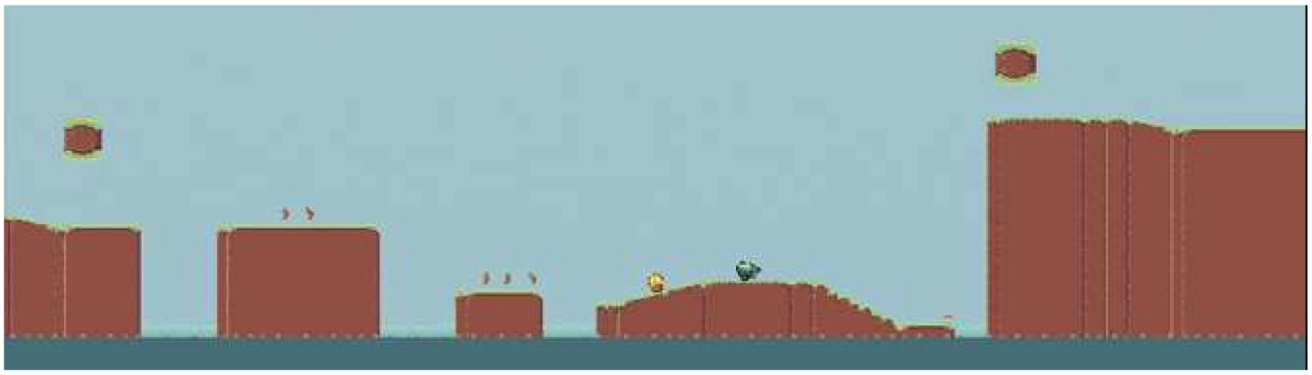


Figure 8. A zoomed out screen shot of a level realized in Torque Game Builder.

2. Reward the risk of jumping over a gap, and provide guidance for the ideal height of a jump, by placing one at the peak of jumps that go over gaps.

It would also be easy to specify new, powerful rules for coin placement, such as along the path of a spring or fall to guide the player in the right direction, due to the separation of rhythm and geometry.

7. CREATING PLAYABLE LEVELS

In order to make the generated levels playable, they are translated from the generator's internal representation of a level to the Torque Game Builder's level format. Torque Game Builder is a 2D game engine that is widely used for independent and commercial games. Because most platformers use tiles for visuals, the translation process requires reasoning about tiles and their size. This would otherwise not be a concern of the generator as it uses simple shapes to represent its geometry. The generator ensures that all platform dimensions are a multiple of a specified tile size. Then when translating to Torque, a tilemap is created that accounts for drawing the correct tile for the geometry. A section of one of our levels is shown in Figure 8.

8. EVALUATION

This section evaluates the usefulness of each component in the level generator, using a process of hypothetically removing each component, then evaluating the impact that removal has on the generated levels.

8.1 Rhythm Generation

We claim that generating rhythms is important for platformer level generation as it introduces structure and context for the levels. Purely randomly placed platforms are not sufficient for an interesting level, and remove the important concept of rhythm-based challenge from the game. Figure 9 shows an early attempt at generating rhythm groups by randomly placing platforms along a line. Stringing these platforms together created playable levels, but they were visually unappealing and not much fun.

In contrast, the rhythm-based levels are more appealing and have a better sense of flow. The length, density, and style of rhythm can be modified for each rhythm group, leading to interesting, highly varied levels.

8.2 Geometry Generation

Obviously any level generator requires a geometry generation component; without one, there would be no levels to evaluate. However, we can examine how our fine-grained, grammar-based approach to geometry generation is an improvement over existing techniques.

The rhythm-based generator provides 14 different kinds of geometry for jumps, 5 different kinds of geometry for moves, three different types of rhythm, four different lengths of rhythm, and three different densities for rhythm. As a result, the number of rhythm groups that can be generated is considerably larger than the number of rhythm groups that a human could author. Infinite Mario Bros! [11], which is the best platformer level generator to our knowledge, has only five manually authored sections of a level. While these can be combined in a potentially infinite number of combinations, the space of levels we can create is much bigger and more varied.

8.3 Critics

Critics are vital to the success of our level generator due to the huge space of potential levels that can be created without them. These levels can be quite repetitive, potentially with too much space devoted entirely to running along long platforms or jumping to kill many enemies. This is a common problem in using grammars to generate content. The critics operate on a set of 1000 candidate levels and serve to ensure that the final generated levels are not only playable, but also interesting. The evaluation of critics answers how distance and component measures react to different permutations of style parameters and how their existence produces better levels.

The distance critic finds the best fit of a generated level to the control line, as described in Section 5.1. Figure 10 shows the

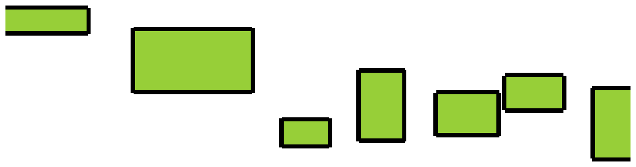


Figure 9. A "rhythm group" created with no sense of rhythm. The algorithm for making this geometry places platforms randomly along a line. Although the levels created with this method are playable, they have no sense of flow and are not visually appealing.

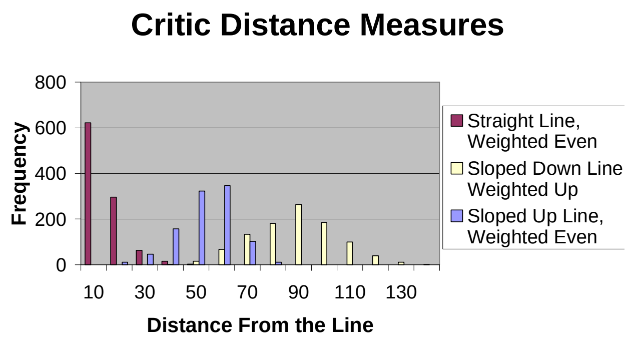


Figure 10. Frequency of distance from the control line with varying style parameters. "Straight Line, Weighted Even" denotes a straight line with all probabilities for jump components weighted equally. "Sloped Down Line, Weighted Up" is a line that slopes downward where probabilities for jumping up are weighted highly. "Sloped Up Line, Weighted Even" is a line that slopes upward where probabilities for jump components are weighted equally.

frequency of the measured distance from the control line given different configurations of style parameters. In the figure, the x-axis contains the bins of values returned by the distance critic, and the y-axis denotes the frequency that a distance inside that bin was recorded over a series of 1000 runs. Figure 11 provides context for these distance values by giving example levels and the distances assigned to them. Given these wildly different line measures, it is clear that it is impossible to rely on a good fit to the line through careful selection of style parameters alone. Therefore, the distance critic is vital to ensure a fit to the control line.

The component critic attempts to minimize the distance between observed and expected frequencies of components, as described in Section 5.2. Figure 12 shows the frequency of these component distances over 1000 runs. As expected, the frequency of generated components is generally close to the component frequency specified in the component style parameters. Figure 13 compares the observed component frequency for two different generated

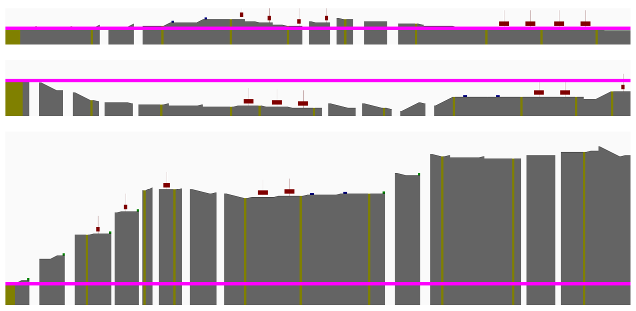


Figure 11. Examples of levels that are different distances from the horizontal control line. The top level has a distance measure of 2.1474, the middle a distance measure of 9.8020, and the bottom a distance measure of 45.3052.

levels (each with its computed critic values) to the expected component frequency. The level with the lower distance (critic value 4.0) has a jump component frequency distribution that is very close to the expected component frequency. The level with high distance (critic value 83.4) is far from the expected frequency; half of the generated jumps are of type 4, and many jump types not present. Although the values for the component critic have less variation than that of the line critic, it is still important to have this critic as it ensures the best fit to the designer's requested style.

In combining the two critics, it is important to find a compromise between the best fit to the line and the best fit to desired component frequency. Figure 14 shows the distribution of the combined critic measure with even weighting of the two critics. Several observations can be made of this frequency distribution. Even though two critics are being combined, the distribution still has a distinct peak far from the lowest value. This highlights the importance of generating a large number of levels, and then choosing the one with lowest combined critic value.

An underlying assumption is that optimizing for both critic values yields the best level. It is certainly possible that one of the 999 rejected levels is fun and interesting to play. The critics select for the level that is closest to the intent of the level designer, as expressed by the control line and style parameters. Other critics

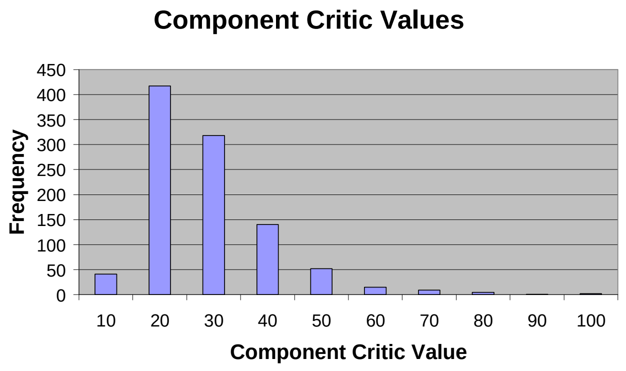


Figure 12. Frequency of component critic values over 1000 generated levels. The x-axis contains the bins of component critic values, and the y-axis denotes the frequency that a critic value falls into one of those bins. These frequencies stay roughly the same regardless of style parameter choices.

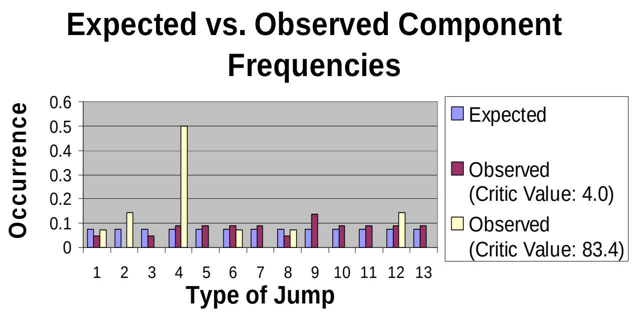


Figure 13. This graph shows expected vs. observed component frequencies for two different critic values. There are 13 different jump types in our generator; for simplicity, we denote each of these by number on the x-axis. The y-axis shows the percentage occurrence of each jump type given the critic value.

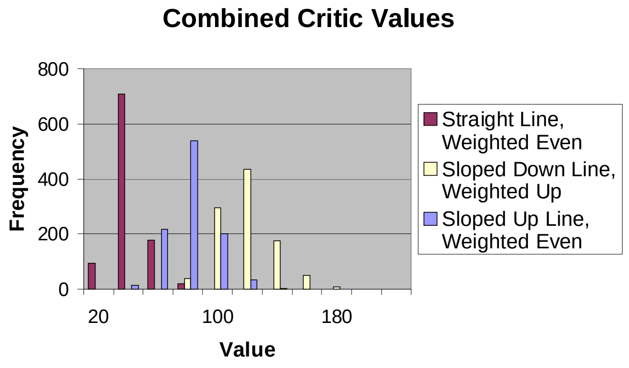


Figure 14. This graph shows the frequency for combined critic values. The same style parameters have been used for this graph as for Figure 10.

are possible, as are different valorizations of existing critics.

8.4 Global Passes

Our base representation for levels gives us the ability to reason over both the actions the player is performing and the geometry associated with them. We argue that this is vital for reasoning over rhythm-based levels, and that reasoning over rhythm-based levels is indeed vital. The two global passes we have implemented – tying platforms to a common ground plane and decorating levels with coins – are only examples of ways to reason over levels. Global passes are also potentially useful for other analysis, such as difficulty assessment.

9. CONCLUSION

This paper presents a method for automatically generating 2D platformer levels, founded on the important concepts of rhythm and pacing. However, also important in designing platformer levels is an idea of how challenging a level will be for the player. Levels often increase in difficulty as the game progresses, gently introducing patterns and encouraging mastery before putting them into a more demanding setting.

With this in mind, we initially considered a difficulty-based approach for level generation, rather than our current rhythm-based approach. This method would have been to assign difficulty measures to different "idioms" for platformer levels, such as jumping onto a platform with a moving enemy, or jumping across a variable width gap. These "idioms" then would be fit together in much the same manner as existing level generation techniques, but with heuristics controlling the difficulty of each chunk which would somehow contribute to the overall difficulty of the level. However, we were concerned that this would not provide sufficiently varied levels, and wanted to recognize the importance of rhythm to platformers.

Difficulty analysis remains both significant and non-trivial; indeed, it is one of many directions that this research could push future work. One way we would like to explore this analysis is to incorporate new critics into the generation process. Other future work directions include pushing the limits of the rhythm-based generator by adding more verbs and geometry types, and performing an ethnographic study of level designers to determine how to turn this research system into an authoring tool. Finally, it would be extremely useful to have more quantitative methods of analyzing levels; this is a research area which, to our knowledge, has not been explored at all. Without such methods, it is difficult to objectively compare our work to manually created levels.

The rhythm-based method for platformer level generation allows for wide variety in generated, playable levels while providing for a strong sense of pacing and flow. The system is easily extensible for new verbs, geometry, and global adjustments to the level, and provides a human designer with stylistic control over the generated levels. We hope that our work will forward interest in procedural level generation and analysis and result in a deeper understanding of 2D platformer levels.

10. ACKNOWLEDGMENTS

We thank Phillip O'Shea, creator of the Torque Platformer Starter Kit, for providing royalty-free art for use in our generated levels.

11. REFERENCES

[1] Adams, T. Slaves to Armok II: Dwarf Fortress (PC Game). Published by Bay 12 Games, September 2008.

[2] Ashmore, C. and Nitsche, M. The Quest in a Generated World. *Proc. 2007 Digital Games Research Assoc. (DiGRA) Conference: Situated Play*, pp. 503-509. Tokyo, Japan. Sept. 24-28, 2007.

[3] Compton, K. and Mateas, M. Procedural Level Design for Platform Games. *Proc. 2nd Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE '06)*, Stanford, CA, 2006.

[4] Csikszentmihaly, M. *Flow: The Psychology of Optimal Experience*. Harper Collins, NY, 1991.

[5] Flagship Studios. 2007. Hellgate: London (PC Game). Published by EA, October 31, 2007.

[6] Nelson, M. and Mateas, M. Towards Automated Game Design. *Proc. 10th Congress of the Italian Association for Artificial Intelligence (AIIA 2007)*, Rome, Italy. September 10-13, 2007.

[7] Nicollet, V. 2004. Difficulty in Dexterity-Based Platform Games. Gamedev.net. Last Accessed: December 11, 2008. http://www.gamedev.net/reference/design/features/platformdiff/

[8] Nintendo. 1994. Donkey Kong Country (SNES).

[9] Nintendo. 1991. Super Mario World (SNES).

[10] Nitsche, M. et al. Designing Procedural Game Spaces: A Case Study. *Proc. FuturePlay 2006*. London, ON. October 10-12, 2006.

[11] Persson, M. Infinite Mario Bros! (Online Game). Last Accessed: December 11, 2008. http://www.mojang.com/notch/mario/

[12] Sega. 1991. Sonic the Hedgehog (Genesis).

[13] Smith, A. et al. Tableau Machine: A Creative Alien Presence. *Proc. 2008 AAAI Spring Symposium on Creative Intelligent Systems*, pp. 82-89. Menlo Park, CA, Mar 2008.

[14] Smith, G., Cha, M., and Whitehead, J. A Framework for Analysis of 2D Platformer Levels. *Proc. 2008 ACM SIGGRAPH Sandbox Symposium*, Los Angeles, CA. August 9-10, 2008.

[15] Togelius, J., De Nardi, R., Lucas, S.M. Towards Automatic Personalised Content Creation for Racing Games. *Proc. IEEE Symp. on Comp. Intelligence and Games 2007*, April 1-5, 2007.

[16] Toy, M., Wichman, G., and Arnold, K. Rogue (PC Game).

[17] Wonka, P. et al. Instant Architecture. *ACM Transactions on Graphics*, vol. 22, no. 4, pp 669-677. July 2003.

[18] Yu, Derek. 2009. Spelunky (PC Game).