Discovering Unique Game Variants

Aaron Isaksen
aisaksen@nyu.edu

Dan Gopstein
dgopstein@nyu.edu

Julian Togelius
julian.togelius@nyu.edu

Andy Nealen
nealen@nyu.edu

NYU Game Innovation Lab

Abstract

We present a method for computationally discovering diverse playable game variants, using a fine-tuned exploration of game space to explore a game's design. Using a parameterized implementation of the popular mobile game Flappy Bird, we vary its parameters to create unique and interesting game variants. An evolutionary algorithm is used to find game variants in the playable space which are as far apart from each other as possible, helping find the most unique versions; we also use a clustering algorithm to find representative variants. Manual playtesting confirms that the discovered game variants are playable and significantly different to the original game in challenge, game feel, and theme.

Introduction

The process of *exploratory computational creativity* (Boden 2004; Wiggins 2006) includes searching the conceptual space of possible creations to find viable and interesting games. As we are interested in computationally creating games, we will call this *game space*.

A game is composed of many parts, including visuals, audio, narrative, systems of rules, level architecture, gameplay, and interactions between those facets (Liapis, Yannakakis, and Togelius 2014). In this paper, we will focus on the parameters which tune gameplay and systems. While rules encode how the system works, parameters are tunable variables which modify values in the system. For example, in Flappy Bird (Nguyen 2013), shown in Figure 1, the bird speed and pipe width are parameters, while the method of movement (e.g. if the player flies or walks) is a rule. We explore a subset of game space defined by only changing game parameters, but not changing or evolving game rules or mechanics (Nelson et al. 2015; Browne and Maire 2010; Cook and Colton 2011).

By changing parameters alone, we can find surprising and interesting game variants, such as our computationally discovered Frisbee Bird (Isaksen, Gopstein, and Nealen 2015), shown in Figure 2. This variant was unexpectedly created when searching Flappy Bird game space for variants of a specific difficulty. In this variant, the player is wide and flat, and smoothly sails through the air like a flying disc. We had not expected that a game of this type was in the same game space

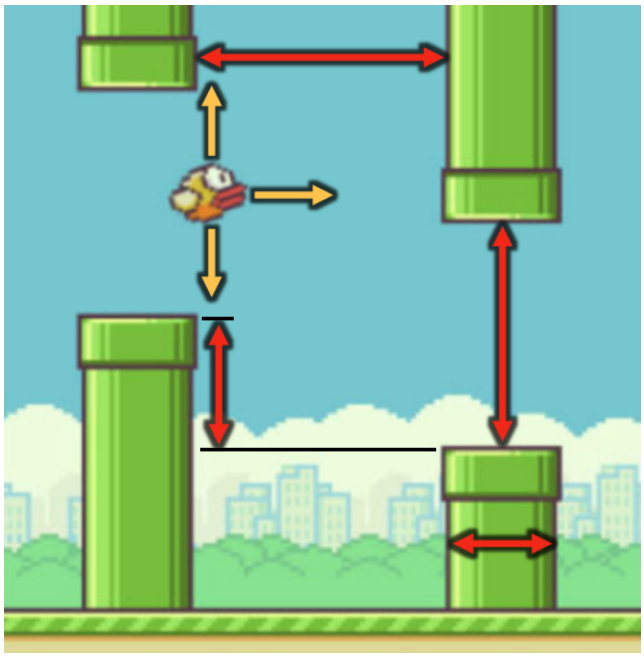


Figure 1: In Flappy Bird, the player must avoid crashing into the pipes. The game is defined by parameters shown here by arrows. By varying these parameters, we can explore different variants in game space to find the most unique variants, or those most inspiring to other game designers.

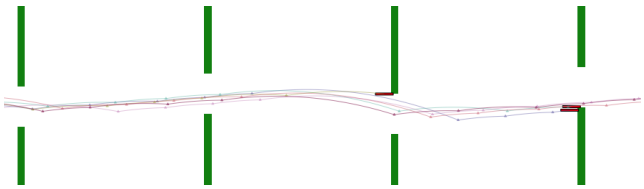


Figure 2: Frisbee Bird was computationally discovered while searching for Flappy Bird variants. Its unique location in game space leads to a different game feel and appearance.

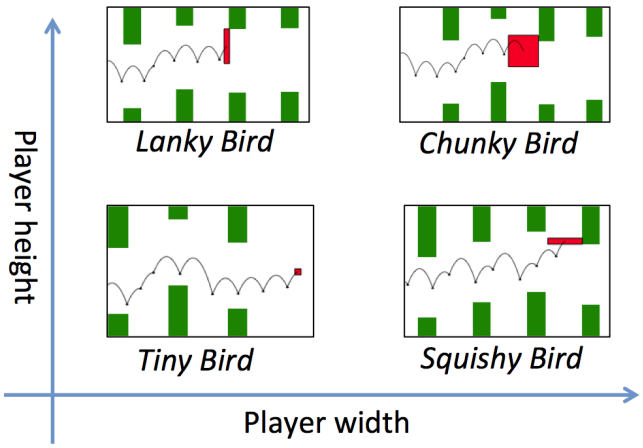


Figure 3: A two-dimensional game space of Flappy Bird variants with different player width and player height. The only difference is the size and aspect ratio of the player, yet we can already distinguish these as unique games.

as Flappy Bird, which encouraged us to develop an algorithm to discover more unique and surprising game variants.

We have several reasons for a fine-grained exploration of game parameter space to create and evaluate new game variants. From the perspective of game design studies, it offers us a way of understanding games more deeply, in particular what kind of variants of a game could be encoded with the same ruleset. From the perspective of work on automatic game design, it helps us understand how to automate the iterative process of tuning a game, required for improving game feel (Swink 2009). As game designers, we are excited about the potential for a computational design assistant which can help us quickly evaluate the effects of various design decisions, and to inspire us to try new, creative ideas. However, to trust these computational results, we must ensure that they are human-like in their creation and evaluation.

We will examine variants of Flappy Bird, one of the most downloaded mobile games of all time. In Flappy Bird, the player must avoid crashing into the pipes by tapping the screen to flap, while gravity is constantly pulling the player downwards. Isaksen, et al. (2015) previously analyzed each game parameter of Flappy Bird, including player size, pipe gap, pipe randomness, pipe width, gravity, jump velocity, bird speed, and pipe separation, showing how this game space can be explored by algorithms that use survival analysis and Monte Carlo simulation to find games of varying difficulty.

In Figure 3, we show different variants created by exploring only the two dimensional game space defined by player width and player height. Each of these games has a different difficulty, and we would expect Tiny Bird to be the easiest version because the bird has more room to maneuver between the pipes. There are upper bounds to the parameters in this example game space: no player can score a point if the bird is so tall it can't fit through the pipe gap. It's notable that by only changing the player size parameters, we can already ascribe a different theme and name to each of them.

Evaluating Games Procedurally

We now present our method for finding the most unique variants in game space. The first step is to separate the games which are playable from those which are unplayable: in an unplayable game, every player will die at or before a specific event in the game, no matter how good the players are. These unplayable games can have artistic merit, for example those intentionally unbeatable games which are trying to convey a message about futility of a political situation via procedural rhetoric (Bogost 2007). However in this paper, we are generally interested in scored action games in which an excellent player, with enough practice and patience, will rise to the top of the leaderboard. This is clearly not the entire space of interesting games, but it is a large and popular one, and provides us with a simpler domain to explore.

It is important to consider the target player when evaluating game space to find playable and unplayable games. The skill, experience, and learning ability of the player has a major impact on their performance. Our algorithm uses a tunable parameter for skill, which we keep fixed throughout the experiments, but for consistency and simplicity, ignores past experience and learning effects. We also need to constrain the time that it takes to play an average game. Although some parameter settings create playable games, they would take too long to play and are therefore not interesting to the player (e.g. a game which takes a year to score a single point).

For every point in game space we wish to test, we generate a unique game and simulate playing it, having an AI repeatedly play thousands of times to estimate the difficulty d and the average time it takes to play t. In our previous work we explain how to simulate different player skills by introducing randomness into the reaction time of the AI. In summary, we play the game variants using an AI player model that makes errors based on emulating motor skill: the AI searches ahead to find the time τ it wishes to flap, and then adds a normal distribution with mean 0 and standard deviation $\sigma = 30$ms which represents motor skill precision. By decreasing or increasing σ, we can simulate better or worse skill. We measure difficulty by analyzing the resulting score distribution after simulating each variant 20,000 times.

Bounding the Search Space

The search domain can be arbitrarily large, but as we are focused on games that are playable for humans, we begin by creating reasonable, but artificial, bounds on the parameter space to explore. Although unbounded creativity could lead to new games, the extra effort required to search the larger space could mean that many computational resources are wasted on searching fruitlessly. Thus, we aim to bound our search to maximize the chance that the computer will find playable games, while minimizing the chance that the computer will miss playable games within the domain.

Many of our parameters have natural upper and lower bounds: for example the gap between the top and bottom pipe must be positive because a negative pipe gap means the pipes would overlap, and it must be less than the world height because a pipe gap extending beyond the game bounds is also nonsensical. Some only have a natural lower bound:

horizontal pipe separation must be ≥ 0 so that the pipes do not overlap, but there is no upper bound how far apart the pipes can be. This technically does not have an upper bound, as one could make them millions of pixels apart so that it takes hours to travel between pipes. However, by ensuring an average or maximum play time for a playable game, we can eliminate these extreme variants. Horizontal speed has an effective upper limit, determined by both the geometry of the level and the skill of the target player, but this can only be discovered through simulation or play testing. Finally, some parameters have no required bounds: for example gravity could be positive or negative and a sensible game would arise, but jump velocity must be in the opposite direction from gravity for the game to be playable.

We begin by seeding reasonable bounds and then generating samples using a stratified sampling scheme that covers the search space without clumping that may occur with uniform, random sampling. With a sufficiently large number of samples, a uniform sampling can be used instead to avoid the extra computation of stratified sampling.

Using the methods described in the previous section, we can simulate each point and calculate its difficulty and average play time. We mark as playable all the games that have a difficulty where between 1% and 60% of players are expected to crash on the first pipe, and the average human play time is between 1 sec and 30 sec. We eliminate games with a difficulty $< 1\%$ as they are too easy and would likely be boring for a player. The play time is calculated as if the game were played by a human, not as the time it takes for the computer to simulate a game ($<$ 1ms).

We now calculate the hypervolume bounding box that tightly contains the playable games. For each dimension, we compare the effective upper and lower bounds that lead to playable games to the original search range. If the effective bounds are near the guessed bounds, we could, if desired, expand the search area and try again because it is possible we have missed sampling some playable games.

Instead of setting bounds, we could create a probability density to estimate the likelihood of discovering a playable game within a parameter space. This is similar to how genetic algorithms work to improve their search domain over successive generations. Novelty Search can also be used to guess the next best location to search (Liapis, Yannakakis, and Togelius 2013).

Playing Unexpected Games

As the difficulty and playability of the game is determined by the AI, it will only find games that it is equipped to play. However, there may be games interesting to players of human intelligence which are ignored by the AI's deficiencies. The method of search can influence which points in conceptual space are discovered (Wiggins 2006).

When designing the AI to play the original game, we made assumptions that ended up being violated when exploring new parts of the game space. Specifically, we assumed that gravity would always point down, and jump would always point up, so the AI heuristic focused on jumping when going *below* a certain target height. Thus, the original AI found inverted gravity games impossible to play, eliminating entire classes of playable games which we might find interesting. By changing the AI heuristics to be smart enough to handle both types of gravity, we were then able to find games like Divey Duck, shown in Figure 4.

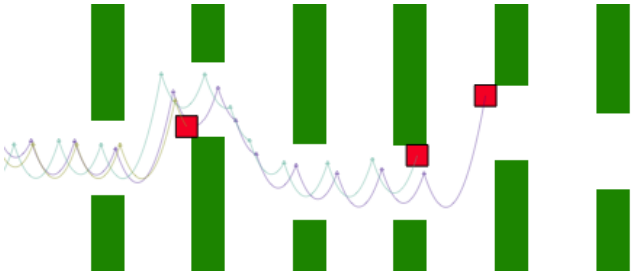


Figure 4: Divey Duck, a variant with negative gravity and negative jump, simulates an underwater duck that naturally floats. However, the original AI couldn't handle negative gravity. Without a change to support both types of gravity, this whole class of games would be labeled unplayable.

There are two insights to be gained here: (1) the AI used to simulate play determines the types of games that are discovered, and (2) using more sophisticated AIs can be important even when a simpler AI works fine for the original game. Regarding better AIs, an A* or Monte Carlo Tree Search algorithm would likely be sufficient for solving Flappy Bird like games. Alternatively, an AI that can learn over repeated plays of a video game could learn how to effectively play inverted gravity (Mnih et al. 2015).

Finding Unique Games

Now that we have a point cloud of thousands of playable games variants, we would like to find a small number of unique games, so we do not have to play all of them for inspiration and further tuning. There are two approaches we tried to narrow down this number.

Finding Representative Games with Clustering

Our first approach is to perform a k-clustering on the playable game point cloud, which will find groups of similar games, where each medoid is the representative of one cluster. We call these *representative games* because they are points that represent an entire cluster of games.

First, one must determine what value of k to use. This could be determined a priori, and then defines how many representative games will be returned. Alternately, one can use a method that determines the ideal k by trying repeated clusterings and stopping when finding the k that gives the optimum average silhouette width (Rousseeuw 1987). In practice, we use the `pamk` function in `R` (Hennig 2014; R Core Team 2015) to find an optimal value of k. One also typically sets a maximum value of k, which is especially important for clustering high dimensional spaces.

Second, since our parameters have different units, it is important to normalize the parameter space so that different variables can be compared. For example, if gravity ranges between 1 and 4000, while pipe gap varies between 10 and

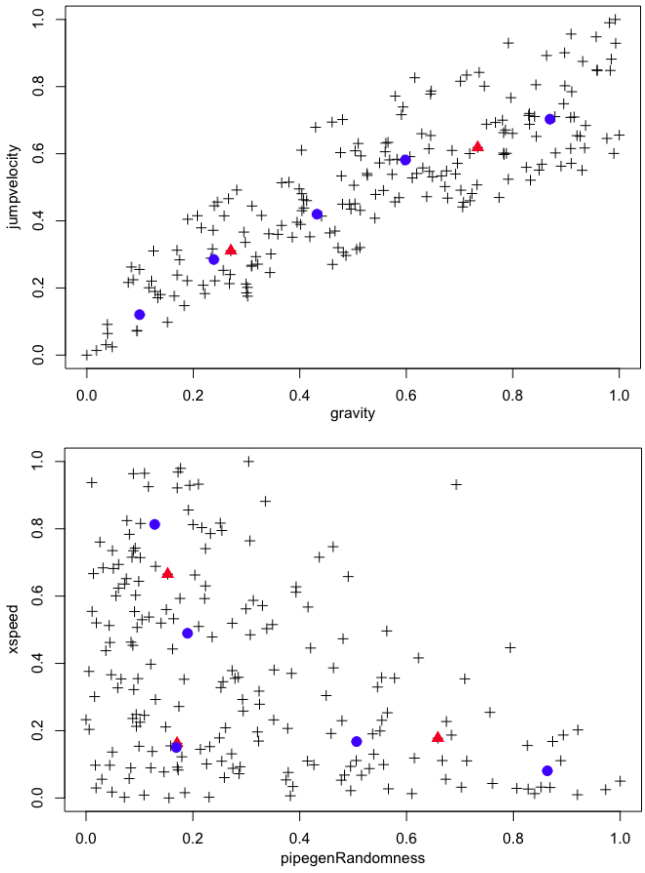


Figure 5: Searching 2D game space to find k representative games using clustering. Blue dots are medoids calculated for $k = 5$. Red triangles are medoids calculated when using a dynamic clustering which finds an optimal value of k.

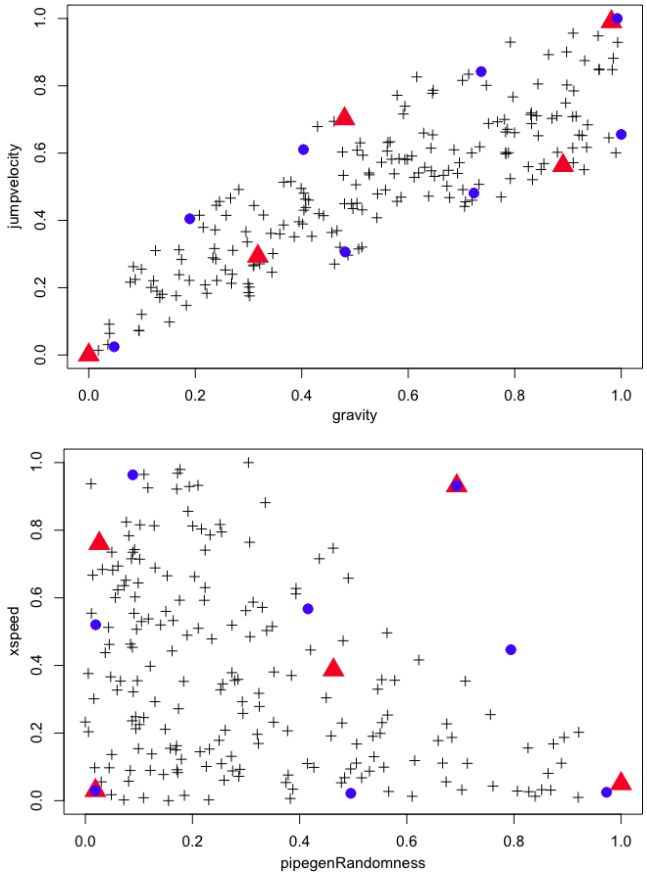


Figure 6: Using genetic optimization, we find games that are far apart from each other in 2D game space. Blue dots indicate the set for $k = 8$ and red triangles are for $k = 5$.

30, the clustering on un-normalized space would incorrectly suggest that changes in gravity are far more important than changes in pipe gap. By normalizing each parameter between 0 and 1, we can compare different parameters using a Euclidean distance metric. We normalize after finding the playable games, such that for each parameter 0 is the minimum playable value and 1 is the maximum playable value.

As we can see in Figure 5, the clustering method avoids games on the frontiers of playable games, because the medoids are at the center of each cluster. Although the clustering works in n-dimensional space, for visualization we have taken 204 playable games and projected them onto two different 2D planes, one for gravity vs jump velocity, and one for pipe location randomness vs player speed. The blue dots indicate cluster medoids when choosing a fixed value of $k = 5$, while the red triangles indicate medoids selected when searching for an optimal k.

Finding the Most Unique Games

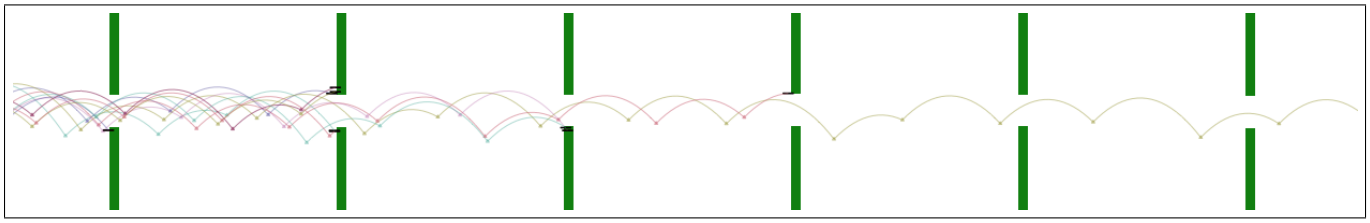
Clustering finds interior points, but we also want to find games which are further apart, showing us what is possible on the frontiers of game space. The games on the frontier are not guaranteed to be the most unique, but they push the parameters to their extremes. We experimented with looking at games on the convex hull, but preferred methods that can find games in both the interior and exterior of the cloud.

To find games as far apart from each other as possible, we find a subset of k games that maximizes the minimum Euclidean distance (in normalized parameter space) between any pair of points in the set. We use a genetic search to evolve the k elements to be included in the set.

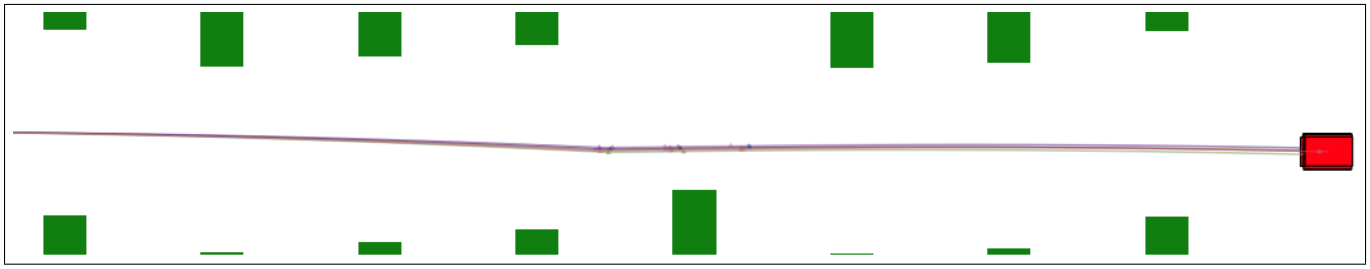
This method does a much better job at finding games on the exterior, since we expect to select points at the fringe of the point cloud (which is furthest from the center of the cloud). By running for more generations, the examples will get closer to optimally unique.

Results and Discussion

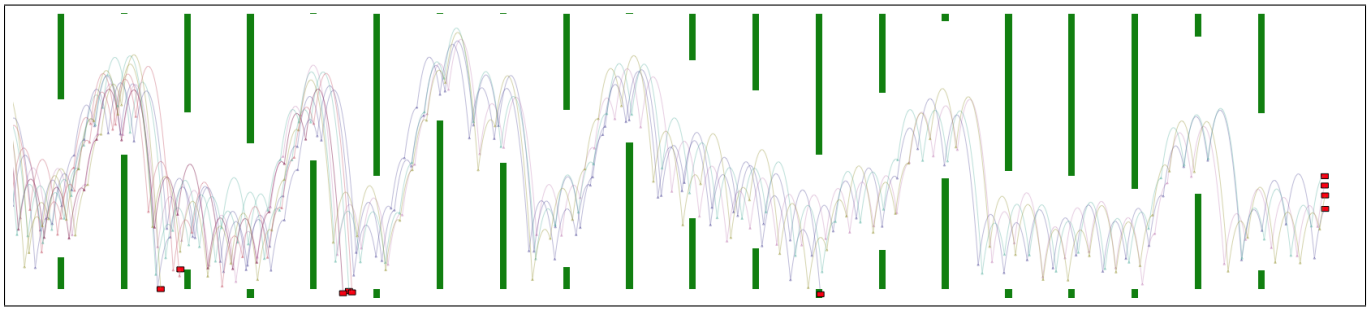
We now present several novel Flappy Bird variants discovered by our methods. Varying all nine game parameters, we asked both algorithms to create 4 unique games. The ”Most Unique” method generated four highly different games. We have named them Needle Gnat, Lazy Blimp, Droppy Brick, and Pogo Pigeon and show examples of them in Figure 7. These appear to us to approximate the creativity a human de-



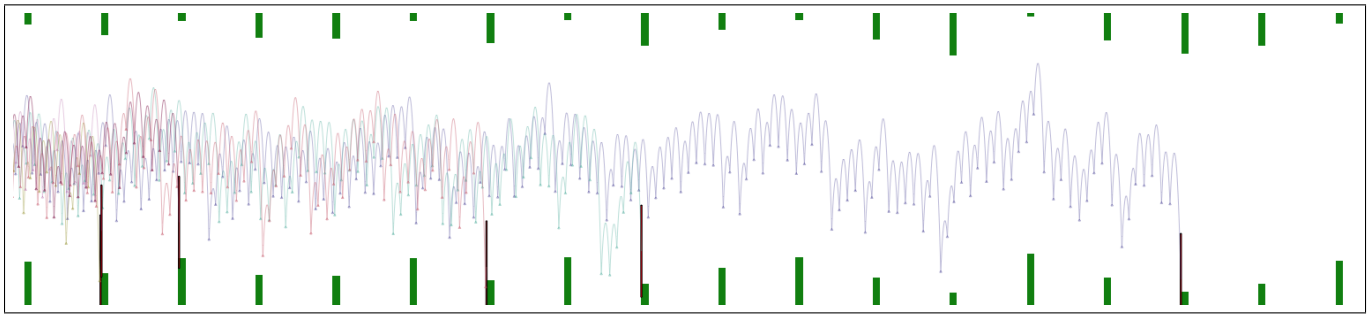
Needle Gnat



Lazy Blimp



Droppy Brick



Pogo Pigeon

Figure 7: The four game variants discovered using the Most Unique evolution method with $k = 4$. This method searches for the k games which maximizes the minimum distance between any two points in the set. The games are generated by the algorithm; the names are provided by the authors. (a) Needle Gnat: tiny player trying to thread a tight horizontal space. (b) Lazy Blimp: slow moving blimp-like player with minimal gravity and jump. (c) Droppy Brick: frequent rise and fall with high gravity. (d) Pogo Pigeon: very tall, thin bird that frequently hops to avoid crashing into the ground.

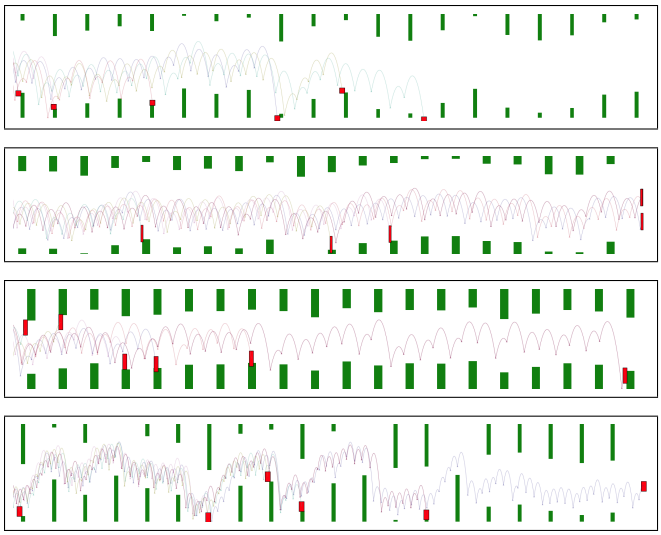


Figure 8: Representative games found by clustering, which as expected do not appear to exhibit the variety of the Most Unique games shown in Figure 7.

signer might have used if tasked with designing four unique variants. The Representative Clustering method generated four versions which do appear closer to each other as expected, as shown in Figure 8. Due to their similarity we have not felt the need to give them names, which is a notable metric that could use further investigation.

We have presented two methods for discovering unique game variants. This space is based on game design parameters, using Euclidean distance for measuring distance between two games. However, game parameters do not necessarily have linear responses, so measuring in Euclidean space can be misleading. That is, games with pipe gap of 10 and 15 (50% difference) are more different for a player's experience than for a pair of games with pipe gap 30 and 45, with a larger absolute change and the same relative change.

A perceptual metric is desirable, so that we can compare two games from the player's perspective, and then find the k most unique games in perceptual space, instead of game parameter space. This perceptual metric also tells us when games are perceptually similar, greatly reducing the search resolution required. Given the complementarity of some of these parameters–for example, we could scale all of the distance metrics by the same amount–it also seems possible that there could be games which are ostensibly far apart in parameter space, but actually appear very similar in perceptual space. This would probably become more likely the larger the parameter space. One could also develop metrics for game similarity based on simulation, where games eliciting similar playing behavior (or learned behavior) from the AI are judged to be similar.

Our methods presented here are focused on uniqueness, which is one aspect that makes games interesting to players and designers, but "interesting" has a much broader meaning and significantly more complex to measure. Future work on developing metrics that can quantitatively measure expected human interest are highly desirable for creative exploration and will require a deeper understanding of player behavior.

We look forward to more creative output both from algorithms generating new game variants, but even more so from teams of humans and computers working together to create better games and inspiring future game designers.

Acknowledgments

Thank you to Ron Carmel, Frank Lantz, and Rob Meyer for suggestions of unplayable games, to the NYU Game Innovation Lab, and to Dong Nguyen for creating the original Flappy Bird and for inspiring our research.

References

Boden, M. A. 2004. *The creative mind: Myths and mechanisms*. Psychology Press.

Bogost, I. 2007. *Persuasive games: The expressive power of videogames*. Mit Press.

Browne, C., and Maire, F. 2010. Evolutionary game design. *Computational Intelligence and AI in Games, IEEE Transactions on* 2(1):1–16.

Cook, M., and Colton, S. 2011. Multi-faceted evolution of simple arcade games. In *CIG*, 289–296.

Hennig, C. 2014. *fpc: Flexible procedures for clustering*. R package version 2.1-9.

Isaksen, A.; Gopstein, D.; and Nealen, A. 2015. Exploring game space using survival analysis. In *Foundations of Digital Games*.

Liapis, A.; Yannakakis, G. N.; and Togelius, J. 2013. Enhancements to constrained novelty search: Two-population novelty search for generating game content. In *Genetic and evolutionary computation*, 343–350. ACM.

Liapis, A.; Yannakakis, G. N.; and Togelius, J. 2014. Computational game creativity. In *Proceedings of the Fifth International Conference on Computational Creativity*, 285–292.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518(7540):529–533.

Nelson, M. J.; Togelius, J.; Browne, C. B.; and Cook, M. 2015. Rules and mechanics. In *Procedural Content Generation in Games*. Springer.

Nguyen, D. 2013. Flappy bird. Apple App Store.

R Core Team. 2015. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Rousseeuw, P. J. 1987. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics* 20:53–65.

Swink, S. 2009. *Game Feel*. Morgan Kaufmann.

Wiggins, G. A. 2006. A preliminary framework for description, analysis and comparison of creative systems. *Knowledge-Based Systems* 19(7):449–458.