Exploring Game Space Using Survival Analysis

Aaron Isaksen aisaksen@nyu.edu
Dan Gopstein dgopstein@nyu.edu
Andy Nealen nealen@nyu.edu
NYU Game Innovation Lab[1]

ABSTRACT

Game designers adjust game parameters to create an optimal experience for players. We call this high-dimensional set of unique game variants *game space*. To help designers explore game space and better understand the relationship between design and player experience, we present a method to find games of varying difficulty. Focusing on a parameterized version of Flappy Bird, a popular minimal score-based action game, we predict each variant's difficulty using automatic play testing, Monte-Carlo simulation, a player model based on human motor skills (precision, reaction time, and actions per second), and exponential survival analysis of score histograms. Our techniques include searching for a specific difficulty, game space visualization, and computational creativity to find interesting variants. We validate our player model with a user study, showing it is effective at predicting game difficulty.

Keywords

player modeling, Monte-Carlo simulation, game design, automated play testing, dynamic difficulty, game space, gameplay metrics, survival analysis, computational creativity, motor skills, Flappy Bird

1. INTRODUCTION

As game designers, we use adjustable game parameters to tune a game to achieve a desirable player experience. Each unique parameter setting creates a new game variant. We refer to this high-dimensional space of game variants as *game space*. A point in game space is a specific vector of game parameters; these settings directly affect the player, enemies, or the level generation [39]. Exploring game space to find specific settings for an optimal experience is a considerable challenge, and we aim to better understand the relationship between game parameters and player experience.

In this paper, we examine how game parameters, without changing the game rules, can affect a game's difficulty. For example, we would expect a variant with fast enemies to be more difficult than one with slow enemies, even if both versions are otherwise identical and have the same rules. Automatically creating new rules [3, 5, 11, 31, 38] is a related problem, but parameters alone have a significant impact on game feel [34]: getting Mario's jump to feel right is more about adjusting parameters than coding accurate physics.

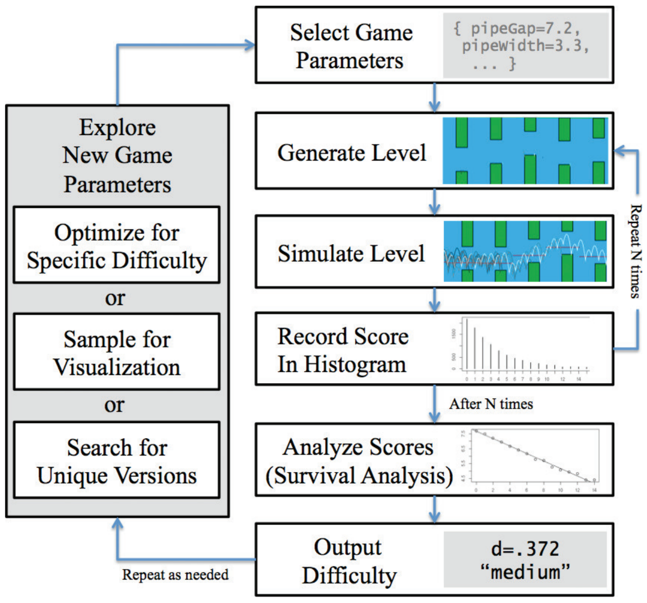

Figure 1: We explore game space by simulating game variants and estimating their difficulty using survival analysis.

The set of all game variants for a specific game is high-dimensional and often impossible to search exhaustively – imagine adjusting hundreds of independent control knobs to search for the perfect game. We reduce the search space by focusing on game variants that only change parameters, not the larger class of variants that include changes to game rules. While tuning games, designers rely on intuition, experience, and user feedback to iteratively search game space. The designer must estimate player skill, set game parameters, play test, evaluate player experience using gameplay metrics, revise parameters, and iterate until the game reaches an appropriate level of difficulty [7, 9]. When a designer becomes an expert at their own game, they can lose perspective on how their game is experienced by new players. It can also be difficult for designers to break out of local optima and explore creative new regions of game space. Automated play testing [20, 47] and visualization [42] help with this process, guiding designers in their exploration to create games best suited to individual skill levels and play styles [10].

We present a methodology for tuning game parameters that can be used by everyday game designers without requiring human play

[1] `game.engineering.nyu.edu/exploring-game-space`

testers. Our general approach is shown in Figure 1. To explore a new game variant, we select a parameter vector from a valid range of values. Using this vector, we generate a level and simulate playing it using an AI that models human imprecision. We repeat the Generate and Simulate steps until we have a reliable histogram of scores for the game variant. We then analyze the histogram using exponential survival analysis [14, 27] to find the decay rate of the exponential distribution. Faster decay predicts a harder game as higher scores are increasingly less likely than lower scores.

For our research, we decided to work with a game that is popular and has relatively few adjustable game parameters. We chose Flappy Bird [21] because it is a commercial and critical success, spawning hundreds of similar games, and the rules are simple to implement and understand. In Flappy Bird, a player must fly a constantly moving bird without crashing into a series of pipes placed along the top and bottom of the screen (see Figure 2). Each time the player taps the screen, the bird flaps its wings, moving upward in an arc while gravity constantly pulls downward. Each time the bird passes through a pipe gap without crashing, the player scores a point. Part of the appeal for Flappy Bird is the comically high difficulty level, especially when typical casual games are easy and forgiving. Flappy Bird could have been much easier with a few small adjustments, such as increasing the gap between pipes or decreasing the width of the pipes, but these would have led to different, potentially less rewarding play experiences.

Accurately simulating game play requires an understanding of how players react to game events: this is the process of player modeling [30]. Our player model assumes that much of the difficulty in simple action games is due to human motor skill, specifically precision and reaction time [15]. Since we use a static, objective, simulation-based player experience model [46], we do not need to analyze prerecorded human play sessions of the specific game to train our player model, and do not rely on live players to estimate the current level of challenge or fun [8, 12, 16, 22, 36, 47].

Our goal is to make the simulator play like a human (novice, average, or skilled), not play with superhuman ability (as might be required for some parameter settings of Cloudberry Kingdom, a game that exposes its adjustable game parameters to players when generating platformer levels [24]). As long as the model properly predicts human perception of difficulty, it fits our purposes. In minimal, well-balanced, and compelling action games like Flappy Bird or Canabalt [28], the player takes a relatively obvious path, but executing that simple path is challenging [19]. In this paper we examine a game with simple path planning and without enemies, so we can focus on the player's ability to control the game – without analytical evaluation of possible player movement [4], multi-factor analysis of player or enemy strategies [10], dynamic scripting of opponents [32], building machine learning estimators [36, 29, 45], or evaluating design via heuristics [6].

2. GAME SPACE

Action games are defined by rules and parameters, designed to produce a specific player experience through challenges. Predicting the qualitative experience is hard, but we can examine the distribution of final scores to predict a quantitative difficulty of the game. Throughout this paper, we refer to this measured difficulty as d.

The rules are implemented in code and define concepts like "if the bird collides with the pipe, the game ends." Because we keep the rules fixed, we can represent our game space as a high-dimensional space over the set of parameters used to tune the game.

We have chosen to use the following parameters for our implementation of Flappy Bird (see Figure 2). The original Flappy Bird, and all our variants, have a constant value for each parameter during a play session since the game does not change as the player progresses. In general, game parameters can change as the player gets further into a level (for example, in Canabalt the game speeds up as the player progresses) but we did not explore variants with dynamic parameter values as they would require a change in the rules that define Flappy Bird.

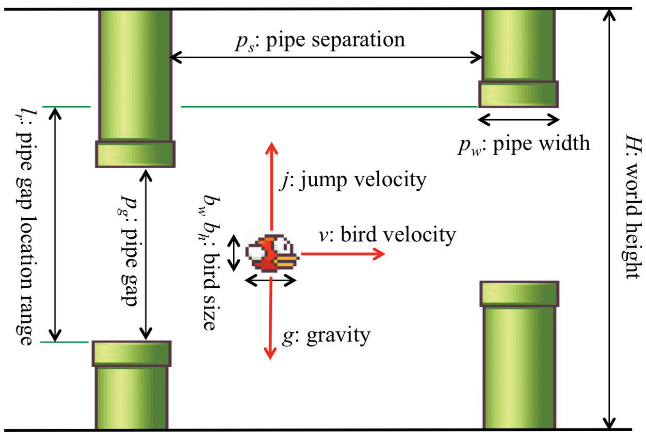

Figure 2: In Flappy Bird, the player must navigate the bird through a series of pipes without crashing. We modify the labeled parameters to generate unique game variants.

Pipe Separation p_s – More distance between pipes is easier to play, giving more time to react to changing gap locations.
Pipe Gap p_g – The distance between the upper pipe and the lower pipe. Narrower gaps are more difficult as the bird has less room to maneuver, requiring better motor skills.
Pipe Width p_w – Wider pipes increase difficulty as the bird spends more time in the narrow pipe gap.
Pipe Gap Location Range l_r – The pipe gap locations are uniformly randomly distributed in a range somewhere between the ceiling and the floor. Larger ranges are harder because there is more distance to travel between a high gap and a low gap.
Gravitational Constant g – Acceleration of the bird in the y direction, subtracted from the bird's y velocity each frame. Higher gravity causes the bird to drop faster, lowering the margin of error.
Jump Velocity j – When the bird flaps, its vertical velocity is set to j, making it jump upward. Higher velocity makes higher jumps.
Bird Velocity v – Speed at which the bird travels to the right (alternately, the speed at which pipes travel to the left).
World Height H – Distance between ceiling and floor. In Flappy Bird, this is defined by the display resolution.
Bird Width and Height b_w, b_h – Size of the bird's hit box. The wider and taller the bird, the harder it will be to jump through gaps.

By varying these parameters within sensible ranges, we can generate all variants of Flappy Bird that use the same set of rules. Many of these parameters have constraints; for example, they all must be positive, and Bird Height b_h can not be larger than Pipe Gap p_g or the bird can't fit through the gap.

3. PLAYER MODEL

We begin with a model of a player with perfect motor skills – a perfect player with instantaneous reaction who would never lose at the original Flappy Bird. Given a version of Flappy Bird defined by its game parameters (a single point in game space as defined in Section 2), we create an AI that finds a path through the pipes with-

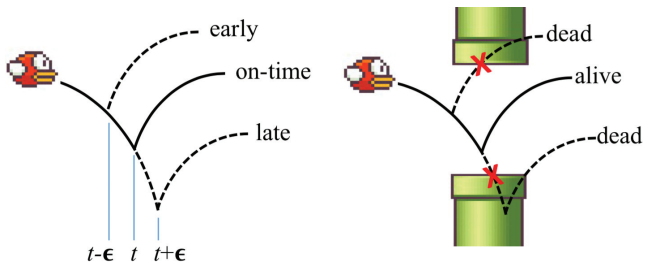

Figure 3: Modeling precision by randomly adjusting the time the simulated player jumps. Moving the jump earlier or later can cause the bird to crash into the lower pipe or upper pipe.

out crashing. Instead of using an A* planner that finds the shortest path, we chose to use a simpler AI which performs well but is easier to implement and faster to run. Each time the bird drops below the target path, the AI immediately executes a flap (which sets vertical bird velocity v_y instantly to jump velocity j). Whatever AI is used, it should play with very good performance on solvable levels, and should mainly only fail on impossible levels, such as a level with a tiny pipe gap where the bird cannot fit through.

We then extend the AI to perform less well by modeling the main components of human motor skill which impact difficulty in these types of action games: precision, reaction time, and actions per second. Adjusting these values lets us model different player types, since novices react slower and are less precise than experts.

3.1 Player Precision

When a player plans to press a button at an exact time, they execute this action with some imprecision. We model this error as a normal distribution with standard deviation proportional to a player's imprecision. Imprecision in an inherent trait, but is also related to the time a subject has to react to an event, called the speed-accuracy tradeoff: the less time they have to react, the less accurately they will respond [43]. For simplification, our player model assumes precision is an independent variable and not dependent on bird speed. In our user study (Section 7), we measured precision as an error with standard deviation ranging between $\sigma_p = 35.9$ ms and $\sigma_p = 61.1$ms, and use this range for game space exploration.

We model imperfect precision in our AI by calculating an ideal time t to flap, then adding to t a small perturbation ϵ, drawn from a random normal distribution with 0 mean and standard deviation σ_p, as shown in Figure 3. By increasing the standard deviation σ_p, the AI plays less well and makes more errors, leading to a higher difficulty estimate (see Section 6.1.1 and Figure 6 for the impact of varying precision). Reducing σ_p to 0 ms lets us test if a level is solvable by the AI without taking into account human error.

3.2 Reaction Time

When a player sees a new pipe show up on the screen, it takes some time to react. The speed of the player's reaction is influenced by factors inherent to the system [13], as well as factors affecting the player themselves [35]. To identify an average reaction time to expect from people playing our games, we measured a mean delay $\tau = 288$ ms in our user study (Section 7).

We constrain the AI to react only after it has observed a new pipe for τ ms of simulated time. We found in our Flappy Bird experiments the delay has minor impact on estimating difficulty, and mostly matters for bird speed settings that are exceedingly fast.

3.3 Actions Per Second

Humans can only perform a limited number of accurate button presses per second. In our user study, we measured an average 7.7 actions per second. We also limit our AI to this same number of actions per second. We simplify our model by keeping this constant, although a more complex model would account for fatigue, since players can't keep actions per second constant for long periods.

4. ESTIMATING DIFFICULTY

Given our game space and player model, we now present our methodology for estimating difficulty. We explore points in game space by performing randomized simulations and examining the distribution of scores. Our goal is to describe the perceived difficulty of a game by a single value $d \in [0, 1]$, related to the difficulty estimated by an AI using a player model with predefined settings for reaction time, precision, and actions per second.

Impossible games ($d = 1$) are those games which can't be played successfully by any player. This can happen, for example, if the jump velocity j is much weaker than gravity g, or if the pipe gap p_g is so small that the bird can't fit through the gap. Impossible games that appear playable can also occur with some parameter combinations: for example, with high bird velocity b_v, low gravity g, and high pipe gap location range l_r, it is often impossible to drop in time from a high gap to a low gap. We eliminate these games by first verifying that a perfect player with precision standard deviation $\sigma_d = 0$ms can reach a goal score with high frequency.

Playable games ($d < 1$) are those games which can be played successfully by some players. In our experiments, we found empirical evidence that the set of playable Flappy Bird games is a single connected space. In general, this is not a requirement for games and may be disjoint for more complicated games.

Trivial games ($d = 0$) exist where all players will be able to achieve the goal score, assuming a reasonable amount of effort. For example, this can arise if the pipe gap p_g is large. The player still needs to pay attention and actively tap the screen, but the game is trivial as long as they are actively participating.

Equal Difficulty Games are points in the game space where the difficulty is approximately equal. These points can be viewed as a (possibly disjoint) subspace: all games in the subspace for a fixed value of d will have the same difficulty, and points near it will have similar difficulties. By finding distant points within such a subspace, we explore a variety of games that are of equal difficulty.

The process to calculate d is composed of three steps: 1) *Generate* - build a new game variant based on the given parameters, 2) *Simulate* - use an AI to play the game with human-like behavior, and 3) *Analyze* - examine the resulting score histogram using exponential survival analysis to estimate the difficulty d of the simulated game. Generate and Simulate steps are repeated until we have a stable score distribution for measuring in the Analyze step.

4.1 Generate

Each simulation begins by taking a parameter vector $\mathbf{p}$ and generating a new game variant. This involves placing the bird and pipes in their starting positions and randomly distributing the pipe gaps. In Figure 4, we show two different generated game variants.

Because the levels are generated using a random process, it is important to generate a new level each time the AI runs, even though the parameters $\mathbf{p}$ do not change. Otherwise, if the same random layout of pipe gaps is used repeatedly, artifacts can arise in the score distribution caused by a particular placement of the gaps. For example, the gap locations can randomly come out approximately equal for a section of the level, making that section easier. These artifacts are averaged out by generating a new level each time.

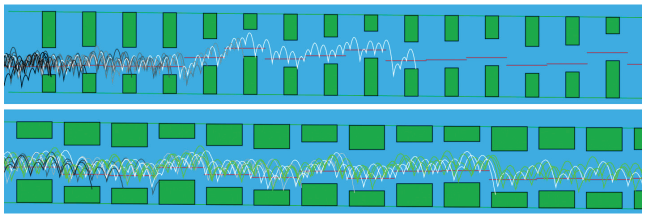

Figure 4: Two different levels created by the Generate step given different parameter sets. More simulations complete the second version, so it has an easier estimated difficulty. The red lines indicate the target location for the AI to flap.

4.2 Simulate

Given the level created in the Generate step, we use a simple heuristic to find a path through the pipes by flapping when the bird reaches a line in the lower half of the gap (these lines are drawn in red in Figure 4). At each frame of the simulation, we predict the next time t in the future when the bird will drop below this ideal flapping location – the ideal player would flap at exactly this time t. By reducing or increasing the standard deviation σ_p of our precision model (Section 3.1), the AI plays more or less well. We quickly check if a variant is impossible by using $\sigma_p = 0$ms on a limited number of simulations, and only continue testing if a majority of the these simulations score highly. It is important to note that our AI does not need to be perfect to detect if a game is possible, as the AI with $\sigma_p = 0$ms performs far better than humans.

To keep the AI from flapping faster than a human could tap, t is limited by the number of actions per second. We also limit the AI lookahead to only use information that has been visible on the screen for at least τ (the time it takes for a player to react to a new event). In our experiments, τ did not have much effect except in extreme situations where humans would perform poorly anyway.

For each simulation, we get a score s, equal to number of pipes that the AI passes before crashing, and we record each score s in a histogram **S**. If the AI reaches a goal score s_{max}, we terminate the run so we do not get stuck simulating easy games where the AI will never crash. Although Flappy Bird can theoretically go on forever, human players tire or get distracted and will eventually make a terminal mistake, but the AI can truly play forever unless we enforce a maximum score. We call simulations with $s < s_{max}$ **lost**, and simulations with $s = s_{max}$ **won**. We discuss how to set s_{max} in Sec. 5.2. The Generate and Simulate steps are run repeatedly until we have enough samples to adequately analyze the histogram. We calculate the proper number of simulations in Sec. 5.1.

4.3 Analyze

After running the Generate and Simulate steps, we examine the distribution of scores, as shown in Figure 5. If a player crashes into pipe x or the ground immediately before it, they will achieve a score of $s = x - 1$ (the first pipe is $x = 1$, giving a score $s = 0$). Using exponential survival analysis, we model the probability that a player crashes into pipe x or the ground before it with an exponential distribution and its cumulative distribution function:

$$P(x) = \lambda e^{-\lambda x} \tag{1}$$

$$P(X \leq x) = \mathrm{CDF}(x) = 1 - e^{-\lambda x} \tag{2}$$

Because the scores from our simulation follow an exponential distribution, we describe the shape of the histogram by the decay rate λ. This one value λ defines the survivability of the game variant. Distributions with higher λ will decay faster and are more dif-

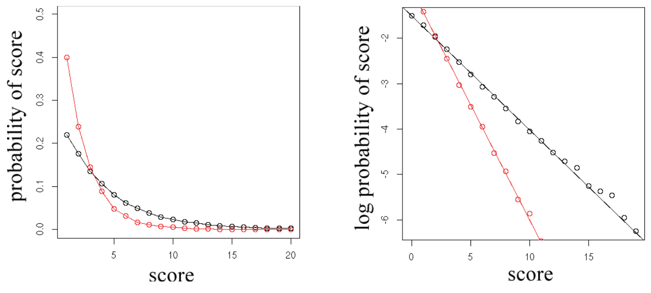

Figure 5: The score distribution is exponential (left). Faster decay is more difficult, so the red line is a harder variant. We take the log of this distribution (right) and weighted least squares fit a line to calculate the decay constant: steeper is more difficult.

ficult: a higher proportion of simulations die on each pipe. Lower values of λ imply the player is more likely to survive each pipe, meaning the game is easier. Since it is harder to reach higher scores than lower scores, $P(x)$ is non-increasing, so $\lambda \geq 0$.

The cumulative distribution function in Eq. 2 is the total probability of crashing into any pipe from 1 to x. Based on λ, we define difficulty d as CDF($x = 1$), the theoretical probability of crashing into the first pipe or the ground before it:

$$d = 1 - e^{-\lambda} \tag{3}$$

With this definition, difficulty d varies between 0 (trivial) and 1 (impossible), and $1 - d$ is the expected fraction of simulations that make it through each pipe.

We employ two methods to find λ and therefore d: via survival analysis of the *lost simulations* (Section 4.3.1) and survival analysis of the *won simulations* (Section 4.3.2).

4.3.1 *Finding d from Lost Simulations*

To find d from the lost simulations where the AI died before reaching the goal score s_{max} and thus achieved a score $s < s_{max}$, we take the natural log of $P(x)$ in Eq. 1 to get:

$$\ln(P(x)) = \ln(\lambda) - \lambda x \tag{4}$$

We substitute $y = \ln(P(x))$, $m = -\lambda$, and $b = \ln(\lambda)$ to see Eq. 4 fits the linear form $y = mx + b$. We then use linear regression to find m and therefore λ. We ignore b because this is only a scale factor to make the probability histogram sum to 1.

Using the scores recorded during the repeated simulations, we know the frequency of crashing into each pipe x, which is the number of simulations that achieved a score of $s = x - 1$. We get probability $P(x)$ by dividing each bucket count by the sum of all buckets. We then fit a line to this log-normal distribution and extract the slope $m = -\lambda$ (see Figure 5). We use the `R MASS` library [25, 41] function `rlm` with weighted least squares fit based on the frequency in the original score distribution, so that we do not give too much weight to the low probability but high valued scores (which can cause overfitting after log transformation).

When fitting, we must not include the histogram bucket for s_{max}, which contains all the scores for every simulation that passes successfully through s_{max} pipes and terminates the simulation for that single iteration. Including this bucket would distort the linearity of the lost simulations histogram.

4.3.2 *Finding d from Won Simulations*

For easy games, most of the simulations will go on forever unless they are terminated at some goal score s_{max}. Easier games will have most scores in the bucket for s_{max}, so we also need a solution for finding d that relies on the won simulations, where $s = s_{max}$. We take a different approach that uses the cumulative distribution function and counts the ratio of won to total simulations.

Let n_s be the number of simulations and $\mathbf{H}[s_{max}]$ be the recorded histogram value for the number of simulations that scored s_{max}. $\mathbf{H}[s_{max}]/n_s$ is the fraction of games that reached this score and did not crash into any pipes. From Eq. 2, $CDF(s_{max} - 1)$ is the fraction of games that crashed into any pipe, and $1 - \text{CDF}(s_{max} - 1)$ is how many games did not crash. In survival analysis, $1 - \text{CDF}(x)$ is called the *survival function* $S(x)$. Putting this together, we have:

$$S(s_{max} - 1) = 1 - \text{CDF}(s_{max} - 1) = \frac{\mathbf{H}[s_{max}]}{n_s} = e^{-\lambda(s_{max}-1)}$$

And solving for λ gives us:

$$\lambda = \frac{-\log(\mathbf{H}[s_{max}]/n_s)}{s_{max} - 1} \tag{5}$$

Since the lost simulation approach works best with a difficult variant, and the won simulations approach works best with an easy variant, we combine the two approaches using a weighted average. We weight the lost simulations result by the fraction of samples with $s < s_{max}$, and the won simulations result by the fraction of samples with $s = s_{max}$.

5. IMPLEMENTATION

5.1 Number of Simulations

When running randomized algorithms, it is essential to have a high number of samples to avoid artifacts: too few samples and the estimate of difficulty will be highly variable, while too many samples requires longer simulation time. We can find a good number of simulations n_s by running an experiment k times for a fixed point in game space, and examining the resulting distribution of d. Each experiment will give a slightly different value of d due to randomness in the stochastic simulation. After k trials, we measure the mean μ_d and standard deviation σ_d. As k increases, μ_d will trend towards the true difficulty value d', giving us a confidence interval that our simulation correctly estimates the difficulty with some given probability. By increasing n_s the standard deviation σ_d will decrease, tightening the confidence interval. Similar to Probably Approximately Correct (PAC) learning bounds [1], we choose a bound ϵ such that with chosen probability δ all estimates of d will fall within ϵ of the true value d'. Using the definition of a CDF of a normal distribution, we find:

$$\sigma_d \leq \frac{\epsilon}{\sqrt{2}\,\text{erf}^{-1}(\delta)} \tag{6}$$

We then increase n_s until the resulting σ_d is under the upper bounds defined by our accuracy thresholds ϵ and δ. Table 1 shows for targeting varying thresholds determines the standard deviation, number of simulations required, and simulation time for a single core on a 2.2 GHz Intel Core i7.

5.2 Maximum Score

We need to terminate our simulations at a maximum score s_{max}, so that easy games do not go on forever. In addition, because our lost simulations data properly fits a log-linear model, we do not need many histogram buckets to get an accurate estimate. Fewer buckets also lead to faster simulations because the simulation ends earlier. However, we still want to ensure enough data points for both the lost and won calculations. Since accuracy of the measurement of d by lost and won simulations is affected by the number of buckets in the histogram, the goal is not necessarily to use a minimal or maximal value of s_{max}. Low values of s_{max} make the lost simulations method less accurate, and high values of s_{max} make the won simulations method less accurate.

ϵ	δ	σ_d	n_s	time
.01	.95	0.005102	~ 5000	60.28 ms
.01	.99	0.003882	~ 9000	101.1 ms
.01	.999	0.003039	~ 12000	127.4 ms
.005	.95	0.002551	~ 20000	214.0 ms
.005	.99	0.001941	~ 30000	415.7 ms
.005	.999	0.001520	~ 50000	584.9 ms

Table 1: Calculating the number of samples n_s required in the Monte-carlo simulation. δ is the probability of being within ϵ of the expected mean of d, implying a standard deviation of σ_d. More samples takes longer but is more accurate.

Using a similar approach to the previous section, we target a specific accuracy and search for a value of s_{max} such that the change in difficulty estimate is bounded by ϵ of the expected mean of d. Using $s_{max} = 20$ gave us a sufficient tradeoff between repeatability ($\sigma_d < .0025$), total simulation time, and balance between accuracy in lost and won simulation estimates of d.

5.3 Anomalies

In some variants, we noticed that the first gap is easier for the AI than other gaps. This can be seen with the first gap having a lower failure rate. Before any pipes arrive on the screen, the most sensible action is for the AI to hover around the middle of the screen, so at worst the bird only needs to travel half of the gap location range $l_r/2$. But for the rest of the game it can be as much as the full gap location range l_r, if a low gap is followed by a high gap. Due to our model relating difficulty to the decay rate of the exponential distribution, we ignore the unique first gap and fit the decay to the remaining gaps for a more accurate estimate of d.

Scoring in Flappy Bird is usually measured at the middle of the pipe – the bird only needs to pass through .5 pipes to get a score of 1, but must pass through 1.5 pipes to get a score of 2. We mitigate this by shifting our scoring location to the end of the gap.

Finally, if the bird starts at the same distance from the first pipe during each run, a beating effect can occur in the score distribution. This happens because some times the player must jump twice within a pipe while other times they only need to jump once. We eliminate this effect by starting the bird at a random x offset on each run of the simulation. This smooths out the beating effect, removing it from the final distribution.

6. EXPLORING GAME SPACE

We require efficient algorithms to explore the high-dimensional space of possible games – an exhaustive search over all parameters will not work. Through intelligent sampling and visualization, we can gain insight about the game (Sec. 6.1), adjust parameters to target a specific difficulty (Sec. 6.2), or generate new versions that vary significantly from the region of game space we are currently exploring (Sec. 6.3).

6.1 Sampling and Visualization Methods

We use the following techniques to sample and visualize difficulty changes as we explore different points in game space. We start this process by picking a point $\mathbf{p}$ in game space, for example by selecting the parameters that define the original Flappy Bird. We extracted the original parameters by examining video capture of the

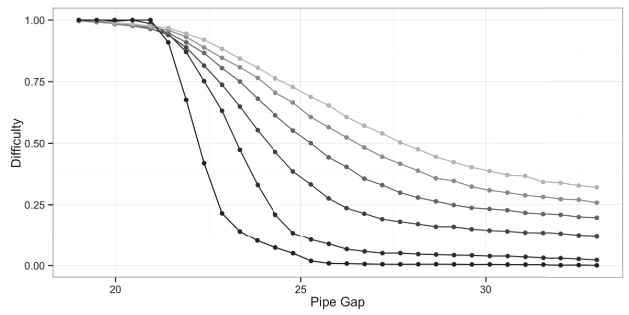

Figure 6: Increasing pipe gap p_g decreases difficulty. Each line shows a different setting for player precision σ_p. Lighter lines plot lower precision, modeling poorer performance, so are more difficult for the player model.

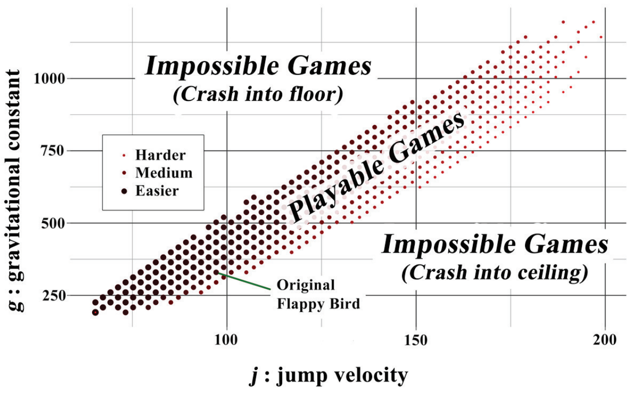

Figure 7: Sampling game space in two dimensions, jump velocity j vs gravity g, shows a narrow band of playable games.

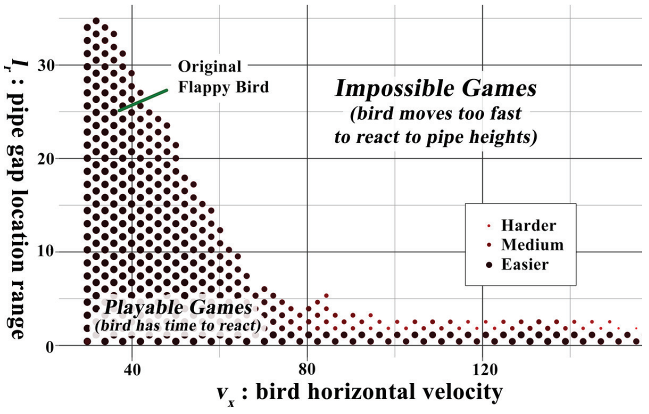

Figure 8: Sampling in two dimensions, l_r pipe gap location range vs v_x horizontal bird velocity. High speeds require a lower pipe range, so the player has enough time to react to the varying gap locations.

game, measuring distances and sizes in pixels, and performing simple computer vision background segmentation and quadratic curve fitting to extract velocity and gravity parameters.

6.1.1 Single Dimensional Sampling

Beginning with the original Flappy Bird, we keep each parameter fixed and vary one at a time, exploring along each 1 dimensional axis, and sampling at fixed intervals. Figure 6 shows a plot of pipe gap p_g vs difficulty d. Each line uses a different value for player precision σ_p. Lighter lines in the figure have a higher standard deviation, so the AI makes more errors, and the game is more difficult for the player model. As one expects, the model predicts that players with less precision will find the same game more difficult to play, and narrower gaps are harder for everyone.

6.1.2 Two-Dimensional Sampling

Varying in two dimensions shows dependent parameters, and can help designers find interesting relationships between dimensions of game space. We visualize these results using dot plots, displaying varying difficulty by the radius and color saturation of each point.

For example, we see in Figure 7 that jump velocity j and gravity g are dependent. When gravity is too high or low relative to jump velocity, the bird crashes into the floor or ceiling. In the middle, gravity and jump velocity are balanced, and we see as they increase together, the game gets more difficult – faster reaction times are required as the bird is moving rapidly up and down. Lower values of gravity and jump velocity give the player more time to react and are easier to play. Holes and islands are due to stochastic simulation, and can be reduced with a larger number of simulations n_s.

In Figure 8 we see the hyperbolic-like relationship between bird horizontal velocity v_x versus pipe gap location range l_r. As the bird moves faster, there is less time to aim for the next pipe gap. As we increase the gap location range, the bird must on average travel further to clear the pipes, so the player requires more time to adjust.

Our best visualization results came from using hexagonal sampling. We also tried evenly spaced rectangular grids, which are easy to implement and work well with standard contour plotting algorithms, and stratified sampling, which avoids clumping from uniform sampling, ensures that the entire space is well covered and eliminates aliasing artifacts from grid sampling.

6.1.3 High-Dimensional Sampling

Proceeding to higher dimensions, we sample the entire space by varying all the parameters without keeping any fixed. Latin Hypercube Sampling [33] provides us with well distributed stratified sampling that covers the entire space. This is useful for sampling, but is not useful to visualize beyond two-dimensions.

With high-dimensional sampling, we can then approximate and reconstruct the difficulty function at any point in game space by using Moving Least Squares (MLS) [18], without running the simulation. This technique is useful if the simulation is slow, either (a) offline when searching for good parameters or (b) online during dynamic difficulty adjustment where there is no time to run expensive simulations. We experimented with MLS reconstruction, showing that it performed well, but our original simulation ran fast enough we did not need to use it during search or visualizations.

6.2 Exploration via Optimization

Global optimization algorithms are designed to efficiently search parameter space to find the optima of a function. We use optimization to find the parameters that will give a specific difficulty d_{target} by searching the parameter space to find $\mathbf{p}$ using:

$$\arg\min_{\mathbf{p}} \sqrt{\left(d(\mathbf{p}) - d_{target}\right)^2}$$

where $d(\mathbf{p})$ is the difficulty of a game with parameters $\mathbf{p}$. Because we evaluate d stochastically, we are optimizing over a noisy func-

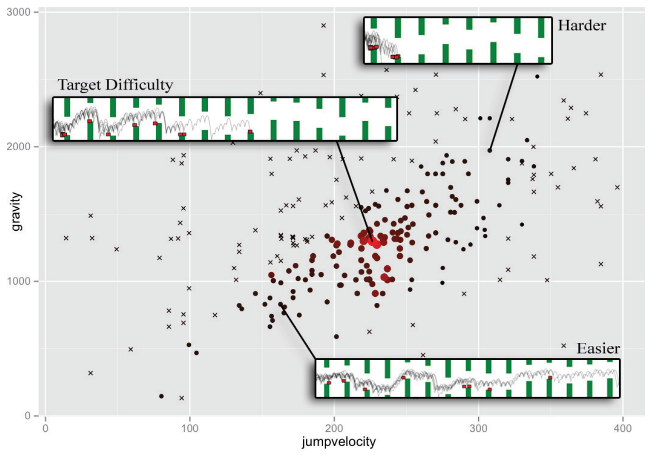

Figure 9: Differential Evolution Optimization helps us search for a target difficulty. Each point indicates a variant tested to find the target; X indicates impossible games, Dot size and color indicates closeness to the target. 280 points were searched to find a value within .1% of the target $d = .3$. Three example games are shown, but every dot represents a unique variant.

tion and can't use strategies that rely on differentiable functions and gradients [26]. Differential Evolution [23] is designed for stochastic functions, and the *DEoptim* optimizer [17] quickly explores game space find our target with .1% accurracy in approximately 10-20 iterations using the default options for the library. Figure 9 shows points explored to find a specific difficulty of $d = .3$ for the user study in Section 7.

Finding all games that match a particular difficulty d is considerably more difficult, as most optimization algorithms aim to find a single global optimum, not a set of values. One solution is to use multiobjective evolutionary algorithms [37] which can handle multiple optima. Another solution is to partition the space using an n-dimensional octree, and search with *DEoptim* for an matching game inside by each cell. If a game is found, the cell is subdivided and searched recursively. This approach increases in speed as the cell sizes get smaller, but requires repeated evaluations of the same space as a cell is subdivided. These search techniques can be sped up using parallelization [40].

6.3 Computational Creativity

We implement *exploratory computational creativity* [2, 44] as the process of finding playable variants that are as different as possible from existing versions that have already been explored. By allowing the system to vary every design parameter, unique combinations can quickly be explored by the AI and optimizer. Unplayable levels are invalidated, and the system only returns the games that can actually be played by humans. The designer then can examine the unique playable games to find inspiration and new ideas.

Using our system, we have generated interesting and surprising variants, such as "Frisbee Bird" which has a very different game feel. This variant, shown in Figure 10, was created by allowing the optimizer to vary every design parameter, including speed, player width, player height, jump velocity, gravity, pipe distance, pipe width, and pipe randomness. The optimizer returned a game with a wide flat bird which moves horizontally very fast but slow vertically and requires short bursts of rapid presses followed by a long pause while the bird floats in the air. This unexpected variant, discovered by our system while using the optimization algorithm, still

Pair	$d(A)$	$d(B)$	n agree perceived	% agree perceived	n agree score	% agree score
1	0	.1	18/20	90%	16/20	80%
2	.3	.2	17/20	85%	15/20	75%
3	.3	.4	8/20	40%	13/20	65%
4	.1	.2	16/20	80%	15/20	75%
5	.5	.4	15/20	75%	16/20	80%
6	.1	.3	18/20	90%	20/20	100%
7	.4	.2	17/20	85%	16/20	80%
TOT	-	-	109/140	77.9%	111/140	79.3%

Table 2: Our predictions generally agree with players' evaluation of perceived difficulty and with their actual scores. Only in one case (Pair 3) do we disagree, and in that situation, the players perceive the pair oppositely to their own performance.

relies on the mechanics and rules of the original but is a significantly different play experience.

To ensure that we have covered the entire game space to find unique variants, we can use stratifed sampling which ensures sampled points are distant from each other in parameter space.

7. USER STUDY

We now compare our difficulty predictions with difficulty rankings obtained through human play testers. Our user study is composed of 3 stages: Questionnaire, Ability Measurement, and Game Difficulty Comparison. Each part is performed in a web browser; participants are monitored to ensure they complete the entire study correctly. Our questionnaire asks for gender, age, game playing experience, and exposure to Flappy Bird games and its clones.

In the Ability Measurement stage, we measure precision, reaction time, and actions per second. We measure the standard deviation of precision σ_p by asking participants to tap a button when a horizontally moving line aligns with a target. We repeat the test 20 times at 3 different line speeds. The standard deviation of the measured time error for each speed is used in our player model (see Sec. 3). We measured precision to range between $\sigma_p = 35.9$ms for the slowest line speeds and $\sigma_p = 61.1$ms for highest line speeds, verifying the speed-accuracy tradeoff [43].

To measure reaction time, the user is asked to press a button as soon as they see a horizontally moving line appear on the right side of the window. The average delay in time between when the line first shows up and the user presses the button is τ. We measured this average value as $\tau = 288$ms, which is in line with classical reaction time measurements [13]. We do not currently use reaction time standard deviation as we found the reaction time does not have a large impact on our AI for playable games.

To measure actions per second, the user rapidly presses a button for 10 seconds. The mean was 7.7 actions per second. This is an upper bound, as players can not be expected to perform this fast with accuracy, and some testers would focus so intently on tapping the button that they would not be able to play accurately at that rate.

In the Game Difficulty Comparison stage, we ask each participant to play 7 pairs of game variants and to rate the difficulty of each game relative to the other in the pair (see Table 2). For example, variant 1A is compared with 1B, 2A is compared with 2B, etc. We also measure the mean score each player achieves on each variant. The user can switch between variants in each pair and can play each variant as many times as they want. For each pair, they are asked to compare the games on a 7-point scale: "A is {much easier, easier, a little easier, the same, a little harder, harder, much harder} than B."

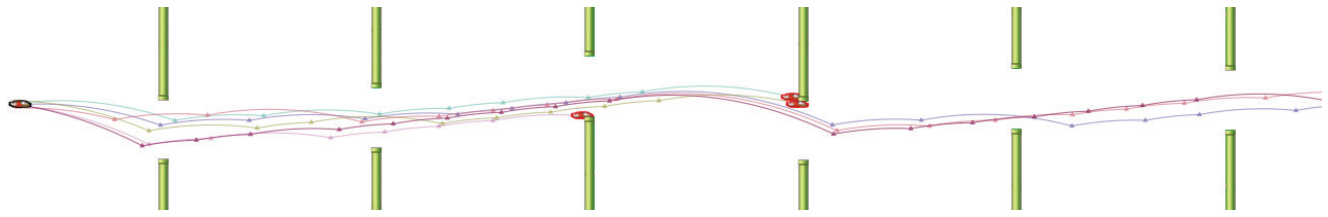

Figure 10: "Frisbee Bird" was unexpectedly created by our system, an example of computational creativity. We show 6 simulated paths with jump points indicated by triangles. The player is very wide and thin, with weak jump velocity and gravity that gives the feeling of floating. We have manually tweaked the generated parameters for better visualization.

To create the pairs, we used the techniques of Section 6.2 to generate 6 unique variants with varying difficulty between $d = 0$ and $d = .5$. If $d(A) < d(B)$, then we predict that A is easier than B. To limit the number of changing variables, we only changed gravity g and jump velocity j while fixing all other parameters. Gravity and jump velocity are interesting variables since they need to be experienced in-game to understand how the variant feels, and they would normally require design iteration and play testing to set correctly.

For each variant pair, we compare our prediction with what each participant rated more difficult, and their actual performance on each variant, as shown in Table 2. We note if the participant (a) *agrees* or *disagrees* with our algorithm's prediction and (b) achieves a higher mean score on the variant predicted to be easier. Since every pair has a different predicted difficulty, if a user indicates "same" we say they disagree with our prediction. We tested on 20 users and found our population agreed with our prediction 77.9% (109/140) of the time when asked, and agreed with our prediction 79.3% (111/140) when comparing mean scores. Both are highly significant under a binomial test. However, some pairs have more agreement than others, as seen in Table 2. In particular, Pairs 1 & 6 have strong agreement, and we agreed with the majority in all cases except one. Pair 3 stands out in that participants performed worse on the variant they perceived to be easier, and more investigation is needed to better understand this case as our model provides no explanation for this behavior.

8. DISCUSSION

In this paper, we argue that using player modeling and survival analysis helps us better understand the relationship between game parameters and player experience in minimal action games like Flappy Bird. The algorithms presented here are not difficult to implement and can be used by game designers to tune games and explore parameter variations. Even if not using optimal methods, designers can implement a simple AI that makes human-like errors and explore their own game designs.

Our approach is focused on modifying game parameters for an existing game, and therefore explores a subset of game variants where only the parameters change. We do not claim to explore the larger class of variants created by adding or subtracting rules.

The framework presented here can be used for many types of games, not just simple action games like Flappy Bird. Given any parameterizable game defined with a vector of game design parameters, these general methods can be used to explore game space: start with a vector that defines a game variant, test the variant with an AI player model that makes human-like mistakes, evaluate the AI's performance, adjust parameters, and then repeat with a new variant. However, the player model presented primarily models dexterity, player accuracy, and timing errors: therefore it is only suited for action games where difficulty is determined by motor skill, not by path planning or strategic decisions.

There is future work to be done on improving the accuracy of our player model. For example, increased time pressure decreases a player's precision, so precision is not entirely independent from the other game space parameters. Future extensions should explore a dynamic player model which adjusts accuracy based on the speed at which the player must react to challenges.

Our survival analysis only looks at a single dimension, d, which describes the score distribution of our simulation. Future work should explore additional output variables such as time played, optimizing for multiple dimensions at once.

In theory, we expect high-dimensional game space to have dependencies which can be squeezed into a lower dimensional space using model reduction techniques, finding the intrinsic dimensionality of a game space. This would reduce the number of knobs a designer needs to adjust, assuming that there is some lower dimensional iso-manifold of estimated difficulty. In addition, lower dimensional spaces are faster and easier to search.

We aim to provide game designers with tools and methods that can help them design better games faster. Setting game parameters can be assisted by algorithms, and take us towards more practical "computer-aided game design" – designers and computers working together to craft a better player experience.

9. ACKNOWLEDGMENTS

Special thanks to Katherine Isbister, Julian Togelius, Frank Lantz, Clara Fernandez-Vara, Bennett Foddy, Charles Joseph Pratt, Chrystanyaa Brown, the NYU Game Center, NYU Game Innovation Lab, and the anonymous FDG reviewers for their suggestions and comments. We also thank Kyle Christopher Fox for extracting the original Flappy Bird parameters, Max McDonnell for providing Flappy Bird score distribution data, and Trevin Hiebert for helping with demonstration software. Finally, we would like to especially thank Dong Nguyen for making Flappy Bird, for giving us permission to use his images, and for inspiring this research.

10. REFERENCES

[1] E. Alpaydin. *Introduction to machine learning*. MIT press, 2004.

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

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

[4] K. Compton and M. Mateas. Procedural level design for platform games. In *AIIDE*, pages 109–111, 2006.

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

[6] H. Desurvire, M. Caplan, and J. A. Toth. Using heuristics to evaluate the playability of games. In *CHI'04 Human factors in computing systems*, pages 1509–1512. ACM, 2004.

[7] A. Drachen and A. Canossa. Towards gameplay analysis via gameplay metrics. In *MindTrek Conference: Everyday Life in the Ubiquitous Era*, pages 202–209. ACM, 2009.

[8] A. Drachen, A. Canossa, and G. N. Yannakakis. Player modeling using self-organization in tomb raider: Underworld. In *CIG*, pages 1–8. IEEE, 2009.

[9] M. S. El-Nasr, A. Drachen, and A. Canossa. *Game analytics: Maximizing the value of player data*. Springer, 2013.

[10] J. Fraser, M. Katchabaw, and R. E. Mercer. A methodological approach to identifying and quantifying video game difficulty factors. *Entertainment Computing*, 2014.

[11] V. Hom and J. Marks. Automatic design of balanced board games. In *AIIDE*, pages 25–30, 2007.

[12] R. Hunicke and V. Chapman. AI for dynamic difficulty adjustment in games. In *Challenges in Game Artificial Intelligence AAAI Workshop*, pages 91–96. sn, 2004.

[13] R. Hyman. Stimulus information as a determinant of reaction time. *Journal of experimental psychology*, 45(3):188, 1953.

[14] E. T. Lee and J. W. Wang. *Statistical methods for survival data analysis*. John Wiley & Sons, 2013.

[15] R. A. Magill and D. Anderson. *Motor learning and control: Concepts and applications*, volume 11. McGraw-Hill New York, 2007.

[16] O. Missura and T. Gärtner. Player modeling for intelligent difficulty adjustment. In *Discovery Science*, pages 197–211. Springer, 2009.

[17] K. Mullen, D. Ardia, D. Gil, D. Windover, and J. Cline. DEoptim: An R package for global optimization by differential evolution. *Journal of Statistical Software*, 40(6):1–26, 2011.

[18] A. Nealen. An as-short-as-possible introduction to the least squares, weighted least squares and moving least squares methods for scattered data approximation and interpolation. *URL: http://www. nealen. com/projects*, 130:150, 2004.

[19] A. Nealen, A. Saltsman, and E. Boxerman. Towards minimalist game design. In *FDG*, pages 38–45. ACM, 2011.

[20] M. J. Nelson. Game metrics without players: Strategies for understanding game artifacts. In *Artificial Intelligence in the Game Design Process*, 2011.

[21] D. Nguyen. Flappy bird. Apple App Store, 2013.

[22] C. Pedersen, J. Togelius, and G. N. Yannakakis. Modeling player experience for content creation. *Computational Intelligence and AI in Games*, 2(1):54–67, 2010.

[23] K. V. Price, R. M. Storn, and J. A. Lampinen. *Differential Evolution - A Practical Approach to Global Optimization*. Natural Computing. Springer-Verlag, January 2006.

[24] Pwnee Studios. Cloudberry kingdom. PC and Console, 2013.

[25] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2014.

[26] A. Redd. Optimising a noisy objective function. `http://www.r-bloggers.com/optimising-a-noisy-objective-function/`, 2013.

[27] H. Rinne. *The Hazard rate : Theory and inference (with supplementary MATLAB-Programs)*. Justus-Liebig-UniversitÃď't, 2014.

[28] A. Saltsman. Canabalt. Apple App Store, 2009.

[29] N. Shaker, G. N. Yannakakis, and J. Togelius. Towards automatic personalized content generation for platform games. In *AIIDE*, 2010.

[30] A. M. Smith, C. Lewis, K. Hullet, and A. Sullivan. An inclusive view of player modeling. In *FDG*, pages 301–303. ACM, 2011.

[31] A. M. Smith and M. Mateas. Variations forever: Flexibly generating rulesets from a sculptable design space of mini-games. In *CIG, 2010*, pages 273–280. IEEE, 2010.

[32] P. Spronck, M. Ponsen, I. Sprinkhuizen-Kuyper, and E. Postma. Adaptive game ai with dynamic scripting. *Machine Learning*, 63(3):217–248, 2006.

[33] M. Stein. Large sample properties of simulations using latin hypercube sampling. *Technometrics*, 29(2):143–151, 1987.

[34] S. Swink. *Game Feel*. Morgan Kaufmann, 2009.

[35] W. H. Teichner. Recent studies of simple reaction time. *Psychological Bulletin*, 51(2):128, 1954.

[36] J. Togelius, R. De Nardi, and S. M. Lucas. Towards automatic personalised content creation for racing games. In *CIG, 2007.*, pages 252–259. IEEE, 2007.

[37] J. Togelius, M. Preuss, and G. N. Yannakakis. Towards multiobjective procedural map generation. In *Workshop on Procedural Content Gen. in Games*, page 3. ACM, 2010.

[38] J. Togelius and J. Schmidhuber. An experiment in automatic game design. In *Computational Intelligence and Games, 2008.*, pages 111–118. IEEE, 2008.

[39] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne. Search-based procedural content generation: A taxonomy and survey. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3):172–186, 2011.

[40] R. A. Valenzano, N. Sturtevant, J. Schaeffer, K. Buro, and A. Kishimoto. Simultaneously searching with multiple settings: An alternative to parameter tuning for suboptimal single-agent search algorithms. In *Third Annual Symposium on Combinatorial Search*, 2010.

[41] W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S*. Springer, New York, fourth edition, 2002. ISBN 0-387-95457-0.

[42] B. Victor. Learnable programming. *Viitattu*, 2:2012, 2012.

[43] W. A. Wickelgren. Speed-accuracy tradeoff and information processing dynamics. *Acta psychologica*, 41(1):67–85, 1977.

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

[45] G. N. Yannakakis and J. Hallam. Real-time game adaptation for optimizing player satisfaction. *Computational Intelligence and AI in Games, IEEE Transactions on*, 1(2):121–133, 2009.

[46] G. N. Yannakakis and J. Togelius. Experience-driven procedural content generation. *Affective Computing, IEEE Transactions on*, 2(3):147–161, 2011.

[47] A. Zook, E. Fruchter, and M. O. Riedl. Automatic playtesting for game parameter tuning via active learning. In *FDG*, 2014.