Dynamic Difficulty Adjustment

Major Qualifying Project Report
submitted to the faculty of
Worcester Polytechnic Institute
in partial fulfillment of the
requirements for the Degree of Bachelor of Science
in Computer Science

Submitted: *January 11, 2012*

Sponsoring Agency: Worcester Polytechnic Institute, Computer Science Department

Submitted to:

Professor Joseph Beck, WPI Professor, Project Advisor

Respectfully submitted by:

Alex Kuang
Thomas Lextrait

Abstract

Game developers are constantly looking for ways to accommodate the widest range of players. One solution is DDA (dynamic difficulty adjustment), where the game attempts to learn the player's preferences and adjust itself accordingly. However, discussion of DDA systems assumes existing functional models of fun in videogames. This paper presents a detailed study in which a multiple linear regression model for fun was built for a simple top-down shooter game (Space Warrior), using two studies with subjects crowd-sourced from Amazon Turk. Analysis of the results shows that player demographics like age and gender have a great influence on fun in games like Space Warrior, while effects from dynamic adjustment systems are weak in comparison.

Contents

List of Figures

1 Introduction

The amount of enjoyment that a player derives from a game is often dependent on the skill level of the player in relation to the difficulty of the game. A ”hardcore” game requiring quick reflexes and great mechanical skill may frustrate a casual gamer; inversely, a casual game may be too easy and bore a more experienced player. A difficult game which causes the player to fail often may be seen as stimulating for a skilled player, but unfair to a more inexperienced gamer.

The idea of DDA (dynamic difficulty adjustment) was born from efforts to mitigate the effects of the range in this difficulty/skill relationship and cater to players with as wide a range of skill levels as possible. The main idea of DDA is to have facilities in the game available to gauge the player's performance and skill, and adjust the difficulty accordingly during gameplay to provide the most consistent (and hopefully most fun) experience for the player. While the core goal is, in effect, to maximize fun for the player through dynamically adjusting the difficulty of the game, this presupposes a model of fun currently exists which can be used to maximize player enjoyment.

Thus, understanding fun is an integral part of any game-related research; formally modelling fun to have a practical tool for use in game design even more so. However, as noted above, differences in skill, background, and preference make what players consider ”fun” vary wildly between each other.

Additional difficulty arises when the differences between game types and genres come into play–for example, the fast reflexes required for an action game versus the relatively slow style of a turn-based strategy game. A gamer more predisposed towards first-person shooters may find a numbers-based puzzle game dull, and the reverse could also be true in that a gamer used to playing puzzle games would find action games too hectic and confusing to be any fun.

Even within the same ranges of experience and genre preferences, gamers can consider themselves "casual" or "hardcore." For example, even assuming two players with the same experience in Tower Defense games, the "casual" player may prefer a game with a more laid-back approach, whereas the "hardcore" player would prefer a game that requires meticulous planning and a lot of trial-and-error through learning from repeated failures.

In both cases, even if the games were (theoretically speaking) well-designed, the player could be not having fun due to being put out of their comfort zone. The above scenario one of the many dynamics that are at play when considering a full model of fun. This complexity in modelling fun as a whole is why it is useful to determine what measurable factors have the most impact on a player's "fun", and offer the most return on investment from the side of game developers.

2 Methodology

This paper contains the results on experiments analyzing a Space Warrior game using a simple method of modelling fun: a multiple linear regression model. First, we built a simple top-down Space Warrior game. Players were allowed to play in rounds of one minute, with a survey after each round asking them to assign a numerical score of funness and fairness to the round played. A before-game survey was also implemented to gain more information on player demographics.

Various logging functions were put in place to record player activity and game progression. Two studies using this game were conducted independently with subjects from Amazon Mechanical Turk, a public service provided by Amazon that allows small tasks to be offered and completed by users in exchange for cash payments. The log data were then aggregated and analyzed to build multiple linear regression models, whose fits were tested against their respective data and against each other.

2.1 Space Warrior Game

2.1.1 Game Design and Implementation Tools

This study was originally set to use an existing game (Starfighter) codebase which had already had DDA-related hooks built in as a part of a Master's Thesis[4]. However, the fully functioning code could not be gathered at a timely manner, and we had to come up with an alternative option.

We decided to rewrite a rudimentary version of the game from scratch. We spent a short time experimenting with various languages and frameworks that were available, ending with a first prototype in C#. However, we found that the C# game ran with an inconsistent frames per second (fps) and did not support many desired features such as diagonal movement or holding down keys to shoot while moving. In addition, the code was hard to maintain and extend and the game would only be playable on windows, so the first prototype was deemed a non-starter.

The final version of the game is built in the Python programming language using the pygame[2] game development framework. Python was chosen because of the faster development speed, and pygame is a well-established framework within online communities and guaranteed additional speed while abstracting away system-level details like enforcing constant fps. In addition, Python (and pygame) are supported across multiple platforms and promised the most flexibility in distribution even to players without Python and pygame installed via tools that supported building self-contained executables and installers across multiple platforms. This particular study ended up targeting windows users, so the game was built and distributed using one such tool, py2exe[3].

2.1.2 General Game Info

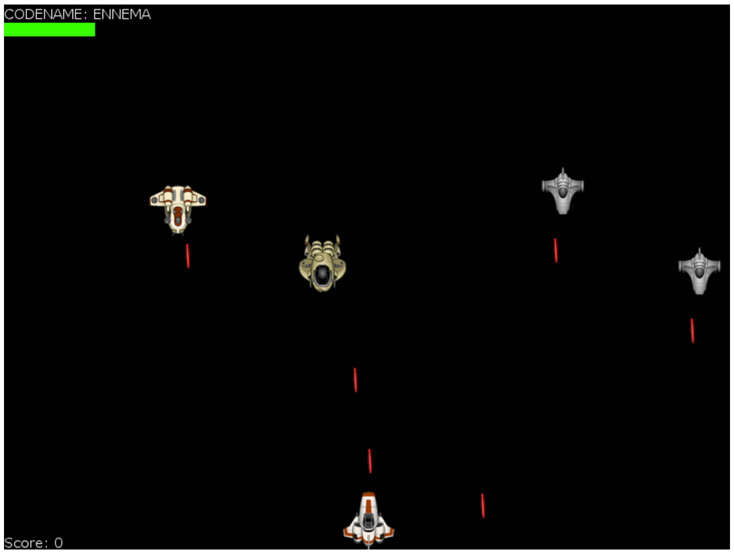


Figure 1: A screenshot of Space Warrior gameplay

The game (henceforth, Space Warrior) allows players to control a single ship facing a variety of enemies. Games were structured into short 60-second rounds (specifics below), and the controls were mainly keyboard-based, with the directional keys used for movement and a single key for firing. At any time within a game round, the Escape key could be used to quit the game. Both players and enemies had only one weapon type, with the player's ”bullets” moving 5 times faster than the enemies'. The game placed enemies every 20 frames at a randomized location on the top of the playing field starting with a random velocity, while the player started at the very bottom. Enemies changed the directions of their movement at random intervals; however, they stayed within the constraints of the game window, bouncing and changing direction if encountering an edge. In addition, they were programmed to not enter the bottom 20% of the screen. The player was given the freedom to move anywhere in the window. Space Warrior also kept track of a player score which was based on the enemies the player killed (the score incremented was the max HP of the enemy killed, multipled by 100 to keep the score from looking too low and discouraging players), with a penalty of 50% of the current score for each player death. Players were given unlimited lives, and scores accumulated across game rounds.

2.1.3 Source Code Location

The full source for the game can be found on bitbucket online at https://bitbucket.org/alexkuang/dda11-spacewarrior/ . Mercurial, commonly known as hg, is recommended to pull the full code from the repository. To run the game from source, both the Python programming language and the pygame framework are required; however, py2exe can be used to build a Windows executable with the existing files and configurations in the source folder. The game can be run with `python starfighter.py`.

For more information on the inner workings of the game, refer to Appendix D - Source Code.

2.2 Subject Selection With Amazon Turk

Subjects were chosen from Amazon Mechanical Turk, a public service where users can sign on and complete small semi-skilled tasks for a payout specified by the owner of the task. The users can be filtered based on previous task experience, custom qualifications, and completion ratings indicative of reliability based on past tasks. This study utilized Mechanical Turk as it is a much easier and timelier way to gather a good pool of anonymized subjects. A reward of $.25 was decided on after a few test runs as a good amount; it is large enough for it to be worth the users' time, but not so large that they play only for the reward. However, the task required a minimum of 2 rounds completed before a reward was given in order to ensure usable data. The reward amount did not change based on number of rounds completed beyond 2 rounds to keep financial gain from becoming a motivating factor in playing longer games.

Since there is a financial element, there is always the possibility that players will try to cheat the game for payment. There were many safeguards to prevent this. Amazon Turk itself provides the Turk user id for each user as a unique way of identification. Turk itself will also prevent users from completing the study multiple times for the reward money. Task creators are also given the option of creating their own verification systems on which to base acceptance of task completions. The Space Warrior study used a custom codename embedded in the game (which the players would not see without at least running the game once). Game log file submissions were required before task acceptance. The server that received the game files kept track of submitted logs to ensure that users did not cheat the system by completing the study multiple times by using the same log file under multiple accounts. The server also analyzed the log files for suspicious behavior, such as a score of 0 across all rounds or no player shots being fired, and decide whether to accept or reject the log file. And finally, if the log file was valid and unique, a hash was returned for entry in amazon turk as a final form of verification that the log file was indeed submitted to the server.

This system served well, as it caught many attempts at user cheating. The most common attempt at circumvention was to try using the same data multiple times for the financial reward. Several users also tried to use data from the preliminary tests (to gauge an appropriate price and to get a feel for the Turk system) in the formal study, and entered the wrong codename. While it is possible they thought the game that was used for the final studies was the same as the alpha version used in the preliminaries, only one user tried to refute the claims that they were trying to cheat, replying that their codename and log file were correct despite evidence to the contrary.

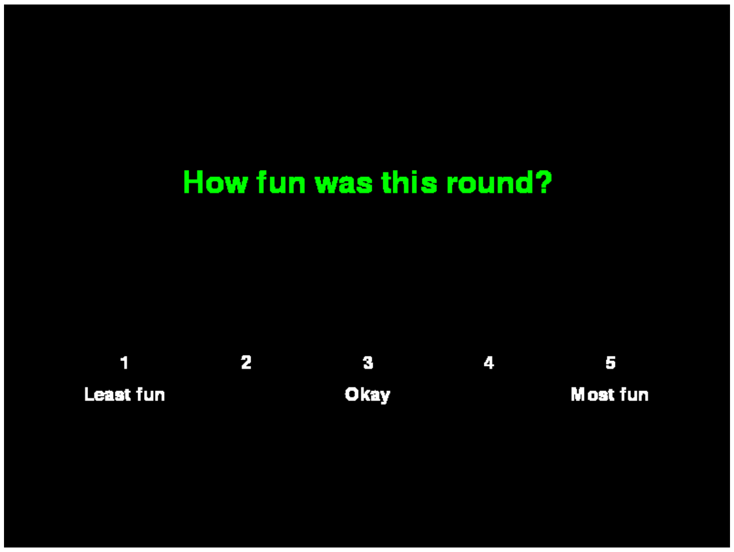
2.3 Trials and Data Collected

Before starting the game, players were given a short survey asking for a small amount of info from them such as gender, age, and propensities as a gamer (see the Appendix B for survey questions). Then the gameplay started, in rounds lasting one minute each. At the beginning of each round, the player began with full health and no enemies on the screen. Enemies then started spawning in the manner described above.

A short survey was presented to the player after each game round, asking them to assign a numerical value to the amount of fun they had (1 - least fun, to 5 - most fun) and how fair they thought the round was (1 - biased towards the computer, to 5 - biased towards the player). The player was also asked how fun and fair the current round was compared to the last round where applicable.

Logging of game events (such as player shots, player hits, enemy shots, enemy spawns, enemy deaths, etc) took place during gameplay. The logged events and attributes for each player between the pre-game survey, between-round survey, and various game events can be found in Appendix C.

In addition to this logging, between rounds, enemy and player attributes were randomized in order to try affecting the perceived fun for the player and gain multiple data points per user. Enemy power (e.g., `battlecruiser_power`, `fighter_power`) and the number of enemies (`max_enemies`) were modified. Player hp, firepower (power per shot), and shot recoil, (`player_base_hp`, `player_fire_power`, `player_fire_recoil` respectively) were also random-


How fun was this round?

1 Least fun | 2 | 3 Okay | 4 | 5 Most fun

Figure 2: One of the survey questions presented between game rounds

ized between rounds.

The goal of this was to explore the total space of game attributes such as could be affected by a DDA system. As such, the bounding values chosen for the randomized game attributes were such that they would make the game deliberately too easy or deliberately too hard. Judging by the results and the game comments overall, the correct values were chosen for all of the attributes excepting `player_fire_recoil`; many players felt that the recoil time between shots was far too long, and the range for that should have been shifted towards faster shots.

While this study focused mostly on player and enemy attributes and their effects due to time and resource constraints, there are other game factors that could be explored as well. Movement speed, for example, on both the player's and enemies' ships, could play a significant role in the difficulty of the game. Similarly, game-wide attributes such as player and enemy spawning time, game round time, and enemy type distribution, could also be modified dynamically.

3 Subject Demographics and Information

As two studies were conducted (see Methodology), two subject pools were used to build 2 models for fun.

The first trial used a total of 123 users. Of the 123 users, 103 were male and 20 were female. Most of the users were in their early 20s: 7, 16, 9, 14, 5, and 10 users were ages 20-25 respectively. The youngest user from this study was 18, and the oldest 59.

The second trial had 165 users, with 134 male and 31 female. Most users in this case were in their early 20s as well: 11, 13, 16, 17, 8, and 13 users were 20-25 respectively. The youngest user in the second study was 17, and the oldest 56. Two users in the second study also indicated an age of 0; it can only be presumed that they were not comfortable revealing their age.

The 123 users from the first study played a total of 442 rounds, averaging 3.6 rounds per user. Though all players were required to complete up to round 2, only 61 of the players who completed round 2 also completed round 3. The number tapers off sharply, with only 33 players who completed round 3 completing round 4, and 21 of the 33 players completing round 5. This trend continues, ending with less than 10 players completing round 8. There was one major outlier, who completed 31 rounds of the game and achieved a final score of 99875.

The second study resulted in a similar number, with 165 users playing a total of 572 rounds for an average 3.5 rounds per user. 94 users completed past round 3, and of the 94, 62 users passed round 4. Trial 2's numbers taper off much more gradually, with the most rounds completed being 20.

Unfortunately, accurate location information for users was not available. However, since game logs were in user local system time and Turk provided a submission time for each log, it was possible to calculate their timezone offsets for a rough idea of where they were. This analysis showed that the majority of the users were from Asia, around the same meridian as India and Malaysia.

4 Models for Fun

For the modelling of fun, a method that was both effective and relatively non-time-consuming was needed; in this case, statistical analysis became the de-facto choice. After doing a preliminary fit on the data and checking that the fit did not indicate that a linear model was an obvious non-starter, a multiple linear regression method was chosen to model fun.

To build the linear regression models, the technique of forward selection of regression variables was used. Forward selection means starting with no variables and adding the variables from the model space (in this case, the parameters logged in the game), and keeping the ones that were statistically significant while throwing away the ones that did not make much of a difference in the model fit.

By Occam's Razor, between comparable models the one with the least complexity is always preferred; therefore, the following were singled out as dependent variables in the models for fun: player firing recoil (`player_fire_recoil`), the current round number (`round_num`), player firepower (`player_fire_power`), and player age (`age`). The resulting linear regression models from each of the two trials are as follows:

$$\begin{aligned} fun_1 &= 4.066 \\ &\quad -0.033 * player\_fire\_recoil \\ &\quad -0.036 * round\_num \end{aligned}$$

$$+0.010 * player\_fire\_power$$
$$-0.017 * age$$

$$\begin{aligned} fun_2 = \; & 3.82 \\ & -0.027 * player\_fire\_recoil \\ & -0.054 * round\_num \\ & +0.011 * player\_fire\_power \\ & -0.0066 * age \end{aligned}$$

Essentially, the two linear models predict fun via a simple linear equation using the parameters as the independent variables. The first number in each equation is the intercept, not tied to any independent variable. Since in this case, it's impossible for all the other variables to be 0, the intercept does not have any significant meaning on its own.

Both models show that as recoil increases, fun decreases (since the modifier for recoil is negative). This means that the less often player shoot, the less fun they generally tend to have. Similarly, round number also has a negative multiplier. This indicates that players have less fun the longer they play; this is supported by the fact that the number of players still playing decreases with round number as shown in the Subject Demographics section. Fun also decreases as age increases, which is in line with how the age is distributed within the subject pools; most of the players are in their early 20s, with the user count tapering off to zero as age increases. Fun increases with player firepower, suggesting that players have more fun the more firepower they have.

There are small differences in the multipliers for both models. However, for the most part the differences are small. The only exception to this is round number, which is a much bigger influence in the second model when compared to the first. This is evidenced by the full 30% increase in the multiplier in the second model. Despite the increase in magnitude though, the sign still remains the same, which indicates that in the second model, people got tired of the game much more quickly. This is probably due to the 30+ round outlier in the first study, which softened the influence of round number on the fun value in the first model.

5 Stability of the Fun Models

As noted in the Subject Demographics section, the two subject pools were very similar, both in age and gender distribution. The two sets of data also resulted in similar-looking models; the coefficients are so close together and the two models had similar adjusted r^2 (.157 and .102). The only substantial difference was, as noted above, in round number and that can be explained at least in part by the anomaly in the data of the first trial.

All of these similarities beg the question of whether the data can be combined into a collective subject pool and used to generate one final model, which would carry more weight by virtue of the combined sample size and simpler to work with than two separate models. More importantly, if the models are similar enough to be combined this way, it would be indicative of a model stability across multiple studies.

To determine whether this would be a viable decision, it's important to see if the two models are stable and quantitatively similar enough to be interchangeable. In order to do this, an analysis was run using the models generated from both sets of data. The predicted values generated by fun_1 in data set 1 were correlated with the values generated by fun_2 on data set 1, and then the predicted values generated by fun_2 on data set 2 were correlated with the values generated by fun_1 on data set 2.

The resulting correlations (r) were .965103 and .9654016. These are very high values, indicating that the cross-predicted values are very closely correlated, and ultimately that the two models are very similar. This means that the two trials' data show that the results were consistent across both studies, which in turn indicates a stable model and a solid methodology. Furthermore, the two models can be combined to form one collective model:

$$\begin{aligned} fun_{final} &= 3.89 \\ &\quad -0.029 * player\_fire\_recoil \\ &\quad -0.043 * round\_num \\ &\quad +0.010 * player\_fire\_power \\ &\quad -0.010 * age \end{aligned}$$

6 Analysis

6.1 Model of Fun

The final fun model kept very closely in line with the separate models from each study, which is no surprise given how similar the models were to each other. For the final model using the pooled data, the multiple linear regression p-value was $2.2 * 10^{-16}$, indicating statistically significant results. However, the adj-r^2 value for the entire model was only 0.1297, indicating that only about 13% of the fun variability in the subjects can be explained by the variables used to build the model. While this is not insignificant, clearly there is room for improvement.

6.2 Differences in Gaming Experience

One major factor that we thought would be a source of huge difference in the way the models of fun behaved was in gaming experience. That is, theoretically an experienced gamer's preferences differ significantly from a new gamer's preferences. As such, an interesting analysis to do would be to sort the subject pool based on gaming experience, split them into three groups ("beginner", "intermediate", "advanced"), and build models of fun on each in the same way we built the overall model. The players were split into 3 groups of equal size based on their answer for the monthly gaming amount pre-game survey question; the beginner group contained gamers whose answers were 0 - 420 minutes per month, the intermediate 420 - 1530, and the advanced 1530 - 36000. Each group consisted of 96 users. The resulting models for fun are as follows:

$$\begin{aligned} fun_{beginner} &= 3.73 \\ &\quad -0.026 * player\_fire\_recoil \\ &\quad -0.016 * round\_num \\ &\quad +0.007 * player\_fire\_power \\ &\quad -0.007 * age \\ (adjusted\_r^2 &= 0.08246) \end{aligned}$$

$$\begin{aligned}
fun_{intermediate} &= 3.68 \\
&\quad -0.038 * player\_fire\_recoil \\
&\quad -0.073 * round\_num \\
&\quad +0.011 * player\_fire\_power \\
&\quad -0.004 * age \\
(adjusted\_r^2 &= 0.1409)
\end{aligned}$$

$$\begin{aligned}
fun_{advanced} &= 4.14 \\
&\quad -0.002 * player\_fire\_recoil \\
&\quad -0.048 * round\_num \\
&\quad +0.013 * player\_fire\_power \\
&\quad -0.023 * age \\
(adjusted\_r^2 &= 0.1781)
\end{aligned}$$

While these results are not wildly varying enough to question experimental methodology, there are some key and interesting differences in the models. One of the biggest differences is in player_fire_recoil, whose effects diminish drastically with the advanced players. This suggests that players who are more experienced in gaming can aim better, and therefore have less trouble with a higher recoil than newer players might. Age also seems to play a smaller role with the beginning and intermediate gamers, and a much bigger one with advanced gamers. This implies that older advanced gamers have much less fun with a game of this type, getting bored very quickly, while younger advanced gamers have relatively more fun. In the case of beginning and intermediate gamers, it's possible that age does not play as much of a factor due to their relative lack of gaming experience.

Also notable is the trend in the adjusted r^2 values. In the three models, the beginner gamers' model had the lowest r^2, and the advanced players the highest. A lower r^2 means that there is more that is unaccounted for in the beginner fun model than there is in the intermediate and advanced models built with these variables. This supports the conclusion that gaming experience level has a significant effect on what is important in building a model for fun.

Both firepower and round_num's effects remain fairly consistent over all three pools though, suggesting that more firepower brings more fun and that fun diminishes as time passes regardless of gaming experience. There is a small difference in the intercepts of the equations as well, with the advanced players' equation's intercept going up past 4, but those have no meaning in the context of the study as explained above.

6.3 Effect of User Demographic Properties

The first area that the studies focused on was properties affected with user demographics like age and gender. What's interesting is to run a model with the pre-game survey answers against fun. This gives us a good idea about how much variability can be accounted for by knowing the information from the survey about the user: e.g. age, gender, gaming experience, game genre preference, etc. Running the numbers just with survey answers as independent variables yields an adj-r^2 of .1202.

To show how much there is to gain from a pre-game research standpoint, a model can be created using user id's against fun. This model sheds light on the things that might differ between each user and the importance of the sum of those differences; essentially, how much room there is for improvement stemming just from having more information on user attributes. While a linear model is still created for this model, an ANOVA is used because user id is a nominal variable, not quantitative. The resulting model yields an adj-r^2 of .3098. This means that if all the information possible about the user beforehand were, it could account for roughly 31% of the variability in fun. The difference between the two, $0.3098 - 0.1202 = 0.1896$, shows that a lot more can be gained by knowing more information about the user beforehand.

6.4 Effect of DDA-Related Properties

Analysis of the efficacy of DDA-related attributes is slightly harder, since there is no easy way to represent the entire space of variables that can be influenced by a DDA system. However, the fact that the variables in the fun model above that turned out most significant include as many non-DDA-related (round number and age) as DDA-related (fire recoil, fire power) hints that DDA is not as powerful a solution as it initially seems.

To test this formally, the rest of the variables that can be of use to a DDA system can be reintroduced into the model for fun. These variables fall under two categories: player and enemy "attributes", and player "performance." The attributes category includes variables that can be manipulated in a DDA system such as: HP, firepower, movement speed, and shot recoil. Player performance includes variables that can be kept track of but not manipulated directly by a DDA system, including: player death rate, player shot accuracy, and player kill rate.

By introducing both kinds of variables into the model, we are in effect pretending that we can manipulate both kinds of variables; a real-life DDA system, which doesn't have access to the performance variables, could certainly not do better than this. Therefore, the difference in the new model's performance in predicting fun will indicate how much of an improvement is afforded by a theoretically ideal DDA system with the ability to manipulate things that a real-life system would not be able to.

The new model with the rest DDA-related attributes used in additional to the original four yielded a p-value of $2.2 * 10^-16$ and an adjusted r^2 of 0.1582. The p-value is identical to the original 4-variable model, and indicates that the results are statistically significant. However, the adjusted r^2 is only slightly higher than the original model's, which had a value of 0.1297. Given that an additional 11 variables were used to calculate the model, a difference of barely 3% is underwhelming, to say the least. These results suggest that even if it were possible to have complete control of all in-game DDA-related attributes (i.e., the perfect theoretical DDA system), it wouldn't help in the model of fun all that much.

7 Conclusions and Future Work

The results of this study indicate that variability in the fun of Space Warrior-style games is very much dominated by a player's demographic attributes. Using just a simple model built with 4 variables 12.9% of the variability in fun could already be explained. Statistical analysis shows that this can be increased to about 31% with more player demographic information. In contrast, a model built with all DDA-related attributes in addition to the simple model's variables only brings the figure up by 3% to around 15.8% despite the amount of additional information used to build the model.

This supports the conclusion that even an ideal DDA system would hav a weak effect on the model of fun, especially in contrast to the impact of information gleaned from the pre-game survey. It further suggests that knowing the demographic well, by finding the most effective questions to ask and obtaining the most important information, is much more important than engineering a dynamic tweaking sysem for in-game attributes.

The major caveat is that these findings apply specifically to Space Warrior-style games–that is, top-down shooter games with short round durations. DDA has been utilized in successful games such as Revenge of the Titans, which have used much longer playing times, and therefore allow for much more information to be collected based solely on gameplay. In addition, games that require more contemplation and intellectual challenge may benefit more from a DDA-based approach for fun than a game such as Space

Warrior which is more reliant on reflex and mechanical dexterity.

Future work should focus on the above shortcomings. Models of fun should be built using games that require a longer playtime and allow for more information to be collected on the player's gameplay. In addition, this approach can be expanded to other genres of games that differ in gameplay from Space Warrior in order to gauge differences between game types, and to determine what factors are most dominant for fun in each.

Work more closely related to this study should focus on trying to expand the space of the variables explored in the game. For example, attributes like spawn time and enemy unit type distribution were left constant in this study. Though it was determined that player-related attributes were dominant in the model for fun, there is still no explanation for the other 18

Lastly, analysis from splitting the subject pool into three groups based on experience showed that while the models remained consistent overall, there were still some key differences, namely in the influence of recoil times and age on fun for players of different gaming experience levels. Further work should be done investigating the effect of gaming experience on what has the most effect on what a player considers fun.

References

[1] ddaex Dynamic Difficulty Adjustment: Destructoid http://www.destructoid.com/good-idea-bad-idea-dynamic-difficulty-adjustment-70591.phtml

[2] pygame Python modules http://pygame.org/wiki/about

[3] py2exe Python Distutils extension http://www.py2exe.org

[4] Applying Causal Models to Dynamic Difficulty Adjustment in Video Games http://www.wpi.edu/Pubs/ETD/Available/etd-042610-090201/

A Survey Questions

A.1 Pre-Game Survey

Pre-game survey questions:

1. What is your gender? (Male / Female)
2. What is your age?
3. Estimate how long you play video games per month (Hours, Minutes)
4. Do you like shooting games? (Yes / No)
5. Provide any comments you may have

A.2 Round Survey

Between game round survey questions:

1. How fun was this round? (1 to 5)
2. How fair was this round? (1 to 5, 1=computer has advantage 5=player has advantage)
3. Compare how fun with previous round (More Fun / Less Fun)
4. Compare how fair with previous round (More Fair / Less Fair)

B Sample Log

```
 2011-10-30 10:25:36,908 ::  Space Warrior initialized, logging active.
2011-10-30 10:25:38,905 ::  Intiating round 1
2011-10-30 10:25:38,907 ::  ================
2011-10-30 10:25:38,907 ::  Player Base Stats
2011-10-30 10:25:38,907 ::  max hp 100, speed 15, weapon power 5,
weapon rate 10
2011-10-30 10:25:38,907 ::  ================
2011-10-30 10:25:38,907 ::  ================
2011-10-30 10:25:38,907 ::  Enemy Base Stats
2011-10-30 10:25:38,907 ::  Drone:  max hp 20, speed 3, weapon power
1, weapon rate 60
2011-10-30 10:25:38,907 ::  Fighter:  max hp 55, speed 4, weapon
power 4, weapon rate 30
2011-10-30 10:25:38,907 ::  Scout:  max hp 40, speed 5, weapon power
2, weapon rate 50
2011-10-30 10:25:38,907 ::  BattleCruiser:  max hp 80, speed 2, weapon
power 8, weapon rate 90
2011-10-30 10:25:38,908 ::  ================
2011-10-30 10:25:38,913 ::  Max number of enemies set to 11
2011-10-30 10:25:38,914 ::  Hivemind set Scout power to 28.
2011-10-30 10:25:38,914 ::  Hivemind set Drone power to 14.
```

```
2011-10-30 10:25:38,914 ::  Hivemind set Fighter power to 56.

2011-10-30 10:25:38,914 ::  Hivemind set BattleCruiser power to 112.

2011-10-30 10:25:38,914 ::  Hivemind set player power to 38.

2011-10-30 10:25:38,914 ::  Hivemind set player fire_rate to 14.

2011-10-30 10:25:39,454 ::  Fighter spawned.

2011-10-30 10:25:40,015 ::  Fighter spawned.

2011-10-30 10:25:40,292 ::  Fighter fired weapon.

2011-10-30 10:25:40,573 ::  Drone spawned.

2011-10-30 10:25:40,851 ::  Fighter fired weapon.

2011-10-30 10:25:41,075 ::  Player fired weapon.

2011-10-30 10:25:41,131 ::  Fighter fired weapon.

2011-10-30 10:25:41,134 ::  Drone spawned.

2011-10-30 10:25:41,694 ::  Fighter fired weapon.

2011-10-30 10:25:41,697 ::  Drone spawned.
```

C Logged Attributes

`gender`

`fighter_power`

`age`

`round_timestamp`

`scout_power`

`player_hits_sec`

`reward`

`predicted_fun`

`user_id`

`player_power`

`player_base_power`

`work_time`

`player_score`

`battlecruiser_power`

`survey_comp_fun`

`player_base_speed`

`timezone`

`round_num`

`player_base_rate`

`avg_enemies_screen`

`max_enemies`

enemy_shots_sec

survey_how_fun

monthly_gaming

player_deaths

shooting_games

enemy_hits_sec

player_shots_sec

player_fire_recoil

survey_how_fair

enemy_kills_sec

survey_comp_fair

log_key

drone_power

player_base_hp

player_kills_sec

D Source Code

D.1 Overview

Note: The overall structure of this game uses an Object-Oriented paradigm, and familiarity with OO vocabulary is highly recommended for comprehension.

The main body of the game runs from starfighter.py. In the main routine, universal logging is set up via python's `logging` module so that messages can be easily logged across any files in the game (more details below). The main body of the game is encapsulated in a Game object, which keeps track of game-wide state variables such as the upper fps limit, player score, round number, and all the various avatars on the screen.

The heavy lifting occurs in `Game.run()`, inside a universal while loop. Each iteration of the loop represents one "tick" in the internal game clock; how fast these "ticks" happen is controlled by the pygame tick function, which makes sure that everything runs consistently and (to the best of its ability) at a constant frame rate. Within each iteration, input is handled and the appropriate changes in state are made (movements, deaths, spawns, etc), and the game screen is updated with `pygame.display.flip()`. This loop also contains the appropriate escapes in the cases of player death, round transition, etc.

D.2 pygame Sprites

Every entity in the game is a subclass of pygame's `Sprite` class. This gives the entity objects access to built-in pygame functionality that tracks live/dead state and membership in sprite groups. The rest of the attributes and actions are up to the sprites themselves. In Space Warrior, each sprite has an `update()` function. This function takes care of the updates required in the progression of the game, and is called from the main game loop after each tick. `update()` takes care of firing shots, handling collisions, and living/dying as appropriate.

Collision detection is done via pygame's `Rectangle` class. While it can be defined in a custom manner, this game gets each rectangle from the image of the sprite in question. pygame has a number of built-in collision functions which allow for such detections using the sprite's `rect` attribute (which should contain an object of the `Rectangle` class). These functions can detect collisions both on an individual sprite basis as well as on a sprite-to-group basis or a group-to-group basis. See the pygame documentation for more detail.

D.3 Players and Enemies

Enemies and players are defined in `enemy.py` and `player.py`, respectively, and both subclass `Sprite`. The player class is relatively simple, as there is only one player at any given time, and defines the key behaviors outlined above. `enemy.py` includes the base `Enemy` class, from which every enemy subclasses. These subclasses largely differ in what sprite they use and unit-specific attributes like HP, speed, and firepower. These attributes are class variables which are referenced in object initiation; this makes it easy for background systems (like a DDA system) to change attributes on the fly and have them apply to all new enemy spawns.

It should also be noted that enemies and players have shot objects that belong to different classes, as it makes it easier to detect the appropriate collisions.

D.4 GUI objects

The game takes place in a window that is drawn by pygame. As noted before, all entities subclass Sprite, and this includes any part of the gui such as the HUD (heads up display) and any prompt text that may appear. The only difference is that these entities do not interact with the others, and only update themselves as needed. No collision detection is performed, and the state changes of the GUI are not dependent on any other parts of the game. The calls to their `update()` methods are placed at the end, since each screen update is done in order, and the HUD must never be obstructed by any other entities on the screen.

D.5 Hivemind

The hooks that don't belong strictly in game implementation are all handled by the Hivemind object. Each Game instance has an attached Hivemind, which is aware of any and all happenings in the game. This makes `hivemind.py` the perfect location for future implementation of AI or any DDA-like features. At the time of this writing, the Hivemind is largely responsible for carrying out the attribute randomization needed at the beginning of each game round for the various parts of this study, as well as any details in the distribution of enemy unit types and spawning.

D.6 Surveys and Rounds

The rounds are implemented in the main game loop by keeping track of the number of ticks that have passed, and using the `Game.round_time` attribute. At the end of each round, a subroutined is launched which initializes a survey for the player. The `Survey` object is essentially a specialized version of the `Game` object, with the same general structure. In the case of the survey though, the main run loop only renders prompts with the survey questions and answers, and takes mouse clicks on the answers as input.

The survey uses `Question` objects which are very specialized for the questions used in this study. This implementation was used because of time constraints; it works for the current study's survey, but should be improved if anything of significantly more complexity is needed in the future. Particularly, instead of having specialized cases for each question, it is much better to have the Question object itself scale the question and answer prompt text on the screen and wrap lines accordingly, which pygame does not handle automatically.

D.7 Logging

The logging in the game is done via python's `logging` module, which allows a top-level redirection of logs to a destination of choice. In this study, the logs are all directed to `game_log.txt`, which is created in the folder in which the game resides. This can be changed easily at the initialization of the game.