Using Player Profiling to Enhance Dynamic Difficulty Adjustment in Video Games

Senior Project

Computer Engineering Department

California Polytechnic State University, San Luis Obispo

Aaron Burke

December, 2012

Advisor

Dr. Michael Haungs

Table of Contents

Introduction 3
Game Description 4

Why Action Script? 4
Controls 4
Head-up Display 5
Enemy Types 6
Power-ups 8
Stages 8

Design 8

Navigation 8
Game Engine 8
Stage Data and Generator 9
Collision Detection 10
Enemy Class 11
Skill Heuristics 13
Additional Considerations 14
Types of Difficulty Adjustment 14

Testing 15

Data Collection 15
Finding the Skill Heuristic 16
Movement 16
Life Loss 18
Fire rate 19
Enemy Destruction 20
Power-ups 22
After Game Interview 23

Related Games 23
Enemy Count 23
Enemy Generation 24
AI Adjustment 24
Fine Tuning from a Set Scale 24
Failure Based 25
Negative Feedback Loop 25

Conclusion 25
Future Work 26
Adding Visually Noticeable Adjustment 28
Final Notes 28

INTRODUCTION

Video games have certainly come a long way since the days of pong. From humble beginnings of the first arcade machines they have bloomed into a multi-billion dollar industry and a global force to be reckoned with. Just like with movies there are enough genres out there to fit any demographic of gamer, and that's not even just the game's content but also the gameplay.

Unlike other types of media though, games are meant to be interacted with and played, which brings up new issues with accessibility and keeping things interesting for different levels of skill. *Dynamic game difficulty adjustment* is a process of automatically changing the way the game behaves based on the skill of the player in order to avoid them from becoming bored (if the game is too easy) or frustrated (if the game is too hard). Traditionally in games the difficulty will increase at a steady rate along the course of the game with earlier levels being easier and later levels being harder. This effectively motivates the player to improve their skill or character's overall effectiveness (through levels, items, and/or stats) in order to gain rewards of access and glory giving them a sense of accomplishment. Usually this difficulty curve is based only on a difficulty level selected at the beginning of a game, regardless of whether or not the player is truly at that skill level. On the other hand by dynamically adjusting the difficulty of a game, it can create a custom difficulty experience for any type of player, keeping the player interested from the beginning to the end without too much frustration or boredom.

Here's where the problem lies: Is it possible to dynamically balance a game without the players noticing? A big giveaway about this system is the tendency for oscillating difficulty level around the player's actual skill level. This has been coined as a "rubber band effect" seen from the a racing game whose CPU cars would gain speed when behind a player and lose speed when in front of a player causing them to rubber band back and forth around the players car. By analyzing the data taken from many different types of gamers through gameplay testing the players linear difficulty graph slope can be found. Once a target slope has been established the difficulty will increase or decrease based on how the player performs; that is, if they are falling too far behind or staying too far ahead of the target curve it will have to readjust again to keep them from getting bored or frustrated.

By including a questionnaire after each testing session the distance off from the difficulty curve players can go before they truly feel bored or overwhelmed can be determined. By adjusting as little as possible players will still feel a sense of achieving greater power by beating something that was difficult enough to them, without being overwhelmed or bored by the difficulty curve that they quit playing. Unfortunately this method is not widespread due to many challenges this system poses, although various forms of it has been implemented in some successful

games already. Through research and testing a concise conclusion can be drawn about the art of doing these dynamic changes as smoothly as possible in a way that keeps the player always in a state of flow.

GAME DESCRIPTION

The game used for testing this is a fairly simple, straightforward and easy to understand one. It was developed in action script and it's a basic top down shooter in the style of the Nintendo Entertainment System. The art includes sprites from many memorable games of that era. After selecting a stage the player is spawned onto a play field that slowly scrolls to right. Enemies will start to come on from off screen and obstacles will move onscreen from the right and move left. Once the player reaches the end of the stage, that stage's boss will appear. If the player dies they will be prompted to go back to stage select or continue. If the boss has been reached they can continue from there.

The arrow keys move the player around the screen and change their firing direction. This means if the player hits up their avatar will not only move up but turn to face upward (and will therefore fire upward when the fire key is pressed) This mechanic helps the player maneuver better since enemies can come from any direction at the player, plus it's a mechanic not often seen in this genre so it introduces a slight learning curve to even experienced players. Pressing fire creates a bullet that travels forward and is removed when it hits an enemy, damaging it for the power of the player. Like-wise when the player is hit by an enemy bullet, or collides with an enemy, they take damage for the amount of power the enemy has. Expanding on the simple concept of a top down shooter game by having many different types of enemies, all with unique behaviors and abilities, it expands the meaningful decisions a player must make adding strategy and not just reflexes to the learning curve.

Why Action Script?

This game was originally designed to be distributed online through various flash gaming sites and gather large amounts of data which would be sent back for analysis. Even though this idea was eventually scrapped it still allowed for much easier cross platform development.

Controls:

- Up, Down, Left, and Right Arrows: move up, down, left, and right.
- Double tap the directional arrows to boost (doubles speed for tight situations) but boosts only last for a limited time until the meter has to recharge.
- Space bar to fire in the direction you are facing

During initial play testing people complained about having to use the keyboard as controls and since this was developed to recreate the experience of playing on an old gaming console a controller peripheral was added during the data collection phase. This was done by simply mapping usb gamepad input the respective keyboard presses using a free program called JoyToKey that is opened up and left running in the background.

Head-up Display:

The HUD is designed to be as minimalistic as possible while still telling the player all they need to know about the current status of the game.

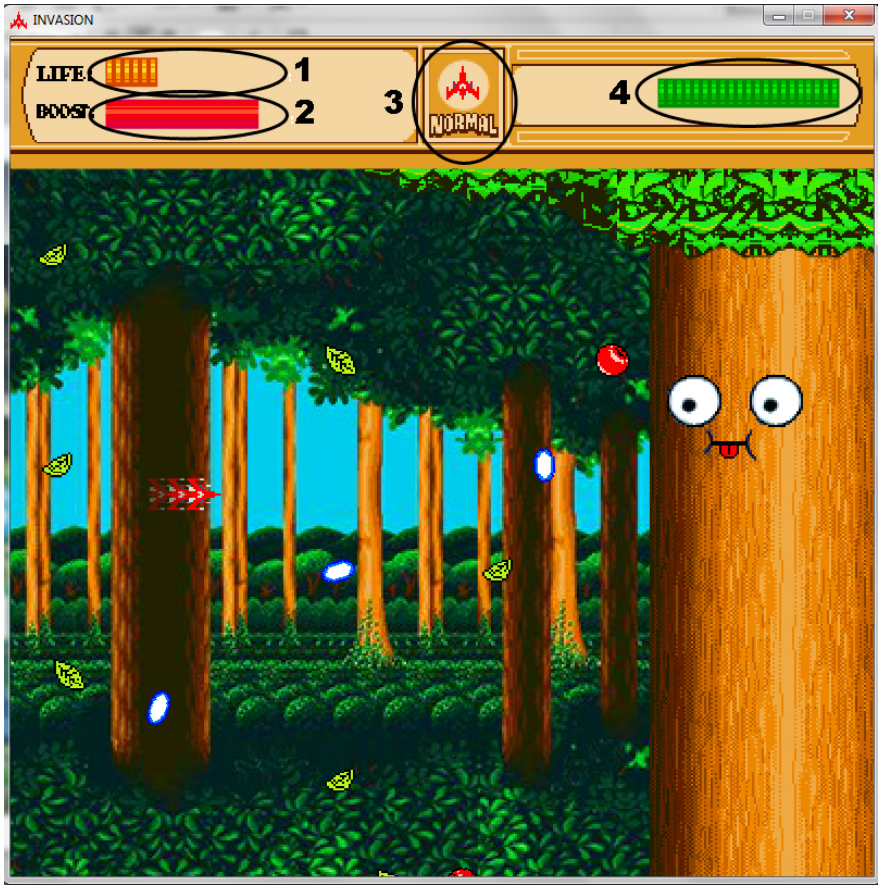

Figure 1- Gameplay screen with HUD labeled

1. Player Life Bar: Displays how much life the player has.

2. Player Boost Meter: Displays how much boost the player has left to use, this meter naturally fills back up over time when the player isn't boosting.

3. Player State: Provides a visual aid informing the player what is happening to them at the moment. For instance, when a player is hit by a ghost the state will change from “normal” to “cursed”.

4. Boss Life: If the boss battle has started it displays how much life the boss has left.

Enemy Types:

- *Bomber*: These simple planes fly by the screen from one side to the other and shoot. They can come from any direction and position but once they’re seen they predictably continue to move forward until destroyed or they escape off screen.

- *Bouncer*: These weird enemies bounce from one end of the screen to the other, either left and right or up and down. They pause for a second upon arrival on the other side and close their eye making them invulnerable to attacks. You have to wait until they are on the move again to try attacking them.

- *Fleet*: These enemies usually move slowly across the screen in packs of 3 to 5. They sometimes will adjust their y position based on where the player is and fire straight ahead.

- *Jumper*: These enemies spring from the lower part of the screen and travel in an arch. Their goal is to collide with the player to inflict damage so be extra careful to avoid them.

- *Follower*: These enemies are slow but will follow the player wherever they go until they are destroyed. To make things worse they also lock on to the player and fire homing bullets while they pursue. As the stage progresses they will become larger and take more hits.

- *Ghost*: These enemies follow the player but only when the player isn't facing them. Turn around to shoot and they turn invisible making them invulnerable to bullets. While they can't be killed they can be pushed off the stage by continuously hitting them with bullets.

- *Magnet*: These enemies putter across the screen seemingly harmless, until you get directly beneath their magnet. As long as you are in their magnetic field they will pull you towards them causing you to collide with them or other enemies in the way.

- *Crushers*: These enemies will come across the screen from the far top or bottom waiting for someone to get in their line of sight. When the player goes directly below them they move down towards the player. Watch out because they can't be killed and when they hit you they will deal massive amounts of damage. The only thing you can do is avoid and wait for them to drift off screen.

- *Lightbulb*: This special type of enemy is only found in the dark cave level. It simply moves from right to left across the stage but can be hard to spot due to the limited amount the player can see on that stage. If this enemy is destroyed it will always drop a special light powerup that increases the players light radius allowing them to see more of the level.

- *Minions*: These are special enemies that bosses summon to help them out. Each boss summons a different type of enemy to help, each with different behaviors. During phase 1 of testing these enemies didn't drop power-ups but after finding no power-ups made it far too hard for beginners to defeat them after enough retries these enemies will also start to drop some.

- *The Boss*: This enemy appears at the end of a level and sports an entire meter of health. Each stage has its own unique boss at the end with a different way of beating it. He can fire bullets, change his attack patterns, and even spawn more enemies. If you manage to beat him any enemies left on the screen will go down with their leader giving you complete victory over the stage!

Power Ups:

In order to give players a reward for destroying enemies sometimes they will drop a power-up. The frequency of their generation can also be used as a sneaky way of assisting with difficulty adjustment.

Life-up: Gives the player extra life. Once the game starts difficulty adjusting the rate at which these can be generated can be increased if they are having trouble progressing through the stage.

Shield: A protective shield forms around the player not only making them invulnerable but also giving them the ability to reflect enemy bullets back at them. If the player is having a hard time destroying enemies or taking hits too often this can be generated to lessen the difficulty.

Light: This is a special power-up only found in the Cave stage. It's used to increase the slowly decreasing spotlight.

Stages:

The player has 4 stages to pick from upon game start. Each stage is labeled with a difficulty rating but a player is free to pick whatever order they want to play the stages in.

Figure 2- Stage Select Screenshot

After all four stages are cleared an extra final stage is available to play.

DESIGN

Navigation:

When an instance of the game is forked off the first class to be created is the Document Class. This class keeps track of creating and destroying all menu screens and the game engine. It also holds the game state information about what stages the player has beat and if they have reached a check point. When it makes a new instance of something it also sticks an eventlistener on it to run the appropriate code when it received button click events or playerdead/playerwin events.

Game Engine:

By far the largest class, this is the heart of the actual game. Once it has set up all the arrays, timers, sound, and backgrounds to the stage it starts its main game timer. Something to note about flash is that when a new class is created it cannot add anything to its stage until it itself is added to its caller's stage. In order to make sure there are no errors an event listener must be added to listen for the ADDED_TO_STAGE event found in flash's event class. Once the game engine is added to the stage this listener will call a function that will finish adding all the necessary objects to stage (determined by what stage it is on) and then start the games main update timer.

Every time the update function is called it loops through the enemy, obstacle, enemy bullets, player bullets, and power-up arrays and checks them for collisions as well as calling their move functions and garbage collecting anything that has moved off the screen. It also checks to see if the player has no more life and will fire off an Avatar Dead event for the Document class to catch as well as a whole lot of other little things like generating power-ups every once and a while when an enemy is destroyed.

There are also a lot of event functions that will run independent of the update timer function. One of these is the keyboard press and release events which set Booleans to true or false letting the update function know how to move the player when it's called. The keyboard press event function also keeps track of the boost timer which determines if the keyboard press is a double tap or not. There is also an event that is called upon the bgm finishing which simply tells it to play again and sets up a new event listener to call the method again upon finishing.

Most important of these many event functions are the various generator functions. These are functions that catch events triggered by enemies and the stage generator class that give the game engine something to add to the stage. While the other respective classes are responsible of knowing what to generate and when, since the main game engine contains the stage it is necessary to pass all pointers to these generated objects to it to add.

Stage Data and Generator:

The stage generator class is what fires off the add enemy events which are caught and added to the stage by the game engine. What enemy it generates depends on what it gets passed from the stage data class. Upon a new game engine being made, a string of the level the player is attempting is passed to the stage data class. Depending on the level passed, the stage data will grab the hard coded data for the given level and start a generation timer. On every tick it passes the next integer to the stage generator. All positive integers refer to a type of enemy or enemies in some type of attack pattern, while negative data represents a

break in enemy generation (ie the time to wait until it generates the next enemy). The individual enemy classes dictate the range of areas it would make sense to spawn that particular type of enemy, which is simply a range of x and or y values multiplied by a random number generated at construction. Once the enemy array has reached its end a boss is automatically added to the stage.

The stage data and generator classes also spawn another type of object: obstacles. These ones are a bit different from enemies because they are not tied to a timer but instead an event. Whenever a new obstacle is sent to the game engine, its width is appended to an integer that keeps track of how much it has to move until it's completely on stage. This way a new obstacle will always be added to the end of an old obstacle allowing for seamless terrain to be added to the stage.

Collision Detection:

Although flash comes with many APIs for handling collision detection, all of these are vector-perfect collision detections. Since all of the images used in the game are bitmaps ripped from sprite sheets all these methods will check when called is the collisions between the bounding-box of the bitmaps (ie it ignores the alpha sections of the bitmaps). While bounding box collision would be fine for more square-like shapes; all of the objects in the game, especially the physical obstacles found in later levels, require detection to be calculated on a pixel level.

While flash doesn't have any direct approach for more accurately detecting collisions between bitmaps, there is a rather clever method posted at this link http://old.troygilbert.com/2007/06/pixel-perfect-collision-detection-in-actionscript3/ discussing the use of blending two layers together to find overlapping pixels. As seen in Figure 3 the method first finds the boundary of intersection between both clips' bounding boxes and then creates a bitmap out of that intersection. It then draws the first clip into the bitmap in red and the other in green, and blends the two together. If there are any overlapping pixels they will now be the color of the green and red channel blended together, so all you have to do is check the resulting bitmap for that color. If the resulting boundaries' of the overlap color in the bitmap is zero (ie there is none of that color) then the function returns there is no collision, but if it did find any it returns true.

```
// get bounding boxes in common parent's coordinate space
var rect1:Rectangle = target1.getBounds(commonParent);
var rect2:Rectangle = target2.getBounds(commonParent);
// find the intersection of the two bounding boxes
var intersectionRect:Rectangle = rect1.intersection(rect2);
if (intersectionRect.size.length> 0)
{
    // the intersection rectangle must be integer size for bitmap data
    intersectionRect.width = Math.ceil(intersectionRect.width);
    intersectionRect.height = Math.ceil(intersectionRect.height);
    // get the alpha maps for the display objects using two separate color channels
    var alpha1:BitmapData = getAlphaMap(target1, intersectionRect, BitmapDataChannel.RED, commonParent);
    var alpha2:BitmapData = getAlphaMap(target2, intersectionRect, BitmapDataChannel.GREEN, commonParent);
    // combine the alpha maps
    alpha1.draw(alpha2, null, null, BlendMode.LIGHTEN);
    // set the search color to the result of the two channels when blended with lighten
    var searchColor:uint;
    searchColor = 0x010100;

    // look for the overlapping color
    var collisionRect:Rectangle = alpha1.getColorBoundsRect(searchColor, searchColor);
    collisionRect.x += intersectionRect.x;
    collisionRect.y += intersectionRect.y;
    // if oa single overlapped pixel was found the boundaries of the intersection will be greater than zero
    if (collisionRect.size.length> 0) {return true;}
    else {return false;}
}
else{return false;}
```

Figure 3- pixel perfect collision detection between two DisplayObjects

Enemy Class:

All of the classes that extend the enemy class have a fire timer. Every time this timer goes off the enemy will generate new bullets, or even new enemies if they are of a boss enemy. Just like the stage generator they communicate with the game engine by firing off an add bullet event or add enemy event.

In addition the boss class, which is just an extension of the enemy class, fire off an event called bossdead when their life is gone. As seen in Figure 4, this event is caught by the game engine so it can go through the necessary steps of ending the game. The Game Engine then fires off an avatar event called avatarwin that is caught by the document class.

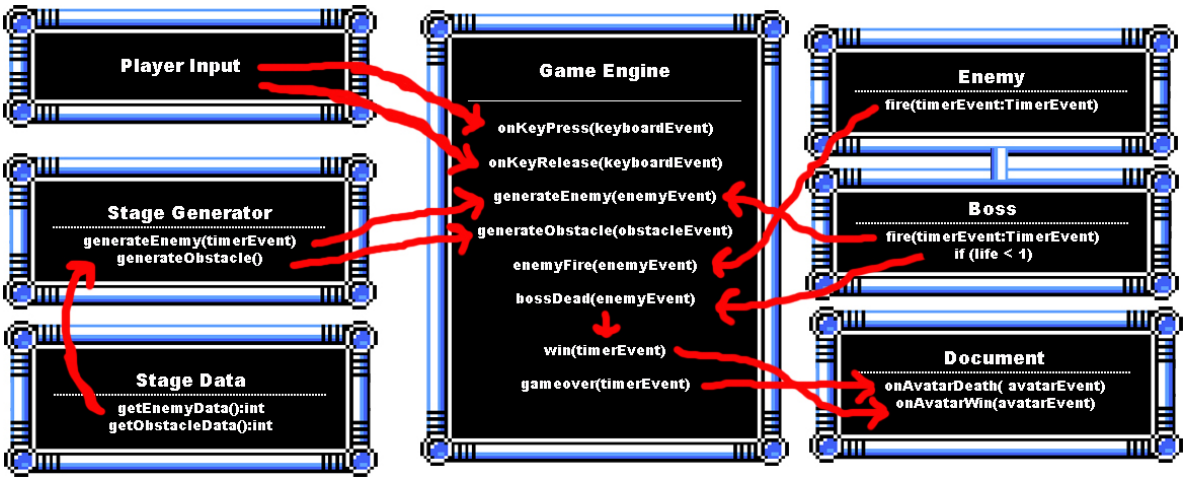

Figure 4- Event Sequence Diagram

Skill Heuristics:

With the main mechanics of the game out of the way, the next step is to keep track of everything the player does and determine what to change based on their actions. To do this we need to make two lists:

What heuristics can be used to judge the players skill level?

- How many bullets they've fired (are they firing at all?)
- How many enemies they hit (are they getting hits on the enemies?)
- How many enemies they destroyed (are they trying to kill enemies or just avoid?)
- How many enemies garbage collected off the screen (ie not destroyed by player)
- How many enemies the player collided with. (are enemies getting too close?)
- The amount of time spent colliding with obstacles. (are they stuck?)
- The ratio of shots fired to shots hit over a current interval of time t. (if too low decrease enemies?)
- The amount of life lost over a current interval of time t. (are they losing life too quickly?)
- The amount of time t between which the player has not been hit. (is it too easy for them? If they haven't taken damage after x seconds spawn more enemies?)
- Amount of powerups/ life amassed (are they taking them?)
- The amount of distance the player has moved (have they found a safe spot?)
- The amount of time they spend boosting (are they moving too much?)
- The use of items strategically (Difficult to measure though…)

- Stats on each enemy (how much that type damaged the player, how much player damaged them)

What can be changed to adjust the difficulty level?

- Power of player
- Player life cap
- Player bullet speed
- Health of player
- Size of boost meter
- Speed of enemies
- Health of enemies
- Power of enemies
- Enemy fire rate
- Enemy bullet speed
- Enemy spawn rate
- Stage scroll speed (obstacle speed)
- Frequency of powerups (types of powerups)
- The amount of benefits the powerups give
- Duration of levels (or duration of gameplay without breaks)

Additional Considerations:

The trick is to adjust difficulty only if there is a good reason. If adjusted too quickly it could result in oscillating along the target point as opposed to converging on it resulting in the "rubber band effect". We don't want them to be overwhelmed but we also don't want to make it so they can never die either, these equations are meant to keep the game BALANCED at their level not prevent anything bad from happening to them.

It also should be cautioned best to only use difficulty adjustment to change what generates as opposed to what's already on the screen. This means a tough type of enemy can be generated and given high stats if the player needs a challenge, but after that its stats should not be adjusted any further. One of the biggest thing players crave is consistency, and if stats and behaviors of enemies start changing right before the player's eyes as opposed to behind the screen it can lead to confusion and will break flow.

Types of Difficulty Adjustment:

We can break up the difficulty adjustment into two distinct types that I would like to do testing on to determine which players prefer, or what combination works best. I can already say that there will have to be a combination of the two, since

both are big pieces of the pie needed to really define difficulty. Ie a player can be really good at aiming and attacking stages of low density enemies and bullets (stat adjustment) but have a hard time with changes in enemy and bullet count.

TYPE 1: STAT ADJUSTMENT

This type if only used makes for a game that looks identical no matter what difficulty you play on. All adjustments are done to stats and abilities

Pros: Keeps things simple, stats are easily adjustable parameters. Makes it harder for players to notice the difficulty switches.

Cons: No visual cue of players skill just the feeling of a negative feedback loop eating at the good players.

TYPE 2: VISUALLY NOTICABLE ADJUSTMENT

This type makes for games that are immediately visually different depending on the difficulty of the game. This would be like changing enemy tactics and fire patterns.

Pros: Good players can tell when they're playing on a higher difficulty (gives them more bragging rights).

Cons: Harder to implement. Any oscillation in difficulty is much more visually noticeable.

TESTING

Data Collection:

By making the project for adobe AIR the file system api can be used to write all data collected to a log file located in the root of the users documents folder. This api allows data to be collected cross platform using the constant File.documentsDirectory which will be different depending on the OS it's running on. A string variable is concatenated throughout runtime, and then appended to the log file upon stage completion on stage failure. By time stamping every level attempt this log file can keep track of everything they do whenever they get on to play and for how long. This log file can be sent back whenever the player has had enough of playing the game.

```
//either creates or accesses the file log.txt in the users documents directory regardless of OS.
var file:File = File.documentsDirectory.resolvePath("log.txt");
var fileStream:FileStream = new FileStream();
var str:String;
//as the game plays data is concatenated onto the string
str += moredatacollectedf;
//once the stage is over (ie the player won or died) all of the new line characters are replaced
//with a constant representing whatever character the OS running it uses for line ends.
str = str.replace(/\n/g, File.lineEnding);
//the file is then open for appending, written to and then closed
fileStream.open(file, FileMode.APPEND);
fileStream.writeUTF(str);
fileStream.close();
```

Figure 5- How to use Adobe Air to write data to a log file

In order to get the most accurate results we have to make sure the test environment is consistent. This means everyone gets the same amount of information about the game, the same controller set-up, and it allows for observation of the player's behavior in addition to collecting their data. This is why the idea of mass distribution and data collection was weeded out. While this method would result in a larger data base, the consistency of the data can vary. Also by running the tests on the same computer notes were able to be taken on their play style and have been noted in addition to the raw data.

Finding the skill heuristic

Although the game will eventually become a difficulty adjusting game, in order to analyze what attributes to skill the difficulty must be set as constant. All players were put in an isolated room with no previous knowledge of the game. They were all given the same one liner on how to play the game, took a quick survey on the type of player they are, given 10 lives to use and see how far they got within that amount, then immediately kicked out. Afterward a survey was taken on their overall experience and their data was put into a determined skill class based on how far they were able to get in 10 lives. If they couldn't pass the 1st stage they were a beginner, if they could at least get to the 2nd or 3rd stage they were an average player, if they could get to the 4th stage they were a good player and if they got all the way to the final stage they were an expert. Time stamps are used in order to keep track of when various data pieces were collected.

In order to proceed onto the difficulty adjustment stage the player's skill graphs must be compared with one another to find patterns and trends. Most importantly, we must find out what the main difference is between the good and the bad players. An interesting thing to note is when presented with a choice of various difficulties of levels all of the players picked the stages in difficulty order.

Movement

The first set of variables we can look at is their movement. A comparison between a bad a good players movement graphs can be seen in Figure 6. These are graphs taken from the same portion of the same level played by different players. Notice how much more complex and fluid a good players movement is. Due to the amount of bullets in the game the player is forced to learn to weave in and out of bullet patterns while taking down enemies on screen. In comparison, bad players are still getting a handle on dealing with everything that's happening on the screen resulting in very simplistic and sporadic movement.

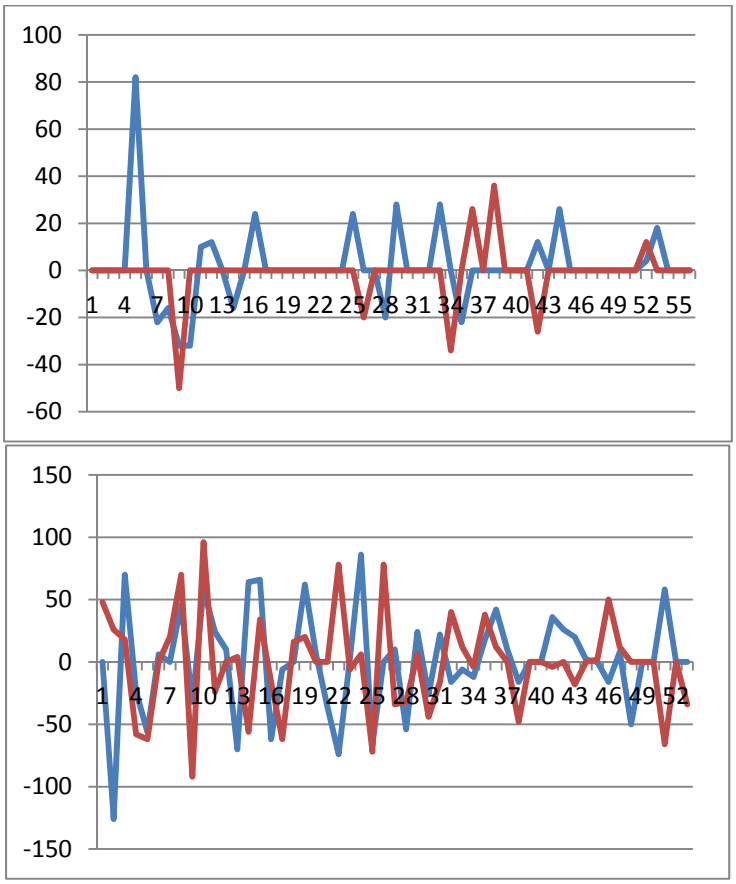

Figure 6- Beginner vs Expert Player Movement

Is this something that needs to be skewed by difficulty adjustment? While there are certainly noticeable differences between the good and bad player's graphs it's also one of the quickest things that the bad players begin to outgrow.

Take Figure 7 for example. This is the movement taken of a bad player from their first run to their final runs. Notice how their graph naturally starts to conform to that of better players over time due simply to the amount of improvement room the player has. This variable doesn't seem like it's one that can be adjusted by force

through game difficulty, but rather a simple game mechanic that is figured out by the player over time.

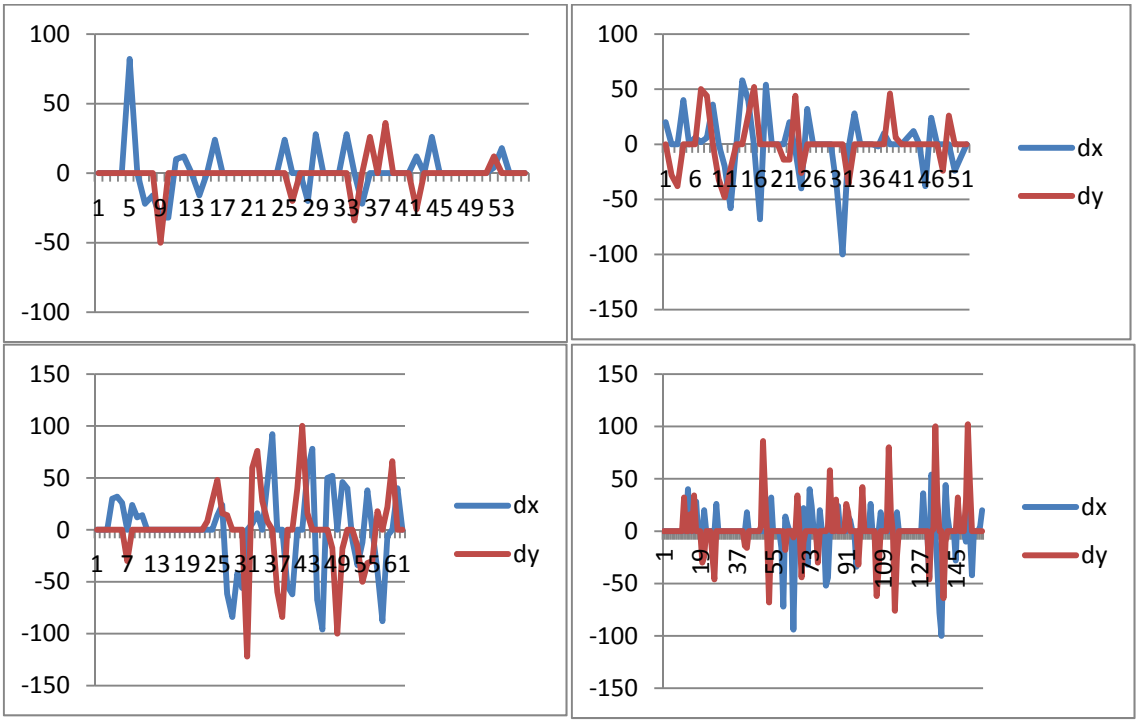

Figure 7- Beginners Movement From First to Last Tries

Life Loss

One of the most apparent differentiators of player skill level is life loss since it's the direct negative consequence that unskilled play results in. A comparison between the good and the bad on the first stage should be apparent.

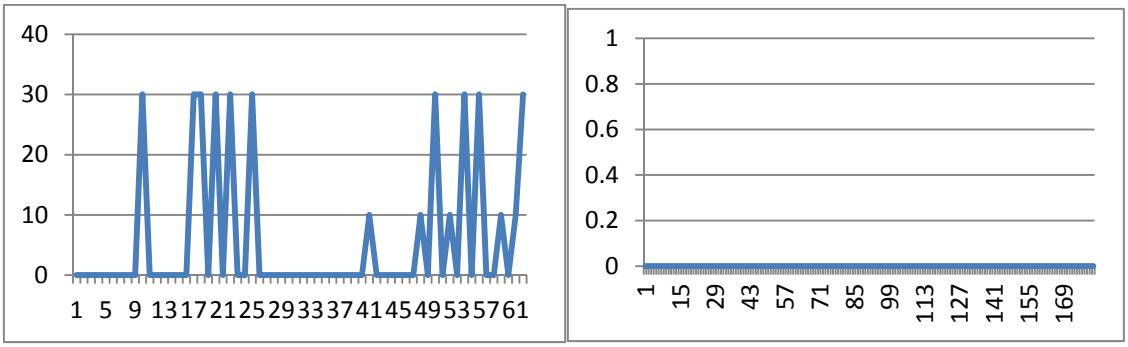

Figure 8- Beginner Life Loss vs. Expert Life Loss

Bad players loose more life more often while good players can go for large amounts of time without ever getting hit. Life loss results in game overs, game overs results in restarting a level, restarting a level results in more time the player has to spend to beat the game. As such life loss is a good candidate for difficulty adjustment,

Fire Rate:

Similar to movement, it seems the better the player is the more active they are. Beginners tended to be more timid with firing at first but quickly began to fire faster like the skilled players do as seen in Figure 9.

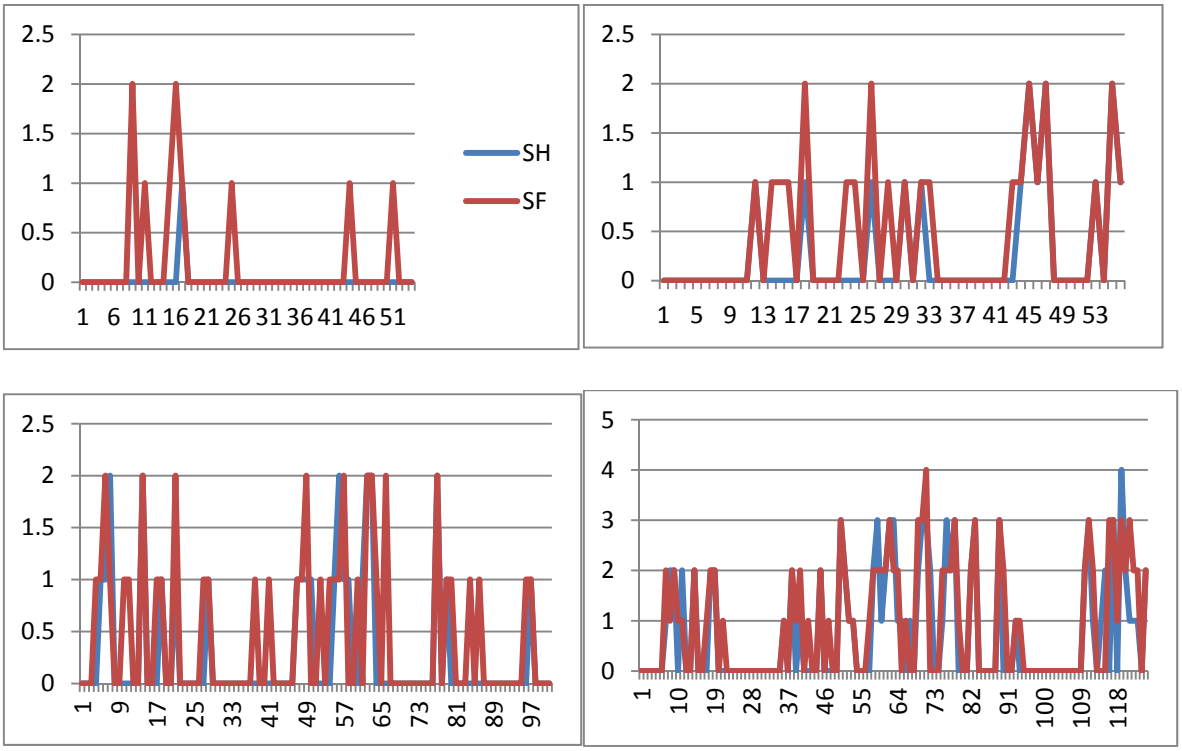

Figure 9- Fire Rate Progression of a Beginner

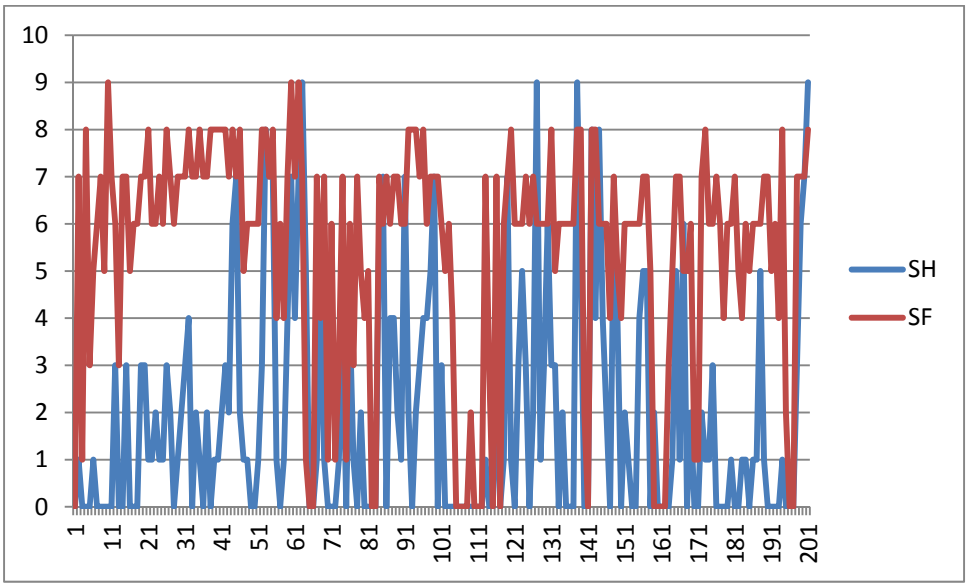

Figure 10- Fire rate of an Expert

So is simply firing more a sign of being a better player? Not necessarily. Since the player is not penalized what so ever for how often they shoot it's only logical that the more you shoot the more likely you'll destroy an enemy. That said what truly determines how much damage they are doing to the enemies is how many of their shots hit. This can be used to separate out skill levels within the quantified classes, with the more successful players having more shots hit than the others.

Enemy Destruction:

These three variables show how the players handle enemies. When comparing Enemy Destruction, Enemies Missed (enemies that escaped off the screen without being destroyed) and Enemies Collided with one can see a distinct difference between players. Beginner players immediately don't know what to do about the kamikaze enemies that head right to them and have a hard time dodging regular enemies so their EC rate is always higher than anything else.

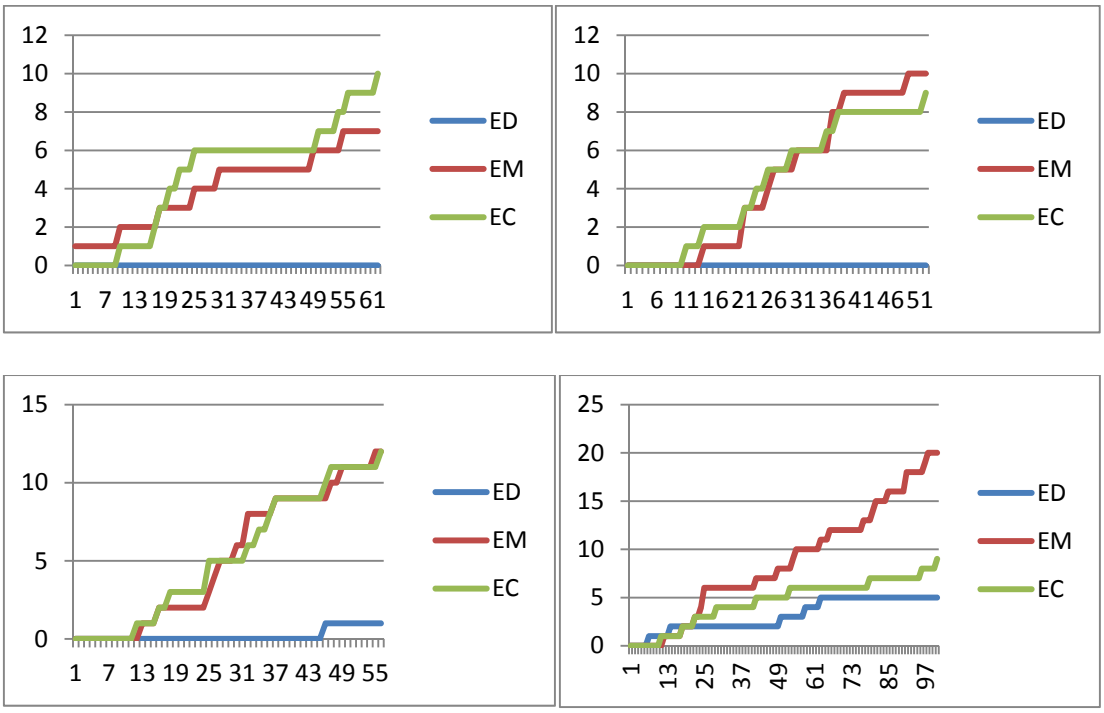

Figure 11- Natural Beginner Progression of Handling Enemies

As seen in Figure 11, just as with the other variables, beginner's skill always increases the fastest. In fact, when comparing a beginner's final run of the first level with an expert's first level run seen in Figure 12, it's clear to see their graph is slowly converging to the expert's as their skill increases.

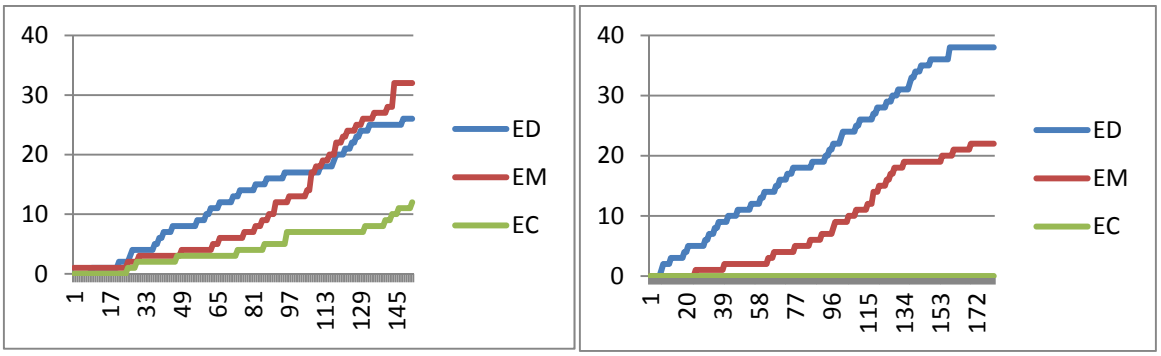

Figure 12- The Final Play of a Beginner vs Expert's First Play (stage 1)

This is a good indicator of player skill and should be part of what is used to determine when to adjust difficulty. The more the player's graph begins to resemble that of the expert's, the farther they get in the level and vice versa.

Power-ups:

These have a huge influence over how the players performance, and one of the major factors of improvement in beginners. As mentioned previously, the life power-ups greatly affect how far they get in the stage, and the shield power-ups increase. As seen in Figure 13 beginner's increase in skill is also accompanied with an increase in power-up acquisition.

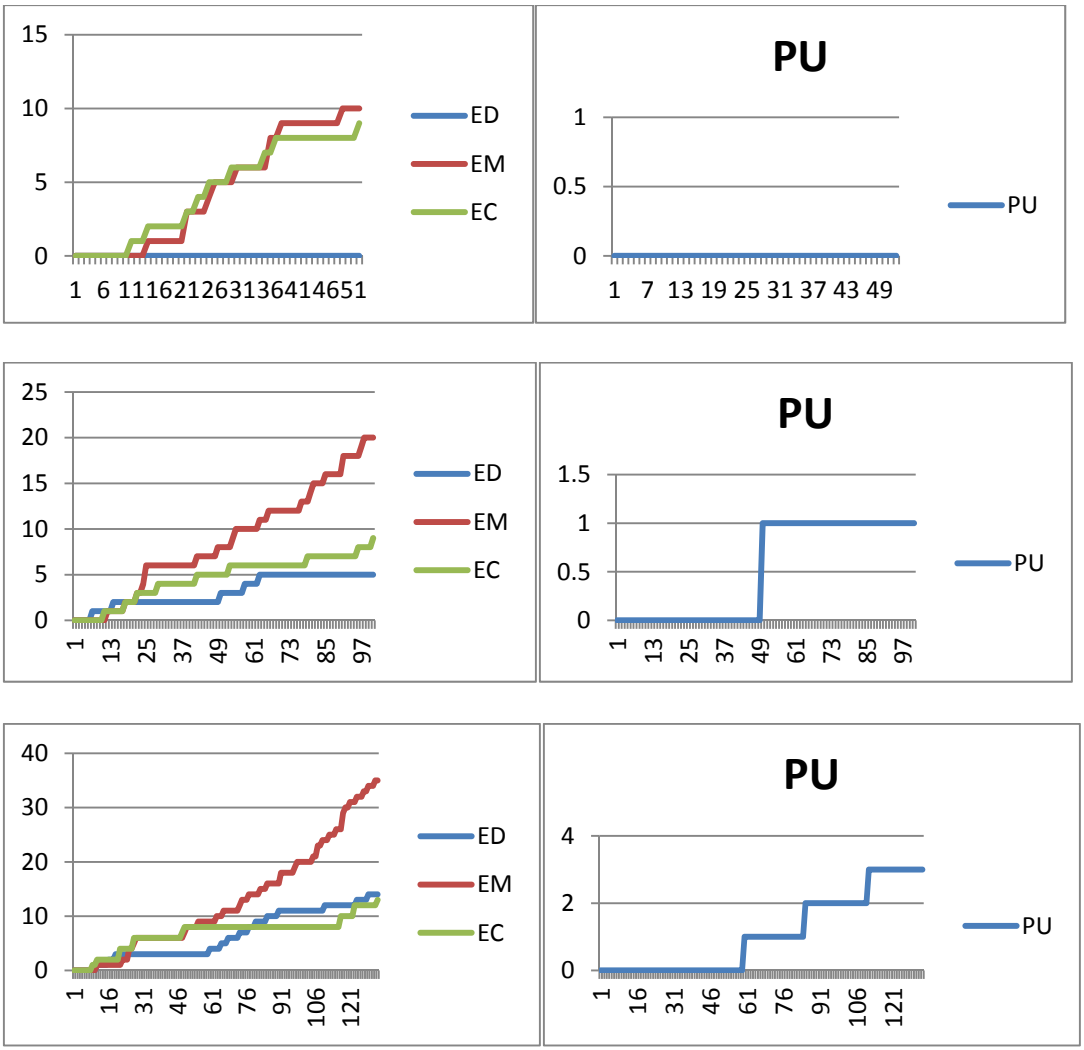

Figure 13- Player Skill vs Power-up Acquisition

Since skill variables and power-ups are related, the frequency of power-up generation can definitely be used to make graphs converge to a target.

After game interview:

Video games are an interactive experience and thus we cannot forget to see how player's feelings match up with the data taken on them. Interesting enough it the players with the most complaints turned out to be in the middle of the skill range as opposed to the two outward extremes.

Those who never played games before ended up progressing the fastest, and when interviewed they wanted to continue playing because they were just starting to get the hang of things and were satisfied with their progress they were making.

On the other hand the skilled players were pleased with how much the game ramped up, and found the easier stages an opportunity to strive for other goals like no-damage runs or avoiding power-ups. Even when things ramped up to their limit they never wanted to give up, in fact they got more determined to beat it because to them it is an issue of pride.

The players in the middle though lack the rapid improvement beginners experience but lack the desire to keep playing once they hit their limit. Most of those tested falling in this range decided to quit after losing consecutively on a stage, and when asked why answered along the lines of even if they could get through the part they're stuck on it just meant they'd have to play an even harder stage after that.

RELATED VIDEO GAMES

As previously discussed, there are many different approaches to implement dynamic game difficulty adjustment. In all cases, it is necessary to measure, implicitly or explicitly, the perceived difficulty the user is facing at a given moment or stage in the game. This measure can be performed by a heuristic function like the ones mentioned before in the design portion. This function maps a given game state into a value that specifies how easy or difficult the game feels to the user at a specific moment. Once the difficulty to a player is determined, it may be deemed necessary to adjust the game environment settings in order to make challenges easier or harder.

Here are some of the varied uses of Dynamic Difficulty Adjustment in consumer video games:

Enemy Count:

A simple but very effective technique can be found in the 1999 video game Homeworld. At the beginning of each mission the number of ships that the AI begins with is set depending on how powerful the player's fleet is. Skilled players

will naturally finish with larger fleets since they take fewer losses compared to worse players. This approach creates a very natural difficulty curve whose slope is recalculated after ever mission and skewed to fit the skill of the player. This means the better the player, the quicker the games difficulty will ramp up.

Enemy Generation:

A system implemented in the video game Fallout 3, uses player level to determine what kind of enemies to spawn. The more the player levels up, the harder the types of enemies generated become (ie enemies with higher and higher statistics and better weapons). The extent of this adjustment can be raised or decreased using an in game slider, with bonus experience given to players using the harder settings. This means the better players not only can be challenged more but also are rewarded for being good, allowing them to level up quicker.

AI adjustment:

In Valve's video game Left 4 Dead a new difficulty adjustment system was implemented using an artificial intelligence technology called "The AI Director". Its goal is not only to adjust the difficulty based on player's performance but also to create a different experience for them each time the game is played. It does this by monitoring individual player's performance, as well as how well they work together with their teammates, and uses this information to determine what and where it will spawn next to keep gameplay continually unique and challenging. This means instead of just ramping up the difficulty to the players skill level, it takes into account how players fared against everything so far and uses this information to add new events that will move the narrative in the direction it wants to take them. To accomplish this it also adjusts the visual and audio cues to set whatever mood it wants to convey at the moment or to draw players' attention to where they want them to go.

Fine tuning from a set difficulty scale:

The game Resident Evil 5 employed a hidden adjustment system called the "Difficulty Scale". The scale is locked at a particular range based on the difficulty they choose, and can move up and down this range based on player performance. This system uses the tried and tested system of locking difficulty at preset values, but adds the fine tuning these systems most desperately need to create a better difficulty fit for all types of players.

Failure based:

Another type of adjustment that happens in-between the game action can be seen in the match-3 game Fishdom. At the end of each level the time limit is simply adjusted based on how well the player performed. Every time the player fails a level, the time limit is increased making it beatable for anybody if they lose enough.

In super Mario 3d land if the player dies too many consecutive times on the same level the game spawns a special power-up at the beginning for them to use if they feel like it. While this simple technique can help players from becoming too frustrated and quitting it also makes the level too easy and feels like a free pass instead of little boost to try to give the player just a little more edge.

Negative Feedback loop:

Anyone who's played the Mario Kart series will quickly realize the blaring negative feedback loop tied to the item system. This feedback loop is a form of dynamic difficulty adjustment used to help drivers who are further back get ahead of their opponents by giving them better items. This means the further back a player is the more likely they are to get an item that will drastically influence the tide of the game whether it sharply ups their speed, lowers the speed of all their opponents, or even the controversial "blue shell" which targets the first place person directly thus enforcing the negative feedback loop even more. This means drivers in first or second place can expect to get nothing but weaker items while being bombarded with stronger items from behind.

The drawback to using feedback loops in real time is the previously mentioned "rubber band effect", since it continually pushes the worse players forward while punishing the better players. Luckily the adjustment is done just enough to still make it possible for skilled players to win, but can lead to frustration for them because winning is no longer an issue of simply driving well, but also dependent on what items the players behind them get (ie how often players get blue shells and when). Nothing is more frustrating than dominating opponents for an entire race just to be blue shelled from behind right at the finish line causing other players to pass at the last second.

Conclusion

After taking into account the data and the after game interviews a general consensus can be found on how adjustment should ideally be carried out. First off, players unanimously enjoy being challenged. The big difference between skill levels though is the amount of challenge they can handle.

Beginners don't want to be babied because they enjoy watching their improvement, but if we ever want them to progress through the entire game we will need to slightly tweak the difficulty down until they can get through a level. Average players' difficulty can generally be left alone but their amount of consecutive deaths should be observed in order to keep them from hitting a wall and giving up. Skilled players generally seem to be pleased as long as there is some time of curve that ramps up over time, using the beginning levels to show off their prowess and get used to the mechanics in preparation for the harder stages to come.

These results all point to a system were difficulty is adjustment is only applied after the game deems the player has died enough to begin slightly decreasing difficulty in a way the player doesn't notice. This way when they hit something too difficult for them it will still require at least few retries in order to get past it without leaving them feeling stuck at any particular part for a frustrating amount of time. Everybody wants to feel like they're progressing through the game, but at a challenging pace, and nobody wanted to be babied despite the fact they hated be stuck on a part. The goal then will be to adjust just slightly enough to allow them access to the next parts of the game if they play enough, without letting them notice the game is purposely lowering the difficulty for them to win. This should produce the greatest sense of accomplishment for all types of players with minimum frustration.

Future Work:

Going off of the idea of adjusting difficulty after a certain amount of deaths, a difficulty curve must be generated to verify what each level of players skill looks like. This can be done using the average amount of lives spent on a stage for each class of player. Since players only had 10 lives to spend all of the graphs max out as soon as they hit that level, but who knows what kind of graphs we would see if we let the players play it however many times it took them to beat the game.

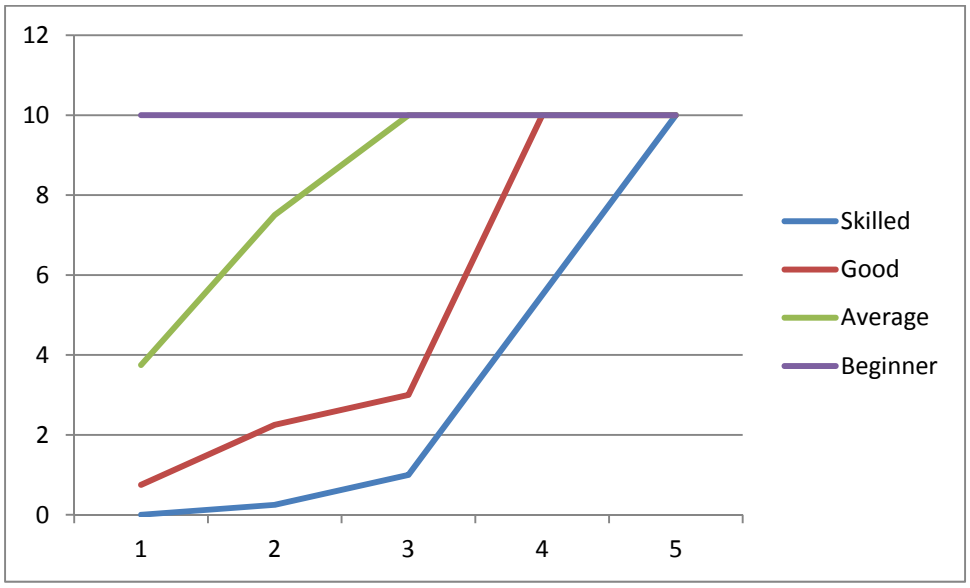

Figure 14- Average Values of Death Counts

Based on the graph in Figure 14 we now have a visual representation of the average values of each skill level. All of them increase as the level increases, but cap out at various points in the game. Ideally we want everyone to have roughly the same death count at each point in the game, giving them a feeling of the difficulty ramping up harder and harder, but with them able to overcome each stage with hard work and perseverance.

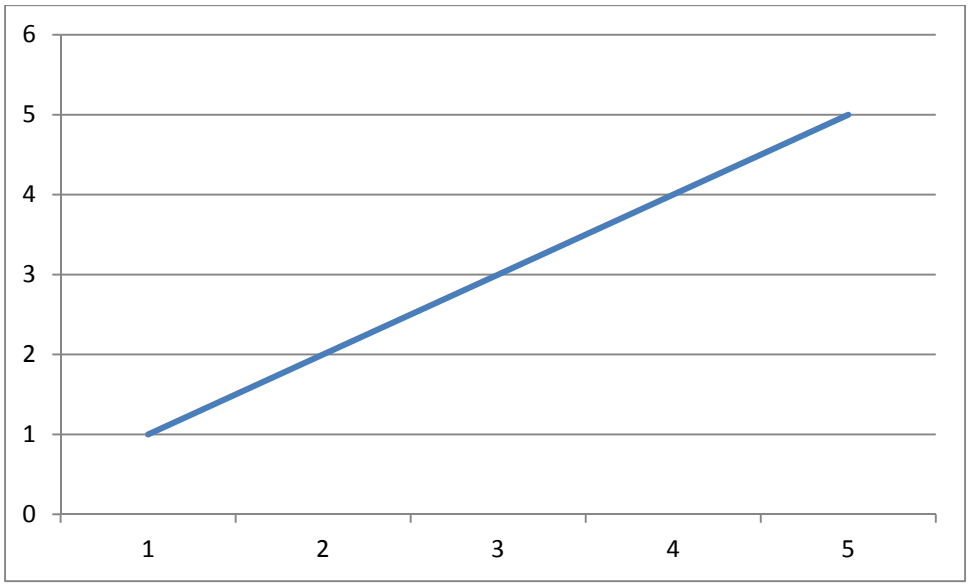

Figure 15- Ideal difficulty ramp.

The actual difficulty increase or decrease can be accomplished using a combination of all the factors mentioned previously. When players have died a sufficient amount of times to give them the ideal sense of difficulty the given stage should be at, then all of the difficulty variables can be slightly adjusted to where the shift isn't visibly noticeable. Power-ups can be generated more frequently to give players that extra boost to get them through the stage, enemies' stats can be raised or lowered (like fire rate, life, spawn rate, speed or even bullet speed), and player's stats can be tweaked as well (power, speed, bullet speed or even how fast the boost meter recovers/depletes).

Adding Visually Noticeable Adjustment:

While not yet implemented in the project, there are visually noticeable types of adjustments that also must be tested. As discussed in the design portion, the game used for testing has a hardcoded level structure that dictates what enemy will be spawned when. In order for level consistency this predetermined level structure should remain intact. Instead the game can throw in extra enemies or breaks it thinks it should add based on player skill in place of other things generated. This can be accomplished by inserting a timer function in the Stage Generator class that will sometimes insert what it thinks will change the difficulty curve in the direction it wants it to go. If the game needs to increase the difficulty it will try to insert what it thinks will be most challenging to the player. This will take into account strategies used by the player (using their skill variables) so it can force the player into situations where they can't simply find a single degenerate strategy and stick with it (for example if they found a safe spot in the game it will notice they aren't moving and spawn an enemy to go to that spot). If the game needs to decrease difficulty it can replace enemies with breaks or easier types of enemies.

Final Notes:

The ideal amount at which these should be tweaked would require additional test phases to determine if these changes can really make the player's skill data all converge to a similar graph. More interviews would also have to be conducted to see if players start to catch on to the adjustments and if this realization would break their flow or cause them to not feel as accomplished for beating a level. The hardest part of this process will also be how much to adjust in the beginning, since beginners are the farthest away from the ideal difficulty graph but their skill also progresses the fastest.

All in all, the main concern when conducting all of these tests is to always keep in mind "will this make the game more enjoyable for all players?" Games after

all, are for enjoyment and if any of the changes made to gameplay is causing people to break flow and lose enjoyment of the experience then the changes should be implemented differently. This is why so much play testing is needed in the development of a game and why we still have so much to learn about what each type of player will bring to the table.