Christopher Lard

Senior Project Paper

Dr. Jackson

A Modern Classic: Programming and Design

With this project, I have set out to create a game that is as reminiscent of classic video games as it is modern in its design sensibilities, programming techniques, and elements of gameplay. The game's graphical style and the foundation of its gameplay are derivatives of video games from the 70's, 80's and 90's, but hopefully many other aspects of the game will reflect a modern rethinking of how a game is created, shared, played and designed. From a very high level, this game is a platformer, a game in which the player character's primary form of action is jumping, styled after classic titles of the genre (e.g. *Super Mario Bros., Megaman, Metroid, Castlevania*, etc.) intended for an arcade-like setting.

One of the first decisions I made for the game was its visual style. This was partly dictated by the environment used to create the game. Flixel, the Actionscript 3.0 library that the game is built on, provides very good support for retro style "pixel" art. Pixel art is a subcategory of computer art in which the value of each pixel in a picture is carefully placed by hand with a focus on controlling the visual space as tightly as possible. While the environment does have good support for the kind of graphics I chose, the main reason I picked this style is because I like it. The graphical style of a game has many direct implications to the gameplay that should not be overlooked.

The clarity that is afforded by limiting a game to two dimensions is unrivaled by "three dimensional" games. I say "three dimensional" because these games obviously are not three-dimensional, they merely use mathematical tricks to render things that appear to be three-dimensional. This is ineffective for a couple of reasons: 1) three dimensions are most useful because of our ability to focus on close up and then far away, but three dimensional games don't allow us to change our focal depth; 2)

navigating pseudo three dimensional terrain with a clumsy video game controller is much less natural than navigating left, right, up and down in a two dimensional game.

Another aspect of the visual style of the game is the use of the original Gameboy's color palette, which is composed of four shades, all tinted a slightly green/yellow, from lightest (almost white) to darkest (almost black). This is of particular importance on this project because I am not a graphic artist and lack all but the most basic training in color theory; thus, I am using a nostalgic grayscale, as seen in fig. 1, to limit the amount of work I have to do on the coloration of my graphical designs.

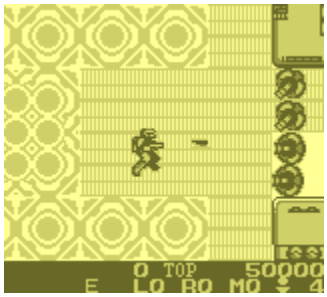

Fig 1: Burai Fighter Deluxe on the GameBoy

As mentioned briefly before, this game is a platform game (or platformer) styled after classic titles that created and defined this genre. A platformer is a game whose focus is on navigating terrain, in this case two dimensional, covered with obstacles like pits, spikes and enemies, with a focus on jumping to avoid and overcome these hazards. Many other mechanics such as running, shooting, hitting, swinging on ropes, and bouncing are often incorporated to broaden the gameplay. In this game, the player has the ability to jump and shoot. The player uses these abilities to navigate from the beginning of the level to that level's exit in order to proceed to the next stage. Points are awarded to the player for every pixel upwards that they move, and points are subtracted for every pixel they fall down. The goal of this project was to create a platform game that incorporates some procedural generation aspects in its level generation. In this I was partly successful.

I have created a platform game that incorporates aspects of procedural generation, but not to the extent that I had hoped. However, upon learning that in the

history of video games there has not been a single commercially successful platform video game that significantly utilizes procedural level generation this didn't feel like as much of a failure as it might have otherwise. The goal of the game is to ascend as high as possible while losing as little ground as possible. This game's design and goal are inspired by the mid-1980's arcade classic, *Ice Climber*.

Technical Challenges

The primary challenge faced in this project was procedurally generating the level, and then figuring out whether that level could be completed or not. The procedural level generation algorithm itself is necessarily simple. Each level of the game is broken up into level chunks that are each 12x12 tiles (fig. 2). The outside of each level chunk is lined with solid tiles with a single entrance/exit to the next level chunk. Within the 10x10 inner area of a level chunk, 6 to 14 blocks of varying widths are placed randomly throughout with a few placement restrictions (e.g. a block can't be placed in the same spot as an already existing block). Before the level is tested for reachability, the starting point and exit of each level chunk are always cleared if a block was placed over either of them, and a block is always placed in such a way that there will be a way to get from this level chunk to the next level chunk. In fig. 2, the tiles labeled "1" and "2" are the tiles that are cleared if a block is placed on them because that would make it impossible to get from the current level chunk to the next. The tile labeled "3" is one that is guaranteed to be occupied by a solid block because it guarantees passage to the next level chunk if the player character can make it to "3".

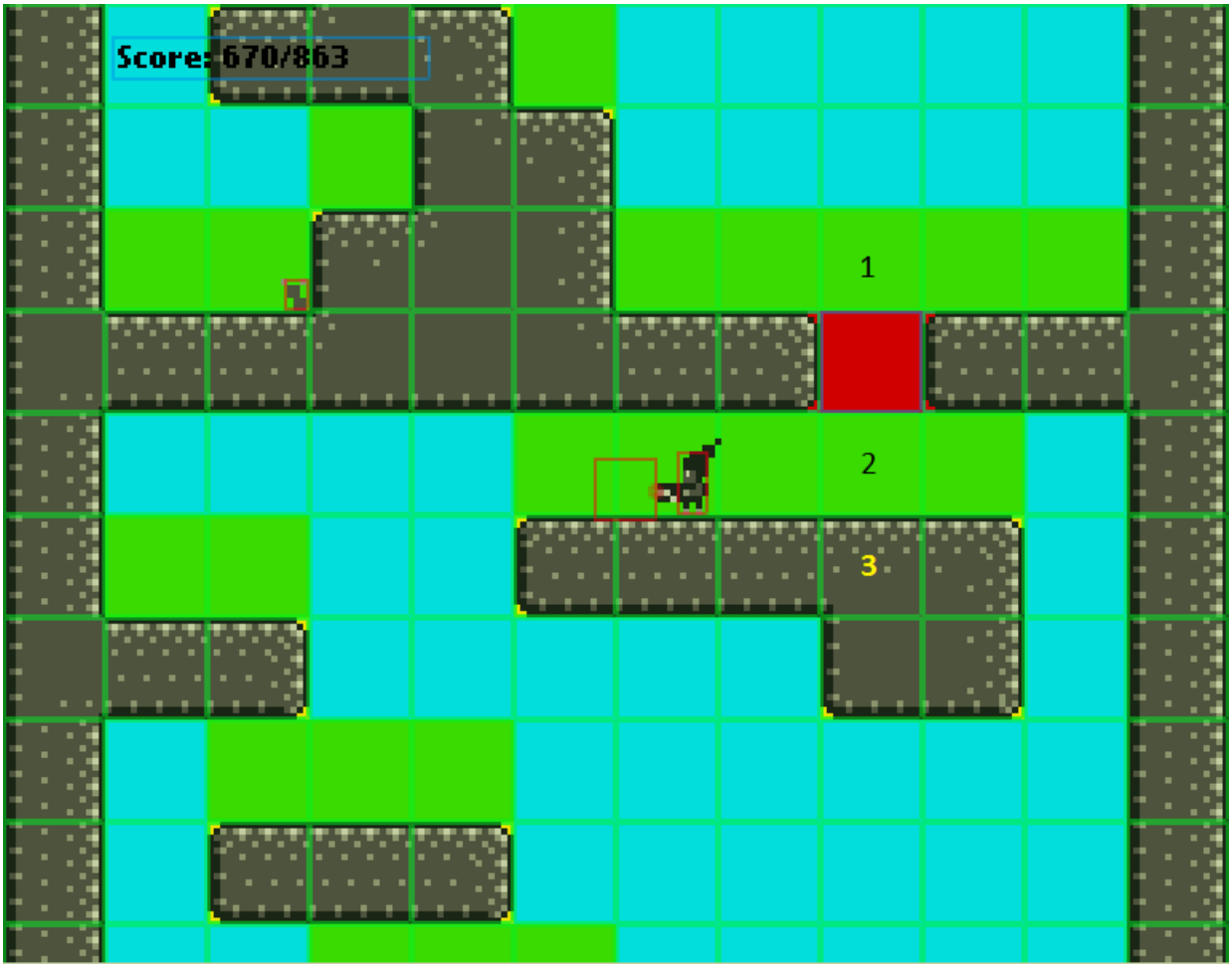

Fig. 2: a screen shot from the game showing the tile grid

Once these steps are completed for a single level chunk, that level chunk is tested to see if the player character (p.c.) can get from the entrance to the exit. This was the most difficult part of the game to write. First, the range of motion of the p.c. was checked. I ran simulations of the player character running and jumping in a stripped down version of the game's environment. These simulations ran on top of a grid of tiles that were marked as the p.c. overlapped each tile. The p.c. jumped at varying levels of jump strength to get a wide range of possible movements. Whenever a tile was overlapped it would be recorded. Thus, a single jump in the simulated environment would produce an ordered set of tiles overlapped during that jump. This set of tiles is then representative of a single "move" that the p.c. can execute. An example set that was produced is [0,1],[1,1],[1,2],[2,2],[2,1],[3,1],[3,0]. This is a set of the x,y coordinates in order of overlap during one jump executed by the p.c. They can

be interpreted as a path of movement such that if the tile that if the tile that is 1 unit above ([0,1]) the current position is empty then that is a spot the p.c. can get to, and if that spot was valid then the p.c. can get to a tile 1 unit to the right of that ([1,1]) provided that tile is empty.

All of these “moves” were then recorded in an array, and the redundant ones were eliminated. The algorithm to actually determine if the p.c. could get from the beginning of the level chunk to the end of the level chunk uses this array of moves to map out all of the positions the p.c. can get to in the current level chunk. The algorithm is given a 2 dimensional array representing a 12x12 level chunk, a start point, and an end point. The start point is pushed onto the queue that holds all of the positions from which the p.c. can execute a jump (henceforth, the “jumpQueue”). Now a loop is started that iterates over all of the positions in the jumpQueue, and, for each position, all of the “moves” that the p.c. can execute are simulated from that position. As each move is executed, all of the positions reached from the current are marked on a second 12x12 array. If the position is one from which the p.c. can execute another jump then it is pushed onto the jumpQueue otherwise it is just marked as a reachable position. This process is repeated until the jumpQueue is empty, and then the second array is checked to see if the p.c. was able to reach the exit point of the level. If the exit is reachable then the next level chunk is generated and tested, otherwise the current chunk is cleared and regenerated.

Pseudo code:

```
testReachability(mapArray, startPoint, endPoint) {
        push startPoint onto jumpQueue

        while jumpQueue is not empty
        {
                or all of the jumps the playerCharacter can execute
                {
                        execute the next jump
                        mark every space on the mapArray that is reached by this jump

                        if the space reached is a space from which another jump can be executed
                                push space onto the jumpQueue
                }
        }

        if the endPoint can be reached
                create the next level chunk or end execution
        else
                clear the current chunk and regenerate it
}
```

The main change that would be beneficial to make to this algorithm is making it general enough that it could be used as a plug-and-play module in any setting in which someone needs to calculate reachability in a two dimensional environment under a certain set of physical constraints. The main difficulty here would be automatically running the simulations, and then deriving the set of "moves" from these. Optimizing this set of moves and eliminating the redundant ones would be difficult to standardize. Many of the moves that come out of the simulation are not moves that would ever be used in any reasonable play of the game, and sometimes they are moves that don't make any kind of positive progress.

Game Design

It is only in recent years that game design has come into its own as a field of study. Programs in game design are emerging in universities across the country, as well as schools that specialize in design and development. Although games have been around since man first threw a rock or a stick, game design has been a somewhat ad hoc and mysterious process done by nameless individuals. With the advent of digital games, game consumption has increased to the point that, in 2009, the video game industry had $10,500,000,000 in revenue (*Video Game Industry Statistics)*. So, not only is the process of designing good games more important because of its huge number of consumers, but it is also more profitable than any time in history.

In my investigation of game design, I immediately wanted to know how the term "game" was defined. There are a number of definitions for game, and I will start, like everyone else, with Webster's: "[a game is an] activity engaged in for diversion or amusement" (Webster). While this is correct, and doesn't do a terrible job of explaining what a game is, it doesn't tell the whole story. Certainly games are activities, and they are usually engaged in for diversion or amusement, but that only tells us the category games fall under and why people play them without actually telling us what a game is at all. Fortunately, there is a whole group of game design academics who are very concerned with what a game is, and they have many ideas to draw from. One prominent academic in the field is Ian Schreiber. His definition of a game, which has

been adopted by the International Game Developers Association (IGDA), is “a play activity with rules that involves conflict.” (Schrieber)

This definition points out a few, very important, aspects of games. First, games are a form of play. People play to have fun, but not everything people play is a game. I play catch, with PLAY-DOH, and even in the mud. These are all things that are done for fun, but they do not have all of the requisite components of a game. Games also require a set of rules that inform the players how to start playing, continue playing, and come to a resolution. Consider a game like the childhood favorite “tag.” Rules are provided for each of these categories.

Setup

- Choose a player that will be “it”
- Define a play area
- Spread the players out within the confines of the play area

Progression

- Players must stay within the confines of the play area
- Players may not immediately re-tag the player that just tagged them
- Players may elude the player that is “it” by foot without external assistance
- Play may continue until a time at which the players decide to discontinue play

Resolution

- That player that is “it” pursues any of the non-“it” players until another player is tagged
- Upon being tagged by the player that is “it”, the tagged player becomes “it”
- Upon tagging another player, the player who was “it” loses their status as “it”

These rules could be elaborated upon to no end with caveats like “players may not employ the use of trained animals in an attempt to catch other players” or “players may not use any sort of constructive or destructive implements to alter the play area.” These are largely unnecessary because rules like these are implicit in the game due to

previous play experience. The necessity of defining rules as explicitly as possible is easy to understand upon witnessing a group of children on the playground arguing over the virtues of playing with or without "tag-backs." Fortunately, when creating digital games, the entire universe of play is designed with rules. The players are only allowed to act in ways that the game designer allows them to within the confines of a universe that the game designer created. The need to specify that no projectile weapons may be used in a digital game of tag is implicit in the universe of the game if no projectile weapons are provided.

The primary weakness of tag is its resolution conditions. Because there is no "win state" in tag, should play continue only until the players no longer want to play? The lack of precise resolution conditions leads to play in which players discontinue only when they are tired of playing. Ensuring that players get tired of your game and *want* to stop playing it each time they engage is probably not the best tactic to ensure a loyal following. Conversely, tag makes excellent use of conflict, which is another key component of games. Tag embodies conflict in one of its purest forms. It pits every single player against all of the other players as they take on alternating roles as the predator and the prey. This type of play emulates the mechanics of human survival allowing the players a chance to both hunt and run in order to survive.

In tag, the players have a limited set of actions which they can take to both capture and elude capture. Likewise, humans have an implicit set of actions which they are capable of executing imposed on them by nature. In games, the set of actions from which a player can draw are called mechanics. Tag allows the players to run, tag players, and become "it". In video games, the "physical" mechanics are tied to buttons. *Super Mario Bros.* presents the player with the options of moving left and right, ducking, jumping, running, and shooting fire balls. Other, indirect mechanics can be observed as a result of the player's actions like getting points for defeating an enemy. Mechanics are key to game design because they define how the player interacts with the universe of the game. Game designers spend most of their time defining and fine-tuning these mechanics in order to make their games more fun and interesting.

Tag is a wonderful example of basic game design precisely because of its primal nature, which forces the player to make decisions in an attempt to prolong survival. This brings up another core component of games which is the importance of decision making. In tag, the decisions the player makes are sometimes trivial and automatic, like choosing which direction to run. Depending on the area of play, decisions could become somewhat more complicated if the player has to choose which obstacle to take cover behind. More strategically complicated decisions can arise depending on the aptitude of the player. "Should I run near other players or not?" "Will speaking to the player that is 'it' help or hurt my chances of survival?" These kinds of questions illustrate the psychology that players can choose to employ when facing other human players. These choices in the types of strategies employed by players are entirely outside of the control of the game designer, but are based on the limitations imposed by the rules of the game.

LeBlanc, Hunicke, and Zabek make an important observation regarding precisely this issue in their work on the idea of "Mechanics, Dynamics, and Aesthetics" (MDA). In their work, they describe the experience of playing a game, and how this experience is related to the game's design. Mechanics are defined as the rules the player must observe, dynamics as the strategies of play that emerge from the mechanics, and aesthetics as the experience that the player has of the game. This illustrates the "twice removed" nature of the game designer's relationship to the player's experience of the game. Two players could go about using the mechanics provided by the game in any way that the universe of the game allows. Differences in physical skill and coordination can also greatly change a person's experience of a video game, but coordination or lack thereof hardly needs to be taken into consideration when watching a movie or looking at a piece of sculpture. This difference arises from the interactivity afforded by games that is entirely absent from other art forms.
(http://www.cs.northwestern.edu/~hunicke/MDA.pdf)

Procedural Generation

Procedural content generation is a method by which content (in video games or otherwise) can be algorithmically generated across a virtual space. In video games

specifically, the goal is to define an algorithm that can create interesting and highly varying play spaces based on a few randomized parameters. These parameters cannot be truly “random” and are usually pseudo-randomly generated by a computer using a linear congruential generator or something similar. The goal of procedural generation mechanisms in games is to provide the player with a larger number of play spaces than any team of designers would be able to produce by hand. There is, of course, a tradeoff here between the complexity provided by handmade content and the “infinite” number of possibilities that procedural generation provides.

One of the first games to use procedural content generation was the computer game *Rogue.* *Rogue* creates its levels by “randomly” placing rectangular rooms throughout a rectangular play space and then creating hallways between those rooms. The generated level is then populated with enemies and loot appropriate to the depth of that particular level (in *Rogue* the depth of the level determines its difficulty). The generated levels are somewhat repetitive and almost boring if examined purely on a level design basis (fig. 3), but they are perfectly suited to the game’s style of play. Unlike many games, in *Rogue*, the player almost always loses except for the few expert players who win the majority of games they play. It would be incredibly tedious to play *Rogue* over and over again if the levels were exactly the same every time, but because they are procedurally generated the game stays interesting far past the first time it’s played. *Rogue* in particular emphasizes learning the system of the game over the particulars of any particular level. While a game like *Super Mario Bros.* is a masterpiece in the field of game design, no one would argue that a large part of mastering the game is memorizing each individual level.

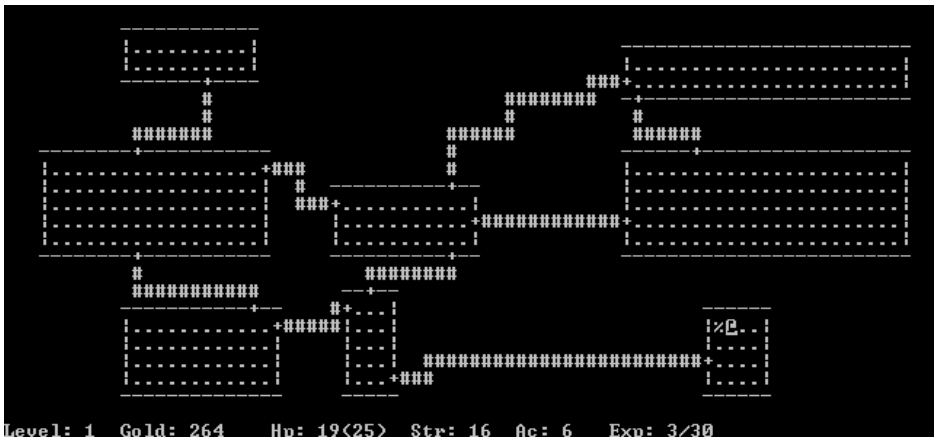

Fig. 3: a screen shot from *Rogue*

In my years of playing video games, there has been a glaring lack of procedurally generated platform games. Why should platformers be left out when so many other genres are benefitting from procedural generation? It turns out that there are excellent reasons why this hasn't been done yet; the foremost reason being that it is very, very difficult. Platform games face a p articular challenge because very small changes in a level can make a level impossible to traverse. For example, if the player character has a maximum jumping range of 100 units and the level generator makes a gap that is 101 units long then that gap is impossible to overcome. Let's assume that the algorithm accounts for that possibility, and will only ever create gaps that are 100 units or less. What if the game has something like spikes that cause the player to lose life if they are contacted? The level generator now creates a gap that is 80 units long but places 20 units of spikes (assuming the spikes have some vertical dimension) on the area where the player character needs to land after clearing the gap. This has created another impossible situation. The more elements and variation provided by the level generator the more cases have to be protected against to ensure playability.

One of the only platform games that makes use of procedural level generation is a recently released computer title, *Spelunky*. *Spelunky* takes a necessarily simple approach to procedural level generation to ensure that each level can be completed. Each level consists of a 4x4 grid of level chunks. Each one of these chunks is a handcrafted mini-level that will always have certain entrances and exits. The procedural generation algorithm takes this into account so that there is always at least one path

from start to finish. Furthermore, each mini-level is created in such a way that it will not always be exactly the same. Derek Yu, *Spelunky*'s creator, accomplishes this by designing the mini-levels as tile grids where the number in each grid segment represents the probability that that particular tile will be solid or not. For example:

```
A A A A A A A A
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 6 6 A A 6 0 0
0 0 0 0 0 0 0 0
A A A 0 0 A A A
```

A grid like this could represent a level that is 8 tiles wide, 6 tiles tall, has a solid ceiling, a floor that is solid on the right and left with a two tile gap in the middle, and a platform in the middle of the room that has two solid tiles surrounded by three tiles that each have a 60% chance of being solid. This method creates a wide variety of levels, but, after playing the game for a few hours, a watchful eye will start to notice the repetition of these level chunks. The part of the game that players can't learn is the placement of loot and enemies because these are placed completely randomly at the generation of each level. While this approach is effective and efficient it doesn't completely procedurally generate every level or really test the limits of procedural generation in platform games.

Conclusions

Creating a complete game involves much more design, planning, and resources than I completely understood upon taking on this project. Despite this I think I have successfully produced a very small game that incorporates interesting aspects of both game and algorithm design. Procedural level and content generation can be successfully employed within platform games, and doing so can increase the overall play space game designers can provide to their players.

Works Cited

Video Game Industry Statistics, Entertainment Software Rating Board, 2009. Web Image. 20 Nov. 2011. < http://www.esrb.org/about/video-game-industry-statistics.jsp>

Schrieber, Ian. Game Design Concepts. 2009. Web Course. Oct. – Dec. 2011. <http://gamedesignconcepts.wordpress.com/2009/06/29/level-1-overview-what-is-a-game/>

Hunicke, Robin, Marc LeBlanc, and Robert Zubek. “MDA: A Formal Approach to Game Design and Game Research“ 2004. <http://www.cs.northwestern.edu/~hunicke/MDA.pdf>