Pitfall: Final Project Report

David Lacayo

Introduction:

For my final project in ECE 385, I decided to recreate an old 80's game on the *Atari 2600* called "Pitfall". In order to accomplish this, I started from lab 6.2's software and hardware, and worked from there to create a game environment, game logic, and user interface to actually play the game. Most of the project was completed using System Verilog, outside of the PIO's and C code for interfacing with the USB keyboard (SPI protocol through AVL bus). Since this was a video game, I also wrote code that converted the PNG's of the obstacles and characters in the game to sprites so that I could access them via a bitmap. This code, along with code that generated a color palette from hex, was written in python, and took inspiration from resources online that are listed in the "resources" section at the end of the article.

Figure 1: Atari's "Pitfall" (1982)

Background:

Before I get more into the details of how I recreated the basic components of Pitfall, it is important to understand how the game works, and what components I decided to include. Pitfall follows a character named "Harry" through the jungle in the search for 32 treasures while avoiding obstacles like pits, snakes, crocodiles and logs. The player is given 20 minutes, 20,000 points and 2 lives to try to find all 32 treasures without running out of lives. If the player runs into a log, points are deducted. If a player finds a treasure, points are added. If a player runs into any other obstacle, a life is deducted. If the player runs out of lives, collects all the treasures, or runs out of time, the game is over. Relatively simple, but a hell of a lot of fun. There is no secret why Pitfall was the #1 selling game from 1982-1983, and was a staple for a lot of people's

childhoods (my Dad included in that). However, unlike developers today, 70's and 80's developers had a LOT less hardware to work with. My god, I swear these guys were geniuses. The creator of the game Pitfall, David Crane, was tasked with creating the game on the Atari 2600, a console with only **128 bytes of RAM**. Because of this, other games that were coming out at the time had very few levels. For example, Pacman had a small number, and other games just included random events or short gameplay. What made Pitfall incredible and ahead of its time is the fact that it had 255– that's right– 255 unique levels. Crane and his team accomplished this by using combinational XOR logic on an 8-bit register that would contain the level data. The binary inside of the 8-bit register would then tell the game logic what each level would contain. Bits 0-2, 3-5, and 6-7 all stood for different aspects of each level, and because of this, Crane was able to create a game with (2^8)-1 = 255 unique levels. Pretty incredible if you ask me. One byte of memory was used to essentially create the entire environment logic for the game. The link to this article that dissects how Pitfall generates its levels is also linked in the resources section below. The article also gets more into the scoring of the game as well.

For me, it was difficult to figure out which components I wanted to include. I knew I could not do the entire game alone– it would be an impossible task. So, I decided to take the core components and implement them on the DE10 Lite FPGA. For my recreation of Pitfall I included the following components:

- Randomly generated levels
- Harry– the main character who interacts with the environment. Is able to move left/right and jump over obstacles. If Harry goes off the right side of the screen, he will move on to the next level, and essentially a new level must be generated.
- Obstacles– generated randomly from the contents of the current level register
    - Stationary and rolling logs– anywhere from 1-3, and if harry collides with them, he will be reset to the starting point of the current level
    - Holes– 1-3, and if Harry falls in he is reset to the starting point of the current level.
    - Snake- if Harry collides with the snake, he will be reset to the starting point of the current level
    - Fire– if Harry collides with the snake, he will be reset to the starting point of the current level
- Treasure– if Harry collects all 32 treasures, he will end the game.

Essentially, instead of having a score counter, I have a treasure counter, where Harry must collect 32 treasures to beat the game. Harry doesn't have a set number of lives, but he will be reset every time he dies by colliding with an obstacle that is not a treasure. Also, a feature I am a little sad I wasn't able to incorporate was the ability for Harry to play the game both ways. What was great about using combinational logic in the original Pitfall was the fact that you could essentially "revert" back to the previous level without storing it in memory. This was probably the coolest thing from a Computer Engineering Perspective, as you only need memory for the current level and not for 255 levels to run the game. While I wasn't able to accomplish this in the given amount of time, I am going to continue to work on accomplishing this over the summer. It would be a cool feat.

Report:

The first step that I needed to do to complete this project was generate the hardware sprites and the color palette using python code. Since I did not fully understand lab 7.2, I decided that it would be easier to stick to hardware in order to do this, so I decided to write python code that took a color palette represented by hex, and converted it into a verilog module that used a 6 bit code to determine the RGB value for the VGA output. I also wrote code that converted PNGs into palette oriented sprite bitmaps. This made it easy to just go on the internet and grab the PNGs from Pitfall and just throw them into the code. This also allowed me to change the dimensions of the sprites on the go, without having to go in and individually change each pixel. After generating the sprites, and drawing a simple background based on the color coding for the original background, I needed to work on Harry and his movement. Harry is allowed to move right and left, and is able to jump over incoming obstacles. Moving right and left was an easy task, as I just manipulated ball.sv from lab 6 to fit my project.

The jump however, might have been the hardest task in this project. The logic, timing, and orientation took hours of debugging. There was also not great documentation online for jumping characters in Verilog, so I had to come up with all of it on my own, and when I ran into issues, it was just me and my FPGA to figure it out. Once I was finished with Harry, I started orienting the obstacles in Harry's environment. I put logs, holes, a snake and fire in his way, separating them in a way such that even if they were all on screen at once (which is not possible given the way levels are generated), Harry would still be able to comfortably jump over them. Next, I worked on the moving logs, which were a little bit challenging due to verilog just being a stubborn language that will not be flexible with combinational logic. Once I was done with all the obstacles and Harry's movement, I moved on to the treasure part. The treasure is determined by the contents of the current level register, and if Harry interacts with the treasure, it disappears– if he collects all 32, the game ends. Most importantly, if Harry collides with an obstacle (not treasure) his position on the current level is reset. If Harry goes off the right edge of the screen, this is the indication to generate a new level, so the current level register is regenerated, and Harry's position is reset.

Finally, I worked on drawing the actual components. The position of the sprite is determined by the DrawX and DrawY signal from the VGA module. Because of the way I stored the sprites in palletized bitmaps, I did not have to do any additional manipulation. The value at the index [DrawY - object_size_Y][DrawX - object_size_X] contains the correct index of the RGB color that is supposed to be drawn at that pixel. This made lifevery simple, and this is why I wrote the additional python code to generate the palette module and the palletized sprite bitmaps. Since the keyboard interface already existed from lab 6.2, once I compiled, flashed, debugged, and ran the eclipse USB keyboard code on the FPGA, the final product was finished. A lot of work, a lot of debugging, but overall a clean, simple and fun product.

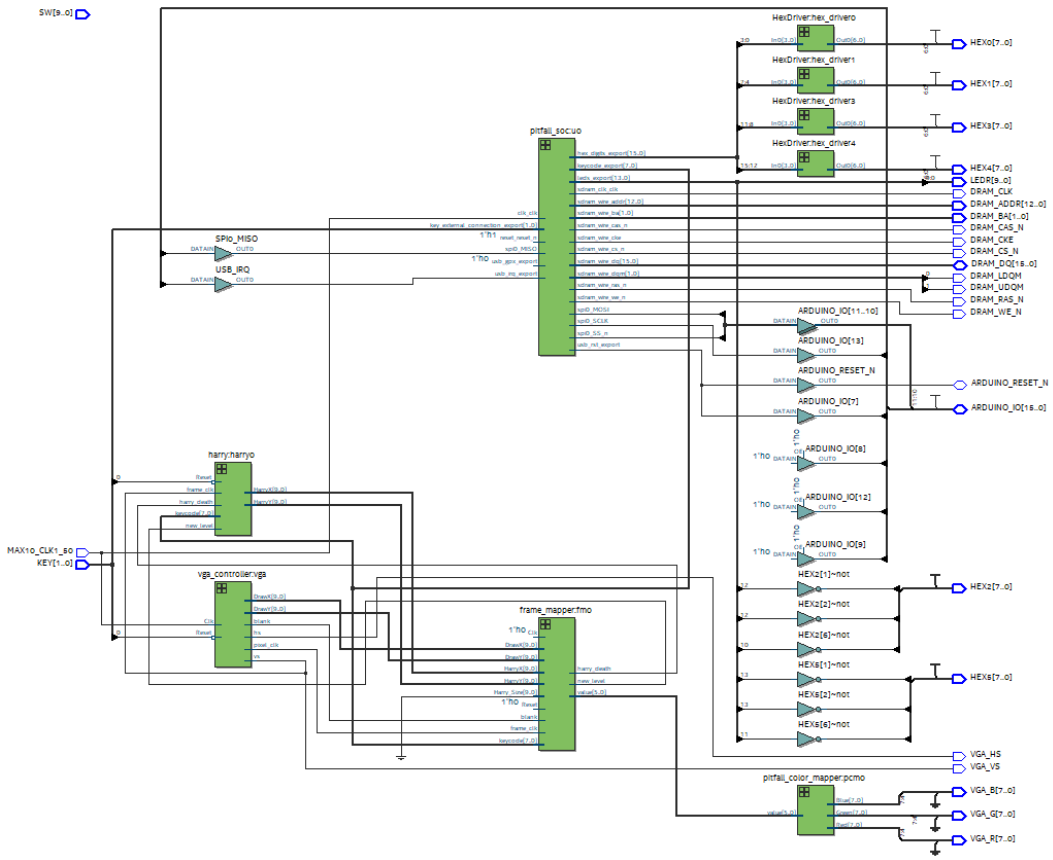

Figure 2: RTL High Level Diagram for Pitfall

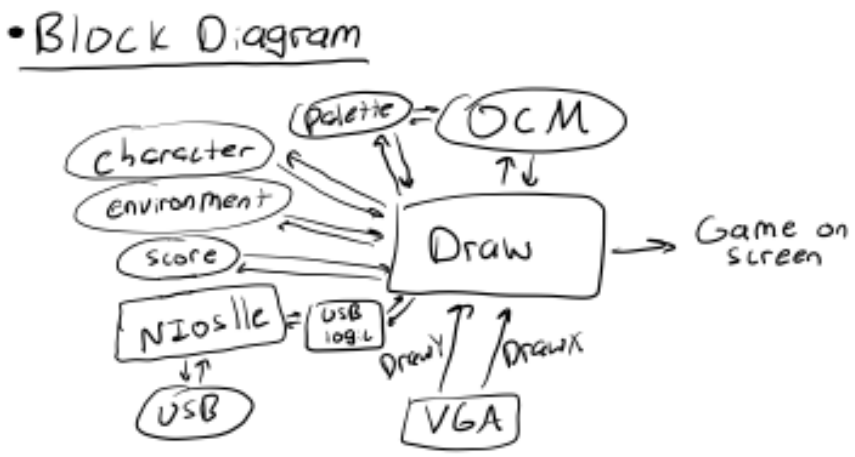

Figure 3: Written Top Level Module Diagram for Pitfall

Hardware Components:

- Clock: clk_0 with a clk and reset export. This is the main clock used for the entire platform designer schematic and runs at 50 MHz.
- Processor: niosII - nios2_gen2_0 connected to our reset and clock, with a data and instruction master signal. This is an efficient processor and not the fully loaded one, but it is enough to get the job done on a relatively low computation lab.
- Onchip_memor2_0 is the on chip memory that we declared. This allowed us to run our program faster, as there was lower access time to memory that is already declared on our chip.
- Sdram & sdram_pll: This is our 16 bit wide SDRAM controller that is connected to our clock and data/instruction master. The pll creates a 1 ns delay across two 50 MHz clock cycles. This is to allow our memory to transfer during the correct window and not miss it.
- Sysid_qsys_0: this is our .qsys connection PIO that is a memory mapped component, which allows us to interface the hardware and software components throughout the entire lab.
- Key: These are the two key buttons on the FPGA board. This allowed us to interface with them in our Eclipse code.
- Jtag_uart_0: This component allows us to debug and interface with our code and hardware by using “printf”. Without it, debugging is much harder, and we are unable to send interrupts to our USB.
- Keycode: this is the usb key that is pressed. This allows us to move the ball on screen
- Usb_irq, usb_rst, usb_gpx: PIOs that allow us to interface with our usb controller
- Hex_digits_pio: allows us to map hex digits from our keyboard to our FPGA interface through the NIOS chip
- Leds_pio: this PIO allows us to interface with the LEDs on our FPGA controller
- Timer_0: This is the interrupt controller that is attached to the USB controller and our NIOS chip that allows us to send interrupts and interface with the clock signals in our USB
- Spi_0: this is our SPI pio, which allows us to send SPI protocol between the NIOS II processor, the USB, and our system verilog code. This allows us to greatly simplify how the code in our MAX3451E file is written. Instead of setting each clock cycle and dataset individually, we are able to call alt_avalon_command() instead, which greatly improves our code.

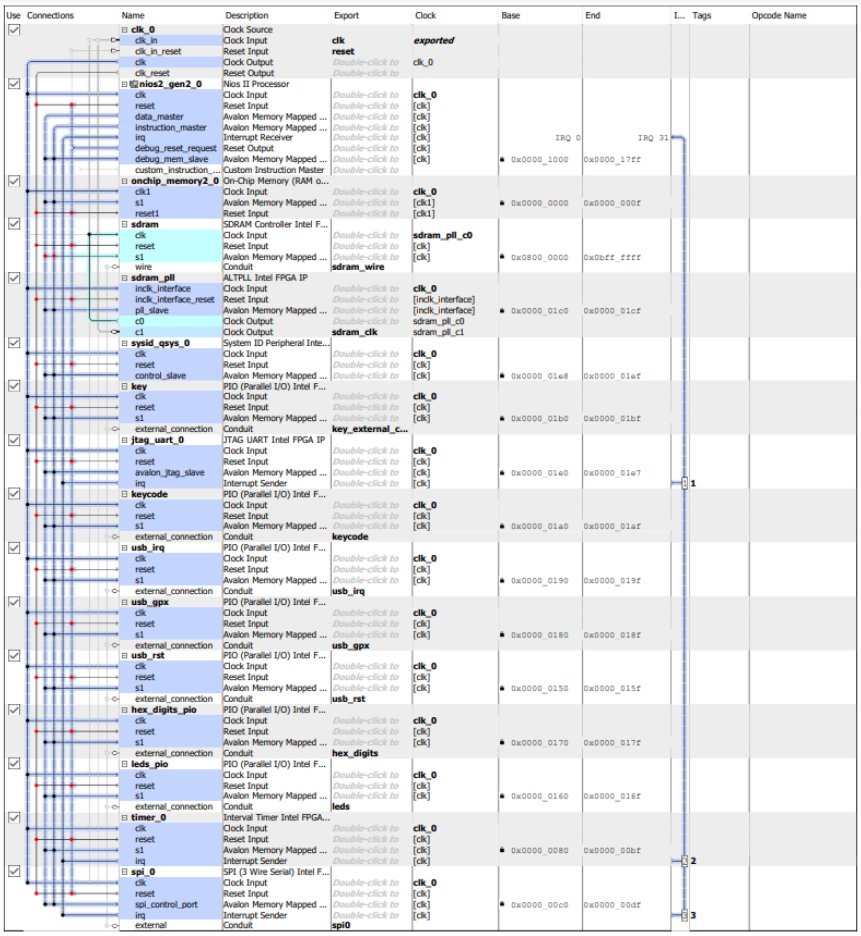

Figure 4: Platform Designer Module for Pitfall

LUT	5,601
DSP	1
BRAM	11,392
Flip Flop	0
Frequency	79.05 MHz
Static Power	96.53 mW
Dynamic Power	63.94 mW
Total Power	182.00 mW

Figure 5: Statistics and Design Sheet

Modules:

Module: Pitfall.sv
Description: This is my top level module that I used to instantiate all other modules and tie all of the blocks together.
Purpose: This top level module allows me to interface with the rest of the program

Module: HexDriver.sv
Description: This module converts 4 bits of input binary to 7 bit hex output
Purpose: I use this module to display the keycode outputs from the USB keyboard on to the FPGA board.

Module: pitfall.sdc
Description: This module contains our clock specifications for this lab, specifically, defining the output delays for our clocks and setting false paths from our modules in platform designer.
Purpose: The purpose for this file is to avoid glitches and to allow our clock to have the right amount of delay to avoid any missing edges (avoid missing the transaction window for memory).

Module: VGA_controller.sv
Description: This module takes in a 50 Mhz and 25 Mhz clock signal and exports the DrawX and DrawY coordinations for our VGA screen. The X & Y represent the horizontal and vertical components of the screen that need to be drawn on with a laser. The system loops through, row by row, column by column, resetting each time it hits the end. This creates the loop that is needed to draw the pixels on the VGA screen. This is also why we need two separate clocks– otherwise there will be errors displaying the pixels on screen.
Purpose: The purpose of this module is to draw the correct pixels row by row, column by column on the VGA screen.

Module: pitfall_color_mapper.sv
Description: This module takes a 6 bit binary input, and exports a RGB value that corresponds to that value that has been inputted. This has been generated and calculated based on the python code written to map a color palette to specific values that can be mapped to a bitmap generated from a sprite.
Purpose: I used this module to basically get the correct value of each sprite at a given location. This makes it easy to pass the correct RGB value to the VGA controller for drawing.
Module: sprite_table.sv
Description: This module contains all the bit maps for the environments that are drawn in our frame_mapper module. That includes the objects: snake, fire, logs, treasure, rat, wall and holes. These bit maps are directly accessed from the frame_mapper module when they need to be drawn on screen using additional logic.

Purpose: This module allows us to store sprites as a bitmap for quick and easy access.

Module: harry_sprite.sv
Description: This module contains the bitmaps for all of the harry movement animation. Not all of them are used– I believe only the default, 2 running bitmaps and the jumping bitmap. I didn't get around to implementing the vines or underground part of pitfall, so the others are just there for reference if I want to continue to work on the harry animation.
Purpose: This allows for quick and easy access of the harry movement sprites via bitmaps

Module: harry.sv
Description: This module is modified from the ball.sv module, and controls all of Harry's movement via w-a-s-d and space bar. Originally, Harry only moves right and left, and can move vertically using the jump movement, but unfortunately, due to the jump bug, I kept the vertical movement in to make Harry playable. If a new level is generated, Harry's position is reset back to the beginning of the screen to start the new level.
Purpose: This module allows the player to control the main character, "Harry's", movement.

Module: frame_mapper.sv
Description: This is the main module of the entire program. It is not the top level, but it contains most of the logic that is used to actually run the game. The collision logic, level logic, and drawing logic is all contained inside of this module. It takes inputs of Harry's position, a 50 & 25 MHz clock, the keycode from the USB interface, and outputs a new level signal and a value that will be used to get the correct RGB hex for the VGA converter. Based on the current level, certain objects will be turned on, and certain objects will be turned off. Harry is always on. If an object is on, it needs to be drawn on screen, which will be determined by the following: if the DrawX and DrawY from the VGA controller is in the margin of the sprite, then we will need to draw the current value at that x-y index. If Harry is also drawn in that x-y position, we have a collision, and if it is not a treasure, I count this as a "Harry death", which resets Harry's position to the beginning of the level. If Harry's position goes off screen, we move on to the next level, and reset Harry's position. If Harry collides with a treasure (represented by a coin sprite), the treasure will disappear, and our treasure counter will increase by 1. If Harry is moving left or right, we will use two different running animations. If Harry is standing still, we use the default Harry sprite. If Harry is jumping, we use the jumping sprite. And that is an overview of the main logic contained in this module.
Purpose: This module contains all collision and sprite drawing logic as well as the level generating logic which controls which obstacles are on or off. This allows us to control the environment of the game, and Harry's interaction with said environment.

Module: log.sv (not used due to bugs)

Description: This module controls the movement of our rolling logs, an object that Harry has to jump over to avoid. It is also a modified module of the ball.sv from lab 6. This module is not used due to bugs, but is kept in for reference in case I want to continue work on it in the future.
Purpose: This module was used to easily control the movement of rolling logs

Module: map.sv (not used due to bugs)
Description: This module controls the shifting of the 8 bit register that controls the level logic. I was not able to include this module due to bugs with generating the level logic. Instead I did the random levels a different way, but I kept this for future use when I try to implement this super cool feature from the original Pitfall.
Purpose: This module allows me to perform shifts and combination logic on the level register to generate random levels.

Conclusion:

In conclusion, I am glad that I put some time and effort into this project. Building a simplified version of Pitfall really allowed me to explore more complex types of graphics on the FPGA, and helped me to explore other sprite table and palette techniques in System Verilog. I would say that the most time consuming part of this project was solely figuring out the color palette and getting the sprites right. Once I did that, it was just about figuring out how to draw the sprites on screen in the right places, collision logic, and finally game logic (like off screen Harry creates a new level). Creating the palette and color logic was so difficult, because I did not originally realize that the VGA adapter on our FPGA is only 12-bit. I was working with a 24 bit palette (directly from the Atari 2600, which was great since it had a ton of documentation), and was struggling to get the right colors on screen. Once I figured out the palette issue, I went online and found a 12-bit 64 color palette that I was able to input into code that I modified (linked below) to create the module “pitfall_color_mapper.sv”, which decides what RGB Hex values to assign to the VGA signal based on an input value. This is the code that I modified– I modified code that turned a palette that can be assigned by value and converted into sprites to fit the palette that I chose. In terms of the rest of the project, there is one bug that I was unable to resolve, and it has to do with Harry’s jumping motion. For some reason, even after letting go of jump, if you don’t hit the “ground”, the next time you press jump you will continue downwards until you hit the ground. With this in mind, the movement of Harry is playable, but really not completely up to par with what I had in mind.

I wish I had more time to really iron out some of the bugs with Harry’s movement, and actually get the 8 bit logic register random level generator working properly with the right score counting from the game, but unfortunately I ran out of time for these add ons. I knew that I would be pushing it to complete all of the “Additional” things I included in my report, but I believe I hit almost everything on the dot. I was able to draw the obstacles and Harry on screen with collision logic. I was able to create a treasure and random levels that Harry is able to go through. I was also able to integrate the background color palette, and the USB interface that would control the movement in the game. In terms of the additional features, the random levels are a great add on to the game, since it feels like a free run everytime you play.

Overall, this was a fun project, and I am glad I chose something a little ambitious with little documentation, as I was able to explore the possibilities and explore my creative side a little more. The final product can be seen in figure 6– it looks slightly different, but has all the core features of the Pitfall game. I think the difficult level of this project comes from the fact that I did almost everything in System Verilog, instead of using C through NIOS. However, I am glad that I did this, since I wanted to understand the sprite drawing and graphics side of System Verilog outside of Lab 7.

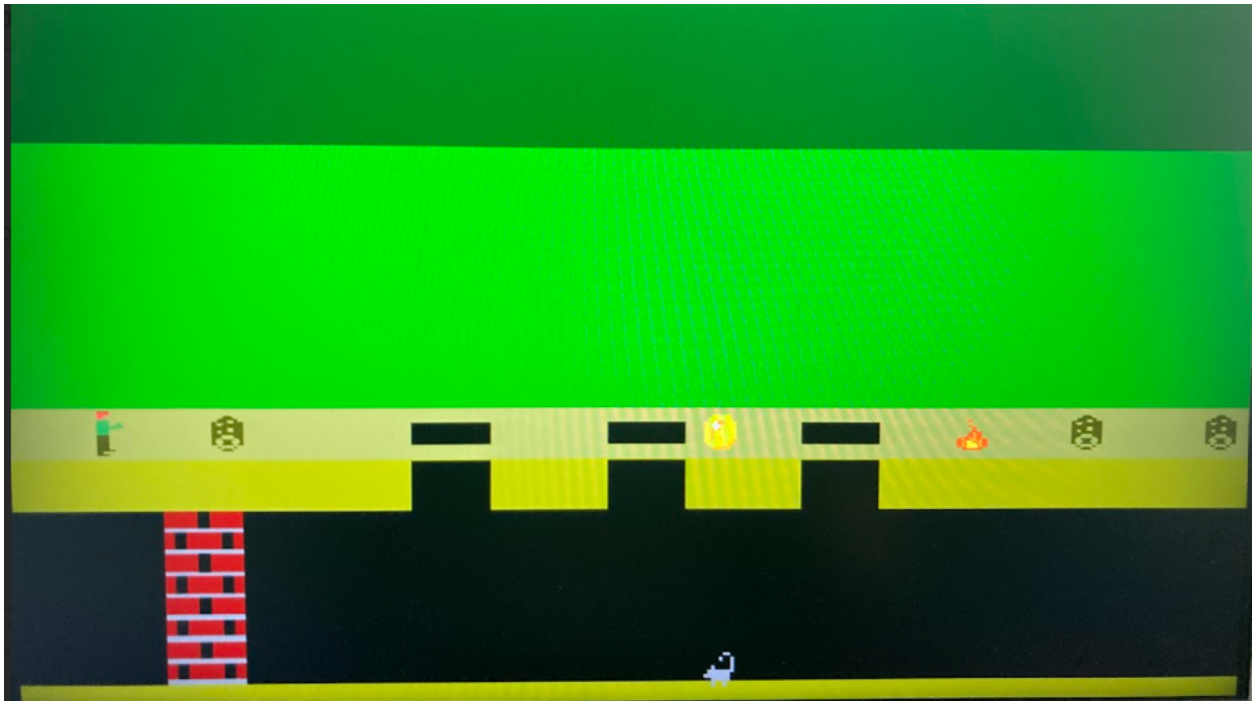

Figure 6: Final Product

Resources:

- Background on Pitfall
- Article on how Pitfall's levels are generated
- Color Palette I used for Pitfall
- Tutorial for more complicated Sprites
- Project I used for inspiration for converting PNGs to Sprites using python code