CompArch Final Project: Retro Game Parallax

Elvis Wolcott and Melissa Kazazic

December 2021

1 Introduction

Parallax in video games is used to simulate a 3D view on a 2D space, creating depth in an otherwise flat world [2]. When a character is made to move left or right in a 2D sidescroller, the objects further from view are made to move slowly, whereas the objects closer in view are made to move faster. Using these concepts, parallax can be used to simulate moving background scenery or near-3D objects, such as a moving waterline in which the waves closest to the viewer move faster than the waves further away. [3].

2 Implementation

YouTube video showing the parallax display in action.

Let's first break down how multiple layers were used to create a parallax effect, by providing a summary of the pieces and parts. To create the parallax effect, we stack each layer on top of each other, with the "closer" top layers moving faster than the "farther" bottom layers. Each layer wraps around individually as it overflows in the pixel column counter. We visualize this on the screen using the ILI9341 display controller. This layer breakdown is seen in Figure 1.

We started with the working touch screen and display controllers from the etch-a-sketch lab to allow us to focus our work on the 2D graphics portion of the project.

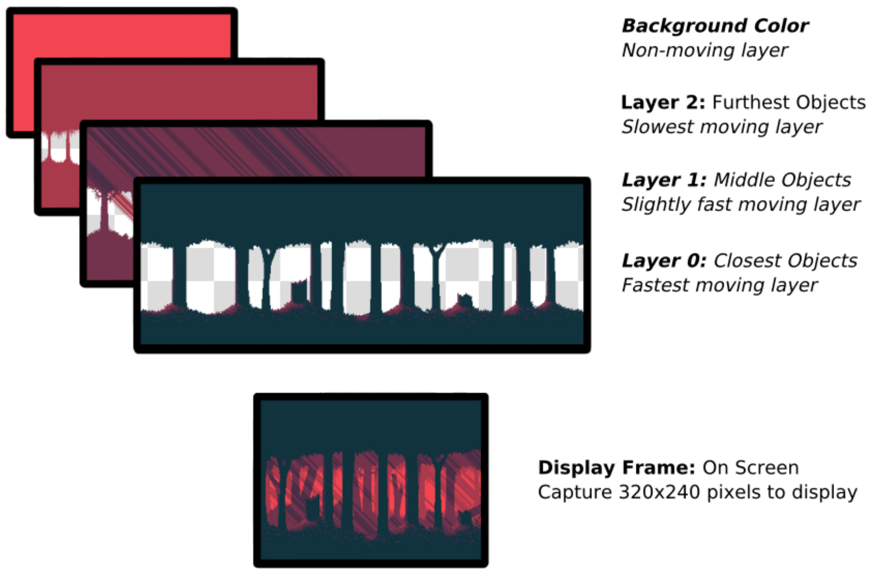
Background Color
Non-moving layer
Layer 2: Furthest Objects
Slowest moving layer
Layer 1: *Middle Objects*
Slightly fast moving layer
Layer 0: *Closest Objects*
Fastest moving layer
Display Frame: On Screen
Capture 320x240 pixels to display

Figure 1: Parallax Layer Breakdown

Next, we need to control which direction, if any, the layers start moving. We want a case where the layers move left, and a case when the layers move right. Using the screen's FT6206 touch controller, we determine at which (x,y) location of the screen the user is touching. Splitting the screen in halves on the horizontal side (based on the display length), we determine the left side and the right side of the screen, providing a middle point of the screen. If the user is touching the left side of the screen, we have the layers move towards the right. If the user is touching the right side of the screen, we have the layers move towards the left. Each layer moves slower according to the layer's distance from the front.

2.1 Choosing Layer Images

For the layers, we needed to have a working sidescroller background for us to use. We searched through free-to-use examples, and found one we liked - Demon Woods Parallax Background by Aethrall [1]. Utilizing all layers, we get - a background color layer, a background trees layer, and middle layer, and a foreground layer. These and the movement of these layers will create the parallax effect.

2.2 Compressing Layer Pixel Colors

One important process we needed to do was to get the layer image data to load on the FPGA's BRAM in order to display on the screen. Initially, we had planned on initializing all of the image data onto BRAM, but it turned out that there was not enough space to do that. So, we had to compress and break these images down to store in the smaller spaces.

We needed to compress the way that the FPGA stores the pixel color data. We had started by storing every pixel in an image as a 16-bit hexadecimal value (rather than the full 24-bit used for colors, essentially using RGB565 rather than RGB888), but that ended up using too much space in BRAM before for just a single layer.

We further compressed the colors used into being defined as 8-bit hexadecimal values, allowing for 256 unique colors to be generated, rather than the millions possible with RGB888. Using a Python script, we take the chosen layer image files, and convert them into a .memh file of the colors of every pixel in the image, using the Python library called Pillow to open the image and to get all the pixels. Transparency is encoded as the color rgb(0,0,0). The images are saved in column first order instead of row first (the standard for images). This is convenient because it allows us to take shift left and right in the image applying a single offset to the read address.

Another color compression scheme that was popular in retro hardware designs was using color tables. Color tables map the 8-bit colors to 256 colors selected by the artists. This allows a richer depth of colors while working with the same number of unique colors. For example, a volcano level can have hundreds of shades of reds and oranges and a forest level can use a different color table with countless greens.

No matter the final color compression scheme, it is vital that the colors read from VRAM by the display controller are decompressed to 16bit colors for the display.

2.3 Moving Layers at Different Speeds

Our next step was to get the different layers to move at different clock speeds, according to which layer was at the back (furthest away) and which layer was at the front (closest, final pixel colors chosen). We needed to set up a clock divider, so at the clock cycles at which we want to move left or right, the layers will move according to each distance, to simulate a 2D-3D space. There is a slower clock that the movements are based off and each layer has it's own counter to adjust the speed.

The foreground layer, or the closest layer, has the fastest speed as it moves across the screen. The background layer must have the slowest speed as it moves across the speed. For every positive edge of the slow clock, the front, middle, and background layers move every 13, 32, and 128 cycles respectively. This approach provides a adjustable level of granularity while keeping motions as smooth as possible with the display. Moves are accomplished by incrementing or decrementing (based on direction) the read offset for the layer. Since the background color layer is all of the same color pixel, it doesn't need to be moved, so we leave it alone.

2.4 Deciding Moving Layer Direction

As described in the summary, a touch input from the user controls the direction in which the layers begin to move. When touching past the defined horizontal half-way point on the screen, the background starts moving towards the left - if a character starts moving in that right direction, the relative position of those objects will start moving towards the left. If touching the left side, the layers start moving towards the right. So, according to the direction of the object - which is done by a ternary function comparing the touch location to the horizontal halfway point - the layer offsets are either increased or decreased in value according to the layer's speed, mod the overall layer's width (as each layer closer to the viewer is wider than the layers in the back).

2.5 Compositing Layers

Given that there are multiple layers, we want to make sure that each writes over the other the way we want. It needs to be that the non-transparent values of a layer that is closer to the viewer shows up over the further back layers. When generating the layer pixel-color values, all of the transparent pixels are set to the value to black - which will define which pixels can be replaced by a layer behind that layer.

We have a composite layer MUX that chooses which color to display when comparing two layers. For every pixel, if the pixel on a top layer is "transparent", the color on the bottom layer is set for that pixel. Otherwise, the top layer's color is chosen. Just like MUXes can be combined in a binary tree to for larger numbers of inputs, we can chain compositors together to composite an arbitrary number of layers. For our 4 layer image, 3 compositor MUXes are required. The first creates a layer comparing a background color layer and the background layer above it, the second composites the middle and front layers, and then a composite is made between these two new composited layers.

Essentially, if the front layer has a transparent pixel, the middle layer decides the color of that pixel. If that pixel is also transparent, then the background layer decides the color of that pixel. If all of these layers have that pixel has being transparent, then it is filled with the background color layer. This is what simulates a layer overlay.

Traditionally layering was accomplished by painting to the VRAM/frame buffer multiple times. However, because our layers are in different BRAMs on the FPGA, we can simultaneously read from each layer and composite them purely combinationally.

3 Future Plans

Initially, our plan had been to implement parallax into a 2D side-scrolling game. If we were to build towards this idea. There are many different directions we could take this. First, we could implement different effects, such as a fog effect, for the visualization. Our next step would be to integrate a controller - for moving the parallax background at first. Then, integrate a sprite moving across the foreground, adding sound and speakers, and building the game. There would be issues from this that we've already encountered - the RAM usage - so we would need to optimize for this. A solution to this is tile based graphics, where graphics are constructed from smaller reusable elements.

References

[1] Demon Woods Parallax Background by Aethrall:
https://aethrall.itch.io/demon-woods-parallax-background

[2] Explanation of parallax in a sidescroller:
https://gamedevelopment.tutsplus.com/tutorials/parallax-scrolling-a-simple-effective-way-to-add-depth-to-a-2d-game--cms-21510

[3] Example of use of parallax for waterline:
https://s3unlocked.blogspot.com/2017/07/the-pseudo-3d-waterline.html