

SuperMode 7 Final Report

Jane Tortorella, David Whitaker, and Kate Friedman
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
 Cambridge, MA
 janetort@mit.edu, dwhit432@mit.edu, katef529@mit.edu

Abstract—This paper presents an extension of the Mode 7 graphics technique utilizing an FPGA. Originally used on the Super Nintendo Entertainment System (SNES), Mode 7 performs affine transformations on a background layer on a scanline-by-scanline basis to create perspective. Our system replicates this on FPGA by reading user input, applying matrix transformations to a background layer, and sending the resulting pixels to a display via HDMI. This system benefits from implementation on an FPGA because it requires specialized, high-throughput computation. We also utilized the FPGA’s resources to achieve a 720p resolution, much higher than the 256x224 resolution outputted by the SNES. Our system outputs video at 60 frames per second.

Index Terms—Mode 7 Graphics, SNES, Field Programmable Gate Arrays, Affine Transformation

I. DESIGN OVERVIEW

Our system runs on a 74.25MHz pixel clock and drives HDMI output. Game and demo modules generate affine parameters based on user input and are passed to the affine module. RGB color values at the transformed pixel coordinates are fetched from a tilemap, tileset, and palette stored in BRAM. Line-specific affine parameters create a pseudo-3D perspective, as used in several SNES games. The user can switch between demo scenes using FPGA switches. An SNES controller provides real-time input.

Although inspired by the SNES, the architecture differs due to the FPGA’s constraints. Our system significantly improves on the SNES, achieving a 720p resolution, much higher than the SNES’s 256x224 output. It also allows for larger tilemaps and tiles, while the SNES is restricted to 8x8 tiles and 128x128 background [1]. Our system uses 16x16 tiles, effectively doubling the background size.

II. MODULE DESCRIPTIONS AND DESIGN

A. Controller Manager and Integration with Hardware

We integrated a physical SNES controller to let users control demo scenes. The controller has 12 buttons, and communicates using a protocol similar to Serial Peripheral Interface (SPI), but more rigid and specialized. To interface with it, we wrote a module that mimics the console’s behavior during this exchange.

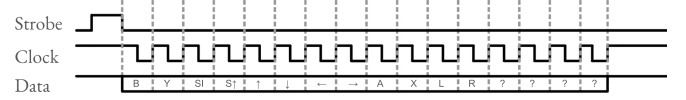


Fig. 2. SNES Controller Timing Diagram

At 60Hz, our controller manager module sends a single cycle “strobe” pulse that tells the controller to capture its button states (active low). Then, the module sends 16 clock pulses. With each pulse, the controller outputs a bit of data in the order shown in Figure 2. We sample bits on the falling edge of the data clock. The first 12 bits represent the buttons, and the remaining four encode the controller’s device ID, which we do not use. The controller manager module outputs button states as a 16-bit bus, used throughout the system.

The SNES controller uses a 5V logic level, whereas the FPGA operates at 3.3V, so we used a bidirectional logic level shifter. The schematic can be seen in Figure 3. FPGA signals are shifted from 3.3V to 5V before reaching controller inputs, and signals from the controller are shifted from 5V to 3.3V en route to the FPGA. We powered the 5V side of the shifter from the FPGA’s 5V pin, which is sufficient given our low current requirements. We faced wire contact issues, and we explored two solutions to this; lengthening wire leads and intertwining wires to create thicker leads. Both approaches ensured stable contact.

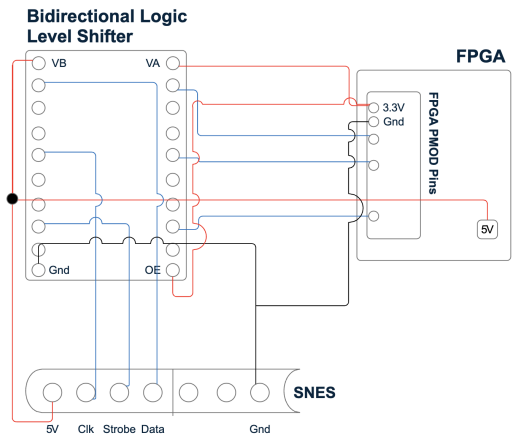


Fig. 3. Schematic illustrating connections between FPGA, logic level shifter, and SNES



The affine transformation module performs the primary operation of Mode 7 by scaling, rotating, translation, and

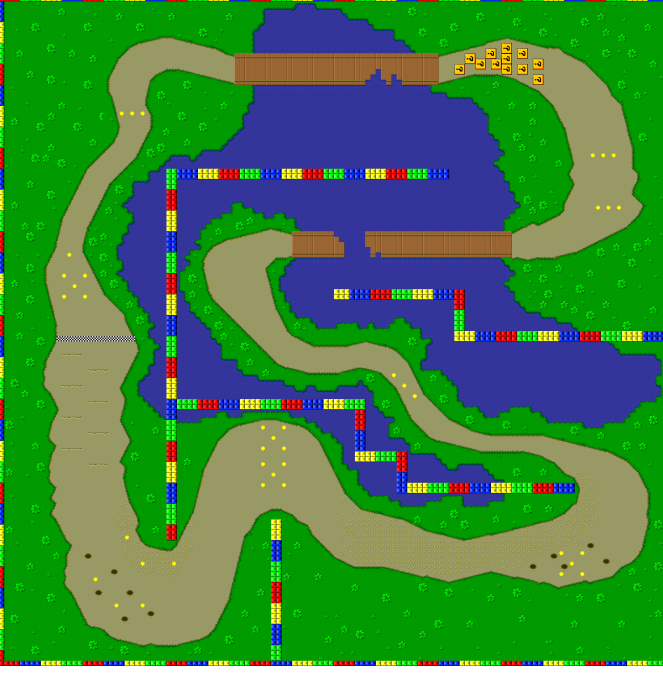


Fig. 5. Full background from Super Mario Kart, assembled using only tiles from Fig. 4

shearing the tilemap as it is displayed. The transformation is described by the following matrix multiplication:

$$\begin{bmatrix} x_{\text{affine}} \\ y_{\text{affine}} \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x + x_0 - x_c \\ y + y_0 - y_c \end{bmatrix} + \begin{bmatrix} x_c \\ y_c \end{bmatrix}$$

where (x, y) are the desired display coordinates, (a, b, c, d) control the rotation, scaling, and shearing, (x_0, y_0) define the translation, and (x_c, y_c) define the center point of the transformation.

The affine transformation implementation on the FPGA takes each of these as inputs. Additionally, our implementation defines new parametric scaling sizes for the constants, introducing `AFFINE_SCALE` and `OFFSET_SCALE`. `AFFINE_SCALE` defines the transformation such that $a...d$ are interpreted where

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} 2^{\text{AFFINE_SCALE}} & 0 \\ 0 & 2^{\text{AFFINE_SCALE}} \end{bmatrix}$$

creates an identity transformation, e.g. there is a 1-to-1 display pixel to tilemap pixel relationship. This allows for finer granularity in choosing our scaling factors by allowing for a larger quantity of fractional values to be computed. This has a low impact on the transformation on its own, but for utilization in horizon modeling with larger screen sizes this becomes a necessary feature, as otherwise changes in perspective scale become too small and we get tearing effects.

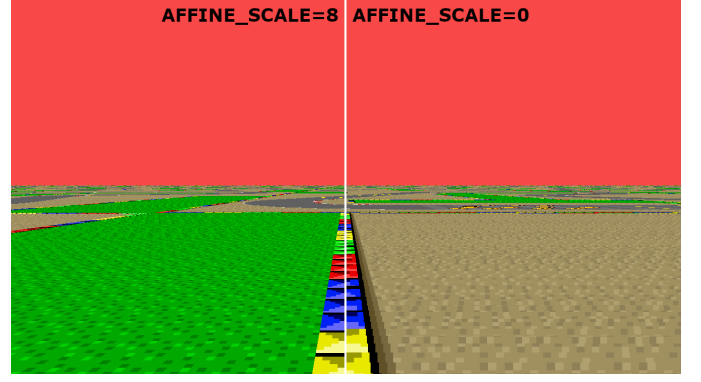


Fig. 6. Effect of `AFFINE_SCALE` parameter on tearing on a 720p display, visualized using Python emulator.

`OFFSET_SCALE` similarly defines a scaling effect applied to x/y_0 and x/y_c , e.g. $x_0 = 2^{\text{OFFSET_SCALE}}$ translates the output coordinate by 1 pixel horizontally on the tilemap. This dictates the smallest step we can take, or how smooth any "motion" across our tilemap/horizon is.

Unlike on the SNES, we additionally implement an instantiation of this affine transformation module with an input θ . For basic rotation transformations, such as in our simple horizon effect, this abstracts the need for a wave lookup-table from whatever module is sourcing inputs $a...d$. Given θ , we utilize a BRAM of width `ROTATION_SCALE` and depth $2^{\text{ROTATION_RESOLUTION}}$, each entry corresponding to a signed value in the range

$$[-2^{\text{ROTATION_SCALE}-2}, 2^{\text{ROTATION_SCALE}-2}]$$

. We then perform the modified matrix multiplication:

$$\begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x + x_0 - x_c \\ y + y_0 - y_c \end{bmatrix} + \begin{bmatrix} x_c \\ y_c \end{bmatrix}$$

All outputs are then shifted by a factor of `ROTATION_SCALE-2` to restore them to the original coordinate space. We have two versions of the affine transformation module, one that has this rotation and one that does not.

D. Basic Demo Module

The basic demo module exists to demonstrate the function of our affine and tile modules. It connects user input directly to the affine parameters, allowing the user to manually change individual parameters with buttons and switches on the FPGA. This is extremely basic and does not allow the user to change parameters on a line-by-line basis, as is needed to create a pseudo-3D horizon effect.

E. Simple Horizon Module

The simple horizon demo is a basic implementation of the pseudo-3D horizon effect with user control. It alters the affine parameters a and d on a per-scanline basis, setting them to the quotient of a scaling factor and the current vertical line. Division is completed in the horizontal blanking period of each scanline. This division by scanline roughly estimates the

$\frac{1}{\text{distance}}$ relationship that describes the scaling relationship of objects in visual projection. The aforementioned scaling factor is determined by the camera height, and was selected entirely by tweaking the value based on visual output until it looked "correct".

For this horizon model, the other affine parameters are constant across all scanlines in each frame. x_c and y_c are set to $x_0 + \text{DISPLAY_WIDTH} / 2$ and $y_0 + \text{DISPLAY_HEIGHT} + \text{HORIZON_OFFSET}$, where HORIZON_OFFSET describes the vertical height of the horizon. Changing these parameters ensures proper centering of the perspective effect, and also allows for accurate rotation. We took these values from Kulor's Guide to Mode 7 Perspective Planes [2], a resource that has been continually proven invaluable in this project.

The user can modify the x_0 and y_0 affine parameters manually with the controller or FPGA buttons, which effectively changes the camera location. The user can also change the angle of rotation.

The video mux only displays this output if the line count is below the horizon offset. This prevents the unintentional¹ reflection of the map at points where the calculated "distance" to the pixel is negative.



Fig. 7. Simple horizon demo displayed on a monitor. Note this is an older project snapshot, and we have since switched to a different horizon background.

This perspective effect is described as "simple horizon" because while it satisfies the requirement of creating a static effect that accurately resembles a horizon, it lacks positional accuracy as well as dynamic control of view angle. This creates undesirable visual effects while modifying parameters, such as movement appearing to zoom in along certain axis, while additionally preventing the use of additional features such as projection calculations for the accurate mapping of sprites onto the world.

F. Perspective Accurate Module

This module aims to resolve the problems raised by the "simple" horizon model. To do this, we initially begin from the reference point of the idealized camera model from Kulor's

Guide [2]. Instead of defining an arbitrary offset and scale factor to closely approximate an accurate horizon projection, we instead define the expected behavior of a camera object within our virtual world.

To instantiate this, we first define our camera's controllable features: Position (cam_x , cam_y , cam_z), Yaw (yaw), Vertical Field of View (fov), and pitch from horizontal (pitch). These inputs are handled using the same parameters OFFSET_SCALE , AFFINE_SCALE , and $\text{ROTATION_SCALE/RESOLUTION}$ to ensure consistency between modules, primarily maintaining a similar relationship between camera position and affine offset nomenclature.

From here, we begin the process of calculating the necessary affine parameters to accurately describe the camera's location. This can be divided into two primary tasks: Computing the offsets and center of transformation given the camera's coordinates, and determining the line-by-line matrix scaling factors.

1) *Positioning*: The first stage in computing the transformation is to determine the center-point on the tilemap from which the transformation is being applied (x_c , y_c). This is a near one-to-one relationship to our camera coordinates (cam_x , $-\text{cam_z}$). Note the change in coordinate definition between y-axis and z-axis as we convert from our 2-dimensional tilemap into our full 3-dimensional projection's coordinate system. This has no impact on the true functionality of the module, but have been left as is for continuity between FPGA implementation and original source [2]. Additionally, to correct for the impact of camera height and changing pitch, we adjust each of the centering coordinates using a factor of $\cos(\text{yaw})$ and $\sin(\text{yaw})$ multiplied by $\tan(\frac{90-\text{pitch}}{2})^2$ to help counteract an apparent "drifting" effect as we change the height of our camera with a non-vertical pitch. From there, we calculate the horizontal and vertical transformation offsets (x_0 , y_0) to ensure the proper part of the world is displayed after the transformation. This is done by first aligning the offsets with the transformation center (x_c , y_c) before offsetting from said location to display the proper section. In the x-direction, this is a simple subtraction of $\text{DISPLAY_WIDTH}/2$. In the y-dimension, this is a little more involved as we here we account for the distance from our first scanline that is being rendered to our transformation center. This simplifies to a simple ratio of trigonometric features of the camera definition, scaled by our zero-indexed screen height to bring us back into the output screen coordinate-space:

$$\text{lineoffs} = \frac{\cos(\text{topa}) - \cos(\frac{\theta_{FOV}}{2})}{\cos(\text{topa}) - \cos(\text{btma})} * (\text{display_height} - 1)$$

where topa and btma are defined as the angles from vertical, facing towards the ground plane, of the top of the camera's FOV and the bottom of the camera's FOV:

$$90 - \theta_{\text{pitch}} \pm \frac{\theta_{FOV}}{2}$$

²Why a half angle? The original explanation used by Kulor cannot be directly applied as multiple simplification layers have restructured the order of the math (and even he doesn't appear to fully understand it), so I am leaving it for outside the scope of this explanation. But it works.

¹A number of SNES games exist where this effect is actually desired to create a ceiling/tunnel effect, in which case this is disregarded.

When our camera is pitched directly downwards, we address the division by 0 by instead setting the value to $DISPLAY_HEIGHT/2$ in similar fashion to our x-dimension, the value that the division converges towards. This accurately places our transformation on our tilemap.

2) *Scaling*: The scale of each line is defined, as we established within simple horizon, proportional to $1/distance$. In a true ray-casting environment, we would calculate the distance from our virtual camera to the ground plane directly, but to reduce the computational complexity of the system we take some liberties to define our camera uniquely for the Mode 7 interface. We achieved an accurate scanline-dependent scaling representation by performing a linear interpolation between the longest and shortest distances from the camera to our transformed plane. These distances are both a factor of cam_y , so simplifying that out of the intermediate result we get an interpolation between our previously calculated values of $\cos(topa)$ and $\cos(btma)$. Using this result, and rescaling by necessary values to convert from camera-space to display-space, we have a linear function of scanline that produces the necessary affine transformation matrix:

$$\begin{bmatrix} \cos(\theta_{yaw}) * \text{scale}(\text{scanline}) & \sin(\theta_{yaw}) * \text{scale}(\text{scanline}) \\ -\sin(\theta_{yaw}) * \text{scale}(\text{scanline}) & \cos(\theta_{yaw}) * \text{scale}(\text{scanline}) \end{bmatrix}$$

3) *Hardware Implementation*: Due to the nature of these calculations, such as camera position and certain scanline scaling constants only changing between frames, we equally divided the computationally intensive parts of these into per-frame and per-scanline math.

During per-frame calculations, we determine all offset/centering parameters, as well as pre-computing sin and cos values necessary for the scanline math. This is implemented as an FSM, where we wait to enter the primary calculation cycle until the rising-edge of our vblank period. To support multiple wave lookups from the same memory block, we periodically input a new input angle to the BRAM, before saving off the resulting value two cycles later while queueing the next angle acquisition. Then, on completion of all angle lookups, our FSM disables our memory block while continuing on with multi-cycle divisions. Where necessary, we use multi-cycle signed division to compute output values. These dividers are variable length dependent on the size of parameters instantiated, including but not limited to previously defined $OFFSET_SCALE$ and physical constants $DISPLAY_HEIGHT$. The resulting values are then pushed to the output registers on the falling edge of the next hblank period to ensure they are ready by the time we begin our first relevant scanline calculation (which will not occur until multiple lines later)

Per-scanline calculations are performed similarly, using an FSM triggered on the a change in vcount. Once triggered, the FSM begins the calculation of the scanline-parameters for the following line, vcount + 1. As with the per-frame division, this result is then stored within the FSM until the next hblank rising edge, at which point the values are pushed to the respective output registers in time for the next line to begin fetching the proper pixels from memory through the remainder of our pipeline.

Finally, the Perspective Accurate model also provides two additional outputs to support our parallax and horizon modeling: `horizon_scanline` and `show_horizon`. The former performs one additional division during the per-frame stage that outputs the top-most line of the ground plane, or the horizon line, limited to zero in the negative direction to prevent accidental wrapping. This allows for the accurate placement of our parallax effect in relation to the ground plane. The `show_horizon` flag similarly indicates whether or not the current line should output from our horizon parallax or from our Perspective Accurate ground plane portion.

G. Perspective Accurate Demo Module

The perspective accurate demo allows the user to "fly" over the ground plane. The module controls the perspective accurate module's parameters using input from the controller. The up and down buttons on the d-pad control the pitch, while the right and left buttons control the yaw. By pressing the A button, the camera will move in the direction of the pitch and yaw, and the B button will move the camera backwards relative to these directions.

The sine and cosine of yaw and pitch are used to control the direction of the movement. The camera's x , y , and z coordinates are proportional to the following:

$$x \propto \sin(\theta_{pitch}) \times -\sin(\theta_{yaw})$$

$$y \propto -\sin(\theta_{pitch})$$

$$z \propto \cos(\theta_{pitch}) \times \cos(\theta_{yaw})$$

These proportions may seem odd, but they align with the coordinate definitions within the perspective accurate model. Figure 8 illustrates these coordinate definitions. Note the starting pitch and yaw angles, as well as the positive z direction.

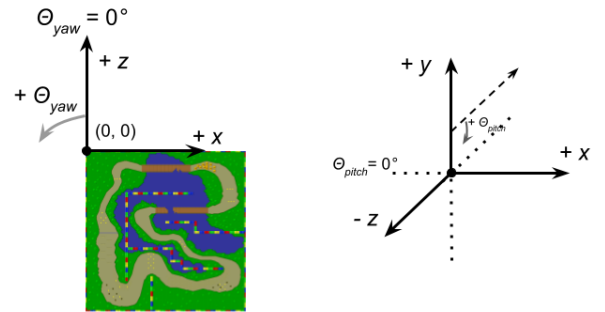


Fig. 8. Coordinate definitions within perspective accurate module.

The sine and cosine values are passed to this module from `perspective_accurate` itself, which is sufficient because it pushes these values once every frame. It is also streamlined because this module already contains the sine lookup table.

The perspective accurate demo allows for pitch beyond the range of $[-\frac{\pi}{2}, \frac{\pi}{2}]$. This creates an odd effect that is a sort of inversion or vertical flip of the scene. Because of this possibility, we also needed to allow for vertical flipping of the

background when this happens. This case is fully accounted for in our design.

H. Kart Game Module

The kart game controls the perspective accurate parameters based on user input to allow for gameplay inspired by Super Mario Kart on the Super Nintendo. The user can use the d-pad's left and right buttons with the A button to drive the kart around the map wherever they wish. This is incredibly simple - there are no game mechanics at all aside from forward movement. It intends to show how our graphics pipeline can be used to create engaging pseudo-3D games, just like the SNES.

The game module uses a simple state machine to control acceleration and deceleration. It is choppy and likely isn't the best approach, but is far cheaper in terms of combinational logic than the alternative of creating a semi-realistic physics engine. Despite this compromise, when the car is moving at full speed, it feels quite similar to Super Mario Kart. An image of gameplay can be seen in Figure 9. Like the perspective accurate demo, this module takes sinusoidal calculations from the main perspective accurate module.

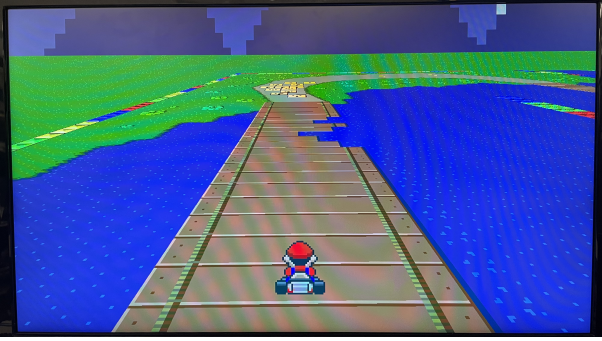


Fig. 9. Mario Kart gameplay on our system.

I. Parallax Module

The parallax module intends to create an accurate parallax effect for all the pseudo-3D demos. It applies a function to transform a horizon's x coordinates such that, when the user rotates the scene, the horizon appears cylindrical. The transformed x indicies are fetched from a lookup table stored in BRAM and then passed to a horizon-specific instantiation of the tile module. This parallax transformation acts in parallel with affine, and only acts on the horizon part of the screen.

The parallax polynomial was initially estimated by averaging a particular third-degree polynomial with the line $y = x$. This looked better than the rudimentary option of simply horizontally scrolling the horizon, but it was still relatively inaccurate. We therefore went back and took a different approach in deriving the function.

We visualized the horizon background as lying on the inside of a theoretical circle. We set our field of view (in the circle) to

120 degrees for simplicity. This is within the range commonly used in racing games.

$$\theta_{FOV} = \frac{2\pi}{3} \quad (1)$$

θ represents the angle as we sweep across the inside of the circle, and we will soon write it as a function of x . The centerline is located at $\frac{\theta_{FOV}}{2}$.

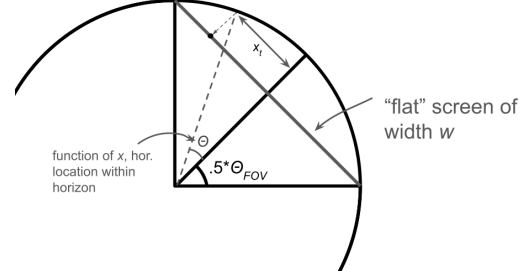


Fig. 10. Visualizing the mapping of a cylindrical layer to a flat screen.

We use a simple proportion to relate x , the horizontal coordinate within the screen, and θ , the angle sweeping across our circle. w represents the width of the screen. We divide w and θ_{FOV} by two because the function is symmetrical about the centerline.

$$\frac{\frac{\theta_{FOV}}{2}}{\theta} = \frac{\frac{w}{2}}{x - \frac{w}{2}} \quad (2)$$

And solve for θ :

$$\theta = \frac{\theta_{FOV}}{w} \times (x - \frac{w}{2}) \quad (3)$$

Taking the sine of this angle results in the relation we are looking for, as it will give us the horizontal position within the circle. We then transform it from the $[-1, 1]$ space to our intended $x \in [0, w)$ space:

$$x_T = \frac{w}{2} + \frac{w}{2} \times \sin\left(\frac{\theta_{FOV}}{w} \times (x - \frac{w}{2})\right) \quad (4)$$

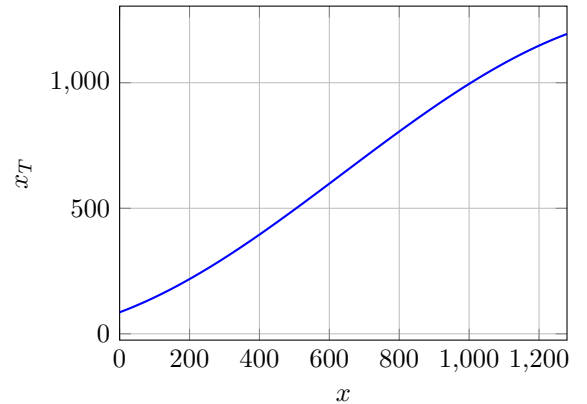


Fig. 11. Parallax function. $\theta_{FOV} = \frac{2\pi}{3}$, $w = 1280$. Can also see on Desmos.

When applied to the background image, this polynomial stretches the edges of the horizon, while leaving the center untouched. When the user rotates and the horizon scrolls, the horizon movement with this transformation appears cylindrical. A still-frame of this effect can be seen in Figure 12.



Fig. 12. The final parallax function applied to a sample background.

J. Sprite Module

The sprite module outputs an RGB pixel value given a sprite position and the current horizontal and vertical draw location. The module also offers support for as many frames as the user requires (within memory constraints), and sprites of any size. We opted not to use tiling for our sprites, as the modularity was not required for such small objects. The module uses a palette for color representation. Upon instantiation, the user can specify exactly how large the palette is, ensuring no memory is "wasted". This is especially useful in our system because the sprites will always be pixel art, and few colors will be used.

To further save memory, the sprite module has an option to flip the sprite horizontally. This is especially valuable in our Super Mario Kart demo, as the sprite will need to "turn" in both the left and right directions. The horizontal flip option allows us to store only the frames of the car turning right, reducing memory needs by approximately 50%. These sprites can be seen in Figure 13.

Of course, sprites have variable shapes, and are almost never squares or rectangles. Our module handles transparency using an output flag. When instantiating the sprite module, the user specifies which palette index represents the "transparent" color. If this color is loaded from the palette, the transparent flag goes high, instructing the video mux to display the background color instead of sprite color at this location.

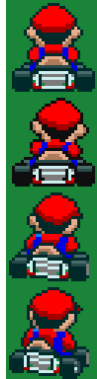


Fig. 13. Example spritesheet from Super Mario Kart. Frames can be flipped horizontally.

A sprite controller determines the frame and horizontal flip inputs. We wrote a basic sprite controller for the Mario Kart demo, which uses button input to determine sprite frame and horizontal flipping. The module is a basic state machine. The car/driver sprite for this case will be 128x128 pixels.

III. TESTING AND EVALUATION OF DESIGN

A. Memory and DSP Usage

The FPGA has 2,700 kb of BRAM available, and our system is well within this limit. Information on BRAM usage can be

found in Table I. Per our Vivado utilization report, our project uses 43.33% of the available BRAM units.

There are still areas in which we could further optimize this - mostly the horizon tilemap, tileset, and palette, which are excessively large. We could reduce these by limiting the pitch of our perspective accurate demo, thus not allowing the user to see as much of the sky/horizon. In this case, we could get away with a smaller horizon background. There are also small optimizations that could be made to the structure of the .mem files in cases where the palette and tileset require a smaller index than 8 bits. This was discussed previously in the tile module section.

Our project also has significant DSP usage, as the affine transformation requires multiplication, as do any modules in the perspective accurate realm. Our system uses 85 out of the 120 DSP units on our board, putting utilization at 70.83%.

TABLE I
BRAM USAGE BREAKDOWN

Source	Description	Usage
Affine sine LUT	10-bit lookup index, 11-bit rotation resolution	11.264 kb
PA sine LUT	12-bit lookup index, 17-bit rotation resolution	69.632 kb
Parallax LUT	1280 entries, each 11 bits wide	14.080 kb
Background palette	226 entries, each representing 24-bit RGB value.	5.424 kb
Background tile-set	144 tiles, with 16x16 8-bit indicies per tile.	294.912 kb
Background tilemap	128x128, with each tile represented by an 8-bit index.	131.072 kb
Horizon palette	226 entries, each representing 24-bit RGB value.	5.424 kb
Horizon tileset	9 tiles, with 16x16 8-bit indicies per tile.	18.432 kb
Horizon tilemap	128x128, with each tile represented by an 8-bit index.	131.072 kb
Sprite palette	14 entries, each representing 24-bit RGB value.	336 b.
Sprite	Four 128x128 frames, with 4-bit palette indicies.	262.144 kb
Total usage		943.792 kb

B. Latency and Timing

We used Xilinx Vivado extensively throughout development to simulate, test, and evaluate our design, ensuring correct timing behavior and module integration. Below are our Worst Negative Slack (WNS) and Total Negative Slack (TNS), taken directly from the Vivado timing summary:

WNS (ns)	TNS (ns)	TNS Failing Endpoints
1.744	0.000	0

As shown by this slack value, our system comfortably meets timing.

The latency of our modules can be summarized in the table II:

During active draw, there are 11 cycles of latency between the video_sig_gen signals and HDMI output, which passes through the affine and tile modules. The other modules work in parallel with these modules, which can be visually understood in the block diagram (Fig. 1).

TABLE II
MODULE LATENCY

Module Name	Latency
Affine transform and rotate	4 cycles
Affine transform (no rotate)	2 cycles
Tile	6 cycles
Basic demo	1 cycle
Simple horizon	33 cycles (only in h. blank, finishes before active draw)
Parallax	4 cycles (to match affine)
Perspective accurate demo	variable cycles (only in h. blank, finishes before active draw)
Kart game	1 cycle
Perspective accurate demo	1 cycle
Sprite	4 cycles
Sprite controller	1 cycle
Controller manager	1 cycle

The perspective accurate module and simple horizon module push important values during the horizontal and vertical blanking period, and thus do not impact the active draw latency. We place them here because they involve division, which requires a significant number of cycles to compute; however, because the parameters they generate only vary on a line-by-line basis, we can get away with performing calculations in the `hblank` period and using them on the next line.

C. Qualitative Feedback

Thus far, we have also evaluated our design on a qualitative basis. We have received positive feedback from a variety of individuals on the visual appeal of all of our pseudo-3D demos - the simple horizon demo, perspective accurate demo, and kart game. Feedback has been extremely helpful, especially regarding qualitative aspects such as the parallax effect and kart movement.

D. Intermediate Testing

We used thorough testbenching in CocoTB throughout the project to ensure module functionality before integrating them into our pipeline. In particular, the affine module, perspective accurate module, and tile module required thorough testing to ensure perfect functionality in all possible use cases, the former two relying on full software emulations of behavior as performed in hardware. These emulators also generated visualizations of expected output in the form of image files. The test benches and simulations also proved very useful even after deployment of modules on hardware, as it made finding bugs significantly easier.

In particular, testing the controller manager module was challenging. Jane used a logic analyzer with Vivado simulations to validate the timing of the communication protocol used. First, she confirmed in simulation that the strobe pulse width, frequency, and 16 clock cycle data clock matched the SNES specification. She then connected the strobe, data clock, and serial data line to a logic analyzer. This capture of waveforms verified that the FPGA-generated signals aligned with the SNES timing diagram prior to connecting the actual controller. The logic analyzer also helped ensure proper power delivery to the controller, along with correct logic level shifting between the controller and FPGA.

IV. POSSIBLE EXPANSIONS AND IMPROVEMENTS

1) *Sprite affine transformations*: A promising usage of the affine module is to transform sprites. This would be particularly useful for general scaling; rather than having sprite frames that vary in size or direction, affine transformations could scale and rotate the sprite extremely smoothly without requiring large amounts of sprite frames. However, this would not be easily scalable across many sprites if only one sprite-dedicated affine instantiation was used. We would likely need to create a buffer for these transformations, which could be performed in the horizontal blanking period to pre-load the buffer.

2) *Improved Super Mario Kart game mechanics*: To expand our system to support more advanced Mario Kart style mechanics, we could improve sprite handling, particularly by using increased rotation, scaling, or priority sorting to allow for smoother animations and movement. We could also improve the player's general movement mechanics to better simulate a racing feel. Additional gameplay ideas such as item pickups, opponents, and lap detection could also be added, with varying levels of complexity. Our system is designed for improvements such as these to be integrated around the existing pipeline.

3) *Pilotwings game*: Another direction for expansion would be recreating elements of Pilotwings, which is another game that uses Mode 7 on the SNES. Pilotwings requires continuous perspective changes, smooth camera tilting, and simulation of altitude. Our perspective accurate demo indicates that our pipeline can handle vehicle-like flight over the background plane, and this would also therefore be a strong expansion on our existing pipeline.

4) *Anti-aliasing*: We hoped to integrate basic anti-aliasing into our pipeline, particularly for the parts of the scene that appear "far" from the camera. This would be feasible using our Gaussian blur convolution kernel developed in class; we would simply need to find an appropriate location for it in our pipeline and correct edge cases. Unfortunately, we did not have time to integrate this.

V. ART

Our system uses a considerable amount of pixel art. Most of this was sourced from the Internet, as we wanted to focus on the technical aspects of our project; however, we did need to manually create parts.

1) *Main background*: This background is Donut Plains 3 from Super Mario Kart, taken from The Spriter's Resource [3]. This is an invaluable site, and contains many graphics ripped directly from Super Nintendo games. Our final tileset is actually a hand-upscaled version of this; because the original has 8x8 tiles, we increased their size and manually redrew parts of them to appear sufficiently detailed for our system.

2) *Sprite*: All frames of our Mario sprite were also taken from The Spriter's Resource. [4]

3) *Horizon background*: Towards the end of development, we created a new horizon background, as existing art ripped from SNES games was not sufficient for this purpose (new background in Figure 14). We used Tiled Map Editor, which

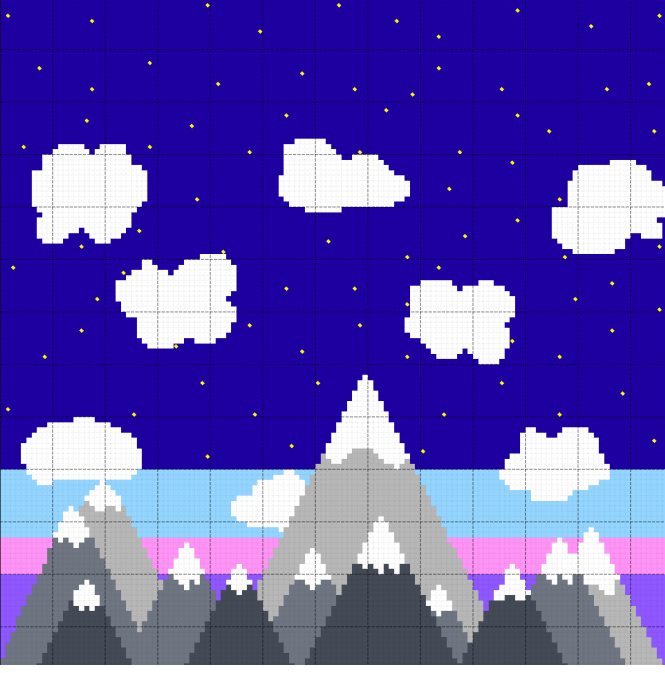


Fig. 14. 2048x2048 pixel background; created in Tiled Map Editor

is a free, open-source, software for 2D game development. The tilemap and tileset are of the same dimensions as the main background. This minimal tileset (nine tiles) was drawn in Pixel Studio, a lightweight pixel art editor, and imported into Tiled. Tiled's interface facilitated designing the scene, reusing tiles, and ensuring cleanness between tile transitions. This horizon background also went through SuperFamiconv and the Python script to generate a .mem file.

VI. DIVISION OF WORK

1) *Jane's Technical Responsibilities:* Jane's primary responsibilities were the controller manager module and the physical interface to the SNES controller, including the level-shifting circuit and mapping controller buttons to affine parameters. She also created pixel art for the horizon.

2) *Kate's Technical Responsibilities:* Kate wrote the tile fetch, simple demo, simple horizon, sprite, sprite controller, parallax, perspective accurate demo, and kart game modules. She also generated tilemaps, tilesets, and palettes from PNG files, converted them into memory files, and manually drew the upscaled background tiles at the new resolution.

3) *David's Technical Responsibilities:* David completed the affine transform module, perspective accurate module, and their respective Python simulators.

4) *Report:* Each of us wrote report sections on our own respective modules, as described above. Jane and Kate collaborated on the abstract and system overview. All of us contributed to the evaluation-related sections. Jane wrote the art section.

REPOSITORY

Our code can be found in the following git repository: GitHub MIT link.

ACKNOWLEDGMENT

The authors would like to thank Joe Steinmeyer, Kiran Vuksanaj, and Elise Wingard for their invaluable feedback and support throughout this project.

REFERENCES

- [1] Backgrounds, SNESDev Wiki. <https://snes.nesdev.org/wiki/Backgrounds>
- [2] Kulor's Guide to Mode 7 Perspective Planes, NESDev Forum. <https://forums.nesdev.org/viewtopic.php?t=24053>
- [3] Donut Plains 3, Super Mario Kart, The Spriter's Resource. <https://www.sprisers-resource.com/snes/smariokart/asset/85010/>
- [4] Mario, Super Mario Kart, The Spriter's Resource. <https://www.sprisers-resource.com/snes/smariokart/asset/6866/>