VSteroid: 2-Player Asteroids-Based Duel Game

Robi Jenik, Jishnu Dey, Kyle Wigdor, Jacky Zhao
https://github.com/nvdaz/vsteroids/tree/main

Overview

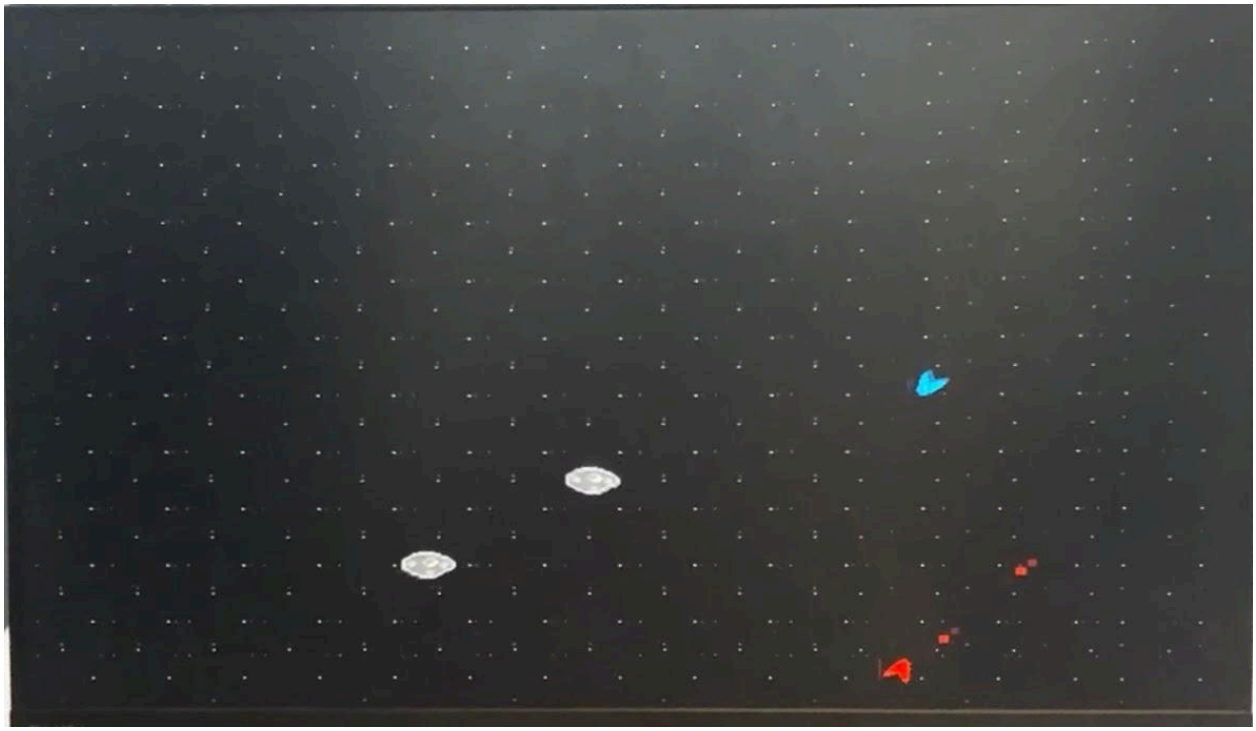
The goal of this project was to create a spin of the classic arcade game Asteroids on an UPduino FPGA.

The original game is a single-player game in which the player controls a ship on a top-down board. The ship is able to move about the screen by rotating and using a main thruster. Important to the movement of the ship is that it is heavily physics-based, with a pronounced acceleration curve, and little to no automatic deceleration, giving an appropriately space-y feel.

The player's objective in the original is to avoid collisions with the titular asteroids which drift around the screen, through a combination of evasive maneuvers and using the ship's gun, which it can use to destroy the asteroids.

In our version, there are now two ships–one per player, and the players' objectives are no longer to survive as long as possible, but rather to kill (or merely outlast) the other player. A tie can result if both players die at the same time, such as if their ships collide. Asteroids still intermittently cross the screen, but are no longer destructible.

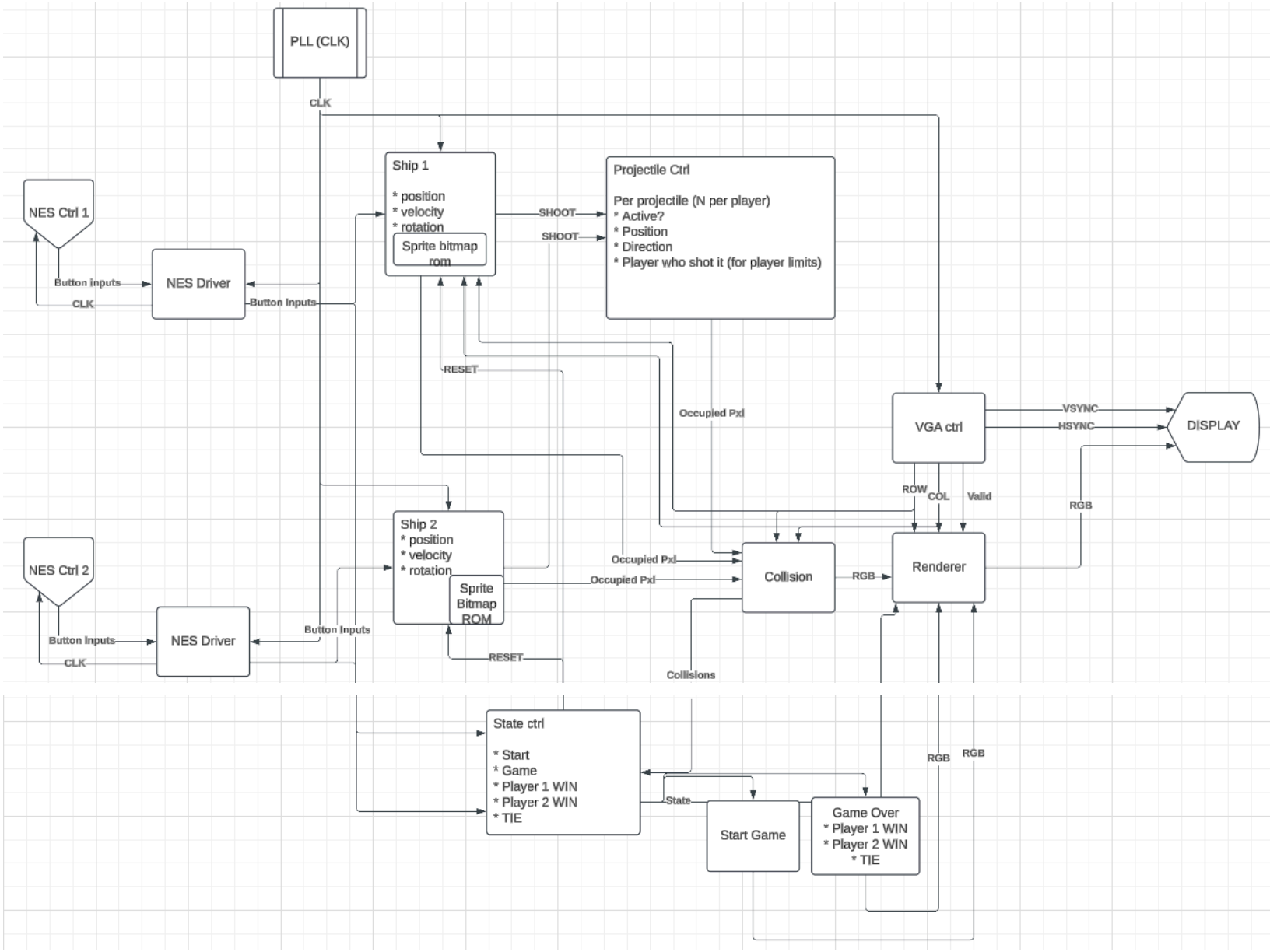
Result Preview:



This image demonstrates the key components of the game, such as the ships, the projectiles, and the asteroids. The asteroids act as unpredictable, moving terrain that the ships must avoid as they engage in combat with each other.

Technical Description

Image of the Block Diagram:

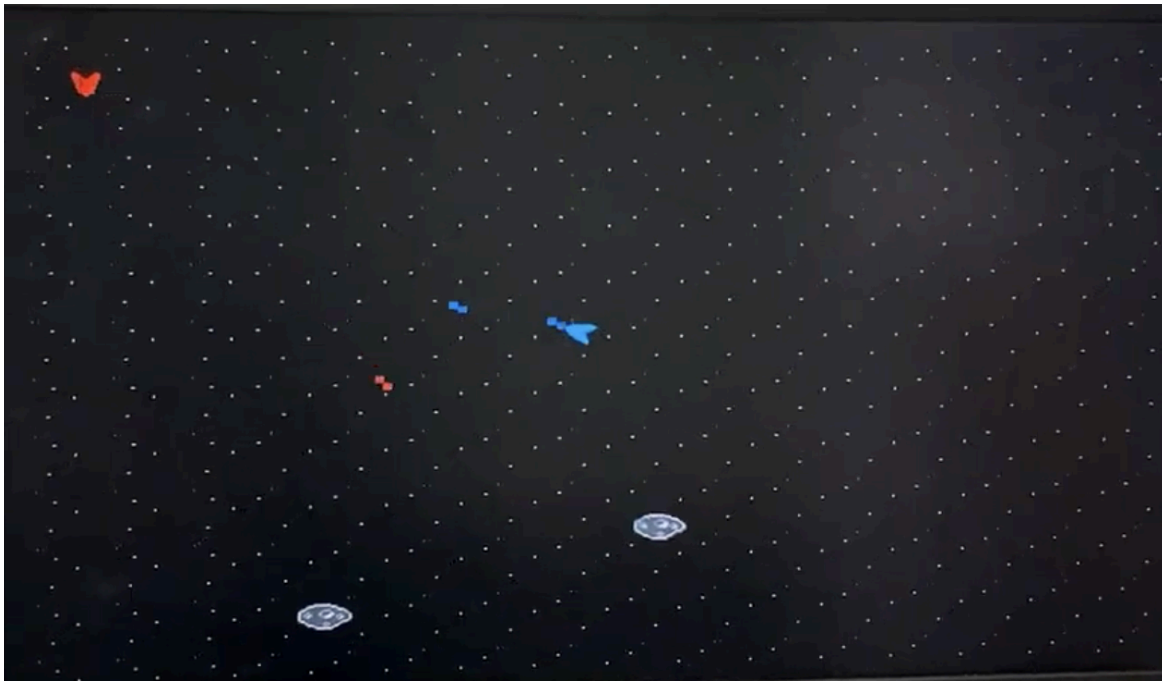


Renderer

The renderer module chooses what color each pixel should be on the screen for every frame. To this end, it receives the current row and column being rendered by the monitor (see the VGA driver module), and x and y positional values (in pixels) from each entity (ships, projectiles, and asteroids). It also has internal positions for text sprites which are displayed on a per-state basis.

These x and y positional values correspond to the top-left corner of the entity's sprite's corner. Thus, the difference of the column and the x-position/the row and the y-position results in the pixel-offset from the top-left corner of the appropriate sprite. This value is used to get the appropriate pixel from a sprite for a given pixel on screen (given that the sprite-index is in-bounds). This is done separately for every entity. The result is that for any pixel on screen, each game entity makes a claim about what the pixel's value should be. If no claim is made, the color of the pixel is rendered as black.

An important realization early on was that the process of resolving which of these values a pixel should take on could be exploited to check for collisions–two in-game entities have collided if and only if both entities' sprites laid claim to the same pixel. Thus an important secondary role of the rendering module was to have a binary output declaring whether each ship was involved in a collision.



The actual pixel value is handled by creating an ordering of priorities to each entity making a claim to a pixel.

Red Ship	Red Projectile	Blue Ship	Blue Projectile	Output
On	Off	Off	Off	Red Pixel
Off	On	Off	Off	Red Pixel
Off	Off	On	Off	Blue Pixel
Off	Off	Off	On	Blue Pixel
Off	Off	Off	Off	Black Pixel
Most other combinations should result in a game over by collision. However, we assign this debug color to any combination we want to detect.				Purple Pixel, Relevant Collision Signal

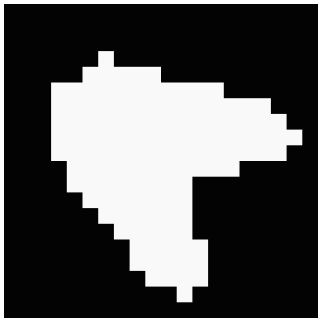
The dynamic background (4 grids of spaced pixels, 2 with dynamic horizontal offsets, and 2 with dynamic vertical offsets, plus some additional row, col, and counter based pattern generation) is the second-lowest priority, with the final being black.

Notably, the resolution is handled per “screen,” so a final pixel value is, in parallel, calculated for each of the different screens in the game. These screens include the “in-game” screen involving the position of ships, the title screen, and all of the game-over screens.

A second round of this pixel value conflict resolution happens based on which screen is currently active. It is not by coincidence that each of these screens corresponds to a single gamestate, so a single “with STATE select RGB” statement suffices to choose the ultimate value sent to the physical pins.

Sprites

Images of the ships, asteroids, and text were created using a pixel art editor and then exported as PNG files. The PNGs were then processed using the Pillow library in Python to convert them to ROM modules in VHDL.



An example output image of the ship rotated 56.25°. There are 32 such images, each procedurally generated from an unrotated, base image. Note that the edges are somewhat ragged, but when actually rendering the sprite on the screen, it is not as obvious.

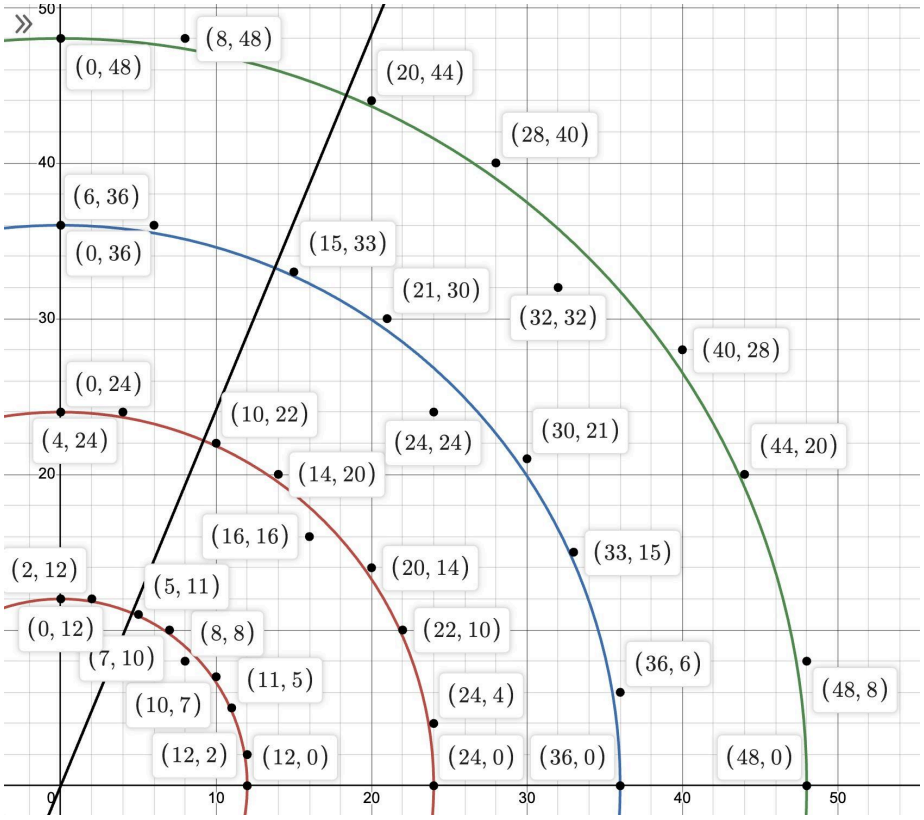
For the ship sprite, we needed to generate 32 sprites corresponding to each 32 different rotations. To do this, we first upscaled the ships from 20x20 to 200x200 using nearest-neighbors to preserve the crisp pixelation of the sprites. Upscaling was done to increase the visual fidelity of the final images. Then, sprites were generated for all rotations with value $\frac{360^\circ}{32}k$ for all $0 \leq k < 32$ so that each sprite would be evenly spaced out. Each sprite was then downscaled to 32x32 since this would be the final size of the ships that would be rendered on the screen.

All images were created with only a few colors to reduce the size needed to store them in ROM. Some images, like ship sprites, were stored as bitmaps, having just two colors, and were designated a final color value downstream of the sprite module itself. While others, like the title screen, had up to four colors. For these, a color table was created that had entries for each color. A case statement was generated that would map indexes of each color to their RGB value. Each pixel in the sprite was mapped to an address in ROM by linearizing the pixel location (x, y) as $i = ((x + y \times width) << log_2(rotations)) \mid rotation\_index$. We decided to create a linear index via multiplication to conserve space. Some of our sprites, such as the 32 different ship rotation sprites, had large non-power-of-two lengths and widths, so that could have cost a lot of space, which we did not have the luxury of having. A case statement was also generated that mapped each pixel location to the color index.

Consumers of the sprite were able to interface with the sprite modules by simply inputting a x and y position (and in the case of the ship, a rotation index). The RGB value was then the output. The linearization was surprisingly tricky to implement in VHDL because the multiplication requires that the bit width of the product should have a bit-width exactly equal to the bit widths of the two factors, and would silently fail otherwise. In practice, this meant we had to create intermediate signals that were precisely the right bit width.

The linearized position was also created as an intermediate signal as the sum of the x position and multiplied y position. Care also had to be taken to ensure that the addends were either padded or truncated to have appropriate widths. With the linearized position, the case statement (which were stored in ROM) could determine the correct color index. The colors' LUT would then take in this color index and output an actual six-bit color, which would then be outputted and used in the rendering module.

Ship Module



This is a representation for our approximations of the four different velocities. The true 2d velocities 1,2,3,4 are represented by the red, orange, blue, and green circles respectively. Only one quarter (8 values per speed) is shown.

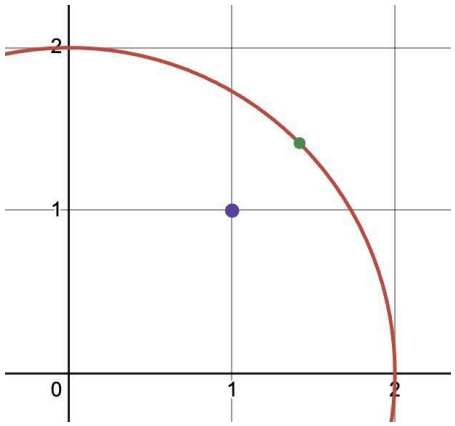
The same ship module was instantiated twice, one for each ship. The ship module accepts the controls that were being pressed for each player as well as a reset signal, and handles the positional and rotational logic of the ship. When the reset signal was high, the ship module synchronously resets the ship's position and rotation (in the

top left or top right corner) depending on whether the ship was player 1 or player 2, which was used for when the start menu state transitioned to the in-game state.

While the game is live, the ship's velocity and rotation are updated depending on the controls that are being pressed by the player. The ship follows a physical acceleration model. When the forward button is pressed, acceleration is applied in the direction that it is facing.

Each frame, if the thrust button is pressed, the appropriate acceleration x and y values are added to the ship's x and y velocities; and the current x and y velocity values are added to the ship's x and y positional values. Additionally, when the forward button is not pressed, an acceleration in the opposite direction is used to simulate a drag force. Perhaps this drag force is not realistic because a real space ship would not experience such high decelerations, but this makes the controls far more ergonomic.

To get these appropriate acceleration values, rather than perform the actual trig calculations, an LUT of approximate x and y acceleration pairs are kept for each rotation angle. Here we must explain a more general problem that emerges in trying to represent acceleration and velocity values. Ideally, movement would be normalized: speed is constant regardless of direction. However, this is not possible to achieve with discrete pixel values, or even rational ones (much to the frustration of Pythagoras and his ilk). For example, for a speed of 2 pxl/frame at a 45° angle, the actual normalized velocity is (root 2 over 2, root 2 over 2). Of course, we must then approximate. But with pixels as our units the best we can do is (1, 1). This problem gets worse the more rotations there are, to the point where at low speeds, all 32 directions are impossible to uniquely represent.



Approximation of 45° in discrete coordinates.

The solution is sub-pixel precision throughout the movement logic, in our case, down to 16th's of a pixel by using what are essentially fixed-point numbers. The problem with approximation occurs when speeds are low, and so relative errors are high. In spite of this, the position reported to the renderer by the ships is whole pixel-based by simply excluding the lower 4 bits.

On each frame, the acceleration is applied to the ship, and its velocity is set to the current velocity plus the current acceleration. Care needs to be taken in this step as this can potentially cause the ship to go off screen. Before setting the new position, the module must first check if it goes beyond the edge of the screen, and if it does, apply appropriate logic so that the ship wraps around to the other side, emulating the original game.

In a similar vein, the ship rotates either counterclockwise or clockwise when the left or right buttons are pressed, respectively. To do so, a counter keeps track of the number of frames passed and on every third frame, it will update the rotation. The purpose of the counter is to slow down rotation and make it easier to control; otherwise, the ship's rotation speed was simply too difficult to make precise rotations necessary to aim at the other player and move agilely.

Projectile

The projectile was rendered as a 5x5 block of pixels, and its velocity and initial position were implemented with respect to the rotation of the ship during its rendering. Because of this, each of the rotations needed to be associated with a unique projectile velocity and initial position offset depending on the location of the ship's front. The different initial offsets of the projectile were obtained by opening the image that contained the ship rotation sprite and counting the pixels on the x and y axis to reach the tip of the ship, starting from the upper-leftmost corner. These x and y pixel values were added onto the position of the ship to accurately spawn the projectile at the front of the ship when fired. Originally, this feature was implemented to prevent players from colliding with their own projectiles after they fired them, causing them to unintentionally lose the game. Different velocities were used per rotation to ensure that the projectile would travel in the direction that the ship was facing. The magnitude of the projectile velocity was 144 sub-pixels per frame, where a sub-pixel represents a 16th of a pixel. A circle of radius 144 was used to calculate the x and y velocity values per rotation.

The projectile also had an output signal that communicated to the program whether it was still active on the screen or if it had despawned. A projectile despawns after a certain number of frames has passed after it was shot, which was controlled by a counter.

Projectile Controller

The projectile controller managed the magazine of projectiles that each player could fire. Each frame, the projectile controller would receive an input that stated whether the fire button was pressed. When pressed, the controller would check whether any of the three projectiles allocated to each player were active. If any of the projectiles were not currently active, then the projectile controller would send a signal to that projectile indicating that it should fire.

Additionally, the projectile controller had to prevent players from shooting all their projectiles at once when the fire button was held down. To do this, an intermediate signal was created that would store the last value of the fired signal every clock cycle. As long as the fire projectile signal was high and the last value was low, the projectile controller would send a signal to the projectile communicating that it should spawn.

Asteroids

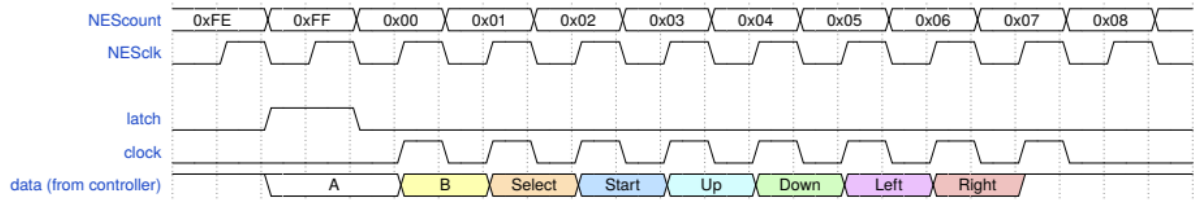
The asteroids aimlessly float about the screen, presenting a neutral threat to both players, and leading to a more interesting and diverse play. The asteroid module accepts an initial position and velocity at which to spawn the asteroid, and a reset signal can be used to reset an asteroid to its initial position, which is used during the transition from the start state to the in-game state. During the game, asteroids simply drift at their initial velocity around the screen.

This presents a potential issue: asteroids can move off the screen if the asteroid's positional logic is left unchecked. Ultimately, the naive solution to this problem of simply letting the position signals overflow naturally was eminently satisfactory. As we were using 10-bit numbers to represent pixel positions, and the x and y range is only up to the mid-1000 range, each asteroid was on screen about half the time.

With diagonal movement, the final result was relatively unpredictable movement since players could easily predict the locations of the asteroids while on screen, but would be very difficult to predict where they would reappear when drifting back on screen. And with three of them we got a good balance of giving the players space to move around, but also still having hazardous terrain that forces players to think fast and improvise.

NES Decoder

NES Controller Timing Diagram:

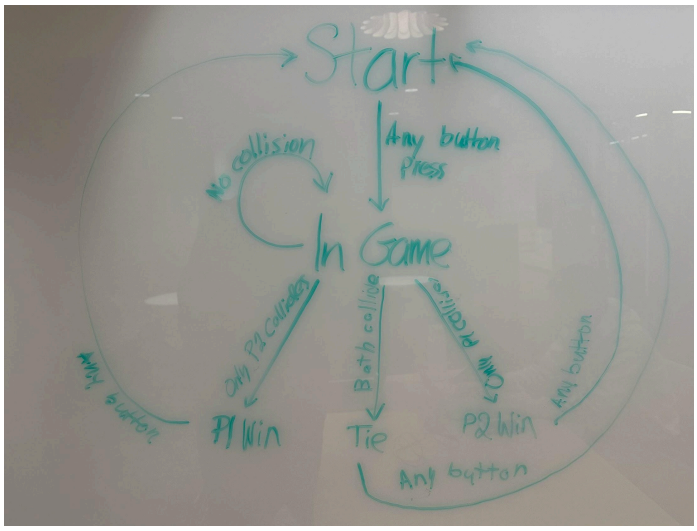


The NES decoder handled receiving data from the NES controllers and outputting the controls to the rest of the program. Using a counter, the NES decoder would read data into an intermediate vector signal with each bit representing a button on the NES controller every 8 clock cycles, which is the period of the NES controller. A shift register was used to store all the data read in, which stored input from all NES controller buttons since each button was only sent every 8 cycles. Furthermore, when the counter reached its maximum value, the latch signal was set to high, which signals to the NES controller that it should output the current button state.

Finite State Machine

We kept track of the game state by using a state machine.

Initial diagram of the game state finite state machine:



The possible states that the game could be in were:

State	Description	Following states
START	start screen displayed, main game disabled	GAME, when either player presses start
GAME	Main game displayed	TIE when both players lose P1W when player 2 loses P2W when player 1 loses
P1W	Player 1 win endscreen	START if either player presses start
P2W	Player 2 win endscreen	START if either player presses start
TIE	Tie endscreen	START if either player presses start

These were defined as 3-bit constants, so that we could output these states to other modules and use their defined values.
The triggers for changing the states were input. The triggers were (the falling of each player alive signal), the start button pressed from each player's NES controllers.

VGA Display

This is a simple module that controls the horizontal and vertical sync pulses for the VGA controller, as well as keeping track of the current row and column being rendered on the screen.

Results

These were our goals for this project, straight from the project proposal (along with what we completed):

State of the Project

These were our original goals for the project (and which ones we completed):
Minimum Viable Product:

- ☑ ~~2 asteroid-style ships~~
- ☑ ~~Tank-turning~~
- ☑ ~~Acceleration/deceleration~~
- ☑ ~~projectiles~~

- ☑ ~~shoot to kill opponent and reset game~~

Stretch Goals (time-based):

- ☑ ~~Start/Victory screen~~
- ☐ Health/Round system (lives)
- ☐ Sound
- ☑ ~~Additional mechanics (ties; collide with other player to both lose a life, e.g.)~~
- ☑ ~~Background and additional pixel art/graphical effects/theming~~
- ☑ ~~Asteroids or other terrain~~
- ☐ Two game modes (original asteroid game)

Current Issues

- Visual Artifacts

During our initial testing, some columns would appear to be swapped or mixed around in certain conditions (mostly when moving to a higher-valued vertical column). It turns out that our hsync was one value off, and this fixed the majority of our problems. However, there were still some similar-looking issues on certain monitors and iterations of our project. The current version of our project has some similar visual artifacts.

- Speed changes

The speed of the whole game appears to differ widely between flashes. We know that this cannot be misinterpretations of our PLL clock or our frame clock, as both would cause our VGA module to glitch out severely.

- Infinite life

If a player holds down the 'start' button while playing the game, their life value seems to permanently be held to 1. This is to pull off while in normal movement, even more difficult to think of during our relatively short game, useful for debugging, shows off the rendering behavior when two objects overlap, and is generally funny to find out as a player. However, we don't quite understand the cause of this behavior, so this is somewhat concerning.

- Reset velocity on start

When a player dies and quickly resets, it can be seen that they may still be moving. This is because the player's velocity is never reset between rounds. This adds some

interesting variation to the starting conditions of the game, and rarely leads to unfair situations like instant deaths. So, we decided to keep this behavior.

Reflection

We completed most of our achievable goals for our project! This could be attributed to having set our expectations reasonably and focusing on scalability of the final product. Once we achieved the minimum viable product, we focused on polishing what we had already and avoided adding complex features, such as sound and multiple lives for the ships. One massive improvement was going from 8 possible directions for a ship to 32 directions, which led to smoother rotations and a more satisfying ship movement. Also, we believe that another factor that greatly contributed to the accomplishment of many of our goals was our time management. Our team started the implementation of the final project very early, which enabled us to successfully set goals for when we wanted our specific features to be completed. Also, the effective communication between team members enabled us to resolve meeting time conflicts and problems with the program.

However, our development process left room for improvement. During our labs and initial collaboration sessions, there were some issues setting up our project so that it could be developed and tested on multiple different computers. Our solution was to have two teammates draft up a VHDL module in VHDLweb, and then two teammates would be working to integrate the previous day's drafts into our project. This had the unintended effect of needing a lot of code to be intensively debugged and even rewritten. Instead, we should have figured out the issues with our Radiant environments, and then worked on different

Work Division

Robi:

- VGA module
- Row-column/x-y positional offset into sprite index logic
- Rendering layering logic
- Collision detection logic
- Backgrounds
- State machine implementation
- Projectile controller module debugging
- Art direction

Kyle:

- Sprite pixel coordinate to linearized ROM indexing logic
- Pixel art to ROM conversions
- Asteroid Module
- General integration and debugging
- Generation of ship rotation images
- Acceleration of the Ship
- Sub-pixel position logic and additional rotations

Jacky:

- Pixel art
- Projectile Module
- Projectile Controller Module
- NES Module
- Asteroid Module
- Projectile Velocity and Offset Logic

Jishnu:

- State machine design and implementation
- Ship design
- Ship rotation behavior
- Ship movement and acceleration design
- Scripting (32 rotation speed calculation and VHDL)
- Projectile Velocity and Offset Logic