Project Overview

The intent of this project is to implement the classic Namco arcade game Dig Dug on an FPGA. In Dig Dug, the player controls a character, Dig Dug, who interacts with a subterranean environment while being pursued by Pookas and Fygars. Both Pookas and Fygars can kill the protagonist by contact and Fygars can also breathe fire. The goal of the game is to eliminate these enemies by either inflating them with an air pump that Dig Dug carries, or by leading enemies into tunnels and dislodge rocks to crush the enemies. The subterranean game map appears to allow both Dig Dug and his enemies to move continuously, but the x and y index of tunnels is actually restricted to a 14 x 15 grid. When only one enemy remains, it will attempt to flee to the surface instead of pursuing Dig Dug.

Dig Dug's movements are controlled by the player. As Dig Dug moves and create tunnels, a map of which portions of the grid have been tunneled to that point is continuously updated. The behavior of Pooka's and Fygars is controlled using multiple instances of two different modules. Events such as rocks falling, Dig Dug attacking, or Dig Dug dying are determined by a module governing collision logic. This module returns information regarding the events of the game to the other modules. Additional modules exist to regulate inputs, maintain the game map, generate appropriate graphics, decide which graphics to display and which audio to play.

Modules

Game State

game_map - Leif

Inputs	Outputs
• level_reset (game_state) • Tunnel_init (level_initializer) • Level_number(game_state) • dig_dug_location (dig_dug)	• tunnel_map

The map module stores and updates a map of which grid sectors have been tunneled out by dig dug. The map is stored as a 13x14 array of for bits. The array indices represent the X and Y index of each specific grid location, and each of the four bits represents the top, bottom, left, and right side of that grid square.

On initialization or level reset, the map module writes zeros to the whole array. Two clock cycles later, the module will write 1's to the appropriate locations in the array for the starting tunnels. This information is hard coded in the module, and the assignment lines used for this were generated by a python script from a list of tuples representing the grid locations of the starting tunnels. This was done for 8 unique levels, although the use of a python script to generate these lines of systemverilog allows the game to be rapidly expanded to more levels with minimal effort

or error.

The map module takes in dig dug's position in pixels and bitshifts his position to determine the grid square location. Based off of the direction and gird square location of dig dug, the map module will overwrite the portion of the map array where dig dug is located to indicate that the grid square has been tunneled. When any of the walls are demarcated as tunneled, the graphics will interpret the center of the grid square to be tunneled.

The map information is continuously sent to the pathfinding module to dictate the movement of enemies and govern the behavior of rocks.

game_state - Rachel

Inputs	Outputs
Inputs • clock	Outputs

• reset • dig_dug_state • pooka_state [3:0] • fygar_state [3:0]	• flee (unimplemented) • high_score • score • lives • level • game_over • level_reset • death_reset

Keeps track of the global game state. These include the score, lives, and level. Whenever a level is completed through the deaths of all enemies, this module also sends the reset signal to all entities to initialize the next level.

The score is incremented when an enemy is killed, 400 if they were inflated and 1000 if they were crushed by a rock.

The number of lives begins at 3 and goes down by one every time dig dug dies. Whenever dig dug is killed, a reset signal (death_reset) is sent to all entities to reset their positions (but not their states). Whenever the number of lives reaches 0, game_over is set to 1 until the game is reset.

The level begins at one on reset and increments every time all enemies are defeated. Whenever the level increments, level_reset is pulled high for a few clock cycles to alert the enemies to reset their locations and states to the new init values. The user can see the level number in the bottom left of the screen.

Level_initializer - Leif

Inputs	Outputs
• level_reset • level_number	• tunnel_init • pooka_loc_init x4 • fygar_loc_init x4 • rock_loc_init x4 • pooka_state_init x4 • fygar_state_init x4 • rock_state_init x4

The level initializer module stores the starting position of each rock, pooka, and fygar in addition to their states at the beginning of each level. Each entity starts in a baseline ‘normal’ state. To create these level layouts, photographs of various levels of the original dig dug game were scaled to a consistent size and overlaid with a grid to determine the positions of each entity in terms of the grid. This was performed using layers in Scribus, odd as that seems. I was familiar with the software and it quickly did what I needed it to. With the grid overlaid on the original level images, the locations of each entity were entered into a python script as tuples of (grid_x, grid_y) and the initial positions in terms of pixels were calculated. The script then generated the systemverilog lines needed to set the initial locations and states of each entity at the beginning of each level. When the level initializer module receives an init_level signal, it sets each entities initial conditions. Each instantiation of each entity takes the appropriate position and state information and, if an appropriate initialization signal is received, each instantiation will set the values from the level initializer as their values before starting the round. Instances of enemies and rocks not in use for a particular level are initialized to the death state.

Collision - Leif

Inputs	Outputs
• pooka_locations • fygar_locations • rock_locations • dig_dug_location • weapon_location • pooka_states • fygar_states • rock_states • dig_dug_state • fire_tip_location • Fygar_orientations • weapons controls	• collision_event x9 • fygar rock collision x4 • pooka rock collision x4

The collision module takes in the position and state of all entities, in addition to the position of dig dug's weapon, the tip of each fygar's fire, and the weapons controls. The collision module relies on many non-mutually-exclusive conditional statements to determine if an event has taken place. This module determines when an entity is crushed by a rock or inflated by dig dug, in addition to if dig dug is caught by an enemy, crushed by a rock, or burned to a crisp by a fygar.

Rock collisions test if any of the entities are within a certain proximity of the underside of a rock, and if that rock is in a falling state. If both conditions evaluate to true, the collision module will assert two values to the entity in question. The first is a collision code indicating that the entity has been hit by a rock. This will be used by the entities FSM to move between states. The other is a rock collision indicator, indicating which rock hit the entity. This is necessary because the behavior of a crushed enemy (ideally) depends on how far the rock falls, and it is necessary for the entity's FSM to reference the state of the falling rock.

Inflation collisions test the state of the enemy, the state of dig dug, and the relative position of dig dug's weapon and the enemy. If the weapon is in contact with the enemy and the state of the enemy is such that it can be inflated (i.e. it is not already dead or being inflated), and the controls input indicates that the attack control is high, then the collision module will send a collision code indicating that an inflation event has occurred to the specific instantiation of that enemy. This module posed a unique challenge due to integration and team communication issues pertaining to the nature of the weapons control signal and diverging assumptions of a proper dig dug model resulting from different experiences on different emulators. More details on inflation collision handling are available in the Fygar module description. The original implementation of this system is preserved as commented out blocks in the submitted code.

All of dig dug's potential deaths take place in one large conditional statement which has a unique alternative condition for all of dig dug's possible deaths. Rock collisions are detected in a manner very similar to how they are detected for other entities. Contact collisions rely on proximity testing and basic state testing (dig dug should not die from contact with a dead, crushed, or inflated enemy). Fire deaths are detected by determining if dig dug's x position is between a fygar in the fire state, and the position of that fygar's fire's tip. Each of these potential situations results in the collision module asserting a collision code indicating that dig dug has died.

Entities

Entities - Rachel

Contains the Dig Dug, Pooka, Fygar, and Rock blobs and connects all inputs and outputs directly between them and top_level. Purely for organization.

Pooka - Leif

Inputs	Outputs
• dig_dig_location • flee • speed • collision_event • tunnel_map • level_reset • death_reset • pooka_loc_init • pooka_state_init	• location • state

The pooka module is effectively a simplified version of the fygar FSM. Specifically, the pooka does not breathe fire. A detailed description of the fygar module is available, and for non-fiery purposes, the pooka should be considered to function in the same manner as the fygar module.

Fygar - Leif

Inputs	Outputs
• dig_dig_location • flee • speed • collision_event • tunnel_map • level_reset • death_reset • fygar_location_init • fygar_state_init	• pooka_location • pooka_state

A somewhat simplified diagram of fygar's FSM is presented at the end of this report.

The fygar module controls the state and movement of the fygar sprite. The Fygar module is a 12-state FSM with additional features included. The fygar module takes in a tunneling Boolean, a death reset signal, a level reset signal, an initial location and state, a collision indicator, an indicator to communicate which rock is squishing the fygar, a speed indicator, dig dug's y-position, and each rock's state. The fygar module outputs the fygar's position, the position of the tip of fygar's fire, the fygar's state, and whether or not the fygar is actively being inflated.

In the normal state, the fygar will move in the operating direction specified by the pathfinding module one pixel every N clock cycles if and only if the fygar is centered in a grid block. This allows the pathfinder to continually update the direction the fygar should be moving without regard for what state the fygar is in. By only changing the actual direction the fygar is moving in when the fygar is centered in a grid square, the fygar is able to cleanly make corners and avoid wandering into the rock when it is not in tunneling state. If the fygar is in the normal state and

the tunneling Boolean is pulled high, the fygar will transition to the tunneling state.

In the tunneling state, the fygar will constantly update its direction of movement to reflect what the pathfinder is currently specifying. This allows the pathfinder to alternate directions of movement rapidly and without regard for the fygar's position in order to give the tunneling fygar the appearance of the up/down, then left/right movement in the classic dig dug game. One the fygar has reached another tunnel, the tunneling Boolean will be pulled low and fygar will return to the normal state.

In both the tunneling and normal state, the number of clock cycles it takes before the fygar moves by a single pixel is determined by a case statement in an always@(speed) block. The speed signal can take on five distinct values. Each different value of the speed signal will linearly detract the parameter SPEED_INCREMENT from the parameter MIN_SPEED to set the N number of cycles between a single pixel movement. If speed adjustment had been implemented in the game state module as initially planned, this would have enabled the speed of enemies to increase throughout gameplay, making the game more difficult at higher levels. SPEED_INCREMENT and MIN_SPEED were parameterized to allow for the game to be tuned after completion.

If the fygar is in the normal, tunneling, charging, or fire states and receives a collision signal indicating an inflation event, the fygar will transition to inflating_1 state. Ideally, upon a timer expiring and the continued or additional pulse input to the weapon control signal in the collision module, the fygar would then transition to the inflating_2 state. Similarly, upon a timer expiring and lack of a continued or additional pulse input to the weapon control signal in the collision module, fygar would transition back to the normal state. The impetus for transition from inflating_2 to inflating_3 and eventually inflating_4 would be controlled in the same manner, with the exception that upon exiting the fourth inflating state with a weapons signal, the transition would be to the death state. The timer used for these transitions was parameterized to allow for the game to be tuned after completion. Unfortunately, due to module integration issues and communication difficulties surrounding the nature of the weapons control signal (i.e. when it would be asserted, the form it would take, etc.) and diverging understandings of how the dig dug game should operate due to experience playing different ports, this escalatory and de-escalatory state behavior was eventually put aside in favor of a one-shot, one-kill, "dirty dig dug Harry" functionality in order to preserve playability.

If a fygar is moving horizontally and is in the normal state (i.e. not being inflated, tunneling, squished, dead, or already breathing fire.) there are two alternative conditions which can transition the fygar into the charging (preparing to breathe fire) state. The first is a 'dumb' control based on a long timer. The intent behind this is to allow the fygar to breathe fire at seemingly random, pointless intervals as he does in the original game. The second condition which can result in fygar breathing fire is a targeted firing mechanism that tests if dig dug's y position is roughly similar to the fygars. This condition also requires a few of the middle bits in dig dug's y position to be greater than a given value in order to add a bit of psuedorandomization to the targeted firing and prevent the fygar from continuously breathing fire while dig dug is in a similar y-position. These functions originally relied on a $random 32-bit value to determine when fygar should breathe fire, but that approach led to (undesirably) unpredictable behavior and was replaced with a more simplistic psuedorandomization effect which, in my opinion, still emulates

the original game fairly well.

When fygar does enter the charging state, a timer is started. When that timer runs out, the fyar will advance to fire_1, fire_2, and fire_3 on a time-schedule. In each of the fire states, the fygar's fire's tip's position will change to reflect a longer fire sprite. The fygar's fire's tip's position is also sent to the collision module to determine if dig dug was in contact with the fire.

When fygar is in the charging, fire, inflating, or normal state (i.e. not tunneling or dead), the collision module may report a rock collision. In an ideal implementation, the fygar will transition to the squished state where it's y-position will be decremented at the same rate as the rock that is pushing it down in the tunnel system until the rock progressed to the death state. The rate at which the fygar falls was parameterized to allow the rock-fall speed to be used for the fygar's (identical) fall speed. This functionality suffered from an issue in the feedback system used by collision to identify which rock is crushing the fygar, and the fygar's state transition rules. The original implementation of this system is preserved as commented out blocks in the submitted code. The crushing functionality was then replaced with a timer during which the fygar would appear as a crushed, falling sprite before transitioning to the death state.

rock - Rachel

Inputs	Outputs
• clock • level_reset • falling • init_x • init_y • init_state	• x_out • y_out • state

Controls the logic of the rock entity. The rock entity is static as long as there is no tunnel below it. When the tile below it is tunneled out, the rock entity will start a timer after which it will begin free fall, where it will move downward until it encounters a wall. Any other entities hit during the rock's freefall will die. When the rock hits a barrier, it will play a breaking apart animation and disappear.

Whenever the rock receives a level_reset signal, it sets its own x, y, and state to those defined by init_x, init_y, and init_state, respectively. The rock will maintain a 'normal' state until it receives a 'falling' signal, after which it will move to the 'waiting' state and begin a timer. After that timer expires, the rock will enter the 'falling' state and move downward. While the rock is in the 'falling' state, collision with pooka, fygar, and dig dug will kill them. When falling is pulled low again, the rock will enter the 'breaking' state and finally the 'dead' state after a certain number of clock cycles.

dig_dug - Rachel

Inputs	Outputs

• clock_in • level_reset • death_reset • move • attack • facing_tunnels • blocked • collision_event	• location_x • location_y • weapon_x • weapon_y • state • direction • weapon_state • moving • digging • inflate_event

Controls the location and state of the player character, Dig Dug. Dig Dug is controlled by the player inputs as passed in by the control module. When Dig Dug moves into a non-tunneled tile that does not contain a rock, he tunnels through it, therefore changing the game board. Aside from moving in the four cardinal directions, Dig Dug can also attack by sending a telescoping tether in the direction he is facing. If the tether collides with an enemy, the enemy and Dig Dug will be frozen in place and the player can repeatedly input or hold down the attack command to inflate and pop the enemy.

Upon a reset, dig_dug will be returned to the middle of the screen and have his state reset back to 'normal'.

The move inputs determine Dig Dug's movement (up, down, left, right, none), but Dig Dug can only move along a row or column he is properly aligned with, so if they are not aligned they must continue moving along a direction until they are aligned with the grid. The blocked variable informs Dig Dug to the presence of rocks to his immediate left, right, up, and down, disallowing movement in that direction. When Dig Dug is moving, even if he is hitting a barrier, 'moving' should be pulled high to inform the graphics module to play the walk animation.

Whenever commanded to attack, Dig Dug will send out his tether and be locked in place until it hits a barrier, updating weapon_x and weapon_y and setting the weapon_state to 'deployed'. If Dig Dug's weapon encounters an enemy, until he moves again, he will be able to input the attack command again to pull inflate_event high for one clock cycle and inflate the enemy by one step. Dig Dug will then ignore the attack input until a certain number of clock cycles have passed.

If Dig Dug becomes squished or killed, as notified through a collision event, he will maintain either the 'squished' or 'dying' state for a certain number of clock cycles, after which he should enter the 'dead' state.

The variable facing_tunnels informs Dig Dug of tunnel barriers in the direction he is facing. If it indicates that the closest tunnel (the one that Dig Dug's 'nose' is in) is indicated as closed off, digging should be high to inform the graphics module to play digging animations. This variable also controls how far Dig Dug's weapon goes until it retracts, which is either until it hits the closest blocked tunnel or until it hits its max length.

pathfinding - Rachel

Inputs	Outputs

• clock • reset • flee • seed • tunnel_map • x_in [12:0] • y_in[12:0] • dig_dug_direction • rock_state	• direction_out [7:0] • tunnelling [7:0] • falling [3:0] • facing_tunnels • blocked

Handles all entity decisions that require use of the tunnel map. For pooka and fygar, this module will inform the entity which direction it should go and if it should be tunnelling or not. For rock, it should inform it whether it should be falling or not. For Dig Dug, it informs the module of the tunnels in front of it and the presence of rocks to his sides.

For Pooka and Fygar's pathfinding, the module will first figure out what directions it can move in (where the tunnel is open) and then what directions it wants to move in (first priority: not turning around, second priority: moving naively towards Dig Dug). Whenever moving through these priorities, if the path becomes ambiguous between two options, the decision is made by the pseudo-random seed boolean input.

For the falling output, the output is high if the space below the rock is open (not a wall).

The facing_tunnels variable will be high for the first tunnel if the tile that Dig Dug's 'nose' is in is non tunnelled and high for the tunnels after it if they are not dug in the direction perpendicular to Dig Dug's. The blocked variable should be high for the directions where Dig Dug's side is touching a rock (corners do not count).

offset_coords - Rachel

Inputs	Outputs
• x_in [12:0] • y_in [12:0]	• x_out [12:0] • y_out [12:0]

A helper function for the pathfinder. Converts the coordinates of Dig Dug and all Pooka, Fygar, and Rock entities from using the background origin as a reference point to using the top left corner of the playfield as a reference point. To assist with pathfinding, it also clips the y of any entity between the surface and the first set of normal tunnels to 1, so for any location where an entity can move horizontally the y value will be divisible by 16.

User Interface

graphics - Rachel

Contains the sprite blobs. Mostly acts as an organizational wrapper for the blob

modules, but also determines the rendering order of the different blobs, with the background being on the bottom, followed by the tunnels, entities, lives, and text modules. Outputs the finalized pixel.

dig_dug_blob, pooka_blob, fygar_blob, rock_blob

Inputs	Outputs
• clock • reset • moving (Dig Dug only) • digging (Dig Dug only) • direction • state • hcount • vcount • x • weapon_x (Dig Dug only) • y • weapon_y (Dig Dug only)	• pixel

The entity blobs take in the location and state information from the respective entities as well as the hcount and vcount from the vga module and output a pixel to be rendered at the given coordinates. For each entity, all of the different sprites comprising the animations are stored in the BRAM as COE files (one image file and 3 color maps, as generated by the staff provided python script). Based on the state input, the module selects which of those sprites to render. An internal counter is maintained to alternate between sprites for animated states such as walking. The direction input helps determine if the sprite needs to be rotated or mirrored from the source file before being rendered. Note that due to all of the calculations that need to be made prior to rendering the output, the system needs is pipelined and a delay is added to the hsync values to properly match with the rest of the image.

bg_blob - Rachel

Inputs	Outputs
• clock • debug • hcount • vcount	• pixel

The bg blob takes in the hcount and vcount from the vga module and outputs a pixel to be rendered at the given coordinates. The debug pin can be pulled high to swap the background image out for one with the grid overlaid. Due to the delay in fetching values from the BRAM, a delay parameter is also added to the hcount.

game_over_blob, lives_blob - Rachel

Inputs	Outputs
• clock • debug • hcount • vcount • lives (lives blob only) • game_over (game over blob only)	• pixel

The game over and lives blobs are the simplest blob modules, as they simply render a static image from that stored in BRAM. Which sprites should be rendered is determined by the lives and game_over variables.

tunnels_blob - Rachel

Inputs	Outputs
• clock • reset • hcount • vcount • tunnels	• pixel

The tunnels blob takes in the hcount and vcount from the vga module and outputs a pixel to be rendered at the given coordinates. At each grid place of the tunnel map, the module determines which tunnel (if any) to render and what rotation needs to be applied to the sprite to match what is indicated by the tunnel map. Since each tunnel is 16 pixels wide, conveniently the tunnel index can be easily determined by offsetting the origin and bit shifting by 4, and the (x,y) of the point within each individual tunnel sprite can be found with those 4 least significant bits. Due to the very large number of steps going into determining the correct output pixel, this blob is carefully pipelined and the delay parameter added to hcount.

score_blob - Rachel

Inputs	Outputs
• clock • reset • hcount • vcount • score	• pixel

The score blob takes in a score in binary and renders it in decimal on the screen. The

module converts to decimal using the binary_to_decimal module and renders the 7 digit outputs in turn. For digits returned with a value of 10, no number is rendered.

binary_to_decimal - Rachel

Inputs	Outputs
• clock • reset • score	• digits [6:0]

Takes in a binary representation of an integer and outputs the 7 decimal digits used to represent them in the score blob. For numbers higher than 9,999,999, the output is clipped to 9,999,999. The module works by calculating the most significant decimal digit's decimal value through a series of inequality comparisons, then subtracting that digit's value (i.e. subtract 2,000 from 2,456) from a running total and repeating until the ones place.

xvga - (preexisting)

Inputs	Outputs
• clock	• hsync • vsync • hcount • vcount • clock • blank

Performs the same function as in the pong lab. Takes in a clock and outputs hsync, vsync, hcount, vcount, clock, and blank to send to the various blob and graphics modules.

controls - Rachel

Inputs	Outputs
• clock • reset • up • down • left • right • attack • use_accel • accel_x_sign • accel_y_sign • accel_x • accel_y	• move_out • attack_out

Handles input signals to the FPGA and translates them into signals for Dig Dug. Debounces all button inputs (up, down, left, right, and attack). If use_accel is high, the module will use the accelerometer inputs to determine the direction, otherwise the face buttons will be used. When the face button controls are used, the direction output is prioritized as up, down, left right. When the accelerometer is used, the direction with the highest magnitude of displacement is used, and when all directions are below a set threshold, move_out is set to indicate no movement. Note that + y for the accelerometer input is right for Dig Dug's controls and + x is up. The signal attack_out is simply tied directly to the debounced attack input.

accelerometer - course staff, modified by Rachel

Inputs	Outputs
• clock • miso	• sclk • mosi • csn • accel_x • accel_y • accel_z • accel_x_sign • accel_y_sign • seed

Translates the signals from the FPGA's onboard accelerometer into filtered x, y, and z magnitudes and direction indicators. This is modified from the accelerometer code provided by the course staff. The seed output is the least significant bit of the raw z acceleration and is used for pseudo-randomness in the pathfinder module.

Audio Decision - Leif

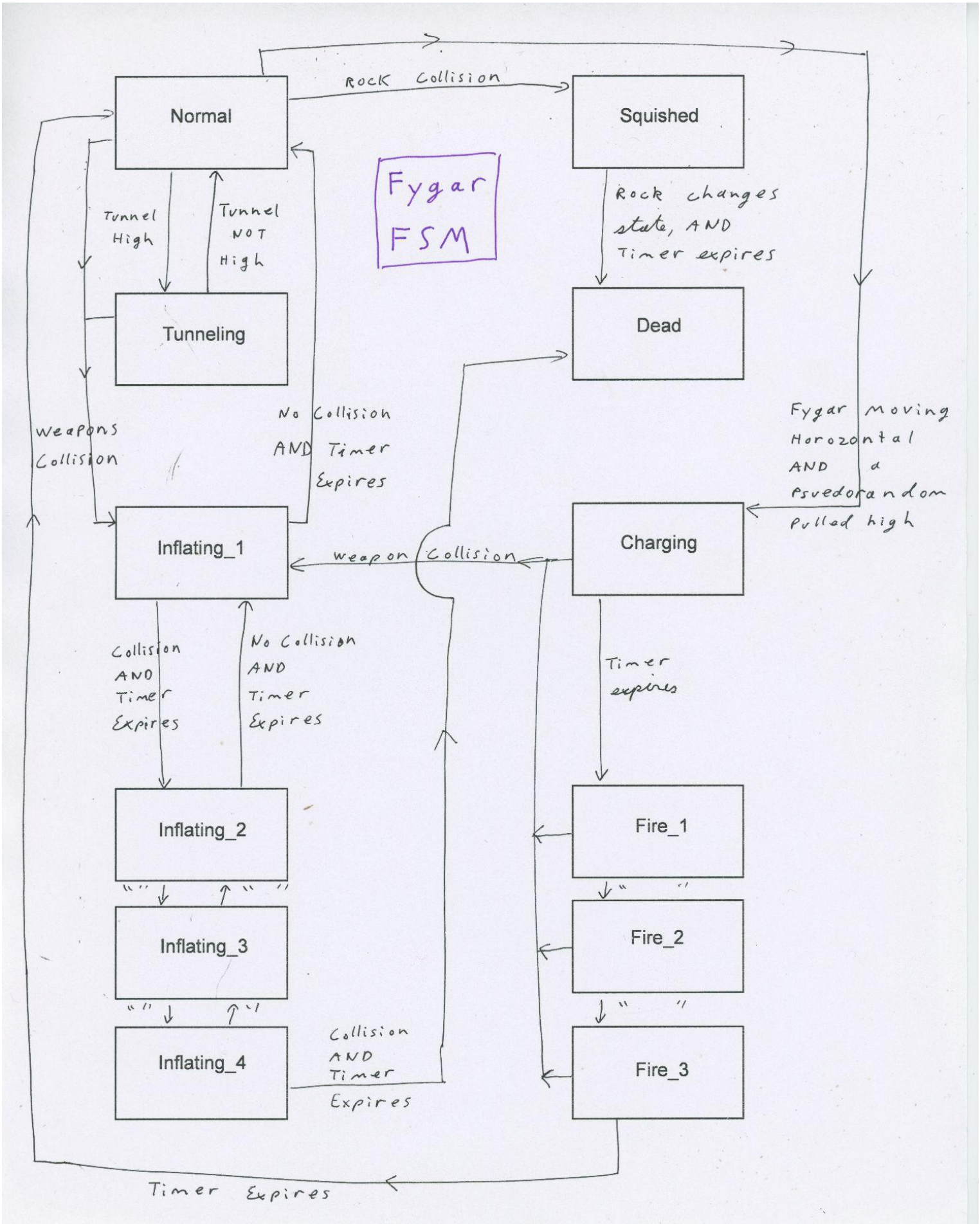
Inputs	Outputs
• Dig_Dug_loc • Death_contact • Death_fire • Death_squish • Fygar_fire	• audio

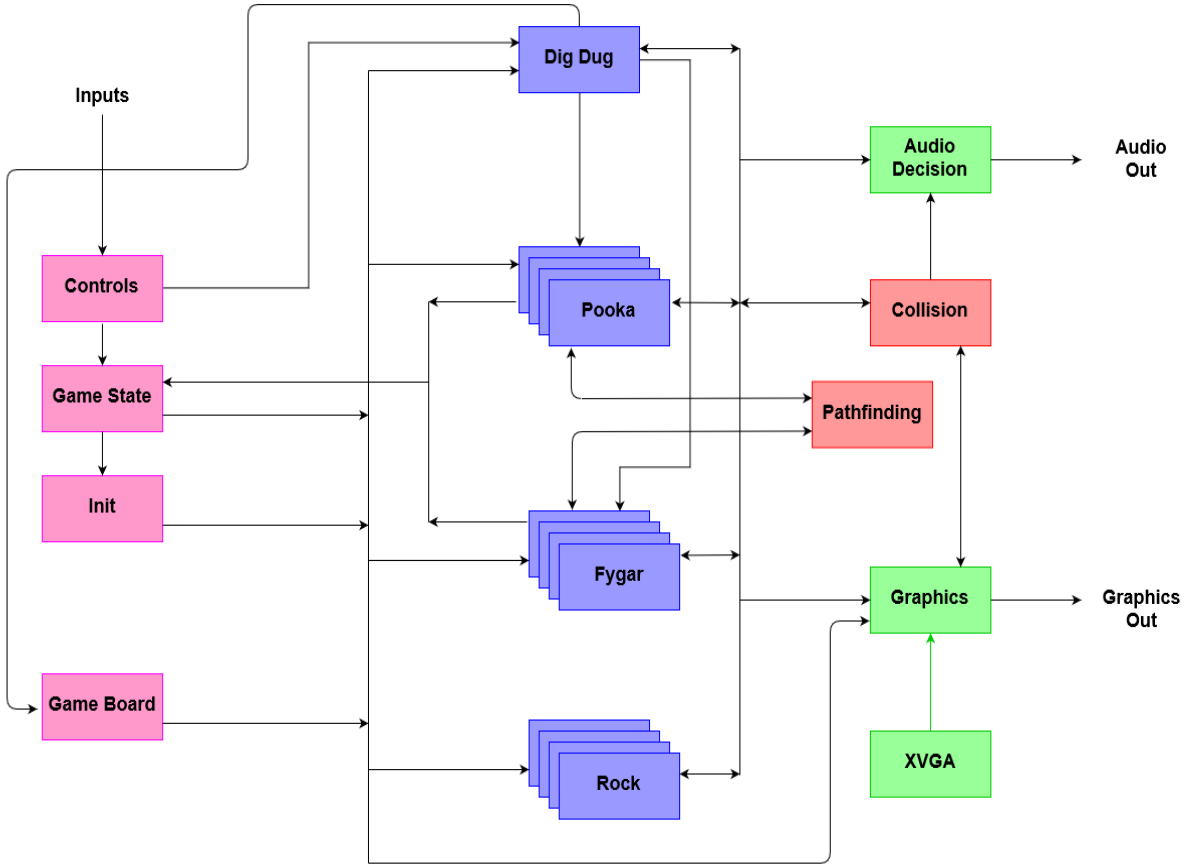
Audio functionality was ultimately scrapped in favor of focusing on game functionality. The planned implementation for audio is included here. Due to the limited number of sounds required, each would have been placed at a different address on the SD card, and upon game initialization, the audio module would sequentially read the first 1024 bytes of each sound and store said bytes in separate 1024 byte FIFO's. Using simple conditional logic, the module would determine which sound should be played. When a particular sound is to

be played, the relevant FIFO would be engaged, as would an address counter loop. After the first 512 bytes are pulled from the FIFO, the address counter would be incremented and the next 512 bytes would be pulled from the next sector on the SD and used to populated the tail-end of the FIFO. Audio samples would be pulled from the FIFO every Z clock cycles, with Z being determined by the ratio of the clock rate and the sample rate of the audio file.

Summary

By using the FPGA, we are able to run enemy logic, game state, and graphics rendering in parallel. Though this presents unique challenges in synchronization, it also allows us to meet our timing requirements with ease. We were able to create a small approximation of the original Dig Dug with some features scaled back within 6 weeks. Though there were many rough patches and communication issues as we attempted to develop features in time, the resulting product runs well and can be clearly recognized as Dig Dug.


Fygar FSM
Normal
Squished
Rock Collision
Tunnel High
Tunnel NOT High
Rock changes state, AND Timer expires
Tunneling
Dead
Weapons Collision
No Collision AND Timer Expires
Fygar moving Horozontal AND a psvedorandom pulled high
Inflating_1
Charging
weapon Collision
Collision AND Timer Expires
No Collision AND Timer Expires
Timer expires
Inflating_2
Fire_1
Inflating_3
Fire_2
Inflating_4
Collision AND Timer Expires
Fire_3
Timer Expires


Inputs
Dig Dug
Audio Decision
Audio Out
Controls
Pooka
Collision
Game State
Pathfinding
Init
Fygar
Graphics
Graphics Out
Game Board
Rock
XVGA

Text