ENSC 452 Individual Report

Chris Rosenauer
301 400 868
ENSC 452
April 12, 2023

Introduction:

For this project Jake and I created snake. My contributions were largely related gameplay and rendering logic. I had some previous knowledge of a few of my contributions (such as rendering and real-time software), however I had little experiencing designing such logic from the ground up. Through this project I gained a lot of experience in making software design decisions, writing performant low-level code, and building code that was maintainable and usable by another person.

Literature Review:

1.) E. Angel and D. Shreiner, *Interactive Computer Graphics: A top-down approach with WebGL*, 7th ed. Person, 2015.

This text supplied a foundation of knowledge for implementing graphics rendering and collision detection. While this book mainly focuses on 3D rendering, the sections on rasterization provided some insight and inspiration on rendering 2D sprites. This knowledge was used when creating this project's rendering code. Additionally, this book covers some basics on collision detection. This knowledge was used when creating this project's collision detection code. I trust the accuracy of the knowledge in this source due to it being used in computer graphics classes here at SFU.

2.) *"Why are memcpy() and memmove() faster than pointer increments?," Stack Overflow, 01-Jul-1958. [Online]. Available: https://stackoverflow.com/questions/7776085/why-are-memcpy-and-memmove-faster-than-pointer-increments?noredirect=1&lq=1. [Accessed: 10-Feb-2023].*

This text is far less formal, and of arguable relevance to the target hardware. The article discusses the rational of the speed of memcpy compared to iterative assignments. This knowledge was used to improve the performance of the rendering code when its earlier iterations were found to be insufficient. In spite of the source's questionable relevance the knowledge in the source proved useful.

3.) "xgpio_intr.c File Reference," *GPIO: XGPIO_INTR.C file reference*. [Online]. Available: https://xilinx.github.io/embeddedsw.github.io/gpio/doc/html/api/xgpio__intr_8c.html. [Accessed: 10-Feb-2023].

This text provided information and usage on software interrupts which I used in created the ultimately scrapped push button-based interrupts. I believe this information to be relevant and accurate as it was produced by the team who developed the used hardware and libraries.

4.) Aguila, Z. (2020, January 16). Vivado and Xilinx's Vitis Tutorial: VGA Controller. ENSC 452 Tutorial, Simon Fraser University.

The VGA tutorial provided through canvas provide the knowledge needed to create video outputting hardware, and a general workflow for creating code to output to the VGA display. That being said, I don't recall explicitly using any code from the VGA tutorial. I believe this information to be relevant and accurate as it was provided by the course and written specifically with the target hardware and tools in mind.

Project Contributions:

My contributions to the project can be summarized through the following milestones:

Milestone 1:

- Created a hardware block to facilitate drawing through VGA
- Created a first pass of gameplay sprites
- Created MATLAB helper scripts to store sprite data in C++ header files
- Created code to accept and interpret interrupt-based push button input
- Created code for sprite rendering

Milestone 2:

- Implemented the snake's movement
- Implemented food spawning and repositioning
- Implemented collision detection when the snake collides with a piece of food or the snake's body
- Implemented logic to reset the game upon collision with the snake's body

Milestone 3:

- Expanded collision detection to detect when the snake's head collides with the bounding walls

Milestone 4:

- Created a game over animation
- Added a game over screen

Milestone 5:

- Created helper functions to draw numbers as sprites
- Implemented showing the current score and game time during gameplay
- Added the player's score to the game over screen
- Implemented storing high scores, and showing the high scores on the high score screen
- Reworked the snake's sprites and movement animation such that the direction of the each the snake's component's movement is apparent through the sprite

Milestone 6 (Final Demo):

- Created sprites for the alternate snake colors (red, and blue)
- Created sprites for the alternate food sprites (orange, cherry, meat, and cheese)
- Created helper functions to obtain the active snake and food sprites
- Implemented the hard mode game option

- Expanded the options menu to include setting hard mode, changing the snake color, and changing the food sprite
- Applied the sprite retrieving helper functions to the gameplay rendering code
- Created new snake sprites to animate sprite movement between grid squares
- Added logic to draw the above snake sprites
- Updated the snake's death animation to be more visually impressive

Milestone 1:

Hardware:

The hardware block for the first milestone was very similar to the hardware block of the VGA tutorial with one difference, push buttons were added to allow for interaction with software so I could cycle between test screens.

Special thanks to Matt Whitehead who helped me debug my hardware block when I was having difficulties connecting the push buttons to the rest of the VGA tutorial hardware.

Gameplay Sprites:

The first pass of the gameplay sprites was entirely based on the visuals provided in the project proposal document. These sprites mainly focused on gameplay sprites, but also included some text. The specific sprites include the following: the snake's head, the snake's body in both a light and dark color, the snake's tail in both a light and dark color, the apple food sprite, a repeating background tile, sprites for the numbers 0 – 9, a line divider for menus, and text for the menu options "ADJUST", "DECREASE", "INCREASE", "VOLUME", "HIGH", "SCORE", and "START". Sprites which would need to be oriented (such as the snake's head and tail) were rotated and duplicated as separate images. This choice was made to avoid the need to create sprite rendering code.

MATLAB Helper Scripts:

MATLAB helper scripts were created to store sprite image and dimension data in a C++ header file. There were three main motivations to this choice:

1.) The sprites used came in various dimensions. To render these sprites properly we needed some way to obtain sprite dimensions.
2.) I expected the project to have a large quantity of sprites. The process in the VGA tutorial of loading 5 images onto the board using the XSCT console was lengthy. This method of upload images to the ZedBoard would clearly not scale well.
3.) I simply disliked uploading images to the ZedBoard through the XSCT console and wanted a way to avoid this process.

The MATLAB helper scripts were very simple. The scripts would accept a folder path, read through all images in the provided folder, and output the following information:

1.) A 2D constant integer pointer array of the sprites' graphical data
2.) A constant integer array of sprite heights
3.) A constant integer array of sprite widths
4.) A constant integer enumerator of the sprites. Each element of the enumerator is the sprite image's name, less the graphical extension.

Of note, the enumerator was generated such that the enumerator could be used as an index for the dimensional and graphical arrays to access that specific sprite's data.

Initially, the MATLAB helper scripts outputted the arrays and enumerators to the console. However, this was clearly a bad idea fueled by my sloth and later the helper scripts were corrected to output the desired data to a C++ header file.

As an additional note, the helper scripts could have been further improved. C++ handles 2D arrays in such a way where every inner array must be the same size, meaning that the memory allocated for each sprite must be the same. Because of this, the graphical data headers contained a lot of unnecessary padding. This padding never proved issue for our project, however it was something I kept track of, just in case we ever needed the extra memory.

<u>Push Button Inputs:</u>

Push button inputs, while not strictly part of this milestone's deliverables, were added for the purpose of accessing testing code. The push button input code was based on the code provided in tutorial *2B. Creating a Zynq System with Interrupts in Vivado* as provided in the audio tutorial.

The push buttons would generate a software interrupt, which would update a buffer that scored which buttons were pressed and if a button had just been pressed. A function was available to access this input buffer. Calling this accessor would update the buffer, consuming the fields that indicates that a button's input was just pressed. Button inputs were separated to indicate if the button is currently pressed and if the button was just pressed to facilitate edge-specific inputs such as menu navigation, and value-based inputs such as supplying snake movement. In hindsight, this separation of usage may not have been necessary.

<u>Graphics Rendering Code:</u>

The graphics rendering code was built with a bottom-up methodology (unlike the rest of my contributions) went through three main iterations until it reached a state which was sufficiently performant. All iterations shared the following qualities:

- Sprite data is organized in Sprite structures. Sprite structures are not created by a user, but are rather accessed by calling a helper function, passing through the enumerator of the desired sprite. Sprite structures contain a sprite's graphical data, height, and width.
- Sprites are initially rendered to a "canvas," which in practice acted as an intermediate framebuffer.
- After the sprites are drawn to the canvas, the canvas is copied to the VGA framebuffer several times to trigger cache flushing.
- Rendering calls clip the passed sprite to be within the screen's dimensions.

The qualities of the three iterations are as follows:

1.) Sprites are upscaled greatly in software to comfortably fit the 1280 x 1024 display. Individual sprites are drawn pixel-by-pixel onto the canvas. Alpha for each pixel is checked to determine if the pixel should replace the current pixel at the pixel's location.
2.) Sprites are upscaled to be twice their height and width and drawn to a 640 x 480 display. Sprites are written to the canvas pixel-by-pixel and visibility is checked, just as in the previous iteration.

3.) All sprite images have been resized to be twice their original resolution. Software upscaling is no longer supported. Sprite visibility is no longer supported. Sprites are drawn to the canvas line-by-line using memcpy.

<u>Iteration 1:</u>

The first iteration of the rendering code was largely based on the rendering code I used for the lab test. Software upscaling was initially supported. The motivation behind this was to reduce memory usage, as I feared the 512MB of memory provided by the ZedBoard may be insufficient. This was not the case. Sprite transparency was supported in attempt to make the game more visually impressive. The idea was that some sprites (such as the food sprites) would not fill their entire sprite with graphical data, and it would be more visually impressive to have sprites overlap. For example, if the snake was moving to eat and apple the snake's head would be visible through the gaps in the apple's sprite.

Iteration 1 quickly proved insufficient, as this iteration was only able to reach a frame rate of 4-5 Hz.

<u>Iteration 2:</u>

The second iteration attempted to preserve the functionality of the first iteration, while improving performance. The screen resolution was lowered to the minimum resolution supported to VGA, 640 x 480. To accommodate the lower resolution sprites were scales to a lesser extent.

Iteration 2 also proved insufficient and was able to reach a slightly higher, but insufficient frame rate of 7-9 Hz.

<u>Iteration 3:</u>

Iteration 3 introduced three main changes from the previous iterations:

- Sprites were no longer scaled. Instead, the raw images were upscaled by a factor of two in each direction.
- The rendering code no longer checked for pixel transparency when drawing pixels.
- Sprites were written to the canvas line-by-line using memcpy.

Iteration saw significant performance improvements and provided a frame rate of about 100 Hz. This improvement is largely due to the highly efficient memcpy, which saw great performance compared to drawing each pixel individually using looped assignments. However, this approach had a major drawback. Since sprite rendering doesn't support transparency, gameplay sprites must include background data. This drawback will become apparent in later milestones.

<u>Sprite Data Retrieval:</u>

Sprite data can be retrieved based on the sprite's corresponding enumerator as generated by the MATLAB helper scripts. Sprite retrieval gives a structure of the sprite's data which can be directly passed to rendering functions. The logic of sprite data retrieval code was made in such a way that the sprites' graphical is not replicated. This absence of replication was made to reduce memory usage.

Testing:

Testing for milestone 1 was primarily tested using automatic unit tests. The rendering code was also profiled by measuring the time it takes for the rendering code to render a screen with 300 sprites 100 times. Other informal tests were used, such as: a graphics test screen which renders every sprite stored in the game's code, and a demo screen showing what a rendered frame of gameplay may look like. The unit tests were created with the following test cases:

1.) Ensuring the screen could be properly cleared
2.) Ensuring that sprites drawn at position (0, 0) are drawn correctly
3.) Ensuring that multiple non-overlapping sprites are drawn correctly
4.) Ensuring that multiple partially overlapping sprites are drawn correctly, and that the sprites are drawn in the correct order
5.) Ensuring that sprites which are positioned fully off of the game screen are not rendered
6.) Ensuring that sprites can be rendered onscreen after attempting to render a fully offscreen sprite

Some helper code needed to be created for these tests, such as a function to check if a subsection of the canvas or framebuffer matched a desired sprite.

Ensuring Compatibility:

The rendering code was made compatible due to its modularity. All rendering and sprite obtaining code is self sufficient and easily accessible. The self sufficiency of the rendering code allows for rendering code and logic to be altered and used without the risk of breaking another component of the code. Low-level functionality of the rendering code, and raw sprite and graphical is hidden, making the code easier to use. The sprite obtaining code is compatible with the rendering code; sprite structures obtained by the sprite obtaining code can be directly passed to rendering code.

Reducing Complexity of Milestone 1:

Initially, the rending code was going to be more complex, and closer to what could be seen in a proper rendering engine. The rendering code would be largely automatic, without the need to make specific calls to draw sprites to the canvas or to draw the canvas to the frame buffer. The rendering code would iterate through all game objects stored in a managed entity buffer. Game objects would contain a layer flag (background, foreground, UI, etc), and would be rendered in order based on their layer flag. However, I quickly realized that this approach to rendering would require a large amount of game object management that was unnecessary for the scope of this project and would have likely made the codebase harder to work with overall.

Milestone 2:

Snake Data Management:

Snake components were stored in a static array. We know how large the game field was, and as such knew the max number of snake components that could be on screen at one time. I took advantage of this to avoid the need to manage dynamic memory. Every snake component stores information about their position, direction, and if they have been initialized yet. The components' position was used for movement and collision detection, direction was used for movement, and the initialization field was used to ensure that only snake components which were active in the game were rendered or queried for collision. The snake data is reset upon starting a new game.

Snake Movement:

Every snake component except the snake's head moves with respect to the snake component next closest to the snake's head. Essentially, each snake component takes the position and direction of the component next closest to the snake's head. The movement and direction of the snake's head is set such that both are in the direction of the user's input.

I cannot recall if it was which of us made this change, but around this time we decided to have the user input be received by keyboard via the UART terminal. The physical sensation of inputting movement using the keyboard led to a much stronger user experience compared to using the onboard push buttons.

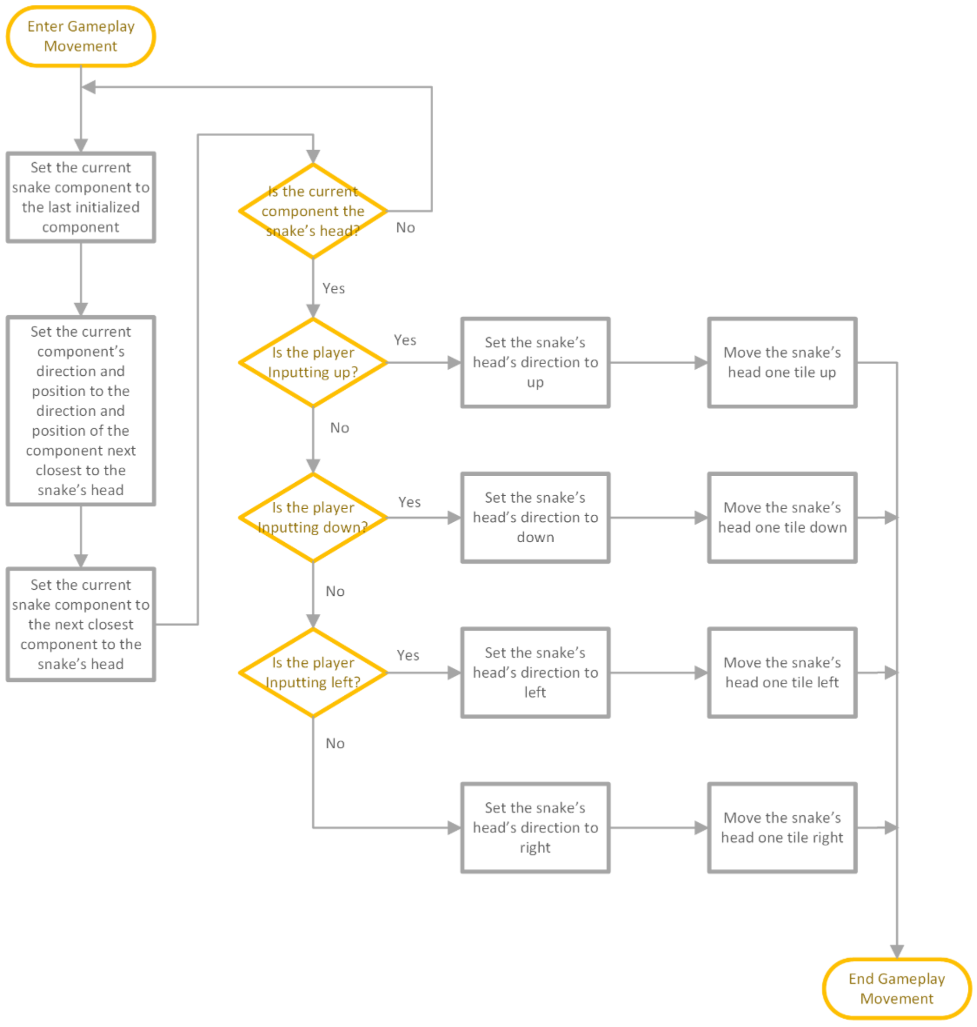

Enter Gameplay Movement
Set the current snake component to the last initialized component
Set the current component's direction and position to the direction and position of the component next closest to the snake's head
Set the current snake component to the next closest component to the snake's head
Is the current component the snake's head?
No
Yes
Is the player Inputting up?
Yes
No
Is the player Inputting down?
Yes
No
Is the player Inputting left?
Yes
No
Set the snake's head's direction to up
Set the snake's head's direction to down
Set the snake's head's direction to left
Set the snake's head's direction to right
Move the snake's head one tile up
Move the snake's head one tile down
Move the snake's head one tile left
Move the snake's head one tile right
End Gameplay Movement

Figure 1: Flowchart of the Gameplay Movement Logic

Snake Growth:

The growth of the snake sees the initialization of a new snake component. During gameplay the position and direction of the tail of the snake is stored before any gameplay logic which alters the position and direction of the snake. If the snake grows during that gameplay loop the newly created snake component takes that recorded position and direction.

Food Spawning:

Jake's milestone to integrate hardware-based pseudo-random number generation was not until much later in the project. To compensate for this, I emulated pseudo-random number generation by generating several pseudo-random numbers and storing them in an array. Whenever I needed a random number to obtain the food's position, I would index this table and increment the index to reference this table.

Whenever the snake eats the food, the food is repositioned to a new location. In the case where the food attempts to spawn on top of the snake I would iteratively move the food across the row, then down the column until I found a tile which did not contain a part of the snake.

Collision Detection:

Collision detection was performed using simple box checks. I would create box shapes using the positions and dimensions of game objects and would perform simple if statement to determine if these two objects were overlapping.

I also wrote a simpler and less robust collision detection method. Every game object is the exact same size and collision is only checked when the snake is positioned directly on a grid tile. Because of this, I could check collision by only comparing the positions of the passed objects. This method was created in case my previous method caused performance concerns. The performance of the old method was acceptable and this new method went unused.

Collision was simplified by only checking collision against the snake's head. If the code works properly there should never be two non-head parts of the snake occupying the same location, and the apple should never be on the same position as a non-head part of the snake.

The collision detection method returns a bit-flag of the detected collision. The bit-flag gives information on the following collisions: if the snake's head collided with an apple, if the snake's head collided with its body, and if the apple was currently overlapping any body part of the snake. The check to evaluate if the apple was overlapping any body part of the snake was added to accommodate the edge case where the apple tries to spawn on top of the snake.

Rendering Gameplay:

During the game's initialization a static background screen containing the game's grid tile is created. At the beginning of every gameplay render this static background is copied to the canvas. The head of the snake is drawn first and the head's sprite is determined based on the direction of the head. Afterwards, the rendering logic iterates over the remaining initialized snake components rendering each component. The logic alternates between drawing dark-colored and light-colored sprites. Finally, the food is drawn based on the food's position. After every sprite has been drawn to the canvas the canvas is copied to the framebuffer.

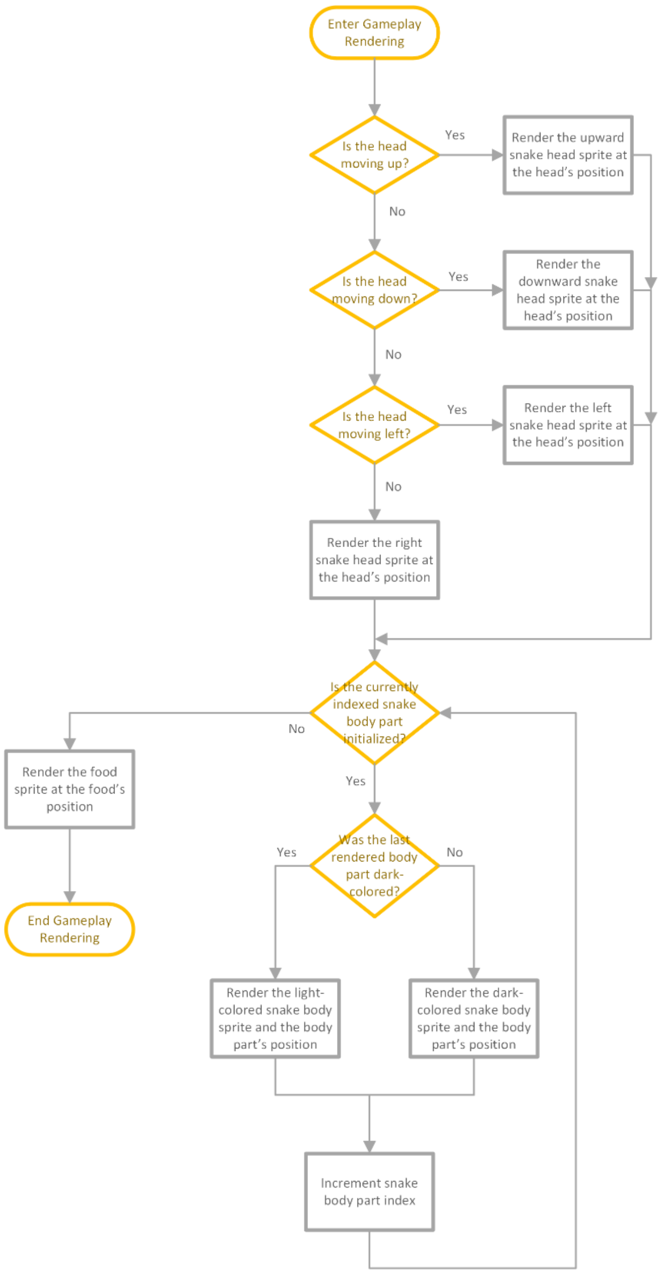

Enter Gameplay Rendering
Is the head moving up?
Yes
Render the upward snake head sprite at the head's position
No
Is the head moving down?
Yes
Render the downward snake head sprite at the head's position
No
Is the head moving left?
Yes
Render the left snake head sprite at the head's position
No
Render the right snake head sprite at the head's position
Is the currently indexed snake body part initialized?
No
Render the food sprite at the food's position
End Gameplay Rendering
Yes
Was the last rendered body part dark-colored?
Yes
No
Render the light-colored snake body sprite and the body part's position
Render the dark-colored snake body sprite and the body part's position
Increment snake body part index

Figure 2: Flowchart of the Gameplay Rendering Logic

Integration:

Jake had already created a great framework for the gameplay logic with his previous milestone. I integrated the gameplay and rendering logic into his stub functions.

Testing:

Milestone 2 saw a different testing methodology from the previous milestone. I felt that writing unit tests would quickly become cumbersome in later milestones and I elected to test the functionality of my code through gameplay; I would perform some scenario and note if that scenario worked as I intended. Testing for milestone 3 was performed using gameplay, and was tested to validate the following functionality:

1.) Ensuring the snake grows when it consumes food
2.) Ensuring the color pattern of the snake is properly displayed
3.) Ensuring a game over occurs when the snake collides with its body
4.) Ensuring the game restarts after a game over
5.) Ensuring that collision and snake data is reset after every new game.

<u>Milestone 3:</u>

Milestone 3 was a relatively simple milestone. This milestone saw an alteration to collision detection to collision detection to detect when the snake is outside of the play field. The logic for this collision detection was made very easy by the facts that during standard gameplay no body part of the snake will be outside of the play area, aside from the snake's head, and that the bounding walls are always at a constant position. These two facts made the check for bounding collision only two check to ensure that the snake's head's x and y coordinates are inside the bounding walls. The result of a detected bounding wall collision is handled the exact same as the rest of the game over code, setting the *wall collision* bit in the returned collision bit flag.

<u>Testing:</u>

Testing for milestone 3 was performed using gameplay, and was tested to validate the following functionality:

1.) Ensuring that collisions with a wall is properly detected with all four bounding walls
2.) Ensuring that collisions with a wall is properly detected regardless of how long the snake is
3.) Ensuring that collisions with a wall is properly detected at various locations of each bounding wall
4.) Ensuring that collisions with a wall properly triggers a game over

<u>Ensuring Compatibility:</u>

The main compatibility concern with this milestone had to do with one of Jake's bonus milestones: *Add an option to enable randomly spawning wall blocks.* I feared that the bounding wall collision detection code may be used to implement this milestone due to the similar names of the collision types as listed in the milestones. This concern was mostly invalid, as such a milestone could easily be implemented using an altered version of the snake body collision detection form the previous milestone.

<u>Milestone 4:</u>

Milestone 4 saw the creation of a game over screen, and a simple game over animation. For this milestone, the game over animation consistent of the snake blinking before a couple seconds before the game over screen. This animation was changed later to be more visually impressive.

Game Over Screen:

The game over screen was implemented to be a simple static screen with only one option: to return to the main menu. Much of the game over screen functionality was built upon work done in previous milestones. Jake had already created a state for the game over menu and logic to leave the menu in his first milestone. The rendering of the game over is performed using the rendering code from my milestone 1. The rendering is performed when the game over state is entered and is not called after. The rendering calls display a sprite of the snake going to the after life, along with the menu option to return to the main menu. The functionality to return to the main menu was facilitated through Jake's milestone 1 code.

Game Over Animation:

The game over animation at this point was very simple. The snake would blink on the game screen several times. After this simple animation played, the game would transition to the game over screen. The addition of the game over animation required the following changes:

1.) The addition of a new menu state, pre game over
2.) The addition of optional arguments to the render function to support rendering gameplay without rendering the snake

When the snake's head collides with a bounding wall, or the snake's body the game enters the pre game over state. Once in the pre game over state, the loops five times, with each loop causing the snake to blink rendering the gameplay screen with the snake, then rendering the gameplay screen without the snake. After the game over animation concludes the logic sets the menu state to be the game over menu.

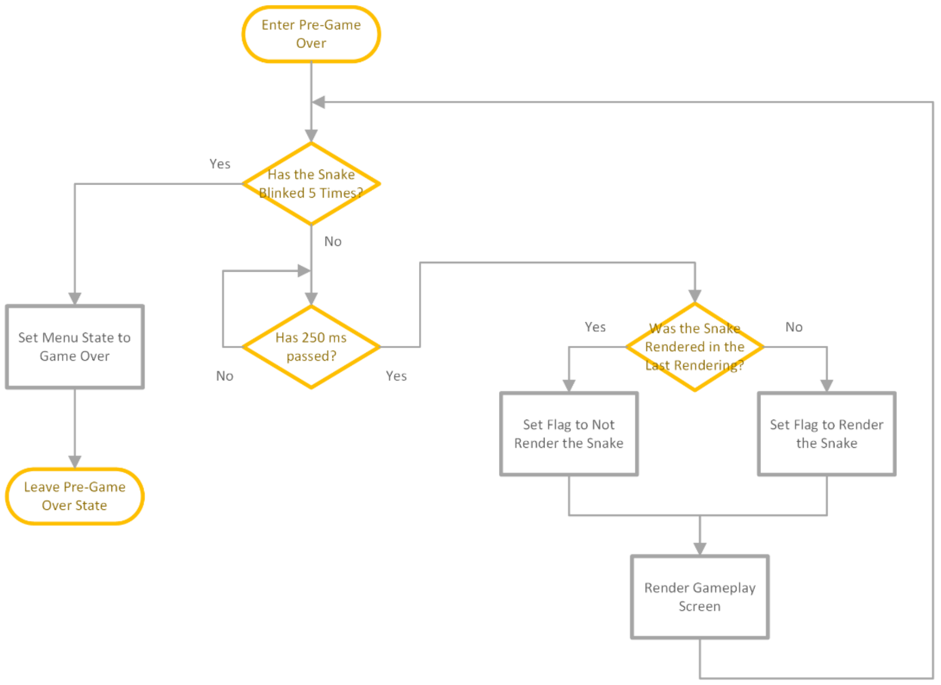
Rendering the gameplay screen without the snake was facilitated through adding optional arguments to rendering functions. The optional arguments include the following:

- An option to render gameplay and the snake without the snake's head
- An option to render gameplay without the snake
- An option to render the player's score as a high score

The option to render the gameplay screen without the snake's head was added specifically for the case where the player gets a game over due to the snake colliding with the bounding wall. When the flag is set such that the snake's head shouldn't be rendered the gameplay rendering code skips drawing the snake's head to the canvas. I felt the animation looked better if all of the snake was within the bounding walls. However, the collision check is performed after the snake moves. Because of this the head will always be positioned outside of the bounding walls during this type of game over. Additionally, I felt the snake's head should not overlap the body of the snake in the case where the player reaches a game over by colliding with the snake's body. When rendering the death animation, the snake is always rendered to have no head. One may think the snake's head never being rendered in the death animation may be undesirable in the case where a game over occurs from the snake colliding with its own body. However, in this case I feel the animation looked better when the body part that snake's head collided with was rendered rather than the snake's head.

The option to render gameplay without the snake was added to provide the blinking animation to the snake. When the flag is set the gameplay rendering code entirely skips drawing the snake's components to the canvas.

The option to render the player's score as a high score was not implemented at this point. However, I felt that it would be likely that in the future we would want to display to the player that they currently have the high score. As such, I added the interface for providing an option to display the score text as "high score."



Enter Pre-Game Over
Yes
Has the Snake Blinked 5 Times?
No
Set Menu State to Game Over
Has 250 ms passed?
No
Yes
Yes
Was the Snake Rendered in the Last Rendering?
No
Set Flag to Not Render the Snake
Set Flag to Render the Snake
Leave Pre-Game Over State
Render Gameplay Screen

Figure 3: Flow Chart of the Logic to Perform Milestone 4's Game Over Animation

Testing:

Testing for milestone 4 was performed using gameplay, and was tested to validate the following functionality:

1.) Ensuring that the game over animation occurs when the snake collides with the bounding wall
2.) Ensuring that the game over animation occurs when the snake's head collides with the snake's body
3.) Ensuring that the game over animation plays for the correct duration
4.) Ensuring that the game over screen occurs after the game over animation finished
5.) Ensuring that the player can navigate to the main menu from the game over screen

Ensuring Compatibility:

I had little concern that this milestone would be incompatible with Jake's contributions. The game over animation and screen were largely independent from the rest of the code base. The optional flags which were added to the rendering function had their defaults set such that all rendering calls that did not use the optional flags functioned the exact same and the rendering call had in previous milestones.

In spite of this, I did have concerns for the longevity of this milestone's code. I feared that adding optional flags and arguments to the rendering function would become unmaintainable eventually if we added more flags in the future. I felt this fear did not require changes to the code, rather is discussed this with Jake and we attempted to avoid adding more optional flags in the future when possible. Additionally, I feared that the gameplay animation would need to be changed in the future, and later we'll see that this fear was correct. The pre game over state was created to remedy this fear. The rendering function treated the gameplay and pre game over states the exact same. If needed in the future, I could create rendering logic to handle the pre game over differently than the gameplay state.

<u>Milestone 5:</u>

Milestone 5 introduced in-game time and score tracking, high score storage and display, and updated snake sprites. The score tracking displays the player's current score during gameplay, and the player's achieved score after receiving a game over. When the player receives a game over the player's score is saved with the high scores, if applicable. The time tracking displays the player's currently elapsed seconds of gameplay. The time displayed on screen updates at the same time a new gameplay screen is drawn. The previous iteration of the snake's graphics had each of the snake's body parts occupy the entire where the body part was located. This type of display was visually unimpressive and made it difficult for the player to determine how each part of the snake was moving.

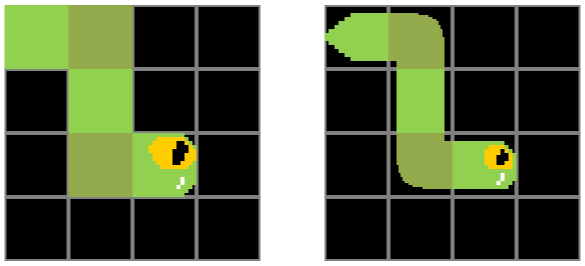


Figure 4: A Comparison of the Previous (left) and Updated (right) Snake Visuals

<u>Score Tracking:</u>

The player's score is incremented in a non-linear fashion. Initially the score in incremented by 10. When then player reaches a score of 100 the score is incremented by 100. This pattern continues until the player's score reaches 10,000, after which the player's score is incremented by 10,000.

<u>Time Tracking:</u>

The player's game time is incrementally updated with each gameplay loop and is stored in two integers: one variable to track the number of elapsed seconds, and one variable to track the number of elapsed microseconds. Every gameplay loop, the timer is incremented by the number of microseconds which have passed in the previous gameplay loop. If the number of elapsed microseconds exceeds 1,000,000 the microsecond counter is decremented by 1,000,000, and the seconds counter is incremented by 1.

Both score and time are drawn to the screen in a very similar way. Logic parses the current high score or time variable and determines the largest non-zero decimal digit of each desired number. Zero digits beyond the largest non-zero digit are not displayed. The score and time and drawn to be right-aligned, and the position of the score and time have been set such that the largest possible value of score and time can be properly displayed without overwriting other information on the screen. The time has no finite limit and will only be displayed to 3 digits.

The score has a conservative upper bound of 1,690,000. As such, the placement of the time and score was chosen such that the time can be displayed to 3 digits, and the score can be displayed to 7 digits.

High Score Storage:

High scores are added using linear insertion. After the player finishes a game logic is run to attempt to insert the just achieved score to the array of high scores. The just achieved score is repeatedly swapped with the next element in the high score array until the just achieved score is in a position where it is smaller than the next highest high score.

Displaying High Scores:

The display of high scores is handled the same way as the rendering of the score to the gameplay screen. Jake had already created the high score screen, I added the logic to retrieve and display the high scores.

Updated Snake Visuals:

Updating the snake's visuals to show the direction of each individual piece of the snake required me to change a decent portion of the snake's rendering logic. Additionally, several new sprites had to be created. Originally, the snake consisted of only 6 sprites. After updating the visuals, the snake consisted of newly created sprites 24 sprites.

Previously the rendering of the snake was performed in two parts: rendering of the head, and rendering of the body pieces. The rendering of the head was largely unchanged by the visual update. The rendering of the body was separated into two parts: logic to render the non-tail portion of the body, and logic to render the tail.

Rendering of the body took advantage of the direction field attached to each piece of the snake. Each body part was rendered based on the direction of the currency body part, and body part next closest to the snake's head.

The tail rendering logic works in a very similar way to the head rendering logic, however, instead of rendering directed head sprites the logic renders directed tail sprites. In hindsight, the logic is so similar I could have implemented the head and tail rendering functions using a common helper function.

The body part rendering code needed to handle 32 cases: 4 for the possible directions of the current body part * 4 for the possible directions of the next body part closest to the head * 2 for the possible shades of the snake's sprite. Handling 32 well defined cases was surprisingly hard. I decided to create a helper function for rendering individual snake pieces. The idea was the I could pass the light or dark variants of each sprite to the helper function based on which color the sprite should be. Essentially, the helper function is comprised of nested switch statements that just check for the directions of the two of interest snake body parts. The helper function's creation was also motivated by a need to future proof the snake rendering code. I knew at some point I wanted to add simple animation to the snake's movement, and figured that it would be best if I could use the same helper function for rendering the animation frames. This intention did not pan out as I had hoped. When writing the helper function, I had it take 6 sprite arguments: a sprite for the snake moving vertically, a sprite for the snake moving horizontally, and 4 sprites for the snake moving in a and L-shape. In the next milestone we will see that this was a mistake.

Recall from milestone 1, because of how the sprites are stored in the header file, each sprite takes as much memory as the largest sprite in the program. With only 6 snake sprites this wasn't a huge problem. However, we no longer had 6 snake sprites. Now the code has 24 new snake sprites, in addition to the leftover 6 snake sprites which I chose not to remove in case they were mistakenly still references in the code. The sprite data had gotten so large I started to run into issues with the sprite data overlapping with other important parts of memory, leading to very cryptic crashes. After doing some debugging, and looking at the memory addresses of the sprite data I realized that the sprite data had gotten so large that some game data has overlapping memory addresses with the VGA frame buffer. In response, I had to change the memory location in the project's linker script. I knew this problem would be likely to occur later, so I set all of the game data to be stored after the VGA frame buffer. I specifically chose this location so the game data could continue to grow without the risk of overlapping with the VGA frame buffer.

<u>Testing:</u>

Testing for milestone 5 was performed using gameplay, and was tested to validate the following functionality:

1.) Ensuring the score increments when consuming food
2.) Ensuring the score increments by the proper amount based on the player's current score
3.) Ensuring that the timer increments throughout gameplay
4.) Ensuring high scores and displayed in the proper order on the high score screen
5.) Ensuring that the player's score is added to the high scores when sufficiently
6.) Ensuring that the snake sprites are properly rendered using the newly made sprites

<u>Ensuring Compatibility:</u>

Compatibility with regards to milestone 5 saw this milestone take advantage of thoughtful choices made by both me and Jake during earlier milestones. However, some consideration was given to ensuring that this milestone would be compatible with future changes.

Much of this milestone's objectives where either partially completed previously, or were independent from much of the other code. However, when I was creating the updated snake visuals, I did give some consideration to how the code would be used in the future. I attempted to create the snake piece drawing helper function to be usable in the future when I got around to adding animation to the snake's movement.

There were also some places where I could have further future proofed code. Specifically, with the number drawing code. Much of the logic used to draw scores was replicated in logic to draw time. In hindsight it would have been better to have the duplicated logic in a common helper function.

Milestone 6 (Final Demo):

Alternative Snake Colors:

Implementing alternative snake colors was somewhat painful. The rendering code did not support palette-based sprites. As a result, I had to recreate every snake sprite for each new color. This process wasn't hard but it was certainly tedious. I created a helper function to obtain all of the snake sprites for the currently set color. This function was used in the gameplay rendering logic to obtain the proper snake sprites. Additionally, I created setters and getters for the current snake color.

Alternative Food Sprites:

The addition of alternative food sprites was a fairly simple change. I created the new food sprites, and created setters and getters to for the current food sprite. In the gameplay rendering code I replaced hard-coded retrieval of the apple sprite with my new food sprite getter.

Hard Mode:

The addition of hard mode was also a fairly simple change. I added a helper function to alter the time a gameplay loop should take. This helper function would reduce the desired delay time by a multiplier based on the player's current score. This helper function, of course, checked if hard mode was enabled before performing any alterations to the gameplay loop's delay.

Options Menu:

After I had implemented alternative snake colors, food sprites, and hard mode I added these options to the options menu. Jake had already done some work with the options menu, all I had to do was to render the new options, and update the options menu logic to make the new options accessible.

Updated Movement Animation:

For this milestone I created a simple two frame movement animation for the snake. The first frame of the animation consisted on the previously made gameplay rendering frame. I mainly needed to create the second frame and relating logic. To support this animation, I had to create 24 new sprites for each supported snake color. You may think to yourself "why didn't you just shift the position of your previously created sprites?" There is a very good reason. The rendering code does not support transparency, and background data had to be baked into the snake's sprites. If I were to shift the position of the previously created sprites then the background tiles would be out of place. I created a second gameplay rendering function that only handled rendering the second frame of the snake's movement animation. For the most part, this function worked the same as the logic to render the first frame, however this function used the newly created sprites and shifted the position of certain sprites such as the snake's head and tail.

Remember from last milestone where I had a helper function to render an individual body part of the snake? I intended to reuse that code when creating the snake's movement animation. However, that function only functioned when there were 6 possible body sprites to be rendered. However, for the second frame of animation each body part could be rendered as one of 12 possible sprites, based on the movement of the snake. As a result, I had to rewrite and retest this helper function.

To integrate this new frame of animation I had to alter the gameplay logic such that the gameplay screen would render twice as frequently and alternate between the two frames of animation. Gameplay inputs would only be processed during every other rendering to preserve the snake's movement being constrained to the grid tiles.

Updated Death Animation

For the updated death animation, I created a simple four frame explosion that would cascade through the snake from the snake's head to tail. This new animation required me to re-work the pre game over state's rendering. I had to completely redo the old game animation. For this second iteration, I iterated over the entire snake and would update the frame and position of the explosions based on the iteration index and the snake pieces' positions.

Ensuring Compatibility:

For this milestone I wasn't concerned with futureproofing. Instead, I ensured compatibility by communicating with Jake about his planned changed to the options menu to ensure that we weren't implementing incompatible code.

Testing:

Testing for milestone 6 was performed using gameplay, and was tested to validate the following functionality:

1.) Ensuring that every snake color appeared properly in gameplay
2.) Ensuring that every snake color was accessible from the options menu
3.) Ensuring that every food sprite appeared properly in gameplay
4.) Ensuring that every food sprite was accessible from the options menu
5.) Ensuring that hard mode functioned during gameplay
6.) Ensuring that the normal gameplay speed was preserved when hard mode was not set
7.) Ensuring that turning on and off hard mode was accessible from the options menu
8.) Ensuring that the snake's movement animation appeared properly in gameplay for every snake color
9.) Ensuring that the death animation was displayed properly and was displayed to completion

Source Control:

For source control I used a mix of locally stored dated directories, and Github. If my code was only partially functional, or had bugs by the time I stopped working I would only store it locally in dated folders. When my code reached a point where it was of sufficient quality, I would create add it to the Github repo in a new branch, and merge the branches when appropriate.

Design Methodology:

For the most part I developed my code with a top-down methodology. I believe the only code I wrote with a bottom-up methodology was the rendering code. I developed the rendering code with a bottom-up methodology as I wasn't entirely sure what the workflow for rendering should be as I was writing the code. I intended to create the base functionality and create a rendering work flow based on what work flow I felt would have been easiest.

Project Contribution Summary:

Throughout this project I gained a lot of experience working with and developing low-level software. Much of my contribution to the project was through gameplay and graphics programming. While I had some previous experience in real-time software, graphics, and collision detection, I hadn't yet had the experience to develop such code from scratch and deal with the resulting design choices. Much more than previous projects, I had to give consideration to the performance and longevity of the code I was writing. Additionally, this project forced me to understand the different portions of memory in a program image, and how that can affect the end program.

As a final note on my contributions, the largest lesson I learned during this project is that *early design decisions can have great and unexpected impacts later on.* Many of the compromises I made in my original rendering code made implementing later graphics changes painful. Since transparency wasn't supported, I needed to draw every gameplay sprite to have the background grid data baked into the sprite. For changes like the snake's movement animation, I couldn't just offset the position of the existing snake sprites as then the background grid would be misaligned every second gameplay frame. Instead, I had to recreate every snake sprite with the proper background animation. Additionally, the need to remake sprites with proper background data made it infeasible to add any more frame of animation to the snake's movement. The lack of palette-based rendering forced me to remake every snake sprite each time I wanted to add a new color. All the above may seem like pure graphics work, however the large number of sprites that were expected to appear in gameplay forced me to make a decent amount of code that only organized these accessing of these sprites. Much of this work may have been avoidable if I had made different design choices earlier on, or if I had more carefully considered the constraints that I was making for myself.

Community Contribution:

I proved MATLAB helper scripts mentioned in milestone 1 to group 1 (Kaj and Divyam). The scripts take a folder of images and converts them into a C++ header file with enumerators for referencing image data, the image's width, height, and graphical data.

Feedback to Xilinx

My concerns towards Xilinx software are largely targeted towards usability, rather than functionality. Keep in mind that I used Vivado and Vitis version 2020.2 and some of my feedback may have already been addressed.

I worked with Vivado less extensively compared to Vitis. For the most part Vivado seemed to perform its functionality as desired. However, my largest concern towards Vivado is in regard to exporting the project between machines. Vivado seems to use absolute paths with projects, which made it very difficult to transport or relocated projects without entirely restarting the project. Additionally, there seemed to be no built-in functionality to export the whole project. I feel this feature would have benefited my workflow.

Vitis was functional but lacked some functionality that would have been desirable. First and foremost, Vitis seemed to have no integration with source control. Additionally, it would be nice to have real-time error and warning checking, similar to what is provided by other IDEs such as Visual Studio, and Visual Studio Code. Finally, I ran into stability issues with Vitis. Frequently Vitis would crash when uploading a ROM to the ZedBoard. Additionally, Vitis would occasionally be able to upload the ROM to the ZedBoard, but would not be able to start the program. Strangely enough, if I disconnected and closed Vitis the program would run on the ZedBoard fine.

Course Feedback:

I really enjoyed this class. I feel it gave me a great opportunity to gain hands on experience in design. This experience was made more engaging by allowing the students to choose their project topics. For more specific thoughts, my feedback to the course is mainly related to the following sections:

- Project timelines
- Open lab concept
- Ordering of course material with respect to the lab

I feel the project milestones and tutorials could have started a week or two earlier in the semester. The first month or so of lab work was sparse almost to the point where I felt students could have easily become disengaged with the course material.

I personally enjoyed the open lab concept. I feel a week in between milestones gave students enough time to complete their milestones, without allowing students to neglect the project. As a point of concern, I believe the open lab concept succeeded in part due to Chris Keilbart. Chris was always helpful, responsive, and easy to contact. I feel a less available TA may not be ideal for an open lab concept. The tutorials were fine for the most part, however I feel it would be beneficial to the students to have an interrupt tutorial in addition to the audio and video tutorials.

If I can give one negative to this course it would be the ordering the lecture material. I feel topics like compilers and formal verification were great tools to haver for the project. However, these lectures were given at the end of the course, too late for group to use this information to aid their project. I would like to see these topics covered earlier in the course, perhaps just before or around the start of the project. That being said, I understand that it may be difficult to obtain guest lecturers that early in the semester for topics like formal verification.