Group 6

Sarahbeth Ramirez

Kevin O'Brien

Mark Armendariz

Zenaida Gamino

Ultimate Punch Frenzy – *Immortal Peacetime*

Fighters in the STREET

By Sarahbeth Ramirez

Our group collaborated to produce this 2 dimensional, 2-Player Arcade fighting game. This report will merge each member's description of a portion of the software engineering process, as well as our contributions to the project.

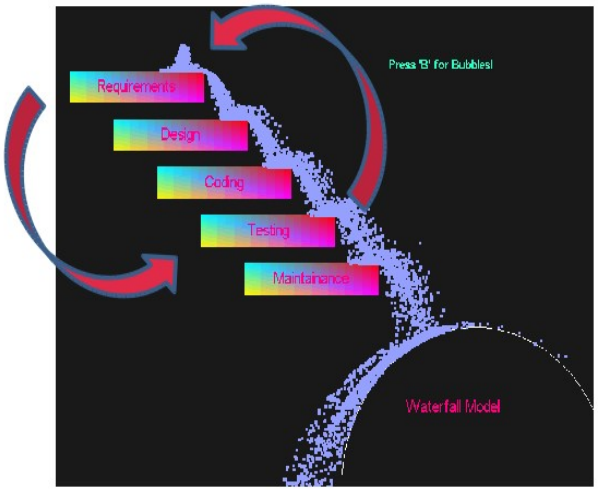
We created a basic 2-Player fighting game with our inspiration coming from games like Mortal Kombat and Street Fighter:

(Greater than)

Our team had the intention to use the waterfall method, but we only loosely followed it. The method most closely resembling what we implemented was the Agile Software Engineering Method, which is a modified of version of the waterfall method. The key feature which attracted us to this model is the *ability to change the requirements throughout development*. This flexibility allowed us to continually develop the game at a rapid pace, for it was conducive to adding new features when inspiration came to mind.

WATERFALL METHOD…

but *Modified*!



Agile Software Engineering Model:

Specific Requirements for the Game

The final set of requirements decided upon can be divided into two groups, functional and non-functional.

Functional Requirements:

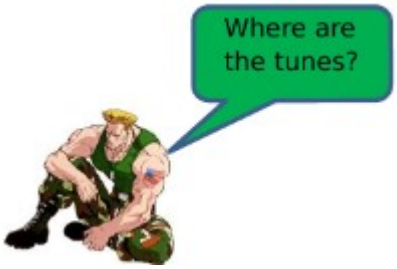
A menu driven environment for the user to navigate intuitively:

Ability to choose two characters from the available four

Ability to choose the specific stage

Sound effects during game-play

Sound tracks for menu screens and during game play



Ability to pause game-play and reset to main menu

Provide user instructions for game play

Animated game play that incorporates punching, blocking, movement and health bars to determine a winner

<u>Non-Functional Requirements:</u>

Used C++ language to code the game

Title screen navigation using keyboard and/or mouse input

Select character and stage using keyboard input

Audio effects of soundtracks are randomized from a selection of songs

Sound effects during game-play occur at character collisions (when punching)

Using keyboard input allows for game-play to pause, view hints and reset to main menu

Key bindings available through the pause help screen

Winner of the game is determined and displayed by utilizing a graphical health bar

In addition to the requirements of the game, there are requirements specific for this project as an assignment for our Software Engineering course. These include:

Project maintained by the version control system Github

Each member of the team is responsible for their own source file which showcases some of the code they contributed to the project

A Makefile to build the project on the local systems

A technical report detailing the group's implementation of a software engineering process

A Presentation to the class which showcases our final product

Most importantly, working as a TEAM to produce a game!

Design

By Mark Armendariz

The basic design we were going for was the basic 2-player fighting game. We took inspiration from games such as Mortal Kombat and Street Fighter, as is semi-obvious by the title of our game. So there were some basic ideas that we knew we wanted in our game.

First, we knew we wanted more than one character to choose from and more than one fighting stage to choose from. We settled on having four characters and two stages to choose from.

Next, we wanted to define a set of fighting moves for our players. We thought about having a more complicated set of moves to choose from, but for the sake of time, we chose to have a far more manageable set of moves. We wanted our

players to have a walk, block, and punch, all controlled from the keyboard for both player one and player two. The block and punch would also require collision detection as well as the use of timers for the animation of these player movements to work properly and flow smoothly.

Next, it was decided that, in the vein of Mortal Kombat and Street Fighter once again, that we have health bars for our players. Each health bar would have a texture frame and the inner part of the bar would be red, much like Mortal Kombat has. With each landed punch, a constant amount of health will be deducted. If a punch is landed on the opposing player who is blocking, a smaller percentage of that constant amount will be deducted.

Next, we wanted our game to be menu driven. It would coordinate the choosing of characters and stages, as well as restarting the game after each match. We got the basic idea from the Mortal Kombat arcade game on how to implement the menu. We wanted our menus to have simple directions on how to choose each option as well. It was also decided that we have music for each of the menus and recordings of a voice telling the players to “Select Your Character” and “Select Your Stage”.

Finally, we wanted our game to be able to restart at the end of every match and allow the option of going back to the character select screen so that the players will be able to choose their characters and stage again to do more matches. So we knew we wanted to have some kind of function that would allow the game to restart to an earlier state.

Implementation

Images and Sounds

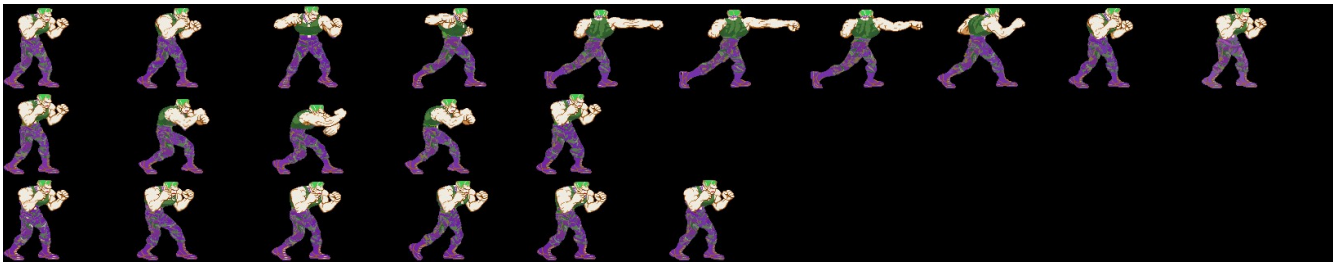
This game needs a ton of a textures to be loaded. Doing this in the middle of the game would take far too long. So a function init_opengl(), that was originally from bump.cpp, was modified to help resolve this issue. All textures were consecutively loaded right at the start of the game in order to avoid any undesirable long loading times. This way, whenever a character is chosen at the character select screen or a stage, the required texture that needs to be binded by OpenGL will be readily available. This function also set up the necessary alpha data to allow for transparency in the texture mapping which is used by the player sprites.

Two more similar functions were then created, called init_healthBars() and init_players(). Init_healthBars() was created in order to initialize all structure values for the layered textures on the health bars. Both health bars require three layered rectangles, with two of them requiring a texture, so a function helped to nicely set them up for use in our render() function. Init_players() was created in order initialize the correct positions and size of both players. This function was also used to initialize the numerous boolean variables each player structure needed for the game logic to work and be more organized.

The sprite sheets for our characters all started from three gif animations found online of the Street Fighter character, Guile. One gif was of Guile punching, one was of Guile walking, and the other of Guile doing a special attack move.

A block had to be created by manipulating the gif of the special attack move. In order to get this character into the game, these gif files were put into GIMP and separated into even sections in order to allow the program to cycle through them for the animations. Each of these split up animations were then placed into single sheets with black backgrounds for transparency in order for them to be loaded into the game and used by OpenGL.

For the sake of time, in order to generate more characters for the game, Adobe Photoshop and GIMP were used together to manipulate this first sprite sheet into three more. The most dramatic example of this is in getting the character Joker into our game.



All voice recordings were recording using an Ipod. In order for them to be ready for use with FMOD, they were split up, and processed in a program called Cool Edit Pro. In this program, each audio file was edited, cleaned up, and processed with echo and pitch lowering to achieve the desired sound for our menus and game play.

Implementation (continued)

By Kevin OBrien

Animations:

Animations in this project include a punching animation, a walking animation, and a blocking animation. These were each implemented in slightly different ways as they each required a unique touch.

The punching animation implementation involved a start clock to hold the time that the punch begins. Every time the punch animation function is called, the difference between the current time and the start time is calculated. The texture that is bound from the sprite sheet to the character box depends on how large this time difference is. When the texture of the fully extended punch is being drawn, collision with the other player is checked and health is deducted accordingly. The punch finishes and the player is able to punch again.

The walking animation was implemented in a slightly different way than the punch. The punch needed to work where if the player presses the button, it will only punch once, but if the player presses the walk, the character needs to go into a continuous walk cycle. So, much like the punch animation, a time difference was calculated to decide which texture is bound at what time. The only difference is that while the walk key is held down, the program continuously calls the walk animation function until the player releases the walk button.

The blocking animation was implemented with a timer used to switch between binded textures until it reaches the texture for a full block. Once it reaches

this texture, it stays bounded until the player releases the block button, making blocking a crisp and fluid animation.

All three of the animations were coded to act independently of each other. Each one cannot occur while another one is in progress, adding strategy and realism to every fight.

Collision:

Collision with the window edges and the other player were fairly simple. It was implemented in such a way that when player 1 runs into player 2, player 1 looks like he is pushing player 2. If both players are pushing against each other, then they are at a stand-still. Both players are not able to go past the window edges or push each other off-screen.

The next important collision is when damage is taken away. If player 2 is inside player 1's hit box during the sweet spot of the punching animation, player 2 is sent backward away from player 1 and his health bar is decreased (health bar implementation involved shortening the health bar and then repositioning the bar). However, if player 2 is blocking when the punch hits, then player 2 is not sent back as far as a regular hit would send him and his damage deduction is severely reduced.

Testing Phase

By Kevin OBrien

The majority of our testing in this project was white-box testing, This is because after one of us made changes, each one of the group members would test it themselves to make sure there were no bugs. Since all our members had extensive knowledge of the code before testing, our individual debugging sessions mainly focused on reviewing the code. If someone saw a possible problem from within the code, that portion of the code was tested in all ways that were thought to possibly produce an error.

Although, we did a lot of white-box testing, black-box testing was also used. Every once and awhile, a set of fresh eyes from someone from outside our group would be chosen to come look at the game. The requirements were the main focus. The tester was told what the game was supposed to do and then the tester would report on any strange anomalies that would occur. This type of testing was extremely useful because something as simple as a new perspective uncovered bugs that would have been overlooked otherwise.

As with any game, bugs were bound to be present. Since we had a plan for how our testing was to take place at the beginning of the project, bugs were more efficiently found. Also, it is important to note that the members of our group have had past experience with finding bugs in video games, as every member plays them. This past experience actually helped to test, through intuition. This helped us to realize errors that were going to occur before we even finished the coding phase of our project.

By Zenaida Gamino

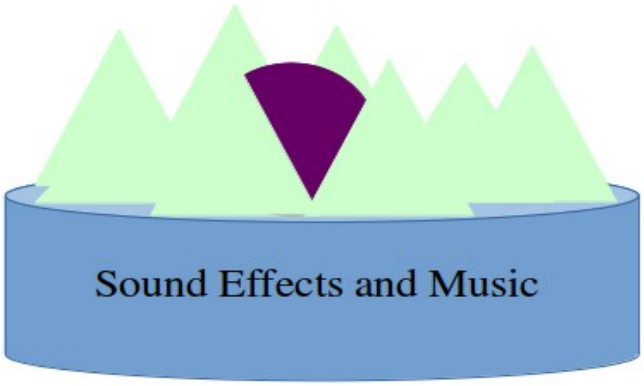
Audio:

In order to get the atmosphere for our game to feel right good fighting music and sound effects where needed. To use sounds in the game it was decided that the sound effect engine fmod was to be used to convert the music so that sound can be played in the game. In the first stages of the game our group concentrated on just having background music. If we could get that running then later in the development of the game they thought we could easily set up the sound effects for the fight.

Unfortunately things would not be that smooth. Setting up the channel for the background music was simple; the most trouble that was had was putting the makefile together so that fmod would work. With help of the menu logic it was relatively simple to start the music at the right moment. At this point our group only had the background music set for the game. Everything was on the right track until we went back to add the sound effects later on in game development.

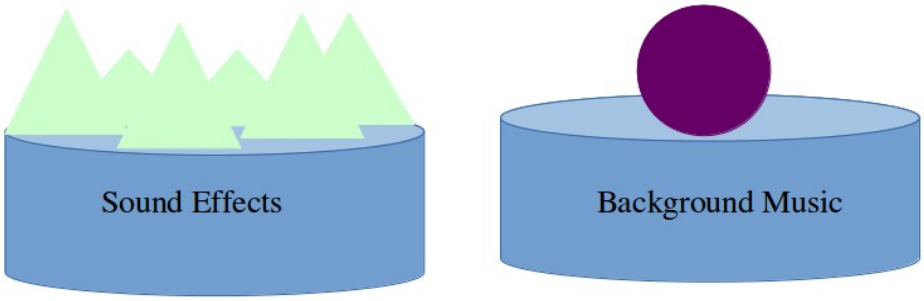
The first problem that was encountered when dealing with the sound effect was the problem that after a certain amount of punching the background music would stop abruptly. Our group came to the conclusion that the single channel was filling up with the punch sound effects, and that was the reason the background music could no longer be played.

Group's initial plan for audio: Use only one channel for all sounds



After doing some research, our group came to a conclusion that a second channel was need to make both the sound effects and the music work. We needed one channel specifically for the background music, and another specifically for the sound effects, that way neither of channels would become full.

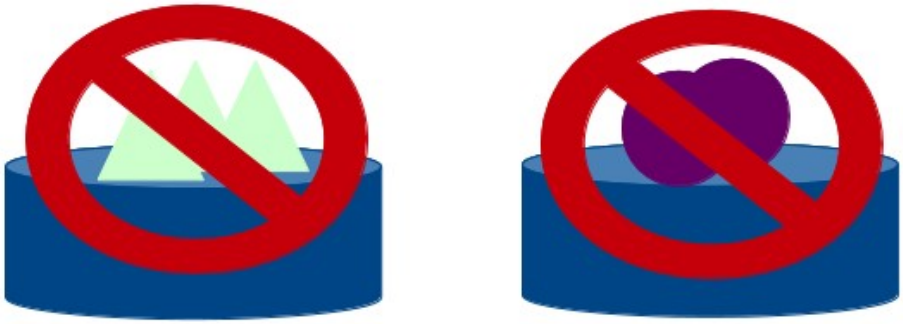
Group's final plan: Use two channels, one for sound effects and one for background music



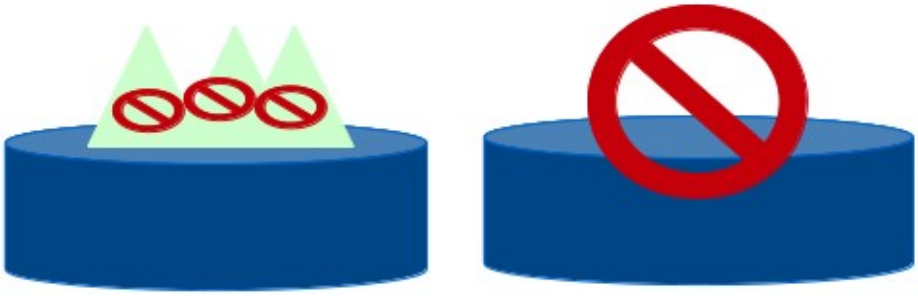
After adding the second channel, things started to look up. It became possible to play both the background music and the sound effects. However after fixing the first problem we encountered another.

The background music would no longer stop if the game ended. Instead when the game went back to the menu screen, the music would continue playing. we realized that they initial way we had been stopping the music was not the right way. We had been removing the channel completely instead of clearing the channel of the sounds. After adding a function to clear the channel the sound effects and the background music came together to make the game feel more interactive.

Groups Initial Plan: Destroy both channels in order to stop audio



Groups Final Plan: Clear channels in order to stop audio



Delivery/Maintenance:

In order to present the game properly to the class, the group decided to separate the presentations into parts. Each member wrote specifically on a phase in the software engineering process as well as their contribution to the project. After everyone finished writing their portion, we got together and compiled everything into one report, and practiced on how we would present it to the class. Due to the nature of this project, the maintenance phase is not applicable. However, if we wished to implement this phase, we could add more characters, moves, music or stages and ensure the project always works.