Digital Media Coursework: Making a Game

Matt Thompson, Module code: CM40198

April 26, 2013

Contents

1 Introduction

This report describes the techniques used to develop a simple "beat-em-up"-style game that runs on Linux, Mac OSX and Windows.

In this game, two characters compete in a fight where they can punch or kick each other to reduce their opponent's health. They can also perform evasive manoeuvres by crouching and jumping.

Instead of drawing or modelling the characters to put in the game, I used live-action footage of some of my friends punching, jumping, etc in real life. I filmed their actions in front of a green screen, which was then later replaced with the game background using a chroma-keying technique described in section 4.1. This video footage was then converted into sprite sheets to use as animations in the game (see section 4.2).

2 Tools used to make the game

In order to save time, I opted not to build the game from scratch using a low-level language like C^{++}. I instead used a minimalistic game framework, LÖVE2D [3], to handle the low-level OpenGL routines.

2.1 Game framework and scripting language

2.1.1 LÖVE2D

LÖVE2D [3] is a 2D game creation framework written in C^{++} that allows game programming using the Lua [4] scripting language. It is released under the zlib license [6], a free and permissive license.

The design of LÖVE2D is very simple. It provides three main callback functions: `love.load()`, `love.update(dt)` and `love.draw()`. There are also callbacks for keyboard events that are called whenever a key is pressed.

The `love.load()` callback is called when the game loads for the first time. It is only called once, so it is used for initial setup of the game. This can be for things like setting the size of the game window, reading in the animation sprite sheets, and initialising the game objects.

The `love.update(dt)` callback is called continuously. The 'dt' parameter stands for 'delta time', and is the amount of seconds since the last time the function was called (which is usually a small value like 0.025). Most of the game logic happens inside this loop.

The `love.draw()` callback continously draws images to the screen. Functions such as `love.graphics.draw()` must be called from inside this function in order to change what is displayed on the screen.

Running the game: To run the game, the LÖVE2D runtime must first be downloaded from `http://love2d.org`. Once this is installed, the game can be run by double-clicking on "`bloodbath.love`", or by entering the directory via the command line and typing "`love .`".

Exiting the game: To quit the game at any time, press the escape key.

2.1.2 Lua

LÖVE uses the Lua [4] scripting language for the programming of game logic. Lua is an 'extension programming language', meaning that it only works embedded in a host client. It is an ideal scripting language for games, as its minimalistic featureset allows for extremely fast execution.

Figure 1: A screenshot of the gameplay

Lua is a dynamically typed language that offers good support for object-oriented, functional and data-driven programming paradigms. Interestingly, the only data structure provided out-of-the-box is a table, but the nature of Lua facilitates the extension of these tables into more complex data structures.

Another interesting feature of Lua is its use of *metatables*, which define the behaviour of values under certain special operations. These metatables allow the extension of Lua to create things like instantiatable classes out of tables.

2.2 Image manipulation tools

I tried to use only free and open source software to make this game. For this reason, I chose to use Blender [1], a 3D modelling tool, for compositing tasks like chroma keying. I used the GIMP [2], an image editor, to stitch together the images produced by Blender into sprite sheets.

Although Blender is traditionally used for 3D modelling and animation, it also has a very sophisticated node editor for compositing work. In section 4.1, I describe how I used this node editor to remove the green screen from the background of the video footage.

I used a script made by a GIMP user for the creation of sprite sheets to stitch together my animation images [5]. More details are found in section 4.2.

3 Game design

This game is designed to be played by two players. When the game first starts, the players are presented with a character selection screen. Once both players have selected their characters, the game begins.

Player 1's health is represented by a bar in the top left corner of the screen. Player 2 has a bar at the top right. At the bottom of the screen, the player characters themselves appear.

The players can move towards and away from each other and try to punch and kick. If a player tries to attack the other player and they are close enough, the other player loses some health points, and their health bar shortens. Once a player has lost all of their health points, they 'die', and the other player is declared the victor. Figure 1 shows the game in action.

Controls Player 1 can move their character around by using the W, A, S, and D keys on the keyboard to jump, walk left, crouch and walk right respectively. The left shift on the keyboard makes the character punch, the left control key makes them kick.

Player 2 moves their character with the arrow keys, and uses the return and right shift keys to punch and kick.

Pressing the escape key at any time quits the game.

Joypad support is partially implemented, however no Linux-compatible joypads could be found for testing.

3.1 Code structure

The code is split into seven files: `main.lua`, `player.lua`, `gfx.lua`, `fight.lua`, `game.lua`, `hud.lua` and `anim8.lua`

- `main.lua` contains the main LÖVE callbacks (load, update, draw).
- `player.lua` contains the Player class.
- `gfx.lua` contains functions to load animations from image files.
- `fight.lua` contains the Fight class, which manages the interactions between Player objects.
- `game.lua` contains the Game class, which handles the entire game and displays the character selection menu
- `hud.lua` contains the StatusBar and Messages classes, which display the players' health bars and shows text on screen.
- `anim8.lua` is a third-party library for the handling of animations.

3.2 Class descriptions

The classes are shown in the class diagram in figure 2. Constructors are indicated by the "new" method of each class. Although Lua is a dynamically typed language, I have included the expected types in the class diagrams. It should also be noted that all class members are public, since Lua does not offer privacy mechanisms.

3.3 Finite state machine

The game is a very simple finite state machine with nine different states:

- Rest
- Walk
- Punch
- Kick
- Jump
- Crouch
- Get hit
- Win
- Lose

Figure 2: Classes

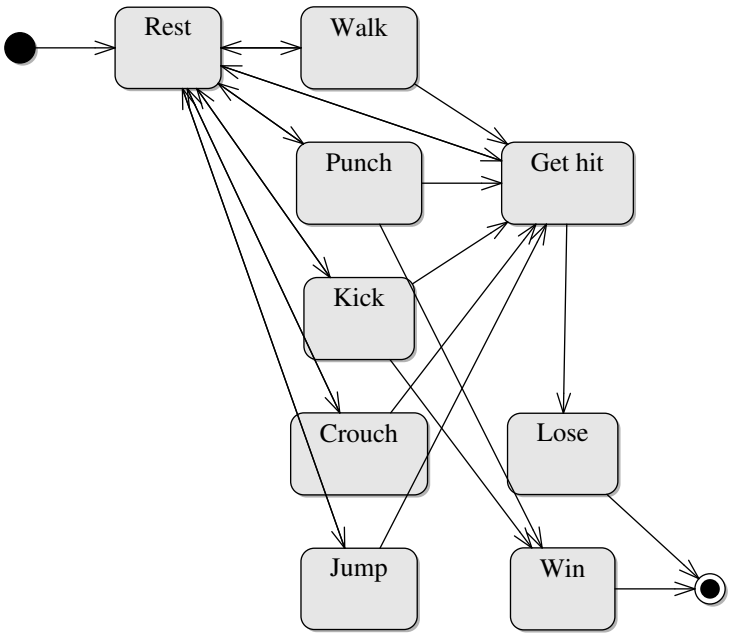

Figure 3: Player state machine

Each player starts off in a 'rest' state. Pressing the left or right keys transitions them into a 'walk' state. Pressing up goes into the 'jump' state, and holding the down key makes the player crouch. The player returns to the rest state once the down key is released. Punch and kick states are entered when their respective keys are pressed, returning to the rest state once their animations have played once through.

When a player gets punched or kicked by the other player, they enter their 'get hit' state, which plays an animation for a while before returning to the resting state.

It should be noted that each state corresponds to an animation. While an animation is playing, the player remains in that state.

If a player has been hit many times by the other player and their health value reaches zero, they go into the 'lose' state, from which there is no return. The other player ends in the 'win' state.

A diagram of the player state machine is shown in figure 3.

4 Animation techniques

4.1 Chroma keying

Chroma key compositing is a technique for layering two images together. One image, the 'foreground image' has a mask of transparent pixels (alpha mask) applied according to a set of colour hues (chroma range). This technique is also commonly referred to as green screen or blue screen, since these are the colours typically chosen to be replaced with alpha pixels.

Typically, a function like this is applied to every pixel in an image:

$$f(r, g, b) \to \alpha$$

If the function returns $\alpha \leq 0$, this means the pixel is in the green screen area and will become transparent. If $\alpha \geq 1$, then the pixel will be part of the foreground. If $0 < \alpha < 1$, then the pixel will be partially transparent, with some of the foreground still visible.

Figure 4: A frame from the captured video footage. A mask has been drawn around the desired area.

In this implementation, I also used Blender's node editor to do screen spill removal. A simple example of such a function, again applied to all pixels in the image, would be:

$$g(r, g, b) \rightarrow (r, \min(g, b), b)$$

This function removes any kind of coloured 'tinge' that may exist around the edges of the foreground image after chroma keying is applied.

Figure 4 shows a frame from the video footage for an animation. A mask has been drawn around the area of interest to be chroma keyed. Figure 5 shows how the image looks in Blender's node editor once the chroma key and color spill functions have been applied.

4.2 Sprite sheets

In the game, each character is represented by a sprite, which is simply a 2D image with a transparent background. Each action for each character has a set of images which make up its animation sprite.

While it would be perfectly possible to store every image for an animation separately, this is often inconvenient. I decided to use sprite sheets to use one file per animation.

A sprite sheet contains many images arranged together in a grid. Each row in the grid represents a separate animation, with each column of the row being one frame of the animation.

Initially, I wanted to store all the animations for each character in a single sprite sheet, with one row per animation. However, this resulted in a final image that was much too large to handle. Since each player sprite is 100 pixels tall by 100 pixels wide, the total dimensions of the image soon exceeded 2048 by 2048, which is the maximum texture size for most graphics cards. Since LÖVE2D stores sprite sheets as OpenGL textures, a sprite sheet that exceeded this maximum size would just be displayed as a white box.

For this reason, I instead used one sprite sheet per character animation. Figure 6 shows the sprite sheet for one character's "winning" animation.

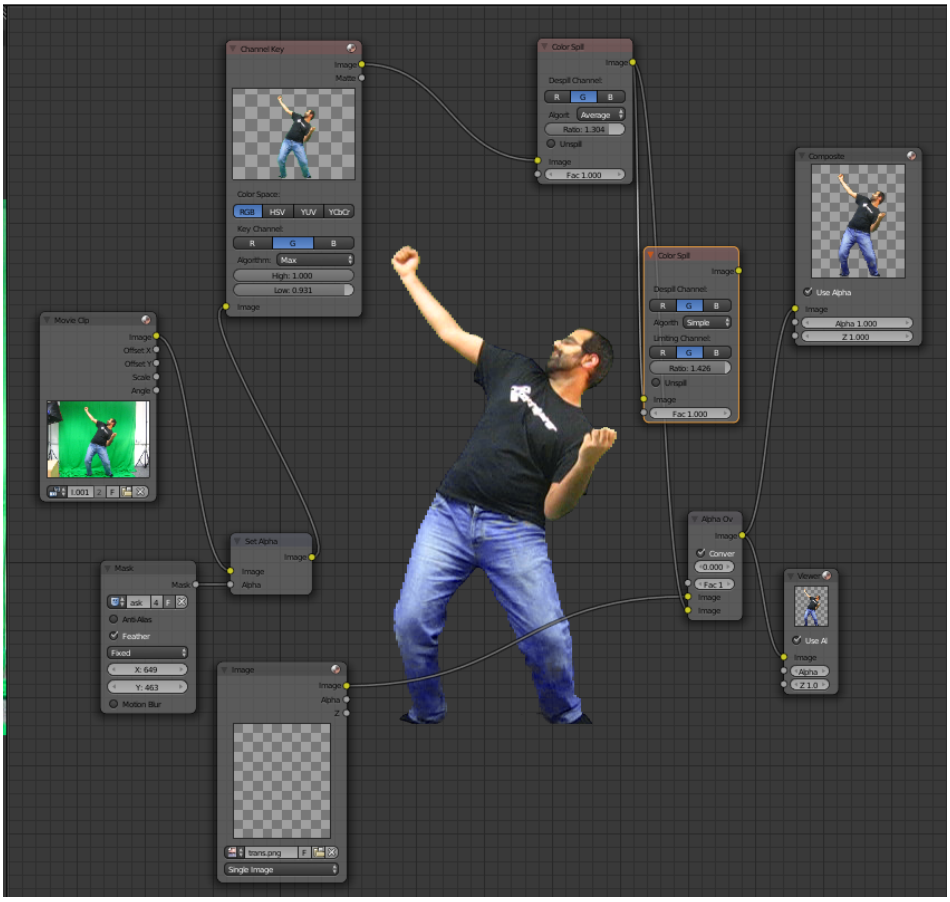

Figure 5: Blender's compositing node editor

Figure 6: The final sprite sheet for Hashim's "winning" animation

5 Evaluation and future work

AI for the game characters will need to be implemented in order to support a single-player game mode. Also, music and sound effects need to be added to make a more immersive game experience. It would also be nice to add more characters and joypad support in the future.

The game has plenty of bugs. For example:

- Selecting the same character for both players results in only one character appearing on the screen, who then beats themselves up.
- Pressing a key while a character is jumping makes them get 'stuck' in midair.
- Players can still inflict damage when attacking in the wrong direction.
- Players can still be moved when dead.
- The only way to replay the game is to close and re-open it.

At the moment, this game is just a proof-of-concept, but I do hope to develop and polish it a little further. The full source code can be found online at `http://github.com/cblop/bloodbath`.

References

[1] The blender project. `http://www.blender.org`.

[2] Gimp: The gnu image manipulation program. `http://www.gimp.org`.

[3] Love2d game engine. `http://love2d.org`.

[4] The programming language lua. `http://www.lua.org`.

[5] Spritesheet from layer groups script. `http://registry.gimp.org/node/27761`.

[6] The zlib license. `http://zlib.net/zlib_license.html`.