Software Systems Final Report

Nathan Lintz

May 2014

1 Introduction

For our Software Systems project, we developed a 2048 clone and an Angry Birds clone using Sprite Builder. We wanted to learn about Objective C as well as the game development framework Cocos 2D and SpriteBuilder. We found tutorials for these game frameworks online and implemented 2048 and an Angry Birds Clone for the iPhone and iPad. The code for our final project can be found here

https://github.com/nlintz/Software-Systems-Final-Project/

The resources we used in this project can be found here:

1. Big Nerd Ranch Objective C
2. Big Nerd Ranch IOS
3. `https://www.makegameswith.us/tutorials/getting-started-with-spritebuilder/installing-spritebuilder/`

2 Learning Goals

Our major learning goals were to be able to create a game using Cocos 2D and to become fluent in Objective C. Another learning goal was to compare the advantages of using Objective C and Sprite Builder over other game development tools. We will have achieved that goal if we have a working game that we can demo and if we can demonstrate that we have learned the Objective C. We prove that we have learned the Objective-C language in section 4 where we discuss the inner workings of the language and its syntax. To prove that we learned Cocos 2D and Spite Builder, you can see section 5 where we discuss our uses of these frameworks in implementing multiple games. Finally, if there is a good explanation of the advantages of Objective C and Sprite Builder, we will have achieved our learning goals.

3 Results

For our deliverable, we wanted to implement 2048 and an Angry Birds clone. Below are screenshots of the game we made to show that we achieved our stated goals.

3.1 Tutorial 1: 2048

The first game we wrote was a 2048 clone. In this game, tiles are moved around and combined until the user reaches 2048. We were curious about different modifications we could make to the original game. The main modification we put in was letting users pin a tile. This means that they can set any tell to be unmoved even when the rest of the tiles in their row/column should move. Theoretically this modification should make the game easier, however we found that it lead to more unpredictable game states making it actually more challenging for some users.

Below are some screenshots of the work we did on this game.

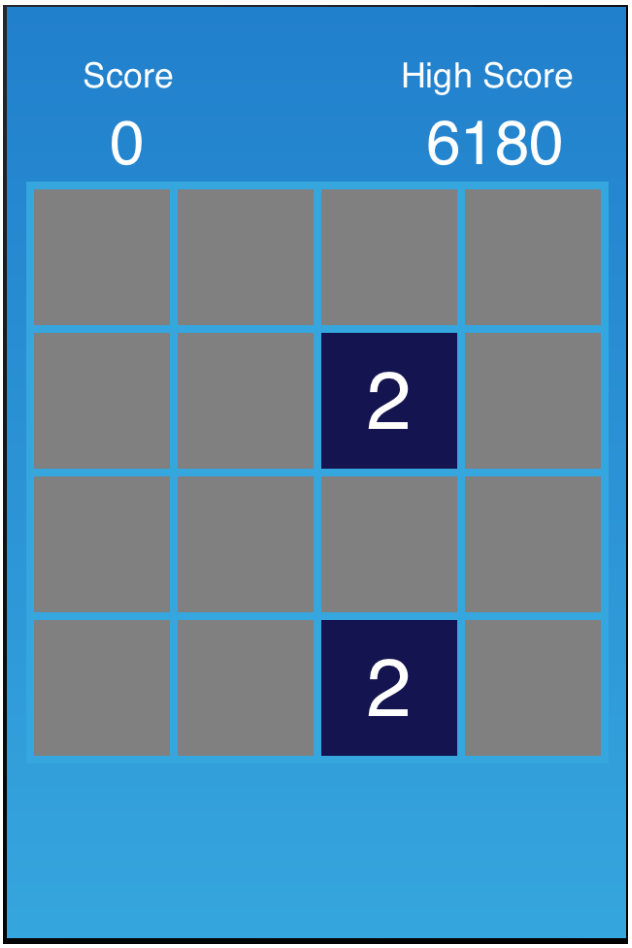

Figure 1: The User Starts a Game

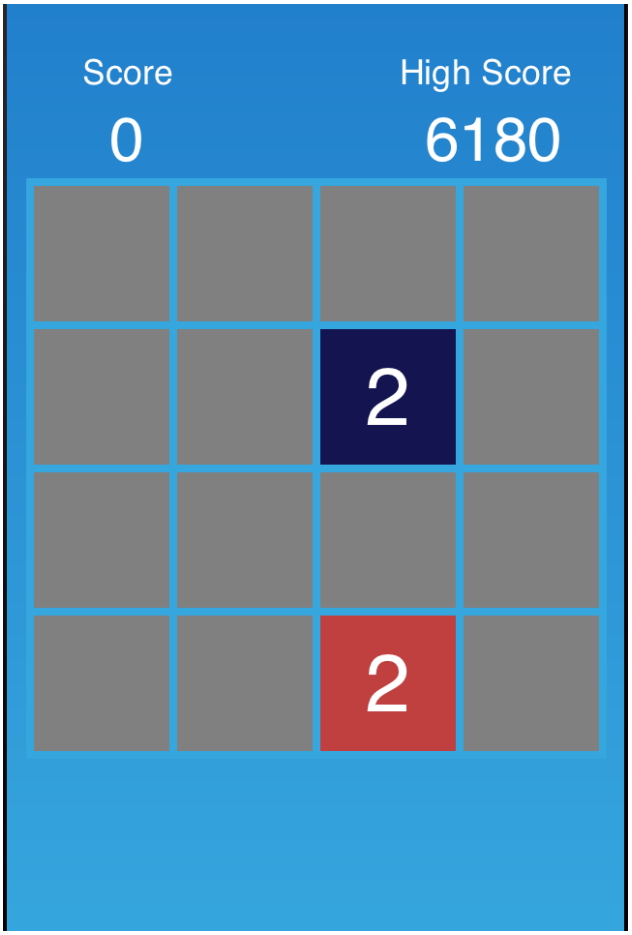

Figure 2: The User Pins a Tile

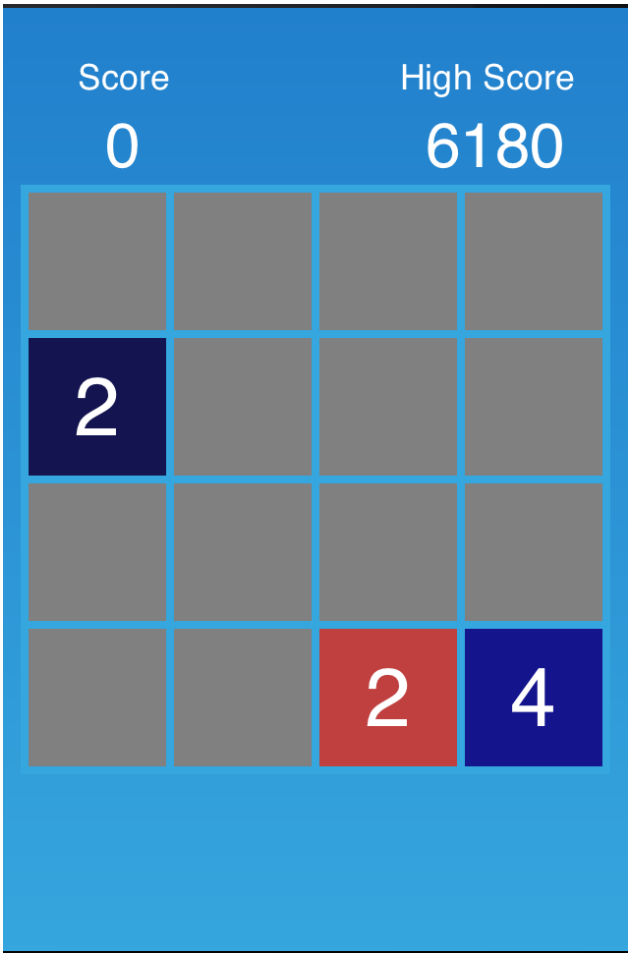

Figure 3: The User Swipes Left and the Pinned Tile Stays Put

3.2 Tutorial 2: Peeved Penguins

The second game we made was an Angry Birds clone called Peeved Penguins. In this game, there is a catapult which fires penguins at seals and tries to destroy all of the enemies on the screen. In this game, we got to explore many of the powerful features of Sprite Builder such as its physics engine and its UI builder. We had planned on writing a kinematics library to plot penguin trajectories however, we were pleasantly surprised to find that the physics engine handled that for us. A second feature of Sprite Builder that we used in this game is its animation and sprite building features. We wanted to animate a moving polar bear. Sprite Builder comes with a tool where you can set the timing and positions of a sprite's components and it will interpolate between the positions to generate an animation. This allowed us to create polished visuals without worrying about having to code keyframes.

To build on the tutorial, we implemented the infrastructure to have additional levels. We also created new assets for the game and changed the gameplay logic. For example, we changed how collisions are handled and we tracked which enemies were still alive using Key-Value Observing, Objective-C's answer to the observer pattern.

Below are some screenshots of the work we did on this game.

Figure 4: Opening Screen

Figure 5: That Bear Looks Ready To Shoot Some Penguins

Figure 6: Beautiful Collisions and Great Looking Particles!

4 Objective-C

Objective-C is the main programming language used throughout Apple's OS X and iOS operating systems. In many ways, the language is similar to C, with the addition of a number of key features and capabilities which allow it to be a much more productive, higher-level language.

4.1 Differences

While Objective-C remains similar in many ways to its predecessor, C, there are a number of key differences we discovered throughout this project.

4.1.1 Object oriented

Unlike C, Objective-C is an object-oriented language. Based upon Smalltalk, the language uses the paradigm of message passing to provide object-oriented features.

Unlike other object-oriented programs, in Objective-C, it is not possible to directly invoke a method on an object. Rather one passes a message to an instance of an Object.

This method passing system leads to a loose coupling in which the target of a method is actually resolved at runtime. In Objective-C, the syntax for this is as follows:

```
[obj method:argument];
```

This syntax is unlike the C++ or Java equivalents which look like

```
obj->method(argument);
```

4.2 Advantages

Two big advantages of Objective C are its garbage collection system, ARC, and its use of delegation for callbacks. Garbage Collection is implemented using automatic reference counting. With ARC, the developer labels instance variables as having a strong or weak reference. When an object has no more strong references to it, it will be automatically collected. With weak references, you can make objects have circular pointers and there will not be a memory leak. This is a big advantage over C because the developer never has to call free. Instead, they can architect their code intelligently and memory will be handled for them.

The delegation pattern is another big advantage over C. With delegation, you don't pass around function pointers to handle callbacks. Instead you make a contract between two objects and when a callback is needed, the contracted class is guaranteed to implement that method. This is a much better system than the C system which uses function pointers because you don't need to explicitly pass a function pointer which leads to tight coupling. Instead, you can write a contract and know that any objects which are the delegate will implement the appropriate methods.

5 Sprite Builder

Sprite builder is a powerful game development tool which integrates with the built in Objective-C game framework, Cocos 2D. It was developed by Apportable as a way to rapidly decrease game development time as it primarily uses a visual interface bridging the gap between designers and developers. We chose to use this tool because we were interested in learning more about game development and wanted to quickly prototype our ideas without wasting time navigating through Cocos 2D's confusing APIs.

5.1 Game Design and Implementation

One of the most powerful features of Spritebuilder that we enjoyed using was its scene layout UI. When programming games, the visual layout is key. Being able to layout your scene's visually is a much better work flow than laying out scene's programmatically because you can make small modifications easily and quickly see the results without having to rebuild your project. Below is a screenshot of the graphical scene builder tool that comes with SpriteBuilder.

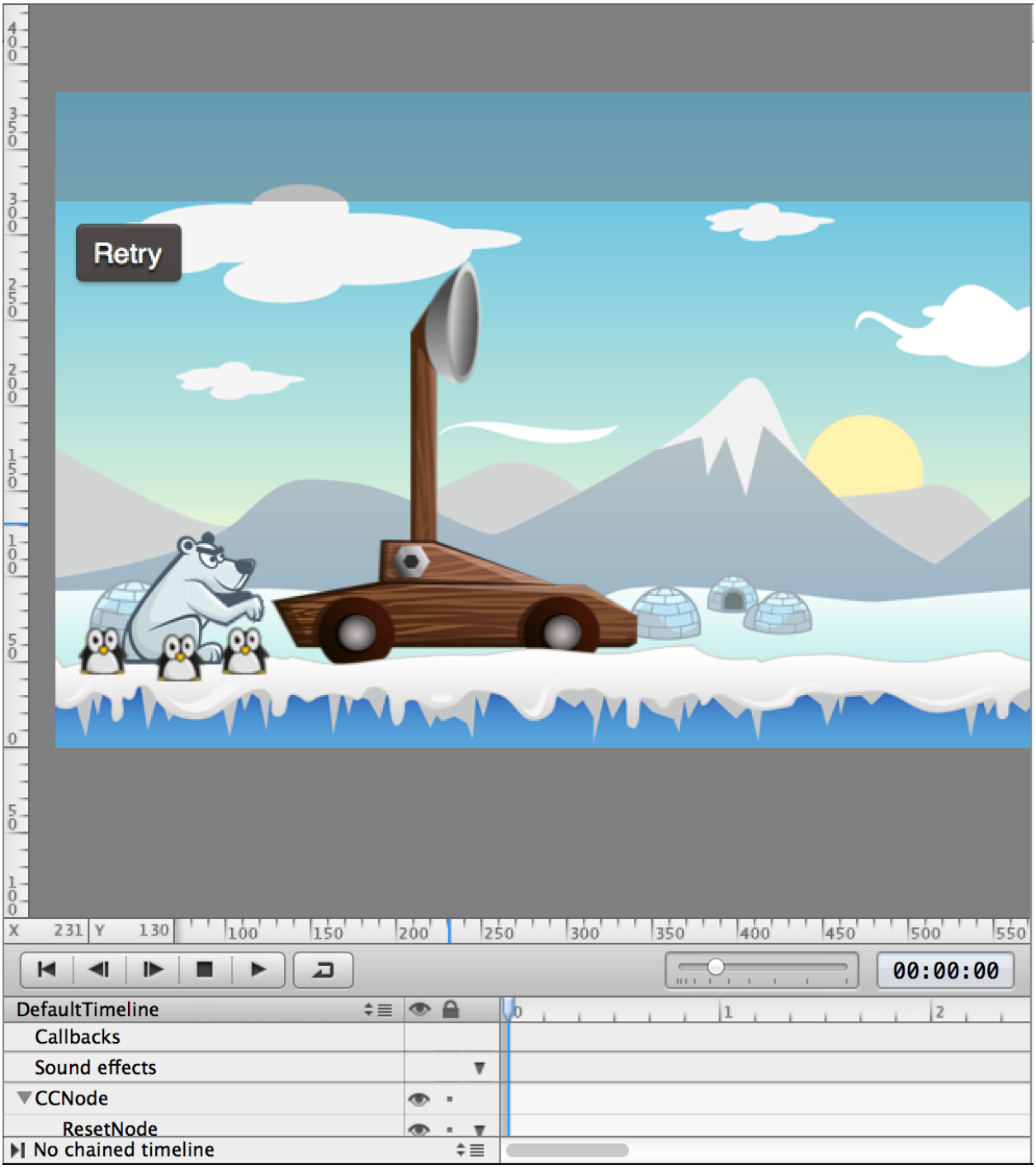

Figure 7: Sprite Builder Scene Builder

As you can see, this tool helped us design our game by making it easy to make fast visual modifications to our game.

Another powerful feature of Sprite Builder is the ability to create connections between the game visuals and code. Once you create an object in Sprite Builder, it is easy to connect it to a class

in your software by setting its class name. This feature allowed us to separate our visuals layer from our logic layer since the image assets aren't coupled to their class definition. Instead we got to create our visuals independently and give the object the same name in XCode and in Sprite Builder.

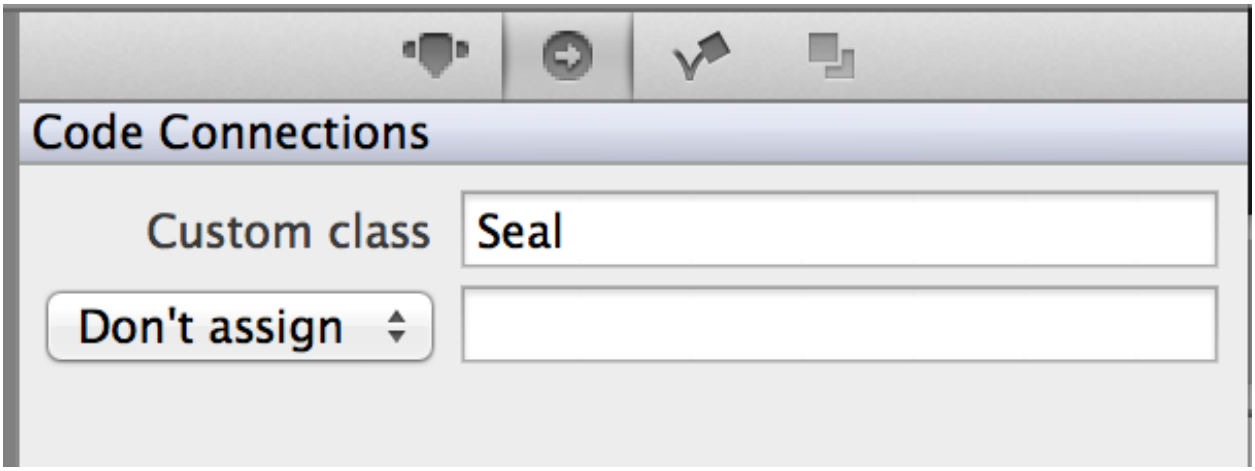

Figure 8: Sprite Object Connection Interface

Besides connecting classes, you can also connect buttons to their callbacks. This is a great feature because it makes creating menus simple. For our Peeved Penguins Game, we wanted to create a button that would bring you to the next level after completing the current level. Using Sprite Builder, it was easy to create the button, link it to a function, and then define the button's behavior in the code. We used this feature to layout all of our buttons and test them with stubbed callbacks which was much easier for us than having to define all of the button layout and logic and code.

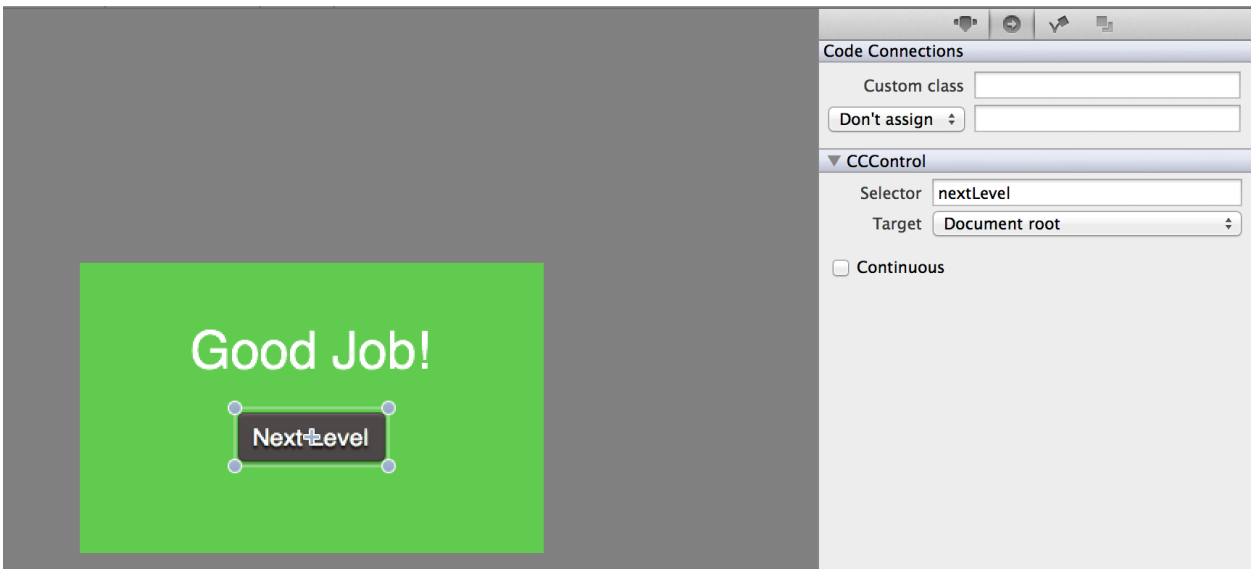

Figure 9: Sprite Object Connection Interface

5.2 Chipmunk Physics

Another powerful feature of Sprite Builder is the physics engine it comes bundled with: Chipmunk Physics. Chipmunk helped us in three primary areas: collision detection, computing kinematic equations, and handling spring-joint interactions.

Collision detection was a key feature of our Peeved Penguins since we needed to know when one of our penguins hit an enemy seal. In order to find collisions, we defined certain sprites as physics bodies and then Chipmunk creates callbacks for collisions between these bodies. For our game, we wanted to detect collisions between a penguin and any other object. A beautiful feature of Chipmunk is that it creates a named method for this type of collision called

```
(void)ccPhysicsCollisionPostSolve:(CCPhysicsCollisionPair *)pair seal:(
    ↪ CCNode *)nodeA wildcard:(CCNode *)nodeB
```

If you look closely, the method signature has "seal:(CCNode *)nodeA" because Chipmunk dynamically creates collision detection methods between objects which are declared as physics nodes. We then had to add logic to be executed after the collision and Chipmunk would handle the rest. This made detecting collisions and defining their results simple and easy for us.

Another powerful feature of Chipmunk that we used extensively is its kinematic processing. Chipmunk tracks the momentum and acceleration of your physics objects and plots their trajectories accordingly. Given that Peeved Penguins is based entirely on throwing penguins and tracking their flight pattern, kinematic handling helped us immensely. For our game, we just had to define which objects were physics bodies and when to shoot them and the kinematics engine did everything else. One of our favorite features is that the kinematics handler will detect the collective momentum of two objects that have collided which allowed us to write a custom collision handler for situations where the penguin hit a seal but had very little momentum e.g. hitting a seal while rolling on the ground. In these cases we would not want the seal to be destroyed. By having the physics engine handle momentum for us, detecting low velocity collisions was a breeze.

6 Conclusion

We believe that we achieved our learning goals. We wanted to learn Objective-C and Sprite Builder and given the results of our game, it is clear we achieved both goals. In the future, we would like to continue learning the rest of the tools in Sprite Builder and more advanced Objective-C tricks such as method swizzling.