Final Project Report

iPhone Game: SuperBall 3

Jazon Burnell

CS470

Spring 2009

University of Alaska Anchorage

Table of Contents

I. Overview

II. Requirements

III. Languages and Technologies

IV. Data Files

V. Data Structures

VI. Algorithms

VII. Development Process

VIII. Time Requirements

IX. Implementation Analysis

X. Conclusion

I. Overview

The main purpose of this project was to update a game I had developed for the iPhone called SuperBall. SuperBall, which was in its second version, was a block breaking game (a “breakout” clone) that featured a massive array of levels, a large selection of bonuses and unique features such as a two player mode.

SuperBall (actually known as SuperPong in its first inception) was one of five hundred applications available when the iTunes App Store first launched on July 11th, 2008, providing a large assortment applications available for download directly to iPhones and iPod Touches. Since then SuperBall has been very successful, with the free “lite” version being played on over a million iPhones and iPod Touches.

Although I am very glad I had the fortune of being able to get the game ready for the launch of the App Store, it did come at a cost. Between being relatively new to the language and platform, combined with the short period of time available to get the game ready, the first version of the game was hastily put together. Each successive update has been added on top of the original code, further compounding the complexity of the application.

For this project I decided to give SuperBall a much needed overhaul, requiring a complete rewrite of the majority of the code.

II. Requirements

The requirements for this project included updates and revisions to every aspect of the application.

3.1 User Interface and Navigation Specifications

- Design the navigation to be easily extensible and managed from a single controller. Completed.
- Provide a scrollable list for displaying high scores, each specific to the particular level set and difficulty it was obtained on. Completed.
- Online high score submissions to be submitted inside of app (without use of iPhone Safari web browser). Completed.

- Provide a help and how to section. Not Completed.
- Provide user interface for creating and saving level sets made via the level builder. Completed.

3.2 Game Play Specifications

- Improve game play rendering and performance using an OpenGL framework. Completed.
- Emulate impacts and collisions using a physics engine. Completed.
- Add particle effects for impacts and special bonuses. Completed.

3.3 Data and Model Layer Specifications

- Represent level information in standard XML format. Completed.
- Be able to save and retrieve level information via XML. Completed.

3.4 New Feature Specifications

- Create a level builder capable of easily and quickly designing levels with blocks. Completed.
- Have user submit-able level sets. Not Completed.
- Have user downloadable level sets. Not Completed.

Every requirement for this project was completed with the exception of a general help and how to play section, planned for a later version, and the user submit-able and downloadable level sets, which was put off for now due to the likely need and associated costs in getting a server that can handle a large volume of submissions and downloads of level set files, each of which can be upwards of 1 megabyte.

III. Languages and Technologies

This project used a large variety of languages and technologies. The majority of implementation was done in Objective-C, the standard object orientated superset of C used in Apple development. C++ was also used in numerous areas for the physics, as the used physics engine, Chipmunk, was written in C++.

HTML, MySQL and PHP was required on the web server side for high score submissions.

IV. Data Files

The main type of data file for this project are level data, containing block arrangements. These are organized into level sets, containing a set of levels and other information such as background image. Whereas in the previous version of the game these levels were hard coded, for this revision they were migrated to XML.

Aside from minor details like background image, the majority of data was block information. The following needed to be modeled:

- Type. What kind of block it is (e.g. normal or unbreakable).
- Hit points. If it is a normal block, the amount of hits required to destroy it.
- Shape. The shape of the block (e.g. rectangle or circle).
- Bonus. The bonus for destroying the block (e.g. wide paddle)
- Angle. The rotation of the block.
- Width. How wide the block is.
- Height. How tall the block is.
- X Position. X coordinate of center position.
- Y Position. Y coordinate of center position.

A sample for a level set with one level containing a single block would be:

```
<levelset>
	<level>
		<block>
			<type>0</type>
			<hp>3</hp>
			<shape>0</shape>
			<bonus>2</bonus>
			<angle>0.79</angle>
			<width>80.00</width>
			<height>40.00</height>
```

```
                  <x>160.00</x>
                  <y>240.0</y>
            </block>
      </level>
</levelset>
```

In practice, however, pure XML proved time consuming to load with the iPhone's limited IO capabilities. Because of this, I switched to a similar format supported by Apple called Property Lists. Although technically still XML, loading the files from disk was much faster, from around eight to ten seconds down to two to three seconds, which was much more expectable. The same level indicated above would be:

```
<key>LevelsArray</key>
<array>
        <dict>
                <key>BlocksArray</key>
                <array>
                <dict>
                        <key>Height</key>
                        <real>80.0</real>
                        <key>HitPoints</key>
                        <integer>3</integer>
                        <key>Shape</key>
                        <integer>0</integer>
                        <key>Type</key>
                        <integer>0</integer>
                        <key>Width</key>
                        <real>40</real>
                        <key>XPosition</key>
                        <real>160.0</real>
                        <key>YPosition</key>
                        <real>240.0</real>
                </dict>
        </dict>
</array>
```

The property list data is still easy to maintain and edit. The only disadvantage was an almost 2x increase in file size for level data. At present this is not a major concern, but the in-

creased file size would significantly increase the burden of a server if level submission and downloading were implemented.

V. Data Structures

Apple's Dictionary class, NSDictionary, was used extensively for modeling various types of data inside the application, like level data. NSDictionary has the advantage of being an organized way to maintain data with fast enumeration and simplified saving and retrieving data from disk.

Object wrappers were created for each type of data I needed to maintain. An abstract class, DictionaryInfoWrapper, was created to provide the functionality of accessing information from the dictionary. Subclasses were then created to model each type of data I need to represent, including LevelSetDictionaryModel, LevelDictionaryModel, and BlockDictionaryModel.

VI. Algorithms

A lot of the complicated algorithms associated with a physics based game were luckily handled by the physics engine. There were however a couple unique areas that required specific algorithms, especially relating to some of the bonuses that required special logic.

One of the neat things I was able to do with the physics engine was to cause explosions to actual impart a force on the surrounding blocks, causing them to be blasted away. Essentially, once an explosion block was detected to have been destroyed, its position was sent to a method that then iterated over each block determining how much force it should receive in response to the explosion as follows:

```
for (BlockElement *block in blocksArray) {
	if (block.isAlive && block.type != kBlockType_Puzzle) {
		float distance = [CGExtension distanceFromVect:p toVect:block.center];
		if (distance < BOMB_BLAST_RADIUS) {
			[block explode];
			[[GamePlayController sharedInstance] ball: nil hitBlock: block
										atPosition: block.center];
		}
		else {
			CGPoint unitVec = [CGExtension unitVectorForPoint:
						CGPointMake(block.center.x - p.x, block.center.y - p.y)];
			float movementVelocity = BOMB_IMPACT_VELOCITY - distance;
			block.vector = cpv(block.vector.x + unitVec.x*movementVelocity,
			block.vector.y + unitVec.y*movementVelocity);
```

```
                }
        }
}
```

Another bonus that required special logic was autopilot, a bonus that automatically moved the paddle to hit the ball. With a single ball, such an algorithm is fairly simple, but with multiple balls it gets slightly more interesting:

```
BallElement *ball = [ballArray objectAtIndex:0];

int i;
for (i=1; i<[ballArray count]; i++) {
        BallElement *ball2 = [ballArray objectAtIndex:i];
        if (ball2.vector.y < 0 && (ball2.center.y < ball.center.y))
                ball = ball2;
}

[self movePaddleTo: CGPointMake(ball.center.x, STARTING_PADDLE_HEIGHT +
 [SettingsController sharedInstance].paddleHeightOffset)];
```

VII. Development Process

Prototyping was used extensively in the development of this project. I had the fortune of gaining the interest of previous version users who were interested in beta testing. Once I had developed enough to where the game was playable and had basic navigation, I began sending builds out. In total, there were four beta prototypes sent out.

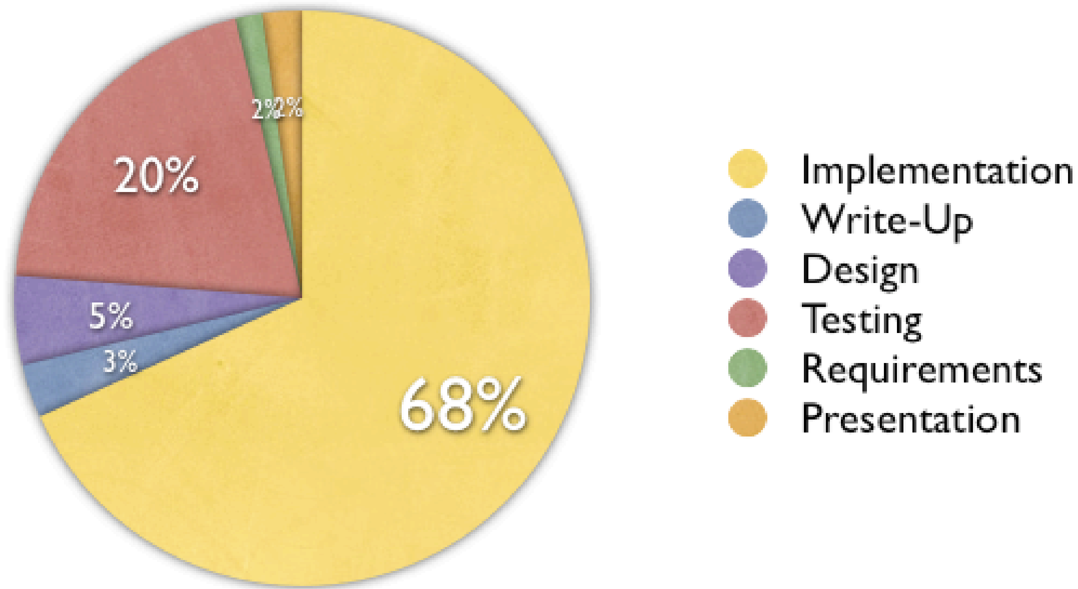
Beta one included just the basic gameplay, containing the blocks, paddle, and bonuses, and the basic navigation required to start a new game. It also including the first version of the level builder, which was one of the first things I started developing so that I could get feed back on the ease of use in creating levels.

Beta two added the basic gameplay status items like scoring and lives left, along with their respective gameplay UI overlays. It also addressed a lot of the early bugs found in power up implementations and in the level builder, and I changed some things like ball speed based on feedback.

Beta three added a precursory version of everything not yet functional, including a lot of UI for items not directly related to gameplay, like maintaining the high score list, and being able to submit high scores online. The two player mode was also added in at this point.

Beta four was the final pass through everything to make sure everything was complete and ready for submission. The original intention was actually for beta three to be final, but enough bugs were found that it warranted sending out one more beta to make sure no other significant bugs remained.

VIII. Time Requirements



Total: 400 Hours

Requirements: 6 hours
Design: 20 hours
Implementation: 273 hours
Testing: 80 hours
Write-up: 12 hours
Presentation: 9 hours

My initial estimate was closer to about 300 hours, which as indicated above was overshot by more than 30%, ultimately totaling about 400 hours. The single largest amount of time was invested into the level builder. Due to the repetitive nature of designing levels, it was always my preference to add features to the level builder rather than to have to manually modify the XML files to do things that would have otherwise been difficult to do in the level builder. This led to

the implementation of new features, like selectively controlling the multi-touch aspects and the ability to copy and paste blocks, which required a larger amount of time.

It is also worth noting that the initial 300 hour estimate included user level set downloads and submissions. Many aspects of this were actually implemented to a certain degree, particularly related to the required UI aspects, but the server-side implementation of downloading and submitting level sets would definitely have required a significant time requirement on top of the 400 hour total.

IX. Implementation Analysis

Final implementation utilized numerous shared instances. Upon loading the application (from applicationDidFinishLaunching: in SuperPongDelegate) the application allocates the GameController instance. This object holds application information like what state the game is in, and controls saving and loading previous games.

The GameController controls what aspect of the application the user is currently in. There are three UI states, Menu Navigation, Gameplay, and Level Building, each with their own respective shared instances. This proved to be a very useful design since it allowed the application to switch to any state at any time. Simply calling the method that shows the level builder, for example, will automatically remove and deallocate all menu navigation items if the user is currently navigating the menu (e.g. just tapped the level they want to edit) or automatically remove and deallocate all gameplay items if they are currently playing a game (e.g. just selected edit this level if such a feature were implemented).

Memory was managed by the shared instances. Whenever the GameController switched state, reset was called on the shared instances to release all unneeded objects. In the case of navigation, handled by MenuController, this involved making sure every bit of navigation was released immediately. Likewise for the level builder, handled by LevelMakerController. In both cases the primary motivation was ensuring maximum performance during gameplay. However in the case of resetting the GamePlayController, certain objects are cached and reused, like particle effects, to maximize performance. Memory for these objects is held as it has little effect on the navigation and level building since gameplay is the only place performance is a concern.

The GamePlayController contains the game loop, in update:, that is the beating heart of the application during gameplay. The GamePlayController also references all of the important

objects for controlling gameplay, including the GamePlayViewsController for maintaing game element views and particle effects, the GamePlayModelController for controlling the physics and underlying gameplay models, the GamePlayValuesController for storing the score, lives etc, and the GamePlayUIViewsController for updating the score and other on screen labels.

Storing and saving levels is handled by the LevelSetDataController. Primarily this class maintains all of the XML files for representing level sets. Initially this worked well for simply loading pre-created files from disk, but once this class started to also be used for creating and saving all of the user built level sets the complexity quickly rose. Due to the unfortunately unforeseen issue of the lengthy load times of bringing the XML files into memory, a separate way of storing level set information like title and number of levels was needed since loading in all of the level sets would have been prohibitally time intensive. However, having these separate structures required that some aspects of level set data had to be stored in both places, requiring mechanisms to deal with possible inconsistencies. In hindsight, it appears it would have been worth the time to implement a SQLite back end, since the hydrating and dehydrating features would have provided a better and more flexible design.

X. Conclusion

Overall I am very happy with everything I was able to do with SuperBall 3. The greatest accomplishment was being able to provide an update to current users, which thus far has been received very well.

It has also been a useful experience sending beta builds directly out to actual users. This method of prototyping proved invaluable in many aspects, and having set up everything required for this type of Ad Hoc iPhone distribution will be very useful in the future.

Lastly, I learned a great deal about OpenGL rendering and physics engines. The knowledge and tools I learned will undoubtably be very useful in the future.