Dave Lyon's Senior Project: Space Game for iPad

Over the course of the winter semester of 2011, I have worked on an interactive 2D game for the iPad in order to expand my knowledge of both iOS development and project management. Overall, I am very pleased with the final result and hope that you will find that the progress made is in line with the outline presented at the beginning of the semester. While I was not able to get to every feature I had envisioned originally, I am quite satisfied with those I was able to achieve, including: an endless play style, sound and particle effects for collisions, relatively accurate collision detection, and touch interaction. In addition to providing an overview of the above features, I would also like to discuss the tools and libraries that I used to aid in the development of this project, including XCode, Cocos2D and the Objective-C programming language. But before diving in to the technology used in creating my game, I would like to explain the object of the game and how it is played.

Space games are far from new in the video game field; from Galaga to PixelJunk Shooter, space games have been around since the beginning, and will likely continue to evolve with the industry. The basis of any space game is more or less the same: you're the last hope for humanity in some sort of small ship, and you have the incredible task of singlehandedly saving the universe by destroying all of the enemy's ships. You are, of course, given much better equipment and a great deal more durability than your opponents, but this seems only fair given that you are up against a seemingly endless stream of evil aliens bent on total annihilation. So what sets my game apart from the pile of antecedents? First, the game is designed to be more casual and relaxed. While you are still the 'one last hope' for humanity, you aren't able to actually lose the

game. Instead, the game will simply reset its difficulty when you are overwhelmed and you'll have to build back up again. In addition, it's possible to lock the difficulty level in order to prevent the game from being more difficult than the player would like. In essence, the game is made to be very casual, and to provide an experience that is designed for a mobile device — that is, the game should lend itself to play sessions of five minutes just as well as five hours.

It is important, however, to consider that there are certain conventions expected in such a game, and to provide tropes that the player might expect. In support of this goal, the game contains the expected health system, power-ups, and waves of enemies. The reasoning behind this is to create a sense of familiarity and to help people quickly understand how the game is played. While games that provide a radical gameplay twist have often proven to be wildly popular, I have decided to take the less radical approach of using a common gameplay formula and adding a simple, but challenging twist.

Tetris, the famous Soviet game enjoyed by millions all over the world, has a very simple but addictive formula that lends a unique re-playability to what might otherwise have been a very dry and boring game. While the game is largely the same every time, the pieces are different, and thus the each game feels like a unique experience. In addition, the point of Tetris is to hold out as long as possible before losing — it's inevitable. What keeps people coming back is the drive to make it just a little bit further than last time; to beat ones own best score. The impetus for returning to this game is the same: Though it is a certainty that you will be overwhelmed, it is the drive to beat ones previous best that will keep people coming back. In addition, the ability to randomly generate waves of enemies means that much like Tetris, the game will feel like a unique experience each time the player begins. The pieces are the same, but how they fall will be

different. My hope is that by combining a familiar space game with Tetris like re-playability, people will be intrigued, and enjoy playing my game.

The final test of my idea will be releasing it to the App Store; allowing people to vote with their wallets as to whether or not I have created something interesting. Releasing to the App Store is really the ultimate test of any developers abilities, ideas and drive. A game released on the App Store can put your game to the test on millions of devices all over the world, and any of those users can provide reviews and feedback (and often will!).

In addition to the reach of the App Store, I chose to make my game for the iPad because of the extremely flexible tool chain provided by Apple for developing apps. Apple provides an excellent editor with testing and debugging environments built in, along with extensive documentation all for free! This makes it an excellent choice for developing a new project for next to nothing up front. Further, there are numerous well developed and well documented libraries to ease the burden of developing apps for iOS. And of course, it cannot be denied that experience is truly the key to success of any project, thus it was helpful to stick to a platform with which I was already familiar.

The choice of language when developing a game has a profound impact on every aspect of development, and is certainly of the most important decision to make when beginning a new project. As a game grows more and more complex, it becomes necessary to abstract as much as possible in order to enable more rapid application development, as well as to avoid coupling parts of the code. The looser the coupling of objects in the codebase is, the easier it will be to make drastic changes to your game, or to introduce seemingly huge new features. To take an example directly from the game, adding new enemies is very easy as all

of the common code has been pulled up to a common base class; thus adding a new enemy is simply a matter of subclassing, and overriding the movement method. Objective-C makes these kinds of abstractions very easy to do by providing a very minimal object layer over general C code. The odd combination of ideas from Smalltalk and C make Objective-C a unique language, but also a very powerful one. This flexibility is another of the many reasons I chose to develop this game on the iPad. Thanks in no small part to the popularity of the iPhone and App Store, there is also an excellent library called Cocos2D that has been ported to Objective-C from python that provides just enough of a base to enable rapid development of iOS games. This, and other libraries provided a base level of abstraction from the lower level of OpenGL and similar tools that enabled much more rapid development, and also helped greatly to optimize the game engine with relatively little effort on my part.

"Don't reinvent the wheel," is common, but perhaps controversial advice in computer programming. If you don't understand how the wheel turns, can you really use it effectively? In this case, I felt that my previous experience in the game programming class had helped me to have a good understanding of how the graphics pipeline worked, and in this case, I opted not to reinvent the wheel. I used a number of libraries in order to build on pre-optimized code, with well documented abstractions in order to help get more done, in less time, with fewer bugs. Overall, I believe this was a good choice, as I truly felt that I was able to solve problems quickly, and have thus far had little trouble maintaining the 60 frames per second (FPS) target I set for myself. I owe this mostly to the wonderful Cocos2D iPhone library; and also to a subset called Cocos2DDenshion which provided nice, simple audio processing. Cocos2D provides an excellent base for creating a game: a "Scene" for the current section

of the game being played, which has “Layers” to encourage separation of concerns for game actors and interface elements, and at the most basic level “Nodes” that provide hooks for animation and other actions, many of which are provided pre-configured.

The Node class proved by far the most useful feature of the library. Providing a small layer of abstraction over the Objective-C base class of NSObject, CCNode is the base class of everything in Cocos2D and allows anything from individual sprites, to the entire scene to have timed animations applied. This made creating enemy paths, and the incredible star background relatively simple: the animation code was there, I simply had to decide how to use it. This also avoided any strange race conditions, or unoptimized OpenGL calls I might have introduced on my own. In addition, the Cocos2DDenshion library made adding sound to the game painless. Finally, thanks to the excellent design of the Cocos2D API and the wealth of available documentation I was able to learn a number of best practices for game design including: using sprite sheets to save memory and allocation time, proper design of sound objects, and using the Command pattern to govern actor movement. Overall, I’m glad I chose to use these libraries, and focus instead on the gameplay and overall design of the system.

The overall design of the game evolved numerous times over the course of the project, and is still not necessarily in what I would consider a perfect state. This is not to suggest it doesn’t work, but simply that at the current point, it can still be difficult to add new features, or to refactor existing ones. A particularly troublesome area of the codebase is collision detection; specifically where it should be, and what object should be responsible for the actual detection. At present, this chain of events is managed by the actor layer. While it seems

logical for this object to be responsible for the dispatching of such messages, it seems less than ideal to have the actual collision handled there, and should instead be pushed to the actors themselves. Further complicating this interaction is the fact that destroyed objects should be replaced by particle explosions, meaning that the actor layer must manage the appearance of one object, and removal of another as close to simultaneously as possible. But presently, there is no noticeable performance loss, and thus I am satisfied that it works well enough to ship.

Another challenging area was the infinite and random star background. Overall, I probably spent more time on this than I really should have, but frankly it was too cool not to get perfect. The initial implementation involved simply creating a list of points where the starts would appear slightly offscreen, stream across the screen for some random amount of time and then be removed once off screen. This proved to be very costly to performance when dealing with large numbers of stars since each new star required memory to be allocated, which is bad practice for memory management and slow. What was needed was a way to keep the appearance of new stars floating by, but without needing to allocate more memory for each new star. After getting the look and feel for the stars working, I discovered that Cocos2D Nodes could be scripted to call a method, and could be given a sequence of events to handle. By combining these with a simple ‘reset’ method, I was able to get the consistency and overall aesthetic I wanted, and a fixed memory cost at startup time. Instead of re-creating the stars over and over, I introduced a small random delay after they moved offscreen after which they will jump back to the top and move down again for a random amount of time. As well, there is a small amount of random x-axis displacement to ensure that the stars do not appear to be on a grid. Thus, while the base

script is the same, the values are random and give the appearance of infinite variability. An accidental discovery was that by having stars at varying speeds, it was possible to create the effect of some stars being more distant than others, even though the sprites are all the same size. Without a doubt, the star field is the most interesting and aesthetically pleasing features of the game.

In addition to the programming aspects of this project, I was also interested in learning how to better plan and manage my time on a large project. I consider myself to have been mildly successful here, but also recognize that there is some room for improvement. I tried to estimate the amount of time that certain aspects of the game would take to implement, and to hold myself to those times; unfortunately I often found myself taking significantly more time on features due to refactoring or feature creep. The largest piece of feature creep was the aforementioned star field. Retrospectively, I recognize that I should probably have shelved this early on and come back to it, but found myself got caught up with how cool it was. Also costly was touch interaction with respect to player movement; while simple to implement getting the right balance of drag to movement proved extremely difficult to perfect. As touch interaction is the basis of the whole game, it seemed logical to spend as much time as necessary to make the game 'feel' like it played smoothly and fairly. To wit, this quickly became a drag on development. However, the touch implementation currently feels very good, and was definitely worth the time invested. However, I now recognize that this is probably something that could have been shelved until other features were in better shape. There are no clear lines on where to cut development on one feature to focus on another, however in hindsight, it would have helped to simply allow myself to say "Good enough."

Overall, this project has provided a great learning experience in both

development and management. This project was fun, and easy to develop thanks in no small part to the libraries available, and the large amount of support and documentation made available online. With the help of a firm deadline, and an overextended list of features, I've also learned a great deal about how to estimate my projects, and how to manage my time. As well, I've learned the importance of being able to say no, even to yourself. Though there is still a great deal of work to be done to make this application ready for the App Store, I am proud of my accomplishments this semester.