

*Orbit*

Ken Hung

Advisor: Dr. Zoe J. Wood

June 2012

Abstract

Developing a professional quality game is a time consuming process. It spans multiple disciplines and even sub-disciplines. In this paper we discuss the implementation details of Orbit, a 2D side scrolling mobile game with innovative features and a strong focus on coding for maintainability,

**Table of Contents**

1. Goals and Motivations
   1. Why Apple’s iOS Mobile Platform? 3
   2. Gameplay and Innovation 4
   3. User-Generated Content 4
   4. Maintainability 5
   5. Model-View-Controller 5
   6. Code Refactoring 6
   7. Usability 6
2. Related Work
   1. Inspirational Works 7
   2. Comparative Analysis 9
3. Algorithms and Implementations
   1. Overview 12
   2. Game Run Loop and Entity Design 12
   3. A\* Path finding and the Grid 13
   4. Level Editor and File Design 15
4. Results
   1. What Was Done Well 18
   2. What Needs Improvement 19
5. Future Work 20
6. Bibliography and References 21

**1. Goals and Motivations**

Video games are a popular entertainment choice. They offer an experience like no other by combining storytelling, interactive engagement, creativity, and problem solving to create total immersion. The following are overarching goals set for Orbit to achieve immersion through effective gameplay and build maintainable code for long-term feature additions.

1. Develop a two-dimensional (2D) game on Apple’s iOS mobile platform that focuses on gameplay, game design, innovative features, and user-generated content.
2. Focus on code maintainability, reusability, and UI usability.

1.1. Why Apple’s iOS Mobile Platform?

Mobile computing represents a big shift in how content is served. While well known game development companies are focusing on triple A desktop and console based titles, mobile platforms offer indie developers an easy way of distributing their own creative games and generate revenue at the same time.

Mobile games also help enhance immersion. Touch devices like the iPad give players a more interactive experience by building intimacy and depth with their fingers. Controlling a character with ones’ finger is much more involved than using a mouse or joystick. Mobile devices also allow users to play whenever and wherever they please. This mobility also makes it easier for players to share their experiences and gameplay data and to play cooperatively.

The two most popular mobile platforms: Android and iOS offer distribution and feedback methods that have little overhead for developers. Both platforms are highly flexible, design oriented, and well supported by various online communities as well as Google Inc. and Apple Inc. respectively.

However, the underlying tools Apple built for iOS have a lower learning curve then Android. Xocde is Apple’s Integrated Development Environment (IDE) for all OSX and iOS development. The usability and responsiveness of Xcode is more refined than Eclipse IDE in terms of project structuring, compilation, debugging, and code completion. This means less time is spent figuring out how to use the tools and more on development time.

In addition, the iOS developer community provides a richer knowledge base of solutions through convenience methods, libraries, category implementations, answers to common questions, and sample apps. This could be because iOS was released before Android. Regardless, this means unique problems can be discovered and tackled faster as less time is spent trying to solve already solved problems.

1.2. Gameplay and Innovation

Gameplay keep players interested. The gameplay’s style, complexity, and intuitiveness are big factors for playability and usability. A 2D scrolling shooter style was chosen because it provides tried and true gameplay that can be easily built on with innovative ideas. The majority of successful app games on iOS are in 2D because the tools and solutions available are developed and easy to learn, but tools for three-dimensions (3D) have only recently emerged and the knowledge base isn’t as mature yet. Learning 3D tools would require significantly more time. In addition, the estimated amount of time and complexity to develop a quality three-dimensional game was too high.

As mentioned earlier, the 2D scrolling shooter gameplay style can easily be built on with innovative ideas. Orbit has features commonly found in current mobile 2D games but some features completely new or rarely found. For example, instead of the traditional gameplay of shooting projectiles to destroy enemies, Orbit gives the player weapons that orbit around the player’s ship. The player has to destroy enemies by getting close to them with a weapon, but players must also try to avoid being hit by enemy units. Since all current iOS devices have touch capabilities, the player’s ship is controlled with a finger. Not only does this improve interactivity but it also adds gameplay depth by putting emphases on precision, skill, and reaction time. Orbit also has another feature that builds on the idea of sharing scores and achievements. Although it’s not a new idea in video games, it is uncommon in mobile games. This feature is the level editor for user-generated content.

1.3. User-Generated Content

User-generated content via a level editor alleviates the need for developers to constantly create more game content. It also extends the longevity of gameplay and adds another kind of content for players to share beyond scores and achievements. Most existing game apps don’t include this feature because their revenue stream relies on players purchasing new versions of their game. While this is a good reason to maintain revenue, apps could incorporate both a level editor and have new version content releases.

Players could generate and share new levels while new versions would add other features such as graphical updates, tooling updates, or add new kinds of enemies or weapons. A level editor would also buy development time for new content releases. Level sharing also encourages feedback on long-term game features. While players design their own levels, issues and flaws will be revealed quicker which improves the game’s quality. This long-term approach means code maintainability is especially important.

1.4. Maintainability

A common issue that occurs in software development is that a rush to release causes poorly designed code. The project may be working in the short term but in the long run the project becomes difficult to maintain. Each additional feature becomes more time consuming to implement than the last. During Orbit’s development new features were being brainstormed and added at the same time. A clean object oriented design had to emerge that is flexible enough to add changes to but rigid enough to for any code reviewer to immediately understand it.

Therefore Orbit’s development process followed an informal agile approach with a time constraint of 20 weeks. A list of core features and a feature completion outline was created to track progress. Bi-weekly updates of project status also gave some visibility of progress. Although there was no burn down chart to measure progress or formal sprints to divide tasks, a backlog of core features was kept to determine next features to implement. The backlog was also compared with the bi-weekly status reports to handle project scoping. To help with initial object oriented designing, Orbit followed the MVC design pattern.

1.5. Model-View-Controller

**Model-View-Controller (MVC)** is a software design pattern used in Orbit. It outlines a clean way of ensuring component modularity, code reusability, and thus maintainability. The pattern works well in Objective-C and it is what iOS frameworks are built on. The **Model** represents any non User Interface (UI) component. In Orbit, this includes the game states, game logic, Artificial Intelligence (AI), the Box2D physics engine, and the file persistence component of the level editor. **Views** represent all UI components. In Orbit this includes the level editor UI itself, the table views, menu screens, the Heads Up Display (HUD), and the drawing of sprites, motion streak effects, and animations. Lastly, the **Controller** serves as the bridge that connects the **Models** and **Views**. It handles all the method calls and data exchanges necessary for **Model** and **View** synchronicity.

In the context of mobile apps, MVC makes perfect sense. Across all device form factors **Models** should be reusable with little to no change. Several different **Views** should be implemented to accommodate different device types and screen sizes. **Controllers** should be the only components that need redevelopment, but similar to **Views**, several different **Controllers** should be implemented to communicate with their respective **Views** while reusing common **Models**. To help enforce MVC, Orbit went through several code refactors.

1.6. Code Refactoring

Two major code refactors occurred to help restructure prototyping of new features. The first involved refactoring all Box2D and sprite initializations and their synchronizing into Entity objects. This meant only one method call is needed to initialize everything thus making it easier for any future maintainer to figure out.

The second refactor involved making the game state and HUD modular by removing them from the main game layer. The game state became the Model in MVC. All game state variables that were previously scattered throughout several files were aggregated into one single file for readability. In addition, a different game state could easily replace the current one in the future because of how decoupled the variables are to the main layer and Entities. The HUD could now be attached to multiple layers if needed or multiple HUD versions could be dynamically swapped. This also means that if any usability issues arise from feedback, fixes can go out faster.

1.7. Usability

Usability and playability are similar terms. The intuitiveness of controls and UI can affect the success and appeal of any software. To help improve usability, several play tests were done to determine how the game is played and what changes were needed to make the experience better. Apple also enforces their Human Interface Guidelines for all submitted apps and ensures that developers create apps that live up to Apple’s user experience standards.

**2. Related Work**

2.1 Inspirational Works

Orbit is inspired by two games in particular. Both games were used as references for gameplay and artistic style.

**Tyrian 2000 – World Tree Games**

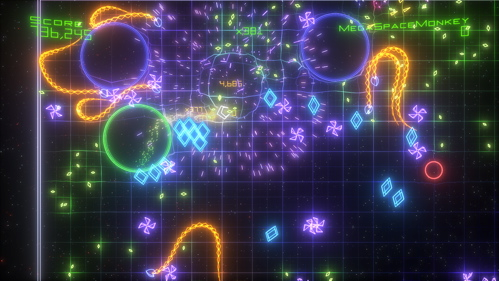
Tyrian is a top-down 2D shooter. It follows an arcade style of gameplay. It’s fast paced with a very incentivized game progression system revolving around weapons. One of the main appeals of Tyrian is the way weapon upgrades are handled. There are upgrades for the ship itself, several types of weapons, shields, and generators. The increasing flamboyant sprite animations and their increasing power made each upgrade very satisfying. The pacing of enemy unit numbers, their movements, and placement of currency throughout each level created a very consistent but unpredictable gameplay experience. Each level consisted of enemy units with predetermined movement paths followed by an end boss with it’s own series of paths, attacks, and special animations and abilities.



Figure 1: An in-game screenshot of Tyrian 2000.

**Geometry Wars™: Touch – Activision Publishing Inc.**

Geometry Wars is also a 2D arcade style game that also has fast paced gameplay with a focus on shooting projectiles at fast moving enemies. Its appeal comes from how flamboyant the bloom effects, grid effects, animations, and particle explosions are. It primarily revolves around score gathering and the motivation to keep playing comes from increasing difficulty. Overall it is a beautiful game even in 2D.

Figure 2: An in-game screenshot of Geometry Wars™: Touch.

Orbit is inspired by these two games having similar gameplay to Tyrian and an artistic style similar to Geometry Wars. From these two games, it was determined that Orbit’s gameplay should include: side-scrolling, enemies with predetermined paths, an end level boss, and satisfying upgrades. Orbits artistic style should include glowing enemies and characters, exaggerated explosions and animations, and grid effects.

Both games however shared the common cliché of shooting projectiles in a generally straight direction. While play testing both games, it was realized that the player was stressed only when enemies came very close to hitting them and that straight projectiles gave too much safety and gameplay quickly became boring. Orbit attempts to alleviate this issue by having orbiting weapons. This constantly puts some stress on the player because they have to maintain their offense and defense at the same time.

Another issue from Tyrian 2000 and Geometry Wars is that once all the levels were complete, there was no new content left. The level editor is a solution to this. It gives users the ability to create their own levels once the default game levels have been played through.

2.2 Comparative Analysis

Once Orbit’s gameplay and artistic style was narrowed down, a comparative analysis was needed of current popular mobile game apps with similar characteristics. Of the millions of game apps on Apple’s App Store, the following two were the highest rated in the “Space Shooter Arcade” category.

**Meteor Blitz – Alley Labs**

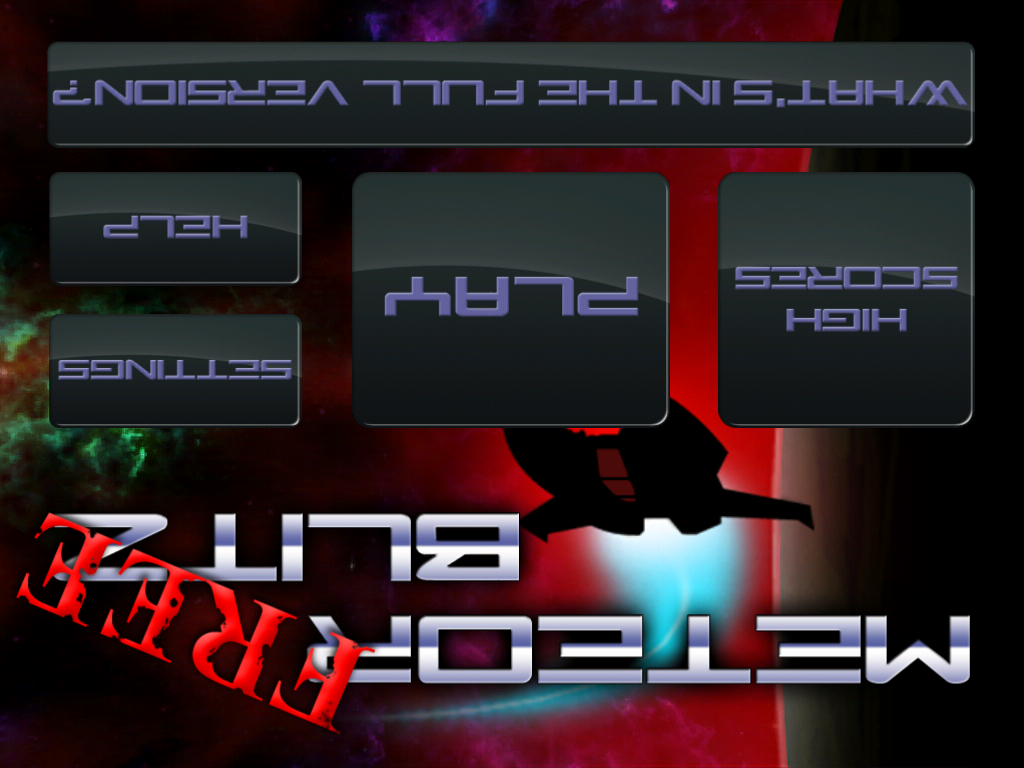


Figure 3: The main menu and an in-game screenshot of Meteor Blitz

**Space Invaders Infinity Gene Lite – Taito Corporation**

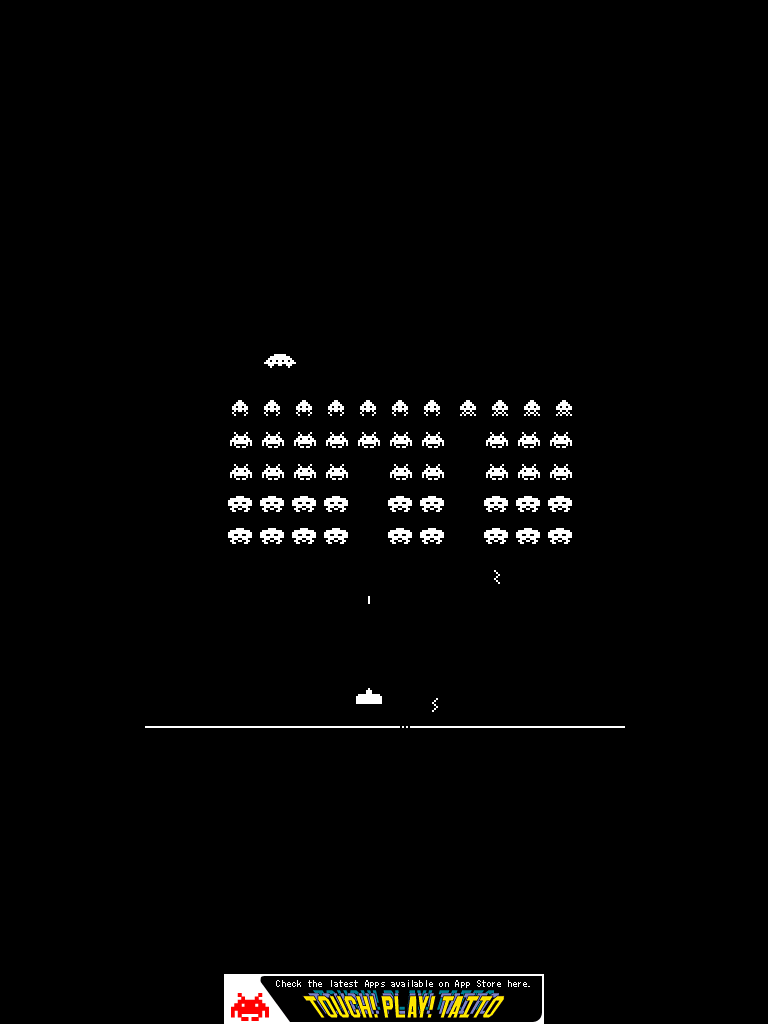
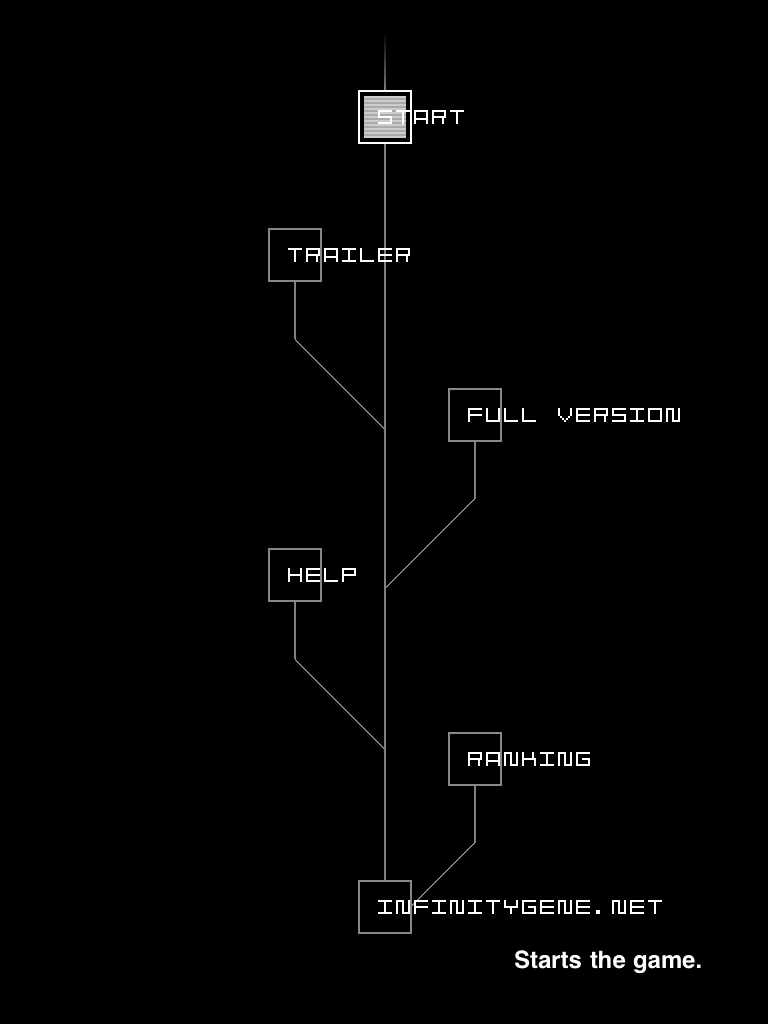


Figure 4: The main menu and an in-game screenshot of Space Invaders Infinity Gene Lite.

The comparison results of Meteor Blitz and Space Invaders can be summarized as follows:

Common features that should be included in Orbit:

* Main Menu - Settings, Help, “About Us” Website Link.
* Flamboyant animations, particles, and effects.
* Not much focus on story.
* Arcade feel and emphasis on gameplay.
* Pausing mechanism.

Common features to improve upon

* Sharing high scores.
* Firing straight projectiles.
* Weapon upgrades and bonus weapons.
* Lite or freemium versions that serve as demos to full paid versions.
* Virtual game pad.

Most of the common features that were incorporated into Orbit were features that are common to most arcade style games. For example, a main menu offers a non-linear flow of communication giving players choice and is overall a good UI element to have. Flamboyant animations and effects were also important to have because they grab the player’s attention. It is apparent not only in Meteor Blitz and Space Invaders, but also in Tyrian and Geometry Wars.

The typical mobile game is usually an arcade style game and many mobile gamers have come to expect it. An example user story would be a player who is commuting to work so he plays a fast paced arcade game to entertain himself for a few minutes, then pauses it to finish later. This interrupt mean story becomes less important to immersion but it does vary from player to player. However for Orbit, one of the main goals is to focus on gameplay.

The features that Orbit tries to improve on with a level editor and orbiting weapons corresponds to the sharing of high scores and the firing straight projectiles

Meteor Blitz had had a very confusing virtual game pad with one virtual stick controlling firing direction and the other controlling the direction of movement. Weapons switch buttons are located above and below the right stick. The issue is that new players are immediately thrown into a situation were they had to keep track of multiple things at once. In addition, the weapon switch buttons cause a disjoint feeling.

Orbit tries to improve on this by having one finger control of the main character. Orbiting weapons means the player is constantly “firing” and weapon upgrades are done with one or two finger tap gestures on the main character. This allows the player to maintain movement and weapon upgrades with little to no pause. It’s also simple to pick up for new players but still retains the multi tasking aspect.

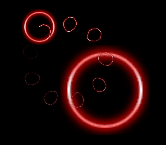
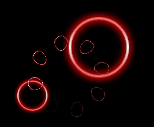


Figure 5: Two fingered tab gesture makes the player’s character larger and increases power.

**3. Algorithms and Implementations**

3.1 Overview

Technologies Used:

* iOS SDK with Xcode IDE
* Cocos2D and Box2D
* Gimp
* Zwoptex

All coding used Objective-C++ with the iOS SDK in Xcode IDE. Cocos2D was used as a wrapper around handling OpenGL and setup on iOS. It also handled much of the rendering side such as sprite maintenance and scene layering. Box2D is a physics engine but it was used primarily for collision detection. Gimp was used to create sprite assets in conjunction with Zwoptex to create sprite sheets for animations.

3.2 Game Run Loop and Entity Design

The first code refactor pass yielded the following Entity class design. Every unit whether it be enemy, the player’s character, or a wall obstacle is represented in an Entity.

**Entity**

- (void) updatePosition: (ccTime) dt

- (void) draw

b2Body

HUD

Animation

CCSprite

**PlayerEntity**

**PathedEntity**

**WeaponEntity**

**ObjectEntity**

**CircleEntity**

Subclass of (inheritance) Has a reference to.

Figure 6: Abstracted design of Entity objects.

All the Entity position updates, game state updates, and drawing to screen are handled from a main Cocos2D Layer class. This main Layer repeatedly calls the method **–(void) tick** that iterates through an array of Entities calling the **updatePosition** and **draw** methods. The main Layer is also responsible for creating the Box2D physics world, game state, and game HUD. When a new Entity is initialized from the main Layer, a reference to the physics world is passed to the new Entity so it can define it’s own kind of physics shape and criteria. Each Entity maintains it’s own **CCSprite**, variable data (such as health), **Animation,** and any **WeaponEntities**. Figure 6 shows a cycle between **WeaponEntity** and it’s parent. This cycle allows for an infinite number of sub weapons that revolve around primary or even sub weapons.

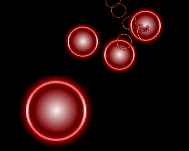
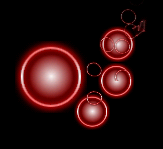


Figure 7: Examples of **WeaponEntity** sub weaponing

Weapon destruction and cleanup uses one singular recursive function that checks and destroys any particular **WeaponEntity** that is invalid and all sub weapons.

A **PlayerEntity** has a HUD parameter in its initialization method that allows a HUD object to utilize Key-Value Observing (KVO) to automatically update itself every time data values change in **PlayerEntity**.

3.3 A\* Path Finding and the Grid

Artificial Intelligence is an excellent feature to have for enemy units. It adds the challenge aspect to gameplay. The A\* Path finding algorithm is a popular implementation for games because it’s quite effective, efficient, and relatively simple to implement.

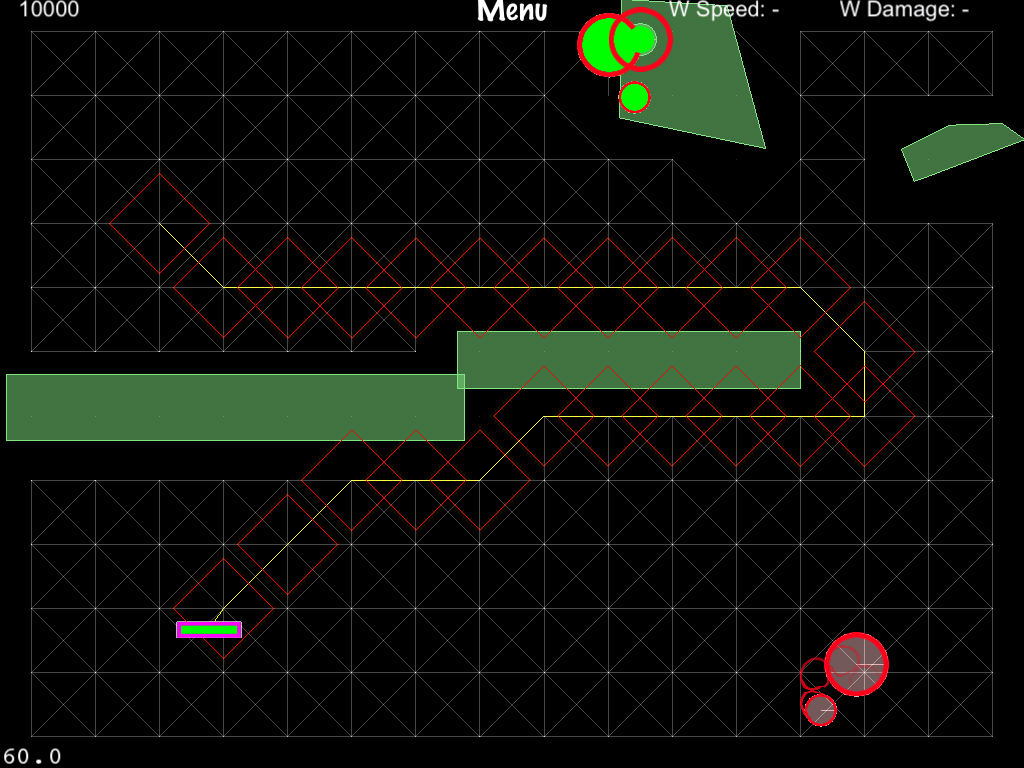


Figure 8: Shows the grid with paths culled based on objects in the physics world. An A\* path has been found for a random destination point. Debug drawing is turned on.

First, a grid is created using an array of Node objects. Each Node data structure contains the coordinates of the Node, an array of neighboring Nodes (that represent edges), and a cost value. The number of Nodes created is determined by screen size. Through trial and error, Nodes that are 64 pixel units apart yielded the best balance between performance and path granularity.

The grid is encapsulated in it’s own Cocos2D Layer called **GridLayer**. Cocos2D allows the **GridLayer** and the main Layer to be placed on top of each other. This allows for easy toggling of debug drawing and the grid is also a separate module component for swapping or replacement.

After the grid is created, game obstacles are either randomly generated or loaded from a user-generated file. The **GridLayer** then does one culling pass where it iterates through all Nodes doing 2D Ray casts against neighboring Nodes. If a Ray cast hits, its associated edge is removed. The result is a grid with Nodes and edges that don’t intersect with any obstacles.

Finally, a reference of the culled grid array is passed to all **PathedEntities** in order to update their own positions during an **updatePosition** call. A **PathedEntity** can also perform the A\* algorithm to find a path from it’s current location to a randomized new location. A found path is stored as a series of Nodes along the path. The algorithm uses a heuristic (using cost value) to perform a best first search algorithm over the Node array.

3.4 Level Editor and File Design

Designing the Level Editor comes from what was learning structuring Entity classes but it also had more emphasis on preserving MVC.

Input

LevelObject

LevelFile

LevelFileBrowerViewController

LevelEditorViewController

LevelCanvasView

LevelEditor

Has an array of. Has a reference to.

Figure 9: An abstracted design of the Level Editor in context of MVC design pattern.

**LevelEditorViewController** is the center point of the Level Editor. It handles displaying and maintaining UI states, user input, and it keeps the **LevelEditor** and **LevelCanvasView** synchronized. When **LevelEditorViewController** initializes a **LevelEditor** object, **LevelEditor** scans a predefined directory inside Orbit’s directory sandbox. It loops through directory names looking for the .olf extension and encapsulates each file path into a **LevelFile** object. Each .olf directory contains a property list that contains a series of key-values with serialized obstacle data the user defines. Obstacle data include: origin, width, height, the number of obstacles in the level, and level name. Serialization is done using NSCoding, which is a convenience framework provided in iOS.

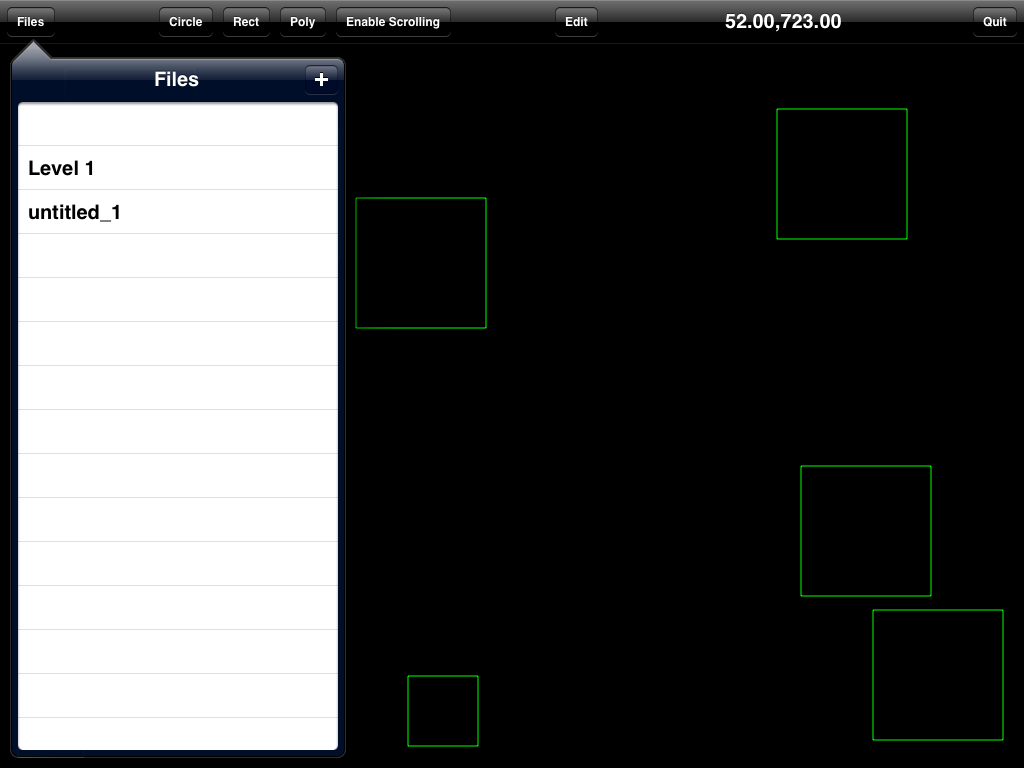


Figure 10: Current iteration of the Level Editor’s UI.

When a user creates a new obstacle, **LevelEditorViewController** handles the input and relays it to **LevelCanvasView**. **LevelCanvasView** then displays the resulting movements and scaling of objects. Manipulating objects like a rectangle is done with one-finger drag gestures. Scaling an object is handled with the two-fingered pinch gesture.

**LevelFileBrowserViewController (LFBVC)** maintains a modular table view of files. It is shown on the left side in Figure 10. **LFBVC** also handles file deletion and creation via the usual methods in the native iOS UI. For file creation, a plus button on the top right of the table view creates a modal view of a form sheet that asks for new file data. The native iOS method of file deletion from a table view is to make a swipe gesture inside of the cell containing the file name. This process of deleting meant that no cluttering confirmation pop ups were needed and makes the UI cleaner.

During an initial iteration over the editor’s UI, the choice was made to auto save everything on file switching, deletion, or quitting the editor. Auto saving takes away players worrying about saving their levels and adds to a clean UI.

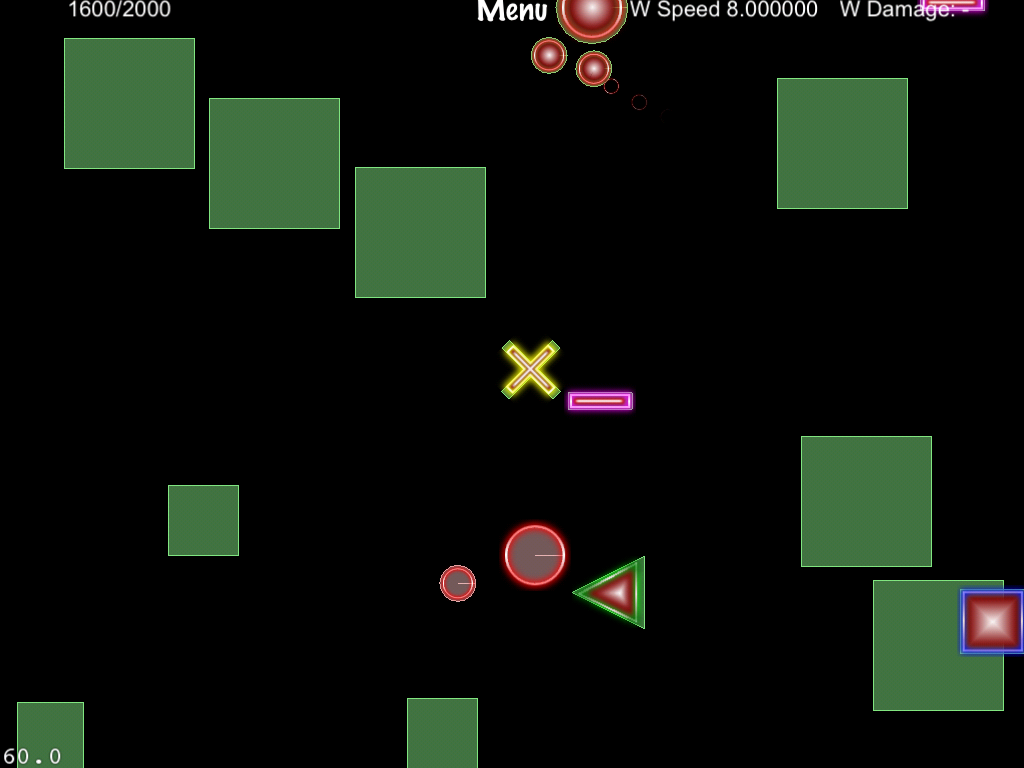


Figure 11: A level file created with the Level Editor is loaded into game. Debug drawing is turned off.

Since **LFBVC** is so modular, when a game loads level data from a file, **LFBVC** is reused in a modal view (from the main menu) for the user to select the file they want to play.

**4. Results**

4.1 What Was Done Well

The level editor was well designed around MVC. The separation of views, controllers, and models means the level editor will be highly maintainable. Components such as the level browser table view are reusable for file selection from anywhere in the app. Iterating the editor's UI to improve usability was made easier because the editor view is separate from the editor logic. In addition, if the editor UI needs an overhaul in the future, the UI can simply be swapped out with no changes needed to the editor logic or controllers.

File persistence was also implemented well. It was built in a hierarchical fashion were each layer hid implementation details of lower levels by wrapping them into convenience methods. This meant at the highest level, developers simple have to call one save method or one delete method and everything is handled.

The integration of level data from the editor and A\* path finding also worked out surprisingly well. The data structures of A\* and level files were not designed together but integration was almost smooth because of iOS serialization, data transfers via dictionaries, and key-value-observing.

Picking iOS as the platform for Orbit was also a good choice. The online developer community and resources were invaluable during implementation. The majority of minor problems (including integration of gesture recognizers with Cocos2D) had several if not more answers and discussions. Solving most minor issues mainly involved searching the right question.

The depth and intuitiveness of gameplay was also done well. New players can easily pick up the game and play knowing only that one finger is needed for controls. The movement and upgrade controls went through several iterations so movement and upgrading were worked seamlessly together. Orbiting weapons and enemies with predetermined and randomized paths made gameplay very unique and challenging. One invaluable feedback from initial play-testers was the need for enemy spawn indicators. This was an important addition because indicators give player's information about what's happening so they can react accordingly.

4.2 What Needs Improvement

Although the usability of the level editor is a good start, it needs more iteration. The editor needs some way of allowing obstacle manipulation and scrolling at the same time. Currently there is a scroll toggle to enable scrolling of the level screens and disables obstacle editing. This solved the technical issue of distinguishing if a player means to drag an object or scroll the screen, but it is still a clunky user experience.

The current weapon upgrade system is seamless with gameplay. However, players wouldn't know how to upgrade weapons if they weren't told before hand that they needed to use one or two-finger tap gestures. In addition, each progressive upgrade is not as graphically flamboyant as they are in games like Tyrian, Meteor Blitz, and Geometry Wars. Currently the player would need to upgrade their weapons for speed or power several times before noticing that their weapon moves faster or is larger.

One of the main issues with project planning was estimation on time required to implement each feature. The initial features list included core features like audio, multiplayer, and shaders. These features are present in most modern day games but their times to implement were greatly underestimated. As a result many of these core features were scoped out of development due to time constraints and disrupted task progression several times.

Orbit's artistic style is not on par with games like Geometry Wars. Many of the obstacles are simply OpenGL and Quartz 2D primitive drawings. Entities are simple sprites that have nice graphical glows, but they aren't flamboyant enough. Animations are rough and don't fit in with the glow effects of the sprites. As mentioned before, weapon upgrades are rather dull and hard to notice.

Adding simple glow effects to even place holder sprites make a huge difference in appeal. More time should've been spent on simple GIMP and other photo editing software tutorials. Finding a dedicated artist would probably be the best option so tasks are better divided and quality won't be compromised.

**5. Future Work**

Orbit is a good first step in creating a professional quality game with innovative features. It is planned for release to the App Store but will require several more months of development.

The following are additional features planned for initial release to the App Store.

List of backlogged features:

* Add additional features to the Level Editor including: undo, redo, different obstacle shapes, and predefining enemies and their travel paths. Animate in game with Bezier curves.
* Utilize shaders for better lighting once a non-beta version of Cocos2D supporting OpenGl ES and shaders is ready.
* Add audio to enhance gameplay and immersion.
* Incorporating multiplayer level file sharing and cooperative play.
* Other varieties of AI including flocking and scripted movement paths.
* Refine the weapon upgrade system to be more apparent to players.

Once Orbit is release, it’ll be exciting to see player feedback and how other game developers will build off of the ideas.

**6. Bibliography and References**

<http://itunes.apple.com/ca/app/geometry-wars-touch/id364175436?mt=8>

<http://world-tree.150m.com/Tyrian/games_tyrian.htm>

<http://meteorblitz.alleylabs.com/>

<http://infinitygene.net/iphone/index.html>

[ttp://nickcharlton.net/post/drawing-primitives-with-quartz](http://nickcharlton.net/post/drawing-primitives-with-quartz)

<http://www.raywenderlich.com/tutorials>

<http://www.raywenderlich.com/1914/how-to-save-your-app-data-with-nscoding-and-nsfilemanager>

<http://stackoverflow.com/questions/313400/nsinvocation-for-dummies>

<http://stackoverflow.com/questions/7921688/iphone-uiscrollview-detecting-the-coordinates-of-a-touch>

<https://developer.apple.com/devforums/>

<https://developer.apple.com/devcenter/ios/index.action>

<https://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Classes/nsinvocation_Class/Reference/Reference.html>

<http://www.iphonedevsdk.com/forum/>

<http://www.cocos2d-iphone.org/>

<http://www.cocos2d-iphone.org/forum/topic/8929>

<http://box2d.org/>

<http://gimp-tutorials.net/gimp-glow-effect>

<http://zwopple.com/zwoptex/>

Burba, Nathan. “Cocos2d for iPhone 1 Game Development Cookbook”, Packt Publishing, December 2011.