Kosmos, an OpenGL Android Game

Senior Project
Computer Engineering Department
California Polytechnic State University, San Luis Obispo

by
Kodie Goodwin
June 2012

Advisor
Dr. Hugh Smith

Table of Contents

Abstract **4**
Introduction **4**

Android Utilities vs. Games *4*
Libgdx, Game Engine *4*
Game Mechanics *5*
Upgrades *5*
Graphics *6*
Text *6*
Multi-touch Input *7*
Collision Detection *7*
Procedural Content Generation *7*

Background **8**

Why an Android Game? *8*
Why Kosmos? *8*

User Experience **8**

Main Menu *8*
Saved Games *9*
Upgrade Screens *9*
Pause Screen *10*
Playing a Wave *11*
Arena 11
Heads up Display 11
User Input 12
Results Screen 12
Music 13
Putting it All Together 14

Libgdx, an Open Source Game Engine **14**
Coding for Mobile Devices **14**

Garbage Collection *14*
Static Variables *15*

Model-View-Controller **16**
Lights, Camera, OpenGL **17**

Camera *17*
Perspective vs. Orthographic *19*

Accommodating Different Screen Sizes **19**

Touch Input *19*

Rendering Objects **20**

Mesh *20*
Indices with GL_LINES *21*
Buffered Vertices and Indices *22*

Base Object **23**

Circle *23*
Polygon *24*

Collision Detection **24**

Broad Phase *25*
Cell 25
Grid 27

Narrow Phase *27*
Separating Axis Theorem 28
Bounding Circles 29

Text **30**

Font Class *30*
Creating Characters *31*

Appendix **34**

Reference *34*
Table of Figures *34*
Table of Listings *34*

Abstract

People primarily use mobile games as a time waster, waiting for the bus, using the bathroom, in the car, etc. Wave based games have become very popular in the mobile gaming market because a user can play a quick wave and get immediate satisfaction. On the developer side this is great for mobile advertising; the user gets to do something fun for a short period of time and in return has to look at an ad. The question of the free app developers is “how do I keep the user coming back to look at the ads”? The answer for me was Kosmos, an OpenGL Android game.

Kosmos is a new take of a familiar style of shooting game. It pays homage to the retro style shooters like Asteroids, Galaga, and Centipede. Featuring a top-down view with simple polygon graphics, Kosmos gives the user a feeling as if you are in an arcade. Ditching the outdated goals of trying to achieve a high score with three lives given to you upon payment of a quarter, Kosmos substitutes those mechanics with a modern RPG style.

Kosmos is set in space where endless waves of enemies are attacking your ship. Each enemy killed gives the user money which must be used to upgrade your ship’s weapons and characteristics to help you progress through the game. Your ship’s arsenal includes a machine gun, rocket launcher, and laser that can be upgraded to help you defend against the onslaught of enemies. Featuring upbeat electronic music, smooth muti-touch controls, and fast paced gameplay, Kosmos is an entertaining game for the casual and even hardcore gamers.

Introduction

Android Utilities vs. Games

All android applications are written and developed using Java. It’s an object oriented language that allows developers to represent objects as data structures. The main difference between utility applications and games is that utilities like Facebook and Instagram use the Android OS API to create the entire application, whereas games primarily use Open Graphics Library. Open Graphics Library, or more commonly called OpenGL, is the “premier environment for developing portable, interactive 2D and 3D graphics applications” [1]. Common graphic rendering operations are performed using the GPU instead of the CPU, and is how developers are able to achieve high frame rate applications.

Libgdx, Game Engine

Prior to this project, I had absolutely no experience with graphics programming and OpenGL. I researched what it would take to develop a game without experience, and came to the conclusion I would need some help. OpenGL has a very steep learning curve and just rendering a square to the screen can be quite a hurdle. A game engine is a set of libraries that are developed to ease the amount of work required for developers to write games. Without a game engine I would have to design everything from the ground up. This becomes more of a challenge, because performance on a mobile device is very critical. Inefficient code means slow frame rate and jerky mechanics. Libgdx allowed me to quickly jump into developing the

mechanics of the game and not worry about the efficiency of OpenGL code and rendering objects to the screen.

Game Mechanics

Kosmos is described as a top-down, 2D shooter, role playing game (RPG). A top-down 2D shooter has the camera position above the action and graphics are rendered on a flat surface or plane. Role playing games is a category that is not as straightforward as the rest. RPGs are games where the user takes the role of a character and improves their attributes throughout experiences in the game.

In Android gaming, 2D shooters are becoming a common sight to see in the Marketplace. Kosmos distinguishes itself among the crowd by utilizing an RPG based upgrade system. Gamers tend to stay interested in a game when they are playing for something other than a high score. They want to see the effects of saving up for a specific upgrade that helps them mow down their enemies.

Upgrades

A machine gun, rocket launcher, and laser are available in Kosmos, and each weapon's damage, fire rate, and special feature can be upgraded to destroy your enemies. Special features give each weapon a unique characteristic that should be taken into consideration when developing a strategy.

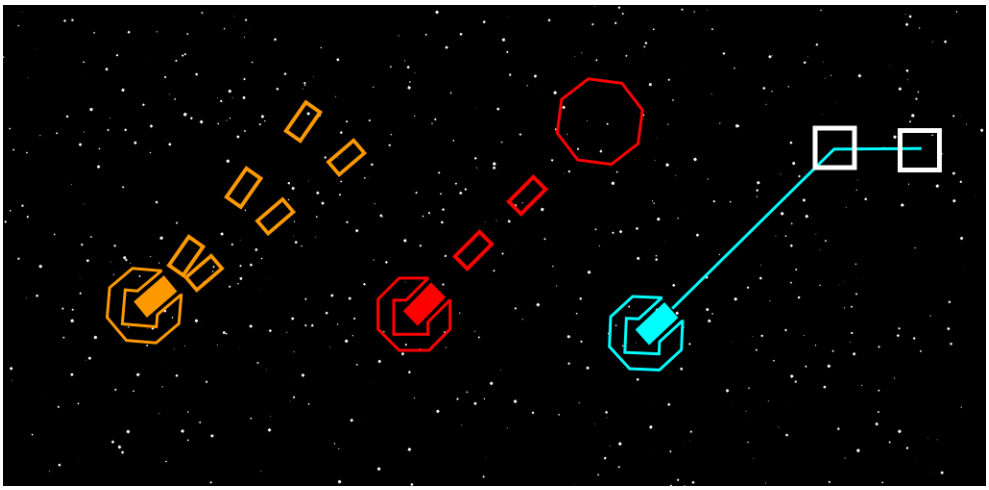

Figure 1- Machine gun, rocket launcher, and laser examples

Figure 1 shows three ships that have one of the weapons equipped and also shows the weapons special feature. The orange machine gun has a multi-shot capability that shoots multiple projectiles per round. It has low damage and high fire rate that will appeal to the "spray and pray" style of gamer. The red rocket launcher explodes upon impact making its special feature the best weapon for crowd control. It has the highest damage but the slowest fire rate which means every rocket counts when dozens of enemies are trying to attack you. In terms of fire rate and damage, the cyan laser is designed to be the median between the machine gun

and rocket launcher. The lasers special feature is a chain effect where the beam hits multiple enemies upon impact of the first one.

Graphics

Kosmos is an OpenGL game written in Java. It differs from most OpenGL games because of the way objects are rendered to the screen. Instead of importing premade images that make up objects in game, lines are drawn to create simple outlined shapes. Figure 2 shows an example of this graphical style. In Kosmos, objects are represented as common shapes, such as squares, diamonds, hexagons, or octagons.

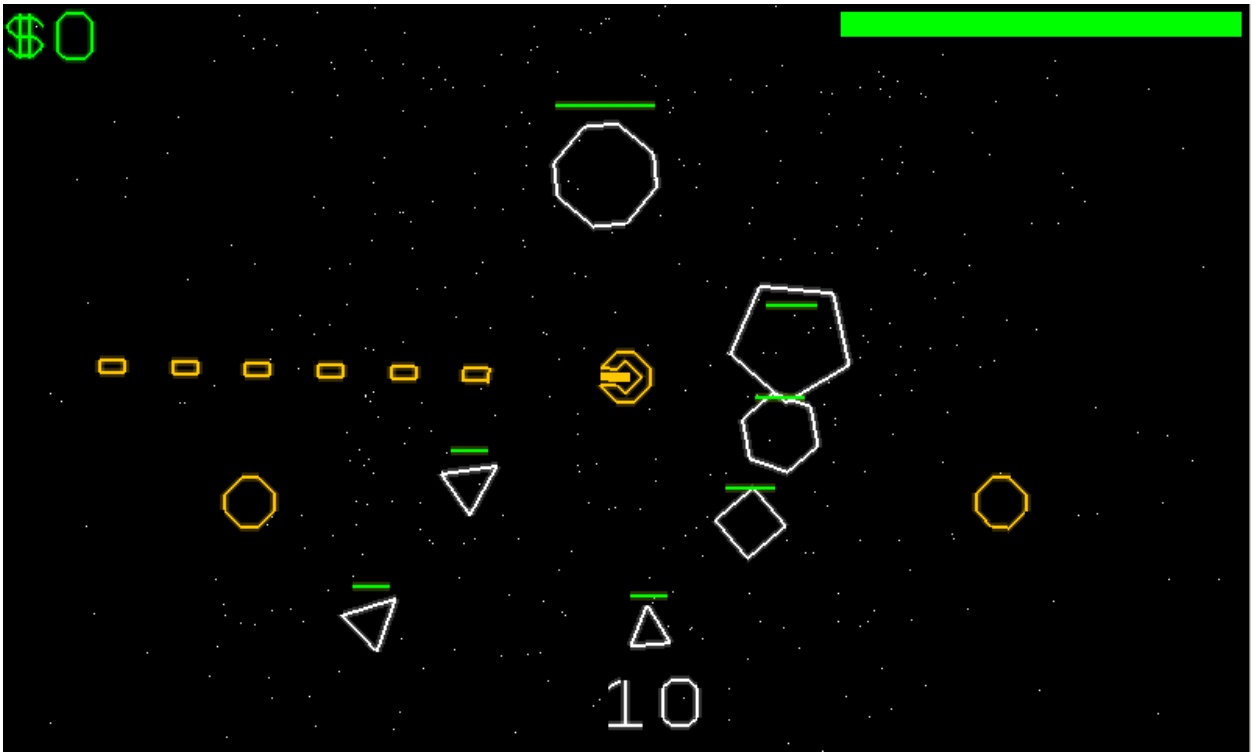

Figure 2- Example Gameplay

Rendering these shapes is fairly straightforward thanks to libgdx and a helper object called a Mesh. All that is needed is an array of vertices and an accompanying array of indices. Each vertex needs to have an X, Y, and Z component that correspond to its position in 3D space. For example, a square enemy will have an array of 12 values; three values per vertex and four vertices to create a square.

Text

Writing text to the screen is usually performed by importing an image that contains every potential character that could be used. That image is then broken up into smaller images and is then displayed to the screen as characters. I chose to implement a font system the same way I chose to render shapes, with lines and arrays holding the vertices of every character. This

added a lot of unnecessary work, but in the end gave me a consistent look and feel to Kosmos that really stands out.

Multi-touch Input

With the improvement of capacitive multi-touch screens, developers are able to create intuitive and smooth touch input systems. Kosmos uses a commonly used input style in which virtual joysticks are used to control the direction of movement and the direction of fire. The two orange conical shapes are the mentioned virtual joysticks, and Figure 3 show them in action. The left joystick controls movement and the right joystick controls the angle projectiles are being fired. In the image below, the ship would be moving directly downward and shooting to the top right.

Figure 3 - Virtual joysticks

Collision Detection

Once I found a way to render a shape to the screen, receive touch input, and move the rendered shape based upon input, the next logical step in development was to implement collision detection. I initially chose to use an advanced collision detection system that uses the separating axis theorem, a mathematical calculation that can tell if any convex polygon overlaps another convex polygon. This provided very accurate collision detection, but poor performance when a lot of objects were on the screen. I was getting roughly 58 frames per second when there were a few enemies, but in later waves gameplay slowed down to 50 or even sometimes 40 frames per second.

To solve the issue I changed to a simpler collision detection system that uses bounding circles. I create a circle around each object and check to see if the circles overlap. This circle is not rendered of course; it is only used for collision detection. Using bounding circles instead of the separating axis theorem drastically improved performance and the accuracy tradeoff was almost unnoticeable.

Procedural Content Generation

I needed a way to allocate enemies to a wave in way that wasn't repetitive and kept the user interested. Dr. Foaad Khosmood, the game design professor at Cal Poly, suggested procedural

content generation (PCG). It's a method in which content is determined by using dynamic elements like user interaction or random numbers. Games like Spore depend upon this concept to completely give a unique experience every time a user plays through the game.

Kosmos uses random numbers to generate the types of enemies set out to destroy the user. Each enemy has a weapon, movement, and size that are determined randomly by weighted probabilities. For example, the probability that an enemy is equipped with a weapon increases by one percent every successful wave. Therefore by wave 100 every enemy will have a weapon equipped. Once the enemy is equipped with a weapon, it has an even chance to be randomly assigned to have one of three weapons; goo, ice, or bomb.

Background

Why an Android Game?

The summer prior to senior project, I interned in the Android group at Intel. There I became familiar with Android development and wrote a few utility applications that test and benchmark our hardware and software against the competition. I really enjoyed developing software for Android OS and decided that I wanted to develop an Android application for my senior project. I chose to develop a game despite my complete lack of experience in mobile game development, because there is more potential to make more money in mobile games than there is in other areas. In Millennial Media's Mobile Mix it states that "gaming applications moved into the number one spot on the Top Mobile Application Categories in 2011, growing 16% year-over-year" [2]. On top of the obvious market trends, I am a long time gamer and know what generally works and what doesn't in games.

Why Kosmos?

I wanted to make a two dimensional space based shooter because it lends well to the style of graphics I wanted to use. Instead of using pre-made images and importing them into my game I wanted to use simple geometric shapes to make up everything; enemies, projectiles, menus, font, etc. The name needed to reflect that the game is set in space and also needed to reflect the simplicity of the game itself. Cosmos quickly became a great fit for the game, and because my name is Kodie with a K, I chose Kosmos to add some of my identity into the game.

User Experience

Main Menu

The menu reflects the style and simplicity of the Kosmos. The user is immediately introduced to the graphical style of bright colors and outlined shapes. Figure 4 shows the main menu and the options the user has once in the game. The music button will take you to a screen where you may turn the music on or off, and find out more about the producer of the in-game music. The tutorial button will allow a new user to play through a tutorial and become familiar with Kosmos. The play button will take you into the load game screen shown in Figure 5.

Figure 4 - Main menu screen

Figure 5 - Load game screen

Saved Games

The load game screen shows the current wave and amount of money available for that specific saved game. The user can have up to three saved games at a time. If a game slot has never been played a new game button will be displayed. If a game has been started and also saved, then the user may resume the saved game, or delete that game to start a new one. If a user does want to delete a game, a confirm screen is displayed to ensure accidents don't happen and a user deletes their game unintentionally.

Upgrade Screens

The upgrade screen, shown in Figure 6, is where the user spends money to improve their ship's characteristics and weapons. The user simply touches a box to go into another upgrade screen. For example Figure 7 shows a weapon upgrade screen where the user has selected to upgrade the laser. The weapon upgrade screen shows how much each upgrade costs and the current

value of the upgrade. You can see in Figure 7 that the user can upgrade its damage and fire rate, but cannot upgrade its chain effect. This is because the user only has 21 dollars and the cost of the chain upgrade is 500. The upgrade button displays green when the upgrade is available and white when unavailable.

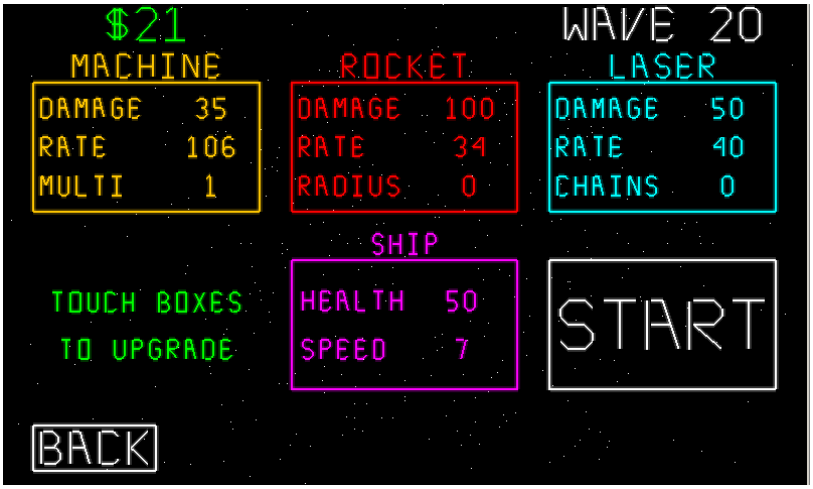

Figure 6 - Upgrade screen

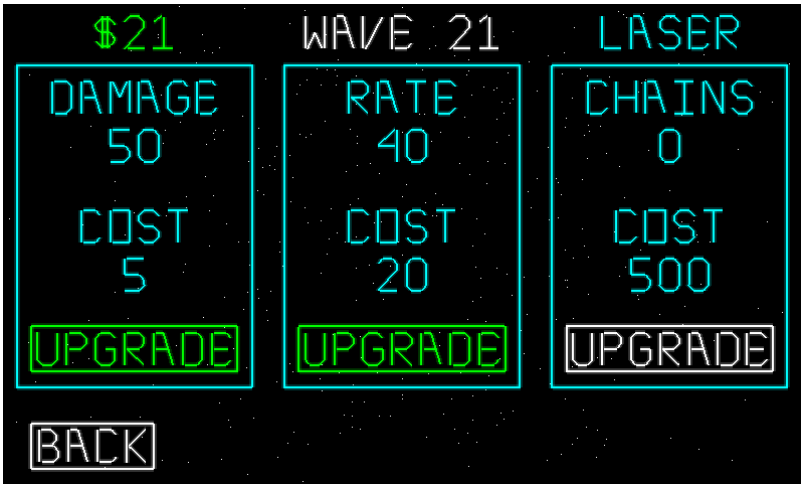

Figure 7 - Weapon upgrade screen

Pause Screen

The pause screen is the last step before getting into the action. This screen, shown in Figure 8, is where you can equip different weapons by pressing in the boxes that say “EQUIP”. Once a user equips a different weapon, the color of the words “PRESS AND HOLD HERE TO PLAY” and the equip box changes to the corresponding weapon color. In order to start playing the user must hold their finger in an imaginary box that surrounds that text. In order to stay in the game the user must have at least one finger touching the screen at all times, hence the text “PRESS AND HOLD HERE TO PLAY”. If the user does lift up on both fingers, then the game will smoothly transition back up to the pause screen. There the user may equip another weapon and with one click and be back in the action. This implementation turned out to be very useful for

Kosmos because sometimes a user may want to switch between weapons in certain situations. For example if several enemies are grouped together, one perfectly placed rocket can take them all out because of the splash damage.

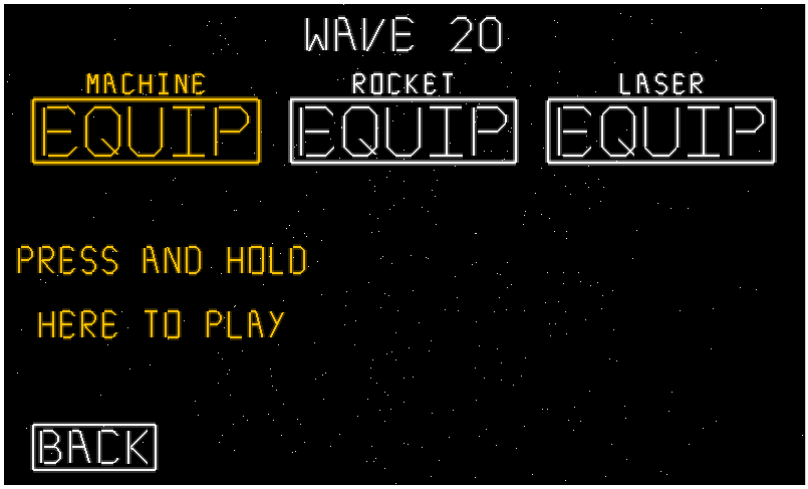

Figure 8 - Pause screen

Playing a Wave

Arena

When the user enters a wave he or she is surrounded by a large octagon that acts as an arena. The user's ship can touch the arena without consequences, but cannot travel outside of the octagon. When projectiles reach the boundary because they did not hit an enemy, the projectile is then removed from the screen. Enemies enter the arena in a random location, but will always be inside the arena. Enemies enter in from the very depths of space at a location of -99 or right at the far clipping plane. This effect gives the user time to see where an enemy is coming in and react accordingly.

Heads up Display

Figure 9 shows the heads up display in Kosmos. The heads up display, or HUD, contains important information the user needs to see at all times, such as health, money, and how many enemies are left on the screen. The money gathered for the current wave is displayed in the top left corner of the screen. The health bar is located in the top right corner and displays the current health of your ship. A bar that is half red and half green indicates that your health is half full. The amount of enemies left for the current wave is displayed by the bottom center of the screen. This gives the user an indication of how long the rest of the wave will be. The two joysticks are located in the bottom corners of the screen.

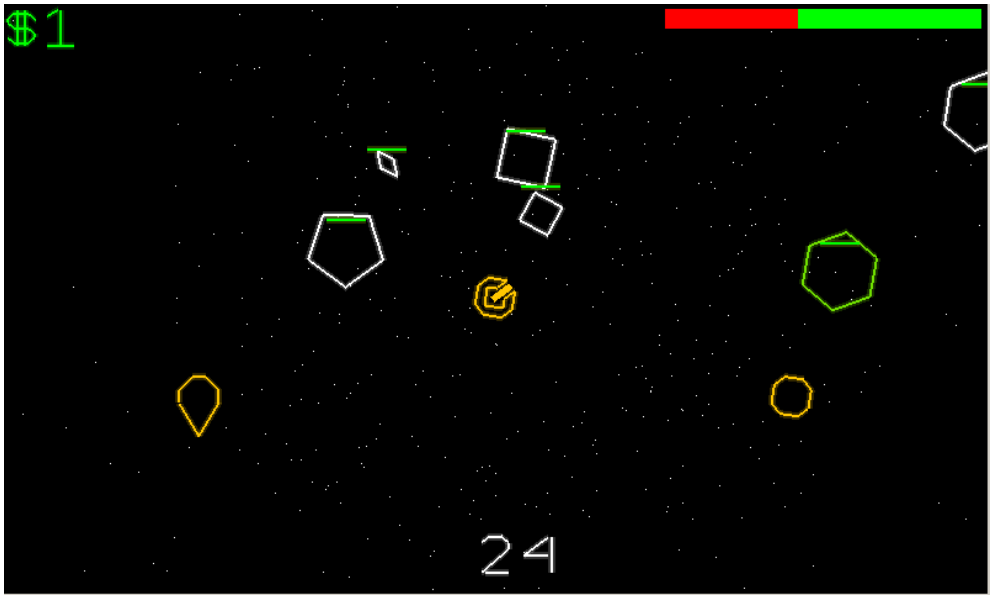

Figure 9 - Gameplay

User Input

The ship in Kosmos has a velocity vector that controls the movement. Every frame the ship moves based upon how much time has elapsed since the last frame and the current velocity of the ship. Velocity is represented by a 2 dimensional vector that has an X and Y component to it. The left joystick is the reference point for an angle between the center of the joystick and the user's finger. That angle directly controls the velocity vector and thus controls the ships movement. If the ship has a speed of 10 and the angle from the joystick to the touch location is 45 degrees, then the velocity of the X and Y component is going to be 7.07. The equation below shows the velocity calculations.

$$Velocity_X = Ship\ Speed * COS(angle)$$
$$Velocity_Y = Ship\ Speed * SIN(angle)$$

The right joystick controls the angle of which the ship fires projectiles. Again, the angle is determined by using the users touch location, and the center of the joystick. While the user has his or her finger down, the ship will fire projectiles as long as the delay between projectiles has elapsed. As stated in the Pause Screen section, the user must hold down at least one finger to remain in game.

Results Screen

The results screen is shown upon completion of a wave regardless of winning or losing. To complete a wave successfully, the user must kill every enemy on that wave and losing a wave will occur when the ships health reaches zero.

The results screen shown in Figure 10 displays some interesting statistics. It displays the damage given to enemies, damage taken from enemies, and how many enemies were killed. It also shows how much money was gathered this wave and how much money the user has total

after the previously completed wave. It also shows statistics about the amount of shots fired, hit, and missed. The accuracy is displayed to the user as a convenient way to quickly see how efficient the user was with its weapon. The results screen is also of importance, because on mobile devices, an ad is displayed in the bottom right section. Screenshots were taken on the desktop version and thus does not show the ads. The results screen takes the user back to the upgrade screen where they can then upgrade their ship and weapons to better their defense against the waves of enemies.

Figure 10- Results screen

Music

Music for video games is more important than most people understand. Music needs to enhance a user's interaction with the game while not being intrusive toward the experience. I wanted music to reflect the emotion of Kosmos and improve the overall experience while playing it. I approached a music producer named Bodusdank, who primarily makes electronic dance music, if he would be interested in producing music for Kosmos. I didn't offer him any money but I did promise him a music section where users can click on his web pages. Being a gamer himself, he agreed to my proposal. Figure 11 shows the music screen that promotes Bogusdank.

Figure 11 - Music screen

Putting it All Together

Combined with smooth multi-touch controls, simple but interesting graphics, upbeat music, and an addicting upgrade system, Kosmos will give an experience that will entertain the casual gamer to the hardcore gamers, and everyone in between. On top of all of those reasons mentioned, Kosmos is free, and who can complain about free software?

Libgdx, an Open Source Game Engine

I chose libgdx as a game engine primarily for the cross-platform capabilities of developing on the desktop and then writing an extra 6 lines of code and running the same game on an Android devices. This sped up development significantly because of the inline debugging and hot-swapping abilities of the eclipse IDE for java. Libgdx and hot-swapping became very important when designing the user interface and menus. I was able to run Kosmos on the desktop and then change position values of the GUI and immediately be able to see the changes in the application. Libgdx was a huge component of Kosmos that allowed me to complete a polished game in less than 6 months.

Coding for Mobile Devices

Developing desktop games is different than developing games on mobile phones. Sure they both run essentially the same code, but the hardware architecture is very different. Desktop applications are focused more on coding convenience, and not strictly performance. When I first complied and ran Kosmos on a phone, I got a max of only four frames per second. I quickly realized that I needed to completely rethink how I was developing Kosmos.

Garbage Collection

The java virtual machine has a neat feature called garbage collection that manages unused memory. It is great for developing utilities and other application, but for games, garbage collection slows frame rates because the processor cleans up lost data instead of rendering the screen. When I first tested Kosmos I was creating new arrays that held the vertex data every

update to the screen. After I set the vertices to a Mesh, the array was discarded. Performing this process several times would evoke garbage collection thus slowing the game down to an embarrassing speed.

Static Variables

To get around garbage collection data allocation must be kept in the back of your mind at all times while developing games. Static variables should be used as often as possible because they alleviate the issue of leaking memory. Java objects are created with the “new” keyword and when an object is created, the developer must ensure that that object is meaningful and will not be wasted.

Listing 1 shows an example of bad code that leaks memory, and listing 2 shows an example of how the developer would get around the memory leak. Do not look too much into the details of the code, but more of how memory is allocated during the steps of the function. At each step a new Vector2 object is created and when the function returns, that memory is leaked and eventually collected by the Java virtual machine. These types of function are very bad for performance, especially because this function is called every time a collision is checked using the separating axis theorem.

```
private static Vector2[] getAxis(Vector2[] verts){

    // Creates the array of axis
    Vector2 [] axis = new Vector2[verts.length];

    for(int i = 0; i < axis.length; i++ ){

        // get the current vertex
        Vector2 p1 = new Vector2( verts[i] );
        // get the next vertex
        Vector2 p2 = new Vector2( verts[i + 1 == verts.length ? 0 : i + 1] );
        // subtract the two to get the edge vector
        Vector2 edge = p2.sub(p1);
        // get either perpendicular vector
        // this method is just (x, y) => (-y, x) or (y, -x)
        Vector2 perp = new Vector2(-edge.y, edge.x);
        // Sets the axis to the array
        axis[i] = perp;
    }
    return axis;
}
```

Listing 1 – Example of leaking memory

Listing 2 shows how I optimized the getAxes function by using static variables to manipulate and return the array of Vector2 objects. This method of allocating the memory at runtime is great because memory is never leaked, therefore garbage collection isn’t evoked.

```
private static Vector2 p1 = new Vector2();
private static Vector2 p2 = new Vector2();
private static Vector2 perp = new Vector2();
private static Vector2[] axis = new Vector2[]{
    new Vector2(), new Vector2(), new Vector2(), new Vector2(), new Vector2(),
    new Vector2(), new Vector2(), new Vector2(), new Vector2(), new Vector2()};

private static Vector2[] getAxis(Vector2[] verts){

    for(int i = 0; i < axis.length; i++ ){

        // get the current vertex
        p1.set(verts[i]);
        // get the next vertex
        p2.set(verts[i + 1 == verts.length ? 0 : i + 1]);
        // subtract the two to get the edge vector
        Vector2 edge = p2.sub(p1);
        // get either perpendicular vector
        // this method is just (x, y) => (-y, x) or (y, -x)
        perp.set(-edge.y, edge.x);
        // Sets the axis to the array
        axis[i] = perp;
    }
    return axis;
}
```

Listing 2 - Example of not leaking memory by using static variables

Model-View-Controller

In order to maintain the project in an organized fashion, I used the Model-View-Controller (MVC) programming technique throughout the entire project. MVC is a way to isolate certain parts of the program which helps increase the efficiency of development and debugging. The model component contains all of the data objects and structures, the View contains all of the rendering logic, and the Controller contains all of the simulation code that manipulates data and the flow of the program.

Kosmos is organized in the MVC style by grouping all of the Java files into four main packages, screens, objects, renderer, and simulation. The screen file is the top level file and is the most important because it handles user input, runs the simulation code, and renders the objects each frame. The object files contain all of the data that is relevant for the logic state. The object files know nothing about how to render their data. Rendering these objects is the job of the render files, and when given an object they know how to render it. The simulation files handle the flow between logic states and force the objects to update certain values during logic state transitions.

Figure 12 shows the simplest state in Kosmos, the Confirm state. The data required for this screen is the string that says “CONFIRM?” and the two Selection boxes. The Screen file, which handles the input, is checking to see if the user selects either of the two boxes. If yes, the

Screen file erases the saved game and then returns to the previous state by setting a global variable that holds the current state of the game. When the current state variable changes from CONFIRM to PLAY, the simulation file then takes over and handles the screen transition. Using the MVC design methodology, I was able to quickly add new features and states, and was able to efficiently debug any issues that arose during development.

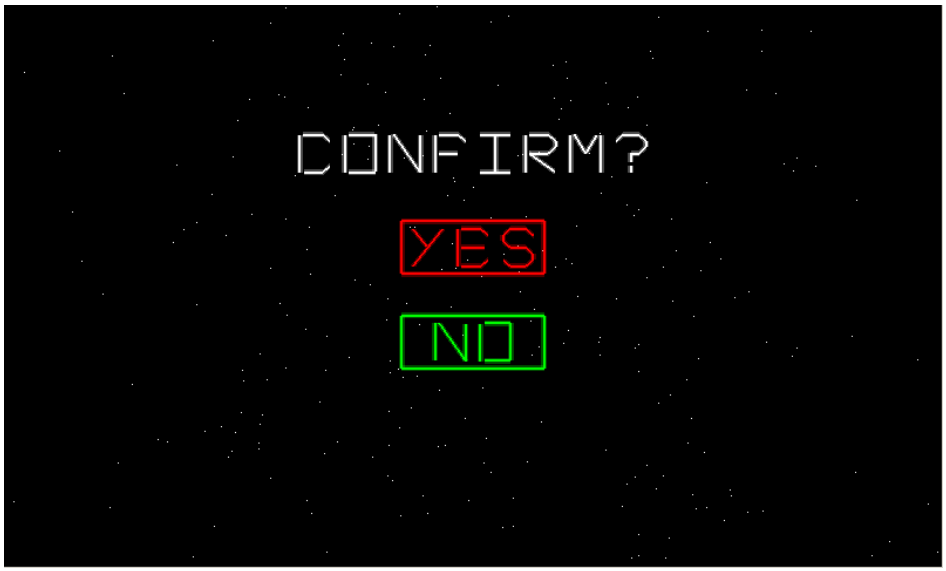

Figure 12 - The Confirm State

Lights, Camera, OpenGL

OpenGL is the standard for graphics programming on mobile devices. OpenGL versions 1.0, 1.1, and 2.0 are supported on Android devices with OS version 1.5 or higher. I chose to use OpenGL version 1.0 because I didn't have a need for shaders due the simplicity of how I render objects to the screen. Before I dive deeper into the details of Kosmos, I need to first describe a few component of OpenGL.

Camera

"A camera is defined by a position in 3D space, a direction given as a unit length vector and it's "up" vector, again given as a unit length vector" [3]. Besides the position and orientation, the view frustum is the third piece to the puzzle. Figure 13 shows a pyramid with the top cut off and an eye representing where a user would be looking. This pyramid is the view frustum, and anything inside this pyramid can be seen, anything outside of the clipping lines is just that, clipped out of view.

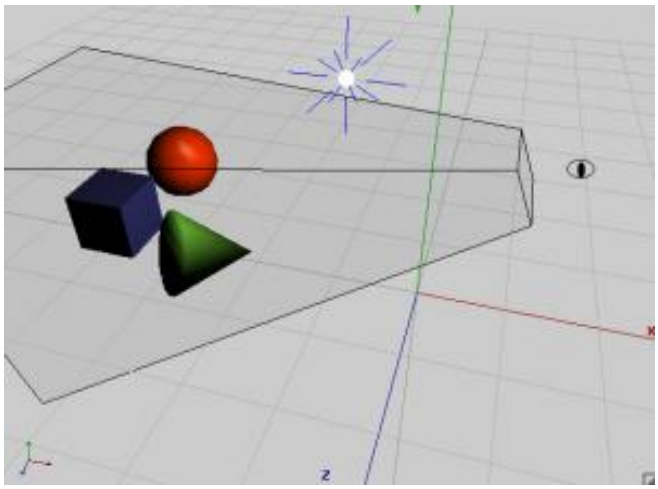

Figure 13 - Frustum example

The frustum is delimited by 6 so called clipping planes: near, far, left, right, top and bottom. In Figure 13 above those planes are the sides of the pyramid. You can think of a camera in OpenGL as a movie producer looking through his hands in “L” shapes to make a rectangle to better put the view into perspective. The near clipping planes shown in Figure 14 have a special role for a developer because that is where all objects seen in 3D are transformed onto this 2D plane through a process called projection.

There are two types of projection used in OpenGL, perspective and orthographic. Orthographic cameras are primarily used in 2D video games because at any depth away from the near clipping plane, the object is still seen as the same size. On the other hand, perspective cameras are used in 3D games because objects appear smaller as they move farther away. The only real difference between perspective and orthographic cameras is the shape of the frustum. Figure 3 shows this difference and how the objects are perceived to the eye.

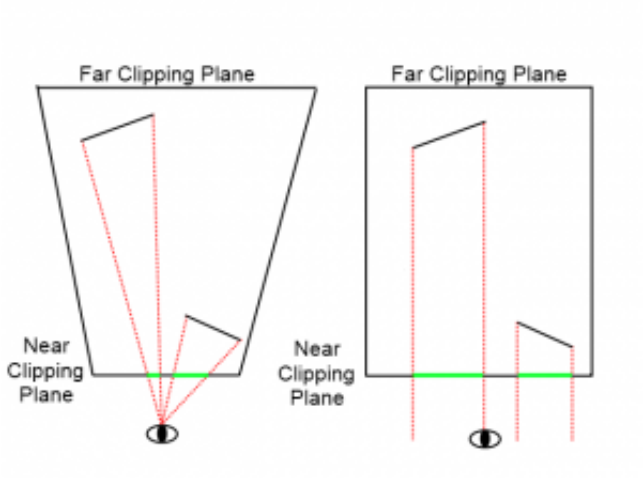

Figure 14 - Top view of frustum

Perspective vs. Orthographic

I chose to use a perspective camera for Kosmos because I use the third dimension to appeal to people's natural perception of depth. The most obvious feature that utilizes depth is the way I transition between screens or logic states. The images below show a transition from the main menu to the load game screen. When the user presses play, the load game screen appears at twice the viewing distance. The simulation code then takes over and updates the plane depths until the load game screen is at the proper viewing distance.

Figure 15 - Main Menu to Save Game Screen Transition

Accommodating Different Screen Sizes

Touch Input

Designing a game for Android is arguably harder than designing a game for Apple's IOS. The reason for this difficulty is due to the varying configurations of hardware a game will be played on. The hardest part about this is usually not the processing power, but the different screen sizes and resolutions. Today's phones are very powerful and can handle most simple games if proper optimization techniques are applied, but screen sizes are something that everyone must account for.

A problem with different screen sizes arose when testing Kosmos on a friend's phone. The main menu screen loaded fine, but the touch input wouldn't register correctly. The issue was because his screen size was smaller than mine and the touch input was being perceived in another location. To resolve this I needed to un-project the 2D touch location to my in-game 3D location. Lucky for me, libgdx already contains all of the necessary functions and classes required for the un-projection.

Libgdx performs un-projection by using the helper classes, Ray and Plane. A Ray is created using the camera, X and Y coordinates of the touch inputs, and the direction vector of the camera. Libgdx has a default camera position at (0, 0, 0) with the camera facing the negative Z direction. In other words, as object move farther away from the near clipping plane, the z component of its position decreases. The game plane is at a depth of -11.3, so we need to figure out it the touch location on the plane at a distance of -11.3.

The function getTouchPoint in listing 3 shows how it easy it is to perform the required un-projection to get in game touch coordinates. We first get the pickRay from the camera, and then use a function in libgdx that performs the required math operations. It sets the 3D location into the Vector3 variable touchPoint and is returned. I can then use the touch location just as if the user is directly touching locations at the game plane depth of -11.3.

```
public static Vector3 touchPoint = new Vector3();
public  Vector3 getTouchPoint(){
       pickRay = Kosmos.pCam.getPickRay(input.getX(0), input.getY(0));
       Intersector.intersectRayPlane(pickRay, Kosmos.gamePlane, touchPoint);
       return touchPoint;
}
```

Listing 3 - 2D to 3D un-projection

Rendering Objects

Mesh

In Mario's book "Beginning Android Games" he said that OpenGL is "a lean mean triangle rendering machine" [4]. It uses a collection of vertices and accompanying indices to make up geometric shapes. I am primarily using lines in Kosmos and will be what I use to explain how to render objects to the screen. Figure 16 below shows a simple 10x10 square that is defined by 4 points in 3D space. Each point contains 7 floats; three floats for the x, y, and z components, and four floats for the RGBA components that make up the color.

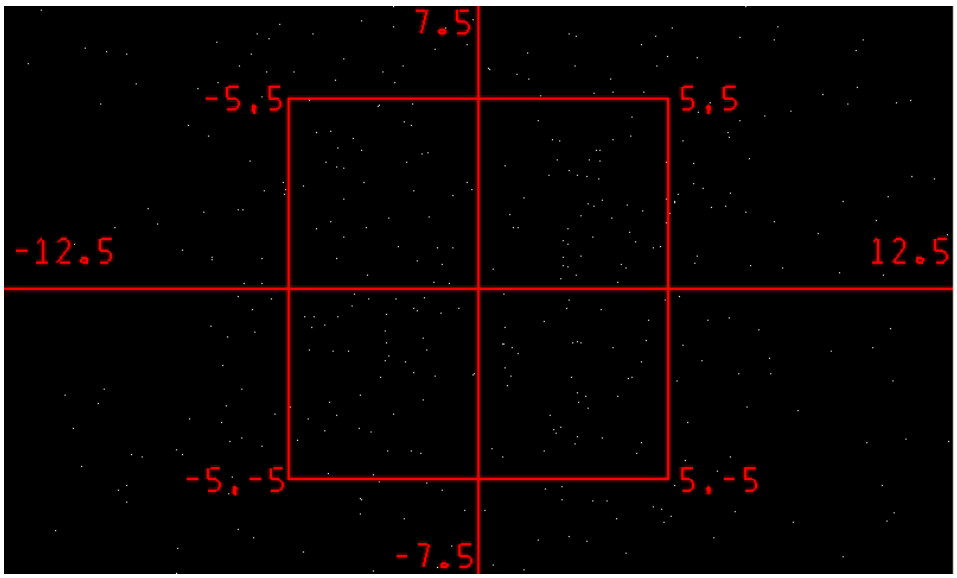

Figure 16 - Simple 10x10 square

The center of the square is at location (0,0) and the four corners are 5 units away in the X and Y directions. The array of vertices that define this square is shown in the table below.

X	Y	Z	R	G	B	A
-5	5	-11.3	1	0	0	1
5	5	-11.3	1	0	0	1
5	-5	-11.3	1	0	0	1
-5	-5	-11.3	1	0	0	1

Once the vertex array is ready you may set the vertices, and optionally indices, to a libgdx object called a mesh. A Mesh is a convenience data structure that allows developer using libgdx to focus more on the game mechanics and less about how to render objects to the screen efficiently. Allocating the vertex array is the last step before rendering objects to the screen. Listing 4 shows the three steps required to render a collection of points. The first step is to set the vertex array, the second step is to set the index array, and the third is to render the mesh with the appropriate OpenGL primitive.

```
mesh.setVertices(vertices, 0, vIndex);
mesh.setIndices(indices, 0, iIndex);
mesh.render(GL10.GL_LINES, 0, iIndex);
```

Listing 4 - Rendering shapes with a Mesh

Indices with GL_LINES

Previously I stated that indices are optional, but when using the OpenGL primitive GL_LINES, indices are required to ensure proper functionality. OpenGL works by specifying two indices that make up a line. Looking back up to Figure 16, there are four lines that make up the square. The top line is made up of the first and second vertex location. The indexes are zero based though,

so they represented as 0 and 1. The right line is made up of the second and third vertex position and is represented as 1 and 2. Listing 5 below shows the required index array to render the square in Figure 16.

```
float [] squareIndices = new float[]{ 0, 1, // Top line
                                      1, 2, // Right line
                                      2, 3, // Bottom line
                                      3, 0  // Left line
                                      };
```

Listing 5 - Index array for a square

Buffered Vertices and Indices

Using the Dalvik Debug Monitor Server (DDMS), I was able to track down a bug due to the large amount of processing time during render functions of Mesh objects. I created a Mesh for every type of object and would set the vertices of the object every frame. I later found that when setting vertices to a Mesh a time consuming function named bind() is called. The combination of the Mesh objects calling bind() every frame was noticeably slowing down the frame rate.

To solve this issue I developed a buffering technique that allowed me to only set the vertices once every frame. To render a shape to the screen I add the vertices of the object to a buffer and keep track of the current vertex position. Listing 5 shows a static function called addVerts in a helper class named RenderUtil. This function adds the vertex data to the large buffer named vertices and keeps track of the index. This index value also functions as the count of the vertices when a call to render occurs.

```
public static void addVerts(float[] verts, int count){
    for(int i = 0; i < count; i++){
        vertices[vIndex] = verts[i];
        vIndex++;
    }
}
```

Listing 6 – RenderUtil's addVerts function

Because we are rendering every frame using this buffer, we need to keep track of the indices in a special way. We cannot just keep the count of the indices, because of the nature of the data. As described in the previous section, indices are dependent upon the vertices and how they are entered into the vertex array. We need to keep track of the next index and do so by calculating the maximum index value. Listing 7 shows the function addInds that adds the indices and also keeps track of the next index.

```
public static void addInds(short[] inds, int count){
	short max = 0;
	for(int i = 0; i < count; i++){
		indices[iIndex++] = (short) (inds[i]+nextInd);
		if( (inds[i]+nextInd)  > max){
			max = (short) (inds[i]+nextInd);
		}
	}
	nextInd = max+1;
}
```

Listing 7 - addInds function

Using the RenderUtil and buffering every object rendered that frame drastically improved the frame rate of Kosmos. The static buffers used for vertices and indices take up a lot of RAM, but is acceptable considering the improvements in rendering speed.

Base Object

Every moveable object in Kosmos needs to extend a BaseObject. Extending a class in Java is a way to conveniently inherit a class's data and functions. Listing 8 shows the public data that enables objects to be rendered and manipulated. The position vector is a 3D point in space that is the center location for an object. Velocity is also a 3D vector, but can be thought of as the current speed of the object. BaseObjects can be rotated with helper functions, and the angle value represents the current angle of the object. The other two classes, Circle and Polygon, are not as straightforward as the others.

```
// 3D point in space representing the objects position
public Vector3 position;
// 3D Vector that represents the objects velocity/speed
public Vector3 velocity;
// The current angle of the object
public float angle;
// Used for collision detection
public Circle circle;
// Contains the vertices required to render a shape
public final Polygon shape;
```

Listing 8 - BaseObject data

Circle

A circle is only used for collision detection and is never rendered. The circle should be created so that it closely fits the size of the object to make for the most accurate collisions. A circle only has an X and Y position, a radius value, and helper functions that check to see if a point lies within itself.

Polygon

A Polygon is a very important class because it holds the array of vertices used to render the object. To render an Enemy, which extends BaseObject, all that is required is to set the Mesh vertices by accessing the shape Polygon of the BaseObject. Polygons also contain functions that translate and rotate the objects vertices. Translation is a simple process of adding the distance you wish to move to the current vertices, but rotation is much more difficult.

Listing 9 displays the logic behind the setRotation function. An array of vertices is created upon initialization to be a reference shape at a rotation of zero degrees located at the origin. This reference array allows me to calculate the rotated position of each vertex centered at the origin. Once I have the rotated vertices, I add the current position to each vertex, and I am left with a rotated shape at its current position.

```
public void setRotation (float newAngle) {
	// Get the cosine value of the new angle
	float cos = Space.COS(newAngle);
	// Get the sine value of the new angle
	float sin = Space.SIN(newAngle);
	// Set the current angle to the new angle
	this.angle = newAngle;
	// Increments of 7 because there are 7 floats per vertex.
	// 3 for position, and 4 for RGBA values
	for(int i = 0; i < verts.length; i+=7){
		float x = refVerts[i];
		float y = refVerts[i+1];
		// X component = Rotated X point of reference + current X position
		verts[i]   = ((cos * x) - (sin * y)) + this.x;
		// Y component = Rotated Y point of reference + current Y position
		verts[i+1] = ((sin * x) + (cos * y)) + this.y;
	}
}
```

Listing 9 - Sets the rotation of a Polygon

Collision Detection

When people ask me what was the hardest part of making Kosmos, I answer “collision detection”. Almost every game needs to have some way to be able to determine if two objects are overlapping. Collisions can happen very frequently, therefore need to be efficient algorithms. On top of efficiency, collision detection needs to be accurate. If a user fires a projectile and travels right through an enemy, they will feel cheated and will not want play your game. There is a sweet spot that every developer must find that is a good balance between efficiency and accuracy. I was able to find this sweet spot by using a few online resources, and a lot of trial and error.

Without optimization, collision detection is a $O(n^2)$ operation because every object must be checked for collision against every other object on screen. In Kosmos this is not acceptable because there could be more than 100 objects on screen at a time. To improve the detection

algorithm I implemented a two-step process that is referred to as broad phase and narrow phase.

Broad Phase

As stated before collision detection is naturally a $O(n^2)$ algorithm. The goal of the broad phase is to improve that algorithm to a O(n*log(n)) operation. To achieve this I implemented a relatively simple spatial hashing technique using the help of an awesome blog post by Metanet Software Inc [5].

Spatial hashing is a grid based algorithm that reduces the number of collision checks by only testing for collisions if there are two or more objects that can collide. Each cell contains a collection of objects and therefore knows if a cell could have a collision. Figure 17 shows the grid lines which normally are not rendered during gameplay.

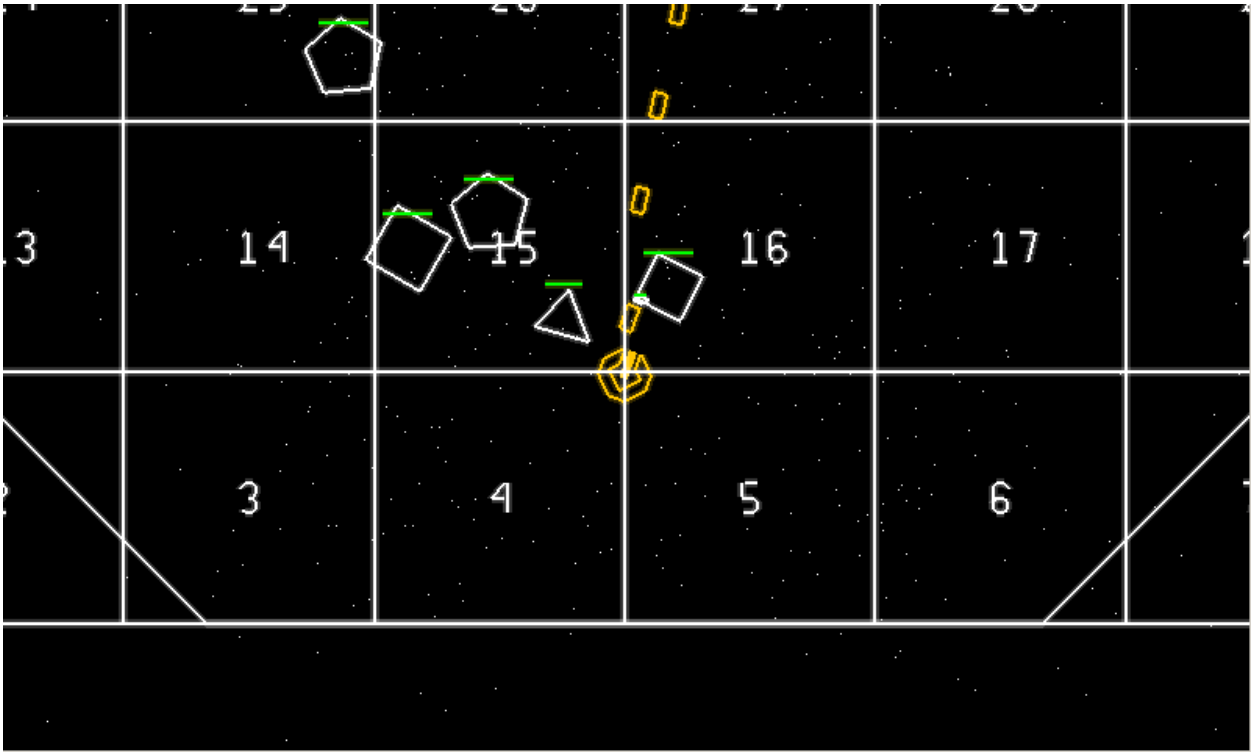

Figure 17 - Grid based collision detection

If and object is located in multiple Cells like the SquareEnemy on the left in Figure 17, that object is allocated to both Cells. So in the situation pictured above, the Ship is allocated to Cell 4, 5, 15, and 16. This will ensure that the Ship can be hit by enemies the Ship when it is located in multiple Cells. Figure 17 displays the power of spatial hashing because even though there is a lot going on, only Cells 15 and 16 will be checked for a collision.

Cell

There are five types of objects that can have collisions in Kosmos; Ship, Projectile, RocketExplosion, Enemy, and EnemyExplosion. At any given time there could be any number of objects allocated to that cell, so it must have a container for each of those objects. Listing 10

shows the declarations of those containers in the Cell class. You might notice that ship is not an array like the rest, because there can only be up to one ship at a time.

```
public class Cell {
	public Ship ship;
	public ArrayList<Enemy> enemies;
	public ArrayList<Projectile>		projectiles;
	public ArrayList<EnemyExplosion>	enemyExplosions;
	public ArrayList<RocketExplosion>	rocketExplosions;
	public Rectangle bounds;
}
```

Listing 10 - Data declaration of Cell class

A Cells job is to keep track of objects on the grid and conveniently tell me if a collision should occur. Every frame the grid is cleared and objects are then allocated to it. This ensures that at any moment during simulation, I can iterate through the grid and ask for a specific type of collision. Listing 11 shows the types of collisions that can occur and the conditions which must be met in order to check for one.

```
// Ship and Enemy
public boolean shouldCheckShipAndEnemy(){
	return (enemies.size() >= 1 && ship != null);
}

// Ship and EnemyExplosion
public boolean shouldCheckShipAndEnemyExplosion(){
	return (enemyExplosions.size() >= 1 && ship != null);
}

// Enemy and EnemyExplosion
public boolean shouldCheckEnemiesAndEnemyExplosion(){
	return  (enemyExplosions.size() >= 1 && ship != null);
}

// Enemy and Projectile
public boolean shouldCheckEnemiesAndProjectiles(){
	return  (enemies.size() >= 1 && projectiles.size() >= 1);
}

// Enemy and RocketExplosion
public boolean shouldCheckEnemiesAndRocketExplosion(){
	return  (rocketExplosions.size() >= 1 && enemies.size() >= 1);
}

public boolean shouldCheckEnemiesAndEnemyExplosion(){
	return  (enemyExplosions.size() >= 1 && ship != null);
}
```

Listing 11 - Collision condition function

Grid

A class named CollisionUtil is used to hold the Cells and provide some helper function that clear and allocate the Cells. CollisionUtil is a class that gets instantiated upon startup of the application. The width and height are passed in to create the 2D array of Cells which is appropriately named grid. Every frame the grid is cleared and allocated with the functions clearGrid and allocateGrid. Listing 12 and 13 show the implementation of the clearGird and allocateGrid functions respectively.

```
public void clearGrid(){
	// Iterate through the grid
	for (int rowIndex = 0; rowIndex < rows; rowIndex++){
		for(int colIndex = 0; colIndex < cols; colIndex++){
			// Clears the cell contents
			grid[rowIndex][colIndex].clearCell();
		}
	}
}
```

Listing 12 - clearGrid function

```
public void allocateGrid(ArrayList<Enemy> enemies,
                  Ship ship,
                  ArrayList<Projectile> projectiles,
                  ArrayList<RocketExplosion> rocketExplosions,
                  ArrayList<EnemyExplosion> enemyExplosions){
   // Iterate through the columns and rows
   for (int rowIndex = 0; rowIndex < rows; rowIndex++){
      for(int colIndex = 0; colIndex < cols; colIndex++){
         // Allocate enemies
         for(int i = 0; i < enemies.size(); i++){
            Enemy enemy = enemies.get(i);
            // Only check enemies if they are OK. Not DEAD or NEW
            if(enemy.state == EnemyState.OK){
               if(grid[rowIndex][colIndex].contains(enemy.shape)){
                  grid[rowIndex][colIndex].enemies.add(enemy);
               }
            }
         }
       . . .
       }
   }
}
```

Listing 13 - allocateGrid function

Narrow Phase

The narrow phase is the next step in determining whether two objects overlap or collide. After reading the post written by William on the CodeZealot website [6], I decided to use his example and implement the separating axis theorem. William states in his post that:

The Separating Axis Theorem, SAT for short, is a method to determine if two convex shapes are intersecting. The algorithm can also be used to find the minimum penetration vector which is useful for physics simulation and a number of other applications. SAT is a fast generic algorithm that can remove the need to have collision detection code for each shape type pair thereby reducing code and maintenance.

Because the separating axis theorem can detect collisions between any two convex polygons and because I am only using convex polygons for Kosmos it was an easy decision to choose to implement SAT as my collision detection algorithm.

Separating Axis Theorem

The separating axis theorem is great for collision detection in terms of accuracy. The downside of this algorithm is that it must perform projection for every axis that makes up the two shapes being tested. Listing 14 shows pseudo-code taken from William's post and displays how much processing is required for every collision test.

Collision detection performance was never an issue until Kosmos was almost complete. I was testing the gameplay with dozens of enemies on screen and shooting roughly 100 projectiles per second. The frames per second dropped in half and looking at the function profiler in the Eclipse IDE, collision detection was the culprit. This forced me to implement a different collision detection algorithm.

```
Axis[] axes1 = shape1.getAxes();
Axis[] axes2 = shape2.getAxes();
// loop over the axes1
for (int i = 0; i < axes1.length; i++) {
  Axis axis = axes1[i];
  // project both shapes onto the axis
  Projection p1 = shape1.project(axis);
  Projection p2 = shape2.project(axis);
  // do the projections overlap?
  if (!p1.overlap(p2)) {
    // then we can guarantee that the shapes do not overlap
    return false;
  }
}
// loop over the axes2
for (int i = 0; i < axes2.length; i++) {
  Axis axis = axes2[i];
  // project both shapes onto the axis
  Projection p1 = shape1.project(axis);
  Projection p2 = shape2.project(axis);
  // do the projections overlap?
  if (!p1.overlap(p2)) {
    // then we can guarantee that the shapes do not overlap
    return false;
  }
}
// if we get here then we know that every axis had overlap on it
// so we can guarantee an intersection
return true;
```

Listing 14 - Separating axis theorem pseudo-code

Bounding Circles

Although performance became my number one priority for collision detection, I still needed an algorithm that was accurate enough for an error to occur. In “Beginning Android Games” Mario discusses 2D collision detection and suggests using a bounding shape approach [4]. This technique works by creating an invisible shape that is as close to the size of the object as possible and then testing for collision between the two predictable shapes. Circles are used in Kosmos because most shapes resemble a circle and will give more accurate results than a square.

Figure 18 shows an example of bounding circles being used. A collision was detected between the closest enemy on the left and the projectile touching it. As you can see, the accuracy of the collision test is dependent upon how close the circle maps to the size of the object. It took some fiddling to get all of the object’s circles to fit closely with their size, but the outcome was very accurate.

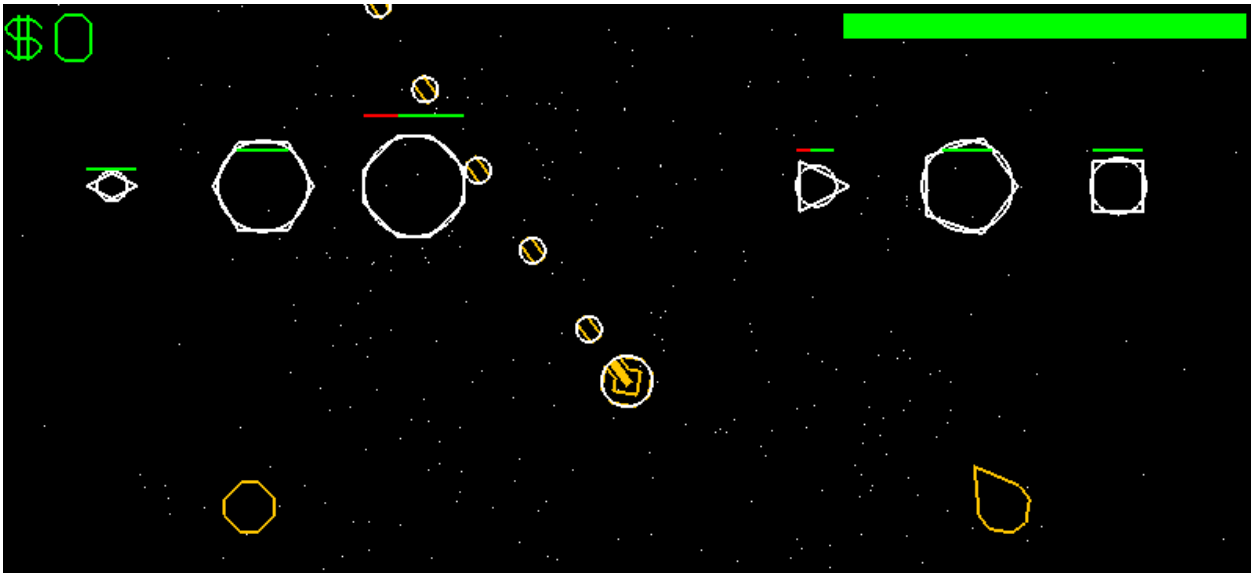

Figure 18- Bounding circles example

The code required to test for a collision between two circles is shown in Listing 15 and as you can see, it is much more efficient that the SAT code. The function works by calculating the distance between the centers of the two circles and comparing that with the sum of the radii. If the distance is less than the sum, then the circles overlap. This optimization proved to be successful and under the extreme condition where the separating axis theorem failed, bounding circles handled the situation without a stutter.

```
 public static boolean overlapCircles (Circle c1, Circle c2) {
      float x = c1.x - c2.x;
      float y = c1.y - c2.y;
      float distance = x * x + y * y;
      float radiusSum = c1.radius + c2.radius;
      return distance <= radiusSum * radiusSum;
}
```

Listing 15 - overlapCircles function

Text

In most games text is generated using an imported image that contains every possible character that will be used in the game. The large image is then separated into individual characters that can be accessed using their ascii representation as the index of the full image. The problem with this is that I'm not using imported graphics and wanted to keep a consistent look to the game. The only way to keep graphical consistency was to manually create every character the old fashioned way, with vertices and indices.

Font Class

The Font class is a utility that helps me write text to the screen. Utilizing Java's inheritance model I created an abstract class named Alphabet that contains two functions, getVerts and getInds. Listing 16 shows the declaration of the abstract class and its function prototypes. The purpose of this class is to allow me to create objects that extend the Alphabet class so I am able to then create an array of Alphabets. The classes that extend the Alphabet class represent characters and will return the vertices and indices of that specific character. In Listing 17 you can see the alphabet array and how I have the classes organized. The classes are arranged in an ASCII order so the character data may be used to index the correct class. For example, in Listing 18 I use the character 'A' to reference the A class and call its getVerts function. Using the array as a lookup table I can easily get the vertices or indices of each class when given a string.

```
 static abstract class Alphabet
{
   public abstract int getVerts(float x, float y, float z,
                                float w, float h, Color color);
   public abstract int  getInds();
}
```

Listing 16 – Alphabet class declaration

```
public static Alphabet [] alphabet = new Alphabet[]{

   _SPCAE,null,null,null,_$,_PERCENT, null, null,null, null,null,null,
   _COMMA,_NEG,_PERIOD,null, _0,_1,_2, _3,_4,_5,_6,_7,_8,_9,_COLON,
   null,null,null,null, _QUESTION,null,A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,
   S,T,U,V,W,X,Y,Z
};
```

Listing 17 - Alphabet classes in ASCII order

```
int numVerts = alphabet['A'].getVerts(X, Y, Z, width, height, color);
```

Listing 18 - Alphabet array example

Creating Characters

Creating each character in the ASCII table was a very long and monotonous task. Every vertex needs to have a position with respect to the origin that can be scaled by a width and height. On engineering paper I designed every character that is displayed in Kosmos and then later translated it into code. Figure 19 shows an example of how I would have designed each character.

The width and height of every character was initially created with units of 8 and 12 respectively. These dimensions gave a blocky look to the character that scaled well will when increasing the size like the character ‘A’ shown in the image below. The white lines represent the maximum size of the character and the top right and bottom left locations are labeled with the green text. The vertices needed to be determined relative to the given width and height and are calculated every render to allow flexibility.

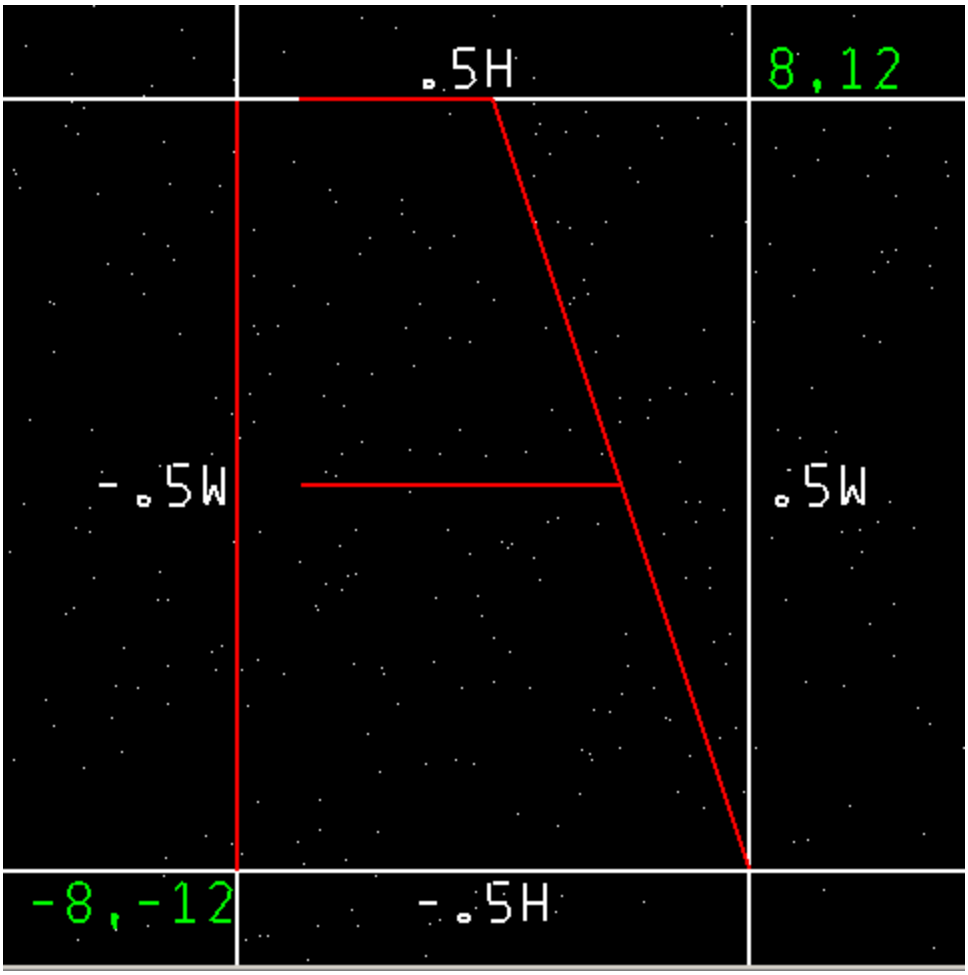

Figure 19 - Character creation

Listing 19 shows the A class and the vertices being set to the Font's static vertex array. This vertex array will then be allocated into the RenderUtils vertex array prior to the screen being rendered. This was a painstaking process and plenty of debugging was required to get the Font class completed and working as expected.

```
public static class A extends Alphabet
{
   public int getVerts(float x, float y, float z, float w, float h, Color color)
   {
      vertices[0] = (-w/2)+x; vertices[1] = (-h/2)+y; vertices[2] = z;
      vertices[3] = color.r; vertices[4] = color.g; vertices[5] = color.b; vertices[6] = color.a;
      vertices[7] = (-w/2)+x;vertices[8] = (h/2)+y; vertices[9] = z;
      vertices[10] = color.r; vertices[11] = color.g; vertices[12] = color.b; vertices[13] = color.a;
      vertices[14] = (-w*3/8)+x;vertices[15] = (h/2)+y;vertices[16] = z;
      vertices[17] = color.r; vertices[18] = color.g; vertices[19] = color.b; vertices[20] = color.a;
      vertices[21] = (0)+x;vertices[22] = (h/2)+y;vertices[23] = z;
      vertices[24] = color.r; vertices[25] = color.g; vertices[26] = color.b; vertices[27] = color.a;
      vertices[28] = (w/4)+x;vertices[29] = (0)+y;vertices[30] = z;
      vertices[31] = color.r; vertices[32] = color.g; vertices[33] = color.b; vertices[34] = color.a;
      vertices[35] = (w/2)+x;vertices[36] = (-h/2)+y;vertices[37] = z;
      vertices[38] = color.r; vertices[39] = color.g; vertices[40] = color.b; vertices[41] = color.a;
      vertices[42] = (-w*3/8)+x; vertices[43] = (0)+y;vertices[44] = z;
      vertices[45] = color.r; vertices[46] = color.g; vertices[47] = color.b; vertices[48] = color.a;

      return 49;
   }


   @Override
   public int getInds() {
      indices[0] = 0;
      indices[1] = 1;
      indices[2] = 2;
      indices[3] = 3;
      indices[4] = 3;
      indices[5] = 4;
      indices[6] = 4;
      indices[7] = 5;
      indices[8] = 4;
      indices[9] = 6;
      return 10;
   }
}
```

Listing 19 - A class

Appendix

Reference

[1] http://www.opengl.org/about/

[2] Millennial Media's Mobile Mix "2011 Year in Review" http://www.millennialmedia.com/mobile-intelligence/mobile-mix/

[3] Badlogicgames.com, "NEW CAMERA CLASSES IN LIBGDX" http://www.badlogicgames.com/wordpress/?p=1550

[4] "Beginning Android Games" by Mario

[5] http://www.metanetsoftware.com/technique/tutorialB.html

[6] http://www.codezealot.org/archives/55

Table of Figures

Figure 1- Machine gun, rocket launcher, and laser examples 5
Figure 2- Example Gameplay 6
Figure 3 - Virtual joysticks 7
Figure 4 - Main menu screen 9
Figure 5 - Load game screen 9
Figure 6 - Upgrade screen 10
Figure 7 - Weapon upgrade screen 10
Figure 8 - Pause screen 11
Figure 9 - Gameplay 12
Figure 10- Results screen 13
Figure 11 - Music screen 14
Figure 12 - The Confirm State 17
Figure 13 - Frustum example 18
Figure 14 - Top view of frustum 18
Figure 15 - Main Menu to Save Game Screen Transition 19
Figure 16 - Simple 10x10 square 21
Figure 17 - Grid based collision detection 25
Figure 18- Bounding circles example 29
Figure 19 - Character creation 32

Table of Listings

Listing 1 – Example of leaking memory 15
Listing 2 - Example of not leaking memory by using static variables 16
Listing 3 - 2D to 3D un-projection 20
Listing 4 - Rendering shapes with a Mesh 21
Listing 5 - Index array for a square 22
Listing 6 – RenderUtil's addVerts function 22
Listing 7 - addInds function 23
Listing 8 - BaseObject data 23

Listing 9 - Sets the rotation of a Polygon **24**
Listing 10 - Data declaration of Cell class **26**
Listing 11 - Collision condition function **26**
Listing 12 - clearGrid function **27**
Listing 13 - allocateGrid function **27**
Listing 14 - Separating axis theorem pseudo-code **28**
Listing 15 - overlapCircles function **30**
Listing 16 – Alphabet class declaration **30**
Listing 17 - Alphabet classes in ASCII order **31**
Listing 18 - Alphabet array example **31**
Listing 19 - A class **33**