ENGR 103 - Spring 2016
Freshman Engineering Design Lab

"Planetary Orbit Simulator"
Final Report

Date Submitted: **May 31, 2016**

Submitted to: *Dr. Richard Knight*

Emmanuel Oyekanlu

Group Members: *Nick Widmann*

Ebed Jarrell

Jiho Yoo

Abstract:

This project encompassed the planning, design, and programming for a planetary orbit simulator application. The goal was to create powerful and intuitive software that could be used in an educational setting to teach students about the effects of gravity in space. To complete the project, it was necessary to learn how to use Java and a series of graphics engines to manipulate bodies on the screen. A physics environment was constructed so that the bodies could experience the effects of gravity. A variety of features were implemented so that the user could interact with the environment and bodies. Finally, a user interface was incorporated to allow easy access to the various features in an on-screen hierarchical menu structure. The final deliverable was a working orbit simulator for use on Android and Windows devices. The app was tested in a high school physics class to see how well it performed. The test results were analyzed to determine in which areas the app needed improvement before it could be released. While the app performed well in some areas, the user interface needs to be further improved before it can be released.

Introduction

1.1 Problem Overview

The primary goal for this project was to create a realistic simulator of gravitational motion for bodies in space that could be used in an educational setting. To meet this objective, a number of tasks were identified that needed to be accomplished. The first task was to decide for what platform the app would be written, and consequently, the language in which it would be written. The user interface and programmatic structure of the app also had to be designed. Finally, the app would need to be built to meet the designs. For each of these steps, it was necessary to do some research in order to make informed decisions and understand how to utilize preexisting software, engines, and libraries in the app. An in-depth discussion of each step in the process can be found in the Section 2 of this report.

The intended audience for the app is anyone wishing to learn about planetary motion and gravity in space. In particular, in a classroom setting students could use the app to study planetary physics by performing calculations and comparing their results with predictions from the simulator.

1.2 Existing Solutions

At the start of the project, there were several orbit simulation applications already on the market providing existing solutions as examples. However, because none of them were open source, their source code could not be accessed or utilized. Despite this limitation, the available orbit simulators served as inspiration for designing the user interface in this project. Several apps were tested to decide which features should be included.

Table 1 compares the available apps. Features considered "advantages" were incorporated in the app developed in this project.

Table 1: A comparison of existing orbit simulators.

Simulator	Advantages	Disadvantages
Name: The Orbit Simulator [1] **Publisher:** Colorado.edu **Platform(s):** Computer	Gives information on orbit radii, planet velocity, and eccentricity.	Does not allow for creation of custom solar systems.
Name: Gravity and Orbits [2] **Publisher:** PHET **Platform(s):** Computer	Adjustable planet mass and velocity. Has preset orbit patterns. Allows turning on/off orbit trail.	Only 3 bodies can be modeled.
Name: Planetary Orbit Simulator [3] **Publisher:** NAAP **Platform(s):** Computer	Provides significant information about Kepler's laws. Has dynamic visuals, graphs, and formulas explaining each law.	Confusing setup. Only one planetary body can be included.
Name: Particle Sandbox Gravity Sim [4] **Publisher:** Neil Burlock **Platform(s):** Android	Many bodies can be modeled. Bodies can be pinned in place so they aren't affected by gravity. Bodies can be deleted and scaled in size.	Complicated setup and interface. Too many options on the screen. Little educational value (no numbers). Buttons too small. Hard to set planet velocity and manipulate planets because they are very small.

1.3 Project Objectives

After analysis of other orbit simulators on the market, it was decided that this project would focus on the creation of one for use primarily in an introductory educational setting. Many of the other simulators were either too complex or did not include the necessary tools to be useful in a classroom setting. The app developed here struck a balance between those two extremes, making it ideal for learning the basics of gravity in space. It was designed specifically for middle school and high school aged students and took into account their limited understanding of the material. The user interface design was intuitive yet powerful enough to run specific simulations of varying complexity.

2 Technical Activities

2.1 Completed Tasks

2.1.1 Planning, Setup, and Research

2.1.1.1 Preliminary Planning and Proposal

This project began with a planning phase where different ideas were considered. Originally, the end goal was to create an educational game. It would involve planetary orbits with an emphasis on being fun to play. There would have been a series of levels where the user was tasked with setting a planet's properties so that it could hit a required position on the opposite end of its orbit. Figure 1 shows an early design concept of how it would look. The proposal was written according to this design.

2.1.1.2 Major Redesign

After work began, there was concern about finding the balance between fun gameplay and educational value. It was decided to focus on the educational aspect of the app. The application would no longer have set levels with rigid constraints defining the user's goal. Instead, the user would be presented with a sandbox-like environment with the freedom to create any kind of solar system they wanted. A number of controls for the user were planned to make this possible. The specific controls are defined in detail in Section 2.1.3.

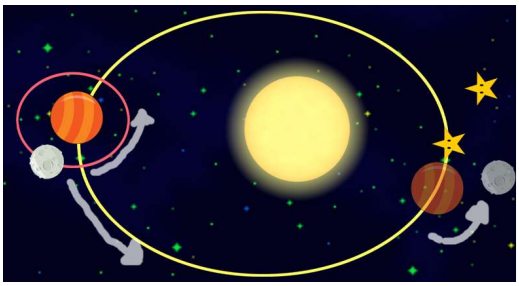

Fig. 1: Early design concept. User would adjust planet and moon properties to hit required points on the opposite side.

2.1.1.3 User Interface Planning

Developing an effective user interface was crucial. The menu and controls needed to be intuitive yet powerful so that the application could be useful to people with varying educational levels. In some of the other simulators analyzed, controls were much more intuitive than others. These simulators used features available in many common applications. For example, many of the simulators that were analyzed included a "scroll" or "pinch-to-zoom" function. This is a common feature in many games and applications, making it familiar to the user. Therefore, it was decided to include this in the application. Another example is "tap and drag back" to launch. This functionality is employed for launching projectiles in many popular games so it feels natural for the user to launch a body in the orbit simulator by tapping on it and dragging back. Incorporation of these intuitive features helped add to the overall simplicity that was desired.

It was determined that there should be an on-screen menu with several controls, both basic and more advanced. When the other simulators were tested, some were found to be too complex and difficult to understand. To avoid this, the menu would be able to contract to avoid cluttering the screen with too many controls. It would also be organized into sub-categories so that the user could easily navigate to various features of the app. An outline of the planned menu structure can be seen in Table 2 below.

Table 2: Structure of the on-screen menu.
Menu categories are in bold with corresponding options below.

Add Body	**Edit Body**	**Camera Options**	**More Options**
Add Single	Scale Size	Zoom	Delete all Bodies
Add Multiple	Set Orbit	Reset Camera	Settings
	Make Static	Center on Body	Help/Tutorial
	Body Properties		App Info
	Delete Body		

2.1.1.4 Early Research

After the initial planning phase, it was necessary to do some research to decide how to best carry out the designs. One team member already had some prior experience with Android app and game development so it was decided that Android would be the mobile platform of choice. Based upon previous experience, it was also known that an engine would need to be used. An engine is an extension to the Android development environment that allows for graphic manipulation and the handling of screen touch events. Many engines were considered but eventually libGDX, [5], was chosen. This engine stood out from the others because of its widespread documentation, large active user-base, and easily implemented cross-platform support [6]. The latter means that the app could be developed for Android but then deployed across other platforms, including Windows and iOS.

2.1.1.5 Setting up Software

Before any progress could be made on the application, it was necessary to install the necessary software. Android Studio [7], the official interactive development environment (IDE) for use with Android, was the first program installed. The next step was to download libGDX, and import its files into the Android Studio project.

Another important component to the software setup task was setting up GitHub. GitHub is an online repository service that allows for easy collaboration on software projects [8]. A repository was setup for this project so that everyone could contribute to the software and always have the most up-to-date files.

2.1.1.6 Learning Java

For all of the programming steps in this project, it was necessary to have a solid understanding of Java. One team member already had experience with the language and helped teach it to the others. Other resources were also used to learn Java [9]. Understanding concepts such as classes, objects, methods, and inheritance was important for completing the project objectives.

2.1.1.7 Learning to use libGDX with Box2D and Scene2D

Considerable time and effort was invested to learn about libGDX. LibGDX operates using the concept of a camera to determine what is shown on the screen and a main *render()* method (function) that is called continuously while the app is running. The camera converts the screen's pixels into alternate "world" units that are consistent across all devices. This is used to handle different screen sizes and create a consistent appearance. It also allows for zooming and panning around because images are given positions and sizes in world coordinates and their properties don't depend on how many total units are on the screen (Section 2.1.3.7 provides additional detail about zooming and panning). A graphic object, or *sprite*, has a set of properties, such as position and size. These can be modified in each frame (i.e. each time the *render()* method is called) to create certain effects such as making an image move or change size [6].

Two other engines, Box2D and Scene2D, were also used. These are both simpler to implement than libGDX and can be thought of as expansion packs. Box2D adds support for physics by allowing the creation of bodies. These bodies can have properties assigned to them that automatically update the state of the body every frame. For example, they can be given a velocity that automatically updates position after each frame based on how much time elapsed since the previous frame. They can also be assigned a mass and shape for automatically handling collisions with other bodies. Similarly, the bodies can have forces applied to them that update the velocity each frame depending on their mass [10].

Scene2D allows for the easy creation of user interface elements such as buttons and textboxes. It also provides tables that can be used to organize these elements into rows and columns that work the same way regardless of device size [10]. Section 2.1.4.1 describes the implementation of the menu using Scene2D.

2.1.2 Setting up the Physics Environment

2.1.2.1 Adding Bodies

The first step in setting up the physics was to setup the bodies that would interact. Box2D was used to create spherical bodies to represent planets, stars, and moons. Bodies were assigned a mass, m, and density, ρ, and the radius, r, was obtained by assuming a spherical shape as in Eq. 1.

$$r = \left(\frac{3m}{4\pi\rho}\right)^{\frac{1}{3}} \qquad \text{Eq. 1}$$

2.1.2.2 Adding Gravity

After the bodies were created, it was necessary to make them interact with each other. This was done using a series of loops. In each frame, the net force applied to each body was obtained from the vector sum of the gravity components acting on it from the other bodies. The component vectors were each calculated using Newton's Universal Law of Gravitation [11] for the force of gravity, F, between two objects of mass m_1 and m_2 separated by distance, r, as shown in Eq. 2 where G is the gravitational constant equal to $1\ m^3kg^{-1}s^{-2}$. Although the actual magnitude of G is $6.67 * 10^{-11}$ [11], changing it to 1 did not affect the behavior of the bodies in the simulation. It only made it easier for the user to enter values and for the program to handle them. Figure 2 shows an example of the net force applied to three bodies due to the gravitational force exerted by the others. In between frames, Box2D handled the movement of the body due to the forces [10].

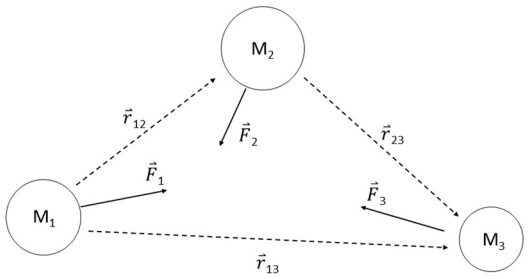

Fig. 2: Gravity acting between 3 bodies

$$\vec{F} = \frac{Gm_1m_2}{\|r\|^2}\hat{r} \qquad \text{Eq. 2}$$

2.1.2.3 Optimizing Scale

The last major challenge in setting up the physics environment was optimizing scale. The challenge here was that Box2D has a limitation of 2 units/frame on the speed for each body in the simulation [10]. Without modifying the scale, at 60 frames per second, bodies would have had a maximum velocity of 120 m/s. This would pose an issue because of the immense size and velocities of objects in space. For bodies to move realistically, it was necessary to incorporate a scale factor to the distances used in the code, r_{code}, to adjust the distances seen by the user, r_{user}, as presented in Eq. 3. Equation 1 could then be rewritten as Eq. 4 and Eq. 2 as Eq. 5.

$$r_{user} = Cr_{code} \qquad \text{Eq. 3}$$

$$r_{code} = \frac{1}{C}\left(\frac{m}{\frac{4}{3}\pi\rho}\right)^{\frac{1}{3}} \qquad \text{Eq. 4}$$

$$\vec{F} = \frac{Gm_1m_2}{(C\|\vec{r}_{code}\|)^2}\hat{r}_{code} \qquad \text{Eq. 5}$$

2.1.3 Adding User Functionality

2.1.3.1 Selecting Bodies

LibGDX's *tap()* method was used to select bodies [6]. When a tap event was registered by the program, it checked to see if the tap was within the radius of a body. If it was, that body was registered as the "selected body" and it could be edited. Otherwise, no body was selected or the current selected body was deselected. The user could then adjust non-body-specific settings such as the camera.

2.1.3.2 Launching Bodies

Using libGDX's built in *touchDown()* and *touchUp()* methods, the ability for users to launch bodies was incorporated [6]. When the user touched (on Android) or clicked (Windows) down on the screen, the program checked to see if the touch event was within the radius of the selected body. If it was, the program recognized that launching had begun. When the user released their finger, a launch vector, $\Delta\vec{x}$, was created between the touch down and touch up locations (in world dimensions). Figure 3 shows how the magnitude of this vector was then assigned to the body's velocity vector, $\vec{v}$, in the opposite direction than they dragged back.

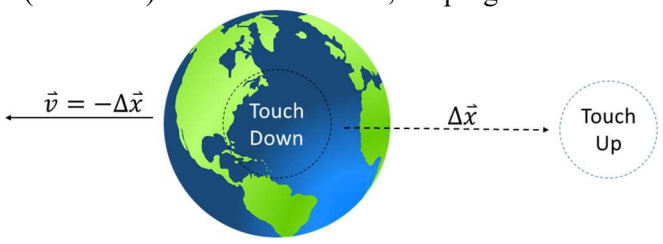

Fig. 3: Tap and drag back to launch

2.1.3.3 Scaling Bodies

Bodies could also be scaled in size by the user. This functionality was added using libGDX's *pinch()* (on Android) and *scroll()* (on Windows) methods [6]. When a body was selected, if a pinch event was registered as shown in Fig. 4, the program used the ratio of the initial distance between fingers, d_1, and the final distance, d_2, to scale the mass of the body, m_0, to a new mass, m, using Eq. 6. The radius, r, was then adjusted accordingly using Eq. 1. Scrolling worked in a similar way. If a scroll event was registered, the mass of the body was increased if the scroll direction was forward and decreased if it was backward. As the user scrolled, the mass, m, increased relative to the square root of m_0. Equation 7 shows the relationship between Δm and m where n is an integer corresponding to the number of clicks of the mouse wheel (forward is positive, backward is negative). To make it easier for the user to input values, the mass of a body was permitted to vary from one kilogram to one million kilograms. Although these masses were much smaller than typical planets and suns, the exact values were not important for the behavior of the simulation (similar to the gravitational constant as explained in Section 2.1.2.2). What was important is that the user could create a body that was one million times more massive than another.

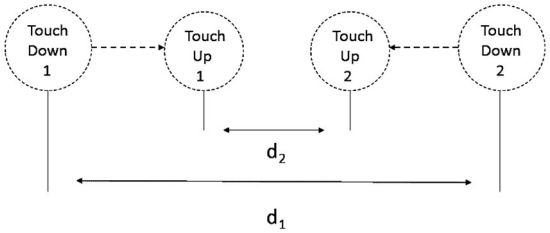

Fig. 4: Registering a "pinch" event

$$m = \frac{m_0 d_2}{d_1} \quad \text{Eq. 6}$$

$$\Delta m = 10n\sqrt{m} \quad \text{Eq. 7}$$

2.1.3.4 Adding and Deleting Bodies

The ability for the user to add bodies was incorporated with the same function used to create the initial bodies when building the physics. When the user tapped or clicked on the screen, a body would be created at the coordinates that the tap event was registered. This body was assigned a default mass of 1000 kilograms but could be adjusted by the user. The user could also choose to add an n-by-n (where n is an integer) array of bodies. This was accomplished using the same function as adding a single body but called many times to create a square pattern. It operated using two loops with one corresponding to the rows and one to the columns.

2.1.3.5 Orbiting Bodies

Functionality was incorporated to automatically give a body the velocity needed to orbit another body in a perfectly circular orbit. This was done using the formulas for centripetal force and the force of gravity [11]. The magnitude of the velocity was calculated using Eq. 8 where m_1 is the mass of the body being orbited, $\vec{r}$, is the distance between the two bodies, and G is the gravitational constant. This magnitude was then used to scale the unit vector perpendicular to $\vec{r}$ and tangent to the desired circular orbit.

$$\|\vec{v}_{orbit}\| = \sqrt{\frac{Gm_1}{\|\vec{r}\|}} \quad \text{Eq. 8}$$

2.1.3.6 Static (Sticky) Bodies

For some situations, a user may want to keep a body in place to analyze only the effects of its gravity on the other bodies. For this, the ability was added to constrain or "stick" a body in place. This was done by giving the bodies a

Boolean property called “movable”. If “movable” was true, it behaved normally. If not, the body would affect other bodies but it would not experience any forces due to other bodies. Linear damping is a Box2D property that can be assigned to bodies to make them experience frictional forces proportional to velocity, similar to air resistance [10]. Sticky bodies were given a linear damping value of 10 million, an arbitrarily large number. Changing this effectively gave the body an extremely large coefficient of friction in space. That way, when other bodies collided with a sticky body, it would not move due to the momentum of the collision.

2.1.3.7 Zoom and Pan Camera

As explained in Section 2.1.1.7, libGDX worked with a camera that showed a specific view of the world [6]. This view could be modified programmatically by zooming or panning around. The functionality was implemented so that a user could control the camera to their liking. The user could pinch their fingers or scroll the mouse wheel forward and backward to zoom in and out respectively. They could also tap/click and drag to translate the camera laterally. These functions were implemented very similarly to scaling and launching bodies, however, a conditional statement in the *touchDown()*, *touchUp()*, and *pan()* methods checked to see if a body was selected. If a body was selected, the body-specific functions discussed in Sections 2.1.3.2 and 2.1.3.3 were performed. Otherwise, the camera was adjusted. Figure 5 shows how the camera functions work.

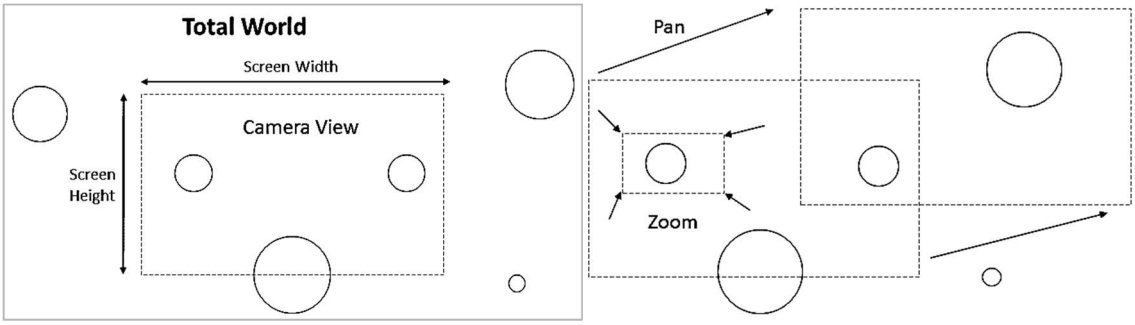

Fig. 5: The camera isolated a region of the total world to display on the screen. The region could be zoomed in and out or panned laterally.

2.1.4 Implementing User Interface and Graphics

2.1.4.1 Adding Menu

The functions discussed in Section 2.1.3 were incorporated into an onscreen menu as buttons within the different submenus as shown in Table 2. Each menu element was organized in a separate Heads Up Display (HUD) file. In a video game, a HUD is an in-game menu that overlays the gameplay with on-screen controls [6]. For this application, the HUD was setup with its own camera and dimensions so that the size of the buttons would not be affected by the main camera’s zoom. Scene2D was used here to organize the buttons and give the menu its structure [10]. The sliding out effect was achieved using Scene2D’s animate function; this function takes in a set of destination coordinates for the graphics element and a transition time as parameters [10].

The scale body function had a popup slider that was used to adjust the radius of the body. The maximum value on the slider corresponded to the radius that would give the body the maximum allowed mass (1 million kilograms). Figure 6 shows a prototype of the menu. The lower row of buttons is the main menu. The row above that is the "edit body" submenu that slid out when the user selected "Edit" from the main menu. At the top is the scale slider that appeared when the user selected "Scale".

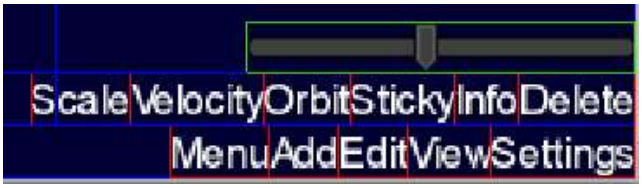

Fig. 6: A prototype of the menu.

2.1.4.2 Importing Textures and Skins

Before any graphics could be used in the application, they needed to be loaded as a texture atlas. The texture atlas was a large image containing many of the graphics used in the app that saved memory and time by limiting the number of individual files that needed to be loaded. The atlas then had an accompanying text file that specified the location and orientation of all of the images [6]. The creation of the atlas was done using libGDX’s Texture Packer [5]. It simplified the process by taking an input directory of images and then automatically formatting the texture atlas to minimize the memory required when running the app. Figure 7 shows the texture atlas used for the menu buttons.

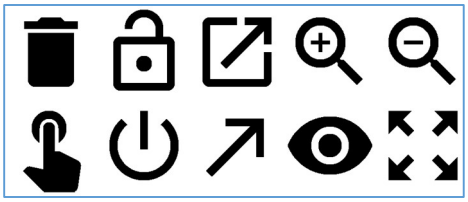

Fig. 7: Texture atlas for menu icons

2.1.4.3 Creating Dynamic Sprites

Unless Box2D was set to debugging mode, the bodies did not actually show up on the screen [10]. A picture could be placed at specific coordinates on the screen using a sprite, libGDX's built in image class, but sprites could not be attached to bodies to move around on their own [10]. To overcome this limitation, the DynamicSprite class was created. This class inherited all of the functionality of the sprite, but included additional features. In particular, it allowed for the attachment of a body and included an *update()* method that synced the sprite's position, size, and orientation to match the body.

2.1.4.4 Creating Launch Simulation

A launch simulation feature was incorporated that used Euler's method [11] to predict the path of a body before it was launched. During a launch, the *drag()* method was called when the user moved their finger or mouse on the screen [6]. The launch simulation then created a launch vector and calculated and displayed a projected path of the body if the user were to release their finger at that moment. This worked using a series of position steps and a constant time step, Δt. The trajectory of the body was calculated by solving simultaneously Eqs. 9 and 10 for the velocity, $\vec{v}$, and position, $\vec{x}$, of the body, respectively. Figure 8 shows an example of four position steps. Three hundred positions were calculated in total. For every 10th position, a dot was placed on the screen at that position. This created a simulation as shown in Fig. 9.

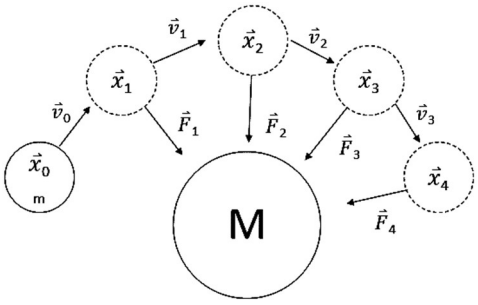

Fig. 8: The force, velocity, and position were recalculated for every step in the simulation.

$$\vec{v}_n = \vec{v}_{n-1} + \vec{F}_G(\vec{x}_{n-1})\Delta t \qquad \text{Eq. 9}$$

$$\vec{x}_n = \vec{x}_{n-1} + \vec{v}_{n-1}\Delta t \qquad \text{Eq. 10}$$

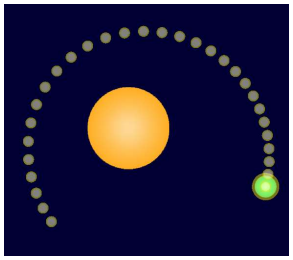

Fig. 9: The launch simulation predicted the path of a body before it was launched

2.1.4.5 Creating Windows

A series of four windows were created using Scene2D. These could all be opened using buttons in the menu. The "body properties" window allowed the user to explicitly view and adjust the position, velocity, and mass of a selected body. This functionality was important if a user wanted to run a specific simulation. Figure 10 shows a screenshot of the "body properties" window in action. The user can change the properties to their liking and then press "Save" to apply the changes and close the window.

Another important window that was added was the tutorial window. This gave general information about the usage of the app, outlined specific controls, and explained the functionality of the buttons. This was designed for new users who need to learn how to use the app. Similarly, an app info window was added that gave the user information about the developers and the origin of the application.

The last window added was the settings window. This gave the user the ability to adjust some factors in the game, such as the length of the launch simulation and the number of bodies created when adding a body matrix.

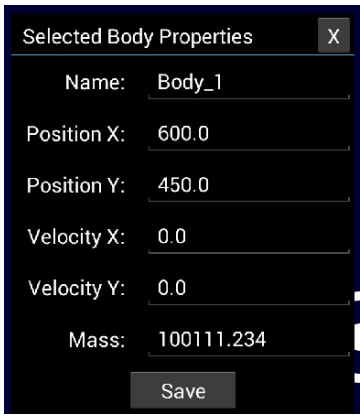

Fig. 10: The "body properties" window allowed a user to view and edit a body's properties

2.2 Project Timeline

Table 3: Gantt chart showing timeline of project

	Week									
Task	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Planning	x	x	x	x	x	x	x			
Major Redesign		x								
Research	x	x	x	x	x	x	x	x		
Software Setup	x									
Creating Physics		x	x							
Adding User Functionality		x	x	x	x	x	x			
Creating Menu					x	x	x	x		
Creating Graphics				x	x	x	x			
Implementing Graphics					x	x	x	x		
Testing		x	x	x	x	x	x	x	x	
Field Testing (Explained in Section 3)									x	
Final Report/Presentation Preparation						x	x	x	x	x

2.3 Project Budget

Table 4: Budget comparing item values with actual cost

Item	Value	Actual Cost
Android Studio	$135.00	$0.00
libGDX, Box2D, Scene2D	$1,350.00	$0.00
Adobe PhotoShop	$90.00	$0.00
Labor Cost	$15,000.00	$0.00
Total	$16,575.00	$0.00

2.3.1 Android Studio

Android Studio was created through a partnership between the software companies Google and JetBrains. Google offers it free to make Android apps. Its value was estimated using JetBrains' other piece of software, IntelliJ IDEA. This has all of the features that Android Studio does except that the user interface isn't designed specifically for creating Android apps [13]. IntelliJ IDEA costs $15.00 per month for an individual license [13]. Three licenses for three months equaled a total value of $135.00.

2.3.2 libGDX, Box2D, and Scene2D Engines

LibGDX, Box2D, and Scene2D are open-source engines that are free to use [1, 6, 7]. Their value was estimated using a similar piece of software called Unity. Unity has all of the same functionality but it is closed-source and more user friendly, coming with its own IDE [14]. It costs $150 per month for an individual license [14]. Using that estimation, three licenses for three months would be valued at $1,350.

2.3.3 Adobe PhotoShop

Adobe PhotoShop costs $30 per month for an individual license [15]. Only one team member required PhotoShop so for three months the value came to $90.00. This person already had an older version of PhotoShop installed on their computer that was not payed for monthly. Therefore, this was not an expense that had to be covered for this project.

2.3.4 Labor Cost

The United States Bureau of Labor Statistics listed the average hourly wage for an Applications Software Developer at $49.12 [16]. For an estimated 300 hours' worth of labor on this project, the cost would be approximately $15,000.

3 Results

The final deliverable for this project is the planetary orbit simulator that was created for Android and Windows platforms. The initial objective of creating a working simulator was met. All of the intended features were added. The features and the appearance of the application are described in detail in Section 2.1.

Besides creating a functioning app, the other major goal for the project was to make the app intuitive and useful for high school physics classes. These factors were tested in the field by sending the working version to a physics teacher in Weare, NH at the beginning of week 9. He tested the app with his students and had them fill out a short survey. The results of the survey were analyzed to determine how well the goals were met and what improvements should be made. Listed below are the survey questions. They were designed to pinpoint specific issues in the app that may need improvement.

1) Rate the following on a scale from 1 (completely disagree) to 10 (completely agree) and (optionally) leave a comment explaining each rating:
   - The physics of gravity in the simulation was accurate for everything I tried.
   - All of the features I want are incorporated.
   - The app is NOT overcomplicated with too many features.
   - The menu icons are not confusing.
   - I would recommend this app to other physics students.
   - I would use this application on my own.
2) What was your favorite feature? Why?
3) What was your LEAST favorite feature? Why?
4) If you could add any feature what would you add? Why?

The survey came back with 27 total respondents. The specific results are shown in Table 5.

Table 5: Survey results

Question 1 - 1 (disagree) to 10 (agree)	**Rating Percentages**									
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
App has accurate physics	0%	4%	4%	0%	12%	4%	8%	19%	12%	38%
App has enough features	4%	0%	0%	4%	0%	12%	12%	20%	12%	36%
App is not complicated	0%	0%	8%	12%	4%	4%	8%	12%	12%	42%
Menu icons are not confusing	4%	8%	4%	12%	8%	0%	12%	12%	8%	35%
I'd recommend it to physics students	4%	0%	4%	0%	8%	15%	15%	19%	8%	27%
I'd use it on my own	19%	8%	8%	15%	15%	4%	8%	8%	4%	12%
Question 2 - Favorite Feature - Most Popular Responses										
• Launching bodies										
• Pausing the simulator										
• Adding bodies										
• Locking bodies in place										
Question 3 - Least Favorite Feature - Most Popular Responses										
• Confusing menu										
• Poor instructions, can't find tutorial										
• Launch simulation was "glitchy"										
Question 4 - Feature to Add - Most Popular Responses										
• Better menu										
• A fun aspect (challenges)										
• The ability to click and move bodies around the screen										

4 Discussion

4.1 Analyzing Results

As discussed in Sections 2 and 3, nearly all of the objectives for this project were accomplished. The app worked and included all of the planned features. To discover how the app could be further improved before it is released, the results of the survey outlined in Table 5 were analyzed and trends were identified. One issue that many of the respondents agreed upon was that the menu and the icons were confusing. This could be addressed by replacing the icons with text buttons and including an interactive walk-through tutorial instead of the window. Many people surveyed also commented that they could not find the tutorial. This could be fixed by automatically opening or starting the tutorial the first time the user runs the application. One recommended feature was the ability to click and move bodies around the screen. The way the app was set up, users could adjust the body velocity by launching. They could also set the velocity and position manually using the "body properties" window. However, there was no simple and intuitive way to change the position of the body. Clicking and dragging to place the body would be a good solution to this issue but it would conflict with the launching process which already functions using the same commands. Another highly requested feature was some kind of fun aspect. A specific suggestion was the inclusion of challenges to create certain orbit patterns, such as a "sun-moon-earth" or "figure-8" configuration. Completing challenges like this could encourage learning by giving students a reason to use the app on their own. According to the survey results, while the majority of students said they would recommend the app to other physics students, most said they would not use the app on their own. This piece of feedback was important for recognizing the overall need for improvement in the app before it can be released.

It is important to recognize possible sources of error in these results. One issue was the fact that there were only 27 students surveyed who were all in the same class. Therefore, the results could be skewed due to a small homogenous sample size. Another issue with the way the survey was conducted was that the students were setup with the app and told to experiment on their own. Because they were not tasked with using it as part of a lab, which was the designed use of the simulator, their responses may not be an accurate portrayal of the app's performance.

4.2 Reflection

Many skills were learned in the process of making this application. Two members of the team had no previous object oriented programming experience (i.e. Java). In addition, everyone learned to use the libGDX, Box2D, and Scene2D engines. Everyone on the team also learned how to collaborate on a programming project by dividing tasks and using a version control system such as GitHub. Another skill everyone developed was documenting progress by writing daily in the project blog.

5 References

[1] *The Orbit Simulator*. University of Colorado Boulder, 2007.

[2] *Gravity and Orbit*. University of Colorado Boulder, 2016.

[3] *Planetary Orbit Simulator*. The University of Nebraska-Lincoln, 2008.

[4] *Particle Sandbox Gravity Sim*. Neil Burlock, 2016.

[5] M. Zachner, *LibGDX*. 2013.

[6] S. Nair and A. Oehlke, *Learning LibGDX Game Development - Second Edition.* Birmingham, UK: Packt Publishing, 2015.

[7] *Android Studio*. Google, 2016.

[8] P. Bell, and B. Beer. *Introducing GitHub: A Non-Technical Guide*. Sebastopol, CA: O'Reilly Media, Inc., 2015.

[9] R. Gallardo, S. Hommel, S. Kannan, J. Gordon, S. Zakhour, *The Java Tutorial: A Short Course on the Basics-Sixth Edition*. Redwood Shores, CA: Oracle, 2015.

[10] James Cook, *LibGDX Game Development by Example*. Birmingham, UK: Packt Publishing, 2015.

[11] Anton, Bivens, and Davis. *CALCULUS: Early Transcendentals-10th Edition. Hoboken,* NJ: John Wiley & Sons, Inc., 2012.

[12] B. Phillips and B. Hardy. *Android Programming: The Big Nerd Ranch Guide*. Atlanta, GA: Big Nerd Ranch, 2013.

[13] *IntelliJ IDEA.* JetBrains, 2016.

[14] *Unity®Pro*. Unity Technologies, 2016.

[15] T. Knoll and J. Knoll, *Photoshop*. Adobe System, 2016.

[16] United States Department of Labor, "Software Developers, Applications", Bureau of Labor Statistics, 2015.