Final Project Report

Marc-Antoine Dubé	40029307
Jonathan Hsu	40053971
Michel Chatmajian	40001987
Kevin Luu	40037514
David-Étienne Pigeon	40068000
Patrick Pin	26789366

COMP 371
Nicolas Bergeron

Concordia University

August 15th 2019

TABLE OF CONTENTS

1.0 Project Overview

1.1 Summary of project

Storm Area 51 is a roller coaster simulation in space. It travels around a world made of multiple stars and planets along some tracks with its rocket spaceship. The spaceship can look at 3 camera modes: The Explore Mode, the Guide Tour mode and the Center of Origin mode. The first mode puts the spaceship in first person camera perspective so that the user is free to move around the world. The second mode puts the spaceship on tracks so that it gets to ride the roller coaster. The last mode is simply a mode where we see the Sun at the center of the world. In addition, the rocket can shoot projectiles.

1.2 Expected Results vs. Actual Results

At the very beginning of the project, we wanted to generate very realistic tracks using a real 3D track model. After trying to generate the tracks with a real model, we realized that it was going to be hard to keep the poles of the track model to be aligned and distorted with the spline. Also, we realized that it was going hard on memory if we loaded the model multiple times throughout our spline.
With the change of scenery, going from realistic roller coaster in the real world to a fictional space environment, most of our models changed. We instead had to generate a random world at every program execution with stars, planets and a spaceship going on floating tracks to each planet.

For the skybox, the final result was as expected. It created a background to our world which added to the scale of the environment of space. It made the scenery more immersive even though the backgrounds were static images.

1.3 Github

https://github.com/DPigeon/COMP371-Project

2.0 Methodology

2.1 Cleanup Boilerplate (Michel)

Beginning from the assignment framework a lot of work had to be done to completely strip the scene that was presented to the user which consisted of some models placed using scene files. Our goal was to have a completely randomly generated world which had no space for any hardcoded models. The world class's draw function was stripped of any draws for hardcoded models such as the animation keys, spheres and cubes. A primitive function which generates planets in random locations was also added in order to have a working boilerplate for all developers to work on in parallel. The spline camera mechanism was also integrated from the lab framework as well making use of the center of the randomly generated planets as the sample points. The spline generation was still quite primitive as the planets were not sorted in any way and they were also positioned way too close to one another causing many collisions into neighbouring planets and too many sharp turns.

Another big effort was ensuring cross platform reliability because half of our team was using MacOS and the other half on windows. This meant that whenever project files were added or deleted we need to update both configurations as soon as possible in order to have the smallest amount of downtime between developers. Rather than working on our features separately we took the approach of making small incremental pull requests which allowed us to quickly modify Xcode and visual studio configs whereas giant PRs would have made this difficult. This meant that we spread out the risk of features being merged together at the end and not working.

Another aspect of this was also the integration of third party libraries which sometimes did not work on one platform or the other. In order to properly integrate these we once again made small atomic PRs which integrated the library and coordinated between each platform's integrator in order to maintain parity across all of our development platforms.

2.2 Planet Generation (Marc-Antoine)

Planet generation was done in many phases. Planets were required to then generate the spline, so the first iteration was imply creating planets at random points with random sizes. This approach obviously had many flaws, the first one being that planets could overlap.

The second solution was thus to add a collision detection algorithm that would compare with all existing points (planets) the distances between then with regards to the max scaling size and the distance ratio between planets. The ratio was set because without it, planets could be really close to each other and we wanted the planets to be more spread out.
The second approach seemed to work fine until we started playing with the world size, the number of planets to generate, the size of the planets and the distance ratios. There is only a finite number of planets that we can generate in a 100x100x100 world, which means that if the

number of planets is too high, there would always be a collision and there would be an infinite loop and the program would never properly start.

We thus decided to add a retry mechanism to the random point generation. If a point could not be placed after 5 tries, the planet would simply not be generated.

The colors of the planets are also set randomly and passed to a vertex shader so that the color can be applied.

2.3 Planet Sorting (Marc-Antoine)

Sorting the planets also posed to be a challenging task. Without any sorting, planets can be very far from each other, thus the spaceship would do back and forths in the world to go to the planets. This was resource intensive as it took much longer to generate the spline and tracks.

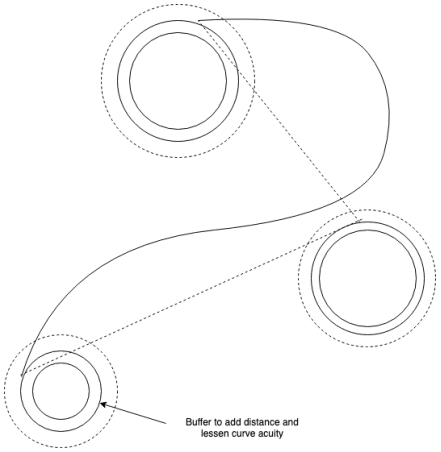
The first solution we came up with was to sort the points by the magnitude. This made sense at first because all points were generated in one quadrant. However, because we are in three dimensions and not two, we would get some weird angles and the spaceship would simply just not flow well through space.

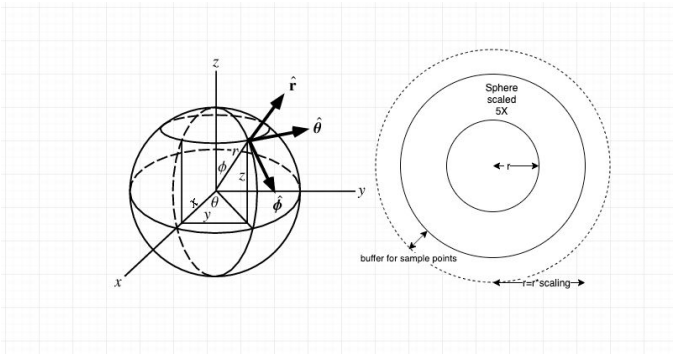
The second iteration was to again start by sorting by magnitude so that we get the closest vectors together first when iterating. The check we added is to measure the angle between the last vertex the spaceship was navigating on (from one planet to another) with the next potential vertex. If the angle was reasonable and realistic, we would use this planet. If not angle was found, simply go to the next planet regardless of the angle.

For the planet generation and sorting, it would have been extremely useful if we would have taken the time to implement a sandbox environment where we could input controlled values instead of random ones. This would have had greatly improved the debugging time we had to take.

2.4 Spline Optimisation (Michel)

In order to increase the entropy of the splines being generated , a mechanism was added which converts the center point of each planet into spherical coordinates using the vector $radius + centerPoint$ in order to extract the phi and theta components which are then randomized and converted into cartesian coordinates this makes it so that the sample points selected now are: for one no longer at the center of the planets making the spline generated a good distance away from the planets themselves lowering the chances of running into a planet in the track, moreover the sample points are now completely randomized and could be either at the top of the sphere or the bottom or even from the sides adding a lot more entropy and excitement to the track being created.





On top of this in order to decrease the chances of colliding inside of a planet a variable buffer was added into the above equation used to get the initial point being converted into spherical coordinates $radius + centerPoint + buffer$. This buffer allows us to fine tune the algorithm to decrease the chances of a collision as much as possible and smooth out the spline possibly making turns less acute. The buffer also allows a lot of planned physics features to be implemented in the future such as a gravitational pull being generated by each planet based on their mass which would potentially pull sample point in closer when the rocket is in orbit distance for example.

2.5 Track Generation (David)

Since our first main idea was to do a real roller coaster simulation, the first big step was to generate tracks in our program. First of all, the spline points were extrapolated into a text file (this technique is enhanced in the next section). Then, all those points were read and loaded into a vector of points. Next, the cylinder tracks were created specifying n number of slices in the circles. Using glBegin and glEnd functions, vertices were created in the middle of a circle. Then, with a certain angle, another point was created to make a slice piece of pie in our circle. Another circle was created with the same characteristics and was linked to the previous one with *GL_TRIANGLE_FAN*. After repeating this algorithm for m - 1 amount of spline points, a track was generated. Finally, two cylinder tracks were done and both were moved parallel to the spline on each side. In addition, perpendicular cylinders were also implemented to get a real 'train' track. However, this was removed because the perpendicular cylinders seemed to be infinite from one side of the skybox to the other side of it. This feature was implemented this way because it was easy to visualize what was happening with the drawing of primitives. It was also a good feature to learn about how one can draw with primitives and vertices.

2.6 Extrapolation (David)

After the optimization of the spline, track points became random every time we launched the program. We realized that it would be better to load these spline points. Therefore, an algorithm to extrapolate and load these points was implemented. First of all, all the spline points were loaded into a vector of points. Then, points were compared by skipping 300 points in case the

first point was the same as the 300 first points. Afterwards, as soon as a point was the same as the first point, a flag was triggered and meaning that loading was done.

2.7 Loading Screen (David)

After extrapolation of all the track points were done, a loading screen was implemented. A package called ImGui was used to integrate this interface and some additional libraries already installed were also used with this feature (GLEW). The loading screen had to be initialized in viewport so that the interface shows in screen space. After track generation and having a loading screen show in viewport, a boolean flag was implemented as a function to check when loading was done. Therefore, as soon as the boolean flag is triggered, the loading screen is not shown anymore.

2.8 Planet Description (David)

At the beginning of the project, we had a discussion on amazing other features we could implement. We had the idea of being able to click on a planet and get a description of them. This feature is about 85% complete. Ray picking was used in this algorithm. First of all, rays had to be converted from viewport to world space using the view and projection matrix. Once rays were obtained in world space, the intersection point between the ray and spheres around the world could be found. Considering that the origin of the ray was the camera position, the ray-sphere intersection test could be implemented using many of the equations seen in class about ray casting and picking. The idea was that once the user clicks on the left mouse button, a ray is casted from screen space to world space. If it intersects any of the spheres planet model, then give its planet name on an interface. Unfortunately, this feature was not completed on time and was implemented differently because of time limitation. Instead if the user clicks on the screen, it will output a random plane name. This alternative is not how ray casting should work, but since of time limitation, another way to implement this feature had to be found. More information for this feature will be found in the future work section.

2.9 Rocket Model (Michel)

In order to draw the rocket so that it followed the position of the camera it was drawn in view space and had its world matrix set to identity as such. It was then translated and rotated into position. In order to add a slight bit of yaw, pitch and tilt the camera's lookat vector was transformed into spherical coordinates and phi and theta angles were extracted allowing us to calculate the necessary rotations for the rocket to make.

2.10 Star Generator (Kevin)

Being in space necessarily means having to have stars to be present. For the generation of the stars, we have implemented textured points drawn randomly in a defined cubic area of the world. Each points vector are generated between a value from 0.0f to 1.0f then these generated positions are divided by the ratio of the max value (which is 1.0f in this case) and the defined area. Given it some sort of percentage of which part of the cube to occupy. Each points are drawn with *glEnable(GL_POINT_SMOOTH)*, which enables the points to be round shaped. 500 points are

generated which provides enough visibility of the stars and doesn't cause too much performance issue. A star texture is then applied to each points and each texture faces the camera at all times. These generated billboards must have *GL_BLEND* enabled to enable transparency blending and *GL_POINT_SPRITE* and *GL_VERTEX_PROGRAM_POINT_SIZE* to change the size of the billboards before drawing the stars.

Implementing the stars this way is probably the simplest way of doing it. Having them as billboards creates the 3D feeling like that we want. Although, it might not look as realistic as other possible alternatives. Another alternative would be turning them into models, or loading them as OBJ files. This would be too expensive since we had to load multiple number of stars, while not causing too much performance issue.

2.10.1 Bloom effect

To implement the bloom effect of the stars, we had to extract the image and blur it. After blurring the image, we had to sum up this blurred image with the original image giving it a bloom effect. To blur the image, for each pixel of the stars we calculate an offset on the texture coordinate of the x-axis and the y-axis. This offset is from -2 to 0 to 2. 0 being the starting point -2 being the left/down pixels depending on the axis and 2 being the right/up pixels. The offset is summed up and divided by 25, since the double for loop does 5 horizontal and 5 vertical passings (5x5). Using 5 as the number of offset is what made it the most performant, using 10 would then do 10x10 passings which caused a lot of lag. While calculating this offset this creates a horizontal blur and a vertical blur of the image. The result is added back to the original color. This enhances the color and creates the desired bloom effect. Bloom effect is an easy concept, you just need to blur the image and add back the original, but making a good bloom effect is hard. Another approach that was tempted was the Gaussian blur, but in the end it did not work out too well due to the difficulty and the lack of time.

2.11 Skybox Generation (Jonathan)

To give an illusion of a bigger world, a skybox was implemented. A skybox is simply a texture applied to the background of the view. This is usually done using a cube that surrounds the world with textures on all six sides. There are also spherical skyboxes which envelopes the world in a sphere with a warped texture to account for the curvature. There is no real advantage between using a skysphere or a skybox and vice-versa. Their performance in rendering is not that different due to shaders.

To implement the skybox, the shaders have to be created, the mesh of the cube has to be setup and the textures have to be binded. For the shaders, the projection and view matrix are passed into it along with the texture color. To make sure that the skybox does not move when the camera moves, the translation is taken off of the view matrix in the vertex shader. Then, the fragment shader simply sets the color of the texture to the assigned pixel. Texture coordinates can use the fragment position since the skybox is essentially at the origin. This makes it so that any fragment position vector is the same as its direction from the origin. To bind the textures, the lightweight image loader library stb_image was used to load the texture files for all 6 faces of the cube. This is a essentially cube map that shows the world around. For each image,

glGenTextures() is used to generate a name for the texture. After that, glBindTexture() is used with the GL_TEXTURE_CUBE_MAP parameter along with the generated texture name. Once the textures are binded, glTextImage2D() is used for each face of the cube so that the shader can read the texture information being passed. The textures sent are clamped to the edges. To setup the mesh of the skybox, we take the cube model vertices and indices (obtained online not using the cube in the assignment framework), create and binded our VAOs, VBOs and also IBO since we are also passing indices to the shaders. The most important part of this implementation is to do glDepthMask(GL_FALSE) at the beginning and re-enabling the depth mask once the skybox is rendered. This makes it so that when the skybox is rendered its depth is 'infinite' so that any other model rendered is always in front of the skybox since it is the background. The skybox is rendered before anything else.

At first, without any research on skyboxes, a cube the size of the world was rendered (with depth mask) and everything else is contained in the cube. This was a big mistake as it created a lot of problems. Since the size of the cube is fixed, any model rendered outside of the cube will not appear inside of it. Also, the camera could travel outside of the cube. Then, the most problematic issue is that the far plane of the view frustum can be before the end of the cube when the camera travels far enough from any side of the cube. That side of the cube will disappear since it is further than the far plane.

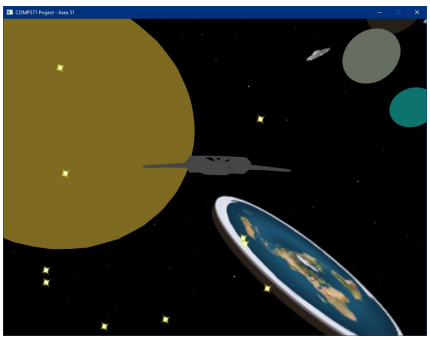
After more research, the skybox does not actually contain the world. The cube contains the camera. This makes it so that it does not matter how big the world actually is and that the far plane would not cut out the skybox. This solution is possible by disabling the depth mask for the skybox rendering so that objects behind the cube can be shown as now the cube is much smaller and contains the camera.

2.12 Sun Model and Lighting (Patrick)

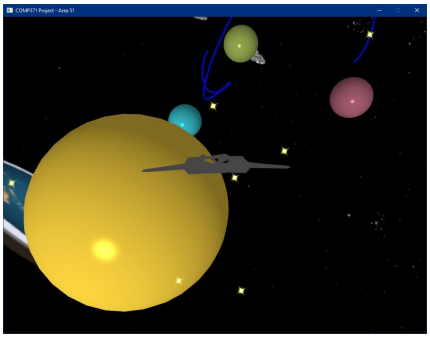
A solar system wouldn't be complete without a sun and the light that comes from it. While we did not replicate the real solar system as we know it, we still felt that it was appropriate to add the sun as a light source in our randomly generated solar system. The idea was to create a sun model and apply Phong lighting to the planets using a point light, just like the real life sun that would emit light in all directions.

The sun was generated using the same algorithm as the planets, except for the fact that it has a predetermined position at the origin (0, 0, 0) along with its scaling set to 10 times the initial radius of a planet (which is 1) and has different material coefficients compared to the planets. The original plan was to place a point light at the sun's position in order to accurately replicate the way it emits light throughout the solar system.

However, placing a point light at the origin seemed to have presented some problems. The point light appeared to have been encased within the sun model itself and ambient light was only shown outside, as shown in the first figure to the right.



A compromise that was implemented to circumvent the light blocking issue was to place the point light in the opposite quadrant of the planets, given that all randomly generated planets were always placed in the positive (+x, +y, +z) quadrant. As a result, the point light was moved to the negative quadrant, at (-10, -10, -10) to be exact. Moving the point light presented its own set of issues, however. For instance, as the sun had the same material coefficients as the planets, it was affected by the lighting. In addition, there was a point light effect on the planets from the Phong lighting, shown on the right, which does not reflect a large light source like the sun.



To prevent the sun from being affected by the lighting, the sun needed to have its own set of material coefficients. Ambient light for the sun was set to 1 to give the sun its full color and diffuse and specular light were set to 0 in order to remove the illumination effect from the lighting. Also, to remove the point light effect from the Phong lighting, the shininess (specular exponent) was set to a very high value, 100,000 to be exact, to give the planets a more uniform illumination effect from the direction of the sun model. Setting things up in this way would effectively emulate how the real life sun illuminates the planets in the solar system.

2.13 Audio Implementation (Patrick)

As OpenGL does not support audio by itself, we must add a third party library in order to implement audio functionality. Some audio libraries that are supported by OpenGL are irrKlang, OpenAL and the Simple and Fast Multimedia Library (SFML). With no prior experience in adding and implementing any library to any application worked on in the past, this was an interesting challenge. It involved a significant amount of research and trial and error until implementation was successful.

The first library that was looked into and eventually successfully implemented was irrKlang, as it offered a simple and intuitive way to add audio into the OpenGL application. This library allows us to create a sound engine that can be used to add 2D and 3D sounds into the world. The difference between 2D and 3D sounds is that 2D sounds always play at the same volume regardless of current position, and 3D sounds are placed at a certain position in the world, where volume is affected by how close the sound is relative to the current position. In the context of this project, we added background music as a 2D sound that loops. In addition, we added a laser sound effect to the projectiles. This sound was also set at 2D, since the rocket the projectiles shoot from is always at the center of the camera and thus always reflects the current position.

Other libraries that were looked into were in fact OpenAL and SFML, but once we got irrKlang working, these libraries were disregarded. Another thing to note is that while the irrKlang library was successfully integrated and fully functional on Windows, we were unable to integrate it on the MacOS platform. As a result, there would be no audio present while trying to run the application on MacOS.

2.14 Projectiles (Patrick)

The ability of rocket to shoot projectiles was inspired by the basic shooter shown in third lab of this course. It involved the addition of a velocity attribute to the models, which are updated based on the position and velocity of the projectile. In the context of this project, projectiles will always shoot in a straight line based on the rocket's current position.

Initially, the projectiles were cube models, similar to what was shown in the lab, except resized to a small rectangular prism to emulate the shape of a laser. However, problems arose when shooting the projectile at different camera angles, particularly when shooting upwards or downwards as the projectiles were locked to the xyz coordinate system. While rotation could have been implemented to orient the projectile to the appropriate camera angle, the projectile was ultimately changed to a sphere (scaled down to 0.05x the initial radius) due to time constraints.

Since the rocket is shooting lasers, we had to make sure that the projectiles weren't affected by lighting, otherwise it would look like that the rocket was shooting pellets. As a result, the projectiles were set to the same material coefficients as the sun and was arbitrarily given the blue color. In addition, the laser sound effect was added to match each instance of a projectile, where they both were given the same key input, which is the spacebar.

2.15 Main Menu and Settings (Marc-Antoine)

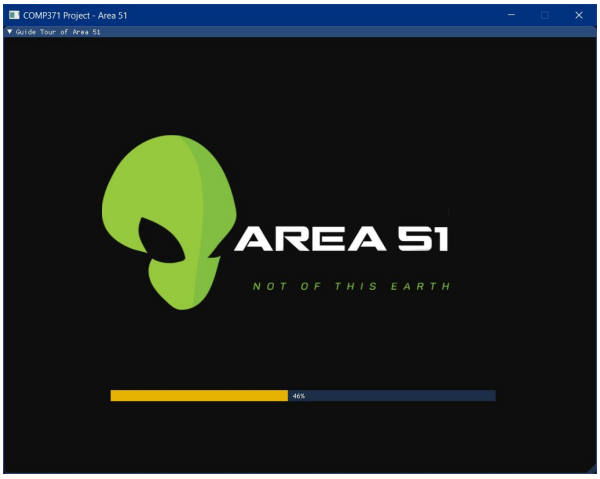
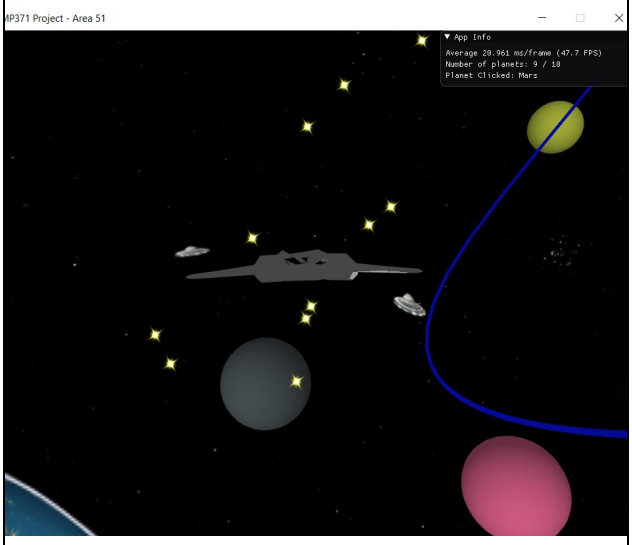
A main menu screen and settings screen was added using the IMGui framework which I did not know and had to learn upon starting this task. The placement of elements was code-based, but as someone coming from the web development world, I understood the concepts fairly quickly. We started by adding a program state variable to tell on which screen the user is currently on (LOADING, MENU, SETTINGS, RUNNING) and added navigation functionalities. For example, instead of closing the program when clicking on the escape key, it would bring to the main menu and from there, it would be possible to exit the program using the exit button. By lack of time, I could not fully implement the settings screen (discussed further and future work).

3.0 Results

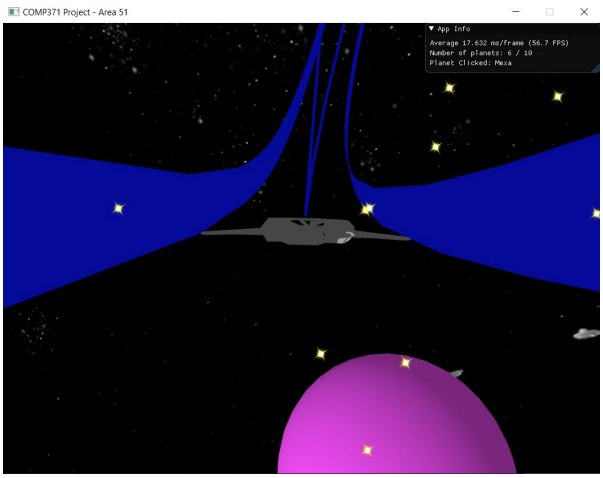
3.1 Explanation of Results

The final outcome for the track generation was good, but tracks could have been generated with better quality and performance. For instance, we could have used a model for our tracks instead of drawing vertices in the world. It could have been better if we had found a way to optimize it. For the skybox, the final outcome was better than expected. Since the background is made up of static images, there was some skepticism associated with the end result. When combined with everything else in the scene, it brings a new layer of immersion into the world. There could have been improvements to the skybox like adding layers in the skybox that produce different effects like having a sun in the skybox and using the position of it in the skybox to produce a directional light.

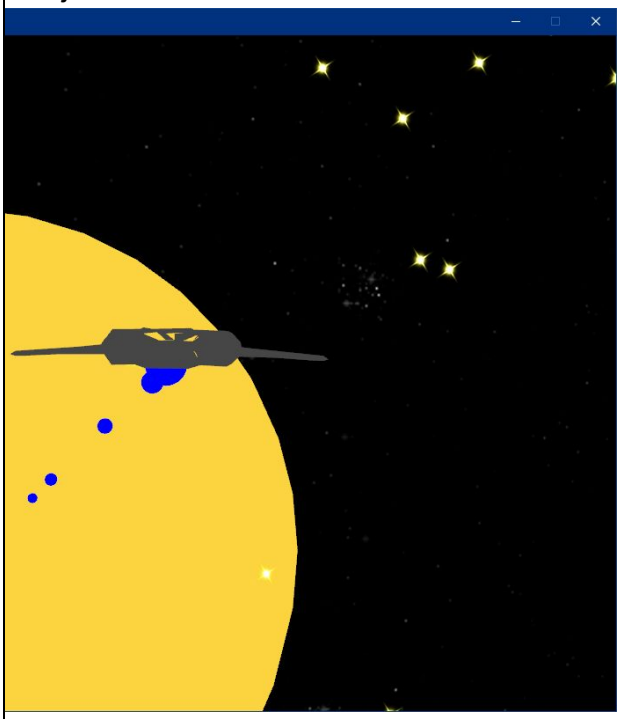
3.2 Screenshots (Major Features)

Loading screen 	Stars and planets 

Track generation



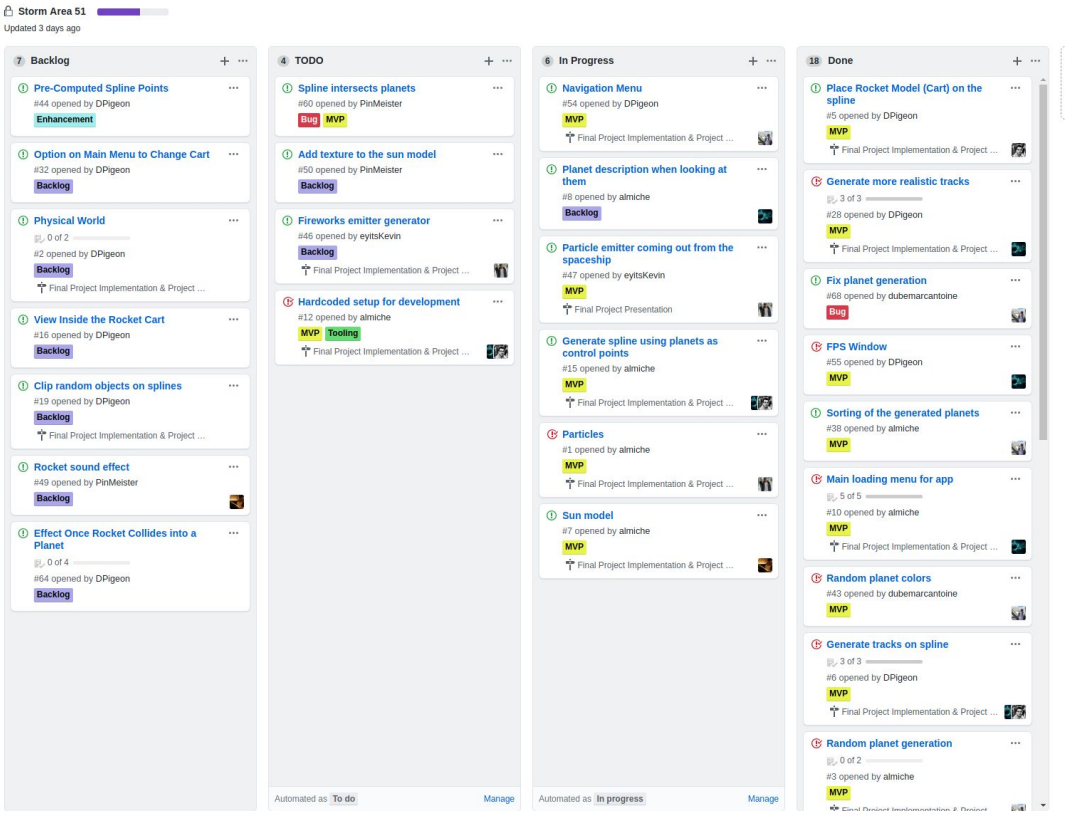
Projectiles and sun



4.0 Discussion

4.1 Approach

We took an iterative approach towards this project. None of us having experience in 3D graphics programming before, this project posed quite a challenge for us. We thus decided to start off from the assignment framework and iterate on it and add each of our goals step by step without doing many tasks at once. We had a project board on Github to define our tasks and issues and we prioritized them according to how much the task was needed for the project or not. The MVP tasks were the ones that had to be done first as they were required to add further functionalities. This was done in order to move most of the risk in our project at the beginning in order for the team to gauge the viability of the project. For example, without planets, we could not generate a spline which we would then generate tracks from. The use of small atomic PRs allowed the team to ship small incremental parts of a feature into the master branch and into the hands of other developers rather than waiting for an entire feature which could slow down other features depending on it being completed.



4.2 Future Work

If we would have had more time, the planet description feature would have been working 100% with a full description of the planet and its name. The planet description algorithm could have even be integrated with color picking. Also by playing with the particle system and particle emitter provided by the assignment 2 could provide the possibility of implementing the firework system. We've wanted to use a texture atlas to support multiple textures. Finally, the projectile feature could be extended to a point system where when a projectile collides with a planet or an object around the world, it gives the user points.

The settings screen would also have been a nice addition to modify the parameters of the program (world size, planet size, planet count, etc) or even update the graphics quality (toggle stars on or off).

4.3 Learning Outcome

Most of us were only beginners when it comes to programming in C++, let alone learning OpenGL which was all new to us.

Working as a team on an OpenGL was harder than we initially thought. Some features that were implemented by different people were working solely on their own, but when it comes to merging everything together into a master branch some aspects were not taking into consideration and would break previous implementations. An example of this would be the texture for the rocket and for the stars. When integrating both of them together, only one texture would apply, and that would affect both the stars and the rocket. In order to fix this issue, we had to generate the texture name at the very beginning and then bind them differently when we draw them.

5.0 User Manual

5.1 Compilation

5.1.1 Libraries

The required libraries packaged with the code are the following:

- GLEW 2.1.0
- GLFW 3.3.0
- GLM
- FreeImage 3170
- irrKlang 1.6.0 (for audio)

5.1.2 Packages

The required packages to run the program are the following:

- ImGui: It is already integrated within the project in the source files under /Source/Packages/ImGui

5.1.3 Windows

To compile and run the program on Windows, the user can simply go under the VS2017 folder and run the Area-51-Project.sln file. Afterwards, the user can click on 'Local Windows Debugger' to compile and run.

5.1.4 MacOS

1. Open the file /Xcode/Area51.xcodeproj which will load XCode
2. Add necessary assets in the build phase
3. Open file Source/RocketModel.cpp and go to line 23.
4. Set the rocketPath to the path leading to the file /Assets/Models/rocket.obj in the project.
5. Click on the run button at the top left
6. The program will compile if necessary, then launch

5.2 Program Interaction

Once the program is up and running, the user has to wait until everything is loaded. Once loaded, the user will be on the main menu screen where they will be able to go into the Guide Tour roller coaster mode. They can press 1 to go to the Explore mode to move freely in the world. Pressing 2 brings the user to the center of the origin where the Sun is. Then, pressing 3 goes back to the

Guide Tour mode. Next, the user can press 4 to add stars in the world. Finally, the user can press the spacebar to shoot projectiles.