**CS 1566 Final Project Report**

Jon Albert

Kevin Ireland

Adnan Khan

Joshua Zwolan

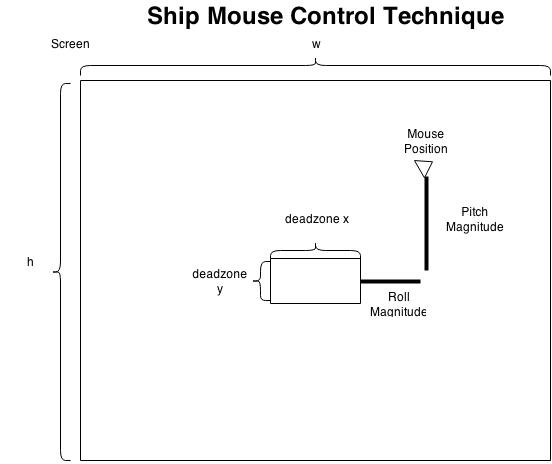
1. **Introduction**

Our project is a simulation of a procedurally generated planetary system. The user is allowed to fly through this system using a freely moving camera that can be turned and accelerated like a spaceship. In addition the user can choose to follow any of the planets in the scene and rotate about them while the planet makes its journey through the system.

This project incorporates every concept worked on during class, from 2D drawing to shadow mapping and ray tracing. Since what we are displaying is a space scene, it allows us to have a considerable amount of freedom in the kinds of techniques we use to display the overall scene.

There are two main view types accessible to the user; one of them is a “spaceship view” that allows the user freedom to move in all directions. The spaceship view has a 2D reticle overlaid onto the 3D view, which gives metrics about orientation and speed to the user. The other view, which allows the user to follow planets, is excellent for capturing aesthetically appealing shots of the system.

1. **Camera and Controls**



**Space** – Stop Motion (full stop)

**W** – Accelerate Forward

**S** – Accelerate Backward

**D** – Yaw Right

**A** – Yaw Left

Mouse Move Up – Pitch Up

Mouse Move Down – Pitch Down

Mouse Move Left – Roll Left

Mouse Move Right – Roll Right

Mouse motion over the window is used for looking up/down and rolling and is calculated as an offset from the center, excluding a small dead zone where the mouse can rest with no motion applied.

The mouse triggers movement as long as it is not located in a dead-zone in the center of the window.

Mathematically, all of the camera movement was accomplished by utilizing a 3x3 Rodrigues rotation matrix to rotate a vector (or a point) around another vector. These simple operations, when combined with the mouse, created a very intuitive user control scheme.

* Pitch was implemented by taking the cross product of the look direction and up vectors, and rotating about the resultant axis.
* Roll was implemented by rotating the up vector around the look direction vector for the camera.
* Yaw was implemented by rotating the direction vector around the up vector.

These calculations led to values for the gluLookAt call which properly oriented the ship in the correct position.

Another feature for camera motion that we implemented was to allow the user to “follow” specific planets as they progressed along their orbits. This lets us get very picturesque views of the planetary system such as the sun rising behind a larger planet while the camera is situated on the planet’s moon. While in this camera mode, the user can rotate around the planets on the xy-plane by pressing either the ‘j’ or ‘k’ keys for clockwise or counterclockwise rotation about the z-axis of the planet. At any point the user can activate the ray tracer with the ‘r’ key, which allows for the display of advanced shadows at the cost of performance. Ray tracing can be turned off with a capital ‘R’ or the ‘m’ key.

1. **User Interface**

Overlaid on the 3D view is a 2D user interface; it pulls values from the spaceship and displays metrics like orientation and speed to the user.

To properly overlay the 2D HUD, a new model view and projection matrix had to be pushed on the stack, and the face culling and lighting bits had to be disabled for the UI to be properly colored. The speed bars were created using custom hollow circle routines and predefined starting and ending points for different speed levels. This gives the appearance of four speed levels. The speed numbers themselves are drawn using gl lines and switch cases to make a custom number font, styled after the 7-segment LED of a desk calculator.

Using glRotate and Gl lines heading markers were marked every 22.5 degrees along the perimeter of the HUD. In order to use these markings the ships rotation about its heading vector were represented by two rotating orientation markers. This allows the user to know which way is the “original” up.

The center reticle uses the mouse x and y to mouse around within the HUD. In the future if we were to implement mouse grabbing through the operating system this could allow the user to see where their look/roll direction is.

1. **Planetary Movement**

Planets in this simulation move in a manner that approximates Keplerian motion. Since our simulation does not incorporate mass and gravity, motion is along an ellipse following the “equal time, equal areas” principle from Kepler’s second law. Precisely calculating the areas would require expensive integration, so areas are instead calculated using trigonometry. This introduces some error as it underestimates the area of each slice, but it results in motion that shows planets slowing as they move further from their orbital focus, and faster when they move closer.

Using the Kepler approximation function, the distance from the center of the ellipse is calculated, and the entire ellipse is then translated such that the focus lies at the center of the object being orbited around. Orbital planes are also defined by a normal. After the position from the focus is calculated, the point is rotated about the axis that results from the cross product of the default orbit plane normal—which is (0,0,1)—and a randomly generated normal vector that defines the rotation of the orbital plane. Without this all of the orbits would lie on the 2D XY plane. To demonstrate the orbital planes, the program has the ability to use GL\_LINE\_LOOP to show the paths that all planets and their satellites would take.

All planetary positions are calculated separately from the rendering sequence; this avoids relying excessively on the matrix stack and better facilitates tasks like ray tracing.

In addition to the orbits, the planets rotate about their own axes; this is handled by storing a randomly generated rotation axis and incrementing an angle based on a randomly generated rotation rate. Many of the same helper functions that were written for the ship rotations were useful in calculating the positions and orbits for the planet.

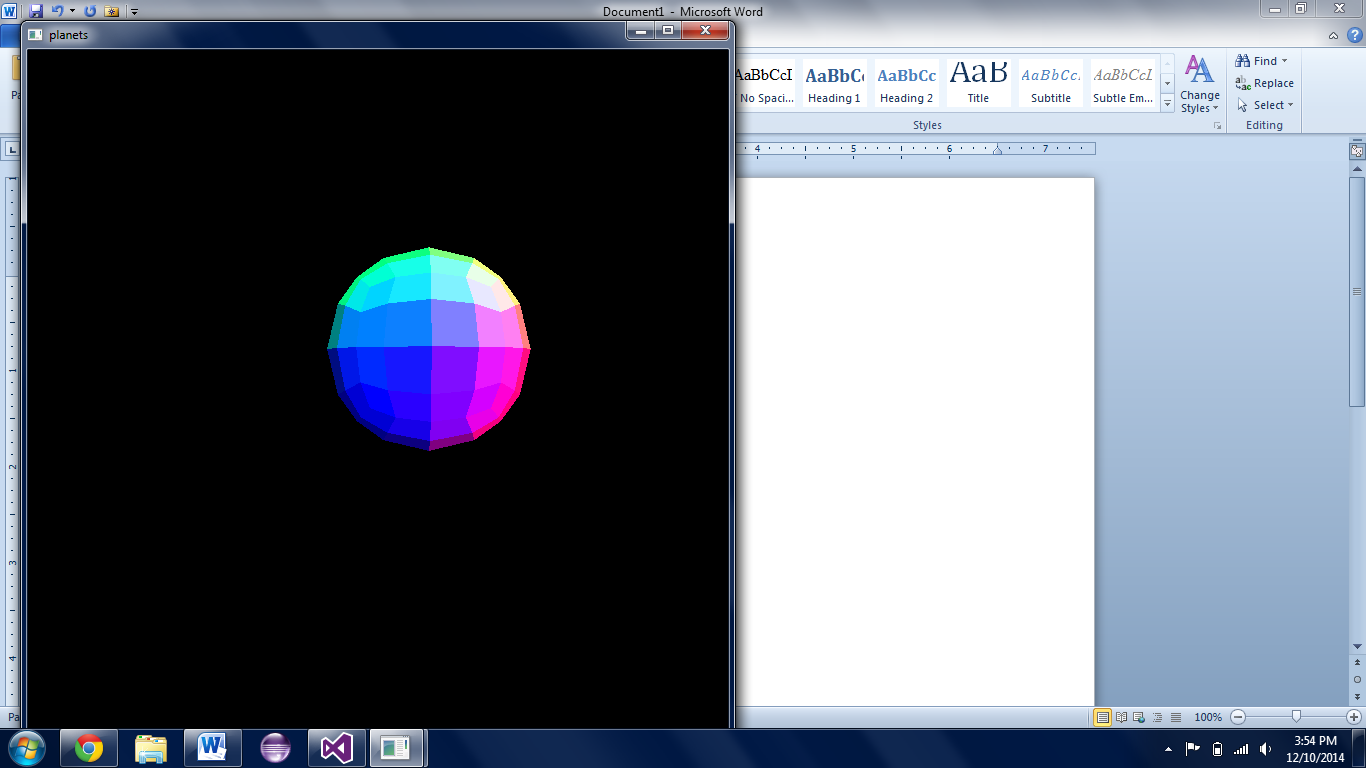
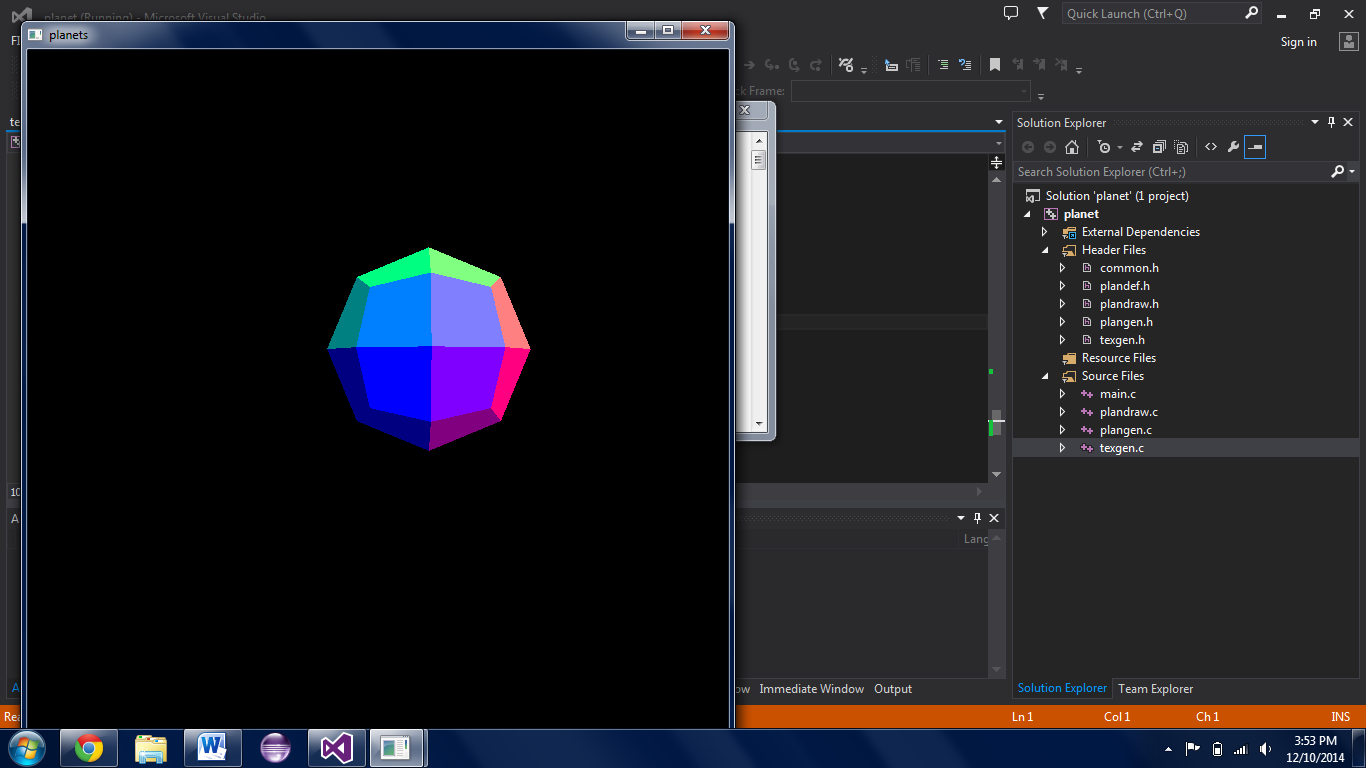
1. **Texturing and Planet Appearance**

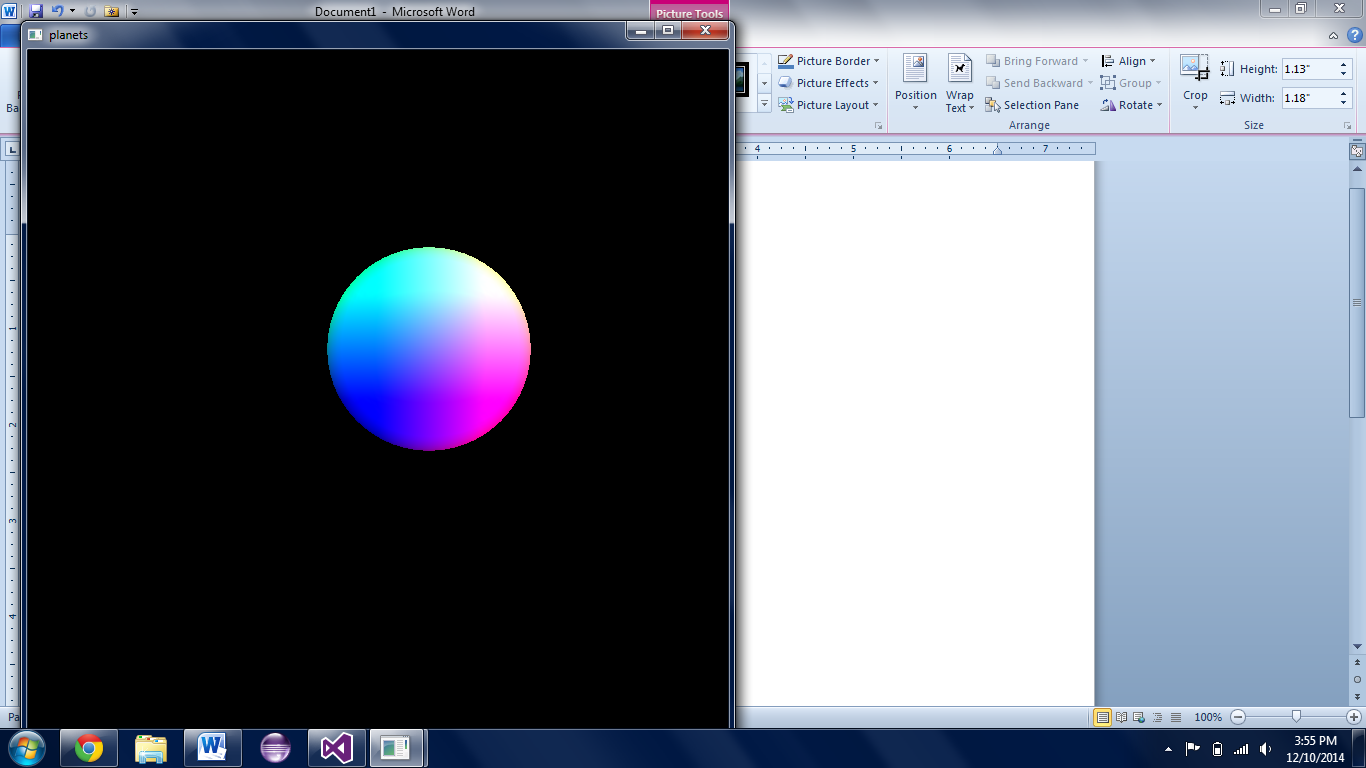
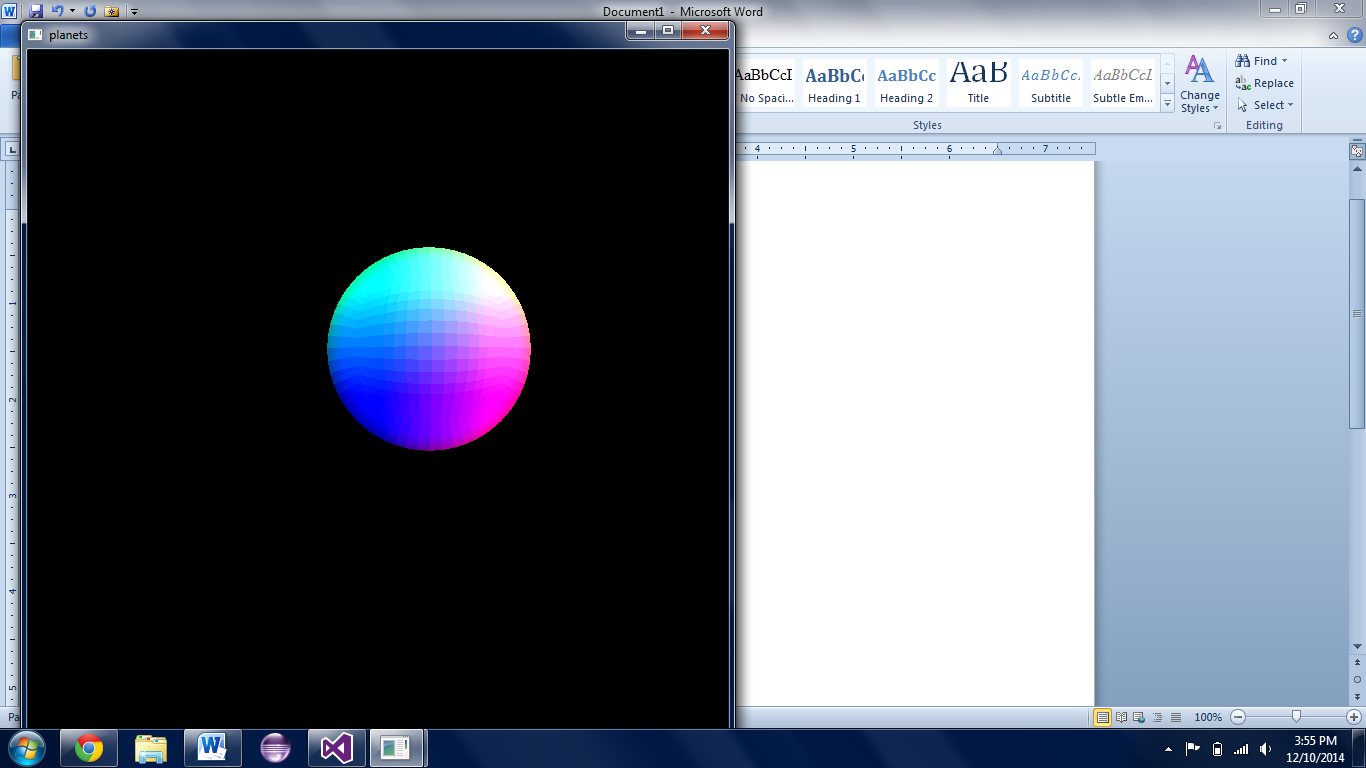
Noise functions for procedural worlds have been used for years, and these concepts were an inspiration for this project. The main challenge for this specific portion was math, and secondarily finding good RGB sets to use in the textures. For texturing the sphere apart from glu’s Quadratic wizardry, a custom sphere drawing function had to be defined through the help of some moderately complex mathematics. This is accomplished by bulging the mini-quads of a cube outward to a specified radius, thus emulating a sphere (see Results). From there, an initial thought was to map the texture on each “side” of the cube sphere, since the textures themselves were periodic and thus could loop around the cube. However, this turned out to produce poor planar mappings with obvious seams due to the textures being oddly oriented or skewed for every side but the front and back. To solve this, we looked to the class notes and decided to use mapping from sphere coordinates to UV coordinates.

Still, there was an obvious seam on one side of every planet, which was due to the fact that sine functions are periodic and that the planets were not of infinite resolution. In other words, the conversion from XYZ to UV assumed we were operating in a perfect world of all real numbers, and thus the texture would loop around itself near the borders of the texture map, producing an unseemly belt. To remedy this, a technique was implemented to check if the function was near the edge of the texture. This was still not perfect, as can be seen in the results. Nevertheless, under a fine enough resolution, the problem almost completely disappears, and, for the scale of our project, this technique will work just fine.

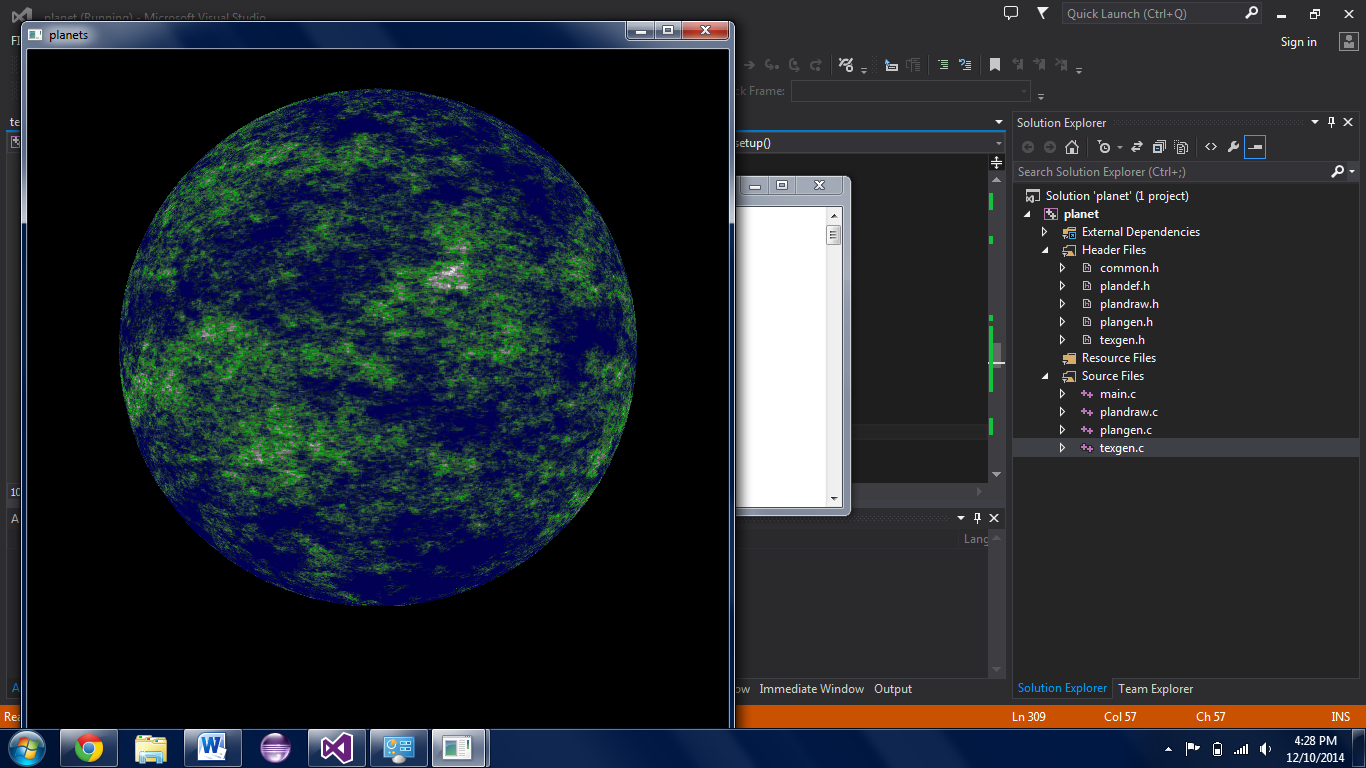
For texture generation the choice was Brownian motion, which was decided after considering noise algorithms. This technique could approximate the look of the more formalized noise functions through a very simple (though perhaps more computationally expensive) algorithm. Essentially, one or more points are placed on the “map” of the texture, and they are “walked” each “time step” to a randomized direction. At each time step, the map is incremented where the walkers “stand.” The borders are made to be periodic to avoid forcing the walkers to bounce off the boundaries of the texture map, which also allows for looping the textures side by side if necessary. This easily generates a noisy image with a “height” component—which values can be fed into a gradient function to evoke the appearance of land on the sphere. Further, the walkers may be biased, so that they can have a greater chance for movement in one or more directions. For example, a function can be configured to predominately move the walkers horizontally. This, depending on the gradient, can make the sphere look like a gas giant and adds all the more variation to our solar system.

**Texturing Results and Discussions --**

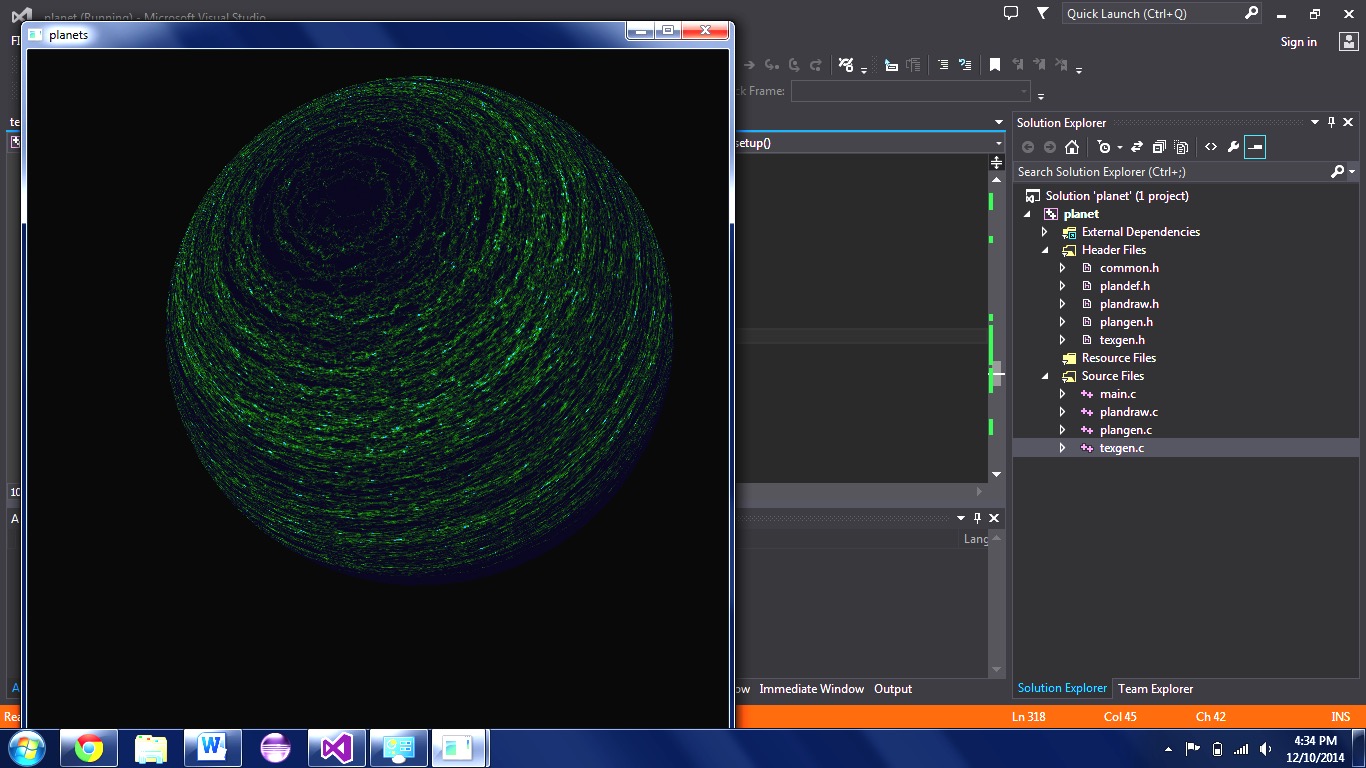




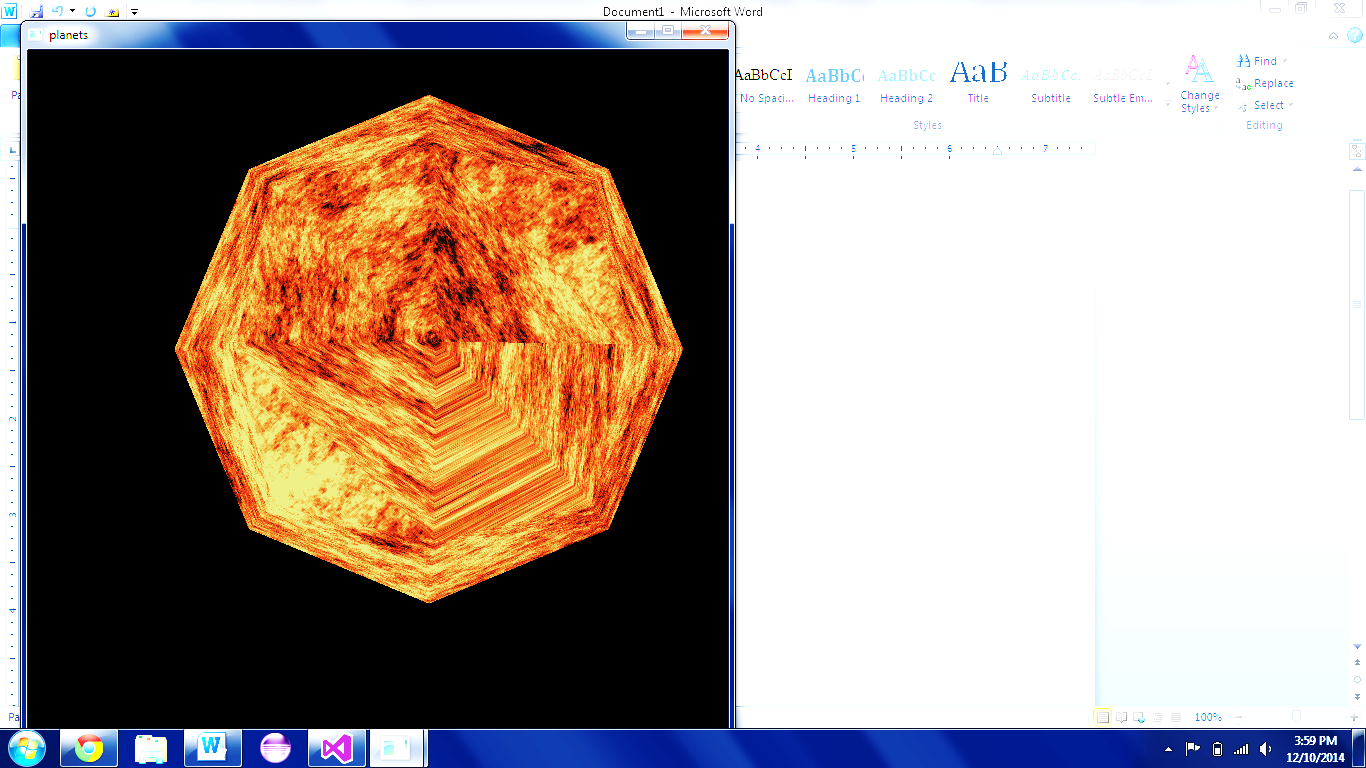
Above is an example of what the different cube resolutions look like when bulged. The colors correspond to each quad’s top left corner’s XYZ value, where X= R, Y = G, etc. With higher resolutions, the cube looks increasingly more like an actual sphere. At very high resolution, however, it becomes too computationally expensive to even draw one sphere by itself and still process user input. Thus, most images hereafter (and the planets in the final package) are drawn with a resolution equal or similar to the bottom left image. Some results of texturing the spheres can be seen on the right; a brief discussion of the difficulties in texture mapping is on the following page.



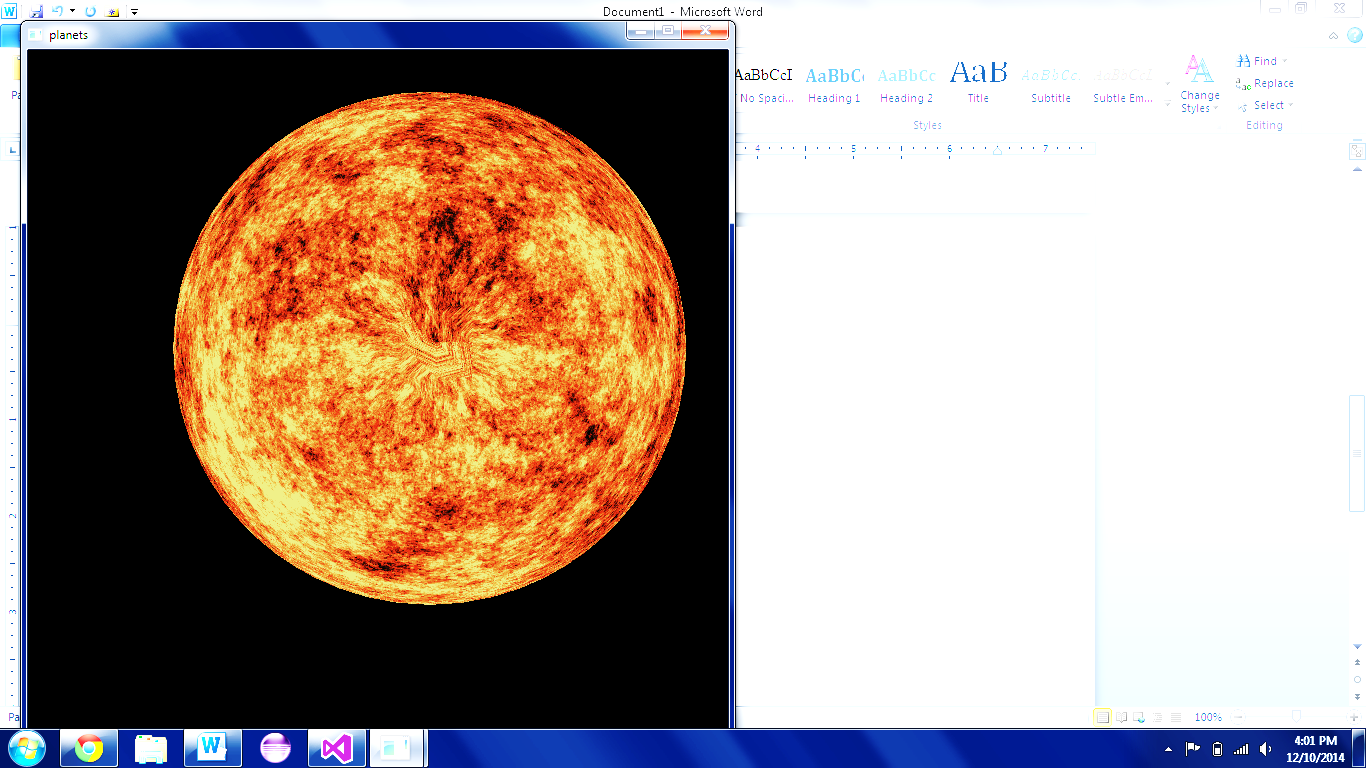
This was an attempt at an earthlike texture, and was among the first I had made. Based on the height value, there are oceanic colors, as well as those corresponding to land and mountains. It was created with equal biases for all directions.



This demonstrates the method of weighting the random walk in a horizontal direction, such that it creates the look of latitudinal winds. The gradient here will, at larger height values (with more time steps), produce a “frozen” looking planet.







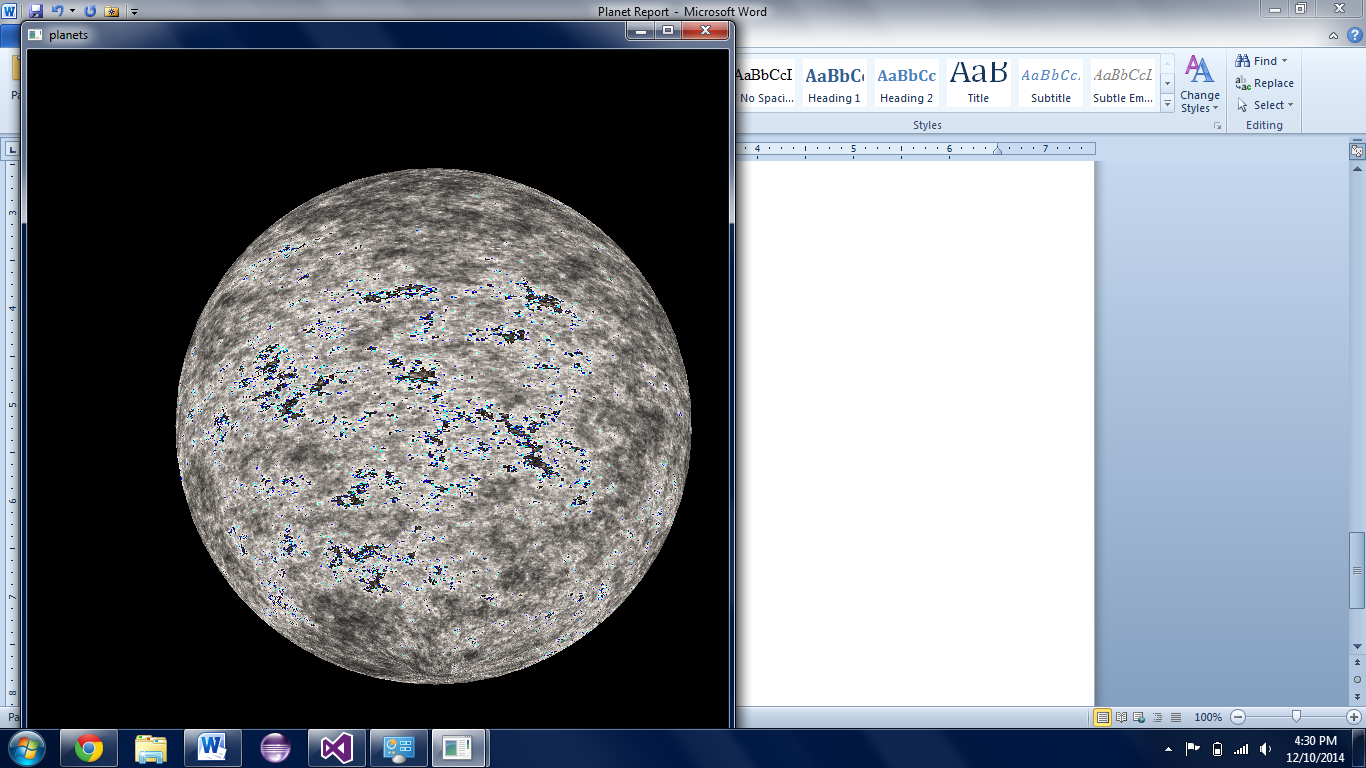
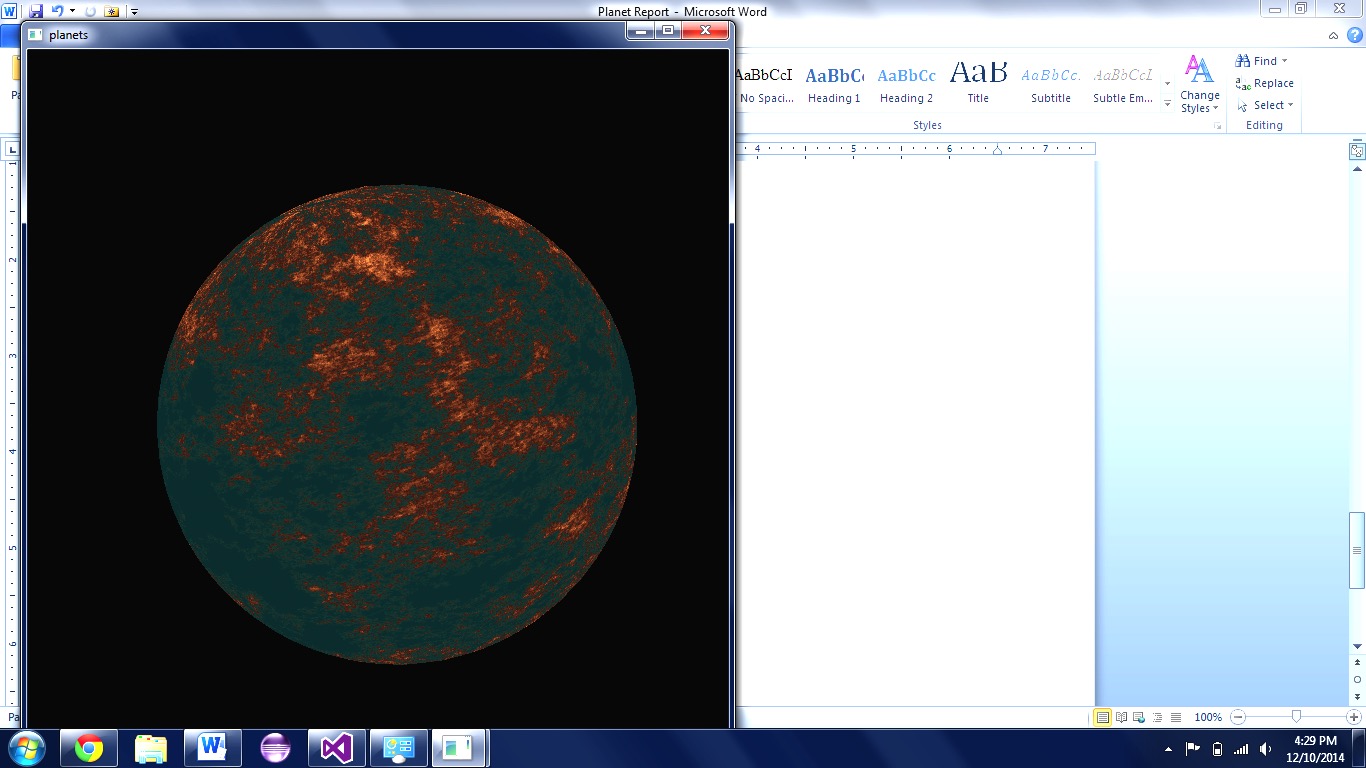
Here is an example of texture mapping to the cube bulged to the resolution of the first (top-left) image on the previous page. As can be seen, the texture wrap around in the bottom right corner is obvious.

(These images, for demonstration, are viewed from the top of the cube/sphere, and have had their brightness increased)

Now, with the cube’s resolution increased, the artifact is visibly smaller, but still noticeable near the center of the image.

At a nice (and still interactive) resolution, the artifact has shrunk and contracted even closer to the center of the image. There is still a “pull” effect near the poles, but it is minimal when viewed from a user’s perspective. As an added bonus, the polygon silhouette has almost become invisible.

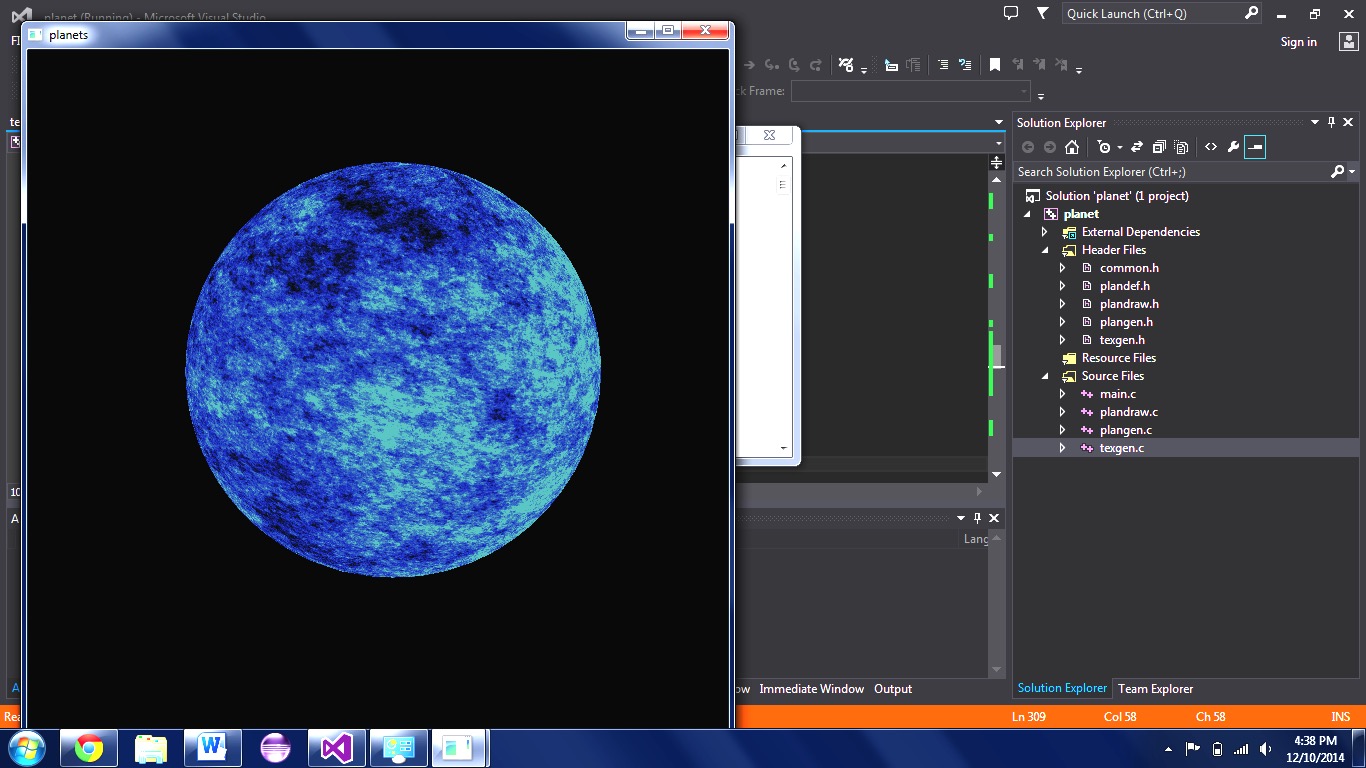
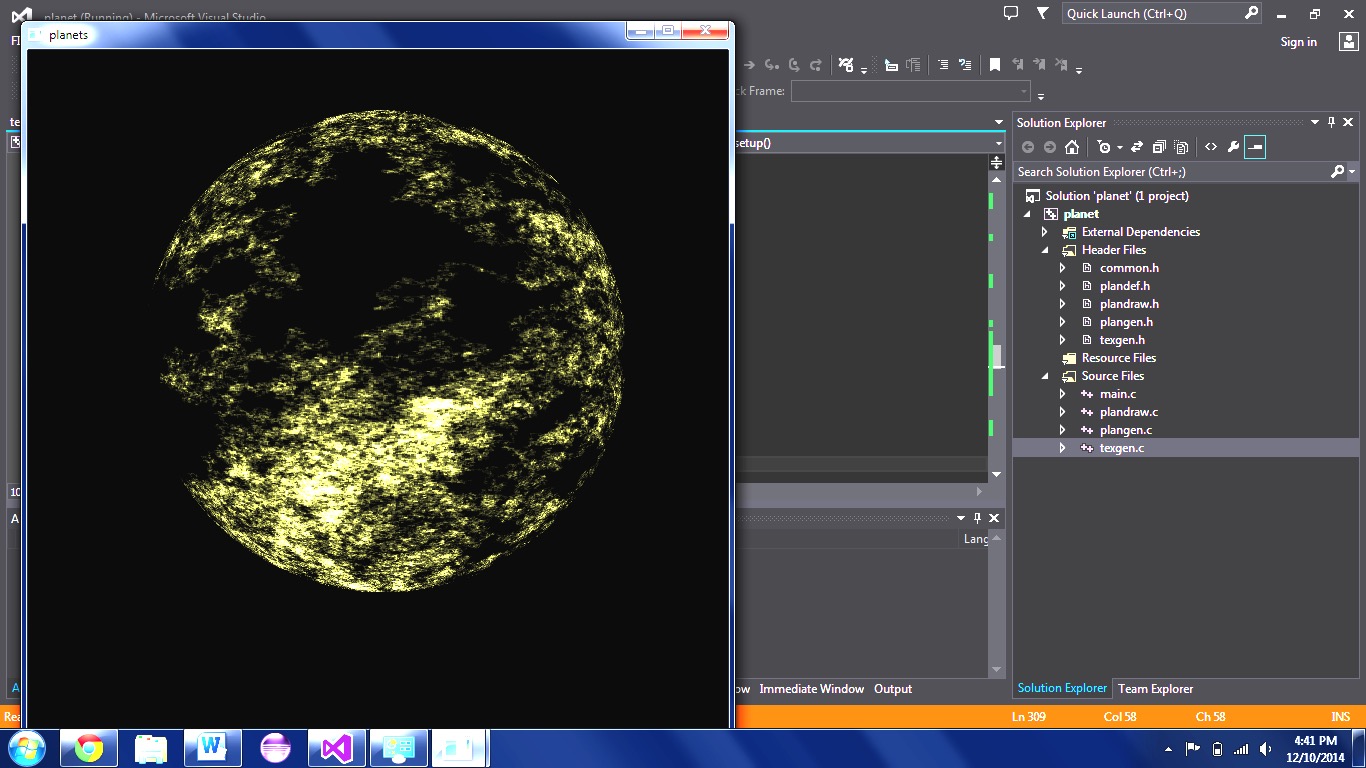
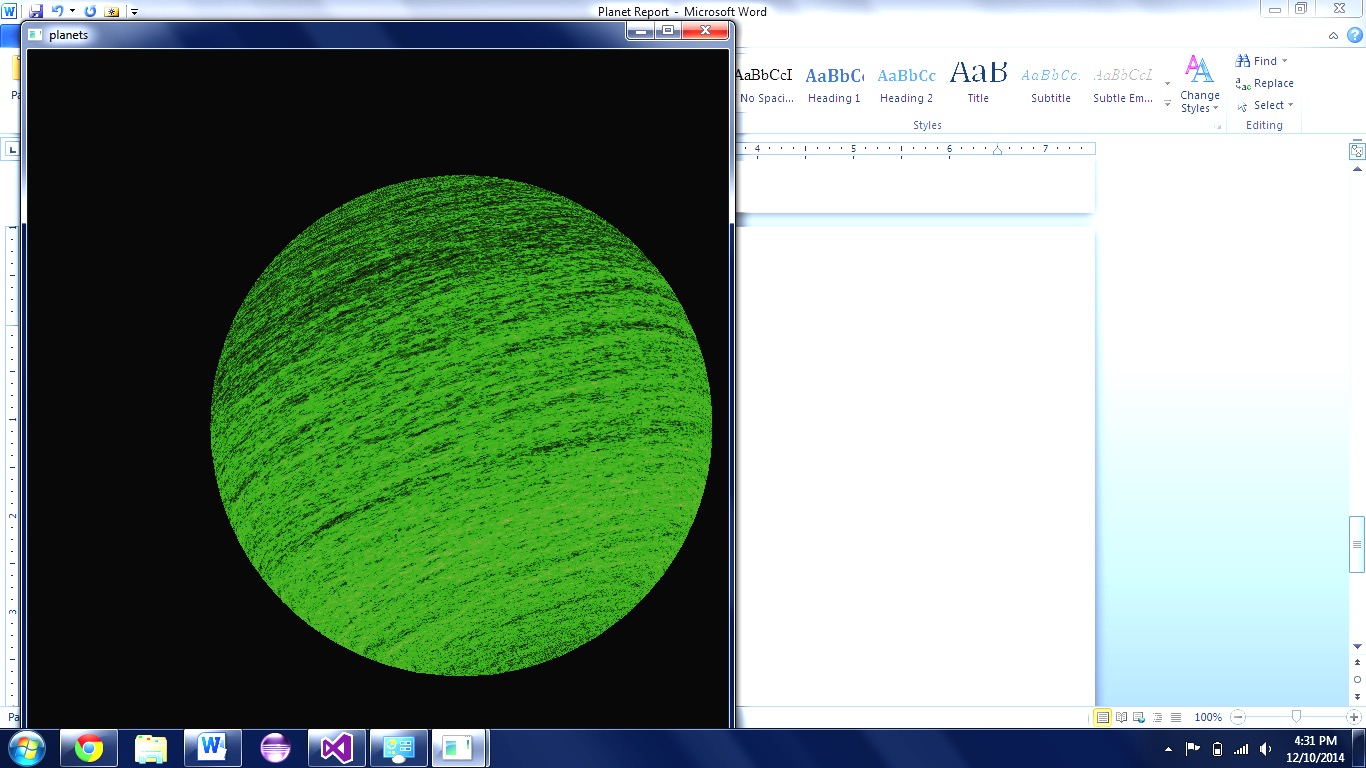
On the next page are more images and some comments on parameterizing the functions.



Above are instances of two more gradients. To the left, the appearance of multiple “islands” is evoked through the use of multiple walkers whose starting positions were randomly determined. With only one walker, though multiple landmasses may appear, they will be connected by a little trail, which may or may not be desired. When more are added, landmasses may form without necessarily being strung together.

The image on the right is an example of a poorly defined gradient function. There is a chance for the gradient to increase the color channels so much that they overflow into other fields in memory, thus resetting some channels back to zero while incrementing others located adjacently in memory. To remedy this checks in the gradient functions are utilized. These obviously require more computations or more memory (to remember the computations and apply them multiple times), but they completely eliminate the overflow.

Below are other examples of correctly checked textures.



1. **Lighting**

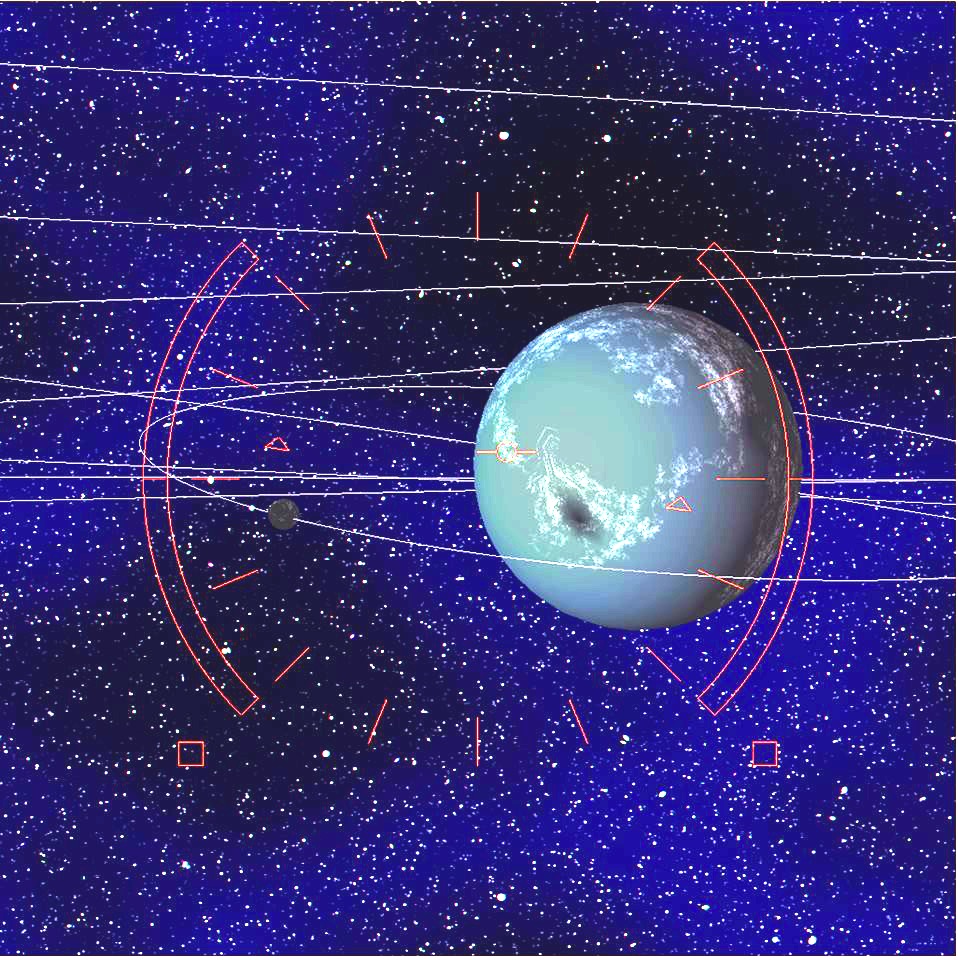
In initial attempts at attempting to light the scene, shadow mapping techniques using the ARB\_depth\_texture to hold the depth buffer and make a new texture for shadows, incorporating the standard OpenGL lighting model were used for small segments of example code. However, when trying to modify much of the example code to fit our own already existing classes and draw functions, we were met with various incompatibility issues and problems with particular hardware. On top of this, the code from our example was difficult to work with and finicky when dealing with an omnidirectional point light. As a result, the idea of finding a solution to shadows in hardware was scrapped early on.

Our next major attempt was to implement our own solution using a generic ray tracing algorithm using much of the information that was taught in class. We had not done any ray tracing in code previously, so taking the concepts from class and applying them directly proved to be a difficult challenge, but certainly not one we could not overcome. In normal ray tracing, and in much of the commented out code of the files which contribute to the ray tracing in the project, a ray is taken from the eye position to each pixel on the screen, and when this ray intersected with a planet, it was marked down to be tested for shading. In order to avoid excess work, it was going to be used in combination with regular OpenGL lighting, and instead of calculating shading based on the Phong illumination model at every point, only testing for shadows would be done at the points intersected with the ray. This procedure unfortunately had to be scrapped as well. The interactive nature of our project did not allow for something that could potentially take minutes to calculate, and even if the precision of the shadows was impressive, the simulation took priority.

What did come of this second experiment was the discovery of many bugs with our data structures, and while this ray tracing code was not specifically used, it unexpectedly served as a great debug tool.

From the previous endeavor, OpenGL’s basic point light was already in place, providing shading for all the planets. With the only thing left being the shadows, a simpler solution was put in place to generate the proper shading for satellites when behind planets and to put the shadows of the planet’s satellites on the planet’s surface. Instead of taking rays pixel-by-pixel and intersecting them, the solution takes advantage of the way that OpenGL handles the way it shades with relation to the normal of a plane. All of our spheres were made up of polygons, so during the most important draw step that calculated the normal for each vertex which was previously used for shading, the additional parameters of the current world coordinates of the planet being drawn and the planet’s radius were passed to the “ray tracer.” While not a ray tracer in the traditional sense of the concept, this function takes in the local coordinates of the vertex being drawn, converts them to world coordinates, and makes a ray from the sun’s location at (0,0,0) to the vertex. It then takes the ray and sees if it intersects with any of the other planets or satellites in the solar system. If it does, it breaks out of the loop early, and at that vertex sets the normal to the plane as the ray, making it as though it were shaded like the back of the planet. If it does not intersect, it keeps sets the normal as the actual normal to the vertex.

This solution is still not particularly efficient, but is much better than the hundreds of thousands of calculations being done with the previous full ray tracer. The other thing that is worth noting about this solution is that the accuracy of the shadows directly reflects the resolution of the planets being drawn. On a more powerful machine, this algorithm could certainly be capable of rendering accurate shadows. While perhaps not the absolute best implementation, it is proof of the concept that basic shadows can be generated using much of what OpenGL gives to the programmer. Included in the project is the final version of this “ray tracer” as well as the old original ray tracer code that could potentially still be used if adapted and fixed at a later date. The program does not automatically run with the ray tracer. It is recommended that the scene is first frozen, and then the ray tracer is turned on as to avoid a serious slowdown of the program. Below are screenshots showing some of the shadows made.



1. **Conclusion**

This project was a very fulfilling exercise in all of the concepts covered throughout the course. Much of the planetary and camera motion used the same core linear algebra concepts for 3D transformations covered throughout the course. We also experimented and succeeded with combining 2D and 3D projection and matrices. The texture mapping allowed us to explore both textures themselves and how color spaces worked, since textures were essentially algorithmically generated bitmap images while the program was running. Lighting and shadows then provided life to the system as planets moved around and moons orbited around those planets.