CS 1566 Final Project Report:

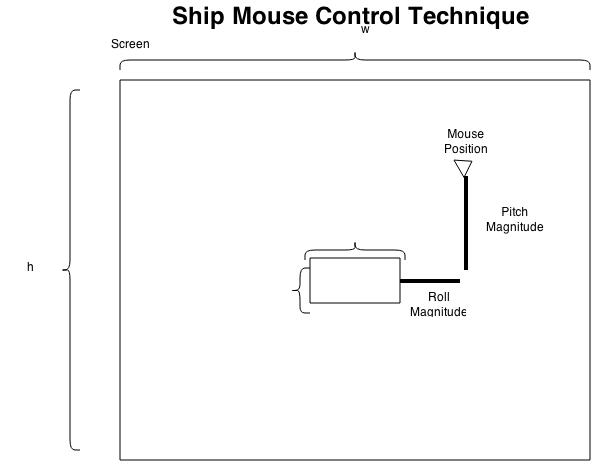
1. **Introduction**

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.

The spaceship has a 2D reticle overlaid onto the 3D view, which gives metrics about orientation and speed to the user.

1. **Camera and Skybox**

Implementation Technique



**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 situation on the planet’s moon.

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.

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 I instead calculate areas using trigonometry. This introduces some error as I end up underestimating 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, 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 raytracing.

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 I had written for the ship rotations were useful in calculating the positions and orbits for the planet.

1. **Texturing and Planet Appearance (Jonathan Albert)**

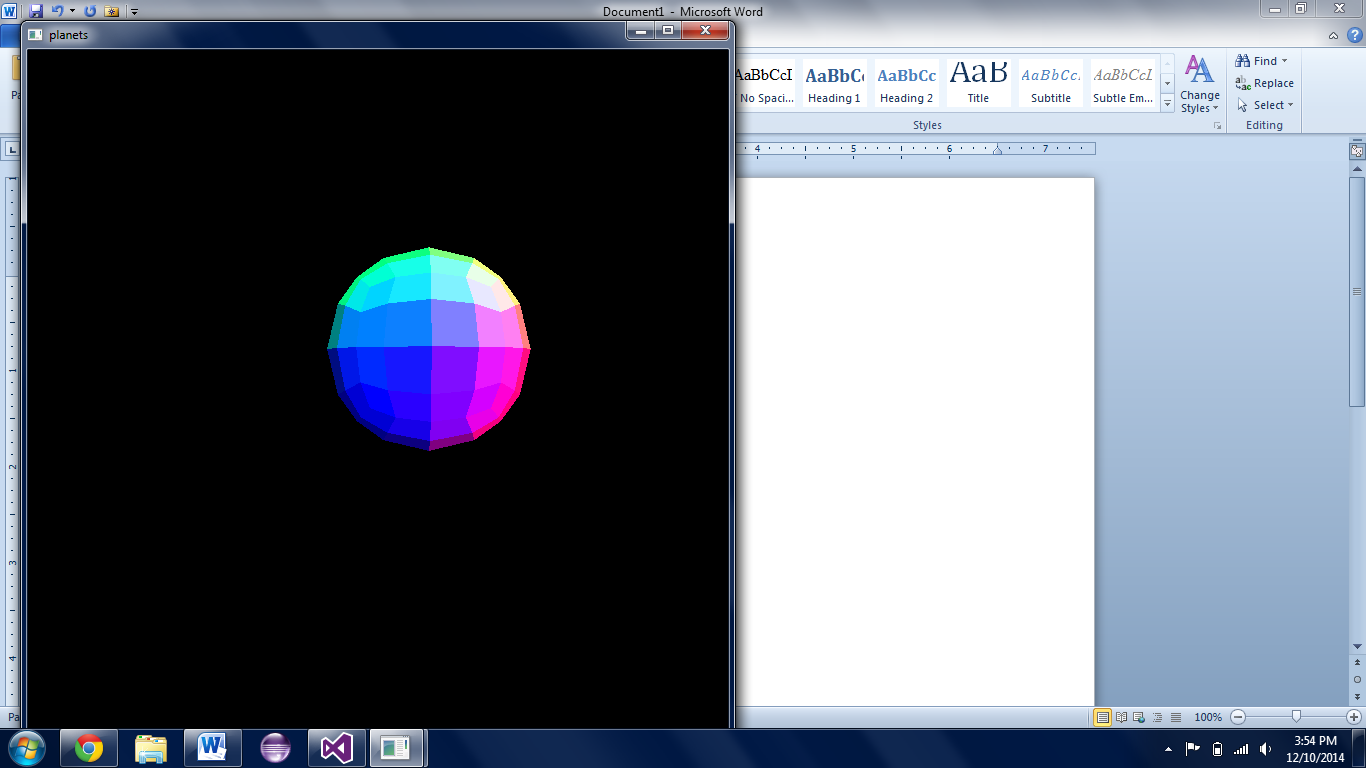
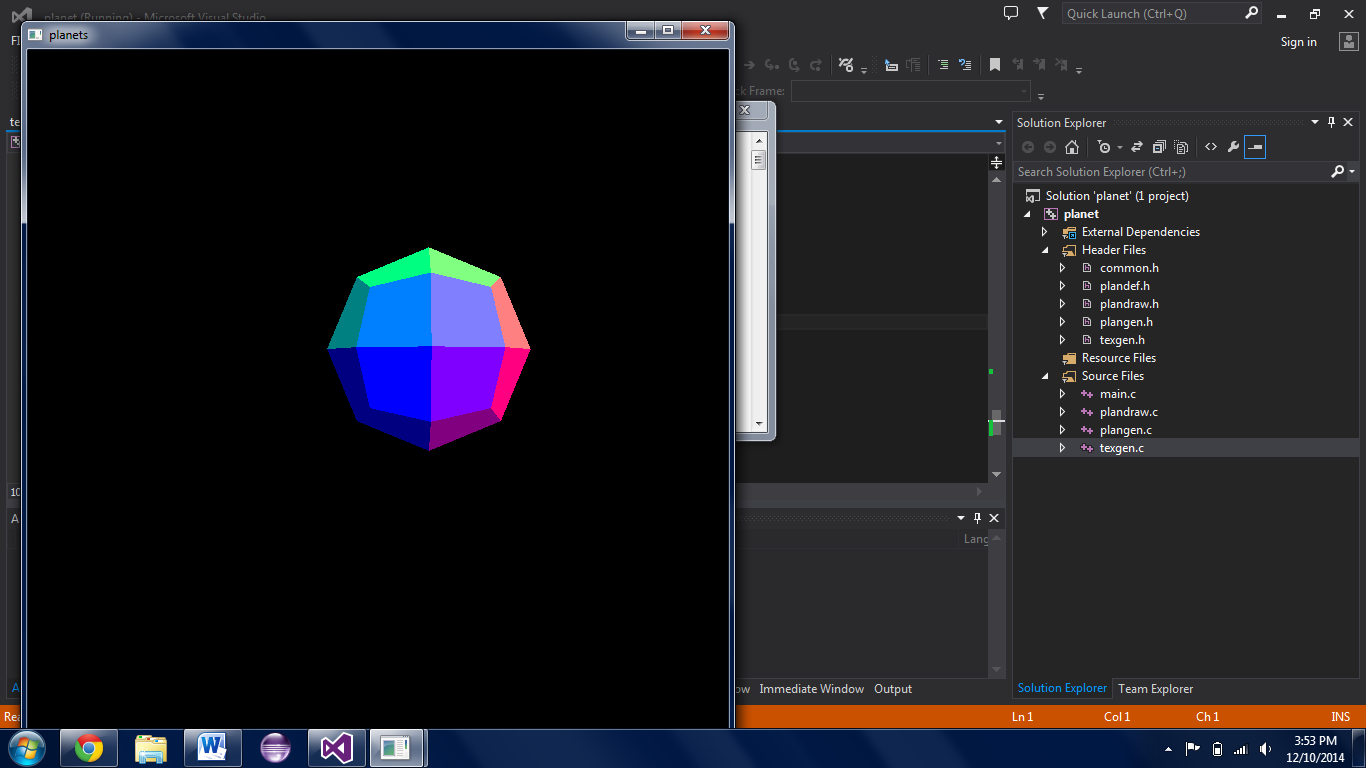
**Introduction --** Noise functions and procedural worlds have piqued my interest for some time now; my being able to work on a project where that is the goal is very nice. Obviously, without planetary bodies, there would be no universe to depict. Since I may delve deeper into this topic in the future, I figured I should get my hands dirty with these functions, so that I would have a basis for further exploration.

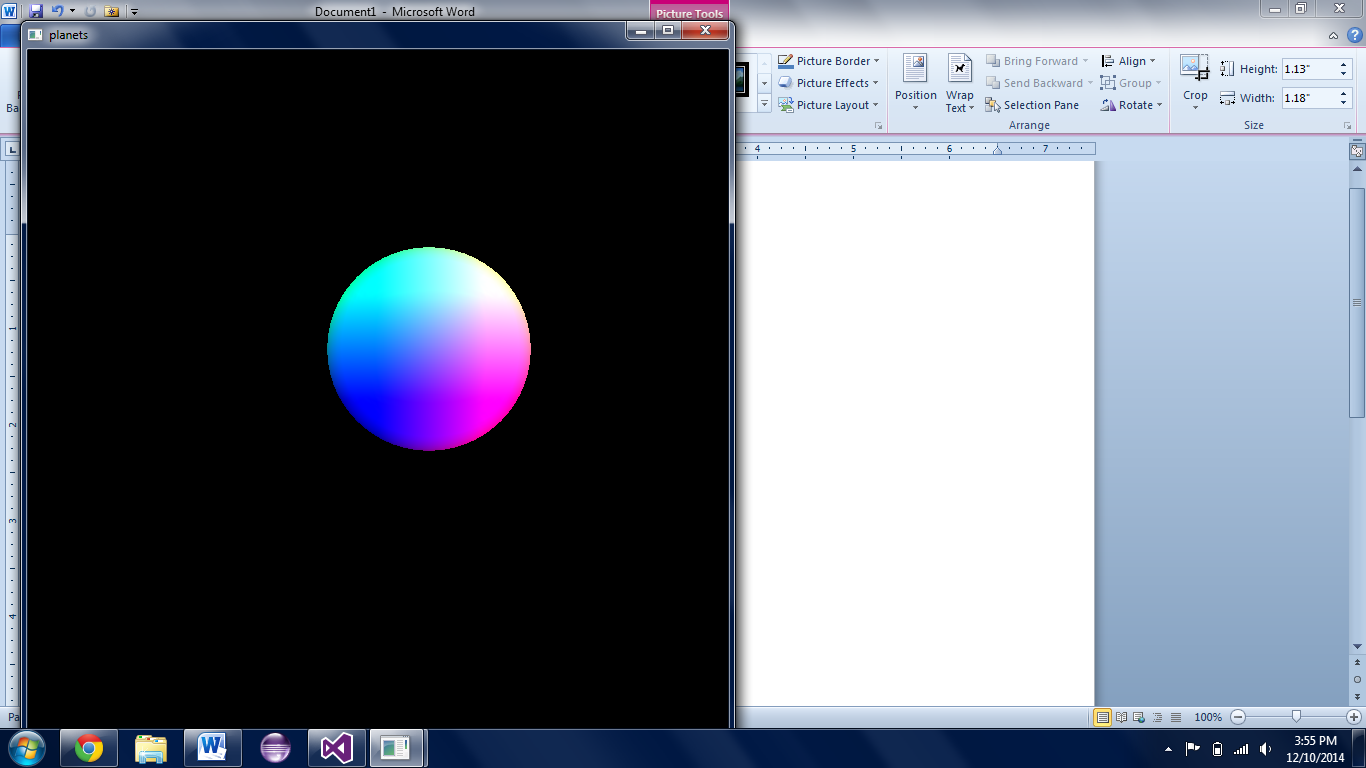
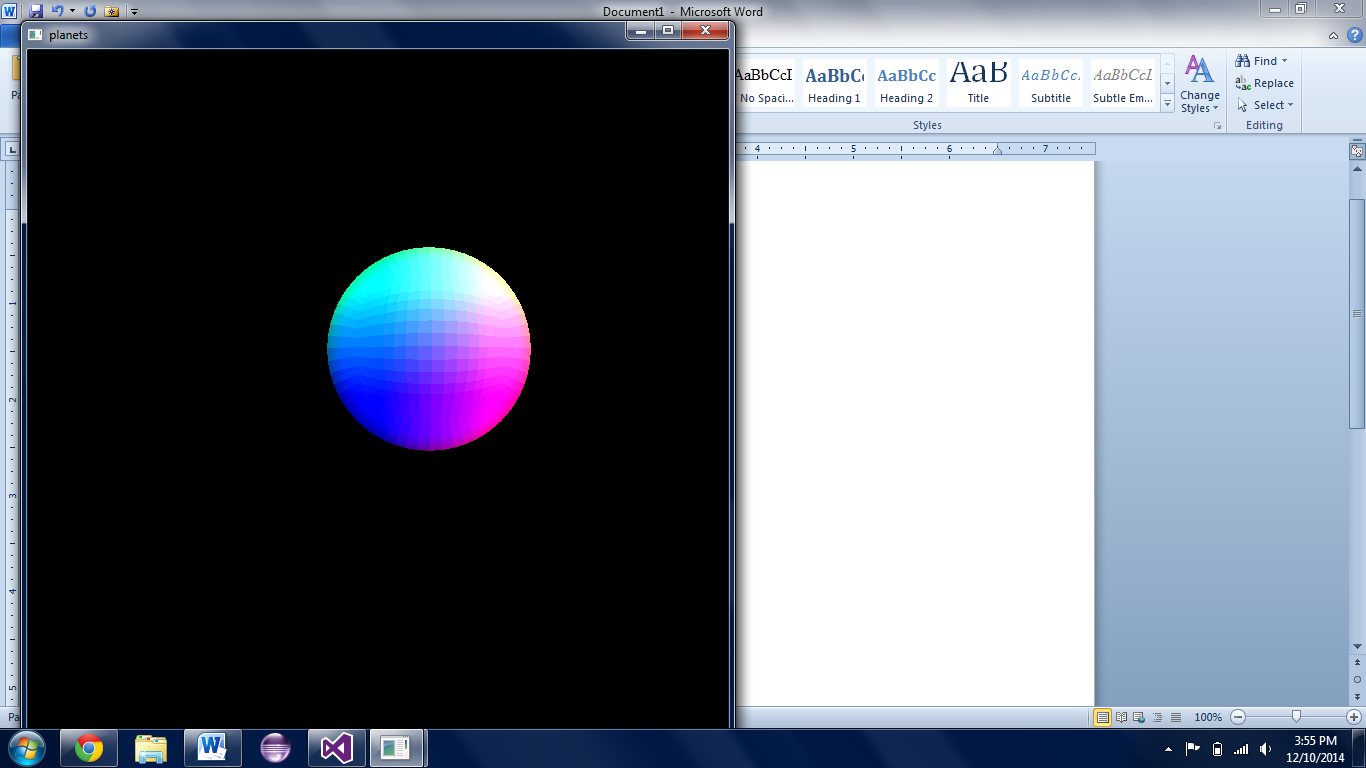
**Approach (Method) --** The main challenge for my portion of the project was math, and secondarily finding good RGB sets to use in the textures. For texturing the sphere apart from glu’s Quadratic wizardry, I had to define my own sphere drawing function through the help of some moderately complex mathematics. Effectively, I bulge the mini-quads of a cube outward to a specified radius, thus emulating a sphere (see Results). From there, I initially thought to just map the texture on each “side” of the cube sphere, since the textures themselves were periodic and thus could loop just fine (more on that soon). However, that 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. Then, I turned to the class notes, and I converted the sphere’s XYZ points to the texture’s UV coordinates like I ought to have done.

Still, there was an obvious seam on one side of every planet, which was due to the fact that sine functions are periodic and, further, that I didn’t have an infinite resolution on the planet. 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, producing an unseemly belt. After some Google searching and tweaks, I managed to modify the conversion function to check if it was near the texture’s border. Even that was 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, it will work just fine.

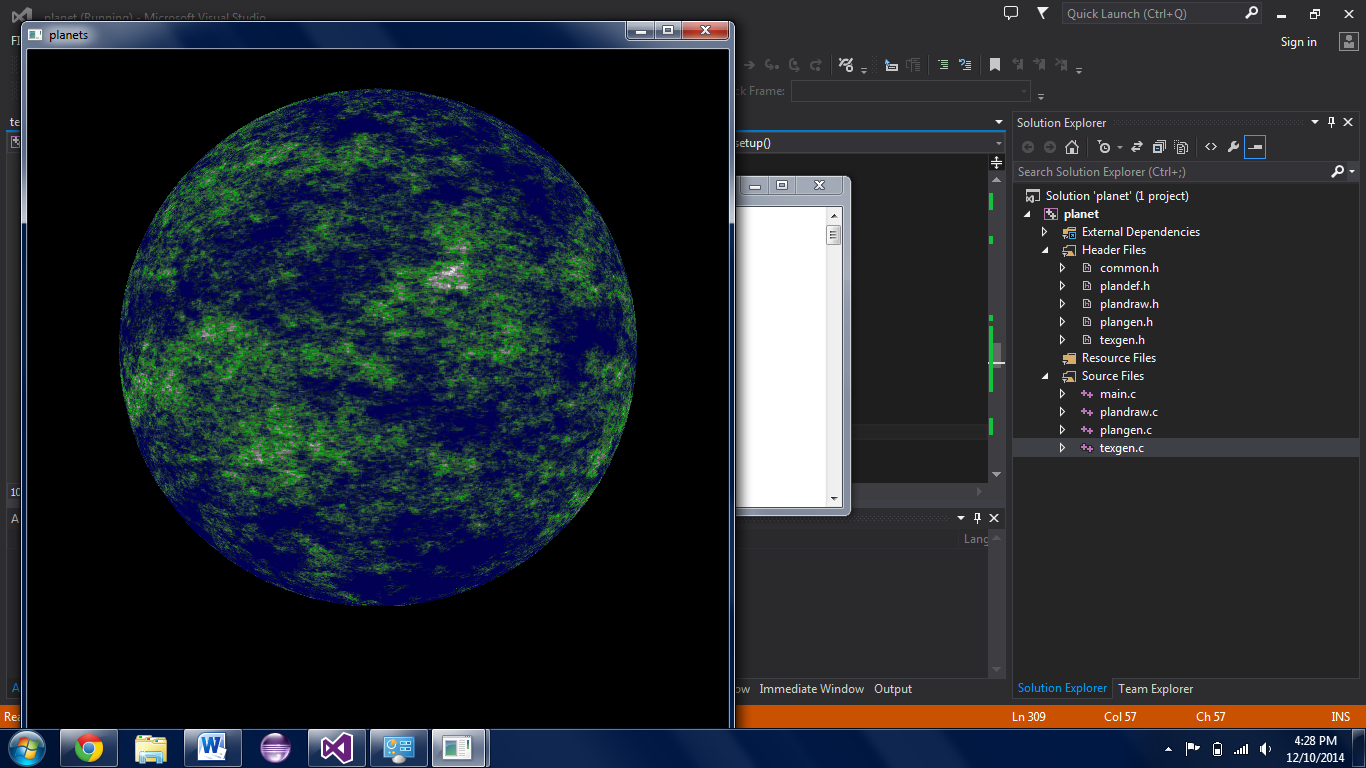
Generating the textures, on the other hand, was another hurdle. After much sifting through noise algorithms and boggling my mind with trying to learn them, I completely pushed those to the side in favor of a method I’ve learned from my freshman year: Brownian motion. I did this partially because I didn’t want to copy someone else’s source code, and because I 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.” I make the borders 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, I wrote a function 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.

**Results and Discussions --**

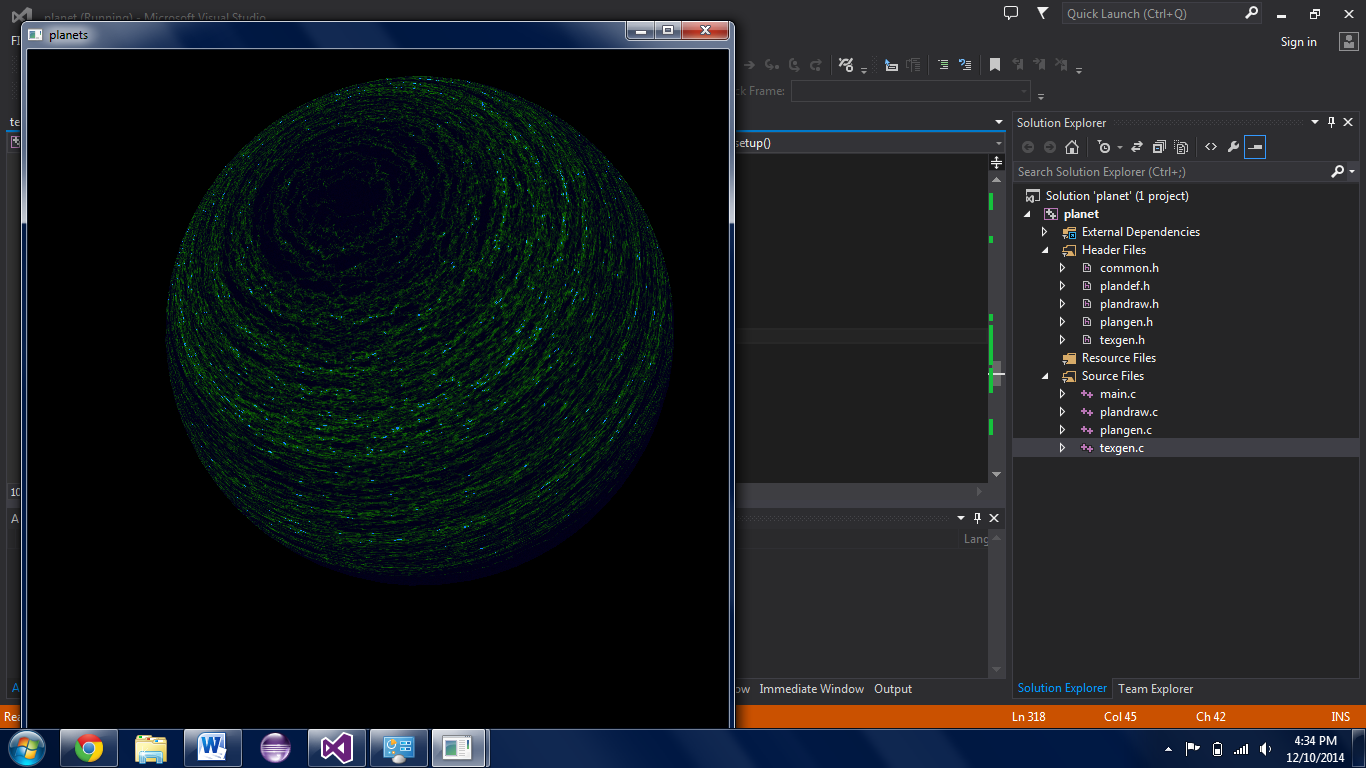




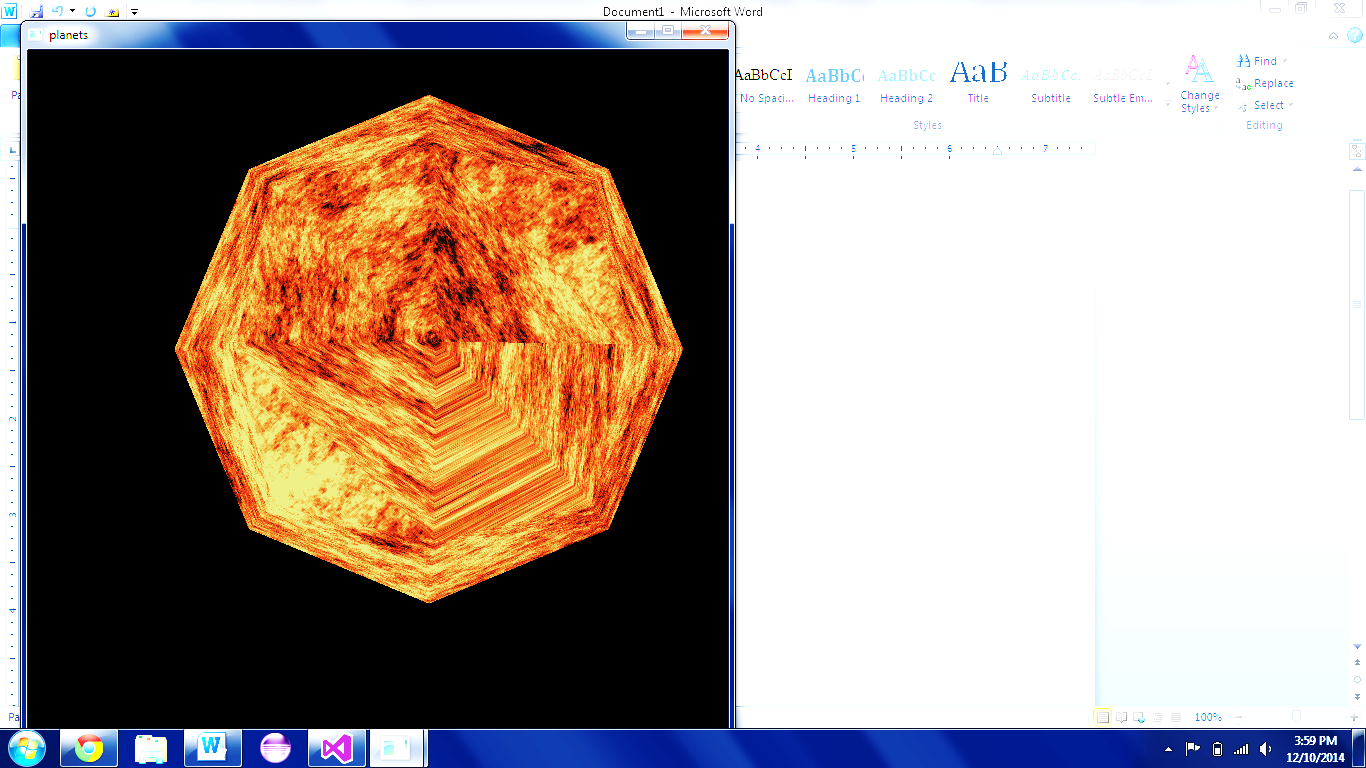
Above is an example of what the different cube resolutions look like when bulged. The colors correspond to each quad’s top left corner’s XYZ value, where X= R, Y = G, etc. With higher resolutions, the cube looks increasingly more like an actual sphere. At very high resolution, however, it becomes too 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 to the right; a brief discussion of the difficulties in texture mapping is on the following page.



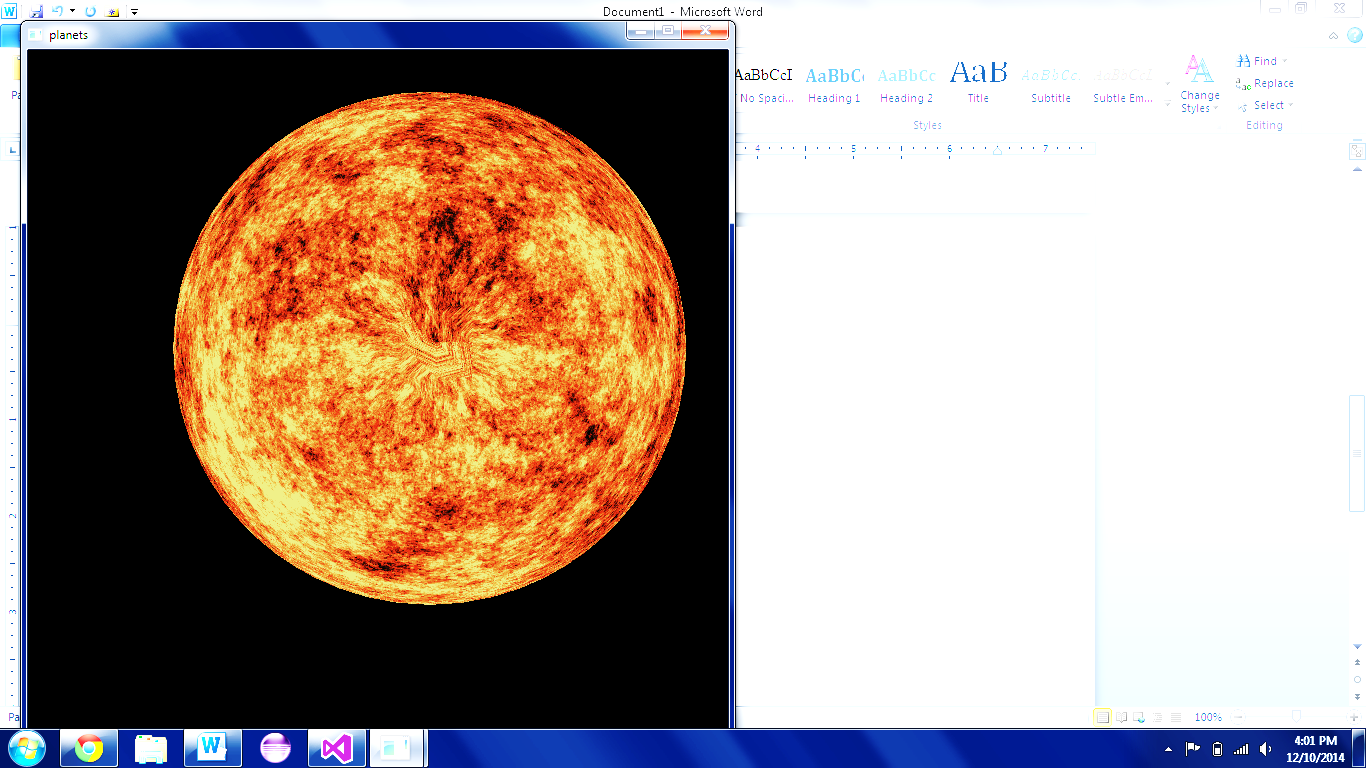
This was an attempt at an earthlike texture, and was among the first I had made. Based on the height value, there are oceanic colors, as well as those corresponding to land and mountains. It was created with equal biases for all directions.



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.







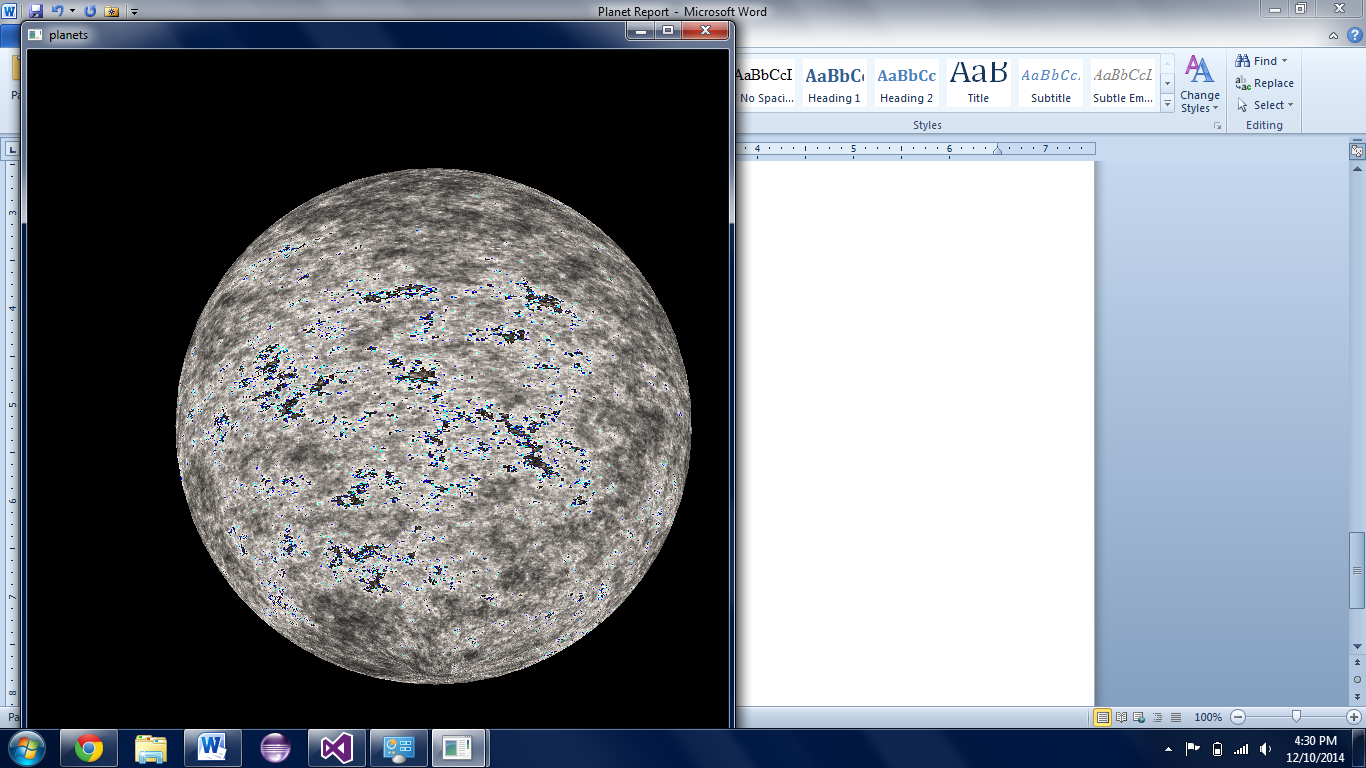
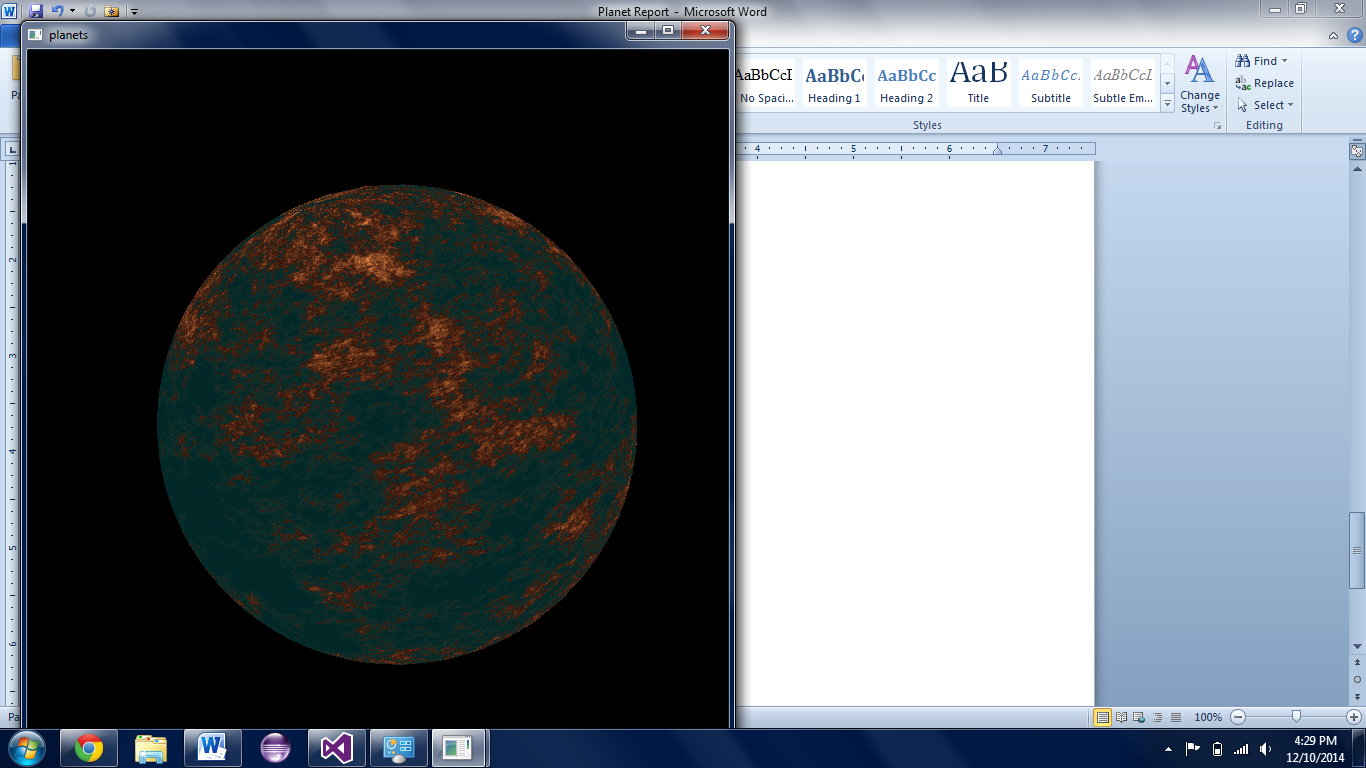
Here is an example of texture mapping to the cube bulged to the resolution of the first (top-left) image on the previous page. As can be seen, the texture wrap around in the bottom right corner is obvious.

(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.

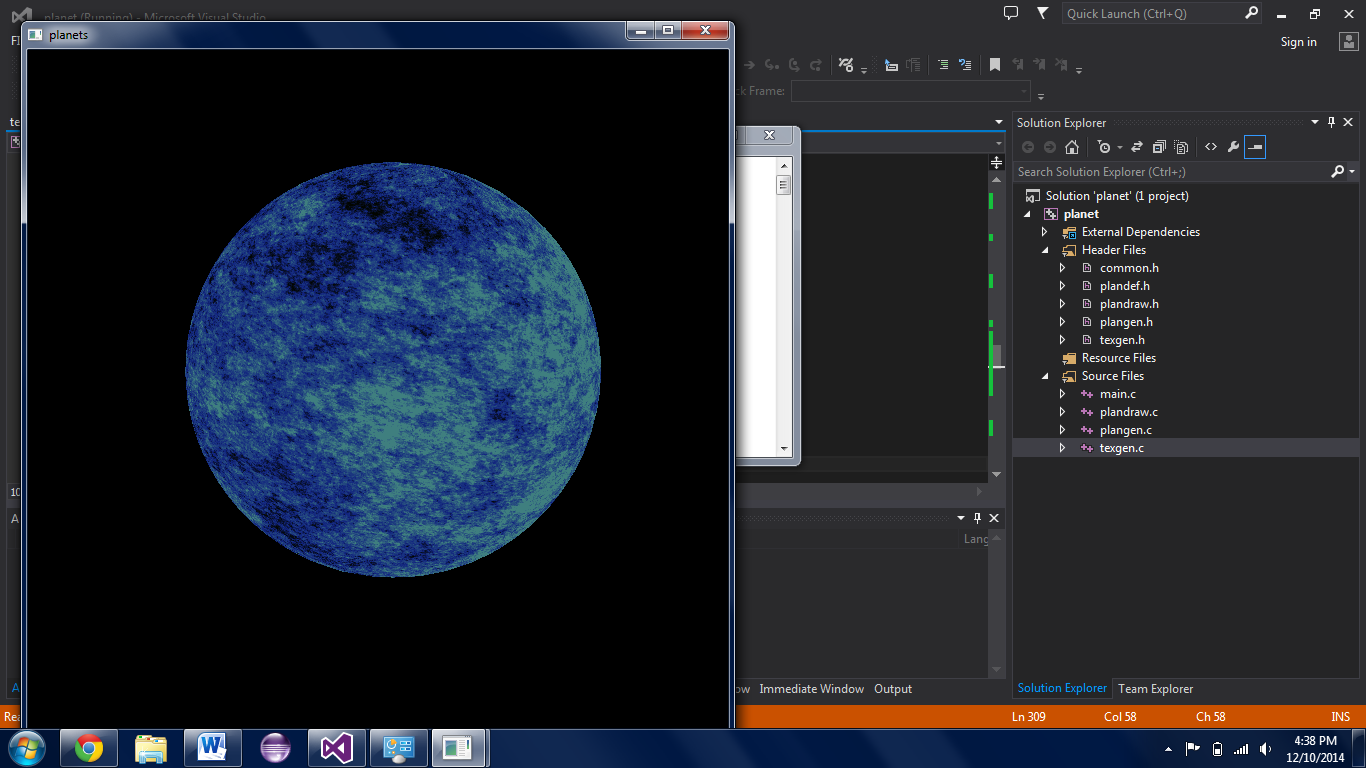
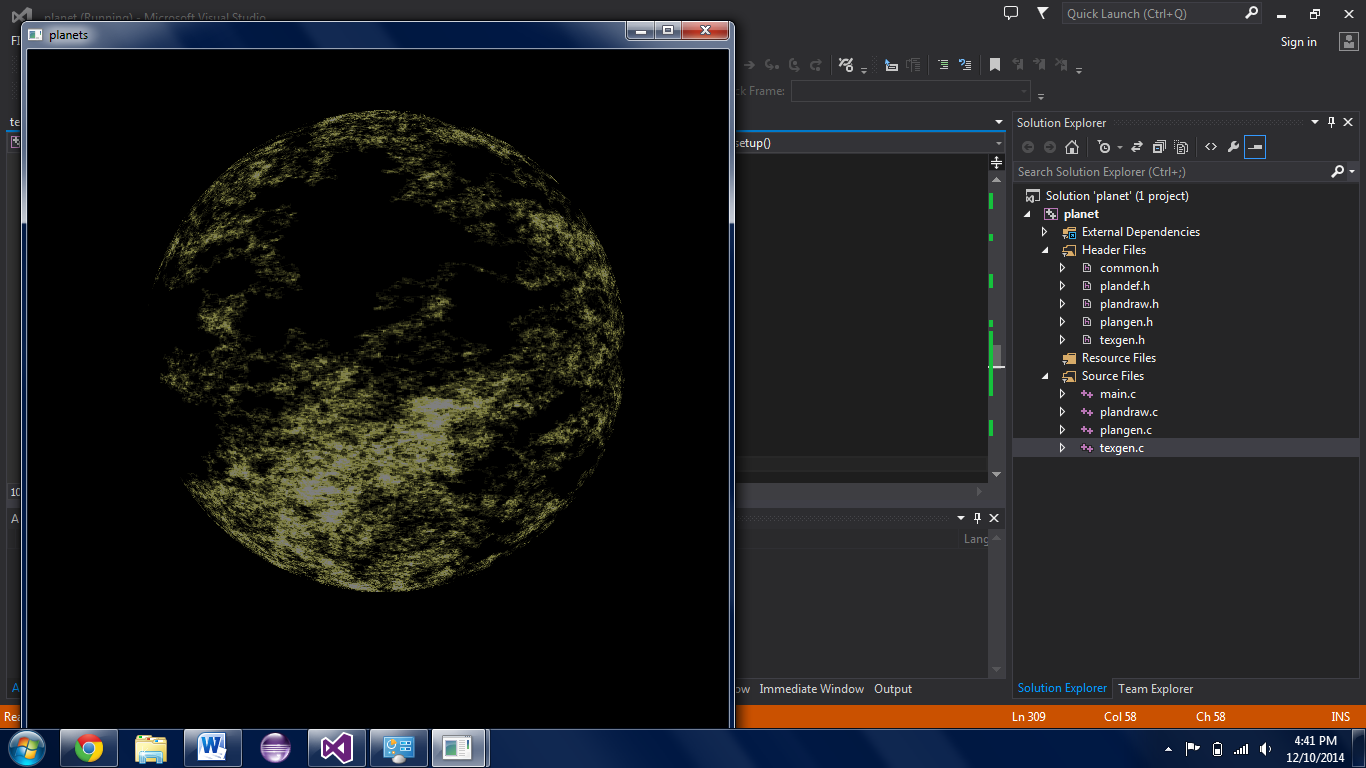
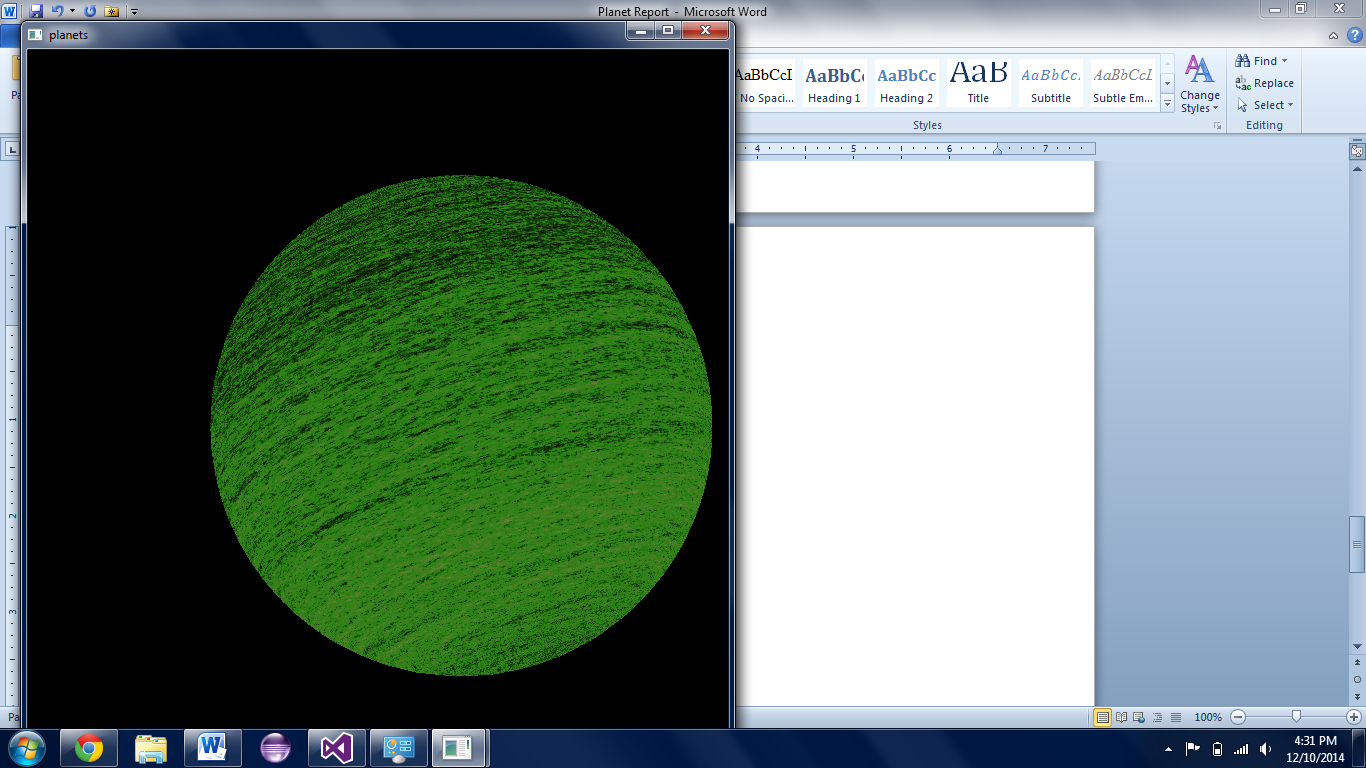
On the next page are more images and some comments on parameterizing the functions.



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, I included checks in the gradient functions. 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.



**Conclusion and Future Work --** I learned from this project firsthand what we’ve been learning all year: graphics takes a lot of math to work, let alone look good. If I had more time, I’d throw more math at the gradient functions to create more realistic textures, and perhaps consider real-time texture generation instead of one large, up-front cost for every texture seen in the program. I’d also like to understand Worley noise and possibly allow for “landing” on the planet with a quadtree-type implementation, but that is beyond the scope of a month’s work.

1. **Lighting**