1203086 Craig Jeffrey
Procedural Methods
Procedural Gas Giant with GPU Simplex Noise

With Blur and Distortion Post Processing

Table of Contents

1. Project Overview

1.1 Background

To begin with I was exploring an idea completely different idea. I was attempting to implement fractal lightning by recursive midpoint displacement along line segments in a geometry shader. However due to difficulty with implementing a recursive pattern without dynamic containers I decided to move away from this. Retrospectively I could have created the lightning mesh on the CPU and handed it to the shader since recursion is innately very difficult to parallelise - especially in this case where each new line segment is a result of a previous calculation.

Now the idea I ended up implementing is far more GPU friendly. As demonstrated in the Seeds of Andromeda blog post (Arnold, B. 2015)[1], it was possible to implement a gas giant shader using gradient noise for 1D texture manipulation. In addition to this I also decided to implement two post processing effects. The first being gaussian blur and the second being screen space texture distortion using basic pseudo-random (digitalerr0r 2009)[2].

1.2 Program Features

The program features several procedural methods. I will list these below:

- Implementation of an Icosphere by recursively midpoint displacing the faces of an Icosahedron. The code I used was written by Frank McCoy (2015)[3].
- Implementation of 3D Simplex Noise in a shader. The specific implementation of noise is used property of Ashima Arts (2013)[4]. I ported it from GLSL to HLSL.
- Use of 3D Simplex Noise to deform a 1D texture on an Icosphere to simulate an evolving gas giant with threshold noise samples being used to simulate storms.
- Implementation of Gaussian Blur post processing. This is the implementation from the rastertek DirectX 11 blur tutorial (Rastertek)[5].
- Implementation of screen space texture distortion using pseudo-random noise. This is based off of digitalerr0rs implementation (2009)[2].
- Implementation of a camera to navigate the scene. This is the camera from the rastertek DirectX 11 tutorial (Rastertek)[6].
- Keybinds to reload the gas giant and blur shaders for experimenting with small changes quickly (R and Z respectively).
- Keybinds to increase and decrease timescale for the gas giant and distortion intensity for the distortion post process (Arrow keys - see readme).

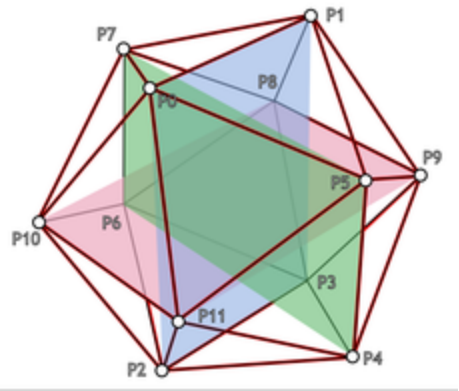
2. Implementation

This section will describe the implementation of the program features.

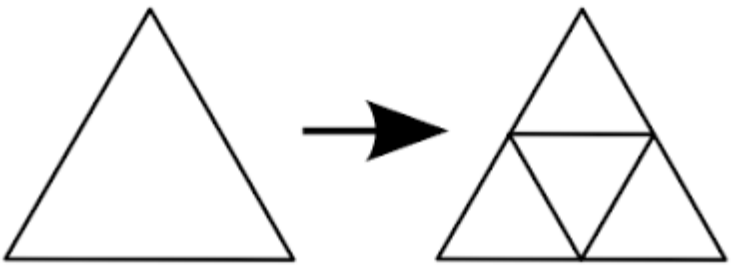
2.1 Icosphere Generation

An icosphere is an amalgamated term that describes a specific way to construct sphere geometry by recursively refining the faces of an icosahedron - hence the ico prefix - through midpoint displacement and vector normalization to create spherical looking geometry. To help me understand the process better I researched Andreas Kahler's blog (2009)[7]. In his blog he clearly explains the process, I will outline my understanding here:

- Begin with the vertices and indices of an icosahedron as below:

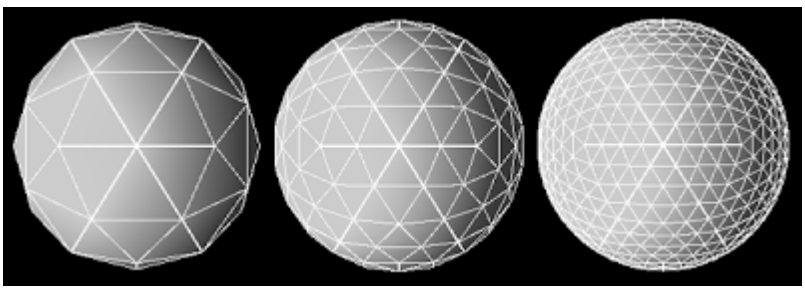


- Find the midpoint of each face and create 4 new triangles, appending them to a new container of faces as shown below:



[1]

- Recur this process on each face of the new container until the desired fidelity is reached (An icosphere can be considered a fractal where fidelity levels are equal to the octaves of the fractal), each recurrence refines the sphere as show below:



[1] All images are from Andreas Kahler's blog (2009)[7]

2.2 Gas Giant Shader

The gas giant shader class contains a gas giant vertex and pixel shader. The vertex shader is very basic, most of the work being done in the pixel shader. The pixel shader contains an implementation of 3D simplex noise[4]. The implementation of noise is extended to fractal noise by Frank McCoy (2015)[3]. The fractal noise functions below take multiple parameters to customize the noise:

---

float noise(float3 position, int octaves, float frequency, float persistence),
float ridgedNoise(float3 position, int octaves, float frequency, float persistence),

---

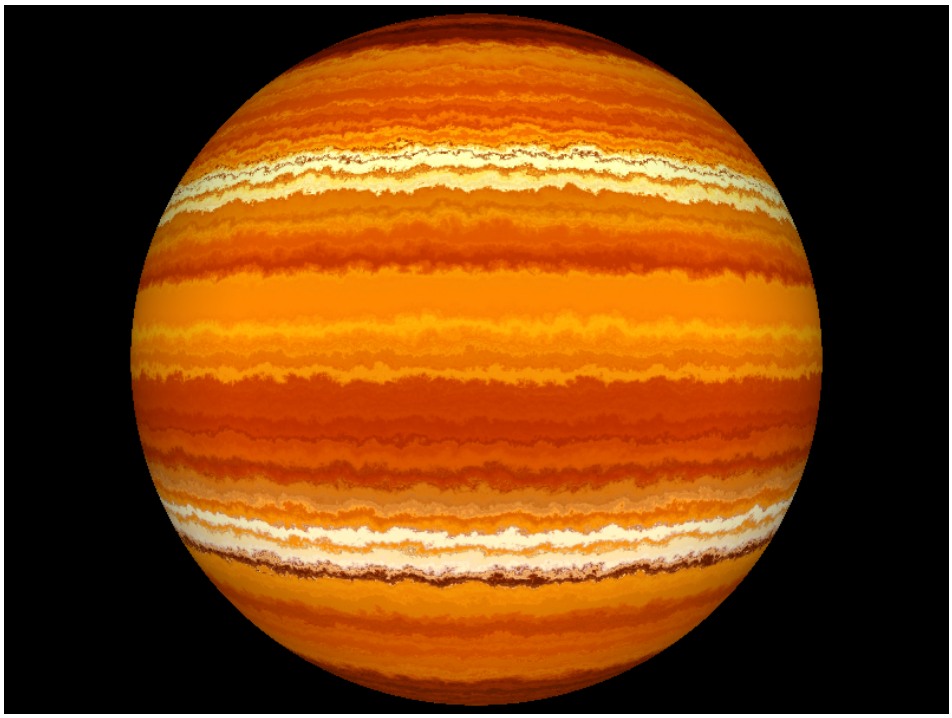
- Position - The 3D input vector for the Simplex Noise function.
- Octaves - The fidelity of the resulting noise. Higher number of octaves means more iterations to refine the quality of the noise. Of course this increases computation cost.
- Frequency - The noise wavelength. Lower value means higher wavelength. Frequency is halved each octave.
- Persistence - How much each successive octave contributes to the final noise sample. Each iteration, the current octaves amplitude is reduced by persistence.

The ridged noise function outputs noise in the range of -1 and 1 whereas noise outputs between 0 and 1.

2.2.1 Surface

To simulate the surface of a gas giant, a ridged noise sample is taken and applied as an offset to texture x coordinate. The input position for the noise function is the normal of the current pixel with time added. Using the pixels normal causes the noise functions return value to be slightly different for each pixel (slightly since we are using gradient noise). Offsetting by time causes the noise to evolve over time. The amplitude of noise is modified to an appropriate amount (-0.15 to 0.15) with -0.01 to prevent positive bias. This produces an evolving texture without storms. Note that the original implementation also used a second noise sample with the other noise function however I believe that the difference in aesthetics were negligible compared to performance cost. I couldn't discern the difference with it implemented and without. I believe that with a higher resolution viewport and texture, the results would be discernable. There is simply not enough pixels on the screen to sample the texture and the texture I used is not high enough resolution to allow for moving closer to the surface of the sphere to increase the quality of the distortion. I settled on 4 octaves for the noise function as this produces results which are varied enough to produce interesting evolution but unrefined enough for the surface to look unstable. A snapshot of the code and result is below.

```
float3 noiseinput = input.normal + float3(time, 0.0f, time);
float n2 = ridgedNoise(noiseinput, 4, 7, 0.8) * 0.015 - 0.01;
float2 texcoord = float2(input.tex.x + n, input.tex.y);
```



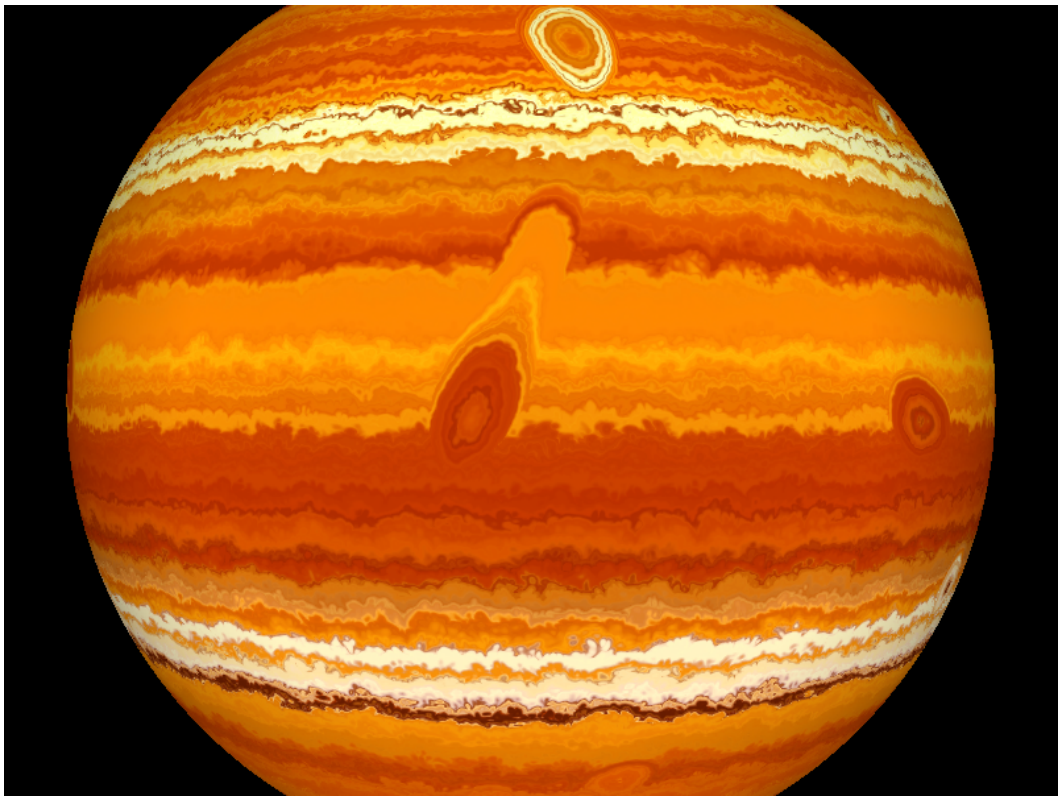
2.2.2 Storms

To produce storms, three noise samples are combined and then a fourth sample is generated with its amplitude multiplied by the threshold sample as shown below.

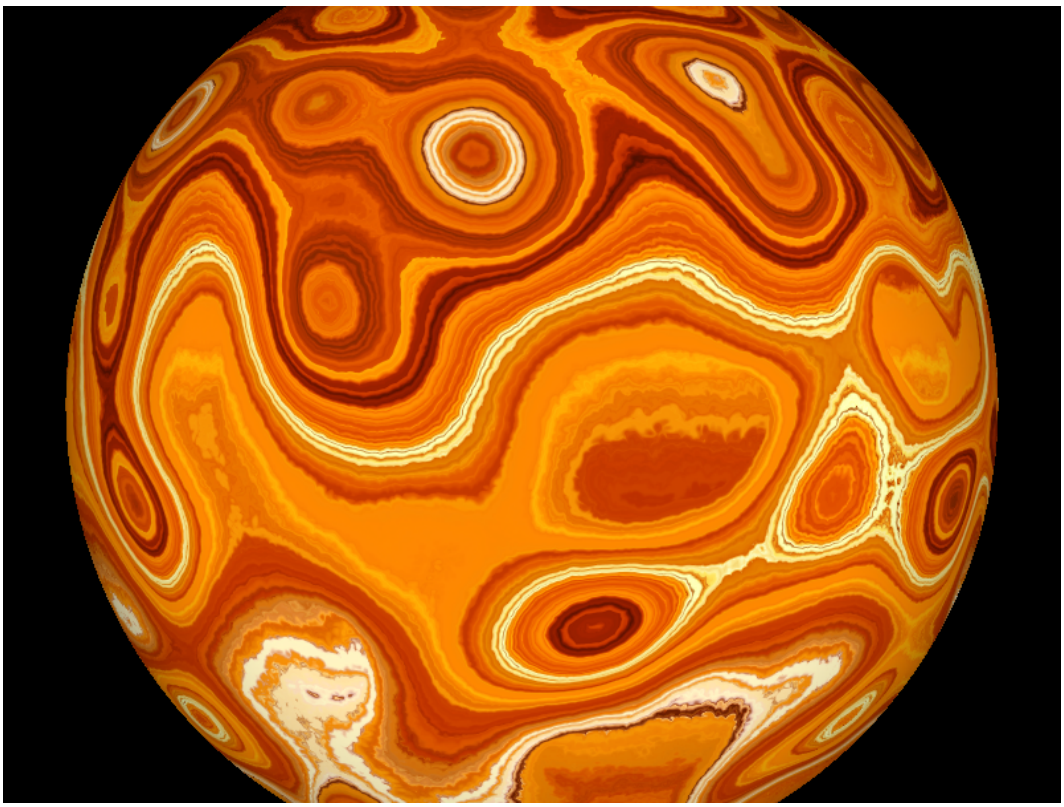
```
float s = 0.6;
float t1 = snoise(noiseinput * 2.0) - s;
float t2 = snoise((noiseinput + 800.0) * 2.0) - s;
float t3 = snoise((noiseinput + 1600.0) * 2.0) - s;
float threshold = max(t1 * t2 * t3, 0.0);
float n3 = snoise(noiseinput * 0.1) * threshold;
```

The resulting noise samples for calculating the threshold are multplied by 2 with a constant s subtracted. This modifies the range to -0.6 and 1.4. Each successive sample also has an offset so that the output is different. The threshold is clamped to above 0 by using the max function.

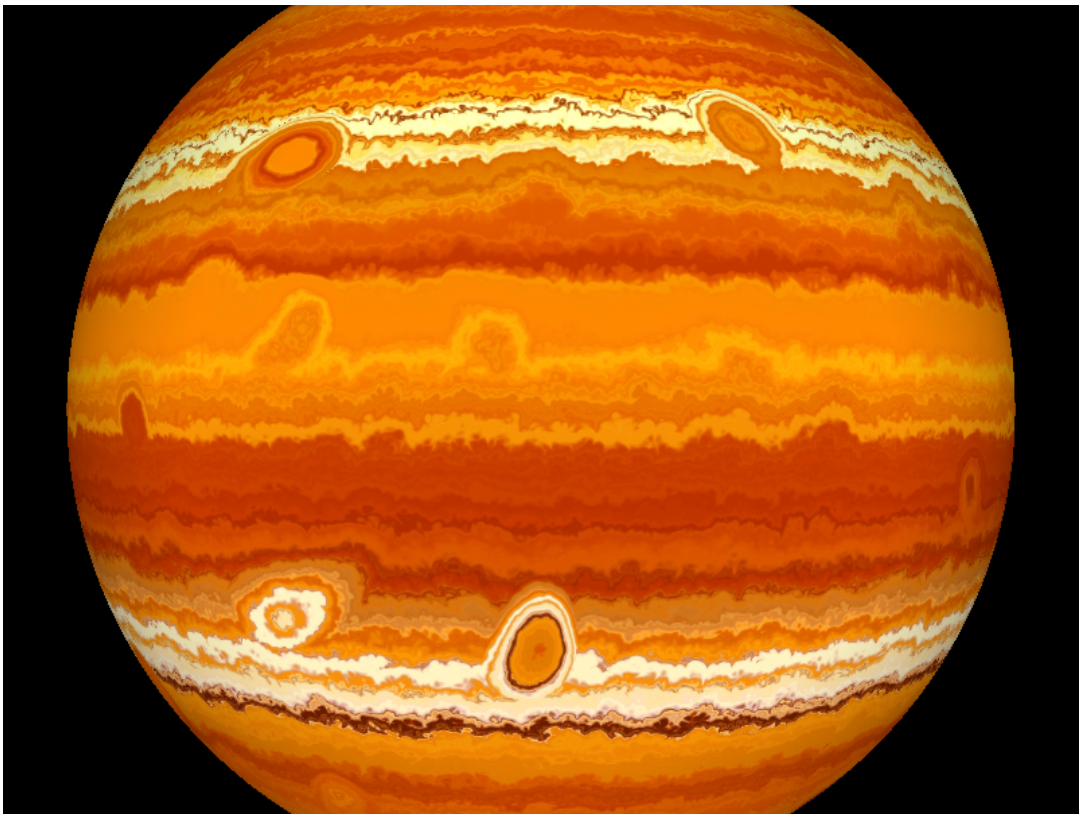
Using only the first sample as a threshold produces usable results however the shapes of the storms are occasionally odd and there are not many storms as in the example below:



Using two produces an unusable mess:



Three samples produces generally circular distortions and enough of them for the planet to be considered a gas giant:

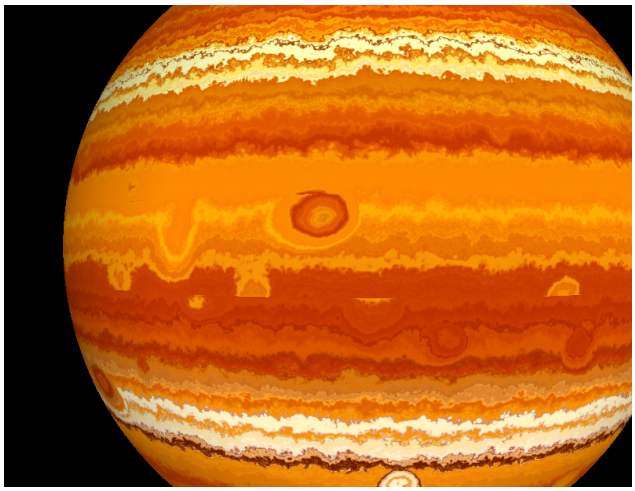


2.2.3 Experimentation

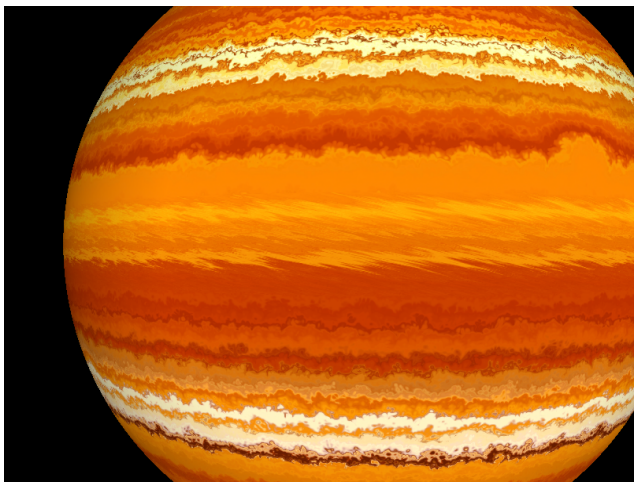
Real gas giants don't always move in one direction like this one does. They have bands of wind going in different directions. In an attempt to implement this I added code to change the direction that the noise samples evolve by manipulating time.

```
if(input.tex.x < height * 0.45f)
{
	noiseinput = input.normal + float3(time, 0.0f, time);
}
else if (input.tex.x > height * 0.55f)
{
	noiseinput = input.normal + float3(-time, 0.0f, time);
}
else
{
	noiseinput = input.normal + lerp(float3(-time, 0.0f, time), float3(time, 0.0f, time), (input.tex.x - (height * 0.45f)));
}
```

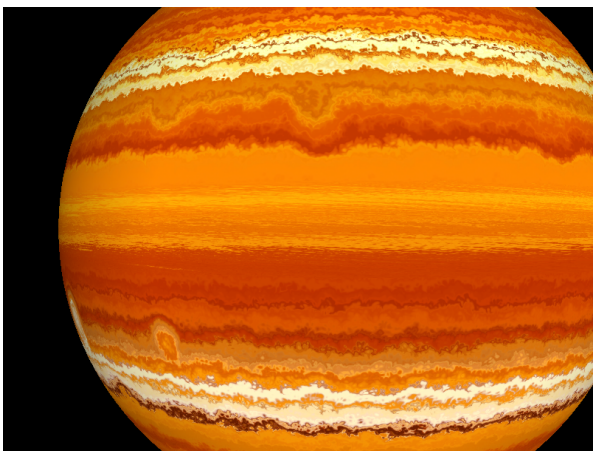
Sadly though it's not this simple. I quickly discovered that just reversing the time flow for a band on the gas giant creates an ugly seam. Then trying to blend the noise input vectors to fix this creates a progressively worsening texture between the seams caused by the difference in time. Storms also get completely broken between seams although this could probably be fixed by using a constant noise input for threshold samples instead of manipulating time for all noise inputs.



Seam very visible on bottom half.



Bad blending after a short amount of time.



Very bad blending after a moderate amount of time.

2.3 Post Processing

For the post processing scenes, the scene is rendered to a texture that is equal to the size of the screen.

2.3.1 Gaussian Blur

With the scene rendered to a full screen texture to begin with, we then render to texture again - downsampling it to a ¼ size texture (half width and height - ¼ area). Then we horizontally blur the down sampled texture by sampling each pixel's color and outputting it's weighting combined with several of its neighbours weightings. Then we do the same vertically. Finally we up sample the texture back to a full screen render target and then to the back buffer. This produces a blurred scene. Note that the number of neighbours sampled is 4. A more blurry texture is possible with more neighbours or less blurry with fewer.

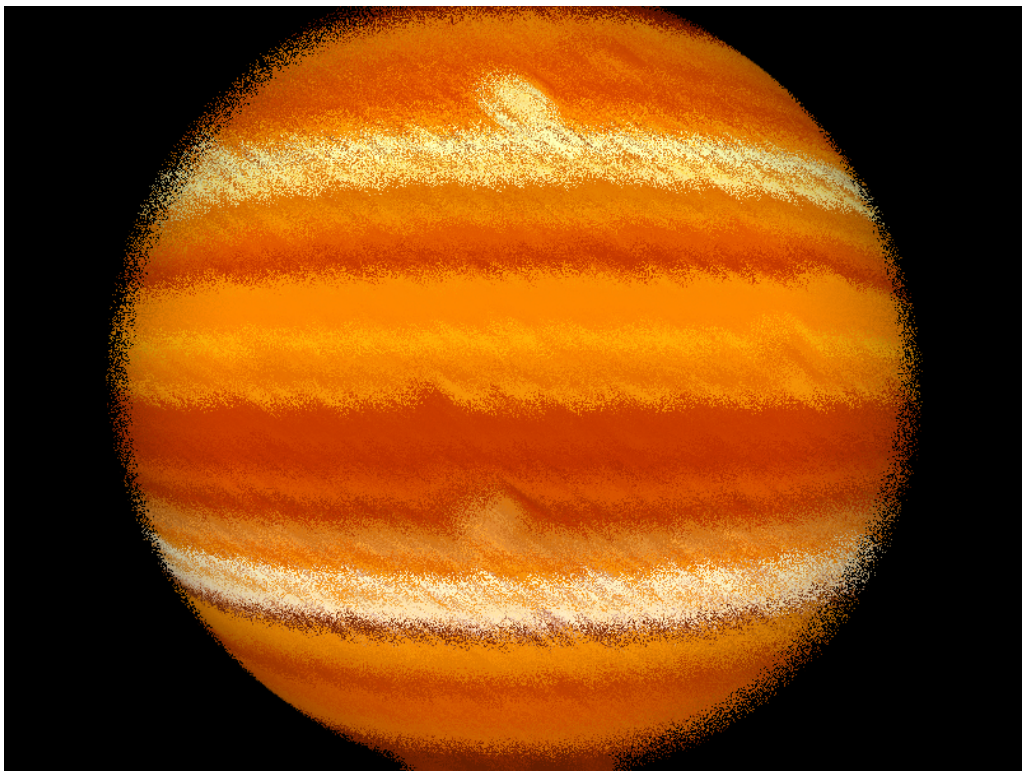
2.3.2 Distortion

With the scene rendered to texture, we then modify the texture coordinate of each pixel to produce a distorted scene. This is done using pseudo-randomly produced offset for the texture coordinate. We pass in time and a constant seed and multiply time by the seed and then sine of the pixels texture coordinates multiplied together to produce pseudo-random noise which is different for each pixel:

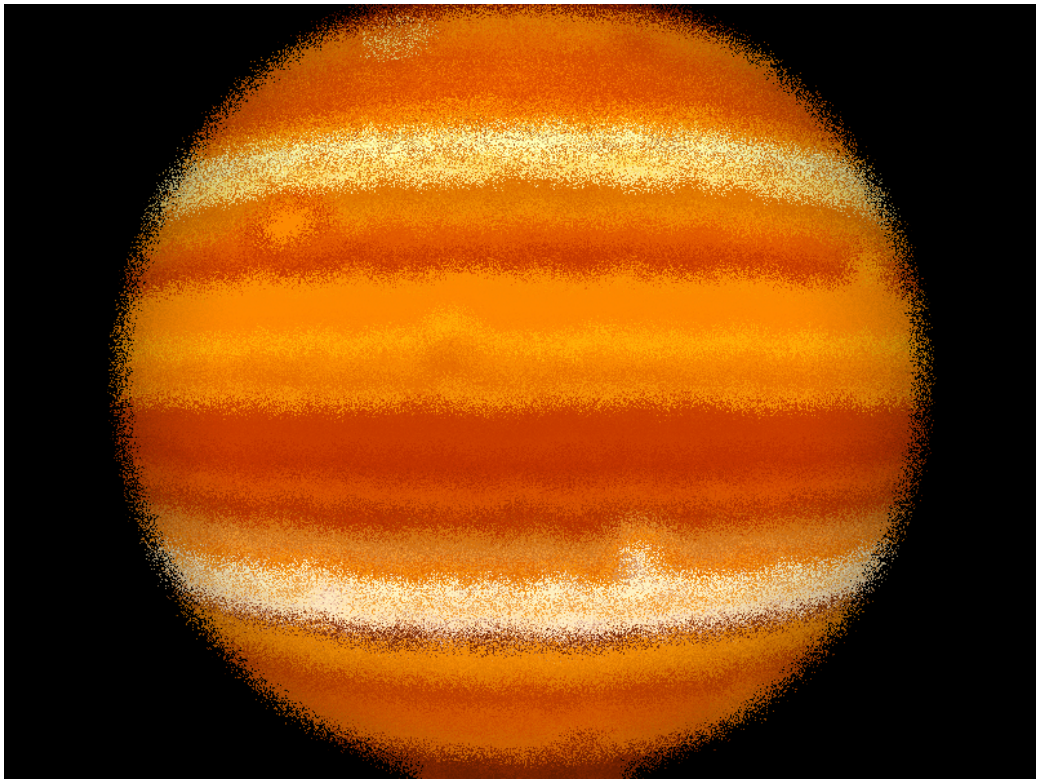
```
float noise = time * seed * sin(input.tex.x * input.tex.y);
noise = fmod(noise, 1) * fmod(noise, 1337);
float2 distortion = float2(fmod(noise, intensity), fmod(noise, intensity + 0.002f));
```

Then the noise is modified using the fmod function. With some experimentation I found that a bigger difference in the divisor value for the fmod functions produced better quality distortion. The offset value for the pixel is then created by returning the floating point remained of the noise divided by the desired distortion intensity. The y intensity offset slightly to produce distortion more uniformly in every direction. See below for example:

Without Y offset:



With Y offset:



Notice the lack of distortion at the bottom left and top right of the gas giant in the first image.

3. Critical Appraisal

I believe my application fulfils the coursework brief completely. I have demonstrated knowledge in the use of procedural generation relevant to the context of a game. Although most of the code is based off of work done by others I understand how it works and the implications of the methods used.

3.1 Improvements

The gas giant shader could be improved by using different noise samples at different texture coordinates to cause the surface to evolve in different directions like in real gas giants since currently the evolution of the texture is always in one direction. The octaves of noise could also be scaled with the distance from the camera and capped at a maximum determined by the texture resolution to improve performance when far away and close to the surface of the gas giant. Additionally I did not implement atmospheric scattering as present in the gas giant on the Seeds of Andromeda blog post (Arnold, B. 2015)[1]. This was mostly due to time constraints as it appears quite complex; I would like to try implementing this in the future.

The blur process can be improved by allowing the number of sampled neighbours to be specified and combining the horizontal and vertical stages into one stage.

The distortion process has a visible artifact at the top left of the screen when the top left is being distorted. I couldn't figure out a way to fix this problem completely. The distortion also seems to tend radially outwards from the bottom right of the screen. I believe this is due to the use of the sin function when first calculating a noise value. However for the effect produced, the distortion shader is very efficient computationally and sufficient for now. Since it's a post processing effect more expensive computations could be done to increase its quality - for example using simplex noise.

3.2 Reflection

In retrospect I shouldn't have tried experimenting with implementing bands. Since the planet itself does rotate, bands should not be very visible as a whole from a far away distance. It should only be a noticeable detail when close to the surface. Trying to manipulate time as a means to change the rate or direction of evolution of noise will most likely always end up producing bad results as time increases. Additionally I would have liked to spend more time experimenting with the fractal noise generated for the gas giant to try and get better looking results for the surface. If I were to do the project again I would recommend experimenting with fractal noise and storm implementation more.

3.3 Conclusion

Overall I'm very happy with the result of this project and I'm interesting in doing similar projects in the future. Another blog post written for Seeds of Andromeda was recently released explaining Procedural Star Rendering (Arnold, B. 2015)[8] which looks very interesting as a possible future project.

References

1. Arnold, B. 2015. [blog] *Procedural Gas Giant Rendering with GPU Noise*. Available from: https://www.seedofandromeda.com/blogs/49-procedural-gas-giant-rendering-with-gpu-noise [Accessed April 20 2015]
2. digitalerr0r. 2009. [blog] *Post Process - Noise/Distortion*. Available from: https://digitalerr0r.wordpress.com/2009/04/22/xna-shader-programming-tutorial-12-post-process-noise/ [Accessed May 1 2015]
3. McCoy, F. 2015. [source code] *Icosphere Generator*. Available from: http://pastebin.com/aFdWi5eQ [Accessed April 29 2015]
4. Ashima Arts. 2013. [source code] *Simplex Noise Functions*. Available from: https://github.com/ashima/webgl-noise [Accessed May 1 2015]
5. Rastertek. [source code] *Gaussian Blur*. Available from: http://www.rastertek.com/dx11tut36.html [Accessed April 29 2015]
6. Rastertek. [source code] *Camera*. Available from: http://www.rastertek.com/dx11tut04.html [Accessed April 29 2015]
7. Kahler, A. 2009. [blog] *Creating an Icosphere mesh in code*. Available from: http://blog.andreaskahler.com/2009/06/creating-icosphere-mesh-in-code.html [Accessed April 29 2015]
8. Arnold, B. 2015. [blog] *Procedural Star Rendering*. Available from: https://www.seedofandromeda.com/blogs/51-procedural-star-rendering [Accesssed May 3 2015]