PLANET RENDERING RESEARCH

TOPIC

In the last few years a considerable amount of games where announced or released that have mechanics involving the player moving from the surface of a planet to space without a loading screen, without a noticeable change in framerate. For my graduation work I am researching some of the techniques involved, like rendering the scene at various levels of detail (LOD) depending on the distance of the camera, along with some atmospheric effects. While LODs are not very complicated for simple assets – assets are switched out with premade lower resolution versions depending on their distance from the camera – significant effort typically goes into displaying the terrain, since it is a large asset where details need to be shown in areas close to the camera while seamlessly showing areas far away at a lower resolution. This means the terrain cannot be a static mesh, and the geometry needs to be recalculated every time the camera moves relative to the landscape.

Ever since games started being rendered in 3D research has been done on terrain rendering, and there are plenty of techniques and algorithms that have been developed, some of which are outdated because of how the graphics hardware has developed. Furthermore, some of the approaches are not suitable for huge terrains, especially planets, and work better in a smaller environment. Therefore, I focused the first couple of weeks of my research on finding good methods to generate the terrain geometry both theoretically and practically in OpenGL.

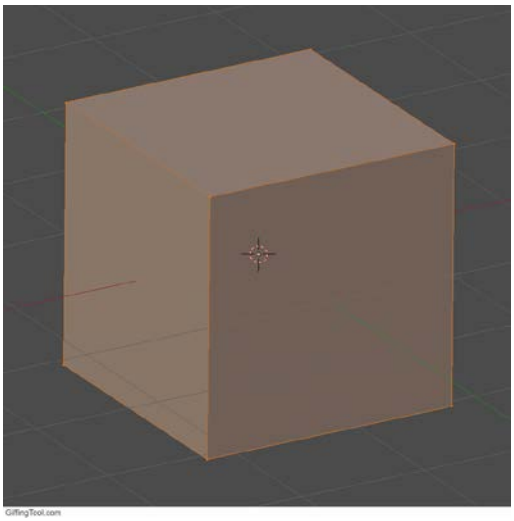
OVERVIEW

The rendering of planetary terrain has a key difference, apart from being larger than typical applications, is that it is spherical. This means the algorithm needs to use some convex base mesh which is rounded off during terrain generation. Depending on the base mesh, different kinds of primitives will be used, namely triangles or triangulated rectangles. These primitives are subdivided into multiple primitives of the same kind, and the level of subdivision depends on the distance from the camera, which should ideally cause the same amount of geometry distribution in every area of the screen, regardless of the cameras proximity. Switching out the subdivision levels should be as unnoticeable as possible, and it is important to avoid cracks on the borders between different subdivision levels. The computer should waste as little time as possible rendering terrain that is not visible to the camera, so an efficient way of determining what terrain not to render (culling) before it is even generated should be found. The height of the terrain needs to be determined at a certain stage using either data prepared on some form, relying on procedural height generation or a combination of the two. Lastly, an ideal algorithm would be able to do a lot of the calculations on the GPU, since on modern computes its parallel processing powers are better at dealing with large amounts of geometry.

BASE SHAPE

I identified two types of base shape, which come with different approaches.

The first category is a spherified cube or quadsphere. A subdivision is grid based could look like this:



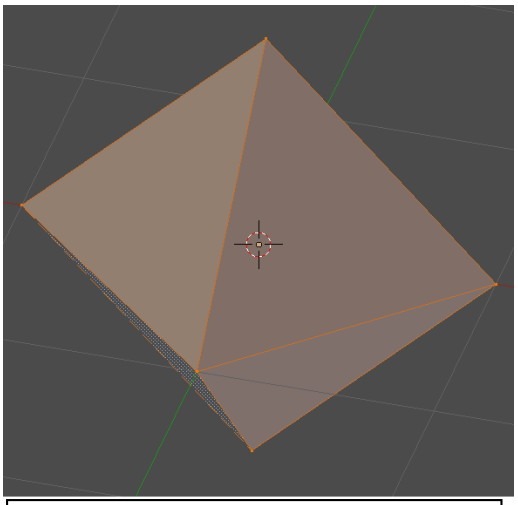
Benefits of basing it on a cube are the accessibility of using common terrain algorithms on the 6 faces and then simply spherifying every face. Also the quad base is likely to be efficient for memory usage. Typically, a quadsphere uses a quadtree (for culling too) and an algorithm such as:

- Continuous Distance-Dependant LOD
- Geometry Clipmaps (GPU)
- Chunked LOD
- Geometrical MipMapping

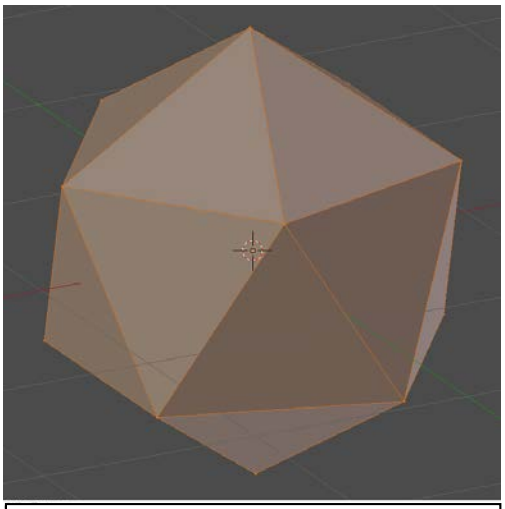
Cube

Another advantage is the accessibility to texturing, while the main downside is the irregularity of triangle scale at corners vs in the middle of the face, and the extra step of triangulating the faces.

The other category is using polyhedrons such as octahedrons or icosahedrons. Spheriphying those would look something like this:



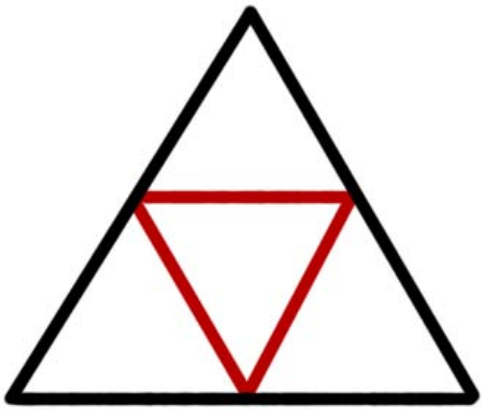
Octahedron

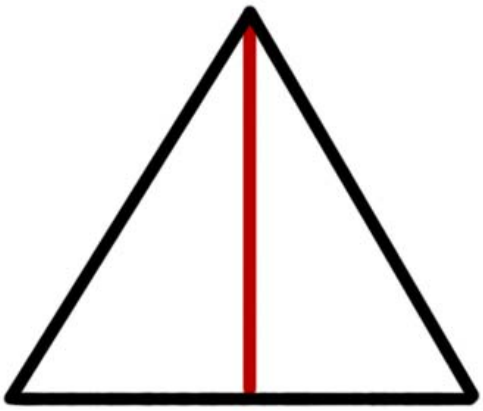


Icosahedron

They have different tree structures because of the shape of their triangles: Octahedrons have right angled triangles, which is ideal for a binary tree, as subdividing the triangle would use the process of splitting the triangle in two halves, resulting in two new right angled triangles.

Icosahedrons on the other hand consist of equilateral triangles and therefore best subdivide into 4 new equilateral triangles, and therefore would use a quaternary triangle tree. In both cases memory alignment should be achievable since there are no odd numbers used in either tree structure.

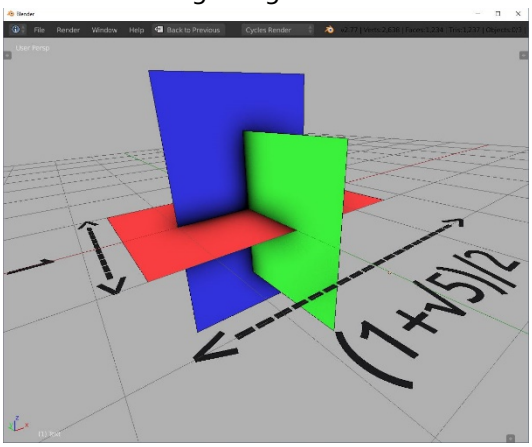




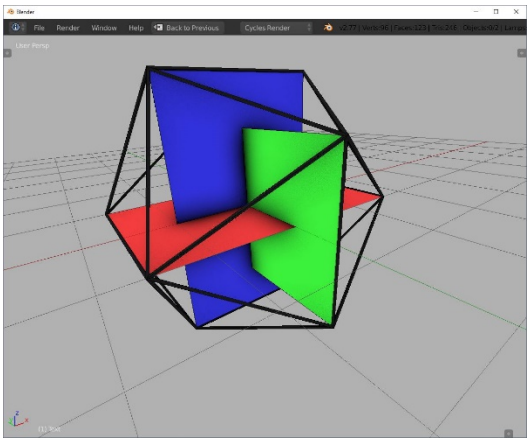
In both cases though the ROAM 2.0 algorithm seem to be most adequate for terrain LOD, since it is designed to handle planetary terrain. The downside of it is that it has a fairly high complexity and CPU workload (compared to most algorithms presented for the quadsphere), but that may in turn free up the GPU workload for other neat effects such as atmospheric scattering etc.

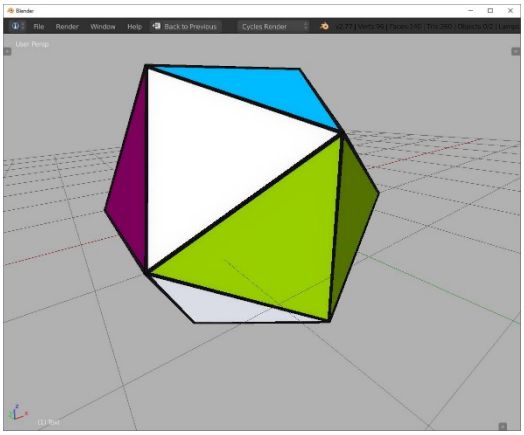
In any case polyhedrons clearly have less problems with evenly distributing triangle scale, especially icosahedrons, which is why I tried an implementation. The process is rather simple:

Define the triangle edges with three intersecting planes that use the golden ratio (1+sqrt(5))/2 : 1

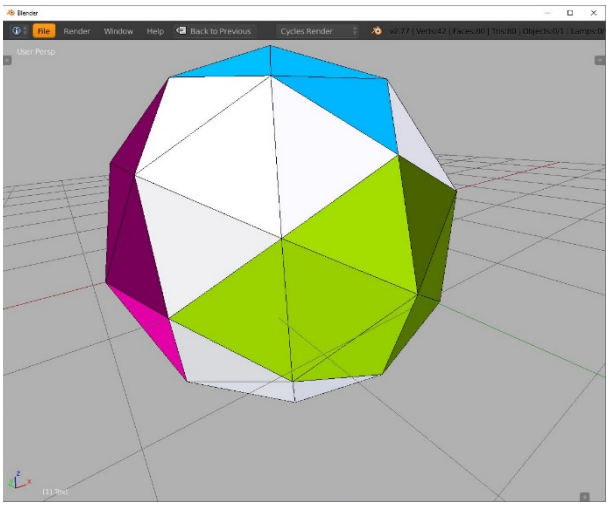


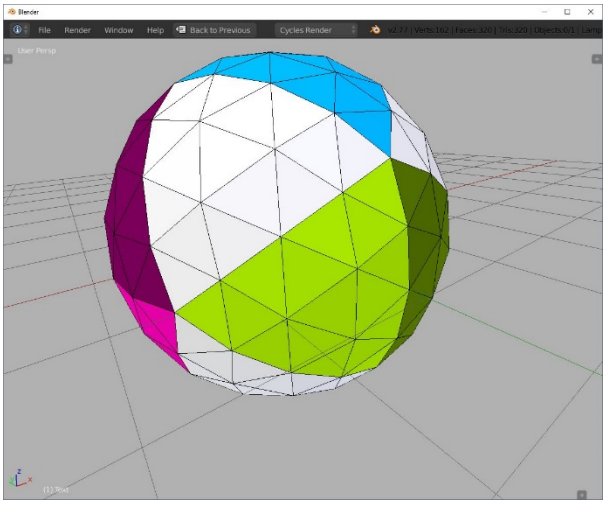
Create 20 triangles between the vertices of those planes:12 vertices

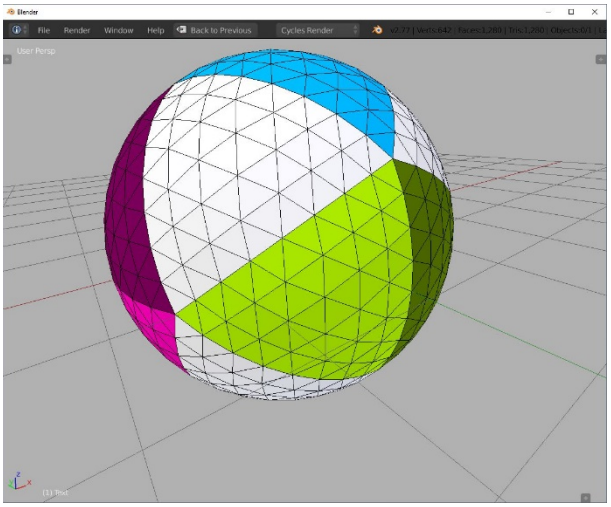


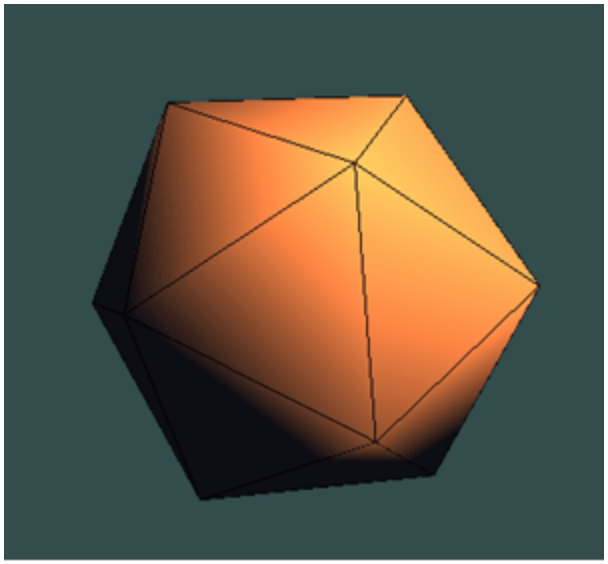


Subdivide those triangles and keep them in the shape of a sphere by making sure the distance from the center is the radius of the sphere:









I implemented this OpenGL. As can be seen on the right.

I reused a bunch of (modified) code from the OpenGL Framework I wrote last spring, without actually writing the project inside in order to avoid making the code unnecessarily complex. The goal after all is implementing a techdemo that has the necessary code for rendering planets and nothing more, so that it is easy to identify the code necessary. It will always be possible to transfer the project into a more game friendly engine.

So for now, the framework contains the following components:

- A shader class that combines all shader types and handles loading and simple preprocessing of .glsl files
- An input manager
- a transform class that contains basic model-world matrix transformations and information about position rotation and scale of an object
- a camera class that handles view and projection matrices
- OpenGL and SDL2 initialization, a main loop
- window, time and settings information
- a scene handling all important objects
- texture support
- sprite text font rendering

By necessity I will probably also add sprite rendering for post processing with framebuffers.

```
void Planet::RecursiveTriangle(glm::vec3 a, glm::vec3 b, glm::vec3 c, short level)
{
    //check if triangle in view
    bool visible = true;
    if (visible)
    {
        //check if subdivision is needed based on camera distance
        bool subdivide = level < m_MaxLevel;
        if (subdivide)
        {
            //find midpoints
            glm::vec3 A = b + ((c - b)*0.5f);
            glm::vec3 B = c + ((a - c)*0.5f);
            glm::vec3 C = a + ((b - a)*0.5f);
            //make the distance from center larger according to planet radius
            A *= m_Radius / glm::length(A);
            B *= m_Radius / glm::length(B);
            C *= m_Radius / glm::length(C);
            //Make 4 new triangles
            short nLevel = level + 1;
            RecursiveTriangle(a, B, C, nLevel);
            RecursiveTriangle(A, b, C, nLevel);
            RecursiveTriangle(A, B, c, nLevel);
            RecursiveTriangle(A, B, C, nLevel);
        }
        else //put the triangle in the buffer
        {
            m_Positions.push_back(a);
            m_Positions.push_back(b);
            m_Positions.push_back(c);
        }
    }
}
```

The nice thing about this approach is that it allows stopping the recursion depending on a different heuristic than being below the maximum level. As an example, the distance from the camera and the visibility in the viewport could be used. For actual planet rendering, the entire thing should be placed in some sort of tree hierarchy, but for now this works.

I also tested the real time updating of the vertex buffer with the simplest approach (simply rebinding it). When changing the buffer, actually recursively generating the geometry takes the longest time. When taking out this step (while still rebinding the vertex buffer), I was able to send approximately 3.5 triangles to the GPU (NVidia Quadro K3100M, Intel i7-4700MQ, built in release mode) at around 180fps. Without rebinding the buffer those triangles ran at around 550 fps. I was quite impressed with the amount of data I was able to push to the GPU, by an extension of these measurements I could probably send one triangle per pixel to the GPU every frame and stay above 60fps.

```
//Change Planet
//***************
bool geometryChanged = false;
if (INPUT->IsKeyboardKeyPressed(SDL_SCANCODE_UP))
{
    m_MaxLevel++;
    geometryChanged = true;
}
if (INPUT->IsKeyboardKeyPressed(SDL_SCANCODE_DOWN) && m_MaxLevel>0)
{
    m_MaxLevel--;
    geometryChanged = true;
}
if (geometryChanged)
{
    //Change the actual vertex positions
    GenerateGeometry();
    //Bind Object vertex array
    glBindVertexArray(m_VAO);

    //Send the vertex buffer again
    glBindBuffer(GL_ARRAY_BUFFER, m_VBO);
    glBufferData(GL_ARRAY_BUFFER, m_Positions.size() * sizeof(glm::vec3), m_Positions.data(), GL_STATIC_DRAW);
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    //Done Modifying
    glBindVertexArray(0);
}
std::cout << "FPS: " << TIME->FPS() << " - Vertices: " << m_Positions.size() <<std::endl;
```

Of course, spending all of the performance on streaming vertices is less than ideal, since in a real world application a lot more calculations need to be done apart from terrain generation (not to mention actually generating the vertex data which was not done every frame here). It is definitely not necessary to have one triangle represent every pixel though. If every triangle covers a 5- 10 pixel square it should still be enough to represent curves rather nicely, and for areas that do not form silhouettes from the camera perspective even less vertex detail is necessary since a lot can be done with nice shaders and textures.

DISTANCE DEPENDANT SUBDIVISION

Using the naive recursive approach to subdivide the sphere can quite easily be extended to make the subdivision localized depending on the camera distance. To do this, some form of heuristic is used to determine if the triangle (or in other applications, quad) should be subdivided or not. The same heuristic also checks if the triangle should be culled, but that will be discussed in a later section.

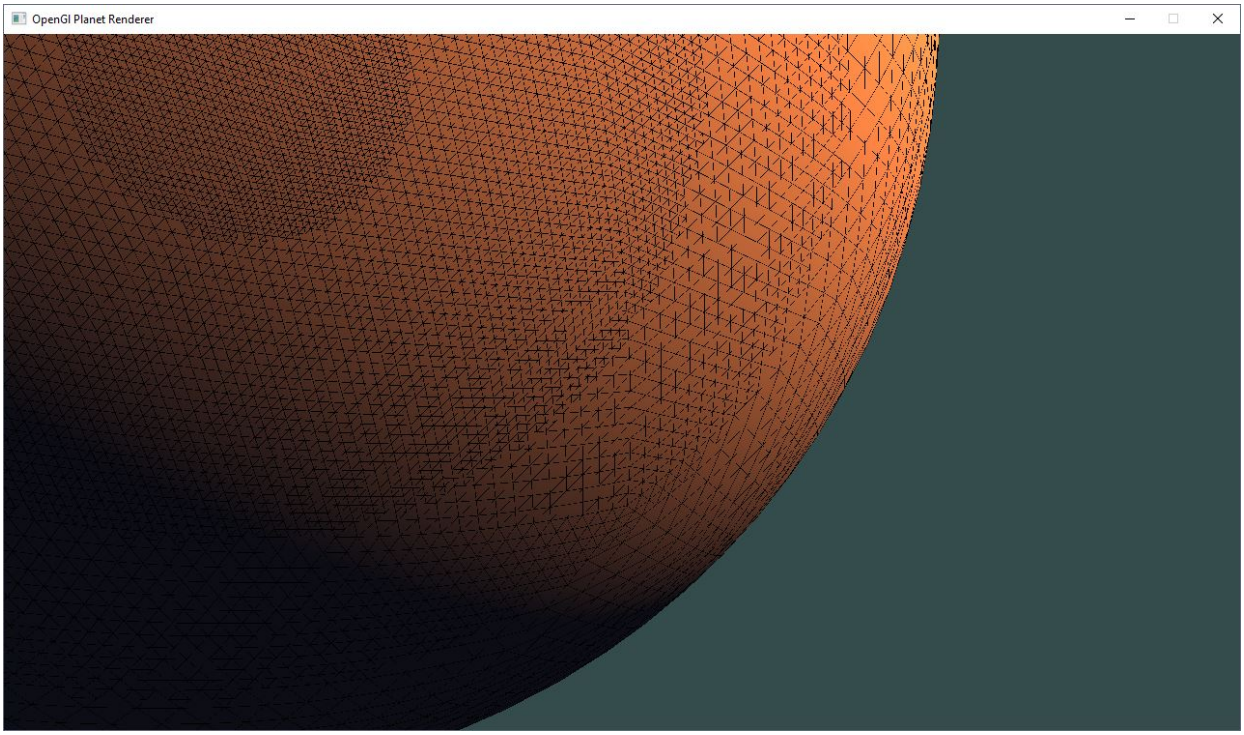
So the first step is performing some form of culling, which I will detail in a later post, but the idea is finding a good method to prevent generating and rendering geometry for invisible terrain, such as terrain that is on the other side of the planet or not visible in the cameras view frustum. If the culling test is positive, subsequent steps won't be calculated anymore and the heuristic returns a CULL command.
If the triangle passes the culling test, it is checked if it should be split (subdivided) based on its position and the cameras position. If this test is positive, the heuristic returns a SPLIT command, otherwise it will return LEAF.

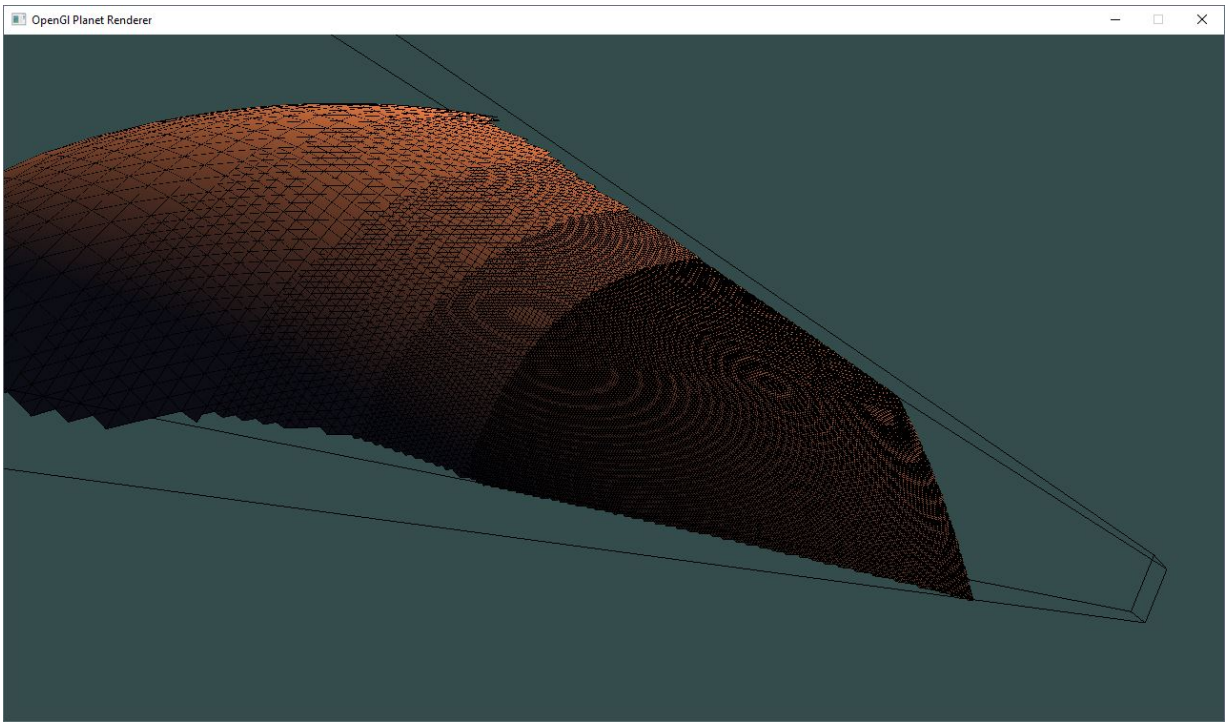
In the recursive subdivision method, this information is used to either discard the triangle, recursively subdivide it or add the triangle positions to the vertex buffer.

For the distance check, at first I used this rather simple formula:

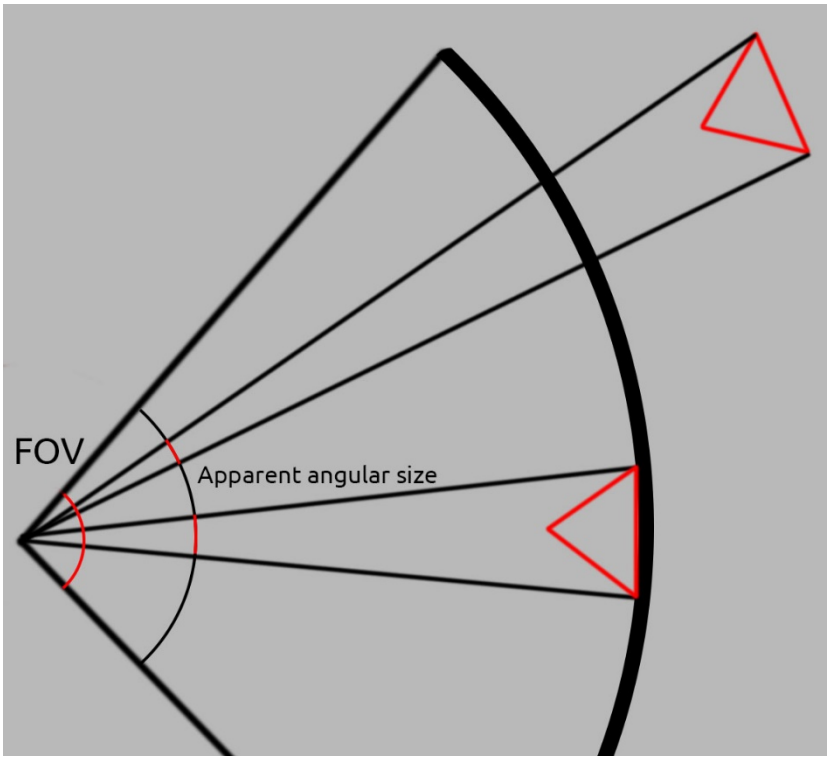
```
if(distance*subdivisionLevel > magicNumber)return SPLIT;
```

Using this approach, I was already able to get some results:





This is already a quite promising result, but it has one problem, which is that the detail level does not appear uniform on the screen size. At a large distance the level of detail is to small and the planet seems really low poly, and at a short distance the level of detail is way to high and causes an uncomfortably small framerate.



Ideally, you would want the triangles to be all be roughly the same size on the screen, regardless of the distance. The naive version of implementing this would be to calculate the surface area in screen space, but that would be extremely expensive since it would require transforming all triangles with the WorldViewProjection matrix, which is a rather slow process when done of the CPU.

Luckily, I was able to find a good alternative, which is calculating the triangles maximum size based on the size of its sides, its distance from the camera and the field of view of the camera.

The idea is that the angular size according to the angle of a triangle made from the camera position to two points on the triangle needs to be smaller than a predefined value.

The size of a triangle at a certain subdivision level can be precalculated. Since I am using an icosahedron as a base shape, all triangles will be roughly equilateral. The base size (level 0) is the distance of two arbitrary corners of an arbitrary triangle of the icosahedron, and the size of every level is half the size of the previous level. All of those triangle sizes can be placed in a look up table, for which any array will do, where the index is the subdivision level:

```
triLevelSizeLUT.clear();
triLevelSizeLUT.add(distance(baseIco[3] – baseIco[1]));
for (i = 1; i < maxSubdivLevel; i++)
{
    triLevelSizeLUT.add( triLevelSizeLUT[i-1]/2 );
}
```

This is precalculated along with the maximum percentage the triangle angle can be of the cameras FOV angle every frame:

```
maxScreenSizePercentage= maxTrianglePixels/ WINDOW.Width;
```

The angular size of a triangle at a certain subdivision level and distance then can be calculated similar to how the near clipping plane is calculated with the usage of an inverse tangent

```
angSize = arctan(size[subdivLevel]/distance);
```

And this information can be used to determine a subdivision:

```
if(angSize/cameraFOV > maxScreenSizePercentage) SPLIT;
```

In order to make all of this more performant, the camera is transformed into the object space of the planet.

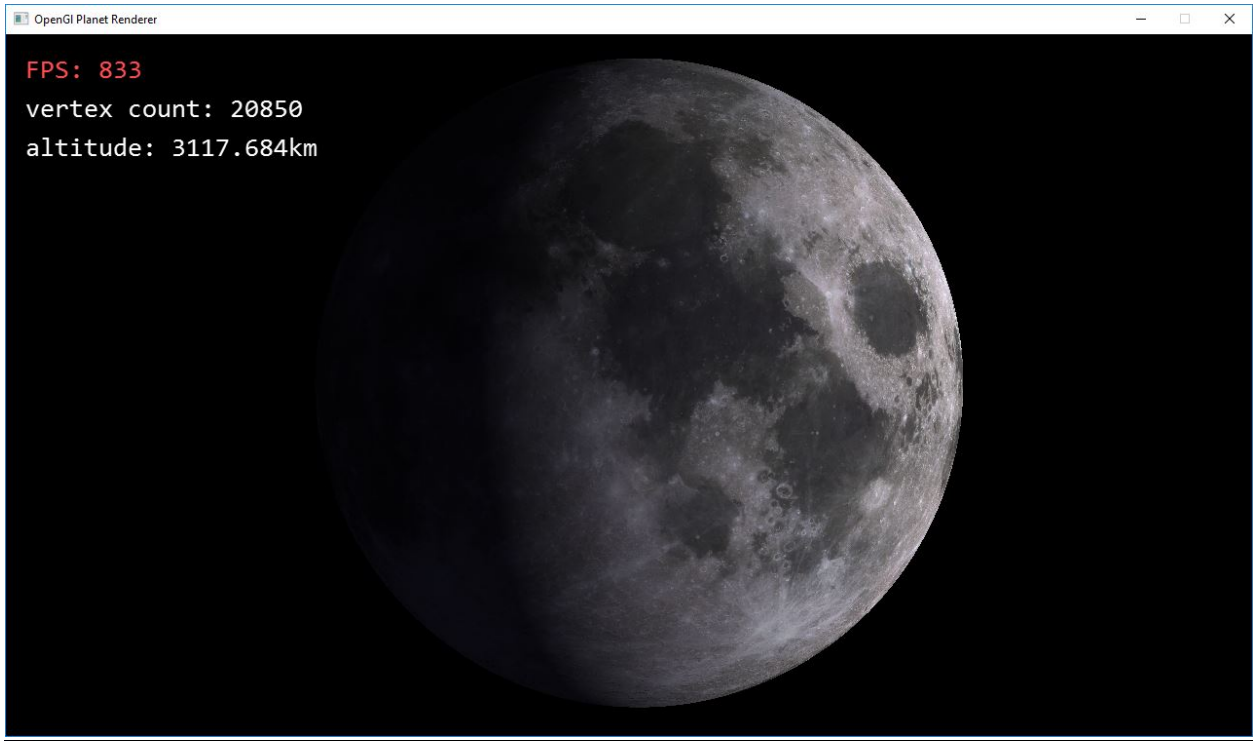
Using some linear algebra, this formula can be flipped around to precalculate the splitting distances, which will give another performance improvement, since only the distance at a certain level needs to be queried and the arc tangent does not have to be calculated for every triangle.

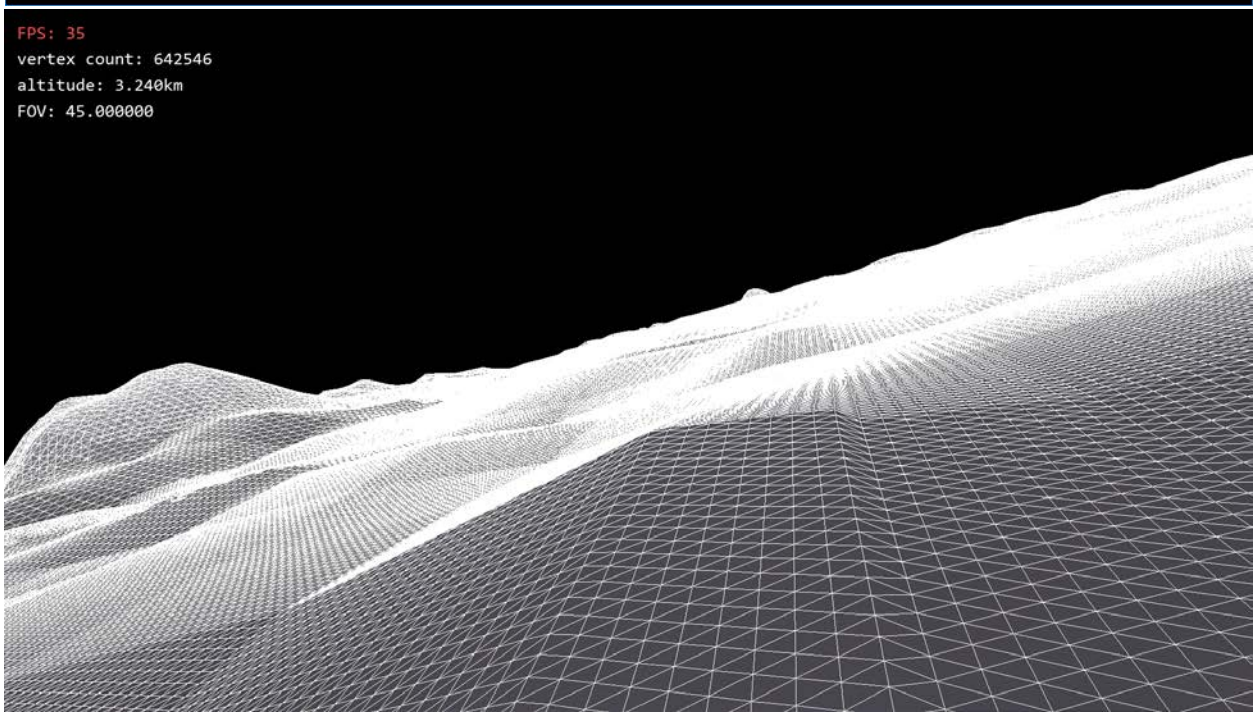
```
distanceLUT.clear();
float sizeL = distance(baseIco[0].a - baseIco [0].b);
float frac = tan((maxTrianglePixels * cameraFOV) / WINDOW.Width);
for (int level = 0; level < maxSubdivLevel; level++)
{
       distanceLUT.add(sizeL / frac);
       sizeL *= 0.5f;
}
```

This makes querying a split as simple as a single lookup:

```
if (distance(center - camPos) < distanceLUT[level])return SPLIT;
```

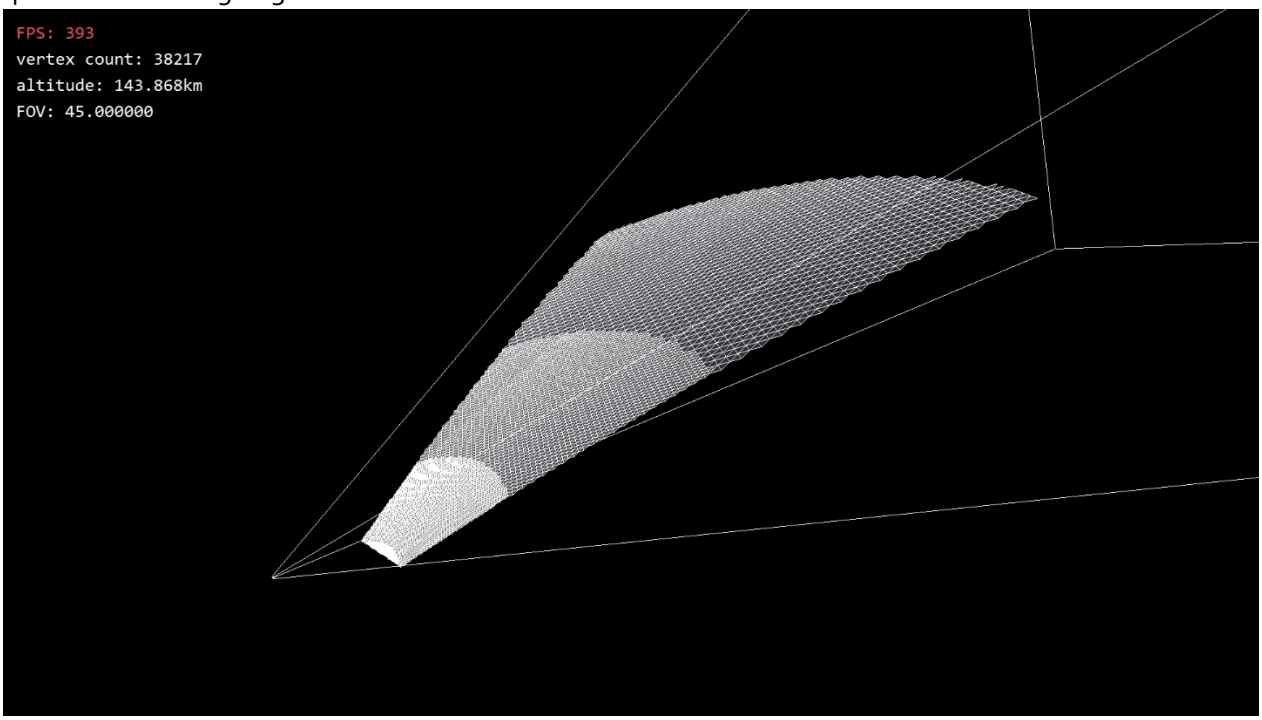
Once I had all of this implemented I added texture support and equilateral texture sampling to the planet shader. Here is what the result looks like with a moon texture:





CULLING

As mentioned above, I will be discussing the culling of invisible geometry. The following picture shows quite well what is going on:



Here only the necessary geometry is generated. The implementation to achieve this uses two methods combined: backface culling and frustum culling. Both methods will be discussed along with their implementation

BACKFACE CULLING

Backface culling can be solved with a relatively simple dot product between the view vector and the normal of the triangle. Since the triangle is on a perfect sphere, we can simply use the position normalized as the normal. Therefore, the simple implementation for this is:

```
vec3 center = (a+b+c)/3;
if(dot(norm(center), norm(center-camPos) ) >= 0) return CULL;
```

For efficiency the camera and frustum have been transformed into the planets object space.

There is a problem with this though. Since we are subdividing the triangles into a sphere, the children of the triangle will have different angles that might not be backfacing even though the parent is. In this case the diverging angle is halved with every subdivision. So if the level 0 angle difference is 16 degrees, the children at level 1 can have a maximum angle difference of 8, then 4, then 2 and so on.

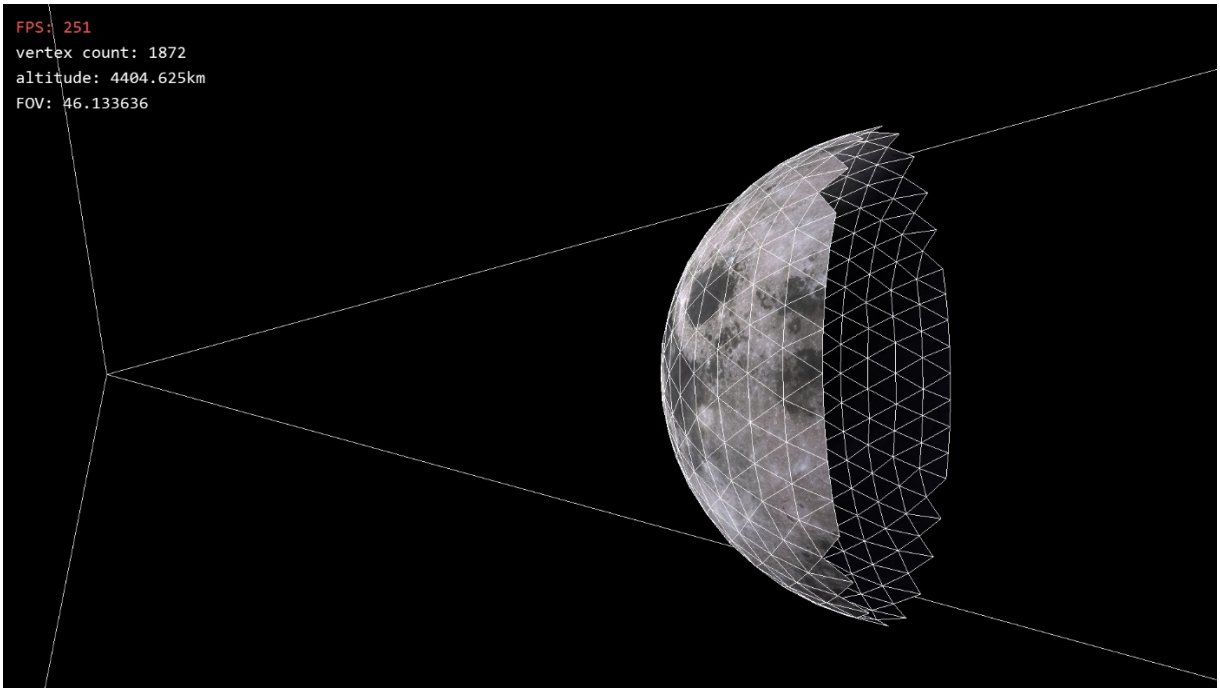
Since a dot product is the cosine of the angle, we can precalculate the minimum dot product required to cull a triangle per level in a look up table. I ended up using the sin of the angle instead of the cosine, because we want the triangles to be at 90 degrees to cull, not 0. Here is what this calculation looks like:

```
triLevelDotLUT.clear();
triLevelDotLUT.add(0.5);
float angle = arccos(triLevelDotLUT[0]);
for (int i = 1; i < maxSubdivLevel; i++)
{
    angle = angle/2;
    triLevelDotLUT.add(sin(angle));
}
```

This needs to be recalculated every time the base mesh or the maximum amount of subdivisions changes. We can now use this table in our dot product from before like so:

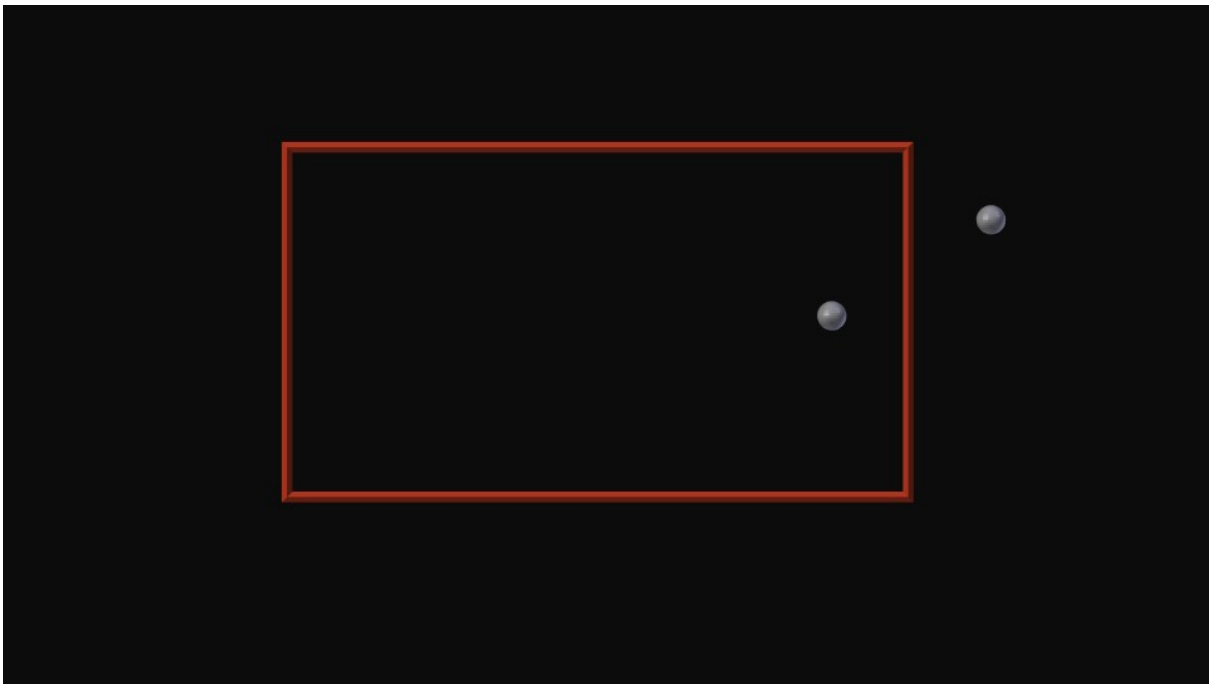
```
if(dot(norm(center), norm(center-camPos) ) >= triLevelDotLUT[level])
    return CULL;
```

The result is that only the front facing half of the planet is generated and rendered:



FRUSTUM CULLING

The next and more complicated step is frustum culling. The idea here is that if something is not in the screen it should not be rendered or calculated. In the following image, the right point should be removed:



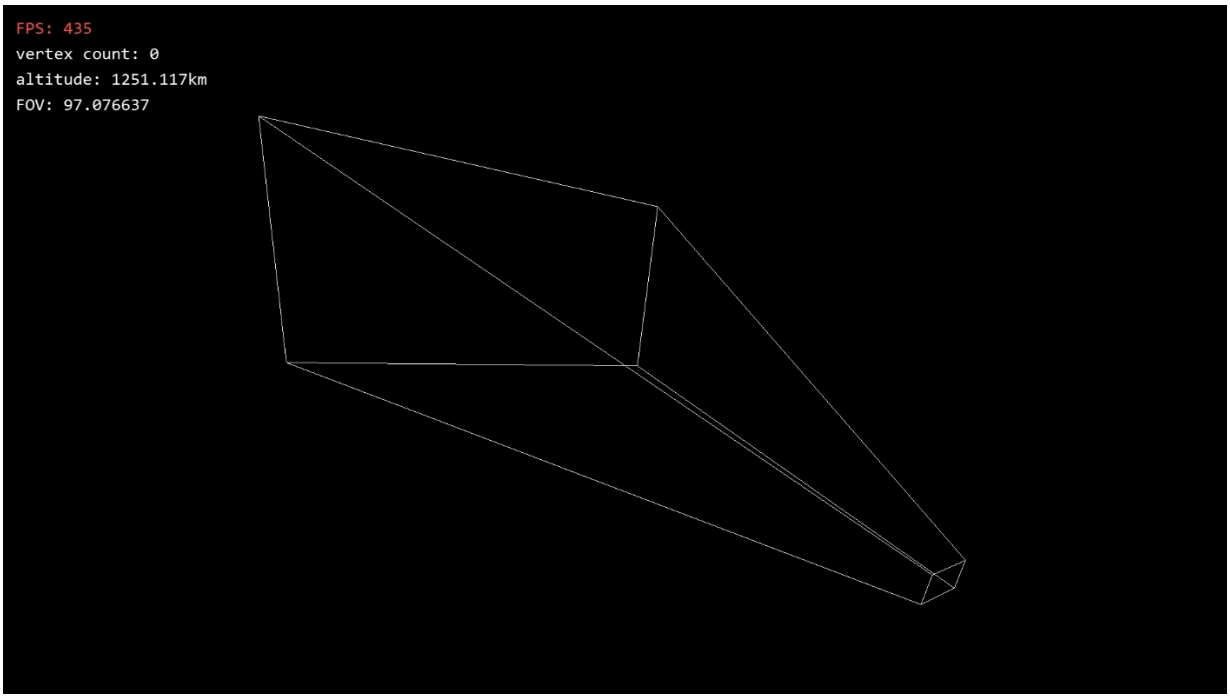
One way of doing this would be to transform every vertex with the world view projection matrix into screen space and then checking it in 2D with the rectangles boundaries. This is rather expensive though, since matrix transformations involve quite a few calculations which is not ideal on the CPU. I was able to avoid transforming all vertices by transforming the camera into object space once at the beginning with the inverse of the planets world matrix.

Therefore, the better solution for the problem is constructing the camera frustum out of 6 planes. Two of them are already familiar as the near and far clipping plane, as OpenGL also uses those. What is left is the Top, Bottom, Left and Right plane. They can be constructed from the corners of the near and far plane, which in turn can be calculated from the camera orientation axes and the width and height of those planes, which in turn can be calculated with the tangent of the FOV:

```
float normHalfWidth = tan(FOV);
float aspectRatio = width / height;

float nearHW = normHalfWidth*nearPlaneDist;
float nearHH = nearHW / aspectRatio;
float farHW = normHalfWidth*farPlaneDist/2;
float farHH = farHW / aspectRatio;
```

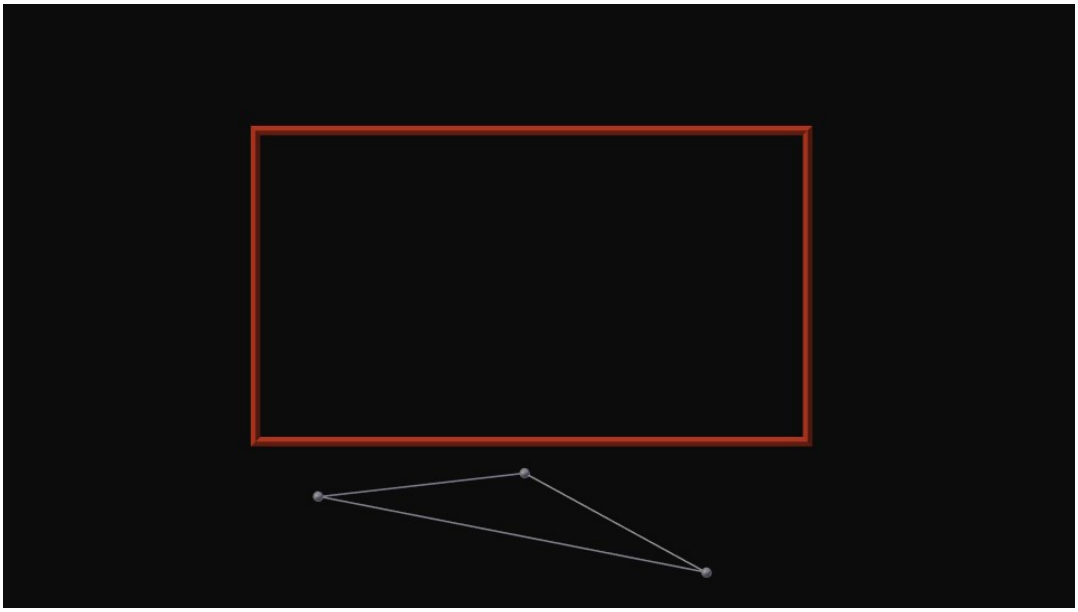
This results in a geometry that looks like a pyramid which has been flattened at the tip:



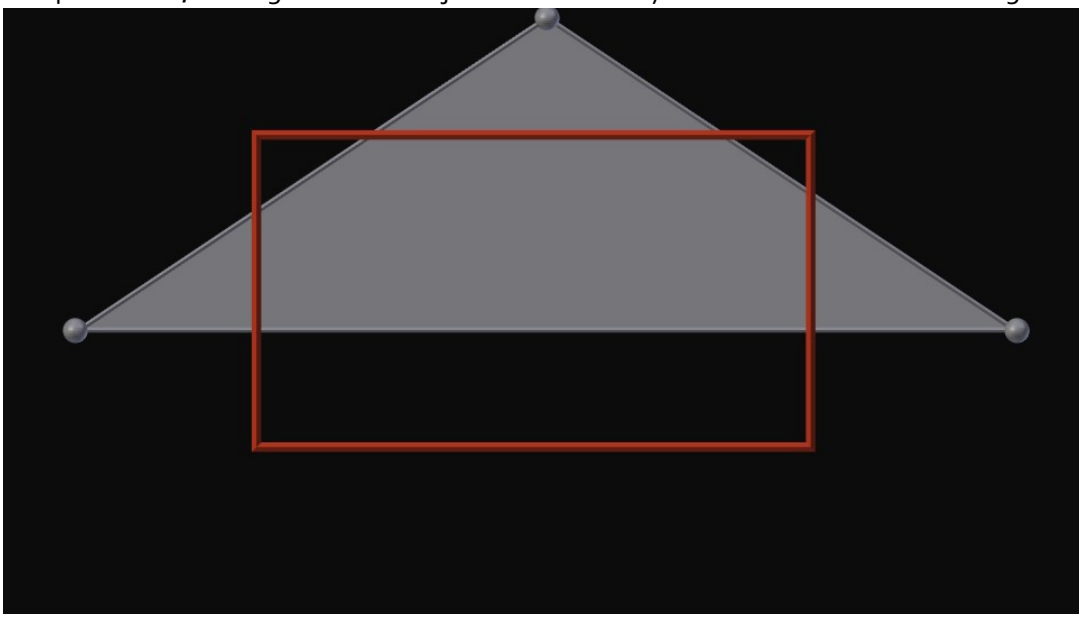
All those planes are described in their Normal form (a position ***D*** and a normal ***N***). When they are constructed in a way that they all face inwards, one can calculate a dot product of the normal and the vector between the planes position and the vertex. If this dot product returns a negative value for any plane, the vertex is outside the view frustum.

```
bool Frustum::Contains(vec3 P)
{
    for each Plane
    {
        if (dot(Plane.N, p – Plane.D) < 0)return false;
    }
    return true;
}
```

This implementation will work if you check every vertex of a triangle in most situations. Triangles outside the frustum will definitely be rejected…

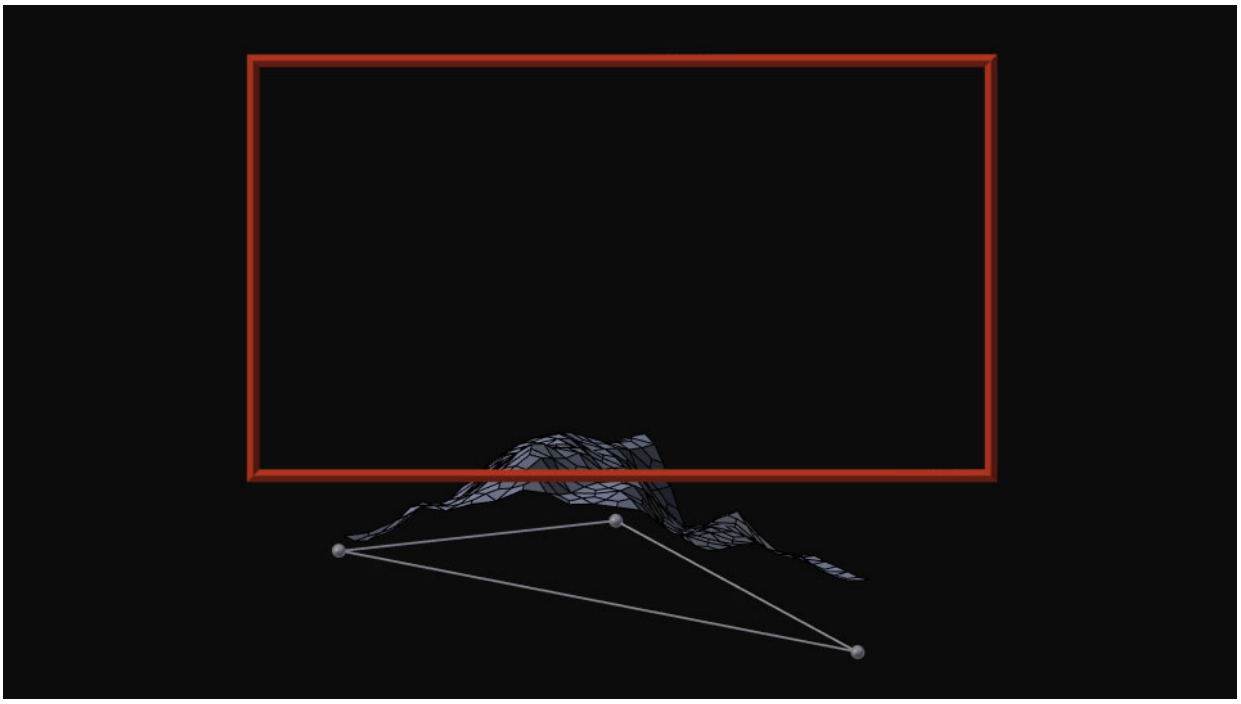


The problem is, this algorithm will reject a few to many cases. Consider the following triangle:

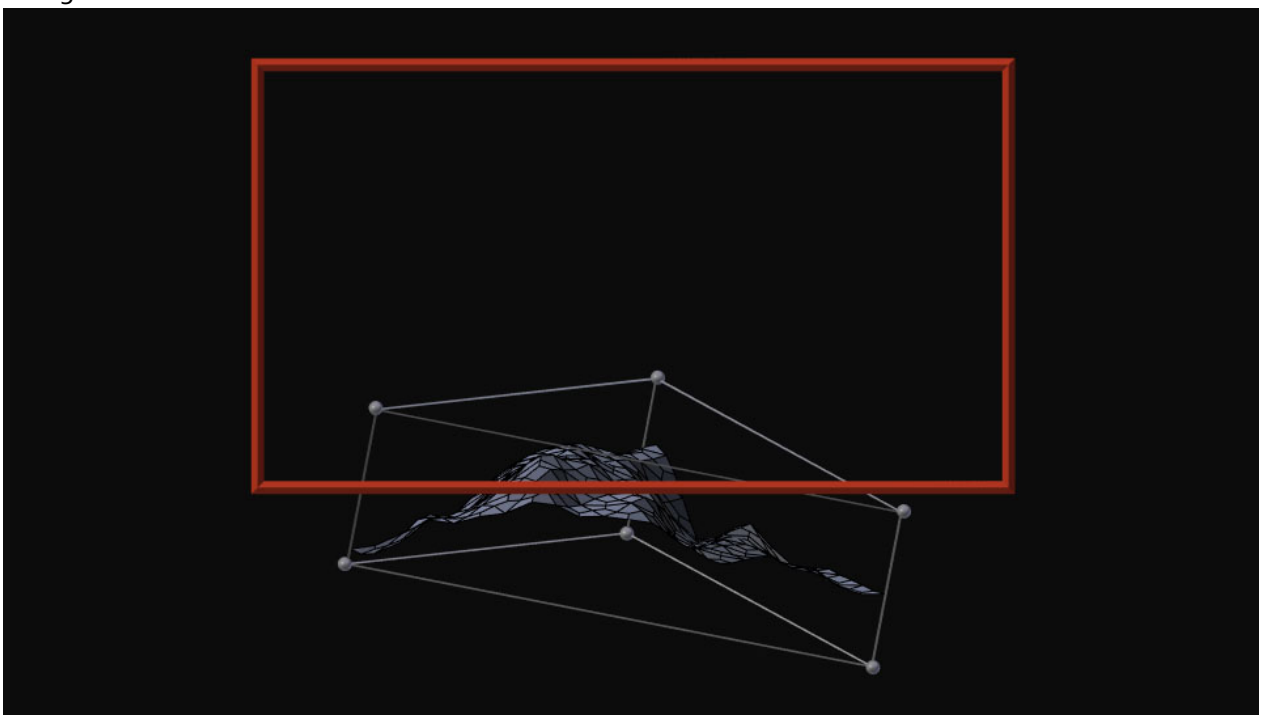


All vertices are outside the frustum, but a part of the triangles surface is still inside. The clean solution for this problem would be an intersection code test of the triangle with the planes, but that would be way to expensive when generating a terrain consisting of thousands of triangles.
The faster way of solving this is saying that ALL vertices of the triangle need to be on the "bad" side of the plane. This may return true for a few triangles that are positioned diagonally next to the corner of the rectangle, but since all triangles are rather small in their final form, the amount of errors is acceptable.

You may think that this solves all problems, but there is a further case in which we cannot allow the triangle to be culled. Because the children of the triangle are curved upwards (and also have terrain that may be above sea level), the children may be inside the screen even though the parent triangle is not.



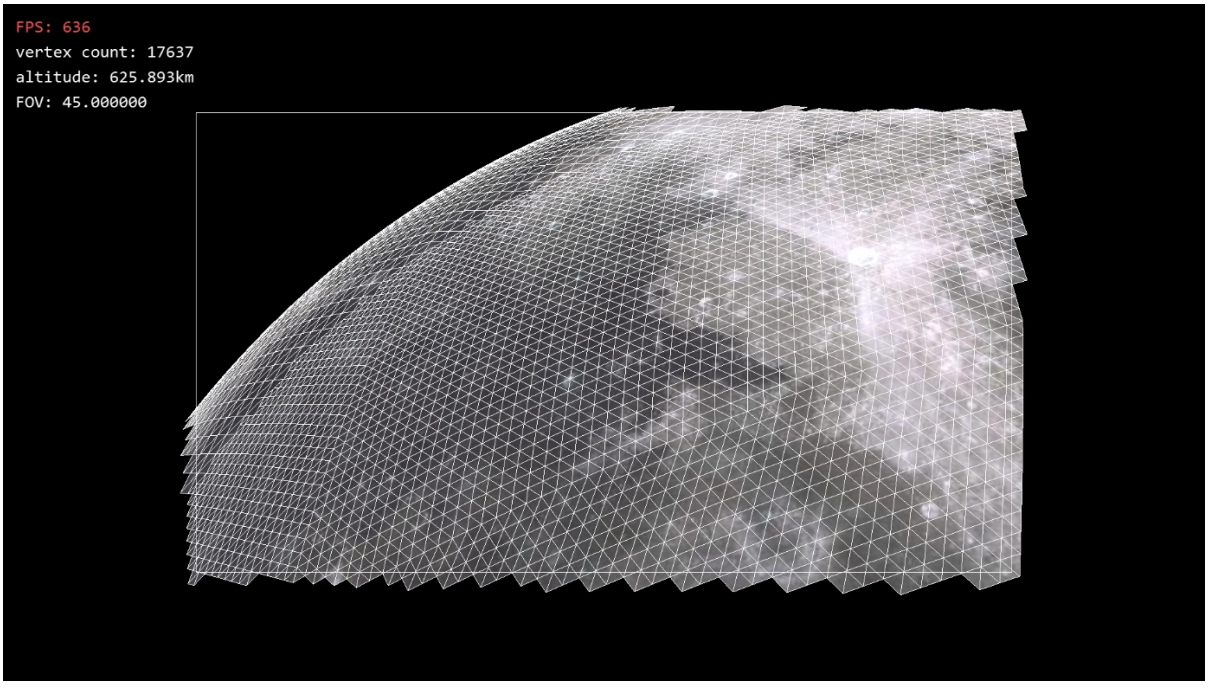
To solve this, some sort of volume is needed that contains all possible children of the triangle. In classical terrain renderers this is typically handled with axis aligned bounding boxes, but since I am dealing with spherical terrain, an axis aligned box would waste a lot of empty space that could never be reached, and boxes are better with quad tree geometry than triangles anyway. My solution is to instead extrude the triangle up by the maximum height and making sure to only cull if all 6 vertices of the triangles volume are outside the frustum.

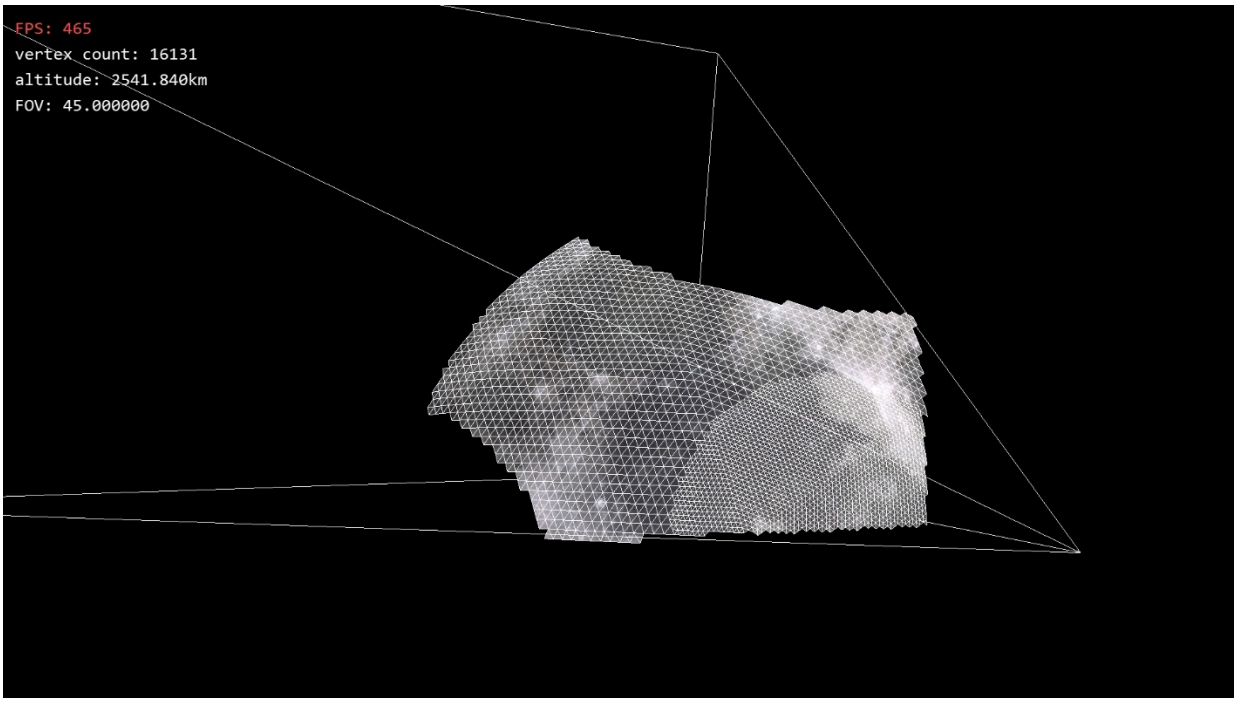


The height will differ per subdivision level, but it can also be precalculated in a lookup table like with the backface culling before. It can be determined as a position multiplier for the spherical terrain. Multiplying the vertex positions with the height value will offset the from the centre of the sphere, so the heights of the table can be determined with the dot product of a corner vertex and the centre of the triangle that has the correct height applied to it:

```
heightMultLUT.clear();
vec3 a = icosahedron[1];
vec3 b = icosahedron[3];
vec3 c = icosahedron[8];
vec3 center = (a + b + c) / 3;
//addMaxTerrainHeight here too
center *= planetRadius / length(center);
heightMultLUT.add(1/dot(norm(a), norm(center)));
for (int i = 1; i < maxSubdivLevel; i++)
{
    vec3 A = b + ((c - b)/2);
    vec3 B = c + ((a - c)/2);
    c = a + ((b - a)/2);
    a = A * planetRadius / length(A);
    b = B * planetRadius / length(B);
    c *= planetRadius / length(c);
    heightMultLUT.add(1/dot(norm(a), norm(center)));
}
```

Taking all this into account, the final result looks something like this (I made the frustum FOV smaller than the cameras FOV for visualization purposes):

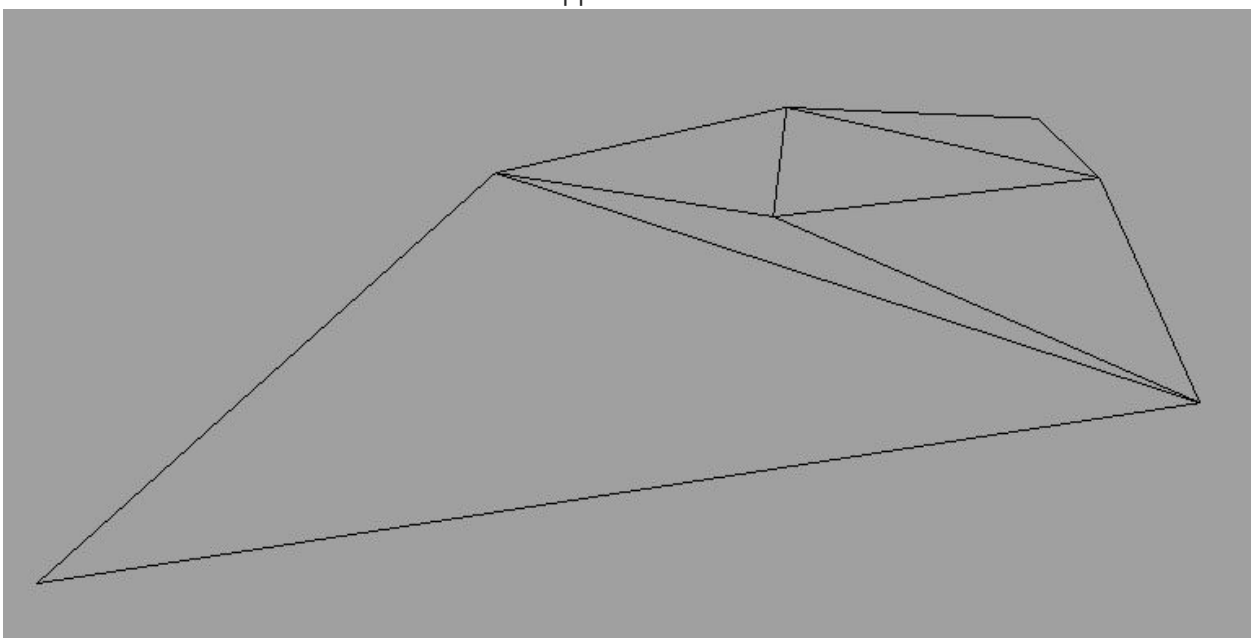




Since, when considering all those problems, frustum culling gets quite expensive, I let the culling function return if the volume is outside, intersecting or contained. If the volume is outside, its children are not generated and it is not rendered. If it is intersecting, the algorithm proceeds as usual. If it is contained, the algorithm also continues, but frustum culling is not checked any more for the children triangles, since their parent volume is already completely contained.

CRACK AVOIDANCE

At the borders of subdivision level cracks can appear due to difference in elevation.



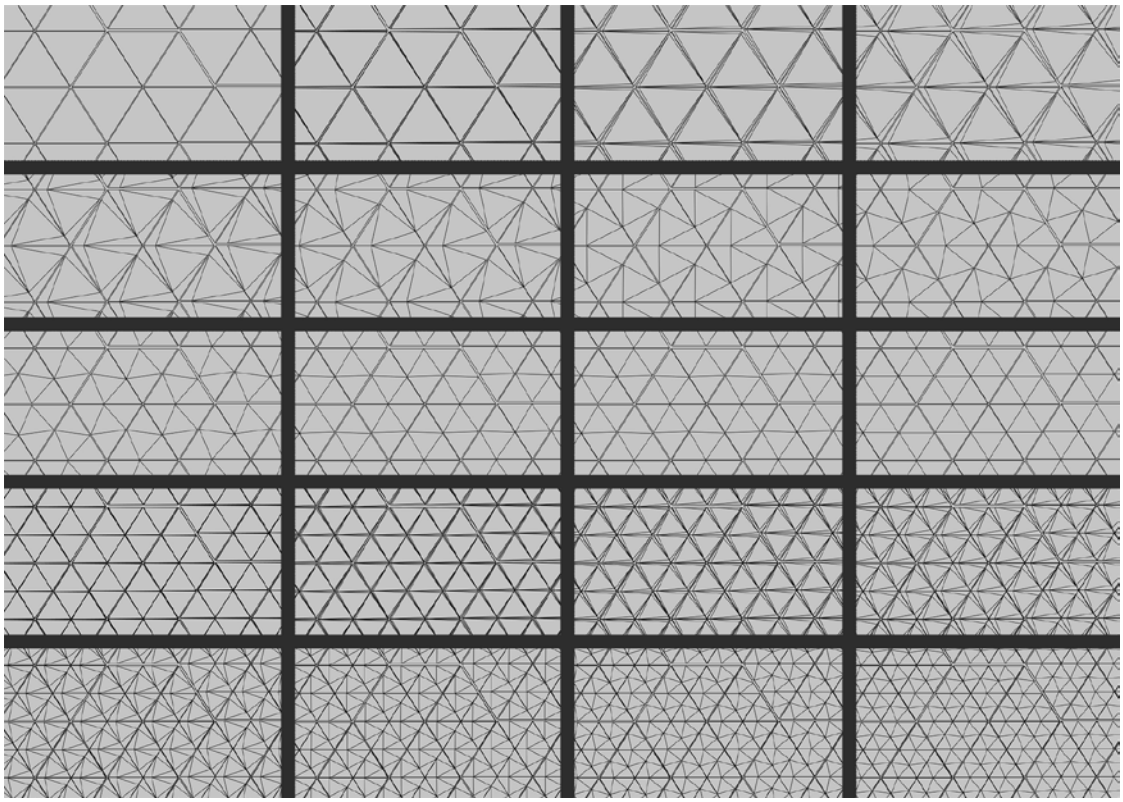
Various algorithms describe different approaches of handling it.

The ROAM algorithm for instance makes every triangle store pointers to its neighbors, and if a triangle is subdivided the neighbors need to be subdivided to at least one level below, before being stitched with the triangles vertices.

The chunked LOD algorithm meanwhile stitches the edges of its chunks, which are quadratic grids in a quad tree, similar to geo mipmapping, although the author of that algorithm also describes a method where instead the center vertex is lowered down to the next mipmap level as an extra option. While this method already lowers the amount of memory consumed to store the neighbors, it still requires, some which is why the last method caught my interest:

The Continuous Distance-Dependent LOD algorithm does not store neighbor positions at all, instead, the geometry is morphed at the edge of a subdivision level to resemble a lower subdivision. This is achieved by querying the distance for every second vertex in the vertex shader, and if the vertex is close to the edge of its subdivision level it starts moving towards its neighbor vertex, until it is completely merged. This completely resolves the necessity of storing the neighbors of every quad, and comes with the extra benefit of preventing popping at the edge of different subdivision levels, so the transition between subdivision levels is completely smooth, which can especially come in handy when moving from space to the surface and prevent problems with the terrain and show mapping appearing to jump around.

While CDLOD is based on quads, I also found a way of getting the same results with subdivided triangles, as can be seen in the following figure:



HEIGHT LOOKUP

One important part of any terrain renderer is displaying differences in height. With average terrain rendering one simply needs to change the position of the vertex on the vertical axis, and on a planet this is not a lot different, but instead the vertex needs to be scaled out along the normalized of the vector of its position.

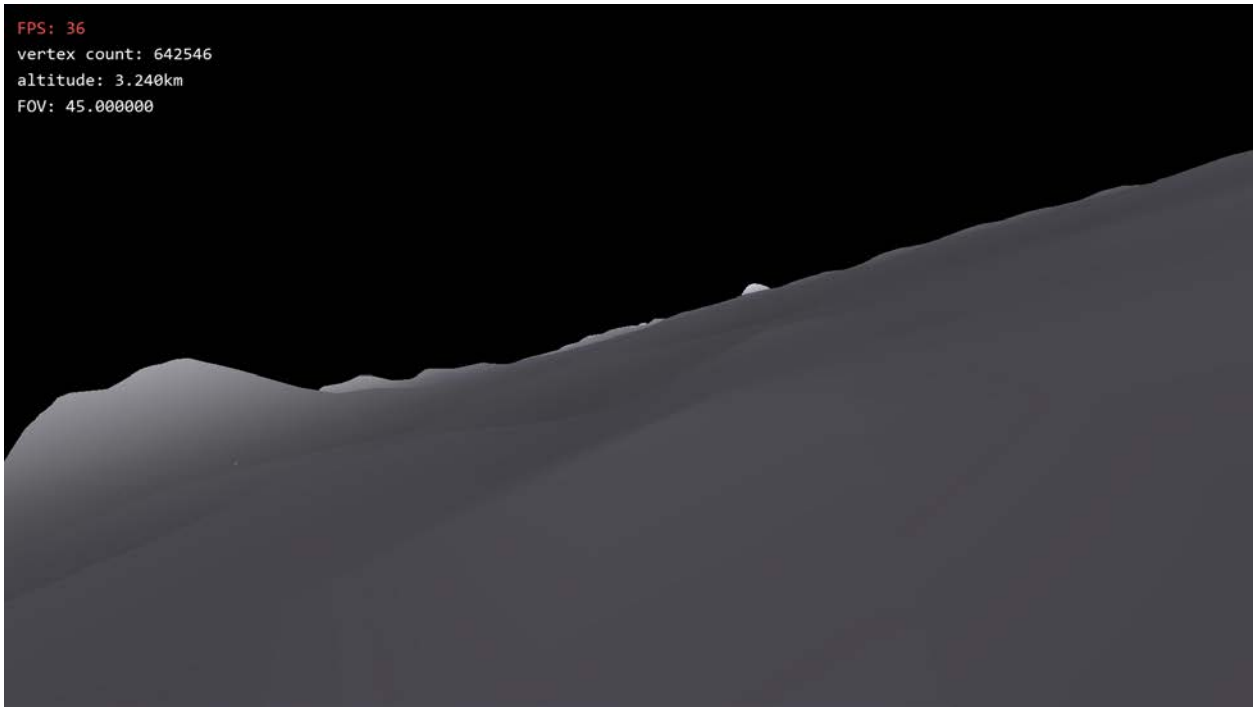
The height is queried from some stored data, and the technique differs between algorithms. A lot of the more sophisticated algorithms like Chunked LOD and ROAM 2.0 use some form of streaming in the data from the hard drive when it is relevant. This requires precalculating a massive amount of data into some hierarchy, and works quite well for less large terrain. For planets though the amount of data stored would need to be way to massive though, so at a certain point some of the data will be need to be calculated procedurally with an algorithm such as perlin noise or simplex noise.

Starting with shader model 3, it is possible to do texture look ups in the vertex shader on the GPU, so it is possible to use a single black and white height map texture to describe the height on a rather undetailed level and transform the vertex accordingly. Here is what the result looks like:

Transforming vertices in the vertex shader is quite performant because it utilizes the GPU's parallel computing powers, which is why it was recommended in the CDLOD algorithm. The height data can be streamed back to the CPU using transform feedback or by calculating it in a compute shader.

In order to allow mountains to peek over the horizon without being culled due to backface culling, an angle threshold needed to be added to the dot product look up table, which depends on the of the planet and the maximum terrain height, which looks something like this:

```
float cullingAngle = acos(planetRadius /( planetRadius + maxTerrainHeight));
```

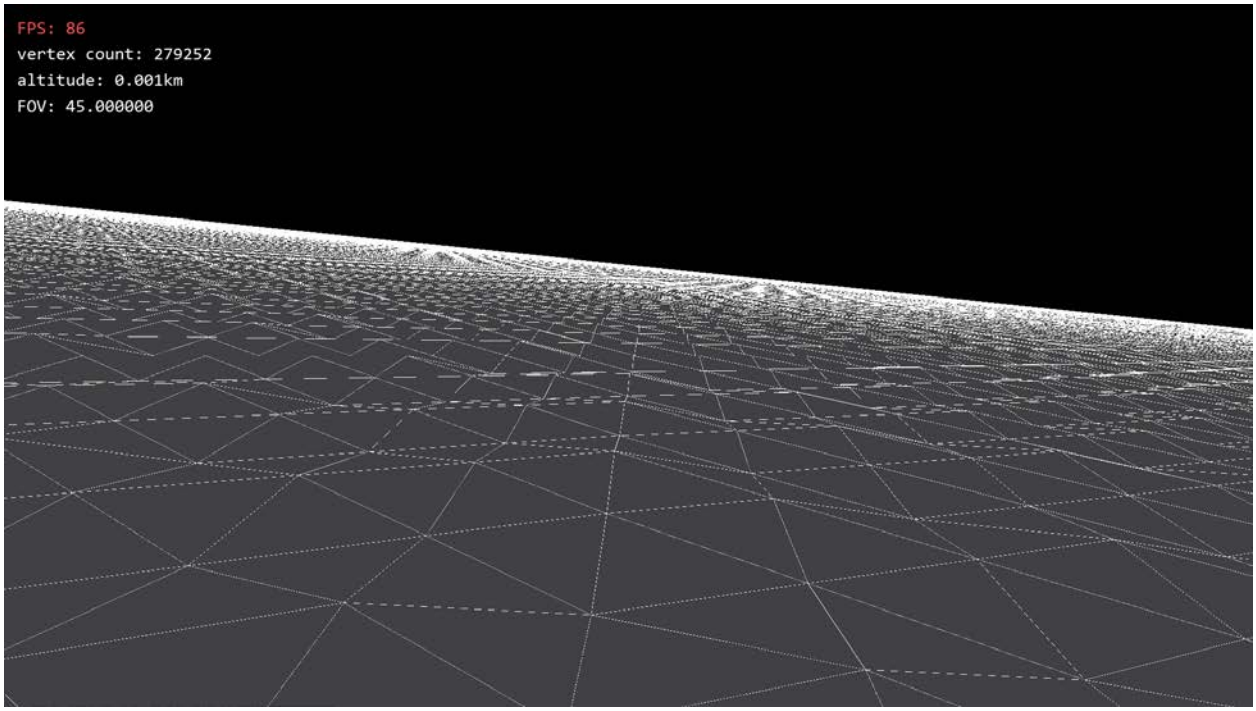


CONCLUSION

After evaluating various techniques, I believe the CDLOD technique might suit the purposes of planet rendering best, since it utilizes the GPU quite well, is quite easily modified to use procedural data and tessellation and has a memory friendly way of merging different LOD levels with a transition. Other algorithms fall short because they are either limited to static terrain or highly CPU and memory intensive.

My current implementation is based on equilateral triangles, which is great for precalculating some of the data to make the terrain generate faster because the size of the triangles is predictable at every subdivision level. As I showed morphing between patch level of details will also be possible for the CDLOD algorithm, so I will make an attempt to implement it based on triangle patches, and if that does not work due to higher computation it is still possible to switch to a cube based approach with a quadtree for all 6 faces. The height data will use a mixture of height map data for the larger details and procedural data to refine the details at a close distance.

One more problem that needs to be solved is floating point precision close to the ground. In my current demo, one unit represents a kilometer, and already generates precision problems close to the ground, and the problem only gets enhanced if the planet size increases. One way of solving this could be keeping the camera at the center of the world and moving the terrain instead. Also making calculations with double precision floating point numbers could solve a lot of those problems. This is what the terrain currently looks like right next to the ground on a perfectly spherical planet:



Also, some fancy shaders that interpolate between different levels of detail textures will be required, along with atmospheric scattering as described by Nishita or the version described Eric Bruneton.

SOURCES

- Geomipmapping: https://www.flipcode.com/archives/article_geomipmaps.pdf
- Roam: https://graphics.llnl.gov/ROAM/roam.pdf
- Roam 2.0: http://www.diva-portal.org/smash/get/diva2:21085/FULLTEXT01.pdf
- Chunked LOD: http://tulrich.com/geekstuff/sig-notes.pdf
- CDLOD: http://www.vertexasylum.com/downloads/cdlod/cdlod_latest.pdf
- Precalculated atmospheric scattering: https://hal.inria.fr/inria-00288758/document

FURTHER LINKS

- Blog: http://robert-lindner.com/blog/category/project/planet-renderer/
- GitHub code repository: https://github.com/Illation/PlanetRenderer