Procedurally generating planetary objects

Exploring the feasibility of isosurface extraction approaches

Kieran Sockalingam

2017

Abstract

This project explores the real-time procedural generation of asteroids and planets, with specific focus on the applicability of polygonization methods for implicit isosurfaces to this task. This report details the design and implementation of a system which is capable of drawing and shading arbitrarily large meshes which represent the isosurfaces of completely generalisable scalar fields, increasing and decreasing detail on parts of the mesh dynamically without introducing visual artefacts or rendering problems.

UNIVERSITY OF OXFORD

Contents

List of Figures

1 Introduction

This project explores one method to procedurally generate astronomical “space scenes” and render them at real-time framerates. The result is a software system that can generate an interactive, navigable 3D scene, composed of a background and several foreground asteroid and planet objects, such as Figure 1. Each element of the scene is defined procedurally, meaning that it is generated algorithmically at runtime, rather than using stored textures, meshes, or other data.

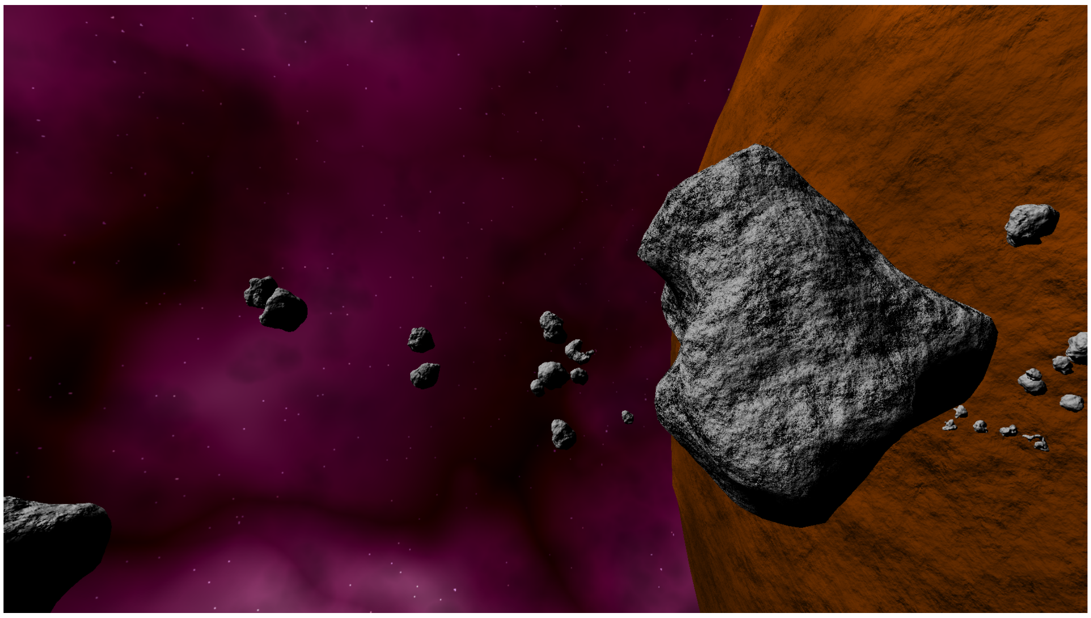


Figure 1: Sample space scene

1.1 Applications

The real world applications of rendering space scenes include creating content for film and television media, video games, simulations or visualisations. The focus of this work is on real-time procedural generation, and so it is relevant to situations where real-time framerates are required, such as video games or interactive visualisations – whereas film and pre-recorded television media can utilise much slower “offline” rendering techniques. The procedural content generation is configured to generate fictional asteroids and planets, but the same principles could be used to model real-life bodies for scientific visualisation.

1.2 Learning goals

On a personal level, I embarked on this project with the aim of learning modern shader-based OpenGL, and about GPU hardware, GPGPU and procedural generation. This is reflected in my decisions to use technologies, languages and techniques which are more complex, powerful, and offer more control, at the expense of ease of development, feasibility in a finished product, or performance, due to the increased learning opportunity.

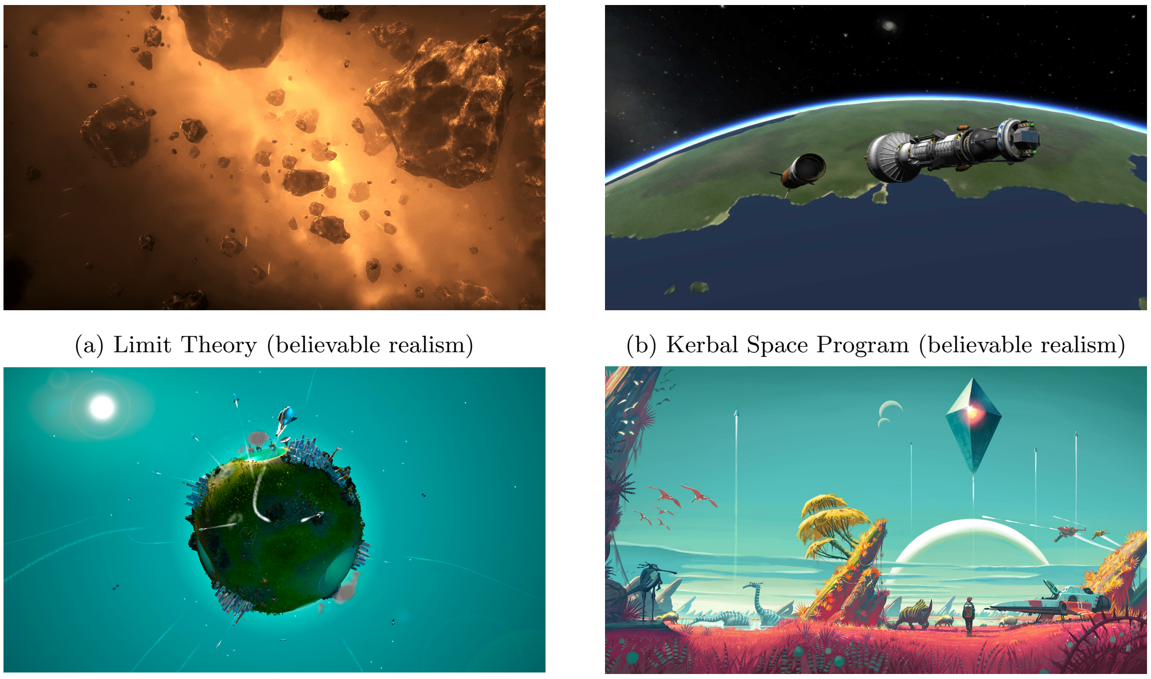
1.3 Areas of interest

Bearing in mind the aforementioned learning goals, the main technical challenges which must be overcome, and the real world requirements for such a system to be useful, there are three key areas of interest with associated goals. They pertain to level of detail scaling, aesthetics, and simplicity, as outlined below.

Level of detail scaling The first area of interest is concerned with level of detail scaling. It is relatively straightforward to create a system that can render astronomical bodies such as asteroids and planets from afar, with the camera viewing from space, or to create one that can procedurally generate flat terrain for surface exploration, due to the constraints that these scenarios introduce. When the camera is very far from a body, small surface features do not need to be modelled, and when the camera is restricted to the surface, the entire visible terrain can be approximated as a flat plane. It is much more difficult, however, to maintain visual quality as the camera moves seamlessly from space to the surface or vice versa, because the memory necessary to load the entire surface at the level of detail required for surface viewing is infeasible. This project presents a solution to this issue, by dynamically increasing the level of detail on the relevant part of the surface as the camera moves closer, aiming to maintain real-time frame rates (24+ frames per second).

Aesthetics Regarding aesthetics – the look and appearance of the resulting scene – the target was visual appeal rather than physical realism. The aim was not to generate models of realistically sized planets or asteroids, or to compute physically accurate lighting and colours. This is because the area of interest here is in procedural generation, rendering and the associated technical challenges rather than the physical rules governing real life astronomical objects. Instead, the goal is visual appeal - that the images look attractive - and visual or believable realism which is that the image is realistic enough that the viewer believes that the image could be real. [Chalmers and Ferko 2008] "Limit Theory" [Parnell 2012] or "Kerbal Space Program" [Squad 2015] are examples of believable realism, whereas "No Man's Sky" [Hello Games 2016] or "Universim" [Crytivo Games 2017] exhibit cartoon-like or scale distorted looks. In particular, in order to look convincing, the surface needs to accurately convey the appearance of the appropriate material – e.g. rock – and a convincing shape. My previous work in this area, and indeed similar commercial software (such as Kerbal Space Program) suffers from a lack of variety of shape, with all objects being only roughly distorted spheres. Further chapters explain how the use of a more complicated approach to mesh creation, based on isosurfaces (a surface containing all the points at which a scalar field has some fixed value), allows incredible diversity and flexibility of the mesh shape. Furthermore, the implementation chapter demonstrates how shadow-mapping techniques can be used to enhance the appearance of the shape and highlight the interesting topologies that can be realised.

Simplicity (of procedural content definitions) Finally, procedural generation allows for complete customisation of the resulting images, so it is desirable for the system to be extremely flexible, with changes to the style, scope and effect of resulting images requiring


(a) Limit Theory (believable realism) (b) Kerbal Space Program (believable realism)

(c) Universim (not believable realism) (d) No Man's Sky (not believable realism)

Figure 2: Believable vs non believable realism

only a small adjustment to the code. A successful implementation would allow for a strong separation of concerns between code that defines the shape, colour, and other properties of the final scene, with the code responsible for generating the scene itself and managing the graphics hardware. Ideally, the code describing the procedural content will be extremely expressive, minimalistic and elegant, greatly increasing the usefulness of the system as it can be applied to new use cases easily. As the implementation chapter explains, the difficulty in implementing a system to meet these requirements lies in the need to make use of both the CPU and the GPU in order to achieve the desired performance. CPU and GPU code are very different and are written in different languages so a naïve implementation requires describing the same procedural content twice, which must be avoided to achieve the goal of simplicity. This issue was overcome by making use of GPGPU techniques. The system that has been created allows for procedural content to be defined in an easy to use, elegant language, loaded and compiled at run time, while the renderer is implemented in performant (statically-compiled) C++.

1.4 Overview

The remainder of this document consists of chapters covering literature, design, implementation, results, and discussion, structured as follows.

One of the current state-of-the-art projects relating to planetary/asteroid rendering is a promising but unreleased project called "Limit Theory" [Parnell 2012], which is a closed-source video game. In a technical blog post, the author mentions that his asteroids are created using an isosurface extraction algorithm called SurfaceNets [Gibson 1998]. This inspired a focused look at isosurfaces, and eventually the decision to use them to implement this system. Despite some limitations regarding performance, isosurfaces have a natural fit with the above goals. Isosurfaces could be used to improve upon many of the existing, commercial video games by allowing for much more diversity in the topology that can be represented, such as surface features like cliffs, caves, and overhangs. More detail about this decision process is provided in the literature chapter, first broadly describing different approaches to planetary generation and rendering, and then comparing different isosurface extraction techniques.

The description of the process of designing and implementing the planet renderer system occupies the bulk of this document. The implementation is capable of taking an arbitrary definition of an isosurface (i.e. a scalar field – more on this in section 2.3), generating a mesh which represents the surface using an isosurface extraction algorithm called Dual Contouring [T. Ju et al. 2002] and then accurately shading it, exposing position and normal information to customisable fragment shaders, including support for dynamically increasing detail to allow for rendering huge meshes. Asteroids are rendered with an accurate self-shadowing system, enhancing their interesting topology. The system is very flexible and can accept any kind of shape, without limits on topology. The details of how the system fits together, including how Dual Contouring is employed to create the meshes, and how accurate lighting can be calculated from the same mathematical definition of the surface that defines its shape, is explained in the design chapter.

The implementation of the system shall be detailed in two distinct stages. The first stage focuses on rendering asteroids, and it will explain the method used to generate a mesh and shade it with appropriate detail, including self-shadowing techniques to enhance topology. This approach will allow the explanation of the system, addressing the aesthetic and simplicity goals, while first overlooking the increased complexity that comes with level of detail scaling. The latter stage focuses on rendering planets, and explains how the approach can be extended to apply to significantly larger bodies, addressing level of detail scaling.

The successes and shortcomings of the project, with reference to the goals set forth above, shall be elaborated upon in the results and discussion chapters, which will also discuss opportunities for future development that have arisen throughout the project, and provide some quantitative and qualitative analysis of the results.

2 Literature

Any method to procedurally generate a planetary surface must produce the same end result: geometry, in the form of a set of triangles, and a shader program to define the colour of fragments as they appear on the screen, because this is the input format that the GPU requires to perform the rendering process. Traditionally, the intention behind this data structure is that the mesh will be present in the geometry of the scene, and then the shader program will be used to perform lighting calculations.

2.1 Quadtree heightmap methods

The most popular approach that is used to draw an asteroid or planet is to create geometry by extruding points on the surface of a sphere along the normal. It is useful to start with a sphere because the topology is simple and thus it is easy to provide the indices that connect the vertices into triangles. The downside is that the topology of the resulting mesh is limited – it can only represent rough spheres. It is not possible for a heightmap approach to represent more complex shapes, such as vertical cliffs or caves, because the surface is limited to just one height at each location, as shown in Figure 3.

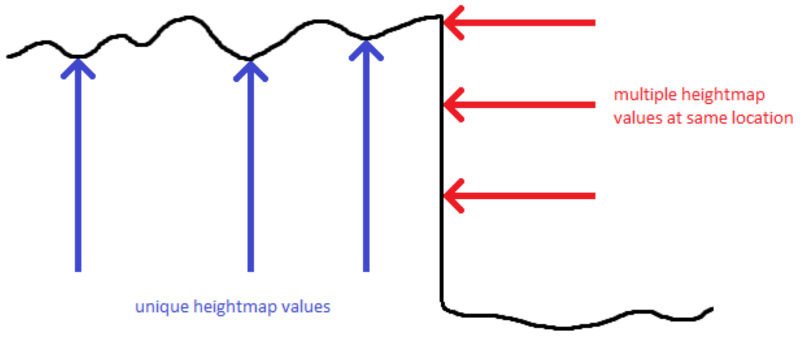


Figure 3: Heightmaps cannot represent cliffs, caves, overhangs etc.

The most common way to implement LOD scaling with this approach is to generate the mesh in a cube shape first, as the cube is easily subdivided into patches that can be tessellated at different levels of detail. Typically, each of the six faces is constructed from patches which are organized in a quadtree, so patches can be subdivided for more detail or merged together to reduce detail. Before drawing, the points are normalized, forming a sphere, and extruded according to a heightmap.

2.2 Implicit methods

In recent years, the performance of graphics cards has increased exponentially, and it is slowly becoming more feasible to skip mesh creation entirely, and instead to draw two triangles which cover the screen and use the fragment shader to perform all the calculations necessary to draw something.

If a shape is defined implicitly, such as by an isosurface, then raytracing or raymarching can be used in the fragment shader to draw the shape. For a particular scalar field function $f(x, y, z)$, an isosurface with isovalue k is defined as the surface containing all points at which $f(x, y, z) = k$.

The advantage of these methods is that they do not have to approximate a shape with a polygonized mesh, allowing for pixel perfect smooth curves or complex topology. Furthermore, the definition of a shape through an implicit function is extremely flexible and general. The main disadvantage is that these methods are very computationally intensive. The fragment shader is executed for every pixel on every frame, storing nothing between frames, meaning that the shape is recomputed on each frame, whereas it would only have been computed once using a mesh. The GPU hardware is also highly optimised for rendering a mesh.

2.3 Isosurface extraction methods

In this section, approaches for creating traditional meshes based on isosurfaces will be explored. These methods combine the advantages of defining the shape of the mesh with an implicit function with the advantage of the speed of rendering a polygonised mesh.

An isosurface extraction algorithm is a method for constructing a mesh of polygons to approximate an isosurface. Generally speaking, the isosurface at which $f(x, y, z) = 0$ is generated, so a simple transformation can be used to generate an isosurface with a different isovalue.

In theory, a scalar field would usually be continuous and defined at every point in a given interval. For practical purposes, the data must be sampled at discrete points, and depending on the kind of data that is being used, it may only be available at discrete points, e.g. medical scan data. Different algorithms sample the data in various ways, but both the algorithms that shall be discussed use the data as sampled on a structured, uniform grid. Thus, it is useful to imagine the data as an array of cubes, with the values of the scalar field tagged on the cubes at their vertices. It is common to interpolate with a trilinear function to define values for the scalar field within the cube. The following is a comparison of two of the most popular algorithms for extracting a polygonised mesh representing an isosurface of a given scalar field.

2.4 Marching Cubes

In many ways the original, and perhaps the most well known of all the algorithms that have been developed to solve this problem is Marching Cubes. Introduced in 1987 by Lorensen and Cline in SIGGRAPH [Lorensen and Cline 1987], it was the first of many similar algorithms, which can all be considered to build upon it. Marching Cubes takes a simplistic divide and conquer approach, breaking the problem down into a separate subproblem for each cube – neighbouring cubes do not affect each other's resulting geometry. Thus the algorithm is easily implemented in a fully data-parallel manner, which is a very useful when implementing Marching Cubes for a parallelized system such as a GPU [Smistad et al. 2011], where parallel computation units perform the same instructions on different data in parallel very efficiently.

For each edge that exhibits a sign change on each cube, linear interpolation is used to place a vertex at the approximate location where the field takes a zero value. These vertices are joined together to form the triangles to represent the surface within the cube. Unlike the positions of the new vertices, the way in which they are connected together only depends on the signs of the scalar field at each corner of the cube, and not the exact value. This means there are only $2^8 = 256$ cases, making a case-table feasible and efficient. For each cube, the value of the scalar field at each of the eight vertices is checked to determine whether the vertex is above (outside) or below (inside) the isosurface. Using these eight binary values as an index, the polygon topology information is retrieved from a 256-value lookup table. The table describes which of the new vertices should be connected together into triangles.

In the original paper, Lorensen and Cline make use of two different symmetries to reduce the number of cases for which the triangulation must be calculated by hand. They note that the topology is the same when the vertices which are above the surface are exchanged with those that are below the surface, thus reducing the possible cases to 128. The number of cases is further reduced to just 15 by considering rotational symmetries by hand.

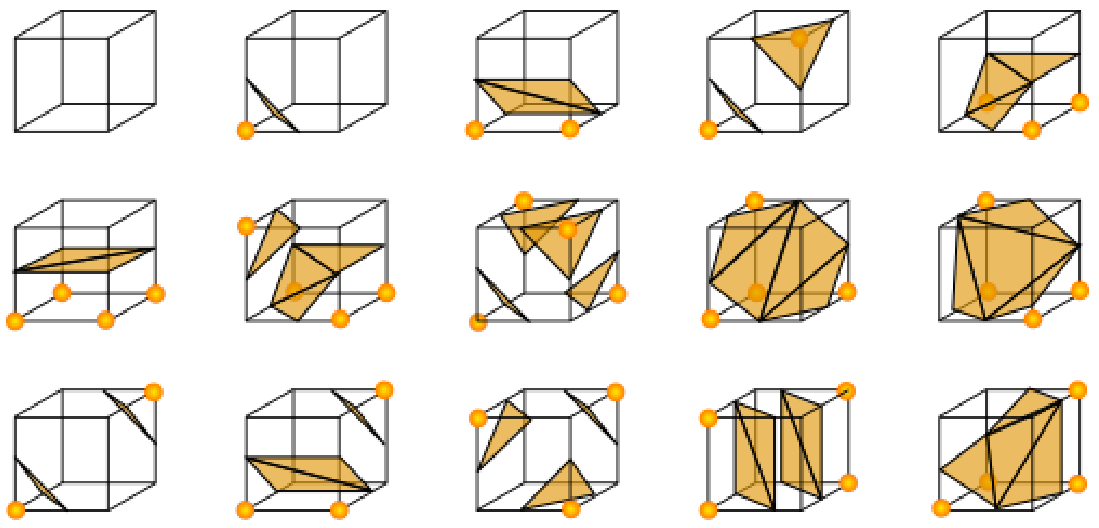


Figure 4: Marching Cubes case table [this version is a remake of the original and was released online under GPL]

Figure 4 is a visualisation which depicts the 15 cases. The most trivial case is the first, where none of the vertices are underneath the surface. There is a single case when only one vertex is included; this can easily be rotated to the relevant position. There are three cases where two vertices are included; when they share a line, when they are diagonally opposite on a face, and when they are opposite on the cube, and so on for cases with three and four vertices included. For cases where more than four vertices are included, symmetry is used.

The standard 15 case table has been criticized and improved upon by several authors, because its simplicity can cause at least two main problems. The first, and perhaps most serious, is that there is a possibility for holes to appear in the mesh due to vertices on a shared face being connected differently in two adjacent cells. An example of this is shown in Figure 5, where the shared face has the new vertices connected differently in each cube.

The second problem is that the behaviour of the field within the cube is not considered when selecting from the case table, so there are ambiguities in some of the cases. For example, in the

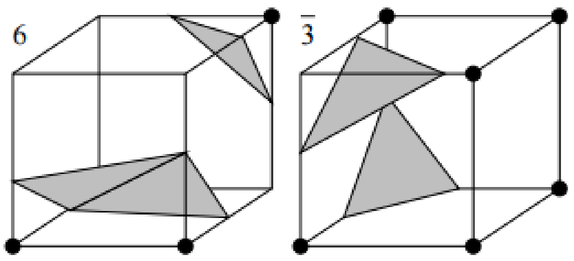


Figure 5: Marching Cubes hole issue [Chernyaev 1995]

case where two diagonally opposite vertices are below the surface (first case on the last row in the above diagram), the two regions may or may not meet within the cube. In Marching Cubes 33 [Chernyaev 1995], this internal ambiguity is solved by treating the behaviour of the field inside the cube as the trilinear function between the values at the vertices, and constructing a 33 case table that, with extra computations, creates a mesh which is topologically equivalent to the trilinear function.

The case table can also be extended in other ways, to solve other issues: Raman and Wenger index a table using three possible labels for each vertex, differentiating the case when the vertex has a value equal to the isovalue, resulting in $3^8 = 6561$ entries. This allows their version of the algorithm to avoid generating small or zero area triangles, short edges and small angles, resulting in a cleaner, more efficient mesh [Raman and Wenger 2008].

2.5 Dual Contouring (of Hermite Data)

There have also been many improvements to the core mechanics of the Marching Cubes algorithm since the 1987 paper. There is a newer class of algorithms which are so-called 'dual' methods, as opposed to the original 'primal' methods [Schaefer and Warren 2002]. Traditional primal algorithms generate one or more polygons inside each cube that intersects the surface, with the vertices lying on the edges of the cube. A dual algorithm is one which places a vertex within each cube, not necessarily on an edge, and makes a polygon connecting the four new vertices in cubes which share a given edge when that edge exhibits a sign change. It thus generates a mesh that is dual to the mesh generated by a primal method in the sense that vertices and polygons are interchanged.

Dual methods have certain advantages. Firstly, the mesh polygons are more evenly sized, because the vertices roughly correspond to a regular grid, whereas Marching Cubes can produce both large and small polygons. Secondly, the resulting mesh better represents the intended isosurface because there is more freedom in the placement of vertices, which can be exploited by carefully picking the best location inside the cube. Primal methods also tend to create visible grid-like structures in the resulting mesh due to the constraint that vertices must be on the edges of the grid.

One of the earliest of the dual methods is SurfaceNets [Gibson 1998], which places the vertex within each cube at the centroid of the intersection points on the edges that would have been generated by Marching Cubes.

Dual Contouring, the algorithm introduced in Dual Contouring of Hermite Data [T. Ju et al. 2002], makes changes to the way the scalar field data is represented. Instead of a flat grid, the method uses an octree structure (a tree structure where each node represents a cubic space and has eight children to represent the eight smaller cubes that fit into it with half the side length). The algorithm requires that the scalar field it operates upon is “hermite data”, which means that the normal at a given point is also available. This can be provided analytically, or by simply calculating a numerical approximation to the normal by evaluating the scalar field at multiple points with a small offset. The normal data is used to preserve sharp features such as edges and corners in the resulting mesh. This is done using a technique whereby the vertex is placed in the location that minimizes a Quadratic Error Function (QEF). The QEF is constructed based upon not only the position of the intersection points, but also the normal of the scalar field at those points. This extra information means that features within the cube are preserved much better than by Marching Cubes, resulting in the distinctive sharp edges which are smoothed-out by Marching Cubes.

Each intersection point and its respective normal define a tangent plane to the isosurface. The QEF is the sum of the distances to each of the planes squared, represented equivalently as a function of the intersection points and the normal as:

$$E[x] = \sum_i (n_i \cdot (x - p_i))^2$$

where $E[x]$ is the QEF at a point x, and (p_i, n_i) are the (intersection point, normal) pairs.

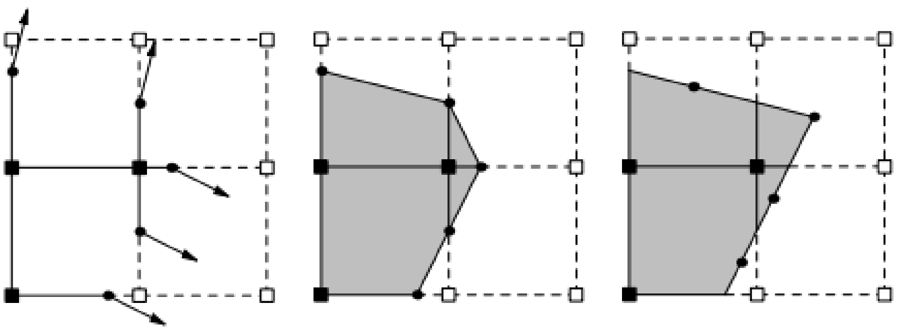


Figure 6: Dual Contouring example [T. Ju et al. 2002]

Figure 6 is a 2D example, showing the scalar field data (left), with the Marching Cubes contour (middle) and the Dual Contouring contour (right). In the upper-right cell, Dual Contouring captures a sharp corner feature that Marching Cubes fails to represent accurately, due to the placement of the vertex so as to be as close as possible to the position where the extension of the incoming tangent lines would meet.

Since its publication, Dual Contouring has also been extended, modified, and improved, with versions of the algorithm offering various advantages, such as Manifold Dual Contouring [Schaefer, Tao Ju, et al. 2007], which better preserves the manifold nature of the surface, or Dual Marching Cubes [Schaefer and Warren 2004], which produces meshes of comparable quality with far fewer polygons by representing the scalar field data on a more flexible grid that is dual to the structured grids used by Marching Cubes and Dual Contouring.

To summarise, Dual Contouring is a far more complex algorithm than the original Marching Cubes, and it reproduces the desired isosurface much more accurately. This makes it ideal for use in applications such as visualising medical scan data, where the accuracy of the visualisation is critical, and the processing time is less of a concern. However, the older Marching Cubes algorithm still finds use today, due to its simplicity, ease of implementation on parallel graphics hardware, and its fast execution time using lookup tables. For real-time applications, even recreating the mesh from scratch every frame is much more feasible with Marching Cubes. In applications such as computer games, where the isosurface mesh can be used to model a variety of different objects procedurally, or metasurfaces like liquids, the accuracy of the mesh is not critical, whereas speed of execution is paramount, so Marching Cubes would be a good choice for a commercial product. However, in this context, the aim is partly to explore the feasibility of the more complex algorithm. Furthermore, it is desired to create a flexible system that can perform well with any scalar field, including those which Marching Cubes fails to accurately represent. For these reasons, the system described in this project implements Dual Contouring.

3 Design

The software system developed in this project (hereafter "the system") will draw one or more asteroids and planets by using a scalar field and Dual Contouring to procedurally generate a mesh, and then use the scalar field to shade the mesh appropriately for visual realism. This section will discuss how to build up an interesting scalar field, and the mathematics necessary to shade the mesh appropriately.

3.1 Scalar fields

The scalar field will be a function of the form $f :: R^3 \rightarrow R$, where the surface of the asteroid will be defined as the surface for which $f(x, y, z) = 0$. The scalar field must be continuous for Dual Contouring to work, but isosurface extraction algorithms typically do not distinguish between the inside and the outside of the mesh, so whether the inside is positive or negative is not important. By changing the function, the shape of the generated asteroid can be controlled, and by introducing deterministic pseudorandom components, such as procedural noise, unique asteroids can be created from each pseudorandom number generator seed value. It is useful to think of the scalar field as being a "signed distance field", where the value of $f(x, y, z)$ is the signed distance between (x, y, z) and the closest point on the mesh, because this can make it easier to understand the link between a given shape and its relevant function, but it is not necessary for the scalar field to have this property. For example, $f(x, y, z) = y - k$ would be a plane parallel to the xz plane, passing through $(0, k, 0)$ (Figure 7). $f(x, y, z) = length(x, y, z) - k$ would be a sphere at the origin with radius k (Figure 8).

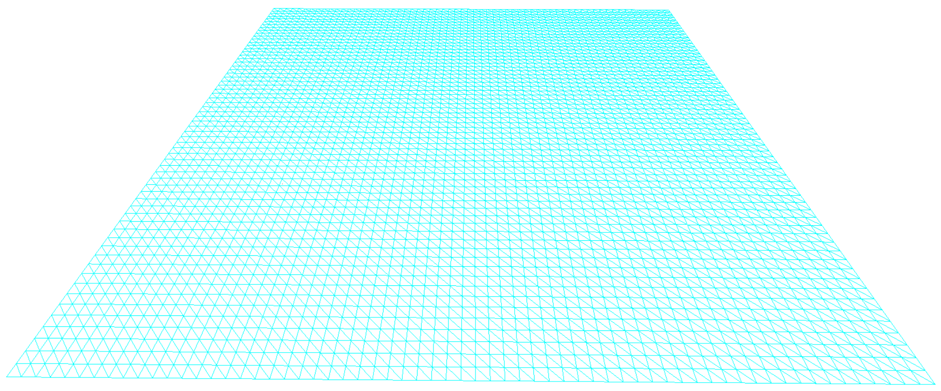


Figure 7: f(x,y,z) = y - k

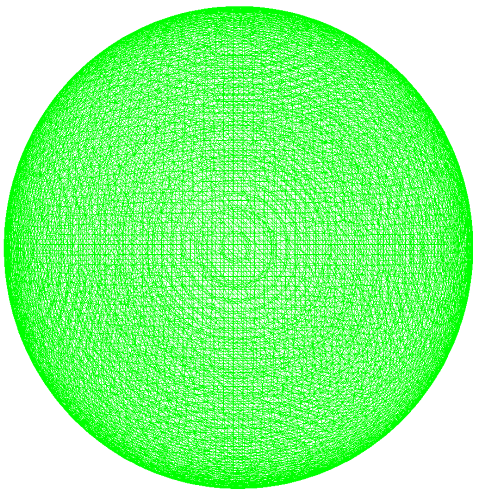


Figure 8: f(x,y,z) = length(x,y,z) - k

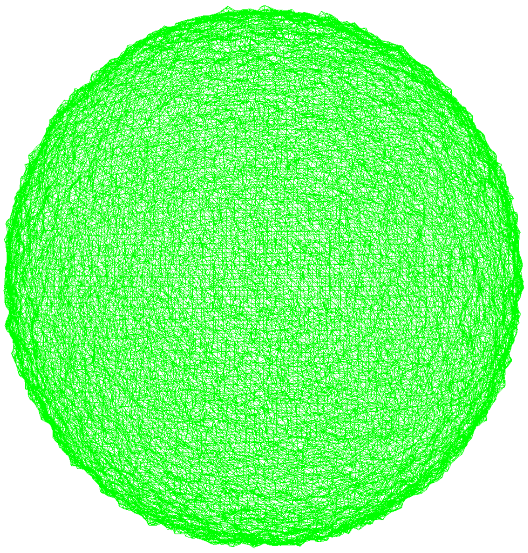


Figure 9: Rough sphere

3.2 Noise

Procedural noise fields are arguably one of the most important and popularly used building blocks for creating procedurally generated content. A noise function deterministically assigns a pseudorandom value to each point in a space in a continuous manner, making noise much more useful than ordinary random numbers for creating content that mimics elements of the natural world, such as textures. Different noise functions create values that differ in their characteristics, and can usually be generalised to n dimensions, with 2D, 3D, and 4D frequently used in videogames. Perlin noise [Perlin 1985] is a very popular kind of gradient noise, meaning that the algorithm computes gradients at fixed points in a lattice structure, and interpolates some function of the gradients to obtain noise values between the lattice points. The system shall make use of Simplex noise [Perlin 2002], which improves over the original Perlin noise by reducing directional artefacts and improving performance [Gustavson 2005]. By combining Simplex noise with the scalar field function for a base shape, a "rough" version of the shape can be created, with pseudorandom perturbations in the surface (Figure 9). A common technique for increasing the level of detail when using noise is to create a "fractal" version, where several layers, or octaves, of the noise are combined, each at a progressively increasing frequency. Using this, Simplex noise can be used to create a texture that closely resembles rock. If all the octaves used have high frequency and low amplitude, then the shape doesn't deviate from the base sphere too much, producing planet-like bodies (Figure 11). Low frequency octaves can be used to create shapes that are very different from the base sphere, suitable for asteroids (Figure 10).

3.3 Lighting

The most popular method for shading a surface for realistic lighting is the Phong reflection model [Phong 1975]. It consists of three components, which are combined to produce the final shading: ambient, diffuse and specular reflection. For a given material, define constants:

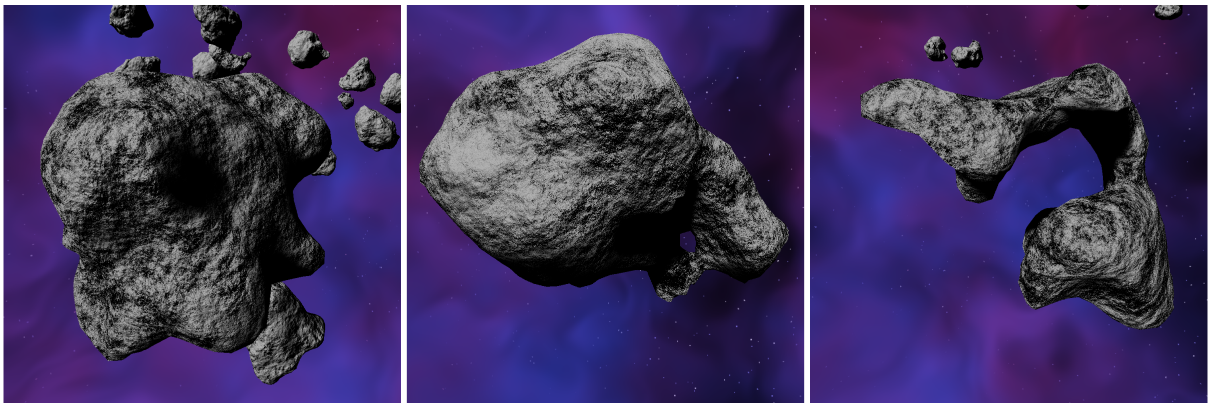


Figure 10: Asteroid shape diversity

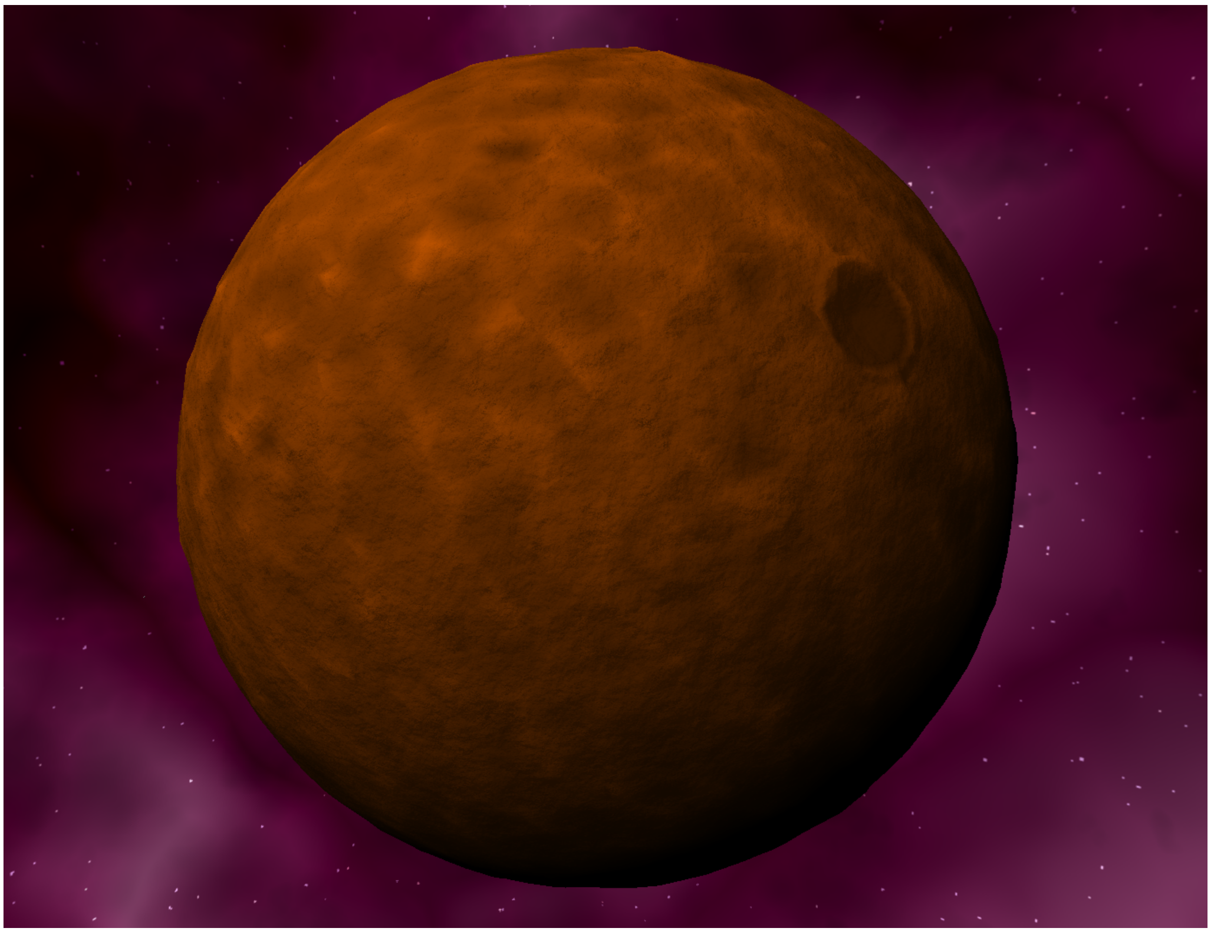


Figure 11: Planet

- K_s (specular reflection constant)
- K_d (diffuse reflection constant)
- K_a (ambient reflection constant)
- α (shininess)

Then, the illumination I_p at a point p is given by:

$$I_p = k_a i_a + \sum_{m \in lights} (k_d(\hat{L_m}\hat{N}) i_{m,d} + k_s(\hat{R_m}\hat{V})^{\alpha} i_{m,s})$$

where:

- $lights$ = set of all lights
- i_a = ambient intensity
- $i_{m,d}$ = diffuse intensity for each light m
- $i_{m,s}$ = specular intensity for each light m
- $\hat{L_m}$ = direction from point on surface to light m
- $\hat{N}$ = normal
- $\hat{R_m}$ = reflection direction
- $\hat{V}$ = direction from surface to viewer

$\hat{R_m}$ can be computed using $\hat{L_m}$ and $\hat{N}$:

$$\hat{R_m} = 2(\hat{L_m}\hat{N})\hat{N} - \hat{L_m}$$

So for each point on the surface that is to be shaded, $\hat{L_m}$, $\hat{N}$, and $\hat{V}$ are required to compute lighting. Section 4.6 shall explain how these values are exposed to the shader.

3.4 Shadows

A mesh shaded by the Phong reflectance model will be darker on faces that are not facing the light source, but still does not look correct because faces that are facing the light source will be lit regardless of whether or not they are in fact occluded by other parts of the geometry. In order to accentuate the interesting topology that the Dual Contouring approach is capable of, it is important to correctly account for occluded faces. The implementation of shadow mapping [Williams 1978] techniques to account for self-shadowing is described in the implementation chapter. The resulting asteroids look realistic (Figures 10, 12, 13).

3.5 Background

Simply drawing the asteroids onto a black background achieves a fairly realistic look, as most of space is simply black in reality. However, it is a desirable feature of the system to have the same level of procedural control over the background, in order to achieve different visual effects. For example, a physically realistic look may include background stars, while a fantasy theme may include exaggerated, colourful nebulae. Bright and interesting colours have been introduced into the backgrounds for these screenshots in order to emphasise that the shadows hide part, or even all of an asteroid, depending on the point of view. For example, an image like this (Figure 12) would be almost completely black.

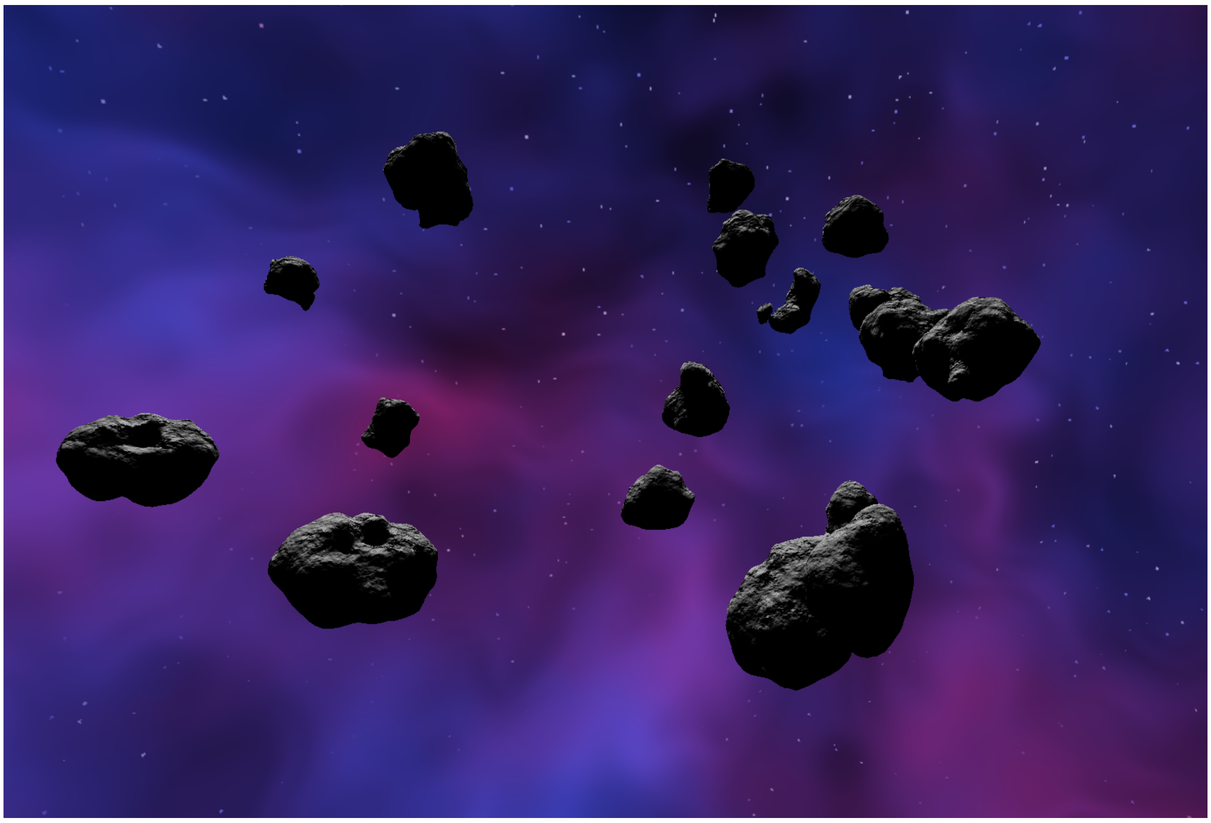


Figure 12: Procedural backgrounds

The screenshot in Figure 13 contains a procedural background defined by the following shader, which demonstrates some techniques that can be used to create interesting procedural shapes and effects.

```
void main()
{
    /* Interpret input uv_frag as a direction vector uv */
    vec3 uv = normalize(uv_frag);
    /* pxcol will contain the final pixel colour */
    vec3 pxcol = vec3(1.0);
    /* Perturb uv with fractal simplex noise to use as an address for other noise */
    vec3 addr = vec3(uv.x + fractalNoise3(uv * 2.0), uv.y + fractalNoise3(uv * 2.0),
        uv.z + fractalNoise3(uv * 2.0));
    /* baseCol contains the desired main colour of the background */
    vec3 baseCol = vec3(1.0, 0.5, 0.8);
```

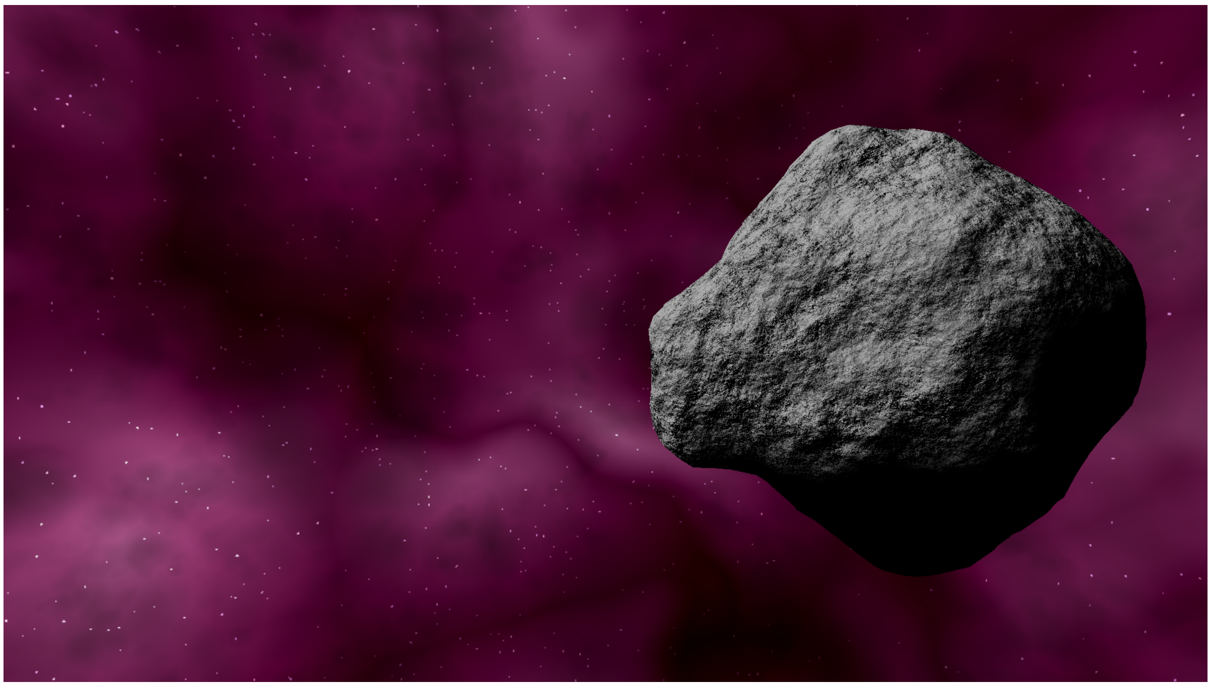


Figure 13: Procedural backgrounds

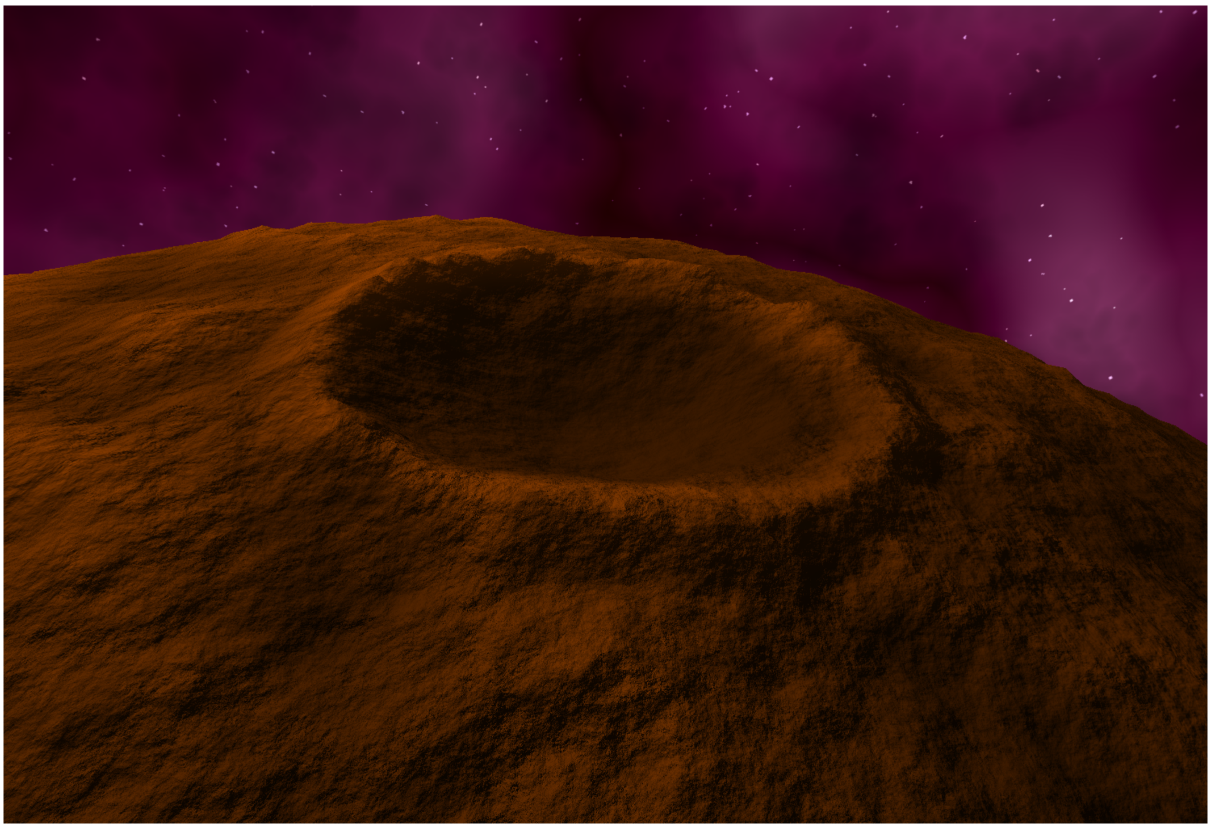
```
    /* use voronoi noise to create dark and light patches */
    pxcol = baseCol * 1.0-voronoi(addr*2.0).x;

    /* use raymarching to create a simple volumetric fog */
    vec3 ro = uv;
    float fog = 0.0;
    for (float d = 1.0; d <= 100.0; d += 1.0) {
        float sample = fractalNoise3(ro * d * 0.1);
        fog = max(fog, sample);
    }
    pxcol = pxcol * (1.2-fog);

    /* stars */
    /* starCol will contain the colour of the star */
    vec3 starCol = vec3(0.0);
    /* create discrete coordinates so stars are larger than fragments */
    float starfactor = 1024.0;
    float pX = floor(uv.x * starfactor / 2.0);
    float pY = floor(uv.y * starfactor / 2.0);
    float pZ = floor(uv.z * starfactor / 2.0);

    /* generate a pseudorandom value */
    float salt = hash(pX);
    float toHash = 17.0*pX + 7.2*pY + 9.3*pZ;
    float val = hash(hash(toHash + salt));

    /* colour the star differently according to the desired distribution */
    if (val > 0.99992) {
        starCol = vec3(1.0, 0.8, 0.8);
    }
    else if (val > 0.99900) {
        starCol = vec3(0.8, 0.8, 1.0);
    }
    else if (val > 0.99865) {
        starCol = vec3(1.0);
    }

    /* add the star into the pixel colour */
    pxcol += starCol * max(1.5*length(pxcol), 0.0);

    /* shader output */
    color = vec4(pxcol,1.0);
}
```

Voronoi noise is used to create light and dark patches using its cell-like structure, and by first perturbing the address used to index the voronoi noise field using fractal simplex noise, a swirling or distorting effect is achieved. The stars are created by generating a pseudorandom, deterministic floating point number and comparing it to various threshold values, adding a star colour if necessary.

3.6 Surface features

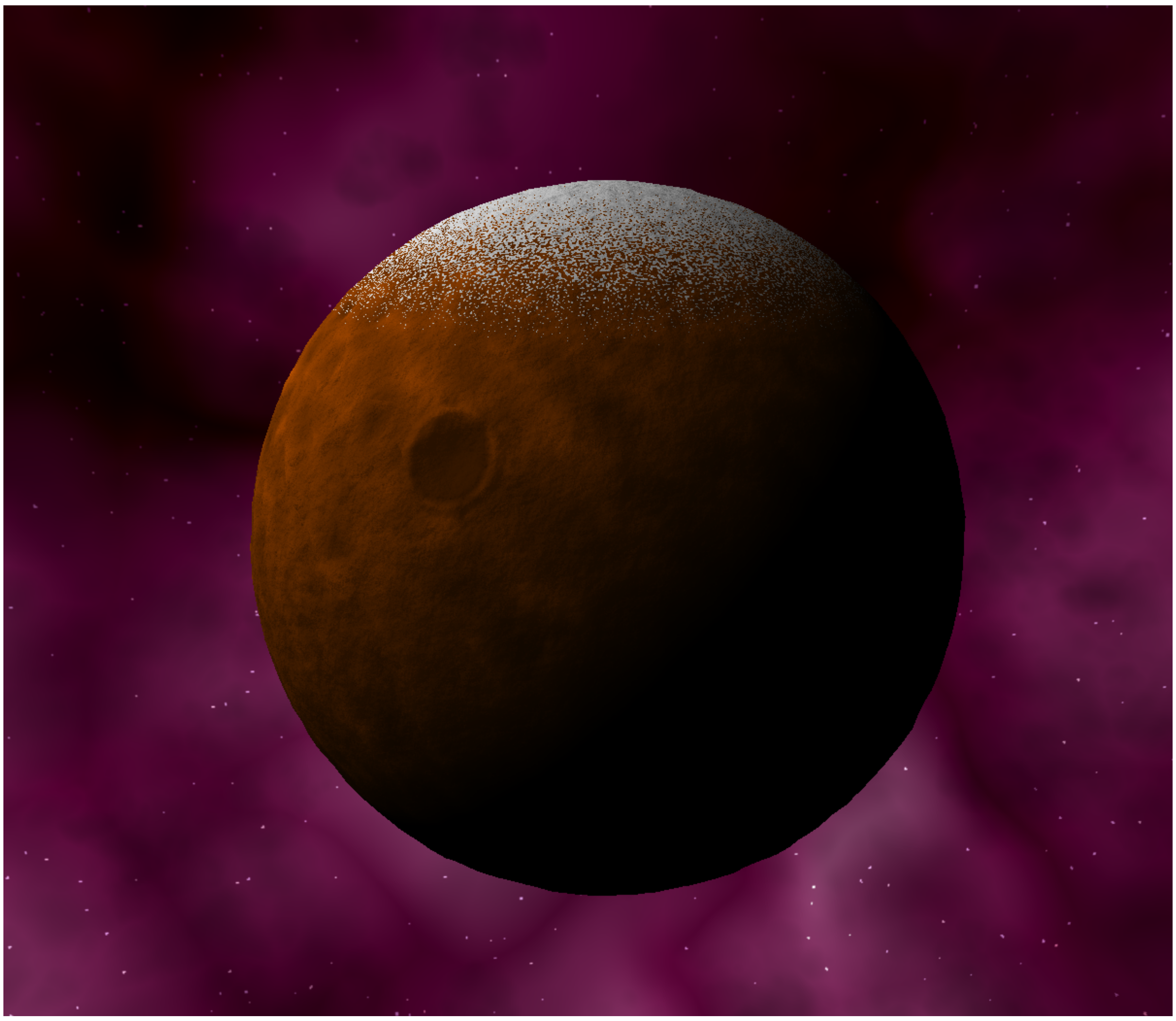
It is extremely easy to add features to the surface of the planet using the scalar field definition. For example, with very small changes to the code, craters can be added (Figure 14).

The appearance of the planet can be easily customised by making small changes in the fragment shader. For example, it only takes a small addition to add polar ice caps (Figure 15).



```
dvec3 crater = radius*normalize(dvec3(1.0, 1.0, -2.0));
double crater_dist = abs(length(crater-p));
double crater_edge_dist = abs(1000.0-length(crater-p));
double crater_contrib = 0.0;
if(crater_dist < 1000.0){
    crater_contrib = (1000.0-crater_dist)/1000.0;
    crater_contrib = 1000.0*crater_contrib*crater_contrib;
}
crater_contrib -= 80.0*(1.0-smoothstep(0,200,crater_edge_dist));
return length(p) - radius + 1.0*crater_contrib - 120.0*f64_noise_func(octaves, np);
```

Figure 14: Surface feature: crater



```
float t = abs(normalize(vec3(pf_pos))).y;
if(t > 0.5 && snoise(vec3(pf_pos*0.01f)) > (4.0-5*t)) {
    baseColor = vec3(0.6);
}
```

Figure 15: Surface feature: polar ice caps

4 Implementation - Asteroids

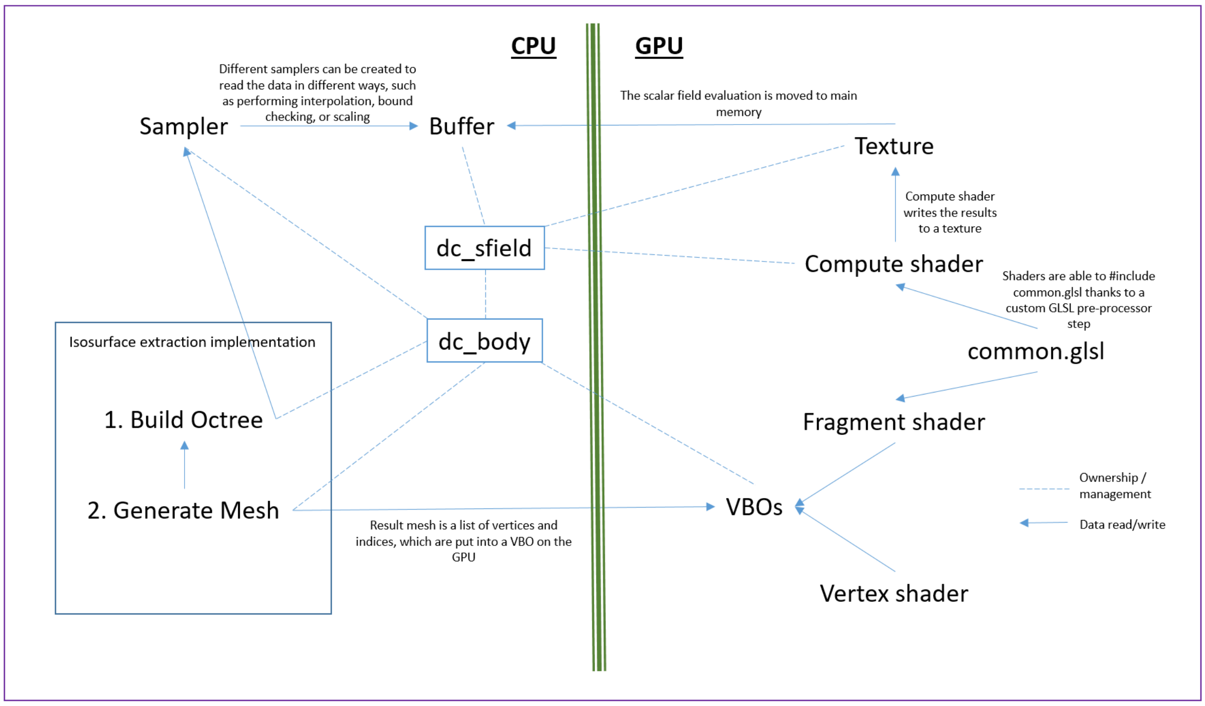

CPU
GPU
Different samplers can be created to read the data in different ways, such as performing interpolation, bound checking, or scaling
Sampler
Buffer
The scalar field evaluation is moved to main memory
Texture
Compute shader writes the results to a texture
dc_sfield
Compute shader
Shaders are able to #include common.glsl thanks to a custom GLSL pre-processor step
dc_body
common.glsl
Isosurface extraction implementation
1. Build Octree
2. Generate Mesh
Fragment shader
VBOs
Ownership / management
Data read/write
Result mesh is a list of vertices and indices, which are put into a VBO on the GPU
Vertex shader

Figure 16: The architecture of the system that generates asteroids

4.1 Evaluation of the scalar field

Upon consideration of the requirements of the system, it quickly becomes apparent that the implementation must make heavy use of the GPU, in order to be flexible, useful, and efficient. The scalar field that defines the shape of an asteroid needs to be evaluated for at least two purposes: creating the mesh using Dual Contouring, and shading the mesh according to the lighting of the scene. In order to perform the latter process on a per-pixel basis, without interpolation for the best quality shading, the fragment shader responsible for shading the mesh must be able to evaluate the scalar field at given points, so the scalar field will need to be defined in GPU code – GLSL, in this case. Thus, if defining the same scalar field multiple times is to be avoided for simplicity, the evaluation of the scalar field for the mesh creation must also occur on the GPU.

Dual Contouring is not fully data parallel, because edges and triangles are created between some (but not all) voxels, so it's not the case, unlike with Marching Cubes, that each piece of input data is transformed into an independent piece of output data. In addition, the reference implementation of Dual Contouring uses several mutually recursive functions, a design that would not be suitable for GPU implementation. For these reasons, it is significantly more feasible to implement Dual Contouring on the CPU. To this end, the CPU also needs to be able to access values of the scalar field.

The solution to satisfy these opposing requirements is to use GPGPU techniques to evaluate the scalar field on a structured grid of points in the region of interest on the GPU, save the results into a 3D texture, transfer the texture from the GPU into the main memory, run Dual Contouring on the CPU, and transfer the resulting mesh back onto the GPU. This is feasible because this is a pre-process step that only needs to occur once as an asteroid is created, rather than during real-time usage of the system, and due to the incredibly high memory bandwidth (transfer rate) of modern GPUs. (128x128x128 voxels, 32bit float in each = 8MB of data. PCIe 3.0 x16 transfer rate to GPU approaching 16GB/s.) This solution results in a very clean and appealing way of defining the scalar function just once in GLSL. There are a variety of GPGPU platforms available that could be used to compute the scalar field values, such as OpenCL, OpenGL compute shaders, or CUDA. OpenCL integrates well with OpenGL, but adds another dependency, OpenGL compute shaders are included in OpenGL by default (in the later versions) but lack more advanced functionality, and CUDA is very powerful but is only supported on Nvidia hardware. The implementation described in this project uses OpenGL compute shaders, because GPGPU is only needed for one simple task and it was easiest to use the features already included in OpenGL, and because the project commenced using an AMD GPU. Furthermore, it is extremely easy to define a function once in GLSL and use it in both types of shader – compute and fragment. There is room for improvement regarding this solution with respect to multitasking, as shall be discussed in section 7.2.

A compute shader is a shader which will be executed in parallel by a specified number of threads, each taking as input a unique multidimensional ID, which can be used to specify what work should be done by a given shader invocation. The desired number of invocations, including the range and dimensionality of the unique invocation identifiers, is specified CPU side by a simple method call. In this case, the scalar field function must be evaluated at points at fixed intervals in a cube of space.

In the first stage of the Dual Contouring algorithm, while building the leaves of the octree, the value of the scalar field will be sampled at every corner of every voxel, hence the grid structure of invocations of the compute shader. However, it will also need to sample at non-integer coordinates, i.e. at points inside individual voxels, such as for determining the intersection point of the isosurface along a given edge of a voxel. To this end, it is necessary to interpolate values from the compute shader's output texture when sampling CPU side. This implementation passes all requests for the value of the scalar field through a sampler, which is an object responsible for returning a useful result using the information from the compute shader. In practice, it performs bound checking and trilinear filtering to obtain values within voxels, and directly reads from the array for integer coordinate values. The sampler for asteroid creation also performs a translation of the requested point by a different offset for each asteroid, so that several asteroids can share the same compute shader result array and still look different. This system uses asteroids of 64x64x64 voxels, and generates 128x128x128 voxels worth of noise using the compute shader, so several asteroids can easily share the same noise array. The same Dual Contouring code will need to work for asteroids and planets, but the samplers have different functionality in each case, so there are multiple different samplers which all implement the same interface.

4.2 Dual Contouring

The next step of the process is to run Dual Contouring on the scalar field data to create the mesh. There are two main stages to the algorithm: creating the octree, and contouring to create the mesh. The two stages are very independent of each other, with information only passing from the first stage to the second stage in the octree. For ease of conceptualisation, programming, debugging and optimization, the stages are implemented separately with only a minimal interface.

4.3 Building the Leaves / Building the Octree

This stage takes as input just the region of interest, which is the total region of space in which the mesh should be created. It returns an octree of nodes, where each node represents a voxel of space, and its eight children each represent an octant of the same space after binary subdivision along each dimension. The tree will have leaf nodes which represent voxels of a predefined size and exist only where the relevant region of space contains the isosurface. The leaf node will contain information about the point within it where the mesh vertex will be placed.

A simple way to implement this is to recursively build the octree "top-down". Starting with the root node, recurse to create each of the children, and when a certain depth is reached, build the leaf nodes. This is inefficient because lots of nodes are created only to be deleted straight away when none of the leaf nodes that descend from them contain the isosurface, but the advantage is that as the process occurs, it automatically creates the tree structure. If each recursive call creates a child for each octant, then clearly this method is only suitable for creating octrees for cubic regions with a power of two side length of voxels. While this is good enough for the asteroids system, which will always have a fixed size region of interest, it is not sufficient for the planets system, which will become apparent in section 5.2, and which motivates a better method.

A more advanced way is to build the tree "bottom-up", by first creating all the leaf nodes that contain the isosurface, then building the non-leaf nodes up from there. A naïve implementation of this method creates all the leaves by simply iterating over each voxel in the region of interest and constructing a leaf if the isosurface is contained in that voxel. A more optimized version assumes that the isosurface is one continuous, connected surface in the region of interest, as it is in this application, and speeds up the leaf building process by starting from one voxel that contains the isosurface (which is found by naïve search) and proceeds to find the rest of the leaves by repeatedly exploring neighbours. The two methods have different performance characteristics, as discussed in chapter 6.

Once the leaves have been created, the rest of the tree structure must be created. This is achieved by putting each leaf into a hashmap, indexed by its integer coordinates within the space, as it is created. To create the layer of nodes above the leaves, a new hashmap for the new layer is created, and for each leaf, the coordinates of the parent node are computed, and used to create the parent node and insert it into the new hashmap, or attach the child to the parent node if it has already been created. This process can simply be repeated until there is a layer containing only one node, and that node is returned as the root of the new tree.

4.4 Contouring the Octree

The octree that has been produced contains the positions of all the vertices in the mesh, but they still need to be connected into polygons. It is required that a polygon is created for each leaf edge that exhibits a sign change, connecting the vertex positions for each of the four cells which contain the edge.

Ju et al present an efficient and neat algorithm for recursively exploring the tree such that a method, ContourProcessEdge, gets called for each edge that exhibits a sign change, correctly handling situations involving leaf nodes that are not all the same size (this does not occur with the asteroids system, but does occur with the planets system, and implementing a contouring method that supports different size leaves allows for mesh simplification to be added, a feature of dual contouring whereby the mesh can be reduced in size with minimal effect on the isosurface quality). It works by introducing three mutually recursive methods:

ContourEdgeProc ensures that ContourProcessEdge is called on every smaller edge that exists along the called edge,

ContourFaceProc ensures that ContourProcessEdge is called along every edge that is contained anywhere within the face,

ContourCellProc ensures that ContourProcessEdge is called for every edge that is contained anywhere within the cell.

Thus, if each of these methods meets these requirements, then calling ContourCellProc on the root cell results in the desired effect.

ContourEdgeProc checks if all four of the cells containing the edge, which are provided as an argument, are leaves. If they are, then the edge contains no smaller edges, so it calls ContourProcessEdge, otherwise, the edge can be divided into at least two sub-edges, so it recurses on each.

ContourFaceProc checks if either of the two cells containing the face, which are provided as an argument, are non-leaves. If so, then the face contains four subfaces, so it calls ContourFaceProc on those, and four extra edges between those faces, so it calls ContourEdgeProc on those. If both the cells are leaves, then the face does not contain sub-faces, so it does nothing.

ContourCellProc works similarly: if the cell is not a leaf, call ContourCellProc on the eight children, ContourFaceProc on the 12 faces between the children, and ContourEdgeProc on the six edges between the children. If it is a leaf, do nothing.

Thus ContourProcessEdge gets called on every edge exactly once, with the appropriate cells containing the edge as arguments, even if the octree contains differently sized leaf nodes. It checks to see if the edge exhibits a sign change, and if so, creates a polygon (two triangles) with the four relevant points, reordering them if necessary for consistency. The triangles get added to a list of triangles that is being passed around as an argument. The vertices are referred to by their index in a list of vertices, so vertices only get added to the datastructure once, even if they are involved in multiple triangles. The index of each vertex in the final datastructure is already known because, before the contouring begins, there is a procedure call which iterates over the tree recursively, adding each leaf's vertex position to a list of vertices, and saving the index of the vertex in that list into the leaf node itself, so that triangles can be created by arranging indices.

4.5 Transferring data to GPU

Once a list of vertices and a list of triangles have been produced, the data can be sent to the graphics card and there is no need to keep a copy of it in the main memory, as the OpenGL implementation will now manage the data and store it on the main memory if the GPU memory becomes full.

4.6 Mesh Shading

The system as described above can draw a wireframe outline of the mesh, or a plain colour version. The mesh needs to be shaded, taking into account the lighting, in order to achieve a realistic look.

Modern OpenGL requires that the developer program each stage in the "graphics pipeline" using shaders. The pipeline roughly works as follows. Data is supplied in the form of vertex buffers. For each vertex, an invocation of the vertex shader is used to compute a transformed position, and optionally perform arbitrary data-parallel computation on other per-vertex data, such as vertex colours, texture coordinates, ambient occlusion or lighting. The vertices are then connected into triangles according to the indices provided in the index buffer. The triangles are rasterized and a fragment shader is invoked once per "fragment" that may appear in the final image. A fragment usually corresponds to a pixel, and can be thought of as such by the developer. The vertex shader outputs are automatically interpolated over the triangle and passed to the fragment shader as input. Lighting calculations can be performed in the vertex shader on a per-vertex basis, or in the fragment shader on a per-pixel basis. The lighting quality is far superior if it is performed separately for each pixel in the fragment shader, rather than interpolating it from the vertex shader.

As described in section 3.3, to compute Phong lighting in the fragment shader, L_m, N, and V must be computed at each fragment. The vertex shader transforms the positions of vertices into screen-space through a series of matrix transformations, and this is the position information that the fragment shader receives by default. In order to access the world-space position in the fragment shader, the vertex shader additionally passes the world-space position of each vertex as an output. The fragment shader then receives the interpolated world-space position. Note that this is slightly different from the corresponding position of the actual isosurface, but is much cheaper than computing the interpolation that follows the isosurface, and shouldn't affect the resulting image much.

With the light and camera positions passed in as uniforms, it is then straightforward to compute L_m and V. The computation of the normal is the main advantage of defining the scalar field in GLSL. Given a point, an approximation to the normal can be calculated using the scalar field.

The mesh looks quite good with shading. A big problem is that there are light patches in regions of the surface that should be dark, because part of the mesh faces the light and is thus being shaded light, even though there is part of the asteroid between it and the light, which should be casting shadow. This is especially noticeable on the dark side of a rough sphere, where light patches should obviously not be present.

4.7 Shadows

The solution is to include shadows in the lighting calculations. It is required to compute whether the mesh occludes a given fragment's lighting. There are several different methods for computing shadows that are frequently used in graphics applications today.

The system implements a popular, straightforward method called shadow mapping [Williams 1978]. The idea is to render the scene from the light's point of view, and store the depth buffer into a texture. By transforming a point into light-space and querying this texture, we obtain the distance from the light to the nearest point on the mesh to the light in that direction. By comparing this distance to the distance from the given point to the light, we can compute whether the point is in shadow or not.

The first step is to create the shadowmap texture by rendering the scene from the point of view of the light. A matrix is created to transform points into light-space, using an orthographic projection because in this scene, the light rays coming from the source are modelled as parallel. A trivial shader is used which only performs the basic transformation using the matrix, and a new texture is bound to the depth buffer so that depth information is written into the texture.

When computing lighting in the fragment shader, a comparison can now be done between the distance between the light and a point, and the distance between the light and the nearest point in that direction. Given a point, projecting it into light space and taking the x,y components results in the coordinates which need to be used to query the shadow map texture. Conveniently, the z component is the depth of the given point. If this value is greater than the result stored in the texture, then the fragment is in shadow and the diffuse and specular lighting components can be ignored. Theoretically, the value should be at least equal to the result stored in the texture, being equal when the fragment is not in shadow. In practice, there are rounding and precision errors.

Quantisation and finite precision result in artefacts in the lit area of the mesh called "shadow acne". This can be fixed by introducing a bias, which increases the minimum distance to the light required to be considered shadow by a fixed small amount, which can be varied to achieve the desired visual quality. Unfortunately, this bias shifts the line between shadow and lit area, an effect known as "peter panning". It becomes clear that softer shadows are required, in order to more realistically reflect the real world. One solution to mitigate the artefacts and produce nicer soft shadows is percentage closer filtering, or PCF [Reeves et al. 1987]. This works by averaging several samples around the point of interest, softening borders between light and dark. With PCF, the system is able to reproduce realistic, soft shadows, without noticeable artefacts.

4.8 Background

Procedurally defining a background entails writing a function which computes the colour that should be displayed when looking in a particular direction. An easy way to represent the direction is simply a direction vector from the camera. Before drawing anything, a unit cube can be drawn around the camera, with the 6 faces of the cube covering every possible direction. If this cube is drawn without writing depth information to the depth buffer, then it will appear behind any other geometry, and can be shaded by a fragment shader which receives the world-space coordinates of the point on the cube as input, which is also the direction vector from the camera. In other words, the fragment shader procedurally defines a background. This approach would involve computing the background for every pixel on screen each frame, which would have a huge effect on performance with a complex, procedural background. A better approach (for a static background) is to use the fragment shader instead to read from a texture. OpenGL includes a specific type of texture that can be indexed by a direction vector, called a Cube Map. The cube map can be written to, once, as a preprocessing step, by the fragment shader that computes the background. In my system, the cube map is written to by rendering a rectangle six times, once for each face of the cubemap, with the cubemap attached to a framebuffer object as the colour attachment.

5 Implementation - Planets

The system that renders asteroids does not scale to larger bodies, such as planets, because a mesh large enough to have sufficient detail to look realistic when viewed from close to the surface is too large to compute, store and render. When viewed from space, however, it is acceptable for the mesh to be a reasonable size. This means that a level-of-detail system is required, which introduces more detail when necessary on a certain part of the mesh as the camera approaches the planet. There are many ways this could be accomplished. The following sections describe the implementation that is in use in the planetary system, which works by recursively introducing new, separate meshes, each produced similarly to the asteroids, as required.

5.1 LOD chunks

The planet is represented by several mesh chunks, rather than a single mesh. The mesh chunks are organized into an octree structure, so that parts of the planet near the camera can be made with several small mesh chunks, while parts of the planet that are far from the camera can be made with fewer, larger mesh chunks. This LOD system is easy to implement in an adaptive way, by replacing a mesh chunk with eight smaller mesh chunks when the camera is within a certain threshold distance, and replacing them with the single larger chunk when the camera retreats. Each mesh chunk creates its own mesh using Dual Contouring on its own scalar field data. By offsetting the chunk-space position appropriately when calling the scalar field function, the meshes will represent parts of the same shape and combine together correctly. For simplicity, in this implementation, each mesh chunk's mesh is created from the same sized cube of voxels, and each layer of the octree of mesh chunks is scaled to be smaller by a factor of two on each dimension.

5.2 Seams

When multiple meshes are created with Dual Contouring, with their scalar field evaluations scaled appropriately such that they represent neighbouring parts of the same shape, are placed next to each other as described above, the resulting meshes do not meet and connect together. (Figure 17) This is because no triangles are generated for the bordering voxels, as each run of Dual Contouring is unaware that the mesh extends further than its set of voxels.

There are two obvious ways to solve this issue. The first and more simplistic approach is to overlap the meshes, both in the rendering space and in the scalar field space, so that the bordering voxels are accounted for in both meshes and the meshes meet. This works well for meshes of the same level of detail, but does not work well when one of the meshes is more detailed than the other. The extra vertices create cracks between the meshes, or holes in the surface. This motivates a more complex and less known solution: an extra seam mesh that sits between the two meshes and 'stitches' them together. (Figure 18)

The seam solution also works for neighbouring meshes with different levels of detail. (Figure 19)

The seam mesh can be created with Dual Contouring in the same way as the main meshes,

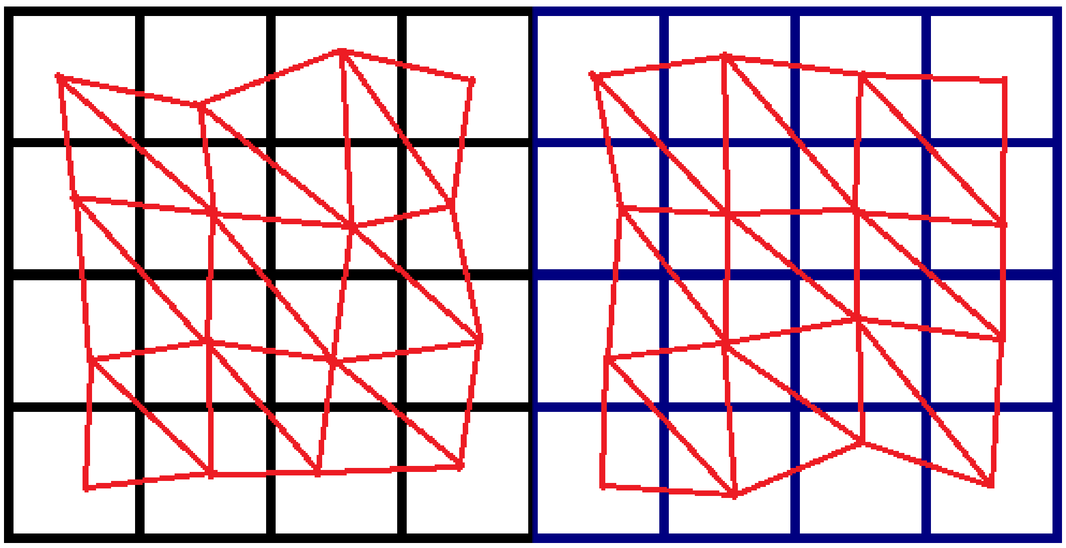


Figure 17: Neighbouring meshes leave a gap

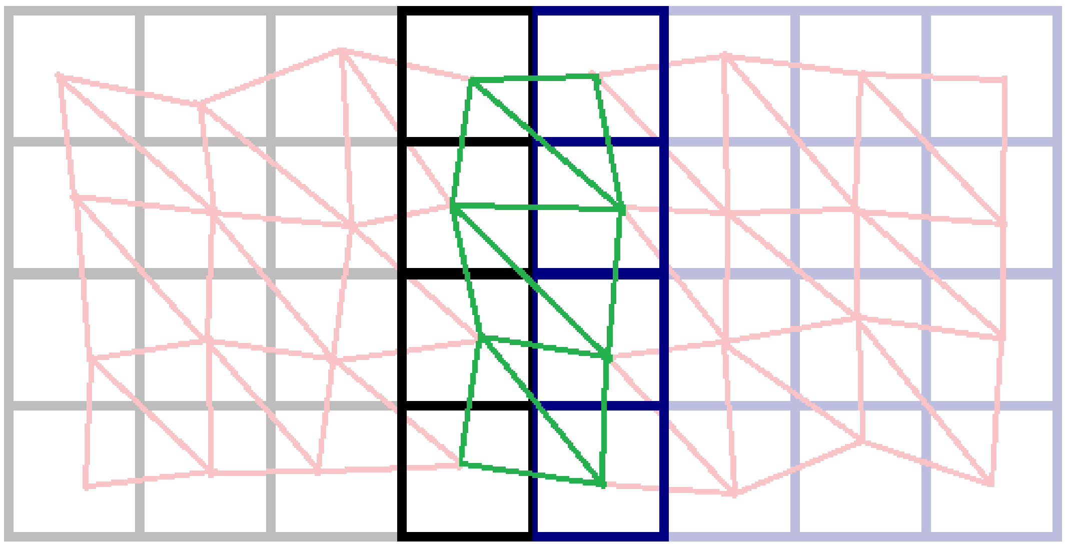


Figure 18: Neighbouring meshes stitched with seam

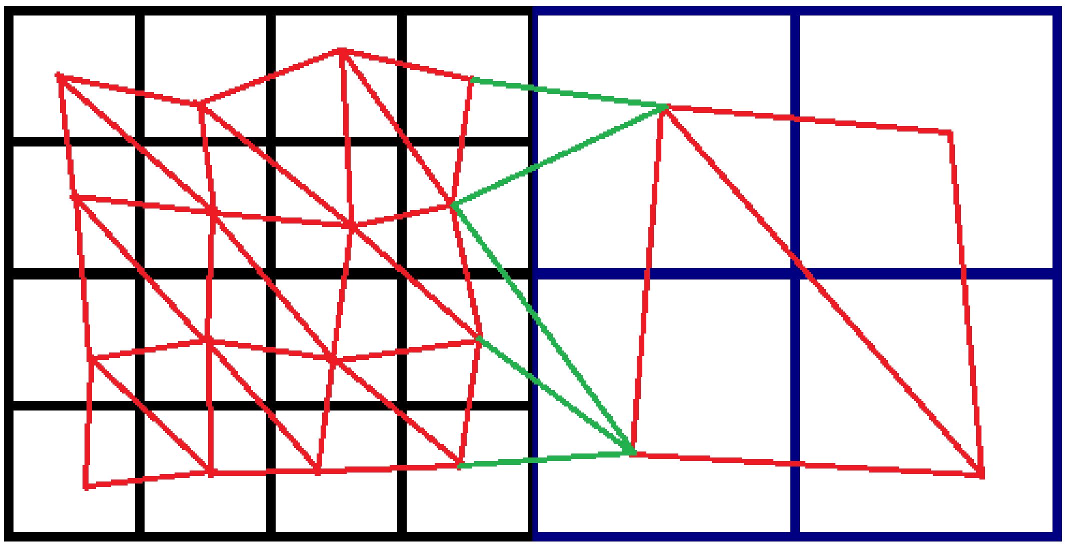


Figure 19: Seam for two meshes of different level of detail

using the relevant voxels from the bordering faces of the neighbouring chunks. Thus, it is required to collate the leaves for the seam based on the level of detail of the neighbours. Given that traversing the tree is required to get the level of the neighbouring chunks and the scalar field information necessary to build the leaves, the implementation contains recursive methods to traverse the tree and add the relevant leaves into a collection upon reaching the appropriate chunk.

```
const int seamLeavesSubchildren[] = { 0,1,4,5, 0,1,2,3, 0,2,4,6, 2,3,6,7, 4,5,6,7, 1,3,5,7};

void MeshChunk::collectSeamLeaves(LeafMapCollection* collection, int direction)
{
    if(this->hasChildren)
    {
        for (int i = 0; i < 4; i++)
        {
            this->children[seamLeavesSubchildren[direction * 4 + i]]
                ->collectSeamLeaves(collection, direction);
        }
    }
    else
    {
        int offset = 6 * direction;
        glm::ivec3 min = glm::ivec3(collectLeavesRegion[offset + 0], collectLeavesRegion[offset + 1],
            collectLeavesRegion[offset + 2]);
        glm::ivec3 max = glm::ivec3(collectLeavesRegion[offset + 3], collectLeavesRegion[offset + 4],
            collectLeavesRegion[offset + 5]);

        LeafMap* leaves = BuildLeafLayer_search(this->iss, min, max);
        collection->emplace(this, leaves);
    }
}
```

Unfortunately, once the leaves have been collected, the values that they contain are in their origin chunk's space, which is scaled and translated differently than the destination chunk's space due to the level of detail system, and the leaf must be transformed to the destination chunk's space. The code to perform this transformation, while concise, was not trivial to create, due to the precise and confusing nature of the transformation. There are three cases, depending on whether the transformation takes a leaf from a chunk of smaller, equal or higher level of detail.

```
/*
* Takes a leaf, the source and destination chunks, and a logsize_increase. Computes the new min corner taking
    these into account.
* The only side effect is that leaf->minCorner is modified.
*/
void TransformLeaf_MinCorner(DC_Octree* leaf, MeshChunk* source, MeshChunk* destination, int logsize_increase)
{
    leaf->minCorner *= 1 << logsize_increase;
    int level_difference = source->getLevel() - destination->getLevel();
    /* source bigger than destination */
    if (level_difference > 0)
    {
        glm::ivec3 temp = leaf->minCorner;
        /* dimensions of target block */
        int s = 64 * (1 << logsize_increase);
        /* decrease factor */
        int ldiff = 1 << level_difference;
        /* addr of translated block */
        glm::ivec3 trans_addr = destination->getAddr() / ldiff;
        /* pos relative to trans block */
        glm::ivec3 trans_minCorner = leaf->minCorner + s * (source->getAddr() - trans_addr);
        /* addr of target within translated block */
        glm::ivec3 off_addr = destination->getAddr() - trans_addr * ldiff;
        /* pos of offset block */
        glm::ivec3 off_pos = s * (off_addr - (ldiff / 2)) + (s / 2);

        leaf->minCorner = trans_minCorner * ldiff - off_pos;
    }
    /* source same size as destination */
    else if (level_difference == 0)
    {
        int s = 64 * (1 << logsize_increase);
        glm::ivec3 temp = leaf->minCorner;
        leaf->minCorner = s * (source->getAddr() - destination->getAddr()) + leaf->minCorner;
    }
    /* source smaller than destination */
    else
    {
        /* dimensions of target block */
        int s = 64 * (1 << logsize_increase);
        /* increase factor */
        int ldiff = 1 << -level_difference;
        /* addr of bigger block before translation and offset within that block */
        glm::ivec3 adj_addr = source->getAddr() / ldiff;
        glm::ivec3 off_addr = source->getAddr() - adj_addr * ldiff;
        /* pos of offset block */
        glm::ivec3 off_pos = (s/ldiff) * (off_addr - (ldiff / 2)) + (s/(2*ldiff));
        //equivalent: glm::ivec3 off_pos = ((s * (off_addr - (ldiff / 2))) + s / 2) / ldiff;
        /* adjusted position */
        glm::ivec3 adj_pos = leaf->minCorner / ldiff + off_pos;
        glm::ivec3 temp = leaf->minCorner;
        leaf->minCorner = adj_pos + s * (adj_addr - destination->getAddr());
    }
}
```

The last change that needs to be made in order for seams to work is that the seam needs to be updated whenever any of the chunks it stitches together is subdivided into a more detailed version. This can be implemented with a straightforward listener and notifier model, and the result is a large, adaptive, correctly stitched octree of meshes, as pictured below.

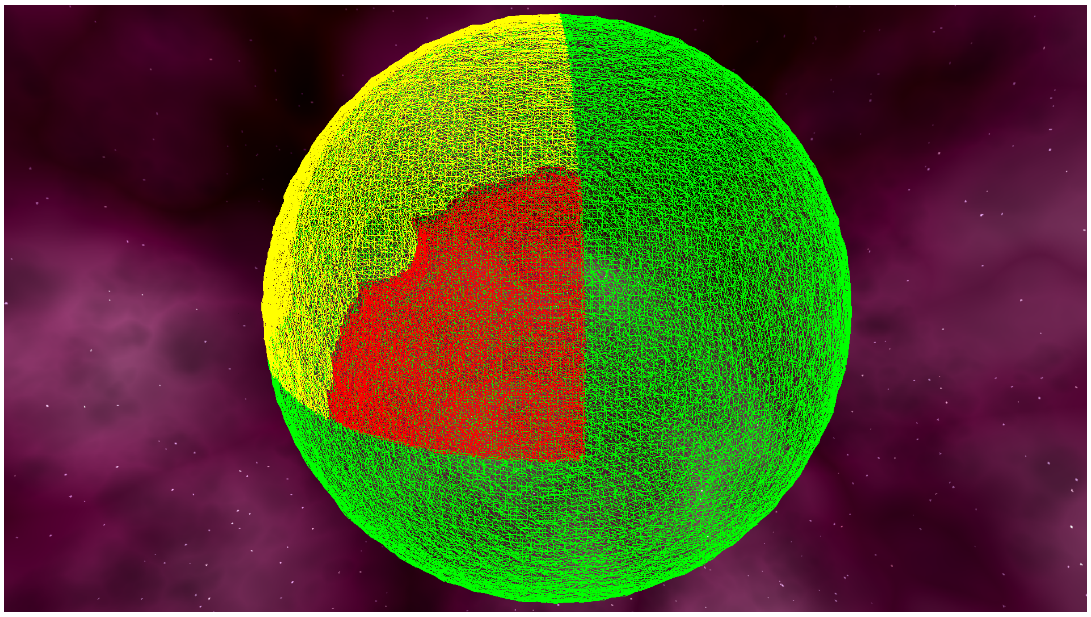


Figure 20: Huge meshes can be made of patches

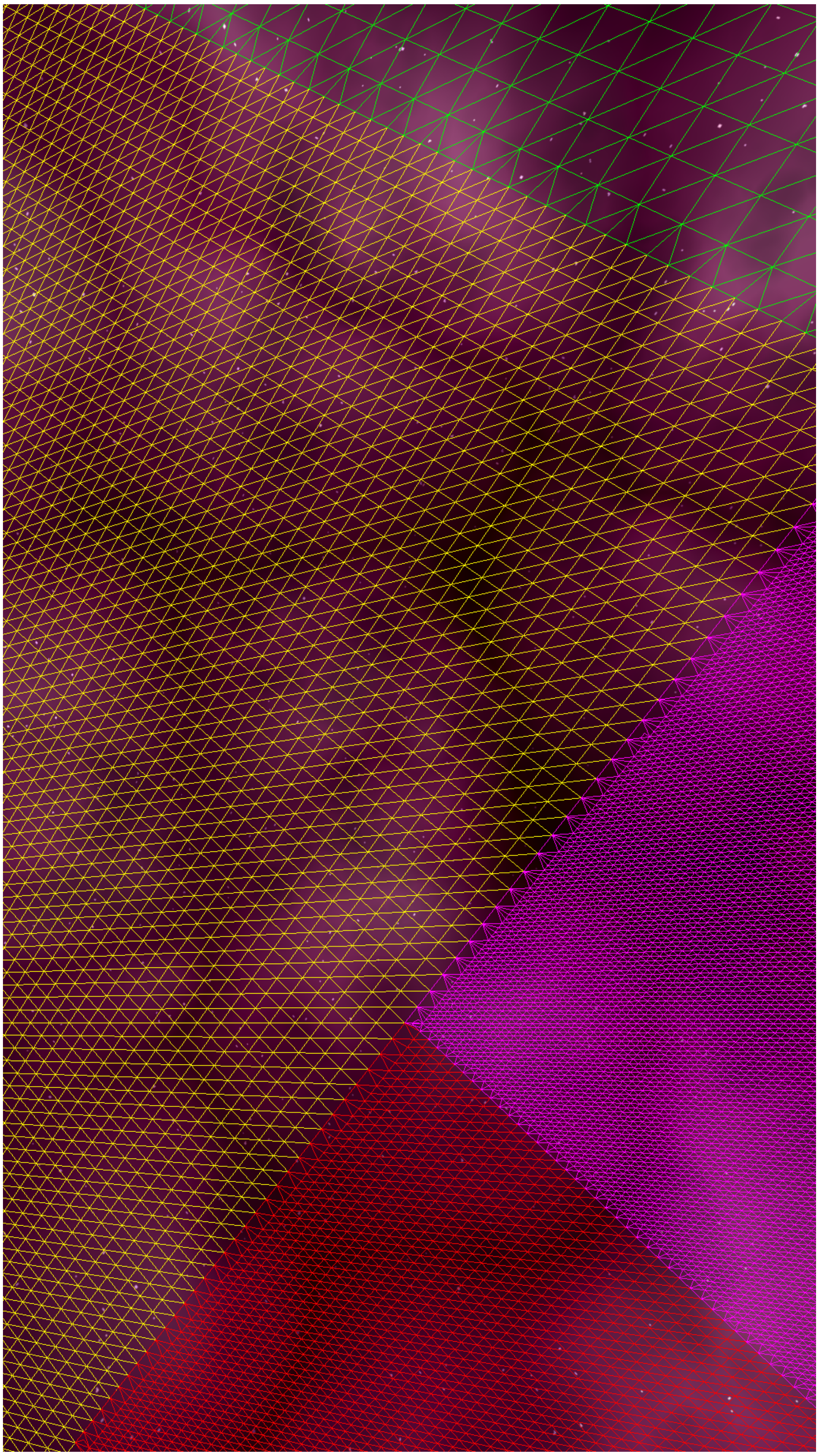


Figure 21: Seams working between patches of several different LODs

6 Performance Results

The following data (Figure 22) is a performance comparison between three different versions of the Dual Contouring implementation, as applied to the generation of asteroids.

For this test, a fixed sample of the same 98 asteroids was generated using each of the different implementations. `Time_Octree` is the time taken to build the octree, while `Time_Mesh` is the time taken to contour the octree into a mesh, corresponding to the two main stages in the Dual Contouring implementation as described in section 4.2.

The first version is a reference implementation from [Nicholas Gildea 2015], and is based on the reference implementation code provided by [T. Ju et al. 2002]. The second version is an initial, naïve implementation that I wrote for this project.

Version	`Time_Octree (avg)`	`Time_Mesh (avg)`
Reference	127ms	0.316ms
Naive	101ms	0.307ms
Search	81.3ms	0.278ms

Figure 22: Performance table

It is immediately clear that a lot more time is spent building the octree than contouring the mesh. Further inspection reveals that the majority of the time is spent building leaves. For this reason, the next version implements an optimisation for building the leaves, attempting to reduce the number of leaves by exploring the space of potential leaves using a tree search on the neighbours of leaves that have already been built, as explained in the implementation chapter.

The full benefit of the search version is not conveyed by simply comparing averages. In the worst case scenario, when there are no leaves to be built, the search version performs worse than the naïve version, which makes the average time much worse. However, for any case other than this, the search version is significantly faster. Consider the following graphs, which plot `Time_Octree` against the size of the resulting mesh for each asteroid in the sample.

The performance characteristic of the reference implementation is a simple, linear relationship, with the smallest meshes taking around 110ms and the largest taking around 150ms (Figure 23).

The same linear relationship is seen in the naïve implementation, but slightly faster, starting around 80ms and the largest taking around 130ms (Figure 24).

However, while the search implementation performs poorly on empty meshes, taking around 100ms, non-trivial meshes perform much faster, in a linear relationship starting around 50ms up to the largest taking around 100ms (Figure 25). Unless further assumptions are to be made about the shape of the isosurface, there is no way to avoid checking every point on the grid when running Dual Contouring to create a mesh that will ultimately be empty.

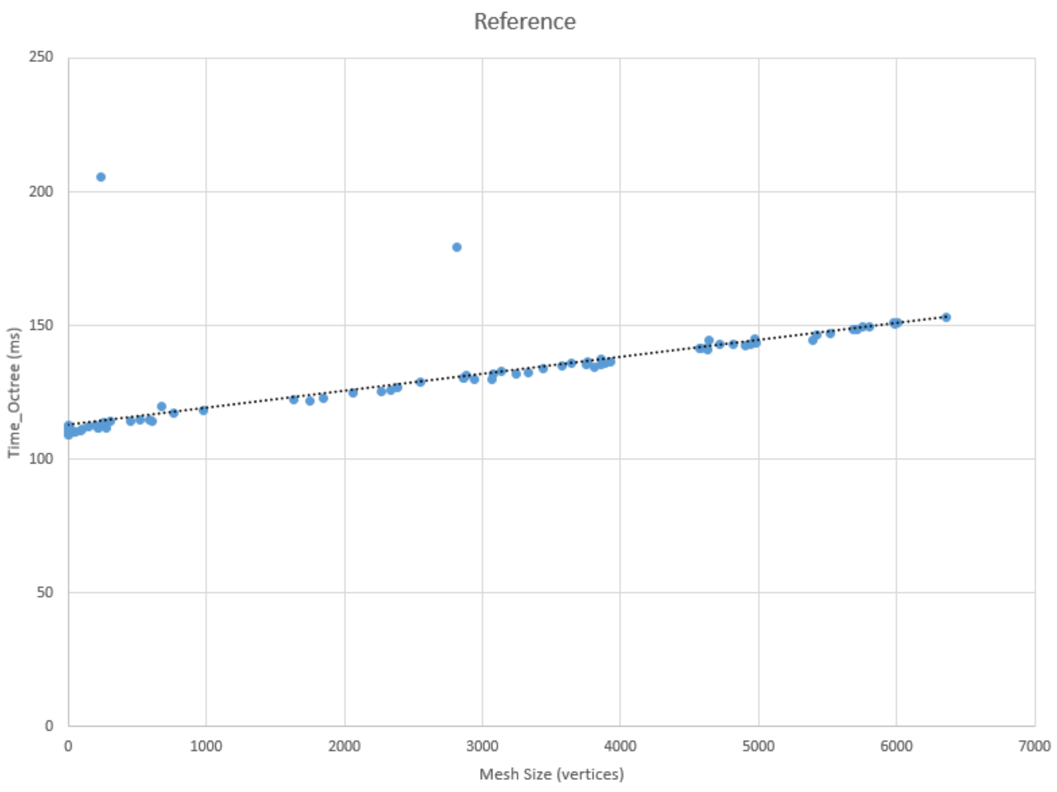


Figure 23: Reference performance

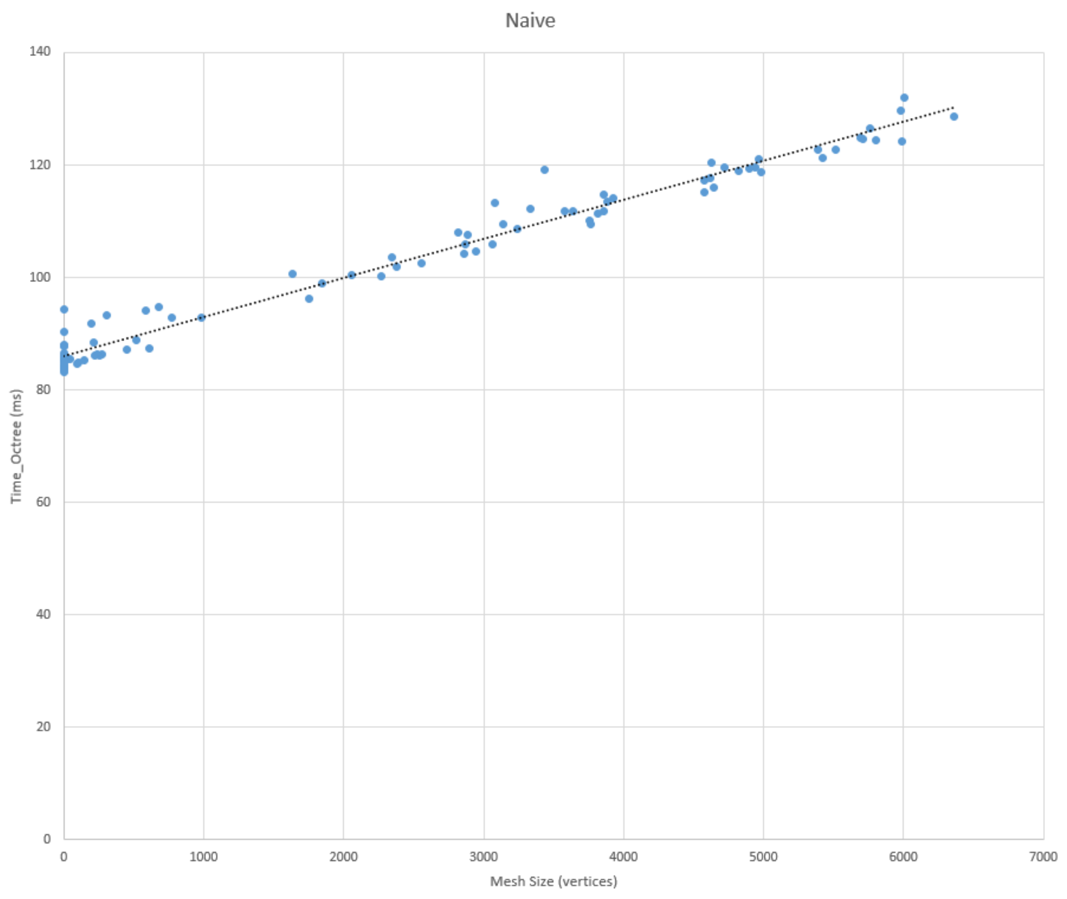


Figure 24: Naive performance

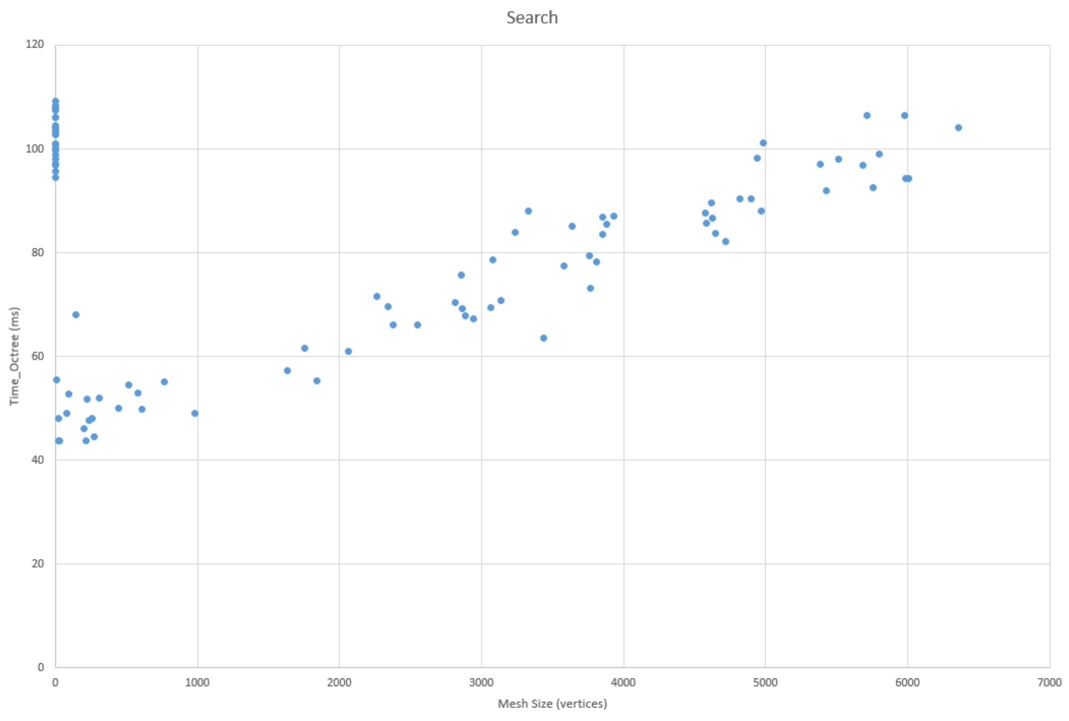


Figure 25: Search performance

7 Discussion

The system is a successful implementation of the design described in the design chapter.

The design decision to implement Dual Contouring resulted in great learning opportunity and the chance to explore the feasibility of using Dual Contouring in a realtime environment, such as a video game. As expected, though the results are visually impressive, there is a significant performance hit when increasing the level of detail of a chunk. Marching Cubes might be more suitable for a video game, because it would be faster and would lend itself much more readily to the data parallel nature of the GPU, avoiding the latency in moving data back and forth from main memory, and the trade-off of less accurate shape reproduction is not a big issue in such an application.

7.1 Areas of interest

Level of detail scaling The system works well for every viewpoint from space to ground, increasing detail as required to maintain visual fidelity. Despite the difficulty in implementation, the seams between different chunks work perfectly. LOD transitions are a bit slow, but it has not been possible within the scope of this project to optimize these yet. It may be possible to mitigate the impact using multitasking as suggested below.

Aesthetics Visually, the renderer has been a success, rendering beautiful images with great detail, creating a convincing rocky look without stored textures or meshes, and asteroids are shaded with soft shadows.

Simplicity The asteroids and planets that the renderer produces are extremely easy to configure, and are described very succinctly in GLSL. The use of Dual Contouring affords great flexibility, allowing any shape to be described and reproduced faithfully, including sharp corners, flat surfaces, and interesting topologies. It is easy to create a new type of asteroid or planet and define a new style, without even having to recompile the code.

7.2 Improvements - fixing problems

- A major weakness of the implementation is that the GPU stops processing the rendering pipeline while the compute shader that evaluates the scalar field is running. Thus, increasing the level of detail causes the renderer to stutter. This could be solved by using some form of GPU multitasking, allowing the compute shader to run in parallel with the rendering pipeline. One way to achieve this would be to use a more modern graphics API, such as Vulkan or DX12, which allow multiple "streams" of computation to run in parallel.
- The outer layers of the planet could be approximated by a simple approach based on quadtrees and heightmaps, as the interesting topology wasn't taken advantage of until the deepest layers. This could be used to speed up the generation of the outer layers, though seams between heightmap layers and isosurface layers would be difficult.
- The indexing system for leaves of octrees in the planet implementation is very complicated and contrived, and it was extremely difficult to write code to produce the seams between

chunks correctly. This was the result of designing the system with asteroids in mind and then extending it to planets afterward. A better solution would be to use a global indexing system for all leaves, regardless of the chunk that contains them. If `uint64_t` integers (unsigned 64 bit integers) were used for each coordinate of a global coordinate system, then there would be enough space for 58 layers of chunks of size 64x64x64, which is more than enough for any reasonable implementation, and would avoid the error-prone code to translate coordinates from one chunk's space to another.

7.3 Improvements - suggested future work

- The visual quality of the renderer could be improved by introducing a model of the atmosphere around a planet, computing lighting effects like atmospheric scattering, such as the one described in "A practical model for daylight" [Preetham et al. 1999].
- Instead of light coming from a predefined, fixed angle, a star could be drawn and emit light for the scene. Post processing effects like HDR and bloom would result in more convincing solar visuals. Volumetric rendering could be used to simulate the effect of solar rays on dust clouds in interplanetary space, creating a more realistic feel to the scene. Similarly, a more advanced shadowing model could be used, such as PCSS [Fernando 2005].
- Scaling the system further to render bigger planets may result in floating point precision issues, because single precision floating point values were used in the vertex buffers and in the evaluation of the scalar fields. The effect of this is mitigated in the existing system because the single precision vertices are in chunk space, and the chunk positions are double precision. Unfortunately, this doesn't completely avoid precision issues, because an entire planet is just one chunk while the camera is far away. Simply switching to double precision is not a good solution, because typical consumer GPU double precision performance is far slower than single precision.
- The entire Dual Contouring pipeline could be moved to GPU only [Chen et al. 2015, Buatois et al. 2006, Goradia 2008].

7.4 Conclusion

In conclusion, approaches to planet rendering based on isosurface extraction techniques are certainly feasible, especially as the computational power of consumer GPUs grows. With careful optimisation of the process of increasing LOD, smooth, realtime performance could be attained, allowing games and other applications to make use of this incredibly powerful and general approach to defining shapes.

References

[1] L. Buatois, G. Caumon, and B. Lévy. Gpu accelerated isosurface extraction on tetrahedral grids. *Advances in Visual Computing*:383–392, 2006.

[2] A. Chalmers and A. Ferko. Levels of realism: from virtual reality to real virtuality. *Proceedings of the 24th Spring Conference on Computer Graphics*:19–25, 2008.

[3] J. Chen, X. Jin, and Z. Deng. Gpu-based polygonization and optimization for implicit surfaces. *The Visual Computer*, 31(2):119–130, 2015.

[4] E. V. Chernyaev. Marching cubes 33: construction of topologically correct isosurfaces. *Institute for High Energy Physics, Moscow, Russia, Report CN/95-17*, 42, 1995.

[5] Crytivo Games. Universim. 2017. URL: `https://theuniversim.com/`.

[6] R. Fernando. Percentage-closer soft shadows. *ACM SIGGRAPH 2005 Sketches*:35, 2005.

[7] S. F. Gibson. Constrained elastic surface nets: generating smooth surfaces from binary segmented data. *International Conference on Medical Image Computing and Computer-Assisted Intervention*:888–898, 1998.

[8] R. Goradia. Gpu-based adaptive octree construction algorithms. 2008. URL: `https://www.cse.iitb.ac.in/~rhushabh/publications/octree`.

[9] S. Gustavson. Simplex noise demystified. *Linköping University*, 2005.

[10] Hello Games. No man's sky. 2016. URL: `https://www.nomanssky.com/`.

[11] T. Ju, F. Losasso, S. Schaefer, and J. Warren. Dual contouring of hermite data. *ACM Transactions on Graphics (TOG)*, 21(3):339–346, 2002.

[12] W. E. Lorensen and H. E. Cline. Marching cubes: a high resolution 3d surface construction algorithm. *ACM siggraph computer graphics*, 21(4):163–169, 1987.

[13] Nicholas Gildea. Dual contouring sample. 2015. URL: `https://github.com/nickgildea/`.

[14] J. Parnell. Limit theory. 2012. URL: `http://ltheory.com/`.

[15] K. Perlin. An image synthesizer. *ACM Siggraph Computer Graphics*, 19(3):287–296, 1985.

[16] K. Perlin. Improving noise. *ACM Transactions on Graphics (TOG)*, 21(3):681–682, 2002.

[17] B. T. Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18(6):311–317, 1975.

[18] A. J. Preetham, P. Shirley, and B. Smits. A practical analytic model for daylight. *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*:91–100, 1999.

[19] S. Raman and R. Wenger. Quality isosurface mesh generation using an extended marching cubes lookup table. *Computer Graphics Forum*, 27(3):791–798, 2008.

[20] W. T. Reeves, D. H. Salesin, and R. L. Cook. Rendering antialiased shadows with depth maps. *ACM Siggraph Computer Graphics*, 21(4):283–291, 1987.

[21] S. Schaefer, T. Ju, and J. Warren. Manifold dual contouring. *IEEE Transactions on Visualization and Computer Graphics*, 13(3):610–619, 2007.

[22] S. Schaefer and J. Warren. Dual contouring: the secret sauce. *Department of Computer Science Technical Report*, 2(408), 2002.

[23] S. Schaefer and J. Warren. Dual marching cubes: primal contouring of dual grids. In *Computer Graphics and Applications, 2004. PG 2004. Proceedings. 12th Pacific Conference on*, pages 70–76. IEEE, 2004.

[24] E. Smistad, A. C. Elster, and F. Lindseth. Fast surface extraction and visualization of medical images using opencl and gpus. *The Joint Workshop on High Performance and Distributed Computing for Medical Imaging*, 2011, 2011.

[25] Squad. Kerbal space program. 2015. URL: `https://kerbalspaceprogram.com/en/`.

[26] L. Williams. Casting curved shadows on curved surfaces. *Computer Graphics Lab*, 1978.