Procedural Modeling of Rocks

Anders Söderlund

Institutionen för informationsteknologi
Department of Information Technology
Abstract

**Procedural Modeling of Rocks**

*Anders Söderlund*

Gaming and other virtual environments are a big part of today’s society, but manual modeling of terrains used in such environments can be a lengthy and tedious process. This thesis serves to explore a few methods of procedurally generating models of rocks or boulders that could be used in such contexts. This includes geometry and shading. A couple of different methods are explored.

Sphere inflation, inspired by a classic sphere modeling method, involves “inflating” a base mesh (usually a platonic solid) to grow towards the boundaries of a sphere using an iterative subdivision approach, halting at a predetermined level of iteration.

The second approach, recursive subdivision of segmented edges, involves dividing a base mesh into edge segments based on a predefined segment size, subdividing a polygon with pre-segmented edges with a recursive subdivision method based on the sphere inflation subdivision scheme. The segmented edges method is followed by a corner cutting step to “carve” the base mesh into a shape approaching a rock.

The sphere inflation method was not successfully finished within due time, but the segmented edge method shows promise in generating fairly believable rock models.

The shading includes GLSL based fragment and vertex shaders employing a Perlin noise based procedural granite 3D texture.
Acknowledgements

I would like to thank my supervisor Stefan Seipel for all help and support and for all the good ideas that I have had a chance to explore.

Thanks also to my reviewer Anders Hast for advice and support during the final phases of this work and to Olle Gällmo for answering all my questions.
## Contents

1 Introduction ........................................ 5  
1.1 Why procedural content in applications? .............. 5  
1.2 Related work ....................................... 5  
1.3 Motivation ......................................... 6  
1.4 Goals ............................................... 7  
1.5 Framework ........................................ 7  

2 Theoretical Background ................................. 8  
2.1 Tessellation ........................................ 8  
2.1.1 Recursive triangle subdivision .................... 8  
2.1.2 Base shapes .................................... 8  
2.1.3 Ear clipping .................................... 8  
2.1.4 Delaunay triangulation ............................ 9  
2.2 Feature Edges ....................................... 10  
2.3 Fractal noise ....................................... 10  
2.3.1 Fractal functions ................................ 10  
2.3.2 Lattice and Gradient Noise ...................... 11  
2.4 Glossary .......................................... 11  

3 Method ................................................ 12  
3.1 Explored ideas ...................................... 12  
3.2 Sphere inflation ..................................... 12  
3.2.1 Extending the algorithm ......................... 13  
3.3 Recursive subdivision of segmented edges ............ 15  
3.3.1 Tessellation ..................................... 15  
3.3.2 Ear-clipping ..................................... 17  
3.3.3 Ear-clipping (modified) .......................... 17  
3.3.4 Recursive subdivision (revisited) ............... 18  
3.3.5 Feature edges ................................... 19  
3.3.6 Corner chipping ................................ 19  
3.4 Shading ............................................ 20  
3.4.1 Normal vectors .................................. 20  
3.4.2 Phong and flat shading ......................... 21  
3.4.3 Fractal noise-based 3D textures ................. 21  

4 Results ............................................... 21  
4.1 Sphere inflation ..................................... 21  
4.2 Recursive subdivision of segmented edges .......... 25  
4.2.1 Ear-clipping .................................... 25  
4.2.2 Ear-clipping (modified) ......................... 25  
4.2.3 Recursive subdivision ............................ 26  
4.3 Shading ............................................ 27  
4.3.1 Fractal noise-based 3D textures ................. 27
4.4 Interesting Icosahedrons, Octahedrons and lumpiness 28
  4.4.1 Icosahedrons 28
  4.4.2 Octahedrons 29
  4.4.3 Chunks and lumps 30
4.5 Speed 31
  4.5.1 Polygon count 31

5 Discussion 32
  5.1 Sphere inflation 32
    5.1.1 Shapes of rocks 32
    5.1.2 Surface detail 32
  5.2 Recursive subdivision of segmented edges 33
  5.3 Goals achieved? 33
  5.4 Comparison with other work 34

6 Conclusions 35

Bibliography 35
1 Introduction

1.1 Why procedural content in applications?

Today, computer games and virtual environments are a part or many people’s lives and we are getting accustomed to fairly realistic, high quality imagery, immense game worlds and immersive environments. This applies not only to entertainment and games but to simulations as well, for the purposes of education or training, for example. Creating natural-looking scenes however, requires a great amount of time and resources, as manual modeling is a time consuming process, and the demand for more 3D content is not likely to decrease. Indeed, content production has become a bottleneck in game budgets and production time, as the increasing demand for content does not scale with an increased number of artists on any given project [1, 2, 3, 4]. Considering how game developer teams and budgets have increased over the years, much due to the increasing complexity of art [5], it is not difficult to imagine that methods of generating 3D models without much time required from modelers would improve the efficiency of these projects a considerable amount, freeing up resources for use elsewhere.

Further advantages of procedural content as stated in the introduction of Texturing and Modeling: a Procedural Approach [6] include abstraction and storage savings (by never explicitly storing every complex detail of a scene) and parametric control and flexibility (to capture the essence of an object and being able to change large amounts of detail with very little work).

1.2 Related work

The field of procedural modeling has been around for decades. In the 80’s and 90’s, much research was done on the modeling of terrain, plants and textures, applying techniques such as height maps based on fractal noise [7, 8] for the creation of terrain, and the use of L-systems [9, 10, 11, 12] for an organic modeling approach to plants.

Further research on terrain modeling includes techniques inspired by natural phenomena such as stream erosion [13], which are typically used to improve an already existing terrain. In more recent years, a number of techniques on terrain modeling have been presented, such as terrain deformation in conjunction with combining terrain features [14], and solutions to some problems with earlier techniques such as height maps lacking overhangs [15] have been suggested. Hnaidi et al. [16] present a method for generating complex terrain models from sets of parameterized curves defining landform features.

At the beginning of the millennium, research began also on urban environments; creating buildings, road networks and entire cities [17, 1].

Thus, a lot of research has been done on the subjects of terrain and more modern environments, and these methods can be quite effective for large scale terrain, but research on certain details such as rocks or boulders seems to be less explored.

There have been a few papers on this subject, however. Peytavie et al. [18] present a method for creating aperiodic piles of rocks using voronoi cells and erosion, while
Dorsey et al. [19] discuss stone as a building material, presenting an algorithm to create weathered stone, using a technique that combines surface-based and volumetric methods by using a volumetric method locally around the surface of an object.

Beardall et al. [20] present a voxel-based method and interactive tool for the generation of sandstone goblins, a type of pillar-like rock formation found in nature, for instance at Goblin Valley State Park, Utah.

Approaching the subject of this thesis, Dart et al. [21] focus more specifically on the creation of rocks or boulders. They present a volumetric method to generate rocks using L-systems with voxels to create the initial mass, followed by an implosion step, pulling the voxels towards the center, eliminating cavities created in the first step. They begin with a cubic matrix as the L-system axiom, and four randomly generated grammar rewriting rules that are then expanded in a number of steps. These rules rewrite a cube into eight new cuboids replacing the space occupied by the previous cube. Each of these eight new cuboids is treated as either solid or as a gap. Another rule expansion creates eight new cuboids for each of the previous cuboids and so on. Expanding these rules for a certain number of steps creates a cube with random voxels removed. The number of steps dictates the voxel resolution of the model.

When this part of the process is done the implosion step begins, where the cuboids are pulled in towards the middle until the model is stable and all gaps have implicitly been moved outside the surface of the rock.

To create a mesh from this voxel-based model, they put the rock inside a sphere mesh and use ray casting to find the surface of the rock shape and move the corresponding vertex from the surface of the sphere to the surface of the rock shape. As they put it: An analogy to this would be vacuum sealing a rock by putting it into a plastic bag and removing the air.

They use this process of expansion and implosion repeatedly in evolutionary search comparing the results to a fitness function based on user specified parameters, stopping the search when no actual progress has been made for a few generations.

They also have an optional erosion step with an algorithm resembling ”a one-step cellular automaton”. It involves deleting cubes with a probability depending on how many neighbors they have and a user input erosion scale. This is applied after evolution.

### 1.3 Motivation

As Dorsey et al. [19] is more focused on piles of rocks, there has been a lack of methods focusing more on single rocks and boulders. Dart et al. [21] has such a method, and from the images in their paper the rocks generally look good and they claim to be fast and diverse. However, the topology with the cuboid building blocks seems to shine through somewhat in cases such as the flat rock in fig 6, Dart et al. [21]. This could perhaps be solved by using a higher resolution of voxels/cuboids, possibly by using their chunkiness variable, but this would probably increase computation time a great deal. Additionally, it would be useful to have a method that is fairly light and fast to accommodate a more interactive modeling experience. Therefore, it would be interesting to find a method with the ability to create rocks with a shape more similar to overall rounded boulders, but with decent surface detail, while using a time-efficient surface-based method.
The surface based methods used in this thesis need no iso-surface extraction or other specialized rendering techniques as volumetric methods do to render an object on common hardware. (Such as the common marching cubes method or the ray casting method used by Dart et al. [21].)

1.4 Goals

Rocks and boulders play an important part in all natural scenes as they add additional complexity and realism to its scene. However, it would be useful to have a method of generating them procedurally instead of modeling them by hand. The main goal of this project is to find a fast method of generating rocks or boulders with fair realism and some diversity. The focus is mainly on geometry, but shading has been included to lesser extent.

Rock characteristics Say that we want to create a few different types of rocks; all rocks in a given category should have some similar characteristic. At the very least they should have some aspect that reminds us of a rock or a boulder. Then perhaps we can extend this division into subsets defined by less general parameters to further specialize into different types of rocks. Shading is a big part of this feel, as different rocks have different texture and color to them. Shape and size would also matter, for instance in the sharpness or roundness of its edges. This could be dictated by different parameters controlling the process.

Surface vs volume There are a couple of different approaches to 3D modeling. Techniques can be categorized as surface oriented or volume oriented. Volume oriented modeling is perhaps more intuitive and creates a more complete model, using 3D arrays of voxels, but it needs rendering techniques such as volumetric ray tracing or a conversion to a surface model using an isosurface extraction method to be rendered. Surface oriented modeling can be more efficient, dealing with only that which is to be seen, however traversing a surface can be much less intuitive. This project is focused on surface oriented modeling for a potentially faster build cycle.

1.5 Framework

The framework used for building models and testing ideas is a C++ application built using OpenGL and Glut. Various interface functionality has been implemented to set and view testing parameters or debug the graphics, such as a command console for input, or highlighting of different parts of the model. The choice of language and API is based on their frequent use in courses on computer graphics at Uppsala University.
2 Theoretical Background

2.1 Tessellation

A tessellation is a tiling of a surface or mesh using a number of geometric shapes such that every tile fits with no gaps or overlaps. A tessellation using triangles is also called a triangulation. A tessellation can be regular (built from polygons of the same shape), semi-regular (more than one shape, but arranged in regular patterns) or irregular. Well known techniques used for surface tessellation include Ear clipping (section 2.1.3) and Delaunay triangulation (section 2.1.4).

2.1.1 Recursive triangle subdivision

This is a commonly known method to approximate a sphere using subdivision of some base shape or model. Examples and an explanation can be found in chapter 6.6 of *Interactive Computer Graphics: A Top-Down Approach UsingOpenGL* [22]. Following is a description of the base method used and extended in this thesis:

- Choose a base shape to apply the algorithm to. These shapes need to be represented with vertices all lying on the surface of the resulting sphere and the tessellation unit is a triangle. We want our triangle mesh to be as regular as possible to avoid problems such as visual artifacts. This method preserves the regularity of the triangles.

- Then, iterate over each triangle and subdivide it into four new triangles: Find the middle of each edge of a triangle, use that point as the corner for a new triangle, effectively subdividing the initial triangle into four triangles. (See figures 1 & 2)

- These new vertices are moved outwards to the surface of the resulting sphere. This is easily done by normalizing the position vector of the vertex if the radius of the sphere is equal to one and the coordinate system origo is located at the center of the mesh.

- Repeat until desired mesh density is achieved.

2.1.2 Base shapes

Good shapes to use with this method are the *Platonic solids* with triangular faces, i.e. the *tetrahedron* (four faces), the *octahedron* (eight faces) and the *icosahedron* (twenty faces). These shapes suit the method well as they are all made of regular, equilateral triangles, properties preserved by the sphere generation algorithm used here [22].

2.1.3 Ear clipping

Triangulation by ear-clipping [23] works by cutting off triangle ears from a simple polygon, thus creating a complete triangulation from the cuts. A convex triangle $\tilde{V}_{i-1}, \tilde{V}_i, \tilde{V}_{i+1}$ that is legal to cut away is called an *ear*, $\tilde{V}_i$ is called the *ear tip*. An ear cannot contain any other vertex than its three corner vertices, and the polygon may not contain any holes.
(requirement from being a simple polygon). This is one of the simplest algorithms to understand and implement, but it is not the most efficient. A common implementation (keeping track of concave and convex vertices) may run in $O(n^2)$ time. There exist more efficient algorithms, but these tend to be more difficult to implement [23].

### 2.1.4 Delaunay triangulation

For a set $P$ of points in a plane, a Delaunay triangulation $DT(P)$ is a triangulation such that no point in $P$ lies within the circumcircle of any given triangle in $DT(P)$ [24]. This can be extended to higher dimensions with circumscribed spheres or hyperspheres.

A delaunay triangulation relates closely to the Voronoï diagram. When connected with straight lines, the centers of the circumcircles in the delaunay triangulation form the Voronoï diagram for the same set of points, thus a Delaunay triangulation is dual to the Voronoï diagram. Delaunay triangulation is named after the Russian-French mathematician Boris Delaunay, a student of the Ukrainian-Russian mathematician Georgy Voronoï. (For the original paper, see Delaunay [25] (in french).)

**Voronoï diagram** The Voronoï diagram divides space into a number of cells based on a number of seed points, one for each cell. Each cell consists of all points on the plane that has that cell’s seed as its closest seed [26].
2.2 Feature Edges

A method used for preserving the main shape of a model when applying mesh smoothing or mesh decimation [27] to it is called Feature edges. It involves tagging a subset of the edges in a polygonal mesh as feature edges to signal any mesh operations to preserve them when performing changes to the mesh such as decimation. This is to preserve the overall shape of the model.

2.3 Fractal noise

Created to fill a void in the computer graphics field when graphics had a too polished appearance, fractal noise brought controlled randomness into the equation. Nature seldom has straight lines or mathematically perfect geometric shapes, as opposed to those that a computer can grasp and easily draw. Perlin noise is a well-known fractal noise algorithm created by Ken Perlin in the 80’s when he was working on the movie Tron, whose look was designed around the current limitations of technology at the time [8, 6, 28].

2.3.1 Fractal functions

The basic idea of fractal methods is to add together several octaves (or scales) of a function, resulting in a new function characterized by a higher complexity. Fractals is one of the most potent ways of adding complexity to any scene. To quote F. Kenton Musgrave [6] a fractal is a geometrically complex object, the complexity of which arises through the repetition of a given form over a range of scales. A common method of creating a fractal function is called fractional Brownian motion, or fBm, and consists of summing a number of consecutive octaves of a base function, each at half the amplitude of the previous octave [29]. See formula (1) for an fBm with 5 octaves based on sine, and fig 3 for a corresponding plot.

\[
\sin(x) + \frac{\sin(2x)}{2} + \frac{\sin(4x)}{4} + \frac{\sin(8x)}{8} + \frac{\sin(16x)}{16}
\]  

(1)

Figure 3: A plot of the fBm of the first 5 octaves of sine.
2.3.2 Lattice and Gradient Noise

Generating one or more uniformly distributed pseudo-random numbers (PRNs) at each point in a space (often 2D or 3D) with integer coordinates forms an integer lattice (grid) and is the start of a lattice noise algorithm. A smooth interpolation of these values takes place. If used in a texture, any translation or rotation would change the texture coordinates ever so slightly, but a PRN generator would give completely different output values from similar inputs and the result would change drastically. Therefore, the noise needs to be smooth, hence the interpolation. There is also an issue with aliasing as the detail of white noise can be of a too high frequency to sample and render. (Related to the Nyquist frequency [6])

Extending the lattice with randomly generated gradients, we approach the method created by Ken Perlin [8], also called gradient noise. A gradient vector is randomly generated at each lattice point and these are used to create the noise function. At each integer lattice point the value of a gradient noise is 0, and the gradient vectors determine the behavior in between these points using interpolation.

Perlin’s original gradient noise method  Here follows a short description of the gradient noise method used by Ken Perlin for three dimensions [6]:

The gradients are generated by first uniformly generating a number of points within the cube $[-1...1]^3$ and then projecting the ones inside the unit sphere onto the sphere surface (the ones outside the sphere are discarded and recomputed or the distribution of points on the sphere surface would be denser near the cube corners). Each lattice point is then given a wavelet, a function that drops off to zero outside of some region and that integrates to zero. In this case, a wavelet is chosen that evaluates to zero outside the unit sphere and in its center. Evaluating the sum of each wavelet defined for a given point (this would be the eight wavelets corresponding to the eight integer lattice points surrounding the point) then gives the gradient noise value for that point.

This results in a smooth noise function that can be evaluated at an arbitrary point without having to precompute other points. This can be quite suitable for a fragment/pixel shader, as a fragment shader evaluates its pixel and does not communicate with its neighbors.

2.4 Glossary

Circumcircle The circumcircle, or circumscribed circle, of a polygon P is a circle with all the vertices of P touching the circle.

Circumscribed Sphere Like a circumscribed circle, but in three dimensions.

Edge An edge of a polygon is a line between two vertices.

Flat shading A simple form of shading where each polygon has one normal vector. This results in flat-looking polygons with sharp edges.

Fragment shader Also called a pixel shader. This shader is run for each screen pixel of an object that is drawn using the shader. Many of these are usually run in parallel using a number of specialized hardware units.
L-system A grammar rewriting system, commonly used to model patterns such as plant stalks and leaf veining.

Nyquist frequency In discrete signal processing the Nyquist frequency is half of the sampling rate. Aliasing will occur if this frequency is exceeded. (The screen resolution cannot handle the smaller detail.)

Phong shading A shading method where normals are interpolated across a polygon face. This results in smooth surfaces with soft transitions between polygons.

Ray casting Casting a ray, often originating from the eye of the viewer, through a volumetric object is a common way to sample the object for rendering.

Ray tracing Similar to ray casting, but often refers to methods where secondary rays are created, often by reflection.

Shader A small program that is compiled and loaded into, and run on, the graphics hardware at runtime. This can contain lighting calculations and control the end result of a pixel color, or even change the position of a vertex.

Vertex A vertex of a polygon is a point where a line segment connects to other line segments, at an endpoint.

Vertex shader The vertex shader is run for each vertex in an object. Can be used for things like moving a vertex or interpolating normals for use in the fragment shader.

Voxel A 3D graphics primitive or building block. (a "Volume pixel")

3 Method

3.1 Explored ideas

A couple of different methods for creating the geometry of rocks have been examined, which will be explained in detail in the following section.

These are:

- **Recursive triangle subdivision**, which will be referred to in this thesis as "sphere inflation" (section 3.2).

- **Render-dynamic mesh with feature edges and corner chipping**. This section also includes some experiments with ear-clipping, initially (section 3.3).

3.2 Sphere inflation

This consists of using the well known method of tessellating a sphere through recursive subdivision of another basic shape, but modifying the positions of created vertices (and the initial ones to some extent) to achieve a desired shape. In this thesis this method will be referred to as *sphere inflation*.

The basic recursive triangle subdivision is described in section 2.1.1.
3.2.1 Extending the algorithm

The main modifications done to this algorithm are:

- rescaling/distorting the initial shape
- modifying the distance that a newly created vertex is moved towards the surface of the sphere
- adding some randomness into the equation

The base shapes used are the tetrahedron, octahedron and icosahedron shapes (fig 4), and also a cube with its sides split into two triangles each, as we are working with a mesh of triangles.

Initial shape The typical base meshes used to approximate a sphere are, as they should be, quite regular in shape. So, for a more naturally random looking (a less regular) shape, it would be a good idea to rescale them somewhat (or perhaps we should say distort, as we do have a large random element). This was done by moving every base vertex by a random distance within different intervals along the different axes. See Randomization below.

New vertices In the basic method (section 2.1.1), when new vertices are created, their positions are simply normalized to the surface of the unit sphere (assuming here that the coordinate system origo is at the center of the original model). By simply multiplying the normalized vertex position vector with a length scalar value, one can easily set the new position (scaled through its distance from the center). This has been the subject of some experimentation with different parameters but the basis of this movement has had two different main ideas: A method based on the global movement of the vertex, let us call it "half-distance movement", and a method based on local attributes, we can call this "edge length movement".
• Global: *Half-distance movement* Move the vertex half the distance between the point it was created and the point in which it would have ended up in the original algorithm (the surface of the sphere). See (2) and fig 5a (let us call $M$ the movement term).

• Local: *Edge length-based movement* With this idea, a newly created vertex is moved outwards a predetermined fraction of the length of the edge on which the vertex is created. (See fig 5b.) The direction remains the same as before, in a straight line from the center. (Before any random displacement occurs). Here, the movement term is $M = L_{edge} \times k$, where $k$ is the predetermined fraction of the edge length $L_{edge}$ that we want to move our new vertex. See (3).

\[
\vec{V} = \vec{V}_{surface} \times (|\vec{V}_{initial}| + M), \quad M = \frac{|\vec{V}_{surface}| - |\vec{V}_{initial}|}{2} \tag{2}
\]
\[
\vec{V} = \vec{V}_{surface} \times (|\vec{V}_{initial}| + M), \quad M = L_{edge} \times k \tag{3}
\]

![Figure 5: Vertex movement: Half-distance vs edge length-based movement](image)

An addition to this is the extending of this vector scalar multiplication to a vector vector multiplication, using a vector of scaling factors that differ in the different axes. This would be for consistency with the axis-depending rescaling of the base shape, and be based on the same calculations and randomization.

**Randomization** Some randomization needs to be in place as we want the ability to create multiple rocks without them looking exactly the same (see 1.4) This means we must somehow apply some random factor to our model. This is also a large part of our rescaling operation.

Firstly, this means distorting or rescaling the base shape to get a form that is more akin to what we want; in this case, the vertices are moved randomly within specific ranges (fractions of the original value) for different coordinate axes. Secondly, the new vertices created in the recursive subdivision algorithm are moved randomly within similar ranges as those of the base shape.

The typical randomization ranges used, which seem to work fairly well, are randomizing the $y$-component of the vertex in the interval $[\frac{1}{3}, \frac{2}{3}]$ and the $z$-component in the interval $[\frac{1}{2}, 1]$. The $x$-component remains the same, or we would change the overall scale of the object for no reason.
For the initial mesh, the vector components of the initial mesh vertices are multiplied with random values in the given ranges, and as for the rest of the vertices, the random values are multiplied with the movement term seen above, see (4).

\[
\vec{V} = \vec{V}_{\text{surface}} \times (|\vec{V}_{\text{initial}}| + M \times R), \quad M = \frac{|\vec{V}_{\text{surface}}| - |\vec{V}_{\text{initial}}|}{2}, \quad R = \text{random factor} \quad (4)
\]

**Recursive smoothing**  In an attempt to better scale the algorithm to the increase in mesh resolution, and perhaps to lessen some of the dramatic effects that randomness can produce, a smoothing effect was added (that could be toggled, to compare its effects), decreasing the magnitude of any vertex movement depending on how far we have come into the recursion. This makes changes to the mesh more and more smooth as we progress through the recursion. This displays a very visible effect as we can see in the results section. This is done by multiplying the movement term with a smoothing factor proportional to the inverse of the recursion step number, see (5).

\[
\vec{V} = \vec{V}_{\text{surface}} \times (|\vec{V}_{\text{initial}}| + M \times s \times R), \quad M = \frac{|\vec{V}_{\text{surface}}| - |\vec{V}_{\text{initial}}|}{2}, \quad s = \frac{1}{\text{step number}} \quad (5)
\]

**Phases**  Another attempt at scaling the algorithm (see 3.2.1 above) is the use of two different phases with different growth parameters for the mesh. The second phase has a weaker growth than the first, approximately by a factor of 2. Replacing half-distance movement (See 3.2.1) with a quarter.

### 3.3 Recursive subdivision of segmented edges, including feature edges and corner chipping

The idea with this method is to create another kind of mesh than the previous, with stored information about feature edges to retain some general shape, while chipping off corners to create some kind of faceted and hopefully rock-like shape. The idea is that this could potentially be a good general method for creating rocks from any desired input shape. Corner cutting/chipping would be a kind of erosion.

**The general idea**  We begin with a mesh having some general shape along the lines of a box or a brick. We then proceed to tessellate it while we add some random factor to the vertices to make it more uneven, or less regular, and finally we start chipping off corners.

#### 3.3.1 Tessellation

So, how does one tessellate this box in a good way? Let us first say that we will keep using the triangle as tessellation unit as this is the most common and simple way. Since long thin triangles is usually undesirable, we want a well formed (near-equilateral) mesh of triangles, but as we want to move the vertices about somewhat to create a more rugged/uneven or random look, it might be a good idea to do the randomization before
or while tessellating. This is because we will otherwise distort the mesh and possibly make it less well-formed if we do a pass on it afterwards for some "roughing up".

Not only do we want the triangles to be well-formed, we also need all the triangles fitting together in a complete mesh. This is our first problem here; how do we tessellate a mesh of triangles in a randomized but still consistent way? What if we were to look at the two longest edges of a triangle and divide them into an equal number of sections? Then what if two neighboring triangles are tessellated differently? And what about the third edge? We could have the issue of triangles not fitting together in a mesh. We cannot simply look at a triangle, tessellate it, and move on with the rest, we have to begin by looking at the edges by themselves, or the resulting triangles will not fit together.

**Edge segments**  A solution to this would be to first divide the triangle edges into segments, and then proceed with the tessellation of the rest.

How do we then define these segments? The choice was made to define a unit size constant $S_{\text{unit}}$ representing the minimum desired edge length in a given triangle in a resulting tessellation. The number of segments $N_{\text{seg}}$ constituting a given edge would then be $N_{\text{seg}} = \text{floor}(S_{\text{edge}}/S_{\text{unit}})$ where $S_{\text{edge}}$ is the size (length) of the edge. The length of every edge will likely not be evenly divided by any unit size. The segment length can thus vary from $S_{\text{unit}}$ up to $S_{\text{unit}} + S_{\text{edge}}/N_{\text{seg}} + r$, where $r$ is a small random number.

**A note on random vs exactness**  The random number can and will change the total edge length when we move the vertices in any direction non-parallel with the general direction of the edge, potentially breaking invariants like the segment length rule above. This makes it important to remember keeping the random changes small so that it does not affect the algorithm too much, but we will assume our algorithms to have some leeway; they are not made to be exact. This needs to be kept in mind, however. Unit size ($S_{\text{unit}}$) will thus be more of a wish than a demand.

To generalize this: For the sake of simplicity, when speaking of polygons in the rest of this section we shall imagine that the polygon lies in a two-dimensional plane and assume that the random displacement in the third dimension is small enough that it does not affect any algorithm too much.

**From triangle to concave n-sided polygon**  We can then generate a number of new vertices along the edges of the input triangle, thus creating an n-sided polygon (although it will initially still look like a triangle). If we then slightly, randomly displace these vertices, we will end up with a concave n-sided polygon (fig 6). Therefore we need a tessellation algorithm that is able to work with a concave polygon.

![Figure 6: Concave polygon](image)
Choosing a tessellation algorithm  There are several possible ways to tessellate a concave polygon, and here are a few methods that were first considered:

1. *Ear clipping* is a tessellation strategy where one marches along the outer edges of a polygon and cuts away triangle by triangle, until nothing remains. (section 2.1.3)

2. *Subdivision of a polygon in "waists"*. A vertex that has the largest overview or visibility of the interior of a polygon is probably a waist, and this can be used as a natural divider of polygons. However, this might not be as relevant in this case, as the triangles are not likely to have waists. If the triangles have predominant waists, they are probably much too distorted by randomization and this is not desired. If we do have a strong random displacement however, this could prove to be useful.

3. *Delaunay triangulation* could be a possible solution to the problem, with points randomly generated inside the polygon, and perhaps special cases for the borders of the polygon. (section 2.1.4)

4. A *volumetric corner chipping* method where one begins with a cube or such and chooses corners to "chip away" to greater or smaller degrees. This leads to an implicit triangulation of the voxels.

5. A *recursive subdivision scheme* like the tessellation used in the *sphere inflation* method, except basing the division on the segments and their physical lengths.

Attempts have been made with #1 and #5 (followed by #4 to approach the shape of a rock).

### 3.3.2 Ear-clipping

(See 2.1.3 for background on the method)

For our purposes, we shall assume that our polygon is shaped well enough that, along the path around a polygon, every convex vertex $V_i$ is a valid ear tip, implying that the potential subtriangle $V_{i-1}, V_i, V_{i+1}$ does not contain any other vertex within itself. The implementation created in this thesis simply marches across the length of the triangle, cutting away the ear formed by the three vertices that are the farthest away from the corner of the two longest edges.

See section 4.2.1 for illustrated results.

### 3.3.3 Ear-clipping (modified)

As results show in section 4.2.1, the implementation of the unmodified ear-clipping algorithm tested here was not very well suited for this task, so a couple of additional criteria were applied to the algorithm to try to determine whether it may still be possible to get better results. The criteria added were these:

1. For each iteration, choose the vertex along the outer edge with the smallest angle, i.e. cut off the sharpest ear.
2. Insert a new vertex if a newly created edge is longer than (or equal to) $2 \times S_{\text{unit}}$

The input of the algorithm will be an n-sided concave polygon, (created from our initial triangle with a number of vertices generated and slightly randomized along its outer edges, see fig 6) This can simply be thought of as a circular list of vertices. We will then walk one lap around the outer edge of the polygon, look at every vertex (or ear tip) $V_i$, and compare the angle it forms with its two connected neighbours $V_{i-1}$ and $V_{i+1}$. Together, these form a potential subtriangle, or an ear. The ear with the smallest angle at vertex $V_i$ will then be chosen and added to the final mesh. However, if the distance between $V_{i-1}$ and $V_{i+1}$ is longer than $2 \times S_{\text{unit}}$, a vertex will be inserted in the middle to make two triangles that are closer to being equilateral (fig 7). This is because we know the distances from $V_i$ to its two neighbours are close to $S_{\text{unit}}$. We then keep iterating, walking another lap finding the new smallest angle, and so on, until everything is done. The simple base case is when there is only three vertices left to tessellate. Since the triangle is the smallest possible tessellation unit this will be the base case. (Assuming that the edges of the original polygon are not randomized to such a degree that they are somehow overlapping, making it a complex polygon. The algorithm is not defined for complex polygons.)

Note that for every new vertex created in this algorithm, a small random displacement was included to add some roughness to the mesh. As this algorithm was not used further, it is not that important. It can however be seen in the illustrations in the results section (4.2.2.)

Figure 7: Splitting the edge that is too long

### 3.3.4 Recursive subdivision (revisited)

This is a recursive subdivision method similar to the one in section 3.2. Beginning with the same box formed mesh as in the ear-clipping experiments in sections 3.3.2 and 3.3.3, we have a mesh in the shape of a box, with every polygon face being a slightly distorted triangle, or in other words, a concave n-sided polygon with the general shape of a triangle. We then split every triangle-like polygon face (i.e. a triangle with segmented edges, slightly randomized, see Edge segments in section 3.3.1 above) into four triangle-like parts much like the triangle subdivision method in 3.2, but instead of creating a vertex in the exact middle of the triangle edge, we use the vertex closest to the middle of each "triangle edge" (the set of segments formed from one edge of the original triangle before segmenting) as a divisor point. These three vertices become the corner points for a new triangle forming the center part of the four new polygons, effectively dividing the face into four "triangles", very much like the previously mentioned method (see fig 8.) This new triangle is, in turn, made into an n-sided (concave) polygon by creating new vertices along the new edges and displacing them randomly within some interval as described in Edge segments and From triangle to concave n-sided polygon in section 3.3.1.
3.3.5 Feature edges

The idea is to preserve the status of the edges of the original shape, in this case a box, as feature edges. This will help algorithms in any following work on the resulting tessellated mesh to better know the general shape of the mesh, and hopefully to be able to preserve some basic form, if desired. This is especially important for identifying corners to chip away later on.

When creating the box shape used for testing these methods, every original edge of the quadrilateral sides is given feature edge status, but not the diagonal edges that divide the quadrilaterals into triangles, as they are not contributing to the general form, only to the mesh topology.

Feature edges are simply incorporated into the mesh data structure and kept as an invariant along every mesh operation.

For the sake of simplicity the mesh is limited to exactly three feature edges per corner, making a general box shape suitable for this. This is to make the corner chipping algorithm simpler, but it should be possible to extend to more cases. (Fig 9)

![Figure 8: Subdivision of a concave polygon](image)

a) ![Figure 9: a) Illustration: A basic box with feature edges in black, regular edges in grey. b) Screenshot.](image)

3.3.6 Corner chipping

The cutting or chipping of corners could represent erosion or other damage done to the sharper corners of a piece of stone or rock. Using many iterations should result in a rounder shape, perhaps a more eroded rock, while using less iterations might represent stone blocks with more flat surfaces, as if broken off from larger pieces of rock by cracking (due to temperature changes for instance).

Finding the corners is done by using the feature edges, as a corner can be defined as a vertex where multiple feature edges connect. Exactly three of them in our case, as per the chosen limit (as stated in Feature edges above).
Chipping a corner  When we have found a corner we want to chip off, we use a predefined proportion constant to decide how much of each feature edge to cut away. We do not move or create any new vertices, we cut away whole mesh edges (or segments, from the feature edge’s point of view). The new end vertices of each of the three shortened feature edges will now form a cutting plane which we can use to complete the cut, removing all vertices that are located on the “wrong” side of the plane, i.e. the outside (or the side that contains the corner to be cut). This hole that we have just made in our mesh needs to have defined borders, and we should not want any feature edge to end abruptly, or end in anything other than a corner along with other feature edges. Thus we need to make these borders into feature edges as well. (Fig 10)

We now have a triangle-like hole in our mesh, and we need to fill it with new triangles. So, we need to tessellate a triangle-like n-sided concave polygon. Sound familiar? Let us simply apply our subdivision-based tessellation algorithm to fill the hole.

The method in detail:

- Traverse the three feature edges away from the corner the given distance, building a cutting plane from these three reached endpoints.
- Check every face with all its vertices outside of the cutting plane, making all its edges feature edges and removing the face. The plan was to implicitly build the feature edges of the corner this way.
- The feature edges are then traversed, finding the corners of the new triangle.
- Apply the tessellation scheme to this triangle.

![Figure 10: A corner defined on the box (only showing feature edges) corner vertices drawn in red, regular vertices in orange](image)

3.4 Shading

The shading in this project has been implemented using GLSL, the OpenGL Shading Language.

3.4.1 Normal vectors

Normal vectors are calculated for faces and for vertices to be able to use different shading methods. This is done in a separate pass. The face normals are calculated simply by iterating over each face and taking the cross product between two of its edges to get a vector orthogonal to both of these edges. The *outwards* direction is determined by the order in which the vertices are referenced within a face. The vertex normal vectors are calculated by iterating over each vertex and searching for the faces connected to it, summing these face normal vectors and normalizing the length of the resulting vector.
3.4.2 Phong and flat shading

(See the glossary for a short explanation of the difference, 2.4) In most images of results in this thesis (the green ones) a mixture of phong and flat shading has been used. (The choice of green was random, and is mainly for the purpose of viewing the modeling results, not for any ultimate rock appearance.) The plan was that pure phong would look much too smooth, while pure flat shading would have a much too geometrical look with its many flat faces, and a combination of them would have the smoothness of phong and the roughness of flat shading. (The application GUI has controls for the interpolation between them, but most of the images show approximately a half/half mix.)

This mixture is implemented simply by, in the vertex shader, using linear interpolation between the normal vector of a triangle face and the normal vector of the vertex. This is then interpolated for each fragment by the GPU.

3.4.3 Fractal noise-based 3D textures

Some of the images in this thesis show results from experimenting with a gray granite shader. This shader is based on the granite shader example from chapter 15 in *OpenGL Shading Language* [29], where a 3D texture is used to precompute and store four octaves of Perlin noise in the four color components (red, green, blue and alpha) of the texture. The 3D texture is passed to the shader and these values can then be used in different ways. A granite shader from the same chapter is the basis of the shader implementation used here. It uses only the fourth stored component for a high frequency noise resulting in a granular pattern. The scale of the granularity can be changed with a scale variable that can be changed within the application. This is simply multiplied to the texture coordinates, based on fractal noise shader examples in *OpenGL Shading Language* [29].

4 Results

4.1 Sphere inflation

An important detail to remember is that, as one iterates over all triangles, every edge is visited twice, and the code needs to take this into account when creating new vertices, so that it does not simply iterate over all triangles and create vertices twice for every edge. This does not matter as much in the original algorithm, but as things start to get randomized, holes will appear in the mesh when doubled vertices are given different random values. This was an effect experienced during early development.

**Number of iterations?** The number of iterations our algorithm runs is a key factor. If too few iterations are run, the results will not be detailed enough and the overall shape will be simply too flat and artificial. If too many are run it will produce artifacts such as artificial ripple effects on the surface. (See figures 11-14.)

Thus, a determining factor for success is finding a good number of iterations that is low enough so that ugly artifacts do not start to appear but high enough so that satisfying results are attained. This depends on the mesh density of the base shape, but also on the shape itself. More on this in the following paragraphs.
**Base shapes**  Varying the parameters, the different base shapes give somewhat different outcomes but unless the parameters have been set to great changes (such as disabling recursive smoothing), similar base shapes result in fairly similar final shapes. This works in favor of the isosahedron, being closer in its initial shape to the shape of a (round) rock than the other shapes are. Also, the sharper angles/corners in the lower density (lower polygon count) base shapes can remain and may give an artificial look to the resulting model, displaying an artificial (i.e. too perfect) sharp corner. Thus, a shape with less sharp corners would be preferable, such as the icosahedron due to its higher mesh density combined with its regular shape.

Figures 11 - 14 show the first few iterations using most shapes that have been explored here. This shows how a higher number of iterations can make the surface highly irregular, especially with the global vertex movement strategy used in those images. This is somewhat compensated for by using the recursive smoothing approach.

![Figure 11: Sequence of 0 to 5 iterations using tetrahedron as base without smoothing, followed by 2 to 5 iterations using smoothing](image1.png)

![Figure 12: Sequence of 0 to 3 iterations using a cube as base without smoothing, followed by 2 to 3 iterations using smoothing](image2.png)

**Recursive smoothing**  This has a strong effect on the resulting model. The difference between the results without and with recursive smoothing can be seen in figures 11
- 14. In many cases it is a necessary step to decrease high frequency surface irregularities at higher iterations, and is used in all images from here on unless explicitly stated otherwise.

**Global vs local vertex movement**  In general, the difference between global and local vertex movement is that the local method produces less erratic surface shapes, while the global can produce greater surface change with fewer iterations. Generally, the local one is preferred as it gives more predictable and coherent shapes. Although a less smooth surface would be desirable on a rock, more predictable lumps and other features can be achieved by varying other parameters (more on this in section 4.4).

See fig 15 for a few (exaggerated) smooth icosahedron based samples using local vertex movement.

**Surface irregularities, valleys and pits**  As vertices never move after their initial placement, an interesting and possibly problematic effect can arise when subsequent neighboring vertices are created and positioned. If the difference in distance from the center of the rock to a vertex compared to its neighbors is large, surface irregularities can grow large. In general, surface irregularities is an important property of a rock: rocks in general should not be too smooth. However, the irregularities can grow and form ugly
visual artifacts if too many iterations are run. This happens especially when using the
global vertex movement strategy, as it does not take into account the surrounding area
of a vertex when determining its placement, apart from its initial position. If the vertices
around a whole edge are moved far a valley can appear, and if the vertices around a lone
vertex are moved far this would create a pit.

A pit or a valley in itself is not really a problem unless too exaggerated. If the recur-
sion keeps going too far this can emphasize such pits or valleys, as can be seen on the
cube in fig 12 and all across the surface on any example gone too far, such as the 4th
iteration icosahedron without smoothing in fig 14. Valleys in particular can be promi-
nent on shapes where the base shape has been randomized in such a way that the angle
between two neighboring triangle faces has become very small and the middle, common
edge is much lower than its two opposing vertices. (see illustration in fig 16 and fig 17)
This is then enhanced with further iterations of the algorithm.

Similar to valleys, when every vertex around a single vertex $V$ is moved significantly
further than $V$, creating a pit in the surface, the feature is further emphasized on further
iterations. The chance of this happening is higher when using a global vertex movement
strategy.

Figure 16: A beginning valley between two faces. Exaggerated for illustrating the point.

Figure 17: Valley cross section: This shows a cross section of the surface of a valley and how
the shape is exaggerated by further iterations.
4.2 Recursive subdivision of segmented edges, including feature edges and corner chipping

4.2.1 Ear-clipping

The ear-clipping strategy is fairly easy to understand and to implement, but results show that the first attempt at its implementation was not very well suited for the current goal. In the first implementation it produced long, thin triangles, and not the near-equilateral triangles preferred (see fig 18.) This would depend on the ratio between the width of the face we want to tessellate and the unit size constant $S_{unit}$. The closer that the width of the face gets to $S_{unit}$, the closer to equilateral our subtriangles should become, as this implementation of the method simply cuts slices across the face (perpendicular to the longest edge). However, nothing here can be done about the fact that the width across the polygon will always vary along the length of the face as we are dealing with a polygon in the general shape of a triangle, and thus we cannot find one single value for our unit size to fit the varying width of the polygon.

It might be useful for tessellating polygons that are very long and thin, where the width does not vary much from end to end. Otherwise, vertices need to somehow be generated within the triangle, but

![Image](image.png)

Figure 18: Standard ear-clipping used on a box with the quadrilateral sides first split into two triangles.

4.2.2 Ear-clipping (modified)

The changes to the standard ear clipping algorithm resulted in an algorithm showing very different results from the original one.

Using larger unit sizes, the tessellation might not be perfect but it is not too bad. On smaller unit sizes however, we can clearly see a trend where it converges to a circle in the center of the face.

The algorithm walks around the outer edge of the triangle, looking at every vertex, comparing the angle it forms with its two neighbours and choosing the one with the smallest angle. Therefore, in retrospect, it feels natural that it would converge towards a circle, as we are cutting away the sharpest corners first, making it more and more smooth for every turn. A smaller unit size means a higher resolution of the tessellation and a more evident trend. See fig 19.
Figure 19: The modified ear-clipping approach applied to the same box as in fig 18, using different values for the unit size constant $S_{\text{unit}}$. The mesh has been highlighted in a purple-blue shade to better see the tessellation. You can clearly see the trend when we get to lower unit sizes (to the right).

4.2.3 Recursive subdivision

Unit size We have not managed to keep the unit size constant invariant during the entire algorithm. Some edges get quite short during the subdivision steps. Every new "triangle" dividing its parent abides by the unit size constant when creating its edges, so this seems to be at the end of recursion, when there is not much space left to create good triangles, and we are left with fewer options to tessellate.

Edge collapse A solution could be edge collapse of too short edges, where edges that are too short are simply removed, also removing the two triangles using that edge, and combining the two end vertices of the edge. Perhaps this might be done either (preferably) built into the recursion or as a separate pass.

Figure 20: Tessellation by recursive subdivision

Figure 21: a) Regular cube with a corner cut. b) Randomized box with a corner cut (including example of a glitch afflicting the code)
Corner chipping Unfortunately, this part was not successfully completed. It is difficult to say whether the random factor discussed earlier affects this, or if it is solely due to bugs in the code, but there are evidently bugs in the code, as chipping of other corners was problematic. So, some sort of geometry or orientation based bug still remains.

Although, some images to display the results thus far can be seen in figs. 21-22. As the project cannot last forever there is not enough time to finalize the code, so there will not be a completed result for this part of the project.

4.3 Shading

4.3.1 Fractal noise-based 3D textures

The granite shader built from examples in OpenGL Shading Language [29] can be seen in several of the illustrations in this thesis, such as fig 20, 21b and 22. It seems to fairly approximate the appearance of granite at certain scales of granularity. Fig 23 c) displays some aliasing when rotated, displaying a too high frequency texture.

Flat to phong shading interpolation Using the shpere inflation technique, the effects of this interpolation approaches pure phong shading when higher iterations of inflation are run, as the effects of the sphere inflation eventually roughly approximates phong shading when using recursive smoothing (section 3.2.1). The effects of the flat shading are not positive, however, as they only enhance the appearance of regular triangles in the mesh (fig 24).
4.4 Interesting Icosahedrons, Octahedrons and lumpiness

4.4.1 Icosahedrons

The Icosahedron using 2-3 iterations of local vertex movement (fig 15) could be a possible candidate if an oval, smoother rock is desired. Especially as they seem to be consistent in not displaying any odd artifacts due to randomness, and can thus be generated in limitless numbers varying the random seed, with some variation in shape. These need some good texturing and shading to appear real, however. Perhaps some bump mapping might be useful.

The icosahedrons in fig 25 and in fig 26 do show some roughness, and the surface irregularities can be positive as they are not too exaggerated.

Another interesting result is the icosahedron made with a higher number of iterations, but using the local vertex movement, which gives a weaker/smmerier growth. This

Figure 24: The images show an icosahedron of 3 iterations of the local variety. a) is shaded with an interpolation between phong and flat shading (not unlike pure phong in this case) and b) is shaded with flat shading, both included within the granite texture shader. (1280 polygons)

Figure 25: Different angles of a rock using 2 phases of global vertex movement. Phase 1: 1 iteration, phase 2: 3 iterations (5120 polygons)

Figure 26: Different angles of a rock using 2 phases of global vertex movement. Phase 1: 1 iteration, phase 2: 3 iterations (5120 polygons)
results in shapes that look generally rock-like and a surface that has some minor cavities (fig 27-28). However, some of these results can display curves along the surface that are somewhat geometrical and regular in appearance. These are parts of main edges of the base shape and are being emphasized by further iterations of inflation (fig 28c). Likely for the same reason as the appearance and emphasis of valleys (see valleys and pits in 4.1 and fig 17.)

4.4.2 Octahedrons

Octahedrons have not been explored much here, but some configurations can yield promising results. A nice example can be seen in fig 30.

![Figure 27: icosa local 5, from different angles (20480 polygons)](image)

![Figure 28: icosa local 4, from different angles (5120 polygons)](image)

![Figure 29: Same as fig 28 but with a bigger vertex randomization interval (i.e. they are generally pushed further outward) This is slightly exaggerated, however. A weaker growth generally yields better looking results.](image)
4.4.3 Chunks and lumps

If a chunkier or more lumpy rock is desired, it is possible to increase the ranges of random factors used to push new vertices further outward, creating greater differences in surface height (from the center) between vertices (especially between new and base mesh vertices as they never move), resulting in a surface with stronger (lumpier) features. The more we do this the higher the risk of receiving strange shapes, but with moderate changes to these variables we get fairly realistic rocks with a more lumpy or chunky surface. Decreasing the number of iterations or employing the use of a weaker phase (see Phases in section 3.2.1) may lessen this risk also. (fig 31, 32)

Figure 31: A more chunky rock, using an icosahedron with two ‘phase 1’ and one ‘phase 2’ iteration, and a bigger vertex movement range than most other images in this thesis. a) shows one using global vertex movement and b) shows one using local vertex movement. (1280 polygons)

Figure 32: Another rock with somewhat stronger features, icosahedron with two ‘phase 1’ and two ‘phase 2’ iterations, bigger vertex movement range and using local vertex movement. Shown from 2 different angles. (5120 polygons)
4.5 Speed

The main tasks performed are precomputing the Perlin noise data and storing it within a 3D texture, creating a base mesh and running our algorithm on it, calculating normal vectors and finally texturing/shading of the model.

Testing computer specs: ASUS Transformer Book T100T with an Intel(R) Atom(TM) Z3740 CPU running at 1.33 GHz with 2.0 GB of RAM and built-in Intel(R) HD Graphics. OS: Windows 8.1

3D noise value texture precomputation runtime

- A size 64 3D texture takes 610ms
- A size 128 3D texture takes 4307ms

As a method of precomputing the noise data suggested in OpenGL Shading Language [29] was used, this is done only once (at application startup).

Running the algorithm and calculating normal vectors on an icosahedron

- 3 iterations of subdivision resulting in 1280 polygons takes approximately 22 ms, and calculation of normal vectors takes around 110 ms
- 4 iterations of subdivision resulting in 5120 polygons takes approximately 83 ms, and calculation of normal vectors takes around 700ms
- 5 iterations of subdivision resulting in 20480 polygons takes approximately 178 ms, and calculation of normal vectors takes around 6725 ms

Normal vector calculation would easily be improved with a mesh keeping track of vertex-to-face connectivity (i.e. doubly linked), much like the mesh used in the second major approach: Recursive subdivision of segmented edges (section 3.3). The one used here simply iterates over all vertices and searches the array of faces (for each vertex) to find the faces connected to each vertex, resulting in a complexity of $O(V \times F)$, where $V$ = the number of vertices and $F$ = the number of faces/polygons in the model. Keeping track of this connectivity with a constant time lookup table for each vertex would make this $O(V)$.

Texturing/shading is done each frame and as such there are no similarly comparable values for time comparison. Although, normal vector calculation could be considered part of this.

4.5.1 Polygon count

In general, the polygon count for the sphere inflation method is given by $B \times 4^s$, $B =$ the number of faces/polygons in the base shape, $s =$ the number of steps/iterations run. This is simply because at each step every face is subdivided into four new ones, multiplying the polygon count by four. The base shape used mostly here is the icosahedron with its
20 faces. (The tetrahedron has four and the octahedron has eight faces. The cube would have \(6 \times 2 = 12\) faces, as the sides are split to create triangles.)

The polygon count of the recursive subdivision of segmented edges approach is not possible to predetermine, as it depends on the shape of the mesh during tessellation, and new vertices are created as needed.

5 Discussion

5.1 Sphere inflation

5.1.1 Shapes of rocks

If a boulder of a fairly eroded/rounded kind is desired, the sphere inflation algorithm does, with the correct input parameters, seem to generate a fair approximation of a rock geometry, and can do so in fairly diverse shapes by changing the input random seed. These do share the tendency towards oblong shapes (A feature decided by the initial base mesh distortion. This is possible to change, but the random factor in this is big enough to create a fair variety of shapes.) and have some differing surface features. The key lies in finding the correct input parameters for the desired shape. (see fig 25 and 26)

The vertex movement outward can be randomized to a higher degree to attain rocks with a more lumpy surface, or to a lesser degree for smoother rocks.

Valleys, as described in 4.1 can be useful by giving the rock some character, as long as it is not too regular or too exaggerated. As stated earlier, the difficulty lies in hitting the balance between the extremes.

5.1.2 Surface detail

Smaller physical detail on a rock surface is probably best left to shading with bump mapping and such techniques. It seems unnecessary to have a too high level of detail at a geometrical level as it would require many polygons to achieve, increasing the hardware requirements of rendering.

However, one major reason for having a slightly higher polygon count, as opposed to only using textures or shaders, is the shape of the contour of a model. Fragment shaders are good at improving the detail of a surface using methods such as bump mapping, but the contour of a model remains unchanged [30].

A possible improvement could be a slight distortion step on all vertices of a rock mesh after everything else is done. This could perhaps add some surface roughness and decrease the appearance of any regular surface curves/lines.

Another use for the smoother rocks could be gemstones, using them as they are with no need for any bump mapping, perhaps only a phong shader using a specular light component, whereas using the model for a regular rock should perhaps be done without specular light. (For instance the smooth shapes in fig 15)
5.2 Recursive subdivision of segmented edges, including feature edges and corner chipping

**Edge segment rules possibly broken by random factors**  Depending on how much we allow the algorithms to randomize things, we will risk breaking the rules of the edge segments, i.e. moving the vertices slightly in some random direction might make the distances between the vertices too great or small, thus making some edge segment too long ($> 2 \times S_{\text{unit}}$) or too short ($< S_{\text{unit}}$). This is not crucial however, and as long as the polygon remains a simple polygon the algorithm should not fail. (However, as this part of the project was not finished, this can not be claimed with certainty.)

**Triangulation methods**  Perhaps a Delaunay triangulation (maybe in 3D space for the model or in 2D for the polygons) might be worth looking into. Alternatively, the ear-clipping method might have been useful if a set of vertices were generated uniformly within a given polygon, as this would create triangles of a more equilateral shape than those of the ear-clipping method used here.

**Corner chipping**  Parameters could control stopping the recursion at sharpness of corner, length of edge, or some other size- or orientation related feature such as making some sides flat and some sides more rounded.

As the corner chipping algorithm was not finished and the prototype still suffers from bugs, it cannot be evaluated properly. However, it remains an interesting option to explore.

5.3 Goals achieved?

How does the end result measure to the goals?

Using the sphere inflation method, the icosahedron is able to grow into both smooth boulders and somewhat chunky rocks. The rocks generally resemble boulders of a slightly eroded kind in the respect that corners and such are rounded.

The recursive subdivision scheme with corner chipping did not result in a useful method here. It remains unknown whether it would have evolved into something useful. However, the results with the sphere inflation method turned out fairly well. (This quality is difficult to quantify, but they seem to appear fairly real.)

The shading could improve considerably, with different procedural shaders for different types of rock, but the granite shader used here will suffice for the scope of this project.

Some diversity was accomplished in regard to shapes, again with the sphere inflation method. Different random seed values can be used as input to create rocks with some diversity within a set of input parameters, and some parameters can be tweaked to change the overall shape and generate rocks that are more or less oval or chunky/lumpy. Some roundness is hard to get away from with this method, however. Also, the base mesh greatly affects the end result as its vertices are fixed. Changing the base mesh changes the main shape of the end result.

Run time depends mainly on the number of iterations that the sphere inflation is run, and this depends on what type of rock is desired. But using the common example of
three iterations, we finish at approximately 132 milliseconds per rock, including normals
calculation but excluding noise matrix precomputation. This gives us the possibility of
using this method fairly interactively in a trial-and-error way or creating a larger number
of rocks quickly. However, as the number of iterations goes up, this quickly takes much
more time, mainly due to normal vector calculation. This can be fairly easily improved,
however.

For the most part, few iterations should be sufficient and this would also allow for
immediate feedback to the user. An interactive experience when modifying parameters
enables the user to explore different options in real time.

5.4 Comparison with other work

How is this project different from other methods?

Looking at Dart et al. [21], one can note some conceptual similarities between the
sphere inflation method described here and the implosion step in their method, although
reversed. They use a sphere to enclose the rock object and extract a surface from their
volumetric method by ray casting inwards and stopping at the rock surface. The sphere
inflation method begins with a base shape (usually a platonic solid with triangular faces)
and expands outwards towards an enclosing sphere, but stopping approximately halfway
(generally less than halfway). This together with the subdivision encompasses the base
of the sphere inflation method, while in the case of Dart et al. [21] it is simply the
isosurface extraction step at the end.

At the point of writing, the application by Dart et al. [21] was not available to test, so
no direct comparison of their efficiency could be made. However, from the complexity
of their volumetric approach with evolutionary search with erosion combined with the
isosurface extraction step it can be assumed sphere inflation is a computationally sim-
pler method being a fairly basic modification of a commonly known sphere generation
algorithm.

It seems the method of sphere inflation allows for a more interactive and rapid rock
generation, however a few downsides would be its dependence of finding the correct
parameters (mainly the number of iterations and the RNG range factor affecting vertex
movement) for different appearances, and its dependence on base mesh used. With a
good combination of parameters however, it can reliably generate many various rocks
with similar characteristics only by varying the input random seed.

Regarding appearances, both the method by Dart et al. [21] and this method have their
merits, and it is difficult to objectively say which is more useful in this regard. Many
rocks in this thesis have a fairly high polygon count, providing high geometrical detail,
which might be more desired in cases such as non-interactive images or precomputed
animations and less desired for such a simple object as a rock among many other objects
in a computationally demanding interactive environment. However, the polygon count
can be decreased a great deal simply with fewer iterations of sphere inflation, sacrificing
geometrical detail. No statistics were available on the polygon count of the rocks by
Dart et al. [21], so no direct comparison of this was made.
6 Conclusions

As virtual environments are widely used in today’s society, many man-hours can be expended to build such scenes by hand. To alleviate the issue with high development times, procedural techniques can be applied to automate certain processes, a task well suited for computers.

Rocks and boulders are often part of virtual scenes to make them look more authentic. This thesis has explored a few methods of procedurally generating these.

The goals in this project were to generate believable rocks with decent diversity, using efficient surface-oriented methods.

The sphere inflation method can, with the right input parameters, generate fairly realistic geometry for models of rocks or boulders. Using different random seeds, this method can generate diverse models with similar characteristics (mainly the overall shape and lumpiness). Generation of specific categories of rock types has not been solved. However, a major part of such classification might be done in programmable shaders, working with surfaces and colors to distinguish different categories of rocks. Although programmable shaders have been used in this project, the focus has been on geometry.

By the nature of this method, however, the resulting models tend towards round shapes, especially on higher vertex densities. Thus, rocks with sharp edges are difficult to generate.

As sphere inflation is a fairly simple method, it runs fast and results can be viewed reasonably quick, mainly depending on the vertex density of the resulting model. This allows for interactive trial-and-error experimentation with parameters to arrive at a desired result, at least with models with a lower vertex density.

The recursive subdivision of edge segments method combined with corner chipping may prove a useful technique, but was not finished within due time, and can thus not be evaluated.

References


