Driftworld Tectonics 1.0: an overview

Adalbert Delong

19th May 2022

Driftworld Tectonics is a project written for Unity editor, used to create basic forms of planets. It utilizes ideas and methods described in an article by Yann Cortial et al. in 2019 [1]. Planets are created by reading basic template topology, performing a simplified tectonic simulation of the planet's crust and then exporting the raw data to customized binary files. Users can display various overlays of the planet during the simulation, as well as see the surface elevation mesh.

This documentation describes some basic theoretical framework, details of the simulation and the implementation. Hopefully it can be of use to anyone interested in this topic, who either just wants to play around with Driftworld or build their own projects.

Acknowledgements

I would like to thank Dr. Yann Cortial of the National Institute of Applied Sciences of Lyon for discussing his article. His answers to my various inquiries helped me decide the scope and form of Driftworld. I would also like to thank Dr. Daniel Meister of AMD Japan Co. Ltd. for his comments on the use of bounding volume hiearchy algorithms. Driftworld implements a part of an algorithm described in one of his publications [2]. More thanks to PROWAREtech for allowing me to include an adapted version of an example implementation of Mersenne Twister random number generator in the project.

I would like to thank Ben Golus for his help with UV texture mapping and his advice on texturing.

Special thanks to my friends Vilém and Matyáš and other members of our Discord server for discussing ideas and for their feedback.

The project was created using Unity Editor, lately in its version 2021.3.1f1. The C# code is kept in a MS Visual Studio Community 2022 project, image materials come from Unity Editor screenshots, Geogebra projects and Blender. Documentation uses LaTeX in its TeX Live implementation.

Disclaimer

Over the past two years, Driftworld evolved both in terms of ideas and terms of implementation. The absolute majority of concepts beyond mathematics were completely new to me when the project started, so the code changed often and many times was almost completely rewritten as I learned. Some older parts remained which can cause a correct impression of inhomogeneity. I do not claim the implementation is flawless and although a lot of the shaky cases were accounted for, some unforseen mistakes may and probably do remain.

I would like to ask anyone using the project to tolerate possible mistakes. There is more work to be done and I would be grateful for feedback.

Thank you for your consideration.

Contents

1 Introduction

Driftworld Tectonics is a Unity project created for use in the Unity editor. The entirety of interactivity is within the editor GUI and the project has no meaningful executable scene. Any feedback is in a console log and the state of the planet is observed within the static scene rendering. This is the most obvious difference from the implementation in the original article from which Driftworld draws inspiration - simulation described in the article offers interactivity while the simulation is running [1].

The workflow follows Cortial et al. in a lot of details, although experience and chosen software tools pose several restrictions. At first a Delaunay triangulation mesh is imported from prepared binary files. Then a set of tectonic plates is created by partitioning said mesh. The planet evolution is performed in repeated tectonic steps. Every step deals with plate subduction, possible continental collision, new crust creation because of diverging ocean plates, slab pull due to subduction influence, erosion and crust damping, and finally, rifting plates. At any time the current state can be saved as a binary file for further use. This follows the original article [1]. Driftworld, however, differs in two rather important steps: continental collisions are always plate-wide and plate rifting follows somewhat different probability mechanics. This is mainly for fine-tuning and can change in future updates, as these changes further simplify an already simplified model and were done as a saving grace from implementation difficulties.

The output of Driftworld is binary planet data with varying resolution of sphere sampling. Provided data are: crust age, elevation, plate assignment, crust thickness and orogeny. The binary file keeps the original topology for easier manipulation. The user can also at any time export the current texture overlay as a PNG image.

This documentation serves both as a user's manual and a quick introduction into the problematic. Section 2 defines basic terms, mathematical objects and their properties. Section 3 follows with details of the used simplified tectonic model. The actual implementation with neccessary details and context are discussed in the section 4. As an important part, performance of the simulation and related issues are the topic of section 5. We conclude the status quo of the project in section 6.

1.1 Motivation

Procedural terrain generation is an important part for a number of computer games [3]. Usually, these games employ random generators to increase variety on a theme, such as a map layout. Indeed, in my subjective opinion a player's experience is greatly enriched by variety, especially in the game environment. This comes with an apparent caveat that purely procedural generation may lack the sense of creativity, leading to mundanely repeating patterns [4].

With the onset of newer technologies (e. g. increased GPU power), we are able to perform more computationally-intensive tasks. When it comes to the terrain generation, even a regular user without access to high-tier hardware can try more sophisticated alternatives to simpler algorithms. Arguably, more realistic worlds bring the feeling of familiarity to the experience. If we can create a more realistic, yet still random map/world/neighbourhood, the possibilities are endless.

Following thoughts are purely my personal view. As a life-long video games fan, I have always gravitated towards story-telling games, especially those taking place in an open world. Among these, I'd like to mention Baldur's Gate series, The Elder Scrolls series, Might & Magic series, Fallout 4 and Mass Effect series. At the same time, I have been also drawn to grand building games taking place in complex worlds. Transport Tycoon or Caesar III and its modern re-implementation Augustus [5] were a heavy influence, lately Factorio or Rimworld. Rimworld stands apart in its uniqueness, as it can be understood rather as a story generator than a game [6]. In large part, the idea of Driftworld came from the works of J. R. R. Tolkien and watching the 1997-2007 Stargate series - the series' take on mythology context in human societies in particular.

Epic stories build on cohesion. In this regard, countless debates take place about details. It takes a great deal of time to create viable environment to match an idea for a story, especially if that story is told over long periods of time. Driftworld aspires to one thing: help create a platform in which stories can take place. First step is this project – to create a rough map.

2 Spherical geometry & topology

The most fundamental object with which Driftworld Tectonics works is a mesh of a sphere in the 3D Euclidean space. For simplicity, we assume the sphere is a unit sphere centered on the origin unless stated otherwise. Because of the spherical nature of the project, several (arguably) uncommon mathematical concepts are described in this section – such as vertex sampling, triangulation, transformations or bounding volume hiearchies. Although the text follows almost a textbook-like mathematical structure, a lot of the formulations and conclusions lack correct proof. Some reasoning is made to carry a point, but meticulous readers are left to their own devices.

2.1 Unity coordinate system

Unity uses a left-handed coordinate system with the x axis pointing to the right, y axis pointing upwards and z axis pointing forward (see Figure 1). This is reflected in the scenes – nevertheless, the mathematical expressions of vectors themselves are identical to a standard right-handed coordinate system, i. e. the following holds for the basis:

$$\mathbf{e}_x \times \mathbf{e}_y = \mathbf{e}_z$$

All implementations must be aware of the fact that the cross product expressions do not distinguish between right-handed and left-handed. It is simply a matter of axes display, where visually 'switching' axes y and z alternates between left-handedness and right-handedness. In the left-handed coordinate system, right-hand rule of cross product shows the inverse final direction of the cross product.

There are several ways to rotate points, vectors or whole transformations. For clarity, let us assume a 3-dimensional vector $\mathbf{u}$ that is to be rotated. We define a rotation unit vector $\mathbf{n}$ and an angle ϕ by which we rotate $\mathbf{u}$ so that $\mathbf{u}$ rotates by ϕ within a plane to which $\mathbf{n}$ is normal. We also assume that the rotation plane passes through the origin. Then from the perspective of a sundial (with $\mathbf{n}$ being the gnomon) $\mathbf{u}$ rotates *clockwise* for positive ϕ (Figure 1). This holds for all relative rotations.

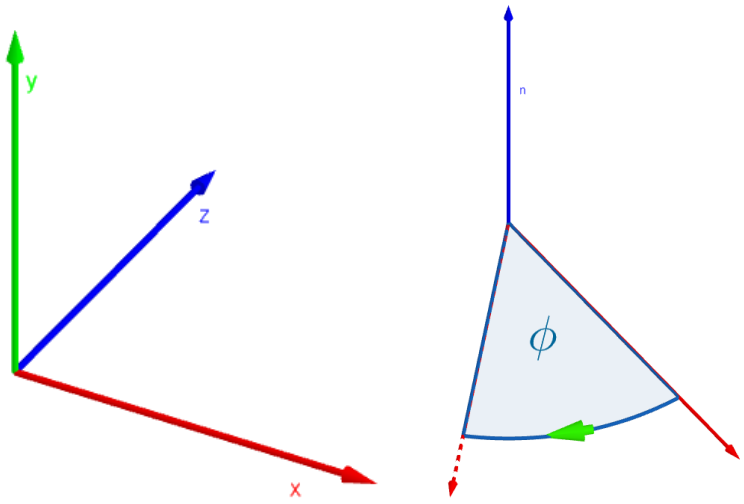


Figure 1: Unity coordinate system

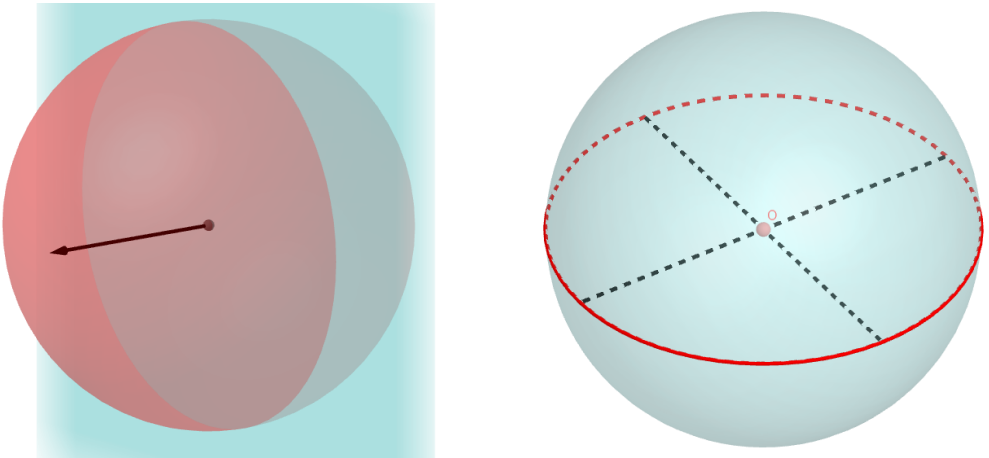
2.2 Sectional planes and great circles

Sphere can have any number of sectional planes, i. e. planes that have some non-empty intersection with the sphere. Planes passing the center of the sphere will be called *sectional central planes* (Figure 2a). Any sectional central plane ρ is characterized by some non-zero normal vector $\mathbf{n}_\rho$ and for any point on the plane represented by their position vector $\mathbf{x}$ it holds that

$$\mathbf{n}_\rho \cdot \mathbf{x} = 0$$

This is synonymous to the fact that any vector lying within a plane passing the origin is perpendicular to the normal vector of the plane. The dot product on the left side of the equality is also important because given a specific normal vector we can decide on which side is any vector $\mathbf{x}$ *outside* the plane – simply take the sign of the dot product, vector on the side of the normal vector will result in a positive dot product value with $\mathbf{n}_\rho$, negative otherwise.

An important object on the surface of a sphere is a great circle. It is any circle that shares its center and radius with the sphere (Figure 2b). It is also the intersection of a plane passing the center of the sphere with its surface.

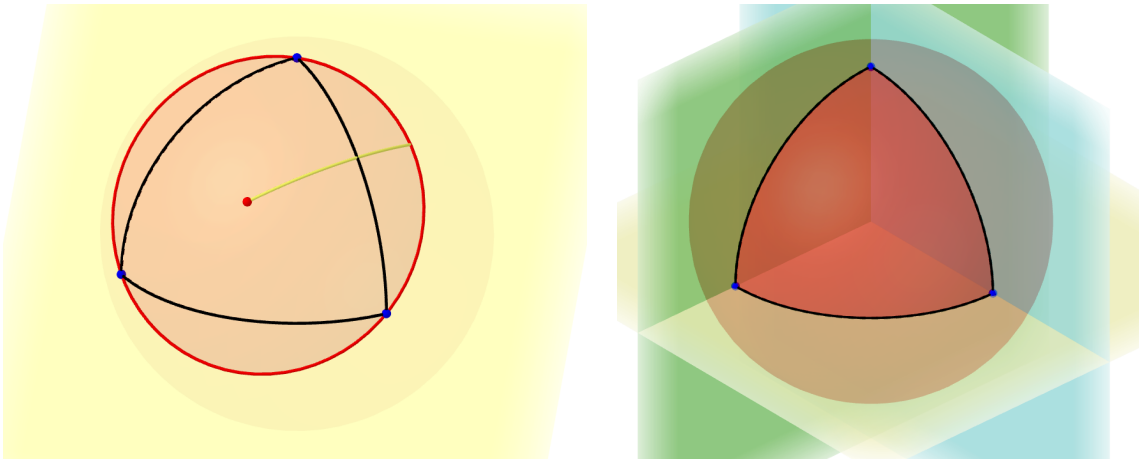


(a) central plane (b) great circle

Figure 2: Sphere section by plane

2.3 Spherical triangles

Any three points on the surface of a sphere that do not lie on a single great circle form a *spherical triangle* (Figure 3). This is the fundamental concept behind many of the computations in the project. However, strictly speaking, there are two triangles defined by such three points. The closure of the complement of any spherical triangle with respect to the sphere surface is also a spherical triangle, albeit one of the two is unintuitive as it is larger than half of the sphere surface area. To get around this, we construct somewhat narrower class of spherical triangles so that any three valid points define a triangle unambiguously.



(a) with sectional plane and circumcircle (b) planes intersections

Figure 3: Spherical triangle

We denote the surface of a unit sphere $\mathcal{S} = \{\mathbf{x} \in \mathbb{R}^3 : ||\mathbf{x}|| = 1\}$. Given a triplet of three linearly independent point vectors $(\mathbf{a}, \mathbf{b}, \mathbf{c}) \in \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ (called *vertices*)[1], we can construct a vector $\mathbf{n}_\lambda$ normal to some sectional plane λ cutting off a spherical cap (Figure 3a):

$$\mathbf{n}_\lambda = (\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})$$

$$\forall \mathbf{x} \in \lambda : \mathbf{n}_\lambda \cdot \mathbf{x} + d_\lambda = 0$$

We can calculate d_λ by assigning e. g. $\mathbf{x} = \mathbf{a}$ and solving the plane equation with respect to d_λ, but that will not be neccessary. We impose a further requirement $\mathbf{n}_\lambda \cdot \mathbf{a} > 0$. This is not always true, in which case it can be ensured by swapping any two vertices in the triplet and recalculating $\mathbf{n}_\lambda$. This means that all three vertices and their circumcenter are all 'on one side' of the sphere. In Unity coordinate system, this also means that for an outside observer, the vertices are oriented *clockwise* on the sphere surface.

Spherical circumcircle l is a set of points on a sphere that has constant spherical distance from a single point $\mathbf{c}_\mathcal{T} \in \mathcal{S}$ called *circumcenter*. Equivalently, we can substitute dot product for distance:

$$\exists t \in \mathbb{R} : (\forall \mathbf{x} \in l : \mathbf{c}_\mathcal{T} \cdot \mathbf{x} = t)$$

Since $\mathbf{n}_\lambda \cdot \mathbf{a} = \mathbf{n}_\lambda \cdot \mathbf{b} = \mathbf{n}_\lambda \cdot \mathbf{c} > 0$, we know that some scalar multiple of $\mathbf{n}_\lambda$ is the circumcenter for the vertices. In fact, there is only one possible circumcircle for all three vertices, which is the intersection $\mathcal{S} \cap \lambda$. We easily find the circumcenter and the circumradius r_l as[2]

$$\mathbf{c}_\mathcal{T} = \frac{\mathbf{n}_\lambda}{||\mathbf{n}_\lambda||}, r_l = \arccos(\mathbf{c}_\mathcal{T} \cdot \mathbf{a})$$

[1] Linear independence of unit vectors is equivalent to the condition that the vectors do not lie on a single great circle.

[2] We have to keep in mind that on a unit sphere, central angle and spherical distance are identical, barring formal dimension.

Previous reasoning allows us now to test if some point $\mathbf{x} \in \mathcal{S}$ is inside a spherical triangle $\mathcal{T} \subset \mathcal{S}$ with clockwise-oriented vertices $(\mathbf{a}, \mathbf{b}, \mathbf{c})$. Geometrically speaking, a spherical triangle is a region bounded by three arcs of great circles [7]. We calculate three normal vectors:

$$\mathbf{n}_\rho = \mathbf{a} \times \mathbf{b},$$

$$\mathbf{n}_\sigma = \mathbf{b} \times \mathbf{c},$$

$$\mathbf{n}_\tau = \mathbf{c} \times \mathbf{a}.$$

These vectors define planes ρ, σ, τ so that

$$\rho = \{\mathbf{x} \in \mathbb{R}^3 : \mathbf{n}_\rho \cdot \mathbf{x} = 0\}$$

$$\sigma = \{\mathbf{x} \in \mathbb{R}^3 : \mathbf{n}_\sigma \cdot \mathbf{x} = 0\}$$

$$\tau = \{\mathbf{x} \in \mathbb{R}^3 : \mathbf{n}_\tau \cdot \mathbf{x} = 0\}$$

intersections $\rho \cap \mathcal{S}, \sigma \cap \mathcal{S}, \tau \cap \mathcal{S}$ are then great circles that always pass two of the vertices. Because of this, each one is divided by them into two arcs. There is only one triplet of arcs connected by the vertices that forms a meaningful region boundary on $\mathcal{S}$ (Figure 3b). There are two such regions but we already bypassed this problem by ensuring orientation. We test the point $\mathbf{x}$ against following condition:

$$\mathcal{T} = \{\mathbf{x} \in \mathcal{S} : \mathbf{n}_\rho \cdot \mathbf{x} \geq 0, \mathbf{n}_\sigma \cdot \mathbf{x} \geq 0, \mathbf{n}_\tau \cdot \mathbf{x} \geq 0\}$$

This simply tells us that $\mathbf{x}$ is inside the spherical triangle $\mathcal{T}$ when it is on the surface of the sphere and also on one specific side of all three planes ρ, σ, τ. This definition does not encompass all possible spherical triangles on a sphere, but it allows us to properly test the properties of any triangles used in reasonable spherical meshes.

2.4 Vertex sampling

Because of memory restrictions, sphere surface data is represented as a set of sampled points. We can identify these points as position vectors $\mathbf{u}_i$ from the global origin to sample points, resulting in a sequence $U = (\mathbf{u}_i)_{i=0}^{N-1}, \mathbf{u}_i \in \mathcal{S}$. Samples are therefore three-dimensional normalized vectors. Driftworld uses spherical Fibonacci sampling [8]. To get the sequence U, another sequence $F = (\mathbf{f}_i)_{i=0}^{N-1}$ is first computed, using the following definition:

$$\mathbf{f}_i = (\phi_i, z_i), \phi_i \in \mathbb{R}, z_i \in \mathbb{R},$$

$$\phi_i = 2\pi \left[\frac{i}{\Phi}\right],$$

$$z_i = 1 - \frac{2i+1}{N}.$$

$[x]$ denotes the fractional part of x, Φ is the golden ratio $\Phi = \frac{\sqrt{5}+1}{2}$. The values of $\mathbf{f}_i$ actually lie on a spiral on the surface of a cylinder with the radius of 1 and the height of 2 [8]. U is finally obtained by mapping $\mathbf{f}_i$ values to $\mathcal{S}$:

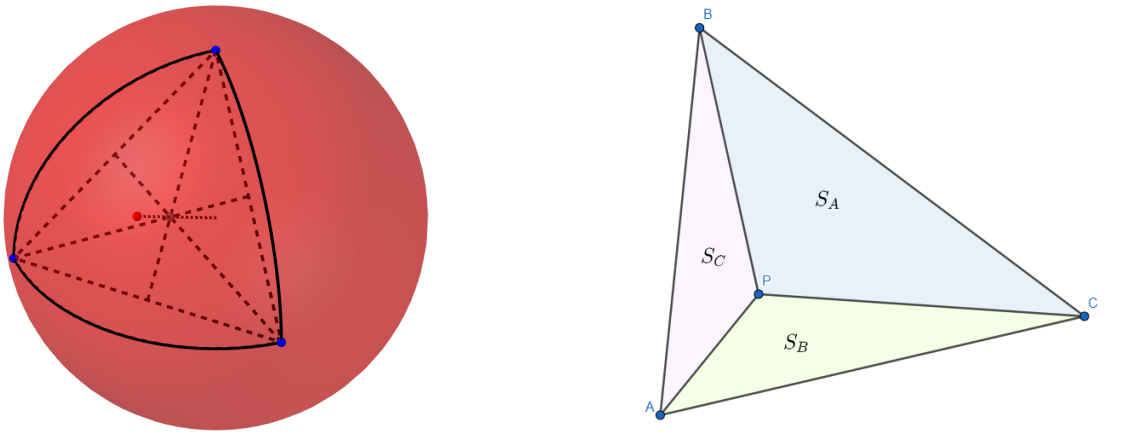
$$\mathbf{u}_i = (\sin(\arccos(z_i)) \cdot \cos\phi_i, z_i, \sin(\arccos(z_i)) \cdot \sin\phi_i)$$

Note that this mapping reflects Unity's axes orientation and the first and the last samples of U do not fall exactly on the poles.

2.5 Centroids, data values and barycentric interpolation

Driftworld often makes computations for points inside sperical triangles - notably, it uses triangle centroids to evaluate triangle neighbours. Calculating these points is not a trivial procedure and although for a long time there have been methods to do so, it would be too resource-consuming when performed on a larger scale. To save computation time, we assume that all evaluated triangles are nearly planar, i. e. their *triangle excess* is negligible (Legendre's Theorem[3] [9]). We calculate the centroid of a triangle by simply normalizing the sum of its vertices (Figure 4a):

$$\mathbf{b}_{\mathcal{T}} = \frac{\mathbf{a}+\mathbf{b}+\mathbf{c}}{||\mathbf{a}+\mathbf{b}+\mathbf{c}||}$$



(a) sectional triangle centroid

(b) barycentric coordinates

Figure 4: Centroid geometry

To store crust data, we must assign values to points on the sphere. These values may be of different types or have different meaning. Formally, we denote a sequence of arbitrary sets $C = (V_i)_{i=0}^{n-1}$, where n is the number of different values assigned to a point and each V_i is a specific set of possible values. The system of all possible value combinations is then a cartesian product of these value sets:

$$V = \prod_{i=0}^{n-1} V_i$$

Stored data can then be defined as a map:

$$s : U \to V$$

$$s_i : U \to V_i$$

We store data only for sphere samples because of limited memory. Other values will be computed as needed using *barycentric interpolation* [10].

[3] This theorem is also known as Saccheri-Legendre theorem.

Now it comes to the following problem: how to compute values anywhere on $\mathcal{S}$? We are effectively looking for some domain extension, since $U \subset \mathcal{S}$:

$$s' : \mathcal{S} \to V, \forall \mathbf{u} \in U : s'(\mathbf{u}) = s(\mathbf{u})$$

$$s'_i : \mathcal{S} \to V_i, \forall \mathbf{u} \in U : s'_i(\mathbf{u}) = s_i(\mathbf{u})$$

Given some arbitrary point $\mathbf{x} \in \mathcal{S}$, we start with an assumption that $\mathbf{x}$ is found inside some spherical triangle $\mathcal{T}$ with negligible triangle excess and vertices $\{\mathbf{a}, \mathbf{b}, \mathbf{c}\} \subset U$ for which we already know the values of s. We would like $s'(\mathbf{x})$ to be computed 'fairly', i. e. the closer $\mathbf{x}$ is to some vertex, the more influence the vertex value should have on $s'(\mathbf{x})$. A good start might be in analogy with a political voting system based on area. If the population is homogeneous, any region vote is weighted by its area and transitionally, by its population.

Let P be some point within a triangle ABC (Figure 4b). This point is represented by a point vector $\mathbf{p} \in \mathcal{S}$. As stated earlier, we assume all points lie nearly on the same plane. If we construct three triangles PBC, APC and APB, the triangle ABC will be divided into three regions, each corresponding to their oposite vertex of ABC. The closer P is to any of the vertices, the larger the corresponding triangle area is. Total area sum of the three triangles is equal to the area of ABC. Therefore, we can use these triangle areas as weights for interpolating values at P – we only need to find the respective areas of S_A, S_B, S_C and S_{ABC}. This can be done using cross product:

$$S_A = \frac{|(\mathbf{b} - \mathbf{p}) \times (\mathbf{c} - \mathbf{p})|}{2}$$

$$S_B = \frac{|(\mathbf{c} - \mathbf{p}) \times (\mathbf{a} - \mathbf{p})|}{2}$$

$$S_C = \frac{|(\mathbf{a} - \mathbf{p}) \times (\mathbf{b} - \mathbf{p})|}{2}$$

$$S_{ABC} = \frac{|(\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})|}{2}$$

Since $S_A + S_B + S_C = S_{ABC}$, we can define normalized weights u, v, w called *barycentric coordinates*:

$$u = \frac{|(\mathbf{b} - \mathbf{p}) \times (\mathbf{c} - \mathbf{p})|}{|(\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})|}$$

$$v = \frac{|(\mathbf{c} - \mathbf{p}) \times (\mathbf{a} - \mathbf{p})|}{|(\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})|}$$

$$w = \frac{|(\mathbf{a} - \mathbf{p}) \times (\mathbf{b} - \mathbf{p})|}{|(\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})|}$$

It is easy to confirm that $u + v + w = 1$.

There are basically two types of values interpolated in the project – real values and categories. Real value interpolation is straightforward:

$$s'_i(\mathbf{p}) = u s_i(\mathbf{a}) + v s_i(\mathbf{b}) + w s_i(\mathbf{c})$$

Categories are simply assigned to $\mathbf{p}$ according to the largest weight:

$$u = \max(\{u, v, w\}) \Rightarrow s'_j(\mathbf{p}) = s_j(\mathbf{a})$$

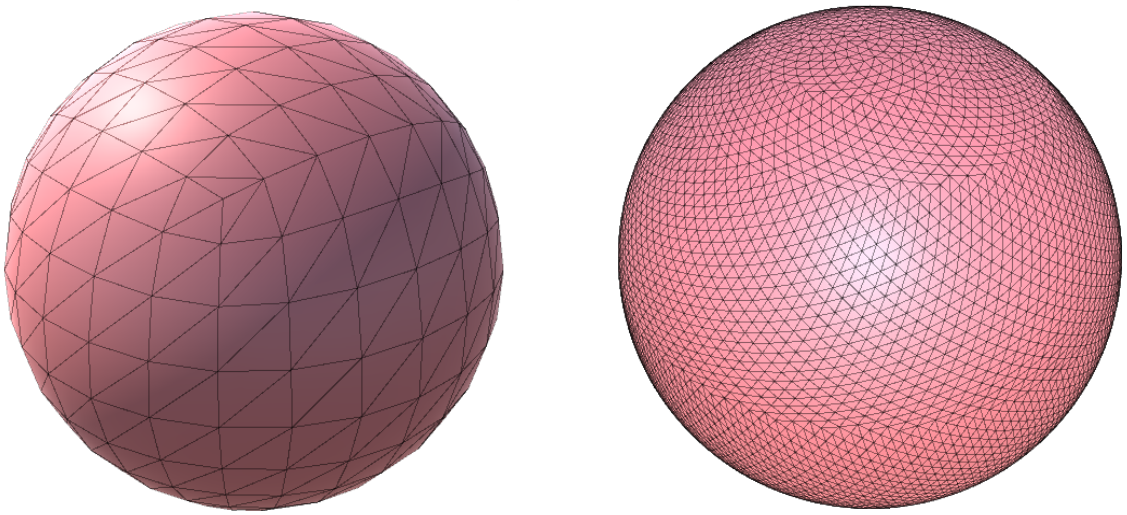
$$v = \max(\{u, v, w\}) \Rightarrow s'_j(\mathbf{p}) = s_j(\mathbf{b})$$

$$w = \max(\{u, v, w\}) \Rightarrow s'_j(\mathbf{p}) = s_j(\mathbf{c})$$

This effectively draws a Voronoi map according to categories [11].

2.6 Spherical mesh and Delaunay triangulation

There is a basic sphere mesh, provided by Unity (Figure 5a). It is like a detailed cubic mesh, projected onto a sphere. However, for finer terrain details, a much more detailed mesh is needed, preferably with uniform triangles (Figure 5b). Scaling the mesh detail is also important.



(a) Unity sphere mesh

(b) Delaunay mesh

Figure 5: Spherical meshes

Definition and description of a 3D mesh is way beyond the scope and purpose of this document. In this context, it is simply an approximation of the sphere surface. All samples U are vertices connected into triangles so that the whole sphere is covered by them without any gaps. For Driftworld, a set of prepared meshes is provided, created by *Delaunay triangulation* [12].

When interpolating surface data such as elevation, it is important that reasonable samples are used for the interpolation. Calculating elevation in a mountain range from a triangle with vertices too far apart may result in meaningless artifacts. Furthermore, the earlier mentioned requirement that the triangles are nearly planar would be undermined by extreme spherical triangles which exhibit considerable excess. It stands to reason that triangles used in the mesh should be as regular as possible. This is the goal and result of a Delaunay triangulation. There is a number of algorithms performing the triangulation on a plane [13]. Since sphere has a closed mesh, an adaptation is needed.

Delaunay meshes for Driftworld use an algorithm which originally triangulates a set of random samples [14]. Because U is ordered, the initial tetrahedron is somewhat difficult to construct, especially because of the requirements imposed on a spherical triangle – in a large number of cases at least one triangle had a circumcircle larger than a great circle. For this reason, the initial structure was set to be a nearly regular octahedron with vertices assigned by a brute-force look-up. Other than that, the algorithm follows the article [14].

2.7 Collisions

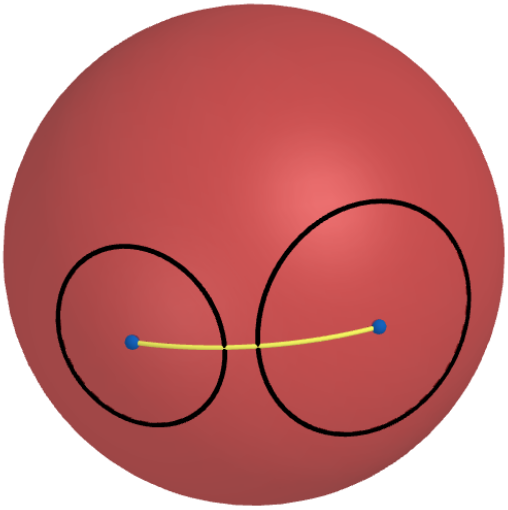
The tectonic model in Driftworld computes many interactions on a regular basis and these computations must be as efficient as possible. There are two basic collisions used for evaluating interactions

– a collision of two circles and a collision of two triangles. To clarify, in both cases we only need to answer the question whether the two objects have a non-empty intersection – not to fully classifiy the intersection. Algorithms for both collisions are fairly simple and the spherical geometry actually helps in the case of triangle collisions.

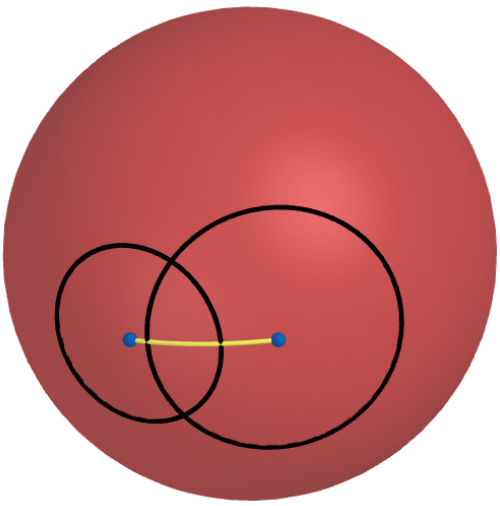
The relative position of two circles $k, l \subset \mathcal{S}$ is governed by several parameters. Each circle has a circumcenter $\mathbf{c} \in \mathcal{S}$ and a radius $r > 0$. We consider full circles to determine the intersection:

$$\forall \mathbf{x} \in \mathcal{S} : \arccos(\mathbf{x} \cdot \mathbf{c}_k) \leq r_k \Rightarrow \mathbf{x} \in k$$

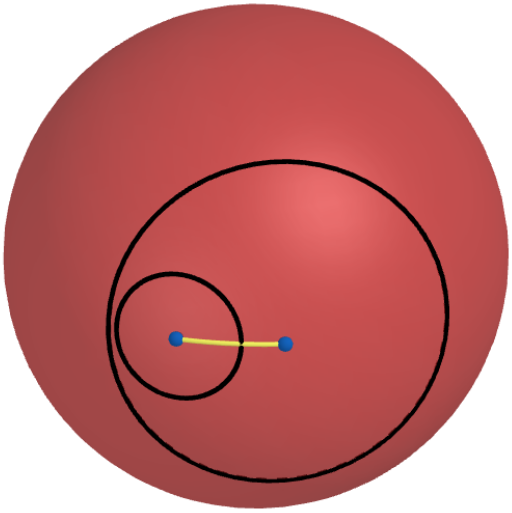
This means that there is only one case of relative circle position that has an empty intersection: disjoint circles. The case of one circle lying inside another has an intersection indentical to the inside triangle. Three major cases of the relative positions of circles are seen in Figure 6.



(a) disjoint circles



(b) circles intersecting at two points



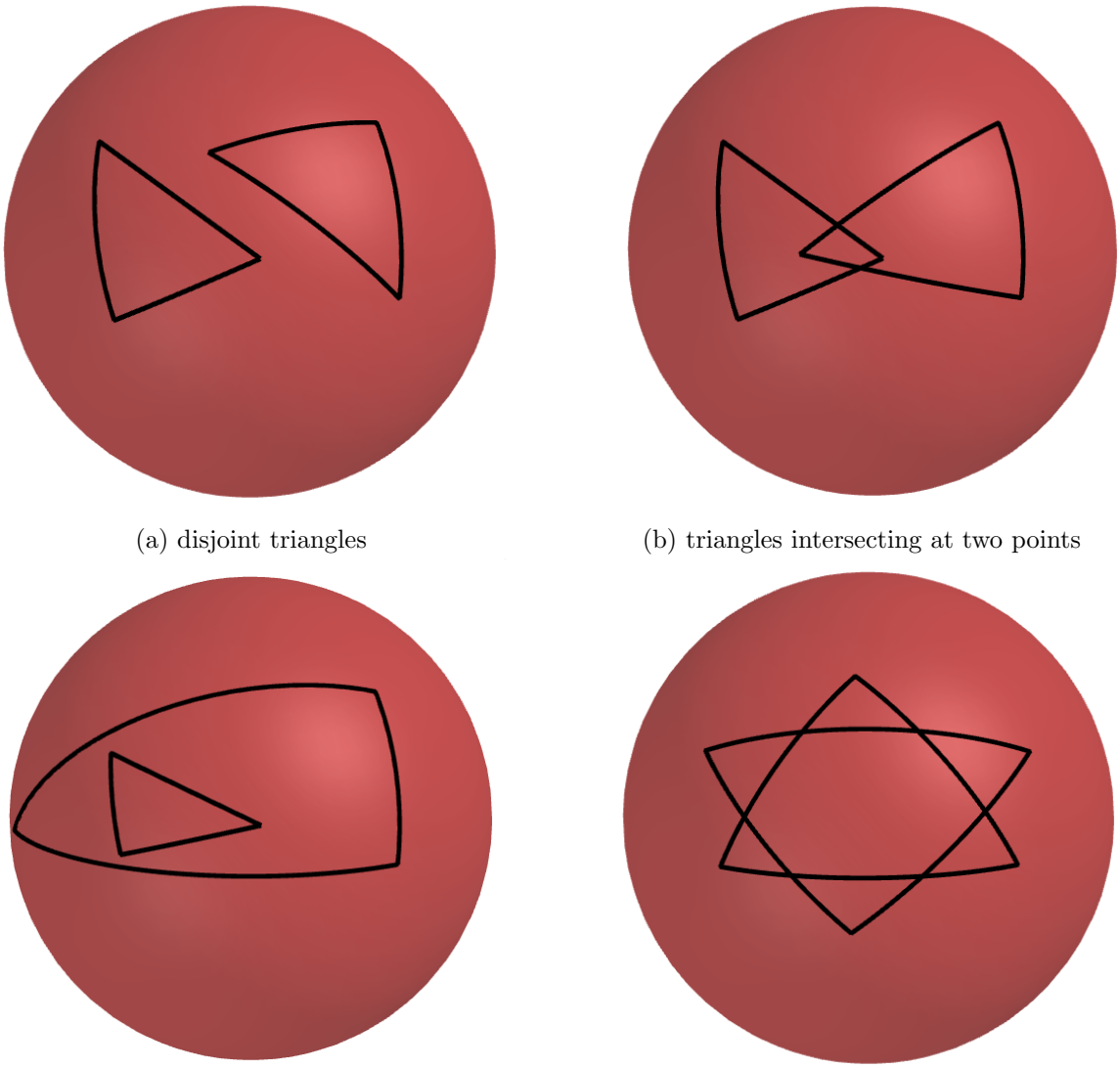
(c) circle lying inside another

Figure 6: Relative positions of two circles

It is therefore easy to decide whether two circles collide or not - circles not colliding have a spherical distance larger than the sum of their radii:

$$k \cap l = \emptyset \iff \arccos(\mathbf{c}_k \cdot \mathbf{c}_l) > r_k + r_l$$

In case of triangles the collision is more complex. There are four major cases of the relative position of two spherical triangles - three similar to the case of circles and one specific (see Figure 7). The second and the third case can be resolved by determining whether any point of one triangle lies within the other (see subsection 2.3). The fourth requires a more thorough test.



(a) disjoint triangles

(b) triangles intersecting at two points

(c) triangle lying inside another

(d) Star of David

Figure 7: Relative positions of two triangles

If neither of the two triangles contain a vertex of the other one, we have to decide if any two edges intersect. Consider two edges defined by pairs of non-identical vertices $(\mathbf{a}_1, \mathbf{a}_2)$ and $(\mathbf{b}_1, \mathbf{b}_2)$. These define planes within which lie their respective great circles. The planes have normal vectors $\mathbf{a}_1 \times \mathbf{a}_2$ and $\mathbf{b}_1 \times \mathbf{b}_2$. Since the planes are central, Any intersection must be along the vector $(\mathbf{a}_1 \times \mathbf{a}_2) \times (\mathbf{b}_1 \times \mathbf{b}_2)$. There are two intersections in $\mathcal{S}$ and we consider the one maximizing the dot product with $\mathbf{a}_1$ (same hemisphere). We denote such intersection $\mathbf{i}$. The final test simply decides if $\mathbf{i}$ lies on both segments or is somewhere else on the great circles. This can be done by testing dot products, as i must be closer to both vertices than the distance of the vertices for both segments:

$$(\mathbf{i} \cdot \mathbf{a}_1 > \mathbf{a}_1 \cdot \mathbf{a}_2) \wedge (\mathbf{i} \cdot \mathbf{a}_2 > \mathbf{a}_1 \cdot \mathbf{a}_2) \wedge (\mathbf{i} \cdot \mathbf{b}_1 > \mathbf{b}_1 \cdot \mathbf{b}_2) \wedge (\mathbf{i} \cdot \mathbf{b}_2 > \mathbf{b}_1 \cdot \mathbf{b}_2)$$

If any two segments of the two triangles intersect, the triangles must by the sign analysis have non-empty intersection and therefore collide. If all tests are negative, the triangles are disjoint.

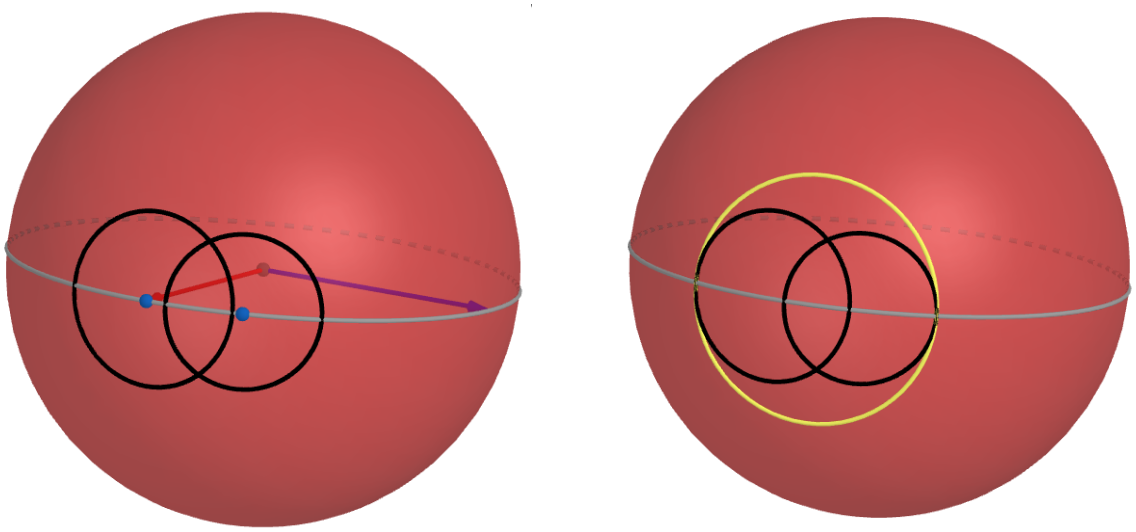
2.8 Merging of spherical circles

Because of the need to build bounding volume hiearchies (see subsection 4.3), there is a task of finding a suitable circle l which has the smallest area possible and which contains two other given circles m, n (see Figure 8b). The center of l must lie on a great circle L containing both centers of the circles. This means we have to find a center and radius of l so that it only touches either both of the two circles m, n or one in case one circle is within the other. The algorithm falls in at least one of the following cases: concentric circles, circles with opposite centers, one circle contained in the other, circles intersecting at two points or disjoint circles. The goal is to determine which case has the deciding influence on the position of the center $\mathbf{c}_l$ and the radius r_l except in the case of concentric circles, where l is found easily. In case of concentric circles the solution is:

$$\mathbf{c}_l = \mathbf{c}_m, r_l = \max(r_m, r_n)$$

Otherwise we find a suitable local basis in which we can easily parametrize all relevant points (Figure 8a). One circle center will be identical to the base vector:

$$\mathbf{e}'_x = \mathbf{c}_m$$



(a) local basis (b) circles merged

Figure 8: Merging of two circles

Computation of the second base vector $\mathbf{e}'_z$ depends on whether the circles are directly opposite, i. e. they have centers opposite on the sphere. If so, it is any vector perpendicular to $\mathbf{e}'_x$ since the centers lie on inifinitely many great circles. If not, it it can be found with a double cross product:

$$\mathbf{e}'_z = \frac{(\mathbf{c}_m \times \mathbf{c}_n) \times \mathbf{c}_m}{||(\mathbf{c}_m \times \mathbf{c}_n) \times \mathbf{c}_m||}$$

There are three variables we need to compute now. First is the distance d_{mn} between $\mathbf{c}_m$ and $\mathbf{c}_n$:

$$d_{mn} = \arccos(\mathbf{c}_m \cdot \mathbf{c}_n)$$

Second is the central angle $\Delta\phi$ of an arc running between $\mathbf{c}_m$ and the center of the merged circle. This allows us to compute the center as a linear combination of the base vectors $(\mathbf{e}'_x, \mathbf{e}'_z)$. Third is the

actual radius r_l. However, we first need to see if one of the circles is contained within the other. If so, then one boundary is pushed by the encompassing circle:

$$-r_m > d_{mn} - r_n \implies \Delta\phi = d_{mn}, r_l = r_n$$
$$r_m > d_{mn} + r_n \implies \Delta\phi = 0, r_l = r_m$$

The first condition is for n encompassing m, the second is the other way around. If neither is true, the computation is slightly more difficult:

$$\Delta\phi = \frac{d_{mn} - r_m + r_n}{2}$$
$$r_l = \frac{r_m + r_n + d_{mn}}{2}$$

Finally, we compute the center of l[4]:

$$\mathbf{c}_l = \cos(\Delta\phi)\mathbf{e}'_x + \sin(\Delta\phi)\mathbf{e}'_z$$

2.9 Texture mapping

The system of overlays requires creating a texture every time the planet is to be rendered. Suppose we have a texture with resolution $w \times h$ that should have a color b_{ij} from some color space $\mathcal{C}$ assigned to each of its points:

$$i \in \mathcal{I} = \{0, 1, 2, ..., w-1\}$$
$$j \in \mathcal{J} = \{0, 1, 2, ..., h-1\}$$
$$b : \mathcal{I} \times \mathcal{J} \to \mathcal{C}$$

We want b_{ij} to reflect the data on the sphere. Let b' be a map from the sphere surface, representing an overlay:

$$b' : \mathcal{S} \to \mathcal{C}$$

We now have to find some map between the pixel indices and the sphere surface. This is very similar to the classical problem in cartography. Our choice will be an equirectangular projection because of its simplicity. We first compute the azimuthal and polar angles ϕ_i, θ_j from i, j:

$$\phi_i = 2\pi\frac{\left(i + \frac{1}{2}\right)}{w}$$
$$\theta_j = \pi(1 - \frac{j + \frac{1}{2}}{h})$$

This means the texture y coordinate increases *upwards* (to north) in the respective cylinder rectangle (Figure 9). From the angles it is simple to find the point on the sphere:

$$\mathbf{x}_{ij} = (\sin\theta_j \cos\phi_i, \cos\theta_j, \sin\theta_j \sin\phi_i)$$

Finally, we can compute b_{ij}:

$$b_{ij} = b'(\sin\theta_j \cos\phi_i, \cos\theta_j, \sin\theta_j \sin\phi_i)$$

This texture is subsequently used for rendering the surface, however, there is an inherent problem with uv mapping. Because of my insecurity about the math involved, the process will not be explained in this text and instead, the reader is encouraged to read a more educated article [16].

There is perhaps a better way to deal with texture mapping, which was discussed in a Unity forum thread [17]. Cubemaps are quite possibly a more logical choice, as the texture information density needlesly increases towards extreme values of the y coordinate.

[4]Note that the value of $\Delta\phi$ can be negative. This is the case of clockwise-oriented arc from $\mathbf{c}_m$ in the local basis.

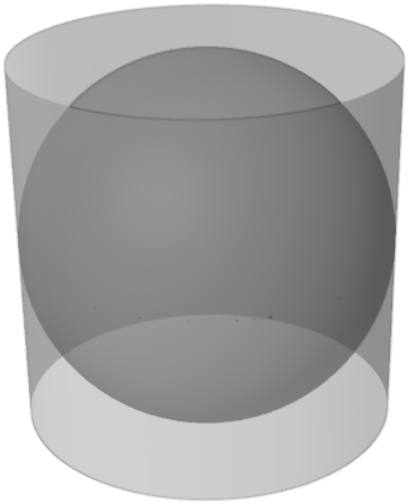


Figure 9: Cylinder rectangle around a sphere

Unit	Conversion
1 km	0.001 u
$1\ \text{km}^{-1}$	$1000\ \text{u}^{-1}$
$1\ \text{mm}\cdot\text{y}^{-1}$	$10^{-3}\ \text{u}\cdot\text{My}^{-1}$

Table 1: Unit conversion table

2.10 Spherical units

Virtually all expressions throughout this section assumed a unit sphere for simplicity, but for rendering and parameter context some scaling is needed. For example, radius R of the planet might be given in kilometers [1]. Since rendering in thousands of 'Unity meters' is impractical, let us denote this unit u and consider:

$$1\ \text{u} = 1000\ \text{km}, \forall \mathbf{x} \in \mathcal{S} : ||\mathbf{x}|| = 1\ \text{u}$$

Then the all the rendered vertices should only be transformed as $\mathbf{x} \to R\mathbf{x}$. Our unit of time will be My. Some useful conversions follow in table 1.

Some given parameters can be set for a sphere with a given radius R. In this case, conversion of the value to unit sphere must take the radius into consideration – quantities of distance must be divided by R, quantities of area by R^2.

Example Maximum plate speed v_0 is $100\ \text{mm}\cdot\text{y}^{-1}$ on a planet with radius $R = 6370$ km. Its unit sphere value is[5]:

$$v_0' = \frac{100\ \text{mm}\cdot\text{y}^{-1}}{6370\ \text{km}} = \frac{0.1\ \text{u}\cdot\text{My}^{-1}}{6.370\ \text{u}} \approx 0.0157\ \text{My}^{-1}$$

This allows us to identify any surface speed with an angular speed.

[5]Because Driftworld Tectonics uses the unit sphere values, we will reference quantities declared on a unit sphere as primed to distinguish from the quantities of the original article [1].

3 Simple tectonic model

4 Implementation & data model

4.1 Rendering

4.2 GPU Computing

4.3 Bounding volume hiearchy

When dealing with interactions of objects consisting of many triangles, such as parts of a mesh or meshes, testing every triangle against each other is very inefficient, leading to algorithmic complexity of $\mathcal{O}(n^2)$. For this, Driftworld implements *bounding volume hiearchy* (BVH) over the triangles of the sphere mesh [15]. Bounding volume (BV) is a simple object that contains a primitive or a group of primitives. Its purpose is to simplify collision tests – for this, bounding volumes are structured in a hiearchy. The most common BVH is a tree – its leaves are bounding volumes of individual primitives. Parent nodes of such a tree are bounding volumes created by convenient merging of their children node BVs. The root of the tree is then some whole object. For instance, it could be a bounding volume of a mesh, while the leaves would be the singular triangles or their bounding volumes. There are multiple types of various bounding volumes and multiple types of constructed BV trees.

For our purpose, we construct a BVH for the triangles inside some region of the sphere triangulation. Our bounding volume is a spherical circle, the tree is a binary tree of merged circles. This choice makes sense because the triangles only move on the surface of the sphere and therefore we do not need spatial bounding volumes. Mathematically

4.4 Use

5 Performance & problems

6 Conclusion

6.1 Continuation of work

7 References

[1] Cortial, Y., Peytavie, A., Galin, E. and Guérin, E. (2019), Procedural Tectonic Planets. *Computer Graphics Forum*, 38: 1-11. https://doi.org/10.1111/cgf.13614

[2] D. Meister and J. Bittner, "Parallel Locally-Ordered Clustering for Bounding Volume Hierarchy Construction," in *IEEE Transactions on Visualization and Computer Graphics*, vol. 24, no. 3, pp. 1345-1353, 1 March 2018, doi: 10.1109/TVCG.2017.2669983.

[3] Wikipedia contributors. (2022, March 20). List of games using procedural generation. In *Wikipedia, The Free Encyclopedia*. Retrieved 06:20, May 19, 2022, from https://en.wikipedia.org/w/index.php?title=List_of_games_using_procedural_generation&oldid=1078237416

[4] Brogan, J. (2016, October 5). *The Daggerfall Paradox*. SLATE. Retrieved May 19, 2022, from https://slate.com/technology/2016/10/the-paradox-of-procedurally-generated-video-games.html

[5] Keriew, *Augustus*, (2021), GitHub repository, https://github.com/Keriew/augustus

[6] Game Developer Conference [GDC]. (2019). *RimWorld: Contrarian, Ridiculous, and Impossible Game Design Methods* [Video]. YouTube. https://www.youtube.com/watch?v=VdqhHKjepiE

[7] Palmer, C. I., & Leigh, C. W. (1934). *Plane and spherical trigonometry*. New York: McGraw-Hill Book Company, Inc.

[8] Keinert, Benjamin & Innmann, Matthias & Sänger, Michael & Stamminger, Marc. (2015). Spherical Fibonacci Mapping. *ACM Transactions on Graphics*. 34. 1-7. 10.1145/2816795.2818131.

[9] Todhunter, Isaac. (1886). *Spherical Trigonometry: For the Use of Colleges and Schools* (5th ed.). London: Macmillan and Co.

[10] *Ray Tracing: Rendering a triangle*. Scratchapixel 2.0. Retrieved June 6, 2022, from https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/barycentric-coordinates

[11] Pokojski, Wojciech & Pokojska, Paulina. (2018). Voronoi diagrams – inventor, method, applications. *Polish Cartographical Review*. 50. 141-150. 10.2478/pcr-2018-0009.

[12] Delaunay, Boris. (1934). Sur la sphere vide. A la memoire de Georges Voronoi. *Bulletin de l'Academie des Sciences de l'URSS. Classe des sciences mathematiques et naturelles*, Issue 6, pp. 793–800.

[13] Guibas, L. J., Knuth, D. E., & Sharir, M. (1992). Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7(1-6), 381–413. doi:10.1007/bf01758770

[14] Ma, Yingdong & Chen, Qian. (2010). Fast Delaunay Triangulation and Voronoi Diagram Generation on the Sphere. *2010 Second World Congress on Software Engineering*, pp. 187-190, doi: 10.1109/WCSE.2010.136

[15] Sulaiman, Hamzah & Bade, Abdullah. (2012). *Bounding Volume Hierarchies for Collision Detection*. 10.5772/35555.

[16] Golus, Ben. (2021). *Distinctive Derivative Differences*. Retrieved June 15, 2022, from https://bgolus.medium.com/distinctive-derivative-differences-cce38d36797b#85c9

[17] (2020). *Texture repeats itself within single border triangles*. Unity forum thread. Retrieved June 15, 2022, from https://forum.unity.com/threads/texture-repeats-itself-within-single-border-triangles.1029907/

List of Figures

List of Tables

A Project parameters