

Tectonic Plate Simulation on Procedural Terrain

Daniel Benedetti
Rensselaer Polytechnic Institute

Evan Minto
Rensselaer Polytechnic Institute

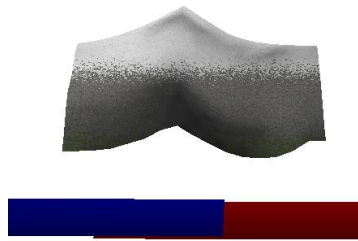


Figure 1: A basic ridge formed by two plates colliding

Abstract

This paper seeks to model the movements of tectonic plates and simulate the effects of tectonic collisions on terrain. Simple 2-D plate collisions generate a Gaussian function for displacement, and programmable shaders enable vertex-texture mapping and multitexturing effects to show a detailed terrain visualization.

1 Motivation

Under the Earth's surface lies a network of large tectonic plates. The boundaries between the plates are known as faults. These plates are constantly moving, colliding with or separating from their neighboring plates. The effects of this movement are called as tectonic activity, with the greatest effects occurring closest to the fault lines. Some major geological events, such as earthquakes, are caused by large, sudden plate movements within plates. Minor movement in the plates will not likely cause immediate change, but sustained, minor tectonic activity is responsible for shaping the face of the Earth, such as moving continents and creating mountains.

It is currently possible to approximate the effects that various forms of tectonic activity will have on a terrain surface over time. Namely, one would could construct a relationship between vertical displacement and time for a given location. Numbers alone, however, do not provide insight into the geological changes that occur as a result of this displacement. The purpose of this project is to create a simple simulation that models the effects of long-term tectonic activity near a fault line on a terrain surface.

2 Related Work

A major inspiration for this paper was Lauri Viitanen's work on tectonic plate-based terrain generation in *Physically Based Terrain Generation: Procedural Heightmap Generation Using Plate Tectonics*. Viitanen's simulation is far more complex and more difficult to replicate than this one, but using 2-D plates and measuring the displacement based on overlap regions is an extremely intuitive way to model plate collisions.

Throughout the course of the year, Daniel has been doing research under the guidance of W. Randolph Franklin. Franklin's research interests include the

visualization of geological data. As a result, Daniel has done work with visualization of terrain surface data. The shader visualization in the fourth Advanced Computer Graphics homework assignment served as an inspiration for several features utilized in the visualization, particularly in the fragment shader. Additionally, a general concept for the steps involved in creating the terrain surface came from a presentation by Nowrouzezahrai [Nowrouzezahrai]. The paper on Geometry Clipmaps [Losasso et al. 2004] provided great insight on how to efficiently handle scale on large terrain surfaces. Lastly, the Sky Illumination Model [Kennelly et al. 2006] paper provided a realistic alternative to the standard fixed-position light source used in traditional terrain renders.

3 Simulation

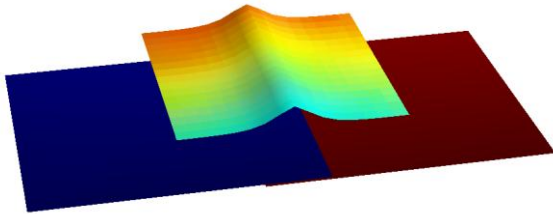


Figure 2: *Displacement applied on the base mesh, without applying shaders. Plates visualized in blue and red.*

The plate simulation is based on a modified version of the simple model proposed by Viitanen [2012]. The program first creates two plates “underneath” the mesh, represented as 2-D rectangles along the XZ plane (these can be specified using only two vertices, the top-left and bottom-right, as the rest can be inferred). These plates are given an initial velocity along the X axis so that they will eventually collide or diverge.

When the plates do collide, they create a displacement on the mesh above them. This is achieved by creating an “overlap plate,” an object

that acts similarly to a plate but actually represents the area of overlap between the two plates. At each timestep, the vertices in the mesh query the simulation for the magnitude of their displacement, and the simulation returns a value based on the size and position of this overlap region.

3.1 Overlap Regions

This overlap region is an essential part of the simulation. Luckily, creating an overlap region is extremely straightforward. Depending on how the plates are oriented, however, the calculation needs to be done slightly differently. If Plate 1 is to the left of Plate 2, the rightmost “vertices” (remember only one actually exists) of Plate 1 will be combined with the leftmost “vertices” of Plate 2. If they are reversed, then Plate 1 will use its leftmost ones, and Plate 2 its rightmost.

To calculate displacement magnitude, the simulation creates a smooth gaussian function across the XY plane, using the velocities of the plates, the area of the overlap, and the midpoint of the overlap to create the curve of the function. More specifically, the function is defined as:

$$Y = M \cdot A \cdot (|V_1| + |V_2|) \cdot \exp\left(\frac{-(P_x - C_x)^2}{\frac{1}{2}W^2}\right) \quad (1)$$

Where Y is the displacement in the Y direction, M is a modifier to modulate the size of the displacement, A is the area of the overlap region, V_1 and V_2 are the velocities of the plates, P_x is the X position of the point being displaced, C_x is the X position of the center of the overlap region, and W is the width of the overlap region.

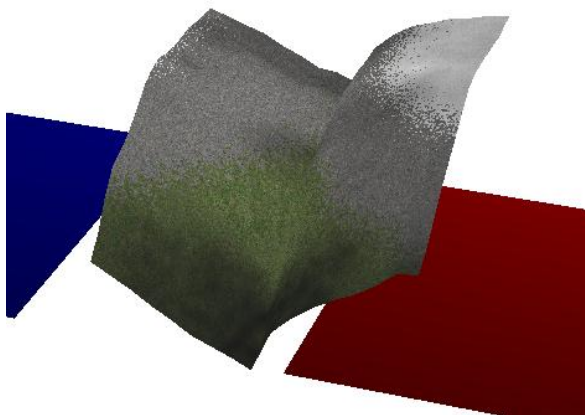


Figure 3: *Two plates move away from each other at high speed, creating a sharp gorge.*

3.2 Negative Overlaps

In a plate tectonics simulation it is important not only to model plate collisions, but also plate divergence. To represent this separation of plates, the simulation uses “negative overlaps.” These are simply overlap regions that, rather than representing the area the two plates share, represents the area between them. This luckily requires very little modification of the collision-based overlap code, and in fact only requires the addition of a boolean called “empty” that represents whether the plate is empty (a negative overlap) or full (a positive overlap). When the overlap is empty, the gaussian function for displacement is flipped, thus creating negative displacement in the areas where the plates diverge.

3.3 Running the Simulation

At each timestep, the simulation moves the two plates based on their velocities and applies forces. Namely, a friction force is added to slow down plate movement and allow the simulation to converge. Whenever a plate moves (whether or not it is colliding with another plate), it receives a friction force proportional to its velocity and the area of overlap (even if it’s negative). This force makes the

plates “grind to a halt” over the course of the simulation.

Once the plates are moved and forces are added, the overlap region is recomputed and the update ends. From here the rendering side of the application makes queries about displacement values and feeds them into the shaders.

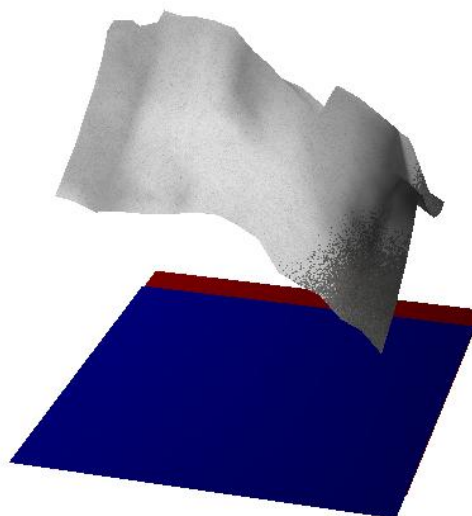


Figure 4: *A series of fault points in these plates create a jagged edge in the resulting mesh.*

3.4 Fault Points

An extension to these basic techniques was also added, allowing for simple modeling of uneven plate boundaries. Instead of only looking at two plates flush against each other, this model selects specific points on the plate edge and shifts them along the X axis to create perturbations in the boundary. These points are called “fault points,” and the simulation using them is called the “Fault Points Method.”

Fault points somewhat complicate the representation of plates and computation of overlap regions. Instead of only using two points for the plate rectangle (as mentioned previously), this

method adds fault points along the plate edge *in addition* to the bounding box points. The fault points include one on the top and bottom of the plate (maximum and minimum Z values) as well as points in between. Since overlap regions are bound on both sides by fault points — unlike plates, which have three straight sides and one jagged side — they must be represented differently. With the Fault Points Method, each overlap region is specified by a series of fault points representing the midpoint of the overlap and a width value. The area of this region is actually equal to the area of an equivalent straight edge, since a parallelogram and a rectangle with the same width and height have the same area.

Finding the displacement with the Fault Points Method involves getting a different midpoint for the overlap depending on the position of the point sending the query. The program finds the nearest fault point in the overlap (nearest in terms of Z only), then picks out the next closest point. The appropriate point along the overlap's jagged central line can be determined by determining the intersection between two lines: the line between the two closest points and the line representing all points with the same Z value as the point sending the query. With this intersected point (the equivalent point along the central line of the overlap), the gaussian function's midpoint can be shifted along the X axis to create a curved displacement shape across the overall mesh.

4 Visualization

The second portion of the terrain simulator is graphical visualization. The purpose of this aspect of the project is to display the effects that the simulated tectonic activity would have on a supplied terrain surface. This visualization utilizes several graphical techniques in order to emphasize the displacement applied to the terrain by the simulation. These techniques include multitexturing, vertex-texture mapping, and semi-procedural textures.

4.1 Framework and Mesh

The basic requirements for this visualization were the ability to render planar meshes, manipulate vertex locations for these meshes, and animate these vertex displacements. For this reason, the fourth homework assignment was used as the foundation for this project. Mesh loading and animation were already present within the foundation code. Additions were made to support vertex displacement, and several unused aspects, such as shadow volumes and the stencil buffer, were removed. Additionally, the vertex buffer objects (VBOs) for the primary mesh were modified to support the storing of texture values.

The mesh is a simple flat plane, but care was taken in the construction of these meshes. This became more of a challenge than had been anticipated. After attempting to write an .obj generator and then trying various tools, it was determined that the Blender software package was capable of creating the required mesh. Blender was used to create a plane, which was then triangulated. The plane was then subdivided to create meshes of higher density. The primary target mesh is called *flat256.obj*, while a smaller test mesh is called *flat16.obj*. Their attributes are as follows:

<i>flat16.obj</i>	Vertices: $17 \times 17 = 289$
	Triangles: $16 \times 16 \times 2 = 512$
<i>flat256.obj</i>	Vertices: $257 \times 257 = 66049$
	Triangles: $256 \times 256 \times 2 = 131072$

The system would support larger mesh sizes. The displacement computation, however, is not parallelized, and larger meshes would not be capable of producing real-time animations.

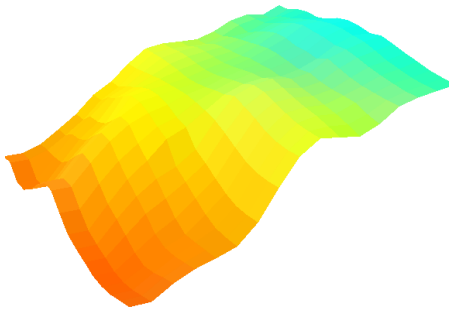


Figure 5: Height map applied as both texture and vertex-texture displacement.

4.2 Textures

Textures are utilized extensively in the visualization. To be precise, five textures are loaded into the visualization: a height map, a normal map, a stone bitmap, a snow bitmap, and a grass bitmap. After finding a height map, the corresponding normal map was created using the CrazyBump software package. Attempts were originally made to generate the normal within the shader, but a proper normal calculation would have required access to adjacent vertices. The size of the height map and normal map are not necessarily governed or constrained by the size of the mesh. It did, however, seem logical to select these meshes such that each pixel corresponds to one square face (two triangle faces) of the mesh to create a one-to-one texture mapping. Large 2048×2048 images were used for the stone, snow, and grass textures in order to minimize pixelation artifacts.

OpenGL by default does not provide the capability of loading textures. For this reason, several texture loading software packages were considered. Difficulties in including the package with *cmake* led to searching for simple source for loading a texture instead. The texture loader that was found works specifically on 24-bit BMP format images, and is inspired by the texture loader described by Guha [2010].

The most time-consuming portion of the visualization was properly placing the textures into GLSL. As the process required several steps, it was default to determine the portion responsible for causing problems. The texture coordinates first needed to be specified in the creation of the VBO. After applying modulation and filtering settings, the texture was then bound in OpenGL. Each texture was then mapped to a corresponding *sampler2d* object in the shaders. The following code snippet shows the binding of the height map texture, stored in "texture[0]", to the *sampler2d* object "terrainMap" in the vertex shader.

```
// Height Map
glActiveTexture(GL_TEXTURE0);
GLint mapLoc =
    glGetUniformLocationARB(GLCanvas::program,
        "terrainMap");
glBindTexture(GL_TEXTURE_2D, texture[0]);
glUniform1iARB(mapLoc, 0);
```

4.3 Vertex Shader

The vertex shader is responsible for vertex-texture mapping, which differs from some of the traditional forms of texture mapping. Unlike bump mapping, this causes an actual change to the geometry and produces realistic silhouettes. Unlike displacement mapping, this effect does not provide features such as self-occlusion and self-shadows. The per-vertex displacement is calculated based on the color attributes for the associated height map pixel. This displacement is then added directly to y-component of the vertex position.

The newly added normals from the vertex-texture mapping, loaded into the vertex shader as a normal map, needed to be combined with the normals of the actual vertex positions, which change during the course of the simulation. Additionally, the vertex normals utilize Gouraud shading, and as such, cannot be computed within the vertex shader. It was then quite difficult to compute the resulting normal from the combined vertex normal and normal map. As a workaround, the two normals are simply averaged, which produce satisfactory results.

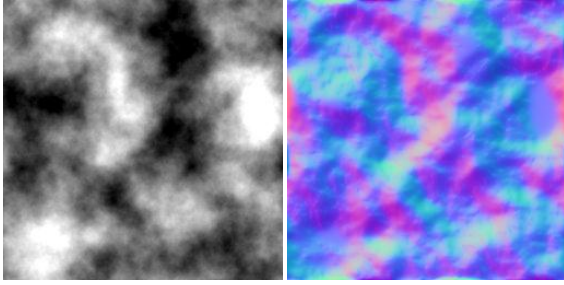


Figure 6: A height map, with corresponding normal map created using CrazyBump.

4.4 Fragment Shader

The fragment shader is responsible for selecting pixel color from a texture lookup, on a per-pixel basis. Simple shading is used, with a single stationary light source. The ambient component is kept to a minimal, and the majority of the lighting seen is diffuse. Basic specular lighting is implemented, but the effect was kept to a minimal. Multitexturing is the result of combining multiple textures on a single surface. In this case, the stone, snow, and grass textures are each sampled. The worldspace y-position is then utilized in order to determine which texture of the three to utilize, with snow selected for high height values, stone selected for middle height values, and grass selected for low height values. These height values are dynamic, and as such, the texture is calculated procedurally and updates with the simulation.

Perlin noise is utilized in order to create a blended transition area at the height boundaries between two textures. As seen in Figure 7, the gradual transition is more natural and less startling than the hard boundary line. Perlin noise is additionally used to adjust the normal for variations in the specular lighting.

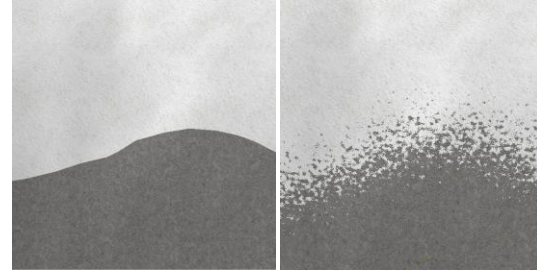


Figure 7: View of the snow-stone height boundary. Left without Perlin noise, right with Perlin noise

4.5 Views

The visualization offers several viewing options. Upon startup of the animation, the rendered scene shows the height map applied as a texture to the flat plane. Additionally, it shows the locations of the tectonic plates situated under the terrain surface, which can be toggled on or off. The shaders can then be enabled, which will displace the terrain and apply the procedural multitexture. The animation can be run while in either of these views.

5 Results

The program was tested on a Intel Core 2 Duo CPU at 2.66 GHz and an NVIDIA GeForce 9600M GT 256 MB GPU, which has 32 cores at 120 MHz. On the 16x16 mesh the program ran in real-time, but on the 256x256 mesh the program slowed to approximately 3 frames per second.

Included in this paper are a number of images from the program, showing a variety of configurations of plates, velocities, fault points, and shader inputs to show the range of plate dynamics and terrain renderings the program can create.

The Gaussian function in the simulation tends to create a smooth curve that changes fairly significantly depending on the position and size of the overlap region, as seen in Figure 1, and the addition of Perlin noise adds some crucial variation to the displacement values.

Fault points create sharp, interesting discontinuities, as seen in Figure 4, but the variation is more than expected, and could probably use some tweaking to create smoother displacement values.

The visualization, when loaded and rendered prior to the simulation, will produce a result such as that found in Figure 8. As the simulation runs, the procedural texture will dynamically update to select which texture to display for each pixel on the terrain surface. A close-up view of the terrain can be seen in Figure 9.

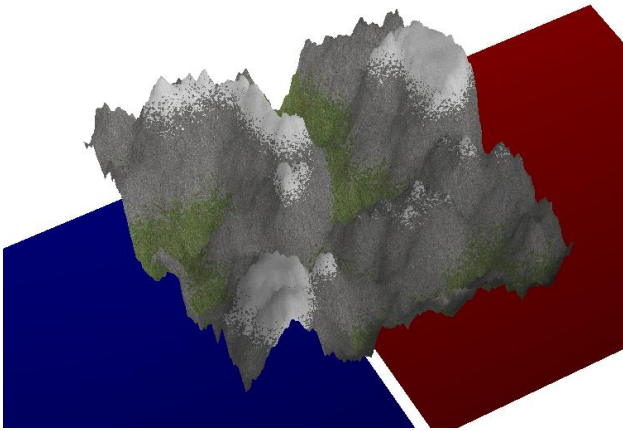


Figure 8: *Complex terrain before plate movement.*

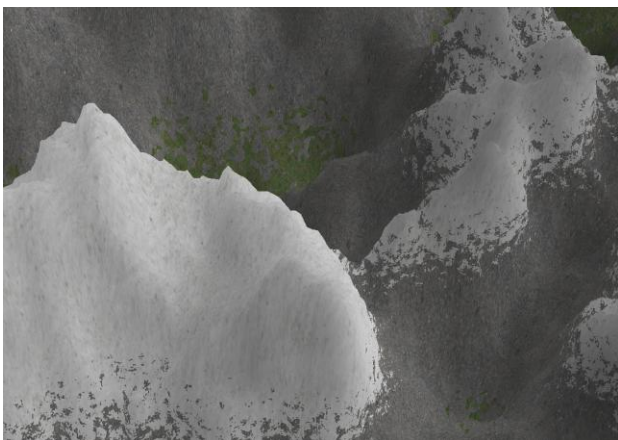


Figure 9: *Close-up of complex terrain after plate movement and vertex displacement.*

6 Limitations and Future Work

Tectonic activity is capable of producing a large variety of effects, and can occur under conditions which are not represented in this simulation. The simulation is limited to focusing on an edge between two plates, and therefore, cannot model tectonic effects occurring at interfaces containing more than two plates. Additionally, the model presented is not capable of simulating short-term, large tectonic events such as earthquakes. An avenue of future work would be to generate a timeline of geological events, and to display the occurrence of the events as the simulation runs. This would include both large and small effects caused by tectonic activity.

The simulation utilizes a 2-D model for the interactions between tectonic plates. Realistic geological changes that take place as a result of tectonic activity are too complicated to model using a simple 2-D structure. An alternate approach would be to simulate the rock in the Earth's crust as an extremely viscous fluid. The tectonic effects would then be modeled by forces being applied to this fluid.

While shaders are utilized for this project, the simulation runs on a single thread on the CPU. The simulation is CPU bound, and its number of vertices in the terrain mesh ultimately serves as the limiting factor in the size of the simulation. As computations occur independently for each vertex displacement, the simulation is embarrassingly parallel. Use of multiple CPU threads or the GPU would allow for larger and denser meshes to be utilized.

As the vertices are displaced both in OpenGL and in GLSL, normal maps and a workaround were utilized in place of computing the proper normals for each vertex. This ultimately limited the capability for interesting lighting effects, as such effects would place emphasis on the flaws in the normals.

An avenue for future visualization work would be to place a water level line. Any terrain that falls below

the water level line would be filled with water. A realistic water simulation would be ideal, but a simple approach would be to create textured solid geometry to fill the space. Adding realistic water effects to the geological timeline could produce interesting effects caused by large tectonic events, including the creation of waterfalls.

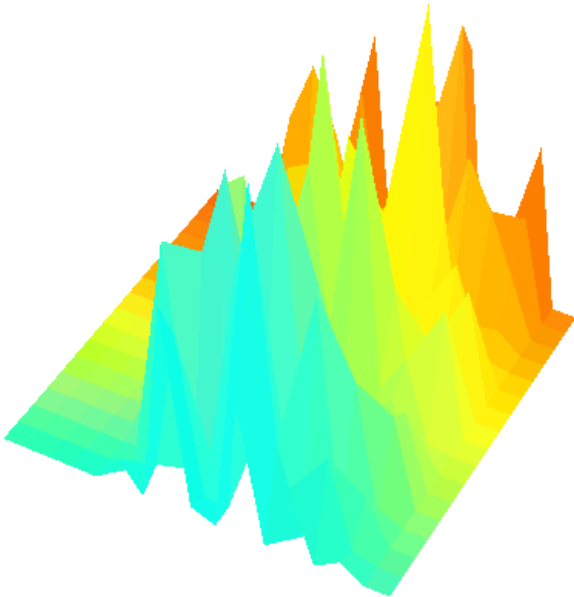


Figure 10: Proper scaling was a major concern. Here noise values have not been correctly scaled, resulting in widely divergent vertex displacements.

7 Conclusions

This project was successful in the creation of a physically plausible simulation of the long-term effects of tectonic activity. Geological effects are modeled using simple geography and physics. The graphical rendering aspect provides a visualization of the geological changes that occur over time as a result of the plate movement.

8 Contribution Summary

Daniel Benedetti's primary contribution was the visualization code, including meshes, textures, and shaders. Additional time was spent aiding in the debugging of the simulation code, for a total of approximately 60 hours of work on the project. Evan Minto created the plate simulation code, and helped debug some of the shader code, for a total of approximately 40 hours of work on the project.

References

- Guha, S. 2010. *Computer Graphics Through OpenGL: From Theory to Experiments*, Chapman and Hall. ISBN #1439846200.
- Kennelly, P. J., and Stewart, A. J. 2006. *A Uniform Sky Illumination Model to Enhance Shading of Terrain and Urban Areas*, *Cartography and Geographic Information Science*, 33:1, 21-36.
- Losasso, F., Hoppe, H. 2004. *Geometry Clipmaps: Terrain rendering using nested regular grids*, *ACM Trans. Graphics (SIGGRAPH)*, 23(3).
- Nowrouzezahrai, D., 2009. *Terrain Shader presentation*, ETH Game Programming Laboratory.
<http://graphics.ethz.ch/teaching/gamelab11/course_material/lecture06/XNA_Shaders_Terrain.pdf>
- Viitanen, Lauri, 2012. *Physically Based Terrain Generation: Procedural Heightmap Generation Using Plate Tectonics*. Bachelor of Engineering thesis, Helsinki Metropolia University of Applied Sciences.