Procedural Planet Generation

A Major Qualifying Project

submitted to the Faculty of the

Worcester Polytechnic Institute

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

_____
Andrey V. Sklyar

Date: _____

_____
Dr. Robert W. Lindeman
Advisor

Abstract for the Final Report for the Procedural Planet Generation Project

by Andrey V. Sklyar

Procedural planet generation is a way of creating computer-generated environments from a set of specified guidelines. In exploring these environments, only enough detail needs to be present so that the current screen seems realistic. This can be accomplished by using simplified versions of the objects in the environment until more detail is required. This paper describes a way to accomplish this level of detail switching using work done by previous researchers on progressive meshes.

This report begins by defining field-specific terminology necessary to understand the previous work and current implementation. It then describes work done by previous researchers in the area of level of detail meshes. It focuses on progressive meshes, which allow a high-resolution mesh to be simplified to a low-resolution approximation and then refined to any level of detail in between. A specific implementation of both the simplification process and the refinement mechanism are presented next, with special care given to the details of the simplification step. The report finishes by proposing improvements to the current implementation and suggesting areas of expansion where progressive meshes could be used.

Most of the previous work centers on progressive meshes. The first paper describes how to, given an initial high-resolution triangle mesh, produce a good approximation of the original with a smaller number of vertices. It does this by performing operations on the edges of the mesh while trying to minimize an energy function that attempts to preserve geometric properties of the original. This work is then used to introduce a continuous level of detail representation, referred to as a Progressive Mesh. Data structures for efficient implementation are introduced in the next summarized paper. The section continues by describing a way of creating real-time, continuous level of detail meshes that allows view-dependent refinement. The section concludes with a description of work that adds similar view-dependent capabilities to progressive meshes.

The next part of the paper describes a specific implementation of progressive meshes. It describes the mesh data structures needed to implement the simplification algorithm and the resulting progressive mesh. The energy function used is based on distance from the vertices of the original mesh. Identification of the edges and details associated with removing detail in the simplification step are described in depth. This is followed by a discussion of the effectiveness of the simplification algorithm used to preserve the qualities of the original model.

The report concludes by proposing improvements to the current implementation and suggesting expansions that could use progressive meshes as a subcomponent. These include implement view-*dependent* refinement, incorporation into a multi-user system architecture, and the continuation of the Procedural Planet Generation project.

Acknowledgements

Thank you Derek and Dr. Lindeman for all your help and guidance on this project.

Table of Contents

List of Figures

1 Introduction

The goal of procedural planet generation is to provide a way to easily create interesting computer-generated (CG) environments for use in videogames, cinematography, or other applications. Currently, applications like Terragen [13] and Mojo World [9] provide a means of doing this by using fractals. Alternatively, a highest resolution version of an object can be generated, and simplified versions produced to ease the rendering load of client applications.

Most of the content for the environments created by Procedural Planet Generation will be created automatically by abstract processes. These processes can be anything from artificial intelligence guided design to rigid city plans. The concept was originally inspired by fractals, though; both because they can be used to generate interesting objects from relatively simple specifications [10] and because of the potential for infinite detail in the models generated [9].

A fractal is defined as "a geometrically complex object, the complexity of which arises through the repetition of a given form over a range of scales" [9]. They have been used to generate terrains, tree leaves, ferns, and objects of infinite resolution. The "infinite resolution" is a result of the repetition of the form over a range of scales. Several examples of natural objects defined using fractals such as pine trees, flowers, and other plants can be found at the University of Calgary's Algorithmic Botany website [15].

Due to the finite nature of computers, computer-generated fractal objects can only be generated to some finite representation. When refinement is deemed necessary, it can be performed. Until then, there is a detailed model available for use by applications. Specifically, they can be used in virtual environments, where their natural complexity could be used to add to the visual experience of exploring that world.

Beyond generating individual objects, fractals can be used to model the growth of mountains [9, 13], ecosystems [15], and cities [16]. Since fractals are capable of filling in many of the interesting details of a model and introducing random variation in such details, it is conceivable that they could be used to generate full environments for use in anything from cinematography to videogames [9]. In the case that precise control of several components of the environment is

required, an initial model could be generated using fractals followed by specific refinements made manually.

Fractals are by no means the only approach for describing all aspects of the world we inhabit [9]. For some objects, there is either no way or no need to specify them using self similarity. At other times, a different representation may contain all the detail about a model. One such example would be using plans for a specific building generated by using a CAD program. These specifications are usually very large and tend to be hard to render at their full resolution. Fortunately, there is no need to store or display the model at its full resolution until it is necessary for visual quality. Therefore, if all that this building is used for is a long-distance shot of a cityscape, a much simpler version could be used than if the viewer were looking up at it directly from its entrance.

In order to use the lower resolution representations of highly complex models they need to be passed through a simplification step that can produce refinable versions of the models. This report describes the way that Progressive Meshes [2] can be used to achieve these levels of detail. It also describes a specific implementation of the simplification algorithm and the resulting refundable model.

2 Background Information

This project implements the part of the procedural planet generator that allows highly detailed models to be simplified after they have been generated or added to the world. The simplifier takes in a three-dimensional mesh and produces a continuous level of detail representation allows for simpler versions of the object to be used when the full detail is not necessary. The background necessary to understand the content of this report is presented in this section.

2.1 Anatomy of a Mesh

A polygonal mesh is a piecewise planar approximation of a three-dimensional (3-D) surface [12]. It is defined by a set of vertices (3-D points) and faces (closed, usually convex, polygons made up of a small subset of the vertices). Triangle meshes are a specific type of polygonal mesh where all the faces are made up of exactly three vertices. Any mesh can be turned into a triangle mesh by triangulating the faces. All the models in this project are assumed to be triangle meshes.

In order to display a mesh on the screen, the computer needs to know how light interacts with it. For example, when drawing a sphere, it should look smooth rather than blocky. This can be accomplished by specifying normal vectors associated with each vertex in the model, allowing the computer to compute the intensity with which the light illuminates the object at that point: fully illuminated if the normal points towards the light source and fully in shadow if it points away. Intermediate values on the vertices can be interpolated by using a shading model such as Gouraud or Phong [12] shading models.

One can also place a texture onto the object by associating texture coordinates with each vertex. The renderer will then be able to take a texture map, or function, and appropriately map it onto the surface.

2.2 The View Frustum

The View frustum is a four-sided, semi-infinite pyramid that represents the volume that the user may be seeing View Dependent Refinement of Progressive Meshes [5]. Testing against the bounds of this figure allows a program to determine whether an object is out of sight, which can be used as criteria to coarsen the current representation of that object. Even though these parts of

the model would never be displayed, determining that they should not be displayed will be done more quickly since fewer vertices will have to be considered.

2.3 Levels of Detail

When an object is far away, it is visually smaller, and therefore takes up fewer pixels on the screen. Suppose that an apartment building were being modeled. A typical apartment building would have windows, balconies, and a fire escape. If the building was so far away that it only took up a 5 x 5 pixel block, it would be hard to distinguish any detail except for maybe that it is a building. At this resolution, a simple box would probably be as convincing an approximation as the full-detail figure. If the camera were to move closer, it would reach a point where the building would be displayed at a high enough resolution that large features like the fire escape and the balconies would this become distinguishable. However, details such as the individual bars on the balconies or the individual steps on the fire escape would still be indistinguishable, calling for another version of this model. At some point the viewer would get close enough to the model that all the details would become discernible. It is only at this point that the viewer would benefit from having the full-detailed model.

This notion of using a set of predefined, lower-resolution models as substitutes for an object when the full detail is not necessary is referred to as "discrete levels of detail (LoD)." It allows for reductions in the amount of information that needs to be sent to the display initially, allowing a user to interact with the world more quickly than if they had to wait for all the information to load. It also reduces the number of polygons that need to be rendered per frame, allowing for higher frame rates.

The decision on which model to use is usually based on how far away the object is, and whether or not it is in the view frustum. When the object comes close enough that a higher resolution model will improve the visual quality, the model being used is switched to the higher resolution. Unfortunately, as the two models being used are different, the change in the detail might be noticed. This noticeable switch between levels of detail is referred to as "popping." One way of reducing the severity of the popping is to introduce the change in the models over several frames. This can be done by interpolating intermediate values, and is referred to as morphing.

When there are many refinements that can be made to the model at any one point in time that introduce a small amount of detail each time, the representation is referred to as a *continuous* LoD representation. These sorts of models lend themselves to view-dependent refinement, which is discussed next.

2.4 View-Dependent Refinement

Discrete levels of detail are good for modeling objects that fit within one screenshot since all the extra data adds to the visual quality of the image. So, for example, if discrete levels of detail were used to model a human-sized statue moving closer to it would increase the amount of detail rendered, all the way until the model fills up the whole screen. At this point, only roughly half the model's detail will be in view, since half of it will be facing away from the viewer. Now consider using discrete levels of detail to model the apartment building in the previous example. By the time that the viewer is close enough that the door to the entrance takes up the majority of the screen, all of the information about all of the windows, stairwells and balconies will be loaded. In any one screenshot the user will only be able to see a small part of the building. The renderer, nonetheless, will have to consider, and possibly render, detail that is far away on the building at a higher resolution than is visually necessary. A solution to this problem is to use view-dependent refinement. This is usually done with *continuous* LoD.

Continuous LoD allow for dynamic refinement of parts of a model (such as specific vertices) based on refinement criteria. These criteria can include distance from a part of the model to the viewer, whether the part is facing towards or away from the viewer, and whether the difference in screen pixels between displaying the original model and the approximation is within a specified tolerance at that location. By repeatedly applying these refinements based on viewing criteria, a model with high resolution near the viewer and other key locations (such as the silhouette borders), and lower resolution at hidden or distant parts can be attained.

Since continuous levels of detail can allow for refinement in specific areas, the amount of popping when switching levels can be significantly reduced, since less of the model is affected. Combined with morphing, the transitions can become practically unnoticeable [5].

3 Previous Work

Much research has been done in the area of levels of detail. Peter Brown [8] provides an in-depth discussion of LoD. This section describes work related to progressive meshes, since they were the main focus of this project. Concepts from all of these papers were used in the implementation described later in the paper.

3.1 Mesh Optimization

The goal of Mesh Optimization [1] Hugues Hoppe *et al.* is, given an initial high-resolution triangle mesh, to produce a mesh with the same topological layout with a smaller number of vertices. They accomplished this by repeatedly performing one of three edge operations while minimizing an energy function.

The three basic operations that can be performed are an edge collapse, an edge split, and an edge swap (Figure 1). An edge collapse removes an edge by removing the two faces that share the edge and then updating the mesh connectivity. An edge split introduces a new vertex in the middle of the edge and adds edges to the remaining vertex in the neighboring faces. An edge swap changes which vertices in the two adjacent triangles share the edge.
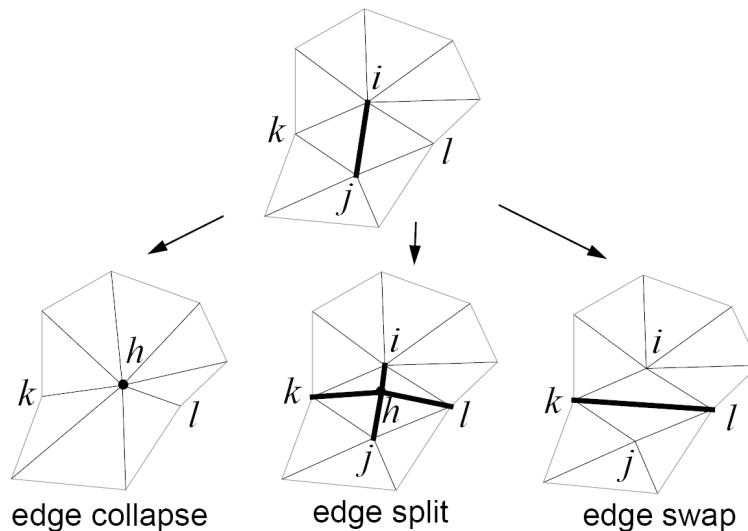

edge collapse   edge split   edge swap

Figure 1: Edge Operations [1]

One thing to notice about these operations is that they only have local effect on the overall mesh. They only possibly deform those faces incident on the vertices defining the edges removed, modified, or created. This adds to the efficiency of the optimization algorithm, since each step the calculation needs to be done on only this local subset.

The three components of the energy function are the distance energy (E_{dist}), the representation energy (E_{rep}), and the edge-spring energy (E_{spring}). Distance energy measures the squared deviation of each point currently used to approximate the mesh from the surface of the data model being optimized. This assures that, at each step, the current approximation stays close to the surface of the original mesh. Representation energy allows the user to specify how important it is to have a small number of vertices at the expense of a less-accurate approximation of the original model. Finally, the edge-spring energy is used to guide the simplification process in the beginning stages when there are many edges that are candidates for operations. Its influence is reduced as the number of vertices in the mesh gets smaller. Mathematically, the energy is expressed as follows.

$E(K, V) = E_{dist}(K, V) + E_{rep}(K) + E_{spring}(K, V)$ (eq. 1: Energy Function [1])

Where K represents the connectivity of the mesh (which vertices make up a face and which faces are adjacent to each other) and V is the set of vertices presence in the mesh. Edge collapses and splits modify both K and V. Edge swaps only affect K.

The algorithm applies one of the above-mentioned deformations to a randomly chosen edge and then optimizes the vertex positions in the neighborhood of the transformation. Edge collapses are considered first, followed by edge swap, and then edge split. The choice of which operation to perform is constrained by a requirement that the resulting model not self intersect (which can happen because of an edge collapse or an edge split). This is accomplished using a heuristic that ensures that the angle between the resulting faces is not too large.

This process is effective in simplifying the original mesh and preserving detail where it needs to be. Regions with high curvature receive more vertices than flat surfaces, which are approximated

by a few large faces with long edges. Since the algorithm preserves the topological type of the model, it is also able to identify creases in the model by specifying the threshold angle for smoothness. Using this information, the models can then be segmented into smooth components, and normals can be calculated for each of the surfaces independently.

3.2 Progressive Meshes

The work described in Progressive Meshes [2] introduces a continuous level of detail representation that takes into account the geometry of the mesh as well as discrete and scalar attributes associated with it. It performs a simplification step similar to Mesh Optimization [1], taking into account discrete attributes associated with faces (e.g., a shader function) and scalar attributes associated with the corners of each face (e.g., normals and texture coordinates).

In order to simplify complicated models for rendering purposes, shading information needs to be included as a parameter in the simplification step. Specifically, progressive meshes consider the textures associated with each face and the normal vector information and texture coordinates at each face corner. Note that the material identifier or shader function associated with the face is a discrete value, while texture coordinates and normals are scalar values that can be interpolated.

Associating the scalar attributes with the corners of the face rather than the vertices themselves allows for creases to be modeled and rendered. For smooth portions of the surface, there is usually only one normal associated with the vertices. At curve discontinuities, each of the surfaces will have different normals associated with the same vertices.

These discontinuities in attributes, both discrete and scalar, happen along edge boundaries (between two faces). The edges where this happens are labeled as "sharp," and define a set of discontinuity curves on the surface. Sharp edges are border edges[1], edges between two adjacent faces that have different, discrete attributes associated with them, or edges that have different scalar attributes at adjacent corners.

---

[1] Border edges are those which only have one incident face, rather than two.

The simplification algorithm implemented for this project only uses edge collapses as defined in [1], as the authors found that this operation alone was sufficient for achieving good approximations. Associated with an edge collapse is a vertex split, which defines the opposite operation. The end result of the simplification process is a simplified version of the original model, and a set of refinements (vertex splits) that, when applied in order, will return the mesh to the highest resolution state. The tuple consisting of the base mesh and the sequence ordered of vertex splits is referred to as a *progressive mesh*.

When an edge collapse is performed, one of the vertices (v_t in Figure 2) is removed and the edges that were incident to it now connect to the other (v_s). It also removes the two faces that were incident on this edge. The collapse moves the corners of the remaining faces and changes the location of v_s and the scalar attributes stored at the corners. These changes are recorded in the vertex split record's attribute field (not shown in the illustration).
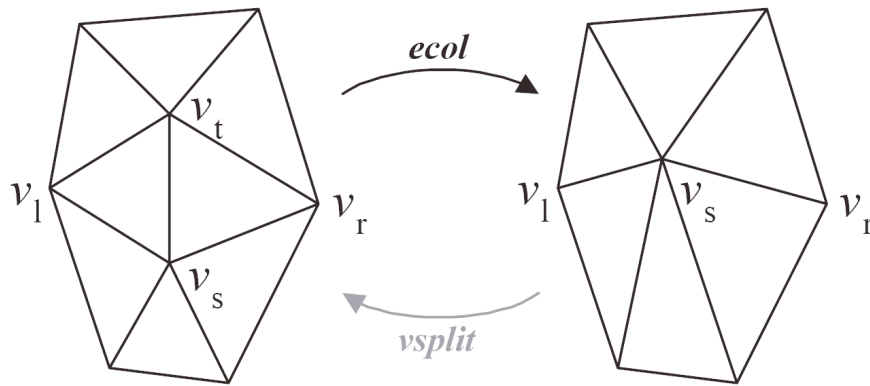
Figure 2: Edge Collapse [2]

The energy function used for the simplification is similar to that of [1], but takes into account discrete and scalar attributes. It penalizes removal of discontinuities when specific sharp edges are collapsed (eq. 2).

$$E(M) = E_{dist}(M) + E_{spring}(M) + E_{scalar}(M) + E_{disc}(M) \qquad \text{(eq. 2)}$$

In this equation, *M* represents a tuple (*K, V, D, S*) where *K* and *V* are the same as in [1], *D* represents the discrete attributes associated with the faces, and *S* is the set of scalar attributes associated with the corners of the faces.

Geometric optimization happens first, using the distance and spring energies. Once the vertex positions have been determined, the scalar attributes are optimized. Rather than randomly choosing the next edge to collapse, the algorithm keeps a priority queue of all the candidate edge collapses sorted by their energy change. After the collapse is performed, the weights of all the affected edges in the neighborhood of the collapse are updated, and the queue is reordered. When removal of an edge would break a discontinuity curve, the algorithm adds an amount to its energy value that compensates for the disruption.

After this simplification algorithm is applied to an initial mesh, a progressive mesh is constructed. It can be refined using vertex splits and coarsened by applying them in reverse. These operations introduce or remove vertices and faces, which can result in popping. This can be made less apparent by using alpha-blending geomorphs [2]. By introducing the change over several frames, any vertex split or collapse can be animated smoothly. The intermediate values can be interpolated either linearly (such as vertex position) or nonlinearly (normals are interpolated over the unit sphere). Discrete attributes are also introduced smoothly, since the faces that reference the discrete attributes are introduced or removed smoothly as well.

Progressive meshes also lend themselves to progressive transmission over a network. The base mesh can be sent and the user can begin interacting. Vertex splits can be sent incrementally, refining the mesh as they arrive. When all the splits arrive, the client will be able to reconstruct the highest-detail mesh.

The paper also describes a way of doing selection of refinements by using the closest living ancestor. A better approach is described in View Dependent Refinement of Progressive Meshes [5].

3.3 Efficient Implementation of Progressive Meshes

Efficient Implementation of Progressive Meshes [3] describes the data structures used to efficiently implement progressive meshes. It introduces the concept of "wedges" to represent corner attributes and defines the dependencies between faces, wedges, and vertices. It then describes a way to add and remove detail from a progressive mesh.

A mesh is represented by a set of vertices, a set of corner attributes called wedges, and a set of faces. The vertices store 3-D locations on the model. Wedges associate scalar attributes with the vertices and represent vertex-adjacent corners that share the same attributes (Figure 3). There is a many-to-one relationship between wedges and vertices. A face stores discrete attributes, three wedges in counterclockwise order referring to the corners of the face, and pointers to the three edgewise-adjacent neighbor faces. The reference to vertices is made indirectly through the wedges.
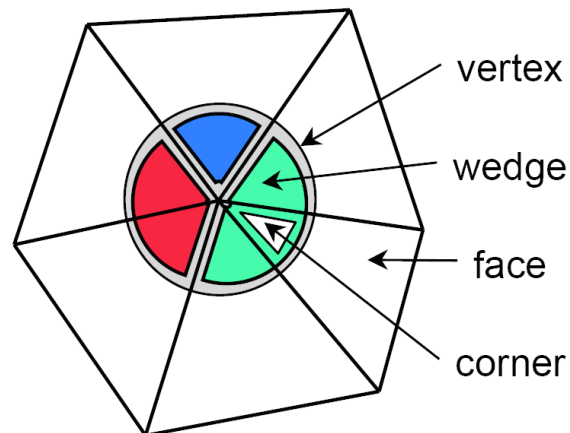
Figure 3: Wedges [3]

A progressive mesh stores a mesh object (the base mesh) and a list of vertex splits. Due to the Face → Wedge → Vertex data dependency, vertex splits need to reference a Face (*flclw*, in Figure 4) in order to have appropriate access to the connectivity of the mesh. In this representation, the parameters of the vertex split are identified by an index within the face pointing to v_s (*vs_index*) and a number of faces to go around v_s in a clockwise order to reach v_r (*vlr_rot*). The left vertex, v_l, is taken to be the next vertex within *flclw* after v_s in clockwise order. Changes to the wedge attributes in vertex positions and extra information, including

continuity of wedges over edge boundaries, is stored as well. This provides enough information to introduce the new faces and vertices, and update the connectivity and appropriate attributes after a split or collapse is performed.
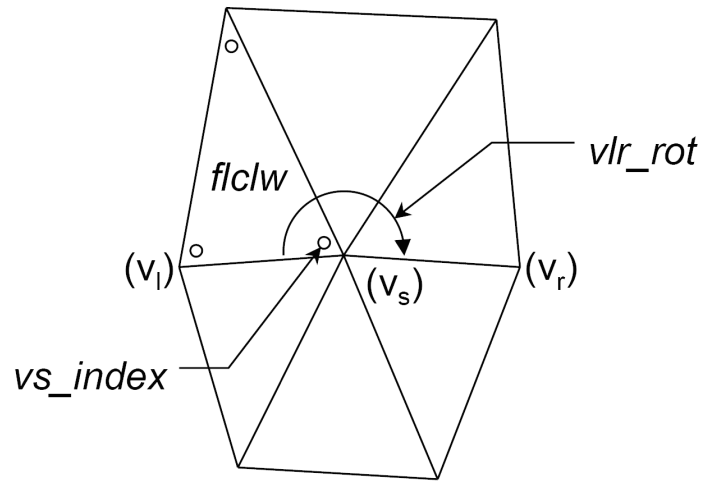
Figure 4: Vertex Split Encoding [3]

A progressive mesh read stream abstracts the source of the progressive mesh. Specifically, it allows progressive meshes to be used when the data is stored in memory, when it is progressively received over an input stream, or when it is received from the stream and cached.

The progressive mesh iterator extends the mesh class with the ability to perform vertex splits and edge collapses. Edge collapses are only allowed for the cases where there is an internal store of all the vertex splits. Since the iterator is itself a mesh, it can be rendered at any point in time. It also provides a way of generating a mesh with a specified number of vertices or faces as well.

Geomorphs are implemented by keeping track of which vertices split or collapse to become which other vertices. Once this ancestry information is known for all modified vertices, the initial and final vertex positions and attributes of the edges can be alpha blended.

3.4 Dynamic View-Dependent Simplification of Polygonal Models

Motivated by visualization of complicated structures for scientific purposes, Dynamic View-Dependent Simplification for Polygonal Models [3] describes a way of creating real-time,

continuous level of detail meshes that allows view-dependent refinement. They also use more-sophisticated heuristics for determining whether detail needs to be added or removed at a region.

The process of creating a refinable mesh is similar to progressive-mesh simplification. In performing the edge collapses, one of the vertices is chosen as the child and the other as the parent. After each collapse, the child is removed and all of its edges are linked to the parent. Collapses are done in a binary fashion, without interpolating the coordinates of the parent vertex after the collapse.

In order to assure that the mesh's surface does not fold over itself, the authors define dependency relationships between the vertices, which they refer to as regions of influence. The region of influence of a vertex is the set of vertices that are edge-adjacent to it. In order to perform a vertex split the region of influence that was present when it was collapsed during the simplification step need to be in the mesh and adjacent to the parent vertex. For edge collapses, the regions of influence for both vertices need to be in the mesh and adjacent to the edge.

The algorithm attempts to make balanced merge trees to allow similar amounts of refinement on all parts of the mesh. It goes about this by performing all the edge collapses where the regions of influence do not overlap. This set of collapses becomes a level in the merge tree. This step is then repeated with the remaining edges until the original mesh is significantly simplified. Edges are considered in order of edge length, from shortest to longest.

In order to allow refinement at runtime, each node holds a set of attribute ranges that the attribute values can take until it needs to be refined. One of the attributes is Euclidean distance from the node to its child in the node to its parent. This allows the renderer to decide whether to split the vertex or collapse an edge by making sure that the screen-projected length is within a certain threshold. If the distance is too small, the edge is collapsed. If the distance is large, then the edge is split and more detail is added.

The other attribute ranges stored at the nodes of the tree are based on local illumination, visibility culling, and silhouette boundaries. Local illumination gives preference to areas that have more-

direct light. Visibility culling attempts to reduce the number of faces on the backward-facing side of the model, or outside of the viewing area. Detail is also preferred at silhouette boundaries, where there is potential for high contrast between the model and either itself or the background.

In order to display the mesh, a set of active vertices and faces is recorded. Whenever a frame is rendered, each of the active vertices is considered for a split or collapse. The choice is made based on whether the above-mentioned ranges fit within the specified threshold from the current viewing position. If a split needs to be performed, the child vertex is added to the list of active vertices, and the resulting faces are added to the list of active faces. The added vertices are then recursively tested to see if they need further refinement. Refinement stops when either the node's range is within the specified threshold or there are no more splits that can be performed in this area. Removal of detail proceeds in the opposite fashion.

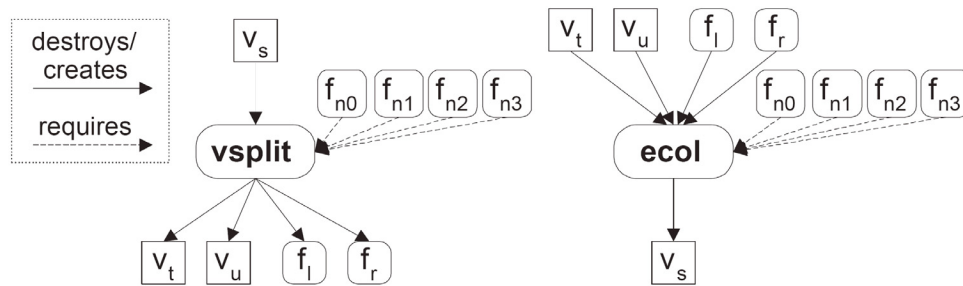## *3.5 View-Dependent Refinement of Progressive Meshes*

View-Dependent Refinement of Progressive Meshes [5] describes a method of implementing view-dependent refinement of progressive meshes. Similar to work described by Xia *et al.* [4], a parent and child relationship between the vertices of the intermediate meshes is determined. The vertex split operation is augmented to include neighborhood information that is used to place constraints on which splits or collapses may be performed. A set of refinement criteria and a generalized triangle strip creation method for progressive meshes and their implementations are described.

The set of vertex splits is computed just as in Progressive Meshes [2], except that the two vertices defining the edge (v_u and v_t) are removed and a new vertex (v_u) is introduced (Figure 5). This new vertex becomes the parent of the two vertices that defined the edge. Also stored in the split (Figure 6) are the two faces, f_l and f_r, and their neighbors (f_{n0}, f_{n1}, f_{n2}, and f_{n3}). This set of four faces provides the necessary link into the mesh that assures the legality of a collapse or a split. When performing a vertex split, these four faces need to be present, with f_{n0} adjacent to f_{n1}, f_{n2} adjacent to f_{n3}, and v_s in the positions illustrated below. When performing a collapse, f_l and f_r need to be active in the mesh, with v_t and v_u in the appropriate places, f_{n0} and f_{n1} have to be adjacent to f_l, and f_{n2} and f_{n3} have to neighbor f_r.
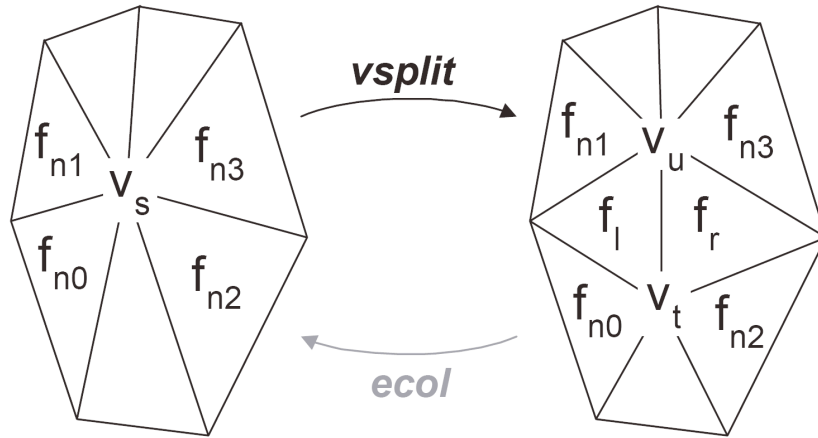
Figure 5: Vertex Split with Neighborhood Information [5]

The criteria used to determine whether to add or remove detail is based on similar metrics as defined in [4]. In order to determine whether an object is within the viewing area, bounding spheres are computed at each node of the merge tree. Determining whether a sphere is within a frustum can be done quickly, and those vertices completely outside are not refined. Surface orientation at each vertex is computed using groups of normals from its children to create a bounding cone. The final metric is the screen-space geometric error that assures that the difference between the original mesh and the approximation mesh on the current screen is below a given tolerance.

Figure 6: View-Dependent Refinement Constraints [5]

Just like in [4], all of the active vertices are considered for a split or collapse before rendering each frame. The decision to perform the refinements is based on the metrics described above. Finally, a greedy algorithm is used to generate triangle strips before display to increase performance.

4 Progressive Meshes for Procedural Planet Generation

As discussed in the introduction, part of the procedural planet generation system is a method by which to generate simplified versions of objects and refine them when more detail will add value to the scene. Progressive meshes provide exactly this mechanism.

Since the first objects generated by procedural planet generation will be terrain landscapes, the output of a fractal terrain generator was used as the model on which to test the simplification algorithm. There exist continuous level of detail representations and simplification algorithms specialized for height field terrains [6, 7]. Progressive meshes were chosen instead since they provide a way of simplifying and refining arbitrary triangulated models not constrained to be height fields.

The ultimate goal was to provide a view-dependent continuous level of detail model, similar to the one described in View Dependent Refinement of Progressive Meshes [5]. Due to time constraints, only view-independent continuous level of detail model generation was implemented. Several parts of the code, such as recording neighborhood information in the vertex splits, reflect this discrepancy. Completing the view-dependent aspect of this representation is part of future work.

4.1 Overview

In order to create a progressive mesh version of a terrain, a triangulated version of a height field created by the Terragen fractal terrain generator [13] was passed to the simplifier. The simplifier then repeatedly performed edge collapses while trying to minimize an energy function based on distance from the vertices of the original mesh. The resulting progressive mesh provides a representation that can be used to generate view-independent continuous level-of-detail meshes.

4.2 Terrain Generation

Fractal terrain generation allows a user to specify the parameters which the terrain should satisfy and produces a randomly generated model that satisfies those constraints. Texturing & Modeling [9] gives a good introduction to both procedural and fractal generated models. For this project, Terragen was used, a free-for-noncommercial-use fractal terrain generator that is capable "of

photorealistic results for professional landscape visualization, special effects, art and recreation," as quoted on the homepage [13].

Terragen provides an export mechanism by which the height field data defining the terrain can be stored to a binary file as 257*257 16-bit little-endian unsigned integers that represent a 257 x 257 height field. This format was chosen because it was easy to load the contents of the file into a mesh object.

4.3 Mesh

A mesh object (Figure 7) represents a 3-D triangle mesh. It stores an array of vertices and an array of faces. Each face contains three indices into the vertex array and three indices into the face array. The vertex references define the three points that make up the triangle. The face references the three faces that are adjacent to this one.

Vertices represent 3-D points, and are implemented using Theilana[2] physics engine's point3d class. Three-dimensional calculations such as finding the distance between two points is also performed using this library.

```
class Vertex {

  point3d point;

};

class Face {

  //indices into the vertices in
  //counterclockwise order.
  int vertices[3];
  //indices into the neighbor faces
  //-1 means no neighbor
  int neighbors[3];

  //the face adjacent to the edge
  //(vertices[i], vertices[(i+1) % 3])
  //will be stored in neighbors[i].

};
```

```
class Mesh {

  //parse and triangulate a height field
  //from a binary file.  Assumes a square,
  //regular, grid of 257x257 2 byte
  //little-endian integer heights.
  void load(char* file_name);

  //traverse the faces and display them
  void draw();

  //list of all vertices (with normals
  vector<Vertex> vertices;

  //list of all current faces
  vector<Face> faces;

};
```

Figure 7: Mesh Classes

The two thee-element vertex and neighbor arrays of the face class are stored in counterclockwise order (Figure 8). This ordering becomes very important when updating mesh connectivity, since

---

[2] Written by Derek Radtke [14]

the index of the next face to be examined can always be determined. Also, the face shares vertices indexed by `vertices[i]` and `vertices[(i+1)mod3]` with the face indexed by `neighbors[i]`.
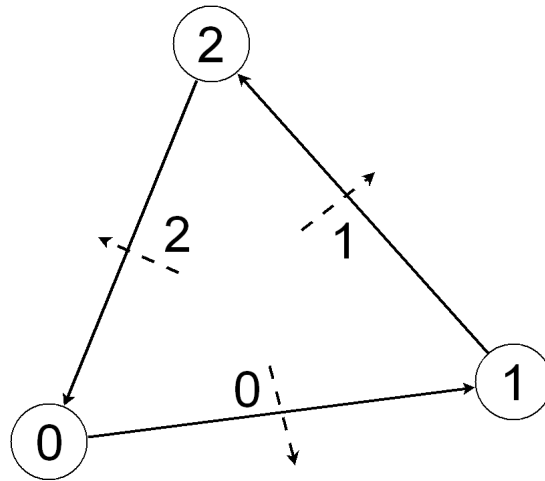
Figure 8: Face Data Structure

In order to load a height field for use in the simplification algorithm, it needs to be triangulated and stored as a mesh object. This is what the load method does. It reads in all the vertices and creates the faces with appropriate connectivity.

A height field can be thought of as a rectangular grid of values. In order to triangulate this grid, each of the squares is given an additional diagonal. The loader creates the diagonal that separates each square into a bottom-left in the top-right triangle. The top-left vertex of the square is at index zero in both triangles, and the rest are numbered in counterclockwise order. The bottom-left triangles are always linked to the top-right neighbor in the same square. They are also linked to the neighbors to the left and the neighbor to the bottom unless the square is either in the leftmost column or the bottommost row, in which case the value of -1 is assigned to the neighbor field. Similarly, each top-right triangle is linked to the bottom-left triangle in its square and to the top in the right neighbors in all but the topmost row and rightmost column.

A mesh can be displayed by calling the draw method. It simply goes through the list of faces and sends face's three vertex coordinates down the rendering pipeline.

4.4 Simplification

The energy function used is described as follows: an edge collapse can be thought of as individually moving each of the two vertices that define the edge to a new location. An energy value can be associated with this motion by placing virtual springs at each of the vertices of the original mesh, so that any deviations from the original have a cost. The energy associated with a vertex is given as the sum of the individual tensions of these springs acting on it. The change in energy of an edge collapse is the difference between the initial energies of the two vertices defining the edge and the final vertex to which the edge collapses.

Edges are identified in terms of a face and a neighbor index. Spring constants are assigned to the endpoints of edges, which are vertices. This allows identification of borders and other surface features which might require higher spring constants.

Next, the energy change of collapsing each of the edges is computed. Since the energy change of an edge collapse depends on the location of the final vertex, several locations are tried. This vertex location and the change in energy are associated with the edges.

After this set up, the algorithm generates a progressive mesh by repeatedly performing the least expensive edge collapse until a specified number of faces are attained. For each collapse performed, the algorithm saves the necessary neighborhood information required to later add this removed detail to the progressive mesh.

4.4.1 Energy Function

The change in energy is calculated by taking the difference between the energy after the collapse and the energy before the collapse. The initial energy is defined as the sum of all the spring energies associated with each of the two initial vertices that define the edge. The final energy is the sum of the spring energies associated with the single vertex that replaces them.

4.4.2 Mesh Simplifier

The mesh simplifier takes in a mesh object, identifies all of its edges, associates spring constants with all of the vertices, and computes the initial energy changes of collapses (Figure 9). It does

so by constructing a simplification mesh, which provides the data structures that implement the above-listed concepts, as they are not part of the mesh class.

```
class MeshSimplifier {

    //return the base mesh of the
    //specified size and a sequence
    //of legal vertex splits
    PMesh getPMesh(int num_faces);

    //The mesh being simplified (with
    //vertex springs and edges).
    SimplMesh mesh;

    //indicies into the simplification
    //mesh's edge list
    vector<int> collapse_pointers;

    //the accumulated vertex splits
    //that will be used to make the
    //progressive mesh
    vector<Vsplit> vsplits;

};
```

Figure 9: Mesh Simplifier Class

The `getPMesh` method of the simplifier is called in order to get a progressive mesh with the desired number of faces in the base mesh. This function repeatedly applies the least expensive edge collapse until the desired number of faces remains. The simplifier keeps a list of indices into the edges of the simplification mesh, finds the index of the cheapest collapse, and calls the `perform_collapse` method of the simplification mesh to perform the collapse. The `perform_collapse` function returns a vertex split and the number of faces removed. In the case that no faces are removed, the split is not recorded (this scenario is discussed in depth in the performing the Section 4.4.4).

The simplifier collects the vertex splits as generated by performing the collapses. They will be in reverse order, since the last collapse performed will be the first split that will be done by the progressive mesh. The getPMesh method finishes by passing the final simplification mesh and list of vertex splits to the progressive mesh constructor, which reorders the vertices and faces to reflect the order in which they are introduced into the model.

4.4.3 Creating the Simplification Mesh

The Simplification Mesh is a Mesh augmented with anchoring springs associated with the vertices of the original mesh and edges associated with the boundaries between faces (Figure 11). The construction of a simplification mesh begins by copying the face information provided in the Mesh (vertex and neighbor indices) into the Simplification Mesh's face array. Edges are then identified using a face index and a neighbor index within that face. This means that edges are defined in terms of faces, not vertices, since the vertices that make up the face will possibly be changing as edges are collapsed (Figure 10).

```cpp
//a vertex that can exert a spring force
//on the simplification vertices
class AnchorVertex : public Vertex {

    //compute the spring energy from
    //this anchor to the provided vertex
    float energyTo(const Vertex &v) const;

    //spring constant for moving this vertex
    float k;

};

//vertices that keep track of ancestry
//for calculating vertex energy
class SimplVertex {

    //the current vertex information
    Vertex v;

    //the sum of the magnitudes of the
    //anchor "forces" acting on this vertex
    float ancestor_energy;

    //the ancestors from the original mesh
    set<AnchorVertex> ancestors;

};
```

```cpp
//The operation that will be used
//on the PMesh
class Vsplit {

    //vu is split into vs and vt when
    //performing inverse (edge collapse),
    //vt and vs collapse to vu
    int vs_i, vt_i, vu_i;

    //face numbers of the faces created.
    int fl_i, fr_i;

    //face numbers of the faces that
    //have to be surrounding fl and fr
    //for collapse, or that will be
    //surrounding them after split.
    int f0_i, f1_i, f2_i, f3_i;

};
```

```cpp
//a weighted edge collapse
class Ecol {

    //All of the fields below are indices
    //to the objects described.

    //face that is being collapsed.
    int fl_i;
    //neighbor who shares this edge.
    int fr_index;

    //vertex to which this edge collapses
    int vu_i;

    //Change in energy
    float dE;

    //whether this edge exists in the mesh
    bool active;

};

//Face used by simplifier
class SimplFace : public Face{

    //index into the edge collapses on
    //the edges of the face.
    //edge[i] is the edge shared with
    //neighbor[i]
    int edges[3];

    //whether face is in the mesh.
    bool active;

};
```

Figure 10: SimplMesh Prerequisites

21

```
//A mesh that takes into account edges and spring energies
class SimplMesh{

    //make it harder to move boundary vertices.
    void make_anchors(const Mesh &m);

    //calculate the change in energy for the given edge collapse
    void assign_collapse_information(const int &e_i);

    //calculate the change in energy for the given edge collapse
    //and assign the miniumum vu to verteices[vu_i]
    void assign_collapse_information(const int &e_i, const int &vu_i);

    //don't create another vu, just modify the current one
    void update_collapse_information(const int &e_i);

    //collapse the edge, update the mesh and the affected edges
    //and save the vsplit record.  Return the number of faces removed
    //and the vsplit record.
    std::pair<int, Vsplit> perform_collapse(int e_i);

    //go around old_v_i and replace it with new_v_i
    void replace_corner_vertex(int old_v_i, int new_v_i, int fs_i, int fe_i);

    //redo the edge collapse calculations for the edges
    //around this vertex
    void update_ecols(int v_i, int fs_i, int fe_i);

    //all the vertices in this simplification.
    vector<SimplVertex> vertices;

    //all the edges that can be collapsed
    vector<Ecol> edges;

    //simplifier faces
    vector<SimplFace> faces;

};
```

Figure 11: SimplMesh Class

Since an edge collapse removes faces, the Simplification Mesh needs to make sure that edges are always defined in terms of active faces (ones that have not been removed yet). This process is described in Section 4.4.4.

Faces also hold references to edges. This makes the process of updating edge collapse information in the neighborhood efficient, since updating the edge collapse information in the neighborhood of a just-performed edge collapse requires a simple traversal of faces in the neighborhood.

The next step in the construction is to create the vertices with their associated sets of ancestors. This set represents points from which the springs acting on this vertex originate and is always a subset of the vertices in the original mesh. Initially, each vertex has only one ancestor (the vertex

at which it is originally located). When edges are collapsed and vertices are moved, the union of the ancestries of the two vertices becomes the ancestry of the vertex to which they collapse.

In assigning spring constants to the initial ancestry, we give border vertices a higher spring constant than center vertices. Border vertices are identified as being the endpoints of border edges (ones whose neighbor index dereferences to -1).

With edges identified and the vertices assigned initial ancestries, the constructor finally calculates the vertex to which each edge collapses and the energy associated with that collapse for all of the edges. The algorithm tries a discrete number of points along the edge and uses the one with the smallest energy change, as described above. A new vertex is introduced for each split in order to provide expansion to view-dependent refinement of the progressive mesh.

4.4.4 Performing the Edge Collapses
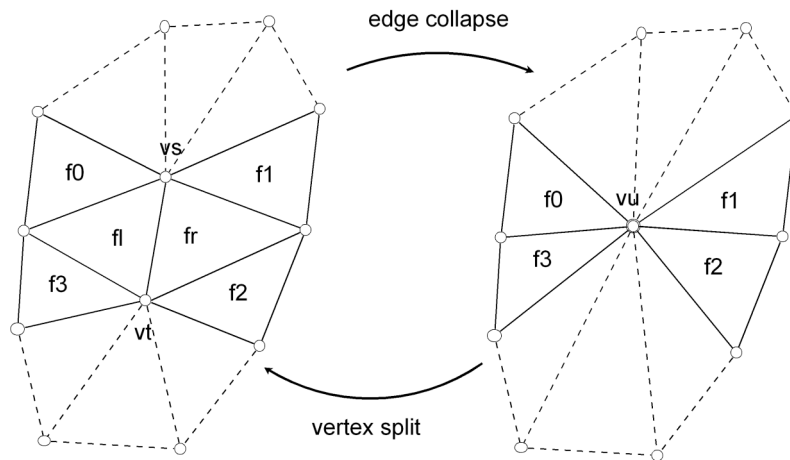
Figure 12: Neighborhood Labeling

If the face referenced by the collapse is *fl* and the specified neighbor as fr, then the faces *f0*, *f1*, *f2*, and *f3* are as labeled in (Figure 12), with *f0* neighboring *fl* on top and *f3* on the bottom, and similarly, *f1* neighboring *fr* on top and *f2* from the bottom. The information stored in a vertex split is (*vs, vt, vu, fl, fr, f0, f1, f2, f3*). A vertex in the place of *vl* and *vr* is required to exist when either a collapse or a split is performed, but they are not constrained to specific vertices and are thus not recorded but rather inferred from the neighbor faces. The vertex split and the number of faces that were removed are returned as the result of each call to `perform_collapse`.

4.4.4.1 General case

In the general case, all of the faces and vertices described in the vertex split exist. The collapse proceeds by removing *fl* and *fr*, which effectively puts a hole in the mesh. We stitch up this hole by connecting the two faces that were neighboring each of *fl* and *fr*, and updating the vertex references of all the faces that had either *vs* or *vt* at a corner to point to *vu*. The process finishes by updating the collapse information of all the edges in the affected faces.

Care must be taken while updating the connectivity of the faces to make sure that all edges' face-indexes point to active faces within the mesh, and that only one edge defines the boundary between two faces. Since edges are defined in terms of faces, removing *fl* or *fr* from the mesh could possibly result in the algorithm eventually[3] considering an edge that does not identify a boundary within the mesh. On the other side of the spectrum, connecting the neighbors of *fl* and *fr* merges two face boundaries into one. We need to make sure that neighboring faces' edges refer to the same edge, since otherwise we might try to collapse the same edge twice, which would be an error, since it would not exist the second time.

Making sure that an edge's face-index references a live face is accomplished by testing whether any of the edges on the face being removed are identified by that face. If they are, we update the edge's face index to point to *fr* (the dereferenced neighbor as specified by the edge) and the neighbor index for the edge becomes the index within *fr* that points to the face being removed. This effectively identifies the boundary, since after the connectivity is updated the neighbor index will point to live face. Double identification of edges is prevented by using only one of the two neighboring face's edges that were incident with the face being removed.

4.4.4.2 Special Considerations

Fr **does not always exist.** This happens when a border edge is collapsed. *Fl* update the connectivity of its neighbors is updated. *Fr*, *f1* and *f2* are all set to -1 in the vertex split record, and the number of faces removed that is returned is one.

---

[3] This will not happen at the first level of removals. It will occur when any of the removed face's neighbors are removed.

One of the neighboring faces could not exist. This is the same situation for *fl* and *fr*, and happens if one of the edges not being collapsed on *fl* or *fr* is a border edge. In that case we remove *fl* or *fr* and link the neighbor that does exist to the border. Those neighbors' indices that did not exist are set to -1 in the vertex split record.

Both neighboring faces might not exist. This happens when the face is a corner, and is not allowed, since in order to perform a vertex split later, we will have to know *vl* and *vr*. This information is inferred from the neighborhood. If *fl* has no neighbors, *vl* will not exist. If *fr* has no neighbors, *vr* will not exist. A large number is added to the energy change of the collapse and no modifications to the mesh are made. This effectively eliminates the collapse from consideration until it is updated due to another collapse. A dummy vertex split and a zero for the number of faces removed are returned.

The neighbors of *fl* or *fr* might already be neighbors. This is not allowed, since performing this collapse would make these two triangles adjacent to each other on two sides. The change in energy of this collapse is adjusted like in the corner case, and a similar dummy pair is returned.

4.4.4.3 Replacing the Vertices

When an edge collapse is performed, the instances of *vs* and *vt* need to be replaced with *vu* in all the remaining faces. This is accomplished by going in counterclockwise order from the faces adjacent to *fl* and *fr* on the top and on the bottom. It is assumed that all faces that reference *vs* and *vt* are transitively connected to either the start or the end face. In other words, there are no peninsulas. It is acceptable for there to be a gap in this traversal, which would happen if a border were encountered while going counterclockwise from the start. In that case, the algorithm proceeds clockwise from the finish until it hits the border from the other side.

A similar approach is taken when updating edge collapse information. The difference is that the edge energies and the final vertex positions of the collapse are recalculated.

5 Progressive Mesh

When the simplifier finishes performing edge collapses, it passes the simplification mesh and list of vertex splits to the progressive mesh constructor. Since all the edges are identified when the simplification mesh is constructed and each edge collapse introduces exactly one vertex, the simplification mesh will have had to store exactly (#original vertices + #original edges) vertices. Faces, on the other hand, will have always been a subset of the faces of the original mesh (Figure 13).

```cpp
//allows a face to be removed from the mesh
class PFace : public Face {

        bool active;

};

//Represents a progressive mesh.
class PMesh {

        //increase the resolution of the mesh.
        //returns the number of faces added.
        int next();

        //decrease the resolution of the mesh.
        //returns the number of faces removed.
        int prev();

        //draw the active faces
        void draw();

        //go around old_v_i and replace it with new_v_i
        void replace_corner_vertex(
                    int old_v_i, int new_v_i, int fs_i, int fe_i);

        //a list of all the vertices that could ever be used by the PMesh
        vector<Vertex> vertices;

        //a list of all the faces that could ever be used by the PMesh
        vector<PFace> faces;

        //an iterator pointing to the next vertex split
        unsigned int next_split_i;

        //the refinement operations that can be performed on the mesh.
        vector<Vsplit> vsplits;

};
```

Figure 13: PMesh Class

5.1 Construction

In constructing the progressive mesh, the vertices and faces are reordered in the order that they are introduced to the model, as discussed in [2]. This is accomplished by putting all of vertices and faces from the base mesh at the beginning of the vertex and face arrays. The rest of the faces

and vertices are introduced in the opposite order that they were collapsed, since we are now adding detail to the base model.

The transformation from the original order to the new order is accomplished by keeping two move vectors, one for vertices and one for faces. These vectors specify where an item in the original ordering is moved to in the new ordering. Concretely, `vert_move[i]` specifies that the vertex at index i in the simplification mesh should be moved to index `vert_move[i]` in the progressive mesh's vertex list. This is accomplished by keeping a counter for the faces and the vertices that specifies the next location to move these objects to in their respective lists.

The constructor first goes through the list of the simplification mesh's faces, adds entries to the move vector, and increments the current face counter for all the active faces. For each face that is moved, all of its vertices that have yet to be moved are added to the move vector as well, and the vertex counter is treated similarly.

Next, the list of vertex splits is traversed in reverse order and *fl*, *fr* (if it exists), *vs* and *vt* are marked to be moved to the next available positions. Neither of these faces or vertices will have existed in the mesh. On the other hand, the neighboring faces and vu will already be in the mesh, since otherwise this vertex split would not be legal. These vertex split values are also updated according to the move vectors, making sure that the indices to the faces in the neighborhood point to the relocated positions.

After the move vectors are populated, the actual translocation of the faces and vertices begins. The vertices are copied as specified by the move vector, and faces are also moved, but their neighbor lists are updated to point to the moved faces.

5.2 Vertex Split (next)

The `next()` method of the progressive mesh class performs the next vertex split and returns the number of faces that were added in. It proceeds by making sure that the mesh is not already at the highest level of detail, in which case it returns that zero faces were added. Otherwise, it introduces at least one face.

The new faces will have *vs* and *vt* as two of their vertices. The other vertex, *vl* or *vr*, is inferred from the neighboring faces. All the faces referencing *vu* are updated to point to *vs* or *vt*. Finally, the method increments the current split index and returns the number of faces added.

5.3 Edge Collapse (prev)

The `prev()` method performs an edge collapse. This is the same operation as the edge collapse of the simplification mesh, except that it does not have to worry about edges. It performs a lower bound check for the vertex split index. If we are at the base mesh, no collapse is performed.

6 Results

The simplification algorithm produced a progressive mesh that could be refined view-independently from a single triangle up to the full resolution of the mesh. Below are screenshots of several levels of detail of a 40x40 height field triangulated and simplified. Notice that the peaks and valleys are preserved well at high resolutions (Figure 17, Figure 18, and Figure 19). This is because the edges lying perpendicular to the areas with large slope are considered first since they are shorter.

As more edges are simplified many springs begin to act on the same vertex. This makes the position that optimizes the edge collapse somewhere in the center of all the ancestors. Unfortunately, this makes the features of the model become smoother and more leveled out, whereas a better approximation would keep the peaks and valleys at their extremes (Figure 14, Figure 15, and Figure 16).

One way to address this problem could be to make the spring constants in areas of high curvature larger. This would give preference to these areas in the simplification algorithm and possibly reduce the distortion. This is the way that the borders were preserved. Other methods for making the energy function produced better estimates of the original model are described in future work.
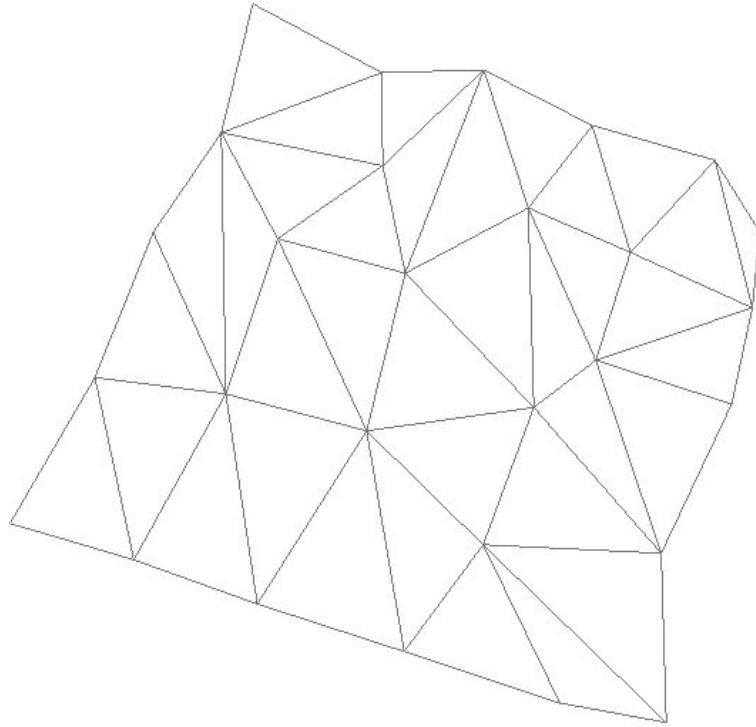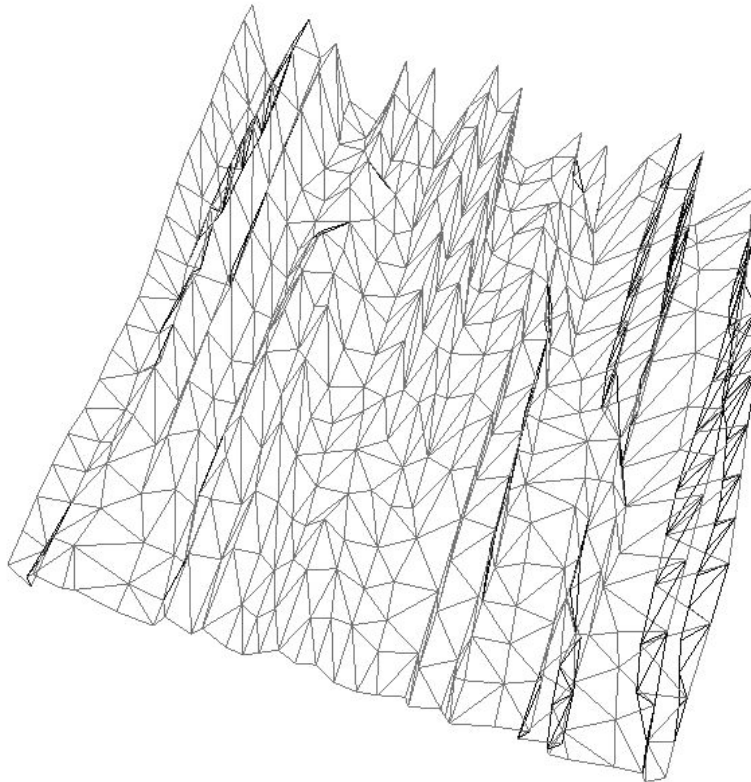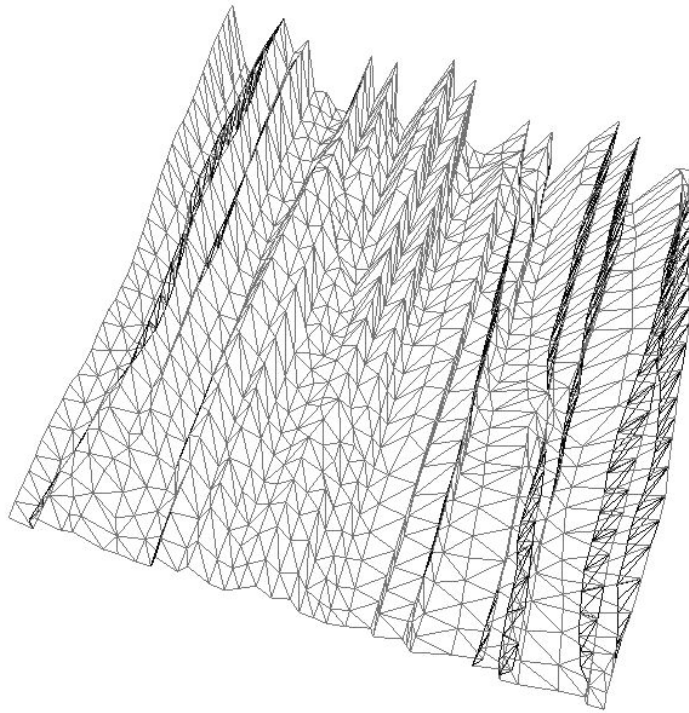
Figure 14: Level 0 Mesh ~35 Faces
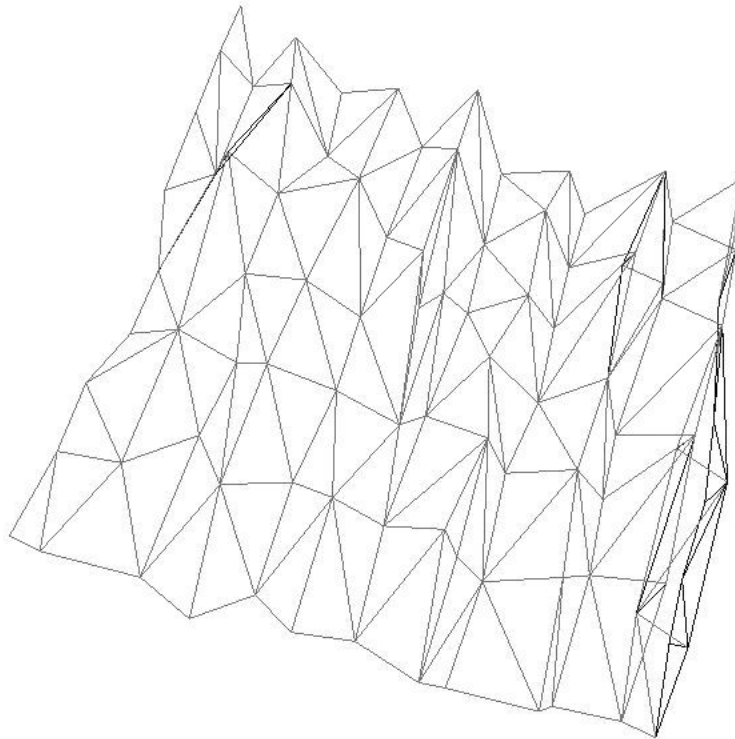
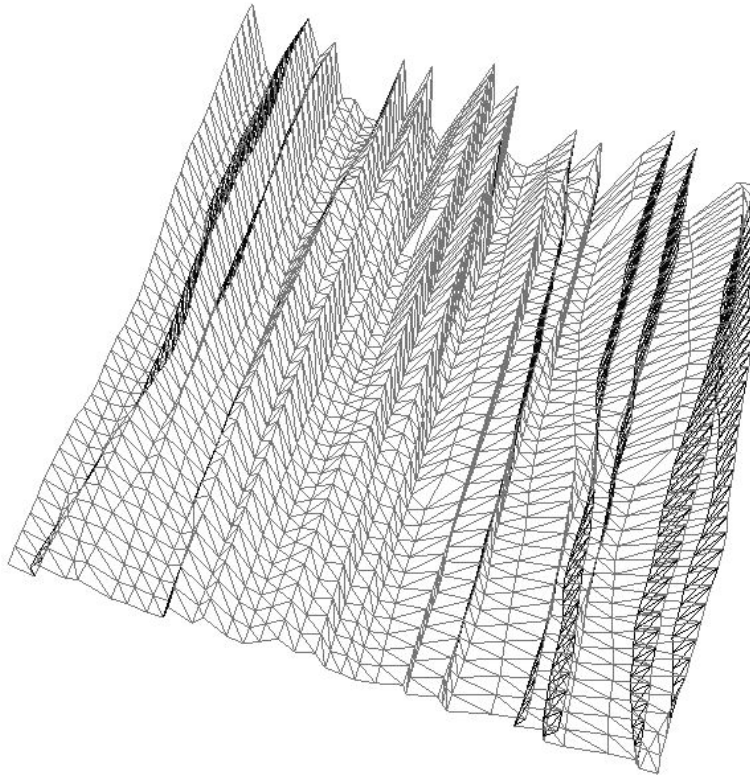**Figure 15: Level 1 Mesh ~ 100 Faces**
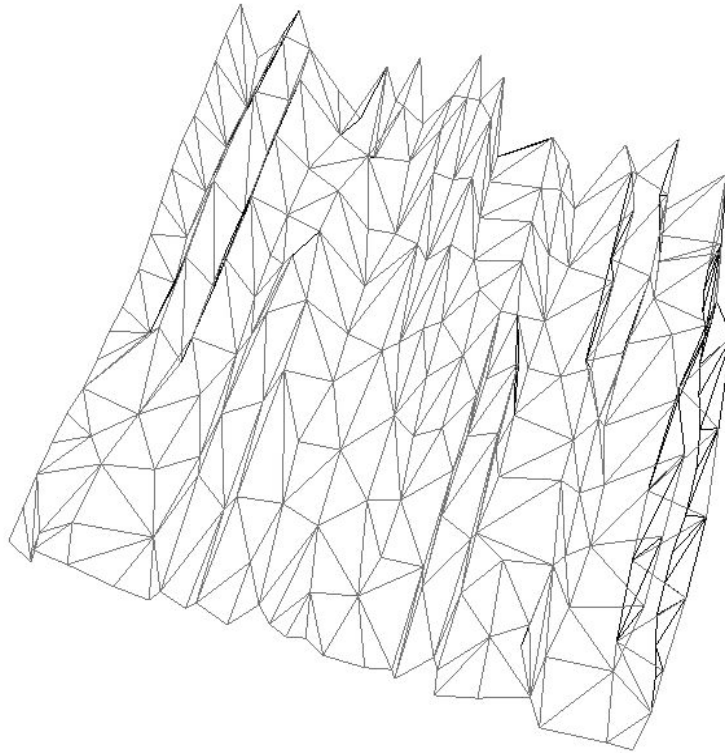
Figure 16: Level 2 Mesh ~ 250 Faces

Figure 17: Level 3 Mesh ~ 600 Faces

31

Figure 18: Level 4 Mesh ~ 1500 Faces

Figure 19: Level 5 Mesh (Original) ~ 3000 Faces

7 Future Work

7.1 Improvements

There are several optimizations that would significantly improve the performance of the current simplification algorithm.

7.1.1 Red-Black Trees

The data structure used to store edge collapses needs to support four operations: one-time insertion of all the collapses; accessing the least expensive edge collapse; updating of individual collapse information; and removal of elements. Currently, using an unsorted array to store the collapses, insertion of all the collapses takes linear time; removal and update take constant time[4]; and accessing the least expensive edge takes linear time, since each element in the array needs to be considered. Because the simplification algorithm repeatedly chooses the least expensive edge, this makes the whole algorithm have quadratic time complexity.

This time complexity could be reduced to n*log(n) by using a red-black tree implemented priority queue, since insertion and deletion are logarithmic complexity and are performed at most the number of edges times.

7.1.2 Anchor Spring Constant Assignment

Currently, only border edges are assigned heavier spring constants. It would make sense to allow similar labeling of other important features of the mesh, such as areas with high curvature. This can be accomplished by adding more predicates on the corners of faces. The predicates cannot be made on the vertices themselves, since there is no direct link between vertices, but there is between faces. Possible criteria could be thresholds for the angles of the edges that are incident on that corner. If the predicate evaluates to true, then an associated value for the importance of the feature can either be assigned or added to the current spring constant at that vertex.

7.1.3 Energy Functions

The energy function used in this project was chosen due to its simplicity. It preserves details well in the early stages of the simplification, when there are but a few anchor vertices, since the

---

[4] Removal is accomplished by setting the active field of the edge collapse to false. Updating an element requires dereferencing the array at a certain index.

springs make the collapses favor edges closer to the original geometry. After many collapses have been performed, though, too many forces end up acting on a single vertex, which causes the model to smooth out.

Even though this can be acceptable for modeling objects that are far away, it is inadequate for closer use. Since part of the criteria for simplification is the processing power of the device being used to render the model, the simplification algorithm will need to be able to produce good approximations of the models even at low resolutions of the mesh.

Mesh Optimization [1] and Progressive Meshes [2] have already addressed this problem. In future work, the energy function from these papers should be used. It would be an interesting exercise, though, to see whether the vertex spring energy function used in this project could be adjusted to attain better low-resolution approximations since it is mathematically much simpler.

Another possible improvement to the energy function is to add a term that attempts to preserve symmetry within the object. This is not as important with terrains, but when modeling man-made objects, the low-level approximations produced by the progressive mesh simplification algorithm do not seem to account for symmetry, as can be seen in the low resolution illustrations of the Figure 20 and Figure 21.
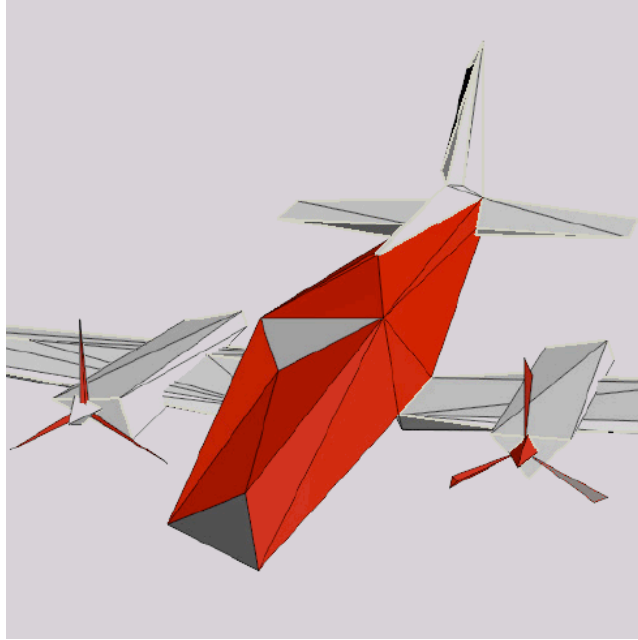
Figure 20: 150 Face Airplane from Progressive Meshes [2]
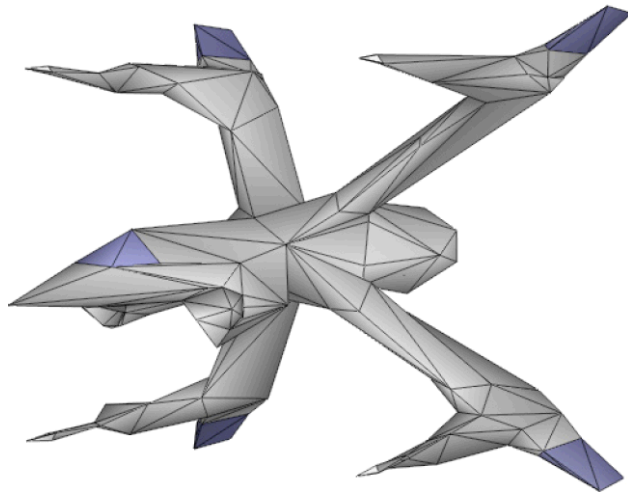
Figure 21: 444 Face Spaceship from Efficient Implementation of Progressive Meshes [3]

7.2 Expansions

Several expansions can be envisioned that use the current implementation as a subsystem. Some of them are listed below.

7.2.1 View-Dependent Refinement

The current progressive mesh data structure is set up for allowing view-dependent refinement as described in [5]. The neighborhood information contained in the vertex split records is enough to allow the construction of a vertex refinement hierarchy. Combined with refinement criteria based

on camera distance from the vertices, possibly optimized using spatial-partitioning data structures and algorithms, the progressive mesh could be used to generate a view-dependent, continuous level of detail mesh.

7.2.2 Procedural Planet Generation

There are many more parts of the procedural planet generation project that need to be implemented. There will need to be a fractal terrain generator that can generate models at varying levels of detail. Architecture for a single user and multi-user exploration of the same planet will need to be created. Some of these concepts may be borrowed from Virtutopia (discussed below).

The same environments could potentially be used for different applications with varying rendering expectations. In a video game, interactive frame rates are important, while rendering an environment for use in a movie would also desire information about special effects to make the environment seem more realistic. This interface to different renderers is possible if standardized data representation and distribution models are created. All the information needed for the highest supported renderer could be stored with the model, and subsets of these properties could be transmitted as required by the more-constrained renderers. For example, a renderer that will not be doing ray tracing will not need to know about the speed of light constants of transparent materials. At the same time, this extra information might be vital in achieving a certain mood in a movie scene.

For the interactive part of procedural planet generation, it might be appropriate to modify the currently generated model. An example of this would be chopping down a forest to build the city or creating a crater due to meteor impact. A system for allowing these changes will also need to be developed.

7.2.3 Virtutopia

Progressive meshes could be used in the Virtutopia [11] project in a similar manner as described for the Procedural Planet Generation project. Virtutopia is meant to be "a gateway into many various virtual environments where thousands of users can interact on artwork, solve engineering problems, or lose themselves in an online video game" [11]. Such a gateway will need an

efficient way of rendering objects on many different platforms of varying processing power. Progressive meshes can provide this mechanism.

Progressive meshes could be incorporated into the Virtutopia server-side object repository by providing a progressive mesh transmission mechanism between the Virtutopia server and Virtutopia clients. The server would contain a precomputed progressive mesh representation of the objects that would be displayed in the world and could send varying levels of detail representations of the object, based on factors like client frame rate, distance from the camera, and current server bandwidth use.

Transmission would proceed as described in the Progressive Meshes [2] and Efficient Implementation of Progressive Meshes [3]. In the general case, the base mesh could be sent from the server to the client where it could be cached and displayed in the renderer. Requests for refinements could be made based on different criteria, including distance from the camera and high frame rates. The client could update the mesh view-dependently or independently and ask the renderer to create the geomorph between the two levels of detail that are being switched.

8 References

1. Hoppe, H., DeRose, T., Duchamp, T., McDonald, J., and Stuetzle, W. *Mesh optimization*. In Computer Graphics Proceedings, Annual Conference series, ACM SIGGRAPH, pages 19--26. 1993

2. Hoppe, H. *Progressive meshes*. Computer Graphics (SIGGRAPH '96 Proceedings), pages 99--108, 1996.

3. Hoppe, H. *Efficient Implementation of Progressive Meshes*. Computers & Graphics Vol. 22 No. 1, 27-36, 1998.

4. Xia, J., and Varshney, A. *Dynamic view-dependent simplification for polygonal models*. In Visualization '96 Proceedings, IEEE, pp. 327--334. 1996

5. Hoppe, H. *View-dependent refinement of progressive meshes*. In SIGGRAPH '97, pages 189--198. 1997

6. Hoppe, H., *Smooth view-dependent level-of-detail control and its application to terrain rendering*, Proc. of IEEE Visualization '98, pp 35-42; 516, 1998.

7. Duchaineau, M., Wolinsky, M., Sigeti, D.E., Miller, M.C., Aldrich, C., Mineev--Weinstein, M.B.: ROAMing *Terrain: Real--time optimally adapting meshes*, in Proc. Visualization '97, 1997.

8. Brown, P. "Selective Mesh Refinement for Rendering." *Selective Mesh Refinement for Rendering*. February 1998. University of Cambridge. April 2007. <http://www.cl.cam.ac.uk/research/rainbow/publications/pjcb/thesis/>.

9. Ebert, D., Musgrave, F., Peachey, D., Perlin, K., Worley, S. *Texturing and modeling a procedural approach Third edition*, Morgan Kaufmann Publishers, 2003

10. Martz, P. "Generating Random Fractal Terrain" *Generating Random Fractal Terrain*, 1997.   http://www.gameprogrammer.com/fractal.html

11. Janc, A., Jarmak, M., Kolk, S., Martin, R., Pedrotti, O. *Virtutopia: A framework for virtual invironments*, WPI Electronic Projects Collection, 2007

12. Hill, F.S., *Computer Graphics Using Open GL* Second Edition, 2001, Prentice Hall

13. "Terragen," *Terragen*, April 2007, http://www.planetside.co.uk/terragen/

14. Derek Radtke. Please contact the author for more information.

15. "Algorithmic Botany: Publications," *BMV Publications* May 2007, University of Calgary http://algorithmicbotany.org/papers/

16. Lorenz, W. "Fractals and Fractal Architecture", *Fractals and Fractal Architecture*, May 2007 http://www.iemar.tuwien.ac.at/modul23/Fractals/subpages/62Cityplanning.html

9 Tools Use

- GNU C++ compiler – This project was implemented using C++.
- Standard Template Library - many of the container classes and algorithms, such as finding the minimum element, used in this project were from the standard template library.
- Theilana - the three dimensional mathematics libraries were taken from Derek Radtke's Theilana physics engine.
- OpenGL - is an open source reference Library and was used to render the meshes.
- Terragen - is a free for noncommercial use fractal terrain generator used to generate the height fields for the model simplified in this project.
- Eclipse - that eclipse C++ development interface was used for the programming.