Real-Time Continuous Level-of-Detail Terrain Rendering With Nested Splitting Space

Finalist: Yuanchen Zhu
Intel ISEF 2001 Project ID: CS012

***Abstract*— Real-time visualization of large-scale terrain models requires complex continuous level-of-detail (LOD) schemes to reduce the prohibitively large geometry complexity of natural terrain to an acceptable level while maintaining high image quality. This paper presents new algorithms and data structures to solve this problem.**

Our solution centers on computing independent per-vertex error bounds, which we call nested splitting space, for vertices in the height field according to screen-space error metrics and vertex dependence hierarchy. Based on this, a framework for circular array based, frame-coherent mesh refinement and reduction is devised. A simple mechanism to support vertex morphing requiring zero storage overhead is proposed, and methods for procedural generation of error bounds are given, providing support for real-time on-demand procedural details generation.

The algorithm accomplishes LOD updates in time proportional to the number of LOD changes needed for each frame. In our sample implementation, updating the mesh takes less than one-tenth of the time spent on processing and rendering the mesh by the graphics hardware.

The contributions of this paper are twofold: First, the idea of computing independent per-vertex error bounds, which can also be utilized by other LOD algorithms, is presented; Secondly, accompanying schemes of frame-coherence optimization, vertex morphing support, and procedural error bounds generation for continuous LOD rendering of terrain, which outperform existing methods in terms of efficiency, storage requirement, supported features and ease of implementation, are proposed.

***Index Terms*— level-of-detail, view-dependent, terrain rendering.**

I. Introduction

REAL-TIME visualization of large-scale terrain models is at the core of display systems for many interactive applications such as geography information systems (GIS), flight simulators, and electronic games with outdoor 3D scenes. In order to accommodate the great geometry complexity of natural terrain while still maintaining interactive frame-rates on current graphics hardware, algorithms for view-dependent level-of-detail (LOD) terrain triangulation are needed.

Typical terrain models have both rough and relatively flat neighborhoods. Further more, distant areas affect screen image quality to a lesser degree than near-by fields because of the perspective nature of 3d-to-2d projection. Hence, efficient terrain LOD schemes can take both into account and provide different tessellation levels for different parts of the terrain. This also means that the terrain has to be dynamically triangulated during run-time, as the distance from the viewpoint to a section of the terrain changes as the viewer moves. This paper presents a new method for performing view-dependent LOD meshing for terrains. An example frame generated by our prototype implementation, with and without wire-frame shown, is shown in Fig. 1 and Fig. 2.

Email: yuanchen_zhu@computer.org

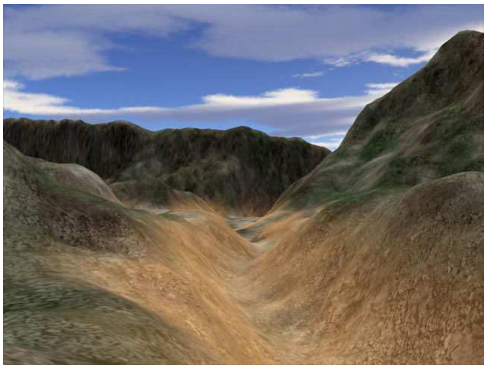

Fig. 1. Example frame generated by our algorithm.

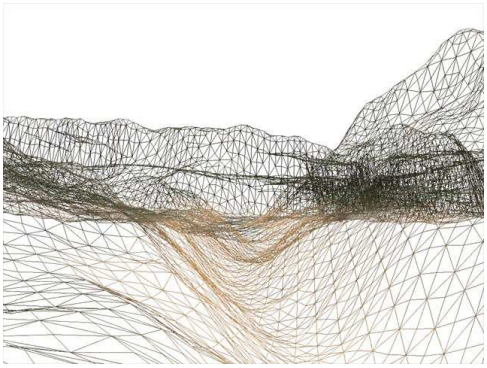

Fig. 2. Same as above, but shows underlying meshes.

Our solution centers on the use of independent per-vertex error bounds, which we dubbed *nested splitting space* (NSS). Terrain LOD algorithms need to address quite a few problems, including LOD selection, triangulation, elimination of discontinuities at boundaries of different sized triangles, and on-demand procedural details generation. The unique aspect of our algorithm lies in that by adjusting and manipulating the error bounds for each vertex directly, we can solve all the problems previously mentioned in a neat and elegant manner, eliminating the overhead of accessing or modifying mesh adjacency, as in the case of several existing algorithms ([1]–[4]). Based on this, a framework for circular array based,

frame-coherent mesh refining and coarsening is devised. We also provide simple mechanisms to support *vertex morphing*, procedural NSS generation, and LOD accuracy adjustment. The testing results of our prototype implementation are encouraging.

Our algorithm can be characterized by the following set of features:

- **Fast triangulation.** By utilizing frame coherence and scheduling lazy LOD evaluation through two distance based circular arrays, we build the triangulation by incrementally updating the mesh for each frame. Hence, the algorithm can accomplish LOD update in time proportional to the number of LOD changes needed for each frame, which typically involves a small fraction of the total mesh. Direct support for partial evaluation to ensure strict frame rates is also provided.
- **High image quality.** Approximation of projected error is used in the prototype implementation as the view-dependent metric for LOD selection. Hence, a screen space error upper bound can be specified and the image errors introduced by detail reduction are guaranteed to be below the bound.
- **Flexible view-independent metrics.** View-independent metrics are incorporated into NSS generation and all kinds of metrics can be used to suit different application needs. Examples are terrain roughness, potential visibility (i.e. above/below sea surface), and topology features.
- **Smooth and continuous LOD transition.** Virtually unnoticeable LOD transition can be achieved by specifying a very small screen space error upper bound and allowing the polygon number to surge. However hardware and polygon budget limitations sometimes preclude this practice. In such circumstances the built-in vertex morphing capability can greatly reduce the visual artifact of vertex popping caused by underlying mesh changes.
- **Right-isosceles triangle based meshes.** The algorithm outputs right-isosceles triangle based triangle meshes. Thus, the potential numerical problems of very thin or degenerated triangles are avoided.
- **Low memory requirement.** In addition to height values, one additional NSS parameter needs to be stored for each vertex in the terrain. Depending on the required precision, this parameter can be packed into one or two bytes. Apart from this, the algorithm takes up additional memory space proportional to the size of output mesh, which is negligible compared to the size of the whole terrain.
- **Easy support for hierarchy based terrain-storing models.** In addition to the traditional *digital elevation map* (DEM) representation, terrain storing models such as *adaptive quadtree* [3] can be used in order to further reduce memory consumption by taking advantage of the relative roughness of the terrain. The hierarchical nature of such models makes it expensive, in terms of either time or space, to access sibling nodes. Since our algorithm does not require additional mesh modification to ensure mesh continuity, etc., it provides a simpler and more efficient solution for performing LOD triangulation for such models.
- **"Unlimited" levels of detail.** We also provide methods for procedurally generating NSS's. On-demand detail generation can be incorporated in this way, providing virtually "unlimited" levels of detail.
- **Great scalability for current and future graphics hardware.** With the latest generation of powerful *graphics processing units* (GPU), the CPUs can hardly keep up performing fine-grained LOD evaluation for every triangle to be rendered. We provide a subdivision-based scheme that allows us to make compromises between CPU usage and LOD selection accuracy. Hence, the CPU can spend less time doing LOD computation by just dumping more triangles to the hungry GPU. This feature provides great scalability for current and future hardware. Results from our prototype implementation clearly verify this.

II. Related Work

Heckbert and Garland give a general survey of multiresolution modeling [5]. They also present more specific surveys on simplification and approximation of polygonal surfaces [6]. A critical survey of various multiresolution models for topographic surfaces can be found in work by Floriani et al [7]. Evans et al. [8] have described the *right triangular irregular network* (RTIN), and an accompanying hierarchical structure. Many algorithms, such as [1], [2], [9] and ours, produce RTINs as output triangulations.

Much of the earlier work on terrain LOD algorithms concentrate on discrete multiresolution *triangulated irregular networks* (TIN) modeling. Several versions of the same landscape tessellated to different detail levels are produced with expensive off-line preprocessing, and then stored in some way. At run-time, they are dynamically combined to provide different tessellation levels for different areas of the terrain. Examples are [10]–[12]. However, this kind of schemes has the inherent disadvantage of discreteness. Fine-grained LOD refinement and coarsening would not be possible unless a large number of TIN models are stored, which burns memory. More over, the irregularity of TINs hampers fast accessing and querying of data sets, and dynamic terrain deformation. Complex data structures are needed to maintain such models, introducing further memory overhead. All of these disadvantages mark multiresolution TIN based LOD schemes less fit for real-time terrain visualization, although they do suit other tasks well, such as geographic analysis.

Hoppe et al. [4], [13], [14] present a mesh representation called *progressive mesh* (PM) that, given an arbitrary input mesh, stores a base mesh and a sequence of vertex split records. They provide methods for performing view-dependent mesh refinement on PMs. However, although their framework has the nice property of supporting arbitrary mesh topology, the algorithm is usually no match of algorithms that are specially designed for terrains. Moreover, as edge collapses are more naturally performed in a predetermined order, the method requires complex data structure and bookkeeping to perform view-dependent mesh refining.

Lindstrom et al. [1] use bottom-up triangle fusion operations to perform fine-grained simplification over height-maps. As the bottom-up process tends to limit performance, they introduce an additional optimization that performs coarse simplification of the surface mesh. Vertex morphing is not supported, and memory requirement is higher than ours. Only height-map based storing models are supported, as vertex simplification works bottom-up from a rectangular block of evenly spaced vertices. Procedural details generation would be difficult to incorporate for the same reason. The algorithm also requires some additional overhead to maintain mesh continuity at block boundaries. A recursive stripping algorithm producing generalized triangle strips is provided, but no other hardware consideration is given.

Duchaineau et al. [2] use two priority queues to drive incremental fine-grained mesh refinement and reduction in their *real-time optimally adapting meshes* (ROAM) algorithm, taking advantage of frame-to-frame coherence. A *binary triangle tree* (BTT), which is basically the same hierarchy described in [8], is maintained explicitly to define the triangulation and additional data structure overhead is needed to perform force-splits, which are necessary to ensure mesh continuity. Vertex morphing can be supported but no detailed description of it is given. No consideration is given for procedural detail generation. A lazy evaluation scheme that uses velocity upper bound of the viewpoint to maintain evaluation lists for future frames is mentioned, but this is less efficient compared to our distance based lazy evaluation, as it does not take into account occasions when the viewer is not moving at its top speed. Blow [15] has proposed maintaining a hierarchy of activation volumes instead of two priority queues. There is a similarity between his concept of activation volume and our nested splitting space, though we place emphasis on entirely different areas: He maintains a hierarchy of spherical error spaces to perform minimal LOD evaluation; We preprocess ellipsoid error spaces to abstract everything: i.e. LOD selection, cracks elimination, vertex morphing, down to a distance check and an interpolation, and rely on two linear distance based tables to perform minimal LOD evaluation. ROAM and Blow's alternative scheme have not investigated the problem of LOD accuracy adjustment, with the help of which our algorithm is able to earn a high performance boost.

Röttger et al. [9] present a compact algorithm for LOD triangulation of height fields. The memory requirement is low and they use a similar triangulation method as ours, based on *restricted quadtree* [7]. However they do not realize nor take advantage of the potential hardware scalability this method can provide through subdivision. They use a height-map sized matrix to mark the traversed quadtree nodes, precluding support for non height-map based terrain storing models and procedural details generation. Vertex morphing is supported. A recent presentation by Louis Castle et al. [16] makes several improvements over the algorithm, including simplified quadtree subdivision and linked quadtree nodes storing which address procedural detail generation. Neither algorithms take advantage of frame-coherence and have to re-triangulate the terrain for each frame. Effort is put into devising a LOD metric that would automatically ensure mesh continuity. They do not realize the more simple and flexible approach of directly enlarging vertex error spaces, and the resulting metric is based on the ratio between area size and distance, perturbed by a roughness factor, which still need to be propagated bottom up during preprocessing. This metric only gives lossy estimate of screen-space error.

Boer [17] introduces *geo-mipmap*, a method that puts special emphasis on tailoring for modern graphic hardware and minimizing CPU usage. Initially, a terrain is divided into blocks of a predetermined size. For each block, equally spaced vertex grids of different resolutions are computed and stored in *static vertex-buffers* [18]. At run-time, vertex grids of different resolutions are selected for different blocks, and blocks within the view frustum are stitched together to form the triangulation. At boundaries where grids of different resolution met, corresponding vertices in the grids of higher resolution are discarded, and this can be achieved by modifying the *index buffer* [18]. Meshes produced by such a process is usually far from optimal and fine-grained LOD refinement and reduction would not be possible unless a small block size is used, in which case CPU usage would soar greatly. The memory requirement is high since multiple copies of vertices are maintained and each vertex must be stored in the graphic card's native format, which can be ten times as large as a usual height-map. The paper mentioned the possibility of tri-linear filtering geo-mipmaps to support vertex morphing, but no hardware supports this kind of operation and doing so on the CPU would be expensive and completely overturns its initial goals of "being hardware friendliness". Procedural details generation is not possible within their framework and frame-coherence is not utilized. In practice, our prototype achieves much higher speed than reported in their paper, which we shall describe in more details in Section X.

III. Overview

In the rest of this paper, we will explain various parts of our algorithm: the triangulation process, the nested splitting space, the frame coherence optimization, and several additional improvements.

The output mesh consists of differently sized right-isosceles triangles. To ensure mesh continuity, *vertex dependence* [1] has been used to describe the relationship between vertices in the terrain. Various triangulation schemes for outputting these kinds of meshes have been proposed ([1]–[3], [9]), and the one used in this algorithm is based on triangle fan generation for nodes in a restricted quadtree structure. We explain it in Section IV.

Section V goes on to describe the concept of nested splitting space (NSS), which is used for LOD evaluation. We first show a NSS model devised according to perspective projection, which approximates the geometry error's projected length. Then its side effects are discussed and an improved model that provides faster computing and better image quality is presented. Since the NSS's are precomputed and preprocessed to ensure mesh continuity, the need and overhead of mesh modification to fix T-vertices during subsequent frames are avoided.

To compute NSS's we must provide view-independent error for each vertex. Various possible metrics are listed and discussed in Section VI, including vertex interpolation error, nested error bound, potential visibility, terrain topology and so on.

In real-world applications, the triangulations of adjacent frames often differ little. Hence, we can build this frame's triangulation by taking the one left by the last frame as input and only applying changes where needed. In practice this scheme is often much more efficient then rebuilding the mesh. In Section VII, we present a distance based frame-coherence optimization that utilizes two circular arrays to schedule lazy evaluations for quadtree nodes.

In Section VIII, we present several improvements to the algorithm, including dynamic mesh resolution adjustment, procedural NSS generation, vertex morphing, LOD accuracy adjustment, and incremental view-frustum culling. These improvements incorporate additional features into the algorithm and allow greatly improved performance.

Testing results from real-world experiments accompanied with discussions are presented in Section X. We present the conclusions in Section XI.

IV. Triangulation

The base of the triangulation process is a restricted quadtree structure, where adjacent nodes have a maximum depth difference of 1. Throughout this paper, the terrain is assumed to have the same width and length. Each node of the quadtree represents a square region in the terrain. The relationship between a father node and its four children is that the four quadrants represented by the children nodes make up a bigger quadrant represented by the father node. The root node covers the entire terrain. See Fig. 3 for an illustration.

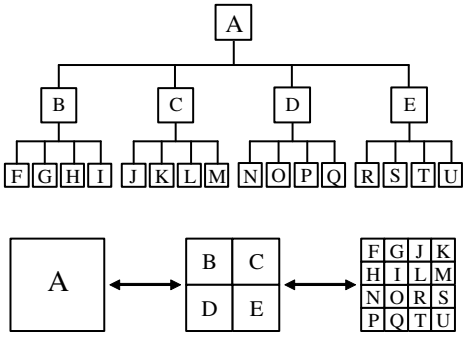

Fig. 3. Illustration of a three-level quadtree structure.

For each quadtree node, nine vertices in the covered quadrant are accessible (see Fig. 4): the center vertex, the four side vertices, and the four corner vertices. Therefore every level of quadtree nodes introduces new details, and LOD selection can be accomplished by starting from the root node and recursively checking to see whether the current node is of sufficient LOD levels, and if not, refine it into four children nodes and repeat this process. For each vertex in the terrain, a view-dependent Boolean function $\mathrm{enabled}(v)$ is defined to indicate whether or not vertex v belongs to the target triangulation:

$$\mathrm{enabled}(v) = \begin{cases} \text{true} & \text{if } v \text{ belongs to the triangulation,} \\ \text{false} & \text{otherwise.} \end{cases}$$

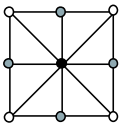

Fig. 4. Illustration of the quadtree node vertices: the center vertex is shown in black, the side vertices in gray, and the corner vertices in white.

When $\mathrm{enabled}(v)$ equals to true, we declare v to be *enabled*, otherwise it is *disabled*. During quadtree traversing, we stop refining when the node's center vertex is disabled. By generating full or partial triangle fans for quadtree nodes that either have unvisited children or do not have any children, the final mesh can be constructed. This process is depicted in Fig. 5.

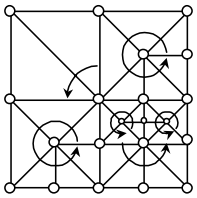

Fig. 5. Triangle fans based mesh, with enabled vertices shown.

Since adjacent square fields are triangulated using the same vertex at their boundary, clearly no cracks will be introduced as long as the enabled value for vertices in the terrain satisfies the so-called vertex dependency [1], which is shown in Fig. 6. Namely, for a given frame and a quadtree node:

- The center vertex should be enabled if any of the side vertices are enabled.
- If one of its child nodes has an enabled center vertex, the two side vertices that made up the child node's corner vertices should be enabled.

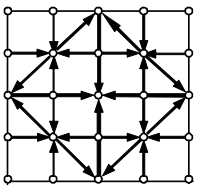

Fig. 6. Vertex dependency.

V. Nested Splitting Space

For a vertex v, the collection of viewpoint positions that causes the view-dependent $\mathrm{enabled}(v)$ to be true forms its *nested splitting space* (NSS). When the viewer moves into the NSS, the view-dependent error introduced by not including v in the final triangulation is so great that v has to be included, i.e. $\mathrm{enabled}(v) = \text{true}$.

Suppose the 3d to 2d perspective projections is of the form $(\lambda_x x/z, \lambda_y y/z)$, where (x, y, z) denotes the camera-space coordinates, for v we can approximate its projected error E by

$$E = \frac{e}{l} \sin^2(\theta) \max\{\lambda_x, \lambda_y\} \tag{1}$$

Where e is the length of its geometry error segment, l is the length of the viewing vector projected on to the horizontal plane, and θ denotes the angle between the viewing vector and the error segment, as shown in Fig. 7 (For simplicity, we assume the error segment to be vertical).

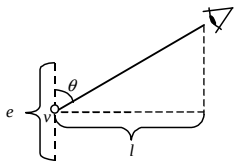

Fig. 7. Illustration of variables in (1).

It is easy to see that for a fixed screen space error upper bound E_m, (1) gives us a circular torus shaped NSS, whose cross section is two identical circles with a diameter of $e \cdot \max\{\lambda_x, \lambda_y\}/E_m$. Such a shape, however, introduces a lot of error in the depth direction when the viewer is above the terrain looking down, i.e. when θ is small, and potentially affects fog calculation and causes inaccuracy in visibility determination. Another problem is that it is complex to manipulate these NSS's to ensure vertex dependency.

In order to simplify evaluation and reduce the artifacts caused by depth inaccuracy, we use an ellipsoid NSS instead. It is a normal sphere squashed to half its original size in the vertical direction with a radius of $e \cdot \max\{\lambda_x, \lambda_y\}/E_m$. Hence, we have:

$$\text{enabled}(v) = \begin{cases} \text{true} & f(\mathbf{v}, \mathbf{v_0}) \le 0, \\ \text{false} & f(\mathbf{v}, \mathbf{v_0}) > 0, \end{cases}$$

where $\mathbf{v_0}$ is the viewpoint and

$$f(\mathbf{a}, \mathbf{b}) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2 + [2(a_z - b_z)]^2} - r \tag{2}$$

where

$$r = \frac{e \cdot \max\{\lambda_x, \lambda_y\}}{E_m} \tag{3}$$

Notice that we double the vertical component in the distance formula. In other words, instead of squashing the sphere, we stretch the coordinate system in z direction. This trick enables us to treat NSS as spheres, as long as all vertical components of other world-space vector quantities, including viewer movement, are doubled. For the rest of the paper, we always work on "sphere shaped" NSS and will take it for granted that all world-space vector quantities have been converted. In practice, this NSS model provides excellent image quality and quick computing. Its cross section and that of the torus shaped NSS are shown in Fig. 8.

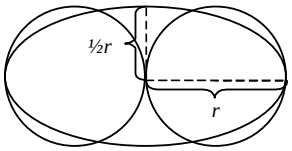

Fig. 8. Cross sections of the two NSS models.

It is apparent that for each vertex, only one NSS parameter, r, needs to be stored. Depending on the required accuracy, r can usually be squeezed into one or two bytes. Once the NSS's are computed, they must be enlarged bottom up according to vertex dependencies (Section IV). This, of course, can be done in the preprocessing stage. Given two vertices, a and b, and their corresponding NSS radius r_a and r_b, if b's NSS is to contain a's NSS, r_b is to be updated as:

$$r_b = \max\left\{ r_b, r_a + \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2 + [2(a_z - b_z)]^2} \right\}$$

One advantage of our NSS methodology lies in that after the preprocessing, no efforts are needed to ensure mesh continuity and avoid cracks caused by T-vertices during triangulation of subsequent frames. Since no neighbor nodes need to be accessed or modified during triangulation, this method can also be used to simplify and accelerate LOD meshing for hierarchy-based terrain storing models such as adaptive quadtree. An example triangulation for a given NSS configuration is shown in Fig. 9.

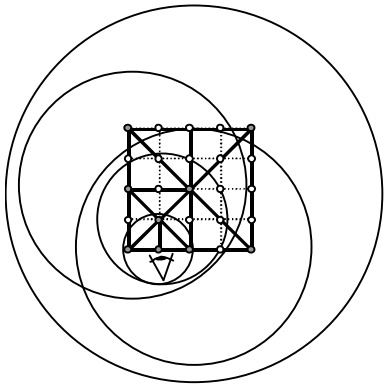

Fig. 9. Illustration of an example triangulation for a given NSS configuration, with NSS's of enabled vertices shown. The enabled vertices are shown in gray. Notice that we do not show the NSS's of the four corner vertices, since they are always enabled.

VI. View-Independent Error Metrics

Different applications would require different view-independent error metrics to get the desired geometry error segment mentioned in the previous section. Listed below are several possible metrics that can be directly used for NSS computing:

- **Vertex interpolation error.** When a vertex is enabled, the mesh changes shape, and the amount of change is the difference between the vertex's height and the original height of the mesh at the vertex's position, as shown in Fig. 10. The displacement vector can be used as the error segment. While only level-to-level error is incorporated, it provides good enough image quality in most cases. [1] use it as well.

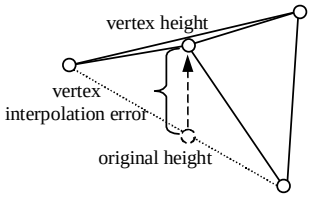

Fig. 10. Illustration of vertex interpolation error.

- **Nested error bound.** By computing tight bounds that encapsulate the vertex displacement vector and the bounds of all vertices that depend on it in the vertex dependency hierarchy, we can estimate the overall geometry error introduced for a given triangulation. [2] computes this kind of bounds for nodes in a binary triangle tree.
- **Potential visibility.** Typical terrains have areas that are potentially invisible to the viewer. For example, for applications with open sea areas, the seabed is obscured by the sea surface from the viewer. The error segments of vertices in these areas can be scaled down in proportion to their potential visibility.
- **Topology features.** Different topology features often use different rendering methods. At the boundaries where the different types of topology meet, it is often useful to manually increase the geometry error segment to allow more accurate tessellation in that area. A good example is in the case of seashores. The water surface is usually drawn as another translucent surface layer over the terrain. With the tides rising and falling all the time, the polygonal silhouette between the water surface and terrain is easily spotted unless higher tessellation level is used for the terrain.
- **Ground object positioning.** For areas where ground object such as buildings and trees are situated, the tessellation level can be tuned up so that more accurate object-landscape interaction can be observed.
- **Data availability.** Some terrain storage models such as adaptive quadtree discard details for flat areas. The geometry error segment for these areas can be manually set to zero.
- **Atmospheric obscureness.** For deep valleys with fog decreasing visibility, the geometry error segment can be tuned down.

Clearly, additional metrics to suit application's requirement would not be difficult to incorporate.

VII. Frame-Coherence Optimization

During interactive terrain visualization, the user typically moves across the terrain in a continuous fashion. In such cases, the difference between triangulations of subsequent frames is often fairly small compared to the size of the whole triangulation. Hence, extra performance can be gained by utilizing this coherence and updating the existing mesh where needed rather than rebuilding the triangulation for each frame.

In order to determine the vertices whose $\mathrm{enabled}()$ values have changed, all vertices in the current triangulation would need to be reevaluated to determine their new viewer-in-NSS states. However, by carefully observing the NSS model, it can be found that: A vertex will only need to be evaluated when the length of the viewer's movement path has exceeded the distance between the viewpoint and the NSS boundary, which we denote as d, during the previous evaluation. By pushing vertices to the back of the evaluation queue according to their d value, we can implement some sort of lazy evaluation to optimize the algorithm for frame-coherence. Here d is just $f(\mathbf{v}, \mathbf{v_0})$ as in (2) and $\mathbf{v_0}$ is the viewpoint. A positive d indicates that the viewpoint is outside the NSS, while a negative or zero d denotes that the viewpoint is inside the NSS or on its boundary.

We maintain two circular arrays, T_s and T_m. Each entry in the table is a linked list of quadtree nodes. T_s is used to store quadtree nodes that are waiting to be split (refined), and T_m for nodes that have already been split and, hence, are waiting to be merged (coarsened). For nodes that have split children, they do not need to be moved into the T_m until their split children nodes are merged. The same rule applies to nodes whose parents are still waiting to be split.

T_s and T_m act as two circular arrays and each has an index pointer, denoted as I_s and I_m. Initially, T_m is to $\emptyset$, and T_s only contains the root node. Then, during each frame, I_s and I_m are incremented by a specific amount depending on the distance of the viewpoint's movement from last frame, and rounded up to the start of the arrays if the maximal size is reached. Then all nodes belonging to entries covered by the gap between the old pointer and new pointer are taken out and evaluated.

The evaluation process consists of a recursive merging checks for nodes in T_m and recursive splitting checks for nodes in T_s, as given in Table I.

A brief description in English is as following: For a node in T_m, if it should not be merged, it is reinserted into T_m. Otherwise first its children are deleted. Next, itself is moved from T_m to T_s. Finally its father, if not already in T_m, is inserted into T_m. For a node in T_s, if it should not be split, it is reinserted into T_s. Otherwise, first its father, if in T_m, is removed from T_m, though not deleted. Next itself is moved form T_s to T_m. Finally all its children are created and inserted into T_s.

For a node to be inserted into T_m, the index of the destination T_m entry is calculated as

$$I_m \leftarrow \mathrm{clamp}\left(I_m + \lceil -d\delta_m \rceil, 0, L_m - 1\right) \bmod L_m. \quad (4)$$

For a node to be inserted into T_s, the index of the destination T_s entry is given by

$$I_s \leftarrow \mathrm{clamp}\left(I_s + \lceil d\delta_s \rceil, 0, L_s - 1\right) \bmod L_s. \quad (5)$$

TABLE I

PSEUDO-CODE FOR RECURSIVE MERGING AND SPLITTING OF NODES.

```
RecMerge(q)
 1  d ← f(q's center vertex)
 2  if d ≤ 0 then
 3     if q has split children or q is already in T_m then
 4        return
 5  else
 6     Recursively remove q's children
 7     Insert q into T_s
 8     if qhas a father node then
 9        RecMerge(q's father)

RecSplit(q)
 1  d ← f(q's center vertex)
 2  if d > 0 then
 3     if q is already in T_s then
 4        return
 5     Insert q into T_s
 6  else
 7     if q has a father node and q's father is in T_m then
 8        Remove q's father from T_m
 9        Insert q into T_m
10        Create q's four children
11        for each c ∈ q's children nodes do
12           RecSplit(c)
```

In (4) and (5), d is the signed distance between the viewpoint and the boundary of the NSS of the node's center vertex. δ_m and δ_s are non zero distance factors for the two circular arrays to adjust the ratio between world unit and table sizes. L_m and L_s are the table sizes.

This scheme can be modified to allow partial evaluations. When the user is moving very fast, the amount of evaluation needed increases. However, to satisfy strict frame-rates, we stop evaluating when the allotted time is to expire. As long as we finish up the nodes linked in the current table entry, mesh continuity will still be maintained. In other words, we have a pseudo-viewpoint that follows the actual viewpoint. We use the pseudo-viewpoint as our evaluation criteria and gave it a velocity upper bound to limit the amount of evaluation, allowing constant frame rate to be maintained. Note that the same philosophy lies behind ROAM's *progressive optimization* [2].

VIII. IMPROVEMENTS

A. Dynamic Mesh Resolution Fine-Tuning

Dynamic mesh resolution adjustment allows the users to tune up or down the complexity of the triangulation at run-time, and is a desired feature for many applications. The NSS methodology readily supports it.

If we enlarge or shrink the NSS's of all the vertices by the same amount, clearly the resulting triangulation will still have strict mesh continuity, i.e. there is no T-vertices, the nested relationship between NSS's is not affected. However, as we have enlarged (shrunk) the NSS's, the number of enabled vertices will increase (decrease), and the terrain will have a higher (lower) tessellation level. Therefore, dynamic mesh resolution adjustment can be supported by simply adding another independently adjustable term to all NSS radii, r.

B. Procedural NSS generation

Natural terrains have unlimited levels of detail. As we go near and near, new details come into our view. To us, a nearby rock has nearly the same amount of details as the distant mountains. However, for today's computers, it is impractical to store every tiny bump of the entire terrain. Further more, although our eyes are keen on details, our brains rarely remember them. Hence, it makes sense that these details can be randomly generated as the user approaches and deleted as the user moves away. To support on-demand details generation, special care must be paid to ensure the seamless incorporation of random details to the base terrain, i.e. there is no crack. For our algorithm, this is relatively simple.

Procedurally generated vertices can be triangulated the same way as regular vertices, provided that they have valid NSS's, i.e. the NSS's still follow the vertex dependency (Section IV). While many potential schemes for generating valid NSS's certainly exist, a particularly simple and efficient one is presented below.

Since these details act as embellishment to the base terrain, the NSS radii of procedurally generated vertices solely depend on the size of quadrant represented by the corresponding quadtree nodes. Assume k to be the width (length) of the quadrant, r_1 the NSS radius of the center vertex, and r_2 the side vertices, as shown in Fig. 11, we define:

$$\begin{aligned} r_1 &= ak \\ r_2 &= bk \end{aligned} \tag{6}$$

Where a, b are just two constant coefficients.

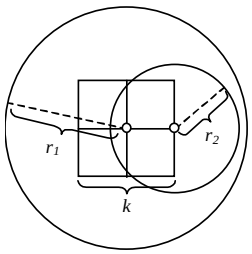

Fig. 11. The relationship between variables in (6)

Because of vertex dependency, we have:

$$\begin{cases} ak \geq \dfrac{1}{2}k + b \cdot k \\ bk \geq \dfrac{\sqrt{2}}{4}k + \dfrac{1}{2}ak \end{cases}$$

Solve to get:

$$\begin{cases} a - \dfrac{1}{2} \geq b \geq \dfrac{1}{2}a + \dfrac{\sqrt{2}}{4} \\ 2b - \dfrac{\sqrt{2}}{2} \geq a \geq b + \dfrac{1}{2} \end{cases}$$

Any pair of (a, b) satisfying this in-equation could be used to generate valid NSS's. An example is:

$$\begin{cases} a = 2 \\ b = \frac{1}{2}\left(a - \frac{1}{2} + \frac{1}{2}a + \frac{\sqrt{2}}{4}\right) = \frac{10+\sqrt{2}}{8} \end{cases}$$

Notice that the in-equation does not take the vertical difference of NSS centers into account. However, this can easily be worked around. One method is to set all NSS centers to have the same height. Another is to have a sufficiently large NSS boundary-to-boundary distance so that the vertical difference can be neglected. Another thing to remember is that during NSS enlargement (Section V), the vertices of the lowest level in the quadtree hierarchy need to have their NSS's enlarged to include its procedural counterparts.

C. Vertex Morphing

The terrain is triangulated dynamically for each frame, and as the viewer moves, the underlying mesh changes. This change can often cause visual instability when tessellation level is low. As a vertex is enabled or disabled, it jumps to its new position instantly, causing an effect called "vertex pop". When the popping distance is long, as in the case of a low-resolution mesh, this side effect becomes especially noticeable. To hide it, we can make the vertex slide gradually to its new position across several frames. This technique, called *vertex morphing* [2], [9], can be easily incorporated into our algorithm.

We do this by adding a buffer distance, B, into r (3), as shown in Fig. 12. Thus a new NSS boundary is formed. When the viewer is in the original NSS, the corresponding vertex is enabled, and when the viewer is outside the new NSS, the vertex is disabled. Vertex morphing takes place when the viewer is caught between the two boundaries.

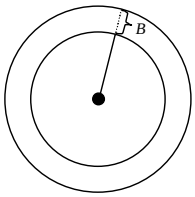

Fig. 12. Illustration of buffer distance, B.

More specifically, for vertex v, assume the old vertex position to be $\mathbf{v}'$, the new position to be $\mathbf{v}$, and function $f(\mathbf{a}, \mathbf{b})$ to be as in (2), we can compute the morphed position for the current frame, $\mathbf{v_p}$ as:

$$\mathbf{v_p} = \mathbf{v}' + \tau(\mathbf{v} - \mathbf{v}')$$
$$\tau = \text{clamp}\left(\frac{f(\mathbf{v}, \mathbf{v_0})}{B}, 0, 1\right)$$

There is a buffer distance associated with every vertex, and it can either be a constant for all vertices, or a function of the size of the quadrant represented by the corresponding quadtree node. The buffer distance can be incorporated during preprocess, or during run-time, in which case we just need to build a small table with each entry corresponding to the total buffer distance accumulated for the first, second, third, etc. levels of node and add it onto each r during triangulation.

D. LOD Accuracy adjustment

The newest generation of GPU can render triangles more quickly than the CPU can keep up with fine-grained LOD selection for each frame. This will result in either a stalled GPU waiting for the CPU to deliver triangles, or a CPU burning up all its time and resource accommodating the hungry graphics card, which is bad as well. It is still not possible, however, to render the terrain with every acre of land in its highest details without any kind of LOD systems, even on these powerful systems. To cope with the problem, we can tune down the accuracy of LOD selection a little, by either sending more triangles using the same amount of CPU time, or sending the same number of triangles using less CPU time. Our algorithm readily supports this.

Every triangle in the mesh can be subdivided into four smaller triangles. It is easy to see that after performing such an operation for all triangles in the mesh, the resulting triangulation is still seamless and continuous, without cracks, as shown in Fig. 13. This refining process can be repeatedly performed to produce even more detailed meshes. Hence, with the same amount of CPU work needed, the graphics hardware can be fed a 4, 16, 64, etc. mesh to render.

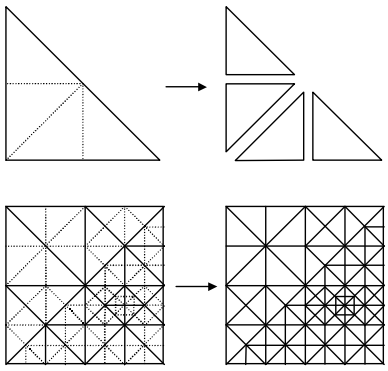

Fig. 13. Illustration of triangle subdivision.

To incorporate it into our quadtree-based triangulation, we take a one-step subdivision as an example. Instead of 9 vertices, 5×5=25 vertices will be used to triangulation each quadtree node. When the subdivision takes place, 13 of these 25 vertices will always be enabled. For the rest, their enabled states depend on the original side vertices which, when enabled, will cause them to be enabled after subdivision. So the NSS's used for their evaluation are simply identical copies of corresponding side vertices' NSS's, as show in Fig. 14.

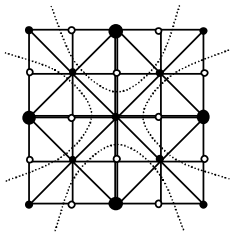

Fig. 14. Black vertices are always enabled. The four dashed curves partition the white vertices into four groups. The NSS's of white vertices are identical copies of that of the big black vertex in the same group.

For each quadtree node, a maximum of 32 triangles are batched up. Of course, this scheme can be extended to provide further subdivisions. Thus, unlimited scalability can be provided for current and future hardware.

IX. Incremental View-Frustum Culling

The hierarchical nature of the quadtree structure lends itself neatly to view-frustum culling. By starting from the root and recursively visiting the quadtree, we can efficiently update view-frustum culling states for nodes in the quad-tree. This process is given in Table II

TABLE II

Pseudo-code for incremental view-frustum culling.

```
RecVfc(QuadtreeNode q, VfcCode v)
 1    // enum VfcCode{TotalIn, TotalOut, PartialIn}
 2  if v = PartialIn then
 3    Compute VfcCode for q and store it in v
 4  switch
 5    case v = TotalOut :
 6      if q is not marked as TotalOut then
 7        Recursively remove all q's children
 8        if q ∉ T_m and q ∉ T_s then
 9          Insert q into T_m
10    case v = TotalIn :
11      if q is marked as TotalOut then
12        Mark q as TotalIn
13        Create q's four children
14        Compute VfcCode for q's children
15        Insert q's children into T_s
16      else if q is marked as PartialIn then
17        Mark q as TotalIn
18        for each c ∈ q's children do
19          RecVfc(c, TotalIn)
20    case v = PartialIn :
21      if q is marked as TotalOut then
22        Mark q as TotalIn
23        Create q's four children
24        Compute VfcCode for q's children
25        Insert q's children into T_s
26      else
27        if q is not marked as PartialIn then
28          Mark q as PartialIn
29        for each c ∈ q's children do
30          RecVfc(c, PartialIn)
```

X. Results and Discussion

Tests of our prototype implementation were done on a single Pentium III 500MHz machine with an ELSA Erazor III Pro (TNT2 Pro) 3D accelerator with 32M onboard VRAM, installed on an AGP2 port. For all the results, a 2048x2048 16bit height field of the Grand Canyon was used, and the animation is rendered in a 1024 768 sized viewport, 16bit color buffer and 16bit depth buffer. To illustrate the final rendering quality, a 2048x2048 texture, synthesized from smaller geo-typical texture patches according to terrain altitude and slope, is used for texture mapping. The flight course had a length of 4000 frames, and contained sharp turnings and altitude changes.

Our implementation was able to maintain and render a mesh containing 60,000 triangles at 30+ frames per second (fps), which is roughly 1.8 million triangles per second. (As a side note, the "optimized mesh" example that came with DirectX 8.0 SDK, which does nothing but repeatedly rendering the same optimized mesh again and again without texture, get 1.94 million triangles per second on the same system. This can be considered an approximation of peak triangle output in practical situations.) The time required for performing view-frustum culling, quadtree nodes evaluating, splitting and merging is tiny compared to that of rendering the mesh. The various portions of time spent during an average frame is given in Table III.

TABLE III

Timing data of various portion of the algorithm.

Algorithm part	*Time(ms)*	*Percentage*
View-frustum culling	2.00	6.13%
Merging quadtree nodes	0.06	0.18%
Splitting quadtree nodes	0.08	0.25%
Morphing vertices(optional)	2.20	0.25%
Rendering mesh	28.31	86.70%
Total	32.64	100.00%

Note that we implemented vertex morphing using Pentium III's SSE instruction sets. Disabling SSE lengthen the time spent morphing vertices to 4.2 ms, and cut the over all triangle rates by roughly 1/10, lowing it to 1.65 million triangles per frame.

As a comparison, the *geo-mipmap* [17] method, which is claimed to be specially designed to suit graphics hardware, reported 11,000 triangles at 50 fps on a slightly inferior machine (Pentium II 434, Viper 550). That is 550,000 triangles per second, far smaller than the 1.8 million triangles per second reached our algorithm. In addition to the exceptionally good performance, the fine-grained LOD selection enhanced with vertex morphing provides our algorithm with extremely good visual quality. It is simply arduous for us to try to catch a single vertex "pop" in our prototype implementation. In contrast, the block based LODing of geo-mipmap is heavily bogged with apparent "pops".

Much of our performance came from the circular array based lazy LOD evaluation mechanism. In most cases, the number of quadtree nodes evaluated is merely a small fraction of the total number of quadtree nodes in the hierarchy, with a typical ratio of 1:15. To see the benefit more clearly, we turned off the frame coherence optimization to force top-down triangulation for each frame, and rebenchmarked it. The frame rates slumped from 30 fps down to 17 fps.

Another performance improvement is the LOD accuracy adjustment optimization introduced in Section VIII-D. All afore mentioned figures are obtained when the implementation performs one step of subdivision. When we turned off accuracy adjustment, the portion of time spent doing LOD selection increases. The difference is shown in Table IV.

Performing subdivisions of more than one step will further reduce the time portion spent on the CPU side. Clearly, the subdivision scheme proposed in the paper provides great scalability for past and future graphics hardware.

The issue of vertex morphing is worth mentioning here. During tests, we have found that to completely hide vertex popping without vertex morphing, very small screen space

TABLE IV
COMPARISON BETWEEN ONE STEP OF SUBDIVISION AND NO SUBDIVISION.

Algorithm part	*×1 subdivision*	*no subdivision*
View-frustum culling	6.13%	11.10%
Splitting quadtree nodes	0.18%	3.02%
Merging quadtree nodes	0.25%	3.89%
Rendering	86.72%	73.41%

error threshold should be used, resulting in a mesh containing over 150,000 triangles. To keep a frame rate of 60Hz, roughly 9 million triangles must be pumped to the graphics card and rendered for each second. While vertex morphing can cut that number down to 200,000 triangles per second, without apparent degradation of visual quality. So the basic guideline here is: for hardware with possible triangle rates lower than 9 million triangles per second (i.e. non T&L hardware: TNT/TNT2, Rage 128, etc.), performing vertex morphing is the best option and doing so using SSE or 3D Now! instructions will provide top speed; Otherwise, just let the hardware draw the triangles and use some vertex buffer caching mechanism to limit the amount of vertex data touched for each frame. Because of the simplicity of our Nested Splitting Space based vertex morphing method, it is also straight forward to convert in into a vertex shader, in which case, you can benefit from the latest hardware with programmable geometry pipelines, such as GeForce 3.

XI. CONCLUSION AND FUTURE WORK

We have presented our Nested Splitting Space based LOD algorithm for terrain, which we have also implemented. First, the novel, yet simple, concept of NSS is presented, which provides an elegant solution for tackling all sorts of problems faced by terrain LOD algorithm, including dynamic mesh resolution fine-tuning, vertex morphing and procedural details accommodation. This methodology can also be utilized in other LOD algorithms. Next, an advanced framework that uses circular arrays to drive the refining and coarsening process is established on top of NSS. Combined with incremental view frustum culling and LOD accuracy adjustment, our algorithm provides high visual quality and high performance in real-world testing cases, and outperforms existing methods.

Future issues include accompanying texturing schemes, adoption of NSS to arbitrary meshes, and possible incorporation of *subdivision surface + displacement map* to support smooth skinned characters LOD.

ACKNOWLEDGMENT

This work was performed as an entry to the Intel ISEF 2001. The project was started in May, 2000 and finished in December, 2000. The author is grateful to all his reviewers and judges for their insightful comments and suggestions. He would like to thank Xiaoqing Yu for his valuable help. He also wants to thank CAST, Intel (China), and Science Service for making his trip to the San José event possible. Last but not the least, he thanks his school and his parents for the kind support they have provided.

REFERENCES

[1] P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hughes, N. Faust, and G. A. Turner, "Real-time, continuous level of detail rendering of height fields," in *Proceedings of SIGGRAPH 96*, ser. Computer Graphics Proceedings, Annual Conference Series. ACM Press, Aug. 1996, pp. 109–118.

[2] M. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein, "ROAMing terrain: Real-time optimally adapting meshes," in *IEEE Visualization '97*, Nov. 1997, pp. 81–88.

[3] U. Thatcher, "Continuous LOD terrain meshing using adaptive quadtree," Available: http://www.gamasutra.com/features/20000228/ulrich_01.htm, 1999.

[4] H. Hoppe, "Progressive meshes," in *Proceedings of SIGGRAPH 96*, ser. Computer Graphics Proceedings, Annual Conference Series, Aug. 1996, pp. 99–108.

[5] P. Heckbert and M. Garland, "Multiresolution modeling for fast rendering," in *Graphics Interface '94*, May 1994, pp. 43–50.

[6] P. S. Heckbert and M. Garland, "Survey of polygonal surface simplification algorithms," Carnegie Mellon University, Techinical Report CMU-CS-95-194, 1995.

[7] L. D. Floriani, P. Marzano, and E. Puppo, "Multiresolution models for topographic surface description," *The Visual Computer*, vol. 12, no. 7, pp. 317–345, 1996.

[8] W. S. Evans, D. G. Kirkpatrick, and G. Townsend, "Right-triangulated irregular networks," *Algorithmica*, vol. 30, no. 2, pp. 264–286, 2001.

[9] S. Röttger, W. Heidrich, P. Slusallek, and H.-P. Seidel, "Real-time generation of continuous levels of detail for height fields," in *Proceedings of the 6th International Conference in Central Europe on Computer Graphics and Visualization*, Feb. 1998, pp. 315–322.

[10] M. de Berg and K. T. G. Dobrindt, "On levels of detail in terrains," in *Proceedings of ACM Symposium on Computational Geometry*, June 1995, pp. C26–C27.

[11] P. J. Brown, "Selective mesh refinement for interactive terrain rendering," Cambridge University, Techinical Report, Computer Laboratory 417, Feb. 1997.

[12] D. Schmalstieg, "Smooth levels of detail," in *Proceedings of 1997 Virtual Reality Annual International Symposium*, Mar. 1997, pp. 12–19.

[13] H. Hoppe, "View-dependent refinement of progressive meshes," in *Proceedings of SIGGRAPH 97*, ser. Computer Graphics Proceedings, Annual Conference Series, Aug. 1997, pp. 189–198.

[14] ——, "Smooth view-dependent level-of-detail control and its application to terrain rendering," in *IEEE Visualization '98*, Oct. 1998, pp. 35–42.

[15] J. Blow, "Terrain rendering at high detail levels," in *Proceedings of the 2000 Game Developers Conference*, Mar. 2000.

[16] L. Castle and J. Lanier, "Real-time continuous level of detail (LOD) for pcs and consoles," in *Proceedings of the 2000 Game Developers Conference*, Mar. 2000.

[17] W. H. de Boer, "Fast terrain rendering using geometrical mipmapping," 2000.

[18] Microsoft, "DirectX SDK documentation," 2002.

[19] C. Bloom, "The Genesis terrain renderer technology," Available: http://www.cbloom.com/3d/, Dec. 1999.

[20] P. Cignoni, E. Puppo, and R. Scopigno, "Representation and visualization of terrain surfaces at variable resolution," *The Visual Computer*, vol. 13, no. 5, pp. 199–217, 1997.

[21] OpenGL Architecture Review Board, *OpenGL Reference Manual*, 2nd ed. Addison-Wesley, 1996.

Yuanchen Zhu is a student researcher in Computer Graphics. His research interests include continuous level-of-detail algorithms, subdivision surfaces, cloth simulation methods, image-based rendering methods, and fast global illumination algorithms. His main achievements include winning, at the Intel ISEF 2001, the Best of Category Award, First Place Award, Intel Best Use of Personal Computer Award, IEEE-Region 6 Award for Technical Excellence, Presentation, and Display, and IEEE Computer Society Third Award. He is a student member of the ACM, the IEEE, and the IEEE Computer Society.