Multiple Camera Considerations in a View-Dependent Continuous Level of Detail Algorithm

Bradley P. Kram
University of Regina
kram@cs.uregina.ca

Christopher D. Shaw
Georgia Institute of Technology
cdshaw@cc.gatech.edu

Abstract

We introduce the Camera Aware View-dEpendent Continuous Level Of Detail (CAVECLOD) polygon mesh representation. Several techniques recently have been developed that use a hierarchy of vertex split and merge operations to achieve continuous LOD. These techniques exploit temporal coherence. However, when multiple cameras are simultaneously viewing a polygon mesh at continuous LOD, the exploitation of temporal coherence is difficult. The CAVECLOD mesh representation enables multiple cameras to simultaneously exploit temporal coherence. Texture coordinates and normal vectors are preserved and the algorithm uses Microsoft DirectX Vertex Buffers and Index Buffers for efficient rendering. Interactive frame rates are achieved on large models on commercially available hardware.

Key Words: continuous level of detail, multiple camera, terrain rendering.

1 Introduction

Continuous level of detail (LOD) techniques have been recently developed as a means of reducing the number of polygons that are rendered during each frame in an interactive system. These techniques reduce visual popping effects that are often visible in static LOD techniques since the mesh is slowly refined. Large models, such as terrains, are often suitable for continuous LOD since the system may draw one part of the mesh in high resolution, and other parts of the mesh at low resolution.

Current continuous LOD algorithms exploit temporal coherence. Rather than performing all necessary calculations to refine the polygon mesh each frame, only a subset of the calculations is performed, and small, incremental changes are made to the polygon mesh. However, exploiting temporal coherence in this manner is difficult when the user is viewing a mesh at continuous LOD from multiple vantage points simultaneously. In this case, a particular region of the mesh may need to be displayed in high resolution for one camera, but in low resolution for another camera. While drawing the scene for the first camera, the system will refine a particular portion of the polygon mesh to be high resolution for one camera, and then refine the same portion of the mesh to be low resolution for another camera. During the next frame, this costly process must be repeated.

There are two primary research contributions of this paper. First, we introduce a technique that enables multiple cameras to simultaneously view a mesh in a continuous LOD system without sacrificing the exploitation of temporal coherence. This is achieved by identifying and separating the camera - specific attributes of the data structures used to implement a continuous LOD system. The empirical results presented in Section 4 show the effects of this separation, and thus provide insight into the value of temporal coherence in the single camera case.

Second, we introduce a technique for the use of indexed primitives as a means of passing the polygons to the graphics API in a continuous LOD system. Our algorithm preserves discrete attributes such as texture coordinates and surface vertex normals during the continuous LOD refinement.

Section 2 provides an overview of several current continuous LOD techniques. Section 3 presents the details of our enhanced continuous LOD technique. Section 4 discusses the results of several experiments that were performed during the development of this technique. Section 5 discusses conclusions that can be drawn from this work and areas for further investigation.

2 Previous work

Several techniques have been developed for continuous LOD. Lindstrom [8] introduces a technique for continuous LOD for triangulations of height fields. A quadtree data structure is used to recursively partition a height field. Each level in the quadtree is dynamically tessellated to achieve continuous LOD. Duchaineau [1] introduces a related technique that uses a triangle bintree structure for continuous LOD.

Luebke [10] introduces a technique that maintains a hierarchy of vertex clusters. The hierarchy is used to dynamically tessellate the polygons in an environment. Each *node* in the vertex hierarchy contains one or more vertices as well as a single *representative vertex*. To simplify part of a triangle mesh, the vertices in a node are *folded* into the representative vertex for the node, and any degenerate polygons are removed.

Hoppe [4] introduces the *progressive mesh* representation for continuous LOD. This representation is based on a vertex split operation and its corresponding edge collapse operations. Each split operation encodes the changes that are made to the geometry of the mesh along with discrete and scalar attributes such as texture coordinates and normal vectors. An intermediate *wedge* structure is used to keep track of vertices that have more than one set of discrete and scalar attributes at the vertex location. Borders of discrete attribute discontinuity are considered during the construction of the representation. Collapse operations that move these important visual cues are penalized when selecting the next edge to collapse.

Hoppe [6] extends the progressive mesh representation to take into account the camera position in the scene. This involves the re-organization of the split operations in [4] into a hierarchy of split operations. This hierarchy of split operations allows different parts of the mesh to be displayed at low resolution and other parts to be displayed at high resolution. A split operation consists of a parent vertex, v_s, and two child vertices v_t and v_u. Polygons f_l and f_r are located to the left and right of the edge introduced in a split operation. Performing a split operation removes v_s from the active list of vertices, adds v_t and v_u to the active list of vertices, and adds f_l and f_r to the active list of polygons. Performing a collapse operation adds v_s to the active list of vertices, removes v_t and v_u from the active list of vertices, and removes f_l and f_r from the active list of polygons.

Selective refinement of the mesh is achieved by traversing the active list of vertices and determining if the vertex should be split into its child vertices, collapsed into its parent vertex (along with its sibling), or left unchanged. A triangle strip representation of the active polygon mesh is generated for optimal rendering. Hoppe reports that the algorithm is able to generate strips long enough to realize performance gains.

Hoppe [5] specializes the view-dependent progressive mesh framework of [6] for the case of real time rendering of terrain. This technique involves a re-organization of the view-dependent progressive mesh data structures in an effort to minimize memory requirements.

Xia and Varshney [14] present a scheme similar to that of Hoppe. Similar to [6], a hierarchy of split operations is constructed off line and traversed at run time. While Hoppe uses an energy function for selecting an edge to collapse during compilation, Xia and Varshney use a shortest edge heuristic while constructing the vertex hierarchy.

To maintain the integrity of the mesh throughout the refinement process, all adjacent polygons to the vertex involved in a split operation must be present in the current mesh before the split may take place. As a result, only gradual changes from high resolution to low resolution are possible in the mesh. In contrast, [6] requires that only the two neighbour polygons of the two new polygons introduced in the split operation are present.

3 CAVECLOD mesh representation

In this section, we introduce the Camera Aware View-dEpendent Continuous LOD (CAVECLOD) mesh representation. A CAVECLOD mesh is a new representation for continuous LOD that allows multiple cameras to simultaneously exploit temporal coherence while viewing the mesh and utilizes Microsoft Direct3D Vertex Buffers and Index Buffers. Similar to [5, 6, 14] the CAVECLOD mesh representation uses a hierarchy of split and collapse operations to achieve continuous LOD.

3.1. CAVECLOD basics

Indexed primitives are an effective mechanism for passing polygons to a 3D graphics API. An application first passes the graphics API an array of vertices that consist of each vertex's location, normal vector, texture coordinates, and so on. The indices in this array are used when defining the list of polygons for the graphics API to draw. This method is more efficient than passing the information about each vertex since vertices are often reused, less data is transmitted, and the graphics API only needs to transform the vertex and perform lighting calculations at the vertex once [11]. Indexed primitives are supported by both Microsoft DirectX and OpenGL.

While several polygons may share a single vertex's location, each polygon may have a different normal vector or different texture coordinates associated with the vertex location. Currently, indexed primitives require vertex locations such as these to be duplicated in the vertex array. This duplication introduces problems in a continuous LOD representation since steps must be taken to ensure that refinement of the mesh does not separate these duplicated vertices [3].

The CAVECLOD mesh representation uses Microsoft DirectX 8 Vertex Buffers and Index Buffers. Each *D3Dvertex* consists of x, y, and z coordinates, a normal vector, and texture coordinates. Since a particular vertex location may be present more than once in the vertex array, the CAVECLOD mesh representation utilizes an intermediate data structure called a *node* to maintain connectivity throughout the mesh. Nodes are stored in an array, and a *node identifier* is used to find a particular node in the array. A polygon consists of three node identifiers. A node maintains a *node position identifier*, which uniquely identifies a location in 3D space where one or more D3Dvertices is found.

A polygon consists of surface properties such as colour and a texture identifier that specifies which texture is to be applied to a surface. In the CAVECLOD mesh representation, polygons contain a *surface identifier* that uniquely identifies the set of surface properties applied to the polygon. Polygons are stored in an array, and a *polygon identifier* is used to find a particular polygon in the array.

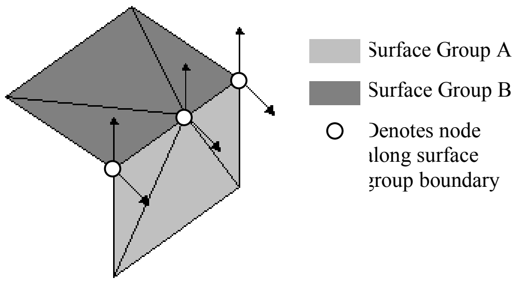

Figure 1: Normal vector discontinuity forms a surface group boundary

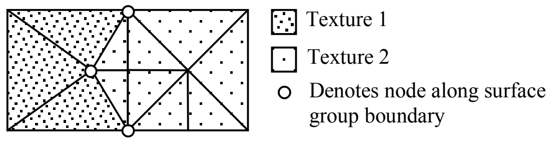

Figure 2: Surface identifier discontinuity forms surface group boundary

Adjacent polygons that share the same surface identifier and that share the same normal vector and texture coordinates at both shared node positions belong to the same *surface group*. A polygon mesh may be partitioned into surface groups by examining the normal vector and texture coordinates at each node position. A surface group is enclosed by a *surface group boundary*[1], which is the set of nodes that are part of more than one surface group. Figure 1 and Figure 2 depict two examples of surface group boundaries in a polygon mesh.

Each polygon is assigned a *surface group identifier*. Since polygons in the same surface group will use the same D3Dvertex at a particular node position, a polygon's surface group identifier and the polygon's three node identifiers can be used to identify the polygon's D3Dvertices. The CAVECLOD mesh representation maintains a two-dimensional array of D3Dvertex indices called the *D3Dindex array*. When drawing a polygon, the polygon's surface group identifier and the node position identifier are used to lookup the appropriate D3Dvertex index in the D3Dindex array.

[1] For meshes with boundaries, nodes along the boundaries of the mesh are also considered surface group boundaries.

3.2. Continuous LOD

Similar to [5, 6, 14], continuous LOD is achieved through a hierarchy of split and collapse operations. Since nodes maintain connectivity information about the vertices in the mesh, split and collapse operations are expressed as a hierarchy of nodes. Each node maintains the node identifier of its *parent node* and two *child nodes*. Similar to [6], a node also stores the identifiers of the two polygons that are created when the node is split (to the left and right of the new edge that is created) along with the required neighbour polygons that must be present in order for the split to take place. Similar to [5], node positions do not move during collapse or split operations. A parent node's location must be the same as one of its child nodes. The formation of the node hierarchy is shown in Figure 3.

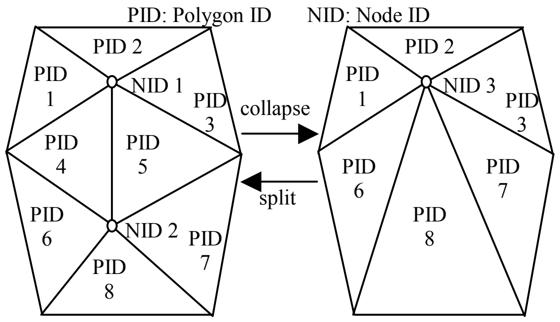

Figure 3: A node hierarchy

An edge collapse operation replaces two child nodes with one parent node. The child node with the same node position identifier as its parent is known as the *stationary child node* (NID1 in Figure 3). The appearance of polygons that contain the stationary child node remains unchanged before and after a split or collapse operation. The child node that does not have the same node position identifier as its parent node is known as the *floating child node* (NID2 in Figure 3). The corners of polygons that contain the floating child node appear to move during split and collapse operations.

To maintain the integrity of the mesh during the entire refinement process, potential split or collapse operations must meet certain conditions before the operations take place. The split and collapse preconditions of [6] are used. In Figure 3, the split of active node NID3 is legal if the polygons PID1, PID3, PID6 and PID7 are active. The collapse of active nodes NID1 and NID2 is legal if the polygons PID1, PID3, PID6 and PID7 are active and in the configuration of Figure 3.

3.3. Surface group preservation

Surface group boundaries such as edges and texture boundaries are often important visual cues for object recognition [2]. Therefore, preserving these boundaries during the refinement process is desirable. In the CAVECLOD mesh representation, a polygon belonging to a particular surface group cannot contain a node at a node position where surface group information does not exist for the surface group.

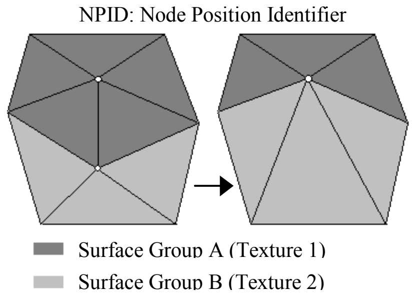
NPID: Node Position Identifier
Surface Group A (Texture 1)
Surface Group B (Texture 2)

Figure 4: An illegal edge collapse

Figure 4 depicts a surface group boundary that is defined by a texture border. After the collapse operation, the polygons in Surface Group B are moved to a node with a node position identifier of 1. Since no polygons in Surface Group B have polygons with a vertex at node position 1 in the high-resolution mesh, there is no D3Dvertex at node position 1 for Surface Group B. As a result, there is no D3Dvertex with texture coordinates that will correctly map Texture 2 onto the polygons in Surface Group B at node position 1.

During CAVECLOD mesh construction, we preserve surface group boundaries by classifying node positions in the high-resolution mesh [4, 13]. Our approach is similar to [4], however, rather than penalizing collapses that move surface group boundaries, we disallow them.

Node positions that are not part of a surface group boundary are classified as *FREE*. Node positions that are part of one surface group boundary are classified as *SHARP*, and node positions that are part of more than one surface group boundary are classified as *RIGID*. Figure 5 depicts these classifications.

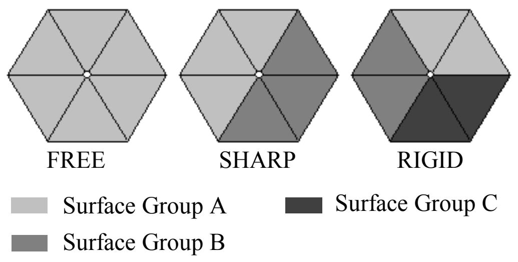

Figure 5: Classification of node positions

In order to maintain surface group boundaries, restrictions are placed on how the various node position types can appear in collapse and split operations. A node at a position labelled RIGID must be a stationary child node since it exists on more than one surface group boundary. A node at a SHARP position may be either a stationary child node or a floating child node. However, a node at a SHARP position may only be involved in a collapse if the stationary child node in the collapse is located along the same surface group boundary. A node at a node position labelled FREE may be either a floating child node or a stationary child node.

These rules ensure that a collapse operation does not move a polygon to a node position where surface group information does not exist for the surface group.

If a polygon mesh consists of q node positions and r surface groups, the D3Dindex array must contain q x r D3Dvertex indices. However, a particular surface group may only span a small portion of the mesh and therefore relatively few node positions may be found in a particular surface group. As a result, the two-dimensional array may be sparsely populated.

We may reduce the size of the D3Dindex array by re-using surface group identifiers. Let s_1 and s_2 be surface group identifiers. Let N_1 be the set of node position identifiers that contain a D3Dvertex for s_1 in the high-resolution mesh and let N_2 be the set of node position identifiers that contain a D3Dvertex for s_2 in the high-resolution mesh. Let p be a polygon with surface group identifier s_1. No legal edge collapse may modify p to contain a node with a node position identifier n, $n \notin N_1$. Therefore, if N_1 and N_2 are mutually exclusive, no series of legal edge collapses will cause them to intersect.

In other words, if two surface groups do not share a common node position in the high-resolution mesh, no series of legal edge collapses will cause the two surface groups to share a common node position at any level of detail. This means that if two surface groups do not share a common node position, they may be assigned the same surface group identifier. Figure 6 depicts an example of two surface groups that are assigned the same surface group identifier.

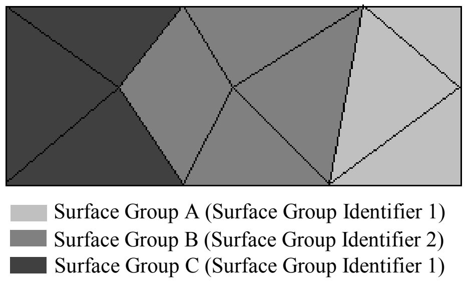
Surface Group A (Surface Group Identifier 1)
Surface Group B (Surface Group Identifier 2)
Surface Group C (Surface Group Identifier 1)

Figure 6: Assignment of surface group identifiers

3.4. Multiple camera considerations

The CAVECLOD mesh representation enables multiple cameras to simultaneously exploit temporal coherence. This is achieved by separating the camera – specific attributes of the data structures.

Several aspects of the data structures need not be duplicated when an additional camera views the mesh. The D3Dindex array, the node hierarchy, polygon surface identifiers and surface group identifiers, and the DirectX Vertex Buffer are examples of these data structures.

In order for each camera to exploit temporal coherence, each camera must maintain its own active polygon list and active node list. In order to properly display and update the mesh, an active polygon in a camera's active polygon list must maintain the node identifiers that define its current geometry as well as the polygon's current neighbours in order to determine whether a potential collapse or split is legal. A *camera refinement* data structure is maintained for each camera that is viewing the mesh. This data structure maintains the active node list and the active polygon list for a particular camera.

The active node lists and active polygon lists are implemented as doubly linked lists. However, during split and collapse operations, we must also quickly be able to determine if a particular node or polygon is active for a given camera. For this reason, each camera refinement data structure maintains an array of pointers to active nodes and an array of pointers to active polygons. To determine if a particular polygon is active, we may use the polygon identifier to lookup the pointer in the camera refinement data structure's array of pointers to active polygons. If the pointer is NULL, the polygon is inactive. If the pointer is not NULL, it points to an *ActivePolygon* data structure that maintains the current information about the polygon.

The data structures are designed to minimize the amount of additional memory required as the number of cameras viewing the mesh increases. The memory requirements are summarized in Equation 1.

$$b = 44n + 12p + 32d + 2se + 12n_a + 36p_a + c(4n + 4p + 16) \tag{1}$$

where

b = the total number of bytes required for the mesh
n = the total number of nodes in the hierarchy
p = the total number of polygons in the high-resolution mesh
d = the total number of D3Dvertices
s = the number of unique surface group identifiers
e = the number of unique node positions in the hierarchy.
n_a = the number of active nodes (total for all cameras)
p_a = the number of active polygons (total for all cameras)
c = the number of cameras

3.5. Rendering

When a CAVECLOD mesh is loaded into memory, a DirectX Vertex Buffer is constructed with the list of D3Dverticies for the mesh. Drawing the scene for a particular camera is performed by traversing the active polygon list. The polygon's node position identifiers, along with the polygon's surface group identifier are used to lookup three D3Dvertex indices in the D3Dindex array. These indices are added to a DirectX Index Buffer that is passed to the DirectX API when traversal of the active polygons is complete. A separate DirectX Index Buffer is maintained for each surface identifier. This approach minimizes the number of texture state changes that are required when rendering.

Two runtime refinement criteria are used. The view frustum criterion of [6] is used to coarsen areas of the mesh that are not visible, and the screen space error metric of [9] is used to estimate the visual importance of polygons in the mesh.

3.6. Algorithm summary

For a particular camera, the incremental selective refinement algorithm of [6] is used to achieve continuous LOD. At each active node, we examine the node to determine if it should be split, collapsed into its parent, or remain unchanged. An *evaluate_node* function examines the node parameters with respect to the criteria described in the previous section.

The polygons surrounding an active node that should be split may be configured in a manner such that the split is illegal. In this case, we first split the nodes that form the polygons. A collapse operation may not be legal even though evaluation of the node states that it should be collapsed. In this scenario, the node is not collapsed. In summary, a portion of the mesh moves from low resolution to high resolution when necessary and moves from high resolution to low resolution when possible [6].

4 Results

We present the results of several tests and experiments that were performed to demonstrate use of the CAVECLOD mesh representation. Testing was performed on a 1.1 GHz AMD Athlon personal computer running Windows 2000 Professional with 384 MB of PC133 RAM and an ATI RADEON™ (64MB) graphics card.

4.1. Testing overview

In order to measure the effectiveness of the CAVECLOD mesh representation, two applications were tested. The first application uses the CAVECLOD mesh representation to achieve continuous LOD for multiple cameras. A second application calculates a continuous LOD representation for each camera without using the CAVECLOD representation. When displaying the scene for several cameras, the mesh is refined to display the scene for Camera A, then refined to display the scene for Camera B, and so on. Temporal coherence is not exploited for each camera since the refinement process that calculates the scene for camera B begins with the mesh that was displayed for Camera A. As a result, this application must perform a significant number of refinement operations each frame to display the scene for each camera.

The system was tested on three terrain models. The Three Lakes terrain model (Figure 7) consists of 80,000 polygons and represents a fictional patch of terrain 70 km by 70 km. The model consists of nine different textures that form surface group boundaries. Five unique surface group identifiers are assigned to the surface groups. The D3Dindex array is a 5 x 40,401 array that is 21% populated since only 21% of the surface group identifier – node position identifier combinations are in use.

The Cliff terrain model (Figure 8) consists of 107,184 polygons and represents a fictional patch of terrain 70 km x 70 km. The model consists of four surface group identifiers and uses five different textures. The edge separating the top of the cliff from the side of the cliff is an example of a surface group boundary that is defined by normal vector discontinuity. The D3Dindex array is a 4 x 54,056 array that is 26% populated.

The Grand Canyon model (Figure 9) is based on data from the United States Geological Survey. The model represents an area surrounding the Grand Canyon that is approximately 122 km by 245 km. The model contains 126,480 polygons and a single texture is applied to the entire model. As a result, the model contains only one surface group, and the D3Dindex array is reduced to a one-dimensional array that contains all of the D3D vertex indices.

Testing was performed through a series of two-minute flyovers over the terrain models. A one-camera flyover was performed with Camera A. A two-camera flyover was performed with Camera A and Camera B. Three and four camera flyovers were performed with Cameras A, B, C and Cameras A, B, C, D respectively. When displaying the scene for one camera, the display covers the entire screen. When displaying the scene for two cameras, the second camera's display occupies the top left quarter of the screen. Three cameras are displayed by placing the second and third cameras across the top half of the screen. The view from each camera is drawn in a separate quadrant for the four-camera flyover.

All testing was performed at a full screen resolution of 800 x 600. The view frustum run time criterion of [6] is used along with the screen space error criterion of [9]. For the screen space error criterion, a 7% error tolerance is used.

4.2. Exploiting temporal coherence

The CAVECLOD mesh representation is designed to enable multiple cameras to simultaneously exploit temporal coherence.

Figure 10 shows the number of refinement (split and collapse) operations that take place per second for a four-camera flyover of the Grand Canyon in the application that does not use the CAVECLOD mesh representation. The terrain model is refined for Camera A, rendered, refined for camera B, rendered, and so on. In this application, a significant number of refinement operations must be made each frame to display the scene for each camera.

With the CAVECLOD mesh representation, each camera maintains its own active polygon and active node lists and incrementally refines these lists. This dramatically reduces the number of refinement operations required each frame. Figure 11 depicts the number of refinement operations that take place per second during the same four-camera flyover of the Grand Canyon model. The number of split operations required per second for each camera stays below 2000 for most of the flyover compared to an average of 14,868 refinements per camera per second without the CAVECLOD representation. As shown in Figure 12 the use of the CAVECLOD representation impacts the frame rate of the system.

Similar flyover experiments were performed on each model with one, two, three and four cameras respectively. Complete details of all flyovers for all models can be found in [7].

4.3. Memory requirements

As outlined in Section 3.4, the introduction of additional cameras does not significantly increase the amount of memory required by the CAVECLOD mesh representation. As shown in Equation 1, three factors affect the memory required to store a CAVECLOD mesh:

- the size of the static structures of the mesh (*n*, *p*, *d*, *s*, and *e* in Equation 1)
- the number of cameras (*c* in Equation 1)
- the number of active polygons and active nodes (n_a and p_a in Equation 1)

Table 1 displays the actual memory requirements for the three models. On average, the amount of additional memory required per camera is 10.5% of the memory requirements for the static portions of the mesh.

Table 1: Memory requirements for each model

	Grand Canyon	**Three Lakes**	**Cliff**
Static structures $44n + 12p + 32d + 2se$	9,297,526 bytes	6,284,962 bytes	8,261,168 bytes
Memory required per camera $4n + 4p + 16$	1,016,184 bytes	643,064 bytes	861,088 bytes

The number of active polygons and active nodes is determined at run time and varies as the user moves throughout the environment. As more cameras are added, the number of active polygons and active nodes will increase. Figure 13 depicts the number of active polygons for each camera during the flyovers of the three lakes terrain model. Figure 14 depicts the memory required for the various flyovers of the model. Clearly, the introduction of additional cameras does not dramatically increase the amount of memory required by the system.

4.4. Vertex buffer coverage

As outlined in Section 3.5, rendering a CAVECLOD mesh is achieved through the use of DirectX Vertex Buffers and Index Buffers. During the rendering process, the vertex buffer is transferred to the graphics card, and index buffers are constructed that refer to the vertices in the vertex buffer. All of the D3Dvertices in the high-resolution mesh are transferred to the graphics card each frame even though not all of the D3Dvertices may be used. As expected, as the number of cameras increases, the number of D3Dvertices that are used will increase. Even though the entire vertex buffer is transferred to the video card, performance gains are still realized. The graphics processor only transforms and performs lighting calculations on vertices that are used. Also, since the number of polygons drawn each frame has been dramatically reduced from the high-resolution model, less processing takes place during the rasterization stage of the rendering process.

5 Conclusions and future work

We have introduced the CAVECLOD mesh representation that enables multiple cameras to simultaneously exploit temporal coherence in a continuous LOD algorithm. The system uses Microsoft DirectX Vertex Buffers and Index Buffers for efficient rendering. We have shown in Section 4 that the system is capable of rendering large models at interactive rates on commercially available hardware. The results in Section 4 also demonstrate the importance of the exploitation of temporal coherence in a continuous LOD algorithm.

While this new representation has been developed, several areas are left as future work. Since each camera maintains its own active node and active polygon lists, a parallelized version of the algorithm may be advantageous. The refinement process for each camera only reads from the shared static portions of the data structures and modifies the active node and polygon lists for a single camera. For this reason, the refinement process for each camera may take place in a separate thread without introducing significant synchronization issues. Future work would explore the performance gains that could be realized by parallelizing the algorithm in this fashion and running it on a multiprocessor platform.

Acknowledgements

We wish to thank James Hall and Dee Jay Randall for coding contributions to the underlying framework used by our current implementation. This work was supported by financial assistance from the Natural Sciences and Engineering Research Council of Canada (NSERC) and from the University of Regina Faculty of Graduate Studies. Data for the Grand Canyon model was obtained from The United States Geological Survey (USGS), with processing by Chad McCabe of the Microsoft Geography Product Unit.

References

[1] Duchaineau. M., Wolinsky, M., Sigeti, D., Miller, M., Aldrich, C., Mineev-Weinstein, M., "ROAMing Terrain: Real-time Optimally Adapting Meshes," *IEEE Visualization '97*: pp. 81-88.

[2] Garland, M., Heckbert, P., "Simplifying Surfaces with Color and Texture using Quadric Error Metrics," *IEEE Visualization '98*: pp. 263 - 269.

[3] Hoppe, H., "Efficient Implementation of Progressive Meshes," *Computers and Graphics* Vol. 22 No.1 (1998): pp. 27-36.

[4] Hoppe, H., "Progressive Meshes," *SIGGRAPH'97 Proceedings* (1997): pp. 99-108.

[5] Hoppe, H., "Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering," *IEEE Visualization '98* : pp, 35-42.

[6] Hoppe, H., "View-Dependent Refinement of Progressive Meshes," *SIGGRAPH'97*: pp. 189-198.

[7] Kram, B.P., *Multiple Camera Considerations in a View-Dependent Continuous Level Detail Algorithm* M.Sc. Thesis, Univ. of Regina, 2002.

[8] Lindstrom, P., Koller, D., Ribarsky W., Hodges, L., Faust, N. and Turner, G., "Real-time continuous level of detail rendering of height fields," *SIGGRAPH'96*: pp 109 - 118.

[9] Luebke, D., *View-Dependent Simplification of Arbitrary Polygonal Environments* Ph.D. Thesis, University of North Carolina - Chapel Hill, 1998.

[10] Luebke, D. and Erikson, C., "View-Dependent Simplification of Arbitrary Polygonal Environ-ments," *SIGGRAPH'97*: pp. 199-208.

[11] Microsoft DirectX 8 Developer FAQ, http://msdn.microsoft.com/library/default.asp?URL =/library/techart/DirectX8faq.htm.

[12] Sander, P., Snyder, J., Gortler, S., Hoppe. H., "Texture mapping progressive meshes," *SIGGRAPH 2001* pp 409-416.

[13] Schroeder, W., "A Topology Modifying Progressive Decimation Algorithm," *IEEE Visualization '97*: pp. 205-212.

[14] Xia, J. and Varshney, A., "Dynamic View-Dependent Simplification for Polygonal Models," *IEEE Visualization '96*: pp. 335-344.

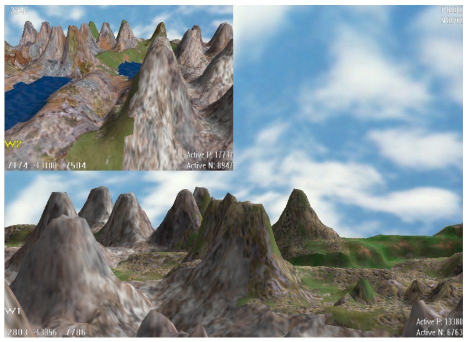
Figure 7: Two-camera flyover of the three lakes model

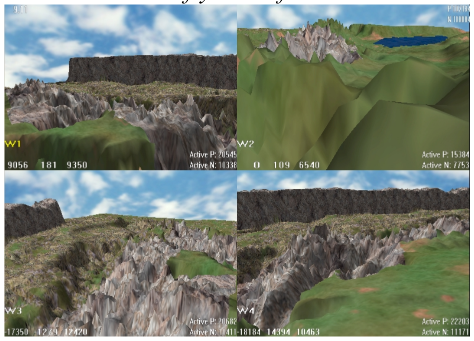
Figure 8: Four-camera flyover of the Cliff model

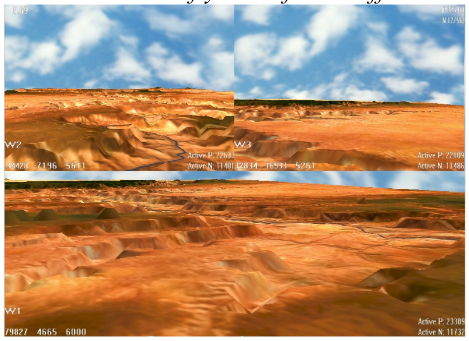
Figure 9: Three camera flyover of the Grand Canyon

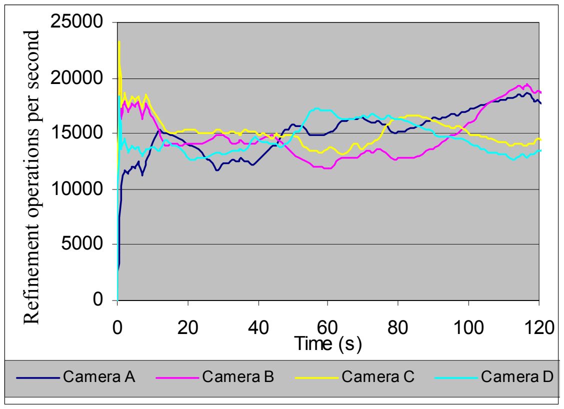

Figure 10:Refinement operations on the Grand Canyon model without the CAVECLOD representation

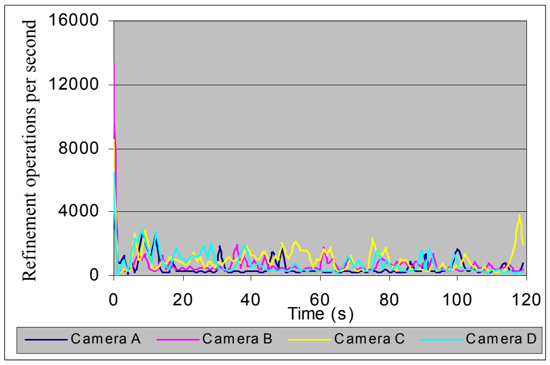

Figure11: Refinement operations/sec on the Grand Canyon model with the CAVECLOD representation

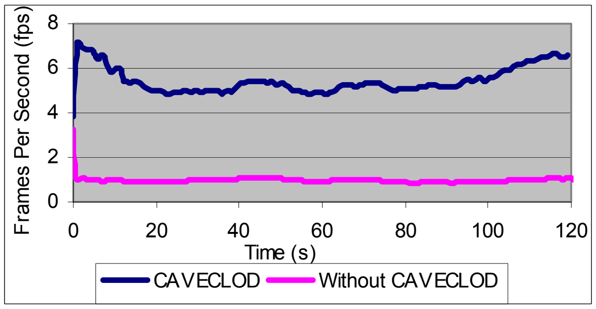

Figure 12: Frame rates with and without CAVECLOD model in the 4-camera flyover of the Grand Canyon model

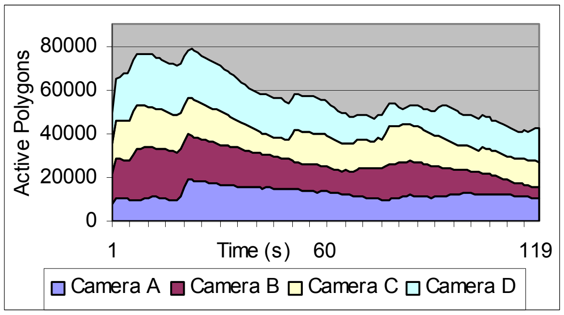

Figure 13 Active polygon counts for the four-camera flyover of the Three Lakes model

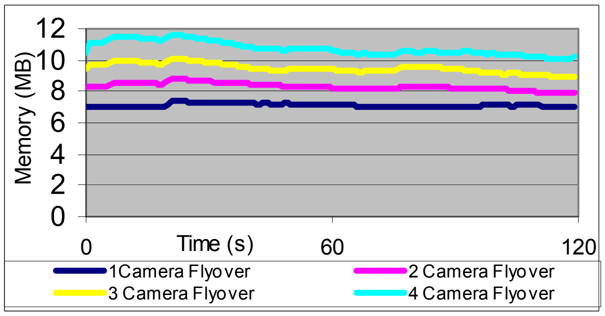

Figure 14 Memory requirements for the flyovers of the Three Lakes model