GPU-Based Adaptive-Subdivision for View-Dependent Rendering

Gilad Bauman
Ben-Gurion University
Beer-Sheva, Israel
baumang@cs.bgu.ac.il

Yotam Livny
Ben-Gurion University
Beer-Sheva, Israel
livnyy@cs.bgu.ac.il

Jihad El-Sana
Ben-Gurion University
Beer-Sheva, Israel
el-sana@cs.bgu.ac.il

Abstract

In this paper, we present a novel view-dependent rendering approach for large polygonal models. In an offline stage, the input model is simplified to reach a light coarse representation. Each simplified face is then assigned a displacement map, which resembles the geometry of the corresponding patch on the input model. At runtime, the coarse representation is transmitted to the graphics hardware at each frame. Within the graphics hardware, the GPU subdivides each face with respect to the view-parameters, and adds fine details using the assigned displacement map. Initial results show that our implementation achieves quality images at high frame rates.

Keywords: GPU Processing, Subdivision Surfaces, Level-of-detail.

1 INTRODUCTION

Interactive rendering of large polygonal models is vital for various visualization and virtual environments applications. The drive for fine details and the availability of technologies that simplify the design and acquisition of graphics models have lead to the generation of large models that exceed the interactive rendering capabilities of contemporary graphics hardware. In addition, some of these applications apply complex animation effects to these models. These further reduce the rendering frame rates. Level-of-detail rendering schemes were suggested to bridge the gap between models' complexities and rendering capabilities.

View-dependent rendering approaches enable the coexistence of different resolutions over the various regions of a level-of-detail representation, based on view-parameters. Early view-dependent rendering algorithms rely on the CPU to extract an appropriate level-of-detail representation. However, the CPU is often incapable of extracting and transmitting the geometry of large datasets within the duration of a single frame. In addition, these algorithms use hierarchies of geometry, which are constructed offline. As a result, they cannot support runtime deformations or animation effects on the processed model, without additional expensive update or reconstruction of the hierarchy. To accelerate the selection of level-of-detail representations, cluster-based view-dependent algorithms were introduced. They overcome CPU incapabilities by representing models using clusters or patches. In such a scheme, the CPU extracts representations for large models by switching between large amounts of geometry, using a small set of operations. However, the separation into patches often limits local adaptivity, and therefore, cluster-based approaches may require more triangles than earlier approaches to maintain the same image quality. Current cluster-based approaches use hierarchies of simplified patches, which prevent efficient runtime deformations or animations.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

In this paper, we present a novel view-dependent level-of-detail rendering algorithm that does not use geometric hierarchies and enables runtime deformation. In addition, the memory requirements of the suggested algorithm are low with respect to previous view-dependent rendering algorithms. In an offline stage, the input model is simplified to reach a light coarse representation, which is used to guide the sampling of the original model. The sampling values are stored in displacement maps, which are assigned to the faces of the coarse representation. At runtime, the coarse representation is transmitted to the graphics hardware at each frame. Within the graphics hardware, the GPU adaptively refines each face with respect to its orientation and the assigned displacement map to generate a view-dependent representation. Deforming or applying animation effects to large 3D models involves expensive computations that usually cannot be completed within the duration of a single frame. For that reason, such computations are usually applied to the skeleton or the coarse representation of 3D models. In our approach, the CPU can deform the geometry of the coarse representation before transmitting it to the GPU. In such a manner, our algorithm provides interactive view-dependent rendering of animated large models.

Our approach transfers most of the computationally intensive operations into the GPU for interactive view-dependent rendering of large models while enabling

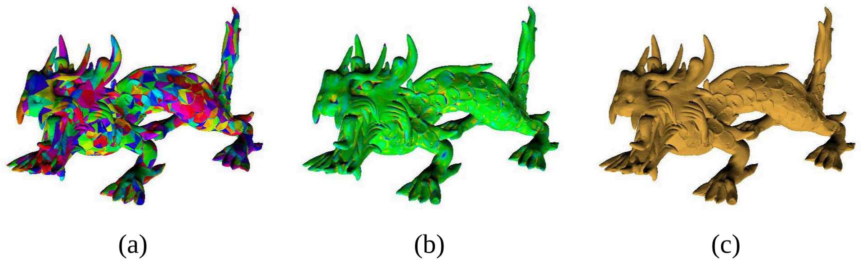


Figure 1: Screenshots of the Asian Dragon model (a) the control-mesh, (b) the sampling error, and (c) the shaded model

fine-grained changes on the generated representation. The coarse representation partitions an input model into disjoint patches in an error-guided manner. These patches are compactly encoded using displacement maps. Our approach manages to eliminate the need for geometric hierarchies, which are used to guide the selection of the level-of-detail representations in view-dependent rendering approaches. It also dramatically reduces the CPU processing load and the CPU-GPU communication load by processing a coarse representation within the CPU and refining it within the GPU. Eliminating the use of geometric hierarchies and extracting the level-of-detail within the GPU enables the CPU to dynamically modify the processed model in real-time.

2 RELATED WORK

Traditional view-dependent schemes rely on hierarchies that encode the geometry of the original model in multiple levels of detail. At runtime the CPU traverses the hierarchy and extracts a level of detail representation based on view-parameters [7,14,20,22].

To improve the selection of the level-of-detail representations, cluster-based view-dependent algorithms were introduced [9, 15]. In these approaches the CPU extracts representations for large models by switching between large amounts of geometry, using a small set of operations. However, simplifying the patches independently imposes severe difficulties in stitching adjacent patches or clusters seamlessly. To overcome these difficulties, several approaches introduce dependencies among patches [6, 26] or introduce sliver/degenerated triangles [1, 25]. Transmitting the extracted level-of-detail representation to the graphics hardware at each frame reduces the rendering rates and forms a severe bottleneck. Several algorithms utilize caching schemes, such as Vertex Buffer Object (VBO), to upload geometry into the video memory in realtime [6, 26] while others use geometry streaming between the CPU and GPU [5, 23, 24].

Current graphics hardware include GPUs that can alter vertices, geometry, and fragments properties in a parallel manner, which influenced the development of view-dependent algorithms. Early GPU-based view-dependent rendering algorithms were designed for terrain visualization [2, 3, 19]. Terrain algorithms usually cache the geometry within the video memory, and utilize temporal coherence to improve performance and reduce CPU-GPU communication [1, 4]. In later algorithms, the programmable GPU was used to refine coarse terrain tessellations using a predetermined template of geometry [12, 17] or to add fine details [8, 11].

GPU-based algorithms for 3D models have lately been presented. One of the starting points for GPU based displacement mapping is the work of [21]. Several past algorithms cache the hierarchy within the video memory and use the multi-pass procedures to extract a view-dependent representation [13]. Others extract a coarse view-dependent representation of the model within the CPU and transmit it to the GPU for refinement [16] or adaptively refine a mesh of a frame to generate the mesh of the next one, in an incremental fashion using the GPU [18].

3 OUR APPROACH

In this section we describe a GPU-based algorithm for view-dependent level-of-detail rendering that does not store multiresolution hierarchies of geometry. Instead, it relies on GPU capabilities to adaptively refine various regions of the model with respect to view-parameters. Our algorithm is divided into offline preprocessing and runtime rendering.

The preprocessing stage starts by creating a simplified representation of the input model, which will be called the *control-mesh* and its faces will be denoted the *control-faces*. The control-mesh is used to recover the original input model at runtime. The faces of the control-mesh are subdivided according to a predefined pattern that guides the sampling of the original model. The mesh results from subdividing the control-mesh will be called the *refined-mesh* and we will refer to its vertices as the *refined-vertices*. We will also refer to the sampling of the original mesh as the *sampled-mesh* and its vertices as the *sampled-vertices*. The sampling of each face of the control-mesh is stored as a displacement map (see Figure 2).

A polyline p is called *x-monotone* if it has one value $p(x)$ for each x, e.g, $p(x)$ is a function of x (see Figure 3a). Similarly, we define a polyline q to be $\widetilde{x}$-*monotone* in the interval $[a,b]$ along the x-axis if the

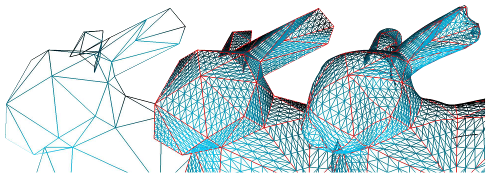


Figure 2: From left to right: the control-mesh, the refined-mesh, and the sampled-mesh

normal N_x at the point $x \in [a,b]$ intersects the polyine q at one point, where N_x is determined by interpolating the two normals at a and b (as in Phong Shading, see Figure 3b and 3c). In $\mathbb{R}^2$ we define a polygonal surface s as $\widetilde{xy}$*-monotone* with respect to the triangle t if the normal at any point $v \in t$, computed using the normals at the vertices of t, intersects the surface s once. The intersection of the normals at the boundary of a triangle t, with a polygonal surface spans a surface patch P_t, and defines a correspondence between t and P_t, i.e., the triangle t corresponds to the patch P_t and vice versa (see Figure 4). A triangular mesh M is $\widetilde{xy}$*-monotone* with respect to another mesh $\widehat{M}$ if every patch P_t in M is $\widetilde{xy}$*-monotone* with respect to its corresponding triangle t in $\widehat{M}$.

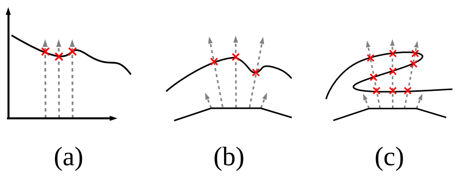


Figure 3: (a) an x*-monotone* polyline, (b) an $\widetilde{x}$*-monotone* polyline, and (c) a non-$\widetilde{x}$*-monotone* polyline

The control-mesh and the displacement maps assigned to its faces are used to recover the original mesh, which is possible only if the original mesh is $\widetilde{xy}$*-monotone* with respect to the control-mesh.

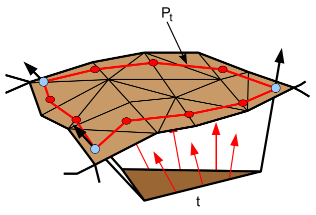


Figure 4: The triangle t and its corresponding patch P_t

To generate a control-mesh M^c from an input mesh M, our algorithm uses the edge-collapse simplification operator with the quadric error metric [10]. It avoids collapsing edges that may violate the $\widetilde{xy}$-monotone property. For that purpose, our algorithm maintains a normal-cone for each vertex v, which encodes the normals of its adjacent triangles, as well as the normals of the triangles which collapsed onto it in past iterations. An edge-collapse is defined as *valid* if it does not result in a normal-cone (for any affected vertex) that exceeds a half-sphere. The simplification algorithm executes only valid edge-collapses, ordered by their quadric errors. It proceeds until it reaches a predetermined target polygon count, or until no valid collapses remain.

3.1 Mesh Sampling

Our algorithm uses the control-mesh to sample the original model's surface. A predetermined triangular grid, which will be called the *sampling-pattern*, is used to guide the sampling process for each control-face. A ray is shot through each refined-vertex v along its interpolated normal N_v and the intersection point v^x of the ray with the original model surface is computed. The distance between v and v^x defines the elevation value, which is stored in the displacement map assigned to the processed face (see Section 3.2).

The sampling-pattern is a uniform subdivision of an equilateral triangle in which the number of vertices along each of the triangle edges is equal. We shall refer to the number of vertices along an edge of the sampling-pattern as the *degree* of the sampling-pattern. A sampling-pattern of degree k has $k(k+1)/2$ vertices and $(k-1)^2$ triangles.

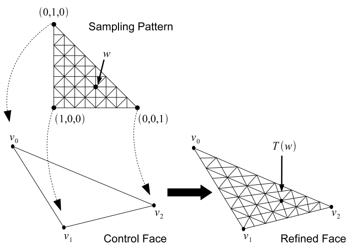
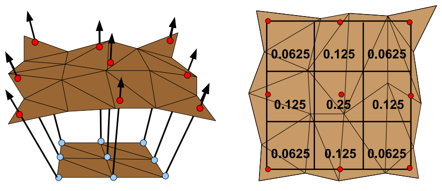


Figure 5: The sampling pattern ($k = 9$)

In the sampling phase the sampling-pattern is mapped to match the processed control-face. The three corner vertices of the sampling-pattern are assigned the coordinates $(1,0,0)$, $(0,1,0)$, and $(0,0,1)$, and the coordinates of the remaining vertices are determined accordingly in a uniform fashion. Mapping the vertices of the sampling-pattern onto a control-face f is performed using Equation 1a, where w_x, w_y, w_z are the coordinates of the pattern's vertex w, and v_0, v_1, v_2 are the vertices of f (see Figure 5). Similarly, the interpolated normals at the mapped vertices are calculated using Equation 1b.

$$T(w) = w_x * v_0 + w_y * v_1 + w_z * v_2 \tag{1a}$$
$$N(w) = w_x * n_0 + w_y * n_1 + w_z * n_2 \tag{1b}$$

The control-mesh usually provides a good approximation of the original mesh. As a result, the sampling process can simply be implemented by computing the normal-surface intersection v^x, as the sample value v^s. However, sometimes the control-mesh fails to correctly resemble the original surface. In such cases the naïve intersection-based sampling is insufficient. To improve the sampling quality, we consider the neighborhood of the intersection point in calculating the sample value. Let v_0,...,v_7 be the adjacent vertices of v on a control-face f and r_0,...,r_7 be the rays shot from these vertices along their interpolated normals. The intersection points of r_0,...,r_7 with M will be denoted v_0^x,...,v_7^x, respectively. The intersection points $v_0^x, \ldots, v_7^x$ define a rectangular patch, which is used to determine the sampling value v^s. We define the *neighborhood* of v as the surface whose center is v^x, and bounded by v_0^x,...,v_7^x (see Figure 6a).



(a) (b)

Figure 6: The sampling scheme (a) the neighborhood, (b) the piecewise interpolation

The sampling point v^s, can be computed by interpolating the triangles within the neighborhood of v^x. One could bilinearly interpolate the centroids of these triangles weighted by the size of each triangle and its Euclidean distance from v^x. However, it is not easy to distribute the triangle weight into the two factors i.e., the triangle size and its distances from v^x. For that reason, we uniformly subdivide the rectangular patch into roughly equal cells and perform a piecewise interpolation. Each cell is assigned an elevation value, which is computed by averaging the elevation of the triangles that intersect it. Since the cells are roughly the same size, we only need to consider their distance from the intersection point v^x when interpolating their elevation values (see Figure 6b). Note that large triangles may fall in more than one cell, and computed separately for each of these cells. The interpolation of all the cells' values produces the elevation of v.

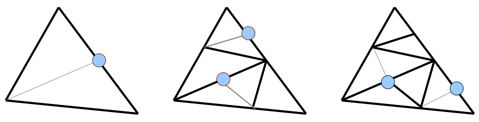
3.2 Generating the Displacement Maps

Our algorithm assigns a single displacement map for every face in the control-mesh. After the sampling pattern has been mapped onto a control-face f, a new displacement-map D_f is created for f such that each vertex v of the subdivided face has an associated elevation value in D_f. The elevation values are generated by sampling the original mesh and the *sampling-error* Δ_f of a face f are determined as $max_{v \in P_f}(min_{u \in f}(\|v-u\|))$, where P_f is the corresponding patch in the original mesh (see Figure 1).

3.3 Runtime Adaptive Subdivision

The runtime stage is executed almost entirely on the GPU, with the CPU acting only as an interface. At each frame, the CPU transmits the control-mesh to the GPU, which recursively subdivides the faces that exceed a certain screen-space projection error. Finally, the GPU elevates each of the refined-vertices according to the displacement-maps assigned to the faces of the input control-mesh.

The CPU transmits the faces of the control-mesh to the GPU, which computes the screen space projection for each face. Let f_p be the projection of the face f and let e_p be the projection of the edge e, which is the longest edge of f_p. The length of e_p, $|e_p|$, is compared against a predetermined screen space tolerance τ. If $|e_p| > \tau$, the face f is subdivided into two new faces, f_a and f_b, by inserting a new vertex v_m in the middle of edge e. The two faces, f_a and f_b, are then sent back to the beginning of the adaptive subdivision process. If $|e_p| \leq \tau$ then f is fine enough, and is sent to the next rendering stage. Note that the generated vertices are a subset of the refined-vertices. The edge e is usually shared with another face g, which will be subdivided at the middle of e by the time the subdivision process is complete. Therefore, by the end of the subdivision process each two adjacent faces have the same vertices along the common edge (see Figure 7), i.e., the final triangulation is crackless.



(a) (b) (c)

Figure 7: Various stages of the subdivision (faint lines depict the current step of the subdivision) (a) a subdivision of a single face, (b) a mid-process triangulation, and (c) the final triangulation

This subdivision scheme results in a semi-uniform screen space subdivision of the control-mesh. The GPU generates a view-dependent adaptive subdivision of the control-mesh, while enabling fine-grained changes that depend on view-parameters. In such a manner, the mesh structure is refined at every frame to adapt to just the right level of detail necessary for visual realism. Therefore, this approach manages to provide better local adaptivity than existing cluster-based rendering schemes.

Performing face subdivision using the screen space projection of edges does not take into account the curvature of the model nor the sampling error. One could argue that the curvature is encoded within the control-mesh since small faces correspond to high curvature and large faces correspond to low curvature. Nevertheless, the local curvature of a control-face, as well as the sampling error, are encoded in its displacement maps and it is important to take them into account. In our approach, the edges guide the face subdivision in order to avoid T-junctions. For this reason, the curvature and error of a face are encoded in its edges.

Let f_a and f_b be the two faces that share the edge e and let h_a and h_b be the maximum elevation value in the displacement maps assigned to f_a and f_b, respectively. The *geometric curvature* Δ_e of the edge e is defined as $max(h_a, h_b) + max(\Delta_a, \Delta_b)$, where Δ_a and Δ_b are the sampling error of f_a and f_b, respectively. When a face f is subdivided into two triangles, the values for the created edges are computed by averaging the previous edges' values (see Figure 8).

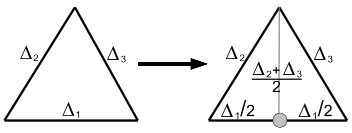


Figure 8: Computing the generated edges errors

During the subdivision process, the product of screen space projection of the longest edge, e_f, and the geometric curvature, $|e_f| \times \Delta_e$, is compared to τ to guide the subdivision process. In such a manner, we consider the local geometry and the sampling error of each face, while generating its adaptive subdivision.

At the final stage of the rendering process, the GPU displaced each vertex, v, according to its assigned elevation value, based on the relative location of v in the face f. Note that the vertices generated by the adaptive subdivision are subset of the refined-vertices of f.

3.4 Optimizations

Fetching the elevation of a vertex v from the displacement-map without considering its adjacent elevation values, may result in missing details in the generated image. To prevent such cases, we have constructed a MipMap hierarchy for every displacement-map. Our algorithm fetches the elevation of a vertex v from the MipMap hierarchy of the appropriate displacement-map. In such a scheme, the adjacent elevation values of a vertex are taken into account in a view-dependent manner.

To improve GPU utilization, a rendering step that removes invisible surfaces in the mesh is performed before applying the GPU-based subdivision step. Since the surface generated for each control-face f is relatively close to f, the visibility check of f is used as the visibility check of its generated surface. The check process, however, takes into account the difference between f and its generated surface by actually checking the visibility of the bounding volume of f's surface, f^v.The bounding volume f^v of f is the triangular-volume received when elevating f using the minimum and maximum elevation values stored in D_f.

4 IMPLEMENTATION DETAILS

In our implementation, we have used the edge-collapse operation with quadric error metric [10] to generate the control-mesh. The visibility-check and the GPU-based subdivision procedures run within the geometry processors, while the elevation procedure is performed by the vertex processors. The GPU-based adaptive subdivision is implemented by using the *stream-out* control, which allows the CPU to terminate the graphics pipeline and emit into a VBO the triangles resulting from one subdivision pass. The GPU performs a recursive subdivision on the processed triangles by switching between two VBOs for each pass.

A triangle is denoted *fine-enough* if it passes the adaptive subdivision test, i.e., it complies with the required screen-space precision and does not need any further subdivision. Transmitting all the control-faces to the GPU using a single VBO and executing the subdivision test uniformly, usually forces many fine-enough triangles to go through the subdivision phase and waste expensive processing cycles. To avoid this waste, we add a rendering pass, after each subdivision, which passes all the fine-enough triangles from the VBO on to the next step of the pipeline.

The size of large models often exceeds the capacity of the available video memory. To support large models, we have implemented an External Video Memory Manager, which uses a single 2D cached texture as a video memory buffer [16].

5 RESULTS

We have tested our implementation using various datasets of different sizes. This section reports experimental results, obtained using an Intel Core 2 Duo processor, 2GB of memory, and an NVIDIA GeForce 8800 GTX with 768MB.

5.1 Preprocessing

We have used sampling-patterns of degrees 17 and 33 that include 256 and 1024 triangles, respectively. To recover the original models using approximately the same number of triangles, the generated control-meshes are 0.4% and 0.1% the size of the original models, respectively.

Model		Memory (*MB*)		Time
Dataset	Size (faces)	Original Model	Sampled Model	(*min*)
A. dragon	7.2M	230	70	39
Lucy	28.1M	1112	158	156
David	56.2M	2113	317	311

Table 1: Preprocessing time, and memory requirement

The results of the offline preprocessing phase are presented in Table 1, which depicts the model size, the memory requirements of the original model, and the sampled model (the control-mesh and its displacement maps). The *time* column reports the offline preprocessing time, which includes the simplification, sampling, and MipMap generation.

The displacement map-based representation reduces the model size by approximately 70%. This is a result of storing a 4*byte* elevation value instead of three 4*byte* values (x, y, and z) for each vertex and without mesh connectivity. The displacement maps of each face are sampled from a relatively close surface (the original surface). Therefore, further reduction in the memory size is achieved by using 2*byte* displacement maps instead of 4*byte* displacement maps. However, using 2*byte* displacement maps may compromise the quality of the recovered model (see Table 3).

The quality of the sampled-mesh is computed by estimating the difference between its surface and that of the original model. We define the average geometric distance d as $(\sum_{v \in M} min_{u \in M^s}(\|v-u\|))/|M|$, where M and M^s are the original model and sampled mesh, respectively.

Dataset	Intersection	Interpolation
A. dragon	0.010	0.009
Lucy	0.024	0.022
David	0.017	0.013

Table 2: The quality of the sampling techniques

Table 2 reports the quality of the sampled-meshes using the two sampling techniques, *intersection* and *interpolation*. Both techniques give small sampling errors, however, the interpolation based sampling provides better quality than the intersection based sampling.

Dataset	Degree 17		Degree 33	
	4*bytes*	2*bytes*	4*bytes*	2*bytes*
A. dragon	0.004	0.004	0.009	0.010
Lucy	0.009	0.010	0.022	0.022
David	0.005	0.006	0.013	0.013

Table 3: The effects of the sampling pattern degrees and elevation values' format on the model quality

Table 3 reports the quality of the sampled-meshes as a function of the degree of sampling-pattern and the depth of the displacement maps. In these experiments the total number of triangles after refinement is similar to those of the original one, i.e., the larger the pattern degree the coarser the control-mesh. Using smaller patterns generates better approximations of the input model. It is easy to conclude that a sampling-pattern of degree 17 and 2*byte* displacement depth provides better quality than a sampling-pattern of degree 33 and 4*byte* displacement depth.

5.2 Runtime Performance

In the reported runtime experiments, we have used a sampling-pattern of degree 33. These results were computed by averaging the performance over a period of 30 seconds of interactive rendering.

Dataset	τ	ε	Proc.	Rendered	fps
A. dragon	2	2.13	45K	18K	220
Lucy	2	2.10	47.5K	19K	220
David	2	2.08	97.5K	39K	154
A. dragon	1	1.22	150K	60K	154
Lucy	1	1.11	190K	76K	81.4
David	1	1.18	200K	80K	81.4

Table 4: Runtime performance

Table 4 presents the runtime performance of our algorithm. The τ and ε columns present the subdivision threshold and the resulted screen-space error of the extracted geometry, respectively. The *processed* and the *rendered* columns present the number of the triangles processed by the GPU and the triangles actually rendered. The *fps* column reports the number of frames generated per second.

It is clear that ε is relatively close to τ, which implies that τ can be used to control the screen-space error. The frame rates (*fps*) are determined by the number of the processed triangles, which is dictated by τ. As a result, similar τ values lead to similar frame rates, regardless of the size of the original model. Refining k control-faces to generate a model with n triangles requires processing at least $2n-2k$ triangles. However, some triangles usually require finer subdivisions, which forces all the triangles to be reprocessed by the geometric processor. For that reason, the column of processed triangles shows a higher factor, 2.5 (on average). In addition, view-frustum and back-face culling are applied to the control-mesh and manage to remove up to 95% of the invisible control-faces.

Our geometric error distribution scheme (see Section 3.3) does not capture all possible cases. To evaluate our error distribution scheme, we have experimentally measured the bias between the actual geometric errors and those computed using our scheme. Figure 9 shows the measured bias using a 56K-triangles control-mesh of the David model. The first column shows that our error distribution scheme matches the actual errors for about 32% of the faces; and about 2% have a bias of 0.5, i.e., the geometric error bias of the resulting triangles is three times more than the error bias of the other triangles.

Figure 10 presents the Asian Dragon model rendered with and without using MipMaps (see Section 3.4). The zoom-in window shows that using Mipmaps provides smoother images. Figure 11 presents screenshots for

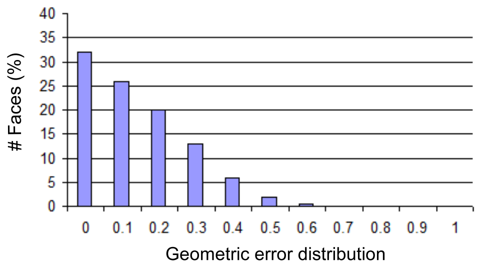


Figure 9: The distribution of the error on the generated faces when subdividing the control-mesh of the David model.

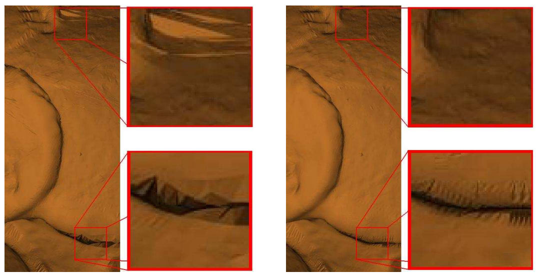


Figure 10: Close-up on the Asian Dragon model (a) the result without MipMaps and (b) the result with MipMaps

the David model at $\tau = 1$. Figure 11c shows that the error in different regions of the generated representations increases in the same rate as the sizes of the subdivision triangles (the green and the red colors represent minimum and maximum errors, respectively).

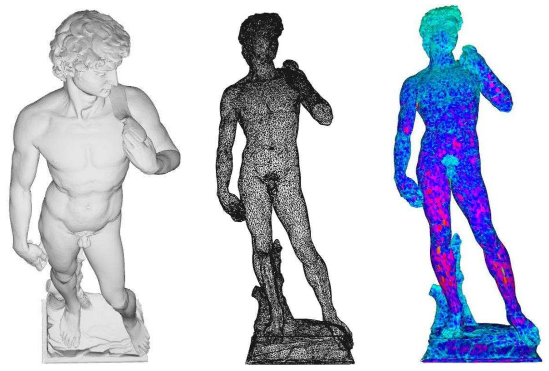


Figure 11: Screenshots of the David model (a) the generated image, (b) the wireframe representation from a different viewpoint, and (c) the geometric error

5.3 Animated Models

Our algorithm supports interactive view-dependent rendering of large 3D models with animation affects, which are applied to the vertices of the control-mesh, using the CPU, at each frame. The deformed representation is transmitted to the GPU, which generates the view-dependent representation. We found that the GPU's performance remains unchanged when deforming the model within the CPU. Since computing the animation transformation is performed by the CPU, it is important to avoid generating a control-mesh that exceeds the CPU capabilities.

5.4 Comparison With Other Algorithms

We compared our algorithm to recent view-dependent cluster-based approaches.

The Quick-VDR [26] and TetraPuzzles [6] algorithms require $1.5n$ *MB* and $2n$ *MB*, respectively, to represent the multiresolution hierarchies of a model of size n *MB*. Our algorithm requires $0.3n$ *MB* to store the same model. Eliminating the use of multiresolution hierarchies enables the support of large datasets without using external memory.

Most cluster-based algorithms do not support local adaptivity for the extracted geometry. Our algorithm uses a GPU-based subdivision to adapt the generated triangulation to the view-parameters. At 2*pixels* error our algorithm requires only 80% of the triangles required for cluster-based algorithms.

Cluster-based approaches render approximately $280M$ triangles per second. Our algorithm manages to process only $88M$ triangles per second ($176M$ with culling algorithms).

6 CONCLUSIONS AND FUTURE WORK

We have presented a GPU-based view-dependent rendering algorithm. The control-mesh, which is a simplified representation of the input model, is an error-guided subdivision of the input model into disjoint patches. The geometry of these patches is encoded into displacement maps. At runtime a view-dependent level-of-detail representation is generated using a GPU-based adaptive subdivision. Such a scheme minimizes the load on the CPU and makes it available for other general purpose computations. Our approach eliminates the need for multiresolution hierarchies of geometry and enables real-time deformation on large geometric models. Our algorithm encodes the geometric errors within the edges of the control-mesh, which generates manifold meshes and stitches the refined faces seamlessly, without adding extra dependencies or sliver polygons.

We observe three possibilities for future work. First, displacement maps which are currently stored as raw images consume significant portions of the video memory. Developing a GPU-based compression scheme for textures could save expensive video memory. Second, in our current method, the animation is applied to the control-mesh, rather than the original mesh, which results in poor accuracy. It may be useful to develop an "animation aware" simplification approach that generates a control-mesh that takes the input animation into account. Third, our current algorithm generates the view-dependent level-of-detail representation from the control-mesh at each frame. It is interesting to utilize temporal coherence among consecutive frames.

REFERENCES

[1] A. Asirvatham and H. Hoppe. *Terrain rendering using GPU-based geometry clipmaps.*, chapter 2, pages 27–45. Addison-Wesley Professional, 2005.

[2] X. Bao, R. Pajarola, and M. Shafae. SMART: An efficient technique for massive terrain visualization from out-of-core. In *Proceedings of Vision, Modeling and Visualization '04*, pages 413–420, 2004.

[3] J. Bolz and P. Schröder. Evaluation of subdivision surfaces on programmable graphics hardware. *Submitted*, 2005.

[4] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. BDAM – batched dynamic adaptive meshes for high performance terrain visualization. *Computer Graphics Forum*, 22(3):505–514, 2003.

[5] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. P-BDAM – planet-sized batched dynamic adaptive meshes. In *Proceedings of Visualization '03*, pages 147–155, 2003.

[6] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Adaptive TetraPuzzles – efficient out-of-core construction and visualization of gigantic polygonal models. *ACM Transactions on Graphics*, 23(3):796–803, 2004.

[7] L. De Floriani, P. Magillo, and E. Puppo. Efficient implementation of multi-triangulations. In *Proceedings of Visualization '98*, pages 43–50, 1998.

[8] W. Donnelly. *Per-Pixel Displacement Mapping with Distance Functions*, pages 123–136. Addison-Wesley, 2005.

[9] C. Erikson and D. Manocha. Hierarchical levels of detail for fast display of large static and dynamic environments. In *Proceedings of symposium on Interactive 3D graphics '01*, pages 111–120, 2001.

[10] M. Garland and P.S. Heckbert. Surface simplification using quadric error metrics. In *Proceedings of SIGGRAPH '97*, pages 209–216, 1997.

[11] G. Gerasimov, F. Fernando, and S. Green. Shader model 3.0 using vertex textures. *Nvidia White Paper*, 2004.

[12] L. Hwa, M. Duchaineau, and K. Joy. Real-time optimal adaptation for planetary geometry and texture: 4-8 tile hierarchies. *IEEE Transactions on Visualization and Computer Graphics*, 11(4):355–368, 2005.

[13] J. Ji, E. Wu, S. Li, and X. Liu. Dynamic lod on gpu. In *Computer Graphics International '05*, pages 108–114, 2005.

[14] J. Kim and S. Lee. Truly selective refinement of progressive meshes. In *Graphics Interface '01*, pages 101–110, 2001.

[15] A. Lee, H. Moreton, and H. Hoppe. Displaced subdivision surfaces. In *Proceedings of SIGGRAPH '00*, pages 85–94, 2000.

[16] Y. Livny, G. Bauman, and J. El-Sana. Displacement patches for gpu-oriented view-dependent rendering. In *Proceedings of GRAPP '08*, pages 181 – 190, 2008.

[17] Y. Livny, N. Sokolovsky, T. Grinshpoun, and J. El-Sana. A gpu persistent grid mapping for terrain rendering. *The Visual Computer*, 24(2):139–153, 2008.

[18] Haik Lorenz and Jurgen Dollner. Dynamic mesh refinement on gpu using geometry shaders. In *Proceedings of WSCG 2008*, 2008.

[19] F. Losasso and H. Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. *ACM Transactions on Graphics*, 23(3):769–776, 2004.

[20] D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. In *Proceedings of SIGGRAPH '97*, pages 199–207, 1997.

[21] Kevin Moule and Michael D. Mccool. Efficient bounded adaptive tessellation of displacement maps. graphics interface 2002. In *In Graphics Interface*, pages 171–180, 2002.

[22] R. Pajarola. Fastmesh: efficient view-dependent meshing. In *Proceedings of Pacific Graphics '01*, pages 22–30, 2001.

[23] A. Pomeranz. ROAM using triangle clusters (RUSTiC). Master's thesis, UNIVERSITY OF CALIFORNIA, 2000.

[24] J. Schneider and R. Westermann. GPU-friendly high-quality terrain rendering. *WSCG*, 14(1-3):49–56, 2006.

[25] T. Ulrich. Rendering massive terrains using chunked level of detail control. In *Proceedings of SIGGRAPH '02*, 2002.

[26] S. E. Yoon, B. Salomon, R. Gayle, and D. Manocha. Quick-vdr: Interactive view-dependent rendering of massive models. In *Proceedings of Visualization '04*, pages 131–138, 2004.