Displacement patches for view-dependent rendering

Yotam Livny, Gilad Bauman & Jihad El-Sana

The Visual Computer
International Journal of Computer Graphics

ISSN 0178-2789

Vis Comput
DOI 10.1007/s00371-011-0609-4

A 20977
Volume 25
Number 1
January 2009
The Visual Computer
International Journal of Computer Graphics
Fourteenth International Multimedia Modeling Conference (MMM2008)
Editor-in-Chief: Prof. N. Magnenat-Thalmann
Guest Editors: Minoru Etoh
Frank Nack
Shin'ichi Satoh
Springer
1

Vis Comput
DOI 10.1007/s00371-011-0609-4

ORIGINAL ARTICLE

Displacement patches for view-dependent rendering

Yotam Livny · Gilad Bauman · Jihad El-Sana

Abstract In this paper we present a new approach for interactive view-dependent rendering of large polygonal data sets which relies on advanced features of modern graphics hardware. Our preprocessing algorithm starts by generating a simplified representation of the input mesh. It then builds a multiresolution hierarchy for the simplified model. For each face in the hierarchy, it generates and assigns a displacement map that resembles the original surface represented by that face. At runtime, the multiresolution hierarchy is used to select a coarse view-dependent level-of-detail representation, which is sent to the graphics hardware. The GPU then refines the coarse representation by replacing each face with a planar tile, which is elevated according to the assigned displacement map. Our results show that our implementation achieves quality images at high frame rates.

Keywords GPU processing · Level-of-detail rendering · View-dependent rendering · Subdivision surfaces

1 Introduction

Polygonal meshes dominate the representations of 3D graphics models due to their compactness and simplicity. Recent advances in design, modeling, and acquisition technologies have simplified the creation of 3D models, which has led to the generation of large 3D models. These models consist of millions of polygons and often exceed the rendering capabilities of advanced graphics hardware, which necessitates reducing their complexity to match the hardware's rendering capability, while maintaining visual appearance. Numerous algorithms have been developed to reduce the complexity of graphics models. These include level-of-detail rendering with multiresolution hierarchies, occlusion culling, and image-based rendering.

View-dependent rendering approaches modify the mesh structure at each frame to adapt to the appropriate level of detail. Traditional view-dependent rendering algorithms rely on the CPU to extract a level-of-detail representation. However, within the duration of a single frame, the CPU often fails to extract the frame's geometry. In addition, communication between the CPU and the graphics hardware often forms a severe transportation bottleneck. These limitations usually result in unacceptably low frame rates.

Cluster-based multiresolution algorithms overcome the CPU limitations by subdividing the data set into disjoint regions called *clusters* or *patches*, which are simplified independently. These algorithms manage to reduce the time required to extract an adaptive level of detail. However, the partition into patches often does not take into account fine object space error.

In this paper, we present a novel cluster-based multiresolution approach and an efficient view-dependent rendering algorithm. In an off-line stage, our algorithm simplifies the input model to reach a simple representation—the *control-mesh*. It then constructs a multiresolution hierarchy for the control-mesh. For each triangle in the hierarchy, it stores a compact representation of an original model's patch.

The currently available graphics hardware supports several functionalities, such as a programmable pipeline and vertex texturing. The availability of these features drives the development of new algorithms and techniques to utilize

Y. Livny · G. Bauman · J. El-Sana (✉)
Ben-Gurion University of the Negev, Beersheba, Israel
e-mail: el-sana@cs.bgu.ac.il

Y. Livny
e-mail: livnyy@cs.bgu.ac.il

G. Bauman
e-mail: baumang@cs.bgu.ac.il

Published online: 13 July 2011

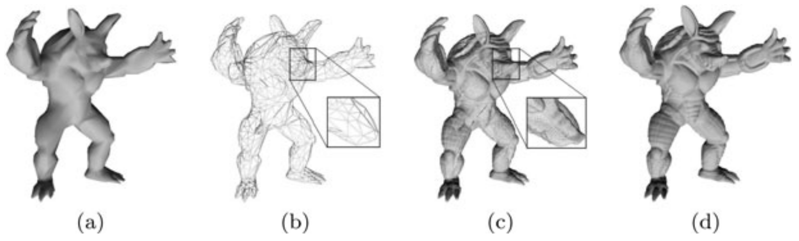


Fig. 1 The wireframe and shaded representation of a coarse front of the Armadillo model and the refined representation

these capabilities. Our algorithm utilizes graphics hardware to efficiently represent patches as displacement maps, which are generated by sampling the surface of the original model. At runtime, the CPU extracts a coarse view-dependent level-of-detail representation from the multiresolution hierarchy and transmits it to the graphics hardware. The GPU refines each triangle by replacing it with a triangular parametric patch and then elevates the vertices using the assigned displacement map.

Our approach utilizes several advanced techniques in the field of level-of-detail rendering and GPU programming, in order to provide quality GPU-based view-dependent rendering. It utilizes an error-guided subdivision to partition the input model into disjoint patches that are compactly encoded using displacement maps. Processing the control-mesh on the CPU and applying mesh refinement on the GPU dynamically balances the rendering load between the CPU and the GPU.

Our approach also manages to seamlessly stitch adjacent patches into a manifold mesh without adding extra dependencies or sliver triangles. Caching geometry as templates of indexed triangle strips and efficiently utilizing the parallel nature of the GPU and its programmable processors, enable our algorithm to leverage the graphics hardware to its fullest.

While our approach is similar to that of Lee et al. [1], their work did not support modern GPU compatibility, as it was unavailable at the time. We provide an efficient algorithm to construct the control-mesh and prove its correctness. In addition, we integrate an adaptive LOD representation for efficient real-time rendering.

This paper presents a summary of the research in [2] in an archival form and augments it with a correctness proof for the control-mesh generation and additional improvements on model representation.

2 Related work

View-dependent rendering schemes usually rely on multiresolution hierarchies that encode various levels of detail of the original model. Earlier approaches [3–8] assume that the multiresolution hierarchy fits entirely into local memory and the extraction of adaptive levels of detail is performed within the CPU.

Several approaches accelerated view-dependent rendering by reducing dependencies limitation, which is used to validate the split and merge operations [9], or integrating occlusion culling within the view-dependent rendering frameworks [10, 11]. To handle large data sets that do not fit in local memory, external memory view-dependent rendering algorithms were developed [12, 13]. In addition, a simplification method that represents geometric details using bump maps was developed by Cohen et al. [14] such that it used displacement mapping for NURB surfaces. Finally, Lee et al. [1] have presented a simple, automatic scheme for converting detailed geometric models into a representation that defines both the domain surface and the displacement function using a unified subdivision framework demonstrating that displaced subdivision surfaces offer a number of benefits.

As the rendering capability of graphics hardware improves and the size of data sets increases, the extraction of appropriate levels of detail within the duration of a single frame becomes impractical for the CPU. To overcome this limitation, cluster-based approaches have been introduced [15–17].

The advances in graphics hardware have led to the development of algorithms that inherently utilize the introduced hardware capabilities. Several approaches used the fragment processor to perform mesh subdivisions [18, 19]. The vertex processor was used to interpolate different resolution meshes in a view-dependent manner [20], deform displacement maps [21], and map relief textures onto polygonal models [22]. Displacement maps and the fragment processor were also used to accelerate image-based rendering [23, 24].

Most GPU-based level-of-detail algorithms were designed for height fields and terrain data sets [25–31]. Little work has been done to deal with general 3D models. The programmable GPU and displacement maps were used to approximate general meshes [32, 33] and for adaptive real-time rendering [34–36]. Livny et al. [2] developed a GPU-based adaptive real-time rendering, which is the short conference version of this paper.

2.1 Background

We will now briefly overview a parametric *side-vertex* scheme, which is used to fit a piecewise smooth surface to a

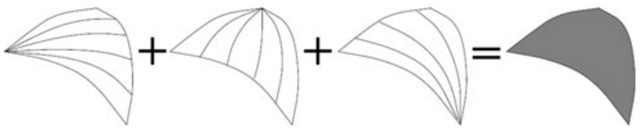


Fig. 2 The side-vertex scheme

triangulated set of points. We use this method to construct a smooth parametric patch for a given triangle.

The side-vertex scheme, initially developed by Nielson [37] and later improved by Mann [38] and Mao et al. [39], aims to interpolate the three vertices of a triangle using their positions and normals. The scheme is applied in three main steps:

– Three boundary curves are constructed, each corresponding to an edge of the input triangle, using the positions and normals of the two vertices.
– Three patches are created, one for each boundary curve and its opposite side. The interior of each patch is defined by the curves passing from each point along the boundary curve to the opposite vertex as shown in Fig. 2.
– The final patch is created by blending the three side patches.

3 Our approach

In this section we present our framework for interactive rendering of large polygonal data sets that utilizes modern GPU capabilities, such as programmability, vertex texturing, and geometry caching. In a preprocessing stage, our algorithm simplifies the input mesh to generate a *control-mesh*, μ, and then builds a multiresolution hierarchy for μ. For each triangle t in the multiresolution, the algorithm constructs and assigns a displacement map that resembles the original surface's patch, corresponding to t. At runtime, the hierarchy is used to select an appropriate view-dependent level-of-detail representation (of the control-mesh)—the *front* mesh. At each frame, the front is sent to the graphics hardware for rendering. The GPU then refines the front triangles by replacing each triangle t with a cached triangular-planar mesh, which we call the *generic tile*. The generic tile is mapped onto a triangular parametric patch determined by the vertices and normals of t. Finally, the vertices of the triangular mesh are elevated according to the assigned displacement map (see Fig. 3).

3.1 Terminology

In this section we present an overview of our terminology to simplify the discussion and the presentation of the algorithm.

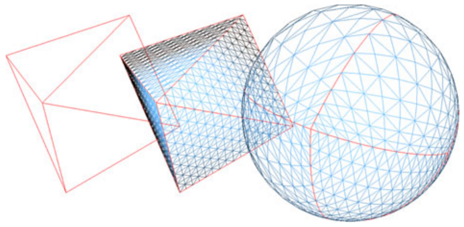


Fig. 3 An 8-triangle front for a sphere model, its generated subdivision, and its refined representation

Let us define the normal n_i at an internal point q_i in the triangle t as the interpolation of the three normals at its vertices according to their distances from q_i. We shall refer to the normal n_i as the *interpolated normal*. Let M be a polygonal mesh, and M^s be a simplified version of M. The vertex $u_i \in M^s$ defines its corresponding point x_i on M by shooting a ray from the vertex u_i along the normal n_i toward the mesh M. An edge $\overline{u_i, u_j} \in M^s$ is projected onto a polyline $x_i, \dots, x_k, \dots, x_j$ on M, where x_i and x_j correspond to u_i and u_j, respectively, and internal points x_k correspond to sample points, u_k, along the edge $\overline{u_i, u_j}$. The triangle $t_i \in M^s$ with vertices $\{u_i, u_j, u_k\}$ determines a patch p_i on M, which is defined by the three intersection points x_i, x_j, and x_k. If the patches $p_0, \dots, p_l$ do not overlap, and every point in M is covered by exactly one patch (except the boundary polylines), we say that M^s *corresponds* to M, and the triangle $t_i \in M^s$ *corresponds* to the patch $p_i \in M$, which is defined by t_i.

A patch p_i is denoted an *elevation* of its corresponding triangle t_i if and only if for any point $q_i \in t_i$, a ray shot along its normal intersects the patch p_i at exactly one point (see Fig. 4). A polygonal mesh M is an *elevation* of another polygonal mesh M^s if and only if M^s *corresponds* to M and every patch $p_i \in M$ is an *elevation* of its *corresponding* triangle $t_i \in M^s$. A simplification algorithm *preserves elevation* if the input mesh is an *elevation* of every approximation generated by this algorithm.

We define the cone of normals C as a triplet (N, n, θ), where N is the set of normals, $\{n_0, n_1, \dots, n_k\}$, $n = \mathit{Average}(N)$ is the average of the normals in N, and $\theta = \mathit{ConeAngle}(N)$ is the angle of the cone, whose center is n and bounds the normals in N. We denote C^i_n, C^i_θ, C^i_N as the normal, the angle, and the set of normals of the cone C^i, respectively. The union of two cones C^a and C^b is computed by unifying their normal sets and then computing the average normal and the cone angle for the resulting normal set; i.e., $C = C^a + C^b$, where $C_N = C^a_N \cap C^b_N$, $C_n = \mathit{Average}(C_N)$, and $C_\theta = \mathit{ConeAngle}(C_N)$.

3.2 Model simplification

In typical view-dependent rendering algorithms, the CPU selects the appropriate level-of-detail representation. However, the CPU is often incapable of traversing and updating

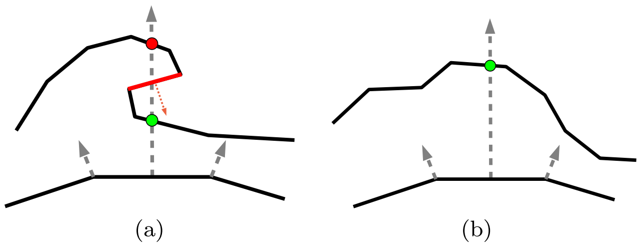


Fig. 4 The preserve elevation property: (**a**) not preserving elevation, and (**b**) preserving elevation

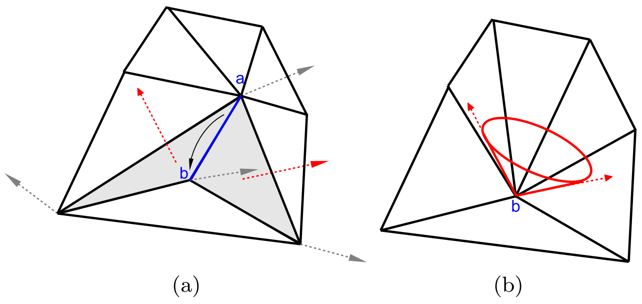


Fig. 5 Determining the validity of an edge-collapse: (**a**) The edge to be collapsed ($a \to b$) is marked with *blue*. The *red arrows* are the normals of the triangles that will become degenerate as a result of the edge-collapse. (**a**) The resulting mesh and the normal cone of vertex b (in *red*), which encodes the normals in the cones of a and b and the normals of the newly degenerated triangles

large adaptive meshes within the duration of a single frame. For that reason, our algorithm simplifies the input model to reach a compact control-mesh and lead to the generation of a relatively small multiresolution hierarchy.

Our framework requires the use of a simplification algorithm that *preserves elevation* to generate a control-mesh. This requirement is essential to define a correspondence between the triangles of the control-mesh and the patches of the original mesh, where every patch is an elevation of its corresponding triangle (see Fig. 3).

In order to generate a control-mesh, the straightforward way would have been to simply use the half-edge collapse operator and the quadric error metric [40]. However, such a naive approach does not preserve elevation and may risk obtaining a control-mesh that cannot recover the original model. Therefore, our algorithm uses a slightly different approach. Given a mesh M^i and an edge $e = \overline{(a, b)} \in M^i$, let M^{i+1} be the mesh resulting from the collapse of a onto b. A triangle on the control-mesh, M^c, samples its corresponding patch, as described in Sect. 3.7. A violation of the elevation property, i.e., a sampling ray intersects the patch at more than one point, indicates the existence of regions that cannot be sampled (see Fig. 4). The projection (inverse sample rays) of a triangle $t \in M$ onto the control triangle determines its sampling region, s_t. If the triangle t faces the sampling region then the elevation property is violated, i.e., $r_s \cdot n_t < 0$, where r_s is an interpolated normal in s_t and n_t is the normal of t.

In our approach the control-mesh is generated incrementally, which prohibits relying on the normal of a triangle and its sampling regions to maintain the elevation property (they have different scopes). The collapse of edge $e = \overline{(a, b)}$ (the collapse of vertex a into b) is performed by contracting the edge e, removing the degenerate triangles, and unifying the cones of its vertices. An edge-collapse is valid if it satisfies the following:

1. The angle between any normal in the union of cones of a and b, and any normal of a degenerated face is less that $\pi/2$; i.e., $n_t \cdot n_j > 0\ \forall t \in D$ and $\forall n_j \in C_N^a \cap C_N^b$, where n_t is the normal of the triangle t and D is the set of the degenerated faces (see Fig. 5).
2. The angle of the resulting cone C does not exceed half sphere; i.e., $C_\theta < \pi/2$, where $C = C^a \cap C^b$.

Executing only valid edge-collapses during the simplification process ensures that the input mesh is an elevation of the generated control-mesh (see Sect. 3.3). Practically, the edge-collapse operation gradually flatten the patch until it reaches one triangle—the control triangle.

For a given mesh M we initialize the normal of each vertex to the average of the normals of its adjacent triangles. Then, the simplification algorithm executes valid half-edge collapses, ordered by their quadric error value, one by one. The algorithm proceeds until it reaches the target polygon count, or until no valid collapses remain. The resulting mesh is used as the control-mesh for the construction of the multiresolution hierarchy. To generate a multiresolution hierarchy for the control-mesh, one can use any of the previously developed schemes (see Sect. 2).

3.3 Correctness proof

In this section we prove that if a control-mesh M^c is generated from M by executing only valid half-edge collapses, then M is an elevation of M^c. The proof of correctness is a carried out through the following claims.

Claim 1 *Let M be a closed simple manifold mesh (polyhedron) and let r be a ray starting from an internal point p_r at direction d_r and intersecting M at two points, p_a and p_b, which belong to the two triangles t_a and t_b, whose normals are n_a and n_b, respectively. If $n_a \cdot d_r > 0$ and $n_b \cdot d_r > 0$, then there exists a triangle $t \in M$ such that $n_t \cdot d_r \leq 0$.*

Proof Let us assume without loss of generality that the ray does not pass through a vertex or an edge. Since the ray starts from an internal point and travels out of the polyhedron, an intersection with a triangle t that satisfies $n_t \cdot d_r > 0$ indicates that the ray is leaving the polyhedron (exit point), and an intersection that satisfies $n_t \cdot d_r \leq 0$ indicates that the ray is entering the polyhedron.

Since the mesh is a closed simple manifold, between two entering points there is an exit point, otherwise the mesh is either not manifold with consistent normals or is not closed. □

Claim 2 *Let M be a closed simple manifold mesh (polyhedron) and let r be a ray that starts from an internal point p_r. If each intersection between the ray r and the mesh M satisfies $n_x \cdot d_r > 0$, where d_r is the direction of the ray r and n_x is the normal of the intersected triangle, then the ray intersects the mesh at only one point.*

Proof We carry out the proof by contradiction. Let us assume that the ray r intersects the mesh, M, at two points, p_a and p_b, that satisfy Claim 1. According to Claim 1, there exists a point p_i between p_a and p_b, along the ray r, that satisfies $n_i \cdot d_r < 0$, where n_i is the normal of the triangle that includes p_i. However, this contradicts the claim assumption. □

Claim 3 *Let t and P_t be a control triangle t and its corresponding patch, respectively, and let n be any interpolated normal on t that intersects P_t at p_x on the triangle t_x. If $n \cdot n_x > 0$, where n_x is the normal of the triangle t_x, then P_t preserves elevation with respect to t.*

Proof Let us assume without loss of generality that the interpolated normal does not intersect any vertex or edge on the mesh P_t. Let us extend the control triangle and the patch by an ϵ to avoid dealing with the edges of the triangle as a special case.

Let us construct a closed polyhedron H_m^t using the control triangle, t, its corresponding patch, P_t, and the mesh that connects the edges of the triangle with the boundary of the corresponding patch. We also flip the normal of t to generate consistent normals on the surface of the polyhedron.

An interpolated normal n is a ray that starts from the surface of the control triangle. If we ignore the point on the triangle, we can view n as a ray that starts from within the polyhedron. Based on Claim 1, n intersects the surface at a single point, which means that P_t preserves elevation with respect to t. □

Theorem 1 *Let M be a manifold mesh. If a control-mesh, M^c, is generated from M by executing valid edge-collapses, then M is the elevation of M^c.*

Proof We prove the theorem by contradiction. Let us assume that M is not an elevation of M^c, which means that there is an interpolated normal, n, in a control triangle t that intersects its corresponding patch M_t twice—at p_a and p_b on the triangles t_a and t_b, whose normals are n_a and n_b, respectively.

One of these two points is an exit point or there is an exit point between them (based on Claim 1), along the interpolated normal. Let us assume without loss of generality that the control triangle is not the evolution of either t_a or t_b. Therefore, the two triangles became degenerate during the reconstruction of the control-mesh, so let us assume t_a becomes degenerate before t_b. Let e_b refer to the edge adjacent to t_b, whose collapse made the triangle t_b degenerate. To complete the proof we distinguish between two cases:

1. The normal n_a considered at the collapse e_b, but this contradicts the validity of the edge-collapse, as $n_a \cdot n_b < 0$.
2. The normal n_a is not considered at the collapse e_b, which implies the existence of an edge-collapse that merges the two normal cones—one includes n_a and the other includes n_b. The resulting cone exceeds half sphere, which contradicts the validity of the edge-collapse (the second requirement). □

We have shown that our algorithm requires a control-mesh that is preserving elevation. We have described a method for acquiring such a control-mesh, M^c, given an input mesh M. Lastly, we have proven the correctness of our method. Therefore, we now have a valid, elevation preserving control-mesh M^c.

3.4 Generic tile structure

The *generic tile* is an equilateral triangle uniformly subdivided—it has the same number of vertices along each of its edges (see Fig. 6). We shall refer to the number of vertices along an edge of a generic tile as the *degree* of the tile. A generic tile of degree k has $(k-1)^2$ triangles and $k(k-1)/2$ vertices.

The generic tile is used in two different stages of our framework, the off-line preprocessing and the real-time rendering (see Sects. 3.6 and 3.8). These two stages involve a geometric transformation that maps the generic tile to match the scale and orientation of the processed triangle, as shown in Fig. 6. We position the generic tile such that the three corner vertices of the generic tile are $(1,0,0)$, $(0,1,0)$, and $(0,0,1)$, and the coordinates of the remaining vertices are determined uniformly.

Our algorithm uses the *side-vertex* interpolation scheme to reduce the displacement values, which are stored using compact maps. Each triangle t in the control-mesh has three vertices and three normals. A parametric triangular smooth patch B_t is computed for t to provide a parametric approximation of the input model, as shown in Fig. 6.

Given an input control triangle t, the transformation $\mathcal{T}$ maps the vertices of the genetic tile onto t using (1), where w_x, w_y, and w_z are the coordinates of the tile's vertex w;

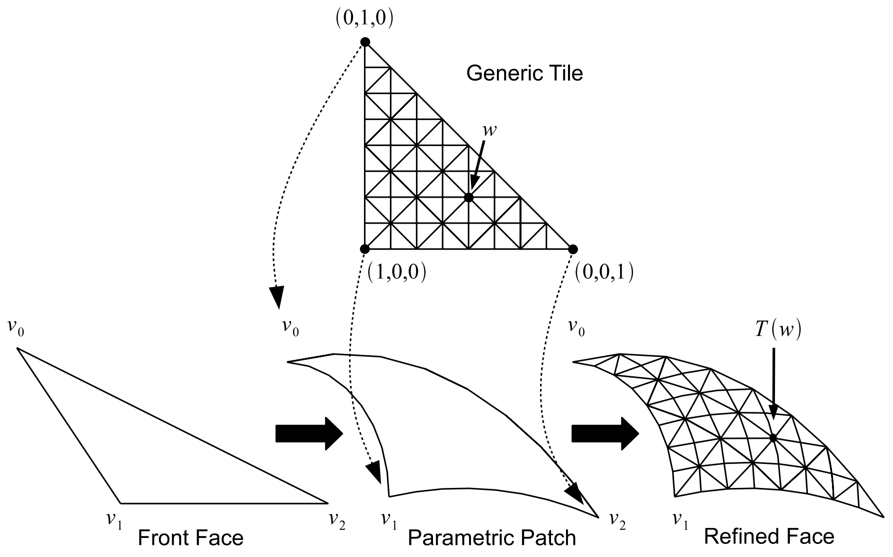


Fig. 6 The mapping of a generic tile onto a front triangle and the mapping of a tile vertex w to $\mathcal{T}(w)$ on the refined front triangle

and v_0, v_1, and v_2 are the vertices of the control triangle t: $(\mathcal{T}([0,1,0]) = v_0$, $\mathcal{T}([1,0,0]) = v_1$ and $\mathcal{T}([0,0,1]) = v_2)$. The normals of the mapped vertices are calculated in a similar manner using (2). The barycentric coordinates of each vertex are used to calculate its corresponding value on the parametric patch, B_t. This scheme maps the vertices on an edge $\overline{v_i, v_j}$ of a control triangle onto the curves defined by the positions of the vertices v_i and v_j and their normals.

$$T(w) = w_x * v_0 + w_y * v_1 + w_z * v_2 \tag{1}$$

$$N(w) = w_x * n_0 + w_y * n_1 + w_z * n_2 \tag{2}$$

3.5 Parametric patch

For any non-degenerate triangle, t, with three vertices $v_i = (x_i, y_i, z_i)$, $i = 0, 1, 2$, any point $v = (x, y, z)$ on the triangle could be expressed as $v = uv_0 + vv_1 + wv_3$, where u, v, and w are the barycentric coordinates and $u + v + w = 1$.

We have adopted the side-vertex scheme to represent the triangle patch. The curves are constructed using the curve operator $c[v_0, n_0, v_1, n_1](t)$, where $c(0) = v_0$, $c(1) = v_1$, and n_0 and n_1 are the normals at v_0 and v_1, respectively. The curve operator constructs a cubic Hermite curve, which interpolates the two end vertices and their derivatives. The derivatives are computed using (3) and (4), as in Neilson's original scheme [37]. However, we determine the scalar parameters a and b using the values sampled from the mesh patch along the edge connecting the two vertices. The values of a and b are determined by forcing the extreme points (maxima, minima, or points of inflection) of the curve to coincide with the sampled values.

$$c'(0) = a(n_0 \times (v_1 - v_0) \times n_0) \tag{3}$$

$$c'(1) = b(n_1 \times (v_1 - v_0) \times n_1) \tag{4}$$

We compute the three side-vertex parametric patches using (5) and (6), where b_{i-1}, b_i, and b_{i+1} are the barycentric coordinates and $\widehat{n_i}$ is the interpolated normal at $\widehat{v_i}$. Each patch, β_i, corresponds to an edge and its opposite vertex, v_i.

$$\widehat{v_i} = c[v_{i-1}, n_{i-1}, v_{i+1}, n_{i+1}]\left(\frac{b_{i+1}}{1 - b_i}\right) \tag{5}$$

$$\beta_i(b_{i-1}, b_i, b_{i+1}) = c\big[v_i, n_i, \widehat{v_i}, \widehat{n_i}\big](1 - b_i) \tag{6}$$

The three side-vertex patches are blended to yield the final parametric patch using (7).

$$G(b_0, b_1, b_2) = \frac{b_1 b_2 \beta_0 + b_0 b_2 \beta_1 + b_0 b_1 \beta_2}{b_0 b_1 + b_1 b_2 + b_2 b_0} \tag{7}$$

3.6 Displacement maps

In the preprocessing stage, a displacement map is assigned to each triangle of the multiresolution hierarchy to enable the recovery of the corresponding patch in real-time rendering. The generation of these displacement maps is performed using the generic tile.

For a generic tile of degree k, we define a set of displacement maps, each containing $k(k-1)/2$ values—as many as the number of vertices in the generic tile. The generic tile is mapped onto an input triangle t and an elevation value is assigned to each vertex v_i of the mapped tile. The elevation value of v_i is determined in four steps:

1 The barycentric coordinates of v_i are used to determine $G(v_i)$, which is the corresponding point on the surface of the parametric patch B_i.
2 A ray is shot from v_i along its interpolated normal n_i and the intersection with the patch p_i is computed; let us refer to this point as x_i.
3 The signed distance between x_i and $G(v_i)$ is taken as the elevation of v_i, i.e., $d_i = x_i - G(v_i)$.

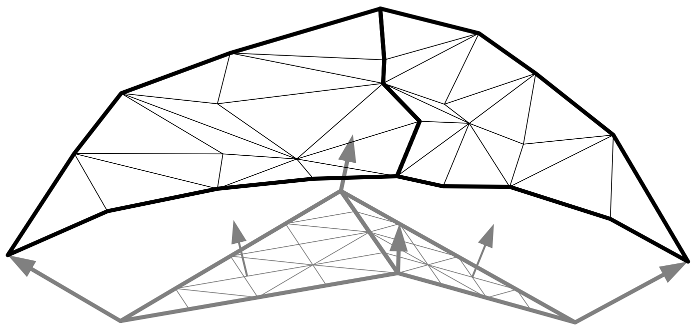


Fig. 7 The refinement of the common edge of two adjacent triangles results in exactly the same polyline

4 The distance d_i is quantized and stored in the displacement map, i.e., $D_t(v_i) = round(d_i/d)$, where d is the quantization factor. For each displacement map we store the largest difference between the displacement map and the patch. We refer to this difference as the *sampling-error* (see (8)).

$$\epsilon_i = \max_{v \in p_i}\left(\min_{v^s \in p_i^s}\left(\|v - v^s\|\right)\right) \tag{8}$$

As a result of using an error-guided subdivision to obtain the patches, the values and the variance in D_t, as well as the generated samples, are usually very small. Nevertheless, considerable reduction in memory usage is obtained by storing sampled values in compressed textures (see Sects. 4 and 5 for technical details and runtime performance).

The vertices along the boundary edges of B_t are determined by the vertices of t and their corresponding normals (refer to (1)). As a result, a refined vertex along a shared boundary has exactly the same position and normal in all its included patches. Such a scheme provides a common polyline for each two adjacent patches (see Fig. 7).

It is important to note that the rays sample the patch according to its control-face—the sampling ray initiated from a point on the control face intersects the patch at one point (see Sect. 3.3). The use of the parametric patch aims to decrease the displacement values, which reduce the memory required to store the generated displacement maps. The parametric patch goes above and below the geometric patch, which inherently is handled by the displacement maps (signed values). In general, cubic parametric curves (and patches) can encounter inflection points, cusp, or loop, which are mathematically detected based on the parametric representation. These undesirable points cannot occur in our parametric patch because we determine the parameters a and b in (3) and (4) per patch, where they aim to reduce displacement values and avoid undesirable properties, such as the existence of inflection points, cusp, and loop.

3.7 Patch sampling

Generating a displacement map for an input triangle involves ray shooting and computing the intersection of the ray with the corresponding patch, P_i. Practically, this is a sampling procedure and according to Shannon's theorem [41], for faithful sampling of a function f, one is required to sample f with more than double the frequency of the highest-frequency component of f. However, in our approach the sampling resolution is dictated by the degree of the generic tile and the sampled mesh is piecewise linear.

To overcome the above sampling limitation, we consider the neighborhood of the intersection point between the ray and the corresponding patch. Our interpolation-based sampling algorithm interpolates the values within the neighborhood δ of v. To compute such an interpolation, one could bilinearly interpolate the values corresponding to the centroids of the triangles $c(t)$ within δ weighted by $w(t)$, which depends on the size of the triangle t and its Euclidean distance from v (see (9)). However, it is not easy to distribute the triangle weight into the two factors: the weight of a triangle is proportional to its size, and inversely proportional to its distance from p. To resolve this quandary, we impose a rectangular grid over the sampled regions, whose boundary is defined by the intersection points corresponding to the eight vertices adjacent to v_i (see Fig. 8(a)). For each cell on the grid, we average the heights of the centers of its triangles and assign its weight according to its distance from the intersection point. Finally, to compute the elevation of v, we interpolate the heights according to the weights of the cells (see Fig. 8(b)). The benefits of our interpolation method can be seen in Fig. 12.

$$p = \sum_{t_i \in \delta(p)} w(t_i)\mathbf{c}(t_i) \tag{9}$$

3.8 Real-time rendering

In real-time, the CPU relies on the multiresolution hierarchy to extract appropriate fronts, based on view-parameters and illumination. The front should be detailed enough to faithfully represent the visualized model with respect to the view-parameters, yet coarse enough to be extracted within the duration of a single frame.

In each frame, the adaptation process, which runs on the CPU, traverses the front nodes. For each node n in the front,

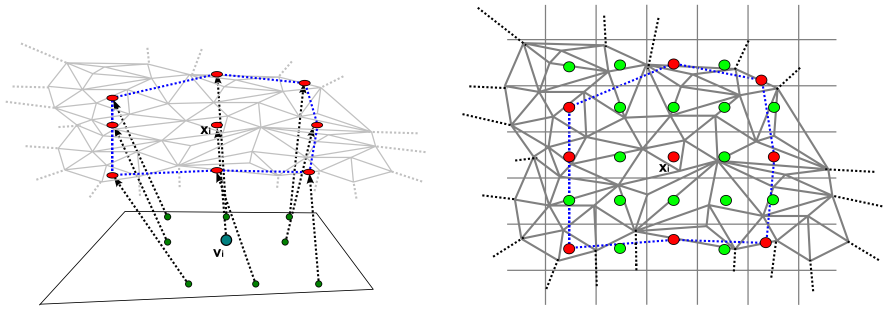


Fig. 8 Our sampling technique: (**a**) Rays are shot from v_i and its surrounding vertices towards the original model. The projections of the neighboring vertices onto the original model define a polygon (blue *dotted line*) which determines the region used to sample x_i. (**b**) The sampling region is subdivided into a regular grid

the CPU determines whether n needs to be refined, simplified, or remain in its current level. The updated front is then sent to the graphics hardware for rendering.

Within the graphics hardware, a single instance of the generic tile is cached in the video memory. In each frame, the received front is refined by mapping the generic tile onto the control triangles and displaced according to the corresponding parametric smooth triangular patch and the assigned displacement map. This procedure is performed in the following steps.

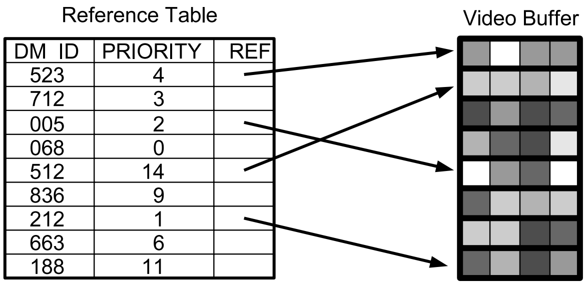
1 The position of the refined vertices and their interpolated normals are determined by (1) and (2), respectively.
2 For each vertex v_i of the refined triangle, the corresponding point, b_i, on the surface of the parametric patch is computed.
3 The displacement d_i, corresponding to the v_i, is fetched from the assigned displacement map.
4 The point b_i is displaced by d_i along the normal of v_i and is taken as the updated position of the vertex v_i.

As a result, each triangle is replaced with $(k-1)^2$ triangles, where k is the degree of the generic tile (refer to Sect. 3.4). The total number of rendered triangles is $(k-1)^2 \times n$, where n is the number of triangles in the front. In order to maintain interactivity, the total number of rendered triangles should not exceed the rendering capabilities of the graphics hardware.

3.9 External video memory

The size of large polygonal data sets often exceeds the capacity of the main memory. Even though we assume that the control-mesh fits in the main memory, the displacement maps for large data sets may not fit in the video memory. Handling large data sets usually involves two uploading stages—uploading from an external media into the main memory and uploading from the main memory into the video memory. Uploading data from an external medium into the main memory has been widely studied (see Sect. 2), and is beyond the scope of this paper.

The limited size of the video memory in current hardware calls for the design of schemes that load data from the main



DM ID	PRIORITY	REF
523	4	
712	3	
005	2	
068	0	
512	14	
836	9	
212	1	
663	6	
188	11	

Fig. 9 The EVMM

memory into the video memory. Our External Video Memory Manager (EVMM) uses a single texture as a buffer, and manages data replacement.

The EVMM allocates a video buffer B, which is implemented as a texture that fits into the video memory. Each cached displacement map is stored as one row in B. In such an approach, it is possible to cache a large number of displacement maps within a single texture, which improves real-time performance by preventing memory paging when using different displacement maps. For every cached displacement map, the EVMM maintains an *id*, a *priority*, and an *index*. The *id* is the displacement map identifier, *priority* measures the probability of reusing the displacement map, and *index* is a reference to B where the data is cached (see Fig. 9). The EVMM caches displacement maps required for the next frame. In addition, it caches displacement maps assigned to several tree levels above and below the front, based on the available memory space. The priority of a displacement map is determined by its distance from the front along the levels of the hierarchy and its assigned error ϵ (refer to Sect. 3.6). When a replacement is required, the entry with the lowest priority is replaced by the newly loaded displacement map.

4 Implementation details

Our algorithm was implemented in C++ and Cg. We have adopted view-dependence trees (VDT) to maintain a multiresolution hierarchy with displacement maps and support view-dependent rendering. We shall denote the VDT with displacement maps *DM-VDT*. The quadric error metric [40] is used to order the execution of the half-edge collapses. In order to use the suggested algorithm, the system must support vertex texturing within the GPU (Shader model 3.0 and above).

A single generic tile, which is cached in the video memory as a Vertex Buffer Object (VBO), is stored in two arrays. One array encodes the vertices' coordinates and the other encodes its triangle-strip order, for efficient rendering.

Our algorithm, which generates the control-mesh, automatically builds correspondence between the control triangles of patches on the original mesh. The control triangle, the normals at its vertices, and the furthest points of the patch determine a prism, μ_t, that bounds the patch B_t corresponding to the control triangle t. During the construction of the control-mesh, we store the order of the half-edge collapse in a directed graph. The triangles of the patch B_t are determined by traversing the vertices of the directed graph in the reverse order of their collapses, starting from the vertices of the triangle. The traversal is guided to explore vertices within the bounding prism corresponding to the triangle t.

Sampling the original model using the control-mesh and the generic tile could be expensive for large triangle meshes—$O(n \times m)$ time complexity to sample a single patch, where m is the number of triangles in the patch and n is the number of samples. We use the generic tile to subdivide the patches into cells, such that each cell includes a small number of triangles, bounded by a constant. This improvement leads to an average time complexity of $O(n)$. Additional improvement is achieved by using the graphics hardware to quickly sample patches in parallel.

Since all displacement maps are of the same size, they can easily be stored in a two-dimensional array, in which each displacement map occupies a single row of elevation values. Such a scheme avoids storing triangular maps in a rectangular texture, which simplifies memory management. In order to reduce memory usage, each displacement map is encoded using standard JPEG compression and is stored in the main memory. At each frame t, the displacement maps, which are necessary to render t, are decoded and cached onto the video memory.

Using lossy JPEG compression for elevation values introduces cracks to the boundaries of adjacent displacement maps. We overcome this limitation by modifying the decoded displacement values before caching them. Two adjacent displacement maps may be decoded and cached on the same frame or in two different frames. If they are decoded on the same frame, we determine the values at the shared boundary by averaging the values of the two boundaries; when they are decoded in two different frames, we replace the shared value at the boundary of the newly decoded displacement map by the cached values from the older displacement map. This approach prevents updating cached displacement maps.

To maintain the original visual appearance, we enable the refinement process to recover the original surface using approximately the same number of triangles. It is important to note that the recovered surface is a sampling of the original one, which may result in small differences between them.

To achieve quality images, the screen-space error must be determined for the selected front. Given a screen-space error τ and a generic tile of degree k, one can use $k\tau$ as the screen-space error to determine the front. The refinement of the front leads to a final screen-space error τ. Unfortunately, in such a scheme, the actual screen-space error may exceed τ as a result of under-sampling. To overcome this limitation, we consider the sampling error ϵ during front extraction. The screen-space error of a triangle t depends on its distance from the viewpoint, its orientation, and its sampling error, ϵ.

In typical rendering applications, back-face and view-frustum culling are used to accelerate the rendering process. The built-in culling mechanisms can only be applied to the triangles of the final mesh, which are created by the refinement process. We implemented the culling on the control-mesh level. The normals at the control triangles are used to guide the back-face culling. Such an approach significantly reduces the number of triangles processed by the GPU.

The suggested EVMM scheme simplifies the management of multiple displacement maps within a single video buffer. This scheme avoids texture switching when accessing different displacement maps. When the displacement maps of a model exceed the maximum size of a single texture (video buffer), an expensive texture update takes place at runtime and reduces the algorithm performance. For that reason, we used a texture array, which is supported as of Shader model 4.0, to store multiple textures.

Displacing the vertices on a triangle within the vertex processors in parallel prevents computing the normal as well as the illumination of each vertex in the vertex processor. This problem can be solved by fetching the normal with the elevation for each vertex. However, this solution increases the size of the displacement maps by a factor of four (storing elevations and normals instead of only elevations). Instead, we use the geometry shader to compute the illumination for the refined surface. For each triangle, t, processed by the geometry shader, the normal is computed and t is shaded and illuminated accordingly. This solution reduces the processing on the vertex processors and simplifies the algorithm code.

5 Results

We have evaluated our implementation using various data sets with different complexities and have obtained impres-

Table 1 Preprocessing time and memory requirements

Model		VDT		DM-VDT		
Data set	Size	Time	Space	Time (m:s)		Space
		(m:s)	(MB)	CPU	GPU	(MB)
A. Dragon	7.2M	63:29	327	49:09	02:03	49.7
T. Statue	10.0M	81:23	461	68:10	03:10	63.6
Lucy	28.1M	217:45	1271	191:58	08:01	178.1
David	56.2M	420:09	2568	332:20	14:05	365.7

sive results. In this section, we report samples of these tests and their results, which were obtained using a PC with 2.13-GHz Pentium Dual-Core, 2 GB of memory, and an NVIDIA GeForce 8800 GTX graphics card with 768-MB video memory.

In our experiments, the control-mesh constitutes 1, 0.3, and 0.1% of the complexity of the input model for generic tiles of degrees $k = 9$, $k = 17$, and $k = 33$, respectively. While processing the test models, our simplification approach was successful at generating control-meshes that preserve elevation.

5.1 Performance

Table 1 reports the off-line construction time of the view-dependence trees. The first two columns include the name and size of the tested models. The *VDT* and *DM-VDT* columns show the preprocessing time and memory requirements of the View-Dependence Tree [42] and the suggested DM-VDT, respectively. The total DM-VDT preprocessing time is less than that of the VDT, since the preprocessing of the DM-VDT begins with a control-mesh. The sampling procedure consumes most of the pre-processing time and performing it on the GPU drastically reduces the preprocessing time, as can be seen in Table 1. For example, the preprocessing time for the Lucy model is reduced from 190 minutes to only 8 minutes. In addition, the space complexity of DM-VDT hierarchy is less than 1/7 of that of the VDT, which results from compactly representing the lower levels, using the corresponding displacement maps.

Real-time performance for data sets that fit entirely in video memory is summarized in Table 2. The first two columns report the degree of the generic tile and the triangle number. The last two columns show the rendering throughput, measured in millions of triangles per second. These results were computed by averaging the rendering throughput of 1000 frames. The peak performance of the used graphics hardware is 280M Δ/sec, rendering a cached VBO of indexed triangle strips. The peak performance drops to 267M Δ/sec when fetching a single elevation per vertex. The transmission cost of a front is inversely proportional to the tile's degree. In practice, our algorithm is GPU-bound for tiles of degree $k = 33$. The difference between the peak performance of 267M Δ/sec and the achieved performance of 264 Δ/sec is the cost of updating the front using the CPU and transmitting it to the GPU. Therefore, for tiles of greater degrees ($k \geq 33$), we also receive 264M Δ/sec. However, such high degree tiles reduce the flexibility of the level-of-detail selection and may exceed the maximal texture size. According to our results, the processing time of a single frame is distributed as follows: 4% for level-of-detail selection, 24% for front transmission, and 72% for GPU-based refinement.

Table 2 Performance with respect to tile degree

Tile degree	Triangles in a tile	VBO (triangles/sec)	Vertex fetch (triangles/sec)
9	64	154M	146M
17	256	216M	212M
33	1024	278M	264M

Table 3 Memory requirements and runtime performance for different displacement map formats

Model		32-bit		16-bit		JPEG	
Data set	Size	MB	fps	MB	fps	MB	fps
A. Dragon	7.2M	49.7	130	26.1	135	4.5	131
T. Statue	10.0M	63.6	89	33.1	92	7.9	88
Lucy	28.1M	178.1	64	93.5	67	23.5	65
David	56.2M	365.7	64	191.0	67	35.3	64

Displacement maps are generated from a relatively close surface, thus their displacement values are small and have slight variances. Therefore, the values of the displacement maps are encoded in 16-bit floating points without compromising model quality. To reduce the memory used to store a given model, we encode its displacement maps using the standard JPEG compression and decode them just before being cached into the video memory. Table 3 shows the memory usage to store the DM-VDT and the runtime performance using 32-bit and 16-bit displacement maps and the JPEG compression of the 16-bit displacement maps. Using 16-bit values with and without applying the JPEG compression reduces memory consumption by 90 and 50%, respec-

Table 4 Screen-space error and runtime performance for tile degree $k = 33$

Model		Sampling quality		Screen-space error		Performance
Data set	Size	ME	MSE	ME	MSE	(fps)
A. Dragon	7.2M	0.44	0.010	0.67	0.12	130
T. Statue	10.0M	0.38	0.033	1.52	0.45	89
Lucy	28.1M	1.23	0.024	1.70	1.08	64
David	56.2M	0.80	0.017	1.68	1.21	64

tively. The 16-bit representation slightly improves the rendering time, as a result of faster texture updates. However, compressing the 16-bit values decreases the rendering performance as a result of runtime decompression. Note that lossy compression of the displacement maps compromises the quality of the recovered model—adjacent patches may not stitch nicely and cracks may appear along the boundaries. In Sect. 5.2 we will discuss the influence of the compression on the quality of final images.

Our algorithm is designed to utilize the NVIDIA GeForce 8 series. However, vertex texturing is already supported from the GeForce 6 series. We have tested our implementation on a GeForce 7800 GTX graphics card with 256-MB video memory, and compared the results with those of the GeForce 8800 GTX. We have found that the triangle throughput of the GeForce 7800 GTX is only 20% of that of the GeForce 8800 GTX. This results directly from the differences in the architecture of the GPUs and the fact that our algorithm relies heavily on vertex texturing, and this performance depends on the number of vertex processors. The GeForce 7800 GTX (as well as previous graphics cards) uses different processors for vertex and fragment shaders. The GeForce 7800 GTX has only 8 vertex processors, which limits the performance of the vertex texturing. The GeForce 8800 GTX has a unified architecture, where the 128 processors can be used as vertex processors to accelerate vertex texturing. Since our algorithm utilizes the vertex processors optimally, despite the limitations of the GeForce 7800 GTX architecture, peak performance of the card is still achieved.

5.2 Image quality

The quality of the resulting images is measured using two metrics—*sampling-quality* and *screen-space error*. The *sampling-quality* for a triangle t is defined as the average distance d between its refined mesh and its corresponding patch. The *screen-space error* is the projection of d onto the screen.

Table 4 shows the error and rendering performance for various data sets. These results are achieved by using the interpolation-based sampling technique. The sampling-quality and screen-error columns show the maximum error (*ME*) and the mean square error (*MSE*). The last column reports the resulting frame rates (fps) for a screen-space error $\tau \leq 2$ pixels. As can be seen, our algorithm achieves high quality images at interactive rates. It also adapts nicely to the finer details of the input model. The VDT requires at least 93% of the triangles used by our algorithm (after triangle refinement) to reach the same image quality. The last two rows in Table 4 report the same performance (fps) for the Lucy and the David models, as a result of using the same screen-space error τ in the tests, thus rendering approximately the same number of triangles for these two models. Using a uniform subdivision scheme for triangles sometimes results in an overly large number of triangles. This occurs when the patch represented by a triangle is close to being planar. However, in order to prevent such cases, our algorithm uses runtime error control to adapt the selected level of detail to the required quality.

Table 5 Sampling quality of the suggested sampling techniques

Model		Intersection		Interpolation	
Data set	Size	ME	MSE	ME	MSE
A. Dragon	7.2M	0.44	0.010	0.45	0.009
T. Statue	10.0M	0.38	0.033	0.40	0.028
Lucy	28.1M	1.23	0.024	1.21	0.022
David	56.2M	0.80	0.017	0.76	0.013

The technique used for sampling the displacement maps implies an important factor in overall rendering performance and image quality. It is clear that different sampling techniques result in different sampling-quality, which means different displacement values. In Sect. 3.7 we show two different sampling techniques—the naive *intersection* technique, in which we simply take the intersection point between the ray shot from the sampled vertex and the original model, and our *interpolation* technique as described in Sect. 3.7. The resulting sampling-quality of these techniques is reported in Table 5. For each technique we present the maximal error (*ME*) and the mean square error (*MSE*) in two columns. The number of triangles required to reach a given screen-space error is inversely proportional to the sampling quality. Compared to the intersection technique, the interpolation-based technique provides a 5% improvement in sampling quality, which improves the rendering time by 7%.

Table 6 compares the introduced errors that resulted from using 16-bit and compressed displacement maps. It shows average, maximum, and variance of the difference between the sampled 32-bit values and the 16-bit representation and also between the 16-bit representation and the compressed displacement maps. Our experimental results show very small differences between the three representations.

Figure 1 shows four images that illustrate the flow of our approach. The images in (a) and (b) of Fig. 1 show the selected coarse front, and the images in (c) and (d) of Fig. 1 show the same front after being refined by the GPU. Figure 10 illustrates the flow and performance of our approach using the Lucy model. Figure 11 shows the front, the sampling error, and the final image of the Asian Dragon model. Figure 12 compares the accuracy of the two sampling techniques—intersection-guided and interpolation-based (see Sect. 3.7)—using the David model. Clearly, the interpolation-based approach provides better results than the intersection-guided approach. The model in Fig. 12 was rendered at approximately 64 fps with a 2-pixel screen error. Figure 13 shows a close-up view of the David model to emphasize potential screen-space error, using a large tolerance value ($\tau = 15$). Figure 14 presents the trade-off between the degree of the generic tile and the image quality, measured on the Lucy model.

Table 6 Degradation in model quality for various displacement map representations

	16-bit		Compressed			
	average	max	var.	average	max	var.
Asian	0.010	0.014	0.006	0.010	0.016	0.005
Lucy	0.024	0.031	0.013	0.024	0.031	0.013
David	0.017	0.023	0.010	0.017	0.024	0.010

5.3 External Video Memory

Table 7 reports the performance of our external video memory manager with respect to different tile degrees. The third column reports the number of displacement maps that fit into a 16-MB EVMM buffer (a single texture). The triangle binding column shows the number of displacement maps we managed to upload into the video memory within a second, without using the EVMM. It was not possible to achieve interactive rates for a front that includes more than 100 triangles. The EVMM column reports the number of displacement maps uploaded into the video memory within a second, using the EVMM. The EVMM improves the caching efficiency by a factor of 30. It also enables temporal coherence among consecutive frames—our results show that at least 94% of the displacement maps required for the next frame are already cached in the video buffer. By comparing the caching efficiency of different tile degrees, we conclude that the larger the generic tile, the better the utilization of the communication channel.

5.4 Comparison with other algorithms

Next, we compare our approach with some of the previously developed level-of-detail rendering approaches based on their published results.

The HLOD [15], the Tetrapuzzles [16], and the Quick-VDR [17] present level-of-detail rendering algorithms that

Table 7 The performance of EVMM

Tile degree	Triangles in a tile	Entries in a buffer	Triangle binding	EVMM
9	64	373K	3.5K	147K
17	256	94K	2.7K	78K
33	1024	29K	2.5K	30K

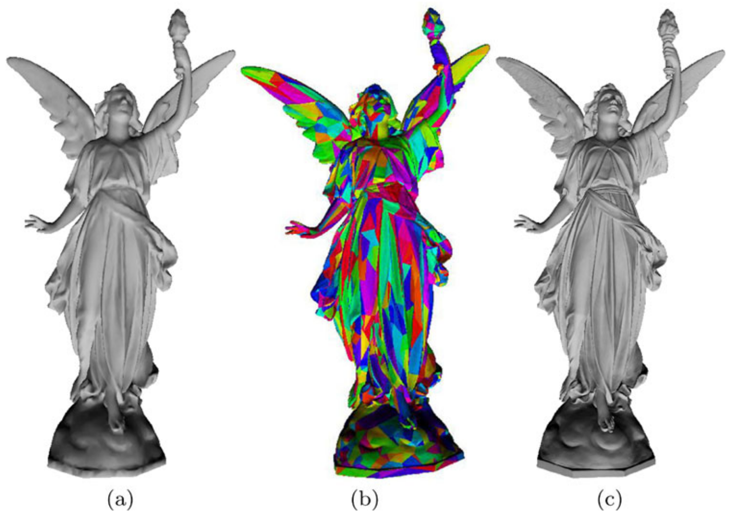


Fig. 10 The Lucy model: (**a**) A coarse front of 8K triangles; (**b**) The mesh partition imposed by the coarse front; (**c**) The refined mesh using generic tile of degree 33

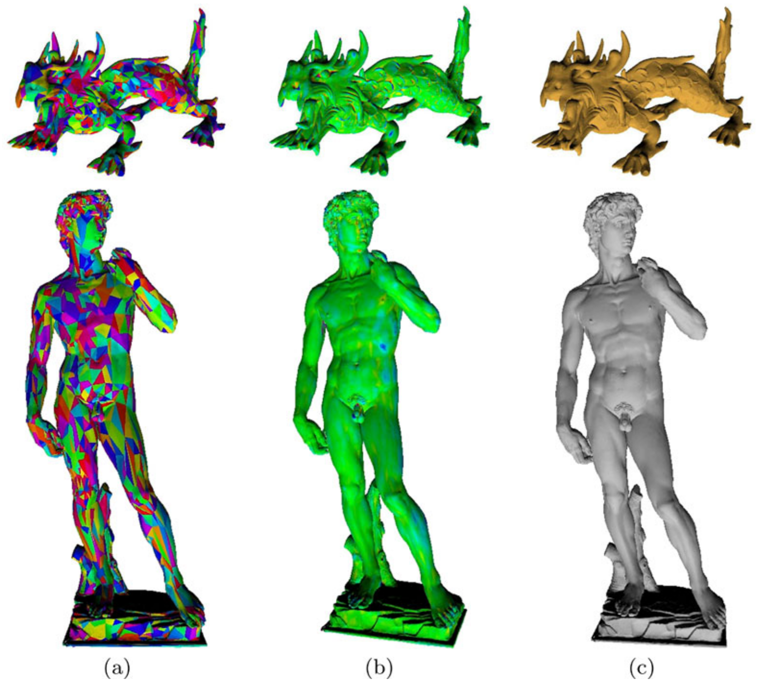


Fig. 11 *First row*: Asian Dragon Model (7.2M triangles); *Second row*: David model (56.2M triangles): (**a**) the partition into patches, (**b**) sampling error using interpolated base sampling, and (**c**) the final image after GPU refinement (*green* and *red colors* represent minimum and maximum error, respectively)

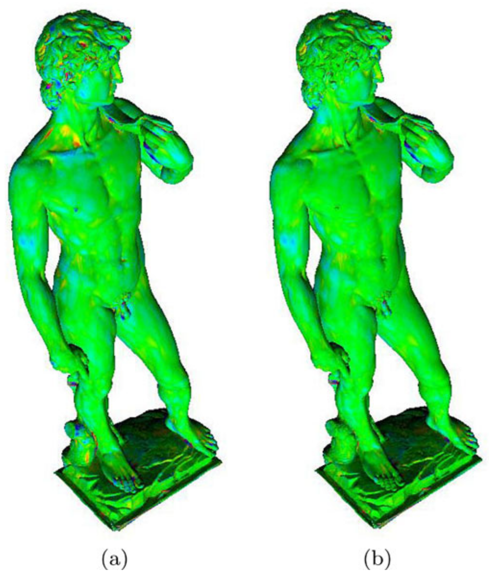


Fig. 12 The accuracy of (**a**) intersection-based, and (**b**) interpolated-based sampling, using the David model (*green* and *red colors* represent minimum and maximum error, respectively)

uniformly subdivide an input 3D model into disjoint patches, which are simplified, ordered in triangle-strip formats, and stored in a VBO. These algorithms explicitly store the geometry of the input model in a hierarchy and use several non-optimized triangle strips to represent each patch. In contrast, our approach represents patches as displacement maps, which saves approximately 85% in memory consumption. In addition, the generic tile is cached in an optimized triangle strip, which requires only a single transmission and rendering pass for each triangle. We compare our algorithm to these algorithms using models that fit entirely into the main memory. In such a scenario, these algorithms are GPU-bound and render approximately 280M Δ/sec, using our PC. As a result of using an error-guided subdivision scheme, our algorithm is able to utilize CPU-based patch culling during runtime, thus virtually increasing the performance beyond the maximum rendering capability of the GPU. It increases the performance by 35%, and renders approximately 356MΔ/sec. It also requires approximately 87% of the triangles used by the aforementioned approaches, to reach a 2-pixel error. In Fig. 15 we fixed the projection of the sampling-quality d to one pixel, as can be seen displacement patches requires less triangles than uniform subdivision sampling techniques for pixel error above one. However, it cannot achieve arbitrary small pixel error, as it samples the original model, while the uniform subdivision schemes are able to reach the original model at their finest resolution. Note that, visually, there are only minor differences between our approach and other works. These differences are almost invisible to the naked eye.

Ji et al. [35] suggested a GPU-based view-dependent rendering that selects and renders a level-of-detail within the GPU. The complexity of their algorithm prevented efficient implementation using the GPU. Instead, they emulated a GPU using a CPU-based implementation. The simplicity of our approach allows for a straightforward implementation that efficiently utilizes the processing power of the GPU.

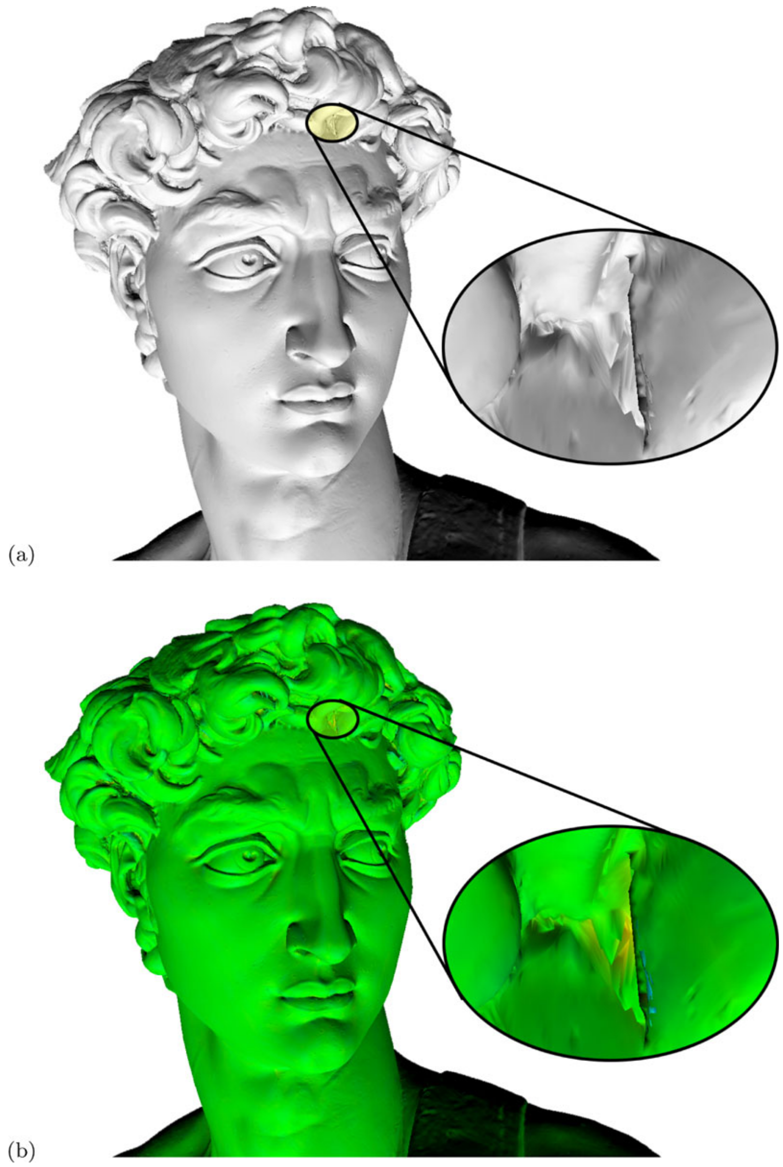


Fig. 13 A close-up of the David model: (**a**) a shaded view of the model; (**b**) the screen-space errors (*green* and *red colors* represent minimum and maximum error, respectively)

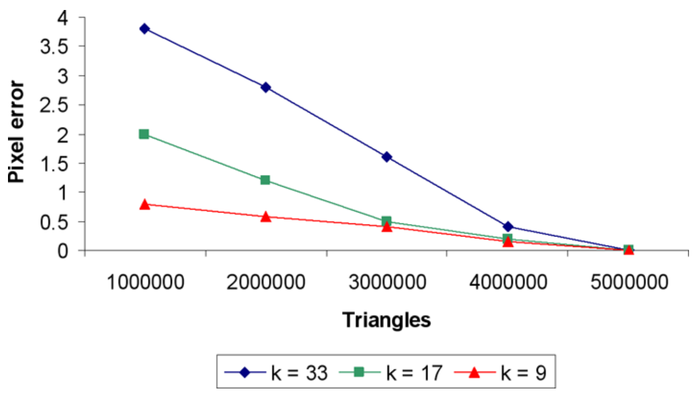


Fig. 14 The screen-space error for Lucy, according to the number of triangles for different tile degrees k. To obtain the same number of triangles for a smaller tile-degree one must render a higher level of the hierarchy, which is closer to the original model. However, as the number of desired triangles grow, the difference in error between tile degrees becomes smaller, making it possible to render large quantities of triangles mainly by means of subdivision

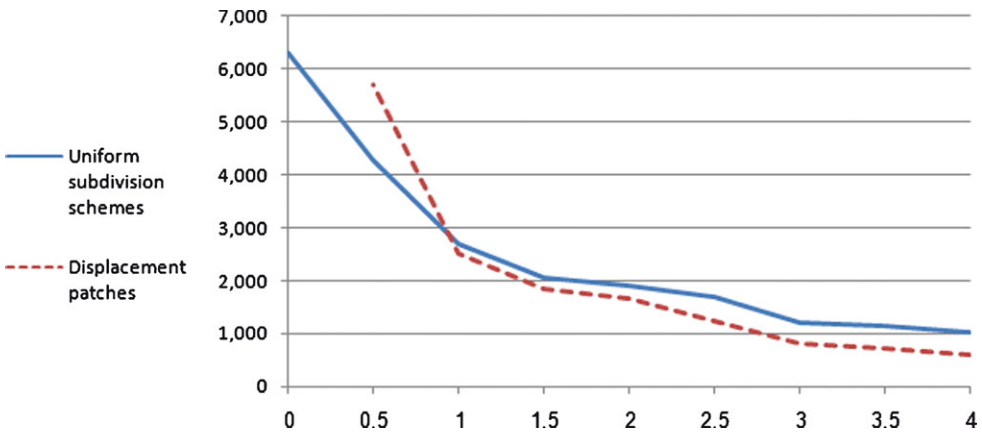


Fig. 15 Comparison between Displacement Patches and other algorithms: The Y-axis denotes the number of polygons that are required in order to achieve a minimal pixel error, which is denoted by the X-axis

6 Discussion

The generation of a control-mesh of an input model is performed using the quadratic error metric and preserving the elevation property. As a result, the different regions of the control-mesh have similar errors. In addition, large and small triangles of the control-mesh correspond to patches with low and high curvature, respectively. Furthermore, the curvature within patches is much smaller than the curvature along the boundaries. Considering the curvature as the angle between adjacent triangles, the definition of the quadratic error metric, and the order of edge-collapses, it is possible to assume that the ratio between the curvature c_i, of a patch p_i and its size s_i, is proportional to the geometric error ϵ_i, of its corresponding triangle t_i. Since ϵ_i is similar for all the patches, the ratio $\{c_i/s_i|\forall i\}$ is almost constant. For that reason, using the same sampling grid to approximate the patches of the input model leads to similar curvature error over the triangles of the sampled representation.

We would like to refine the control-mesh by sampling the input model such that the error over the different regions of the refined representation is similar and bounded. Let us assume that the geometric error is distributed uniformly over each patch. A uniform distribution of the geometric error could be achieved by using a generic tile with a degree $d \approx \sqrt{2\epsilon_m/\epsilon_t}$, where ϵ_m is the geometric error of the tile with the greatest error and ϵ_t is the required geometric error bound for the refined representation.

The generic tile is determined in an off-line preprocessing stage and may be independent of the real-time environment. Considering the diversity of the available graphics hardware, one may consider changing the degree of the generic tile of a given DM-VDT representation. One option is to regenerate the DM-VDT from the original model. However, the absence of the original model and the expensive preprocessing time may stipulate the need to resample the displacements at runtime. It is interesting to consider reducing the degree of the generic tile, which is performed by resampling the displacement maps at a lower rate. It is often necessary to take into account the neighborhood of the sample point in order to improve the sampling accuracy. Since fetching values from the texture memory is not cheap, it is possible to avoid fetching the neighborhood values in real-time by using the mipmap representation of the displacement maps. In such cases it is possible to let the hardware generate the mipmap for each displacement map in the video memory and fetch the value from the right level, based on the ratio between the degree of the original generic tile and the degree of the used one.

7 Conclusions and future work

We have presented a framework for a GPU-based view-dependent rendering of general 3D polygonal data sets. The control-mesh is an error-guided subdivision of the input model into disjoint patches. The geometry of these patches is encoded into displacement maps. Processing smaller multiresolution hierarchies within the CPU reduces its processing load and makes the CPU available for other general purpose computations. In addition, the transmission of a coarse level-of-detail along with displacement maps reduces the communication load between the CPU and the GPU. This is a result of utilizing the video memory for caching and transmitting only uncached primitives. In summary, our approach manages to seamlessly stitch adjacent patches without adding extra dependencies or sliver polygons.

We propose several possibilities for future enhancements to our algorithm and for further research. One might wish to eliminate the case of overly large triangulations of planar patches. It would be interesting to implement a GPU-based adaptive subdivision, instead of a uniform patch subdivision. Furthermore, for an additional boost in performance, the integration of existing occlusion culling techniques into our algorithm may be considered. In addition, using the latest DirectX 11 tessellation unit may allow us to migrate our entire algorithm to the GPU, thus enhancing its performance even further.

References

1. Lee, A., Moreton, H., Hoppe, H.: Displaced subdivision surfaces. In: Proceedings of SIGGRAPH '00, pp. 85–94 (2000)
2. Livny, Y., Bauman, G., El-Sana, J.: Displacement patches for GPU-oriented view-dependent rendering. In: Proceedings of GRAPP '08, pp. 181–190 (2008)
3. Hoppe, H.: View-dependent refinement of progressive meshes. In: Proceedings of SIGGRAPH '97, pp. 189–198 (1997)
4. Luebke, D., Erikson, C.: View-dependent simplification of arbitrary polygonal environments. In: Proceedings of SIGGRAPH '97, pp. 199–207 (1997)
5. De Floriani, L., Magillo, P., Puppo, E.: Efficient implementation of multi-triangulations. In: Proceedings of Visualization '98, pp. 43–50 (1998)
6. Hoppe, H.: New quadric metric for simplifying meshes with appearance attributes. In: Proceedings of Visualization '99, pp. 59–66 (1999)
7. Cignoni, P., Montani, C., Rocchini, C., Scopigno, R., Tarini, M.: Preserving attribute values on simplified meshes by re-sampling detail textures. Vis. Comput. **15**(10), 519–539 (1999)
8. Pajarola, R.: Fastmesh: Efficient view-dependent meshing. In: Proceedings of Pacific Graphics '01, pp. 22–30 (2001)
9. Kim, J., Lee, S.: Truly selective refinement of progressive meshes. In: Proceedings of Graphics Interface '01, pp. 101–110 (2001)
10. El-Sana, J., Sokolovsky, N., Silva, C.: Integrating occlusion culling with view-dependent rendering. In: Proceedings of Visualization '01, pp. 371–378 (2001)
11. Yoon, S., Salomon, B., Manocha, D.: Interactive view-dependent rendering with conservative occlusion culling in complex environments. In: Proceedings of Visualization '03 (2003)
12. El-Sana, J., Chiang, Y.: External memory view-dependent simplification. Comput. Graph. Forum **19**(3), 139–150 (2000)
13. DeCoro, C., Pajarola, R.: Xfastmesh: Fast view-dependent meshing from external memory. In: Proceedings of Visualization '02, pp. 363–370 (2002)
14. Cohen, J., Olano, M., Manocha, D.: Appearance-preserving simplification. In: Proceedings of SIGGRAPH '98, pp. 115–122 (1998)
15. Erikson, C., Manocha, D.: Hierarchical levels of detail for fast display of large static and dynamic environments. In: Proceedings of Symposium of Interactive 3D Graphics '01, pp. 111–120 (2001)
16. Cignoni, P., Ganovelli, F., Gobbetti, E., Marton, F., Ponchio, F., Scopigno, R.: Adaptive TetraPuzzles – efficient out-of-core construction and visualization of gigantic polygonal models. ACM Trans. Graph. **23**(3), 796–803 (2004)
17. Yoon, S.E., Salomon, B., Gayle, R., Manocha, D.: Quick-VDR: Interactive view-dependent rendering of massive models. In: Proceedings of Visualization '04, pp. 131–138 (2004)
18. Losasso, F., Hoppe, H., Schaefer, S., Warren, J.: Smooth geometry images. In: Proceedings of Eurographics/ACM SIGGRAPH Symposium in Geometry Processing, pp. 138–145 (2003)
19. Bolz, J., Schröder, P.: Evaluation of subdivision surfaces on programmable graphics hardware. Computational Geometry (2002)
20. Southern, R., Gain, J.: Creation and control of real-time continuous level of detail on programmable graphics hardware. Comput. Graph. Forum **22**(1), 35–48 (2003)
21. Schein, S., Karpen, E., Elber, G.: Real-time geometric deformation displacement maps using programmable hardware. Vis. Comput. **21**(8), 791–800 (2005)
22. Policarpo, F., Oliveira, M., Comba, J.: Real-time relief mapping on arbitrary polygonal surfaces. In: Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games '05, pp. 155–162 (2005)
23. Kautz, J., Seidel, H.: Hardware accelerated displacement mapping for image based rendering. In: Proceedings of Graphics Interface '01, pp. 61–70 (2001)
24. Baboud, L., Décoret, X.: Rendering geometry with relief textures. In: Proceedings of Graphics Interface '06, pp. 195–201 (2006)
25. Cignoni, P., Ganovelli, F., Gobbetti, E., Marton, F., Ponchio, F., Scopigno, R.: P-BDAM – planet-sized batched dynamic adaptive meshes. In: Proceedings of Visualization '03, pp. 147–155 (2003)
26. Wagner, D.: Terrain geomorphing in the vertex shader. ShaderX2: Shader Programming Tips & Tricks with DirectX 9 (2004)
27. Dachsbacher, C., Stamminger, M.: Rendering procedural terrain by geometry image warping. In: Proceedings of Eurographics/ACM SIGGRAPH Symposium in Geometry Processing, pp. 138–145 (2004)
28. Hwa, L., Duchaineau, M., Joy, K.: Real-time optimal adaptation for planetary geometry and texture: 4-8 tile hierarchies. IEEE Trans. Vis. Comput. Graph. **11**(4), 355–368 (2005)
29. Asirvatham, A., Hoppe, H.: Terrain rendering using GPU-based geometry clipmaps. In: GPU Gems 2, pp. 27–45. Addison-Wesley, Reading (2005)
30. Schneider, J., Westermann, R.: GPU-friendly high-quality terrain rendering. J. WSCG **14**(1–3), 49–56 (2006)
31. Livny, Y., Sokolovsky, N., Grinshpoun, T., El-Sana, J.: A GPU persistent grid mapping for terrain rendering. Vis. Comput. **24**(2), 139–153 (2008)
32. Doggett, M., Hirche, J.: Adaptive view dependent tessellation of displacement maps. In: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware '00, pp. 59–66 (2000)
33. Guskov, I., Vidimče, K., Sweldens, W., Schröder, P.: Normal meshes. In: Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques '00, pp. 95–102 (2000)
34. Hirche, J., Ehlert, A., Guthe, S., Doggett, M.: Hardware accelerated per-pixel displacement mapping. In: Proceedings of Graphics Interface '04, pp. 153–158 (2004)
35. Ji, J., Wu, E., Li, S., Liu, X.: Dynamic LOD on GPU. In: Proceedings of Computer Graphics International '05, pp. 108–114 (2005)
36. Donnelly, W.: GPU Gems 2, chap. Per-Pixel Displacement Mapping with Distance Functions, pp. 123–136. Addison-Wesley, Reading (2005)
37. Nielson, G.: The side-vertex method for interpolation in triangles. J. Approx. Theory **25**, 318–336 (1979)
38. Mann, S.: An improved parametric side-vertex triangle mesh interpolant. In: Graphics Interface, pp. 35–42 (1998)
39. Mao, Z., Ma, L., Tan, W.: A modified Nielson's side-vertex triangular mesh interpolation scheme. Comput. Sci. Appl. **3480**, 776–785 (2005)
40. Garland, M., Heckbert, P.: Surface simplification using quadric error metrics. In: Proceedings of SIGGRAPH '97, pp. 209–216 (1997)
41. Shannon, C.E.: A mathematical theory of communication. Bell System Technical Journal **27**, 379–423, 623–656 (1948)
42. El-Sana, J., Varshney, A.: Generalized view-dependent simplification. Comput. Graph. Forum **18**(3), C83–C94 (1999)

Yotam Livny is a Ph.D. student of Computer Science at the Ben-Gurion University of the Negev, Israel. His research interest includes interactive rendering of large 3D graphic models using multiresolution hierarchies and programmable graphics hardware. Yotam received a B.Sc. in Mathematics and Computer Science from Ben-Gurion University of the Negev, Israel in 2003. He is currently a Ph.D. student in Computer Science under the supervision of Dr. Jihad El-Sana.

Gilad Bauman is a Ph.D. student at the Department of Computer Science, Ben-Gurion University of the Negev, Israel. He is studying under the supervision of Dr. Jihad El-Sana. His main research field is Computer Animation. Bauman received his B.Sc. and M.Sc. in Computer Science from Ben-Gurion University of the Negev in 2005 and 2008. Bauman had published 2 papers during his M.Sc. studies in level-of-detail rendering and GPU computing.

Jihad El-Sana is an Assistant Professor of Computer Science at Ben-Gurion University of the Negev, Israel. He is the Chair of AHD, the organization that founded and implemented the School. El-Sana received a B.Sc. and M.Sc. in Computer Science from Ben-Gurion University of the Negev, Israel in 1991 and 1993. He received a Ph.D. in Computer Science from the Stony Brook University, NY in 1999 supported by a Fulbright. El-Sana has published over 50 papers in leading conferences and scientific journals. He is also member of the IEEE and Eurgo-Graphics societies.