Simplified and Tessellated Mesh for Realtime High Quality Rendering

Yazhen Yuan, Rui Wang[1], Jin Huang, Yanming Jia, Hujun Bao

State Key Lab of CAD&CG, Zhejiang University, China

Abstract

Many applications require manipulation and visualization of complex and highly detailed models at realtime. In this paper, we present a new mesh process and rendering method for realtime high quality rendering. The basic idea is to send a simplified mesh to hardware pipeline, while use the online tessellation on the GPU to facilitate the rendering of complex geometric details. We formulate it into an inverse tessellation problem that first computes the simplified mesh, and then optimizes the tessellated mesh with geometric details to approximate the original mesh. To solve this problem, we propose a two-stage algorithm. In the first stage, we employ an iterative surface simplification technique, where we take the requirement of hardware tessellation into consideration to obtain an optimal simplified mesh. In the second stage, to better utilize the hardware tessellation, we propose a moving vertex strategy to approximate the tessellated mesh to the original mesh. Results show that our method achieves 2-4 times faster at rendering but still retains high quality geometrical details.

Keywords: hardware tessellation, mesh simplification, mesh rendering and visualization

1. Introduction

Realtime rendering of complex, highly detailed model is of great interest in variety of performance demanding applications, such as game, visualization, virtual reality, etc. However, due to the bottleneck of I/O, it becomes a popular strategy to highly simplify the complex model so as to achieve desired frame rate. But, with the simplification, the rendering quality significantly degenerates in general.

As a result of recent advances in graphics hardware, large number of geometry primitives can now be efficiently and flexibly generated online with the highly parallel GPU tessellation units. Technically, to fully utilize the power of the GPU tessellation units, it requires a two-layer representation of model. A coarse model to be sent from CPU to GPU, and a fine model that are tessellated at rendering. However, given a complex model, it is challenge to decompose the input model into such a two-layer representation. First, the coarse model provides a base to be rendered and tessellated at runtime. The hardware tessellation performance and quality depend on this coarse representation. Thus, how to get the best coarse model for hardware tessellation is a problem. Second, the hardware tessellation requires tessellation parameters and vertex data to recreate the details of original model. How to obtain optimal parameters and vertex data to approximate the original mesh is another problem.

Inverse subdivision techniques [1, 2, 3] can be regarded as potential solutions for these two problems. They decomposed the connectivity of original mesh into a coarse representation for subdivision. Guskov et al. [4] proposed normal meshes to compress the storage of meshes by constructing a multiresolution mesh with normal offsets. Cook [5] introduced the ideal of displacing a surface by a function. Lee et al. [6] introduced the displacement maps as an inverse subdivision process so that the original mesh can be approximated by displaced subdivision surface. However, even with the recent progress to carry out realtime subdivision using hardware tessellations [7], these inverse approximation approaches [4, 6] still suffer from two main limitations. First, in computing the coarse representation, these methods do not consider the hardware tessellation stage, therefore the coarse representation is not optimal for hardware tessellation. Second, no matter the normal offset or the displacement map, these methods only move vertices along normals, which may fail at capturing some geometric features of original mesh, especially in case of using a small number of tessellated surfaces.

To address these limitations, in this paper, we propose a new solution to find optimal decomposition for the

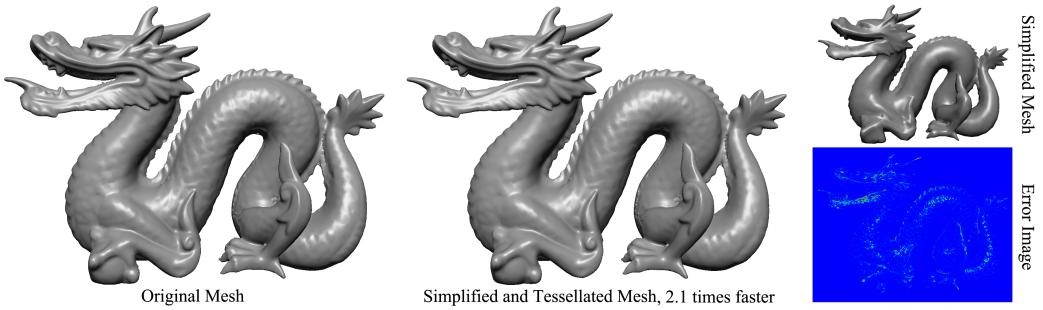


Figure 1: (Left): the original mesh with 100K triangles. (Middle): the simplified and tessellated mesh generated by our method. It is 2.1 times faster to render it than directly render the original mesh. Total 222K triangles are tessellated at runtime in tessellation shaders. (Right Top): the simplified mesh with 8K triangles before the tessellation. (Right Bottom): the error image between the original mesh and ours.

hardware tessellation. In Fig. 1, we show the computed *simplified and tessellated mesh* of the Dragon model. Our algorithm takes two stages. In the first stage, we employ an iterative surface simplification technique, and take the requirement of hardware tessellation into consideration to optimally generate the simplified mesh. In the second stage, the tessellation stage, we compute tessellation factors and vertex offset data to tessellate the simplified mesh into the tessellated mesh. A new approximation strategy, we name it as moving vertex, is proposed to move positions of new tessellated vertices to better preserve geometric features of the original mesh. In the entire approximation, there are several parameters impacting on the generation of our simplified and tessellated mesh, which are non-intuitive for users to tweak. Therefore, we present an optimization scheme that takes the time budget as input and automatically optimizes to an optimal approximation with lowest error. Results show that our method achieves 2-4 times faster at rendering but still retains high quality geometry details. The main contributions of this work include:

- A new mesh simplification and tessellation approach to utilize hardware tessellation for realtime high quality mesh rendering and visualization.
- A new mesh simplification scheme that considers factors of fast hardware tessellations.
- A new vertex optimization that moves tessellated vertices to better approximate geometric features of original mesh.
- An automatic optimization scheme that takes the time budget as input, and produces optimal simplified and tessellated mesh with lowest approximation error.

2. Related Work

Mesh Simplification. Mesh simplification has been addressed of great interest in a variety of applications. It takes a polygonal mesh as input and generates an approximated one with less number of vertices and faces. In past decades, various methods have been proposed. A popular strategy is to remove primitives greedily leading to the lowest error. Error metric of simplification is the key of this strategy. Different metrics have been proposed to measure the quality of simplifications [8]. As one of the typical methods, QEM [9] uses local geometric quadric error as the simplification metric. To preserve more information on the input mesh, Hoppe [10] extended it by introducing additional attributes such as color and normals. The metric is even extended to consider tangents, texture coordinate, and animation information [11]. Appearance-preserving simplification [12] applies texture deviation metric to both the texture and normal maps during simplification process. There are also other strategies being proposed to simplify the mesh in different ways. Vertex clustering [13] overlays a 3D grid on the model and collapses all vertices within each cell of the grid to the single most important vertex within the cell. Simplification envelopes [14] use two offset surfaces to guide the simplification process. Progressive and parallel simplifications [15, 16] accelerate the simplification using GPUs. There are several surveys to compare those different technologies in theory [17, 18] and from the view of practise [19]. However, the designing of these metrics ignores the specific requirement of GPU tessellation units.

Inverse Subdivision. Taubin [1] introduced an inverse subdivision algorithm to detect and reconstruct subdivision connectivity. However, his interest was only at extracting the connectivity of a subdivided mesh but not at the geometric approximation of the mesh. Sandrine and Marc [2] proposed a reverse Catmull-Clark subdivision algorithm to generate a coarse mesh from a

subdivided mesh. Sadeghi et al. [3] presented a smooth reverse subdivision algorithm. Guskov et al. [4] proposed normal meshes to compress the storage of meshes by constructing a multiresolution mesh with normal offsets. But all of these methods focused mainly on data compression. Compared with them, our method targets on approximating a general complex, highly detailed mesh, which considers the utilization of hardware both at mesh simplification and mesh rendering. Therefore, their methods can not be directly applied in our application.

Displacement Maps. Cook [5] introduced the ideal of displacing a surface by a scalar function. Cohen et al. [12] and Cignoni et al. [20] proposed normal map to reconstruct mesh's normal during rendering. Lee et al. [6] combined the idea of surface subdivision and displacement map to approximate detail surfaces. But none of these approaches considered the impact of hardware tessellation during generating the coarse representation. In addition, displacement maps and normal maps both try to approximate dense mesh in texture space, where the approximation quality heavily depends on texture resolutions. Compared with them, our method moves tessellated vertices to geometric feature vertices instead of only moving vertices along normals. In this way, our method is able to better approximate the original meshes even using less tessellated vertices.

Hardware Tessellation. While the development of GPU hardware, hardware tessellation has been introduced along with the DirectX 11 [21] and OpenGL 4.1 [22]. The basic idea of hardware tessellation is to generate highly detailed geometry at runtime from a coarser representation. Polygons are directly processed by parallel streaming processors without involving further global memory access, which minimizes memory I/O, and achieves high-performance. Many methods have been proposed to utilize the hardware tessellation for realtime rendering [23]. But, as we know, the goal of our work that an inverse mesh simplification process for hardware tessellation is new.

3. Overview

The goal of our paper is to take a triangular model as input, and generate a simplified and tessellated model, the two-layer representation, to approximate the original model as output. To practically solve the problem, we divide the entire mesh process into two stages, the simplification stage and the tessellation stage. At the simplification stage, we apply mesh simplification on the original mesh to obtain a simplified mesh. Then, at the second stage, we tessellate the simplified mesh into a denser mesh, where new tesselated triangles are optimized to compensate the quality loss in the simplification stage so as to better approximate the original mesh.

While taking the tessellation stage into consideration, there are several parameters impacting on the approximation error and rendering time. It becomes non-intuitive for users to deduce the error or rendering time from the number of simplified mesh or the level of tessellations. To address this problem, we then develop an optimization algorithm to find an optimal simplified and tessellated mesh by giving a certain rendering time. The optimal mesh is searched in the time-error space with different parameters so as to best approximate the original mesh.

In Section 4, we describe the mesh simplification stage that generates the simplified mesh for further tessellation under a certain number of triangles. In Section 5, we introduce the tessellation stage that approximates the original mesh by newly tessellated triangles under certain number of tessellations. In Section 6, we present the optimization algorithm that explores a number of simplified meshes and tessellations to find the best approximation under a certain rendering time budget.

4. Mesh Simplification

Many methods and error metrics have been developed to solve the mesh simplification problem. But, in our method, due to a different goal, we have our own evaluations on simplified mesh that the simplified mesh should be an optimal mesh for our next tessellation stage. We consider three criteria at evaluating a simplified mesh.

The multiresolution ability. Since the hardware tessellation process can be regarded as the generation of triangles from coarse to fine – with appropriate levels of details, the simplification method should have the ability to generate multiresolution models.

Better surface approximation. In our tessellation stage, the approximation to the original mesh is computed from a simplified base mesh. Thus, if the simplified mesh approximates original mesh well, it will benefit the approximation in the tessellation stage.

The uniform distribution of feature vertices. This criteria is new for mesh simplification. It is proposed under the consideration of hardware tessellations. Using hardware tessellation, the number of features on a triangle is sometimes more important than the degree of feature importance. This is because new vertices can be generated and positioned at any positions. We found that the variation of numbers of tessellations on different triangles have impacts on the rendering time. If all

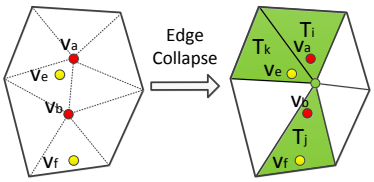


Figure 2: Edge $v_a v_b$ are collapsed to a new vertex. v_a, v_b are mapped to triangle T_i, T_j separately. Other feature vertices, v_e and v_f relevant to this operation are re-projected to nearest triangles

triangles have the same tessellation factors, the tessellation stage tends to run faster (please refer to Section 7.3 for more discussions). In this sense, we hope every simplified triangle has the same number of features, so that we can uniformly subdivide triangles into same tessellations to approximate these features.

Based on these criteria, we combine the quadric error [9] and the features per triangle as the simplification error metrics, where the quadric error satisfies first two criteria, and the features per triangle is used to control the number of feature vertices on simplified mesh:

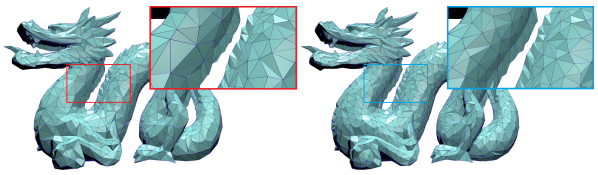
$$Q(v) = v^\top E v + 2 f^\top v + g \quad (1)$$

$$D(T) = \sum (c(v_i) > c_{min}) \quad (2)$$

where $Q(v)$ is the quadric error, E is a symmetric matrix, f is a vector and g is a scalar (more details can be found in [9]), $D(T)$ is the the number of feature vertices on a simplified triangle T, v_i is the feature vertex that has been simplified and mapped to triangle T. The feature vertex defined in our method is the vertex, whose gaussian curvature, $c(v_i)$, is larger than a predefined threshold, c_{min}(0.005 by default).

Using such error metrics, we adapt the established framework [9] in our method. It iteratively applies edge collapse operations on geometry primitives, and updates the cost of a small portion of the mesh being simplified. The features per triangle is used when the simplification operation is performed. If one geometry primitive has too many features, any further simplification operations will be canceled.

More precisely, the simplification process takes following steps. First, we assign the quadric error as the cost for each valid operation. Then, we choose the operation with lowest cost and apply it on the mesh. After that, we re-evaluate the features per triangle of all triangles relating to this operation. We illustrate an example of such a process in Fig. 2. After an edge collapse operation, these two collapsed vertices, v_a and v_b, are projected along their normals, and assigned to the new triangle with least distance. Additionally, feature vertices on all influenced faces adjacent to the two vertices,



(a) With features per triangle (b) Without features per triangle

Figure 3: Comparison of simplification results with or without using features per triangle. With the features per triangle, simplified triangles tend to preserve regions with more vertices. As can be seen, in (a) more triangles are distributed on the back regions than those in (b), because these regions have plenty of vertices with high curvatures. On the contrast, fewer triangles are distributed on the neck, since the region is relatively flat.

v_e and v_f, are re-projected to the newly generated triangles. Once one of the features per triangle exceeds a pre-defined threshold, we cancel this operation. In this way, the features per triangle of simplified triangle can be controlled. If the operation goes through the test, we then update the costs of operations with modified triangles of mesh, and reorder operations according to newly updated costs. These steps are iteratively executed until no more operations can be applied (e.g. all triangles are with maximum features per triangle, or a desired triangle number is achieved). After the simplification step, we compute a globally smooth parameterization to map vertices of original mesh to the simplified mesh [24]. Such a parameterization is used to compute the tessellated mesh by mapping features on the original mesh to the simplified mesh. In Fig. 3, we illustrate two simplified results with or without the features per triangle criteria in mesh simplification. It can be seen that with the features per triangle criteria, more triangles will be used to preserve vertices with high curvatures.

5. Approximation by Tessellated Mesh

After we obtain the simplified mesh, we then simulate the tessellation in tessellation shader to compute the tessellated mesh. Newly tessellated triangles are used to approximate the original mesh. In tessellation shader, the number of tessellations is determined by two tessellation factors, the outer tessellation factor, l^{out}, that determines the subdivision on triangle edge, and the inner tessellation factor, l^{in}, that determines the numbers of vertices and sub-triangles inside the triangle [21].

We use equal spacing model to make sure all tessellation factors are integers. The modern hardware supports different tessellations per triangle. But, to void cracks and T-joins, we need to make sure the outer tessellation factors on two adjacent triangles are the same. Therefore, the inner tessellation factor is set per triangle, and

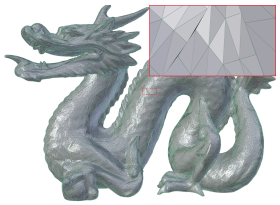


Figure 4: Applying our moving vertex strategy, T-joints happens when outer tessellation factors on adjacent triangles are different.

the outer tessellation factor is set per edge. We show a failure example in Fig. 4, when the outer tessellation factors on two adjacent triangles are different.

In this section, we first introduce the error metric used to evaluate the quality of approximations. Then, we describe our new approximation strategy, the moving vertex, to compute the vertex offset data for tessellated vertices. An illustrative example is shown in Fig. 5. We approximate geometrical features on the original mesh by moving the new tessellated vertices to the positions of feature vertices. For the computation efficiency, we store the tessellated vertices' new position and normals and use them at runtime. Due to the utilization of hardware tessellation, we can reduce the cost both in time and memory.

5.1. Error Assessment

To measure the error produced by the approximation, we both consider the geometric difference and normal variations as:

$$e(S', S) = d(S', S) + w \cdot n(S', S) \tag{3}$$

where S is the original surface, S' is the approximation surface and w is the weight to combine these two terms. $d(S', S)$ is the Hausdorff distance between two surfaces [25], which is defined as

$$\begin{aligned} d(S', S) &= \int\int_{x' \in S'} \| d(x', S) \|_\infty, \\ d(x', S) &= \inf_{x \in S} \| x' - x \|_\infty \end{aligned} \tag{4}$$

where x and x' are points on surfaces S and S' respectively. We only use the non-symmetric distance since we only want our simplified and tessellated mesh best approximates the original mesh. $n(S, S')$ is the normal distance between two meshes. It is defined as:

$$n(S', S) = \int\int_{x' \in S'} \| dot(n(\mathcal{P}(x')), n(x')) \|_2 \, dx' \tag{5}$$

where $\mathcal{P}(x')$ is the mapping from the point x' to the surface S. The reason to consider the normal variations is mainly because in many rendering cases, perceptual

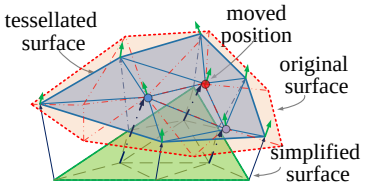


Figure 5: Illustration of Moving vertex strategy.

errors brought by normals are more obvious than geometric difference.

To compute the error between the tessellated mesh M_t and the original mesh M_o, we first generate a set of samples $\{x\}_K$ on each triangle of the tessellated mesh M_t, where $K = 1 \sim 32$ according to the size of tessellated triangle. Then, we use Eq.(3) to compute the error between two meshes as

$$e(M_t, M_o) = \sum_i e(T_i, M_o) \tag{6}$$

where T_i are triangles on the tessellated mesh M_t.

5.2. Approximation by Moving Vertex Strategy

Given inner and outer tessellation factors, we employ a hardware tessellation simulator to generate subdivided triangles and vertices. To achieve a better approximation, we hope most features on the original mesh should be preserved. Therefore, we propose a new strategy that moves tessellated vertices to feature vertices so as to better reconstruct the original mesh.

To carry out this strategy, we differently process vertices on edges or vertices inside of simplified triangles. For tessellated vertices on edges, we need guarantee the boundaries between triangles are kept. Thus, we first interpolate the normal from the base vertices, and then project the vertex along normal direction or the reverse normal direction. If there exists a projection on the original mesh, we store the position of hit point. However, in some rare cases, there may not exist a valid projection, we will store the vertex for later interpolation.

For those tessellated vertices inside of the simplified triangles, the key of this strategy is to find an appropriate match between tessellated vertices and feature vertices. To achieve this, we first use the local parametrization to map the feature vertices onto the local barycentric coordinate of simplified triangle. Then, we formulate the matching as a minimum weighted bipartite matching problem [26], where the distance in the barycentric coordinate between tessellated vertices and feature vertices is set as the weight. After solving this problem, we have the matching to move tessellated vertices.

To avoid the flips of triangles, we check the normals of all subdivided triangles after moving vertices.

For these triangles with flipped normals, we cancel the movements of vertices one by one. The feature vertex, which has smaller curvature, will be unanchored first. Once there is no flipped triangle, we anchor all moved matching vertices and project unmatched vertices to original mesh. After that, another triangle flipping check is taken to guarantee there are no flips of triangles. For these vertices produce flipped triangles, we cancel the projections and interpolate their positions and normals from adjacent vertices. In Algorithm 1, we illustrate main steps of this strategy.

Algorithm 1 Moving Vertex

```
1:  procedure MovingVertex(M_s, M_o)
2:      for T_i in M_s
3:          {v_j} = TessellateTriangle(T_i)
4:          ProjectVertexOnEdge({v_j})
5:          // Get the feature vertices on this face
6:          {u_k} = ExtractFeatureVertex(M_o, T_i)
7:          // Vertices match and move
8:          {(v_jp, u_kp)}= MinBiGraphMatch({v_j},{u_k})
9:          MoveAndValidate({(v_jp, u_kp)})
10:         // Project remaining vertices
11:         ProjectAndValidate({v_j})
12:         v_list ← VertexCauseFlip({v_j})
13:     end for
14:     Interpolate(v_list, M_s)
15: end procedure
```

6. Optimization for Optimal Simplified and Tessellated Mesh

Given one original mesh, we can generate numerous simplified and tessellated meshes with different simplified triangles and tessellation factors. However, given one rendering time budget, there exists one optimal simplified and tessellated mesh that produces the least error. In this section, we introduce the optimization algorithm to automatically find such an optimal mesh.

In the simplification stage, we generate a simplified mesh under certain number of triangles, n_s, and in the tessellation stage, we recreate details by presetting tessellation factors, $l_t = (l^{in}, l^{out})$. Thus, the tessellated mesh can be represented as a function of these two kinds of parameters as $M_t(n_s, l_t)$. Obviously, these two kinds of parameters are non-intuitive for users. Therefore, we introduce an optimization algorithm to automatically explore these two kinds of parameters after user gives a budget of time, and then optimize a simplified and tessellated mesh with minimum approximation error. We use t_{max} to denote the given time. The optimization problem can be formulated as

$$\begin{aligned} \min \quad & e(M_t(n_s, l_t), M_o) \\ \text{s.t.} \quad & t(M_t(n_s, l_t)) < t_{max} \end{aligned} \tag{7}$$

where $t(M_t(n_s, l_t))$ is the time to render tessellated mesh $M_t(n_s, l_t)$.

Since there is no analytical formation of $M_t(n_s, l_t)$, we take the numerical solution to optimize it. Given a t_{max}, we first compute the estimated upper bound of triangles of simplified mesh by scaling the triangle number of original mesh as $N_{max} = t_{max}N_o/t_o$, where N_o and t_o are the triangle number and rendering time of the original mesh respectively. Though the time to render mesh is not completely linear to the number of triangles, in our practice, N_{max} is a good bound to generate a simplified mesh without any tessellation factors. We then always use 1K triangles as the minimum bound of simplified triangles. To find the optimal number of simplified mesh, we use binary search between N_{max} and N_{min}. To find the best tessellation factors for a given triangle number of simplified mesh, n_s, we first enumerate the tessellation factors by setting the same outer and inner tessellation factors for all triangles, so that we could estimate the maximum bound of tessellation factors, when $t(M_t(n_s, L_{max})) > t_{max}$. We set the minimum bound of tessellation factors as 0, i.e. no tessellations. We also take a binary search to find the tessellation factor l_t to produce the minimum approximation error. Note that l_t is set as a global estimation for all triangles. For each triangle, we enumerate the outer and inner tessellation factors from 0 to l_t on each edge and in each triangle respectively to find the local optimal approximation to the original mesh. In summary, our optimization globally employs binary search to explore two parameters n_s and l_t, but locally takes linear search to find best tessellation factors per edge and triangle.

7. Results

We have implemented our algorithm on a PC with an Intel Core™ i7 4770 CPU, 16GB RAM. We test our method on several graphic cards with different hardware tessellation capabilities, NVIDIA GeForce GTX 680, GTX 760 and GTX 780. We use the hardware counters, *gpu_time*, provided by the NVIDIA PerfSDK to measure the rendering time.

7.1. Projecting Vertex Strategy

To compare our method with previous inverse subdivision methods [4, 6], we adapted the idea of computing

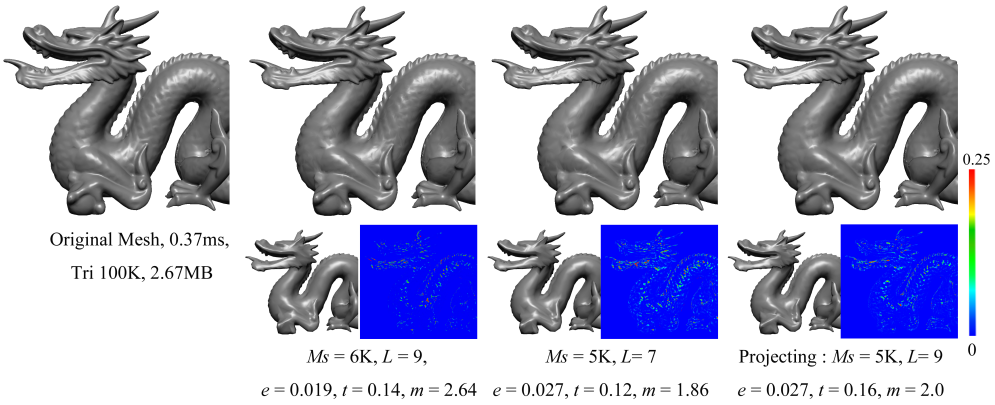


Figure 6: Dragon model.

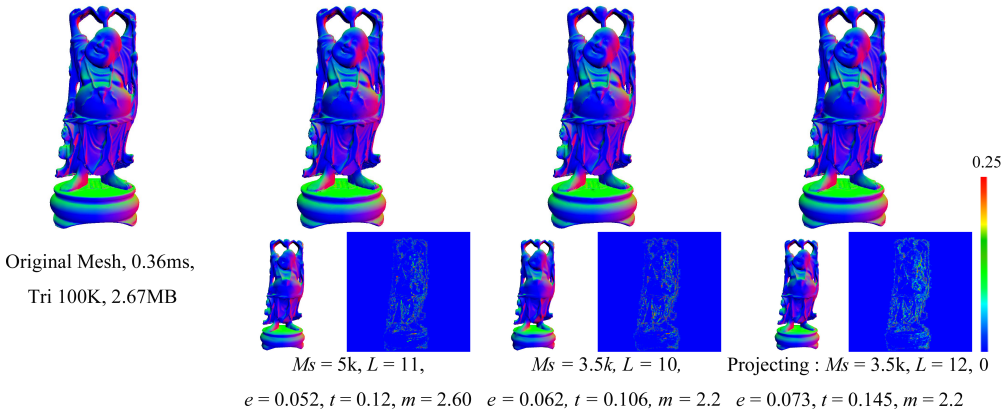


Figure 7: Buddha model.

displaced vertices [6] from the Loop subdivision to our hardware tessellation. We name it the projecting vertex strategy. While applying this strategy, we compute and store offset values along the interpolated normals of tessellated vertices, i.e. the height values of tessellated vertices. Each offset value is computed by projecting the vertex along the normal or the reverse normal direction to the original mesh. If there exists a projection on the original mesh, and it is within the maximum distance threshold based on mesh size, we regard it as a valid projection, and store the height and normal of the projection on the original mesh in height array and normal array respectively. This strategy is used in our example models for comparisons.

7.2. Example Models

Three models are used to testify our method. Results of Dragon, Buddha and Lucy models are shown in Fig. 1 6 7 8. For each model, The first image shows the original mesh. Following two results are optimized by our method under two different time budgets. The final result is generated by the projecting vertex strategy for comparison, where the rendering time, memory cost or the error was set to be the same as that of the second result. For every result, we report the simplified mesh (bottom left), error image (bottom right), and the triangle number of simplified mesh M_s , the maximum tessellation factor L, the error e, the time t and the memory consumption m. To be distinct with the moving vertex strategy, we use "Projecting" to denote the results of using the projecting vertex strategy.

Dragon. The Dragon Model is a model with 100K triangles and tested on a graphic card GeForce GTX 760, Fig. 1. The resultant simplified and tessellated mesh has a maximum 10 tessellation factors, and the average tessellation is 4.13. The rendering time of original mesh is 0.37 ms, and the time to render simplified and tessellated mesh is only 0.18 ms. More results of dragon model can be found in Fig. 6, in which we set two time budgets: 0.14 ms and 0.12 ms. The Dragon model has some sharp features and bumps on its head and body. If we set the target triangle number very small, the mesh

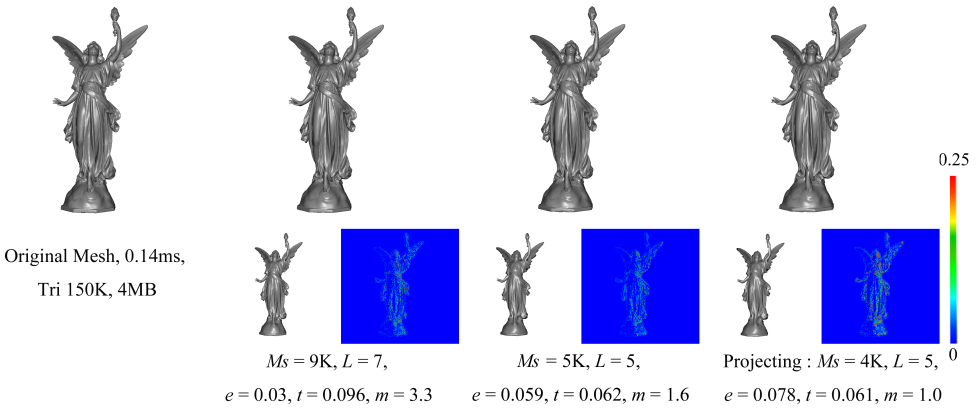


Figure 8: Lucy model.

simplification method tends to remove small details to maintain mesh's shape. For example, the head and body of the dragon model are flat after mesh simplification, which can be seen in the simplification mesh image below every result. But our method has the advantage to tessellate new vertices to compensate the loss of geometric details, and does not incur much costs. The first result, which was bound to 0.14 ms, shows that the tessellated mesh keeps all main features and is very similar to the original mesh. The resultant mesh is tessellated from 6K to 130K triangles, but is still 2.6 times faster than that of directly rendering the original mesh. And because of the use of coarse mesh save large index buffer of triangle, the memory cost is still less than original mesh. If we set a lower budget time, we get a simplified mesh with smaller tessellation factor, $L = 7$. Under the new budget time, our method still captures main details, while the rendering time has about 3 times speedup. Compared with the projecting vertex strategy, it can be seen that at the same error, $e = 0.027$, the result generated by the projecting vertex strategy is slower and costs more memory than our method.

Buddha. The Buddha Model has about 100K triangles and was tested on a graphic card GeForce GTX 680, Fig, 7. This model is shaded by its normals, which directly reveal the quality of reconstructed normals. We set the time budget to 0.13 ms and 0.11 ms. As shown in the simplification image, the mesh simplification process remove the geometrical details on Buddha's face and belly, and the table below the Buddha model. But after we tessellate and reconstruct the mesh with our moving vertex strategy, as can be seen in the second image, we get a very smooth normal reconstruction of the Buddha model, and with 3 times improvement on rendering efficiency. When we set the time budget to 0.11 ms, the resultant simplified and tessellated mesh has a maximum 10 tessellation factors, and 4.6 average factors. The rendering time of original mesh is 0.36 ms, while our tessellated mesh speeds up 3.4 times. Beyond the performance improvement, we save additional 0.47 MB memory usage. In this model, we bound the memory to be 2.2 MB and then apply the projecting vertex strategy. Result is shown as the third result. The resultant mesh has a larger error and slower rendering time comparing to our result. It demonstrates our method is supreme than the projecting vertex strategy.

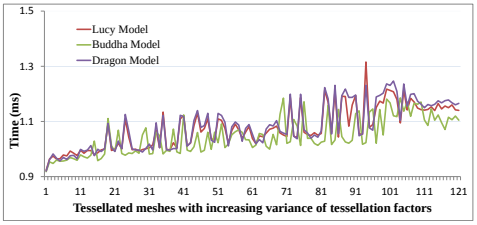


Figure 9: Experiment on variance of tessellation factors

Lucy. The Lucy Model is a model with 150K triangles and tested on a graphic card GeForce GTX 780, (Fig. 8). The wrinkle on her cloth is the most complex part in all example models. From these results, it can be seen that these features have been well reconstructed by our tessellated meshes. Additionally, it also can be seen that when we set the time budget to 0.1 ms, our method produces a simplified and tessellated mesh with a very small error and less memory. The second result shows that the rendering archives 2.3 times faster but has few noticeable errors at sharp edges, as well as reduces memory to 1.6 MB. We also apply the projecting vertex strategy under the 0.07 ms time budget. The resultant mesh quality is much worse and the memory is more than those of our method.

7.3. The Impact of Tessellation Factors on Triangles

In the simplification stage, we introduce a new error metric, the features per triangle, to control the simpli-

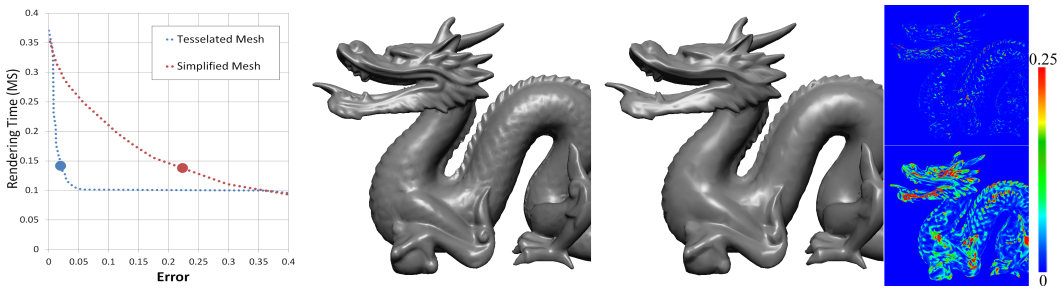
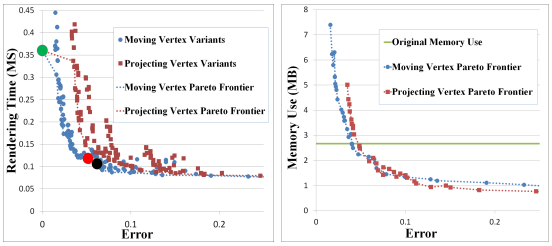


Figure 10: Comparison between the simplified mesh and our final tessellated mesh. Two meshes are rendered at the same rendering time. Our simplified and tessellated mesh (middle left) is much more accurate than the mesh being only simplified (middle right).



(a) Time-error (b) Memory-error

Figure 11: The capability of simplified and tessellated mesh.

fication of triangles. The use of this metric is based on the observation that the more uniform the tessellations are on the mesh, the less time the tessellations are taken. We carry out an experiment to validate this observation. We first simplify these three test models used in our paper into simplified meshes with approximately the same triangle numbers. Then for every simplified model, we randomly set maximum and minimum tessellation factors on the model to generate a tessellated mesh. No matter what tessellation factors are set on the mesh, we always keep the final number of tessellated triangles same, and record the rendering time. Once get the tessellated model, we compute the variance of tessellation factors of triangles over the entire model. For each model, we use such a method to generate 120 tessellated meshes. After generating these meshes, we sort these tessellated meshes according to the variance of tessellation factors, and plot them in Fig. 9. The x-axis is the index number of sorted tessellated meshes, where the index number increases with the increase of variance of tessellation factors. The smaller the index is, the less variance of tessellation factor is. The y-axis is the rendering time. From this experiment, it can be seen that, for all three models, the more uniform the tessellation factors are, the less rendering time is.

7.4. The Capability of Simplified and Tessellated Mesh

To analyze the capability of our simplified and tessellated mesh, we conducted an experiment on the Buddha model using Geforce GTX 680. We enumerated the triangle number of simplified mesh, n_s, and the tessellation factors, l_t. For each pair of n_s and l_t, we then used the algorithm proposed in Section 6, i.e. the linear search of local tessellation factor on each triangle, to optimize the best approximation for the original mesh. The rendering time, memory and error are plot in Fig. 11. The green dot and line are the rendering time and memory usage of the original mesh. The blue dots and red dots are tessellated meshes generated by the moving vertex and projecting vertex strategies respectively. In these two diagrams, we draw out the Pareto frontiers by dash lines. Based on the definition of Pareto frontier, the tessellated meshes on the Pareto frontier are more preferable and better than other tessellated meshes. Thus, the shape of the Pareto frontier indicates the best performance that our method can achieve to trade-off between the error and the time or the memory for this model. From the time-error plot, the maximum speedup is approximately 4 times, and at that time, the error is approximately 0.05. From the memory-error chart, the best memory save is approximately one third and at that time, the error is 0.1.

We also highlight two optimal simplified and tessellated meshes optimized by our method in Fig. 11(a). As can been seen, these two results (marked as red and black dots) are both on the Pareto frontier. Not that results plotted in Fig. 11 are computed by enumerating two global parameters, (n_s, l_t), and linearly searching local l_t on each triangle. Our optimized results on the Pareto frontier demonstrates that our method provides optimal simplified results.

7.5. Comparison

Comparison with Simplified Mesh. To better understand the advantage brought by our tessellation stage, we compared our simplified and tessellated meshes with meshes that are only simplified by [9]. We generated a sequence of simplified meshes of the Dragon model from triangle number 95K to 4K, and

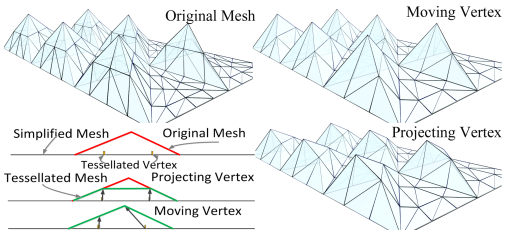


Figure 12: A simple case to illustrate the different of the projecting vertex and moving vertex strategies.

render them to get the rendering times and errors. Then, from each simplified mesh, we compute the corresponding tessellated mesh, and render them as well. The time-error diagram of these two meshes are shown in Fig. 10. The red line is composed by only simplified meshes, and the blue dot line is the Pareto frontier of simplified and tessellated meshes generated by our method. As we can see, after applying the tessellation stage, the tessellated mesh is far more accurate than the simplified mesh. Two meshes with the same rendering time are compared. It demonstrates that the tessellation stage is necessary to better approximate the original mesh.

Comparison with Projecting Vertex Strategy. To better compare the projecting vertex strategy used in [4, 6] with our new moving vertex strategy, we applied the projecting vertex strategy in the experiment that we described in last section, and plotted the rendering time, memory and error in charts shown in Fig. 11. As we can see, the Pareto frontiers of the moving vertex strategy is lower than the Pareto frontiers of the projecting vertex strategy in both of the time-error chart and the memory-error chart. It demonstrates that our new strategy has better performance and quality. When the error budget is 0.05, the moving vertex strategy takes 0.12 ms to render a scene, which is about 1.3 times faster than the projecting vertex strategy. Such an improvement of efficiency at runtime is mainly from the reduction of instruction in tessellation shader and less memory I/O. Specifically, the moving vertex strategy only needs one assign operation, while the projecting vertex strategy needs normal interpolation, normalization and some algorithmic operations to calculate the final position of tessellated vertex. The memory save of our moving vertex strategy is because our strategy is able to achieve good approximations by only using much less vertices.

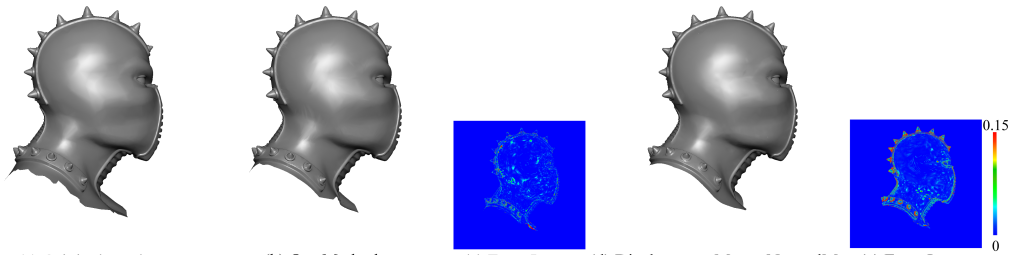
Comparison with Displacement Maps. Our projecting vertex strategy only employs the idea of displaced vertices along normals [4, 6], but does not actually compute in the texture space. To better compare with results tessellated from displacement maps, we use a 3rd-party software, AMD GPU Meshmapper demo [27], to generate the displacement map and the normal map for a low-resolution mesh based on the details from a high-resolution mesh. Results are shown in Fig. 13. The ninja model (Fig. 13(a)) is an example model distributed with Meshmapper, where the original mesh and the coarse mesh are both provided to compute the displacement map and the normal map. In our experiment, we set the resolution of computed displacement map and normal map to 1024×1024, and use 5 inner and outer tessellation factors while render it (Fig. 13(d)). To generate our result (Fig. 13(b)), we first simplify the original mesh into a coarse mesh with the same triangle number of that provided by Meshmapper, and then use 5 as global tessellation factor to compute the tessellated mesh. These two error images (Fig. 13(c)(e)) clearly show that our result is superior, especially at regions with sharp features. The rendering time of our method and displacement map + normal map are 0.18 ms and 0.34 ms respectively. At the same time, our method consumes about 6 MB memory to store positions and normals of tessellated vertices, while the costs of these two maps are 12 MB (normal map) and 4 MB (displacement map).

To better explain why our method is better at capturing sharp features, we show an illustrative example in Fig. 12. We built a mesh with several sharp bumps, and used it to test on our method and the displacement maps. From the tessellated results, it can been see that the moving vertex strategy perfect matches the original mesh, while the offsets along normals miss the key geometric features. A 2D illustration is shown in Fig. 12 bottom left.

8. Conclusion and Future work

Inspired by the advance in graphic hardware, we present a new mesh process and rendering method for realtime high quality rendering. By formulating the mesh approximation as an inverse tessellation problem, we propose a two-stage algorithm to process the mesh, the mesh simplification and the mesh tessellation. Then, we take an optimization to iterate on these two stages to obtain an optimal simplified and tessellated mesh by given a fixed time budget. Results show that our method is capable of achieving 2 to 4 times speedup, but retaining good rendering quality.

In future, there are several directions worthy of further explorations. First, currently, we partition the inverse tessellation approximation into two stages. It will be an interesting and important topic to explore a global approximation solution to optimize the simplification



(a) Original Mesh (b) Our Method (c) Error Image (d) DisplacementMap + NormalMap (e) Error Image

Figure 13: Comparison between our method and the Displacement Map + Normal Map. (a) Original Mesh. (b) Tessellated mesh using our method. (c) Our method's error image. (d) Tessellated mesh using Displacement map + Normal map (e) Displacement map + Normal Map's error image

and tessellation together. Second, view-dependent rendering is an important application of hardware tessellation. The adaption of our algorithm to dynamic and view-dependent tessellations are regarded as interesting future directions.

Acknowledgements

This work was partially supported by NSFC (No. 61472350), the 863 program of China (No. 2012AA011902) and the Fundamental Research Funds for the Central Universities (No. 2012XZZX013).

References

[1] G. Taubin, Detecting and reconstructing subdivision connectivity, The Visual Computer 18 (5-6) (2002) 357–367.

[2] S. Lanquetin, M. Neveu, Reverse catmull-clark subdivision.

[3] J. Sadeghi, F. F. Samavati, Smooth reverse subdivision, Computers & Graphics 33 (3) (2009) 217–225.

[4] I. Guskov, K. Vidimče, W. Sweldens, P. Schröder, Normal meshes, in: Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '00, ACM Press/Addison-Wesley Publishing Co., NY, USA, 2000, pp. 95–102.

[5] R. L. Cook, Shade trees, ACM Siggraph Computer Graphics 18 (3) (1984) 223–231.

[6] A. Lee, H. Moreton, H. Hoppe, Displaced subdivision surfaces, in: Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH'00, ACM Press/Addison-Wesley Publishing Co, NY, USA, 2000, pp. 85–94.

[7] M. Niessner, C. Loop, Analytic displacement mapping using hardware tessellation, ACM Trans. Graph. 32 (3) (2013) 26:1–26:9.

[8] O. Matias van Kaick, H. Pedrini, A comparative evaluation of metrics for fast mesh simplification, in: Computer Graphics Forum, Vol. 25, Wiley Online Library, 2006, pp. 197–210.

[9] M. Garland, P. S. Heckbert, Surface simplification using quadric error metrics, in: Proceedings of the 24th annual conference on Computer graphics and interactive techniques, ACM Press/Addison-Wesley Publishing Co., 1997, pp. 209–216.

[10] H. Hoppe, New quadric metric for simplifying meshes with appearance attributes, in: Proceedings of the 10th IEEE Visualization 1999 Conference (VIS '99), VISUALIZATION '99, IEEE Computer Society, Washington, DC, USA, 1999, pp. –.

[11] A. Willmott, Rapid simplification of multi-attribute meshes, in: Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics, HPG '11, ACM, New York, NY, USA, 2011, pp. 151–158.

[12] J. Cohen, M. Olano, D. Manocha, Appearance-preserving simplification, in: Proceedings of the 25th annual conference on Computer graphics and interactive techniques, ACM, 1998, pp. 115–122.

[13] J. Rossignac, P. Borrel, Multi-resolution 3D approximations for rendering complex scenes, Springer, 1993.

[14] J. Cohen, A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. Brooks, W. Wright, Simplification envelopes, in: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, ACM, 1996, pp. 119–128.

[15] C. Peng, Y. Cao, A gpu-based approach for massive model rendering with frame-to-frame coherence, in: Computer Graphics Forum, Vol. 31, Wiley Online Library, 2012, pp. 393–402.

[16] E. Derzapf, M. Guthe, Dependency-free parallel progressive meshes, in: Computer Graphics Forum, Vol. 31, Wiley Online Library, 2012, pp. 2288–2302.

[17] P. Cignoni, C. Montani, R. Scopigno, A comparison of mesh simplification algorithms, Computers & Graphics 22 (1997) 37–54.

[18] P. S. Heckbert, M. Garland, Survey of polygonal surface simplification algorithms, SIGGRAPH 1997 Course #25 Notes (1997).

[19] D. P. Luebke, A developer's survey of polygonal simplification algorithms, IEEE Comput. Graph. Appl. 21 (3) (2001) 24–35.

[20] P. Cignoni, C. Montani, C. Rocchini, R. Scopigno, A general method for preserving attribute values on simplified meshes, in: Visualization'98. Proceedings, IEEE, 1998, pp. 59–66.

[21] Microsoft, D3D 11 reference., http://msdn.microsoft.com, 2013.

[22] M. Segal, K. Akeley, C. Frazier, J. Leech, P. Brown, The OpenGL Graphics System: A Specification (Version 4.4(Core Profile) - October 18, 2013), http://www.opengl.org/registry/doc/glspec44.core.pdf, 2013.

[23] H. Schäfer, M. Niessner, B. Keinert, M. Stamminger, C. Loop, State of the art report on real-time rendering with hardware tessellation, in: Eurographics 2014-State of the Art Reports, The Eurographics Association, 2014, pp. 93–117.

[24] A. Khodakovsky, N. Litke, P. Schröder, Globally smooth parameterizations with low distortion, in: ACM Transactions on Graphics (TOG), Vol. 22, ACM, 2003, pp. 350–357.

[25] N. Aspert, D. Santa Cruz, T. Ebrahimi, Mesh: measuring errors between surfaces using the hausdorff distance., in: ICME (1), 2002, pp. 705–708.

[26] D. B. West, et al., Introduction to graph theory, Vol. 2, Prentice hall Upper Saddle River, 2001.

[27] AMD, GPU MeshMapper documentation, http://developer.amd.com/tools-and-sdks/archive/legacy-cpu-gpu-tools/amd-gpu-meshmapper/, 2008.