3D Res (2017) 8:6
DOI 10.1007/s13319-017-0117-z

3DR EXPRESS

A Feature-adaptive Subdivision Method for Real-time 3D Reconstruction of Repeated Topology Surfaces

Jinhua Lin · Yanjie Wang · Honghai Sun

Received: 25 January 2017 / Revised: 28 January 2017 / Accepted: 31 January 2017 / Published online: 8 February 2017

Abstract It's well known that rendering for a large number of triangles with GPU hardware tessellation has made great progress. However, due to the fixed nature of GPU pipeline, many off-line methods that perform well can not meet the on-line requirements. In this paper, an optimized Feature-adaptive subdivision method is proposed, which is more suitable for reconstructing surfaces with repeated cusps or creases. An Octree primitive is established in irregular regions where there are the same sharp vertices or creases, this method can find the neighbor geometry information quickly. Because of having the same topology structure between Octree primitive and feature region, the Octree feature points can match the arbitrary vertices in feature region more precisely. In the meanwhile, the patches is re-encoded in the Octree primitive by using the breadth-first strategy, resulting in a meta-table which allows for real-time reconstruction by GPU hardware tessellation unit. There is only one feature region needed to be calculated under Octree primitive, other regions with the same repeated feature generate their own meta-table directly, the reconstruction time is saved greatly for this step. With regard to the meshes having a large number of repeated topology feature, our algorithm improves the subdivision time by 17.575% and increases the average frame drawing time by 0.2373 ms compared to the traditional FAS (Feature-adaptive Subdivision), at the same time the model can be reconstructed in a watertight manner.

Keywords GPU · Feature-adaptive subdivision · Hardware tessellation · Rendering

Electronic supplementary material The online version of this article (doi:10.1007/s13319-017-0117-z) contains supplementary material, which is available to authorized users.

J. Lin (✉) · Y. Wang
Changchun Institute of Optics, Fine Mechanics and Physics, Chinese Academy of Sciences, Changchun 130033, China
e-mail: ljh3832@163.com

H. Sun
University of Chinese Academy of Sciences, Changchun 130033, China

J. Lin
Changchun University of Technology, Changchun 130012, China

1 Introduction

Nowadays, hardware tessellation shows a strong ability in GPU (Graphics Processing Unit) graphics pipeline [1, 17]. Researchers have proposed a variety of offline rendering methods to adapt to this new GPU rendering pipeline [14]. Stam [19] proposed a precise calculation method for Catmull–Clark subdivision surface in 1998. The subdivision surface was treated as a parametric surface. The parametric surfaces adapted

well to the patch processing mechanism of GPU hardware subdivision unit, but Stam's method could not be used for the feature region such as arbitrary vertices or creases, both the vertex and normals at boundaries could not be exactly calculated bit-by-bit due to the influence of feature space mapping. In order to solve this problem, several approximate subdivision methods were introduced. Vlachos et al. [21] proposed a PN triangle method. Only the coordinates and normals of the three vertices in each triangular patch were used to create a cubic Bezier triangle. The PN triangle appeared earlier than hardware tessellation, but it was still suitable for the GPU hardware tessellation pipeline. Although this method had improved the rendering effect of triangular meshes, the rendering rate decreased exponentially with the increase of the number of triangular patches, PN triangle was not the best method for representing the approximate surface. Loop et al. [7] proposed that the subdivision surface could be expressed as a bi-cubic Bessel patch by inserting the vertices and tangent planes that generated from Catmull–Clark subdivision rule. The advantage of using the approximate Catmull–Clark subdivision surface algorithm is that there are many off-the-shelf tools for creating Catmull–Clark control meshes, but the disadvantage is that the resulting mesh is not crack-free. In 2008, Loop et al. [8] proposed an ACC-2 (Approximating Catmull–Clark) algorithm based on the Gregory block, which was a fast method by inserting finite coordinates at corners, smoother approximations were obtained at other locations. However, the subdivision model created by this approximation method still had inconsistency phenomenon. He et al. [4] proposed an algorithm to reduce this inconsistency. Thus, the hardware tessellation for watertight Catmull–Clark surface rendering had become an important research issue [12, 19, 24]. Nieener et al. [11] proposed a Feature-adaptive subdivision method in 2012, which was an algorithm for rendering Catmull–Clark surface with crease edges. It combined the technology of table-driven subdivision [5, 22] and hardware tessellation. Schafer et al. [16] proposed a dynamic FAS method in 2015, which assigned different tessellation factors to the feature region, it greatly improved the rendering speed. However, the decrease of subdivision depth leads to the decrease of the rendering precision in some feature region.

The traditional FAS method establishes the bi-cubic B-spline patches only in feature regions so the number of rendering patches are reduced. But when the irregular region appears repeatedly, the conventional FAS method will subdivide feature region to the same depth one by one. Although the details of each irregular region can be drawn accurately, the efficiency of traditional FAS method will also be reduced due to the large number of repeated cusps or creases.

In this paper, an improved Feature-adaptive subdivision method is proposed based on Octree-primitive, an uniform Octree-primitive data structure is establish in feature regions which have the same arbitrary vertices or creases. When these feature points appearing repeatedly on the Catmull–Clark subdivision surface, a set of Octree-primitives similar to the structure of bi-cubic B-spline patches are build to guarantee reconstruction precision, this uniform Octree-primitive for a large number of repeated feature points will improve the efficiency of hardware tessellation. Then, the breadth-first strategy is used to re-encode the meta-data, and the meta-table is generated to GPU hardware tessellation unit to realize the fast reconstruction [15].

2 Related Work

2.1 GPU Hardware Tessellation Unit

GPU is modern graphics processor in a single instruction and multiple data mode [3, 20]. Our algorithm is designed upon the GPU to finish subdivision for repetitive feature region, combining the traditional CPU with GPU graphics pipeline to achieve real-time reconstruction. GPU graphics pipeline is composed of five kinds of programmable shaders, as shown in Fig. 1, in which the hull shader, fixed pipeline tessellation and domain shader constitute the GPU hardware tessellation unit. We use Direct 3D application programming Interface to do programming in GPU [10, 24]. The principle of GPU graphics pipeline is briefly described by five steps.

First of all, the vertex shader is responsible for 'vertex-by-vertex' operations. It receives an input vertex and then produces an output vertex. It works like an input streamline including coordinate transformation, skinning, animation, and vertex lighting.

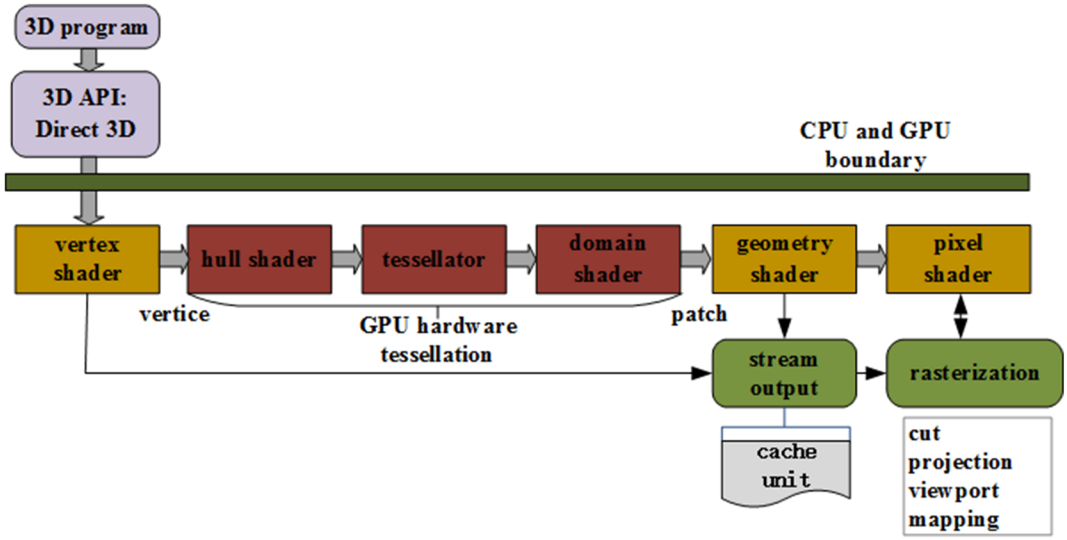


Fig. 1 GPU graphics pipeline (Direct3D 11 pipeline). GPU pipeline allows for real-time communication between CPU applications and screen. GPU hardware tessellation unit is composed of hull shader, tessellator and domain shader, in which the hull shader and the domain shader are programmable. We use Direct 3D application programming Interface to do programming in GPU

Second, GPU hardware tessellation unit introduced by Direct3D 11 is composed of three stages. They are shell shading, fixed pipeline tessellation and domain shading. Hardware tessellation receiving part of input control vertices produces a large number of output polygons. It's able to save GPU memory. The simple input mesh is converted into patch style in the stage of hull shading. During this stage, a number of patch-by-slice calculations is being done to guarantee the next two sub-stages. Fixed tessellator consists of a fixed pipeline function that subdivides a primitive (quadrilateral, triangle, or line segment) into smaller one. These primitives are tiled in a canonical coordinate domain. For example, a square domain is subdivided into a series of squares. The domain shader outputs the coordinate information for each vertices within the patch.

Once again, the geometry shader is a new application-specific stage introduced by Direct3D 10. It generates new output vertices based on input vertices. In this stage, the input is a primitive, and the output is a set of vertices with the same topology. The outputs will be attached to the stream object. For example, triangular patches.

At last, pixel shader allows for pixel-by-pixel output through constant, texture, interpolation and something else, such as pixel-by-pixel illumination and post-processing. The pixel shading program will be executed once a time for each pixel by Raster processor. It also can be programmed to skip this step.

2.2 Repeated Topology Surface

GPU hardware tessellation allows for high order primitive. Especially parametric patches will be mapped from quadrilateral or triangular mesh to three-dimensional space. This step can be finished by programming the hull shader and the domain shader. According to the Catmull–Clark subdivision rule [2], the algorithm subdivides the initial control mesh repeatedly to obtain a smooth bi-cubic B-spline surface. Repeated topology surface is a kind of bi-cubic B-spline surface where a large number of feature topology are repeated. The topology of the patch can be a feature region containing arbitrary vertices or crease edges, as shown in Fig. 2a. As the figure shows, the pineapple model as well as the XO model consists of a large number of repeating feature detail, which is represented by repeated bi-cubic B-spline patches. In Fig. 2b, the vertex mapping of the model is shown. The symmetrical structure between adjacent meshes can be clearly seen. Our algorithm performs real-time subdivision rendering for this kind of feature surface with repeated topological structure. The bi-cubic B-

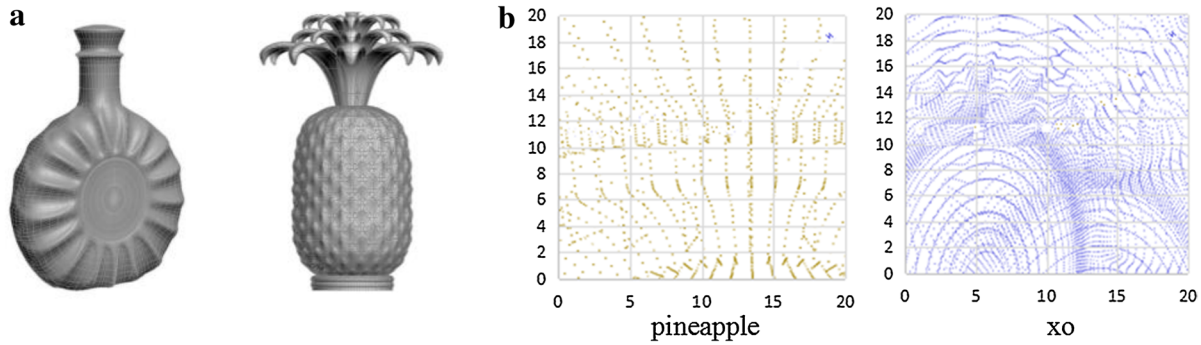


Fig. 2 Repeated topological feature model. There are a large number of repeated feature points on the surface of XO and pineapple. These feature details are represented by repeated bi-cubic B-spline patches. The graph shows the 2D distribution of the vertices in repeated feature region, the symmetrical structure between adjacent meshes can be clearly seen. **a** XO model and pineapple model. **b** The 2D mapping of repeated region

spline surface is a segment mapping [9] between $[0, m+1]$ and $[0, n+1]$:

$$P(u,v) = \sum_{i=0}^{m} \sum_{j=0}^{n} \mathbf{d}_{i,j} N^3(u-i) N^3(v-j). \quad (1)$$

$\mathbf{d}_{i,j}$ is a control vertex matrix for B-spline surface, $N^3(t)$ is a cubic B-spline basis function, and the cubic polynomial curve is composed of four nonzero curves. Since these basis functions are tangent continuous (C^2 continuous), their linear combinations are still C^2 continuous. Each bi-cubic B-spline surface is determined by 16 control vertices. The adjacent patches own the same 12 control vertices, which are used to determine the vertex coordinates and the normals at the boundary. This makes it easier to construct a repeating topology feature surface with continuous curvature.

3 The Building of Octree Primitive

For the repeated feature region of initial mesh, a corresponding Octree primitive is built to cover the geometric information of the patch. The patches to be subdivided store the primitive constant function. The patch geometry information is composed of three kinds of information, they are three-dimensional vertex coordinates, displacement mapping information and texture attributes of high-order parametric patches. The primitive constant function is used to generate a tessellation factor for the feature region. For other recurring feature regions, the first Octree primitive is used to generate their own meta-table.

3.1 Octree-primitive Data Structure

The octagonal tree structure is used to convert the feature region into a three-dimensional primitive logically, as shown in Fig. 3a. The sub-primitive region is established to correspond to each subdivision level, as shown in Fig. 3b. The access to each sub-region keeps corresponding to the subdivision around arbitrary vertices. The algorithm uses the index pointer and octal-linear codes to build the Octree-primitive. The codes are written into GPU hardware tessellation unit to drive reconstruction.

There are several steps will be done to keep the Octree-primitive topology matching with the feature region. First of all, the irregular patches are stored as a triangular patch table, which is used to store the property of patches. Second, the index list is used to make the tree-shaped topology list converting into a Octree-primitive structure, as shown in Fig. 4. Each node contains eight data elements representing the sub-primitive. While all the vertices in the sub-primitive have the same texture information, the vertex geometry information is stored into the corresponding data element. While the arbitrary vertice is included into the element, the corresponding data element holds a pointer to the next sub-primitive.

At last, the octal-linear method is accomplished to encode the Octree-primitive index list. The vertex information for the next stage of the algorithm is generated based on the index list. Octree-primitive

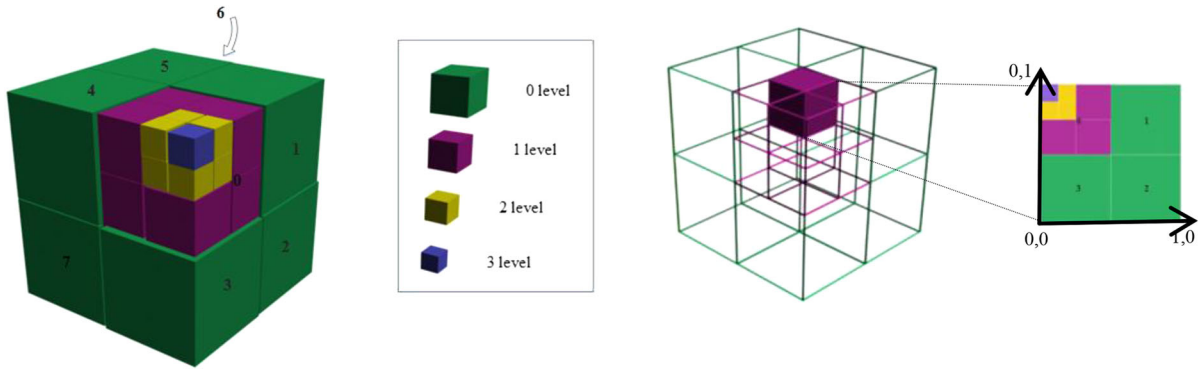


Fig. 3 Three-dimensional Octree primitive model (arbitrary vertice valence 3). The subdivision details are shown level-by-level in an Octree-primitive area, *colors* denote different subdivision levels. A subordinate primitive is highlighted to display the patches around an arbitrary vertex in local coordinate. **a** Octree-primitive, **b** sub-primitive region. (Color figure online)

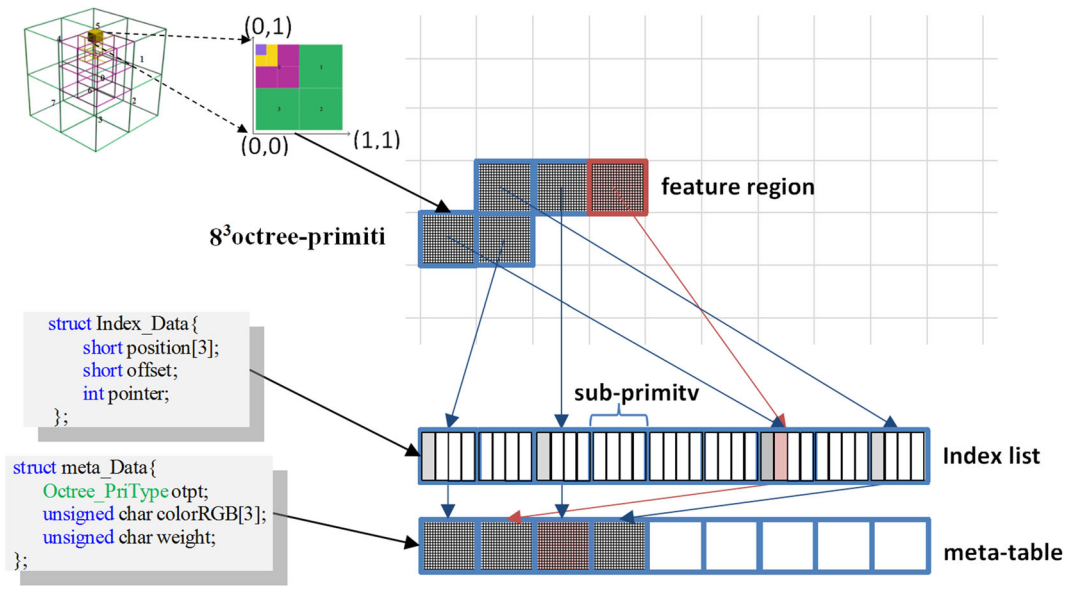


Fig. 4 Octree-primitive index list. Each node contains eight elements representing the sub-primitive, when all the vertices in the sub-primitive have the same texture information, the vertex geometry information is stored into the corresponding element. The arbitrary vertice is also been included into the element, the corresponding data element holds a pointer to the next sub-primitive

data structure is shown in Fig. 5. Octree primitive construction procedure is shown in Fig. 6.

The octal-linear coding in the primitive list is to facilitate the geometric transformation. It is time-saving for the shader to get the primitive data quickly before transferring to GPU hardware tessellation unit. Each node has the same code length. According to the sub-primitive code, its parent-primitive code as well as the local origin coordinate can be generated. These coordinates include eight vertices among Octree-primitive owning arbitrary vertices with the smallest coordinate values.

3.2 Octree-primitive Access Operation

3.2.1 The Octree-primitive Being Inserted into the Index Table

First, the index value corresponding to the Octree-primitive block is calculated, and the target sequence

```
typedef struct Octree_PriType{
 PType pbox;
  //primitive type
 Vertex po;
  //primitive vertex coordinate
 char * occode;
  //octal-linear code
 int l;
  //primitive tessellation factor  l = 2^i,
    i is subdivision level
 RGB texture;
  //primitive texture
 struct Octree_PriType * next;
  //recursively points to the next primitive
    (quadratic primitive)
 struct Octree_PriType * child;
  //recursively points to sub-primitive
 Tlist * Trintermesh;
  //adjacent primitive pointer
}Octree_PriType;
```

Fig. 5 Octree-primitive data structure

is determined. Second, the sequence is traversed, including the list attached to the sequence. When an element with the same position as the world coordinate is found, an index is returned, otherwise, the first null position in the sequence is searched. When a position in the sequence is available, a new Octree-primitive is inserted. Finally, when the sequence is full, the voxel item is appended to the linked list of the sequence, as shown in Fig. 7.

When the GPU inserts the Octree-primitive into the sequence in parallel, in order to avoid the preemptive phenomenon, when an Octree-primitive finds an empty position, the algorithm locks the sequence where the Octree-primitive is located, and other primitives can not be written. When the sequence is locked to write, the allocation of other Octree-primitives within the sequence is slow down until the processing of the next frame of data begins. In this paper, it is proved that because the volume reconstruction method supports the sequence-independent updating, the slight delay does not cause the reconstruction quality to decrease.

3.2.2 *Reading the Sub-Primitive from the Index Table*

In order to find the position of the sub-primitive in the index table, firstly, the index address value of the sub-primitive is calculated; secondly, linear traversal is carried out in the sequence according to the index value; when the specified item is not found in the sequence, Finally, since the deletion of the primitives causes fragmentation to occur, the traversal will be repeated until the empty position is found.

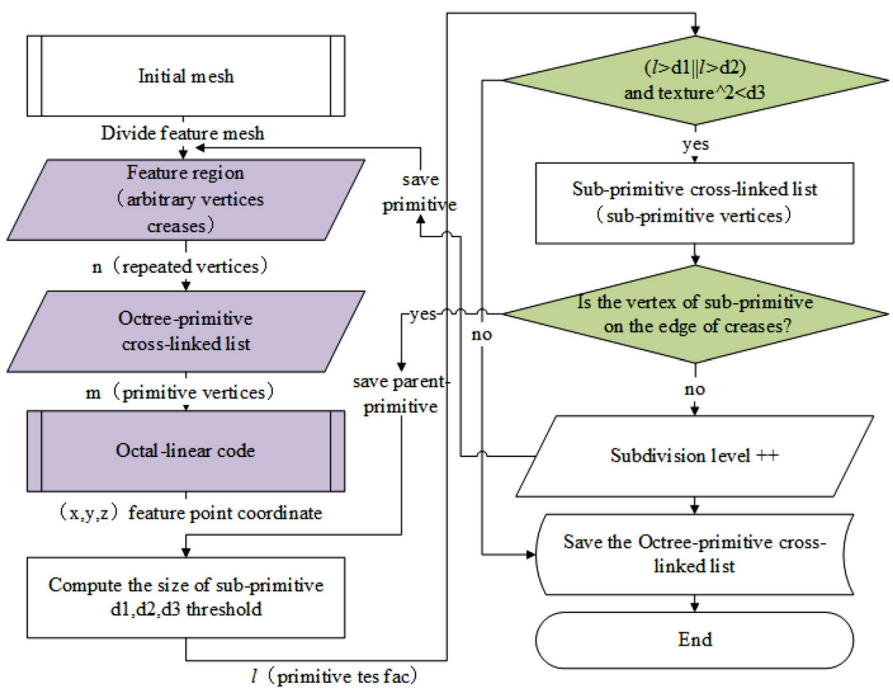


Fig. 6 Octree-primitive construction procedure. Our algorithm improves the FAS by converging repeated feature regions to an uniform Octree-primitive. Three steps are being done to construct the Octree-primitive, as shown in *purple box*. (Color figure online)

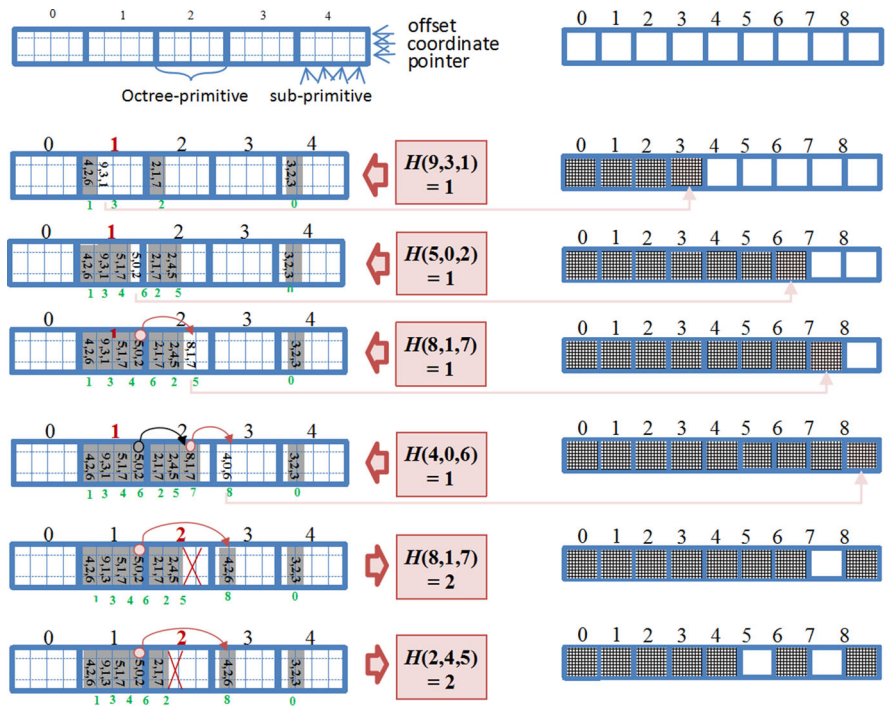


Fig. 7 Octree-primitive access operation. When multiple primitives are mapped to the same index cell, a conflict occurs. In this paper, the indexing unit is divided into a sequence of multiple spatial voxels, each of which is assigned a unique indexed address. Each sub-primitive corresponds to a unique voxel of world coordinates. While a collision occurs, the pointer of the sub-primitive is stored in the next unoccupied memory location in the sequence

3.2.3 Delete the Sub-primitive in the Index Table

Similar to the insert operation, for a given world coordinate position, we first calculate the index address value, then traverse the sequence corresponding to the sub-primitive linearly (including linked list). Then, when the specified sub-primitive is found and the primitive is not linked, the index entry is deleted. When the sub-primitive is the last element of the sequence and there is a non-zero offset indicating that the primitive is the head of a linked list, the address of the index entry being pointed by the offset is copied to the sequence, and removes the sub-primitive from the sequence. Finally, due to the modification of the linked list, the algorithm locks the sequence and delays the associated operations of the linked list, specifying the next frame position.

3.3 Primitive Constant Function

The primitive constant function is used to generate tessellation factor for each irregular region, which is divided into two types of regions including triangle and quadrilateral. The number of vertices outside of the triangle is 3, and the vertices outside of the quadrilateral is 4. Similarly, the number of vertexes inside the triangle is 1, and the number of vertexes inside the quadrilateral is 2. Two regions own different numbers of edges and vertices. The connection between regions is determined by transform patches. In order to achieve seamless connection between patches, tessellation factor of the irregular region must be consistent with adjacent patch.

During the stage of GPU hull shading, a dynamic tessellation factor is assigned to each patch. The simple mesh is converted into a patch, and the data needed for tessellator and domain shader is computed patch-by-patch. Tessellation factor being used to control seamless connection as well as the subdivision depth. The octree structure is used to logically fit the depth and irregularity of the patch. The vertex of the octree is correspond to the arbitrary vertice in feature region. The new vertexes born from each subdivision level is correspond to the vertices on the octree. The maximum subdivision depth of GPU is 6. The subdivision depth to each feature vertex is determined by the layers of octree, then the dynamic subdivision factor is generated. The algorithm set the tessellation factor as t, subdivision depth as d. According to the type of octree-primitive, there are three kinds of calculation. First, if there are arbitrary vertices with valence three, then d is 3, and t is 8. Second, if the

valence is 5, then d is 6, and t is 32. At last, if there are crease edges, then d is 5, and t is 32.

4 Breadth-first Strategy

4.1 Meta-table Generation

The algorithm uses the top-down, out-in strategies to access the data element of octree primitive. Since the primitive is represented by the index list, we only need to get the pointer to access each feature primitive and the geometry information toward each feature vertex. The access sequence is stored in the meta-table for next GPU stage. The access numbers are generated from Catmull–Clark subdivision rule, as shown in Fig. 8.

The control points are subdivided to generate Catmull–Clark surface in feature region. The initial control points being encoded by 0–15 numbers are correspond to 16 control vertices on the Octree-primitive, as shown in Fig. 9. Three groups of access sequences are presented based on the breadth-first strategy toward the octree structure of feature region. The sequences being stored into meta-table are correspond to the points of face, edge and vertex. Our algorithm achieves the real-time subdivision through the meta-table programming the domain shader which is a part of GPU hardware tessellation unit.

4.2 Meta-table Structure

According to the access sequences which is produced from the breadth-first strategy, meta-table is created for each patch during the pre-processing stage. Since

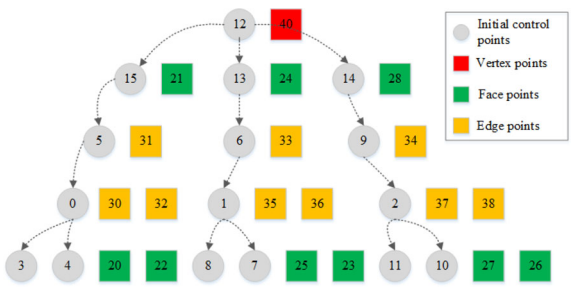


Fig. 8 Encode for an Octree-primitive around the arbitrary vertex with a valence of three. *Colored squares* show vertex points in *red*, face points in *green*, and edge points in *yellow*, based on the rules of Catmull–Clark. (Color figure online)

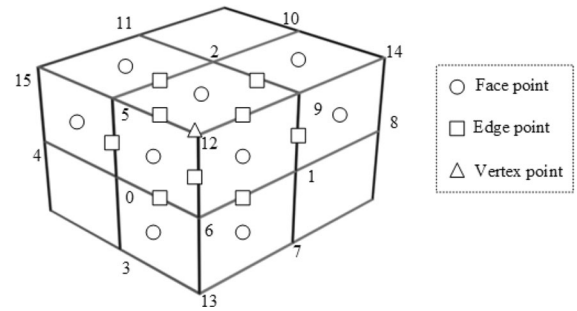


Fig. 9 Control points in Octree-primitive. Octree-primitive encoded by different kinds of vertex, it's 3D version of Fig. 8, omitting the number of *colored* points. (Color figure online)

the Octree primitive is established to match the feature region topology, the geometry information of vertex being stored into meta-table have been ordered in subdivision rule. Each element in the meta-table represents a control point number for patches to be subdivided. GPU domain shader gets the geometry information of vertexes based on the control point number. The level-by-level subdivision is being done by the fixed tessellator. The algorithm is going into the next stage of GPU rendering pipeline.

The meta-table is stored as a texture map in RGB format [6]. The d row texture includes the texture information of the d level patches which is used to produce the indices of the patches in $d+1$ level, as shown in Fig. 8. For an ordinary edge point E with depth 2, three values need to be find including (2,0).r, (2,0).g, (2,0).b. These three values being contained with the index of the four vertexes in the patch texture contribute to the ordinary edge in the template; for an ordinary vertex point V. The values including (3,0).r, (3,0).g, (3,0).b, (3,1).r, (3.1).g, and (3.1).b need to be find, these six values being contained with the index of the seven vertexes in the patch texture contribute to the ordinary vertex in the template. For an boundary edge point E_1 with depth 4, only two values need to be find, they are (4.0).r, (4.0).g.

The algorithm specifies the maximum degree of vertices as eight. In this way, the first nine RGB channels can satisfy the largest Catmull–Clark subdivision template. The template type is stored in the tenth channel. The eleventh channel is for markings such as subdivision level information, crease edge and boundary information. The first row of meta-table corresponds to a vertex with degree n in the center position, which is determined by the adjacent n

vertices as well as itself. The location of vertexes keeps in queue in the GPU memory, so the other vertex locations are been obtained through ascending order. The first row of the meta-table can be omitted.

While a new subdivision hierarchy is added, the index for the new vertex is simply appended to the meta-table. The d level meta-table owns more items than the $d-1$ level. When the meta-table reaches the maximum subdivision level, all lower-level indices are available.

5 Algorithm Overview

This presentation is composed of two stages including preprocess and hardware tessellation stage. In the first stage, we get the initial 3D model from depth sensor. Then, the model is divided around the arbitrary vertexes as well as the crease boundary edge to produce patches for hardware tessellation. The Octree primitive is used to generate the meta-table based on the feature patches divided before, while the same Octree-primitive is used to the repeated feature region. Then the vertex information in the initial control mesh are transferred to the GPU hull shader together with meta-table.The hull shder dispatches a thread to each patches. Every input vertexes share the same hull shading program among patches. The number of input vertexes is limited to 32. In the second stage, each feature mesh is subdivided by the GPU tessellator, and the newly generated vertex is calculated as the input of GPU domain shader. Domain shader processes the properties of vertex including texture coordinate as well as tangent information based on the tessellation factor, then outputs the mesh. Finally, GPU rendering pipeline is used to draw the final model.

The processing stages between CPU and GPU is shown in Fig. 10. The control mesh is divided into ordinary patches and feature patches. The patches are continuously subdivided in the GPU pipeline (parallel processing in multi-channel GPUs). The final patches strictly matching boundaries can be directly rendered.

The preprocessing stage is done in CPU, while the subdivision is performed in GPU hardware tessellation unit. The data for reconstruction is stored in the frame buffer. During the process of rendering, the data transferring between CPU and GPU is finished by five steps. In the first step(preprocess stage), the feature patch and the meta-table are transferred from the CPU memory to graphic memory in the form of texture, as shown in Fig. 9. In the second step(tessellation stage), the patches to be subdivided are obtained from graphic memory through the indices accessing from meta-table. The new vertices boring from Catmull–Clark rule are transferred to frame buffer. In the third step, the vertex information is transferred from frame buffer to texture unit for the next time of subdivision. In the fourth step, according to the tessellation factor, the feature patch is subdivided to the maximum level. The data being ready to render the finial mesh is delivered back to CPU memory. In the last step, the information including vertex data, normal vector, illumination parameters and displacement mapping data are transferred to GPU graphic pipeline for reconstruction.

6 Results and Discussion

6.1 Experimental Environment

Operating System: Windows 8.1
Programming language: C + + .
Development tools: Microsoft Visual Studio 2014.
3D graphics programming interface: Direct3D 11.
CPU type/frequency: INTEL Core i5/3.0 GHz.
Memory: 8G.
Graphics Processors: NVIDIA GTX 960 [13], GTX 960 graphics processor owns 11 pixel pipeline, support OpenGL4.4/DirectX11 [11].
Depth sensor: Asus Xtion Pro,RGB-D 30 Hz.

6.2 Simulation Test

6.2.1 Comparison to FAS for Refinement

The base model is composed of quadrilateral meshes including 11,724 triangular patches, as shown in Fig. 11a. The feature region is surrounded by a rectangular box, and the zoom picture shows that the arbitrary vertices appear in the irregular region near the cusp. The octagon primitive is emphasized by the grad hexagon mark. An Octree-primitive is produced from this octagon patch. The GPU hardware tessellation unit is driven by the meta-table generated from Octree-primitive. The result of rendering shows that our algorithm is similar to FAS. The differences

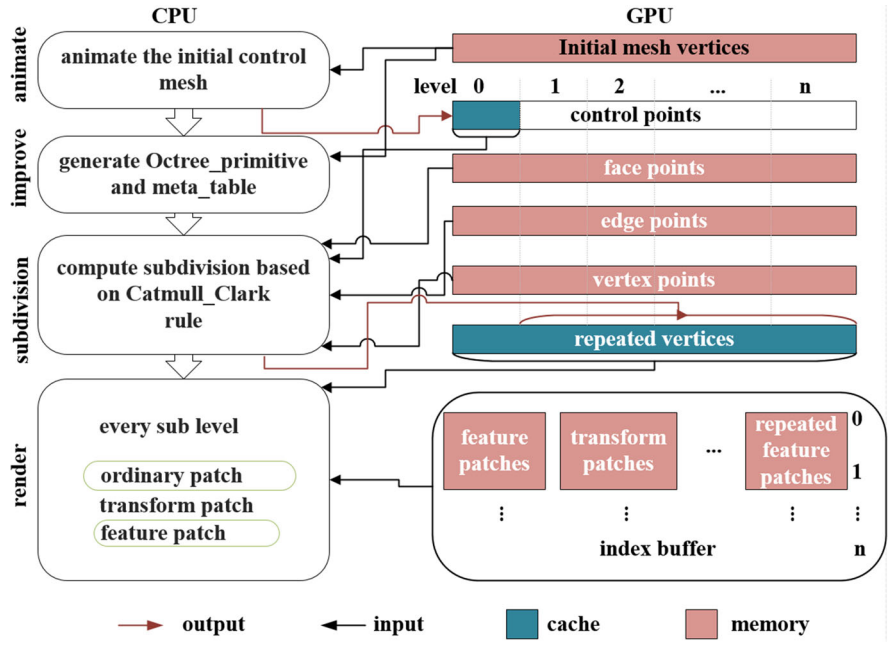
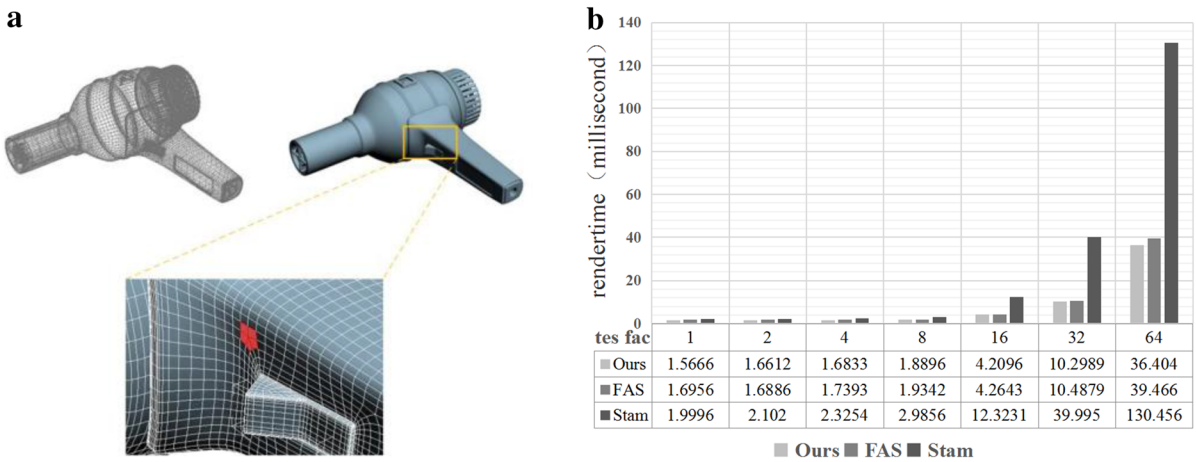


Fig. 10 The reconstruction framework. The pre-processing stage is done in CPU, while the subdivision is performed in GPU hardware tessellation unit. The data for reconstruction is stored in the cache and GPU memory



tes fac	1	2	4	8	16	32	64
Ours	1.5666	1.6612	1.6833	1.8896	4.2096	10.2989	36.404
FAS	1.6956	1.6886	1.7393	1.9342	4.2643	10.4879	39.466
Stam	1.9996	2.102	2.3254	2.9856	12.3231	39.995	130.456

Fig. 11 **a** Rendering of the hair dryer model. The hair dryer mesh is rendered by our algorithm, the zoom region shows an irregular patch corresponding to an Octree-primitive in *red* color. **b** Comparison of three algorithms in different tessellation factor applied to the hair dryer model

between two algorithm can not be distinguished by naked eyes.

In Fig. 11b, the time required to render the hair dryer model is shown by using traditional FAS algorithm, Stam algorithm and our algorithm. When the tessellation factor is less than 8, our algorithm is equivalent to FAS algorithm. When the subdivision factor reaches 64, our algorithm draws faster than FAS algorithm. The histogram shows that when the same model is drawn under a higher subdivision depth, our algorithm is more outstanding than the FAS. That means the more deep the subdivision done, the more quick our algorithm subdivided. The result shows that the classical FAS algorithm uses a uniform subdivision depth for every irregular patches, and the rendering efficiency plays well, but the patch computation is larger. Our algorithm extends the traditional FAS algorithm, the Octree-primitive matches the irregular blocks in repeated regions efficiently. The matching result plays better than FAS. The

computation of our algorithm is small and the rendering effect is more realistic.

6.2.2 Comparison to FAS for Memory Requirement

The traditional FAS algorithm allocates a certain amount of GPU memory for each feature region to store the subdivision table. It also needs a large enough buffer to store the vertex information of each level. So when the feature region repeats normally, the subdivision table and the number of vertex in each subdivision level will be increased. The GPU occupancy is increased. In order to reduce the occupancy rate of GPU and improve the efficiency of rendering, Our algorithm distributes an uniform Octree-primitive to the repeated feature region, reducing the memory space occupied by GPU and CPU for large amount of data transferring.

See from the Fig. 12a, the memory requirement of models including XO model, candlestick model, vase model, pineapple model and hair dryer model (see Fig. 13) are shown in different subdivision depths. The memory occupied by our algorithm is not significantly increased when the number of repeated cusps or creases on the model is increased.

In Fig. 12b, both algorithm is used to subdivide the pineapple model and the XO model. The result shows that when the arbitrary vertice or crease edge appears repeatedly. Our algorithm's GPU occupancy rate is also been reduced. It is mainly due to the GPU hull shader programmed to the repeated feature of the different patches in uniform processing saving the GPU cache memory. Therefore, our algorithm ensures real-time rendering more efficiently. The GPU cache is used to maximize when complex high-order model being rendered.

6.2.3 Comparison to FAS for Render Time

Our algorithm and FAS algorithm are applied to the four kinds of rendering models. The rendering result is shown in Fig. 13. The number of irregular patches being rendered is calculated to evaluate the processing ability of two algorithms, as shown in Table 1. XO model is composed of 40,640 quadrilateral patches. The pineapple model is composed of 50,210 quadrilateral patches. The candlestick and vase model are composed of 19,166 and 29,280 patches. These models with significant repeating topological characteristics are rendered to evaluate the render time of two algorithms, which can effectively evaluate the rendering ability.

In Fig. 13a, XO model is the model with the largest number of repeated crease edges compared with the other four models. The subdivision time of the XO model is about 0.5 ms per frame, and the FAS algorithm is about from 0.5 to 0.6 ms. Since the FAS algorithm uses the same subdivision depth for each feature region, the subdivision time will be increased when the crease edge is repeated in large area. Contrast to the traditional FAS method. Our algorithm using an uniform Octree-primitive to match the feature region is more competent for the rendering of surfaces with large number of repeated features such as arbitrary vertices or crease edges.

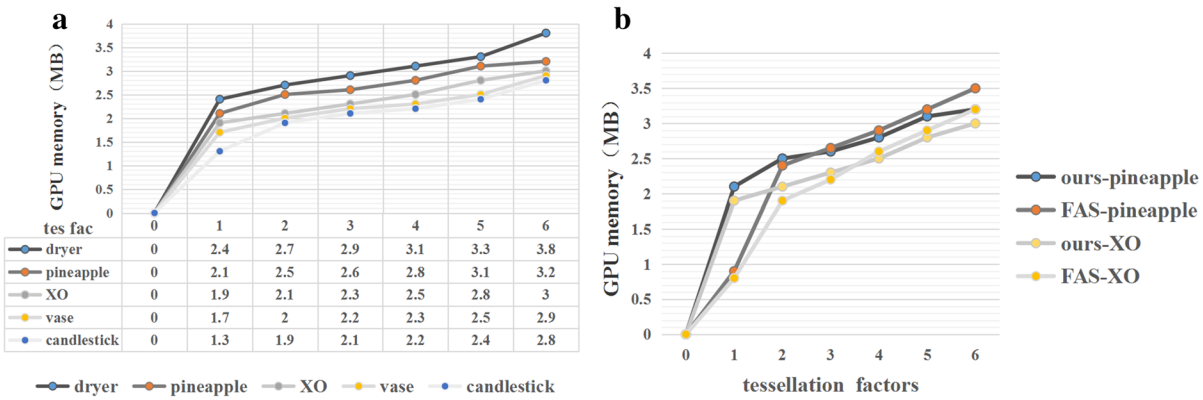


Fig. 12 Comparison of GPU memory requirement. **a** GPU memory requirement of rendering five kinds of models using our algorithm. **b** Comparison between our algorithm and FAS in memory requirement for XO and pineapple model

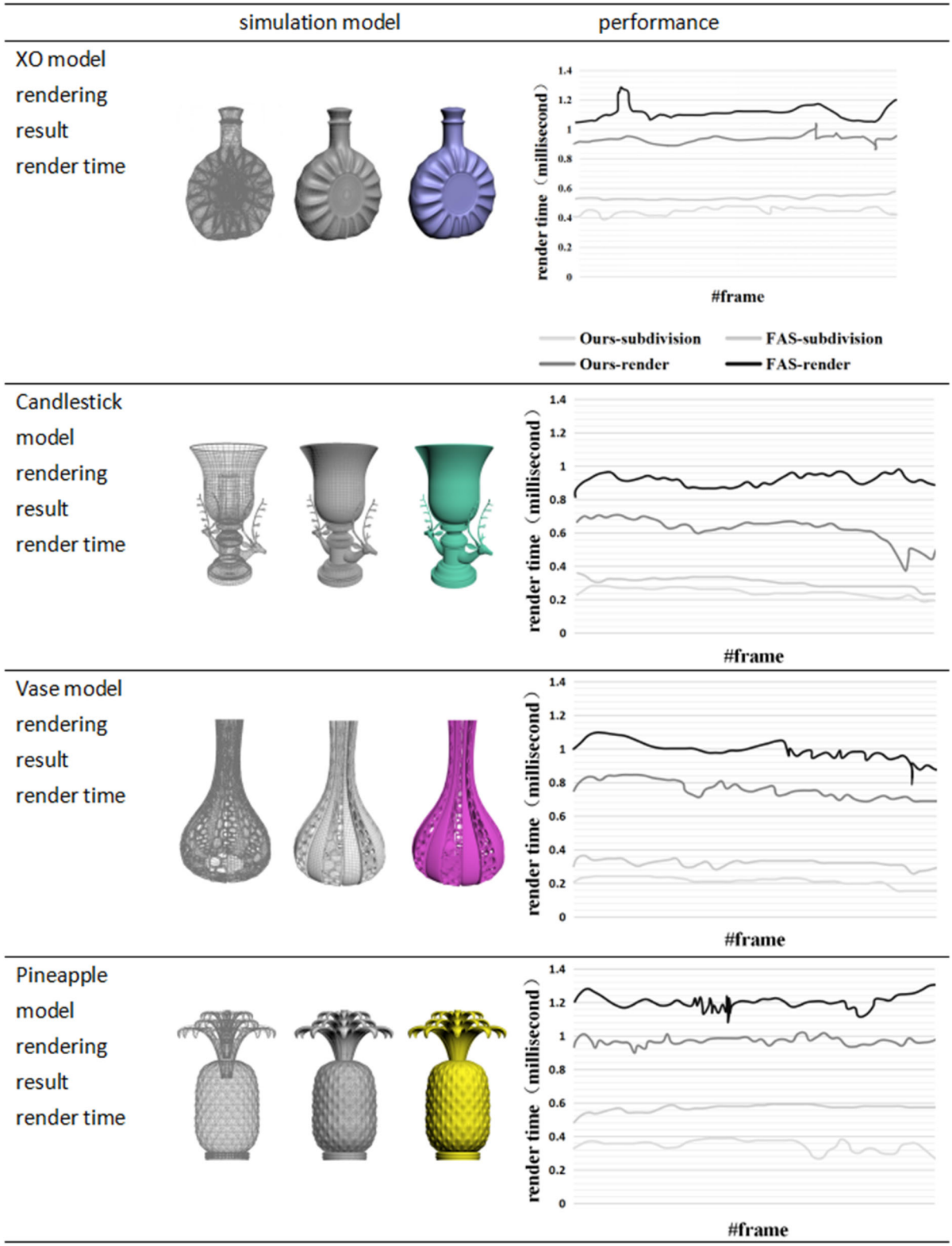


Fig. 13 Comparison between our algorithm and FAS method in rendering performance

Table 1 Comparison between our algorithm and FAS in terms of subdivision and render time for the models shown in Fig. 13 (ms)

Model	XO		Vase		Pineapple		Candlestick	
Base patches	1882		1550		2260		1310	
Feature patch	872		742		902		664	
Feature vertex	398		274		480		244	
Algorithm	FAS	Ours	FAS	Ours	FAS	Ours	FAS	Ours
Render time	0.623	0.562	0.611	0.536	0.687	0.624	0.627	0.58
Subdivision time	0.576	0.392	0.391	0.258	0.598	0.388	0.388	0.212
Sub-time	0.245		0.208		0.273		0.223	
Draw total	1.199	0.954	1.002	0.794	1.285	1.012	1.015	0.792

Similarly, in Fig. 13g, the pineapple model contains the largest number of arbitrary vertices comparing with other three modes, and these arbitrary vertices repeat the same topological structure. In our algorithm, the Octree-primitive is constructed only for the first feature region containing arbitrary vertices. The render time will be more reduced when the repeated region is more complicated in topology. The experimental result shows that the overall rendering time of our algorithm is 0.2 ms less than that of FAS.

Analysis of the data in Table 1 leads to the following conclusions. First, in spite of the additional time spent to the construction of Octree-primitives, our algorithm performs better than that of FAS in terms of the subdivision time for repeated feature regions. Second, four kinds of models with repeated topological characteristics are selected to compute the sub-time between the FAS algorithm and ours. The more repeated feature the model owns, the faster the model is drawn by our algorithm.

6.3 Reconstruction result

In this presentation, our algorithm performs high-quality 3D reconstruction at higher frame rates, as

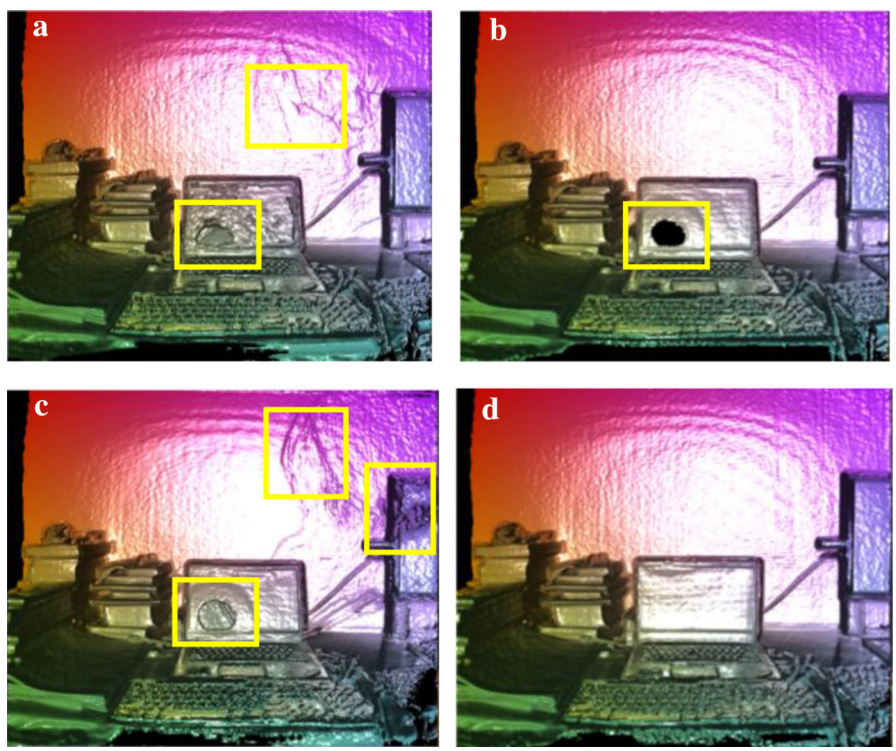


Fig. 14 Desk scene used to compare the performance of several reconstruction system. **a** Whelan algorithm, **b** Whelan algorithm (4mm), **c** Newcombe algorithm, **d** our algorithm

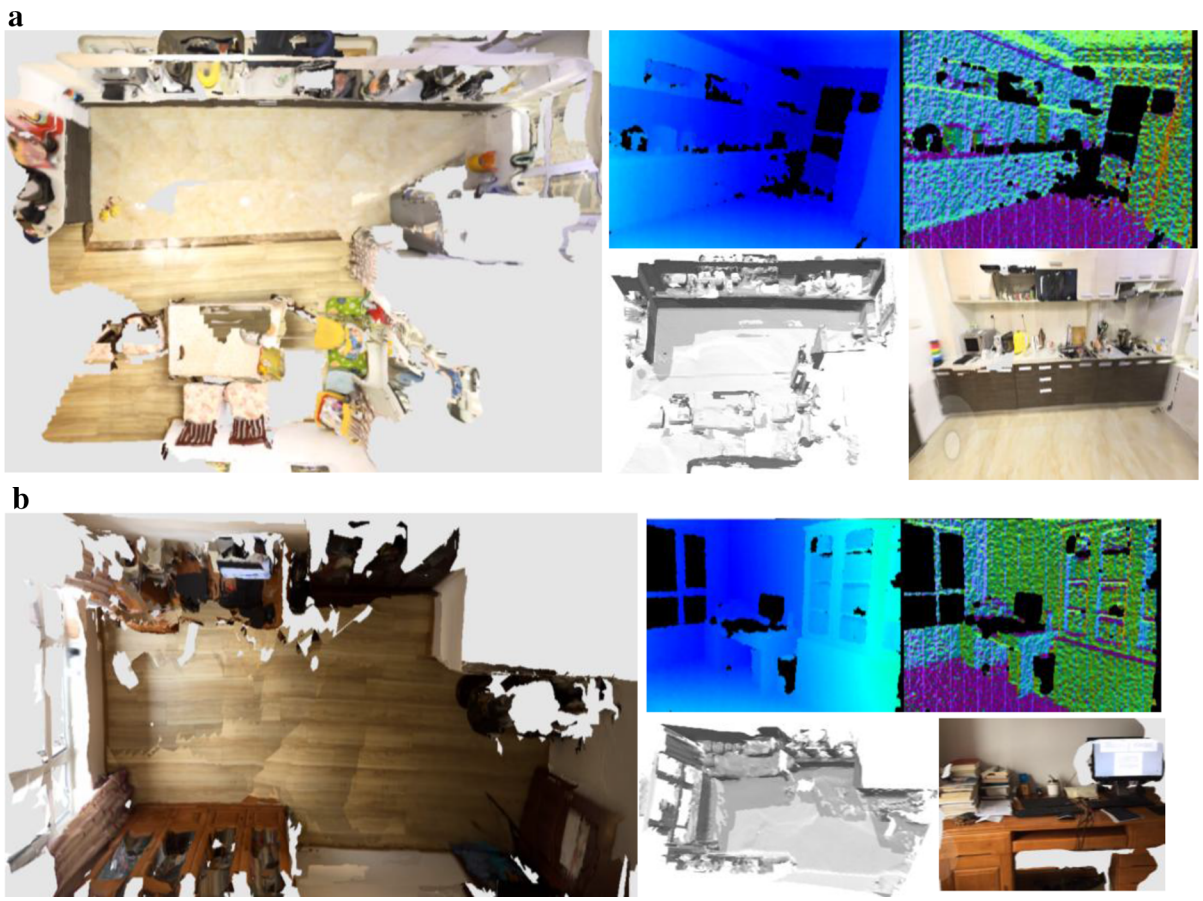


Fig. 15 Scenes used to test the performance of our reconstruction system. **a** The dining room reconstruction is shown from *top view*, the reconstruction quality is acceptable trading off with the reconstruction speed. This large scale scene is composed of 199,958 faces, and the *right* screenshot shows the depth map, normal map, fusion map and zoom map. **b** The office room is reconstructed from *top view*, the reconstruction quality is acceptable trading off with the reconstruction speed. This large scale scene is composed of 249,505 faces, and the right screenshot shows the depth map, normal map, fusion map and zoom map. (Color figure online)

shown in Fig. 14d. The large-scale dense fusion algorithm has a more coarse voxel resolution [25], as shown in Fig. 14a with yellow rectangle, but there is voxel loss (shown in Fig. 14b). The layer volume algorithm [26] results in a larger number of voxel blocks and the overall reconstruction quality is reduced (as shown in Fig. 14c). Because the reconstructed quality of dense fusion algorithm is limited by the small spatial extent of the moving volume, some sensor data are not completely integrated beyond the range. The poor frame rate of the hierarchical fusion algorithm causes the input data to be skipped, which affects the accuracy of attitude estimation. Inaccurate surface fusion and drift phenomena occur.

In a variety of lighting conditions, multiple scenes are taken and the performance of the reconstruction system based on our algorithm is tested, as shown in Fig. 15. The reconstruction of dining room about 3 m high (16 m^2) takes about 5 min, the reconstruction time of 12 m long office room is 4 min. The average reconstruction rate of these large-scale scenes reaches 30 Hz, and the proposed algorithm performs well in reconstruction quality and rate.

The average time of reconstructing is 21.6 ms, the ICP posture is estimated 15 times, the time spent 8.0 ms, accounting for 37% of the total reconstruction time, the surface fusion is 4.6 ms (21%), the surface extraction took 4.8 ms (22%), Ms (20%).

7 Conclusions

In this paper, a feature-adaptive subdivision method is presented based on GPU hardware tessellation, and an reconstruction framework based on GPU hardware acceleration and depth Sensor is performed in real-time. The optimized algorithm establishes an unified Octree-primitive for the repeated feature regions including arbitrary vertices or crease edges without causing additional reconstruction time. It is particularly efficient for drawing the surfaces with a large number of repeated area. For example, the reconstruction of tiles ground with similar structure, hedgehogs in animated models, and plant in botany representations. Under the development of GPU hardware tessellation, real-time reconstruction for repeated feature regions appears to be more important. In the near future, the new method based on hardware tessellation will be widely used in mobile devices, and the 3D reconstruction technology based on our algorithm will be more prospective in the field of mobile graphics.

Acknowledgements This work is funded by National High-tech R&D Program (863 Program) (No.2014AA7031010B), Science and Technology Project of the thirteenth Five-Year Plan (JJZ[2016]345).

References

1. Bonaventura, X. (2011). Terrain and ocean rendering with hardware tessellation. In W. Engel (Ed.), *GPU Pro 2* (pp. 3–14). Natick: A K Peters Ltd.
2. Catmull, E., & Clark, J. (1978). Recursively generated B-spline surfaces on arbitrary topological meshes. *Computer-Aided Design, 10,* 350–355.
3. Jang, H., & Han, J. H. (2012). Feature-preserving displacement mapping with graphics processing unit (GPU) tessellation. *Computer Graphics Forum, 31,* 1880–1894.
4. He, L., Loop, C., & Schaefer, S. (2012). Improving the parameterization of approximate subdivision surfaces. *Computer Graphics Forum, 31,* 2127–2134.
5. Yun-cen, Huang, & Jie-qing, Feng. (2014). Mapping driving subdivision surface upon GPU rendering. *Journal of Computer-Aided Design & Computer Graphics, 26,* 1567–1575.
6. Kornuta, T., & Laszkowski, M. (2016). Perception subsystem for object recognition and pose estimation in RGB-D images. *Automation, 44*(10), 995–1003.
7. Loop, C., & Schaefer, S. (2008). Approximating Catmull–Clark subdivision surfaces with bicubic patches. *ACM Transactions on Graphics, 27,* 1–8.
8. Loop, C., Schaefer, S., & Ni, T. (2009). Approximating subdivision surfaces with gregory patches for hardware tessellation. *ACM Transactions on Graphics, 28*(5), 89–97.
9. Loop, C., & Schaefer, S. (2008). Approximating Catmull–Clark subdivision surfaces with bi-cubic patches. *ACM Transactions on Graphics, 27*(1), 1–11.
10. Marvie, J.-E., Buron, C., & Gautron, P. (2012). GPU shape grammars. *Computer Graphics Forum, 31,* 2087–2095.
11. Microsoft corporation. Direct 3D 12 Programming Guide (2016). https://msdn.microsoft.com/en-us/library/dn899121(v=vs.85).aspx.
12. Niessner, M., Loop, C., & Greiner, G. (2012). Efficient evaluation of semi-smooth creases in Catmull–Clark subdivision surfaces. *Eurographics shot pagers*. http://graphics.stanford.edu/∼niessner/niessner2012efficient.html.
13. Niessner, M., Loop, C., & Meyer, M. (2012). Feature-adaptive GPU rendering of Catmull–Clark subdivision surfaces. *ACM Transactions on Graphics, 31*(1), 1–11.
14. Nvidia. NVIDIA's Next Generation CUDA Compute Architecture. (2015). *Kepler GK110*. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-ArchitectureWhitepaper.pdf.
15. Pixar. OpenSubdiv. (2015). http://graphics.pixar.com/opensubdiv.
16. Pätzold, M., & Kolb, A. (2015). Grid-free out-of-core voxelization to sparse voxel octrees on GPU. In *Proceedings of the 7th conference on high-performance graphics* (pp. 95–103).
17. Schäfer, H., Raab, J., Keinert, B. (2015). Dynamic feature-adaptive subdivision. In *Proceedings of the 19th symposium on interactive 3d graphics and games* (pp. 31–38), New York: ACM.
18. Schäfer, H., Niessner, M., Keinert, B., Stamminger, M., Loop, C. (2014). *State of the art report on real-time rendering with hardware tessellation*. Eurographics 2014-State of the Art Reports (pp. 93–117).
19. Schäfer, H., Keinert, B., & Niessner, M. (2014). Local painting and deformation of meshes on the GPU. *Computer Graphics Forum, 52*(2 suppl 1), S82.
20. Stam, J. (1998). Exact evaluation of Catmull–Clark subdivision surfaces at arbitrary parameter values. In *SIGGRAPH'98 Proceedings of the 25th annual conference on computer graphics and interactive techniques* (pp. 395–404). New York: ACM.
21. Steinberger, M., Kenzel, M., Kainz, B. (2012). Scatter Alloc: Massively parallel dynamic memory allocation for the GPU. In *Innovative Parallel Computing* (pp. 1–10).
22. Vlachos, A., Peters, J., Boyd, C. (2001). Curved PN triangles. In *Proceedings of the 2001 symposium on interactive 3D graphics* (pp. 159–166).
23. Yazhen, Y., Rui, W., & Jin, H. (2016). Simplified and tessellated mesh for realtime high quality rendering. *Computer and Graphics, 54,* 135–144.
24. Yeo, Y.I., Bin, L., Peters, J. (2012). Efficient pixel-accurate rendering of animated curved surfaces. In *I3D'12 Proceedings of the ACM SIGGRAPH symposium on interactive 3D graphics and games* (pp. 165–174).
25. Yusov, E. (2012). Real-time deformable terrain rendering with DirectX 11. In W. Engel (Ed.), GPU Pro 3 (pp. 13–40), Natick, MA: A K Peters/CRC.

26. Whelan, T., Kaess, M., Johannsson, H., Fallon, M., Leonard, J. J., & Mcdonald, J. (2014). Real-time large-scale dense RGB-D SLAM with volumetric fusion. *International Journal of Robotics Research, 34*(4–5), 598–626.

27. Newcombe, R. A., Izadi, S., Hilliges, O., Molyneaux, D., & Kim, D. et al. (2011). KinectFusion: Real-time dense surface mapping and tracking. In *Proceedings of IEEE international symposium. Mixed and augmented reality* (pp. 127–136).