Computer Sciences Department

Real-time GPU-based Voxelization and Applications

Hsu-Huai Chang
Yu-Chi Lai

Technical Report #1688

March 2011

UNIVERSITY OF WISCONSIN MADISON

Noname manuscript No.
(will be inserted by the editor)

Real-time GPU-Based Voxelization and Applications

Hsu-Huai Chang · **Yu-Chi Lai**

Received: date / Accepted: date

Abstract This paper proposes a new real-time voxelization algorithm using newly available GPU functionalities. Our voxelization algorithm is efficient and able to real-time transform a highly complex surface-represented scene into a set of high-resolution voxels in **only one** GPU pass using the newly available geometry shader. The usage of 3D texture allows our algorithm to record the existence, color and normal information in a voxel directly without specific encoding and decoding mechanism. This allows us to adjust the voxel resolution according to hardware limitation and the need of applications without strenuous modifications to the encoding and decoding scheme. At the same time extra surfacial and volumetric information also allows us to render more realistic lighting effects. This paper demonstrates the usage of our voxelization results in rendering transparent shadow, transmittance and refraction. The results show that our algorithm can voxelize deformable models and render those complex lighting effects in real time without any preprocessing step.

Keywords voxelization · hardware voxelization · transparent shadow · volume ray-tracing

1 Introduction

Volumetric representation receives more and more research interest due to the increasing request in representing and processing 3D medical data (e.g. scanned results from computed tomography (CT) or magnetic resonance imaging (MRI)) and the requirement in newly available volumetric applications such as model simplification and repair [18], visibility determination [19], shadow rendering [15] and collision detection [7, 16, 8]. Because surface-boundary representation is the dominant force in Computer Graphics, a voxelization algorithm which transforms a surface-represented model to a volumetric representation is needed before applying volumetric applications and algorithms. Kaufman et al. [14] are the first to propose a voxelization algorithm. Generally, volumetric data stores properties of an object in a set of regular 3D grids. The smallest unit used to represent the 3D volumetric space is called a voxel which is similar to a pixel in 2D image space. Voxelization strategies can be classified based on how to describe the existence of a model as surface voxelization [22, 11, 12] which uses the existence of a voxel to describe the boundary and solid voxelization [21, 9, 4] which both describes the existence of the boundary and interior of the entire model. Another common classification is based on how the existence of a voxel is represented and can be described as binary [5, 2, 3] and non-binary [24, 21, 5, 10, 11, 20, 23] voxelization approaches. The latter can be further divided into filtered voxelization [24, 21], multivalued voxelization [5, 10], object identification voxelization [11] and distance transform [20, 23]. Because of extreme computation requirement in voxelization, a large number of existing algorithms are mainly used in a pre-processing step by assuming a static scene. A major drawback of this paradigm is its limited support for dynamic scenes and interactive applications. Because surface representations are generally more convenient, accurate and cost-effective in applying modeling, animation and interaction techniques, the voxelization process needs to be done on-the-fly after each change to the surface-represented model for volume rendering and other volume-related applications such as layered manufacturing and finite element analysis. In addition, real-time voxelization also allows the

Hsu-Huai Chang
National Taiwan University of Science and Technology, R.O.C. E-mail: td458193@hotmail.com

Yu-Chi Lai
National Taiwan University of Science and Technology, R.O.C. E-mail: cheeryuchi@gmail.com

intermixed usage of both geometric and volumetric representations to have scanned volume data working with the geometric models. For example, a surgical simulation system needs to combine medical scanning data sets, surgical tools and synthetic models (e.g. artificial organs) in one operational procedure in order to perform interactive surgical operations with both surface and volume representations. Thus, more and more research focuses on developing a real-time voxelization algorithm for interactive manipulation.

Thanks to the advance in graphics processing unit (GPU). With its powerful flexibility and programmability, real-time GPU voxelization becomes possible. Chen et al. [1] presented a slicing-based voxelization algorithm to slice the underlying model in the frame buffer by setting appropriate clipping planes in each pass and extracting each slice of the model to form volumetric representation. The method was extended and published later in [5]. Ignacio et al. [17] extended the same idea by setting 3D volume texture as the render target. Karabassi et al. [13] and Dong et al. [2] proposed two-pass algorithms by first projecting the triangle to a set of intermediate sheets and then reading back to generate the voxel set. These slicing-based algorithms suffer a serious limitation in computational efficiency because of the requirement in multiple GPU passes. As a result Eisemman et al. [3] presented an extension to the slicing-based algorithm by examining the intersection of each primitive with each voxel grid only once with a special encoding mechanism. They encoded the existence of a voxel into a 32-bit RGBA color and used multiple render target (MRT) to reduce the required number of GPU passes for finishing the slicing process. Forest et al. [6] overcame the limitation of possible holes existing in the boundary representation. Although the encoding algorithm can be really efficient, unfortunately the algorithm is limited to use triangles as the represented primitive and the performance is influenced by a dynamic update of the sorted triangles required for voxelizing deformable objects. The encoding methods [13, 2, 3] generally use projection to determine whether a voxel is inside a model or not and the projection induces errors and artifacts in voxelization results. Additionally these grid encoding methods have two other limitations: first, the encoding method must determine the resolution of voxelization and design the encoding mechanism accordingly and this makes the process to change resolution highly strenuous; second, the surfacial and volumetric properties of a model are hard to directly encoding into the representation. In addition the number of GPU passes for a high-resolution representation still has chance to be more than one. Thus, this paper seeks an voxelization algorithm which can overcome these limitations.

Our algorithm is also a GPU-based voxelization algorithm which slices the geometry models using the clipping plane algorithm [5]. Zhang et al. [26] prove that this choice can avoid the projection artifacts existing in encoding algorithms [13, 2, 3]. In addition, our algorithm is also designed to overcome several limitations existing in previous slicing-based voxelization methods. First is the inefficiency in voxelization due to the requirement of a large number of passes when slicing the model independently. Second is the modification complexity due to the requirement of a special grid encoding mechanism when slicing the model in one or several GPU passes. Third is the difficulties in encoding other surfacial and volumetric data with the existence in a voxel for grid encoding methods. Our algorithm achieves these by taking advantage of the newly available functionalities in GPU. These functionalities include the geometry shader and the ability to use 3D volume texture as the render target. The geometry shader is used to duplicate the triangles or quadrilaterals for each possibly intercepted slice in order to voxelize a scene in only one GPU pass. The usage of the geometry shader relieves the need of multiple passes in original slicing-based algorithms and enhances the voxelization efficiency. Then, the adjustable 3D volume texture is used to store the slicing result with other surfacial and volumetric information. Since the size of the 3D texture can be easily adjusted according to the need of application and the limitation of graphics hardware, this can ease the burden of changing encoding and decoding mechanism when adjusting the voxel resolution for a general encoding voxelization method. In addition our algorithm also has the ability to store extra surfacial and volumetric information such as color, transparency and normal with the volumetric data. This ability allows our algorithm to render more interesting effects such as rendering fantastic refraction effect of a glass artifact with different interior colors for different parts of that artifact. Since the voxelized results are stored in a 3D volume texture and ready for GPU applications, we demonstrated the strengths of our voxelization algorithm in rendering the transparent shadow, transmittance and refraction effects in real-time applications. Results show that our algorithm can gain improvement in voxelization efficiency and render all three different lighting effects in real time for a high-resolution voxelization process. The result is also visually realistic and nice.

The remainder of this paper is organized as follows: Section 2 presents the main steps of our voxelization algorithm. Section 3 demonstrates the usage of the voxelization results with extra voxel information in rendering highly realistic transparent shadow, transmittance, and refraction effects. Section 4 gives the performance measurement of voxelizing different surface-represented models and using the voxelization in different rendering applications. Finally, Section 5 describes the conclusion of our algorithm and a few extensions to the current work.

2 Voxelization

Our algorithm is a slicing-based method using newly available GPU functionalities. In addition to transform a surface-represented model into a volumetric representation, our voxelization algorithm also computes and stores certain surfacial and volumetric information such as normal, color and transmittance. Although our algorithm assumes that the surface-boundary model uses triangles as the fundamental primitives, it is intuitive to extend our slicing-based algorithm to handle quadrilaterals without any major modification. Thus, the voxelization steps are described as follows: First, the vertices of a triangle is sent to the vertex shader and their positions in camera coordinate are computed. Then, the geometry shader determines which slices along the z-axis direction have chance to intersect the rasterized triangle. The geometry shader is perfect for this task because it handles the assembly and construction of the triangle and can duplicate the triangle for rasterization in the pixel shader according to the need in the voxelization process. This duplication mechanism is the core of our algorithm and the detail will be given in Section 2.2. Finally, the pixel shader rasterizes the triangle by setting proper clipping planes to fill in boundary voxels. At the same time the shader also computes and stores extra surfacial and volumetric voxel information for the rasterized pixel. Extra solid voxelization step described in Section 2.3 may be applied to compute the interior voxels of the model if necessary. The voxelization results are transferred and saved to the disk for later usage or ready for other applications such as rendering transmittance, transparent shadow and refraction effects.

2.1 Voxel Storage

Before developing our voxelization algorithm we must decide how to store the volumetric representation of a model. A volumetric representation uses a uniform-sized voxel structure which is schematically similar to a 3D volume texture. Thus, using a texel in a 3D volume texture is the simplest way to store the volumetric data in a voxel. There are mainly two advantages of using a 3D volume texture for storing the voxelization result:

- The voxelization result can be used directly in later hardware application since the result is already in GPU memory.
- The newly available hardware feature of setting the 3D volume texture as the rendering target allows applications to use the 3D volume texture as the canvas for the geometric information. In addition, any slice in the 3D volume texture can be specified as the render target for our slicing-based voxelization algorithm.

Figure 1(a). shows an example of voxel index scheme for a simple 3D volume texture and the computation of voxel center location. Additionally, the memory size of a texel is adjustable in current version of graphics hardware. Therefore, the adjustable texel memory space also gives us the flexibility of adding extra surfacial and volumetric information such as transmittance, normal and color to generate more realistic lighting effect as described in Section 3.

2.2 Surface Boundary Voxelization

Detecting the voxels intersected with triangle

1 For each triangle, Tri
2 $z_0 = \mathbf{Z}(Tri.V0)$, $z_1 = \mathbf{Z}(Tri.V1)$,$z_2 = \mathbf{Z}(Tri.V2)$
3 $max_{slice} = \mathbf{max}(z_0, z_1, z_2)/thickness$
4 $min_{slice} = \mathbf{min}(z_0, z_1, z_2)/thickness$
5 For $i = min_{slice}$ to max_{slice}
6 $Plane_{near} = i * thickness$
7 $Plane_{far} = Plane_{near} + thickness$
8 Set $Plane_{near}$ and $Plane_{far}$ to projection matrix
9 If (**Intersect**(Tri))
10 Rasterize Tri into the slice

Fig. 2 This is the pseudo code for computing the boundary voxels of a triangle, $Tri.V$ denotes a vertex of a triangle, *thickness* is the voxel size which is a user specified value, $\mathbf{Z}()$ is a function to extract the depth value of a vertex after transforming the position of the vertex into the camera coordinate, **max**() / **min**() computes the maximum/minimum value among the set of input values and **Intersect**() is a function to test whether the triangle is valid after being culled by the clipping planes.

The pseudo code of slicing a triangle is shown in Fig. 2. Generally, a surface-represented triangle is stored as 3 vertices with their position, normal and other information. When vertices of a triangle are queued into the graphics pipeline, the position of a vertex is first transformed into the camera coordinate. Our voxelization algorithm computes which slices from the 3D volume texture have the chance to intersect the triangle. The range of slices which possibly intersect the triangle can be calculated with the depth of all three vertices using step 3 and 4 listed in Fig. 2. The geometry shader duplicates the triangle according to the number of slices in the possible range. At the end of the process the slicing algorithm sets up the far and near clipping plane according to the index of the destined slice in order to correctly compute the boundary voxels for the triangle. Fig. 1.(b) shows a simple example of the searching process.

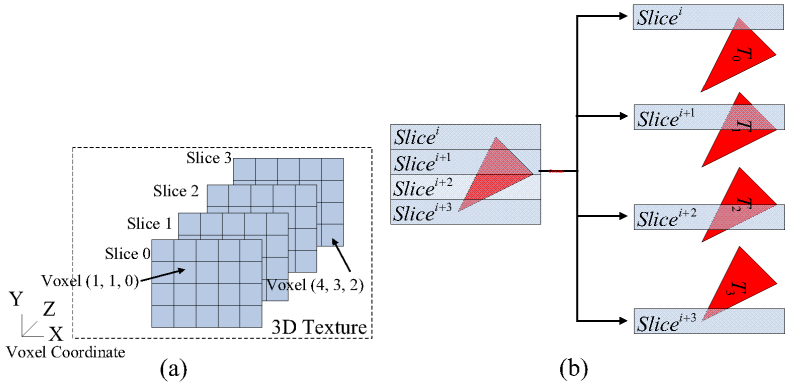


Fig. 1 (a). This shows a simple example of the voxel index scheme for a $5 \times 5 \times 4$ 3D volume texture. In this example the origin of the texture coordinate is set at the top left corner. A texel of the texture corresponds to a voxel of the volumetric representation. The center location of each voxel in camera coordinate is easily computed by one transformation of the texture coordinate. Therefore, the voxel indices of two voxels pointed by the arrows are $(1,1,0)$ and $(4,3,2)$ and their corresponding center positions in camera coordinate are $(0.75,0.75,0.25)$ and $(2.25,1.75,1.25)$ when the voxel physical size is $0.5 \times 0.5 \times 0.5$. Later the center location of voxel in the world coordinate can be computed using the camerat-to-world matrix in the graphics pipeline.
(b). In this example the triangle may intersect four possible slices during the voxelization process. Therefore, the voxelization algorithm must slice the triangle four times by setting proper clipping planes. Thus, the geometry shader must duplicate the triangle four times for the corresponding slicing process

Computing thickness

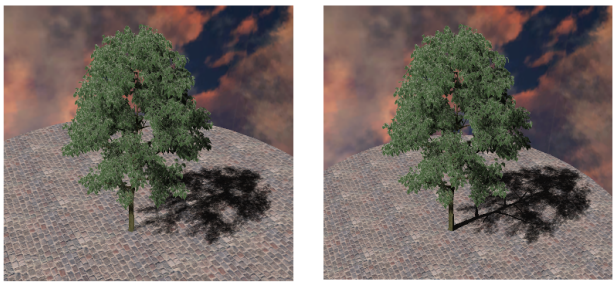
```
1   ThickSum = 0, SurfaceCount = 0, Inside = false
2   For each slice i
3     If( Voxel(x,y,i) is boundary voxel)
4       Inside = not(Inside)
5       SurfaceCount++
6     If(Inside)
7       ThickSum+ = thickness/cosθ
8   If (SurfaceCount >= 2&& SurfaceCount is even)
9     Thickness = ThickSum and marked the pixel as O.K.
10  Else 11       Marked the pixel as Questionable
```

Fig. 3 This is pseudo code used to compute the thickness of ray traversal through an object. $ThickSum$ is the accumulated thickness in the increasing z direction, $SurfaceCount$ counts the boundary voxel number, $Inside$ is a flag to indicate whether the current voxel is inside the mesh or not, (x,y) is the texture coordinate of a texel in the volume texture slice, **Voxel**() grabs the voxel information at index (x,y,i), **not** is the not operation, θ is the angle between the view and the slice index increasing direction and $thickness$ is the thickness of each slice.

2.3 Solid Voxelization

After the surface voxelization step described in the previous section, only boundary voxels are defined for the volumetric representation. However, the voxels existing inside the interior of the mesh is still not computed. Thus, our voxelization algorithm provides two strategies to fill in the interior voxels and their corresponding information.

- The first strategy immediately uses the boundary voxels from the previous step to compute the interior voxels and their information. The process applies XOR operation to all slices in sequential order as proposed in [5]. However, holes or discontinuities may exist in the boundary due to the slice resolution limit. Generally, an extra pass may be applied to fill those holes by using the neighborhood information.
- The second strategy is to fill in the interior voxels and their data on the fly when they are needed. This can be more efficient because most applications use only part of but not all the volumetric data. In addition they require to traverse through the voxels no matter whether the interior voxels are filled or not. The late-filling evaluation can be conducted based on the following observation: Since the surface representation is water-tight, the number of intersections with a model should be an even number; in other words when traversing along the increasing slice index direction, an entrance to the object and an exit is expected sequentially. Therefore, when applying the voxelization result to compute the lighting effects, the rendering camera is aligned to traverse the slices in the increasing z direction. When encountering a boundary voxel in one slice, the late-filing evaluation sets a flag and then marks all voxels below as interior voxels until seeing another boundary surface voxel. However, the hole problem may still happen due to the slice resolution limit. A view ray will be marked as questionable if there is an odd number of boundary voxel encountering during the traversal and later the questionable ray will be corrected using information from its neighborhood. Fig. 3 shows an example to compute the thickness of an object along the view ray traversal using implicit interior evaluation. In this example a low-pass filter is applied to



(a) (b)

Fig. 4 Rendering the shadow of a surface-represented tree model using ray tracing is too time consuming and thus transparent shadow map is a better choice. The left is a tree rendered with the same absorption coefficient for the entire tree. The right is a tree rendered with two different absorption coefficients for the trunk and leaves and the trunk shadow is darker even with a short passing length.

correct thickness of the questionable ray by using averaging thickness of its neighborhood.

All our applications described in Section 3 can use the implicit interior filling method to enhance the voxelization and rendering efficiency.

3 Applications

Volumetric representation has different applications such as visibility determination [19] and collision detection [7, 16, 8] in Computer Graphics. In this section three applications are used to demonstrate that our voxelization algorithm is efficient enough to real-time voxelize a complex surface-represented scene and the voxelization results with the computed surfacial and volumetric information can be used to generate realistic lighting effects including transparent shadow, transmittance and refraction.

3.1 Transparent Shadow Map

Shadow gives the sense of existence and is an important cue for human perception. Traditional shadow map is a simple method to render shadow when all objects in the scene are opaque. When there are transparent objects in the scene, the partial occlusion of a light ray passing through these transparent objects makes the situation more complex. The degree of occlusion is affected by the distance of a light ray passing through the object and the absorption along the traversal path. Eisemann et al. [3] used the voxelization result to estimate the passing distance for rendering transparent shadow but their method did not take the absorption into account. Therefore, our algorithm can compute the shadow according to the passing length and the absorption along the traversal path using the following steps:

1. The voxelization camera is set at the position of the light source and aligned with the light direction.
2. Our algorithm voxelizes the scene and computes and stores the absorption of each voxel.
3. During the rendering process, the voxel position, (x,y,s), of the first intersection point from the view is computed.
4. The amount of occlusion can be computed using the following equation:

$$\sum_{i=0}^{s} \alpha(x,y,i) \times E(x,y,i) \tag{1}$$

where $\alpha()$ describes the light absorption in this voxel and $E()$ is an occupation flag which 1 represents that the voxel is occupied by some object.

A tree rendered with its transparent shadow is shown in Fig. 4. The trunk is opaque and the leaves are partially transparent with low absorption. As a result the trunk shadow which has a short traversal path is darker than the leaf shadow which has a long traversal path.

3.2 Refraction

Refraction is the change in propagation direction of a light ray when it transport from one medium to another and the light propagation direction change can be described by the Snell's Law. But single refraction is not enough to describe the light transport through a transparent object because generally a light ray enters and exits an object in a pair and it is a multiple refraction phenomenon.

3.2.1 Two-surface refraction

Computing refraction

```
1   For each pixel
2     T1 = ref(I, Voxel(P1).N)
3     d = FindThickness(Voxel(P1), I)
4     P2 = P1 + d * T1
5     N2 = Voxel(P2).N
7     T2 = ref(T1, N2)
```

Fig. 5 **I** is the incident light direction, **Voxel**() is a function to locate the voxel with the location, P_1 is the position of the rendering point, d is the transmittance distance, P_2 is the second refraction position, $\mathbf{T}_1$ and $\mathbf{T}_2$ are the first and second refraction directions, **FindThickness**() estimates the thickness with position and ray direction and **ref**() calculate the refraction direction according to the Snell's Law.

Wyman et al. [25] proposed that multiple refraction may be simplified to a two-surface-refraction effect: one happens when light enters the object and the other happens when

light exits. The first refraction can use the normal of the intersection point and the incident direction to compute the first refraction direction, $\mathbf{T}_1$. If the traversal distance, d, between the first and the second refraction point can be estimated, the second refraction position can be estimated with the following equation:

$$P_2 = P_1 + d\mathbf{T}_1 \tag{2}$$

where P_1 and P_2 are the first and second refraction position, and $\mathbf{T}_1$ is the refraction direction after the first refraction. Wyman et al. [25] proposed an image-based method to estimate d without considering the first refraction direction. We realized that our voxelization result can find a better estimate of d and our two-surface refraction algorithm can estimate $\mathbf{T}_2$ in the following steps:

1. The voxelization camera is aligned with the rendering camera.
2. Our algorithm voxelizes the scene and computes and stores the surface normals and refraction indices.
3. After voxelization, $\mathbf{T}_1$ is computed with the view direction, position, surface normal and refraction index at the first scene intersection point of the view ray.
4. The voxelization result is used to estimate the transmittance distance, d, with a similar manner described in Fig 3.
5. P_2 can be computed using Eqn. 2 and projected into the voxel space to extract the surface normal, $\mathbf{N}_2$ and refraction index.
6. The second refraction direction, $\mathbf{T}_2$, can be computed with $\mathbf{T}_1$, $\mathbf{N}_2$ and refraction index.

Fig. 5 lists the pseudo code for the two-surface-refraction algorithm.

3.2.2 Multiple-surface refraction

The two-surface-refraction method cannot render all transmittance lighting effects when light hits a transparent object in a scene. In addition it also has some limit in the allowable models and transmittance. Thus, a multiple-surface-refraction algorithm is proposed to simulate the refractions and reflections inside a scene. The same voxelization process described in the two-surface-refraction method is used. When rendering the scene, the view initiates a view ray passing through the center of a pixel. Then, the ray is propagated inside the voxel space and every time when the ray hits a boundary voxel, the ray is refracted according to the Snell's law as shown in Fig. 6. In order to properly locate the boundary voxel for refraction, the propagating distance must be set properly to prevent missing the boundary voxel during the traversal procedure and wasting efforts in extra propagation. Our implementation chooses the physical distance to propagate through a voxel as the propagation step distance. Then,

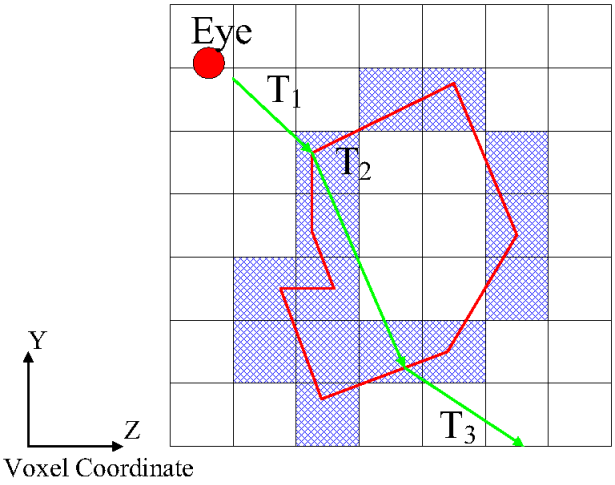


Fig. 6 This schematic diagram shows a view ray passing through the object inside the voxel space. The view path is represented by the green line segments. $\mathbf{T}_1$ is the direction of the initial view ray, $\mathbf{T}_2$ is the first refracted direction and $\mathbf{T}_3$ is the second refracted direction. The blue squares denote the boundary voxels and the red dot denotes the camera center which is the starting point of the view ray.

the position where the next refraction event happens can be computed as follows:

$$P_i = P_{i-1} + thickness \times T_{voxel}(\mathbf{T}_i)/cos\theta \tag{3}$$

where P_{i-1} is the current position, *thickness* represents the physical size of the voxel, $\mathbf{T}_{i-1}$ is the current ray propagation direction, $T_{voxel}()$ is a function to transform the ray into the voxel coordinate for locating the voxel which records the normal and transmittance information and θ is the angle between the ray and the dominant component axis of the ray. The next step computes the refracted view ray direction according to the following equation:

$$\mathbf{T}_i = \begin{cases} ref(\mathbf{T}_{i-1}, N_v(P_i), T_v(P_i)) & if(E(P_i)) = 1 \\ \mathbf{T}_{i-1} & otherwise \end{cases} \tag{4}$$

where $ref()$ computes the new ray direction according to the Snell's law, $E(P_i)$ is a flag which indicates whether the voxel at P_i is a boundary voxel or not, $N_v(P_i)$ extracts the normal stored in the voxel at P_i and $T_v(P_i)$ extracts the refraction index stored in the voxel at P_i. The process continues to find the intersection and refracted ray direction until the ray hit the boundary of the volume.

However, when applying the multiple-surface-refraction algorithm described in previous paragraph, several situations may happen to induce serious artifacts into the rendering result. The one happens most frequently is multiple boundary voxels along the traversal path which is shown in Fig 7(a). This happens because the boundary voxels will occupy a certain volume in the voxel space. Therefore, even when the ray passes through the voxel without really intersecting the surface, our algorithm will misjudge the hitting of the

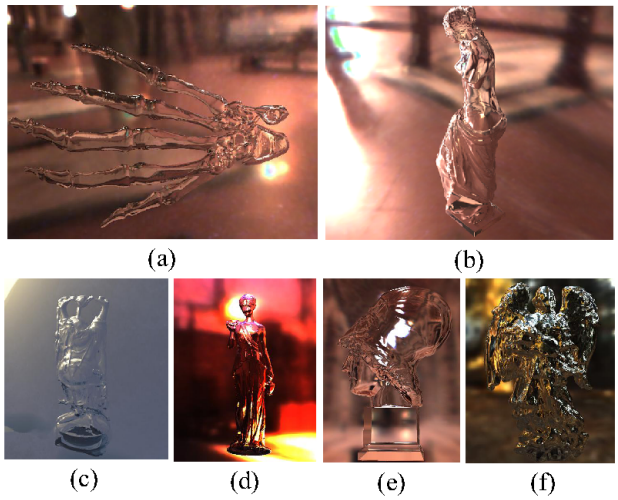


Fig. 8 These are the result using multiple-refraction rendering with our voxelization algorithm. The refraction index is set to be 1.14 for the models used in (a), (b), (c) and (d) and 1.3 in (e) and (f).

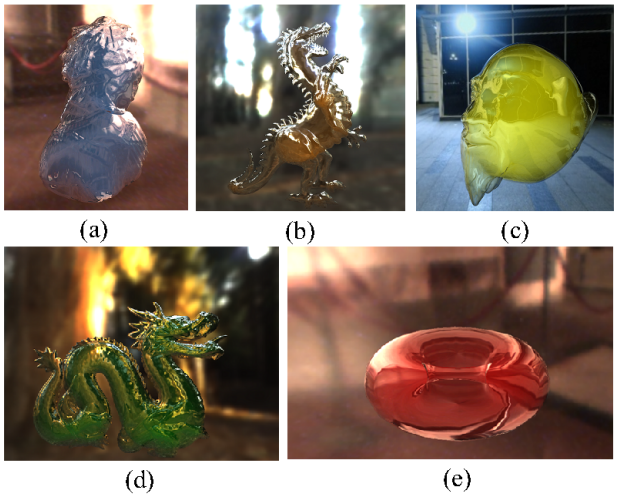


Fig. 9 These demonstrate the transmittance rendering of different models. When the traversal length of the view ray through the model becomes larger, the rendering of the ray becomes opaque i.e. the amount of blending with background is less. For example, the tier part in (c) is dark green and blends with no background component because the view ray cannot see through the model. All these examples use a homogeneous material and thus the thickness of the view path is the only affecting factor.

boundary voxel by the ray and initiate the refraction computation. This leads to unwanted artifacts. Our algorithm uses the surface locality information to relieve this issue. We observed that when the angle between the normals of surfaces where the consecutive refractions happens is small, the possibility of misjudge is high. Thus, our algorithm uses a threshold to determine whether the refraction mechanism initiates or not and this can reduce a large amount of artifact in this type.

When tracing the view ray, there is another condition as shown in Fig 7(b). which may cause the failure of refraction computation. This missing boundary voxel problem happens because the voxelization resolution limit induces a hole on the water-tight boundary surface and the ray may pass through the corner and edge of the boundary voxels and miss the hit when tracing the ray through the voxel space. This problem is similar to the hole problem mentioned in Section 2.3. Generally a low pass filter should be able to reduce this problem. In addition, the filter technique can also reduce the multiple intersecting voxel issues described previously. Our algorithm proposed another relief to this missing voxel issue based on the observation that a missing voxel issue is much harder to handle properly than a multiple intersected voxel issue. Thus, when slicing the scenes, our voxelization algorithm extends the clipping region of the slice to make the extent of a voxel overlap with others' to increase the chance of multiple intersected voxel situations and reduce the chance of missing voxel situations. Then the angle threshold discussed in the previous paragraph can be used to get a good rendering result.

The multiple-surface-refraction algorithm can simulate the multiple refraction effects to generate realistic refraction in real time. It is generally more efficient than traditional ray-tracing. This is because a fixed number of voxels can be used to represent a complex surface model. Thus, the traversal cost can be limited in a controllable amount and so is the efficiency of rendering. When comparing the shark animation rendered with our multiple-surface-refraction algorithm with the animation rendered with the two-surface-refraction algorithm, our result has less aliasing artifact. Both animations are provided in the supplement material with our submitted paper. This is because our algorithm can get a more precise simulation to the real condition of light refracted inside the transparent object than the two-surface-refraction method and Wymman's image-based refraction method. Our method also overcomes the limitation of convex models and possible transmittance values existing in the two-surface-refraction and image-based refraction methods. Fig. 8 shows the rendering results using this multiple-surface-refraction method.

3.3 Transmittance

When light passes through a medium, the amount of energy passing through will decrease and this phenomenon can be described by transmittance. The simplest method [3] to estimate transmittance uses a parameter, transparency, which depends on the traversal length of the light ray through the transparent object. Then, the transmittance is used to determine the amount of transparent blending between the object and the background. However, the method [3] is limited to an object with homogeneous material. Our voxelization re-

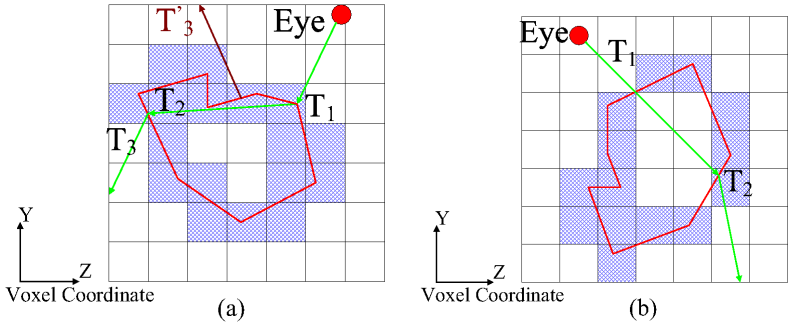


Fig. 7 (a). This demonstrates the problematic scheme during the ray traversal. The green line segments demonstrate the correct view path and the brown line segments show the possibly problematic view path. The reason of this is because $\mathbf{T}_2$ intersects with multiple boundary voxels and the closest one is not the desired one along the traversal path.
(b). Because of the hole in the voxelization result, the first surface information will be neglected and thus the refraction path cannot be currently tracing which is similar to the problemable ray marking.

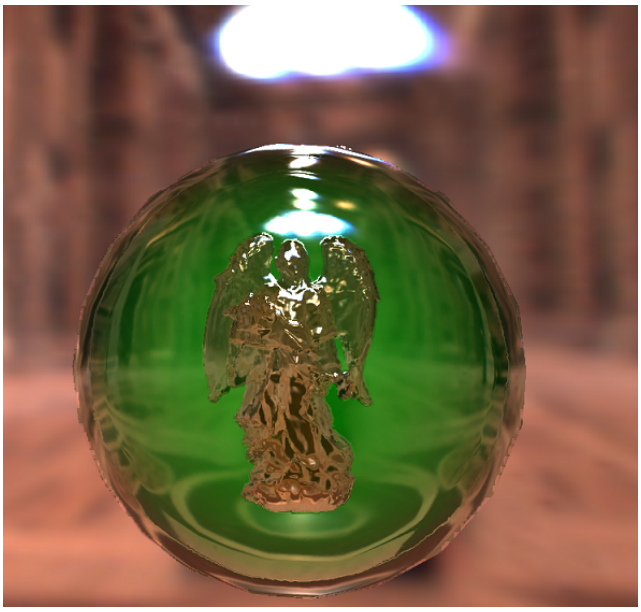


Fig. 10 This demonstrates the strength of recording surface normal and transmittance. Our algorithm can render an object with multiple different materials.

sult contains the surface normal information, the transmittance coefficient and refraction index. This allows our rendering method to compute transmittance with higher precision by both considering the length and the different transparent attenuation of the traversal voxels along the path. The traversal distance between refraction points can be estimated using $\| \ thickness \times T_{voxel}(\mathbf{T}_i)/cos\theta \ \|$ which is a byproduct of Eqn. 3. Then the transparent attenuation can be computed by the following equation:

$$transparency = \prod_{i=0}^{N} e^{sigma(d_i)} \tag{5}$$

where N is the total number of refraction points along the traversal path and $\sigma()$ calculates the transparent attenuation of the object [3].

Fig. 9 shows the results rendered with refraction and transmittance effects with a uniform transmittance coefficient and refraction index for the entire model. As the discussion in Section 3.2, the multiple-surface refraction can trace all possible refraction and compute the transparent attenuation along the traversal path and thus, it is very easy for our application to render an object that contains parts with different transparent materials as shown in Fig. 10 while other algorithms such as [3, 25] can only render the refraction of the ball without considering the refraction effect of the angel.

4 Results

All the results in this paper are rendered and measured using a computer with ATI HD5850, Intel Core 2 duo E6750 and 2 GB main memory. Our voxelization algorithm is implemented with DirectX 11 but the same program is also compatible to DirectX 10. The vertex, geometry and pixel shaders are written using HLSL 4.0. The 3D volume texture is implemented with the format of 2D texture array which is a set of 2D textures with the same format in each pixel and the same resolution for each texture. The array provides the required properties to totally support the need of our algorithm and gives our algorithm more freedom in setting the format of each pixel during the implementation process. And a 32-bit RGBA floating point format is chosen to record the voxel information in our current implementation.

Additionally, each graphics hardware device has a different limitation in the allowable voxelization resolution and the limitation depends on the available GPU memory space. For example a resolution of $256 \times 256 \times 256$ with 32-bit information per voxel requires a memory space of $128MB$ to store the data. According to the graphics card used in the test, our voxelization algorithm are tested on voxelizing different models with various triangle counts under three different resolution settings which are $128 \times 128 \times 128$, $256 \times$

Name	Alg.	# Tris	128^3	256^3	512^3
Torus	Ours	800	1.66	3.73	10.91
	Grid		2.67	7.29	641.03
Venusm	Ours	43357	3.19	5.04	12.41
	Grid		12.41	73.02	746.27
Horse	Ours	96966	4.28	7.65	15.09
	Grid		15.09	156.99	925.93
Hand	Ours	654666	11.5	20.00	27.68
	Grid		27.68	632.91	2500
Dragon	Ours	871414	16.54	23.85	31.06
	Grid		31.06	1282.05	3448.28
Happy	Ours	1087716	16.36	24.71	31.58
	Grid		31.58	1724.14	4347.82

Table 1 This shows the time needed to voxelize models with different triangle counts with different resolution settings using our voxelization algorithm marked with **Ours** and the voxelization algorithm proposed by Ignacio [17] et al. marked with **Grid**. The performance is measure in *ms*.

256×256 and $512 \times 512 \times 512$. The results are shown in Table 1. In addition the time required to voxelize the same set of models under these three resolutions using the algorithm proposed by Ignacio et al. [17] is also shown in Table 1. The main reason to compare against their algorithm is due to being the derivative of slicing-based algorithm and the same usage of 3D texture to store the voxelization result. However, their algorithm processes the primitives in a model once per slice. Thus, the amount of computation increases with the increase in the voxelization resolution as shown in Table 1. Obviously, our algorithm can get much better efficiency when voxelizing models in higher resolution. However, because their algorithm can process the slices in sequential order to set up proper stencil buffer in voxelization process, their algorithm can get interior information with higher precision for relieving aliasing artifact when using the results.

Table 1 demonstrates that the main factor of efficiency is the triangle count of the model and the voxelization resolution. In addition our algorithm is also affected by the size of the model and the viewing angle of the voxelization process. When analyzing the contribution of these other factors, we found that the difference between the best and worst performance is roughly $10ms$. In addition to the cost factors from the voxelization process there are other cost factors when applying the voxelization result to render the lighting effects proposed in Section 3. The main factor which affects the efficiency of rendering is the amount of voxel accessed through the process. Because the rendering process has to run through a set of slices, when the number of processing slices increases, the amount of texture access will also increase. It is even worse that the cost to access the texture data in the GPU memory space is much higher than the cost to access CPU memory . Thus, general practice is to reduce the number of slices with the increase in the slice resolution to reduce the number of processed slices with acceptable rendering quality.

5 Conclusion

New general-purpose GPU algorithms are developed to take advantage of efficient and highly-parallel computation abilities of GPU in order to enhance the efficiency of problem solving. In this paper a real-time algorithm is proposed to voxelize the surface-represented scene based on two newly available functionalities, the geometry shader and 3D volume texture. The geometry shader duplicates the triangle during the voxelization process to reduce the GPU rendering pass down to one time. Although the price paid is the extra triangles drawn per voxelized triangle, the voxelization efficiency is still improved. The usage of 3D volume texture as the render target give us the flexibility of adjusting the voxel resolution according to the hardware capability and the requirement of applications without the strenuous modification in the encoding and decoding process required by the grid encoding voxelization algorithms. In addition 3D volume texture also allows applications to compute and store more information in a voxel. The information includes the normal, color and transparency to generate more realistic lighting effects. We have demonstrated this ability in rendering more realistic transparent shadow, transparency, and refraction effects in real-time applications. However, there are still several future research directions to take advantage of our efficient and flexible voxelization algorithm. First, since different surface and volume information can be recorded in each voxel, it would be interesting to design a user interface to specify and edit the material of each individual voxel in order to give glass artists the ability to simulate the fantastic color of a glass artifact. In addition the voxelization result should be used to incorporate with global illumination algorithms such as ray-marching and photon mapping to generate other realistic lighting effects such as caustics. In our application the maximum resolution is set to $512 \times 512 \times 512$. However, there may be some situation required even higher resolution and the required resolution may be over the limit of the hardware capability. There are several ways to reduce the memory requirement when increasing the resolution:

- The usage of less bits to record the voxel information can reduce the memory requirement but this will reduce the precision of information in each voxel.
- A multiple pass method can be used to resolve the memory limitation. However, the trade-off is the efficiency of the program because each model has to be drawn multiple times and data transfer between CPU and GPU memory increases.

In addition a proper recording scheme should be sought to optimizing the processing time and stored memory size. Fi-

nally, the results presented in this paper demonstrate that our voxelization algorithm is efficient and flexible for real-time voxelization and applications.

References

1. Chen H, Fang S (1998) Fast voxelization of three-dimensional synthetic objects. J Graph Tools 3(4):33–45
2. Dong Z, Chen W, Bao H, Zhang H, Peng Q (2004) Real-time voxelization for complex polygonal models. In: PG '04: Proceedings of the Computer Graphics and Applications, 12th Pacific Conference, IEEE Computer Society, Washington, DC, USA, pp 43–50
3. Eisemann E, Décoret X (2006) Fast scene voxelization and applications. In: ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, ACM SIGGRAPH, pp 71–78
4. Eisemann E, Décoret X (2008) Single-pass gpu solid voxelization and applications. In: GI '08: Proceedings of Graphics Interface 2008, Canadian Information Processing Society, ACM International Conference Proceeding Series, vol 322, pp 73–80
5. Fang S, Chen H (2000) Hardware accelerated voxelization. Computers and Graphics 24:200–0
6. Forest V, Barthe L, Paulin M (2009) Real-time hierarchical binary-scene voxelization. Journal of Graphics Tools 29(2):21–34
7. Gibson SFF (1995) Beyond volume rendering: Visualization, haptic exploration, and physical modeling of element-based objects. In: In Proc. Eurographics workshop on Visualization in Scientific Computing, pp 10–24
8. Harada T, Koshizuka S (2006) Real-time cloth simulation interacting with deforming high-resolution models. In: SIGGRAPH '06: ACM SIGGRAPH 2006 Research posters, ACM, New York, NY, USA, p 129
9. Haumont D, Warz?e N (2002) Complete polygonal scene voxelization
10. Heidelberger B, Teschner M, Gross M (2003) Real-time volumetric intersections of deforming objects. In: Proceedings of Vision, Modeling, and Visualization 2003, pp 461–468
11. Huang J, Yagel R, Filippov V, Kurzion Y (1998) An accurate method for voxelizing polygon meshes. In: Proceedings of the 1998 IEEE symposium on Volume visualization, ACM, New York, NY, USA, VVS '98, pp 119–126
12. Ix FD, Kaufman A (2000) Incremental triangle voxelization
13. Karabassi EA, Papaioannou G, Theoharis T (1999) A fast depth-buffer-based voxelization algorithm. journal of graphics, gpu, and game tools 4(4):5–10
14. Kaufman A, Shimony E (1987) 3d scan-conversion algorithms for voxel-based graphics. In: Proceedings of the 1986 workshop on Interactive 3D graphics, ACM, New York, NY, USA, I3D '86, pp 45–75
15. Kim TY, Neumann U (2001) Opacity shadow maps. In: Proceedings of the 12th Eurographics Workshop on Rendering Techniques, Springer-Verlag, London, UK, pp 177–182
16. Li W, Fan Z, adn A Kaufman XW (2004) GPU Gems 2, Ch. 47: Simulation with Complex Bounaries. Addison-Wesley
17. Llamas I (2007) Real-time voxelization of triangle meshes on the gpu. In: SIGGRAPH '07: ACM SIGGRAPH 2007 sketches, ACM, New York, NY, USA, p 18
18. Nooruddin FS, Turk G (2003) Simplification and repair of polygonal models using volumetric techniques. IEEE Transactions on Visualization and Computer Graphics 9(2):191–205
19. Schaufler G, Dorsey J, Decoret X, Sillion F (2000) Conservative volumetric visibility with occluder fusion. In: SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, pp 229–238
20. Sigg C, Peikert R, Gross M (2003) Signed distance transform using graphics hardware. Visualization Conference, IEEE 0:12
21. Sramek M, Kaufman A (1999) Alias-free voxelization of geometric objects. Visualization and Computer Graphics, IEEE Transactions on 5(3):251 –267
22. Stolte N (1997) Robust voxelization of surfaces. Tech. rep., State University of New York at Stony Brook
23. Varadhan G, Krishnan S, Kim YJ, Diggavi S, Manocha D (2003) Efficient max-norm distance computation and reliable voxelization. In: SGP '03: Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, pp 116–126
24. Wang SW, Kaufman AE (1993) Volume sampled voxelization of geometric primitives. In: VIS '93: Proceedings of the 4th conference on Visualization '93, IEEE Computer Society, Washington, DC, USA, pp 78–84
25. Wyman C (2005) An approximate image-space approach for interactive refraction. ACM Trans Graph 24(3):1050–1053
26. Zhang L, Chen W, Ebert DS, Peng Q (2007) Conservative voxelization. Vis Comput 23:783–792