MASTER THESIS: GPU VOXELIZATION

Student: Carlos Tripiana Montes

Advisor: Carlos Antonio Andújar Gran

September, 8th 2009

Course: Master in Computing

LSI Department

Polytechnic University of Catalonia

To the best of my life: Anna; to my family and friends. They are the reason of who I am.

Contents

Introduction

During the last few decades, several algorithms have been proposed to convert a 3D model into a voxel representation. This process involves identifying which voxels are intersected by the surface of the model.

Voxelizations are used in a variety of fields. Two main examples are assigning volumetric attributes and distance field computation; with applications in crowd simulation, fast path finding, boolean operations and modeling continually varying heterogeneous materials.

This process is done traditionally in CPU, using geometrical computations to perform the intersection tests. However, in the last few years, several algorithms have been proposed to to dynamically calculate a voxel-based representation of a scene using programmable graphics hardware (GPU), allowing the real-time creation of voxelizations even for complex and dynamic scenes containing more than one million polygons. GPU-based voxelization algorithms exploit the rasterization process of current graphic cards, which is a highly optimized task, taking advantage of hardware parallelism.

In contrast, GPU-based voxelization algorithms have a number of limitations. The most remarkable one is the lack of accuracy: all GPU-based algorithms proposed so far (even the so called conservative ones) provide only an approximate solution to the problem, failing to identify all intersected voxels, or identifying as intersected voxels that do not intersect.

These problems are related mainly with the rasterization process. This process involves determining what pixels will be drawn in the screen, and what color have. To do this, for each primitive it is necessary to know the pixels that are intersecting the primitive and what is the distance between the camera and the primitive for each pixel. But the rasterization stage in GPU does not capture all the pixels that intersect the primitive, only those with their center inside the primitive. Also, if a perspective camera is used, the normalized distance –depth– between the camera and the primitive for each pixel it is non-uniform distributed.

The main contributions of this thesis are:

- A comparison and evaluation of state-of-the-art GPU-based voxelization algorithms, in terms of running time and accuracy, with respect to an exact, CPU-based, reference algorithm.
- A new GPU-based algorithm which, unlike competing approaches, computes the voxelization in an exact way.
- A numerical comparison of our new algorithm with competing approaches, using a variety of test models and grid resolutions. Our experiments show that our approach is much faster than the CPU-based algorithm. The robustness of our algorithm makes it suitable for those applications requiring a high level of accuracy.

Chapter 1

State of the art

1.1 Problem definition

Given a triangulated model, we want to identify which voxels of a voxel grid are intersected by the boundary of this model. There are other branch of implemented voxelizations, in which not only the boundary is detected, also the interior of the model.

Often these voxels are cubes. But it is not a restriction, there are other presented techniques in which the voxel grid is the view frustum, and voxels are prisms.

There are different kind of voxelizations depending on the rasterization behavior. Approximate rasterization is the standard way of rasterizing fragments in GPU. It means only those fragments whose center lies inside the projection of the primitive are identified. Conservative rasterization (Hasselgren *et al.* , 2005) involves a dilation operation over the primitive. This is done in GPU to ensure that in the rasterization stage all the intersected fragments have its center inside the dilated primitive. However, this can produce spurious fragments, non-intersected pixels. *Exact voxelization* detects only those voxels that we need.

1.2 OpenGL rasterization in depth

We start by discussing some key aspects of the OpenGL 3D API which directly impact GPU-based voxelization algorithms.

1.2.1 Depth computation

Figure 1.1 shows the process of computing the depth value for a given vertex.

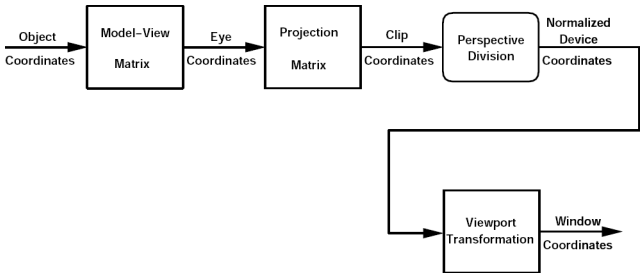

Figure 1.1: OpenGL transformation pipeline. A 3D point is transformed as if the origin of the coordinate system will be placed in the eye point. After that the point is projected onto the near plane. Coordinates are normalized to perform a fast frustum clipping after the perspective division. Finally the window coordinates will be computed.

OpenGL has some parameters to perform this computation. Let F be the field of view, we use the following notation:

- $orig_x$ (*int*): Window coordinate system origin for X axis.
- $orig_y$ (*int*): Window coordinate system origin for Y axis.

The pixel (0, 0) is the bottom left window pixel.

- $size_x$ (*sizei*): Window size -in pixels- for X axis -width-.
- $size_y$ (*sizei*): Window size -in pixels- for Y axis -height-.
- Depth range: Window depth range and behavior (Shreiner *et al.* , 2005, p. 141). Values must lie inside $[0, 1]$:
  - d_{Near} (*clampd*): Value that indicates "the most closest to the camera".
  - d_{Far} (*clampd*): Value that indicates "the most farthest to the camera".

The camera parameters are:

- Extrinsic:
  - OBS (*double*): The camera's location.
  - VRP (*double*): Reference point which this camera is pointing.
  - VUV (*double*): A vector that indicates the camera's rotation over Z axis.
- Intrinsic:
  - Perspective:
    - FOV (*double*): The camera's vertical field of view.
    - AR (*double*): Aspect ratio.
    - Clipping planes: Inside this planes (perpendicular to the view direction) the geometry is rendered but out of there.
      - z_{Near} (*double*): Distance between OBS and the near plane.
      - z_{Far} (*double*): Distance between OBS and the far plane.

    These parameters are being used to compute the view *frustum* which is defined as a truncated pyramid. The bottom is on the far plane and the top is the OBS point but it is truncated by the near plane.
  - Perspective/Orthogonal:
    - l, r, b, t (*double*): Distances between the view's center and the left, right, bottom and top clipping planes. This distances are being measured on the far plane.
    - Clipping planes: Inside this planes (perpendicular to the view direction) the geometry is rendered but out of there.
      - z_{Near} (*double*): Distance between $OBS$and the near plane.
      - z_{Far} (*double*): Distance between $OBS$and the far plane.

    An orthogonal camera is a degenerated perspective case with the camera placed at the infinity.

Given the previous parameters and given $q_o = \begin{pmatrix} x_o & y_o & z_o & w_o \end{pmatrix} \Rightarrow q_o^T = \begin{pmatrix} x_o \\ y_o \\ z_o \\ w_o \end{pmatrix}$ a point in object coordinates, this is the coordinate transformation process (Segal & Akeley, 2006, pp. 40–46):

- $q_e^T = \begin{pmatrix} x_e \\ y_e \\ z_e \\ w_e \end{pmatrix} = Mq_o^T$ in camera coordinates (C.C.).
  - M is the viewing matrix.
- $q_c^T = \begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = Pq_e^T$ in clip coordinates (projected and normalized in range $[-1, 1]$. Used to perform the frustum clipping).
  - $P = P_{proj} = \begin{pmatrix} \frac{2z_{Near}}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2z_{Near}}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-z_{Far}-z_{Near}}{z_{Far}-z_{Near}} & \frac{-2z_{Far}z_{Near}}{z_{Far}-z_{Near}} \\ 0 & 0 & -1 & 0 \end{pmatrix}$ (Shreiner *et al.* , 2005, p. 755).
  - $P = P_{ortho} = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & \frac{-r-l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & \frac{-t-b}{t-b} \\ 0 & 0 & \frac{-2}{z_{Far}-z_{Near}} & \frac{-z_{Far}-z_{Near}}{z_{Far}-z_{Near}} \\ 0 & 0 & 0 & 1 \end{pmatrix}$ (Shreiner *et al.* , 2005, p. 755).

$$q_{c_{proj}}^T = \begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = P_{proj}q_e^T = \begin{pmatrix} \frac{2z_{Near}x_e}{r-l} + \frac{r+l}{r-l}z_e \\ \frac{2z_{Near}y_e}{t-b} + \frac{t+b}{t-b}z_e \\ \frac{-z_{Far}-z_{Near}}{z_{Far}-z_{Near}}z_e + \frac{-2z_{Far}z_{Near}w_e}{z_{Far}-z_{Near}} \\ -z_e \end{pmatrix}.$$

$$q_{c_{ortho}}^T = \begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = P_{ortho}q_e^T = \begin{pmatrix} \frac{2x_e}{r-l} + \frac{-r-l}{r-l}w_e \\ \frac{2y_e}{t-b} + \frac{-t-b}{t-b}w_e \\ \frac{-2z_e}{z_{Far}-z_{Near}} + \frac{-z_{Far}-z_{Near}}{z_{Far}-z_{Near}}w_e \\ w_e \end{pmatrix}.$$

- $q_d^T = \begin{pmatrix} x_d \\ y_d \\ z_d \end{pmatrix} = \begin{pmatrix} \frac{x_c}{w_c} \\ \frac{y_c}{w_c} \\ \frac{z_c}{w_c} \end{pmatrix}$ in normalized display coordinates.

$$q_{d_{proj}}^T = \begin{pmatrix} x_d \\ y_d \\ z_d \end{pmatrix} = \begin{pmatrix} \frac{-2z_{Near}x_e}{(r-l)z_e} - \frac{r+l}{r-l} \\ \frac{-2z_{Near}y_e}{(t-b)z_e} - \frac{t+b}{t-b} \\ \frac{z_{Far}+z_{Near}}{z_{Far}-z_{Near}} + \frac{2z_{Far}z_{Near}w_e}{(z_{Far}-z_{Near})z_e} \end{pmatrix}.$$

$$q_{d_{ortho}}^T = \begin{pmatrix} x_d \\ y_d \\ z_d \end{pmatrix} = \begin{pmatrix} \frac{2x_e}{(r-l)w_e} - \frac{r+l}{r-l} \\ \frac{2y_e}{(t-b)w_e} - \frac{t+b}{t-b} \\ \frac{-2z_e}{(z_{Far}-z_{Near})w_e} - \frac{z_{Far}+z_{Near}}{z_{Far}-z_{Near}} \end{pmatrix}.$$

We can see this step is the responsible of providing more resolution to the nearest z values and less to the farthest (only for perspective cameras –see below–).

- $q_w^T = \begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} = \begin{pmatrix} \frac{size_x}{2}x_d + o_x \\ \frac{size_y}{2}y_d + o_y \\ \frac{d_{Far}-d_{Near}}{2}z_d + \frac{d_{Far}+d_{Near}}{2} \end{pmatrix}$.

o_x, o_y represents the "center" point of the viewport which is computed as:

  - $o_x = orig_x + \frac{size_x}{2}$.
  - $o_y = orig_y + \frac{size_y}{2}$.

$$q_{w_{proj}}^T = \begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} =$$
$$\begin{pmatrix} \frac{size_x}{2}\left(\frac{-2z_{Near}x_e}{(r-l)z_e} - \frac{r+l}{r-l}\right) + orig_x + \frac{size_x}{2} \\ \frac{size_y}{2}\left(\frac{-2z_{Near}y_e}{(t-b)z_e} - \frac{t+b}{t-b}\right) + orig_y + \frac{size_y}{2} \\ \frac{d_{Far}-d_{Near}}{2}\left(\frac{z_{Far}+z_{Near}}{z_{Far}-z_{Near}} + \frac{2z_{Far}z_{Near}w_e}{(z_{Far}-z_{Near})z_e}\right) + \frac{d_{Far}+d_{Near}}{2} \end{pmatrix}.$$

$$q_{w_{ortho}}^T = \begin{pmatrix} x_w \\ y_w \\ z_w \end{pmatrix} =$$
$$\begin{pmatrix} \frac{2x_e}{(r-l)w_e} - \frac{r+l}{r-l} \\ \frac{2y_e}{(t-b)w_e} - \frac{t+b}{t-b} \\ \frac{d_{Far}-d_{Near}}{2}\left(\frac{-2z_e}{(z_{Far}-z_{Near})w_e} - \frac{z_{Far}+z_{Near}}{z_{Far}-z_{Near}}\right) + \frac{d_{Far}+d_{Near}}{2} \end{pmatrix}.$$

The general depth value in window coordinates is, following the OpenGL specification:

$$z_{w_{proj}} = \frac{d_{Far}-d_{Near}}{2}\left(\frac{z_{Far}+z_{Near}}{z_{Far}-z_{Near}} + \frac{2z_{Far}z_{Near}w_e}{(z_{Far}-z_{Near})z_e}\right) + \frac{d_{Far}+d_{Near}}{2}$$

$$z_{w_{ortho}} = \frac{d_{Far}-d_{Near}}{2}\left(\frac{-2z_e}{(z_{Far}-z_{Near})w_e} - \frac{z_{Far}+z_{Near}}{z_{Far}-z_{Near}}\right) + \frac{d_{Far}+d_{Near}}{2}$$

Assuming $w_e = 1$, these are the final equations:

$$z_{w_{proj}} = \frac{d_{Far}+d_{Near}}{2} + \frac{d_{Far}-d_{Near}}{2}\left(\frac{z_{Far}+z_{Near}}{z_{Far}-z_{Near}} + \frac{2z_{Far}z_{Near}}{(z_{Far}-z_{Near})\,z_e}\right) \tag{1.1}$$

$$z_{w_{ortho}} = \frac{d_{Far}+d_{Near}}{2} - \frac{d_{Far}-d_{Near}}{2}\frac{z_{Far}+z_{Near}+2z_e}{z_{Far}-z_{Near}} \tag{1.2}$$

For a fixed OpenGL state –suppose a common one such as $d_{Near} = 0$, $d_{Far} = 1$, $z_{Near} = 0.5$, and $z_{Far} = 1$–, all variables of these equations were fixed unless z_e –commonly it takes negative values–. This variable in the equations makes the following distribution for $z_{w_{proj}}$ and $z_{w_{ortho}}$ for values of $z_e \in [-0.5, -1]$:

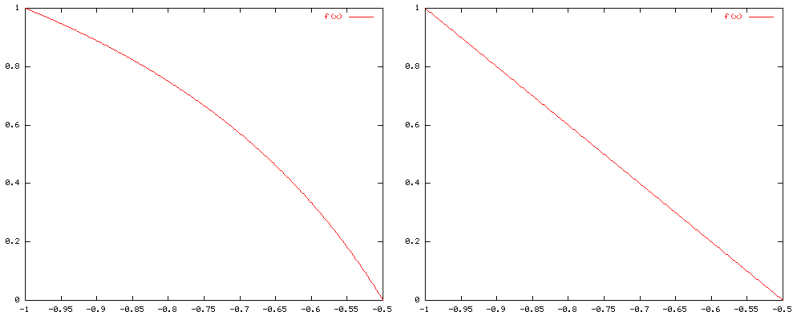

Figure 1.2: Distributions. Left projective, right orthogonal.

As we mentioned above the z resolution has linear distribution for orthogonal cameras.

The OpenGL specification assumes that the z_w is represented as an integer with as many bits as the depth buffer has (let N be this number of bits). Each value in the representation k maps the range $\frac{k}{2^N-1}$, $k \in \{0, 1, \ldots, 2^N-1\}$.

Finally, the conclusion is that, for each window coordinate (x, y) the depth buffer value is computed as:

$$z_{depth\,buffer} = \left(2^N - 1\right) z_w \tag{1.3}$$

Please, note that the window is defined as an integer number of pixels but window coordinates in OpenGL are *double*. This is also important to multisampling techniques and other questions. But it is clear that in this description the x_w and the y_w are *double*.

1.2.2 Polygon rasterization

Rasterization can be defined as the process by which a primitive is converted to a two-dimensional image (Segal & Akeley, 2006, pp. 108–110). Each point of this image contains information such as color and depth. Thus, rasterizing a primitive consists of two parts:

- Determine which squares of an integer grid in window coordinates are occupied by the primitive.
- Assign a depth value and one or more color values to each such square.

The results of this process are passed on to the next stage (per-fragment operations), which uses the information to update the appropriate locations in the framebuffer.

The color values assigned to a fragment are initially determined by the rasterization operations and modified by either the execution of the texturing, color sum, and fog operations, or by a fragment shader. The final depth value is initially determined by the rasterization operations and may be modified or replaced by a fragment shader.

A grid square along with its parameters of assigned colors, z (depth), fog coordinate, and texture coordinates is called a *fragment*. A fragment is located by its lower left corner, which lies on integer grid coordinates. Rasterization operations also refer to a fragment's *center*, which is offset by $(\frac{1}{2}, \frac{1}{2})$ from its lower left corner (and so lies on half-integer coordinates). Grid squares need not actually be square in the OpenGL. Rasterization rules are not affected by the actual aspect ratio of the grid squares. Display of non-square grids, however, will cause rasterized points and line segments to appear fatter in one direction than the other. We assume that fragments are square, since it simplifies antialiasing and texturing. *The way OpenGL rasterizes polygons is very important for GPU voxelization since this process determines which fragments will be generated and hence which voxels are detected.*

Filtering primitives: A polygon results from a polygon **Begin**/**End** object, a triangle resulting from a triangle strip, triangle fan, or series of separate triangles, or a quadrilateral arising from a quadrilateral strip, series of separate quadrilaterals, or a **Rect** command.

The first step of polygon rasterization is to determine if the polygon is back facing or front facing. This determination is made by examining the sign of the area computed by equation

$$A = \frac{1}{2} \sum_{i=0}^{n-1} x_w^i y_w^{i \oplus 1} - x_w^{i \oplus 1} y_w^i \tag{1.4}$$

where x_w^i and y_w^i are the x and y window coordinates of the ith vertex of the n-vertex polygon (vertices are numbered starting at zero for purposes of this computation) and $i \oplus 1 = (i+1) \, mod \, n$. The interpretation of the sign of this value is controlled with

```
void FrontFace( enum dir );
```

Setting *dir* to `CCW` (corresponding to counter-clockwise orientation of the projected polygon in window coordinates) indicates that if $a \leq 0$, then the color of each vertex of the polygon becomes the back color computed for that vertex while if $a > 0$, then the front color is selected. If *dir* is `CW`, then a is replaced by $-a$ in the above inequalities. This state is initially set to `CCW`.

This determination is used in conjunction with the **CullFace** enable bit and mode value to decide whether or not a particular polygon is rasterized. The **CullFace** mode is set by calling

```
void CullFace( enum mode );
```

mode is a symbolic constant: one of `FRONT`, `BACK` or `FRONT_AND_BACK`. Culling is enabled or disabled with **Enable** or **Disable** using the symbolic constant `CULL_FACE`. Front facing polygons are rasterized if either culling is disabled or the **CullFace** mode is `BACK` while back facing polygons are rasterized only if either culling is disabled or the **CullFace** mode is `FRONT`. The initial setting of the **CullFace** mode is `BACK`. Initially, culling is disabled.

If we are interested in rasterize all the primitives when performing a voxelization, culling must be disabled.

Rasterization Process: The rule for determining which fragments are produced by polygon rasterization is called *point sampling*. The two-dimensional projection obtained by taking the x and y window coordinates of the polygon's vertices is formed. Fragment centers that lie inside of this polygon are produced by rasterization. Special treatment is given to a fragment whose center lies on a polygon boundary edge. In such a case we require that if two polygons lie on either side of a common edge (with identical endpoints) on which a fragment center lies, then exactly one of the polygons results in the production of the fragment during rasterization. As for the data associated with each fragment produced by rasterizing a polygon, we begin by specifying how these values are produced for fragments in a triangle. Define barycentric coordinates for a tri-

angle. Barycentric coordinates are a set of three numbers, a, b, and c, each in the range [0, 1], with $a + b + c = 1$ (absolute barycentric coordinates). These coordinates uniquely specify any point $p = (x, y)$ within the triangle or on the triangle's boundary as

$$p = ap_a + bp_b + cp_c$$

where p_a, p_b, and p_c are the vertices of the triangle. a, b, and c can be found as

$$a = \frac{A(pp_bp_c)}{A(p_ap_bp_c)},\ b = \frac{A(pp_ap_c)}{A(p_ap_bp_c)},\ c = \frac{A(pp_ap_b)}{A(p_ap_bp_c)}$$

where $A(lmn)$ denotes the area in window coordinates of the triangle with vertices l, m, and n. Denote an associated datum at p_a, p_b, or p_c as f_a, f_b, or f_c, respectively. Then the value f of a datum at a fragment produced by rasterizing a triangle is given by

$$f = \frac{\frac{af_a}{w_{c_a}} + \frac{bf_b}{w_{c_b}} + \frac{cf_c}{w_{c_c}}}{\frac{a}{w_{c_a}} + \frac{b}{w_{c_b}} + \frac{c}{w_{c_c}}}$$

where w_{c_a}, w_{c_b} and w_{c_c} are the clip w coordinates of p_a, p_b, and p_c, respectively. a, b, and c are the barycentric coordinates of the fragment for which the data are produced. a, b, and c must correspond precisely to the exact coordinates of the center of the fragment.

The value for this datum is:

$$f_{proj} = \frac{\frac{af_a}{-z_{e_a}} + \frac{bf_b}{-z_{e_b}} + \frac{cf_c}{-z_{e_c}}}{\frac{a}{-z_{e_a}} + \frac{b}{-z_{e_b}} + \frac{c}{-z_{e_c}}} = \frac{\frac{af_a}{z_{e_a}} + \frac{bf_b}{z_{e_b}} + \frac{cf_c}{z_{e_c}}}{\frac{a}{z_{e_a}} + \frac{b}{z_{e_b}} + \frac{c}{z_{e_c}}} \tag{1.5}$$

$$f_{ortho} = \frac{\frac{af_a}{w_{e_a}} + \frac{bf_b}{w_{e_b}} + \frac{cf_c}{w_{e_c}}}{\frac{a}{w_{e_a}} + \frac{b}{w_{e_b}} + \frac{c}{w_{e_c}}}$$

Since p_a, p_b, and p_c are points, $w_{e_a} = 1$, $w_{e_b} = 1$ and $w_{e_c} = 1$, so for orthogonal case then,

$$f_{ortho} = \frac{af_a + bf_b + cf_c}{a + b + c} = af_a + bf_b + cf_c \tag{1.6}$$

Once again we can see the efficiency benefits of a orthogonal camera. Since it has a linear z distribution is not necessary to revert this situation. It is good to avoid precision errors and make interpolation process fast.

However, depth values for polygons must be interpolated by

$$z_w = az_{w_a} + bz_{w_b} + cz_{w_c} \tag{1.7}$$

where z_{w_a}, z_{w_b}, and z_{w_c} are the depth values of p_a, p_b, and p_c, respectively. The z values which OpenGL needs are in window coordinates: is not necessary to revert its distribution. Vertices has z_w computed by the transformation process, however for fragments it must be interpolated. Remember that now these z values are integers but a, b, and c.

For a polygon with more than three edges, we require only that a convex combination of the values of the datum at the polygon's vertices can be used to obtain the value assigned to each fragment produced by the rasterization algorithm. That is, it must be the case that at every fragment

$$f = \sum_{i=1}^{n} a_i f_i \tag{1.8}$$

where n is the number of vertices in the polygon, f_i is the value of the f at vertex i; for each i $0 \leq a_i \leq 1$ and $\sum_{i=1}^{n} a_i = 1$. The values of the a_i may differ from fragment to fragment, but at vertex i, $a_j = 0$, $j \neq i$ and $a_i = 1$.

1.2.3 Antialiasing by multisampling

In Segal & Akeley, 2006, pp. 92–95 multisampling is defined as a mechanism to antialias all OpenGL primitives: points, lines, polygons, bitmaps, and images. The technique is to sample all primitives multiple times at each pixel, modifying the default OpenGL behavior: to sample the fragment's center. The color sample values are resolved to a single, displayable color each time a pixel is updated, so the antialiasing appears to be automatic at the application level. Because each sample includes color, depth, and stencil information, the color (including texture operation), depth, and stencil functions perform equivalently to the single-sample mode.

An additional buffer, called the multisample buffer, is added to the framebuffer. Pixel sample values, including color, depth, and stencil values, are stored in this buffer. Samples contain separate color values for each fragment color. When the framebuffer includes a multisample buffer, it does not include depth or stencil buffers, even if the multisample buffer does not store depth or stencil values. Color buffers (left, right, front, back, and aux) do coexist with the multisample buffer, however.

Multisample antialiasing is most valuable for rendering polygons, because it requires no sorting for hidden surface elimination, and it correctly handles adjacent polygons, object silhouettes, and even intersecting polygons. *If only points or lines are being rendered, the "smooth" antialiasing mechanism provided by the base GL may result in a higher quality image.* This mechanism is designed to allow multisample and smooth antialiasing techniques to be alternated during

the rendering of a single scene.

If the value of SAMPLE_BUFFERS is one, the rasterization of all primitives is changed, and is referred to as multisample rasterization. Otherwise, primitive rasterization is referred to as single-sample rasterization. The value of SAMPLE_BUFFERS is queried by calling GetIntegerv with *pname* set to SAMPLE_BUFFERS.

During multisample rendering the contents of a pixel fragment are changed in two ways. First, each fragment includes a coverage value with SAMPLES bits. *The value of SAMPLES is an implementation-dependent constant*, and is queried by calling GetIntegerv with *pname* set to SAMPLES.

Second, each fragment includes SAMPLES depth values, color values, and sets of texture coordinates, instead of the single depth value, color value, and set of texture coordinates that is maintained in single-sample rendering mode. An implementation may choose to assign the same color value and the same set of texture coordinates to more than one sample. The location for evaluating the color value and the set of texture coordinates can be anywhere within the pixel including the fragment center or any of the sample locations. The color value and the set of texture coordinates need not be evaluated at the same location. Each pixel fragment thus consists of integer x and y grid coordinates, SAMPLES color and depth values, SAMPLES sets of texture coordinates, and a coverage value with a maximum of SAMPLES bits.

Multisample rasterization is enabled or disabled by calling Enable or Disable with the symbolic constant MULTISAMPLE.

If MULTISAMPLE is disabled, multisample rasterization of all primitives is equivalent to single-sample (fragment-center) rasterization, except that the fragment coverage value is set to full coverage. The color and depth values and the sets of texture coordinates may all be set to the values that would have been assigned by single-sample rasterization, or they may be assigned as described below for multisample rasterization.

If MULTISAMPLE is enabled, multisample rasterization of all primitives differs substantially from single-sample rasterization. It is understood that each pixel in the framebuffer has SAMPLES locations associated with it. These locations are exact positions, rather than regions or areas, and each is referred to as a sample point. The sample points associated with a pixel may be located inside or outside of the unit square that is considered to bound the pixel. Furthermore, the relative locations of sample points may be identical for each pixel in the framebuffer, or they may differ.

If the sample locations differ per pixel, they should be aligned to window, not screen, boundaries. Otherwise rendering results will be window-position specific. The invariance requirement is relaxed for all multisample rasterization, because the sample locations may be a function of pixel location. Also, it is not possible

to query the actual sample locations of a pixel.

We can conclude that for some voxelization techniques this improves its accuracy, but it has an overhead associated. *And this multisampling cannot be used to ensure an exact or conservative voxelization.*

1.3 Characterization of voxelization algorithms

There are multiple ways of classifying voxelization algorithms, Table 1.1 is a possible classification:

RESPECT TO	**OPTIONS**
Processing unit	CPU
	GPU
Voxel data	binary
	non-binary
Identified voxels	boundary voxels
	boundary + in voxels
View dependence	view-independent
	view-dependent
Render passes	number of render passes needed
Accuracy	approximate
	conservative
	exact

Table 1.1: Classification of voxelization.

Voxelization algorithms are classified in two main groups, CPU-based and GPU-based. In our implemented work, an efficient CPU-based voxelization (Akenine-Möller, 2001) is used as the reference one.

On the one hand, voxelization algorithms can be characterized by the information associated to voxels. A binary voxelization is when there are only information is about presence (voxel is present/not present). Non-binary voxelizations are when voxels store additional data.

In the other hand, boundary voxelization algorithms are those which only voxels intersected by the surface of the model are detected. There are voxelization algorithms capable to detect voxels lying completely inside the model, performing what is known by a solid voxelization.

GPU-based voxelization algorithms are classified in two branches, namely: view-dependent and view-independent. View-dependent algorithms get different results depending on how the camera is placed, while view-independent voxelization algorithms get the same result because the camera is used as is

needed: the user do not have the possibility to place the camera. We remark also how many render passes are needed to complete the voxelization process for GPU-based algorithms.

The accuracy of a voxelization algorithm has three levels: approximate, conservative and *exact*. When we talk about approximate voxelization algorithms we assume these methods miss voxels or add spurious ones. When we talk about conservative voxelization we can ensure these processes do not miss any voxel while adding spurious (we are talking about over-conservative voxelizations). There are semi-conservative algorithms: some GPU-based algorithms have a different process to overestimate each axis.

An *exact voxelization* algorithm ensures theoretically perfect results. This is restricted by floating point errors. Please, note that all kind of voxelization algorithm has some precision errors.

1.4 Implemented work

Table 1.2 summarizes the features of the voxelization algorithms we analyzed (described in detail below). We have included the most representative GPU-based voxelization algorithms, and one CPU-based exact voxelization algorithm to be used as reference. We use the criteria introduced in Table 1.1.

TECH.	UNIT	DATA	IDENT.	VIEW	PASSES	ACCUR.
Akenine-Möller, 2001	CPU	binary	boundary	independent	0	exact
Eisemann & Décoret, 2006	GPU	binary	boundary	dependent	1	approx.
Eisemann & Décoret, 2008	GPU	binary	boundary+in	dependent	1	approx.
Dong *et al.* , 2004a	GPU	binary	boundary	independent	3	approx.
Hasselgren *et al.* , 2005	GPU	(rasterization method)				conserv.

Table 1.2: Classification of voxelizations.

1.4.1 Fast 3D triangle-box overlap testing (Akenine-Möller, 2001)

1.4.1.1 Main idea

This work presents an algorithm for applying the separate axis theorem to perform a single triangle-box intersection test.

As pointed-out by Akenine-Möller, it is possible to use this technique to perform a CPU voxelization. Given a model represented as a triangle mesh,

and a voxel grid, the algorithm allows to compute which voxels are intersected by the surface of the model.

1.4.1.2 The method

In a common sense we develop an efficient way to apply the triangle-box test. Given a model and a voxel grid declared in a common world coordinate system, the algorithm performs the following steps:

1. Get the X, Y and Z of the three vertices.
2. Compute what are the MIN_X, MIN_Y and MIN_Z, and MAX_X, MAX_Y and MAX_Z, of the AABB of the triangle.
3. Compute the intersection between this AABB and the voxel grid to identify those voxels potentially intersecting the triangle.
4. Perform a triangle-box test with the triangle and each of these voxels. Mark the voxel as occupied if an intersection is detected.

The above process is repeated for each triangle of the model. The triangle-box test is executed only if the voxel is not marked as occupied, so as to avoid doing redundant tests. This algorithm gets an exact computation of a voxelization (of course, it is exact without considering small precision –floating point– errors).

The triangle-box test itself is based on the Separate Axis Theorem (SAT), which can be stated as follows:

Theorem. *Two convex polyhedra, A and B, are disjoint if they can be separated along either an axis parallel to a normal of a face of either A or B, or along an axis formed from the cross product of an edge from A with and edge from B.*

We focus on testing an axis-aligned voxel (AAVOX), defined by a center c, and a vector of half lengths, h, against a triangle $\triangle u0u1u2$. To simplify the tests, we first move the triangle so that the box is centered around the origin, i.e., $v_i = u_i - c$, $i \in \{0, 1, 2\}$.

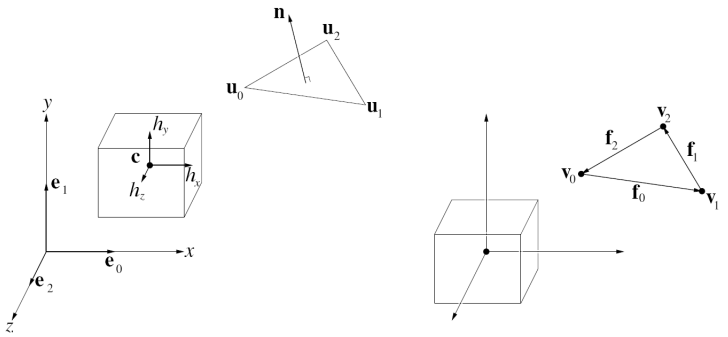

Figure 1.3: Notation used for the triangle-box overlap test. To the left the initial position of the box and the triangle are shown, while at the right, the box and the triangle have been translated so that the box center is at the origin.

Based on SAT, we test the following 13 axes:

1. [3 tests] $e0 = (1,0,0)$, $e1 = (0,1,0)$, $e2 = (0,0,1)$ (the normals of the AAVOX). Test the AAVOX against the minimal AABB around the triangle.

2. [1 test] n, the normal of $\triangle$. We use a fast plane/AABB overlap test, which only tests the two diagonal vertices, whose direction is most closely aligned to the normal of the triangle.

3. [9 tests] $a_{ij} = e_i \times f_j$, $i,j \in \{0,1,2\}$, where $f_0 = v_1 - v_0$, $f_1 = v_2 - v_1$, and $f_2 = v_0 - v_2$. These tests are very similar and we will only show the derivation of the case where$i = 0$ and $j = 0$. $a_{00} = e_0 \times f_0 = (0, -f_{0z}, f_{0y})$ so, now we need to project the triangle vertices onto a_{00} (hereafter called a):

   $p_0 = a \cdot v_0 = (0, -f_{0z}, f_{0y}) \cdot v_0 = v_{0z}v_{1y} - v_{0y}v_{1z}$

   $p_1 = a \cdot v_1 = (0, -f_{0z}, f_{0y}) \cdot v_1 = v_{0z}v_{1y} - v_{0y}v_{1z} = p_0$

   $p_2 = a \cdot v_2 = (0, -f_{0z}, f_{0y}) \cdot v_2 = (v_{1y} - v_{0y})v_{2z} - (v_{1z} - v_{0z})v_{2y}$

   Normally, we would have had to find $min(p0, p1, p2)$ and $max(p0, p1, p2)$, but fortunately $p_0 = p_1$, which simplify the computations. Now we only need to find $min(p_0, p_2)$ and $max(p_0, p_2)$, which is faster.

   After the projection of the triangle onto a, we need to project the box onto a as well. We compute a "radius", called r, of the box projected on a as

   $r = h_{xj} \left|a_{xj}\right| + h_{yj} \left|a_{yj}\right| + h_{zj} \left|a_{zj}\right| = h_{yj} \left|a_{yj}\right| + h_{zj} \left|a_{zj}\right|$

where the last step comes from that $a_x = 0$ for this particular axis. Then this axis test becomes:

`if` $(min(p_0, p_2) > r$ `or` $max(p_0, p_2) < -r)$ `return false;`

Now, if all these 13 tests pass, then the triangle overlaps the box.

1.4.1.3 Implementation issues

A robustness issue appears when the normal of the triangle is computed; $n = f_0 \times f_1$. If the triangle has an area close to zero, then the normal computation is not robust, and the above code does not solve that problem. However, in most applications thin long triangles are best avoided.

1.4.2 Fast Scene Voxelization and Applications (Eisemann & Décoret, 2006)

1.4.2.1 Main idea

The main idea of this algorithm is to achieve a voxelization based on a slicing method of a scene with one rendering pass.

1.4.2.2 The method

The algorithm takes as input a polygonal scene. Now they define a grid by placing a camera in the scene and adjusting its view frustum to enclose the area to be voxelized. The camera must be placed at any position outside the zone of interest. Then, they associate a viewport to the camera with (w, h) dimensions which indicate the resolution of the grid in the X and Y directions, so the voxelization is constructed over the framebuffer. A pixel (x, y) represents a column in the grid using the color buffer. Each cell within this column is encoded via the RGBA value of the pixel considering this value as a vector of 32 bits, each one representing a cell in the column.

Now the corresponding image represents a $w \times h \times 32$ grid with one bit of information per cell. We will use that bit to indicate whether a primitive passes through a cell or not. The union for all columns of voxels corresponding to a given bit defines a slice. Consequently, the image/texture encoding the grid is called a slicemap.

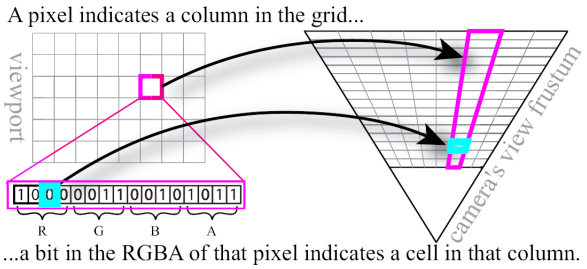

Figure 1.4: Encoding of a grid in the viewport of a camera. For clarity 4 bits per channel in the color buffer (16 slices) are assumed.

Procedure: Given a model, we will rasterize it, and for each fragment they determine which slice is intersected by the underlying primitive. Since the fragment's position is implicit and hence does not need to be computed. The first step is to clear the framebuffer, create the transformation matrices according to the desired camera position and frustum, and set the viewport resolution properly. After that, the render process can be started.

Rasterizing the primitive will produce a single fragment for each of the columns intersected and the depth d of that fragment indicates in which slice it falls. This depth value –in window coordinates– are in the range $[0, 1]$. The authors say that the distribution of this range is not uniform in world coordinates, however, using this depth for slices would put too much resolution close to the near plane and not enough close to the far plane. We must apply to this affirmation some corrections: As OpenGL specification describes this is only valid if the used camera is perspective. In contrast, if the camera is orthographic the distribution is linear (Segal & Akeley, 2006). In the rest of the description, because the above assumption, we consider that the authors used a perspective camera.

The distance between the camera's COP and 3D position of the vertex, computed by applying the modelview matrix –the z_e value– is passed to the fragment shader as texture coordinates. This decision is chosen in order to apply the on-surface interpolation –this process occurs in eye coordinates, which is linear space– to this distance which is also in eye coordinates, and get the distance for each fragment, not only for vertices. This process creates the correct z value in $[-z_{near}, z_{Far}]$. They used to map this value to linear $[0, 1]$ range the following function

$$z' = \frac{z + z_{Near}}{z_{Near} + z_{Far}} \tag{1.9}$$

This normalized distance is used to perform a texture lookup in a 1D texture that gives the 32 bits mask corresponding to the slice in which z' falls. The resulting texture will be referred to as the cellmask texture. Its format is RGBA with 8 bits per channel. Note that it is independent of the actual voxel grid's position and is computed only once. The convention for the cellmask texture implies that the values in the mask are between 2^0 for the nearest one and 2^{31} for the farthest cell.

With the color/bitmask they apply the logical operation *OR* over the color buffer to set inside it the position of the voxel. Of course initially the fragment color is set to $(0, 0, 0, 0)$ –black–.

1.4.2.3 Implementation issues

This method relies in using the color buffer to perform a binary voxelization, so it is limited to 32 voxels' depth. Since we want to perform high resolution voxelizations we decided to use a multiple render targets (MRT) technique, based on the framebuffer object (FBO) extension. Each FBO has a fixed number of color buffers (the MRTs). This number of MRTs depends on the GPU model.

For each render pass the program can write to the MRTs, and it is necessary to perform one pass for each needed FBO. How many resources we will need depends only in the voxel resolution.

It is also important to remark that the resolution is bounded as well by the maximum texture resolution. More resolution can be used performing a texture patching method. The process basically is to split the width and height in more textures. This can by easily mixed with the previous technique to increase the grid resolution, using as many MRTs and FBOs as the GPU allows.

As we explained in Paragraph 1.2.2, if a polygon is almost parallel to the view direction then probably its projection wont capture any fragment's center. This implies that primitive wont be rasterized. Also, if an intersecting fragment does not have the (X, Y) coordinates of its center inside the primitive's projection wont be detected at fragment stage.

1.4.3 Single-Pass GPU Solid Voxelization for Real-Time Applications (Eisemann & Décoret, 2008)

1.4.3.1 Main idea

Based on the previous work (Eisemann & Décoret, 2006) this article has the same underlying idea. The only difference becomes from the fact that this article talks about how to perform a solid voxelization, instead of boundary one. As a consequence, the algorithm requires a solid, watertight model as

input.

1.4.3.2 The method

To perform this voxelization there are now some changes. The first one is the color buffer logic operation that performs OpenGL when applies a color over the buffer. Now we will use a XOR operation. The second modification is the values of the texture mask. Now, each textel has enabled all the bits which are corresponding to the previous voxels ($mask\,[0] = 0$, $mask\,[1] = 1$, $mask\,[2] = 3$, ...).

Now when we apply the mask for a given voxel depth, we mark as 1 the whole column finishing in the previous voxel. Of course the XOR operation modifies this as the next picture shows:

Figure 1.5: Solid Voxelization for a column in the slicemap. To simplify the illustration, only one framebuffer with two bit color channels is shown. Left: The scene, consisting of two watertight objects, is voxelized in the column along the view direction. 1-4): During rendering, fragments can arrive in an arbitrary order. For each fragment, a bitmask (upper row) is computed in the shader which indicates all voxels that lie in front of the current fragment. This mask is accumulated in the framebuffer (bottom, initialized at zero) using a XOR operation. Once the rendering is complete (4), the framebuffer contains a center sampled solid voxelization in a grid shifted by half a voxel.

Due to the way rasterization is performed on current cards and the choice of the bitmask, the voxelization samples centers of a voxel grid shifted by half a voxel along the z-axis. There is no imprecision introduced due to the XOR operator. The shift comes from the fact that they choose the bitmask based on the voxel the fragment falls into. Thus, the separations are naturally at the boundary between two column voxels. The offset can be counteracted though by a adding half a voxel to the fragments distance, thus virtually shifting the column.

1.4.3.3 Implementation issues

To this technique the same issues as in Eisemann & Décoret, 2006 appear. But it has other problems.

This voxelization only detects those voxels with their center inside the model (not only by its (X, Y) coordinates, Z as well). Some of those detected voxels are completely inside the model but others are intersecting the boundary. Due to the XOR blending, some detected boundary voxels –see Figure 1.5– are lost. This imposes a second pass with a boundary voxelization technique to obtain a better voxelization, but we didn't implemented.

As we mentioned above, the original models must be watertight. This ensures an even number of intersections over each column. If we are voxelizing a non-watertight model, unexpected results may be obtained.

1.4.4 Real-time Voxelization for Complex Polygonal Models (Dong *et al.* , 2004a)

1.4.4.1 Main idea

The objective is to avoid projection problems in the rasterization stage performing three passes, each of those viewing the scene in front of one of each axis direction. The coordinate system will be placed at the center of the voxel grid, with their axis parallel to the grid edges. This method uses an orthogonal camera, which implies linear distribution of depth.

1.4.4.2 The method

The algorithm takes as input a triangulated geometric model. Now we suppose that this model is inside its axis-aligned bounding box and it describes a discretization of the space that lies inside itself. These discretization has the form of a 3D regular grid. Let be B a bounding box, it is number of voxel is $V = WHD$ where W is the width –over x axis–, H the height –over y axis–, and D the depth –over z axis– of B.

The bounding box is split following the depth axis in slabs –suppose it is z axis–. Each slab –which is the same as a slice– has the above width an height, an its depth range is related to the number of bits one texture element (*textel*) has. Let be C the maximum value of the depth range of each slab. The representation does not fix how many data is stored for each voxel, but they develop their explanation using binary surface voxelization. Moreover they use standard 32 bits –8 per color channel– per texel 2D textures, so these parametrization falls to be $C = 32$.

Now we can use one slab to store WHC voxels. In order to fill the bounding box completely the number of used slabs is $N = \lceil \frac{D}{C} \rceil$. All the slabs merged into one same texture called *"sheet buffer"*. Each slab mapped into the texture is a *"patch"*.

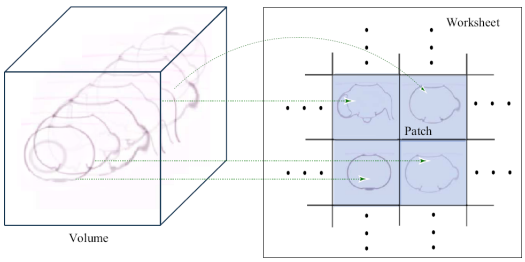

Figure 1.6: Mapping example.

They now have three of these *sheet buffers* one for each axis direction. Finally one more texture called *"composed worksheet"* is made from the contents of those three others. This resultant texture is indeed the final result and, of course, the voxelized model.

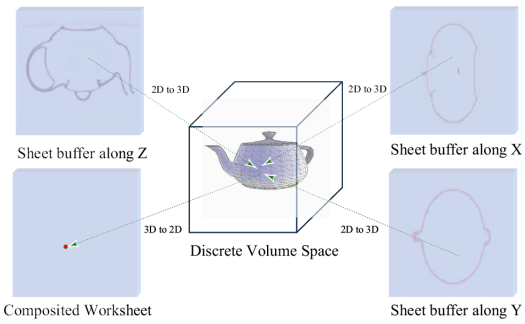

Figure 1.7: Elements of the representation.

Procedure: The voxelization process has three steps: *Rasterization*, *texelization* and *synthesis*.

1. *Rasterization* involves the triangle rasterization and a process to determine which voxels are being intersected with the triangle. For these voxels their 3D coordinates are being computed.

2. *Texelization* step determines in which *sheet buffer*, and texel the voxel is stored and what are the correct offset inside the texel.

3. *Synthesis* stage gets the three *sheet buffers* as an input and as an output takes the *worksheet*. *Rasterization* and *texelization* are done first.

For each axis direction, 1. and 2. must be accomplished, so it is possible to rasterize the whole model three times, but this is time-hard. To avoid this situation the authors say that they adds a preprocess stage before. It consists in reorder the geometry in groups as follows:

- One group for each axis direction. One triangle goes to the group on which its projection over this axis direction has the maximum projected area.
- Inside each axis group have one group for each slab. One triangle goes to group if it intersects with this slab.

This sorting process is done on CPU before execute the voxelization algorithm.

Now they rasterize each axis group separately, storing the resulting voxels into the corresponding *sheet buffer*. For each of these groups, they rasterizes slab by slab, adjusting the near and far clipping planes to the slab boundary to get accurate results. They do not say nothing about what type of camera they used or how they avoid the z distribution problem so we think that an orthogonal camera is a good choice.

To be able to write one bit on a texture they used a lookup texture as follows:

- To store a bit into a component, an 8×1 texture is created. Its sth texel stores 2^s. By setting the alpha blending operation as addition and the source/destination blending factors as one/one, the required bit value can be put at correct location during rasterization.

This way of rendering a model has the advantage of traverse more or less one time the whole model. Of course, this is only correct in some sense, since all the triangles that lies on the interior boundary of a slab must be in two slab groups. Due to this way of rendering, there are some other problems. These boundary triangles has some vertices outside the slab and in order to put the near and far plane, these vertices must taken into account. Other problem is the voxel repetition because if the same triangle is rasterized two times it generates its voxels twice.

The last step is to merge the three *sheet buffers* in one. Since one triangle is only in one axis group, there are no repetition problems here. The idea here is get all the texture information, reproduce the 3D volume coordinates and finally map its to the *worksheet*. It isn't necessary to do this process in this two steps, moreover they say that takes the z-axis *sheet buffer* as the reference and maps the two others, but the process is almost the same if other axis is taken as the reference. In this situation, the reference $-z-$ *sheet buffer* maps directly, the other two trough have to be mapped.

1.4.4.3 Implementation issues

We decided to implement the method over the MRT-FBO technique. Each target is a slab. When a render pass is finalized, a readback process is done to map all patches in the active worksheet.

We know the addition blending would be a problem if two triangles intersect the same voxel.

1.4.5 Conservative Rasterization (Hasselgren *et al.* , 2005)

1.4.5.1 Main idea

This work presents a technique for conservative rasterization. There are two types of conservative rasterization, overestimated and underestimated. For our interest only the overestimated one is necessary. Therefore, only overestimated way is explained here.

In addition, this work presents an over-conservative computation for the depth.

The main goal using conservative rasterization for voxelizations is to recognize some of those voxels which lie in the boundary of a polygon (by using those fragments which intersect the boundary of the projection). We assume that if the input model is not a triangulated mesh, before start the rasterization, a triangulation process is performed.

We use this technique inside the previously presented techniques to know how it fits in voxelization process and how well the presented algorithms handle conservative rasterization. To know how it fits read Chapter 3.

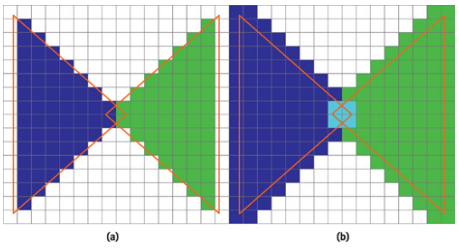

Figure 1.8: Comparison between standard and overestimated conservative rasterization.

1.4.5.2 The method

The *exact* bounding polygon that contains exactly those fragments which intersects the boundary of a triangle will be computed. This is an example of exact

bounding polygon:

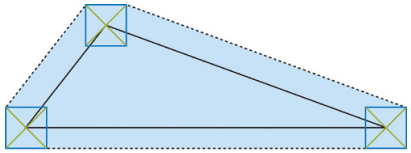

Figure 1.9: This polygon is the result of the mathematical dilation operator.

To compute this new polygon, we get the vertices of the triangle, and for each one we compute how many new vertices appear and which is its positions. The position for a new vertex is *always* one corner of a *virtual* fragment cell centered on the triangle vertex. There are intuitively three possible cases, denoted as if one, two, or three vertices are created. Given two edges e_1 and e_2 connected in a vertex v, the three cases are the following:

- If the normals of e_1 and e_2 lie in the same quadrant, the convex hull is defined by the point found by moving the vertex v by the semi-diagonal in that quadrant (Figure 1.10a).
- If the normals of e_1 and e_2 lie in neighboring quadrants, the convex hull is defined by two points. The points are found by moving v by the semi-diagonals in those quadrants (Figure 1.10b).
- If the normals of e_1 and e_2 lie in opposite quadrants, the convex hull is defined by three points. Two points are found as in the previous case, and the last point is found by moving v by the semi-diagonal of the quadrant between the opposite quadrants (in the winding order) (Figure 1.10c).

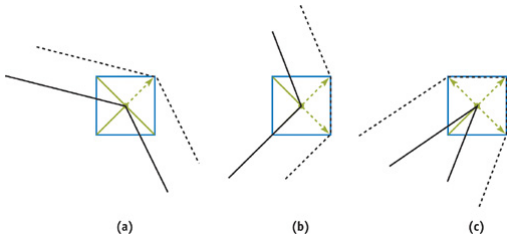

Figure 1.10: Computing an optimal bounding polygon.

Procedure: We must rasterize the whole model, and for each triangle the above considerations are used to create the new bounding polygon. After this, this new polygon replaces the original triangle on the pipeline process, and the per-fragment operations are done over it.

Since a vertex program cannot create new geometry, we cannot create these new vertices and we cannot remove the original ones. We need to send more vertices to the vertex program. Exactly, for each vertex it may generate one, two or three new vertices. Using the original as a new one we need two more. The model is modified before start this process in order to create these geometry.

We create, for each triangle, a triangle fan, from three vertices to nine vertices –three for each– which is the worst case. There are three vertices with the same coordinates. This fan of triangles has coordinate coherence with the original but if you think how a triangle fan is drawn (sharing the first vertex as the first of all) you may note that knowing how is the previous and the next matters since there are not edge information at vertex level:

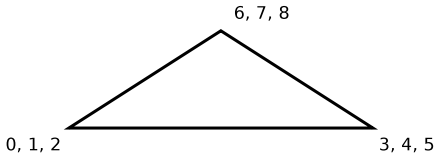

Figure 1.11: The triangle fan before modifications.

For each vertex we send the previous an the next coordinates as texture coordinates as well as the local index in the range $[0, 2]$ (local means which of the three points in the same position it is). The positions and indices are needed to compute which case and which semi-diagonal to use when computing the new vertex position.

The simpler cases from Figure 1.10, resulting in only one or two vertices, are handled by collapsing two or three instances of a vertex to the same position and thereby generating degenerate triangles.

Finally, we output the modified triangle fan from vertex shader, it passes the interpolation process and goes to the per-fragment step. Without any more special, the process ends since we created a new polygon that contains all fragment centers that we need, so we convert the problem to the point-inside-triangle problem which is the policy to create fragments.

In cases with input triangles that have vertices behind the eye, we can get projection problems that force tessellation edges out of the bounding polygon in

its visible regions. To solve this problem, we perform basic near-plane clipping of the current edge. If orthographic projection is used, or if no polygon will intersect the near clip plane, we skip this operation.

1.4.5.3 Conservative depth

When performing conservative rasterization, you often want to compute conservative depth values as well. By conservative depth, we mean either the maximum or the minimum depth values, z_{max} and z_{min}, in each pixel cell.

When an attribute is interpolated over a plane covering an entire pixel cell, the extreme values will always be in one of the corners of the cell. We therefore compute z_{max} and z_{min} based on the plane of the triangle, rather than the exact triangle representation. Although this is just an approximation, it is conservatively correct. It will always compute a z_{max} greater than or equal to the exact solution and a z_{min} less than or equal to it. This is illustrated in Figure 1.12.

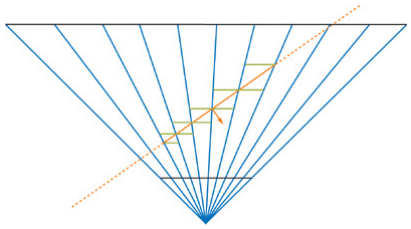

Figure 1.12: A view frustum (in black), with pixel cells (blue lines) and a triangle (orange), as seen from above. The dashed line is the plane of the triangle, and the orange arrow indicates its normal. The range of possible depth values is also shown for the rasterized pixels. The direction of the normal can be used to find the position in a pixel cell that has the farthest depth value. In this case, the normal is pointing to the right, and so the farthest depth value is at the right side of the pixel cell.

The depth computation is implemented in a fragment program. A ray is sent from the eye through one of the corners of the current pixel cell. If z_{max} is desired, we send the ray through the corner found in the direction of the triangle normal; thez_{min} depth value can be found in the opposite corner. We compute the intersection point between the ray and the plane of the triangle and use its coordinates to get the depth value. In some cases, the ray may not intersect the plane (or have an intersection point behind the viewer). When this happens, we simply return the maximum depth value.

We can compute the depth value from an intersection point, if the projection matrix is simple (as produced by **glFrustum**) in a simpler way. Under the assumption that the input is a normal point with $w_e = 1$ (eye-space w component), we can compute the z_w (window-depth) component of an intersection point from the z_e (eye-space z) component. For a depth range $[d_{Far}, d_{Near}]$, we compute z_w as:

$$z_{w_{proj}} = \frac{d_{Far} + d_{Near}}{2} + \frac{d_{Far} - d_{Near}}{2} \left(\frac{z_{Far} + z_{Near}}{z_{Far} - z_{Near}} + \frac{2 z_{Far} z_{Near}}{(z_{Far} - z_{Near})\, z_e} \right) \tag{1.10}$$

$$z_{w_{ortho}} = \frac{d_{Far} + d_{Near}}{2} - \frac{d_{Far} - d_{Near}}{2} \frac{z_{Far} + z_{Near} + 2 z_e}{z_{Far} - z_{Near}} \tag{1.11}$$

They propose use w_c instead z_e. We propose to change this. If depth computation is done in a fragment program, the available values to compute this must be passed. So it better to pass z_e since it enables to compute conservative depth for orthogonal projections as well (they use $z_{w_{proj}}$ equation because inside this $z_e$is the value of w_c but w_c does not appear in $z_{w_{ortho}}$).

1.4.5.4 Implementation issues

We describe algorithm in *window space*, for clarity, but in practice it is impossible to work in window space, because the vertex program is executed before the clipping and perspective projection. Fortunately, our reasoning maps very simply to *clip space*. For the moment, let us ignore the z component of the vertices (which is used only to interpolate a depth-buffer value). Doing so allows us to describe a line through each edge of the input triangle as a plane in homogeneous (x_c, y_c, w_c)-space. The plane is defined by the two vertices on the edge of the input triangle, as well as the position of the viewer, which is the origin, $(0, 0, 0)$. Because all of the planes pass through the origin, we get plane equations of the form

$$ax_c + by_c + cw_c = 0 \Leftrightarrow a(x_d w_c) + b(y_d w_c) + cw_c = 0 \Longrightarrow ax_d + by_d + c = 0 \tag{1.12}$$

The planes are equivalent to lines in two dimensions. In many of our computations, we use the normal of an edge, which is defined by (a, b) from the plane equation.

The algorithm is robust in terms of floating-point errors but may generate front-facing triangles when the bounding polygon is tessellated, even though

the input primitive was back-facing. To solve this problem, we first assume that the input data contains no degenerate triangles. We introduce a value, e, small enough that we consider all errors caused by e to fall in the same category as other floating-point precision errors. If the signed distance from the plane of the triangle to the viewpoint is less than e, we consider the input triangle to be back-facing and output the vertices expected for standard rasterization. This hides the problems because it allows the GPU's culling unit to remove the back-facing polygons.

Chapter 2

Exact GPU Voxelization

We now present a novel GPU-based algorithm for computing theoretically exact voxelizations –without considering precision errors–. Its strength raises from its simplicity, that minimizes floating point errors and makes it faster than a CPU method.

This method exploits high-end hardware capabilities such as geometry shaders, framebuffer objects or multiple render targets. This enables us to obtain a high-resolution voxelization with less render passes and less complexity.

Hereafter, our method is referred also as "Tripiana, 2009".

2.1 Main idea

We start with the idea that the AABB of a triangle can be computed very quickly. For each input triangle, we compute its AABB on the fly (in a geometry shader) and pass it to the rasterization pipeline (instead of the triangle), to rasterize a 2D rectangle that covers all pixels corresponding to voxels potentially intersected by the triangle.

The fragment shader will identify which voxels in voxel grid are intersected. We use the color buffer of the framebuffer to store a bit flag identifying which voxels are intersected at some (x, y, z) object coordinates –(x', y') pixel, bit $z' \in [0, 31]$–. This is also called a slicing method of voxelization, because if we want more depth resolution than 32, we need more elements.

2.2 The method

We start with a triangulated surface-based model M. Let L be the length of its optimal axis aligned bounding cube (AABC) , and let res be the desired grid resolution. Each voxel has $l = \frac{L}{res}$ as edge length.

We use an orthographic camera (which implies fast and simple computations), and this enables us to forget about depth distribution problems –see Subsection 1.2.1 and Paragraph 1.2.2–. The camera is placed in front of the AABC's center, the view direction is parallel to the Z axis and has the same direction of it. We first assume that the grid resolution is small, and it is possible to perform the whole process in a single render pass. So we create a view frustum that matches the AABC of M and z_{Near} and z_{Far} match as well with the AABC.

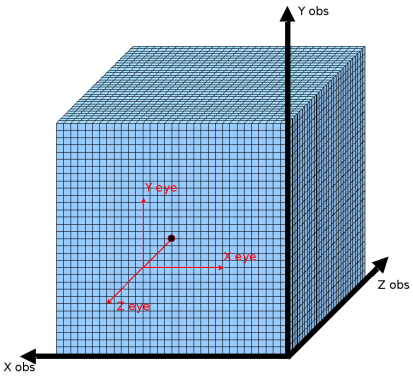

Figure 2.1: An example for $res = 32$ showing how to place the camera. The view frustum matches with the blue voxel grid.

The viewport is made of $res \times res$ pixels with a depth range of $[0, 1]$ as $[z_{Near}, z_{Far}]$. Then, we start an OpenGL render pass. This pass has three steps, namely: vertex, geometry and fragment, which corresponds with the kind of programs OpenGL has. Figure 2.2 shows the whole process.

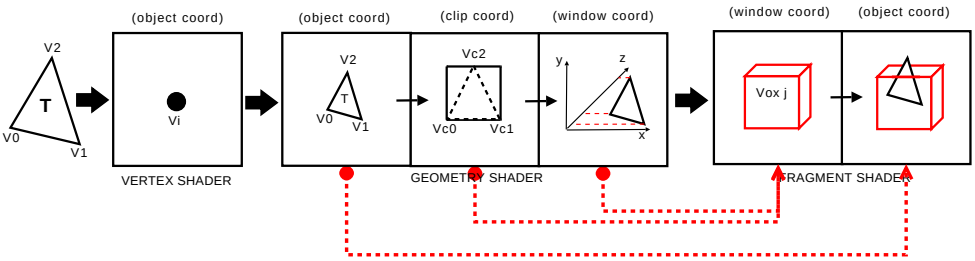

Figure 2.2: Vertices pass trough the vertex shader. Next, in geometry shader, the bounding rectangle and the depth range are computed using the triangle. In fragment shader we use the previous data to know those voxels which potentially intersect the triangle. Finally, the triangle is checked against these voxels to know its intersections.

Vertex Shader

The vertex program in OpenGL is mainly designed to perform operations at vertex level, and to transform the vertex to clip space. But now we omit this transformation and we pass the untransformed vertex to the next stage –see Section A.1–.

Geometry Shader

The geometry shader receives the three vertices of the triangle. This triangle is used in two ways. On the one hand, we get the X, Y and Z of each vertex and pass it to the next stage –in object coordinates–, so that the fragment shader will know about the triangle's geometry.

On the other hand, the triangle vertices are transformed to clip coordinates, which is convenient for the upcoming computations. A vertex in clip coordinates maps to the projection plane its X, Y and the Z, but values are between the range $[-1, 1]$. So the clip X and Y coordinates are the 2D projection that will be used to determine which pixels must be in use. These 2D coordinates are used to compute the axis aligned bounding rectangle (AABR) of the 2D projected triangle. Once we get this AABR, we expand it by a half of a pixel size in clip coordinates ($\frac{1}{res}$). This is important to guarantee that all fragments' centers of the potentially intersecting fragments are covered. If we omit this expansion, some fragments wont be present at fragment stage –see Section A.2–. The resulting 2D rectangle is what OpenGL will rasterize for this primitive.

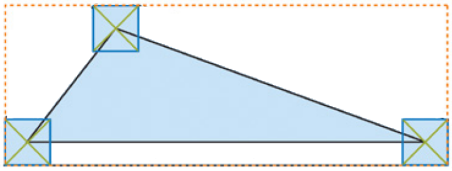

Figure 2.3: The initial triangle is replaced by its AABB. Once it is replaced, its AABB is expanded by a half of a pixel size.

The geometry shader also computes the depth range for that triangle. We transform the Z coordinate for a given vertex from clip to window space. We select the maximum and minimum values –it is similar to the AABR computation– and now the range is expanded by a half of a voxel size but in window coordinates ($\frac{1}{2res}$). Using this information and the AABR we get the AABB of the

triangle. Now we have the X and Y in clip and the Z in window coordinates –we fix the Zs for AABRs in the middle of the view frustum, $z = 0$, to make it visible–. The AABR replaces the triangle in the rasterization pipeline, so this is the geometry that will be rasterized. The depth range is passed for each vertex of the new primitive identically. This avoids the interpolation process for fragment's values. The initially mentioned triangle vertices are as well passed identically for each vertex.

Fragment Shader

Each execution of the fragment program knows the coordinates of the fragment's center (by `gl_FragCoord`), which corresponds to the (X, Y) of the center of a voxel row in window space. We first set to black $-(0, 0, 0, 0)-$ the fragment's color for this fragment, which is the same as mark as unused the whole voxel row –Figure 2.4–.

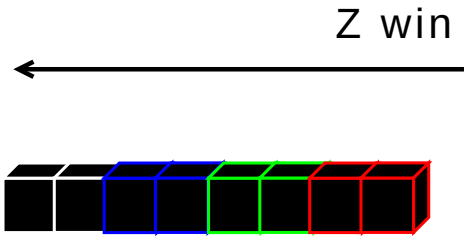

Figure 2.4: The initial voxel row (simplified with 2 bits per color channel).

We also know the depth range for that voxel row, and it is used to create a loop stopping in each z_k which is corresponding to a center of a voxel in window coordinates. These coordinates are transformed to object space, getting the voxel's center in object space –Figure 2.5–.

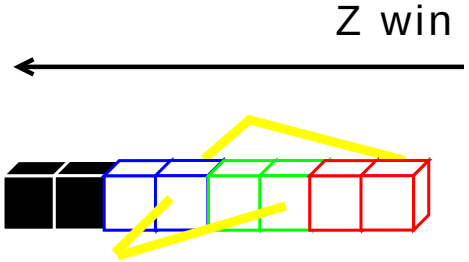

Figure 2.5: The loop range in the column row, detected the depth range, are colored in white in this picture, but still are set to 0.

We pass also the voxel's edge half-length, which is constant, to the fragment shader. Using the voxel's center, the triangle's vertices, and the edge half-length we apply the separate axis theorem (SAT) to test if a voxel intersects the triangle. If the voxel intersects the triangle we set the kth bit of the color buffer to 1 –Figure 2.6–.

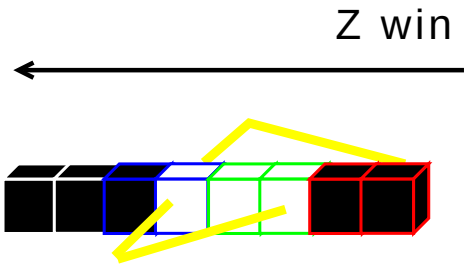

Figure 2.6: Intersected voxels inside the depth range.

To make that possible, we use a 1D texture bitmask of 32 texels, each of those with a value between $[2^0, 2^{31}]$, for the kth bit mask is 2^k. Also, as we possibly are setting more than 1 bit to 1 in this voxel row, we need to sum the previous value in the buffer to the current one in the loop. This is not a problem since each step in the loop corresponds to one voxel of the row (a bit in the buffer), so the addition does not make carry bit operations –see Section A.3–. The last operation is to merge the current value of the color buffer in the fragment shader with the old one from previous executions (other triangles). This can be easily accomplished by the OR logic operation that OpenGL provides in the color buffer.

2.3 Implementation issues

In our case the depth resolution limitation is solved by setting more color buffers to a FBO –MRT technique–. And, if we need more, we use more than 1 FBO. For each FBO we perform a single render pass. For example, given a GPU with up to 8 color buffers for each FBO, we can perform a voxelization of $256 \times 256 \times 256$ in a single rendering pass. Please, note that X and Y coordinates could grow as high as the maximum texture size. Inside the fragment program it is possible to use the current color buffers for a given FBO transparently using `gl_FragData[gl_MaxDrawBuffers]`. Other example, a high-resolution

voxelization, such as $2048 \times 2048 \times 2048$ needs $\frac{2048}{256} = 8$ render passes. We use in our implementation vertex buffer objects (VBO), to store the whole geometry in GPU one time and use it to perform multiple render passes quickly. Also, performing multiple render passes with FBOs is more efficient than using the framebuffer because we don't need to restore the framebuffer status for each render pass.

For each FBO we set the near and far planes $-z_{Near}$, $z_{Far}-$ to enclose only the voxelized area. This enables us to simplify the computation and get small values, also guarantying more bits to the fractional part, so as to minimize floating point errors. But the best improvement is to discard triangles in geometry stage. If all triangle's vertices have the Z coordinate less than -1 or all are greater than 1 (in clip space), then this triangle is completely outside the current $[z_{Near}, z_{Far}]$ range. There are no *discard* instruction in geometry shading, but we can modify the Z coordinates of the AABR vertices to put the geometry outside the frustum. The current hardware has the *early z-cull* test, which means that all the geometry outside the Z planes is clipped and wont generate fragments. This improvement greatly improves the performance of the algorithm, as the hardest task in our algorithm is the triangle-voxel intersection test done in the fragment shader.

In fragment shader the SAT is used to test the intersection against the portion of the voxel row identified by one fragment. When we are testing one voxel against the minimal AABB around the triangle, we apply the Z axis first, because for these three tests this is the most frequent failing test.

Chapter 3

Comparison

3.1 Considerations

Some of the algorithms we discussed are view-dependent (i.e. Eisemann & Décoret, 2006 and Eisemann & Décoret, 2008). We have implemented and tested also these algorithms but, in order to perform comparison test, these techniques do not fit well with the view-independent ones. View-dependent algorithms use the current view frustum as the voxel grid, and thus the grid wont be axis-aligned. The usage of a perspective camera would replace cubic voxels by prisms.

To be able to compare these techniques, we have developed new algorithms (modifications of the original ones). These algorithms use an orthogonal camera, and the camera's position is fixed parallel to the Z axis, looking to the center of the bounding cube of a given model.

The implemented conservative rasterization (Hasselgren *et al.* , 2005) works without conservative depth.

In Subsection 3.3.2 and Subsection 3.3.3 we have computed the voxels that a technique misses or adds compared with the CPU voxelization, that we take as ground truth (Akenine-Möller, 2001). The Eisemann & Décoret, 2008 technique is omitted here. The algorithm is used to detect the interior voxels for a given model, so it is not possible to compare the results.

We have tested our work in a workstation with an *Intel i7 processor at* 2.93 *GHz*, with 3 *GB of RAM type DDR3* and 2 *GB of SWAP*. The system is equipped with a *nVIDIA GeForce GTX 295* with 896 *MB of GRAM type DDR4 for each GPU -it has two-*. Only one GPU for each connected screen can be used with the current OpenGL implementation. We developed our testing application in C++ –ISO/IEC 14882:1998 compliant– over a *Linux x32 OS* – kernel 2.6.28-15 i686 SMP–. Our graphic card supports up to 8 color buffers

for each FBO. To test possible hardware/driver issues we decided to sample our tests with those voxel resolutions that match with a fixed number of complete FBOs and the following one, which is the same number of complete FBOs, and 1 more FBO with only 1 target in use (resolutions are in the succession 32 –1 FBO with 1 target–, 256 –1 FBO with 8 targets–, 288 –2 FBO, $8+1$ targets–, 512 –2 FBO, $8+8$ targets–, ...).

Table 3.1 shows the models we used in the evaluation.

MODEL	KNOW ISSUES	No. OF TRIANGLES
Stanford Bunny	non-manifold	69,451
Armadillo	none	345,944
Skeleton hand	high depth complexity	654,666
Happy Buddha	through holes & geometry cracks	1,087,716
Turbine blade	geometrically complex	1,765,388

Table 3.1: Tested model.

3.2 Execution time

Figures from 3.1 to 3.5 show the execution time of the implemented techniques for the different test models.

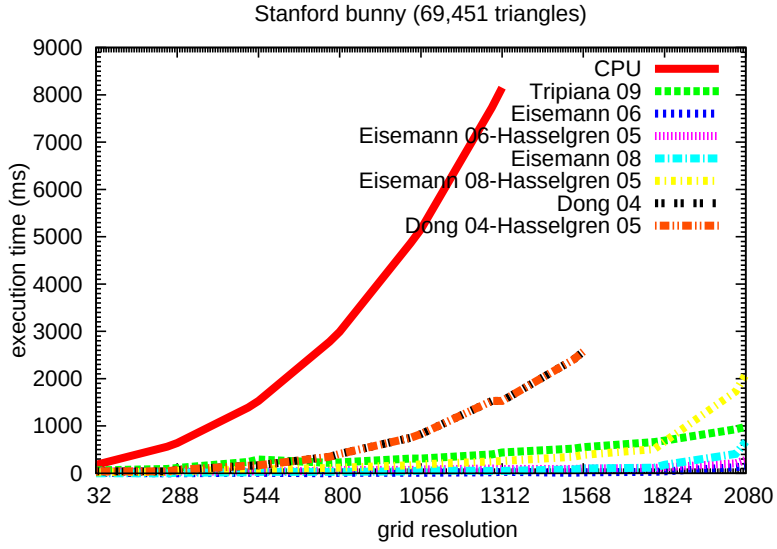

Figure 3.1: Timing result for "Stanford Bunny".

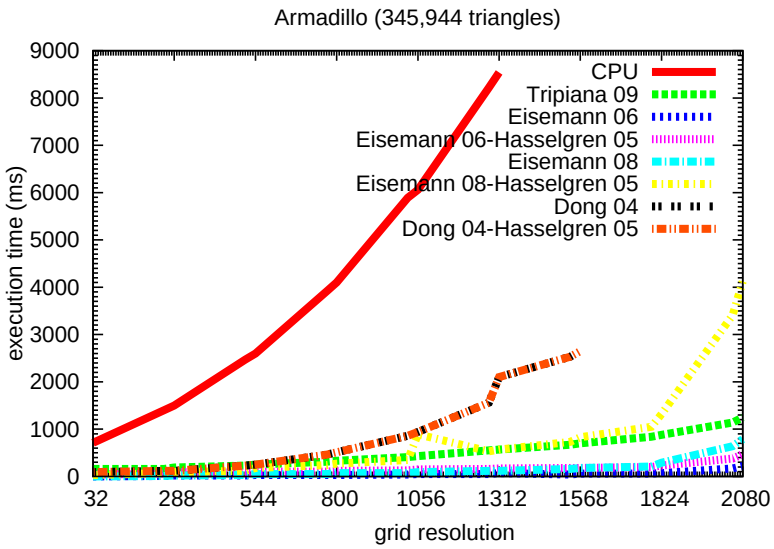

Figure 3.2: Timing results for the "Armadillo".

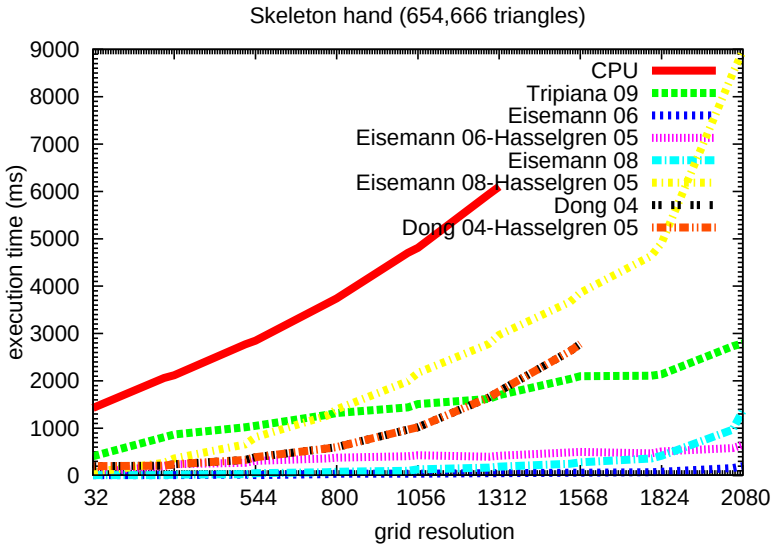

Figure 3.3: Timing result for the skeleton hand.

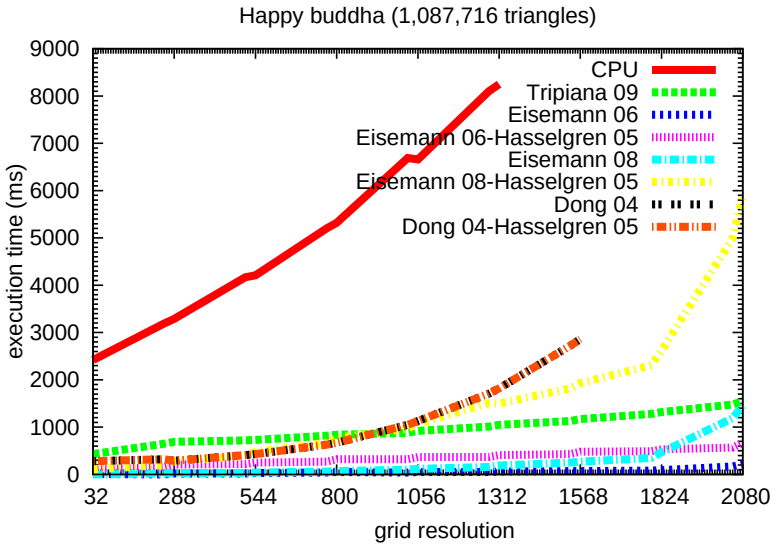

Figure 3.4: Timing results for the "Happy Buddha".

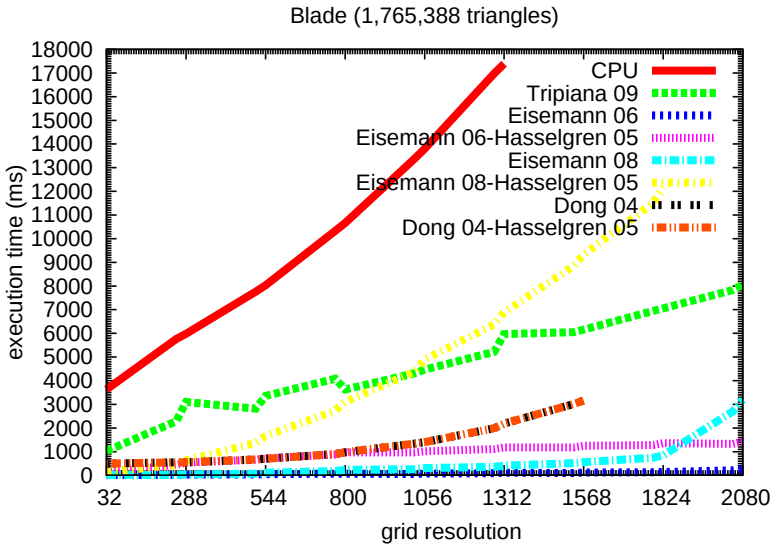

Figure 3.5: Timing result for the "Turbine blade" model.

From the timing results, we can conclude that our method is always faster than the CPU method. The inherent parallelism of the GPU gives to our method a speed-up of $\times 4$ approximately for the "Turbine blade" model.

Our method is a bit slower that other GPU-based approaches, but as it is shown in accuracy tests, our method is *exact*. This implies generate more fragments, so it is obvious it will take some more time to finish the process. Other methods approximates the voxelization, the lower number of detected voxels make these algorithms faster.

For Eisemann & Décoret, 2008 in combination with Hasselgren *et al.* , 2005, up to some resolution (depending on the model) becomes slower than our method. We know this solid voxelization works similarly as the boundary one (Eisemann & Décoret, 2006). The only thing to make slow the execution is the XOR blending, and this is hardware/driver dependent.

3.3 Accuracy

3.3.1 Number of voxels detected

Figures from 3.6 to 3.10 show the number of detected voxels, including those which may be erroneous.

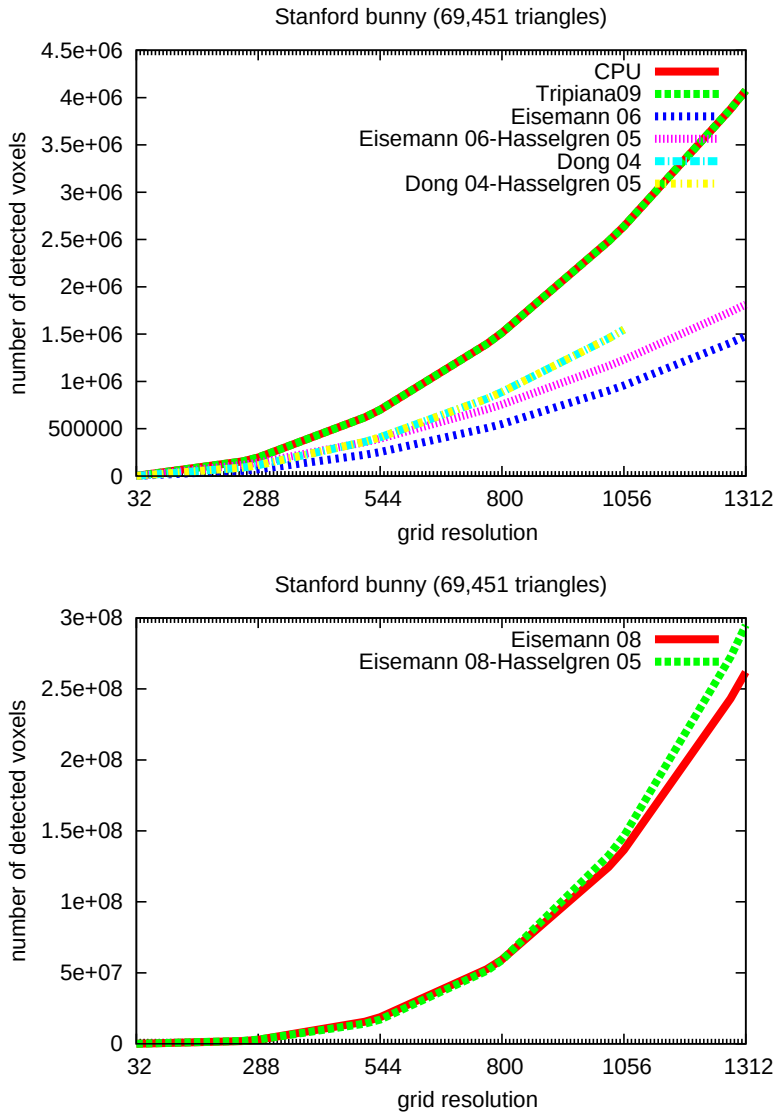

Figure 3.6: Voxels detected for "Stanford Bunny".

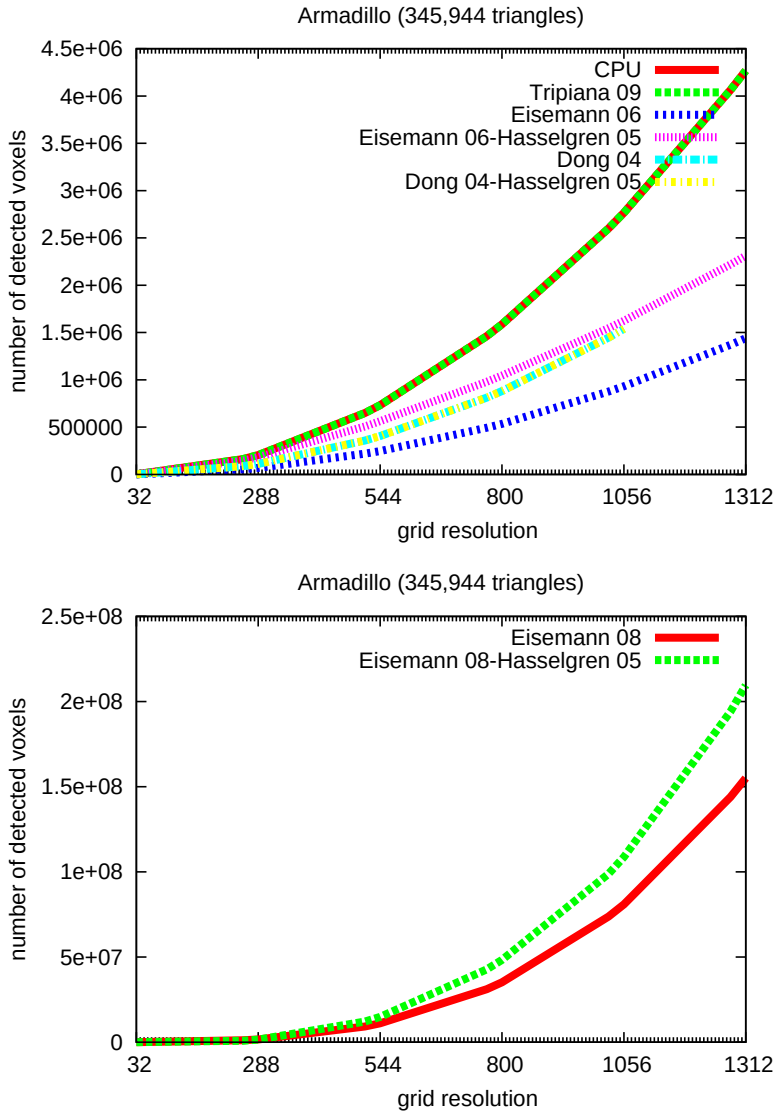

Figure 3.7: Voxels detected for the "Armadillo".

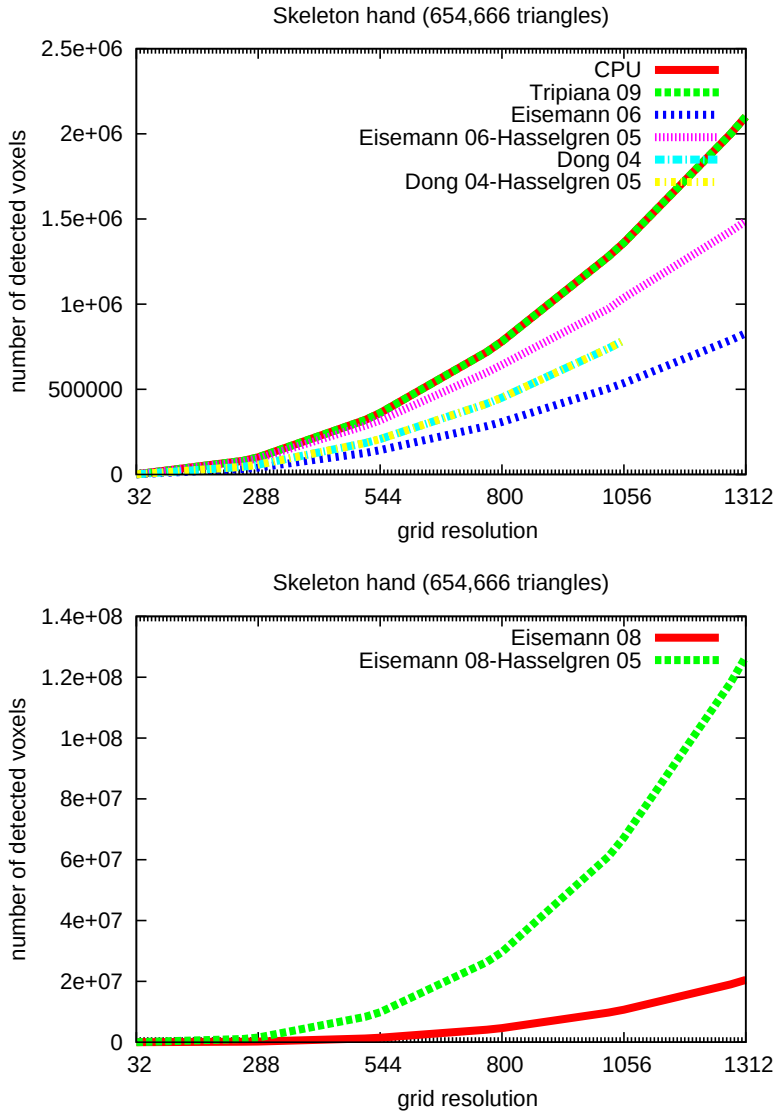

Figure 3.8: Voxels detected for the skeleton hand.

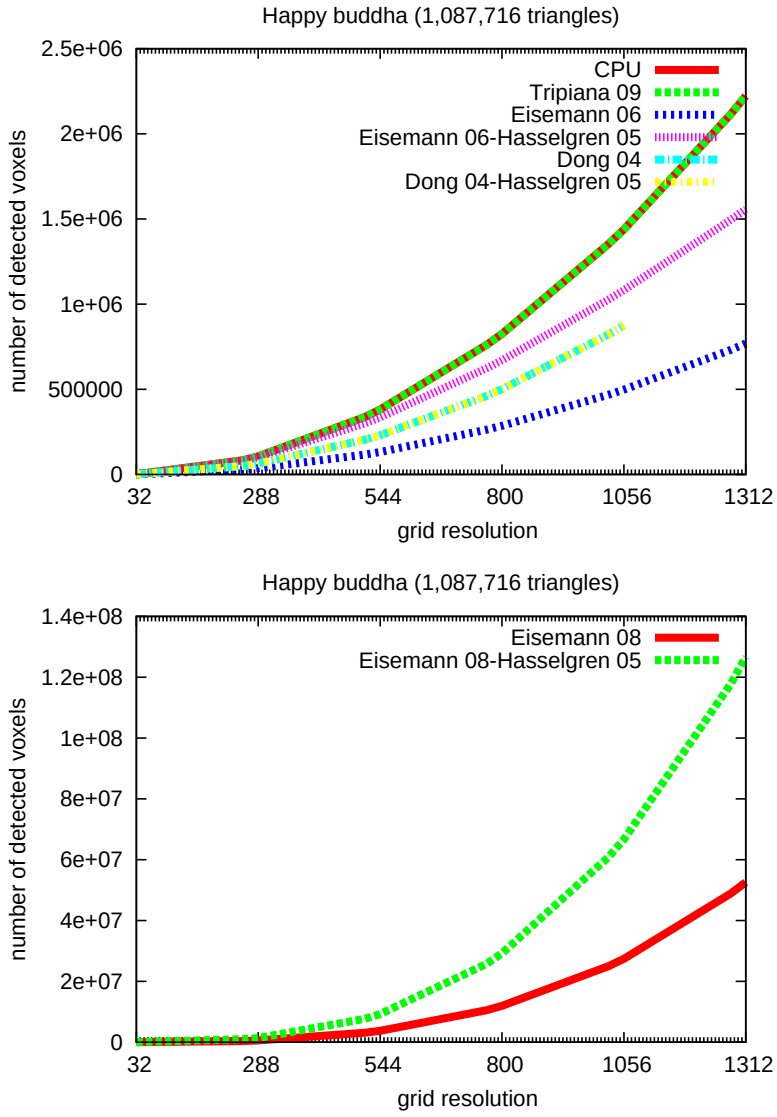

Figure 3.9: Voxels detected for the "Happy Buddha".

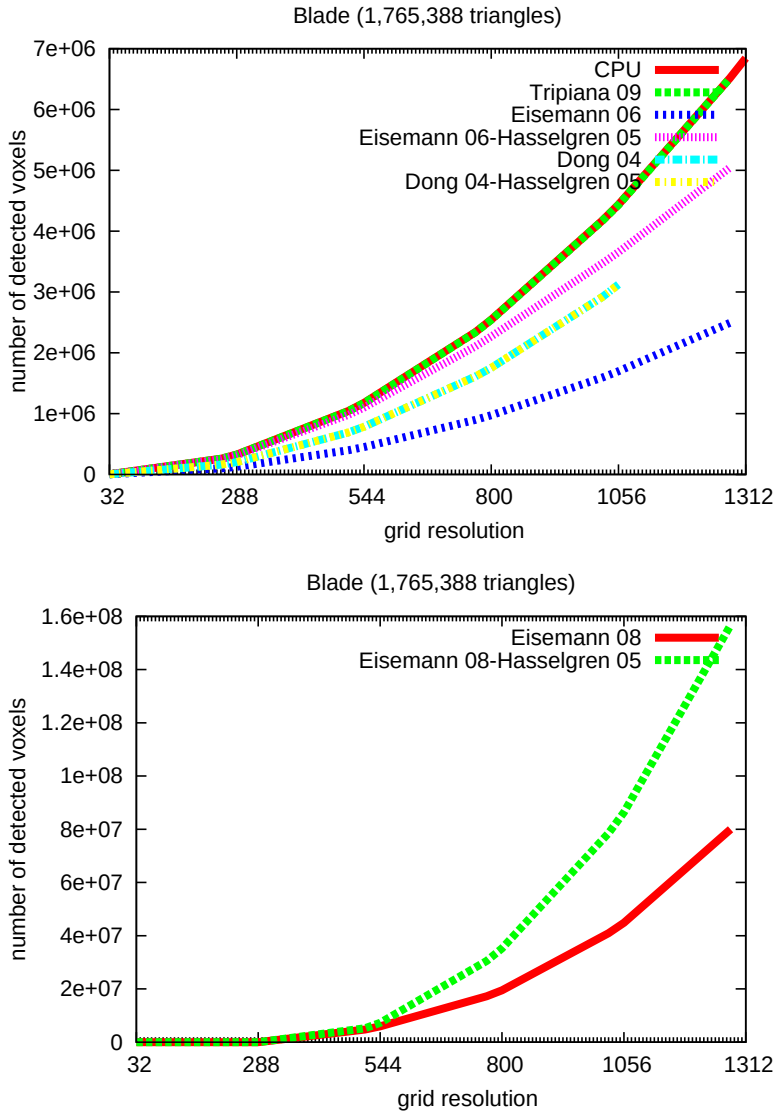

Figure 3.10: Voxels detected for the "Turbine blade" model.

Our algorithm detects the same number of voxels as the CPU-based reference algorithm, while others do not detect properly all the intersecting voxels.

The multipass technique (Dong *et al.* , 2004a) has the same accuracy for conservative or non-conservative. This is because it is rasterizing each triangle to the plane in which it has its maximum projection. This is more or less as if we are doing a conservative voxelization. But this technique uses the alpha blending technique to store voxel bits, and this has carry bit problems. This is the reason to get less voxels than other methods.

3.3.2 Missing voxels

Figures from 3.11 to 3.15 show the number of missed voxels, i.e. voxels labeled as non-intersecting which are detected as such by the reference algorithm.

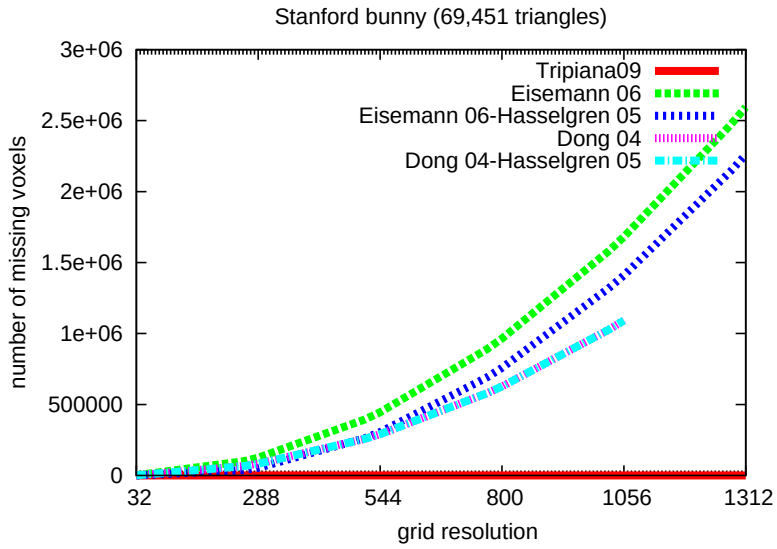

Figure 3.11: Missing voxels for "Stanford Bunny".

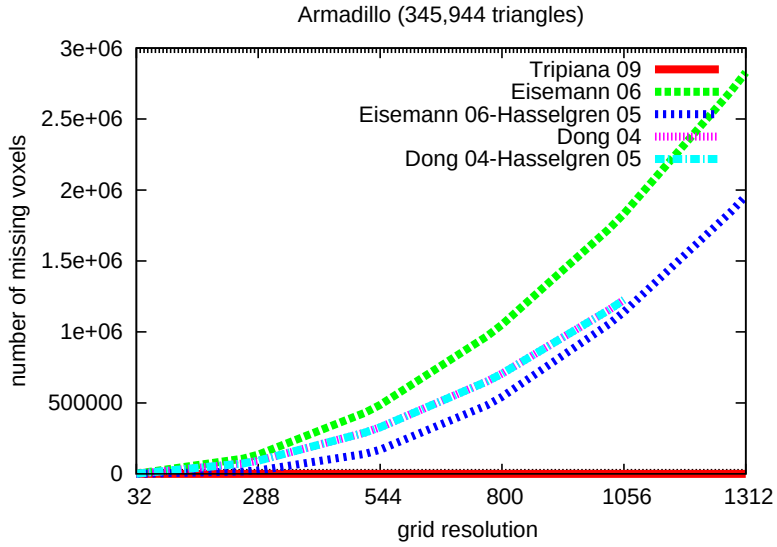

Figure 3.12: Missing voxels for the "Armadillo".

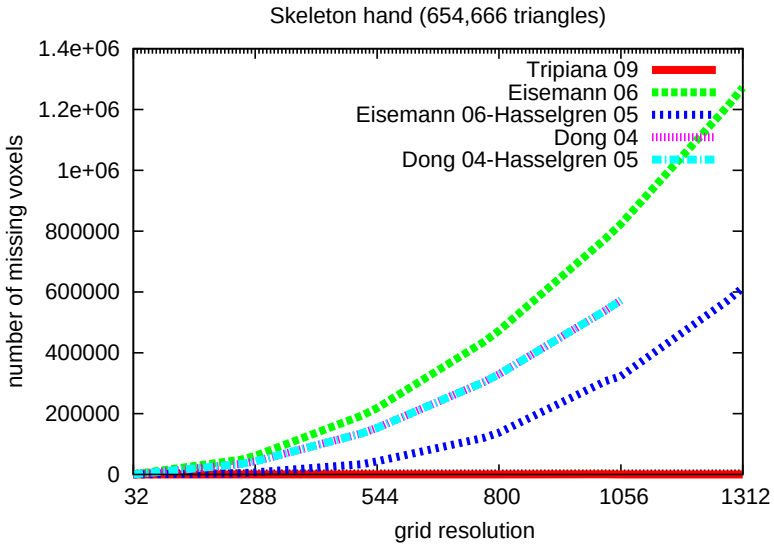

Figure 3.13: Missing voxels for the skeleton hand.

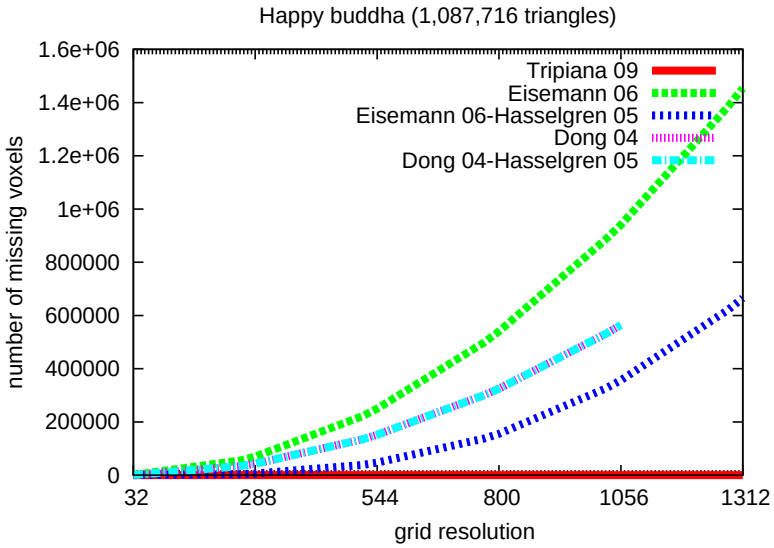

Figure 3.14: Missing voxels for the "Happy Buddha".

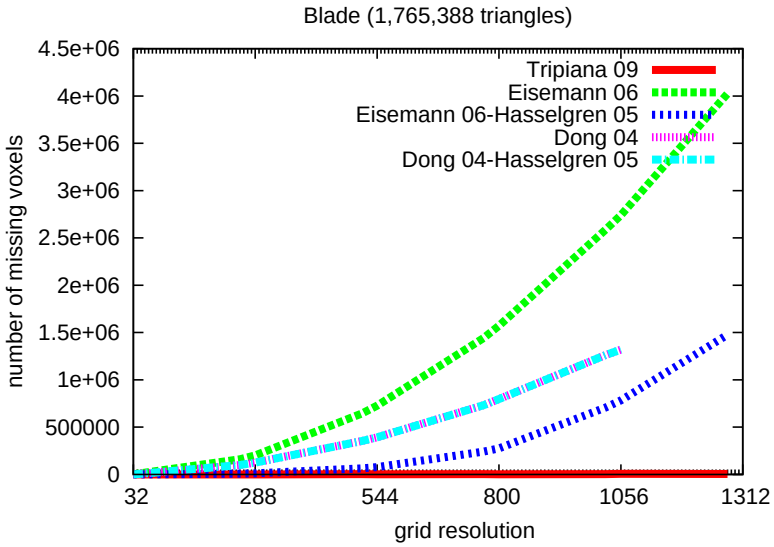

Figure 3.15: Missing voxels for the "Turbine blade" model.

Note that other techniques miss a large number of intersected voxels, as Subsection 3.3.1 tests show as well, but our algorithm keeps all the voxels.

These algorithms has some problems. On the one hand, non-conservative methods have projection/non-conservative rasterization problems. In the other hand Dong *et al.* , 2004a has the alpha blending issue.

Conservative rasterization (Hasselgren *et al.* , 2005) enables a better approximation in use with Eisemann & Décoret, 2006, but still have problems with the Z axis direction.

Figure 3.16 shows a detailed comparison of Eisemann & Décoret, 2006 (using conservative rasterization) and Dong *et al.* , 2004a between our method for the "Stanford Bunny" model.

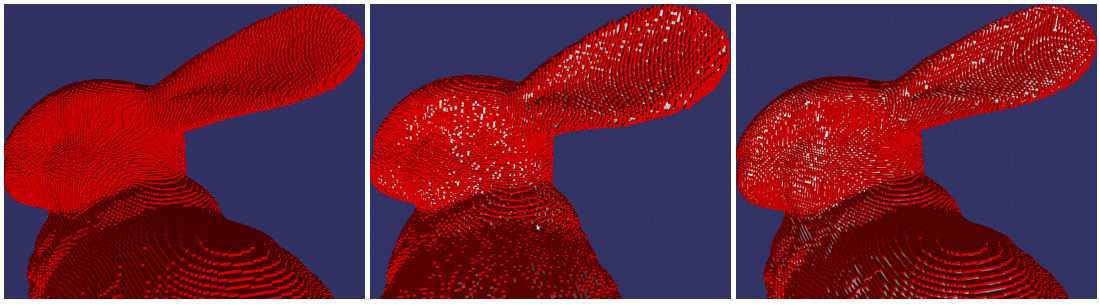

Figure 3.16: From left to right. The "Stanford Bunny" voxelized at 288^3 with our method, Eisemann 06-Hasselgren 05 and Dong 04.

3.3.3 Spurious voxels

Figures from 3.17 to 3.21 show the number of spurious voxels, i.e. voxels labeled as intersected which are not detected as such by the reference algorithm.

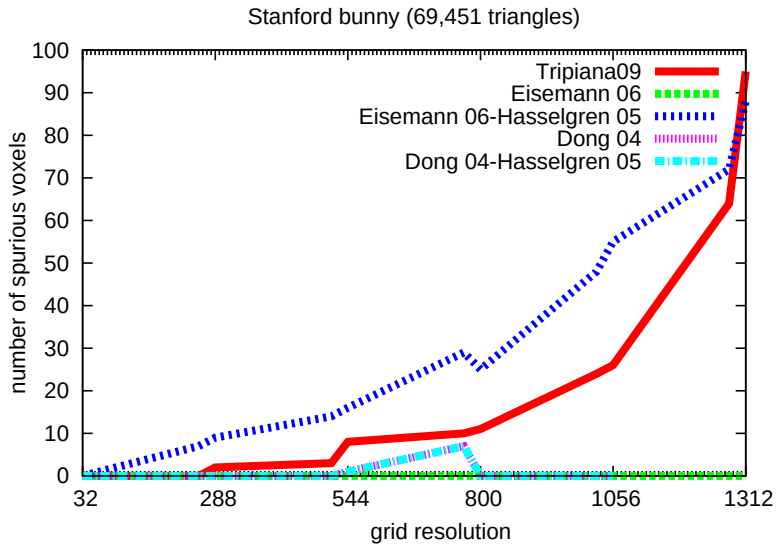

Figure 3.17: Spurious voxels for "Stanford Bunny".

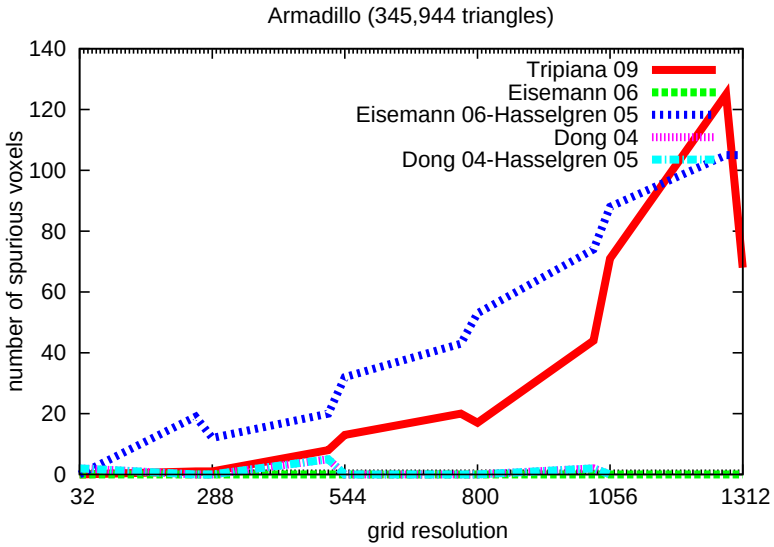

Figure 3.18: Spurious voxels for the "Armadillo".

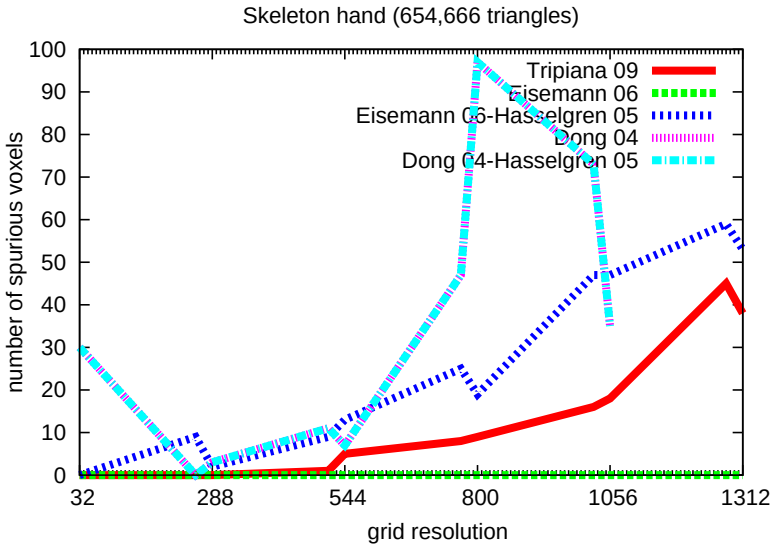

Figure 3.19: Spurious voxels for the skeleton hand.

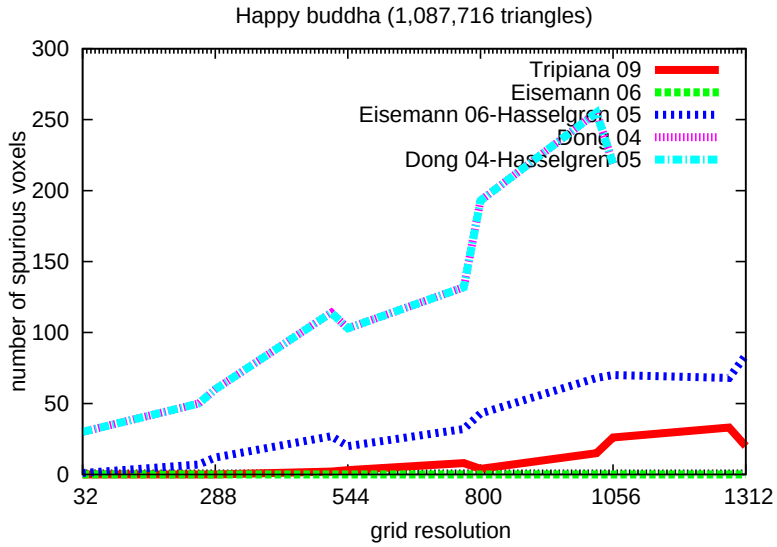

Figure 3.20: Spurious voxels for the "Happy Buddha".

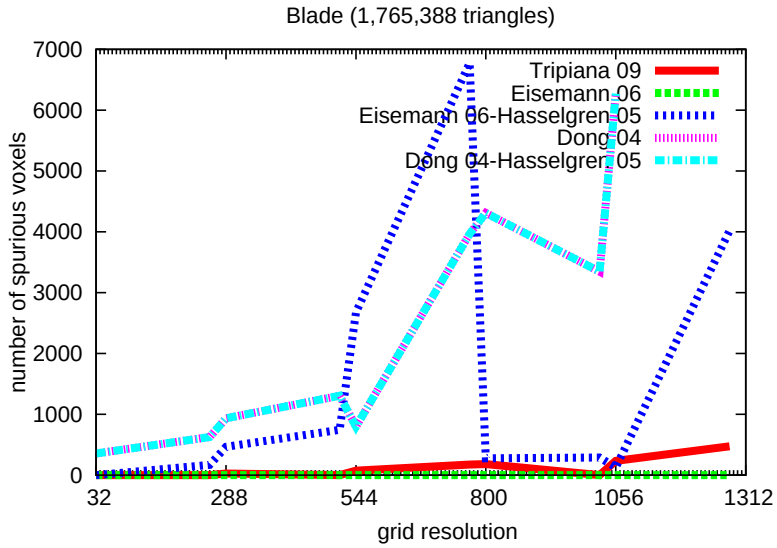

Figure 3.21: Spurious voxels for the huge "Turbine blade" model.

A detailed graph of spurious voxels shows that our process adds some non-intersecting voxels. These number of spurious voxels is similar to the number of missing voxels. This affirmation is only valid for our method.

We detect, in our resulting data set, that these spurious voxels are 26-neighbor of a missing one. We think this is due to floating point errors, coming from the interpolation process or the machine error ε of our graphic card. We use the CPU method as reference, but it is affected by the ε errors as well as our method. We think the fact of one voxel detected by our method and not by CPU probably wont be a problem since it is possible that this voxel may be detected properly. Of course we may still having other spurious voxels coming from the interpolation process.

Conservative algorithms add some voxels, because it is an overestimating rasterization, but the multipass technique (Dong *et al.* , 2004a) has the same number of spurious voxels for conservative or non-conservative rasterization.

Figure 3.22 shows a comparison between the CPU method and the Dong *et al.* , 2004a (using conservative rasterization). These pictures show one spurious voxel which is not present in the CPU method.

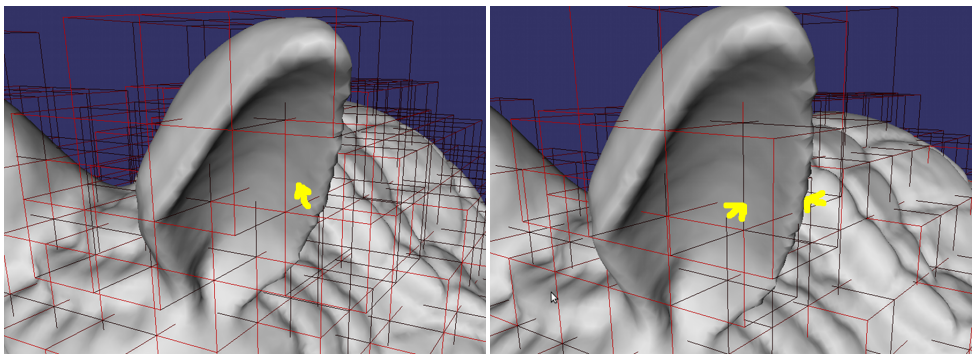

Figure 3.22: From left to right. The "Armadillo" voxelized at 32^3 with CPU method and Dong04.

Chapter 4

Conclusion

We have studied, implemented and tested the most remarkable GPU-based voxelization algorithms in the literature over a variety of model and test cases. We have shown that current GPU-based voxelization algorithms provide only an approximate solution to the problem, with many missing voxels and even some spurious voxels. This lack of accuracy makes these algorithms less attractive for real use.

We have presented a novel GPU-based voxelization achieving exact voxelizations. Our algorithm performs much faster than a state-of-the-art CPU-based reference version and provides an accurate voxelization. Previously developed GPU-based voxelization algorithms are far from achieving an exact solution but our method produces it.

We created a multi-purpose, extensible voxelization API with about 15,000 lines of code, including all implemented techniques as well as an interface to load, store and handle 3D models and their voxelizations. We used it to test our work and we will probably use our API in future projects.

Chapter 5

Future work

There are several lines to extend our work.

We hypothesize that most of our floating point errors (differing from those which CPU has) come from the interpolation process done for the *varying* variables which are used to make the triangle vertices available to the fragment shader. We know OpenGL 3.2 supports GLSL 1.50. This version makes it possible to declare output variables from one shader as an input to the next shader with a ***flat*** qualifier. This qualifier disables the interpolation process: in fragment stage the value for this variable is the value assigned to the provoking vertex. Also GLSL 1.50 lost the concept of *varying* to add a new concept of *input/output* variables. This follows the Cg behavior. With this, the interpolation deviation will be solved.

Our algorithm can be improved in a number of ways. In the implemented version, it checks in a loop those voxels, for a given voxel row, that potentially intersect the triangle. This loop runs over a portion of the voxel row. In rare cases this loop will become long, but we have devised an easy way to speed-up this loop. The loop is split in half, and we execute one step of these new loops at each step. If both steps detect intersection or non-intersection for a given index we will continue running both branches, but if both are non-intersecting and one becomes intersecting the other branch is discarded. On the other hand, if both are intersecting and one becomes non-intersecting this branch is finished.

It would be also interesting to explore a multi-GPU based algorithm, which appears to be simple to implement. It is well know nowadays there are multi-GPU cards and also the possibility to interconnect many graphic cards between then (for example the LSI). It is a good improvement. Our test platform offers to us this possibility. To make it possible, we can use the "alternate frame rendering" (AFR) technique developed by nVIDIA. The concept of frame rendering matches with an OpenGL render pass easily, and it is optimized to do it with

VBOs and FBOs, as our algorithm uses.

We also plan to extend our work to identify also in/out voxels.

Chapter 6

Acknowledgments

We would like to acknowledge the authors of those work that we have implemented. Inside each work, we discovered great ideas, and some problems also. With each of those, we learned a lot.

We also acknowledge to the *"Stanford 3D Scanning Repository"* and *"Large Geometric Models Archive at Georgia Tech"* maintainers.

The student wants to thank his family and friends the support they have offered, and the encouragement and confidence they have shown. He would like to acknowledge also the good work of the teacher staff who makes possible the visual computing intensification of this master. Their great job is the best reference, and the student has his knowledge due to them.

Finally my last words are for my advisor. I have learned working with you that if you like something, if you want it, there are no reasons, and there are no things, that will keep you off the way of achieving this.

Bibliography

Akenine-Möller, Thomas. 2001. Fast 3D triangle-box overlap testing. *Journal of Graphics Tools*, **6**(1), 29–33.

Akenine-Möller, Tomas, & Aila, Timo. 2005. Conservative and Tiled Rasterization Using a Modified Triangle Set-Up. *Journal of Graphics Tools*, **10**(3), 1–8.

Dong, Zhao, Chen, Wei, Bao, Hujun, Zhang, Hongxin, & Peng, Qunsheng. 2004a. Real-time Voxelization for Complex Polygonal Models. *Pages 43–50 of: Proceedings of 12th Pacific Conference on Computer Graphics and Applications*, vol. 0. IEEE Computer Society.

Dong, Zhao, Chen, Wei, Bao, Hujun, Zhang, Hongxin, & Peng, Qunsheng. 2004b. A Smart Voxelization Algorithm. *Pages 73–78 of: Proceedings of 12th Pacific Conference on Computer Graphics and Applications*, vol. 0. IEEE Computer Society.

Eisemann, Elmar, & Décoret, Xavier. 2006. Fast Scene Voxelization and Applications. *Pages 71–78 of: Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*. Association for Computing Machinery, Inc.

Eisemann, Elmar, & Décoret, Xavier. 2008. Single-Pass GPU Solid Voxelization for Real-Time Applications. *Pages 73–80 of: Proceedings of Graphics Interface*. Canadian Information Processing Society.

Hasselgren, Jon, Akenine-Möller, Tomas, & Ohlsson, Lennart. 2005. *Conservative Rasterization*. GPU Gems, no. 2. Addison-Wesley Professional. Chap. 42, pages 677–690.

Hsieh, Hsien-Hsi, Lai, Yueh-Yi, Tai, Wen-Kai, & Chang, Sheng-Yi. 2005. A Flexible 3D Slicer for Voxelization Using Graphics Hardware. *Pages 285–288 of: Proceedings of the 3rd International Conference on Computer Graphics and Interactive Techniques in Australasia and South East Asia*. Association for Computing Machinery, Inc.

Kessenich, John. 2006 (September). *The OpenGL(R) Shading Language (Version 1.20, rev. 8).* Tech. rept. 3Dlabs, Inc. Ltd.

Llamas, Ignacio. 2007. Real-Time Voxelization of Triangle Meshes on the GPU. *Page 18 of: ACM SIGGRAPH 2007 Sketches.* Association for Computing Machinery, Inc.

Martz, Paul. 2006. *OpenGL(R) Distilled.* Addison-Wesley Professional.

Martz, Paul. 2007. *OpenSceneGraph Quick Start Guide: A Quick Introduction to the Cross-Platform Open Source Scene Graph API.* Skew Matrix Software.

Rost, Randi J. 2005. *OpenGL(R) Shading Language (2nd Edition).* Addison-Wesley Professional.

Segal, Mark, & Akeley, Kurt. 2006 (December). *The OpenGL(R) Graphics System: A Specification (Version 2.1).* Tech. rept. Khronos Group.

Shreiner, Dave, Woo, Mason, Neider, Jackie, & Davis, Tom. 2005. *OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2 (5th Edition).* Addison-Wesley Professional.

Zhang, Long, Chen, Wei, Ebert, David S., & Peng, Qunsheng. 2007. Conservative voxelization. *The Visual Computer*, **23**(9), 783–792.

Appendix A

Exact GPU Voxelization shader code

A.1 Vertex shader

```
#version 120


void voxelizationEngine(void)
{
    gl_Position = gl_Vertex;
}
```

A.2 Geometry shader

```
#version 120
#extension GL_EXT_geometry_shader4 :  enable


#define ZNEARCLIP vec3(-1.0)
#define ZFARCLIP vec3(1.0)
#define ZINSIDECLIP 0.0
#define ZOUTSIDECLIP -2.0
```

```
uniform float numVoxels;

uniform float halfVoxelSizeNormalized;

uniform vec2 halfPixelSize;


varying out vec3 vertex0;

varying out vec3 vertex1;

varying out vec3 vertex2;

varying out vec2 voxDepthRange;


void voxelizationEngine(void)

{

    vec4 triV0, triV1, triV2, AABB;

    vec3 depths;

    float zd1, zd2;

    vertex0 = gl_PositionIn[0].xyz;

    vertex1 = gl_PositionIn[1].xyz;

    vertex2 = gl_PositionIn[2].xyz;

    triV0 = gl_ModelViewProjectionMatrix * gl_PositionIn[0];

    triV1 = gl_ModelViewProjectionMatrix * gl_PositionIn[1];

    triV2 = gl_ModelViewProjectionMatrix * gl_PositionIn[2];

    depths = vec3(triV0.z, triV1.z, triV2.z);

    AABB = triV0.xyxy;

    if (all(lessThan(depths, ZNEARCLIP) ) ||
all(greaterThan(depths, ZFARCLIP) ) )
```

```
    {
        voxDepthRange = vec2(ZOUTSIDECLIP);
        gl_Position = vec4(AABB.xw, ZOUTSIDECLIP, 1.0);
        EmitVertex();
        gl_Position = vec4(AABB.xy, ZOUTSIDECLIP, 1.0);
        EmitVertex();
        gl_Position = vec4(AABB.zw, ZOUTSIDECLIP, 1.0);
        EmitVertex();
        gl_Position = vec4(AABB.zy, ZOUTSIDECLIP, 1.0);
        EmitVertex();
    }
    else
    {
        AABB = vec4(min(min(AABB.xy, triV1.xy), triV2.xy),
max(max(AABB.zw, triV1.xy), triV2.xy) );
        AABB += vec4(-halfPixelSize, halfPixelSize);
        voxDepthRange.xy = vec2(++triV0.z * 0.5);
        zd1 = ++triV1.z * 0.5;
        zd2 = ++triV2.z * 0.5;
        voxDepthRange = vec2(min(min(voxDepthRange.x, zd1), zd2),
max(max(voxDepthRange.y, zd1), zd2) );
        voxDepthRange += vec2(-halfVoxelSizeNormalized,
halfVoxelSizeNormalized);
        voxDepthRange = floor(clamp(voxDepthRange, 0.0, 1.0) *
numVoxels);
        gl_Position = vec4(AABB.xw, ZINSIDECLIP, 1.0);
```

```
        EmitVertex();
        gl_Position = vec4(AABB.xy, ZINSIDECLIP, 1.0);
        EmitVertex();
        gl_Position = vec4(AABB.zw, ZINSIDECLIP, 1.0);
        EmitVertex();
        gl_Position = vec4(AABB.zy, ZINSIDECLIP, 1.0);
        EmitVertex();
    }
}
```

A.3 Fragment Shader

```
#version 120

#define INVDEPTH 0.03125

uniform float numRenderTargets;
uniform sampler1D bitmask;
uniform vec3 origBBox;
uniform float halfVoxelSize;
uniform float voxelSize;

varying in vec3 vertex0;
varying in vec3 vertex1;
varying in vec3 vertex2;
varying in vec2 voxDepthRange;
```

```
bool planeBoxOverlap(in vec3 normal, in float d, in float maxVox)
{
    vec3 vMin, vMax;
    if(normal.x > 0.0)
    {
        vMin.x = -maxVox;
        vMax.x = maxVox;
    }
    else
    {
        vMin.x = maxVox;
        vMax.x = -maxVox;
    }
    if(normal.y > 0.0)
    {
        vMin.y = -maxVox;
        vMax.y = maxVox;
    }
    else
    {
        vMin.y = maxVox;
        vMax.y = -maxVox;
    }
    if(normal.z > 0.0)
```

```
    {
        vMin.z = -maxVox;
        vMax.z = maxVox;
    }
    else
    {
        vMin.z = maxVox;
        vMax.z = -maxVox;
    }
    if (dot(normal, vMin) + d > 0.0) return false;
    if (dot(normal, vMax) + d >= 0.0) return true;
    return false;
}

bool triBoxOverlap(in vec3 voxCenter, in float voxHalfSize, in
vec3 vertex0, in vec3 vertex1, in vec3 vertex2)
{
    vec3 v0, v1, v2, e0, e1, e2, fe0, fe1, fe2, normal;
    float minValue, maxValue, p0, p1, p2, rad, d;
    v0 = vertex0 - voxCenter;
    v1 = vertex1 - voxCenter;
    v2 = vertex2 - voxCenter;
    e0 = v1 - v0;
    e1 = v2 - v1;
    e2 = v0 - v2;
```

```
fe0 = abs(e0);

// AXISTEST_X01(e0.z, e0.y, fe0.z, fe0.y)

p0 = e0.z * v0.y - e0.y * v0.z;

p2 = e0.z * v2.y - e0.y * v2.z;

if (p0 < p2)

{

    minValue = p0;

    maxValue = p2;

}

else

{

    minValue = p2;

    maxValue = p0;

}

rad = fe0.z * voxHalfSize + fe0.y * voxHalfSize;

if (minValue > rad || maxValue < -rad) return false;

// AXISTEST_Y02(e0.z, e0.x, fe0.z, fe0.x)

p0 = -e0.z * v0.x + e0.x * v0.z;

p2 = -e0.z * v2.x + e0.x * v2.z;

if (p0 < p2)

{

    minValue = p0;

    maxValue = p2;

}

else
```

```
{
    minValue = p2;
    maxValue = p0;
}
rad = fe0.z * voxHalfSize + fe0.x * voxHalfSize;
if (minValue > rad || maxValue < -rad) return false;
// AXISTEST_Z12(e0.y, e0.x, fe0.y, fe0.x)
p1 = e0.y * v1.x - e0.x * v1.y;
p2 = e0.y * v2.x - e0.x * v2.y;
if (p2 < p1)
{
    minValue = p2;
    maxValue = p1;
}
else
{
    minValue = p1;
    maxValue = p2;
}
rad = fe0.y * voxHalfSize + fe0.x * voxHalfSize;
if (minValue > rad || maxValue < -rad) return false;
fe1 = abs(e1);
// AXISTEST_X01(e1.z, e1.y, fe1.z, fe1.y)
p0 = e1.z * v0.y - e1.y * v0.z;
p2 = e1.z * v2.y - e1.y * v2.z;
```

```
if (p0 < p2)
{
    minValue = p0;
    maxValue = p2;
}
else
{
    minValue = p2;
    maxValue = p0;
}
rad = fe1.z * voxHalfSize + fe1.y * voxHalfSize;
if (minValue > rad || maxValue < -rad) return false;
// AXISTEST_Y02(e1.z, e1.x, fe1.z, fe1.x)
p0 = -e1.z * v0.x + e1.x * v0.z;
p2 = -e1.z * v2.x + e1.x * v2.z;
if (p0 < p2)
{
    minValue = p0;
    maxValue = p2;
}
else
{
    minValue = p2;
    maxValue = p0;
}
```

```
rad = fe1.z * voxHalfSize + fe1.x * voxHalfSize;
if (minValue > rad || maxValue < -rad) return false;
// AXISTEST_Z0(e1.y, e1.x, fe1.y, fe1.x)
p0 = e1.y * v0.x - e1.x * v0.y;
p1 = e1.y * v1.x - e1.x * v1.y;
if (p0 < p1)
{
    minValue = p0;
    maxValue = p1;
}
else
{
    minValue = p1;
    maxValue = p0;
}
rad = fe1.y * voxHalfSize + fe1.x * voxHalfSize;
if (minValue > rad || maxValue < -rad) return false;
fe2 = abs(e2);
// AXISTEST_X2(e2.z, e2.y, fe2.z, fe2.y)
p0 = e2.z * v0.y - e2.y * v0.z;
p1 = e2.z * v1.y - e2.y * v1.z;
if (p0 < p1)
{
    minValue = p0;
    maxValue = p1;
```

```
}
else
{
    minValue = p1;
    maxValue = p0;
}
rad = fe2.z * voxHalfSize + fe2.y * voxHalfSize;
if (minValue > rad || maxValue < -rad) return false;
// AXISTEST_Y1(e2.z, e2.x, fe2.z, fe2.x)
p0 = -e2.z * v0.x + e2.x * v0.z;
p1 = -e2.z * v1.x + e2.x * v1.z;
if (p0 < p1)
{
    minValue = p0;
    maxValue = p1;
}
else
{
    minValue = p1;
    maxValue = p0;
}
rad = fe2.z * voxHalfSize + fe2.x * voxHalfSize;
if (minValue > rad || maxValue < -rad) return false;
// AXISTEST_Z12(e2.y, e2.x, fe2.y, fe2.x)
p0 = e2.y * v1.x - e2.x * v1.y;
```

```
    p1 = e2.y * v2.x - e2.x * v2.y;
    if (p0 < p1)
    {
        minValue = p0;
        maxValue = p1;
    }
    else
    {
        minValue = p1;
        maxValue = p0;
    }
    rad = fe2.y * voxHalfSize + fe2.x * voxHalfSize;
    if (minValue > rad || maxValue < -rad) return false;
    // FINDMINMAX(v0.z, v1.z, v2.z, minValue, maxValue)
    minValue = maxValue = v0.z;
    minValue = min(min(minValue, v1.z), v2.z);
    maxValue = max(max(maxValue, v1.z), v2.z);
    if (minValue > voxHalfSize || maxValue < -voxHalfSize) return
false;
    // FINDMINMAX(v0.x, v1.x, v2.x, minValue, maxValue)
    minValue = maxValue = v0.x;
    minValue = min(min(minValue, v1.x), v2.x);
    maxValue = max(max(maxValue, v1.x), v2.x);
    if (minValue > voxHalfSize || maxValue < -voxHalfSize) return
false;
    // FINDMINMAX(v0.y, v1.y, v2.y, minValue, maxValue)
```

```
    minValue = maxValue = v0.y;

    minValue = min(min(minValue, v1.y), v2.y);

    maxValue = max(max(maxValue, v1.y), v2.y);

    if (minValue > voxHalfSize || maxValue < -voxHalfSize) return
false;

    normal = cross(e0, e1);

    d = -dot(normal, v0);

    if (!planeBoxOverlap(normal, d, voxHalfSize) ) return false;

    return true;

}


void voxelizationEngine(void)

{

    vec3 voxCenter;

    float targetDepth, target;

    for (int index = 0; index < int(numRenderTargets); ++index)

    {

        gl_FragData[index] = vec4(0.0);

    }

    voxCenter.xy = origBBox.xy + vec2(-voxelSize, voxelSize) *
gl_FragCoord.xy;

    for (float index = voxDepthRange.x; index <= voxDepthRange.y;
++index)

    {

        voxCenter.z = origBBox.z + voxelSize * (index + 0.5);

        if (triBoxOverlap(voxCenter, halfVoxelSize, vertex0,
vertex1, vertex2) )
```

```
        {
            targetDepth = index * INVDEPTH;
            target = floor(targetDepth);
            if (target >= 0 && target < numRenderTargets)
            {
                gl_FragData[int(target)] += texture1D(bitmask,
fract(targetDepth) );
            }
        }
    }
}
```