Finding Surface Normals From Voxels

Carlos Eduardo Vaisman Muniz, Esteban Walter Gonzalez Clua (advisor)

Instituto de Computação - Universidade Federal Fluminense, R. Passo Da Pátria, 156, 24210-240 Niterói, RJ, Brasil
cmuniz@superig.com.br, esteban@ic.uff.br

Abstract

Volumetric rendering is a way to represent 3D models on devices with a low use of 3D resources. It allows the use of high poly models, quick collision detection and rasterization operations, making them interesting for the games that are being created for the current generation of mobiles and pocket pcs that cannot handle OpenGL operations with a fast frame rate. However, the lack of knowledge of the shape of the object has proven to be a problem to preprocess its lighting. This short paper presents an heuristic that finds an approximation of the direction of normal vector of a voxel without the previous knowledge of the mesh, vertexes, edges and surfaces of the volumetric model where it belongs.

1. Introduction

Volumetric rendering is a technique used to visualize 3D data in a 2D projection [1]. While it is mostly used on medical applications, specially with 3D scanners, it has also been used in games made for devices with low 3D graphics resources, such as mobile phones.

The volumetric models used by these games are 3D grids of pixels, where each of them, called voxel in this article, may have at least one color, one unitary normal vector and a density value. Voxel files uses more disk space than normal geometry, since it stores all used volumetric elements, even when compressed. In order to optimize the disk space, the number of distinct colors and normal vectors might be limited, so a game can hold more models using less space.

While most of the volumetric models are created with 3D modeling programs, voxel editors are useful by featuring a reliable and intuitive way to texturize the model and preview it. They allow users to paint the object pixel by pixel, which is not good to create very complex shapes, specially when the surface normals also need to be painted. Due to lack of voxel editors available and interest on creating them, researches on automatic ways to detect the normals from voxels are rare and are mostly focus on quick ways to preprocess and render the volume, like High Quality Lighting and Efficient Pre-Integration for Volume Rendering [2]. Other works may have simple calculation of the normals as a mean to achieve their objectives, such as Cline et al. [3], where the diagonal neighbor voxels are ignored. Hux et al [4] is able to extract the surface where a voxel belongs to, but it's not clear how normals would be calculated from its results.

The algorithm developed for this work calculates the normal vector of each of voxels by only knowing if a certain pixel is solid or not, with the purpose of making a automatic detection of normals for a voxel editor.

The section 2 explains the algorithm and its implementation. The section 3 shows the results and compare them with the 3D modeler exporters. It also analyzes the limitations of the technique and the voxel file format that was used. Finally, section 4 has the final considerations and section 5 presents the future works.

2. The algorithm

2.1 Overview

The concept behind the algorithm consists in to getting the surface around the voxel that is being analyzed and build a tangent plane. Then, it extracts the normal direction from this plane, as showed on figure 1.

The technique starts by detecting which of the voxels belongs to the surface of the model. After that stage, for each voxel, the surface around it is determined by a spherical region centered in the pixel, that is being analyzed at the moment, with a ray determined by the programmer. It computes the derivative of this region in order to determine a plane that is tangent to this surface. It provide us two normal vectors with the opposite directions. Then, it uses a ray casting to compare the presence of material on both directions and choose the one where that matches the direction of the reflection of the light..

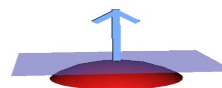

Figure 1 Normals from a tangent plane of a surface (red).

2.2 Finding the surfaces

In order to detect the voxels that are part of the surface, it finds what is inside the object, by using a 3D flood and fill. It can be done by placing the volume grid inside a bigger 3D grid and starting to paint from the position (0,0,0) of this new grid. What remains unpainted is part of the volume of the object. Then, each voxel is scanned and the ones that are neighbor to external elements are part of the surface of the volume. Note that the voxels that are inside the volume and not part of the surface must be marked for the ray casting procedure detailed in the section 2.5.

2.3 Finding the tangent plane

From now on, each volume element is analyzed separately. We cannot find a plane that is tangent to a point, since it could be any plane, but we can find one that can be tangent to a surface composed with the voxels that are around the one being analyzed. This algorithm simplifies the process of finding an ideal surface around this point, by picking the neighbor ones in a sphere, where its ray (range) is determined by the programmer. This may have a minor cost in the final quality of the result, but it speeds up the algorithm considerably.

In order to find a tangent plane of a surface, the algorithm needs to find the two dimensions with the highest variations from it. These two dimensions are “locked” and the last dimension will determine the inclination of the plane. This variation can be found by computing the derivative of the function of this surface in the 3 axis and picking the 2 highest ones. The problem here is that there is no way to know the function of this surface, however the farther a voxel is from the center of a surface, the lower its effect in the normal vector of the center. Assuming this property as function f, (Xp,Yp,Zp) as the coordinates of a variable point of the surface and (Xc,Yc,Zc) as the coordinates of the center of the surface, we have the following function:

$$f(x_p, y_p, z_p) = \frac{1}{\sqrt{(x_c - x_p)^2 + (y_c - y_p)^2 + (z_c - z_p)^2}} \quad \textbf{(1)}$$

After computing the three partial derivatives of the function above, we have the following equations:

$$\frac{\partial f}{\partial x_p} = \frac{(x_c - x_p)}{\left(\sqrt{(x_p - x_c)^2 + (y_p - y_c)^2 + (z_p - z_c)^2}\right)^3} \quad \textbf{(2)}$$

$$\frac{\partial f}{\partial y_p} = \frac{(y_c - y_p)}{\left(\sqrt{(x_p - x_c)^2 + (y_p - y_c)^2 + (z_p - z_c)^2}\right)^3} \quad \textbf{(3)}$$

$$\frac{\partial f}{\partial z_p} = \frac{(z_c - z_p)}{\left(\sqrt{(x_p - x_c)^2 + (y_p - y_c)^2 + (z_p - z_c)^2}\right)^3} \quad \textbf{(4)}$$

Since the sum of the derivatives of each point P is the derivative of the sum of all points [5], the variation of the surface to its center can be found by using the sum of the partial derivatives (2,3 and 4) applied for its all points.

Each one of the four vertices of the tangent plane will be based on the two “locked” dimensions from the previous procedure. Splitting the surface in 4 quarters, based on these two dimensions where the points are in the middle may belong to more than one quarter. The coordinates of each vertex of this plane can be found by using the same derivative computing process, but it will result only the sum of the points located in the quarter where it belongs.

2.4 Finding the right normal.

The tangent plane may have four vertexes, but only 3 are needed to calculate the normal vector by making the cross product of them [6]. The normalized result of this cross product gives one normal vector and, inverting all signs, it is possible to get another one, which can also be the answer, as seen in figure 2.

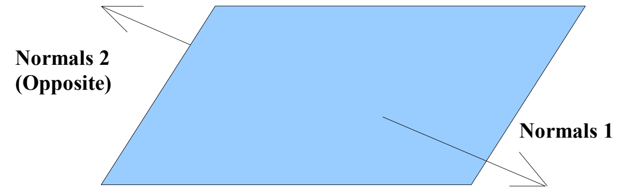

Figure 2 Tangent plane can have two normals.

In order to decide which of the two normals applies to the plane, the algorithm casts two rays from the center of the voxel, one in the direction of each normal, computing the amount of solid volumetric elements intersecting among the path. The direction with less solid materials is the final result.

2.5 Optimizing the algorithm

The partial derivative functions in 2.3 can be done only once and their values can be cached in a 3D grid. The values of the grid can be multiplied by the presence of a solid voxel - represented as 1 - or lack of it, which would be represented with 0.

Some of the processes described above can be multi-threaded, which would considerably speed up the execution of the algorithm on processors with multiple cores.

3. Results

3.1 The tool

The algorithm was tested on Voxel Section Editor III 1.37 [7]. This program was selected because it is open source, it is able to read the voxel file format from Westwood Studios [8], featured in Command & Conquer: Red Alert 2 and Command & Conquer: Tiberian Sun used in this work. It also has a 3D view that can real time preview these models with a free camera.

Both games mentioned above use a 256 color palette and a limited set of normal vectors, 36 for Tiberian Sun and 244 for Red Alert 2. Each model is composed of sections, where each has its own 3D grid with one transformation matrix per frame, an axis aligned bounding box for collision detection and scale settings. Each voxel has one color and one normal vector and, in the editor, it is either solid or empty, with no alpha channel which is why each element of the volume can behaves like a single sided face, since it reflects light to only one direction.

The program renders voxels in a primitive way, by rendering cubes and giving to all its faces the normal and color from the voxel. In order to verify the efficiency of the algorithm, figure 3 shows the option to view only the normals in shades of gray that tends to white when normal points to the camera and black when it doesn't reach it.

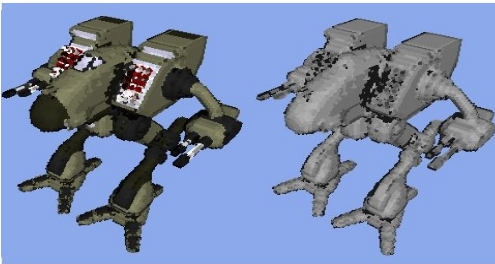

Figure 3 Colors at the left side and normals at the right.

More than 30 models were tested. Many of them were distributed by Westwood Studios, used for the commercial game “Red Alert 2”. Other models were taken from Red Alert 2 fan sites. The models analyzed had a minimum of 5000 filled voxels. The Kirov in figure 4 is the most complex model tested with 35584.

Figure 4 Kirov in Red Alert 2 and in the VXLSE III

3.2 Model data loss and distortion

The algorithm is optimized to find only one surface normal per face. It totally ignores the original geometry of the model and the normal vector found for each voxel is only based on its opaque neighborhood, which means that each voxel is analyzed locally only. This results in a distortion between the original models, obtained from Westwood Studios, and the same models with normals calculated by Voxel Section Editor III, specially when the original 3D model had its normals affected by texture mapping. This distortion is showed on figures 5.

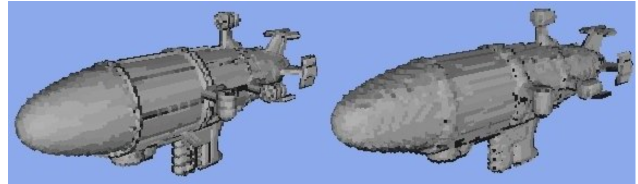

Figure 5 Kirov: Westwood and VXLSE III respectively.

Despite the limitations previously discussed, voxels done with the voxel editor had acceptable results, as seen on figure 6. On all models from the figures above, the range of the neighborhood region is close to 3.54. Values lower than 2 resulted on a great amount of noise. Values above 5 usually get smoother results than it should, with not so much defined borders and, in some cases, there are some additional distortions. The figure 7 shows the Kirov auto normalized with ranges 1.69 and 5.2 respectively.

Figure 6 Models made with VXLSE III and auto normals algorithm from this work.

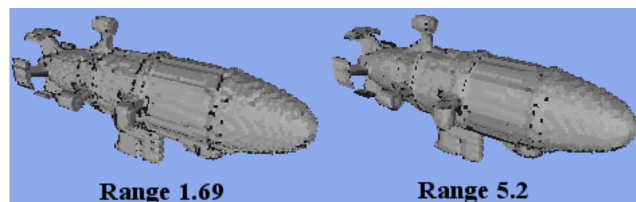

Figure 7 A Kirov with a noised front at the left and a smoother Kirov at the right side.

3.3 Other limitations of the presented models

The limit of only one normal per voxel prevents the algorithm from finding boxes. 3D exporters bypass this problem by choosing one of the normals of the faces where the corner belongs, while this technique tends to find the average of the normals of these faces.

There is no correct normal value for isolated voxels or voxels that can reflect light to opposite directions. However, unlike 3D exporters, the technique showed in this article may not necessarily choose the same directions for the voxels that belongs to the same wall that reflects lights for two opposite directions.

Surfaces with discontinued functions will have their results distorted, since the detection of the tangent plane vertices, that uses the derivative computation, will fail. It is not possible to compute derivatives of discontinued functions [9]. This problem appears on some black dots from the Kirov (figure 5) and some corners from the Lybian demo truck (figure 6).

3.4 Complexity

The range is what most affects the complexity of this algorithm. The operation with the highest complexity is the calculation of the 4 vertexes of the tangent plane. Assuming m as the range and n as the number of voxels, the complexity of this procedure is $O(m,n) = m^3 \times n$. During the tests, Kirov airship with range 3.54 took less than 5 seconds, to execute while using range 100 took over an hour.

4. Final Considerations

This heuristic may not reach perfect results due to the discontinuity problems, but it achieves the objective of finding high quality normals with a quick speed. Some of the texturing effects lost in the voxel conversion process can only be repainted by the user. Most of the errors are hard to be noticed on low resolution, which makes voxels normalized with this algorithm useful for games.

The source code of this work is available in the version 1.37 of the Voxel Section Editor III or later [10].

5. Future Works

The only known way to fix the discontinuity issue is to find all vertexes, connections and surfaces of the volumetric model. However, doing this, the whole process described at this article will change and the complexity of the operation will increase considerably. We plan to research a way to do this with the minimum cost possible.

Acknowledgments

Command & Conquer, Tiberian Sun, Red Alert 2, Westwood Studios and the voxel format used in this experience are property of Electronic Arts. While this research may violate the end user license agreement of these games, Electronic Arts do support modifications of their games and the creation of modding tools by fans, both for non commercial purposes.

References

[1] W. E. Lorensen, H. E. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm"; Computer Graphics; Siggraph '87 Conference Proceedings; Jul. 27-31, 1987; vol. 21, No. 4; ACM Siggraph; pp. 163-169.

[2] E. B. Lum, B. Wilson, K. L. Ma, “High Quality Lighting and Efficient Pre-Integration for Volume Rendering”, In EUROGRAPHICS/IEEE Symposium on Visualization (2004), pp. 25-34.

[3] H. E. Cline, S. Ludke, (2000) “Fast method of creating 3D surfaces by stretching cubes”, retrieved August 28th, 2007 from http://www.freepatentsonline.com/6115048.html

[4] W. A. Hux, (2005) “Generating Sirface Normals”, retrieved August 28th, 2007 from http://www.patentstorm.us/patents/6867773-fulltext.html

[5] H. Anton, “Calculus: A New Horizon”, *John Wiley & Sons Inc* 6 Sub edition (1998), 332--335.

[6] E. Azevedo and A. Conci, “Computação Gráfica - Teoria E Prática”, *Elsevier* (2003), 266--267.

[7] Voxel Section Editor III (2007), “Project Perfect Mod”, retrieved July 1st, 2007 from http://www.ppmsite.com/index.php?go=vxlseinfo

[8] EoL and DMZ, (2000) “XCC Home Page by Olaf Van der Spek”, retrieved July 1st, 2007 from http://xhp.xwis.net/documents/VXL_Format.txt.

[9] H. Anton, “Calculus: A New Horizon”, *John Wiley & Sons Inc* 6 Sub edition (1998), 342—345.

[10] Open Source Voxel Tools Subversion, “Project Perfect Mod”, retrieved August 27th, 2007 from http://svn.ppmsite.com/