

Semiautomatic Shader Code Generation for Rendering Voxelized Polygonal Models

Jan Fischer¹, David Whittaker¹, Aaron Lefohn², Bruce Gooch¹

¹ University of Victoria, Canada ² Intel Corp., Seattle, USA

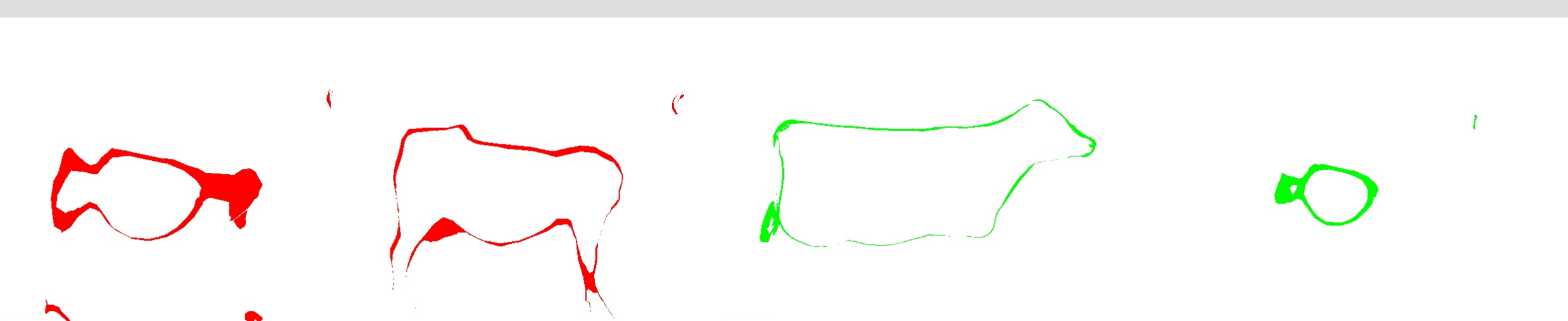
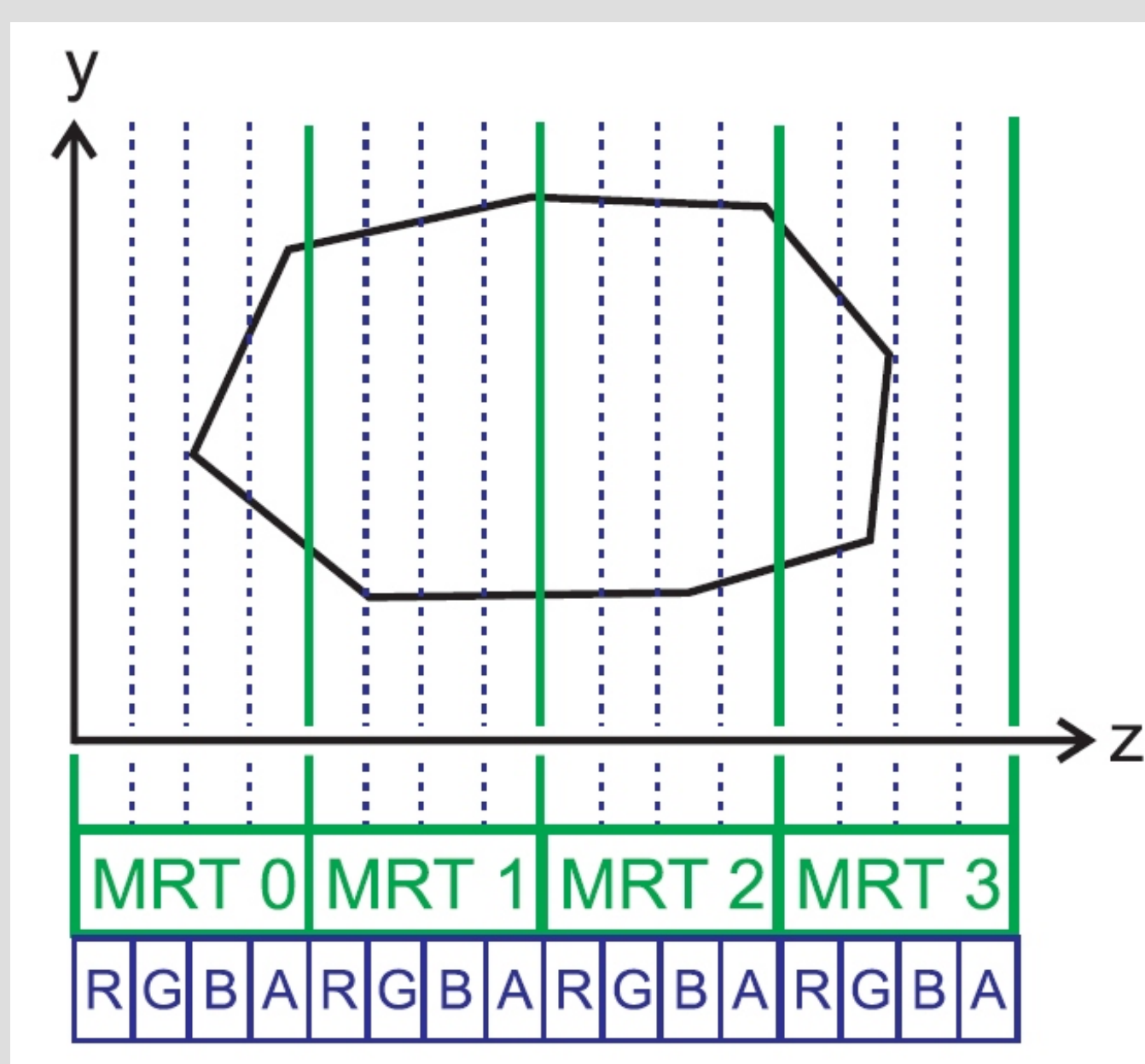
Abstract

The conversion of polygonal models into a volumetric representation is an important basic operation for many applications in computer graphics and related areas. Here, we present a voxelization method for general, complex models, which generates a high-resolution volume in a single pass. The entire process is executed on the GPU and is capable of voxelizing animated scenes in real time. One interesting application is the simultaneous display of all layers of a model, e.g., in illustrative visualization. We demonstrate the use of voxelization data for rendering the total solid depth as well as depth-attenuated silhouettes for all parts of an object. Due to the representation of the voxelization data as bitstrings encoded in multiple textures, writing rendering code based on it can be a tedious task. We therefore, present an automated code generation technique that abstracts and encapsulates binary data access. This allows us to quickly write programs that generate renderings based on voxel data; we call these “voxel shaders”.

Voxelization

We present a high-resolution single-pass voxelization approach similar to the systems described in [Dong et al. 2004] and [Eisemann and Decoret 2006]. In its simplest form, this voxelization generates a binary volume storing only occupancy information (i.e., whether a voxel is intersected by any primitive). For the vertices of all model primitives, we determine the transformed vertex positions in world coordinates. These global z coordinates are then linearly interpolated over the 2D area of the current primitive p during rasterization, yielding an interpolated fragment depth $z_p(x,y)$. The depth range containing the polygonal model is subdivided into several volume slabs consisting of a certain number of slices each. (Without loss of generality, we assume here that the entire polygonal scene has been translated and scaled to fit into the volume containing the resulting voxelization.)

Our system uses four render targets, each containing an RGBA framebuffer. The binary occupancy values of the individual voxels are encoded as bits in the color components of each render target. Each volume slab contains as many slices as there are usable bits in a color component. Subdivision of the depth range into individual render targets and color components is illustrated here:



Four slices of the voxelization of a cow model. Red: front-facing voxels. Green: back-facing voxels.

DONG, Z., CHEN, W., BAO, H., ZHANG, H., AND PENG, Q. 2004. Realtime Voxelization for Complex Polygonal Models. In *Proc. of ACM Pacific Conference on Computer Graphics and Application*, 43–50.

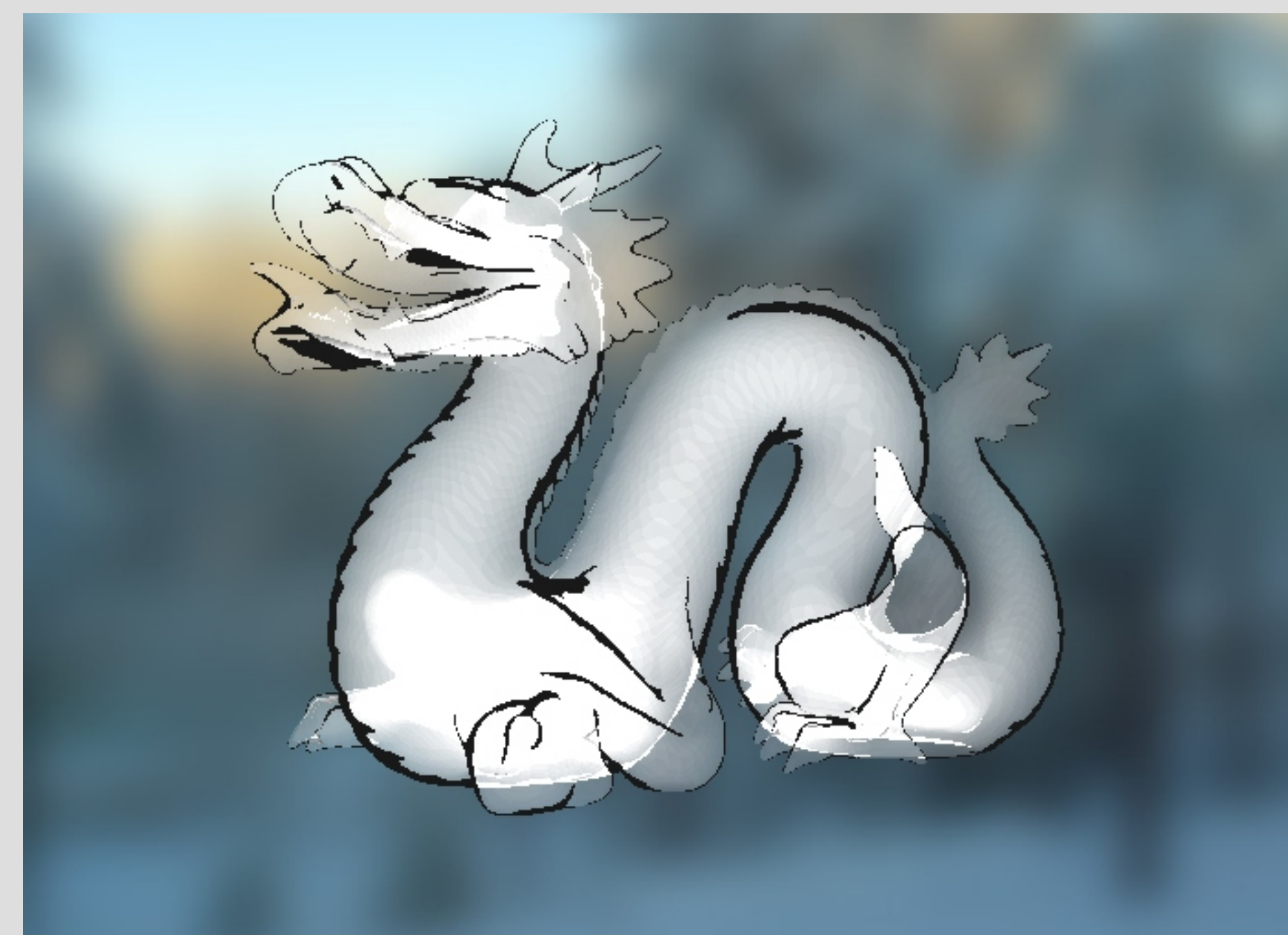
EISEMANN, E., AND DECORET, X. 2006. Fast Scene Voxelization and Applications. In *Proc. of ACM Symposium on Interactive 3D Graphics*, 71–78.



Due to the high voxelization performance, per-frame voxelization of an animated dataset is possible. In this example, the robot model is rendered with a walking animation from a constant viewpoint. Voxelization slices for four example frames from the animation are shown in the panels on the right hand side (all panels show the same slice index).

Voxel-based Illustrative Visualization

We describe an illustrative visualization method based on the computed voxelization. The output images are composed of a display of the total solid depth augmented with approximated silhouettes. Thanks to the voxelization of the complete model, inner structures are automatically visualized. The combined per-frame voxelization and rendering process is capable of generating images in real time.



Automated Rendering Code Generation

In order to simplify the software engineering aspect of working with voxelization data, we propose the use of a shader code generation technique. Our shader code generator pre-processes an HLSL fragment shader, which has been augmented with some additional keywords. These keywords, which we have chosen to have an XML-like look, are replaced by the corresponding full HLSL code. The following code fragment shows the core section of our combined total solid depth and silhouette visualization shader.

```
<VOLUME>
  <SEGMENTCOUNT>8</SEGMENTCOUNT>
  <BITSPERSEGMENT>22</BITSPERSEGMENT>
  <VOLUMECOUNT>2</VOLUMECOUNT>

  numFrontFaces += V1FRONT_SLICE(i);

  innerSlices += step(0.1, numFrontFaces);

  numFrontFaces -= V1BACK_SLICE(i);

  if(edgeAtten == 0 && V2FRONT_SLICE(i)){
    edgeAtten = innerSlices;
  }
</VOLUME>
```