Real-Time Voxelization of Triangle Meshes on the GPU

Ignacio Llamas *
NVIDIA Corporation

Abstract

We present two algorithms to generate volumetric representations of triangle meshes on the GPU. Because all the data is generated in video memory, the algorithms are particularly useful for GPU-based physical simulation. We demonstrate their application to fluid simulation with a real-time smoke simulation where dynamic obstacles (skinned meshes) drive the fluid motion. The first algorithm provides an inside-outside representation of the triangle mesh, while the second one provides interpolated attributes (such as velocity) for those voxels intersected by the boundary of the mesh. Both of these algorithms are efficient on current graphics hardware allowing their use in real-time applications such as video games.

1 Introduction

Recent work has showed voxelization can be performed on graphics hardware to obtain both boundary [Fang and Cheng 2000], [Eisemann and Decoret 2006] and solid [Fang and Cheng 2000] volumetric representations. For fluid simulation purposes we usually need both solid and boundary voxelizations to know whether a cell in the simulation domain is inside or outside an obstacle and what velocity the obstacle has at its boundary. While solid voxelization is not new, to the best of our knowledge, no previously published work offers a solution to obtain a volumetric representation of boundary attributes (such as velocity). We present two algorithms that work at real-time rates in current GPUs and generate the desired information.

2 Inside-outside Voxelization

The first algorithm is inspired by the stencil shadow volumes algorithm and is very similar to that described in [Fang and Cheng 2000]. The idea is simple: we render the input triangle mesh once into each slice of the destination 3D texture using an orthogonal projection. The far clip plane is set at infinity and the near plane matches the position of the slice that we are rendering into. When drawing this geometry we use a stencil buffer (of the same dimensions as the slice) that is initialized to zero. We set the stencil operations to increment for back faces and decrement for front faces (with wrapping in both cases). The result is that any voxel inside the mesh gets a non-zero stencil value. We do a final pass to resolve the stencil buffer into the final obstacle texture. Note that because this method depends on having one back face for every front face, it is best suited to watertight closed meshes, where the interior is well-defined.

3 Boundary Attribute Voxelization

The second voxelization algorithm is used to compute interpolated per-vertex attributes, such as velocity, at each grid cell that contains part of the objects boundary. First, however, we need to know the attribute at each vertex. As with the inside-outside voxelization, the mesh is rendered once for each slice of the grid. This time, however, we must determine the intersection of each triangle with the current slice. The intersection between a slice and a triangle may be a triangle, a segment, a point, or empty. If the intersection is a segment, we draw a thickened version of the segment into the slice using a quad. This quad consists of the two endpoints of the original segment and two additional points offset from these endpoints (Figure 1). The offset distance $w$ is equal to the diagonal length of one texel in a slice of the 3D texture, and the offset direction is the projection of the triangles normals onto the slice.

These quads can be generated using a geometry shader (a new programmable pipeline stage in the latest generation of graphics hardware). This geometry shader takes as input a mesh triangle at a time, producing four vertices if the intersection is a segment and zero vertices otherwise. Since geometry shaders cannot output quads, we use a two-triangle strip. To compute the triangle-slice intersection we intersect each triangle’s edge with the slice. If exactly two edge-slice intersections are found, the corresponding intersection points are used as endpoints for our segment. Using linear interpolation we determine the attribute values at each intersection point and assign them to the corresponding vertices of the quad. When the quad is rasterized, these values get interpolated across the grid cells as desired.

References
