Volume-Surface Collision Detection

J. Ortegano[1], H. Navarro[1] and R. Carmona[1]

[1]Centro de Computación Gráfica. Universidad Central de Venezuela

ABSTRACT

The presence or not of collisions between objects is usually required to study the interaction between them, increasing the realism in virtual environments. Collision detection between polygonal objects has been widely studied, and more recently some studies have been made concerning collisions between volume objects. Collision detection between volume datasets and polygonal objects is introduced in this work. This kind of mixed scenes appears naturally in many applications such as surgery simulation and volume edition. To detect the collision, first the volume dataset is represented by a single 3D texture. Then, a mapping from eye space to volume space is established, such as each mesh fragment has a 3D texture coordinate. The collision is verified by fragment during the rasterization stage. We use OpenGL® occlusion query extension to count the number of mesh fragments colliding with the volume. Our tests show that up to 3800 *pairs of volume-mesh may be evaluated in one second.*

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual reality

1. Introduction

Collision detection is a common problem in computer graphics applications, including computer animation, virtual reality, and computational geometry. The goal of collision detection is to determine the presence or not of intersections between objects. The response to the collision is commonly a separated problem, referred as collision response [Bou01], which is related to physic simulation; for example, the objects should not pass through each other. Since the scene may contain hundreds of objects with thousands of polygons, checking collisions between all of them ($O(N^2)$) is not acceptable for real-time walkthrough, where the interactivity imposes a fundamental constraint on the system. Researchers have focused on reducing the time required for detecting collisions by representing the scene (defined by meshes or NURBs) in a hierarchical way [GLM96]. These representations are based on space subdivision [FKN80] (e.g. binary space partition trees) and hierarchical bounding volumes. More recent work has considered the collision between volume datasets [BF03]. In this case, the scene is sliced from the near clipping plane to the far clipping plane, testing the collision on each of the slices, which can be defined as an image space approach.

While these researches are focused on polygonal meshes or volumes, none of them consider a mixed scene, composed of volumetric datasets and polygonal meshes. The possibility of representing volumes and surfaces in the same scene is a consequence of the current popularity of graphics hardware supporting 3D texturing. Virtual surgery, medical planning and volume edition are only a few examples of applications which require the interaction of volumetric datasets with virtual objects represented commonly by polygonal meshes. Currently a virtual surgery room is being developed at Universidad Central de Venezuela, with the goal of simulating knee surgeries. This kind of application requires mixed data because the bones and muscles are volumetric data from CT scans, while the surgical instruments are represented as meshes. This imposes the need for special collision detection techniques that can handle the interactions between volumes and meshes. For this work we will only consider collisions between one mesh and one volume, because in our application (virtual surgery) the user will always handle one instrument (mesh) which will interact with the volumetric data.

In this work we introduce a hardware-accelerated technique for collision detection between volumes and meshes. To the best of our knowledge, this is the first approach for surface-volume collision detection in the literature. It consists in defining a mapping between the volume space (also referred as texture space) and the eye space, such as a texture coordinate can be assigned to each vertex of the mesh, and thus, to each fragment during rasterization. A collision exists if a mesh texel is not transparent. This process can be efficiently achieved by combining alpha testing and occlusion query [CG03]. Since our method is hardware-accelerated, it can test up to 3800 volume-surface collisions per second. Another benefit of our method is that it does not depend on the size of the volume, but the size of the mesh. This is an

advantage because usually the mesh is used as a tool for interaction with the volume, and does not require high details.

2. Prior Works

Hierarchical decomposition of the scene and objects has been widely used to reduce the number of pairs for intersection testing. Space subdivision and hierarchical bounding volumes are used to create a hierarchical representation of the scene. The goal of such hierarchies is to approximate the mesh topology on different levels of detail, usually by using simple polyhedra, leading to faster intersection tests.

BSPs and octrees are common examples of techniques based on space subdivision. BSP trees [FKN80] are mostly used for partitioning the static part of the scene. Each node of the BSP tree contains a single plane which divides the scene in two: one part located below the plane, and the other one located above it. Each half-space can be divided recursively in two, using the same principle. Different criterion can be used for selecting each plane; a basic approach chooses the plane which splits the largest axis of the axis aligned bounding box of the remaining scene. A more accurate approach may consider the difference in number of triangles of both sides and the number of polygons (or objects) intersected with the plane [CKN02]. On the other hand, octrees [WVG92] divide the scene in eight equal boxes. Each of those boxes are then recursively subdivided until all the data inside of the box is uniform, and no more subdivisions are required.

As for bounding volumes, several approaches exist. There have been works on using bounding spheres [Lar08], axis aligned bounding boxes [vdB98] (ABBs), oriented bounding boxes [GLM96] (OBBs) and discrete oriented polytopes [KHM*98] (K-DOPs). As the complexity of the bounding volume increases, it fits better the shape of the mesh, but requires more time to build. There has been also some research on using hierarchies of bounding volumes, such as [CWK10], where OBBs and bounding spheres are used together, using first the bounding sphere for quickly discarding collisions and then OBBs when the objects are closer. The selection of the best bounding volume depends on the shape of the objects that compose the scene, so there is no "magic bounding volume" which can be used for every application.

In some complex scenes there might be hundreds of objects interacting in the scene. In such cases, a broad collision detection phase can be applied as suggested by Cohen et al. [CLMP95] in 1995 (this phase is known as sweep and prune). The sweep and prune algorithm discards collisions between objects which are not close enough, using the AABBs of the objects for quickly discarding pairs of not colliding objects.

Another approach is to keep track of the distance of the objects that collide. In 1997, Mirtich [Mir98] proposed a method that keeps track of the closest pair of features of two convex polyhedra (a feature can be either a vertex, an edge or a face of the polyhedra). Each time that one of the objects changes position or orientation, the closest pair of features is updated very fast, based on the fact that the objects are convex. For not convex objects a decomposition needs to be done in order to transform the object into a set of convex objects.

In 2003 Govindaraju et al. [GRLM03] proposed a method which uses the GPU for collision detection in image space. The method first discards which objects or parts of objects are not colliding in image space. If two different objects can be drawn without any interference (the fragments of one object do not overlap fragments of the other object) then a collision can be discarded. When the objects overlap in image space, then it is not possible to discard a collision and another approach is necessary. Based on this, a Potentially Colliding Set (PCS) is built which contains the objects which might be colliding, discarding objects that do not need to be tested. The same process can be applied on parts of the object in order to find out which polygons of the meshes might be colliding with other ones, discarding the parts of objects which are not colliding. This approach has been enhanced in order to make the collision detection faster and allow self-collisions as shown by Govindaraju in 2007 [GKLM07]. Also in 2007 Jang et al. [JJH07] proposed another image space method which computes the PCS on the triangle level, having the advantage of not requiring any preprocessing of the data. In 2008, Hanyoung et al. proposed another method [HJ08] where AABBs are first used in order to prune groups of triangles not colliding, and then PCSs are computed using the GPU for those pairs of AABBs that collide.

Recently works have focused on the design of hardware with specific operations that support collision detection. In 2006 Raabe et al. [RHAZ06] proposed and implemented an architecture to detect collisions using hierarchical data structures that support DOPs directly in hardware. In his PhD thesis Raabe shows hardware implementation of low level triangle-triangle collision detections [Raa08].

Recent development and research on volume rendering techniques make it necessary to develop special techniques for manipulating collisions between volumes. The main problem is that volumes do not represent explicitly the surface of the model, so classic approaches cannot be applied.

Kaufman et al. in 1997 [HK97] proposed a probabilistic method for handling collisions between volumetric objects. The interaction between two volumes is described by finding the intersecting regions between the volumes. A hierarchical approach is used to be able to detect collisions faster, discarding regions of objects that are not colliding.

In 2003 Boyles and Fang [BF03] proposed a technique centered on volume rendering using 3D textures [WVW94], where a proxy mesh is used to display the volume. This method is novel because it does not follow the classic collision detection scheme based on hierarchical bounding volumes, and explodes the frame buffer of the GPU. It is a very fast technique which requires very low extra memory. When rendering volumes using the 3D textures technique, the volume is sliced with polygons from the near clipping plane to the far clipping plane. This polygons are texturized loading the volume as a 3D texture (and possibly modifying the color with a transfer function). In Boyles approach a rendering of the scene is done following this slicing scheme, but different colors are assigned to each object. Fragments are written in the frame buffer using a bitwise `OR` as blending function that will force a white fragment if two different voxels are on the same position. Finally, detecting a collision is reduced to checking if there is a white fragment on the frame buffer.

Our work is inspired in Boyles approach, performing the collision detection in image space. However, instead of slicing the volume with a proxy geometry, the mesh is texturized with the volume itself to detect collisions in the GPU. In the next section our method is detailed.

3. Volume-Surface Collision Detection in Texture Space

Our method for volume-surface collision is an image-space approach. It consists in texturizing the polygons of the mesh with the volume, detecting the collision in the fragment shader. The approach assigns a 3D texture coordinate to each mesh vertex via mapping between object space and texture space. The fragment shader receives an interpolated 3D texture coordinate to evaluate if the fragment is texturized with an opaque voxel, which means, the polygon collides with the volume.

To assign a texture coordinate for each mesh vertex, the vertex is first transformed from object space into volume space, where each point of this space represents a 3D texture coordinate (see Fig. 1). Let M be the model-view matrix which transforms a mesh vertex from object space to eye space; also, let T be the texture matrix which transforms the volume from texture space into eye space. The 3D texture coordinate $c(s,t,r,1)$ of a mesh vertex $v(x,y,z,1)$ can be obtained from $c = T^{-1}Mv$. Notice that the matrix $T^{-1}M$ is constant for a given frame.

The texture coordinates of mesh vertexes are interpolated during rasterization. Thus, the fragment shader receives the mesh fragment with its corresponding 3D texture coordinates. The fragment shader first validates the texture coordinate, verifying if it belongs to the texture domain; i.e. $0 \leq s,t,r \leq 1$. If (s,t,r) is outside the texture domain, the fragment is discarded. Otherwise, we assign the fragment opacity according to the transfer function. The fragment can be discarded if it is considered transparent, i.e. the opacity is lower than an alpha threshold. Thus, we implement the alpha test in the fragment shader. Using this pipeline configuration, the collision occurs if any mesh fragment is not discarded. Since this configuration is only used for collision detection (and not for generating an output image), the color-buffer and z-buffer are disabled during this process.

OpenGL® provides a mechanism to count the number of fragments which pass the tests of the last pipeline stage (per fragment operators, e.g. z-test, alpha test and stencil test). This mechanism is the oclussion query OpenGL® extension [CG03], specifically named `ARB_oclussion_query`. We do not apply any per fragment operator, since we are interested in simply counting the fragments produced by the fragment shader. The number of fragments can be counted after processing each single polygon (collision per polygon), of after processing all polygons of a mesh (collision per surface). Fig. 2 shows in grayscale the opacity of the fragments which collide with a large triangle. Blue pixels represent the fragments discarded by the fragment shader. They are not counted by the occlusion query extension.

4. Limitations

Since the method depends of the polygon projection into the image space, the rasterization module may generate few fragments (or zero) when the polygon is perpendicular to the viewport. Fig. 3 shows how the same polygon generates a different number of fragments depending on its orientation. In case (c) the number of generated fragments is very low, and in some cases it can even be zero. In such a case, the collision detection may fail for this polygon. A possible workaround for this problem is testing for collisions from two orthogonal views. If a polygon was originally perpendicular to the view (generating zero fragments) then when the orthogonal view is computed the same polygon will not be perpendicular to the view, ensuring that it will generate fragments.

5. Implementation

The implementation in GPU needs a vertex shader and a fragment shader which are shown in Listing 1:

```
// VERTEX SHADER
void main()
{
 // surface verteces in eye space
 gl_TexCoord[0] = gl_ModelViewMatrix *
                  gl_Vertex;
 // vertex projection
 gl_Position    = gl_ProjectionMatrix *
                  gl_TexCoord[0];
 // surface vertexes in texture space
 gl_TexCoord[0] =
          gl_TextureMatrixInverse[0] *
          gl_TexCoord[0];
}

// FRAGMENT SHADER
// 3D texture identifier for the volume
uniform sampler3D volume;
// precomputed integrals table
uniform sampler1D transfer_function;

void main()
{
// validate texture coordinates:
 if (gl_TexCoord[0][0] < 0.0 ||
     gl_TexCoord[0][1] < 0.0 ||
     gl_TexCoord[0][2] < 0.0 ||
     gl_TexCoord[0][0] > 1.0 ||
     gl_TexCoord[0][1] > 1.0 ||
     gl_TexCoord[0][2] > 1.0)
   discard;

// Obtain fragment opacity for testing
 if (texture1D(transfer_function,
     texture3D(volume,
     gl_TexCoord[0].xyz).r ).a < 0.1)
       discard;

// Eureka! Collision detected
 gl_FragColor = vec4(0.0, 0.0, 0.0, 1.0);
}
```

Listing 1: Vertex and fragment shaders for our method

For volume rendering, we are using GPU based Ray Casting [KW03] with early ray termination. Thus, the front-faced polygons of the bounding box are rasterized, and each fragment represents the entry point of a ray into the volume. We define the texture matrix T as the transformation from texture space into eye space. With this matrix, the unitary bounding box of the volume in texture space is transformed

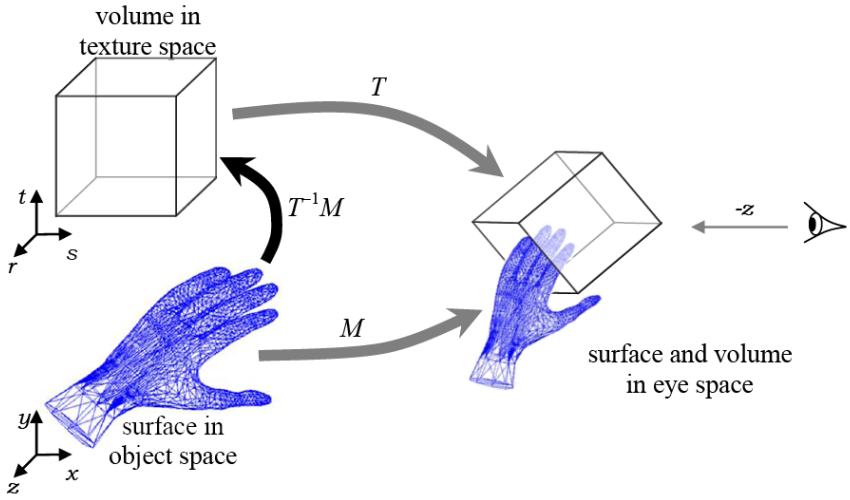

Figure 1: *Transformation between spaces*

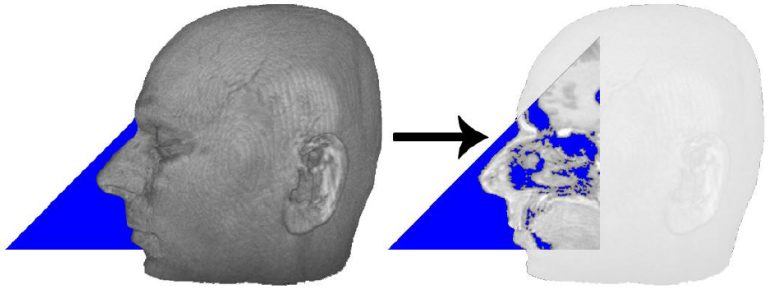

Figure 2: *Volume-polygon collision. (a) Rendering of volume and polygon colliding in eye space. (b) Polygon fragments colliding with the volume*

into eye space. The back-faced bounding-box polygons are culled in eye space, and the front faces are rasterized, interpolating the texture coordinates of its vertexes. The fragment program receives the interpolated texture coordinates, which represent the entry point of one ray into the volume. The eye position is transformed to texture space, using the inverse texture matrix T^{-1}. With the eye and the entry point in texture space, a ray can be casted to compute the volume rendering integral. We use pre-integrated classification to enhance the rendering quality as in [EKE01]. A fast pre-integration 2D table is computed in real time (about 0.04 seconds) to keep the interactivity when the user is editing the transfer function [LWlM04].

The rendering of the mesh uses the fixed OpenGL® pipeline, with Gouraud Shading. For collision detection, the programmable OpenGL® functionality is used as shown before on the shaders. In this context, the mesh is rendered in its simplest form; thus, the lighthing is disable and the normals are not send to the graphics card. Each polygon vertex is mapped into texture space by using the T^{-1} transform. During rasterization, the vertex 3D texture coordinates are interpolated per fragment through the field `gl_TexCoord[0]`. To classify the volume sample, we simply fetch a 1D quantized transfer function. The fragment is discarded if the corresponding texture coordinates are out the interval $[0,1]$ or if the classified sample is transparent according to the alpha threshold.

As a possible optimization, the "if" statement of fragment shader can be replaced by a simple texture clamping if the volume boundary voxels are forced to be transparent. For medical data, the boundary voxels can be replaced by air (zero), which is commonly mapped to 100% transparent voxel via the transfer function. Thus, when a texture coordinate is out the interval $[0,1]$, it is clamped to $[0,1]$, and the corresponding texel (zero) would be always discarded later in the fragment shader, since it is completely transparent after classification.

6. Tests and Results

We implemented a system in Visual C++ for Windows XP to validate our approach between just one volume and one mesh. For multiple volumes and multiple meshes a sweep and prune approach [CLMP95] can be used to determine the potential pair of volume-mesh collisions. We use the OpenGL® library for rendering, and GLUT to manage the user interaction.

Our method was tested on two different hardware environments. The first environment is a PC with Intel Core 2 Quad processor of 2.4 GHz each, 3.25 GB of main memory and Nvidia Quadro FX 3700 graphics board with 640MB of memory (Windows XP, 32 bits). The second environment is a PC with Intel Core i3 processor of 3.07 GHz, 4 GB of main memory and an NVidia GeForce GTX 470 graphics

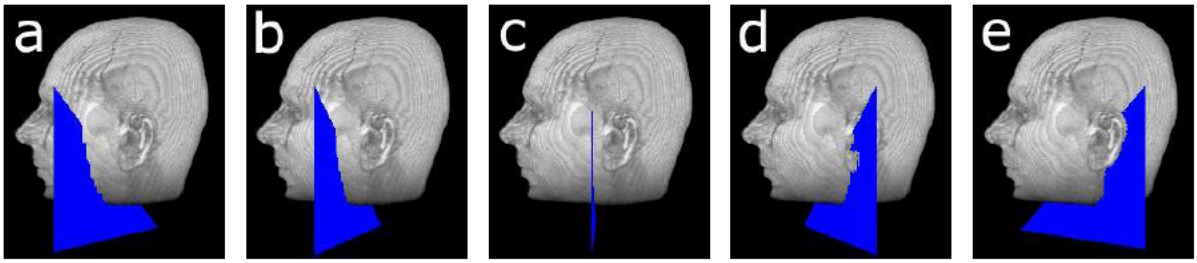

Figure 3: *Limitation of the method when a polygon is $\perp$ to the viewport, in image (C) a few fragments are generated. In some cases no fragments are generated*

board with 1280 MB of memory (Windows 7, 64 bits). In both cases the application was compiled for 32 bits.

We combine two volumes (v_1 and v_2) and 2 meshes (s_1 and s_2) in our tests (see Fig. 4), generating four scenes of one volume and one mesh: v_1s_1, v_1s_2, v_2s_1, v_2s_2. The volumes contain 256^3 8 bits samples (v_1) and $256^2 \times 128$ 8 bits samples (v_2), while the meshes contain 3068 triangles and 2894 vertexes (s_1), and 49998 triangles and 25001 vertexes (s_2). Notice that v_1 doubles the size of v_2, while v_2 is quite sparse. Screenshots of the four scenes are shown on Fig. 5.

Tables 1 and 2 show the average time required to render each scene on the environment #1 and #2 respectively. The first column shows the description of the scene. The second column shows the rendering time for the mesh. The third column shows the rendering time for the GPU-based volume ray casting. The fourth column shows the collision detection time. The fifth column shows the total rendering time (mesh and volume). The sixth column shows the total time (combining rendering and collision), and the last column shows the collision detection overhead on the rendering time. The rendering time is increased in between 8.6% and 14.2% by enabling collision detection in environment 1, and between 4.3 and 15.9 percent in environment 2. However, the rendering still performs in real-time ($\gg$ 30 fps). Also notice that the collision time takes between $0.481\,ms$ and $0.79\,ms$ in environment 1, and between $0.26\,ms$ and $0.338\,ms$ in environment 2. It follows that between 1123 and 2079 surface-volume pairs can be tested for collision in one second (environment 1) and between 2958 and 3846 for environment 2. The rendering time is dominated by the GPU-based ray casting, as can be seen in Fig. 6. There are several factors that influence the difference between collision detection time and rendering time:

- In collision detections an extra computation is required on each fragment, because two textures are sampled (the 3D texture and the transfer function). For rendering meshes, we are using Gouraud shading for lighthing, without texturing. Therefore, the illumination model (Blinn) is computed per vertex, requiring extra computation and loading normals into the GPU. In environment #1 the computation of the illumination model is faster than sampling the two textures, but in environment #2 the mesh takes more time to render (due to the computation of the illumination).
- Depending on the number of vertexes to process and to send to the GPU, as well as the number of fragments to process, the relationship between mesh rendering time and collision detection times varies.
- Environment #2 has very slow mesh rendering compared to environment #1, but in environment #2 the volume rendering is faster than the mesh rendering, thus having better results when we add up the mesh rendering time, volume rendering time and the collision detection time. The slow mesh rendering in environment #2 is probably due to lighting computation because for collision detection there is no lighting computation and the rendering times are not as slow.

A comparison with other methods cannot be done since this is the first study of a method for detecting collisions between volume datasets and meshes.

The proposed method indicates if the collision exists, but does not indicate where the collision occurs. To get a more precise location of the colliding area, the naive approach would test the occlusion query polygon by polygon. However, in our tests, the collision detection takes too long for meshes with many polygons, up to 0.7 seconds per collision, for environment #1, since the occlusion query blocks the CPU (and the loading of polygons) until the GPU flushes its pipeline and sends back the result, causing a bottleneck. To efficiently reduce the number of queries, an OBB tree can be used to represent hierarchically the mesh. Thus, many polygons can be discarded if a high level OBB is outside the volume. For OBBs intersected with the volume bounding box, the algorithm performs recursively for each child node, until a leaf node colliding with the volume bounding box is met. The occlusion query can be performed on the set of triangles of each leaf node instead of individual polygons for speed.

7. Conclusions and Future Works

We introduced an image-based approach for volume-surface collision detection. Establishing a mapping between the surface space and the texture space, the collision detection performs in the fragment processors by checking the fragment texture coordinate and its opacity. If more than one texturized fragment is opaque, the collision occurs. In our tests, each collision detection takes between 0.262 and 0.789 milliseconds. Thus, up to four thousand volume-surface pairs may be tested for collision per second, in our hardware platform.

Our collision detection approach introduces an overhead between 4.3% and 15.9% in the response time. While the overhead is related to the rendering time, it mainly depends on the number of fragments generated by the mesh, as well as on the number of mesh vertexes.

The proposed method indicates if the collision occurs, but

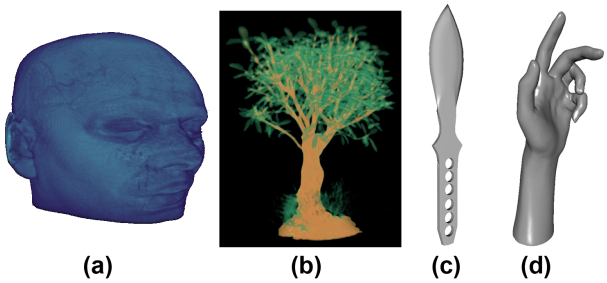

Figure 4: *Test datasets. (a) MRHead with* 256^3 *8 bits samples. (b) MRBonsai with* $256^2 \times 128$ *8 bits samples. (c) Mesh with* 3068 *triangles and* 2894 *vertexes (d) Mesh with* 49998 *triangles and* 25001 *vertexes*

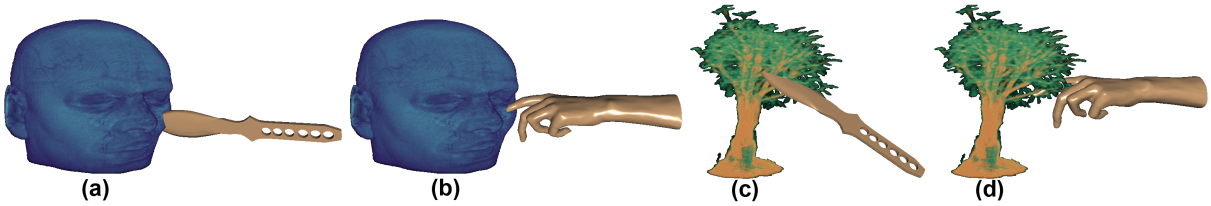

Figure 5: *The four scenes used for testing our method. (a) is* v_1s_1 *(b) is* v_1s_2 *(c) is* v_2s_1 *and (d) is* v_2s_2

does not indicate where it occurs. The method can be enhanced with hierarchical trees to determine the specific area where the collision occurs.

Another possible enhancement is the use of NURBS instead of meshes in the volume-surface collision detection because they provide a better representation of surgical instruments.

Using multi-texturing it is possible to detect collisions between a surface and multiple volumes simultaneously in a single pass. However, it will not be possible to know which volumes collided, only that there was a collision.

References

[BF03] BOYLES M., FANG S.: 3Dive: an immersive environment for interactive volume data exploration. *J. Comput. Sci. Technol. 18* (January 2003), 41–47. 1, 2

[Bou01] BOURG D.: *Physics for Game Developers*, 1 ed. O' Reilly Media, November 2001. 1

[CG03] CRAIGHEAD M., GINSBURG D.: ARB_ocllusion_query extension specification. http://oss.sgi.com/projects/ogl-sample/registry/ARB/occlusion_query.txt. 1, 3

[CKN02] CARMONA R., KIENHOLZ P., NAVARRO H.: Construcción de árboles BSP via Algoritmos Genéticos. XXVIII Conferencia Latinoamericana de Informática. 2

[CLMP95] COHEN J. D., LIN M. C., MANOCHA D., PONAMGI M.: I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In *In Proc. of ACM Interactive 3D Graphics Conference* (1995), pp. 189–196. 2, 4

[CWK10] CHANG J.-W., WANG W., KIM M.-S.: Efficient collision detection using a dual obb-sphere bounding volume hierarchy. *Comput. Aided Des. 42* (January 2010), 50–57. 2

[EKE01] ENGEL K., KRAUS M., ERTL T.: High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (New York, NY, USA, 2001), HWWS '01, ACM, pp. 9–16. 4

[FKN80] FUCHS H., KEDEM Z. M., NAYLOR B. F.: On visible surface generation by a priori tree structures. *SIGGRAPH Comput. Graph. 14* (July 1980), 124–133. 1, 2

[GKLM07] GOVINDARAJU N. K., KABUL I., LIN M. C., MANOCHA D.: Fast continuous collision detection among deformable models using graphics processors. *Comput. Graph. 31* (January 2007), 5–14. 2

[GLM96] GOTTSCHALK S., LIN M., MANOCHA D.: OBB-Tree: A hierarchical structure for rapid interference detection. In *Computer Graphics* (1996), SIGGRAPH '96, pp. 171–180. 1, 2

[GRLM03] GOVINDARAJU N. K., REDON S., LIN M. C., MANOCHA D.: Cullide: interactive collision detection between complex models in large environments using graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (Aire-la-Ville, Switzerland, Switzerland, 2003), HWWS '03, Eurographics Association, pp. 25–32. 2

[HJ08] HANYOUNG JANG J. H.: Fast collision detection using the a-buffer. *The Visual Computer 24*, 7 (2008), 659–667. 2

[HK97] HE T., KAUFMAN A.: Collision detection for volumetric objects. In *Proceedings of the 8th conference on Visualization '97* (Los Alamitos, CA, USA, 1997), VIS '97, IEEE Computer Society Press, pp. 27–ff. 2

[JJH07] JANG H.-Y., JEONG T., HAN J.: Image-space collision detection through alternate surface peeling. In *Proceedings of the 3rd international conference on Advances in visual computing - Volume Part I* (Berlin, Heidelberg, 2007), ISVC'07, Springer-Verlag, pp. 66–75. 2

[KHM*98] KLOSOWSKI J. T., HELD M., MITCHELL J. S. B., SOWIZRAL H., ZIKAN K.: Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics 4* (January 1998), 21–36. 2

[KW03] KRUGER J., WESTERMANN R.: Acceleration techniques for gpu-based volume rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)* (Washington, DC, USA, 2003), VIS '03, IEEE Computer Society, pp. 38–. 3

[Lar08] LARSSON T.: Fast and tight fitting bounding spheres. In *Proceedings of the Annual SIGRAD Conference* (November 2008). 2

[LWlM04] LUM E. B., WILSON B., LIU MA K.: High-quality lighting and efficient pre-integration for volume rendering. In *In Proceedings Joint Eurographics-IEEE TVCG Symposium on Visualization 2004 (VisSym Š04* (2004), pp. 25–34. 4

[Mir98] MIRTICH B.: V-Clip: fast and robust polyhedral collision detection. *ACM Trans. Graph. 17* (July 1998), 177–208. 2

Scene	Mesh Rendering time(ms)	Volume Rendering time(ms)	Collision detection time(ms)	Total Rendering time(ms)	Total time(ms)	Collision detection overhead (%)
v_1s_1	0.043 ms	5.793 ms	0.502 ms	5.835 ms	6.337 ms	8.6%
v_1s_2	0.356 ms	5.793 ms	0.789 ms	6.148 ms	6.937 ms	12.8%
v_2s_1	0.027 ms	4.983 ms	0.481 ms	5.01 ms	5.491 ms	9.6%
v_2s_2	0.343 ms	4.983 ms	0.758 ms	5.325 ms	5.325 ms	14.2%

Table 1: *Rendering time and collision time for our test scenes, in environment #1*

Scene	Mesh Rendering time(ms)	Volume Rendering time(ms)	Collision detection time(ms)	Total Rendering time(ms)	Total time(ms)	Collision detection overhead (%)
v_1s_1	0.381 ms	1.811 ms	0.334 ms	2.192 ms	2.526 ms	15.2%
v_1s_2	4.324 ms	1.811 ms	0.272 ms	6.134 ms	6.407 ms	4.4%
v_2s_1	0.447 ms	1.688 ms	0.338 ms	2.135 ms	2.473 ms	15.9%
v_2s_2	4.452 ms	1.688 ms	0.262 ms	6.1394 ms	6.401 ms	4.3%

Table 2: *Rendering time and collision time for our test scenes, in environment #2*

[Raa08] RAABE A.: *Describing and Simulating Dynamic Reconfiguration in SystemC Exemplified by a Dedicated 3D Collision Detecion Hardware*. PhD thesis, Rheinische Friedrich-Wilhelms Universitat Bonn, August 2008. 2

[RHAZ06] RAABE A., HOCHGÜRTEL S., ANLAUF J., ZACHMANN G.: Space-efficient fpga-accelerated collision detection for virtual prototyping. In *Proceedings of the conference on Design, automation and test in Europe: Designers' forum* (3001 Leuven, Belgium, Belgium, 2006), DATE '06, European Design and Automation Association, pp. 206–211. 2

[vdB98] VAN DEN BERGEN G.: Efficient collision detection of complex deformable models using aabb trees. *J. Graph. Tools 2* (January 1998), 1–13. 2

[WVG92] WILHELMS J., VAN GELDER A.: Octrees for faster isosurface generation. *ACM Trans. Graph. 11* (July 1992), 201–227. 2

[WVW94] WILSON O., VANGELDER A., WILHELMS J.: *Direct volume rendering via 3D textures*. Tech. rep., Santa Cruz, CA, USA, 1994. 2

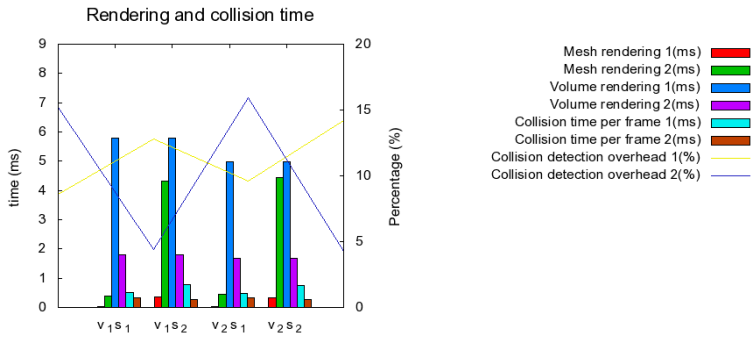

Figure 6: *Rendering and collision times from Tables 1 amd 2*