Chapter 5

Volumetric Shaders Used In The Production Of *Hollow Man*

Laurence Treweek and Brian Steiner,
Sony Pictures Imageworks
laurence@imageworks.com
brian@imageworks.com

Abstract

Volume rendering solid textures and deriving shadeable surfaces from 3D noise textures.

A description of techniques and tools used in the creation of the Transformation Sequences for the film Hollow Man. This talk will cover some of shaders and plugins used to animated and render a disintegrating human body.

5.1 Goals

Our goals were:

- To create at render time the look of a mass of tissue inside animating NURBS geometry.
- The shape of the animating disintegrating volumes must be completely controllable over time.
- The inner surfaces and textures must look realistic and be capable of shading with the same quality and control as the geometric surface.
- Must work with camera and object motion blur from fast camera movement and frantic animation of the body.
- Volumetric shapes created by the shader must be self shadowing and able to cast shadows.
- Rendering times must be reasonable enough to be used on a large area of the screen for eighty shots in production.

5.2 Methods

At the beginning of the project we analyzed the storyboards that needed to brought to screen and started to determine what kind of technology could be used to produce the desired look and animation.

5.2.1 Method 1 – Particles

To fill the enclosed NURBS geometry with particles that pack inside the volume. The particles could then be switched on or off to create an animating volume.

Pro: This method would be fairly easy to setup and animate using traditional particle creation and animation routines.

Con: Using particles would create a huge amount of data to keep around considering the number of body parts that would have to use this effect. It would also be very difficult to shade these particle to look like the variety of tissue type we needed to emulate.

5.2.2 Method 2 – The Visible Human Project

To use the existing imagery contained in the Visible Human Project to fill the volume.

Pro: The Visible Human Project contains images of slices through a human body taken at 2mm intervals. When these are stacked on top of each other you have a voxel-set that describes the external and internal look of the body with medical accuracy.

Con: The problem with using this data lies in the difference between the proportions of the geometry and the voxel data. The geometry used had to match the scale and proportions of Kevin Bacon and the voxel data would line up as it was extracted from a much heavier body. To get these to volumes to occupy the same space would entail developing a palette of complex tools to morph the two together. The isolation of individual objects within the slice data would also have been a complex task as they were visually indiscernible within each image. A possible solution to this would have been to use the MRI dataset to try and isolate different parts based on relative density, but this method would also be inaccurate and need cleaning up afterwards. The static size of this data would be enormous to distribute to servers and artists needed to render and work with it.

5.2.3 Method 3 – Procedural generation of volumetric data at render time

To generate, using 3D noise, the surfaces and textures inside the volume.

Pro: Storage requirements are virtually zero. The same 3D noise used in the shading of the geometric surface can be extended to the interior volume to form a continuous effect.

Volumetric data is not static and locked. Shader parameters can be used to change the density and look of the volumetric data. As the data is created by the shader, it can also be sampled, and averaged at arbitrary rates.

Con: Procedural data has to look convincing. Image based data cannot be used within the volume to add a realistic look as in the way a texture map would be used on a surface.

5.3 Why use RenderMan?

There are varieties of translating a piece of volumetric data to a static image. These can range from transforming a block of voxel data into screen space to "ray marching." RenderMan does not seem to be the obvious choice of renderer to solve this problem — so why did we use Renderman?
Reasons to use Renderman:

- Shaders are comparatively easy to create. Many arithmetical and vector functions already built-in. More time can be devoted to created higher level functionally instead of creating basic libraries.
- Final "look" is a known quantity. This factor was one of the major influences on deciding which package to use as a development base. The majority of shader writers and artists at Imageworks are more familiar with the workings of RenderMan compared to other renderers and there is also an expectation of what the final image quality will be. There is also familiarity with important functions that determine that "look" such as texture mapping and noise functions.
- It was advantageous to use one renderer for parts of the body. Because the outside surface of the object was visible at the same time as the interior volume on any one frame, it meant that a shader that could render both surface and volume would make a better blend.
- The alternative would have been to render the surface with RenderMan and depth composite it with the volume from another renderer.

With the addition of DSOs to RenderMan the possibility of extending functionality greatly increases beyond the traditional shader additions to the package.

Writing new renderers or adding new rendering software to a facility is always a large undertaking and goes beyond the initial problem of producing an image that work. There are other considerations such as fitting into to an existing production pipeline and training artists to use new tools. Even though we did create a specific renderer for Hollow Man, we kept its usage to a minimum and only used it where absolutely necessary.

5.3.1 Added RenderMan DSOs

Even though RenderMan gave us a good base to start working with we knew that extra functionality was going to be needed to make the shaders work.

5.3.2 Ray Tracing

In order to ray march through a volume we needed the ability to ray trace the geometry.

We experimented with using BMRT in rayserver mode together with the rayserver DSO to test our theories. At the beginning of the project we realized that memory consumption and processor usage would be high so we decided to develop our own rayserver DSO instead of running two renderers in parallel with duplicated scenes.

Our ray tracing DSO worked in two modes, read and write. The "write" mode was used in a pre-rendering phase to create a file that could be randomly accessed for ray intersection calculations. During the pre-render phase all the micropolygons that RenderMan created were written to a file along with all of their attributes. Inside the file the micropolygons were stored in world space and could be converted to a Wavefront `.obj` file for debugging purposes.

In "read" mode the file could be randomly accessed so that a ray could be fired into a cell of polygons rather than the whole scene, giving much faster calculations times. Although this was not an ideal solution it did work well enough in production by reducing the memory footprint needed to

ray trace. The DSO was also able to pass back to the shader extra attributes such as the name of the texture map applied to the intersected geometry or its uv values.

5.3.3 Animation Data

The shaders needed to aware of animation curves that would drive the disintegration effect. Different parts of the body needed to know their relative position within the overall transformation animation. To give the point being shaded a reference point we created a DSO that could read curve data. While marching through objects we could search for the closest point on one of these curves. When the closest point was found, the arc-length attribute was calculated and passed back to the shader. Giving us a simple means of animating the visibility of P.

```
point Pcurr = transform("current", "object", P);
float anim_length = 195; /* Make visible every point that is within 195 units of  heart. */
float current_length = BIG_NUMBER;
while(current_length > anim_length) {
    current_length = find_closest_point(Pcurrent); /* call reference curve DSO */
    Pcurr += step_size * trace_direction;
}
calculate shading at position Pcurr;
```

In the code example above if we animate the value of anim_length then we get a wipe effect across every object with this shader and curve data. Figure 5.1 shows the placement of the curves inside an object (a femur in this case) which were used by the DSO.

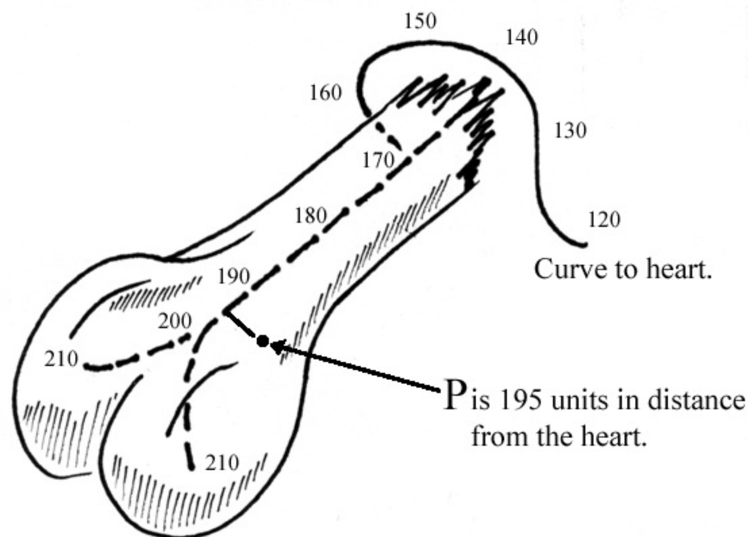

Figure 5.1:

This technique was used extensively in animating the transformation shots. The curves could be applied to many different type of objects and the same DSO could be used in many different shaders.

The curves themselves were actually generated by a Maya plugin developed by Scott Schinderman that let artists chain curves together to form a road map of the animation flow through the body. Adding or deleting curves from the tree could change the overall effect of the animation.

Figure 5.2: *Hollow Man*

5.4 Transforming Bones

The disintegrating bone effect was designed to look as if a cascading chemical reaction traveled from a source point in the bone, spreading outwards making the space occupied by the bone invisible in its path. The effect also needed to have a semi transparent amber-like leading edge that would illustrate the position of the chemical reaction between the bone and the reactive formula developed in the film's storyline. The effect needed to show five different types of materials during the course of the animation. In Figure 5.2, the bone effect is used exclusively but in all other shots the shader had to work with vary kinds of geometry, shaders and renderers surrounding it. The shader also had to hold up under a variety of viewing angles and animation conditions. The lighting design of the original plate photography also called for many soft shadow-casting lights to be used to mimic the fluorescent lighting of the laboratory.

5.4.1 The Layers

Layer 1) Wet Look.

This material needed to look as if a layer of fascia and collagen were coating the fat and outer bone materials. This layer was producing by taking point `P` at the surface and offsetting it by a fairly smooth noise value as in a regular bump calculation and then using the normal produced to calculate an extra specular contribution that was added in to the final lighting calculations.

```
point Pwet = P + wet_noise * N;
normal Nwet = normalize(calculatenormal(Pwet));
color CiWet = wet_colour * my_specular_function(Cl, Nwet);
Ci = surface_colours + (CiWet * WetAmount);
```

This kind of offset specular lookup produces a parallax shift in animation between the specular shading and the diffuse shading, giving the impression that there are two surfaces instead of one.

2) Fat.

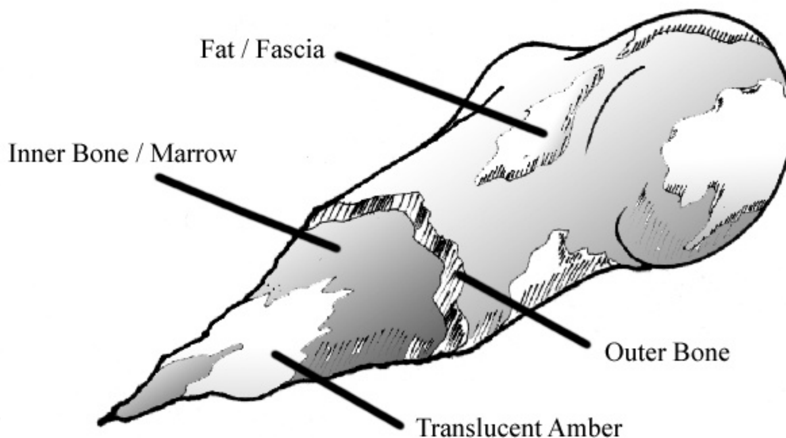

Figure 5.3: Bone layers.

The exterior of the bone needed a layer of fat added to it that also had to disintegrate along with the bone. This layer was produced by displacing the outer surface to form the impression of a clumps of fat stuck to the outer bone surface. The amplitude of the fat displacement was also used to blend in the amount of fat material at the surface. When the bone animated the amplitude decreased making the fat shrink closer to the surface and also mix in more of the outer bone surface until it disappeared completely.

3) Amber chemical reaction.

The amber like material needed to occupy a thin layer within the volume. The material needed to be semi transparent and reveal the bone interior below it. This was achieved by selecting a region in the depth of the bone that would be treated more like a traditional ray march through a gas cloud. Each sample added to the amber opacity and was clamped to a maximum accumulation value. A small amount of noise affected each sample as the result needed to look more glassy than cloudy. The accumulation of each sample differed in this region of the shader as the bone marrow needed to seen through the amber. This meant that instead of just doing an “under” compositing calculation on each sample, we need to use the density of the amber as a shadowing factor on the marrow underneath and use a “darken” operator instead.

4) Bone Surface.

This material as well as the fat and wet needed to work as static materials for the majority of the shots as well as being an element in a dynamically changing surface. The bone surface material contained bump maps and texture maps that were always calculated no matter if the actual surface was visible or not. This was done for several reasons.

a) It is not a good idea in shader to have texture calls and assignments inside “if” statements. You get unprecidictable shading when points on the same micropolygon do not have consistent data to blend between.

This is bad:

```
point Pobj = transform("current", "object", P);
point Pcurr = Pobj + step_size * ray_vector;
if (length(Pcurr - ObjCntr) < surface_depth)
    Ci = interior_colour;
```

```
else
    Ci = exterior_colour;
```

This is better:

```
point Pobj = transform("current", "object", P);
point Pcurr = Pobj + step_size * ray_vector;
float blend = length(Pcurr - ObjCntr) / length(P - ObjCntr);
Ci = (blend * interior_colour) + ((1 - blend) * exterior_colour);
```

How the blend value is filtered and manipulated is entirely up to the user, the important thing is that all points on the micropolygon have the same attribute list to interpolate.

b) Calculating surface texture map values was not a large percentage of render time.

c) Surface shading could used to blend from the outside to the inside. Instead of making a hard transition going from painted textures to procedural ones, a blend region could be employed to fade between the exterior surface and the interior volume.

5) Bone Interior.

This material was used to make up the majority of the texture and surface characteristics of the volume inside the bone. As the disintegration effect went deeper into the bone the density of the volume became more porous. This effect was produced by using a cylindrical coordinate system which meant that when the radius of the cylinder decreased the frequency of the noise increased. This section of the shader was used for the bulk of the volume inside the bone as it handled the calculation of data needed to do shading away from the geometric surface.

This shader was initially based on Clint Hanson's previous work with ray marching in RenderMan. It then proceeded to get more complicated with multiple surface types. We also took Clint's theories to the next level by adding the ray tracing ability mentioned above. With this trace DSO we could ray march through arbitrary shaped objects and not limit ourselves to spheres or geometry hard coded into the shader. The animation curve DSO also mentioned above gave us some key information to start constructing an animating shader and also a lightable volume. Figure 5.4 illustrates how more information than just the curve length was gained from that DSO.

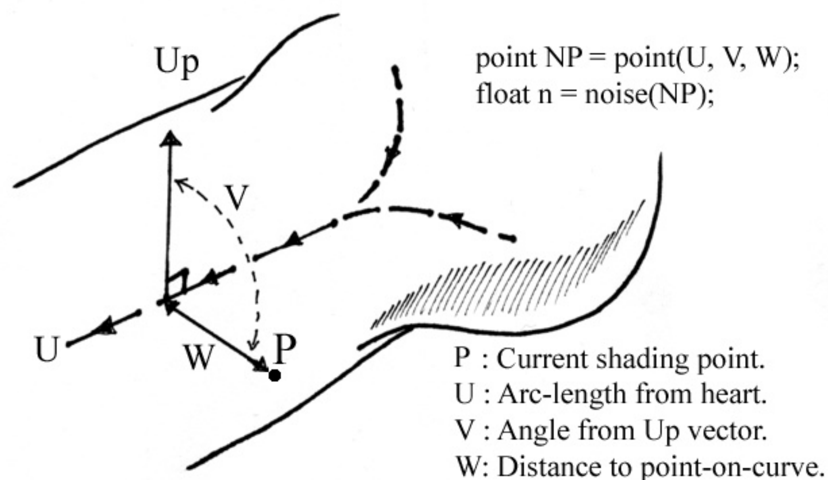

Figure 5.4: Cylindrical Coordinate System

The cylindrical coordinate system in Figure 5.4 was used to make the shaded and animating surface more interesting by adding noise. By comparing P's relationship to U (U being the arc-length of the curve) a point could be turned on or off based on this distance, $(P < U)$, giving a wipe effect along the length of U. By adding the value of W into the equation we could form a shape with a conical leading edge. Going further and adding a noise multiplication to W made the leading edge a noisy conical shape, which was more in the realm of the desired effect. The pseudo-code below gives you some idea of this worked.

```
float blood_length = 190; /*current animation value of the distance from heart to reveal */
float blood_length_offset = 10; /* length of conical leading edge */
float max_bone_depth = 5; /* maximum bone radius to expect */
float max_bone_depth_scaled = max_bone_depth *
                              clamp((U - (blood_length - blood_length_offset)) /
                              (blood_length - blood_length_offset), 0, 1);
if (W <= max_bone_depth_scaled) {
    shade point;
} else {
    keep marching;
}
```

Same with noise factored in:

```
float depth_noise = noise (point(U, V, W)) * noise_scalar;
if (W + depth_noise <= max_bone_scale) {
    shade point;
} else {
    keep marching;
}
```

By changing the value of `blood_length_offset` it was possible to change the length and steepness of the conical profile.

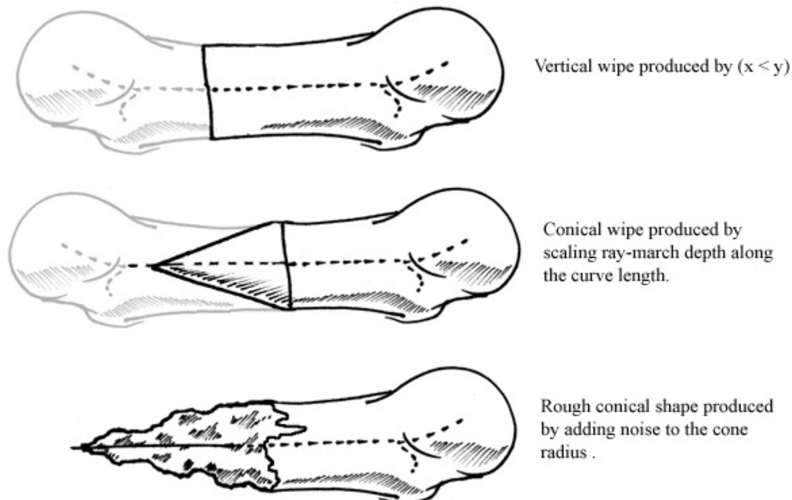

Figure 5.5: Bone wipes.

5.4.2 Going into shader "no man's land"

When writing RenderMan shaders you can get used to taking some high level functionality for granted, such as having `dPdu` and `dPdv` calculated for you giving you the ability to easily call

functions like calculatenormal. Filtered texture map lookups are also much easier when you have `s`, `t`, `u` and `v` right there on the microplygon. When you are dealing with volumes and marching away from the surface you soon begin to realize that these attributes and functions quickly become useless are you are trying to shade the look of a point in space that is really away from the one that RenderMan is trying to shade in screen space. This is easier to understand in diagrammatic form below.

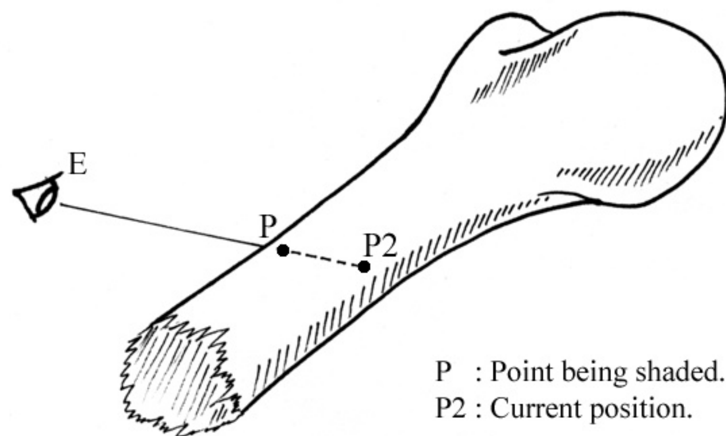

Figure 5.6: Bone ray marching.

Information created at point `P` by RenderMan becomes more and more meaningless the further we march towards point `P2` inside the object. Viewed from location `E`, `P2` occupies the same place on screen as `P` but in object or world space it is in an entirely different location. Executing `calculatenormal(P2)` will not give you the result you expected as `dPdu` and `dPdv` refer to attributes calculated for point `P`.

To remedy this problem it was necessary to calculate a new shading normal from within the shader. This was done by making use of the coordinate system described by U, V and W, mentioned above. In its simplest form the shader created 3 points to build a triangle. `P2` was already calculated as one point on the triangle as this was our current point. The other 2 points were calculated by gaining new noise values for U, V and W at offset locations based on a sampling size parameter. The code below shows how the triangle was constructed.

```
/* size for sub sampling based upon size of ray march step*/
float sampleSz = 0.001; /* the smaller this number the sharper the image*/
/* Current ray marching point in object space */
point PcurrObj;
/* closest point on curve in object space */
point CP0;

/* create vector for direction from the closest point on curve to the current point */
vector Vec0 = vector(normalize(PcurrObj - CP0));
float depth = length(PcurrObj - CP0);

/* curve tangent passed back by curve reading DSO */
vector crvTan;
```

```
/* make a new point further along the curve build P1 from*/
point CP1 = CP0 + (sampleSz * crvTan);
point P2 = CP0 +  (depth  + (noise(point(U,V,W)) * Vec0);
/* P1 is parallel to P2 but further along the curve */
point P1 = CP1 + (depth + (noise(point(U + sampleSz, V, W)) * Vec0);

/* point PP1 is P2 rotated around curve tangent by translating sampleSz to an angle*/
/* Vec1 is now the direction to P1 */
vector Vec1 = normalize(PP1 - CP0);
/* calculate new value for V because PP1 has been rotated around curve tangent */

/* Upvector was set to (0, 0, 1) as most bones are aligned vertically in rest position */
float V0 = Vec1 . Upvector;
point P0 = CP0 + (depth + (noise(point(U, V0, W)) * Vec1;
```

P2->P0 and P2->P1 are now 2 vectors that are perpendicular to each other and describe the triangle. Figure 5.7 shows this triangle and its relationship to the curve.

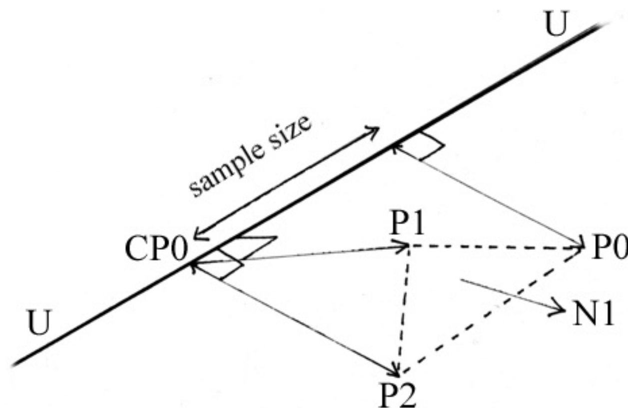

Figure 5.7: Calculating Normals.

By normalizing the vectors and take a cross product we had a vector (N1) that could be transformed into current space, (using ntransform), and then used for lighting calculations. Instead of just calculating the noise gradient, as you would do if you were ray marching through a cloud texture, this method gave us a cylindrical biasing for the normal which made the lighting fit closer to the outer surface lighting. If we had not done this we would have had a surface that dissolved to reveal a cloud and the shape definition would have been lost early on in the effect. When the current ray marched point was near to the geometry surface the normals N1 and N were blended to form a transition from smooth outer surface to a more honeycombed marrow interior.

The method described above can be turned into an iterative process whereby smaller and smaller triangles can be created until a minimum error tolerance has been reached. By constructing a larger triangle with a larger `sample_size` value we could create a lower frequency normal that be used for calculating a “wet look” specular contribution as we did for the geometric surface.

5.5 Rendering The Heart With Volumes

As well as the bones, organs and muscle needed to transform from invisible to visible. Each piece needed to appear as if it had thickness and volume as it made it's transition. Different rendering techniques were used on different body parts, as each piece would pose its own set of unique problems. We will look at the rendering for the heart in this section. The basic ray marching technique will be discussed in some detail, along with some of the problems and features that were implemented in order to produce production quality renders.

The heart was one of the more complex body parts that went through the transformation. It has four hollow chambers that twist around each other, with very thin walls that separated them. The shape is constantly deforming in multiple directions at once. The heart seemed like a good candidate for ray marching despite the complex geometry. In the original story boards, the heart would be transforming full screen, so the renders needed to be able to maintain a high level of detail. Because the shots would be so close to camera, we felt that surface rendering would not work. In the final transformation sequence, because of timing issues, the heart transformation was not featured full screen as originally thought, but the ray marching technique still worked well for it and many other object.

5.5.1 Basic Ray Marcher

Ray Intersection

The first task that the ray marcher needed to do is to define an "in point" and a "out point" for every eye ray that intersected the object. The in and out points basically defining the volume that will be marched though for each pixel of the image. If a ray tracer is used this task can be easy, since the render should know where all the geometry is in the scene. If a ray is fired from any point the render will return any object that was hit by the ray. RenderMan is not a ray tracer so finding the location of surfaces other than the current shading point can be difficult. If the volumes are limited to simple objects such as cubes, spheres, and other primitive shapes, they can be reconstruct in the shader with very little information. If the volumes being render are more complex, the problem of finding the in and out points becomes a lot more difficult in a non ray tracer. In RenderMan there is no way to find these complex surfaces without some outside aid. Luckily RenderMan has the ability to use DSOs that can extend its functionality. It is possible to write a DSO to raytrace objects. This is a good way to taylor a raytracer to meet all the needs that may arise when doing a particular ray march, but may not necessarily be the easiest thing to do. using BMRT as a ray server can also be a good idea with a lot less overhead than writing a custom raytracer. Depth map renders of backside geometry can also be used to get the depth of simple object but may not always work if the object has a lot of overlapping parts.

Marching Through The Volume

Once an "in" and an "out" point are found for the ray, the next step is to break the ray up into small steps along its length. Every time we make a step down the ray, we run a density function which tells us how thick the volume is at that point. To get the position of the sample point we could write something that looks like this.

```
volume_distance = length(out_point - in_point)
num_of_steps = volume_distance/step_size
step_vector = (out_point -  in_point)/num_of_steps
current_position = inpoint
while(current_step < num_of_steps) {
      current_position += step_vector
}
```

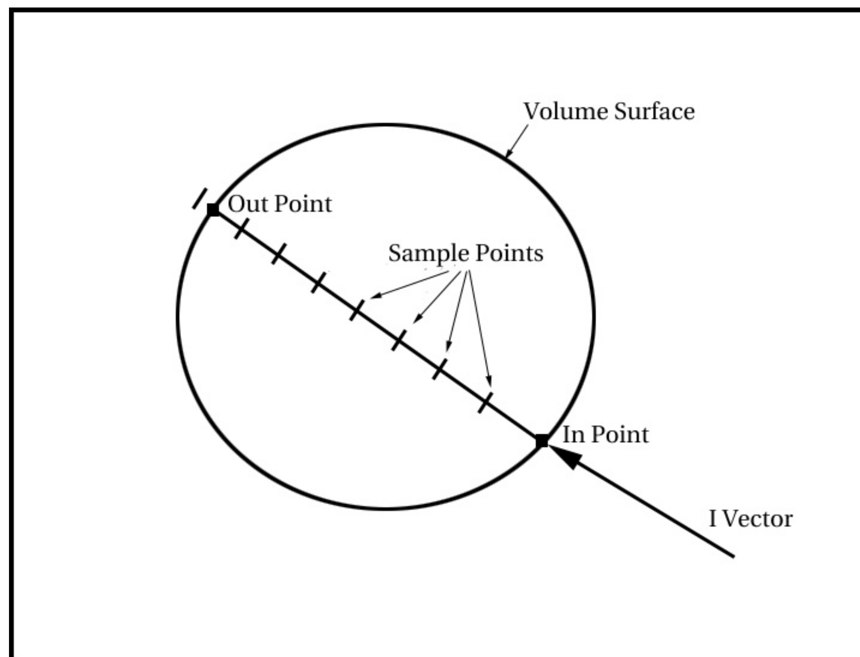

Figure 5.8: Sample points for ray marching.

If bigger steps are used, rendering times can be reduced dramatically, at the cost of low detail and banding artifacts. If smaller step are taken, the render will be smoother, but it can also slow down to a crawl if the step sizes are to small. Two things can be done to help minimize banding in the ray marcher, which will in turn allow for larger steps sizes. The first place banding can occur is at the back surface of the object. If the objects surface is curved or at an angle to the viewer, the number of steps taken through the volume may change from pixel to pixel. If the steps are very small, or the volume is thick enough that we don't see the back end, the problem may never be noticed. In semi transparent volumes, with larger step sizes banding will start to show up. To fix this banding, we can multiply the result of the last sample by distance to the "out" point divided by the step size.

As the ray marcher steps though the volume it may also be producing banding artifacts. If the steps are too big, or the noise in the volume is too detailed or too dense, the discrete steps between the samples start to show up. Jittering the sample points can be a great way to break up this type of banding artifacts. Jittering will produce a noise that looks like grain in the volume, but our eye finds this noise less distracting than the banding. It is not necessary to jitter every step taken though the volume. If the start point for the ray is jittered, the visual effect is the same as if every step was jittered, but only one noise call is made per ray instead of one call per step.

Density Function

At every sample point that the ray marcher takes, it need to run a density function. The sole purpose of this function is to return the thickness of the volume for that sample point. This function can sometimes be very complex if the volume needs to have a lot of detail. The density function will be mostly responsible for the final look if the volume. Along with the density function another function could be run to get color. This can be useful if the color of the volume has a different noise than the density.

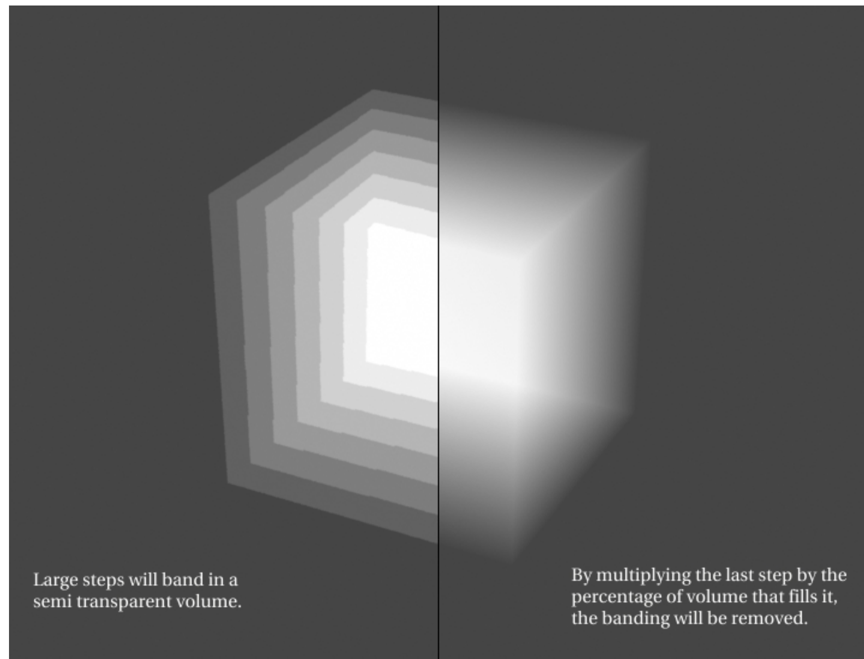

Figure 5.9: Eliminating banding with big step sizes.

Light In The Volume

If the volume needs to have shading, a normal needs to be calculated at every step. To calculate this normal, the density of the current sample point must be taken. Next the sample position is offset in each axis. For each offset another density function is calculated and the density from the original sample is subtracted. Each of these results is put into the corresponding component of a new vector. This new vector is called a gradient vector and always points towards the less dense parts of the volume. This vector works very well for the normal.

```
current_density = get_density(sample_P)
x_grade = get_density(sample_P - offset_in_x)-current_density
y_grade = get_density(sample_P - offset_in_y)-current_density
z_grade = get_density(sample_P - offset_in_z)-current_density
N_vol  = normalize(x_grade,y_grade,z_grade)
```

This normal calculation is very easy to make, but if you look the the above pseudo code you can see that the density function has now been run four times instead of one. If the density calculation is complex there could be an incredible time hit on the render. After the normal has been found for the sample, lighting is run as it normally would be with the exception that an illuminance loop should be used instead of the default diffuse and specular calls. The `diffuse()` and `specular()` functions assume that the light will be calculated at `P`. In the case of the volume render, the light should be calculated at the current sample point.

Compositing Samples

After the color and opacity has been calculated at a sample point, the value must now be added to the total accumulated volume of the ray. In some cases, the color and opacity can just be summed up to produce a volume that gets brighter and more dense as each step is sampled. This produces a

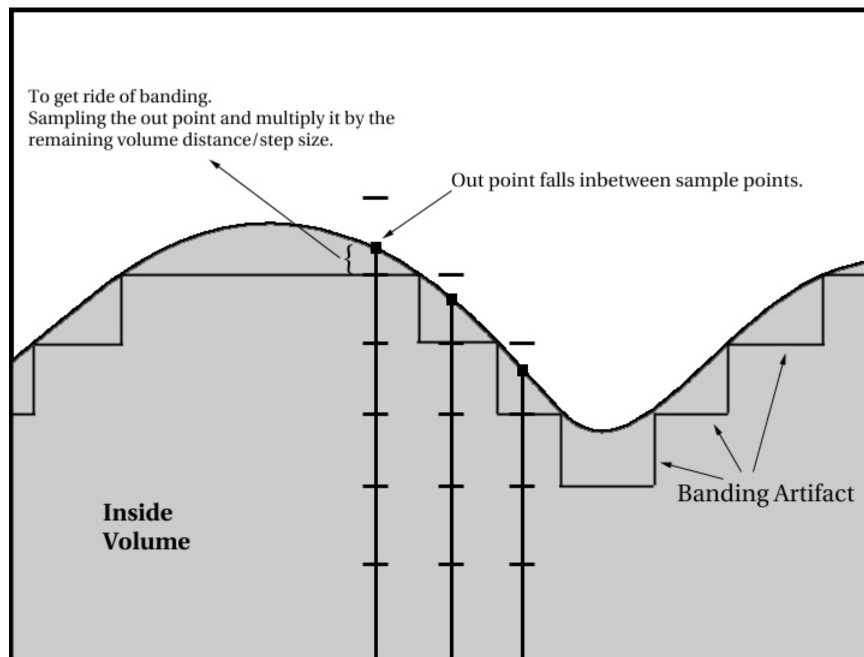

Figure 5.10:

volume which looks like it has some self illumination. If the render needs to match existing lighting from a plate, a better composite may need to be done. A simple over composite of each sample can produce nicer lighting for more solid volumes and will retain dark areas and give better depth cueing as the volume moves. Once the samples are composited, they are essentially collected at the shading point P on the object. This fact is important to keep in mind. RenderMan shades surfaces, not volumes. which means that we may run into trouble later when trying to do things like motion blur or shadows.

After all the steps are complete, the ray march has a structure that looks something like this:

```
get in and out intersection points of ray I with object
divide ray into sample steps
while (current sample < total number of samples) {
      call density function
      call color function
      calculate normal
      calculate lighting
      composite samples
      jump to next sample point
}
set output color to the totaled color samples
set output opacity to the totaled density samples
```

5.5.2 Speeding Up A Ray Marcher

It is easy to produce impressive results with a very simple ray marcher. Because it can be so easy to set up simple tests that show a lot of potential, we can sometimes be lured down the path of using a ray marcher on a production, only to abandon it later because of speed issues. As a simple ray marcher is extended with all the things it needs to produce production quality images, we find that

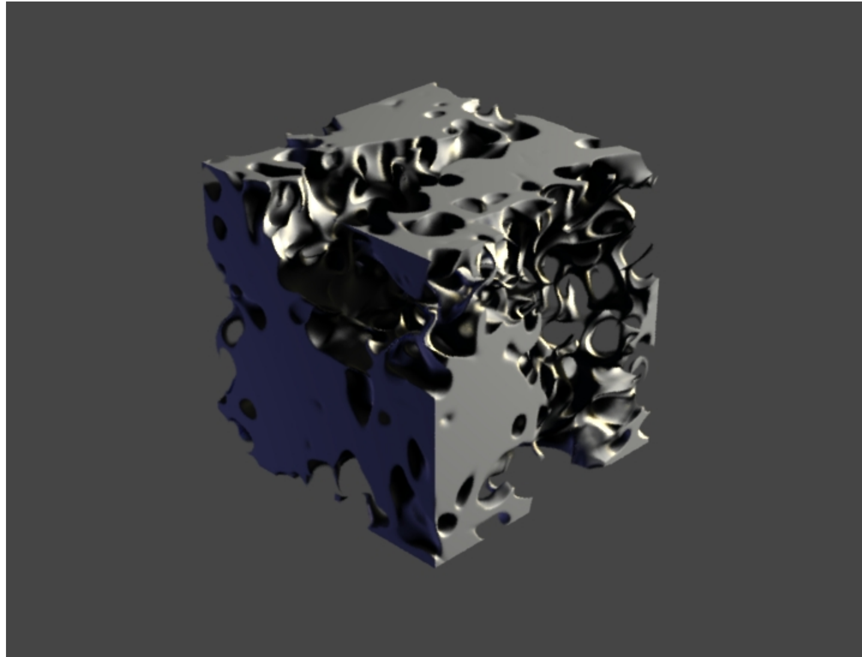

Figure 5.11: Shading.

it can become bogged down very quickly to the point of being impractical. The ray marcher is very repetitive, because of this, it sometimes only takes one slow calculation placed in a frequently run part of the shader to slow the render down to unbearable speeds. This was a concern when deciding to use ray marching for the transforming sequences in *Hollow Man*. Everything that needed to be added to the ray marcher had to be carefully considered based on the benefit of the feature compared to the speed. Optimizing the ray marcher became an essential part of the success or failure of this technique.

Density Function

The density function will probably be the place where the render is most likely to get bogged down. This function usually starts out very simple, but quickly grows in complexity as the look of the volume is refined. It is usually filled with fractal noises and power functions or what ever else is needed to achieve the desired look. This function is also most likely to be called more than any other function. As an example if a density function was created that took .01 of a second to calculate, at first this may not seam like that much time. If we render a volume that fills a 720x546 video frame, with an average of 100 samples per pixel. the function would get called 39,312,000 times in an unoptimized ray marcher. This would take 109 hours to run. If we added shading calculations to this, the time to run the destiny functions would increase four times, taking 436 hours to run. Add a raytraced shadow that runs another 100 density samples per step and it will now take 11,247 hours to calculate all the density functions. In comparison the .01 second function would take 1.09 hours if it was calculated only once per pixel in the video frame. In this worse case scenario the ray marcher could take over ten thousand times the render time of the surface render.

Minimizing Density Calls And Other Calls

In the renders that were done on HollowMan, the volume would animate from a totally empty void that would grow across the object with an abrupt transition from empty to solid densities. In the early frames of the animation there was a lot of empty space. This empty volume would take a incredible amount of time to render, with a mostly black frame as the final result. The easy way to speed up these empty areas was the addition of a conditional inside the step loop that checks the density and only executes any other calculations if it is grater than zero.

```
while (taking samples) {
      get density
      if(densiy > 0){
              get color
              get normal
              do shading
              composite samples
      }
      goto next sample
}
```

With this simple addition the render runs quicker through the empty areas. But the time which it takes to run the density function in these empty areas still adds up. In most of the volume renders done on *Hollow Man* the density function could be separated into two phases. The first calculating a low frequency animated pass that would drive where the volume was on a particular frame. The second was a set for fractal noises which created the detail of the volume. It was possible to split these into two functions, the first one which we will call the `active_volume` and the second one the actual "density" function.

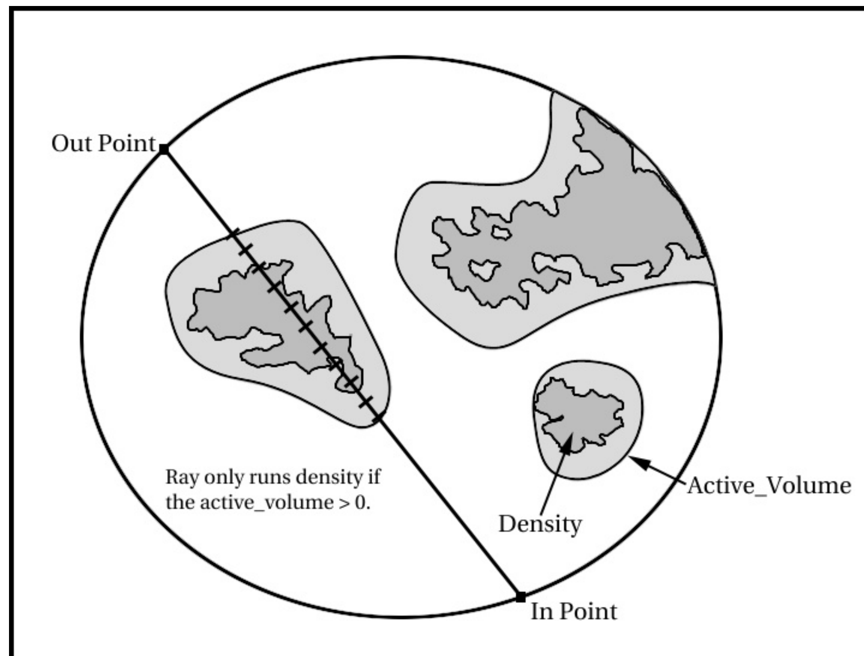

Figure 5.12: Active volumes.

The `active_volume` can be any low frequency function that returns values greater that zero in places where there is a chance that the density function might produce results greater than zero. If

the active_volume is 0, then there is no chance that the density will return anything. When the density function is called it will first call the active_volume to see if it should continue running though the rest of it's function.

```
function get_active_volume()
{
      return quick low frequency pattern
}

function get_density()
{
      get_active_volume
      if (active_volume > 0) {
              run slow fractal noises
      }
      return density * active_volume
}
```

In this case the density is multiplied by the active_volume to ensure that there can not be any density grater than zero outside the active part of the volume. Now as the ray marcher steps though empty space the density function is only run when there is a good chance that it will produce more than a black void. Once these two speed ups were implemented, early frames of animation which were taking hours to render, were now taking minutes.

Minimizing Step Calls

Another way to speed up the ray marcher is to cut down on the number of sample points all together. The easiest place were steps can be cut out is when the density of the volume adds up to one. Once the density of and object has reaches one, there is no chance of seeing any thing behind it at that point. The ray march can stop early, saving valuable time, especially on very dens parts of the volume. The other place where it makes sense to cut down on the sample points is in the empty parts of the volume. At fist this sounds easy to do, take big steps until you hit some density. Once you hit something, backup a little and start marching through with a finer step size to get the detail. This does not work quite that easy though. As large steps are taken through the volume there is a chance of totally missing small details all together. Not knowing that the step has just jumped over a detail, there is no indication that the area should be run though in finer detail. If large steps are used to only look for the active_volume there is less of a chance of missing pieces because this function should be a lower frequency than the density function, but near the edges there will still be pieces that are sometimes missed. The easiest way around this is to add a border around the active function, so the bigger steps don't have to actually hit the active area before it reduces the step size. The bigger the border the less chance of errors, but the less the time savings as well.

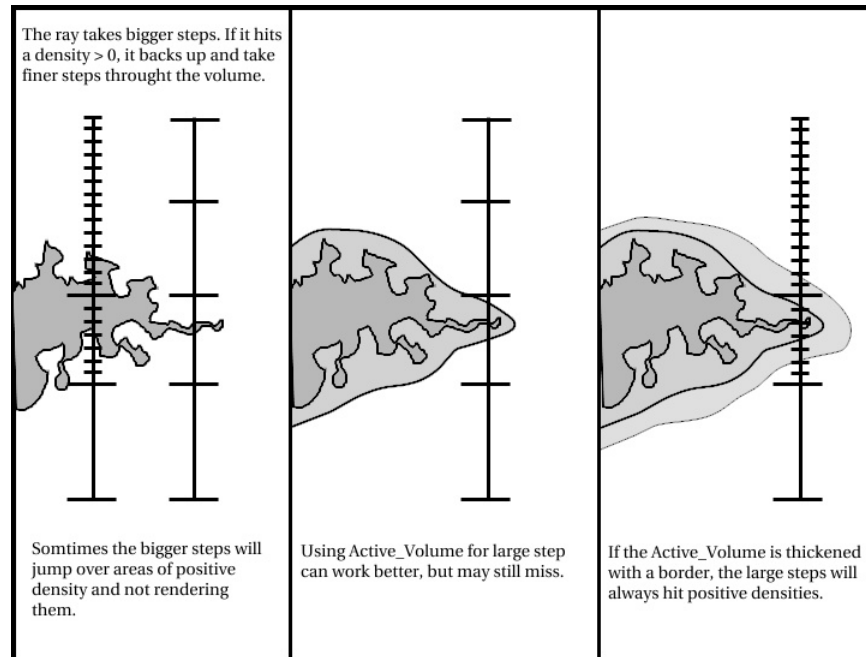

Figure 5.13:

Listing 5.1 is a surface shader that implements a simple ray marcher.

Listing 5.1: `srf_vol_cube.sl`

```
/* srf_vol_cube - Brian Steiner - Sony Pictures Imageworks

   This shader raytraces a box that is one unit in size,
   and then ray marches through the volume
   StepSize            - distance between sample points.
   StepJitter          - 0-1 jitter the sample position.
   Density             - volume thickes per unit.
   Epsilon             - offset for calculating gradient normal.
   Vol_Mult, Vol_Offset - animation controls.
   Do_Shading          - if 1, shading will be calculated.
   SurfNormalDepth     - the mixing depth from surface
                         normal to volume normal.
   Additive            - if 1 add samples, if 0 over samples .
   ShowActiveVol       - if 1 show the active volume instead of density.
   RunShadowPass       - set to 1 if running a shadow pass.
*/


/*----------------------------------------------------*/
/* fnc_traceBox returns an intersection point on a box */
point
fnc_traceBox (float XMin;
              float XMax;
              float YMin;
              float YMax;
              float ZMin;
              float ZMax;
              float idx;
              string refractSpace;)
{
    extern point P;
```

```
    extern vector I;
    extern normal N;
    vector Rd,Ro;
    point Ri;
    vector Pn;
    float D,T;
    float TMin = 1000000;
    vector IN;
    normal NN;

    IN = normalize(I);
    NN = normalize(N);
    Rd = vtransform(refractSpace,IN);
    Ro = transform(refractSpace,P);

    /*plane_z_min*/
    Pn = (0,0,1);
    D = ZMin * -1;
    T = -(Pn . Ro + D) / (Pn . Rd);
    if(T > 0){
        TMin = T;
        Ri = Ro + T*Rd;
    }
    /*plane_z_max*/
    Pn = (0,0,-1);
    D = ZMax;
    T = -(Pn . Ro + D) / (Pn . Rd);
    if(T > 0 && T < TMin){
        TMin = T;
        Ri = Ro + T*Rd;
    }
    /*plane_x_min*/
    Pn = (1,0,0);
    D = XMin * -1;
    T = -(Pn . Ro + D) / (Pn . Rd);
    if(T > 0 && T < TMin){
        TMin = T;
        Ri = Ro + T*Rd;
    }
    /*plane_x_max*/
    Pn = (-1,0,0);
    D = XMax;
    T = -(Pn . Ro + D) / (Pn . Rd);
    if(T > 0 && T < TMin){
        TMin = T;
        Ri = Ro + T*Rd;
    }
    /*plane_y_min*/
    Pn = (0,1,0);
    D = YMin *-1;
    T = -(Pn . Ro + D) / (Pn . Rd);
    if(T > 0 && T < TMin){
        TMin = T;
        Ri = Ro + T*Rd;
    }
    /*plane_y_max*/
    Pn = (0,-1,0);
    D = YMax;
    T = -(Pn . Ro + D) / (Pn . Rd);
    if(T > 0 && T < TMin){
        TMin = T;
        Ri = Ro + T*Rd;
    }
    return Ri;
}


/*--------------------------------------------------*/
```

```
/* active_volume - controls animation in the volume */
float
active_volume(point Pos; float vol_mult, vol_offset;)
{
    return (noise((Pos+30.445)*2)-.5+vol_offset)*vol_mult;
}



/*-----------------------------------------------------*/
/* density function will return the final volume density */
float
get_density(point Pos; float vol_mult, vol_offset;)
{
    float dens = 0;
    float activeVol = 0;
    float offset_active = .1;
    float mult_active = 20;
    activeVol = active_volume(Pos,vol_mult,vol_offset);
    dens = pow(1-abs(noise(Pos*7)*2-1),3);
    dens += pow(1-abs(noise((Pos+24.72)*7)*2-1),3);
    return activeVol + (dens-2);
}


/*-------------------------------------*/
/* normal calculation inside the volume */
normal calcGradeNorm(point Pos; float vol_mult, vol_offset, dens, epsilon;)
{
    normal Nd;
    Nd = normal (get_density(point (xcomp(Pos) - epsilon, ycomp(Pos),
                                    zcomp(Pos)),vol_mult,vol_offset) - dens,
                 get_density(point (xcomp(Pos),
                                    ycomp(Pos) - epsilon, zcomp(Pos)),vol_mult,vol_offset) - dens,
                 get_density(point (xcomp(Pos),
                                    ycomp(Pos), zcomp(Pos) - epsilon),vol_mult,vol_offset) - dens);
    Nd = ntransform("object","current",Nd);
    return Nd;
}




/*---------------------------------------------*/
/* shading function returns diffuse ans specular */
void get_shading (point Pos;
                  normal Nf;
                  vector V;
                  float Roughness;
                  output color diff;
                  output color spec;)
{
    extern vector L;
    extern color Cl;
    diff = 0;
    spec = 0;
    illuminance (Pos, Nf, radians(90)){
        diff += Cl * max(0,normalize(L).Nf);
        spec += Cl * specularbrdf(L, Nf, V, Roughness);
    }
}



/*-------------*/
/* nomal mixer */
normal
fnc_normalMix (normal N1; normal N2; float mixer)
{
```

```
    float N1_mag = 1;
    float N2_mag = 1;
    normal NN1 = normalize(N1);
    normal NN2 = normalize(N2);
    normal result;
    N1_mag *= 1-mixer;
    N2_mag *= mixer;
    result = normalize(NN1 * N1_mag + NN2 * N2_mag);
    return result;
}



/*--------------------------*/
/* main ray marching shader  */
surface
srf_vol_cube(float  StepSize       = 1;
             float  StepJitter     = 0;
             float  Density        = 1;
             float  Epsilon        = .001;
             float  Vol_Mult       = 1;
             float  Vol_Offset     = 0;
             float  Do_Shading     = 1;
             float  SurfNormalDepth = .05;
             float  Additive       = 1;
             float  ShowActiveVol  = 0;
             float  RunShadowPass  = 0;
    )
{
    point  inPoint_obj  = transform("object",P);
    point  outPoint_obj = fnc_traceBox(-.501,.501,-.501,.501,-.501,.501,1,"object");
    vector V  = normalize(-I);
    normal Nf = normalize(N);
    float  Roughness    = .21;
    color  diff = 1;
    color  spec = 0;
    float  vol_length   = length(outPoint_obj-inPoint_obj);
    float  numOfSteps   = vol_length/StepSize;
    vector step_obj     = (outPoint_obj-inPoint_obj)/numOfSteps;
    vector step_cur     = vtransform("object","current",step_obj);
    float  curStep      = 0;
    float  density_sum  = 0;
    color  color_sum    = 0;
    float  shad_sum     = 0;
    float  remainder    = 100;
    float  cur_density  = 0;
    color  cur_color    = 0;
    float  density      = StepSize * Density;
    float  jitter       = (random() - .5) * StepJitter;
    float  cur_depth    = 0;
    point  Pcur_obj     = inPoint_obj + jitter * step_obj;
    point  Pcur        = P  + jitter * step_cur;

    Oi = 0;
    Ci = 0;

    /*----------------------------*/
    /*     step loop              */
    while(curStep < numOfSteps && density_sum < 1){
        cur_density = 0;
        cur_color = 0;

        /*--- Run Density Function ---*/
        if(ShowActiveVol == 1)
            cur_density = active_volume(Pcur_obj,Vol_Mult,Vol_Offset);
        else
            cur_density = get_density(Pcur_obj,Vol_Mult,Vol_Offset);

        /*--- If Density > 0 Run The Rest Of The Loop ---*/
```

```
        if(cur_density > 0 && RunShadowPass == 0){
            cur_color = cur_density;
            cur_color = 1;
            if(Do_Shading > 0){
                if(cur_depth > 0){
                    normal Vol_Nf = calcGradeNorm(Pcur_obj,Vol_Mult,Vol_Offset,
                                                  cur_density,Epsilon);
                    Vol_Nf = normalize(Vol_Nf);
                    Nf = fnc_normalMix(Nf,Vol_Nf,clamp(cur_depth/SurfNormalDepth,0,1));
                }
                get_shading(Pcur,Nf,V,Roughness,diff,spec);
            }
            cur_color = (cur_color * diff) + spec*(1,.8,.2);

            /*---- if sample is not a full step ----*/
            remainder = numOfSteps - curStep;
            if(remainder < 1){
                cur_density *= remainder;
            }

            cur_density *= density;
            cur_color *= clamp(cur_density,0,1);

            /*---- Composite Sample ----*/
            if(Additive > 0){
                /* Just Add Up Density */
                density_sum += max(0,cur_density);
                color_sum += clamp(cur_color,color 0, color 1);
            }
            else{
                /* Do Over Instead of Add */
                cur_color = clamp(cur_color,color 0,color 1);
                cur_density = clamp(cur_density,0,1);
                color_sum = ( cur_color) * (1-density_sum) + color_sum;
                density_sum = (cur_density) * (1-density_sum) + density_sum;
            }
        }
        else {
            /* if Shadow Pass */
            if(Additive > 0){
                shad_sum += max(0,cur_density);
            }
            else{
                cur_density = clamp(cur_density,0,1);
                shad_sum = (cur_density) * (1-shad_sum) + shad_sum;
            }
            if(shad_sum >= .5){
                density_sum = 1;
                color_sum = 1;
            }
            P = Pcur;
            /* Displace Point To Current Sample */
        }

        /* jump to the next sample point */
        curStep += 1;
        Pcur_obj += step_obj;
        Pcur += step_cur;
        cur_depth += StepSize;
    }

    Ci = color_sum;
    Oi = density_sum;
}
```

5.5.3 Integration

Once we have got the basic ray marcher running in a some what optimized form. It's time to start thinking about how we are gone to integrate the volume with our specific geometry and fit the final renders into the plates. The first problem was to figure out the best way to trace the geometry. In the case of the heart the object has a very complex shape. It is an object that has two hollow chambers in the main part and two other hollow chambers on either side.

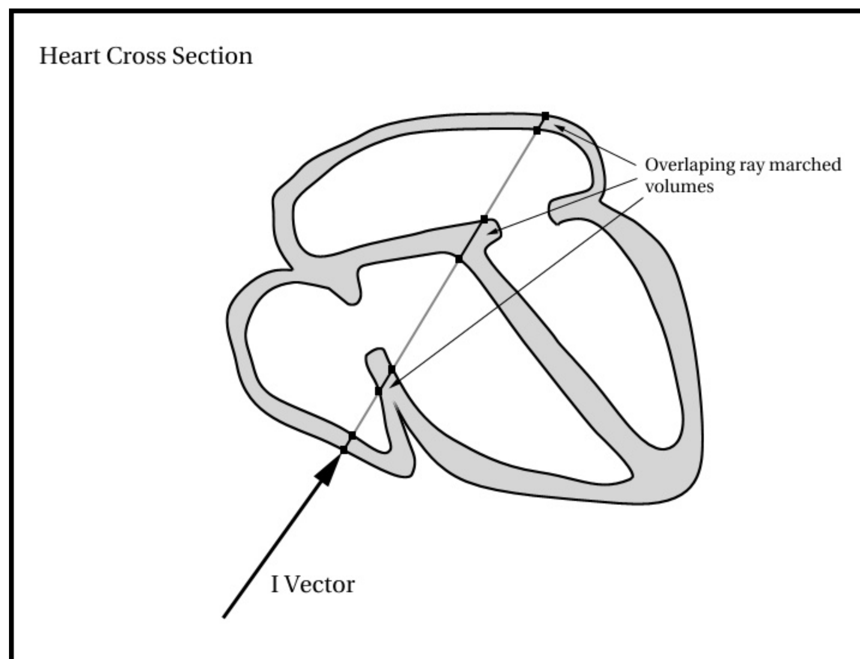

Figure 5.14: Heart cross-section.

Raytracing

Because all the chambers of the heart move in different directions at different times, and some of the surfaces were so close, there was concern about surfaces interpenetrating. Because of possible interpenetration, the first version of the ray marcher ran only on the closest point `P` for any `I` vector. From that closes point it would raytrace all the surface behind it. The shader kept track of when it entered a volume or left a volume by checking to see if the normal was pointing in the same or opposite direction of the ray. A layer variable was kept, to which the number one was added to when it went into the volume and one was subtracted as it left the volume. The ray marcher was then told to only march on a specific layer.

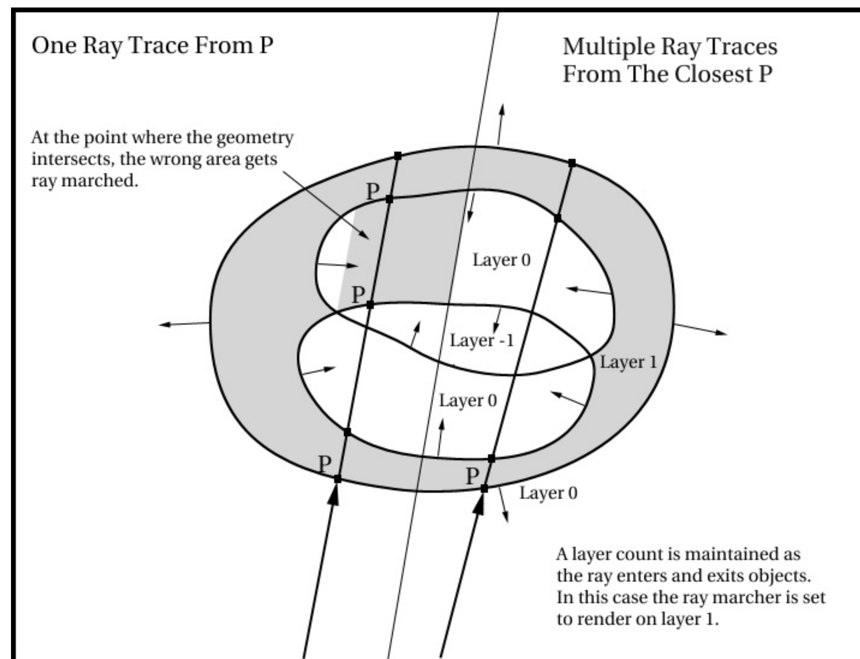

Figure 5.15: Volume booleans.

This made it possible to maintain the proper volumes even if they interpenetrated with other volumes. In the end this technique created more problems that it solved. Motion blur, displacement, and transparency became difficult to deal with. It seamed to make more sense to just insure the the surface would never interpenetrate and only do one ray march per shading point. The one reason this layered approach is mentioned, is that it can be used as a way of making Booleans on very simple shapes to get interesting volumes.

5.5.4 Moving And Deforming Geometry

Making the volume work with deforming and translating geometry was also needed to complete the shots of transforming organs. Most of the deformations on the model are usually done with clustering CV's to skeleton joints and blend shapes. The problem with these deformations working in the volume is that they are only surface effects. Each CV holds the information that tells it how to deform. What skeleton am I attached to? What weighting do I have? Where is my next blend shape position? As soon as we try to deform a sample point which lies off of the surface these questions become very unclear. Now, not only does the deformation need to be recreated, but deciding which mixture CVs get the information from needs to be figured out as well. Lattices are a lot easier to deal with inside a volume, because instead of deforming only the CVs, they deform the space and the CVs just come along for the ride. The nice thing about this is that you can add a point any where in the base lattice and it will automatically jump to its position in the deformed lattice. This means that any sample point can be deformed inside the volume and no information needs to be known about the surface. So the decision was made to deform ray marched geometry with lattices. When rendering in the volume we actually want to do the inverse of the lattice deformation. A DSO was written for RenderMan that could read the positions of the base and deformed lattice and return any point to it's undeformed position in the volume. The thought was that this DSO would need to be run at every step to accurately represents the deformed volume, but after some tests it was apparent that only the begin and end points needed to be run through the inverse lattice, then the steps could

be linearly interpolated through the volume. This worked very well for the heart's deformations because the walls were thin, so the inner volume would naturally deform in a similar way as the surface. Since the inverse deformation was now run only on the surface, Pref could be used instead, which could use other types deformation as well. This does, however, have limitations. Depending on how the object is deformed, a ray traveling through it may be a curved line, instead of a straight line from in point to exit point. If the volume is radically deformed, then running the inverse lattice at every sample point may be the better solution.

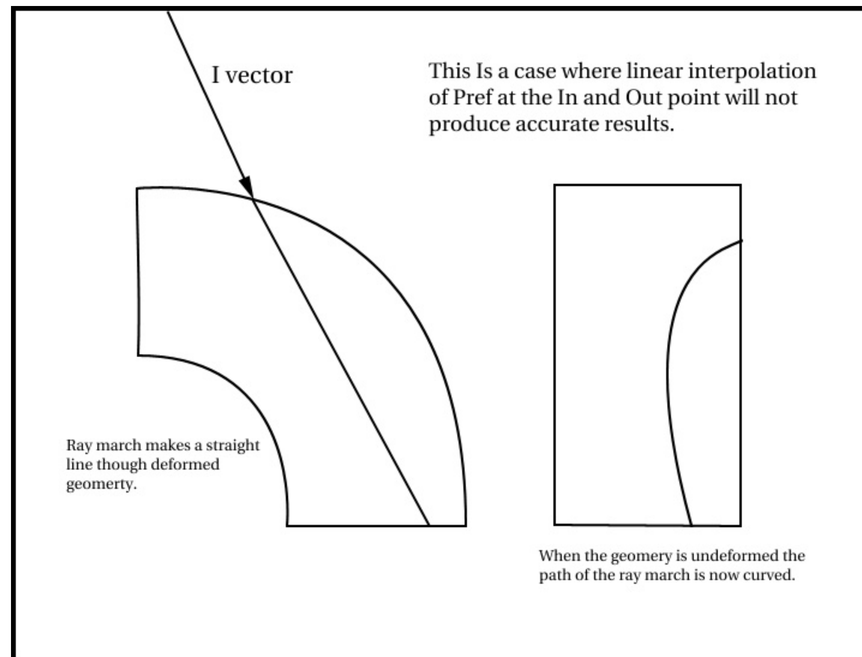

Figure 5.16: Ray marching through deformed volumes.

Motion Blur

Motion blur inside a volume can be very hard to do. One way in which it could be achieved in the volume is to sample different slices of time at everly step and average them together. Provided that you average enough slices you could get very accurate motion blur, but the render times would be extremely long. If the actual object that contains the volume is moving, we can use RenderMan's motion blur and get very good results. If the object is translating, the motion blur is fairly accurate. If the object is rotating the volume does not blur correctly. Since all the march samples are projected onto the front surface the blur will only go in one direction. If the rotation is blurred accurately, the back half should be blurring in the opposite direction than the front. Even though RenderMan's motion blur is not always accurate in the volume it looked good in most situations.

Shadow Maps

In order for our renders to fit into the plate, shadows need to be rendered as well. If the volume is very transparent ray marching the shadows may be the only option to get an accurate shadow. Although this can produce very impressive results, it is a big hit on rendering time. The other problem with ray marching shadows is that they work great for self shadowing, but not so great for cast shadows onto other objects. In the case of the heart, the volume was either totally empty or

totally solid. There were not very many semi transparent regions. This made the object perfect for shadow maps. Because shadow maps are either on or off, they are good at representing the solid volumes better than the soft semi transparent ones. When RenderMan calculates the shadow map it uses the distance to P as it's zdepth value. This is not necessarily the position where the volume becomes solid. In order to get the proper zdepth, P needs to be displaced to the sample position were the volume turned solid. This displacement can be very abrupt and will most likely tear apart the geometry, but since the displacement is done in the direction of the I vector, any tearing or stretching is unnoticeable from the rendering view.

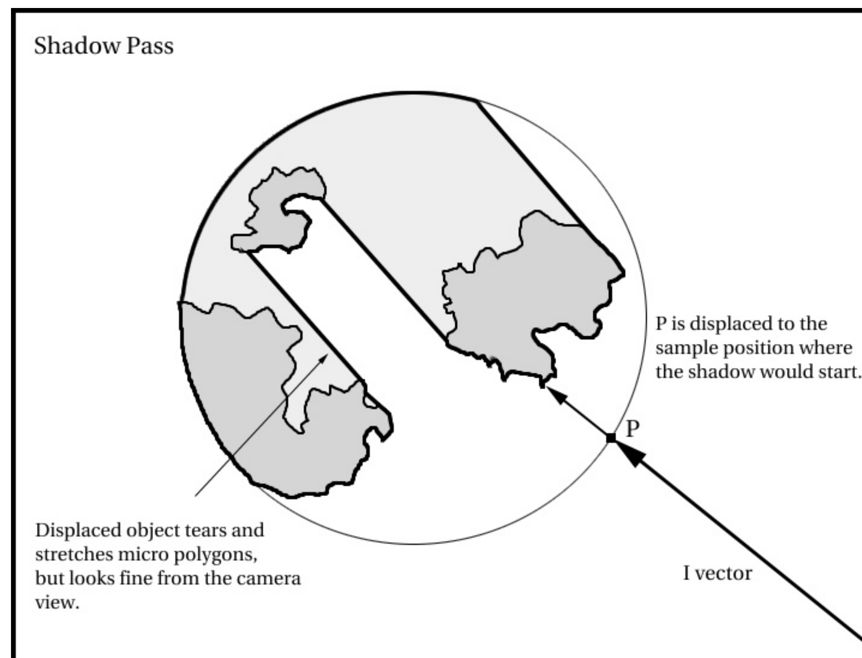

Figure 5.17: Shadow pass.

Animation In The Volume

Animation was another consideration, when we tried to figure out if ray marching would be a viable solution. We needed to have something that was controllable, and quick enough to test many iterations. Early on it was decided that curves could be used to fill the space of the volume and drive the animation. Each curve had the distance from the point of the original injection where the transformation started from. Then the animation was controlled by a parameter that told the shader how far down the blood stream the effect had gotten too. Some custom tools were written to help connect groups of curves and keep track of the arc length of all the connected curves. Individual or groups of curves could also be scaled or offset to adjust the starting point and speed at which a curve would animate through the volume. The curves were then converted into particles that would lie along the length of the curve, with it's arc length mapped to them. At render time the shader called a DSO that read through the particle and figured out which particles were within in a certain radius of the current volume sample. If the animation curve was greater than the particle's mapped arc length, then the particle receives a radius and strength which would get bigger as the animation curve got greater than the arc length of the particle. The particles are then summed together so that they blend with other particles from the same curve, and from different curves in the case of the heart. Finally the DSO returns a float that would be zero in the places where there was empty space and one were

the volume was fully on. In the case of the heart this process would be run in the function that found the `active_volume`. If the function returned a number greater the zero then the density function would be run.

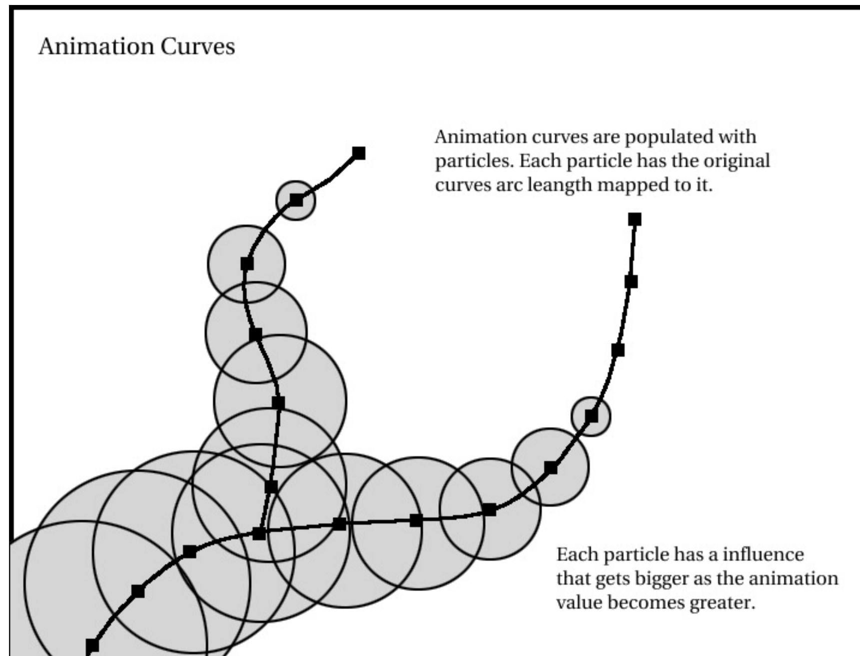

Figure 5.18: Animation curves.

The curves weren't always necessarily used for all the animations. In some causes simple falloff functions or grades were written in the shader. Then the falloff function would be transformed into coordinate systems that were animated inside Maya. This made it easy to layer more animation onto existing curve animation, or layer a bunch of these simple fields on top of each other to get a complex animation without the curves.

Volume Age

When the heart animated on, the muscle formed and filled out the volume. A few seconds later the muscle was covered over with a smooth outer casing and then a layer of fat followed. In order to run these secondary effects through the heart, the volume needed to have some sort of age associated with it. By having densities become more dense as the animation progresses we can tell which areas of the surface are older than others by looking at the density. By having densities greater than one we could trigger the secondary effects at higher values like three, ten, or a hundred. The higher the density needed to be to trigger the effect, the longer the delay would be until it happened in the animation. when the actual density of the volume was calculated a clamped version of the density was used.

5.5.5 Conclusion

Surface rendering can not always accurately represent objects that have inner volumetric detail. There are many interesting ways to to fake thickness and volume with surfaces. Displacement, transparency, and other shading tricks can often be used to create fast and believable objects that feel as if they have volume. In a production the fastest techniques are usually preferred, but sometimes

there is no good substitute for actually calculating the volume for an object. Ray marching can be a good way to accurately calculate these volumes, and it can be implemented in a surface renderer like RenderMan. In a ray marcher some features we usually take for granted may become a lot harder to implement, debugging becomes a little trickier, and render time can get a lot longer. If we are willing to bear with and/or overcome these obstacles, the final results can be well worth the effort.

Bibliography

Arvo, J., Cook, R., Glassner, A., Haines, E., Hanrahan, P., Heckbert, P., Kirk, D.; 1989. *An Introduction To Ray Tracing.* San Diego: Academic Press Limited.

Ebert, D., Musgrave, F., Peachey, D., Perlin, K., Worley, S. 1994. *Texturing And Modeling A Procedural Approach*. Cambirdge: AP Professional.

Color Atlas of Anatomy : Rohen, Yokochi and Lütjen-Drecoll Fourth Edition published by Williams & Williams.

Credits

Peter Polombi
Laurent Chabonnell
Tasso Lappas
Clint Hanson
Brian Steiner