Chapter 9: Surface Reality Techniques

Modeling and rendering of every 3-D detail of a surface is very tedious and in many applications inefficient solution. A common method for adding surface detail is to map texture patterns onto the surfaces of objects. A solid space allows the generalization of traditional solid texturing in 1-D, 2-D or 3-D domains. The solid space can be represented either by an array or by a mathematical function. Such solid spaces are either mapped onto a surface of an object in 3-D or directly projected into a screen space by using the volume rendering visualization techniques.

9.1 Mathematical Description of Solid Spaces

Solid spaces are three-dimensional spaces associated with an object that allow for control of an attribute of the object. The solid space framework encompasses traditional solid texturing, hypertextures, and volume density functions within a unified framework. Solid texturing applied to the object, is as if the defining space is being carved away similar to carving the object made from wood and marble. Solid-space examples include also the geometry defined by volume density functions (as is discussed in Chapter 11), roughness, bump mapping, reflectivity, transparency, illumination characteristics, and shadowing of objects.

Definition 9.1: Solid space is a function mapping the *m* dimensional unit cube (m=2, or 3) into the n-dimensional space, $S: D \subset (0,1)^m \rightarrow H \subset R^n$. The two- or three-dimensional solid space is defined as follows

$$S(s,t) = F, F \in R^n, n = 1, 2, 3, \ldots$$
$$S(r,s,t) = F, F \in R^n, n = 1, 2, 3, \ldots$$

With no loss of generality, we scaled the solid space coordinates *r, s,* and *t* to vary over the interval (0,1). The definition of solid space can change over time and then the time could be considered to be an additional dimension to the solid space function, *S(r,s,t,T)*. The solid space, *S* is usually a continuous function throughout definition space *D*, but it is not a necessarily requirement. Note that the choice of *F* determines the frequencies in the resulting solid spaces, and, therefore, the amount of aliasing artifacts that may appear in a final image.

9.2 The Mappings

The mapping algorithms can be thought of as modifying the shading algorithm by using the solid space to alter surface parameters, such as material properties and normals. There are three major approaches:

1. Texture mapping uses a pattern or texture to determine the color of a pixel giving detail by painting patterns onto smooth surfaces.
2. Bump mapping distorts the shape of the surface to create variations, such as the bumps or waves on water surface.
3. Environmental mapping allows us to create images that have the appearance of ray-traced images without having to trace reflected rays. As a result, the environment is painted onto the surface as that surface is being rendered.

All the methods rely on the texture being stored as a discretized two- or three-dimensional solid space.

9.3 Two-Dimensional Texture Mapping

Let us start with a two-dimensional texture pattern defined by a solid space *S(s,t)*, where variable *s*, and *t* are known as texture coordinates. We can thing of *S* as continuous, although, in discrete form it occupies the memory as an $n \times m$ array of texture elements called texels.

A texture mapping, $M: (s,t) \rightarrow (x_s,y_s)$ associates a unique point of solid space , *S* with each point on a geometric object that is itself mapped to screen coordinates (x_s,y_s) for display. Let the object be represented in spatial object coordinates *(x,y,z)* we can represent a texture map *M* as composition of a mathematical function that maps from texture coordinates to geometric coordinates, $M_T: (s,t) \rightarrow (x,y,z)$, and a projection function that maps from geometric coordinates to screen coordinates, $M_{VP}: (x,y,z) \rightarrow (x_s,y_s)$. Formally, we can write that

$$M = M_{VP} \cdot M_T.$$

Function M_T is usually used to map the rectangular area to an arbitrary region in three-dimensional space. It may be a complicated function, or may have undesirable properties, like distortion of shapes at distances.

If the geometric object is defined by using parametric surfaces, $F: (u,v) \rightarrow (x,y,z)$ we should use the two concurrent mappings, the first from texture coordinates to parametric coordinates, $M_P: (s,t) \rightarrow (u,v)$, and the second from parametric coordinates to geometric coordinates, *F*, as shown in Figure 9.1. Formally, we can write that

$$M_T = F \cdot M_P.$$

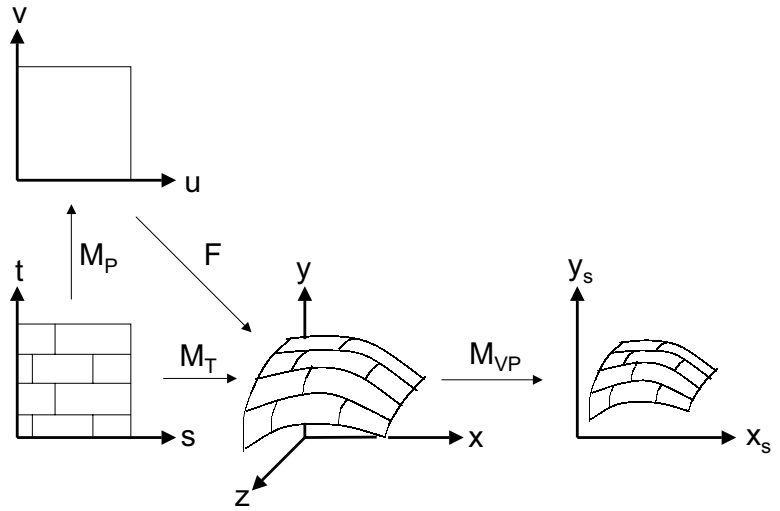


Figure 9.1. Mapping compositions used in texture mapping.

9.3.1 Forward Texture Mapping Algorithm

The forward mapping uses the map, *M*, from texture coordinates to screen coordinates. The screen space defined by screen coordinates is dicretized as an $n_s \times m_s$ array of image elements called pixels. A small rectangular area of the texture pattern *S(s,t)*, maps to the curved area in the screen space. The texture values *S(s,t)*, can then be used to either modify the color or assign a color to the pixels covered by the projected curved area in the screen space. A disadvantage of this mapping is that a selected rectangular

area in texture coordinates does not match up with the pixel boundaries in the discrete screen space, thus requiring calculation of the fractional area of pixel coverage.

9.3.2 Backward Texture Mapping Algorithm

The inverse mapping from screen coordinates to texture coordinates is the most commonly used texture-mapping method. The method avoids pixel subdivision calculation and allows antialising (filtering) procedures to be easily applied. However, the backward method requires calculation of the inverse viewing-projection transformation M_{VP}^{-1} and the inverse texture-map transformation M_T^{-1}. We are determining the color of a squared pixel centered at screen coordinates (x_s,y_s) from a corresponding curved area in texture coordinates.

One simple method is to use the point (s,t) obtained by inverse projection of the pixel center to find a texture value $S(s,t)$. Although, a simple method, it is subject to serious aliasing problem particularly if the texture is periodic, as shown in Figure 9.2. Neglecting the finite size of a pixel can lead to moiré patterns in the screen space. A better approach is to assign a texture value based on averaging of the values in the curved area in texture coordinates.

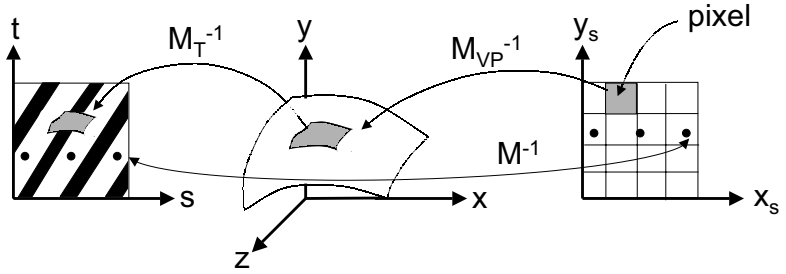


Figure 9.2. Aliasing in texture mapping.

Examples 9.1: Invertible M_P maps.

1. Given a parametric surface, we can often map a point in the texture $S(s,t)$ to a point on the surface p(u,v) by a linear map of the form

$u = as + bt + c,$

$v = ds + et + f.$

Providing that $ae \neq bd$ this linear map is invertible.

2. Linear mapping can also trivially map the texture to a parametric patch. When a parametric patch corners (u_{min}, v_{min}) and (u_{max}, v_{max}) corners correspond to the texture corners (s_{min}, t_{min}) and (s_{max}, t_{max}), as shown in Figure 9.3, we can write the mapping

$$u = u_{\min} + \frac{s - s_{\min}}{s_{\max} - s_{\min}}(u_{\max} - u_{\min}),$$

$$v = v_{\min} + \frac{t - t_{\min}}{t_{\max} - t_{\min}}(v_{\max} - v_{\min}).$$

The linear mapping is easy to implement but it does not take into account the curvature of the surface and therefore texture patches are stretched.

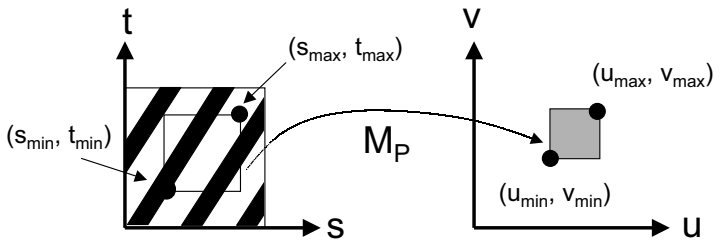


Figure 9.3. Linear mapping.

Examples 9.2: Texture mapping.

Suppose that the texture coordinates vary over the unit square, we consider the transfer of the pattern to a cylindrical surface of height h and radius r. Points on the quarter of the cylinder are given by the parametric equations $F: (u,v) \rightarrow (x,y,z)$,

$x = r\cos(u), y = r\sin(u), z = v/h, 0 \le \theta \le \pi/2.$

We cam map the pattern to the surface with the following linear transformation M_P: $(s,t) \rightarrow (u,v)$,

$u = s\pi/2, v = t.$

Next, when we perform the inverse viewing transformation from screen coordinates to the object coordinates. Object coordinates are then mapped to the surface parameters with the inverse transformation, F^{-1},

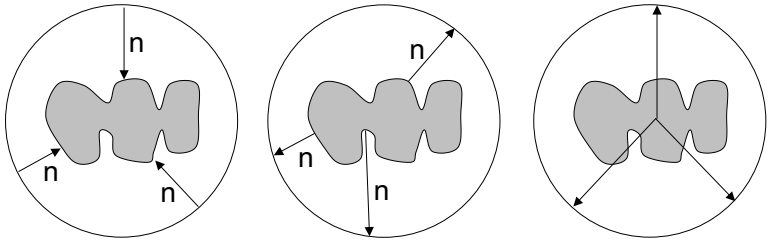
$u = \tan^{-1}(y/x), v = z*h,$

and to the texture space with, M_P^{-1},

$s = 2u/\pi, t = v.$

9.4 Two-Part Texture Mapping

The two-part mapping uses the intermediate surface, such as a sphere, cylinder, or cube for texture mapping of complicated object surfaces. In the first step, the texture is mapped to the intermediate shape. In the second step, the intermediate surface containing the texture is mapped to the object surface being rendered. There are three possible strategies to perform the second step. First method takes the value of a texture and goes in the direction of the intermediate surface normal until the object is intersected, and he color o f the intersection point is the color of the texture. The second method uses the inverse approach, starting at the object surface and going in the object surface normal direction until we interest the intermediate object, where we read the texture value. The third possibility is to draw the lines from the center of the object and find both the intersection with object and intermediate surface. The texture at the point of intersection with the intermediate surface is assigned to the intersection point on the object as shown in Figure 9.4.



a) b) c)

Figure 9.4. Two-part mapping. (a) Going in the direction of intermediate surface normal. (b) Going in the normal of he object surface. (c) Going in the direction of central line.

9.5 OpenGL Texture Mapping

The OpenGL contains the functionality o map one- and two-dimensional textures to one- through four-dimensional graphical objects. The texture mapping is done, as primitives are rasterized by mapping three-dimensional points to locations on the display. Each fragment of the object is tested for visibility and is shaded if visible. Suppose that we have a 512×512 my_texels representing our discretized solid space

```
glubyte my_texels[512][512];
```

We specify that this array is to be used as a two-dimensional texture

```
glTexImage2D(GL_TEXTURE_2D, level, components, width, height, border, format, type, my_texels);
```

The format of image having three color components R, G, B is described by GL_RGB, the type is GL_UNISIGNED_BYTE, and the value components is equal to 3 for RGB components. To enable he texture mapping we use

```
glEnable(GL_TEXTURE_2D);
```

Other part of setting up a texture mapping is to specify the mapping of the texture onto a geometric object. We can do this by assigning the correspondence between corner texture coordinate and the object coordinate of a quadrilateral by the following code

```
glBegin(GL_QUAD)
 glTexCoord2f(0.0,0.0);      /* Assign the texture coordinate (0,0) */
 glVertex2f(x1, y1, z1);     /* to (x1, y1, z1) object coordinates. */
 glTexCoord2f(1.0,0.0);
 glVertex2f(x2, y2, z2);
 glTexCoord2f(1.0,1.0);
 glVertex2f(x3, y3, z3);
 glTexCoord2f(0.0,1.0);
 glVertex2f(x4, y4, z4);
glEnd();
```

We also could set new normal or colors before we specify each vertex. When projecting the pixel to the texture coordinates, it can be smaller or larger than one texel. To avoid this problem of aliasing the OpenGL uses a 2×2 average on the four closest texels if we specify

```
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
```

A better technique but more memory consuming is called mipmapping. We do not need the resolution of the original texel array, for objects that project to an area of screen space that is smaller compared with the size of the texel array. In OpenGL we can generate texture array at reduced sizes automatically. For example a 64×64 texture array, can be reduced to series of arrays with sizes 32×32, 16×16, 8×8, 4×4, 2×2, and 1×1 by a function

```
gluBuild2Dmipmaps(GL_TEXTURE_2D,3,64,64,GL_RGB,GL_UNSIGNED_BYTE,my_texels);
```

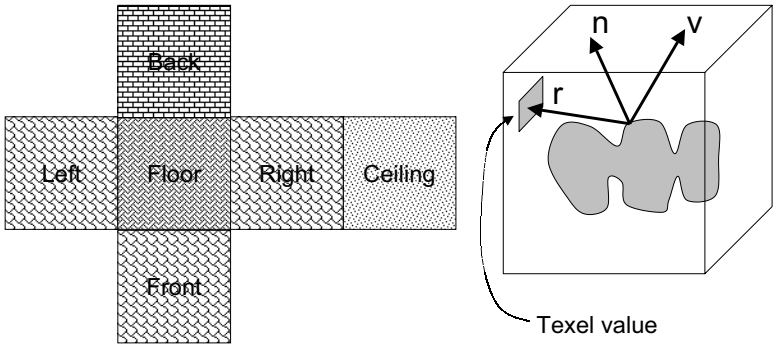
The mipmaps are put in use automatically if we specify

```
glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST_MIPMAP_NEAREST);
```

Advance texture features provide the possibility to map surface textures directly onto a three-dimensional mesh. OpenGL can also provide a spherical mapping, as we discussed in the Section for two-part texture mapping.

9.6 Environment Mapping

High reflective surfaces are characterized by specular reflections that mirror the environment. For example, a shiny ball in the center of a room reflects all the walls of the room. We can extent the texture mapping techniques to project the reflected environment onto the object surface. The idea is similar to two-part texture mapping. At first, we obtain an image of the environment on an intermediate projection surface; usually a box is used for environments such as a room. The environmental image is obtained by putting the center of projection at the center of the reflective object. An example of environmental image of a room projected on the box is shown in Figure 9.5. Similar to two-part texture mapping, the texture value at a point on the object is obtained from the corresponding point on the intermediate surface calculated from the reflection vector, *r* and location of the viewer (view direction *v*), Figure 9.5.



a) b)

Figure 9.5. Environmental mapping with a box. a) Image of the environment. b) Environmental mapping from the intermediate surface

In OpenGL we invoke this algorithm by the lines

```
glTexGenfv(GL_S, GL_SPHERE_MAP, 0);
glTexGenfv(GL_T, GL_SPHERE_MAP, 0);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
```

9.7 Bump Mapping

The real strawberry is characterized primarily by small variations in its surface, rather than by variations in its color. The bump mapping technique generates small variations of the surface by perturbing the normal vectors as the surface is rendered; the highlights and colors than give the impression of surface variations. This method is based on the idea that the surface shape at a point is characterized by the normal at this point and small neighborhood. Small perturbation to the normal vectors applied on a smooth surface gives the appearance of a complex surface.

Let $\boldsymbol{P}(u,v) = (x(u,v),y(u,v),z(u,v))$ be appoint on a parametric surface, the unit normal vector at that point is derived from a cross product

$$\mathbf{n} = \frac{\mathbf{P}_u \times \mathbf{P}_v}{|\mathbf{P}_u \times \mathbf{P}_v|},$$

where $\mathbf{P}_u = \left(\frac{\partial x}{\partial u}, \frac{\partial y}{\partial u}, \frac{\partial z}{\partial u}\right)$, and $\boldsymbol{P}_v$ are the partial derivatives of P with respect to parameters u and v. Suppose we displace the surface in he normal direction by a function called the bump function, d(u,v):

$\boldsymbol{P}' = \boldsymbol{P} + d(u,v)\boldsymbol{n}$.

We prefer to use only the normal of perturbed surface in shading calculations obtained as

$\mathbf{n}' = \boldsymbol{P}_v' \times \boldsymbol{P}_v'$.

Where the perturbed partial derivatives are obtained as

$$\mathbf{p}'_u = \mathbf{p}_u + \frac{\partial d}{\partial u}\mathbf{n} + d(u,v)\mathbf{n}_u,$$

$$\mathbf{p}'_v = \mathbf{p}_v + \frac{\partial d}{\partial v}\mathbf{n} + d(u,v)\mathbf{n}_v.$$

Assuming the $d(u,v)$ is small, we can neglect the last term of both equations. Substituting the equation into the above cross product and noting that $\mathbf{n} \times \mathbf{n} = 0$ we obtain the perturbed normal

$$\mathbf{n}' = \mathbf{n} + \frac{\partial d}{\partial u}\mathbf{n} \times \mathbf{p}_v + \frac{\partial d}{\partial v}\mathbf{n} \times \mathbf{p}_u.$$

The final step is to normalize **n'** for use in the shading model.

9.8 Blending and Compositing Techniques

The mechanism called alpha (α) blending can among other effects, create images with transparent objects. The alpha channel is the fourth color in RGBA (or RGBα) color mode that controls how the RGB values are written into the frame buffer. In the solid space terminology we write $S(s,t) = F = (RGBA)$, $F \in R^4$.

The opacity of a surface is a measure of how much light penetrates through that surface. An opacity (α) equal to 1 corresponds to a completely opaque surface that blocks all light, while an opacity of 0 is completely transparent surface. Having several polygons with defined alpha channel, the combination of their color is similar to joining two pieces of colored glass into a single piece of glass that has higher opacity and a color different from either of the original pieces.

Let us have two texels represented with four elements (RGBα) $\boldsymbol{s} = [s_r, s_g, s_b, s_a]$, $\boldsymbol{d} = [d_r, d_g, d_b, d_a]$, then a blending operation replaces **d** with

$\boldsymbol{d'} = [bs_r + cd_r, bs_g + cd_g, bs_b + cd_b, bs_a + cd_a]$.

The constants b and c are the source and destination blending factors. As it can occur after blending the colors, a value of $\alpha > 1$ is limited (clamped) to the maximum of 1.0, and $\alpha < 0$ is clamped to 0.0.

Usually, we wish to keep our RGB colors between 0 and 1 in the final image, without having to clamp those values greater than 1. Suppose, we have n images that should contribute equally to the final display. We can either scale the values of each texel, or use the source and destination blending factors by $1/n$.

In OpenGL we enable blending by

```
glEnable(GL_BLEND);
```

The desired source and destination factors can be set up by

```
glBlendFunc(source_factor, destination_factor);
```

Many applications use the source factor α and the destination factor 1-α. The resulting color in OpenGL is

$$(R_{d'}, G_{d'}, B_{d'}, \alpha_{d'}) = (\alpha_s R_s + (1-\alpha_s)R_d, \alpha_s G_s + (1-\alpha_s)G_d, \alpha_s B_s + (1-\alpha_s)B_d, \alpha_s\alpha_s + (1-\alpha_s)\alpha_d).$$

This formula ensures that both transparent and opaque polygons are handled correctly and that neither colors nor opacities can saturate. The major difficulty with this blending technique is that the order in which we render the polygons affects the image. To get a desired effect, we must now control this order within the application.

9.9 Fog

We can create the illusion of depth by drawing textures father from the viewer dimmer than textures closer to the viewer, a technique known as depth cueing. A simple implementation uses the α blending technique.

A fog creates the illusion of partially translucent space between the object and the viewer, by blending each texture fragment in a distance dependent manner. If the texel has a value $\mathbf{C_s}$ and the color of fog is $\mathbf{C_f}$, then the final color of pixel in screen coordinates is

$$\mathbf{C}_{s'} = f\mathbf{C}_s + (1-f)\mathbf{C}_f.$$

The depth-cueing effect is obtained if f varies linearly depending on the distance from eye position. Fog density function is the exponential function of distance, z, from eye position

$$f = e^{-0.5z^2}.$$

OpenGL supports the fog densities in RGBA mode by using the function calls

```
GLfloat fcolor[4] = { …}
glEnable(GL_FOG);
glFogf(GL_FOG_MODE, GL_EXP);
glFogf(GL_FOG_DENSITY, 0.5);
glFogfv(GL_FOG_COLOR, fcolor);
```

9.10 Volumetric Rendering of Solid Spaces

Let us focus now on the three-dimensional solid space $S(r,s,t)$. We can thing of S as continuous, although, in discrete form it occupies the memory as an $n \times m \times l$ array of elements called voxels. Function S can be defined by procedural methods or by using the three-dimensional volume density functions that define the density of a continuous three-dimensional space. The volumetric density function is the natural extension of solid texturing to describe the actual geometry of objects. They are extensively used in computer graphics for modeling and animating gases, fire, fur, liquids, and other soft objects.

For true three-dimensional images and solid spaces, volume rendering must be performed. Volume rendering is a mapping, $M: (r,s,t) \rightarrow (x_s,y_s)$ which associates a unique point of solid space, S with each point on a screen coordinates (x_s,y_s) for display. We will discuss the method, which is described in detail in Ebert and Parent 1990. The ray

from eye through the pixel (x_s,y_s) is traced through the defining solid space. For each increment (voxel) along the ray through the volume sections, the solid space function is evaluated. The color, density, opacity, shadowing, and illumination of each sample are then accumulated based on illumination model for gases and atmospheric attenuation.

9.10.1 Algorithm: Volume Rendering

```
for each section of volume
        for each increment (voxel) along the ray
                get color, density, opacity of this voxel
                if (self shadowing)
                   retrieve the shadowing of this element from the solid shadow table
                color = calculate the illumination of the volume using the
                          opacity, density, and the appropriate model
                final_color = final_color + color;
                sum_density = sum_density + density;
                if ( transparency < 0.01 )
                   stop tracing
        increment sample point (voxel)
create a buffer fragment
```

9.10.2 Illuminating of Gaseous Phenomena

The opacity is the density obtained from evaluating the volume density function $\rho(x(u),y(u),z(u))$ multiplied by the step size Δu along the ray approximated as

$$opacity = 1 - \exp\left(-\tau \cdot \sum_{u_{near}}^{u_{far}} \rho(x(u), y(u), z(u)) \cdot \Delta u \right),$$

where τ is the optical depth of material, u_{near} is the entering point of the ray to the volume, u_{far} is the ending point. The density function is defined in the solid space, $\rho(x(u),y(u),z(u)) = S(r,s,t)$.

The intensity of a pixel (x_s,y_s) in screen coordinates is calculated by the following illumination model

$$B = \sum_{u_{near}}^{u_{far}} \exp\left(-\tau \cdot \sum_{u_{near}}^{u} \rho(x(u'), y(u'), z(u')) \cdot \Delta u' \right) \times I \cdot \rho(x(u), y(u), z(u)) \cdot \Delta u,$$

$$I = \sum_i I_i(x(u), y(u), z(u)) \times phase(\theta).$$

$I_i(x(u),y(u),z(u))$ is the amount of light from light source i reflected from voxel (x,y,z), $phase(\theta)$ is the phase function characterizing the total brightness of a voxel as a function between the light and the eye. Self-shadowing of the gas is incorporated into I by attenuating the brightness of each light source.

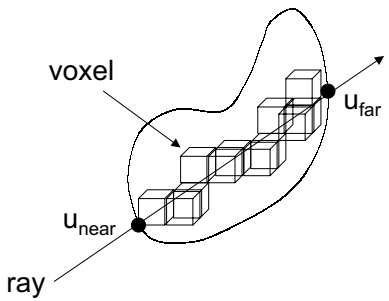


Figure 9.6. Volume rendering of solid spaces.

9.11 Geometry of Gases

After some background material has been discussed, this section will describe detailed procedures for modeling gases. The geometry of gases is modeled using turbulent-flow-based volume density functions. For each location in the solid space, we generate the noise and apply the turbulence function. The noise implementation uses trilinear interpolation of random numbers stored at the lattice points of regular grid to calculate the noise for any point.

The *turbulence()* function given below is the standard turbulence function:

```
float turbulence(pnt, pixel_size)
        xyz     pnt;
        float   pixel_size;
{
  float t, scale;
  t=0;
  for(scale=1.0; scale > pixel_size; scale/=2.0)
    {
      pnt.x /= scale; pnt.y /= scale; pnt.z /= scale;
     t += calc_noise(pnt) * scale;
    }
  return(t);
}
```

Several basic mathematical functions are used to shape the geometry of the gas. The simplest shape function is the power function

$$density = (t * p_1)^{p_2},$$

where t is turbulated point of the solid space, and p_1, p_2 are parameters. In the following example of a code, we demonstrate a simple gas procedure:

```
basic_gas(pnt, density, parms)
        xyz     pnt;
        float   *density, *parms;
{
  turb = turbulance(pnt, pixel_size);
  density = pow(turb*parms[1], parms[2]);
}
```

This procedure takes as input the location of the point being rendered in the solid space, *pnt*, and a parameter array. The returned value is the density of the gas. *Parms[1]* is the maximum gas density within the range (0.0, 1.0), and *Parms[2]* is the exponent for the power function.

9.12 Smoke in OpenGL

One easy technique involves capturing a two-dimensional cross section or image of a puff of smoke with both luminance and alpha channels for the image. The image can then be texture mapped onto a quadrilateral and blended into the scene. The color and alpha value of the quadrilateral can be used to control the color and transparency of the smoke in order to simulate different types of smoke. The size, position, orientation, and opacity of the quadrilateral can be varied as a function of time to simulate the puff of smoke enlarging, drifting and dissipating over time as shown in Figure 9.7.

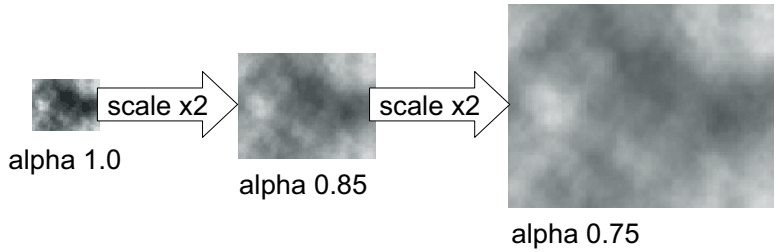


Figure 9.7. Scaling the image of a smoke.

9.13 Vapor trails emanating from a jet or a missile

A texture image consisting only of alpha values is used to modulate the alpha values of a white billboard polygon. The trajectory of the airborne object is painted using multiple overlapping copies of the billboard as shown in Figure 9.8. Over time, the individual billboards gradually enlarge and fade. The program for rendering a trail requires maintaining an active list of the position, orientation and time since creation for each billboard used to paint the trail.

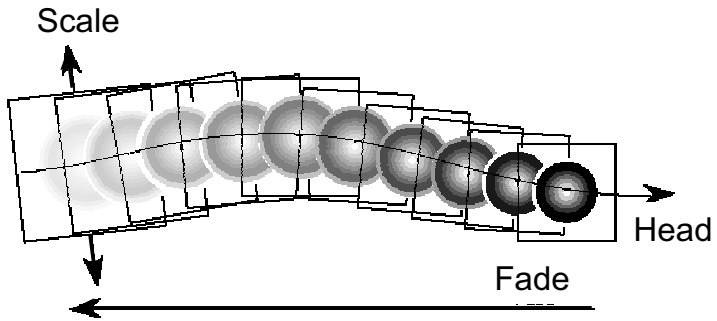


Figure 9.8. Vapor trails.

Exercises

9.1 Write a program to implement texture mapping for the spherical surfaces and for polyhedrons.
9.2 Given a spherical surface, write a bump mapping procedure to generate the bumpy surface of an strawberry.
9.3 Write a bump-mapping routine to produce surface-normal variations for any specified bump function.
9.4 How is an image produced with an environmental map different from a ray-traced image of the same scene?
9.5 In the movies, the wheels of cars often appear to be spinning in the wrong direction. What causes the effect? Can anything be done to fix this problem? Explain your answer.
9.6 Why do the patterns of striped shirts and ties change as an actor moves across the screen of you television?
9.7 Why should we do anitialiasing by preprocessing the data, rather than by post processing them?
9.8 Suppose that we have translucent surfaces characterized by opacities α_1 and α_2. What is the opacity of the translucent material that we create by using the two in series? Give an expression for the transparency of the combined material.
9.9 Devise a method of using texture mapping for the display of voxel data.
9.10 Suppose that a set of objects is texture mapped with regular patterns such as stripes and checkerboards. What is the difference in aliasing patterns that we would see when we switch from parallel to perspective views?
9.11 Consider a scene composed of simple objects, such as parallelepipeds, that have different sizes. Suppose you had a single texture pattern and you are asked to map this texture to all the objects. How would you map the texture so that the pattern would be the same size on each face of the object?
9.12 Write a program to generate the gas solid space and write the volume visualization program of generated solid space. Describe the noise, turbulence, shaping and ray tracing techniques you have used.