Chapter 5 Advanced Mapping techniques and Ray Tracing on the GPU

Introduction

In Chapter 3 we described how to implement bump mapping – the popular and long lived method that enables apparent surface detail to appear on the surface of a rendered object without perturbing the surface geometry. In this chapter we will look at ways to expand this illusion. We want to do this by retaining the overall advantage of bump mapping – that it is a texture map approach. In particular we want to be able to map relief textures onto an arbitrary polygonal model.

We will also in the course of this chapter see that we can use GPUs to implement algorithms such as ray tracing. By that we mean GPUs can be used for algorithms other than shading equations and their associated geometric calculations. It is the case that GPUs are beginning to be used for applications that lie outside the rendering field – fluid mechanics is an example [MARK04]. The motivation here is the greater power of the hardware c.f. the CPU. This culture has come to be known as GPGPU or General Purpose computing on a GPU.[1]

The following sections show the evolution of methods that cache detailed surface geometry in texture maps. Specifically height maps (or equivalently depth maps). These are presented in approximate order of complexity and cost. A particular method can be chosen depending on the demands of the application. The series also demonstrates the evolution and refinement of a single idea which is how much of computer graphics research progresses.

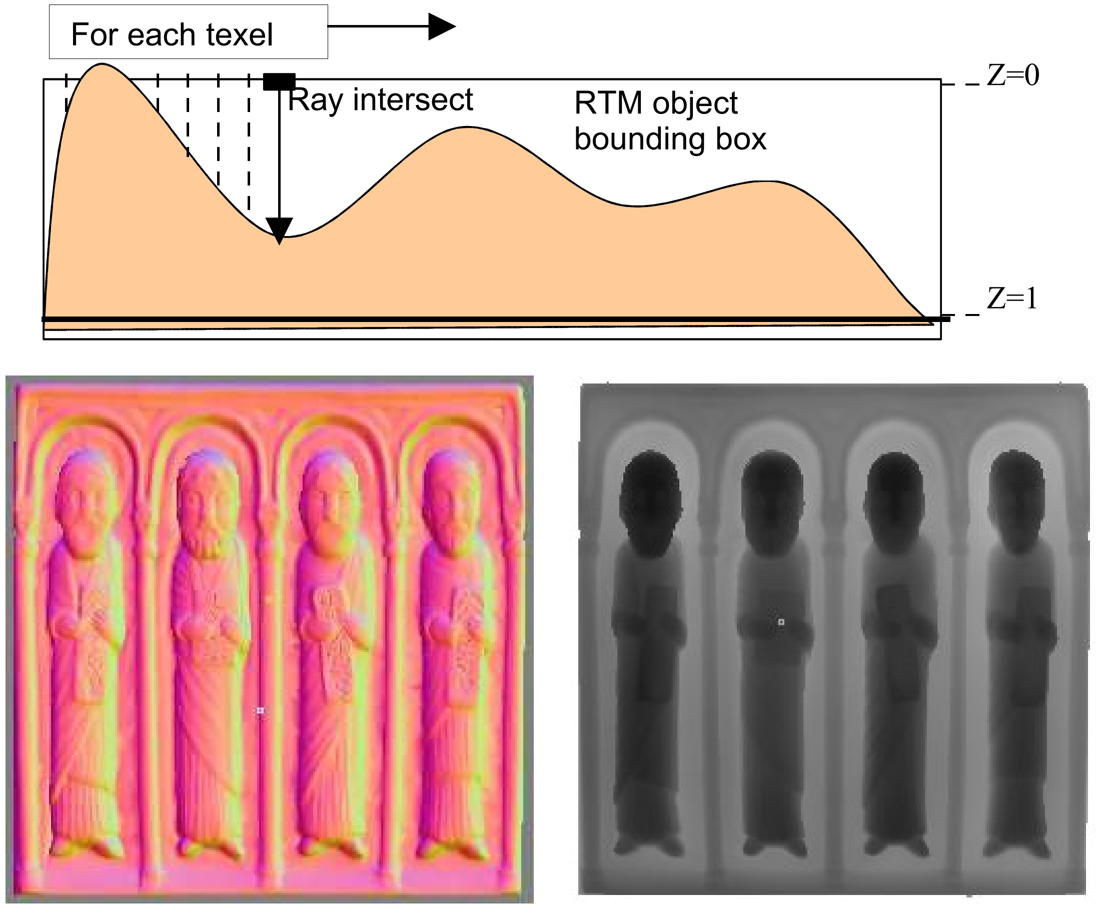
The methods will demonstrate that by including depth in textures we can design methods that result in much higher image quality than is normally achievable with maps. Depth is calculated and stored per texel and this is the main difference bewteen these techniques and conventional texture/bump mapping. The aim of all the described methods is to render small scale surface geometry using texture maps.

Connected with most of the described procedures are ray tracing methods. Although these part of the mapping algorithms they can be used in other applications.

Depth and Normal Maps

In the methods that follow we store the render attributes – colour, normal and depth in texture maps. Normal and depth map calculations are carried out off-line. Figure 5.1 shows the idea. We consider a depth map to be coplanar with a bounding box face of the object and invoke a parallel ray cast to find the depth at each pixel. This is subsequently normalised by the depth of the bounding box and stored in byte format. The normal is calculated at the same time from the surface geometry. This is identical to the normal calculation method used in Chapter 3 except for the addition of the derivation of the normalised depth map.

[1] See www.gpgpu.org for examples and links to applications.



Normal map (RGB)

Depth Map (ALPHA)

Figure 5.1
Calulating a normal and depth map for a 'bas relief' object.

A general problem with the technique, which is obvious from the illustration, is that we can only deal, in this implementation, with convex objects. We can only 'recover' information in the render phase that has been obtained in the pre-calculation phase. A parallel ray cast will not necessarily detect all concavities. It can only 'see' those that are visible in a viewing direction equal to the ray cast direction.

Having represented the surface detail geometry in this way we can invoke calculations such as ray tracing in this bounding box space, or a transformation of it. Effectively the depth and normal map replaces the geometry and we define an object as a texture map (RGBA) where the normal map is stored in RGB and the depth of the surface from the plane of the bounding box is stored in ALPHA as shown in Figure 5.1.

Parallax mapping

Parallax mapping aka offset texture mapping [KANE01] is the simplest of the depth oriented techniques. The rendering attributes for the object, normal and colour, are stored in texture maps. The principle is shown in Figure 5.2 where a view vector is intersecting the surface of an object in tangent space. For any view ray we need to find the intersection with the surface. The information we have at the fragment is the depth of *P* and we use this to find an approximation of the desired intersection *P''*.

In a fragment program we make two texture accesses, the first one gets the depth information *d* at the fragments position in texture space. Then we offset the texture coordinates for the fragments by the product of this value and the view vector in tangent space. The normal and colour are now fetched using the offset texture coordinates. Offsetting the coordinates causes the normal and colour at point *P'* to be used. This is an approximation to the correct point *P''* but we hope that P' is closer to *P''* than *P*.

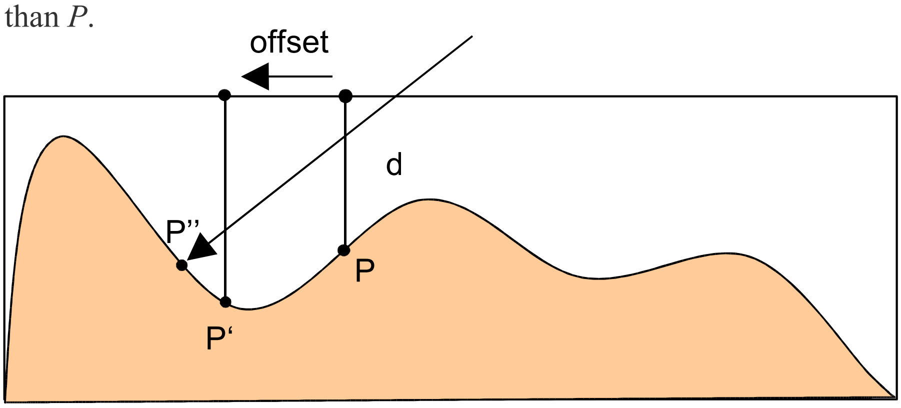


Figure 5.2
Parallax mapping – point *P'* is rendered

The Cg implementation of this technique is given in Listing 5.1

```
f2s main_frag_parallax(
      v2f IN,
      uniform sampler2D rmtex:TEXUNIT0,         // relief map
      uniform sampler2D colortex:TEXUNIT1,      // color map
      uniform float4 ambient,       // ambient color
      uniform float4 diffuse,       // diffuse color
      uniform float4 specular,      // specular color
      uniform float tile)                 // tile factor
{
      f2s OUT;

        // view and light directions
      float3 v = normalize(IN.vpos);
      float3 l = normalize(IN.lightpos.xyz-IN.vpos);

      float2 uv = IN.texcoord*tile;

      // parallax code
      float3x3 tbn = float3x3(IN.tangent,IN.binormal,IN.normal);
      float height = tex2D(rmtex,uv).w * 0.06 - 0.03;
      uv += height * mul(tbn,v).xy;

      // normal map
      float4 normal=tex2D(rmtex,uv);
      normal.xyz=normal.xyz*2.0-1.0; // trafsform to [-1,1] range

      // transform normal to world space
      normal.xyz=normalize
(normal.x*IN.tangent+normal.y*IN.binormal+normal.z*IN.normal);

      // color map
      float4 color=tex2D(colortex,uv);

      // compute diffuse and specular terms
      float att=saturate(dot(l,IN.normal.xyz));
      float diff=saturate(dot(l,normal.xyz));
      float spec=saturate(dot(normalize(l-v),normal.xyz));

      // compute final color
      float4 finalcolor;
      finalcolor.xyz=ambient.xyz*color.xyz+
        att*(color.xyz*diffuse.xyz*diff+specular.xyz*pow
(spec,specular.w));
      finalcolor.w=1.0;

      OUT.color=finalcolor;

      return OUT;
}
```

Listing 5.1
The fragment program for parallax mapping

An image using this shader is presented in Figure 5.9 where it is compared with a bump or normal map render and a RTM render. Although as we discuss later, there are artefacts, for the cost of one extra texture access and simple arithmetic, parallax mapping gives a significant visual image improvement over normal or bump mapping. However, because of its inaccuracy it is only suitable for irregular surface detail where the visual effect of the inaccuracies are less noticeable. In the next section we describe how to use depth enhanced texture maps in a more accurate manner.

Relief Texture Mapping (RTM)

Basic algorithm

This method, introduced by Oliveira *et* al [OLIV00] was named Relief Texture Mapping (RTM) and its goal is again the rendering of complex surface 3D detail using a depth enhanced texture map as rendering data. A fragment program uses the map together with the view direction to generate different views of the object.

Figure 5.3 show views from different angles generated from a single texture map containing normal and depth data. A single square polygon (a quad) is being drawn and the program also performs per-fragment lighting calculations with diffuse and specular components.

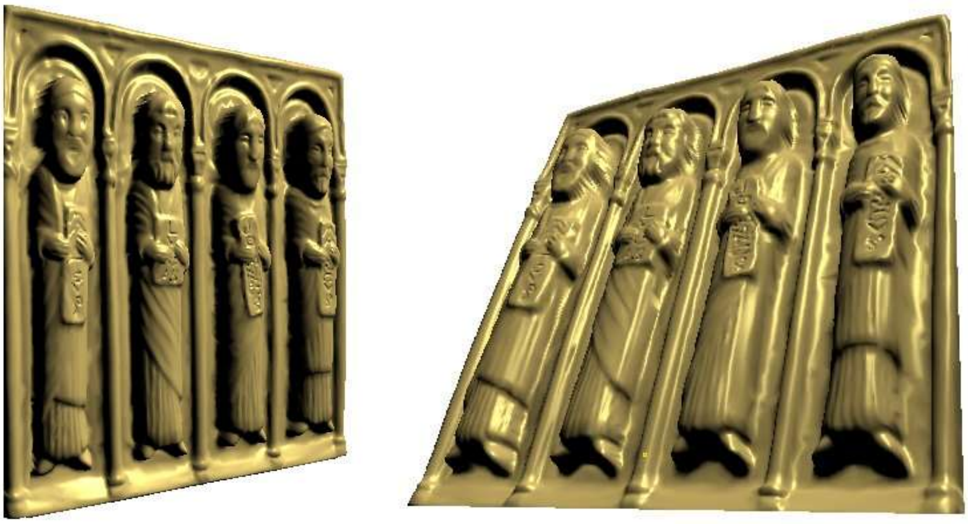


Figure 5.3
Rendered RTM object

Now the object shown in the preceding illustrations is effectively a $2\frac{1}{2}$ D object which we wish to view from the front. Thus a single RTM will suffice in this case. If we want to view a 3D object from any angle then we require 5 or 6 RTMs as Figure 5.4 demonstrates.

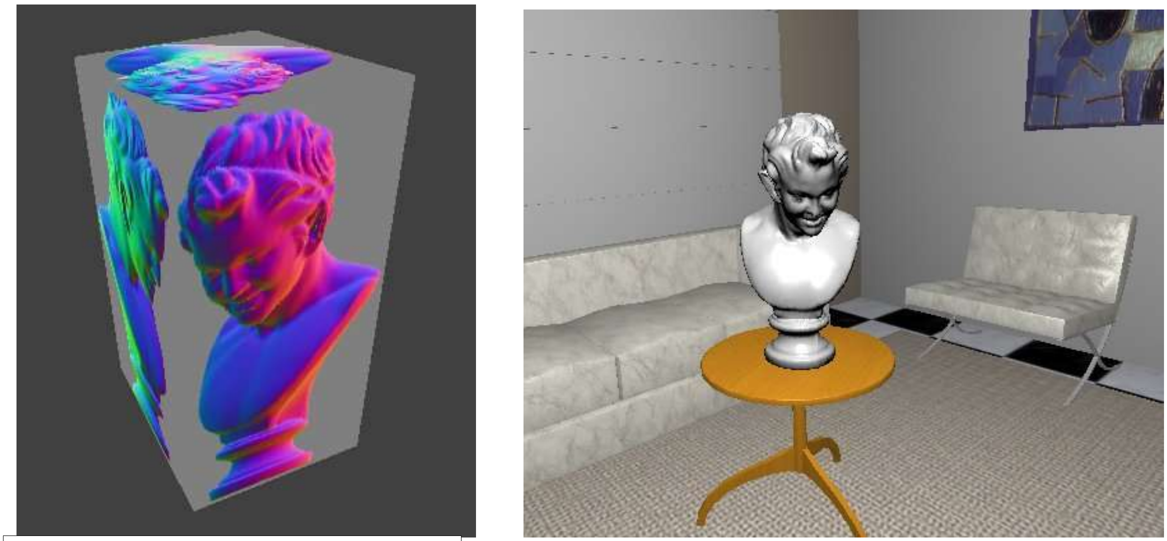


Figure 5.4
An object which can be viewed from front, back, left side, right side and top need 5 RTMs.

In this case we never want to view from below so we only have 5 RTMs. A maximum of 3 RTMs will be used at any given time in the render.

We now describe the algorithm. The technique as originally reported in [OLIV00] was an off-line software implementation and used forward mapping. Our implementation is real-time and uses inverse mapping. The difference between forward and inverse mapping in this context will be explained later. To render the RTM we use an inverse mapping process where, for each pixel, we must compute what RTM texel must be moved to that pixel. (This contrasts with forward mapping, where for each texel in the RTM map, we compute its destination pixel and move the texel there.)

Thus for a given pixel (A) in the RTM and a camera view direction we define a ray (Figure 5.5). This ray produces an intersection point at the surface and we assign the colour calculated at that hit point from the corresponding RTM texel (B) to A. Inverse mapping is required because in a fragment shader each pixel must compute its colour.

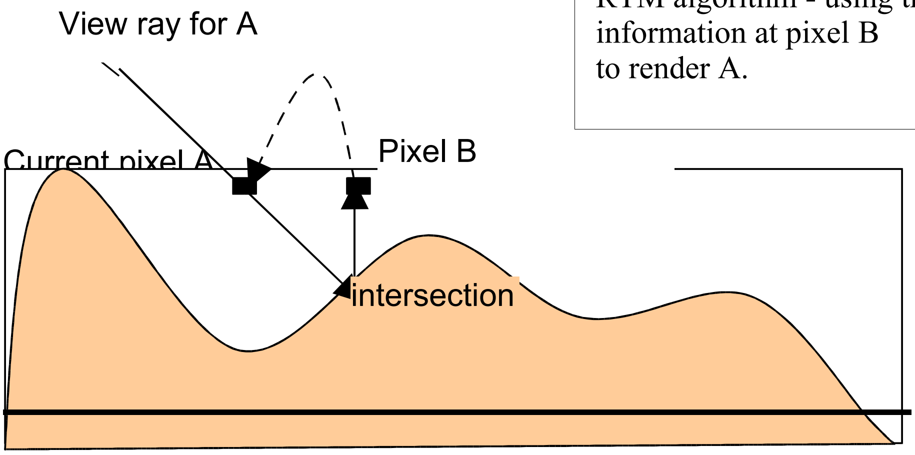


Figure 5.5
RTM algorithm - using the information at pixel B to render A.

The remaining question is: how do we find the intersection point of the view-pixel ray with the object? We will show three ways of solving this which depend on the nature of the object.

RTM – Multi pass for planar objects

In this approach we divide the object's depth information into 'depth slices' or planes parallel to the RTM plane. This produces what is known as a layered depth image as shown, for example in Figure 5.6. In this example the view ray through pixel A produces 3 intersection points on the 3 slices.

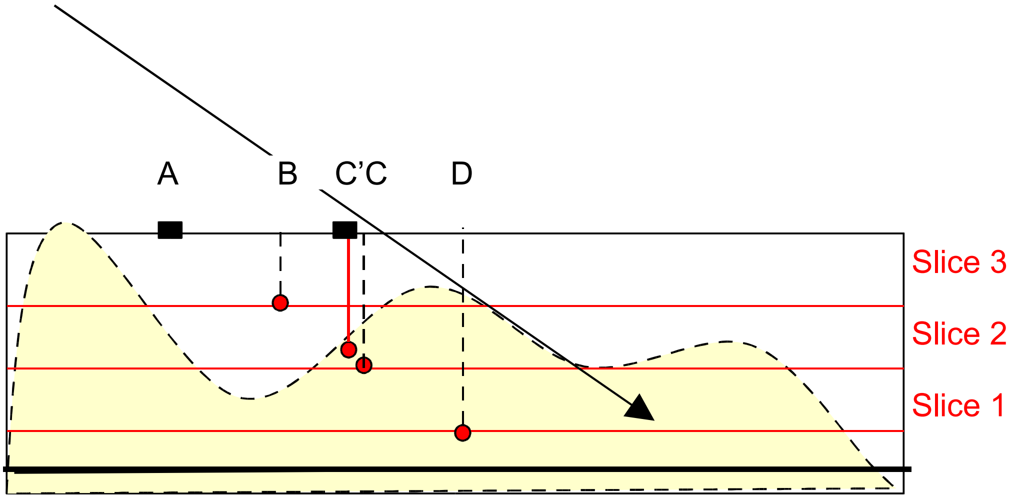


Figure 5.6
Inverse mapping to assign a render colour to the current pixel A. In this example C is selected incurring an error. The correct pixel is C'.

To find the ray slice intersects we effect as many passes as there are slices. Again given the current pixel A we ideally need to find the RTM texel C; corresponding to the intersection point. In this particular example the scheme would work as follows:

Pass 1, Slice 1

Ray intersects plane 1.
Depth is greater than RTM object depth at D
Render using information at D

Pass 2, Slice 2

Ray intersects plane 2.
Depth is greater than RTM object depth at C
Render using information at C

Pass 3, Slice 3

Ray intersects plane 3.
Depth is less than RTM object depth at B
Discard

The net result in this case is that A is rendered using the information at C. This incurs an error as the correct pixel is C'. The magnitude of the overall error is a function of the number of slices or passes.

The total number of passes required is thus:

(number of RTMs used) x (number of slices)

One of the advantages of depth slicing is that it provides useful options. Since the cost of the pre-warp is a function of the number of passes or slices we can easily set up a LOD scheme. Figure 5.7 shows 4 LODs obtained by varying the number of slices.

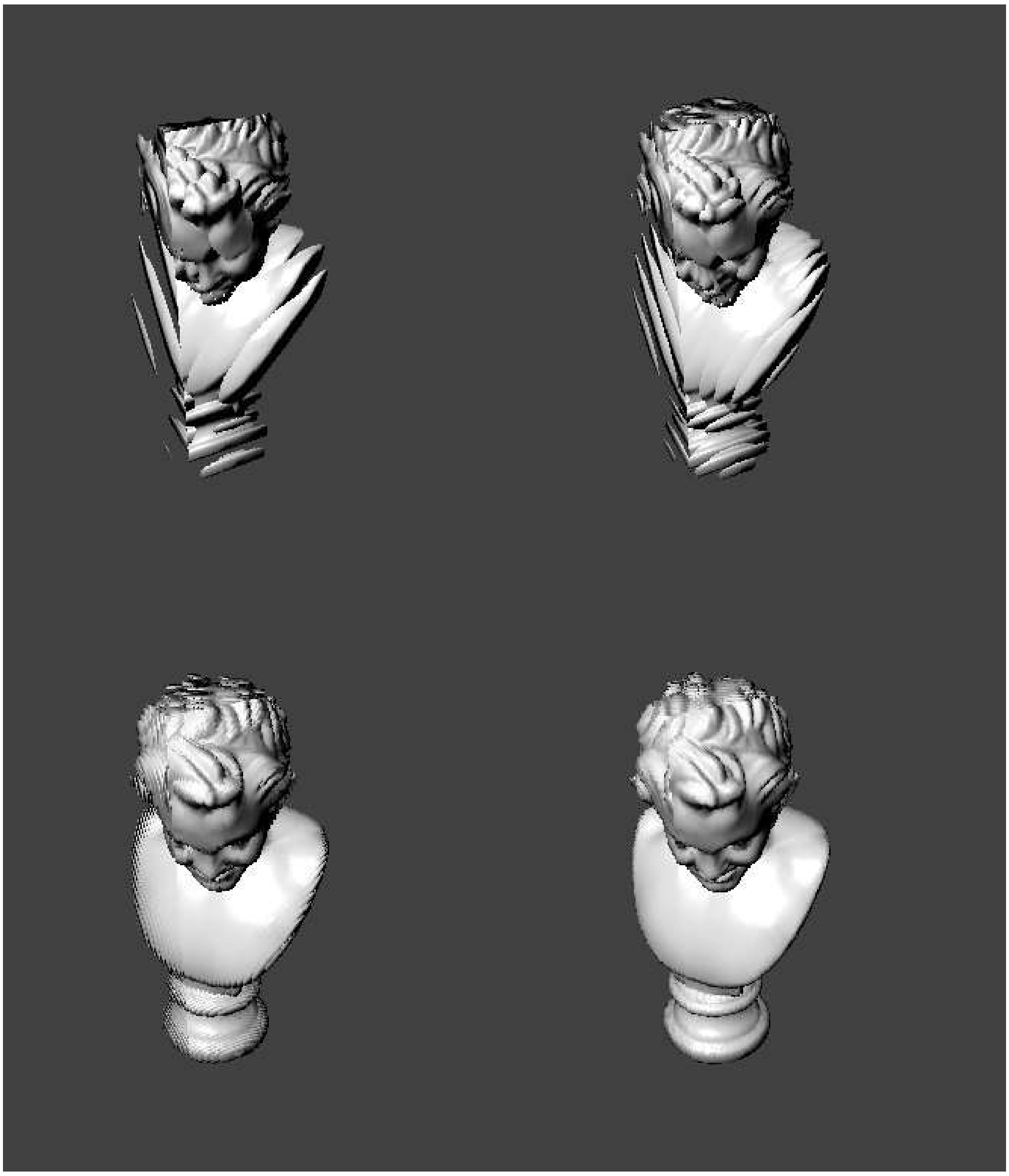


Figure 5.7
Rendering the head from 3 RTMs using a varying number of depth slices

As is evident from the illustration the slices exhibit strong coherence at a low slice resolution. Another option available using this algorithm is to define slices through the depth field that are normal to the view direction. Rendering the head using view aligned depth planes approach is compared to the RTM face aligned depth planes approach in Figure 5.8.

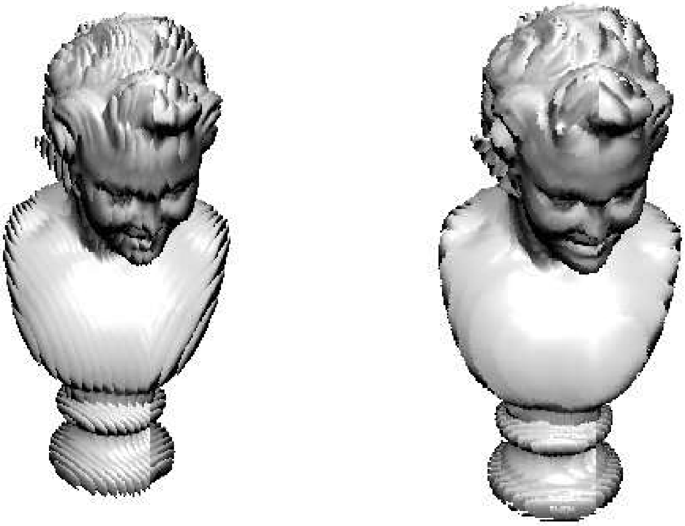


Figure 5.8
The image on the left uses RTMs aligned with the bounding box planes. The image on the right has slices normal to the view direction.

Listing 5.2 is the shader for this multi-pass depth slicing algorithm.

```
struct app2vert
{
        float4 pos : POSITION;
    float4 color : DIFFUSE;
        float4 texcoord: TEXCOORD0;
};

struct vert2frag
{
        float4 hpos : POSITION;
        float4 color : COLOR0;
        float4 texcoord : TEXCOORD0;
};

struct frag2screen
{
        float4 color : COLOR;
};

float2 project_uv(float3 p,float4 u,float4 v)
{
        return float2(dot(p,u.xyz)/u.w,dot(p,v.xyz)/v.w);
}

vert2frag main_vert(
        app2vert IN,
    uniform float4x4 modelviewproj)
{
    vert2frag OUT;

        OUT.hpos = mul(modelviewproj, IN.pos);
        OUT.color = IN.color;
        OUT.texcoord  = IN.pos;

    return OUT;
}
```

```
frag2screen main_frag(
        vert2frag IN,
        uniform sampler2D texdtb1,    // texture map front (rgb:normal, alpha:depth)
        uniform sampler2D texdtb2,    // texture map back (rgb:normal, alpha:depth)
        uniform float4 plane,         // plane (xyz:normal, w:distance)
        uniform float4 pos,                   // base vertex pos (xyz)
        uniform float4 u,                     // base u axis (xyz:normalized, w:length)
        uniform float4 v,                     // base v axis (xyz:normalized, w:length)
        uniform float4 depth,         // depth related constants
        uniform float4 camerapos,     // camera position (xyz)
        uniform float4 lightpos,      // lightposition (xyz)
        uniform float4 specular)      // specular color (xyz:rgb, w:exponent)
{
        frag2screen OUT;

        float4 c=float4(0,0,0,0);
        float4 t1,t2;
        float3 viewdir,lightdir,p;
        float2 uv;

        // compute view direction
        viewdir = normalize(IN.opos.xyz-camerapos.xyz);

        // transform point into local space
        p = IN.opos.xyz - pos.xyz;

        // intersect depth plane
        p+= depth.w*(-viewdir/dot(plane.xyz,viewdir));

        // fetch texture maps
        uv      = project_uv(p,u,v);
        t1      = f4tex2D(texdtb1,uv);                                  //     front
texture
        t2      = f4tex2D(texdtb2,float2(1-uv.x,uv.y));      // back texture
        t2.w    = 1.0-t2.w;

        // test is pixel depth is inside range
        if (t2.w>t1.w && (depth.x-t1.w)*(depth.x-t2.w)<0)
                c.w=1;

        // transform point into global space
        p+=pos.xyz;

        // expand normal from normal map
        t1.xyz=normalize(t1.xyz*2-1);

        // compute light direction
        lightdir=normalize(lightpos.xyz-p);

        // diffuse+specular lighting
        float diff=saturate(dot(lightdir,t1.xyz));
        float spec=pow(saturate(dot(normalize(lightdir-viewdir),t1.xyz)),specular.w);

        // compute final color
        c.xyz=IN.color.xyz*diff.xxx+specular.xyz*spec.xxx;

        OUT.color=c;
        return OUT;
}
```

Listing 5.2
Multi-pass depth slice RTM shader

RTM – Single pass approach for planar objects

This methods uses the same basic approach as in the previous section but now we make all computations in a single pass with more precision for the intersection and with self shadows included. The efficacy of these can be seen in Figure 5.9. Also shown in the figure is a comparison with bump mapping again showing the visual superiority of this technique.

The crux of the shader is the use of an efficient ray intersect routine, enabling an exact depth of the intersection along the ray to be returned. The addition of shadows results in more computation (two ray intersect calculations per fragment) and to facilitate this we incorporate an efficient search algorithm (linear and binary) to calculate the required ray intersections and also whether the point is in shadow or not.

Self-shadows are a useful addition, but general shadowing would be even better. The fact that an accurate depth is calculated means that it would also be possible to use shadow maps or stencil shadows in addition to the self-shadows. However, the current complexity compiles to around 200 (400 with shadows) assembly instructions (compared to around 30 for bump mapping) a very large increase.

In this example we use a hybrid of a linear and binary chopping search algorithms to drastically speed up the ray intersection calculations required. (This should be compared with the algorithm used in the previous section that requires up to 256 passes for a good image quality).

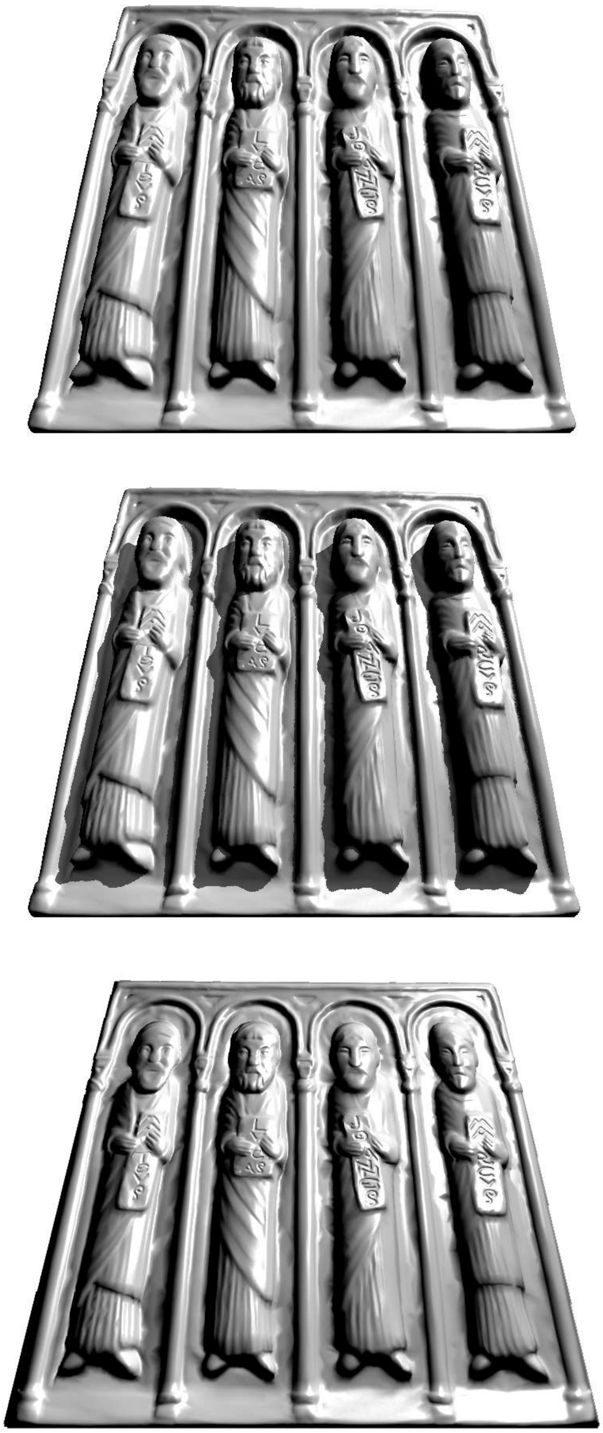


Figure 5.9
RTM with self shadows
a) RTM without self shadows
b) RTM with self shadows
c) The same object rendered using a conventional bump map technique

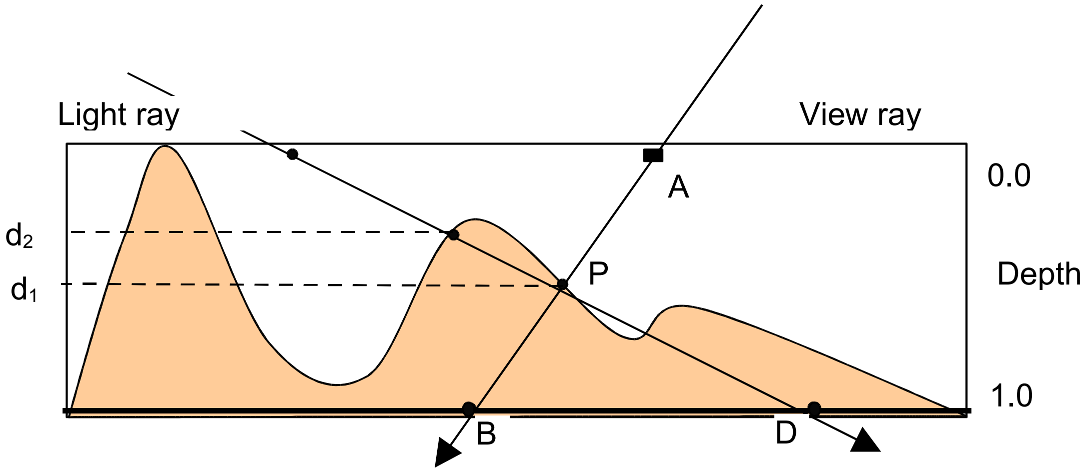


Figure 5.10
Two rays are used to render the RTM object and to determine whether it is in self shadow

The algorithm is easily understood by referring to Figure 5.10. Here A is the pixel currently being rendered. We find the surface intersection point *P* by calling a ray intersect calculation which takes as input the start and end points of the ray (A) and (B) (see the pseudo code in Figure 5.11). Point *P* defines depth *d1*. An identical procedure is followed with the light ray to find depth *d2*. If *d2* is less than *d1* then the point *P* is in self shadow (the case in the figure) and rendered accordingly; otherwise standard illumination is applied. The pseudo code for the main algorithm is shown in Figure 5.11.

```
float4 main_rtm()
        {
        A = pixel position in global space (passed in to shader in texcoord0)
        Viewdir = normalize( A – camera_pos );
        size = relief_depth / dot( -relief_normal, Viewdir);
        B = A + Viewdir * size;
        V = B – A;
        d1 = ray_intersect_rtm(A, V);
        P = A + d1 * V;
        Lightdir = normalize( P – light_pos );
        size = relief_depth / dot( -relief_normal, Lightdir);
        C = P – d1 * size * Lightdir;
        D = C + Lightdir * size;
        V = D – C;
        d2 = ray_intersect_rtm( C, V );
        if (d2<d1)
                // pixel in shadow
                color = shadow_color()
        else
                // light pixel, apply standard lighting
                color = phong_lighting();
        return color;
        }
```

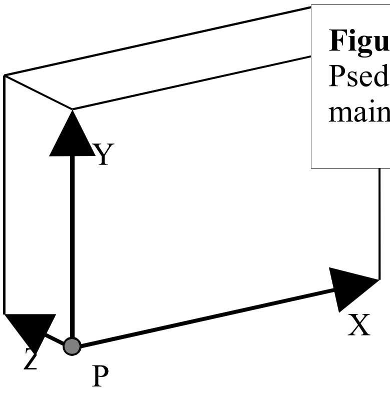


Figure 5.11
Psedo-code and conventions for the main algorithm

Z).

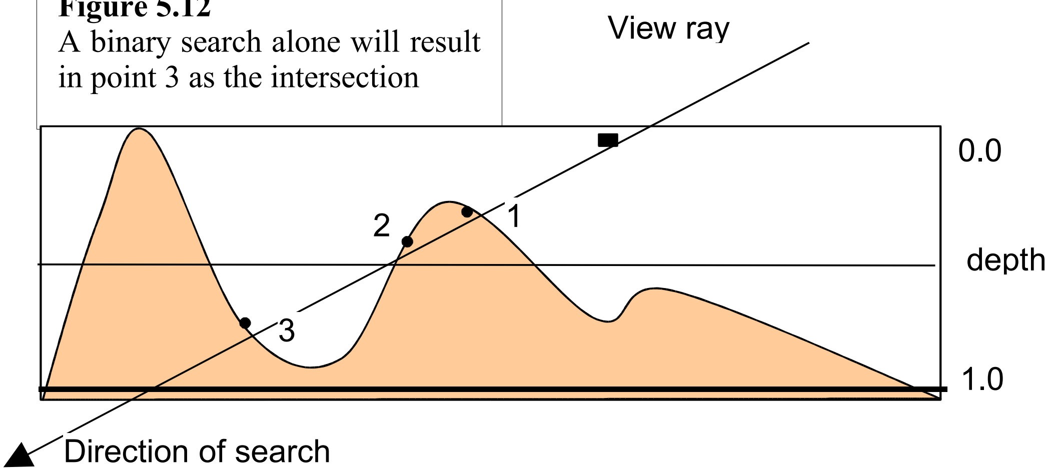
relief_depth = length(Z);
relief_normal = -Z / relief_depth;

For reasons that will be explained shortly the ray intersect routine is a coarse resolution linear search followed by a fine resolution binary search. First we examine the binary search. This is a simple binary chop between points A and B if we ignore for a moment the linear search. Starting at a point halfway along the ray we invoke the test inside/outside the RTM object. If inside we move backwards otherwise we move forwards, the recursion being implemented as a deterministic loop as follows:.

```
float ray_intersect_rtm( float2 P, float2 V )
{
        linear_search_steps = 10;
        binary_search_steps = 6;

        depth = 0.0;
        size = 1.0/linear_search_steps;

        // find first point inside object
        loop from 1 to linear_search_steps
                depth = depth + size;
                d = rtm depth value at pixel (P+V*depth);
                if ( d<depth )
                        break loop;

        // recurse with binary search around
        // first point inside object to find best depth
        best_depth = 1.0;
        loop from 1 to binary_search_steps
                size = size*0.5;
                d = rtm depth value at pixel (P+V*depth);
                if ( d < depth )
                        // if point is inside object, store depth and move backward
                        best_depth=depth
                        depth = depth – size;
                else
                        // else it is outside object, move forward
                        depth = depth + size

        return best_depth;
}
```

Now we examine the need for a linear search that precedes the binary search. Figure 5.12 shows what would happen, in this particular case if we applied only a binary chop.

Figure 5.12
A binary search alone will result in point 3 as the intersection



In the figure there are three ray surface intersections labelled 1,2 and 3. The first invocation of the chop (intersection of view ray and depth plane *Z=0.5*) is ’outside’ and we move forward, ending up at point 3. We note that this error is a consequence of concavities in the surface. If the surface were everywhere convex then a binary search would suffice.

This is easily corrected by applying a linear search first. Consider an example. Say we choose a resolution for the linear search of 10 (0.1,0.2,…1.0). We terminate the linear search when we first enter the object. We then switch to the binary search with a starting resolution of 0.1 at this point. Say the first invocation of the binary search returns ‘0.3 inside’. We then move to 0.25 (0.3-0.05) and recurse within the new 0.05 depth interval (each time reducing the interval by half).

But even with this enhancement there is still a problem. If the linear search encounters a thin object part (less than the search interval of, say, 0.1) it will ‘jump over’ it. Thus there is a limit to the thinness of object protrusions for a fixed search interval. This problem is exacerbated as the view ray incidence angle approaches being parallel to the surface. These errors can be noticeable on shadow edges from the pyramid (Figure 5.13) where the width of the object at its edges is too small) it looks like the shadows have lower resolution at such points as you can see in Figure 5.13.

We have to reduce the linear search intervals as a function of such surface detail and view/surface angle. To do this correctly we would need to define an interval that is smaller than the smallest path through any protrusion. This would require a lengthy pre-processing phase that sampled a set of low incidence view vectors at every texel of the RTM object.

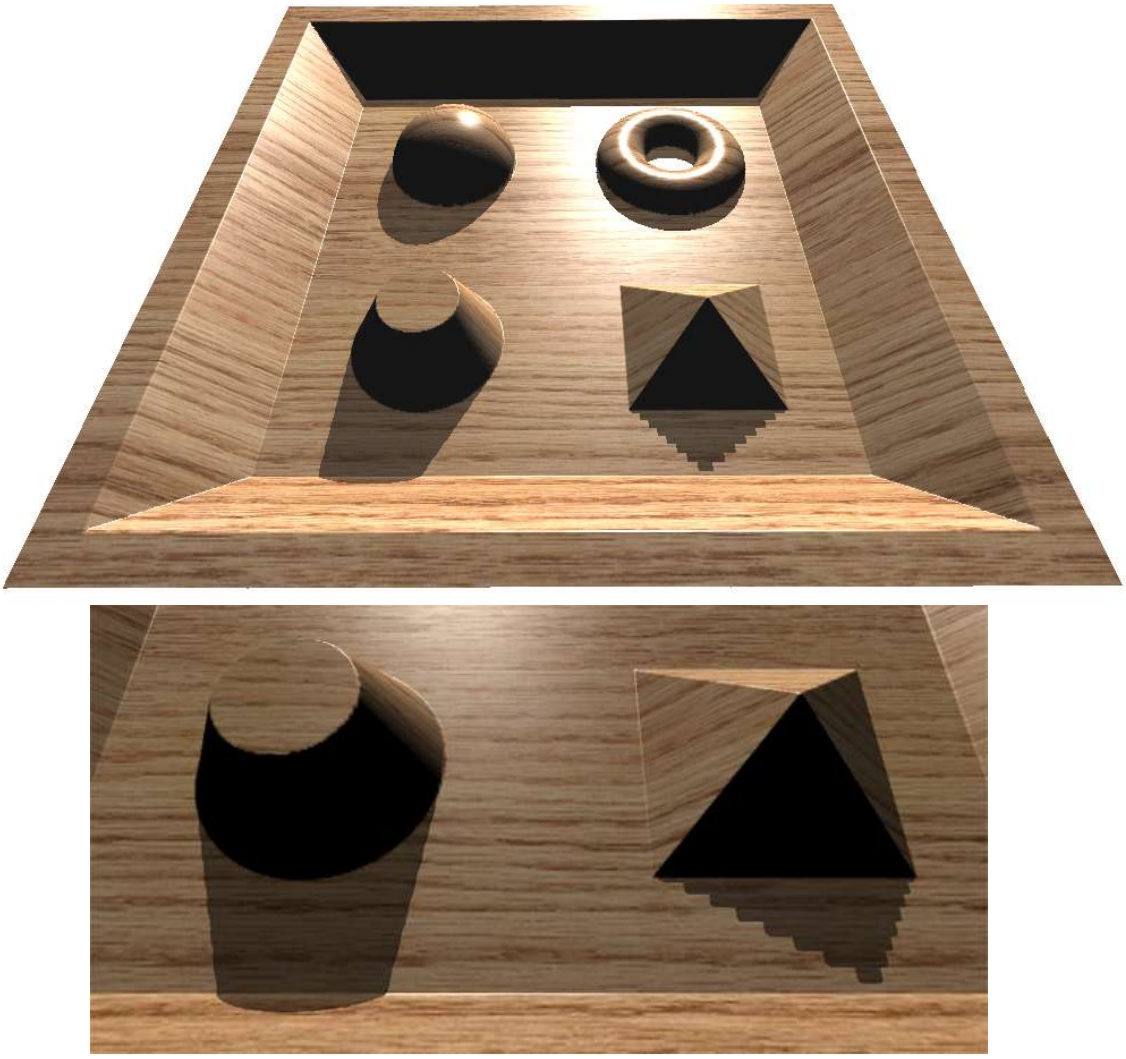


Figure 5.13
Artefacts generated by object parts that are too thin for the linear search

Texture Filtering and Mip-mapping

Both texture filtering and mip-mapping work well with this shader. Texture filtering will facilitate good normal interpolation around surface corners and mip-mapping will speed up the shader considerably (especially when using large tiles where lower mip-map levels are used). Both normal and depth components are mip-mapped to lower resolutions.

Pixel Shader 3.0 Optimizations

The new pixel shaders ver 3.0 allows variable loops and true if/else statements. The Relief Mapping shader could take advantage of this especially in the linear search loop. There we could break out of the loop on the first point found to be inside the object, theoretically saving time of several unneeded loop passes.

Listing 5.3 is the complete Cg code for the shader (main and ray intersection functions):

```
frag2screen main_frag_rm(
        vert2frag IN,
        uniform sampler2D rmtex,      // relief map
        uniform sampler2D colortex,   // color map
        uniform float4 axis_pos,      // base vertex pos (xyz)
        uniform float4 axis_x,                // base x axis (xyz:normalized, w:length)
        uniform float4 axis_y,                // base y axis (xyz:normalized, w:length)
        uniform float4 axis_z,                // base z axis (xyz:normalized, w:length)
        uniform float4 camerapos,     // camera position (xyz)
        uniform float4 lightpos,      // lightposition (xyz)
        uniform float4 specular)      // specular color (xyz:rgb, w:exponent)
{
        frag2screen OUT;

        float3 v,l,p,s;
        float2 dp,ds;
        float d,dl;

        const float shadow_threshold=0.02;
        const float shadow_intensity=0.4;

        // ray intersect in view direction
        v  = normalize(IN.opos.xyz-camerapos.xyz);
        p  = IN.opos.xyz - axis_pos.xyz;
        s  = axis_z.w*v/dot(axis_z.xyz,v);
        dp = project_uv(p,axis_x,axis_y);
        ds = project_uv(s,axis_x,axis_y);
        d  = ray_intersect_rm(rmtex,dp,ds);

        // get relief map and color texture points
        float2 uv=dp+ds*d;
        float4 t=f4tex2D(rmtex,uv);
        float3 color=IN.color.xyz*f3tex2D(colortex,uv);

        // expand normal from normal map in local polygon space
        t.xy=t.xy*2.0-1.0;
        t.z=sqrt(1.0-dot(t.xy,t.xy));
        t.xyz=normalize(t.x*axis_x.xyz+t.y*axis_y.xyz-t.z*axis_z.xyz);

        // compute light direction
        p+=axis_pos.xyz+s*d;
        l=normalize(p.xyz-lightpos.xyz);

        // compute diffuse and specular terms
        float diff=saturate(dot(-l,t.xyz));
        float spec=saturate(dot(normalize(-l-v),t.xyz));

#ifdef RM_SHADOWS
        // ray intersect in light direction
        s  = axis_z.w*l/dot(axis_z.xyz,l);
        p -= d*s+axis_pos.xyz;
        dp = project_uv(p,axis_x,axis_y);
        ds = project_uv(s,axis_x,axis_y);
        dl = ray_intersect_rm(rmtex,dp,ds);
        // if pixel in shadow
        if (dl<d-shadow_threshold)
        {
                color*=shadow_intensity;
                specular=0;
        }
#endif

        // compute final color
        OUT.color.xyz=color*diff+specular.xyz*pow(spec,specular.w);
        OUT.color.w=(d<1.0?1.0:0.0);
        return OUT;
}

// RAY INTERSECT DEPTH MAP WITH BINARY SEARCH
// RETURNS INTERSECTION DEPTH OR 1.0 ON MISS
float ray_intersect_rm(
                in sampler2D rmtex,
                in float2 dp,
                in float2 ds)
{
#ifdef RM_DOUBLEPRECISION
        const int linear_search_steps=20;
        const int binary_search_steps=5;
#else
        const int linear_search_steps=10;
        const int binary_search_steps=5;
#endif
        float depth_step=1.0/linear_search_steps;

        // current size of search window
        float size=depth_step;
        // current depth position
        float depth=0.0;
        // best match found (starts with last position 1.0)
        float best_depth=1.0;

        // search front to back for first point inside object
        for( int i=0;i<linear_search_steps-1;i++ )
        {
                depth+=size;
                float4 t=f4tex2D(rmtex,dp+ds*depth);

                if (best_depth>0.996)  // if no depth found yet
                if (depth>=t.w)
                        #ifdef RM_DOUBLEDEPTH
                                if (depth<=t.z)
                        #endif
                                        best_depth=depth;      // store best depth
        }
        depth=best_depth;

        // recurse around first point (depth) for closest match
        for( int i=0;i<binary_search_steps;i++ )
        {
                size*=0.5;
                float4 t=f4tex2D(rmtex,dp+ds*depth);
                if (depth>=t.w)
                #ifdef RM_DOUBLEDEPTH
                        if (depth<=t.z)
                #endif
                        {
                                best_depth=depth;
                                depth-=2*size;
                        }
                depth+=size;
        }

        return best_depth;
}
```

Listing 5.3
The complete RTM fragment shader

Single RTM with Double Depth values (restricted 3D objects)

This extension gives an effect that approximates the 5 map method we used to produce Figure 5.4. The significant advantage of the method is that it only uses a single map rather than 5 or 6. It represents closed objects whose geometry can be captured using two depth maps. It means that the complete geometry of a certain type of 3D object can be represented by a single map. The right hand column of Figure 5.15 shows two results achieved using this method. The left column shows the RTM rendered using the unenhanced method (that is from a single depth RTM). Here the expected extruded effect is visible where bundles of parallel lines emerge along the *z* direction

This is a simple enhancement and involves adding a second depth map to the blue channel that will hold the depth values looking from the back of the object as shown in Figure 5.14. This means now we will have the normal *x* and *y* components encoded only in red/green and two depth maps (back and front) encoded in blue/alpha. However, since the normal vector is normalized, the *z* component can be retrieved in the fragment shader from:

$$z=\sqrt{1-x^2-y^2}$$

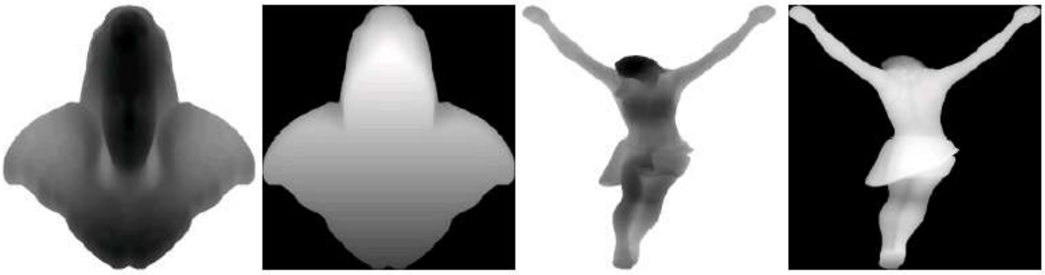


Figure 5.14
Double depth relief maps (Angel and Christ). Front and back depth maps encoded in blue and alpha channels.

In this method we consider a point to be inside the ob
front depth map and smaller than the back depth map.

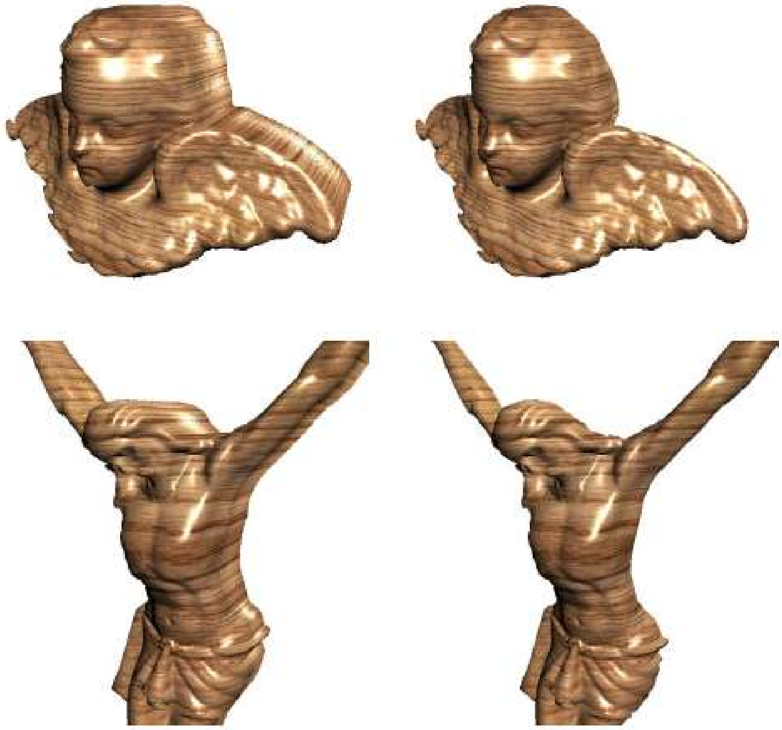


Figure 5.15
Angel and Christ relief maps using double depth. Conventional single depth RTM used to render left images; double depth was used to render the right column (back of figures are capped by back depth map).

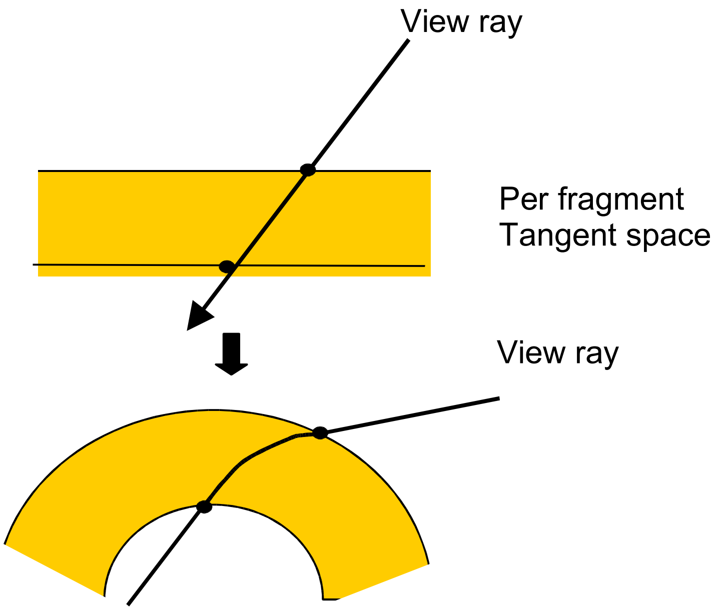
Comparing the two columns in Figure 5.15 you can see the 'capping' effect of using the back depth map. In the left hand column the rendering extends to maximum depth. The capping occurs in the following way. If the search returns a depth of 1.0 then this means that the ray has missed the object and may be going through a region of space rendered as the extrusion region in the left hand imagery in the figure. Thus all the shader has to do to effect a capping operation is to set alpha to 0 if the depth is 1.0:

```
    OUT.color.w = (d<1.0?1.0:0.0)
```

The double depth option is included in the ray intersection function from Listing 5.3 in the form of an extra if statement comparing the back depth value both in linear and binary search steps.

Relief Texture mapping – for arbitrary 3D objects

We now present the final variation of the RTM technique that can be applied to any object geometry. Here we are effectively have two geometries – the geometry of the RTM and the geometry of the object. (In the previous sections the RTM was the object.) Consider the graphical representation of the algorithm in the previous technique with the geometry of the RTM represented as a slab (Figure 5.16a). Now consider removing the planar constraint so that we can deal with arbitrary shaped objects by bending the relief volume. We end up with Figure 5.16b. If we maintain the same simple ray intersection code, we will be effectively bending the ray through the relief volume. We can effect this process by working in the per fragment tangent space of the object. We can consider the object to be locally planar and use the same algorithm.



Object space

Figure 5.16
Working in per fragment tangent space is analogous to 'bent' ray tracing

Thus for each fragment the method remains effectively the same and we only have to calculate the ray intersection start point and its direction in the fragment tangent space. We do this by transforming the view ray into the tangent space of the fragment.

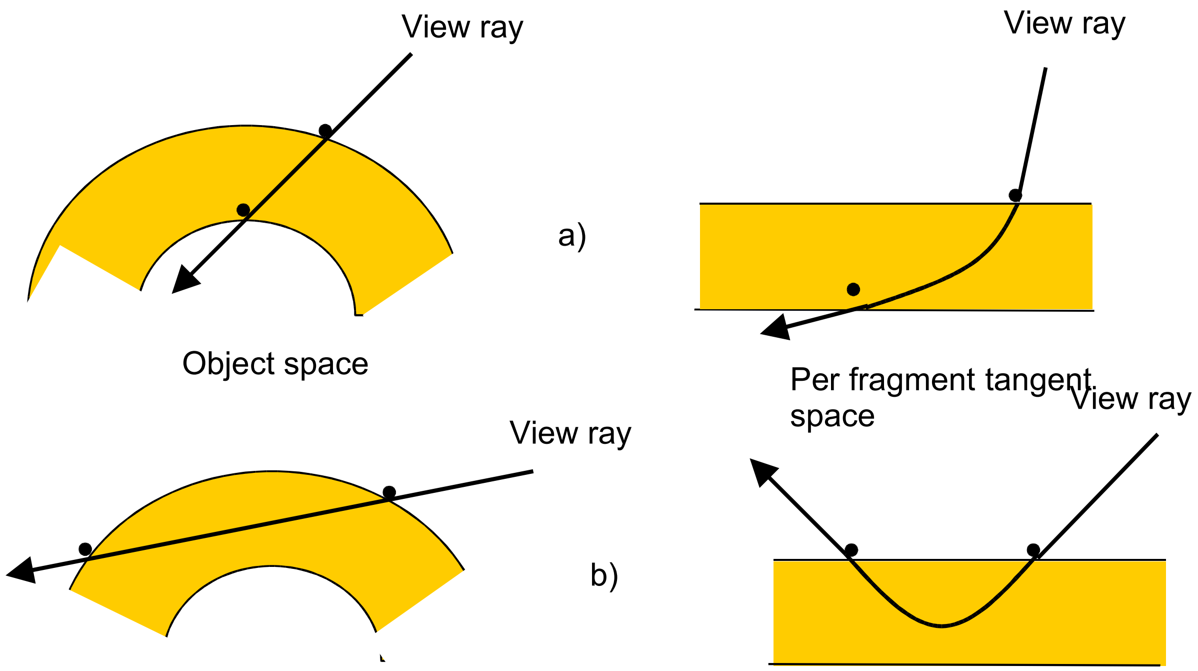


Figure 5.17
Ray events as a function of incidence angle, relief depth and curvature of object.

Bending the intersection ray - a consequence of considering the surface to be planar at every fragment - is not without its problems. Consider Figure 5.17a which shows a bent ray entering and exiting the surface. This ray will intersect with the surface of the RTM. Figure 5.17b shows a bent ray that will enter the top surface bounding volume of the RTM and then exit through the same surface. This ray may intersect the surface or it may not. Such ray events will occur near silhouette edges. These ray events are a consequence of the angle of incidence, the relief depth and the curvature of the receiving surface. We deal with this proble4m in the next section.

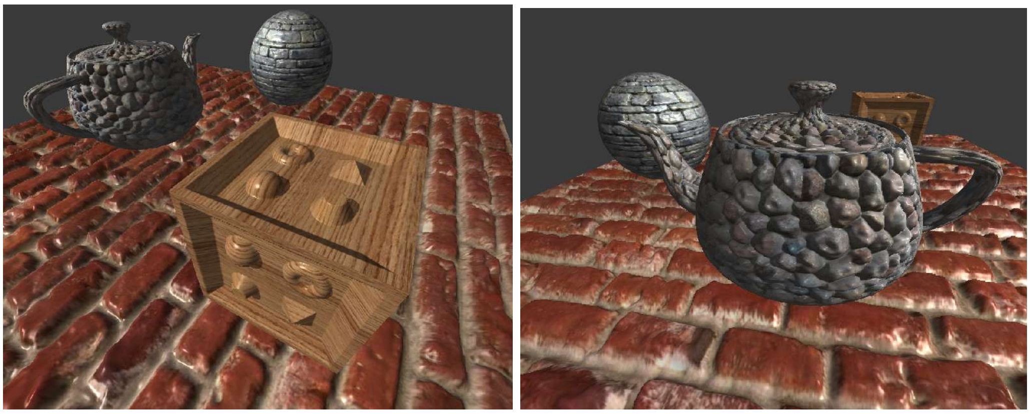


Figure 5.18
RTM for arbitrary polygonal objects

An image using this approach is shown in Figure 5.18 which was produced using the shader in Listing 5.4. There are two problems with this enhancement: silhouette edges are smooth and the shadow calculation is more complex. Silhouette edges are smooth because we use the same ray intersection function as the previous approach and this considers the object locally planar per fragment.

We now consider self shadow in RTM objects. (Shadows cast by RTM objects and shadow cast onto RTM objects are dealt with in Chapter4.) Self shadows need to be evaluated in tangent space. We effectively invoke the algorithm (for planar surfaces) for each fragment in each fragment's tangent space. Referring to Figure 5.19, the process begins at point p_1 in tangent space which is the fragment texture coordinate. We then transform the view vector into tangent space to find the ray intersection direction ***(V)***. Calling the ray intersection function will return the intersection depth *(d_1)*. We can then calculate the intersection point p_2:

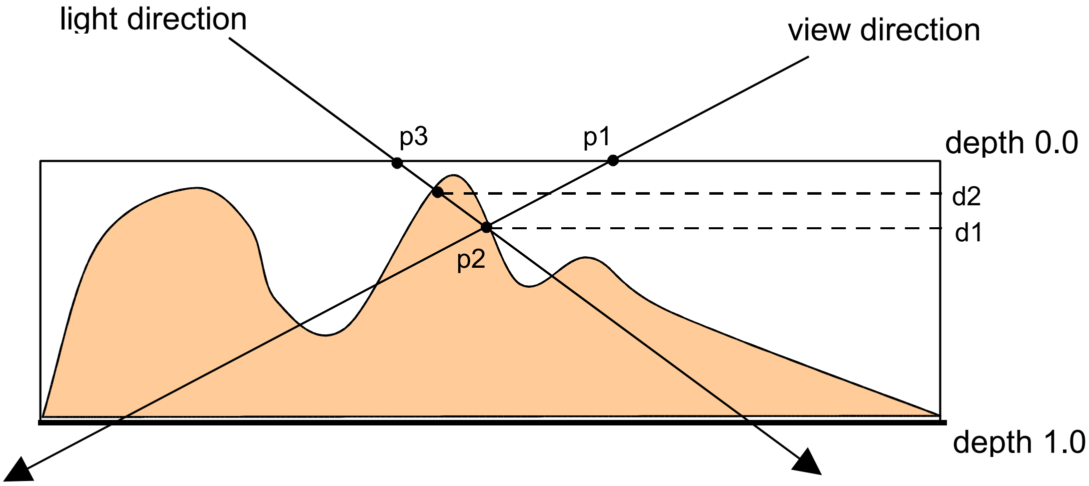
$$p_2 = p_1 + d_1 * \mathbf{V}$$

We can now calculate the light direction vector ***(L)*** from light position to p_2. But we need the light vector entry point in order to call the ray intersection function from the light direction. This is given by:

$$p_3 = p_2 - d_1 * \mathbf{L}$$

Calling the ray intersection function with p_3 (entry point) and ***L*** (direction) will return the intersection depth *(d_2)* along the light direction. If d_2 is smaller than d_1 it means we are in shadow.

Figure 5.19
Per fragment tangent space



```
f2s main_frag_relief(
        v2f IN,
        uniform sampler2D rmtex:TEXUNIT0,               // rm texture map
        uniform sampler2D colortex:TEXUNIT1,    // color texture map
        uniform float4 camerapos,               // camera position
        uniform float4 lightpos,                // lightposition
        uniform float4 ambient,                         // ambient color
        uniform float4 diffuse,                         // diffuse color
        uniform float4 specular,                // specular color
        uniform float4 projz,                   // 3rd column from projection matrix
        uniform float4 planes,                          //  near  and  far  plane  distances
(near,far,near*far,1/(far-near))
        uniform float tile,                             // tile factor
        uniform float depth)                    // depth factor
{
        f2s OUT;

        float4 t,c;
        float3 p,v,l,s;
        float2 dp,ds,uv;
        float d,a;

        // ray intersect in view direction
        p  = IN.vpos;
        v  = normalize(p);
        a  = dot(IN.normal,-v);
        s  = normalize(float3(dot(v,IN.tangent),dot(v,IN.binormal),a));
        s  *= depth/a;
        ds = s.xy;
        dp = IN.texcoord*tile;
        d  = ray_intersect_rm(rmtex,dp,ds);

        // get rm and color texture points
        uv=dp+ds*d;
        t=tex2D(rmtex,uv);
        c=tex2D(colortex,uv);

        // expand normal from normal map in local polygon space
        t.xyz=t.xyz*2.0-1.0;
        t.xyz=normalize(t.x*IN.tangent+t.y*IN.binormal+t.z*IN.normal);
        // compute light direction
        p += v*d*a;
        l=normalize(p-IN.lightpos.xyz);

#ifdef RM_DEPTHCORRECT
        // compute currected depth value for displaced pixel
        float linear_depth=dot(-p,projz.xyz)+projz.w;
        // (far*near/depth+far)/(far-near)
        OUT.depth=(planes.z/linear_depth+planes.y)*planes.w;
#endif

        // compute diffuse and specular terms
        float att=saturate(dot(-l,IN.normal));
        float diff=saturate(dot(-l,t.xyz));
        float spec=saturate(dot(normalize(-l-v),t.xyz));

        float4 finalcolor=ambient*c;

#ifdef RM_SHADOWS
        // ray intersect in light direction
        dp+= ds*d;
        a  = dot(IN.normal,-l);
        s  = normalize(float3(dot(l,IN.tangent),dot(l,IN.binormal),a));
        s *= depth/a;
        ds = s.xy;
        dp-= ds*d;
        float dl = ray_intersect_rm(rmtex,dp,s.xy);
        if (dl<d-0.05) // if pixel in shadow
        {
          diff*=dot(ambient.xyz,float3(1.0))*0.333333;
          spec=0;
        }
#endif

        // compute final color
        finalcolor.xyz+=att*(c.xyz*diffuse.xyz*diff+specular.xyz*pow(spec,specular.w));
        finalcolor.w=1.0;

        OUT.color=finalcolor;

        return OUT;
}
```

Listing 5.4
RTM shader for arbitrary polygonal objects

Depth Correct

We can update each fragment's depth value to reflect the newly generated relief surface. These values are required to integrate RTM objects into scenes, for example, to generate correct shadows. To find the corrected depth value to output from the shader, we first compute the fragment depth by making a 4 component dot product of the fragment position with the 3rd column of the model view projection matrix. Then we transform the fragment depth into the current Z buffer range, defined by the near and far plane distances, with the following formula:

$$z_{\text{depth}} = \frac{\dfrac{\text{far*near}}{\text{depth}} + \text{far}}{\text{far-near}}$$

Using corrected depth values we can compose several different relief maps – one for each object represented in this way and they will intersect each other correctly at the fragment level – implementing a 3D compositing operation Also standard triangle based geometry can inter-penetrate the relief maps correctly as shown in Figure 5.20.

Another benefit that emerges from corrected depth values in the relief map is that it enables shadows from other objects to project onto the relief map. The projected shadows will follow the displaced relief surface correctly. Both stencil shadows and shadow maps will work with the corrected depth relief map. (Shadows from RTM objects and from other objects onto RTM objects are dealt with in Chapter 4.)

The code for the corrected depth computation is as follows:

```
// 3rd column from model view projection matrix
uniform float4 modelviewprojz;
// near and far plane distances (near,far,near*far,1/(far-near))
uniform float4 planes;
// compute point z coordinate
float depth=dot(-p,modelviewprojz);
// (far*near/depth+far)/(far-near)
OUT.depth=(planes.z/depth+planes.y)*planes.w;
```

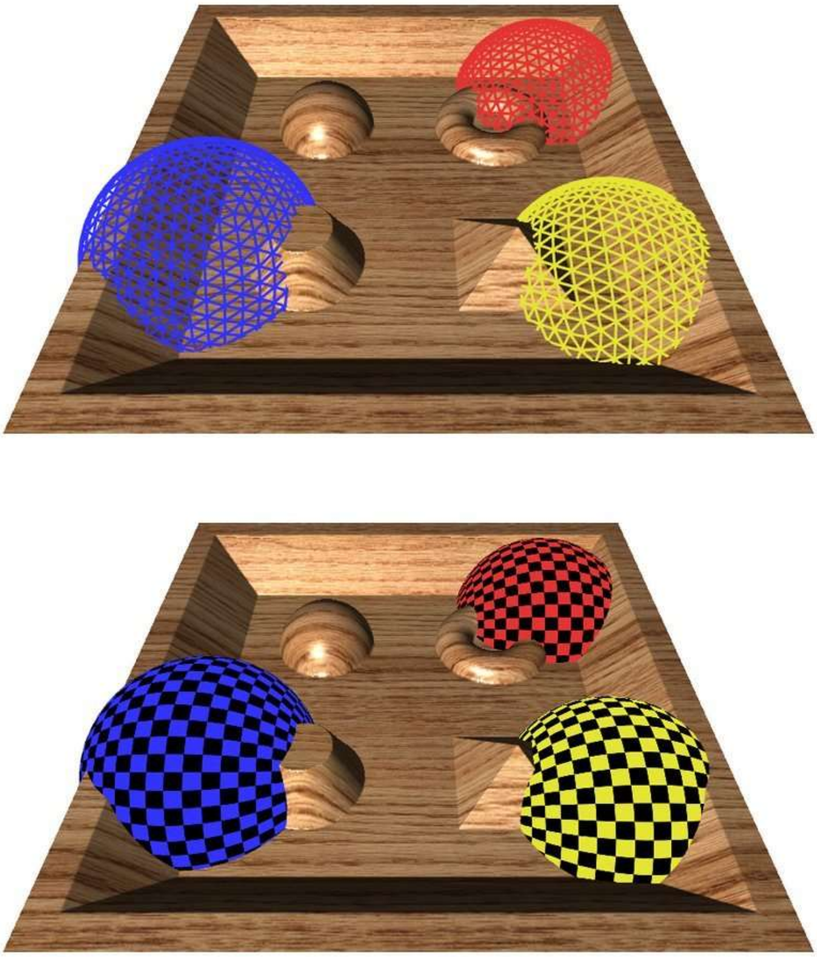


Figure 5.20
Interpenetration of relief mapped objects at fragment level.

A visual comparison of three methods

This section presents a comparison for three of the methods used in Chapters 3 and 5: Normal/Bump Mapping, Parallax Mapping, Relief Mapping

An image rendered using the same texture for each of the three methods in shown in Figure 5.21. We have deliberately chosen a 'bad' viewing angle to highlight the differences amongst the methods.

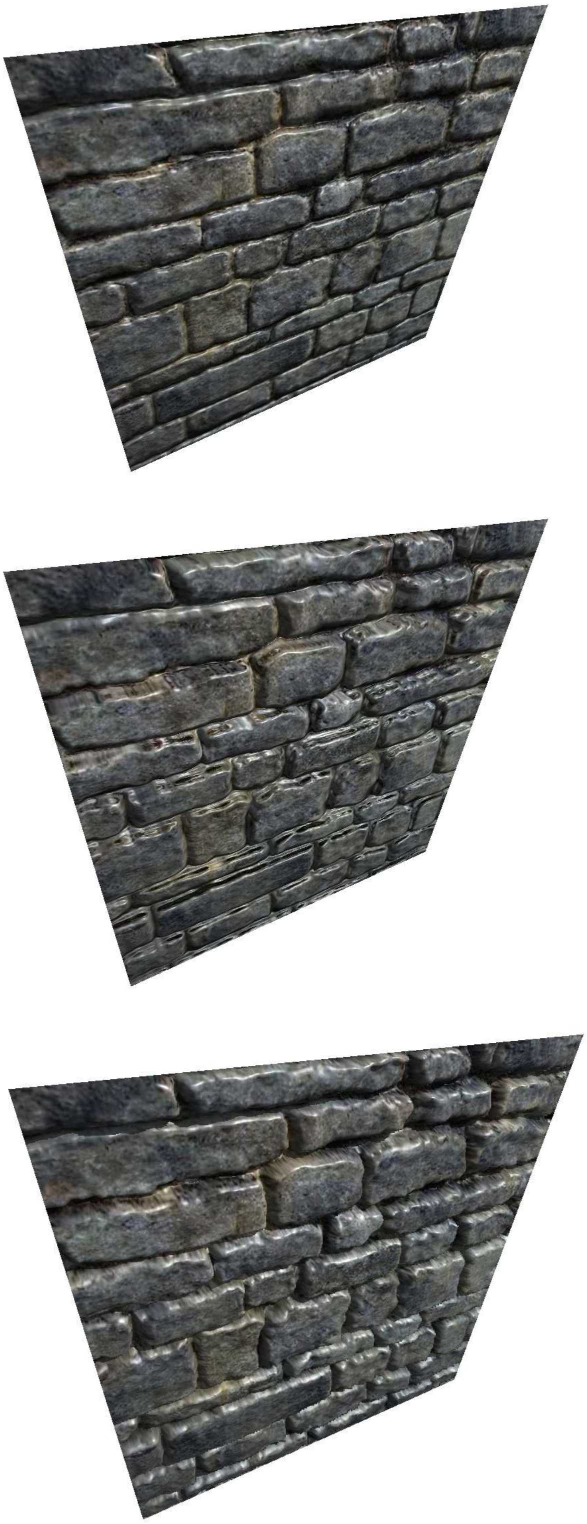


Figure 5.21
Comparing bump mapping, parallax mapping and RTM mapping

As expected Normal Mapping exhibits little feel of depth. Parallax Mapping is much better in this regard, but there are significant artefacts and distortions. We conclude at the third image (Relief Mapping) gives the best result.

Silhouette edges and Relief Texture mapping

In previous sections we treated the object as being locally planar at each fragment. In this way the ray travelling through the object was considered a line through tangent space. This planar approach is simple and efficient but lacks the ability to produce the silhouette edge correctly.

An interesting extension to the planar approach is to consider estimating the curvature of the surface receiving the relief map at every fragment. As we shall see this will enable us to implement a curved ray in texture space implement and decide whether a ray should be discarded because it misses the surface, or whether it should render the fragment.

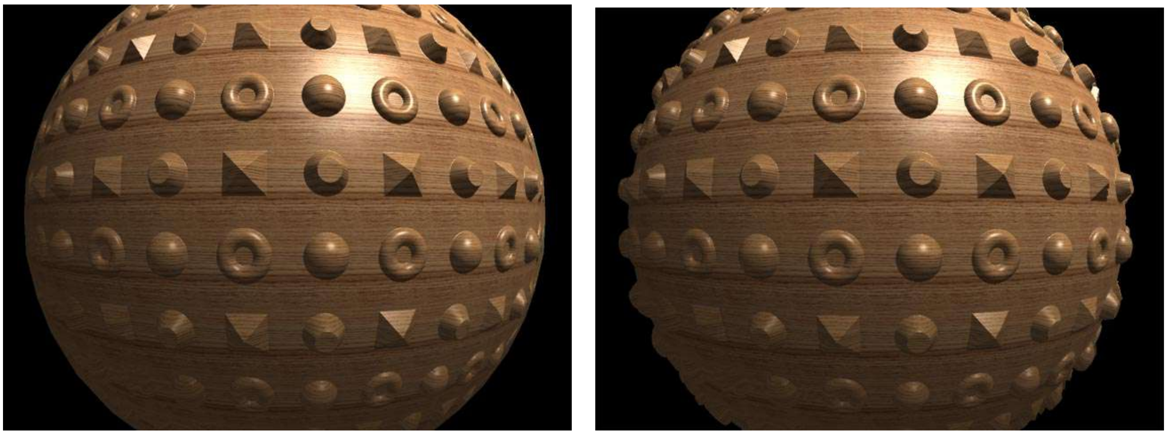
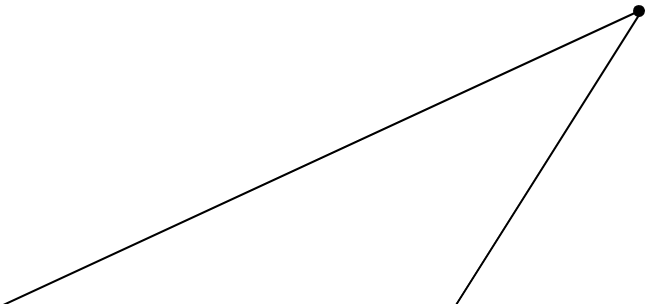


Figure 5.22
Relief mapped sphere with and without the silhouette edge enhancement.

A RTM sphere is shown in Figure 5.22 with and without a correct silhouette edge. In Figure 5.23 two view rays are shown. $\boldsymbol{V_A}$ **intersects** the height field whereas $\boldsymbol{V_B}$ misses. However, the ray intersection procedure will return the coordinates of the intersection of $\boldsymbol{V_B}$ with a tiled version of the texture and all fragments resulting from the scan conversion of the object will contribute to an intersection and the silhouette of the object will match the silhouette of the underlying polygonal object, which in the case of the object in Figure 5.22 is a circle. This situation is shown in Figure 5.23a).



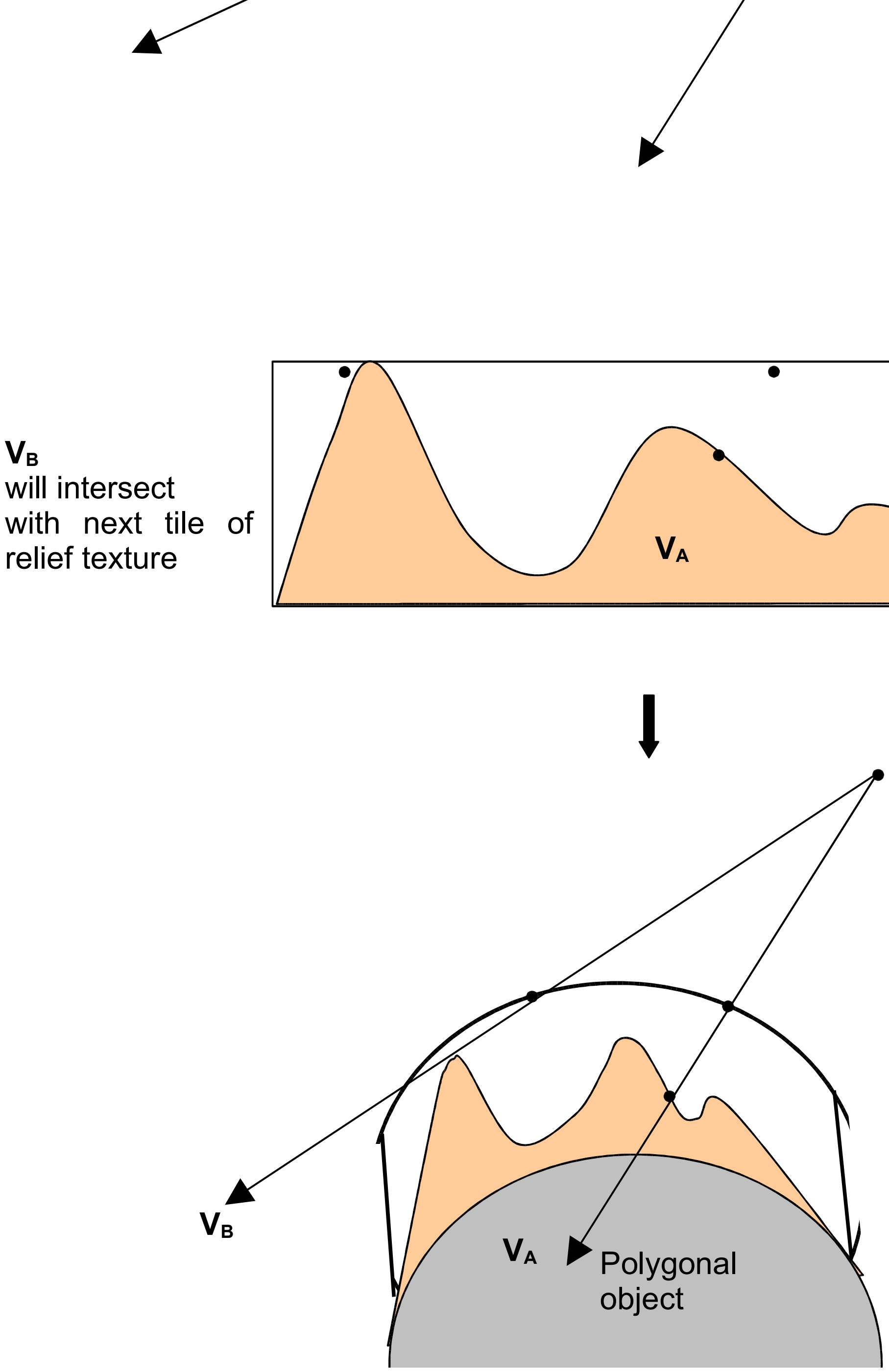


Figure 5.23
a) $\mathbf{V_B}$ will intersect with a relief texture
b) Relief object is locally deformed to fit the object's surface. $\mathbf{V_B}$ can be discarded

A method that can decide whether ray V_B should be discarded, so that a correct silhouette can emerge, is to bend the relief texture to fit the curvature of the surface. Figure 5.23b) shows that V_B can be discarded. This abstraction can be implemented in texture space. To do this we need a method of estimating surface curvature. Thus we now require a method that determines if a ray can be discarded and a method that finds the intersection in the deformed relief object.

Curvature estimation

In this section we will use an estimation of the per fragment curvature. To do this we estimate surface curvature at each vertex and have this information linearly interpolate to the fragments. Curvature will be evaluated in the two tangent space directions (which are aligned with texture space u and v directions at each vertex). In practice curvature estimation per vertex would be calculated off-line and the vertex information passed to the GPU enhanced with the curvature parameters.

A simple estimation of curvature of a mesh object is given by fitting a quadric to the surface (explained full in the next section). This quadric effectively becomes the upper and lower surfaces of the relief texture's bounding box (Figure 5.23b). We use the set of paraboloids (Figure 5.27) defined by:

$$Ax^2 + By^2 + z = 0 \qquad (5.1)$$

and estimate *A* and *B* at each vertex to quantify the local curvature around that vertex (see the next section for details). This quadric is considered to be aligned with the vertex tangent space so the vertex is positioned at (0,0,0) in the quadric space and *z* is aligned with direction (0,0,1). The quadric will then be used in the pixel shader to bend the ray as it goes through the relief map.

We must clearly differentiate between object space, tangent space and texture space. In object space coordinates are in relative to the world origin (0,0,0) and main axis alignment such as (1,0,0), (0,1,0) and (0,0,1). In tangent space (vertex or fragment) the coordinates are relative to the vertex/fragment position and oriented as the vertex/fragment normal and tangent vectors. But the scaling of the tangent space is the same as the scaling in object space (only translation and rotation differs between them).

In texture space we scale tangent space so that one texture tile (size 1.0 in texture space) maps to some distance in object space. Also the depth expressed in the range [0.0,1.0] in texture space will map to some depth in object space. Thus it is important to know at each vertex/fragment the size of the mapping (or how big one texture tile is in object space at that position). As mapping a complex object is not a simple task, usually the mapping scale at each point on the object will vary and texture will compress/expand though the object surface (only planar surfaces can have a simple mapping that will have constant mapping scale at every point).

Thus we end up requiring 4 new floats at each vertex (curvature in two directions and mapping scale in two directions). These can be calculated offline and stored in the geometry file as they are constants for a non-deformable mesh. For the mapping scale, all we need to do is compute the texture space positions (0,0), (1,0) and (0,1) in object space for every triangle in the mesh, and then average the length of each *u* and *v* texture axis at the vertices. This can be done in the tangent space computation method and stored in the tangent space vector 4th component (scale of tangent vector in object space stored in *w*). Thus in order to move from tangent space to texture space we divide *x* and *y* by the texture scale in each tangent direction and *z* by the desired relief depth in object space.

Ray-quadric distance calulation

To calculate intersections in texture space we need the distance along the ray from the quadric. We compute the depth in texture space at point *P* for a given value of *t*:

$$P = (\mathbf{V}_{texture\_space} * t)$$

Consider the case shown in Figure 5.24a). ***U*** is a unit vector perpendicular to ***V*** and coplanar to ***V*** and ***N.*** *R* is the point on the quadric reached by moving distance *s* from *P* along ***U.*** That is:

$$R = P + \mathbf{U}s$$

s is obtained by substituting *R* into the equation for the quadric (Equation 5.1)

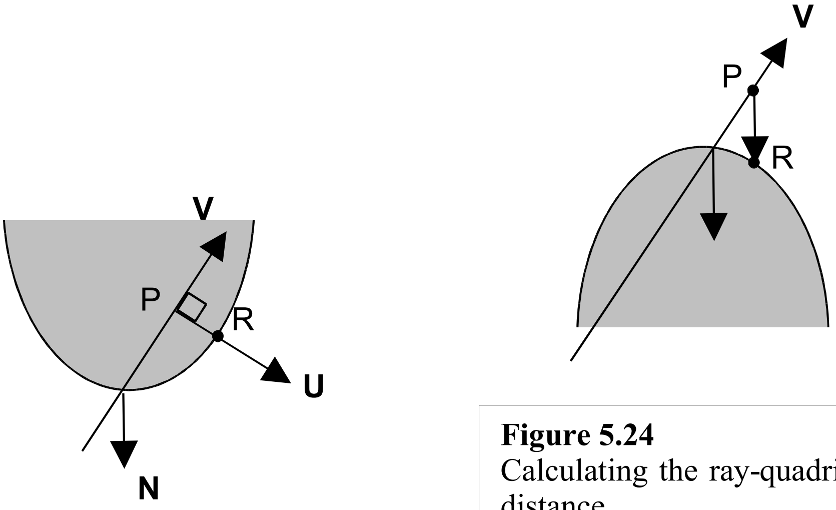


Figure 5.24
Calculating the ray-quadric distance
a) Positive curvature
b) Negative curvature

$$A(P_x + U_x s)^2 + B(P_y + U_y s)^2 - (P_z + U_z s) = 0$$

Giving the positive value of *s* as:

$$s = (-b + (b^2 - 4ac)^{1/2})/2a \qquad (5.2)$$

where:

$$a = AU_x^2 + B\,U_y^2$$
$$b = 2AP_x U_x + 2BP_y U_y$$
$$a = AP_x^2 + B\,P_y^2 - P_z$$

If the discriminant of Equation 5.2 is negative then the situation shown in Figure 5.24b) pertains and the ray quadric distance is given by:

$$d = P_z - (A.P_x^2 + B.P_y^2) \qquad (5.3)$$

which is the difference between the z coordinate of P and the z coordinate of the quadric evaluated at (P_x,P_y).

Computing the discriminant of Equation 5.2 and then evaluating Equation 5.2 or 5.3 is very costly as the procedure is repeated many times per fragment. A simpler approximate solution is to use Equation 5.3 for all cases. This gives a twofold speed-up with barely noticeable visual difference between the approximate and the accurate evaluation.

In the fragment program (Listing 5.6) we use a linear/binary search approach just like in previous relief mapping implementations. But now we use a *ray_position* function (Listing 5.5) which computes texture space position for a given search parameter *t.*

```
float3 ray_position(
       in float t,          // search parameters
       in float2 tx,              // original pixel texcoord
       in float3 v,         // view vector in texture space
       in float f)          // quadric constant for this view (A.Vx^2+B.Vy^2)
{
       float3 r=v*t;

       r.z  -= t*t*f;
       r.xy += tx;

       return r;
}
```

Listing 5.5
Ray position function

The *ray_intersect* function (Listing 5.6) now calls the *ray_position* at each search step to evaluate the current bent ray position and compares texture depth to bent ray depth for an intersection.

```
float ray_intersect_rm(
       in sampler2D reliefmap,    // relief map
       in float2 tx,              // original pixel texcoord
       in float3 v,         // view vector in texture space
       in float f,          // quadric constant for this view (A.Vx^2+B.Vy^2)
       in float tmax)       // max distance for search
{
   const int linear_search_steps=10;
   const int binary_search_steps=6;

   float t=0.0;
   float best_t=tmax+0.001;
   float size=best_t/linear_search_steps;

   // search front to back for first point inside object
   for( int i=0;i<linear_search_steps-1;i++ )
   {
             t+=size;
             float3 p=ray_position(t,tx,v,f);
             float4 tex=tex2D(reliefmap,p.xy);
             if (best_t>tmax)
             if (p.z>tex.w)
                   best_t=t;
   }

   t=best_t;
   // recurse around first point (depth) for closest match
   for( int i=0;i<binary_search_steps;i++ )
   {
      size*=0.5;
      float3 p=ray_position(t,tx,v,f);
      float4 tex=tex2D(reliefmap,p.xy);
      if (p.z>tex.w)
      {
         best_t=t;
         t-=2*size;
      }
      t+=size;
   }

   return best_t;
}
```

Listing 5.6
Ray intersect function

We can optimise the search by considering the cases detailed in Figure 5.25 for positive curvature:

- The ray is confined inside the texture space depth range *($d_{max}<1.0$)*. In this case we may or may not intersect with the relief object; we search until t_{max} at *$z=0$.*
- The bent ray crosses the greatest possible depth *($d_{max}>1.0$)*, the ray must intersect the relief object and cannot be discarded; we search until t_{max} at *$z=1$.*

For a negative curvature quadric the ray must intersect and cannot be discarded; we search until t_{max} at *$z=1$.*

To calculate the maximum search parameter *(t_{max})* we solve Equation 5.3 for *$d=0$* and for *$d=1$* and take the smaller of the two solutions. This will distribute the search samples along the minimum distance needed making the best possible use of the samples.

$$V_z . t - f . t^2 = \{ 0 \text{ or } 1 \}$$

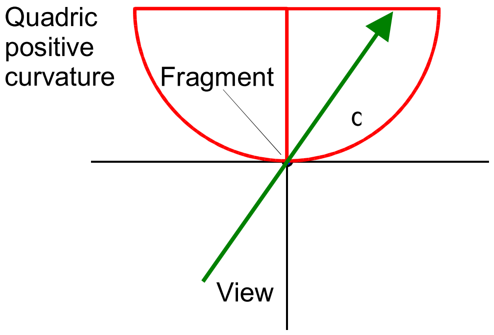
where:

$f = A.V_x^2 + B.V_y^2$ (V changes only by fragment).

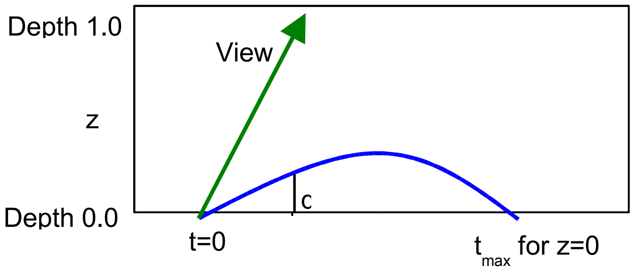
for d=0, the solution is: $-V_z/f$

for d=1, the solution is: $(V_z - \text{sqrt}(V_z^2\text{-}4.f))/(2.f)$

Tangent Space



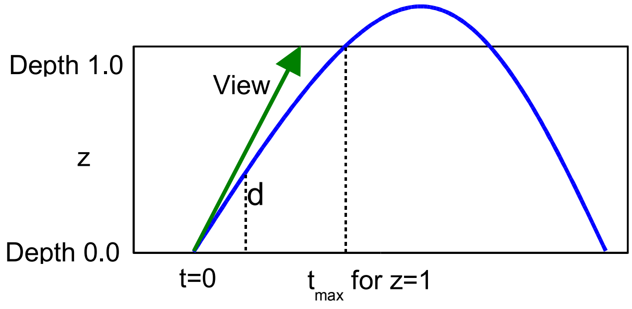
Texture Space



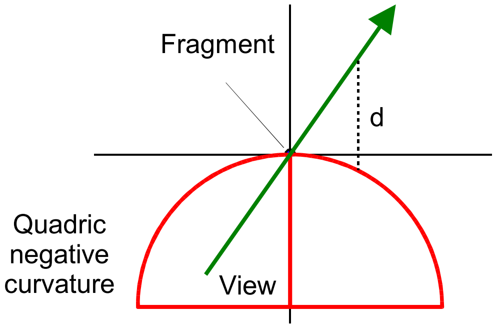
Positive curvature, two cases:

d_{max}<1, may or may not intersect,
search until t_{max} at z=0.

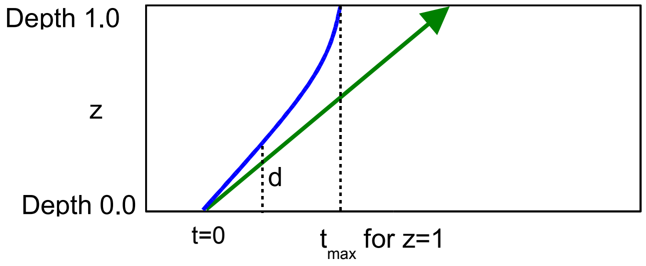
d_{max}>1, will intersect,
search until t_{max} at z=1.



Tangent Space



Texture Space



Negative curvature:

Must intersect,
Search until t_{max} at z=1.

Figure 5.25
Ray tracing a fragment with curvature as a quadric

The main function (Listing 5.7) starts by transform texture space and then calculates t_{max} (maximum s ray intersection function and if no intersection (fragment outside silhouette edge). If an intersecti colour at the intersected position in a standard lighting computation. The pixel depth is also updated to reflect the displaced position.

```
f2s main_frag_relief(
        v2f IN,
        uniform sampler2D rmtex:TEXUNIT0,     // rm texture map
        uniform sampler2D colortex:TEXUNIT1,  // color texture map
        uniform float4 camerapos,             // camera position
        uniform float4 lightpos,              // lightposition
        uniform float4 ambient,               // ambient color
        uniform float4 diffuse,               // diffuse color
        uniform float4 specular,              // specular color
        uniform float4 projz,                 // 3rd column from projection matrix
        uniform float4 planes,                        // near and far plane distances
                                              // (near,far,near*far,1/(far-near))
        uniform float tile,                   // tile factor
        uniform float depth)                  // depth factor
{
        f2s OUT;

        // view vector in eye space
        float3 view=normalize(IN.vpos);

        // view vector in tangent space
        float3 v=normalize(float3(dot(view,normalize(IN.tangent.xyz)),
                dot(view,normalize(IN.binormal.xyz)),dot(-view,normalize(IN.normal))));

        // mapping scale from object to texture space
        float2 mapping=float2(IN.tangent.w,IN.binormal.w)/tile;

        // quadric coefficients transformed to texture space
        float2 quadric=IN.curvature.xy*mapping.xy*mapping.xy/depth;

        // view vector in texture space
        v.xy/=mapping;
        v.z/=depth;

        // quadric applied to view vector coodinates
        float f=quadric.x*v.x*v.x+quadric.y*v.y*v.y;

        // compute max distance for search min(t(z=0),t(z=1))
        float d=v.z*v.z-4*f;
        float tmax=100;
        if (d>0)                // t when z=1
                tmax=(-v.z+sqrt(d))/(-2*f);
        d=v.z/f;                // t when z=0
        if (d>0)
                tmax=min(tmax,d);

        // ray intersect depth map
        float t=ray_intersect_rm(rmtex,IN.texcoord*tile,v,f,tmax);
        if (t>tmax)
                discard; // no intesection, discard fragment

        // compute intersected position
        float3 p=ray_position(t,IN.texcoord*tile,v,f);

        // get normal and color at intersection point
        float4 n=tex2D(rmtex,p.xy);
        float4 c=tex2D(colortex,p.xy);

        // compute displaced pixel position in view space
        p=IN.vpos.xyz+view*t;

#ifdef RM_DEPTHCORRECT
        // compute currected depth value for displaced pixel
        float linear_depth=dot(-p,projz.xyz)+projz.w;
        // (far*near/depth+far)/(far-near)
        OUT.depth=(planes.z/linear_depth+planes.y)*planes.w;
```

```
#endif

        // compute light direction
        float3 l=normalize(p-IN.lightpos.xyz);

        // expand normal from normal map into view space
        n.xyz=n.xyz*2.0-1.0;
        n.xyz=normalize(n.x*IN.tangent.xyz+n.y*IN.binormal.xyz+n.z*IN.normal);

        // compute diffuse and specular terms
        float att=saturate(dot(-l,IN.normal));
        float diff=saturate(dot(-l,n.xyz));
        float spec=saturate(dot(normalize(-l-view),n.xyz));

        // compute final color
        float4 finalcolor=ambient*c;
        finalcolor.xyz+=att*(c.xyz*diffuse.xyz*diff+specular.xyz*pow(spec,specular.w));
        finalcolor.w=1.0;

        OUT.color=finalcolor;
        return OUT;
```

Listing 5.7
Main Function – rendering a polygonal object with RTM for correct silhouette edge

Examples of this final variation of the RTM approach are shown in Figure 5.26. In the case of the teapot, the rendering polygonal resolution (1600 triangles) is superimposed in red wireframe. This demonstrates the power of the method: high quality detail rendering, including silhouette edge, using a low polygon count. The sphere demonstrates the use of a high depth RTM.

Although the shaders are long, the teapot example renders at greater than 100 frames per second. The cost is predominantly a function of the precision of the linear search which itself is dependent on the size of the smallest detail in the map. Another factor which determines cost is the maximum depth of the relief map which, by definition, requires more search samples.

A not insignificant advantage is that authoring content is straightforward. If you have a tool for creating normal maps then adding depth is simple. Finally once the map is created it requires no extra processing as is the case with a related method [WANG03] – a technique called VDM or View-dependent Displacement Mapping.

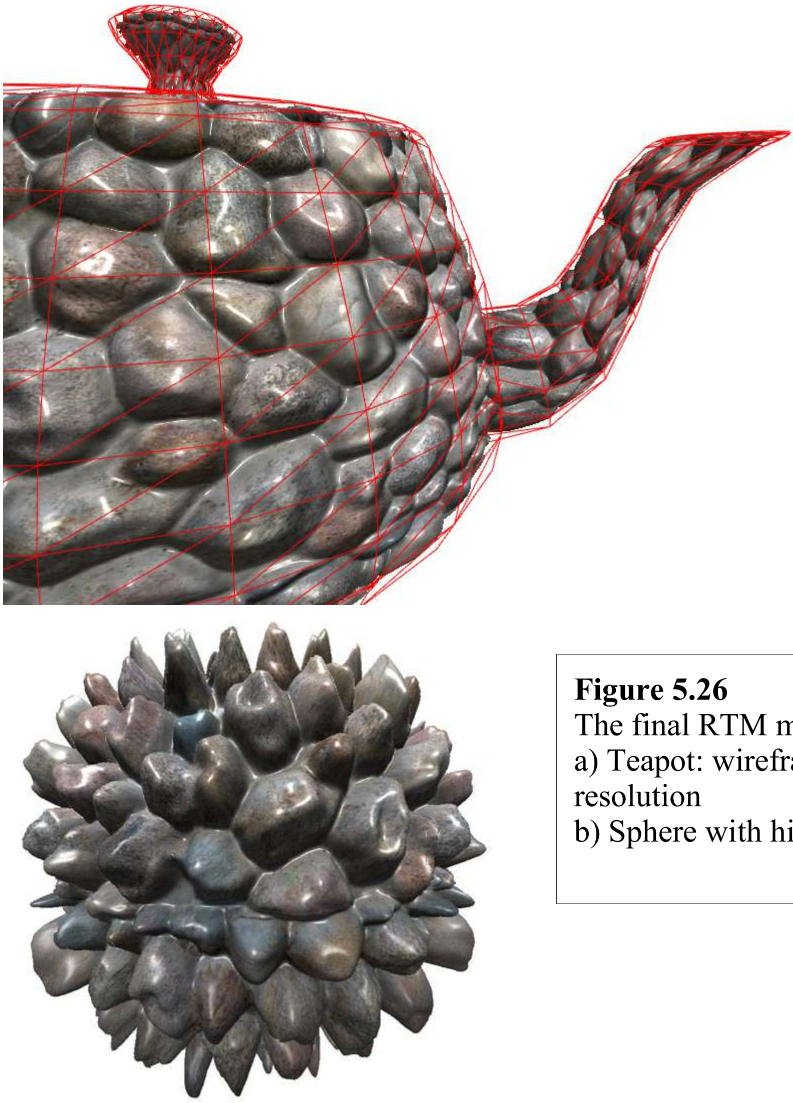


Figure 5.26
The final RTM method.
a) Teapot: wireframe shows rendering resolution
b) Sphere with high depth RTM

Per vertex curvature estimation using quadrics

In the previous section we described a method that will allow us to render correctly the geometry of silhouette edges. To do this we need to calculate the surface curvature at any point on the mesh. Differential geometry gives us a way of measuring the curvature of a surface at a point *P*. If we consider that for a very small region surrounding *P,* our surface is locally planar, we can say informally that curvature is related to how rapidly the surface departs from the tangent plane at *P*. This leads us to say that, for a sufficiently small neighbourhood around *P,* to a first approximation the curvature is given by a plane. If the surface is twice differentiable then it can be shown (see, for example [PETI02]) to a second approximation a small neighbourhood around *P* takes the form of a so-called osculating paraboloid or quadric of one of 9 types. One way of measuring curvature therefore is to find the coefficients of the quadric that best fits the surface. We can restrict ourselves, for efficiency in the method, to a subset of 4 of the 9 quadrics without losing too much accuracy. These are the quadrics defined by the simple quadratic equation:

$$Ax^2 + By^2 + z = 0$$

where:

A = B = 0 defines a plane
A or B = 0 defines a parabolic cylinder
A and B have the same sign defines a paraboloid
A and B have different signs defines a hyperbolic paraboloid

and we can thus classify the curvature at a point *P* by calculating the coefficients *A* and *B*. These forms are shown in Figure 5.27. If we can calculate *A* and *B* for a polygon mesh we can then consider the surface in the vicinity of the vertex to possess one of these 4 forms.

Having calculated *A* and *B* we can pass these parameters into a vertex program and the interpolator will produce values at each fragment. Thus we have at each fragment a quadric centred at the pixel with orientation aligning with the pixel tangent space. The pixel is always point (0,0,0) in quadric space.

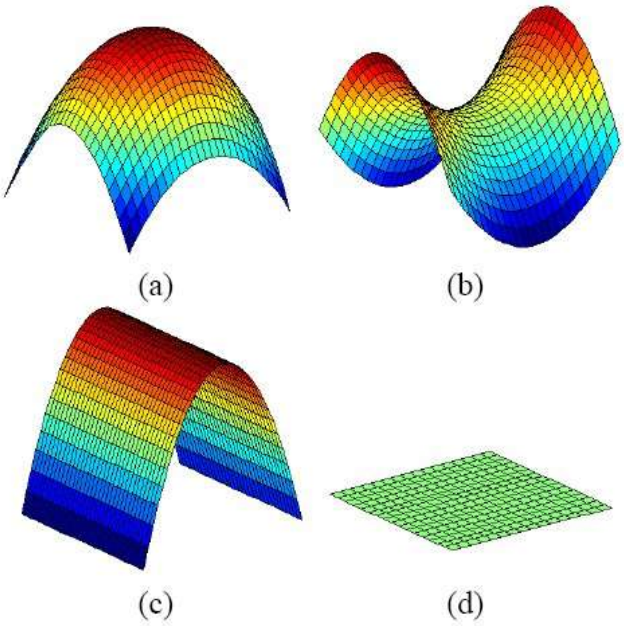


Figure 5.27
Quadrics defined by the equation:
$Ax^2 + By^2 + z = 0$
a) Paraboloid
b) Hyperbolic paraboloid
c)Parabolic cylinder
d) plane

In order to find the coefficients A and B for a given vertex we use the vertex ring neighbourhood around it. We simply transform the neighbouring vertices to the tangent space of the vertex tangent space and inject them the following equation form (for *n* neighbor vertices):

$$\begin{bmatrix} x_0^2 & y_0^2 \\ x_1^2 & y_1^2 \\ x_2^2 & y_2^2 \\ \dots & \dots \\ x_n^2 & y_n^2 \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix} = \begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ \dots \\ z_n \end{bmatrix}$$

This is a simple over determined system we need to solve in order to find A and B. We can use a least squares algorithm to quickly resolve the system. A matrix system in form *Ex=F* can be solved as follows:

$$E \cdot x = F$$
$$E^T \cdot E \cdot x = E^T \cdot F$$
$$x = (E^T \cdot E)^{-1} \cdot (E^T \cdot F)$$

In our case the E matrix with *(n x 2)* elements have all neighbor vertices *x* and *y* positions squared (one line per vertex). The *F* matrix with *(n x 1)* elements has all *Z* positions (as the vertex is in tangent space of centre vertex this will be the distance to the plane defined by the centre vertex normal).

Thus all we need is to multiply the matrix E with its transpose resulting in a *(2 x 2)* matrix. Then we multiply E transpose with F resulting in a *(2 x 1)* matrix. To find our quadric curvature parameters A and B we simply multiply the two previous matrices together resulting in a *(2 x 1)* matrix with *A* and *B* values. The code is given in Listing 5.8.

```
// solve matrix system Ex=F using least squares
// E[n,2] and F[n,1] result stored in (x,Y)
// R = (RT*R)^-1 * (RT*F)
void pMesh::solve_least_squares(
       int n,const float *E,const float *E,float& x,float& y)
{
       x=0;
       y=0;
       if (n<3) return;

       int i,j,k;

       // ET = transpose of E
       float *ET=new float[n*2];
       for( i=0;i<n;i++ )
       {
              ET[i]=E[i*2];
              ET[i+n]=E[i*2+1];
       }

       // M = ET * E
       float M[2][2];
       for( j=0;j<2;j++ )
       for( i=0;i<2;i++ )
       {
              M[j][i]=0;
              for( k=0;k<n;k++ )
                     M[j][i]+=ET[j*n+k]*E[k*2+i];
       }

       // MI = inverse M
       float det,MI[2][2];
       det=M[0][0]*M[1][1]-M[0][1]*M[1][0];
       if (fabs(det)>0.001f)
       {
              det=1.0f/det;
              MI[0][0]=M[1][1]*det;
              MI[0][1]=-M[0][1]*det;
              MI[1][0]=-M[1][0]*det;
              MI[1][1]=M[0][0]*det;

              // ETB = ET * F
              float ETF[2]={ 0,0 };
              for( i=0;i<2;i++ )
              for( j=0;j<n;j++ )
                     ETF[i]+=ET[i*n+j]*F[j];

              // (x,y) = MI * ETF
              x=MI[0][0]*ETF[0]+MI[0][1]*ETF[1];
              y=MI[1][0]*ETF[0]+MI[1][1]*ETF[1];
       }

       delete ET;
}
```

Listing 5.8
Solves matrix system *Ex=F* using least squares

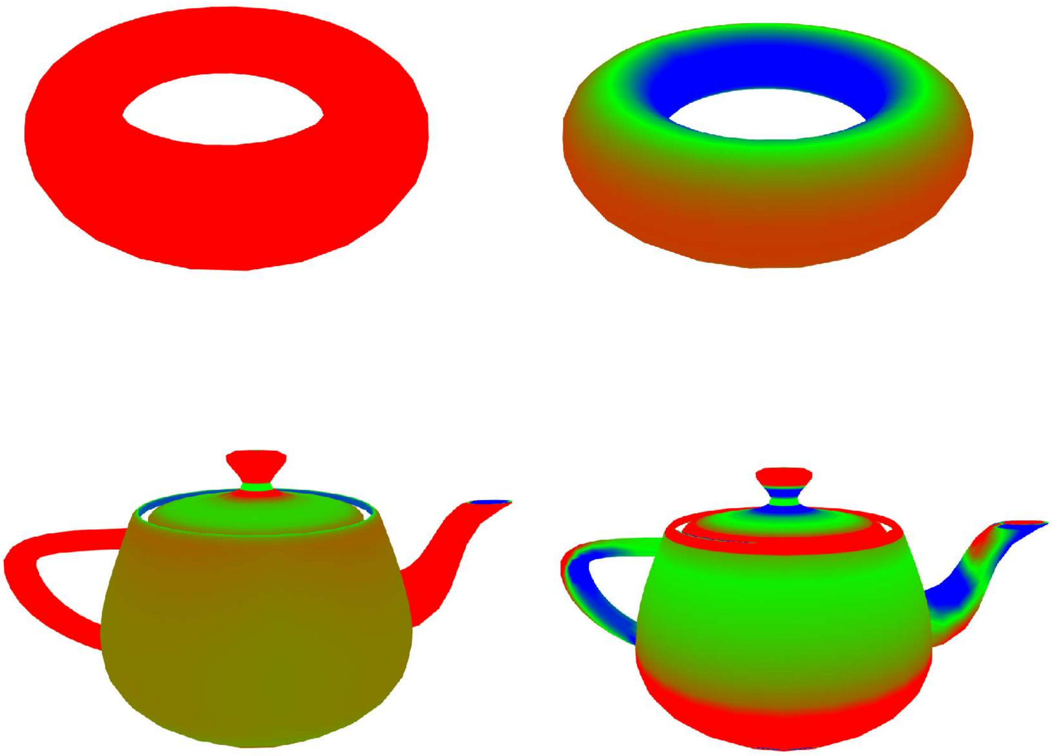


Figure 5.28
Colour representing curvature for the teapot and a torus
Red = positive curvature
Blue = negative curvature
Green = planar
Left is tangent direction and right is binormal direction

Figure 5.28 shows results from the method: the curvature for a torus and a teapot (red means positive curvature, blue negative and green planar). Left images for curvature on tangent direction and right images for binormal direction.

Omni-directional displacement maps (ODD)

This final application is useful for caching caches surface in detail that tiles infinitely (a brick wall, for example).

Such surfaces are commonly used in computer games levels and architectural applications. The application uses what is effectively a 4D texture map encoded as a 2D map and has very large memory requirements – the reason for restricting it to a tileable surface. It is similar in effect, but very different in detail to a (periodic) bump map.

We begin by considering the rendering principle since this will enable us to understand the construction of the 4D map better. To render a point on the surface of an object we calculate a 4D texture coordinate *(x,y,a,b)*. The *(x,y)* coordinate is the conventional pre-authored mapping and the *(a,b)* component is calculated. Each *(x,y)* pixel is a 'super' pixel - an array of 16x16 *(a,b)* pixels. This array contains information associated with 16x16 rays which is the surface normal ***N*** at the point intersected by each ray and the distance *z* of that intersection along the ray.

Figure 5.29 shows the idea in some detail. Rays are cast in a pre-calculation phase from a plane positioned close to the tileable surface (i.e. *h* should be small w.r.t. w as shown in Figure 5.30). For a current view direction ***V*** we calculate the intersection angle α of the view ray with the surface (Figure 5.29a). The *(a,b)* coordinate is calculated from α and ***V*** and used to index the 16x16 array; thus matching the current view ray to the stored directions (Figure 5.29b). The texture pixel *(*N,*z)* is then used to render in the normal manner. The distance *z* along the ray enables a more accurate specification of ***L*** – one which includes the distance from the light.

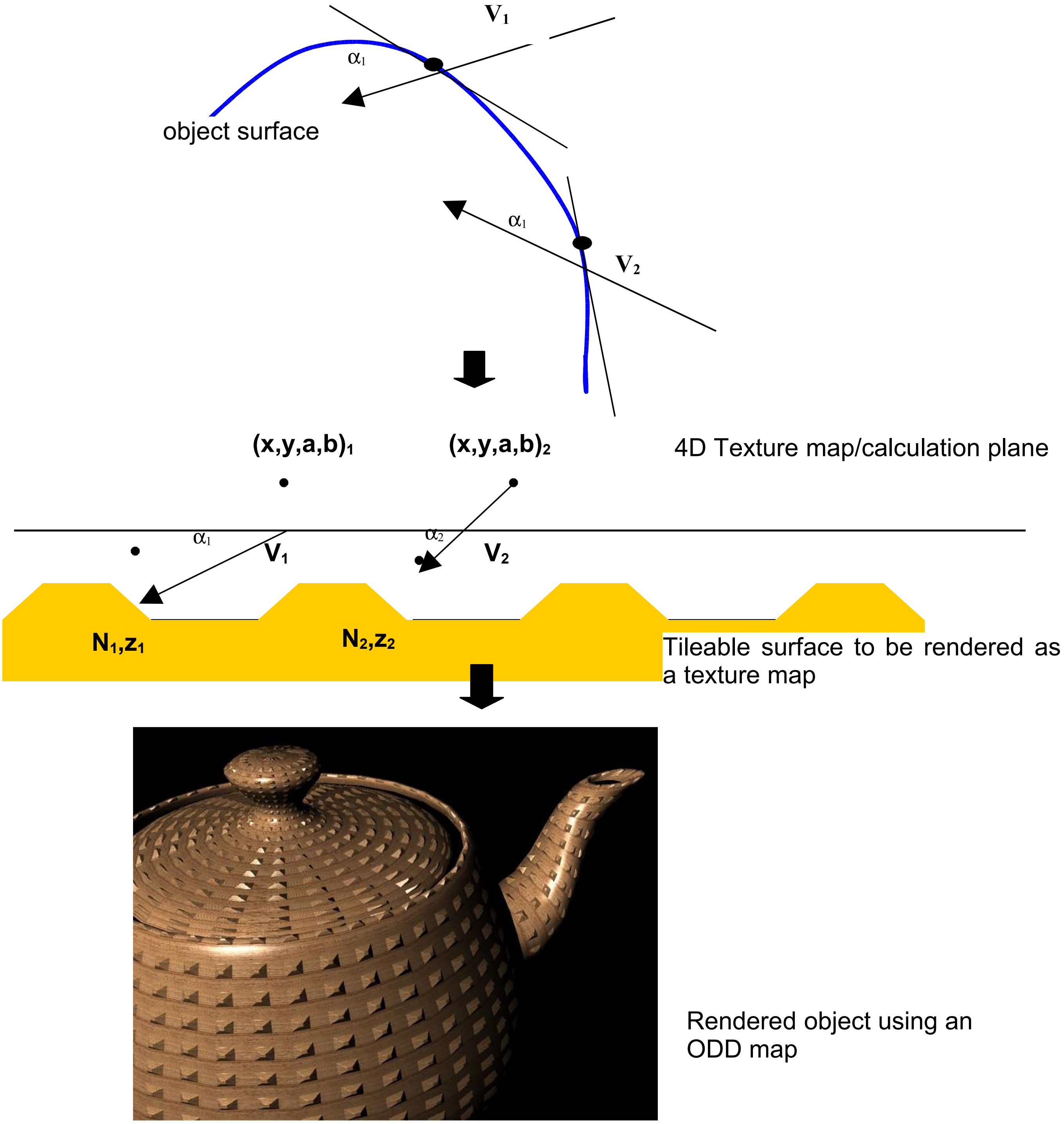


Figure 5.29
Rendering sequence in omni-directional depth mapping (ODD)

The vertex/fragment program division is as follows. The vertex program transforms ***L*** and the tangent space vectors to view space and the fragment program calculates the 4D texture coordinate.

We now look in some detail at the construction of the 4D texture map, how it caches the information we require and how the 4D coordinates are calculated in the render phase. For each of 32 x32 (say) super pixels we cast 16x16 (say) rays into a hemisphere placed on a calculation or cache surface placed close to the tileable surface. This is shown in Figure 5.30 where 4 rays in 2D space are illustrated. The ray directions are given by:

$$R_x = \cos(a)\sin(b)$$

$$R_y = \cos(b) \qquad\qquad 0 <= a,b <= \pi$$

$$R_z = \sin(a)\sin(b)$$

These rays intersect the geometry of the tileable surface and the surface normal ***N*** at the point of intersection, together with the distance *z* to the intersection, is stored in the pixel.

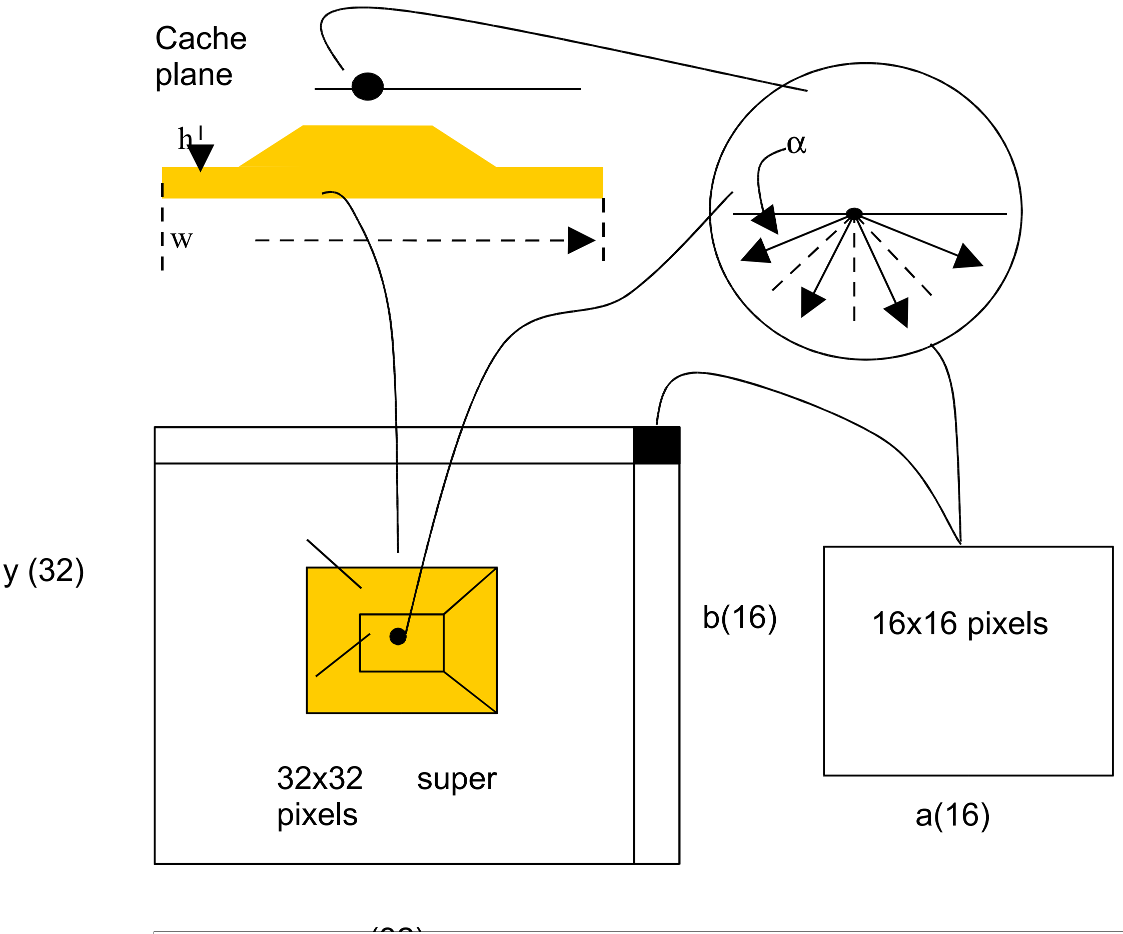


Figure 5.30
4D texture map (32x32x16x16) – for each pixel 16x16 rays are cast into a hemisphere positioned on the calculation plane.

At render time to retrieve the duple (*N,z)* we need to calculate a 2D texture map coordinate (u,v) from the 4D coordinate (x,y,a,b). The (x,y) coordinate is the normal database texture coordinate. The (a,b) coordinate is calculated for a particular (x,y) by finding the pixel that contains the direction closest to the view direction:

$$a = \cos^{-1}\left(\frac{A}{(1\text{-}B^2)^{\frac{1}{2}}}\right)/\pi$$

$$b = \cos^{-1}(B)/\pi$$

where: A = **T.V**
B = **B.V**

V is the view vector
T is the tangent vector

B is the binormal

Now that we have the 4D coordinate *(x,y,a,b)* we can calculate the 2D *(u,v)* coordinate to index into the map:

u = (floor(x,32)+a)/32

v = floor(y,32)+b)/32

We observe that we can order the 4D map in two ways (x,y,a,b) or (a,b,x,y). The relevance of this is that the different orders will exhibit different image structure, which then has ramifications for the anti-aliasing hardware for the texture mapping.

For the pin/pyramid surface used in Figure5.29 we show in Figure 5.31 the (a,b,x,y) organization. This colour or RGB encoding shows the cached normals. Thus the centre super pixel of this map is a 32x32 array which stores the information gathered by all the vertical arrays. This gives a perfect normal image of a pin or pyramid with four normals together with the floor normal. As we move away from the centre of the map one or two faces predominate depending on the direction. At the edges of the map there is spatial or geometric aliasing because the ray sampling density is low and the distance is large.

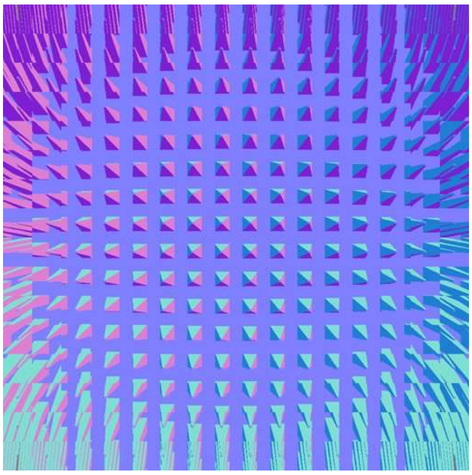


Figure 5.31
An (a,b,x,y) organized 4D texture map for the pin/pyramid surface

Finally Figure 5.32 shows an *(x,y,a,b)* and an *(a,b,xy)* organization for a brick surface. Objects rendered using this method are shown in Figure 5.33. The *(x,y,a,b)* shows the information associated with the 16x16 ray directions for a single *(x,y)* super pixel. Compared to the pin surface the *(a,b,x,y)* does not exhibit spatial aliasing. This is because the height of the bricks above the plane on which they are placed is less than the height of the pins.

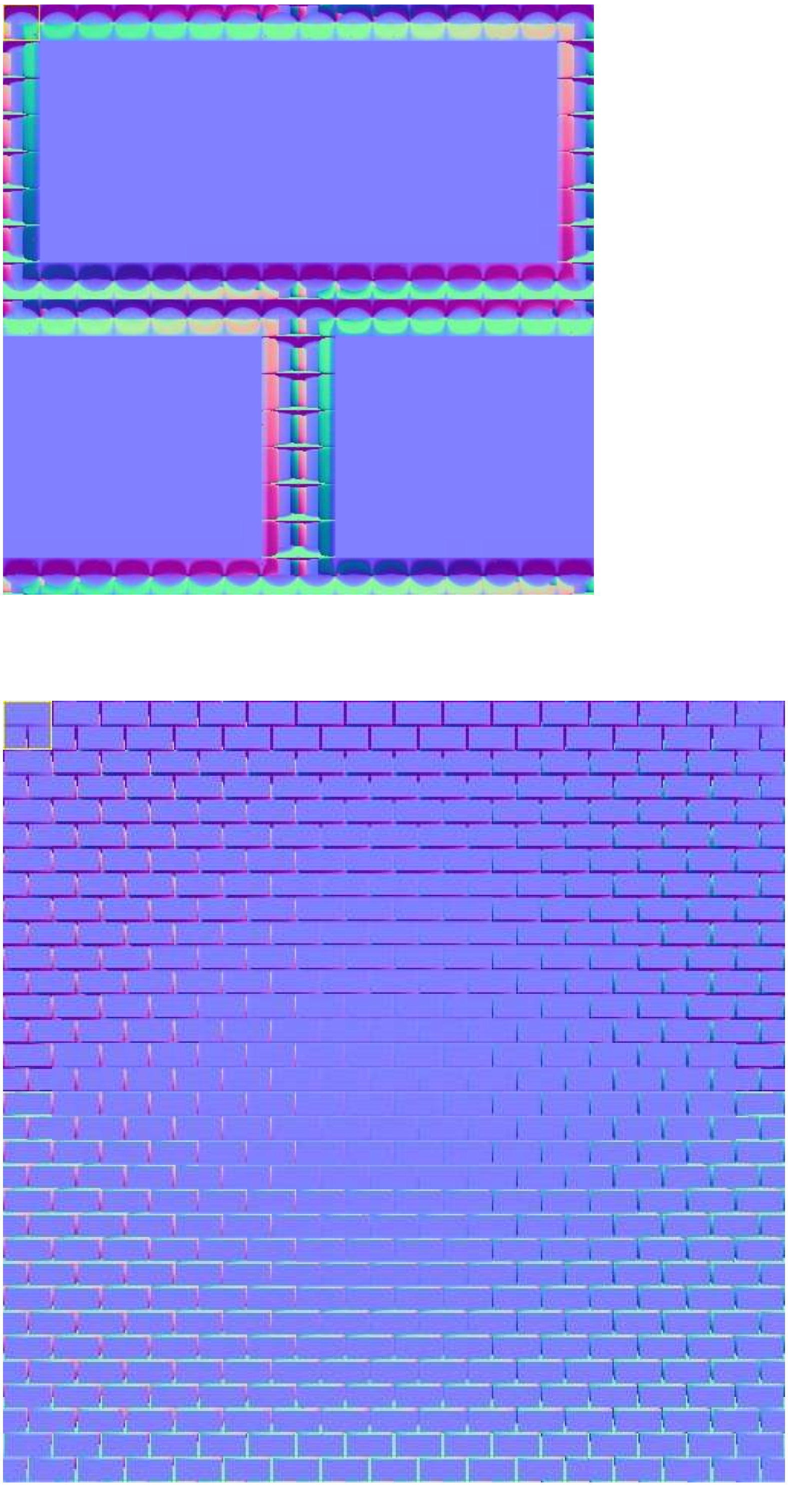
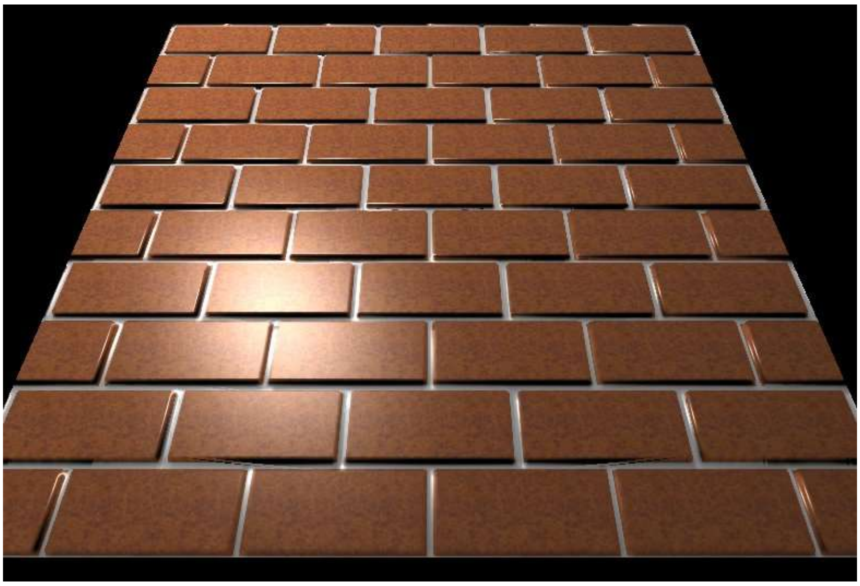


Figure 5.32
a)An *(x,y,ab)* organization for a brick surface
b)An *(a,b,x,y)* organization for the same brick surface



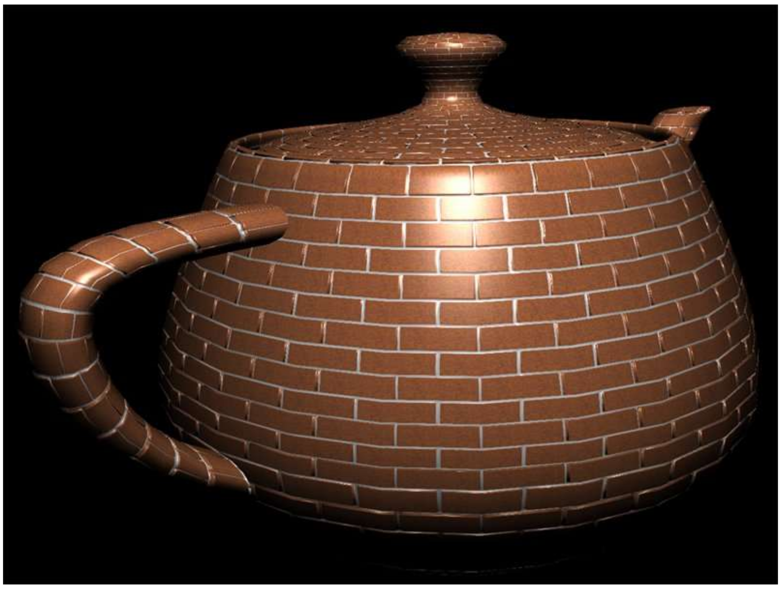


Figure 5.33
Two objects rendered using the brick texture

Code for the complete shader is given in Listing 5.9

```
v2f displace_map_vert(
        a2v IN,
        uniform float4 lightpos,
        uniform float4x4 view,
        uniform float4x4 modelview,
        uniform float4x4 modelviewproj)
{
        v2f OUT;

        // vertex position in object space
        float4 pos=float4(IN.pos.x,IN.pos.y,IN.pos.z,1.0);

        // compute modelview rotation only part
        float3x3 modelviewrot;
        modelviewrot[0]=modelview[0].xyz;
        modelviewrot[1]=modelview[1].xyz;
        modelviewrot[2]=modelview[2].xyz;

        // vertex position in clip space
        OUT.hpos=mul(modelviewproj,pos);

        // vertex position in view space (with model transformations)
        OUT.vpos=mul(modelview,pos).xyz;

        // light position in view space
        OUT.lightpos=mul(view,float4(lightpos.x,lightpos.y,lightpos.z,1));
        OUT.lightpos.w=lightpos.w;

        // tangent space vectors in view space (with model transformations)
        OUT.tangent=mul(modelviewrot,IN.tangent);
        OUT.binormal=mul(modelviewrot,IN.binormal);
        OUT.normal=mul(modelviewrot,IN.normal);

        // copy color and texture coordinates
        OUT.color=IN.color;
        OUT.txcoord=IN.txcoord.xy;

        return OUT;
}

float4 displace_map_frag(
        v2f IN,
        uniform float4 diffuse,
        uniform float4 specular,
        uniform float4 constants,
        uniform float2 tile,
        uniform float depth,
        uniform sampler2D texmap,
        uniform sampler2D dispmap,
        uniform sampler2D dispindxmap) : COLOR
{
        float4 t;
        float3 viewdir,lightdir,halfdir;
        float2 uv,uv1,uv2,uv3;
        float HdotL,NdotL;

        // compute directions
        viewdir=normalize(IN.vpos);
        lightdir=normalize(IN.lightpos.xyz-IN.vpos);
        halfdir=normalize(lightdir-viewdir);

        // normalize tangent space
        float3 tangent=normalize(IN.tangent);
        float3 binormal=normalize(IN.binormal);
        float3 normal=normalize(IN.normal);

        // project viewdir XY into tangent space (uv1)
        uv1.x=dot(viewdir,tangent);
        uv1.y=dot(viewdir,binormal);

        // displace AB texture coordinate (uv2)
        // simple mapping
        t=f4tex2D(dispindxmap,uv1.xy*0.5+0.5);
        uv2=t.xy+t.zw*0.005;
        // full mapping
//      uv2=1.0-0.318309886*float2(acos(uv1.x/sqrt(1.0-uv1.y*uv1.y)),acos(uv1.y));

        // compute displace XY texture coordinate (uv3)
        uv3=fmod(IN.txcoord.xy*tile,1.0);

        // compute 4D->2D displace texture coordinate (uv2,uv3)->(uv)
        uv=(floor(uv2*constants.xy)+uv3.xy)*constants.zw;

        // get displace pixel with normal (xyz) and depth (w)
        t=f4tex2D(dispmap,uv)-float4(0.501961,0.501961,0.501961,0.0);
        if (depth==0) t=float4(0,0,1,0);

        // project normal into tangent space
        t.xyz=normalize(tangent*t.x+binormal*t.y+normal*t.z);
        // diffuse texture mapping (using depth as offset)
        diffuse*=f4tex2D(texmap,uv3+uv1*t.w*depth);

        // diffuse and specular lighting
        NdotL=saturate(dot(lightdir.xyz,t.xyz));
        HdotL=saturate(dot(halfdir,t.xyz));
        t.xyz=diffuse.xyz*NdotL+specular.xyz*pow(HdotL,specular.w);
        t.xyz*=sqrt(saturate(dot(normal,lightdir)));
        t.w=diffuse.w;

        return t;
}
```

Listing 5.9
Vertex and fragment shader for ODD mapping