Relief Mapping in a Pixel Shader using Binary Search

Policarpo, Fabio (fabio@paralelo.com.br)

1. Introduction

The idea of this shader (called Relief Mapping) is to implement a better quality Bump Mapping[1] technique similar to Parallax Mapping[2] (also called Offset Bump Mapping) and Relief Texture Mapping[3]. Actually, Relief Mapping corresponds to an inverse (or backward) approach to Relief Texture Mapping. It uses a standard normal map encoded in the RGB portion of a texture map together with a depth map stored in the alpha channel as shown in Figure 1.

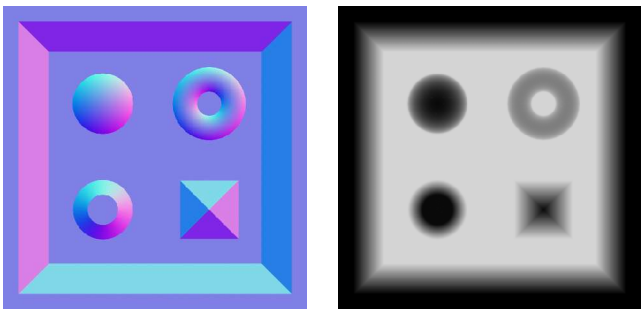


Fig 1. Normal map and depth map that comprise an example relief map

2. How it works

In Parallax Mapping we displace the texture coordinates along the view direction by an arbitrary factor times the depth value stored in the depth map. This adds a nice displacement effect at a very low cost but is only good for noisy irregular bump, as the simplified displacement applied deforms the surface inaccurately.

Here in Relief Mapping we want to find the exact displacement value for every given fragment making sure all forms defined by the depth map are accurately represented in the final image.

In Relief Mapping, the surface will match exactly the depth map representation. This means that we can represent forms like a sphere or pyramid and they will look correct from any given view angle.

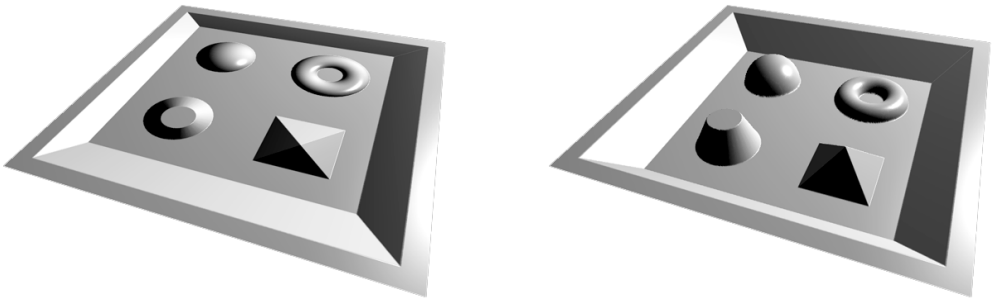


Fig 2. Bump map (left) and relief map (right) comparison. One pass single quad polygon drawn.

---

[1] Bump Mapping – Blinn, Jim
http://research.microsoft.com/users/blinn/

[2] Parallax Mapping – Welsh, Terry
http://www.infiscape.com/doc/parallax_mapping.pdf

[3] Relief Texture Mapping – Oliveira, Manuel - SIGGRAPH2000
http://www.cs.unc.edu/~ibr/pubs/oliveira-sg2000/RTM.pdf

3. Adding Texture

Adding a color texture map makes the effect look sharper and helps to hide any artifacts from lighting and low map resolutions. With Relief Mapping the texture map projects correctly along the geometry and you can see the mapping deformations as it follows any curved surfaces much better than in Bump Mapping (where they look somehow flat). Figure 2 shows the same examples from Figure 1 but using a wood texture map.

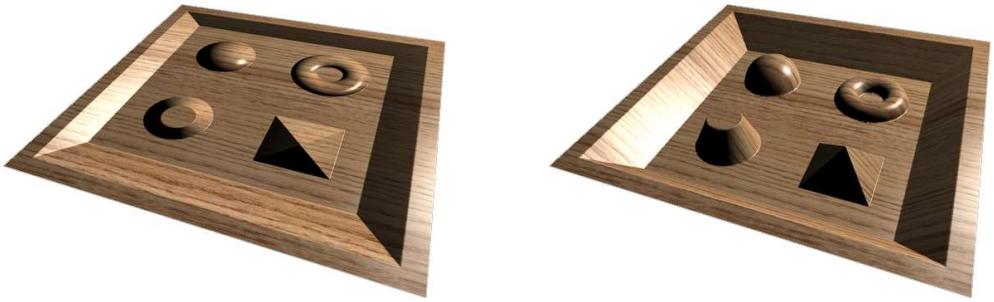


Fig 3. Normal map (left) and relief map (right) with color texture mapping

3. Adding Self Shadows

Another good thing with Relief Mapping is that we can add self shadows without much trouble. This makes the effect even better giving a better perception of apparent depth. Figure 4 shows two images from two different light positions creating a nice and detailed self shadows effect over the object.

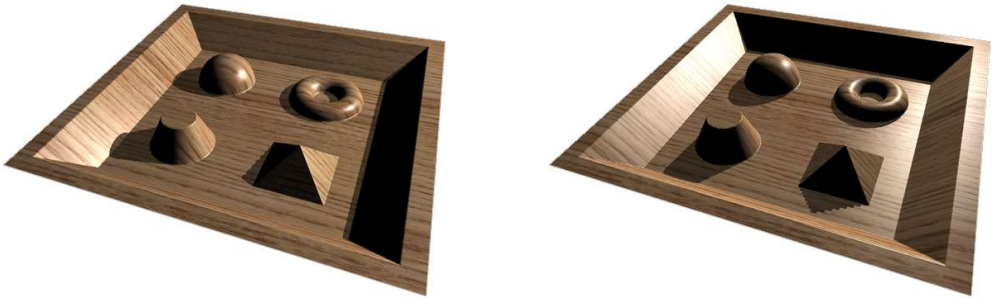


Fig 4. Self shadows using two different light positions

4. Advantages and Disadvantages

Advantages:

- Much better looking using same resources as Parallax Mapping (same texture)
- Self-shadows add to the effect
- Can be tiled many times across polygon
- Lighting is in object space so we can transform relief polygon anywhere in scene

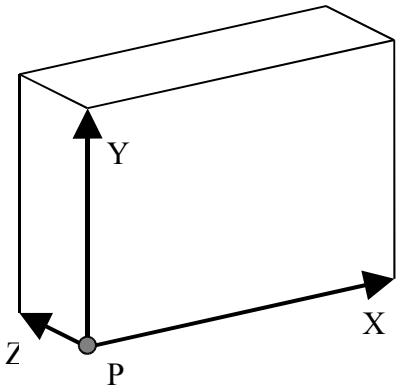
Disadvantages:

- Complex pixel shader with about 200 assembler shader instructions (against 30 for a simple bump or parallax mapping effect).
- Shadows requires doubling the work, ending up with around 400 assembler instructions.
- Currently can only be applied to planar surfaces

5. How it really works

To find out exactly the correct value to displace to apply to each pixel's texture coordinate, we implemented a function that ray intersects the depth map. A start position (fragment position) and direction (view direction through fragment), is passed into the function which returns the depth for the closest intersection. This is all we need to accurately displace the texture coordinate for this fragment.

We start with the relief object as an oriented bounding box defined by a position and three vector as in Figure 5. The rectangle defined by X and Y is the quad actually drawn and Z is used to define the depth range.




```
Relief object box defined by point
P and 3 vectors (X, Y, Z).

X and Y will be the relief map
axis and Z will give the maximum
depth.

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

Fig 5. Relief object definition

In Figure 6 we see a cross section of an example relief object. We are rendering fragment ***A*** and for that we call the ray intersection function passing point ***A*** and direction (***B-A***). This will return depth ***d1*** and point ***P*** can be defined as ***A*** + ***d1****(***B-A***).

If we want to compute shadows, we can simply call the ray intersection function once again but now from point ***C*** with direction (***D-C***). This new ray intersection call will return ***d2*** and comparing ***d1*** to ***d2*** we can determine if the fragment is in shadow (if ***d2<d1*** we are in shadow).

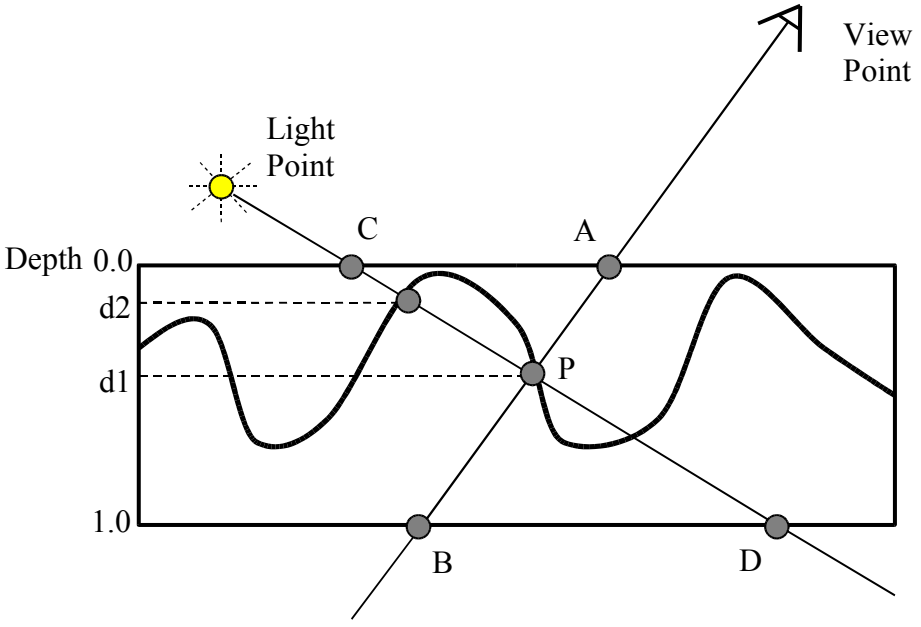


Fig 6. Relief Mapping cross section

The following pseudo code gives more details about how points ***B***, ***C*** and ***D*** are calculated using only point ***A***, the view and light directions, and the ray intersect function.

Pseudo code (main):
Relief object box defined by P, X, Y, Z (as in Figure 5).
Currently rendering pixel A (from Fig 2).

```
float4 main_rm()
{
     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_rm(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_rm( C, V );
     if (d2<d1)
               // pixel in shadow
               color = shadow_color();
     else
               // apply lighting
               color = phong_lighting();
     return color;
}
```

The ray intersection function that returns the depth of the closest intersection along the ray through the depth map. It uses a binary search to quickly converge to the solution with only a few steps (using a binary search we can locate an element among a list of 256 with only 8 tests as 2^8=256).

So given a segment ray through the relief object, we will loop for a fixed number of times and at each step we will get closer to the object surface. We start by testing the pixel at half depth (0.5). If this pixel is outside the object (depth map value is bigger than current depth of 0.5) we move forward adding 0.25 to the current depth. But if pixel was inside the object (depth map value smaller the current pixel depth) we store this as a good solution and move backwards (subtracting 0.25 from current depth). At each step we add/subtract half the previous value (0.25, 0.125, 0.0625, ...).

But there is a problem with using only a binary search to get our solution. As the depth map can represent objects that are not convex (some parts can occlude other parts) simply using the binary search can result in errors as shown in Figure 7. In this example, the ray intersects the object in 3 points. We start at depth 0.5 (hollow circle) where we are clearly outside the object... this will move us forward and we will end up in the wrong intersection (C instead of A which would be the correct one in this case).

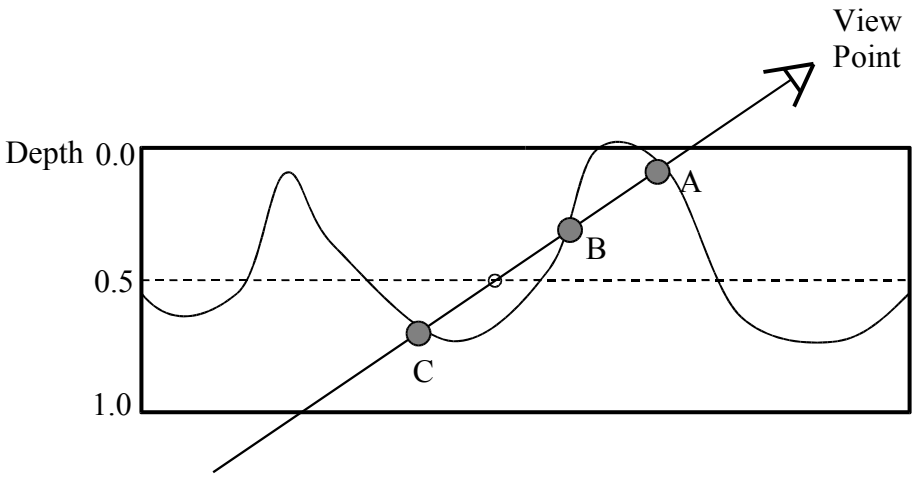


Fig 7. Using only a Binary Search can generate errors as in this case

To solve this, we use a linear search with large steps first in order to find any point inside the first section of the object and then use the binary search to refine our solution around that point.

Say we decide for a 10 step linear search (which require only 9 tests with 1/10=0.1 intervals), we test point at depth 0.1, then 0.2, then 0.3 until we find a point that is inside the object. Then at this point we apply a binary search with start interval equal to the interval size.

If we guarantee all parts of the object intersected by the ray will result in ray segments larger than 0.1 we will have no problems. This size is the thinnest object protrusion we can represent without errors (anything thinner than this would get lost by the linear search sampling). These errors can be noticeable on shadow edges from the pyramid 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 8. You can always increase the number of liner steps but it will always be possible to have thinner object parts that will cause this problem.

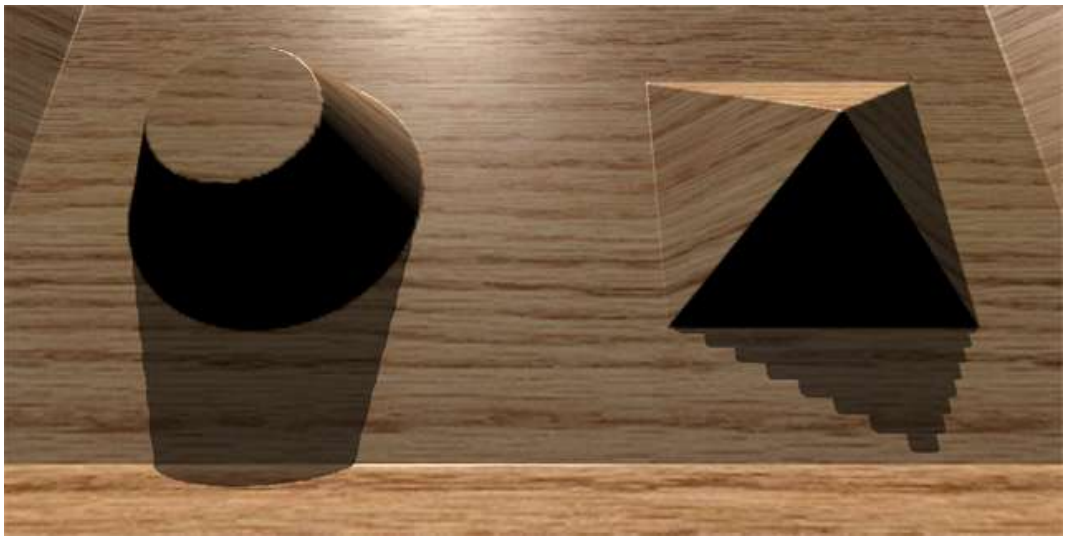


Fig 8. Artifacts generated by too thin object parts (cone is ok, pyramid has thin edges)

Pseudo code (ray_intersect_rm):

```
float ray_intersect_rm( 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 = relief map 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 = relief map 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;
}
```

7.Texture Filtering and Mip-mapping

Another nice feature is that both texture filtering and mip-mapping work fine with this shader. Texture filtering will allow nice normal interpolation around surface corners and mip-mapping will speeds up the shader considerably (especially when using large tiles where lower mip-map levels are used). Both the normal and depth components are mip-mapped to lower resolutions.

Also full scene anti-aliasing and anisotropic texture filtering will not generate artifacts and can be used freely.

8. PS 3.0 Optimizations

The new pixel shaders version 3.0 available in latest nVidia NV40 based video cards allows variable loops and true if/else statements. The Relief Mapping shader could take advantage of it specially 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.

But using loops in your shader also adds a big penality in execution and tests show that the version using unrolled loops performs better in current available hardware. Maybe in future hardware generations it might be better to use true loops and early exiting them when possible.

9. Double Depth

For relief maps that do not fill in all image space we can use a double depth relief map. In this case we add 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 9. 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.

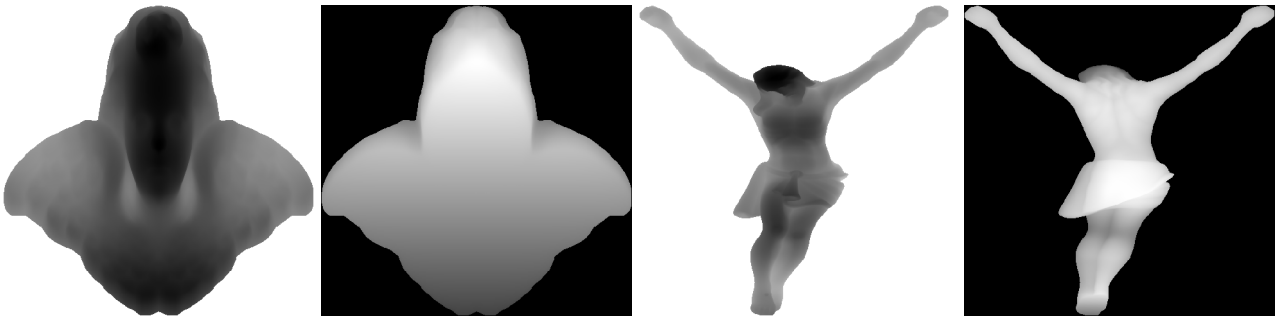


Fig 9. Double depth relief maps (Angel and Christ). Front and back depth maps encoded in blue and alpha channels.

Here will only consider a point to be inside the object if its depth is bigger than the front depth map and smaller than the back depth map.

$$FrontDepth \leq PointDepth \leq BackDepth$$

The lost normal Z component (now replaced with the back depth map) can be recovered in the fragment shader using the following formula:

$$Z = \sqrt{1 - X^2 - X^2}$$

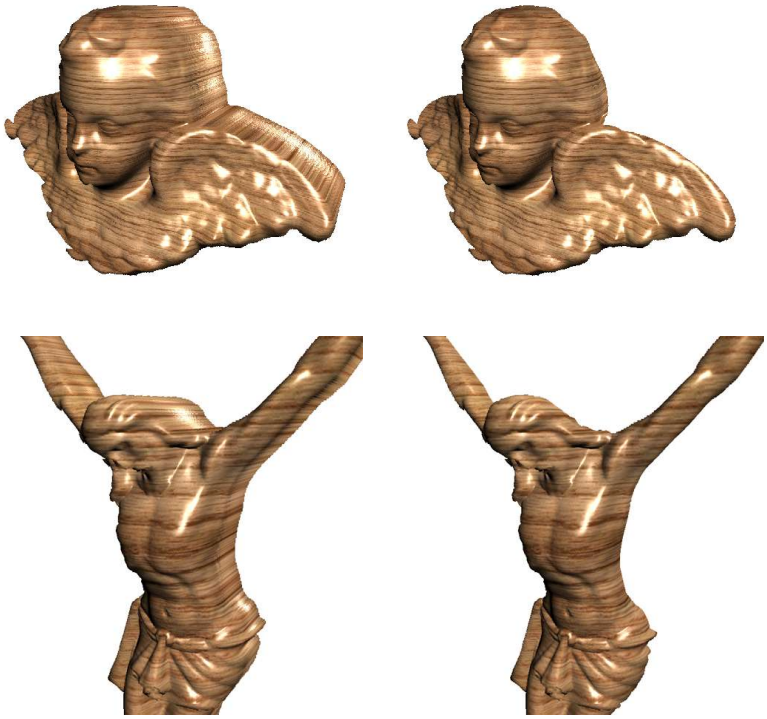


Fig 10. Angle and Christ relief maps using double depth (back of figures are caped by back depth map).

10. Conclusions

We think this new method is an interesting substitute for standard Bump Mapping and Parallax Mapping in cases where object detail is not smaller the safe distance defined by linear search step size. Finer detail could be included only in the normal map component using a sort of mixed Relief/Bump Mapping (general lower frequency structure defined by relief and small detail using only bump). It runs in a single pass and can be easily integrated to current available rendering systems that makes use of pixel shaders for lighting. It uses the same normal and depth texture maps already available and required by such systems and produces better quality images in all cases (even at highly parallel angles).

Current graphics hardware (nVidia GeForce 6800 for example) can run this effect at monitor refresh rates (85 fps) in full screen mode even with the relief polygon taking over all screen space.

We hope to see nice quality art as found in latest top games using Bump/Parallax Mapping making use of this idea... would look awesome!

11. Future Work

There are several things we could work on to make this shader even better. Here are some ideas we think should be investigated:

- Supporting curved surfaces would be an excellent addition... a tangent space version would help on that but the current implementation also needs the original mapping information and in tangent space this is not available.
- Self-shadows is good but also being able to receive shadows from other object is a must if this is to be used in a real game environment. We could think on ways to receive stencil or shadow maps and this would all require correct Z values for each fragment (as the fragment position in global space is available, could update the Z-buffer from the shader adding depth displacement to each fragment).
- Depending on view angle in relation to the relief polygon normal direction we could change the number of passes in the linear search (rays too parallel to polygon requires more linear steps as ray segment trough relief object there is longer).

11. Demo Application Options

The included demo application uses Cg and OpenGL. It allows you to load different relief maps and move the camera around to see it from different angles. Several options are available and can be turned on/off with the following hot-keys:

Ctrl+O	Open relief map	Ctrl+B	Selects Bump Mapping mode
Ctrl+I	Open color texture map	Ctrl+R	Selects Relief Mapping single precision
		Ctrl+T	Selects Relief Mapping double precision
Ctrl+F	Texture filtering on/off		
Ctrl+M	Mip-mapping on/off	Ctrl+D	Double depth on/off
		Ctrl+H	Shadows on/off
Ctrl+L	Light animation on/off		
Ctrl+W	Show quad wireframe on/off	F3	Orbit camera mode
+,-	Increase/decrease tile factor	F4	Free camera mode

Use keys 'S' and 'X' to move forward and backward. The arrow keys will rotate around the object while in orbit camera mode and rotate around the camera position while in free camera mode. You can switch from standard Bump Mapping (Ctrl+B) and new Relief Mapping (Ctrl+R) to see the differences at any given view orientation. The Relief Mapping double precision (Ctrl+T) uses a linear search step size two times bigger than in single precision mode.

11. Apendix: Actual Cg Code

```
struct vert2frag
{
     float4 hpos : POSITION;
     float4 color : COLOR0;
     float4 texcoord : TEXCOORD0;
     float4 opos : TEXCOORD1;
};

struct frag2screen
{    float4 color : COLOR; };

// PROJECT A 3D POINT INTO 2D PLANAR MAPPING TEXTURE SPACE
float2 project_uv(in float3 p,in float4 u,in float4 v)
{
     return float2(dot(p,u.xyz)/u.w,dot(p,v.xyz)/v.w);
}

// 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=6;
#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==1.0)    // if no depth found yet
          if (depth>=t.w)
               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)
          {
               best_depth=depth;
               depth-=2*size;
          }
          depth+=size;
     }
     return best_depth;
}
```

```
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;
}
```