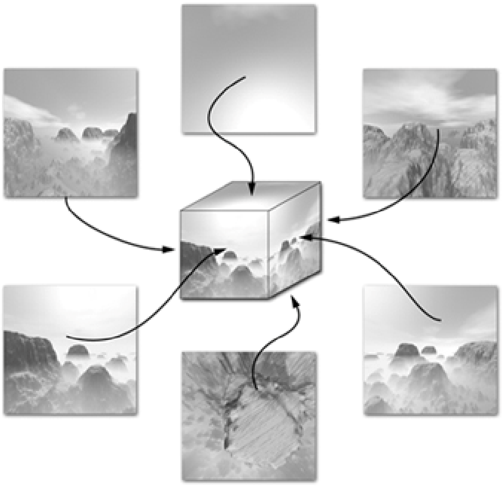
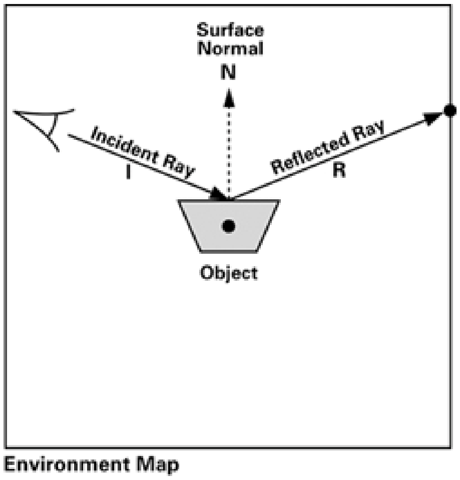
Cg Hacking

Environment Mapping

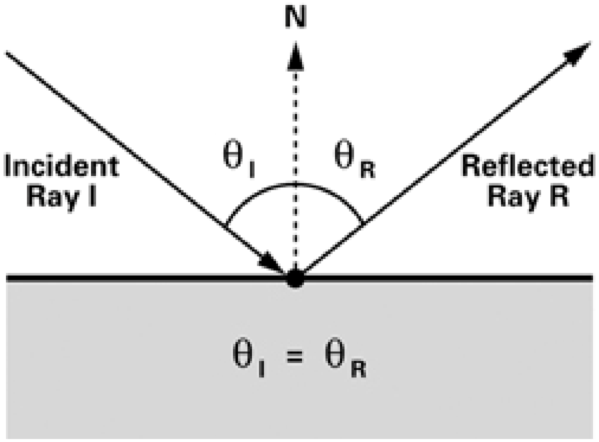


Computing Reflection Vectors



Assumptions?

Computing Reflection Vectors



Computing Reflection Vectors

$$R = I - 2N(N \bullet I)$$

- **reflect(I, N)**
  - Returns the reflected vector for the incident ray **I** and the surface normal **N** . The vector **N** should be normalized. The reflected vector's length is equal to the length of **I** . This function is valid only for three-component vectors.

Though you are better off using the Cg Standard Library routine because of its efficiency, the straightforward implementation of reflect is as follows:

```
float3 reflect (float3  I, float3 N)

{

  return I - 2.0 * N * dot(N, I);

}
```

Cg Environment Mapping

Vertex Program
- transforming the position into clip space
- passing through the texture coordinate set for the decal texture.
- computes the incident and reflected rays.

Fragment Program
- reflected ray looks up the environment map
- uses it to add a reflection to the fragment's final color.
- blend the reflection with a decal texture.
- A uniform parameter called reflectivity allows the application to control how reflective the material is.

Transforming the Vectors into World Space

Environment maps are typically oriented relative to world space, so you need to calculate the reflection vector in world space (or whatever coordinate system orients the environment map). To do that, you must transform the rest of the vertex data into world space.

- Assume:
  - the modeling transform is affine (rather than projective)
  - uniform in its scaling (rather than nonuniformly scaling x, y, and z)
  - w component of position is 1

```
float3 positionW = mul(modelToWorld, position).xyz;
float3 N = mul((float3x3)modelToWorld, normal);
```

If the modeling transform scales positions nonuniformly, you must multiply normal by the inverse transpose of the modeling matrix (modelToWorldInvTrans), rather than simply by modelToWorld . That is:

```
float3  N = mul ((float3x3)modelToWorldInvTrans, normal);
```

If the modeling transform is projective or the w component of the object-space position is not 1, you must divide positionW by its w component. That is:

```
positionW /= positionW.w;
```

Incident / Reflection Vectors

The incident vector is the vector from the eye to the vertex (whereas the view vector is from the vertex to the eye). With the world-space eye position (eyePositionW) available as a uniform parameter and the world-space vertex position (positionW) available from the previous step, calculating the incident vector is a simple subtraction:

```
float3 I = positionW – eyePositionW;
```

You now have the vectors you need—the position and normal, both in world space—so you can calculate the reflection vector:

```
float3 R = reflect(I, N);
```

Word about normalization

The vertex normal needs to be normalized:

N = normalize(N);

Note: In certain cases, we can skip this normalize function call. If we know that the upper 3x3 portion of the modelToWorld matrix causes no nonuniform scaling and the object-space normal parameter is guaranteed to be already normalized, the normalize call is unnecessary.

Why normalize I or R ?

Normalization is not needed here because the reflected vector is used to query a cube map. The direction of the reflected vector is all that matters when accessing a cube map. Regardless of its length, the reflected ray will intersect the cube map at exactly the same location.

And because the reflect function outputs a reflected vector that has the same length as the incident vector as long as N is normalized, the incident vector's length doesn't matter either in this case.

There is one more reason not to normalize R . The rasterizer interpolates R prior to use by the fragment program in the next example. This interpolation is more accurate if the per-vertex reflection vector is not normalized.

Vertex Program

```
void  C7E1v_reflection(float4  position : POSITION,
                       float2 texCoord : TEXCOORD0,
                       float3 normal : NORMAL,

                       out float4 oPosition : POSITION,
                       out float2 oTexCoord : TEXCOORD0,
                       out float3 R        : TEXCOORD1,

                       uniform float3   eyePositionW,
                       uniform float4x4 modelViewProj,
                       uniform float4x4 modelToWorld)

{

  oPosition = mul(modelViewProj, position);
  oTexCoord = texCoord;


  // Compute position and normal in world space

  float3 positionW = mul(modelToWorld, position).xyz;

  float3 N = mul((float3x3)modelToWorld, normal);

  N = normalize(N);


  // Compute the incident and reflected vectors

  float3 I = positionW – eyePositionW;

  R = reflect(I, N);

}
```

Fragment Program

```
void  C7E2f_reflection(float2 texCoord : TEXCOORD0,
                        float3 R       : TEXCOORD1,

                           out float4 color : COLOR,

                           uniform float reflectivity,
                           uniform sampler2D decalMap,
                           uniform samplerCUBE environmentMap)

{

  // Fetch reflected environment color

  float4 reflectedColor = texCUBE(environmentMap, R);

  // Fetch the decal base color

  float4 decalColor = tex2D(decalMap, texCoord);


  color = lerp(decalColor, reflectedColor, reflectivity);

}
```

Vertex vs. Fragment Program

You could achieve higher image quality by using the fragment program (instead of the vertex program) to calculate the reflected vector. Why is this? It is for the same reason that per-fragment lighting looks better than per-vertex lighting.

As with specular lighting, the reflection vector for environment mapping varies in a nonlinear way from fragment to fragment. This means that linearly interpolated per-vertex values will be insufficient to capture accurately the variation in the reflection vector. In particular, subtle per-vertex artifacts tend to appear near the silhouettes of objects, where the reflection vector changes rapidly within each triangle. To obtain more accurate reflections, move the reflection vector calculation to the fragment program. This way, you explicitly calculate the reflection vector for each fragment instead of interpolating it.

Despite this additional accuracy, per-fragment environment mapping may not improve image quality enough to justify the additional expense. As explained earlier in the chapter, most people are unlikely to notice or appreciate the more correct reflections at glancing angles. Keep in mind that environment mapping does not generate physically correct reflections to begin with.

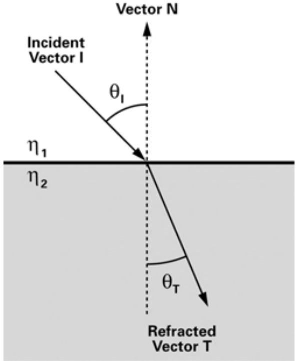
Demo

Refractive Environment Mapping

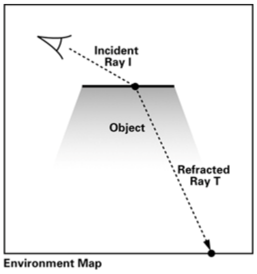
Trivial now!

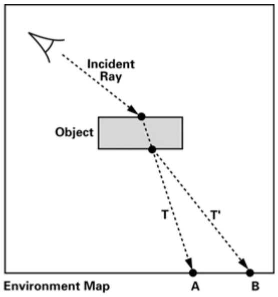
Snell's Law describes what happens to light at a boundary (or interface, as such boundaries are called in the context of refraction) between two media. The refracted vector is represented by T, which stands for "transmitted."

$$\eta_1 \sin\theta_I = \eta_2 \sin\theta_T$$



Refractive Environment Mapping





Snell's Law yourself, because Cg has a refract function that will do it for you.
Here is the function definition:

refract(I, N, etaRatio) Given incident ray direction I , surface normal N , and relative index of refraction etaRatio
The vector N should be normalized.
The refracted vector's length is equal to the length of I .
etaRatio is the ratio of the index of refraction in the medium containing the incident ray to that of the medium being entered.
This function is valid only for three-component vectors.

Refraction

```
float3 refract (float3  I, float3  N, float etaRatio)

{

  float cosI = dot(-I, N);

  float cosT2 = 1.0f - etaRatio * etaRatio * (1.0f – cosI * cosI);

  float3 T = etaRatio * I + ((etaRatio * cosI - sqrt(abs(cosT2))) * N);

  return T * (float3)(cosT2 > 0);

}
```

Vertex Program

```
void  C7E3v_refraction(float4  position : POSITION,
                       float2  texCoord : TEXCOORD0,
                       float3  normal   : NORMAL,

                       out float4 oPosition : POSITION,
                       out float2 oTexCoord : TEXCOORD0,
                       out float3 T         : TEXCOORD1,

                       uniform float etaRatio,
                       uniform float3 eyePositionW,
                       uniform float4x4 modelViewProj,
                       uniform float4x4 modelToWorld)

{

  oPosition = mul(modelViewProj, position);
  oTexCoord = texCoord;


  // Compute position and normal in world space

  float3 positionW = mul(modelToWorld, position).xyz;

  float3 N = mul((float3x3)modelToWorld, normal);

  N = normalize(N);


  // Compute the incident and refracted vectors

  float3 I = normalize (positionW – eyePositionW);

  T = refract(I, N, etaRatio);

}
```

Fragment Program (same)

```
void  C7E2f_refraction(float2 texCoord : TEXCOORD0,
                       float3 R        : TEXCOORD1,

                            out float4 color : COLOR,

                            uniform float transmittance,
                            uniform sampler2D decalMap,
                            uniform samplerCUBE environmentMap)

{

   // Fetch refracted environment color

   float4 refractedColor = texCUBE(environmentMap, R);

   // Fetch the decal base color

   float4 decalColor = tex2D(decalMap, texCoord);


   color = lerp(decalColor, refractedColor, transmittance);

}
```

Demo

Fresnel Effect

Fresnel Effect: when light reaches an interface between two materials, some light reflects off the surface at the interface, and some refracts through the surface.

Fresnel equations describe how much light is reflected and how much is refracted.

Fish and reflection

The Fresnel equations, which quantify the Fresnel effect, are complicated.

So?

Fresnel Effect

Instead of using the equations themselves, we are going to use the empirical approximation:

$$reflectionCoefficient = \max(0, \min(1, bias + scale \times (1 + I \bullet N)^{power}))$$

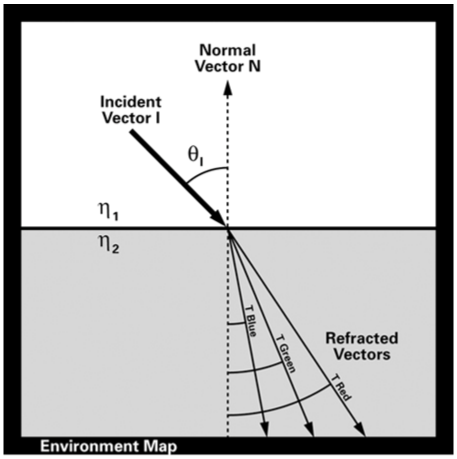
when I and N are nearly coincident, the reflection coefficient should be 0 or nearly 0
[Mostly Refraction]

As I and N diverge, the reflection coefficient should gradually increase and eventually abruptly increase (due to the exponentiation) to 1
[Mostly Reflection]

The range of the reflection coefficient is clamped to the range [0, 1], because we use the reflection coefficient to mix the reflected and refracted contributions according to the following formula (where C stands for color):

CFinal = *reflectionCoefficient* x *CReflected* + (1 -*reflectionCoefficient*) x *CRefracted*

Chromatic Dispersion



Chromatic Dispersion

```
TRed   = refract(I, N, etaRatio.x);

TGreen = refract(I, N, etaRatio.y);

TBlue  = refract(I, N, etaRatio.z);
```

Demo