Half-Life® 2 / Valve Source™ Shading

22 MARCH 2004 / **Gary McTaggart**

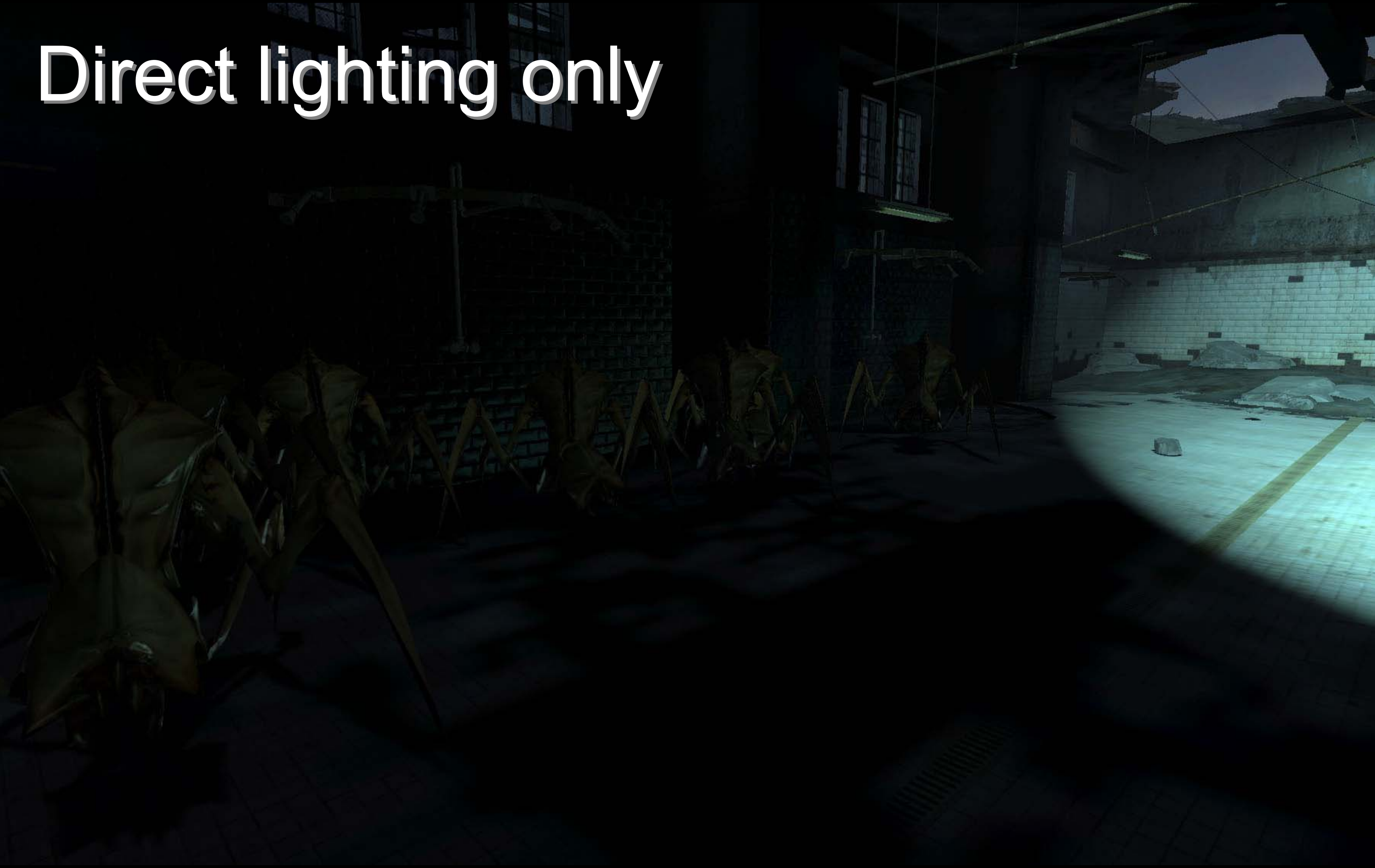
Outline

- "Radiosity Normal Mapping"
- World Geometry pixel shading
  - Lighting equation
  - Managing shader permutations
- Model Geometry vertex shading
- Reflection and Refraction

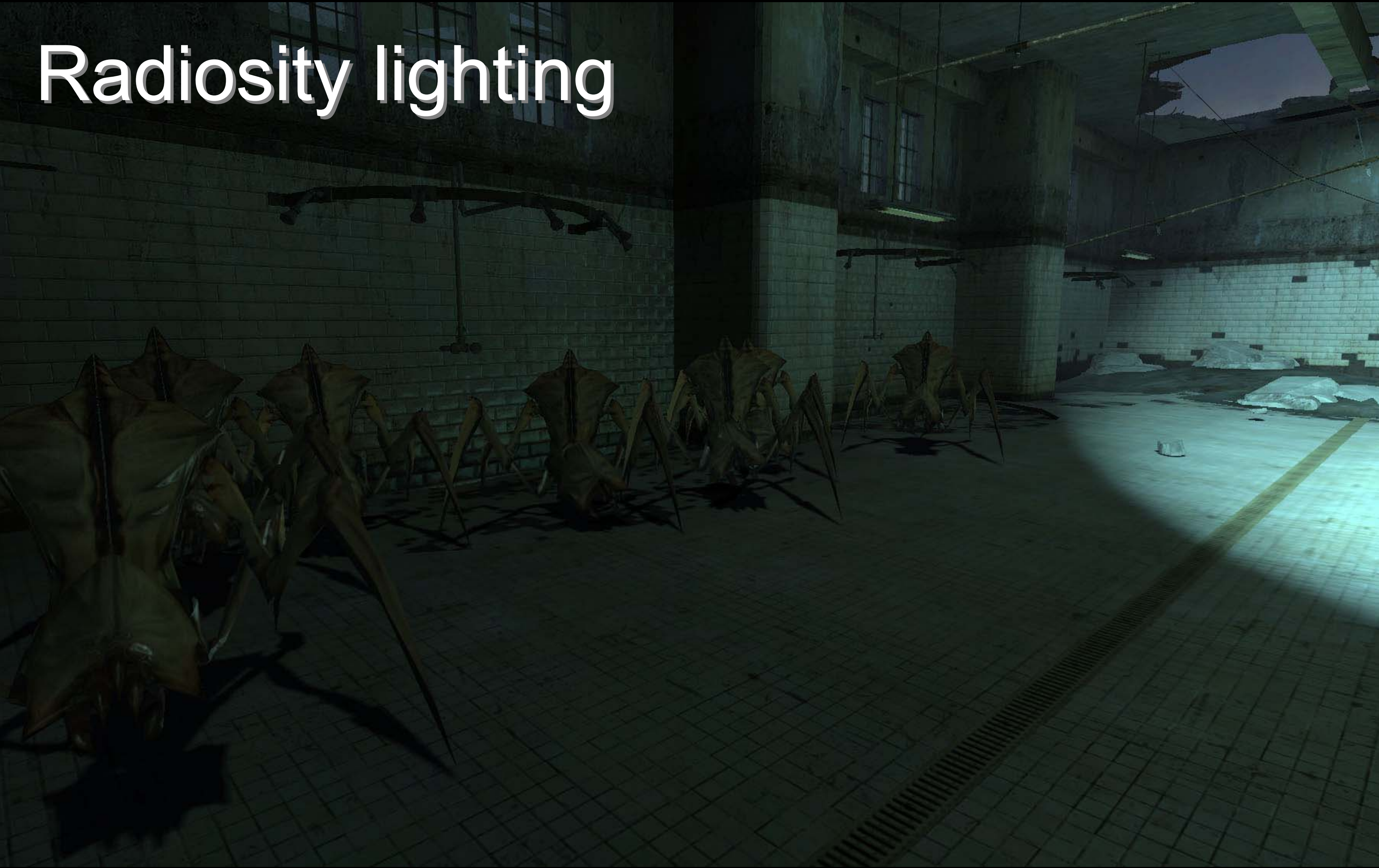
Why Radiosity?

- Realism
- Avoids harsh lighting
- Less micro-management of light sources for content production
- Can’t tune lights shot-by-shot like movies. Don’t know what the shots are, and don’t want to take the production time to do this.

Direct lighting only



Radiosity lighting



Why normal-mapping?

- Reusable high-frequency detail
- Higher detail than we can currently get from triangles
- Works well with both diffuse and specular lighting models
- Can now be made to integrate with radiosity

“Radiosity Normal Mapping”

- We created this technique to address the strengths of both radiosity and normal mapping
- Highly efficient
- This is the key to Half-Life 2®/ Valve Source™ shading.

World vs. Model

- The Valve Source™ engine uses two classes of geometry
  - World/Displacement geometry
    - Large/static geometry
    - Radiosity light maps
  - Model geometry
    - Static props, physics props, and animated characters
    - Ambient cube

Radiosity Normal Mapping

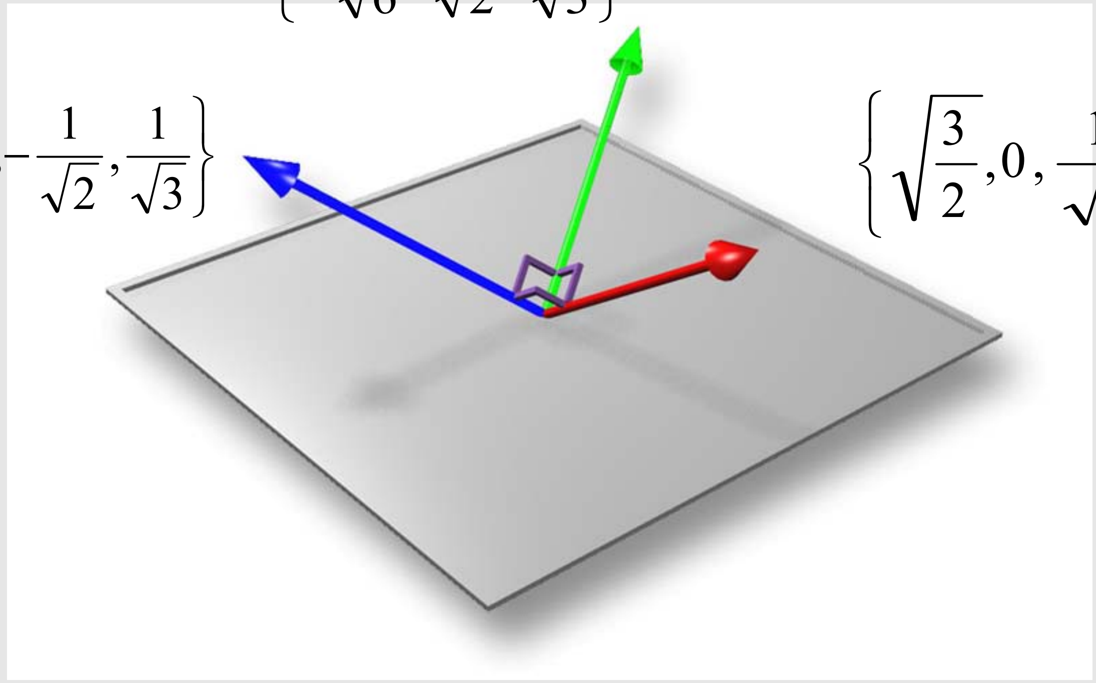
- Normal mapping is typically accumulated one light at a time
  - Multiple diffuse lights handled by summing multiple N•L terms within or between passes
- Radiosity Normal Mapping effectively bump maps with respect to an arbitrary number of lights in one pass

Basis for Radiosity Normal Mapping

$$\left\{-\frac{1}{\sqrt{6}},\frac{1}{\sqrt{2}},\frac{1}{\sqrt{3}}\right\}$$

$$\left\{-\frac{1}{\sqrt{6}},-\frac{1}{\sqrt{2}},\frac{1}{\sqrt{3}}\right\}$$

$$\left\{\sqrt{\frac{3}{2}},0,\frac{1}{\sqrt{3}}\right\}$$



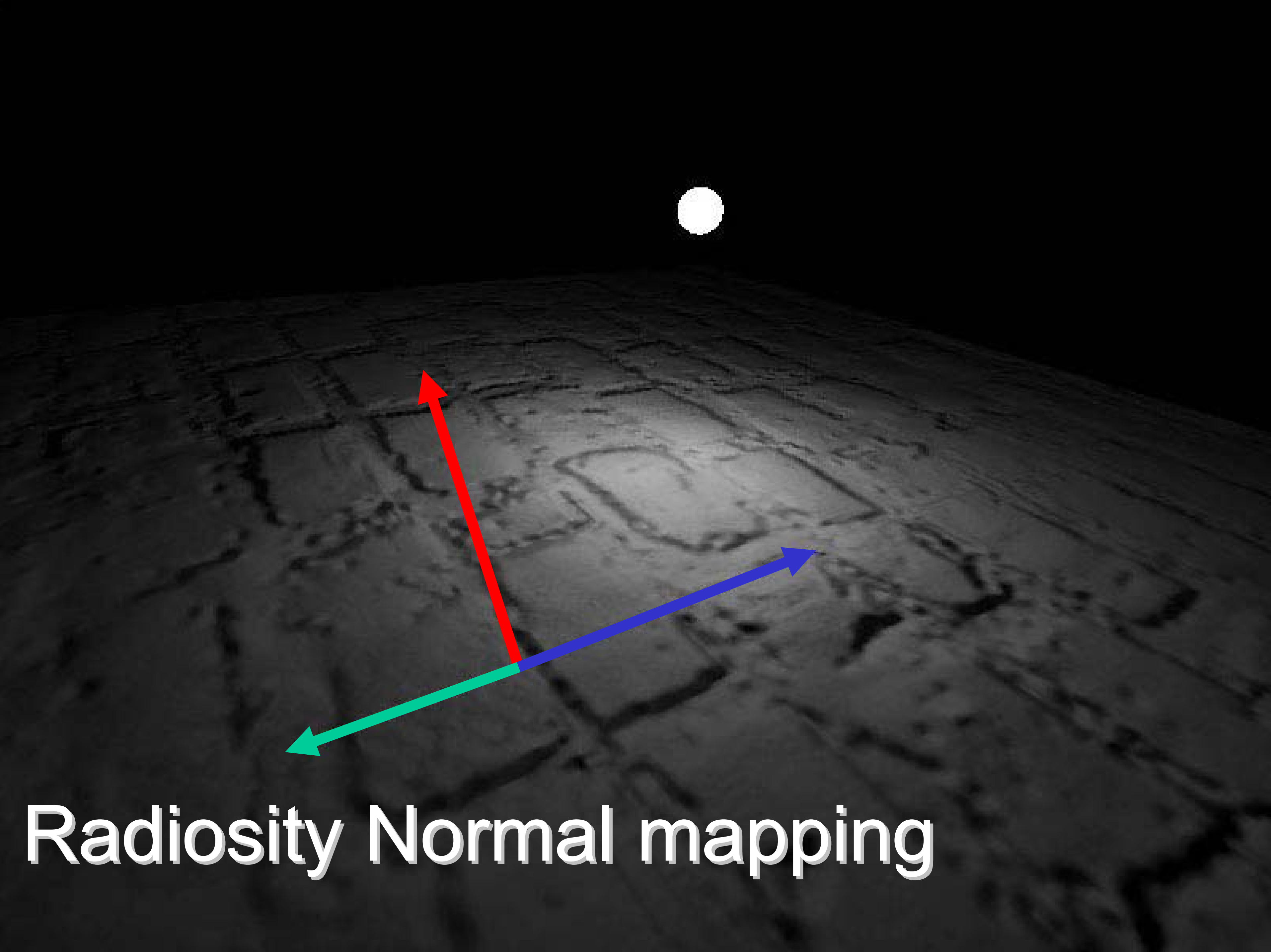
Computing Light map Values

- Traditionally, when computing light map values using a radiosity preprocessor, a single color value is calculated
- In Radiosity Normal Mapping, we transform our basis into tangent space and compute light values for each vector.

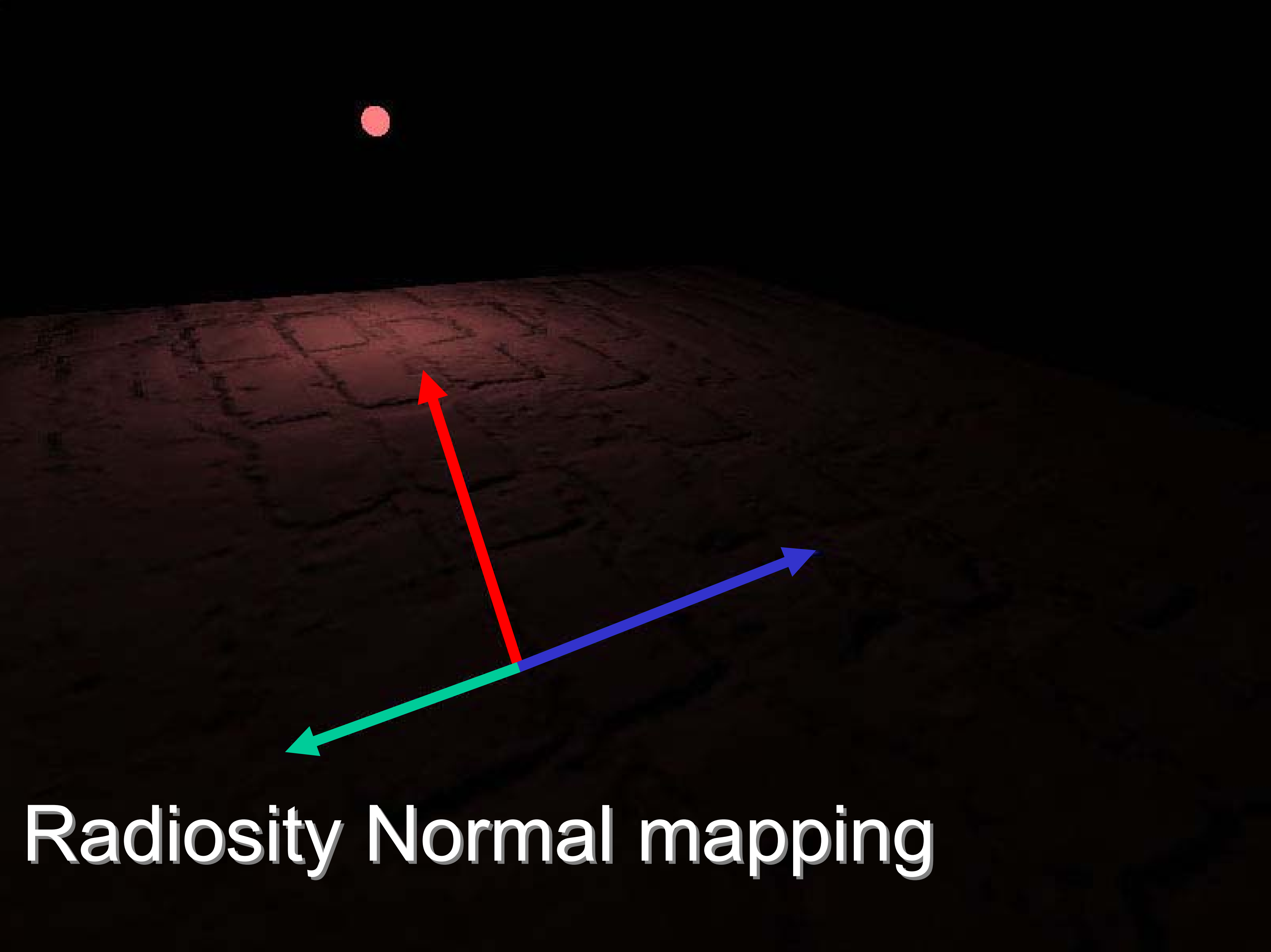
At the pixel level. . .

- Transform the normal from a normal map into our basis
- Sample three light map colors, and blend between them based the transformed vector

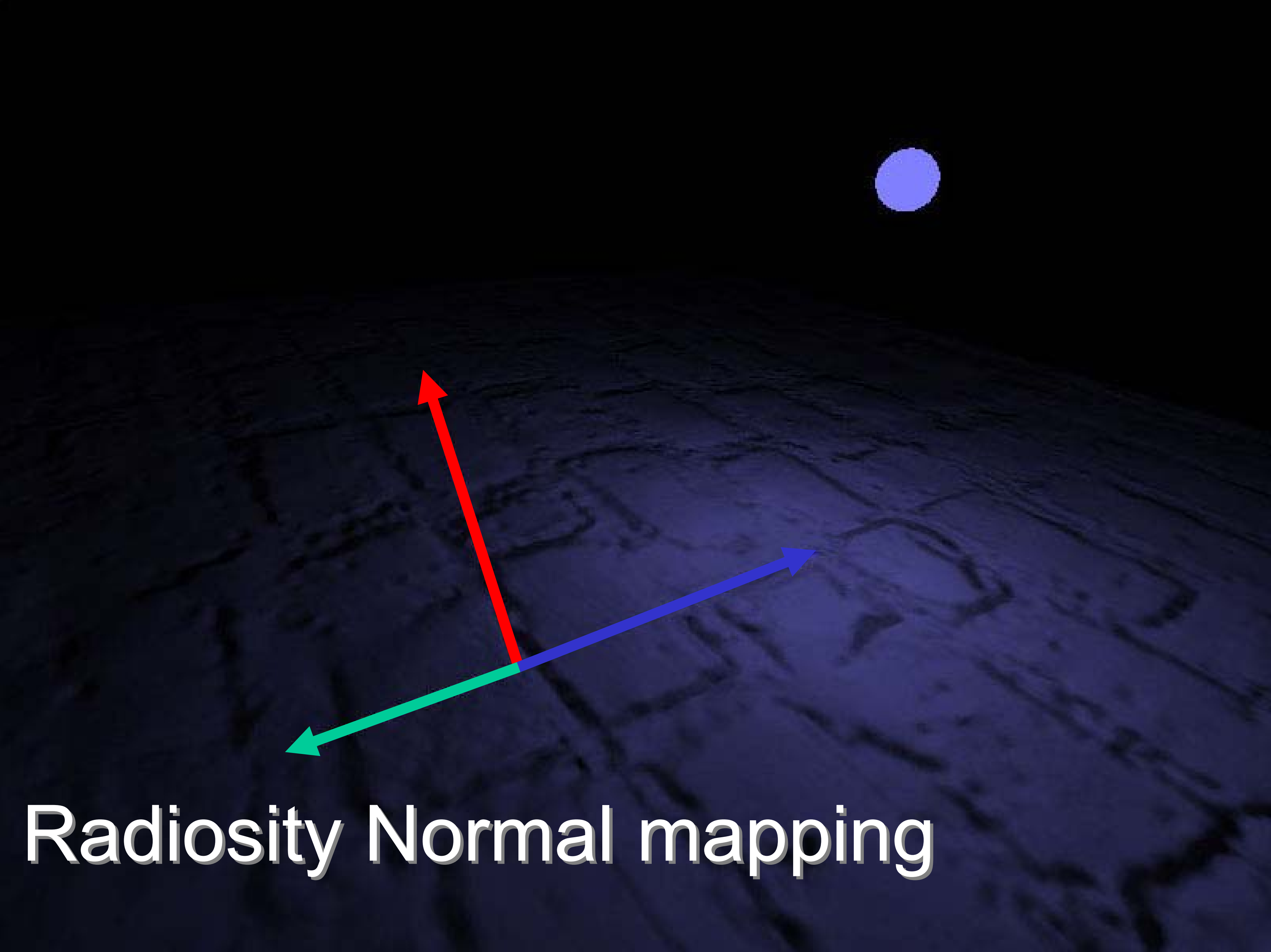
```
lightmapColor[0] * dot( bumpBasis[0], normal )+
lightmapColor[1] * dot( bumpBasis[1], normal )+
lightmapColor[2] * dot( bumpBasis[2], normal )
```



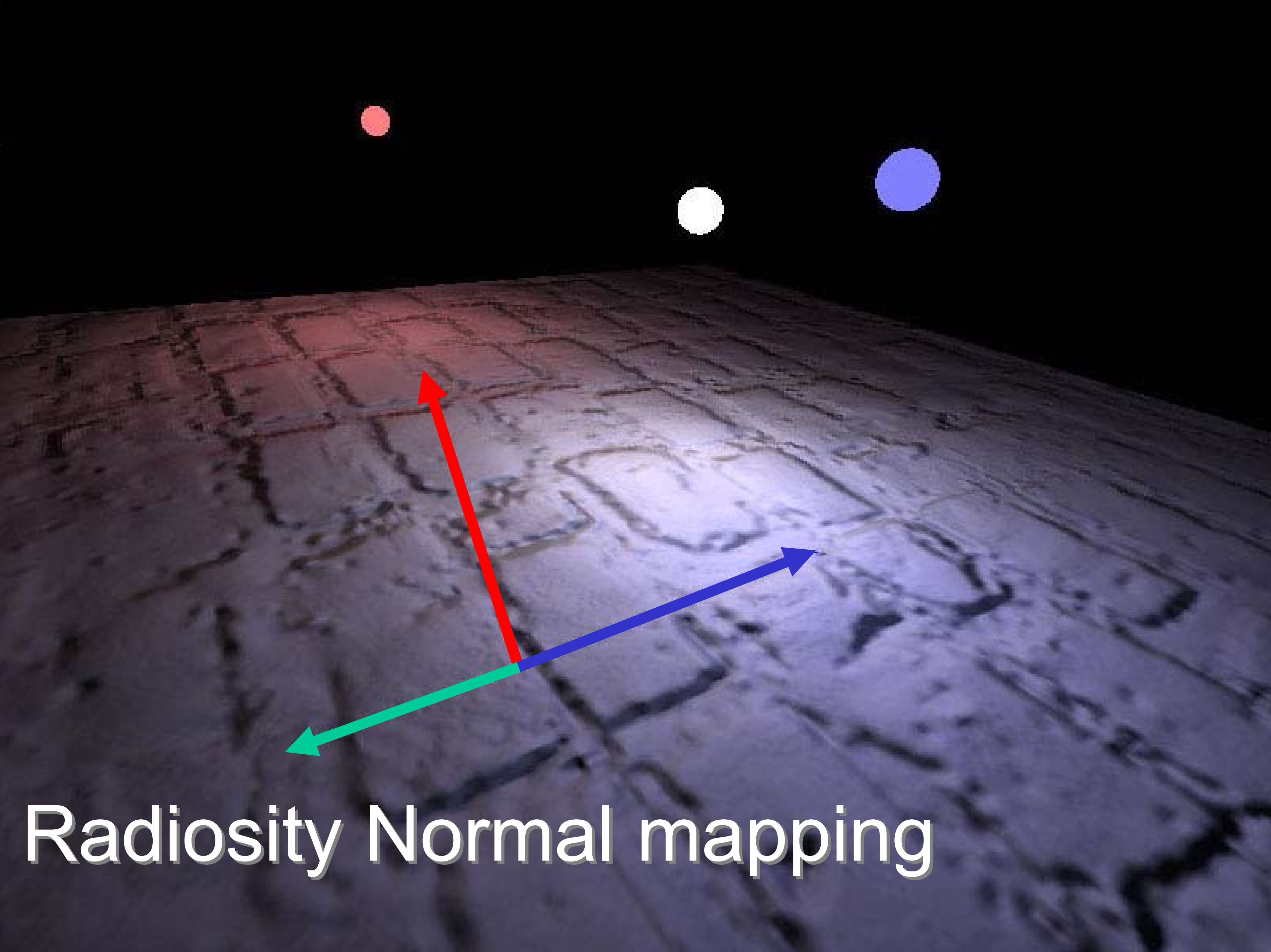
Radiosity Normal mapping



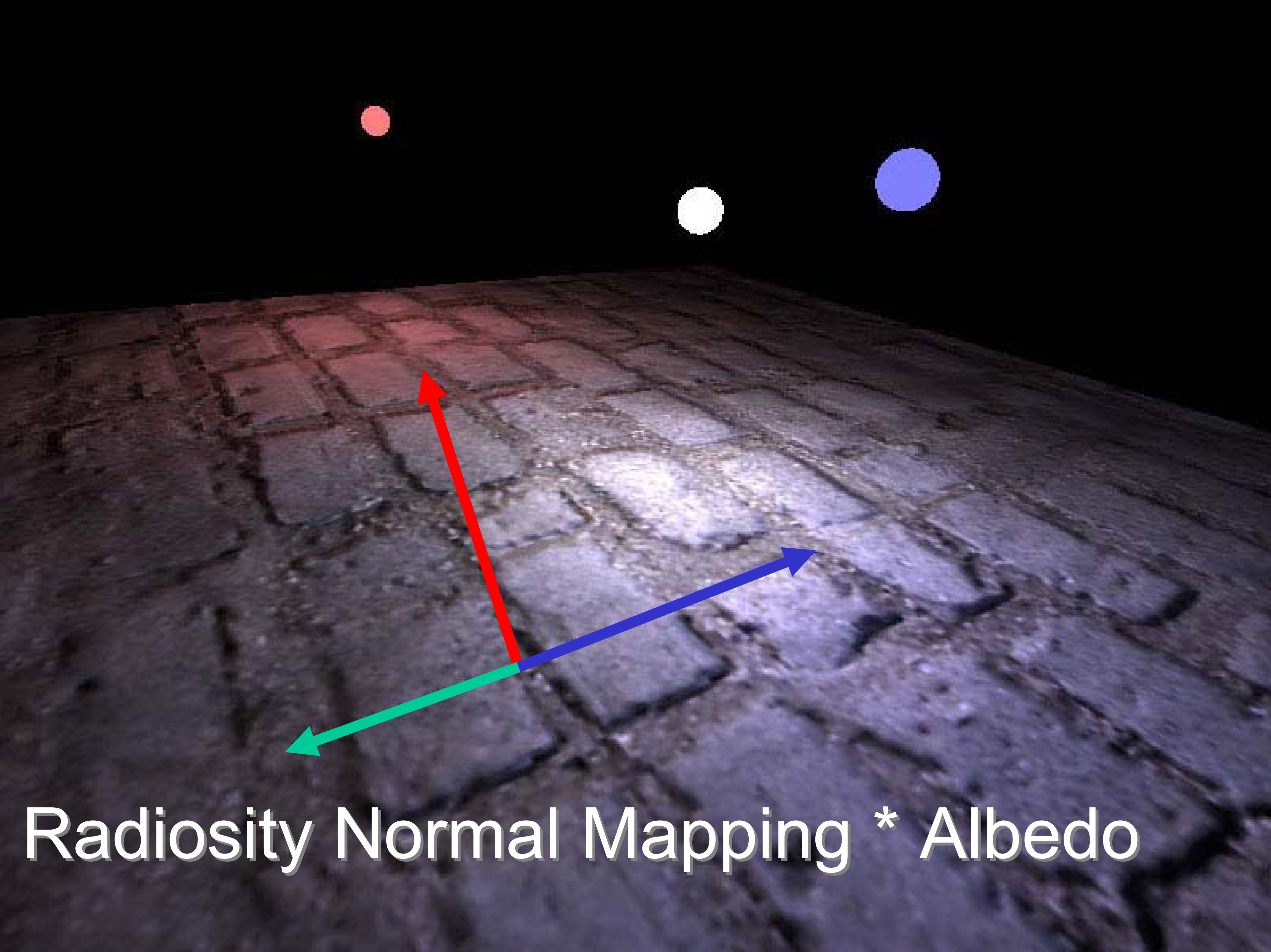
Radiosity Normal mapping



Radiosity Normal mapping



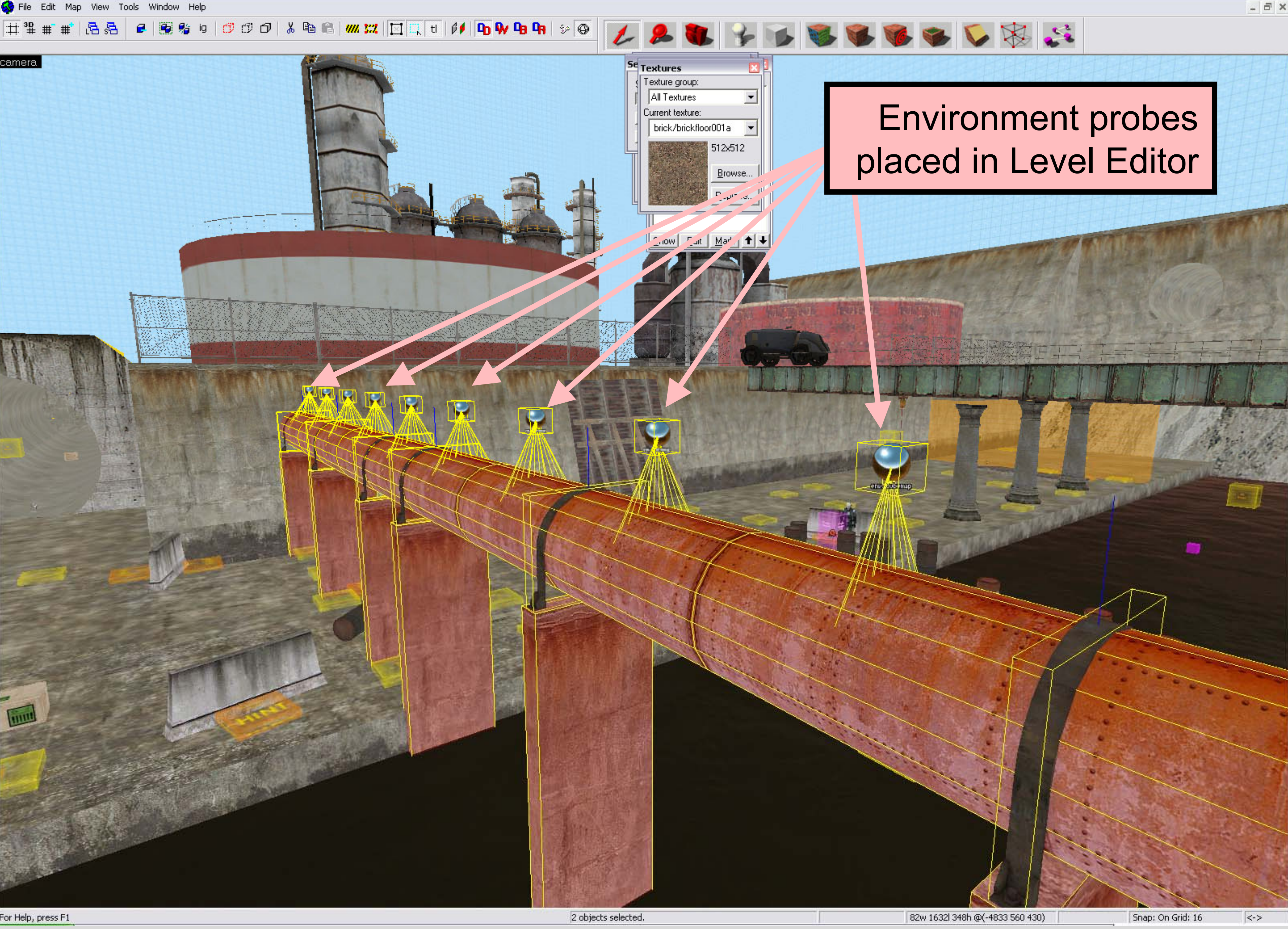
Radiosity Normal mapping



Radiosity Normal Mapping * Albedo

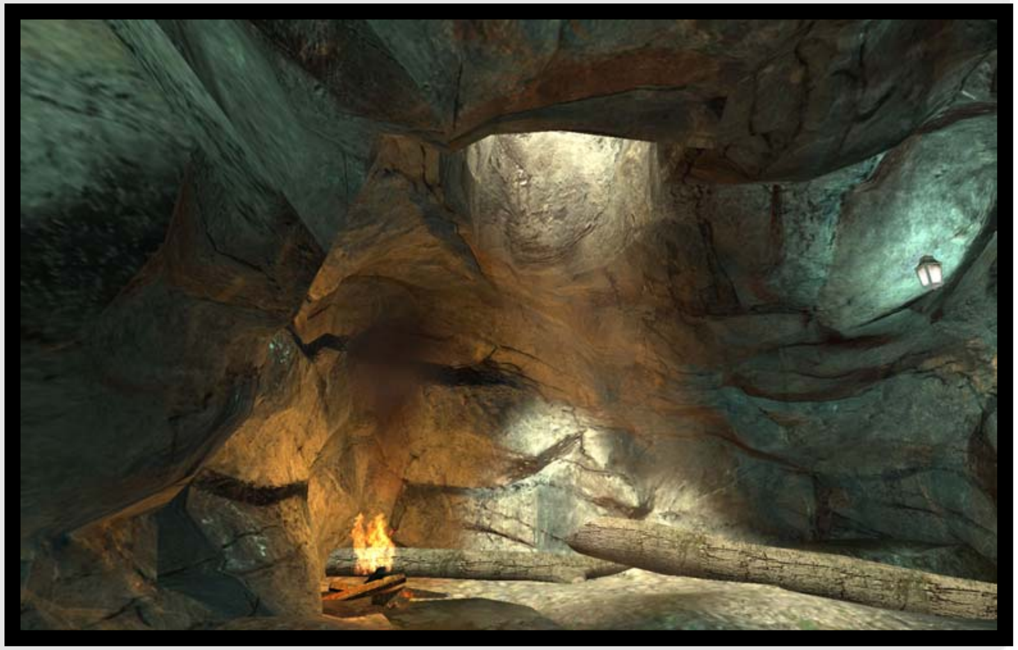
World Specular Lighting

- Use cube maps for specular lighting.
- Designers place point entities in their maps which are the sample points for specular lighting.
- Cube maps are pre-computed in engine from the level data using rendering.
- World surfaces pick up the “best” cube map, or cube maps can be manually assigned to surfaces to fix boundary problems.

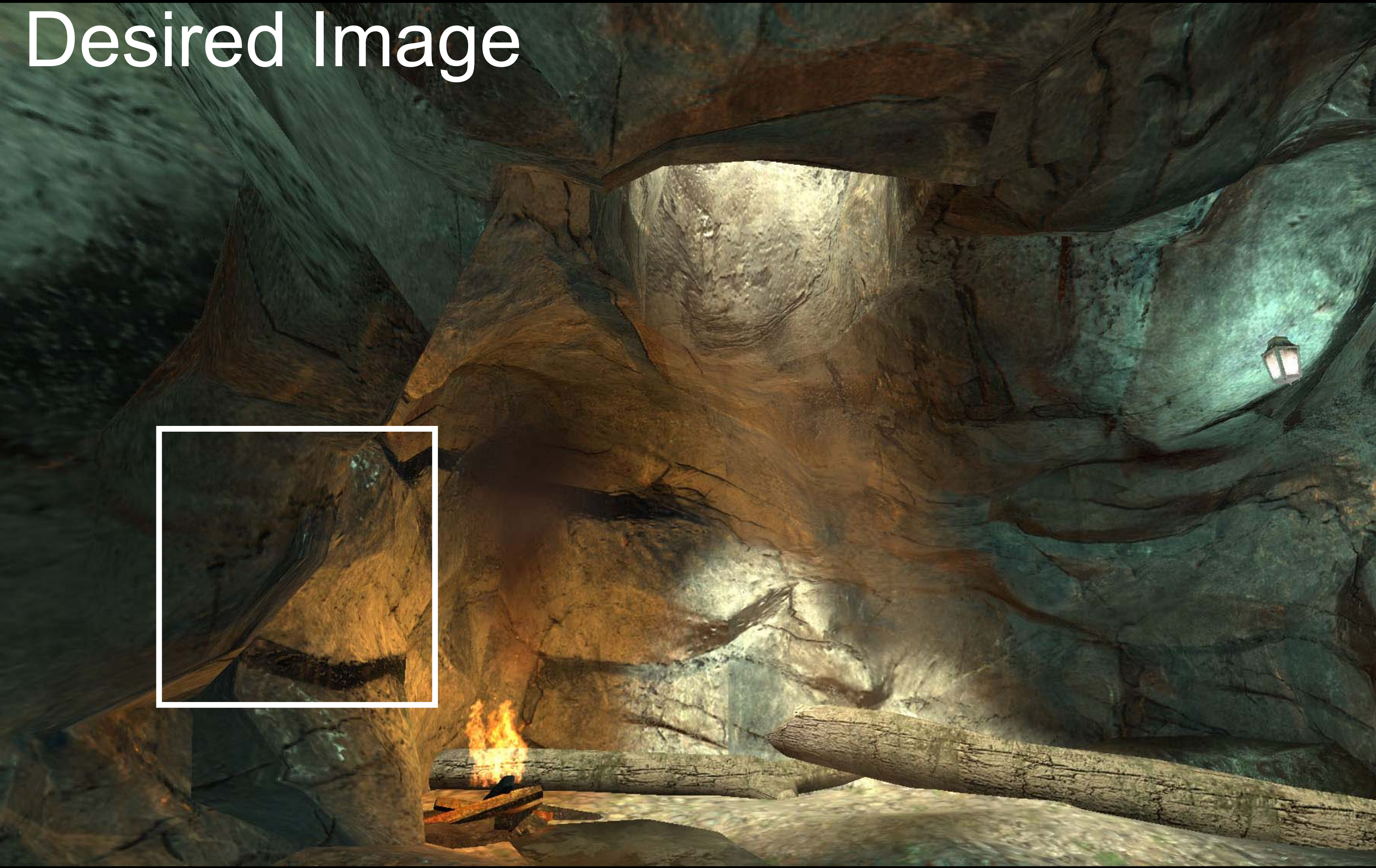

File
Edit
Map
View
Tools
Window
Help
camera
Textures
Texture group:
All Textures
Current texture:
brick/brickfloor001a
512x512
Browse...
Environment probes placed in Level Editor
HINT
For Help, press F1
2 objects selected.
82w 1632l 348h @(-4833 560 430)
Snap: On Grid: 16

World Lighting Equation

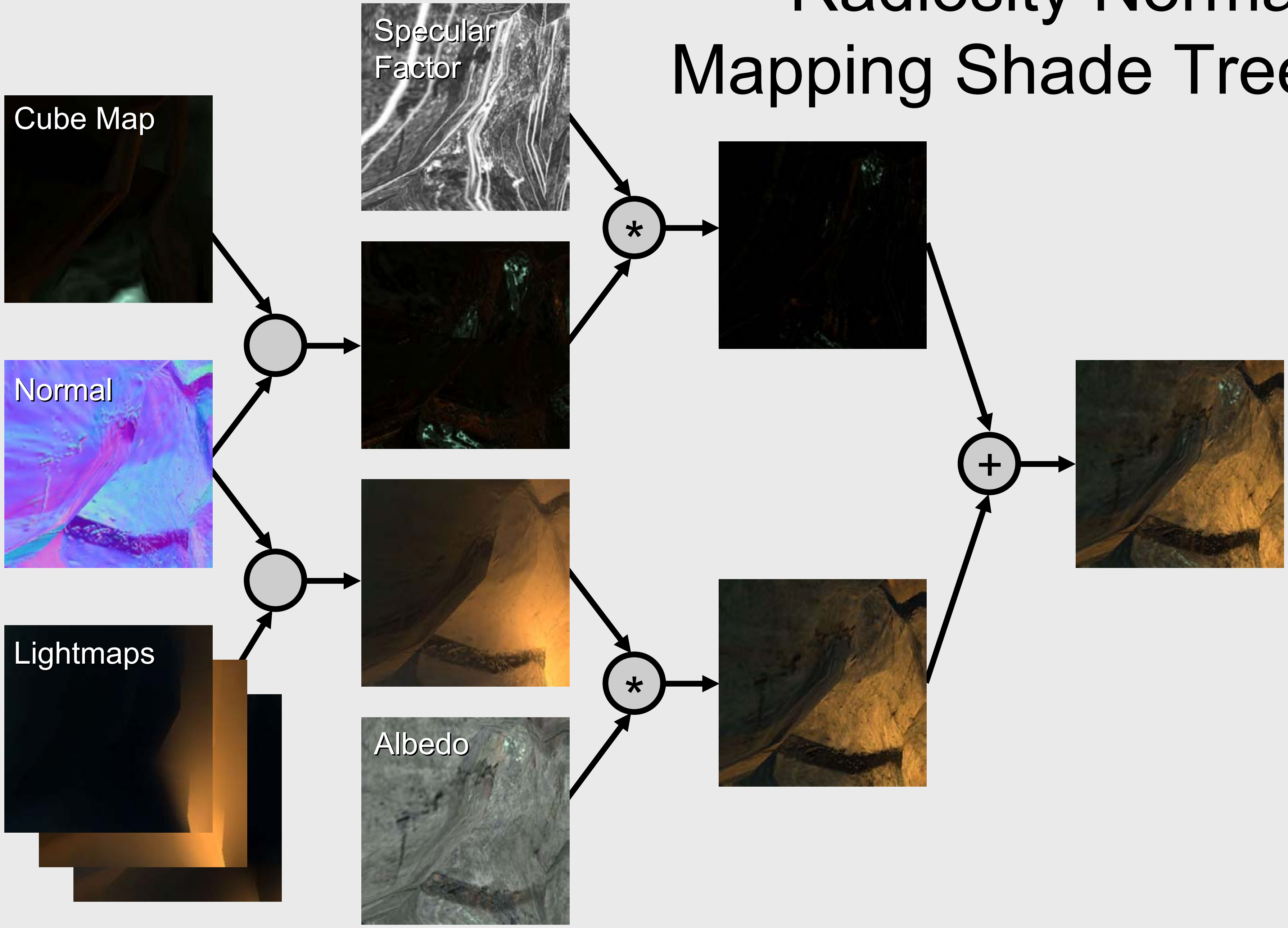
- We will now illustrate Half-Life® 2 world lighting one term at a time
- Our desired image:



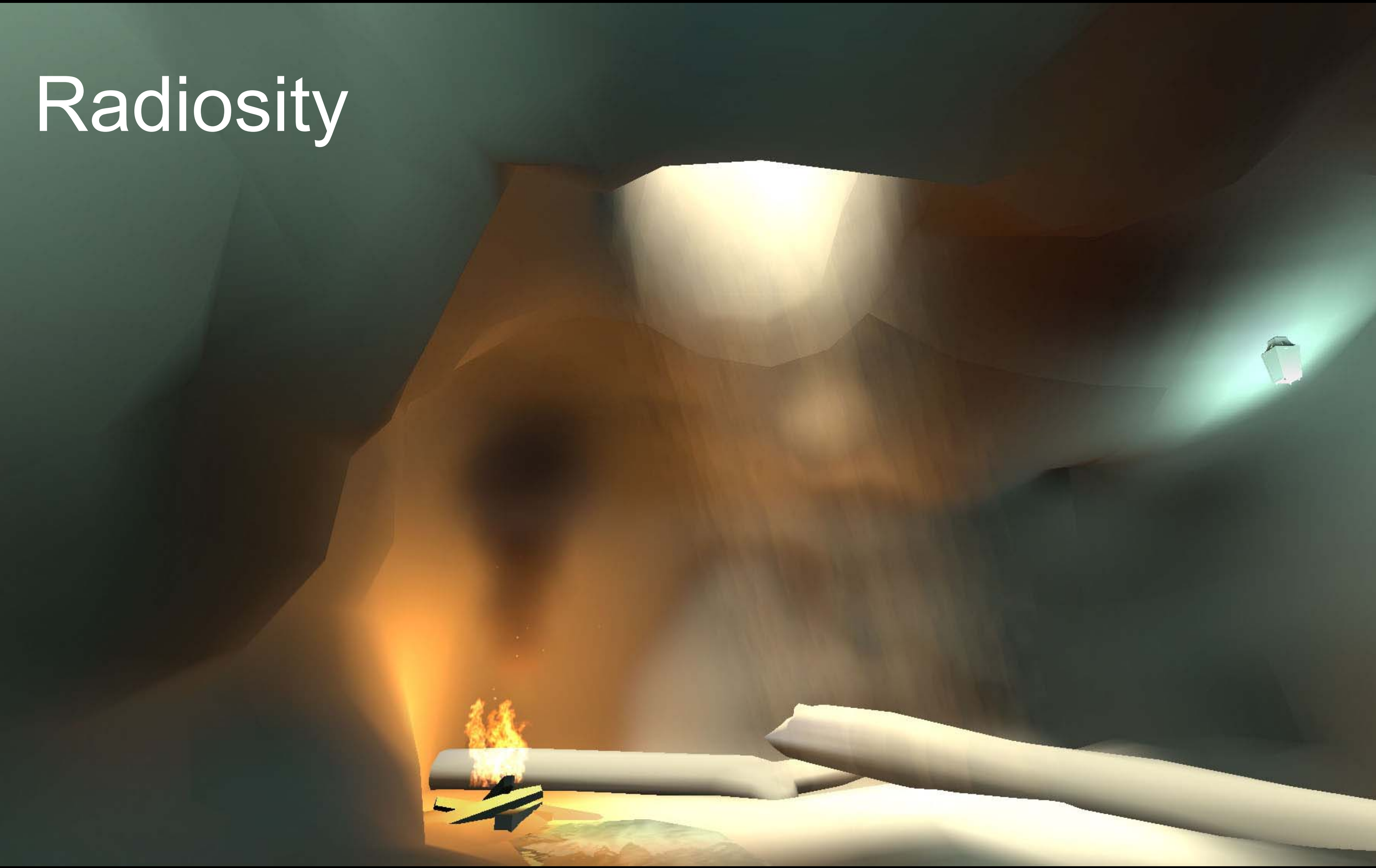
Desired Image

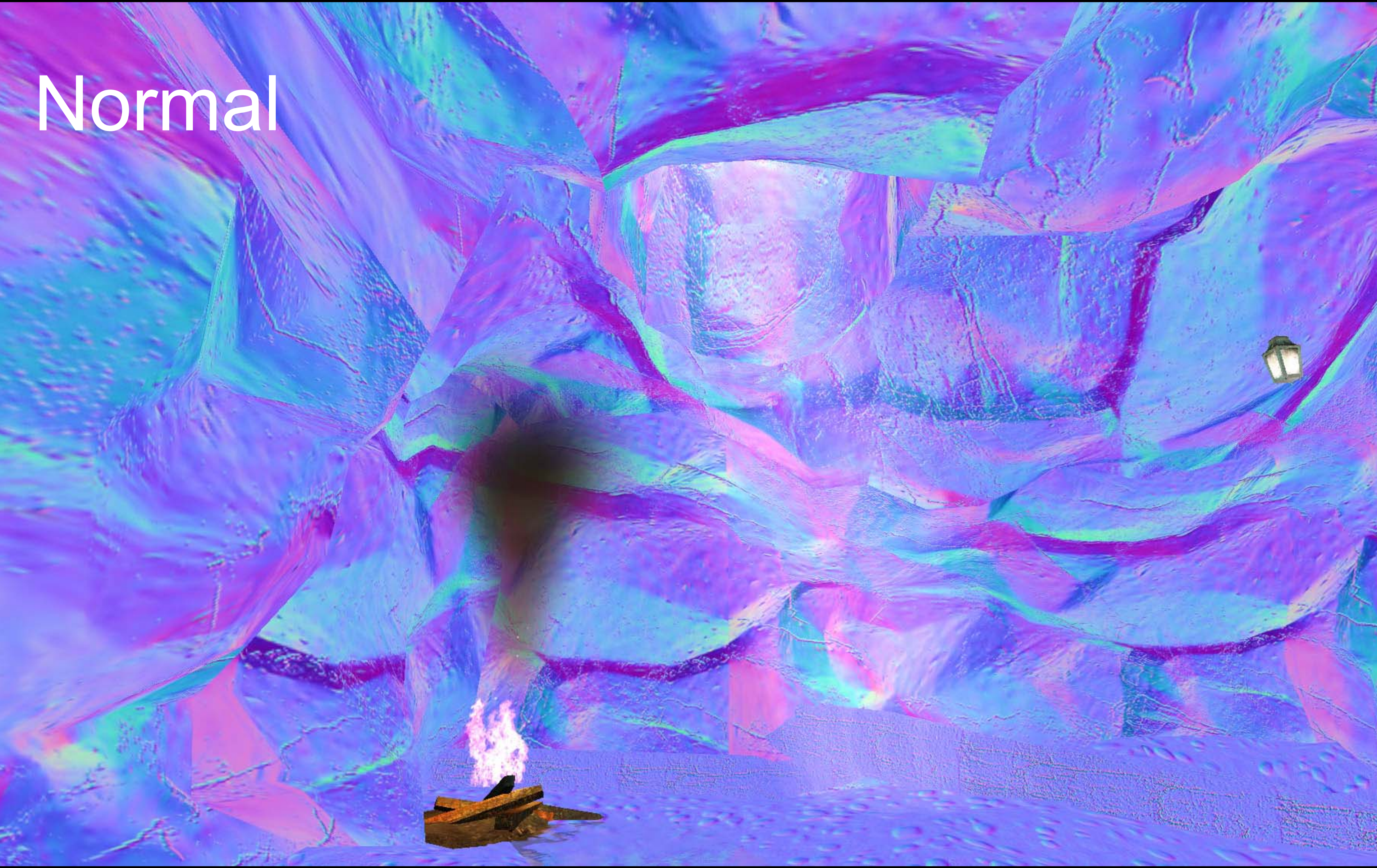


Radiosity Normal Mapping Shade Tree

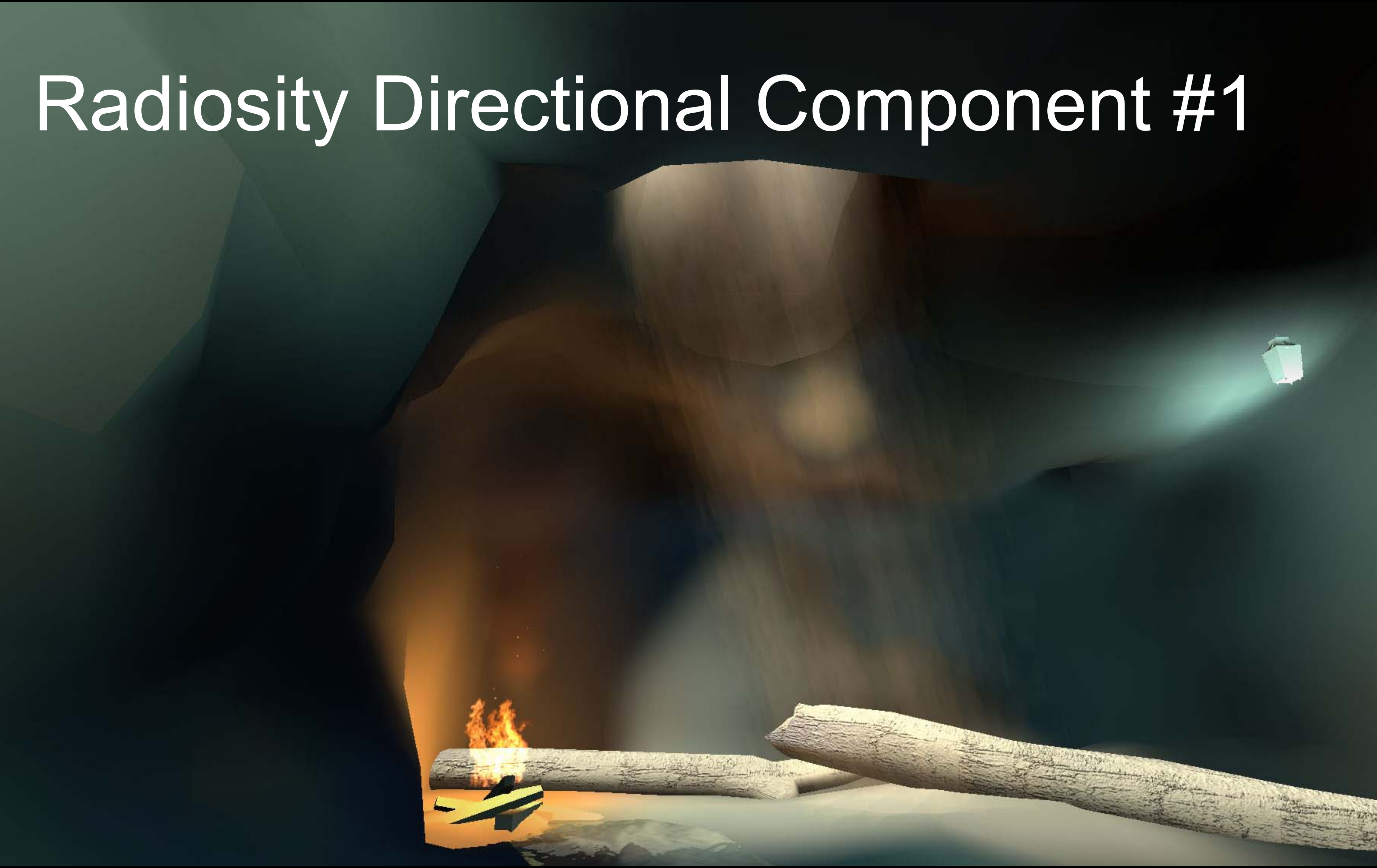


Radiosity

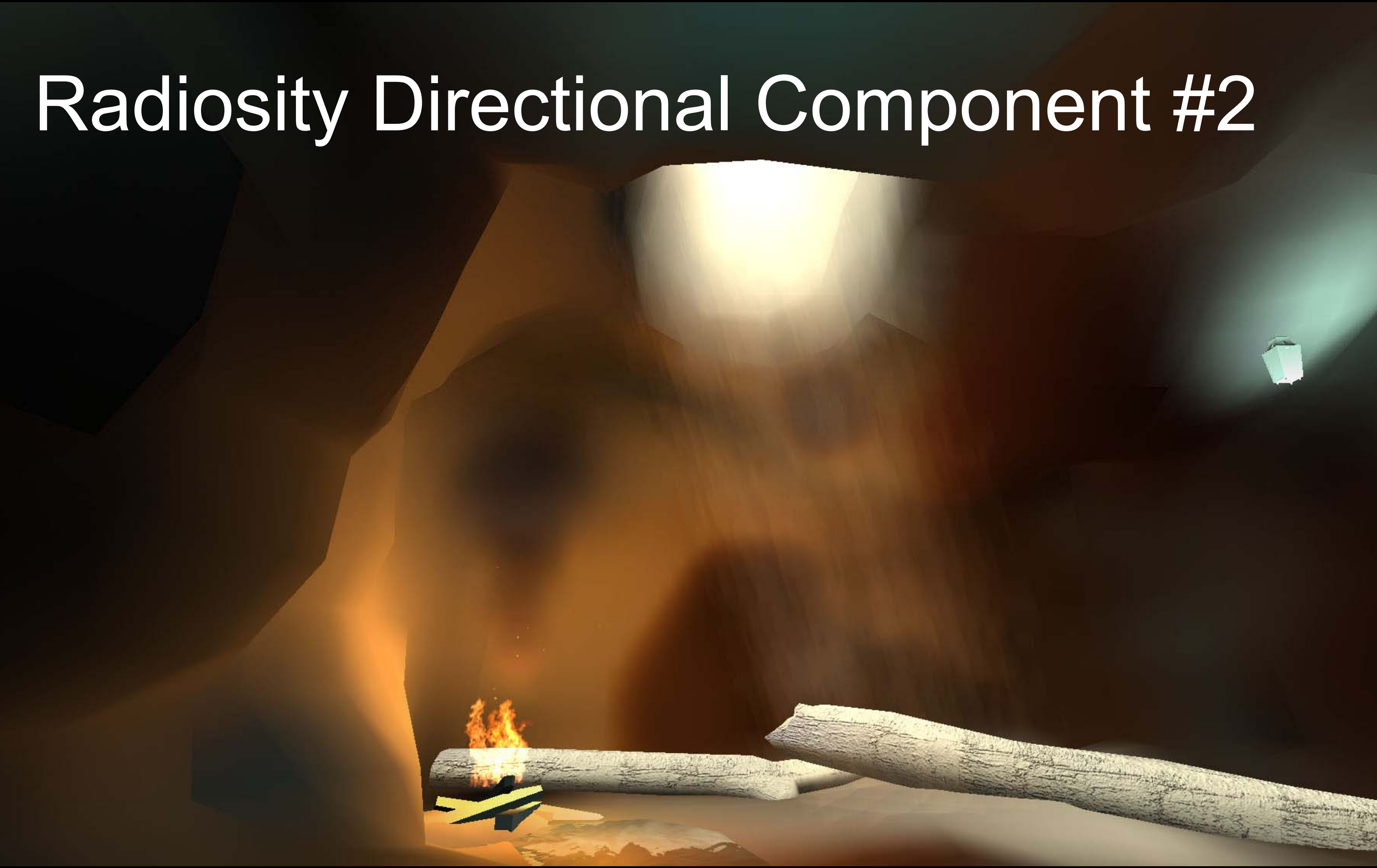



Normal

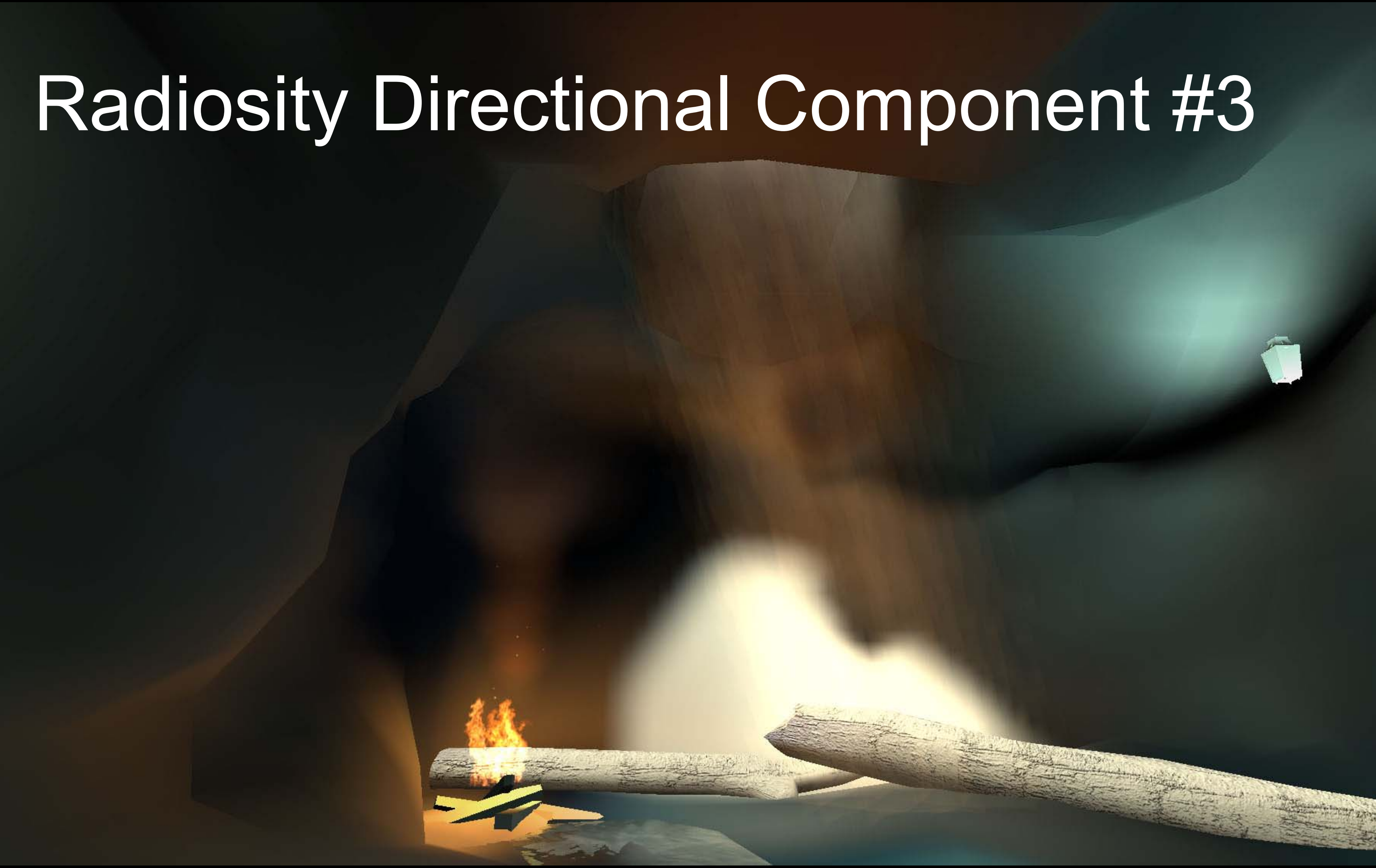
Radiosity Directional Component #1



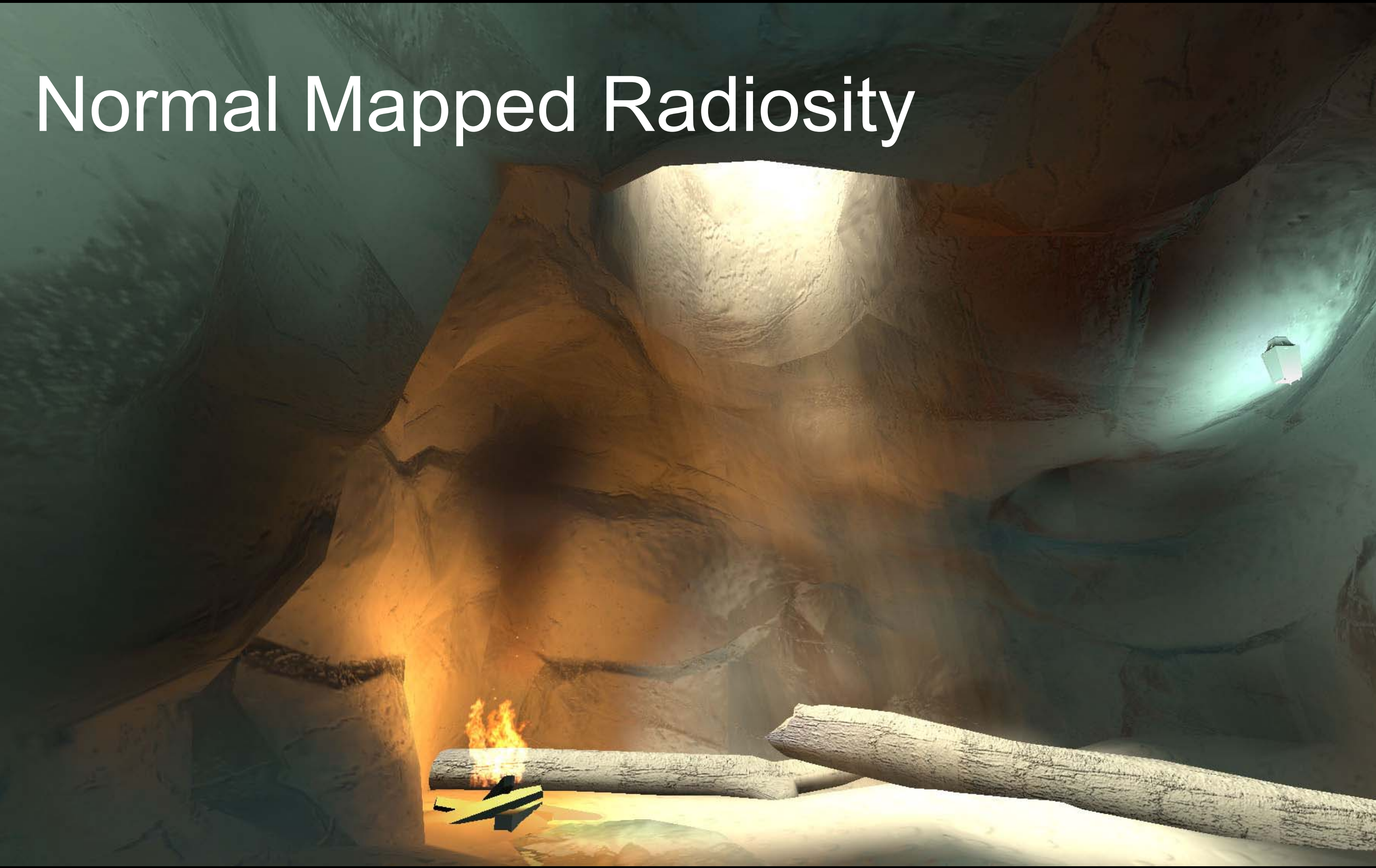
Radiosity Directional Component #2



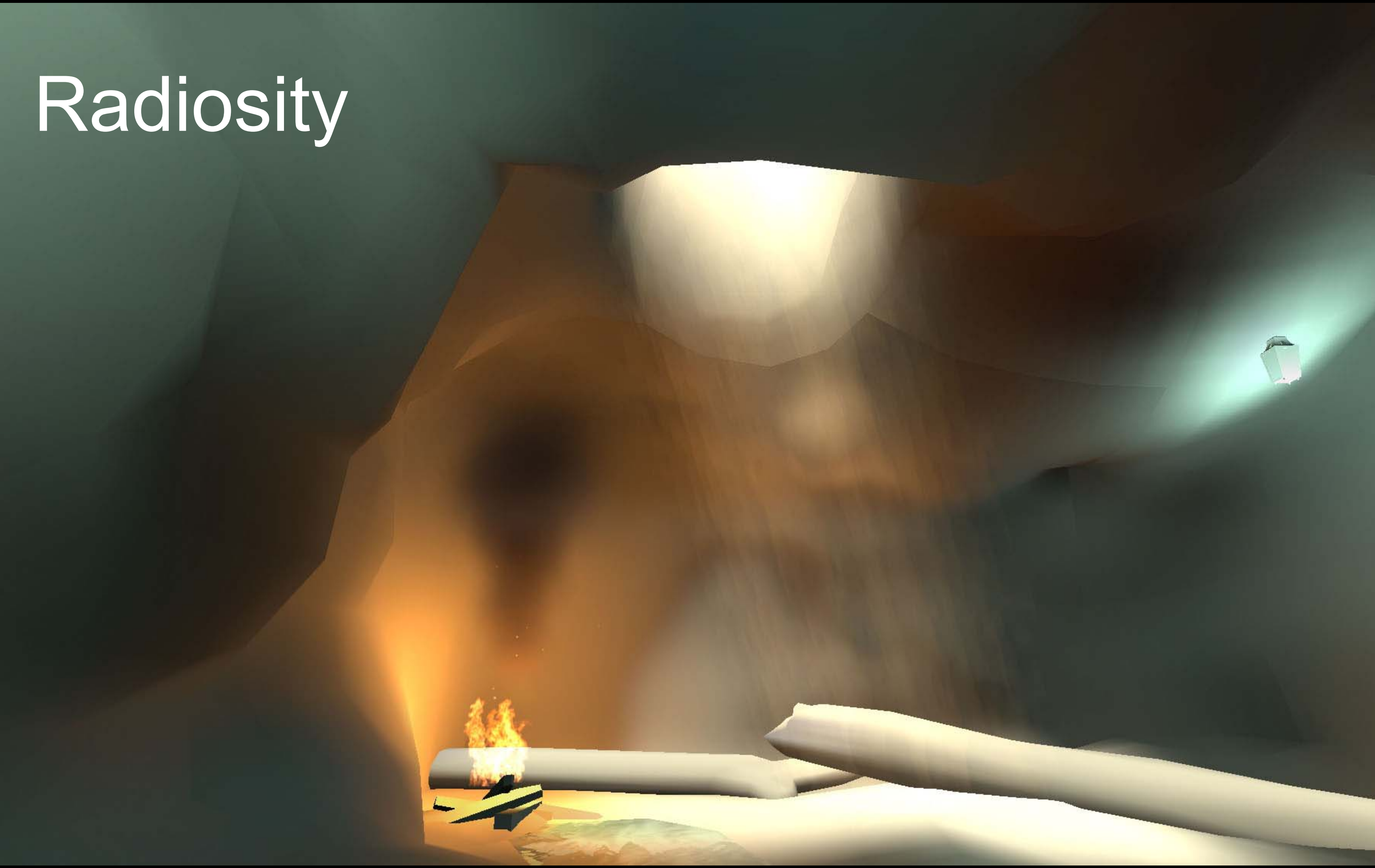
Radiosity Directional Component #3

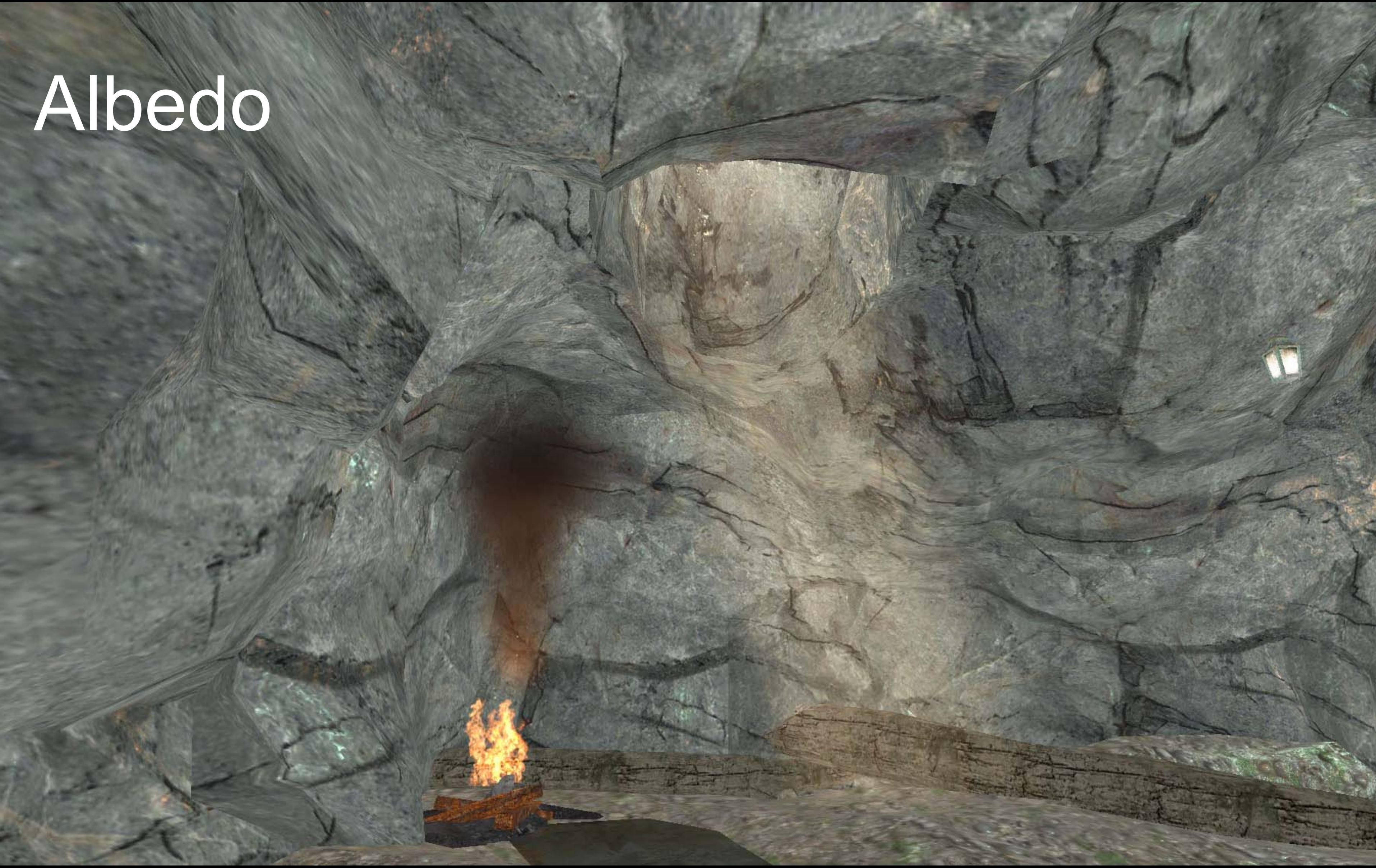


Normal Mapped Radiosity

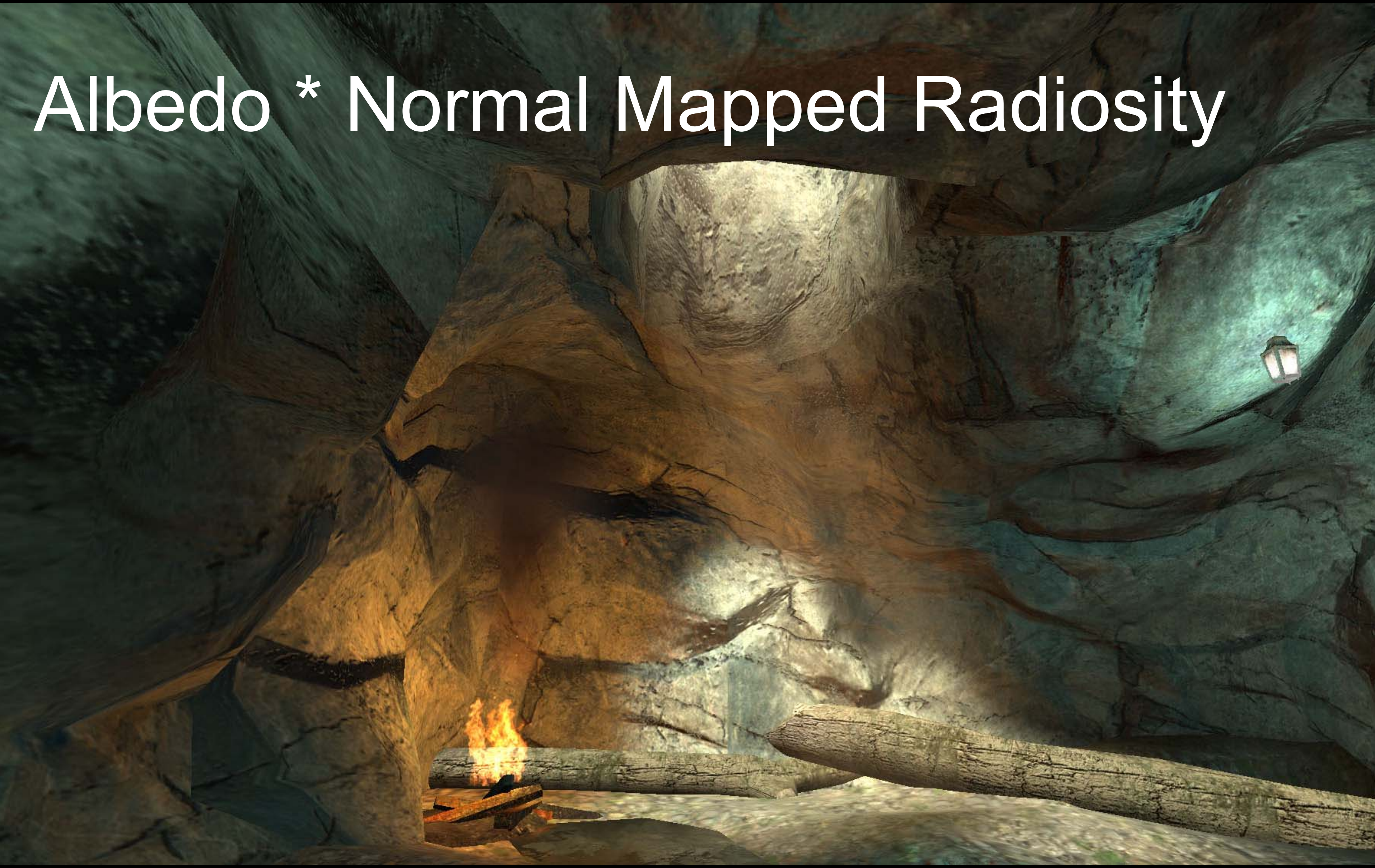


Radiosity

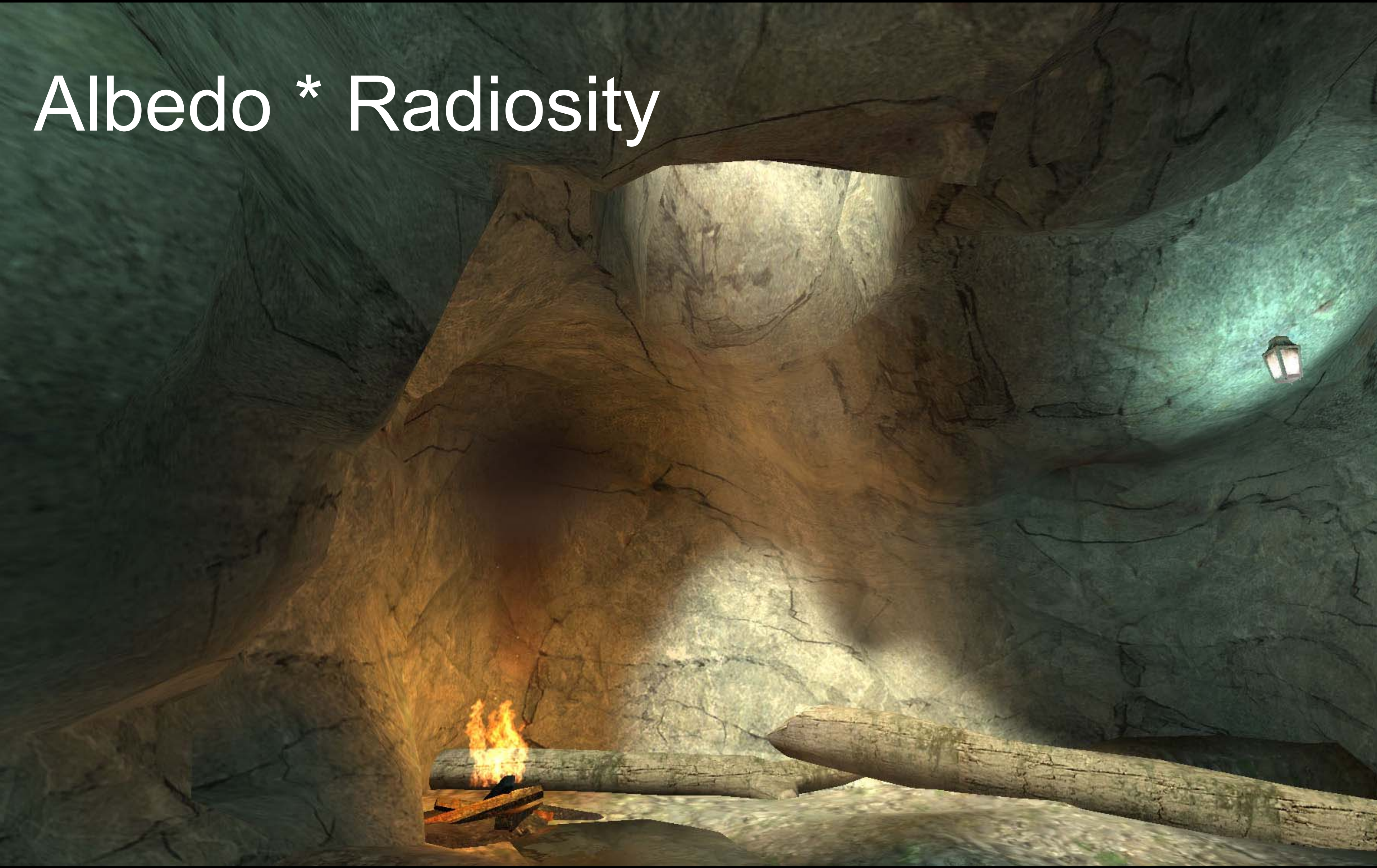



Albedo

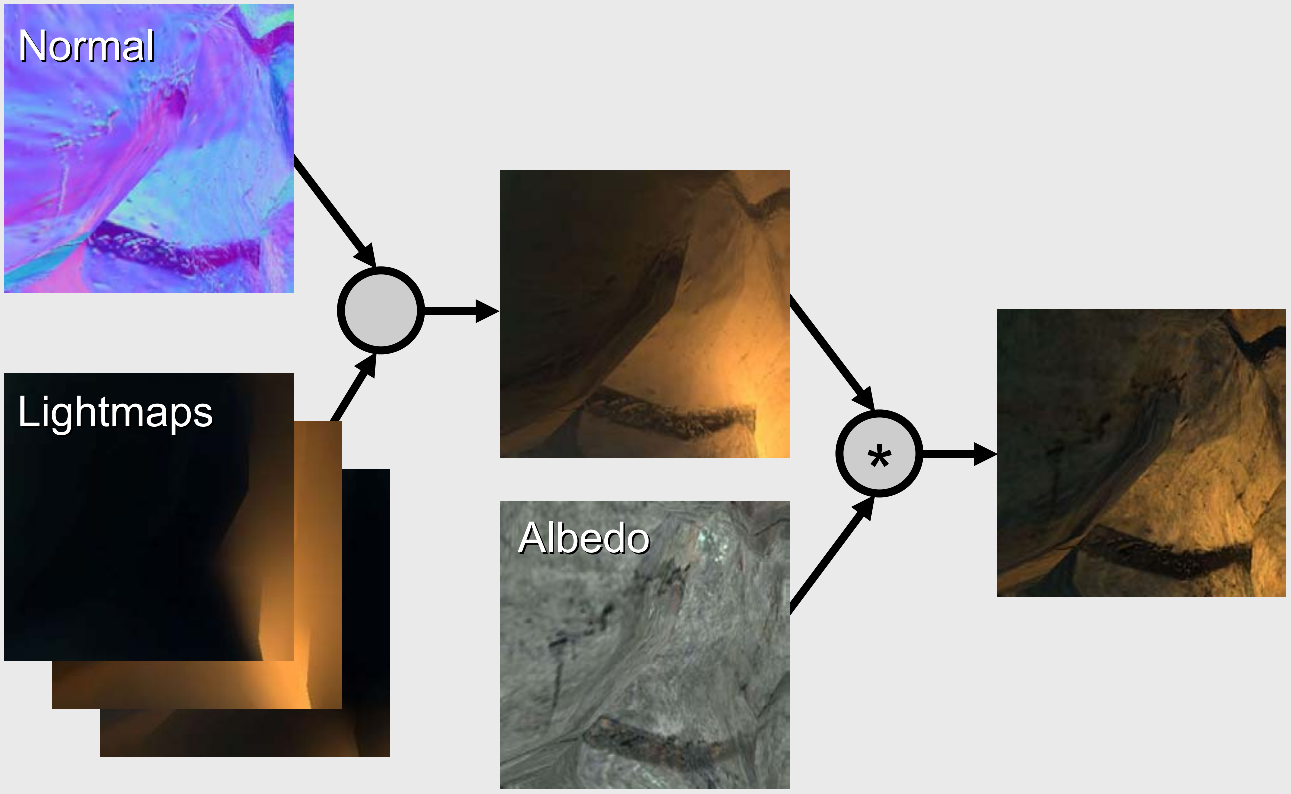
Albedo * Normal Mapped Radiosity

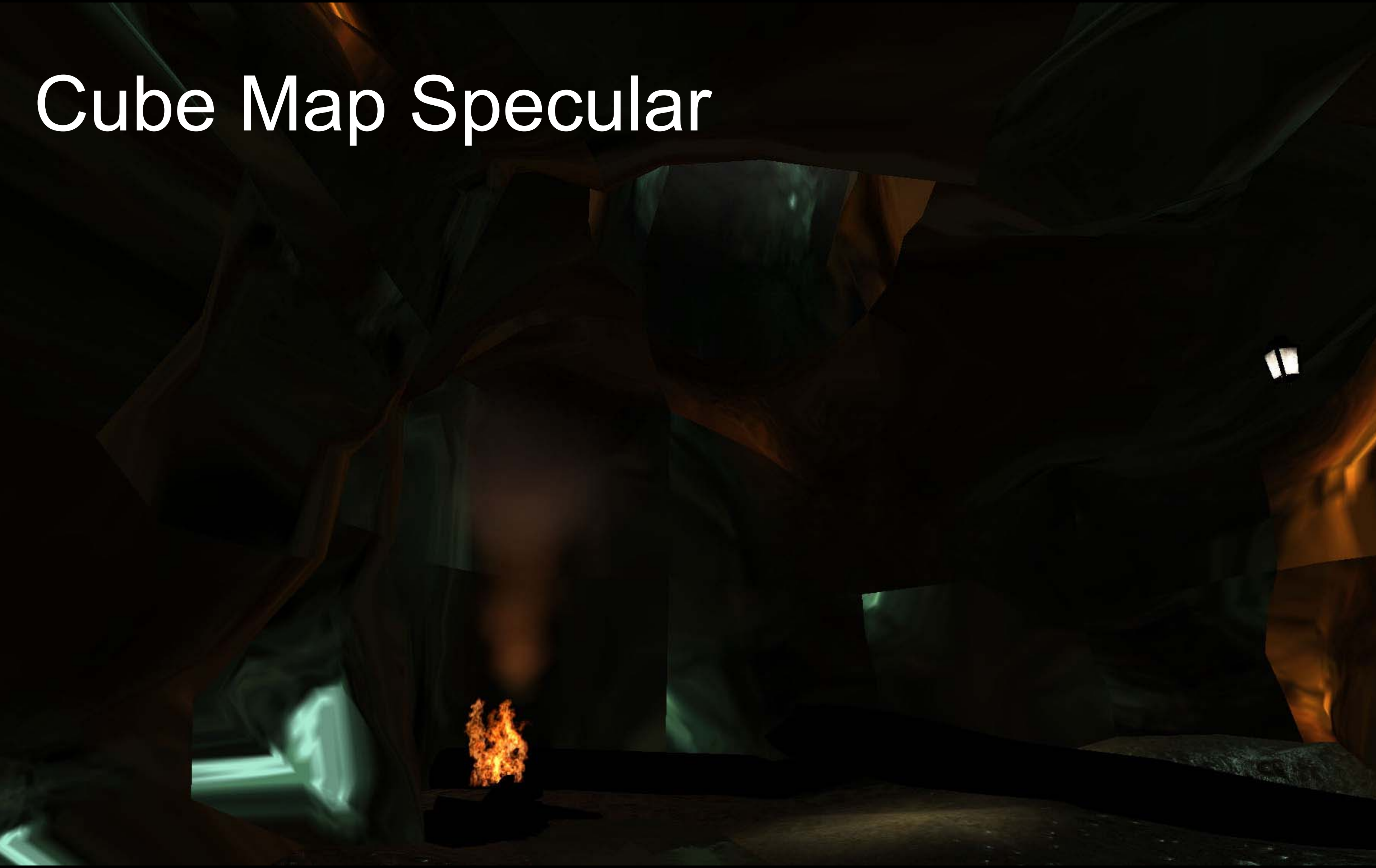


Albedo * Radiosity



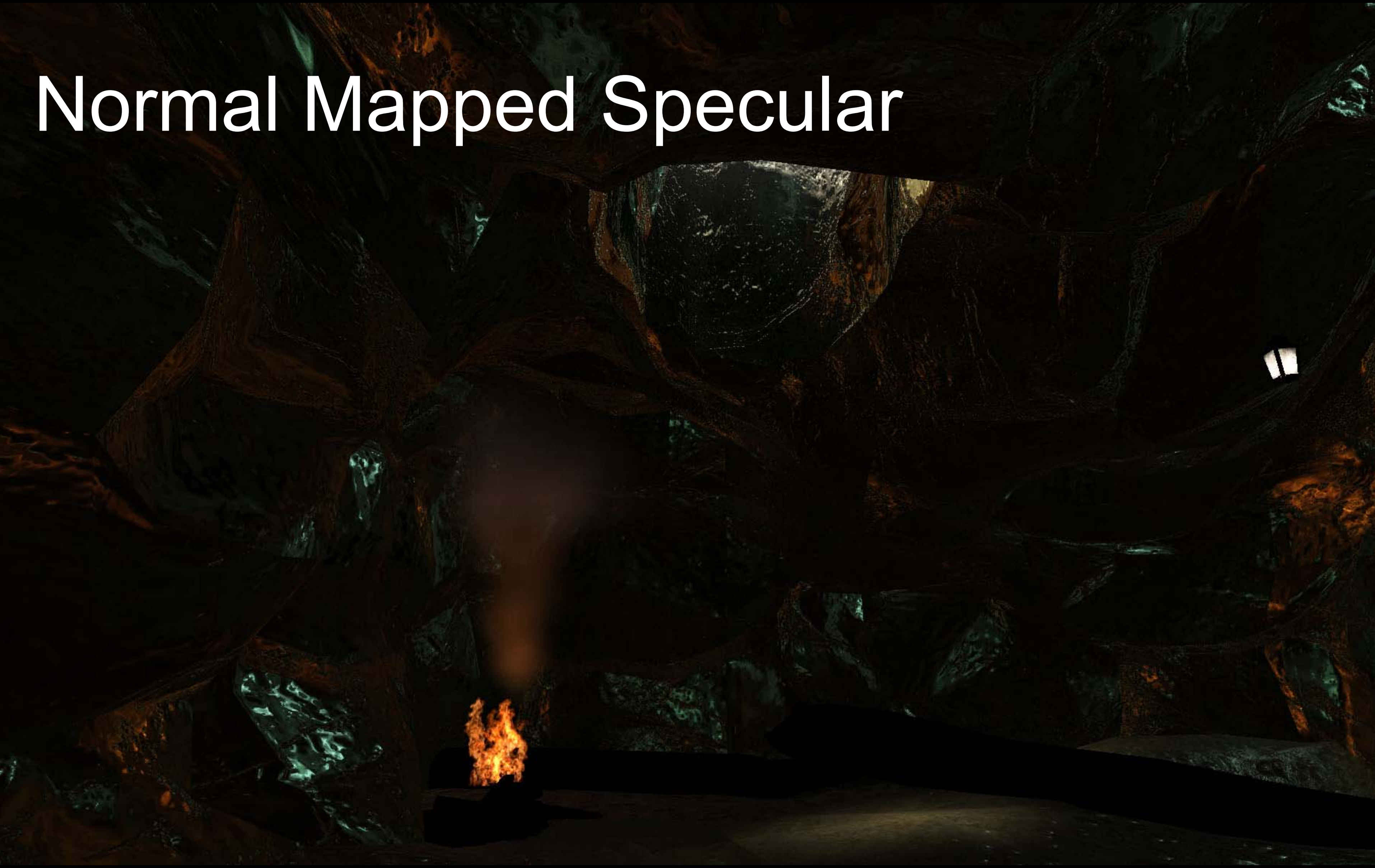
Radiosity Normal Mapping Shade Tree



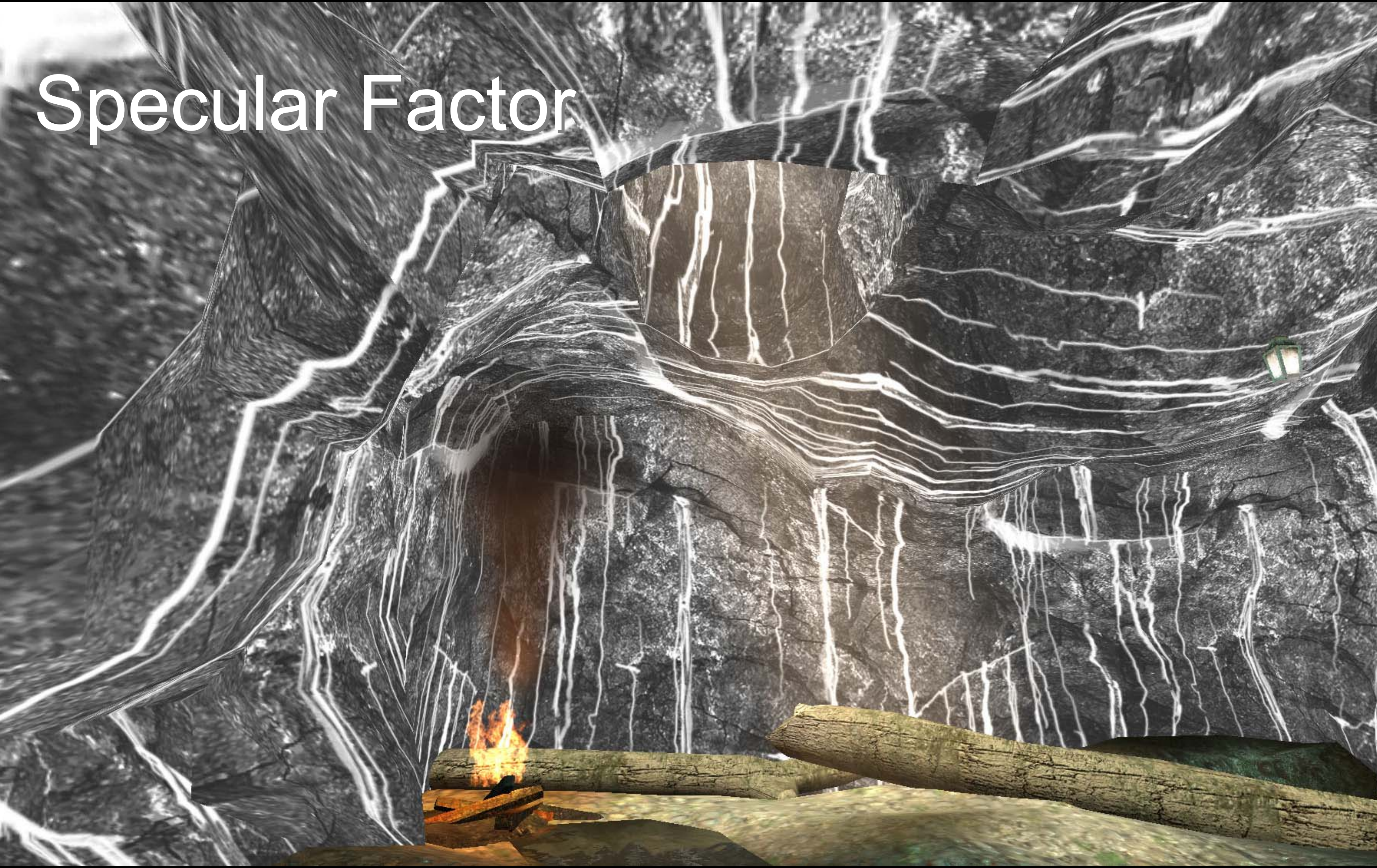


Cube Map Specular

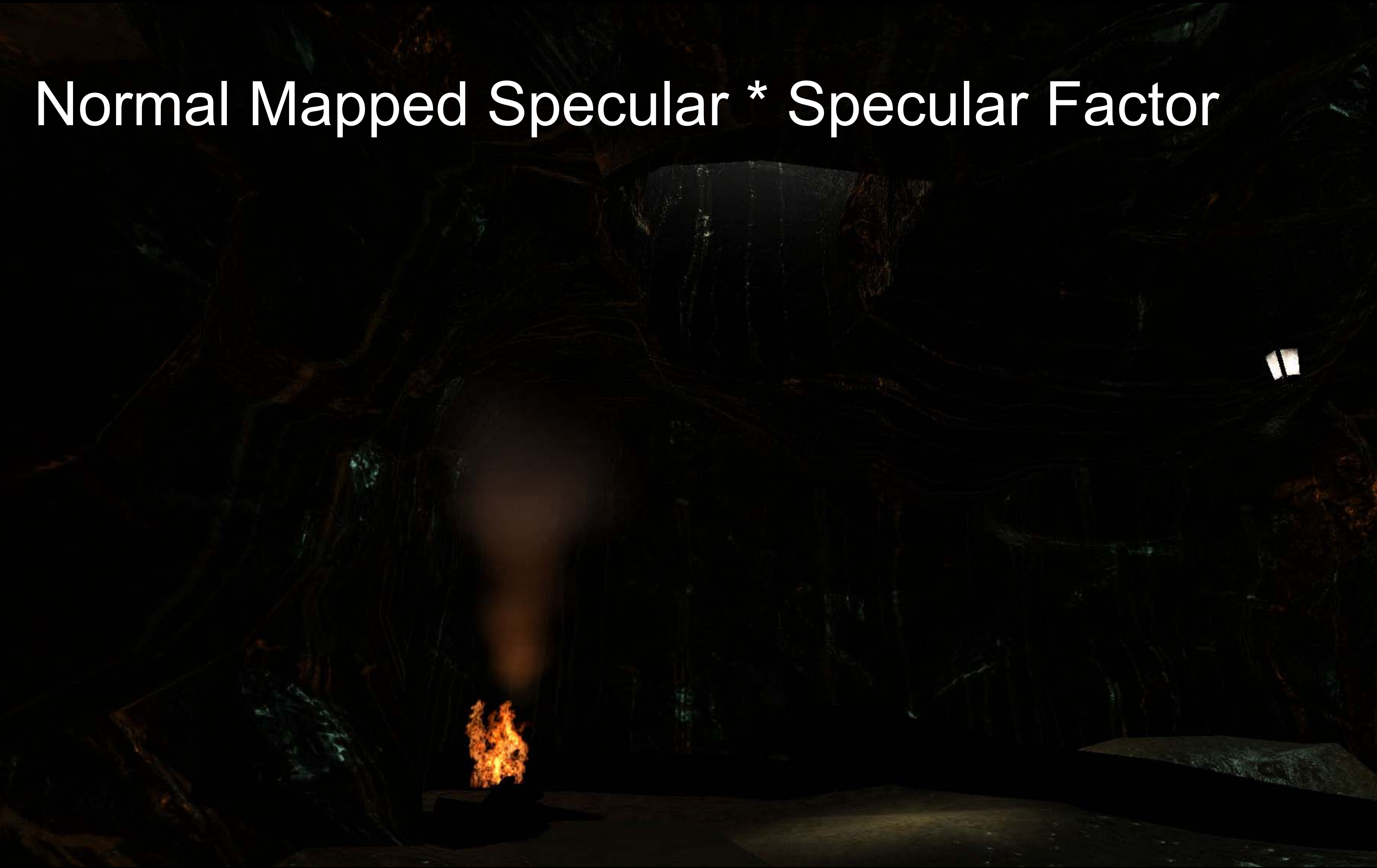
Normal Mapped Specular



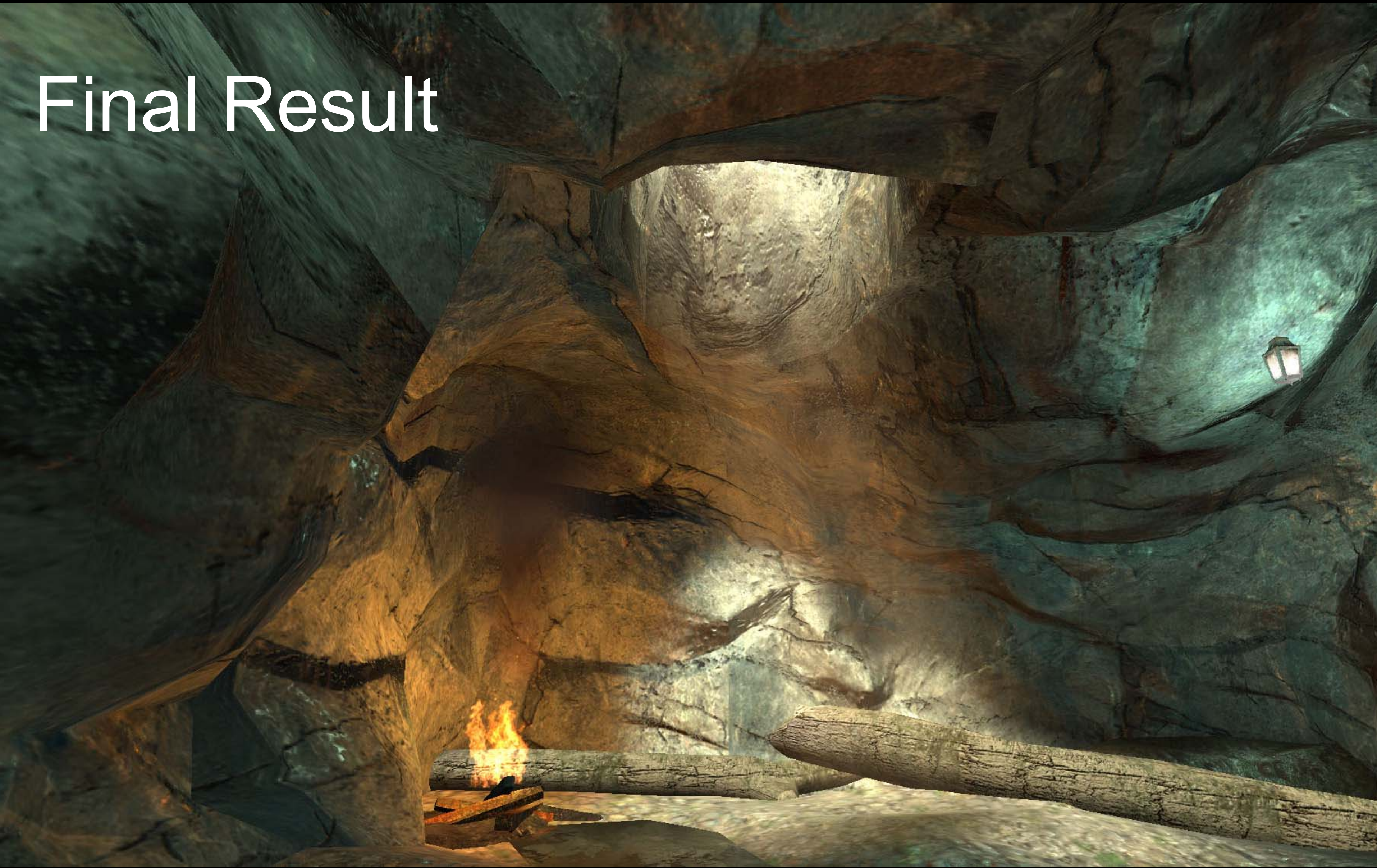
Specular Factor



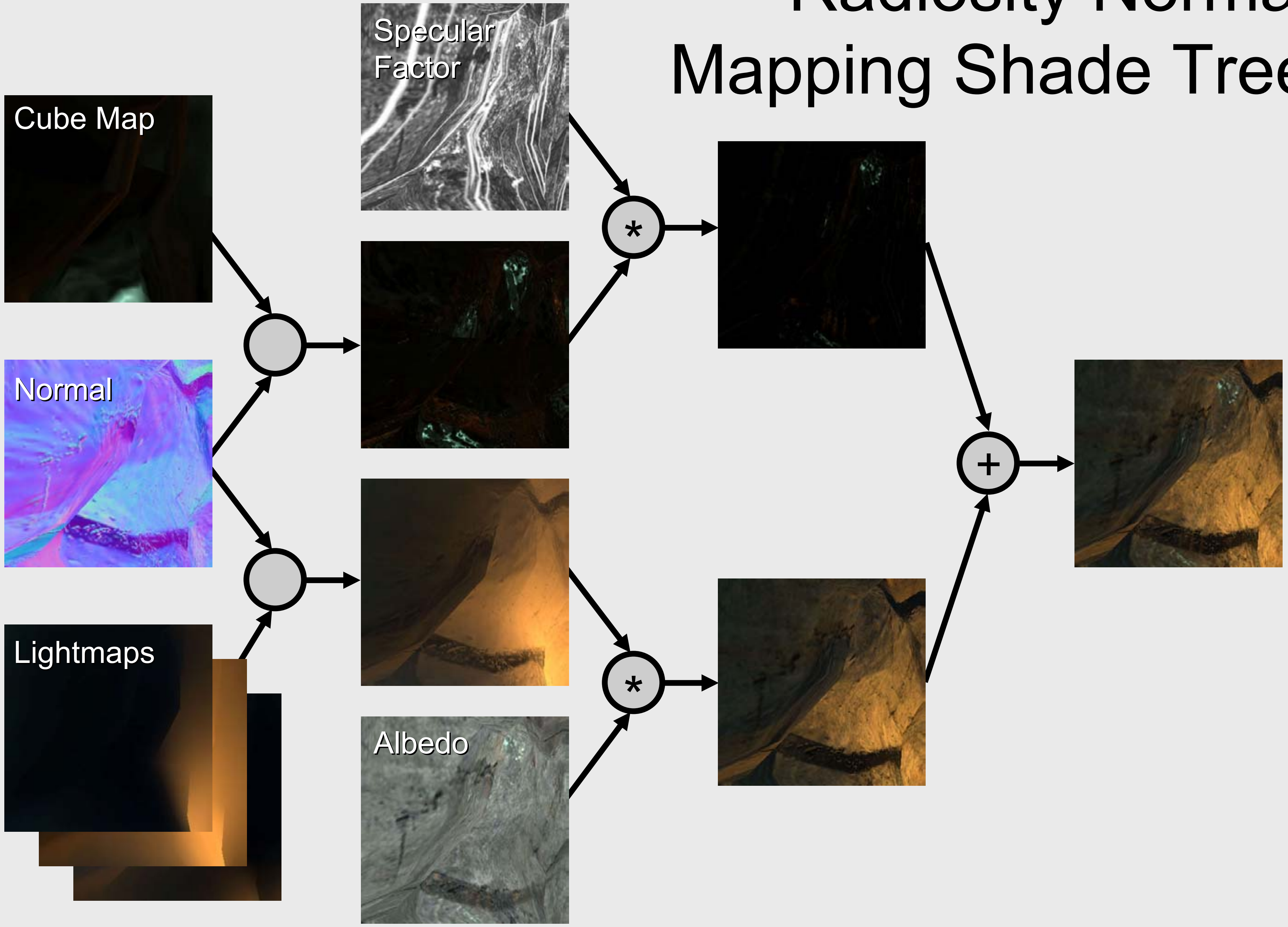
Normal Mapped Specular * Specular Factor



Final Result



Radiosity Normal Mapping Shade Tree



Code Specialization

- Shader permutations generated offline
- Static constants control code specialization
  - Like #ifdef
- 1920 different pixel shaders

Constant Pixel Shader Controls

```
static const bool g_bBaseTexture;
static const bool g_bDetailTexture;
static const bool g_bBumpmap;
static const bool g_bDiffuseBumpmap;
static const bool g_bCubemap;
static const bool g_bVertexColor;
static const bool g_bEnvmapMask;
static const bool g_bBaseAlphaEnvmapMask;
static const bool g_bSelfIllum;
static const bool g_bNormalMapAlphaEnvmapMask;
```

Invalid combinations

- Some of these booleans have interactions and we can disable certain combinations in our offline process to save runtime memory and speed up compilation.
- Some things like detail textures and normal maps we consider to be mutually exclusive.
- The base map’s alpha can be used for one of several things like an environment map mask, or emissive mask so there are some combinations skipped based on this

Constant Controls

- Direct-mapped to specific registers
  - Consistency with legacy paths
  - Don’t have to deal with HLSL constant table

```
const float4 g_EnvmapTint                : register( c0 );
const float3 g_EnvmapContrast            : register( c2 );
const float3 g_EnvmapSaturation          : register( c3 );
const float4 g_FresnelReflectionReg      : register( c4 );
const float  g_OverbrightFactor          : register( c6 );
const float4 g_SelfIllumTint             : register( c7 );
```

Samplers

```
sampler BaseTextureSampler    : register( s0 );
sampler LightmapSampler       : register( s1 );
sampler EnvmapSampler         : register( s2 );
sampler DetailSampler         : register( s3 );
sampler BumpmapSampler        : register( s4 );
sampler EnvmapMaskSampler     : register( s5 );
sampler NormalizeSampler      : register( s6 );
```

Pixel Shader Input

```
struct PS_INPUT
{
   float2 baseTexCoord                         : TEXCOORD0;
   float4 detailOrBumpAndEnvmapMaskTexCoord : TEXCOORD1;
   float4 lightmapTexCoord1And2               : TEXCOORD2;
   float2 lightmapTexCoord3                   : TEXCOORD3;
   float3 worldVertToEyeVector                : TEXCOORD4;
   float3x3 tangentSpaceTranspose            : TEXCOORD5;
   float4 vertexColor                          : COLOR;
};
```

Pixel Shader **main()**

```
float4 main( PS_INPUT i ) : COLOR
{
    float3 albedo = GetAlbedo( i );
    float  alpha  = GetAlpha( i );

    float3 diffuseLighting = GetDiffuseLighting( i );
    float3 specularLighting = GetSpecularLighting( i );

    float3 diffuseComponent = albedo * diffuseLighting;
    diffuseComponent *= g_OverbrightFactor;

    if( g_bSelfIllum )
    {
        float3 selfIllumComponent = g_SelfIllumTint * albedo;
        diffuseComponent = lerp(diffuseComponent, selfIllumComponent, GetBaseTexture(i).a );
    }

    return float4( diffuseComponent + specularLighting, alpha );
}
```

Diffuse Lighting

```
float4 main( PS_INPUT i ) : COLOR
{
    float3 albedo = GetAlbedo( i );
    float  alpha  = GetAlpha( i );

    float3 diffuseLighting = GetDiffuseLighting( i );
    float3 specularLighting = GetSpecularLighting( i );

    float3 diffuseComponent = albedo * diffuseLighting;
    diffuseComponent *= g_OverbrightFactor;

    if( g_bSelfIllum )
    {
        float3 selfIllumComponent = g_SelfIllumTint * albedo;
        diffuseComponent = lerp(diffuseComponent, selfIllumComponent, GetBaseTexture(i).a );
    }

    return float4( diffuseComponent + specularLighting, alpha );
}
```

GetDiffuseLighting()

```
float3 GetDiffuseLighting( PS_INPUT i )
{
   if( g_bBumpmap )
   {
        return GetDiffuseLightingBumped( i );
   }
   else
   {
        return GetDiffuseLightingUnbumped( i );
   }
}
```

GetDiffuseLightingUnbumped()

```
float3 GetDiffuseLightingUnbumped( PS_INPUT i )
{
   float2 bumpCoord1 = ComputeLightmapCoordinates(
                                       i.lightmapTexCoord1And2,
                                       i.lightmapTexCoord3.xy );

   return tex2D( LightmapSampler, bumpCoord1 );
}
```

GetDiffuseLightingBumped()

```
float3 GetDiffuseLightingBumped( PS_INPUT i )
{
   float2 bumpCoord1, bumpCoord2, bumpCoord3;
   ComputeBumpedLightmapCoordinates( i.lightmapTexCoord1And2,
                                     i.lightmapTexCoord3.xy,
                                     bumpCoord1, bumpCoord2, bumpCoord3 );

   float3 lightmapColor1 = tex2D( LightmapSampler, bumpCoord1 );
   float3 lightmapColor2 = tex2D( LightmapSampler, bumpCoord2 );
   float3 lightmapColor3 = tex2D( LightmapSampler, bumpCoord3 );
   float3 normal = GetNormal( i );

   float3 diffuseLighting =
                 saturate( dot( normal, bumpBasis[0] ) ) * lightmapColor1 +
                 saturate( dot( normal, bumpBasis[1] ) ) * lightmapColor2 +
                 saturate( dot( normal, bumpBasis[2] ) ) * lightmapColor3;

   return diffuseLighting;
}
```

Specular Lighting

```
float4 main( PS_INPUT i ) : COLOR
{
    float3 albedo = GetAlbedo( i );
    float  alpha  = GetAlpha( i );

    float3 diffuseLighting = GetDiffuseLighting( i );
    float3 specularLighting = GetSpecularLighting( i );

    float3 diffuseComponent = albedo * diffuseLighting;
    diffuseComponent *= g_OverbrightFactor;

    if( g_bSelfIllum )
    {
        float3 selfIllumComponent = g_SelfIllumTint * albedo;
        diffuseComponent = lerp(diffuseComponent, selfIllumComponent, GetBaseTexture(i).a );
    }

    return float4( diffuseComponent + specularLighting, alpha );
}
```

GetSpecularLighting()

```
float3 GetSpecularLighting( PS_INPUT i )
{
   float3 specularFactor = GetSpecularFactor( i );
   float3 normal = GetNormal( i );

   float3 specularLighting = float3( 0.0f, 0.0f, 0.0f );
   if( g_bCubemap )
   {
      float3 worldSpaceNormal = mul( normal, i.tangentSpaceTranspose );
      float fresnel = Fresnel( i, worldSpaceNormal );
      float3 reflectVect = CalcReflectionVectorUnnormalized( worldSpaceNormal,
                                                             i.worldVertToEyeVector );

      specularLighting = texCUBE(EnvmapSampler, reflectVect) * specularFactor * g_EnvmapTint;
      specularLighting = fresnel * specularLighting;
   }

   return specularLighting;
}
```

Self Illumination

```
float4 main( PS_INPUT i ) : COLOR
{
    float3 albedo = GetAlbedo( i );
    float  alpha  = GetAlpha( i );

    float3 diffuseLighting = GetDiffuseLighting( i );
    float3 specularLighting = GetSpecularLighting( i );

    float3 diffuseComponent = albedo * diffuseLighting;
    diffuseComponent *= g_OverbrightFactor;

    if( g_bSelfIllum )
    {
        float3 selfIllumComponent = g_SelfIllumTint * albedo;
        diffuseComponent = lerp(diffuseComponent, selfIllumComponent, GetBaseTexture(i).a );
    }

    return float4( diffuseComponent + specularLighting, alpha );
}
```

Final Composite

```
float4 main( PS_INPUT i ) : COLOR
{
    float3 albedo = GetAlbedo( i );
    float  alpha  = GetAlpha( i );

    float3 diffuseLighting = GetDiffuseLighting( i );
    float3 specularLighting = GetSpecularLighting( i );

    float3 diffuseComponent = albedo * diffuseLighting;
    diffuseComponent *= g_OverbrightFactor;

    if( g_bSelfIllum )
    {
        float3 selfIllumComponent = g_SelfIllumTint * albedo;
        diffuseComponent = lerp(diffuseComponent, selfIllumComponent, GetBaseTexture(i).a );
    }

    return float4( diffuseComponent + specularLighting, alpha );
}
```

Resulting Shaders

- Many shaders are generated by this process
- Longest ps_2_0 shader is 43 ALU ops
- Up to 7 texture fetches
- Everything is single-passed on ps_2_0
- Takes three passes on ps_1_1

Most Complex Resulting Pixel Shader

```
texld r6, t1, s4
mad r7.xyz, c1.x, r6, c1.y
dp3 r8.x, r7, t5
dp3 r8.y, r7, t6
dp3 r8.z, r7, t7
dp3 r1.x, r8, t4
dp3 r0.x, r8, r8
add r0.w, r1.x, r[illegible].x
mul r0.xyz, r0.x, t4
mad r0.xyz, r0.w, r8, -r0
mov r1.xy, t2.wzyx
texld r5, r0, s2
texld r4, t4, s6
texld r3, r1, s1
texld r2, t2, s1
texld r1, t3, s1
texld r0, t0, s0
```

```
mul r5.xyz, r6.w, r5
mul r6.xyz, r5, c0
mad r5.xyz, r6, r6, -r6
mad r4.xyz, c1.x, r4, c1.y
dp3 r4.x, r8, r4
dp3_sat r8.x, r7, c8
mul r3.xyz, r3, r[illegible].[illegible]
dp3_sat r8.[illegible], r[illegible], c[illegible]
dp3_sat r7.[illegible], r7, c[illegible]
mad r2.xyz, r8.x, r2, r3
mad r1.xyz, r7.x, r1, r2
add r1.w, -r4.x, c1.z
mul r0.xyz, r0, v0
mul r2.w, r1.w, r1.w
mul r1.xyz, r1, r0
mul r2.w, r2.w, r2.w
mul r1.xyz, r1, c6.x
```

```
mul r1.w, r1.w, r2.w
mad r0.xyz, c7, r0, -r1
mad r1.w, r1.w, c4.z, c4.w
mad r0.xyz, r0.w, r0, r1
mad r2.xyz, c2, r5, r6
max r2.w, r0.x, r0.y
dp3 r3.x, r2, c10
max r3.w, r0.z, c1.z
lrp r1.xyz, c3, r2, r3.x
max r0.w, r2.w, r3.w
mul r1.xyz, r1.w, r1
rcp r0.w, r0.w
mad r0.xyz, r0, r0.w, r1
mov r0.w, v0.x
mov oC0, r0
```

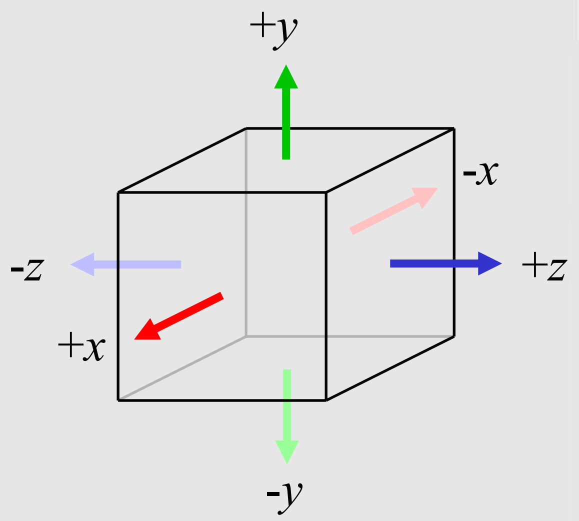
Use HLSL

Model Shader

- Input to vertex shader is 2 local lights plus a directional ambient term via an “ambient cube” sourced from the radiosity solution.

Ambient cube

- Used to realistically integrate models with the world via indirect lighting.
- Ambient illumination for model geometry is sampled at runtime from data generated by the radiosity solution. Any local lights that aren't important enough to go directly into the vertex shader are also added to the ambient cube.
- Six colors are stored spatially in our level data:
  - Represent ambient light flowing through that volume in space
  - Application looks this up for each model to determine the six ambient colors to use for a given model.



Unbumped Ambient Cube Math

```
float3 AmbientLight( const float3 worldNormal )
{
   float3 nSquared = worldNormal * worldNormal;
   int3 isNegative = ( worldNormal < 0.0 );
   float3 linearColor;
   linearColor = nSquared.x * cAmbientCube[isNegative.x]   +
                 nSquared.y * cAmbientCube[isNegative.y+2] +
                 nSquared.z * cAmbientCube[isNegative.z+4];
   return linearColor;
}
```

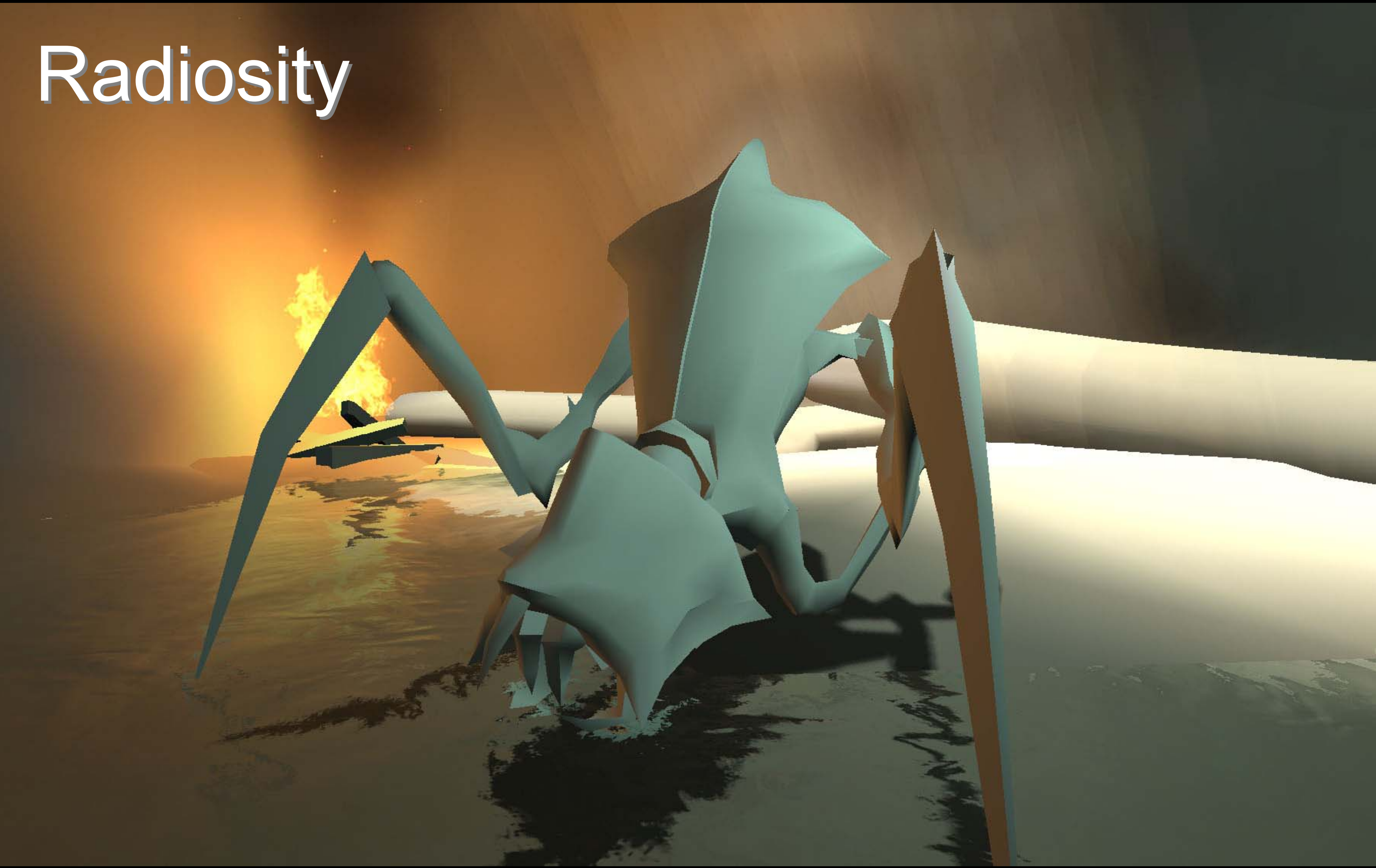
Radiosity normal mapped models

- Similar to world shader
- Accumulates lighting onto the same basis in the vertex shader

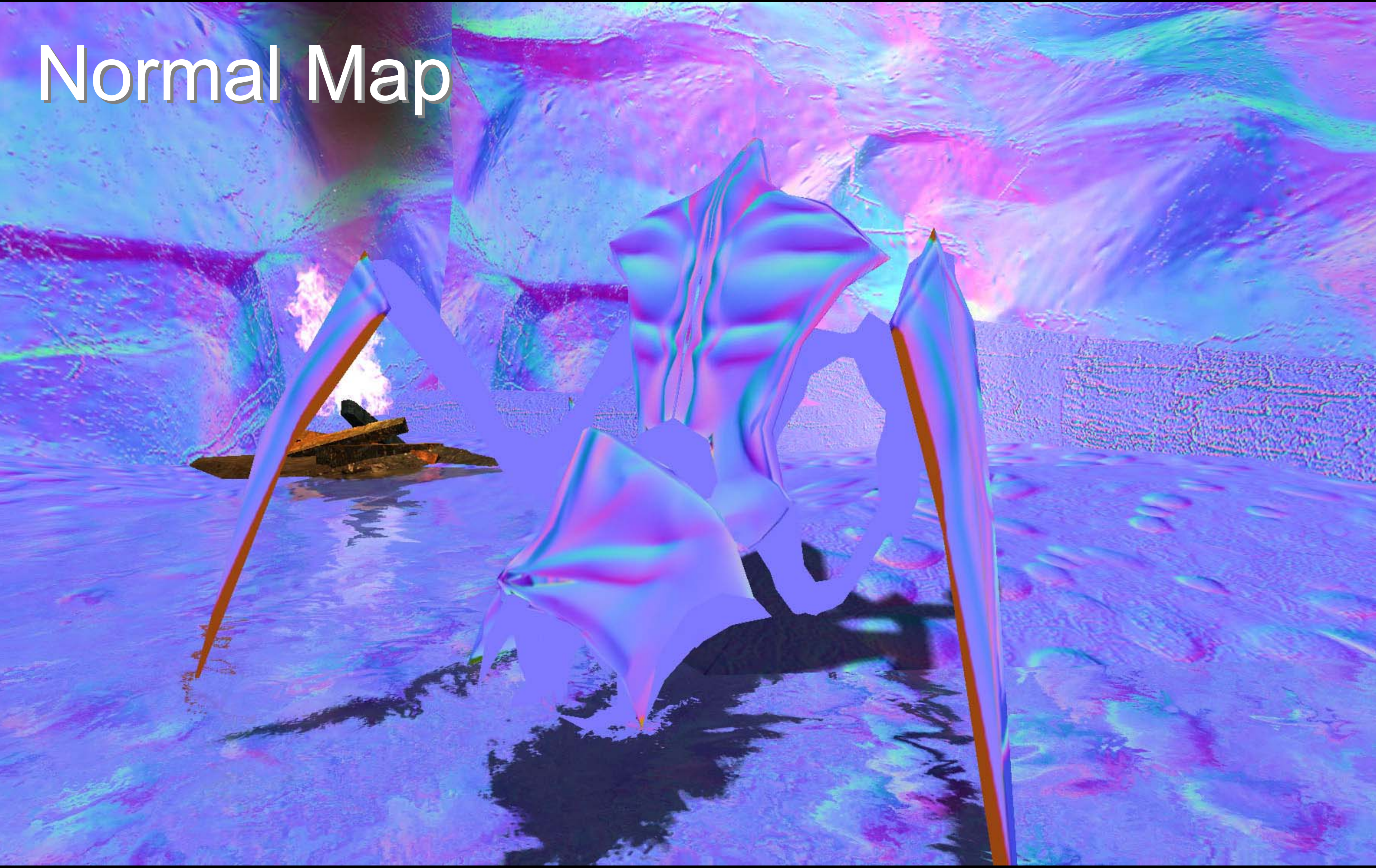
Specular lighting for models

- Similar to world shader
- Pick the nearest cube map sample and use it
- Could blend between samples, but we don’t currently

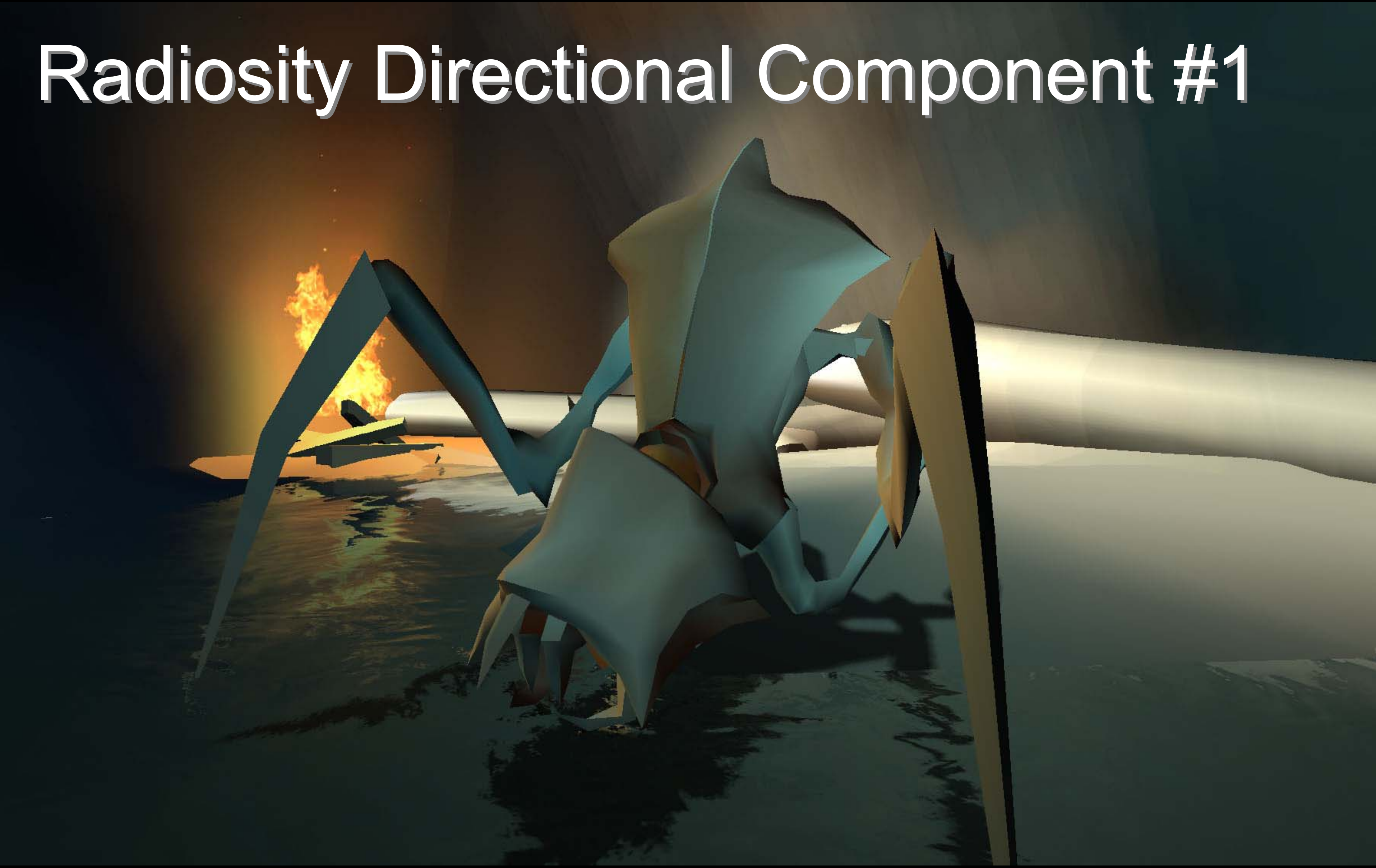
Radiosity



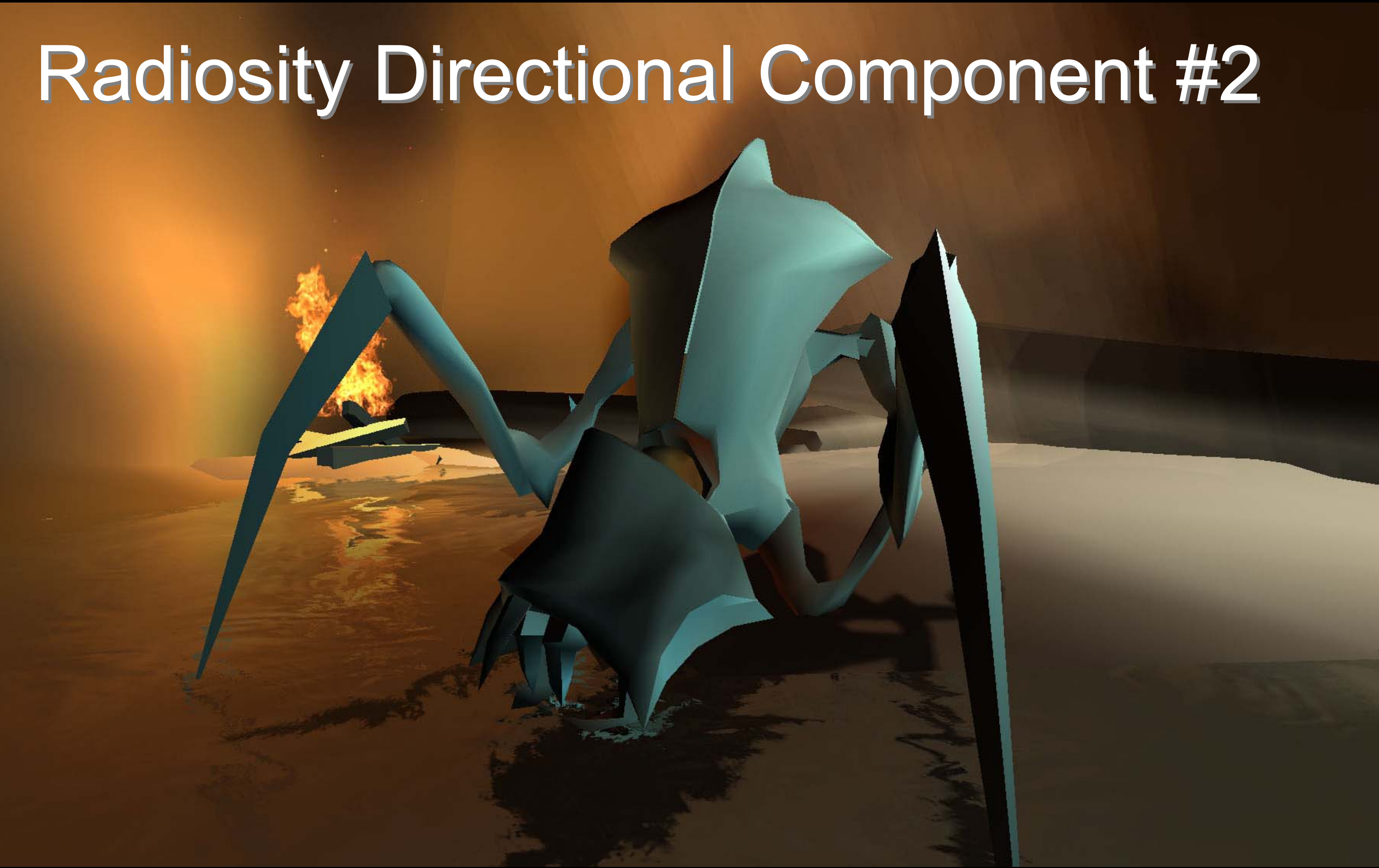
Normal Map



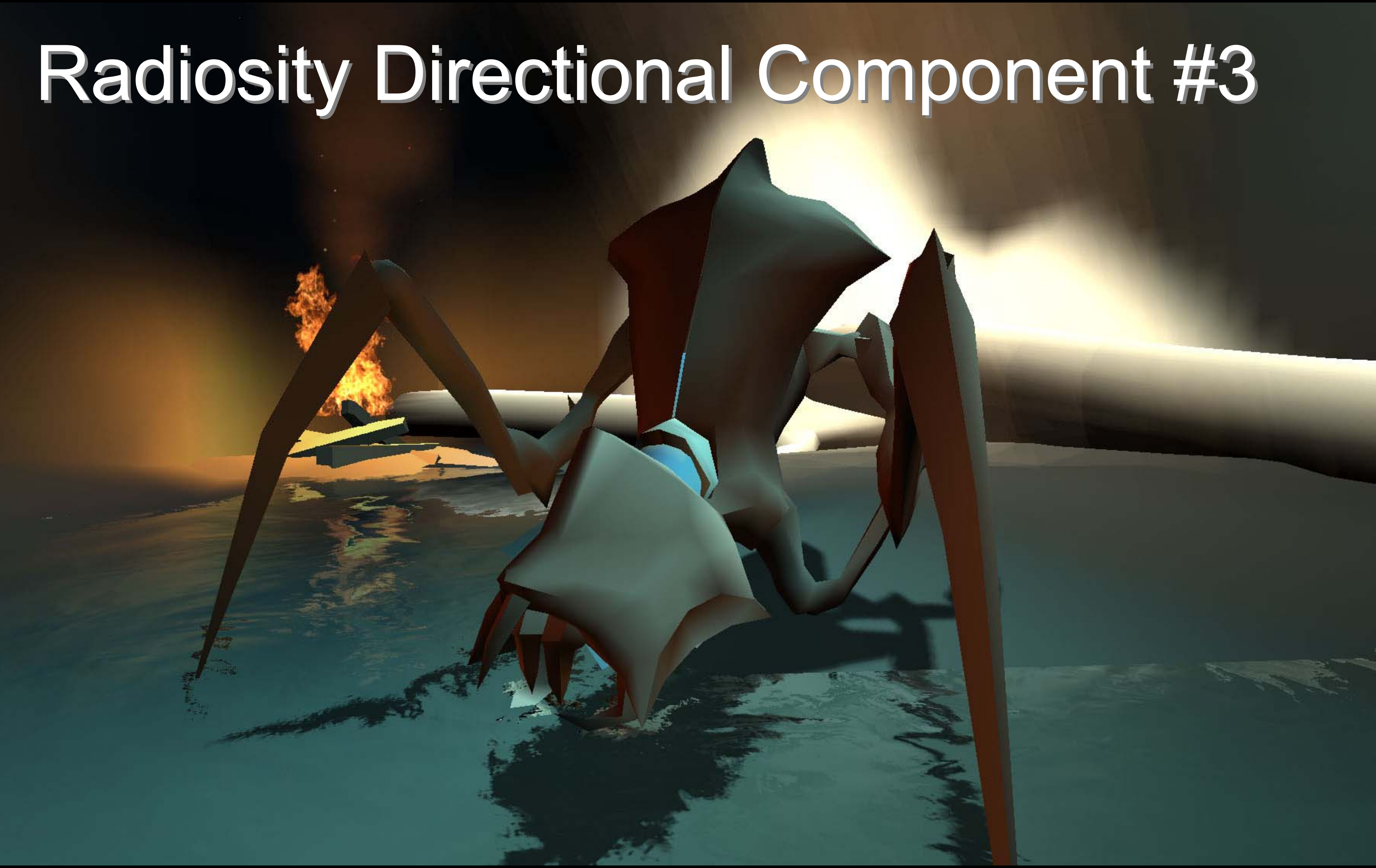
Radiosity Directional Component #1



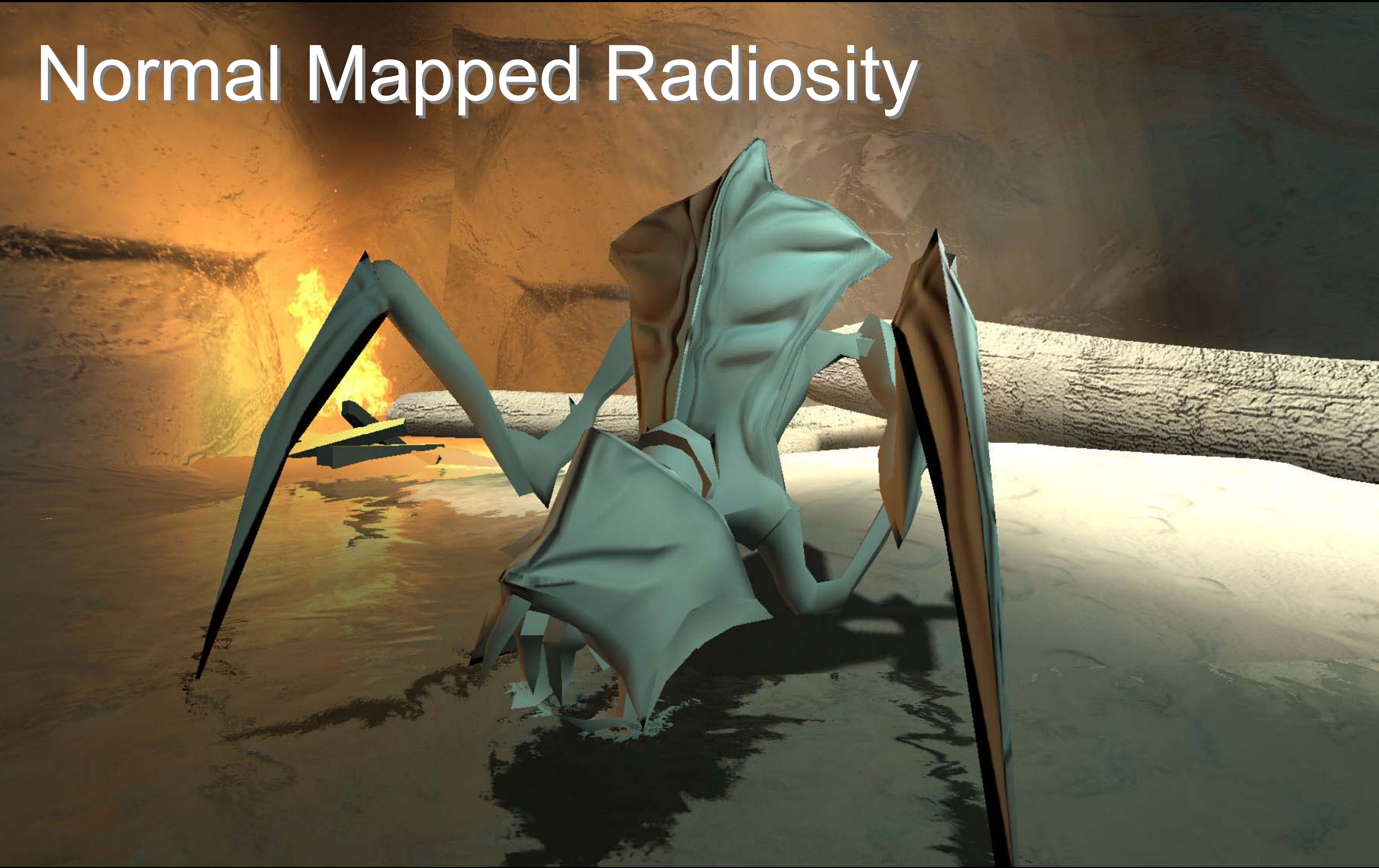
Radiosity Directional Component #2



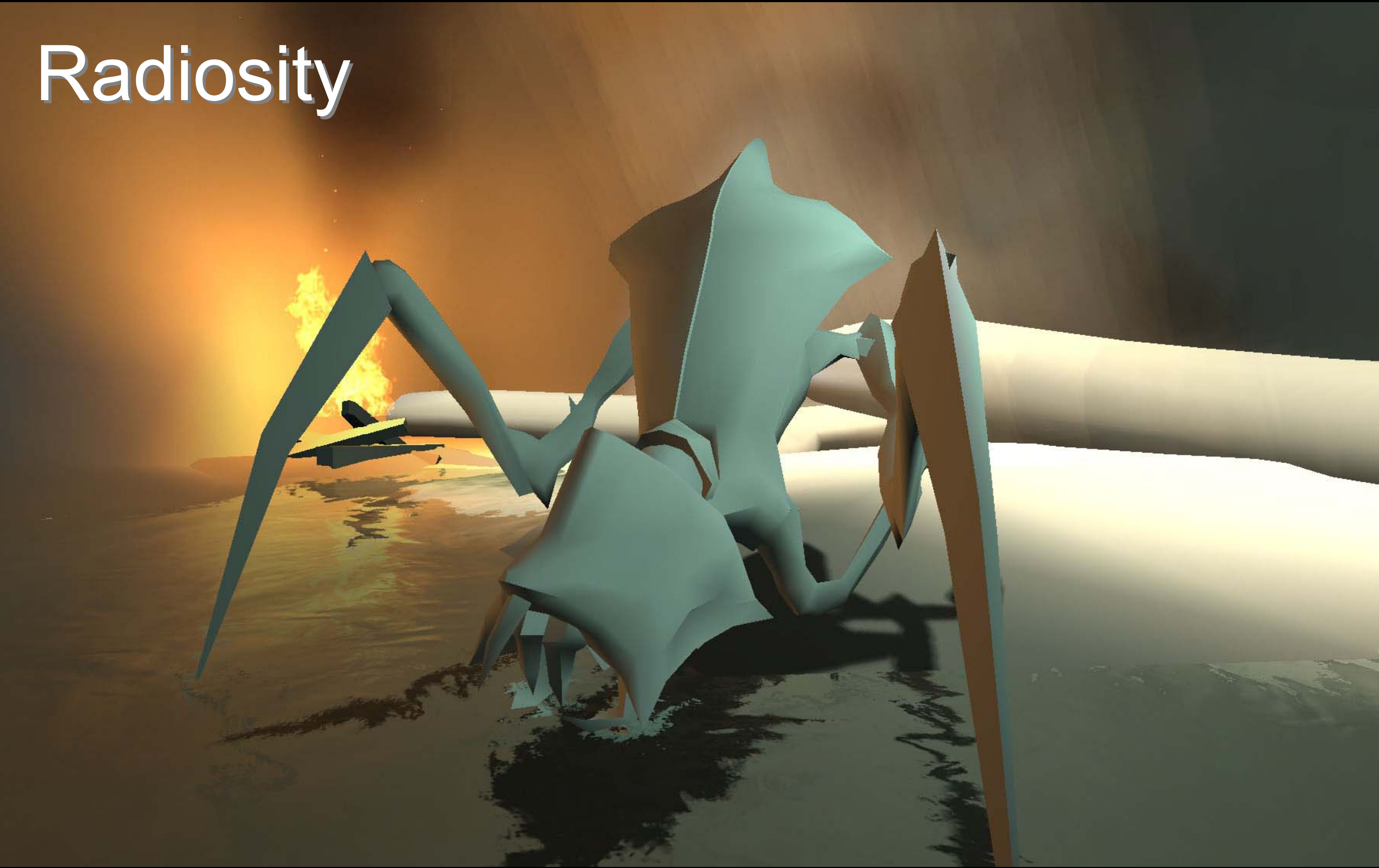
Radiosity Directional Component #3

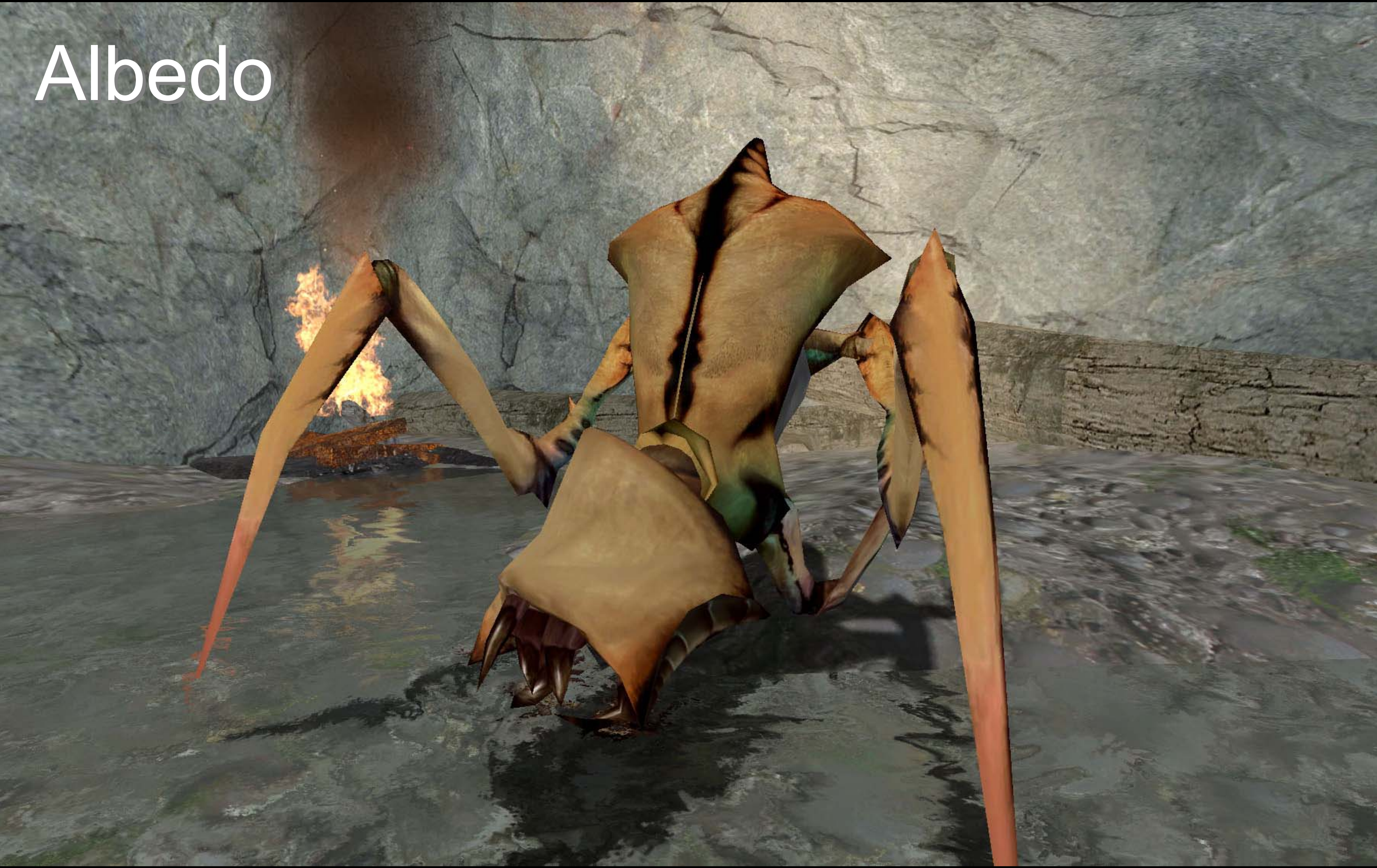


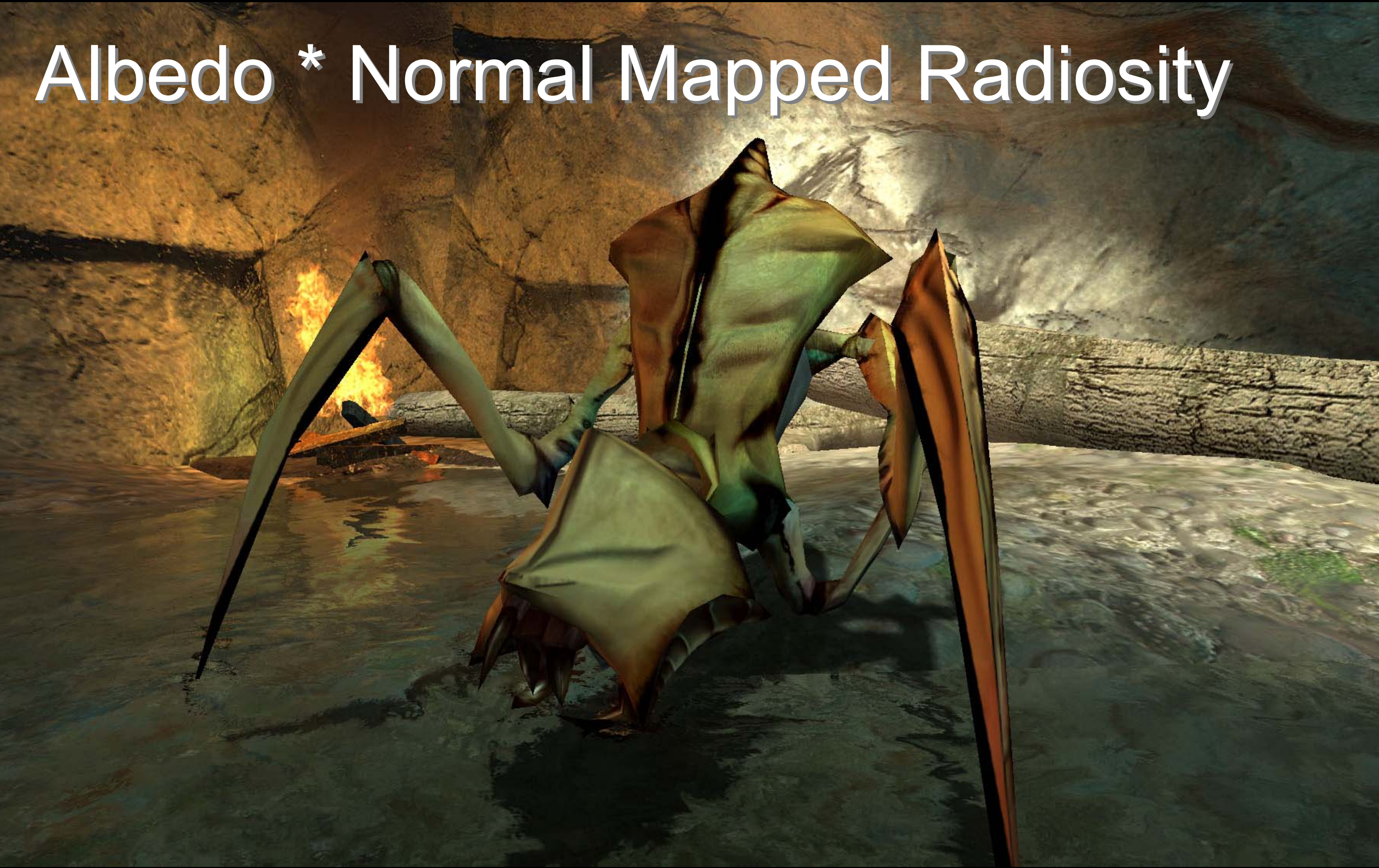
Normal Mapped Radiosity



Radiosity

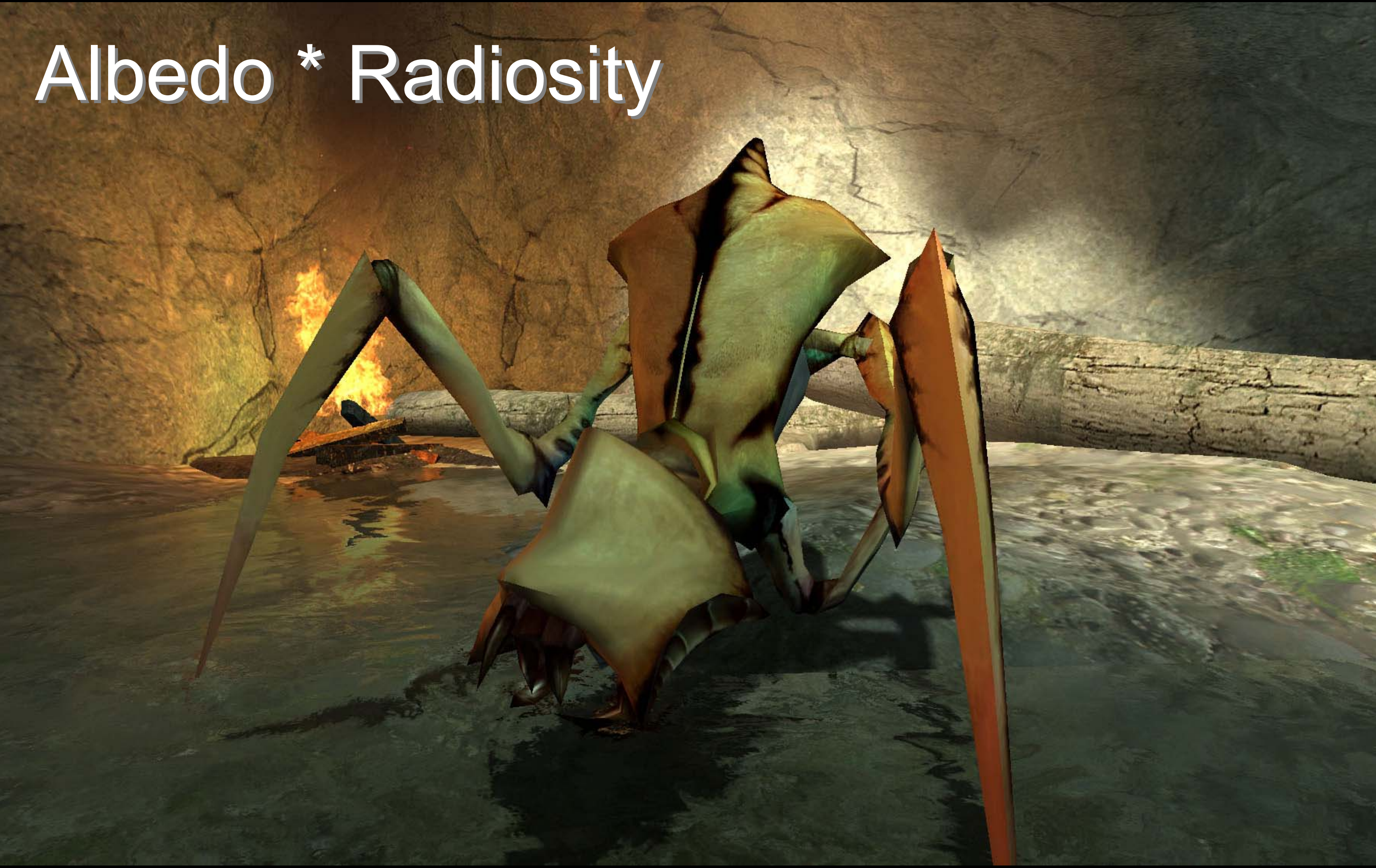



Albedo

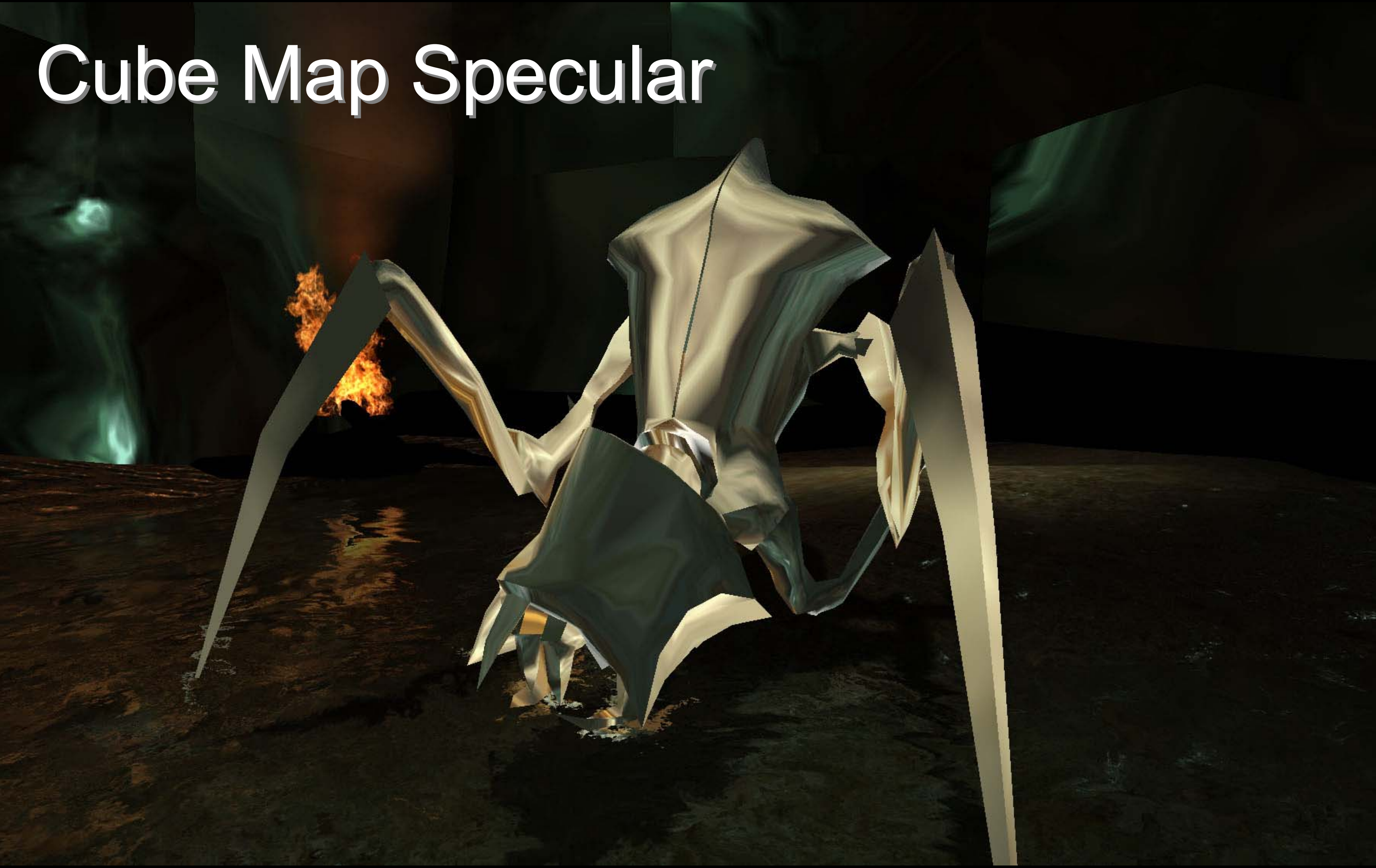


Albedo * Normal Mapped Radiosity

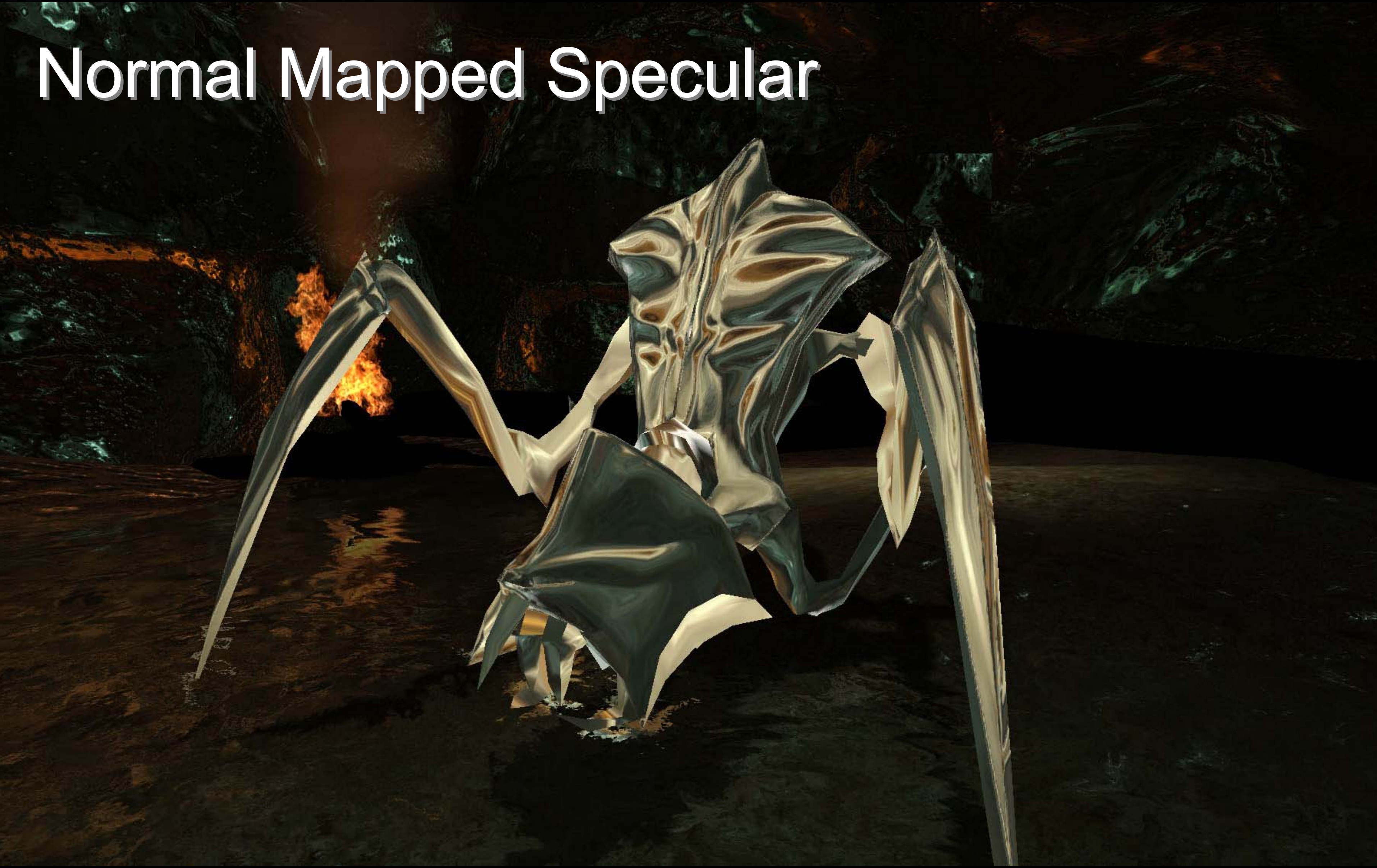
Albedo * Radiosity



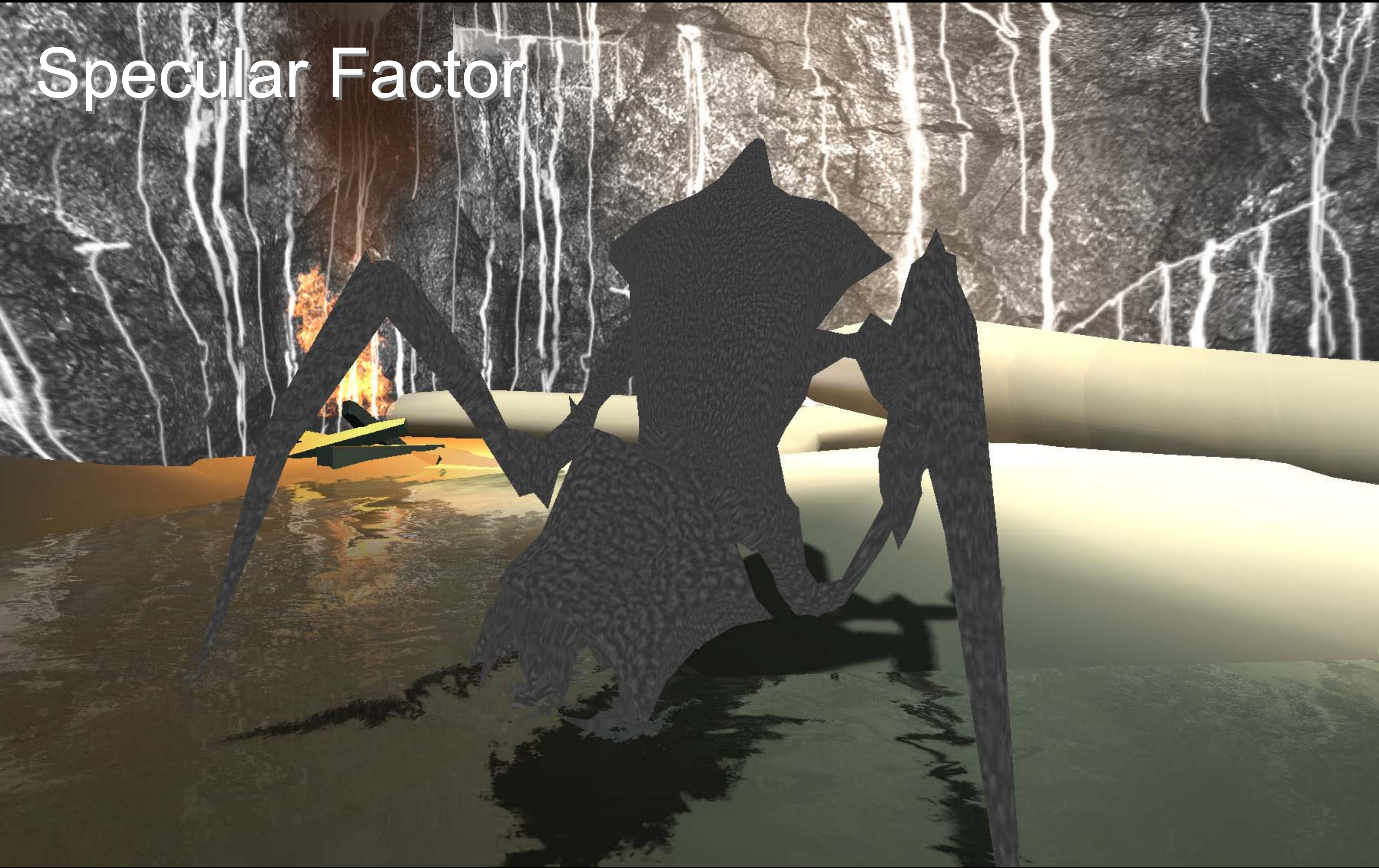
Cube Map Specular



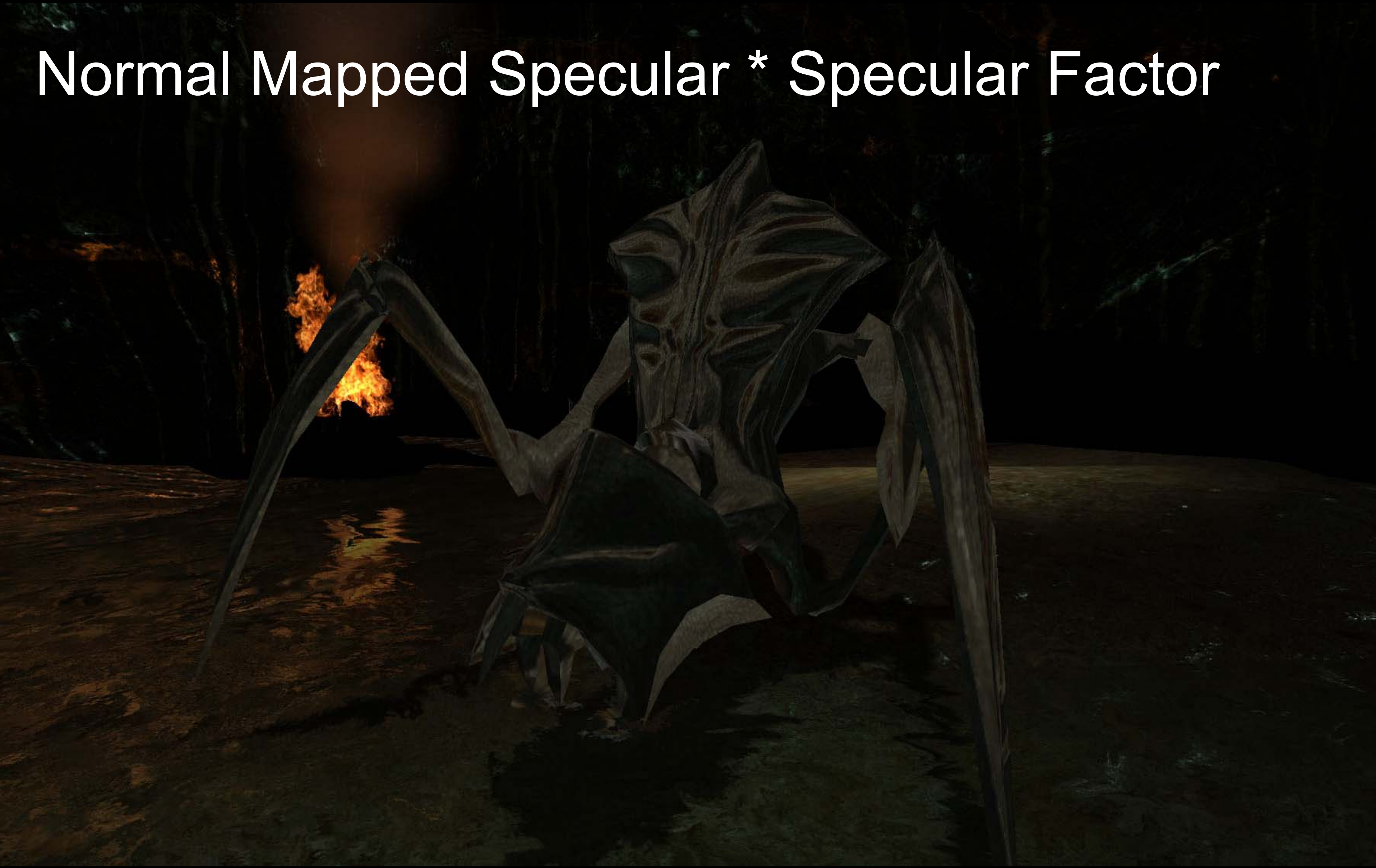
Normal Mapped Specular



Specular Factor



Normal Mapped Specular * Specular Factor



Final Result

Vertex Shader Combinations for Models

```
static const int  g_LightCombo
static const int  g_FogType
static const int  g_NumBones
static const bool g_bBumpmap
static const bool g_bVertexColor
static const bool g_bNormalOrTangentSpace

static const int g_StaticLightType  = g_StaticLightTypeArray[g_LightCombo];
static const int g_AmbientLightType = g_AmbientLightTypeArray[g_LightCombo];
static const int g_LocalLightType0  = g_LocalLightType0Array[g_LightCombo];
static const int g_LocalLightType1  = g_LocalLightType1Array[g_LightCombo];
```

Model Vertex Shader Inputs

```
struct VS_INPUT
{
   float4 vPos            : POSITION;
   float4 vBoneWeights    : BLENDWEIGHT;
   float4 vBoneIndices    : BLENDINDICES;
   float3 vNormal         : NORMAL;
   float4 vColor          : COLOR0;
   float3 vSpecular       : COLOR1;
   float4 vTexCoord0      : TEXCOORD0;
   float4 vTexCoord1      : TEXCOORD1;
   float4 vTexCoord2      : TEXCOORD2;
   float4 vTexCoord3      : TEXCOORD3;
   float3 vTangentS       : TANGENT;
   float3 vTangentT       : BINORMAL;
   float4 vUserData       : TANGENT;
};
```

Model Vertex Shader Outputs

This is what we're computing on the next few slides:

```
struct VS_OUTPUT
{
    float4 projPos                      : POSITION;
    float  fog                          : FOG;
    float2 baseTexCoord                 : TEXCOORD0;
    float2 detailOrBumpTexCoord         : TEXCOORD1;
    float2 envmapMaskTexCoord           : TEXCOORD2;
    float3 worldVertToEyeVector         : TEXCOORD3;
    float3x3 tangentSpaceTranspose      : TEXCOORD4;
    float4 color1                       : COLOR0;
    float3 color2                       : COLOR1;
    float3 color3                       : TEXCOORD7;
};
```

Vertex Shader **main()**

```
VS_OUTPUT main( const VS_INPUT v ) {

  … skin …fog…

     DoBumped( worldPos, worldNormal, worldTangentS,
               worldTangentT, v.vSpecular, v.vSpecular,
               v.vSpecular, v.vSpecular, g_StaticLightType,
               g_AmbientLightType, g_LocalLightType0,
               g_LocalLightType1, 1.0f,
               o.color1.xyz, o.color2.xyz, o.color3.xyz );

    …
}
```

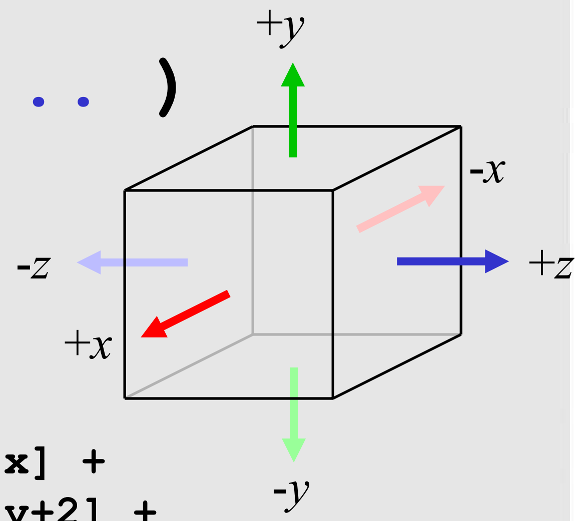
void DoBumped(...)

```
{
   // special case for no lighting
   if( staticLightType == LIGHTTYPE_NONE && ambientLightType == LIGHTTYPE_NONE &&
        localLightType0 == LIGHTTYPE_NONE && localLightType1 == LIGHTTYPE_NONE )
   {
        color1 = color2 = color3 = 0.0f;
        gammaColorNormal = 0.0f;
   }
   else if( staticLightType  == LIGHTTYPE_STATIC &&
            ambientLightType == LIGHTTYPE_NONE &&
            localLightType0  == LIGHTTYPE_NONE &&
            localLightType1  == LIGHTTYPE_NONE )
   {
        DoBumpedStaticLightingOnly( staticLightingColor1, staticLightingColor2,
                                    staticLightingColor3,color1, color2, color3 );
   }
   else
   {
        DoBumpedLighting( worldPos, worldNormal, worldTangentS, worldTangentT,
                          staticLightingColor1, staticLightingColor2,
                          staticLightingColor3, staticLightingColorNormal,
                          staticLightType, ambientLightType, localLightType0,
                          localLightType1, modulation, color1, color2, color3 );
   }
}
```

void DoBumpedLighting(...)

```
{
  float3 worldBumpBasis1, worldBumpBasis2, worldBumpBasis3;
  CalculateWorldBumpBasis( worldTangentS, worldTangentT, worldNormal,
                           worldBumpBasis1, worldBumpBasis2, worldBumpBasis3 );

  CalcBumpedStaticLighting(staticLightingColor1, staticLightingColor2,
                           staticLightingColor3,
                           staticLightType, color1, color2, color3 );

  if( ambientLightType == LIGHTTYPE_AMBIENT )
     AddBumpedAmbientLight( worldBumpBasis1, worldBumpBasis2, worldBumpBasis3,
                            worldNormal, color1, color2, color3 );


  if( localLightType0 != LIGHTTYPE_NONE )
     AddBumpedLight( worldPos, worldNormal, worldBumpBasis1, worldBumpBasis2,
                     worldBumpBasis3, 0, localLightType0, color1, color2,
                     color3 );


  if( localLightType1 != LIGHTTYPE_NONE )
     AddBumpedLight( worldPos, worldNormal, worldBumpBasis1, worldBumpBasis2,
                     worldBumpBasis3, 1, localLightType1,
                     color1, color2, color3 );
}
```

```
void AddBumpedAmbientLight( ... )
{
   float3 nSquared;
   int3 isNegative;

   nSquared = worldBumpBasis1 * worldBumpBasis1;
   isNegative = ( worldBumpBasis1 < 0.0 );
   color1 += nSquared.x * cAmbientCube[isNegative.x] +
             nSquared.y * cAmbientCube[isNegative.y+2] +
             nSquared.z * cAmbientCube[isNegative.z+4];

   nSquared = worldBumpBasis2 * worldBumpBasis2;
   isNegative = ( worldBumpBasis2 < 0.0 );
   color2 += nSquared.x * cAmbientCube[isNegative.x] +
             nSquared.y * cAmbientCube[isNegative.y+2] +
             nSquared.z * cAmbientCube[isNegative.z+4];

   nSquared = worldBumpBasis3 * worldBumpBasis3;
   isNegative = ( worldBumpBasis3 < 0.0 );
   color3 += nSquared.x * cAmbientCube[isNegative.x] +
             nSquared.y * cAmbientCube[isNegative.y+2] +
             nSquared.z * cAmbientCube[isNegative.z+4];
}
```



void AddBumpedLight(...)

```
{
   if( lightType == LIGHTTYPE_SPOT )
   {
      AddBumpedSpotLight( worldPos, lightNum, worldBumpBasis1,
                          worldBumpBasis2, worldBumpBasis3,
                          worldNormal, color1, color2, color3 );
   }
   else if( lightType == LIGHTTYPE_POINT )
   {
      AddBumpedPointLight( worldPos, lightNum, worldBumpBasis1,
                           worldBumpBasis2, worldBumpBasis3,
                           worldNormal, color1, color2, color3 );
   }
   else
   {
      AddBumpedDirectionalLight( lightNum, worldBumpBasis1,
                                 worldBumpBasis2, worldBumpBasis3,
                                 worldNormal, color1, color2, color3 );
   }
}
```

void AddBumpedPointLight(...)

```
{
   float3 lightDir = cLightInfo[lightNum].pos - worldPos;  // Light direction
   float lightDistSquared = dot( lightDir, lightDir );     // Light distance^2
   float ooLightDist = rsqrt( lightDistSquared );          // 1/lightDistance
   lightDir *= ooLightDist;                                // Normalize

   float4 attenuationFactors;                              // Dist attenuation
   attenuationFactors.x = 1.0f;
   attenuationFactors.y = lightDistSquared * ooLightDist;
   attenuationFactors.z = lightDistSquared;
   attenuationFactors.w = ooLightDist;

   float4 distanceAtten = 1.0f / dot( cLightInfo[lightNum].atten,
   attenuationFactors );

   // Compute N dot L
   float3 lambertAtten = float3( dot( worldBumpBasis1, lightDir ),
                                 dot( worldBumpBasis2, lightDir ),
                                 dot( worldBumpBasis3, lightDir ) );

   lambertAtten = max( lambertAtten, 0.0 );

   float3 color = cLightInfo[lightNum].color * distanceAtten;
   color1 += color * lambertAtten.x;
   color2 += color * lambertAtten.y;
   color3 += color * lambertAtten.z;
}
```

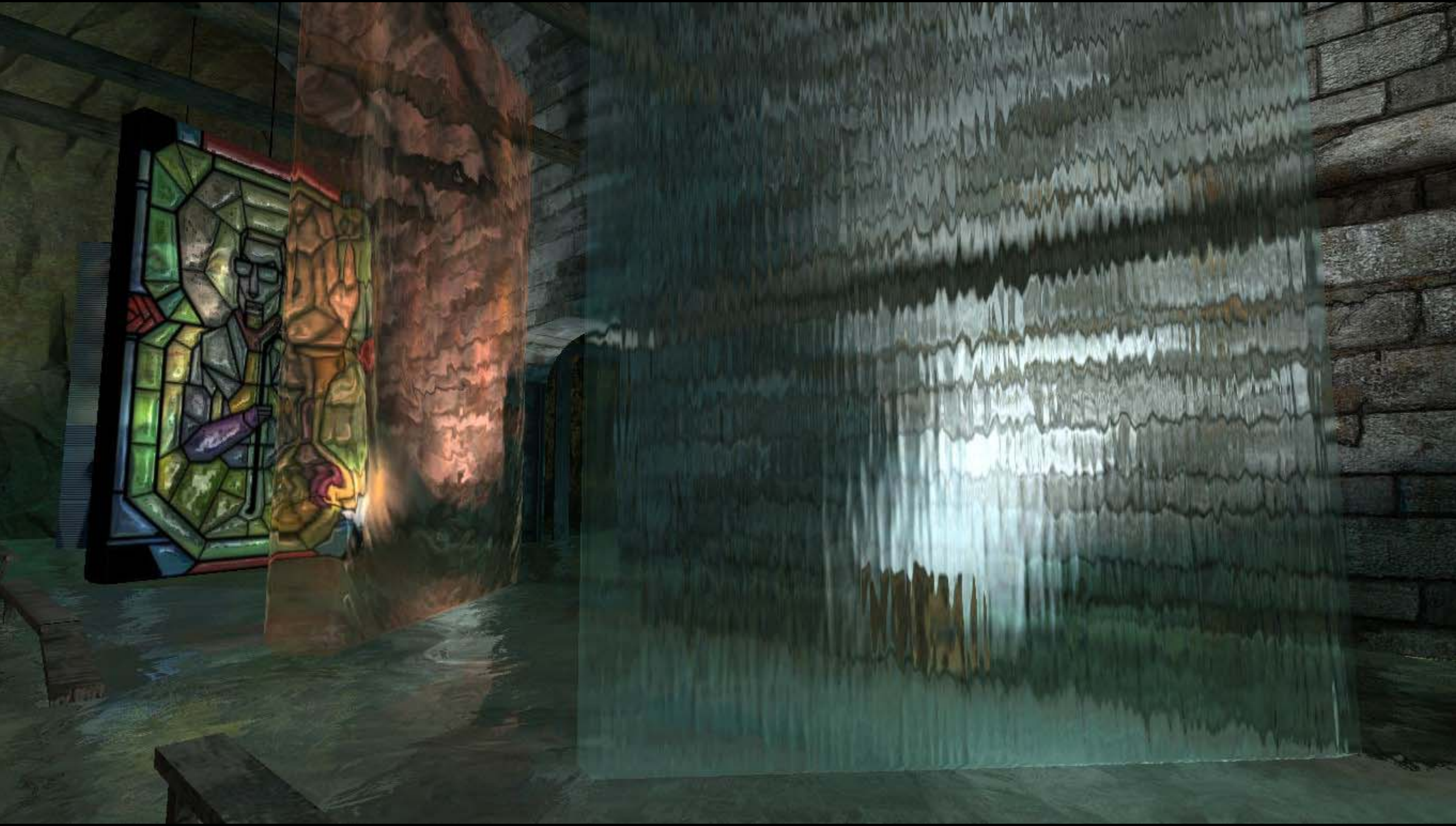
Bumped Model Pixel Shader

```
diffuseLighting =
  saturate( dot( tangentSpaceNormal, bumpBasis[0] ) ) * i.color1.rgb +
  saturate( dot( tangentSpaceNormal, bumpBasis[1] ) ) * i.color2.rgb +
  saturate( dot( tangentSpaceNormal, bumpBasis[2] ) ) * i.color3.rgb;
```

Refraction Mapping

- Refractive materials and water surfaces
- Render to multisample antialiased back buffer normally
- Use **`StretchRect()`** to copy to texture
- Project onto geometry, offsetting in screen space with either a dudv map, or a normal map

Refraction Mapping



Samplers and Constant Inputs

```
sampler RefractSampler              : register( s2 );
sampler NormalSampler               : register( s3 );
sampler RefractTintSampler          : register( s5 );

const float3 g_EnvmapTint           : register( c0 );
const float3 g_RefractTint          : register( c1 );
const float3 g_EnvmapContrast       : register( c2 );
const float3 g_EnvmapSaturation     : register( c3 );
const float2 g_RefractScale         : register( c5 );
```

Refraction Input

```
struct PS_INPUT
{
   float2 vBumpTexCoord                : TEXCOORD0;
   float3 vWorldVertToEyeVector        : TEXCOORD1;
   float3x3 tangentSpaceTranspose      : TEXCOORD2;
   float3 vRefractXYW                  : TEXCOORD5;
   float3 projNormal                   : TEXCOORD6;
};
```

Refraction Shader

```
float4 main( PS_INPUT i ) : COLOR
{
   // Load normal and expand range
   float4 vNormalSample = tex2D( NormalSampler, i.vBumpTexCoord );
   float3 tangentSpaceNormal = vNormalSample * 2.0 - 1.0;

   float3 refractTintColor = 2.0 * g_RefractTint * tex2D(
   RefractTintSampler, i.vBumpTexCoord );

   // Perform division by W only once
   float ooW = 1.0f / i.vRefractXYW.z;

   // Compute coordinates for sampling refraction
   float2 vRefractTexCoordNoWarp = i.vRefractXYW.xy * ooW;
   float2 vRefractTexCoord = tangentSpaceNormal.xy;
   float scale = vNormalSample.a * g_RefractScale;
   vRefractTexCoord = vRefractTexCoord * scale;
   vRefractTexCoord += vRefractTexCoordNoWarp;

   float3 result = refractTintColor * tex2D( RefractSampler,
                                             vRefractTexCoord.xy );

   return float4( result, vNormalSample.a );
}
```

Water

- Render to multisample antialiased back buffer normally for both refraction and reflection textures
- Use **`StretchRect()`** to copy to texture
- Render water surface

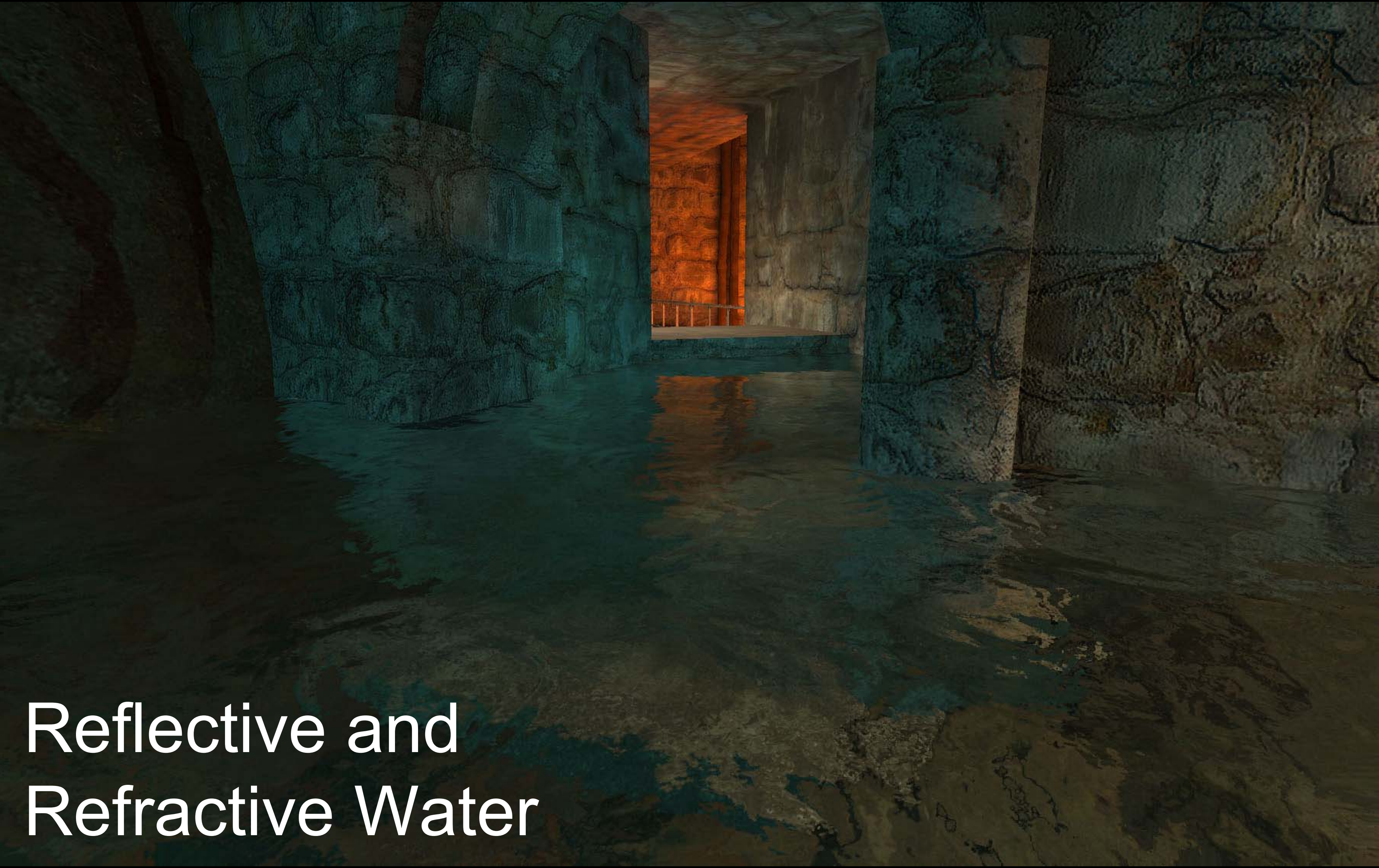
Water Samplers and Inputs

```
sampler RefractSampler    : register( s2 );
sampler ReflectSampler    : register( s4 );
sampler NormalSampler     : register( s3 );
sampler NormalizeSampler  : register( s6 );

const float4 vRefractTint : register( c1 );
const float4 vReflectTint : register( c4 );

// xy - reflect scale, zw - refract scale
const float4  g_ReflectRefractScale : register( c5 );

static const bool g_bReflect;
static const bool g_bRefract;

struct PS_INPUT
{
    float2 vBumpTexCoord         : TEXCOORD0;
    float3 vTangentEyeVect       : TEXCOORD1;
    float4 vReflectXY_vRefractYX : TEXCOORD2;
    float  W                     : TEXCOORD3;
};
```

Used during preprocess for code specialization

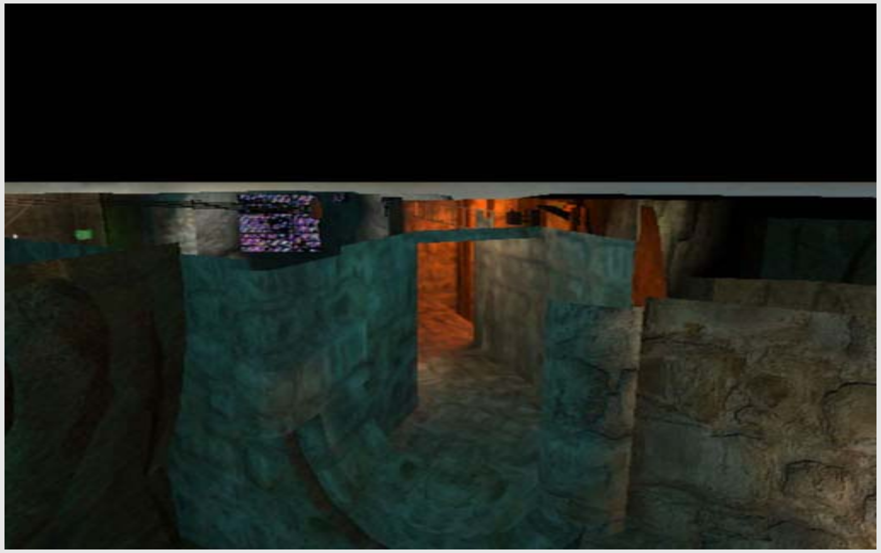
Water Shader

```
float4 main( PS_INPUT i ) : COLOR
{
   // Load normal and expand range
   float4 vNormalSample = tex2D( NormalSampler, i.vBumpTexCoord );
   float3 vNormal = vNormalSample * 2.0 - 1.0;

   float ooW = 1.0f / i.W;  // Perform division by W only once

   float2 vReflectTexCoord, vRefractTexCoord;

   float4 vN; // vectorize the dependent UV calculations (reflect = .xy, refract = .wz)
   vN.xy = vNormal.xy;
   vN.w = vNormal.x;
   vN.z = vNormal.y;
   float4 vDependentTexCoords = vN * vNormalSample.a * g_ReflectRefractScale;

   vDependentTexCoords += ( i.vReflectXY_vRefractYX * ooW );
   vReflectTexCoord = vDependentTexCoords.xy;
   vRefractTexCoord = vDependentTexCoords.wz;

   float4 vReflectColor = tex2D( ReflectSampler, vReflectTexCoord ) * vReflectTint; // Sample reflection
   float4 vRefractColor = tex2D( RefractSampler, vRefractTexCoord ) * vRefractTint; // and refraction

   float3 vEyeVect = texCUBE( NormalizeSampler, i.vTangentEyeVect ) * 2.0 - 1.0;

   float fNdotV = saturate( dot( vEyeVect, vNormal ) ); // Fresnel term
   float fFresnel = pow( 1.0 - fNdotV, 5 );

   if( g_bReflect && g_bRefract ) {
      return lerp( vRefractColor, vReflectColor, fFresnel );
   }
   else if( g_bReflect ) {
      return vReflectColor;
   } else if( g_bRefract ) {
      return vRefractColor;
   } else {
      return float4( 0.0f, 0.0f, 0.0f, 0.0f );
   }
}
```

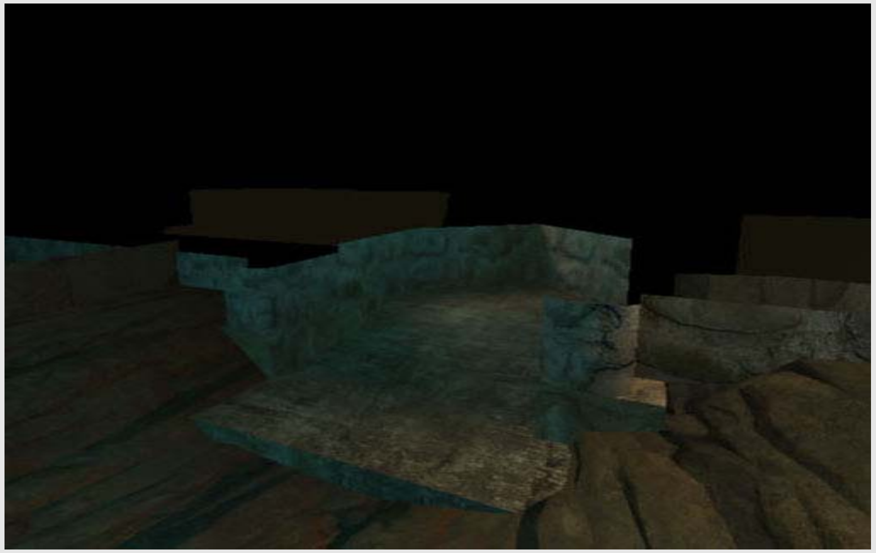


Reflective and Refractive Water

Reflection and Refraction Maps



Reflection Map



Refraction Map

Summary

- Radiosity Normal Mapping
- World Geometry pixel shading
  - Lighting equation
  - Managing shader permutations
- Model Geometry vertex shading
- Reflection and Refraction