Phat Lewt: Drawing a Diamond

David Gosselin
3D Application Research Group
ATI Research, Inc.

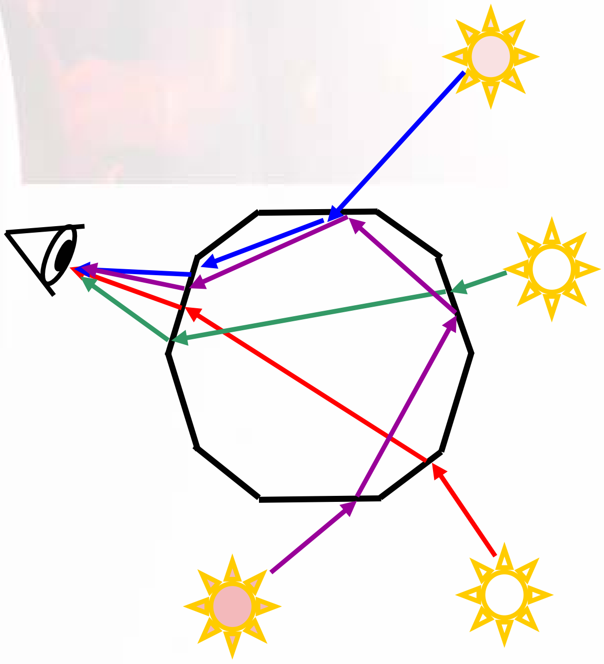
Overview

- Background
- Refractions
- Reflections
- Sparkles
- Demo

What Happens in the Real World

- Light from the environment can take multiple paths to get to the eye

- High index of refraction (IR) causes high visual complexity because light bounces due to total internal reflection



Basic Algorithm

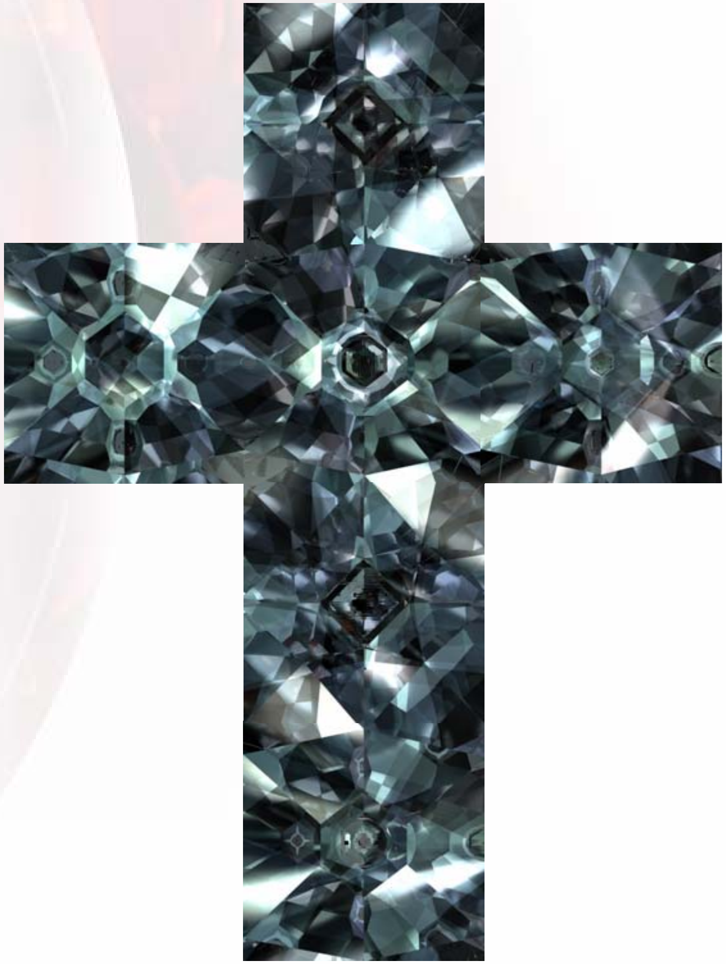
- Draw back face refractions to the back buffer
- Additively blend on top of back face refractions:
  - Front Face Refractions
  - Front Face Reflections (Environment Cube Map)
  - Front Face Specular Lighting
- Draw sparkles based on Illumination

Faking Refractions

- Look up into a refraction cubemap
- Use multiple refraction vectors
  - Straight up refraction vector
  - Refraction with different IR, then reflected by a vector random to each face
    - To prevent sampling close to first refraction ray
- Use multiple normals (lerp between smooth and face for more variation)
- Can also add an “edge” map to give even more hard edges (more visual complexity)

Creating a Refraction Cubemap

- Rendered with Maya
- Camera inside of gem looking out
- Lighting environment approximated by an environment map

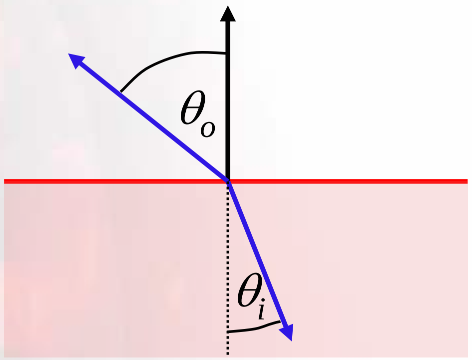


Computing Refraction Rays

- Derived from Snell's law:

$$\eta_{inside} \sin(\theta_{inside}) = \eta_{outside} \sin(\theta_{outside})$$

$$\theta_{inside} = \sin^{-1}\left(\frac{\eta_{outside}}{\eta_{inside}} \sin(\theta_{outside})\right)$$




```
float3 SiTransmissionDirection (float fromIR, float toIR,
                                float3 incoming, float3 normal)
{
   float eta = fromIR/toIR; // relative index of refraction
   float c1 = -dot(incoming, normal); // cos(theta1)
   float cs2 = 1.-eta*eta*(1.-c1*c1); // cos^2(theta2)
   float3 v = (eta*incoming + (eta*c1-sqrt(cs2))*normal);
   if (cs2 < 0.) v = 0; // total internal reflection
   return v;
}
```

Refractions

Into Back Buffer

( + ) *  = 

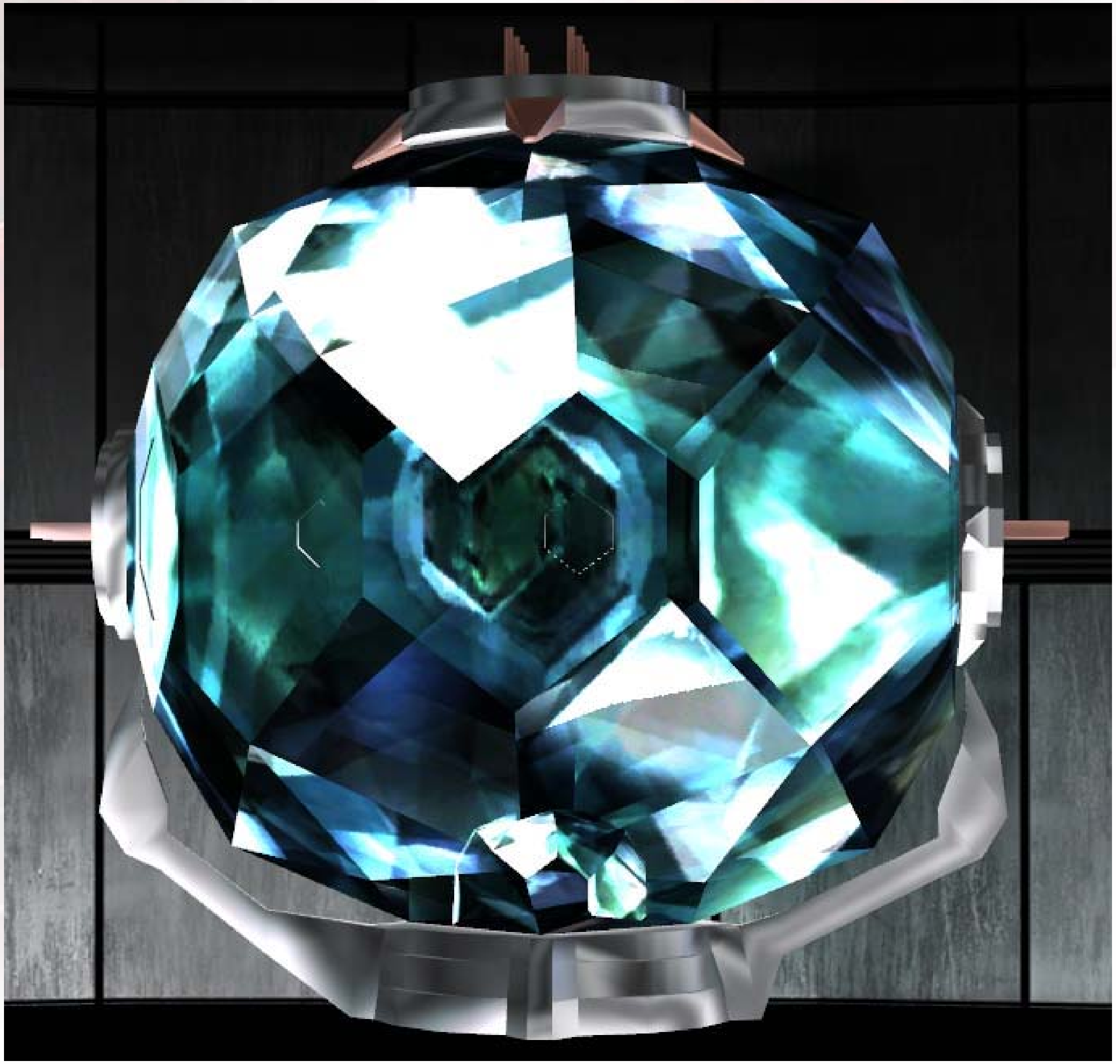
Back Face Refractions **Edge**

Additive Blend With Back Buffer **+**

( + ) *  = 

Front Face Refractions **Edge**

Combined Refractions



Vertex Shader

```
VsOutput main (VsInput i)
{
   // Matrix Skin position
   VsOutput o;
   float4x4 mSkinning = SiComputeSkinningMatrix (i.weights,
                                                  i.indices);
   float4 pos = mul (i.pos, mSkinning);
   o.pos = mul (pos, mVP);

   // Texture coordinates
   o.uv = i.uv;
   o.noiseUV = dot (i.normal, float3(1, 1, 1));

   // Compute normal and perturbed normal
   float3 faceNormal = mul (i.normal, mSkinning);
   float3 smoothNormal = mul (i.normal2, mSkinning);
   float3 mixedNormal = normalize (lerp (faceNormal,
                                          smoothNormal, 0.3));
   o.normal = faceNormal;
   o.normal2 = mixedNormal;

   // Compute Light and view vector
   . . .
```

Refraction Pixel Shader

```
float4 main (PsInput i) : COLOR
{
   // Normalize interpolated vectors
   float3 vNorm2 = normalize(i.normal2);
   float3 vView  = normalize(i.view);

   // Compute refraction vectors
   float3 vRefract = SiTransmissionDirection (1.0, 2.4,
                                              i.view, vNormal2);
   float3 vReflectRefract = SiTransmissionDirection (1.0, 1.8,
                                              i.view, vNormal2);

   // Reflect second vector by a vector random to each face
   float3 rnd = tex2D(tNoise, i.noiseUV);
   rnd = normalize (SiConvertColorToVector (rnd));
   vReflectRefract = SiReflect (vReflectRefract, rnd);

   .
   .
   .
```

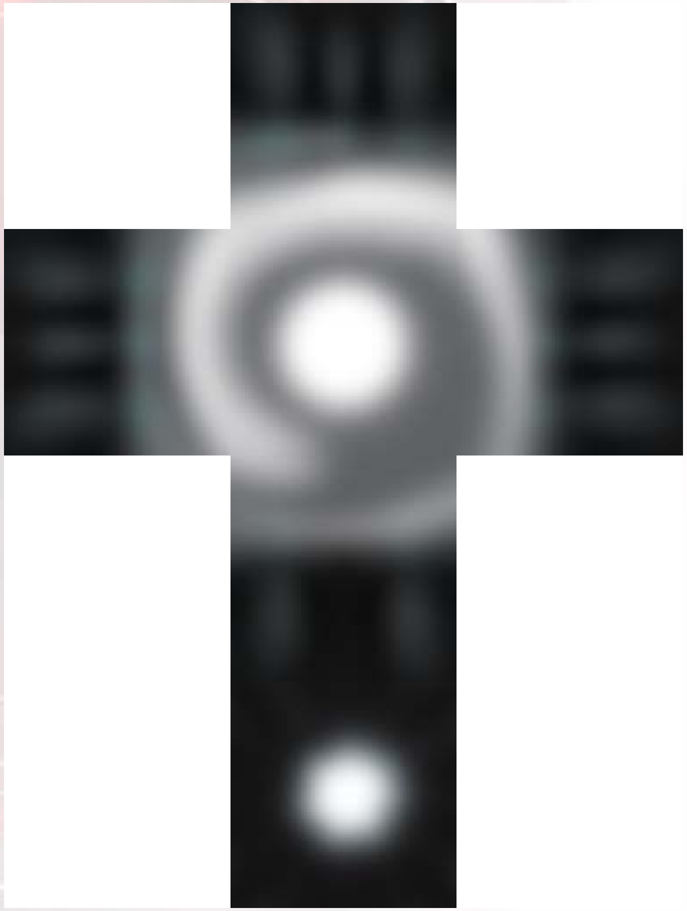
Refraction Pixel Shader

```
    .
    .
    .
    // Lookup into refraction cubemap and apply gamma
    float3 cRefract = texCUBE (tRefraction, vRefract);
    cRefract += texCUBE (tRefraction, vReflectRefract.yxz);
    cRefract = pow (cRefract, 4.0);

    // Edge term
    float3 edge = lerp (1.0, tex2D (tEdge, i.uv.xy), 0.4);

    // Final Output
    float4 o;
    o.rgb = cRefract * edge * 0.5; // 0.7 for Front Faces
    o.a = 0.0;
    return o;
}
```

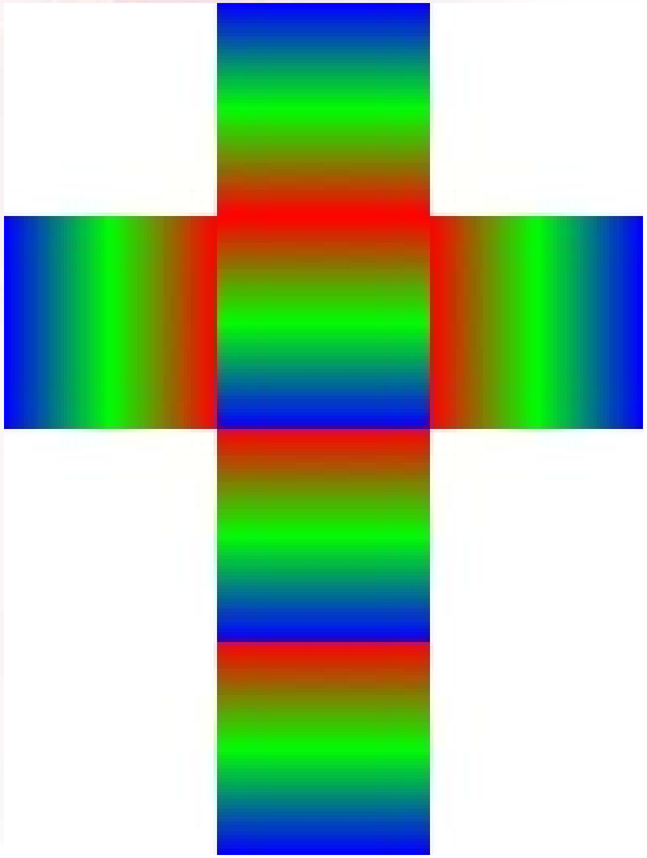
Reflections

- Reflection cube map lookup
- To fake dispersion:
  - “Rainbow” cubemap lookup
  - Modulate rainbow sample with reflection sample
- Lerp between modulated and original reflection sample to control dispersion strength
- Modulate with Fresnel term
- Add specular highlights

Cube Maps

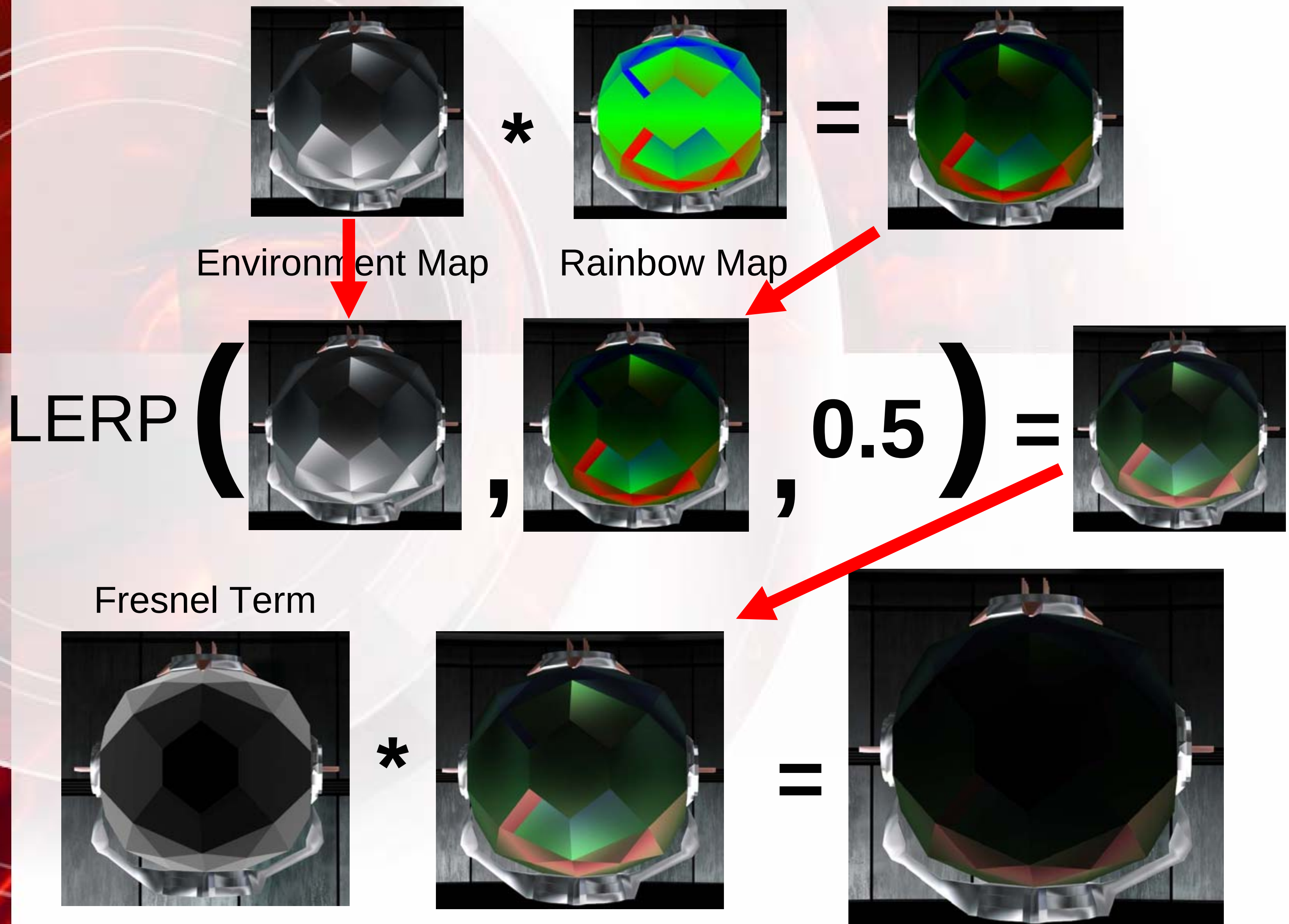


Blurred Environment Map

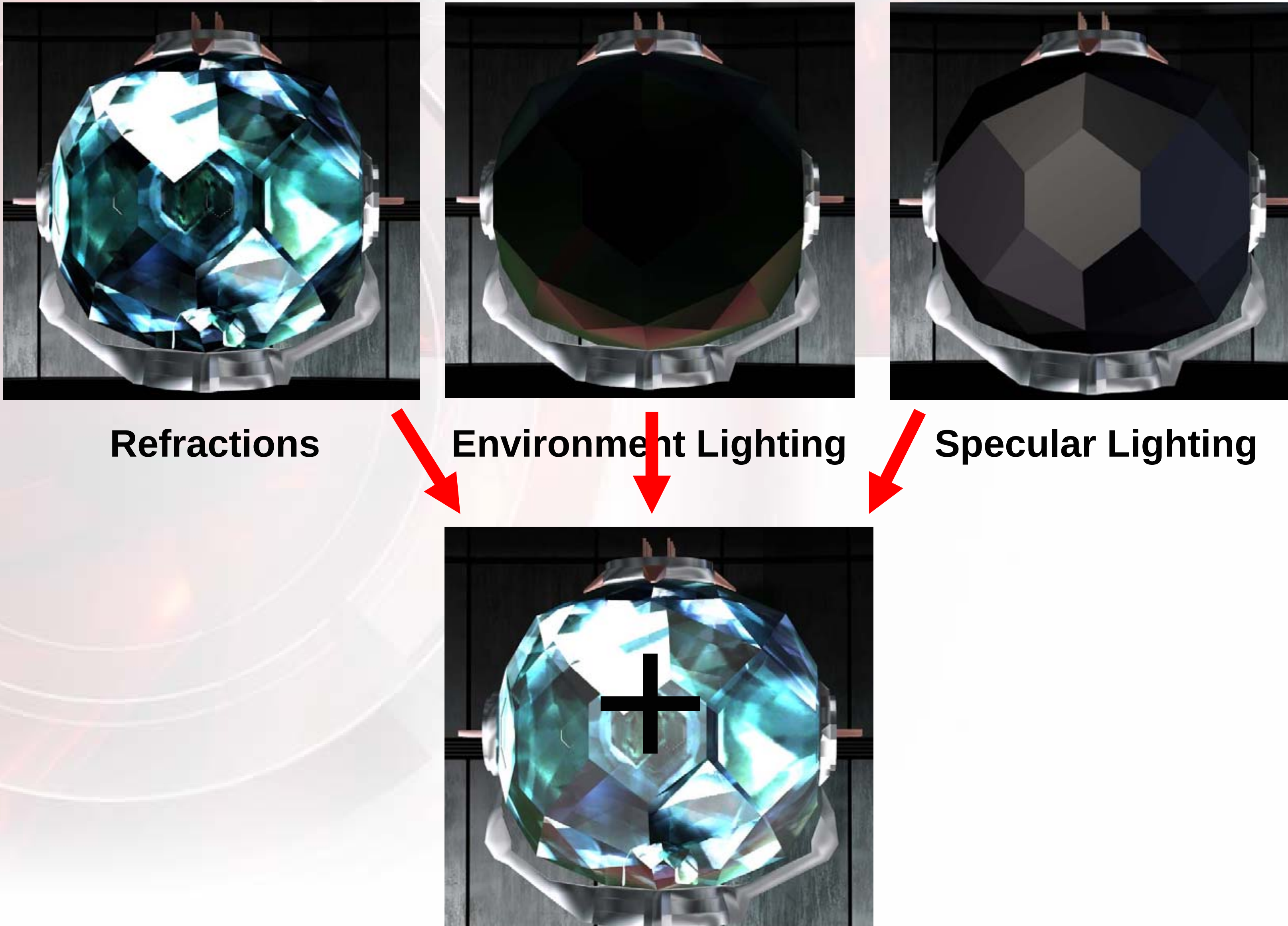


Rainbow Map

Environment Lighting



Final Look



Final Front Face Pixel Shader

```
float4 main (PsInput i) : COLOR
{
   // Compute refraction vectors as shown previously
   . . .

   // Specular Lighting
   float3 vReflection = SiReflect (vView, vNormal);
   float RdotL = saturate (dot (vReflection, i.lightVec0);
   float3 specular = (pow (RdotL, specPower) * lightColor0);
   RdotL = saturate (dot (vReflection, i.lightVec1);
   specular += (pow (RdotL, specPower) * lightColor1);
   RdotL = saturate (dot (vReflection, i.lightVec2);
   specular += (pow (RdotL, specPower) * lightColor2);

   // Look up environment map
   float3 vReflection2 = SiReflect (vView, vNormal2);
   float3 cEnv = texCUBE (tEnvironment, vReflection2);
   float3 cSpectral = texCUBE (tSpectral, vReflection2);

   .
   .
   .
```

Final Front Face Pixel Shader

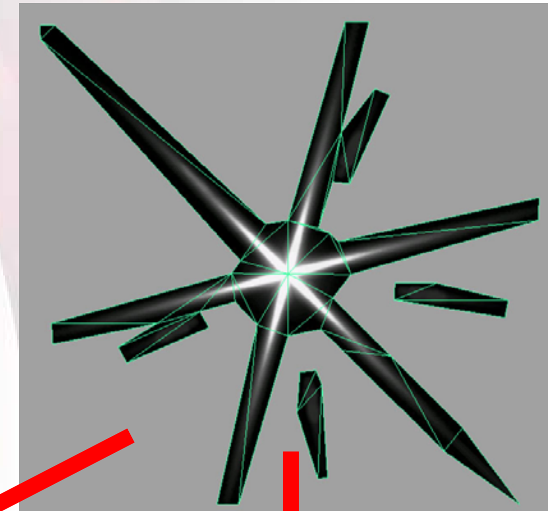
```
    .
    .
    .
    // Combine Environment and Rainbow (spectral) maps
    float fresnel = pow(1.0-saturate(dot (vNormal, vView), 2.0);
    cEnv = lerp (cEnv, cSpectral * cEnv, 0.5);
    cEnv = fresnel * cEnv;

    // Put it all together
    float4 o;
    // Refractions
    o.rgb = cRefract * edge * 0.7;

    // Environment lighting
    o.rgb += (cEnv*reflectionStrength*cReflectionColor);

    // Specular lighting
    o.rgb += saturate(specular)

    o.a = 1.0;
    return o;
}
```

Sparkles

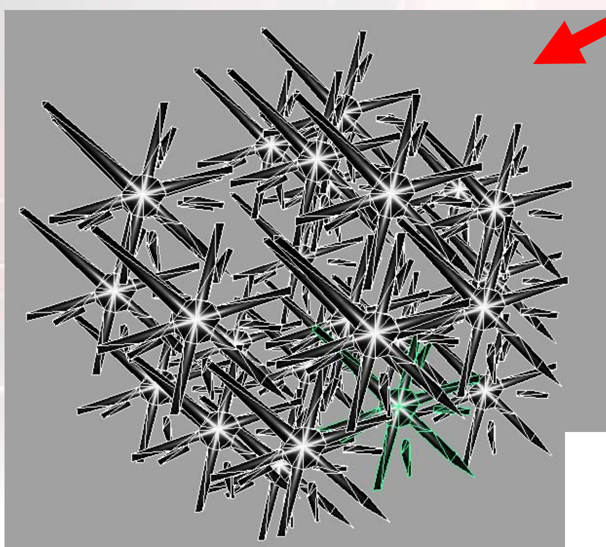
- Placed at strategic points on geometry
- Sparkles move rigidly with gem
- Expanded based on their texture coords
  - Screen-aligned
- Faded in based on an off-screen texture luminance at center of sparkle
- Modulate with a noise value to make them flicker a little bit

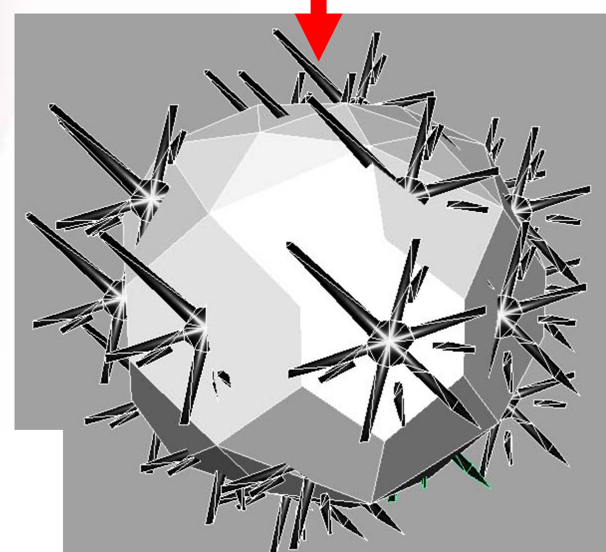
Flare Geometry

- Only center matters
- “Cloud” works well
- No need to reside only on faces, inside gem works too



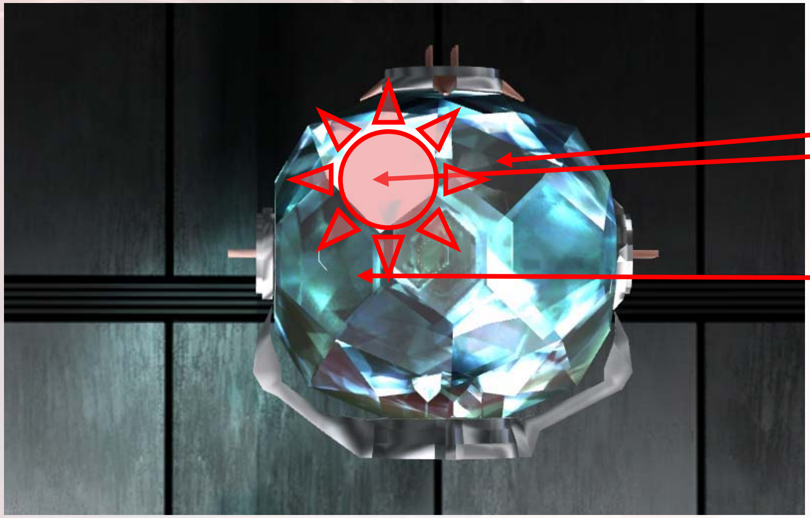
Single Flare



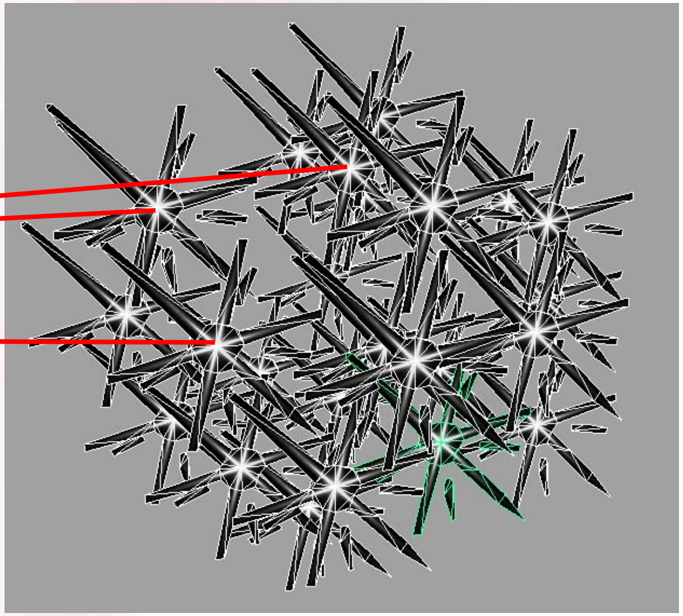


Flares Positioned

Conceptual Flare Process



Off-screen buffer



Flare Geometry

For each flare

- Look up luminance of its center in off-screen
- If luminance is > threshold
  - Draw (in reality don't kill)

Sparkle Vertex Shader

```
VsOutput main (VsInput i)
{
   // Skin center of flare
   VsOutput o;
   float4x4 mSkinning = SiComputeSkinningMatrix (i.weights,
                                                 i.indices);
   float4 pos = mul (float4(0,0,0,1), mSkinning);
   o.pos = mul (pos, mVP);

   // Figure out texture coordinates for off-screen texture
   o.screenUV = o.pos.xy/o.pos.w;
   o.screenUV.y = -o.screenUV.y;
   o.screenUV = 0.5 * o.screenUV + 0.5;

   // Scale flare in post transform space
   float fRadius = 10.0;
   o.pos.xy += fRadius * 2 * // Flare size
               (i.texCoord - float2(0.5, 0.5)) *
                mP._m00_m11; // Scale based on projection matrix
    .
    .
    .
```

Sparkle Vertex Shader

```
   .
   .
   .
   // Compute View vector
   float3 view = normalize (worldCamPos - pos);

   // Pass along texture coordinate
   o.texCoord = i.texCoord;

   // Compute texture coordinates for the noise map
   float rnd = dot (pos.xyz, float3(1, 1, 1));
   o.noiseUV.x = fmod (abs (rnd)), 2.0f);
   rnd = dot (view, float3(1, 1, 1));
   o.noiseUV.y = fmod (abs (2.0 * rnd)), 2.0f);
   return o;
}
```

Sparkle Pixel Shader

```
float4 main (PsInput i) : COLOR
{
  // Get noise value for flare intensity and size
  float noise = tex2D (tNoise, i.noiseUV);
  noise = lerp (0.6, 1.0, noise);

  // Get off-screen luminance at flare center
  float3 cScreen = tex2D (tScreen, i.screenUV);
  float lum = dot (cScreen, float3 (0.3, 0.59, 0.11));

  clip (lum - 0.8); // Kill pixels that are not bright enough

  // Compute the output color based on luminance
  lum = smoothstep (0.8, 1.0, lum);
  lum *= lum;

  // Compute final lighting
  float4 o;
  o.rgb = noise * lum;
  o.a = tex2D (tAlpha, i.texCoord);
  return o;
}
```

Demo