
# ADVANCED TOOLS FOR DEVELOPING HIGHLY REALISTIC COMPUTER GAMES

# STAND-ALONE COMPUTATION PACKAGE FOR ILLUMINATION ALGORITHMS

| | |
|---|---|
| Document identifier: | **GameTools-5-D5.3-03-1-1-Stand-Alone Computation Package for Illumination Algorithms** |
| Date: | **04/05/2006** |
| Work package: | **WP05: Illumination** |
| Partner(s): | **BUTE, UdG, Unilim** |
| Leading Partner: | **BUTE** |
| Document status: | **Final Version** |
| Deliverable identifier: | **D5.3** |

Abstract: This technical report describes the initially working modules of the illumination work package.

## Delivery Slip

| | Name | Partner | Date | Signature |
|---|---|---|---|---|
| **From** | Laszlo Szirmay-Kalos | BUTE | 03/05/06 | |
| **Reviewed by** | | | | |
| **Approved by** | | | | |

## Document Log

| Issue | Date | Comment | Author |
|---|---|---|---|
| 1.1 | 26-04-06 | Draft Version | Laszlo Szirmay Kalos |
| 2.0 | 02-05-06 | Reviewed Version | Laszlo Szirmay Kalos |
| | | | |
| | | | |

## Document Change Record

| Issue | Item | Reason for Change |
|---|---|---|
| | | |
| | | |
| | | |

## Files

| Software Products | User files / URL |
|---|---|
| Word | gametools-ist-2-004363-5-d5.3-03-2-0-stand-alone computation package for illumination.doc |

# CONTENT

# 1. INTRODUCTION

## 1.1. OBJECTIVES OF THIS DOCUMENT

This document describes the initially working modules for the Illumination Work Package. Its aim is to describe the modules and explain how they work.

## 1.2. DOCUMENT AMENDMENT PROCEDURE

Any project partner may request amendments but each amendment must be analysed and approved by the GameTools Project Coordinator or Project Manager.

## 1.3. TERMINOLOGY

### Glossary

| | |
|---|---|
| GUI | Graphics User Interface |
| GPU | Graphics Processing Units |
| GTP | GameTools Project |
| GPU | Graphics Processing Units |
| PC | Project Coordinator |
| PM | Project Manager |
| WP | Work Package |

# 2. DESCRIPTION OF THE MODULES

## 2.1. INDIRECT ILLUMINATION GATHERING

This is a standalone module that can be found in the *EnvMap* directory of the repository.

### 2.1.1. Role of the module

Final gathering, i.e. the computation of the reflection of the indirect illumination toward the eye, is one of the most time consuming steps of realistic rendering. In case of indirect illumination, the reflected radiance of point *x* can be expressed by an integral (*rendering equation*). The evaluation of this integral usually requires many sampling rays from point *x*. These sampling rays find illuminating points at different directions (Figure 1), and the radiance of these illumination points is inserted into a numerical quadrature approximating the rendering equation.

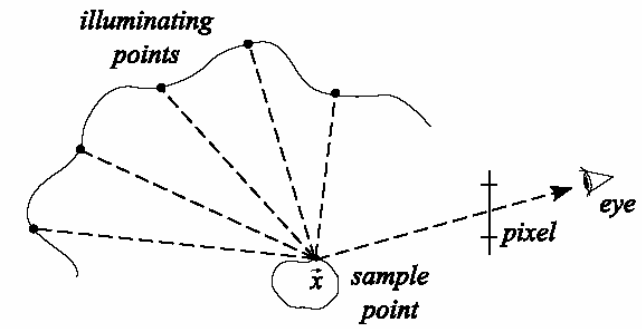



*Figure 1. Calculating indirect illumination*

The classical solution is to calculate a preconvolved cube map called *diffuse (or specular) environment map*. This means that the contribution of each illuminating point – weighted with the angular variation of the BRDF – is summed up in a preprocessing step. Then, during run-time we can determine the illumination of an arbitrarily oriented surface patch with a single environment map lookup toward the lookup direction associated to that surface patch. When the object to be shaded is moving, the diffuse/specular environment map must be regularly updated to reflect the changes.

Another approach is to carry out the convolution real-time, enabling us to *localize* the result depending on the position of the sample point. In case of dynamic objects, this localization allows us to decrease the frequency of the environment map updates while achieving the same quality. To maintain reasonable frame rates, the resolution of the environment map must be decreased before the actual convolution. This *downsampling* step corresponds to the clustering of small (texel) lights into larger (area) light sources (Figure 2).

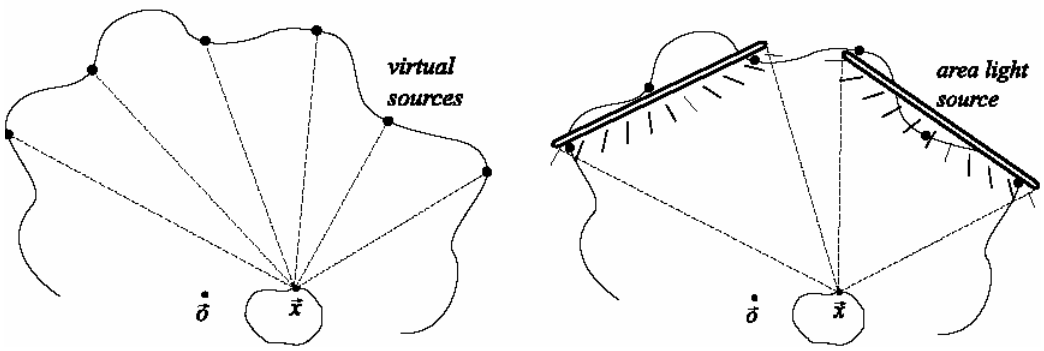



*Figure 2. Calculating indirect illumination*

### 2.1.2. Rendering modes

In the application, the following rendering modes are defined. The corresponding shader techniques are also listed.

| Rendering mode | Shader techniques |
|---|---|
| **IDEAL** | `EnvMapClassic(Metal)` (`EnvMapClassicMetal` for metals) |
| **IDEAL_LOCALIZED** | `EnvMapImpostor` (`EnvMapImpostorMetal` for metals) |
| **DIFFUSE_SPECULAR** | `EnvMapDiffuse` for rendering and `Convolution` for precalculation |
| **DIFFUSE_SPECULAR_LOCALIZED** | `EnvMapDiffuseLocalized`. |
| **DIFFUSE_SPECULAR_LOCALIZED_COSTEX** | `EnvMapDiffuseLocalizedWithCosLookup` for rendering and `EnvMap::GenerateCosTexture()`, `EnvMap::eval()` for precalculation |

#### *2.1.2.1. Ideal reflections / refractions with classic environment mapping*

Rendering mode: **IDEAL**

This program simply determines the ideal reflection/refraction direction and performs a cube map lookup into that direction.

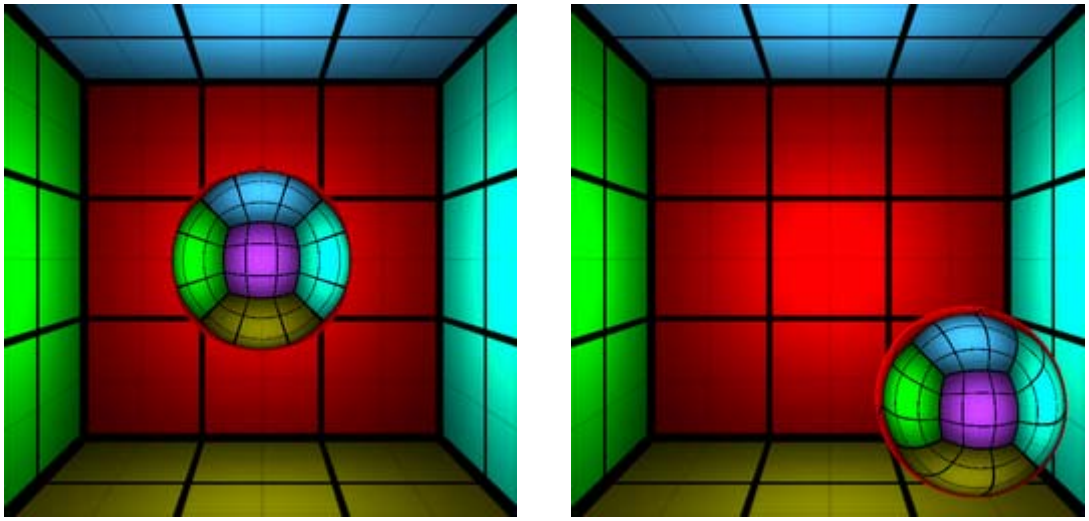


*Figure 3. Classic environment mapping with fixed reference point (i.e. the center of the box)*

If the object is moving, the cube map must be regularly updated. An update requires rendering the environment six times so it will affect performance (e.g. frame rate drops from 200 FPS to 75 FPS).

Note: To update the cube map in each frame, press 'A'. To update the cube map once, press space.

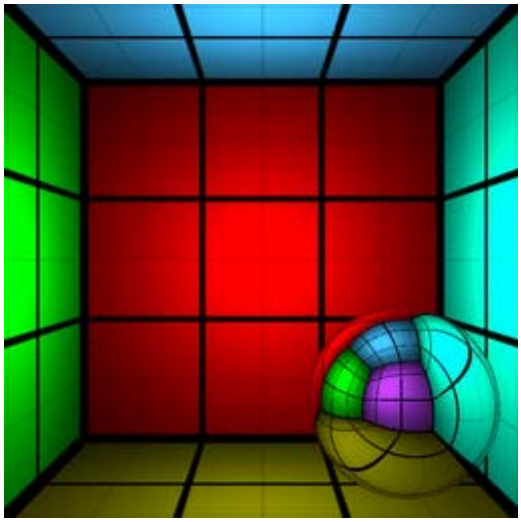


*Figure 4. Classic environment mapping with updated environment maps. Reference point is the center of the moving object.*

#### *2.1.2.2. Ideal reflections / refractions localized with distance impostors*

Rendering mode: **IDEAL_LOCALIZED**

After determining the ideal reflection/refraction direction, a ray is traced approximately toward that direction using the *Hit()* shader function. Note that even if we do not update the cube map at all, we

get results very similar to the classic case with frequent updates. This increases performance (150-200 FPS) and allows us to decrease the frequency of cube map updates in the general case.

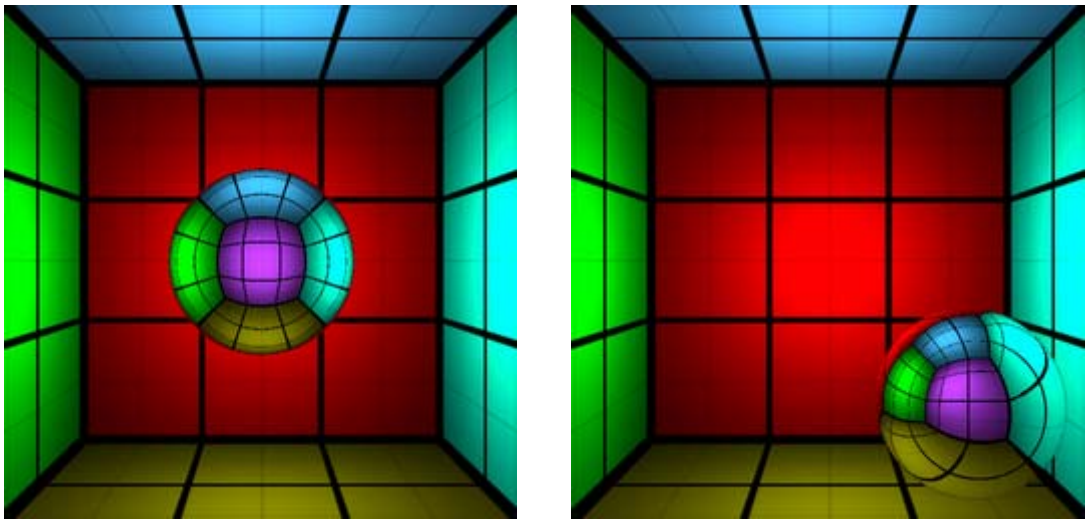


*Figure 5. Localized environment mapping with distance impostors. No cube map update is performed during the object movement.*

#### *2.1.2.3. Diffuse reflections with classic (preconvolved) diffuse environment mapping*

Rendering mode: **DIFFUSE_SPECULAR**

Using the environment map generated off-line, we can determine diffuse illumination with a single texture lookup during run-time (150-200 FPS).

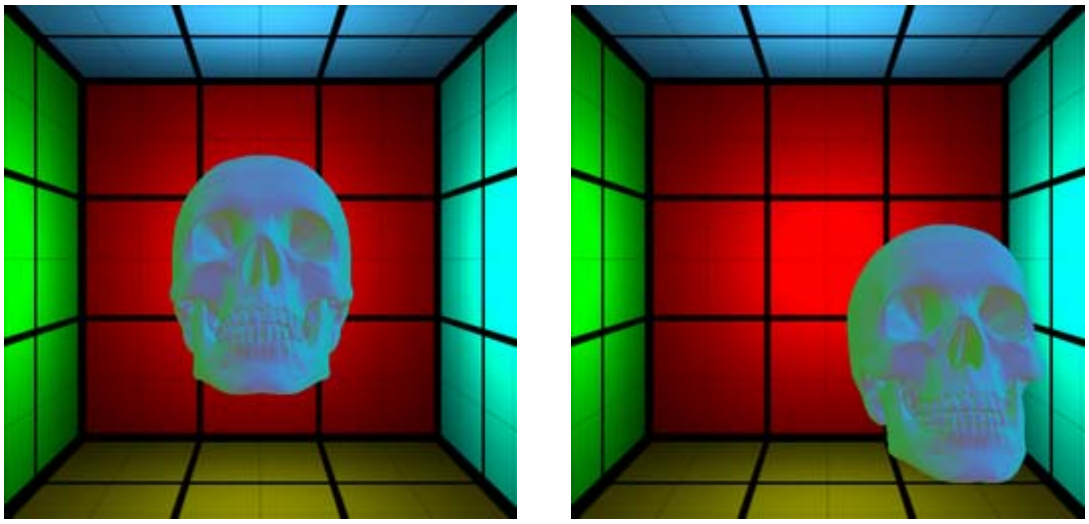


*Figure 6. Diffuse reflections with preconvolved diffuse maps. No localization effects occur.*

If the object is moving, the cube map must be regularly re-convolved that will affect performance (e.g. frame rate drops to 15-20 FPS).

Note. To update cube maps in each frame, press ‘A’. To update cube maps once, press space.

*Figure 7. Diffuse reflections with preconvolved diffuse maps that are regularly updated. The reference point is the center of the moving object.*

#### *2.1.2.4. Diffuse, localized reflections*

Rendering mode: **DIFFUSE_SPECULAR_LOCALIZED**

Diffuse reflections are obtained with on-the fly convolution. The reflectivity integral is approximated with only one sample considering the center of the texel. This can cause problems when the object is close to that texel (10-15 FPS).

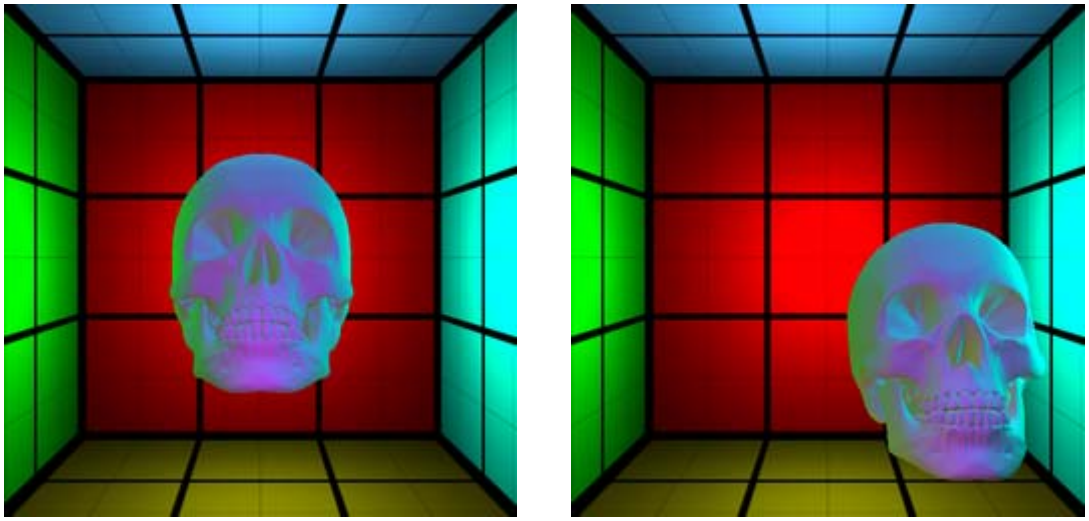


*Figure 8. Diffuse reflections with convolution on the fly. No cube map update is performed during object movement.*

### *2.1.2.5. Diffuse, localized reflections using precomputed geometry factors*

Rendering mode: **DIFFUSE_SPECULAR_LOCALIZED_COSTEX**

Diffuse reflections with convolution on-the fly (10-15 FPS). To correctly represent the reflectivity of a texel, the reflectivity integral is precalculated.

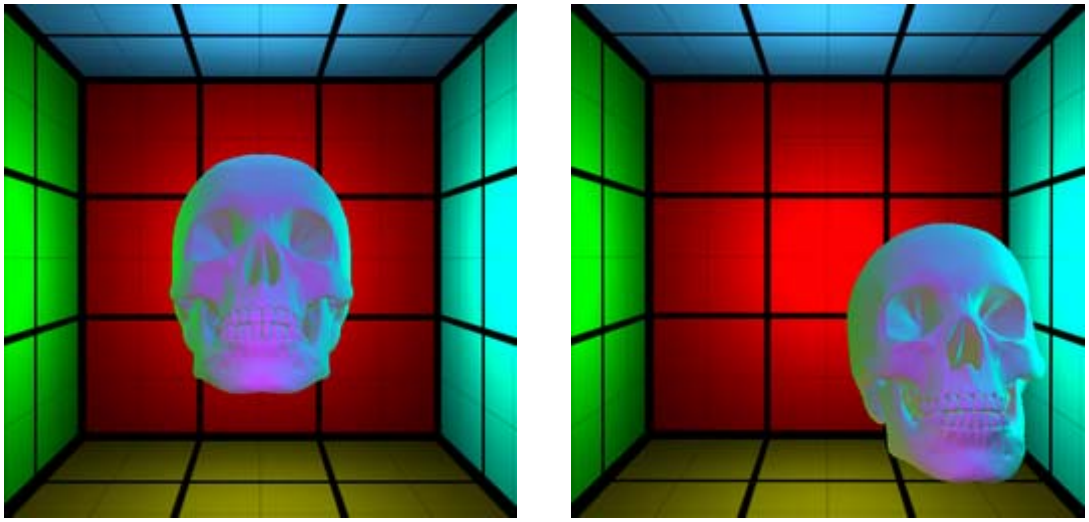


*Figure 9. Diffuse reflections with convolution on the fly. Reflectivity integral is precalculated. No cube map update is performed during object movement.*

### *2.1.2.6. Glossy reflections with classic (preconvolved) specular environment mapping*

Rendering mode: **DIFFUSE_SPECULAR**

Using the environment map generated off-line, we can determine specular illumination with a single texture lookup during run-time.

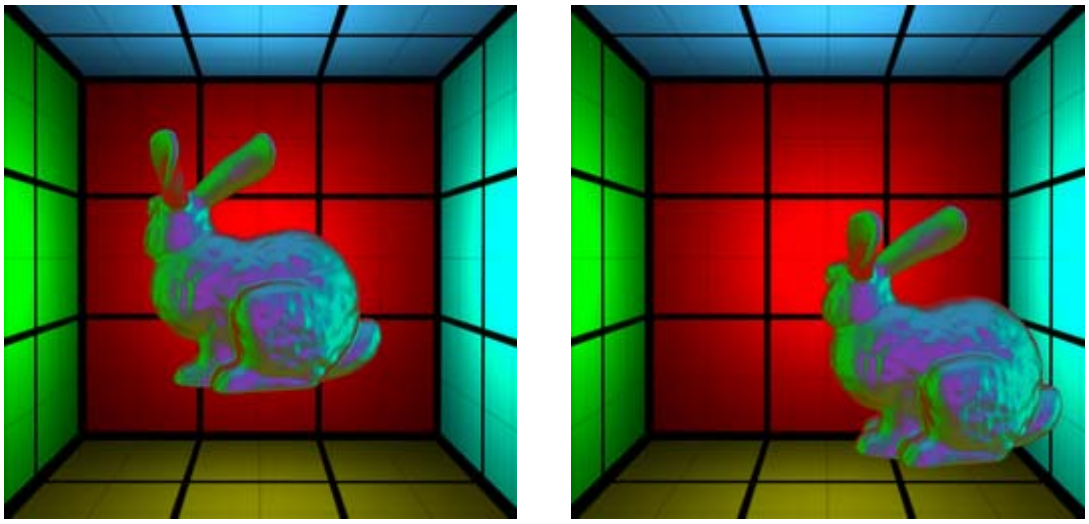


*Figure 10. Glossy reflections with preconvolved specular maps. No localization effects occur. (shininess = 6)*

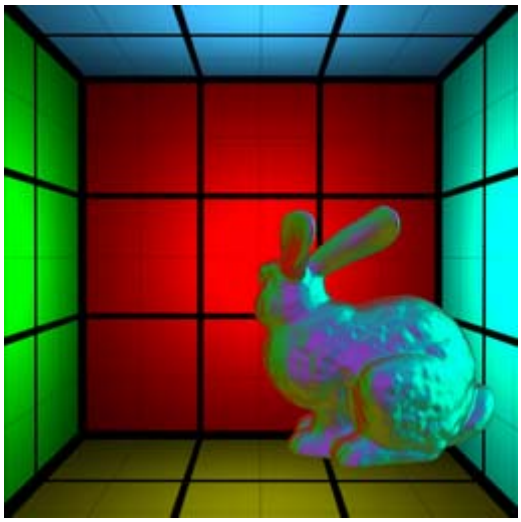


*Figure 11. Glossy reflections with preconvolved specular maps that are regularly updated. Reference point is the center of the moving object. (shininess = 6)*

#### *2.1.2.7. Glossy, localized reflections with convolution on-the fly*

Rendering mode: **DIFFUSE_SPECULAR_LOCALIZED**

The reflectivity integral is approximated with only one sample considering the center of the texel. This can cause problems when the object is close to that texel.

### *2.1.2.8. Glossy, localized reflections using precomputed geometry factors*

Rendering mode: **DIFFUSE_SPECULAR_LOCALIZED_COSTEX**

To correctly represent the reflectivity of a texel, the reflectivity integral is precalculated.

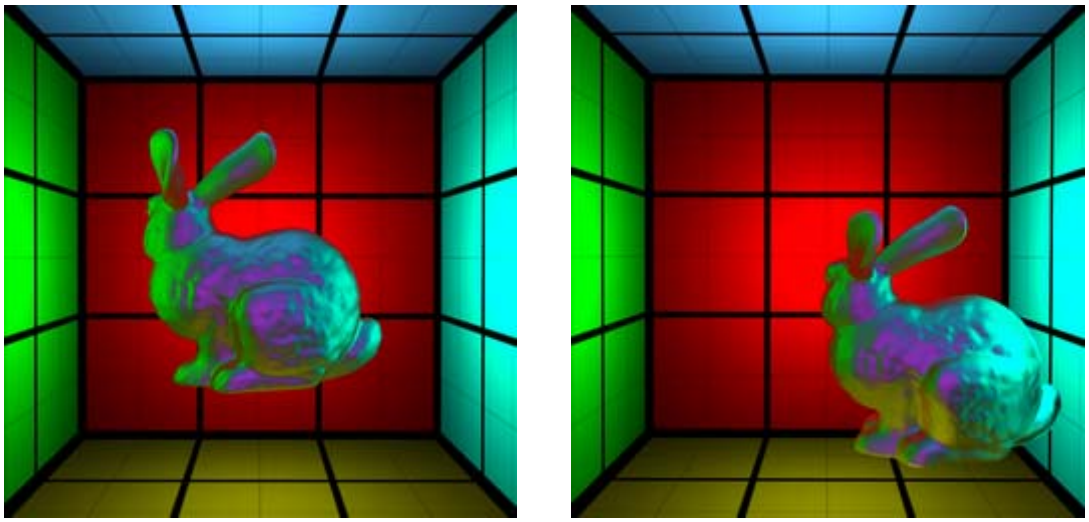


*Figure 12. Glossy reflections with convolution on the fly. Reflectivity integral is precalculated. No cube map update is performed during object movement. (shininess = 6)*

### *2.1.2.9. Realistic metals*

Using a simple approximation to the Fresnel term, we can display realistic metals. Four different metals are implemented by supporting their refraction indices:

**copper** **gold** **silver** **aluminum**

*Figure 13. Metal shaders*

### 2.1.3. Program structure

#### *2.1.3.1. Program resources*

**class Main**

Main.cpp contains global callback functions that are responsible for DirectX initialization. Also supports screenshot capturing (press '**S**').

**class Parameters**

This is a utility class to manage application-specific parameters. Using this class, you can easily add numeric/boolean parameters to your application. These parameters are represented by sliders or checkboxes/radio buttons on the graphical user interface. You can easily get the current values of these parameters when needed. All parameters can be saved to file or read from file in one step.

The following application-specific parameters are defined here:

Boolean parameters*:* `bShowHelp, bCubeMapFromFile, bAutoGenCubeMap.`

These values are defined in the `bool_t` enumeration. Thus, they are just constant identifiers and their values cannot be read directly. Use function `Get(bool_t i)` instead, e.g. `Get(bShowHelp)`.

Numerical parameters: `iWhichMethod, iWhichMetal, iShowCubeMap, sFresnel, refractionIndex, fIntensity, iShininess.`

These values are defined in the `number_t` enumeration. Thus, they are just constant identifiers and their values cannot be read directly. Use function `GetInt(number_t i)` to get the parameter value as an integer, or use `Get(number_t i)` to get the parameter value as a float in range 0..1.

The available rendering modes are defined in the following enumeration. Numerical parameter `iWhichMethod` will take one of these values.

```
enum method_t {
      IDEAL,
      IDEAL_LOCALIZED,
      DIFFUSE_SPECULAR,
      DIFFUSE_SPECULAR_LOCALIZED,
      DIFFUSE_SPECULAR_LOCALIZED_COSTEX
}
```

**class Mesh**

This is a utility class to manage a mesh object. During loading (`Load(fileName)`), it removes mesh offset (i.e. the bounding box of the object will be centered to the origin) and sets the specified object diameter (`preferredDiameter`). Object position can be modified via method `Move(D3DXVECTOR3 movement)`.

`SetContainerSize (D3DXVECTOR3 size)` allows us to specify the room environment where the object can move freely, also considering the extent of the object, not only its center. If the object is too close to the wall, further movements are disabled.

**class Cube**

This is a utility class to represent a cubic mesh. It can be more easily textured with procedurally generated textures than an .x file.

**texture pRoomTexture**

simple grid texture with thin and thick lines, used as a base for room texture generation (intensity falloff and color will be added)

**texture pCubeTexture**

cubemap that stores the environment as seen from the reference point

**texture pCubeTextureFromFile**

cubemap that stores a HDRI environment loaded from file

**texture pCubeTextureSmall**

downsampled (4x4) version of one of the previous cubemaps

**texture pCubeTexturePreConvolved**

preconvolved (4x4) cube map

*Figure 14. Basic room texture (left) and HDRI environment (uffizi.dds, right)*

### *2.1.3.2. Generating an environment map*

**void EnvMap::RenderCubeMap (IDirect3DCubeTexture9 *pCubeTexture)**

renders the environment into the specified cubemap. The camera is placed into `reference_pos`, and uses technique `IlluminatedScene` for rendering.

**Technique IlluminatedScene**

renders the cubic room with a simple shading effect (coloring and exponential falloff).

### *2.1.3.3. Downsampling an environment map*

**void EnvMap::ReduceCubeMapSize( IDirect3DCubeTexture9*, IDirect3DCubeTexture9*)**

Reduces the resolution of a cube map by averaging multiple texels. Calls technique `ReduceTexture` for rendering.

**Technique ReduceTexture**

downsamples the `nFace`-th face of a cube map from resolution `CUBEMAP_SIZE` to `LR_CUBEMAP_SIZE` by averaging the corresponding texel values. `nFace` uniform parameter identifies the current face (0...5).

### *2.1.3.4. Pre-calculating the reflectivity*

**void EnvMap::GenerateCosTextureIfNotFound()**

calls `GenerateCosTexture`() if the specified texture is not yet calculated and stored as a texture in the **cos** subdirectory.

**IDirect3DTexture9* EnvMap::GenerateCosTexture()**

generates precomputed texture to store values for the reflectivity integral. It uses the `eval`() function.

**float EnvMap::eval (float cos_theta, float dw)**

Calculates the reflectivity integral for a given texel. The angle between the surface normal and texel center is described by cos_theta and the solid angle occupied by the texel is denoted by dw.

Instead of evaluating the reflectivity integral with only one sample belonging to the texel center (that would give us a result of cos_theta x dw), we use *2M x 2M x M* regularly spaced samples, and discard those that lie outside the unit hemisphere. The remaining samples are regularly distributed over the hemisphere.

For each sample, we check if it lies inside the cone of the specified solid angle. If yes, its contribution is considered.

### *2.1.3.5. Ideal reflection/refraction*

**Technique EnvMapClassic**

Simply determines the ideal reflection/refraction direction and performs a cube map lookup into that direction. Uniform parameter `EnvironmentMap` is bound to `EnvMap::pCubeTexture` (cube map of resolution `CUBEMAP_SIZE`).

```
float4 EnvMapClassicPS( EnvMapVS_output IN ) : COLOR
{
        IN.View = normalize( IN.View );
        IN.Normal = normalize( IN.Normal );

        float3 R = reflect(IN.View, IN.Normal);
        float3 T = refract(IN.View, IN.Normal, refractionIndex);
        float3 p0 = IN.Position;

        // reading from the cubemap
        float4 reflectcolor = texCUBE(EnvironmentMapSampler, R);
        float4 refractcolor = texCUBE(EnvironmentMapSampler, T);
        float cos_theta = -dot( IN.View, IN.Normal );              // Fresnel approximation
        float F = (sFresnel + pow(1-cos_theta, 5.0f) * (1-sFresnel));
        return intensity * (F * reflectcolor + (1-F) * refractcolor);
}
```

### *2.1.3.6. Ideal reflection for metals*

**Technique EnvMapClassicMetal**

The only difference from the previous shader is that no refraction is allowed and the reflectivity is calculated from the wavelength-dependent, complex refraction index (see function `metal_reflectivity`).

### *2.1.3.7. Ideal reflection/refraction with localization*

**Technique EnvMapImpostor**

Determines the ideal reflection/refraction direction and approximately traces a ray toward that direction using the `Hit()` function.

Uniform parameter `EnvironmentMap` is bound to `EnvMap::pCubeTexture` (cube map of resolution `CUBEMAP_SIZE`).

```
float4 EnvMapImpostorPS( EnvMapVS_output IN ) : COLOR
{
        IN.View = normalize( IN.View );
        IN.Normal = normalize( IN.Normal );
        float3 R = reflect(IN.View, IN.Normal);            // reflection direction
        float3 T = refract(IN.View, IN.Normal, refractionIndex);

        // translate reference point to the origin
        float3 p0 = IN.Position - reference_pos.xyz;
        float3 RR = 0, TT = 0;

        // -------------------------- approximate raytracing ----------------------
        // using depth impostors + interpolation
        RR = Hit(p0, R, EnvironmentMapSampler);

        // single refraction
        TT = Hit(p0, T, EnvironmentMapSampler);

        // reading from the cubemap
        float4 reflectcolor = texCUBE(EnvironmentMapSampler, RR);
        float4 refractcolor = texCUBE(EnvironmentMapSampler, TT);
        float cos_theta = -dot( IN.View, IN.Normal );              // Fresnel approximation
        float F = (sFresnel + pow(1-cos_theta, 5.0f) * (1-sFresnel));
        return intensity * (F * reflectcolor + (1-F) * refractcolor);
}
```

Function `Hit()` approximately traces a ray from point `x` towards direction `R`. Depth information is obtained from the alpha channel of `mp`. The function returns the approximate hit point.

```
float3 Hit(float3 x, float3 R, sampler mp)
{
        float rl = texCUBE(mp, R).a; // |r|
        float dp = rl - dot(x, R);           // parallax
        float3 p = x + R * dp;
        return p;
}
```

### *2.1.3.8. Classic diffuse/glossy illumination (off-line convolution)*

**void EnvMap::PreConvolve ( IDirect3DCubeTexture9*, IDirect3DCubeTexture9* )**

convolves data from the source cubemap and returns the result in the target cubemap. It uses technique `Convolution`. The result will be stored in `EnvMap::pCubeTexturePreConvolved`.

**Technique Convolution**

Calculates the diffuse/specular irradiance map of resolution `LR_CUBEMAP_SIZE` (= 4) by summing up the contributions of all cube map texels with regard to the current query direction.

Uniform parameter `SmallEnvironmentMap` is bound to `EnvMap::pCubeTextureSmall` (cube map of resolution `LR_CUBEMAP_SIZE`).

```
float4 ConvolutionPS(ConvolutionVS_output IN) : COLOR
{
    // input position = query direction for the result
    float3 q = normalize( IN.Position );
    float4 color = 0;

    for (int i = 0; i < LR_CUBEMAP_SIZE; i++)
       for (int j = 0; j < LR_CUBEMAP_SIZE; j++) {
              float u = (i+0.5) / (float)LR_CUBEMAP_SIZE;
              float v = (j+0.5) / (float)LR_CUBEMAP_SIZE;
              float3 pos = float3( 2*u-1, 1-2*v, 1 );
              float r = length(pos);
              pos /= r;
              float4 dcolor = 0;
              float3 L;
              L = float3(pos.z, pos.y, -pos.x);  dcolor += GetContibution( q, L );
              L = float3(-pos.z, pos.y, pos.x);  dcolor += GetContibution( q, L );
              L = float3(pos.x, pos.z, -pos.y);  dcolor += GetContibution( q, L );
              L = float3(pos.x, -pos.z, pos.y);  dcolor += GetContibution( q, L );
              L = float3(pos.x, pos.y,  pos.z);  dcolor += GetContibution( q, L );
              L = float3(-pos.x, pos.y, -pos.z); dcolor += GetContibution( q, L );
              float dw = 4 / (r*r*r);              // using accurate solid angles
              color += dcolor*dw;
       }
    return color / (LR_CUBEMAP_SIZE * LR_CUBEMAP_SIZE);
}
```

**Technique EnvMapDiffuse**

determines diffuse or specular illumination with a single lookup into `PreconvolvedEnvironmentMap`.

Parameters: `PreconvolvedEnvironmentMap` is bound to `EnvMap::pCubeTexturePreConvolved` (cube map of resolution `LR_CUBEMAP_SIZE`).

```
float4 EnvMapDiffusePS( EnvMapVS_output IN ) : COLOR
{
	IN.View = normalize( IN.View );
	IN.Normal = normalize( IN.Normal );
	float3 R = reflect(IN.View, IN.Normal);
	if (shininess <= 0)    // diffuse
		return intensity * texCUBE(PreconvolvedEnvironmentMapSampler, IN.Normal);
	else                   // specular
		return intensity * texCUBE(PreconvolvedEnvironmentMapSampler, R);
}
```

### 2.1.3.9. Localized diffuse/glossy illumination (real-time convolution)

**Technique EnvMapDiffuseLocalized**

calculates diffuse or specular contributions of all texels in `SmallEnvironmentMap` to the current point. For each texel of `SmallEnvironmentMap`, function `GetContibution()` is called. Uniform parameter `SmallEnvironmentMap` is bound to `EnvMap::pCubeTextureSmall`.

```
float4 EnvMapDiffuseLocalizedPS( EnvMapVS_output IN ) : COLOR
{
	IN.View = -normalize( IN.View );
	IN.Normal = normalize( IN.Normal );

	// translate reference point to the origin
	IN.Position -= reference_pos.xyz;
	float3 R = -reflect( IN.View, IN.Normal );          // reflection direction
	float4 I = 0;

	for (int x = 0; x < LR_CUBEMAP_SIZE; x++)           // foreach texel
	 for (int y = 0; y < LR_CUBEMAP_SIZE; y++) {
		// compute intensity for 6 texels with equal solid angles
		float2 tpos;
		tpos.x = x/(float)LR_CUBEMAP_SIZE;          // 0..1
		tpos.y = y/(float)LR_CUBEMAP_SIZE;          // 0..1
		// offset to texel center
		tpos.xy += float2(0.5/LR_CUBEMAP_SIZE, 0.5/LR_CUBEMAP_SIZE);
		float2 p = float2(tpos.x, 1-tpos.y);        // reverse y
		p.xy = 2*p.xy - 1;                                  // -1..1

		float3 L;
		L = float3(p.x, p.y, 1);
		I += GetContibution( L, IN.Position, IN.Normal, IN.View );
		L = float3(p.x, p.y, -1);
		I += GetContibution( L, IN.Position, IN.Normal, IN.View );
		L = float3(p.x, 1, p.y);
		I += GetContibution( L, IN.Position, IN.Normal, IN.View );
		L = float3(p.x, -1, p.y);
		I += GetContibution( L, IN.Position, IN.Normal, IN.View );
		L = float3(1, p.x, p.y);
		I += GetContibution( L, IN.Position, IN.Normal, IN.View );
		L = float3(-1, p.x, p.y);
		I += GetContibution( L, IN.Position, IN.Normal, IN.View );
	}
	return I;
}
```

### Function GetContibution

Calculates the contribution of a single texel of `SmallEnvironmentMap` to the illumination of the shaded point. Parameter `L` vector pointing to the center of the texel under examination. We assume that the largest coordinate component of `L` is equal to one, i.e. `L` points to the face of a cube of edge length of 2. Parameter pos is the position of the shaded point, `N` is the surface normal and `V` is the viewing direction at the shaded point, respectively.

```
float4 GetContibution(float3 L, float3 pos, float3 N, float3 V)
{
	float l = length(L);
	L = normalize(L);
	float4 Lin = texCUBE(SmallEnvironmentMapSampler, L);
	float dw = 4 / (LR_CUBEMAP_SIZE*LR_CUBEMAP_SIZE*l*l*l + 4/2/3.1416f);
	float dws = dw;

	//r
	float doy = texCUBE(SmallEnvironmentMapSampler, L).a;
	float dxy = length(pos - L * doy);

	// dws localization
	dws = (doy*doy * dw) / (dxy*dxy*(1 - dw/2/3.1416f) + doy*doy*dw/2/3.1416f);

	// L should start from the object (and not from the reference point)
	L = L * doy - pos;
	L = normalize(L);

	float3 H = normalize(L + V);
	float4 color = 0;
	float a = 0;
	if ( shininess <= 0 )
		a = kd * max(dot(N,L),0);
	else
		a = ks * pow(max(dot(N,H),0), shininess) * (shininess+2)/(2*PI);
	return Lin * a * dws;
}
```

In case of precalculated reflectivity, the only difference is an additional texture lookup at the end of the function:

```
float4 GetContibutionWithCosLookup(float3 L, float3 pos, float3 N, float3 V)
{
	.............
	float cos_value;
	if (shininess <= 0)
		cos_value = dot(N,L); // diffuse
	else
		cos_value = dot(R,L); // specular

	float2 tex;
	tex.x = (cos_value + 1)/2;
	tex.y = dws/2/PI;
	return Lin * ks * tex2D(DecorationSampler, tex).g;
}
```

### 2.1.4. Using the standalone Illumination Gathering program

The standalone DirectX application demonstrates the indirect illumination techniques described above, coupled with approximate ray tracing techniques and fast rendering of realistic metals. Lots of parameters can be controlled using the graphics user interface or hotkeys.

The most important parameters are listed here. For a complete list of available parameters, press F1 in the application.

Mouse: rotate/zoom camera
Arrow keys: move object
Keys 0-9: choose object mesh
TAB/Q: choose rendering mode
F: Load environment from a HDRI **F**ile
A: **A**utomatically re-generate environment maps in each frame.
Space: manually re-generate environment maps.
S: **S**ave a screenshot

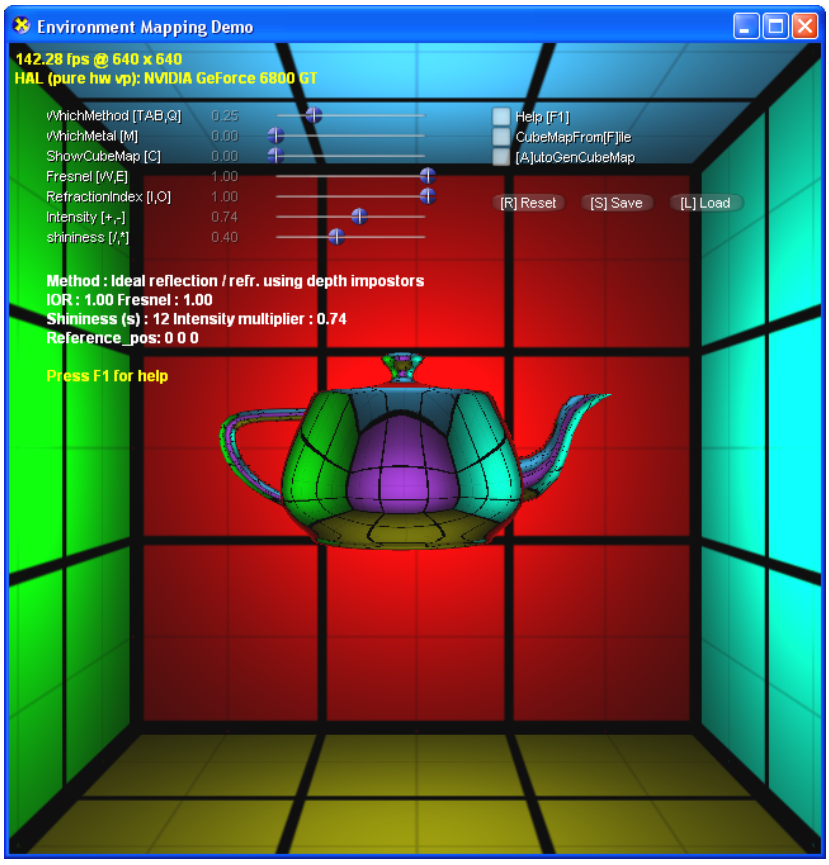
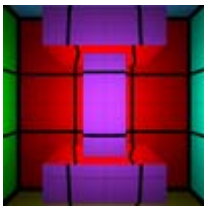



*Figure 15. Screenshot from the application*

You can switch between different meshes using keys 0..9. All media files are placed in the **Media** subfolder.

| 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| room.x<br>a cubic room | sphere.x<br>a sphere | column.x<br>I-beam shaped column | teapot.x | lamp01.x |
|  |  |  |  |  |

| 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|
| chair29.x | couch01.x | tiger.x | bunny.x<br>Stanford bunny | skullocc.x<br>skull |
|  |  |  |  |  |

## 2.2. RAY TRACING EFFECTS

This is a standalone module that can be found in the *RayTraceEffects* directory of the repository.

### 2.2.1. Role of the module

This module is based on a fast, *iterative* approximation method to obtain the point hit by a reflection or refraction ray. The calculation uses the *distance impostors* storing distance values in environment map texels. This approximation is used to localize environment mapped reflections and refractions, that is, to make them dependant on where they occur. On the other hand, placing the camera to the light source, the method is also good to generate real-time caustics. Computing a map for each refractor surface, we can even evaluate multiple refractions without tracing rays. The method is fast and accurate if the scene consists of larger planar faces, when the results are similar to that of ray-tracing.

When caustics are generated, the caustics patterns are rendered into textures. It means that every caustics receiver must have its own texture atlas. In order to allow comparisons, the standalone module also implements *classical environment mapping*.

The heart of the module is the *Hit* shader function that is responsible for finding a ray-surface intersection. It calculates the position in texture space of the static environment to generate real reflection/refraction/caustics effects. Unlike in the final gathering module, here the *Hit* method is iterative, thus allowing more iterations and time, it can obtain more precise results.

The other key part is the blending of the photon snippets into the objects atlases to generate caustics. Setting the snippet size and intensity are crucial to obtain quality results. The caustics are rendered into a texture that must support blending.

The algorithm works well for one or at most a few reflection/refraction/caustics effect generator objects. The rendering time is nearly independent of its complexity. The module can also handle more than one reflection/refraction/caustics effect generator objects, but the number of this kind of objects has a significant influence on the performance. The number of caustics receiver objects is also important, because this implementation works in texture space, so a shader must be run for every receiver object. Note that when the same algorithm was integrated into the Ogre engine, we used a different approach that solves this problem. The number of photons used for generate caustics effect, containing the photon hit locations is also important. The bigger its number, the smoother the caustics, but the algorithm gets slower.

### 2.2.2. Rendering modes

This program can demonstrate three different effects, reflection, refraction, and caustics. All three effects are implemented both in the classical and our localized environment mapped method. The effects have control parameters. For example the reflection is controlled by Fresnel factor, the refraction is controlled by refraction index. These parameters can be changed at runtime by a GUI slider.

The effects are generated by environment map lookups. The main task is to define the necessary direction. The classical method always uses the same direction, the direction where the point was at the environment map generation. The classical method can not handle the position change of the

object and the distortion caused by an object the size of which is comparable with its surrounding. The direction is independent from its occurrence. If the environment map is regenerated as the object moves the first problem is solved, but the regeneration is not free. The scene must be rendered 6 times per environment update. The distortion caused by object size can not be solved by regeneration.

Our method can handle both problems without environment map regeneration. In our method the lookup direction depends on the point of occurrence. We use approximation to find the correct direction. For the approximation we use several look ups from the environment map. This step is packed in a function called *Hit* in our shader code.

Function `Hit()` approximately traces a ray from point *x* towards direction *R*. Depth information is obtained from the alpha channel of *mp*. The function returns the approximate hit point.

```
float3 Hit( float3 x, float3 R, samplerCUBE mp )
{
	float rl = texCUBE( mp, R ).a;                              // |r|
	float ppp = length( x ) / texCUBE( mp, x ).a;               // |p|/|p'|
	float dun = 0, pun = ppp, dov = 0, pov;
	float dl = rl * ( 1 - ppp );                                // eq. 2
	float3 l = x + R * dl;                                      // ray equation

	// iteration
	for( int i = 0; i < g_iNumberOfIteration; i++ )
	{
		float llp = length( l ) / texCUBE( mp,l ).a;        // |l|/|l'|
		if ( llp < 0.999f )                                 // undershooting
		{
			dun = dl; pun = llp;                        // last undershooting
			dl += ( dov == 0 ) ? rl * ( 1 - llp ) :     // eq. 2
				( dl - dov ) * ( 1 - llp ) / ( llp - pov ); // eq. 3
		} else if ( llp > 1.001f )                          // overshooting
		{
			dov = dl; pov = llp;                        // last overshooting
			dl += ( dl -dun ) * ( 1 - llp ) / ( llp - pun );// eq. 3
		}
		l = x + R * dl;                                     // ray equation
	}
	return l;                                                   // computed hit point
}
```

The rendering methods featured in the application are detailed in the following sections.

### *2.2.2.1. Reflections with classical or localized environment mapping method*

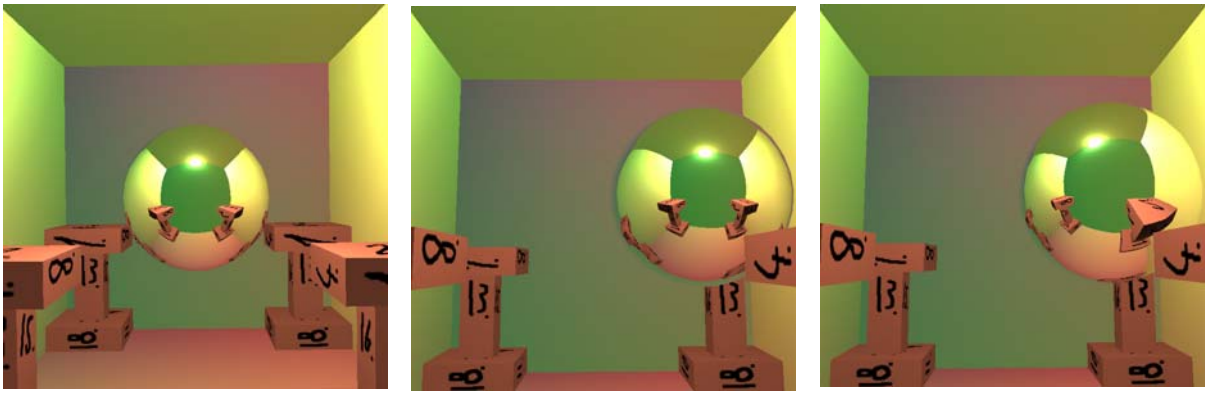


*Figure 16. Reflection with classical environment mapping and our method. Both are reasonably accurate if the object is near to the reference point, where the environment map was generated from (left). If the object moves (the environment map is not updated) the classical method suffers visible error (center), our method works accurately without noticeable error.*

### *2.2.2.2. Refractions with classical or localized environment map method*

This implementation uses single refraction. We found that double refraction is too expensive compared to its visual improvement. However, it is possible to implement the double refraction with this application.

*Figure 17. Refraction with classical environment mapping is highly inaccurate (left), our method works fine for this propose, too.*

### *2.2.2.3. Caustics with classical or localized environment map method*

The caustics effect is based on refraction. That is why the classical method can not generate what we want to see. The classical method fails with the refraction direction.

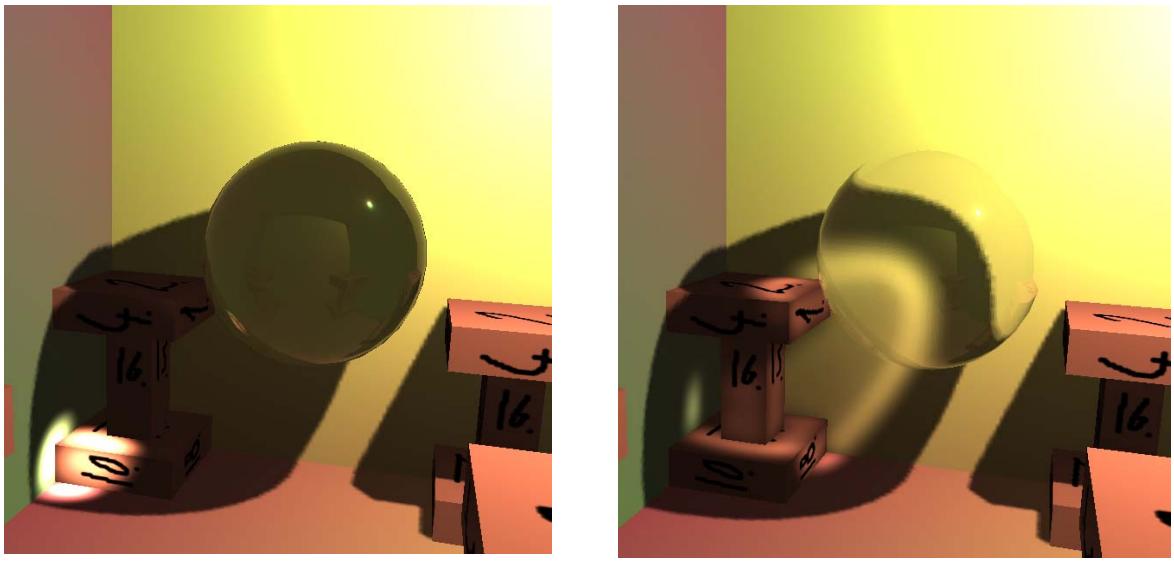


*Figure 18. Caustics effect with classical environment mapping is appalling (left), but our method generates convincing and realistic patterns (right).*

The caustic effect has numerous control parameters. You can change

- the number of iteration used in the `Hit()` function

  The further you iterate the better the approximation, the slower the algorithm. Good results can be seen from 4-6 iterations.
- the refraction index

  Influences the refraction directions and the area over which the caustic pattern is spread. The effect can be focused or smooth.
- the intensity of the light source

  The brightness of extra illumination due to the caustics effect.
- the number of photon hits

  Defines how many sample points are used for the effect generation. The bigger this number the smoother the visual effect.
- the snippet size

  Defines the area affected by one photon hit.

### 2.2.3. Program structure

For executing the program shader model 3.0 support is necessary. Some precalculated info is needed for drawing the caustic photons, so we have perform texture lookups in the vertex shader, which is only possible in shader model 3.0.

#### *2.2.3.1. Using the DirectX framework (DXUT)*

We are using the DXUT framework to create and setup the DirectX device (see the `DXUT...` calls inside the `WinMain` method). A convenient way to generate such a code is to run the SampleBrowser DirectX application and to install an EmptyProject (C++) application.

In the DXUT framework, after creating a new DirectX device, two functions are called to perform application specific initialization: `OnCreateDevice` and `OnResetDevice`. Before destroying a DirectX device, `OnLostDevice` and `OnDestroyDevice` functions are called.

The application starts with device creation and ends with device destruction. The device is destroyed and another one is created when the window is resized. In these cases, `OnLostDevice` and `OnResetDevice` function calls are performed to adapt the program to the new window size (e.g. the textures are released and recreated).

#### *2.2.3.2. Texture atlases for mesh objects*

To texture the room and the columns, texture atlases are used. This means that – for example – a single texture file contains all necessary data to texture a whole column (consisting of 18 quad faces).

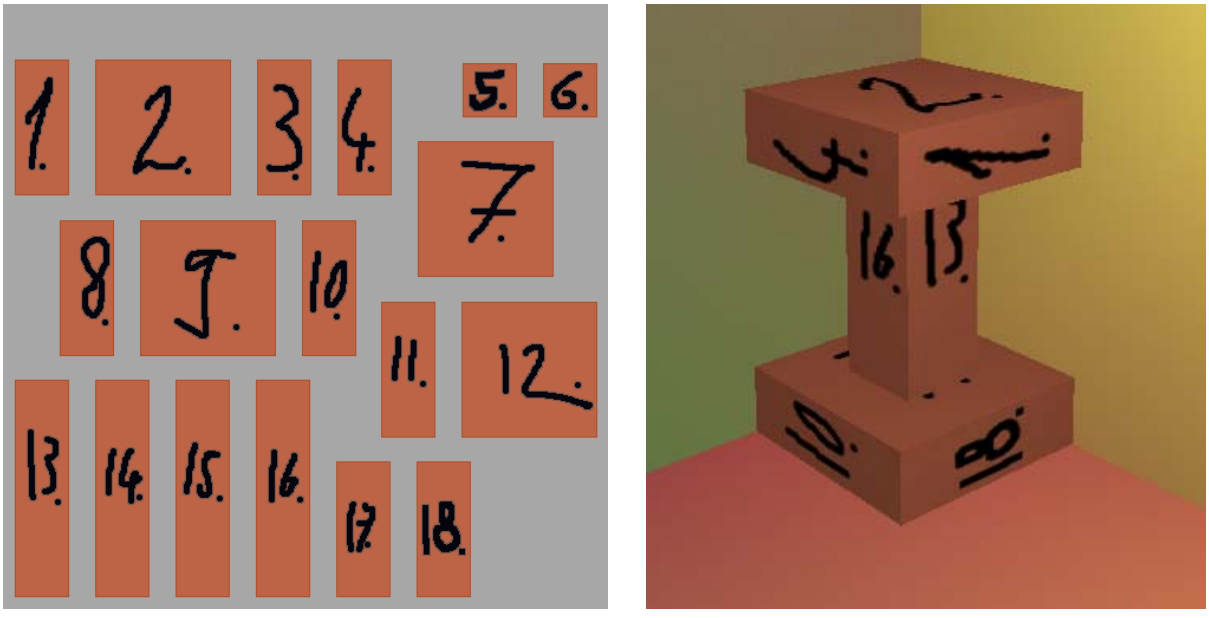



*Figure 19. Texture atlas for a column in the scene. Each surface element has its own texture in the atlas (left) and the textured column (right).*

Texture atlases play an important role in caustics generation since photon hits are rendered directly into them. The usage of the atlases causes errors at the edges of the faces which are not neighboring in the atlas, i.e. if their UV's are not neighboring. For example, suppose that we render the column with its atlas, and currently the first and the fourth sides are processed. A gray line may be seen at the edge of the two sides. The problem is that there is a background color where we read the atlas. It occurs because of rounding errors, we read and write the texture at different position. To solve this problem, we extend the texture of each face by two pixel wide borders. (A more sophisticated and economic solution to the same problem is described in the chapter Light Path Maps, section Rendering to a texture atlas.)

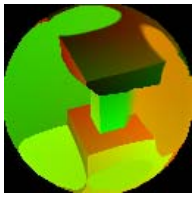
### *2.2.3.3. Program resources*

In the following we list the main resources used by the module grouped according to their types.

**TEXTURE**

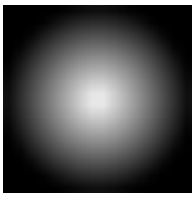
*g_pPhotonUVTexture*

This D3DFMT_A32B32G32R32F texture contains the positions of caustics hits in texture space. Its resolution defines the number of snippets to render. One snippet represents one photon hit for the caustics effect.



*g_pPowerOfSnippetTexelTexture*

This texture contains the power distribution inside a single caustic spot. It is used for every rendered snippet. It is a Gaussian filter.



*g_pRoomTexture*

Room's original Texture (brdf).

*g_pRoomModifyTexture*

This D3DFMT_A16B16G16R16F texture is used for blending the caustics effects into the light map of the room object representing the environment. It contains brdf, shadow, and caustics effects.

*g_pRoomLastTexture*

This is a D3DFMT_A16B16G16R16F texture. This is used for eliminating the error edges in *g_pRoomModifyTexture*. This texture is used for the Room object when we render it to the screen.

*g_pColumnModifyTexture[4]*

These D3DFMT_A16B16G16R16F textures are used for blending the caustics effects into the light map of the column objects representing the static environment. They contain brdf, shadow, caustics effects, and errors at the edges. There are four columns, that is why we need four atlases

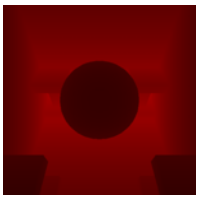
*g_pColumnLastTexture[5]*

They are D3DFMT_A16B16G16R16F textures. The first four are used for eliminating the error edges in *g_pColumnModifyTexture[4]*. These textures are used for the column objects when we render them to the screen.

The last texture contains the original brdf loaded from file.

*g_pShadowMapTexture*

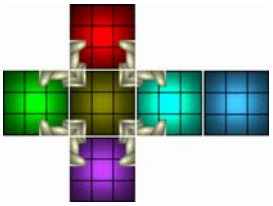
Shadow map texture with depth values from the light sources. This is used for shadow generation.



*g_pRoomCubeMapColorDistTexture*

A CubeMap to store the distance values between the surrounding objects and the center of the environment map.

The image is just an illustration of a cube map. Now it is not containing the distance values



*g_pRoomCubeMapUVTexture*

A CubeMap to store the UVs and the ObjectIDs. The ObjectID is the identification for every object in the scene.

**VERTEXBUFFER**

*g_pFullScreenQuadVertexBuffer*

It contains vertex data for a full screen quad, and is used for removing the error edges from the atlases.

*g_bSnippetVertexBuffer*

It contains vertex data for the photon snippets. This is rendered when the caustics effect is generated into the atlases. The number of these objects can be changed. More objects create softer caustics transitions, but need more time to accomplish.

**MESH**

*g_pCenterObjectMesh*

The mesh for the central object. This is the caustic generator object. This object is changeable at runtime.

*g_pRoomMesh*

Mesh representing the room (wall, floor, and ceiling)

*g_pColumnMesh*

Mesh representing one Column. In the scene there are 4 columns, they are moved by different transformations to different positions.

*g_pLightMesh*

Mesh representing the light object.

### *2.2.3.4. Shader description*

The shaders call the following two global functions:

*float3 Hit( float3 x, float3 R, samplerCUBE mp )*

It calculates the hit points after refraction on the receiver object.

*float4 FlowLight( sampler2D mp, float2 Tex )*

It removes the black edges from the texture stretching it by two pixels at the edges.

Every shader uses only one pass. The shaders are as follows:

*RenderRoomColorDistance*

It renders the environment (*Room* and the object set of *Columns*) from the *CenterObject* position and stores the result in *g_pRoomCubeMapColorDistTexture*. The render target contains *COLOR(x,y,z) + Distance(w).*

*RenderRoomUV*

It renders the environment (Room and the object set of Columns) from the *CenterObject* position and stores the result in *g_pRoomCubeMapUVTexture*. The render target contains *UV(x,y) + ObjectID(z) + "1".*

*RenderUmbra*

It renders the environment (*Room* and the object set of *Columns*) from the *CenterObject* position and stores the result in their texture atlases. The render target contains *Color(r,g,b,a)*. This shader copies the brdf color from the original texture atlas of the object to the *g_p…ModifyTexture* atlas of the actual object. It is needed, because the color is modulated by the shadow.

*RenderRoomAndColumnsScreen*

It renders the environment (*Room* and the object set of *Columns*) from the camera position and presents the image on the screen.

*RenderPhotonUVMap*

It renders the *CenterObject* from the light position and stores it in *g_pPhotonUVTexture*. The render target contains: *UV(x,y) + ObjectID(1) + "1"* or *"-1"*. Here "-1" means that the actual pixel does not represent a valid photon hit. At snippet render time, such fragments are ignored. This shader creates the *g_pPhotonUVTexture* with the *Hit* function, which implements the distance impostor based approximate ray tracing algorithm.

*RenderPhotonUVMapClassic*

It renders the *CenterObject* from the light position and stores it in *g_pPhotonUVTexture*. The render target contains *UV(x,y) + ObjectID(1) + "1"* or *"-1"*. Here "-1" means that the actual pixel does not represent a valid photon. At snippet render time, such fragments are ignored. This shader creates the *g_pPhotonUVTexture*. This is the classical environment mapping based method, which we implemented for comparisons.

*RenderPhotonHit*

It renders and blends impostors into the atlases of the environment objects (Room and the object set of Columns). The render target contains *Color(r,g,b,a)*. This shader uses the *g_pPhotonUVTexture* as an input, and renders as many snippets to the object's atlas as many pixels the *g_pPhotonUVTexture* has. If a pixel in the *g_pPhotonUVTexture* is not connected to the actual object (not the same ObjectID), it will not be rendered. The result is blended into the Object's Atlas.

*RenderRefractObjectScreen*

It renders *CenterObject* from the camera position and shows it on the screen. The render target contains *Color(r,g,b,a)*. This shader calculates the color of the *CenterObject* with the *Hit* function, computing reflection and refraction effects. The result is displayed on the screen.

*RenderRefractObjectScreenClassic*

It renders *CenterObject* from the camera position and shows it on the screen. This shader calculates the color of the *CenterObject* with classical environment mapping method to allow comparisons. The result is displayed on the screen.

*RenderLightScreen*

This shader renders the light source object to the screen with a diffuse color.

*RenderShadow*

It renders *CenterObject*, *Columns* and *Room* from the light position and stores the depth image in *g_pShadowMapTexture*, which contains *Depth(x)*.

*FullScreenQuad*

It renders a full screen quad and modifies the Room and Columns atlases, which contain *Color(r,g,b,a).* This shader modifies the atlases by removing the black edges from them.

*PhotonMapScreen*

It renders a full screen quad which is transformed into the lower left corner of the screen in the shader. The render target contains *Color(r,g,b,a)*. This shader renders the photon map texture (*g_txPhotonUVMap*) into the screen lower left corner. The result is displayed on the screen.

Concerning the usage of the shaders we can distinguish static and dynamic shaders. Static shaders run just once, for example, during initialization and when the light source or the *CenterObject* is moved. Dynamic shaders run in every frame. On the other hand, we can talk about global shaders that run for every object, or local ones that are executed just for specific objects.

During initialization we call the following global/static shaders:

- `RenderRoomColorDistance`
- `RenderRoomUV`

If the light source or the *CenterObject* moves, the caustics and shadow effects should be re-calculated with the following shaders in the specified order:

- `RenderPhotonUVMapClassic` OR `RenderPhotonUVMap`
- `RenderShadow`
- `RenderUmbra`
- `RenderPhotonHit`
- `FullScreenQuad`

The following shaders run in every frame in the specified order:

- `RenderRoomAndColumnsScreen`
- `RenderLightScreen`

- `RenderRefractObjectScreenClassic`
- `RenderRefractObjectScreen`
- `PhotonMapScreen`

If the caustics or shadow effect need to be updated, caustics and shadow recalculation shaders are invoked. They generate textures which will be rendered at every frame for the objects. Note that this shader runs only if the caustic generator or the light source moves.

### 2.2.4. Using the standalone Ray Trace Effects module

There are many parameters that can be set on the graphics user interface (e.g. sliders, checkboxes, or keyboard shortcuts) to experiment with the system.

F1 displays information about the general controls. Note that for easier reading, the actual scene will be darkened while the help text is displayed. Press F1 again to hide the help text and restore the original brightness of the scene.

To move the central object, use the arrow keys and Page Up / Page Down, respectively. To move the light source, you should press the ALT button simultaneously.

You can change all parameters using the GUI sliders and checkboxes. If you prefer using your keyboard, press SPACE to enter to “Edit parameters” mode. In this mode, you can use the arrow keys to select a parameter and to change its value. Press Alt + Arrow keys to change the parameters in bigger steps.

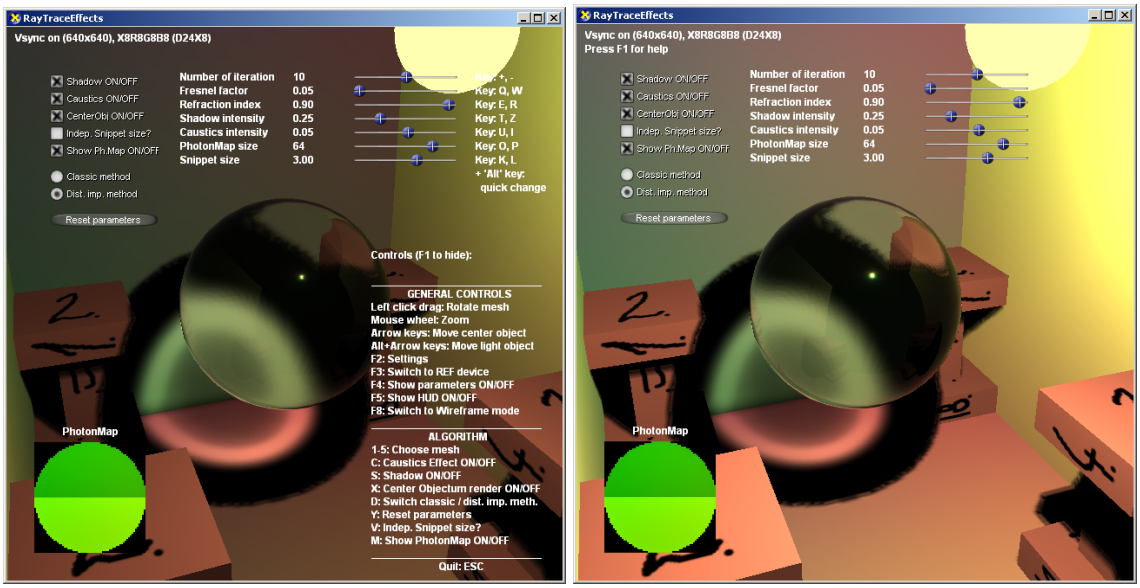



*Figure 20. The main parameters of the algorithm are adjustable on the graphics user interface. The content of the photon map is also shown in a small image in the lower left corner. This map contains the photon hits used for caustics generation.*

The user interface controls are the followings:

- Left click drag: Rotate mesh
- Mouse wheel: Zoom

- Arrow keys: Move central object – the reflector/refractor object
- Alt+Arrow keys: Move light object
- F2: Settings
- F3: Switch to REF device
- F4: Show parameters ON/OFF
- F5: Show HUD ON/OFF
- F8: Switch to Wireframe mode
- 1-5: Choose mesh
- C: Caustics Effect ON/OFF
- S: Shadow ON/OFF
- X: Central Object render ON/OFF
- D: Switch classic / our new distance impostor method
- Y: Reset parameters
- V: Independent Snippet size ON/OFF – we can adjust the snippets size automatically or it can be constant
- M: Show Texture sample ON/OFF

If the central object rendering is switched off, the central object is rendered with a single color.

For the sake of uniformity, all meshes (including the rather simple meshes of the surrounding: *room* and *columns*) are stored in X file format of DirectX.

The user can switch between different central objects by pressing the 1, 2, 3, 4, 5 and 6 buttons. Each central mesh is rescaled to (approximately) the same size and is translated to the center of the room.

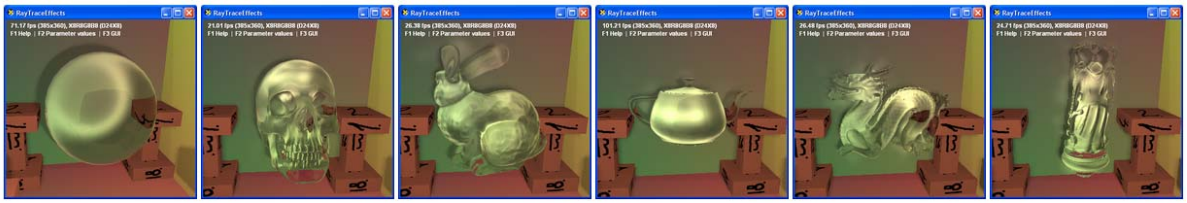


*Figure 21. The possible choices for the central object: SPHERE, SKULL, BUNNY, TEAPOT, DRAGON, and HAPPY BUDDHA.*

## 2.3. ILLUMINATION MODULE IN OGRE

Some of the modules discussed in this document were integrated into the Ogre engine. The implemented modules are:

- Ideal reflections and refractions with localized environment mapping
- Metal materials
- Diffuse reflections with preconvolved environment mapping
- Caustics effects with the localized environment method
- Spherical billboard particle rendering
- Glow

The Illumination Module collects and organizes these modules, standardizes them, manages the creation and refresh of render targets, sets the appropriate shaders for rendering.

### 2.3.1. Role of the module

This module is an organizing tool which enables different modules to work efficiently with each other. This is the collection of the other modules but organized in a way that the different tools can be treated similarly and common resources can be easily shared between modules and objects too.

The module is based on the fact that all the different effects have some common features. They all need some kind of resources. These resources are usually textures. To create these resources one or more rendering passes should be made.

Two different modules may require the same resources (e.g.: a distance impostor cubemap), so these resources can be shared between the modules. It is also possible that two objects that are used by two modules (not necessary the same module) can have common resources. This can happen in case of global resources (e.g.: scene depth map) or per light resources (e.g.: depth shadow map from a light source), but it can also happen in more general cases (e.g.: for efficiency reasons we would like to compute a common cubemap for several small objects that are close to each others). Managing shared resources in case of moving objects is complicated, as resources need to be joined and split dynamically.

It is also necessary to automate the creation of resources, the assignment of resources to the different modules and the assignment of modules to be used to the objects.

### 2.3.2. Program structure

The source code of the illumination module library can be found in the repository. It consists of two projects: an abstract module (`IllumModule`) and an Ogre implementation (`OgreIllumModule`). The documentation generated by *Doxygen* can also be found in the repository. Hereinafter we do not discuss the source in details but will concentrate on the basic structure and usage. Neither do we discuss the abstract module nor the Ogre implementations separately as their basics are the same.

### *2.3.2.1. Base classes*

In the followings we list the basic objects which correspond to the basic concepts of the Illumination Module.

**ELEMENTARYRENDERABLE**

A renderable is an object with one material. It can be moved, rendered, it can be hidden and shown if needed. A renderable will contain information about which modules use it and how. The Ogre implementation of this class is a wrapper class that collects different Ogre `Renderable` types.

**RENDERINGRUN**

A rendering run is created to compute some kind of resource that can be used by the modules. A run typically - but not necessarily or exclusively - consists of a series of rendering passes. `RenderingRuns` are the resources which can be shared between objects and modules. Each run has an update frequency and a starting frame where the first update takes place. This can help to evenly distribute the updates of separate runs resulting in a smoother animation.

E. g.: a `DistanceCubeMapRenderingRun` is responsible for creating a depth impostor cubemap of the surrounding environment of an object, or group of objects if the impostor can be shared (the objects are relatively small and close to each other). This cubemap can be used by the environment mapping module to display the object with refracting material, or by the ray tracing effects module to compute caustics, or even both (the run is shared between modules).

**RENDERINGTECHNIQUE**

A `RenderTechnique` gives a description how an object should be rendered, and what kinds of resources are needed to do this. A `RenderTechnique` does not define the whole process of rendering only one property of the display, for example: this object will need a cubemap, or this object is going to be a caustic caster. The technique defines the type of `RenderingRuns` that should be created.

A `RenderTechnique` is always assigned to one renderable. On the other hand one renderable can have several `RenderTechnique` objects attached (e.g.: an object can be a caustic caster and a refracting material at the same time). The class that implements a group of techniques attached to a single renderable is `TechniqueGroup`. Its main task is to receive and forward messages to each `RenderTechnique` it contains.

Whenever a `RenderingRun` is changed or updated, a messaging function is called to inform the `RenderTechnique`.

**SHAREDRUNS**

This is the class that is responsible for sharing the resources. It collects the common resources of a group of techniques. It is also acts like a node of a binary tree so `SharedRuns` can be joined too. This way group of objects can be grouped further and treated as one object.

`RenderingRuns` will operate on `SharedRuns` nodes as they represent the largest group of objects that can share the type of resource. As they represent groups of objects they should have functions for hiding and showing all the objects under the group. The Ogre implementation also has functions for retrieving the bounding sphere and axis aligned bounding box of the whole group as well as functions for adding all the objects to a `RenderQueue` or calling `notifyCamera()` for the objects.

It is important to notice that one object can be a member of several groups as a binary tree of `SharedRuns` represent several groups (each node is a group of objects). The grouping takes place on the `RenderingRun` type level.

**ILLUMINATIONMANAGER**

The illumination manager is responsible for refreshing rendering techniques connected to visible renderables, and to render the scene with these updated resources. It also has the responsibility to manage shared runs, to join and split them if needed.

The Ogre implementation of the class has even more tasks. It is responsible for creating the techniques and to assign them to renderables. It stores global and per light resources and updates them if necessary. It also stores information about the player camera and viewport.

### *2.3.2.2. Implemented techniques and runs*

In the following a brief description is given about the implemented techniques and rendering runs.

Rendering runs:

- **ColorCubeMapRenderingRun, OgreColorCubeMapRenderingRun**
  generate a color cubemap of the surrounding environment of an object or group of objects.
- **DistanceCubeMapRenderingRun, OgreDistanceCubeMapRenderingRun**
  generate a distance impostor cubemap of the surrounding environment of an object or group of objects.
- **ReducedCubeMapRenderingRun, OgreReducedCubeMapRenderingRun**
  generate a reduced sized version of the color cubemap. It is created with averaging the original cubemap.
- **CausticCubeMapRenderingRun, OgreCausticCubeMapRenderingRun**

generate a cubemap that stores caustic light spots caused by a caustic emitter object. The caustic spots are created with caustic particles cast on a cubemap. The directions where the particles should be put are read from a photon hit map.

- **PhotonMapRenderingRun, OgrePhotonMapRenderingRun**
  generate a photon hit map texture. A photon hit map stores the directions where the incoming photons are refracted by a caustic emitter object. One pixel of the photon map represents one photon hit; the direction is encoded in the RGB channels (the directions are not normalized so the hit surface point can be exactly identified). If the alpha channel has zero value, the hit is invalid.
- **SceneCameraDepthRenderingRun, OgreSceneCameraDepthRenderingRun**
  generate a scene depth map texture. The depth map stores the scene's camera space z coordinates (rendered from the player's view).
- **DepthShadowMapRenderingRun, OgreDepthShadowMapRenderingRun**
  generate a depth shadow map texture. A shadow map stores depth or distance values from the light source.

Rendering techniques:

- **CubeMapRenderTechnique, OgreCubeMapRenderTechnique**
  define that the rendering will need a color cubemap. The color cubemap can be used to generate reflections and refractions. This technique creates a `ColorCubeMapRenderingRun` and binds its result to the material of the object.
- **DistanceCubeMapRenderTechnique, OgreDistanceCubeMapRenderTechnique**
  define that the rendering will need a distance impostor cubemap. The distance cubemap can be used to improve the accuracy of ray hits. This technique creates a `DistanceCubeMapRenderingRun` and binds its result to the material of the object.
- **ConvolvedCubeMapRenderTechnique, OgreConvolvedCubeMapRenderTechnique**
  define that the rendering will need a reduced sized color cubemap. The lower resolution cubemap can be convolved faster and can efficiently be used in effects like diffuse reflection. This technique creates a `ColorCubeMapRenderingRun` and a `ReducedCubeMapRenderingRun` and binds its result to the material of the object. It also sets the connection between the `ColorCubeMapRenderingRun` and the `ReducedCubeMapRenderingRun`.
- **CausticRecieverRenderTechnique, OgreCausticRecieverRenderTechnique**
  defined that the object can receive caustics. The technique forces the nearest caustic casters to refresh their resources and binds their caustic cubemaps to the material of the object.
- **CausticCasterRenderTechnique, OgreCausticCasterRenderTechnique**
  define that the object casts caustics. This technique creates a `PhotonMapRenderingRun`, a `CausticCubeMapRenderingRun` and a `DistanceCubeMapRendering` run if distance impostors are needed.
- **DepthShadowRecieverRenderTechnique, OgreDepthShadowRecieverRenderTechnique**

Defines that the object receives shadows with depth map shadow technique. This technique searches for the nearest light sources and forces them to update their shadow map, and then bind the map to the material of the object.

- **SBBRenderTechnique, OgreSBBRenderTechnique**

  define that the object (a particle system) uses the spherical billboard method. This technique creates a `SceneCameraDepthRenderingRun` and binds its result to the material of the object.

### 2.3.3. Usage of the illumination module in Ogre

The Ogre implementation of the illumination module was designed to have a simple usage, and to fit to the concepts of a typical Ogre application. A few modifications in the Ogre core had to be made to fit the module into the engine. We have to store a `RenderTechniqueGroup` for each renderable so a new attribute was added to the `Renderable` class that stores a `RenderTechniqueGroup` reference. The material serializer was also changed for the following reasons.

In Ogre RenderTechniques should operate on a Pass of the object's material (note that most of the techniques set outputs from a resource to some input of a Material, typically to a texture unit state of a Pass). It would be very comfortable to define the RenderTechniques to be used in the material script within the pass the technique should operate on.

We followed this solution so added a new keyword `IllumTechniques` within the pass definition of the materials. Within this scope we can define the techniques to be used for the given pass with the keyword `RenderTechnique`. This keyword needs a parameter that defines the type of the technique. Within the `RenderTechnique` scope we can set some technique specific parameters which will be passed to the technique's constructor. An example of this technique definition which defines an object as a caustic and a shadow receiver is the following:

```
material exampleMaterial
{
  technique
  {
    pass
    {
      IllumTechniques
      {
       RenderTechnique DepthShadowReciever
       {
          max_light_count            1
       }
       RenderTechnique CausticReciever
       {
          max_caster_count           2
       }
      }
       /// … usual pass definition
    }
  }
}
```

In the following we describe the material definitions for each `RenderTechnique` type. First the type name is given which should be set to the `RenderTechnique` keyword. Then the available parameters with their default values and a brief description are listed.

### Color cubemap

**type name:** `ColorCubeMap`

**parameters:**

`start_frame` `1`

number of the frame when the rendertarget created by this technique should be first updated

`update_interval` `1`

update frequency of the rendertarget created by this technique

`resolution` `256`

the resolution of the cubemap texture to be created

`texture_unit_id` `0`

the texture unit state id to bind the cubemap

`distance_calc` `true 2.0`

if this is on, the cubemaps of the object is refreshed rarely if the object is far away from the viewer. The second value is a tuning value, the higher it gets the more precise, but slower the method will be.

`face_angle_calc` `false 2.0`

if this is on, the cubemap faces which are not facing directly the camera are refreshed rarely. The second value defines how the refresh rate depends on the facing, the higher it gets the more precise, but slower the method will be.

`update_all_face` `false`

if this is on, all six cubemap faces will be refreshed in a frame. If it is off, only one face will be updated in a frame.

### Distance cubemap

**type name:** `DistanceCubeMap`

**parameters:**

`start_frame` `1`

(same as in ColorCubeMap)

`update_interval` `1`

(same as in ColorCubeMap)

`resolution` `256`

(same as in ColorCubeMap)

`texture_unit_id` `1`

(same as in ColorCubeMap)

`distance_calc` `true 2.0`

(same as in ColorCubeMap)

`face_angle_calc` `false 2.0`

(same as in ColorCubeMap)

`update_all_face` `false`

(same as in ColorCubeMap)

### Convolved cubemap

**type name:** `ReducedColorCubeMap`

**parameters:**

`start_frame` `1`

(same as in ColorCubeMap)

`update_interval` `1`

(same as in ColorCubeMap)

`resolution` `256`

resolution of the color cube map

`reduced_resolution` `8`

resolution of the reduced sized color cubemap

`texture_unit_id` `0`

(same as in ColorCubeMap)

`distance_calc` `true 2.0`

(same as in ColorCubeMap)

`face_angle_calc` `false 2.0`

(same as in ColorCubeMap)

`update_all_face` `false`

(same as in ColorCubeMap)

### Caustic caster

**type name:** `CausticCaster`

**parameters:**

`start_frame` `1`

(same as in ColorCubeMap)

`update_interval` `1`

(same as in ColorCubeMap)

`photonmap_resolution` `64`

the resolution of the photon hit map to be created

`caustic_cubemap_resolution` `256`

resolution of the caustic cubemap texture to be created

`photon_map_material` `GameTools/PhotonMapCaustic`

name of the material to use when rendering the photon hit map

`caustic_map_material` `GameTools/Cau`

name of the material to use when rendering the custic cubemap

`photon_map_tex_id` `0`

texture unit id of the caustic material where the photon hit map should be bound to

`distance_impostor` `true`

use distance impostor cubemap when rendering the photon hit map (recommended)

`update_all_face` `false`

if this is on, all six cubemap faces of the caustic cubemap will be refreshed in a frame. If off, only one face will be updated in a frame.

### Caustic receiver

**type name:** `CausticReciever`

**parameters:**

`max_caster_count` `1`

the maximum number of caustic casters to take into account (the nearest casters will be used)

`vertex_program_name` `GameTools/Caustic/DefaultVS`

the name of the vertex program to be used when running the caustic gathering passes

`fragment_program_name` `GameTools/Caustic/DefaultPS`

the name of the fragment program to be used when running the caustic gathering passes

### Spherical billboards

**type name:** `SphericalBillboard`

**parameters:**

`texture_unit_id` `0`

the texture unit state id of this pass where the scene depth texture should be bound to

### Depth shadow receiver

**type name:** `DepthShadowReciever`

**parameters:**

`max_light_count` `1`

the maximum number of light sources to take into account (the nearest ones will be used)

`vertex_program_name` `GameTools/ShadowMap/ShadowVS`

the name of the vertex program to be used when running the shadowing passes

`fragment_program_name` `GameTools/ShadowMap/ShadowPS`

the name of the fragment program to be used when running the shadowing passes

If we would like to build an Ogre application that uses the `OgreIllumModule` and its techniques, only a few changes should be needed in the `ExampleApplication` provided by the Ogre SDK. In the `createScene()` function we can set some global parameters of the

`OgreIlluminationManager` if needed. E.g.: we can set the maximum bounding radius used in joining `SharedRuns`, or if we use the spherical billboard method we should set the main camera.

After mesh loading and material setup `OgreIlluminationManager` should search for `RenderTechniques` in the material definitions and initialize them (this can be done with the `initTechiques` function).

In each frame the IlluminationManager should refresh the techniques of the visible objects. This can be done automatically if we add the IlluminationManager to the frame listeners, or it can be done manually if the application calls the `update()` function in each frame (the demos follow this method). The following listing shows a sample code:

```
Class App : public ExampleApplication
{
public:
 Virtual void go(void)
 {
  if (!setup()) return;
  renderScene();
  destroyScene();
 }
protected:
 void renderScene(){
  mWindow->resetStatistics();
    while(true){
        //message handling
        #if OGRE_PLATFORM == OGRE_PLATFORM_WIN32
        // Pump events on Win32
        MSG  msg;
        while( PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE ) )
        {
         TranslateMessage( &msg );
         DispatchMessage( &msg );
        }
        #endif
        //call framlisteners
        if(!mRoot->_fireFrameStarted()){
               break;
        }
        unsigned long framenum = mRoot->getCurrentFrameNumber();
        //update techniques
        OgreIlluminationManager::getSingleton().update(framenum, mWindow);
        //Ogre's standard rendering
        mRoot->_updateAllRenderTargets();
        //call framlisteners
        mRoot->_fireFrameEnded();
   }
 }

void createScene(void) {
        …
        //usual Ogre code (mesh loading, transformations, material setup)
        …
        //set global parameters
        OgreIlluminationManager::getSingleton().setMaxJoinRadius(420);
        OgreIlluminationManager::getSingleton().setMainCamera(mCamera);
        //search for techniques and initialize them
        OgreIlluminationManager::getSingleton().initTechniques();
 }
}
```

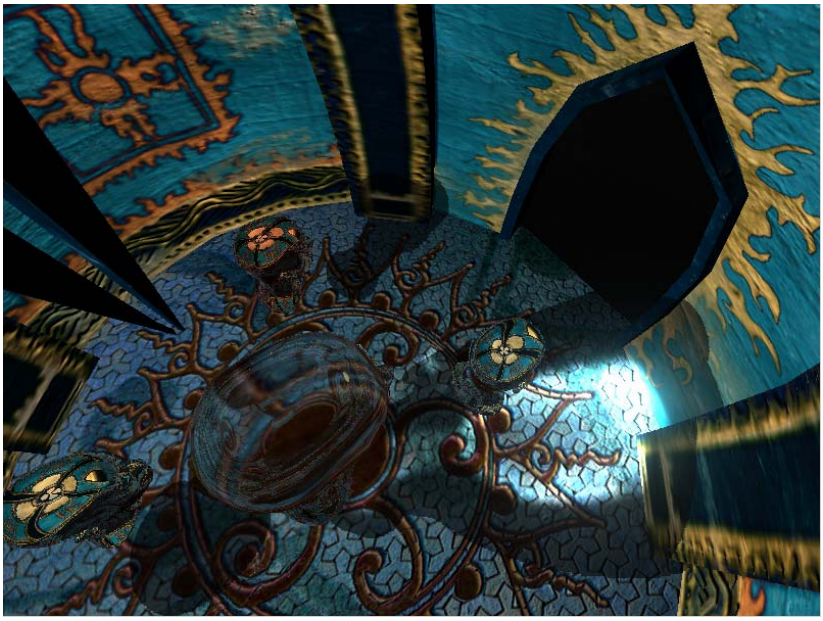


*Figure 22. Caustics and environment mapping implemented in Ogre. The demo shows four reflecting head with different metal shaders (as introduced in* `2.1.2.9`*) rotating around a refracting head which casts caustics on the walls and floor. The reflecting and refracting objects use localized environment mapping. The program uses a caustic generating method similar to the method described in 2.2.2.3. Also a glow effect was added to the scene (see 2.6.1).*

*Figure 23. Spherical billboards implemented in Ogre. The demo shows an ogre head inside a particle system. Clipping and popping artifacts were removed by the spherical billboards method described in details in the GameTools technical documentation deliverable. Also a glow effect was added to the scene (see 2.6.1).*

## 2.4. ILLUMINATION NETWORKS MODULE

### 2.4.1. The role of the module

Illumination networks estimate the discretized volumetric rendering equation by a Monte Carlo quadrature with a given set of directions. The process of calculation is described in details in the GameTools technical documentation deliverable (chapter 13). The main idea behind this method is to calculate incoming light radiance for a given set of directions for each particle. For any given direction and particle it is defined which other particle is the closest visible one from the given particle. This way the particles form a network according to this visibility information. This network can be used to calculate incoming radiance for every direction with an iteration process. During rendering we need the amount of radiance heading towards the eye to shade the individual particles.

The tool offers real time rendering of realistically shaded participating media with multiple anisotropic scattering for multiple dynamic (both position and color can change) light sources. Even dynamic light sources inside the medium can be modeled.

### 2.4.2. Rendering algorithm

We need a set of sample directions which are randomly chosen and have uniform distribution on the unit sphere. In order to update the radiance of a particle, we should know the indices of the particles visible in sample directions, and also the distances of these particles to compute the opacity. This information can be stored in two-dimensional arrays as two dimensional textures. Incoming radiances can also be stored in a two dimensional texture. Initially this texture is black. Later on this texture will be refreshed in each iteration step (once in a frame). To add an initial radiance to this texture to start the iteration with we also store the particles emission in the given directions.

The direct illumination of light sources should also be added. To do this we should identify those particles that are visible from the light source and store this information in an array (1D texture). This array can be initialized by rendering the volume from the point of view of the light source and identifying those particles that are directly visible on the resulting image.

The visibility array can also be calculated in a similar way, but placing the camera at the given particle and the looking in the given direction. When a particular direction is processed, particles are orthographically projected onto the window, and rendered using a standard z-buffer algorithm, having set the color of a particle equal to its index. The contents of the image and depth buffers are read back to the CPU memory, and the indices and depths of the visible particles are stored together with the pixel coordinates. This information can also be calculated on the CPU as it is run only once.

During iteration for each particle and for each direction we identify the visible particle in that direction with the help of the visibility map and calculate the light it scatters toward the examined particle with the help of the illumination texture. After collecting the light contributions from each direction their sum gives the refreshed value to be written to the illumination texture. The direct illumination and emission should also be taken into account. Note that both reading and writing is needed in the illumination texture, which can not be done on the GPU at the same time, so we need a copy with last state of this texture which can be read while the other one is written.

When the final image is needed, we can use a traditional participating media rendering method, which sorts the particles according to their distance from the camera, splats them, and adds their contributions with alpha blending. When the outgoing reflected radiance of a particle is needed, we compute the reflection from the sampled incoming directions to the viewing direction. Finally the sum of particle emission and direct illumination of the external lights is interpolated from the sample directions, and is added to the reflected radiance.

### 2.4.3. Program structure

The demo application named `Illumination Networks Demo` that demonstrates the described technique can be found in the repository. It is a small OpenGL/Glut application. It loads particle positions from a text file. It uses several simple wrapper classes (e.g.: camera, texture, render texture, Cg GPU program wrapper classes) which don't concern strictly the technique described above, so they won't be documented here. Main initializations event handling and the rendering loop is located in `main.cpp`.

One main class is responsible for implementing a particle system shaded with the illumination networks method named `PreIllumSystem`. It contains a `ParticleSystem` instance which handles particle position loading and sends the particles to the graphics card as quads during rendering. `PreIllumSystem` is only responsible for the rendering process described above. It has functions for the main rendering tasks:

`CreateGivenDirections()` is for generating random directions,

`CreateVisibilityTexture()` builds the visibility texture,

`CreateLVisMap()` creates the map that stores the visible particles from the light source,

`RefreshDirectIllumTexture()` refreshes direct illumination and emission,

`Iterate()` does one iteration step,

`CreateEyeRadTexture()` gathers radiance to the eye.

The following Cg shader programs correspond to the main rendering tasks, most of them are used with full screen quad rendering:

`LightIlluminatePrograms.cg:` is used in direct illumination and emission refresh

`IllumIteratePrograms.cg:` is used in one iteration step

`FinalizeIllumMap.cg:` is to compute eye radiance

`FinalDisplay.cg:` creates the final display of the particles (point sprite rendering)

### 2.4.4. Using the illumination network application

This sample application displays a cloud shaded with two light sources and skylight. The camera can be rotated around the cloud, one of the light sources can be moved, light colors and the cloud properties (albedo, opacity, scattering symmetry) can be changed.

**Controls:**

- w, s : moves camera closer and further from cloud
- a, d : rotates camera around cloud
- r, f : moves camera up and down
- left, right : rotates light source around cloud
- up, down : moves light source up and down
- F1, shift + F1: increase, decrease iterate count (recommended to set to 1)
- F2, shift + F2 : increase, decrease cloud albedo
- F3, shift + F3 : increase, decrease cloud opacity
- F4, shift + F4 : increase, decrease red color value of light 1.
- F5, shift + F5 : increase, decrease green color value of light 1.
- F6, shift + F6 : increase, decrease blue color value of light 1.
- F7, shift + F7 : increase, decrease red color value of light 2.
- F8, shift + F8 : increase, decrease green color value of light 2.
- F9, shift + F9 : increase, decrease blue color value of light 2.
- F10, shift + F10 : increase, decrease red color value of skylight
- F11, shift + F11 : increase, decrease green color value of skylight
- F12, shift + F12 : increase, decrease blue color value of skylight
- p : start/stop screenshot saving
- h : change heads up display
- l : change display

**Display types:**

- Normal: shows the shaded cloud.

The following display types are for debugging and representation:

- Visible particles from light source colored with their ids.
- Light visibility map: binary map contains 1 in column j if the particle is visible from the light source in direction j. It has two rows for the two light sources.
- Illumination map1: last result of the illumination map.
- Illumination map2: refreshed illumination map.
- Eye radiance map: outgoing radiances of the particles towards the eye.

*24. Screenshots made with the illumination networks demo application.*

## 2.5. HIERARCHICAL PARTICLE SYSTEMS MODULE

### 2.5.1. The role of the module

Hierarchical particle systems deal with the problem of high particle count that should be used to achieve realistic high quality effects. If we want to add detail to the individual particles, we can use a detail texture (usually a video texture) that visualizes the desired natural phenomena. This texture gives detail by perturbing the particle opacity and or color. This usually gives nice results but the cheating is usually visible if the camera moves in or rotates around the medium, because these images are only 2D and has no depth information. The main task of hierarchical systems is to create the correct image for all view directions. We use a particle system that will visualize the described phenomena and give a detail texture for all view directions. This way we build a particle system made out of duplications of a smaller particle system, but the smaller system needs to be rendered only once and the resulting image will be multiplied.

Another great advantage of this method is that not only color and opacity but depth information can be stored for the given direction, which gives us a good substitute of the small system. These color or opacity images with their depth information are called depth impostors. To handle volumetric media as one depth information is not enough we have to store both front and back depth values. This information can be used to determine correct opacity taking into account the objects located inside particle system, so billboard clipping can be avoided which results even higher realism.

The simplified modeling of the particle system also makes real time shading of the medium possible. Handling light volume interaction on the particle level would be too computation intensive since the light pass requires the incremental rendering of all particles and the read back of the actual result for each of them. To speed up the process, we render particle blocks one by one, and separate light volume interaction calculation from the particles. During a light pass we classify particle blocks into groups according to their distances from the light source, and store the evolving image in textures at given sample distances. These textures are called slices. The first texture will display the accumulated opacity of the first group of particle blocks; the second will show the opacity of the first and second groups of particle blocks and so on. The required number of depth samples depends on the particle count and the cloud shape. For a roughly spherical shape and relatively few particles (where overlapping is not dominant), even four depths can be enough. This method provides real time shading of animated particle systems under changing light conditions (both light color and position can be changed).

The main concepts of particle hierarchies, depth impostors and the volume shading algorithm are described in details in the GameTools technical documentation deliverable (chapter 12).

### 2.5.2. Hierarchical particle system rendering algorithm

First we should make an image of the smaller particle system (a block). We make a frustum that tightly encloses the block and the eye point is in the player's eye position. Then we render an opacity, a front depth and a back depth. Note that these tree passes can be done in one rendering pass if we set the proper blending factors.

We need a depth map of the scene. This will help us to identify the objects positions. Finally we can render the main particle system; the particles are processed in back to front order. Each particle will

have the opacity of the precomputed block opacity. This opacity will also be altered according to the ray segment length light should travel in one particle before reaching an object. This can be calculated from the scene depth, the particle position and the precalculated depth information.

The shading also needs additional passes. We should create the slices of the light volume texture. The number of passes needed can be reduced by reducing the number of slices used and with storing separate slices in separate color channels. The particles are sorted in front to back order and rendered in groups according to the slices. During final rendering for any given shaded point we can decide between which too depth sliced it is and can interpolate the opacities stored in the light volume texture slices.

### 2.5.3. Program structure

The demo application named `Hierarchical Particle Systems Demo` that demonstrates the described technique can be found in the repository. It is a small OpenGL/Glut application. It loads particle positions, scene data and animation data from a text files. It uses several simple wrapper classes (e.g.: camera, texture, render texture, Cg GPU program wrapper classes) which don't concern strictly the technique described above, so they won't be documented here. Main initializations event handling and the rendering loop is located in `main.cpp`.

One main class is responsible for implementing a shaded hierarchical particle system named `AdvancedParticleSystem`. It contains a `ParticleSystem` instance which handles particle position loading, particle emitting, animation (needed in the smaller particle system block) and sending the particles to the graphics card as quads during rendering. `AdvancedParticleSystem` is only responsible for the rendering process described above. It has functions for the main rendering tasks:

`RefreshDepths()` generates front, back depth and opacity textures of the system block

`RefreshIllumTextures()` refreshes light volume texture slices

`RenderObjectDepths()` renders scene depth

`Display()` final rendering

The demo uses four light slices stored in the four channels, so the light volume rendering can be done in a single render pass.

The main Cg shader programs used by the application:

`PSystemDensityOnlyPrograms.cg`: creates the opacity image of the block (used if no depth calculation used)

`PSystemFrontDepthPrograms.cg`: creates the opacity, front and depth image of the block. Front depth, back depth and opacity will be stored in the red, green and blue color channels respectively.

`IllumPrograms.cg`: used in light volume calculation.

`ObjDepthPrograms.cg`: renders scene depth.

`Psystem_Default.cg`: renders particles with uniform color without depth calculation.

`Psystem_Default_Depth.cg`: renders particles with uniform color with depth calculation.

`Psystem_Single_Phase.cg`: renders particles with single scattering without depth calculation (the illumination texture is not used).

`Psystem_Single_Phase_Depth.cg`: renders particles with single scattering with depth calculation (the illumination texture is not used).

`Psystem_Multiply_Forward.cg`: renders particles with multiple forward scattering without depth calculation (the illumination texture is used).

`Psystem_Multiply_Forward_Depth.cg`: renders particles with multiple forward scattering with depth calculation (the illumination texture is used).

### 2.5.4. Using the demo application

This sample application displays a cloud shaded with one light source. As the application starts a plane flies into the cloud and the camera follows it. The animation loops and can be stopped. If the animation is stopped we can control the camera ourselves. The scattering properties of the cloud can be changed interactively. The displayed cloud is made out of 100 blocks each with 400 particles resulting in a total count of 40000 modeled particles. The positions of the particles inside a block are animated which give a swirling effect to the cloud, while the positions where the blocks should be placed (particle positions of the bigger system) are read from a file.

**Controls:**

- w, s : moves camera forward and back
- a, d : slides camera left and right
- r, f : moves camera up and down
- left, right : rotates camera (also can be done with mouse while holding left button)
- F1, shift + F1 : increase, decrease cloud density
- F2, shift + F2 : increase, decrease cloud albedo
- F3, shift + F3 : increase, decrease cloud scattering symmetry
- h : change heads up display
- l : change lighting mode (see shaders above)
- i : toggle depth calculation
- p : start/stop screenshot saving
- o : start/stop plane and camera animation

*Figure 25. Screenshots made with the hierarchical particle systems demo application.*

## 2.6. POST PROCESSING EFFECTS

### 2.6.1. Glow

Glow is the effect when a very shiny object in the picture causes the neighboring pixels to be brighter than they would be normally. It is caused by scattering in the lens and other parts of the eye, giving a glow around the light and dimming contrast elsewhere in the scene. In video production, the video camera captures an image by converting photons to charge using a charge coupled device (CCD). Glow occurs in a video camera when a charge site in the CCD gets saturated and overflows into neighboring sites.

This effect can easily reconstructed on modern graphics hardware with post-processing or image-space rendering technique.

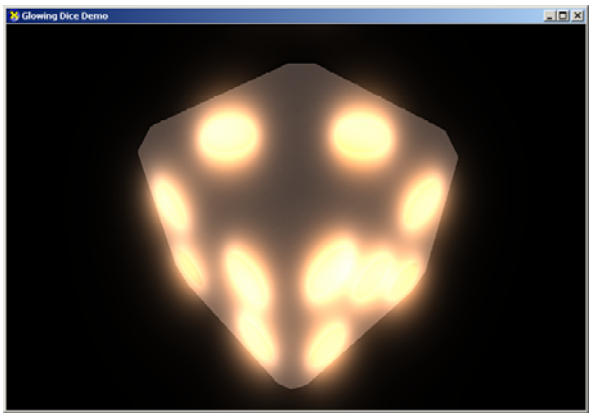



*Figure 26. Die with glow.*

#### *2.6.1.1. Post-processing method*

If the scene rendered to floating point buffer, we can use a post-processing pass to create the desired glow effect. In the buffer the pixels with high intensities represents the glowing parts of the scene. We use a low pass filter to separate these parts to another image buffer. The second step to blur this image using convolution with Gaussian kernel. This part requires the most computation. Instead of using a single $n \times n$ kernel with the appropriate values of the Gaussian distribution function, we can use separable convolution. This is a two pass convolution, in the first pass we use an $n$ pixel wide kernel to blur the pixels horizontally. The second pass is the same with the exception that we use a transposed kernel. After the blurring pass we can combine the original image with the generated glow image during or before the tone-mapping with adding the two images pixel by pixel. To the best result we need to use high dynamic range source image, otherwise we may catch too much or too less "glowing emitter" part of the image.

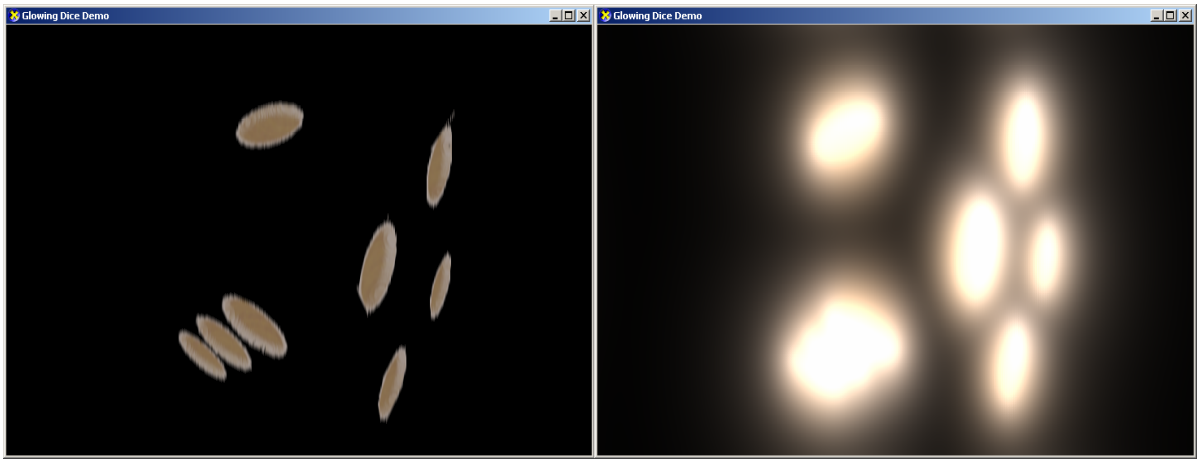



*Figure 27. Die with and without glow.*

### *2.6.1.2. Image-space method*

The image-space method is almost the same as the post-processing technique. The main advantage is that we have full control over what parts of the picture are glowing. Other advantage is that we can use this technique with LDR rendering. On the counterpart this method needs more computation than the post-processing technique. The difference is the creation of the initial glow image. In this method we use the original geometry with a special shader to render only the glowing parts. In the sample implementation we use a shader that renders normally when the texture's alpha value is 1.0f and renders black pixel otherwise.

With this shader we can precisely define where the glowing part is. After this pass we use the same Gaussian blur as the other technique. The final composition is also the same.

We can add an interesting trailing effect to the glow easily. If we store the blurred glow image we can modulate the next frame's glow image with it. We can control the long of the trail during the composition with a dimming parameter.

### *2.6.1.3. Example shaders*

To achieve the right results, we used the vertex shader presented below during all the effects. This shader modifies the texture coordinates for the right sampling.

```
struct VS_INPUT {
        float4 pos : POSITION;
        float2 tex0 : TEXCOORD0;
    };

    struct VS_OUTPUT {
        float4 pos : POSITION;
        float2 tex0 : TEXCOORD0;
    };

    uniform float4x4 modelViewProjection;
    uniform float dsWidth, dsHeight;
```

```
    VS_OUTPUT VS_VertexBlurShader(in VS_INPUT input){
        VS_OUTPUT output0;
        output0.pos = mul(input.pos,modelViewProjection);
        output0.tex0 = input.tex0 + 0.5*float2(1.0f/dsWidth, 1.0f/dsHeight);
        return output0;
    }
```

The following shader creates the glow texture. It also respects of the alpha value of the source texture.

```
texture SourceTexture;
    sampler2D SourceTextureSampler = sampler_state {
        texture = <SourceTexture>;
        mipfilter = LINEAR;
        AddressU = Clamp;
        AddressV = Clamp;
    };

    half4 PS_Glow(in float2 tex0:TEXCOORD0) : COLOR {
        half4 texLookUp;
        texLookUp.rgba = tex2D(SourceTextureSampler,tex0).rgba;

        if (texLookUp.a == 1.0f) texLookUp = half4(0,0,0,0);;
        return texLookUp;
    }
```

These two shaders blur the glow image. In this step we combine the old glow image with the newly computed one. The **GlowGain** controls the addition of the old glow image, and the **Stretch** parameter modifies the sampling points.

```
texture Glow;
    sampler2D GlowSampler = sampler_state {
        texture = <Glow>;
        mipfilter = LINEAR;
        AddressU = Wrap;
        AddressV = Wrap;
    };

    uniform float dsWidth, dsHeight;
    uniform float Stretch;

    half4 PS_Blur_h(in float2 tex0:TEXCOORD0) : COLOR {
        half4 texLookUp_h;
        texLookUp_h =    tex2D(GlowSampler, float2(tex0.x-(3.8697f*Stretch/dsWidth), tex0.y))+
                         tex2D(GlowSampler, float2(tex0.x-(1.7229f*Stretch/dsWidth), tex0.y))+
                         tex2D(GlowSampler, tex0)+
                         tex2D(GlowSampler, float2(tex0.x+(1.7229f*Stretch/dsWidth), tex0.y))+
                         tex2D(GlowSampler, float2(tex0.x+(3.8697f*Stretch/dsWidth), tex0.y));
        half4 oldGlow = tex2D(OldGlowSampler, tex0);
        return texLookUp_h + oldGlow * GlowGain;
    }

    half4 PS_Blur_v(in float2 tex0:TEXCOORD0) : COLOR {
        half4 texLookUp_v;
        texLookUp_v =   tex2D(GlowSampler, float2(tex0.x, tex0.y- .8697f*Stretch/dsHeight)))+
                        tex2D(GlowSampler, float2(tex0.x, tex0.y-1.7229f*Stretch/dsHeight)))+
                        tex2D(GlowSampler, tex0)+
                        tex2D(GlowSampler, float2(tex0.x, tex0.y+(1.7229f*Stretch/dsHeight)))+
                        tex2D(GlowSampler, float2(tex0.x, tex0.y+(3.8697f*Stretch/dsHeight)));

        half4 oldGlow = tex2D(OldGlowSampler, tex0);
```

```
        return texLookUp_v + oldGlow * GlowGain;
    }
```

The final shader combines the original image with the glow image to create the final image.

```
texture Original;
    sampler2D OriginalSampler = sampler_state {
        texture = <Original>;
        mipfilter = LINEAR;
        AddressU = Wrap;
        AddressV = Wrap;
    };

    float4 PS_Final(in float2 tex0:TEXCOORD0) : COLOR {
        float4 Orig = tex2D(OriginalSampler,tex0).rgba;
        half4 Glow = tex2D(GlowSampler,tex0).rgba;
        return Orig + Glow;
    }
```

### 2.6.2. Star effect

This effect differs in the blurring part from the glow effect. Instead of creating a flowing aura this effect creates a "star" shape around the light spots of the source image. To generate the "emitting" image we can use both of methods described in the previous effect. The difference is in the blur process. Instead of using Gaussian blur we use directional blur. To create the star shapes we need at least four blur pass, each for one direction. The final composition is a simple addition of the source and "starry" images.

### 2.6.3. Lens flare

When we see a light source in low light conditions (eg. the lamps along the road at night), we can see flaring around the light emitter. This flare is composed of a lenticular halo and a ciliary corona caused by the camera lens or the human eye.

Rays of the ciliary corona appear as radial streaks emanating from the center of the source. Similar ray patterns associated with other coronas have been studied by physicists and are caused by random fluctuations in refractive index of the ocular media.

The lenticular halo is observed as a set of colored, concentric rings, surrounding the light source and distal to the ciliary corona. The irregular rings are composed of radial segments, where the color of each segment of the ray varies with its distance from the source. The apparent size of the halo is constant and independent from the distance between the observer and the source.

This phenomenon is caused by the radial fibres of the crystalline structure of the lens.

We can use billboards to simulate the lens flare affect. The billboards are spaced along the line between the center of the screen and the light source. The intensity of the billboards weakens by the distance.

Off the shelf monitors can control the intensity just in a limited, low dynamic range (LDR). Therefore the values written into the frame buffer are unsigned bytes in the range of [0,255]. However, realistic

rendering involves the reproduction of high dynamic range (HDR) luminance values on the computer screen. The variation in light levels we experience in the real world environment is vast. The average luminance in an outdoor scene can be 100 million times greater during a shiny day than at night. The dynamic range of luminances can also be large from shadows to highlights. So the luminance levels can change dramatically over time and from place to place.

Vision functions can adapt to these changes fairly quickly. The adaptation mechanism based on the pupil, the rod and cone receptors, photopigment bleaching and regeneration and neural gain control. Vision is not equally good under all conditions. The daylight scene appears very bright and colorful, but under low light conditions like at night or in a dark room everything looks greyish and darker.

The conversion of HDR image values to displayable LDR values is called tone mapping. The perceptual effects must be taken into account during the tone mapping process.

### 2.6.4. Tone mapping operators

The tone mapping can be categorized according to how they map the HDR data to LDR. The two main groups are the global and local operators. Global operators apply a single mapping function to the whole image, whereas the local operators modulate the mapping depending on different parts of the image.

The other distinction is between perceptually and empirical operators. Perceptually based operators strive to produce images that are predictive visual simulations of scene appearance. Empirical operators seek to meet criteria such as detail preservation, artefact elimination, or dynamic range compression.

Third distinction is between dynamic and static operators. Static operators are developed for still images. The dynamic operators are designed for processing image streams.

### 2.6.5. Ward's visibility preserving operator

The main goal of this operator to produce displayable images that accurately present the threshold visibility of scene features. To do this, Ward's operator uses histogram adjustment constrained by a visual adaptation model.

This operator contains several stages. In the first stage the input image is downsampled to create a low resolution “foveal image” in which each pixel covers $1^o$ of visual angle. In the second stage a visual model is applied on this image to simulate the effects of veiling glare and low-luminance color and acuity loss. Next a histogram adjustment algorithm is applied to define the mapping from scene luminances to display luminances. This histogram adjustment is constrained by the human threshold versus intensity function.

This operator is a simple but elegant solution to the perceptually based tone mapping problem. However, it is a static operator, designed to use on single images, and it is unable to correctly represent the changes in scene visibility over time.

#### *2.6.5.1. Pattanaik's time dependent operator*

To address the issues of Ward's operator on image streams, Pattanaik developed a new operator. This operator incorporates a model of the temporal dynamics of vision. The goal is to process a stream of input images of the scene and produce an output stream that simulates the changes in visual appearance caused by variations in scene luminance. To accomplish this, the operator merges components of an advanced color appearance model with psychologically based model of temporal dynamics of visual adaptation.

At the highest level the operator consists of a forward and inverse pair of visual models. First, an adaptation model transforms input scene intensities into retina-like responses. After that, these responses are transformed by a simplified version of Hunt's color appearance model, to produce a representation of the scene's suprathreshold “blackness/whiteness” and “colorfulness” appearance correlates. To complete the tone mapping process, the adaptation state of the display observer is determined, and the inverse appearance model transforms the appearance correlates into display values that are calculated to produce corresponding responses in the display observer.

The time dependent features of the operator are derived from the forward adaptation model. In the neural visual system both neural and photochemical adaptation mechanism are responsible for altering the sensitivity of the rod and cone systems and shifting the S-shaped response profiles across the luminance range. Neural adaptation is a fast and symmetric process that can alter sensitivity within milliseconds, however the magnitude of this effect is limited. Photopigment bleaching and regeneration can have much greater impact on sensitivity, but it is an asymmetric process, with a relatively fast bleaching followed by a slow regeneration and recovery of sensitivity. Pattanaik models the dynamics of the adaptation with four low-pass exponential filters.

This operator is one of the most advanced perceptually based operators for tone mapping in image streams. Despite, it still has limitations. It uses a suprathreshold color appearance model rather than a threshold visibility model so the images produced may not accurately represent visibility. It produces a simple S-shaped global mapping function so high dynamic range scenes may not be mapped correctly.

### 2.6.6. A real-time tone mapping operator

The algorithm proposed by Reinhard et al. operates solely on the luminance values which can be extracted from RGB intensities using the standard CIE XYZ transform (D65 white point). The **Y** component of the **XYZ** vector is the luminance, so we can compute it with the following equation:

$$Y = 0.2126 * R + 0.7132 * G + 0.0722 * B.$$

The method is composed of global a scaling function and local dodging and burning techniques, which allow to preserve fine details. The results are driven by two parameters: the adapting luminance for the HDR scene and a key value. The adapting luminance ensures that the global scaling function provides the most efficient mapping of luminance to the display intensities for given illumination conditions in the HDR scene. The key value controls whether the tone mapped image appears relatively bright or relatively dark.

The source luminance values **Y** are first mapped to the relative luminance $Y_r$:

$$Y_r = \alpha * Y / Y$$

where Y is the logarithmic average of the luminance in the scene, which is an approximation of the adapting luminance, and alpha is the key value. The relative luminance values are then mapped to the displayable pixel intensities L using the following function:

$$L = Y_r / 1+Y_r.$$

The above formula maps all luminance values to the [0:1] range in such way that the relative luminance $Y_r = 1$ is mapped to the pixel intensity $L = 0.5$. This property is used to map a desired luminance level of the scene to the middle intensity on the display. Mapping higher luminance level to middle gray results in a subjectively dark image whereas mapping lower luminance to middle gray will give a bright result. Images which we perceive at low light condition are relatively dark compared to what we see during a day. We can simulate this impression by modulating the key value with respect to the adapting luminance in the screen.

This tone mapping function may lead to the loss of details in the scene due to the extensive contrast compression. Reinhard et al. propose a solution to preserve local details by employing a spatially variant local adaptation value V in the second equation:

$$L(x,y) = Y_r (x,y) / 1+V(x,y),$$

where x,y are the pixel coordinates.

The local adaptation V equals to an average luminance in a surround of the pixel. The problem lies however in the estimation of how large the surround of the pixel should be. The goal is to have as wide surround as possible, however too large area may lead to well known inverse gradient artefacts, halos. The solution is to successively increase the size of a surround on each scale of the pyramid, checking each time if no artefacts are introduced. For this purpose a Gaussian pyramid is constructed with successively increasing kernel. The Gaussian for the first scale is one pixel wide, setting kernel size to $s = 1 / (2\sqrt{2})$, on each subsequent scale s is 1.6 times larger.

### *2.6.6.1. Temporal Luminance adaptation*

While tone mapping the sequence of HDR frames, it is important to note that the luminance conditions can change drastically from frame to frame. The human vision reacts to such changes through the temporal adaptation processes. The time course of adaptation differs depending on whether we adapt to light or to darkness, and whether we perceive mainly using rods (during night) or cones (during a day). To take into account the adaptation process, a filtered $Y_a$ value can be used instead of the actual adapting luminance Y. The filtered value changes according to the adaptation processes in human vision, eventually reaching the actual value if the adapting luminance is stable for some time. The process of adaptation can be modeled using an exponential decay function:

$$Y^{new}_a = Y_a + (Y - Y_a) (1 - e^{-T/\tau})$$

where T is the discrete time step between the display of two frames, and $\tau$ is the time constant describing the speed of the adaptation process. These time constants are different for rods and cones:

$$\tau_{rods} = 0.4s, \tau_{cones} = 0.1s$$

Therefore, the speed of the adaptation depends on the level of the illumination in the scene. The time required to reach the fully adapted state depends also whether the observer is adapting to light or dark

conditions. The above numbers describe the adaptation to light. The full adaptation to dark takes up to tens of minutes, so it's not simulated.

#### *2.6.6.2. Scotopic vision*

On low light conditions only the rods are active, so color discrimination is not possible. The cones start to loose sensitivity at 3.4cd/$m^2$ and become completely insensitive at 0.03cd/$m^2$ where the rods are dominant. We can model the sensitivity of rods with the following equation:

$$\sigma(Y) = 0.04/0.04 + Y$$

where Y denotes the luminance. The value sigma = 1 describes the monochromatic vision and sigma = 0 the full color discrimination.

#### *2.6.6.3. Veiling Luminance*

Due to the scattering of light in the optical system of the eye, sources of relatively strong light cause the decrease of contrast in their vicinity. Such an effect cannot be naturally evoked while perceiving an image on a display due to different viewing conditions and limited maximum luminance of such devices. It is therefore important to account for it while tone mapping.

The amount of scattering for a given spatial frequency ρ under a given pupil aperture *d* is modelled by an Ocular Transfer Function:

$$OTF(\rho, d) = e^{-\rho/20.9 - 2.1*d} \quad 1.3 - 0.07 * d$$

where $d(Y) = 4.9 - 3 \tan (0.4 \log_{10} Y + 1)$.

### 2.6.7. Implementation

*Figure 28. Original image, dark mapped, normal mapped, bright mapped images*

#### *2.6.7.1. The key value*

The key value determines, whether the tone mapped image appears relatively dark or bright. The exact value can be left as user choice, or it can be estimated automatically based on the relations between minimum, maximum, and average luminance in the scene. Although the result of this method is appealing, this solution does not necessary correspond to the impressions of everyday perception. The critical changes in the absolute luminance values may not always affect the relation between the three values. This may lead to dark night scenes appearing too bright and very light too dark.

Krawczyk proposed an empirical method to calculate the key value. His method is based on the absolute luminance. Since the key value was introduced in photography, there is no scientifically based experimental data which would provide an appropriate relation between the key value and the luminance. The low key is 0.05, the typical choice for moderate illumination is 0.18 and 0.8 is the high key. Krawczyk empirically specified key values for several illumination conditions and interpolated the rest using the following formula:

$$\alpha(Y) = 1.03 - 2/2+\log_{10}(Y + 1)$$

where $\alpha$ is the key value and Y is an approximation of the adapted luminance.

### *2.6.7.2. Hardware implementation*

In order to perform tone mapping with perceptual effects, we need to compose two maps: a local adaptation map for the tone mapping, and a map of light scattering in the eye for the glare effect. We will refer to these maps as perceptual data. Because different areas of these maps require different spatial processing, they cannot be constructed in one rendering pass. Instead, we render successive scales of the Gaussian pyramid and update the maps by filling in the areas for which the current scale has appropriate spatial processing. In the last step we use these maps to compose the final tone mapped result.

We start with calculating the luminance from the HDR frame and mapping it to the relative luminance. We calculate the logarithmic average of the luminance Y in the frame using the down sampling approach described in Goodnight's paper, and apply the temporal adaptation process. The map of relative luminance values constitutes the first scale of the Gaussian pyramid. At each scale, we render the successive scale by convolving the previous scale with the appropriate Gaussian. The convolution performed in two passes: one for the horizontal and one for the vertical convolution. Having the current and the previous scales, we update the perceptual data on a per pixel basis in a separate rendering pass. The local adaptation is computed using the measure of the difference between the previous and the current scale as described in Reinhard's paper. For the glare map, we first estimate the proper scale for the luminance of the current pixel. It depends on the adapting luminance and it is uniform for the whole frame so we supply it as a parameter to the fragment program. Before descending to the next scale of the Gaussian pyramid, the texture containing the current scale becomes the previous scale, and the texture with the current set of the perceptual data becomes the previous set.

After descending to the lowest scale of the Gaussian pyramid, the perceptual data texture is complete. In the final rendering step, we tone map the HDR frame and apply the perceptual effects with the equation $L(x,y) = (Y_r + Y_{glare})/(1+V(x,y))$, where L is the final pixel intensity value, $Y_r$ the relative luminance, $Y_{glare}$ is the amount of additional light scattering in the eye, and V is the local adaptation map. We account for the last perceptual effect, the scotopic vision, while applying the final pixel intensity value to the RGB channels in the original HDRframe.

### *2.6.7.3. Shaders*

We use the following declarations and vertex shader code all across the tone mapping process:

```
struct VS_INPUT {
    float4 Position : POSITION;
    float  TexCoord : TEXCOORD0;
};

struct VS_OUTPUT {
    float4 hPosition: POSITION;
    float  TexCoord : TEXCOORD0;
};

uniform float4x4 ModelViewProj;

VS_OUTPUT VertexShader(VS_INPUT IN) {
    VS_OUTPUT OUT;
    OUT.hPosition = mul(IN.Position, ModelViewProj);
    OUT.TexCoord = IN.TexCoord;
    return OUT;
}
```

The pixel shader computing the luminance for each pixel is:

```
texture SourceTexture;

sampler2D SourceTextureSampler = sampler_state {
    texture = <SourceTexture>;
    mipfilter = LINEAR;
    AddressU = Wrap;
    AddressV = Wrap;
}

float PS_Luminance(in float2 TexCoord : TEXCOORD0) : COLOR {
    float3 color = tex2D(SourceTextureSampler, TexCoord).rgb;
    float avg = dot(color, float3(0.21f, 0.39f, 0.4f));
    return avg;
}
```

In order to downscale the luminance image, Gaussian filter is used. For performance reasons, the convolution is executed separately for the two directions (this is possible because of the separability of the Gaussian filter), thus this shader should be called multiple times to obtain a 2D convolution:

```
texture LuminanceTexture;

sampler2D LuminanceTextureSampler = sampler_state {
    texture = <LuminanceTexture>;
    mipfilter = LINEAR;
    AddressU = Wrap;
    AddressV = Wrap;
}

half4 PS_DownScale_H(in float2 tex0:TEXCOORD0) : COLOR {
    half4 texLookUp;
    texLookUp = tex2D(LuminanceTextureSampler, float2(tex0.x-(3.8697f/Width), tex0.y)).r +
                tex2D(LuminanceTextureSampler, float2(tex0.x-(1.7229f/Width), tex0.y)).r +
                tex2D(LuminanceTextureSampler, tex0).r +
                tex2D(LuminanceTextureSampler, float2(tex0.x+(1.7229/Width), tex0.y)).r +
                tex2D(LuminanceTextureSampler, float2(tex0.x+(3.8697f/Width), tex0.y)).r;

    return half4(texLookUp / 5, 0, 0, 1.0f);
}

half4 PS_DownScale_V(in float2 tex0:TEXCOORD0) : COLOR {
    half4 texLookUp;
    texLookUp = tex2D(LuminanceTextureSampler, float2(tex0.x, tex0.y-(3.8697f/Height))).r +
                tex2D(LuminanceTextureSampler, float2(tex0.x, tex0.y-(1.7229f/Height))).r +
                tex2D(LuminanceTextureSampler, tex0).r +
                tex2D(LuminanceTextureSampler, float2(tex0.x, tex0.y+(1.7229f/Height))).r +
                tex2D(LuminanceTextureSampler, float2(tex0.x, tex0.y+(3.8697f/Height))).r;

    return half4(texLookUp / 5, 0, 0, 1.0f);
}
```

The final pass takes the average luminance value of the neighborhood and scales the color accordingly:

```
texture AvgLumiTexture;
sampler2D AvgLumiTextureSampler = sampler_state {
    texture = <AvgLumiTexture>;
    mipfilter = LINEAR;
    AddressU = Wrap;
```

```
    AddressV = Wrap;
}

//--------------------------------------------------------------------------------
float4 PS_Final(in float2 TexCoord : TEXCOORD0) : COLOR {
    float Key = 1.03f - 2 / (2 + log10(tex2D(AvgLumiTextureSampler, TexCoord).r + 1));
    float RelLum = Key * tex2D(LuminanceTextureSampler, TexCoord).r /
                         tex2D(AvgLumiTextureSampler, TexCoord).r;
    float Lum = RelLum / (1+RelLum);
    float4 Color = tex2D(SourceTextureSampler, TexCoord) * Lum;
    Color = pow(Color * float3(1.05, 0.97, 1.27), 1.0 / 2.2);
    return Color;
}
//--------------------------------------------------------------------------------
```

### *2.6.7.4. Tone mapping standalone application and its Ogre integration*

The glowing dice program demonstrates the glow and the tone mapping technique. The various parameters controlled by the on-screen sliders. The tone map has one control to adjust the automatically determined key value.

The glow technique has gain and stretch parameters. The gain controls the participation of the glow map to the final image, and the stretch parameter modify the Gaussian kernel's sample points.

These techniques are ported to the Ogre3D framework. The 1.2 release has a new so called Compositing framework, so these effects are converted to compositor and material scripts.

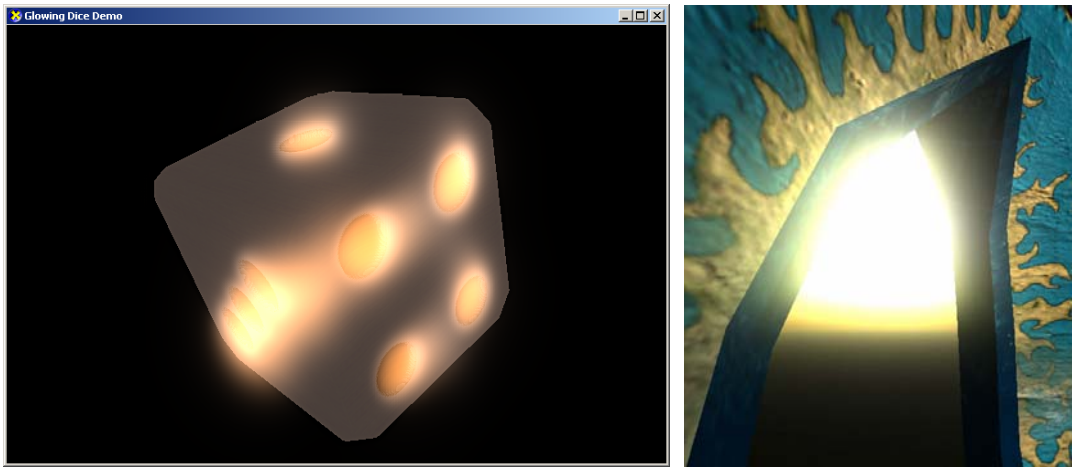



*Figure 29. Glow effect in the standalone module and integrated into Ogre.*

## 2.7. LIGHT PATH MAPS

### 2.7.1. Rendering to texture atlases

This section describes a solution to preprocess geometry so that seam artifacts of texture atlas renderings are eliminated. This is not considered to be a module in itself, but it is a prerequisite for tools using texture atlases. Path Maps, described in the section 2.6, are based on texture atlases, and utilize the solution described here.

Render-to-texture has become a very basic operation in graphics hardware programming, allowing for computations to be decomposed into consequent passes. Most prominently, all kinds of lighting algorithms, including stochastic iteration, photon maps, caustics, or light animation can be transposed into the two-dimensional surface domain by using texture atlases. Computations are carried out using a render-to-atlas operation, and data is later accessed by texturing. However, the rasterization algorithm supported by the hardware does not necessarily trigger rendering to every texel that can be addressed by surface texture coordinates. This causes artifacts near texturing seams. In order to eliminate them, we have to simulate conservative overestimated rasterization of triangle charts when rendering to the texture atlas. In the standalone illumination package, the Path Map and the Ray Tracing effects modules use texture atlases. In the Path Map module, we will use the technique described in this chapter.

In this section, we describe a method that processes the original model geometry and constructs a minimal set of line segments that will rasterize to complete the texture atlas. There is no increase in the complexity of the geometry, and the overlying computation algorithm does not have to be modified.

#### *2.7.1.1. Texture atlases*

The most straightforward and often unavoidable means to pre-compute and store illumination data is the usage of texture atlases. Whenever we need to know the illumination of surface elements without fixing a specific viewing direction, we may carry out computations at sample points of the surface: these will be the texels of our texture atlas. In case of a fixed light position, we may also pre-compute a shadow map to speed up lighting, but it may not be as effective as the using the atlas. Furthermore, if we wish to pre-compute inter-reflections independent of both light position and viewing direction, the use of a texture atlas is imperative. Similarly, if we have to blend impostors on surfaces, like in the caustic generation method using approximate ray tracing (see the Ray Tracing Effects chapter), it also has to be performed in the texture atlas domain.

An important feature of textures is the possibility of filtering. While a texture atlas may not sample surface points very densely, bilinear interpolation of pre-computed values comes practically for free. Still, quality will depend on the resolution, and poor sampling may appear as texturing artifacts. However, similar effects appear for any algorithm that reuses stored data.

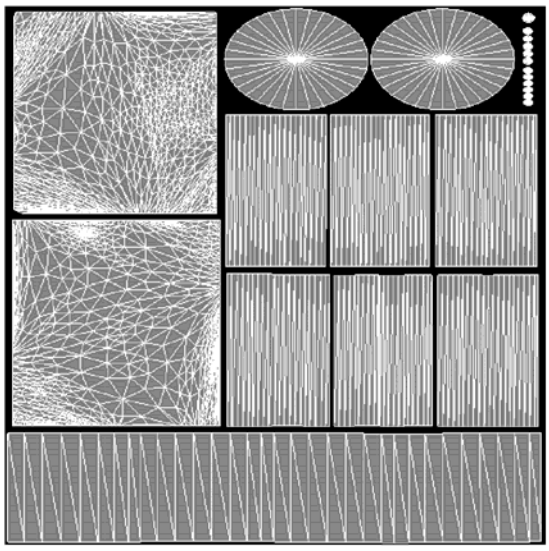


*Figure 30. An atlas texture with triangle edges. The mesh is decomposed to triangle charts.*

A texture atlas has to ensure a unique mapping of texture coordinates to surface points. While some of the texture space may be empty, every surface point has to have unique texture coordinates. It is also desirable to avoid discontinuities. Generally, the texture atlas mapping for a triangular mesh will contain a number of flattened charts of adjacent triangles, separated by a few texels wide empty margin. The contours of these charts are the seams of the mapping: that is where adjacent surface points are mapped to distant coordinates. Seams are usually unavoidable without extremely distorting triangles. There are several approaches to generate atlas mappings with as few seams as possible or to find those seams that are visually least disturbing. While these approaches are very useful to reduce artifacts, it cannot always be assumed that the virtual world model an algorithm has to work on is practically seamlessly textured.

#### *2.7.1.2. Render to texture*

Using a texture atlas involves two basic steps. The first one is to render the triangles of the model onto the texture atlas, to invoke vertex and pixel shaders that compute illumination data and output it to atlas texels. This is the render-to-atlas operation, which is simply performed by issuing a draw call with the original vertex and index buffers of the mesh. In the vertex shader, however, the vertices are displaced to their respective texture atlas coordinates, transforming the triangles to texture space. Theoretically, the pixel shaders should be invoked for all valuable texels of the atlas, filling it with the desired data. Having computed the atlas, it can be accessed from the pixel shaders of later passes, typically addressed by texture coordinates interpolated between vertices. The problem this chapter addresses is how to ensure that these texture reads will always address a texel for with the render-to-atlas has been performed and thus contains valid information.

#### *2.7.1.3. Triangle rasterization*

How to convert render primitives (points, line segments and polygons) to screen pixels effectively was exhaustively researched by pioneers of computer graphics. Variant algorithms to render smooth anti-aliased edges, or to include pixels only touching the area of primitives were also invented. However,

only the most effective and clear-cut variant could make it to current incredibly fine-tuned graphics hardware. The issue of anti-aliasing is more effectively solved by the multi-sampling post-filtering approach. On the other hand, this specialization forces a need for workarounds when we need a different rasterization logic.

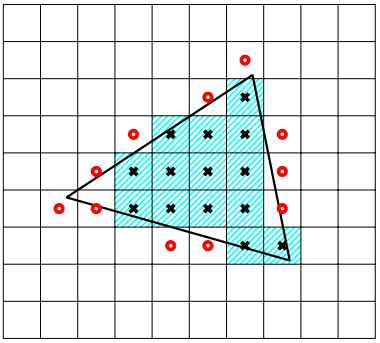


*Figure 31. Hardware supported triangle rasterization. Pixels marked with circles are not colored, even though they overlap with the triangle*

The general rule for rasterizing a triangle is to color those pixels, whose center is within the triangle (Figure 31). With adjacent triangles, this ensures there are no holes and no doubly colored pixels in the image. However, if we apply this rasterization rule when rendering to the texture atlas, there may be some texels whose center is not within the triangle, but they are overlapping with it. The texture coordinates in these areas will address a texel which was not considered by the rasterization, and therefore the pixel shader computing the value that should be there was not invoked. These texels will appear as non-initialized or black blocks or stripes near the seams of the texture atlas. These cases can be avoided by very careful texturing, if the seams are all horizontal or vertical in texture space, but they will always appear in the general case of slant-edged triangle mesh atlases. Furthermore, if the texture is read using linear filtering, these non-initialized values will influence and even larger area.

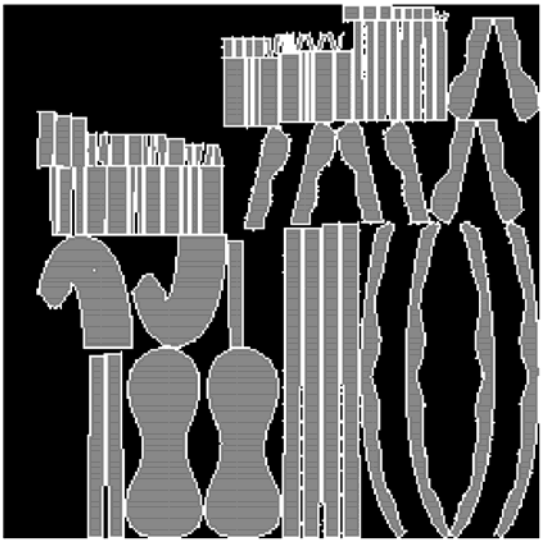


*Figure 32. A texture atlas. Gray texels are rasterized by conventional rasterization. White texels should also be computed.*

A simple but neither effective nor accurate method is to filter the texture atlas, and copy valid neighbouring texel values to non-initialized ones. This approach is really easy to implement, but it needs a full-screen rendering of the texture atlas, invoking a moderately expensive pixel shader using a number of texture reads for every texel. The overhead may be critical for a real-time algorithm. Furthermore, it is impossible to plausibly decide which neighbour's value is actually valid for a texel without knowledge of the geometry. Usually the heuristics of choosing the maximum is applied to keep visual artifacts low.

### *2.7.1.4. The render to texture algorithm*

Looking at the texture atlas, we can realize that we actually need to extend the contours of the charts of adjacent triangles. Instead of exchanging all the triangles with more complex extended polygons, we only need to rasterize a few line segments along the seams. In order to do this, we need to identify those edges of the model mesh which are duplicated and placed on seams, and find out how to offset the line segments to augment the triangles to cover all the pixels the triangle is overlapping with.

A line segment is rasterized according to the principles set by the DDA or the Bresenham algorithms. Cases are separated depending on whether the line segment is closer to the vertical or the horizontal direction. Then, in every line or every column respectively, that pixel is colored the center of which is closest to the line (Figure 33). We always measure distances on the scan line or the pixel column, thus we can state that pixels whose center is within the half-a-pixel distance form the line will be colored.

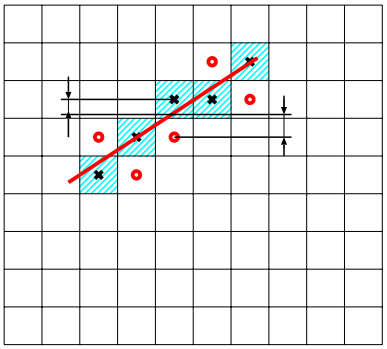


*Figure 33. Line segment rasterization The pixel whose center is nearest to the line is colored is every column of pixels.*

Fortunately, albeit not by coincidence, this produces the exact same pattern as the one that appears at the edge of a rasterized triangle. In order to extend the triangle by one pixel, we can simply draw a line next to it. This line has to cover those pixels, the center of which is not within the triangle, but is within the one pixel distance. As line rasterization will color the pixels within half-a-pixel distance to the line, we have to offset the edge exactly by half a pixel. Therefore, the vertices serving as the endpoints of the line segment can be derived from the original triangle vertices by offsetting their texture coordinates by an amount corresponding to half a pixel, in the positive or negative u or v direction, depending on the orientation of the edge (Figure 34).

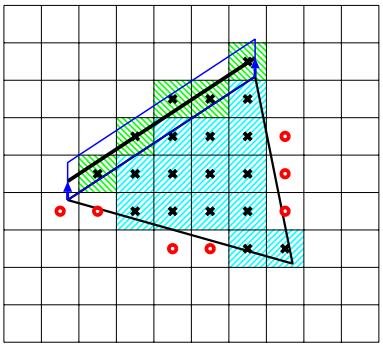


*Figure 34. Line segment drawn to rasterize overlapping pixels along edge.*

It is possible that two consecutive edges along the contour of a chart have different orientations, and the pixels of offsetted edges do not cover the corner (Figure 35). To avoid missing these small areas, the edge should be extended by one pixel measured perpendicular to the offset direction at both ends.

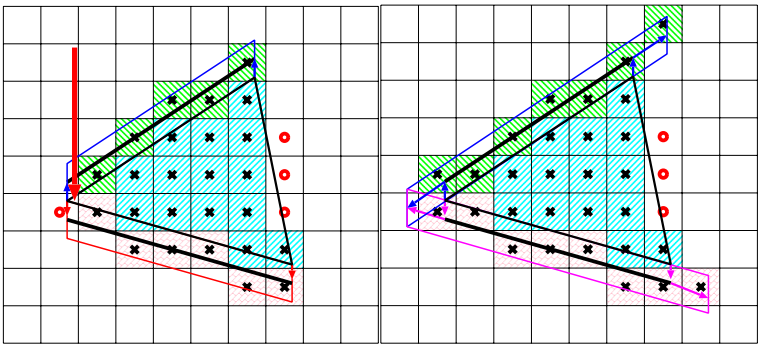


*Figure 35. Non-covered area may appear at corners where two differently aligned edges meet (above). Extending the edge line segment to cover corner pixels (below).*

Thus we acquire a supplemental set of geometry made up of line segments. Whenever rendering the original geometry, we can also render the supplemental geometry as a vertex buffer describing line primitives. As all vertex data are derived from the original triangle vertices, and the texture coordinates have been manipulated, there is no need to change the shaders. Neither the vertex shader nor the pixel shader will experience any difference from what happens when rendering the original triangles.

### *2.7.1.5. Finding the seam edges*

In order to assemble the geometry of offsetted seam edges, we have to identify such edges in the mesh texture mapping. In mesh description file formats or data structures, seam vertices are already replaced by two identical vertices with different texture coordinates, referenced by different triangles. Adjacency of triangles is usually not stored. However, we may build an adjacency graph based on the index buffer, disregarding the fact that duplicated edges actually would connect adjacent triangles. What we get is exactly the adjacency graph of the mesh charts in the texture domain. For this graph it is easy to identify those triangles that miss a neighbour, and thus, the edges that are on the seams of the texture atlas. We also need the texture coordinates of the third vertex of the triangle, to be able to find the proper offset direction for the extra line segment.

The program code assembling a vertex buffer of seam edges for a D3D mesh can be found in the Path Map tool, described in the following section. The method responsible is `PathMapEffect::RenderMesh::buildEdgeVertexBuffer()`. When rendering to the texture atlas, not only the mesh, but the edges are also rendered.

### *2.7.1.6. Results of the render to texture application*

The approach has been effective at eliminating artifacts, without any measurable performance impact. Even when the texture atlas is accessed using bilinear filtering, the undesired influence of black pixels in gaps between triangle charts has vanished.

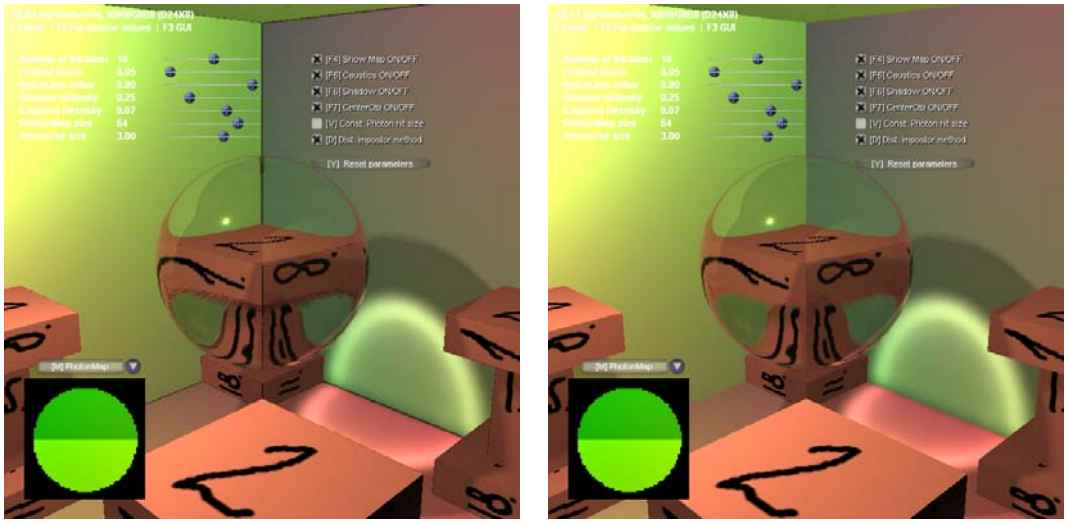


*Figure 36. Seam artifacts in the Ray Tracing Effects module, and a correct image rendered using the seam edge buffer.*

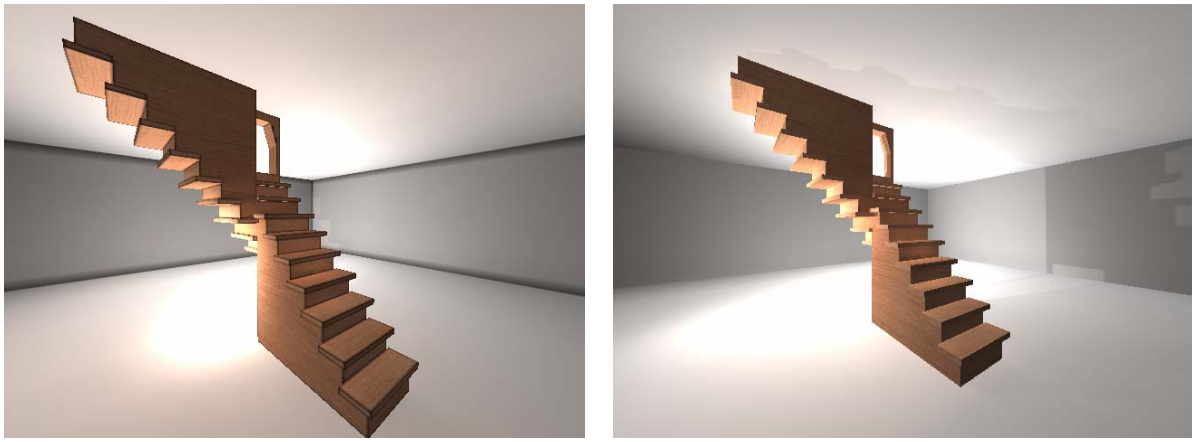


*Figure 37. Seam artifacts in the Light Path Maps module, and a correct image rendered using the seam edge buffer.*

### 2.7.2. The role of the Path Map tool

The **Path Map** standalone module implements the **Light Path Map** approach for indirect lighting, as was described previously in the GameTools technical documentation deliverable. Its purpose is to add realism to computer games by computing dynamically changing indirect illumination. It is an improvement over ambient lighting or static light maps. Not unlike light maps, global illumination computations are performed in a precomputing step, using ray casting and indirect photon mapping (the virtual light sources method). The contributions of virtual light samples are computed on the GPU, with depth mapping. However, instead of computing a single light map, multiple texture atlases (constituting the PRM) are generated for the scene objects, all corresponding to a cluster of indirect lighting samples. Then these atlases are combined according to actual lighting conditions. Weighting

factors depend on how much light actually arrives at the sample points used for PRM generation. This computation is also performed in a GPU pass.

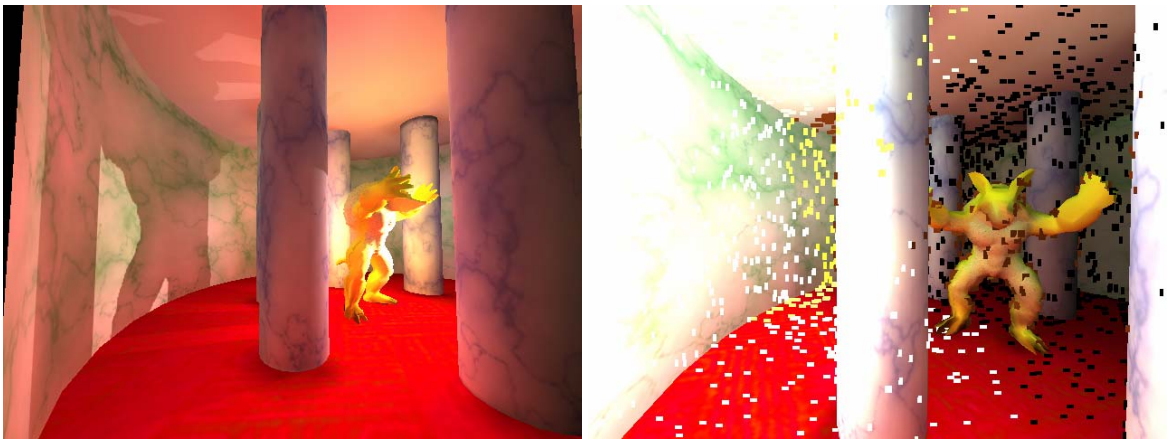


*Figure 38. Marble chamber scene rendered in the Light Path Maps module. Entry points are also displayed in the image on the right.*

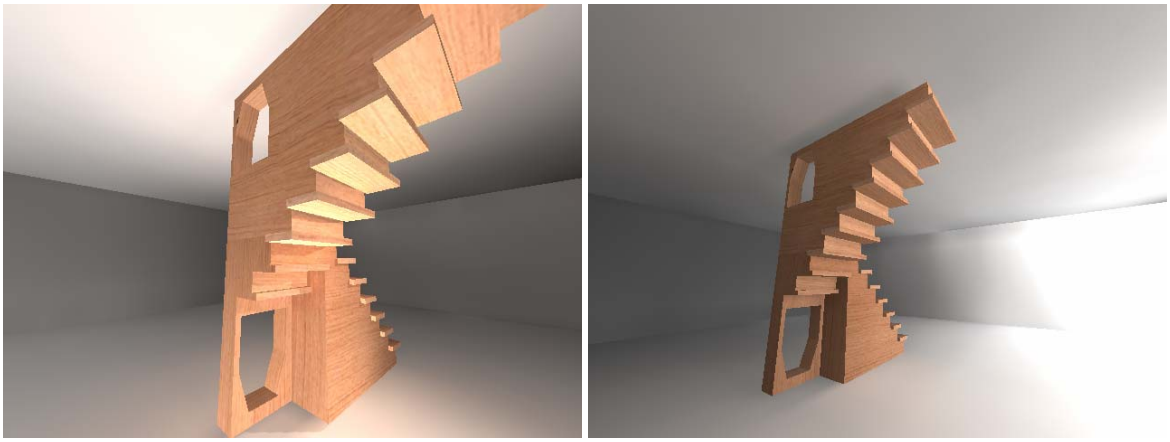


*Figure 39. Escher steps scene rendered in the Light Path Maps module. Indirect shadows and color bleeding is observable.*

The final result will be a plausible rendering of indirect illumination. Indirect shadows and color bleeding effects will appear. Indirect illumination will change as the light sources move. However, this comes at the price of fetching data from all PRM texture panes (clusters of indirect illumination) instead of just fetching a light map color. This limits the number of panes we can use. The accuracy will depend on a number of factors:

- First of all, the number of samples (entry points). Increasing this number will make preprocessing longer, but not influence rendering times.
- Then, the number of PRM panes (independent entry point clusters). In the standalone implementation this number is 32, which provides pretty accurate, lighting dependant indirect illumination, but may not be affordable if the approach is combined with other effects.

Decreasing this number will still produce nice indirect illumination, although with less fidelity to the changes of actual lighting.

- The resolution of the PRM atlas is of course important. In the program this is 256 x 256 for every pane, requiring 4 Mbytes of video RAM for every static object. As indirect illumination tends to be low frequency, this resolution is usually sufficient. However, large objects will have large texels, and should be avoided. Large objects like complete levels should be separated into smaller ones like rooms. This will also make it possible to use a large number of indirect illumination clusters, but only store and use some of them for individual objects.

The program performs two tasks: compute the PRM, and use it to display the scene with indirect illumination. The PRM, along with the location of the sample points, is saved to files, and does not need to be computed every time. When this approach will be integrated into a game engine, only the final rendering must be implemented. This tool should be adapted to process the level description format used, and compute the PRM textures for the static geometry. The textures and sample point data should be associated with the levels and their entities, and applied in the game engine.

### 2.7.3. Program structure of the path map tool

#### *2.7.3.1. Program resources*

The `PathMap` application is a DirectX program. It loads meshes for its scene from the .x file format. Class `PathMapEffect` manages all resources necessary to represent the scene entities, compute the PRMs and render to the screen. Shaders are contained in effect (.fx) files, which *PathMap.fx* referencing all the others using `#include` directives. As rendering the PRMs is a combined ray-tracing and GPU rendering task, a representation of all meshes and entities that supports the ray intersection operation is also maintained.

Therefore, `PathMapEffect` contains the following types of resources:

Scene description:

- `materialTextures`: a vector of material textures
- `renderMeshes`: a vector of meshes
  - `mesh`: a D3D mesh
  - `materials`: an array of D3D materials of submeshes
  - `textures`: an array of references to the material textures of submeshes
  - `rayTraceMesh`: a raytraceable `TriangleMesh`
  - `rayTraceMaterial`: a material used for raytracing
- `entities`: a vector of entities
  - `renderMesh`: a reference to a mesh

- `modelWorldTransform`, `inverseTransposedModelWorldTransform`: modeling transformations
- `rayTraceEntity`: a raytraceable entity, referencing the associated mesh's `TriangleMesh`

PRM:

- `bushStarters`: a vector of entry points (Entry points are primary sample points on the surface. As they are the starting points of a bush of light paths, they will act as virtual light sources, just like any node of a light path. Just like all virtual light sources, they are referred to in the code as 'radions', or, being at the root of a radion bush, 'bush starters')
- `clusterLength`: an array of cluster length (the vector of entry points is sorted so that it can be divided into cluster of samples near to each other. How long each cluster is, is given in this array.)
- `radionTexture`: a texture containing entry point positions and surface normals
- in every entity in the vector of entities (`entities`)
  - `nearClusterIndices`: an array of cluster indices (A large level geometry may require a higher number of clusters, but most of them will not influence a given object. This array contains the indices of those clusters, which should be computed and used for the illumination of this entity.)
  - `prmTexture`: a PRM texture (The PRM texture contains mini-lightmap atlases, tiled next to each other. These atlases (the PRM panes) correspond to those entry point clusters, which have been listed in the array of near cluster indices for this entity.)

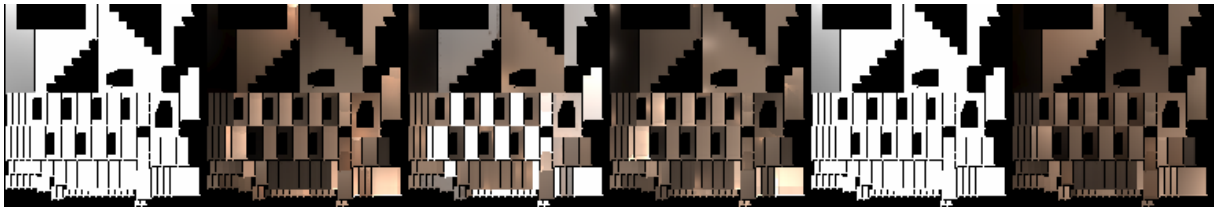


*Figure 40. The first few tiles of a PRM texture. Illumination corresponding to clusters of entry points is stored in tiled atlases.*

Resources for PRM computation (Technique *BushToAtlas*, *Depth*):

- in every mesh (`renderMeshes`)
  - a vertex buffer of line primitives ( Described in detail in the previous chapter 2.5.) This is a set of lines, that should be rendered along with the mesh when rendering to a vertex buffer. These are the edges along texture atlas seams, offset by half a pixel to assure valid values along texture atlas edges. However, care must be taken, because several pixels may be rendered to multiple times, which is a problem when using blending.)

- `prmBlendingDepthStencilBuffer`: a PRM-texture-sized stencil buffer (When rendering to the PRM atlas, we will use blending. We have to avoid rendering to the same pixel multiple times for the same virtual light source. We use the stencil buffer for this.)
- `depthMapTexture`: a depth texture (When rendering the contribution of virtual light sources into PRM atlases, we need to accont for shadows. For every virtual light source, we will prepare a depth map, and use it when rendering its illumination. The same depth texture is reused for every light source.)

Resources for indirect illumination rendering (Techniques *ComputeWeights*, *Walk*, *Depth*):

- `depthMapTexture`: a depth texture (used for depth shadow mapping of direct illumination and weight computation)
- `weightsTexture`: a weights texture (a render target to compute the form factor, or weight, corresponding to all entry points)
- `weights`: a weights array (an array to hold the averaged weights of entry points in all clusters)

#### *2.7.3.2. Shader description*

The following techniques are used. All of them contains a single pass, with a unique vertex and pixel shader. They are defined in their respective effect files.

*Depth*

Renders the scene with the given transformation (`modelToProjMatrix`). Color is irrelevant. This technique is invoked for every entity, with appropriate model and camera settings to render a depth map. It is more robust to render back faces when rendering the depth map, to avoid z-fighting when determining shadows.

*BushToAtlas*

Renders the contribution of a single virtual light source (Radion, described by `lightPos`, `lightDir`, `lightPower`) into a texture atlas. This technique is invoked with a tile of a PRM set as the render target and viewport. Blending is used to add up the contributions of radions that belong to the same cluster in a tile of the entity's PRM. Comparison with a depth map is used to determine shadows. The point-to-point form factor between the virtual light source and the shaded fragment is modified if the shaded fragment is near to the plane of the radion. This avoids typical virtual light source artifacts like spikes and dark corner edges. The basic assumption is that the virtual light source is on a plane. Form factors near this plane are (distance squared less than `cutNearness2`), are considered inaccurate because of insufficient sampling. `cutNearness2` should be set according to the sampling probability. Any shaded fragment that is considered too close will be projected onto the cutting plane perpendicular to the surface normal at the virtual light source. This ensures that a planar surface perpendicular to the surface of the virtual light source will receive uniform illumination near the corner.

*ComputeWeights*

Takes a texture containing an array of entry point positions and surface normals as input (`radionSamper`), and computes the weights corresponding to every individual entry point into a target texture (`weightsTexture`). Weights are actually the incoming illuminations of the entry points: they involve finding the visibility and the form factor. Visibility is determined using a depth map (`depthMapSampler`), which should be previously rendered using technique *Depth*. The form factor is computed for a spotlight (`lightPos`, `lightDir`, `lightPower`). When the direct illumination of the scene will be computed in technique *Walk*, the same spotlight characteristics should be used.

*Walk*

The final rendering technique, named as such because it is used for walkthroughs of the scene when precomputing is already done. It computes indirect and direct lighting for a spotlight (`lightPos`, `lightDir`, `lightPower`). For direct lighting shadows, a depth map is used (`depthMapSampler`, the same depth map that was used to find entry point visibilities for weight computation.) For indirect lighting, tiled PRM panes from the texture assigned to `filteredAtlasSampler` are combined, according to the weights specified in uniform parameter array `weightsa`. The weight should be the average weight of those entry points, whose contribution has been rendered to the PRM pane. Individual weights are computed by technique *ComputeWeights*, averages must be computed on the CPU.

### *2.7.3.3. Data flow*

2.7.3.3.1. Preprocessing

For the indirect lighting of the scene, we need the PRM textures and the array of entry points along with clustering information. These computations are carried out in method `PathMapEffect::precompute()`. It is invoked from the constructor, after setting up the scene in `renderMeshes` and `entities`. Precomputation has the following steps:

- Generate entry points on the surfaces using method `sampleSurfaceRadion`. Entry points will be stored as `Radion` instances in `bushStarters`.
- Entry points are clustered according to their position and orientation.
    - Initial clustering is performed by method `clusterRadions`, building a balanced kd-tree, forming uniformly sized clusters.
    - More refined clustering is done in `clusterRadionsKMeans`, where the initial clusters are iteratively reclustered. Empty clusters are illegal, if any cluster would be empty, half of the contents of another cluster is moved to it.
- For every entity, the cluster relevant for its illumination are found, and stored in `Entity::nearClusterIndices`.
- For all entry points in a cluster, a bush of virtual light sources is generated using ray shooting (`shootRadionBush()` invokes `sampleShootingDiffuseDirection()` and

`KDTree::traverse()`). Then, for all entities in vector `entities`, their contribution is added to the PRM texture `Entity::prmTexture` in `Entity::renderPRM()`. In that method:

- For the actual cluster, the corresponding tile in the PRM texture is found using the `nearClusterIndices` array. If it is not listed, nothing is rendered.
- For all virtual light sources in the cluster (passed in parameter `bushRadions`):
  - A depth map is rendered using technique *Depth*, into depth buffer `depthMap`.
  - `prmBlendingDepthStencilBuffer` is set as the stencil buffer. For one light source, one pixel is added to only once.
  - Using blending, the contribution of the virtual light source, respecting shadows as per the depth map, is rendered to the appropriate PRM tile with technique *BushToAtlas*. For conservative atlas rendering, `RenderMesh::edgeVertexBuffer` has to be drawn besides `RenderMesh::mesh`.

After these steps, the PRM textures for all entities are ready. All precomputed data might be saved to disk (`savePathMaps()`) or used instantly. With the appropriate weighting the RPM textures can be used to render indirect illumination. Weighting factors are computed at final rendering time, based on current lighting.

2.7.3.3.2. Final rendering

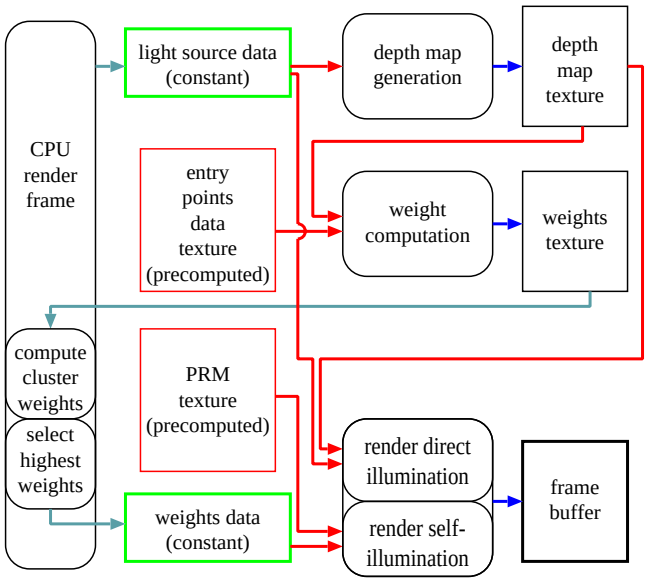



*Figure 41. Data flow between shaders during the final rendering (walkthrough) phase. Rounded rectangles correspond to shader techniques Depth, ComputeWeights, and Walk.*

During the final walkthrough, we have to find weights based on the light position, and entry point visibility, and then use them to render objects combining texture atlases in PRMs. This functionality is provided in method `PathMapEffect::renderWithPRM()`. After computing the weights with technique *ComputeWeights*, the result texture `weightsTexture` has to be read in system memory back using `sysMemWeightsTexture`. The weights are averaged in every cluster, and cluster weights are passed to final rendering pass *Walk* in shader parameter `weightsa`. Both for weights computation and direct illumination computation in the final rendering, a depth map (`depthTexture`) has to prepared for the light source in advance.

### 2.7.4. Using the standalone Path Map tool

#### *2.7.4.1. Compilation*

The PathMap program can be compiled in two modes, set by the compiler directive `#define GENERATE_PATH_MAPS`. If it is defined, the tool will compute the PRM textures corresponding to the scene entities, and the entryPoints, and save them in the *prm* subfolder. Then it will launch the realtime visualization of the scene with the computed PRM. If `GENERATE_PATH_MAPS` is not defined, however, the PRM data will not be computed, but restored from the *prm* folder. There are a few constants that can be set in file *PathMapEffect.h* to customize PRM quality. `NRADIONS` must be a multiple of 4096, and defines the number of entry points. Increasing this will increase image quality at the cost of more preprocessing, but without effecting final rendering frame rates. `NCLUSTERSPERENTITY` defines how many of the most significant clusters will be computed and used for a single entity. It must be a multiple of 32, though anything more than 32 is practically not feasible. `NCLUSTERS` defines the number of clusters for the complete scene. This number should be higher than 32. The entry points will be divided into this number of clusters. However, only `NCLUSTERSPERENTITY` clusters can be considered when shading an entity. Therefore, if there are too many clusters within the scene, there may be some significant clusters omitted. For a large level, `NCLUSTERS` should be set so that `NCLUSTERSPERENTITY` clusters approximately cover a visually confined segment, like a room. `DEPTHMAPRES` sets depth map resolution.

#### *2.7.4.2. Runtime controls*

There is a number of on-screen controls available to navigate the scene and adjust lighting. Use the mouse to look around, the A, S, D, W keys to move. If the “Move light” box is checked, the same controls move the light source. If “Look from light” is checked, the scene is rendered as seen from the light source instead of the camera. It is convenient to check both boxes when positioning the light source. Selecting the “Entry points” checkbox will add small squares to the scene to indicate entry points. Colors indicate weights. The clustering also becomes visible. The “Turbo” checkbox toggles camera movement speed. Pressing the “Tab” button will display a PRM texture, or revert to scene rendering. The technique *ShowTex* may be modified to display other internal textures for illustration. The slider “Tone scale” allows scaling the resulting image, which can also be viewed as changing the light source intensity.


*Figure 42. Screen controls in the Path Map standalone tool.*

## 2.8. IMAGE BASED LIGHTING

### 2.8.1. Role of the module

The image based lighting application implements the environment map lighting algorithm with self shadows as it was described in the technical documentation. Its main purpose is to render outdoor scenes where lighting from the environment is widely distributed. The environment map is decomposed into directional samples using Delaunay triangulation and Lloyd's relaxation. Then the contribution of these directional light samples is rendered to the screen using blending. Depth maps are used for accurate shadows, including self-shadowing.

*Figure 43. Scenes rendered with image based lighting, using the Uffizi environment. Multiple objects cast shadows on each other, and environment lighting enters a room through windows.*

### 2.8.2. Program structure

Rendering the scene with the lighting due to the directional light samples is implemented in class `IBLEffect`. An `IBLEffect` instance is created whenever a new D3D device context is initialized, and destroyed if the context is lost. Some callbacks, most importantly the `OnFrameRender` callback, invoke `IBLEffect` methods. `IBLEffect` also manages virtual world objects. When it is constructed, it loads meshes and textures from the "media" subfolder and the environment map form the "hdr" subfolder.

The most important resources encapsulated by the `IBLEffect` class are the following:

- `LPDIRECT3DCUBETEXTURE9 environmentMap` is the environment cube map texture, loaded from a .dds file using `D3DXCreateCubeTextureFromFile`
- `HdriSampler* hdriSampler` is used to generate directional light samples when `IBLEffect` is instantiated.
- `LPDIRECT3DTEXTURE9 lightSampleDepthTextures[8]` are 8 pre-allocated depth textures. 8 samples will be processed in one pass, therefore 8 maps are needed. They are created in `IBLEffect::IBLEffect` by:

```
device->CreateTexture(depthMapResolution, depthMapResolution, 1,
D3DUSAGE_RENDERTARGET, D3DFMT_A16B16G16R16F, D3DPOOL_DEFAULT,
&lightSampleDepthTextures[iDepthMap], NULL);
```

We need to set the usage to `D3DUSAGE_RENDERTARGET` because we will render the depth maps into the textures using a shader. `D3DFMT_A16B16G16R16F` is used because it allows linear filtering, and we can store the surface normal along with the depth. This will allow us to get artifact-free results from lower resolution depth maps.

### 2.8.3. Shader description

The functionality that comprises the image based lighting algorithm is in method `IBLEffect::render()`. Using the `LPDIRECT3DDEVICE9 device` and `LPD3DXEFFECT effect` references stored at construction, this method assigns resources and values to effect file variables, which are associated with uniform shader parameter registers and texturing stages by the effect framework. Non-uniform parameters like vertex position, normal and texture coordinates are always encoded in vertex streams. Vertex streams for rendering primarily come from the meshes loaded at construction time. These meshes are therefore converted to the desired vertex format in the `IBLEffect::loadMesh(LPCWSTR)` method by invoking `IBLEffect::RenderMesh::setVertexFormat(DWORD, LPDIRECT3DDEVICE9)`, which replaces the mesh with a clone having appropriate vertex format, created using `CloneMeshFVF`. When we render a full-screen quad, the Delaunay mesh or the light samples, then vertex data is assembled in the code, and sent to the hardware using `DrawPrimitive` calls. These tasks are implemented in:

- `IBLEffect::renderFullScreen()`
- `VoronoiGenerator::d3dDrawDelaunay(LPDIRECT3DDEVICE9 device)`
- `hdriSampler::d3dDrawPowers(LPDIRECT3DDEVICE9 device).`

Rendering a scene with possibly several hundred lights and on-the-fly generated depths maps for all of them requires a large number off passes, with different render targets, device settings, shader programs and parameters. The skeleton of the algorithm is the following:

1. Lay down depth. We render the scene geometry, with the camera transformation, with Z buffering enabled, but color writes disabled. Further passes rendering to the frame buffer will be able to use early Z tests to discard pixels that will not be visible. (Technique `renderBlack` in file *iblFinal.fx*.)
2. Take a batch of directional light samples and compute their depth maps. To do this, we have to loop through these samples, construct their world-to-texture transformations, and render the scene to the depth map textures set as render targets. (Technique `renderDepth` in file *iblDepth.fx*.) We also need to set a depth-stencil buffer which has the same dimensions as the depth maps. It is also important that the contents of the frame buffers depth-stencil must not be discarded. Hence is the callback in *ImageBasedLighting.cpp* necessary:

```
bool CALLBACK ModifyDeviceSettings(
DXUTDeviceSettings* pDeviceSettings, const D3DCAPS9* pCaps, void* pUserContext ) {
    pDeviceSettings->pp.Flags &=
    ~D3DPRESENTFLAG_DISCARD_DEPTHSTENCIL;
    return true;
}
```

3. Render the contribution of the light samples in the current batch to the frame buffer. (Techniques `renderFinal`, `renderFinalPerPixelTransform`,

`renderFinalShiny` in file `iblFinal.fx`.) We want to add contributions of batches in the frame buffer, and we use blending to accomplish that. It is now important that we have laid down depth in step 1., as only visibly surfaces will be rendered.

4. If there are light samples not yet processed, go back to step 2.

Render the environment cube. (Technique `renderBackground` in file *ibl.fx*.) This is done by rendering a full-screen quad at projected Z depth of (1 - ε).

The following subsections elaborate on the most complicated steps and explain the shaders used.

#### *2.8.3.1. Rendering the depth maps*

A depth map is a texture that contains depth values as seen from a light source. Transforming any world point into the texture space of the depth map, a comparison of the Z value and the value stored in the map answers the question, whether there is a surface occluding the light source. For a point-like light sources the transformation is the generic perspective projection, and for directional lights it is a simple orthogonal projection.

Depth mapping is directly supported in graphics hardware. A texture may be created with the usage `D3DUSAGE_DEPTHSTENCIL`, allowing it to be set as a depth-stencil surface. Reading this map with a world point transformed into texture space only returns 0 or 1 depending on the result of the comparison. assigning linear filtering to the texture only interpolates these values, not the depth stored in the map.

In our application, we need a lot of rapidly generated depth maps, but any of them will only contribute to a fraction of the total illumination, so accurate shadow silhouettes are not important. On the other hand, we want to avoid periodic artifacts due to low resolution. Depth maps have a finite resolution, both in terms of dimensions and bit depth. If the comparison of performed on nearest-neighbor-sampled depth values, then world points on slant surfaces will appear to be behind the depth map Z values. This is generally handled by subtracting a constant bias from Z values of points checked for illumination. However, if the depth map is small, this bias needs to be large to avoid artifacts (they appear as jigsaw-edged stripes on completely smooth surfaces). A large bias means that light may leak through thin objects, with very disturbing results. Much of these concerns are eliminated if depth values are interpolated. In case of an orthogonal projection, depth values within a triangle will change linearly, providing perfect results. However, for geometries that are concave as seen from the light, linear interpolation can also cause undesirable shadows. We can help this if we take the orientation of the surfaces into account. If we save the surface normals along with the depth values to the depth map, we can check whether it is oriented similarly to shaded the surface point. If it is, then it is probably on the same surface, so a large bias to eliminate artifacts is justified. If it is not, then it is certainly not on the same surface, and the bias should be small. Linear interpolation of depth values and storing normals in depth maps is not supported by hardware depth mapping, so we use conventional floating point render target textures to accomplish this goal.

Theoretically, we have to use the same transformation matrix when rendering to the depth map, and when reading the depth value for a world point to be shaded. However, we have use the screen coordinates when rendering, and texture coordinates when addressing the texture. This means we have to append the screen-to-texture matrix to the depth map's world-to-screen matrix after rendering the depth maps.

Note that both in the program code and in the shaders transformation matrices are always named according to which transformations of the graphics pipeline they include. The depth map's world-to-depth map transformations are therefore referred to as `DepthViewProj` in shaders and `depthViewProjTransform` in C++ code. Thus, this transformation includes the view (world-to-camera) and projection (camera-to-screen, orthographic) matrices. The matrix which transforms the world points to texture space so that the coordinates can be used to address the depth map is called `DepthViewProjTex` and `depthViewProjTexTransforms`, respectively.

The matrix has to be constructed so that all the geometry is visible on the depth map. When loading meshes, we compute their bounding spheres using `D3DXComputeBoundingSphere`. Then we compute the bounding enclosing sphere of all model-transformed bounding spheres in method `IBLEffect::buildDepthViewProjTransform`. Then the camera is set up so that it looks towards the center of the sphere, from the surface of the sphere, from the direction of the light sample. The projection matrix is then a simple orthographic projection of the cube with twice-the-radius length edges to unit screen space.

```
    D3DXVECTOR3 from = center + lightDirection * radius;
    D3DXVECTOR3 at = center;
    D3DXVECTOR3 up;
    //build a view matrix to look at the center from the given direction, from the sphere
surface
    D3DXMatrixLookAtLH(&depthViewProjTransform, &from, &at, &up);
    D3DXMATRIX projMatrix;
    //transform the sphere into [-1 -- 1, -1 -- 1, 0 -- 1]
    D3DXMatrixOrthoLH( &projMatrix, radius * 2.0f, radius * 2.0f, 0.001f, 2.0f * radius);
    D3DXMatrixMultiply( &depthViewProjTransform, &depthViewProjTransform, &projMatrix);
```

The shader rendering the depth-and-normal textures is then very simple.

```
struct VS_INPUT_DEPTH {
    float4  Position    : POSITION;
    float3  Normal      : NORMAL;
};

struct VS_OUTPUT_DEPTH {
    float4  Position    : POSITION;
    float3  Normal      : NORMAL;
    float1  Depth       : TEXCOORD0;
};

VS_OUTPUT_DEPTH
    DepthVS(VS_INPUT_DEPTH IN)
{
    VS_OUTPUT_DEPTH OUT = (VS_OUTPUT_DEPTH)0;
    OUT.Position = mul(IN.Position, World);
    OUT.Position = mul(OUT.Position, DepthViewProj);
    OUT.Normal = mul(float4(IN.Normal.xyz, 0.0), WorldIT);
    OUT.Depth = OUT.Position.z;
    return OUT;
}

float4
    DepthPS(VS_OUTPUT_DEPTH IN) : COLOR0
{
    return float4(normalize(IN.Normal), IN.Depth);
}
```

### *2.8.3.2. Rendering illumination*

In the effect files, the techniques to render the scene geometry with the contribution of 8 light samples using the depth maps are called `renderFinal`, with postfixes for variants. They are the final step of the algorithm in the sense that they render to the screen using the previously computed depth maps. There are four variants:

- `renderFinal` is the fastest method. All 8 depth map coordinates are computed in the vertex shader, and interpolated by the hardware. This is efficient as there are usually fewer vertices then pixels, and the method is pixel-shader intensive. However, all possible registers are occupied, so if we wanted to add further pixel shader input data, e.g. non-constant material properties, we would have to decrease the number of light samples processed in one pass.

- `renderFinalPerPixelTransform` remedies this situation by passing the world space coordinates only, and depth map texture addresses are computed in the pixel shader. This should be slightly slower, but leaving a lot of registers free. `renderFinalShiny` is the same as `renderFinalPerPixelTransform`, but it adds a Phong component to the BRDF model. The shininess and albedo of this component can be set using the application GUI.

- `renderFinalShinyRadius` is a complicated method to smear specular highlights that might be disturbing when there are few samples and very shiny materials. With one hundred or more samples it is less of an issue, and the technique is only there for the sake of completeness. It is not available for the user interface.

As the techniques are very similar, let us examine the most complete `renderFinalShiny`. The vertex shader receives the raw mesh vertex: the vertex position and normal in model space, and the associated texture coordinate. It has to output the position transformed to screen coordinates, the position, normal and ideal reflection vectors in world space, and the unaltered texture coordinates. Uniform variables are camera position `EyePosition`, the model-to-world transform `World`, its inverse transposed `WorldIT`, and the current camera's world-to-screen `ViewProj` matrix. The vertex shader simply uses the matrices to transform input data to the appropriate coordinate systems. `OUT.Reflect` is the reflected eye direction vector. Note we have to use `OUT.Normal`, the world-space surface normal vector for the reflection computation.

```
float3   EyePosition;
float4x4 World;
float4x4 WorldIT;
float4x4 ViewProj;

struct VS_INPUT_FINAL_SHINY {
    float4 Position         : POSITION;
    float3 Normal           : NORMAL;
    float2 TexCoord         : TEXCOORD0;
};

struct VS_OUTPUT_FINAL_SHINY {
    float4 Position         : POSITION;
    float3 Normal           : NORMAL;
    float2 TexCoord         : TEXCOORD0;
    float4 WorldPosition    : TEXCOORD1;
    float3 Reflect          : TEXCOORD2;
};
```

```
VS_OUTPUT_FINAL_SHINY
    FinalShinyVS(VS_INPUT_FINAL_SHINY IN)
{
    VS_OUTPUT_FINAL_SHINY OUT = (VS_OUTPUT_FINAL_SHINY)0;

    OUT.WorldPosition   = mul( IN.Position,                 World);
    OUT.Position        = mul( OUT.WorldPosition,           ViewProj);
    OUT.Normal          = mul( float4(IN.Normal.xyz, 0.0),  WorldIT);
    OUT.TexCoord        = IN.TexCoord;

    float3 eyeDir = normalize(EyePosition.xyz - OUT.Position.xyz);
    OUT.Reflect = reflect(eyeDir, OUT.Normal);
    return OUT;
}
```

The pixel shader receives the interpolated vertex shader output. It has to have access to all the light sample information, including the depth map textures, the `DepthViewProjTex` transformation matrices, light sample directions and powers. It also needs to access surface material properties encoded in the brdf texture `brdfMap` and variables `ShinyKS` and `ShinyExponent`.

```
texture brdfMap;
sampler2D BrdfMapSampler =  sampler_state{
        texture = < brdfMap >;
        MinFilter = LINEAR;
        MagFilter = LINEAR;
        MipFilter = None;
        AddressU  = Wrap;
        AddressV  = Wrap;
        };

float ShinyExponent;
float ShinyKs;

float4x4    DepthViewProjTex[8];
float3      LightDir[8];
float3      LightPower[8];

//There is no such thing as a texture array.
//We use a macro to define 8 textures and samplers.

#define DECLARE_DEPTH_MAP_WITH_SAMPLER(index)   \
    texture depthMap##index ;                   \
    sampler2D DepthMapSampler##index =  sampler_state{  \
            texture = < depthMap##index >;      \
            MinFilter = LINEAR;                 \
            MagFilter = LINEAR;                 \
            MipFilter = None;                   \
            AddressU  = Clamp;                  \
            AddressV  = Clamp;                  \
            };

DECLARE_DEPTH_MAP_WITH_SAMPLER(0)
DECLARE_DEPTH_MAP_WITH_SAMPLER(1)
DECLARE_DEPTH_MAP_WITH_SAMPLER(2)
DECLARE_DEPTH_MAP_WITH_SAMPLER(3)
DECLARE_DEPTH_MAP_WITH_SAMPLER(4)
DECLARE_DEPTH_MAP_WITH_SAMPLER(5)
DECLARE_DEPTH_MAP_WITH_SAMPLER(6)
DECLARE_DEPTH_MAP_WITH_SAMPLER(7)
```

Unfortunately, we cannot use a loop to process the 8 samples, because we cannot access our 8 textures using an index. Instead of typing the same for all eight of them, we use a macro. In the pixel shader,

we first normalize incoming `Normal` and `Reflection` vectors, then define the variables used for lighting.

We use variable `rcolor` to sum the contributions of the eight samples. `DepthTexCoordPos` will be the position transformed to the depth map texture space, which can be used to address the texture. `cosTheta` is NL, `cosPsi` is RL in the usual notation N is the normal, L is the light direction, R is the ideal reflected direction of the eye direction.) In the macro, we first compute NL. If it is not facing the light, then we compute depth map coordinates and read the depth map for an occluding depth and surface normal value. We compare the depth values, allowing a larger bias for similarly aligned (probably identical) surface elements. If the surface is visible form the light, we compute $RL^n$ for the Phong component. Finally, we multiply by the light sample's power. Before we return the sum accumulated in `rcolor`, we modulate the color with the value read from the material texture `brdfMap`.

```
float4
    FinalShinyPS(VS_OUTPUT_FINAL_SHINY IN) : COLOR
{
    IN.Normal = normalize(IN.Normal);
    IN.Reflect = normalize(IN.Reflect);
    float3 rcolor = 0;
    float3 DepthTexCoordPos;
    float cosTheta;
    float cosPsi;
    float4 mapNormalAndDepth;

#define COMPUTE_LIGHT_SAMPLE_CONTRIB_SHINY( iLight )    \
        cosTheta = dot(LightDir[ iLight ], IN.Normal);  \
        if(cosTheta > 0.0)  \
        {   \
            DepthTexCoordPos = mul(IN.WorldPosition, DepthViewProjTex[ iLight ]);   \
            mapNormalAndDepth = tex2D( DepthMapSampler##iLight , DepthTexCoordPos.xy);  \
            if( DepthTexCoordPos.z - 0.01 * dot(normalize(mapNormalAndDepth.xyz), IN.Normal) <
mapNormalAndDepth.w )    \
            {   \
                float as = 0.0; \
                cosPsi = -dot(IN.Reflect, LightDir[ iLight ]);  \
                if(cosPsi > 0.0)    \
                {   \
                        as = pow(cosPsi, ShinyExponent);    \
                }   \
                rcolor +=  (cosTheta + as * ShinyKs) * LightPower[ iLight ]; \
            }   \
        }

    COMPUTE_LIGHT_SAMPLE_CONTRIB_SHINY(0)
    COMPUTE_LIGHT_SAMPLE_CONTRIB_SHINY(1)
    COMPUTE_LIGHT_SAMPLE_CONTRIB_SHINY(2)
    COMPUTE_LIGHT_SAMPLE_CONTRIB_SHINY(3)
    COMPUTE_LIGHT_SAMPLE_CONTRIB_SHINY(4)
    COMPUTE_LIGHT_SAMPLE_CONTRIB_SHINY(5)
    COMPUTE_LIGHT_SAMPLE_CONTRIB_SHINY(6)
    COMPUTE_LIGHT_SAMPLE_CONTRIB_SHINY(7)

    return float4(ToneScale * rcolor * tex2D(BrdfMapSampler, IN.TexCoord).rgb, 1.0);
}
```

### 2.8.4. Using the standalone Image Based Lighting module


*Figure 44. Screen controls in the Image Based Lighting standalone module.*

The A, W, S, D buttons and the mouse can be used to navigate the scene by moving the camera. The following options are available:

- The *Regenerate* button will re-build the scene, and re-compute the directional samples. It is required if the scene is changed by adding a second model, or any of the resources (depth map, directional samples) must to be updated.
- *Second model* shows a second entity based on the loaded mesh. It can be observed how objects cast shadows on each other. You need to regenerate the scene for this change to take effect.
- *Num of samples* is a slider to set how many directional samples should be used. You need to regenerate the scene for this change to take effect.
- *Depth map size* sets the resolution of the depth map. You need to regenerate the scene for this change to take effect.
- *Show Delaunay & lights* toggles whether the Delaunay decomposition of the environment and the directional samples are displayed.
- Tone scale set scene light intensity.
- *Phong shininess* and *Phong albedo* let you set the material properties of the models.
- *Model rotation* sets the angular velocity of the primary model's rotation.
- *Tab* can be used to switch between rendering modes.

## 2.9. HIERARCHICAL RAY ENGINE

### 2.9.1. The role of the module

CPU-based acceleration schemes are spatial object hierarchies. For a ray, they try to exclude as many objects as possible from intersection testing. This cannot be done in the ray engine architecture, as it follows the per primitive processing scheme instead of the per ray philosophy. Therefore, we also have to apply an acceleration hierarchy the other way round, not on the objects, but on the rays.

In typical applications, realtime ray casting augments scan conversion image synthesis where recursive ray tracing from the eye point or from a light sample point is necessary. In both scenarios, the primary ray impact points are determined by rendering the scene from either the eye or the light. As nearby rays hit similar surfaces, it can be assumed that reflected or refracted rays may also travel in similar directions, albeit with more and more deviation on multiple iterations. If we are able to compute enclosing objects for groups of nearby rays, it may be possible to exclude all rays within a group based on a single test against the primitive being processed. This approach allows for an acceleration of the ray engine algorithm, making its use feasible in realtime environments.

The Illumination Workpackage includes modules addressing ray tracing effect generation, most notably the Ray Trace Effects module itself. The Indirect Illumination Gathering module extends on its capabilities, and the OGRE Illumination Module hosts it in the game engine. However, these modules use approximate methods, suitable for a game environment, but prone to artifacts for difficult geometries. The hierarchical ray engine performs accurate ray tracing, and it could work with ray tracing primitives other than triangles.

In the original ray engine, we store all the rays to be traced in a texture. Then, all primitives are rendered as full-texture quads, and pixel shaders compute ray-primitive intersections. In the hierarchical version, we find a way not to render it on the entire screen as a quad, but invoke the pixel shaders only where an intersection is possible. The solution is to split the render target into tiles, render a set of tiles (DirectX point primitives) instead of a full screen quad, but make a decision for every tile beforehand whether it should be rendered at all. This way we utilize vertex shaders, while pixel shader load is significantly decreased.

### 2.9.2. Program structure

The standalone ray engine module can be found in folder *HierRayEngine*. It is a DirectX application that displays a scene of refractive objects in an environment defined by a cube map. Shaders can be found in the effect file *rayEngine.fx*.

#### *2.9.2.1. Program resources*

The class that contains the main functionality is the `HREEffect` class. It encapsulates the resources and operations necessary to render scenes with recursive ray tracing of refractive objects. This includes a scene entity and mesh structure similar to the one described for the Light Path Maps module. For every mesh, a ray engine representation (a buffer of vertices encoding primitives) is maintained. Furthermore, textures storing ray information and various intersection test output buffers are required.

Therefore, `HREEffect` contains the following types of resources:

Scene description:

- `meshes`: a vector of meshes
  - `mesh`: a D3D mesh
  - `materials`: an array of D3D materials of submeshes
  - `textures`: an array of references to the material textures of submeshes
  - `primitiveVertexBuffer`: ray engine representation of the mesh (list of D3D point primitives with a triangle encoded in each) The vertex data describing a triangle primitive is composed as:
    - `position`: enclosing sphere center and radius
    - `normal`: the normal and offset of the triangle's plane
    - `texture`: 0-2 pre-processed triangle vertex position data for fast intersection computation
    - `further texture registers`: normals at the triangle's vertices
    - `further texture registers`: texture coordinates at the triangle's vertices
- `entities`: a vector of entities
  - `mesh`: a reference to a mesh
  - `modelWorldTransform, inverseTransposedModelWorldTransform`: modeling transformations

Ray buffers:

- `rayOriginTableTexture, rayDirTableTexture`: textures containing a two-dimensional array of rays to be traced. Rays in nearby pixels will be considered likely to be coherent. Both textures have four channels, three of which contain the 3D coordinates and direction vector for the ray that belongs to the texel. The fourth channel in the directions texture encodes a color modulation associated with the ray.
- `rayOriginTableTexture2, rayDirTableTexture2`: render target textures to hold refracted rays, as intersection results. These results are then copied back to the ray origin and direction textures for the next iteration.
- `conePeakTexture, coneDirTexture`: render target textures containing enclosig cone data for tiles of the ray textures. Every texel in the cone textures corresponds to a tile. `conePeakTexture` contains the 3D apex coordinates, `coneDirTexture` contains the 3D direction vector and the cosine of the opening angle. These textures must always be computed

from the textures describing rays to be traces with the *ComputeCones* technique. Then they are used by the vertex shader of the intersection shader *RayCast*.

- `conePeakTextureMem, coneDirTextureMem`: System-memory copies of the cone data textures. As these textures are small, they can be read into system memory fast, and then their contents passed to the *RayCast* vertex shader as uniform parameters. This way there is no need to perform texture fetches in the vertex shader.
- `depthSurface`: The depth and stencil buffer surface for rendering refracted rays, for both *RenderPrimaryRayArray* and *RayCast* techniques. The depth test ensures that always the nearest intersection is stored. The stencil buffer is used to exclude those pixels where there is no ray to be traced, because the ray path corresponding to the texel has already terminated.
- `environmentCubeTexture`: Environment texture loaded from a file (media/uffizi_lowrange.dds). It is used in shader technique Background to fetch colors corresponding to eye rays, or refracted rays, where a refractive object is visible.

#### *2.9.2.2. Rendering a scene*

To render a scene of refractive objects, the following passes are required:

- Clear the stencil buffer in `depthSurface` indicating which ray paths have already terminated.
- The “current stencil bit” is the LSB of the stencil buffer.
- Render the refractive objects to the ray textures, with technique *RenderPrimaryRayArray*. The output for every surface fragment will be the fragment position and the direction of the refracted eye ray, in world coordinates. Z-buffering is required. The current stencil bit is set for a pixel if any object was rendered.
- For a number of iterations, repeat:
    - The “actual stencil bit” is shifted towards the MSB.
    - Using shader *ComputeCones*, render the cone data to the cone textures. Ray textures containing the latest rays to be traced are used as input. Then the render targets are read back to system memory.
    - Render the intersections to the duplicates of the ray textures.
        - The ray textures are inputs as textures. The information from cone textures is provided as uniform parameter input.
        - Depth must be cleared.
        - For every tile of the render target:
            - The vertex buffers containing the refracted objects encoded into D3D point primitives are rendered using technique *RayCast*. Cone data for the tile is provided as a uniform parameter.

- For every vertex (encoding a ray-casting triangle), the vertex shader checks if there is an intersection between the cone and the enclosing sphere of the triangle. If there is no intersection, it outputs a position which assures the point is clipped away and no pixel shader is invoked. Else, triangle data is passed down to the pixel shader.
- If the previous stencil bit is not set, the stencil test will discard the pixel shader result. (Or, with early stencil test, it preempts it.)
- For every pixel, the ray data is read from the ray textures. The ray-triangle intersection is computed. If there is an intersection, the refracted ray is computed. The output is the refracted ray data. The color of the object is blended with the current color modulation of the ray path (fourth channel of the direction texture.) Depth output is the intersection distance. The current stencil bit is set if any intersection was found.
- Late depth testing discards the result if a nearer intersection has already been found.

  - In texels where the actual stencil bit is set (there was a valid result), the results are copied back to the "rays to be traced" textures using technique *CopyBack*.

- Now the ray textures contain the refracted rays after all the iterations. Where no object was visible, we have a zero value, so we have to use the original eye ray. Where the ray paths has terminated, we have the final ray exiting the scene towards the environment. Rendering a full screen quad to the screen, the directions of these exiting rays are used to address the cube map environment texture in `environmentCubeTexture`. Incoming radiances are modulated according to the color value stored in the directions texture. This produces the final image.

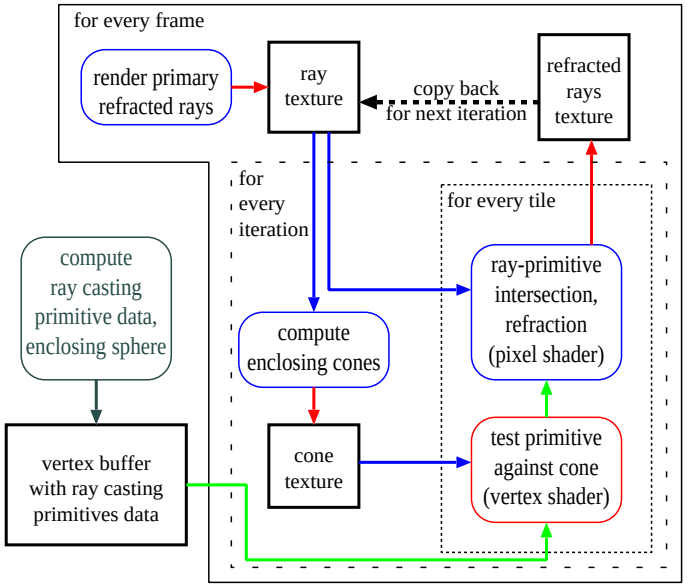



*Figure 45. Block diagram of the hierarchical ray engine.*

### *2.9.2.3. Shader description*

*RenderPrimaryRayArray*

Simply renders objects, transformed according to the current modeling, view and perspective transformation. Outputs refracted ray origin and direction data to its two render target textures.

*ComputeCones*

Enclosing infinite cones of rays within a tile are described by an origin (first render target: `conePeakTexture`), a direction and an opening angle (second render target: `coneDirTexture`). The infinite enclosing cones are constructed using this technique. The algorithm in the pixel shader proceeds as follows:

1) Start with the zero angle enclosing cone of the first ray (fetched from the ray textures).

2) For each ray (fetched from the ray textures)

   a) Check if the direction of the ray lies within the solid angle covered by the cone, as seen from its apex. If it does not, extend the cone to include both the original solid angle and the new direction.

   b) Check if the origin of the ray is within the area enclosed by the cone. If it is not, translate the cone so that it includes both the original cone and the origin of the ray. The new cone should touch both the origin of the ray and the original cone, along one of its generator lines.

*RayCast*

This technique computes intersections for an array of rays, and return refracted rays in its two (origin and direction) render targets. Primitive data is taken from the vertex buffer, through the incoming non-uniform registers. The technique is invoked for a tile, by rendering a buffer of point primitives covering pixels of the tile. The vertex buffer does not contain the position of the tile; it is passed as the uniform parameter `tilePos`. The `POSITON` vertex buffer slot is used to pass the enclosing sphere data to the vertex shader. The vertex shader transforms the cone to modeling space, and performs the sphere-cone intersection test. If the test fails, a position not in the unit cube is output in the `POSITON` output register. Else, `tilePos` is output in `POSITION`. In the pixel shader, the ray origin and direction are fetched from the textures. They are transformed to modeling space. Ray-triangle intersection is performed by computing the barycentric coordinates of the ray-plane hit point. There is no hit, if any of the coordinates is negative. Normals are interpolated using the barycentric weights. Then, knowing the normal and the hit point, the refracted ray is computed, and output as an origin and direction into the first two render targets. Object color given in uniform parameter `colOrder` is appended to the color modulation stored in the fourth channel of the direction texture.

*CopyBack*

This shader does not do anything but copy pixels of two textures, when a full screen quad is rendered. Combined with the stencil buffer, it can be used to copy the new, refracted rays back to the ray textures only where an intersection has been found. All other pixels -- containing already terminated, exiting rays -- are left intact.

*Background*

This shader should be invoked by rendering a full-screen quad on the screen. It fetches the directions of exiting rays, plus the corresponding color modulations from the ray directions texture, and uses them to look up an environment map. If zero is read, the camera-to-pixel ray is used. This shader renders a final image, displaying visible refractive objects.

### 2.9.3. Using the standalone Hierarchical Ray Engine module

The A, W, S, D buttons and the mouse can be used to navigate the scene by moving the camera.

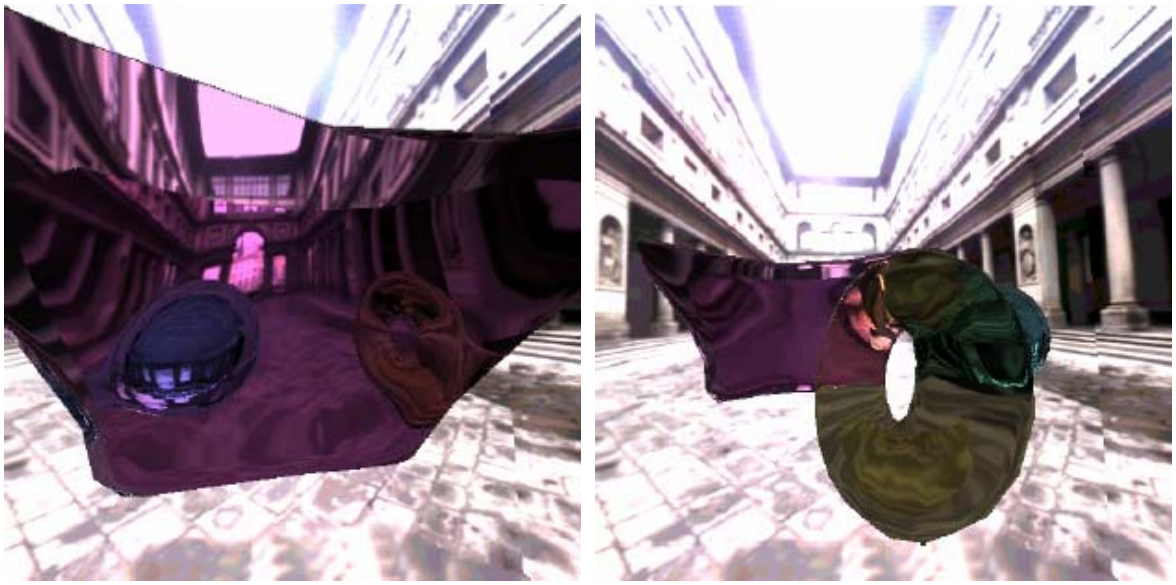


*Figure 46. Scene of refractive objects rendered in the hierarchical ray engine standalone module.*

## 2.10. OBSCURANCE

The standalone obscurance module can be found in the *Obscurances* directory of the repository.

### 2.10.1. Role of the module

The obscurance method is a powerful technique that simulates the effect of diffuse interreflections. It approximates radiosity, but requires much lower computational cost. Its main advantage lies in the fact that this technique considers only neighboring interactions instead of attempting to solve all the global ones. Another advantage of this technique is that it is decoupled from direct illumination computation. Obscurances allow realistic and fast illumination of scenes.

This technique can deal with any number of moving light sources with no added cost since the indirect illumination and direct illumination are effectively decoupled. The obscurance technique also allows the addition of color bleeding to your lighting. The algorithm to compute obscurances uses the depth peeling technique as explained in the following sections.

### 2.10.2. Algorithm explanation

Given a geometry and a desired obscurance map resolution, the application returns the corresponding obscurance map. The obscurance map simulates the indirect illumination that receives each part (patch) of a scene. This indirect illumination can be combined with the direct illumination and an ambient light to increase the realism of the scene. This obscurance map assigns to each patch in the scene a degree of darkness (obscurance) that is given by the geometry that surrounds the patch.

In order to create the obscurance map, a technique called "*Global lines using depth peeling*" is used. The process starts calculating the bounding sphere of the scene (Figure 47, left). After this step we start taking random points at the surface of this sphere to create the global directions (Figure 47, right).

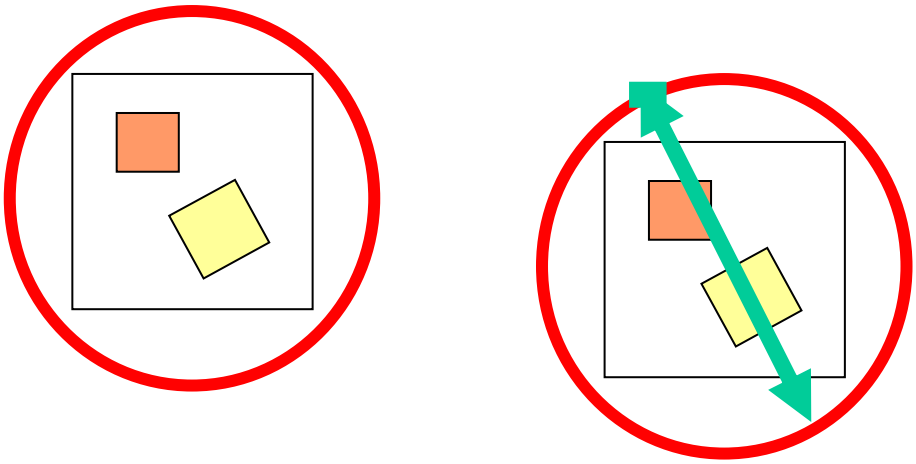


*Figure 47. Global directions*

For each selected point we traverse the scene using a technique called "*Depth Peeling*". This technique consists in peeling layers of the scene using the depth of the previous layer. In the following figure

you can see the layers that would be created traversing the scene with the RGB components in the pictures with blue background and the depth in the pictures with white background (Figure 48).

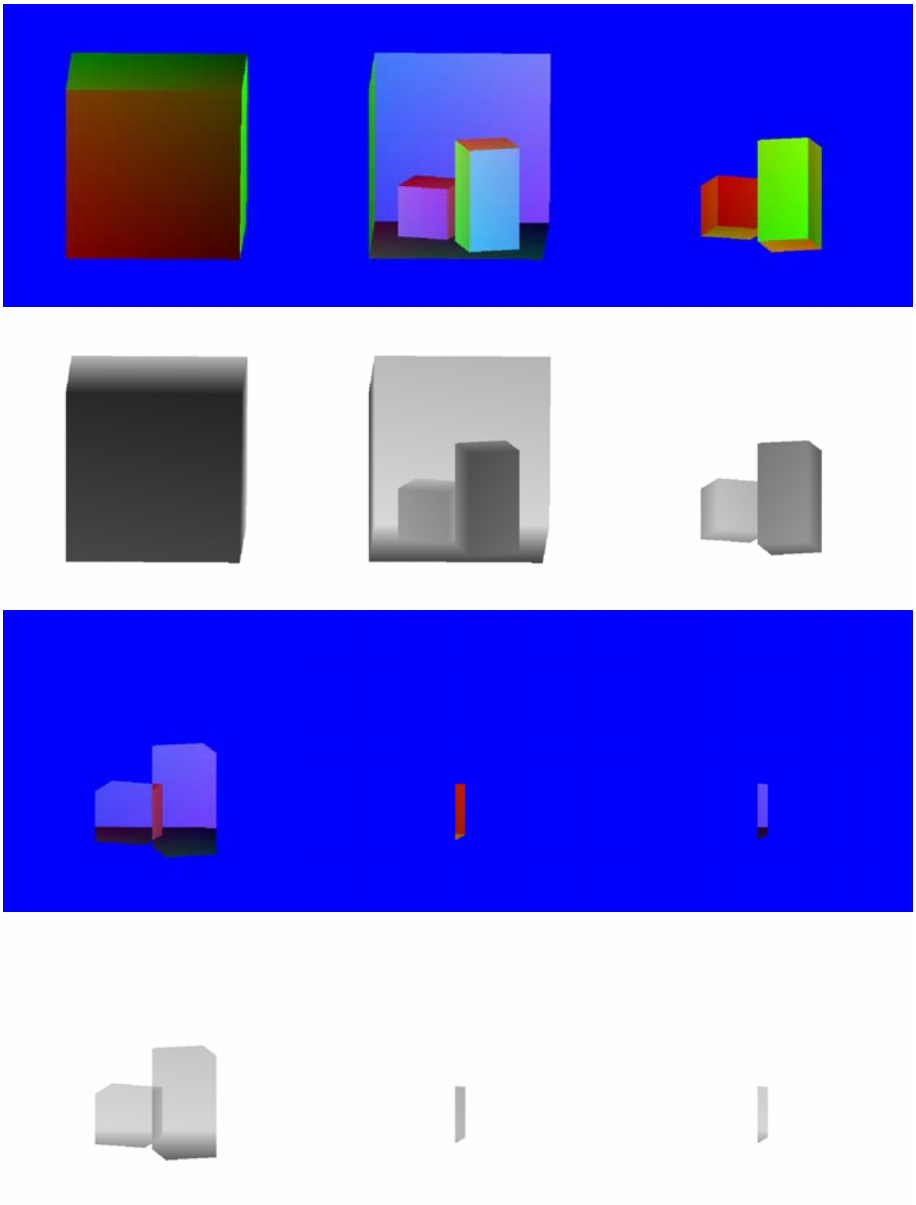


*Figure 48. Peel Layers*

Once the layers are obtained, we start calculating the obscurance transfer between layers. To have transfer between patches in a given direction, both patches have to be projected onto the same position in neighboring layers, have to look at each other, and be at a distance smaller than a user defined threshold $d_{max}$.

If these conditions are met, the transferred obscurance will be given by the following function (Figure 49):

$\sqrt{d/d_{\max}}$ if $d<d_{max}$ and 1 otherwise.

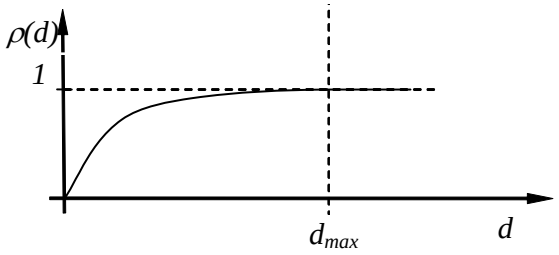



*Figure 49. Transfer function*

Once the obscurances are calculated for several directions we obtain the obscurances map (Figure 50). This map has to be normalized. This normalization consists in dividing the RGB components of each texel by its Alpha component.

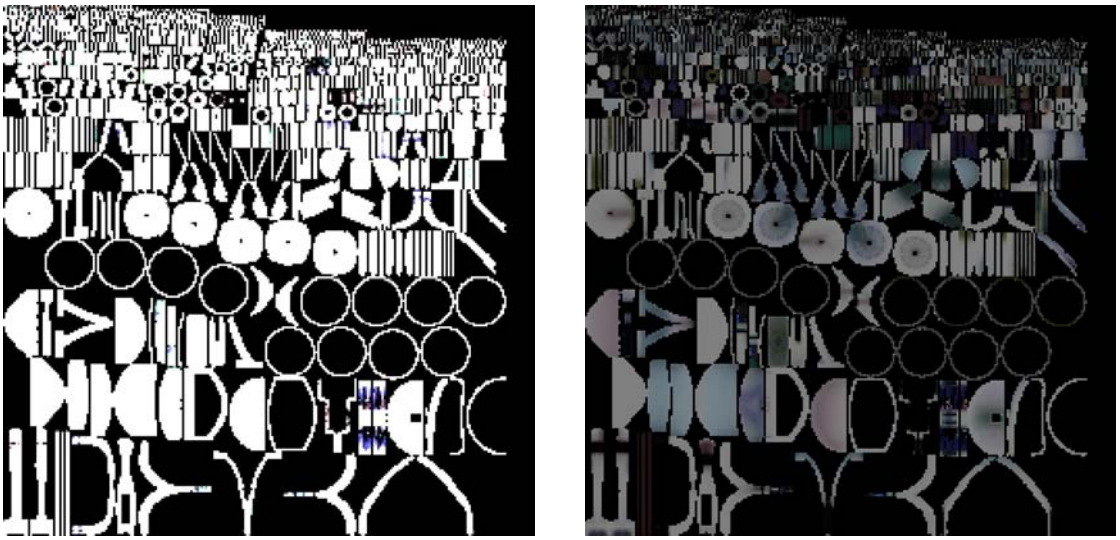
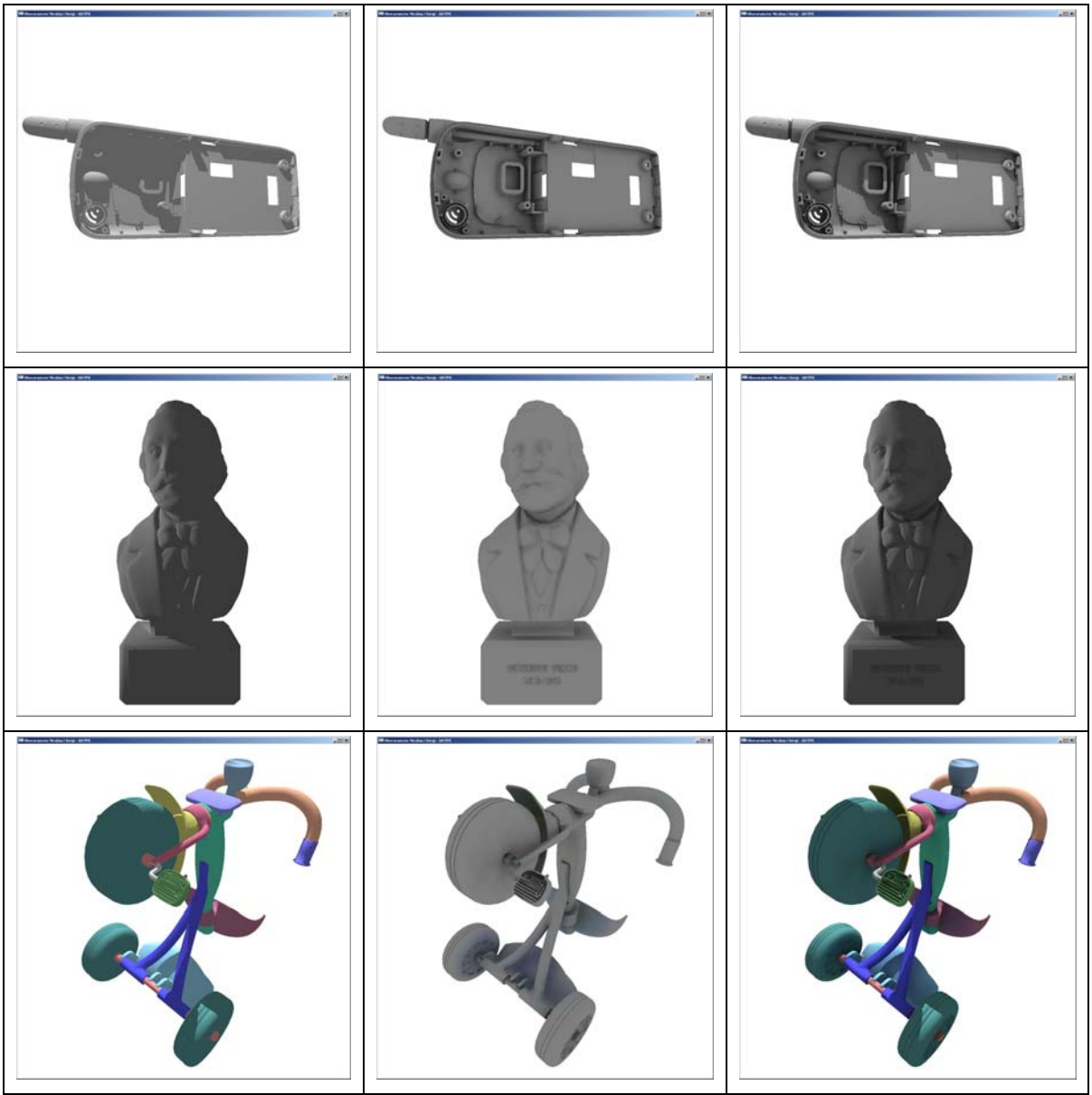


*Figure 50. Obscurance maps*

The obscurance map can then be applied as a kind of lightmap.



| Direct + Ambient | + Obscurance Map | = Final Result |
|---|---|---|

*Figure 51. Final results generated by the obscurances method*

### 2.10.3. Ogre integration

The *GPUObscurances* module has not been embedded into Ogre, it has been instead implemented as a standalone application which loads Ogre's mesh and XML files and generates an image file containing the Obscurance information. This image file can be applied to Ogre scenes as a lightmap.

### 2.10.4. Implementation details

The most important classes of the application are `vcObscuranceMap` and `CMesh`. The first contains all the necessary information and functionality to compute the obscurance map and has a method that returns a pointer to the resulting image (`vcObscurerGenerateImage`). `CMesh` contains the geometry of the scene and carries the scene parsing plus the hardware-accelerated structures used in the process to generate the obscurance map.

Some of the parameters that can affect the performance of the application and the image quality of the result are already adjusted. The number of directions taken is set to 180 (3 steps * 60 directions / step), but this number can be modified changing the step and iteration variables in the code. Changing steps is not a problem. The greater the number, the better the result you achieve, but it takes longer to finish. But if you try to increment the iterations per step it can result in some image glitches due to saturation of the `fp16` values used for accumulative blending.

Another variable that can be changed is $d_{max}$. The default is 0.3. The variable can be found in the `RenderTransfer` function.

Another parameter that can be tweaked is the relative resolution between the projection planes used to calculate obscurance transfer and the resolution of the resulting obscurance map. This can be done in `vcObscurerGenerateImage` at the definition of `ResX` and `ResY`. Now they are set to 2.0 so the projection images have double the resolution of the desired obscurance map. This can be changed to 4.0 or 8.0 but then you have to decrease the iterations variable to avoid the saturation of fp16 buffer used for accumulation.

Finally, there can be some problems while filtering the obscurance map. There are three ways to solve that:

- The first method is to expand the charts of the resulting obscurance map.
- In the first method the obscurance map created by the application is normalized (RGB components are divided by A component). If the application does not normalize components and the normalization is done on the GPU just before visualizing the model with obscurances, filtering problems will be avoided and chart expansion will not be necessary anymore.
- A third solution can be not using the hardware filtering and program the filtering in the shader. This filtering shader should discard all the samples with zero alpha component.

### 2.10.5. Using the standalone obscurances generation module

The application has a GUI that guides the user through four steps in order to set up the parameters needed to calculate the lightmap.

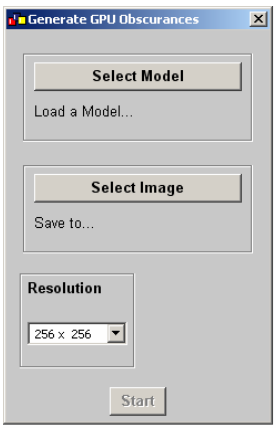



*Figure 52. GUI*

- The first step consists of selecting the mesh we want to calculate the obscurance map for.
- The second step asks you to choose the name and destination folder of the resulting image.
- The third step lets you choose the desired image resolution.
- Finally you only have to push the Start button. A progress bar will appear. After completion of the progress bar, you will have your obscurance map stored as a .bmp in the destination folder you choose in step 2.

Once finished all the steps you get an image that contains the obscurance map. This image can be used as the indirect lighting of your scene.

*Figure 53. Obscurance map*

## 2.11. IBR BILLBOARD CLOUD TREE MODULE

### 2.11.1. IBR Billboard Cloud Tree Generator

This is a standalone module has been integrated into the Ogre engine.

#### *2.11.1.1. Role of the module*

The *IBR Billboard Cloud Tree Generator* provides a set of tools for creating and previewing 3D tree models for being used in real-time applications. The generated tree is thus represented by a set of billboards, called *billboard cloud*. The billboards are built automatically by a clustering algorithm. Unlike classical billboards, the billboards of a billboard cloud are not rotated when the camera moves, thus the expected occlusion and parallax effects are provided. On the other hand, this approach allows the replacement of a large number of leaves by a single semi-transparent quadrilateral, which considerably improves the rendering performance. A billboard cloud well represents the tree from any direction and provides accurate depth values, thus the method is also good for shadow. The billboard cloud decomposes the original object into subsets of patches and replaces each subset by a billboard. These billboards are fixed and the final image is the composition of their images.

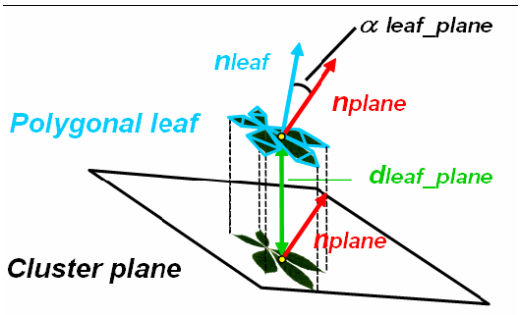



*Figure 54. Billboard image generation.*

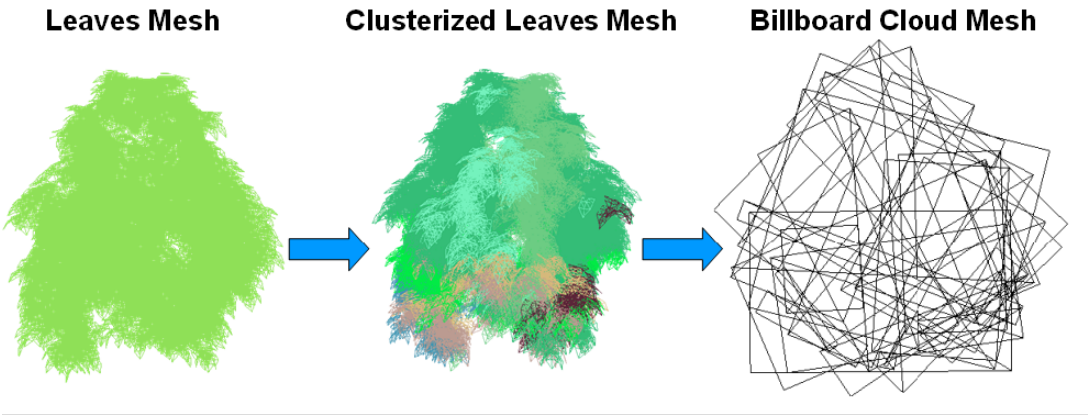



*Figure 55. Decomposition of the original leaves model.*

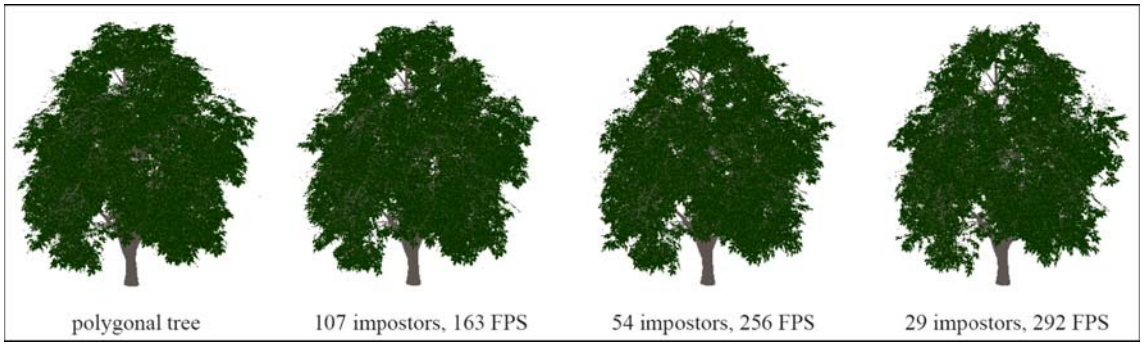



*Figure 56. Visual comparisons of the original polygonal tree and the tree rendered with different number of billboards.*

### *2.11.1.2. Application components overview*

The *IBR Billboard Cloud Tree Generator* have been developed using the Ogre3D graphic engine, being compatible with both, the OpenGL and Direct3D render systems, the other optional dependency is OpenEXR. The system has been tested under Windows but there shouldn't be big problems for making it to work under Linux or Mac platforms.

### *2.11.1.3. Using the IBR Tree module*

Creating a tree suitable for real-time rendering must take into account some restrictions, basically the polygon count and texture memory used. The sample tree shown has 11291 leaves defined by 112910 faces and 338731 vertices. The trunk of the tree has 46174 faces. Current graphics hardware cannot handle a forest using this tree model. In those cases this tool is necessary.

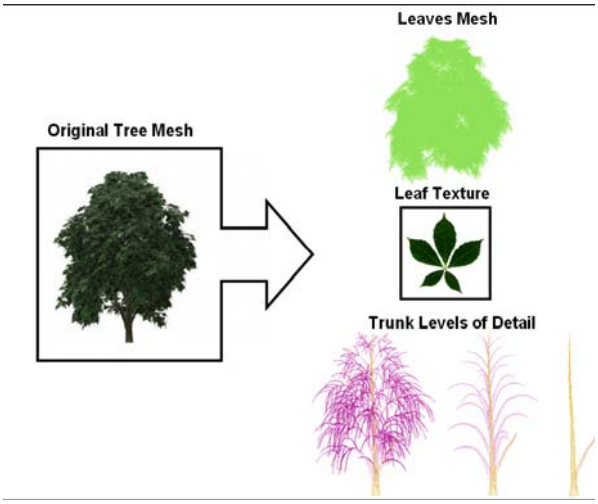



*Figure 57. Sample polygonal tree decomposed in two models. The leaves model will be processed with the IBR Billboard Cloud Tree Generator.*

1. The 3d content creator at first have to create a high detailed tree with some of the available tools, such as *Xfrog* or *PovTree*.
2. Then the high-detailed tree model should be decomposed into the leaves and the trunk.
3. The leaves model must be converted into the Ogre3D Mesh file format, there are exporters available for the main 3D DCC, such as Maya or 3ds Max. Check the Ogre 3D webpage (http://www.ogre3d.org) in the Downloads / Tools section.

The trunk model won't be processed with the generator tool and the designer must create a set of levels of detail of the original mesh using a simplification software. The *IBR Billboard Cloud Tree Generator* receives as input the leaves model and the texture used to map each leaf.

The output generated is a billboard cloud mesh and three texture atlases as described here:

1. The billboard cloud model that replaces the original leaves model. The billboard cloud model will be used in with two techniques later, in the in game visualization.
2. The texture atlas that contains the diffuse color texture atlas has all the textures that will be used with the billboard cloud mesh. This texture will be used by the lower-end technique with standard texture mapping.
3. The rotated leaf texture atlas contains the leaf placed with different orientations.
4. The leaves distribution atlas texture. This texture will be used with the rotated leaf texture atlas in the technique *indirect texturing technique*.

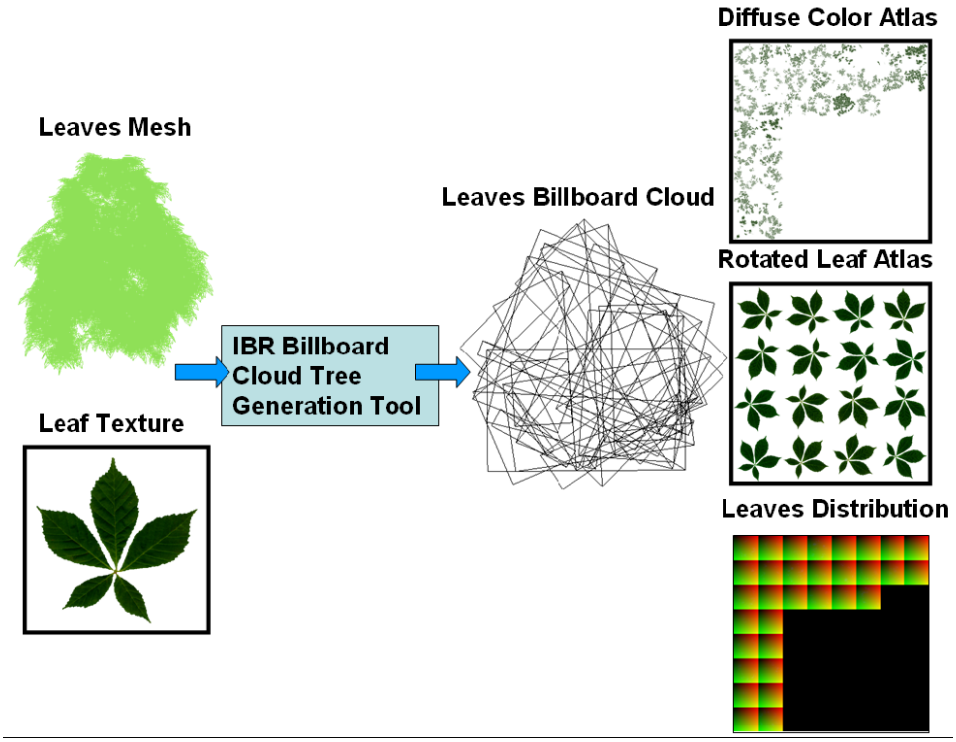



*Figure 58. Texture atlas generation*

The *IBR Billboard Cloud Tree Generator*, is a command line application managed through configuration files. For each kind of tree we want to process we should create a configuration file. The application must have as input parameter the configuration file as shown here:

```
IBRBillboardCloudTreeGeneratorCmd.exe –cfg
../../media/chestnut/leaves/sample.cfg
```

These configuration files contain all the parameters you have to specify in order to obtain the desired output.

Those parameters can be summarized as:

- The media folder parameters.
- The input file parameters.
- The output file parameters.
- The user custom parameters.

### *2.11.1.4. Media folders*

Input folders:

```
# Folder where the sample input leaves model can be found …
Entities Folder ../../ media/chestnut/leaves/
# Folders where the sample temporary data will be stored …
Entity Distribution Folder ../ ../ media/chestnut/leaves/
Entity Clusters Folder ../ ../media/chestnut/leaves/
# Folder where the sample output files will be stored …
Billboard Cloud Folder ../../media/chestnut/leaves/
```

Temporary folders:

```
# Folders where the sample temporary data will be stored …
Entity Distribution Folder ../ ../media/chestnut/leaves/
Entity Clusters Folder ../ ../media/chestnut/leaves/
```

Output folders:

```
# Folder where the sample output files will be stored …
Billboard Cloud Folder ../../media/chestnut/leaves/
Input files parameters
# This is the sample input leaves model filename
Entities Mesh Name chestnutLeaves.mesh
# This is the sample input leaf texture filename
Entity Clusters Grouped Texture Unit 0 Name chestnutLeaf.png
```

### *2.11.1.5. Output files parameters*

Rotated leaf texture atlas parameters:

```
# This is the sample input leaf texture filename
Diffuse Color Entity Texture Name chestnutLeaf.png
# The output texture atlas file name
Diffuse Color Entity Texture Atas Name chestnutRotatedLeafAtlas.png
# The output texture atlas bit range (8 bits , 16 bits , 32 bits per
# channel )
Diffuse Color Entity Texture Atlas Bit Range 8
# The output texture altas size in pixels
Diffuse Color Entity Texture Atlas Size 512
# The number of  textures with different leaf orientations we want
Diffuse Color Entity Texture Atlas NumSamples 16
```

Diffuse color texture atlas parameters:

```
# The output texture atlas file name
Diffuse Color Texture Atlas Name diffuseColorAtlas.png
# The output texture atlas bit range (8 bits , 16 bits , 32 bits per
# channel )
Diffuse Color Texture Atlas Bit Range 8
# The output texture altas size in pixels
Diffuse Color Texture Atlas Size 1024
# The size that must have each billboard texture in the texture atlas
Diffuse Color Texture Size 16
```

Leaves distribution texture atlas parameters:

```
# The output texture atlas file name
Indirect Texture Atlas Name indirectTextureAtlas.png
# The output texture atlas bit range (8 bits , 16 bits , 32 bits per
#channel )
Indirect Texture Atlas Bit Range 8
# The output texture altas size in pixels
Indirect Texture Atlas Size 1024
# The size for each billboard texture in the texture atlas
Indirect Texture Size 16
```

User custom parameters

```
# This is the maximum number of billboards desired
EntityClusters MaxClusters 1024
```

Note: The IBR Billboard Cloud Tree Generator has more custom parameters, for more details read the user manual included with the tool.

#### *2.11.1.6. View Modes*

The IBR Billboard Cloud Tree Generator has three different view modes:

- *Diffuse Color Texture Atlas View Mode*: This view mode shows the diffuse color texture atlas generation process, and each billboard plane with its texture mapped leaves.
- *Indirect Texture Atlas View Mode*: This view mode shows the leaves distribution impostor texture atlas generation.
- *Rotated Leaf Texture Atlas View Mode*: This view mode shows the rotated leaf texture atlas.
- *Billboard Cloud Final View Mode*: In this mode you can look at your billboard cloud mesh using indirect texturing or standard texture mapping techniques, is good to check that the resulting billboard cloud tree has been generated correctly. Here we can apply a tweak into the indirect texturing leaf position.

#### *2.11.1.7. Keyboard Controls*

Views controls:

• `Key F1:` Enable Diffuse Color Texture Atlas View Mode.

• `Key F2:` Enable Indirect Texture Atlas View Mode.

• `Key F3:` Enable Rotated Leaf Texture Atlas View Mode.

• `Key F4:` Enable Billboard Cloud Final View Mode.

• `SPACE:` This key iterates along all the billboards generated to see how each group of leaves has been placed on them. This key is working only in the Diffuse Color Texture Atlas View Mode and Indirect Texture Atlas View Mode.

Navigation controls:

These controls are enabled only in the Billboard Cloud Final View Mode. With them you can go around the billboard cloud, using the mouse as the looking forward direction.

- `Key E/Key D:` Accelerate/Brake
- `Key S/Key F:` Turn
- `Key PGUP/Key PGDOWN:` Shift

The next version will handle `Collada` as input/output file format for more compatibility with the 3D DCC tools.

### 2.11.2. Using IBR Billboard Cloud Trees in Games

For using the billboard cloud trees in games, two techniques are shown:

- *Standard texturing technique*: Use only diffuse color texture atlas. This technique is for low-end graphic cards.
- *Indirect texturing technique*: This technique uses the rotated leaf texture atlas and the leaves distribution texture atlas in order to have high resolution rendered leaves.

We will cover only the *Indirect texturing technique* in more detail, because the *direct texturing technique* is straightforward.

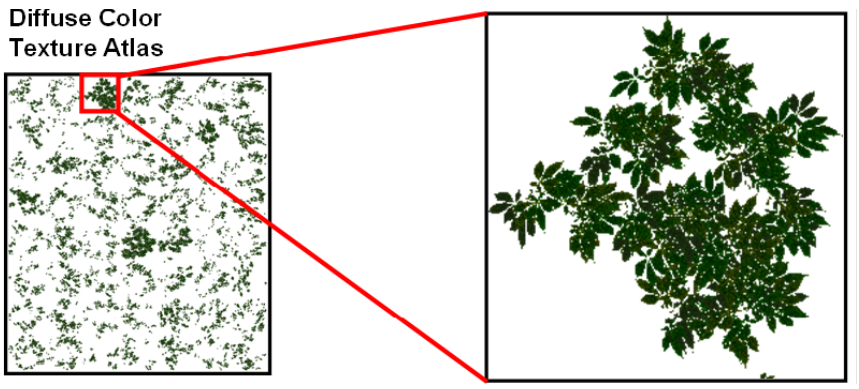



*Figure 59. Diffuse color texture atlas that contains all the billboard textures. This texture atlas is used in the standard texturing technique.*

#### *2.11.2.1. Indirect texturing technique*

The indirect texturing technique is a one pass technique. The billboard cloud is the input geometry. The billboard cloud model generated with the *IBR Billboard Cloud Tree Generator* encodes some information in the `TEXCOORD0` and `COLOR0` parameters.

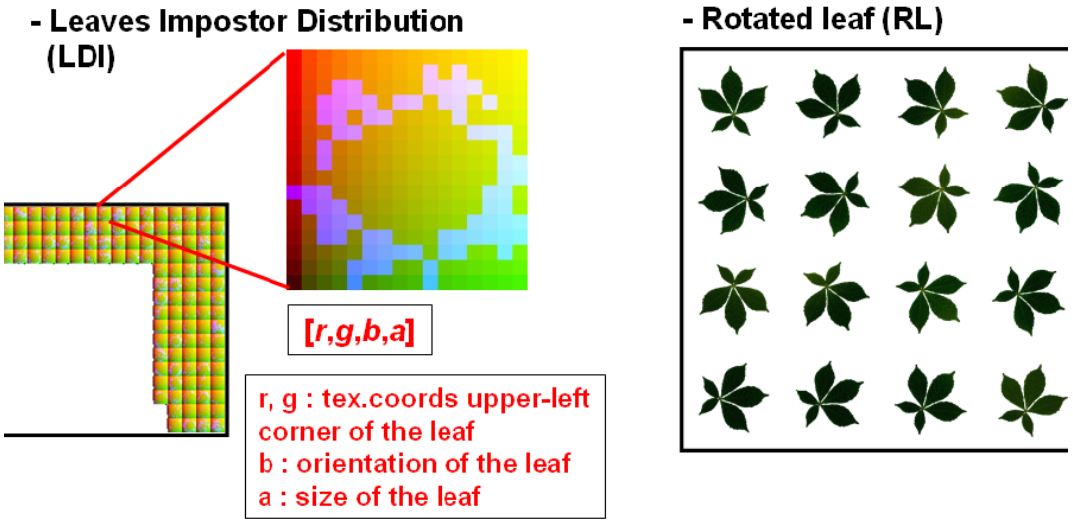



*Figure 60. Leaves distribution texture atlas and rotated leaf texture atlas. These texture atlases are used in the Indirect texturing technique.*

- `COLOR0` encodes the uniform texture coordinates of each the billboard from [0,0] to [1,1]. They are used when computing which texel will be looked up from the rotated leaf texture atlas.
- `TEXCOORD0` encodes the texture coordinates for looking up the texture information of the leaves distribution texture atlas for each billboard.

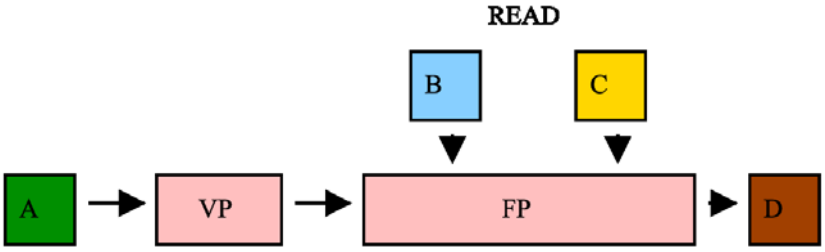



*Figure 61. Indirect texturing rendering pass. A is the billboard cloud geometry, B is the rotated leaf texture atlas, C is the leaves distribution texture, and D is the frame buffer.*

During the rendering pass, the leaf distribution texture is accessed using the `TEXCOORD0(u,v)` coordinates.

(a) If the value stored in the `b` channel is equal to 0, then this texel is not indexing a leaf position inside the billboard. Therefore it will be rendered as a transparent texel on the billboard plane.

(b) 1. Else if the `b` channel is not equal to 0, the stored coordinates at `(r,g)` will be the lower-left coordinates of the position where the leaf will be shaded.

2. Using the leaf orientation value, stored in `g`, we will select the desired rotated leaf texture atlas.

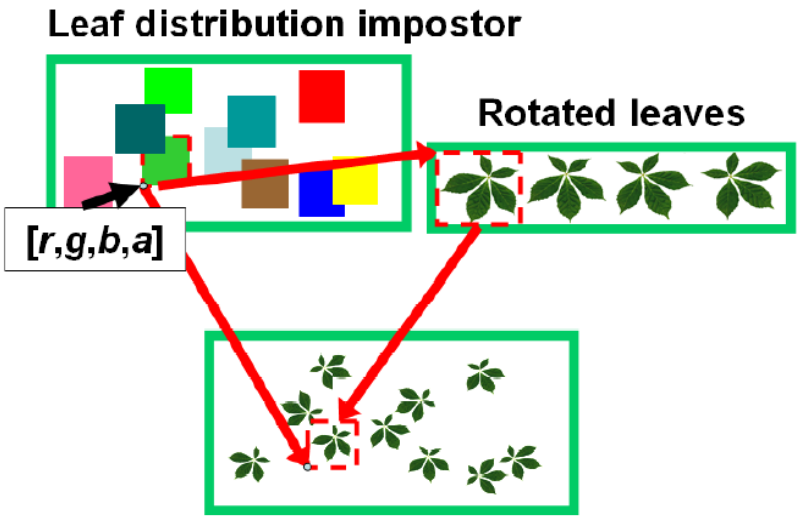



*Figure 62. Indirect texturing. The billboard is replaced by a leaf distribution impostor and the rotated images of the leaves.*

The *billboard cloud trees* have been used in two applications:

Ogre Demo: The demo has been tested under OpenGL and DirectX using GLSL/Cg Cg/HLSL shaders respectively.

RenderMonkey Effect: This demo has the same effects as the Ogre3D Demo, using GLSL/HLSL shaders. This application should be the easiest to understand for those developers who want to use this technique with other game engines and applications.

*Figure 63. Screenshot of the Ogre Demo application.*

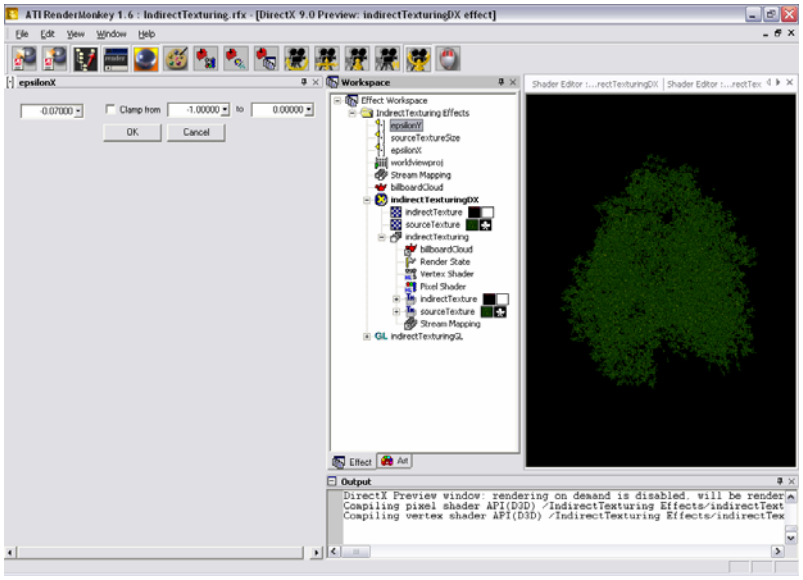



*Figure 64. Screenshot of the RenderMonkey indirect texturing effect.*

The indirect texturing technique is good also for shadows. The Ogre Demo has been implemented using depth mapped shadows.

*Figure 65. Ogre Demo application with shadow mapping.*

### *2.11.2.2. Implementation Issues*

One problem that can appear is due to the wrong placement of the leaf texture, this can be avoided adjusting an epsilon shift displacement.


*Figure 66. The first picture shows the result of the indirect texturing before fixing the shift displacement. In the second picture all the leaves are placed correctly.*

The epsilon value must be tweaked for the generation of each kind of different tree.