GPUGI: Global Illumination Effects on the GPU

László Szirmay-Kalos

Budapest University of Technology and Economics, Budapest, Magyar Tudósok krt. 2., H-1117, HUNGARY
Email: szirmay@iit.bme.hu
URL: http://www.iit.bme.hu/˜szirmay

László Szécsi
Budapest University of Technology and Economics, Budapest, Magyar Tudósok krt. 2., H-1117, HUNGARY
Email: szecsi@iit.bme.hu
URL: http://www.iit.bme.hu/˜szecsi

Mateu Sbert

University of Girona, Campus Montilivi, Edifici PIV, 17071 Girona, Spain
Email: mateu@ima.udg.es
URL: http://ima.udg.es/˜mateu

Abstract
In this tutorial we explain how global illumination rendering methods can be implemented on Shader Model 3.0 GPUs. These algorithms do not follow the conventional local illumination model of DirectX/OpenGL pipelines, but require global geometric or illumination information when shading a point. In addition to the theory and state of the art of these approaches, we go into the details of a few algorithms, including mirror reflections, reflactions, caustics, diffuse/glossy indirect illumination, precomputation aided global illumination for surface and volumetric models, obscurances and tone mapping, also giving their GPU implementation in HLSL or Cg language.

Keywords: Global illumination, GPU programming, HLSL, Radiosity, Soft shadow algorithms, Environment mapping, Diffuse/Glossy indirect illumination, Mirror Reflection/Refraction, Caustics generation, Monte-carlo methods, Pre-computation aided global illumination, PRT, Ambient Occlusion, Obscurances, Participating Media, Multiple scattering.

Contents of the tutorial

This tutorial presents techniques to solve various subproblems of global illumination rendering on the Graphics Processing Unit (*GPU*). The state of the art is discussed briefly and we also go into the details of a few example methods. Having reviewed the global illumination rendering problem and the operation of the rendering pipeline of the GPUs, we discuss six categories of such approaches.

1. *Simple improvements of the local illumination lighting model.* First, to warm up, we examine two relatively simple extensions to the local illumination rendering, *shadow mapping* and *image based lighting*. Although these are not considered global illumination methods, they definitely represent the first steps from pure local illumination rendering toward more sophisticated global illumination approaches. These techniques already provide some insight on how the basic functionality of the local illumination pipeline can be extended with the programmable features of the GPU.
2. *Ray-tracing.* Here we present the implementation of the classic ray-tracing algorithm on the GPU. Since GPUs were designed to execute rasterization based rendering, this approach fundamentally reinterprets the operation of the rendering pipeline.
3. *Specular effects with rasterization.* In this section

we return to rasterization and consider the generation of *specular effects*, including *mirror reflections*, *refractions*, and *caustics*. Note that these methods are traditionally rendered by ray-tracing, but for the sake of efficient GPU implementation, we need to generate them with rasterization. Having surveyed the proposed possibilities, we concentrate here on the method called *Approximate ray tracing with distance impostors*.

4. *Diffuse/glossy indirect illumination.* This section deals with non-specular effects, which require special data structures stored in the texture memory, from which the total diffuse/glossy irradiance for an arbitrary point may be efficiently retrieved. Of course, these representations always make compromises between accuracy, storage requirements, and final gathering computation time. We present two algorithms in detail. The first is the implementation of the *stochastic radiosity* algorithm on the GPU, which stores the radiance in a color texture. The second considers final gathering of diffuse and glossy indirect illumination using *localized cube maps*.
5. *Pre-computation aided global illumination.* These algorithms pre-compute the effects of light paths and store these data compactly in the texture memory for later reuse. Of course, pre-computation is possible if the scene is static. Then during the real-time part of the process, the actual lighting is combined with the prepared data and real-time global illumination results are provided. Having presented the theory of *finite element methods* and *sampling*, we discuss three methods in details: *Pre-computed radiance transfer* (*PRT*) using finite-element representation, *Light path maps* that are based on sampling, and *Participating media illumination networks*, which again use sampling.
6. *Fake global illumination.* There are methods that achieve high frame rates by simplifying the underlying problem. These approaches are based on the recognition that global illumination is inherently complex because the illumination of every point may influence the illumination of every other point in the scene. However, the influence diminishes with the distance, thus it is worth considering only the local neighborhood of each point during shading. Methods using this simplification include *obscurances* and *ambient occlusion*, from which the first is presented in details. Note that these methods are not physically plausible, but provide satisfying results in many applications.

When the particular methods are discussed, images and rendering times are also provided. If it is not stated explicitly, the performance values (e.g. frames per second) have been measured on an Intel P4 3 GHz PC with 1GB RAM and NVIDIA GeForce 6800 GT graphics card in full screen mode (1280×1024 resolution).

Notations

In this tutorial we tried to use unified notations when discussing different approaches. The most general notations are also listed here.

- $L(\vec{x},\vec{\omega})$: the radiance of point $\vec{x}$ at direction $\vec{\omega}$.
- $L^r(\vec{x},\vec{\omega})$: the reflected radiance of point $\vec{x}$ at direction $\vec{\omega}$.
- $L^e(\vec{x},\vec{\omega})$: the emission radiance of point $\vec{x}$ at direction $\vec{\omega}$.
- $L^{env}(\vec{\omega})$: the radiance of the environment illumination from direction $\vec{\omega}$.
- $f_r(\vec{\omega}',\vec{x},\vec{\omega})$: BRDF function at point $\vec{x}$ for illumination direction $\vec{\omega}'$, viewing direction $\vec{\omega}$. If the surface is diffuse, the BRDF is denoted by $f_r(\vec{x})$.
- θ': the angle between the illumination direction and the surface normal.
- $\vec{x}$: the point to be shaded, which is the receiver of the illumination.
- $\vec{y}$: the point that is the source of the illumination.
- $v(\vec{x},\vec{y})$: visibility indicator which is 1 if points $\vec{x}$ and $\vec{y}$ are visible from each other and zero otherwise.
- `World`: a uniform parameter of the shader program of type `float4x4`, which transforms from modeling to world space.
- `WorldIT`: a uniform parameter of the shader program of type `float4x4`, the inverse-transpose of `World`, used to transform normal vectors from the modeling space to the world space. This matrix is also used to transform rays from world space to modeling space, transposed in shader.
- `WorldView`: a uniform parameter of the shader program of type `float4x4`, which defines the transformation matrix for place vectors (points) from the modeling space to the camera space
- `WorldViewIT`: a uniform parameter of the shader program of type `float4x4`, which defines the inverse-transpose of `WorldView`, used to transform normal vectors from the modeling space to the camera space.
- `WorldViewProj`: a uniform parameter of the shader program of type `float4x4`, which defines the transformation matrix from the modeling space to the clipping space.
- `DepthWorldViewProj`: a uniform parameter of the shader program of type `float4x4`, which defines the transformation matrix from the modeling space to the clipping space used when rendering the depth map.
- `DepthWorldViewProjTex`: a uniform parameter of the shader program of type `float4x4`, which defines

the transformation matrix from the modeling space to the texture space of the depth map.

- `EyePos`: a uniform parameter of the shader program of type `float3`, which defines the camera position in world space.
- `Pos, wPos, cPos, hPos`: vertex or fragment positions in modeling, world, camera and clipping spaces, respectively.
- `Norm, wNorm, cNorm`: vertex or fragment normals in modeling, world and camera spaces, respectively.
- `oColor, oTex`: vertex shader output color and texture coordinates.

1. Global illumination rendering

Global illumination algorithms should identify all light paths connecting the eye and the light sources via one or more scattering points, and should add up their contribution to obtain the power arriving at the eye through the pixels of the screen [Kaj86]. The scattering points of the light paths are on the surface in case of opaque objects, or can even be inside of translucent objects (*subsurface scattering* [JMLH01]).

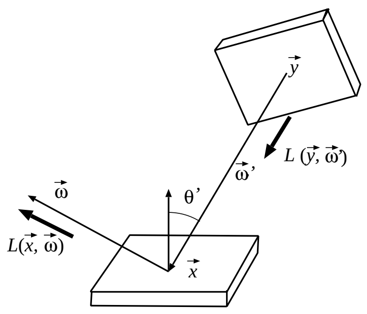


Figure 1: *Notations of the computation of the reflected radiance*

Let us first consider just a single scattering or one-bounce light transport (figure 1). Denoting the radiance of point $\vec{y}$ in direction $\vec{\omega}'$ by $L(\vec{y},\vec{\omega}')$, reflected radiance $L^r(\vec{x},\vec{\omega})$ at scattering point $\vec{x}$ is the sum of contributions from all incoming directions $\vec{\omega}'$:

$$L^r(\vec{x},\omega) = \int_{\Omega'} L(\vec{y},\vec{\omega}') \cdot f_r(\vec{\omega}',\vec{x},\vec{\omega}) \cdot \cos^+ \theta'_{\vec{x}} \, d\omega', \quad (1)$$

where $\vec{y}$ is the point visible from $\vec{x}$ at direction $-\vec{\omega}'$, Ω' is the directional sphere, $f_r(\vec{\omega}',\vec{x},\vec{\omega})$ is the bi-directional reflection/refraction function (*BRDF*), and $\theta'_{\vec{x}}$ is the angle between the surface normal and direction $-\omega'$ at $\vec{x}$. If $\theta'_{\vec{x}}$ is greater than 90 degrees, then the negative cosine value should be replaced by zero, which is indicated by superscript $^+$.

In order to consider not only single bounce but also multiple bounce light paths, the same integral should also be recursively evaluated at visible points $\vec{y}$, which leads to a sequence of high dimensional integrals:

$$L^r = \int_{\Omega'_1} f_1 \cos^+ \theta'_1 \left(L^e + \int_{\Omega'_2} f_2 \cos^+ \theta'_2 \cdot (L^e \ldots) \, d\omega'_2 \right) d\omega'_1 \quad (2)$$

where L^e is the emission radiance.

A straightforward technique to compute high-dimensional integrals is the *Monte Carlo* (or quasi-Monte Carlo) method [Sob91, SK99a, DBB03], which generates finite number of random light paths and approximates the integral as the sum of the individual path contributions divided by the probability of generating this sample path.

Accurate results need a huge number of light paths. For comparison, in reality a 100 W electric bulb emits about 10^{42} number of photons in each second, and the nature "computes" the paths of these photons in parallel with the speed of the light independently of the scene complexity. Unfortunately, when it comes to computer simulation, we shall never have 10^{42} parallel processors running with the speed of light. It means that the number of simulated light paths must be significantly reduced and we should accept longer rendering times. For real-time applications, the upper limit for rendering times comes from the requirement that to maintain interactivity and to provide smooth animations, computers must generate at least 20 images per second. Note that on an 1000×1000 resolution display this allows 50 nsec to compute the light paths going through a single pixel.

To meet this performance requirement, the problem to be solved is often simplified. One popular simplification approach is the *local illumination model* that ignores *indirect illumination* (figure 2). The local illumination model examines only one-bounce light paths having a single scattering point and thus can use only local surface properties when the reflection of the illumination of a light source toward the camera is computed. In local illumination shading, having obtained the point visible from the camera through a pixel, the reflected color can be determined without additional geometric queries. In the simplest case when even shadows are ignored, visibility is needed only from the point of the camera. To solve such visibility problems, the GPU rasterizes the scene and finds the visible points using the *z-buffer* hardware.

Global illumination algorithms also compute indirect illumination. It means that we need visibility information not only from the camera but from every

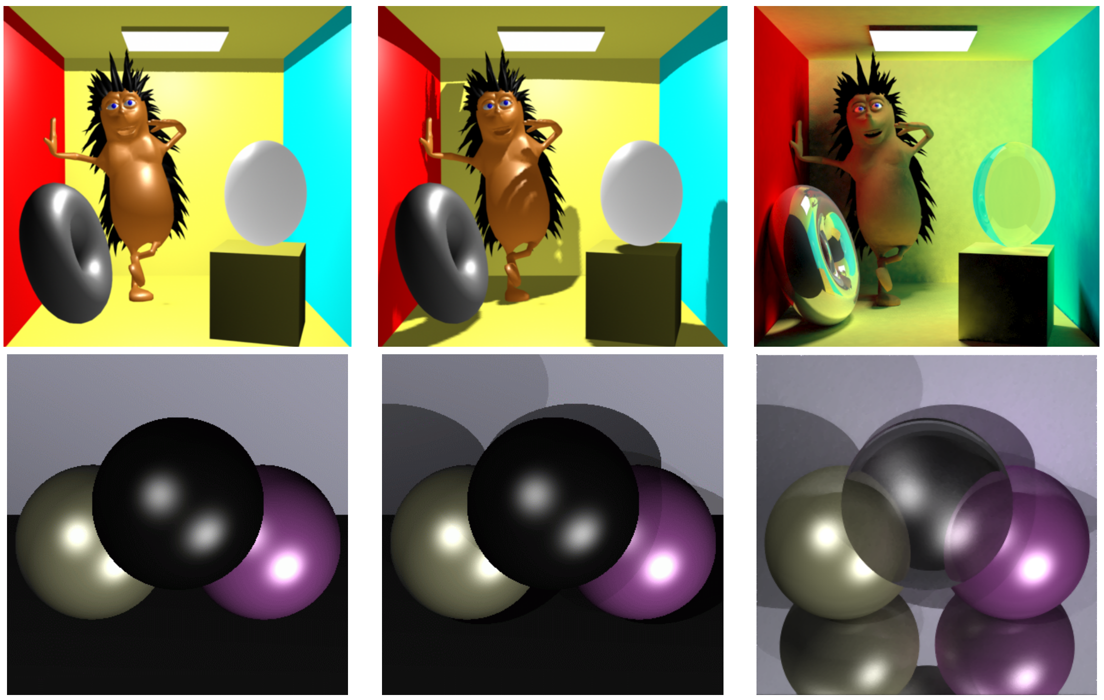


Figure 2: *Comparison of local illumination rendering (left), local illumination with shadows (middle), and global illumination rendering (right)*

shaded point. This is a requirement GPUs are not built for, which makes GPU based global illumination algorithms hard and challenging.

One option is to follow the research directions of the era when global illumination was fully separated from the graphics hardware and its algorithms were running on the CPU, and try to port those algorithms onto the GPU. For example, it is time to revisit the research on efficient ray-shooting and to consider what kind of space partitioning schemes and algorithms can be implemented on the GPU [PBMH02b, PDC*03, OLG*05, FS05]. Another family of techniques that has been proven to be successful in CPU implementations recognizes that it is not worth generating paths completely independently from scratch, but the visibility and illumination information gained when generating a path should be *reused* for other paths as well. *Photon mapping* [Jen96], *instant radiosity* [Kel97], and deterministic or stochastic iterative radiosity [SK99b] all reuse parts of the previously generated paths to speed up the computation. Furthermore, if the scene is static, then paths can be *pre-computed* only once and reused during rendering without repeating the expensive computation steps. Reuse and pre-computation are promising techniques in GPU based real-time global illumination algorithms as well.

On the other hand, GPUGI is not just porting already known global illumination algorithms to the GPU. The GPU is a special purpose hardware, so to efficiently work with it, its special features and limitations should also be taken into account. This consideration may result in solutions that are completely different from the CPU based methods.

2. Local illumination rendering pipeline of current GPUs

2.1. Evolution of the fixed function rendering pipeline

Current, highly programmable graphics hardware evolved from simple monitor adapters. Their task was barely more than to store an image in memory, and channel the data to control the electron beam lighting monitor pixels. Already this had to be done at a then incredible speed, making it a feat that could only be achieved through parallelization. However, raster adapters began their real revolution when they started supporting incremental 3D graphics, earning them the name graphic accelerators. Indeed, the first achievement was to implement *linear interpolation* in hardware very effectively, by obtaining values in consecutive pixels using a single addition [SKe95].

The coordinates of the internal points of a *triangle* can be obtained by linearly interpolating the vertex coordinates. Other attributes, such as color, texture coordinates, etc. can be approximated by linear interpolation. Thus, when a triangle is *rasterized*, i.e. its corresponding pixels are found, linear interpolation can be used for all data.

Any virtual world description can be translated to a set of triangle mesh surface models with some level of fidelity. The basic assumption of incremental 3D rasterization is that we render triangles, defined by triplets of vertices, the position of which is given as 3D vectors, in coordinates relative to the screen (later this space will be referred to as *normalized device space* or *clipping space* depending on whether Cartesian or homogeneous coordinates are used). Thus, the third Cartesian coordinate, z, denotes depth. For every vertex, a color is given, which is obtained from shading computations. The array in graphics memory containing records of vertex data (position and color) is called the *vertex buffer*.

When rendering, every triangle is clipped to the viewport and rasterized to a set of pixels, in which the color and the depth value is computed via incremental linear interpolation. Besides the *color buffer memory* (also called *frame buffer*), we maintain a *depth buffer*, or *z-buffer*, containing depth related to the current color value in the color buffer. Whenever a triangle is rasterized to a pixel, the color and depth are only overwritten if the new depth value is less, meaning the new triangle fragment is closer to the viewer. As a result, we get a rendering of triangles correctly occluding each other in 3D. The quality of shading depends on how we computed the colors for the vertices. But even if those are highly accurate, colors will be smeared over triangles, meaning that image quality depends on the level of tessellation. Furthermore, every pixel will be filled with a uniform color, computed for a single surface point. Therefore, aliasing artifacts, jagged edges will appear, making image quality highly dependent on image resolution, too. The rendering time already depends linearly on two factors: the number of triangles to be rendered, and the number of pixels to be colored, whichever is the bottleneck.

Triangle mesh models have to be very detailed to offer a realistic appearance. An ancient and essential tool to provide the missing details is *texture mapping*. Texture images are stored in graphics card memory as 2D arrays of color records. How the texture should be mapped onto triangle surfaces is specified by texture coordinates assigned to every vertex. Thus, the vertex buffer does not only contain position and color data, but also texture coordinates. These are linearly interpolated within triangles just like colors, and for every pixel, the interpolated value is used to fetch the appropriate color from the texture memory. A number of filtering techniques combining more texel values may also be applied. Then the texture color is used to modulate the original color received from the vertices.

Textures already allow for a great degree of realism in incremental 3D graphics. Not only do they provide detail, but missing shading effects, shadows, indirect lighting may be painted or precomputed into textures. Clearly, these static substitutes do not respect dynamic scenes or changing lighting or viewing conditions.

The architecture described above requires the vertex positions and colors to be computed on the CPU, involving *transformations* and *local illumination lighting*. However, these are well-established procedures integrated into the pipeline of graphics libraries. Supporting them on the graphics card was a straightforward advancement.

The vertex records in the vertex buffer store the raw data. Vertex positions are given in modeling coordinates. The transformation to camera space and then from camera to screen space (or clipping space) are given as 4×4 homogeneous linear transformation matrices, the world-view, and the perspective one, respectively. They are of course *uniform* for all vertices, not stored in the vertex buffer, but in a few registers of the graphics card. Whenever these matrices change due to object or camera animation, the vertex buffer does not need to be altered.

Instead of including already computed color values in the buffer, the data required to evaluate the local shading formula are stored. This includes the surface normal and the diffuse and Phong-Blinn reflection coefficients. Light positions, directions and intensities are specified as uniform parameters over all the vertices.

For every vertex, the coordinates are transformed and shading is evaluated. Thus, the screen position and vertex color are obtained. Then the rasterization and linear interpolation are performed to color the pixels, just like without transformation and lighting.

When a final fragment color is computed, it is not directly written to the color buffer. First of all, as discussed above, the depth test against the depth buffer is performed to account for occlusions. However, some more computations are also supported in the hardware. A third buffer called the *stencil buffer* is also provided. For most of the time, stencil buffer bits are used as flags set when a pixel is rendered to. While

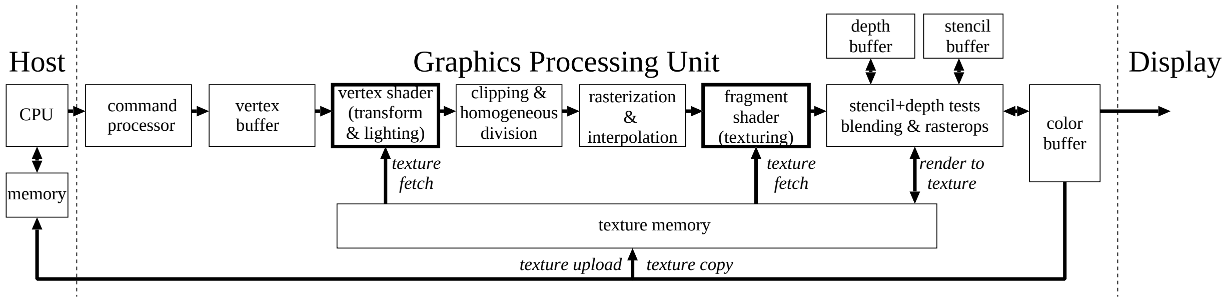


Figure 3: *Shader Model 3.0 GPU architecture*

drawing other objects, the stencil test may be enabled, discarding pixels previously not flagged. This way, reflections in a planar mirror, or shadows might be rendered. The functionality called *blending* allows for combining the computed fragment color with the color already written to the color buffer. This is the technique commonly used to achieve transparency. Colors are typically given as quadruplets of values, containing a so-called alpha channel besides the red, green and blue ones. The alpha value generally represents some opacity measure, and the alpha values already in the color buffer and that of the computed fragment color are used to weight the colors when combining them. Multiple blending formulae are usually supported.

With transformation and lighting modules, the hardware is able to render images using local illumination with per-vertex lighting. The computed color can be replaced or modulated using texture maps. Every improvement towards global illumination must make use of this architecture.

2.2. Architecture of programmable GPUs

Current, *Shader Model 3.0* — also called *DirectX 9 compatible* — GPUs implement the complete process of rendering triangle meshes. Figure 3 shows a typical GPU architecture. Figure 4 depicts the dataflow of such systems. Note that in this tutorial we do not cover hardware having *DirectX 10* features [Bly05].

The commands to the graphics API (e.g. DirectX or OpenGL) are passed to the *command processor*, which fills up the vertex buffer with modeling space vertices and their attributes, and also controls the operation of the whole pipeline. Whenever the data belonging to a vertex is ready, the *vertex shader* module starts working. It gets all attributes belonging to a vertex in its input registers. In the fixed-function pipeline, this module is responsible for transforming the vertex to homogeneous *clipping space* and may modify the color properties if lighting computations are also requested.

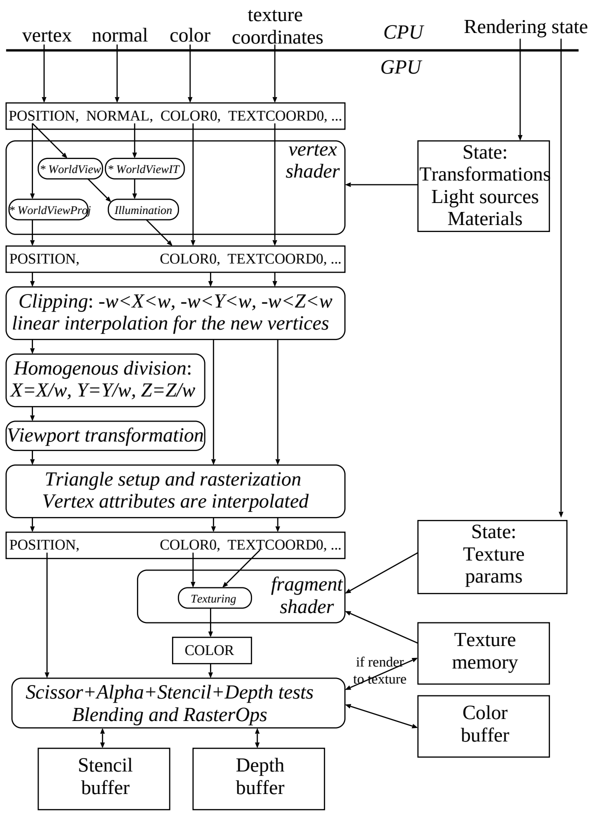

vertex
normal
color
texture coordinates
CPU
GPU
Rendering state
POSITION, NORMAL, COLOR0, TEXTCOORD0, ...
* WorldView
* WorldViewIT
* WorldViewProj
Illumination
vertex shader
State: Transformations Light sources Materials
POSITION, COLOR0, TEXTCOORD0, ...
Clipping: -w<X<w, -w<Y<w, -w<Z<w linear interpolation for the new vertices
Homogenous division: X=X/w, Y=Y/w, Z=Z/w
Viewport transformation
Triangle setup and rasterization Vertex attributes are interpolated
POSITION, COLOR0, TEXTCOORD0, ...
State: Texture params
Texturing
fragment shader
COLOR
Texture memory
if render to texture
Scissor+Alpha+Stencil+Depth tests Blending and RasterOps
Color buffer
Stencil buffer
Depth buffer

Figure 4: *Dataflow in a GPU assuming standard local illumination rendering*

Then the fixed pipeline waits for a complete triangle and the *clipping* hardware keeps only those parts where the $[x, y, z, w]$ homogeneous coordinates meet the following requirements defining an axis aligned, origin center cube of corners $(-1,-1,-1)$ and $(1,1,1)$ in normalized screen space:

$$-w \le x \le w, \ -w \le y \le w, \ -w \le z \le w.$$

These equations are valid in OpenGL. In DirectX, however, the normalized device space contains points of positive z coordinates, which consequently modifies the last pair of inequalities to $0 \leq z \leq w$.

Clipping may introduce new vertices for which all properties (e.g. texture coordinates or color) are linearly interpolated from the original vertex properties. After clipping the pipeline executes *homogeneous division*, that is, it converts homogeneous coordinates to Cartesian ones by dividing the first three homogeneous coordinates by the fourth (w). The points are then transformed to *viewport space* where the first two Cartesian coordinates select that pixel in which this point is visible. If triangle primitives are processed, the *rasterization* module waits for three vertices, forms a triangle from them, and fills its projection on the x, y plane, visiting each pixel that is inside the projection. During filling the hardware interpolates all vertex properties to obtain the attributes of a particular fragment. The *fragment shader* hardware takes the attributes of the particular fragment and computes the fragment color. This computation may involve texture lookups if texturing is enabled, and its multiplication with the color attribute in case of modulative texturing. The computed color may participate in raster operations, such as alpha blending, and its z coordinate goes to the z-buffer to detect visibility. If the fragment is visible, the result is written into the color buffer, or alternatively to the texture memory.

Programmable GPUs allow the modification of the fixed-function pipeline at two stages. We can customize the vertex shading and the fragment shading steps, using assembly language or high level shading languages, such as *HLSL*, *Cg*, etc.

2.2.1. Vertex shader

A vertex shader is connected to its input and output register sets defining the attributes of the vertex before and after the operation. All registers are of type `float4`, i.e. are capable of storing a four element vector. The input register set describes the vertex position (`POSITION`), colors (`COLOR0`, `COLOR1`), normal vector (`NORMAL`), texture coordinates (`TEXCOORD0`,..., `TEXCOORD8`), etc. The vertex shader unit computes the values of the output registers from the content of the input register. During this computation it may also use global, also called *uniform*, variables.

The following example shader realizes the vertex processing of the fixed-function pipeline when the lighting is disabled. It applies the *World*, *View*, and *Projection* transformations to transform the vertex from modeling to world, from world to camera, and from camera to homogeneous clipping space, respectively:

```
// homogenous linear transformation
// from modeling to homogeneous clipping space
float4x4 WorldViewProj;

void StandardNoLightingVS(
   in  float4 Pos    : POSITION, // modeling space
   in  float3 Color  : COLOR0,   // vertex color
   in  float2 Tex    : TEXCOORD0,// texture uv
   out float4 hPos   : POSITION, // clipping space
   out float3 oColor : COLOR0,   // vertex color
   out float2 oTex   : TEXCOORD0 // texture uv
) {
      // transform to clipping space
   hPos = mul(Pos, WorldViewProj);
   oColor = Color;  // copy input color
   oTex = Tex;      // copy texture coords
}
```

The second example executes local illumination computation for the vertices, and replaces the color attribute by the result. The illumination is evaluated in camera space where the eye is in the origin and looks at the $-z$ direction assuming OpenGL, and at the z direction in DirectX. In order to evaluate the Phong-Blinn illumination formula, normal, lighting, and viewing directions should be obtained in camera space. Note that if the shaded point is transformed to camera space by the `WorldView` matrix, the transformation of its associated normal vector should multiply with the inverse-transpose of the same matrix (`WorldViewIT`). We consider just a single point light source in the example.

```
// from modeling to homogeneous clipping space
float4x4 WorldViewProj;

// from modeling to camera space
float4x4 WorldView;

// Inverse-transpose of WorldView
// to transform normals
float4x4 WorldViewIT;

// Light source properties
float3 LightPos; // pos in camera space
float4 Iamb, Idiff, Ispec; // intensity

// Material properties
float4 ka, kd, ks; // reflectances
float shininess;

void StandardLightingVS(
   in  float4 Pos    : POSITION, // modeling space
   in  float3 Norm   : NORMAL,   // normal vector
   in  float2 Tex    : TEXCOORD0,// texture uv
   out float4 hPos   : POSITION, // clipping space
   out float3 oColor : COLOR0,   // vertex color
   in  float2 oTex   : TEXCOORD0 // texture uv
) {
   hPos = mul(Pos, WorldViewProj);
   // transform normal to camera space
   float3 N = mul(Norm, WorldViewIT).xyz;
   N = normalize(N);
   // transform vertex to camera space and
   // obtain the lighting direction
   float3 cPos = mul(Pos, WorldView);
   float3 L = normalize(LightPos - cPos);
   // evaluate the Phong-Blinn reflection
   float costheta = sat(dot(N, L));
   // Obtain view direction using that
   // the eye is the origin in camera space
   float3 V = normalize(-cPos); // viewing direction
   float3 H = normalize(L + V); // halfway vector
   float cosdelta = sat(dot(N, H));
   oColor = Iamb * ka + Idiff * kd * costheta +
            Ispec * ks * pow(cosdelta, shininess);
   oTex = Tex;      // copy texture coords
}
```

2.2.2. Fragment shader

The fragment shader (also called *pixel shader*) receives the fragment properties of those pixels which are inside of the clipped and projected triangles, and also uniform parameters. The main goal of the fragment shader is the computation of the fragment color.

The following example program executes modulative texturing. Taking the fragment input color and texture coordinates interpolated from vertex colors and texture coordinates, respectively, the fragment shader looks up the texture memory with the texture coordinates, and the read texture data is multiplied with the fragment input color:

```
sampler2D texture; // 2D texture sampler

float4 TexModPS( in float2 Tex   : TEXCOORD0,
                 in float3 Color : COLOR0
               ) : COLOR  // output
{
   return tex2D(texture, Tex) * Color;
}
```

2.3. Modification of the standard pipeline operation

Programmable vertex and fragment shaders offer a higher level of flexibility on how the data from the vertex buffer is processed, and how shading is performed. However, the basic pipeline model remains the same: a vertex is processed, the results are linearly interpolated, and they are used to find the color of a fragment. The flexibility of the programmable stages will allow us to change the shading model, implement per-fragment lighting, or render unfolded triangle charts instead of the models themselves, among the infinite number of other possibilities.

What programmable vertex and pixel shaders alone do not help us with is non-local illumination. All the data passed to shaders is still only describing local geometry and materials, or global constants, but nothing about other pieces of geometry. When a point is shaded with a global illumination algorithm, its radiance will be the function of all other points in the scene. From a programming point of view it means that we need to access the complete scene description when shading a point. While this is granted in CPU based ray tracing systems [WKB*02, WBS03], the stream processing architecture of current GPUs fundamentally contradicts to this requirement. When a point is shaded on the GPU we have just its limited amount of local properties stored in registers, and may access texture data. Thus the required global properties of the scene must be stored in *textures*.

If the textures must be static, or they must be computed on the CPU, then the lions share of illumination is not making use of the processing power of the parallel hardware, and the graphics card merely presents CPU results. Textures themselves have to be computed on the GPU. The *render-to-texture* feature allows this: anything that can be rendered to the screen, may be stored in a texture. Such texture render targets may also require depth and stencil buffers. Along with programmability, various kinds of data may be computed to textures. These data may also be stored in floating point format in the texture memory, unlike in the color buffer which usually stores data of 8 bit precision.

To use textures generated by the GPU, the rendering process must be decomposed to *passes*, where one pass may render into a texture and may use the textures generated by the previous passes. Since the reflected radiance also depends on geometric properties, these textures usually contain not only conventional color data, but they also encode geometry and prepared, reusable illumination information as well.

Straightforwardly, we may render the surroundings of a particular scene entity. Then, when drawing the entity, the fragment shader may be written so that it retrieves colors from this texture based on the reflected eye vector (computed from the local position and normal, plus the global eye position). This is the technique known as *environment mapping* (figure 5).

It is also possible to write a vertex shader which exchanges the texture and position coordinates. When drawing a model mesh, the result is that the triangles are rendered to their positions in texture space. Anything computed for the texels may later be mapped on the mesh by conventional texture mapping. This technique assumes that the mapping is unique, and such a render texture resource is usually called a *texture atlas* (figure 6).

When textures are used to achieve various effects, it

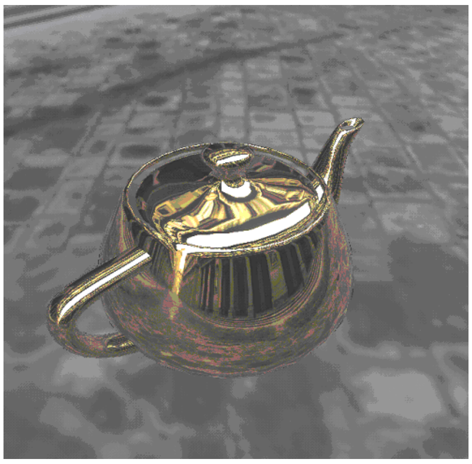


Figure 5: *Environment mapping using a metal shader*

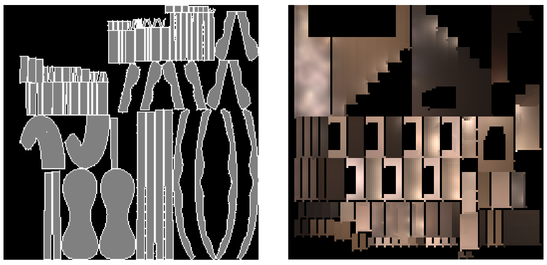


Figure 6: *A texture atlas of a rocking horse (left), and a texture atlas of a staircase storing radiance values. (right)*

becomes a necessity to be able to access multiple textures when computing a fragment color. This is called *multi-texturing*. This day, fragment shaders are able to access as many as 16 textures, any one of them multiple times. They all can be addressed using different modes or sets of texture coordinates, and their results can be combined freely in the fragment program. This allows the simultaneous use of different techniques like environment mapping, bump mapping, light mapping, shadow mapping, etc.

It is also a likely scenario that we need to compute multiple values for a rendering setup. This is accomplished using *multiple render targets*: a fragment shader may output several pixel colors or values, that will be written to corresponding pixels of respective render targets. With this feature, computing data that would not fit in a single texture is feasible. For instance, *deferred shading* [HH04] renders all visible geometry and material properties into screen-sized textures, and then uses these textures to render the shaded scene without actually rendering the geometry.

Programmability and render-to-texture together make it possible to create some kind of processed representation of geometry and illumination as textures, and then access the data when rendering and shading other parts of the scene. This is the key to addressing the self-dependency of the global illumination rendering problem. In all GPUGI algorithms, we use multiple passes to different render targets to capture some aspects of the scene like the surrounding environment, the shadowing or the refracting geometry, illumination due to light samples, etc. These passes belong to the *illumination information generation* part of rendering (figure 7). In a final pass, also called *final gathering*, scene objects are rendered to the frame buffer making use of previously computed information to achieve non-local shading effect like shadows, reflections, caustics, or indirect illumination.

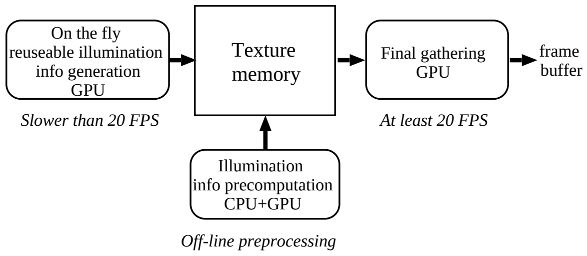


Figure 7: *Structure of real-time global illumination shaders*

Passes of the illumination information generation part are responsible for preparing the reusable illumination information and storing it in the texture memory, from which the final gathering part produces the image for the particular camera. To produce continuous animation, the final gathering part should run at high frame rates. Since the scene may change, the illumination information generation should also be repeated. However, if the illumination information is appropriately defined, then its elements can be reused for many points and many frames. Thus the illumination information data structure is compact and might be regenerated at significantly lower frequency than the final gathering frame rate.

As we shall discuss in these tutorial, these features give us enough freedom to implement global illumination algorithms and even ray-tracing approaches. However, we must be aware that it is not worth going very far from the original concepts of the pipeline, namely rasterization and texturing, because it might have serious performance penalties. This is why GPU implementation often means the invention of brand new approaches and not just adapting or porting existing ones.

3. Simple improvements of the local illumination lighting model

Local illumination models simplify the rendering problem to shading a surface fragment according to a given point-like light source. This is based on several false assumptions, which results in less realistic images:

The light source is always visible. In reality, incoming lighting depends on the materials and geometry found between the light source and the shaded point. Most prominently, solid objects may occlude the light. Neglecting this effect, we will render images without shadows. In order to eliminate this shortcoming, we may capture occluding geometry in texture maps, and test the light source for visibility when shading. This technique is called *shadow mapping*.

The light illuminates from a single direction. In reality, light emitters occupy some volume. While the point-like or directional model is suitable for small artificial lights or the sun, in most environments we encounter extended light sources. Most prominently, the sky itself is a huge light source. In *image based lighting*, we place virtual objects in a computed or captured environment, which is also a lighting problem where the environment image is an extended light source. Volumetric or area lights generate more elaborate shadows, as they might be only partly visible from a given point. These shadows are often called *soft shadows*, as they do not feature a sharp boundary between shadowed and lighted surfaces. Generally, point-sampling of extended light sources is required to render accurate shadows. However, with some simplifying assumptions for the light source, faster approximate methods may be obtained, generating perceptionally plausible soft shadows.

No indirect lighting. In reality, all illuminated objects reflect light, lighting other objects. While this indirect lighting effect constitutes a huge fraction of light we perceive, it tends to be low-frequency and less obvious, as most surfaces scatter the light diffusely. However, for highly specular, metallic, mirror-like or refractive materials, this does not apply. Indirect illumination may exhibit elaborate high frequency patterns called caustics, and of course the color we see on a mirror's surface depends on the surrounding geometry. These issues require more sophisticated methods, based on the approach of *environment mapping*, capturing incoming environment radiance in textures.

3.1. Shadow mapping

Shadows are important not only to make the image realistic, but also to allow humans to perceive depth and distances. Shadows occur when an object called *shadow caster* occludes the light source from another object, called *shadow receiver*, thus prevents the light source from illuminating the shadow receiver. In real time applications shadow casters and receivers are often distinguished, which excludes self shadowing effects. However, in real life all objects may act as both shadow receiver and shadow caster.

Point and directional light sources generate *hard shadows* having well defined boundaries of illumination discontinuities. However, realistic light sources have non zero area, resulting in *soft shadows* having continuous transition between the fully illuminated region and the occluded region, called *umbra*. The transition is called the *penumbra* region. With hard shadows only the depth order can be perceived, but not the distance relations. Shadows should have real penumbra regions with physically accurate size and density to allow the observer to reconstruct the 3D scene.

The width and the density of the penumbra regions depend on the size of the area light source, on the distance between the light source and shadow caster object, and on the distance between the shadow caster and shadow receiver object.

Real-time shadow algorithms can be roughly categorized as image space *shadow map* or object space *shadow volume* techniques. Shadow volumes construct invisible faces to find out which points are in the shadow and require geometric processing. Exact geometric representation allows exact shadow boundaries, but the computation time grows with the geometric complexity, and partially transparent objects, such as billboards become problematic. Furthermore, these methods cannot cope with geometries modified during rendering, as happens when displacement mapping is applied.

Shadow maps, on the other hand, work with a depth image, which is a sampled version of the shadow casters. Since shadow map methods use only a captured image of the scene as seen from the light, they are independent of the geometric complexity, can conveniently handle displacement mapped and transparent surfaces as well. However, their major drawback is that the shadowing information is in a discretized form, as a collection of shadow map pixels called *lexels*, thus sampling or aliasing artifacts are likely to occur.

For the sake of simplicity, we assume that the light sources are either *directional* or *spot* lights having a main illumination direction. *Omnidirectional* lights are not considered. Note that this is not a limitation since an omnidirectional light can be replaced by 6 spot lights radiating towards the six sides of a cube placed around the omnidirectional light source.

Shadow map algorithms use several coordinate systems which are briefly reviewed here:

World space: This is the arbitrary global frame of reference for specifying positions and orientations of virtual world objects, light sources and cameras. Object models are specified using modeling coordinates. For an actual instance of a model, there is a transformation that moves object points from modeling space to world space. This is typically a homogeneous linear transformation, called the modeling transformation or `World`.

Eye's camera space: In this space the eye position of the camera is in the origin, the viewing direction is the z axis in DirectX and the $-z$ direction in OpenGL, and the vertical direction of the camera is the y axis. Distances and angles are not distorted, lighting computations can be carried out identically to the world space. The transformation from modeling space to this space is `WorldView`.

Light's camera space: In this space the light is in the origin, and the main light direction is the z axis. This space is similar to the eye's camera space having replaced the roles of the light and the eye. The transformation from modeling space to this space is `DepthWorldView`, which must be set according to the light's position.

Eye's normalized device space: Here the eye is at an ideal point $[0, 0, 1, 0]$, thus the viewing rays get parallel. The visible part of the space is an axis aligned box of corners $[-1, -1, 0]$ and $[1, 1, 1]$ in Cartesian coordinates. The transformation to this space is not an affine transformation, thus the fourth homogeneous coordinate is usually not equal to 1. The transformation from modeling space to this space is `WorldViewProj`.

Light's normalized device space: Here the light is at an ideal point $[0, 0, 1, 0]$, thus the lighting rays get parallel. The illuminated part of the space is an axis aligned box of corners $[-1, -1, 0]$ and $[1, 1, 1]$ in Cartesian coordinates. The transformation to this space is not an affine transformation, thus the fourth homogeneous coordinate is usually not equal to 1. The transformation from modeling space to this space is `DepthWorldViewProj`. This matrix should be set according to light characteristics. For a directional light, the light rays are parallel without any non-affine transformation, so we only need an orthographic projection to scale the interesting, illuminated objects into the unit box. For point lights, a perspective projection matrix is needed, identical to that of perspective cameras. The field of view should be large enough to accommodate for the spread of the spotlight. Omnidirectional lights need to be substituted by six 90° FOV angle lights.

Shadow mapping has two stages, shadow map generation when the camera is placed at the light, and image generation when the camera is at the eye position.

3.1.1. Shadow map generation

Shadow map generation is a regular rendering pass where the z-buffer should be enabled. The actual output is the z-buffer texture with the depth values. Although a color target buffer is generally required, color writes should be disabled to increase performance.

Transformation `DepthWorldViewProj` is set to transform points to the world space, then to the light-camera space, and finally to the light's normalized device space. The shader executes a regular rendering phase. Note that the pixel shader color is meaningless since it is ignored by the hardware anyway.

```
//model to depth map's screen space
float4x4 DepthWorldViewProj;

void DepthVS(in  Pos  : POSITION,
             out hPos : POSITION) {
    hPos = mul(Pos, DepthWorldViewProj);
}

float4 DepthPS( ) : COLOR0 {
    return 0;
}
```

3.1.2. Rendering with the shadow map

In the second phase, the scene is rendered from the eye camera. Each visible point is transformed to the light space, then to texture space, and its depth value is compared to the stored depth value. The texture space transformation is responsible for mapping spatial coordinate range $[-1, 1]$ to $[0, 1]$ texture range, inverting the y coordinate, since the spatial y coordinates increase from bottom to top, while the texture coordinates increase from top to bottom. Furthermore, the transformation shifts the u, v texture address by half a texel, and possibly also adds a small bias to the z coordinate to avoid self-shadowing.

The necessary transformation steps convert point `p` in light's normalized device space to projective texture space `t`:

```
         // center is 0.5 and add half texel
offset = 0.5 + 0.5 / SHADOWMAP_SIZE;
t.x =  0.5*p.x + offset; //[-1,1]->[0,1]+halftex
t.y = -0.5*p.y + offset; //[1,-1]->[0,1]+halftex
t.z = p.z - bias;
t.w = p.w;
```

Here `SHADOWMAP_SIZE` denotes the resolution of the shadow map and `bias` the z bias.

It is often a delicate issue to choose an appropriate bias for a given scene. Values exceeding the dimensions of geometric details will cause *light leaks*, non-shadowed surfaces closely behind shadow casters. With a bias not large enough, z-fighting will cause interference-like shadow stripes on lighted surfaces. Both atrifacts are extremely disturbing and unrealistic. The issue might be more severe when the depth map is rendered with a large FOV angle, as depth distortion can be extreme. A convenient solution is the *second depth value* technique. Assuming all our shadow casters are non-intersecting, opaque manifold objects, the backfaces cast the same shadow as the object itself. By reversing the backface culling mechanism, we can render depth values into the depth map that do not coincide with any front face depth. The bias in the above code can be set to zero.

The texture space transformation can also be implemented as a matrix multiplication. The following T_T, or `TexScaleBias` matrix must be appended to the transformation to light's normalized device space:

$$\begin{bmatrix} 0.5 & 0 & 0 & 0 \\ 0 & -0.5 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \texttt{offset} & \texttt{offset} & -\texttt{bias} & 1 \end{bmatrix},$$

producing `DepthWorldViewProjTex`. The vertex shader executes this transformation:

```
// To the normalized screen space of the eye camera
float4x4 WorldViewProj;

// To the texture space of the depth map
float4x4 DepthWorldViewProjTex;

void ShadowVS(
  in  float4 Pos   : POSITION, // model space
  in  float4 Color : COLOR0,   // input color
  out float4 hPos  : POSITION, // clip space
  out float4 depthPos : TEXCOORD0, // depth tex
  out float4 oColor : COLOR0)  // output color
{
  oColor = Color; // copy color

  // transform model-space vertex position
  // to light's (depth map's) texture space
  depthPos = mul(Pos, DepthWorldViewProjTex);

  // transform model-space vertex position
  // to eye's normalized device space:
  hPos = mul(Pos, WorldViewProj);
}
```

In the pixel shader we check if the stored depth value is smaller than the given point's depth. That is, the point is in shadow:

```
sampler2D  ShadowMap;   // depth map in texture

float4 ShadowPS(
  float4 depthPos : TEXCOORD0,// depth tex
  float4 Color    : COLOR0    // input color
) : COLOR
{
  // returns 0 or 1 as a comparison result
  //   [if shadowMapSampler uses linear
  //   interpolation, these 0/1 values are
  //   interpolated, not depth])
  float vis = tex2Dproj(ShadowMap, depthPos).r;
  return vis * Color;
}
```

The code for the shadow map query is virtually identical to the code for projective textures. Projective texturing `tex2Dproj(sampler, p)` divides `p.x`, `p.y`, `p.z` by `p.w` and looks up the texel addressed by `(p.x/p.w, p.y/p.w)`. Shadow maps are like other projective textures, except that in projective texture lookups instead of returning a texture color, `tex2Dproj` returns the boolean result of the comparison of `p.z/p.w` and the value stored in the texel. To force the hardware to do this, the associated texture unit should be configured by the application for depth compare texturing; otherwise, no depth comparison is actually performed. In DirectX, this is done by creating the texture resource with the usage flag `D3DUSAGE_DEPTHSTENCIL`. Note that `tex2D` will not work on this texture, only `tex2Dproj` will.

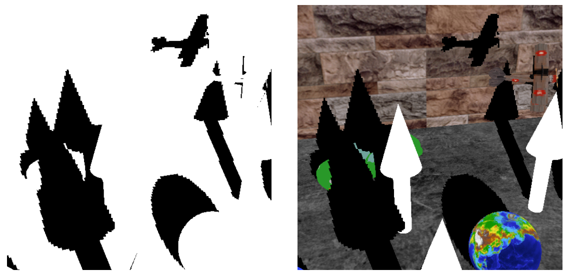


Figure 8: *Hardware shadow mapping with* 512 × 512 *shadow map resolution at 210 FPS.*

Classical shadow mapping requires just a single texture lookup in the pixel shader. This naive implementation has many well known problems, caused by storing only sampled information in the depth map. These problems include shadow acnes and aliasing.

3.2. Image based lighting

In many computer graphics applications it is desirable to augment the virtual objects with high dynamic range images representing a real environment (sky, city, wood, etc.). In order to provide the illusion that the virtual objects are parts of the real scene, the illumination of the environment should be taken into account when the virtual objects are rendered.

The very same approach can also be used for purely virtual scenes if they can be decomposed to smaller dynamic objects and to a larger static or slowly changing part (such distinction is typical in games and virtual reality systems). In this case, the illumination reflected off the static part of the scene is computed from a *reference point* placed in the vicinity of the dynamic objects, and is stored in images. Then the static part of the scene is replaced by these images when dynamic objects are rendered.

In both cases, the illumination of virtual objects is defined by these images, also called *environment maps.* The illumination computation process is called *environment mapping* [BN76].

The radiance values representing environment illumination may differ by orders of magnitude, thus they cannot be mapped to the usual $[0, 255]$ range. Instead, the red, green, and blue colors of the pixels in these images should be stored as floating point values to cope with the high range. Floating point images are called *high dynamic range images.*

Environment mapping assumes that the illumination stored in images comes from very (infinitely) far surfaces. It means that a ray hitting the environment becomes independent of the ray origin. In this case rays can be translated to the same *reference point*, and environment maps can be queried using only the direction of the ray.

Environment mapping has been originally proposed to render ideal mirrors in local illumination frameworks, then extended to approximate general secondary rays without expensive ray-tracing [Gre84, RTJ94, Wil01]. Environment mapping has also become a standard technique of *image based lighting* [MH84, Deb98].

In order to compute the image of a virtual object under infinitely far environment illumination, we should evaluate the *reflected radiance* L^r due to the environment illumination at every visible point $\vec{x}$ at view direction $\vec{\omega}$ (figure 9):

$$L^r(\vec{x}, \vec{\omega}) = \int_{\Omega'} L^{env}(\vec{\omega}') \cdot f_r(\vec{\omega}', \vec{x}, \vec{\omega}) \cdot \cos^+ \theta'_{\vec{x}} \cdot v(\vec{x}, \vec{\omega}')\, d\omega', \tag{3}$$

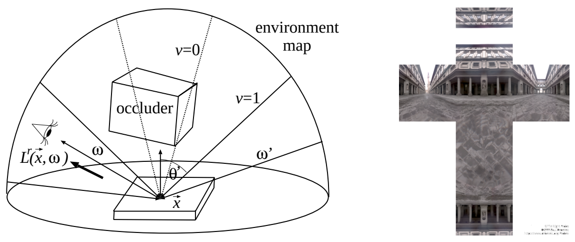


Figure 9: *The concept of environment mapping and an environment map stored in a cube map.*

where $L^{env}(\vec{\omega}')$ is the radiance of the environment map at direction $\vec{\omega}'$, and $v(\vec{x}, \vec{\omega}')$ is the *visibility factor* checking whether no virtual object is seen from $\vec{x}$ at direction $\vec{\omega}'$ (that is, the environment map can illuminate this point from the given direction). Note that the assumption that illumination arrives from very distant sources allowed the elimination of the positional dependence from the incoming radiance and its replacement by direction dependent environment radiance $L^{env}(\vec{\omega}')$.

The illumination of the environment map on the virtual objects can be obtained by tracing rays from the points of the virtual object in the directions of the environment map, and checking whether or not occlusions occur [Deb98, KK03]. The computation of the visibility factor, that is the shadowing of objects, is rather time consuming. Thus most of the environment mapping algorithms simply ignore this factor and take into account the environment illumination everywhere and in all possible illumination directions.

A natural way of storing the direction dependent environment map $L^{env}(\vec{\omega}')$ as an angular mapped floating point texture. Direction $\vec{\omega}'$ is expressed by spherical angles θ', ϕ' where $\phi \in [0, 2\pi]$ and $\theta' \in [0, \pi/2]$ in case hemispherical lighting and $\theta' \in [0, \pi]$ in case of spherical lighting. Then texture coordinates $[u, v]$ are scaled from the unit interval to these ranges. For example, in case of spherical lighting

$$\vec{\omega}' = (\cos 2\pi u \cdot \sin \pi v,\ \sin 2\pi u \cdot \sin \pi v,\ \cos \pi v),$$

where $u, v \in [0, 1]$.

Another, more GPU friendly possibility is to parameterize the directional space as sides of a cube centered at the origin and having edge size 2. A point x, y, z on the cube corresponds to direction

$$\vec{\omega}' = \frac{(x, y, z)}{\sqrt{x^2 + y^2 + z^2}}.$$

One of the three coordinates is either 1 or −1.

For example, the directions corresponding to the right ($z = 1$) face of the cube are

$$\vec{\omega}' = \frac{(x, y, 1)}{\sqrt{x^2 + y^2 + 1}}, \qquad x, y \in [-1, 1].$$

Current GPUs have built in support to compute this formula and to obtain the stored value from one of the six textures of the six *cube map* faces (`texCUBE` in HLSL).

3.2.1. Mirrored reflections and refractions

Let us assume that there are no self occlusions, so $v(\vec{x}, \vec{\omega}') = 1$. If the surface is an ideal mirror, then its BRDF allows the reflection just from a single direction, thus the rendering equation simplifies to:

$$L^r(\vec{x}, \vec{\omega}) = \int_{\Omega'} L^{env}(\vec{\omega}') \cdot f_r(\vec{\omega}', \vec{x}, \vec{\omega}) \cdot \cos^+ \theta'_{\vec{x}} \cdot v(\vec{x}, \vec{\omega}')\, d\omega'$$

$$= L^{env}(\vec{R}) \cdot F(\vec{N}, \vec{R}),$$

where $\vec{N}$ is the unit surface normal at $\vec{x}$, $\vec{R}$ is the unit reflection direction of viewing direction $\vec{\omega}$ onto the surface normal, and F is the *Fresnel function*. We can apply an approximation of the Fresnel function, which is similar to Schlick's approximation [Sch93] in terms of computational cost, but can take into account not only *refraction index n* but also *extinction coefficient k*, which is essential for realistic metals [LSK05]:

$$F(\vec{N}, \vec{R}) = F_\perp + (1 - F_\perp) \cdot (1 - \vec{N} \cdot \vec{R})^5,$$

where

$$F_\perp = \frac{(n-1)^2 + k^2}{(n+1)^2 + k^2} \qquad (4)$$

is the Fresnel function (i.e. the probability that the photon is reflected) at perpendicular illumination. Note that $F_\perp$ is constant for a given material, thus this value can be computed on the CPU from the refraction index and extinction coefficient and passed to the GPU as a global variable.

Environment mapping approaches can be used to simulate not only reflected but also refracted rays, just the direction computation should be changed from the law of reflection to the Snellius-Descartes law of refraction, that is, the `reflect` operation should be replaced by the `refract` operation in the pixel shader. The intrinsic function `refract` will return a zero vector when total reflection should occur. We should note that tracing a refraction ray on a single level is a simplification since the light is refracted at least twice to go through a refractor. Here we discuss only this simplified case (a method addressing multiple refractions is presented in [SKALP05]).

The amount of refracted light can be computed using weighting factor $1 - F$ where F is the Fresnel function. However, this is true just at the point of refraction. While the light traverses inside the object, its intensity decreases exponentially according to the extinction coefficient. For metals, where the extinction coefficient is not negligible, the refracted component is completely eliminated (metals can never be transparent). For dielectric materials, on the other hand, we usually assume that the extinction coefficient is zero, thus that the light intensity remains constant inside the object. The following shader uses this assumption and computes both the reflected and refracted illumination of an infinitely distant environment map:

```
float    Fp; // Fresnel at perpendicular dir.
float    n;  // index of refraction

void EnvMapVS(
   in  float4 Pos  : POSITION, // modeling space
   in  float3 Norm : NORMAL,   // modeling space
   out float4 hPos : POSITION, // clipping space
   out float3 cNorm : TEXCOORD0,// camera space
   out float3 cView : TEXCOORD1 // camera space
) {
   hPos = mul(Pos, WorldViewProj);
   cNorm = mul(Norm, WorldViewIT);
   cView = -mul(Pos, WorldView);
}

samplerCUBE EnvMap; // environment map

float4 EnvMapPS(
   float3 Norm : TEXCOORD0, // camera space
   float3 View : TEXCOORD1  // camera space
) : COLOR {
   float3 Norm = normalize( IN.Norm );
   float3 View = normalize( IN.View );

   float3 R = reflect(View, Norm);
   float3 T = refract(View, Norm, 1/n);

      // sampling from the cube map
   float4 refl = texCUBE(EnvMap, R);
   float4 refr = texCUBE(EnvMap, T);
      // approximation of the Fresnel Function
   float cos_theta = -dot(View ,Norm);
   float F = Fp + pow(1-cos_theta, 5.0f) * (1-Fp);
   return F * refl + (1-F) * refr;
}
```

3.2.2. Diffuse and glossy reflections without self-shadowing

Classical environment mapping can also be applied for both glossy and diffuse reflections. If we ignore self occlusions ($v(\vec{x}, \vec{\omega}') = 1$), the usual trick is the convolution of the angular variation of the BRDF with the

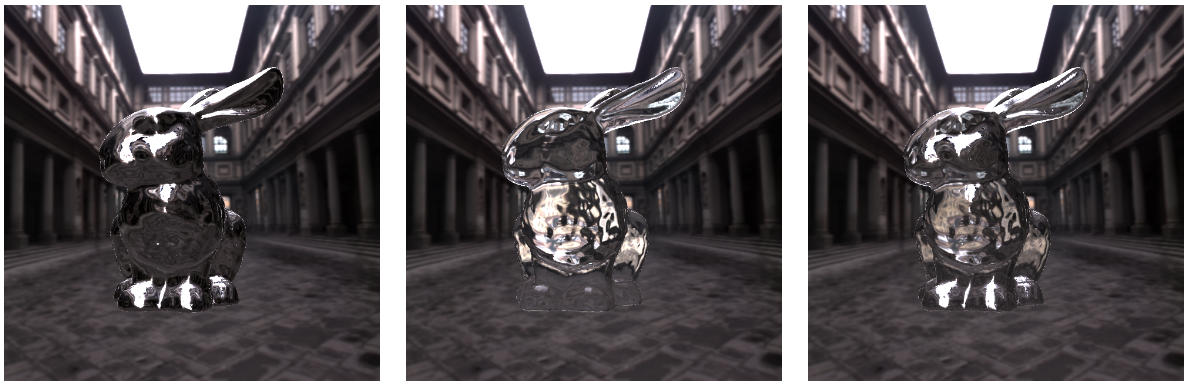


Figure 10: *Environment mapped reflection (left), refraction (middle), and combined reflection and refrction (right).*

environment map during preprocessing [RH01]. The integral of equation 3 is evaluated using a finite number of samples. For example, approximating the rendering equation with N samples requires the evaluation of the following numerical quadrature (i.e. a sum approximating the integral):

$$L^r(\vec{x},\vec{\omega}) \approx \sum_{i=1}^{N} L^{env}(\vec{\omega}_i') \cdot f_r(\vec{\omega}_i',\vec{x},\vec{\omega}) \cdot \cos^+\theta_i' \;\Delta\omega_i'$$

In case of diffuse objects the BRDF is constant, thus the reflected radiance becomes:

$$L^r(\vec{x},\vec{\omega}) \approx f_r \cdot \sum_{i=1}^{N} L^{env}(\vec{\omega}_i') \cdot \cos^+\theta_i' \;\Delta\omega_i'$$

Note that the only factor in this sum that depends on shaded point $\vec{x}$ is $\cos^+\theta_i' = (\vec{N}_{\vec{x}} \cdot \vec{\omega}_i)^+$, which is the cosine of the angle between the sample direction and the surface normal of the shaded point. These sums are pre-computed for a sufficient number of normal vectors and store irradiance values

$$I_j = \sum_{i=1}^{N} L^{env}(\vec{\omega}_i') \cdot (\vec{N}_j \cdot \vec{\omega}_i)^+ \;\Delta\omega_i'.$$

In order to compute the diffuse reflectance we need to determine the normal at the shaded point, look up the corresponding irradiance value, and modulate it with the diffuse BRDF. Irradiance values are stored in an environment map called *diffuse environment map* or *irradiance environment map*. Assuming that each value is stored in the direction of the corresponding normal vector, we can determine the illumination of an arbitrarily oriented surface patch during rendering with a single environment map lookup toward the normal direction of the surface. In other words, the *query direction* for the environment map lookup will be the surface normal. Assuming cubic environment maps, the pixel shader implementation is as follows:

```
float4 PS_Diffuse( float3 N : NORMAL ) : COLOR0 {
    return texCUBE(EnvMap, N);
}
```

In case of glossy objects, assuming Phong illumination model, the illumination depends on both the normal direction and the view direction, but this dependence can be described with a single vector that is obtained by mirroring the view direction on the surface normal. This *reflection direction* will serve as query direction in case of glossy objects:

```
float4 PS_Glossy( float3 N : NORMAL,
                  float3 V : TEXCOORD1 ) : COLOR0
{
    V = normalize( V );
    N = normalize( N );
    // compute the reflection direction
    float3 R = reflect(V, N);
    return texCUBE(EnvMap, R);   // query
}
```

The precalculation of the irradiance environment map is rather computation intensive for both diffuse and glossy objects. In order to obtain a single irradiance value, we have to consider all possible incoming directions and sum up their cosine-weighted incoming intensities. This *convolution* must be performed for each texel of the irradiance environment map. The computation process can be sped up using spherical harmonics [Kin05].

If we perform the convolution only once, at startup, the frame rate is above 600 FPS.

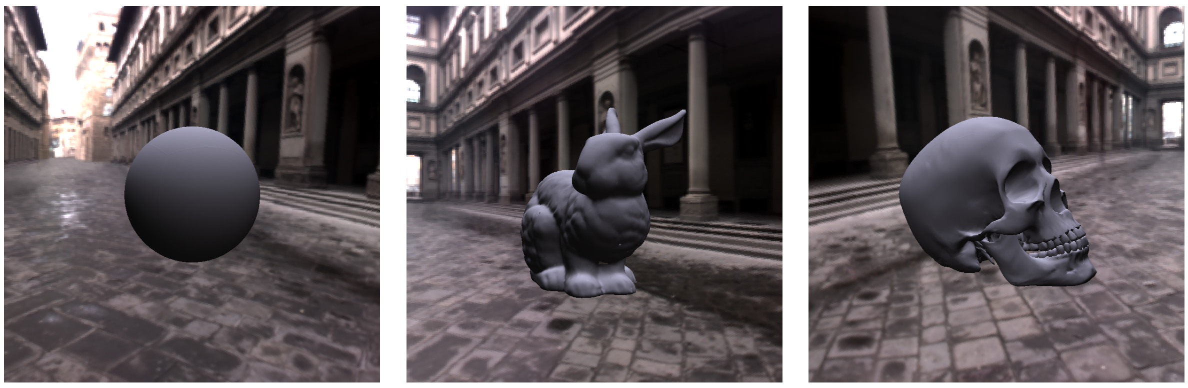


Figure 13: *Diffuse objects.*

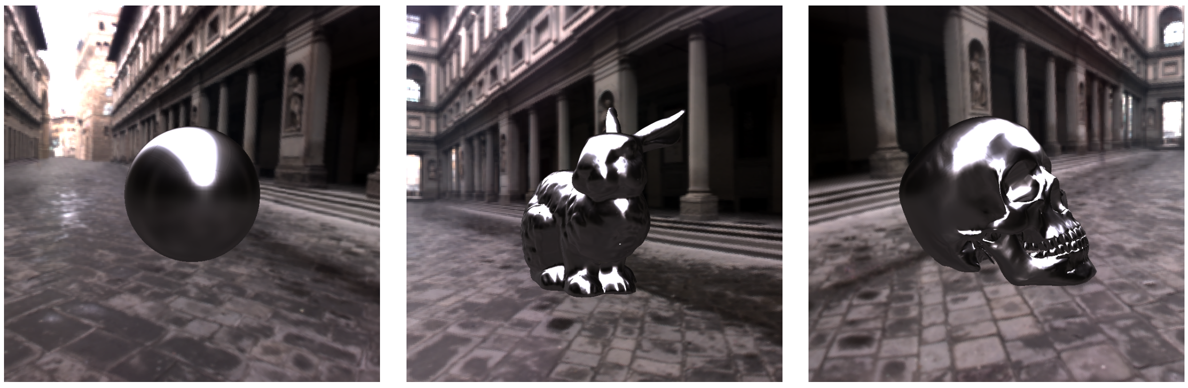


Figure 14: *Specular objects, shininess = 60*

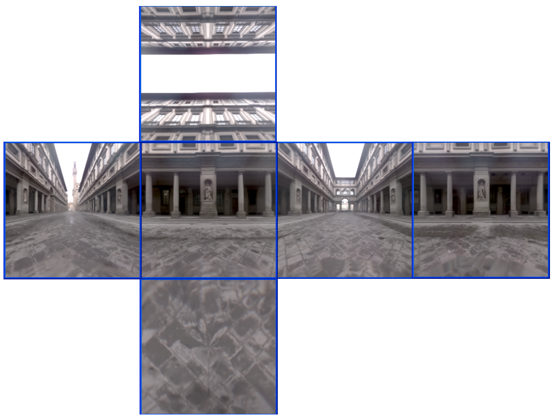


Figure 11: *Original high dynamic range environment map (uffizi_cross.dds).*

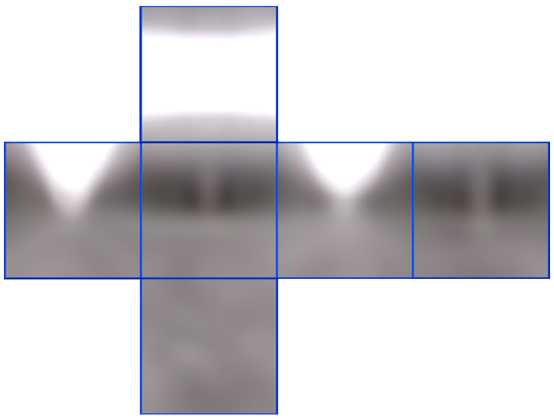


Figure 12: *Diffuse irradiance map obtained by convolution.*

3.2.3. Diffuse and glossy reflections with shadowing

Classical image based lighting algorithms ignore occlusions and assume that the environment is visible from everywhere. If shadows are also needed, occlusions should also be taken into account, thus shadow mapping should be combined with image based lighting.

The key idea of the solution is to decompose the environment map to a finite number of directional domains. In order to estimate the integral of equation 3, directional domain Ω' is decomposed to solid angles $\Delta\omega'_i, i = 1, \ldots, N$ meeting the following criteria:

- the radiance is roughly uniform within each domain,
- the solid angles are small and light flux arriving from every domain has the same magnitude, thus it is enough to test the visibility of the environment map with a single sample in each solid angle:

$$L^r(\vec{x}, \vec{\omega}) =$$

$$\sum_{i=1}^{N} \int\limits_{\Delta\omega'_i} L^{env}(\vec{\omega}') \cdot f_r(\vec{\omega}', \vec{x}, \vec{\omega}) \cdot \cos^+ \theta'_{\vec{x}} \cdot v(\vec{x}, \vec{\omega}')\, d\omega' \approx$$

$$v(\vec{x}, \vec{\omega}'_i) \cdot \tilde{\Psi}_i^{env} \cdot a(\Delta\omega'_i, \vec{\omega}),$$

where

$$\tilde{\Psi}_i^{env} = \int\limits_{\Delta\omega'_i} L^{env}(\vec{\omega}')\, d\omega'$$

is the *total incoming power* from solid angle $\Delta\omega'_i$, and

$$a(\Delta\omega'_i, \vec{\omega}) = \frac{1}{\Delta\omega'_i} \cdot \int\limits_{\Delta\omega'_i} f_r(\vec{\omega}', \vec{x}, \vec{\omega}) \cdot \cos^+ \theta'_{\vec{x}}\, d\omega'.$$

is the *average reflectivity* from solid angle $\Delta\omega'_i$ to viewing direction $\vec{\omega}$. In order to use this approximation, the following tasks need to be solved:

1. The directional domain should be decomposed meeting the prescribed requirements, and the total radiance should be obtained in them. Since these computations are independent of the objects and of the viewing direction, we can execute them in the preprocessing phase.
2. The visibility of the environment map in the directions of the centers of the solid angles needs to be determined. Since objects are moving, this calculation is done on the fly. Note that this step is equivalent to shadow computation assuming directional light sources.
3. The average reflectivity values need to be computed and multiplied with the total radiance and the visibility for each solid angle. Since the average reflectivity also depends on the normal vector and viewing direction, we should execute this step as well on the fly.

For a static environment map, the generation of sample directions should be performed at loading time, making it a non time-critical task. The goal is to make the integral quadrature accurate while keeping the number of samples low. This requirement is met if solid angles $\Delta\omega'_i$ are selected in a way that the environment map radiance is roughly homogeneous in them, and their area is inversely proportional to this radiance. The task is completed by generating random samples with a probability proportional to the power of environment map texels, and applying *Lloyd's relaxation* [Llo82] to spread the samples more evenly. A weighted version of the relaxation method is used to preserve the density distribution. As the basic idea of Lloyd's relaxation is to move the sample points to the center of their respective Voronoi areas, the relatively expensive computation of the Voronoi mesh is necessary.

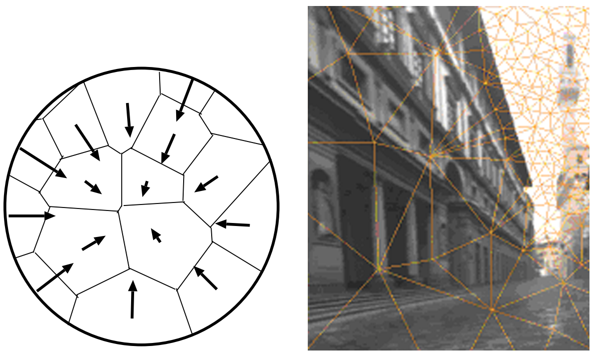


Figure 15: *Radiance values of texels within the Voronoi areas are summed to compute their total power and the final Delaunay grid on high dynamic range image. The centers of Voronoi cells are the sample points.*

A Voronoi cell is defined as the set of points to which a given sample is the closest one. The dual of the Voronoi decomposition is the Delaunay mesh of triangles, with the defining property that no circumcircle of any triangle contains any of the sample points. On the unit sphere of directions, this means that no sample can lie on the outer side of a triangle's plane. Inserting a new sample point goes according to the Bowyer-Watson algorithm:

1. you start with a Delaunay mesh that is a convex

polyhedron which only has triangle faces, and all nodes on the unit sphere

2. delete all triangles for which the new sample direction lies on the outer side of the triangle's plane (which the outer side is is defined by the order of the triangles nodes, but for usual meshes the inner side is where the origin is)
3. connect the new sample to the edges of the cavity just created (it will always be a convex cavity in the sense that the resulting mesh will be a legal convex Delaunay polyhedron)

Alternatively, extremely fast algorithms using *Penrose tiling* [ODJ04] and *Wavelet importance sampling* have been published to attack the well-distributed importance sampling problem. These methods should be considered if non-static environment maps are to be used. However, if we wish to assign the most accurate light power values to the sampled directions, we have to add up the contributions of those texels that fall into the Voronoi region of the direction. In this case, Lloyd's relaxation means no significant overhead, as rigorously summing the texel contributions takes more time.

Visibility determination

We have decomposed the environment map to solid angles and we want to test visibility with a single sample in each subdomain. The problem is traced back to rendering shadow caused by directional light sources. In order to render shadows effectively, a hardware-supported shadow technique has to be applied. *Depth map shadows* [Wil78] generated for every discrete direction are well suited for the purpose (figure 16).

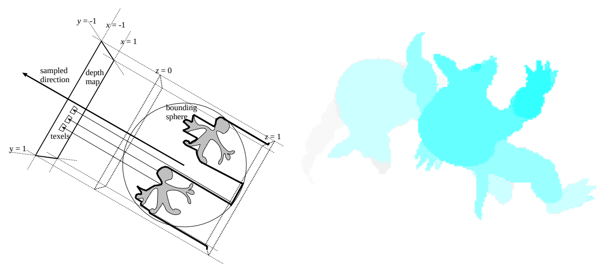


Figure 16: *The concept of depth mapped shadows for the directional lights corresponding to Voronoi cells and a particular depth image.*

Lighting using the samples

When average reflectivity $a(\Delta\omega'_i, \vec{\omega})$ from solid angle $\Delta\omega'_i$ to viewing direction $\vec{\omega}$ is computed, we have to accept simplifications to make the method real time. We consider a standard diffuse + specular BRDF, where the specular part is defined by the Phong model:

$$f_r(\vec{\omega}', \vec{x}, \vec{\omega}) \cdot \cos\theta' = k_d \cdot \cos\theta' + k_s \cdot \cos^n \psi,$$

where k_d is the diffuse reflection parameter, k_s is the specular reflectivity, n is the shininess of the surface, and ψ is the angle between the ideal reflection direction of view vector $\vec{\omega}$ and the illumination direction $\vec{\omega}'$. We assume that the integrand has low variation, which is true for not highly specular materials and small directional domains (meaning we have enough directional samples). In this case the integral is estimated from a single sample associated with the center of the Voronoi region:

$$a_d(\Delta\omega'_i, \vec{\omega}) \approx k_d \cdot \cos\theta_c$$

where θ_c is the angle between the surface normal and direction $\vec{\omega}'_i$ corresponding to the center of this Voronoi region. This means lighting happens just like in the case of any ordinary directional light source.

However, if the surface is more glossy or the Voronoi cells are large, then the approach results in artifacts. The reflection will be noisy even at points where there is no self occlusion. The problem is that both the albedo and the shadowing factor are estimated with the same, low number of discrete samples.

In such cases, the two computations should be separated. We need better approximations for the albedo integral, but use the low number of discrete samples for shadow estimation [BSKS05].

Implementation

The demo application ImageBasedLighting realizes the above algorithm. Contributions of directional light samples are rendered in batches.

1. Depth is laid down using technique `renderBlack`. Using the z-test, in further rendering passes to the frame buffer, only visible pixels will be evaluated.
2. For every batch of light samples:
   a. A set of depths maps are rendered for all light samples in the batch using technique `renderDepth`.
   b. Using blending, the contribution of all light samples in the batch is added to the frame buffer (technique `renderFinal`).
3. The environment map is rendered behind the scene using technique `renderBackground`.

Rendering times depend heavily on the number of samples used. Real-time results (30 FPS) are possible for moderately complex scenes with a few objects (Figure 18). However, difficult scenes (Figure 19) require more samples for acceptable quality, and therefore only run at interactive frame rates (5 FPS).

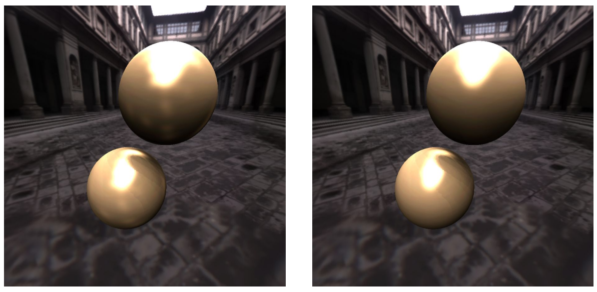


Figure 17: *Spheres in Florence rendered with taking a single sample in each Voronoi cell, assuming directional light sources (left) and using a better albedo approximation for the average reflectance over Voronoi cells (right). The shininess of the spheres is* 100.

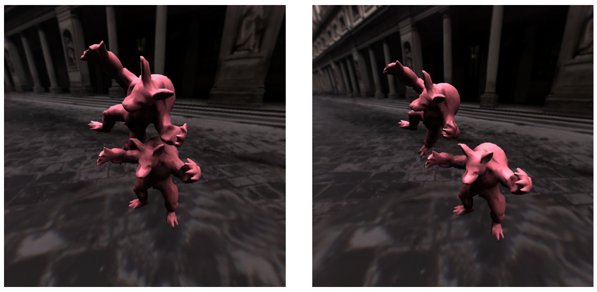


Figure 18: *Two images from a video rendered on 18 FPS. Observe how the illumination of the standing Armadillo changes when the other Armadillo flies over.*

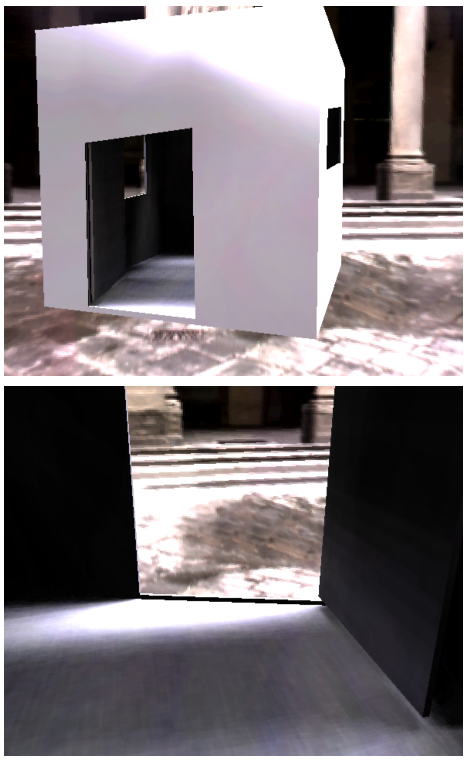


Figure 19: *Two images of difficult scene, where only a small fraction of the environment is visible from surface points inside the room. 1000 directional samples are necessary, animation with 5 FPS is possible.*

4. Ray tracing on the GPU

Due to the success of texture based approaches, accurate ray casting has lately been confined to performing preprocessing in realtime applications. Though GPU based ray casting implementations outperform the CPU now, they either do not scale well for higher primitive counts, or require the costly construction of spatial hierarchies. In this section we describe an improved algorithm based on the Ray Engine approach, which builds a hierarchy of rays instead of objects, completely on the graphics card.

Generally, our objective would be to eliminate time consuming ray casting from illumination algorithms, or to move it to a preprocessing step computing texture maps. However, there are some light transport effects that exhibit inherently recursive behavior, most prominently visible refractive objects, or caustics via multiple reflections or refractions. Accurate maps or transport factor matrices cannot be constructed with a feasible storage requirement. However, these problems are effectively handled by recursive ray tracing or photon tracing, both based on ray casting. Moreover, if we consider eye rays or light rays from small light sources, hitting reasonably smooth objects, the rays to be traced will be coherent, even after multiple reflections of refractions.

One delivering research direction has spawned from the approach of Purcell et al.[PBMH02a], implemented by Foley and Sugerman[FS05]. The kd-tree acceleration hierarchy, formerly confined to the CPU, is traversed on the GPU using algorithms not optimal in the worst-case algorithmic sense, but eliminating the need of a stack. This offers a competitive alternative to CPU ray tracing, but it is not directly targeted on real-time applications, and it is ill-suited for highly dynamic scenes because of the construction cost of the kd-tree.

However, there is a solution which does not rely on a pre-built acceleration structure: the Ray Engine[CHH02]. Based on the recognition that ray casting is a crossbar on rays and primitives, while scan conversion is a crossbar on pixels and primitives, the ray engine computes all possible ray-primitive intersections on the GPU.

4.1. The ray engine

As the ray engine serves as the basis of our approach, let us reiterate its working mechanism in current GPU terminology. Figure 20 depicts the rendering pass realizing the algorithm. Every pixel of the render target is associated with a ray. The origin and direction of rays to be traced are stored in textures that have the same dimensions as the render target. One after the other, a single ray casting primitive is taken, and it is rendered as a full-screen quad, with the primitive data attached to the quad vertices. Thus, pixel shaders for every pixel will receive the primitive data, and can also access the ray data via texture reads. The ray-primitive intersection calculation can be performed in the shader. Then, using the distance of the intersection as a depth value, a depth test is performed to verify that no closer intersection has been found yet. If the result passes the test, it is written to the render target and the depth buffer is updated. This way every pixel will hold the information about the nearest intersection between the scene primitives and the ray associated with the pixel. The pitfall of the ray engine is that it implements the naive ray casting algorithm of testing every ray against every primitive.

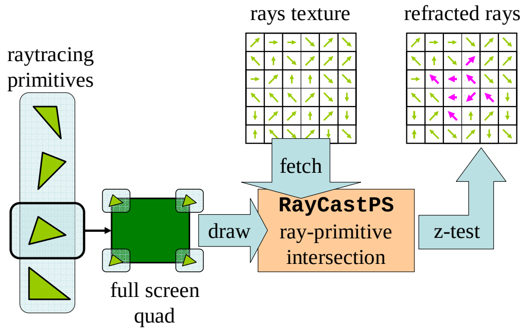


Figure 20: *Rendering pass implementing the ray engine.*

From this point on, we will refer to the primitives for the ray casting as triangles, this being the more general case. However, please note that the method is applicable to any other type of object for which an intersection test against a ray can be implemented in a shader.

4.2. Acceleration hierarchy built on rays

CPU-based acceleration schemes are spatial object hierarchies. The basic approach is that, for a ray, we try to exclude as many objects as possible from intersection testing. This cannot be done in the ray engine architecture, as it follows a 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

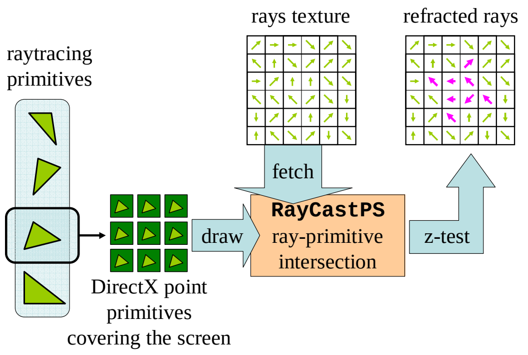


Figure 21: *Point primitives are rendered instead of full screen quads, to decompose the array of rays into tiles.*

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 fits well with the ray engine. Whenever the data of a primitive is processed, we should 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 (as illustrated in Figure 21) is to split the render target into tiles, render a set of tile quads instead of a full screen one, but make a decision for every tile beforehand whether it should be rendered at all. At a first glimpse, this may appear counterproductive, as, apparently, far more quads will be rendered. However, there is a set of issues that disprove concerns.

- The ray engine is pixel shader intensive, and makes practically no use of the vertex processing unit. The number of pixel shader runs, which remains crucial, is by no means increased.
- Instead of small quads, one can use point primitives, described by a single vertex. This eliminates the fourfold overhead of processing the same vertex data for all quad vertices, and needlessly interpolating values.
- The high level test of whether a tile may include valid intersections can be performed in the vertex shader. If the intersection test fails, the vertex is transformed out of view, and discarded by clipping. Moving the vertices out of view does not require any computation, they are simply assigned an outlying extreme position.
- We can render all the triangles (the primitives of ray casting) for a single tile at once. With a vertex buffer encoding the triangles, this will be a single draw call of point primitives. Tile data will be constant for all triangles, and can be passed in uniform registers.

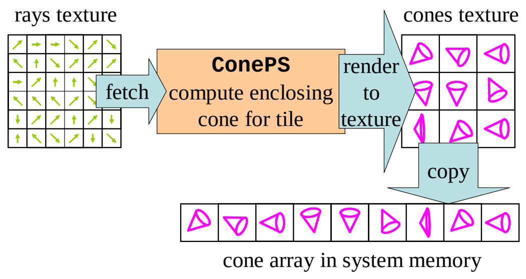


Figure 22: *The shader computing enclosing cones for tiles of rays. The result is read back to system memory.*

To be able to perform the preliminary test, for rays grouped in the same tile an enclosing object should be computed. This object will be an infinite cone. If we test it against the enclosing sphere of the triangle, we can exclude tiles not containing any intersections. As rays are described in textures, and are not static, the computation of ray-enclosing cones should be performed on the GPU, in a rendering pass, computing data to a texture. This step is shown in Figure 22. The shader is detailed in Section 4.6. To set the uniform parameters for the tiles, this texture has to be read back from the graphics card. However, as it contains only as many texels as many tiles are used (32×32 is typical), this is not an expensive operation.

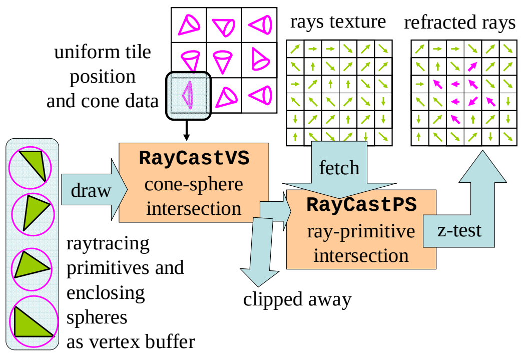

uniform tile position and cone data
rays texture
refracted rays
RayCastVS
cone-sphere intersection
draw
fetch
RayCastPS
ray-primitive intersection
z-test
raytracing primitives and enclosing spheres as vertex buffer
clipped away

Figure 23: *The rendering pass implementing the hierarchical ray engine. For every tile, the vertex buffer containing triangle and enclosing sphere data is rendered. The vertex shader discards the point primitive if the encoded triangle's circumsphere does not intersect the cone of the tile.*

Figure 23 shows how the hierarchical ray engine pass proceeds. The vertex buffer must also contain enclosing sphere data. For all the tiles, this vertex buffer is drawn. The tile position, and the ray-enclosing object description for the current tile are uniform parameters to the vertex shader. Based on the intersection test between the current triangle's and the tile's enclosing objects, the vertex shader either transforms the vertex out of view, or moves it to the desired tile position. The pixel shader performs the classic ray engine ray-triangle intersection test.

4.3. Implementation of recursive ray tracing using the ray engine

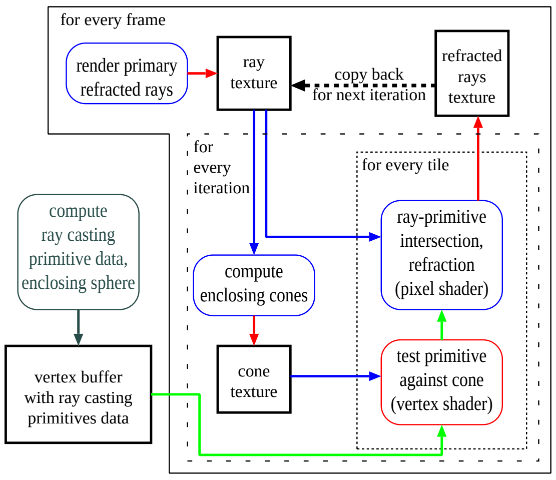

for every frame
render primary refracted rays
ray texture
copy back for next iteration
refracted rays texture
for every iteration
for every tile
compute ray casting primitive data, enclosing sphere
ray-primitive intersection, refraction (pixel shader)
compute enclosing cones
vertex buffer with ray casting primitives data
cone texture
test primitive against cone (vertex shader)

Figure 24: *Block diagram of the recursive ray tracing algorithm. Only the initial construction of the vertex buffer is performed on the CPU.*

Figure 24 depicts the data flow in the demo application for tracing refracted rays. The ray engine passes are iterated to render consecutive refractions of rays. The pixel shader performing the intersection tests outputs the refracted ray. That is, the ray defining the next segment of the refraction path is written to the render target. Then these results are copied back to the ray texture, and serve as input for the next iteration. Those pixels in which the path has not already terminated must not be processed. In the beginning of every iteration, an enclosing cone of rays is built for every tile, and stored in a texture. Data from this texture is used in ray-casting vertex shader runs to carry out preliminary intersection tests.

Note that the cones have to be reconstructed for every new generation of rays, before the pass computing the nearest intersections.

The steps of the complete algorithm therefore can be listed as follows:

1. Assembly of the vertex buffer encoding triangles (Section 4.4).
2. For every frame:
   a. Generation of the primary ray array. This is done by rendering the scene with a shader that outputs primary refracted rays (Shader technique `PrimaryRays` in Section 4.5).
   b. Compute the enclosing cones for tiles of rays, and read back the result to system memory (Shader technique `Cone` in Section 4.6).
   c. For every tile, draw the vertex buffer, rendering refracted rays where an intersection is found (Shader technique `RayCast` in Section 4.7). Valid results are flagged in the stencil buffer.
   d. Copy valid refracted rays back to the ray texture (Shader technique `CopyBack`).
   e. Repeat from step 2.b. until desired refraction depth is reached.
   f. Draw a full-screen quad, using the refracted exiting rays in the ray texture to address an environment map. This renders the refractive objects with environment mapping (Shader technique `Background`).

4.4. Construction of the vertex buffer

We need to be able to perform the intersection test between the enclosing object and the ray casting primitive as fast as possible. At the same time, representation of both the triangles and the cones must be compact, because triangle has to be passed in a very limited number of vertex registers, and enclosing cones must be described by a few texels when computed to a render target.

The triangle description must be adequate for two operations: cone-circumsphere intersection, and ray-triangle intersection. For the cone-circumsphere intersection, the center (3 floats) and the radius (1 float) of the sphere are stored. As this information will only be necessary in the vertex shader, we can use the `POSITION` slot in the vertex description. Evaluation of the ray-triangle intersection consist of following steps:

1. Find the intersection point of the ray and the plane of the triangle. For this we store the distance vector from the origin to the plane (3 floats). We use the `NORMAL` slot.
2. It has to be decided, whether the point is within the triangle. We can transform the Cartesian world coordinates to the triangle's Barycentric coordinates with a multiplication by a 3×3 matrix. If all barycentric coordinates are positive, there is an intersection. The rows of the transformation matrix are stored in `TEXCOORD` slots.
3. The data assigned to the vertices of the triangle should be interpolated. In our case this means the normals. These are the vectors assigned to the

model vertices in modeling, not to be confused with the normal vector of the triangles plane (the flat normal). We can find the interpolated surface normal by weighting triangle vertex normals with the barycentric coordinates. Normals are stored in additional `TEXCOORD` slots. If we also wish to add texturing, we need to encode and interpolate the texture coordinates of the triangles vertices in a similar manner.

4. The ray origin and direction should be read from the input textures, and knowing the surface normal, the refracted ray should be found and its origin and direction written to the render targets.

The shader implementation of the intersection algorithm will be detailed in Section 4.7. The structure representing a vertex in system memory is defined as follows:

```
// a vertex encoding a triangle
struct ProcessedTriangle  {
  // enclosing sphere centre  : POSITION.xyz
  float3 sphereCentre;
  // enclosing sphere radius : POSITION.w
  float  radius;
  // triangle plane point
  // nearest to origin : NORMAL
  float3 planePos;
  // Cartesian-to-barycentric
  // transformation matrix  : TEXCOORD0-2
  float3 inverseVertexMatrix[3];
  // model normals
  // at triangle vertices : TEXCOORD3-5
  float3 normals[3];
  // model texture coords
  // at triangle vertices : TEXCOORD6-8
  float3 tex[3];
};
```

The 3×3 transformation matrix is computed as the inverse of the matrix containing vertex coordinates. Barycentric coordinates are defined as:

$$\begin{bmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \\ a_z & b_z & c_z \end{bmatrix} \begin{bmatrix} B_a \\ B_b \\ B_c \end{bmatrix} = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix},$$

where a, b and c are Cartesian 3D triangle vertex positions, and B_a, B_b and B_c are the Barycentric coordinates of point p. To get the transformation from Cartesian coordinates, we have to invert this matrix.

4.5. Rendering primary rays

In order to create the initial rays for the ray engine, we must render the scene using standard modeling and camera transformations. The pixel shader outputs the fragment position in world coordiantes as the origin, and computes the refracted view direction. The ray origin and direction textures are set as render targets.

```
void PrimaryRaysVS (
  in float4   Pos          : POSITION,
  in float3   Norm         : NORMAL,
  out float4  hPos         : POSITION,
  out float3  wNorm        : TEXCOORD0,
  out float3  wPos         : TEXCOORD1)
{
    hPos = mul(Pos, WorldViewProj);
    wNorm = mul(Norm, WorldIT);
    wPos = mul(Pos, World).xyz;
}

void PrimaryRaysPS (
  in float3   wNorm        : TEXCOORD0,
  in float3   wPos         : TEXCOORD1,
  out float4  Origin       : COLOR0,
  out float4  Dir          : COLOR1)
{
  Origin = float4(wPos, 1);

  wNorm = normalize(wNorm);
  float3 ViewDir =  wPos - EyePos;
  ViewDir = normalize(ViewDir);

  float3 refractedDir =
    refract(ViewDir, wNorm, RefractionIndex);
  //total internal reflection
  if(dot(refractedDir, refractedDir) < 0.5)
    refractedDir = reflect(ViewDir, wNorm);

  //4th channel is used to store object color
  Dir = float4(refractedDir, color);
}
```

4.6. Construction of an enclosing cone

The intersection test between an infinite cone and a sphere is simple is simple enough to be quicky performed in the vertex shader. Enclosing infinite cones of rays are described by an origin, a direction and an opening angle.

Enclosing spheres for all ray casting primitives can easily be computed when the model mesh is loaded and the vertex buffer is assembled. A sphere is described by a 3D position and a radius. Note that these are always given in modeling coordinates, and transformations have to be applied when using them in shaders. In the vertex buffer, the `POSITION` value slot can be used for passing the enclosing sphere data, as it will simply be exchanged with the tile position in the vertex shader.

The infinite enclosing cones must be constructed in a pixel shader, in a pass before rendering the intersection records themselves. Note that in a practical application, the rays to be traced will be different for every frame, and for every level of refraction, so the reconstruction of the cones is also time critical. Therefore, a fast incremental approach is preferred over a tedious one, which could possibly produce more compact results, via, for instance, linear programming. The algorithm goes as follows:

1. Start with the zero angle enclosing cone of the first ray.
2. For each ray
   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 volume 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.

```
void ConeVS(
  in  float4  Pos         : POSITION,
  in  float2  Tex         : TEXCOORD0,
  out float4  oPos        : POSITION,
  out float2  oTex        : TEXCOORD0
  )
{
  oPos = Pos;
  oTex = Tex;
}

void
ConePS(
  in  float2  Tex         : TEXCOORD0,
  out float4  oConePeak   : COLOR0,
  out float4  oConeDir    : COLOR1)
{
  //current cone data
  //will be extended for every ray
  //if necessary
  float3 peak = 0;
  float3 dir = 0;
  float  cosAngle = 1.0;
  float  sinAngle = 0.0;

  // for all rays in tile
  for(int i=0; i < 16; i++)
    for(int j=0; j < 16; j++)
    {
      float3 currentRayDir =
        tex2D(rayDirTableSampler, Tex
          + float2(
            (float)j/RAY_TABLE_SIZE,
            (float)i/RAY_TABLE_SIZE) );
      if(dot(currentRayDir, currentRayDir)>0.5)
      { // a valid ray
        float3 currentRayOrigin =
          tex2D(rayOriginTableSampler, Tex
            + float2(
              (float)j/RAY_TABLE_SIZE,
              (float)i/RAY_TABLE_SIZE) );
        if(dot(dir,dir) < 0.5)
        { // no valid ray found before this one
            dir = currentRayDir;
            peak = currentRayOrigin;
        }
        else
        { // extend the cone to include ray
          float ncos = dot(currentRayDir,dir);
          if(ncos < cosAngle)
          { //open cone wider
            float3 perpdir = normalize(
              currentRayDir - ncos * dir);
            float3 farConeEdge =
              cosAngle * dir
              - sinAngle * perpdir;
            dir = farConeEdge + currentRayDir;
            dir = normalize(dir);
            cosAngle = dot(dir, currentRayDir);
            sinAngle =
              sqrt(1.0 - cosAngle * cosAngle);
          }
          float3 pd = currentRayOrigin - peak;
          float3 ptop = normalize(pd);
          float cospp = dot(dir, ptop);
          if(cospp < cosAngle)
          { //transplate cone to include origin
            float3 perpdir =
              ptop - cospp * dir;
            perpdir = normalize(perpdir);
            float3 farConeEdge =
              cosAngle * dir
              - sinAngle * perpdir;
            float3 nearConeEdge =
              cosAngle * dir
              + sinAngle * perpdir;
            float3 g =
              currentRayOrigin - peak
              - nearConeEdge
              * dot(nearConeEdge, pd);
            peak += farConeEdge
            * dot(g,g) / dot(farConeEdge, g);
          }
        }
      }
    }
  oConePeak = float4(peak, 1.0);
  oConeDir = float4(dir, cosAngle);
  if(dot(dir, dir)<0.5)//no valid rays in tile
  {
    //cone far away not intersecting anything
    oConePeak = float4(1000000.0, 0 , 0, 0.0);
    oConeDir = float4(1, 0, 0, 1.0);
  }
}
```

4.7. Ray casting

Using shader technique `PrimaryRays` we have rendered an array of rays to be traced, specified by the origin and the direction in world space. We have built a texture of enclosing cones, also in world space, corresponding to rays of tiles with `Cone`. Now, in the `RayCast` pass, we have to test these against primitives encoded in vertices, given in modeling space. We transform the cones and rays to modeling space for the intersection computation, and transform the results back, if necessary. The algorithm for the ray-triangle intersection was described in Section 4.4.

The cone intersects the sphere if

$$\varphi > \arccos[(\vec{v} - \vec{a}) \cdot \vec{x}] - \arcsin[r/|\vec{v} - \vec{a}|],$$

where $\vec{a}$ is the apex, $\vec{x}$ the direction and φ the half opening angle of the cone, $\vec{v}$ is the center of the sphere and r is its radius. The vertex shader has to test for this, and pass all the information necessary for the ray intersection test to the pixel shader. The cone is transformed using the inverse `World` transform to model space.

```
void RayCastVS(
  in float4   Sphere      : POSITION,
  in float3   PlanePos    : NORMAL,
  in float3   Invmx0      : TEXCOORD0,
  in float3   Invmx1      : TEXCOORD1,
  in float3   Invmx2      : TEXCOORD2,
  in float3   Normals0    : TEXCOORD3,
  in float3   Normals1    : TEXCOORD4,
  in float3   Normals2    : TEXCOORD5,

  out float4  hPos        : POSITION,
  out float3  oPlanePos   : TEXCOORD6,
  out float3  oInvmx0     : TEXCOORD0,
  out float3  oInvmx1     : TEXCOORD1,
  out float3  oInvmx2     : TEXCOORD2,
  out float3  oNormals0   : TEXCOORD3,
  out float3  oNormals1   : TEXCOORD4,
  out float3  oNormals2   : TEXCOORD5
  )
{
  oPlanePos = PlanePos;
  oInvmx0   = Invmx0;
  oInvmx1   = Invmx1;
  oInvmx2   = Invmx2;
  oNormals0 = Normals0;
  oNormals1 = Normals1;
  oNormals2 = Normals2;

  hPos = float4(TilePos, 1);

  ConeDirAndCosAngle.xyz =
    mul(WorldIT,
      float4(ConeDirAndCosAngle.xyz, 0));
  ConePeak.xyz =
    mul(WorldIT, float4(ConePeak.xyz, 1));

  float3 sfc = Sphere.xyz - ConePeak.xyz;
  float lsfc = length(sfc);

  if(Sphere.w < lsfc)
  { // cone peak not in sphere
    // angle difference between cone's
    // main direction
    // and direction to sphere centre
    float angSpMidConeMid =
      acos(dot(ConeDirAndCosAngle.xyz,
        sfc) / lsfc);
    // the half of the opening angle
    // at which the sphere is seen
    float angSpRad = asin(Sphere.w / lsfc);
    // cone opening angle
    float angCone = acos(ConeDirAndCosAngle.w);

    // if sphere direction not
    // within (cone direction + sphere angle),
    // discard tile for this triangle
    if(angCone + angSpRad < angSpMidConeMid)
    {
      hPos = float4(10000000.0,
                    10000000.0,
                    10000000.0, 1.0);
    }
  }
}
```

The pixel shader performs the intersection test on ray data read from the textures. The ray is transformed into model space, and the refracted ray origin and direction are transformed back usign the `World` transformation.

```
void RayCastPS(
  in float3   PlanePos    : TEXCOORD6,
  in float3   Invmx0      : TEXCOORD0,
  in float3   Invmx1      : TEXCOORD1,
  in float3   Invmx2      : TEXCOORD2,
  in float3   Normals0    : TEXCOORD3,
  in float3   Normals1    : TEXCOORD4,
  in float3   Normals2    : TEXCOORD5,

  in float2   vPos        : VPOS,
  out float4  oOrigin     : COLOR0,
  out float4  oDir        : COLOR1,
  out float1  oDepth      : DEPTH)
{
  // to texture coordinates
  float2 pixpos = vPos.xy;
  pixpos /= RAY_TABLE_SIZE;

  float3 rayOrigin =
    tex2D(rayOriginTableSampler, pixpos.xy);
  rayOrigin =
    mul(WorldIT, float4(rayOrigin, 1));
  float3 rayDir =
    tex2D(rayDirTableSampler, pixpos.xy);
  rayDir = mul(WorldIT, float4(rayDir, 0));

  //ray-plane
  float hitDepth =
    (dot(PlanePos, PlanePos)
    - dot(rayOrigin, PlanePos))
    / dot(rayDir, PlanePos);
  if(hitDepth < MIN_RAY_DEPTH
    || hitDepth > MAX_RAY_DEPTH)
  {
    oDepth = 1000000.0;
    oOrigin = oDir = 0;
  }
  else
  {
    float3 hitPoint =
      rayOrigin + (rayDir * hitDepth);

    float3 worldDist =
      mul(float4(hitPoint - rayOrigin, 0),
          World ).xyz
    oDepth = length(worldDist) / MAX_RAY_DEPTH;

    float baryA = dot(Invmx0, hitPoint);
    float baryB = dot(Invmx1, hitPoint);
    float baryC = dot(Invmx2, hitPoint);
    if(baryA > -0.001
      && baryB > -0.001
      && baryC > -0.001)
    {
      oOrigin =
      mul(float4(hitPoint, 1), World);

      float3 normalAtHit = Normals0 * baryA;
      normalAtHit += Normals1 * baryB;
      normalAtHit += Normals2 * baryC;
      normalAtHit = normalize(normalAtHit);
      if(dot(normalAtHit, rayDir) > 0)
      { // exiting ray
        normalAtHit = -normalAtHit;
        RefractionIndex =
          1.0 / RefractionIndex;
      }
      float3 refractedDir =
        refract(rayDir, normalAtHit,
          RefractionIndex);
      // total internal reflection
      if(dot(refractedDir,refractedDir)<0.5)
        refractedDir =
          reflect(rayDir, normalAtHit);
      oDir = float4(
        mul(float4(refractedDir,0), World).xyz,
        1);
    }
    else
    {
      oOrigin = float4(10.0, 0, 0, 1);
      oDir = float4(10.0, 0, 0, 1);
      oDepth = 1000000.0;
    }
  }
}
```

4.8. Excluding terminated ray paths

In our demo recursive ray tracing application, we render refractive objects in a cube map environment. To render a frame, we first have to generate a texture of initial rays. Then, we follow these rays through multiple refractions, until the exiting rays do not hit a refractive surface any more. As a result, we get the exiting rays, which can be used to query the cube map, realizing environment mapping.

Firstly, there may be pixels in which no refractive surface is visible. Furthermore, in every iteration replacing rays with their refracted successors, there will be rays not arriving on any refractive surface, producing no output. For those pixels where there is no ray to trace, the pixel shader is not invoked at all. We achieve this using the stencil buffer and early stencil testing. When rendering intersections, every bit of the stencil serves as a flag for a specific iteration. The stencil read and write masks select the flag bit of the previous and current iterations, respectively. Should ray casting fail to hit any object, the stencil bit will not be set, and in the next iteration the pixel will be skipped. With an eight bits deep stencil buffer, this allows for eightfold reflection or refraction to be traced. Further iterations are possible without excluding further terminated paths.

In the `PrimaryRays` pass, we mark valid pixels:

```
device->SetRenderState(
  D3DRS_STENCILENABLE, TRUE);
device->SetRenderState(
  D3DRS_STENCILFUNC, D3DCMP_ALWAYS);
device->SetRenderState(
  D3DRS_STENCILPASS, D3DSTENCILOP_REPLACE);
device->SetRenderState(
  D3DRS_STENCILREF, 0x1);
hr=device->SetRenderState(
  D3DRS_STENCILMASK, 0x1);
hr=device->SetRenderState(
  D3DRS_STENCILWRITEMASK, 0x01);
```

Before every `RayCast` pass, we have to set:

```
device->SetRenderState(
  D3DRS_STENCILREF, 0xff);
device->SetRenderState(
  D3DRS_STENCILFUNC, D3DCMP_EQUAL);
device->SetRenderState(
  D3DRS_STENCILPASS, D3DSTENCILOP_REPLACE);
device->SetRenderState(
  D3DRS_STENCILFAIL, D3DSTENCILOP_KEEP);
device->SetRenderState(
  D3DRS_STENCILMASK, 0x1 << iterationCount);
device->SetRenderState(
  D3DRS_STENCILWRITEMASK, 0x2 << iterationCount);
```

This will preemt output for pixels not successfully processed previously, and set a flag for those pixels where an intersection was found. The flag shifting is necessary because multiple primitives will be rendered onto the same pixel in a single pass. Overwriting the stencil bit would be unacceptable.

4.9. Conclusion

The DirectX demo application renders multiple refractive objects in a cube map environment (Figure 26). The rendering resolution is 512 × 512, divided into 32 × 32 tiles, all rendered as 16 × 16 sized DirectX point primitives.

```
13.03 fps (512x512), X8R8G8B8 (D24S8)
HAL (pure hw vp): NVIDIA GeForce 6800 GT
Press 'F2' to configure display
Use arrow keys to update input
World State: 0.000, 0.000
13.03 fps (512x512), X8R8G8B8 (D24S8)
```
```
9.75 fps (512x512), X8R8G8B8 (D24S8)
HAL (pure hw vp): NVIDIA GeForce 6800 GT
Press 'F2' to configure display
Use arrow keys to update input
World State: 0.000, 0.000
9.75 fps (512x512), X8R8G8B8 (D24S8)
```

Figure 25: *Images rendered using the hierarchical algorithm at* 256 × 256 *resolution.*

As the application demonstrates, the ray engine approach is capable of performing recursive ray tracing at interactive frame rates. However, performance depends linearly on ray-casting primitive count. You should also remember that not oly triangles, but any kind of raytracable objects can be used. Therefore, when the ray engine is integrated into an interactive environment, it should be used cleverly together with incremental 3D to add highly accurate reflections, refractions and caustics generated by relatively low primitive count objects.

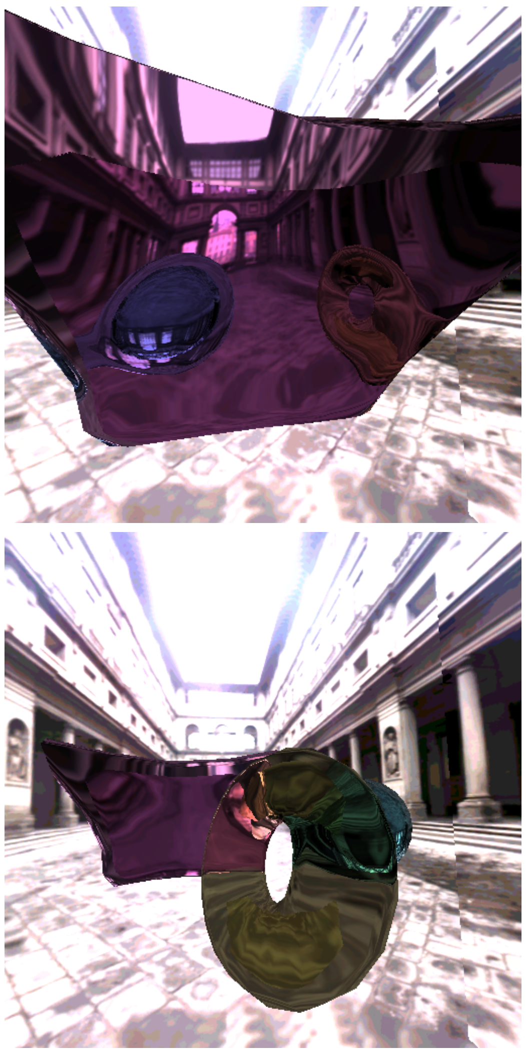


Figure 26: *Images rendered using the hierarchical algorithm at* 512 × 512 *resolution.*

5. Specular effects with rasterization

When computing the light transfer, the basic operation is tracing a ray from its origin point at a direction to find that point which is the source of illumination. To obtain a complete path, ray tracing should be continued at the hit point to get second and further scattering points. Current graphics processing units (GPU) trace rays of the same origin very efficiently taking a "photo" from the shared ray origin. The photographic process involves the rasterization of the scene geometry and the exploitation of the z-buffer to find the first hits of the rays passing through the pixels. Note that this process finds light paths of length 1 starting at the same point.

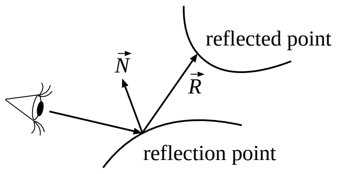


Figure 27: *Specular effects require searching for complete light paths in a single step.*

However, in reflection, refraction, and caustic computations the rays required to be traced are not so coherent. Having identified the points visible from the camera, from each point just a single ray needs to be cast into the reflection or refraction direction. Ideal reflection, refraction, or caustic computation can be regarded as a search process that finds light paths containing more than one scattering points (figure 27). For example, in case of reflection we first find the point visible from the camera, which is the place of reflection. Then from the reflection point we need to find the *reflected point* to reflect its radiance at the reflection point toward the camera. Refraction is similar, just we have to follow the refraction rather than the reflection direction. Note also that even caustics is similar with the difference that the search process starts from the light source and not from the camera. It might be mentioned that in all cases the multiple points correspond to the shortest optical path according to the Fermat principle.

Searching for a complete light path at a single step is not easy since it requires the processing of the scene multiple times, which is not compatible with the stream processing architecture of the GPU (this architecture assumes that each vertex and fragment are processed at once and independently). A GPU friendly approximation technique to simulate paths containing two scattering points is *environment mapping* [BN76], which assumes that the reflected points are very (infinitely) far, and thus the hit points of the rays become

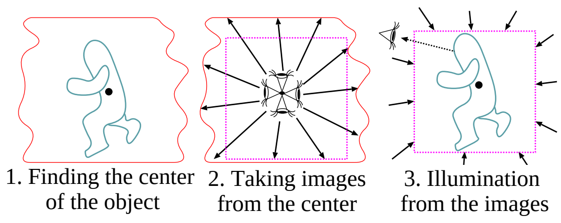

1. Finding the center of the object
2. Taking images from the center
3. Illumination from the images

Figure 28: *Steps of environment mapping*

independent of the reflection points, i.e. the ray origins. In this case rays can be translated to the same *reference point*, so we get that case back for which the GPU is an optimal tool. A fundamental problem of environment mapping is that the environment map is the correct representation of the direction dependent illumination only at its reference point. For other points, accurate results can only be expected if the distance of the point of interest from the reference point is negligible compared to the distance from the surrounding geometry (figure 29).

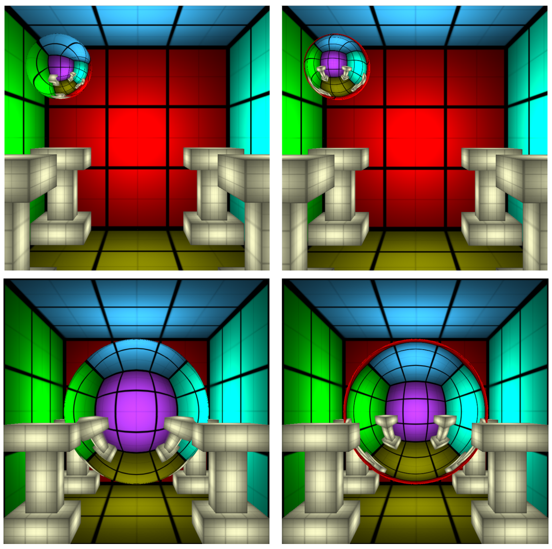


Figure 29: *Ray tracing (left) compared to approximate environment mapping (right)*

To obtain more accurate results, the origin of the ray and the distances from the environment surfaces must also be taken into account during ray-tracing [SKALP05, Bjo04, PMDS06]. However, instead of working directly with the meshes of the surfaces, we use the depth [Pat95] or distance values between the reference point and the points of the environment surface visible at texels of the environment map. The environment map storing per texel distance information is called *distance impostor*. Note that the distance impostor is the sampled representation of the environment geometry. Thus when a ray is traced, the intersection calculation can use this information instead of the triangular meshes. Since vertex and pixel shaders can lookup texture maps but cannot directly access meshes, this replacement is crucial for the GPU implementation.

Of course, a single map cannot always guarantee correct results if view dependent occlusions occur. As the ray origin gets far from the reference point of the distance impostor, the probability of such occlusions increases. Thus we might have to maintain multiple maps or regenerate the environment map when the movement exceeds a threshold. Since the environment map is not refreshed in every frame, the amortized cost of its generation becomes negligible.

Note that distance impostors may support searches for the second, third, etc. points of the light path, while the first point is identified by rasterization. Searches can also be organized differently. For example, we can start at the reflected points, i.e. at the vertices of the environment, and search for the reflection points, i.e. identify those points that reflect the input points. This means searching on the reflector surface rather than on the environment surface [EMD*05, RHS06, EMDT06]. The comparative advantages of searching on the environment surface or searching on the reflector depend on the geometric properties of these surfaces. The algorithm searching on the environment surface can handle arbitrary reflector surfaces and is efficient if the environment surface is either far or simple. On the other hand, the algorithm searching on the reflector surface can cope with arbitrary environment surfaces, but is limited to either concave or convex reflectors.

Here we detail an algorithm searching on the environment surface represented by a distance impostor [SKALP05].

5.1. Approximate ray-tracing with distance impostors

The basic idea is discussed using the notations of figure 30. Let us assume that center $\vec{o}$ of our coordinate system is the reference point of the environment map and we are interested in the illumination of point $\vec{x}$ from direction $\vec{R}$. We suppose that direction vector $\vec{R}$ has unit length.

Classical environment mapping would look up the illumination selected by direction $\vec{R}$, that is, it would use the radiance of point $\vec{r}$. However, $\vec{r}$ is usually not equal to really reflected point $\vec{q}$, which is in direction

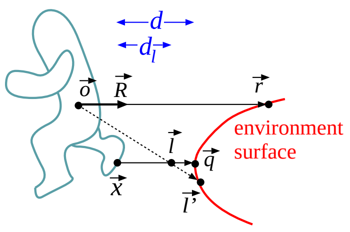


Figure 30: *Localization of the environment map having reference point $\vec{o}$. When computing the incoming radiance at point $\vec{x}$ from direction $\vec{R}$, ray tracing would select point $\vec{q}$, classical environment mapping would read the radiance of point $\vec{r}$, while the proposed method calculates $\vec{l}$ approximating the hit point on the ray and looks up the environment map in this direction obtaining the radiance of point $\vec{l}'$.*

$\vec{R}$ from $\vec{x}$, and thus satisfies the following ray equation for some distance d:

$$\vec{q} = \vec{x} + \vec{R} \cdot d. \quad (5)$$

Ray parameter d can be found by an iterative process working with distances between the environment and reference point $\vec{o}$. The required distance information can be computed during the generation of the environment map. While a normal environment map stores the illumination for each direction in R,G,B channels, now we also obtain the distance of the visible point for these directions and store it, for example, in the alpha channel.

5.1.1. Finding initial ray hit approximations

In order to start the iterative ray intersection method, we need initial guesses for the solution. Classical environment mapping would look up the illumination selected by direction $\vec{R}$, that is, it would use the radiance of point $\vec{r}$. This can be considered as the first guess for the ray hit.

To find a second guess, we assume that the environment surface at $\vec{r}$ is perpendicular to ray direction $\vec{R}$ (figure 31). In case of perpendicular surface, the ray would hit point $\vec{p}$. Points $\vec{r}$, $\vec{x}$ and origin $\vec{o}$ define a plane, which is the base plane of figure 31. This plane also contains visible point approximation $\vec{p}$ and unit direction vector $\vec{R}$. Multiplying ray equation

$$\vec{x} + \vec{R} \cdot d_p = \vec{p}$$

by direction vector $\vec{R}$ and substituting $\vec{R} \cdot \vec{p} = |\vec{r}|$, which is the consequence of the perpendicular surface assumption, we can express ray parameter d_p:

$$d_p = |\vec{r}| - \vec{R} \cdot \vec{x}. \quad (6)$$

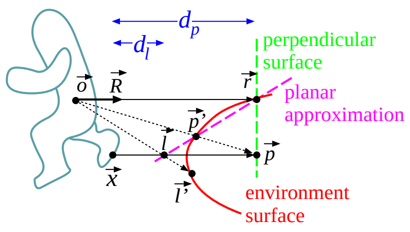


Figure 31: *Identifying first approximation point $\vec{p}$ assuming that the surface is perpendicular to $\vec{R}$, and second approximation point $\vec{l}$ supposing that the surface is planar between points $\vec{r}$ and $\vec{p}'$.*

If we used the direction of point $\vec{p}$ to lookup the environment map, we would obtain the radiance of point $\vec{p}'$, which is in the direction of $\vec{p}$ but is on the surface.

The accuracy of an arbitrary approximation $\vec{l}$ can be checked by reading the distance stored with the direction of $\vec{l}$ in the environment map ($|\vec{l}'|$) and comparing it with $|\vec{l}|$. If $|\vec{l}| = |\vec{l}'|$, then we have found the intersection. If visible point approximation is in front of the surface, that is $|\vec{l}| < |\vec{l}'|$, the current approximation is an *undershooting* of distance parameter d. On the other hand, the case when point $\vec{l}$ is behind the surface ($|\vec{l}| < |\vec{l}'|$) is called *overshooting*.

Since point $\vec{r}$ corresponds to infinite ray parameter, it is the projected point of an overshooting. Point $\vec{p}$ obtained with the perpendicular surface assumption can be either undershooting or overshooting. On the other hand, if the object does not intersect the environment, shaded point $\vec{x}$ is an undershooting.

The iteration process finds the real intersection starting with two initial guesses.

5.1.2. Refinement by iteration

Suppose that we have two initial guesses of the ray parameter d_p and d_l, and consequently two points $\vec{p}$ and $\vec{l}$ that are on the ray, but are not necessarily on the surface, and two other points $\vec{p}'$ and $\vec{l}'$ that are on the surface, but are not necessarily on the ray (figure 32). Note that if $\vec{l}'$ is equal to $\vec{r}$, then $\vec{l}$ does not exist in the Euclidean space and ray parameter d_l gets infinite, thus this case requires special considerations.

If the surface were a plane between $\vec{p}$ and $\vec{l}$, then the intersection of the planar surface and the plane defined by points $\vec{p}$, $\vec{l}$ and $\vec{o}$ would be a line, and the point visible from $\vec{x}$ at direction $\vec{R}$ would be $\vec{l}_{new}$. Let us assume that the surface can be well approximated by a plane between points $\vec{p}$ and $\vec{l}$, and find intersection $\vec{l}_{new}$ of the plane with the ray.

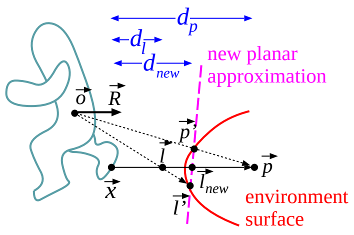


Figure 32: *Refinement by iteration.*

Since new approximation $\vec{l}_{new}$ is on the ray, it satisfies the following ray equation:

$$\vec{l}_{new} = \vec{x} + \vec{R} \cdot d_{new}. \tag{7}$$

Point $\vec{l}_{new}$ is also on the line of $\vec{l}'$ and $\vec{p}'$, thus it can be expressed as their combination with unknown weight α:

$$\vec{l}_{new} = \vec{l}' \cdot \alpha + \vec{p}' \cdot (1 - \alpha).$$

If one of the $\vec{p}$ and $\vec{l}$ points is an undershooting while the other is an overshooting, then α is in $[0, 1]$. If both points have the same type, then α is not restricted to $[0, 1]$ but can have an arbitrary value.

Assuming first that $\vec{l}' \neq \vec{r}$ and substituting identities $\vec{p}' = \vec{p} \cdot |\vec{p}'|/|\vec{p}|$ and $\vec{l}' = \vec{l} \cdot |\vec{l}'|/|\vec{l}|$, as well as the ray equation for d_p and d_l, we get:

$$\vec{l}_{new} = (\vec{x} + \vec{R} \cdot d_l) \cdot \frac{|\vec{l}'|}{|\vec{l}|} \cdot \alpha + (\vec{x} + \vec{R} \cdot d_p) \cdot \frac{|\vec{p}'|}{|\vec{p}|} \cdot (1 - \alpha).$$

Comparing this expression with equation 7, we obtain the following requirements for unknowns α and d_{new}:

$$\frac{|\vec{l}'|}{|\vec{l}|} \cdot \alpha + \frac{|\vec{p}'|}{|\vec{p}|} \cdot (1 - \alpha) = 1,$$

$$d_l \cdot \frac{|\vec{l}'|}{|\vec{l}|} \cdot \alpha + d_p \cdot \frac{|\vec{p}'|}{|\vec{p}|} \cdot (1 - \alpha) = d_{new}.$$

Solving this equation, we get:

$$d_{new} = d_l + (d_l - d_p) \cdot \frac{1 - |\vec{l}|/|\vec{l}'|}{|\vec{l}|/|\vec{l}'| - |\vec{p}|/|\vec{p}'|}. \tag{8}$$

Examining the special case when $\vec{l}' = \vec{r}$, then we obtain:

$$\vec{l}_{new} = \vec{r} \cdot \alpha + (\vec{x} + \vec{R} \cdot d_p) \cdot \frac{|\vec{p}'|}{|\vec{p}|} \cdot (1 - \alpha).$$

Solving this equation we get:

$$d_{new} = d_p + |\vec{r}| \cdot \left(1 - \frac{|\vec{p}|}{|\vec{p}'|}\right). \tag{9}$$

Having obtained new distance approximation d_{new} we can replace one of the previous approximations d_l or d_p and proceed with the same iteration step. We can select from different options. On the one hand, we can replace the last but one guess d_p by last guess d_l and d_l by d_{new} together with their associated points on the ray and on the surface [SKALP05]. On the other hand, we can examine the types of the two previous guesses and of the new point, and replace the previous guesses in a way that we always have an undershooting and an overshooting approximation [SKAL05].

From mathematical point of view, the proposed iteration method solves ray equation $f(\vec{x} + \vec{R} \cdot d) = 0$ where $f(\vec{r}) = 0$ is the implicit equation of the environment surface, which is represented by discrete distance samples of the environment map.

The solution algorithm is equivalent to the *secant method* [Wei03] if always the last but one guess is dropped. However, when we always keep one overshooting and one undershooting approximations, the solution algorithm is equivalent to the *false position method* [Wei03]. Both the secant and the false position method obtains exact results in a single step if the equation is linear. Note that this is the case if the two guesses happen to be on the same polygon. Thus having the algorithm iterated until the two guesses are on the same polygon, the next step provides exact solution.

The false position method is known to converge surely. The secant method usually converges even faster than the false position method [Wei03], but it may not converge for high variation functions. Note that in equation 8 the absolute value of denominator $|\vec{l}|/|\vec{l}'| - |\vec{p}|/|\vec{p}'|$ can be smaller than the absolute value of enumerator $1 - |\vec{l}|/|\vec{l}'|$, which means that step size $|d_l - d_p|$ may also increase. While increasing the step size where necessary improves convergence, it might also cause divergence. It may happen, for example, that wild fluctuations cause the ray parameter to get a negative value, which should be avoided. Thus when the secant method is used, the sign of the ray parameter is checked and the ray parameter is forced to be always positive.

Note that even with the guaranteed convergence of the false position method, the proposed method is not necessarily equivalent to exact ray tracing in the limiting case. Small errors may be due to the discrete surface approximation, or to view dependent occlusions. For example, should the ray hit a point that is not visible from the reference point of the environment map, then the presented approximation scheme would obviously be unable to find that. However, when the object is curved and moving, these errors can hardly be recognized visually.

5.2. Reflections

In order to render objects specularly reflecting their environment, we need to generate the distance impostor of the environment, then run the approximate ray tracing scheme if the reflection object is shaded in a fragment.

The computation of distance impostors is very similar to that of classical environment maps. The only difference is that the distance from the reference point is also calculated, which can be stored in a separate texture or in the alpha channel of the environment map. For the sake of simplicity, we suppose that the distance values are stored in the alpha channel of the environment map. However, when implementing the method it is worth putting the distance values into a separate texture to increase performance.

Since the distance is a non linear function of the homogeneous coordinates of the points, correct results can be obtained only by letting the pixel shader compute the distance values. Environment maps are usually parameterized by *cube mapping*, which projects onto the six faces of a cube. Having placed the camera at the reference point and set its viewing direction to the directions of coordinate axes, the scene is rendered six times.

Having the distance impostor, we render the reflective objects and activate custom vertex and fragment shader programs. The vertex shader transforms the reflective object to clipping space, and also to the coordinate system of the environment map first applying the modeling transform, then translating to the reference point. This space may be called as *environment map space*. View vector $\vec{V}$ and normal $\vec{N}$ are also obtained in this space.

```
void SpecularReflectionVS(
   in  float4 Pos  : POSITION, // modeling space
   in  float3 Norm : NORMAL,   // normal vector
   in  float2 Tex  : TEXCOORD0,// texture uv
   out float4 hPos : POSITION, // clipping space
   out float3 x    : TEXCOORD1,// env. space
   out float3 N    : TEXCOORD2,// normal in env.
   out float3 V    : TEXCOORD3 // view in env.
) {
   hPos = mul(Pos, WorldViewProj);
   x    = mul(Pos, World) - refpoint;
   N    = mul(Norm, WorldIT);
   V    = x - EyePos;
}
```

Having the graphics hardware computed the homogeneous division and filled the triangle with linearly interpolating all vertex data, the pixel shader is called to find ray hit $\vec{l}$ and to look up the cube map in this direction.

In the fragment shader we can call the following

HLSL function computing hit point approximation $\vec{l}$ with initial secant approximation and then turning to the false position method is shown below:

```
float3 Hit(half3 x, half3 R, sampler mp) {
   half rl = texCUBE(mp, R).a;       // |r|
   half dp = rl - dot(x, R);
   half3 p = x + R * dp;
   half ppp = length(p)/texCUBE(mp,p).a; //|p|/|p'|
   half dun = 0, dov = 0, pun = ppp, pov = ppp;
   if (ppp < 1) dun = dp; else dov = dp;
   half dl = max(dp + rl * (1 - ppp), 0);
   half3 l = x + R * dl;
                                     // iteration
   for(int i = 0; i < NITER; i++) {
      half ddl;
      half llp = length(l)/texCUBE(mp,l).a; //|l|/|l'|
      if (llp < 1) {                 // undershooting
         dun = dl; pun = llp;
         ddl = (dov == 0) ? rl * (1 - llp) :
               (dl-dov) * (1-llp)/(llp-pov);
      } else {                       // overshooting
         dov = dl; pov = llp;
         ddl = (dun == 0) ? rl * (1 - llp) :
               (dl-dun) * (1-llp)/(llp-pun);
      }
      dl = max(dl + ddl, 0);         // avoid flip
      l = x + R * dl;
   }
   return l;
}
```

This function gets ray origin x and direction R, as well as cube map mp of the environment, and returns hit approximation l.

Ratios $|\vec{l}|/|\vec{l}'|$ and $|\vec{p}|/|\vec{p}'|$ are represented by variables llp and ppp, respectively. Note that variables dun and dov store the last undershooting and overshooting ray parameters. If there has been no such approximation, the ray parameters are zero. In this case default point $\vec{r}$ takes their roles. In order to avoid ray flipping, the algorithm limits ray parameters for the non-negative domain.

An alternative implementation starts with overshooting $\vec{r}$ and undershooting $\vec{x}$ thus the false position algorithm can be executed from the beginning:

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

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

The fragment shader calls function Hit and looks up cube map envmap again to find illumination I of the hit point. The next step is the computation of the reflection of incoming radiance I. If the surface is an ideal mirror, the incoming radiance should be multiplied by the Fresnel term evaluated for the angle between surface normal $\vec{N}$ and reflection direction $\vec{R}$. The Fresnel function is approximated according to equation 4 [LSK05].

```
samplerCUBE envmap; // distance impostor

float4 SpecularReflectionPS(
   float3 x : TEXCOORD1, // env. space
   float3 N : TEXCOORD2, // normal in env.
   float3 V : TEXCOORD3  // view in env.
) : COLOR
{
   V = normalize(V);
   N = normalize(N);
   float3 R = reflect(V, N);     // reflection dir.
   float3 l = Hit(x, R, envmap); // ray hit float3
   float3 I = texCUBE(envmap, l).rgb; // in radiance

   // Fresnel reflection
   float F = Fp + pow(1-dot(N, -V), 5) * (1-Fp);
   return F * I;
}
```

Figure 33 compares images rendered by the proposed method with standard environment mapping and ray tracing. Note that for such scenes where the environment is convex from the reference point of the environment map, and there are larger planar surfaces, the new algorithm converges very quickly. In fact, even the initial guesses are usually accurate, and iteration is needed only close to edges and corners.

Figure 34 shows a difficult case where the box makes the environment surface concave and of high variation. Note that the convergence is still pretty fast, but the converged image is not exactly what we expect. We can observe that the green edge of the box is visible in a larger portion of the reflection image. This phenomenon is due to the fact that a part of the wall is not visible from the reference point of the environment map, but are expected to show up in the reflection. In such cases the algorithm can go only to the edge of the box and substitutes the reflection of the occluded points by the blurred image of the edge.

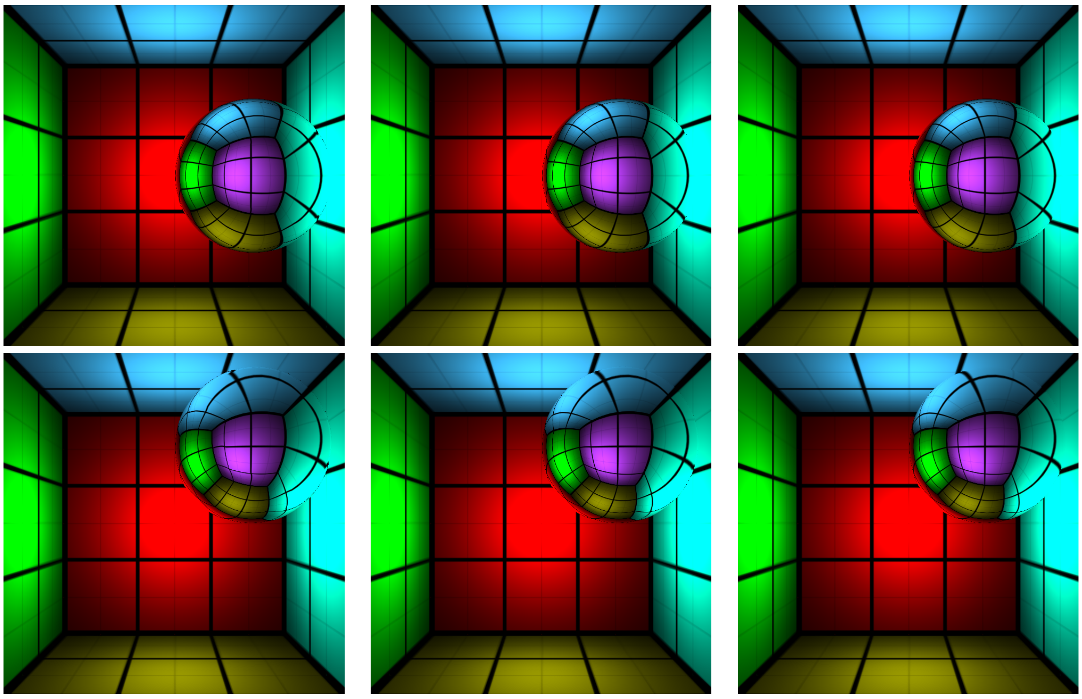


Figure 33: *Comparison of classical and localized environment map reflections with ray traced reflections placing the reference point at the center of the room and moving a reflective sphere to different locations. Note that even the initial guess made with the distance impostor is accurate almost everywhere but the corners where one iteration step is enough. The FPS values are measured with* 700×700 *resolution on an NV6800GT.*

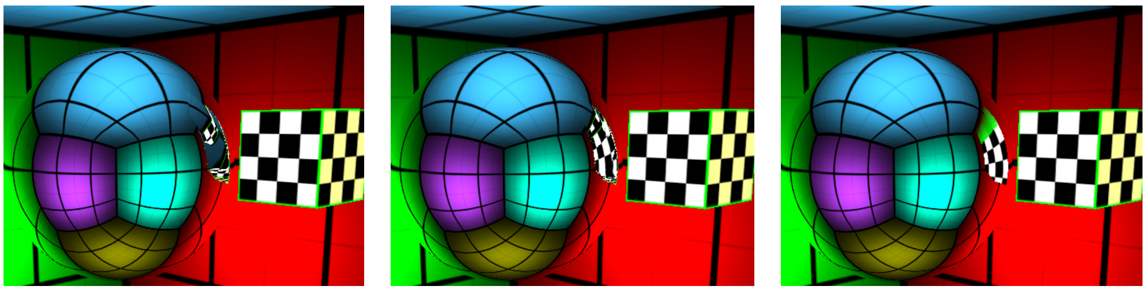


Figure 34: *A more difficult case when the room contains a box that makes the scene strongly concave and is responsible for view dependent occlusions.*

5.3. Multiple refractions

The proposed method can be used to simulate not only reflected but also refracted rays, just the direction computation should be changed from the law of reflection to the Snellius-Descartes law of refraction, that is, the `reflect` operation should be replaced by the `refract` operation in the pixel shader. However, tracing a refraction ray on a single level is usually not enough since the light is refracted at least twice to go through a refractor. The location of the second refraction as well as the normal vector at this point depend on the geometry of the object, which can only be analyzed by ray-tracing unless the refractor is very special (e.g. a sphere, a cylinder, etc.). This problem is usually solved by ignoring the second and further reflections, which is obviously a drastic simplification [Wym05].

Applying distance impostors, however, we can solve this problem as well if the refractor is not strongly concave, i.e. all surface points can be seen from its center point. We create a distance impostor for each refractor, which stores the distance of the refractor surface from its center and the normal vector of the surface. If the refractor has static geometry, these impostors can be obtained during preprocessing. We call this distance impostor as the *refractor map* (`refrmap` in the program).

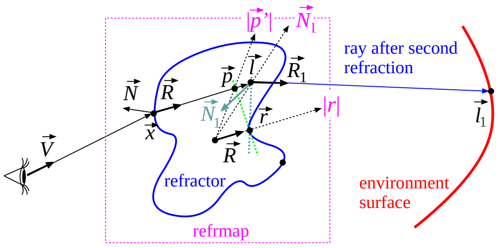


Figure 35: *Multiple refractions without iterative refinement. The ray refracts at $\vec{x}$ to direction $\vec{R}$. The refractor map is looked up in direction $\vec{R}$ to obtain $\vec{r}$. Using the perpendicular surface and planar surface assumptions we get $\vec{p}$ and $\vec{l}$, respectively. The refractor map is looked up in direction $\vec{l}$ to find normal $\vec{N}_1$ of the second refraction. Then the ray is refracted again at $\vec{l}$ using normal $\vec{N}_1$, and the process is continued with the environment map localization.*

Now let us consider point $\vec{x}$ on the surface of the refractor visible from the camera (figure 35). Using the proposed method for the refractor map, we obtain that point which is hit by the refraction ray. The normal vector at this point can be read from the distance impostor. Computing another refraction at this point and setting the origin of the refraction ray to the previously identified point, we can continue with the real environment map and find that point and color, which is visible after two refractions.

The pixel shader of double refractions uses the refractor map (`refrmap`) with reference point stored in variable called `refpoint`:

```
float Fp; // Fresnel at perpendicular dir.
float n;  // index of refraction
samplerCUBE envmap; // distance impostor of the environment
samplerCUBE refrmap;// refractor map

float4 SpecularMultipleRefractionPS(
   float3 x : TEXCOORD1,// env. space
   float3 N : TEXCOORD2,// normal in env.
   float3 V : TEXCOORD3 // view in env.
) : COLOR
{
   V = normalize(V);
   N = normalize(N);
   float3 R = refract(V, N, 1/n); // first refract.
   float3 l = Hit(x-refpoint, R, refrmap);
   // Fresnel
   float F1 = Fp + pow(1-dot(N, -V), 5) * (1-Fp);
   float3 N1 = texCUBE(refrmap, l).xyz;
   R1 = refract(R, N1, n);    // second refract.
       // distance impostor lookup
   float3 l1 = Hit(l+refpoint, R1, envmap);
   // Fresnel
   float F2 = Fp + pow(1-dot(N1, -R1), 5) * (1-Fp);
   float3 I = texCUBE(envmap, l1); // in rad.

   // Fresnel attenuation of the two refractions
   return (1-F1) * (1-F2) * I;
}
```

Figure 36 shows a refracting sphere rendered with classical environment map and also by the new method with single and double refractions. Note that multiple refraction has a significant effect even when the refraction index is close to 1, and also that the proposed method is quite accurate even with only one additional iteration step.

Reflection and refraction are computed by the same basic mechanism, thus their combination is straightforward. At each step the Fresnel function is responsible for weighting the two contributions.

When the ray arrives at the surface boundary from an optically dense material (e.g. glass) and goes to a less dense material (air), the relative index of refraction becomes less than one. In such cases, at higher incident angles, the sine of the refraction angle computed by the Snellius-Descartes law gets larger than 1, which prohibits any refraction. The phenomenon is called the case of *total reflection*, when the light is completely reflected at the surface. In such cases, the `refract` function returns with a zero vector. When it happens, only the reflection direction should be fol-

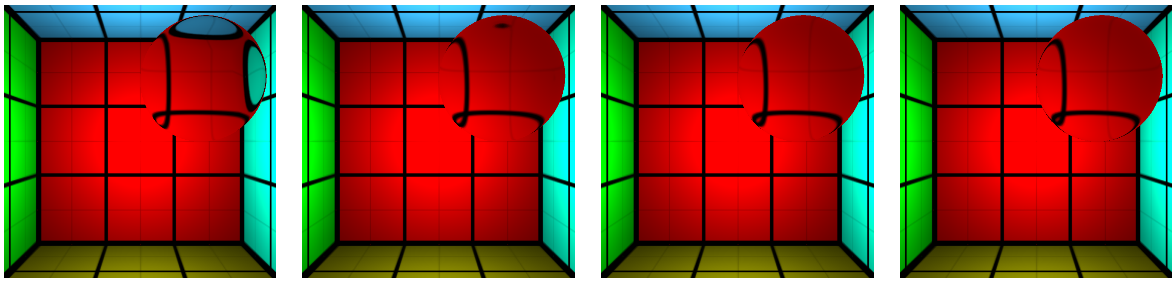


Figure 36: *Refractions of a sphere having refraction index* $n = 1.1$.

lowed and the Fresnel function should be adjusted to 1.

5.4. Caustics

The method presented so far can compute the hit point after the first (or higher order) reflection or refraction of the visibility ray. If we replace the eye by a light source, the same method can also be used to determine the first (or higher order) bounce of the light ray, thus we can compute the indirect illumination bounced off dynamic objects onto static ones [SKALP05, SP05, WD06b, WD06a].

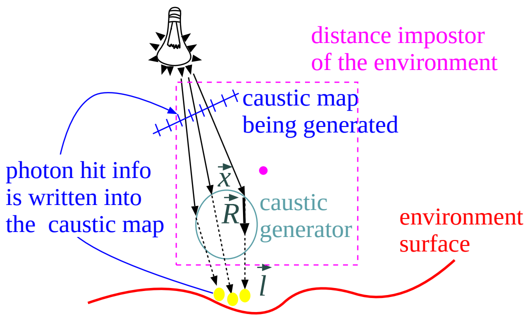


Figure 37: *Caustics generation with distance impostors*

These indirect effects have a significant impact on the final image if the dynamic object is close to be an ideal reflector or refractor, when these effects show up in forms of caustic spots [Jen01, TS00, WS03]. The proposed distance impostors can thus be used to compute *caustics*.

When rendering the scene from the point of view of the light source, the view plane is placed between the light and the refractor (figure 37). The image on this view plane is called *caustic map*. Note that this step is very similar to the generation of depth images for shadow maps. In fact, if we combine the method with shadow mapping, we obtain caustics almost for free. However, implementing the caustic map generation separately allows us to optimally set the position and the resolution of the view plane of the caustic map.

Supposing that the surface is an ideal reflector or refractor, point $\vec{l}$ that receives the illumination of a light source after a single or multiple reflection or refraction can be obtained by the proposed approximate ray tracing, and particularly by calling the `Hit` function, after making the following substitutions: point $\vec{x}$ is visible from the light source through a caustic map pixel, $\vec{R}$ is the refraction (or reflection) of the direction from the light source onto the surface normal at $\vec{x}$. The localized environment map lookups provide an approximation of that point $\vec{l}$, which is hit by a photon after a single reflection, or alternatively, $\vec{l}$ is an approximation of the direction of the photon hit from the reference point of the environment map.

The photon hit parameters are stored in that caustic map pixel through which the primary light ray arrived at the caustic generator object. There are several alternatives to represent a photon hit. Considering that the reflected radiance caused by a photon hit is the product of the BRDF and the power of the photon, the representation of the photon hit should identify the surface point and its BRDF. A natural identification is the texture coordinates of that surface point, which is hit by the ray. A caustic map pixel stores the identification of the texture map, the u and v texture coordinates, and finally the luminance of the power of the photon. The photon power is computed from the power of the light source and the solid angle subtended by the caustic map pixel.

The identification of the u and v texture coordinates from the direction of the photon hit requires another texture lookup. Suppose that together with the environment map, we also render another map, called `uvmap`, which has the same structure, but stores the u, v coordinates and the texture id in its pixels. Having found the direction of the photon hit, this map is read to obtain the texture coordinates, which are finally written into the caustic map.

The vertex shader of caustic map generation transforms the points and illumination direction $\vec{L}$ to the coordinate system of the environment map.

```
void CausticGenerationVS(
   in  float4 Pos  : POSITION, // modeling space
   in  float3 Norm : NORMAL,   // normal vector
   out float4 hPos : POSITION, // clipping space
   out float3 x    : TEXCOORD1,// env. space
   out float3 N    : TEXCOORD2,// normal in env.
   out float3 L    : TEXCOORD3)// light in env.
{
   hPos = mul(Pos, WorldLightProj);
   x    = mul(Pos, World) - refpoint;
   N    = mul(Norm, WorldIT);
   L    = x - LightPos; // light dir
}
```

Then the pixel shader computes the location of the photon hit and puts it into the target pixel. For the sake of simplicity, we did not include the computation of the Fresnel attenuation in the following pixel shader code:

```
float       power;  // power of a single photon
samplerCUBE uvmap;  // texcoords of visible points
samplerCUBE envmap; // distance impostor

float4 CausticGenerationPS(
   float3 x : TEXCOORD1,// env. space
   float3 N : TEXCOORD2,// normal in env.
   float3 L : TEXCOORD3 // light in env.
) : COLOR
{
   N = normalize(N);
   L = normalize(L);
   R = refract(L, N, 1/n);    // or reflect ...
   float3 l = Hit(x, R, envmap); // photon hit
      // get the texture coord of point l
   float3 hituv = texCUBE(uvmap, l).xyz;
      // Fresnel
   float F = Fp + pow(1-dot(N, -L), 5) * (1-Fp);
   return float4(hituv, (1-F)*power); // to caustmap
}
```

In order to recognize those texels of the caustic map where the refractor is not visible, we initialize the caustic map with -1 alpha values. Checking the sign of the alpha later, we can decide whether or not it is a photon hit.

The generated caustic map is used to project caustic textures onto surfaces [GD01], or to modify their light map in the next rendering pass. Every photon hit should be multiplied by the BRDF, and the product is used to modulate a small filter texture, which is added to the texture of the surface. The filter texture corresponds to Gaussian filtering in texture space. In this pass we render as many small quadrilaterals (two adjacent triangles in DirectX) or point sprites as texels the caustic map has (figure 38).

The caustic map texels are addressed one by one with variable `caustCoord` in the vertex shader shown below. The center of these quadrilaterals is the origo, and their size depends on the support of the Gaussian filter. The vertex shader changes the coordinates of the quadrilateral vertices and centers the quadrilateral at the u, v coordinates of the photon hit in texture space if the alpha value of the caustic map texel addressed by `caustCoord` is positive, and moves the quadrilateral out of the clipping region if the alpha is negative. This approach requires the texture memory storing the caustic map to be fed back to the vertex shader, which is possible on 3.0 compatible vertex shaders.

The vertex shader of projecting caustic textures onto surfaces is as follows:

```
void CausticRenderVS(
   in  float4 Pos        : POSITION,
   in  float2 caustCoord : TEXCOORD0,
   out float4 hPos       : POSITION,
   out float2 filtCoord  : TEXCOORD1,
   out float  Power      : TEXCOORD2,
   out float4 Tex        : TEXCOORD3)
{
    // Photon position in texture space
   float4 ph = tex2Dlod(caustmap, IN.caustCoord);
   filtCoord = Pos.xy; // filter coords
    // Place quad vertices in texture space
   Tex.x = ph.x + Pos.x / 2;
   Tex.y = ph.y - Pos.y / 2;
    // Transforms to clipping space
   hPos.x = ph.x * 2 - 1 + Pos.x + HALF;
   hPos.y = 1 - ph.y * 2 + Pos.y - HALF;
    // Is it a real Hit?
   if (ph.a > 0 ) hPos.z = 0; // valid
   else           hPos.z = 2; // ignore
   hPos.w = 1;
    // pass power
   Power = ph.a;
}
```

Note that the original x, y coordinates of quadrilateral vertices are copied as filter texture coordinates, and are also moved to the position of the photon hit in the texture space of the surface. The output position register (`hPos`) also stores the texture coordinates converted from $[0,1]^2$ to $[-1,1]^2$ which corresponds to rendering to this space. The `w` and `z` coordinates of

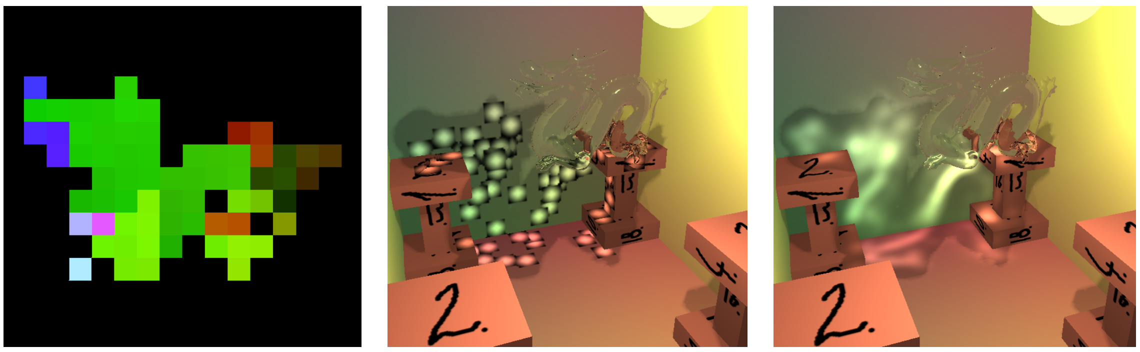


Figure 38: *A photon map, a room without blending, and a room with blending enabled.*

the position register are used to ignore those caustic map elements which have no associated photon hit.

The pixel shader computes the color contribution as the product of the photon power, filter value and the BRDF:

```
float4 CausticRenderPS(
   float2 filtCoord : TEXCOORD1,
   float  Power     : TEXCOORD2,
   float4 Tex       : TEXCOORD3 ) : COLOR
{
   float4 brdf = tex2D(brdfmap, TexCoord);
   float4 w = tex2D(filter, filtCoord );
   return power * w * brdf;
}
```

The target of this rendering is the light map or the modified texture map. Note that the contribution of different photons should be added, thus we should set the blending mode to "add" before executing this phase. Note that it is not obligatory to render the caustics directly to a light map. Instead we can generate caustic patterns in a shadow map or a cube map, and project these patterns on the surfaces similarly to shadow mapping algorithms [SP05]. Note also that image space filtering is only an approximation, but since the billboards are usually small, this approximation is generally used in not real-time photon mapping algorithms as well.

Figure 39 shows the implementation of the caustics generation, when a 64×64 resolution caustic map is obtained in each frame, which is fed back to the vertex shader. Note that even with shadow, reflection, and refraction computation, the method runs with 182 FPS.

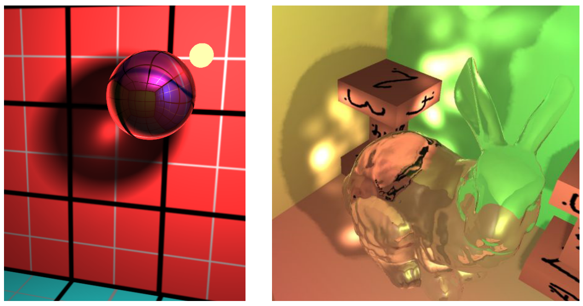


Figure 39: *Real-time caustics caused by a glass objects ($n = 1.3$), rendered by the proposed method*

5.5. Combining different specular effects

The different techniques based on approximate ray-tracing can be combined in a complete real-time rendering algorithm. The input of this algorithm include

- the definition of the static environment in form of triangle meshes, material data, textures, and light maps having been obtained with a global illumination algorithm,
- refractor maps of those dynamic objects which are expected to refract (or reflect) the light multiple times,
- the definition of dynamic objects set in the actual frame,
- the current position of light sources and the eye.

The image generation requires a preparation phase, and a rendering phase from the eye. The preparation phase computes the distance impostors maps. Depending on the distribution of the dynamic objects, we may generate only a single cube map for all of them, or we may maintain a separate map for each of them. Note that this preparation phase is usually not executed in each frame, only if the object movements are large enough. If we update these maps after every 100 frames, then the cost amortizes and the slow down becomes negligible. If the scene has caustic generators, then a caustic map is obtained for each of them, and caustic maps are converted to light maps during the preparation phase.

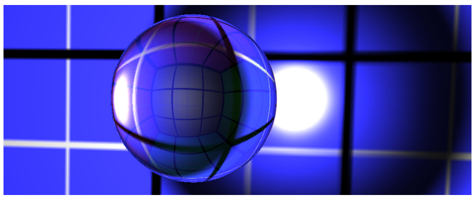


Figure 40: *Caustics seen through the refractor object.*

The final rendering phase from the eye position consists of three steps. First the static environment is rendered with their light maps making also caustics visible. Then dynamic objects are sent to the GPU, having enabled the proposed localized environment mapping and also multiple refraction computation. Note that in this way the reflection or refraction of caustics can also be generated (figure 40).

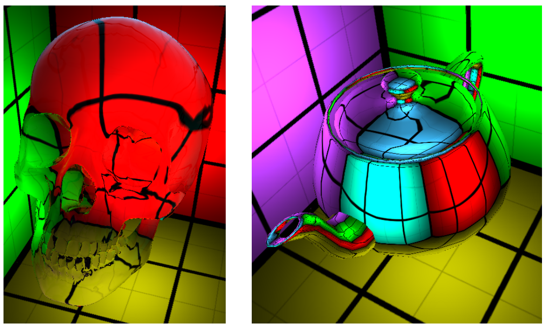


Figure 41: *Left: a glass skull ($n = 1.3$, $k = 0$) of 61000 triangles rendered on 130 FPS. Right: an alu teapot ($n = 0.5..2.3$, $k = 5..9$) of 2300 triangles rendered on 440 FPS.*

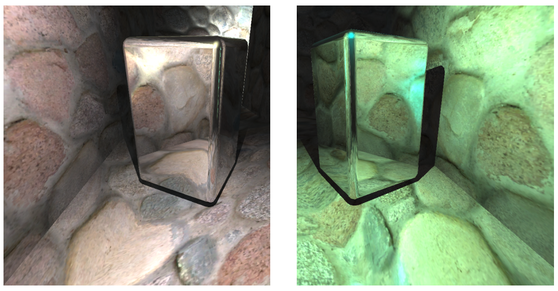


Figure 42: *A reflective box with shadows in a stone environment illuminated by dynamic lights* (160 *FPS*)

The presented algorithm has been implemented in Direct3D environment, first as a test application, and then in a game. The images of the test application are shown by figures 33, 34, 36, 39, 40, and 41. The test application computes the environment map only once to show that the proposed localization gives good results even if the objects moved significantly from their original positions. Note that this application runs typically with few hundred frames per second even in full screen mode on an NV6800GT graphics card and P4/3GHz CPU, and can maintain this speed for tens of thousand of triangles. For comparison, the peak performance is 1095 FPS for the scene of figure 33 when the pixel shader executes a return instruction for the sphere.

We also included the proposed method in a game executing shadow computation, collision detection, etc. (figures 42, 43, and 44) that can run with about a hundred FPS. In this game we used $6 \times 256 \times 256$ resolution cube maps that are recomputed in every 150 msec. We have realized that the speed improves by an additional 20 percent if the distance values are separated from the color data and stored in another texture map. The reason of this behavior is that the pixel shader reads the distance values several times from different texels before the color value is fetched, and separating the distance values increases texture cache utilization.

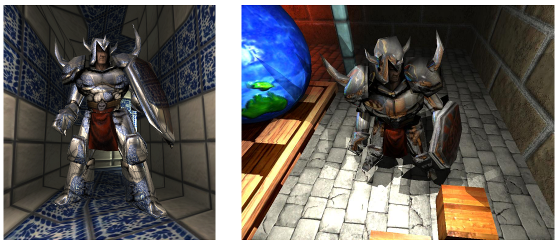


Figure 43: *A knight in reflective armor in textured environment illuminated by dynamic lights (130 FPS).*

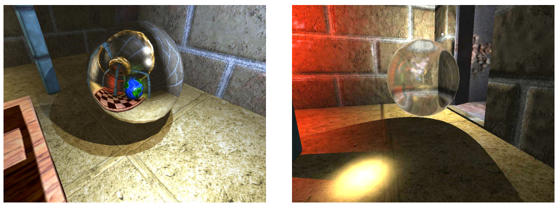


Figure 44: *Reflective and refractive spheres in a game environment (105 FPS)*

6. Diffuse/glossy indirect illumination computation

In case of diffuse/glossy surfaces a path may be continued at infinitely many directions (figure 45), thus even a single pixel center may correspond to infinitely many paths. Unlike in case of specular effects, the identification of paths providing dense enough sampling is not possible in a single step. Instead these algorithms should be decomposed into multiple passes where each pass advances just certain subset of paths.

Light paths can be built from the eye [Kaj86] by *gathering* the illumination, from the light sources [DLW93] by *shooting* the power of the lights, or even starting paths simultaneously from the eye and from the light sources and connecting them [LW93, JC95]. A particularly efficient but simple combination of shooting and gathering computes multiple bounces by shooting and then a single bounce by gathering. Such algorithms distribute the power of the light sources in the scene during shooting steps providing similar error level everywhere in the scene. Then, the points visible from the camera are identified, and the radiance is obtained at these points as the reflection of the radiance of those points that are visible from here. Such a view dependent refinement is called *final gathering.* Note that we could use the radiance of the shooting step directly when a point visible from the camera is identified. However, final gathering significantly reduces the error of the image, since it reiterates the computation once more only for those points that are visible from the camera. Monte Carlo random walk algorithms usually find light paths sequentially sending rays from the last visited point. However, this approach is not GPU friendly since using rasterization the GPU can efficiently identify a set of points visible from a single point, thus it is better to advance a path into many directions in a single step. Rasterization corresponds to tracing a bundle of rays that are either parallel or sharing the same origin. Thus it is worth worth organizing rays to be traced at once as *ray bundles* [SKP98, CHL04, Hac04, BSKS05, Hac05].

GPU native local illumination algorithms compute a single light bounce. Since global illumination requires multiple bounces, the results of a single bounce should be stored and whole computation be repeated. The main problem is the temporary storage of the illumination information, which requires data structures to be stored in the GPU.

A *color texture* of the surface can be interpreted as the radiance function sampled at points corresponding to the texel centers [BGZ97, NC02]. An elementary surface area $A^{(i)}$ is the surface which is mapped onto texel i. The radiance is valid in all outgoing directions for diffuse surfaces, or can be considered as sampled

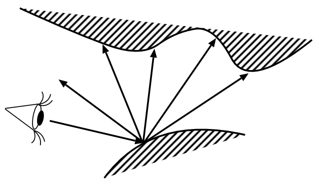


Figure 45: *At diffuse/glossy surfaces light paths split*

at a single direction per each point if the surfaces are non-diffuse [BSKS05]. Alternatively, the texture can be viewed as the finite element representation of the radiance function using piece-wise constant or linear basis functions depending on whether bi-linear texture filtering is enabled or not. Such radiance representation is ideal for obtaining the image from a particular camera since the texturing algorithm of the graphics hardware automatically executes this task.

In addition to textures, illumination information can be stored with the vertices, or even independently of the scene geometry. For example, Greger et al. [GSHG98] calculated and stored the direction dependent illumination in the vertices of a *bi-level grid* subdividing the object scene. During run-time, irradiance values of an arbitrary point are calculated by tri-linearly interpolating the values obtained from the neighboring grid vertices. While Greger et al. used a precomputed radiosity solution to initialize the data structures, Mantiuk et al. [MPM02] calculated these values during run-time using an iterative algorithm that simulates the multiple bounces of light.

A *shadow map* is the sampled representation of those points that are visible from the light source, thus a shadow map can mediate the first bounce illumination to obtain two-bounce reflection at the surfaces of the scene [DS03][DS05].

Finally, a *cube map* is also a representation of the incoming illumination for a point (the cube map center), or approximately for a set of points that are not very far from the center.

In this section we examine two techniques in detail, a GPU based stochastic radiosity algorithm [BSKS05], which uses color texture radiance representation, and a cube map based final gathering method [SKL06], which also stores distance information in texels, which allows the application of the stored illumination information for a wider set of points.

6.1. Radiosity on the GPU

The iterative radiosity method is one of the oldest global illumination techniques, and is also a pioneer of the exploitation of the graphics hardware [CG85].

It has been recognized for a long time that the form factors belonging to a single point can be obtained by rendering the scene from this point, for which the graphics hardware is an ideal tool. In the era of non-programable GPUs the result of the rendering, i.e. the image, was read back to the CPU where the form factor computation and the radiosity transfer took place.

The programmability of GPUs, however, offered new possibilities for radiosity as well. Now it is feasible to execute the complete algorithm on the GPU, without expensive image read backs.

Let us revisit the reflected radiance formula (equation 1) for diffuse surfaces, but now express it as an integral over the surfaces, and not over the illumination directions:

$$L^r(\vec{x}) = \mathcal{T}_{f_r} L = \int_S L(\vec{y}) \cdot f_r(\vec{x}) \cdot G(\vec{x}, \vec{y}) \; dy, \qquad (10)$$

where S is the set of surface points, f_r is the BRDF and

$$G(\vec{x}, \vec{y}) = v(\vec{x}, \vec{y}) \cdot \frac{\cos^+ \theta_{\vec{x}} \cdot \cos^+ \theta_{\vec{y}}}{|\vec{x} - \vec{y}|^2}$$

is the *geometric factor*, where $v(\vec{x}, \vec{y})$ is the mutual visibility indicator which is 1 if points $\vec{x}$ and $\vec{y}$ are visible from each other and zero otherwise, $\theta_{\vec{x}}$ and $\theta_{\vec{y}}$ are the angles between the surface normals and direction $\omega_{\vec{y} \to \vec{x}}$ that is between $\vec{x}$ and $\vec{y}$.

If the emission radiance is L^e the first bounce illumination is $\mathcal{T}_{f_r} L^e$, the second bounce illumination is $\mathcal{T}_{f_r} \mathcal{T}_{f_r} L^e$, etc. The full global illumination solution is the sum of all bounces. *Iteration* techniques take some initial reflected radiance function, e.g. $L^r_0 = 0$, and keep refining it by computing the reflection of the sum of the emission and the previous reflected radiance estimate. Formally, the global illumination solution which includes paths of all lengths is the limiting value of the following iteration scheme:

$$L^r_m = \mathcal{T}_{f_r} L_{m-1} = \mathcal{T}_{f_r}(L^e + L^r_{m-1}).$$

Each iteration step adds the next bounce in the reflected radiance. Iteration works with the complete radiance function, whose temporary version should be represented somehow. The classical approach is the *finite-element method*, which approximates the radiance function in a function series form. In the simplest diffuse case we decompose the surface to small elementary surfaces $A^{(1)}, \ldots, A^{(n)}$ and apply a piecewise constant approximation, thus the reflected radiance function is represented by the reflected radiance of these patches, that is by $L^{r,(1)}, \ldots, L^{r,(n)}$. CPU radiosity algorithms usually decompose general surfaces to triangular patches. However, in GPU approaches this is not feasible since the GPU processes patches independently thus the computation of the interdependence of patch data is difficult. Instead, the radiance function can be stored in a texture, thus the elementary surfaces will correspond to different texel areas. Texture based radiance representation is practically independent of the patch decomposition, surfaces describe only the geometry. We may call this approach as *patchless rendering*, since having detected the visibility between two points, all computations are done on texels independently of the surfaces. It allows us to work with the original geometry, no patch subdivision is necessary.

If the elementary surfaces are small, we can consider just a single point of them in the algorithms, while assuming that the properties of other surface points are similar. Surface properties, such as the BRDF and the emission can be given by values $f_r^{(1)}, \ldots, f_r^{(n)}$, and $L^{e,(1)}, \ldots, L^{e,(n)}$, respectively, in each finite element, i.e. in each texel. In case of small elementary surfaces we can check the mutual visibility of two elementary surfaces by inspecting only their centers. In this case the update of the *average* reflected radiance corresponding to texel i in a single iteration can be approximated in the following way:

$$L^{r,(i)}_m = \frac{1}{A^{(i)}} \cdot \int_{A^{(i)}} L^r_m(\vec{x}) \; dx =$$

$$\frac{1}{A^{(i)}} \cdot \int_{A^{(i)}} \int_S (L^e(\vec{y}) + L^r_{m-1}(\vec{y})) \cdot f_r^{(i)} \cdot G(\vec{x}, \vec{y}) \; dydx \approx$$

$$\sum_{j=1}^{n} (L^{e,(j)} + L^{r,(j)}_{m-1}) \cdot f_r^{(i)} \cdot G(\vec{y}_j, \vec{x}_i) \cdot A^{(j)}, \qquad (11)$$

where $\vec{y}_j$ and $\vec{x}_i$ are the centers of elementary surfaces $A^{(j)}$ and $A^{(i)}$ belonging to texels j and i, respectively.

Iteration simultaneously computes the interaction between all surface elements, which has quadratic complexity in terms of the number of finite elements, and its GPU implementation turned out not to be superior than the CPU version [CHH03]. The complexity of a single iteration step can be attacked by special iteration techniques, such as Southwell iteration (also called *progressive radiosity*) [WEH89], hierarchical radiosity [HSA91, AH93, Bek99], or by randomization [Shi91, Neu95]. Southwell iteration computes the interaction of the element having the highest unshot radiosity and all other surface elements [CCWG88]. It is quite simple to implement on the GPU [CHL04], but has also quadratic complexity [SKM95], which limits these approaches for simple models and low texture resolutions.

Monte Carlo techniques, on the other hand, have sub-quadratic complexity, and can greatly benefit from the hardware. The method discussed here is based on the *stochastic iteration* [SK99b, Neu95, Sbe96, Bek99]. Stochastic iteration means that in the iteration scheme a random transport operator $\mathcal{T}^*_{f_r}$ is used instead of the light-transport operator $\mathcal{T}_{f_r}$. The random transport operator has to give back the light-transport operator in the expected case:

$$L^r_m = \mathcal{T}^*_{f_r}(L^e + L^r_{m-1}), \qquad E[\mathcal{T}^*_{f_r} L] = \mathcal{T}_{f_r} L.$$

Note that such an iteration scheme does not converge, but the iterated values will fluctuate around the real solution. To make the sequence converge, we compute the final result as the average of the estimates of subsequent iteration steps:

$$L^r = \frac{1}{m} \cdot \sum_{k=1}^{m} L^r_k.$$

The core of all stochastic iteration algorithms is the definition of the random transport operator. We prefer those random operators that can be efficiently computed on the GPU and introduce small variance. Note that randomization gives us a lot of freedom to define an elementary step of the iteration. This is very important for GPU implementation since we can use iteration steps that fit to the features of the GPU.

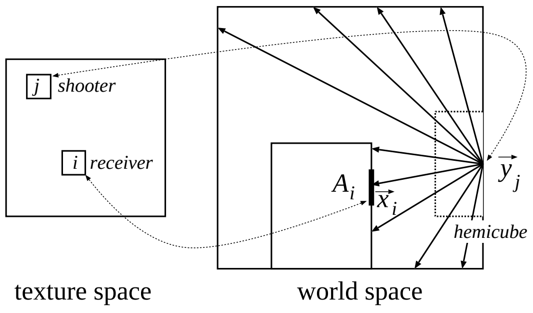


Figure 46: *A single iteration step*

One approach meeting these requirements is the *random hemicube shooting* (perspective ray bundles) [SKAB03], which selects a patch (i.e. a texel) randomly, assumes that the total power of the scene is concentrated here, and this power is shot toward other surfaces visible from here. Note that this elementary step is similar to that of progressive radiosity. However, while progressive radiosity selects that patch which has the highest unshot radiosity, and shoots this unshot radiosity, stochastic hemicube shooting finds a patch randomly, and shoots the total power of the scene.

Note that the proposed scheme is equivalent to the Monte Carlo evaluation of the sum of equation 11. A surface element $A^{(j)}$ is selected with probability p_j, and $\vec{y}_j$ is set to its center. This randomization allows to compute the interaction between shooter surface element $A^{(j)}$ and all other receiver surface elements $A^{(i)}$, instead of considering all shooters and receivers simultaneously. Having selected shooter $A^{(j)}$ and its center point $\vec{y}_j$, the radiance of this point is sent to all those surface elements that are visible from here. The Monte Carlo estimate of the reflected radiance of finite element $A^{(i)}$ after this transfer is

$$\hat{L}^{r,(i)}_m = \frac{L^{(j)}_{m-1} \cdot f^{(i)}_r \cdot G^{(i)}_m}{p_j}. \tag{12}$$

In order to realize this random transport operator, two tasks need to be solved, including the random selection of a texel identifying point $\vec{y}_j$, and the update of the radiance at those texels which correspond to points $\vec{x}_i$ visible from $\vec{y}_j$.

Random texel selection

The randomization of the iteration introduces some variance in each step, which should be minimized in order to get an accurate result quickly. According to importance sampling, the introduced variance can be reduced with a selection probability that is proportional to the integrand or to the sampled term of the sum of equation 11. Unfortunately, this is just approximately possible, and the selection probability is set proportional to the current power of the selected texel. If the light is transferred on several wavelengths simultaneously, the luminance of the texel should be used.

The random selection proportional to the luminance can be supported by a *mipmapping* scheme. Mipmapping has originally been proposed for texture filtering. Later it was also used to find the maximum value in an image [CHL04]. In this approach, however, we use mipmapping to sample randomly, proportional to the stored luminance value. A mipmap can be imagined as a quadtree, which allows the selection of a texel in $\log_2 \mathcal{R}$ steps, where $\mathcal{R}$ is the resolution of the texture. Each texel is the sum of the luminance of four corresponding texels on a lower level. The top level of this hierarchy has only one texel, which contains the average of the luminance of all elementary texels. Both generation and sampling require the rendering of a textured rectangle (also called a full screen quad) by $\log_2 R$ times.

The generated mipmap is used to sample a texel with a probability that is proportional to its luminance. First the luminance of the top level texel is retrieved from a texture and is multiplied by a random number uniformly distributed in the unit interval.

Then the next mipmap level is retrieved, and the four texels corresponding to the upper level texel is obtained. The coordinates of the bottom right texel are passed in `uv`, and the coordinates of the three other texels are computed by adding parameter `rr` that is the distance between two neighboring texels. The luminance of the four texels are summed until the running sum (denoted by `cmax` in the program below) gets greater than selection value `r` obtained on the higher level. When the running sum gets larger than the selection value from the higher level, the summing is stopped and the actual texel is selected. A new selection value is obtained as the difference of the previous value and the luminance of all texels before the found texel (`r-cmin`). The new selection value and the texture coordinates of the selected texel are returned in `c`. Then the same procedure is repeated in the next pass on the lower mipmap levels. This procedure terminates at a leaf texel with a probability that is proportional to its luminance.

The shader of a pass of the mipmap based sampling selects according to random value `r` passed from the upper level and originally set randomly with uniform distribution between zero and the total luminance of the scene:

```
float cmin = 0;
float cmax = tex2D(texture, uv);  // texel 1
if(cmax >= r) {
   c = float3(r-cmin, uv.x, uv.y);
} else {
   cmin = cmax;
   float2 uv1 = float2(uv.x-rr, uv.y);
   cmax += tex2D(texture, uv1);   // texel 2
   if(cmax >= r) {
      c = float3(r-cmin, uv1.x, uv1.y);
   } else {
      cmin = cmax;
      uv1 = float2(uv.x, uv.y-rr);
      cmax += tex2D(texture, uv1);// texel 3
      if(cmax >= r) {
         c = float3(r-cmin, uv1.x, uv1.y);
      } else {                    // texel 4
         cmin = cmax;
         uv1 = float2(uv.x-rr, uv.y-rr);
         c = float3(r-cmin, uv1.x, uv1.y);
      }
   }
}
return c; // r and uv on the next level
```

Update of the radiance texture

The points visible from selected shooter $\vec{y}$ can be found by placing a hemicube around $\vec{y}$, and then using the z-buffer algorithm to identify the visible patches. Since it turns out just at the end, i.e. having processed all patches by the z-buffer algorithm, which points are really visible, the application of the random transfer operator requires two passes.

The center and the base of the hemicube are set to $\vec{y}$ and to the surface at $\vec{y}$, respectively. Taking the hemicube center and side faces as the eye and window, respectively, the scene is rendered five times, also computing the z coordinate of the visible points. These values are written into a texture, called the *depth map*.

We also tried the application of hemispherical mapping [CHL04], which would require just a single rendering pass, but we gave it up because the distortion of hemispherical projection introduced many artifacts. The problem is that hemispherical projection is non-linear, and maps triangles to regions bounded by ellipses. However, the current graphics hardware always assumes that the points leaving the vertex shader are triangle vertices, and generates those pixels that are inside these triangles. This is acceptable only if the triangles are very small, i.e. when the distortion caused by the hemispherical mapping is negligible. Recall that in our "patchless" rendering approach, no tessellation of the geometry is required, thus hemispherical projection is not feasible.

In the second pass we render into the rectangle of the radiance texture. It means that the pixel shader visits each texel, and updates the stored actual radiance. The vertex shader is set to map a point of the unit texture square onto the the full screen. Normalized device space coordinate `hpos` (which is identical to clipping space coordinate since the fourth homogeneous coordinate is set to 1) is computed from texture coordinate `texx` as follows:

```
hpos.x = 2 * texx.x - 1;
hpos.y = 1 - 2 * texx.y;
hpos.z   = 0;
hpos.w   = 1;
```

This transformation between texture and normalized device space coordinates is necessary because the device space coordinates must be in $[-1, 1]$, while the texture coordinates are expected in $[0, 1]$. Direct3D has a texture space which defines the upper left corner as $(0, 0)$ and the lower right corner as $(1, 1)$, so we need to flip y coordinates.

The vertex shader also transforms input vertex `Pos` to camera space (`x`), as well as its normal vector `xnorm` to compute radiance transfer, determines homogeneous coordinates `vch` for the location of the point in the depth map.

```
x     = mul(Pos, WorldView).xyz;
xnorm = mul(xnorm, WorldViewIT).xyz;
vch   = mul(Pos, WorldViewProj);
```

The geometric factor depends on the receiver point,

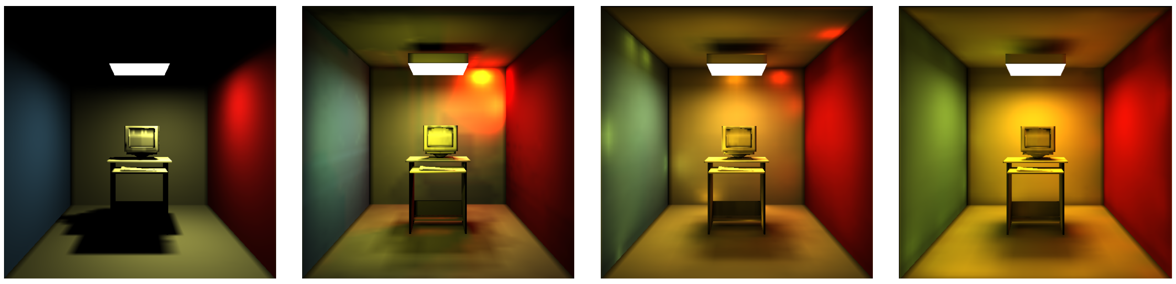


Figure 47: *Images rendered with stochastic hemicube shooting. All objects are mapped to a single texture map of resolution* 128×128*, which corresponds to processing 16 thousand patches.*

thus its accurate evaluation could be implemented by the fragment shader:

```
float3 ytox = normalize(x); // dir y to x
float  xydist2 = dot(x, x); // |x - y|^2
float  cthetax = dot(xnorm, -ytox);
if (cthetax < 0) costhetax = 0;
float3 ynorm = float3(0, 0, 1);
float  cthetay = ytox.z;
if (cthetay < 0) costhetay = 0;
float G = cthetax * cthetay / xydist2;
```

Note that we took advantage of the fact that $\vec{y}$ is the eye position of the camera, which is transformed to the origo by the `WorldView` transform, and the normal vector at this point is transformed to axis z.

When a texel of the radiance map is shaded, it is checked whether or not the center of the surface corresponding this texel is visible from the shooter by comparing the depth values stored in the visibility map. The pixel shader code responsible for converting homogeneous coordinates (`vch`) to Cartesian coordinates (`vcc`) and computing the visibility indicator is:

```
float3 vcc = vch.xyz / vch.w; // Cartesian
vcc.x = (vcc.x + 1) / 2;      // Texture space
vcc.y = (1 - vcc.y) / 2;
float depth = tex2D(depthmap, vcc).r;
float vis = (abs(depth - vcc.z) < EPS);
```

Instead of coding these steps manually, we could use projective depth texturing as well as discussed in the section on *Shadow mapping*.

To obtain the radiance transfer from shooter $\vec{y}$ to the processed point $\vec{x}$, first the radiance of shooter $\vec{y}$ is calculated from its reflected radiance stored in `radmap`, and its emission stored in `emissmap`. Shooter's texture coordinates `texy` are passed as uniform parameters:

```
float3 Iy = tex2D(radmap, texy);
float3 Ey = tex2D(emissmap, texy);
float3 Ly = Ey + Iy;
```

The new reflected radiance at $\vec{x}$ is obtained from the radiance at $\vec{y}$ multiplying it with visibility indicator `vis` and geometric factor `G` computed before, and divided by shooter selection probability `p` passed as a uniform parameter. The emission and the surface area of this texel are read from texture map `emissmap`. Texel luminance (`lum`) at $\vec{x}$ is also computed to allow importance sampling in the subsequent iteration step, and stored in the alpha channel of the reflected radiance.

```
float4 brdfx = tex2D(brdfmap, texx);
float3 Lx = Ly * G * vis * brdfx / p;
float4 Ex = tex2D(emissmap, texx);
float  Ax = Ex.a;            // surface area
  // compute the luminance
float3 em  = float3(0.21, 0.39, 0.4);
float  lum = dot(E, em) + dot(Lx, em);
return float4(Lx, lum);
```

The implementation has been tested with a room scene of figure 47, and we concluded that a single iteration requires less than 20 msec for a few hundred vertices and for 128×128 resolution radiance maps, while keeping the depth maps at 256×256 (the algorithm is pixel shader limited). Using 64×64 resolution radiance maps introduced a minor amount of shadow bleeding, but increased iteration speed by approximately 40%. Since we can expect converged images after 40 – 80 iterations for normal scenes, this corresponds to 0.5 – 1 frames per second, without exploiting frame-to-frame coherence. In order to eliminate flickering, we should use the same random number generator in all frames. On the other hand, as in all iterative approaches, frame to frame coherence can be easily exploited. In case of moving objects, on the other hand, we can take the previous solution as a good guess to start the iteration. This trick not only improves accuracy, but also makes the error of subsequent steps highly correlated, which also helps eliminating flickering.

6.2. Final gathering of diffuse/glossy indirect illumination with cube maps

A cube map can represent the incoming illumination of a single point in all directions, thus is a primary candidate for storing illumination [SKL06]. However, the problem is that a single cube map is good only for a single point, and the application of many cube maps is limited by memory constraints. To solve this problem, we can store geometric information in cube map texels, for example, the distance between the cube map center and the point visible in a given texel. Then when the illumination of a point not being in the cube map center is computed, this geometric information is used to correct the incoming radiance of the cube map texels.

Let us assume that we use a single cube map that was rendered from reference point $\vec{o}$. Our goal is to reuse this illumination information for other nearby points as well. To do so, we apply approximations that allow us to factor out those components from the reflected radiance formula (equation 1) which strongly depend on shaded point $\vec{x}$.

In order to estimate the reflected radiance integral, directional domain Ω' is partitioned to solid angles $\Delta\omega'_i, i = 1, \ldots, N$, where the radiance is roughly uniform in each domain. After partitioning, the reflected radiance is expressed by the following sum:

$$L^r(\vec{x}, \vec{\omega}) = \sum_{i=1}^{N} \int_{\Delta\omega'_i} L(\vec{y}, \vec{\omega}') \cdot f_r(\vec{\omega}', \vec{x}, \vec{\omega}) \cdot \cos^+ \theta'_{\vec{x}} \, d\omega'.$$

Let us consider a single term of this sum representing the radiance reflected from $\Delta\omega'_i$. If $\Delta\omega'_i$ is small, then we can use the following approximation:

$$L^r(\vec{x}, \vec{\omega}) \approx \tilde{L}^{in}(\Delta y_i) \cdot \int_{\Delta\omega'_i} f_r(\vec{\omega}', \vec{x}, \vec{\omega}) \cdot \cos\theta'_{\vec{x}} \, d\omega' \quad (13)$$

where $\tilde{L}^{in}(\Delta y_i)$ is the *average incoming radiance* from surface Δy_i seen at solid angle $\Delta\omega'_i$. Expressing average incoming radiance $\tilde{L}^{in}(\Delta y_i)$ on surface area Δy_i, we obtain:

$$\tilde{L}^{in}(\Delta y_i) = \frac{1}{\Delta y_i} \cdot \int_{\Delta y_i} L(\vec{y}, \vec{\omega}_{\vec{y}\to\vec{x}}) \, dy.$$

Note that this average is independent of shaded point $\vec{x}$ if the environment is diffuse, and can be supposed to be approximately independent of the shaded point if the environment is moderately glossy.

The second factor of the product in equation 13 is the *reflectivity* integral, which is also expressed as the product of the average integrand and the size of the integration domain:

$$\int_{\Delta\omega'_i} f_r(\vec{\omega}', \vec{x}, \vec{\omega}) \cdot \cos^+ \theta'_{\vec{x}} \, d\omega' = a(\Delta\omega'_i \to \vec{x} \to \vec{\omega}) \cdot \Delta\omega'_i$$

where $a(\Delta\omega'_i \to \vec{x} \to \vec{\omega})$ is the *average reflectivity* from solid angle $\Delta\omega'_i$. The average reflectivity can be either obtained using only one directional sample on the fly, or precomputed and stored in a texture addressed by the angle of direction and the size of solid angle where averaging happens.

Putting the results of the average incoming radiance and the reflectivity formula together, the reflected radiance can be approximately expressed as

$$L^r(\vec{x}, \vec{\omega}) \approx \sum_{i=1}^{N} \tilde{L}^{in}(\Delta y_i) \cdot a(\Delta\omega'_i \to \vec{x} \to \vec{\omega}) \cdot \Delta\omega'_i. \quad (14)$$

Average incoming radiance values $\tilde{L}^{in}(\Delta y_i)$ are independent of the shaded point in case of diffuse or moderately glossy environment, thus these values can potentially be reused for all shaded points. To exploit this idea, visible surface areas Δy_i need to be identified and their average radiances need to be computed first. These areas are found and the averaging is computed with the help of a cube map placed at reference point $\vec{o}$ in the vicinity of the shaded object.We render the scene from reference point $\vec{o}$ onto the six sides of a cube. In each pixel of these images we store the radiance of the visible point and also the distance from the reference point. The pixels of the cube map thus store the radiance and also encode the position of small indirect lights (figure 48).

The small virtual lights are clustered into larger area light sources while averaging their radiance, which corresponds to downsampling the cube map. A pixel of the lower resolution cube map is computed as the average of the included higher resolution pixels. Note that both radiance and distance values are averaged, thus finally we have larger lights having the average radiance of the small lights and placed at their average position. The total area corresponding to a pixel of a lower resolution cube map will be elementary surface Δy_i, and its average radiance is stored in the texel.

The solid angle subtended by a cube map lexel can be approximated by the formula of the solid angle of a circle. If a circle of area A is perpendicular at its center to the viewing direction, and is at distance d, then it subtends solid angle:

$$\Delta\omega = 2\pi \cdot \left(1 - \frac{1}{\sqrt{1 + \frac{A}{d^2\pi}}}\right). \quad (15)$$

When the surface is not perpendicular, its area should

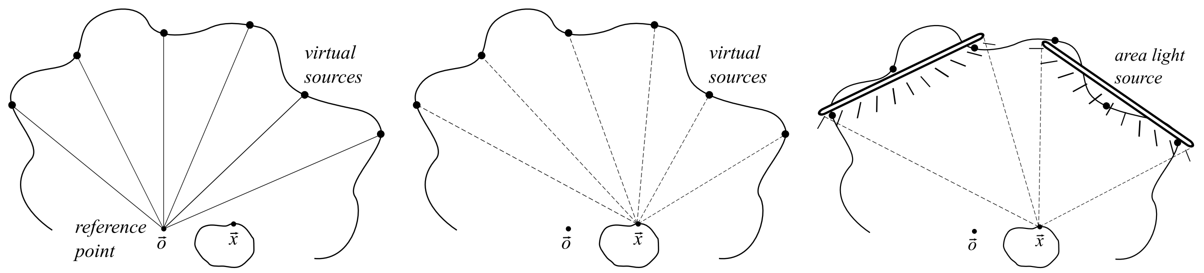


Figure 48: *The basic idea of diffuse/glossy final gathering: first virtual lights sampled from reference point $\vec{o}$ are identified, then these point lights are grouped into large area lights. At shaded points $\vec{x}$ the illumination of a relatively small number of area lights is computed without visibility tests.*

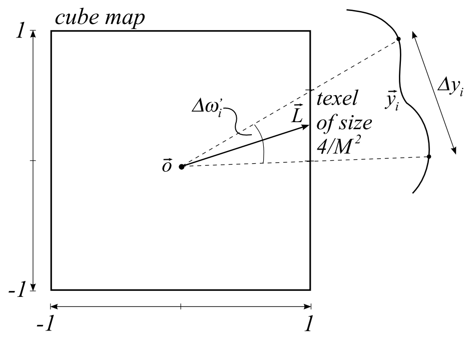


Figure 49: *Solid angle in which a surface seen through a cubemap pixel*

be multiplied with $\cos\theta$ where θ is the angle between the normal vector at the viewing direction.

Supposing that the edge size of the cube map is 2, the area of a lexel is $2/M$ where M is the resolution of a single cubemap face. Let vector $\vec{L}$ be the vector pointing from the center of the cubemap to the texel (figure 49). In this case the distance is $|\vec{L}|$ and $\cos\theta = 1/|\vec{L}|$. Thus the solid angle subtended by a texel is:

$$\Delta\omega_i' \approx 2\pi \cdot \left(1 - \frac{1}{\sqrt{1 + \frac{4}{M^2\pi|\vec{L}|^3}}}\right).$$

According to equation 14 the reflected radiance at the reference point is:

$$L^r(\vec{o},\vec{\omega}) \approx \sum_{i=1}^{N} \tilde{L}^{in}(\Delta y_i) \cdot a(\Delta\omega_i' \to \vec{o} \to \vec{\omega}) \cdot \Delta\omega_i'.$$

Let us now consider another point $\vec{x}$ close to the reference point $\vec{o}$ and evaluate a similar integral for point $\vec{x}$ while making exactly the same assumption on the surface radiance, i.e. it is constant in areas Δy_i:

$$L^r(\vec{x},\vec{\omega}) \approx \sum_{i=1}^{N} \tilde{L}^{in}(\Delta y_i) \cdot a(\Delta\omega_i' \to \vec{x} \to \vec{\omega}) \cdot \Delta\omega_i^*, \tag{16}$$

where $\Delta\omega_i^*$ is the solid angle subtended by Δy_i from $\vec{x}$. Unfortunately, the solid angle values can be obtained directly from the geometry of the cubemap only if the shaded point is the center of the cube map. In case of other shaded points, special considerations are needed that are based on the distances from the environment surface.

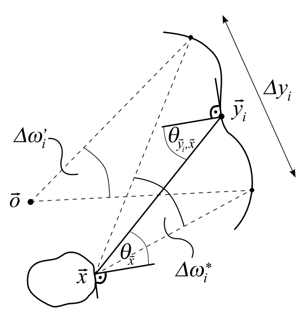


Figure 50: *The notations of the evaluation of subtended solid angles*

Solid angle $\Delta\omega_i^*$ is expressed from $\Delta\omega_i'$ using the formula of the solid angle of a circle (equation 15). Assume that the environment surface is not very close compared to the distances of the reference and shaded points, thus the angles between the normal vector at $\vec{y}_i$ and reflection vectors from $\vec{o}$ and from $\vec{x}$ are similar. In this case, using equation 15, we can establish the following relationship between $\Delta\omega_i^*$ and $\Delta\omega_i'$:

$$\Delta\omega_i^* \approx 2\pi - \frac{2\pi - \Delta\omega_i'}{\sqrt{(2\pi - \Delta\omega_i')^2 \cdot \left(1 - \frac{|\vec{o}-\vec{y}_i|^2}{|\vec{x}-\vec{y}_i|^2}\right) + \frac{|\vec{o}-\vec{y}_i|^2}{|\vec{x}-\vec{y}_i|^2}}}$$

Note that the estimation could be made more accurate if the normal vectors were also stored in cube map texels and the cosine angles were evaluated on the fly.

The proposed algorithm first computes an environment cube map from the reference point and stores the radiance and distance values of the points visible in its pixels. We usually generate $6 \times 256 \times 256$ pixel resolution cube maps. Then the cube map is downsampled to have `M`×`M` pixel resolution faces (`M` is 4 or even 2). Texels of the low-resolution cubemap represent elementary surfaces Δy_i whose average radiance and distance are stored in the texel. The illumination of these elementary surfaces is reused for an arbitrary point `x`, as shown by the following HLSL pixel shader program calculating the reflected radiance at this point:

```
half3 RefRadPS (
   half3 N : TEXCOORD0,    // normal
   half3 V : TEXCOORD1,    // view
   half3 x : TEXCOORD2 ) : COLOR0
{
  half3 Lr = 0;  // reflected radiance
  V = normalize( V );
  N = normalize( N );
  for (int X = 0; X < M; X++) // for each texel
   for (int Y = 0; Y < M; Y++) {
     half2 t = half2((X+0.5f)/M, (Y+0.5f)/M);
     half2 l = 2 * t - 1;  // [0,1]->[-1,1]
     Lr += Cntr(x, half3(l.x,l.y, 1), N, V);
     Lr += Cntr(x, half3(l.x,l.y,-1), N, V);
     Lr += Cntr(x, half3(l.x, 1,l.y), N, V);
     // + similarly for the 3 remaining sides
   }
  return Lr;
}
```

The `Cntr` function calculates the contribution of a single texel of downsampled, low resolution cubemap `LREnvMap` to the illumination of the shaded point. Arguments `x`, `L`, `N`, and `V` are the relative position of the shaded point with respect to the reference point, the unnormalized illumination direction pointing to the center of the texel from the reference point, the unit surface normal at the shaded point, and the unit view direction, respectively.

```
half3 Cntr(half3 x, half3 L, half3 N, half3 V) {
    half  l    = length(L);
    half  dwc  = 1 / sqrt(1 + 4/(M*M*l*l*PI));
    half  doy  = texCUBE(LRCubeMap, L).a;
    half  doy2 = doy * doy;
    half3 y    = L / l * doy;
    half  doy_dxy2 = doy2 / dot(y-x, y-x);
    half  dws  = 2*PI - dwc /
                 sqrt((dwc*dwc*(1-doy_dxy2)+doy_dxy2));
    half3 I = normalize(y - x);
    half3 H = normalize(I + V);
    half3 a = kd * max(dot(N,I),0) +
               ks * pow(max(dot(N,H),0),n);
    half3 Lin  = texCUBE(LRCubeMap, L).rgb;
    return Lin * a * dws;
}
```

First the solid angle subtended by the texel from the reference point is computed and stored in variable `dw`, then illuminating point `y` is obtained looking up the distance value of the cube map. The square distances between the reference point and the illuminating point, and between the shaded point and the illuminating point are put into `doy2` and `dxy2`, respectively. These square distances are used to calculate solid angle `dws` subtended by the illuminating surface from the shaded point. Phong-Blinn BRDF is used with diffuse reflectance `kd`, specular reflectance `ks`, and shininess `n`. Illumination direction `I` and halfway vector `H` are calculated, and the reflection of the radiance stored in the cube map texel is obtained according to equation 16.

In order to demonstrate the results, we took a simple environment consisting of a cubic room with a divider face in it. The object to be indirectly illuminated is the bunny, happy buddha, and the the dragon, respectively. Each of these models consists of approximately 50-60 thousand triangles. Frame rates were measured in 700×700 windowed mode on an NV6800GT graphics card and P4/3GHz CPU. The first set of pictures (Figure 51) shows a diffuse bunny inside the cubic room. The images of the first column are rendered by the traditional environment mapping technique for diffuse materials where a precalculated convolution enables us to determine the irradiance at the reference point with a single lookup. Clearly, these precalculated values cannot deal with the position of the object, thus the bunny looks similar everywhere. The other columns show the results of our method using different sized cube maps. Note that even with extremely low resolution (2×2) we get images similar to the large-resolution reference.

The second set (Figure 52) and the third set (Figure 53) of pictures show a glossy buddha and a dragon inside a room. The first column presents the traditional environment mapping technique while the other three columns present the results of our localized algorithm. Similarly to the diffuse case, even cube map resolution of 2×2 produced more pleasing results than the classical technique. We have also implemented the proposed method in a game running at 30 FPS. Images of this game are shown by figure 54.

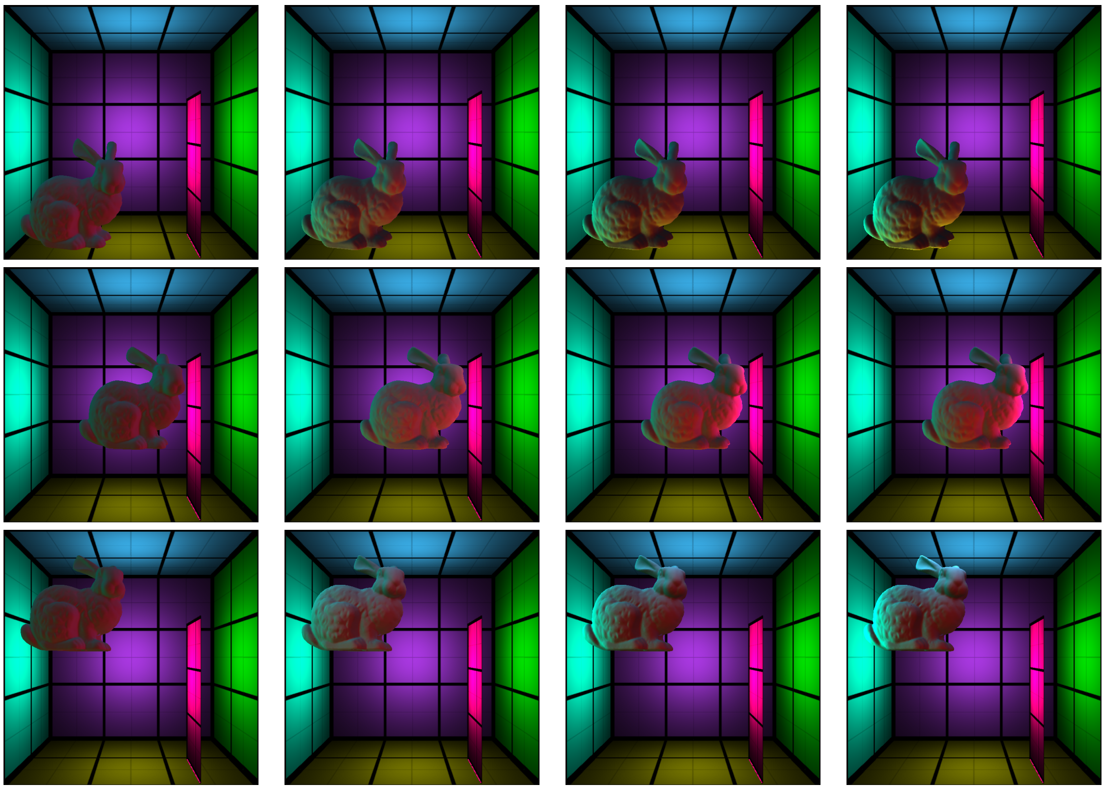


Figure 51: *Diffuse bunny rendered with the classical environment mapping (left column) and with localized cube maps using different environment map resolutions.*

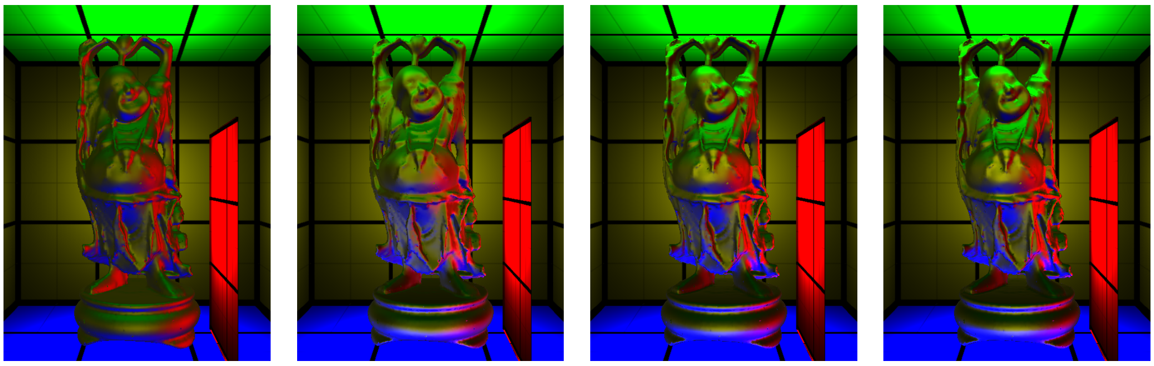


Figure 52: *Specular buddha (the shininess is 5) rendered with the classical environment mapping (left column) and with localized cube maps using different environment map resolutions.*

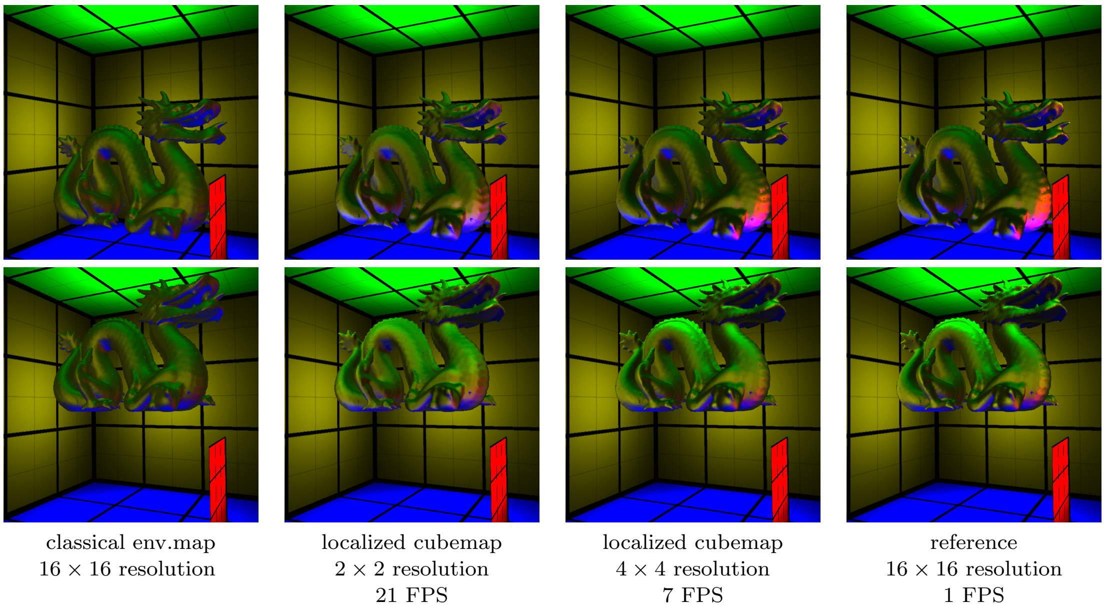


Figure 53: *Specular dragon (the shininess is 5) rendered with the classical environment mapping (left column) and with localized cube map mediators using different cube map resolutions.*

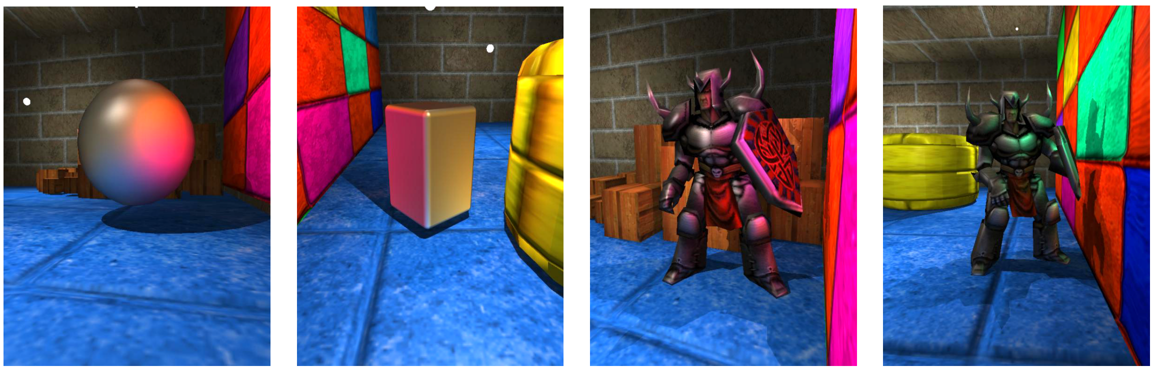


Figure 54: *Glossy objects rendered with cube map mediators in a game running at 30 FPS.*

7. Pre-computation aided global illumination

As stated, global illumination algorithms should integrate the contribution of all light paths connecting the eye and the light sources via scattering points. If the scene is static, that is, its geometry and its material properties do not change in time, then the structure of these light paths remain the same except for the first light and the last viewing rays, which might be modified due to moving lights and camera. Pre-computation aided approaches pre-compute the effects of the static parts of the light paths.

Let us consider a light beam of radiance $L^{in}(\vec{y}, \vec{\omega}_i)$ approaching *entry point* $\vec{y}$ on the object surface from direction $\vec{\omega}_i$ (figure 55). Note that this light beam may be occluded from $\vec{y}$, prohibiting the birth of the path at $\vec{y}$, and making the contribution zero. If the light beam really reaches $\vec{y}$, then it may be reflected, refracted even several times, and finally it may arrive at the eye from *exit point* $\vec{x}$ with viewing direction $\vec{\omega}_o$. Entry and exit points can be connected by infinitely many possible paths, from which a finite number can be generated by Monte Carlo simulation, for example, by path tracing, light tracing, bi-directional path tracing, or by iterative techniques [SK99a, DBB03].

The total contribution of the set of these paths to the visible radiance $L^{out}(\vec{x}, \vec{\omega}_o)$ at $\vec{x}$ is

$$L^{in}(\vec{y}, \vec{\omega}_i) \cdot T(\vec{y}, \vec{\omega}_i \to \vec{x}, \vec{\omega}_o),$$

where $T(\vec{y}, \vec{\omega}_i \to \vec{x}, \vec{\omega}_o)$ is the *transfer function.* The total visible radiance $L^{out}(\vec{x}, \vec{\omega}_o)$ is the sum of the contributions of all paths entering the scene at every possible point and from every possible direction, thus it can be expressed by the following double integral:

$$L^{out}(\vec{x}, \vec{\omega}_o) = \int_S \int_\Omega L^{in}(\vec{y}, \vec{\omega}_i) \cdot T(\vec{y}, \vec{\omega}_i \to \vec{x}, \vec{\omega}_o) \; d\omega_i dy \tag{17}$$

where S is the set of surface points and Ω is the set of directions. In the general case the output radiance is obtained as a double integral of the product of a four-variate and a two-variate functions.

There are a couple of important special cases:

- *Diffuse surfaces* have direction independent outgoing radiance, thus both exit radiance L^{out} and transfer function T become independent of $\vec{\omega}_o$:

$$L^{out}(\vec{x}) = \int_S \int_\Omega L^{in}(\vec{y}, \vec{\omega}_i) \cdot T(\vec{y}, \vec{\omega}_i \to \vec{x}) \; d\omega_i dy.$$

  The number of variables is reduced by one in the transfer function.
- In case of *directional lights* and *environment map lighting* (also called *sky illumination*) the incoming radiance L^{in} gets independent of entry point $\vec{y}$, thus we can write:

$$L^{out}(\vec{x}, \vec{\omega}_o) = \int_S \int_\Omega L^{in}(\vec{\omega}_i) \cdot T(\vec{y}, \vec{\omega}_i \to \vec{x}, \vec{\omega}_o) \; d\omega_i dy =$$

$$\int_\Omega L^{in}(\vec{\omega}_i) \cdot T^{env}(\vec{\omega}_i \to \vec{x}, \vec{\omega}_o) \; d\omega_i.$$

  where

$$T^{env}(\vec{\omega}_i \to \vec{x}, \vec{\omega}_o) = \int_S T(\vec{y}, \vec{\omega}_i \to \vec{x}, \vec{\omega}_o) \; dy.$$

  Again, we can observe the reduction of the number of independent variables by one in the transfer function.
- Diffuse surfaces illuminated by directional lights or environment maps have an even simpler formula where the number of variables of the transfer function are reduced to two:

$$L^{out}(\vec{x}) = \int_\Omega L^{in}(\vec{\omega}_i) \cdot T^{env}(\vec{\omega}_i \to \vec{x}) \; d\omega_i.$$

The transfer function depends on just the geometry and the material properties of the objects, and is independent of the actual lighting, thus it can be pre-computed for certain entry and exit parameters.

The problem is then how we can determine and represent a function defined for infinitely many points and directions, using finite amount of data. There are two straightforward solutions, *sampling* and the *finite-element method.*

Sampling

Sampling does not aim at representing the function everywhere in its continuous domain, but only at finite number of sample points.

For example, we usually do not need the output radiance everywhere only at sample points $\vec{x}_1, \ldots, \vec{x}_K$. Such sample points can be the points visible by a static camera, or vertices of a highly tessellated mesh (*per-vertex approach*), or even the points corresponding to the texel centers (*per-vertex approach*). In the last two cases it might happen that non sample points may also turn out to be visible, and their radiance is needed. In such cases, *interpolation* can be applied taking the neighboring sample points. Linear interpolation is directly supported by the graphics hardware. When sample points are the vertices, Gouraud shading can be applied, or when the points correspond to the texel centers, bi-linear texture filtering executes the required interpolation.

Similarly, when equation 17 is evaluated, some kind

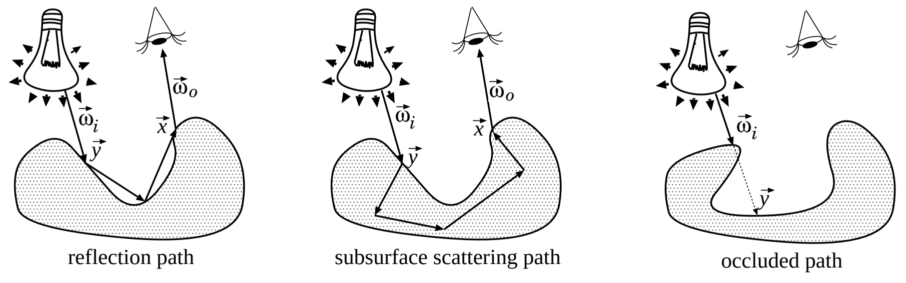


Figure 55: *Light paths sharing the same light and viewing rays*

of numerical quadrature is used, which takes finite number of $\vec{y}_1, \ldots, \vec{y}_N$, and $\vec{\omega}_{in,1}, \ldots, \vec{\omega}_{in,M}$ samples, thus the transfer function is needed only for these samples.

Interestingly, such approaches may completely eliminate pre-processing and provide real-time visualization exploiting data obtained during previous frames of the animation. *Light path reuse* [SSSK04] is a Monte Carlo technique to speed up animated light sequences assuming that the objects and the camera are still. Paths of many frames are combined together in a single frame. The combination is governed by multiple importance sampling guaranteeing that the variance of the solution in a frame is close to what could be obtained if we dedicated all paths solely to this particular frame.

Radiance transfer precomputation has also been applied to translucent objects [LGB*03], where the vertex-to-vertex radiosity transfer factors are stored and the response to an illumination is obtained by a finite-element series using these factors as weights.

On the other hand, Mei et al. [MSW04] have proposed a solution to render both shadows and self illumination. Their method is based on *spherical radiance transfer maps* including spherical shadow maps for both mutual and self-shadow rendering. These maps have to be pre-computed for every mesh vertex, and their resolution should be high enough to cover hundreds of sampled directions.

Finite-element method

According to the concept of the finite element method, the transfer function is approximated by a finite function series form:

$$T(\vec{y}, \vec{\omega}_i \to \vec{x}, \vec{\omega}_o) \approx$$

$$\sum_{n=1}^{N}\sum_{m=1}^{M}\sum_{k=1}^{K}\sum_{l=1}^{L} T_{nmkl} \cdot s_n(\vec{y}) \cdot d_m(\vec{\omega}_i) \cdot S_k(\vec{x}) \cdot D_l(\vec{\omega}_o).$$

where $s_n(\vec{x})$ and $S_k(\vec{x})$ are pre-defined spatial basis functions, and $d_m(\vec{\omega}_i)$ and $D_l(\vec{\omega}_i)$ are pre-defined directional basis functions. Since the basis functions are known, the transfer function at any $\vec{y}$, $\vec{\omega}_i$, $\vec{x}$, $\vec{\omega}_o$ is determined by coefficients T_{nmkl}. In the general case we have $N \times M \times K \times L$ number of such coefficients.

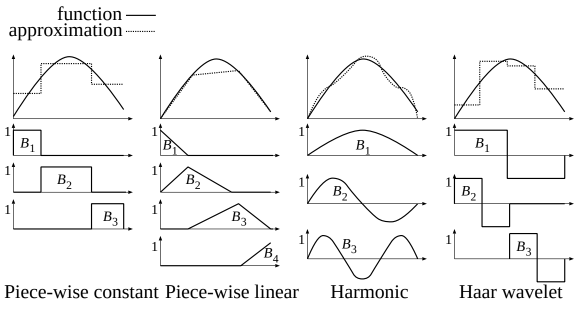


Figure 56: *Basis function systems and finite-element approximations*

There are many possible choices for the basis functions with different advantages and disadvantages (figure 56). Note that we can use different basis function sets for the spatial and directional domains, and even for the entry and exit parameters. The particularly popular choices are the following:

- *Piece-wise constant basis functions* partition the domain into subdomains and each basis function is 1 in exactly one subdomain and zero elsewhere. Piece-wise constant basis functions became very popular in radiosity algorithms.
- *Piece-wise linear basis functions* require the identification of sample points in the subdomain. Each basis function is non-zero at one sample point and linearly decreasing to zero between this sample point and the neighboring sample points. Piece-wise linear basis functions are equivalent to *linear interpolation* which is directly supported by the graphics hardware. Sample points can be the vertices when Gouraud shading can be applied, or the points

corresponding to the texel centers when bi-linear texture filtering executes linear interpolation. Note that sampling and finite-element approaches are close relatives in this sense.

- *Harmonic functions* correspond to the Fourier series approximation.
- *Real spherical harmonics* are similar to Fourier series but work in the directional domain.
- *Haar wavelets* also provide piece-wise constant approximation, but now a basis function is non-zero in more than one subdomains.

Having selected the basis functions, the coefficients are obtained evaluating the following integrals, using usually Monte Carlo quadrature:

$$T_{nmkl} =$$

$$\int_S \int_\Omega \int_S \int_\Omega T(\vec{y}, \vec{\omega}_i \to \vec{x}, \vec{\omega}_o) \cdot \tilde{A}_{nmkl}(\vec{y}, \vec{\omega}_i, \vec{x}, \vec{\omega}_o) \, d\omega_i dy d\omega_o dx,$$

where

$$\tilde{A}_{nmkl}(\vec{y}, \vec{\omega}_i, \vec{x}, \vec{\omega}_o) = \tilde{s}_n(\vec{y}) \cdot \tilde{d}_m(\vec{\omega}_i) \cdot \tilde{S}_k(\vec{x}) \cdot \tilde{D}_l(\vec{\omega}_o)$$

and $\tilde{s}_n(\vec{y}), \tilde{d}_m(\vec{\omega}_i), \tilde{S}_k(\vec{x}), \tilde{D}_l(\vec{\omega}_o)$ are *adjoints* of basis functions $s_n(\vec{y})$, $d_m(\vec{\omega}_i)$, $S_k(\vec{x})$, $D_l(\vec{\omega}_o)$, respectively. Basis functions $\tilde{s}_1, \ldots, \tilde{s}_N$ are said to be adjoints of $s_1, \ldots, s_N$ if the following conditions hold:

$$\int_S \tilde{s}_i(\vec{y}) \cdot s_j(\vec{y}) \, dy = \delta_{ij},$$

where $\delta_{ij} = 0$ if $i \neq j$ and $\delta_{ii} = 1$. A similar definition holds for directional basis functions as well.

Piecewise constant basis functions are adjoints with themselves. To guarantee that normalization constraint is also met, that is $\int_S \tilde{s}_i(\vec{y}) \cdot s_i(\vec{y}) \, dy = 1$, the adjoint constant basis functions are equal to the reciprocal of the size of their respective domains. Real spherical harmonics and Haar wavelets are also self-adjoint.

Substituting the finite-element approximation into the exit radiance, we can write

$$L^{out}(\vec{x}, \vec{\omega}_o) =$$

$$\sum_{k=1}^{K} \sum_{l=1}^{L} \sum_{n=1}^{N} \sum_{m=1}^{M} S_k(\vec{x}) \cdot D_l(\vec{\omega}_o) \cdot T_{nmkl} \cdot$$

$$\cdot \int_S \int_\Omega L^{in}(\vec{y}, \vec{\omega}_i) \cdot s_n(\vec{y}) \cdot d_m(\omega_i) \, d\omega_i dy. \qquad (18)$$

Note that after pre-computing coefficients T_{nmkl} just a low dimensional integral needs to be evaluated. This computation can further be speeded up if illumination function $L^{in}(\vec{y}, \vec{\omega}_i)$ is also approximated by a finite-element form

$$L^{in}(\vec{y}, \vec{\omega}_i) \approx \sum_{n'=1}^{N} \sum_{m'=1}^{M} L_{n'm'} \cdot \tilde{s}_{n'}(\vec{y}) \cdot \tilde{d}_{m'}(\vec{\omega}_i)$$

where the basis functions $\tilde{s}_{n'}$ and $\tilde{d}_{m'}$ are the *adjoints* of s_n and d_m, respectively.

Substituting the finite-element approximation of the incoming radiance into the integral of equation 18, we can write

$$\int_S \int_\Omega L^{in}(\vec{y}, \vec{\omega}_i) \cdot s_n(\vec{y}) \cdot d_m(\vec{\omega}_i) \, d\omega_i dy \approx$$

$$\int_S \int_\Omega \sum_{n'=1}^{N} \sum_{m'=1}^{M} L_{n'm'} \cdot \tilde{s}_{n'}(\vec{y}) \cdot \tilde{d}_{m'}(\vec{\omega}_i) \cdot s_n(\vec{y}) \cdot d_m(\vec{\omega}_i) \, d\omega_i dy$$

$$= L_{nm}$$

since only those terms are non-zero where $n' = n$ and $m' = m$. Thus the exit radiance is:

$$L^{out}(\vec{x}, \vec{\omega}_o) =$$

$$\sum_{k=1}^{K} \sum_{l=1}^{L} \sum_{n=1}^{N} \sum_{m=1}^{M} S_k(\vec{x}) \cdot D_l(\omega_o) \cdot T_{nmkl} \cdot L_{nm}. \qquad (19)$$

Different pre-computation aided real-time global illumination algorithms can be classified according to where they use sampling or finite-element methods, and to the considered special cases. For example, all methods published so far work with sample points and apply linear interpolation to handle the positional variation of exit point $\vec{x}$. However, the incoming direction is attacked both by different types of finite-element basis functions and by sampling.

Compression of transfer coefficients

The fundamental problem of pre-computation aided global illumination algorithms is that they require considerable memory space to store the transfer function coefficients. To cope with the memory requirements, data compression methods should be applied. A particularly popular approach is the *principal component analysis* (*PCA*) [SHHS03]. Values T_{nmkl} can be imagined as K number of points $\mathbf{T}^1 = [T_{nm1l}], \ldots, \mathbf{T}^K = [T_{nmKl}]$ in an $N \times M \times L$ dimensional space. To compress this data, we find a subspace (a "hyperplane") in the high dimensional space and project points $\mathbf{T}^i$ onto this subspace. Since the subspace has lower dimension, the projected points can be expressed by fewer coordinates, which results in data compression.

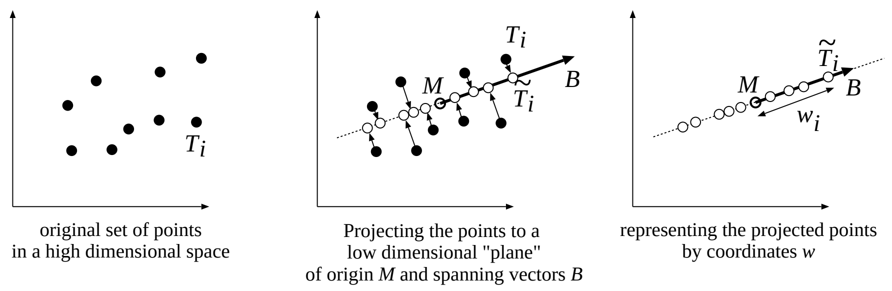

T_i
original set of points
in a high dimensional space
T_i
M
B
$\tilde{T}_i$
Projecting the points to a
low dimensional "plane"
of origin M and spanning vectors B
$\tilde{T}_i$
M
B
w_i
representing the projected points
by coordinates w

Figure 57: *The basic idea of principal component analysis. Points in a high dimensional (two dimensional in the figure) space are projected onto a lower, D-dimensional subspace ($D = 1$ in the figure, thus the subspace is a line), and are given by coordinates in this low dimensional subspace. To define these coordinates, we need a new origin $\mathbf{M}$ and basis vectors $\mathbf{B}_1, \ldots, \mathbf{B}_D$ in the lower dimensional subspace. The origin can be the mean of original sample points. In the example of the figure there is only one basis vector, which is the direction vector of the line.*

Let us denote the origin of the subspace by $\mathbf{M}$ and the basis vectors of the low-dimensional subspace by $\mathbf{B}_1, \ldots, \mathbf{B}_D$. Projecting into this subspace means the following approximation:

$$\mathbf{T}^i \approx \tilde{\mathbf{T}}^i = \mathbf{M} + w_1^i \cdot \mathbf{B}_1 + \ldots + w_D^i \cdot \mathbf{B}_D,$$

where $w_1^i, w_2^i, \ldots, w_D^i$ are the coordinates of the projected point in the subspace coordinate system. Of course, origin $\mathbf{M}$ and basis vectors $\mathbf{B}_1, \ldots, \mathbf{B}_D$ must be selected to minimize the total approximation error, i.e. the sum of the square distances between the original and projected points. As can be shown, the error is minimal if the origin is the mean of the original data points

$$\mathbf{M} = \sum_{k=1}^{K} \mathbf{T}^k,$$

and the basis vectors are the eigenvectors corresponding to the largest D eigenvalues of *covariance matrix*

$$\sum_{k=1}^{K} (\mathbf{T}^k - \mathbf{M})^T \cdot (\mathbf{T}^k - \mathbf{M}),$$

where superscript T denotes transpose operation (a row vector is turned to be a column vector).

If the transfer function samples are compressed, the reflected radiance can be obtained as follows:

$$L^r(\vec{x}_i) \approx \mathbf{L} \cdot \mathbf{T}(\vec{x}_i) \approx$$

$$\mathbf{L} \cdot \left(\mathbf{M} + w_1^i \cdot \mathbf{B}_1 + \ldots + w_n^i \cdot \mathbf{B}_D\right) =$$

$$\mathbf{L} \cdot \mathbf{M} + \sum_{d=1}^{D} w_d^i \cdot (\mathbf{L} \cdot \mathbf{B}_d).$$

If there are many original points, we usually cannot expect them to be close to a plane, thus this approach may have large error. This error, however, can be significantly decreased if we do not intend to find a subspace for all points at once, but cluster points first in a way that points of a cluster are roughly in a plane, then carry out PCA separately for each cluster. This process is called *Clustered Principal Component Analysis* (*CPCA*). Clustering can be based on the normal vectors, or on a real clustering method, such as the *K-means algorithm* [KMN*02].

Let us denote the means and the basis vectors of the cluster c by $\mathbf{M}_c$, and $\mathbf{B}_1^c, \ldots, \mathbf{B}_D^c$, respectively. Each sample point $\vec{x}_i$ belongs to exactly one cluster. If point $\vec{x}_i$ belongs to cluster c, then its reflected radiance is

$$L^r(\vec{x}_i) \approx \mathbf{L} \cdot \mathbf{T}(\vec{x}_i) \approx \mathbf{L} \cdot \mathbf{M}^c + \sum_{d=1}^{D} w_d^i \cdot (\mathbf{L} \cdot \mathbf{B}_d^c). \quad (20)$$

Implementing the clustering algorithm, then computing the means and the eigenvectors of the clusters, and finally projecting the points of the clusters to subspaces, are non-trivial tasks. Fortunately, functions exist in DirectX 9.0 that can do these jobs for us.

7.1. Pre-computed radiance transfer

The classical pre-computed radiance transfer [SKS02, Gre03] assumes that the lighting comes from directional lights or from environment lighting and that the surfaces are diffuse, reducing the variables of the transfer function to exit point $\vec{x}$ and incoming direction $\vec{\omega}_i$. The exit radiance is obtained just at sampling points $\vec{x}_k$. The remaining directional dependence is represented by directional basis functions. In this special case the output radiance at the sample point $\vec{x}_k$ is:

$$L^{out}(\vec{x}_k) = \sum_{m=1}^{M} T_m \cdot L_m. \tag{21}$$

If $\mathbf{T} = [T_1, \ldots, T_M]$ and $\mathbf{L} = [L_1, \ldots, L_M]$ are interpreted as M dimensional vectors, the result becomes an M dimensional dot product. Note that this formula is able to cope with translucent materials (subsurface scattering) [LGB*03] as well as opaque objects. When Sloan obtained this formula, he proposed the application of real spherical harmonics basis functions in the directional domain [SKS02]. While even surprisingly short spherical harmonics series are good to approximate smooth functions, they fail to accurately represent sharp changes. To solve this problem, Ng [NRH03] replaced spherical harmonics by Haar-wavelets to allow high frequency environment maps. However, working with wavelets we lose the rotational invariance of spherical harmonics, which allows the fast computation of the exit radiance when either the scene or the environment map is rotated [WLHN06].

If non-diffuse scenes are also considered under environment map illumination, the output radiance formula gets a little more complex:

$$L^{out}(\vec{x}_k, \vec{\omega}_o) = \sum_{l=1}^{L}\sum_{m=1}^{M} D_l(\omega_o) \cdot T_{ml} \cdot L_m, \tag{22}$$

which means that the radiance vector is multiplied by both a vector and a matrix [KSS02, LK03].

Classical pre-computed radiance transfer assumed infinitely far illumination source (directional or environment map illumination), which eliminated the dependence on entry point $\vec{y}$. To cope with light sources in finite distance, the dependence of entry point $\vec{y}$ should also be handled. Such approaches are called *local methods*.

In [AKDS04] a first-order Taylor approximation has been used to consider midrange illumination.

Kristensen et al. [KAMJ05] discretized the incident lighting into a set of localized (i.e. point) lights. Having a point light, we should integrate only in the domain of incoming directions since for a point light, the direction unambiguously determines entry point $\vec{y}$.

7.2. Implementation the diffuse pre-computed radiance transfer in DirectX

Since DirectX 9.0 PRT has become an integral part of the API. DirectX 9.0 implements both the per-vertex and the per-pixel approach. In the case of the per-vertex approach the algorithm requires highly tesselated models and vs_1_1 compatible hardware. In the case of per-pixel PRT, ps_2_0 compatible pixel shader is needed.

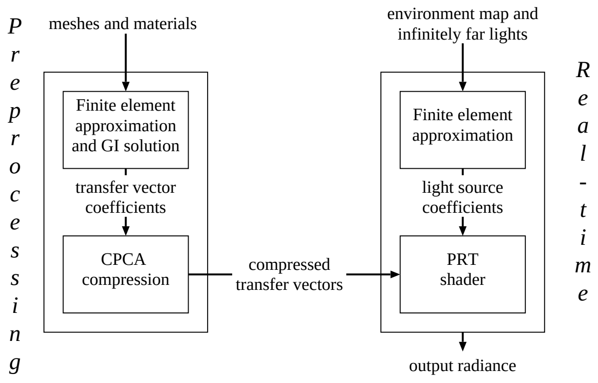

Preprocessing
meshes and materials
Finite element approximation and GI solution
transfer vector coefficients
CPCA compression
compressed transfer vectors
environment map and infinitely far lights
Finite element approximation
light source coefficients
PRT shader
output radiance
Real-time

Figure 58: *The simplified block diagram of PRT*

The process of the application of the built-in functions during pre-processing is:

1. Preprocessing of the transfer functions: we have to calculate the transfer function values $\mathbf{T}^i$ at each vertices of the highly tessellated mesh.
2. Compression of the transfer function values using CPCA.
3. Preprocessing of the environment lighting resulting in $\mathbf{L} = [L_0, \cdots, L_N]$.

We have to calculate outgoing radiance as the sum of the dot products of the transfer functions and environment lighting as given by equation 20. This operation is executed partly on the CPU and partly on the GPU. The CPU computes dot products $\mathbf{L} \cdot \mathbf{M}^c$ and $\mathbf{L} \cdot \mathbf{B}^c_d$ for each cluster $c = 1, \ldots C$, dimension $d = 1 \ldots D$, and for each channel of red, green, blue. The result is a sequence of $C \cdot 3 \cdot (D+1)$ floats that can be stored in an array, and passed as a uniform parameter. To take into account that GPUs work with `float4` data elements, these constants can also be stored in $C \cdot (1 + 3 \cdot (D+1)/4)$ number of `float4` registers (`Dots` in the implementation). The array offset of the data belonging to cluster k is passed as an input parameter.

The GPU is responsible for multiplying with weights w^d_i and adding the results together. The weight and the cluster id of a sample point are passed as texture coordinates. In the following function the number of clusters C is denoted by `NCLUSTERS` and number of dimensions $D+1$ by `NPCA`:

```
// Dot products computed on the CPU
float4 Dots[NCLUSTERS*(1+3*(NPCA/4))];

float4 GetPRTDiffuse( int k, float4 w[NPCA/4] ) {
   float4 col = dots[k]; // (M[k] dot L)
   float4 R = float4(0,0,0,0);
   float4 G = float4(0,0,0,0);
   float4 B = float4(0,0,0,0);

   // Add (B[k][j] dot L) for each j = 1.. NPCA
   // Since 4 values are added at a time,
   // the loop is iterated until j < NPCA/4
   for (int j=0; j < (NPCA/4); j++) {
      R += w[j] * dots[k+1+(NUM_PCA/4)*0+j];
      G += w[j] * dots[k+1+(NUM_PCA/4)*1+j];
      B += w[j] * dots[k+1+(NUM_PCA/4)*2+j];
   }
   // Sum the elements of 4D vectors
   col.r += dot(R,1);
   col.g += dot(G,1);
   col.b += dot(B,1);
   return col;
}
```

A possible implementation calls this function from the vertex shader to obtain the vertex color. The fragment shader takes the interpolated color and assign it to the fragment.

```
void PRTDiffuseVS(
   in  float4 Pos            : POSITION,
   in  int    k              : BLENDWEIGHT, // cluster
   in  float4 w[NUM_PCA/4]   : BLENDWEIGHT1 // weights
   out float4 hPos           : POSITION;
   out float4 Color          : COLOR0;
) {
   hPos = mul(Pos, WorldViewProj);
   Color = GetPRTDiffuse( k, w );
}

float4 StandardPS( in Color : COLOR0 ) : COLOR0 {
   return Color;
}
```

Figures 59, 60, and 61 show diffuse scenes rendered with the discussed method. During preprocessing a single bounce is computed with 1024 random rays. The environment map resolution is 256×256, and the order of spherical harmonics approximation, i.e. the number of directional basis functions is 6. Figures 59 and 61 computed just single bounces, thus these images are local illumination solutions using image based lighting. Figure 60, on the other hand, was computed simulating 6 bounces of the light. After preprocessing the camera can be animated at 466 FPS.

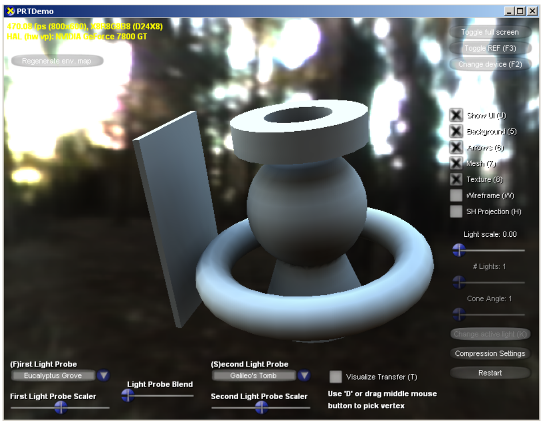


Figure 59: *During preprocessing 1 light bounce is computed in 12 sec*

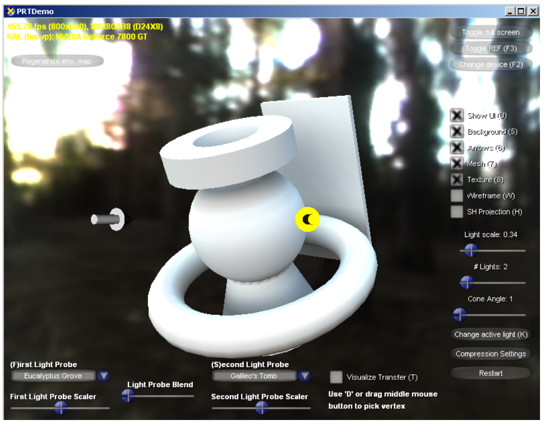


Figure 60: *During preprocessing 5 bounces are computed in 173 sec*

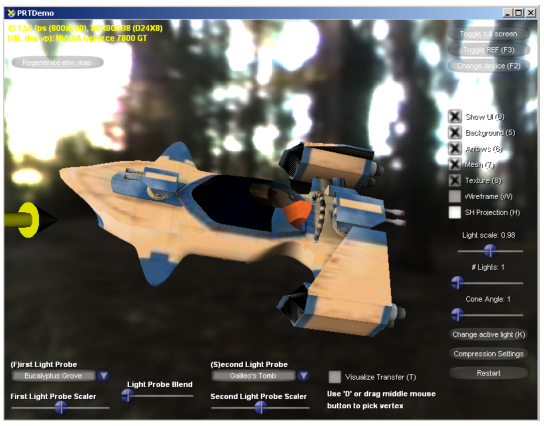


Figure 61: *During preprocessing 1 bounce is computed in 723 sec*

7.3. Light path maps

Pre-computed radiance transfer approaches use finite-element representation for all variables except for exit point $\vec{x}$, which is handled by sampling. However, we could handle other variables with sampling as well, and the collection of sampled data could be stored in *maps*. Using sampling instead of finite-element representation has both advantages and disadvantages. Wavelet or spherical harmonics based finite-element methods provide more compact representation, and thus require less memory. However, they are more difficult to compute and update to cope with slowly changing dynamic scenes. Thus sampling can be better when we want to trade pre-processing or update time for storage.

If we use sampling, the transfer coefficients are stored for sampled entry-exit point pairs, and stored in textures. The discussed method [SSKS06] consists of a preprocessing step and a fast rendering step.

Preprocessing

The preprocessing step determines the indirect illumination capabilities of the static scene. This information is computed for finite number of *exit points* on the surface, and we use interpolation for other points. Exit points are depicted by symbol $\times$ in figure 62. The exit points can be defined as points corresponding to the texel centers of the texture map of the surface.

The first step of preprocessing is the generation of certain number of *entry points* on the surface. These entry points are samples of first hits of the light emitted by moving light sources. During preprocessing we usually have no specific information about the position and the intensity of the animated light sources, thus entry points should cover the surfaces densely for all possible light source positions, and unit incoming radiance is assumed at these sample points. Entry points are depicted by symbol $\bullet$ in figure 62. Entry points are used as the start of a given number of light paths. A light path is a random or a quasi-random walk [Kel97] along the surface. In order to limit the length of paths, we can use random termination (Russian-roulette), or some deterministic decimation scheme [Kel97] to determine the maximum length of each path.

The visited points of the generated paths are connected to all those exit points that are visible from them. In this way we obtain a lot of paths originating at an entry point and arriving at one of the exit points. The contribution of a path divided by the probability of the path generation is a Monte Carlo estimate of the indirect illumination caused by the given exit lighting environment from which the rays are sampled. The sum of the Monte Carlo estimates of paths associated with the same entry and exit point pair is stored. We call this data structure the *precomputed radiance map*, or *PRM* for short. Thus a PRM contains *items* corresponding to groups of paths sharing the same entry and exit points. Items that belong to the same entry point constitute a PRM *pane*.

Rendering

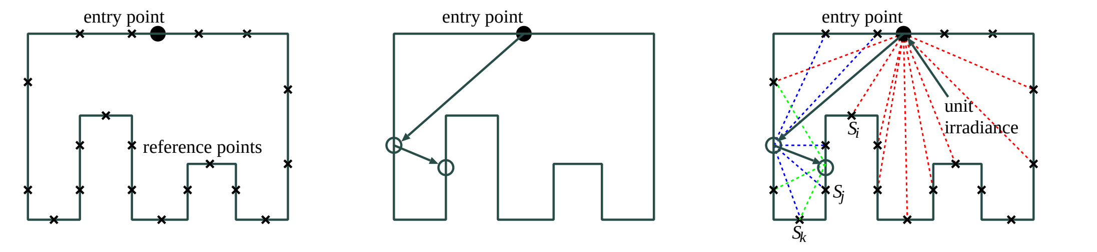
During real-time rendering, PRM is taken advantage of to speed up the global illumination calculation. The lights and the camera are placed in the virtual world (figure 63). The direct illumination effects are computed by standard techniques, which usually include some shadow algorithm to identify those points that are visible from the light source. PRM can be used to add the indirect illumination. This step requires visibility calculations, which is for free, since this visibility information was already obtained during direct illumination computation when shadows were generated [DS05].

A PRM pane stores the indirect illumination computed for the case when the respective entry point has unit irradiance. During the rendering phase, however, we have to adapt to a different lighting environment. The PRM panes associated with entry points should be weighted in order to make them reflect the actual lighting situation. Doing this for every entry point and adding up the results, we can obtain the visible color for each exit point. Then the object is rendered in a standard way with linear interpolation between the exit points.

Implementation

The light path maps algorithm consists of a preprocessing step, which builds the PRM, and a rendering step, which takes advantage of the PRM to evaluate indirect illumination. For the preprocessing step we implemented a combined CPU ray-tracing and GPU method. On the other hand, the rendering step was realized completely on the GPU.

First, a CPU program samples entry points, and generate random paths starting from them. Having completed a path, we compute the visibility between each path point and each exit point. Assuming that the entry point has unit irradiance, the contribution from the exit points are evaluated. For each exit point, the sum of the contributions is stored together with the index of the entry point, which constitutes the PRM. While generating random walks is best done on the CPU, the GPU is better in computing the effect of path points on exit points. In fact, we should execute a virtual light source algorithm [Kel97, WKB*02]. Since the exit points are the texels of a texture map, the virtual light source algorithm should be implemented

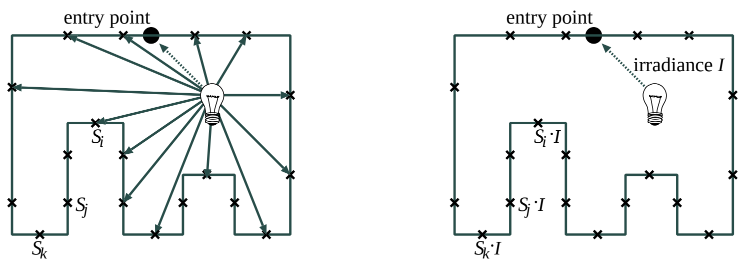


1. Exit points are defined and entry points are sampled

2. Path generation from the entry points

3. S is the illumination of exit points PRM = $\{(\bullet, \times, S)\}$

Figure 62: *Overview of the preprocessing phase of the light path maps method. Entry points are depicted by* $\bullet$*, and exit points by* $\times$*. The PRM is a collection of (entry point* $\bullet$*, exit point* $\times$*, illumination* S_k *) triplets, called items.*



1. Direct illumination + entry point visibility

2. Weighting irradiance I with items S

Figure 63: *Overview of the rendering phase of the light path maps method. The illumination of the entry points are computed, from which the illumination of the exit points is obtained by weighting according to the PRM.*

in a way that it renders into a texture map. Items are computed by rendering into the texture with the random walk nodes as light sources. Visibility can be determined using the shadow map technique.

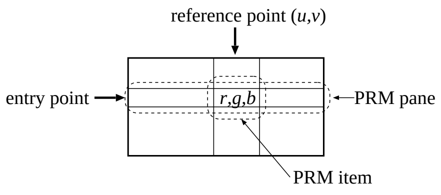


Figure 64: *Representation of a PRM as an array indexed by entry points and exit points. A single element of this map is the PRM item, a single row is the PRM pane.*

PRMs are stored in textures for real-time rendering. A single texel stores a PRM item that represents the contribution of all paths connecting the same entry point and exit point. A PRM can thus be imagined as an array indexed by entry points and exit points, and storing the radiance on the wavelengths of red, green, and blue (figure 64). Since a exit point itself is identified by two texture coordinates (u, v), a PRM can be stored either in a 3D texture or in a set of 2D textures (figure 65), where each represents a single PRM pane (i.e. a row of the table in figure 64, which includes the PRM items belonging to a single entry point).

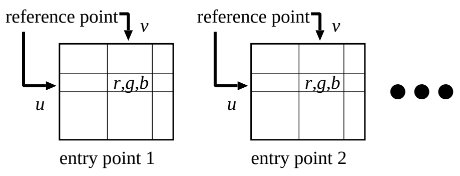


Figure 65: *PRM stored as 2D textures*

The number of 2D textures is equal to the number of entry points. However, the graphics hardware has just a few texture units. Fortunately, this can be sidestepped by tiling the PRM panes into one or more larger textures.

Using the method as described above allows us to render indirect illumination interactively with a typical number of 256 entry points. While this figure is generally considered sufficient for a medium complexity scene, difficult geometries and animation may emphasize virtual light source artifacts as spikes or flickering, thus requiring even more samples. Simply increasing the number of entry points and adding corresponding PRM panes would quickly challenge even the latest hardware in terms of texture memory and texture access performance. To cope with this problem, we can apply an approximation (a kind of lossy compression scheme), which keeps the number of panes under control when the number of entry points increase.

The key recognition is that if two entry points are close and lay on similarly aligned surfaces, then their direct illumination will be probably very similar during the light animation. Of course this is not true when a hard shadow boundary separates the two entry points, but due to the fact that a single entry point is responsible just for a small fraction of the indirect illumination, these approximation errors can be tolerated and do not cause noticeable artifacts. This property can also be understood if we examine how clustering affects the represented indirect illumination. Clustering entry points corresponds to a low-pass filtering of the indirect illumination, which is usually already low-frequency by itself, thus the filtering does not cause significant error. Furthermore, errors in the low frequency domain are not disturbing for the human eye. Clustering also helps to eliminate animation artifacts. When a small light source moves, the illumination of an entry point may change abruptly, possibly causing flickering. If multiple entry points are clustered together, their average illumination will change smoothly. This way clustering also trades high-frequency error in the temporal domain for low-frequency error in the spatial domain. Our clustering approach aims at compressing indirect illumination information similarly to precomputed radiance transfer and to Lightcuts [WFA*05]. However, in our case not the incoming radiance field is compressed, but the indirect illumination capabilities, which have low-frequency characteristics.

To reduce the number of panes, contributions of a cluster of nearby entry points are added and stored in a single PRM pane. As these *clustered entry points* cannot be separated during rendering, they will all share the same weight when the entry point contributions are combined. This common weight is obtained as the average of the individual weights of the entry points. Clusters of entry points can be identified by the K-means algorithm [KMN*02] or, most effectively, by a simple object median splitting kd-tree. It is notable that increasing the number of samples via increasing cluster size N_c has only a negligible overhead during rendering, namely the computation of more weighting factors. The expensive access and combination of PRM items is not affected. This way the method can be scaled up to problems of arbitrary complexity at the cost of longer preprocessing only.

While rendering the final image, the values stored in the PRM should be weighted according to the current lighting and be summed. Computing the weighting factors involves a visibility check that could be done using ray casting, but, as rendering direct illumination shadows would require a shadow map anyway, it can effectively be done in a shader, rendering to a one-dimensional texture of weights. Although these values would later be accessible via texture reads, they can be read back and uploaded into constant registers for efficiency. Furthermore, zero weight textures can be excluded, sparing superfluous texture accesses.

In order to find the indirect illumination at an exit point, the corresponding PRM items should be read from the textures and their values summed having multiplied them by the weighting factors and the light intensity. In the uncompressed case we can limit the number of entry points to those having the highest weights. Selection of the currently most significant texture panes can be done on the CPU before uploading the weighting factors as constants.

Figures 66 and 67 show a marble chamber test scene consisting of 3335 triangles, rendered on 1024×768 resolution. We used 4096 entry points. Entry points were organized into 256 clusters ($N_c = 4096/256$). Splitting factor N_s was 2. We set the PRM pane resolution to 256×256, and used the 32 highest weighted entry clusters. In this case the peak texture memory requirement was 128 Mbytes. The constant parameters of the implementation were chosen to fit easily with hardware capabilities, most notably the maximum texture size and the number of temporary registers for optimized texture queries, but these limitations can be sidestepped easily. As shown in figure 66, a high entry point density was achieved. Assuming an average albedo of 0.66 and a splitting factor of 2, the 4096 entry points displayed in figure 66 translate to approximately 24000 virtual light sources.

For this scene, the preprocessing took 8.5 sec, which can further be decomposed as building the kd-tree for ray casting (0.12 sec), light tracing with ray casting (0.17 sec), and PRM generation (8.21 sec). Having obtained the PRM, we could run the global illumi-

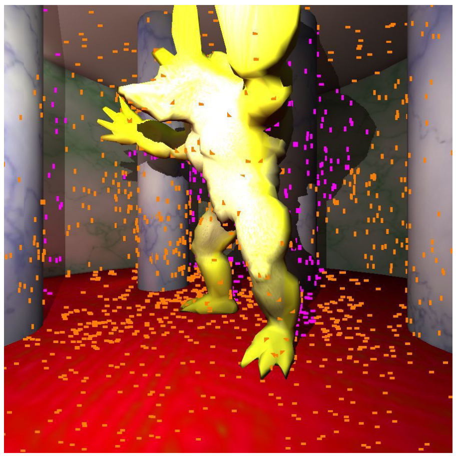


Figure 66: *Entry points*

nation rendering at 40 frames per second interactively changing the camera and light positions.

Figure 67 shows screen shots where half of the image was rendered with the new algorithm, and the other half with local illumination to allow comparisons. The effects are most obvious in shadows, but also notice color bleeding and finer details in indirect illumination that could not be achieved by fake methods like using an ambient lighting term.

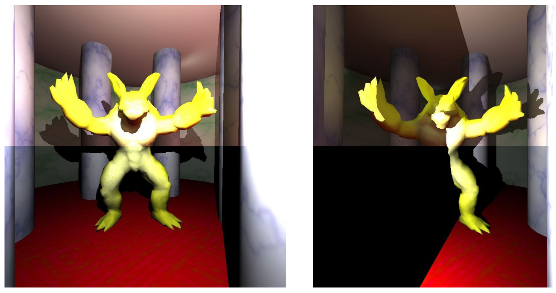


Figure 67: *Comparison of local illumination and the proposed global illumination rendering methods. The lower half of these images has been rendered with local illumination, while the upper half with the proposed global illumination method at 40 FPS.*

Conclusion

The role of the light path map method in interactive applications is similar to that of *light maps*. It renders indirect lighting of static geometry, but allows for dynamic lighting. Global illumination computations are performed in a precomputing step, using ray casting

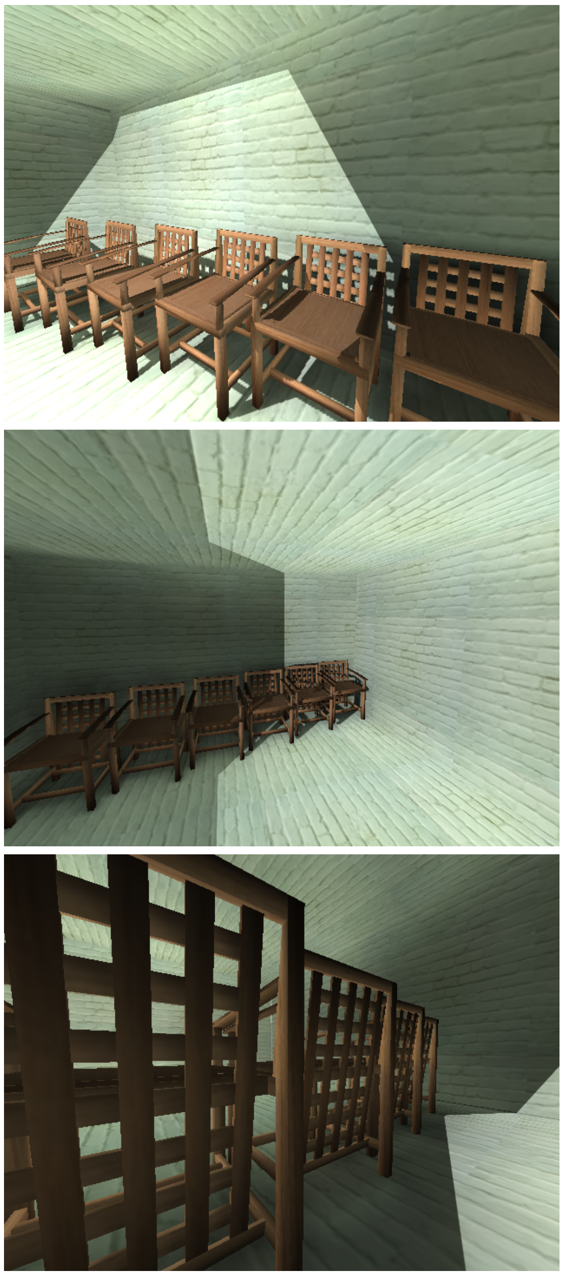


Figure 68: *The chairs scene lit by a rectangular spot light, whose illumination can be easily identified. The rest is pure indirect illumination obtained with the proposed method at 35 FPS.*

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. 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, which is also computed in a GPU pass.

The final result will be a plausible rendering of indirect illumination. Indirect shadows and color bleeding effects will appear, 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 demo implementation this number is 32, which provides pretty accurate, lighting dependent 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 demo 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.

Program resources

The PathMap application is a DirectX program. Class `PathMapEffect` manages all resources necessary to represent the scene entities, compute the PRMs and render to the screen. 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. Class `PathMapEffect` contains the following PRM-related resources:

- `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
- for every scene entity:
  - `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 (Figure 69). 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.)
- `depthMapTexture`: a depth texture (When rendering the contribution of virtual light sources into PRM atlases, we need to account 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.)
- `weightsTexture`: a weights texture (a render target to compute the form factor, or weight, corresponding to all entry points)
- `weights`: a cluster weights array (an array to hold the averaged weights of entry points in clusters)

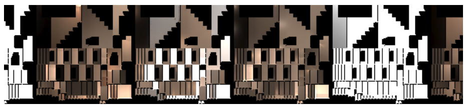


Figure 69: *A few tiles of a PRM texture. Illumination corresponding to clusters of entry points is stored in tiled atlases.*

Shader techniques

All of the shader techniques contains a single pass, with a unique vertex and pixel shader.

- `Depth`: Renders the scene with the given `WorldViewProj` transformation. 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. For a detailed description of a depth map rendering pass, refer to Section 3.1.1.
- `ToAtlas`: Renders the contribution of a single virtual light source (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. We explain this shader in detail in Section 7.3.
- `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.

Precomputing

For the indirect lighting of the scene, we need the PRM textures and the array of entry points along with clustering information. Precomputation has the following steps:

- Entry points are generated on the surfaces.
- Entry points are clustered according to their position and orientation.
  - Initial clustering is performed by building a balanced kd-tree, forming uniformly sized clusters.
  - More refined clustering is done using *k-means clustering*, 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 clusters relevant for its illumination are found, and stored.
- For all entry points in a cluster, a bush of virtual light sources is generated using ray shooting. Then, for all entities in the vector entities, the contribution of all virtual light sources is added to the PRM texture. This is done in the following steps:
  - For the actual cluster, the corresponding tile in the PRM texture of the entity is found using the `nearClusterIndices` array. If the current cluster is not listed, it is not relevant for the shading of the entity, so nothing is rendered.
  - For all virtual light sources in the cluster:
    - A depth map is rendered using technique `Depth`, into depth buffer `depthMap`.
    - 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 `ToAtlas`.

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

Final rendering

During the final walkthrough (Figure 70), 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. After computing the weights with technique `ComputeWeights`, the result texture `weightsTexture` has to be read into system memory. 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.

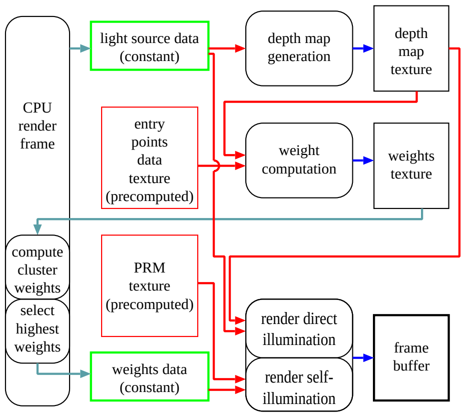


Figure 70: *Data flow between shaders during the final rendering (walkthrough) phase. Rounded rectangles correspond to shader techniques* `Depth`*,* `ComputeWeights`*, and* `Walk`*.*

Render-to-atlas shader implementation

The key technique for the generation of the PRM is technique `ToAtlas`. One pass using the technique renders the illumination due to a single point-like virtual light source into a texture atlas. The texture atlas is in a tile of the PRM texture: the rendertarget viewport is set to be that tile. Blending must be enabled to add the contribution of multiple light sources. When rendering to a texture atlas, all the triangles should be transformed to their positions in texture space. This is done in the vertex shader. Furthermore, the modeling transformation `World` has to be applied to get world space coordinates for per-pixel shading. We also compute the depth map texture position `dhPos` for shadowing.

```
void ToAtlasVS(
  in  float4 Pos    : POSITION,
  in  float2 Tex    : TEXCOORD0,
  in  float3 Norm   : NORMAL,
  out float4 tPos   : POSITION,
  out float4 wPos   : TEXCOORD0,
  out float3 wNorm  : TEXCOORD1,
  out float4 dhPos  : TEXCOORD2) //depth map
{
  tPos.xy = float2(2,-2) * Tex - float2(1,1);
    tPos.y = -tPos; //Tex space to screen space
  tPos.zw = 0;

  wPos = mul(Pos, World);
  wNorm = mul(Norm, WorldIT);
    dhPos = mul(wPos, DepthViewProjTex);
}
```

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 light source. 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 for points near to this plane (distance squared less than `cutNearness2`) are considered inaccurate because of insufficient sampling. `cutNearness2` is set according to the sampling probability. Any shaded fragment that is considered too close will be moved futher away for the purpose of form factor computing. Geometrically, it is projected onto the cutting plane, along 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. Otherwise, a regular per-pixel shading step with hardware shadow mapping is perfomed.

```
void ToAtlasPS(
  in  float4 wPos   : TEXCOORD0,
  in  float3 wNorm  : TEXCOORD1,
  in  float4 dhPos  : TEXCOORD2,
  out float4 oColor : COLOR)
{
  //homogenous division
  dhPos /= dhPos.h;

  wNorm = normalize(wNorm);
  float3 diff = lightPos - wPos.xyz;

  float3 diffDir = normalize(diff);
  float cosa = dot(diffDir, wNorm);
  float cosb = -dot(diffDir, normalize(lightDir));
  float dist2 = dot(diff, diff);

  oColor = 0;

  float visibility = tex2Dproj(depthMapSampler, dhPos);
  float formFactor;
  if( cosa > 0.0 && cosb > 0.0)
  { \\facing the light
    \\distance from light source plane
    float nearness = cosb * cosb * dist2;
    if( nearness < cutNearness2)
    { \\modified FF
      float e2 = dist2 * (1.0 - cosb * cosb);
      float dist2dash = e2 + cutNearness2;
      formFactor =
        cosa * sqrt(cutNearness2 / dist2dash )
        / dist2dash;
    }
    else \\standard FF
      formFactor = cosa * cosb / dist2;

    oColor = float4(
      lightPower * formFactor * visibility, 1);
    }
}
```

If we render to a texture atlas, and apply this texture on the object later, artifacts near texturing seams may appear (Figure 71). This happens because we might access a texel which we have never rendered to. The underlying cause is how the hardware rasterizes

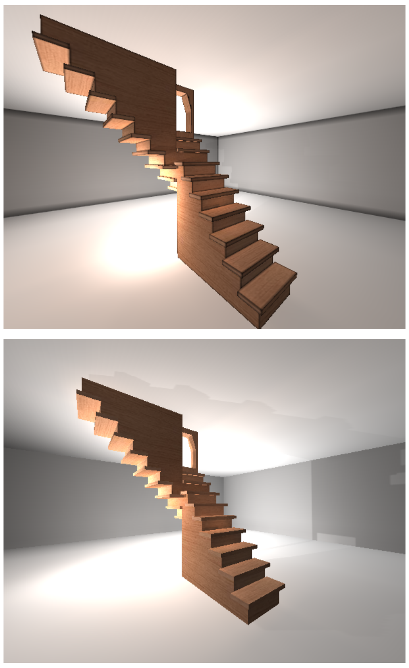


Figure 71: *Stairs rendered with indirect illumination using light path maps, without and with supplemental seam edge rendering.*

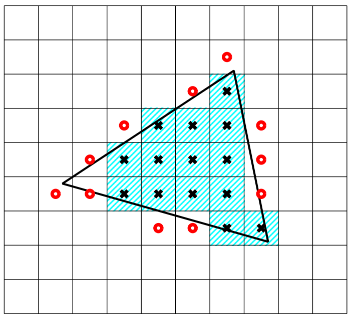


Figure 72: *The hardware supported triangle rasterization scheme.*

triangles. The general rule for drawing a triangle is to color those pixels, whose center is within the triangle (Figure 72. 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.

Looking at the texture atlas, we can realize that we actually need to extend the contours of the charts of adjacent triangles. 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 offset the line segments to augment the triangles so that they cover all texels they were overlapping with, as depicted in Figure 73.

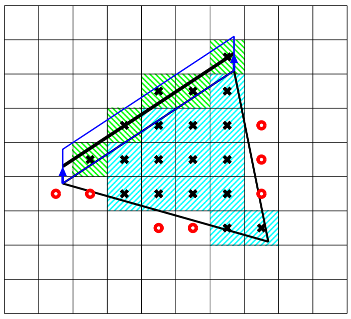


Figure 73: *Triangle rasterization with supplemental edges at texture seams.*

Thus we acquire a supplemental set of geometry made up of line segments. Whenever rendering the original geometry to the atlas, 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 not 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.

However, in the special case of rendering the contribution of a single light source into the PRM, we also use blending. Therefore, we must not render to a texel more than once, which is not guaranteed using the supplemental edge geometry. Therefore, we use the stencil test to exclude a texel once it has been rendered to.

7.4. Participating Media Illumination Networks

This section discusses a real-time method to compute multiple scattering in non-homogeneous participating media having general phase functions. The volume represented by a particle system is supposed to be static, but the lights and the camera may move. Lights can be arbitrarily close to the volume and can even be inside. Real-time performance is achieved by reusing light scattering paths that are generated with global line bundles traced in sample directions in a preprocessing phase. For each particle we obtain those other particles which can be seen in one of the sample directions, and their radiances toward the given particle. This information is stored in an illumination network that allows the fast iteration of the volumetric rendering equation. The illumination network can be stored in two-dimensional arrays indexed by the particles and the directions, respectively. Interpreting these two-dimensional arrays as texture maps, the iteration of the scattering steps can be efficiently executed by the graphics hardware, and the illumination can spread over the media in real-time.

Physically plausible rendering of participating media simulates multiple scattering effects [Max94, NDN96, LBC95]. As in case of surface models, accurate images require a lot of light paths, whose computation can be speeded up by reusing previously generated path segments. Paths to be reused are often built of parallel lines [BF89, Sbe96, SK99b, DMK00]. The set of global line bundles form an *illumination network*, which can replace the geometry of the surfaces or the density of the volume in illumination computations [HDKS00, SMKY04].

In the presented method [SKSU05] we use the global line bundle illumination network concept to participating media represented by a particle system, and obtain the particle radiance with iteration. Since the iteration works on the illumination network, it does not require ray casting or queries of the particle system. Encoding the illumination networks by textures, the GPU can calculate a scattering step on all particles and in all sampled directions rendering a single textured quadrilateral.

7.4.1. Multiple scattering in volumes

Let us consider how the light goes through participating media. The change of radiance L on path of length ds and of direction $\vec{\omega}$ depends on different phenomena:

- *Absorption* and *outscattering*: the light is absorbed or scattered out from its path when photons collide with the particles. If the probability of collision in a unit distance is τ, then the radiance changes by $-\tau \cdot L \cdot ds$ due to the collisions. After collision a particle may be either absorbed or reflected with the probability of *albedo* a.
- *Emission*: the radiance may be increased by the photons emitted by the participating media (e.g. fire). This increase is $L^e \cdot ds$ where L^e is the emission density.
- *Inscattering*: photons originally flying in a different direction may be scattered into the considered direction. The expected number of scattered photons from differential solid angle $d\omega'$ equals to the product of the number of incoming photons and the probability that the photon is scattered from $d\omega'$ to $\vec{\omega}$. The scattering probability is the product of the collision probability (τ), the probability of not absorbing the photon (a), and the probability density of the reflection direction, called *phase function* P. We use the Henyey-Greenstein phase function [HG40, CS92]:

$$P(\vec{\omega}', \vec{\omega}) = \frac{1}{4\pi} \cdot \frac{3(1-g^2) \cdot (1 + (\vec{\omega}' \cdot \vec{\omega})^2)}{2(2+g^2) \cdot (1 + g^2 - 2g(\vec{\omega}' \cdot \vec{\omega}))^{3/2}},$$

where $g \in (-1, 1)$ is a material property describing how strongly the material scatters forward or backward.

Taking into account all incoming directions Ω', we obtain the following radiance increase due to inscattering:

$$ds \cdot \tau(s) \cdot a(s) \cdot \left(\int_{\Omega'} L(s, \vec{\omega}') \cdot P(\vec{\omega}', \vec{\omega}) \, d\omega' \right).$$

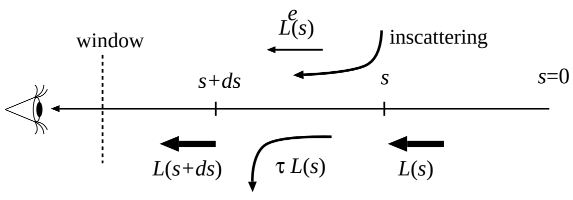


Figure 74: *Modification of the radiance of a ray in participating media.*

Adding the discussed changes, we obtain the following *volumetric rendering equation* for radiance L of the ray at $s + ds$:

$$L(s+ds, \vec{\omega}) = (1 - \tau(s) \cdot ds) \cdot L(s, \vec{\omega}) + ds \cdot L^e(s, \vec{\omega}) + ds \cdot \tau(s) \cdot a(s) \cdot \int_{\Omega'} L(s, \vec{\omega}') \cdot P(\vec{\omega}', \vec{\omega}) \, d\omega'. \qquad (23)$$

The particle system model of the volume corresponds to a discretization, when we assume that scattering can happen only at N discrete points called *particles.* We assume that particles are sampled randomly, preferably from a distribution proportional to collision density τ, and we do not require them to be placed at grid points [GRWS04]. As demonstrated in [HL01] such particle systems can generate acceptable clouds with a few hundred particles. Let us assume that particle p represents its spherical neighborhood of diameter $\Delta s^{(p)}$, and introduce its *opacity* as $\alpha^{(p)} = 1-e^{-\tau^{(p)}\cdot\Delta s^{(p)}}$, its *emission* as $E^{(p)} = L^{e,(p)}\cdot\Delta s^{(p)}$, its *incoming radiance* by $I^{(p)}$, and its *outgoing radiance* by $L^{(p)}$. The *discretized volumetric rendering equation* at particle p is then:

$$L^{(p)}(\vec{\omega}) = (1-\alpha^{(p)})\cdot I^{(p)}(\vec{\omega}) + E^{(p)}(\vec{\omega})+$$

$$\alpha^{(p)}\cdot a^{(p)}\cdot \int_{\Omega'} I^{(p)}(\vec{\omega}')\cdot P^{(p)}(\vec{\omega}',\vec{\omega})\, d\omega'.$$

In homogeneous media, albedo a and phase function P are the same for all particles. In non-homogeneous media, these parameters are particle attributes [REK*04].

We solve the discretized volumetric rendering equation by iteration. The volume is represented by a set of randomly sampled particle positions. Suppose that we have an estimate of particle radiance values (and consequently, of incoming radiance values) at iteration step $m-1$. The new particle radiance in iteration step m is obtained by substituting these values to the right inside of the discretized volumetric rendering equation:

$$L_m^{(p)}(\vec{\omega}) = (1-\alpha^{(p)})\cdot I_{m-1}^{(p)}(\vec{\omega}) + E^{(p)}(\vec{\omega})+$$

$$\alpha^{(p)}\cdot a^{(p)}\cdot \int_{\Omega'} I_{m-1}^{(p)}(\vec{\omega}')\cdot P^{(p)}(\vec{\omega}',\vec{\omega})\, d\omega'. \qquad (24)$$

This iteration is convergent if the opacity is in $[0,1]$ and the albedo is positive and less than 1, which is always the case for physically plausible materials.

In order to calculate the directional integral representing the inscattering term of equation 24, we suppose that D random directions $\vec{\omega}_1,\ldots,\vec{\omega}_D$ are obtained from uniform distribution of density $1/(4\pi)$, and the integral is estimated by Monte Carlo quadrature:

$$\int_{\Omega'} I^{(p)}(\vec{\omega}')\cdot P^{(p)}(\vec{\omega}',\vec{\omega})\, d\omega' \approx$$

$$\frac{1}{D}\cdot\sum_{d=1}^{D} I^{(p)}(\vec{\omega}'_d)\cdot P^{(p)}(\vec{\omega}'_d,\vec{\omega})\cdot 4\pi.$$

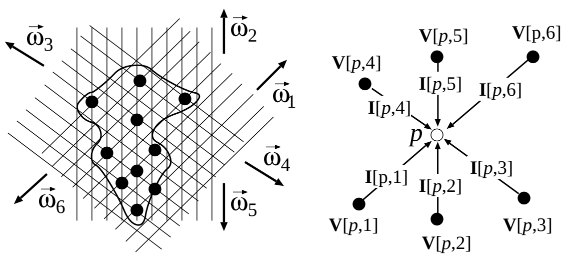


Figure 75: *Illumination and visibility networks*

7.4.2. Building the illumination network

If we use the same set of sample directions for all particles, then the incoming radiance and therefore the outgoing radiance are needed only at these directions during iteration. For a single particle p, we need D incoming radiance values $I^{(p)}(\vec{\omega}_d)$ in $\vec{\omega}_1,\ldots,\vec{\omega}_D$, and the reflected radiance needs to be computed exactly in these directions. 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 **I** and **V** of size $N\times D$, indexed by particles and directions respectively (figure 75). Array **I** is called the *illumination network* and stores the incoming radiance values of the particles on the wavelengths of red, green, and blue. Array **V** is the *visibility network* and stores index of visible particle vp and opacity α for each particle and incoming direction, that is, it identifies from where the given particle can receive illumination (figure 76).

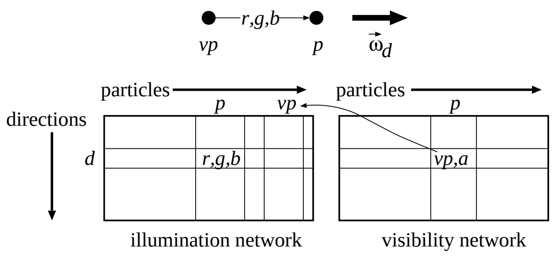


Figure 76: *Storing the networks in arrays*

In order to handle emissions and the direct illumination of light sources, we use a third array **E** that stores the sum of the emission and the reflection of the direct illumination for each particle and discrete direction. 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. At a particular particle, the discrete direction closest to the illumination direction is found, and the reflection of the light source is computed from the incoming discrete direction for each outgoing discrete direction.

Visibility network **V** expressing the visibility between particles is constructed during a preprocessing phase (figure 77). The bounding sphere of the volume is constructed and then D uniformly distributed points are sampled on its surface. Each point on the sphere defines a direction aiming at the center of the sphere, and also a plane perpendicular to the direction. A square window is set on this plane to include the projection of the volume, and the window is discretized to $M \times M$ pixels.

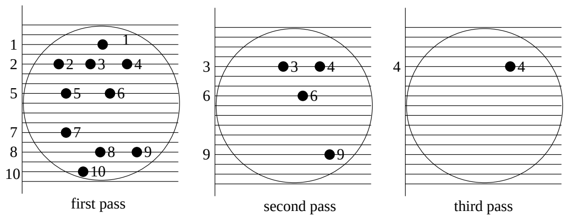


Figure 77: *Constructing the visibility network*

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 particle p equal to its index p (figure 77). 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. The particles that were visible in the preceding rendering step are ignored in the subsequent rendering steps. Repeating the rendering for the remaining particles and reading back the image and depth buffers again, we can obtain the indices of those particles which were occluded in the previous rendering step. Pairing these indices to those previously obtained ones which have the same pixel coordinates, we can get the pairs of particles that occlude each other in the given direction. On the other hand, the difference of the depth values is the distance of the particles, from which the opacity can be computed. Repeating the same step until the image is empty, we can build lists of particles that are projected onto the same pixel. Subsequent pairs of these lists define a single row of array **V** corresponding to this direction (figure 76). Executing this algorithm for all predefined directions, the complete array can be filled up.

Note that multiple z-buffer steps carry out a *depth-peeling* procedure. Since this happens in the preprocessing step, its performance is not critical. However, if we intend to modify the illumination network during rendering in order to cope with animated volumes, then the performance should be improved. Fortunately, the depth peeling process can also be realized on the GPU as suggested by [Eve01, Hac04].

7.4.3. Iterating the illumination network

The solution of the global illumination problem requires the iteration of the illumination network. A single step of the iteration evaluates the following formula for each particle $p = 1, \ldots, N$ and for each incoming direction $i = 1, \ldots, D$:

$$\mathbf{I}[p,i] = (1 - \alpha_{\mathbf{V}[p,i]}) \cdot \mathbf{I}[\mathbf{V}[p,i],i] + \mathbf{E}[\mathbf{V}[p,i],i] +$$

$$\frac{4\pi \cdot \alpha_{\mathbf{V}[p,i]} \cdot a_{\mathbf{V}[p,i]}}{D} \cdot \sum_{d=1}^{D} \mathbf{I}[\mathbf{V}[p,i],d] \cdot P_{\mathbf{V}[p,i]}(\vec{\omega}'_d, \vec{\omega}_i).$$

Interpreting the two-dimensional arrays of the emission, visibility and illumination maps as textures, the graphics hardware can also be exploited to update the illumination network. The first texture is visibility network **V** storing the visible particle in red and the opacity in green channels, the second stores emission array **E** in the red, green, and blue channels, and the third texture is the illumination map, which also has red, green and blue channels. Note that in practical cases number of particles N is about a thousand, while number of sample direction D is typically 128, and radiance values are half precision floating point numbers, thus the total size of these textures is quite small ($1024 \times 128 \times 8 \times 2$ bytes = 2 Mbyte).

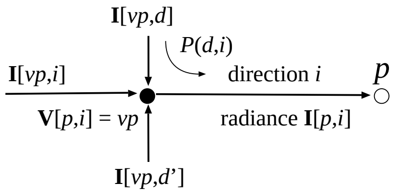


Figure 78: *Notations in the pixel shader code*

In the GPU implementation a single iteration step is the rendering of a viewport sized, textured rectangle, having set the viewpoint resolution to $N \times D$ and the render target to the texture map representing the updated illumination network. A pixel corresponds to a single particle and single direction, which are also identified by input variable `texcoord`. The pixel shader obtains the visibility network from texture `Vmap`, the emission array from texture `Emap`, and the illumination map from texture `Imap`. Function `P` is responsible for the phase function evaluation, which is implemented by a texture lookup of prepared values

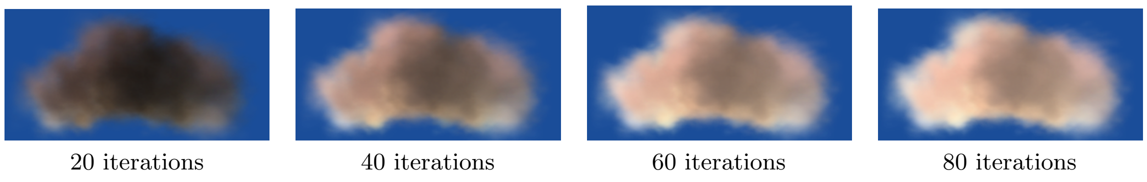


Figure 79: *A cloud illuminated by two directional lights rendered with different iteration steps*

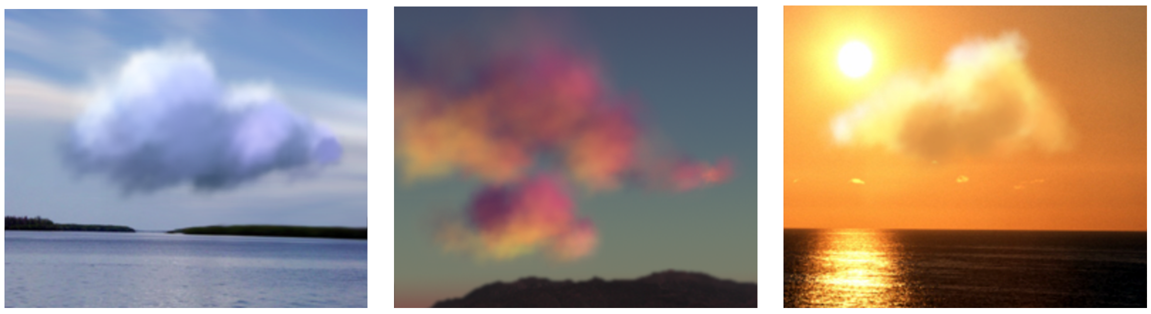


Figure 80: *Globally illuminated clouds of 512 particles rendered with 128 directions at 45 FPS.*

allowing other phase functions to be also easily incorporated [REK*04]. In this simple implementation we assume that opacity `alpha` is precomputed and stored in the visibility texture, but albedo `alb` are constant for all particles. Should it not be the case, the albedo could also be looked up in a texture.

When no other particle is seen in the input direction, then the incoming illumination is taken from the sky radiance (`sky`). In this way not only a single sky color, but sky illumination textures can also be used [PSS99, REK*04].

For particle `p` and direction `i`, the pixel shader finds opacity `alpha` and visible particle `vp` in direction `i`, takes its emission or direct illumination `Evp`, and computes and radiance `Ip` as the sum of the direct illumination and the reflected radiance values for its input directions `d = 1...D` (figure 78):

```
float  p  = texcoord.x; // particle
float  i = texcoord.y;  //input direction
float  vp = tex2d(Vmap, float2(p,i)).r;
if (vp >= 0) { // another particle is seen
   float alpha = tex2d(Vmap, float2(p,i)).g;
   float3 Evp = tex2d(Emap, float2(vp,i)).rgb;
   float3 Ivp = tex2d(Imap, float2(vp,i));
   float3 Ip = (1 - alpha) * Ivp + Evp;
   for(int d = 0; d < 1; d += 1.0/D) {
      Ivp = tex2d(Imap, float2(vp, d));
      float3 BRDF = alb * alpha * P(d,i);
      Ip +=  BRDF * Ivp * 4 * PI / D;
   }
   return Ip;
} else return sky; // no particle is seen
```

The illumination network provides a view independent radiance representation. 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.

In the current implementation we compute one iteration in each frame, and when the light sources move, we take the solution of the previous light position as the initial value of the iteration, which results in fast convergence.

The results are shown in figures 79 and 80. The cloud model consists of 1024 particles, and 128 discrete directions are sampled. With these settings the typical rendering speed is about 26 frames per second, and is almost independent of the number of light sources and of the existence of sky illumination. The albedo is 0.9 and material parameter g is 0. In figure 79 we can follow the evolution of the image of the same cloud after different iteration steps, where we can observe the speed of convergence.

8. Fake global illumination

Global illumination computation is inherently costly since all other points may affect the illumination of a single point. The inherent complexity can be reduced if the physically plausible model is replaced by another model that is simpler to solve but provides somehow similarly looking results. The dependence of the radiance of a point on all other points can be eliminated if we recognize that the illumination influence diminishes with the distance, thus it is worth considering only the local neighborhood of each point during shading. In directions where the local neighborhood is not visible, a constant or direction dependent *ambient* illumination is assumed, similarly to local illumination models.

In the *obscurances method* the "illumination of the local neighborhood" is expressed by the average of the the spectral reflectivities of its points, thus even color bleeding effects can be cheaply simulated [ZIK98, IKSZ03, MSC*05].

On the other hand, *ambient occlusion* [Hay02, PG04, KA06] approximates only the solid angle in which the neighborhood is seen, and computes a scalar occlusion factor. Recent work [Bun05] extended this algorithm as well to incorporate color bleeding.

Here we discuss only the obscurances method in details.

8.1. The obscurances method

In the obscurances method the effects of direct and indirect diffuse illumination are decoupled. Instead of working with the physically based reflected radiance formula of equation 1, for the sake of simplification, the reflection of the indirect diffuse illumination for point $\vec{x}$ is defined as:

$$L^{ind}(\vec{x}) = f_r(\vec{x}) \cdot L^a \cdot \int\limits_{\vec{\omega}' \in \Omega'} \rho(d(\vec{x}, \vec{\omega}')) \cdot \cos^+ \theta' \, d\omega' \quad (25)$$

where

- $f_r(\vec{x})$ is the BRDF at $\vec{x}$,
- L^a is the ambient light intensity,
- $d(\vec{x}, \vec{\omega}')$ is the distance between $\vec{x}$ and the next surface at direction $\vec{\omega}'$ or ∞ is there is no occlusion in this direction,
- $\rho(d)$ is the scaling of ambient light incoming from distance d, which takes values between 0 and 1,
- θ' is angle between direction $\vec{\omega}'$ and the normal at $\vec{x}$.

Function $\rho()$ increases with distance d thus the illumination of distant occluders is gradually replaced by the ambient light (figure 81).

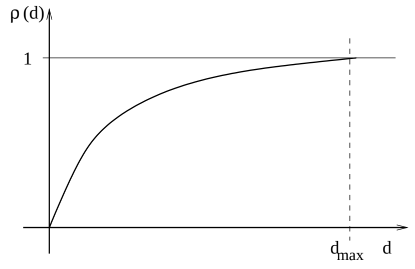


Figure 81: *Shape of $\rho(d)$ function.*

Maximum distance for interaction d_{max} is defined so that $\rho(d) = 1$ when $d \geq d_{max}$. This means that we take into account only a d_{max}-neighborhood of $\vec{x}$. In other words, we are not taking into account occlusions farther than d_{max}. The function used for this purpose is

$$\rho(d) = \sqrt{d/d_{max}}$$

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

The *obscurance* of point $\vec{x}$ is then defined as:

$$W(\vec{x}) = \frac{1}{\pi} \cdot \int\limits_{\vec{\omega}' \in \Omega'} \rho(d(\vec{x}, \vec{\omega}')) \cdot \cos^+ \theta' \, d\omega' \quad (26)$$

Since $0 \leq \rho(d) \leq 1$, we can assume that $0 \leq W(P) \leq 1$. The obscurance for a patch is the average of the obscurances for all points within the patch. An obscurance value of 1 means that the patch is totally open (or not occluded by neighboring polygons), while a value of 0 means that it is totally closed (or occluded by neighboring polygons).

For a closed environment, the ambient light in (25) can be computed as the average light intensity in the scene using the following formula:

$$L^a = \frac{R_{ave}}{1 - R_{ave}} \cdot \frac{\sum_{i=1}^{n} A_i L_i^e}{A_T} \quad (27)$$

where

$$R_{ave} = \frac{\sum_{i=1}^{n} A_i f_i}{A_T} \cdot \pi \quad (28)$$

is the average reflectivity and A_i, L_i^e, and f_i are the area, emission radiance, and reflectivity of patch i, respectively, A_T is the sum of the areas, and n is the number of patches in the scene. The ambient term considered here corresponds to the indirect illumination only, as direct illumination is computed separately.

The obscurance approach as presented so far lacks one of the features which comes from radiosity lighting, *color bleeding*. Since the light reflected from a patch acquires some of its color, the surrounding patches receive colored indirect lighting. To account for color bleeding, the obscurances formula (26) is modified slightly to include the reflectivity term of the patches:

$$W(\vec{x}) = \int\limits_{\vec{\omega}' \in \Omega'} f_r(\vec{y}) \cdot \rho(d(\vec{x}, \vec{\omega}')) \cdot \cos^+ \theta' \, d\omega' \qquad (29)$$

where $f_r(\vec{y})$ is the diffuse BRDF of point $\vec{y}$ seen from $\vec{x}$ in direction $\vec{\omega}'$. When no surface is seen at a distance less than d_{max} in direction $\vec{\omega}'$, the obscurance takes the value of R_{ave}.

For coherency, the ambient light equation (27) also has to be modified, yielding the following value:

$$L^a = \frac{1}{1 - R_{ave}} \cdot \frac{\sum_{i=1}^{n} A_i L_i^e}{A_T}. \qquad (30)$$

The usual option to compute the obscurance equation (29) uses Monte Carlo (or quasi Monte Carlo) ray casting technique, which casts several rays from a patch at random directions sampled with probability density $\cos^+ \theta'/\pi$. The obscurance for patch i will then be the average of the values gathered by the rays cast from this patch:

$$W(i) \approx \frac{\pi}{N_i} \cdot \sum_{j=1}^{N_i} \rho(d_j) f_r(\vec{y}_j) \qquad (31)$$

where N_i is the number of rays cast from patch i, $\vec{y}_j$ is the point hit by ray j, and d_j is the length of this ray.

Since casting cosine distributed rays from all patches in the scene is equivalent to casting global ray bundles of uniformly distributed random directions [Sbe97], obscurances can also be obtained with global ray bundles (Figure 82). The term *global ray* means that we use all ray-surface intersections not only the hit point closest to the ray origin as in the case of *local rays*.

Bundles of parallel global rays can be efficiently cast on the graphics hardware using the *depth peeling algorithm* [SKP98, Eve01, Hac05].

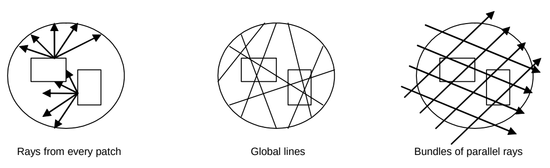


Figure 82: *Different sampling techniques to generate cosine distributed rays.*

The depth peeling algorithm renders the scene using orthogonal projection and sets patch colors to their IDs. Generating this image is equivalent to tracing a bundle of parallel rays through the scene where each pixel corresponds to a ray in the bundle. Each of these rays may intersect several surfaces in the scene, but the z-buffer algorithm keeps only the ID of the closest surface in the color buffer and its distance from the front clipping plane in the depth buffer. Since we need not only the first hits of the rays but all surface ray intersections, the rendering process is repeated. However, in the second run, the pixel shader will ignore those surfaces are not farther than the respective distance stored of the z-buffer of the previous run. The first layer of the scene is peeled away and the second rendering identifies the second ray-surface intersections. Repeating this process we can identify all ray-surface intersections not only the closest ones.

We assume that in a pre-processing step the scene is completely mapped to a single texture atlas. A texel of the texture atlas corresponds to a small surface patch. When a patch is referenced, we can simply use the texture address of the corresponding texel. The obscurance computation picks a random direction and carries out depth-peeling process in this direction. When we let the GPU do it for us, we use an orthogonal projection, and from the sampled direction we render the scene setting the model-view transform to rotate the sample direction to the z axis.

We use the *pixel* (RGBA) of the floating point format render target to store the patch identification, a flag that indicates whether the patch is front-facing or back-facing to the global direction, and the patch distance from the bounding sphere in the global direction. Our render target is initialized with $(-1.0, -1.0, 1.0, 1.0)$, giving reasonable default values.

The facing direction of a pixel can be determined by using the cosine of the angle between the camera's $-z$ vector and the normal vector of the patch. If the result is greater than 0, it is front-facing, otherwise it is back-facing.

As we store the pixels in a four-component float array (or the RGBA color), we use the first two components to store the patch ID ($RG \leftarrow (u, v)$), the third to store the cosine ($B \leftarrow \cos\alpha$), and the fourth component to store the distance between the front clipping plane and the patch ($A \leftarrow z$).

The vertex shader receives the vertex coordinates, the texture coordinates, and the normal, and it generates the cosine and the transformed vertex position. Note that the following shaders are written in Cg and assume OpenGL API, thus uniform parameters are also passed formally, and matrix vector multiplications have different order than in DirectX/HLSL programs:

```
void PeelVS(
   in  float4 Pos       : POSITION, // modeling space
   in  float2 PatchId   : TEXCOORD0,// patch ID
   in  float4 Norm      : NORMAL,   // normal vector
   out float4 oPos      : POSITION, // camera space
   out float2 oPatchId  : TEXCOORD0,// patch ID
   out float cosine     : TEXCOORD1
   uniform float4x4 WorldView,
   uniform float4x4 WorldViewIT)
{
   oPos = mul(WorldView, Pos); // To camera space
   oPatchId = PatchId;

   // sample direction is rotated to (0,0,1)
   cosine = mul(WorldViewIT, Norm).z;
}
```

The fragment shader receives the interpolated texture coordinates of the fragment, the position (where z is the depth), the cosine and the interpolated texture coordinates of the patch. For the first layer, the depth does not need to be compared with the previous one. However, for all subsequent layers, we sample the previous layer using the texture coordinates and discard the current fragment if the depth of the previous layer (the fourth component of the sample) is closer to the camera than the current fragment thus getting the peeling effect (Figure 83).

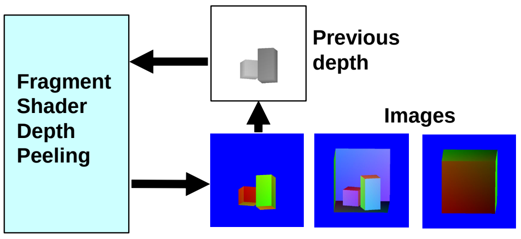


Figure 83: *Depth peeling with GPU.*

This rendering step is repeated until all pixels are discarded. The images of all the rendered layers define all ray-surface intersections.

The fragment shader code is:

```
float4 PeelPS(
   float4 wPos     : WPOS,       // viewport space
   float2 PatchId  : TEXCOORD0, // patch ID
   float  cosine   : TEXCOORD1,
   uniform sampler2D ztex, // prev. layer
   uniform float res,      // target resolution
   uniform float first     // is first layer?
   ) : COLOR
{
   if (first == 0.0) { // if not first layer -> peel
          // get depth of the previous layer
      float depth = tex2D(ztex, wPos.xy/res).a;
      if (wPos.z < (depth + eps))
         discard; // ignore previous layers
   }
   float4 color;
   color.rg = PatchId;
   color.b = cosine;
   color.a = wPos.z;   //new depth
   return color;
}
```

In the obscurance computation method random directions are sampled. Once we have chosen a random direction for the bundle, the computation is divided into two phases. In the first phase, layers are obtained using depth-peeling. In the second phase, the obscurances between each pair of layers are computed and the result is added and averaged in the corresponding obscurance map position.

Now, for each pair of consecutive layers, the obscurance formula has to be computed. We configure the camera to obtain a one-to-one mapping between pixels and texels. The size of the viewport is set to the same resolution as the obscurance map, starting from $(0, 0)$, with an orthogonal projection from -1 to $+1$ in both dimensions.

Now each pair of consecutive images is taken from the texture memory and sent to the graphic pipeline as a stream of points of size 1.0 (render to vertex array). This way we can update a single position in the target buffer for each element of the image. This will generate a pair of point streams A and B that are merged together and sent to the vertex shader. Stream A is sent as vertex positions and stream B as texture coordinates. As we generate the streams in both images in the same way, points at the same position in streams A and B are at the same position in consecutive images, thus may see each other in the sampling direction and transfer energy consequently (Figure 84).

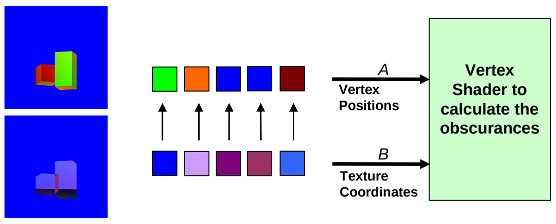


Figure 84: *Two consecutive layers (left) generate two streams of points carrying patch ID's (middle) that are merged together and processed by the vertex shader (right).*

The obscurance computation needs to be done bidirectionally but we cannot generate two values in different positions of the target buffer in a single pass and thus we have to do a two-pass transfer. In the first pass, we update the patches in the projection that generated stream A using the information in pixels of stream B (Figure 84). In the second pass the streams are exchanged, thus the same set of shaders are used in both directions.

If a patch in stream A cannot see the corresponding patch in stream B, the vertex carrying this patch is eliminated by moving it out of the view frustum. If patches see each other and the difference between their distances to the camera is less than d_{max}, then the transfer is done. If the distance is greater, then the patch gets the ambient reflectivity. When the transfer process is done, we generate vertex coordinates to update the position in the obscurance map that corresponds to the patch identified by the two first components of the current pixel element in stream A.

The vertex shader needs to generate vertex coordinates in homogeneous clipping space. The desired position is encoded as texture coordinates, and is consequently in the unit interval. The following formula computes homogeneous clip coordinates corresponding to given normalized device coordinates assuming OpenGL API (this is slightly different in DirectX):

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = \begin{pmatrix} 2x_w - 1 \\ 2y_w - 1 \\ 2z_w - 1 \\ 1 \end{pmatrix} \tag{32}$$

Note that the clipping process only keeps the fragment if $-w_c \le z_c \le w_c$ assuming OpenGL API. As we set w_c to 1, the clipping process will only keep the fragment when $-1 \le z_c \le 1$. If $z_c = 2.0$, then the fragment is out of the view frustum and is discarded. If we set $z_c = 0.0$, then the vertex is kept by the clipping process. So we can use the z_c value as a way to accept or discard vertices. The vertex shader for the obscurance transfer process is:

```
void ObsTransVS(
   in  float4 A     : POSITION,  // ID+cosine+depth
   in  float4 B     : TEXCOORD0, // ID+cosine+depth
   out float4 oPos  : POSITION,
   out float2 pA    : TEXCOORD0, // IDs of A
   out float2 pB    : TEXCOORD1, // IDs of B
   out float  dist  : TEXCOORD2, // length of ray
   uniform float dir) // forward/backward
{
// If patches see each other, i.e. both exist
// and one is front facing, the other is back facing
   if (((dir == 0) && (A.r != -1) && (A.b < 0)
           && ((B.b > 0) || (B.r == -1)))
    || ((dir == 1) && (A.r != -1) && (A.b > 0)
           && ((B.b < 0) || (B.r == -1)))) {
      pA = A.rg;
      pB = B.rg;
      // Create vertex to update desired position.
      oPos = float4(2 * pA.r - 1, 2 * pA.g - 1, 0, 1);
      // Calculate distance.
      dist = (texCoord.b != 1.0)? abs(B.a - A.a) : 1;
   } else // If no transfer, move out of the view frustum.
      oPos = float4(0, 0, 2, 1);
   }
}
```

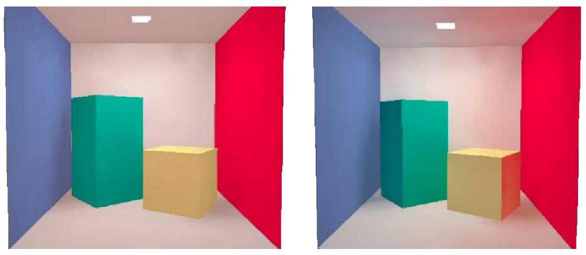


Figure 85: *Obscurances with (left) and without (right) color bleeding.*

The fragment shader evaluates the obscurance formula:

```
float4 ObsTransPS(
   float2 pB : TEXCOORD0,  // patch A
   float2 pB : TEXCOORD1,  // patch B
   float  dist : TEXCOORD2,// distance of patches
   uniform sampler2D refl, // reflectivities
   uniform float dmax,     // max distance
   uniform float3 ambient  // ambient light
   ) : COLOR
{
   float4 Color;
   if(d >= dmax)
        Color.rgb = ambient;
   else Color.rgb = tex2D(refl, pB).rgb *
                    sqrt(distance/dmax);
   Color.a = 1.0;
   return Color;
}
```

In figure 85 we show the Cornell box scene computed using the obscurances method without and with color bleeding.

Figures 86, 87, and 88 show the application of the obscurances map, obscurances with direct illumination, and direct illumination with constant ambient term.

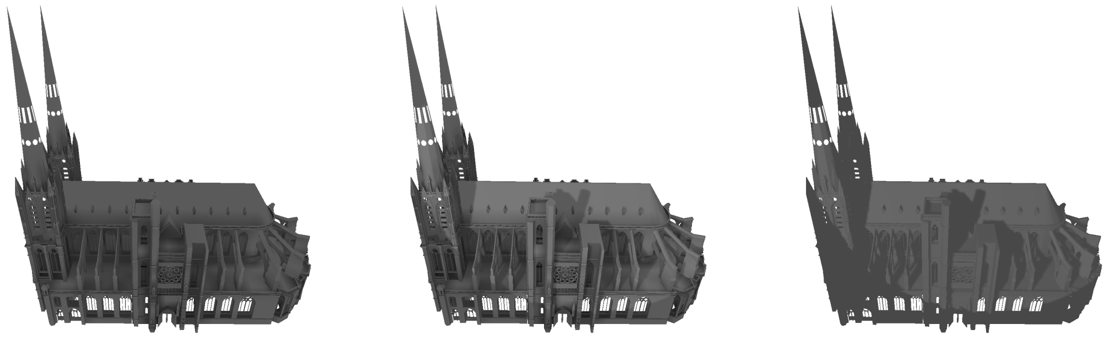


Figure 86: *Cathedral model, 193180 polygons, obscurances computed in 38 seconds. Left: obscurances map, middle: obscurances with direct illumination, right: constant ambient term with direct illumination.*

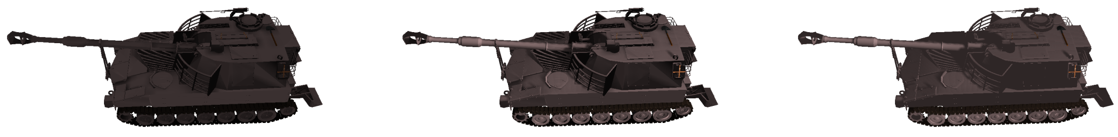


Figure 87: *Tank model, 225280 polygons, obscurances computed in 38 seconds. Left: obscurances map, middle: obscurances with direct illumination, right: constant ambient term with direct illumination.*

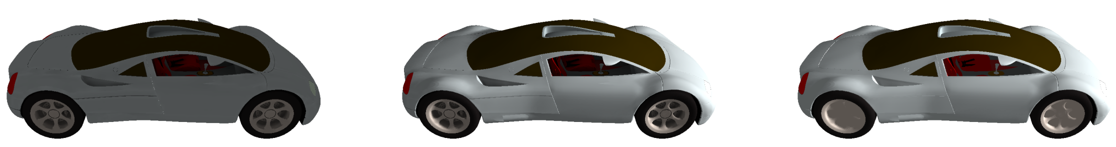


Figure 88: *Car model, 97473 polygons, obscurances computed in 32 seconds. Left: obscurances map, middle: obscurances with direct illumination, right: constant ambient term with direct illumination.*

9. Conclusions

In this tutorial, we have described the basic concepts and methods applied in global illumination computations on the GPU. Based on these elements, we have detailed a number of sophisticated rendering algorithms addressing various global illumination effects. Together, these provide a wide spectrum of tools, which can be combined to render realistic images interactively.

The scope of this tutorial could not encompass all techniques possibly required in a general rendering engine application. Well known hardware supported features like bump, normal or vertex displacement mapping are always a valuable asset. High dynamic range computations we performed in GI algorithms allows for post-processing effects that mimic the characteristics of image capture equipment or the human eye: tone mapping, glow, motion blur or lens flare [GWWH03][Tót06]. While those techniques do not strictly address global illumination problems, they are also helpful in creating a more immersive viewer experience.

We also did not go into details about the wide variety of advanced shadow rendering techniques that either reduce shadow map artifacts (shadow map focusing, perspective shadow maps [SD02], light space perspective shadow maps [WSP04], silhouette map [SCH03]) or render plausible approximate soft shadows [HLHS03]. Depending on the area of application, these improvements might make a crucial performance difference.

The algorithms we have described are all targeted at a range of speeds from real-time to interactive. However, using a combination of them for more complex geometries may challenge current graphics cards. A full, global illumination solution for a general dynamic scene remains a task of extreme complexity for which no processing power is undepletable. The future capabilities of graphics cards will call for new ways to accommodate existing general global illumination methods and extend current the CPU-based ones. Currently, with adequate precomputation and a careful selection of tools, it is absolutely possible to render a scene in a real-time application like a computer game, with plausible, if not accurate, global illumination.

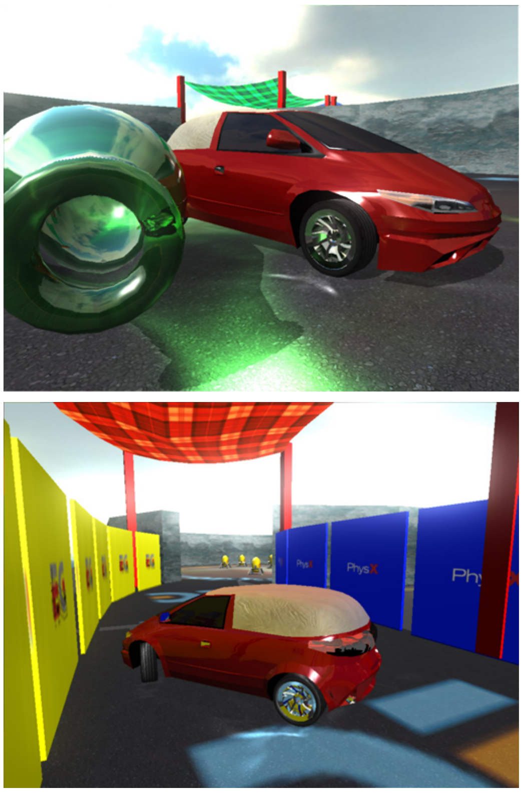


Figure 89: *Two scenes rendered in an interactive game environment, using global illumination methods including shadow mapping, environment mapping, diffuse indirect illumination, approximate ray-casting with distance impostors for reflections and caustics.*

10. Acknowledgement

This work has been supported by OTKA (ref. No.: T042735), GameTools FP6 (IST-2-004363) project, by the Spanish-Hungarian Fund (E-26/04), and by TIN2004-07451-C03-01 from the Spanish Government.

References

[AH93] Aupperle L., Hanrahan P.: A hierarchical illumination algorithms for surfaces with glossy reflection. *Computer Graphics (SIGGRAPH '93 Proceedings)* (1993), 155–162.

[AKDS04] Annen T., Kautz J., Durand F., Seidel H.-P.: Spherical harmonic gradients for mid-range illumination. In *Eurographics Symposium on Rendering* (2004).

[Bek99] Bekaert P.: *Hierarchical and stochastic algorithms for radiosity.* PhD thesis, University of Leuven, 1999.

[BF89] Buckalew C., Fussell D.: Illumination networks: Fast realistic rendering with general reflectance functions. *SIGGRAPH '89 Proceedings 23*, 3 (1989), 89–98.

[BGZ97] Bastos R., Goslin M., Zhang H.: Efficient radiosity rendering using textures and bicubic reconstruction. In *ACM-SIGGRAPH Symposium on Interactive 3D Graphics* (1997).

[Bjo04] Bjorke K.: Image-based lighting. In *GPU Gems*, Fernando R., (Ed.). NVidia, 2004, pp. 307–322.

[Bly05] Blythe D.: The Direct3D 10 system. In *SIGGRAPH '2006 Proceedings* (2005).

[BN76] Blinn J. F., Newell M. E.: Texture and reflection in computer generated images. *Communications of the ACM 19*, 10 (1976), 542–547.

[BSKS05] Barsi A., Szirmay-Kalos L., Szijártó G.: Stochastic glossy global illumination on the GPU. In *Proc. Spring Conference on Computer Graphics (SCCG '2005)* (Slovakia, 2005), Comenius University Press.

[Bun05] Bunnel M.: *GPU Gems II.* Addison-Wesley, 2005, ch. Dynamic Ambient Occlusion and Indirect Lighting, pp. 223–233.

[CCWG88] Cohen M. F., Chen S. E., Wallace J. R., Greenberg D. P.: A progressive refinement approach to fast radiosity image generation. In *Computer Graphics (SIGGRAPH '88 Proceedings)* (1988), pp. 75–84.

[CG85] Cohen M., Greenberg D.: The hemicube, a radiosity solution for complex environments. In *Computer Graphics (SIGGRAPH '85 Proceedings)* (1985), pp. 31–40.

[CHH02] Carr N., Hall J., Hart J.: The ray engine. In *Proc. of Graphics Hardware* (2002).

[CHH03] Carr N., Hall J., Hart J.: GPU algorithms for radiosity and subsurface scattering. In *Proc. of Workshop on Graphics Hardware* (2003), pp. 51–59.

[CHL04] Coombe G., Harris M. J., Lastra A.: Radiosity on graphics hardware. In *Graphics Interface* (2004).

[CS92] Cornette W., Shanks J.: Physical reasonable analytic expression for single-scattering phase function. *Applied Optics 31*, 16 (1992), 31–52.

[DBB03] Dutre P., Bekaert P., Bala K.: *Advanced Global Illumination.* A K Peters, 2003.

[Deb98] Debevec P.: Rendering synthetic objects into real scenes: Bridging traditional and image-based graphics with global illumination and high dynamic range photography. In *SIGGRAPH '98* (1998), pp. 189–198.

[DLW93] Dutre P., Lafortune E., Willems Y. D.: Monte Carlo light tracing with direct computation of pixel intensities. In *Compugraphics '93* (Alvor, 1993), pp. 128–137.

[DMK00] Dachille F., Mueller K., Kaufman A.: Volumetric global illumination and reconstruction via energy backprojection. In *Symposium on Volume Rendering* (2000).

[DS03] Dachsbacher C., Stamminger M.: Translucent shadow maps. In *Proceedings of the Eurographics Symposium on Rendering* (2003), pp. 197–201.

[DS05] Dachsbacher C., Stamminger M.: Reflective shadow maps. In *SI3D '05: Proc. of the 2005 Symp. on Interactive 3D Graphics and Games* (2005), pp. 203–231.

[EMD*05] Estalella P., Martin I., Drettakis G., Tost D., Devilliers O., Cazals F.: Accurate interactive specular reflections on curved objects. In *Proc. of VMV 2005* (2005).

[EMDT06] Estalella P., Martin I., Drettakis G., Tost D.: A GPU-driven algorithm for accurate interactive specular reflections on curved objects. In *Proceedings of the 2006 Eurographics Symposium on Rendering* (2006).

[Eve01] Everitt C.: *Interactive order-independent transparency.* Tech. rep., NVIDIA Corporation, 2001.

[FS05] Foley T., Sugerman J.: Kd-tree acceleration structures for a GPU raytracer. In *Proceedings of Graphics Hardware 2005* (2005).

[GD01] Granier X., Drettakis G.: Inremental updates for rapid glossy global illumination. *Computer Graphics Forum 20*, 3 (2001), 268–277.

[Gre84] Greene N.: Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications 6*, 11 (1984), 21–29.

[Gre03] Green R.: *Spherical Harmonic Lighting: The Gritty Details.* Tech. rep., 2003. http://www.research.scea.com/gdc2003/ spherical-harmonic-lighting.pdf.

[GRWS04] Geist R., Rasche K., Westall J., Schalkoff R.: Lattice-boltzmann lighting. In *Eurographics Symposium on Rendering* (2004).

[GSHG98] G. G., Shirley P., Hubbard P., Greenberg D.: The irradiance volume. *IEEE Computer Graphics and Applications 18*, 2 (1998), 32–43.

[GWWH03] Goodnight N., Wang R., Woolley C., Humphreys G.: Interactive time-dependent tone mapping using programmable graphics hardware. In *Rendering Techniques 2003: 14th Eurographics Symposium on Rendering* (2003), pp. 26–37.

[Hac04] Hachisuka T.: Final gathering on GPU. In *ACM Workshop on General Purpose Computing on Graphics Processors* (2004).

[Hac05] Hachisuka T.: High-quality global illumination rendering using rasterization. In *GPU Gems II*, Parr M., (Ed.). Addison-Wesley, 2005, pp. 615–634.

[Har02] Harris M. J.: Real-time cloud rendering for games. In *Game Developers Conference* (2002).

[Hay02] Hayden L.: *Production-Ready Global Illumination.* Tech. rep., SIGGRAPH Course notes 16, 2002. http://www.renderman.org/RMR/Books/ sig02.course16.pdf.gz.

[HDKS00] Heidrich W., Daubert K., Kautz J., Seidel H.-P.: Illuminating micro geometry based on precomputed visibility. In *SIGGRAPH 2000 Proceedings* (2000), pp. 455–464.

[HG40] Henyey G., Greenstein J.: Diffuse radiation in the galaxy. *Astrophysical Journal 88* (1940), 70–73.

[HH04] Hargreaves S., Harris M.: *Deferred Shading.* Tech. rep., http://download.nvidia.com/ developer/presentations/ 2004/6800_Leagues/ 6800_Leagues_Deferred_Shading.pdf, 2004.

[HL01] Harris M., Lastra A.: Real-time cloud rendering. *Computer Graphics Forum 20*, 3 (2001).

[HLHS03] Hasenfratz J.-M., Lapierre M., Holzschuch N., Sillion F. X.: A survey of realtime soft shadow algorithms. In *Eurographics Conference. State of the Art Reports* (2003).

[HSA91] Hanrahan P., Salzman D., Aupperle L.: Rapid hierachical radiosity algorithm. *Computer Graphics (SIGGRAPH '91 Proceedings)* (1991).

[IKSZ03] Iones A., Krupkin A., Sbert M., Zhukov S.: Fast realistic lighting for video games. *IEEE Computer Graphics and Applications 23*, 3 (2003), 54–64.

[JC95] Jensen H. W., Christensen N. J.: Photon maps in bidirectional Monte Carlo ray tracing of complex objects. *Computers and Graphics 19*, 2 (1995), 215–224.

[Jen96] Jensen H. W.: Global illumination using photon maps. In *Rendering Techniques '96* (1996), pp. 21–30.

[Jen01] Jensen H. W.: *Realistic Image Synthesis Using Photon Mapping.* AK Peters, 2001.

[JMLH01] Jensen H., Marschner S., Levoy M., Hanrahan P.: A practical model for subsurface light transport. *Computer Graphics (SIGGRAPH 2001 Proceedings)* (2001).

[KA06] Kontkanen J., Aila T.: Ambient occlusion for animated characters. In *Proceedings of the 2006 Eurographics Symposium on Rendering* (2006).

[Kaj86] Kajiya J. T.: The rendering equation. In *Computer Graphics (SIGGRAPH '86 Proceedings)* (1986), pp. 143–150.

[KAMJ05] Kristensen A., Akenine-Moller T., Jensen H.: Precomputed local radiance transfer for real-time lighting design. In *SIGGRAPH 2005* (2005).

[Kel97] Keller A.: Instant radiosity. In *SIGGRAPH '97 Proceedings* (1997), pp. 49–55.

[Kin05] King G.: Real-time computation of dynamic irradiance environment maps. In *GPU Gems II*, Parr M., (Ed.). Addison-Wesley, 2005, pp. 167–176.

[KK03] Kollig T., Keller A.: Efficient illumination by high dynamic range images. In *Eurographics Symposium on Rendering* (2003), pp. 45–51.

[KM00] Kautz J., McCool M.: Approximation of glossy reflection with prefiltered environment maps. In *Graphics Interface* (2000).

[KMN*02] Kanungo T., Mount D., Netanyahu N., Piatko C., Silverman R., Wu A.: An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Trans. Pattern Analysis and Mach. Int. 24*, 7 (2002), 881–892.

[KSS02] Kautz J., Sloan P., Snyder J.: Fast, arbitrary BRDF shading for low-frequency lighting using spherical harmonics. In *12th EG Workshop on Rendering* (2002), pp. 301–308.

[KVHS00] Kautz J., Vzquez P., Heidrich W., Seidel H.-P.: A unified approach to prefiltered environment maps. In *11th Eurographics Workshop on Rendering* (2000), pp. 185–196.

[LBC95] Languenou E., Bouatouch K., Chelle M.: Global illumination in presence of participating media with general properties. In *Eurographics Workshop on Rendering* (1995), pp. 69–85.

[LGB*03] Lensch H., Goesele M., Bekaert P., Kautz J., Magnor M., Lang J., Seidel H.-P.: Interactive rendering of translucent objects. *Computer Graphics Forum 22*, 2 (2003), 195–195.

[LK03] Lehtinen J., Kautz J.: Matrix radiance transfer. In *SI3D '03: Proceedings of the 2003 symposium on Interactive 3D graphics* (2003), pp. 59–64.

[Llo82] Lloyd S.: Least square quantization in pcm. *IEEE Transactions on Information Theory 28* (1982), 129–137.

[LSK05] Lazányi I., Szirmay-Kalos L.: Fresnel term approximations for metals. In *WSCG 2005, Short Papers* (2005), pp. 77–80.

[LW93] Lafortune E., Willems Y. D.: Bi-directional path-tracing. In *Compugraphics '93* (Alvor, 1993), pp. 145–153.

[Max94] Max N. L.: Efficient light propagation for multiple anisotropic volume scattering. In *Eurographics Workshop on Rendering* (1994), pp. 87–104.

[MH84] Miller G. S., Hoffman C. R.: Illumination and reflection maps: Simulated objects in simulated and real environment. In *SIGGRAPH '84* (1984).

[MPM02] Mantiuk R., Pattanaik S., Myszkowski K.: Cube-map data structure for interactive global illumination computation in dynamic diffuse environments. In *International Conference on Computer Vision and Graphics* (2002), pp. 530–538.

[MSC*05] Mendez A., Sbert M., Cata J., Sunyer N., Funtane S.: Real-time obscurances with color bleeding. In *ShaderX4: Advanced Rendering Techniques*, Engel W., (Ed.). Charles River Media, 2005.

[MSW04] Mei C., Shi J., Wu F.: Rendering with spherical radiance transport maps. *Computer Graphics Forum (Eurographics 04) 23*, 3 (2004), 281–290.

[NC02] Nielsen K., Christensen N.: Fast texture based form factor calculations for radiosity using graphics hardware. *Journal of Graphics Tools 6*, 2 (2002), 1–12.

[NDN96] Nishita T., Dobashi Y., Nakamae E.: Displaying of clouds taking into account multiple anisotropic scattering and sky light. In *SIGGRAPH '96 Proceedings* (1996), pp. 379–386.

[Neu95] Neumann L.: Monte Carlo radiosity. *Computing 55* (1995), 23–42.

[NRH03] Ng R., Ramamoorthi R., Hanrahan P.: All-frequency shadows using non-linear wavelet lighting approximation. *ACM Trans. Graph. 22*, 3 (2003), 376–381.

[ODJ04] Ostromoukhov V., Donohue C., Jodoin P.-M.: Fast hierarchical importance sampling with blue noise properties. In *Proc. SIGGRAPH 2004* (2004).

[OLG*05] Owens J. D., Luebke D., Govindaraju N., Harris M., Krüger J., Lefohn A. E., Purcell T. J.: A Survey of General-Purpose Computation on Graphics Hardware. In *EG2005-STAR* (2005), pp. 21–51.

[Pat95] Patow G. A.: Accurate reflections through a z-buffered environment map. In *In Proceedings of Sociedad Chilena de Ciencias de la Computacion* (1995).

[PBMH02a] Purcell T., Buck I., Mark W., Hanrahan P.: Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics 21*, 3 (2002), 703–712.

[PBMH02b] Purcell T. J., Buck I., Mark W. R., Hanrahan P.: Ray tracing on programmable graphics hardware. *ACM Transactions of Graphics 21*, 4 (2002), 703–712.

[PDC*03] Purcell T., Donner C., Cammarano M., Jensen H. W., Hanrahan P.: Photon mapping on programmable graphics hardware. In *Proceedings of Graphics Hardware* (2003), pp. 41–50.

[PG04] Parr M., Green S.: *GPU Gems*. Addison-Wesley, 2004, ch. Ambient Occlusion, pp. 279–292.

[PMDS06] Popescu V., Mei C., Dauble J., Sacks E.: Reflected-scene impostors for realistic reflections at interactive rates. *Computer Graphics Forum (Eurographics'2006) 25*, 3 (2006).

[PSS99] Preetham A., Shirley P., Smits B.: A practical analytic model for daylight. In *SIGGRAPH '99 Proccedings* (1999), pp. 91–100.

[REK*04] Riley K., Ebert D., Kraus M., Tessendorf J., Hansen C.: Efficient rendering of atmospheric phenomena. In *Eurographics Symposium on Rendering* (2004), pp. 374–386.

[RH01] Ramamoorthi R., Hanrahan P.: An efficient representation for irrandiance environment maps. *SIGGRAPH 2001* (2001), 497–500.

[RHS06] Roger D., Holzschuch N., Sillion F.: Accurate specular reflections in real-time. *Computer Graphics Forum (Eurographics'2006) 25*, 3 (2006).

[RTJ94] Reinhard E., Tijssen L. U., Jansen W.: Environment mapping for efficient sampling of the diffuse interreflection. In *Photorealistic Rendering Techniques*. Springer, 1994, pp. 410–422.

[Sbe96] Sbert M.: *The Use of Global Directions to Compute Radiosity*. PhD thesis, Catalan Technical University, Barcelona, 1996.

[Sbe97] Sbert M.: *The Use of Global Random Directions to Compute Radiosity: Global Monte Carlo Techniques*. PhD thesis, Universitat Politecnica de Catalunya, Barcelona, Spain, 1997. Available from http://ima.udg.es/ mateu.

[Sch93] Schlick C.: A customizable reflectance model for everyday rendering. In *Fourth Eurographics Workshop on Rendering* (1993), pp. 73–83.

[SCH03] Sen P., Cammarano M., Hanrahan P.: Shadow silhouette maps. *ACM Trans. Graph. 22*, 3 (2003), 521–526.

[SD02] Stamminger M., Drettakis G.: Perspective shadow maps. In *SIGGRAPH 2002* (2002), pp. 557–562.

[SHHS03] Sloan P., Hall J., Hart J., Snyder J.: Clustered principal components for precomputed radiance transfer. In *SIGGRAPH 2003* (2003).

[Shi91] Shirley P.: Time complexity of Monte-Carlo radiosity. In *Eurographics '91* (1991), Elsevier Science Publishers, pp. 459–466.

[SK99a] Szirmay-Kalos L.: *Monte-Carlo Methods in Global Illumination*. Institute of Computer Graphics, Vienna University of Technology, Vienna, 1999. http: //www.iit.bme.hu/˜szirmay/script.pdf.

[SK99b] Szirmay-Kalos L.: Stochastic iteration for non-diffuse global illumination. *Computer Graphics Forum 18*, 3 (1999), 233–244.

[SKAB03] Szirmay-Kalos L., Antal G., Benedek B.: Global illumination animation with random radiance representation. In *Rendering Symposium* (2003).

[SKAL05] Szirmay-Kalos L., Aszódi B., Lazányi I.: Ray-tracing effects without tracing rays. In *ShaderX4: Lighting & Rendering*, Engel W., (Ed.). Charles River Media, 2005.

[SKALP05] Szirmay-Kalos L., Aszódi B., Lazányi I., Premecz M.: Approximate ray-tracing on the GPU with distance impostors. *Computer Graphics Forum 24*, 3 (2005), 695–704.

[SKe95] Szirmay-Kalos (editor) L.: *Theory of Three Dimensional Computer Graphics*. Akadémia Kiadó, Budapest, 1995. http://www.iit.bme.hu/˜szirmay.

[SKL06] Szirmay-Kalos L., Lazányi I.: Indirect diffuse and glossy illumination on the GPU. In *SCCG 2006* (2006), pp. 29–35.

[SKM95] Szirmay-Kalos L., Márton G.: On convergence and complexity of radiosity algorithms. In *Winter School of Computer Graphics '95* (Plzen, Czech Republic, 14–18 February 1995), pp. 313–322. http//www.iit.bme.hu/˜szirmay.

[SKP98] Szirmay-Kalos L., Purgathofer W.: Global ray-bundle tracing with hardware acceleration. In *Rendering Techniques '98* (1998), pp. 247–258.

[SKS02] Sloan P., Kautz J., Snyder J.: Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In *SIGGRAPH 2002 Proceedings* (2002), pp. 527–536.

[SKSU05] Szirmay-Kalos L., Sbert M., Umenhoffer T.: Real-time multiple scattering in participating media with illumination networks. In *Eurographics Symposium on Rendering* (2005), pp. 277–282.

[SMKY04] Slater M., Mortensen J., Khanna P., Yu I.: A virtual light field approach to global illumination. In *Computer Graphics International* (2004), pp. 102–109.

[Sob91] Sobol I.: *Die Monte-Carlo Methode*. Deutscher Verlag der Wissenschaften, 1991.

[SP05] Shah M., Pattanaik S.: *Caustics Mapping: An Image-Space Technique for Real-Time Caustics*. Tech. Rep. CS-TR-50-07, University of Central Florida, August 2005.

[SSKS06] Szécsi L., Szirmay-Kalos L., Sbert M.: Light animation with precomputed light paths on the GPU. In *GI 2006 Proceedings* (2006).

[SSSK04] Sbert M., Szécsi L., Szirmay-Kalos L.: Real-time light animation. *Computer Graphics Forum (Eurographics 04) 23*, 3 (2004), 291–300.

[Tót06] Tóth B.: Real-time tone mapping on the GPU. In *BUTE Technical Report, 2006-05-01* (2006).

[TS00] Trendall C., Stewart A.: General calculations using graphics hardware, with application to interactive caustics. In *Rendering Techniques 2000* (2000), pp. 287–298.

[WBS03] Wald I., Benthin C., Slussalek P.: Interactive global illumination in complex and highly occluded environments. In *14th Eurographics Symposium on Rendering* (2003), pp. 74–81.

[WD06a] Wyman C., Dachsbacher C.: *Improving Image-Space Caustics Via Variable-Sized Splatting*. Tech. Rep. Technical Report UICS-06-02, University of Utah, 2006.

[WD06b] Wyman C., Davis S.: Interactive image-space techniques for approximating caustics. In *Proceedings of ACM Symposium on Interactive 3D Graphics and Games* (March 2006).

[WEH89] Wallace J. R., Elmquist K., Haines E.: A ray tracing algorithm for progressive radiosity. In *Computer Graphics (SIGGRAPH '89 Proceedings)* (1989), pp. 315–324.

[Wei03] Weisstein E.: *World of Mathematics*. Tech. rep., 2003. http://mathworld.wolfram.com/Curvature.html.

[WFA*05] Walter B., Fernandez S., Arbree A., Bala K., Donikian M., Greenberg D. P.: Lightcuts: A scalable approach to illumination. In *SIGGRAPH 2005* (2005).

[Wil78] Williams L.: Casting curved shadows on curved surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)* (1978), pp. 270–274.

[Wil01] Wilkie A.: *Photon Tracing for Complex Environments*. PhD thesis, Institute of Computer Graphics, Vienna University of Technology, 2001.

[WKB*02] Wald I., Kollig T., Benthin C., Keller A., Slussalek P.: Interactive global illumination using fast ray tracing. In *13th Eurographics Workshop on Rendering* (2002).

[WLHN06] Wang R., Luebke D., Humphreys G., Ng R.: Efficient wavelet rotation for environment map rendering. In *Proceedings of the 2006 Eurographics Symposium on Rendering* (2006).

[WS03] Wand M., Strasser W.: Real-time caustics. *Computer Graphics Forum 22*, 3 (2003), 611–620.

[WSP04] Wimmer M., Scherzer D., Purgathofer W.: Light space perspective shadow maps. In *EG Symposium on Rendering* (2004).

[Wym05] Wyman C.: An approximate image-space approach for interactive refraction. *ACM Transactions on Graphics 24*, 3 (July 2005), 1050–1053.

[ZIK98] Zhukov S., Iones A., Kronin G.: An ambient light illumination model. In *Proceedings of the Eurographics Rendering Workshop* (June 1998), pp. 45–56.