Calculating diffuse SH coefficients on GPU

Michal Staniszewski

December 13, 2005

Contents

1

Fundamentals of global illumination

The understanding of nature of light and its diffusion in surrounding environment is a key that will show us, how to properly simulate the phenomenon of global illumination. It's because of global illumination algorithms that are trying to imitate the properties of real light in modelled scenes.

In the following section I will describe pnly those fundamentals of global illumination, that will be needed in later chapters as mathematical tools in introduced algorithms.

More information about mathematics and physics behind global illumination can be found here [7].

1.1 BRDF

BRDF (which is an acronym for Bidirectional Reflectance Distribution Function) [4] is a function that is used as a good approximation for most of simulated materials. For BRDF function we assume that light reflects from the exact point it hits.

BRDF, f_r, defines a connection between reflected radiance L_r, and irradiation E_i:

$$f_r(x, \overrightarrow{\omega}', \overrightarrow{\omega}) = \frac{dL_r(x, \overrightarrow{\omega})}{dE_i(x, \overrightarrow{\omega}')} = \frac{dL_r(x, \overrightarrow{\omega})}{L_i(x, \overrightarrow{\omega}')(\overrightarrow{\omega}' \cdot \overrightarrow{n})d\overrightarrow{\omega}'}, \tag{1.1}$$

where $\overrightarrow{n}$ is a normal vector at the surface point x, ω is an outgoing direction of light and ω' is an incoming direction of light. Because of the fact that amount of the reflected radiance is proportional to incoming irradiation we must include solid angle in the denominator of the equation presented above.

BRDF is used to describe the light behavior whenever it hits the surface. If we know the exact amount of incoming radiation (from every direction) that reaches a certain point of the surface we can calculate amount of the radiation reflected in every direction:

$$L_r(x, \overrightarrow{\omega}') = \int_\Omega f_r(x, \overrightarrow{\omega}', \overrightarrow{\omega}) dE_i(x, \overrightarrow{\omega}') = \int_\Omega f_r(x, \overrightarrow{\omega}', \overrightarrow{\omega}) L_i(x, \overrightarrow{\omega}')(\overrightarrow{\omega}' \cdot \overrightarrow{n}) d\overrightarrow{\omega}' \quad (1.2)$$

1.2 Rendering equation

Rendering equation is used for obtaining amount of radiation, leaving a certain point in every direction. Outgoing radiation L_o is a sum of emitted radiation (special properties of the material) L_e and the reflected L_r.

$$L_o(x, \overrightarrow{\omega}) = L_e(x, \overrightarrow{\omega}) + L_r(x, \overrightarrow{\omega}). \quad (1.3)$$

Knowing the formula 1.2 we can rewrite the equation into the following:

$$L_o(x, \overrightarrow{\omega}) = L_e(x, \overrightarrow{\omega}) + \int_\Omega f_r(x, \overrightarrow{\omega}', \overrightarrow{\omega}) L_i(x, \overrightarrow{\omega}')(\overrightarrow{\omega}' \cdot \overrightarrow{n}) d\overrightarrow{\omega}'. \quad (1.4)$$

In order to apply the rendering equation to methods to be described in the further sections we need to transform them into a form that will include the dependencies between points that are mutually visible.

From now on we will integrate on a surface visible from point x. We will replace the integration over the sphere Ω directions by integration over the visible surface S using the following formula:

$$d\overrightarrow{\omega}' = \frac{(\overrightarrow{\omega} \cdot \overrightarrow{n}')dA'}{\|x' - x\|^2}, \quad (1.5)$$

where x' is a point visible from x, $\overrightarrow{n}'$ is a normal vector in the point x'. For simplification I am going to introduce G factor:

$$G(x, x') = \frac{(\overrightarrow{\omega} \cdot \overrightarrow{n}')(\overrightarrow{\omega}' \cdot \overrightarrow{n})}{\|x' - x\|^2}, \quad (1.6)$$

which as a result lets us rewrite the equation 1.4 in a more approachable form:

$$L_o(x, \overrightarrow{\omega}) = L_e(x, \overrightarrow{\omega}) + \int_S f_r(x, \overrightarrow{x'x}, \overrightarrow{\omega}) L_i(\overrightarrow{x'x}) V(x, x') G(x, x') dA'. \quad (1.7)$$

where $L_i(\overrightarrow{x'x})$ is radiation, coming from the point x' in the direction of point x, S is a set of all the points of the surface visible from x, and V describes the mutual visibility of x and x':

$$V(x, x') = \begin{cases} 1 & \text{if } x \text{ and } x' \text{ are mutually visible} \\ 0 & \text{in other case} \end{cases}$$

Rendering equation is a basis for the global illumination algorithms described in further sections. It is worth mentioning that from now I will call the factor $G(x, x')$ a cosine term and I will reduce it to simple $\max(0, \overrightarrow{\omega'} \cdot \overrightarrow{n})$ because of spherical lighting that we'll be using.

2

Diffused light simulation

Simulation of diffused light is considered to be the basis for the process of creating realistic pictures due to the fact that most of the objects found in the real world diffuse the light. In this section I am introducing a new approach to simulate global diffused light, using spherical harmonic functions. The functions will therefore be used to approximate the visibility (amount of the light that reaches the surface) to the selected points of our geometry.

It is recommended to know issues discussed in [5], [9] and [8] before reading the following section. In these publications authors will introduce you to fundamentals of global illumination using spherical harmonics functions.

2.1 Calculation of SH coefficients using GPU

Up to this time the most widely used method described in the papers referring to the problem of lighting based on spherical harmonic functions, was the raytracing method [5] [9] [8]. I am going to present an approach to calculate those coefficients using GPU and a rasterization algorithm. This method can be used in order to calculate SH coefficients for both vertices of models located on the scene and for the light source of those models as well.

2.2 Calculation of the geometry coefficients

Owing to the fact that we can only operate on discrete data, it is essential for us to somehow approximate the surface of our models.The most widely used method for surface approximation is firstly applying tesselation (a process of dividing the surface into polygons which share the same vertexes) and then triangularization (a process of dividing

the polygons into triangles). By doing so, we can receive a set of vertices, which we can then treat as a set of points, lying on the surface.

In case of geometry of our models, the algorithm is more complex and is divided into a few significant phases. Our task is to estimate BRDF function for diffused light for each of the vertices. In order to do so, we are obliged to create corresponding structures and auxiliary textures. Furthermore, we have to check the visibility from viewpoint of every single geometry vertex (in order to estimate the amount of the light that reaches it). Having estimated the BRDF function for the vertices, earlier approximated by a given amount of samples, we are now able to transform it into SH coefficients (using the Monte Carlo integration method). In order to use the whole potential modern video cards can offer, all of the processes described above should be performed simultaneously. Therefore, this means that the process of calculating the coefficients for one vertex should be completely independent and should not interfere in any way with the process of calculating other vertex's SH coefficients. The approach I am going to describe below calculates the necessary data for 1024 vertices contemporarily. If amount of the object vertices modulo 1024 is not equal to zero then so-called empty vertices are used in the calculations of the last data set. It is worth mentioning that one pass of the algorithm is able to calculate only four coefficients for each vertex (in order to gain more results, algorithm needs to be repeated). The algorithm can be used for different amount of samples for each vertex and different amount of vertices can be used simultaneously. Nevertheless, having conducted many quality/efficiency tests, and knowing the capabilities of modern video cards, it is safe for me to state that the best performance is obtained when simultaneously using 1024 samples for 1024 vertices. As a result, the algorithm operates on a floating-point texture of 1024x1024 resolution (16777216 bytes).

2.2.1 Initial preparations

In order to initially improve the algorithm, we can perform some precalculations in the initial phase and store them for future reference. First to be performed is the calculation of cosine term from the rendering equation 1.2. The method suggested below relies on calculating the whole cosine term for all possible dot products between the normal vector and the incoming light vector. Due to the fact that only half of the sphere, surrounding the polygon and directed to the direction of the normal vector, is important to us, the cosine term texture will be calculated only for the angles, ranging $\left[0, \frac{\pi}{2}\right]$. Because of the fact that we have earlier agreed the algorithm was supposed to work on 1024 samples, the texture has a size of 32x32 pixels.

Listing 2.1: Creating the texture responsible for the cosine term

```
unsigned char tex[32*32];

for(i=0,t=-1.0f;i<32;i++,t+=2.0f/31.0f)
{
    for(j=0,s=-1.0f;j<32;j++,s+=2.0f/31.0f)
    {
        float u=1-s*s-t*t;
        if(u<0)
        {
            tex[i*32+j]=0;
        }
        else
        {
            u=(float)sqrt(u);

            tex[i*32+j]=unsigned char(u*255.0); /* texture is written as
                bytes so we have to project [0,1] range into [0,255]*/
        }
    }
}
```

The next step of initial preparations is to create textures which would then store the basis harmonic spherical functions. In this case we need to calculate the function for the whole sphere, meaning $[0, \pi] \times [0, 2\pi]$ range. Listing 2.2 calculates 16 basis functions (in order to calculate 16 SH coefficients). For that we are going to use 4 floating-point textures of 32x64 resolution (it is an optimal resolution for 1024 samples). In order to calculate basis functions SH function described in [5] will be used.

Listing 2.2: Creating the textures, storing spherical harmonic basis functions. SH function is available in [5]

```
    for(j=-i;j<=i; ++j)
    {
        int index = i*(i+1)+j;
        int sh_index=index/4;
        for(t=0,k=0;k<32;t+=PI/31.0f,k++)
            for(s=0,l=0;l<64;s+=(2*PI)/63.0f,l++)
                SH_Functions[sh_index][(k*64+l)*4+index%4] = SH(i,j,t,s);
    }
}
```

In order to properly transform any two-dimensional function into SH coefficients, we need to associate every point of our function to a corresponding point on the sphere. In

our algorithm, this association depends on the normal vector, in the direction of which the visibility is going to be sampled. Due to the fact basis functions 2.2 are going to be sampled only owing to the spherical coordinates, we need to surround every vertex with a sphere, which will provide us with information regarding the θ and ϕ angles. Listing 2.3 shows how to create buffers, containing sphere geometry. The sphere will have the θ and ϕ angles' values stored on each of its vertices as color values (in the R and G elements, we store 0 in B element). Due to the fact the color is going to be interpolated, we may use lower sphere geometry fidelity in order to accelerate the calculations (in the listing 2.3 it is 16×32 which gives 512 sphere vertices).

Listing 2.3: Creating the sphere geometry buffer with additional information regarding the θ and ϕ angles

```
#define SPHERE_PHI      32
#define SPHERE_THETA    16

for(i=0,k=0,s=0;i<SPHERE_THETA+1;i++,s+=1.0f/(float)(SPHERE_THETA))
{
    for(j=0,t=0;j<SPHERE_PHI+1;j++,t+=1.0f/(float)(SPHERE_PHI))
    {
        float phi=2*t*PI;
        float theta=s*PI;

        sphere_buf[(i*(SPHERE_PHI+1)+j)*6+0]=(float)(sin(theta)*cos(phi));
        sphere_buf[(i*(SPHERE_PHI+1)+j)*6+1]=(float)(sin(theta)*sin(phi));
        sphere_buf[(i*(SPHERE_PHI+1)+j)*6+2]=(float)cos(theta);

        sphere_buf[(i*(SPHERE_PHI+1)+j)*6+3]=t;
        sphere_buf[(i*(SPHERE_PHI+1)+j)*6+4]=s;
        sphere_buf[(i*(SPHERE_PHI+1)+j)*6+5]=0;

        if(j<SPHERE_PHI && i<SPHERE_THETA)  /*create connections between
            the vertices*/
        {
            sphere_ind[k++]=(i*(SPHERE_PHI+1)+j);
            sphere_ind[k++]=((i+1)*(SPHERE_PHI+1)+j);
            sphere_ind[k++]=((i+1)*(SPHERE_PHI+1)+(j+1));
            sphere_ind[k++]=(i*(SPHERE_PHI+1)+(j+1));
        }
    }
}
```

2.2.2 First phase - drawing the spherical coordinates and visibility from each vertex

In order to find out how much light reaches every vertex of our geometry, we need to draw the whole scene from viewpoint of every vertex. We are going to avail ourselves of hemicube and hemisphere issues which have been often discussed in the computer graphics literature [5] [6] [1].

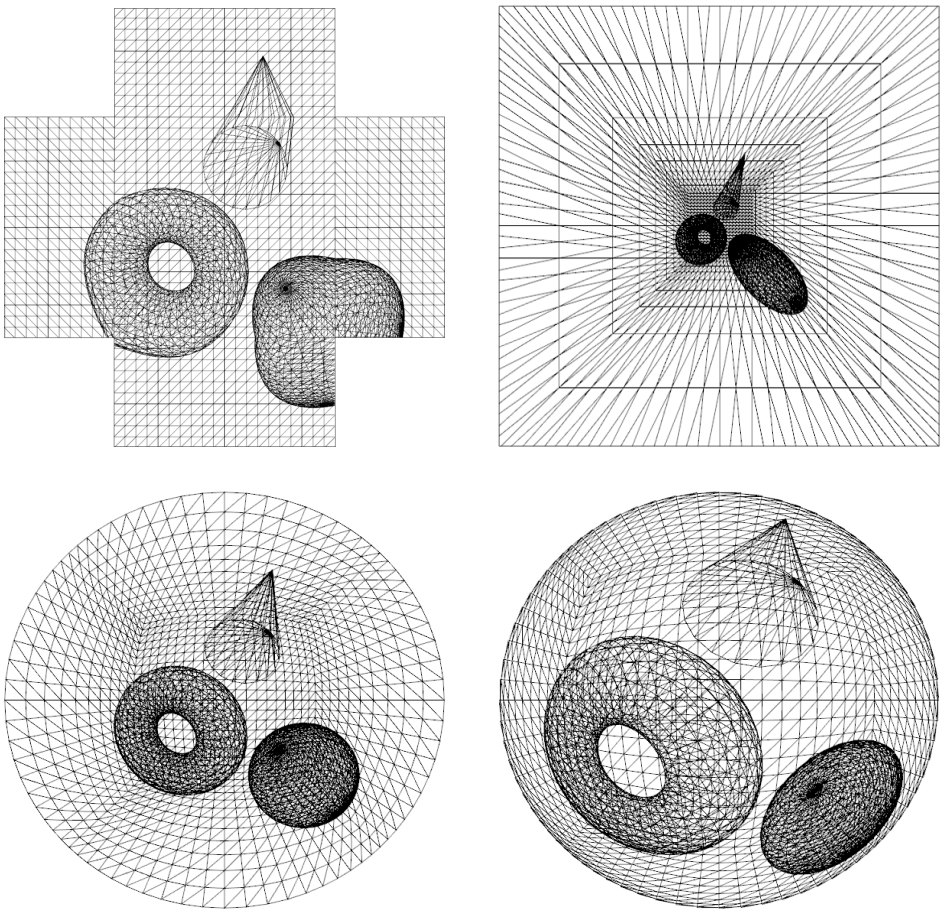

Figure 2.1: Comparison of different kinds of projections. Starting from the top left figure cube projection has been presented (perspective with an angle of view equal to 90°), perspective with an angle of view equal to 165°. In the bottom left row we can see paraboloid projection and stereographic projection.

Due to the fact that we are going to integrate the received visibility in the next phase

using the Monte Carlo method, we need to choose such a projection on the surrounding sphere in which the samples would be the most uniformly arranged. In order to find such a uniform sample arrangement on a sphere it is recommended to use the ray-tracing method[5]. Unfortunately, modern video cards are not able to perform that task with an acceptable speed. We will have to approximate the visibility from the vertex viewpoint, using rasterization. In order to do so, a proper projection of the modelled scene onto a sphere is required.

A basic approach would be to use a hemicube instead of a hemisphere (figure 2.1, left bottom corner). Unfortunately, the method mentioned above requires quintuple redrawing of the whole scene for each vertex. It is a requirement we cannot afford. Additionally, it turns out that other types of projections offer more uniform sample on the hemisphere arrangement. [10].

The next temporary solution would be to use the perspective projection for a wide angle of view equal to $(160° - 179°)$ (figure 2.1, right upper corner). Such a method causes a very improper sample arrangement - the integration concentrates on the light coming from the sides, not from the front of the observer (what is illogical, by the way). It is still worth mentioning that it is the fastest method and does not require additional computations like projecting with vertex shaders.

Another method is called paraboloid projection (figure 2.1 left bottom corner). A considerable improvement of the sample arrangement can be noticed, yet the samples are still concentrated on the sides. Stereographic projection (figure 2.1 right bottom corner) seems to be solving the uniform sample arrangement problem. The only requirement is to use an applicable projection vertex shader (Listing 2.4), which is going to perform the projection process.

Listing 2.4: A program written in GLSL language, performing the stereographic projection

```
uniform float n; /*distance of the near clipping plane*/
uniform float f; /*distance of the far clipping*/

void main(void)
{
    vec4 picoVertex = gl_ModelViewProjectionMatrix * gl_Vertex;
    /*normalize vector to the camera space*/
    vec3 normalizedVertex=normalize(picoVertex.xyz);
    /*normalize vertex - points will be mapped onto a sphere of 1.0 radius
       */
    gl_Position=vec4( normalizedVertex.x, normalizedVertex.y, (-2.0*
       picoVertex.z-f-n)/(f-n), 1.0); /*x and y coordinates are extracted
       from the sphere, z coordinate is tested for the clipping surfaces*/
    gl_FrontColor=gl_Color; /*move the vertex color to a part of the shader
        */
}
```

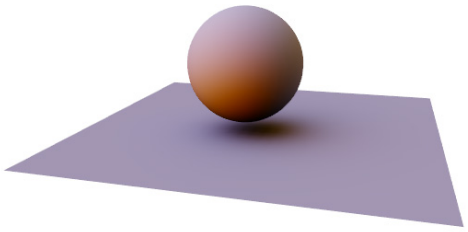

Figure 2.2: Figure showing test scene with geometry that we are going to analyze the visibility for.

In order to draw the visibility we are going to use only one element - B. R and G elements will store the spherical coordinates of the sphere, surrounding the vertex. Those coordinates are essential for associating the harmonic spherical basis functions to the corresponding visibility samples at current visibility stage, what we are going to do in the next phase.

2.2.3 Second phase - multiplicative stage

In the second phase (which I called multiplicative stage), a part of the rendering equation 1.2, located after the integral sign, is going to be performed.

Our task is to multiply vertex visibility v by cosine term C and a corresponding basis function from the spherical harmonics basis. The whole process is reduced to drawing a square in a frame-buffer of a 1024×1024 resolution (for assumed 1024 samples per vertex), mapping him with corresponding textures and executing the shader 2.5 responsible for SH projection.

Listing 2.5: Program written in GLSL language, responsible for spherical harmonic projection

```
uniform sampler2D visibilitySampler; /*visibility from vertex*/
uniform sampler2D cosineTermSampler; /*cosine term*/ uniform
sampler2D SHSampler; /*spherical harmonic basis functions - four of
them*/

void main(void)
{
    vec4 texel=texture2D(visibilitySampler,vec2(gl_TexCoord[0]));
    vec4 color=vec4(texel.b); /*blue color stores visibility from the
        vertex viewpoint*/
```

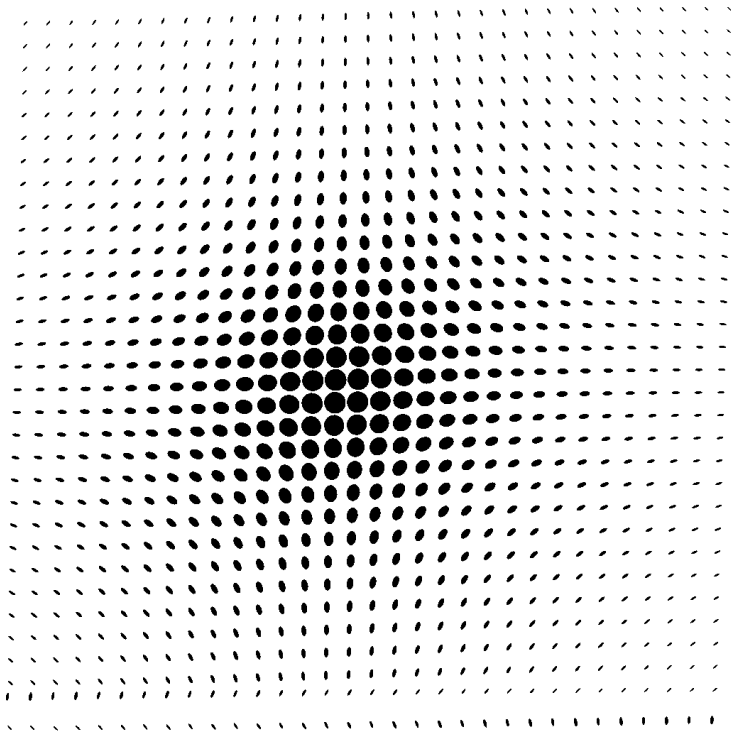

Figure 2.3: Visibility from the floor viewpoint for the scene 2.2.

```
    vec4 texel2=vec4(texture2D(cosineTermSampler,vec2(gl_TexCoord[0])*vec2
        (32.0)).r);
    /*texture with cosine term has a 32x32 size so we need to multiply
        vertex UV coordinates by 32, taking texture repeating on the edges (
        GL_REPEAT) into account */
    color*=texel2;
    texel2=texture2D(SHSampler,vec2(texel)); /*basis functions are sampled,
         using the spherical coordinates that have been stored in r (theta)
        and g (phi) elements in the previous phase.*/
    gl_FragColor=texel2*color; /*do the projection*/
}
```

It is worth mentioning that it is only possible to acquire 4 SH coefficients from one multiplicative phase. Still, visibility calculated in the earlier phases can be used to calculate the coefficients from the upper bands. That's why the process of calculating upper coefficients should be started from the second phase (visibility determination is the most time-consuming process).

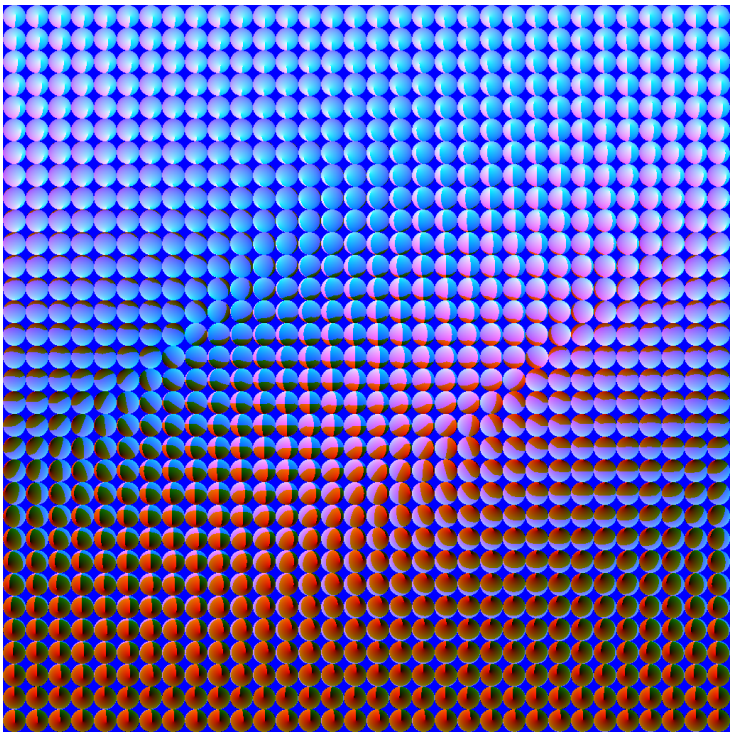

Figure 2.4: Visibility from the sphere viewpoint for scene 2.2 is stored in B element. In R and G elements we store the spherical coordinates of the hemisphere, surrounding the vertex.

2.2.4 Third phase - Monte Carlo integration in four passes

Following the rendering equation1.2, we find that we need to integrate the incoming light to sphere that surrounds the vertex. In our case where we use spherical harmonics functions, in order to bind the lighting environment to the visibility of the vertex we need to perform spherical harmonics projection. Unfortunately, lack of a visibility function that could be described using a formula forces us to use the Monte Carlo integration.

Listing 2.6: Program written in GLSL language, which integrates given texture with a given sampler: previousPass. During the first and third pass we have 8 samples (texelKernels), during the second and fourth pass only four. Additionally, during the last pass the acquired sum is to be multiplied by a proportion of the hemisphere area to amount of the samples per vertex (which equals 1024).

```
uniform sampler2D previousPass;
uniform vec2 texelKernels[8];
```

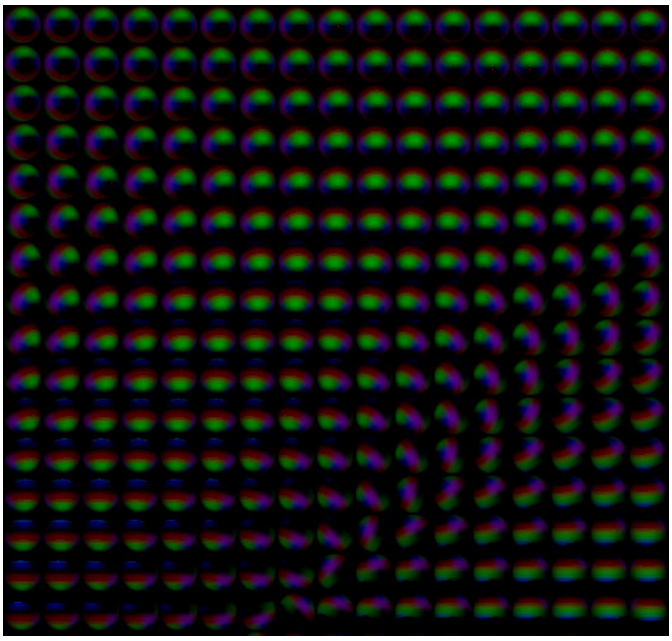

Figure 2.5: A fragment of the texture created by program 2.5. R,G,B and A (invisible) elements store the rendering equation results, multiplied by harmonic spherical basis functions (at the figure provided these are the functions from the fourth band.

```
void main(void)
{
    vec4 sum=vec4(0,0,0,0);
    for (int i = 0; i <8; i++)
    {
        sum += texture2D(previousPass,vec2(gl_TexCoord[0])+texelKernels[i])
            ;
    }
    gl_FragColor=sum;
}
```

Below is a pseudocode of Monte Carlo integration algorithm, using a graphics card. it is possible to execute the whole algorithm, using only two buffers, swapped with every step. Still, I decided to use N buffers (where N is equal to the amount of algorithm passes), in order to make the code clearer.
In case of the integration algorithm using only four passes, in the initialization phase $N = 1$. Let us asume that texture with zero index is a result of the previous (multiplicative) stage.

Peform following steps until $N = 5$:

1. **set framebuffer to Nth** - setting Nth framebuffer causes every OpenGL primitive to be stored in it.

2. **bind active framebuffer to Nth texture** - performs ,,conversion on fly" of the framebuffer to the OpenGL texture. We will use Nth texture in the next pass.

3. **set (N-1)th texture as active** - we are going to map the texture using everything that the frame buffer stored from the last algorithm pass.

4. **start Nth shader responsible for the integration** - due to the fact we integrate smaller and smaller areas integrating shaders are different for each pass.

5. **calculate texture coordinates offsets** - this process is required for correct summing of each texel. Data, regarding the coordinates offsets, should be sent to the shader.

6. **draw a rectangle of Nth frame buffer size** - we should not forget that with every pass the rectangle gets correspondingly smaller.

7. **stop Nth shader responsible for the integration** - having drawn the rectangle with summed texels turn off the Nth shader.

8. **increment N**

Listing 2.7: An example of Monte Carlo integration pseudocode implementation for the fourth integration pass

```
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, _4thPassBuffer); /*set
framebuffer to fourth*/

glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                          GL_COLOR_ATTACHMENT0_EXT,
                          GL_TEXTURE_2D, _4thPassBufferTexture, 0); /*bind active
                               framebuffer to fourth texture*/

glActiveTextureARB(GL_TEXTURE0_ARB); glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, _3rdPassBufferTexture); /*set the third
texture as active*/

_4thPassIntegrationProgram->executeProgram(); /*launch the fourth
shader responsible for the integration*/ uniformLocation =
glGetUniformLocationARB(_4thPassIntegrationProgram->program,
"previousPass"); glUniform1iARB(uniformLocation,0); /*send
information regarding the third texture in a sampler*/
```

```
for(int j=0;j<4;j++)
{
    texelKernels[j][0]=0;
    texelKernels[j][1]=1.0f/128.0f*(j-2); /*calculate the texture
        coordinates offsets*/
}

uniformLocation =
glGetUniformLocationARB(_4thPassIntegrationProgram->program,
"texelKernels");
glUniform2fvARB(uniformLocation,4,(GLfloat*)texelKernels); /*send
coordinates offsets to the shader*/

drawQuad(32,32); /*draw a rectangle of a size of the fourth
frame-buffer*/

_4thPassIntegrationProgram->stopProgram(); /*stop the fourth program
written in GLSL language responsible for the integration*/
```

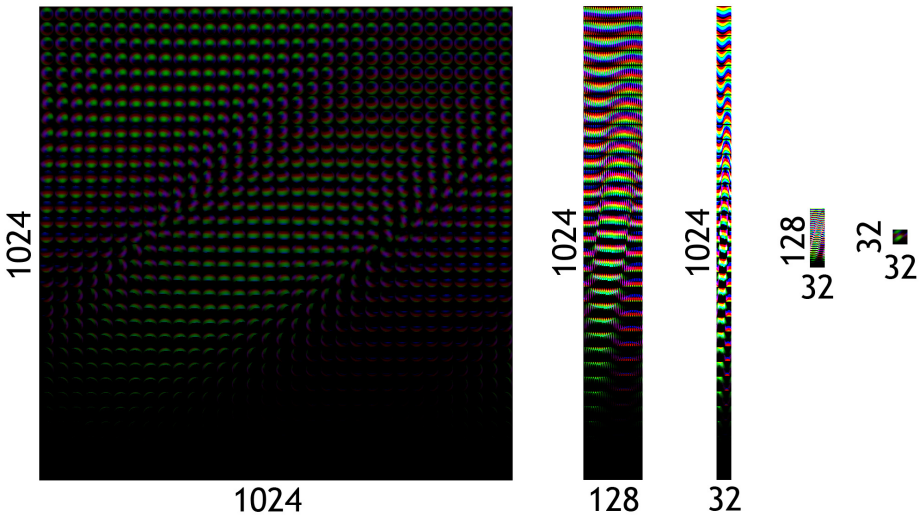

Figure 2.6: Process of the texture integration 2.5 using the program 2.7 and the algorithm 2.2.4 with smaller and smaller frame-buffers.

2.3 Calculating light coefficients

The process of calculating SH coefficients for light is very similar to the one for geometry. The basic difference is that the algorithm needs to be divided into three phases, in which three elements of the light color: R, G, B are analyzed separately. This time I am going to concentrate only on one element.

Using the method, described in the point 2.2, we check the visibility from each of the vertices of our geometry. If we want to introduce the lighting environment to SH coefficients we choose one point (usually located in the center of the coordinate system) and treat it as if it was not covered by any other geometry (in other words, we assume light can reach it from all directions). In order to assert uniform arrangement of probability for directions of the incoming light we should eliminate the cosine term from the calculations as well.

It is relatively easy to predict that we are going to approximate the intensity of incoming light that reaches the unit sphere, surrounding the chosen point in space using SH coefficients. Therefore we can keep the previous approach, assuming we swap the visibility function with the lighting environment intensity function (by mapping the lighting environment cubemap texture onto the unit sphere).

2.4 Simulating diffused reflections using GPU

Having calculated geometry and light coefficients, we need to perform a simple dot product directly on the GPU. Therefore, we need to write a corresponding shader in GLSL language which would draw the material geometry with light diffusing attributes as a result of its execution. SH coefficients of the geometry will be provided as vertex attributes (each attribute has 4 floating-point elements). Light coefficients will be provided as uniform variables.

Listing 2.8: Program written in GLSL language (vertex shader), calculating the dot product of two SH vectors, consisting of 16 coefficients

```
uniform vec3 diffuseColor; /*material color (constant BRDF
function)*/ uniform vec4 SHMatrixR[4]; /*16 coefficients for the
lighting environment. Following tables respond to R, G and B
elements */ uniform vec4 SHMatrixG[4]; uniform vec4 SHMatrixB[4];

void main(void)
{
    vec3 diffuseSH;

    gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix *
```

```
        gl_Vertex; /*transform the geometry */

    vec4 tempDiff;
    vec4 SHone=vec4(1.0); /* auxiliary variable used to sum up individual
        dot products*/

    tempDiff.x=dot(gl_MultiTexCoord0,SHMatrixR[0]); /*dot function performs
         dot product of two 4-dimensional vectors*/
    tempDiff.y=dot(gl_MultiTexCoord1,SHMatrixR[1]);
    tempDiff.z=dot(gl_MultiTexCoord2,SHMatrixR[2]);
    tempDiff.w=dot(gl_MultiTexCoord3,SHMatrixR[3]);
    diffuseSH.x=dot(tempDiff,SHone); /*dot product of a vector of which
        elements are equal to zero sums elements of the first vector*/
    tempDiff.x=dot(gl_MultiTexCoord0,SHMatrixG[0]);
    tempDiff.y=dot(gl_MultiTexCoord1,SHMatrixG[1]);
    tempDiff.z=dot(gl_MultiTexCoord2,SHMatrixG[2]);
    tempDiff.w=dot(gl_MultiTexCoord3,SHMatrixG[3]);
    diffuseSH.y=dot(tempDiff,SHone);
    tempDiff.x=dot(gl_MultiTexCoord0,SHMatrixB[0]);
    tempDiff.y=dot(gl_MultiTexCoord1,SHMatrixB[1]);
    tempDiff.z=dot(gl_MultiTexCoord2,SHMatrixB[2]);
    tempDiff.w=dot(gl_MultiTexCoord3,SHMatrixB[3]);
    diffuseSH.z=dot(tempDiff,SHone);

    diffuseSH*=diffuseColor; /*in the end multiply it by the material color
        */
    gl_FragColor = diffuseSH;
}
```

3

Global illumination implementation in graphics engines

The process of implementing the techniques, described in the sections 2 and **??** in a typical graphics engine is not difficult but requires some essential assumptions to be made. First of all we need to decide how many SH coefficients we are going to use. In order to use GLSL structures in the best possible manner it is advised they be multiplicities of four.

Mutual visibility in the scene is another problem we have to face - in order to optimize the engine and obtain more control over it we can design a system that would distinguish the occluded objects from the occluding ones. As a result, it would allow us to detect which objects should the global illumination be calculated for.

Up to now we have been dealing with scenes in which all objects, for which SH lighting was used, were static. It was only the lighting environment that would change (rotate around the scene). In the following section I am going to introduce an approach that will let us animate the scene models, using the SH coefficients interpolation.

Also, I am going to deal with the problem of storage of the calculated SH coefficients in this section. In the end I am going to present an example of implementation of the algorithms, presented in this paper, with a well-known graphics software called Maya using an engine - a plug-in called picoEngine v2.5. I am going to present the results as well.

3.1 Visibility and a system distinguishing the occluded objects from the occluding ones

Visibility is an important issue we have to deal with when designing a system that employs SH lighting. Listing 2.4 includes parameters, controlling the near and the far clipping

planes in a stereographic projection. They allow us to remove the geometry, that is located either too close, causing artifacts like blackouts (Figure 3.1), or is occluding all incoming light causing artifacts as well (Figure 3.2). Proper adjustment of near and far clip plane parameters enables us to avoid those problems. Another improvement would be to use gradient during the visibility sampling process. Objects, located further from the illuminated point would be drawn with a color near 1.0 and - as a result - they would be treated as less important during the process of calculating the visibility. Models located closer would be drawn using a color near 0.0. As a result they would be more significant for the process of calculating the visibility. In the algorithms, described in the section 2, all objects are drawn with 0.0 color, meaning they are equally significant when calculating the SH coefficients.

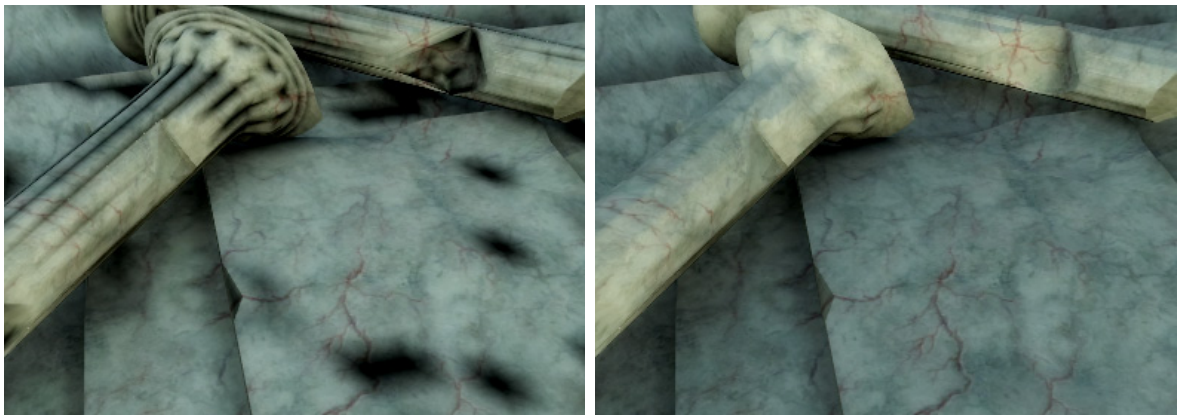

Figure 3.1: Blackout artifacts, appearing with the near clipping plane parameter being incorrectly set (left figure), and the same scene without the artifacts (right figure).

When modelling the lighting in 3D scenes it is often needed to obtain more control over the lighting than in the real world (like in case of the lighting technicians in the theater or a movie set). The simplest way to achieve that is to assume which objects are going to cast shadows on the scene and which ones are going to receive those shadows. Those operations are a part of the system that distinguishes the occluded objects from the occluding ones. We call the occluded objects those models that the SH coefficients calculation process is performed for. An occluded object must be associated to a set of occluding objects. If the occluding objects set is empty then the occluded object is not going to receive any shadows (from itself as well). As a result of this rule, contemporarily the occluded object can be a occluding object for itself.

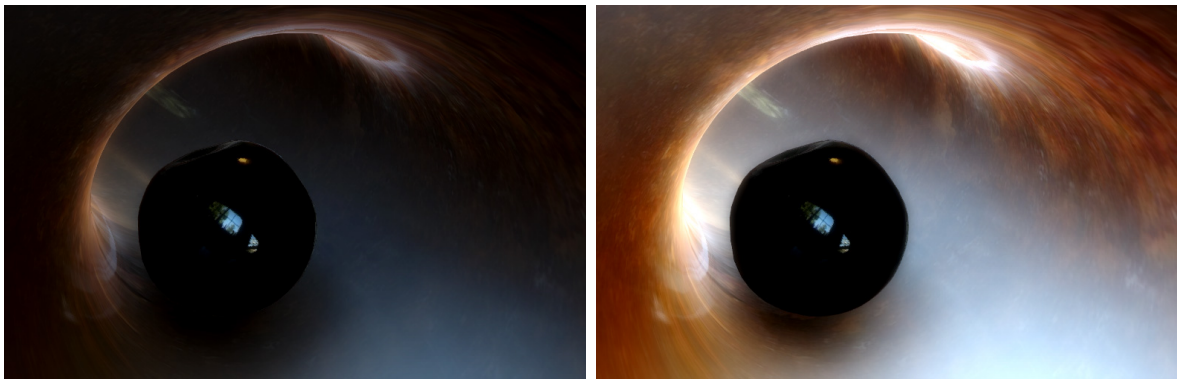

Figure 3.2: With the far clipping plane parameter being incorrectly set, the tunnel can see its wall located on the back side, causing loss of visibility (left figure). Correctly illuminated scene is shown on the right figure.

3.2 SH coefficients storage

Due to the time-consuming nature of the coefficients calculations, it is advised to store the calculated SH lighting. The main problem is to choose amount of the coefficients. Figure 3.3 is a comparison of the same model, illuminated by different amounts of the coefficients. Every four of the coefficients takes 16 bytes for every vertex. In order to save some space we can try to store the vertices in the same way the colors are stored in RGBE format. Owing to the fact that RGBE is not a lossless format it is advised to group the coefficients so that the precision losses are the lowest. As a result, it is best to group the coefficients by bands (eg. first coefficient of a certain vertex with first coefficient of another vertex). Variation of coefficients is lower in the same band.

3.3 Animation of the diffused lighting using the SH coefficients interpolation

Due to the dot product operator being a linear operation we can linearly interpolate corresponding coefficients pairs. Thanks to that we are able to simulate animated global illumination for diffused reflection. Alas, this approach is going to be only useful for predefined animations. It is not going to work with interactive events. Still, in software using interactive 3D visuals (games, presentations) parts of the animations are often repeated in few second loops (eg. character movement) or even are limited (eg. wheels turning in a car). In such cases usage of SH interpolated lighting seems to be a very good solution.

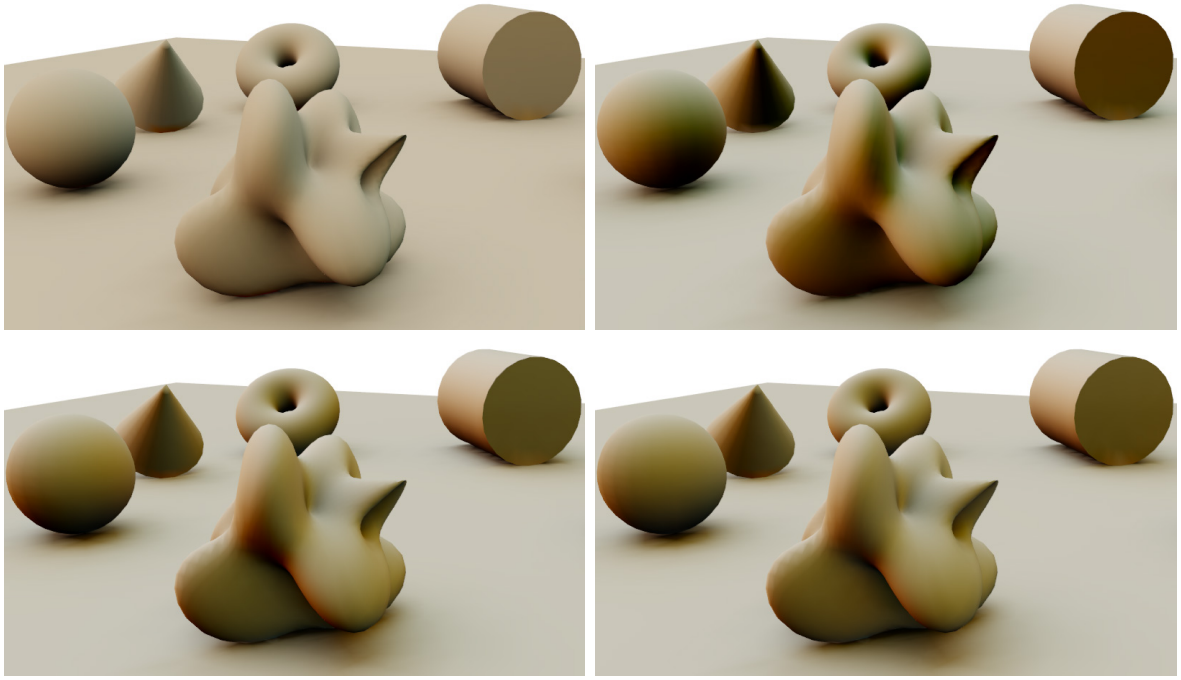

Figure 3.3: Comparison of the lighting simulation for diffused reflection, using 4,8,12,16 coefficients (counting from the left upper figure). The most significant color difference can be noticed between the usage of four and eight coefficients. With bigger amount of available coefficients the only noticeable difference is sharpness of the shadows.

Due to the fact that a pair of coefficients, coming from the same band, is essential for the process of interpolation we need to send two times more coefficients to the shader. Unfortunately, the capabilities of modern video cards, even of the latest ones, do not allow us to send a arbitrary (limited by amount of free memory) amount of coefficient bands. Usually it is 40 bands [3] [2] what is enough, but in case of interpolation it is only going to be two times 20 bands. It is worth mentioning that the lighting itself is not enough to simulate realistic appearance of the models (in case we are using single-pass rendering) so amount of the bands is going to be limited even more. When planning the optimisation it is wise to first think how to use the GLSL language capabilities well. From the properties of dot product we know that we can interpolate the coefficients on the calculated color values both before and after applying the dot product of lighting environment. Listing 3.1 shows code written in GLSL language, responsible for calculating the diffused lighting for 8 interpolated coefficients.

Listing 3.1: Program written in GLSL language (a vertex shader) performing interpolation of 8 SH coefficients and SH projection on interpolated coefficients

```
uniform vec3 diffuseColor; uniform float SHblend; uniform vec4
SHMatrixR[2]; /*8 lighting environment coefficients for R
element*/ uniform vec4 SHMatrixG[2]; uniform vec4 SHMatrixB[2];

void main(void)
{
    vec3 diffuseSH;
    gl_Position = gl_ProjectionMatrix * gl_ModelViewMatrix * gl_Vertex;
    vec4 animatedSH[2];

    animatedSH[0]=mix(gl_MultiTexCoord0,gl_MultiTexCoord2,SHblend); /*
        function
    mix performs linear interpolation between coefficients set as
    texture coordinates. SHblend is value of the interpolation, ranging
        [0,1]*/
    animatedSH[1]=mix(gl_MultiTexCoord1,gl_MultiTexCoord3,SHblend);

    diffuseSH.x=dot(animatedSH[0],SHMatrixR[0]);
    diffuseSH.x+=dot(animatedSH[1],SHMatrixR[1]);
    diffuseSH.y=dot(animatedSH[0],SHMatrixG[0]);
    diffuseSH.y+=dot(animatedSH[1],SHMatrixG[1]);
    diffuseSH.z=dot(animatedSH[0],SHMatrixB[0]);
    diffuseSH.z+=dot(animatedSH[1],SHMatrixB[1]);
    diffuseSH*=diffuseColor; /* finally multiply it by the surface color
        value*/

    gl_FragColor = vec4(diffuseSH,1.0);
}
```

When writing an engine that supports animated SH lighting, the basic question we have to answer is exactly how many SH keys should be generated within one second. We shall call a SH key a set of coefficients, respondent to a certain arrangement of models in the scene for a given moment. This problem cannot be solved using a mathematical equation because every animation is different. That is the reason for which the amount of SH keys should be determined individually for every scene. It usually ranges from 5 to 15 keys per second.

3.4 Implementation of global diffused and reflected illumination in Maya graphics software, using a sample plug-in engine called picoEngine v2.3

Maya is a software, or rather a system that allows us to create photorealistic graphics for off-line movies. Due to the increasing popularity of high-end games and existence of art, which aims at creating movies - realtime animations, called demos (and Machinima), Maya producers decided to release an interface, allowing programmers to write plug-ins that would further expand their product's capabilities in terms of realtime graphics. picoEngine plug-in (version 2.3) implements all algorithms described in my papers and fully integrates with Maya 6.0 system in order to gain resources like 3D models, textures and animations.

For instance, SH diffused and reflected lighting (max 16 coefficients for static scenes and max 8 coefficients for the animated ones), support for HDRI lighting environments in RGBE format, Fresnel equations for reflections dimming, system distinguishing the occluding objects from the occluded ones and tone-mapping have already been implemented.

Source code, documentation, project's goals and description of how the picoEngine engine works can be found at the web-site: http://www.plastic-demo.org.

picoEngine v2.3 plug-in requires minimum Maya 6.0 (Complete version) system and a ATI Radeon 9500 or NVIDIA geForce FX5 video card or better.

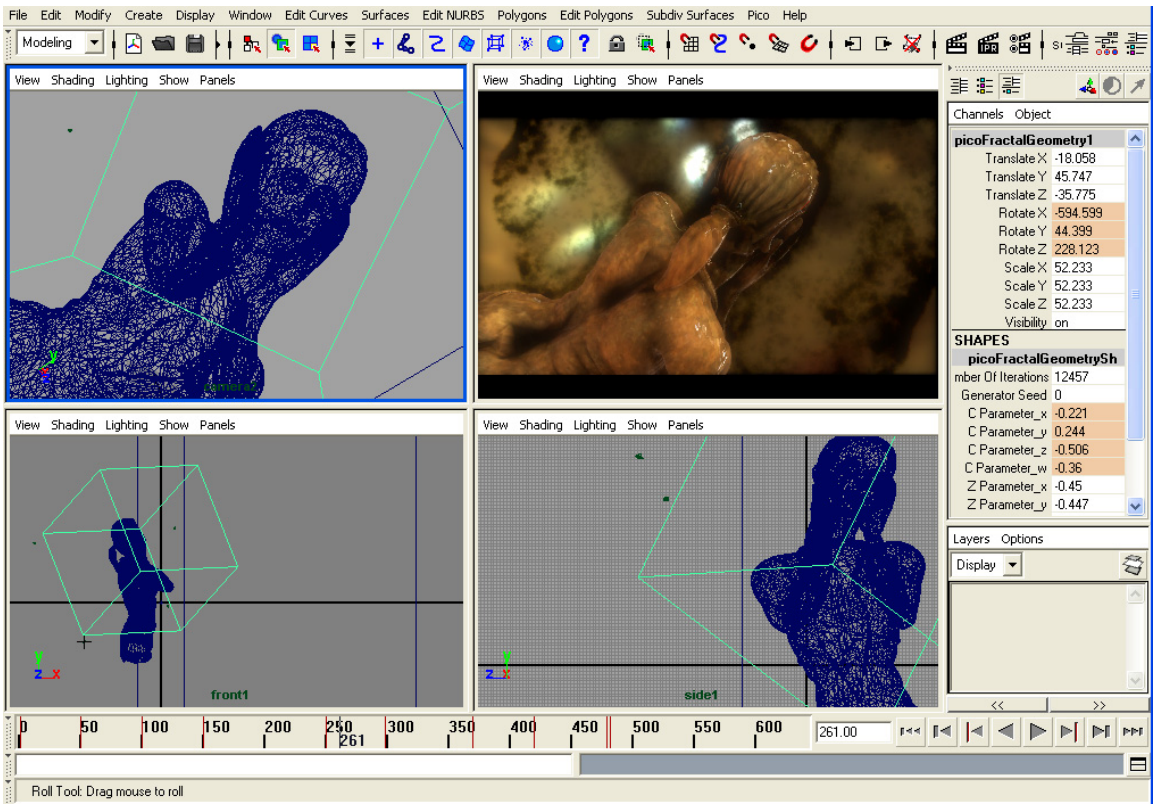

Figure 3.4: picoEngine v2.3 plugin in Maya 6.0 environment. Figure presents a model, illuminated by HDRI lighting environment (a courtesy of Paul Debevec) and a few post-processing effects applied

Bibliography

[1] Michael F. Cohen and Donald P. Greenberg. The hemi-cube: A radiosity solution for complex environments. 1985.

[2] ATI Corporation. Radeon x1800 product website, http://www.ati.com/products/radeonx1800/index.html. 2005.

[3] NVIDIA Corporation. geforce7 product website, http://www.nvidia.com/page/specs_gf7800.html. 2005.

[4] J. J. Hsia I. W. Grinsberg F. E. Nicodemus, J.C Richmond and T. Limperis. *Geometric considerations and nomenclature for reflectance*. Monograph 161, National Bureau of Standards (US), 1997.

[5] Robin Green. Spherical harmonic lighting: The gritty details. 2003.

[6] Anselmo Lastra Greg Coombe, Mark J. Harris. Radiosity on graphics hardware. 2003.

[7] Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping*. A. K. Peters, 2001.

[8] John Snyder Peter-Pike Sloan, Jan Kautz. Fast, arbitrary brdf shading for low-frequency lighting using spherical harmonics. 2003.

[9] John Snyder Peter-Pike Sloan, Jan Kautz. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. 2003.

[10] John Snyder and Don Mitchell. Sampling-efficient mapping of spherical images. 2001.