

Real-time Obscurances with Color Bleeding

(GPU Obscurances with Depth Peeling)

Àlex Méndez-Feliu, Mateu Sbert, Jordi Catà, Nicolau Sunyer and Sergi Funtané
GGG, IliA, Universitat de Girona

1 Introduction

The obscurance method is a powerful technique that simulates the effect of diffuse interreflections, i.e. radiosity, on an object at a much lower performance cost. Its main advantage lies in the fact that this technique considers only neighbouring interactions instead of attempting to solve all the global ones. Another advantage of this technique is that it is decoupled from direct illumination computation. In this article, we will show how obscurances can be used in a videogame environment, allowing realistic and fast illumination of the scene.

Radiosity techniques [Goral84] are commonly used to simulate diffuse global illumination -- although very powerful and increasingly faster [Bekaert98], they do not yet fulfil the requirements for fast and efficient real-time scene editing or rendering. With radiosity, the interaction of each surface in the scene with each other surface has to be (at least potentially) considered; the obscurance method [Zhukov98], [Iones03] being designed to offer a fast alternative to radiosity.

This technique can deal with any number of moving light sources with no added cost since the indirect illumination and direct illumination are effectively decoupled. The obscurance technique also allows the addition of color bleeding [Mendez03] to your lighting.

Obscurances have already been used in 3D computer games and animations in a simplified form commonly called ambient occlusions [Christensen03], [Pharr04], [Bunnel05]. In this article a new algorithm to compute obscurances using depth peeling [Everitt01] is presented. In addition we will discuss the real-time update of obscurances for moving objects within a scene.

2 Obscurances

The obscurance illumination model ([Zhukov98] and [Iones03]) has been defined to take account of the indirect indirect (diffuse) illumination while being totally decoupled from direct illumination. Within the obscurance model, the indirect illumination for a point P is defined as:

$$I(P) = \frac{1}{\pi} \times R(P) \times I_A \times \int_{\omega \in \Omega} \rho(d(P, \omega)) \cos \theta d\omega \quad (1)$$

where

- $d(P, \omega)$ is the distance between P and the next surface at direction ω
- $\rho(d)$ is a function that determines the magnitude of ambient light incoming from neighbourhood d , and taking values between 0 and 1.
- θ is the angle between direction ω and the normal at P .
- I_A is the ambient light intensity.
- $R(P)$ is the reflectivity at the point P .
- $\frac{1}{\pi}$ is a normalization factor such that if $\rho()=1$ over the whole hemisphere Ω , then $I(P)$ is $R \cdot I_A$.

Direct illumination can then be added to (1) to obtain the final illumination of the point.

As you can see in Figure 1, the function $\rho()$ increases with the distance d .

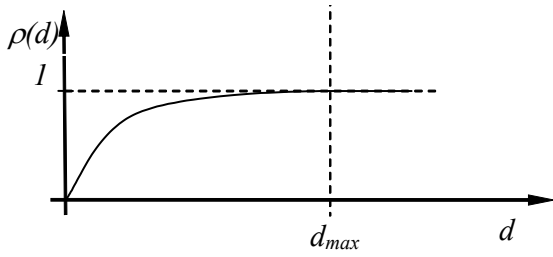


Figure 1. Function $\rho(\mathbf{d})$.

Function $\rho(d)$ will be equal to 1 for $d \geq d_{max}$. This means that we are not taking into account occlusions farther than d_{max} . We only take the “neighbouring” ones. The function used in this paper is $\sqrt{d/d_{max}}$ if $d < d_{max}$ and 1 otherwise.

The obscurance of the point P is then defined as:

$$W(P) = \frac{1}{\pi} \times \int_{\omega \in \Omega} \rho(d(P, \omega)) \cos \theta d\omega. \quad (2)$$

Since $0 \leq \rho(d) \leq 1$, we can assume that $0 \leq W(P) \leq 1$. The obscurance for a patch (in radiosity terms, a polygon from a subdivided mesh) 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 neighbouring polygons), while a value of 0 means that it is totally closed (or occluded by neighbouring polygons).

For a closed environment, the ambient light in (1) can be computed as the average light intensity in the scene using the following formula:

$$I_A = \frac{R_{ave}}{(1 - R_{ave})} \cdot \frac{\sum_{i=1}^n A_i E_i}{A_{total}}, \quad \text{where } R_{ave} = \frac{\sum_{i=1}^n A_i R_i}{A_{total}}, \quad (3)$$

where A_i , E_i and R_i are the area, emissivity and reflectivity of patch i , respectively, A_{total} 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. Since the obscurance computation only takes into account the neighbourhood of a patch (i.e. near than d_{max}), the computation can be done very efficiently. For further details see [Zhukov98] and [Iones03].

3 Color bleeding

The obscurance approach as presented in previous section 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. Here, we present a straightforward technique to account for this color bleeding, with no added computational cost.

The obscurances formula (2) is modified slightly to include the reflectivity term of the patches:

$$W(P) = \frac{1}{\pi} \times \int_{\omega \in \Omega} R(Q) \rho(d(P, \omega)) \cos \theta d\omega \quad (4)$$

where $R(Q)$ is the reflectivity of point Q as seen from P in direction ω . When no surface is seen at a distance less than d_{max} in direction ω , the obscurance takes the value of R_{ave} .

For coherency, the ambient light equation (3) also has to be modified, yielding the following value:

$$I_A = \frac{1}{(1 - R_{ave})} \cdot \frac{\sum_{i=1}^n A_i E_i}{A_{total}} \quad (5)$$

The improved obscurance model can be computed in several ways. The usual option is to compute the obscurance equation (4) using Monte Carlo (or quasi Monte Carlo) ray casting technique, which casts several rays from a patch distributed along $\cos \theta$. The obscurance for the patch i will then be the average of the values gathered by the rays cast from this patch:

$$W(i) = \frac{1}{N_i} \sum_{j=1}^{N_i} \rho_j R_{int} \quad (6)$$

where N_i is the number of rays cast from patch i , R_{int} is the reflectance at the intersected patch (if no patch is intersected, then we take R_{ave}) and ρ_j is the value of the function $\rho()$ for ray j . Several Monte Carlo and quasi-Monte Carlo sampling techniques have been tested for obscurance computation in [Mendez04a], and the Halton quasi-Monte Carlo sequence gave the best performance.

But this quasi-Monte Carlo approach is not our only option. Sbert [Sbert97] demonstrated that casting cosine distributed rays from all patches in the scene is equivalent to casting global lines joining random points of the bounding sphere of the scene. Furthermore, it is also equivalent to casting bundles or parallel rays of random directions (see Figure2). Bundles of parallel rays can be efficiently cast on the graphics hardware using the depth peeling algorithm as shown in the next section.

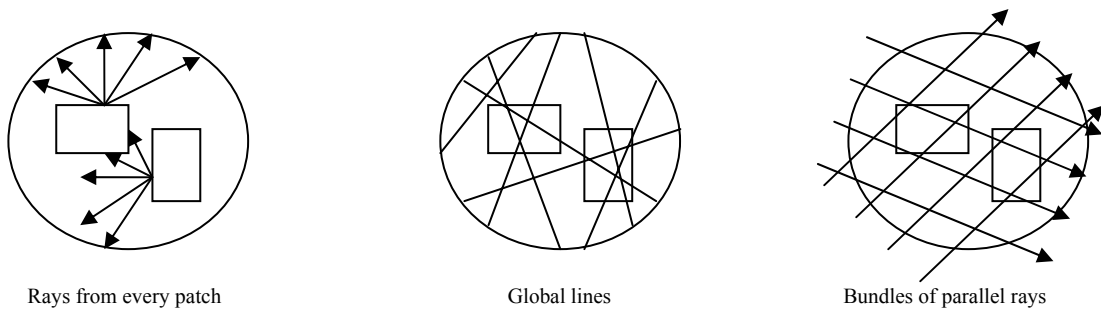


Figure 2: Different ray-tracing techniques for computing obscurances.

In Fig.3a we show the Cornell box scene computed with the obscurances but without color bleeding, while in Fig.3b we have used our improved algorithm. Color bleeding is clearly visible, adding a lot of realism to the image, with no added cost. We can compare these images with the one obtained with a more complete radiosity algorithm (Fig.3c) and see that the improved obscurance method represents a step forward towards simulating a radiosity image.

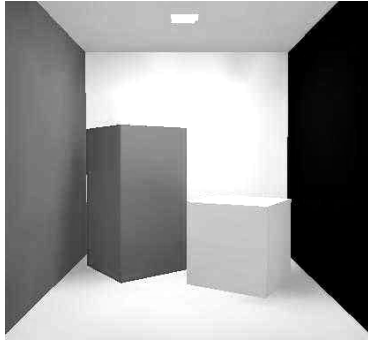


Fig 3a. Obscurances without color bleeding

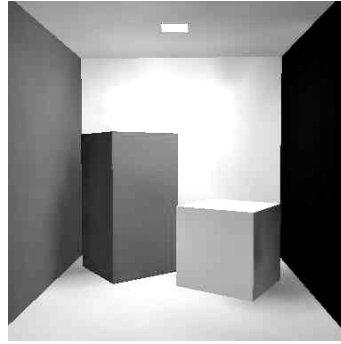


Fig.3b. Obscurances with color bleeding

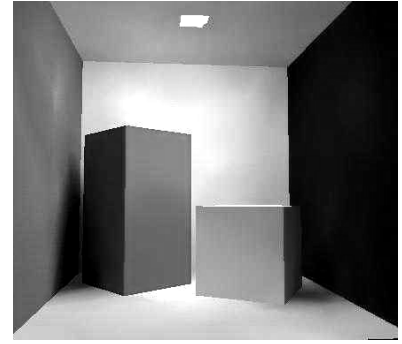


Fig.3c. A radiosity solution

4 GPU obscurances using depth peeling

The basic idea behind the depth-peeling technique is to extract visibility layers from the scene in order to do some computation between them. In [Everitt01], the technique is used to achieve order-independent transparency. Global illumination [Szirmay98, Hachisuka05] has been done also with depth-peeling.

We can see the pixel image resulting from depth-peeling as as being 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, and through depth-peeling we can discover all of the intersections in the form of image layers and not only the closest one obtained by the z-buffer algorithm.

Once we have chosen a random direction for the bundle, the computation of obscurances with depth peeling 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.

4.1 Depth Peeling

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 area, that corresponds to obscurance patches. 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 to 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 an image layer to store the patch identification, a flag indicates whether the patch is front-facing or back-facing to the camera and the camera to patch distance. Our pixel buffer is initialized with $(-1.0, -1.0, 1.0, 1.0)$, giving us 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. The cosine can be determined by using the z component of the dot product between the inverse transpose model-view matrix and the normalized normal vector of the patch.

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 ($R \leftarrow u, G \leftarrow v$), the third to store the cosine ($B \leftarrow \cos\alpha$), and the fourth component to store the distance between the camera and the patch ($A \leftarrow z$).

The vertex shader receives the vertex coordinates, the texture coordinates (in (u,v) , identifying the texel), and the normal, and it generates the cosine and the transformed vertex position:

```
void main( float4 position : POSITION,
          float2 texCoord : TEXCOORD0, //Patch ID
          float4 Norm      : NORMAL,   //Patch Normal

          out float4 oposition : POSITION,
          out float2 otexCoord : TEXCOORD0,
          out float cosine     : TEXCOORD1,

          uniform float4x4 modelView,
          uniform float4x4 modelViewInvTrans)
{
    oposition = mul(modelView,position);
    otexCoord = texCoord;

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

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 it if the depth of the previous layer (the fourth component of the sample) is closer to the camera than the actual fragment thus getting the peeling effect (Figure 4).

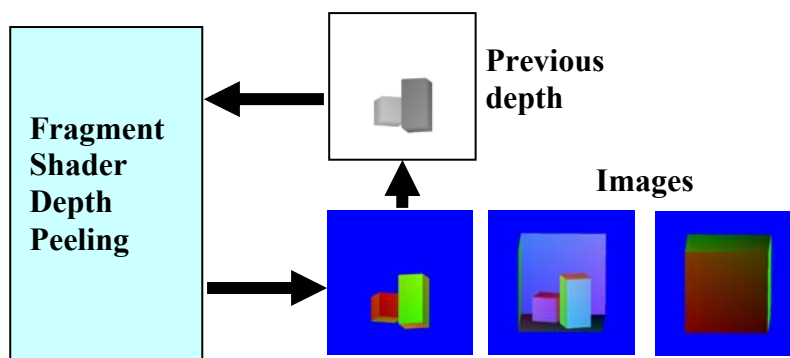


Figure 4: Schema of the depth peeling with GPU.

This rendering step is repeated until all pixels are discarded. The images of all the rendered layers define all ray-surface intersections (Figure 5).

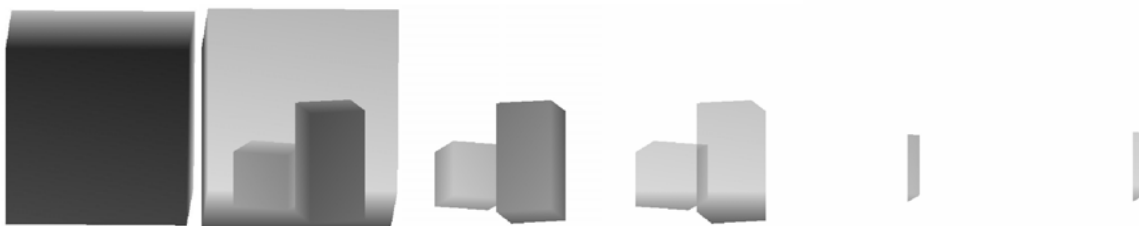


Figure 5: Six different image layers showing depth information for each pixel for the Cornell Box scene.

```

void main(float4 position : WPOS,
         float2 texCoord : TEXCOORD0, //Patch ID
         float cosine : TEXCOORD1, //Patch Orientation

         out float4 color : COLOR, //Patch ID + Orientation + Depth

         uniform sampler2D ztex, //previous depth image
         uniform float res, //resolution of projection window
         uniform float first) //is first layer?
{
    if( first == 0.0 ) // not first -> peel
    {
        float depth = tex2D(ztex,position.xy/res).a; //last depth
        if (position.z < (depth + 0.000001)) discard; //ignore previous
                                                    // layers
    }
    color.rg = texCoord;
    color.b = cosine;
    color.a = position.z; //new depth
}

```

4.2 Obscurances

For each pair of consecutive layers, the obscurance formula is 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 6).

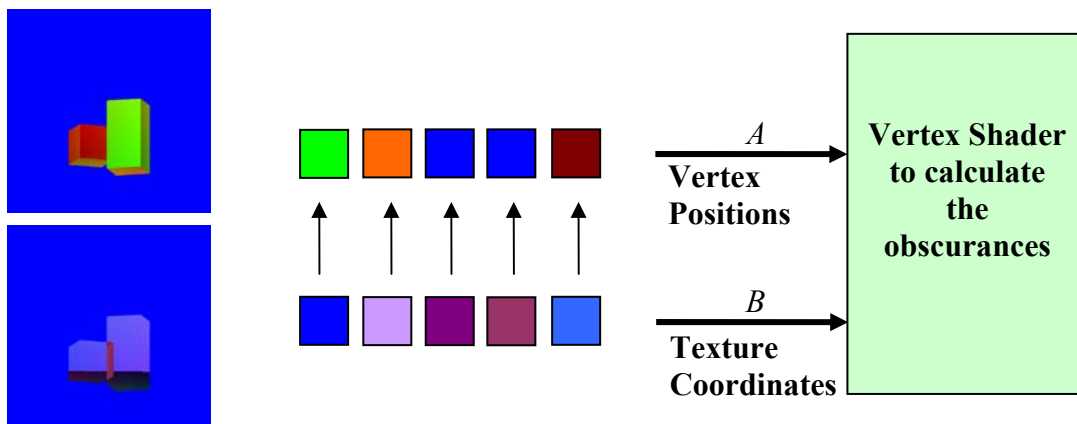


Figure 6. Two consecutive layers (left) generate two streams of points carrying patch ID's (middle) that are merged together and processed by the vertex shader (right).

The obscuration 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 6). 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 or transfers with the background, then the patch gets the ambient reflectivity. When the transfer process is done, we generate vertex coordinates to update the position in the obscuration 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 clip space. The desired position is encoded as the patch ID but is in a normalized form (as 2D texture coordinates are in the range [0..1]). The following formula computes which homogeneous clip coordinates we need to generate to obtain the desired normalized window coordinates given a camera and a viewport set as explained earlier:

$$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = \begin{pmatrix} 2x_w - 1 \\ 2y_w - 1 \\ 2z_w - 1 \\ 1 \end{pmatrix} \quad (7)$$

Note that the clipping process only keeps the fragment if $-w_c \leq z_c \leq w_c$. As we define a w_c of 1, the clipping process will only keep the fragment when $-1 \leq z_c \leq 1$. If $z_c=2.0$ then $z_w=1.5$ and the fragment is out of the view frustum and is discarded. If we set $z_c=0.0$ then $z_w=0.5$, thus 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 obscuration transfer process is:

```
void main( float4 A : POSITION,          //x,y = ID; z = Orientation; w = Depth;
           float4 B : TEXCOORD0,      //x,y = ID; z = Orientation; w = Depth;

           out float4 oposition : POSITION,
           out float2 pA        : TEXCOORD0, //Texture coordinates of A.
           out float2 pB        : TEXCOORD1, //Texture coordinates of B.
           out float distance   : TEXCOORD2, //Distance.

           uniform float direction) //Switch of direction
{
    //If patches see each other, i.e. both exist and one is front facing and the
    //other is back facing.
    //Direction tells if we are transferring in the camera -z direction or +z.
    //A.r contains the patch ID. If it contains -1.0 means that it does not belong
    //to the scene.
    //A.b contains the cosine.
    if(((direction == 0) && (A.r != -1.0) && (A.b < 0.0)
        && ((B.b > 0.0) || (B.r == -1.0)))
        || ((direction == 1) && (A.r != -1.0) && (A.b > 0.0)
            && ((B.b < 0.0) || (B.r == -1.0))))
    {
        pA = float2(A.r, A.g); //Set the texture coordinates for A.
        pB = float2(B.r, B.g); //Set the texture coordinates for B.
    }
}
```

```

//Create vertex to update desired position. z = 0.0 => kept by clipping
oposition = float4((pA * 2.0) - float2(1.0, 1.0), 0.0, 1.0);
//Calculate distance. If patch in stream B not in scene => distance = 1.0
distance = (texCoord.b != 1.0)? abs(B.a - A.a) : 1.0;
}
else //If there's no transfer move out from the view frustum.
{
// z = 2.0 to get the id ignored by clipping.
oposition = float4( 0.0, 0.0, 2.0, 1.0 );
p1 = p2 = float2(1.0, 1.0);
distance = 0.5;
}
}
}

```

The fragment shader just applies the obscurance formula $\rho = \sqrt{d/d_{\max}}$ if $d < d_{\max}$ and 1 otherwise.

```

void main( float2 pA : TEXCOORD0, //Texture coordinates of A.
float2 pB : TEXCOORD1, //Texture coordinates of B.
float distance : TEXCOORD2,

out float4 ocolor : COLOR,

uniform sampler2D reflectivity,
uniform float dmax,
uniform float3 ambient)
{
if(d>=dmax) ocolor.rgb = ambient; //If distance > dmax, add ambient
//else we apply the obscurances formula
else ocolor.rgb = tex2D(reflectivity,pB).rgb * sqrt(distance/dmax);
ocolor.a = 1.0;
}
}

```

Figures 7, 8 and 9 show the results of applying our algorithm to models of the De Espoña library. We show respectively the obscurances map, obscurances with direct illumination, and direct illumination with constant ambient term. Observe the quality of the illumination obtained with obscurances.

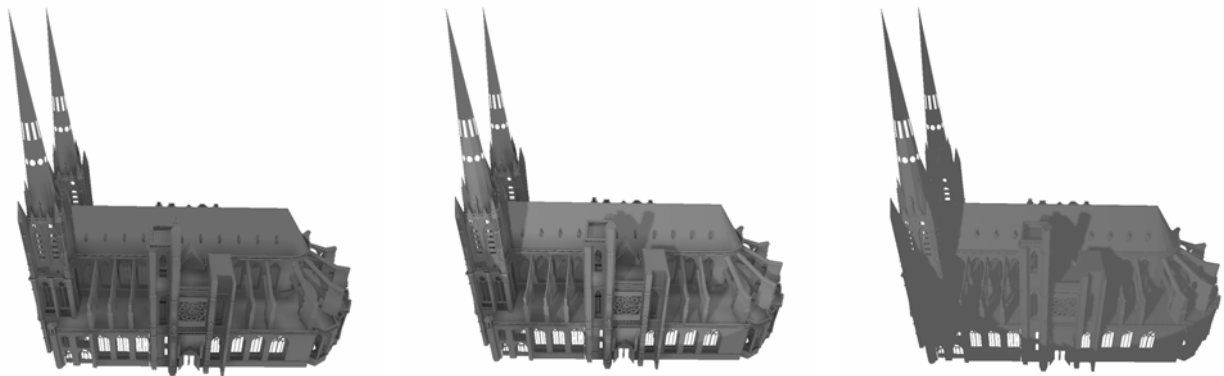


Figure 7. Cathedral model, 193180 polygons, obscurances computed in 38 seconds. Left: obscurances map, middle: obscurances with direct illumination, right: constant ambient term with direct illumination.



Figure 8. 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 9. Car model, 97473 polygons, obscurances computed in 32 seconds. Left: obscurances map, middle: obscurances with direct illumination, right: constant ambient term with direct illumination.

5 Real-time update for moving objects

Although the computation of the initial obscurance value is not done in real-time, we can update them in real-time efficiently for a moderate number of polygons.

We store the patches initially influenced by the dynamic objects, i.e., the patches that have been used in the computation of the obscurances for a dynamic object. When the object moves to a new position the obscurances from this list have to be recalculated. Obviously, you will also have to recalculate the obscurances of the patches of the moving object, and this causes an update of the list of influenced patches. Lastly, the obscurances of these new patches have to be recalculated, completing the whole update process.

The depth-peeling algorithm, as shown in section 4, is not the ideal algorithm to recompute the obscurances for a small part of the scene or moving objects, since by its nature it processes the whole scene at a time. More practical candidates are the computation by ray-tracing or hemi-cube projection.

For this article, we will consider the ray-tracing approach. The algorithm starts by creating a list of influenced patches. For each patch we compute the initial obscurance. If a ray from a patch hits a moving object we store the patch in the list of influenced patches. The patches of the moving objects are also stored in this list.

When an object moves to a new position, the obscurances are recomputed for each patch in the list of influenced patches, creating a new list of influenced patches.

In Figure 10 we show the result of recomputing in real-time the obscurances for a moving object, using the ray-tracing technique.

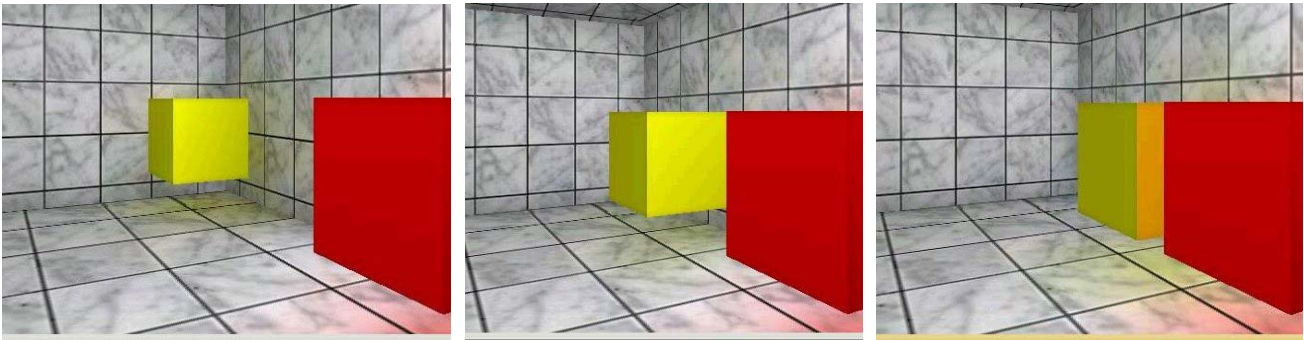


Figure 10. Three positions of a moving object showing the dynamic recomputation of obscurances.

6 Conclusions and Future work

We have shown in this paper how the obscurances method, a simple way to simulate diffuse global illumination, can be implemented in an efficient way in the GPU using the depth peeling technique. We have also seen that, as we query just a limited space around an object, obscurances can be updated in real-time for moving objects in the scene and the surfaces around, using a ray-casting technique.

As future work we plan to improve the efficiency of the depth peeling technique, by reading back in the CPU the textures to be analyzed using the symmetric PCI Express buffer. We will also study the efficiency of the hemicube projection method for the real-time update for moving objects.

The extension of obscurances to non-diffuse environments is also considered.

References

- [Bekaert98] Bekaert, P., Neumann, L., Neumann, A., Sbert, M., Willems, Y.: “Hierarchical Monte Carlo Radiosity”, Springer-Wien, New York (Proc. of Eurographics Rendering Workshop’98 - Vienna, Austria), pp. 259-268
- [Bunel05] Bunel, M., “*Dynamic Ambient Occlusion and Indirect Lighting*”, GPU Gems 2 Editors Matt Pharr and Randima Fernando, Addison-Wesley Professional, 2005.
- [Christensen03] Christensen, P. H., “Global illumination and all that”, *SIGGRAPH 2003 course notes #9* (RenderMan: Theory and Practice), pages 31-72, ACM, July 2003.
- [Everitt01] Everitt, C, “Interactive order-independent transparency”, Technical report, NVIDIA Corporation, 2001.
- [Goral84] Goral C., et al.: “Modeling the interaction of light between diffuse sources”, ACM SIGGRAPH’84 Conf. Proc
- [Hachisuka05] Hachisuka, T., “*High-Quality Global Illumination Rendering Using Rasterization*”, GPU Gems 2 Editors Matt Pharr and Randima Fernando, Addison-Wesley Professional, 2005.
- [Iones03] Iones, A., Krupkin, A., Sbert, M, Zhukov, S. *Fast realistic lighting for video games*, to appear in IEEE Computer Graphics&Applications, may-june 2003
- [Mendez03] Méndez-Feliu, À., Sbert, M., Catà, J., “*Real-time Obscurances with Color Bleeding*”, Proceedings of the Spring Conference on Computer Graphics 2003,

Budmerice, Slovak Republic, pp. 171-176, ACM Press, New York, NY, USA, 2003.

- [Mendez04a] Méndez-Feliu, À., Sbert, M., “*Comparing Hemisphere Sampling Techniques for Obscurance Computation*”, International Conference on Computer Graphics and Artificial Intelligence 31A 2004, Limoges, France, May 2004.
- [Nagy03] Nagy, Z., Klein, R., “*Depth-Peeling for Texture-Based Volume Rendering*” in proceedings of Pacific Graphics 2003, October 2003.
- [Pharr04] Pharr, M., Green, S., “*Ambient Occlusion*”, in GPU Gems by Addison-Wesley Professional, 2004
- [Sbert97] Sbert, M., “*The use of global random directions to compute radiosity. Global Monte Carlo Techniques*”, PhD dissertation, Technical University of Catalonia, 1997
- [Szirmay98] Szirmay-Kalos L. and Purgathofer, W., “*Global ray-Bundle tracing with Hardware Acceleration*”. In Proc. Of 9th Eurographics Rendering Workshop
- [Zhukov98] Zhukov, S., Iones, A., Kronin, G.: “*An Ambient Light Illumination Model*”, *Rendering Techniques'98*, Springer-Wien, NewYork (Proc. of Eurographics Rendering Workshop'98 - Vienna, Austria), pp. 45-55