A Practical and Fast Rendering Algorithm for Dynamic Scenes Using Adaptive Shadow Fields

Naoki Tamura[1], Henry Johan[2], Bing-Yu Chen[3] and Tomoyuki Nishita[1]

[1]The University of Tokyo [2]Nanyang Technological University [3]National Taiwan University

Abstract

Recently, a precomputed shadow fields method was proposed to achieve fast rendering of dynamic scenes under environment illumination and local light sources. This method can render shadows fast by precomputing the occlusion information at many sample points arranged on concentric shells around each object and combining multiple precomputed occlusion information rapidly in the rendering step. However, this method uses the same number of sample points on all shells, and cannot achieve real-time rendering due to the rendering computation rely on CPU rather than graphics hardware. In this paper, we propose an algorithm for decreasing the data size of shadow fields by reducing the amount of sample points without degrading the image quality. We reduce the number of sample points adaptively by considering the differences of the occlusion information between adjacent sample points. Additionally, we also achieve fast rendering under low-frequency illuminations by implementing shadow fields on graphics hardware.

1 Introduction

Creating photo-realistic images is one of the most important research topics in computer graphics and lighting plays an important role in it. Traditionally, people simulated the lighting environment by placing local light sources, such as point light sources, and area light sources. Recently, there are many methods using a dome-like lighting environment (environment illumination) to create photo-realistic images. However, since most of them use the ray-tracing method for rendering, they need a lot of computation time.

Sloan et al. presented the precomputed radiance transfer (PRT) method [23] to render a scene in real-time under environment illumination. Then, many methods were proposed to enhance both of the rendering quality and the computation efficiency: PRT methods using wavelet transform [16, 17], a method to compress the precomputed data using the principal component analysis (PCA) [22], etc. However, these methods have a problem: objects in the scene cannot be translated nor rotated.

Zhou et al. extended the PRT method to render dynamic scenes by using an environment illumination and local light sources together and proposed the precomputed shadow fields (PSF) method [28]. In this method, they precompute the *shadow fields* which describe the occlusion information of an individual scene entity at some sampled points arranged on concentric shells placed in its surrounding space. When rendering dynamic scenes, for each object, the occlusion information stored in its shadow fields is interpolated to compute its occlusion information at an arbitrary location. Then by quickly combining the occlusion information of all objects, the final occlusion information at an arbitrary location is computed efficiently. As a result, the radiance at a location can be computed fast.

Their method has two limitations. First, they store the shadow fields using the same number of sample points at all the concentric shells. Second, they perform the rendering on CPU that limits the performance. In this paper, we propose an algorithm to solve the limitations of the original PSF method, which includes two methods. We observe that the occlusion information stored in the shadow fields varies slowly with their neighbors. Hence, we first propose a method to optimize the number of sample points by considering the difference between the occlusion information at the nearby sample points. To increase the rendering performance under low-frequency illuminations, we also propose a method to use shadow fields on graphics hardware (GPU) for fast rendering. We assume that a scene consists of triangular meshes. Generally, if we approximate the light source and the occlusion information for all-frequency effects, the precomputed data size will become large. Moreover, for capturing rapid changes in radiance, it is necessary to subdivide the mesh data much finely. For these two reasons, all-frequency approximation is not suitable to be used in practical applications, such as computer games and virtual reality. Hence, in this paper we focus on the fast rendering under low-frequency illuminations which is sufficient for practical applications.

The rest of this paper is organized as follows. Section 2 describes the related work. Since our method is based on the PSF method, the algorithm and limitations of the PSF method are introduced in Section 3. The details of our algorithm are explained in Section 4. Then, Section 5 describes the implementation on graphics hardware. The results are shown in Section 6 and Section 7 describes the conclusion and future work.

2 Related work

Our method uses the dome-shaped light source and local area light sources as the lighting environment. Since the light sources have areas, it is important to simulate the soft shadows. Moreover, to render a scene under the dome-shaped light sources in real-time is related with the precomputed radiance transfer (PRT) method. Hence, the related work of these two categories is described in this section.

2.1 Rendering soft shadows

Nishita et al. proposed methods [18, 20] to render soft shadows caused by linear or area light sources. Moreover, they also proposed a method [19] to calculate soft shadows due to the dome-like sky light which is similar to the environment illumination. However, to use their methods to generate soft shadows needs a lot of calculation, and thus it is difficult to render in real-time.

Recently, there are many methods have been proposed for quickly calculating soft shadows by using GPU, which can be divided into the shadow map [27] based methods and the shadow volume [4] based ones. Heckbert and Herf used the shadow map method to project multiple shadows to the object and then combine the projected shadows to calculate soft shadows [6]. Heidrich et al. proposed a method to use GPU to calculate soft shadows caused by linear light sources [7]. In their method, they first put several sample points on the linear light source, and use the shadow map method to project the shadows to the object from the sample points. Then, the soft shadows are calculated by summing the generated shadows. Soler and Sillion presented a method to calculate soft shadows by using the fast fourier transform (FFT) method [25]. Agrawala et al. proposed a method to calculate soft shadows in screen space [1], but their method did not focus on the real-time calculation.

Akenine-Moller and Assarsson extended the shadow volume method to render soft shadows by using GPU [2, 3]. However, the calculation of their methods depends on the geometric complexity of the scene. Hence, it is difficult to calculate the soft shadows of a complex scene efficiently. In addition, their methods did not deal with the environment illumination. A fast soft shadows algorithm for ray tracing was proposed by Laine et al. [13]. This method, however, does not compute soft shadows in real-time.

2.2 Precomputed radiance transfer

Dobashi et al. used basis functions for fast rendering under skylight [5]. Ramamoorthi and Hanrahan proposed a method to render a scene under environment illumination in real-time by using the spherical harmonics (SH) basis [21]. However, their method did not take the shadows into account. To extend their method, Sloan et al. proposed the PRT method [23] which can render the soft shadows, interreflections, and caustics in real-time. Then, to improve the PRT method, Kautz et al. presented a method for arbitrary bidirectional reflectance distribution function (BRDF) shading [10], and Lehtinen and Kautz proposed a method to efficiently render the glossy surfaces [14]. Moreover, Sloan et al. proposed a method to compress the precomputed data by using the principal component analysis (PCA) [22], and also a method to render with the PRT method and bidirectional texture function (BTF) together [24]. Furthermore, Ng et al. used wavelet transform for all-frequency relighting [16, 17]. However, in these methods, the rendering objects cannot be translated nor rotated in the precomputed scene.

James and Fatahalian applied the PRT method to capture several scenes, and then they can interpolate them to simulate the translation, rotation, and deformation of the objects in the scene [8]. However, the transformation of the objects is limited to the ones captured at the preprocessing step. Mei et al. used the spherical radiance transport maps (SRTM) to make the object being able to have free translation and rotation [15]. However, in their method, the radiances of the vertices are calculated by using CPU only, and thus the performance is not so good. Since the SRTM needs many texture images while rendering, it is difficult to shift the calculation to GPU. Hence, their method cannot render a complex scene fast. Kautz et al. used hemispherical rasterization for all vertices and all frames under environment illumination and made the object capable of free deformation [9]. However, for a complex scene, the calculation is too complex to render the scene in real-time even after applying several optimizations. The method for fast rendering of soft shadows in dynamic scenes which distinguished between self-shadow and shadows cast by other objects was proposed by Tamura et al. [26]. This method, however, cannot deal with local light sources. Efficient soft shadows rendering under ambient light was proposed by Kontkanen et al. [12]. However, this method cannot take into account the illumination from distant lighting and local light sources.

Zhou et al. proposed the PSF method which precomputed the shadow fields to store the occlusion information of some sample points arranged on concentric shells placed at the surrounding of the object. When rendering, by quickly

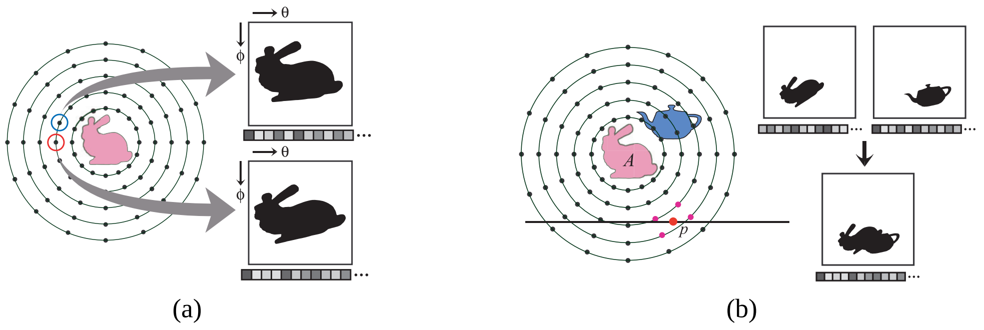


Figure 1. The concept of the precomputed shadow fields (PSF) method. (a) Precomputation of the shadow fields. (b) The calculation of the occlusion information due to other objects at point p.

combining the occlusion information stored in the shadow fields, they can render dynamic scenes which may contain several objects [28]. In their method, however, they use the same number of sample points at all shells. In this paper, we present a method to adaptively sample the shadow fields and thus our method reduces the data size of the shadow fields. Moreover, we present a GPU implementation for rendering using shadow fields under low-frequency illuminations. Hence our method can be used for practical applications, such as computer games and virtual reality.

3 Original precomputed shadow fields

In this section, we describe the overview and the limitations of the original PSF method [28].

3.1 Overview

In the PSF method, the shadow fields of each local light source and object, which will be translated and rotated, are precomputed as Fig. 1(a). To calculate the shadow fields, concentric shells are placed at the surroundings of the object. Then, a large number of sample points are generated on each shell, and the object occlusion field (OOF) and source radiance field (SRF) of the object are calculated at each sample point. The occlusion and radiance information at each sample point are calculated in longitude ϕ and latitude θ directions. The calculated information is approximated using spherical harmonics as [23] or using Haar wavelet transform as [16]. Different approximation methods will cause different qualities of shadows, rendering performance, and memory consumption. Furthermore, the self-occlusion (occlusion due to its own geometry) of each point is also precomputed.

To render using shadow fields, the radiance at each vertex is first calculated. Then, the scene is rendered by interpolating the radiance at each vertex. The occlusion information due to other objects during the radiance calculation is calculated by referring to the shadow fields as shown in Fig. 1(b). The occlusion information of object A at point p is calculated by interpolating the information at the sample points near p. The occlusion information due to more than one object is combined by using the triple product [17].

In the original PSF method, the locations of sample points are decided by projecting a cubemap to concentric shells. Cubemap based scheme is indeed efficient on sampling distribution, however, it is difficult to keep continuous interpolation near the cube edges when we optimize the sample points on each cubemap face independently. To simplify the interpolation, we employ polar coordinates model for the locations of sample points. In our method, the coefficient vectors of the orthonormal basis transformed from the occlusion information (one dimensional array under the occlusion information in Fig. 1) are called occlusion coefficient vectors (OCV). As for the source radiance information, we call them radiance coefficient vectors (RCV).

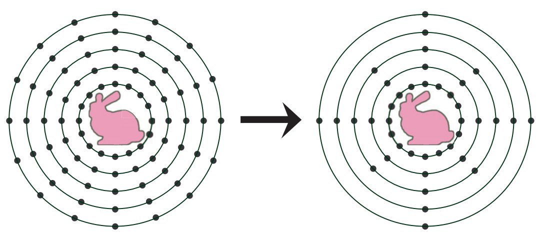
3.2 Limitations

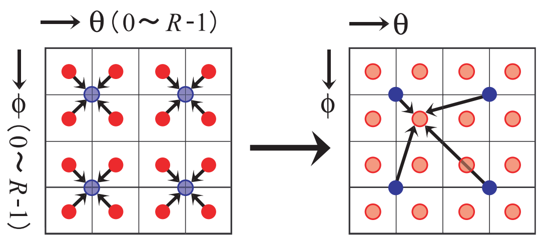
The PSF method proposed by Zhou et al. [28] has the following two limitations:

- Same number of sample points at all shells. If we set the sampling resolution R in ϕ and θ directions on C concentric shells and each sample point stores OCV with E elements where each element needs D byte, in the case of fix sampling resolutions, the data size of the shadow fields is $C \times R^2 \times E \times D$ bytes.
- Relatively low-rendering performance due to the computation using only CPU.

4 Adaptive shadow fields

In this section, the adaptive sampling method for the OOF is described. The SRF can also be adaptively sampled by using the same method.


(a)



(b)

Figure 2. Optimization of the number of sample points. (a) Previous method [28] uniformly put the sample points (left), but our method considers the variation of the occlusion information among the neighboring sample points to optimize the number of sample points (right). (b) Reducing the number of sample points by halving the sampling resolution in each direction (left) and checking if the new sample points (blue points) can approximate the initial sample points (red points) (right).

Based on our observation, the occlusion information stored in the shadow fields varies slowly. Therefore, we can reduce the data size of the shadow fields by removing some unnecessary sample points at each concentric shell respectively (Fig. 2(a)). We perform the optimization of sample points at each shell independently.

The details of the algorithm for optimizing the number of sample points at each concentric shell is as follows (see Fig. 2(b)).

1. Set the ***initial sample points*** on the shell with resolution R, that is, $R \times R$ sample points (red points).

2. Compute the occlusion information at all initial sample points and transform them to OCV O. We use spherical harmonics for low-frequency shadow fields and Haar wavelet transform for all-frequency shadow fields.

3. Arrange the ***new sample points*** on the shell with $R/2 \times R/2$ sample points (blue points).

4. Calculate the OCV $\bar{O}$ of the new sample points by linearly interpolating the occlusion information contained in its four nearest initial sample points.

5. Obtain the OCV $\tilde{O}$ of each initial sample point by linearly interpolating its four nearest new sample points (however, we use the nearest point for the corner, and the two nearest points for the boundary).

6. Calculate the difference between OCV $\tilde{O}$ and OCV O using Equation (1).

$$Error(s,t) = \frac{1}{4\pi}\int_{\Omega} |\sum_{g=0}^{G-1} \tilde{O}_g(s,t)\Psi_g(\omega) - \sum_{g=0}^{G-1} O_g(s,t)\Psi_g(\omega)| d\omega, \quad (1)$$

where Ψ is the basis function (spherical harmonics or wavelet), G is the number of the basis functions, $\frac{1}{4\pi}$ is the normalization term, and s and t are the indices of sample points in θ and ϕ directions, respectively.

7. For all the initial sample points, if the differences are lower than a specified threshold, the initial sample points are replaced with the new sample points, then halve the value of R and return to Step 3. The threshold will be explained in Section 6.1.

To perform the optimization recursively, we keep the number of sample points to be power of two. In our implementation, we set $R = 64$ in the initial state. By replacing the OCV with the RCV, we can use the above mentioned algorithm to adaptively sample the SRF.

5 GPU implementation

Our GPU implementation is for rendering using shadow fields whose OCV and RCV are approximated using four-th order spherical harmonics (16 bases), where each spherical harmonics coefficient is quantized to 8-bits. Fig. 3 shows the outline of the radiance computation using shadow fields. Since the radiance computation of each vertex v is independent of each other, it is possible to perform the computations in parallel and is hence suitable for GPU implementation. The underlined parts in the figure are performed on GPU.

In our method, we first prepare radiance texture T_B and vertex array texture T_L with sizes $N \times N$ ($N^2 >$ the number of vertices) for each object. We then use CPU to perform the visibility culling operation to calculate the visible vertex array L and store it in T_L. Next, we make the one-to-one correspondence between the k-th vertex v_k of L and the texel (x, y) of T_B, where $x = k \bmod N$ and $y = \lfloor k/N \rfloor$. After this operation, we perform the calculation of each individual vertex to be that of each texel, and most of the radiance computations are transferred to GPU as shown in Fig. 3. Finally, the radiance of each vertex of L is stored in T_B. In the rendering stage, we use vertex shader to reference the correspondence texel in T_B to obtain the vertex color.

To perform GPU based radiance computations, we have to keep U_v, S_j and O_j on GPU. We use the *Frame Buffer Object (FBO) extension* [11] to keep them. In our implementation, one FBO F is created and the e-th element of each U_v, S_j and O_j is stored in the $(e \bmod 4 + 1)$-th channel at the $(e/4 + 1)$-th COLOR_ATTACHMENT [11] of F. If we try to operate the computation of K-th order spherical harmonics on GPU, one FBO with $\lceil K^2/4 \rceil$ COLOR_ATTACHMENTs is needed. Current maximum number of the available COLOR_ATTACHMENT is four. Thus, our GPU based radiance computation is restricted to fourth order spherical harmonics due to hardware capability.

The underlined parts in Fig. 3 mainly consist of the following four computations.

1. Reconstruct the OCV (RCV) of each object (light source) at each vertex from the adaptive shadow fields.
2. Rotate the axes of the local coordinates of the OCV (RCV) to the axes of global coordinates.
3. Combine the OCVs by calculating the triple product.
4. Compute the radiance by calculating the double product of coefficient vectors.

The details of Step 1 and Steps 2, 3, 4 are described in Section 5.1 and Section 5.2, respectively. Moreover, the culling operation and sorting of occluders are explained in Section 5.3.

```
// $T_B$ : the exitant radiance texture //
// $O_v$ : the self-occlusion information at vertex $v$ //
// $\tilde{\rho}$ : the product of the BRDF and a cosine term //
rotate distant lighting $S_d$ to align with global coordinate frame
**For** each entity $I$ that is an object **do**
    $L$ = visible vertices of $I$ that are visible from camera
    compute distance from center of $I$ to each scene entity
    sort entities in order of increasing distance
    **For** each visible vertex $v$ in $L$ **do**
        $\underline{T_B(v) = 0}$
        $\underline{U_v = \text{TripleProdut}(O_v, \tilde{\rho})}$
        $\underline{\text{rotate } U_v \text{ to align with global coordinate frame}}$
        **For** each entity $J$ **do**
            **If** $J$ is a light source
                $\underline{\text{calculate RCV } S_j(v)}$
                $\underline{\text{rotate } S_j(v) \text{ to align with global coordinate frame}}$
                $\underline{T_B(v) += DoubleProduct(S_j(v), U_v)}$
            **Else**
                $\underline{\text{calculate OCV } O_j(v)}$
                $\underline{\text{rotate } O_j(v) \text{ to align with global coordinate frame}}$
                $\underline{U_v = TripleProduct(O_j(v), U_v)}$
            **End If**
        **End For**
        $\underline{T_B(v) += DoubleProduct(S_d, U_v)}$
    **End For**
**End For**
```

Figure 3. Outline of the rendering process. The underlined parts are performed in GPU.

5.1 Reconstruction of the OCV (RCV) on GPU

Fig. 4 shows the outline of the OCV reconstruction process. In our method, we first convert the visible vertex v of target object I to coordinates v_J at the local coordinates of the occluder J. Then, we calculate the nearest two concentric shells H_1, H_2 at v_J. For each H, we refer to the OOF to compute the OCV by interpolating the OCV at the nearest four sample points. Furthermore, the computed OCV at each shell is interpolated according to the distance from v_J to H_1, H_2. The process here is fully performed on a fragment shader, and hence can obtain high performance. The RCV reconstruction is also performed as the OCV.

To realize the computation in Fig. 4, the data of the adaptive shadow fields have to be stored in textures T_O. To construct T_O, we first create the texture T_O^c to store the OCVs of the c-th $(c < C)$ concentric shell, which has $R_c \times R_c$ sample points (that is, R_c is the sampling resolution at the c-th shell). To create T_O^c, we use four RGBA (four channels) textures $(size : R_c \times R_c + 2)$. For keeping continuous interpolation at the boundaries of ϕ direction, we allocate extra texels on T_O^c borders and the OCVs at the boundary are duplicated to the other boundary. On each texture, the e-th $(e : 0, ..., 15)$ element of the OCV of a sample point $(s, t : 0, ..., R_c - 1)$ is stored in the $(e \bmod 4 + 1)$-th channel of the $(s, t + 1)$ texel at the $(e/4 + 1)$-th texture of T_O^c. In the extra texels $(s, 0)$ and $(s, R_c + 1)$, we duplicate the OCV of sample points $(s, R_c - 1)$ and $(s, 0)$, respectively.

After creating T_O^c for all concentric shells, all T_O^c are packed to construct T_O as shown in Fig. 5. As described in Section 4, R_c is different for each concentric shell. In our packing method, we sort the T_O^c according to R_c to tile T_O^c as a rectangle. Although this R_c packing method may have some gaps, it does not pose a memory consumption problem since we only take low-frequency data into consideration in our GPU implementation and the memory consumption is relatively small. Furthermore, in this packing method, since T_O^c are usually preserved as a rectangle, we can use the bilinear interpolation functions on GPU to efficiently interpolate four points when performing TextureFetch(T_O, t_J) in Fig. 4. Since T_O^c in T_O is arranged according to the size of R_c, we use an additional address texture to store the position of T_O^c on T_O.

Since T_O is quantized to 8-bits, it is necessary to create four RGBA textures T_q $(size : 2 \times C)$ to store the minimum value and step of quantization for the restoration of the OCV on GPU. Hence, we store the minimum value of the e-th

```
// $I$ : target object, $J$ : occluder object //
Input:
    $t$ texture coordinates of screen pixel
Output:
    $O$ OCV
Constants:
    $M_t$ transformation matrix from local coordinates
        to global coordinates of $I$
    $M_r$ inverse transformation matrix from global coordinates
        to local coordinates of $J$
    $V_r$ radius of the bounding sphere $J$
    $V_{sr}$ smallest radius of the first concentric shell of $J$
    $V_i$ distance between the shells of $J$
    $C$ the number of shells of $J$
Texture:
    $T_L$ visible vertex position data
    $T_O$ shadow fields data (8 bit quantized), 4 textures
    $T_q$ quantization constant data (min value and step value), 4 textures
Note:
    TextureFetch( $T$, $u$ ) texture fetch from texture $T$ using texture coordinates $u$

$O$ = 0
local vertex position $v_l$ =TextureFetch( $T_L$ , $t$ )
global vertex positon $v_w = M_t \times v_l$
$v_J$ = transform $v_w$ to local coordinates of $J$ by using $M_r$
calculate the spherical coordinates $t_J$ of $v_J$
calculate nearest two concentric shells $H_1$ , $H_2$ by using $v_J, V_r, V_{sr}, V_i, C$
For each $H$ do
    calculate weight $w$ of $H$
    min, step = TextureFetch( $T_q$ , $t_J$ )
    coeff = TextureFetch( $T_O$ , $t_J$ )
    $O$ += $w$ * ( coeff * step + min )
End For
```

Figure 4. Outline of our OCV reconstruction fragment shader.

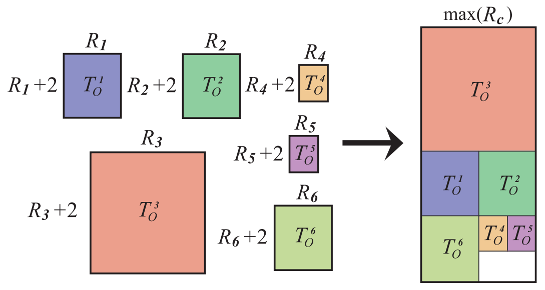


Figure 5. The concept of texture storage for the adaptive shadow fields.

element of the c-th concentric shell as the $(e \bmod 4+1)$-th element of the texel $(1, c)$ on the $(e/4+1)$-th texture of T_q. The step value is stored in the position $(2, c)$.

5.2 SH rotation, double product, and triple product

We use the ZXZXZ Rotation method [10] to perform the spherical harmonics rotation. Fig. 6 shows the portions of our nVIDIA Cg fragment shader code to compute the spherical harmonics rotation. The number of total instructions of the shader is 137. Since the values of alpha, beta, gamma in Fig. 6 depend only on the amount of rotation of the object, that means their values are the same for all the vertices, we hence compute their values on CPU and set them as the shader constants.

To compute the double product of the coefficient vectors on GPU, we employ the vector dot product command

```
#define  SQRT6_4       0.61237243569579447
#define SQRT10_4       0.79056941504209488
#define SQRT15_4       0.96824583655185426

half4x4 shRotXp16( const half4x4 src )
{
    half4x4 dest; // rotated SH coefficients //

    dest[ 0 ].r =  src[ 0 ].r; dest[ 0 ].g = -src[ 0 ].b;
    dest[ 0 ].b =  src[ 0 ].g; dest[ 0 ].a =  src[ 0 ].a;

    // ...   calculate dest[ 1 ].r - dest[ 2 ].a   ... //

    dest[ 3 ].r = -SQRT10_4 * src[ 2 ].g - SQRT6_4 * src[ 2 ].a;
    dest[ 3 ].g = -0.25 * src[ 3 ].g - SQRT15_4 * src[ 3 ].a;
    dest[ 3 ].b =  SQRT6_4 * src[ 2 ].g - SQRT10_4 * src[ 2 ].a;
    dest[ 3 ].a = -SQRT15_4 * src[ 3 ].g + 0.25 * src[ 3 ].a;

    return dest;
}

half4x4 shRotXn16( const half4x4 src )
{
    // ...   abbreviated   ... //
}

half4x4 shRotZ16( const half4x4 src, const half2 sinCos )
{
    // ...   abbreviated   ... //
}

half4x4 SHRotate16( half4x4 src, half2 alpha, half2 beta, half2 gamma )
{
    half4x4 temp1, temp2;

    temp1 = shRotZ16( src, gamma );
    temp2 = shRotXn16( temp1 );
    temp1 = shRotZ16( temp2, beta );
    temp2 = shRotXp16( temp1 );
    temp1 = shRotZ16( temp2, alpha );

    return temp1;
}
```

Figure 6. nVIDIA Cg fragment shader code for the fourth order spherical harmonics rotation.

provided in the fragment shader. Since there are 16 coefficients, we compute the dot product on every 4 coefficients and sum up the results.

As mentioned in Ng et al.[17] and Zhou et al.[28], the number of the non-zero tripling coefficients of the fourth order spherical harmonics are relatively small (77). Therefore, we determine all the non-zero tripling coefficients and use them to compute the projection coefficients. Fig. 7 shows some portions of the fragment shader code for performing the triple product. The number of total instructions of the shader is 415.

5.3 Culling and sorting

As in the original PSF, we perform the visibility culling for the vertices of each object on CPU. We cull the vertices outside the view volume and the vertices which do not belong to the front face triangles. The vertices that passed the culling test are put in visible vertex array L. The coordinates of the vertices in L are then put in T_L and the correspondences between the vertices in L and the texels in radiance texture T_B are redetermined. Since the number of vertices in L (referred as $|L|$) differs for every objects and in every frames, for efficient computation, we only execute the pixel shader on a rectangular region with width N and height $|L|/N$. The array of the self-occlusion information

```
half4x4 tripleProductSH16( half4x4 coeff1, half4x4 coeff2 )
{
    half4x4 projCoeff; // resulting basis coefficients //

    // ...   calculate coefficients of the first - ninth basis   ... //

    // calculate coefficient of the tenth basis //
    projCoeff[ 2 ].g  =  0.282095 * ( coeff1[ 0 ].r * coeff2[ 2 ].g +
                                      coeff1[ 2 ].g * coeff2[ 0 ].r );
    projCoeff[ 2 ].g +=  0.226179 * ( coeff1[ 0 ].g * coeff2[ 2 ].r +
                                      coeff1[ 2 ].r * coeff2[ 0 ].g );
    projCoeff[ 2 ].g +=  0.226179 * ( coeff1[ 0 ].a * coeff2[ 1 ].r +
                                      coeff1[ 1 ].r * coeff2[ 0 ].a );
    projCoeff[ 2 ].g += -0.094032 * ( coeff1[ 1 ].r * coeff2[ 3 ].g +
                                      coeff1[ 3 ].g * coeff2[ 1 ].r );
    projCoeff[ 2 ].g +=  0.148677 * ( coeff1[ 1 ].g * coeff2[ 3 ].b +
                                      coeff1[ 3 ].b * coeff2[ 1 ].g );
    projCoeff[ 2 ].g += -0.210261 * ( coeff1[ 1 ].b * coeff2[ 2 ].g +
                                      coeff1[ 2 ].g * coeff2[ 1 ].b );
    projCoeff[ 2 ].g +=  0.148677 * ( coeff1[ 1 ].a * coeff2[ 2 ].b +
                                      coeff1[ 2 ].b * coeff2[ 1 ].a );
    projCoeff[ 2 ].g += -0.094032 * ( coeff1[ 2 ].r * coeff2[ 2 ].a +
                                      coeff1[ 2 ].a * coeff2[ 2 ].r );

     // ...   calculate coefficients of the eleventh - sixteenth basis   ... //

    return projCoeff;
}
```

Figure 7. nVIDIA Cg fragment shader code for the fourth order spherical harmonics triple product.

O_v of the vertices in L are also stored in textures in every frame.

Since for each object, we compute the radiances of its vertices simultaneously on GPU, to efficiently perform the radiance computations, for each object, we sort its occluders and use the results when computing the radiance of all its vertices. However, if the object is very large, the sorting results may not applicable at some of the vertices. To avoid this problem, for large objects, we divide the mesh into several sub-meshes and perform the radiance computation in the unit of sub-mesh.

6 Results

In this section, we show the rendering results using adaptive shadow fields. In our experiments, we use a desktop PC with a Intel Pentium D 3.0GHz CPU and a GeForce 7800GTX GPU. The occlusion and the radiance information are computed as maps with resolution 64×64. For low-frequency shadow fields, we use 32 concentric shells and 64×64 sample points as the initial sampling resolution. The information in the shadow fields is approximated using the fourth order spherical harmonics with 16 coefficients, where each coefficient is quantized to 8 bits. For all-frequency shadow fields, we use 32 concentric shells and 64×64 sample points as the initial sampling resolution. The information is approximated using wavelets and 5% of the largest coefficients are kept, where each coefficient is also quantized to 8 bits. The center of the concentric shells is placed at the center of the object and the radius of the c-th $(c = 0, ..., 31)$ shell is $0.2V_r(1 + c)$, where V_r is the radius of the bounding sphere of the object.

Table 1. Optimal number of sample points.

	Sampling resolution R_c at the shells															
Teapot (L)	8	64	64	64	64	64	64	64	64	32	32	32	16	16	16	16
	16	16	8	8	8	8	8	8	8	8	8	8	8	8	8	8
Teapot (A)	8	64	64	64	64	64	64	64	64	64	64	64	64	64	32	32
	32	16	16	16	8	8	8	8	8	8	8	8	8	8	8	8
Statue (L)	64	64	64	64	64	64	64	64	32	16	16	16	16	8	8	8
	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
Statue (A)	64	64	64	64	64	64	64	64	64	64	32	32	16	16	8	8
	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
Plane (L)	64	64	64	64	64	64	64	64	64	64	64	32	16	16	16	16
	16	16	16	8	8	8	8	8	8	8	8	8	8	8	8	8
Plane (A)	64	64	64	64	64	64	64	64	64	64	64	64	64	32	32	32
	16	16	16	8	8	8	8	8	8	8	8	8	8	8	8	8

Table 2. The data sizes of the shadow fields.

	Fix sampling (MB)	Adaptive sampling (MB)
Teapot (L)	2.0	0.59
Teapot (A)	21.9	13.0
Statue (L)	2.0	0.55
Statue (A)	21.3	11.7
Plane (L)	2.0	0.75
Plane (A)	19.5	11.9

6.1 Determining the threshold values

In order to determine the threshold value to be used when we optimize the number of sample points, we performed several experiments as follows. We made a scene consisting of three different types of objects, that is, an almost isotropic object (a teapot), a long object (a statue), and a plane (thin rectangular solid). Then, we rendered it by varying the threshold using non-adaptive and adaptive, low- and all-frequency shadow fields. The results are shown in Fig. 8. We can notice the difference between the images generated by non-adaptive shadow fields and adaptive shadow fields when setting the threshold to 0.020, but the difference is not notable when seting the threshold to 0.010. Therefore, we set the threshold to 0.010 for all the rest of the examples presented in this section.

6.2 Optimal sampling resolutions

Table 1 shows the sampling resolutions at the concentric shells of the shadow fields of the three objects described in Section 6.1. The shells are sorted in order of increasing radius. For each object, the upper and the lower row are the results for low-frequency (L) and all-frequency (A) shadow fields, respectively. Table 2 shows the data sizes of the shadow fields. By using adaptive sampling, generally, we achieve about 60% - 70% and 40% - 45% reduction on the data sizes of the low-frequency and all-frequency shadow

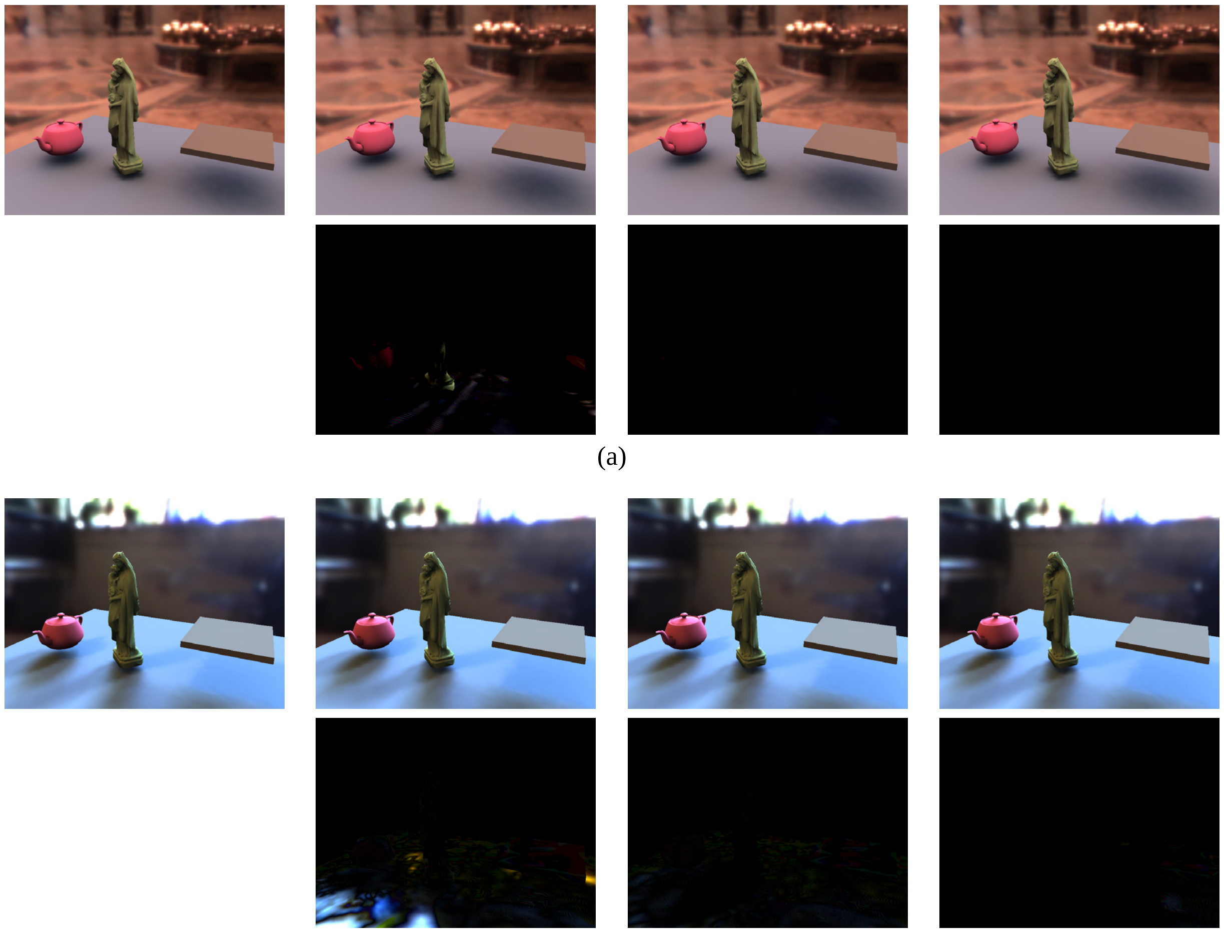


Figure 8. Determining the threshold values for (a) low-frequency shadow fields and (b) all-frequency shadow fields. The images from left to right are the result of using non-adaptive shadow fields (for comparison) and the results of using adaptive shadow fields and setting the threshold to 0.020, 0.010, 0.005, respectively. The images at the bottom show the differences (scaled 15 times) between the result of non-adaptive shadow fields and the results of adaptive shadow fields.

fields, respectively. The reason why the first shell of the teapot has the smallest resolution is that all sample points of the shell are located inside the teapot, and thus the differences of occlusion information are everywhere zero. Note that in our implementation for low-frequency shadow fields, we use 32 shells, each consisting of 4,096 samples, and each sample has an OCV at size 16 bytes. Therefore, the data size of our non-adaptive low-frequency shadow fields is $32 \times 4,096 \times 16 = 2$ MB.

6.3 Rendering results

Fig. 9 - 11 show the rendering results using our algorithm. We rendered three scenes while changing the environment illumination and moving the objects. The objects are moved by user in Fig. 9, and by rigid body simulation in Fig. 10 and 11. For the rigid body simulation, we use the PhysX Engine[1]. The statues scene (Fig. 9) has 6 objects and 32,875 vertices, the bowling scene (Fig. 10) has 15 objects and 41,340 vertices, and the falling objects scene (Fig. 11) has 20 objects and 88,739 vertices. For the statues and the falling objects scenes, some of the objects have glossy BRDF. For the bowling scene, during the animation, we changed the BRDF of the ball to glossy BRDF.

Table 3 shows the total sizes of the shadow fields and the comparison of the rendering performances using CPU and GPU of the three scenes. It is obvious that using the proposed GPU implementation, we were able to significantly speed up the rendering process.

[1]Ageia (PhysX Engine): http://www.ageia.com/

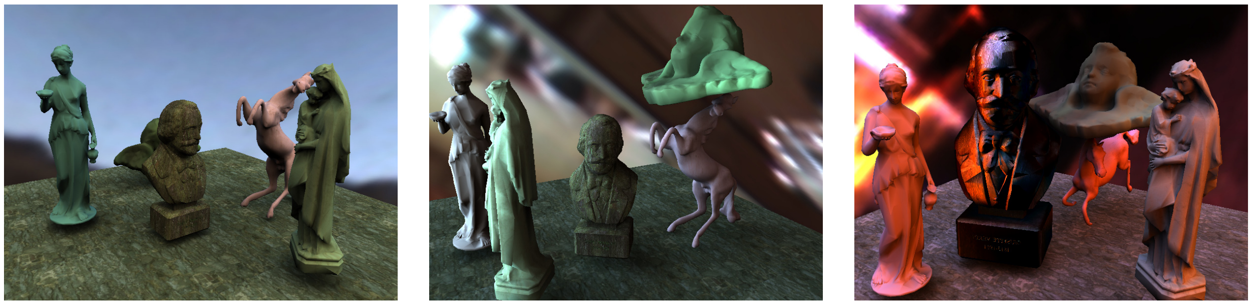


Figure 9. The statues scene (6 objects, 32,875 vertices).

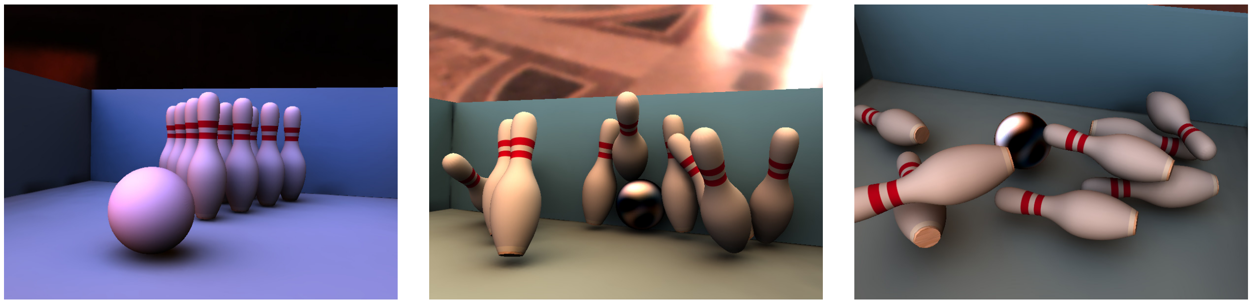


Figure 10. The bowling scene (15 objects, 41,340 vertices).

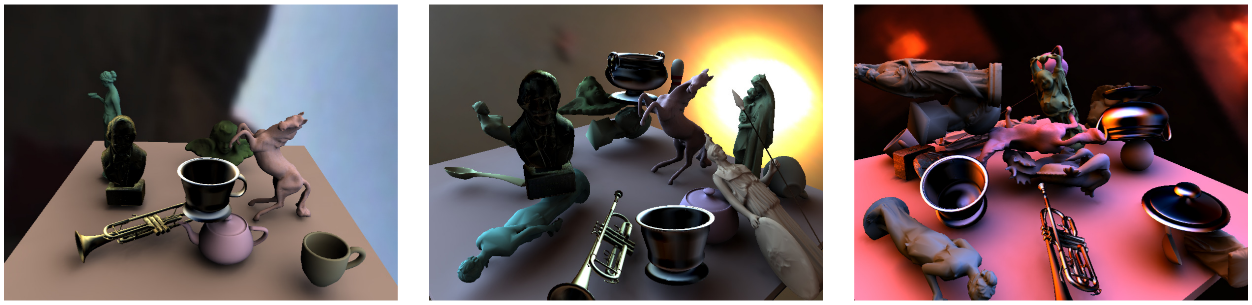


Figure 11. The falling objects scene (20 objects, 88,739 vertices).

Table 3. The total data sizes of the shadow fields and the rendering performances.

	Sizes of SF (MB)	CPU (fps)	GPU (fps)
Statues	4.6	4 - 18	70 - 100
Bowling	4.2	1 - 3	30 - 40
Falling objects	12.9	0.5 - 1	12 - 16

7 Conclusion and future work

In this paper, we have presented an algorithm for adaptively sampling the shadow fields and for fast rendering of dynamic scenes under environment illumination and local light sources. Concretely, we solved the limitations of the original PSF method. We decrease the data size of shadow fields by reducing the amount of the precomputed data, since the difference between the nearby precomputed data is small. Thus, we can adaptively optimize the number of sample points of the shadow fields. Furthermore, we realize the fast radiance computation under low-frequency illuminations by implementing the PSF method on GPU.

The next thing to do is to explore the possibility of compressing the all-frequency shadow fields. We also believe that our algorithm can be extended to adaptively control the number of the concentric shells.

References

[1] M. Agrawala, R. Ramamoorthi, A. Heirich, and L. Moll. Efficient image-based methods for rendering soft shadows. In *Proc. SIGGRAPH 2000*, pages 375–384, 2000.

[2] T. Akenine-Moller and U. Assarsson. Shading and shadows: Approximate soft shadows on arbitrary surfaces using penumbra wedges. In *Proc. 13th Eurographics Workshop on Rendering*, pages 297–306, 2002.

[3] U. Assarsson and T. Akenine-Moller. A geometry-based soft shadow volume algorithm using graphics hardware. *ACM Transactions on Graphics*, 22(3):511–520, 2003. (Proc. SIGGRAPH 2003).

[4] F. C. Crow. Shadow algorithms for computer graphics. In *Proc. SIGGRAPH 77*, pages 242–248, 1977.

[5] Y. Dobashi, K. Kaneda, H. Yamashita, and T. Nishita. A quick rendering method for outdoor scenes using sky light luminance functions expressed with basis functions. *The Journal of the Institute of Image Electronics Engineers of Japan*, 24(3):196–205, 1995.

[6] P. S. Heckbert and M. Herf. Simulating soft shadows with graphics hardware. 1997. Technical report CMU-CS-97-104, Carnegie Mellon University, January 1997.

[7] W. Heidrich, S. Brabec, and H. P. Seidel. Soft shadow maps for linear lights. In *Proc. 11th Eurographics Workshop on Rendering*, pages 269–280, 2000.

[8] D. L. James and K. Fatahalian. Precomputing interactive dynamic deformable scenes. *ACM Transactions on Graphics*, 22(3):879–887, 2003. (Proc. SIGGRAPH 2003).

[9] J. Kautz, J. Lehtinen, and T. Aila. Hemispherical rasterization for self-shadowing of dynamic objects. In *Proc. Eurographics Symposium on Rendering 2004*, pages 179–184, 2004.

[10] J. Kautz, P. P. Sloan, and J. Snyder. Shading and shadows: Fast, arbitrary BRDF shading for low-frequency lighting using spherical harmonics. In *Proc. 13th Eurographics Workshop on Rendering*, pages 291–296, 2002.

[11] M. J. Kilgard, editor. *nVIDIA OpenGL Extension Specifications*. nVIDIA Corporation, 2004.

[12] J. Kontkanen and S. Laine. Ambient occlusion fields. In *Proc. Symposium on Interactive 3D Graphics and Games 2005*, pages 41–48, 2005.

[13] S. Laine, T. Aila, U. Assarsson, J. Lehtinen, and T. Akenine-Moller. Soft shadow volumes for ray tracing. *ACM Transactions on Graphics*, 24(3):1156–1165, 2005. (Proc. SIGGRAPH 2005).

[14] J. Lehtinen and J. Kautz. Matrix radiance transfer. In *Proc. Symposium on Interactive 3D Graphics 2003*, pages 59–64, 2003.

[15] C. Mei, J. Shi, and F. Wu. Rendering with spherical radiance transport maps. *Computer Graphics Forum*, 23(3):281–290, 2004. (Proc. Eurographics 2004).

[16] R. Ng, R. Ramamoorthi, and P. Hanrahan. All-frequency shadows using non-linear wavelet lighting approximation. *ACM Transactions on Graphics*, 22(3):376–381, 2003. (Proc. SIGGRAPH 2003).

[17] R. Ng, R. Ramamoorthi, and P. Hanrahan. Triple product wavelet integrals for all-frequency relighting. *ACM Transactions on Graphics*, 23(3):477–487, 2004. (Proc. SIGGRAPH 2004).

[18] T. Nishita and E. Nakamae. Continuous tone representation of three-dimensional objects taking account of shadows and interreflection. In *Proc. SIGGRAPH 85*, pages 23–30, 1985.

[19] T. Nishita and E. Nakamae. Continuous tone representation of three- dimensional objects illuminated by sky light. In *Proc. SIGGRAPH 86*, pages 125–132, 1986.

[20] T. Nishita, I. Okamura, and E. Nakamae. Shading models for point and linear sources. *ACM Transactions on Graphics*, 4(2):124–146, 1985.

[21] R. Ramamoorthi and P. Hanrahan. An efficient representation for irradiance environment maps. In *Proc. SIGGRAPH 2001*, pages 497–500, 2001.

[22] P. P. Sloan, J. Hall, J. Hart, and J. Snyder. Clustered principal components for precomputed radiance transfer. *ACM Transactions on Graphics*, 22(3):382–391, 2003. (Proc. SIGGRAPH 2003).

[23] P. P. Sloan, J. Kautz, and J. Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In *Proc. SIGGRAPH 2002*, pages 527–536, 2002.

[24] P. P. Sloan, X. Liu, H. Y. Shum, and J. Snyder. Bi-scale radiance transfer. *ACM Transactions on Graphics*, 22(3):370–375, 2003. (Proc. SIGGRAPH 2003).

[25] C. Soler and F. X. Sillion. Fast calculation of soft shadow textures using convolution. In *Proc. SIGGRAPH 98*, pages 321–332, 1998.

[26] N. Tamura, H. Johan, and T. Nishita. Deferred shadowing for real-time rendering of dynamic scenes under environment illumination. *Computer Animation and Virtual Warld*, 16:475–486, 2005.

[27] L. Williams. Casting curved shadows on curved surfaces. In *Proc. SIGGRAPH 78*, pages 270–274, 1978.

[28] K. Zhou, Y. Hu, S. Lin, B. Guo, and H. Y. Shum. Precomputed shadow fields for dynamic scenes. *ACM Transactions on Graphics*, 24(3):1196–1201, 2005. (Proc. SIGGRAPH 2005).