Effective Use of Procedural Shaders in Animated Scenes

Polina Kondratieva, Vlastimil Havran, and Hans-Peter Seidel

MPI Informatik,
Stuhlsatzenhausweg 85,
66123 Saarbrücken, Germany,
{polina,havran,hpseidel}@mpi-sb.mpg.de

Abstract. Complex procedural shaders are commonly used to enrich the appearance of high-quality computer animations. In traditional rendering architectures the shading computation is performed independently for each animation frame which leads to significant costs. In this paper we propose an approach which eliminates redundant computation between subsequent frames by exploiting temporal coherence in shading. The shading computation is decomposed into view-dependent and view-independent parts and the results of the latter one are shared by a number of subsequent frames. This leads to a significant improvement of the computation performance. Also, the visual quality of resulting animations is much better due to the reduction of temporal aliasing in shading patterns.

1 Introduction

Creation of photo-realistic images with low computational requirements is one of the main goals of computer graphics. Procedural shaders can be used as an effective mean for rendering high-quality realistic images due to some distinct advantages, such as simplicity of procedural shading for arbitrarily complex surfaces and the possibility to change the shaded surface with time, viewing angle or distance [7].

The approach presented in this paper extends the research in Havran et al. [3]. We show that a significant part of shading computation can be reused in subsequent frames. There are two different techniques to prepare the shader data for reusing. While the first approach is based on the 3D-texture notion, the second one is related to analytical splitting of the procedural shader into the view-dependent and view-independent parts. The algorithm of reusing the view-independent data for both techniques is similar.

A key aspect of most procedural shading is the use of a shading language which allows a high-level description of the color and shading of each surface. Shaders written in the RenderMan Shading Language can be used by any compliant renderer, no matter what rendering method it uses [8]. For this reason all examples of shaders in this paper are similar to the *RenderMan* shaders.

The paper is organized as follows. Section 2 discusses the properties of three rendering architectures. The algorithm of reusing the view-independent data

is presented in Section 3. Section 4 describes preprocessing techniques for the preparation of shading data for reusing. Examples of such a preparation are also presented in this section. The achieved results are shown in Section 5. Finally, Section 6 concludes the paper and proposes some directions for future work.

2 Related Work

Here we discuss the advantage of Efficient Spatio-Temporal Architecture for Rendering Animation (ESTARA) [3] compared to the well-known rendering architectures using procedural shaders, such as REYES [2] and Maya renderer [5].

Rendered images have the property of similarity between the consecutive frames known as *temporal coherence*, which can be used to accelerate the rendering. Both Maya and REYES architectures compute images of animation sequence frame by frame. On the other hand, ESTARA exploits the property of temporal coherence by splitting the shading function into the view-independent and view-dependent parts, whereas the first one is computed only once for a given sample point and the second one is recomputed for each frame. In this way, ESTARA outperforms both REYES and Maya by considerable reducing the computational cost of shading computation as well as the temporal aliasing (flickering). ESTARA can be used for pixel based renderers including bidirectional path tracing, ray tracing, etc. Here we extend the ideas in [3] for ray tracing with procedural shaders.

3 Algorithm of Reusing the View-Independent Data for Procedural Shaders

3.1 Notions of View-Dependent and View-Independent Shader Parts

Before discussing the features of the algorithm of reusing the data, we give a brief definition of *view−dependent* (VD) and *view−independent* (VI) data with respect to procedural shaders. The symbolic notation used throughout the rest of the paper and adopted from RenderMan Shading Language is shown in Table 1.

The computation of the color for each sample can be split into two parts: VD and VI. The VI data do not change when the camera moves (see Fig. 1(a)). A simple example of the VI data is the $diffuse\ surface\ color$ which can be computed according to Lambertian law as follows:

$$diffuseCol \ = \ Kd \ \cdot Cs \ \cdot cos(\angle \mathbf{N}, \mathbf{L}) \ . \tag{1}$$

In contrast to the VI data, VD data change whenever the camera moves. More precisely, the VD data depend on the reciprocal location of the surface point (hit point) and camera (viewer) in 3D space. A simple example of the VI data is the *specular shading color* of the surface. According to the well-known Phong model, specular color can be computed as following:

$$specularCol \ = \ Ks \ \cdot \ specular \cdot cos^n(\angle \mathbf{R_m}, \mathbf{V}) \ . \tag{2}$$

In the following chapters we show examples of shader decompositions into the VD and VI data.

Table 1. Symbolic notation used in the document

Symbol	Description
Ka, Kd, Ks	coefficients for the ambient, diffuse and specular color
$rough$	roughness coefficient
Kt, Kr	refraction and reflection coefficients
$\mathbf{T}$, $\mathbf{R}$, $\mathbf{R_m}$	transmitted, reflected and mirror-reflected ray directions
Cs, Os	surface self color and opacity
Ci, Oi	incident ray color and opacity
$specular$	specular color of the surface
$\mathbf{N}$, $\mathbf{N_f}$	geometric and face-forwarded normals
$\mathbf{L}$, $\mathbf{V}$	incoming light and opposite view directions
$\mathbf{P}$	hit point position on the surface of an object
du, dv	changes in surface parameters
dPdu, **dPdv**	derivatives of surface position along u, v directions.

3.2 Algorithm of Reusing the Shading Data

The problem of wasting time for unnecessary recomputations of unchanged data is addressed by the algorithm of reusing the shading data described in detail here. The main idea of the algorithm is to save the VI data into the cache and then reuse it for subsequent frames.

The aliasing artifacts caused by poor sampling of procedural textures as well as ordinary 2D-textures can be reduced by increasing the number of samples, thus increasing rendering time. By reusing the VI data for shading in ESTARA we can decrease the computational cost for one sample. In addition, since the algorithm of reusing the VI data spreads the shading data into the time domain (Fig. 1(b)), similarly to [6], the temporal aliasing known as flickering and scintillation is highly reduced.

Let us describe the algorithm of reusing in more detail. For a sequence of N frames (N camera positions) the VI component of the pixel color in a given frame can be reused for subsequent frames. In this way, for the first frame of a sequence the ray with ID number unique for each hit point is shot and the color of the hit point is computed. The VI data of the computed pixel color is saved into the cache with the corresponding ray ID as a search key.

Afterwards the hit point is reprojected onto the image plane of subsequent frames. Note, that due to the fact that the camera position can change within the sequence of frames, the positions of the correspondent reprojected pixels on the image plane for subsequent frames are different. After the reprojection the hit point is checked for occlusion. If the ray is not occluded, the VI data can be used for shading computation.

For a range of frames the VI data are possibly combined with recomputed VD data to get the pixel color. This reusing of the VI data for shading is performed for all the pixels in the sequence of frames. Since the samples are obtained in a low cost, the total time of computations decreases.

An example of hit point reprojection for subsequent camera positions is shown in Fig. 1(b). For more details, see original paper [3]. Since any pixel of an image in a sequence of frames can be accessed by the reprojection, all the image data for a sequence of frames are saved in the main memory.

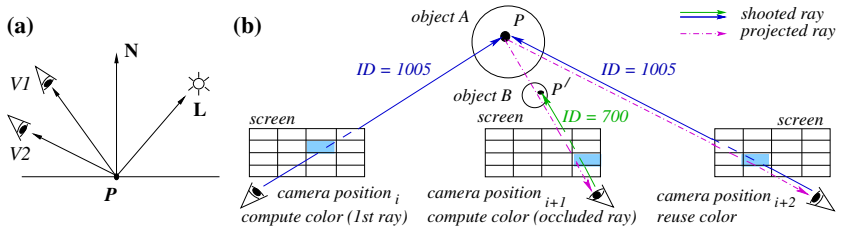

Fig. 1. Illustrations for the algorithm of reusing the data: (a) Example of the incoming/outgoing vectors for a given hit point **P**, (b) Hit point reprojection for subsequent camera positions followed by reusing or recomputation the shading data

3.3 Speedup Analysis

Let us now compare the timings for shading computation required for the traditional frame-by-frame approach and for the proposed algorithm of reusing the VI data. Suppose the pixel color computation for each camera position in this case takes T_s^1 time. Since, without reusing, the color of each sample for each camera position should be recomputed, the time T^1 required to compute the pixel color for $\boldsymbol{n}$ camera positions is:

$$T^1 = T_s^1 \cdot \boldsymbol{n} \ . \tag{3}$$

If the algorithm of reusing the data is involved, the situation changes. For the first camera position times T_{vd} and T_{vi} are required to compute the VD and VI data correspondingly, $T_{combine}$ to combine these parts, and T_{save} to save the VI data into the cache. For the remaining $\boldsymbol{n}-1$ camera positions times T_{vd}, $T_{combine}$ are needed as above, and $T_{extract}$ is required to extract (reuse) the VI data from the cache. Thus, the time needed to compute the pixel color for the first (T_s^2) and for the remaining $\boldsymbol{n}-1$ (T_s^3) camera positions can be calculated as follows:

$$T_s^2 = T_{vd} + T_{vi} + T_{combine} + T_{save}, \quad T_s^3 = T_{vd} + T_{extract} + T_{combine} \ .$$

The total time T^2 with reusing the VI data is then:

$$T^2 = T_s^2 + T_s^3 \cdot (\boldsymbol{n} - 1) \ . \tag{4}$$

The speedup of shading computation achieved by ESTARA with the algorithm of reusing the VI data can be evaluated from (3) and (4). It is clear that $T_s^1 < T_s^2$ and $T_s^1 > T_s^3$. Hence, $T_s^1 \cdot (\boldsymbol{n} - 1) > T_s^3 \cdot (\boldsymbol{n} - 1)$. If $\boldsymbol{n} > \lceil (T_s^2 - T_s^3)/(T_s^1 - T_s^3) \rceil$, then $T^1 > T^2$. Therefore, the maximum theoretical speedup achieved by applying the algorithm of reusing the data for $\boldsymbol{n}$ camera positions can be evaluated as follows:

$$\lim_{n \to \infty} speedup = \boldsymbol{n} \ \cdot \ T_s^1 / (T_s^2 \ + \ (\boldsymbol{n} - 1) \cdot T_s^3) = T_s^1 / T_s^3 \ . \tag{5}$$

Formula (5) shows that the computational cost of rendering can be reduced by the algorithm of reusing the shading data. Note that the main point is to use a fast data structure for saving the VI data, otherwise no speedup can be achieved. For this purpose some kind of the fixed-size cache with LRU replacement policy [4] is used in ESTARA.

4 Preprocessing Techniques for Shading Data Reusing

We distinguish two different procedural shader classes: representable as 3D-textures and non-representable as 3D-textures.

4.1 Shaders representable as 3D-textures

The main feature of these shaders is that all the properties of the shader are defined by the complex color (Cs in formula (1)), which represents some pattern on the surface and it is independent of both **V** and **N**.

Analysis of the VD and VI data described in section 3.1 allows to define whether a given shader is representable as a 3D-texture. So, the shader can be represented as a 3D-texture if it has the following properties:

- its VD data contain only the computation of the glossy specular color,
- the computed complex diffuse color does not depend on **V** and can be used together with the function $diffuse()$ in the same way as Cs color,
- it does not create any surface displacements (**N** perturbations).

Good examples of such shaders are *CWoodTexture* [11], *CStoneTexture* [12], and *CCobbleStoneTexture* [10]. The pseudo-code of the function which computes the complex diffuse color for the *CCobblestoneTexture* is shown in Listing 1.

```
CobbleStone_DiffuseColor(float Kd, jitter, sscale, ttscale, txtscale; color selfCol, varCol)
 {
    Scale P with txtscale, u with sscale, and v with ttscale
    Compute Voronoi noise voronoi(ss, tt,jitter, f1, spos1, tpos1, f2, spos2, tpos2)
    paintColor = selfCol,    Cgrout = selfCol · 0.5
    paintColor ·= (cellnoise([ spos1-67,tpos1+55 ])+1.5)· varCol · 0.5
    //Create cellular pattern with f2-f1.
    blendval = smoothstep( 0.03, 0.07, f2-f1)
    diffuseColor = Kd · (paintColor · blendval + Cgrout · (1— blendval))
 }
```

Listing 1. Example of procedural 3D-texture

If the shader is representable as a 3D-texture the complex diffuse color can be saved into the cache as a simple Cs color and then reused for the next frames.

4.2 Shaders non-representable as 3D-textures

The shaders of this class have implicit VD and VI data closely interacting with each other; the shading computation for them is decomposed into *layers*. There is a great variety of non-representable as 3D-textures shaders: some of them have only one layer, such as *velvet* [11], some others consist of many complicated layers involving Fresnel function, Ward reflection model, and/or some other functions for anisotropic reflection, such as *RCSkin* [11]. Despite of their complexity, even these shaders can be usually split into VD and VI data.

At this point, let us consider an example of complicated shader - *RCSkin*. It consists of four layers and the computation of the color for each layer is highly time consuming. The pseudo-code of the function which calculates the pixel color for this shader is presented in Listing 2.

```
RCSkin( float Kd, eta, thickness, angle, Xrough, Yrough, maxfreq, blemishfreq, blemishthresh,
      blemishopac, oily, brightness, poresfreq, poresthresh, poresdepth; color sheen, Cs, Os)
 {
      //--- layer 0 - pores ------------------
      Spread the pores over the surface, compute displaced normal (N_N)

      //--- layer 1 - skin main color --------
      color_skin = Cs, Oi = Os

      //--- layer 2 - blemishes subsurface ---
      PP = transform(object, P)· blemishfreq;   turb = 0;
      for (f = 1; f < maxfreq; f *= 2)   turb += abs(snoise(PP· f)) / f;
      blemishColor = spline(color_1,...,color_n,turb^blemishtresh)
      Compute Kr, Kt, R, T for the view ray
      illuminance cycle(P, N_f, π/2)
      {
        if(cos(∠(L + V), N_f) > 0)
          glossy = Kr · sheen · Cl · cos(∠L, N_f) · cos^4(∠(L+V),L)
        glossy += 2 · Kr · sheen · Cl · abs(cos(∠L, N_f))
        Compute Kr_2, Kt_2, R_2, T_2 for L, and single scattering approximations s_1, s_2, s_3
        glossy += blemishColor · Cl · cos(∠L, N_f) · Kt · Kt_2 · (s_1+s_2+s_3)
      }
      Mix color glossy with color_skin

      //- layer 3 - anisotropic Ward model ---
      Compute anisotropic directions anisDir(dPdu, N_f,angle), XaDir, YaDir
      illuminance cycle(P, N_f, π/2)
      {
        Compute Ward coefficient rho(XaDir, N_f, YaDir, L, V)
        if(Light source is specular) Canis = (Cl · cos(∠L, N_f) · rho)/(4 · Xrough · Yrough)
      }
      Diff = Kd · diffuse(N_f))
      color_skin += (Canis · 0.1 · oily + Diff) · brightness
      Save Diff, blemishColor, Os, N_N, P, XaDir, YaDir into the cache
 }
```

Listing 2. An example of non-representable as 3D-texture 4-layer shader

The *RCSkin* shader presented in Listing 2 computes a number of specific VD functions. For example, the Ward reflection model and the Fresnel function are quite computationally demanding. Fortunately, after the careful analysis the following components of the shader can be considered as the VI data:

- At ***layer 0*** the displaced normal $\mathbf{N_N}$ for pores,
- At ***layer 1*** the skin color *color_skin* and *Oi*,
- At ***layer 2*** the *blemishColor* computed by the *spline* function for 3D-vectors,
- At ***layer 3*** the anisotropic directions (**XaDir**, **YaDir**).

In the same way all the other non-representable as a 3D-texture shaders can be split into the VD and VI data. The main point is that the time required to compute the VI data should be greater than the time required to insert/extract the data from the cache.

5 Results

We have verified the efficiency of the described algorithm of reusing the shading data embedded by ESTARA on three scenes applying different shaders. A computer with processor Intel(R) Xeon(TM) CPU 1.706MHz and 1024MB of memory was used for rendering. All the shaders were taken from the [1], [9], or RenderMan sites [10],[11], [12] and adapted for our renderer, as described above.

At the first step, the speedup of the shading color computation for each shader was evaluated for a simple scene avoiding the visibility test. The timing results in seconds for all the shaders are shown in Table 2. In Table 2 column NOREUSE presents the shading time results for the traditional frame-by-frame approach. Column REUSE shows the timing results for the algorithm of reusing the shading data. Column SPEEDUP depicts the speedup (SPEEDUP = NOREUSE/REUSE).

At the next step, the speedup evaluation was accomplished for two more complex scenes: scene *Face*, containing *RCSkin* and *greenmarble* shaders, and scene *Interior*, containing all the shaders from Table 2 except *RCSkin*, performing the visibility test. Note, that speedup was achieved by the combined reusing of the VI data and visibility information. The timing results in seconds for both scenes are presented in Table 3.

Table 2. Timing and speedup results for shaders applied to the simple scene

	50 camera positions			100 camera positions		
SHADER	NOREUSE	REUSE	SPEEDUP	NOREUSE	REUSE	SPEEDUP
blocks	25.47	8.02	3.18	52.00	15.96	3.26
carpet	27.35	9.84	2.78	55.14	18.92	2.91
cmarble	77.89	18.24	4.27	154.76	34.93	4.43
colormarble	73.36	18.29	4.01	148.43	34.93	4.25
cobblestone	26.99	11.83	2.28	62.25	27.20	2.29
greenmarble	50.97	17.44	2.92	97.91	33.06	2.96
spatter	21.57	12.33	1.75	47.55	27.17	1.75
stone	15.68	13.13	1.19	32.91	25.99	1.27
velvet	12.27	12.44	0.99	24.45	23.84	1.03
wood	25.44	15.27	1.67	51.34	29.85	1.72
RCSkin	97.12	46.98	2.07	185.75	78.05	2.38

Table 3. Timimg results for scenes *Interior* and *Face*

	Time		
Scene	REUSE	NOREUSE	SPEEDUP
Face	2.24e+03	2.73e+03	1.22
Interior	1.25e+05	3.28e+05	2.62

The resulting images for the scene *Face* with different values of parameters for *RCSkin* shader are presented in Fig. 2(a) and 2(b). The resulting images for the scene *Interior* with procedural shaders are depicted in Fig. 2(c).

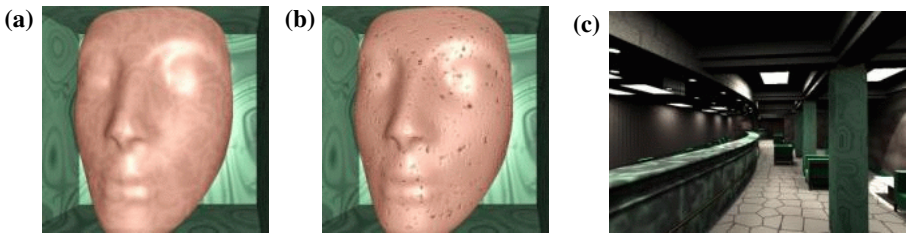

Fig. 2. Images rendered by ESTARA with reusing: (a), (b) scene *Face* with different parameter settings for *RCSkin* shader; (c) scene *Interior*

6 Conclusion and Future Work

In this paper we have described techniques, which significantly reduce the computational cost of procedural shading in animation rendering, while improving the quality of resulting images in the context of ESTARA rendering architecture [3]. The speedup is achieved by splitting the shader into two parts: the view-dependent (VD) and the view-independent (VI).

Applying the algorithm of reusing the shading data for ray tracing a moderately complex scene with procedural shaders, we received significant speedup up to a factor of 2.62. Since the VI data of the color are the same for the pixels corresponding to the reprojection of the shaded point in the object space to the image plane of subsequent frames, the temporal aliasing (flickering) is reduced.

The main disadvantage of the proposed algorithm is the fact that all shaders should be split into the VD and VI data manually. Intuitively, this time consuming and laborious process could be done by computer. We envision the automation of the splitting process as the next step in the development of the algorithm of reusing the shading data.

7 Acknowledgments

The authors would like to thank Karol Myszkowski for helpful discussions and suggestions during the preparation of the paper.

References

1. Apodaca, A.A., and Gritz, L. Advanced RenderMan. Morgan Kaufmann, 1999
2. Cook, R.L., Carpenter, L., Catmull, E.: The Reyes Image Rendering Architecture. ACM Computer Graphics SIGGRAPH'97 Proc. (1987) 95-102
3. Havran, V., Damez, C., Myszkowski, K., and Seidel, H.-P.: An Efficient Spatio-temporal Architecture for Animation Rendering. Eurographics Symposium on Rendering (2003)
4. Knuth, D.E.: The Art of Computer Programming, Vol.3 (Sorting and Searching). Addison-Wesley Series (1973).
5. Sung, K., Craighead, J., Wang, C., Bakshi, S., Pearce, A., and Woo, A.: Design and implementation of the Maya Renderer. Pacific Graphics'98 Proc. (1998) 150-159
6. Martin, W., Reinhard, E., Shirley, P., Parker, S. and Thompson, W.: Temporally coherent interactive ray tracing. Journal of Graphics Tools **2** (2002) 41-48
7. Olano, M.: A Programmable Pipeline for Graphics Hardware. PhD dissertation, University of North Carolina, Chapel Hill (1998)
8. Olano, M., Lastra, A.: A Shading Language on Graphics Hardware: The PixelFlow Shading System. ACM Computer Graphics SIGGRAPH'98 Proc (1998) 159-168
9. Upstill, S.: The RenderMan Companion. A programmer's Guide to realistic Computer Graphics. Addison-Wesley publishing company (1990)
10. http://www.cs.unc.edu/~stewart/comp238/shade.html
11. http://www.renderman.org/RMR/Shaders/
12. http://www-2.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15462/arch/sgi_65/prman/lib/shaders/stone.sl