Computers & Graphics 36 (2012) 193–200

Contents lists available at SciVerse ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Technical section

Memory efficient light baking

Henry Schäfer [a,*], Jochen Süßmuth [a], Cornelia Denk [b], Marc Stamminger [a]

[a] Computer Graphics Group, University of Erlangen-Nuremberg, Germany
[b] Realtime Technology AG, Munich, Germany

ARTICLE INFO

Article history:
Received 2 September 2011
Received in revised form
1 December 2011
Accepted 1 December 2011
Available online 19 December 2011

Keywords:
Baked lighting
Pre-computed lighting
Lighting texture
Shading

ABSTRACT

In real-time rendering, global lighting information that is too expensive to be computed on the fly is typically pre-computed and *baked* as vertex attributes or into textures. Prominent examples are view independent effects, such as ambient occlusion, shadows, indirect lighting, or radiance transfer coefficients. Vertex baking usually requires less memory, but exhibits artifacts on large triangles. These artifacts are avoided by baking lighting information into textures, but at the expense of significant memory consumption and additional work to obtain a parameterization. In this paper, we propose a memory efficient and performant hybrid approach that combines texture- and vertex-based baking. Cheap vertex baking is applied by default and textures are used only where vertex baking is insufficient to represent the signal. Seams at transitions between both representations are hidden using a simple shader which smoothly blends between vertex- and texture-based shading. With our fully automatic approach, we can significantly reduce memory requirements without negative impact on rendering quality or performance.

© 2011 Elsevier Ltd. All rights reserved.

1. Introduction

High quality rendering requires global illumination effects, in particular soft shadows, environmental lighting, indirect illumination, and reflections. All of them are computationally expensive and cannot be computed in real-time for complex scenes. The common solution is to precompute view independent effects, e.g., soft shadows, ambient occlusion, indirect illumination or radiance transfer coefficients and store them as vertex attributes or textures.

Storing precomputed lighting data is commonly referred to as *baking*. Two methods are well established: storing the data with the mesh vertices or in textures. *Vertex baking* requires only memory for lighting data at the mesh vertices, however, sub-triangle information cannot be reconstructed. Hence, *texture-based baking* is typically preferred, since textures can provide a larger number of samples and a more uniform sample distribution. Besides memory for the samples, textures require a parameterization of the mesh, which often involves manual intervention and texture coordinates as an additional vertex attribute.

The motivating application for this paper is the high-quality visualization of complex objects for product design and review, where important decisions are made upon the visual appearance of the models. This usually requires high resolution textures for lighting, which in practice occupy the majority of available video memory. Between review sessions, models are edited, so manual intervention during unwrapping and tweaking parameters is time-consuming. Hence, fast and fully automatic solutions are desirable. Industry models are typically modeled using free form surfaces, which are tessellated into very small triangles for interactive rendering. Therefore, we observe that vertex baking is often sufficient for the majority of scene triangles and only for a fraction of triangles artifacts appear (we will discuss vertex versus texture baking in detail in Section 3).

In this paper, we propose a simple hybrid representation which stores lighting data with the vertices by default, and only uses textures where necessary. To realize this simple idea, we use the following pipeline:

- First, the triangles for which vertex-based interpolation is sufficient are determined. The lighting information at the vertices is computed by sampling and solving a linear optimization problem (Section 4.1).
- All triangles that require texture are extracted, segmented, and unwrapped to a texture atlas. Since these chunks are usually much smaller than the original mesh, and typically form simple strips, unwrapping them is much cheaper than unwrapping the complete mesh. The lighting information is then baked into the generated texture atlas (Section 4.2).
- During rendering, the lighting data is merged using a simple shader that can do both, vertex-based interpolation and texture lookups, and smoothly blends the results (Section 4.3).

An example of our method can be seen for the Utah Fairy scene with baked ambient occlusion in Fig. 1. As can be seen from the classification, only a subset of the triangles require texture (red triangles in center image).

* Corresponding author. Tel.: +49 9131 8529928; fax: +49 9131 8529931.
E-mail address: henry.schaefer@cs.fau.de (H. Schäfer).

0097-8493/$ - see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cag.2011.12.001

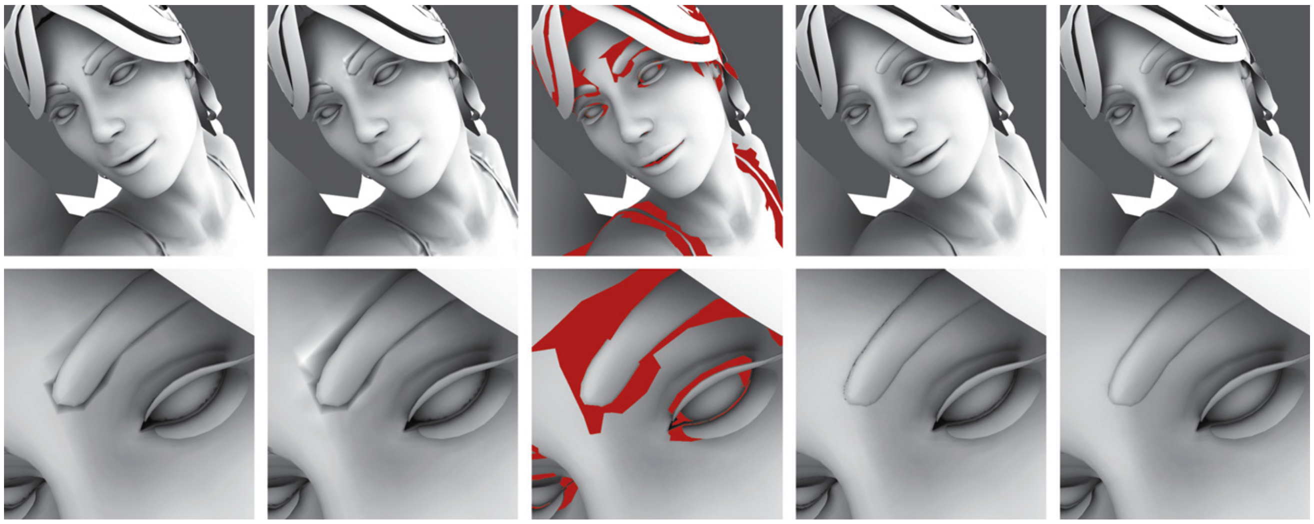


Fig. 1. Baked ambient occlusion for the Utah Fairy model. From left to right: vertex baking with sampling at vertices; vertex baking with least-squares fitting; classification of triangles where signal cannot be reconstructed using fitting approach; our method; ray traced reference. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

2. Previous work

Precomputed lighting: Much work has been done on precomputing lighting using, e.g., Monte–Carlo ray tracing, photon mapping or radiosity solvers. Besides effects like shadows and indirect lighting, more advanced methods have been developed such as precomputed radiance transfer [19], incident light probes using spherical harmonics [16,2], radiosity normal mapping [6] or irradiance normal mapping [7]. The resulting data ranges from 1D 8-bit scalar data up to 36D RGB coefficients. Most modeling packages provide tools to bake such precomputed lighting information into texture atlases or as vertex attributes.

Texture compression: For interactive applications it is important that texture access is fast and can be random, which typically results in fixed-rate compression schemes with 2–4 bits per pixel [1,4]. Some of these schemes are also part of DirectX and OpenGL [14,9]. While these schemes are tailored for 8-bit color data, extensions to HDR content is presented in [15]. These block-based compression schemes can introduce artifacts making the blocks visible. Stachera et al. solve this problem by using a hierarchical compression scheme [20]. For higher dimensional data, channel triples are usually compressed separately [8]. Hardware-supported compression schemes such as S3TC perform less optimal for 1D data such as shadows or ambient occlusion. Rasmusson et al. describe a texture compression scheme based on profile functions [17], which is tailored for such single channel lighting data.

An orthogonal approach to better exploit texture memory is to use signal-specialized texturing [18]. Here, texture coordinates are optimized such that texture space is gained where needed. In the computer game Halo 3 [8], lighting textures are compressed using a simple variant of signal-specialized parameterization. Smooth regions of the resulting atlas are cut out and parameterized separately to assign less space in the texture domain. Finally, standard texture compression is used. This approach has some similarities with our method, however, no hybrid storage scheme is used.

Vertex-based storage: The idea of Mesh Colors [21] is to store texture data directly with the triangles. In the simplest case, this corresponds to vertex-based storage, but it is also possible to store many more samples per triangle by virtual subdivision. In general, our method could also be based this approach, but the Mesh Colors signal reconstruction would reduce rendering performance. Recently, vertex baking was examined in detail and the computation of vertex colors was improved [11]. Ambient occlusion information is sampled densely on a mesh and least-squares fitting is employed to compute optimal color values at the vertices. A regularization term is introduced to smooth out mach-band artifacts that appear when least-squares optimization generates large gradient discontinuities.

3. Vertex baking vs. texture baking

Baking light information as vertex data is very simple: the information is attached to the vertices as an attribute which is automatically interpolated during rasterization [5]. However, sample density is static and no sub-triangle information can be represented. Subdividing the mesh until the signal can be adequately represented makes it necessary to also duplicate all other attributes. In practice this is mostly not an option, as this results in significant memory overhead.

Vertex-based baking generally works well for low frequency signals or if the vertices are dense enough to reconstruct the signal. In particular for lighting data, vertex density often corresponds very well with the frequency of the signal. Consider Fig. 2 as example, where we stored lighting from an area light source with the vertices (center) and in textures (right). Along the clearance of the door, vertex-baking reconstructs the signal very well (center top) because the high curvature results in both a high-frequency signal but also a finely tessellated surface. Even high resolution textures ($2k \times 2k$) cannot achieve comparable quality (right top). However, along shadow boundaries on flat regions, vertex-baking generates artifacts (center bottom), whereas lighting textures can resolve this detail much better (bottom right). Assuming a good parameterization of the surface, texture resolution is directly related to sample density. Hence, the user can easily control quality by adapting texture resolution.

One possible solution to improve quality or decrease memory consumption for lighting textures is to adapt texture resolution depending on lighting data, i.e., to distort the parameterization such that texture space is increased in areas with high lighting detail and decreased in smooth regions. Such *signal-specialized* parameterizations have been examined for color textures previously [18] and are also applied in a very simple form to light maps in current computer games [8].

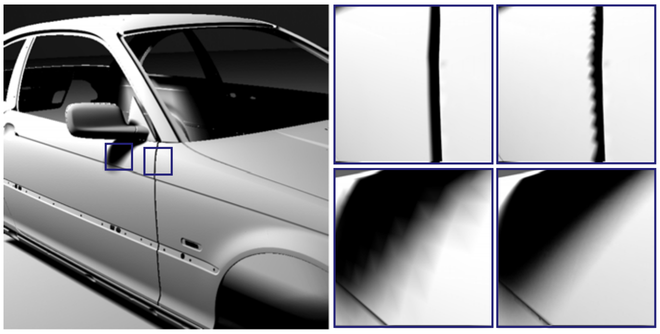


Fig. 2. Artifacts related to vertex-based (center) and texture-based storage (right) for baked soft shadows. While texture-based storage is much better in the shadow boundary region (bottom), the limited texture resolution causes problems in areas of high curvature like the clearance of the door (top). Vertex-based storage works well in this case since the model is densely tessellated in this area.

We also experimented with this option by computing an optimal signal-dependent texture size for each triangle and passing this to a parameterizer. Unfortunately, none of the examined parameterizers, including well-established ones such as LSCM [12] and ARAP [13], were able to generate a good parameterization based on this input. The reason is that the requested triangle sizes are very different, and rapidly change between neighboring triangles. As a result, the parameterization often contained overfolds, and the requested texture space could not be provided for many triangles. Robust parameterization was only possible by smoothing the requested texture sizes or by applying excessive segmentation into charts. We finally turned down this option due to its lack of robustness.

The result of our experiments was that we should ideally apply a combination of vertex- and texture-based storage. Vertex-based data is cheap and usually suffices for the majority of triangles. Texture should be applied only where necessary. A parameterization is then to be found only for the high-detail triangles. In combination with a proper segmentation, these triangles can then be parameterized well and stored in a texture atlas.

4. Pipeline

In the following, we describe a pipeline that realizes this hybrid storage scheme. We first sample the lighting data uniformly and compute optimized vertex values using a method similar to [11]. Then we determine for each triangle whether vertex-based storage is sufficient to reconstruct the signal, or texture-based storage is necessary (Section 4.1). For triangles containing high frequency detail, we compute a parameterization and store lighting data in a texture (Section 4.2). At render time, we use a hybrid shader blending between the two modes (Section 4.3). Finally, we describe a memory efficient storage scheme for scalar and multi-channel attributes (Section 4.4).

4.1. Sampling and classification

We start by distributing Poisson samples on the input geometry and sampling the lighting at these positions. The radius for Poisson disc sampling can be chosen such that detail up to a desired resolution in world space is captured. Then, we compute the colors at the mesh vertices. Theoretically, these values could also be directly sampled from the given signal. However, this simple *point sampling* approach does not consider the signal within the triangles surrounding a vertex.

In practice, we found that we obtain the best results if we fit the vertex colors to the sampled signal in a least-squares sense

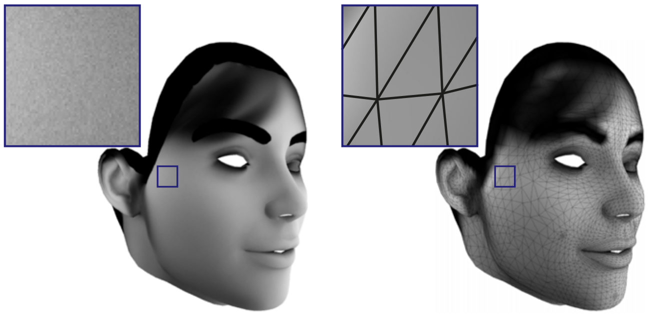


Fig. 3. Ambient occlusion information stored in point cloud generated by Poisson sampling the input geometry (left) and the least-squares fitted vertex colors (right).

similar to [11]. Given the Poisson samples $\mathcal{P} = \{p_1, \ldots, p_m\}$, where for each sample p we store the color value $f(p)$ and the index T_i of the triangle from which p is sampled. Let $\alpha(p)$, $\beta(p)$, $\gamma(p)$ be the barycentric coordinates of a sample p within triangle T_i and r, s, t the indices of the respective mesh vertices. Let $c(k)$ be the color at mesh vertex k. The color value $I(p)$ when using vertex-based coloring is computed as the barycentric interpolation of the colors at the vertices of triangle T_i

$$I(p) = \alpha(p) \cdot c(r(p)) + \beta(p) \cdot c(s(p)) + \gamma(p) \cdot c(t(p)). \tag{1}$$

The least-squares error F for a given set of vertex colors is then defined as the sum of squared differences between an actual sample color $f(p)$ and the linearly interpolated color $I(p)$ at this position, weighted by the area a_p represented by the sample p

$$F = \sum_{p \in \mathcal{P}} a_p (I(p) - f(p))^2. \tag{2}$$

Since $\mathcal{P}$ is a Poisson sampling of the triangle mesh, we can assume a_p to be constant and neglect it during the optimization. The set of optimal color values at the mesh vertices minimizes the above energy function F, which is a quadratic function of the unknown vertex colors. Thus, minimizing F results in a system of linear equations

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{3}$$

with

$$\mathbf{A}_{ij} = \begin{cases} \sum_{p \in T_{ij}} \text{bary}(p,i) \cdot \text{bary}(p,j), & i \neq j, \\ \sum_{p \in F_i} (\text{bary}(p,i))^2, & i = j. \end{cases}$$

$$\mathbf{b}_i = \sum_{p \in F_i} \text{bary}(p,i) \cdot f(p), \tag{4}$$

where T_{ij} is the set of triangles adjacent to the edge connecting vertex i and vertex j, F_i is the set of triangles at the fan of vertex i, and bary(p,i) the barycentric coordinate of the sample p that corresponds to the vertex i. $\mathbf{x}$ is the vector containing the unknown vertex colors $\mathbf{x} = [c_0\ c_1\ \cdots\ c_n]^T$. The system matrix $\mathbf{A}$ in Eq. (3) is sparse as it contains on average only seven non-zero entries per row. Therefore, the linear system can be efficiently solved using a sparse direct solver such as UMFPACK [3]. Finally, we store the fitted colors $\mathbf{x}$ as color attribute of the mesh vertices (Fig. 3).

Having obtained the lighting information at the mesh vertices and at the Poisson samples, we can easily classify the triangles. For each triangle T_i, we define the fitting error $E(T_i)$ as the maximal difference between the color value $f(p)$ at a Poisson sample inside the triangle and the respective interpolated value $I(p)$

$$E(T_i) = \max \|f(p) - I(p)\|_\infty \quad \forall p \in T_i. \tag{5}$$

If the error $E(T_i)$ exceeds a user defined threshold, the signal cannot be reconstructed by vertex colors and the triangle is added to a list of texture triangles.

4.2. Texture atlas generation

After we have identified the triangles that require texture to accurately represent the contained signal, we automatically generate a texture atlas for these triangles. An overview of the steps described in the following is shown in Fig. 4.

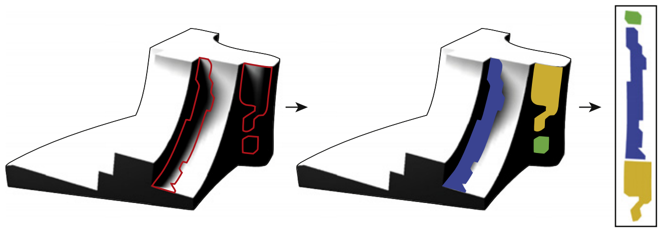


Fig. 4. Automatic texture atlas generation: after determining the triangles that require texture (red), we segment these parts of the mesh into almost developable charts which are further parameterized and packed into a texture atlas (right). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Usually, the classification stage does not optimally separate the mesh into texture- and color-triangle sub meshes. In practice, there will be color-triangles within areas of texture-triangles and vice versa. By applying a sequence of morphological closing operations on the set of texture-triangles, we close small holes and merge isolated texture-triangles to form larger charts. Both effects will later improve robustness of parametrization and utilization of the texture atlas. The closing operation consists of two steps as shown in Fig. 5. Given an input mesh and a classification into texture and interpolation triangles, we first mark all triangles that share an edge with a texture triangle as texture triangles as well. Then, all triangles that share only one edge with a texture triangle are reverted to interpolation triangles. In our implementation we apply this operation twice, which significantly reduces the number of generated charts.

To unwrap the set of texture-triangles onto a texture atlas, we use a variant of the D-Charts [10] algorithm. The algorithm segments a triangle mesh into almost developable parts using a Dijkstra-like chart growing combined with an iterative Lloyd optimization of the charts.

Next, we compute a parameterization for each chart individually using the As-Rigid-As-Possible (ARAP) surface parameterization method by Liu et al. [13]. The parameterizations of the charts are then rotated such that they have their largest extend in y direction. Since we have segmented the mesh into almost developable parts and parameterized the chart in an ARAP manner, we can assume that the distortion of the single triangles in the parametrization is minimal. We use this assumption to automatically determine the texture resolution that is required to faithfully capture the details at which the signal on the triangle mesh was initially sampled. Let w be the width of the bounding box of the chart and $\boldsymbol{\delta}$ the radius used for Poisson sampling (both units are defined in world space). If we choose the chart width in pixels as $w/\boldsymbol{\delta}$, the size of a texel in world space will roughly correspond to the Poisson sample distance.

All parameterized charts are then packed into a texture atlas using the method described by Lévy et al. [12]. Finally, we compute the lighting information at the texels of the texture atlas by lifting the texels into 3D space and sampling the lighting at these positions. To avoid texture interpolation artifacts at chart boundaries, we dilate the textures of the single charts into the empty space of the atlas. A smooth transition between adjoining charts is accomplished by cloning vertex colors from the neighboring chart instead of simply repeating the boundary colors.

4.3. Shading

As a result of the previous step, we get a classification into vertex color triangles and texture triangles. Considering the transition region between vertex color triangles and texture triangles, two problems arise. First, a vertex adjacent to a color-triangle and a texture-triangle requires a color *and* a texture coordinate attribute. Second, along the edges between color- and a texture-triangles, small discontinuities may appear that are well perceived.

Both problems are solved by transferring our classification from the triangles to the vertices. We classify all vertices with at least one adjacent texture triangle as *texture-vertices* and all vertices surrounded only by vertex color triangles as *color vertices*. Texture-vertices are then attributed the texture coordinate. The fitted vertex colors are assigned to the color-vertices. In Section 4.4 we describe how these hybrid attributes can be stored efficiently. Next, we have to find a shading scheme that generates a continuous transition between vertex-based and texture-based shading.

We explain our shading scheme using the illustrating example in Fig. 6, showing a simple mesh with four triangles. The leftmost triangle with vertices (v_1,v_2,v_3) requires texture to make the checker board texture visible. The right most triangle (v_4,v_5,v_6) has three color-vertices and thus uses linear interpolation to determine

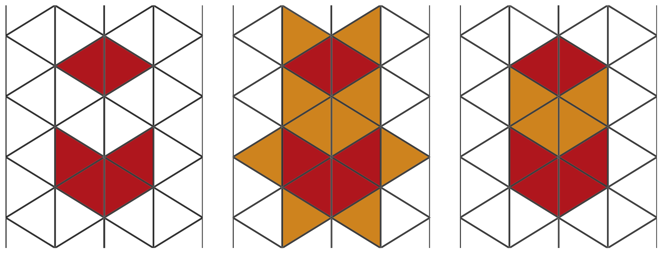


Fig. 5. Example of a single closing step on an input classification with red texture triangles (left). First, direct neighbors of texture triangles are also marked as texture triangles (center). Then, the marker flag is removed on triangles that share only one edge with texture triangles (right). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

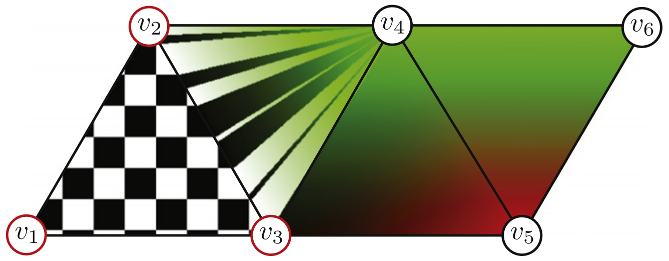


Fig. 6. Illustrating example of our interpolation scheme showing mesh with four triangles corresponding to the four possible input cases: all vertices belong to texture triangles (left with red vertices), one or two vertices adjacent to texture triangles (inner triangles) and a pure vertex color interpolation triangle (right). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

colors. The inner triangles (v_2,v_3,v_4) and (v_3,v_4,v_5) are mixed forms with two or one vertices adjacent to a texture triangle, respectively. In triangle (v_2,v_3,v_4) the checker board texture along edge (v_2,v_3) is linearly blended with the color at vertex v_4, resulting in a continuous transition along edge (v_2,v_3). Triangle (v_3,v_4,v_5) with one texture- and two color-vertices is rather simple again. We fetch the texture value at the texture-vertex in the vertex shader and output this as vertex color, which is then interpolated with the other vertex colors. Note that this example is for illustration purpose only, since triangle (v_2,v_3,v_4) is obviously not suited for linear interpolation and would be detected by our classifier. But even with a strict classifier, a discontinuity in the texture along an edge next to an interpolation triangle appears and may be well visible as shown in the left image of Fig. 7 where we increased contrast to visualize the problem. Our scheme allows smooths transition as shown in the right image of Fig. 7.

All described triangle cases shown in Fig. 6 can be handled using a simple pair of a vertex and a pixel shader with three varying attributes: a color C, a texture coordinate attribute UV, and a scalar mixing value η. If a vertex v_i is a texture-vertex, its texture coordinate is uv_i, for a color-vertex its lighting attribute is a color c_i (any other attribute type could also be used). For a color-vertex, the vertex shader sets C to c_i and UV to $(0,0)$, for a texture-vertex C is set to zero and UV to uv_i. The mixing parameter η is set to 1 for color- and to 0 for texture-vertices. Let $\texttt{tex}(UV)$ be the texture lookup function at UV. If, for example, we render an inner point of a triangle with barycentric coordinates (α,β,γ), the cases above can be handled as follows:

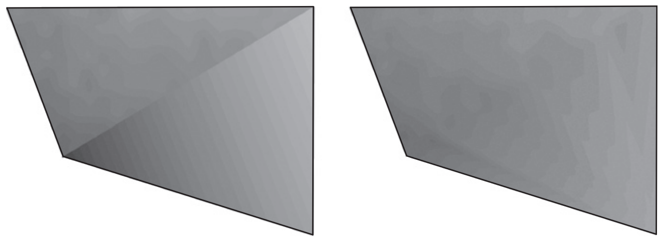


Fig. 7. Example of a seam between a texture (upper) and an interpolation triangle (lower) in the left image. Our interpolation scheme allows smooth transition between both triangles as shown in the right image. Note that we increased contrast to exaggerate the effect for visualization.

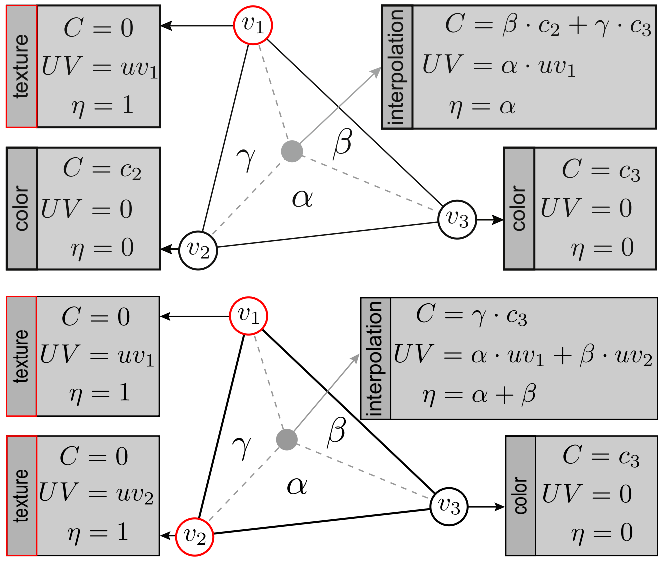


Fig. 8. Interpolation on triangles with one texture-vertex and two color-vertices (top) or two texture-vertices and one color-vertex (bottom).

All vertices are texture-vertices: UV is interpolated and used to fetch the lighting data from texture, C is ignored. The output is $\texttt{tex}(UV)$.

All vertices are color-vertices: C is interpolated, UV is ignored. The output color is the interpolated C.

Two color-vertices and one texture-vertex: We fetch the color of the texture-vertex and use barycentric interpolation (see Fig. 8 (top)), resulting in $\alpha \cdot \texttt{tex}(uv_1)+\beta \cdot c_2+\gamma \cdot c_3$. We can compute this color from the interpolated values of UV, C and η as $\eta \cdot \texttt{tex}(UV/\eta)+C$.

Two texture-vertices and one color-vertex: In order to be continuous along a texture edge (see Fig. 8 (bottom)), we interpolate the texture coordinates along the edge using β and γ, resulting in a texture color $\texttt{tex}((\beta \cdot uv_1+\gamma \cdot uv_2)/(\beta+\gamma))$. This value is blended with the color c_3 using α. Using the interpolated attributes, we can compute the final color as $\eta \cdot \texttt{tex}(UV/\eta)+C$.

We notice that the expression $\eta \cdot \texttt{tex}(UV/\eta)+C$ handles all cases correctly. Transformed to shader code this results in the following pair of vertex and pixel shader:

```
Vertex shader
out float C, vec2 UV, float eta;
C=is-color-vertex() ? color : 0;
UV=is-color-vertex() ? 0 : texcoord;
eta=is-color-vertex() ? 0 : 1;

Pixel shader

in float C, vec2 UV, float eta;
uniform sample2D lightMap;
float color =C;
if(eta>0)color += eta
            * texture(lightmap, UV/eta);
return color;
```

4.4. Hybrid attribute storage

Our interpolation scheme requires that each vertex stores either a color (or any other lighting data) or a texture coordinate. Instead of passing three vertex attributes (color, texture coordinate, selector flag), we can easily optimize.

Lighting data is scalar value: We only use a single texture coordinate attribute. For color-vertices with scalar lighting value c, we set the texture coordinate to $(c,2)$. Since the texture coordinates of texture-vertices point to the unit square, we can easily identify color-vertices and extract the lighting value.

Lighting data is RGB color: We use a single texture coordinate attribute. For color-vertices, we store a table with unique vertex color data in a separate uncompressed texture. The texture coordinate is set to the corresponding position and 2 is added to the second texture coordinate to flag color-vertices. In practice, we can avoid the second texture and store the vertex colors in free spots of the atlas texture. However, this generates artifacts when texture compression is used.

5. Results

In Fig. 9 we show results of our method for the Utah Fairy scene with baked ambient occlusion, a complex car model with soft shadows of an area light source and baked indirect lighting for a car seat. Detail comparisons to vertex-based baking, a ray traced reference using 1024 samples per pixel and classification results are available in Figs. 11–13. For vertex-baking we examine point sampling (sampling the signal at the vertex positions) and least-squares fitting as described above.

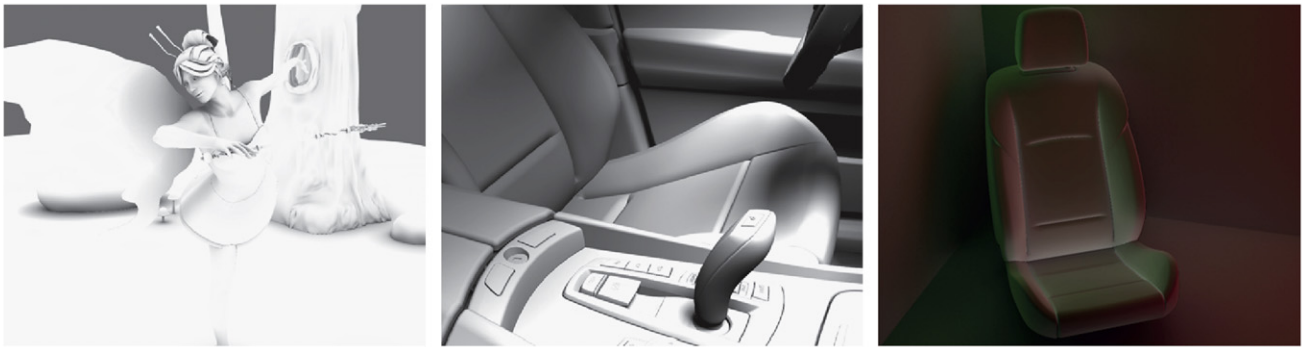


Fig. 9. Results for test scenes with baked ambient occlusion (left), soft shadows (center) and indirect illumination (right) using our hybrid method.

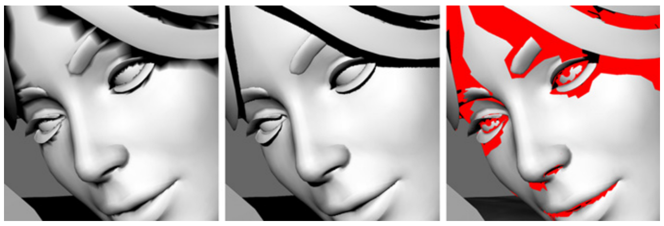


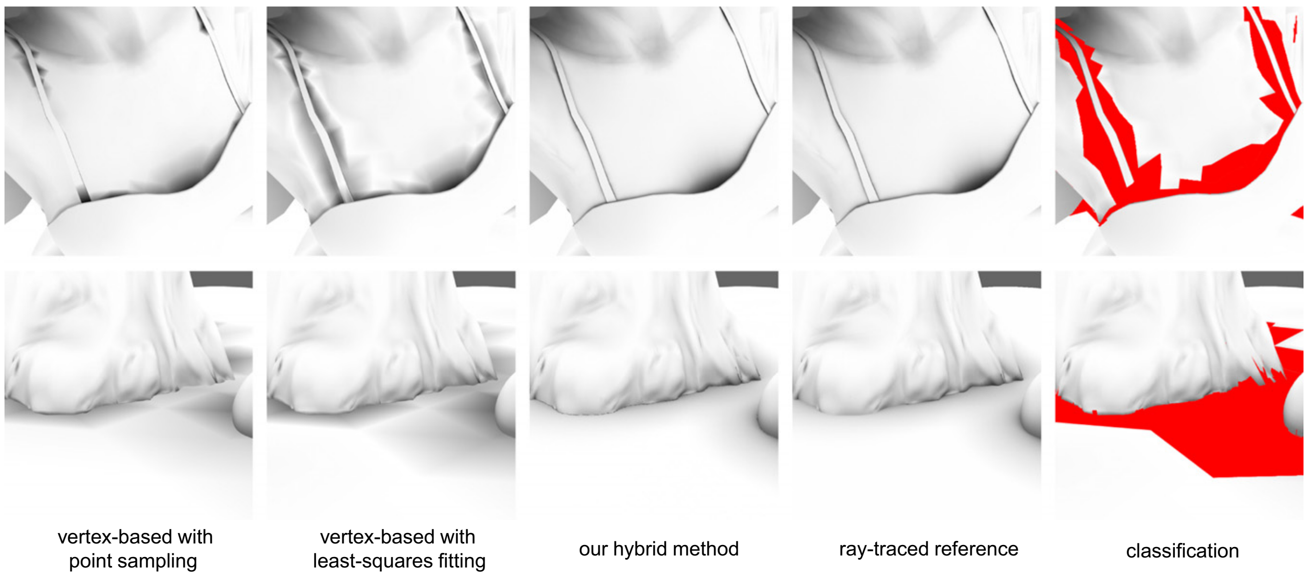
Fig. 10. Illustrating example of combined high and low frequency signal using baked hard shadows and Lambertian shading for the fairy head. A pure vertex based representation (left) cannot represent the shadow boundary, our method (center) uses texture in these regions which can be seen in the classification (right).

Vertex baking with point sampling is the cheapest method in terms of computation time but introduces artifacts on large triangles since sub-triangle information cannot be reconstructed. The least-squares fitting approach already improves the quality by smoothing the result but introduces new artifacts on triangles with large gradient variation. These can be reduced using a regularization as proposed by [11], however dark seams remain and the regularization also removes desired detail.

In Fig. 10 we examine a combination of a low and high frequency signal on the fairy head. We store lighting information for a point light source to generate hard shadow boundaries combined with Lambertian shading which changes smoothly on the surface. While the pure vertex based representation (left) works well in smooth regions, artifacts are visible at the shadow boundary. Our method detects these regions (right) and switches to texturing (center). This example also shows that no regularization is required, since triangles with large gradient variation are recovered by textures.

We only require the user to define a desired sample density and an upper bound on the interpolation error for the classification. For the examined test scenes we choose the sample density just fine enough to represent all relevant detail. If for example detail with an accuracy of 4 mm is desired, a sample distance of 2 mm has to be chosen (Nyquist's theorem). We set the interpolation error bound to 3% to remove noise introduced by the lighting algorithms. For the Utah Fairy scene with baked ambient occlusion we reduce the mesh area that requires texturing in the worst case to 32% for the body and in the best case to 2% for the tree. The tree behaves particularly well because it is finely tessellated, and textures are mainly required where the tree penetrates the ground.

The rendering times were measured using an Nvidia GTX 480 graphics card at a resolution of 1024^2. Although our method requires additional shader code, we did not observe a negative impact on rendering performance. Table 1 summarizes the results, including rendering times and compression rates.


vertex-based with point sampling
vertex-based with least-squares fitting
our hybrid method
ray-traced reference
classification
Fig. 11. Utah Fairy scene with ambient occlusion.

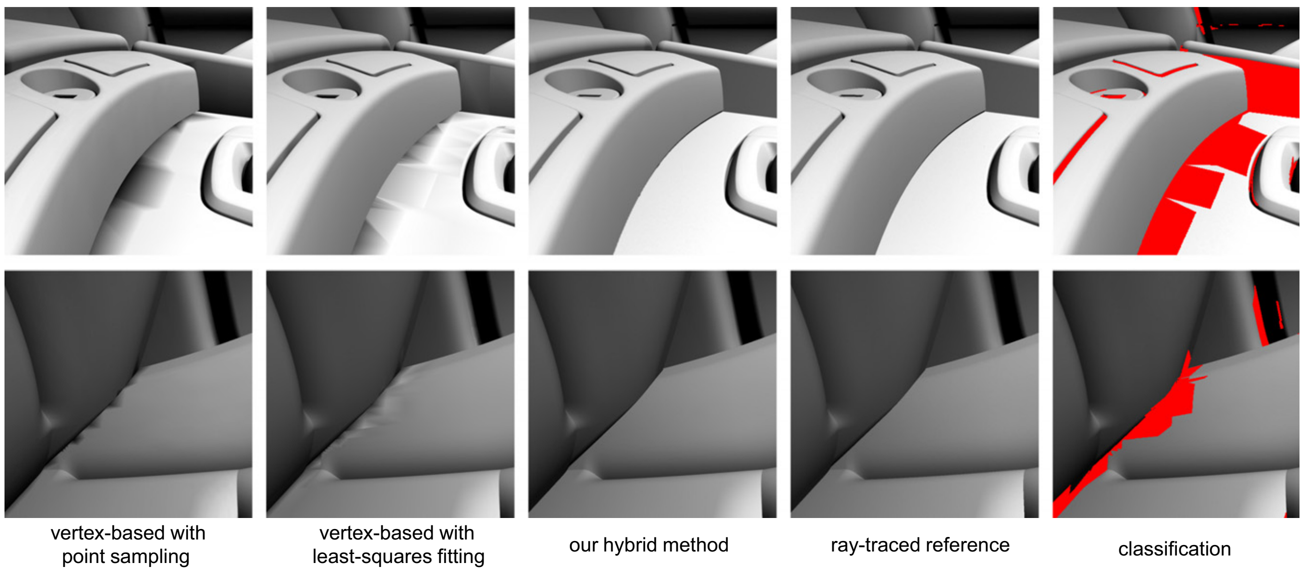

vertex-based with point sampling
vertex-based with least-squares fitting
our hybrid method
ray-traced reference
classification

Fig. 12. Complex car scene with soft shadows.

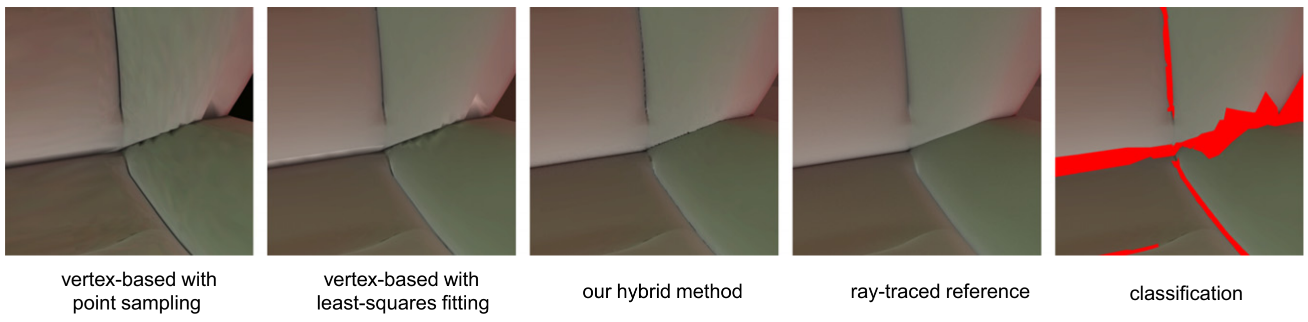

vertex-based with point sampling
vertex-based with least-squares fitting
our hybrid method
ray-traced reference
classification

Fig. 13. Car seat with indirect illumination.

Table 1
Comparison of results for our test scenes with baked ambient occlusion for the fairy, soft shadows from area lights for the car and indirect lighting for the seat scene. Max E_{ps} and max E_{lsqr} show the maximum difference between the sampled points and the linear interpolation, where the vertex values were determined using point sampling and least squares fitting. Further, rendering times are given for pure texture and our approach. The last column contains the average textured mesh area using our hybrid method where otherwise the complete mesh would have to be textured.

Scene	Vertices	Triangles	Max. E_{ps}	Max. E_{lsqr}	Rendering (in ms)		Avg. textured mesh area (%)
					Texture	Our	
Fairy (ambient occlusion)	115k	199k	0.68	0.41	4.88	4.85	23
Car (shadow)	7.531M	13.578M	0.89	0.57	8.62	8.48	17
Seat (indirect illumination)	118k	160k	0.21	0.09	1.27	1.24	12

Besides sampling of the lighting information, most of the pre-processing time is spent on segmentation, unwrapping and generating the texture atlas. With our method, these stages are usually required only on a fraction of the initial mesh, which greatly improves pre-processing time. Additionally, the classification into texture and interpolatable triangles serves as a pre-segmentation, which also improves robustness of segmentation and parametrization.

6. Limitations and conclusion

We introduced a simple and memory efficient technique for baking precomputed lighting. Our method combines the advantages of vertex-based (low memory footprint) and texture-based storage (high resolution also on large triangles). We showed that our method generates a close to optimal solution in a fully automatic process. Rendering requires only little additional shader code which has no negative impact on rendering time. Our method allows to use standard texture compression methods to further reduce memory requirements. Moreover, the proposed sample distance allows to intuitively control quality. The achieved compression rates depend on the number of triangles that allow to represent the lighting information by linear interpolation. Hence, the memory footprint cannot be reduced significantly in areas with high signal variation and for low polygon models. When a vertex based representation is not sufficient we always use textures as fallback solution. So in the worst case, the whole mesh has to use textures, whereas in the best case only vertex-based baking is employed.

References

[1] Campbell G, DeFanti TA, Frederiksen J, Joyce SA, Leske LA. Two bit/pixel full color encoding. Computer graphics (Proceedings of SIGGRAPH'86) 1986;20:215–23.
[2] Chen H, Liu X. Lighting and material of Halo 3. In: ACM SIGGRAPH 2008 classes; 2008.
[3] Davis TA, Duff IS. An unsymmetric-pattern multifrontal method for sparse *LU* factorization. SIAM J Matrix Anal Appl 1997:140–58.
[4] Fenney S. Texture compression using low-frequency signal modulation. In: Proceedings of HWWS'03; 2003. p. 84–91.
[5] Gouraud H. Continuous shading of curved surfaces. IEEE Trans Comput C 1971;20:623–9.
[6] Green C. Efficient self-shadowed radiosity normal mapping. In: ACM SIGGRAPH 2007 courses; 2007.
[7] Habel R, Wimmer M. Efficient irradiance normal mapping. In: Proceedings of I3D 2010; 2010. p. 189–95.
[8] Hu Y, Wang X. Lightmap compression in Halo 3. In: Game developer conference 2008; 2008.
[9] Iourcha K, Nayak K, Hong Z. System and method for fixed-rate block-based image compression with inferred pixels values. US Patent 5,956,431; 1999.
[10] Julius D, Kraevoy V, Sheffer A. D-Charts: quasi-developable mesh segmentation. In: Computer graphics forum (Proceedings of EG'05); 2005. p. 581–90.
[11] Kavan L, Bargteil AW, Sloan PP. Least squares vertex baking. Computer graphics forum (Proceedings of EGSR'11) 2011;30:1319–26.
[12] Lévy B, Petitjean S, Ray N, Maillot J. Least squares conformal maps for automatic texture atlas generation. ACM Trans Gr 2002;21:362–71.
[13] Liu L, Zhang L, Xu Y, Gotsman C, Gortler SJ. A local/global approach to mesh parameterization. In: Proceedings of SGP'08; 2008. p. 1495–504.
[14] McCabe D, Brothers J. DirectX 6 texture map compression. Game Developer Mag 1998;5(8):42–6.
[15] Munkberg J, Clarberg P, Hasselgren J, Akenine-Möller T. High dynamic range texture compression for graphics hardware. ACM Trans Gr 2006;25:698–706.
[16] Ramamoorthi R, Hanrahan P. An efficient representation for irradiance environment maps. In: Proceedings of SIGGRAPH'01; 2001. p. 497–500.
[17] Rasmusson J, Ström J, Wennersten P, Doggett M, Akenine-Möller T. Texture compression of light maps using smooth profile functions. In: Proceedings of HPG'10; 2010. p. 143–52.
[18] Sander PV, Gortler SJ, Snyder J, Hoppe H. Signal-specialized parametrization. In: Proceedings of EGRW'02; 2002. p. 87–98.
[19] Sloan PP, Kautz J, Snyder J. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In: Proceedings of SIGGRAPH'02; 2002. p. 527–36.
[20] Stachera J, Rokita P. Hierarchical texture compression. In: Proceedings of WSCG'06; 2006. p. 108–20.
[21] Yuksel C, Keyser J, House DH. Mesh colors. ACM Trans Gr 2010;29: 15:1–15:11.