Hardware Accelerated Per-Texel Ambient Occlusion Mapping

Tim McGraw, Brian Sowers

Department of Computer Science and Electrical Engineering, West Virginia University

Abstract. Ambient occlusion models the appearance of objects under indirect illumination. This effect can be combined with local lighting models to improve the real-time rendering of surfaces. We present a hardware-accelerated approach to precomputing ambient occlusion maps which can be applied at runtime using conventional texture mapping. These maps represent mesh self-occlusion computed on a per-texel basis. Our approach is to transform the computation into an image histogram problem, and to use point primitives to achieve memory scatter when accumulating the histogram. Results are presented for multiple meshes and computation time is compared with a popular alternative GPU-based technique.

1 Introduction

Ambient occlusion is a visual effect that can be used in computer graphics to improve the realism of simple lighting models. Local lighting models, such as the Phong lighting model [1] take into account the local surface geometry and the relative positions of light sources and the viewer, but neglect effects such as self-shadowing and occlusion.

Ambient occlusion is a *view-independent*, *indirect* lighting effect, so for rigid objects it can be precomputed. The values can be computed and stored per-vertex, per-triangle or per-texel. Per-vertex and per-triangle approaches may suffer from undersampling artifacts in areas of coarse triangulation and from long computation time for large meshes. After offline computation the ambient occlusion map can be used in real-time applications [2] with very little performance penalty. For example, the ambient occlusion factor can be incorporated in the Phong model by using it to modulate the constant ambient material color.

In this paper we will describe a hardware accelerated technique for precomputing ambient occlusion maps on a per-texel basis, demonstrate the effects of ambient occlusion on synthetic meshes, and compare computation time for our approach with another GPU-based implementation.

2 Background

Ambient occlusion [3] was suggested as a way of giving the appearance of global illumination [4] at a fraction of the computational cost. It quantifies the fraction of the hemisphere of ambient illumination which cannot reach the surface. This value can be reduced by concavities and shadowing. Ambient occlusion (AO) is formulated as

$$AO(x) = 1 - \frac{1}{\pi} \int_{\omega \in \Omega} V(x, \omega)(\omega \cdot n)\, d\omega \tag{1}$$

where x is a point on the surface, ω is a light direction, $V(x,\omega)$ is a visibility function which has value 0 when x is not visible from direction ω and has value one otherwise, and Ω is the hemisphere with $\omega \cdot n > 0$. The values of $AO(x)$ range from 0 for unoccluded points, and 1 for completely occluded points as illustrated in Figure (1). The idea of ambient occlusion has its roots in the more general concept of "obscurances" [5]. Obscurance values depend on distance to occluding objects and can be used to incorporate color bleeding effects. Precomputed radiance transfer [6] models more general light transport, including AO. Hardware approaches to AO computation have included per-

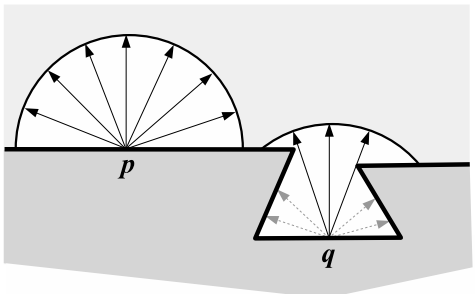


Fig. 1. Ambient occlusion at points p and q. Point p is unoccluded ($AO(p) = 0$) and point q is partially occluded ($AO(q) > 0$).

vertex [7] and per-triangle techniques using shadow maps [8], per-vertex techniques using occlusion queries [9], depth peeling [10] and using the fragment shader to compute *local* screen-space ambient occlusion based on neighborhood depth values [11],[12].

Figure (2) shows an example of how ambiguity between convex and concave regions can be resolved using AO. In general, these cases may be disambiguated my knowing the light direction or by moving the camera. By using AO the darker concave region is easily distinguished from the brighter convex region. Surface darkening in AO can also

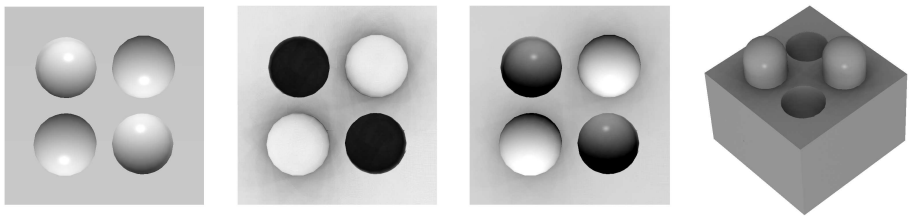


Fig. 2. Convex/concave ambiguity resolved with AO. From left to right : Phong lighting, AO only, AO + Phong, perspective view of Phong + AO.

be due to proximity of surfaces. The "contact shadows" provided by AO, as in Figure (3), can be a useful cue to suggest that two surfaces are touching, or nearly so. While shadowing techniques (shadow volumes or shadow mapping) can resolve convexity and proximity ambiguities, their appearance depends on the position of light sources in the scene, so they cannot be precomputed for dynamic scenes as AO can.

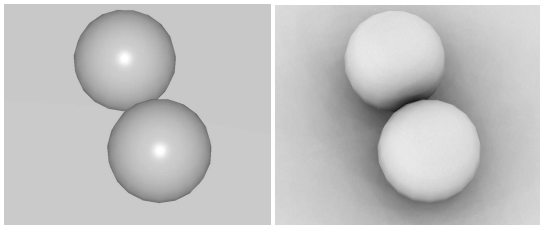


Fig. 3. Proximity cues from AO. Phong lighting (left), AO only (right).

3 Implementation

A brute-force approach to computing AO is to discretize the hemisphere of ambient light directions and form rays from the surface in each of these directions, then perform intersection tests between the rays and occluding objects. This is an example of what is commonly referred to as the inside-out approach. The other approach is outside-in : considering each irradiance direction and querying which surface points have unoccluded accessibility to this light direction.

We present an outside-in AO algorithm implemented in the OpenGL Shading Language (GLSL) [13]. The algorithm entails multiple renders to texture of the mesh with hidden surface removal provided by z-buffering. The approach we present to AO calculation does not require building a spatial data structure, such as a octree or k-d tree, which can be used to accelerate methods based on intersection queries. Our algorithm does require that mesh vertices have associated texture coordinates. This can be achieved by various mesh parameterization algorithms [14], [15] or texture atlas generation [16].

Since we are storing per-texel ambient occlusion, and we cannot assume that a texel is infinitesimally small, we define AO for a surface patch, S, as the average AO over the patch

$$AO(S) = 1 - \frac{\frac{1}{\pi}\int_{x\in S}\int_{\omega\in\Omega} V(x,\omega)(\omega\cdot n)\,d\omega\,dS}{\int_{x\in S} dS}. \tag{2}$$

Letting S be the patch covered by texel R, we can write AO in terms of the texture coordinates (u,v) as

$$AO(R) = 1 - \frac{\frac{1}{\pi}\int_{(u,v)\in R}\int_{\omega\in\Omega} V(x(u,v),\omega)(\omega\cdot n)|\frac{\partial x}{\partial u}\times\frac{\partial x}{\partial v}|\,du\,dv\,d\omega}{\int_{(u,v)\in R}|\frac{\partial x}{\partial u}\times\frac{\partial x}{\partial v}|\,du\,dv} \tag{3}$$

where $|\frac{\partial x}{\partial u} \times \frac{\partial x}{\partial v}| \, du\,dv$ is the surface area element of the parametric surface $x(u,v)$. Since the texture mapping functions $u(x), v(x)$ are linear over a triangle, the inverse mapping - the surface parameterization $x(u,v)$ is also linear over a triangle. So within a triangle $|\frac{\partial x}{\partial u} \times \frac{\partial x}{\partial v}|$ is a constant. Since the goal of most mesh parameterization and texture atlas generation algorithms is to minimize stretch, we will assume that the stretch is constant over a texel. Note that we have already shown that stretch is constant for texels entirely within a triangle. We are not assuming that stretch is constant over the entire surface, only that it changes slowly enough that we can assume that it is constant over each texel. We can then factor $|\frac{\partial x}{\partial u} \times \frac{\partial x}{\partial v}|$ out of the numerator and denominator and observe that

$$\frac{|\frac{\partial x}{\partial u} \times \frac{\partial x}{\partial v}|}{|\frac{\partial x}{\partial u} \times \frac{\partial x}{\partial v}| \int_{(u,v)\in R} du\,dv} = 1 \tag{4}$$

for a single texel, R. We can then write Equation (3) as

$$AO(R) = 1 - \frac{1}{\pi} \int_{(u,v)\in R} \int_{\omega\in\Omega} V(x(u,v),\omega)(\omega \cdot n)\, du\,dv\,d\omega. \tag{5}$$

Now we will rewrite the formulation in terms of light space coordinates. This coordinate system contains the image plane for each of the rasterized images of the mesh. Let x_ω, y_ω be the coordinates in this plane where the subscripts denote the dependence on the light direction ω. Let P be the patch in the image plane covered by the image of texel R. Let the Jacobian of the transformation from texture space to the image plane be given by $|J| = \frac{\partial(u,v)}{\partial(x_\omega, y_\omega)}$. Then we can rewrite Equation (5) as

$$AO(P) = 1 - \frac{1}{\pi} \int_{(x_\omega,y_\omega)\in P} \int_{\omega\in\Omega} V(x_\omega, y_\omega)(\omega \cdot n)|J|\, dx_\omega\, dy_\omega\, d\omega. \tag{6}$$

Equation (6) will by solved by discretizing the hemisphere Ω. The view vectors for each image are randomly generated. Uniformly distributed unit vectors can be produced by drawing each component from a zero mean normal distribution and normalizing the vector [17]. The image domain, will be discretized by the rasterization process. Z-buffering will eliminate pixels from the surface with zero visibility but the function V will still be used to discriminate mesh pixels from background by assuming $V = 0$ for the background. An overview of the algorithm implemented on the vertex processor (VP) and fragment processor (FP) is as follows:

Stage 1 : Render from light direction, ω. In VP, transform mesh vertices into light coordinates. In FP, compute $(\omega \cdot n)|J|$ using interpolated normals. Set the output fragment color as $[u(x,y), v(x,y), 1.0, (\omega \cdot n)|J|]$ where u, v are the quantized texture coordinates (as shown in Figure (4)).

Stage 2 : Sum over pixels. Render point primitives with additive alpha blending. In VP, use vertex-texture-fetch to read the output from stage 1. Set point position to $[u(x,y), v(x,y)]$ and color to $(\omega \cdot n)|J|$. Accumulate AO by repeating stages 1 and 2 for each light direction.

Stage 3 : Postprocessing Fix texture atlas seams and normalize the map in FP.

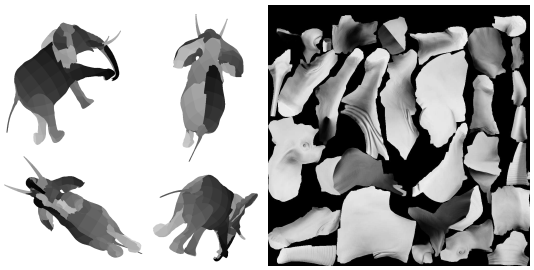


Fig. 4. (Left) Output of stage 1 for 4 random light directions: red channel = u(x,y), green channel = v(x,y), blue channel = 1. All channels = 0 in background. (Right) Output of stage 2.

Our approach to ambient occlusion computation is similar to computation of image histograms. To compute AO we count the number of times each texture coordinate appears the rendered images and scale this count by the stretch correction factor $|J|$ and the cosine of the angle of incidence $\omega \cdot n$. The image histogram can be defined as

$$H(u,v) = \sum_x \sum_y \delta(u - u(x,y), v - v(x,y)) \tag{7}$$

where δ is the 2D discrete dirac delta function defined as $\delta(x,y) = 1$ for $x = y = 0$ and $\delta(x,y) = 0$ otherwise.

Our hardware approach to computing the histogram is to render the image to a vertex buffer, and in the fragment program set the point location to the appropriate histogram bin location. By rendering the points with additive alpha blending the result is an image in the framebuffer of the histogram. Likewise, the ambient occlusion computation is a summation of the visible surface elements over a set of images acquired from multiple viewing directions. For per-texel ambient occlusion computed from the visible (u,v) images we have

$$AO(u,v) = 1 - C \sum_d \sum_x \sum_y \delta(u_d(x,y) - u, v_d(x,y) - v)|J(x,y)|(\omega_d \cdot n_d(x,y)) \tag{8}$$

Comparing Equation (8) to Equation (6) we see that the continuous integral over the hemisphere has been replaced by a discrete summation over directions. The summation over the light-space coordinates x and y is due to the fact that a single texel may cover more than one pixel. The delta function has replaced the visibility function. Enabling back face culling, or constraining the input mesh to be closed will take care of the case when the surface element faces away from the light, so that we do not accumulate values for the inside of the surface. In practice we do not explicitly compute the constant C. We simply assume that some point on the surface will be unoccluded and normalize the map so that the maximum value of $1 - AO(u,v)$ is one.

Output from stage 1 is rendered to floating-point frame buffer attached textures. In the second stage we render this same data as point primitives. This can be achieved by copying the attached textures as OpenGL pixel buffer objects, or by texture fetch in the vertex shader in stage 2. If desired, the "bent normal" [3] (the average unoccluded direction) can also be computed and accumulated in the first 2 stages of the algorithm.

3.1 Stretch Correction

The surface area covered by a texel may not be constant over the mesh, as shown in Figure (5). Note that this does not invalidate our earlier assumption that stretch is constant over each *texel*. The degree of stretch can be quantified by the determinant of Jacobian matrix of the mapping function from light coordinates (x,y) to texture coordinates (u,v).

$$J = det \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} \\ \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} \end{bmatrix} \tag{9}$$

If the stretch of the texture mapping is approximately constant then the Jacobian computation is not necessary as it can be absorbed into the constant C. The derivatives of

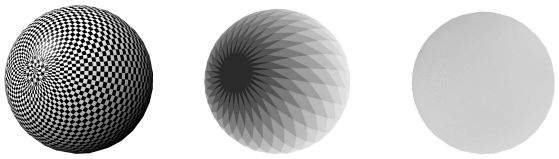


Fig. 5. Stretch correction with Jacobian determinant. Checkerboard texture illustrating stretch in the texture mapping (left) at the poles of the sphere, AO map computed without stretch correction (middle) and AO map computed with stretch correction (right).

varying quantities which are interpolated over rendered primitives can be queried in many shading languages. In OpenGL the dFdx(...) and dFdy(...) commands provide this functionality. The argument to the function is an expression whose derivative is to be computed. Typically these commands are used in mipmapping to determine the appropriate texture level-of-detail. The equivalent commands are ddx(...) and ddy(...) respectively in the DirectX HLSL [18] and in NVidia's Cg language [19].

In stage 1 the GLSL fragment shader computes the stretch factor by

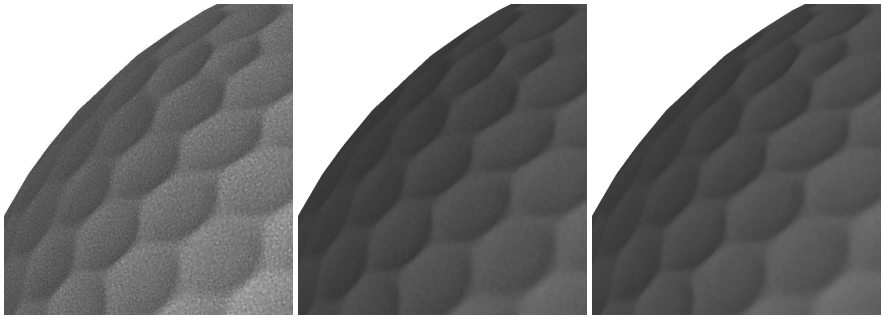
```
vec2 dx = dFdx(gl_TexCoord[0].st);
vec2 dy = dFdy(gl_TexCoord[0].st);
float J = abs(dx.x * dy.y - dx.y * dy.x);
```

Later, in stage 2, when additive alpha blending is used for summation, the fragment alpha value is multiplied by J, effectively implementing Equation (8).

3.2 Map Smoothing

The size of the points rendered in stage 2 can be used to impose smoothness on the resulting ambient occlusion map. When the point size is greater than 1 pixel, the points will overlap in the framebuffer, blurring sharp features in the map. This can help counteract the slight variations in intensity due to light direction randomness and undersampling of the ambient hemisphere. If antialiased points are rendered the size can include

fractional pixels, giving fine control over the smoothness of the final map. The effect of increasing the point size from 1 to 2 pixels is shown in Figure (6). In these images the texture intensity is AO^4 and nearest-neighbor texture filtering has been used to emphasize the variations in intensity.



(a) 1.0 (b) 1.5 (c) 2.0

Fig. 6. Contrast enhanced detail of golfball[5] AO map demonstrating smoothing using point size.

3.3 Fixing Seams

The results using this method will show small errors at seams in the texture atlas. This can be avoided by computing a mesh parameterization, or using a texture atlas method which results in no seams [20]. Otherwise we correct the results in stage 3 using a post-processing step which fixes the texels located at seams.

The morphological *erosion* and *dilation* operators are often used for binary image analysis and can be generalized to gray-scale images [21]. The erosion operator shrinks and thins regions in an image. Applying this to our AO image results in the boundary texels being discarded. Following this step we apply two steps of dilation. This operator grows and thickens regions in the image. The first dilation step replaces the boundary pixels with the nearest valid AO value, and the second step expands the AO map by 1 pixel further so that bilinear texture filtering will be correct at the chart boundaries in the AO texture atlas.

4 Results

The AO algorithm was implemented in OpenGL 2.0 and GLSL. Experiments were run on a desktop PC featuring Intel Quad Core QX6700 2.66 GHz CPU and 4 GB RAM, and the GPU was a GeForce 8800 GTX with 768 MB VRAM. For all images shown in this paper 512 directions were used to compute a 1024×1024 AO map.

Figure (7) shows the AO map computed for an elephant and the bones of the hand. The folds in the skin and concavities behind the ears of the elephant are clearly visible.

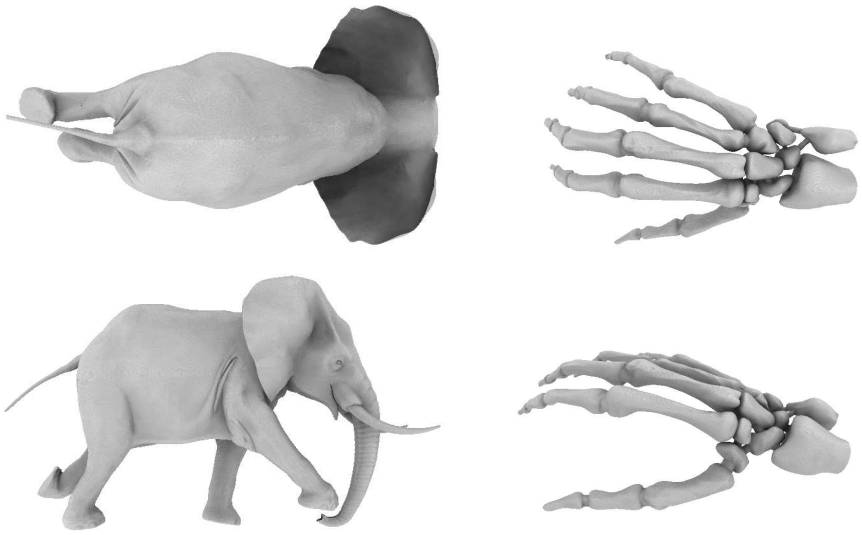


Fig. 7. Ambient occlusion for elephant (left) and hand (right).

Fine details in the trunk are emphasized. The spatial relationships of the small bones in the wrist are made apparent by the darkening due to proximity. Figure (8) shows the AO map computed for a mesh of the cortical surface of the brain. Note that the ridges and creases (gyri and sulci) are emphasized and clearly distinguishable from one another. Figure (9) shows the AO map computed for a rocker arm, brain surface and pelvic bone.

Fig. 8. Ambient occlusion for brain2.

4.1 Timing

Timing results are shown in Table (1). AO maps of resolution 512×512 and 1024×1024 were generated with 512 and 1024 light directions. For comparison, times required

Fig. 9. Ambient occlusion for rocker arm, brain1 and pelvis (left to right).

for GPU computation using shadowmaps are also tabulated. The implementation used was ATI's GPU MeshMapper tool. We have observed, both theoretically and ex-

mesh	**triangles**	**vertices**	**Our Time(sec)**	**Shadow Map Time(sec)**	**Our Time(sec)**	**Shadow Map Time(sec)**
			512 directions		1024 directions	
crank[1]	14998	7486	5.9	11.2	11.3	20.7
rocker arm[2]	20088	10044	5.9	11.1	11.8	18.5
pelvis[3]	63012	31438	6.2	11.3	12.3	22.6
elephant[4]	84638	42321	6.7	11.6	12.1	23.4
brain1[2]	290656	144996	7.3	19.6	14.6	39.3
brain2[5]	588032	294012	8.5	38.8	16.9	77.0
hand[6]	654666	327323	9.3	40.5	18.7	81.2

Table 1. Timing results for 1024×1024 AO map computations.

perimentally, that our algorithm is linear in the number of directions, so the times for 1024 view directions are approximately double those given in the Tables. Profiling our method shows that the majority of time is spent in the vertex processing phase of stage 2. Specifically, the vertex texture fetch is the most time consuming aspect of this phase. A more direct render-to-vertex buffer mechanism would likely result in a significant speed-up.

[1] Sam Drake, Amy Gooch, Peter-Pike Sloan (http://www.cs.utah.edu/ gooch/model_repo.html)
[2] Aim@Shape(http://shapes.aim-at-shape.net/)
[3] 3D-Doctor(http://ablesw.com/3d-doctor/)
[4] Robert Sumner, Jovan Popovic (Computer Graphics Group at MIT) (http://people.csail.mit.edu/sumner/research/deftransfer/data.html)
[5] Princeton University Suggestive Contour Gallery (http://www.cs.princeton.edu/gfx/proj/sugcon/models/)
[6] Georgia Institute of Technology Large Geometric Models Archive(http://www-static.cc.gatech.edu/projects/large_models/)

5 Conclusions

In this paper we have presented a new approach to precomputing ambient occlusion maps on the GPU. This technique was demonstrated on various surfaces including range scan data and several meshes from engineering and medical applications. Timing comparisons show that our algorithm is $4\times$ faster than the GPU-based shadow mapping technique for large meshes. For smaller meshes our method still has a time advantage of about $2\times$.

References

1. Phong, B.: Illumination for computer generated pictures. Communications of the ACM **18** (1975) 311–317
2. McReynolds, T., Blythe, D.: Advanced Graphics Programming Using OpenGL. Morgan Kaufmann (2005)
3. Landis, H.: Production-ready global illumination. RenderMan in Production (SIGGRAPH 2002): Course 16 (2002)
4. Dutre, P., Bala, K., Bekaert, P., Shirley, P.: Advanced Global Illumination. (2006)
5. Zhukov, S., Iones, A., Kronin, G.: An ambient light illumination model. Rendering Techniques **98** (1998) 45–55
6. Sloan, P., Luna, B., Snyder, J.: Local, deformable precomputed radiance transfer. Proceedings of ACM SIGGRAPH 2005 **24** (2005) 1216–1224
7. Bunnell, M.: Dynamic ambient occlusion and indirect lighting. GPU Gems **2** (2005) 223–233
8. Pharr, M., Green, S.: Ambient occlusion. GPU Gems **1** (2004) 279–292
9. Sattler, M., Sarlette, R., Zachmann, G., Klein, R.: Hardware-accelerated ambient occlusion computation. Vision, Modeling, and Visualization 2004 (2004) 331–338
10. Àlex Méndez-Feliu, Sbert, M., Català, J., Nicolau Sunyer, S.F.: Real-Time Obscurances with Color Bleeding (GPU Obscurances with Depth Peeling). ShaderX 4 (2006)
11. Shanmugam, P., Arikan, O.: Hardware accelerated ambient occlusion techniques on GPUs. Proceedings of the 2007 symposium on Interactive 3D graphics and games (2007) 73–80
12. Mittring, M.: Finding next gen: CryEngine 2. International Conference on Computer Graphics and Interactive Techniques (2007) 97–121
13. Rost, R.: OpenGL Shading Language. Addison-Wesley (2006)
14. Gotsman, C., Gu, X., Sheffer, A.: Fundamentals of spherical parameterization for 3 D meshes. ACM Transactions on Graphics **22** (2003) 358
15. Floater, M., Hormann, K.: Surface Parameterization: a Tutorial and Survey. Advances In Multiresolution For Geometric Modelling (2005)
16. Lévy, B., Petitjean, S., Ray, N., Maillot, J.: Least squares conformal maps for automatic texture atlas generation. ACM Transactions on Graphics (TOG) **21** (2002) 362–371
17. Devroye, L.: Non-uniform random variate generation. Springer-Verlag New York (1986)
18. St-Laurent, S.: The Complete Effect and Hlsl Guide. Paradoxal Press (2005)
19. Fernando, R., Kilgard, M.: The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA (2003)
20. Sheffer, A., Hart, J.: Seamster: inconspicuous low-distortion texture seam layout. Visualization, 2002. VIS 2002. IEEE (2002) 291–298
21. Gonzalez, R., Woods, R.: Digital image processing (3rd Edition). Prentice Hall (2007)