Deep Screen Space

Oliver Nalbach[1] Tobias Ritschel[1,2] Hans-Peter Seidel[1]
MPI Informatik[1] Saarland University / MMCI[2]

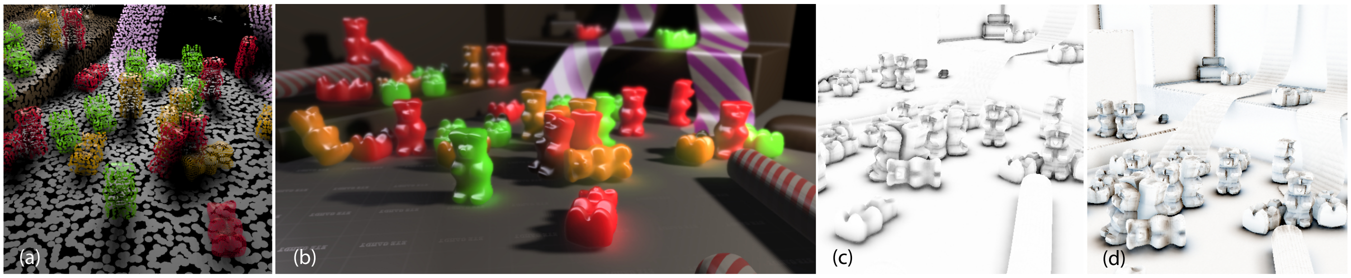


Figure 1: *Our approach replaces the framebuffer by a collection of view-dependent surfels produced on-the-fly* (a) *to compute shading such as subsurface scattering* (b, 1024×512, 27 ms)*, ambient occlusion* (c, 512×512, 22 ms) *or directional occlusion* (d, 512×512, 24 ms).

Abstract

Computing shading such as ambient occlusion (AO), subsurface scattering (SSS) or indirect light (GI) in screen space has recently received a lot of attention. While being efficient to compute, screen space methods have several key limitations such as occlusions, culling, under-sampling of oblique geometry and locality of the transport. In this work we propose a *deep screen space* to overcome all these problems while retaining computational efficiency. Instead of projecting, culling, shading, rasterizing and resolving occlusions of primitives using a z-buffer, we adaptively tessellate them into surfels proportional to the primitive's projected size, which are optionally shaded and stored on-GPU as an unstructured surfel cloud. Objects closer to the camera receive more details, like in classic framebuffers, but are not affected by occlusions or viewing angle. This surfel cloud can then be used to compute shading. Instead of gathering, we propose to use splatting to a multi-resolution interleaved framebuffer. This allows to exchange detailed shading between pixels close to a surfel and approximate shading between pixels distant to a surfel.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, Shading, Shadowing, and Texture;

Keywords: Interactive global illumination: Level-of-detail, Surfels: Splatting, Graphics hardware

1 Introduction

Computing convincing indirect light in large and dynamic scenes is still an elusive goal. Different effects, such as ambient occlusion, subsurface scattering or indirect lighting in screen space have recently received a lot of attention. (We subsume global illumination, subsurface scattering and ambient occlusion as "shading" here.) While being efficient to compute, screen space methods have several key limitations. Occluded pixels cannot send or receive shading. Consequently, shading appears and disappears, leading to the impression that it is not part of the *scene*, but merely a part of the *medium* (the "shower door" effect [Meier 1996]). Culling is the most extreme case: Here primitives behind other primitives or outside the frustum are ignored completely. More subtly, oblique primitives only receive a low number of pixels and hence their contribution is underestimated. Finally, screen space is limited to shading between nearby pixels as other exchange would require excessively large filters.

In this work, we propose a novel method to overcome all these problems while retaining computational efficiency. Instead of projecting, culling, shading, rasterizing and resolving occlusions of primitives using a z-buffer, we adaptively tessellate them into surfels proportional to the primitive's projected size, which are optionally shaded and stored on-GPU as an unstructured surfel cloud. Thanks to the tessellation, geometry closer to the camera receives more details, like in classic framebuffers. However it is not affected by occlusion or undersampling. This surfel cloud can then be used to compute shading.

Instead of gathering, we propose to use a splatting to a multi-resolution, interleaved framebuffer. This allows to exchange detailed shading between pixels close to a surfel and approximate shading between pixels distant to a surfel without the need of building a hierarchical representation. All of those steps fit the fine-grained parallelism of modern GPUs without the need of a stack or any per-thread state as required in many other hierarchical approaches, such as when tracing rays in a bounding volume hierarchy [Aila and Laine 2009] or enumerating a light cut [Walter et al. 2005].

Our approach requires no pre-computation, allowing for the fully dynamic scenes that are possible using screen space methods. A final important quality our approach shares with screen space methods is its output sensitivity and the fact that it strictly adapts the computation to the current view. While a screen space method achieves this by first quickly finding important pixels through rasterization, we use fast tessellation hardware. All these parallels motivate us to term our work *deep screen space*.

2 Previous work

Screen space shading was first used for ambient occlusion [Mittring 2007; Shanmugam and Arikan 2007; Bavoil et al. 2008] and later

I3D '14, March 14 - 16 2014, San Francisco, CA, USA

http://dx.doi.org/10.1145/2556700.2556708

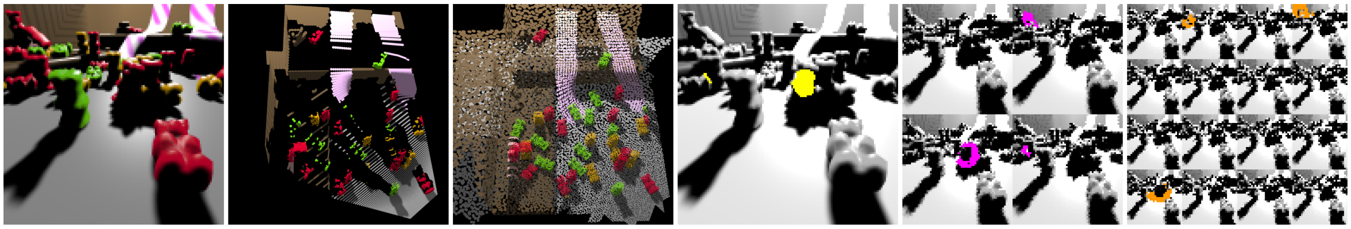


Figure 2: *For shading an image* (a) *a classic framebuffer* (b) *has incomplete information while our deep framebuffer* (c) *is complete. To compute its contribution to the original framebuffer, its surfels are splat in multiple resolutions* (d–f)*. For pixels that are near to a surfel position in image space, the full resolution is used* (d)*, for more distant pixels, a hierarchy of random subsamples of the framebuffer* (e, f) *is employed. The subsampling results in same-sized splats to cover increasingly large image regions without much overdraw.*

extended to subsurface scattering [Jimenez et al. 2009] or diffuse bounces [Ritschel et al. 2009]. In the following, we will focus on attempts to overcome its limitations.

First, *multi-resolution* computation can improve screen space shading, demonstrated for the case of diffuse bounces by Nichols and Wyman [2009]. Here, a hierarchy is built in screen space as a regular quad tree. The contribution of VPLs [Keller 1997] to the screen is computed by splatting at different increasingly coarse resolutions. Instead of reducing the resolution of each layer and then splatting one VPL to one layer, we splat to multiple randomly subsampled images. In combination with feature-aware blurring, spatially small details also receive a shading contribution, just with fewer samples. An example is a thin blade of grass: When reducing the resolution, the blade at some point completely disappears. In our approach, only the number of subsampled and randomized framebuffers containing pixels of the thin object decreases (Fig. 9).

Second, *occlusion* is an issue for screen space shading. The restriction to the first visible surface was addressed using multiple views [Ritschel et al. 2009] and shadow maps [Vardis et al. 2013], but could also be solved using layered depth images (LDIs) [Shade et al. 1998]. However, occlusion is not the only problem of screen space shading: Under-sampling of geometry under grazing view angles is never resolved, even when using LDIs, while there is no good reason to not consider occluders or emitters that are seen under a grazing angle.

Next, *sweeps* along a discrete number of directions [Timonen 2013] were proposed leading to significantly improved performance but also to banding artifacts. In our approach, we blur to reduce high-frequency noise which is unbiased with respect to the lighting from the surfels, whereas banding is hard to remove later.

The gathering of common screen space image filtering is replaced by *splatting* in the work of Sloan et al. [2007] and McGuire et al. [2010]. The first uses pre-computed sphere proxy geometry with limited geometric detail, the latter generates splatting primitives from triangles. In contrast, we generate our splatting primitives on-the-fly to capture important details that vary across a detail or proxy (shadow edges, textures) and combine their contribution in a novel multi-resolution scheme to reduce fill-rate.

Besides screen space, *hierarchies of surfels* are popular to compute GI [Bunnell 2005; Christensen 2008] or scattering [Jensen and Buhler 2002]. However, this hierarchy needs to be built and traversed, which is both not the ideal solution for a GPU and applications are limited to medium-sized scenes undergoing minor deformations in practice. Further, the discretization into a surfel cloud is usually done once and limits the geometric detail. We avoid a pre-defined discretization and traversing or building hierarchies altogether. Our approach produces new surfels even when zooming in and is only limited by the (procedural) geometric detail the scene contains.

Screen space shading bears similarities to *Instant Radiosity* [Keller 1997], where discrete points (VPLs) replaced the finite element polygons of radiosity. Reflective shadow maps [Dachsbacher and Stamminger 2005] are a particularly efficient way to produce such points for indirect illumination from a single light. The idea is to rasterize the scene from the view of the light and use the visible parts as emitters to splat illumination. Such approaches work well for a single primary light, but no obvious way exists to extend it to general emitters or occluders e. g., in the presence of environment maps or to ambient occlusion occluders.

Finally, our approach relates to the classic idea of *micro-polygons* in REYES [Cook et al. 1987] that subdivides primitives to become pixel-sized triangles and shades their vertices. Replacing polygons by points was also used to render large [Wand et al. 2001] or procedural geometry [Stamminger and Drettakis 2001] efficiently. In all cases, points create the final image when they are shaded and used to resolve visibility. In this paper, we compute shading contribution from the surfels onto the framebuffer pixels, but the surfels never become visible in the final image themselves.

3 Deep screen space

We will now explain our approach which is independent of a specific shading effect such as ambient occlusion, subsurface scattering or indirect light. In our pipeline, we first compute a view-dependent point-based representation from our scene primitives on-the-fly (Sec. 3.1) which is then splatted onto a multi-resolution deferred shading buffer (Sec. 3.2) which is finally combined into a single image (Sec. 3.3).

3.1 Tessellating the scene into a point cloud

The first step comprises turning the input triangle mesh of the scene into a surfel cloud [Pfister et al. 2000]. Each surfel can be seen as an oriented disk that is defined by its position, normal and radius and may also be equipped with additional attributes like reflectance, depending on the shading whose computation is desired. The surfels in the cloud are supposed to roughly approximate the original scene geometry but using more uniform primitives which significantly simplifies the shading computations.

We use hardware tessellation to adaptively generate surfels from triangles efficiently [Stamminger and Drettakis 2001; Bark et al. 2013]. Our goal is to achieve an approximately equal size in screen space for all surfels, independent from their distance in world space and orientation. This explicitly includes surfels that are very oblique or even back-facing, which are never present in a common framebuffer. As the target (world space) radius for the surfels stemming from one triangle, we choose $r^* = s \cdot \tan(\alpha/2) \cdot (d + d_{\text{near}})$, where s scales the surfel size (relative to the size of the screen), α is the vertical opening angle of our camera, d the distance of the triangle's center to the camera's near clipping plane and d_{near} the distance of the near clipping plane, making surfels become larger again directly behind the near plane (Fig. 3).

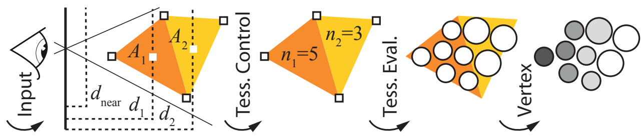


Figure 3: *Our surfel tessellation pipeline for two triangles* (see text). *Vertices are depicted as rectangles, surfels as disks.*

Tessellation in OpenGL consists of a control and an evaluation stage (Fig. 3). First, the *tessellation control shader* (TCS) computes into how many output primitives an input primitive is to be tessellated. Second, the *tessellation evaluation shader* (TES) computes each output primitive. We are using tessellation in "point mode" where, in difference from the common use to turn a triangle into many triangles forming a smooth surface, triangles are converted into points. OpenGL's tessellation method will produce

$$v(i) = \begin{cases} {}^{3}\!/_{4}\, i^2 + {}^{3}\!/_{2}\, i + 1 & i \text{ is even} \\ 3 \lceil {}^{i}\!/_{2} \rceil^2 & i \text{ is odd} \end{cases}$$

vertices for a triangle if all tessellation levels are set to i in the TCS. Using the formula for the even case, we approximate the necessary tessellation level for a triangle with area A by solving $v(i) \cdot \pi r^{*2} = A$ for possible cases (i. e., where $A \geq 3\pi r^{*2}$ and $i > 0$) and taking the ceil which yields

$$i = \left\lceil \frac{\sqrt{12\pi A - 3\pi^2 r^{*2}}}{3\pi r^*} - 1 \right\rceil .$$

After determining the tessellation level, we adjust the actual radius r such that the area of all surfels sums to A. The evaluation in the TES simply computes averaged positions, normals etc. using a barycentric coordinate which is made available by the tessellator unit. Optionally, shading is computed and stored for every surfel as well.

Discussion Using hardware tessellation to generate a point cloud has a few shortcomings. First, surfels at edges will protrude from the shape of the original triangle by r. In particular we get overlapping surfels from adjacent triangles. To avoid this, we simply move the three vertices of the original triangle based on which the new vertex positions are computed towards the triangle's center of mass by r (done in the TCS). Second, the tessellation creates regular patterns of surfels which might become visible in one way or the other when computing the effect. We therefore jitter each surfel on the triangle plane by a maximum distance of r. Third, the TES will create at least three surfels for each original triangle, which might be too small, especially for distant objects. Here, we have to assume that a general LOD solution already reduced the number of triangles to a reasonable pixel-to-vertex ratio [Luebke 2003]. Otherwise, the two negative consequences will be reduced effect distance (due to the too-small surfels) and possibly higher computation cost (due to the increased number of surfels). An example is given in Fig. 4.

3.2 Splatting the point cloud onto a framebuffer

We will now describe how to compute the particular shading contribution from the deep framebuffer to a common framebuffer. Different from the commonly used screen space methods which gather from nearby pixels, we use splatting to scatter the shading contribution from a surfel to multiple pixels. While complexity is the same (number of surfel-pixel interactions compared to the number of pixel-pixel interactions), this pattern appears to be a "step back", as gathering, in particular in regular stencils, is known as the preferred pattern on modern GPUs. However, no obvious way exists to enumerate neighbors of a 2D pixel in a 3D surfel cloud without a costly hierarchy, in particular proportional to world space distance, such as achieved by our multi-resolution scattering explained next.

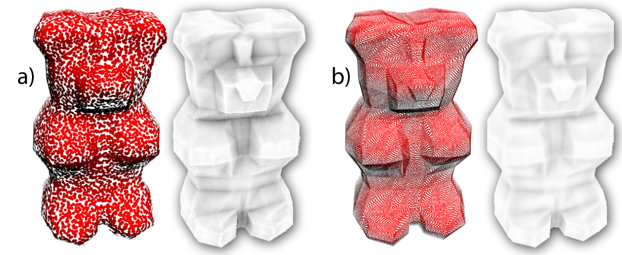


Figure 4: *The problem with too many triangles per pixel. The desired surfel size* (a, left) *is not achieved by our tessellation method if the initial mesh already is too finely tessellated* (b, left). *As a consequence, the effect distance will be less despite the exact same settings being used* (a/b,right). *While this poses an inconsistency, the result still looks similar, just lighter in this case.*

To compute the shading, we employ a special framebuffer layout based on interleaved sampling [Segovia et al. 2006], but using multiple image levels with multiple interleaved patterns at the same time. While the system of Segovia et al. [2006] is not competitive or required on current GPUs anymore when gathering, it is very useful for splatting, in particular in combination with our extension to multiple image levels.

We splat into an array texture with $l_{\max}$ layers where each layer corresponds to a different image resolution level. On image level $l \geq 0$ we partition the pixels of the image into $2^l \times 2^l$ smaller "sub-buffers": For this, we take $2^l \times 2^l$ neighborhoods of pixels and randomly assign one pixel to each sub-buffer, taking the same relative position in the sub-buffer that the neighborhood had in the original image. Fig. 2 d–f show a possible layout for image levels 0 to 2. Different from MIP maps, the total number of pixels on each image level is identical. The original pixels are merely distributed among more and more sub-images as the level number increases. If we now draw a quad of size $n \times n$ on image level l, we will invoke the fragment shader for a random subset of (at least) n^2 pixels in a $2^l n \times 2^l n$ neighborhood of the original framebuffer. This allows us to subsample the effect, balancing precision and effect distance in different ways at the same computation costs, depending on the level we choose for splatting. By retaining the whole set of pixels on all levels, every detail of the scene still has the chance to receive shading from each surfel, only the probability decreases.

One main goal of our approach is to sample the shading contribution of a surfel to the framebuffer finely for nearby and coarsely for far-away geometry. To this end, we compute the shading contribution to disjunct, increasingly large *shells* around the surfel's center (in world space) on increasing levels of our framebuffer. On each level, each surfel splats a shell into one of the sub-buffers. A schematic of this is shown in Fig. 5 while Fig. 2-d–f shows an actual rendering of the first three framebuffer levels including several shells which have been splatted.

On the implementation side, we invoke a geometry shader $l_{\max}$ times for each surfel, emitting a same-sized point primitive at the

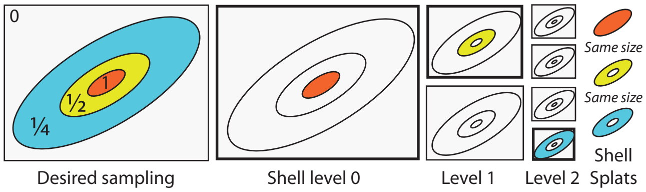


Figure 5: *To splat with radially decreasing density, we splat multiple shells - each to a random element of a set of images which have subsampled with decreasing density. E.g. the innermost shell (shown in red) affects all pixels within a certain range, while the shell on level 1 (shown in yellow), which has the same size in the framebuffer, may cover pixels twice as far away but can only have an effect on half of them, namely the ones which have been assigned to the chosen sub-buffer. While each splat only covers a small subset of all pixels, using a sufficient number of levels, the whole screen can be covered by the sum of all splats.*

position of the surfel's center relative to the chosen sub-buffer. Note that, although we consider shells in world space, we can bound them using small quad-shaped point primitives. The sub-buffer to use is chosen in a round-robin manner, based on the index of the surfel. On image level l, the surfel's radius used in shading computations is to be scaled by a factor of 2^l (so the surfel's area is multiplied by $2^l \cdot 2^l$) to compensate for the fact that each pixel can only be affected by $1/(2^l \cdot 2^l)$ of all surfels. In other words, the enlarged surfel is used as approximation for $2^l \cdot 2^l$ original surfels. As there is no clipping between individual sub-buffers, we later need to check for every fragment whether it really belongs to its given sub-buffer.

The size of the splat is determined by a function `getMaxDist` taking a surfel-struct (encapsulating all the information about a surfel) and a threshold ϵ as arguments. Semantically, it should return the world-space distance from the surfel's center in which the effect of the surfel will be smaller than ϵ. This is similar to the bounds used in light cuts [Walter et al. 2005], classic radiosity oracles or the proxy sphere scaling in Sloan et al. [2007]. The screen space size for the point primitive is then computed based on this distance. The definition of the function is effect-dependent and may, for example, take a stored amount of irradiance into account. Sec. 4 discusses the effect-specific implementations of this function.

The only requirement for our approach to work is that the function is linear in the surfel's radius which is naturally the case for the effects we demonstrate. Note that the size of the surfel splats in screen space then will be the same for all image levels because with each level, the surfel's radius, and therefore also the radius where its effect is larger than the threshold, increases by a factor of 2^l while the extent of the sub-buffers shrinks accordingly (Fig. 5, right column). In particular, this means that the maximal distance of the effect increases by a factor of two in each image level.

The fragment shader is passed the surfel information as well as the inner and outer radius of the shell associated with each splat. In the fragment shader, we first look up the world space position and normal associated with the fragment's position for the respective image level. To accelerate this lookup, we shuffle the position and normal buffers in advance according to the chosen framebuffer layout, so reads avoid indirection and are spatially coherent [Segovia et al. 2006]. If the world space position is not inside the shell given by the surfel center and the inner and outer radius, we discard the fragment. Otherwise, the effect is computed by a function `computeEffect` that takes the surfel and the fragment's position and normal. (Details are given in Sec. 4.) We enable a suitable blending to sum up the effect of all splats on each framebuffer pixel. To avoid unnecessary computations, we can perform view frustum culling of a sphere around the surfel's center with the outer radius of the shell in each geometry shader invocation.

3.3 Reconstructing the final image

After splatting, we have an array texture where different image levels are still partitioned into sub-buffer grids. "Unshuffling" them will typically leave us with noisy layers because of subsampling. We blur each image level with a separated blur in x- and y-direction [Bavoil et al. 2008]. We use weights similar to the ones for bilateral upsampling [Sloan et al. 2007]. Summing up the layers produces the final image.

4 Applications

Next, we discuss some shading operations enabled by our approach and compare them to reference solutions (Fig. 7).

Ambient occlusion The ambient occlusion contribution of a surfel to a pixel (in the `computeEffect` function) as well as its influence radius, returned by `getMaxDist`, are computed using the point-to-disk form factor [Wallace et al. 1989]. We achieve quality similar to the ray-tracing reference but at speed similar to HBAO [Bavoil et al. 2008] (Fig. 7-a). The "shower door" effect [Meier 1996] is present in screen space ambient occlusion and all following shading effects, but not when using our approach, as seen when the camera moves as in the accompanying video. Remaining artifacts are due to overestimation of occlusions, discretization into disks and clamping of the distance term.

Directional occlusion Directional occlusion [Ritschel et al. 2009] is an improved occlusion computation to be combined with image-based lighting [Heidrich and Seidel 1999]. Again, the point-to-disk form factor is used for `computeEffect` and `getMaxDist`. Instead of just accumulating the result, a lookup into the pre-convolved environment map is made and the occluded value is subtracted. This results in colored shadows and directional occlusion effects (Fig. 7-b). Here, our approach can provide better quality at higher speed. The improvement in quality is due to the absence of occlusion problems. The increased performance is observed because the scene requires a large gathering radius that would result in a large image-space filter.

One-bounce indirect illumination To simulate one bounce of indirect light, the surfels are additionally shaded in the tessellation evaluation stage. Again, point-to-disk form factors are used, but we can now include the shading infomation: A surfel with radius r and radiosity $B \in \mathbb{R}^3$ uses $r/\epsilon \max(B_\mathrm{r}, B_\mathrm{g}, B_\mathrm{b})$ in `getMaxDist`. Note, that this discards surfels which are in shadow automatically. We can achieve results that improve over SSGI [Ritschel et al. 2009] in terms of quality, in particular regarding the possible effect distance, and speed as seen in Fig. 7-c for the reason explained in the previous paragraph. We also demonstrate specular SSGI (Fig. 7-d), which would be very difficult to resolve using gathering. Our approach, as well as all screen space approaches we are aware of, does not support indirect visibility.

Multiple scattering in translucent materials For subsurface scattering, we first compute irradiance along with each sample. Computation of the effect then is done as suggested by Jensen and Buhler [2002] but replacing the gathering using a hierarchy by our splatting technique. For simplicity, we chose `getMaxDist` to be the same as for diffuse bounces as after fixing the scattering parameters, the maximal distance will only vary depending on the amount of irradiance and the surfel's radius. We compare our results to screen space scattering [Jimenez et al. 2009] in terms of performance and a solution akin to Jensen and Buhler [2002] where we distributed 2 M points on the objects and evaluated the dipole in respect to each, for a comparison in terms of quality. We observe quality similar to Jensen and Buhler which takes several seconds to compute. Our speed is in the order of milliseconds, as for Jimenez et al. which cannot capture global details e. g., scattering from light not visible in the framebuffer (Fig. 7-e).

5 Discussion

Parameters Running time and quality depend on the number of pixels, the number of surfels, the splat size and the depth complexity of the framebuffer. The number of splats depends on the chosen surfel size and scene complexity: scenes with high depth complexity are more costly but at least the cost for splatting one depth layer is limited by the constant screen space size of the surfels. To capture small-scale effects, e. g., occlusion of a floor on the leg of a chair, we need to choose a sufficiently small surfel size. The size of the splats depends on the value we choose for our effect-threshold ϵ (as well as on surfel properties for some effects). In combination with a suitable number of image levels, we need to choose ϵ small enough for the effect to cover a sufficient distance in screen space. A larger value for ϵ in conjunction with a larger number of image levels can yield the same maximal distance, however the effect will become less-detailed because of coarser sampling. Our method has three parameters that on the one hand need to be tuned, but on the other hand offer fine control over quality vs. speed (Fig. 8).

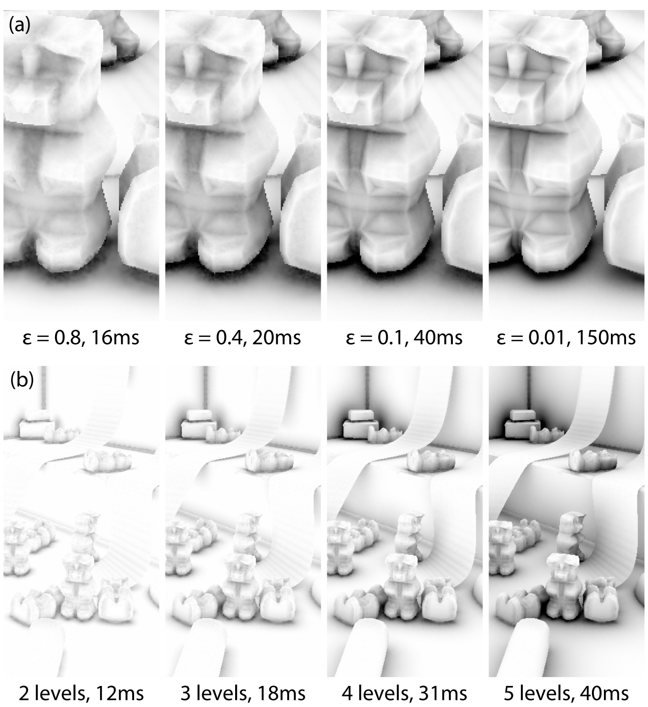


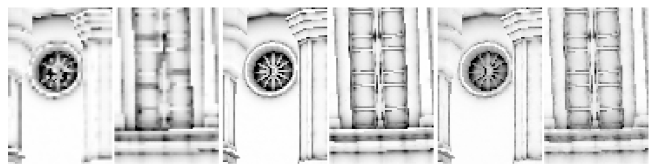
Figure 8: *Decreasing ϵ reduces blur and noise while computation time increases* (a). *Increasing the number of levels and keeping ϵ constant increases computation time and effect distance* (b).

Performance Tbl. 1 gives a performance breakdown for Fig. 7 (512×512, 50 k tris).

Table 1: *Computation time for different stages and shading effects.*

Stage	AO	DO	GI	SSS
Tessellation	1.6 ms	1.6 ms	1.6 ms	1.6 ms
Shuffling	2 ms	2 ms	3 ms	2 ms
Splatting	14 ms	16 ms	44.7 ms	14.6 ms
Unshuffling	1 ms	1 ms	1.2 ms	1 ms
Blurring	3 ms	3 ms	5 ms	2.5 ms
Summing	0.4 ms	0.4 ms	0.5 ms	0.3 ms
Total	22 ms	24 ms	56 ms	22 ms

Algorithmic alternatives Upsampling and blurring are common operations in interactive global illumination. Either the image is upsampled from a low resolution, where splatting is efficient, but details might be lost, or splatting is used, resulting in high quality but suffering from reduced performance due to overdraw. We compare our approach to those alternatives in Fig. 9.



47ms (a) Upsampled
165ms (b) Full resolution
42ms (c) Ours

Figure 9: *When simply computing the effect in half the resolution and upsampling it, high frequencies are observed but small details, such as this window from the Sibenik cathedral mesh, cannot be recovered* (a). *Splatting in full resolution* (b) *looks crisp but is slower. Our approach* (c) *balances the two and is fastest.*

Limitations Our results share the problem of over-occlusion with similar approaches [Bunnell 2005; Sloan et al. 2007] which cannot be removed in an iteration between surfels, as we cannot efficiently compute interactions between the huge number of surfels themselves. Therefore, it is unclear how to add multiple bounces.

6 Conclusions and Further Work

We proposed a general technique for indirect shading of dynamic deforming geometry that overcomes most limitations of screen space methods, while providing similar efficiency. Our main limitations are the approximate surfel discretization, the lack of a visibility operator between surfels and receivers as well as the limitation to one bounce. In future work, we would like to use our deep screen space to extend screen space depth of field and motion blur.

Acknowledgements

The Sibenik Cathedral mesh used for Fig. 9 was originally created by M. Dabrovic. The elephant gallop mesh data used in Fig. 6 and the supplemental video was made available by Robert Sumner and Jovan Popovic from the Computer Graphics Group at MIT.

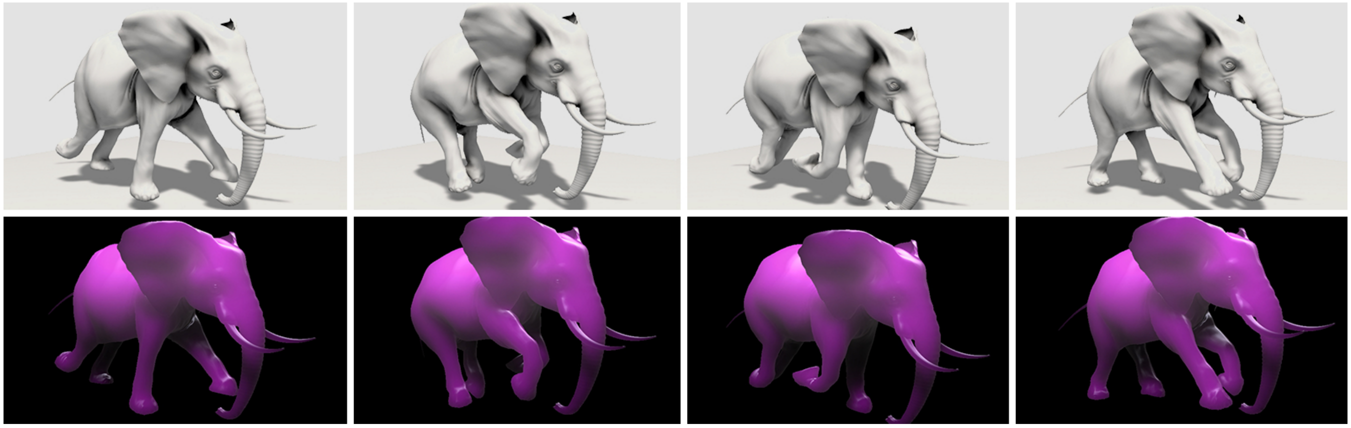


Figure 6: *Frames of an animation (*800 × 600*) showing temporally coherent ambient occlusion* (Top, 27 ms) *and subsurface scattering* (Bottom, 22 ms).

References

AILA, T., AND LAINE, S. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proc. HPG*, 145–149.

BARK, T., BITTNER, J., AND HAVRAN, V. 2013. Temporally coherent adaptive sampling for imperfect shadow maps. *Computer Graphics Forum (Proc. EGSR) 32*, 4, 87–96.

BAVOIL, L., SAINZ, M., AND DIMITROV, R. 2008. Image-space horizon-based ambient occlusion. In *SIGGRAPH 2008 talks*.

BUNNELL, M. 2005. Dynamic ambient occlusion and indirect lighting. In *GPU Gems 2*, M. Pharr, Ed. Addison-Wesley, 223–33.

CHRISTENSEN, P. H., 2008. Point-based approximate color bleeding.

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The reyes image rendering architecture. In *ACM SIGGRAPH Computer Graphics*, vol. 21, 95–102.

DACHSBACHER, C., AND STAMMINGER, M. 2005. Reflective shadow maps. In *Proc. I3D*, 203–31.

HEIDRICH, W., AND SEIDEL, H.-P. 1999. Realistic, hardware-accelerated shading and lighting. In *Proc. SIGGRAPH*, 171–78.

JENSEN, H. W., AND BUHLER, J. 2002. A rapid hierarchical rendering technique for translucent materials. In *Proc. SIGGRAPH*, 576–81.

JIMENEZ, J., SUNDSTEDT, V., AND GUTIERREZ, D. 2009. Screen-space perceptual rendering of human skin. *ACM Trans. Appl. Percept. 6*, 4, 23:1–23:15.

KELLER, A. 1997. Instant radiosity. In *Proc. SIGGRAPH*, 49–56.

LUEBKE, D. P. 2003. *Level of Detail for 3D Graphpics*. Morgan Kaufmann Pub.

MCGUIRE, M. 2010. Ambient occlusion volumes. In *Proc. HPG*.

MEIER, B. J. 1996. Painterly rendering for animation. In *Proc. SIGGRAPH*, 477–484.

MITTRING, M. 2007. Finding next gen: CryEngine 2. In *SIGGRAPH courses*, 97–121.

NICHOLS, G., AND WYMAN, C. 2009. Multiresolution splatting for indirect illumination. In *Proc. I3D*, 83–90.

PFISTER, H., ZWICKER, M., VAN BAAR, J., AND GROSS, M. 2000. Surfels: Surface elements as rendering primitives. In *Proc. SIGGRAPH*, 335–342.

RITSCHEL, T., GROSCH, T., AND SEIDEL, H.-P. 2009. Approximating dynamic global illumination in image space. In *Proc. i3D*, 75–82.

SEGOVIA, B., IEHL, J. C., MITANCHEY, R., AND PÉROCHE, B. 2006. Non-interleaved deferred shading of interleaved sample patterns. In *Proc. Graphics Hardware*, 53–60.

SHADE, J., GORTLER, S., HE, L.-W., AND SZELISKI, R. 1998. Layered depth images. In *Proc. SIGGRAPH*, 231–42.

SHANMUGAM, P., AND ARIKAN, O. 2007. Hardware accelerated ambient occlusion techniques on gpus. In *Proc. I3D*, 73–80.

SLOAN, P.-P., GOVINDARAJU, N. K., NOWROUZEZAHRAI, D., AND SNYDER, J. 2007. Image-based proxy accumulation for real-time soft global illumination. In *Proc. Pacific Graph.*, 97–105.

STAMMINGER, M., AND DRETTAKIS, G. 2001. Interactive sampling and rendering for complex and procedural geometry. In *Proc. Rendering Techniques*. 151–62.

TIMONEN, V. 2013. Line-sweep ambient obscurance. In *Comp. Graph. Forum*, vol. 32, 97–105.

VARDIS, K., PAPAIOANNOU, G., AND GAITATZES, A. 2013. Multi-view ambient occlusion with importance sampling. In *Proc. I3D*, 111–18.

WALLACE, J. R., ELMQUIST, K. A., AND HAINES, E. A. 1989. A ray tracing algorithm for progressive radiosity. *SIGGRAPH Comput. Graph. 23*, 3, 315–24.

WALTER, B., FERNANDEZ, S., ARBREE, A., BALA, K., DONIKIAN, M., AND GREENBERG, D. P. 2005. Lightcuts: A scalable approach to illumination. In *ACM Trans. Graph. (Proc. SIGGRAPH)*, vol. 24, 1098–1107.

WAND, M., FISCHER, M., PETER, I., MEYER AUF DER HEIDE, F., AND STRASSER, W. 2001. The randomized z-buffer algorithm: Interactive rendering of highly complex scenes. In *Proc. SIGGRAPH*, 361–70.

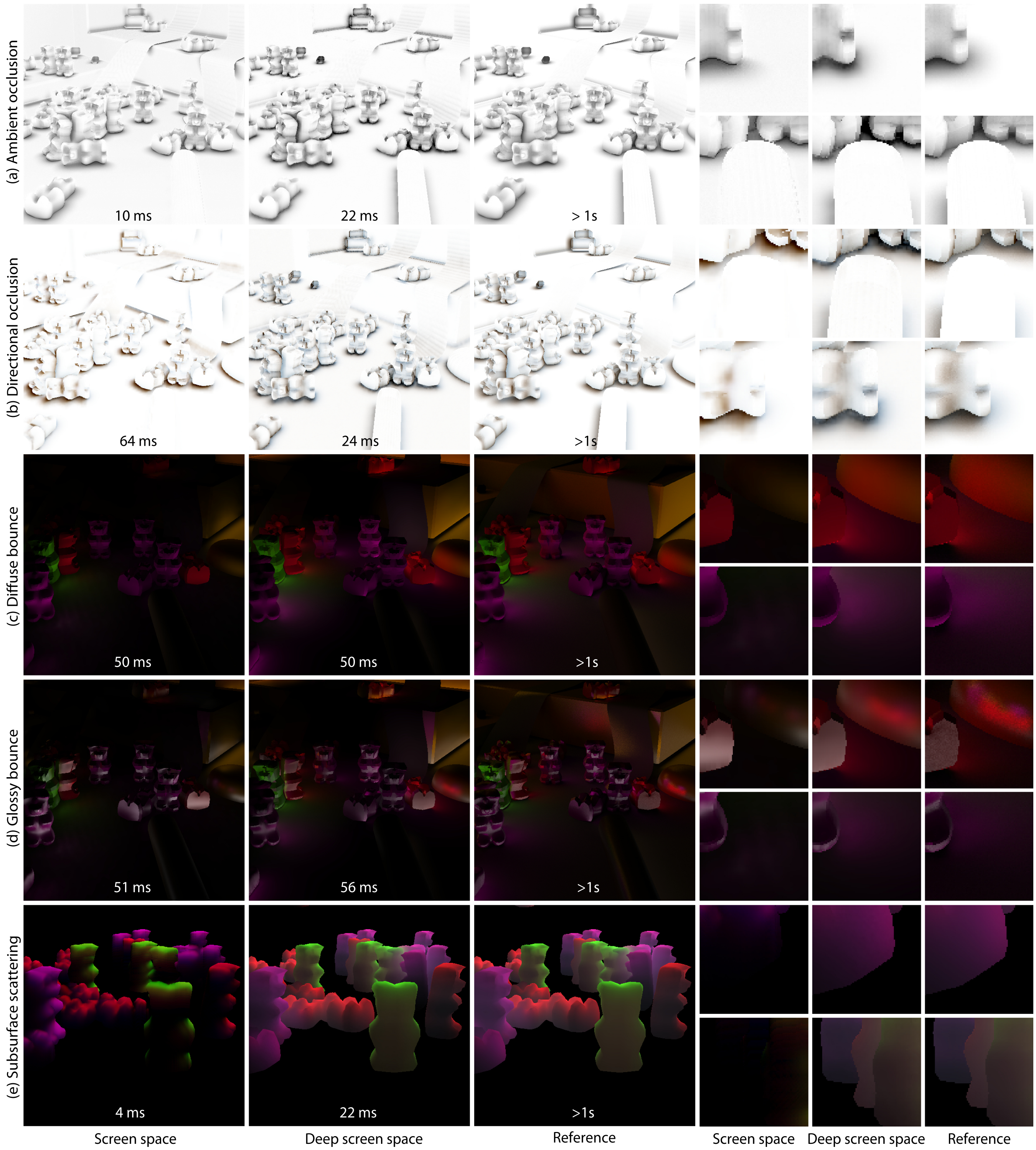


Figure 7: *Ambient occlusion, directional occlusion, GI and scattering for our test scene* (512×512, $50\,k$ *tris*)*. Please see the video for an animated version. The first column is a screen space reference (comparable in speed to ours); the second our result; the third is a reference (comparable in quality to ours). The rightmost columns show details in the same order. Ambient occlusion (first row) in screen space [Bavoil et al. 2008] looks plausible, but our approach is much more similar to a reference, producing shadows from triangles invisible in the framebuffer. Directional occlusion (second row) in screen space [Ritschel et al. 2009] lacks occlusion from objects not present in screen space while our approach reproduces it. Due to the larger filter size required for classic directional occlusion, our approach can even produce better quality at higher speed. Diffuse and specular bounces (third and fourth row) in screen space [Ritschel et al. 2009] are usually spatially limited or become slow. At the same speed, we can produce quality similar to the reference. Subsurface scattering is fast in screen space [Jimenez et al. 2009], but does not reproduce light that is not present in the framebuffer. Our solution is slower, but similar to a reference, computed using* $2\,M$ *irradiance samples by the method of Jensen and Buhler [2002], as rendering the scene using path tracing is prohibitive.*