Chapter 9

# Fast Approximations for Global Illumination on Dynamic Scenes

Alex Evans[14]
Bluespoon

## 9.1 Abstract

An innovative lighting algorithm is presented that allows scenes to be displayed with approximate global illumination including ambient occlusion and sky-light effects at real-time rates. The method is scalable for high polygonal scenes and requires a small amount of pre-computation. The presented technique can be successfully applied to dynamic and animated sequences, and displays a striking aesthetic style by reducing traditional constraints of physical correctness and a standard lighting model.

## 9.2 Introduction

The introduction of pixel and fragment level programmable commodity GPU hardware in 2002 led to a significant shift in the way in which real-time graphics algorithms were researched and implemented. No longer was it a case of exploiting a few simple fixed function operations to achieve the desired rendered image – instead programmers could explore a wealth of algorithms, along with numerous variations and tweaks.

One specific class of algorithms of particular interest in the games industry, are those real-time algorithms which take as their target an art director's 'vision', rather than a particular subset of the physics of light in the real world.

These algorithms opened up the 'field of play' for graphic engine programmers, and greatly increased the opportunity for each engine to differentiate itself visually in a crowded and (in the case of games), fiercely competitive market. [Evans04]

Many problems in computer graphics are not yet practical to compute in real-time on commodity hardware, at least in the general case. Instead, techniques that can balance

[14] alex@bluespoon.com

the computational load between run-time and pre-processing costs allow different trade-offs that are applicable in different rendering systems (for example, pre-computed radiance transfer (PRT) techniques, which allow complex light transport to be computed in a static scene, then cached in a variety of ways which allow novel views and/or re-lighting of a scene to be rendered interactively).

In these course notes, we will describe one such approximation algorithm. There are many points at which decisions were made between a number of different, potentially useful algorithmic options. It is this process of iteration and choice, made possible by GPU programmability, more than the particular end result, that is the emphasis of the following discussion.

## 9.3 Algorithm outline

The goal of the algorithm is to allow real-time rendering of dynamic scenes from a movable viewpoint, lit under some aesthetically pleasing approximation to 'sky lighting'. An example of a sky-lit scene is shown in Figure 1. The key requirements set by our fictional art director are:

- Dark 'contact' shadows where objects come into contact with each other
- Darker regions inside the creases and valleys of complex organic shapes – often called 'ambient occlusion'.
- Ability to handle multiple objects each of which can move both rigidly and preferably, freely deform.

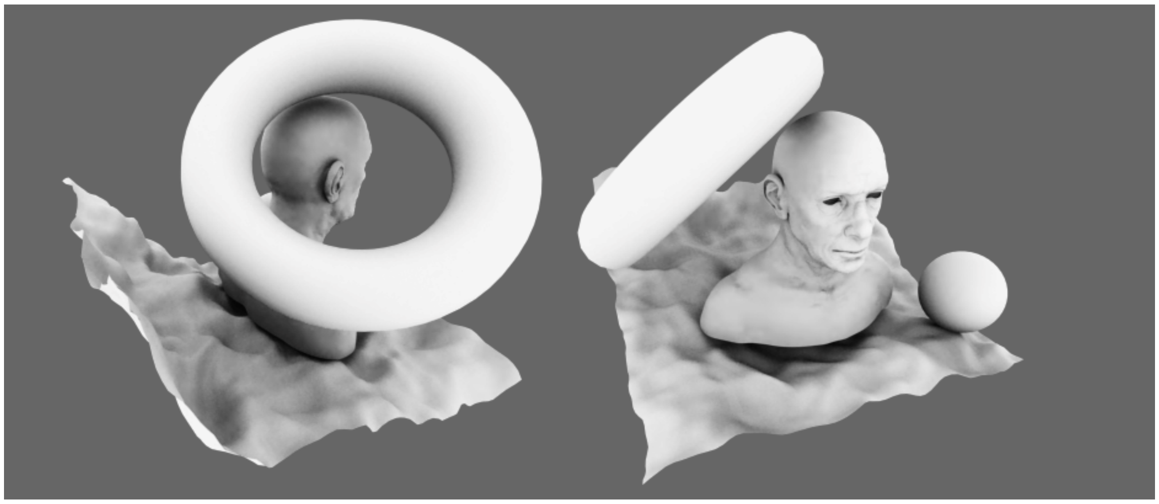

***Figure 1****. A detailed head model rendered using Autodesk's 3D Studio Max default renderer with a single skylight. (Model by Sebastian Schoellhammer, www.sebster.org)*

These kinds of effects are a key visual indicator in giving the impression of physical objects are in contact with each other, and have been widely investigated, dating back to the earliest days of graphics research. Path tracing [Kajiya86] gives a simple but very slow way of understanding and evaluating sky lighting: at each point P to be rendered, the fraction of the sky dome visible at that point is estimated by shooting a large number of rays towards the sky (considered to be an evenly light emitting hemisphere centered at the origin of the scene, of infinite radius). These rays bounce off any surfaces that they come across, until they are eventually considered to be absorbed or reach the sky dome itself. (Figure 2)

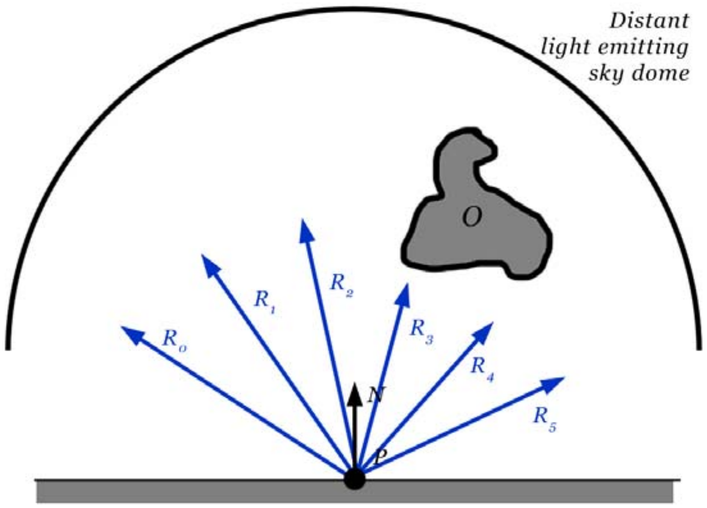

***Figure 2**. Path tracing a point P – by tracing rays from P towards a large number of points on the distant sky hemisphere, and measuring the fraction which make it to the sky, we can arrive at an estimate of the surface radiance of the point P as a result of the sky lighting. In the case illustrated, rays $R_3$ and $R_4$ are blocked by object O.*

By summing the results of a suitably large number of random rays, this Monte Carlo approach to solving the diffuse part of Kajiya's rendering equation eventually converges to an estimate of the surface radiance of the point $P$ – and thus, when applied to all points visible to the eye, a nice-looking sky-lit scene. Ignoring ray bounces, this process amounts to computing an approximation to a simple integral giving the visibility of the sky in the hemisphere above $P$:

$$I_P = \int_{\Omega_P} V(\omega) B(v,\omega) d\omega$$

where $I_P$ is the surface radiance of point $P$, $\Omega_P$ is the hemisphere above the surface at $P$, $B(v, \omega)$ is the BRDF of the surface being lit (including the diffuse cosine term, $max(N{\cdot}L,0)$ ), $v$ is the direction towards the viewer from $P$, and $V(\omega)$ is a function defined such that $V(\omega)=1$ when rays in that direction reach the sky, and 0 when they do not.

Normally, $V$ is the expensive term to compute, and thus the term that is normally chosen for various kinds of approximation by real-time rendering algorithms.

Many techniques exist which try to make the sky lighting problem more tractable by making various trade-offs. As such, real-time algorithms are often better characterized by the cases they can't handle than by the cases that they can. For example, many diffuse global illumination solutions rely on pre-computed representations of the geometry of the scene and the light flow around it. These include various forms of radiosity, pre-computed radiance transfer (PRT), irradiance caching, and table based approximations based on simplified occluders, such as spherical caps [Oat06], [Kontikanen05].

However, to illustrate the breadth of approach that programmable GPUs have given us, we are going to outline an unusual variant that has the following properties:

- Requires very little pre-computation and hence works well in dynamic scenes
- Has enormously large deviation from the accurate solution, but still looks aesthetically pleasing
- Allows limited bounce light effects without computational penalties

Note that we have to specify the following constraint for our algorithm: it will only support scenes that can be encompassed in a small volume. This is a direct result of the fact that the approach is based on the idea of signed distance fields (SDFs) which will be stored in a volumetric texture on the GPU. The above properties and constraints will provide us with a GPU-friendly data structure, whose computation is expensive but highly parallelizable, which encodes enough of the geometric layout of the scene that we can render approximate sky-lighting efficiently.

## 9.4 Signed Distance Fields

In a given scene consisting of solid bodies, a signed distance field is a simple scalar function S(P) defined at every point $P$ in a (2D or 3D) space, such that

$S(P) = 0$ when it is on the surface of a body
$S(P) > 0$ when it is inside any body
$S(P) < 0$ when it is outside all bodies

Its magnitude is the minimum distance from that point to the surface of any body. A simple example of a 2D SDF is visualized in Figure 3.

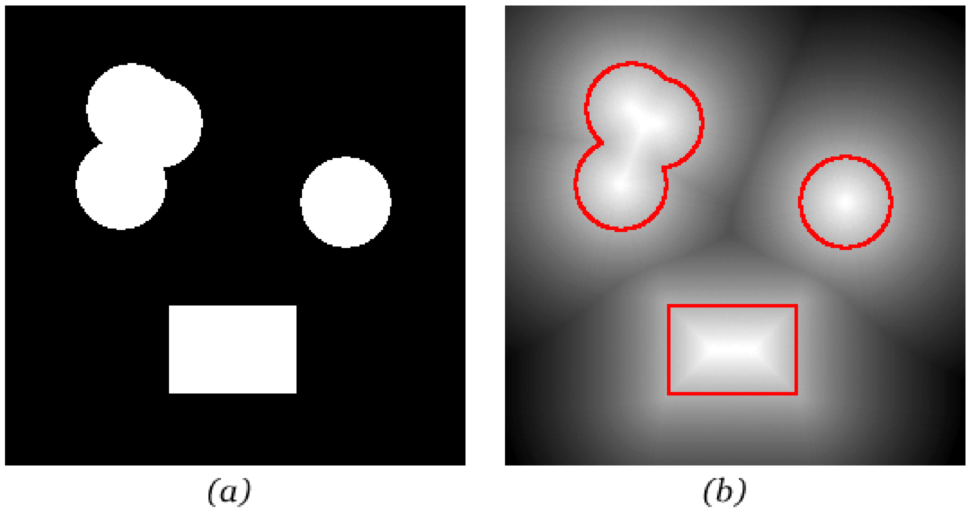

***Figure 3.*** *a) shows a binary image representing a 2D scene, with points covered by an object marked white. b) Shows the SDF computed from (a) using 2D Euclidian distances, with -1 to 1 mapped to a grayscale gradient from black (-1) to white (1). The boundary of the objects (where SDF=0) is marked in red for clarity.*

Computing the SDF of an arbitrary polygonal 2D or 3D scene on a regularly sampled grid is a computationally expensive process, often approximated by repeated application of small (3x3x3) window-size non-linear filters to the entire volume [Danielson00, Grevera04]. We will deal with efficient generation of the SDF on the GPU later, but for the time being it's sufficient to give the motivation for computing it: in an approximate sense, it will serve as our means of rapidly computing the visibility of the sky from any point in the scene we wish to light.

Signed distance functions find application in several areas: they are an example of a 'potential function' which can rendered directly or indirectly using ray marching, level sets or marching cube techniques – for example to help accelerate the anti-aliased rendering of text and vector graphics [Pope01, Frisken02, Frisken06]. They are also useful in computer-vision in helping to find collisions, medial axes and in motion planning [Price05].

Signed distance fields can be computed analytically for simple shapes – and we shall be returning to this shortly – or tabulated in grids, octrees, or other spatial data structures. In this course we'll be using the simplest mapping to a GPU – namely an even sampling of the SDF over a volume, storing the values of the function in a volume texture. This kind of direct GPU representation is also used for a particular type of efficient GPU ray tracing called 'sphere tracing' – as used in the implementation of per-pixel displacement mapping in [Donnely05].

## 9.5 Curvature of a surface as an approximation to ambient occlusion

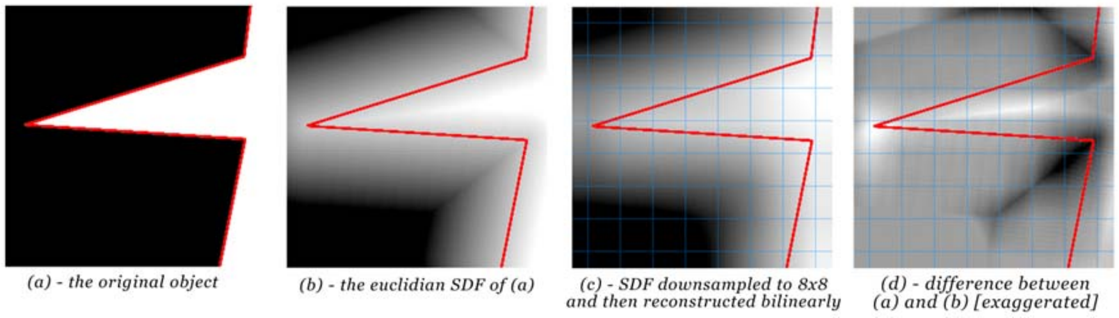
(a) - the original object
(b) - the euclidian SDF of (a)
(c) - SDF downsampled to 8x8 and then reconstructed bilinearly
(d) - difference between (a) and (b) [exaggerated]

***Figure 4**. (a) & (b) shows a part of a concave 2D object with its SDF. (c) shows the smoothing effect of sampling the SDF on a regular grid and then reconstructing it bilinearly. (d) Shows the difference between the smoothed and unsmoothed versions of the SDF. For points on the surface, the difference is related to the curvature of the surface – positive near concave corners (creases) such as the apex at the left, and negative near convex corners.*

The property we wish to make use of here is to do with using the SDF to find an approximation to the curvature of the surfaces of meshes in a scene [Masuda03]. In particular, we make the observation that in order to create the visual effect of darkening inside creases and crevices; we could darken the mesh wherever it has significant ‘concavity’ – inside wrinkles, behind ears, and so on – to simulate the effect of the sides of the concave region occluding the sky light. Figure 4 shows an example of how a regularly sampled SDF can help us – around a sharp concave feature, the SDF itself is also sharp and concave (Figure 4b). However, the errors introduced by the process of sampling the SDF on a regular, coarse grid and then reconstructing it using a simple trilinear filter have a smoothing effect (Figure 4c). A shader which sampled the SDF on the surface would expect the answer ‘0’ if the SDF representation was error-free; however in the case of a blurred SDF, the result will be related to the curvature of the surface – slightly positive in concave regions and slightly negative in convex regions (Figure 4d).

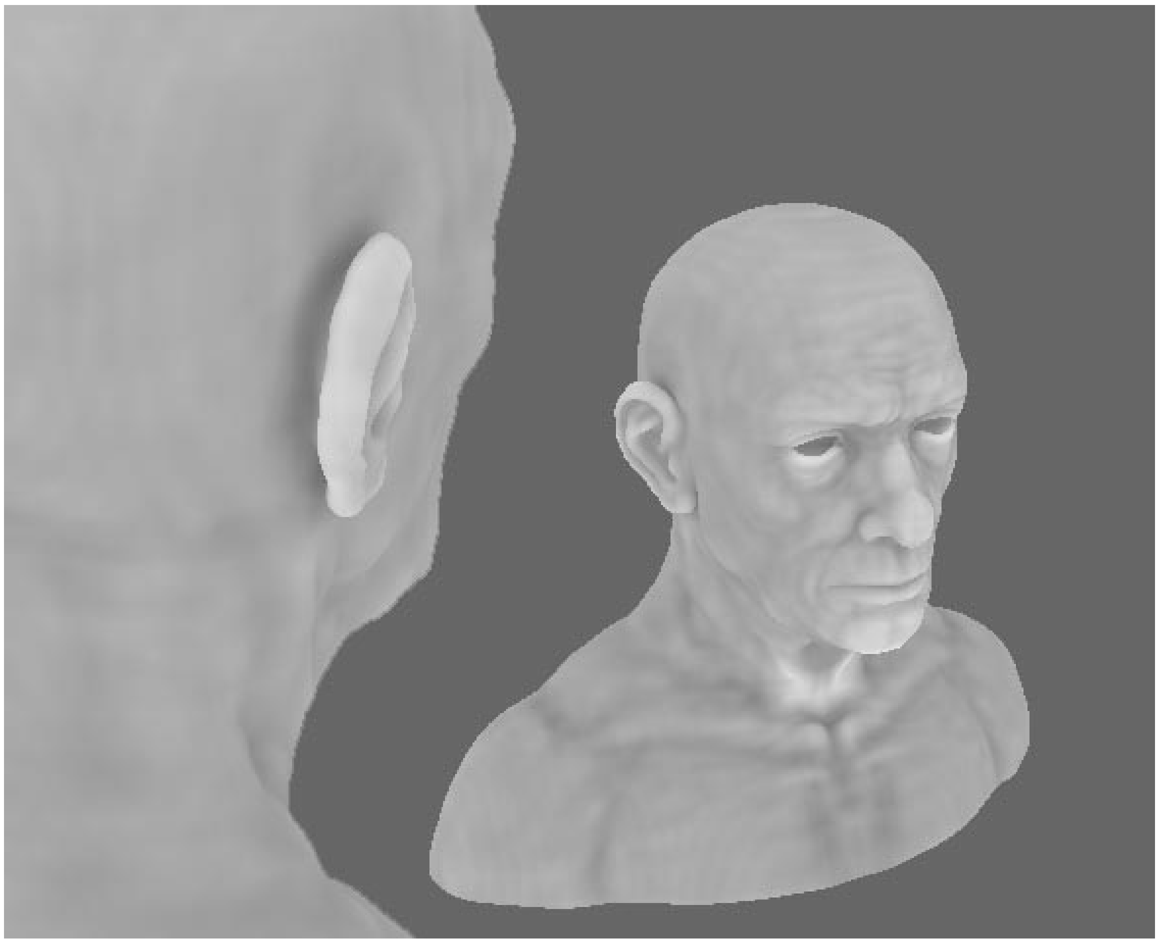

***Figure 5.*** *The result of visualizing the error in the sampled SDF, stored as an 8 bit per pixel 64x64x64 volume texture. (Two views).*

Figure 5 shows the result of visualizing this effect directly. The SDF of the mesh was pre-computed on the CPU once and uploaded as a 64x64x64 volume texture to the GPU. The mesh was then rendered with a simple shader that sampled the volume at the point to be lit (on the surface of the mesh). The output colour was $0.5+exp(k * S)$ where $k$ sets the overall contrast, and $S$ is the value of the SDF texture sampled at the rendered point. While Figure 5 doesn't yet look much like Figure 1, it shows some promise and exhibits the desired 'creases are dark' ambient occlusion aesthetic look, for example, behind the ears.

Point sampling the SDF and then reconstructing it trilinearly introduces high frequency aliasing, visible in the output (and Figure 5) as slight banding across the mesh surface. These artifacts can be greatly reduced by pre-filtering the SDF by low-pass filtering it with (for example) a separable Gaussian low-pass filter. An additional advantage of this pre-filtering step is that the width of the Gaussian filter allows us to effectively choose the characteristic scale of the concavities that the algorithm highlights. We will use this feature in the next section.

It should be noted that the distance encoded in the signed distance function may be measured in a number of different ways – Euclidian, 'Manhattan', chess-square etc., and may be exact or approximate. For the particular application of estimating curvature of a mesh outlined in these course notes, it turns out that the particular distance metric chosen isn't particularly important. The only required property of the 'pseudo SDF' is that we can estimate curvature in the manner outlined in Figure 4. Any function which

- can be quickly computed
- is positive inside objects, and negative outside
- decreases monotonically as the sample point moves away from object surfaces

It turns out that even a heavily blurred version of the binary image (eg blurring Figure 3a directly) satisfies these conditions; however the underlying concept of an SDF provides us with an intuitive theoretical basis for experimenting with different variations. This is simply another example of the breadth of experimentation that art-driven programmability affords us – the 'correct' choice of distance metric or blur kernel can only be guided by the particular 'look' required.

## 9.6 Achieving the skylight look via SDFs

Only one more observation is required to achieve convincing sky lighting using our SDF representation of the scene. We wish to incorporate sky shadowing effects on any point $P$, from *all* objects in the scene, whatever their distance from the point $P$, not just the effect of nearby concavities. Objects at a greater distance should have a lesser, 'blurrier' effect than those near the point $P$. Since the focus here is on achieving a desired visual affect without worrying too much about the physicality of the situation, we look to our SDF to see if there is some way that it can help us express these broader scale interactions.

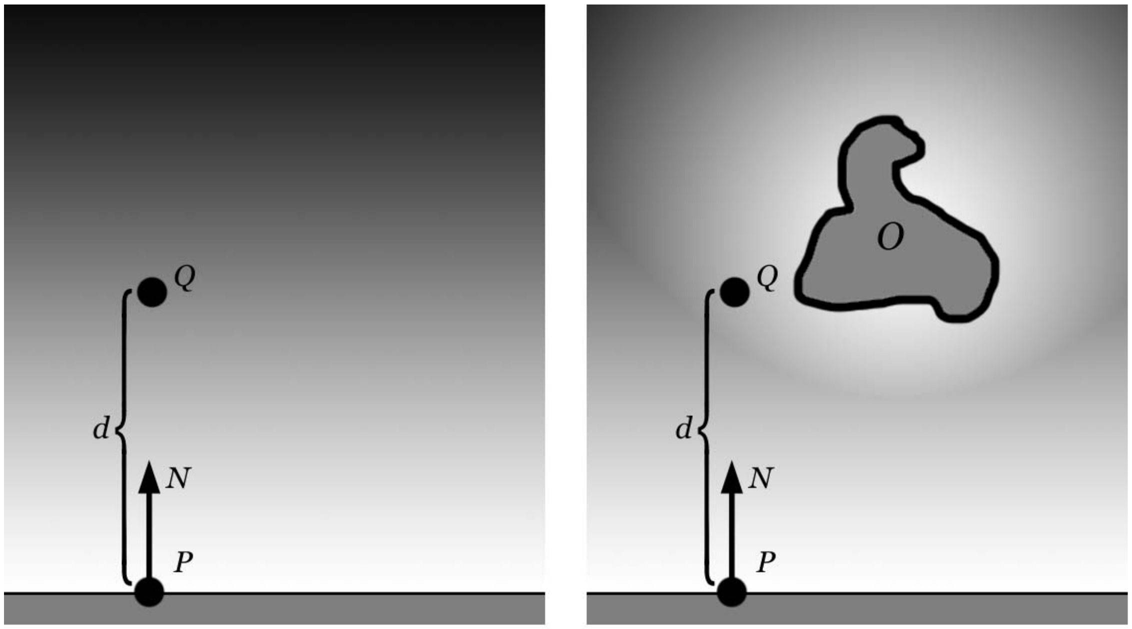

***Figure 6**. (a) As we move away from a point P in the normal direction N, the SDF will be proportional to the distance from P – for example, at point Q - in the absence of any*

*nearby ‘occluding’ objects. (b) Another object causes the SDF to be closer to 0 (white) than expected, at point Q.*

Figure 6 shows what happens at increasing distances from the surface of an object, with and without the presence of multiple objects (or oven a single object with a large-scale concave shape). According to the definition of our SDF, at a distance $d$ from $P$ in a direction $N$ perpendicular to the surface at $P$, we expect the SDF to be:

$$SDF(P+d\,N) = -d$$

This holds only in a scene with no objects ‘in front of’ $P$. However, in the presence of another object in the vicinity of the sampled point, the SDF will be closer to 0 (Figure 6b). (The value cannot be more negative, due to the property that at every point, the SDF records the *minimum distance* to the closest surface point).

This property allows us to take measurements of the occlusion / ‘openness’ of space at increasing distances from $P$. In particular, if we sample at $n$ points away from $P$, at distances $d_0, d_1, \ldots d_n$ with $d_0=0,$ and $d_i < d_{i+1}$, we expect

$$\sum_{i=0}^{n} SDF(P + d_i N) = -\sum_{i=0}^{n} d_i$$

in the absence of occluders.

Any difference from this represents some degree of occlusion of $P$ along the direction $N$ – and we visualize this difference as before, using an exponential:

$$C = e^{k\sum SDF(P+d_i N)+d_i}$$

where $C$ is the output of the shader.

Each sample (at a distance $d_i$) can be thought of as capturing the occlusion effects of objects at a distance $d_i$. To simulate the increased blurring effect of distant shadows, and to avoid artifacts that look like edges in the shadows caused by the point at $P+ d_i\ N$ ‘swinging’ around rapidly as $P$ and $N$ change, we sample different, pre-filtered copies of the SDF at each sample point. The pre-blur filter kernel size for each sample is set to be proportional to each $d_i$. That is, we sample a blurred copy of the SDF with the radius of blurring proportional to the distance of the sample from $P$.

This maps well to the GPU volume texture implementation when the $d_i$’s are spaced according to powers of two – that is, $d_i = 2\ d_{i-1}$ for $i > 1$, with $d_1$ set to the width of the voxels of the highest resolution volume texture. The pre-filtered volume textures for each SDF are then stored in the mip-map chain for the volume texture, and can be efficiently sampled using mip-level biased texture load instructions in the pixel shader.

Everything discussed so far makes no reference to the orientation of the sky hemisphere itself. As described, the result 'C' of the shader given above approximates ambient occlusion, which does not depend on any kind of directional light. To emulate the effect of the sky lighting coming from an oriented, infinitely distant hemisphere, we simply need to skew the sample positions 'upwards' towards the sky. This naturally introduces a bias in the sample positions that causes shadows to be cast downwards, and upward facing surfaces to be lit more brightly. We simply 'bend' $N$ towards the skylight by replacing $N$ with:

$$N' = N + \alpha U$$

(where $U$ is an upward pointing unit vector giving the orientation of the sky, $N$ is our normal at $P$ and $\alpha$ is a 'sky weighting factor' from 0 to 1).

Figure 7 shows the result of rendering a scene with these techniques, and overall quite a convincing skylighting effect is achieved in real time. (cf Figure 1, rendered using Autodesk 3D Studio Max 8's skylight). Note that there is not even any explicit '$N \cdot L$' term used – the lighting comes purely from the use of the bent normals described above, with a value of $\alpha$ of 0.5. The entire scene is represented by a single 128x128x128 SDF stored in an 8 bits per pixel monochrome volume texture. The shader used to render Figure 7 is given in listing 1. The image in Figure 7 was made with a *static* SDF computed once on the CPU. All that remains is to describe a technique by which we can compute the SDF of an arbitrary scene, at runtime. This will allow the rendering of dynamic scenes.

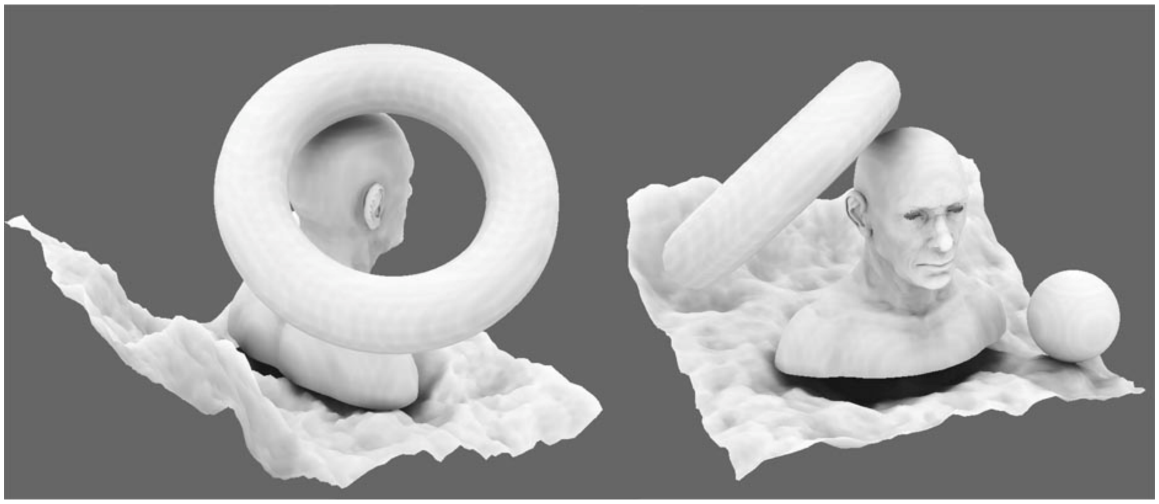

***Figure 7**. A scene rendered using a single, static 128x128x128 SDF sampled 4 times per rendered pixel at 4 different mip levels. All shading comes directly from the SDF technique.*

```
struct VolSampleInfo
{
 float3 minuv;
 float3 maxuv;
 float3 uvscale;
 float4 numtiles; // across in x, down in y, total slices in z,
                  // zfix = 0.5/numtiles.z
};

float4 SampleVolTex( sampler       bigtex,
                     float3        uvw,
                     VolSampleInfo vi,
                     sampler       vtl)
{
 uvw.xy*=vi.uvscale.xy;
 uvw.z-=vi.zfix;
 uvw.xyz=clamp(uvw.xyz,vi.minuv,vi.maxuv);

 float2 uv=uvw.xy;//*0.125;
 float4 fix = tex2D(vtl, uvw.z);
 float4 s1 = tex2D(bigtex,uv+fix.xy);
 float4 s2 = tex2D(bigtex,uv+fix.wz);
 return lerp(s1,s2,frac(uvw.z*vi.numtiles.z));
}

float4 MeshPS(BASIC_OUTPUT i) : COLOR0
{
 float3 n =normalize(i.Normal);
 float3 vec2light(0,0,1);
 float3 pos = i.WorldPos ;
 float3 delta = n* 0.03 + vec2light * 0.03;
 float4 light = exp(
  SampleVolTex(BasicSampler , pos+delta, vi,  VolTexLookupSampler)        +
  SampleVolTex(BasicSampler1, pos+delta*2, vi1, VolTexLookupSampler1)*1.2 +
  SampleVolTex(BasicSampler2, pos+delta*4, vi2, VolTexLookupSampler2)*1.4 +
  SampleVolTex(BasicSampler3, pos+delta*8, vi3, VolTexLookupSampler3)*1.8
 );
 return i.Diffuse * light * GlobalBrightness;
}
```

***Listing 1.*** *DirectX HLSL pixel shader code used to render figure 7. Since current GPUs don't allow rendering directly to the slices of a volume texture, the SampleVolTex function emulates a trilinear sample by sampling a 2D texture on which the slices are laid out in an order determined by a point sampled 1D lookup texture (VolTexLookupSampler in the code)*

## 9.7 Generating the SDF on the GPU

The final area to describe is how the SDF volume texture is generated in real-time. Many grid based approaches take as their starting point, a volume texture (in 3D) or texture (in 2D) initialized with binary values representing whether the given voxel is inside or outside the objects in the scene (see Figure 3a for an example). We call this the 'binary image'.

The simple brute force approach consists of simply looping over every voxel in the space, and then for each one, searching for the nearest voxel with opposite inside/outside-ness and recording its distance. Figure 3b was created in this manner, and Listing 2 shows C code for computing a 2D SDF on a 256x256 binary bitmap in this manner.

```
void ComputeSDF(unsigned char in[256][256], unsigned char
out[256][256])
{
 for (int y=0;y<256;y++) for (int x=0;x<256;x++)
 {
       int d=W*H;
       int p=in[y][x];
       for (int x2=0;x2<256;x2++) for (int y2=0;y2<256;y2++)
       {
             if (in[y2][x2]!=p)
             {
                   int d2=(x-x2)*(x-x2)+(y-y2)*(y-y2);
                   if (d2<d) d=d2;
             }
       }
       d=(int)(sqrt(d));
       if (d>127) d=127;
       if (p) d=-d;
       out[y][x]=d+128;
 }
}
```

***Listing 2***. *This code was used to generate Figure 3b*

The brute force method however is generally not tractable in 3D, since it is of cost $O(v^2)$ where $v$ is the total number of voxels in the scene. It is certainly not fast enough to use per-frame in an interactive application. Many techniques – such as the Chamfer Distance Algorithm (CDA) have been explored which use small nonlinear kernels to iteratively 'grow' an approximate band of distance values around the boundary of the objects, starting with the binary representation. Each iteration of the filter over the voxel space expands the area which has been correctly initialized by one voxel. See [Danielson80], [Price05], [Grevera04] for examples and pseudo-code.

Figure 5 was rendered using a SDF computed on the CPU using an algorithm similar to the dead reckoning algorithm of [Grevera04].

As mentioned in section 12.5, the SDF need not contain Euclidian distances to achieve an attractive result; indeed, simple Gaussian blurring of the binary image suffices to be able to judge curvature in the sense of Figure 4. Figure 7 was rendered using an SDF generated by repeatedly blurring and downsampling a 512x512x512 binary volume image of the scene, to create a 128x128x128 pseudo-SDF volume texture with 4 mip-maps down to 16x16x16.

## 9.8 Generating the binary image of a dynamic scene

Whether the SDF is generated through blurring, the Chamfer Distance Algorithm or a full Euclidian distance calculation, some GPU friendly way of generating the SDF volume texture is required. A key observation is that in many cases, a scene will be made up of a number of (possibly moving) objects. Each object need only be able to rapidly compute its own local SDF, before being composited into the final volume texture using the ‘min’ blending mode of the GPU to generate the ‘global’ SDF.

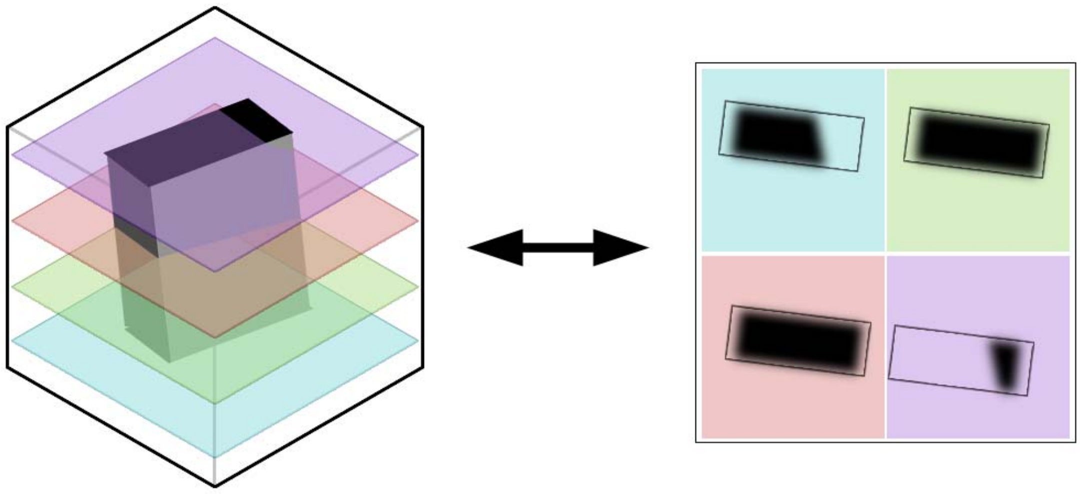

***Figure 8**. a) due to the limitations of current GPUs / APIs which do not allow direct rendering to the voxels of a volume texture, the slices of the SDF volume are laid out on a large 2D texture. The code to sample it is given in Listing 1. (b) a simple vertex shader which intersects the 4 edges of an OBB which most align with the volume slicing axis (normally ‘z’) allows the intersection of the OBB with the volume slices to be computed efficiently in a single draw call. In the right of the diagram, the vertex shader’s output quads are shown outlined; a pixel shader computes the actual SDF at each point within those quads (shown as blurry black regions).*

Since we are rendering into a volume rather than to the usual 2D render target supported by GPUs, we have to lay out the volume in slices (Figure 8). In our implementation, we created a vertex shader that was able in a single D3D draw call, to calculate the intersection of an arbitrary oriented bounding box with all the slices of the volume, and execute a pixel shader to update the SDF for only those ‘voxels’ which are inside the bounding box (Figure 8b). In this way, the SDF updating algorithm consists of:

```
Clear SDF render target representing the volume V of the entire
scene – assuming a 2D layout of the 3D slices as in Figure 8
For each object O in the scene
      Compute the oriented bounding box (OBB) of O
      Compute the intersection of the OBB of O with the
          slices of V
```

```
For each pixel within the intersection regions,
     Compute the approximate SDF of O,
     'min' blend the result into the SDF render target
```

Depending on the type of object O, we compute its SDF in one of several ways:

1. For a cuboid or ellipsoid, the SDF can be computed analytically.

2. For a rigidly deforming body with a complex shape, the SDF can be precomputed and stored as a volume texture in the local space of O

3. For a 'height field' object whose surface can be described as a single valued function of 2 coordinates (x, y), the SDF can be approximated directly from a blurred copy of the height field data describing the surface shape.

4. For a star shaped object (such as that shown in Figure 9) where the surface can be described as a single valued function giving the radius of the object along the direction to the object's local origin, the SDF can be approximated directly from a blurred copy of the 'height field' / Z buffer stored in a cubemap.

Options 1 and 2 are suitable for non-deforming (but affine transformable) objects. Options 3 and 4 (including a variant of 3 in which two height fields are glued back-to-back to create a more general shape) are useful for objects which may deform their shape from frame to frame. In these cases, the height field texture (or cube map for option 4) can be rendered dynamically to an offscreen buffer, using the GPU in a similar manner to the way in which shadow maps are computed on the GPU. The blurring of the Z buffer can also take place efficiently on the GPU; by rendering both the Z value and its square and then blurring both, (as described in [Donnely05] in the context of Variance Shadow Maps) the variance of Z over a range of blurred pixels can be used along with the mean Z to compute the width of the falloff of the SDF along the Z axis. (Figure 9b).

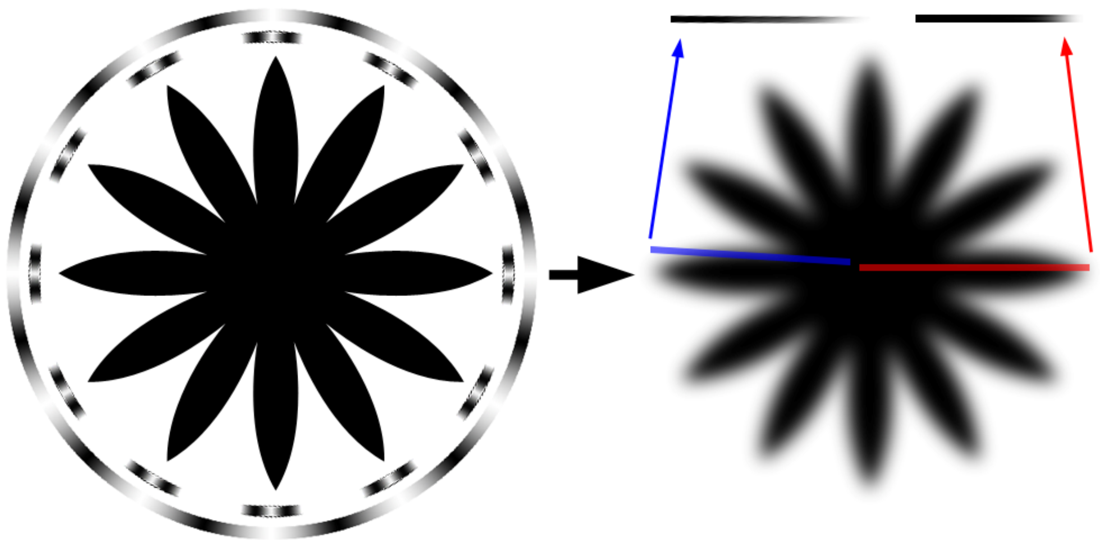

***Figure 9.*** *A star shaped object can have its SDF computed approximately from a radial cube map (or, in 2D as in this figure, a radial 1D texture) which stores the surface Z for*

*each theta. This is visualized above by the outer ring in (a). When blurring this Z buffer, the variance in Z is tracked as in [Donnely05] and used to scale the rate of falloff of Z in the SDF, since areas of high Z variance (marked by black parts of the inner ring in (a)) have a softer falloff from black to white along the radial direction, as shown in (b) by comparing the blue radial direction (high variance), with the red radial direction (low variance).*

Figure 10 shows a 'Cornell' type box rendered using this algorithm, consisting of 6 cuboids (5 walls and one central cube) whose local SDF are computed analytically in a pixel shader and composited into a 128x128x128 SDF for the whole scene. Figure 11a shows a similar scene with the central box replaced with a complex mesh whose SDF was precomputed in local space (option 2 above); Figure 11b shows a star shaped object with dynamically generated SDF based on a cube map (option 4).

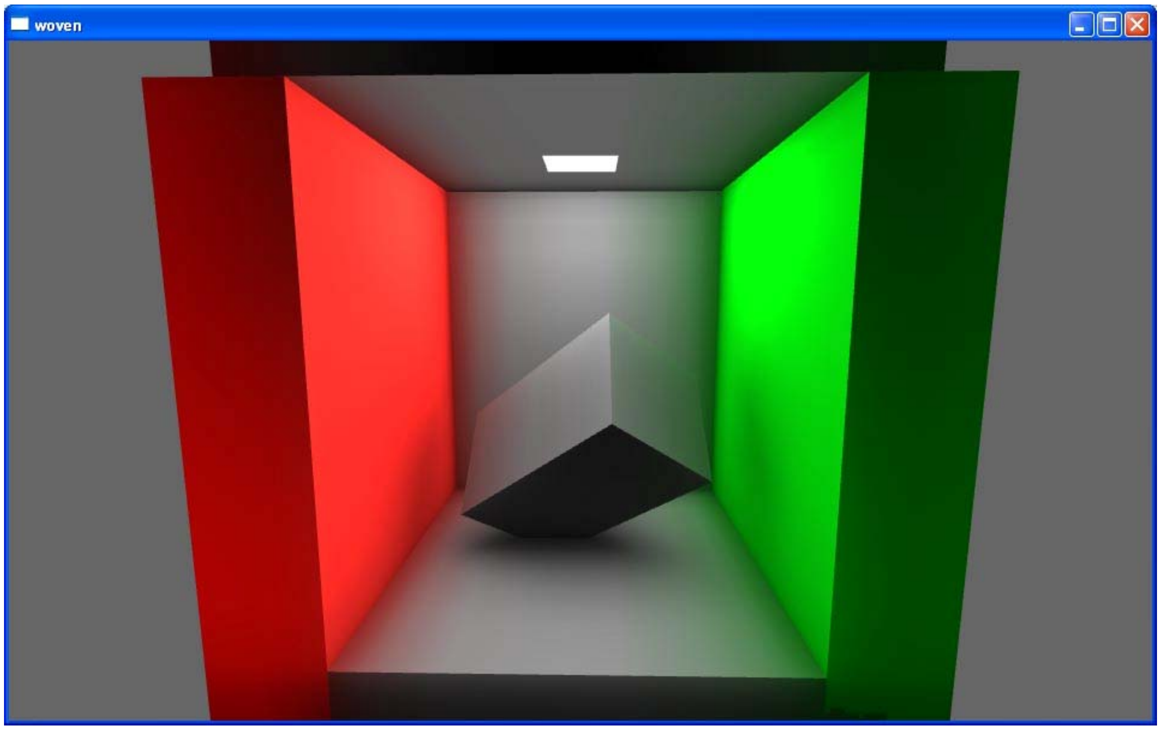

***Figure 10**. An oriented cube in a Cornell box. All shadowing in this image comes from the SDF; in addition surfaces were lit using an N · L term scaled by the results of the SDF lookups.*

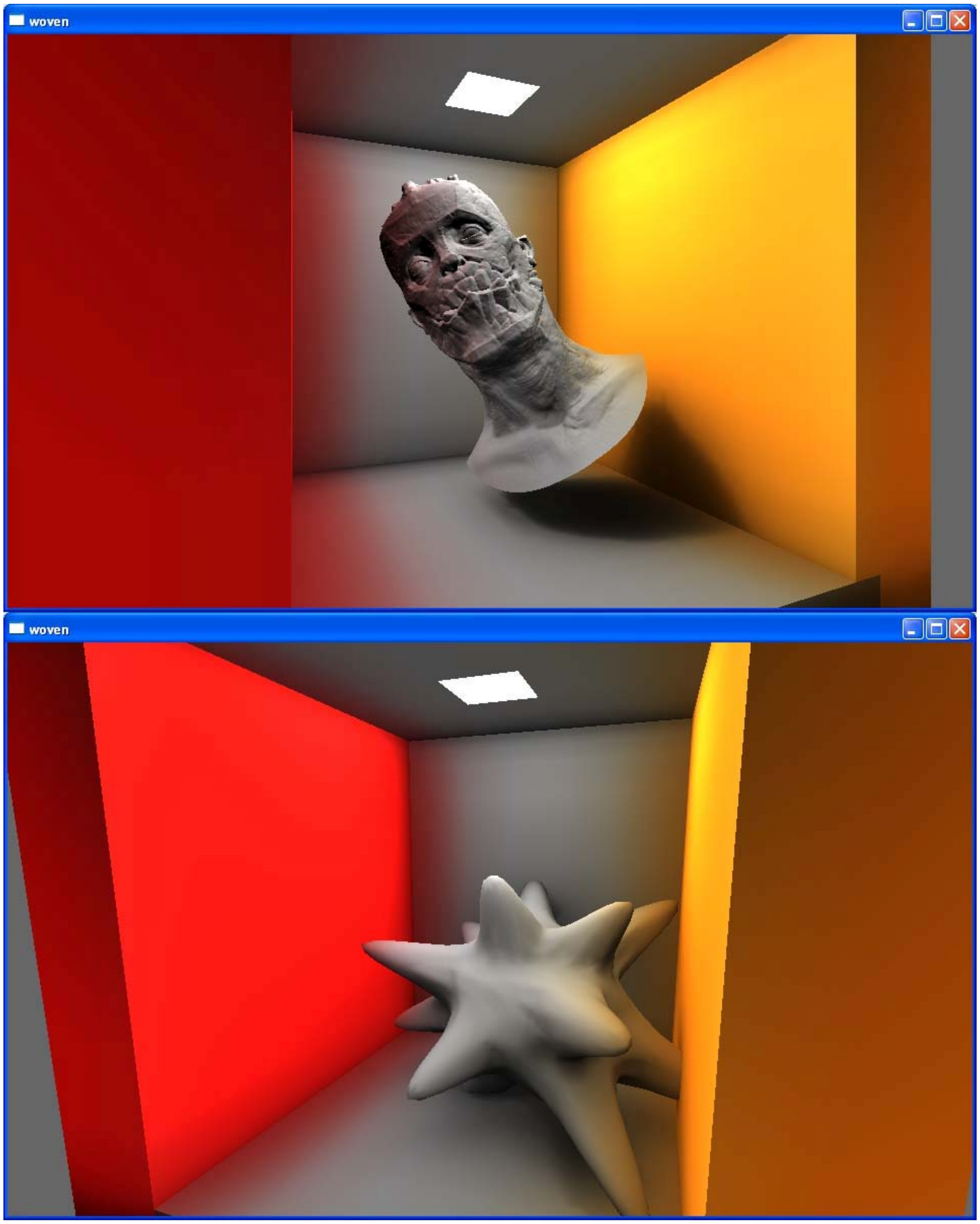

***Figure 11**. Two example scenes with more complex object shapes. (a) has 2 million polygons and a precomputed local SDF stored in a 32x32x32 volume texture. (b) has a dynamically rendered 32x32x6 cube map of radii which is used to approximate its local SDF.*

## 9.9 Approximate colour bleeding and future directions

Figures 10, 11a and 11b all exhibit small amounts of colour bleed. This was achieved by computing the SDFs separately for red, green and blue channels, and then biasing the SDF in each channel for each object by its diffuse surface colour. The red reflective wall had its red channel of its local SDF increased by a small constant factor, causing all nearby SDF lookups to return a slightly red hue. Despite being physically incorrect, in the sense of not accurately capturing the physics of light bouncing from one diffuse surface to another, it's this kind of simple, free yet aesthetically pleasing trick which can go a long way in a real-time environment.

In the spirit of 'learning from mistakes', the success of this kind of non-physical 'short-cut' also suggests an extension to arbitrary light sources (rather than just sky-lights) based on an interpretation of this skylighting algorithm that has more in common with the Ambient Occlusion Fields of [Kontikanen05] than SDFs. If the volume texture placed over the scene, is considered more as an approximation of the light reaching each point in space, as opposed to the SDF of the occluders in that scene, then an alternative algorithm supporting an arbitrary number of emissive objects in the scene (rather than just a distant skylight). The idea is to start with a blurry copy of the 'binary image' of the scene, stored in a volume texture. In addition, an even lower resolution volume texture is computed containing a vector for each point, containing a direction pointing away from the average of nearby light sources. This volume can be thought of as the first 4 terms (constant and linear) spherical harmonic coefficients of an irradiance volume that does not take into account occluders.

The latter volume, which we shall term 'the light direction' function, can be used to 'advect' the contents of the pseudo-SDF volume texture. This effect is best visualised by observing the music visualisation features of media players such as Apple's iTunes™ or WinAmp. These take a simple bitmap image, and at each frame radially zoom and blur the image, feeding the result back into the next frame. If you imagine radially blurring the SDF volume texture according to the directions stored in the 'light direction' volume texture (in the case of a single point light source, radially from the light), the resulting volume will converge towards a good-looking approximation to the irradiance at every point in space.  Figure 12 shows this effect in 2D.

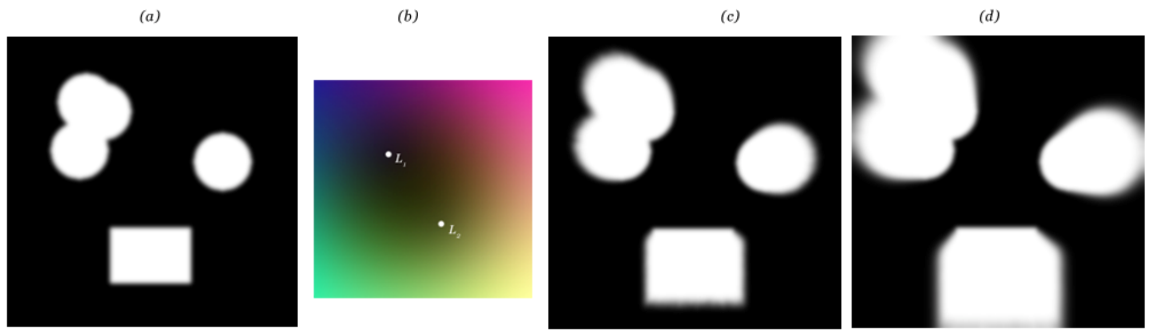

***Figure 12**. (a) starting with a binary image of the scene's occluders ('SDF'), and a lower-resolution texture encoding light flow direction at every point (b), the SDF texture is repeatedly zoomed, blurred and blended with itself (c) to create an approximation to*

*the irradiance at every point (d) – shown with colours inverted. This result can then be used to render the scene with soft shadowing from the light sources included in the original light flow texture.*

## 9.10 Conclusion

The flexibility of GPUs was exploited in this description of an unusual soft-shadow rendering algorithm. Although it is of limited use since it requires the whole scene to be represented in a volume texture, it demonstrates a process of art-led discovery, approximation and exploration in real-time graphics rendering which the author believes will continue to be one of the driving forces behind exciting, visually unique games and real-time applications, running on commodity GPU hardware.

## 9.11 Bibliography

DANIELSON, P.-E., 1980. Euclidian Distance Mapping. *Computer Graphics and Image Processing* 14, pp. 227-248

DONNELLY, W. 2005. Per-Pixel Displacement Mapping with Distance Functions, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*, Chapter 8, Matt Pharr (ed.), Addison-Wesley. http://download.nvidia.com/developer/GPU_Gems_2/GPU_Gems2_ch08.pdf

DONNELLY, W., LAURITZEN, A. 2006. Variance Shadow Maps. In the proceedings of *ACM SIGGRAPH 2006 Symposium on Interactive 3D Graphics and Games*.

EVANS, A. 2005. Making Pretty Pictures with D3D. GDC Direct3D Tutorial 2005, http://www.ati.com/developer/gdc/D3DTutorial07_AlexEvans_Final.pdf

EVERITT, C. 2001. Interactive Order-Independent Transparency. NVIDIA white paper, 2001, http://developer.nvidia.com/object/Interactive_Order_Transparency.html

FRISKEN, S.F.; PERRY, R.N. 2002. Efficient Estimation of 3D Euclidean Distance Fields from 2D Range Images, *Volume Visualization Symposia* (VolVis), pp. 81-88

FRISKEN, S. F., 2006. Saffron: High Quality Scalable Type for Digital Displays, Mitsubishi Electric Research Laboratory (MERL), http://www.merl.com/projects/ADF-Saffron/

GREVERA, G. J. 2004. The ‘‘Dead Reckoning’’ Signed Distance Transform, *Computer Vision and Image Understanding* 95 (2004) pp. 317–333.

KAJIYA, J. T. 1986. The Rendering Equation. *Computer Graphics* 20 (4), 143-149

KONTKANEN, J., LAINE, S. 2005. Ambient Occlusion Fields, In the proceedings of *ACM SIGGRAPH Interactive Symposium on 3D Graphics and Games*

MASUDA, T. 2003. Surface Curvature Estimation from the Signed Distance Field. 3dim, p. 361, *Fourth International Conference on 3-D Digital Imaging and Modeling* (3DIM '03)

OAT, C. 2006. Ambient Aperture Lighting, "Advanced Real-Time Rendering in 3D Graphics and Games", Course 26, ACM SIGGRAPH

POPE, J., FRISKEN, S. F., PERRY, R.N. 2001. Dynamic Meshing Using Adaptively Sampled Distance Fields, Mitsubishi Electric Research Laboratory (MERL) Technical Report 2001-TR2001-13

PRICE, K. 2005. Computer Vision Bibliography Webpage. http://iris.usc.edu/Vision-Notes/bibliography/twod298.html