Rendering Fake Soft Shadows with Smoothies

Eric Chan
Massachusetts Institute of Technology

Real-Time Soft Shadows

Goals:

- Interactive framerates
- Hardware-accelerated
- Good image quality
- Dynamic environments

NVIDIA

Challenge:

- How to balance quality and performance?

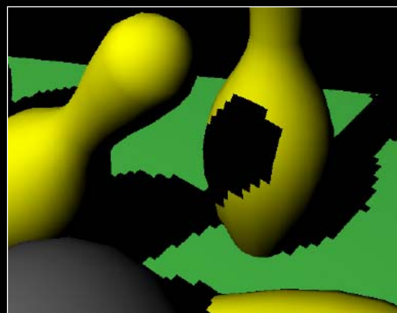
There are many (often conflicting) goals in the design of real-time soft shadow algorithms. On the one hand, we want interactive framerates, which usually means we need to have an algorithm simple enough to map directly to graphics hardware. On the other hand, we want high image quality and the ability to use the algorithm for dynamic scenes where anything – light, objects, camera – can move from frame to frame. Not surprisingly, any real-time shadow algorithm will involve tradeoffs. In a nutshell, the algorithm presented in this session attempts to balance the quality and performance requirements: it is not geometrically accurate, but the results appear qualitatively like soft shadows, and the algorithm scales well to complex scenes.

Ordinary Shadow Maps

Image-space algorithm:

- Fast and simple
- Supported in hardware
- Aliasing artifacts

NVIDIA



Sen et al. [SIGGRAPH 2003]

Our work can be seen as an extension of the shadow map technique. As we discussed in earlier sessions, shadow maps simply use a depth map to identify regions of the scene that are visible to the light source. The algorithm is fast, simple, and general. It is accelerated in modern graphics hardware. However, the method is susceptible to undersampling artifacts such as aliasing, and we have seen a number of techniques developed to combat this problem. Such techniques include perspective shadow maps and shadow silhouette maps. As we will see later, the method proposed in this session is primarily designed to produce soft shadows, but a nice side effect is that the algorithm also tends to mask aliasing effects. In that sense, we'll be seeing yet another algorithm which reduces the aliasing of shadow maps.

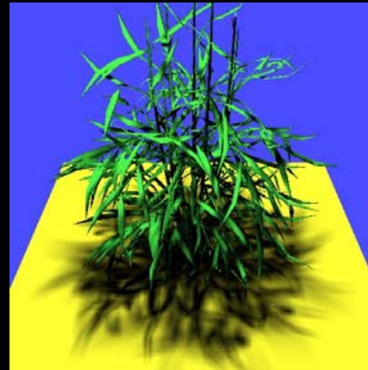
Soft Shadow Maps

Techniques:

- Filtering
- Stochastic sampling
- Image warping

Examples:

- Percentage closer filtering (Reeves et al., SIG1987)
- Deep shadow maps (Lokovic and Veach, SIG2000)



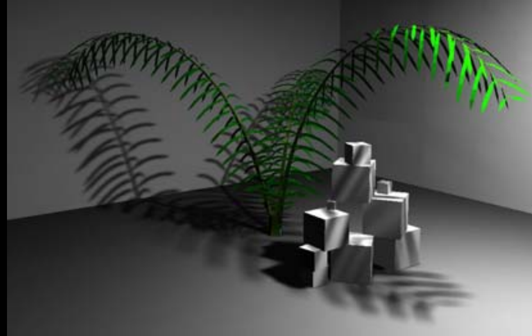
Agrawala et al. [SIGGRAPH 2000]

But: need dense sampling to minimize artifacts

Over the years, many researchers have extended the original shadow map algorithm to support antialiased and soft shadows using a combination of filtering, stochastic sampling, and image warping techniques. Unfortunately, noise and banding artifacts appear in the results unless a high number of samples are used (usually at least 64). Therefore, even though these techniques have been used successfully in the motion picture industry by companies such as Pixar, they are not easily adapted for real-time applications.

Soft Shadow Maps (cont.)

Approximations



Soler and Sillion

Examples:

- Convolution (Soler and Sillion, SIGGRAPH 1998)
- Linear lights (Heidrich et al., EGRW 2000)

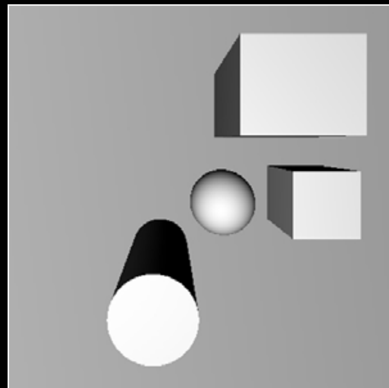
Another class of soft shadow techniques use various approximations to the true soft shadow. For instance, the soft shadows in the image shown here were generated by convolving blockers against an area light source. The method proposed in this session is also an approximate method.

Idea

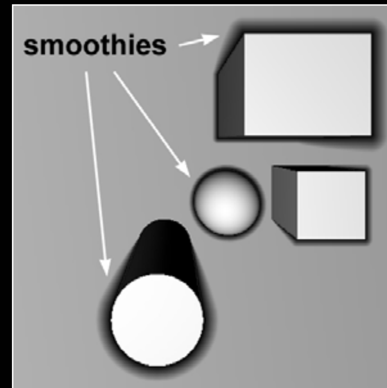
SIGGRAPH2004

Extend basic shadow map approach

Extra primitives (smoothies) soften shadows



light's view (blockers only)



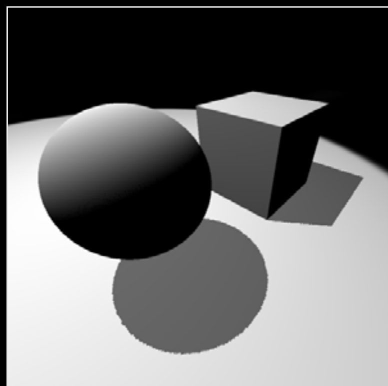
light's view (blockers + smoothies)

The idea of our soft shadow algorithm is just an extension of the shadow map approach. We use extra geometric primitives called "smoothies" to soften shadow edges. These primitives are attached like fins to the blockers' silhouettes. The image on the left shows the blockers, seen from the point of view of the light source. The image on the right shows the same blockers, but with the smoothies attached to the silhouettes. The idea is to render the smoothies into an alpha map, then apply the alpha map as a projective texture so that the resulting shadow edges will be smooth.

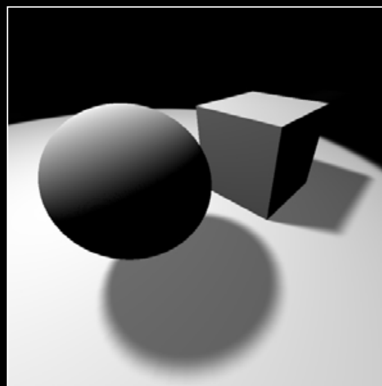
Fake Soft Shadows

Shadows not geometrically correct

Shadows appear qualitatively like soft shadows



Hard shadows



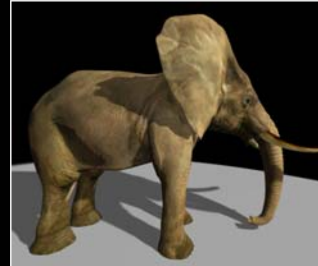
Fake soft shadows

I should mention up front that the shadows generated using this method are not geometrically correct, i.e. we're not actually modeling an area light source. In fact, we don't even taken the shape or orientation of the light source into account. However, the resulting shadows do have some of the important qualitative aspects of soft shadows. For instance, look at the box casting a shadow onto the ground plane. The shadow edge is sharp at the point where the box meets the ground, and it is softer farther away from the contact point. This dependence on the ratio of distances between the light source, blocker, and receiver is a key property of soft shadows that we attempt to simulate.

Smoothie Algorithm

Properties:

- Creates soft shadow edges
- Hides aliasing artifacts
- Efficient (object / image space)
- Hardware-accelerated
- Supports dynamic scenes



Here are some of the properties of the smoothie algorithm. Its primary purpose is to create soft shadow edges, and a side effect is that the typical aliasing artifacts of shadow maps are masked. The algorithm achieves efficiency by combining both image-space and object-space techniques. It is easy to implement on programmable graphics hardware, and there's no precomputation, so it works fine for dynamic scenes.

References

Rendering Fake Soft Shadows with Smoothies

- E. Chan and F. Durand [EGSR 2003]

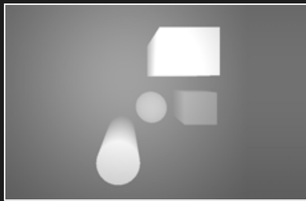
Penumbra Maps

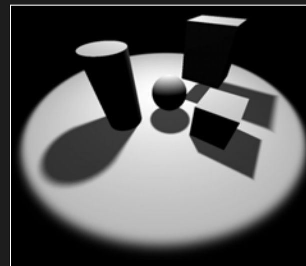
- C. Wyman and C. Hansen [EGSR 2003]

Before we dive into the algorithm details, let me mention two research papers that describe the algorithm in more detail. The idea was developed independently by Fredo Durand and myself at MIT, and by Chris Wyman and Chris Hansen at the University of Utah. The two papers were published simultaneously at the Eurographics Symposium on Rendering in 2003. The algorithms described in these two papers are essentially the same. The main difference lies in the way the extra geometric primitives are constructed, but this difference is not very important.

Algorithm

Algorithm Overview



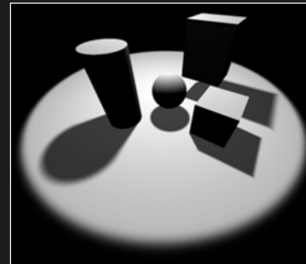


Focus on concepts

Implementation details later

Here is an overview of the smoothie algorithm. We'll first discuss the algorithm conceptually and then see how to implement it using graphics hardware.

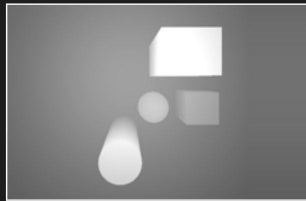
Algorithm Overview

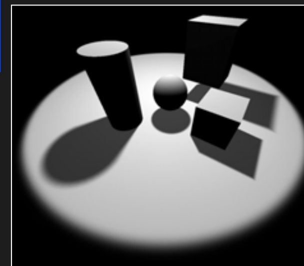


The algorithm consists of three rendering passes. In the first step, we create a standard shadow map, i.e. render the scene from the light's viewpoint and store the nearest depth values into a buffer.

Algorithm Overview

Step 2

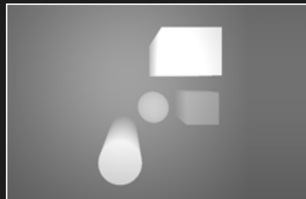


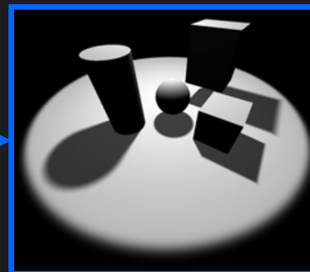


Create smoothie buffer

In the second step, we construct extra geometric primitives called "smoothies" and render them from the light's viewpoint. You can think of this extra geometry as "fins" that are attached to the blockers' silhouettes. When drawing the smoothies, we compute two quantities at each pixel, a depth value and an alpha value, and store them together into the smoothie buffer.

Algorithm Overview

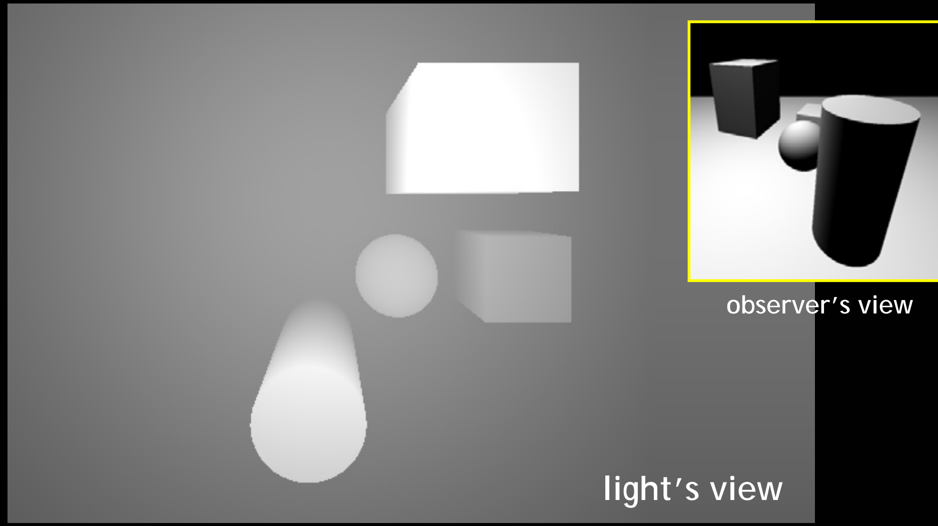




Render scene + shadows

In the final step, we draw the scene from the observer's viewpoint and compute the shadows. We refer to both the shadow map and the smoothie buffer to compute shadows with soft edges.

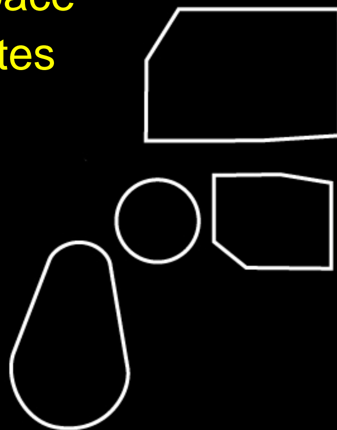
You may have noticed that this algorithm is similar in structure to the shadow silhouette map algorithm, which was covered in an earlier session. Both algorithms involve three rendering passes. The first one creates a standard shadow map, the second one creates an auxiliary buffer of some sort, and the third one performs lookups into both buffers to compute shadows. Of course, the two algorithms are designed with different goals in mind, but it is interesting to note the similarities.

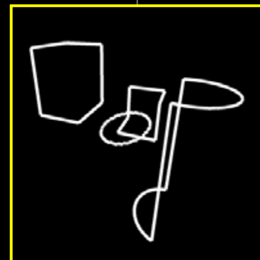


Now let's take a closer look at the algorithm. We first create a shadow map from the light's viewpoint.

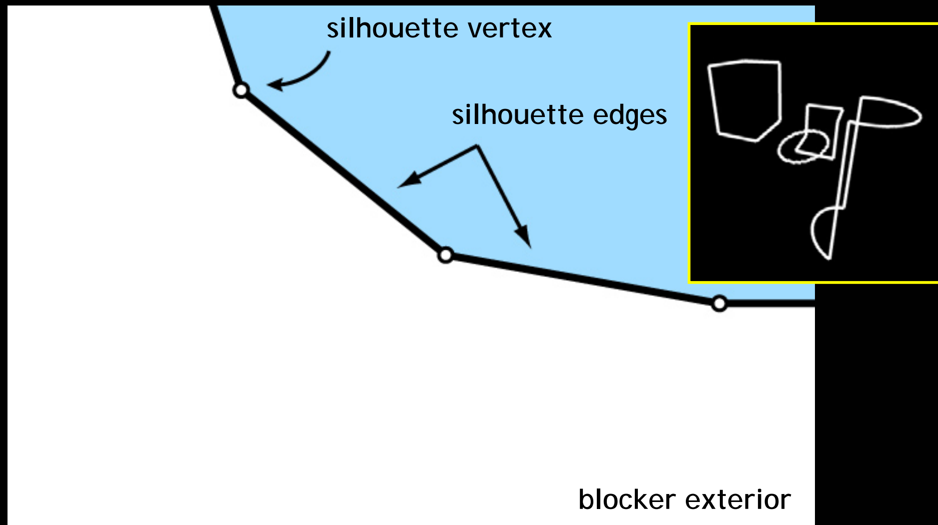
Find Silhouette Edges

Find blockers' silhouette edges in object space





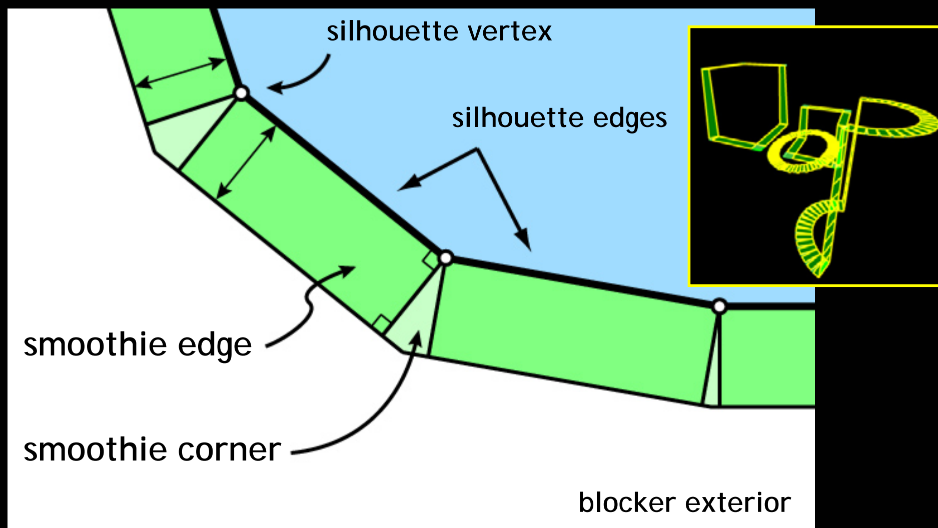
Next, we identify the object-space silhouettes, seen from the point of view of the light source. (This exact same step is also required for the shadow volume algorithm.) There are many ways to compute object-space silhouettes. A simple way is to loop through all the edges in the model and check if one adjacent face is facing the light source and the other face is facing away. Note that we've implicitly made an assumption here: our blockers are represented as polygons.



Now that we have the silhouette edges, we can go ahead and construct the smoothies. The diagram here shows a blocker (light blue) and its silhouette edges and vertices.

Construct Smoothies

Blocker + smoothies:



The smoothies (edges and corners) are shown in light green. Smoothie edges are fixed-width rectangles in the screen space of the light source. Smoothie corners are quads that connect adjacent smoothie edges, as shown in the diagram.

The diagram shows a convex silhouette curve. For concave silhouettes, we omit the smoothie corners. This causes smoothie edges to overlap each other, but ultimately it doesn't cause problems because, as we will see, we use minimum blending to handle the case when multiple smoothies overlap in screen space.

How thick do we make the smoothie geometry? Intuitively, the wider we make a smoothie, the softer the shadow becomes. Thus, the size of a smoothie should depend on the size of the light source being simulated.

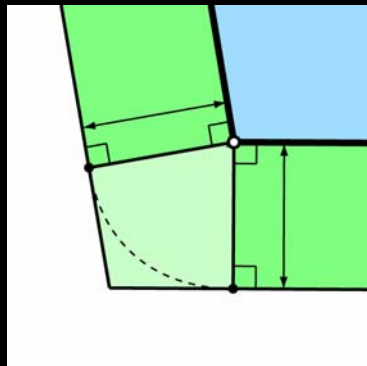
Chris Wyman and Charles Hansen [EGSR 2003] describe a different geometric construction using "cones" and "sheets." This construction was originally proposed by Eric Haines for rendering soft planar shadows [JGT 2001].

Construct Smoothies

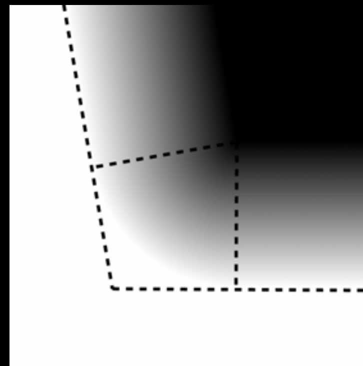
SIGGRAPH2004

Smoothie edges are fixed-width rectangles in screen space

Smoothie corners connect adjacent smoothie edges



geometry

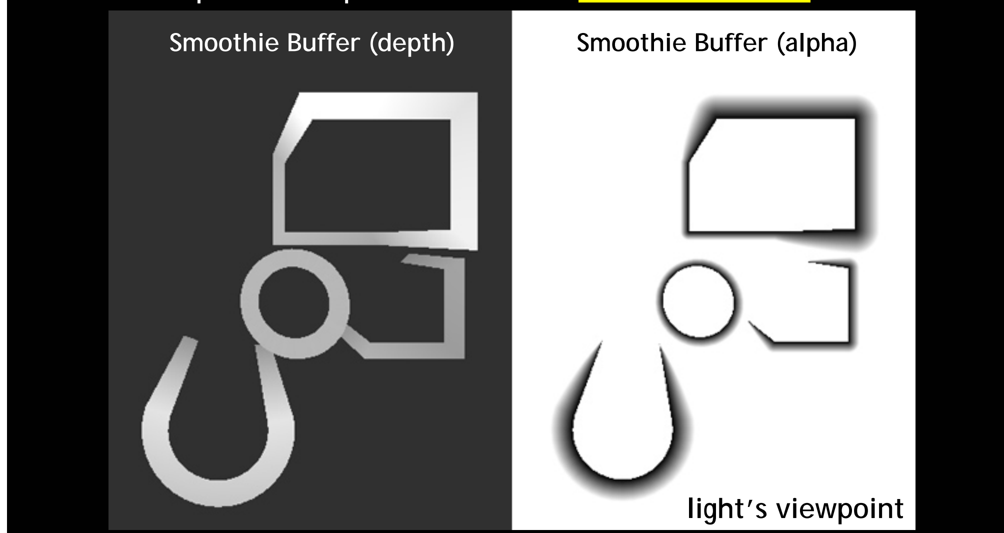


shading

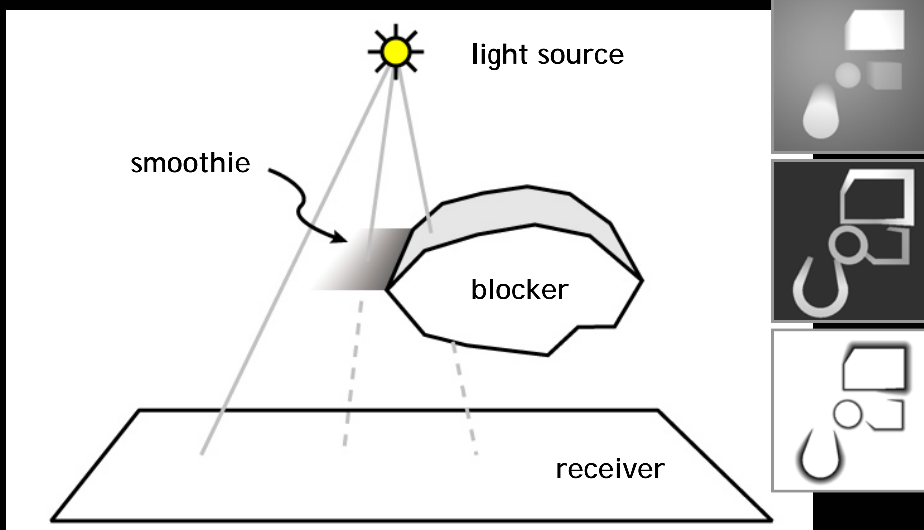
In case it's unclear where we're going with the smoothie construction, perhaps these diagrams will help. The idea is that we'll compute alpha values for the smoothies (shown on the right) in such a way so that when we project them onto the scene from the light's viewpoint (via projective texture mapping), the shadow edges will appear smooth.

Render Smoothies

Store depth and alpha values into smoothie buffer



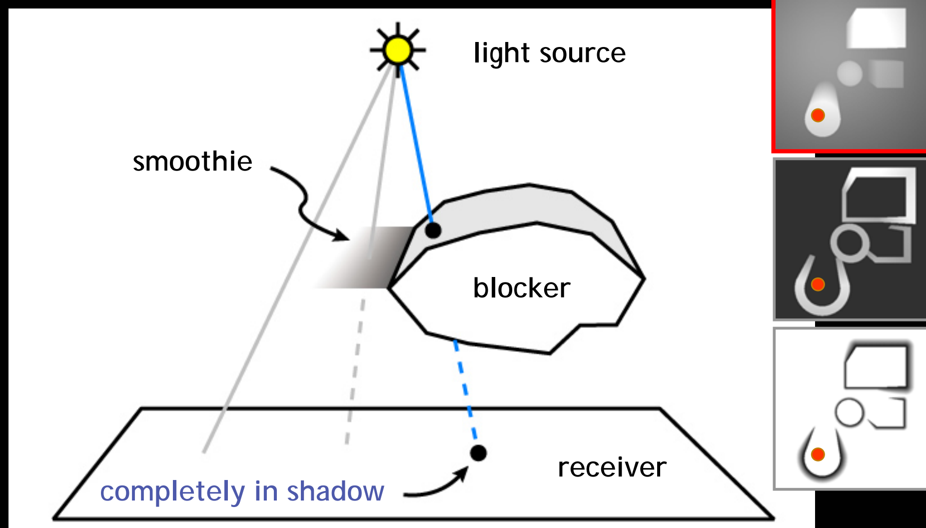
Now that we've constructed the smoothies, we draw them from the light's viewpoint into the smoothie buffer. We compute both depth (shown on left) and alpha values (shown on right) for each smoothie pixel. Rendering depth is straightforward (the same as when rendering a shadow map). We'll come back and discuss how we compute the alpha values. We ought to be careful how we compute them, since they will ultimately determine how our shadows appear in the scene!



Finally, we render the scene from the observer's viewpoint. For each sample, we check for three cases.

Compute Shadows (1 of 3)

Image sample behind blocker (intensity = 0)

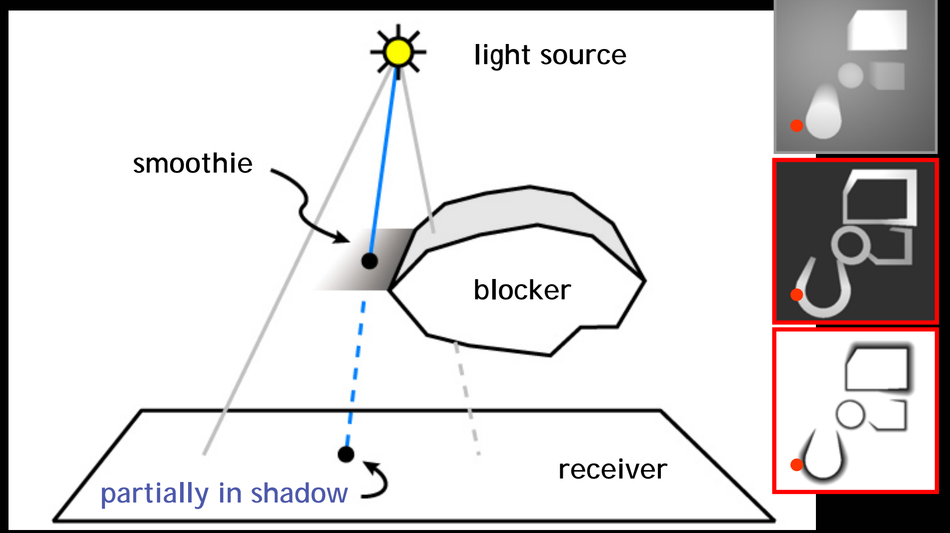


We project the sample into light space so that we can perform lookups into the shadow map and smoothie buffer. If the sample is behind the shadow map, i.e. the depth of the sample is greater than the depth value stored in the shadow map, then we say the sample is completely in shadow and assign it zero intensity.

In the diagram, the three images on the right show the depth map (top), smoothie buffer depth (middle), and smoothie buffer alpha (bottom).

Compute Shadows (2 of 3)

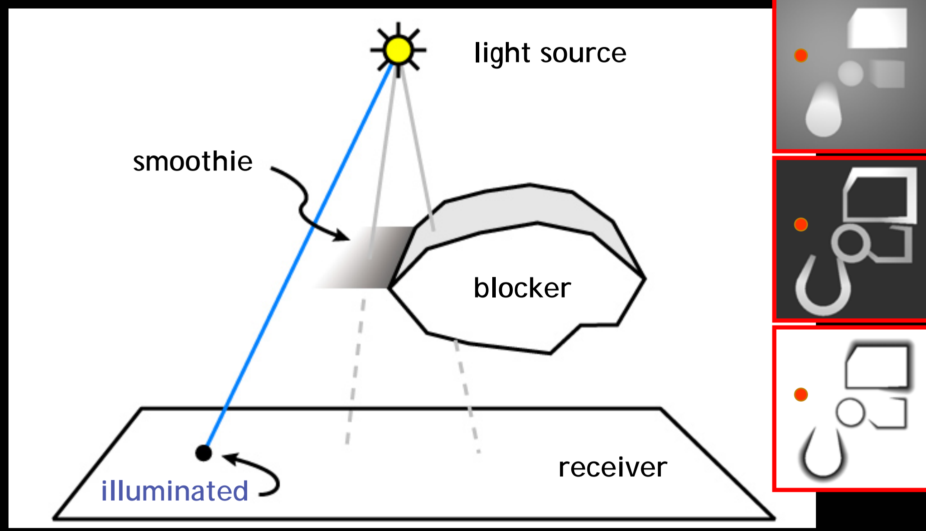
Image sample behind smoothie (intensity = α)



If the sample is not behind a blocker, but it is behind a smoothie, then we shade the sample using the smoothie's alpha value. We use the depth value stored in the smoothie buffer to make the smoothie depth comparison, and we retrieve the alpha value from the alpha part of the smoothie buffer.

Compute Shadows (3 of 3)

Image sample illuminated (intensity = 1)



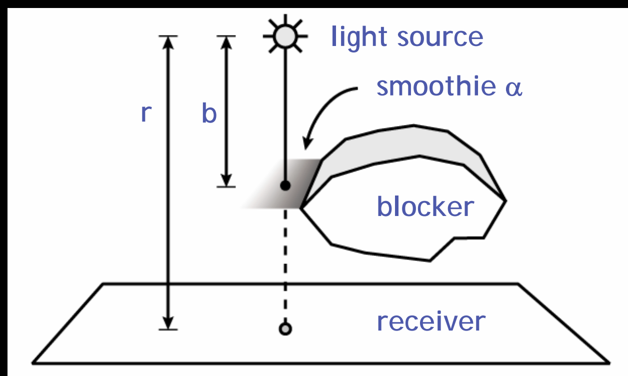
Finally, if the sample is neither behind a blocker nor behind a smoothie, then it is considered fully illuminated.

It is important to realize that this method is not geometrically accurate. For instance, since the smoothies always grow outwards from the blockers, we only capture the “outer penumbra” of the shadow, meaning that shadows always have a full umbra with our approach. In reality, as the size of the light source increases, the umbra of the shadow should decrease, and in particular for a sufficiently large light source the umbra should vanish entirely. Later on we will see how this limitation of our approach affects the image quality.

Computing Alpha Values

Intuition:

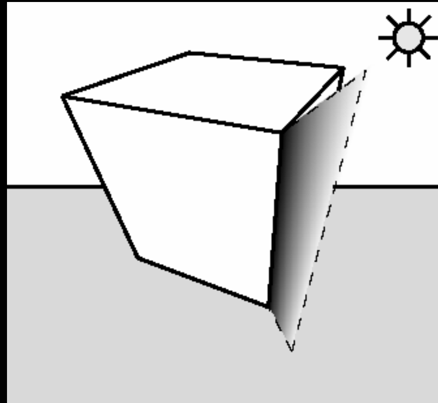
- Alpha defines penumbra shape
- Should vary with ratio b/r



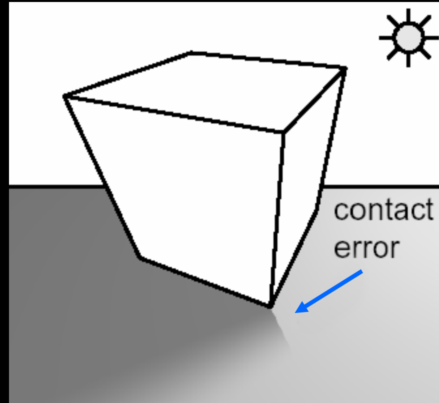
Now that we've seen the sequence of steps involved in the smoothie algorithm, let's get back to how we compute the alpha values when rendering the smoothies. Consider the diagram shown here, which shows the high-level geometric relationship between a light source, blocker, and receiver. One of the properties of soft shadows is that the width of the penumbra depends on the ratio of b (the distance from the light to the blocker) and r (the distance from the light to the receiver). As b becomes close to r, the shadow becomes sharper (less penumbra). As b becomes much smaller than r, then the shadow becomes very soft (large penumbra). Thus, intuitively, our computation alpha should somehow involve the ratio of b/r.

Without Alpha Remapping

Linearly interpolated alpha → undesired results!



smoothie

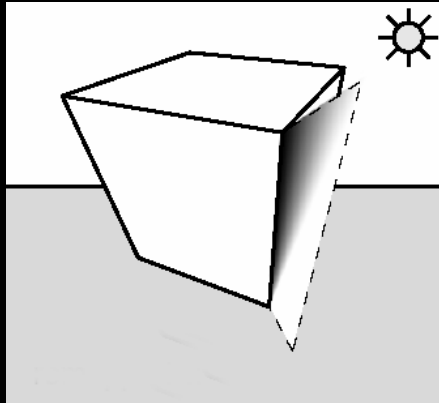


contact problem

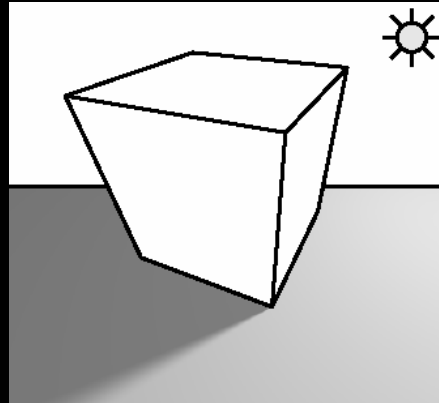
Let's see what happens if we don't take into account this ratio b/r. Intuitively, it seems that we should have alpha start from 0 at the blocker (0 meaning complete occlusion by the blocker) and fade to 1 at the edge (1 meaning 100% visible). Notice what happens, though, if we just linearly interpolate the alpha value across the smoothie (shown in left image) and then project the smoothie onto the ground plane. Although the edge itself is soft, there is clearly a problem at the contact point between the box and the ground. Whereas we expect to see a hard shadow near the contact point, we instead get a "disconnected" shadow because we did not taken into account the ratio of distances.

With Alpha Remapping

Remap alpha at each pixel using ratio b/r: $\alpha' = \alpha / (1 - b/r)$



smoothie



fixed contact problem

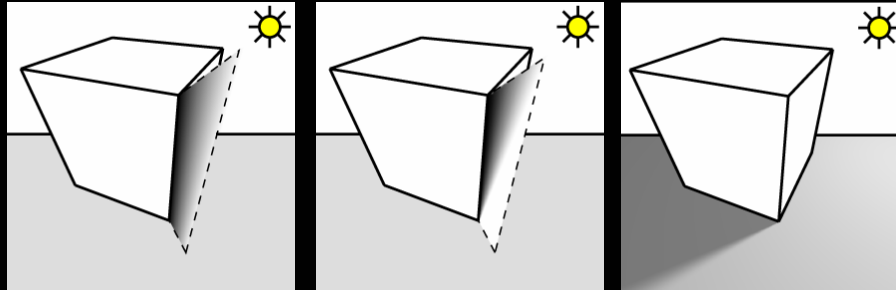
All we have to do instead is remap the alpha values at each pixel using a simple formula, shown here. The original alpha (without the prime) is obtained by linearly interpolating from 0 to 1 across the smoothie. Remapping the alpha using this equation creates the “warped” alpha effect shown in the image on the left. When projected onto the ground plane, we obtain the desired effect: the shadow is hard near the contact point and gets softer farther away.

Computing Alpha Values

SIGGRAPH2004

1. Linearly interpolate alpha
2. Remap alpha at each pixel using ratio b/r:

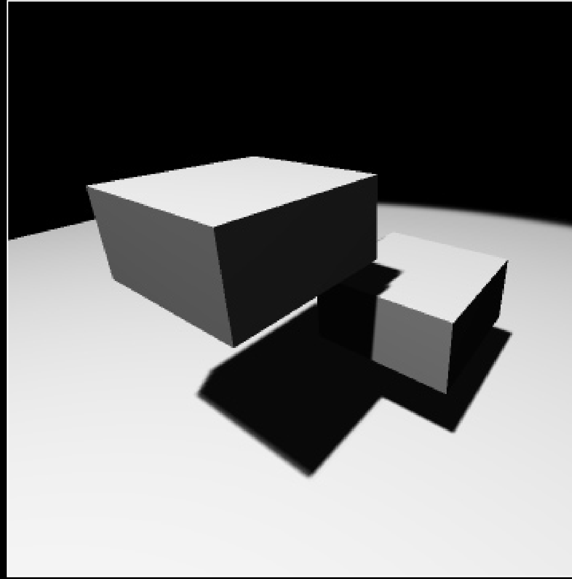
$$\alpha' = \alpha / (1 - b/r)$$



original α　　remapped α　　result

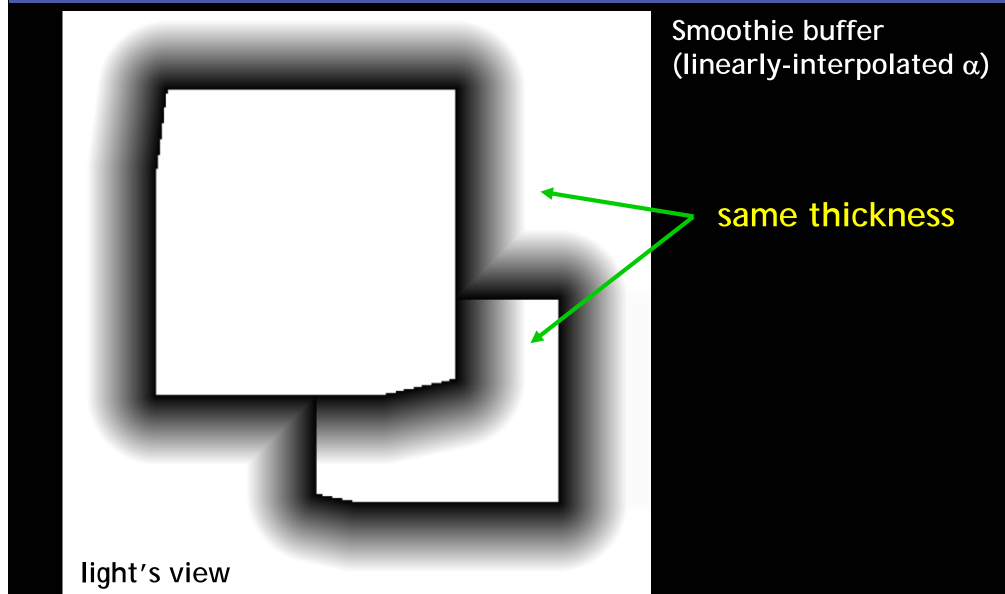
Multiple Objects

SIGGRAPH2004

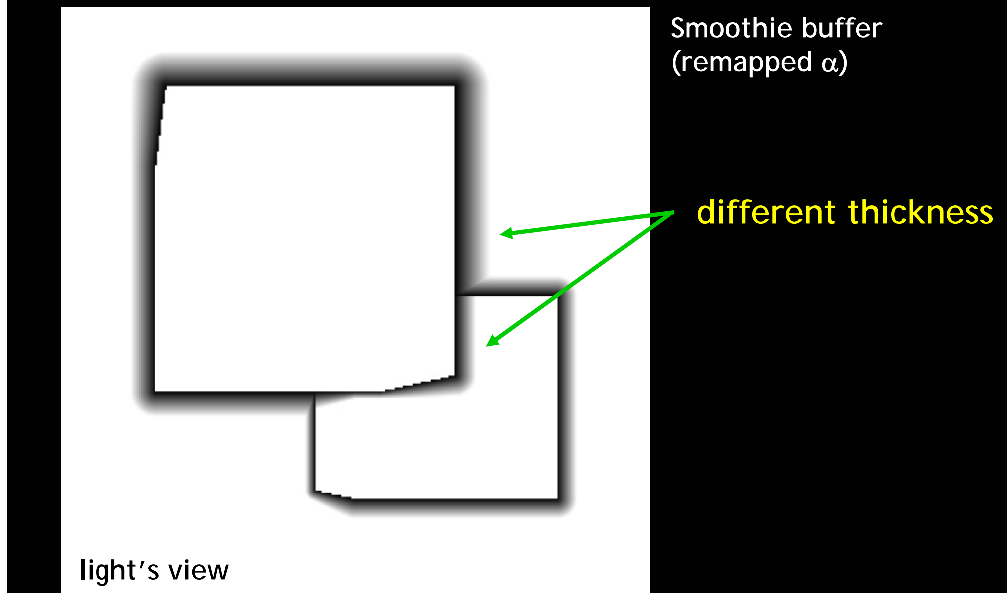


So far we have only considered simple examples, ones in which there is a single blocker and receiver. In more complex scenes, however, there are multiple blockers and receivers. We'll use this scene of two boxes and a floor to study how these blockers and receivers interact. For instance, in the image shown here, the box closer to the ground acts as both a blocker and a receiver. It receives the shadow from the first box and casts a shadow onto the ground. Notice how the shadows from the top box overlap with the shadows from the bottom box.

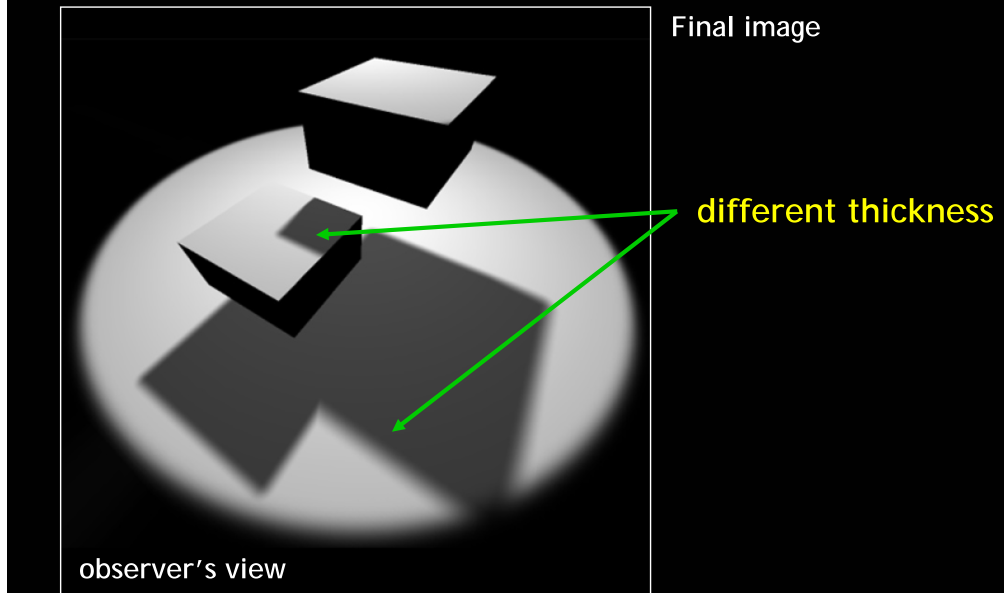
Multiple Receivers



Let's first consider the case of multiple receivers. In the image of the smoothie buffer shown here, we have neglected to remap the alpha values as suggested earlier. Instead, we have simply used the linearly-interpolated alpha. Notice that the top box, in principle, should cast a harder shadow on the bottom box and a softer shadow onto the ground plane (because the ground plane is farther away). Without the remapping of alpha, however, the smoothie alpha values have the same "thickness" in both places (indicated by green areas). This is qualitatively incorrect.



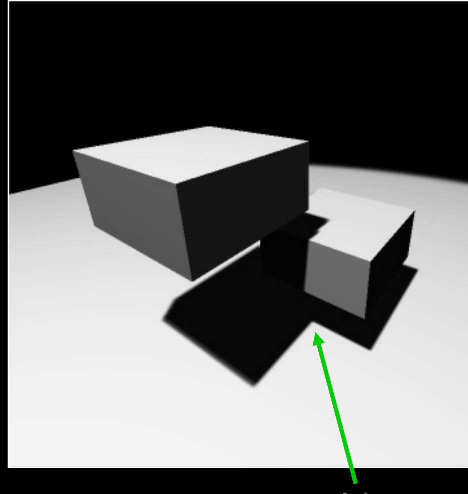
Now we have applied the remapping of alpha using the equation shown earlier. Notice how the thickness changes across the two receiving surfaces.



Here is the final image shown from the observer's viewpoint. As expected, the top box casts a sharper shadow on the lower box and softer shadow onto the ground plane.

Multiple Blockers

What happens when smoothies overlap?



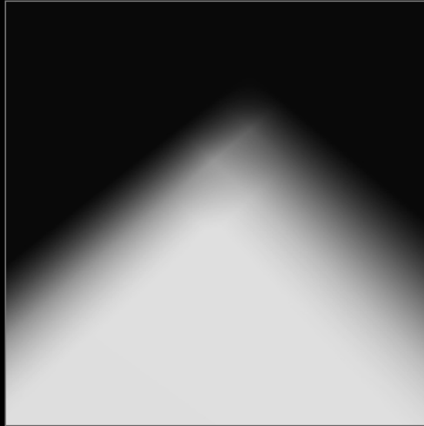
Now let's turn to the case of multiple blockers and understand how to handle overlapping shadows. The case of overlapping shadows means that multiple smoothies overlap in the screen space of the light source. Computing the smoothies' depth values is the same: just store the nearest depth value into the buffer. But how do we compute alpha values?

It turns out that this is a tricky issue, because smoothies are rendered independently of one another. If we were to try to determine the correct visiblity due to multiple overlapping blockers, it would be very costly because then we could no longer consider blockers independently of one another. So what do we do?

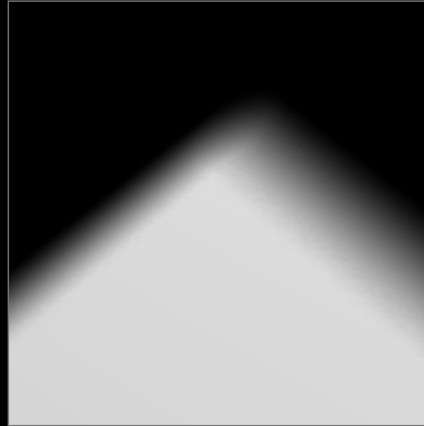
Multiple Blockers

SIGGRAPH2004

Minimum blending: just keep minimum of alpha values



smoothie



ray tracer

It turns out that a simple solution works reasonably well: minimum blending. In other words, when multiple smoothies overlap, compute alpha values independently for each, then apply minimum blending so that the darkest (minimum) alpha value is kept. The reason for doing so is that it avoids continuity problems when multiple smoothies overlap. The images here compare the minimum blending of two smoothies (left) versus the correct result obtained using a ray tracer.

Implementation

SIGGRAPH2004

We have finished covering the details of the algorithm. Now we can move on to understanding how we implement the smoothie algorithm in hardware.

Implementation

- Details (OpenGL)
- Hardware acceleration
- Optimizations

The smoothie algorithm can be implemented on DirectX 9-class hardware. This means specifically that you need to have programmable vertex and fragment units, and floating-point precision (at least 16 bits of floating-point) must be available in the programmable fragment unit. This precision is necessary for storing linear depth values, as we'll see in a moment. Examples of suitable hardware include the ATI R300 chips (e.g. Radeon 9700 and later) and the NVIDIA NV30 chips (e.g. GeForce FX and later).

The silhouette map algorithm can be implemented using both OpenGL and DirectX. However, any code snippets I show here will be in OpenGL.

Create Shadow Map

Render to standard OpenGL depth buffer

- 24-bit, window space
- Post-perspective, non-linear distribution of z

Also write to color buffer (using fragment program)

- Floating-point, eye space
- Pre-perspective, linear distribution of z
- Unlike regular shadow maps

Why? Need linear depth for next rendering pass

To create a shadow map, we place the OpenGL camera at the light position of the light source, aim it at the scene, and draw. Unlike rendering a regular shadow map, however, we have to perform some additional steps. In a standard depth map in OpenGL, depth values are stored in fixed point (usually with 24 bits of precision) in window space, i.e. after perspective projection has been applied. This causes z values to be non-linearly distributed.

However, we need to store linearly-distributed z values because the next pass (rendering of smoothies) will require such a z value for computing alpha properly. The easiest way to store linearly-distributed z values is to use a vertex shader to compute the depth of a vertex, which is just the z component of the vertex's position in eye-space. Then, using a fragment program, simply copy this eye-space z value to the color output. Note that the output color buffer needs to be a floating-point buffer (and at least 16 bits of precision). Otherwise, there will simply not be enough precision to handle a wide range of z values.

Create Smoothie Buffer

Conceptually, draw the smoothies once:

- store depth and alpha into a buffer

In practice, draw smoothies twice:

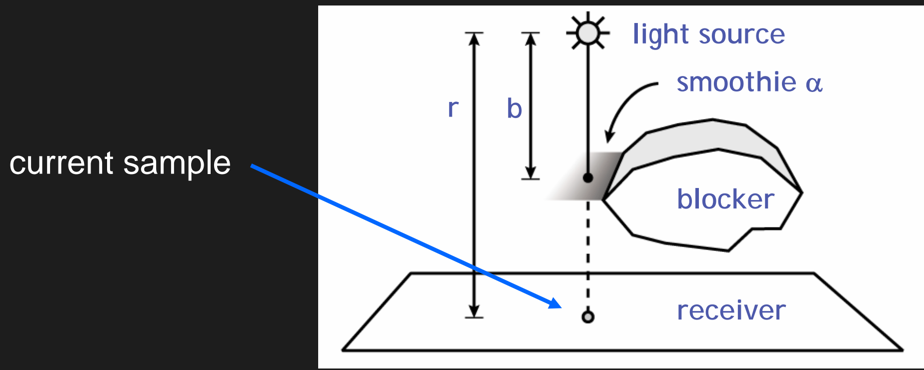
1. store nearest depth value into depth buffer
2. blend alpha values into color buffer

Next, we draw the smoothies twice, once to store the depth values, and once to store the alpha values (which we'll actually end up storing in the color buffer). The reason we cannot compute and store both quantities in the same pass is because we use minimum blending when rendering the smoothies' alpha, which is not compatible with using depth testing.

Computing Alpha

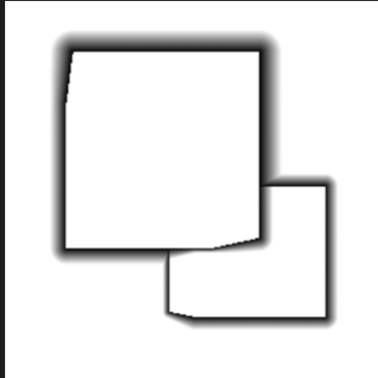
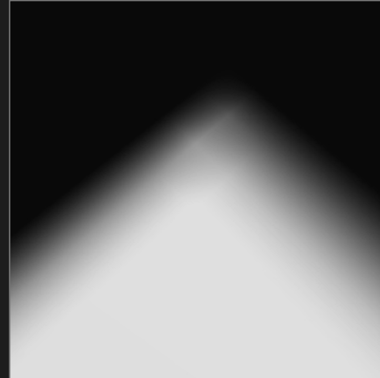
How to compute alpha? Recall $\alpha' = \alpha / (1 - b/r)$

- α is linearly interpolated from 0 to 1 across quad
- b is computed in fragment program
- r is obtained from shadow map (linear depth!)



We use a fragment program to perform the alpha computation. The distance b from the light to the smoothie is easy to compute. Since we are drawing the smoothies from the light's viewpoint, if a vertex has been transformed (i.e. in a vertex program) to eye space, then the z component is exactly the distance from the light to the smoothie. Now how do we compute r? The interesting point here is that, looking at the diagram, the surface lying immediately behind the smoothie is (by definition) the receiver, and we want to know the distance r to that surface. We have already computed this distance – it's stored in the depth map that we rendered in the first pass!

This is why we needed to store eye-space linear z values in the first pass. We now perform a texture lookup in that depth map, and the result is our distance r. The original alpha (no prime) can be computed in a vertex program or on the CPU. Now we have all the information we need to perform the alpha remapping, which can be done in a fragment program and written to the color output.

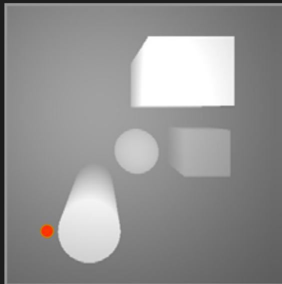



It is simple to implement minimum blending in OpenGL because it is supported natively in hardware. Just use the shown GL command.

Final Rendering Pass

Implementation using fragment program:

- Project each sample into light space
- Multiple texture lookups



shadow map
(depth)

smoothie buffer
(depth)

smoothie buffer
(alpha)

In the final rendering pass, we project each sample into light space (just as is done with ordinary shadow maps) to perform lookups into the shadow map, the smoothie (depth) buffer, and the smoothie (alpha) buffer. Then we can perform the necessary depth comparisons to shade the sample appropriately. Using the OpenGL ARB_shadow extension, the hardware actually does the depth comparisons for you, which leads to a big performance boost.

Additional Details

Combination of methods:

- percentage closer filtering (2 x 2 filtering in shader)
- perspective shadow maps

See paper (course notes) for Cg shader code

There are a number of ways to extend the smoothie algorithm. For instance, in the final rendering pass, instead of performing a single lookup into each texture map to shade the sample, we can perform the lookups multiple times using neighboring texels (e.g. the neighboring 2x2 grid), perform the shading using each set of samples, and then filter the results. This is similar to percentage closer filtering.

Our method can also be combined with perspective shadow maps to further reduce aliasing.

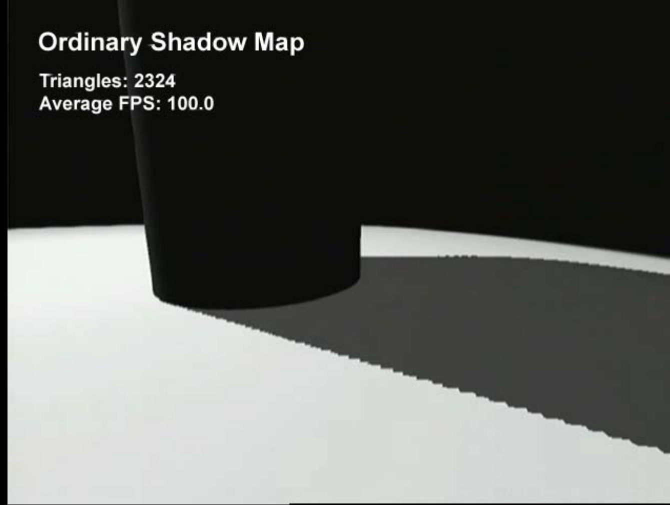
The original paper (E. Chan and F. Durand, EGSR 2003) is provided in the course notes and contains sample Cg shader source code for each of the rendering passes.

SIGGRAPH2004

Examples

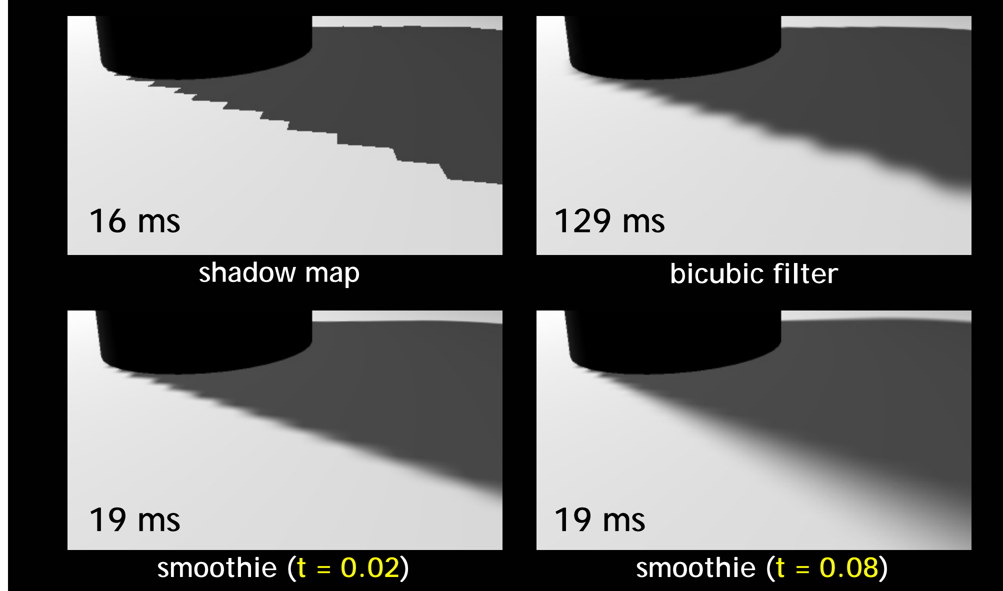
Video

SIGGRAPH2004



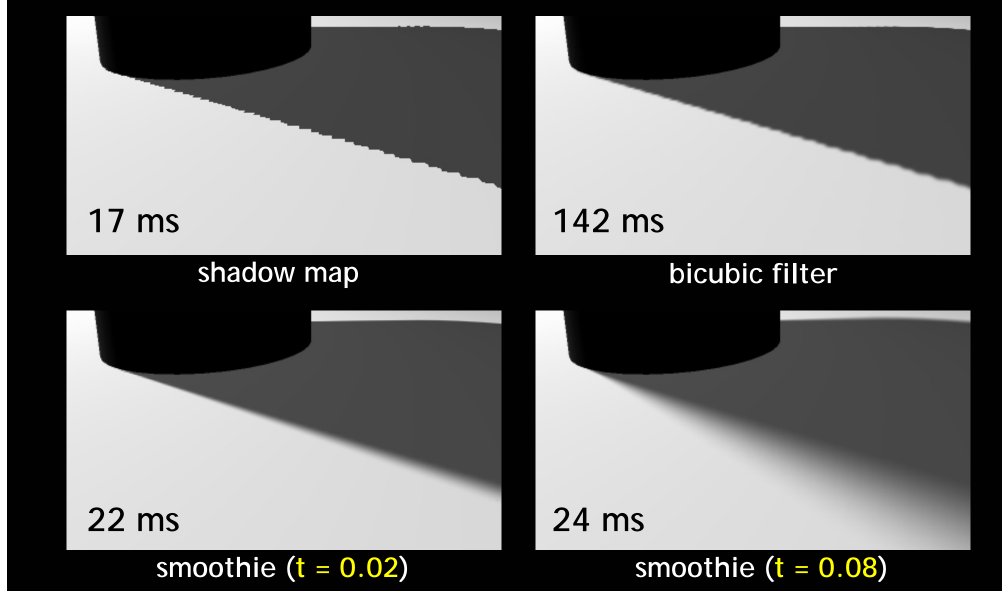
This video, which compares regular shadow maps, bicubic-filtered shadow maps, and the smoothie algorithm, is not very useful in these course notes, but fortunately it is available on the paper web site: http://graphics.csail.mit.edu/~ericchan/papers/smoothie/

Hiding Aliasing (256 x 256)

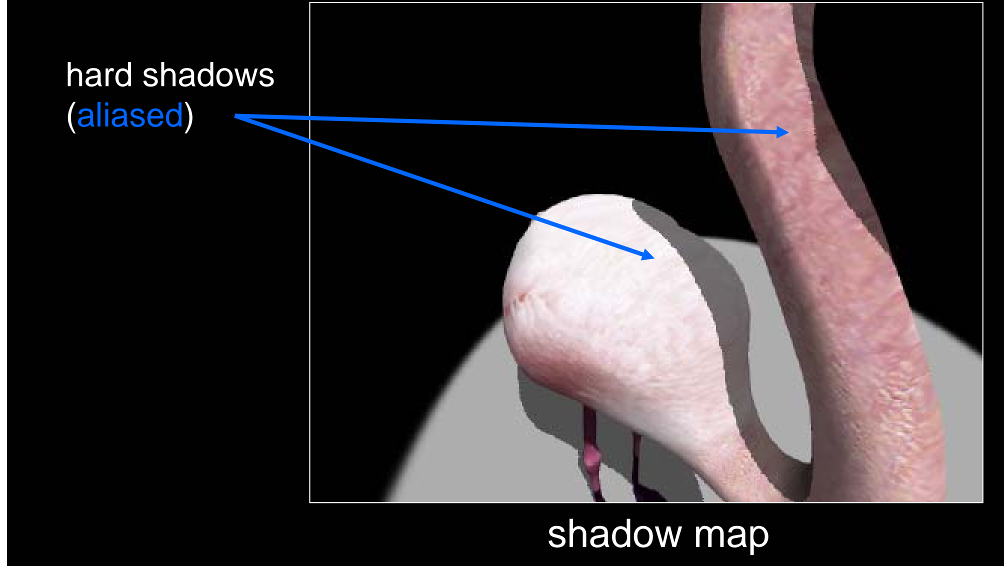


I mentioned earlier that the smoothie algorithm is useful for hiding aliasing artifacts. Let's consider an extreme case: using low-resolution (256 x 256) buffers. We're looking a simple scene where a cylinder casts a shadow onto a ground plane. The top-left image shows the extremely aliased shadow edge from a regular shadow map. In the top-right image, we have applied percentage closer filtering with a bicubic reconstruction filter (16 samples). The shadows are somewhat smoother, but the rendering time has increased considerably. The bottom of row images were generated using the smoothie algorithm. The bottom-right image simulates a larger area light source, which equates to using larger smoothies. Although the aliasing artifacts are still apparent, they are less objectionable than in the top row of images, and the performance remains high.

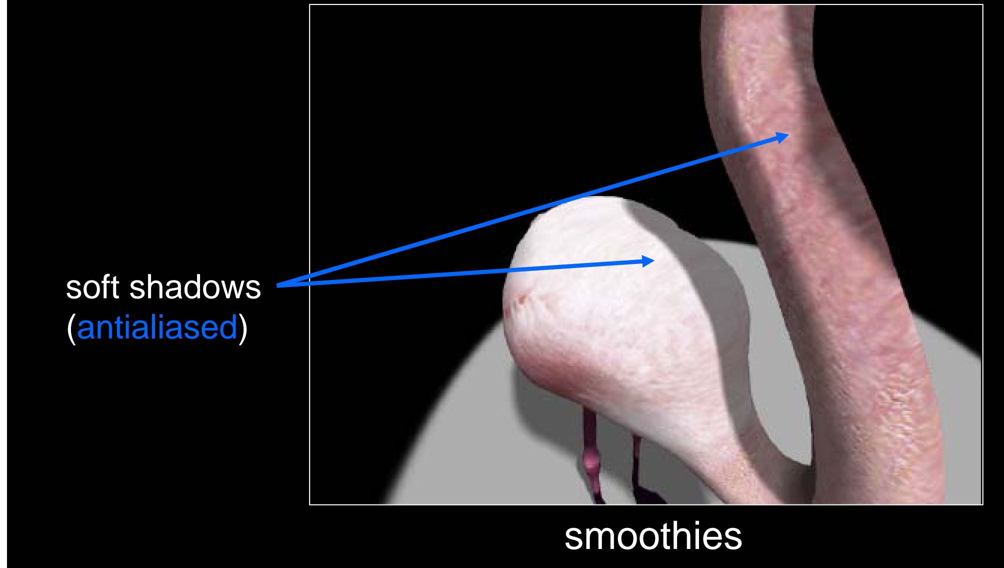
Hiding Aliasing (1k x 1k)



This is the same scene, except using 1024 x 1024 buffers. Whereas aliasing is still noticeable in the top of images, it is completely masked in the images generated using the smoothie algorithm. Furthermore, the bottom-row image looks convincingly like a soft shadow, with the shadow getting softer farther away from the cylinder.



Here are some additional examples with more complex blockers and receivers. We are looking at the shadow cast by a flamingo's neck onto its body. This image was created using a standard 1024 x 1024 shadow map.

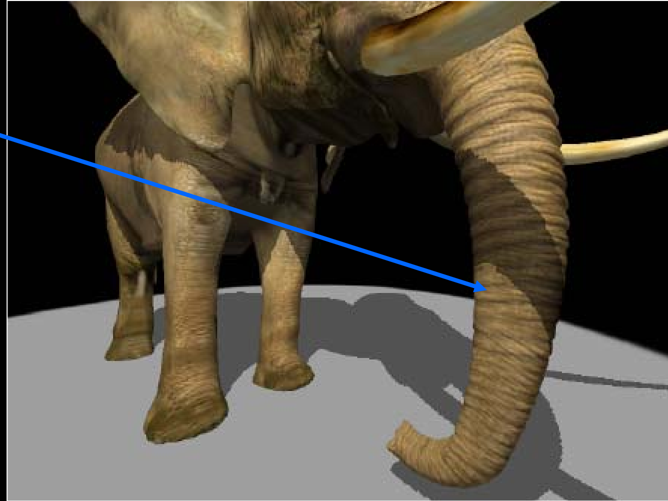


This image was generated using the smoothie algorithm. The shadow edges appear softer and antialiased.

Antialiasing Example #2

SIGGRAPH2004

hard shadows (aliased)



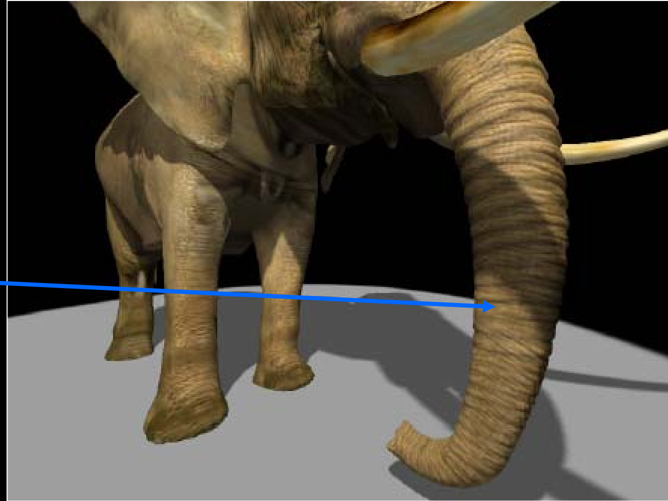
shadow map

In this image the elephant's tusk casts a shadow onto its trunk. Again we are using a 1024 x 1024 shadow map.

Antialiasing Example #2

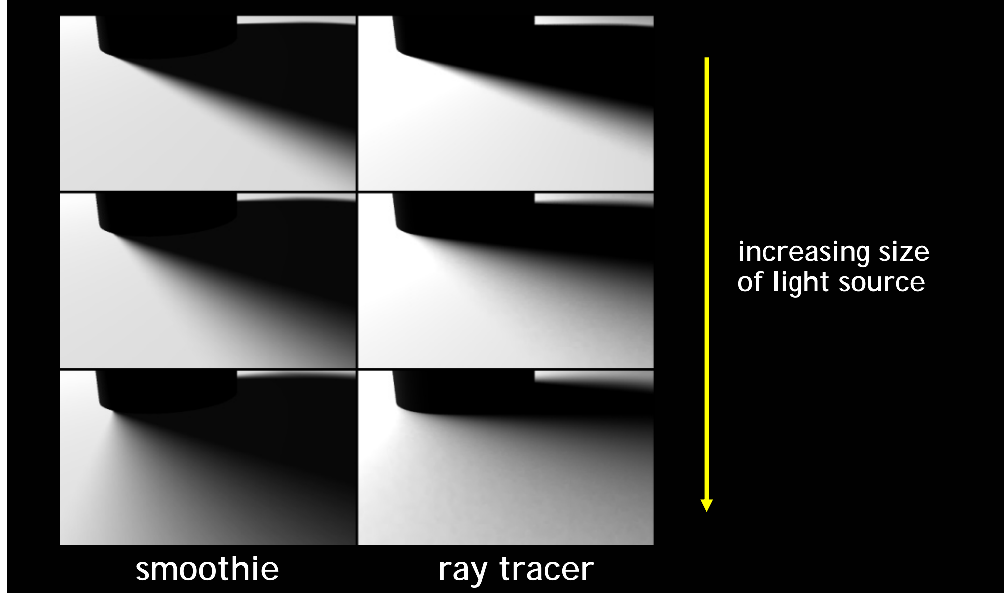
SIGGRAPH2004

soft shadows
(antialiased)



smoothies

The shadows using the smoothie algorithm are soft and antialiased.



Now let's consider what happens as we increase the size of the light source (i.e. make the smoothies bigger). As we mentioned earlier, this is actually a limitation of our approach, since we aren't accurately modeling the area light source (in fact, we aren't really modeling it at all). These images compare the smoothie algorithm against a Monte Carlo ray tracer. As can be seen in the left column, as we increase the size of the light source, the shadow does indeed get softer, but the umbra always has the same size. In contrast, the ray-traced images show the correct result, i.e. the umbra decreases as the size of the light source increases.

Video

SIGGRAPH2004

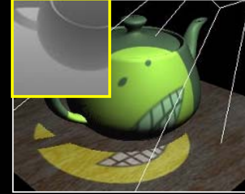
original md2shader demo courtesy of Mark Kilgard

This video is also available on the paper web site:
http://graphics.csail.mit.edu/~ericchan/papers/smoothie/

Tradeoffs

Shadow maps:

- Assumes directional light or spotlight
- Discrete buffer samples



Now let's discuss qualitatively the tradeoffs involved in the smoothie algorithm. Since the method works in both image space (i.e. the use of a discrete buffer for the shadow map and smoothie buffer) and object space (the use of object-space silhouettes to construct the smoothies), the method inherits some the limitations from both sets of techniques.

Like the shadow map method, the smoothie algorithm assumes some form of directional light. Covering the entire sphere of directions requires additional rendering passes. In contrast, shadow volumes automatically handle omnidirectional point light sources as well as directional light sources.

Tradeoffs

SIGGRAPH2004

Shadow maps:

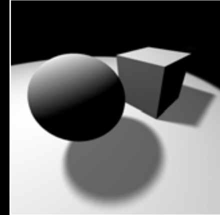
- Assumes directional light or spotlight
- Discrete buffer samples

Shadow volumes:

- Assumes blockers are closed triangle meshes
- Silhouettes identified in object space

As with the shadow volume method, the smoothie algorithm requires finding object-space silhouettes. This implies that our blockes must be represented as polygons. In contrast, shadow maps automatically handle any type of geometry that can be represented in a depth buffer.

Tradeoffs



Shadow maps:
- Assumes directional light or spotlight
- Discrete buffer samples

Shadow volumes:
- Assumes blockers are closed triangle meshes
- Silhouettes identified in object space

Smoothies:
- Rendered from light's viewpoint
- Occupy small screen area → inexpensive

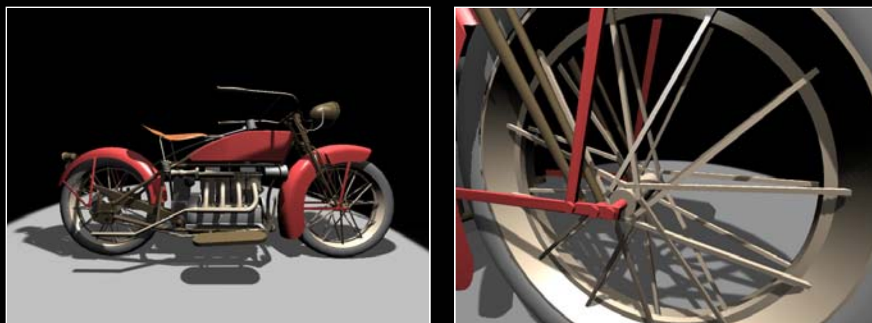
One might wonder about the expense of rendering the smoothies themselves. After all, it sounds similar to rasterizing shadow volumes, which is known to be a costly operation. The main difference here is that shadow volume polygons are drawn from the observer's viewpoint. They occupy substantial screen area and thus are expensive to rasterize. On the other hand, smoothies are drawn from the light's viewpoint. They occupy relatively little screen area and thus are cheaper to render.

Summary

SIGGRAPH2004

Main points:

- Simple extension to shadow maps
- Shadows edges are fake, but look like soft shadows
- Fast, maps well to graphics hardware



In summary, the smoothie algorithm tries to balance the quality and performance goals of real-time soft shadow algorithms. While it is not geometrically accurate, it captures an important aspect of soft shadows, namely the dependence on the ratio of distances between the light source, blocker, and receiver. Since the umbra of the shadow does not decrease with increasing light source size, the algorithm is best suited to small area light sources. Furthermore, the algorithm is useful for the antialiasing of shadow edges, regardless of whether or not soft shadows are desired.

Acknowledgments

Hardware, drivers, and bug fixes

- Mark Kilgard, Cass Everitt, David Kirk, Matt Papakipos (NVIDIA)
- Michael Doggett, Evan Hart, James Percy (ATI)

Writing and code

- Sylvain Lefebvre, George Drettakis, Janet Chen, Bill Mark
- Xavier Décoret, Henrik Wann Jensen

Funding

- ASEE NDSEG Fellowship

SIGGRAPH2004