

Siggraph 2004

Full-Day Course #26

“Real-Time Shadowing Techniques”

Course Title

Real-Time Shadowing Techniques

Category

Rendering – Real-Time or for Games

Presentation Venue

Session Room

Summary Statement

How to incorporate shadows in real-time rendering. Basic shadowing techniques, more advanced techniques that exploit new features of graphics hardware, the differences among these algorithms, and their strengths and weaknesses. The course includes implementation details.

Names of all lecturers in the format

Tomas Akenine-Moeller,
Lund Institute of Technology,
tam@cs.lth.se

Eric Chan,
Massachusetts Institute of Technology,
ericchan@graphics.csail.mit.edu

Wolfgang Heidrich,
University of British Columbia,
heidrich@cs.ubc.ca

Jan Kautz,
Massachusetts Institute of Technology,
kautz@graphics.csail.mit.edu

Mark Kilgard,
NVIDIA,
mjk@nvidia.com

Marc Stamminger,
University of Erlangen-Nuremberg,
stamminger@cs.fau.de

Course abstract

Shadows heighten realism and provide important visual cues about the spatial relationships between objects. But integration of robust shadow shadowing techniques in real-time rendering is not an easy task. In this course on how shadows are incorporated in real-time rendering, attendees learn basic shadowing techniques and more advanced techniques that exploit new features of graphics hardware.

The course begins with shadowing techniques using shadow maps. After an introduction to shadow maps and general improvements of this technique (filtering, depth bias, omnidirectional lights, etc.), the first section describes two methods for reducing sampling artifacts: perspective shadow maps and silhouette maps. Both techniques can significantly improve shadow quality, but they require careful implementation. The course continues with extensions of the shadow mapping method that allow soft shadows from linear and area light sources. The second part of the course discusses recent advances in efficient and robust implementation of shadow volumes on graphics hardware and then shows how shadow volumes can be extended to generate accurate soft shadows from area lights. Finally, the course summarizes real-time shadowing from full lighting environments using the technique of precomputed radiance transfer.

The course explains the differences among these algorithms and their strengths and weaknesses. Implementation details, often omitted in technical papers, are provided. And throughout the course, the tradeoffs between quality and performance are illustrated for the different techniques.

Prerequisites

Working knowledge of a low-level graphics API such as DirectX or OpenGL. Some knowledge of shadowing algorithms is useful, but not required.

Level of difficulty

Intermediate

Intended audience

Everybody who is interested in realtime and interactive graphics.

Syllabus

8:30 - 8:45 **Introduction** (Jan Kautz)

- * Why Shadows?
- * Definitions
- * Problem Statement
- * Classification of Algorithms:
 - Hacks (no shadow, projected blob, etc.)
 - Shadows using Shadow Maps (point lights and linear lights)
 - Shadows using Shadow Volumes (point lights and area lights)
 - Shadows using Radiance Transfer (full lighting environments)

Component I: Shadow Maps

8:45 - 9:15 **Shadow Maps** (Marc Stamminger)

- * Introduction to hardware-accelerated Shadow Mapping
- * Robustness problems
- * Optimizing the shadow map
 - Tight fitting light source frustum
- * Optimizing the shadow test
 - Improving the depth precision
 - 2nd depth shadow maps
 - Linear-spaced depth values

9:15 - 10:00 **Perspective Shadow Maps** (Marc Stamminger)

- * Concept
- * Sample density considerations
- * Implementation details
- * Examples, results, and limitations

10:00 - 10:15 **Silhouette Maps - Part I** (Eric Chan)

- * Problems with shadow maps (and shadow volumes)
- * Key idea: only shadow silhouettes need precise reconstruction

Break

10:30 - 10:55 **Silhouette Maps - Part II** (Eric Chan)

- * Approximate edge data structure: the silhouette map
- * Hardware implementation details
- * Examples, results, and limitations

10:55 - 11:35 **Linear Light Sources** (Wolfgang Heidrich)

- * Why soft shadows?
- * Linear light sources
- * Visibility from line segments
- * Adaptation of the shadow map algorithm

11:35 - 12:15 **Smoothies** (Eric Chan)

- * Fake soft shadows: a qualitative / phenomenological approach
- * Observation: appearance of soft shadows depends mostly on ratio of distances between blocker and receiver
- * Key idea: extra geometry (a.k.a. "smoothies") and projective texture mapping
- * Hardware implementation details
- * Examples, results, and limitations
- * Comparison of all presented shadow mapping methods

Lunch

Component II: Shadow Volumes

13:45 - 14:45 **Shadow Volumes** (Mark Kilgard)

- * Shadow volume introduction
- * Implementation with the stencil buffer
- * Robustness problems
- * Improvements:
 - Infinite view frusta
 - Z-fail test
 - Depth clamp
- * Comparison: shadow volume vs. shadow map algorithms

14:45-15:30 **Soft Shadow Volumes - Part I** (Tomas Akenine-Moeller)

- * Why enhance the shadow volume algorithm?
- * The penumbra wedge primitive
- * Robust penumbra wedge construction
- * Visibility computation inside wedge
- * Implementation using programmable hardware

Break

15:45-16:00 **Soft Shadow Volumes - Part II** (Tomas Akenine-Moeller)

- * Hardware optimizations
- * Comparison: Soft Shadow Volumes, Smoothies, and Linear Light Sources

Component III: Radiance Transfer Shadows

16:00-17:00 **Radiance Transfer with Shadows** (Jan Kautz)

- * Introduction to precomputed radiance transfer (PRT)
- * Application to shadows
- * Extension to animated models
 - Precomputation + compression
 - On-the-fly computation
- * Comparison: PRT shadows vs. area light shadows

17:00-17:30 **Conclusions** (Jan Kautz)

- * Final comparison between all algorithms
- * Compare: quality, application to what kind of lighting, speed, usability, limitations, and artifacts.
- * Questions & answers (all)

Course presenter information for the organizer and each lecturer

Tomas Akenine-Moeller, Lund Institute of Technology, Sweden

Tomas is an associate professor at Lund Institute of Technology, with research focus on real-time computer graphics algorithms, in particular shadow algorithms and graphics for mobile platforms. He is a co-author of the book "Real-Time Rendering" with Eric Haines, and he has published many papers at international conferences. In 2000, he was a postdoc at UC Berkely, and that year he also taught a course at SIGGRAPH. Last year he finally managed to get the first (ever) Swedish paper(s) accepted at SIGGRAPH, and so he might retire any day now.

Eric Chan, Massachusetts Institute of Technology, Cambridge, USA

Eric Chan is a 2nd-year Ph.D. student at the Massachusetts Institute of Technology, where he studies computer graphics with Frédo Durand. His current research interests includes graphics architectures, shadow algorithms, antialiasing, and digital photography. Before attending graduate school, Eric was a research staff member in the Computer Graphics Laboratory at Stanford University. He worked on the Real-Time Programmable Shading project under the supervision of Pat Hanrahan and Bill Mark. He enjoys digital photography and spends an unreasonable amount of his free time playing with cats.

Wolfgang Heidrich, University of British Columbia, Canada

Wolfgang Heidrich holds an Assistant Professor position in Computer Science at the University of British Columbia. He received a PhD in Computer Science from the University of Erlangen in 1999, and then worked as a Research Associate at the Computer Graphics Group of the Max-Planck-Institut für Informatik in Saarbrücken, Germany, where he lead the research efforts in hardware-accelerated and image-based rendering. Heidrich's research interests include computer graphics and computer vision, in particular image-based modeling, measuring, and rendering, geometry acquisition, hardware-accelerated and image-based rendering, and global illumination. Heidrich has written over 50 refereed publications on these subjects and has served on numerous program committees. He is a co-author of the recently published book "Real-Time Shading".

Jan Kautz, Massachusetts Institute of Technology, Cambridge, USA

Jan is currently a Post-Doc with the graphics group at the Massachusetts Institute of Technology. He received his PhD from the Max-Planck-Institut für Informatik, Saarbrücken, Germany. His thesis was on real-time shading and rendering. He is particularly interested in the realistic shading using graphics hardware, about which he has published several articles at various conferences including SIGGRAPH. Jan has been a tutorial speaker on real-time shading at Eurographics, Web3D, and others.

Mark J. Kilgard, NVIDIA Corporation, Santa Clara, USA

Mark J. Kilgard is a Graphics Software Engineer at NVIDIA Corporation where he contributes to the NVIDIA OpenGL driver. Mark is particularly interested in providing better interfaces to today's programmable graphics hardware. Mark authored the book "Programming OpenGL for the X Window System" and implemented the popular OpenGL Utility Toolkit (GLUT) for developing portable OpenGL examples and demos. Previously, Mark worked at Silicon Graphics on the Onyx InfiniteReality graphics supercomputer and on the SGI's X Window System implementation. Mark has taught many courses at SIGGRAPH, the Computer Game Developers Conference, and other conferences. Mark's Karaoke rendition of Dolly Parton's "9 to 5" can't be beat.

Marc Stamminger, University of Erlangen-Nürnberg, Germany

Marc Stamminger is professor at the University of Erlangen, Germany. He wrote a PhD thesis on radiosity methods, but his recent research focuses on interactive rendering techniques in the virtual reality context, in particular on point based rendering and interactive shadow generation. In this area, he published several papers at international conferences including SIGGRAPH, and has been tutorial speaker at Eurographics and other conferences.

Organizer Contact Information

Jan Kautz
Computer Graphics Group - CSAIL
Massachusetts Institute of Technology
77 Massachusetts Avenue, 32-D530
Cambridge, MA, 02139
USA
Phone: +1 617 253 8871
EMail: kautz@graphics.csail.mit.edu

Marc Stamminger
Informatik 9
Friedrich-Alexander-Universität Erlangen-Nürnberg
Am Weichselgarten 9
91058 Erlangen
Germany
Phone: +49 9131 8529920
E-Mail: stamminger@cs.fau.de



Real-Time Shadowing Techniques

Course #26, Tuesday, Full Day

Schedule



- **Introduction**
 - 8:30 Introduction (Kautz)
- **Shadow Mapping**
 - 8:45 Introduction to Shadow Maps (Stamminger)
 - 9:15 Perspective Shadow Maps (Stamminger)
 - 10:00 Silhouette Maps (Chan)
 - 10:15 – 10:30 Break
 - 10:55 Linear Light Sources (Heidrich)
 - 11:35 Smoothies (Chan)
 - 12:15 – 13:45 Break
- **Shadow Volumes**
 - 13:45 Shadow Volumes (Kilgard)
 - 14:45 Soft Shadow Volumes (Akenine-Moeller)
 - 15:30 – 15:45 Break
- **Radiance Transfer**
 - 16:00 Radiance Transfer with Shadows (Kautz)
- **Conclusions**
 - 17:00 Conclusions (Kautz)



SIGGRAPH2004

Shadow Mapping

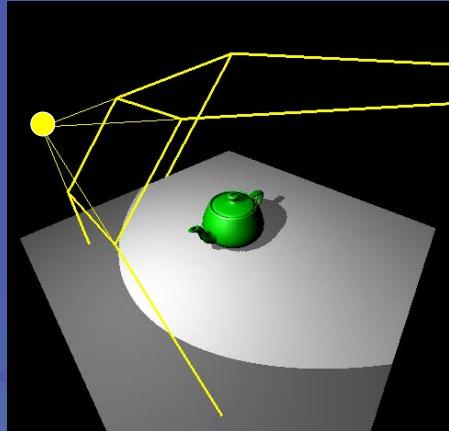
Marc Stamminger, University of Erlangen-Nuremberg

Idea

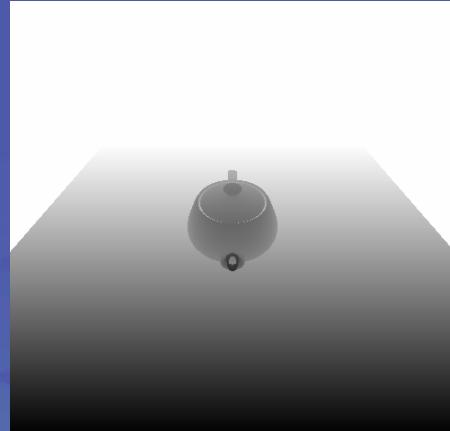
- assumption:
spot light with spot angle $\ll 180^\circ$
- render scene from the view of the light
 - camera position = light position
 - camera direction = spot direction
 - opening angle = spot angle
 - light frustum
- shadow map = resulting depth buffer

Mostly, shadow maps are generated for spot lights. All scene points illuminated by such a spot light are also visible by a perspective camera in the light source. We thus render the scene from the view of the light source and store the depth buffer as „shadow map“.

Idea



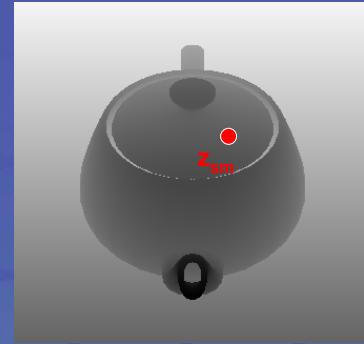
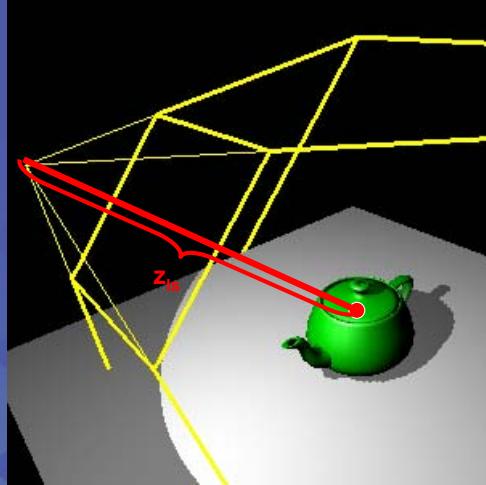
observer's view with light frustum



shadow map

On the left one can see a simple scene, illuminated by a light source from the left. The corresponding shadow map is on the right.

Example

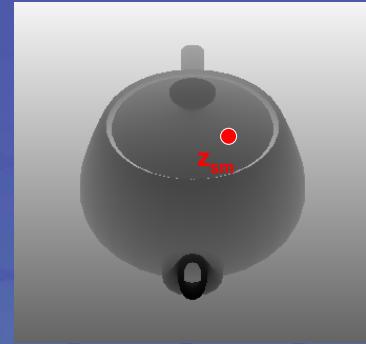
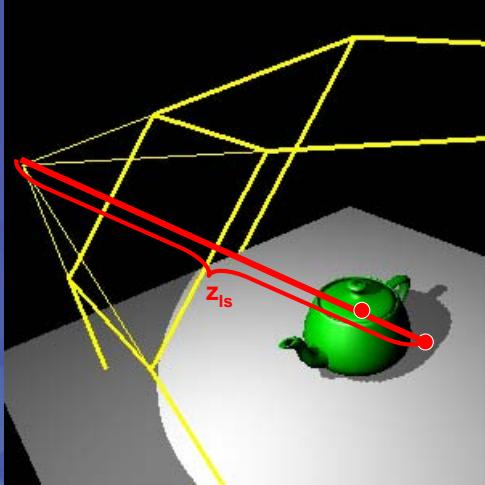


$$z_{ls} = z_{sm} \rightarrow \text{point lit}$$

In order to determine whether a point is in shadow, we compute its distance to the light source z_{ls} and compare it with the corresponding depth value z_{sm} in the shadow map. If $z_{ls} = z_{sm}$, the point is visible from the light and thus lit.

Example

SIGGRAPH2004



$z_{ls} > z_{sm} \rightarrow \text{point in shadow}$

Another point behind the teapot has a larger distance to the light z_{ls} . Its projection in the shadow map is at the same position, but this time $z_{ls} > z_{sm}$. Thus the point is in shadow.

Idea



- per pixel (x, y) of the camera view
 - point in camera space is $(x, y, \text{depth}(x, y))$
 - map back to world space
 - map to light source space $\rightarrow (x_{ls}, y_{ls}, z_{ls})$
 - compare z_{ls} with shadow map value at (x_{ls}, y_{ls})

This is what we have to do per pixel more precisely.

Idea

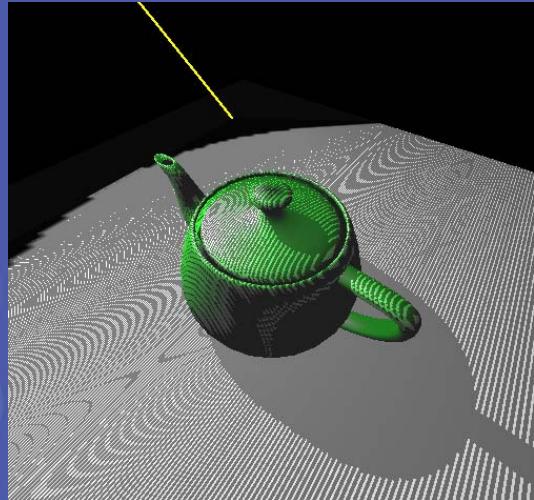


- demo „shadowmaps“

Bias



- bias needed
 - if point is lit
 $z_{ls} \approx z_{sm}$
 - if we test for
 $z_{ls} > z_{sm}$,
also lit points
are shadowed
- test for
 $z_{ls} > z_{sm} + \text{bias}$

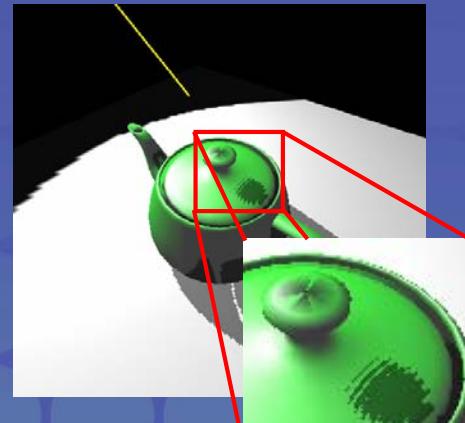
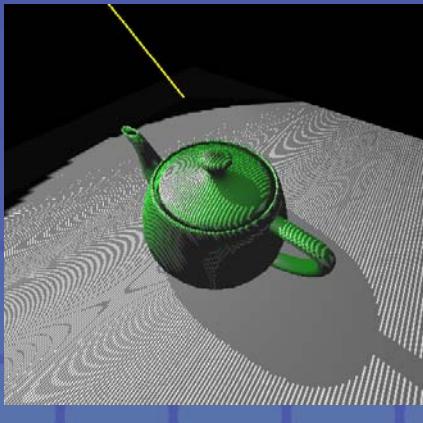


The „lit“ condition $z_{ls} = z_{sm}$ is usually not fulfilled for lit points, because of numerical inaccuracies. So we have to define a corridor around z_{sm} and classify all points as lit, as long as $|z_{ls} - z_{sm}| < \text{epsilon}$. Because the case $z_{ls} < z_{sm}$ cannot appear (except for numerical reasons), we can replace the test by $z_{ls} > z_{sm} + \text{bias}$.

Bias



- bias too small → surface acne
- bias too large → shadow leaks



The bias has to be chosen carefully. Too small bias results in surface acne, where surfaces shadow themselves. If the bias is too big, shadows get lost if the occluder and shadow receiver are close.

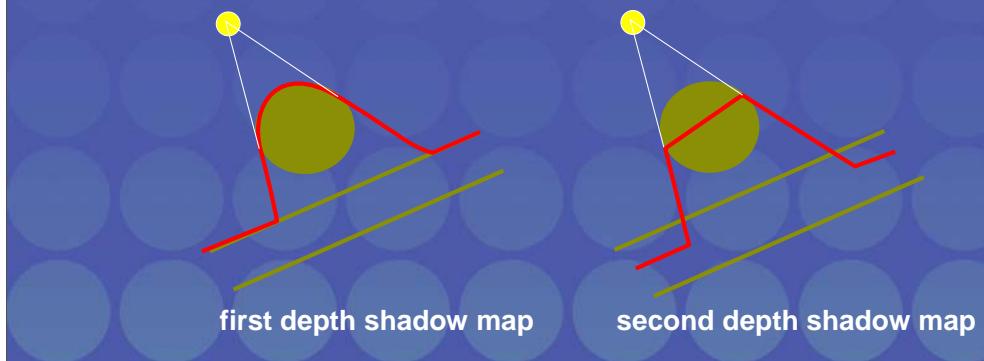
Bias

- better solution:
use PolygonOffset when rendering the
shadow map
→ offset is adapted to surface slope

polygon offset as supported by normal hardware is an even better means for biasing,
because it also considers surface slope.

Bias

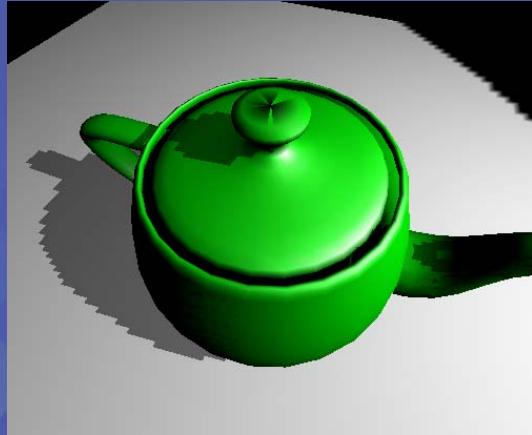
- even more robust:
Second Depth Shadow Maps
 - store in shadow map average depth of first and second visible surface



Second depth shadow maps use the average between first and second visible surface from the view of the light source. By this, the shadow map depth values have maximum distance to the lit and the first shadowed surfaces. On the downside, the generation is more expensive and requires to generate the first and second visible layer from the view of the light source.

Aliasing

- finite shadow map resolution
→ pixelized shadows

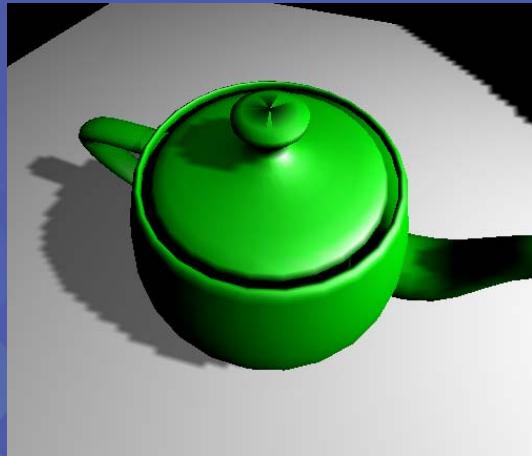


The major problem of shadow maps is aliasing. Because of the image based nature of the shadow map, shadows appear pixelized, when shadow map and camera image resolution mismatch.

Aliasing



- percentage closer filtering
 - filter shadow test result 0/1 instead of depth



When filtering shadow maps, it is important not to filter the depth values, but the comparison results.

Aliasing

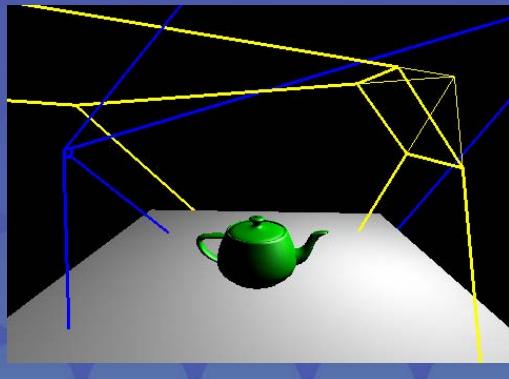
- bad aliasing cases
 - large scenes
→ high shadow map resolution required
 - close-ups to shadow boundaries
 - shadow stretches along receiver

Aliasing is particularly apparent for several cases. In particular for large outdoor scenes, standard shadow maps are almost useless. Furthermore, when we zoom into a shadow boundary, we can always recognize the pixelized shadow structure at some point.

Aliasing



- duelling frusta
 - light shines opposite to viewing direction

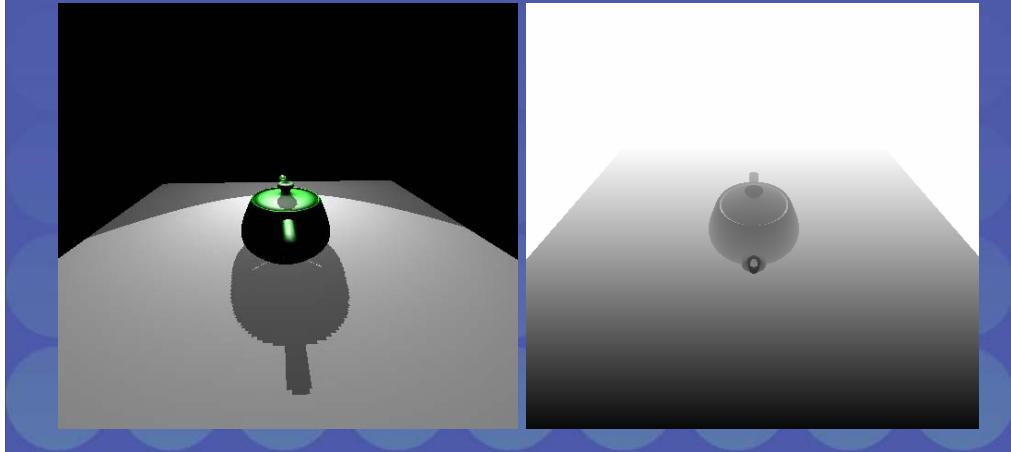


A typical bad case are duelling frusta.

Aliasing



- duelling frusta
 - resolution mismatch

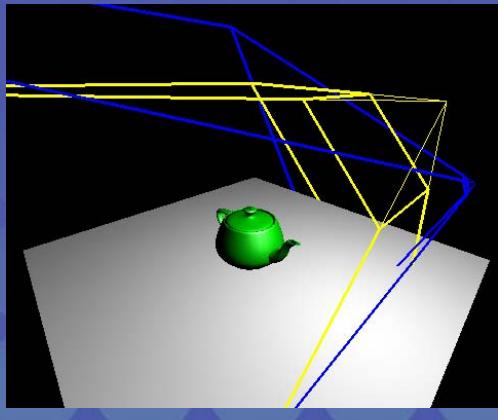


For duelling frusta, the resolution mismatch is maximal. The regions that are large in the camera view, are small in the shadow map and vice versa. As a result, the shadow of the handled is clearly pixelized.

Aliasing



- best case: miner's headlamp
 - light position close to camera

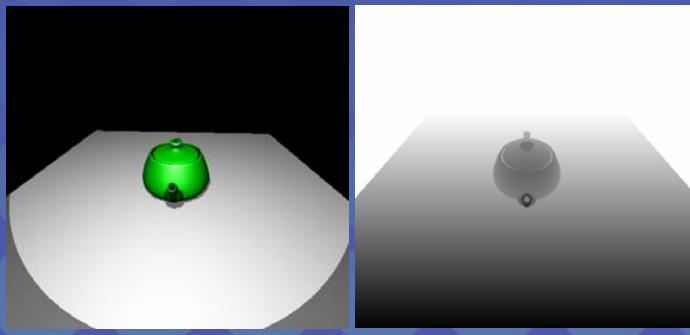


The optimal case appears if camera and light source are at similar positions.

Aliasing



- miner's headlamp
 - similar frusta
 - similar sampling densities
 - „one shadow map pixel per image pixel“



In this case, the resolutions are similarly distributed over the scene, and we get a one-to-one match between shadow map and image pixels.

Hardware Support

- OpenGL 1.4
 - **GL_ARB_DEPTH_TEXTURE**
internal texture format to store depth values
 - **glTexGen**
generation of light space coordinates as
texture coordinates
 - **GL_ARB_SHADOW**
special texture mode:
`return texture(s,t) < r ? Black : White;`

The big advantage of shadow maps is that they are fully supported by hardware. In order to apply them efficiently, two extensions are necessary, which became part of OpenGL 1.4, as well as the glTexGen command.



Hardware Support

- **P-Buffers**
 - offscreen rendering of shadow map
 - large shadow maps sizes
- **GL_ARB_RENDER_TEXTURE and WGL_NV_RENDER_TEXTURE**
 - bind P-Buffer depth buffer as depth texture
 - no copy needed

Further improvements are possible using P-Buffers and the render to texture functionality.



Hardware Support

- Register Combiners / Fragment Programs
 - for shadow application
 - if point is in shadow:
 - leave ambient light unchanged
 - ambient light has no origin, thus cannot be shadowed
 - dim diffuse light
 - some shading remains to show surface orientation
 - remove specular light
 - no highlights in shadow

Register combiners or fragment programs can be used to apply the shadows.

Pros and Cons

- + general
 - everything that can be rendered can cast and receive a shadow
 - works together with vertex programs
- + fast
 - full hardware support
 - (almost) no overhead for static scenes
 - two passes needed for dynamic scenes
- + robust
- + easy to implement
- aliasing

Extensions

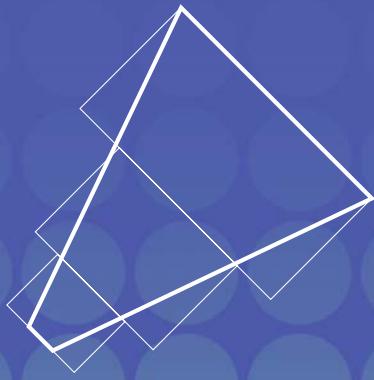


- concentrating on visible part [Brabec02]
 - restrict shadow map to visible part of scene
 - increased shadow map resolution
 - less aliasing
 - shadow map view dependent
 - two rendering passes needed, also for static scenes

Brabec describes an efficient method to restrict the shadow map to these parts of the scene that are visible. No shadow map resolution is wasted for invisible objects. However, by this the shadow map becomes view dependent and must be regenerated per frame, which needs to be done for dynamic scenes anyhow.

Extensions

- outdoor scenes [Tadamura01]
 - multiple shadow maps along view frustum
 - not interactive



Tadamura use multiple depth buffers to cover the trapezoidal view frustum. The method is not suited for interactive rendering.

Extensions



- Adaptive Shadow Maps [Fernando01]
 - shadow map as hierarchical quadtree
 - generate only required part of tree
 - many rendering passes needed
 - evaluation of shadow map in software

Adaptive Shadow Maps store a quadtree representation of a very high resolution shadow map. Only part of the quadtree is stored that is needed for the current view. For every frame it is determined, whether a node of the hierarchy is still sufficient or needs to be replaced by higher resolution children. The approach requires several rendering passes per frame, and the evaluation of the shadow map is not possible in hardware (yet?).

Extensions



- Perspective Shadow Maps
[Stamminger02]
 - shadow map sees world after perspective transformation
 - reduced aliasing
 - very large scenes possible
 - see later

Perspective shadow maps exploit a remaining degree of freedom in the generation of shadow maps. When rendering the scene from the view of the light source, previously symmetric frusta have been used. However, by using asymmetric frusta, aliasing effects can be largely reduced in several cases. Other cases, however, remain critical and prone to aliasing. We will discuss perspective shadow maps in detail in the next talk.

Extensions



- omni-directional light sources
 - use cubical shadow maps
 - six passes needed to generate shadow map
- dual-paraboloid shadow maps [Brabec02]
 - use two parabolic shadow maps for omnidirectional light sources
 - only two passes needed to generate shadow map
 - non-projective mapping
→ fine tessellation assumed

As mentioned in the beginning, shadow maps are usually generated for spot lights, because their illumination solid angle can be covered by a perspective camera frustum. For omnidirectional lights, multiple shadow maps or non-projective mappings must be used.

Extensions

- Soft Shadows from shadow maps
 - [Agrawala00]
 - [Brabec00]
 - ...
 - see later lesson

Shadow maps can also be used to generate soft shadows. However, this requires rather involved additional computations, and the performance penalty is significant. Later, we will describe one such approach in detail [Brabec00]



SIGGRAPH2004

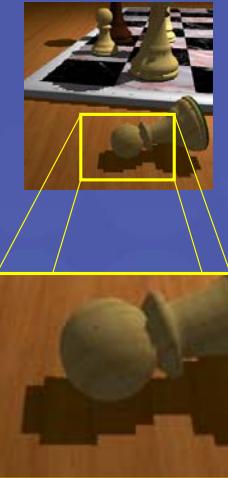
Perspective Shadow Maps

Marc Stamminger, University of Erlangen-Nuremberg

When compiling these slides, there were a couple of novel papers about perspective shadow maps under review. Of course, we cannot describe these now, but when this course will be given, some of these novel approaches will probably have been published and can be covered.

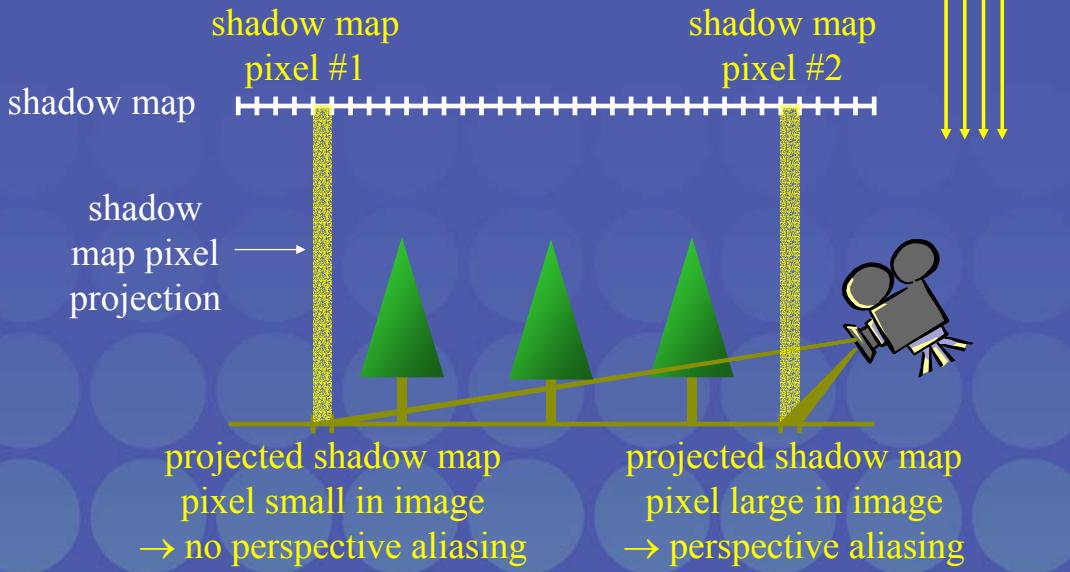
Shadow Map Aliasing

- in particular for large scenes
- shadow maps impossible for infinite scenes



Shadow Map Aliasing

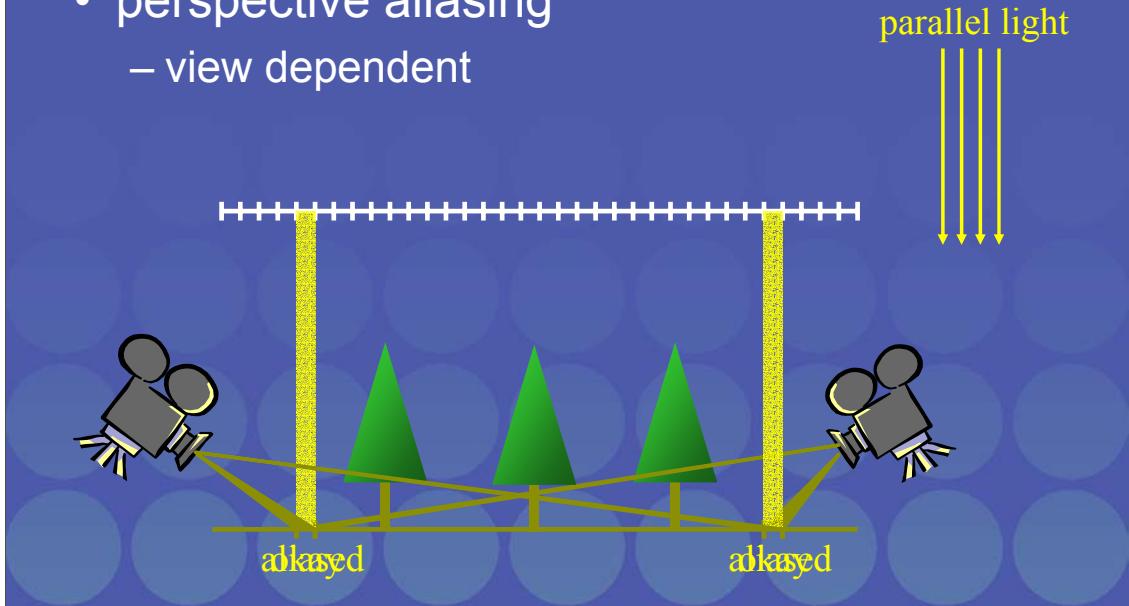
- perspective aliasing



We differentiate two kinds of shadow map aliasing. Perspective aliasing appears, if the camera is zooming to a shadow boundary.

Shadow Map Aliasing

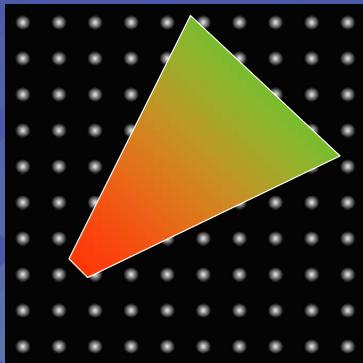
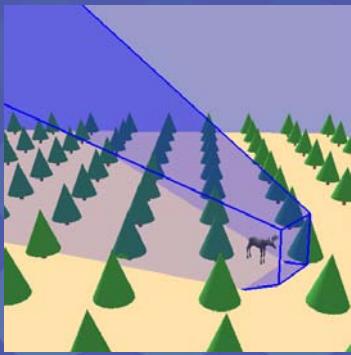
- perspective aliasing
 - view dependent



Perspective aliasing is view dependent, so it can only be handled with a view dependent shadow map. Standard shadow maps are view-independent, thus perspective aliasing is a problem.

Shadow Map Aliasing

- perspective aliasing
 - view dependent
 - close to camera: perspective aliasing
 - distant regions: oversampling

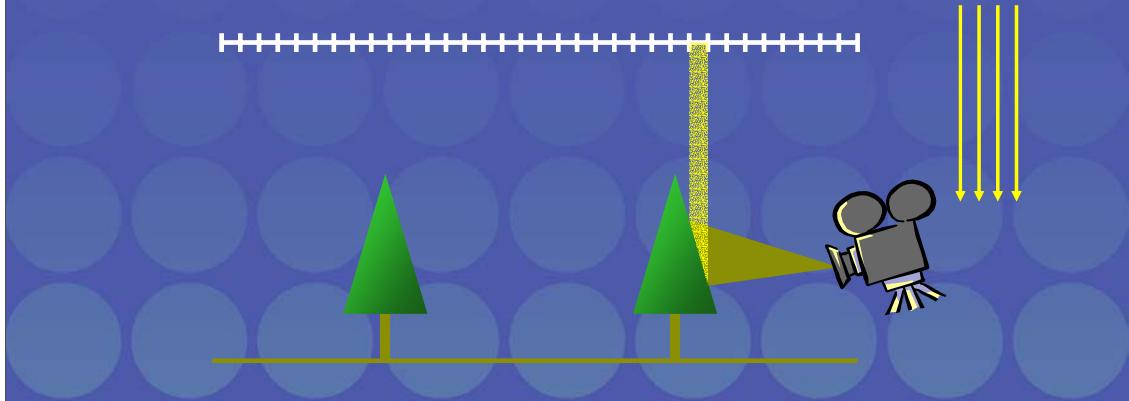


Perspective aliasing is generated by different sampling densities of the image and the shadow map. In general, distant regions are oversampled by a normal shadow map, whereas close regions exhibit undersampling and thus perspective aliasing. Note that of course also the undersampling can generate aliasing as any texture does, e.g. for shadows from a picket fence. Mipmapping is at least difficult for shadow maps. However, this shadow map texture aliasing is usually not considered a big problem, and in the following we only speak of aliasing in the sense of pixelization.

Shadow Map Aliasing

- projection aliasing
 - depends on angle of light incidence
 - very local

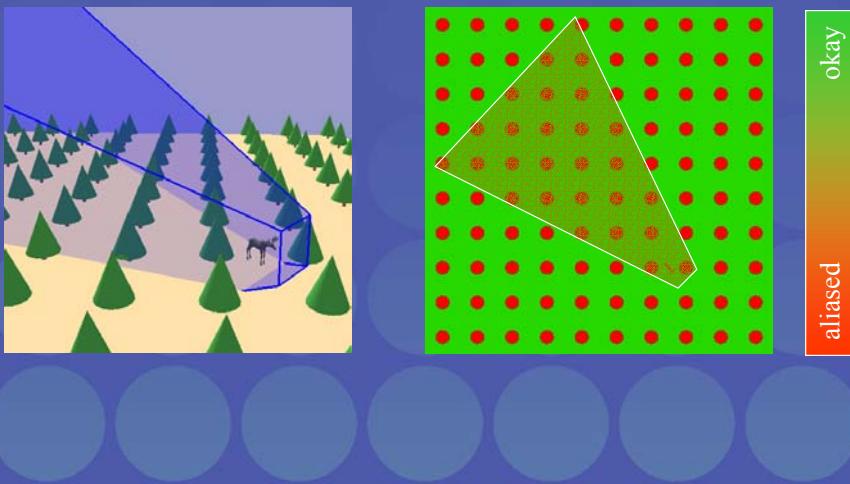
parallel light



Projection aliasing happens, if the light direction is almost parallel to a surface. The shadow map pixel footprint stretches along the surface, thus increasing the pixelization effect in the light direction.

Shadow Map Aliasing

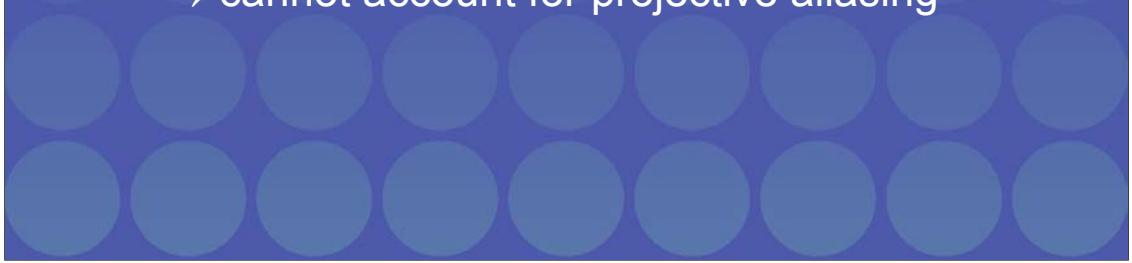
- projection aliasing
 - very local



Projection aliasing is much more local than perspective aliasing. It can happen everywhere, in close and distant regions.

Perspective Shadow Maps

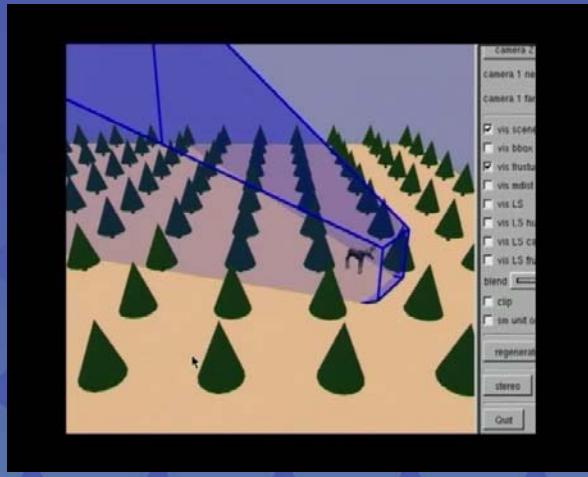
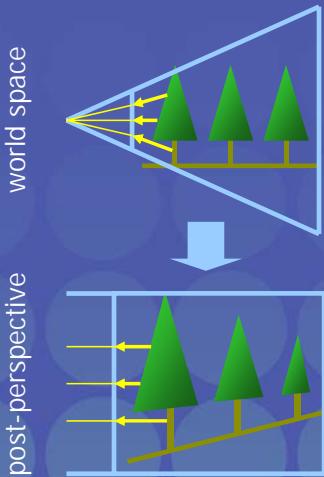
- Idea:
 - generate and apply shadow map in post-perspective space (clip space)
 - „Perspective Shadow Maps“ see the world after the perspective projection
 - account for perspective aliasing
 - cannot account for projective aliasing



The idea of perspective shadow maps is to generate the shadow map in a space that exhibits no or less perspective. We use the space after the perspective projection (normalization), i.e. the clip space. By this, we can reduce perspective aliasing, but not projective aliasing.

Post-Perspective Space

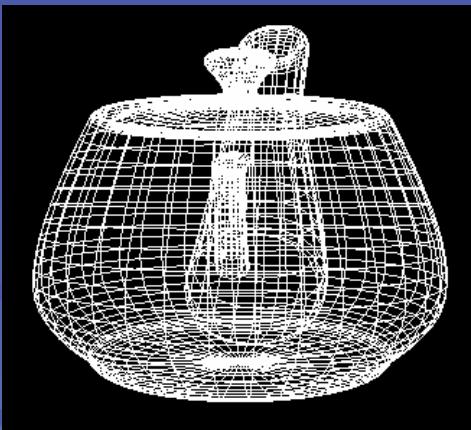
- perspective transformation
(gl uPerspective)



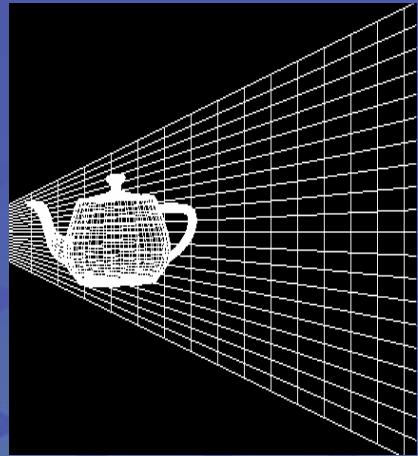
Here we see the effect of the perspective transformation. Objects close to the camera are enlarged, distant objects are shrunk. But lines remain lines. After the perspective transformation, the camera is always a parallel one, so the image size of a post-perspective region does no longer depend on the viewing distance. This is also why the image size of the footprints of shadow map pixels becomes independent of the viewing distance.

Post-Perspective Space

- properties of perspective transformation



camera view

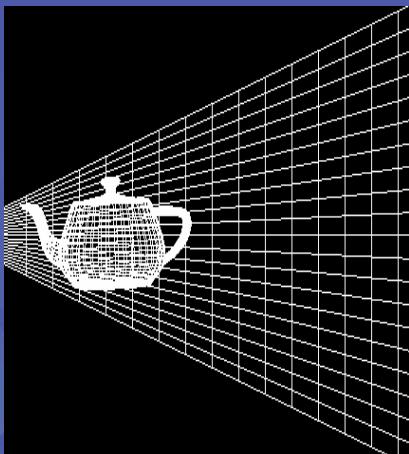


view frustum from side

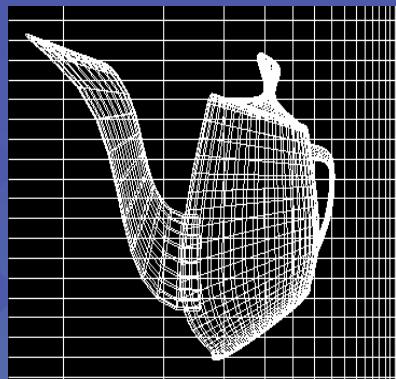
Just some remarks how the perspective transformation works. In the left, we see a camera view, in the right we see the corresponding camera frustum from the side.

Post-Perspective Space

- properties of perspective transformation



view frustum from side



after perspective transformation

Now we see, how the camera frustum (left) is transformed under the perspective transformation. Regions close to the viewer (sprout) are enlarged, distant regions (handle) shrunk. All lines converging in the camera (left) become parallel (right), so we need a parallel camera after the perspective transformation.

Post-Perspective Space

- properties of perspective transformation
- demo:
 - rotate object
 - move object forwards/backwards
 - change field of view
 - change near/far value

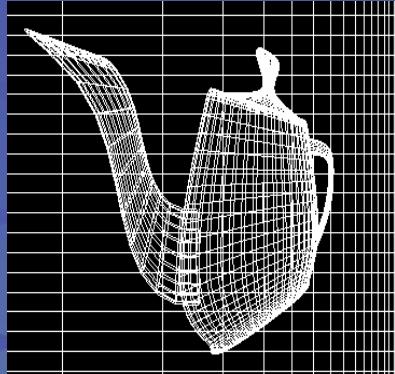


This demo shows how perspective space works. Its usage should be intuitive.

Post-Perspective Space

- properties of perspective transformation
 - objects close to camera are enlarged
 - distant objects are shrunk

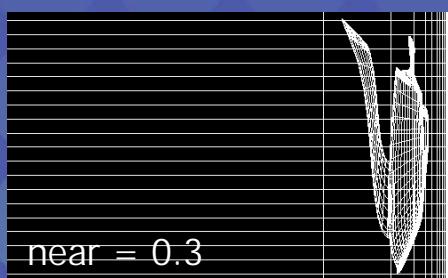
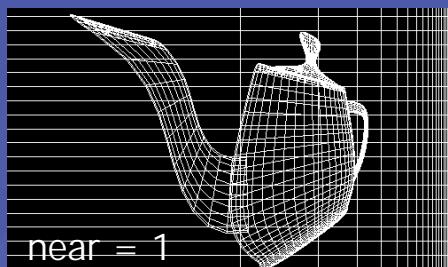
camera



Post-Perspective Space

- properties of perspective transformation

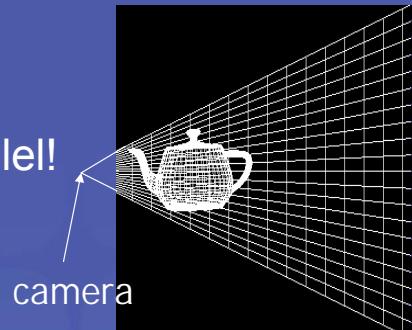
- non-linear mapping:
$$z \rightarrow 1 - 1/z$$
- near-value critical!
- too little near value
 - post-perspective cube mostly empty
 - waste of depth precision
- far value not critical
 - little influence
 - can become infinity!



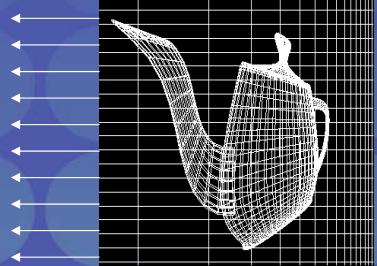
The selection of the near plane is very critical. Due to the hyperbolic mapping, the region from near to 2near covers half of the z-Range in post-perspective space! In most applications, the near plane is selected ways too conservative, so the scene is concentrated at the right end of the post-perspective cube.

Post-Perspective Space

- properties of perspective transformation
 - camera rays become parallel!
 - camera moved to infinity



camera

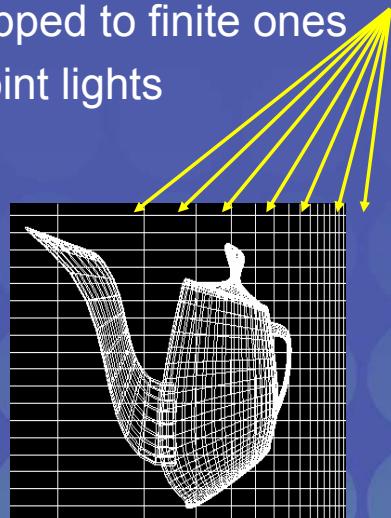
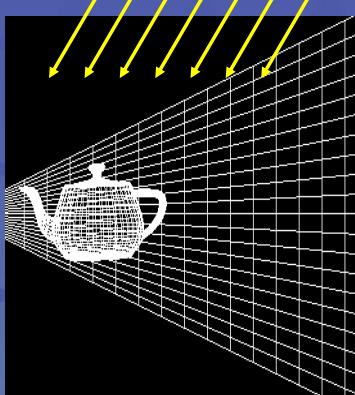


camera

As described previously, the camera point is mapped to infinity by the perspective mapping.

Post-Perspective Space

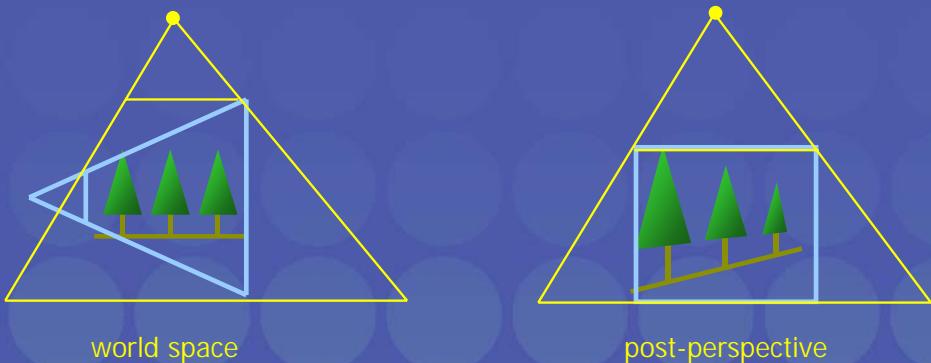
- vice versa:
 - infinite points can be mapped to finite ones
 - parallel lights become point lights



Something similar can happen with light sources: parallel lights can become post-perspective point lights and vice versa.

PSM idea

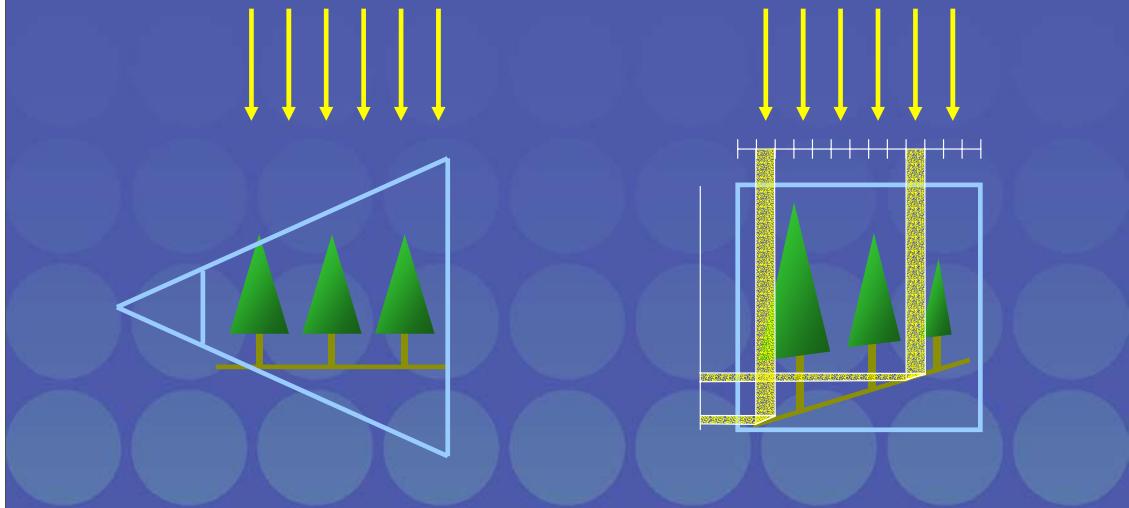
- uniform shadow map
- perspective shadow map



Back to PSM: the idea is to generate a shadow map in post-perspective space.

Perspective Shadow Maps

- standard shadow map
- perspective shadow map

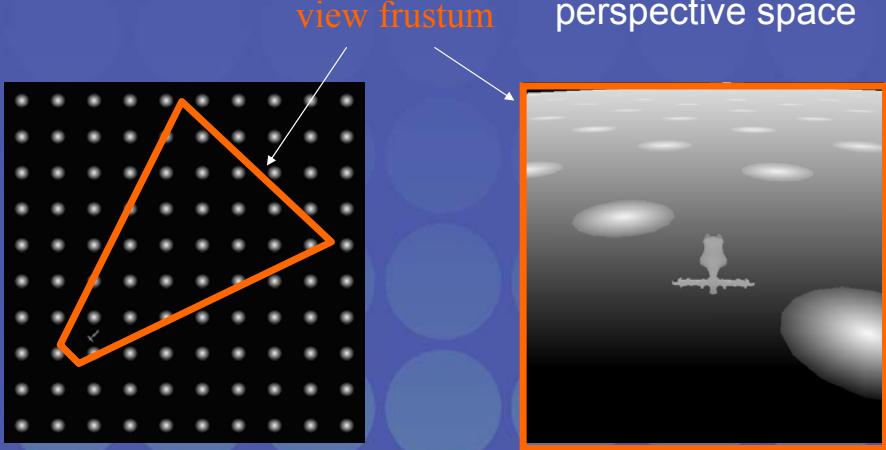


As a result, the image size of the footprints of single shadow map pixels is independent of their distance to the camera.

Perspective Shadow Maps

- standard shadow map
 - sees normal world

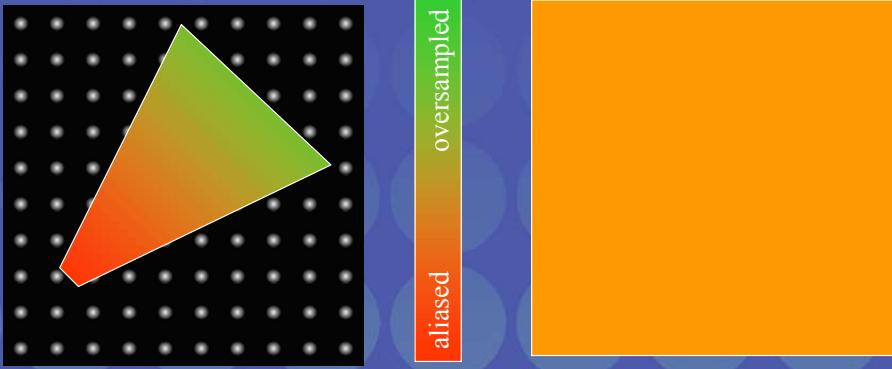
- perspective shadow map
 - sees world in post-perspective space



Here we see a uniform shadow map for the previous example on the left, and a perspective shadow map on the right. Note how the moose is enlarged.

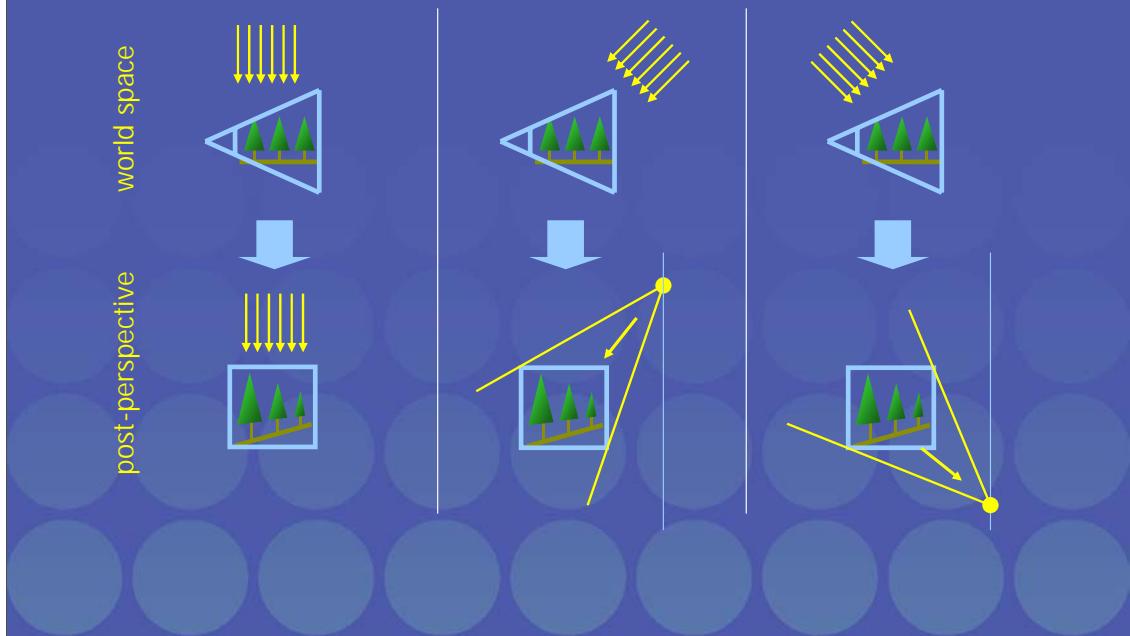
Perspective Shadow Maps

- standard shadow map
 - perspective aliasing close to camera
 - oversampling in distant regions
- perspective shadow map
 - uniform perspective aliasing over entire frustum



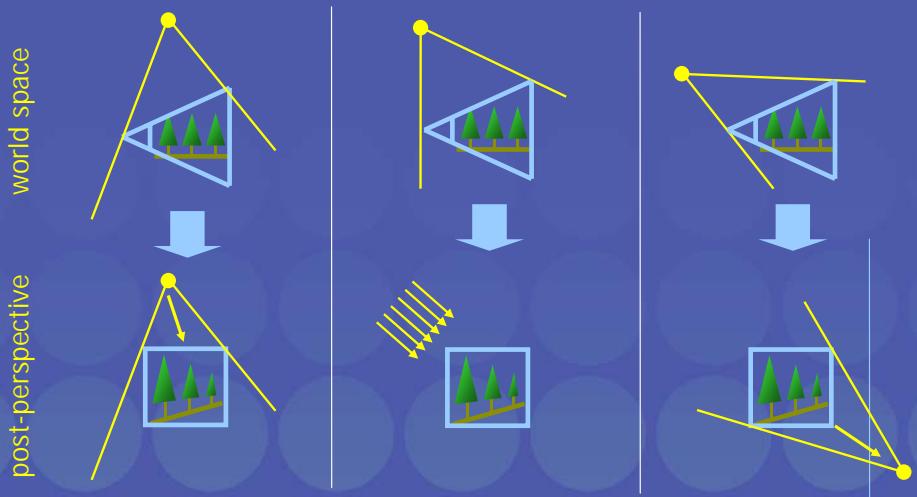
Ideally, with a perspective shadow map both oversampling and undersampling of the shadow map is avoided.

Parallel Light Transformation



Light source transformations: a parallel light perpendicular to the view direction remains parallel. This is ideal for PSMs. A parallel light shining towards the viewer becomes a point light on the infinity plane, this is a plane just behind the far plane, to which all infinite world points are mapped. By generating a shadow map for such a light in post-perspective space, we need a perspective light frustum, which introduces perspective distortion, and thus aliasing again. A parallel light from behind is also mapped to the infinity plane, but this time it becomes a light sink, i.e. a light source for which all light rays converge in a point. Such inverse lights have no physical counterpart, but they can be easily handled by using a shadow map with inverted depth test.

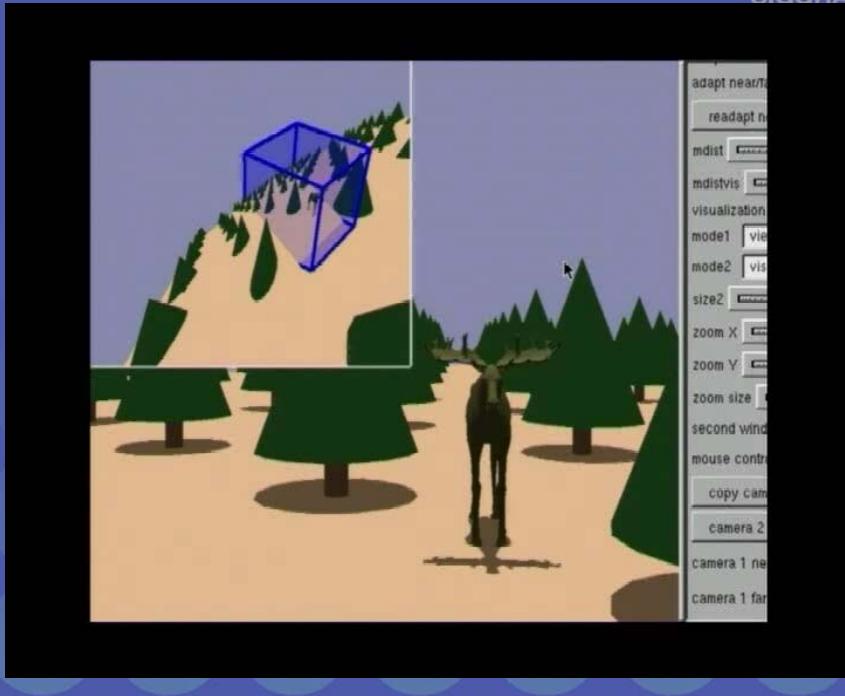
Point Light Transformation



Point lights in general remain point lights, except lights in the camera plane (plane through camera, which is perpendicular to viewing direction). Such lights become parallel lights.

Point Light Transformation

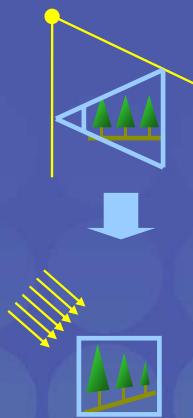
SIGGRAPH2004



Here we see our test scene with a moving parallel light. In the window in the upper left, we see the corresponding post-perspective version with the corresponding moving point light.

Discussion

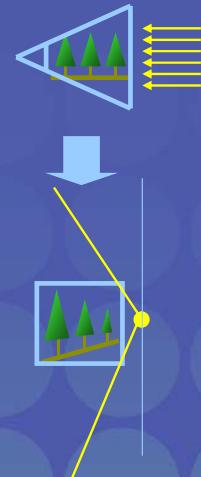
- ideal case:
parallel light in post-perspective space
(→ no perspective aliasing)
 - directional light parallel to image plane
 - point light in camera plane (miner's lamp)



The ideal case for PSM is that the post-perspective light is parallel. In this case, a shadow map pixel is parallel projected onto a scene object, and the footprint is projected parallel to the image, so the image footprint size only depends on the projection angles, but not on viewing distances. Perspective aliasing is completely avoided in this case.

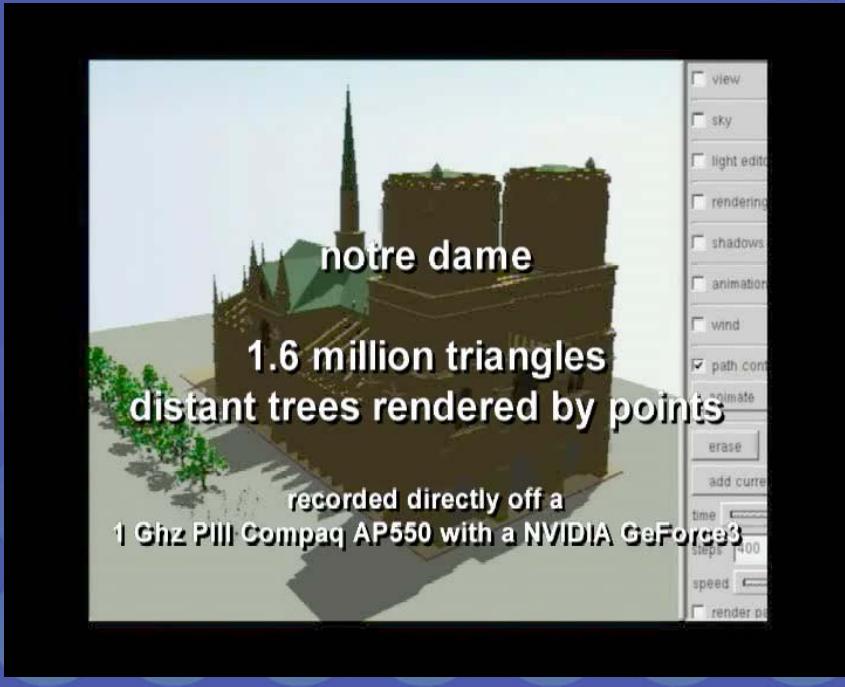
Discussion

- worst case:
point light in post-perspective space
close to unit cube
(→ severe new distortions)
 - e.g. parallel light from
front or behind



The worst case are post-perspective point lights which are very close to the post-perspective unit cube. These lights require a light frustum with large opening angle, which can re-introduce severe aliasing.

Results



Problems of original paper

- robustness
 - quality changes for moving lights/camera
 - „swimming“ shadows
 - depth bias
 - constructions of original paper lead to singularities
 - idea is simple, but robust implementation awkward

The original paper described an approach, which suffered from several robustness problems. A lot of following work has been spent on making these cases more robust.

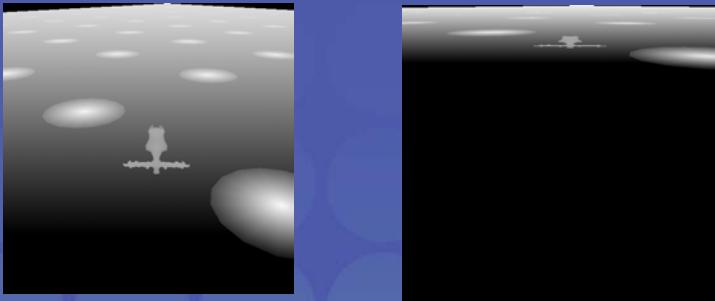
Issues and extensions

- camera near/far selection
- shadows from the back
- deep light sources
- usage of cube maps
- generalization

In the following, we will describe several critical issues of PSMs, and some approaches to solve them.

Near/Far Selection

- close near plane → strong perspective distortion



It is extremely important to select the near plane as far as possible. Otherwise, a lot of shadow map space is wasted.

Near/Far Selection

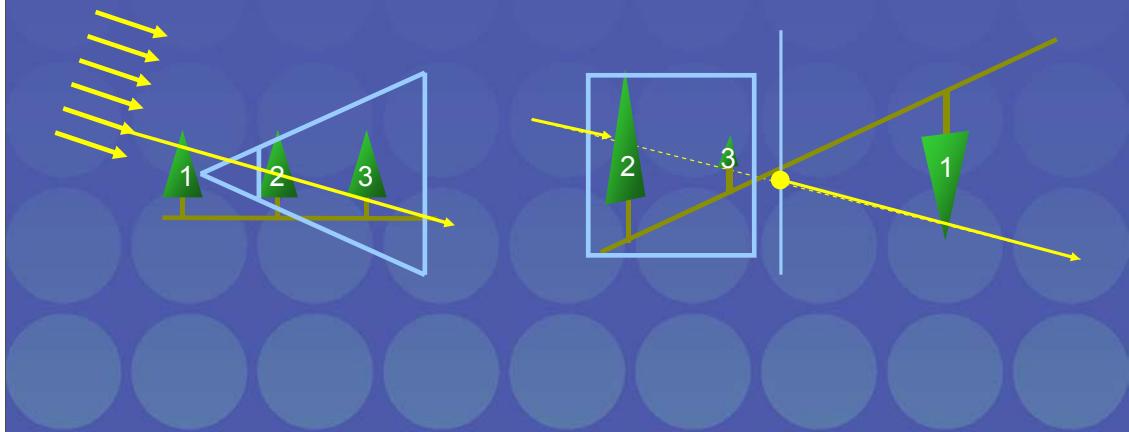
- solutions:
 - select near plane as far as possible
(e.g. read back depth buffer → slow)
 - virtually move camera back,
compute psm for slightly larger
frustum
 - move near plane
forward to original
plane [Kozlov04]



If the near plane cannot be moved easily, it is better to virtually move the camera back and leave the near plane fixed. By this, we obtain some wasted pixels on the sides, but also a much better depth distribution. The offset then controls how the shadow map resolution is distributed between near and far regions.

Shadows From the Back

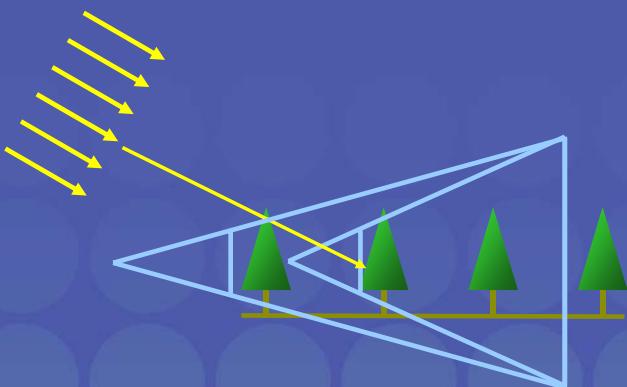
- object behind viewer casts shadow
- wrap around at infinity



Shadows from behind the viewer are a big problem: the perspective mapping maps objects behind the viewer to beyond the infinity plane, so the order of hits along a ray changes! In order to see occluding points 1, 2 and 3 in the right image, we would need a shadow map looking into both directions.

Shadows From the Back

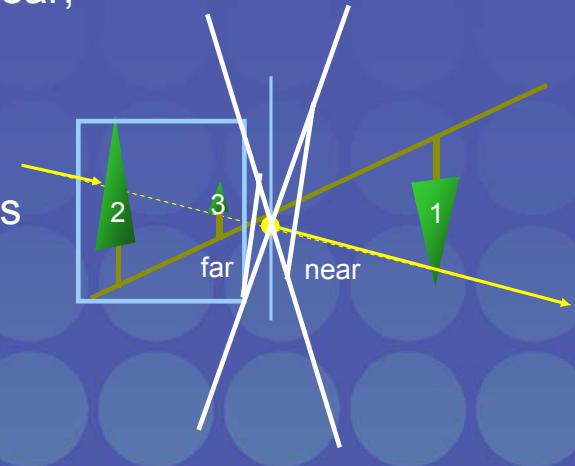
- solution 1:
virtually move
back camera
until
no shadows
from
behind



Our solution from the original paper is to again move back the camera virtually, in order to move the camera behind potential occluders.

Shadows From the Back

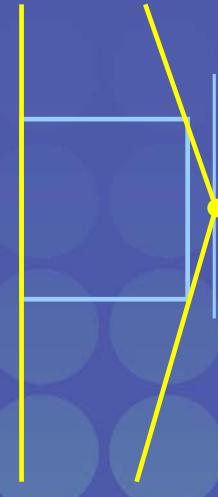
- surprising solution 2 [Kozlov04]: set near plane to a negative value
 - eye rays start at –near, go to –infinity, wrap to infinity and end at far
 - sign of homogenous w happens to be correct!



A much more clever solution is described in Kozlov04: the perspective matrix allows to use a negative near plane! The resulting shadow map then also sees the objects behind the camera, which solves the problem.

Deep Light Sources

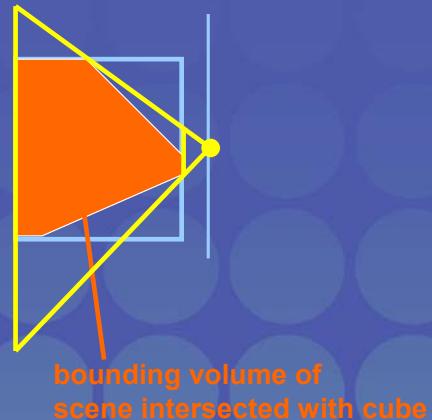
- deep lights
 - close to post-persp. cube
 - wide frustum
 - large distortions



Deep lights are difficult, because in post-perspective space a wide frustum is needed that contains the entire post-perspective cube.

Deep Light Sources

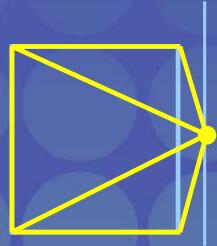
- however:
 - scene's bounding box shrinks towards the far plane
 - intersection of bounding box and cube much smaller close to light



However, the scene also shrinks towards the right side of the cube. If the scene can be bounded, the bounding box will become small towards the right end, so a narrower frustum is sufficient.

Cube Maps

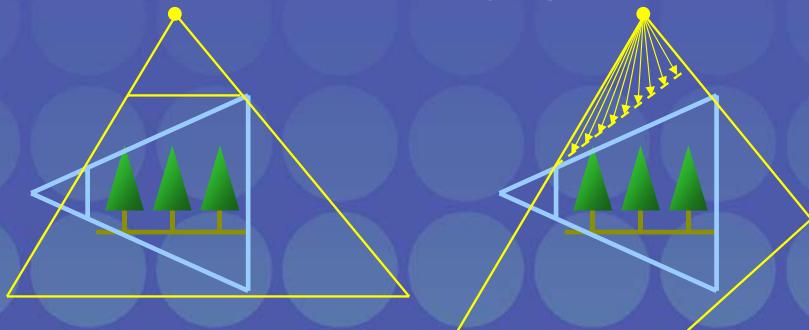
- [Kozlov04]: use cube maps to better sample the unit cube
 - one shadow map for every backfacing side of the unit cube
 - can be achieved with cube maps: $\text{texCoord} = \text{vertex coord} - \text{light pos}$
 - much better quality
 - also useful for point lights
 - generation of cube maps requires multiple passes



In Kozlov04 an elegant approach is described that uses cube maps to better sample the post-perspective cube.

Thesis of Hamilton Yu-Ik Chong

- generalization of PSM idea
- observation:
shadow maps have a degree of freedom that have been unused before PSMs: the orientation of the image plane



In his Master thesis, Hamilton Yu-Ik Chong noticed, that the PSM exploits one remaining degree of freedom for shadow maps: the tilting of the image plane. He describes approaches to optimize this remaining degree of freedom.

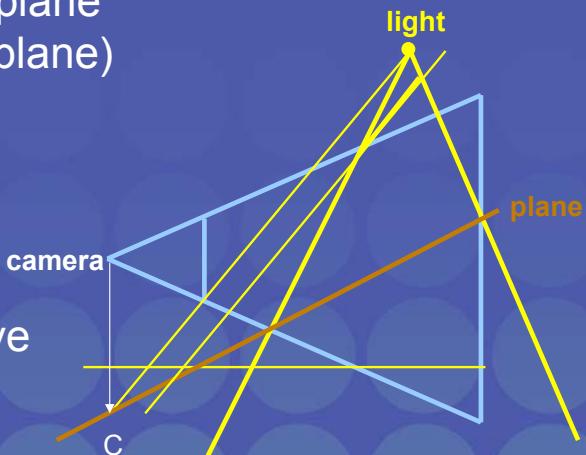
- optimal angle for single plane geometry

- project camera to plane
(parallel to image plane)

- C

- light plane parallel
to connection
of light and C

- minimal perspective
aliasing on plane



In order to optimize the shadows for a single plane and a point light, the following construction can be used: project the camera down to the plane (parallel to the image plane) -> point C. Use an image plane for the light frustum that is parallel to the connection of the point light and C.



Shadow Silhouette Maps

Eric Chan

Massachusetts Institute of Technology



Game Plan



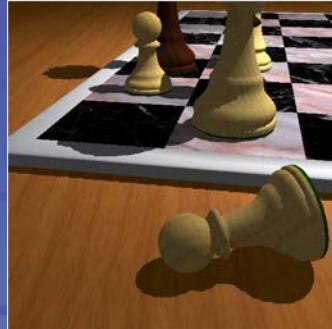
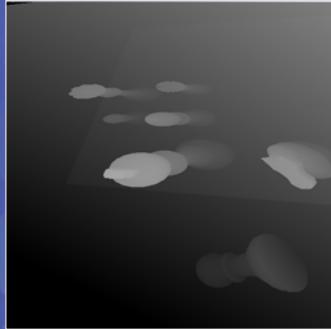
Motivation
Algorithm
Implementation
Examples
Comparison



Motivation

Why another shadow algorithm?

Why not use perspective shadow maps?



Stamminger and Drettakis, SIGGRAPH 2002

In a nutshell, the topic of this session – the shadow silhouette map – is an extension to the shadow map algorithm that addresses aliasing problems. Perspective shadow maps, described in another session, address the same problem. So why are we bothering with this algorithm if we already have that one? It turns out that the two approaches have different tradeoffs. Hopefully these tradeoffs will become clear as we proceed.

Perspective Shadow Maps



Addresses perspective aliasing

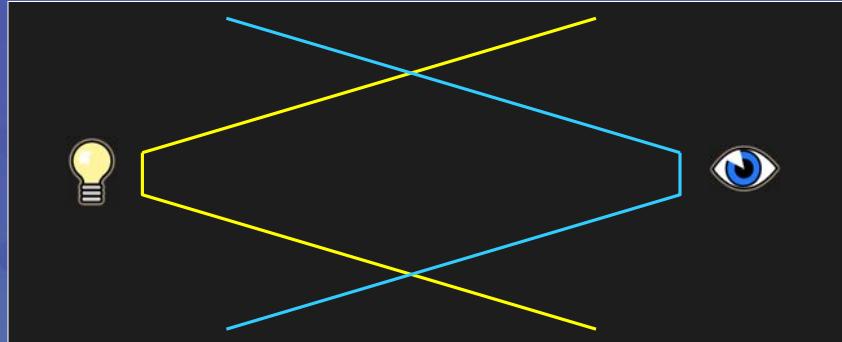
Optimizes distribution of depth samples

Difficulties:

- Does not handle projection aliasing
- Dueling frusta problem

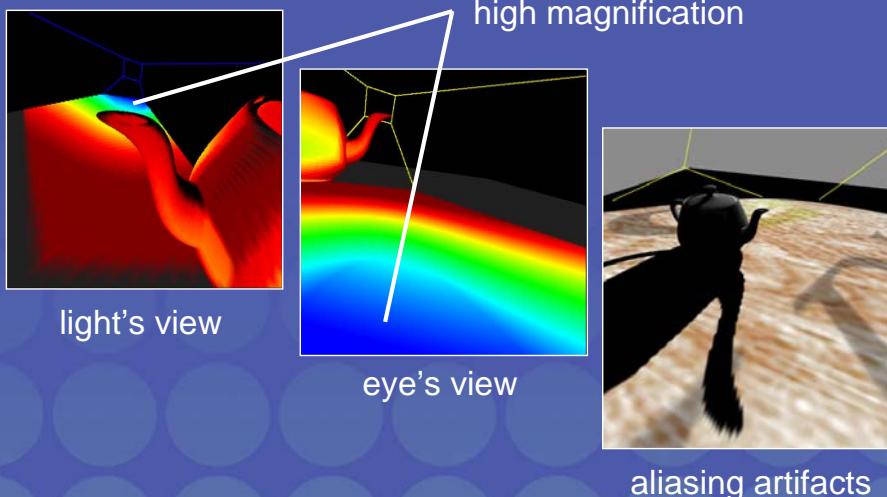
There are two types of aliasing caused by shadow maps: perspective and projection aliasing. See Marc Stamminger's session on Perspective Shadow Maps for a discussion of these aliasing types. Perspective Shadow Maps (PSM) only addresses perspective aliasing, and then only in some cases. In particular, the PSM optimizes the distribution of the depth samples so that more samples are located toward the viewer. However, when the light and camera roughly face each other, PSMs don't work as well. This scene configuration is called the "dueling frusta" problem.

Dueling Frusta Problem



Here is a diagram that shows the light and camera frusta facing each other.

Dueling Frusta Problem



Mark Kilgard, NVIDIA

Here is a visualization of what happens from both the light's view and the observer's view. The left two images are color-coded so that red pixels show areas of the scene for which the shadow map is least magnified, and the blue pixels show areas where the shadow map is most magnified. In the left image, we see the scene from the light's viewpoint. The dark blue outline is the observer's view frustum.

In the middle image, we see the same scene, but from the observer's view. The yellow lines represent the light's view frustum, which faces the observer. Here, we see that areas of the image close to the viewer are precisely the areas where the shadow map has the greatest magnification when projected onto the image. Unfortunately, when using PSMs, the two perspective transforms (one from the light, one from the camera) mutually cancel and the result is that we're back to a standard, uniform shadow map.

The resulting shadow aliasing artifacts are seen in the right image.

In summary, PSMs are very useful, but they only address aliasing in certain cases. This is the main motivation for exploring another shadow algorithm, such as shadow silhouette maps.

Shadow Silhouette Maps



- Research at Stanford University
 - P. Sen, M. Cammarano, and P. Hanrahan
 - Proceedings of SIGGRAPH 2003
- See course notes
- Also available online

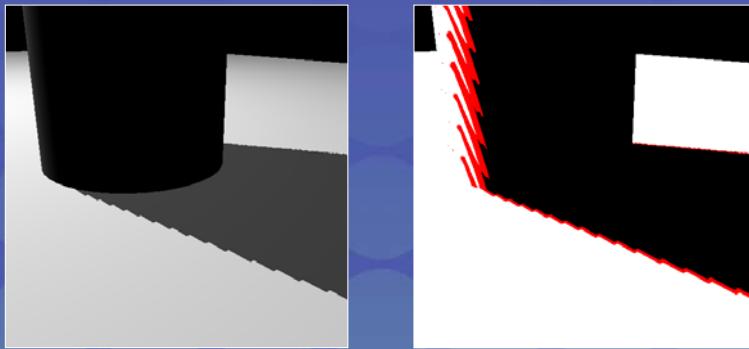
The shadow silhouette map algorithm was developed at Stanford University by Pradeep Sen, Mike Cammarano, and Pat Hanrahan. Their original paper which describes the method in detail was published in the Proceedings of ACM SIGGRAPH 2003. A copy of the paper should be included with these course notes. You can of course also download the paper online.

Pradeep Sen gave a nice talk at SIGGRAPH 2003, presenting the silhouette map algorithm. Many of the figures in these slides and notes are borrowed from his presentation.

Observation

Shadow maps

- undersampling can occur anywhere
- artifacts visible only at shadow edges



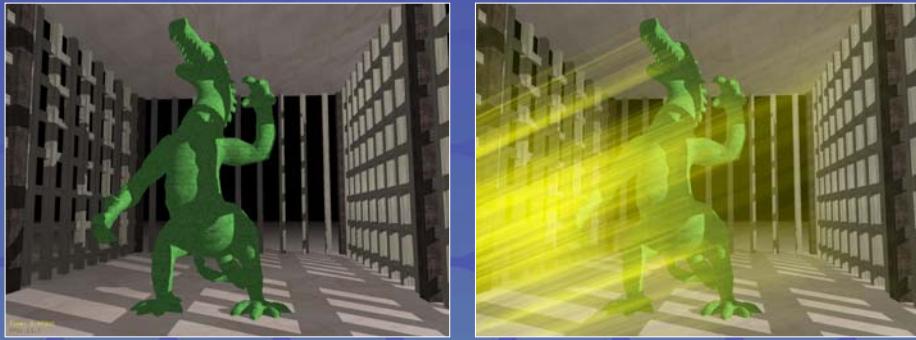
The silhouette map algorithm is based on the following simple observation.
Shadow maps can lead to undersampling, but the artifacts are visually
objectionable only at the shadow silhouettes, i.e. the edges between
shadowed and illuminated regions.

Observation



Shadow volumes

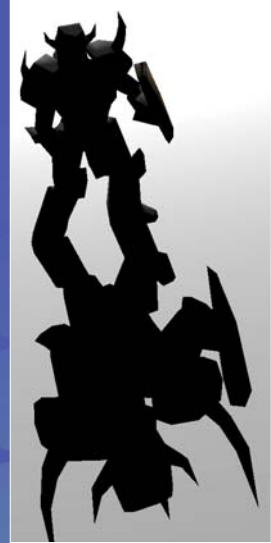
- accurate everywhere, but high fillrate
- accuracy only needed at silhouettes



In contrast, shadow volumes give per-pixel accuracy everywhere, but this degree of accuracy is only needed at the silhouettes. The price for being accurate everywhere with shadow volumes is high fillrate and bandwidth consumption, illustrated in the right figure. The yellow polygons visualize shadow volume polygons. Clearly there is a lot of shadow volume overdraw in this scene, which leads to high fillrate and bandwidth. One of the characteristics of shadow maps, as we shall see, is relatively low bandwidth and fillrate consumption. This helps to keep the algorithm scalable to large scenes.

Algorithm Goals

- Accuracy of shadow volumes
- Efficiency of shadow maps
- Treats perspective and projection aliasing
- Supports dynamic scenes
- Maps to graphics hardware



We want a hybrid algorithm that combines the best characteristics of shadow maps and shadow volumes. The silhouette map algorithm will focus on the shadow silhouettes, since those are the pixels in the image that are critical to get right.

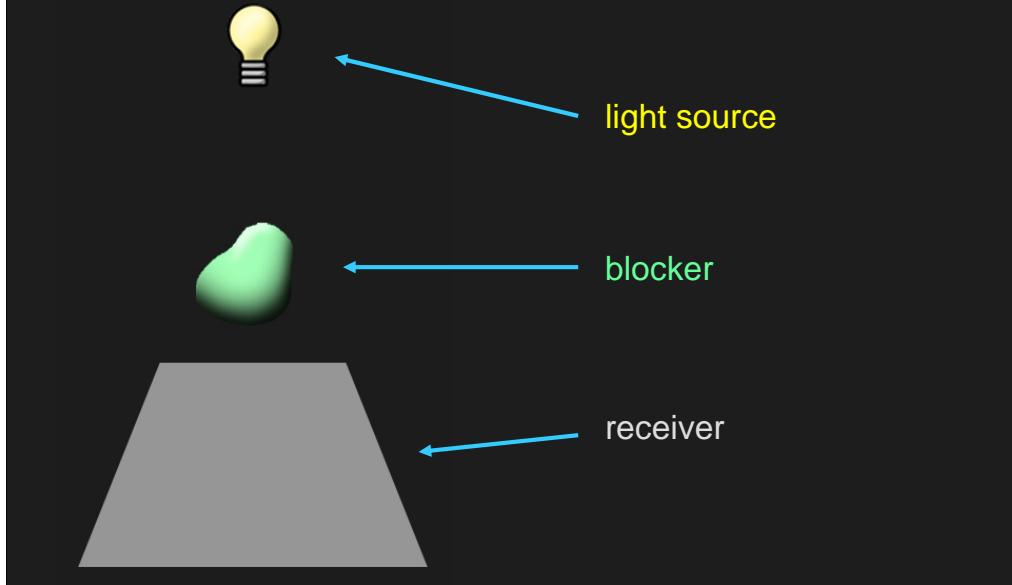
These are the goals of the silhouette map algorithm. Ideally, we would have the accuracy of shadow volumes and the efficiency of shadow maps. As we'll see, silhouette maps don't quite achieve this goal, but they do offer an excellent tradeoff. Silhouette maps are designed to work on dynamic scenes, i.e. the light, observer, and objects can move from frame to frame; no precomputation is required. Finally, silhouette maps are designed to be simple enough to implement on graphics hardware. As we'll see, though, they make heavy use of the programmable features of modern graphics hardware.

Overview



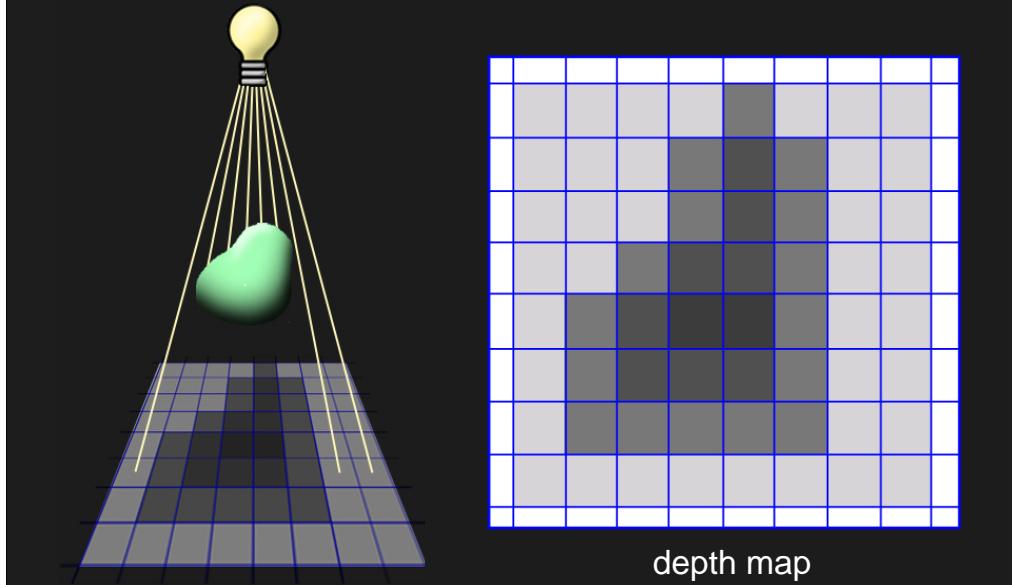
The basic idea is to augment the normal depth map with an extra buffer, called a silhouette map, which helps to represent shadow silhouettes more accurately.

Shadow Map (Review)



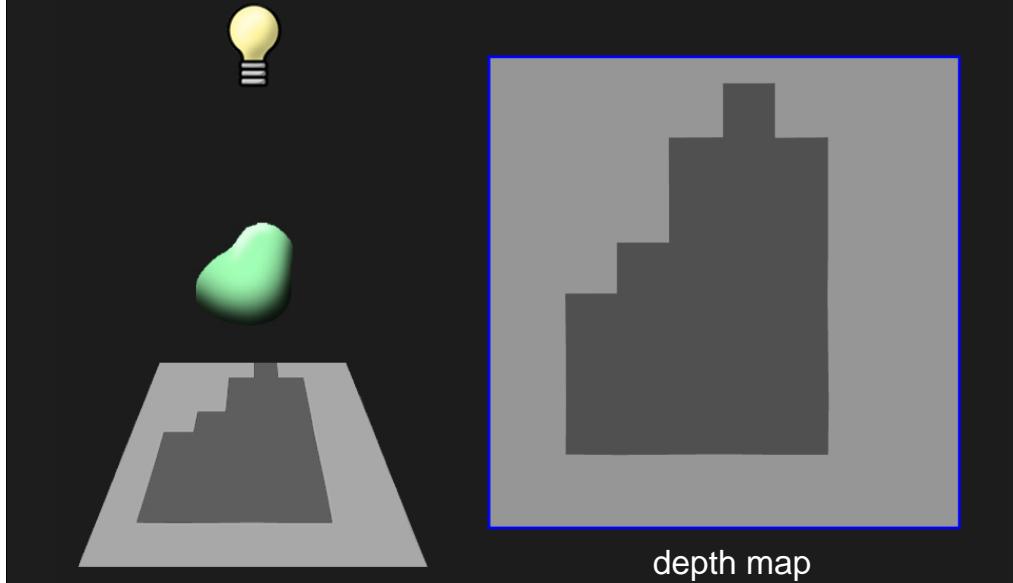
To understand how the silhouette map algorithm works, let's review how the regular shadow map algorithm works. We have a light source, blocker, and receiver.

Shadow Map (Review)



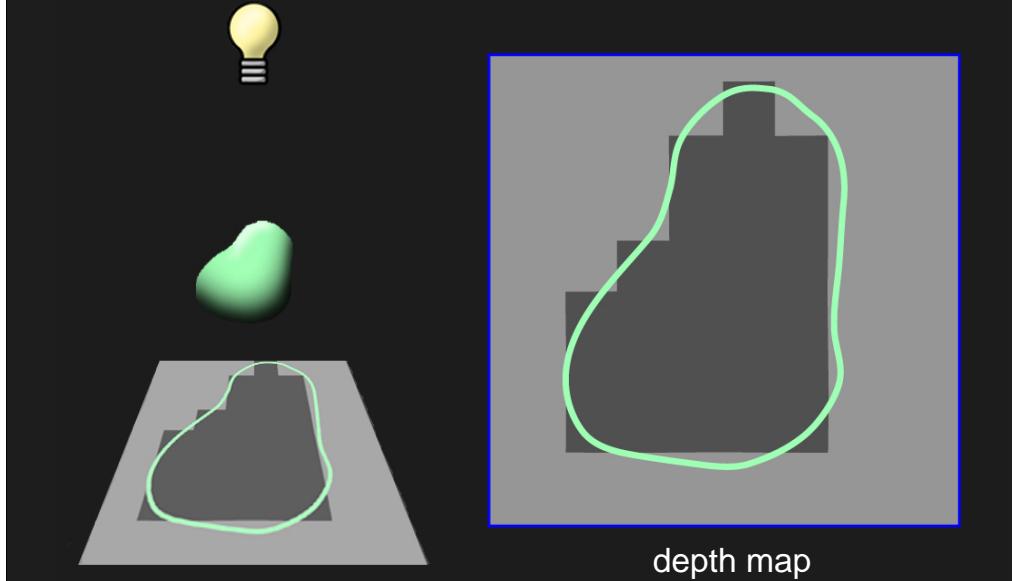
We rasterize the blocker into a depth map. In the visualization on the right, dark values represent small depths (close to the light), and light values represent large depths (far from light).

Shadow Map (Review)



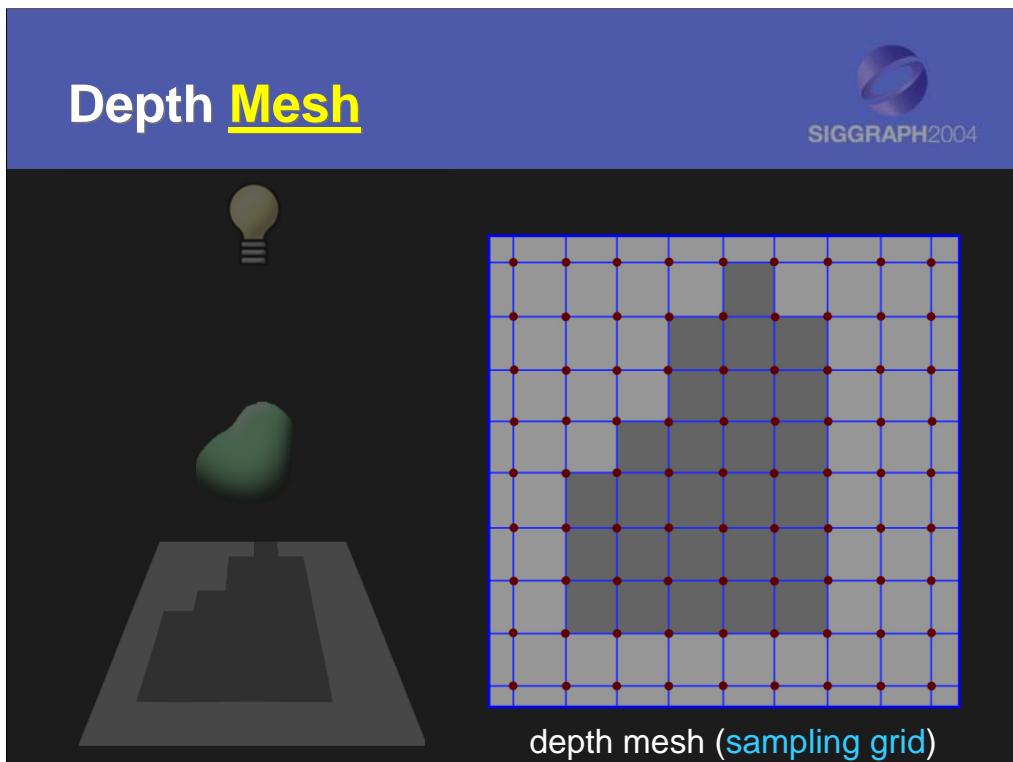
Due to limited depth buffer resolution, we obtain a poor representation of the blocker's shadow silhouette.

Shadow Map (Review)



For reference, the true silhouette curve is shown in green. We can think of the standard shadow map algorithm as providing a piecewise-constant approximation to the shadow contour. This is because all samples in the final image that get mapped to a given texel in the shadow map will have the same binary value: 0 if the depth test fails, 1 if the depth test passes. It is this piecewise-constant approximation that leads to blocky aliasing artifacts in the final image.

Depth Mesh

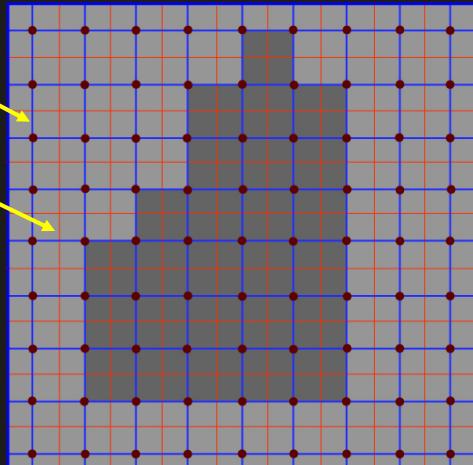


Now let's see how we can improve the approximation. Let's stretch our minds a bit and think of the depth map not as a buffer, but as a mesh where each sample is a vertex of the mesh. In 2D, the mesh is just a uniform, rectilinear grid as shown here.

Depth Mesh

original grid (blue)

dual grid (red)



The depth mesh is shown in blue. As we'll soon see, it will be useful also to consider the dual mesh, shown in red. Practically, this is the same as the original grid, but offset by $\frac{1}{2}$ a pixel in both x and y.

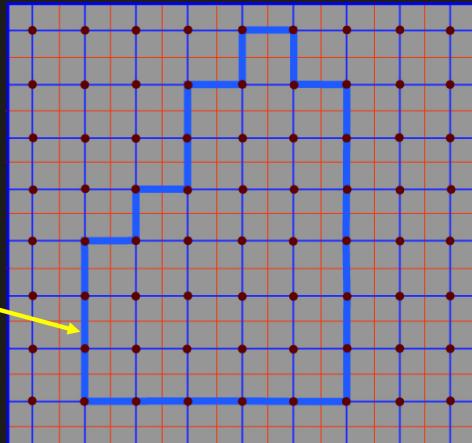
Depth Mesh



original grid (blue)

dual grid (red)

discrete silhouette boundary



depth mesh + dual mesh

With the standard shadow map algorithm, we get discrete shadow boundaries that are aligned with the rectilinear grid of the depth mesh.

Depth Mesh

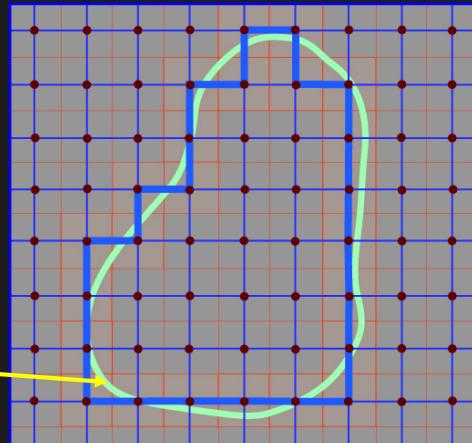


original grid (blue)

dual grid (red)

discrete silhouette boundary

continuous silhouette boundary (green)



Here's the magic: why restrict ourselves to a regular depth mesh? It would be better if we could somehow deform the depth mesh so that the samples are better aligned with the true silhouette boundary (shown in green). In fact, we can do just that.

Depth Mesh



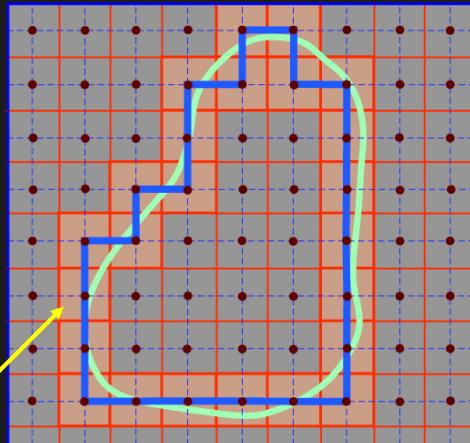
original grid (blue)

dual grid (red)

discrete silhouette boundary

continuous silhouette boundary (green)

silhouette map pixels

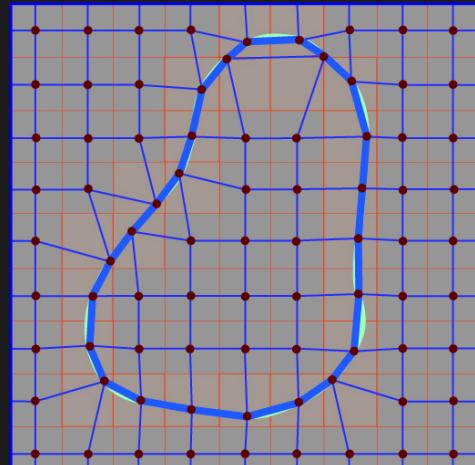


Here's where the dual mesh comes in handy. Look at all the cells of the dual grid (highlighted in red) that contain the continuous silhouette curve. These are also the cells that contain the discrete, grid-aligned shadow boundary (dark blue).

Depth Mesh Deformation



Move depth samples
to lie on silhouette curve



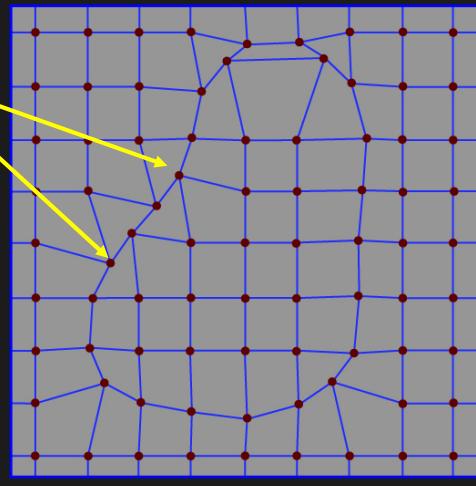
deformed depth mesh

The idea is to deform the depth mesh: move the relevant depth samples so that they lie precisely on the silhouette curve itself. This results in the deformed depth mesh shown here.

Depth Mesh Deformation



adjusted depth samples



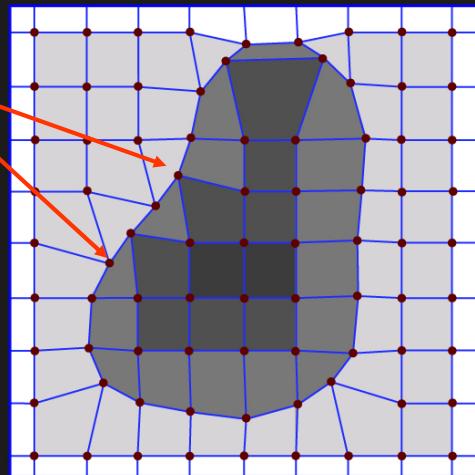
deformed depth mesh

We still have the same number of depth samples as before. It just so happens that some of them have been moved from their original position to lie on the silhouette curve.

Depth Mesh Deformation



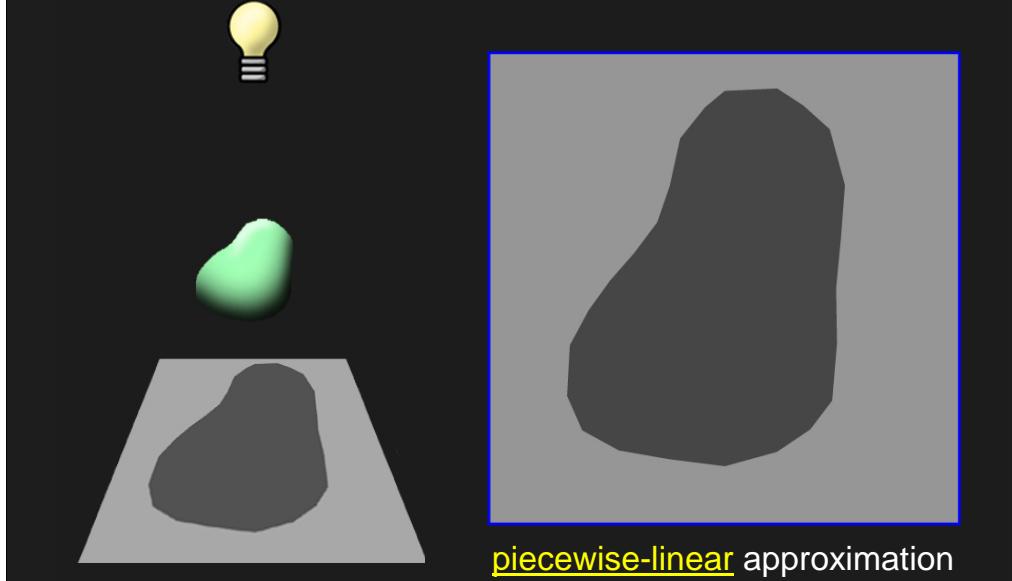
adjusted depth samples



deformed depth mesh

Now let's see what happens when we apply the shadow map algorithm, this time with the deformed mesh.

Better Approximation



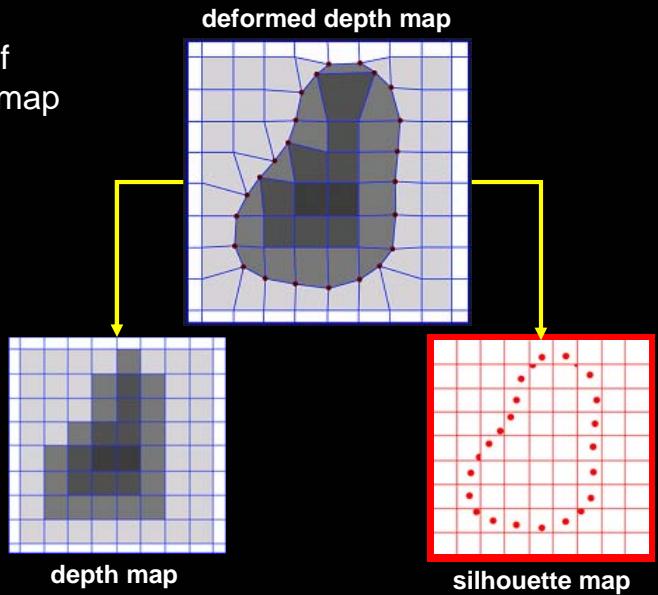
piecewise-linear approximation

Notice that now we get a much better approximation to the shadow silhouette. Instead of a piecewise-constant approximation to the contour, we have piecewise-linear approximation (imagine connecting the dots of the deformed depth samples).

Conceptually, this is all there is to the silhouette map algorithm.

Silhouette Map

Decomposition of deformed depth map



In practice, however, we can't easily create or use deformed depth meshes on graphics hardware. But we can look at the problem from a slightly different angle. Instead of working directly with a deformed depth mesh, we can use two buffers: a regular depth map and a silhouette map, a 2D image that we'll describe in a moment. Together, these buffers effectively give you a deformed depth map that can be used to give the piecewise-linear approximation of the shadow boundary.

What is a Silhouette Map?



Many ways to think about it:

- Edge representation
- 2D image, same resolution as depth map
- Offset from depth map by $\frac{1}{2}$ pixel in x, y
- Stores xy-coordinates of silhouette points
- Stores only one silhouette point per texel
- Piecewise-linear approximation

So what exactly IS a silhouette map? There are many ways to think about it. Abstractly, it's an edge representation, meaning that its main purpose is to store accurate edge information. After all, the overall goal here is to improve the shadow silhouette. Concretely, the silhouette map is just a buffer offset from the depth map by $\frac{1}{2}$ pixel in both x and y. Each texel in the silhouette map stores a single point; for texels that are crossed by the silhouette curve, the texel stores a point that lies on the curve. Note that only one silhouette point is stored per texel. We'll see some implications of this restriction.

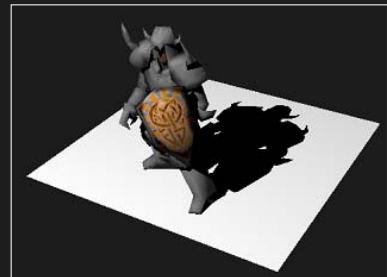
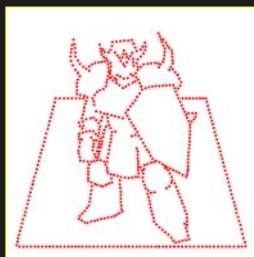


Algorithm

Algorithm Overview



Image-space algorithm



Here's an overview of the algorithm. There are 3 rendering passes.

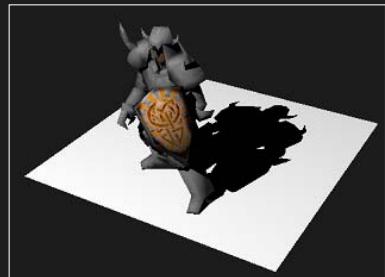
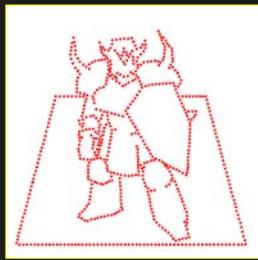
Algorithm Overview



Step 1



Create depth map

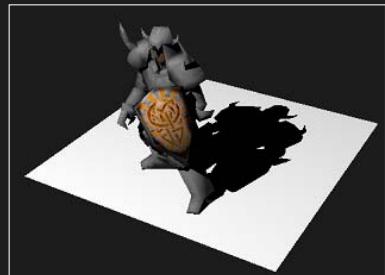
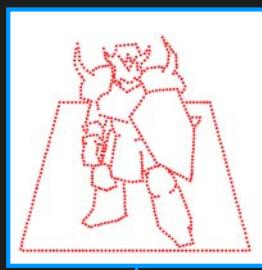


The first pass draws a regular depth map from the light's viewpoint. This is exactly the same as in the standard shadow map algorithm.

Algorithm Overview



Step 2



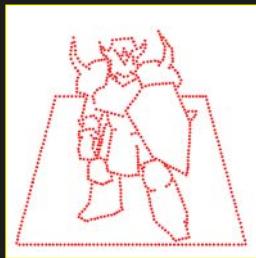
Create silhouette map

The second step is to render a silhouette map, also from the light's viewpoint. We'll see later exactly how this is done.

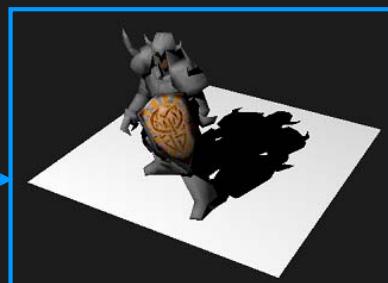
Algorithm Overview



Step 3



Render scene and shadows

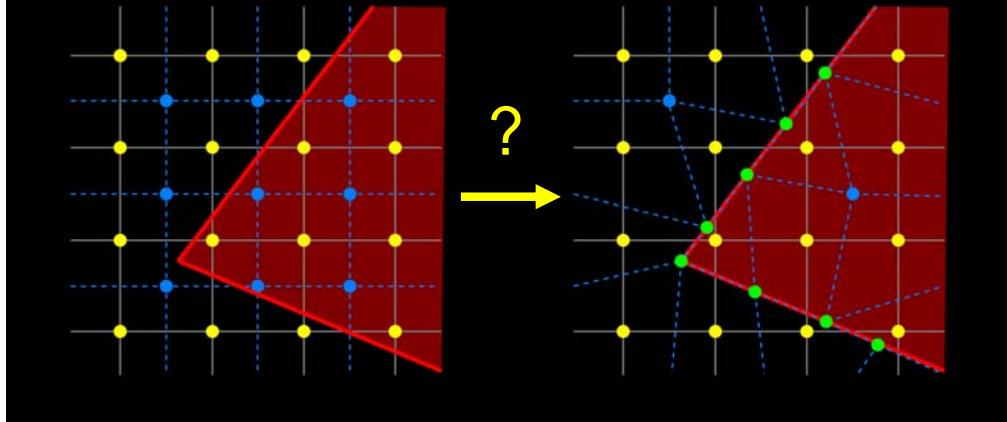


Finally, we render the scene from the observer's viewpoint and refer to both the shadow map and silhouette map to obtain accurate shadows.

Algorithm Details



- Focus now on concepts
- Worry later about implementation



This part of the discussion will focus on the concepts of the algorithm. It may not yet be clear how to implement this in hardware. Don't worry; we'll cover this later.

Create Depth Map



Same as in regular shadow maps

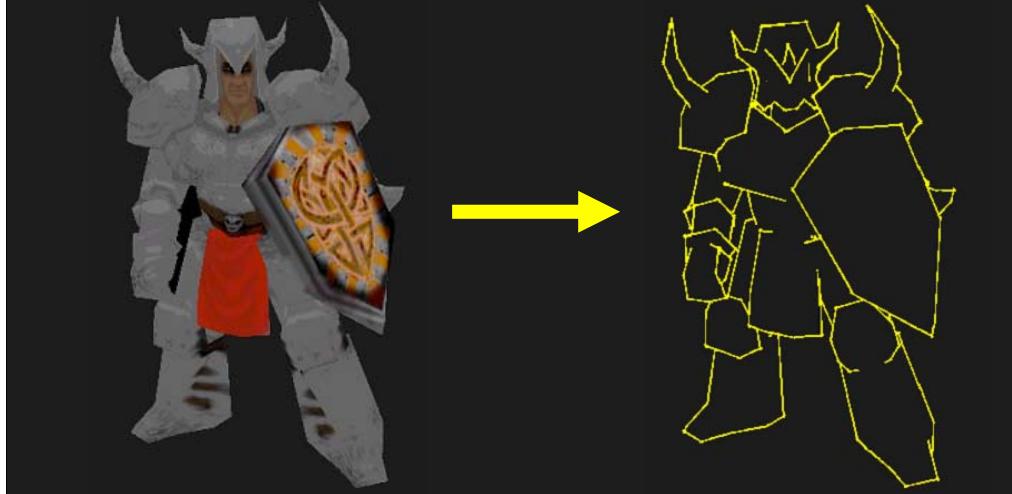


The first step is to render a depth map.

Identify Silhouette Edges



Find object-space silhouettes (**light's view**)



Before we can create a silhouette map, we have to identify the silhouette edges. This task is sometimes referred to as object-space silhouette extraction. For polygonal models, a simple way to perform this task is to loop through all edges and check to see if one of its adjacent faces is facing the light and another is facing away. This approach may sound overly simplistic, but in fact it's one of the best methods available for dynamic scenes with animated characters. Best of all, it's easy to implement and always works.

Note the assumption that we've made here: objects (in particular, the blockers in the scene) are represented as polygons.

Create Silhouette Map



- Rasterize silhouette edges (**light's view**)
- Find points that lie on silhouette edges
- Store one such point per texel

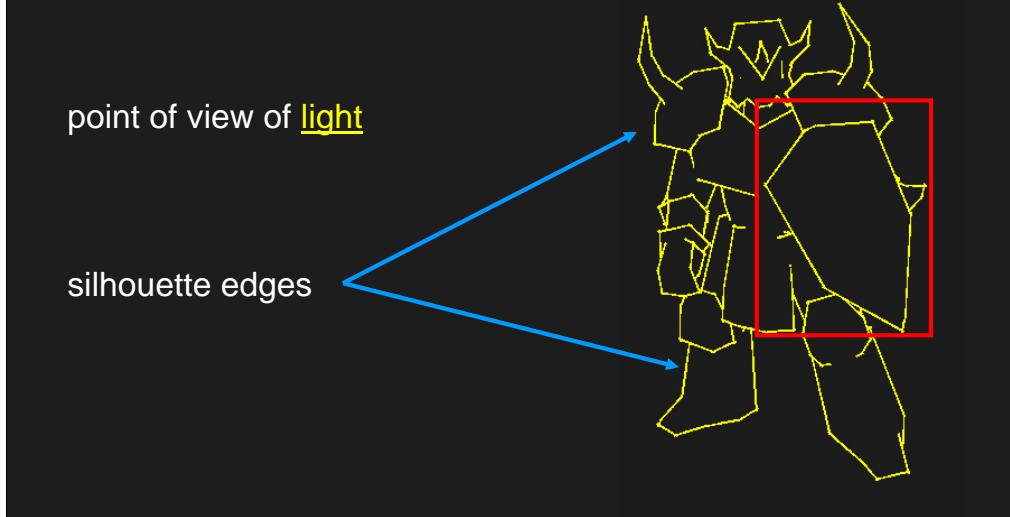


Now that we have the silhouette edges, we draw them (again from the light's viewpoint) to generate the silhouette map. The overall idea is to pick points that lie on the edges and store these points in the silhouette map.

Compute Silhouette Points



Example:

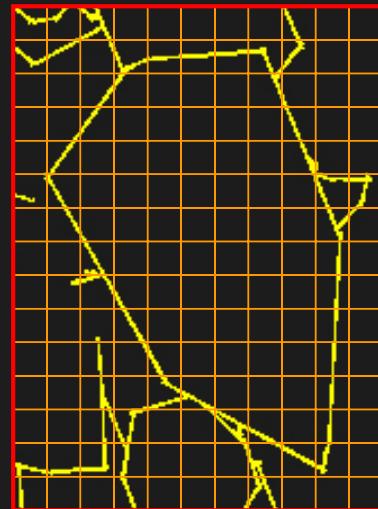


Let's use an example to understand exactly how this process works. Here is a visualization of the silhouette edges of a Knight character, seen from the point of view of the light source. We'll focus on the part outlined in red, i.e. the Knight's shield.

Compute Silhouette Points



silhouette map (dual grid)

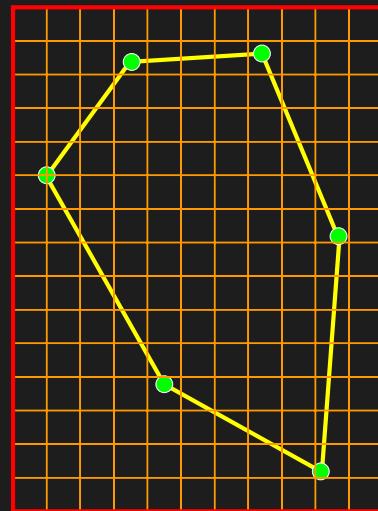


The grid lines in this image show the pixel boundaries of the silhouette map. Remember that the silhouette map is the dual to the original depth map, i.e. it's offset by $\frac{1}{2}$ pixel in both x and y.

Compute Silhouette Points



rasterization of silhouettes



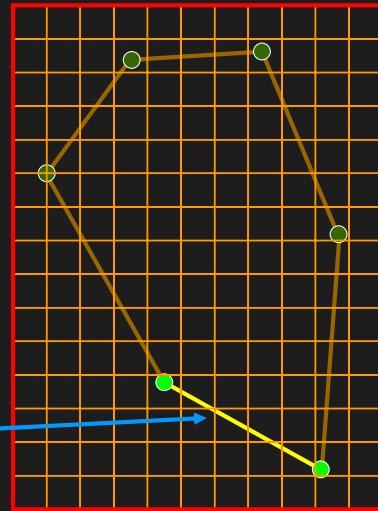
Now let's see how we rasterize the silhouette edges and wind up with silhouette points.

Compute Silhouette Points



rasterization of silhouettes

pick an edge



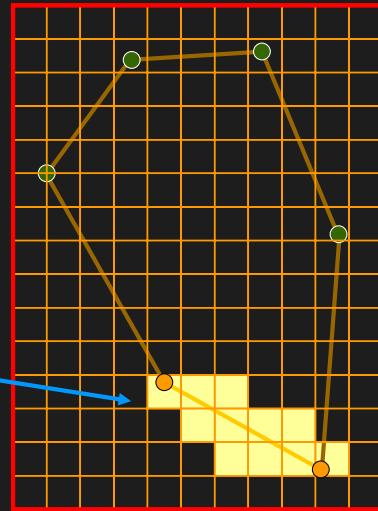
Pick any edge to start with.

Compute Silhouette Points



rasterization of silhouettes

rasterize edge conservatively:
be sure to generate fragments
for silhouette pixels



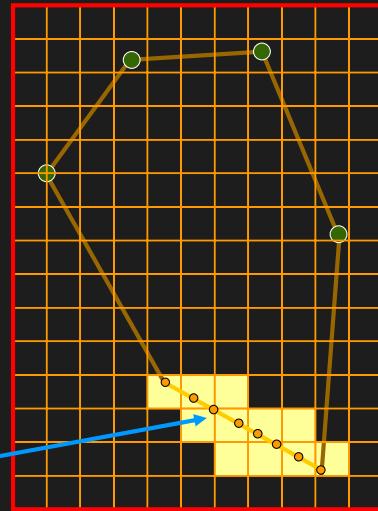
We rasterize the edge conservatively, meaning that we must guarantee that all fragments (i.e. pixels) that are crossed by the silhouette edge are rasterized. (We'll see later how to guarantee this.) The generated fragments are highlighted above. Note that since we are rasterizing conservatively, it is possible that some fragments will be generated that are not crossed by any silhouette edge. Again, later we'll see how to handle this situation.

Compute Silhouette Points



rasterization of silhouettes

for each fragment:
pick a point on the edge



Each fragment generated by the rasterizer will eventually end up in a texel in the silhouette map. For each fragment, we pick a point that lies on the silhouette edge.

Compute Silhouette Points



rasterization of silhouettes

silhouette points



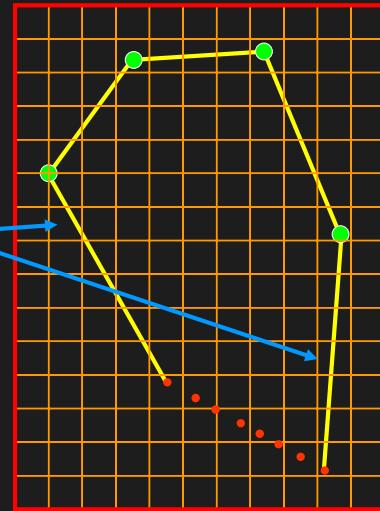
We store the coordinates of these points into the silhouette map.

Compute Silhouette Points



rasterization of silhouettes

do the same for other edges



Repeat this step for all the silhouette edges.

Compute Silhouette Points



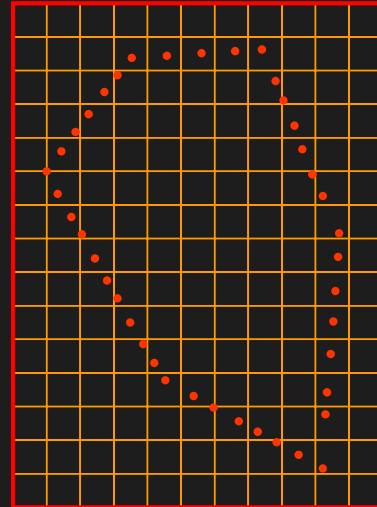
rasterization of silhouettes

completed silhouette map →

subtle issues:

- only one point per texel
- new values overwrite old ones

how to pick silhouette points?



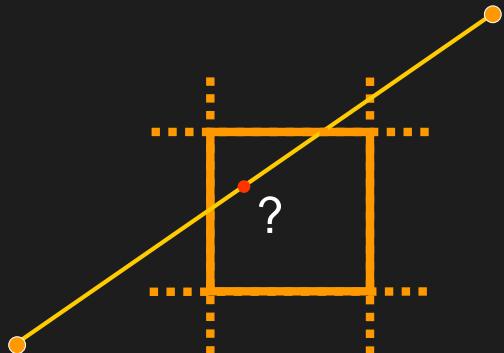
Here is a visualization of what the final silhouette map might look like after rasterizing all the silhouette edges and storing the associated silhouette points into the buffer. There are two issues to be aware of. Remember that we only store one point per texel in the silhouette map. (The reason for storing just one point is that it makes the algorithm consistent and easier to implement on graphics hardware. The alternative, storing multiple points per texel, complicates matters.) The second issue is that multiple edges may cross a single texel of the silhouette map. Since only one value may be stored per texel, we (somewhat arbitrarily) choose to let new silhouette points overwrite old ones. Thus only the last silhouette point written to a texel will be kept.

Now that we've seen the overall approach, the main question is: how do we pick the silhouette points?

Picking Silhouette Points



Pick a point on the line that lies inside the texel



Let's say we've drawn a silhouette edge, and the rasterizer has generated a bunch of fragments. For each fragment covered by the edge, we want to pick a point.

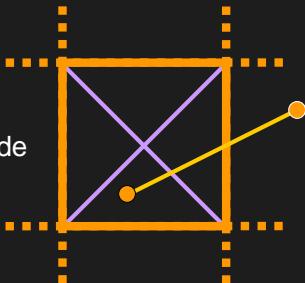
We'll break the problem of picking points into a few cases.

Silhouette Point Algorithm



Case 1:

vertex inside



In the first case, one of the endpoints of the edge (i.e. a vertex) lies within the fragment, as shown here. The thick orange lines represent the fragment boundary. Ignore the diagonal purple lines for the moment.

Silhouette Point Algorithm



Case 1:

vertex inside

pick the vertex itself

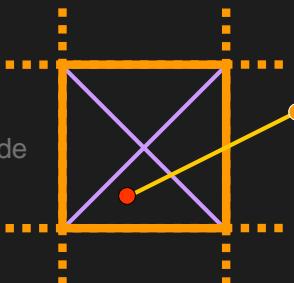
In this case, we simply pick the vertex itself as the silhouette point and store its coordinates into the silhouette map.

Silhouette Point Algorithm



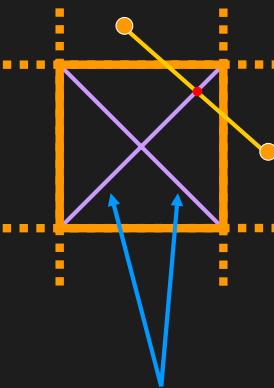
Case 1:

vertex inside



Case 2:

one intersection



test for intersection against two diagonals

If neither endpoint of the silhouette edge lies within the fragment, we need to check for intersections. One way to do this is to perform a line segment intersection test against the two diagonals, shown in purple. Clearly, the silhouette edge will intersect the fragment if and only if it intersects at least one of the diagonals.

In the second case, suppose the silhouette edge intersects only one of the diagonals.

Silhouette Point Algorithm



Case 1:

vertex inside

pick the intersection point itself

Case 2:

one
intersection

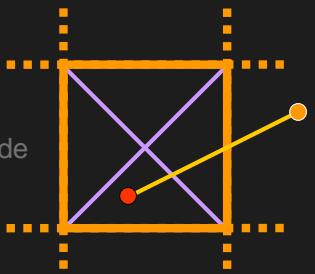
For this case, we pick the intersection point itself as the silhouette point to be stored.

Silhouette Point Algorithm



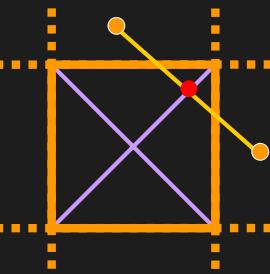
Case 1:

vertex inside



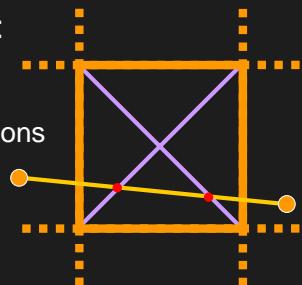
Case 2:

one intersection



Case 3:

two intersections



In the third case, there are two intersections, one with each diagonal.

Silhouette Point Algorithm



Case 1:

vertex inside

Case 2:

one intersection

Case 3:

two intersections

use midpoint

In this situation, we pick the midpoint of the two intersections as the silhouette point.

Silhouette Point Algorithm



Case 1:

vertex inside

Case 3:

two intersections

Case 2:

one intersection

Case 4:

no intersections

Finally, it is possible that there is no intersection between the silhouette edge and the fragment. This is possible because we are rasterizing the silhouette edge conservatively, and thus such fragments may be generated by the rasterizer.

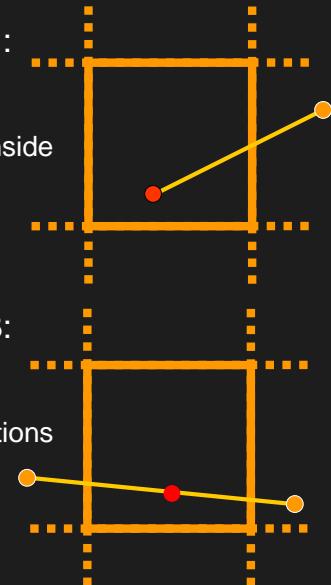
In this case, nothing is written to the silhouette map for this fragment.

Silhouette Point Algorithm



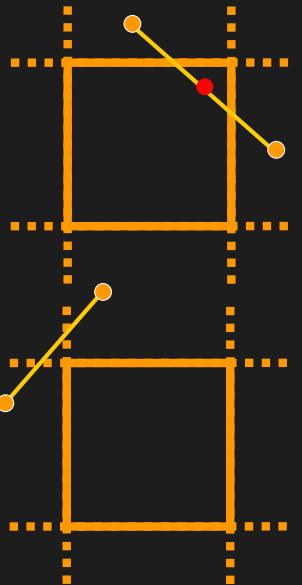
Case 1:

vertex inside



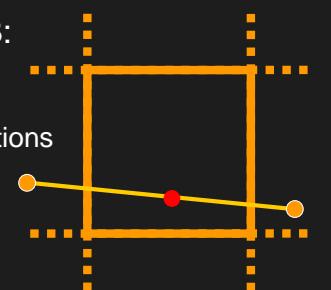
Case 2:

one intersection



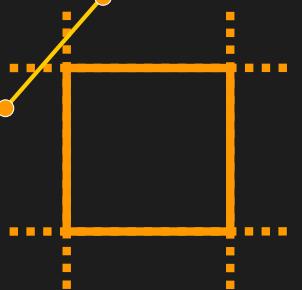
Case 3:

two intersections



Case 4:

no intersections



That's all there is to the silhouette-point-picking algorithm. In summary, we rasterize the silhouette edges (conservatively) and for each fragment generated, we perform the four tests shown here.

Render scene



How to compute shadows?

Split problem into two parts:

- non-silhouette pixels: use shadow map
- silhouette pixels: use silhouette map

In the final step (third rendering pass) of the algorithm, we draw the scene from the observer's viewpoint and compute shadows. How do we use the information gathered so far (a depth map and a silhouette map) to compute these shadows?

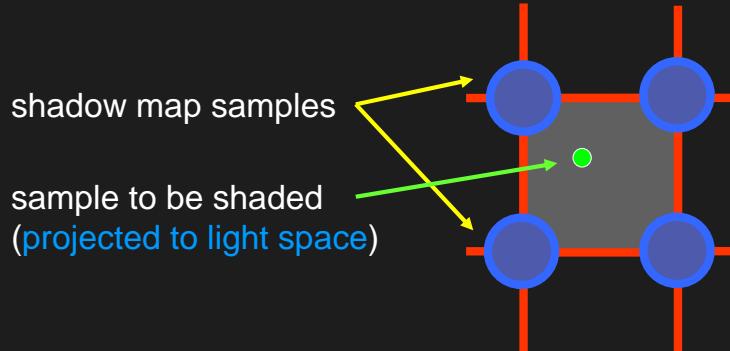
A conceptually simple way to think about the problem involves splitting the problem into two parts. Let's use the term "silhouette pixels" to refer to the pixels in the final image (seen from the observer's view) that contain shadow discontinuities, i.e. the boundary between shadowed and illuminated regions. Recall that with standard shadow maps, these pixels give us the most grief because they are precisely the ones that exhibit aliasing artifacts. All other pixels in the scene look fine because they are completely illuminated or completely in shadow.

Therefore, let's consider silhouette pixels and non-silhouette pixels separately. Since non-silhouette pixels don't show aliasing artifacts, we'll use the standard shadow map to compute shadows for those pixels. On the other hand, for the silhouette pixels, we'll use the silhouette map to obtain a good, piecewise-linear reconstruction of the shadow silhouette. We'll see exactly how to do this in a moment.

Find Silhouette Pixels



- Project sample into light space
- Compare depth against 4 nearest samples in shadow map

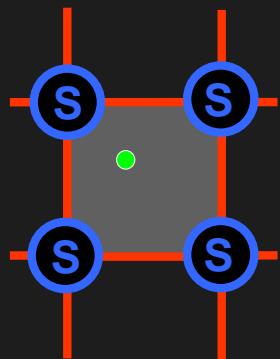


First, we need a way to identify silhouette pixels. This task turns out to be surprisingly easy. Let's say we have a sample (pixel) in the final image, and we want to know whether it's a silhouette or non-silhouette pixel. We transform the pixel into light space, just as we would do when using the standard shadow map algorithm. In general, the transformed sample (shown in green in the diagram) will lie between four samples of the shadow map (shown in blue). Compare the depth of the sample against the depths associated with the four adjacent samples of the shadow map.

Find Silhouette Pixels



results agree:
non-silhouette pixel



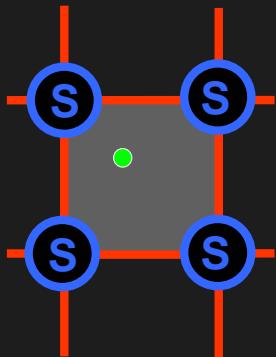
If all the depth comparison results agree, then the sample is a non-silhouette pixel. In the example shown here, all the depth comparison results indicate that the sample is in shadow (S).

Find Silhouette Pixels



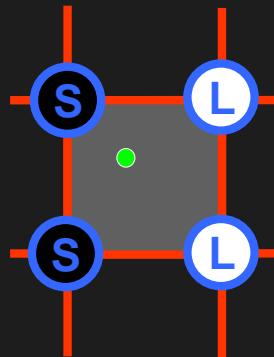
Case #1

results agree:
non-silhouette pixel



Case #2

results disagree:
silhouette pixel



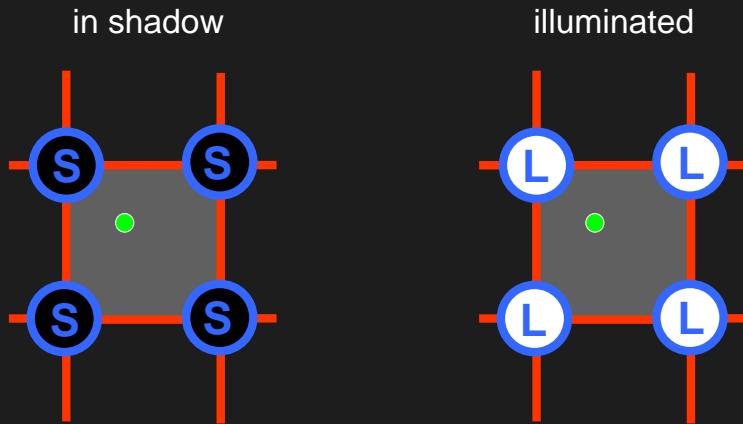
On the other hand, if the depth comparison results disagree, then the sample is a silhouette pixel. In the example here, two of the four tests declare that the sample is in shadow, but the other two declare that the sample is illuminated.

(You may have noticed this technique is very similar to the idea of Percentage Closer Filtering [Reeves et al. 1987].)

Treat Non-Silhouette Pixels



Easy: use depth comparison result

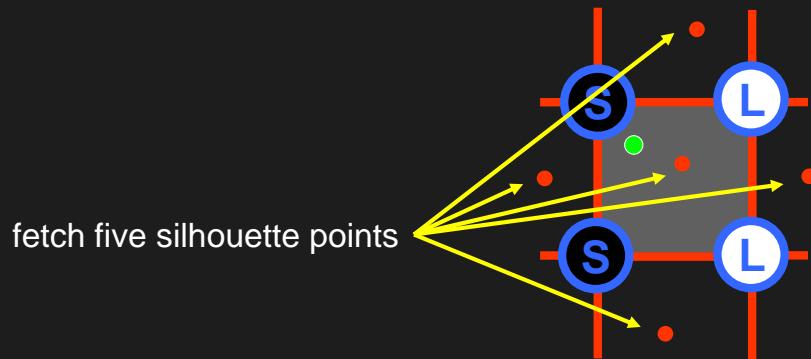


For non-silhouette pixels, computing shadows is easy. We just use the results of the depth comparisons to shade the sample. If all results say the sample is in shadow (left image) then the sample is in shadow. Similarly, if all results say the sample is illuminated (right image), then the sample should be illuminated.

Treat Silhouette Pixels



Reconstruct edge using silhouette map



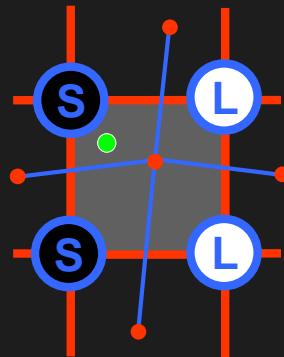
The more interesting part is handling silhouette pixels. In this case, we rely on the silhouette map to help us reconstruct shadow edges. First, transform the sample to light space (just as in the previous step) and lookup the current silhouette point and the four neighbors (shown as red points in the diagram). This essentially amounts to five texture lookups.

Treat Silhouette Pixels



Reconstruct edge using silhouette map

splits cell into four quadrants



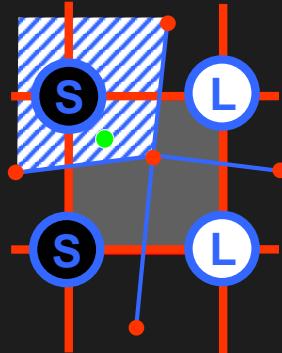
If we imagine drawing line segments between these silhouette points as shown, we see that the segments split up the texel into four quadrants.

Treat Silhouette Pixels



Shade sample according to quadrant

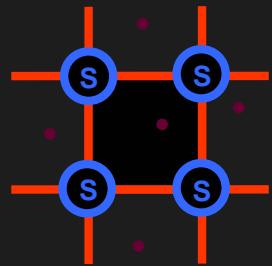
example: sample in shadow



Find the quadrant that contains the sample and shade the sample according to the depth comparison result of the associated depth sample. In the above example, the top-left quadrant contains the sample, and the depth sample associated with that quadrant indicates the sample should be in shadow. Thus we render this sample as being in shadow. If the sample had instead fallen into the top-right quadrant, it would have been illuminated.

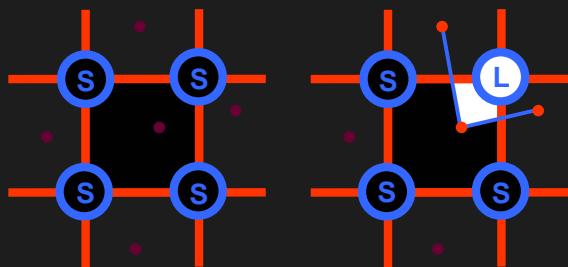
Hopefully it is clear from the diagram why we obtain a piecewise-linear approximation to the shadow edge, as opposed to the previous piecewise-constant approximation (using a regular shadow map). The quadrants essentially define different shadow boundaries within a single texel, i.e. at sub-texel precision. All samples that fall into a given quadrant will be shaded the same. In contrast, with a regular shadow map, all samples that lay within the texel itself are shaded the same.

Six Combinations (1 of 6)

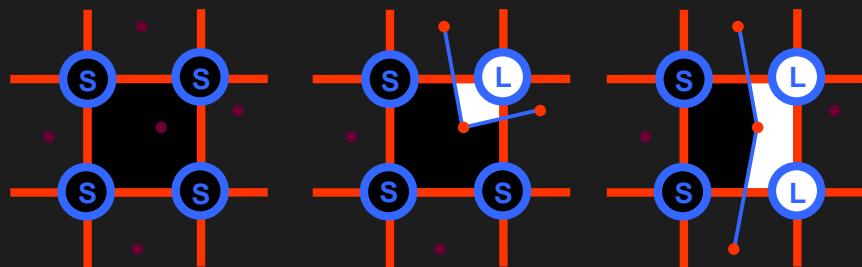


For clarity, let's consider all the possible cases that can arise. There are only six of them. The first possibility, shown here, is when all four depth samples indicate the current sample (i.e. the sample to be shaded) is in shadow. (This actually falls into the case of the non-silhouette pixels, which we covered earlier.)

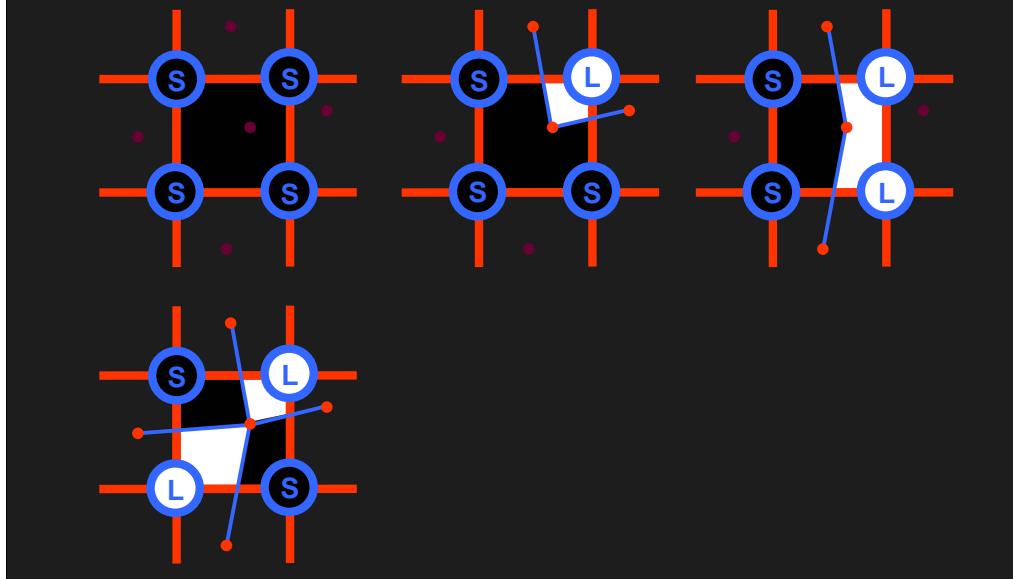
Six Combinations (2 of 6)



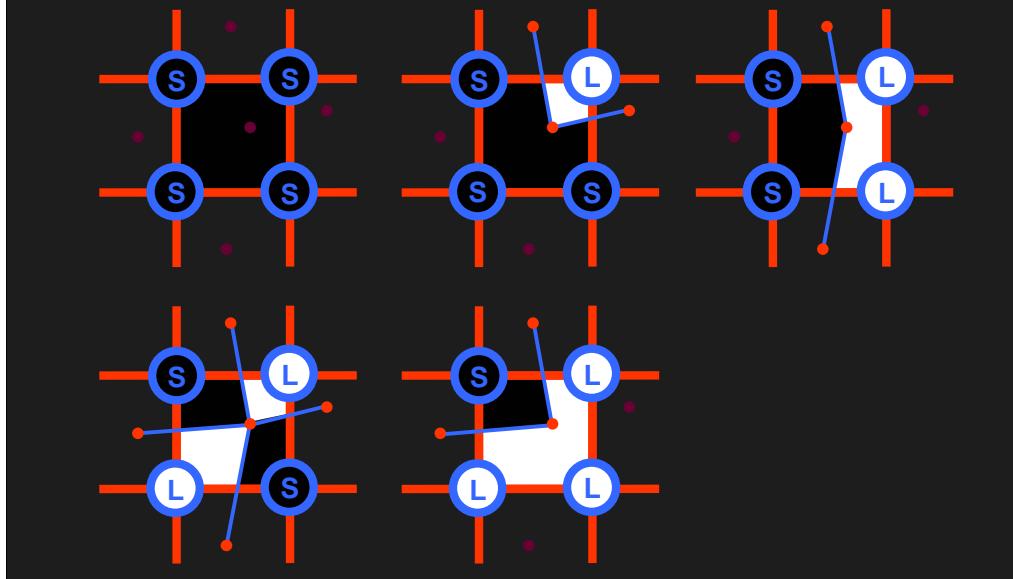
Six Combinations (3 of 6)



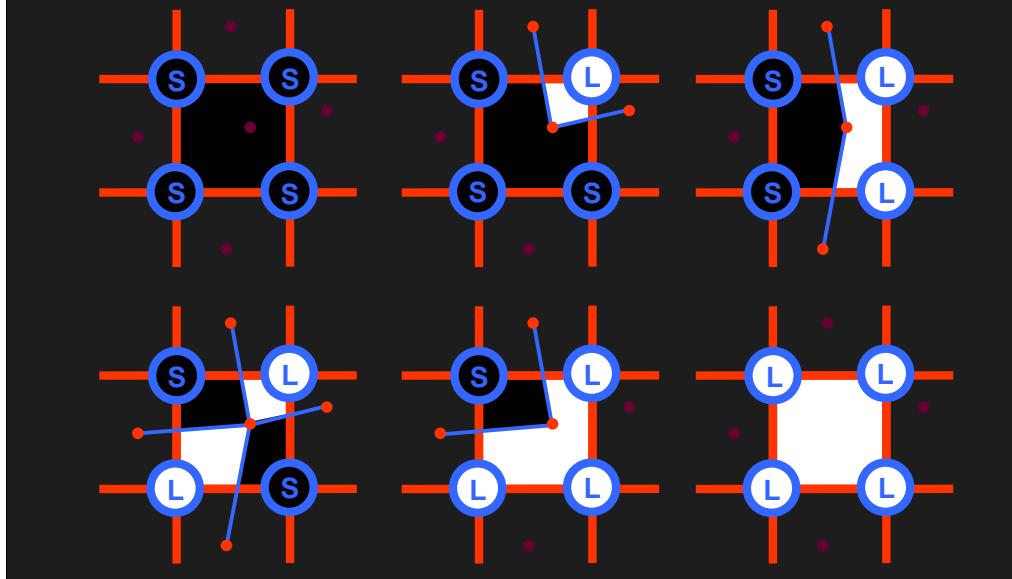
Six Combinations (4 of 6)



Six Combinations (5 of 6)



Six Combinations (6 of 6)



In the final case, all depth comparisons agree and the sample is illuminated. Again this falls into the case of non-silhouette pixels.

In summary, there are 2 cases (top-left, bottom-right) for non-silhouette pixels and 4 cases for silhouette pixels.

Algorithm Recap

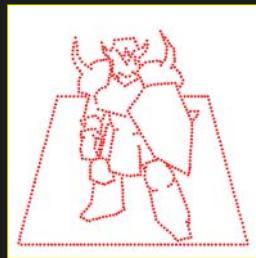


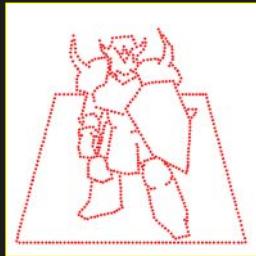
Image-space algorithm



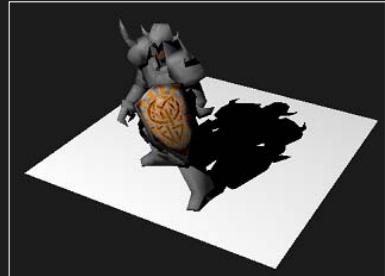
We have now covered all the algorithm's details. Let's take a step back and review the algorithm at a higher-level.

First, note that the algorithm works in image space. All relevant information (depth samples, silhouette points) are stored using 2D image representations. The silhouette map is really just an image-based edge representation.

Algorithm Recap (1 of 3)



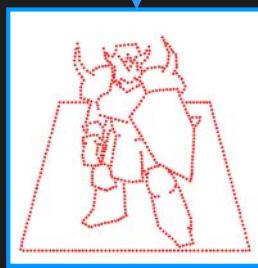
Create depth map



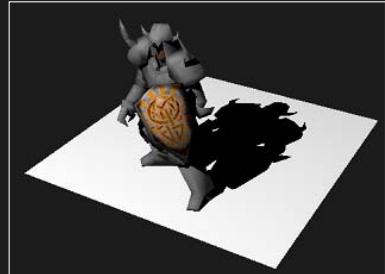
Easy: just like regular shadow map

The first step is to create a depth map from the light's viewpoint.

Algorithm Recap (2 of 3)



Create silhouette map

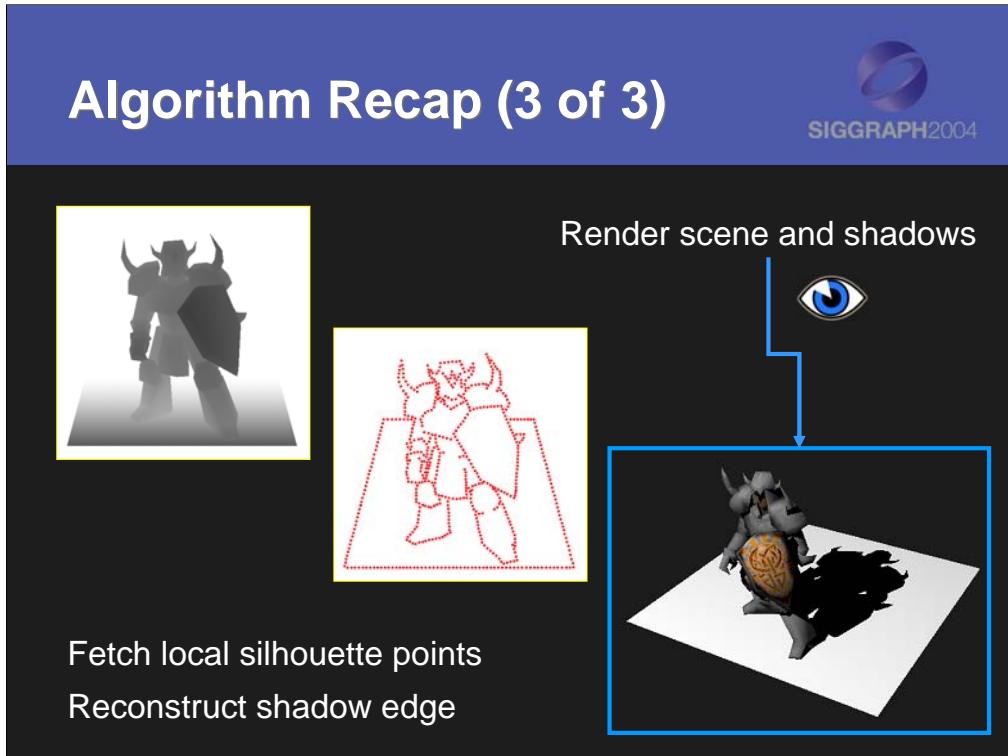


Rasterize silhouette edges

Pick silhouette points, 1 per texel

The second step is to create the silhouette map, also from the light's viewpoint. During this step, we rasterize silhouette edges conservatively and pick points that lie exactly on the edge and store the coordinates of these points into the silhouette map. The idea is to check for edge-fragment intersections by checking against the two diagonals of the fragment. There are a few simple cases to consider, as described in detail earlier.

Algorithm Recap (3 of 3)



In the final pass, we draw the scene from the observer's viewpoint and draw the shadows. For non-silhouette pixels, just use the shadow map. For silhouette pixels, we fetch five silhouette points (current point + four neighbors) and use these points to reconstruct a piecewise-linear approximation to the true shadow silhouette.



Implementation

Up till now, we've been discussing the concepts of the algorithm. It's time to see how we can implement the algorithm on modern graphics hardware.

Implementation



- Details ([OpenGL](#))
- Hardware acceleration
- Optimizations

The silhouette map 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 a number of tasks we have to perform. For instance, we need to perform intersection tests when generating the silhouette map. 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

Optimizations

- for closed models, cull back faces
- turn off shading, color writes
- only send vertex positions
- draw roughly front-to-back



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. Keep in mind there are a number of optimizations that we can perform. For closed models, turn on back-face culling, e.g.

```
glEnable(GL_CULL_FACE);
```

since those faces won't be seen anyways. In addition, since we only care about drawing depth values, we don't have to perform shading. Therefore, turn all fancy shaders off. Furthermore, we don't even have to write anything to the color buffer, so turn off color writes:

```
glColorMask(0, 0, 0, 0);
```

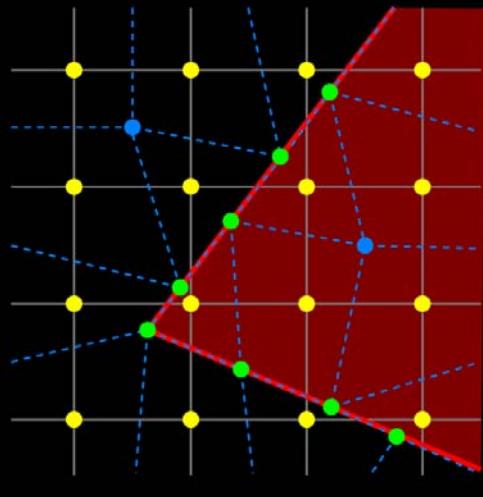
In terms of transferring data from the host processor (CPU) to the graphics processor (GPU), we only need to send the vertex positions. Since we're not doing any shading, don't send extra information like texture coordinates and normals.

Finally, draw the objects roughly in front-to-back order. Doing so maximizes the hardware's ability to perform early Z rejection (i.e. occlusion culling).

Create Silhouette Map



Goal: store points that lie on silhouette

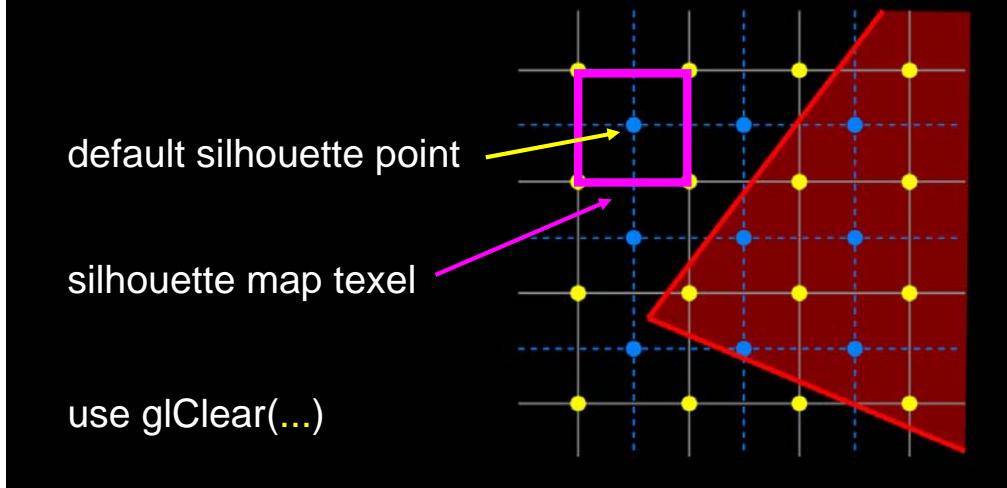


Now let's see how to compute the silhouette points in the second step of the algorithm.

Create Silhouette Map



Place default point at texel center



Remember that the silhouette map is the dual grid of the depth map and is offset from the depth map by $\frac{1}{2}$ a pixel. Earlier, we discussed the concept of having a deformed depth mesh in which the depth samples are moved to lie along silhouettes. Our strategy for creating the silhouette map will be as follows. First, let's start with an undeformed depth mesh, meaning that all depth samples lie at their original, undeformed positions. Then we'll rasterize silhouette edges and compute silhouette points to perform the deformation on some of the depth samples.

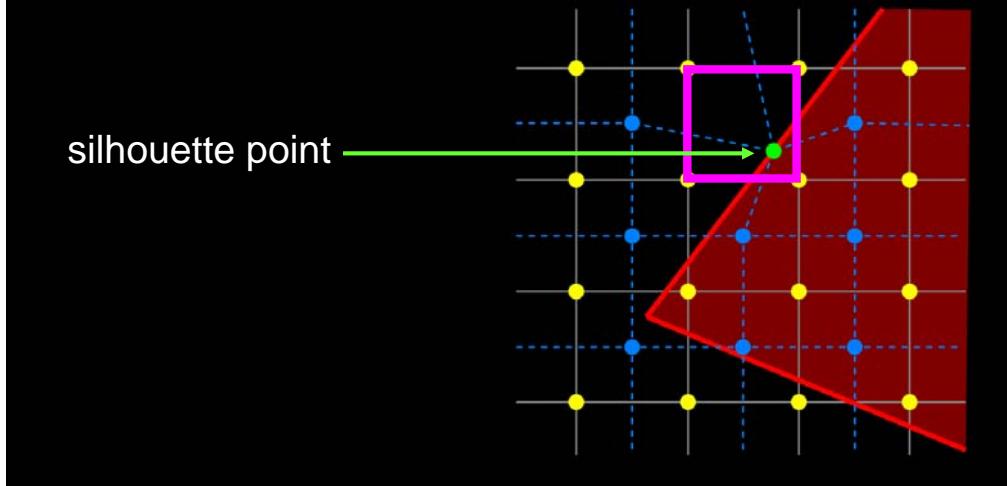
To implement this, we start by placing a default silhouette point at the center of every texel of the silhouette map. This essentially places the silhouette points directly on top of the depth samples of the depth map. This construction is shown in the diagram. The purple box represents the boundary of a single texel of the silhouette map. The blue dots are the locations of the depth samples. We initialize the silhouette map by placing silhouette points at these blue points.

In practice, it's easy to perform this initialization by using the `glClear` call to clear the whole buffer.

Create Silhouette Map



Fragment program finds silhouette points



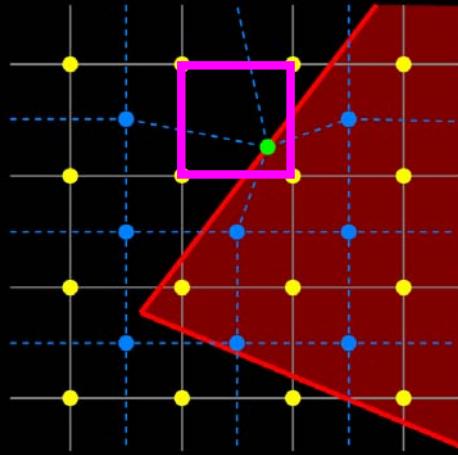
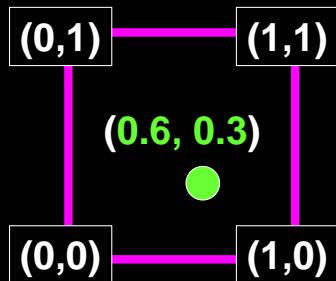
We'll use a fragment program (pixel shader) to compute the silhouette points for texels that contain silhouette edges. To make this concrete, consider the purple texel shown in the diagram. A fragment associated with this texel will be generated by the rasterizer. We want a fragment program that checks that a silhouette edge passes through this fragment, computes the silhouette point, and writes it to the output buffer (the silhouette map).

Create Silhouette Map



Fragment program finds silhouette points

- use local coordinates
- store only xy offsets



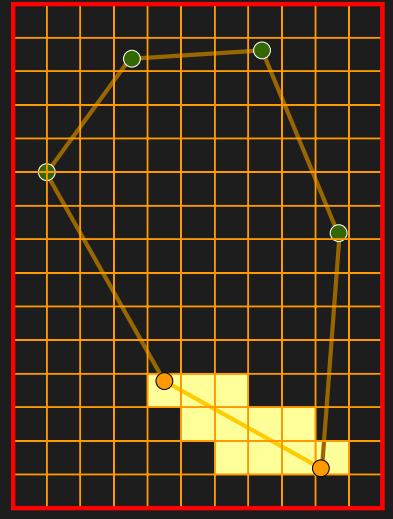
The fragment program simply performs the various intersection tests discussed earlier in the silhouette-point-picking algorithm. For fragments that are generated by the rasterizer but not crossed by a silhouette edge, the fragment program uses a “fragment kill” to throw away the fragment (i.e. write nothing to the output).

One way of computing and storing the silhouette points is to use a local coordinate system, in which the texel area is taken to be a unit square (see the lower-left diagram). The default silhouette point is at the center, (0.5, 0.5). For silhouette points computed via intersection tests, we just store xy offsets in the local coordinate system into the silhouette map. These offsets are in the range [0,1].

Rasterizing Silhouettes

Two issues:

- must guarantee generation of silhouette pixels
- discard occluded silhouettes



There are two issues to be aware of when rasterizing silhouette edges. All along we've talked about performing conservative rasterization to guarantee that all fragments crossed by a silhouette edge will be generated. Now we'll see how to do that. The second issue is that some silhouette edges, seen from the light's viewpoint, will be hidden by blockers (occluders). Since the fragments from these edges aren't seen by the light, we don't want to process them.

Rasterizing Silhouettes



Rasterize conservatively

- Be careful using OpenGL wide lines
- Use width of at least 3

glLineWidth(3);

- Make lines slightly longer to cover endpoints

Another solution: use thin quads, not lines

- See Sen et al. [SIG2003] paper

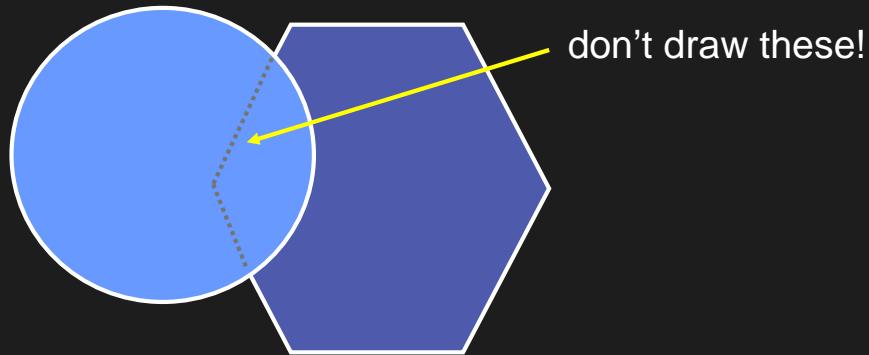
Let's address the conservative rasterization first. Sen et al., in their original paper on shadow silhouette maps, recommend drawing “thin quads” – thick enough to guarantee the generation of all fragments crossed by a silhouette. Their reason for doing so is that the alternative, drawing wide lines in OpenGL, may vary in behavior across different graphics hardware.

It turns out, however, that by choosing a line width that is large enough, all necessary fragments will in fact be generated. In practice, I've found that using a width of at least 3 works consistently. Another detail to remember is that you have to make the line slightly longer than the original edge to guarantee that the fragments containing the endpoints of the edge will also be generated.

Occluded Silhouette Pixels



Example:



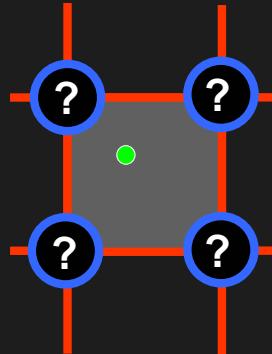
The second issue is dealing with occluded silhouette pixels. In the example shown here, the hexagon is partly occluded by the circle, seen from the point of view of the light source. The fragments belonging to the occluded silhouette edges, shown as dotted gray lines, should be ignored.

Occluded Silhouette Pixels



Implementing occlusion:

- Use depth map from first pass
- Recall silhouette map offset by $\frac{1}{2}$ pixel
- Use fragment kill if depth is greater than 4 nearest samples in depth map



This case is easy to check for, because we already have a depth map of the blockers from the first rendering pass. In the fragment program that computes silhouette points, we also perform the following check. If the depth of a fragment belonging to a silhouette edge lies behind all four of the neighboring depth samples in the shadow map, then this fragment is occluded and should be discarded. To throw away the pixel, issue a fragment kill.

Rendering Final Image



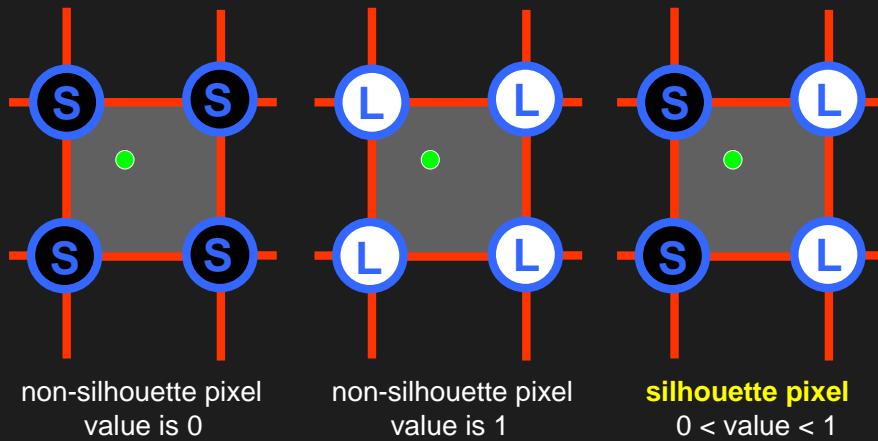
Recall

- Draw from observer's view
- Identify silhouette vs. non-silhouette pixels
- Use shadow map for non-silhouette pixels
- Use silhouette map for silhouette pixels

In the final rendering pass, we need to distinguish between silhouette and non-silhouette pixels. Just as a reminder, earlier we saw how to accomplish this by transforming a sample into light space and checking its depth against the 4 nearest samples of the shadow map. If the depth comparison results agree, then the pixel is a non-silhouette pixel. Otherwise, it's a silhouette pixel.

Identify Silhouette Pixels

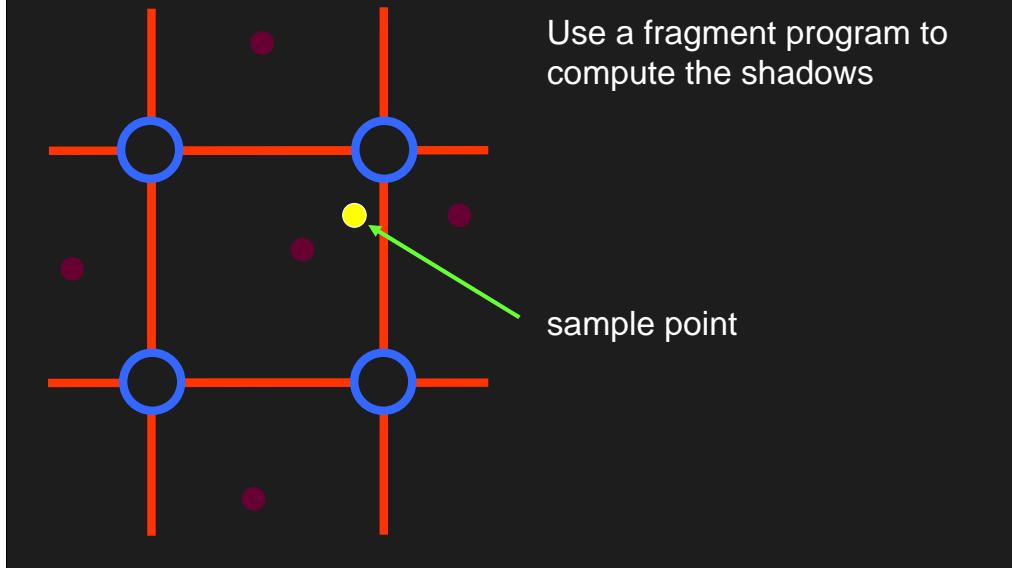
- Take advantage of hardware shadow mapping
- Use percentage closer filtering



Implementing this step is a breeze using graphics hardware. The hardware supports percentage closer filtering (see Reeves et al. [1987]), meaning that instead of performing a single depth comparison of a sample against the depth map, it actually compares the depth against the 4 nearest depth samples and filters the binary results. This means that if the depth comparison results agree, then the final result will be either 0 (for a shadowed pixel) or 1 (for an illuminated one). In contrast, if the depth comparison results disagree, then, since the results are filtered, the final value will lie in between 0 and 1.

The nice thing is that this entire operation can be performed using a single shadow map (texture) lookup in a fragment program. The hardware takes care of performing the depth comparisons and returns the filtered result to your fragment program. This is both simple and fast.

Silhouette Reconstruction

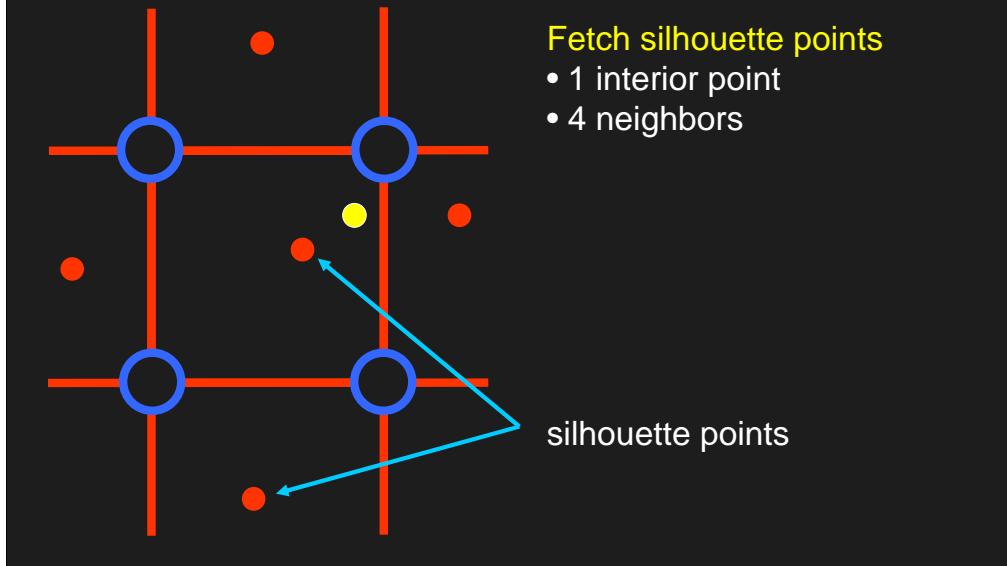


Use a fragment program to compute the shadows

sample point

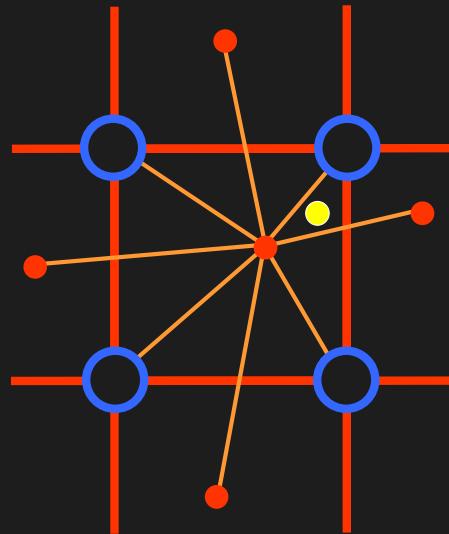
For silhouette pixels, we also use the fragment program to perform accurate shadow edge reconstruction. Let's see exactly how it works.

Silhouette Reconstruction



First, project the sample into light space, which maps the sample to a particular texel in the silhouette map. We fetch the silhouette points from the silhouette map, 1 point for the current texel, and its four immediate neighbors. This amounts to five texture fetches.

Silhouette Reconstruction



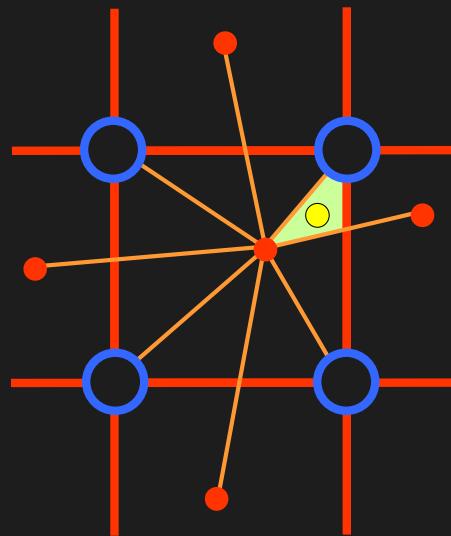
Fetch silhouette points

- 1 interior point
- 4 neighbors

Create eight wedges

Now, recall that we want to know which of the four neighboring depth samples should be used for the depth comparison. Imagine using the five silhouette points and the four depth samples to carve up the texel into eight wedges (shown in orange in the diagram).

Silhouette Reconstruction



Fetch silhouette points

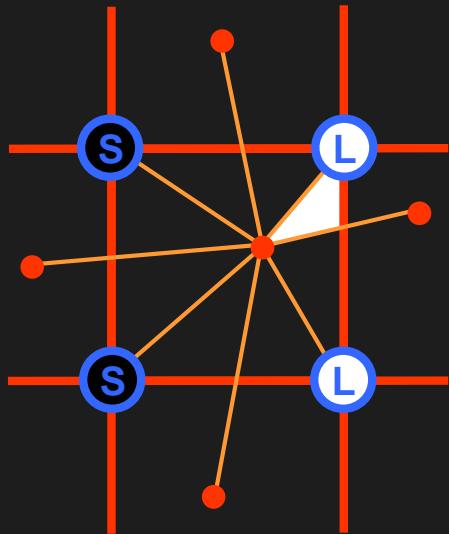
- 1 interior point
- 4 neighbors

Create eight wedges

Find enclosing wedge
• point-in-triangle tests

Then we simply find which wedge contains our sample. This amounts to performing point-in-triangle tests.

Silhouette Reconstruction



Fetch silhouette points

- 1 interior point
- 4 neighbors

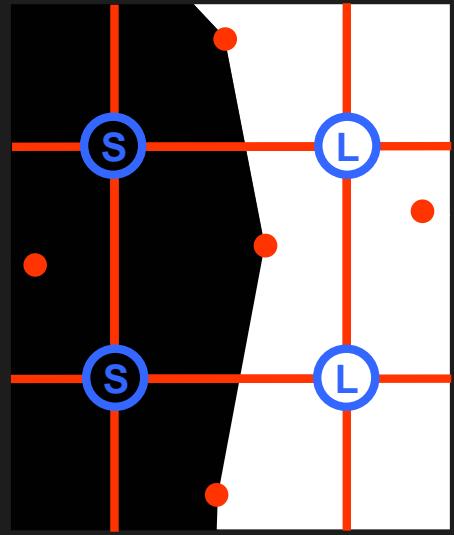
Create eight wedges

Find enclosing wedge
• point-in-triangle tests

Shade the sample using
wedge's depth test result

We shade the sample according to the depth comparison result associated with the wedge. In the example here, the relevant wedge is associated with the top-right depth sample, and the depth comparison against that sample indicates that the sample should be illuminated.

Silhouette Reconstruction



Fetch silhouette points

- 1 interior point
- 4 neighbors

Create eight wedges

Find enclosing wedge
• point-in-triangle tests

Shade the sample using
wedge's depth test result

Repeat for all samples

Repeat this step for all samples in the image. That's all there is to computing the shadows.

Optimizations



Fragment program is expensive

- lots of arithmetic
- lots of texture reads (**5 silhouette points**)

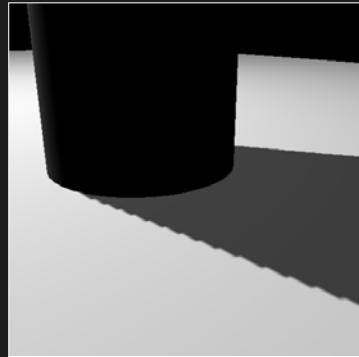
However, only required for silhouette pixels!

Now let's discuss some potential optimizations when computing the shadows. Using the silhouette map for silhouette pixels is rather expensive because it requires 5 texture reads (to gather the silhouette points) and a lot of arithmetic to perform the point-in-wedge tests. The good news, however, is that this extra work is required only for silhouette pixels.

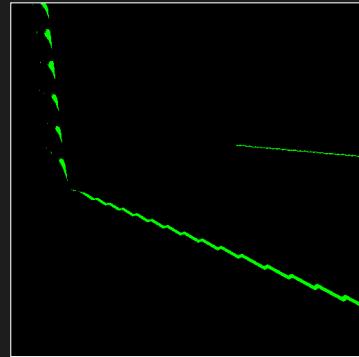
Optimizations



Very few silhouette pixels in practice



original scene



silhouette pixels
(1% total image)

In practice, the number of silhouette pixels accounts for only a small fraction of the total number of pixels in the image. This example shows a cylinder casting a shadow onto the ground plane. The number of silhouette pixels (shown in green on the right) occupy less than 1% of the total image!

Optimizations



Use fragment program branching

- Potentially huge performance wins
- Only available in latest hardware

To take advantage of this observation, we simply use if/else branching in a fragment program. Most of the pixels are non-silhouette pixels, so the branching will enable us to skip the 5 texture lookups for gathering the silhouette points and all the point-in-wedge arithmetic.

Keep in mind, however, that branching in fragment programs is a very recent addition to graphics hardware, and at the time of this writing is supported only by the NVIDIA GeForce 6 series graphics cards.



Examples and Analysis

Now let's take a look at some examples and comparisons between various shadow algorithms.

Example 1



shadow maps



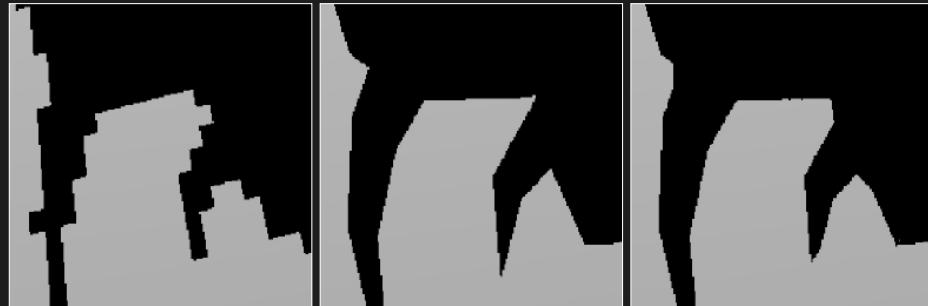
shadow volumes



silhouette maps

Here is a scene with the Knight character casting shadows on the ground plane. On the left is the result obtained using shadow maps. Aliasing artifacts are apparent. Shadow volumes generate accurate shadows, as shown in the middle image. The result on the right is obtained using shadow silhouette maps. Notice that it dramatically reduces aliasing artifacts, and the image is very similar to the middle image.

Example 1 (closeup)



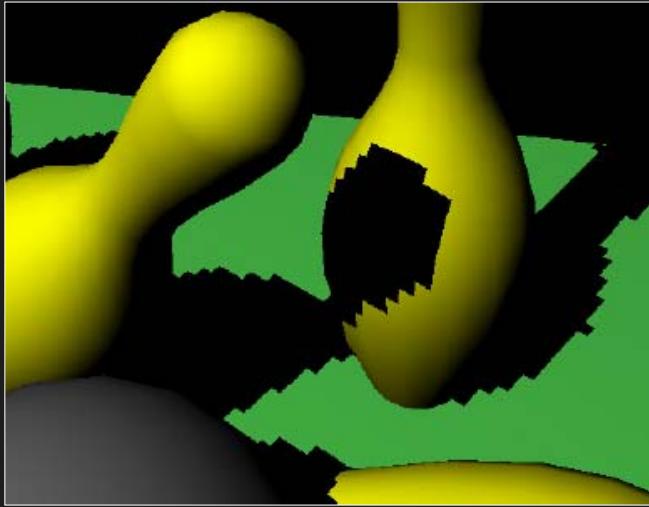
shadow maps

shadow volumes

silhouette maps

Here's a closeup of the same scene.

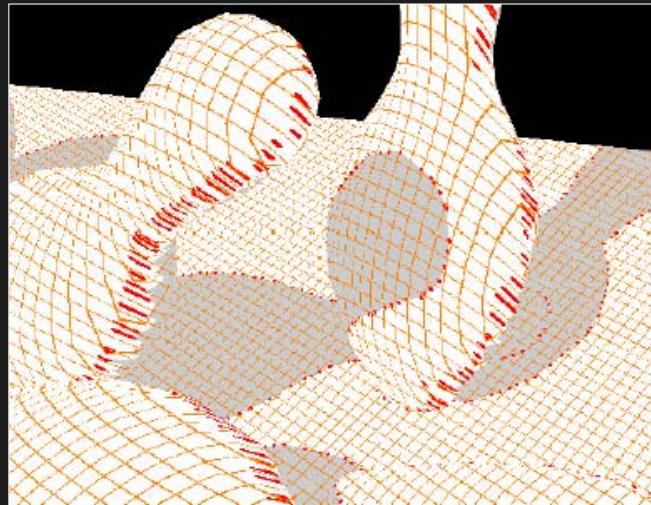
Example 2



shadow maps

Here's a second example with bowling pins (yellow) casting shadows onto each other and the ground plane. Ordinary shadow maps lead to aliasing artifacts.

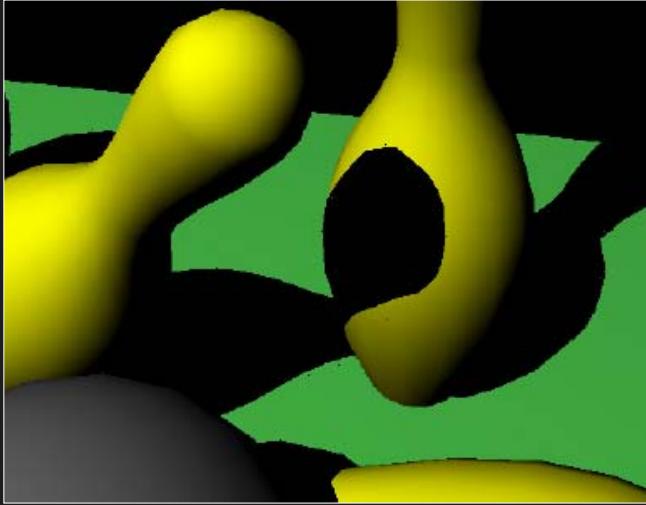
Example 2



projected silhouette map

Here's a visualization of the silhouette map, projected from the point of view of the light source onto the scene.

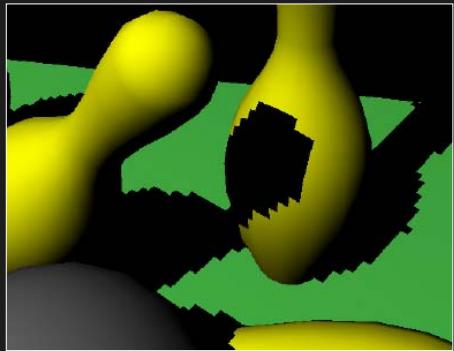
Example 2



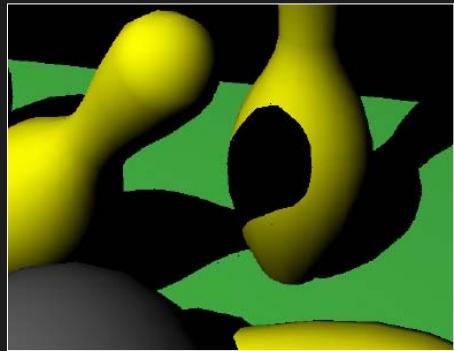
shadows using silhouette map

Here are the resulting shadows computed using a silhouette map.

Quality Comparison



shadow map

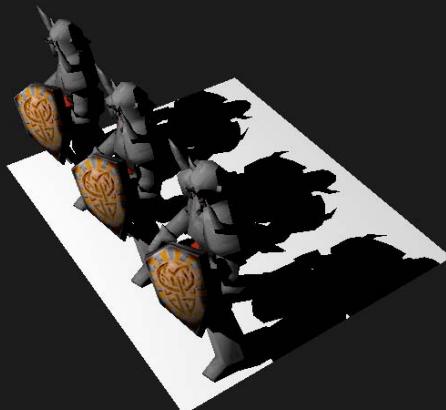


silhouette map

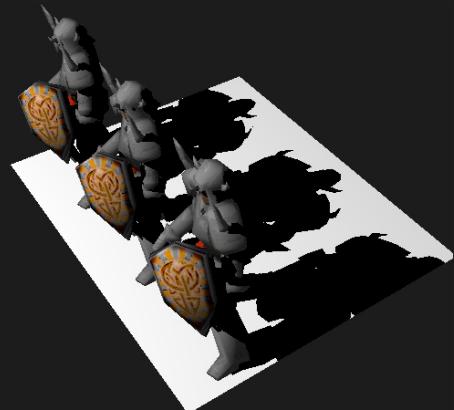
Bandwidth Comparison



shadow volumes



silhouette maps



One of the advantages of silhouette maps over shadow volumes is that they consume far less bandwidth. Consider this example with three knights standing on the ground plane.

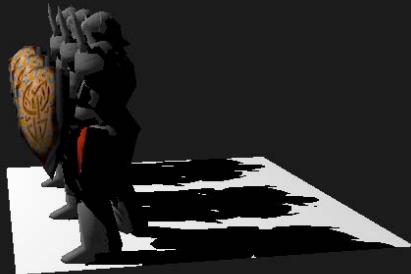
Bandwidth Comparison



shadow volumes



silhouette maps

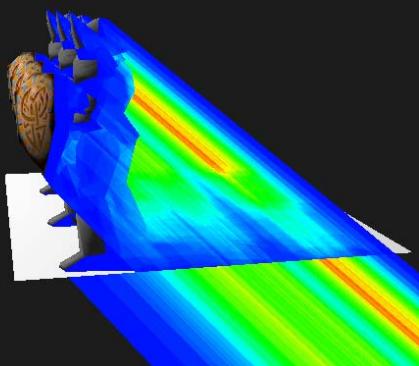


Here's a view of the scene from the side.

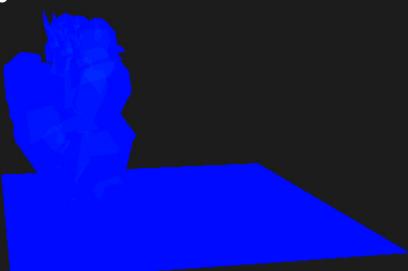
Bandwidth Comparison



shadow volumes



silhouette maps



Here's a visualization of the amount of overdraw when rendering the scene using the two algorithms. Dark blue regions indicate low overdraw, whereas red and yellow regions indicate high overdraw. The extra polygons in the left image show the shadow volume polygons rasterized from the observer's point of view. Clearly, the shadow volumes consume far more fillrate and bandwidth than silhouette maps.

Bandwidth Comparison



1200 triangles



14,800 triangles



Shadow volumes

5.94 MB

Silhouette maps

1.53 MB

126.3 MB

1.07 MB

Bandwidth ratio

3.9 : 1

118:1

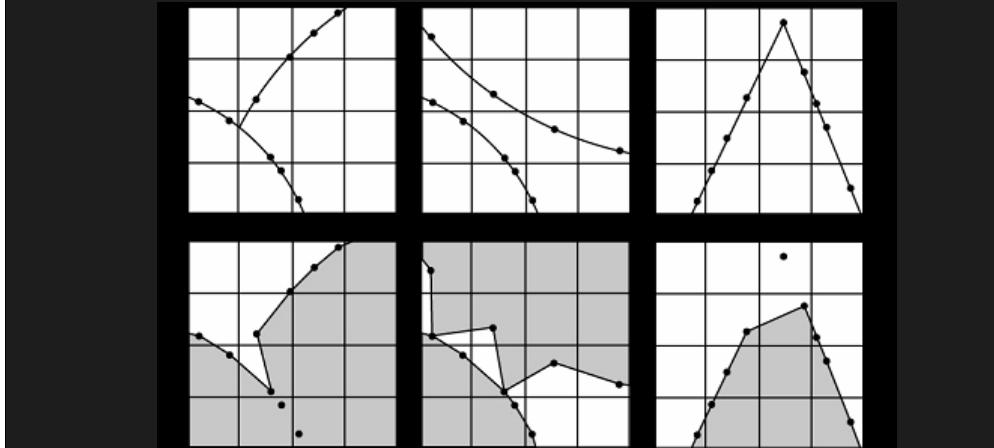
To make this more quantitative, here are two test scenes that compare bandwidth usage between the two algorithms. In both cases, silhouette maps consume far less bandwidth than shadow volumes.

Keep in mind, however, that bandwidth usage does not translate directly to performance. Shadow volumes perform many operations, but each of those operations (a stencil update) is relatively simple. In contrast, the operation performed on each pixel for silhouette maps can be rather complex. The actual performance differences between the two algorithms is highly scene-dependent and will clearly vary on a case-by-case basis.

Artifacts



- Silhouette map: one point per texel
- Multiple edges inside a texel

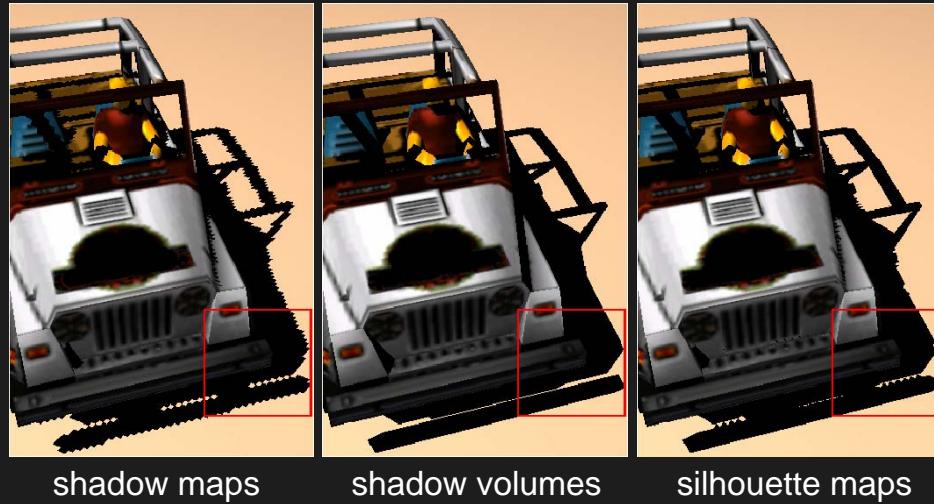


Now let's take a look at some of the artifacts that can arise with the shadow silhouette map algorithm. We should expect to get artifacts in some cases because, after all, we are sampling the scene with a discrete buffer, limited in size by our choice of resolution for the depth map and silhouette map.

I promised earlier that I would discuss the implications of storing only one silhouette point per texel in the silhouette map. The silhouette map provides a reasonable approximation as long as only one silhouette edge passes through the texel. The main problem occurs when you have multiple, different silhouette edges that pass through the texel, as shown in the three cases above. Each column shows a different situation where artifacts can occur. In the left column, two curves meet at a T-intersection. One of the texels contains the T-intersection and the two edges, but only one point can be stored. Since there is no explicit knowledge about the T-intersections (since silhouette edges are found and rasterized independently of each other), the choice of silhouette point is rather arbitrary. In this case, a silhouette point is chosen for the lower curve, and information about the upper curve is lost. This leads to the shadow reconstruction artifact shown in the bottom-left image.

Another problematic situation is when you simply have two curves that pass near each other without touching. These may be silhouettes belonging to completely different objects. Again, texels may be crossed by two or more silhouette edges, but ultimately only one silhouette point is stored, so information about one of the curves is lost. This leads to the zig-zag reconstruction artifacts shown in the bottom-center image.

Artifacts



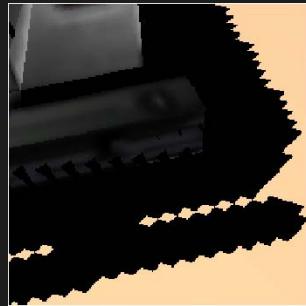
To summarize, the silhouette map algorithm can lead to artifacts whenever multiple silhouette edges cross a texel. Clearly this is related to the silhouette map resolution. The lower the resolution, the more likely that multiple edges will cover a given texel. Also, scenes with fine geometry tend to have silhouette edges that are close to one another.

In this jeep scene, artifacts can be seen in the shadow cast by the jeep's fender onto the ground plane.

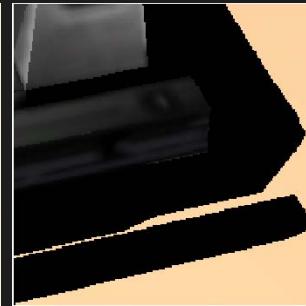
Artifacts (closeup)



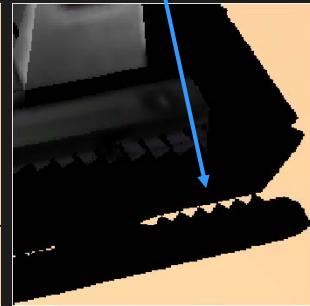
Artifacts due to multiple edges
More noticeable when animated



shadow maps



shadow volumes



silhouette maps

Here's a closeup of the artifacts. Note that these artifacts are generally more visible in animations, because the shadow artifacts tend to "pop" abruptly depending on how the image samples get projected onto the silhouette map. A more severe version of popping occurs with regular shadow maps.

Algorithm Comparison

Perspective Shadow Maps:

- same generality as shadow maps
- minimal overhead (2 passes)
- doesn't address aliasing in all cases

Shadow Silhouette Maps:

- addresses aliasing more generally
- more overhead (3 passes + big shaders)
- less general than shadow maps

Now that we've seen both the perspective shadow map and silhouette map techniques, let's compare these two methods qualitatively. Perspective shadow maps require minimal changes to the original shadow map method; conceptually, they just involve an extra perspective transform. Thus they require only 2 rendering passes and have the same level of generality as regular shadow maps: they automatically handle any geometry that can be represented in a depth buffer, such as polygonal models, points, sprites, and so on. However, they do not fix aliasing in all cases. In particular they do not solve projection aliasing, and they also cannot solve perspective aliasing for all scene configurations.

The shadow silhouette map algorithm fixes aliasing in a manner independent of the relationship between the light and camera perspective transforms. Thus shadow silhouette maps handle aliasing in all situations, including projection aliasing, though small artifacts remain due to undersampling. Shadow silhouette maps are also fundamentally more complicated than perspective shadow maps: they rely on fragment programs, consume more memory, and require an extra rendering pass. Additional hardware may eventually reduce this overhead. Finally, the price to be paid for the higher quality of using shadow silhouette maps is that the algorithm is less general than perspective shadow maps. Since the silhouette map algorithm requires that we explicitly find silhouette edges (in order to rasterize them), it means that we must use polygonal models. This is not a major concern for many real-time applications, since modern graphics hardware is dedicated to polygonal rendering. In the future, however, other types of primitives such as higher-order surfaces may be supported.

Combination of Algorithms



Why not combine techniques?

Perspective shadow map:

- Optimizes depth sample distribution
- More samples closer to viewer

Shadow silhouette map:

- Optimizes depth sample information
- Exact silhouette edge locations

Fortunately, shadow silhouette maps and perspective shadow maps are complementary. It is possible to combine them to get the best of both worlds, at very little added cost. Keep in mind that the two techniques address aliasing in different ways. Perspective shadow maps optimize the distribution of the depth samples in the shadow map so that more samples are assigned to regions of the image closer to the viewer. In contrast, shadow silhouette maps optimizes the amount of information provided by each sample. In particular, depth samples are effectively deformed so that they lie along silhouette edges. This explicit edge information is used to reconstruct shadow edges accurately.

Summary

- Image-space algorithm
- Silhouette map: deformed depth map
- Piecewise-linear approximation
- Scalable (compared to shadow volumes)

Compared to (perspective) shadow maps:

- Removes aliasing in more cases
- Additional overhead and requirements

There are a few key ideas to remember about shadow silhouette maps. It is an extension of the regular shadow map algorithm, and it also works in image space. In addition to the shadow map, we add a silhouette map, which conceptually helps us to represent a deformed depth map in which depth samples are located where we need them: on the blockers' silhouettes, which give rise to hard shadow edges. Since we store one point per texel in the silhouette map, we obtain a piecewise-linear reconstruction of the true silhouette curve. This looks much better than the piecewise-constant reconstruct obtained using a standard shadow map. Compared to shadow volumes, which work in object-space, the silhouette map consumes less fillrate and bandwidth and scales better to complex scenes.

Compared to existing shadow-map-based approaches like perspective shadow maps, silhouette maps offer better quality, but the tradeoff is additional overhead and less generality. This is a classic tradeoff in shadow algorithms.





Soft Shadow Maps for Linear Lights

Wolfgang Heidrich

***(with Stefan Brabec,
Hans-Peter Seidel, MPI
Informatik)***

Wolfgang Heidrich



Overview

Motivation

Soft Shadow Maps

Hardware Implementation

Sampling the Light Source

Results

Conclusion

Wolfgang Heidrich



Motivation

"Real-Time" Shadow Algorithms

- Shadow volumes
- Shadow maps

Soft Shadows

- E.g. sampling the light source
N samples only give N-1 levels of penumbra

Here

Soft penumbra regions with very few samples !

Wolfgang Heidrich



Soft Shadow Maps

Outline

- Soft penumbra regions for linear light sources
- Based on “traditional” shadow map algorithm
- Suitable for hardware and software rendering
- Very small number of light source samples
- soft shadows at real-time / interactive frame rates

Wolfgang Heidrich



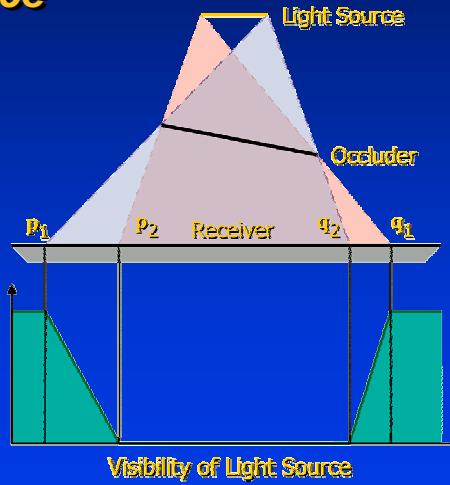
Soft Shadow Maps

Visibility of Light Source

- 100% to 0% for $[p_1, p_2]$
- 0% for $[p_2, q_2]$
- 0% to 100% for $[q_2, q_1]$

Idea

- Normal shadow maps for umbra and completely lit regions
- Linear interpolation of visibility for penumbra regions



Wolfgang Heidrich

Soft Shadow Maps

Linear Interpolation of Visibility

- Rational function:

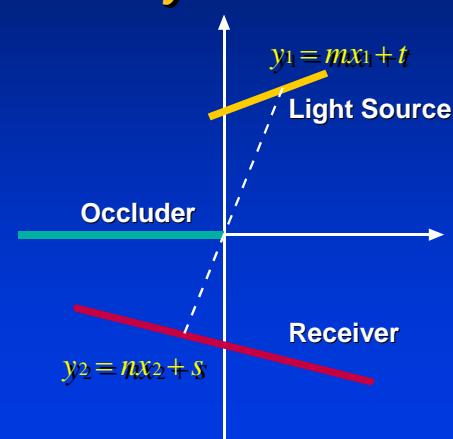
$$x_1 = \frac{x_2 t}{n x_2 - m x_2 + s}$$

- Approximation:

$$x_1 \approx -\frac{t}{s} x_2$$

valid because large penumbra regions when

$$n \approx m$$



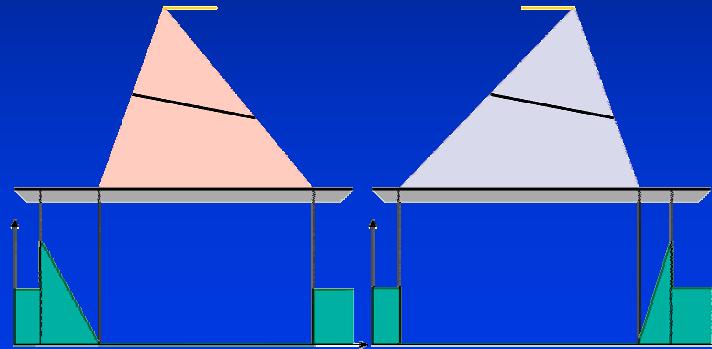
Wolfgang Heidrich



Soft Shadow Maps

Visibility Map

- Additional shadow map channel (percentage visibility)
- Two-channel shadow map for each sample point



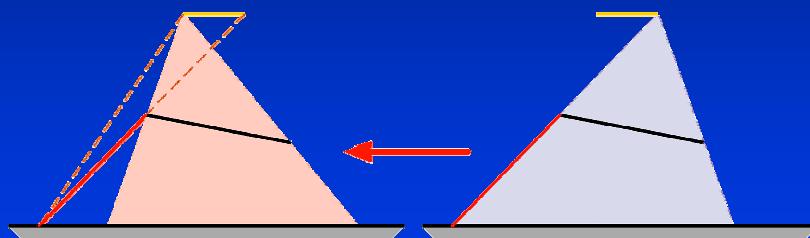
Wolfgang Heidrich



Soft Shadow Maps

Generating the Visibility Map

- Triangulate depth discontinuities (shadow map)
- Warp resulting skin polygons to other view



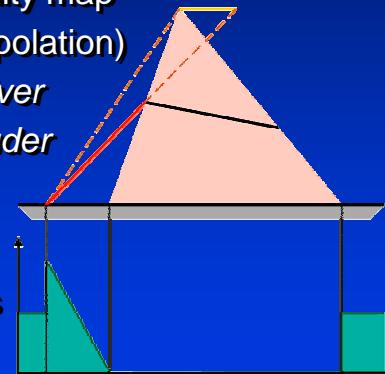
Wolfgang Heidrich



Soft Shadow Maps

Generating the Visibility Map

- Render skin polygons to visibility map
- Gouraud-Shading (linear interpolation)
 - “white” for vertices on receiver
 - “black” for vertices on occluder
- Completely lit regions
 - Default visibility 0.5
- Completely shadowed regions
 - First shadow map channel



Wolfgang Heidrich



Soft Shadow Maps

New shadow map algorithm

```
shade(p) {  
    if( depth1(p) > s1[p] )  
        l1 = 0;  
    else  
        l1 = v1[p] * illum(p,L1);  
    if( depth2(p) > s2[p] )  
        l2 = 0;  
    else  
        l2 = v2[p] * illum(p,L2);  
    return l1+l2;  
}
```

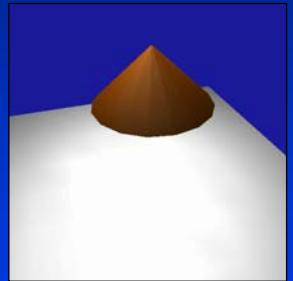
Wolfgang Heidrich



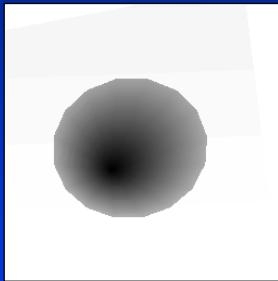
Hardware Implementation

Step 1: Generating Shadow Maps

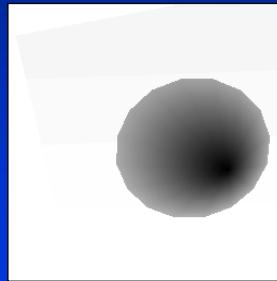
- OpenGL shadow maps [Brabec et al. '00]



camera view



left sample point



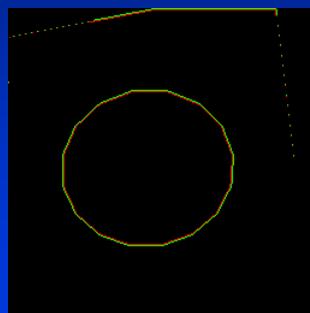
right sample point

Wolfgang Heidrich

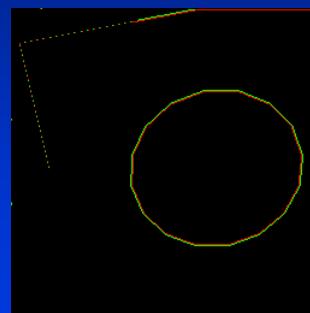
Hardware Implementation

Step 2: Edge Detection

- Laplacian-of-Gaussian (OpenGL Imaging Subset)



left sample point



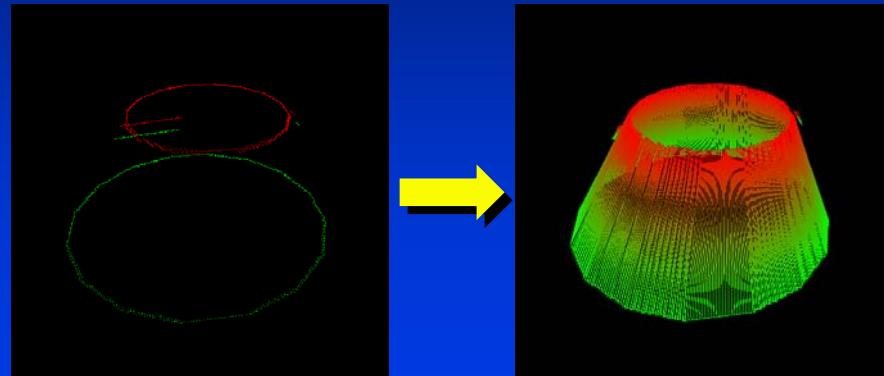
right sample point

Wolfgang Heidrich

Hardware Implementation

Step 3: Generate Visibility Map

- Triangulate depth values

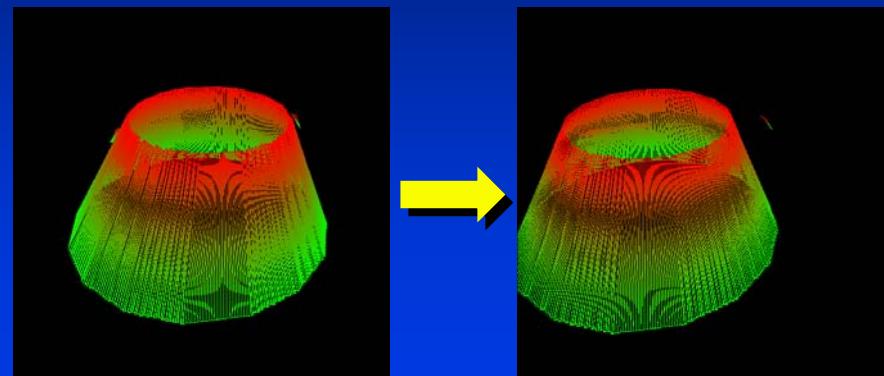


Wolfgang Heidrich

Hardware Implementation

Step 3: Generate Visibility Map

- Warp skin polygons



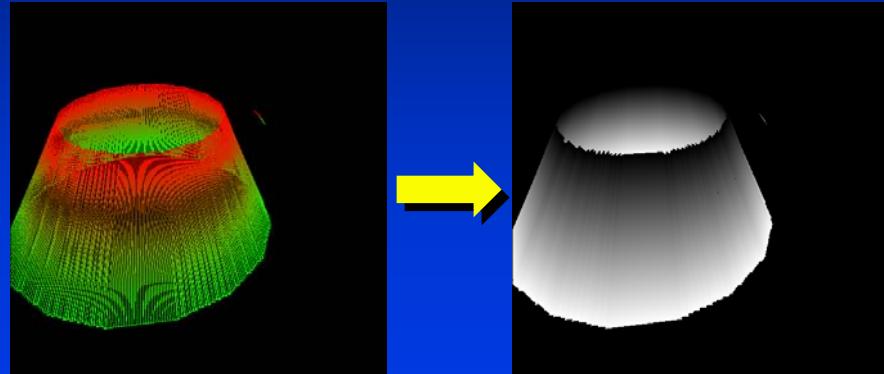
Wolfgang Heidrich



Hardware Implementation

Step 3: Generate Visibility Map

- Gouraud-Shading (linear interpolation)

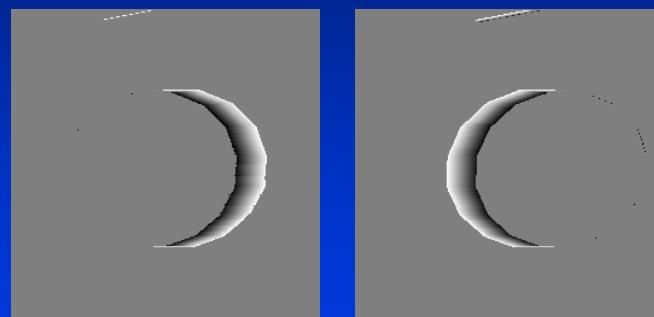


Wolfgang Heidrich



Hardware Implementation

Step 3: Generate Visibility Map



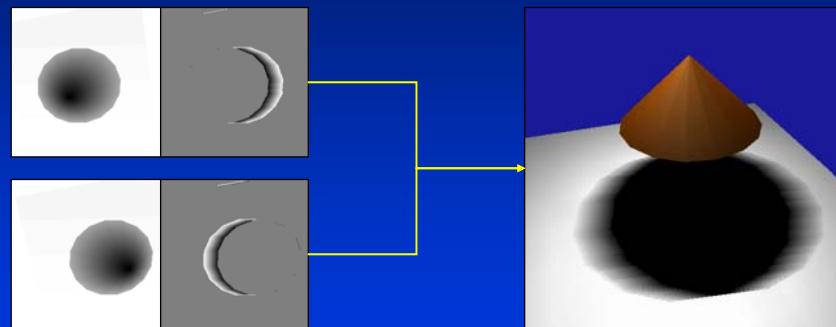
left sample point

right sample point

Wolfgang Heidrich

Hardware Implementation

Step 4: Render Scene



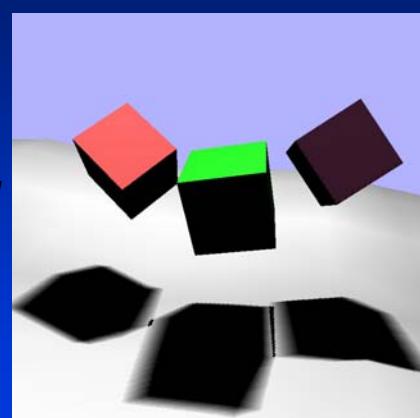
- shadow & visibility maps only need to be re-computed if light and/or scene changes
- minimal overhead for “static walk-throughs”

Wolfgang Heidrich

Sampling the Light Source

Problem:

- Undersampling artifacts:
regions where portions of the light source are visible, but none of the end points!



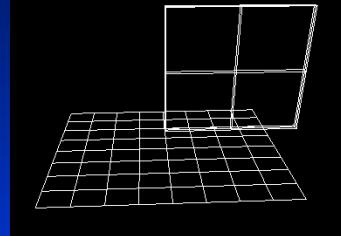
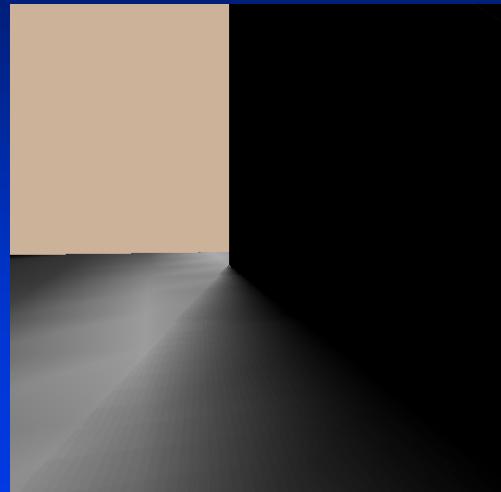
Solution:

- Increase sampling rate:
subdivide light source (smaller linear lights)

Wolfgang Heidrich



Results

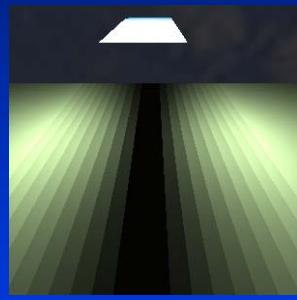


Wolfgang Heidrich

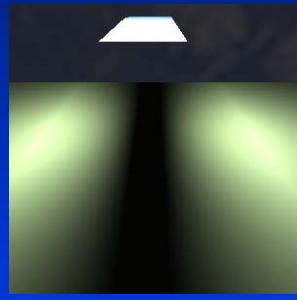


Results

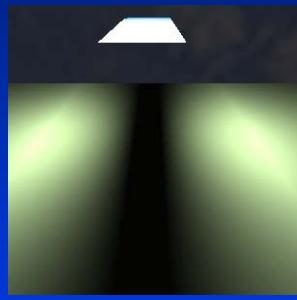
Comparison



*ray traced
10 samples*



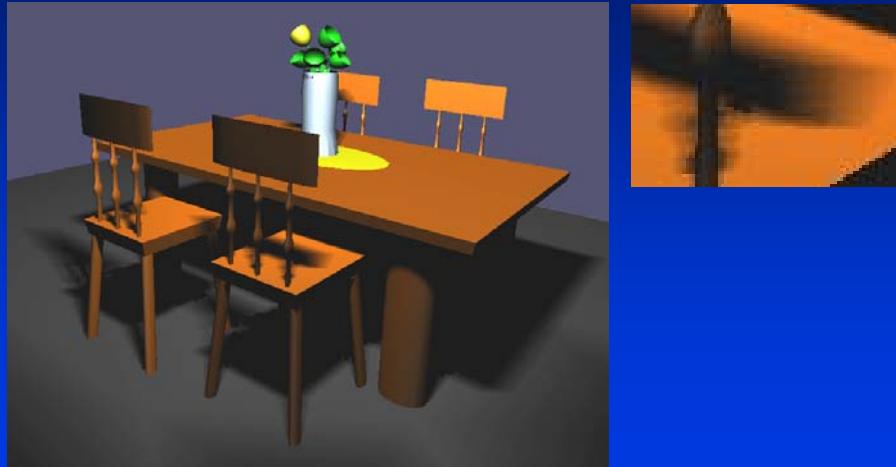
*ray traced
200 samples*



*our method
2 samples*

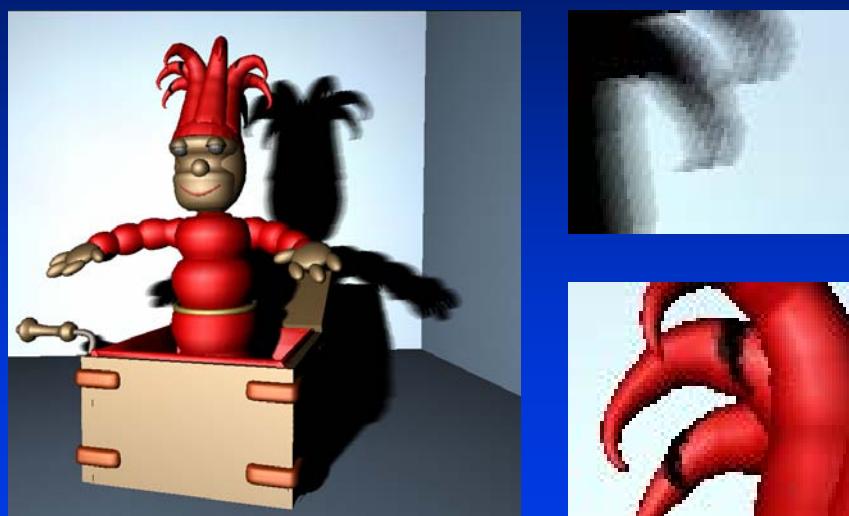
Wolfgang Heidrich

Results



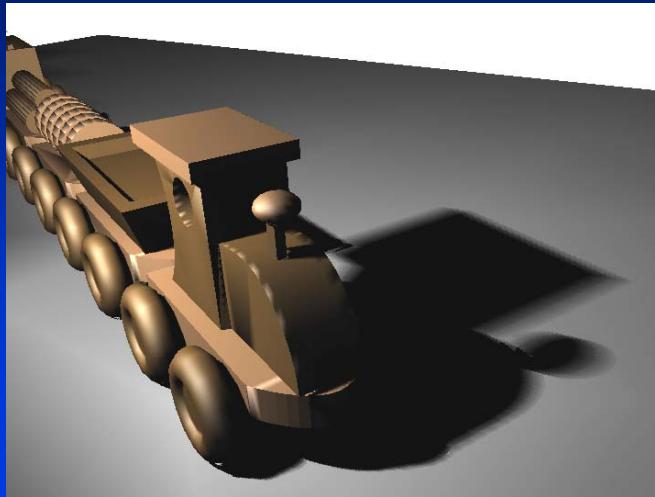
Wolfgang Heidrich

Results



Wolfgang Heidrich

Results



Wolfgang Heidrich

Conclusion

Soft Shadow Maps for Linear Lights

- New soft shadow algorithm based on shadow maps
- High-quality penumbra regions
- Very small number of light source samples
- Suitable for hardware rendering (interactive)

Future Work

- Best place to insert samples
- Extend to area light sources

Wolfgang Heidrich



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?

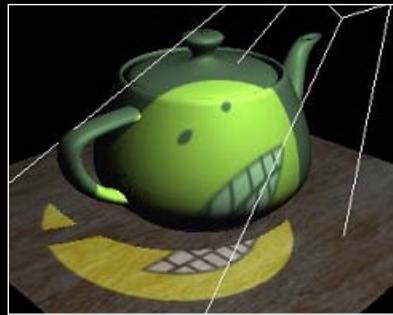
There are many (often conflicting) goals in the design of real-time soft shadow algorithms. On the one hand, we want interactive framerates, which usually means we need to have an algorithm simple enough to map directly to graphics hardware. On the other hand, we want high image quality and the ability 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

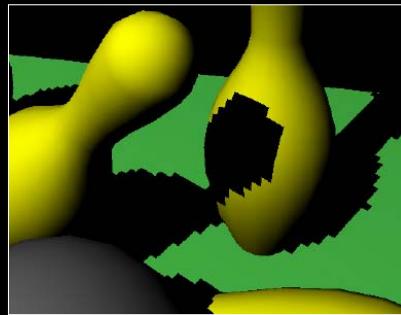


Image-space algorithm:

- 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

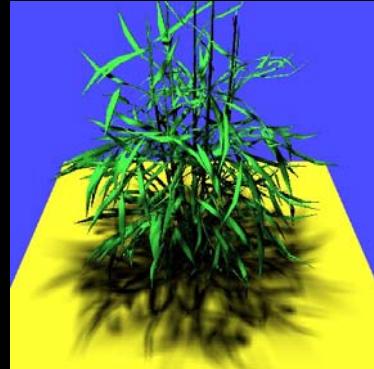


Techniques:

- 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](#)]

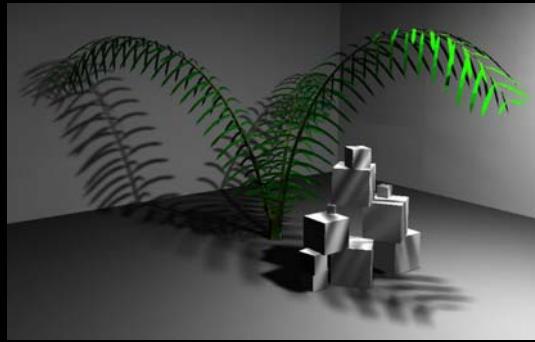
But: need dense sampling to minimize artifacts

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](#))

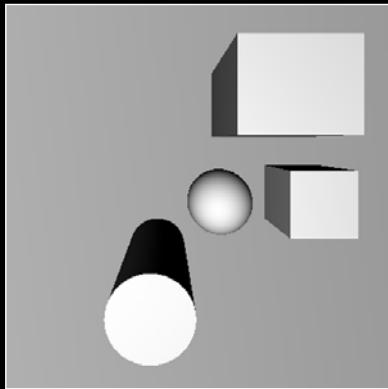
Another class of soft shadow techniques use various approximations to the true soft shadow. For instance, the soft shadows in the image shown here were generated by convolving blockers against an area light source. The method proposed in this session is also an approximate method.

Idea

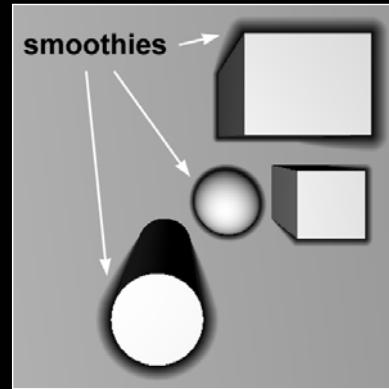


Extend basic shadow map approach

Extra primitives (**smoothies**) soften shadows



light's view (blockers only)



light's view (blockers + smoothies)

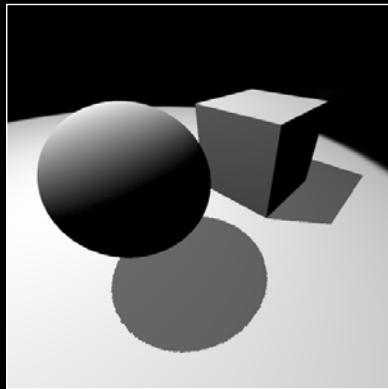
The idea of our soft shadow algorithm is just an extension of the shadow map approach. We use extra geometric primitives called “smoothies” to soften shadow edges. These primitives are attached like fins to the blockers’ silhouettes. The image on the left shows the blockers, seen from the point of view 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

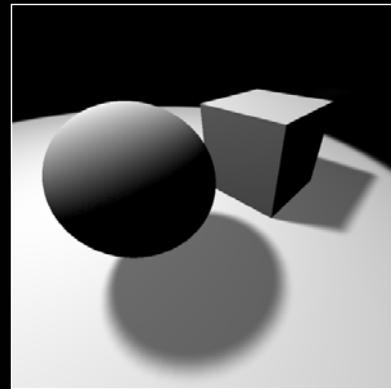


Shadows not geometrically correct

Shadows appear qualitatively like soft shadows



Hard shadows



Fake soft shadows

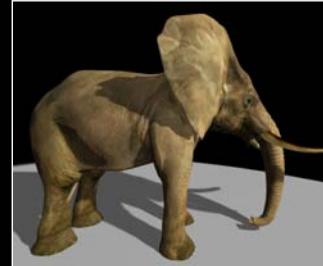
I should mention up front that the shadows generated using this method are not geometrically correct, i.e. we're not actually modeling an area light source. In fact, we don't even take the shape or orientation of the light source into account. However, the resulting shadows do have some of the important 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

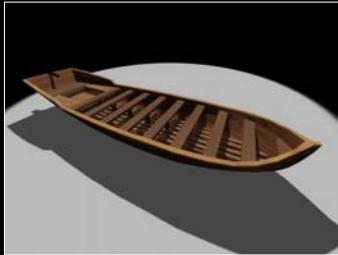


Rendering Fake Soft Shadows with Smoothies

- E. Chan and F. Durand [[EGSR 2003](#)]

Penumbra Maps

- C. Wyman and C. Hansen [[EGSR 2003](#)]

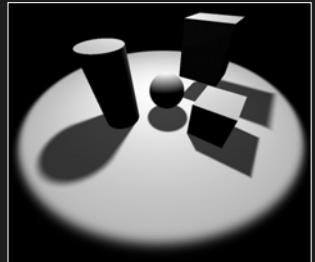
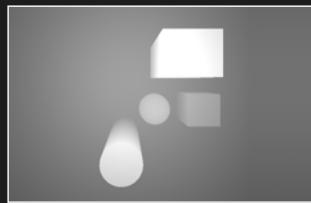


Before we dive into the algorithm details, let me mention two research papers that describe the algorithm in more detail. The idea was developed independently by Fredo Durand and myself at MIT, and by Chris Wyman and Chris Hansen at the University of Utah. The two papers were published simultaneously 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



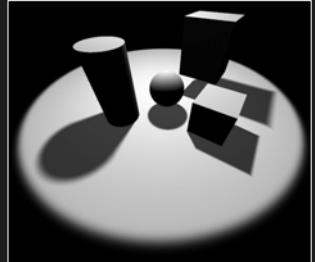
Focus on concepts
Implementation details later

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



Step 1



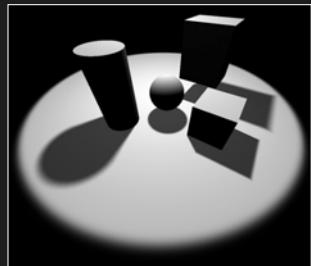
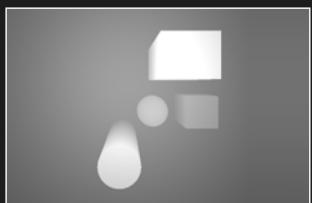
Create depth map

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



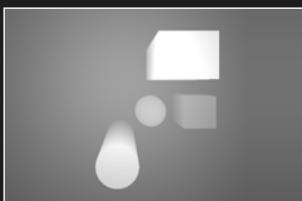
Create smoothie buffer

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.

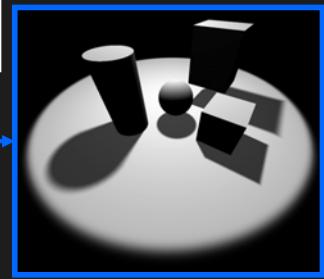
Algorithm Overview



Step 3



Render scene + shadows

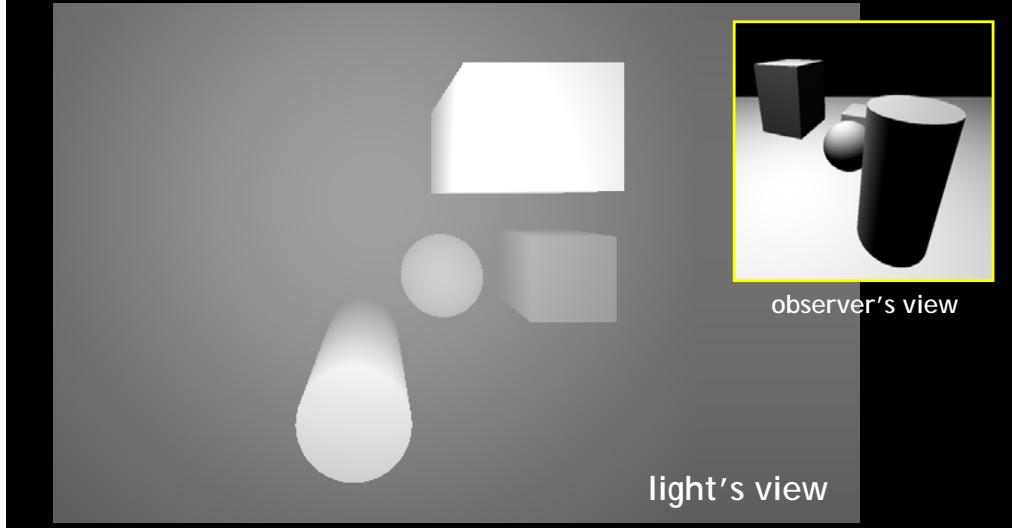


In the final step, we draw the scene from the observer's viewpoint and compute the shadows. We refer to both the shadow map and the smoothie buffer to compute shadows with soft edges.

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.

Create Shadow Map

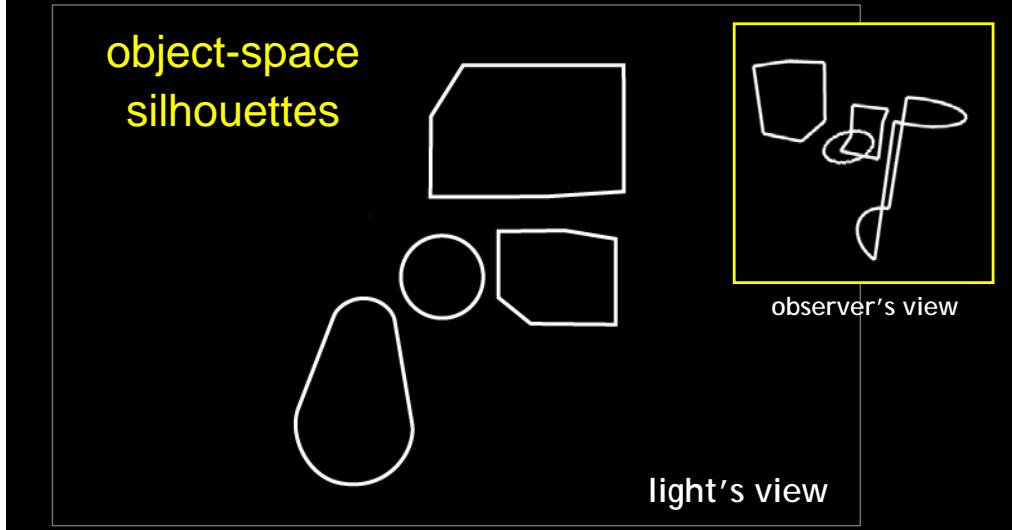
Render blockers into depth map



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

Find Silhouette Edges

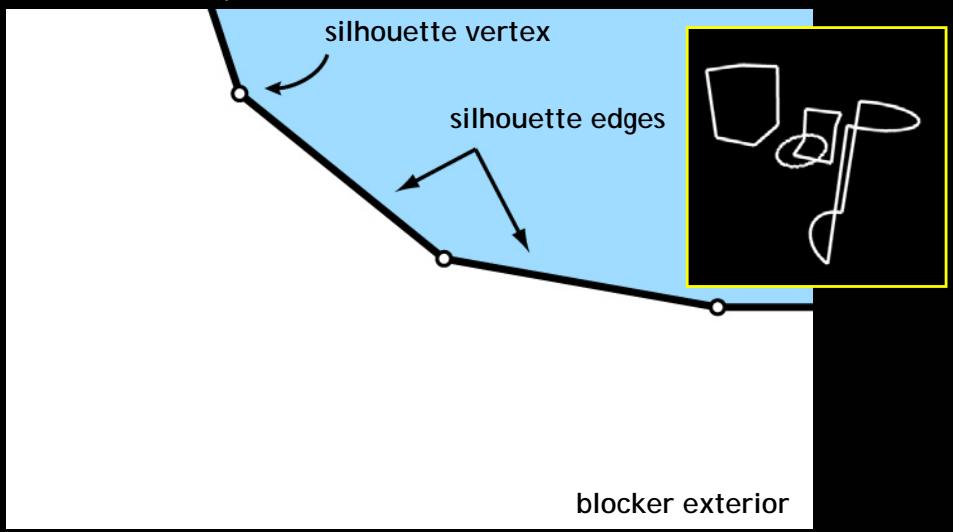
Find blockers' silhouette edges in object space



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.

Construct Smoothies

Blocker only:

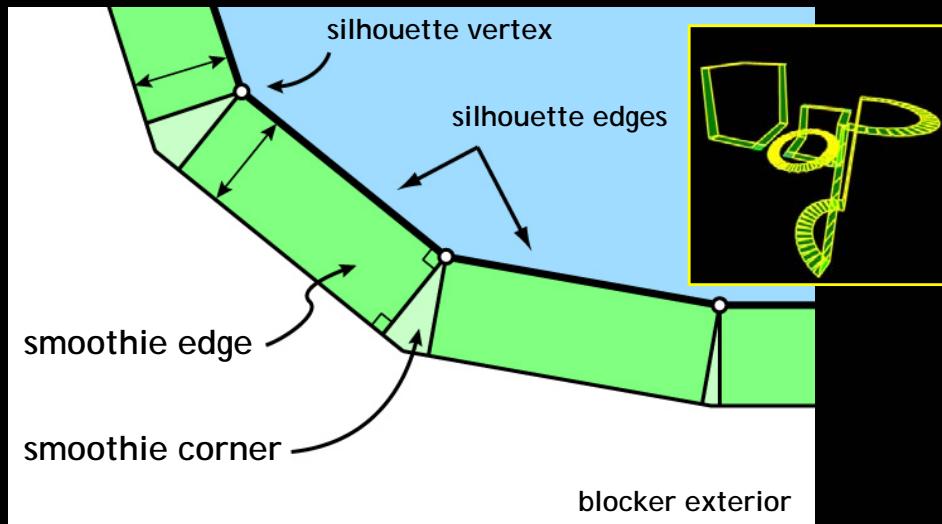


Now that we have the silhouette edges, we can go ahead and construct the smoothies. The diagram here shows a blocker (light blue) and its silhouette edges and vertices.

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.

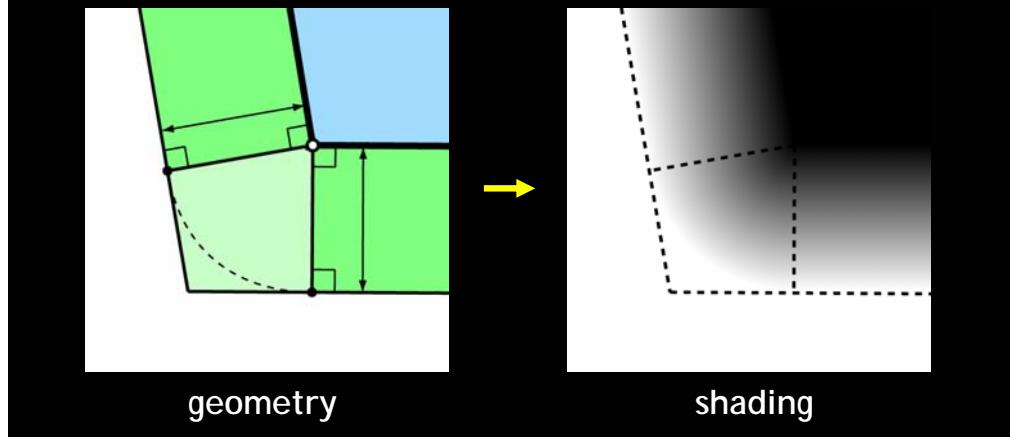
How thick do we make the smoothie geometry? Intuitively, the wider we make a smoothie, the softer the shadow becomes. Thus, the size of a smoothie should depend on the size of the light source being simulated.

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



Smoothie edges are fixed-width rectangles in screen space
Smoothie corners connect adjacent smoothie edges

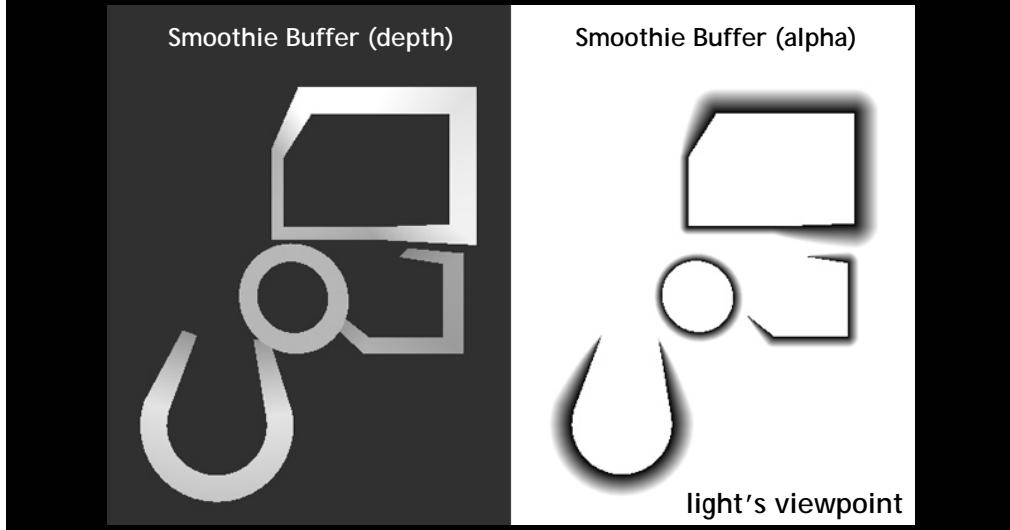


In case it's unclear where we're going with the smoothie construction, perhaps these diagrams will help. The idea is that we'll compute alpha values for the smoothies (shown on the right) in such a way so that when we project them onto the scene from the light's viewpoint (via projective texture mapping),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

Render Smoothies



Store depth and alpha values into smoothie buffer

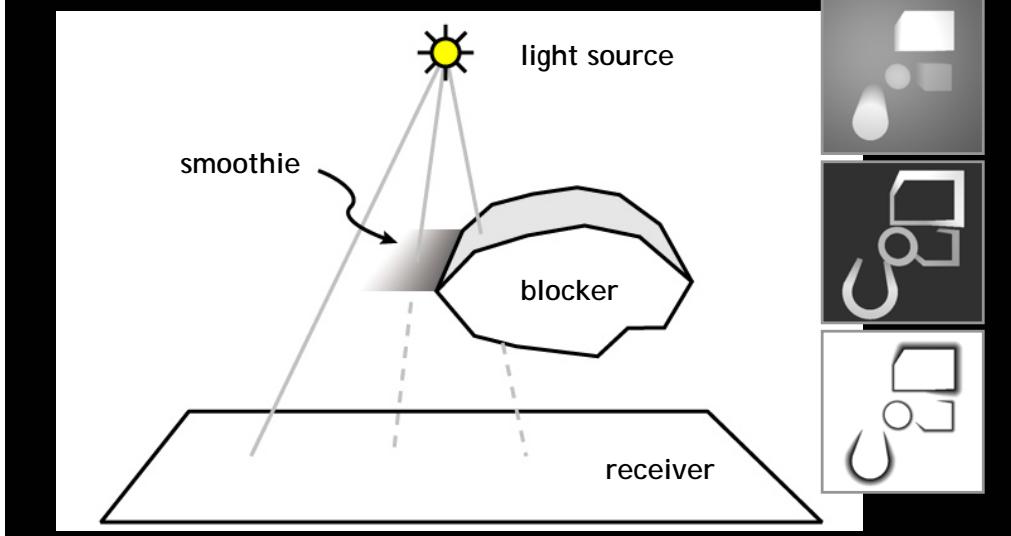


Now that we've constructed the smoothies, we draw them from the light's viewpoint into the smoothie buffer. We compute both depth (shown on left) and alpha values (shown on right) for each smoothie pixel. Rendering depth is straightforward (the same as when rendering a shadow map). We'll come back and 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!

Compute Shadows



Compute intensity using depth comparisons

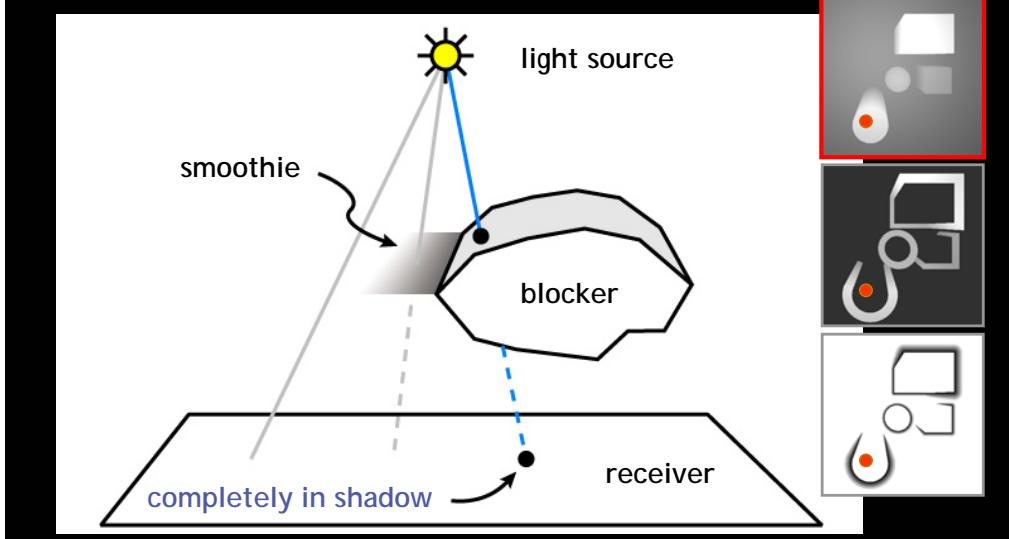


Finally, we render the scene from the observer's viewpoint. For each sample, we check for three cases.

Compute Shadows (1 of 3)



Image sample behind blocker (intensity = 0)



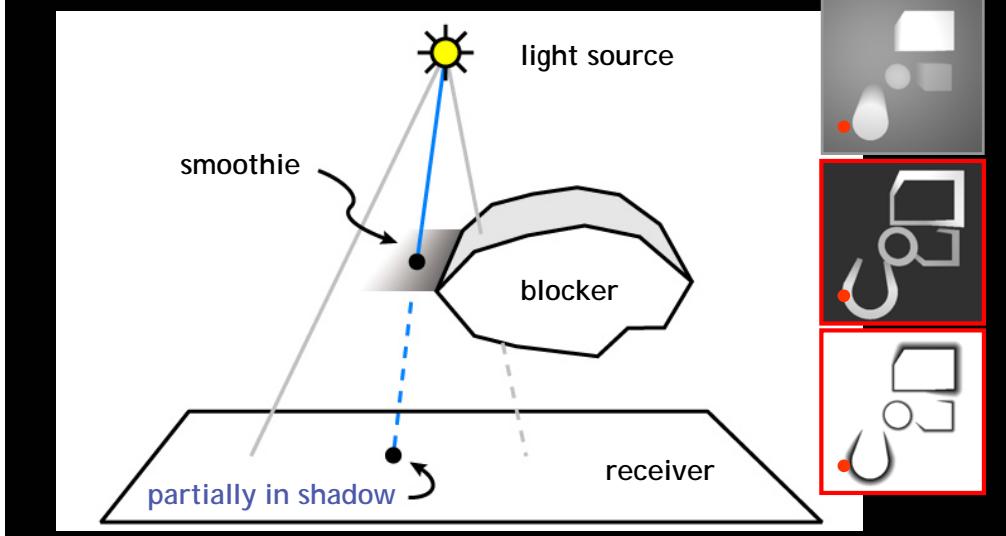
We project the sample into light space so that we can perform lookups into the shadow map and smoothie buffer. If the sample is behind the shadow map, i.e. the depth of the sample is greater than the depth value stored in the shadow map, then we say the sample is completely in shadow and assign it zero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

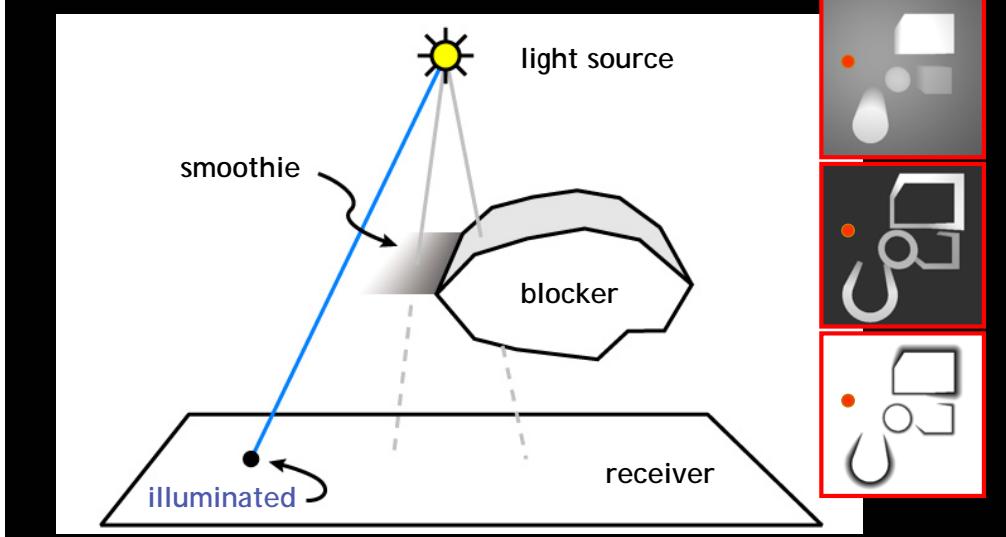


If the sample is not behind a blocker, but it is behind a smoothie, then we shade the sample using the smoothie's alpha value. We use the depth value stored in the smoothie buffer to make the smoothie depth comparison, and we retrieve the alpha value from the alpha part of the smoothie buffer.

Compute Shadows (3 of 3)



Image sample illuminated (intensity = 1)



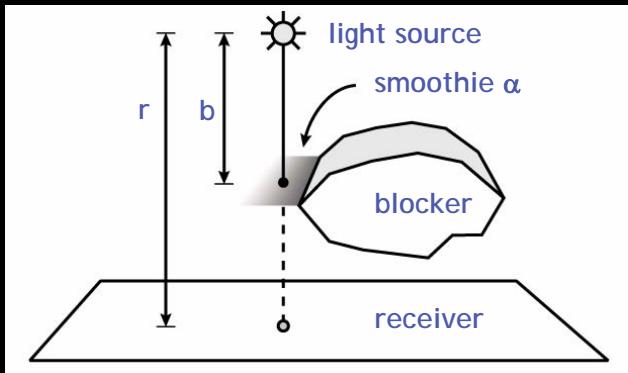
Finally, if the sample is neither behind a blocker nor behind a smoothie, then it is considered fully illuminated.

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

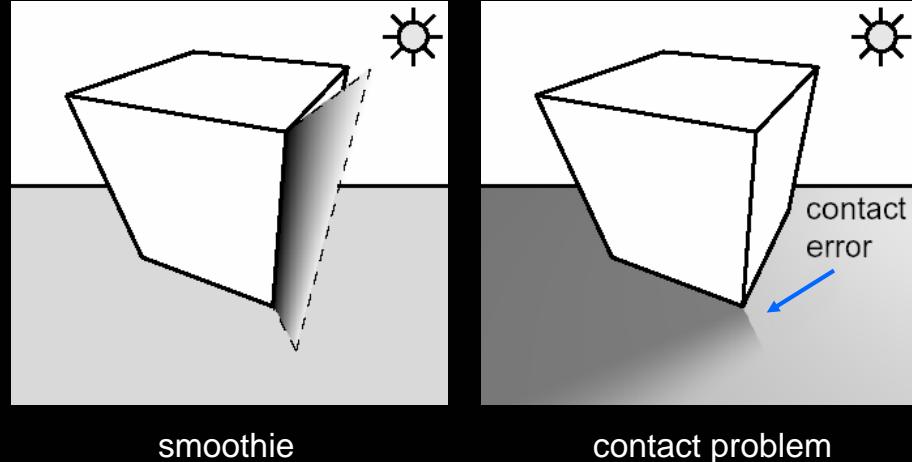


Now that we've seen the sequence of steps involved in the smoothie algorithm, let's get back to how we compute the alpha values when rendering the smoothies. Consider the diagram shown here, which shows the high-level geometric relationship between a light source, blocker, and receiver. One of the properties 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



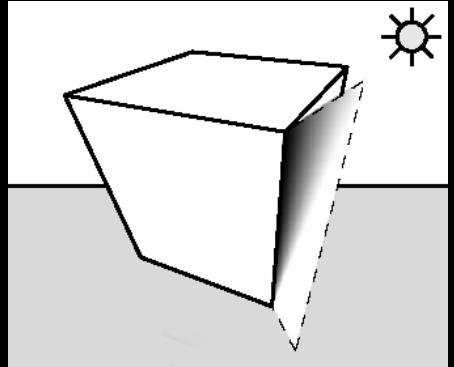
Linearly interpolated alpha → undesired results!



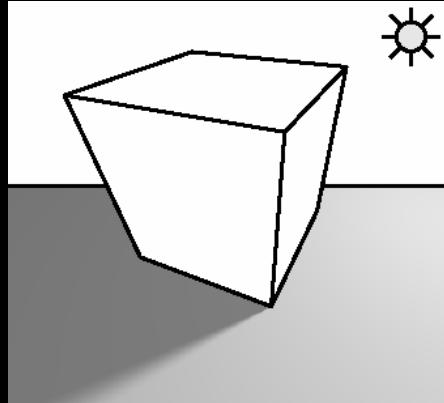
Let's see what happens if we don't take into account this ratio b/r . Intuitively, it seems that we should have alpha start from 0 at the blocker (0 meaning complete occlusion by the blocker) and fade to 1 at the edge (1 meaning 100% visible). Notice what happens, though, if we just linearly interpolate 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 take into account the ratio of distances.

With Alpha Remapping

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



smoothie



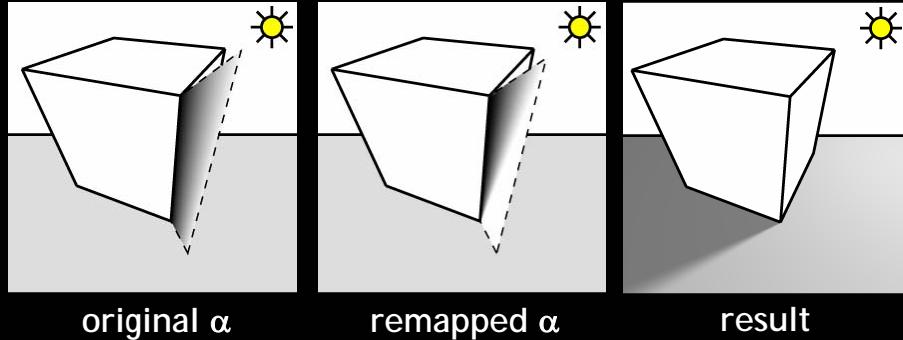
fixed contact problem

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.

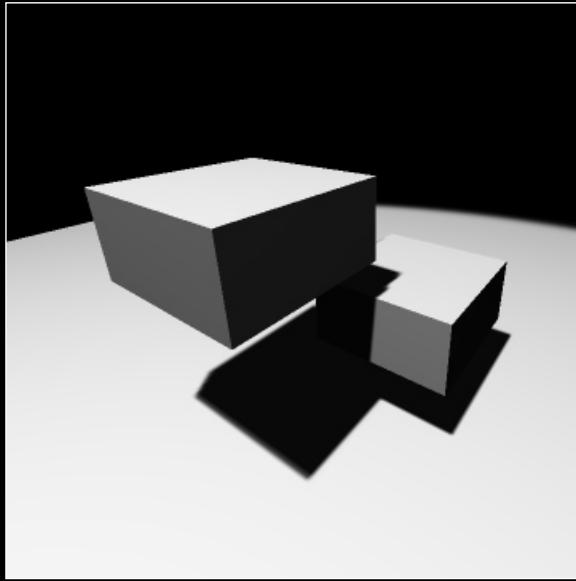
Computing Alpha Values

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

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

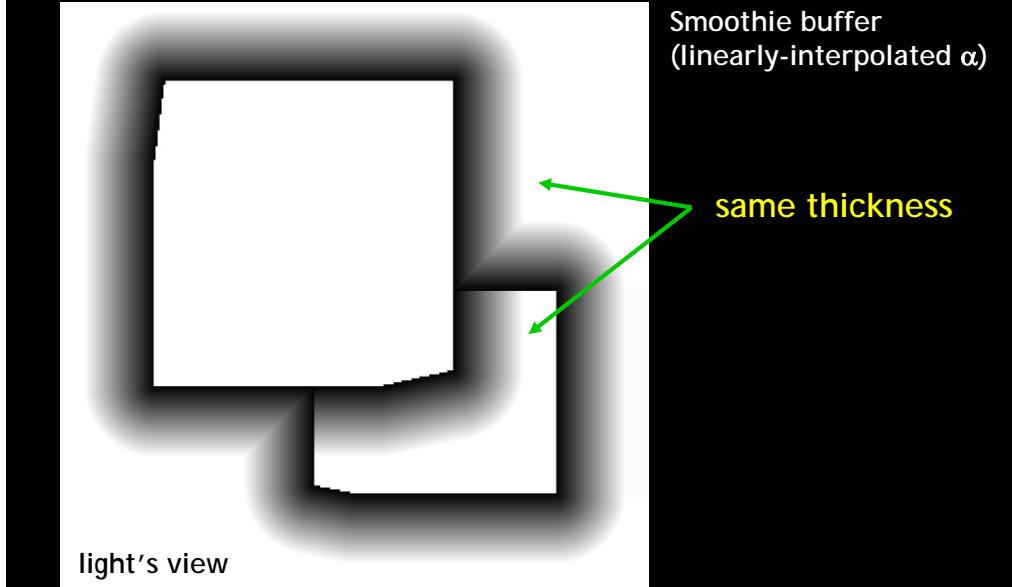


Multiple Objects



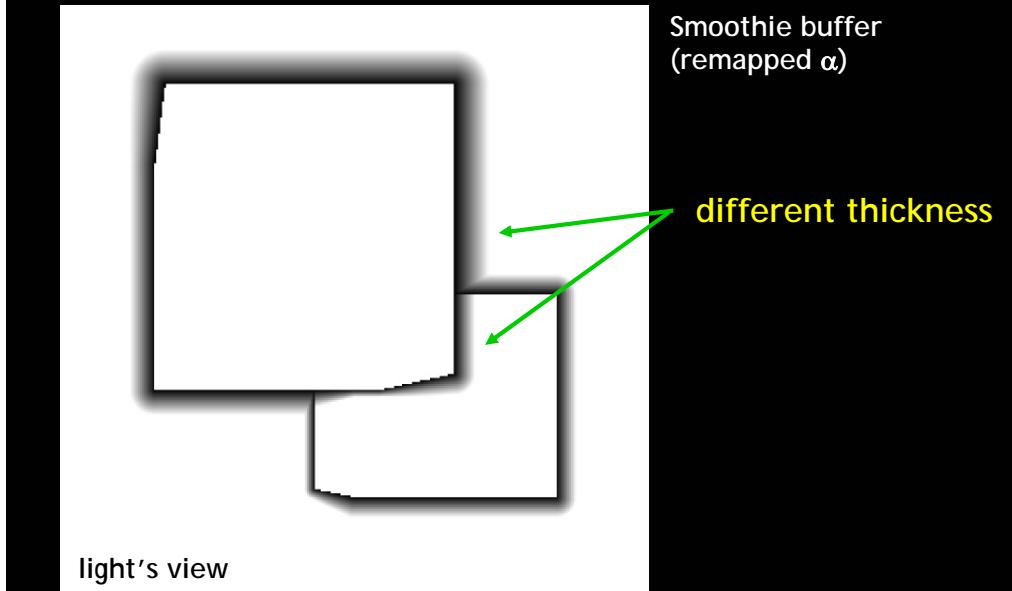
So far we have only considered simple examples, ones in which there is a single blocker and receiver. In more complex scenes, however, there are multiple blockers and receivers. We'll use this scene of two boxes and a floor to study how these blockers and receivers interact. For instance, in the image 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



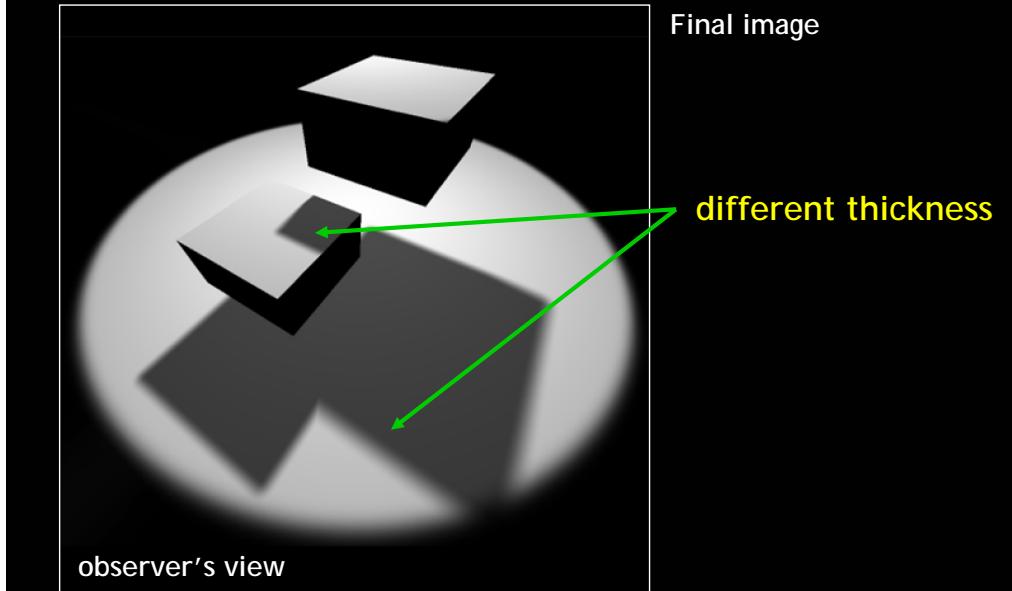
Let's first consider the case of multiple receivers. In the image of the smoothie buffer shown here, we have neglected to remap the alpha values as suggested earlier. Instead, we have simply used the linearly-interpolated alpha. Notice that the top box, in principle, should cast a harder shadow on the 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.

Multiple Receivers



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

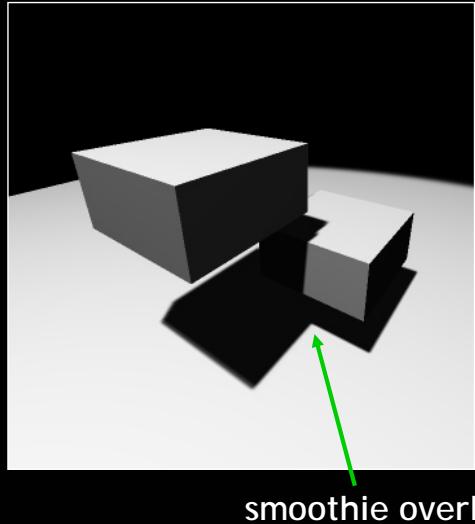
Multiple Receivers



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?



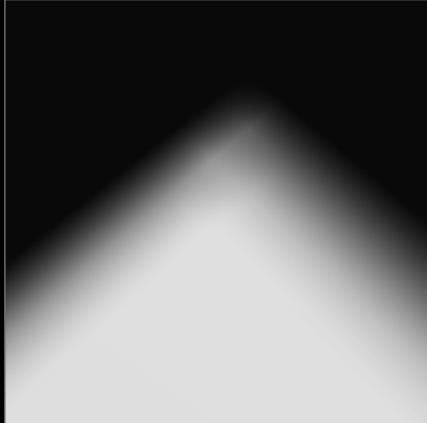
Now let's turn to the case of multiple blockers and understand how to handle overlapping shadows. The case of overlapping shadows means that multiple smoothies overlap in the screen space of the light source. Computing the smoothies' depth values is the same: just store the nearest depth value into the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

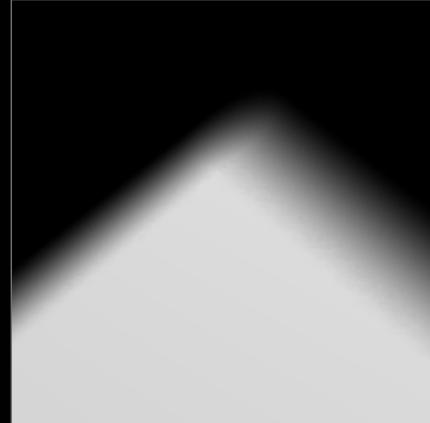
Multiple Blockers



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

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

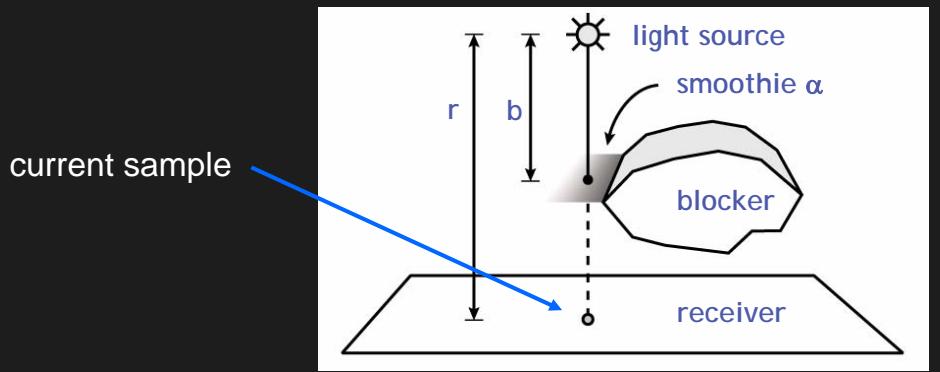
Next, we draw the smoothies twice, once to store the depth values, and once to store the alpha values (which we'll actually end up storing in the color buffer). The reason we cannot compute and store both quantities in the same pass is because we use minimum blending when rendering the smoothies' alpha,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



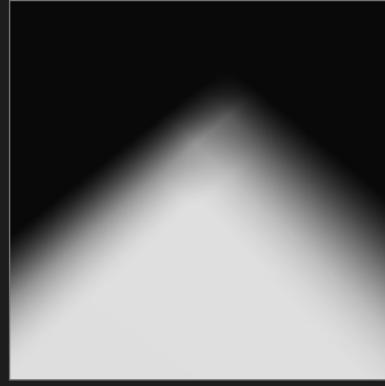
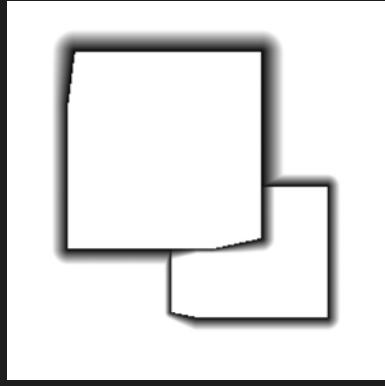
We use a fragment program to perform the alpha computation. The distance b from the light to the smoothie is easy to compute. Since we are drawing the smoothies from the light's viewpoint, if a vertex has been transformed (i.e. in a vertex program) to eye space, then the z component is exactly the distance 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.

Minimum Blending

Implementation in OpenGL:

- Supported natively in hardware
- use `glBlendEquationEXT(GL_MIN_EXT)`

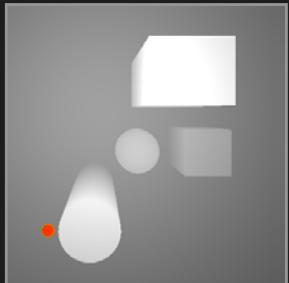


It is simple to implement minimum blending in OpenGL because it is supported natively in hardware. Just use the shown GL command.

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.



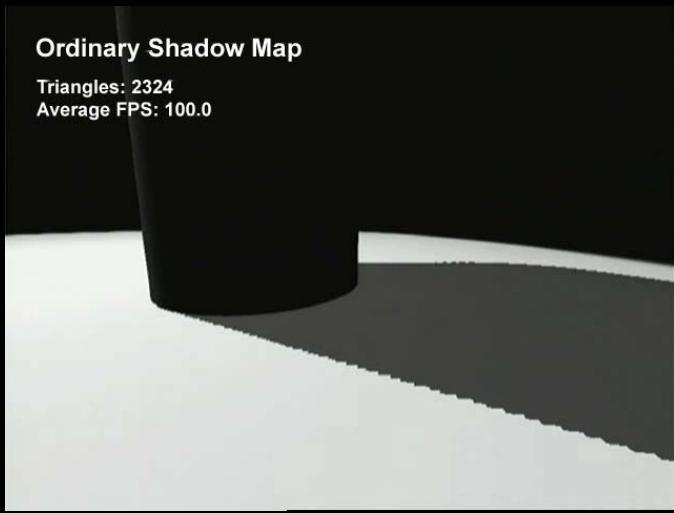
Examples

Video

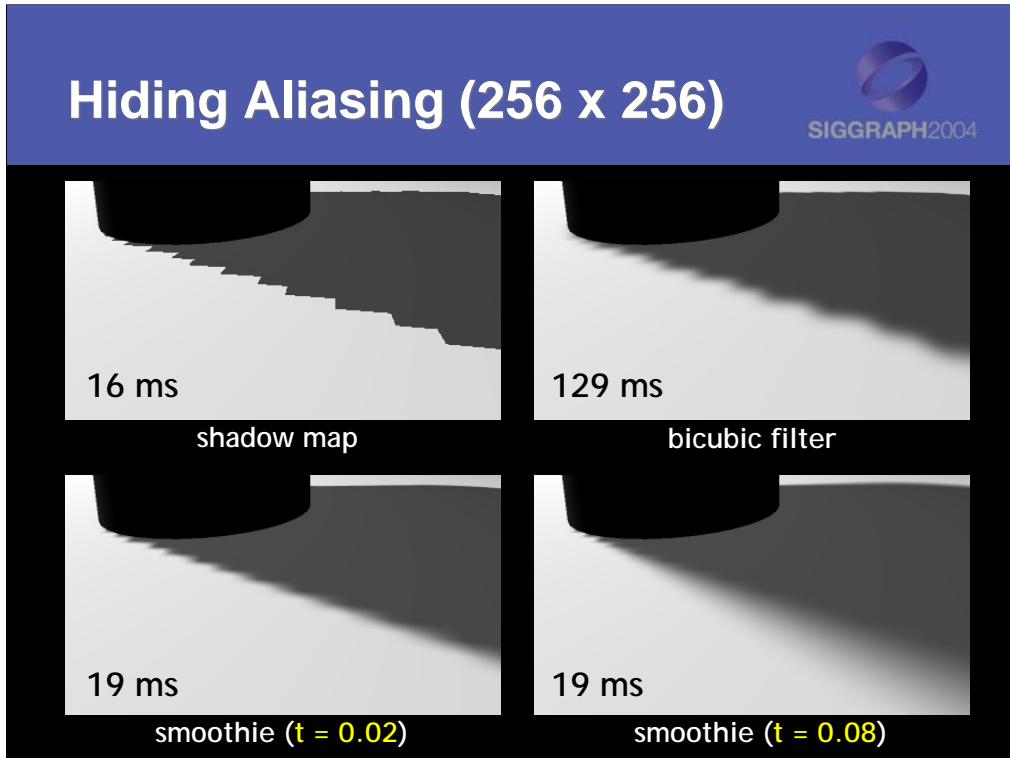


Ordinary Shadow Map

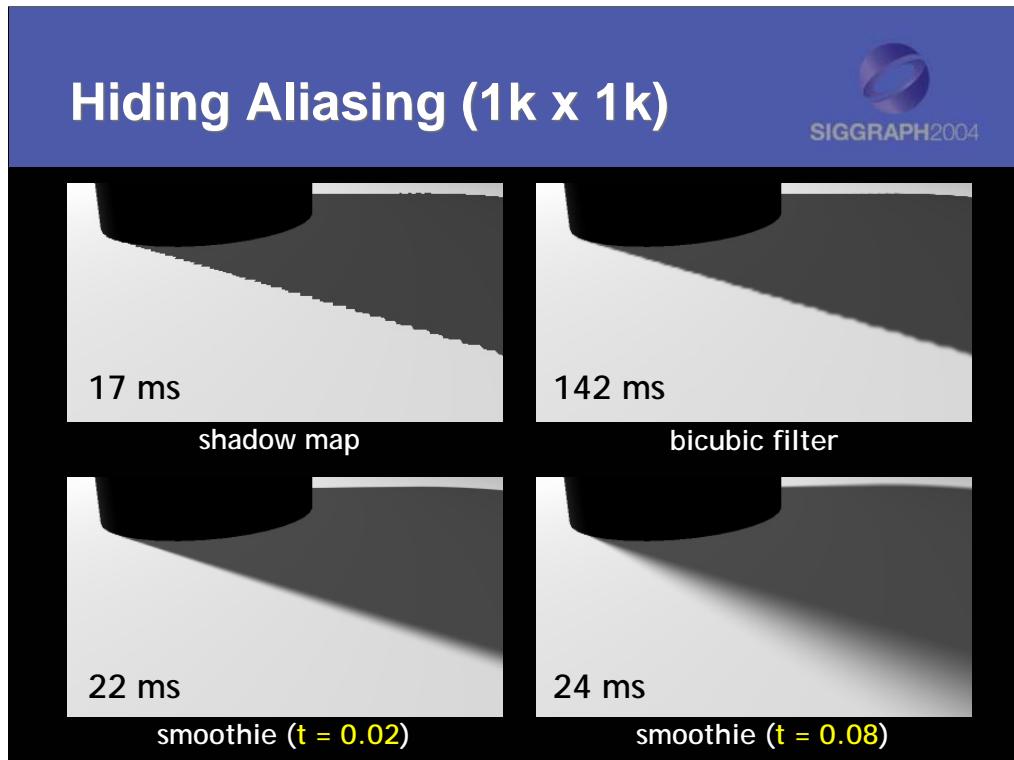
Triangles: 2324
Average FPS: 100.0



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/>



I mentioned earlier that the smoothie algorithm is useful for hiding aliasing artifacts. Let's consider an extreme case: using low-resolution (256×256) buffers. We're looking at 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 row of 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.

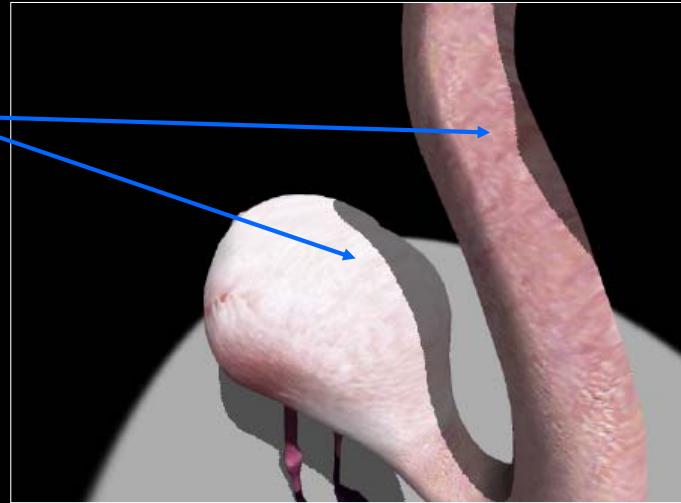


This is the same scene, except using 1024 x 1024 buffers. Whereas aliasing is still noticeable in the top of images, it is completely masked in the images generated using the smoothie algorithm. Furthermore, the bottom-row image looks convincingly like a soft shadow, with the shadow getting softer farther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

Antialiasing Example #1



hard shadows
(aliased)



shadow map

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.

Antialiasing Example #1



soft shadows
(antialiased)

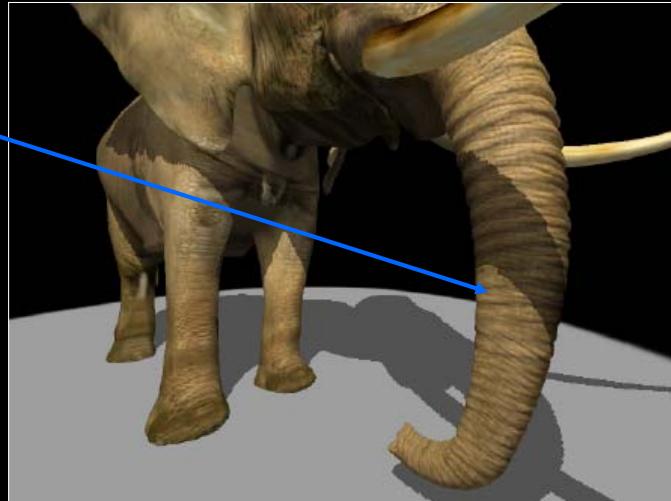
smoothies

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

Antialiasing Example #2



hard shadows
(aliased)



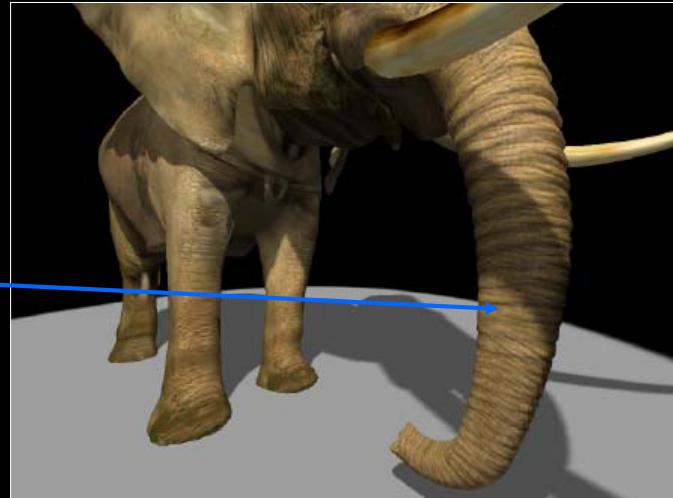
shadow map

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



soft shadows
(antialiased)



smoothies

The shadows using the smoothie algorithm are soft and antialiased.

Limitations



increasing size
of light source

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



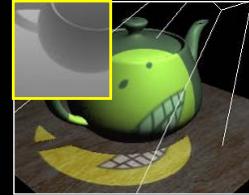
original md2shader demo courtesy of Mark Kilgard

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

Shadow maps:

- Assumes directional light or spotlight
- Discrete buffer samples



Shadow volumes:

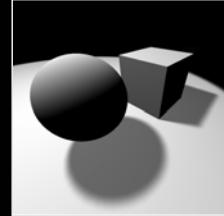
- Assumes blockers are closed triangle meshes
- Silhouettes identified in object space

As with the shadow volume method, the smoothie algorithm requires finding object-space silhouettes. This implies that our blockers 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

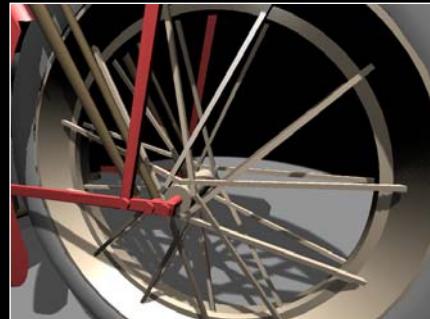
One might wonder about the expense of rendering the smoothies themselves. After all, it sounds similar to rasterizing shadow volumes, which is known to be a costly operation. The main difference here is that shadow volume polygons are drawn from the observer's viewpoint. They occupy substantial screen 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



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



Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering

Cass Everitt and Mark J. Kilgard

March 12, 2002

NVIDIA Corporation, Copyright 2002
Austin, Texas

ABSTRACT

Twenty-five years ago, Crow published the shadow volume approach for determining shadowed regions in a scene. A decade ago, Heidmann described a hardware-accelerated stencil buffer-based shadow volume algorithm.

However, hardware-accelerated stenciled shadow volume techniques have not been widely adopted by 3D games and applications due in large part to the lack of robustness of described techniques. This situation persists despite widely available hardware support. Specifically what has been lacking is a technique that robustly handles various "hard" situations created by near or far plane clipping of shadow volumes.

We describe a robust, artifact-free technique for hardware-accelerated rendering of stenciled shadow volumes. Assuming existing hardware, we resolve the issues otherwise caused by shadow volume near and far plane clipping through a combination of (1) placing the conventional far clip plane "at infinity", (2) rasterization with infinite shadow volume polygons via homogeneous coordinates, and (3) adopting a *zfail* stencil-testing scheme. *Depth clamping*, a new rasterization feature provided by NVIDIA's GeForce3 & GeForce4 Ti GPUs, preserves existing depth precision by not requiring the far plane to be placed at infinity. We also propose *two-sided stencil testing* to improve the efficiency of rendering stenciled shadow volumes.

Keywords

Shadow volumes, stencil testing, hardware rendering.

1. INTRODUCTION

Crow's shadow volume approach [10] to shadow determination is twenty-five years old. A shadow volume defines a region of space that is in the shadow of a particular occluder given a particular ideal light source. The shadow test determines if a given point being tested is inside the shadow volume of any occluder. Hardware stencil testing provides fast hardware acceleration for shadow determination using shadow volumes. Despite the relative age of the shadow volume approach and the widespread availability of stencil-capable graphics hardware, use of shadow volumes in 3D games and applications is rare.

We believe this situation is due to the lack of a practical and robust algorithm for rendering stenciled shadow volumes. We propose here an algorithm to address this gap. Our algorithm is practical because it requires only features available in OpenGL 1.0. The algorithm is robust because shadow volume scenarios that vexed previous algorithms, such as a light within an open container, are handled automatically and correctly.

We focus on robustly solving the problem of hardware-accelerated stenciled shadow volume rendering for a number of reasons, many noted by other authors [9][10][11][19]:

- Shadow volumes provide omni-directional shadows.



- Shadow volumes automatically handle self-shadowing of objects if implemented correctly.
- Shadow volumes perform shadow determination in window space, resolving shadow boundaries with pixel accuracy (or sub-pixel accuracy when multisampling is available).
- Lastly, the fundamental stencil testing functionality required for hardware-accelerated stenciled shadow volumes is now ubiquitous due to the functionality's standardization by OpenGL 1.0 (1991) and DirectX 6 (1998) respectively. It is near impossible to purchase a new PC in 2002 without stencil testing hardware.

Stenciled shadow volumes have their limitations too. Shadow volumes model ideal light sources so the resulting shadow boundaries lack soft edges. Shadow volume techniques require polygonal models. Unless specially handled, such polygonal models must be closed (2-manifold) and be free of non-planar polygons. Silhouette computations for dynamic scenes can prove expensive. Stenciled shadow volume algorithms are inherently multi-pass. Rendering shadow volumes can consume tremendous amounts of pixel fill rate.

2. PREVIOUS WORK

2.1 Pre-Stencil Testing Work

Crow [10] first published the shadow volume approach in 1977. Crow recognizes that the front- or back-facing orientations of consistently rendered shadow volume polygons with respect to the viewer indicate enters into and exits out of shadowed regions. Crow also recognizes that some care must be taken to determine if the viewer's eye point is within a shadow volume.

Crow's formulation fundamentally involves walking a pixel's view ray originating at the eye point and counting the number of shadow volume enters and exits encountered prior to the first visible rasterized fragment.

Brotman and Badler [8] in 1984 adapted Crow's shadow volume approach to a software-based, depth-buffered, tiled renderer with deferred shading and support for soft shadows through numerous light sources all casting shadow volumes.

Pixel-Planes [13] in 1985 provides hardware support for shadow volume evaluation. In contrast to Crow's original ray walking

approach, the Pixel-Planes algorithm relies on determining if a pixel is within an infinite polyhedron defined by a single occluder triangle plane and its three shadow volume planes. This determination is made for every pixel and for every occluder polygon in the scene. Each “point within a volume” test is computed by evaluating the corresponding set of plane equations.

Pixel-Planes is a unique architecture because the area of a rasterized triangle in pixels does not affect the triangle’s rasterization time. Otherwise, the algorithm’s evaluation of *every* per-triangle shadow volume plane equation at *every* pixel would be terribly inefficient.

Bergeron [3] in 1986 generalizes Crow’s original shadow volume approach. Bergeron explains how to handle open models and models containing non-planar polygons properly. Bergeron explicitly notes the need to close shadow volumes so that a correct initial count of how many shadow volumes the eye is within can be computed.

Fournier and Fussell [12] in 1988 discuss shadow volumes in the context of frame buffer computations. In their computational model, each pixel in a frame buffer maintains a depth value and shadow depth count. Fournier and Fussell’s frame buffer computation model lays the theoretical foundations for subsequent hardware stencil buffer-based algorithms.

2.2 Stencil Testing-Based Work

2.2.1 The Original Approach

Heidmann [14] in 1991 describes an algorithm for using the then-new stencil buffer support of SGI’s VGX graphics hardware [1]. Heidmann recognizes the problem of stencil buffer overflows and demonstrates combining contributions from multiple light sources with the accumulation buffer to simulate soft shadows.

Heidmann’s approach is a multi-pass rendering algorithm. First, the color, depth, and stencil buffers are cleared. Second, the scene is drawn with only ambient and emissive lighting contributions and using depth testing for visibility determination. Now the color and depth buffers contain the color and depth values for the closest fragment rendered at each pixel. Then shadow volume polygons are rendered into the scene but just updating stencil.

Front-facing polygons update the frame buffer with the following OpenGL per-fragment operations (for brevity, we drop the gl and GL prefixes for OpenGL commands and tokens):

```
Enable(CULL_FACE);           // Face culling enabled
CullFace(BACK);             // to eliminate back faces
ColorMask(0,0,0,0);          // Disable color buffer writes
DepthMask(0);                // Disable depth buffer writes
StencilMask(~0);            // Enable stencil writes
Enable(DEPTH_TEST);          // Depth test enabled
DepthFunc(LEQUAL);           // less than or equal
Enable(STENCIL_TEST);         // Stencil test enabled
StencilFunc(ALWAYS,0,~0)     // always pass
StencilOp(KEEP,KEEP,INCR);    // increment on zpass
```

Similarly, back-facing polygons update the frame buffer with the following OpenGL state modifications:

```
CullFace(FRONT);             // Now eliminate front faces
StencilOp(KEEP,KEEP,DECR);    // Now decrement on zpass
```

Heidmann’s described algorithm computes the front- or back-facing orientation of shadow volume polygons on the CPU. We

note (as have other authors [5][15]) that the shadow volume polygons can be rendered in two passes: first, culling back-facing polygons to increment pixels rasterized by front-facing polygons; second, culling front-facing polygons to decrement pixels rasterized by back-facing polygons. This leverages the graphics hardware’s ability to make the face culling determination automatically and minimizes hardware state changes at the cost of rendering the shadow volume polygons twice. Utilizing the hardware’s face culling also avoids inconsistencies if the CPU and graphics hardware determine a polygon’s orientation differently in razor’s edge cases.

After the shadow volume polygons are rendered into the scene, a pixel’s stencil value is equal to zero if the light illuminates the pixel and greater than zero if the pixel is shadowed. The scene can then be re-rendered with the appropriate light configured and enabled, with stencil testing enabled to update only pixels with a zero stencil value (meaning the pixel is not shadowed), and “depth equal” depth testing (to update only visible fragments). The light’s contribution can be accumulated with either the accumulation buffer or additive blending.

This can be repeated for multiple light sources, clearing the stencil buffer between rendering the shadow volumes and summing the contribution of each light.

2.2.2 Near and Far Plane Clipping and Capping

Heidmann fails to mention in his article a problem that seriously undermines the robustness of his approach. With arbitrary scenes, the near and/or far clip planes may (and, in fact, often will) clip the infinite shadow volumes. Each shadow volume is, by construction, a half-space (dividing the entirety of space into the region shadowed by a given occluder and everything else). However, near and far plane clipping can “slice open” an otherwise well-defined half space. Disturbing the shadow volume in this way leads to incorrect shadow depth counting that, in turn, results in glaringly incorrect shadowing.

Diefenbach [11] in 1996 recognized the problem created by near plane clipping for shadow volume rendering. Diefenbach presents a method that he claims works “for any shadow volume geometry from any viewpoint,” but the method, in fact, does not work in several cases. Figure 1 illustrates three cases where Diefenbach’s method fails.

Another solution to the shadow volume near plane clipping problem mentioned by Diefenbach is capping off the shadow volume’s intersection with the near clip plane. Other authors [2][4][9][16][17] have also suggested this approach. The problem with near plane capping of shadow volumes is that it is, as described by Carmack [9], a “fragile” procedure.

Capping involves projecting each occluder’s back-facing polygons to the near clip plane. This can be complicated when only one or two of a projected polygon’s vertices intersect the near plane and careful plane-plane intersection computations are required in such cases. The capping process is further complicated when a back-facing occluder polygon straddles the near clip plane.

Rendering capping polygons at the near clip plane is difficult because of the razor’s edge nature of the near clip plane. If you are not careful, the very near plane you are attempting to cap can clip your capping polygons! Additionally if the capping polygons are not “watertight” (2-manifold) with the shadow volume being

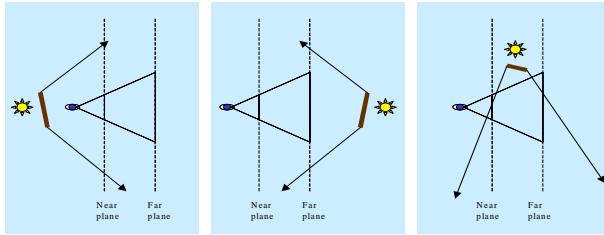


Figure 1: Three cases where Deifenbach’s capping algorithm fails because some or all pixels requiring capping are covered by neither a front-nor back-facing polygon so Diefenbach’s approach cannot correct these pixels.

capped then rasterization cracks or double hitting of pixels can create shadowing artifacts. These artifacts appear as exceedingly narrow regions of the final scene where areas that clearly should be illuminated are shadowed and vice versa. These artifacts are painfully obvious in animated scenes.

Watertight capping is non-trivial, particularly if shadow volumes are drawn using object-space geometry so that fast dedicated vertex transformation hardware can be exploited. Kilgard [16] proposes creating a “near plane ledge” whereby closed capping polygons can be rendered in a way that avoids clipping by the near clip plane even when rendering object-space shadow volume geometry. This approach cedes a small amount of depth buffer precision for the ledge. Additionally shadow volume capping polygons must be rendered twice, incrementing front-faces and decrementing back-facing geometry because the orientation (front- or back-facing) of a polygon can occasionally flip when a polygon of nearly zero area in window space is transformed from object space to window space due to floating-point numerics. Otherwise, shadow artifacts result.

Even when done carefully, shadow volume near plane capping is treacherous because of the fragile nature of required ray-plane intersections and the inability to guarantee identical and bit-exact CPU and GPU floating-point computations. In any case, capping computations burden the CPU with an expensive task that our algorithm obviates.

2.2.3 Zpass vs. Zfail Stenciled Shadow Volumes

The conventional stenciled shadow volume formulation is to **increment** and **decrement** the shadow depth count for front- and back-facing polygons respectively when the depth test **passes**. Bilodeau [5] in 1999 noted that reversing the depth comparison works too. Another version of this alternative formulation is to **decrement** and **increment** the shadow depth count for front- and back-facing polygons respectively depth test **fails** (without reversing the comparison).

Carmack [9] in 2000 realized the equivalence of the two formulations because they both achieve the *same* result, if in the “depth test fail” formulation, the shadow volume is “closed off” at both ends (rather than being open at the ends). Compare the following OpenGL rendering state modifications with the settings for conventional shadow volume rendering in section 2.2.1.

Front-facing shadow volume rendering configuration:

```
StencilOp(KEEP,DECR,KEEP); // decrement on zfail
```

Back-facing configuration:

```
StencilOp(KEEP,INCR,KEEP); // increment on zfail
```

What Carmack describes is projecting back faces, with respect to the light source, some large but finite distance (importantly, still within the far clip plane) and also treating the front faces of the occluder, again with respect to the light source, as a part of the shadow volume boundary too. This still has a problem because when a light source is arbitrarily close to a single occluder polygon, any finite distance used to project out the back faces of the occluder to close off the shadow volume may not extend far enough to ensure that objects beyond the occluder are properly shadowed.

Still Carmack’s insight is fundamental to our new algorithm. We call Bilodeau and Carmack’s approach *zfail* stenciled shadow volume rendering because the stencil increment and decrement operations occur when the depth test fails rather than when it passes. We call the conventional approach *zpass* rendering.

One way to think of the *zfail* formulation, in contrast to the *zpass* formulation, is that the *zfail* version counts shadow volume intersections from the opposite direction. The *zpass* formulation counts shadow volume enters and exits along each pixel’s view ray between the eye point and the first visible rasterized fragment. Technically due to near plane clipping, the counting occurs only between the ray’s intersection point with the near clip plane and the first visible rasterized fragment. The objective of shadow volume capping is to introduce sufficient shadow volume enters so that the eye can always be considered “out of shadow” so the stencil count can reflect the true absolute shadow depth of the first visible rasterized fragment.

The *zfail* formulation instead counts shadow volume enters and exits along each pixel’s view ray between *infinity* and the first visible rasterized fragment. Technically due to far plane clipping, the counting occurs only between the ray’s intersection with the far plane and the first visible fragment. By capping the open end of the shadow volume at or before the far clip plane, we can force the idea that *infinity* is always outside of the shadow volume.

3. OUR ALGORITHM

3.1 Requirements

For our algorithm to operate robustly, we require the following:

- Models for occluding objects must be composed of triangles only (avoiding non-planar polygons), be closed (2-manifold), and have a consistent winding order for triangles within the model. Homogeneous object coordinates are permitted, assuming $w \geq 0$.
- Light sources must be ideal points. Homogeneous light positions ($w \geq 0$) allow both positional and directional lights.
- Connectivity information for occluding models must be available so that silhouette edges with respect to a light position can be determined at shadow volume construction time.
- The projection matrix must be perspective, not orthographic.
- Functionality available in OpenGL 1.0 [18] and DirectX 6: transformation and clipping of homogeneous positions; front and back face culling; masking color and depth buffer writes; depth buffering; and stencil-testing support.
- The renderer must support N bits of stencil buffer precision, where 2^N is greater than the maximum shadow depth count ever encountered during the processing of a given scene.

This requirement is scene dependent, but 8 bits of stencil buffer precision (typical for most hardware today) is reasonable for typical scenes.

- The renderer must guarantee “watertight” rasterization (no double hitting of pixels or missed pixels along shared edges of rasterized triangles).

Support for non-planar polygons and open models can be achieved using special case handling along the lines described by Bergeron [3].

3.2 Approach

We developed our algorithm by methodically addressing the fundamental limitations of the conventional stenciled shadow volume approach. We combine (1) placing the conventional far clip plane “at infinity”; (2) rasterizing infinite (but fully closed) shadow volume polygons via homogeneous coordinates; and (3) adopting the *zfail* stencil-testing scheme.

This is sufficient to render shadow volumes robustly because it avoids the problems created by the far clip plane “slicing open” the shadow volume. The shadow volumes we construct project “all the way to infinity” through the use of homogeneous coordinates to represent the shadow volume’s infinite back projection. Importantly, though our shadow volume geometry is infinite, it is also fully closed. The far clip plane, in eye-space, is infinitely far away so it is impossible for any of the shadow volume geometry to be clipped by it.

By using the *zfail* stencil-testing scheme, we can always assume that infinity is “beyond” all closed shadow volumes if we, in fact, close off our shadow volumes at infinity. This means the shadow depth count can always start from zero for every pixel. We need not worry about the shadow volume being clipped by the near clip plane since we are counting shadow volume enters and exits from infinity, rather than from the eye, due to *zfail* stencil-testing. No fragile capping is required so our algorithm is both robust and automatic.

3.2.1 Far Plane at Infinity

The standard perspective formulation of the projection matrix used to transform eye-space coordinates to clip space in OpenGL (see `glFrustum`[18]) is

$$\mathbf{P} = \begin{bmatrix} \frac{2 \times Near}{Right - Left} & 0 & \frac{Right + Left}{Right - Left} & 0 \\ 0 & \frac{2 \times Near}{Top - Bottom} & \frac{Top + Bottom}{Top - Bottom} & 0 \\ 0 & 0 & \frac{Far + Near}{Far - Near} & -\frac{2 \times Far \times Near}{Far - Near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

where *Near* and *Far* are the respective distances from the viewer to the near and far clip planes in eye-space.

\mathbf{P} is used to transform eye-space positions to clip-space positions:

$$[x_c \ y_c \ z_c \ w_c]^T = \mathbf{P} [x_e \ y_e \ z_e \ w_e]^T$$

We are interested in avoiding far plane clipping so we only concern ourselves with the third and fourth row of \mathbf{P} used to compute clip-space z_c and w_c . Regions of an assembled polygon with interpolated clip coordinates outside $-w_c \leq z_c \leq w_c$ are clipped by the near and far clip planes.

We consider the limit of \mathbf{P} as the far clip plane distance is driven to infinity (this is not novel; Blinn [7] mentions the idea):

$$\lim_{Far \rightarrow \infty} \mathbf{P} = \mathbf{P}_{inf} = \begin{bmatrix} \frac{2 \times Near}{Right - Left} & 0 & \frac{Right + Left}{Right - Left} & 0 \\ 0 & \frac{2 \times Near}{Top - Bottom} & \frac{Top + Bottom}{Top - Bottom} & 0 \\ 0 & 0 & -1 & -2 \times Near \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

The first, second, and fourth rows of \mathbf{P}_{inf} are the same as \mathbf{P} ; only the third row changes. There is no longer a *Far* distance.

A vertex that is an infinite distance from the viewer is represented in homogeneous coordinates with a zero w_e coordinate. If the vertex is transformed into clip space using \mathbf{P}_{inf} , assuming the vertex is in front of the eye, meaning that z_e is negative (the OpenGL convention), then $w_c = z_c$ so this transformed vertex is *not* clipped by the far plane. Moreover, its non-homogeneous depth z_c/w_c must be 1.0, generating the maximum possible depth value.

It may be surprising, but positioning the far clip plane at infinity typically reduces the depth buffer precision only marginally. Consider how much we would need to shrink our window coordinates so we can represent within the depth buffer an infinite eye-space distance in front of the viewer. The projection \mathbf{P} transforms $(0,0,-1,0)$ in eye-space (effectively, an infinite distance in front of the viewer) to the window-space depth $Far/(Far-Near)$. The largest window coordinate representable in the depth buffer is 1 so we must scale $Far/(Far-Near)$ by its reciprocal to “fit” infinity in the depth buffer. This scale factor is $(Far-Near)/Far$ and is very close to 1 if *Far* is many times larger than *Near* which is typical.

Said another way, using \mathbf{P}_{inf} instead of \mathbf{P} only compresses the depth buffer precision slightly in typical scenarios. For example, if *Near* and *Far* are 1 and 100, then the depth buffer’s precision must be squeezed by just 1% to represent an infinite distance in front of the viewer.

3.2.2 Infinite Shadow Volume Polygons

We assume that given a light source position and a closed model with its edge-connectivity, we can determine the subset of *possible silhouette* edges for the model. A possible silhouette edge is an edge shared by two triangles in a model where one of the two triangles faces a given light while the other triangle faces away from the light.

We call these edges “possible silhouette” edges rather than just silhouette edges because these edges are not necessarily boundaries between shadowed and illuminated regions as implied by the conventional meaning of silhouette. It is possible that an edge is an actual silhouette edge, but it is also possible that the edge is itself in shadow.

Assume we have computed the plane equations in the form $Ax+By+Cz+Dw=0$ for every triangle in a given model. The plane equation coefficients must be computed using a vertex ordering consistent with the winding order shared by all the triangles in the model such that $Ax+By+Cz+Dw$ is non-negative when a point (x,y,z,w) is on the front-facing side of the triangle’s plane. Assume we also know the light’s homogeneous position L in the coordinate space matching the plane equations. For each triangle, evaluate $d=AL_x+BL_y+CL_z+DL_w$ for the triangle’s plane equation coefficients and the light’s position. If d is negative, then the

triangle is back-facing with respect to L ; otherwise the triangle is front-facing with respect to L . Any edge shared by two triangles with one triangle front-facing and the other back-facing is a possible silhouette edge.

To close a shadow volume completely, we must combine three sets of polygons: (1) all of the possible silhouette polygon edges extruded to infinity away from the light; (2) all of the occluder's back-facing triangles, with respect to L , projected away from the light to infinity; and (3) all of the occluder's front-facing triangles with respect to L .

Each possible silhouette edge has two vertices A and B , represented as homogeneous coordinates and ordered based on the front-facing triangle's vertex order. The shadow volume extrusion polygon for this possible silhouette is formed by the edge and its projection to infinity away from the light. The resulting quad consists of the following four vertices:

$$\begin{aligned} &\langle B_x, B_y, B_z, B_w \rangle \\ &\langle A_x, A_y, A_z, A_w \rangle \\ &\langle A_x L_w - L_x A_w, A_y L_w - L_y A_w, A_z L_w - L_z A_w, 0 \rangle \\ &\langle B_x L_w - L_x B_w, B_y L_w - L_y B_w, B_z L_w - L_z B_w, 0 \rangle \end{aligned}$$

The last two vertices are the homogeneous vector differences of $A-L$ and $B-L$. These vertices represent directions heading away from the light, explaining why they have w coordinate values of zero. We do assume $A_w \geq 0$, $B_w \geq 0$, $L_w \geq 0$, etc.

When we use a perspective transform of the form \mathbf{P}_{inf} , we can render shadow volume polygons without the possibility that the far plane will clip these polygons.

For each back-facing occluder triangle, its respective triangle projected to infinity is the triangle formed by the following three vertices:

$$\begin{aligned} &\langle A_x L_w - L_x A_w, A_y L_w - L_y A_w, A_z L_w - L_z A_w, 0 \rangle \\ &\langle B_x L_w - L_x B_w, B_y L_w - L_y B_w, B_z L_w - L_z B_w, 0 \rangle \\ &\langle C_x L_w - L_x C_w, C_y L_w - L_y C_w, C_z L_w - L_z C_w, 0 \rangle \end{aligned}$$

where A , B , and C are each back-facing occluder triangle's three vertices (in the triangle's vertex order).

The front-facing polygons with respect to L are straightforward. Given each triangle's three vertices A , B , and C (in the triangle's vertex order), the triangle is formed by the vertices:

$$\begin{aligned} &\langle A_x, A_y, A_z, A_w \rangle \\ &\langle B_x, B_y, B_z, B_w \rangle \\ &\langle C_x, C_y, C_z, C_w \rangle \end{aligned}$$

Together, these three sets of triangles form the closed geometry of an occluder's shadow volume with respect to the given light.

3.3 Rendering Procedure

Now we sketch the complete rendering procedure to render shadows with our technique. Pseudo-code with OpenGL commands is provided to make the procedure more concrete.

1. Clear the depth buffer to 1.0; clear the color buffer.

```
Clear(DEPTH_BUFFER_BIT | COLOR_BUFFER_BIT);
```

2. Load the projection with \mathbf{P}_{inf} given the aspect ratio, field of view, and near clip plane distance in eye-space.

```
float Pinf[4][4];
Pinf[1][0] = Pinf[2][0] = Pinf[3][0] = Pinf[0][1] =
Pinf[2][1] = Pinf[3][1] = Pinf[0][2] = Pinf[1][2] =
Pinf[0][3] = Pinf[1][3] = Pinf[3][3] = 0;
Pinf[0][0] = cotangent(fieldOfView)/aspectRatio;
Pinf[1][1] = cotangent(fieldOfView);
Pinf[3][2] = -2*near; Pinf[2][2] = Pinf[2][3] = -1;
MatrixMode(PRODUCTION); LoadMatrixf(&Pinf[0][0]);
```

3. Load the modelview matrix with the scene's viewing transform.

```
MatrixMode(MODELVIEW); loadCurrentViewTransform();
```

4. Render the scene with depth testing, back-face culling, and all light sources disabled (ambient & emissive illumination only).

```
Enable(DEPTH_TEST); DepthFunc(LESS);
Enable(CULL_FACE); CullFace(BACK);
Enable(LIGHTING); Disable(LIGHT0);
LightModelfv(LIGHT_MODEL_AMBIENT, &globalAmbient);
drawScene();
```

5. Disable depth writes, enable additive blending, and set the global ambient light contribution to zero (and zero any emissive contribution if present).

```
DepthMask(0);
Enable(BLEND); BlendFunc(ONE, ONE);
LightModelfv(LIGHT_MODEL_AMBIENT, &zero);
```

6. For each light source:

- A. Clear the stencil buffer to zero.

```
Clear(STENCIL_BUFFER_BIT);
```

- B. Disable color buffer writes and enable stencil testing with the *always* stencil function and writing stencil..

```
ColorMask(0,0,0,0);
Enable(STENCIL_TEST);
StencilFunc(ALWAYS, 0, ~0); StencilMask(~0);
```

- C. For each occluder:

- a. Determine whether each triangle in the occluder's model is front- or back-facing with respect to the light's position. Update `triList[i].backfacing`.
- b. Configure *zfail* stencil testing to increment stencil for back-facing polygons that fail the depth test.
- c. Render all possible silhouette edges as quads that project from the edge away from the light to infinity.

```
Vert L = currentLightPosition;
```

```
Begin(QUADS);
for (int i=0; i<numTris; i++) // for each triangle
// if triangle is front-facing with respect to the light
if (triList[i].backFacing==0)
    for (int j=0; j<3; j++) // for each triangle edge
        // if adjacent triangle is back-facing
        // with respect to the light
        if (triList[triList[i].adjacent[j]].backFacing) {
            // found possible silhouette edge
            Vert A = triList[i].v[j];
            Vert B = triList[i].v[(j+1) % 3]; // next vertex
```

```

Vertex4f(B.x,B.y,B.z,B.w);
Vertex4f(A.x,A.y,A.z,A.w);
Vertex4f(A.x*L.w-L.x*A.w,
           A.y*L.w-L.y*A.w,
           A.z*L.w-L.z*A.w, 0); // infinite
Vertex4f(B.x*L.w-L.x*B.w,
           B.y*L.w-L.y*B.w,
           B.z*L.w-L.z*B.w, 0); // infinite
}
End(); // quads
d. Specially render all occluder triangles. Project and
    render back facing triangles away from the light to
    infinity. Render front-facing triangles directly.
#define V triList[i].v[j] // macro used in Vertex4f calls
Begin(TRIANGLES);
    for (int i=0; i<numTris; i++) // for each triangle
        // if triangle is back-facing with respect to the light
        if (triList[i].backFacing)
            for (int j=0; j<3; j++) // for each triangle vertex
                Vertex4f(V.x*L.w-L.x*V.w, V.y*L.w-L.y*V.w,
                           V.z*L.w-L.z*V.w, 0); // infinite
        else
            for (int j=0; j<3; j++) // for each triangle vertex
                Vertex4f(V.x,V.y,V.z,V.w);
End(); // triangles
e. Configure zfail stencil testing to decrement stencil
    for front-facing polygons that fail the depth test.
    CullFace(BACK); StencilOp(KEEP,DECR,KEEP);
f. Repeat steps (c) and (d) above, this time rendering
    front facing polygons rather than back facing ones.
D. Position and enable the current light (and otherwise
    configure the light's attenuation, color, etc.).
Enable(LIGHT0);
Lightfv(LIGHT0, POSITION, &currentLightPosition.x);
E. Set stencil testing to render only pixels with a zero
    stencil value, i.e., visible fragments illuminated by the
    current light. Use equal depth testing to update only the
    visible fragments, and then, increment stencil to avoid
    double blending. Re-enable color buffer writes again.
StencilFunc(EQUAL, 0, ~0); StencilOp(KEEP,KEEP,INCR);
DepthFunc(EQUAL); ColorMask(1,1,1,1);
F. Re-draw the scene to add the contribution of the current
    light to illuminated (non-shadowed) regions of the
    scene.
drawScene();
G. Restore the depth test to less.
DepthFunc(LESS);
7. Disable blending and stencil testing; re-enable depth writes.
Disable(BLEND); Disable(STENCIL_TEST); DepthMask(1);

```

3.4 Optimizations

Possible silhouette edges form closed loops. If a loop of possible silhouette edges is identified, then sending QUAD_STRIP primitives (2 vertices/projected quad), rather than independent quads (4 vertices/projected quad) will reduce the per-vertex transformation overhead per shadow volume quad. Similarly, the independent triangle rendering used for capping the shadow volumes can be optimized for rendering as triangle strips or indexed triangles.

The **INCR** *zpass* stencil operation in step 6.E avoids the double blending of lighting contributions in the usually quite rare circumstance when two fragments alias to the exact same pixel location and depth value. Using the **KEEP** *zpass* stencil operation instead can avoid usually unnecessary stencil buffer writes, improving rendering performance in situations where double blending is deemed unlikely.

In the case of a directional light, all the vertices of a possible silhouette edge loop project to the same point at infinity. In this case, a **TRIANGLE_FAN** primitive can render these polygons extremely efficiently (1 vertex/projected triangle).

If the application determines that the shadow volume geometry for a silhouette edge loop will never pierce or otherwise require capping of the near clip plane's visible region, *zpass* shadow volume rendering can be used instead of *zfail* rendering. The *zpass* formulation is advantageous in this context because it does not require the rendering of any capping triangles. Mixing the *zpass* and *zfail* shadow volume stencil testing formulations for different silhouette edge loops does not affect the net shadow depth count as long as each particular loop uses a single formulation.

Shadow volume geometry can be re-used from frame to frame for any light and occluder that have not changed their geometric relationship to each other.

4. IMPROVED HARDWARE SUPPORT

4.1 Wrapping Stencil Arithmetic

DirectX 6 and the OpenGL *EXT_stencil_wrap* extension provide two additional *increment wrap* and *decrement wrap* stencil operations that use modulo, rather than saturation, arithmetic. These operations reduce the likelihood of incorrect shadow results due to an increment operation saturating a stencil value's shadow depth count. Using the wrapping operations with an N -bit stencil buffer, there remains a remote possibility that a net 2^N increments (or a multiple of) may alias with the unshadowed zero stencil value and lead to incorrect shadows, but in practice, particularly with an 8-bit stencil buffer, this is quite unlikely.

4.2 Depth Clamping

NVIDIA's GeForce3 and GeForce4 Ti GPUs support *depth clamping* via the *NV_depth_clamp* OpenGL extension. When enabled, depth clamping disables the near and far clip planes for rasterizing geometric primitives. Instead, a fragment's window-space depth value is clamped to the range $[\min(zn,zf), \max(zn,zf)]$ where zn and zf are the near and far depth range values. Additionally when depth clamping is enabled, no fragments with non-positive w_c are generated.

With depth clamping support, a conventional projection matrix with a finite far clip plane distance can be used rather than the \mathbf{P}_{inf} form. The only required modification to our algorithm is enabling **DEPTH_CLAMP_NV** during the rendering of the shadow volume geometry.

Depth clamping recovers the depth precision (admittedly quite marginal) lost due to the use of a \mathbf{P}_{inf} projection matrix. More significantly, depth clamping generalizes our algorithm so it works with orthographic, not just perspective, projections.

4.3 Two-Sided Stencil Testing

We propose *two-sided stencil testing*, a new stencil functionality that uses distinct front- and back-facing stencil state when enabled. Front-facing primitives use the front-facing stencil state for their stencil operation while back-facing primitives use the back-facing state. With two-sided stencil testing, shadow volume geometry need only be rendered once, rather than twice.

Two-sided stencil testing generates the same number of stencil buffer updates as the two-pass approach so in fill-limited shadow volume rendering situations, the advantage of a single pass is marginal. However, pipeline bubbles due to repeated all front-facing or all back-facing shadow volumes lead to inefficiencies using two passes. Perhaps more importantly, two-sided stencil testing reduces the CPU overhead in the driver by sending shadow volume polygon geometry only once.

Because stencil increments and decrements are intermixed with two-sided stencil testing, the wrapping versions of these operations are mandatory.

5. EXAMPLES

Figures 2 through 5 show several examples of our algorithm.

6. FUTURE WORK

Because of the extremely scene-dependent nature of shadow volume rendering performance and space constraints here, we defer thorough performance evaluation of our technique. Still we are happy to report that our rendering examples, including examples that seek to mimic the animated behavior of a sophisticated 3D game (see Figure 4), achieve real-time rates on current PC graphics hardware.

Yet naïve rendering with stenciled shadow volumes consumes tremendous amounts of stencil fill rate. We expect effective shadow volume culling schemes will be required to achieve consistent interactive rendering rates for complex shadowed scenes. Portal, BSP, occlusion, and view frustum culling techniques can all improve performance by avoiding the rendering of unnecessary shadow volumes. Additional performance scaling will be through faster and cleverer hardware designs that are better tuned for rendering workloads including stenciled shadow volumes.

Future graphics hardware will support more higher-order graphics primitives beyond triangles. Combining higher-order hardware primitives with shadow volumes requires automatic generation of shadow volumes in hardware. Two-sided stencil testing will be vital since it only requires one rendering of automatically generated shadow volume geometry. Automatic generation of shadow volumes will also relieve the CPU of this chore.

7. CONCLUSIONS

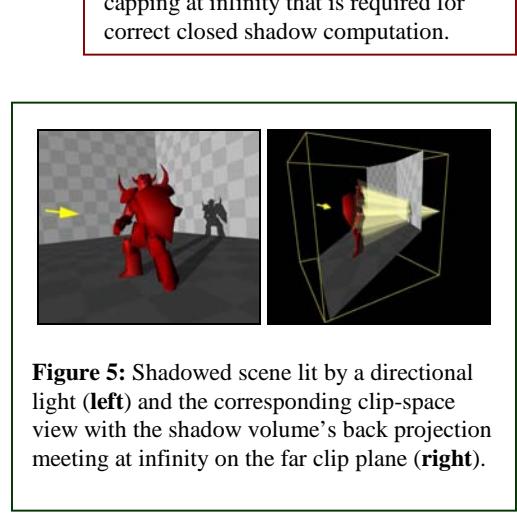
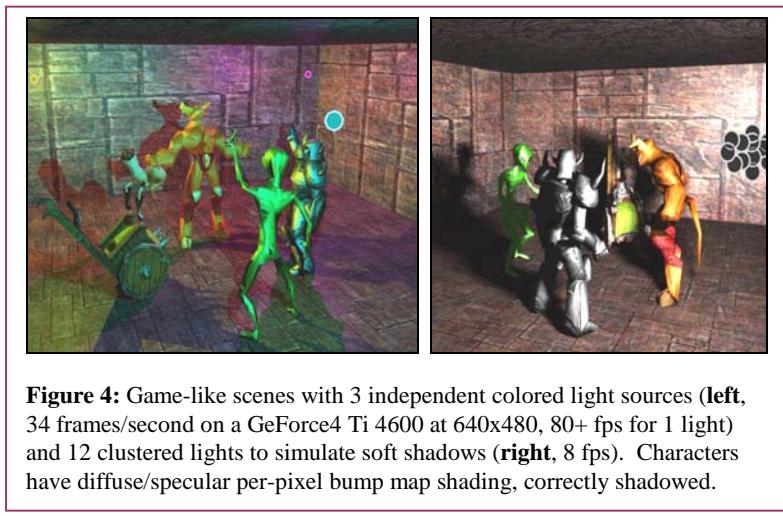
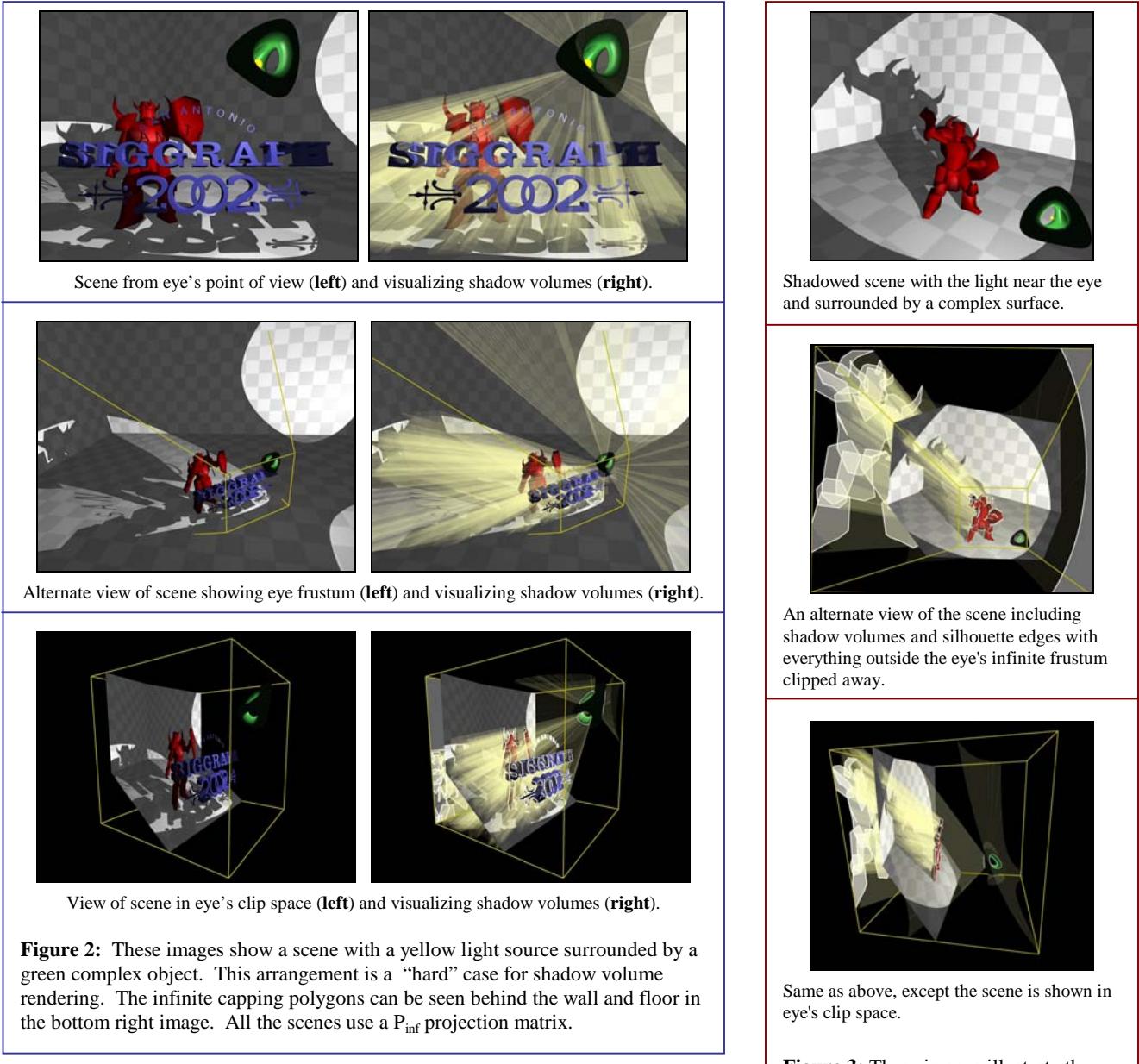
Our stenciled shadow volume algorithm is robust, straightforward, and requires hardware functionality that is ubiquitous today. We believe this will provide the opportunity for 3D games and applications to integrate shadow volumes into their basic rendering repertoire. The algorithm we developed is the result of careful integration of known, but not previously integrated, techniques to address methodically the shortcomings of existing shadow volume techniques caused by near and/or far plane clipping.

8. ACKNOWLEDGEMENTS

We are grateful to John Carmack, Matt Craighead, Eric Haines, and Matt Papakipos for fruitful discussions, Steve Burke for modeling the SIGGRAPH logo and Loop subdivision surface for us; and James Green, Brian Collins, Rich B, and Stecki for designing and animating the Quake2 models that we picture.

REFERENCES

- [1] Kurt Akeley and James Foran, "Apparatus and method for controlling storage of display information in a computer system," *US Patent 5,394,170*, filed Dec. 15, 1992, assigned Feb. 28, 1995.
- [2] Harlen Costa Batagelo and Ilaim Costa Junior, "Real-Time Shadow Generation Using BSP Trees and Stencil Buffers," *XII Brazilian Symposium on Computer Graphics and Image Processing*, Campinas, Brazil, Oct. 1999, pp. 93-102.
- [3] Philippe Bergeron, "A General Version of Crow's Shadow Volumes," *IEEE Computer Graphics and Applications*, Sept. 1986, pp. 17-28.
- [4] Jason Bestimt and Bryant Freitag, "Real-Time Shadow Casting Using Shadow Volumes," Gamasutra.com web site, Nov. 15, 1999.
- [5] Bill Bilodeau and Mike Songy, Creative Labs sponsored game developer conference, unpublished slides, Los Angeles, May 1999.
- [6] David Blythe, Tom McReynolds, et.al., "Shadow Volumes," *Program with OpenGL: Advanced Rendering*, SIGGRAPH course notes, 1996.
- [7] Jim Blinn, "A Trip Down the Graphics Pipeline: The Homogeneous Perspective Transform," *IEEE Computer Graphics and Applications*, May 1993, pp. 75-88.
- [8] Lynne Brotman and Norman Badler, "Generating Soft Shadows with a Depth Buffer Algorithm," *IEEE Computer Graphics and Applications*, Oct. 1984, pp. 5-12.
- [9] John Carmack, unpublished correspondence, early 2000.
- [10] Frank Crow, "Shadow Algorithms for Computer Graphics," *Proceedings of SIGGRAPH*, 1977, pp. 242-248.
- [11] Paul Diefenbach, *Multi-pass Pipeline Rendering: Interaction and Realism through Hardware Provisions*, Ph.D. thesis, University of Pennsylvania, tech report MS-CIS-96-26, 1996.
- [12] Alain Fournier and Donald Fussell, "On the Power of the Frame Buffer," *ACM Transactions on Graphics*, April 1988, 103-128.
- [13] Henry Fuchs, Jack Goldfeather, Jeff Hultquist, Susan, Spach, John Austin, Frederick Brooks, John Eyles, and John Poulton, "Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes," *Proceedings of SIGGRAPH*, 1985, pp. 111-120.
- [14] Tim Heidmann, "Real Shadows Real Time", *IRIS Universe*, Number 18, 1991, pp. 28-31.
- [15] Mark Kilgard, "Improving Shadows and Reflections via the Stencil Buffer," *Advanced OpenGL Game Development* course notes, Game Developer Conference, March 16, 1999, pp. 204-253.
- [16] Mark Kilgard, "Robust Stencil Volumes," CEDEC 2001 presentation, Tokyo, Sept. 4, 2001.
- [17] Michael McCool, "Shadow Volume Reconstruction from Depth Maps," *ACM Transactions on Graphics*, Jan. 2001, pp. 1-25.
- [18] Mark Segal and Kurt Akeley, *The OpenGL Graphics System: A Specification*, version 1.3, 2001.
- [19] Andrew Woo, Pierre Poulin, and Alain Fournier, "A Survey of Shadow Algorithms," *IEEE Computer Graphics and Applications*, Nov. 1990, pp. 13-32.



A Soft Shadow Volume Algorithm



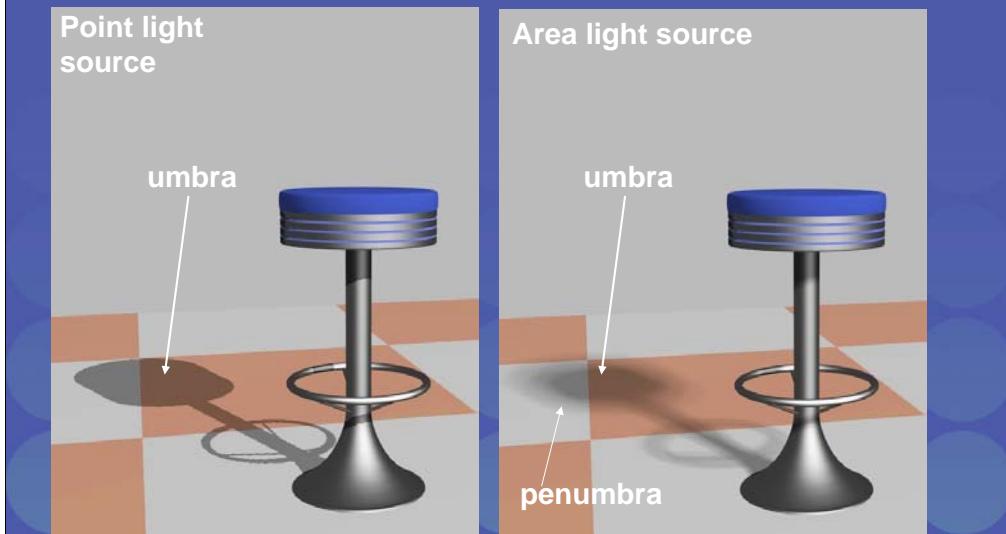
- Contents of my presentation
 - Motivation
 - Penumbra wedges
 - Visibility calculations
 - Implementations (SW/HW)
 - Load balancing
 - Disadvantages
 - Comparison: Hard vs Soft
 - Live demo



Q1: Why soft?



- Answer #1:
 - Soft shadows are addictive!



Reasons for soft shadows:

- 1) increases the level of realism of the rendered images – the large majority of all light sources have some extensions in space (even the sun)
- 2) spatial relationships get even simpler to determine for a human, since sharp shadow edges imply that the shadow caster is close to the receiver, and vice versa.
- 3) puts off the focus from the shadows, i.e., a hard shadow can sometimes be misinterpreted for a geometrical edge, but that is hardly ever the case with soft shadows
- 4) Atmosphere: imagine a setting sun...

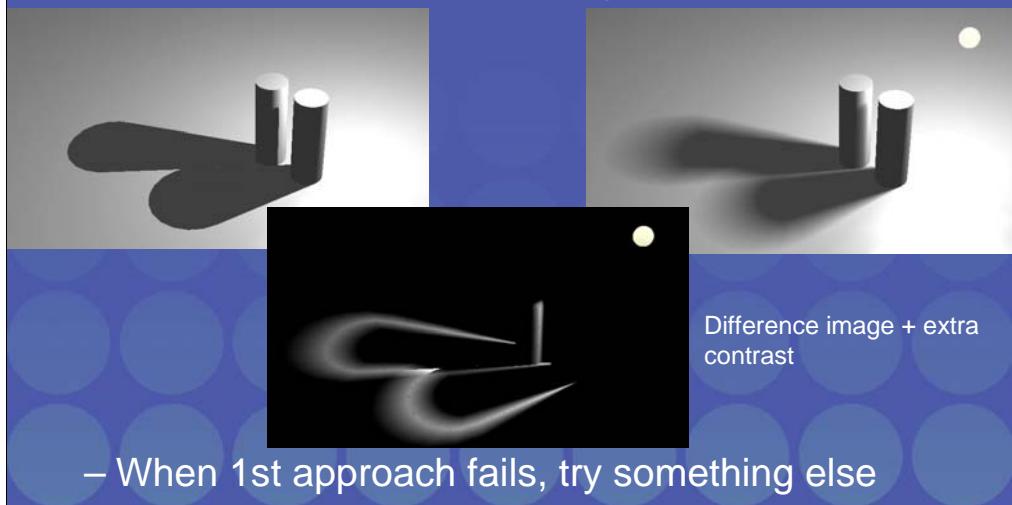
For animated real-time graphics, the addiction is even more severe...

Q2: Why shadow volumes?



- Answer #2:

- Very hard to make the size of the umbra decrease with shadow map



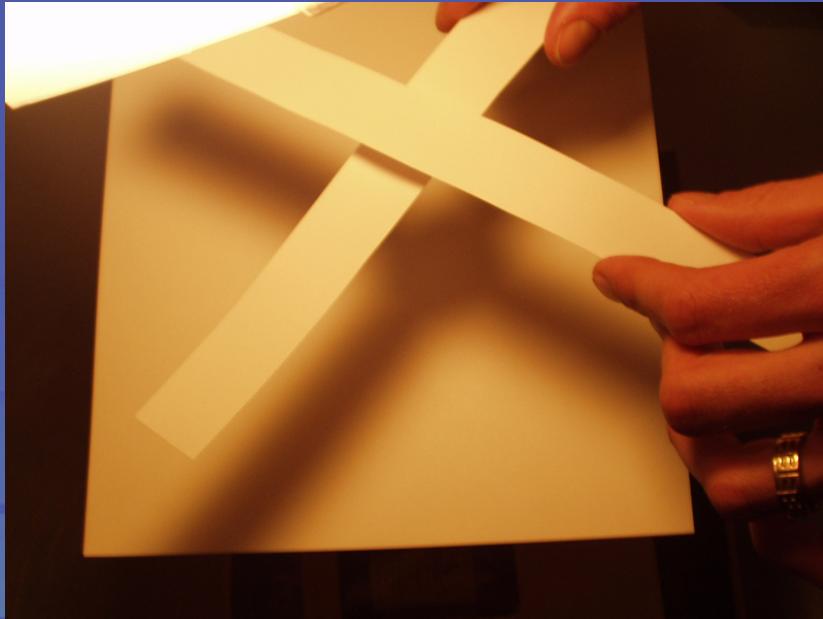
- When 1st approach fails, try something else

Reasons for shadow volumes:

- 1) Shadow maps seemed harder to extend into handling soft shadows, that is, the size of the umbra must decrease when light source size increases
- 2) The complexity of shadow volumes are often considered to be much higher, but in research one should not really put any limits on what to do research on (if the results are convincing enough, but too slow, then let's do research on accelerating the shadow rendering)

Introducing my graphics laboratory...

SIGGRAPH2004



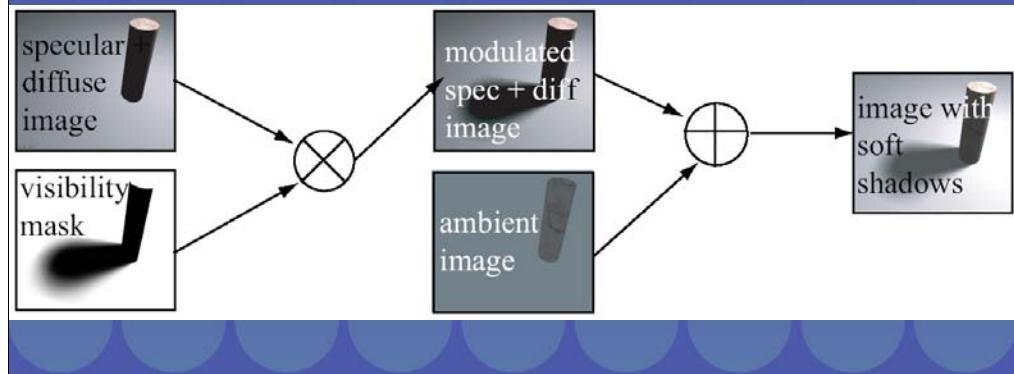
Do this in the kitchen, and you'll learn stuff that is hard to learn at other places!

Cut a square hole in a hard paper, and place a thin paper over the hole. This thin paper should spread the light diffusely. Place the hard + thin paper construction over the light source. Turn off all other light sources, then play with various shadow casters and receivers.

Soft shadow algorithms in general



- Fundamental visibility problem in CG
 - Soft shadow rendering and view cell occlusion culling are essentially the same thing
 - Inherently difficult



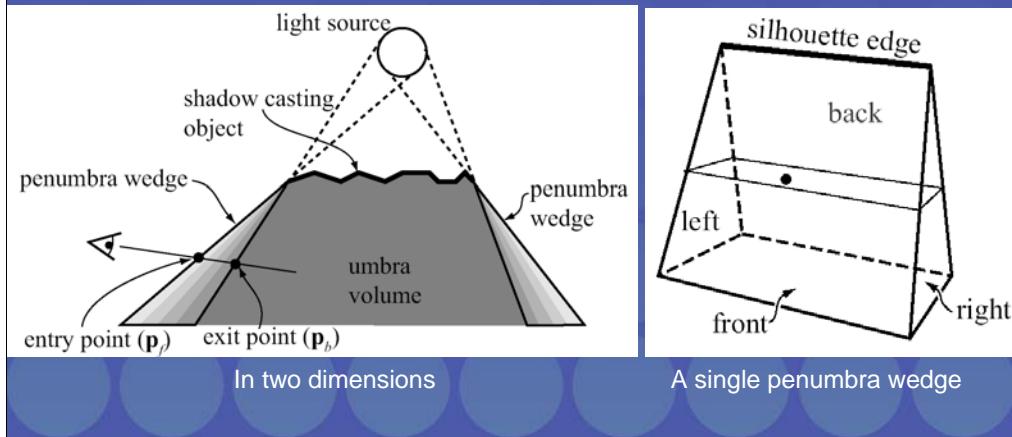
Lots of research on this every year

Often at least one SIGGRAPH paper on the topic

Difficult because need to have visibility information for each point to be shaded to every point on the area light source

The general idea: Penumbra Wedges

- A new primitive for bounding the penumbra region imposed by a silhouette edge



This is the basic INSIGHT behind our algorithm:

Penumbra wedges are part of the core of the algorithm, without the penumbra wedge, we will have a hard time implementing this algorithm.

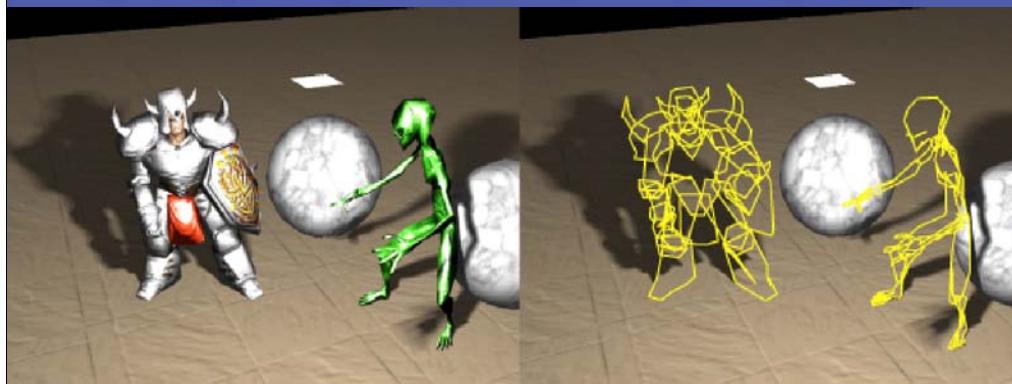
The nice thing about the penumbra wedge is that it is possible to rasterize them quite efficiently using today's graphics hardware. This is true, even though the wedge is a 3D entity (not 2D, like a triangle or quad).

Can use more planes than just 5 if you want to.

Important simplification



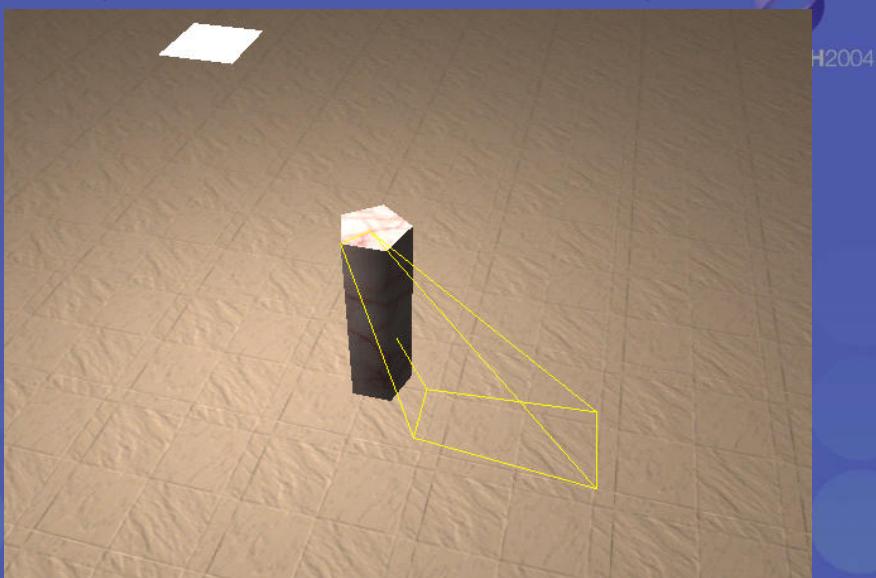
- Compute potential silhouette edges as seen from the center of the light source
- Use one penumbra wedge per such edge



This simplification makes the problem much simpler to solve.

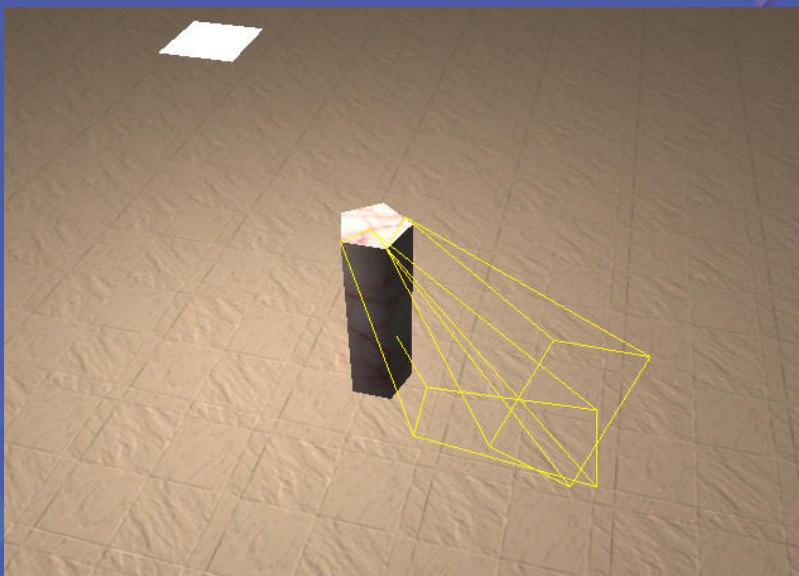
Disadvantage: popping can occur for simple objects, such as a cube.

A wedge for each silhouette edge...



A wedge is generated for each silhouette edge, enclosing a part of the penumbra region .

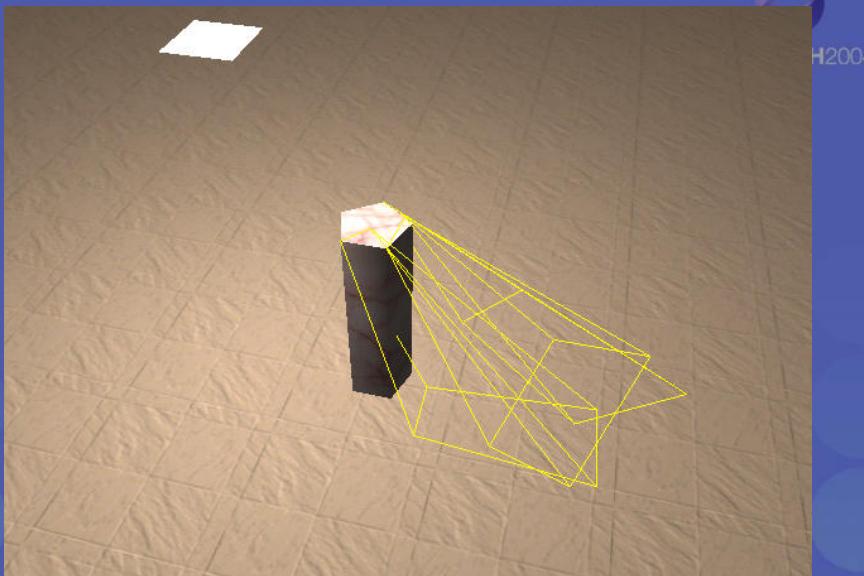
A wedge for each silhouette edge...



Together, the wedges will enclose the whole penumbra region.

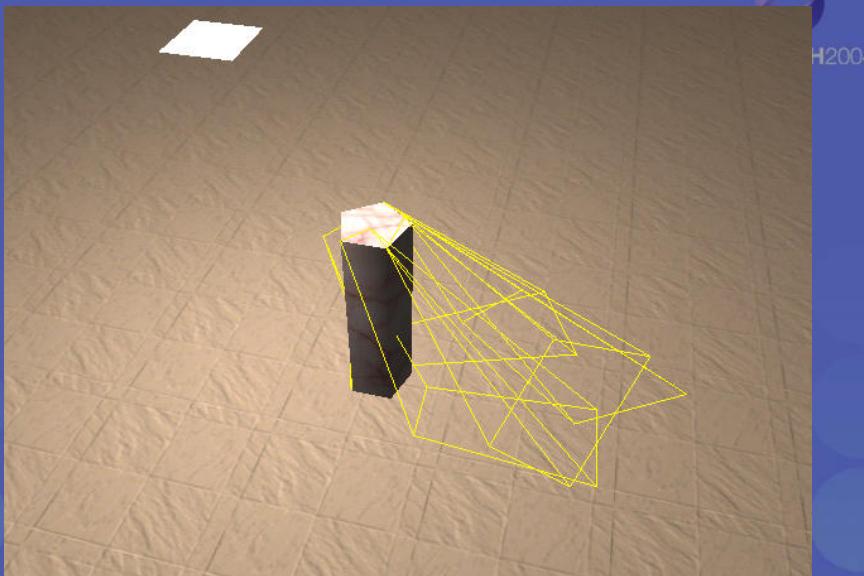
They don't have to correspond exactly to the penumbra region – it is sufficient that they enclose it. And this is a major advantage of our algorithm, since the correct penumbra region can be complicated to compute.

A wedge for each silhouette edge...



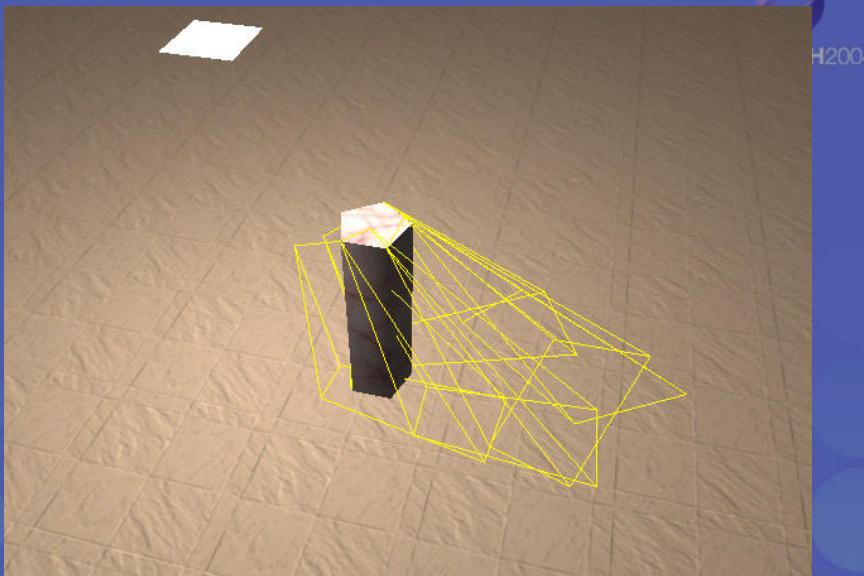
H2004

A wedge for each silhouette edge...



H2004

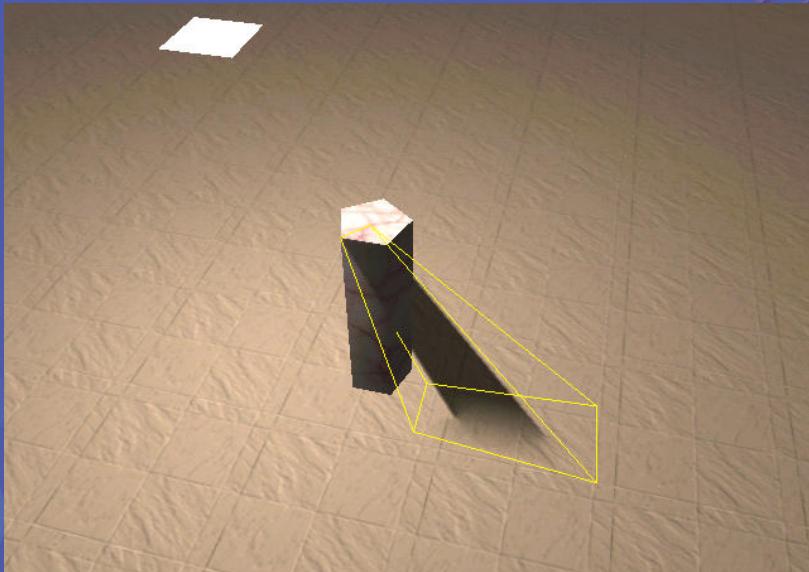
A wedge for each silhouette edge...



Rasterizing the wedges



H2004



We then rasterize each wedge with a pixel shader.

The scene is first rendered into the frame buffer and z-buffer.

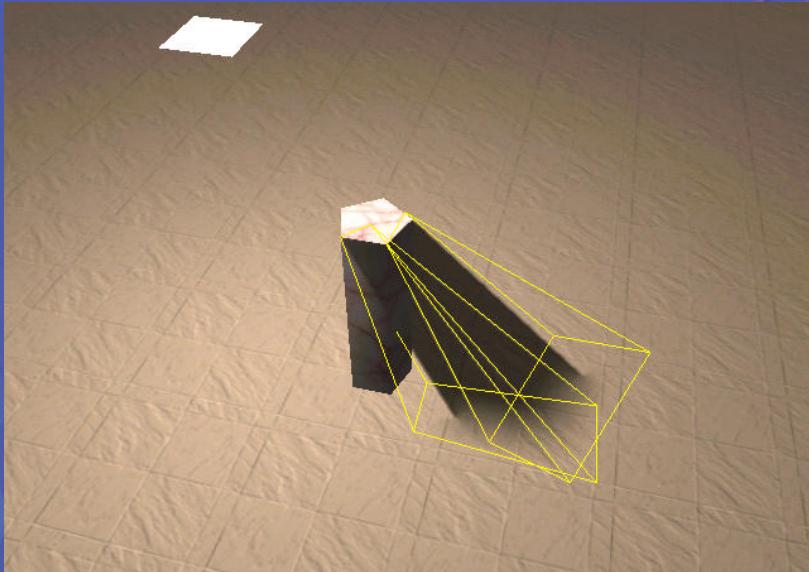
The umbra and penumbra contribution is then rasterized, wedge by wedge by our algorithm. The pixel shader reads out a point from the z-buffer and uses that point to compute a shadow contribution that is stored in a separate buffer. And this buffer is later used to modify the whole frame buffer to "add" on the soft shadows to the image.

I have here visualized the rendering of the umbra contribution and penumbra contribution simultaneously. Typically, we use Crow's shadow volume algorithm for hard shadows first to fill the umbra, and then compensate with our penumbra pass.

Rasterizing the wedges



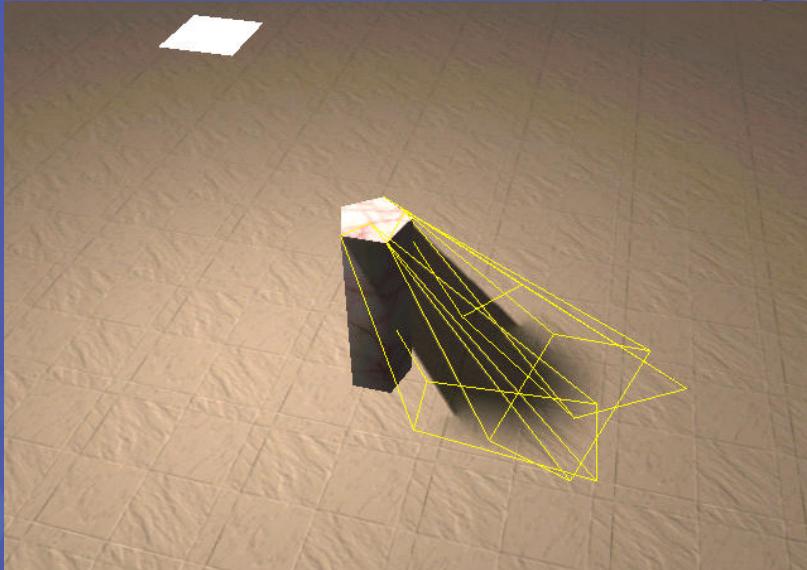
H2004



Rasterizing the wedges



H2004

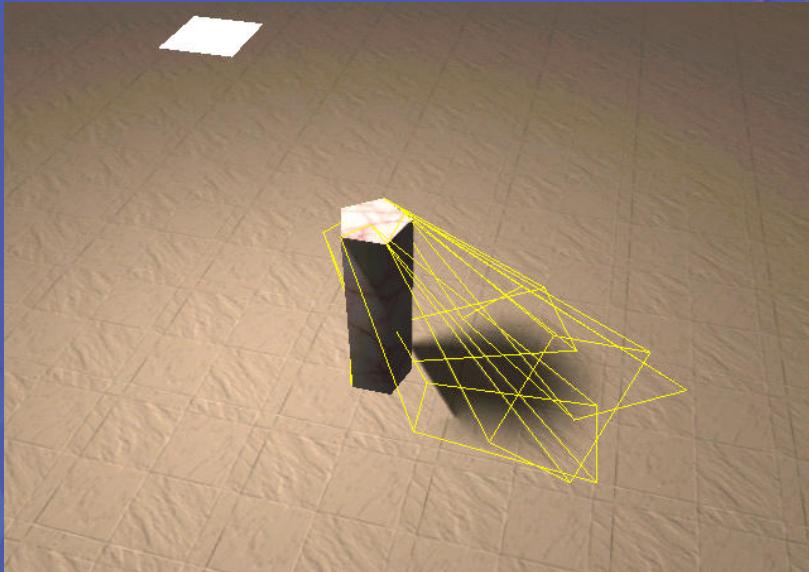


And as they are rasterized, the final soft shadow will gradually appear.

Rasterizing the wedges



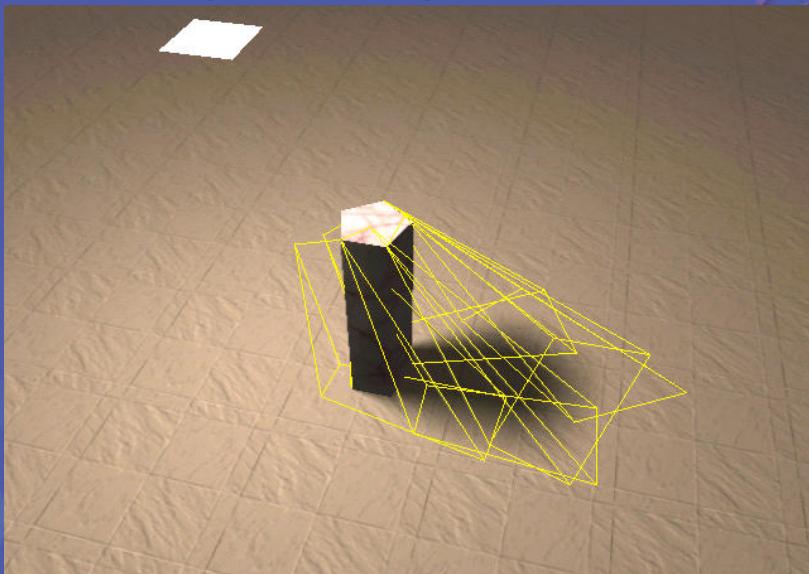
H2004



Rasterizing the wedges



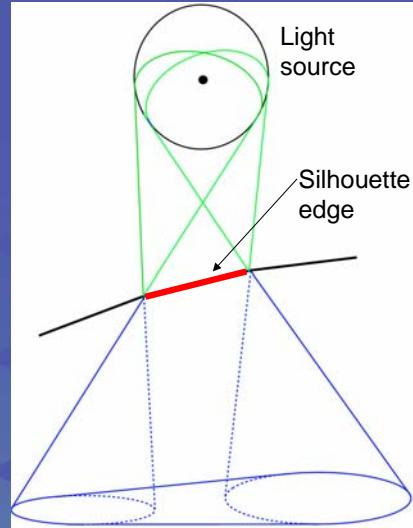
H2004



Wedge construction: in theory

SIGGRAPH2004

1. Form a cone from light source to edge vertex
2. Sweep the vertex from one end to the other
3. The swept surface is the "best" wedge
 - Not planar: consists of Coon's patches

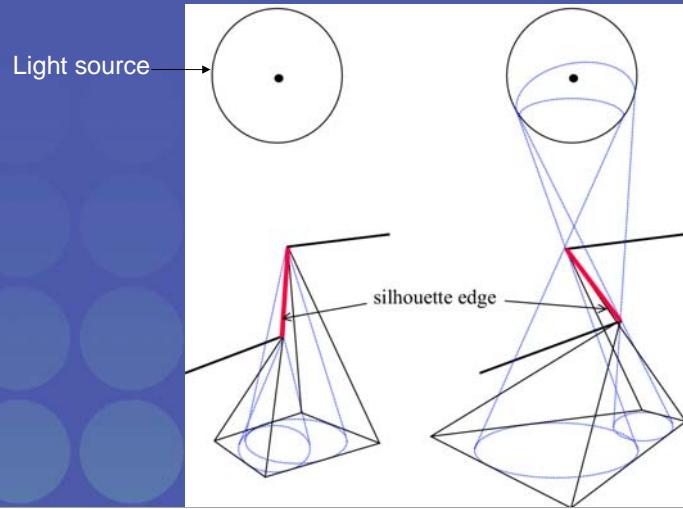


Can do exactly this with any light source.

Wedge construction: in practice



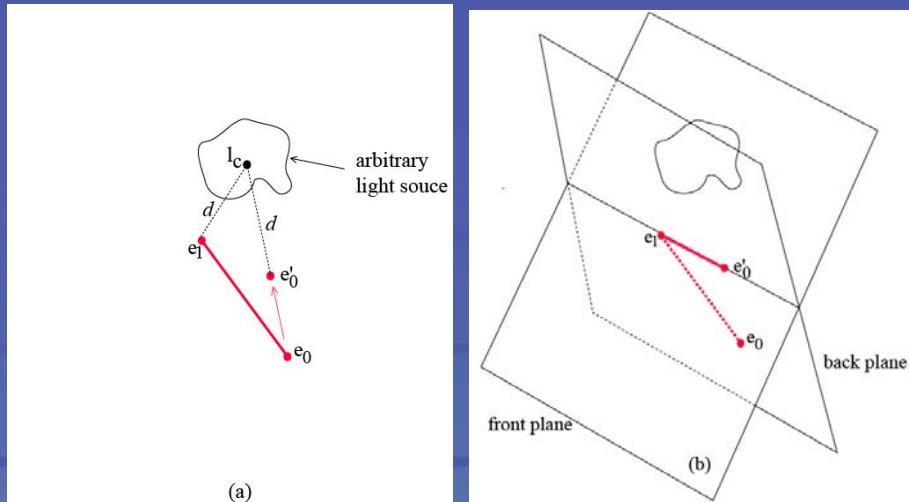
- Must consist of planar faces
- Must be robust and handle all weird cases



Must be planar because we need to rasterize them!

These cases are difficult because either one cone intersects the other edge end point or cones intersect with each other or both.

Practical wedge construction (1)



A: first move the farthest edge end point (in this case e_0) towards the center of the light source, and stop when the new point is as far from l_c as the other edge end point e_1 . This makes it possible to get planar faces on the wedge.

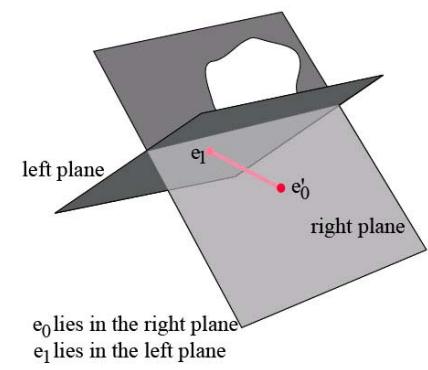
B: Place two planes that passes through the edge $e_0' \rightarrow e_1$.

Rotate the first plane until it barely intersects with one far side of the light source.

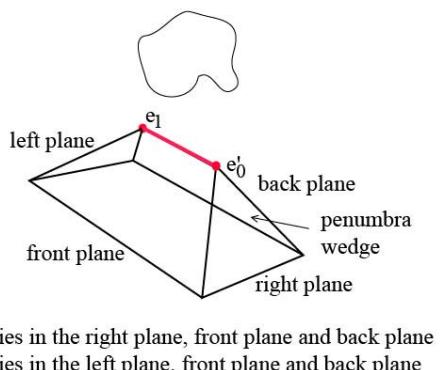
Rotate the other plane until it barely intersect with the other side of the light source.

At this point we have created two of the planes of the wedge, namely, the front and back plane.

Practical wedge construction (2)



(c)



(d)

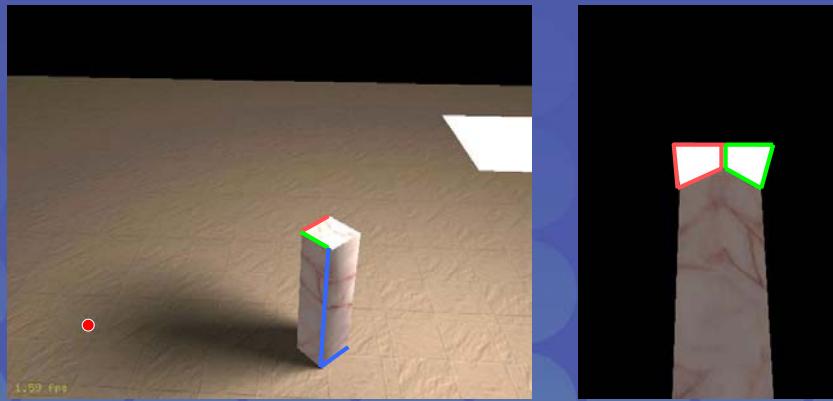
C: Create left and right planes.

Example: for the right plane, create a plane that can rotate around an axis that goes through e'_0 and passes through the vector which is formed as the cross product of $e_1 - e'_0$ and $I_c - e'_0$ (where I_c is the center of the light). Then rotate this plane until it barely touches the far side of the light. Do similar things to create the left plane.

D: at this point all planes have been created, and we might limit the extension of the wedge by placing a bottom plane for the wedge as well.

Visibility calculations

- Really want to compute how much of the light source that we can see



In the left image, the silhouette edges as seen from the center of the light source are marked with blue, green, and red lines.

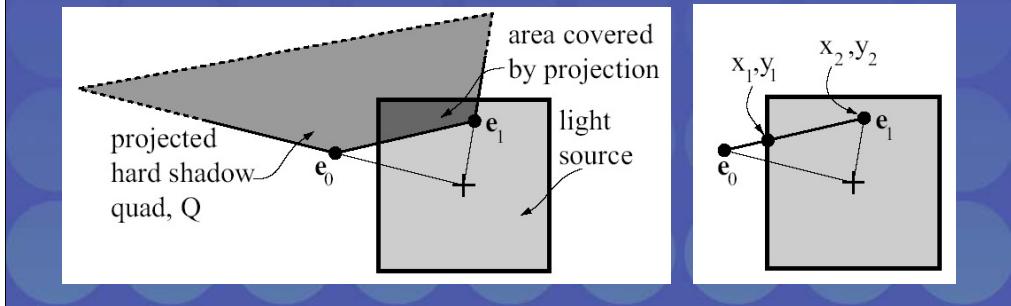
Imagine that you can jump down to the red dot, and look up towards the light source. Then you see the image to the right.

The only silhouettes that project onto the light as seen from the red dot is the red and green silhouette edges.

They each compute a contribution of how much they can "see" of the light source.

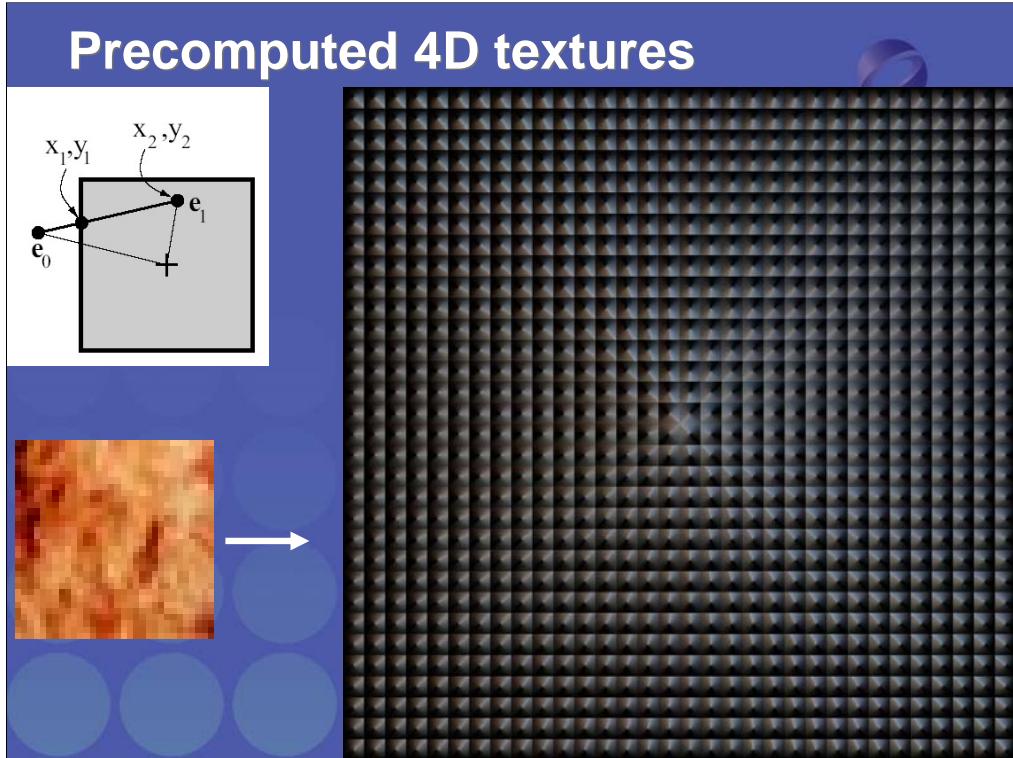
Each silhouette edge's contribution to visibility

- From the point-to-be-shaded, project the edge onto the light source
- Compute the area, called *coverage*, of the dark gray region
- Add/subtract to a visibility buffer



The visibility buffer will be described in detail later

Precomputed 4D textures



Next we clip the projected edge against the borders of the light source (in this case a square light).

This gives us four coordinates: x_1, y_1, x_2, y_2 .

And thus we can precompute the coverage based on these coordinates into a 4D texture (which when flattened out

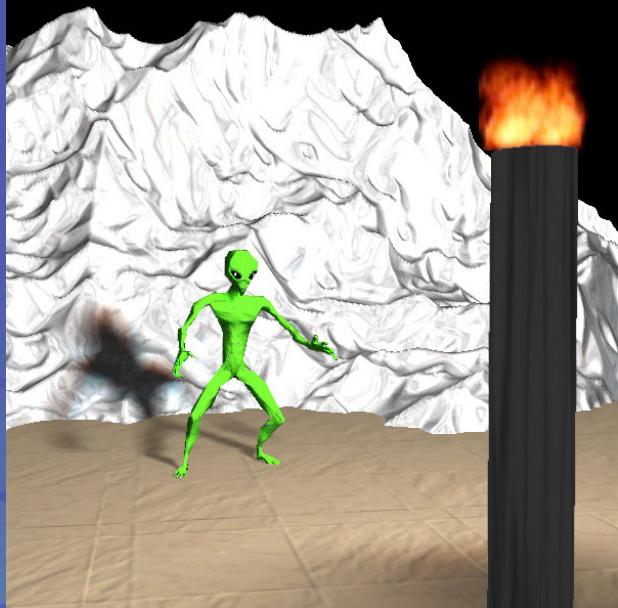
Looks like the image to the right).

In fact, this even allows us use colored textures as light sources. Just precompute the sum of the colors of the coverage area (dark gray in previous slide).

Why 4D textures?

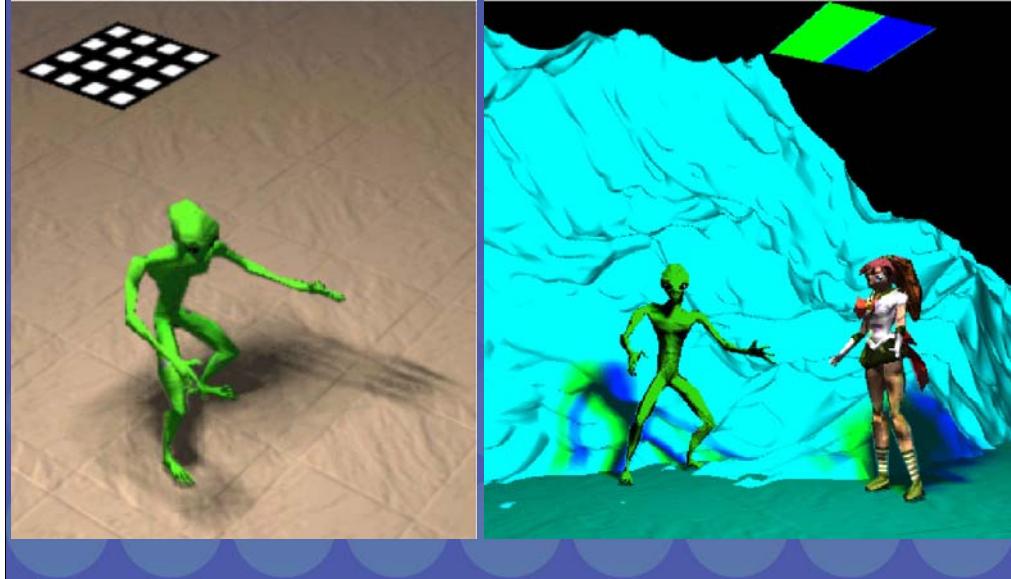
SIGGRAPH2004

- Used as a look-up table
- Due to intelligent caches → fast!
- Can have textured lights (and animated)
- 32x32 light texture → 3 MB



Higher resolution would give better quality, but seldom a problem in practice.

Some examples using textured light sources



Left: this image was rendered using a single light source, but the texture on it shows 4x4 small area light sources. Due to the precomputed 4D texture, this can be handled in one pass.

Right: a simpler case that shows that we're doing the right thing.

A soft shadow volume algorithm



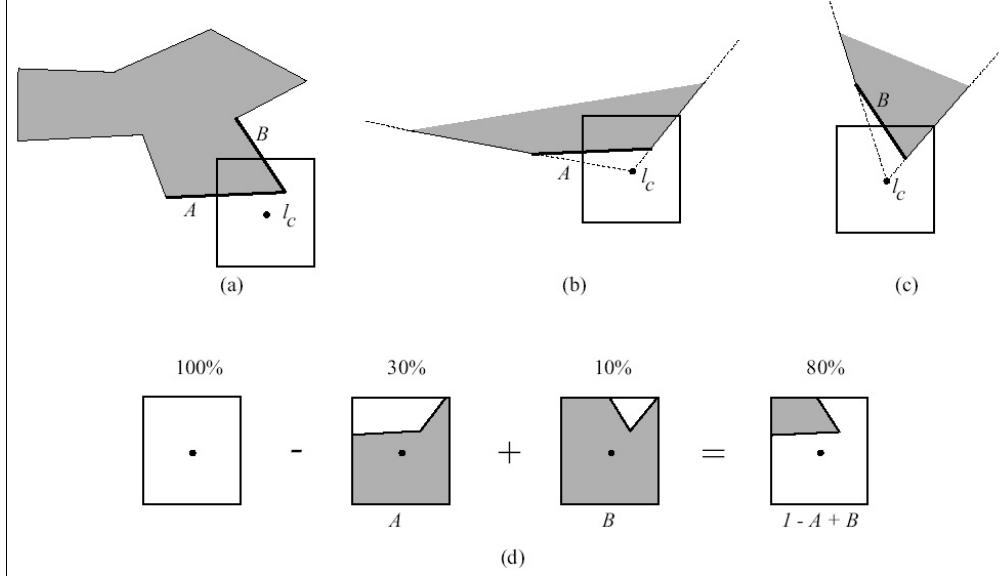
- 1st pass: Render hard shadow quads (as usual)
 - To be sure that we register when we enter/exit umbra
- 2nd pass: compensate for overstated umbra

```
1: rasterizeWedge(wedge W, hard shadow quad Q, light L)
2: for each pixel (x,y) covered by front facing triangles of wedge
3:   p = point(x,y,z); // z is depth buffer value
4:   if p is inside the wedge
5:     v_p = projectQuadAndComputeCoverage(W,p,Q);
6:     if p is in positive half space of Q
7:       v̄(x,y) = v̄(x,y) - v_p; // update V-buffer
8:     else
9:       v̄(x,y) = v̄(x,y) + v_p; // update V-buffer
10:    end;
11:  end;
12: end;
```

The coverage can be at most 0.5, and the hard pass adds 1.0 when whe enter the hard shadows.

Therefore, when the point is in the positive half space of Q, we need to subtract the coverage, and otherwise add it.

Example of how it works...



A: the square is a light source with center I_c , and the gray polygon is a shadow casting object. Imagine that we're at the point to be shaded a look up against the shadow caster (the gray object) and the light source.

The only projected edges that can influence visibility is A and B , and their respective contributions are shown in figure B and C.

D: At the bottom we show the light source as fully visible at first (leftmost image), and then subtract the contribution of A due to its orientation with respect to I_c , and then because the orientation is reversed for B , we add its coverage value. The result is the expected visibility of the light source.



SW vs HW implementation

- Implemented everything in SW due to lack of HW
- Performance was poor when we finally got a highly programmable graphics card
 - $fps(\text{SW rendering}) > fps(\text{HW rendering})!$
- Had to fine tune the implementation:
 - Tighter wedges for rectangular lights
 - Optimized pixel shaders
 - Frame buffer blending
 - Culling

We will not cover the "tighter wedges" – see our Graphics Hardware 03 paper.

Implementation – overview



- Compute position buffer (once per frame)
- Compute hard shadows into V-buffer
 - Give overestimation of umbra
- Correct for that by rendering the wedges
 - Use culling for faster rendering
 - Split into 2 buffers: one additive, one subtractive
- Additive – subtractive = shadow mask
- Combine shadow mask with rendered scene

There are several limitations on current graphics hardware that makes the implementation a bit awkward.

This outline shows how we currently do it, but that can change with a newer graphics card.

Position buffer

- The subsequent fragment programs need the (x,y,z) of each pixel
- Interpolate world space coordinates over each triangle (using texture coordinates)
- One vertex & fragment program compute the (x,y,z) and stores in a floating point pixel buffer $(R,G,B)=(x,y,z)$

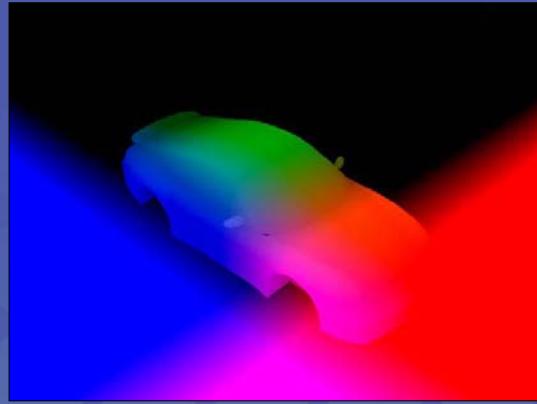


Image courtesy of Jonas Svensson and Ulf Borgenstam

The position buffer is computed once at the beginning of each frame

V-buffer (1)

- Need to hold the coverage for each pixel
 - A number from 0.0 to 1.0 would be nice
 - 0.0 == full shadow, >1.0 no shadow
 - Must use >8 bits blending – how?
- Practice:
 - One buffer for positive values, one for negative
 - V-buffer = "additive buffer" – "subtractive buffer"
 - Each is 8 bits RGBA
 - Only deals with gray scale textured lights
 - This is the current solution – will evolve...

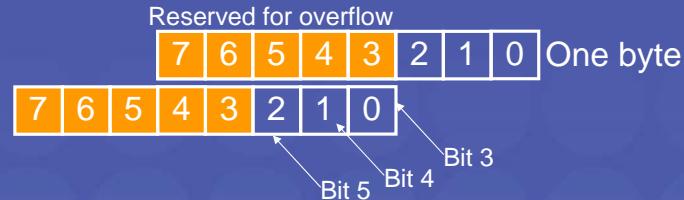
Normal stencil shadows use 8 bits of stencil per pixel. This is so we can have several overlapping shadow volumes.

Here each coverage value should be somewhere between 0 and 255, and we would need

Overlapping objects as well. This we need about 12-16 bits per pixel.

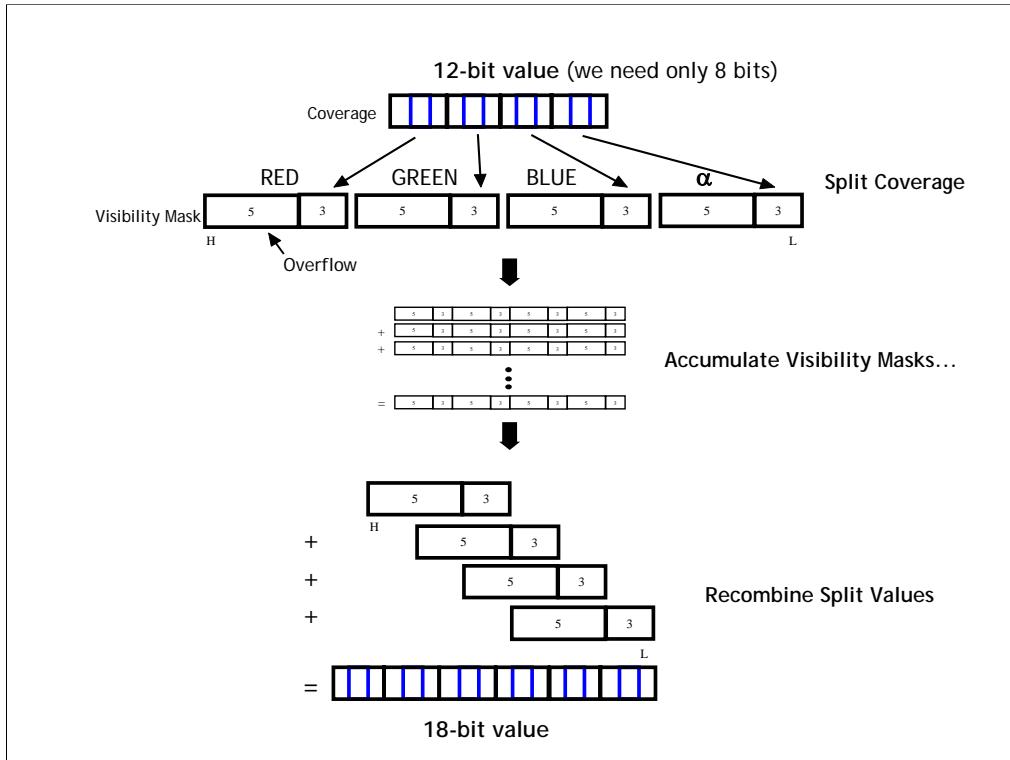
V-buffer (2)

- Reserve highest 5 bits for overflow, and store part of value in 3 bits (per RGBA)



- 1D texture is used to split an incoming coverage value into 4 pieces of 3 bits each
- The split values are recombined with a dot product that scale each channel and adds them together

5 bits for overflow means that we can have <32 overlapping objects!



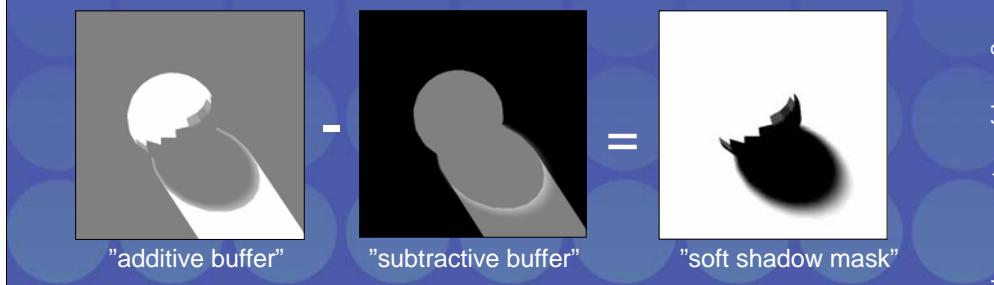
The last step shows how the RGBA is combined with a dot product to form the hires value that we need.

V-buffer (4)



SIGGRAPH2004

- Clear V-buffer: 1.0 → additive, 0.0 → subtractive == fully lit
- Then, rasterize standard SV polys into V-buffer
 - +1.0 for frontfacing → subtractive buffer
 - +1.0 for backfacing → additive buffer
- Rasterize penumbra wedges



Images courtesy of Jonas Svensson and Ulf Borgensjöam

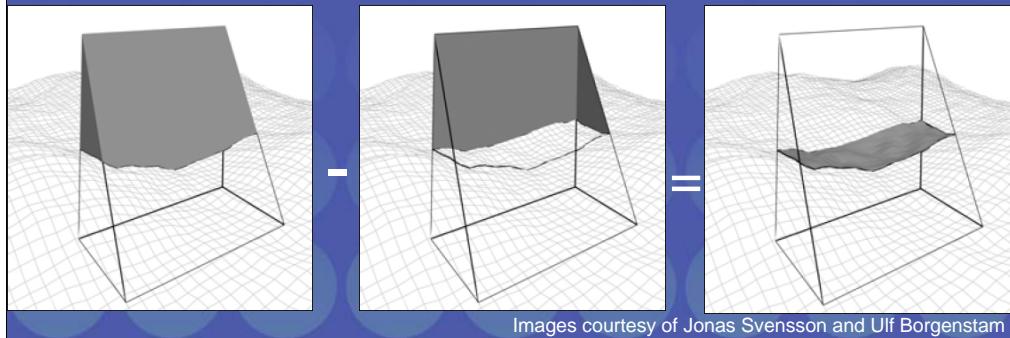
Assume ZFAIL!

The result: ADD – SUB = FINAL SHADOW MASK

Penumbra wedge rasterization



- Render frontfacing triangles of wedge
 - Execute fragment shader that computes coverage
- Fragment shader expensive → use culling
 - Only want to execute shader for (x,y,z) inside wedge
- Front facing & pass depth → +1 stencil
- Back facing & pass depth → -1 stencil



Images courtesy of Jonas Svensson and Ulf Borgenstam

INSIGHT: use standard shadow value algorithm on the wedge!

The images shows the ZPASS version.

Reduces artifacts!

Penumbra wedge rasterization



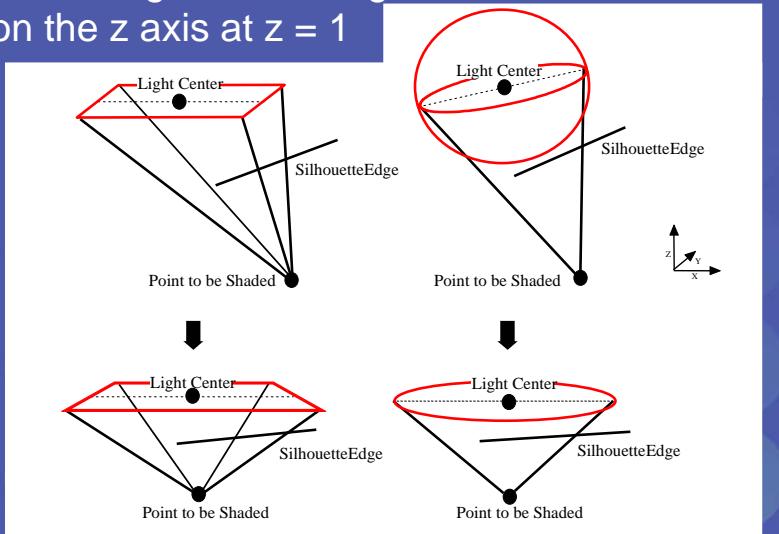
- Use culling step as described on prev slide
- Render frontfacing polygons of wedge, and execute fragment shader where stencil == +1
- → execute shader only in penumbrae
- What does the fragment shaders look like?

The culling helps performance quite a bit as well as avoiding artifacts that would otherwise come from testing whether a point is inside all wedge planes.

Fragment shaders: Transform the Silhouette Edge to Projection Space



- Projection space is defined as the point to be shaded at the origin and the light center located on the z axis at $z = 1$

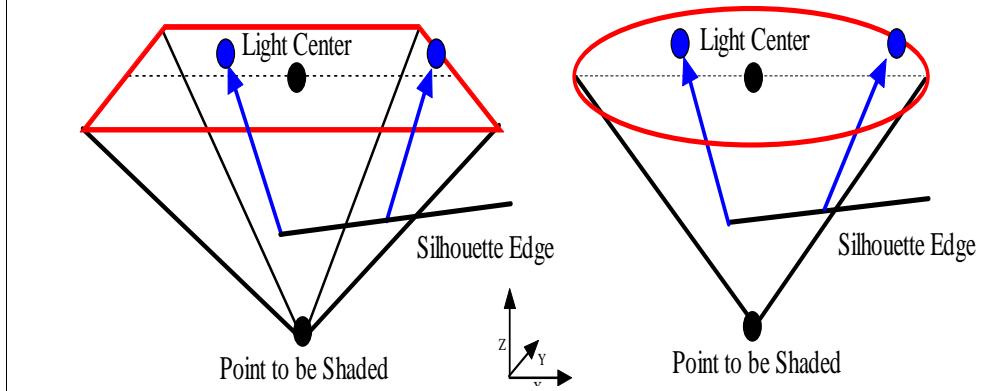


Projection is a SHEAR and a SCALE.

Clip edge and do perspective divide



- The square light shader clips edge to planes in homogenous space.
- The spherical light shader solves the equation of an intersection with a cone and line in homogenous space.
- The perspective divide is simply x/z and y/z .



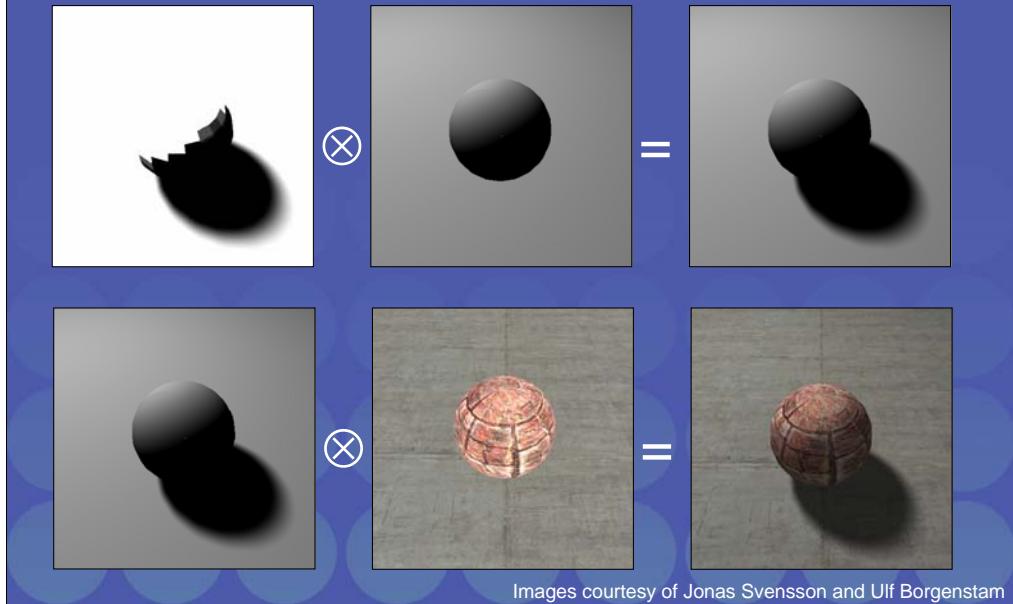
The shader programs become shorter when we performed clipping first, and then projection.

When edge has been clipped... compute coverage of edge



- For rectangular lights:
 - Use 4D coverage texture or
 - Compute it analytically + using 2D textures
(for constant lights)
 - Better accuracy!
- For spherical lights:
 - Compute analytically + using 2D textures
- We're done.

Combine shadow mask with lighting and texturing



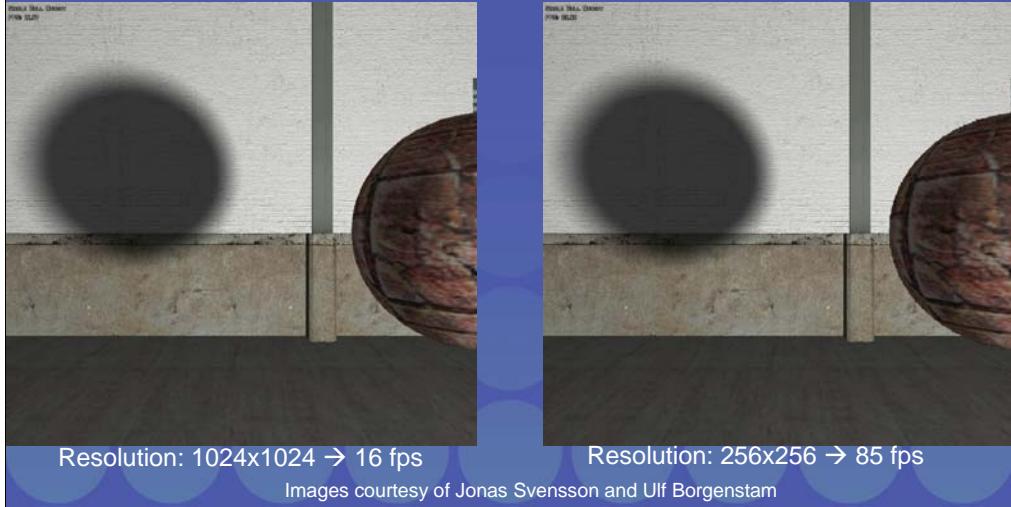
Images courtesy of Jonas Svensson and Ulf Borgenstam

Or more EFFICIENTLY: render an image with diffuse lighting plus texturing, and then modulate with soft shadow mask.

Load-balancing for constant frame rate rendering



- Simple idea: scale the size of the output V-buffer
 - Smaller resolution → faster rendering
- Usual tradeoff: speed vs image quality



Images courtesy of Jonas Svensson and Ulf Borgenstam

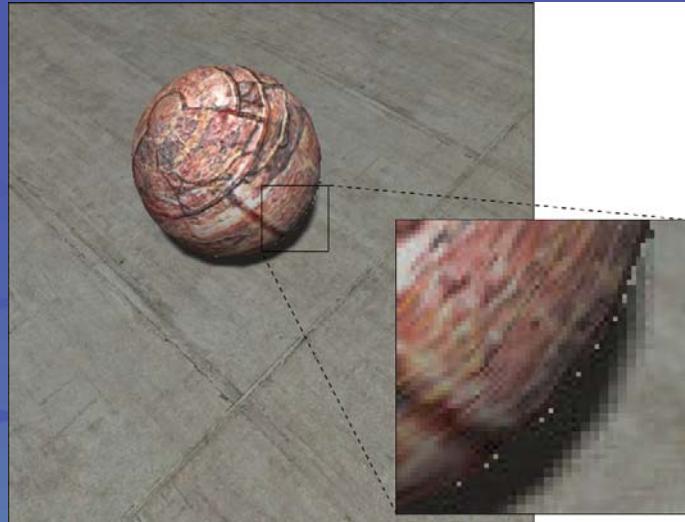
Always nice to be able to get higher frame rates. This is one way to do it.

Load balancing... cont'd

- Use a simple reactive algorithm
- Will get rendering errors?



Images courtesy of Jonas Svensson and Ulf Borgensham



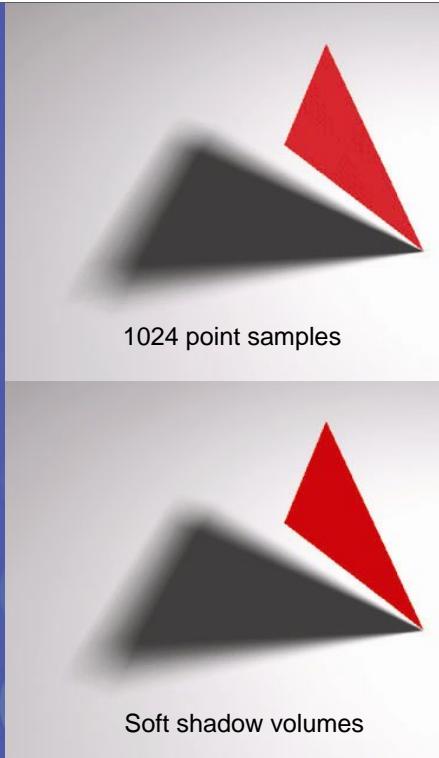
Is it worth it?

Will get flickering...

Can use bilinear filtering, but that costs...

Disadvantages of Soft Shadow Volumes

- Cases handled 100% correctly:
 - An arbitrary non self-intersecting, planar polygon
 - If the silhouette of the object is the same from all points on the area light source
- There are still a few approximations
- Shadow volumes do in general not scale well with scene complexity
- Recommended reading:
 - Jukka Arvo, Mika Hirvikorvi, Joonas Tyystjärvi, “A General Soft Shadow Map Algorithm using Graphics Hardware”, Eurographics 2004.

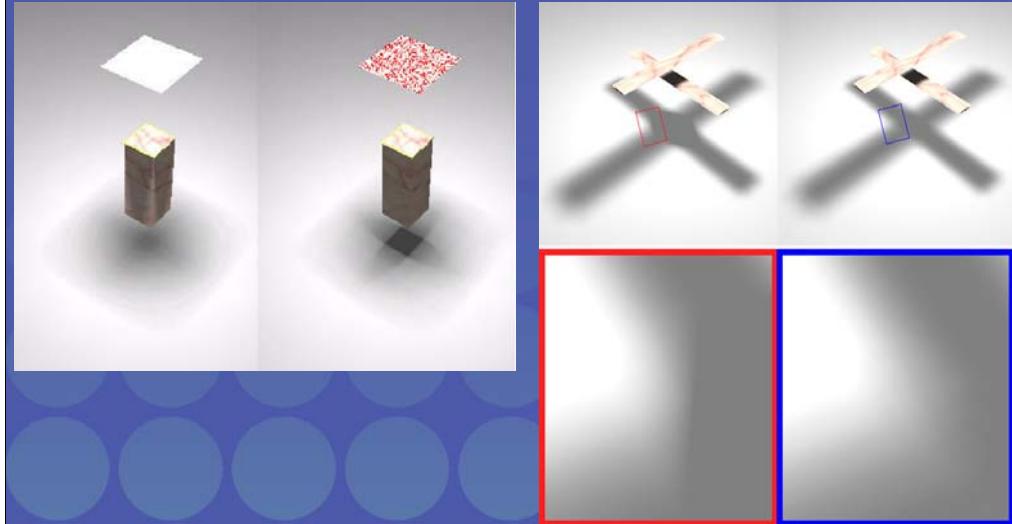


One further disadvantage is due to the single silhouette approximation: when the object is simple, the silhouette will change abruptly as will the shadow...

Example of object: cube

Artifacts:

- Single silhouette error
- Overlapping geometry is handled incorrectly



Single silhouette: can give a popping effect (e.g., for a cube with a moving light source → gives sudden and large changes of the silhouette).

**For better quality, use 2x2 area lights...
(and 2x2 does not cost 4 times as much!)**



Figure 8: One area light source, 2×2 area light sources, 1024 point light sources.



Figure 9: One area light source, 2×2 area light sources, 3×3 area light sources, 1024 point light sources.

In theory, we believe that 2×2 would cost about twice as much using ideal hardware...

Not sure what happens using real hardware.

Comparison Hard vs Soft

- Hard to beat the soft!
 - Soft **is** more realistic
 - Soft contains **less** high frequency content
 - Soft provides **better** spatial relationship cues
 - Soft **seldom** gives aliasing effects
- But...
 - Soft costs more
- More research to be done before the final answer is here!!

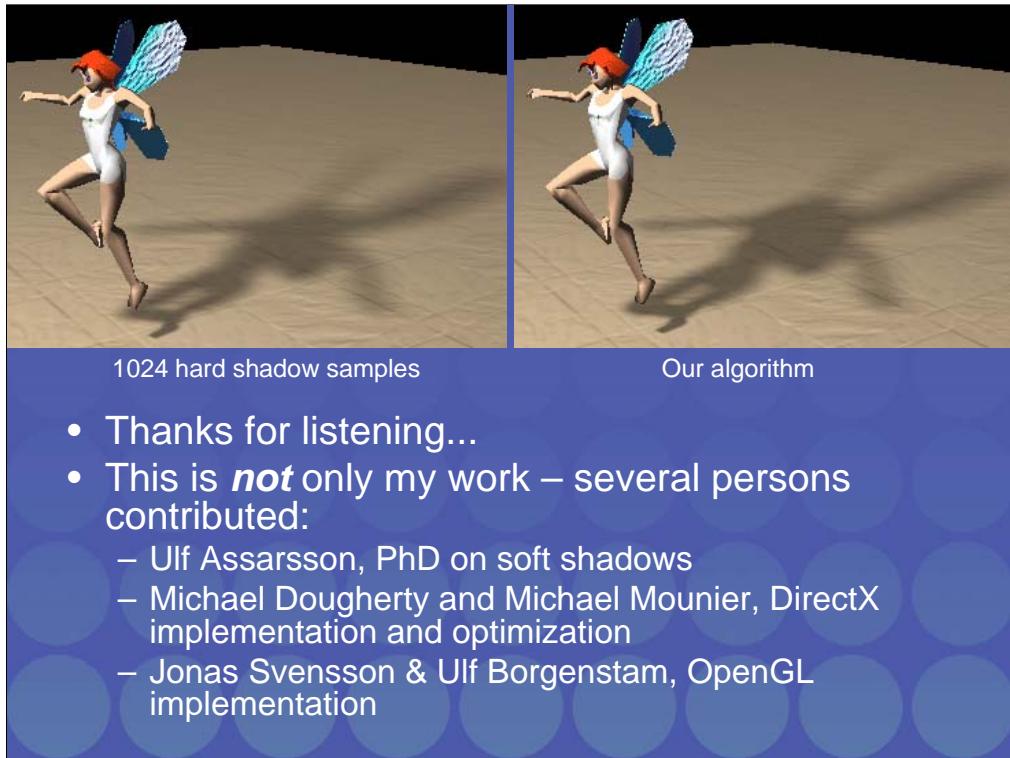
When the shadow border contain only high frequency content (which is the case for hard shadows), the shadow boundary can be misinterpreted for a geometrical feature.

The battle goes on between shadow mapping and shadow volumes, both for hard and soft shadows. I do not favor either of these – we'll see in a few years.



Time for real-time demo...

- Frame work
 - Written in OpenGL using GL_ARB_fragment_program etc.
 - There is also a smaller DirectX demo coded by Michael Dougherty & Michael Mounier, XBOX Advanced group
 - Open and free source code
 - <http://www.cs.lth.se/~tam/shadows/>
 - Need graphics hardware:
 - ATI Radeon 9700 and up...
 - Any NVIDIA GeForce FX
 - Coded by: Jonas Svensson and Ulf Borgenstam



- Thanks for listening...
- This is ***not*** only my work – several persons contributed:
 - Ulf Assarsson, PhD on soft shadows
 - Michael Dougherty and Michael Mounier, DirectX implementation and optimization
 - Jonas Svensson & Ulf Borgenstam, OpenGL implementation

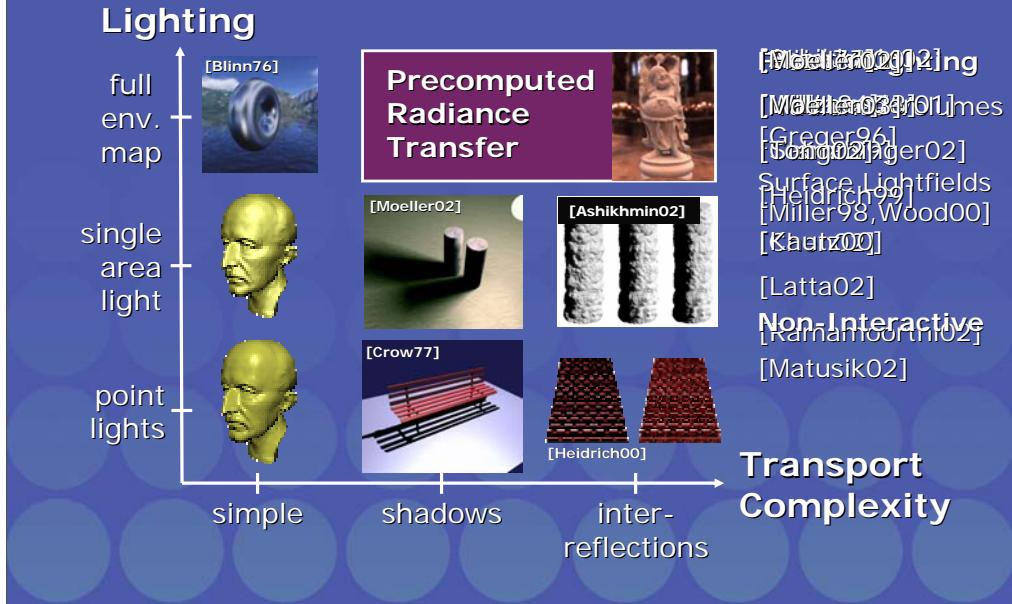


SIGGRAPH2004

Precomputed Radiance Transfer

Jan Kautz, Massachusetts Institute of Technology

Motivation – Transport vs. Light Complexity



Early interactive rendering techniques were limited to simple point lighting environments, and had almost no transport complexity – global effects like shadows were ignored.

Algorithms to generate interactive shadows exist – most notably crow's shadow volume technique and the shadow zbuffer technique by Williams – there is a nice extension to this work at this year's siggraph. However these techniques are restricted to point lights. Such algorithms can be combined with multipass rendering such as with an accumulation buffer to generate soft shadows, but the number of passes gets impractically large as the light sources get bigger.

Heidrich et al. modeled soft shadows and inter reflections with bump maps – but the technique is limited to point lights and performance isn't quite real-time. Polynomial texture maps can handle shadows and inter-reflections in real-time, but they too are limited to point lights.

Early work also extended lighting from point lights to general environment maps.

You have just seen two papers that generalize the early ideas to fairly general isotropic BRDFs. But environment map methods ignore transport complexity – they can't model shadows or inter-reflections.

As we have seen, shadow volumes can be extended to deal with single spherical light sources which may achieve real time rates on the next generation of hardware.

Ashikhmin et al. have a TOG paper that just came out that allows you to steer a small light source over a diffuse object.

Precomputed Radiance Transfer handles arbitrary, low-frequency lighting environments and provides transport complexity – including shadows, inter reflections and caustics.

Note that it is precisely soft shadows from low-frequency lighting that current interactive techniques have the greatest difficulty with.

Some earlier work had interesting transport complexity with frozen lighting environments. In particular the Irradiance volume paper allowed you to move a diffuse object through a precomputed lighting environment, dynamically lighting the object. However the object didn't shadow itself or the environment. Surface lightfields can model complex transport complexity and deal with interesting lighting environments, but the lighting environments are frozen. A recent example of that is Chen's paper this year.

Matusik et al. have a nice paper here that captures both a surface light field and a reflectance field – allowing you to relight objects in novel lighting environments. The reconstruction is not interactive however.

Motivation – What we want



- What we want:
 - Illuminate objects with environment maps
 - Change lighting on-the-fly
 - Include self-shadowing and interreflections
 - In real-time



No Shadows



Shadows

Background – Spherical Harmonics



- Spherical Harmonics $y_i(\vec{s})$:
 - Orthonormal basis over the sphere
 - Analogous to Fourier transform over 1D circle
- Projection: $f_i = \int_{\Omega} f(\vec{s}) y_i(\vec{s}) d\vec{s}$
- Reconstruction: $\tilde{f}(\vec{s}) = \sum_i f_i y_i(\vec{s})$
 - Note, this is an approximation!

The basis functions we used are the spherical harmonics – they are equivalent to the fourier basis on the plane, but mapped to the sphere.

They have several nice properties:

Since the basis is orthonormal projection is simple, evaluation is also very simple.

Background – Spherical Harmonics



- Important properties:
 - Rotational invariance \Rightarrow no aliasing artifacts
(no wobbling etc...)
- Integration:

$$\int_{\Omega} \tilde{a}(\vec{s}) \tilde{b}(\vec{s}) d\vec{s} = \sum_{i=1}^n a_i b_i$$

- Rotation: linear transform on coefficients
(matrix-vector multiplication, will not explain)

The most important property for our application is that they are rotationally invariant. This means that given some lighting environment on the sphere, you can project that lighting environment into SH, rotate the basis functions and integrate them against themselves (ie: rotating the projection and re-projecting) you get identical results to rotating the original lighting environment and projecting it into the SH basis.

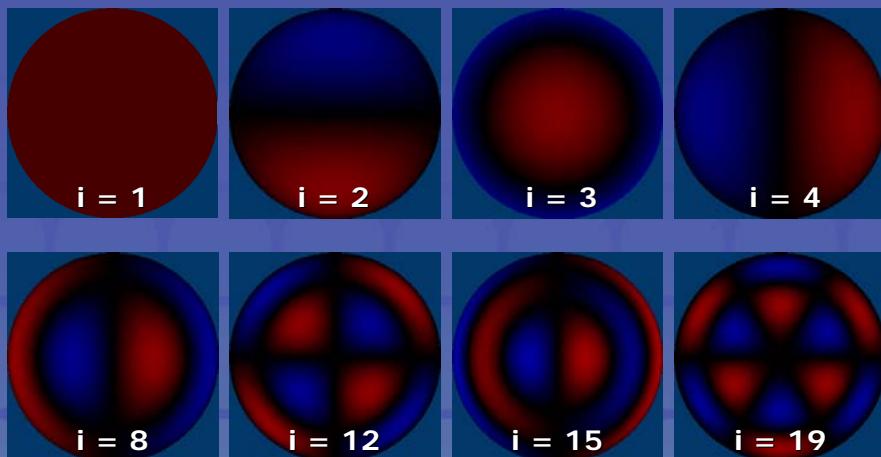
Integration of a product of two functions, which are represented in SH becomes a simple dot-product! Something we will exploit later on.

Rotation is simple (efficient evaluation formulae, just a linear operator on the SH coeffs). We won't explain it in this talk.

Background – Spherical Harmonics



- Basis functions (examples):



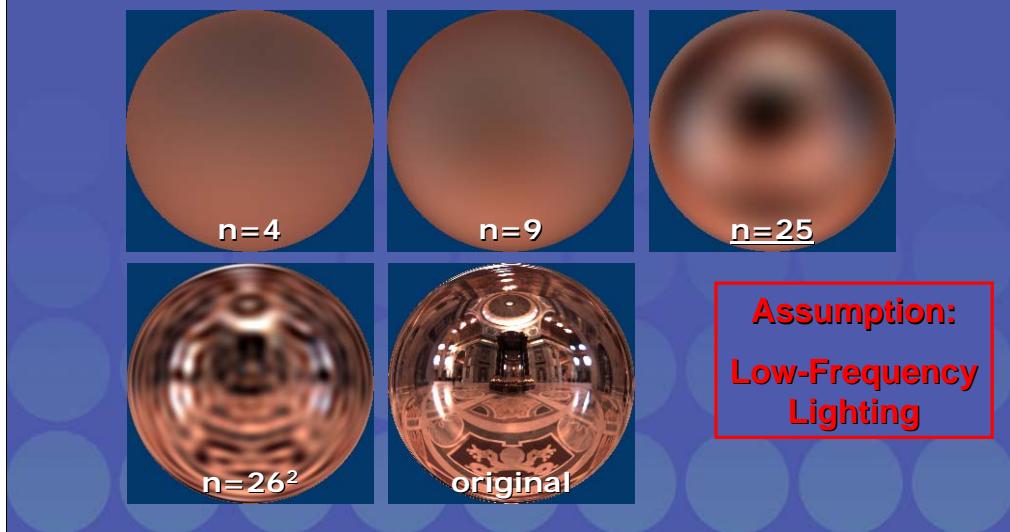
Examples of a few basis functions. They are defined over the sphere. We show them here as sphere maps.

Note, that the higher basis function have higher frequencies too (like Fourier).

Background – Spherical Harmonics



- Example: projection of environment



Here, we first convert a spherical function (environment map) into SH and then reconstruct the function (formula from before).

Again, the more basis functions are used, the higher frequencies can be represented...

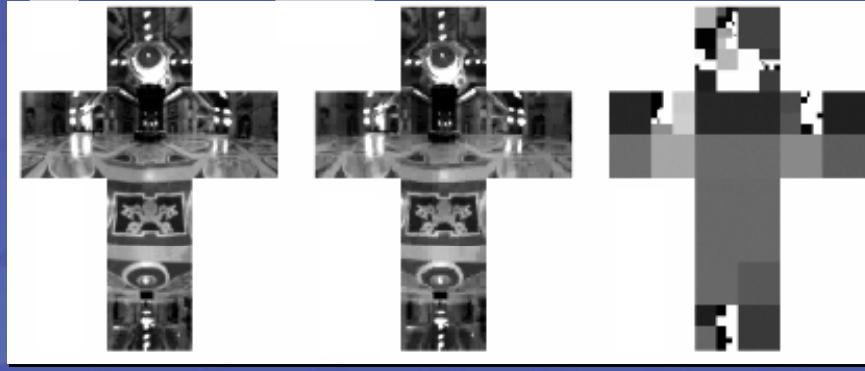
For all examples in this talk, we've used 25 coefficients (assuming low-frequency lighting).

Background – Haar Wavelets



- Example: projection of environment

Courtesy
Ren Ng



Reference

4096 coeffs.

100 coeffs.

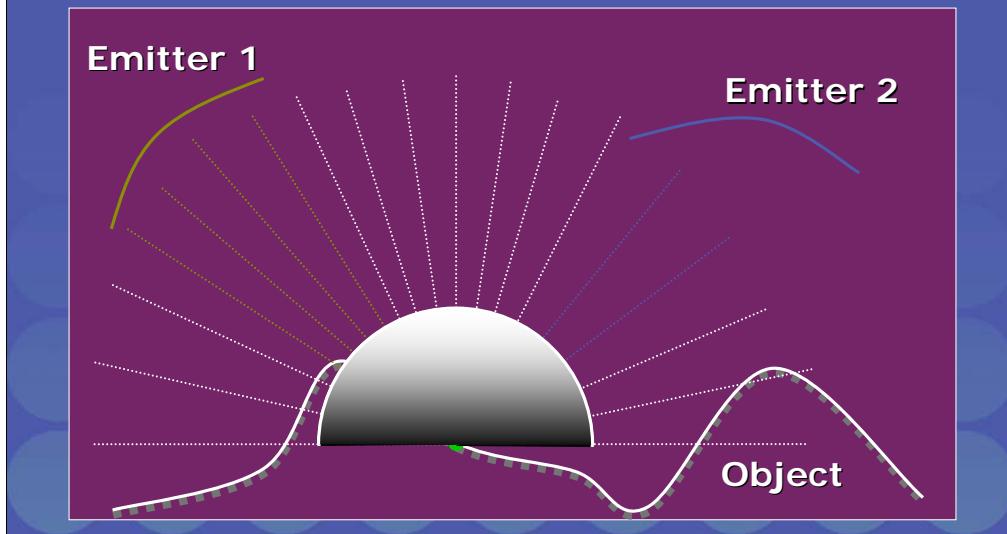
Alternatively, the Haar wavelet basis functions can be used (instead of SH). We will talk briefly about this at the end of the talk.

It should be noted, that wavelets are good at representing all-frequency detail (e.g. with 100 coeffs the bright windows are represented well, the not so important darker areas (floor) is represented with less accuracy).

Global Illumination (Rendering Equation)



- Integrate incident light * $V()$ * diffuse BRDF



To compute exit radiance from a point p , we need to integrate all incident lighting against the visibility function and the diffuse BRDF (dot-product between the normal and the light direction).

Rendering Equation



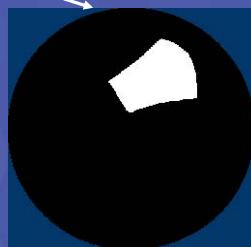
- Math:

$$L_p^{out}(\vec{v}) = \int L^{in}(\vec{s}) V_p(\vec{s}) \max(\vec{s} \cdot \vec{n}_p, 0) d\vec{s}$$

Reflected Light



Incident Light



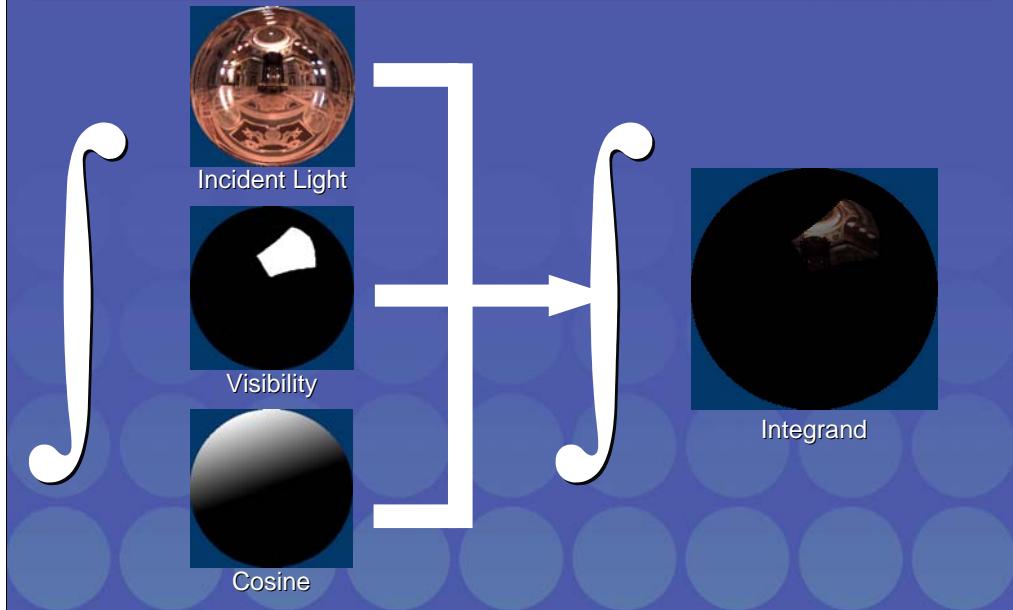
Visibility



Cosine

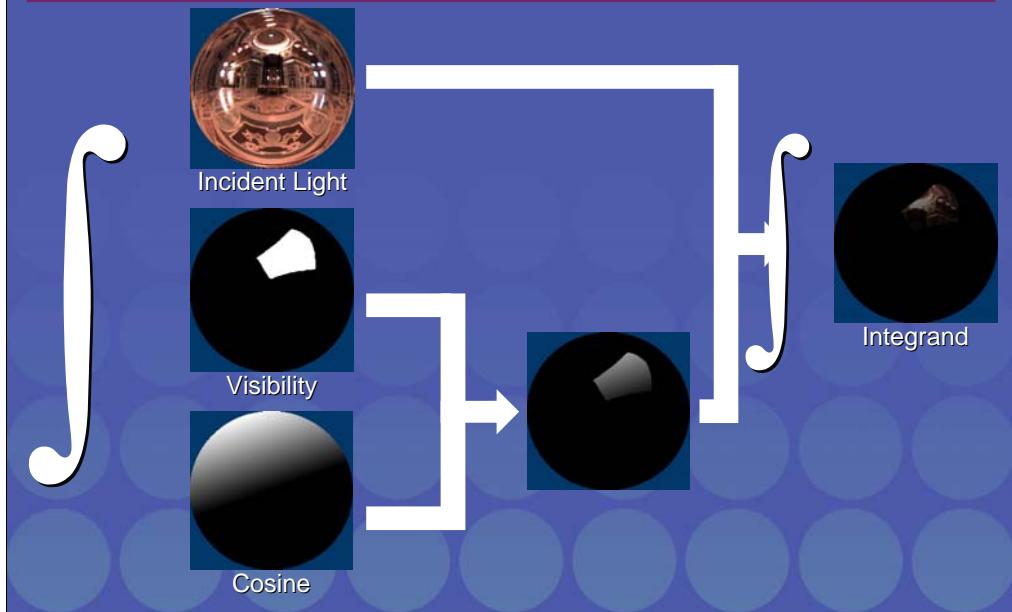
Same thing written down more accurately.

Rendering Equation – Visually



Visually, we integrate the product of three functions (light, visibility, and cosine).

Precomputed Radiance Transfer – Visually



The main trick we are going to use for precomputed radiance transfer (*PRT*) is to combine the visibility and the cosine into one function (*cosine-weighted visibility* or *transfer function*), which we integrate against the lighting.

Problems



- Problems remain:
 - How to encode the spherical functions?
 - How to quickly integrate over the sphere?

This is not useful per se. We still need to encode the two spherical functions (lighting, cosine-weighted visibility/transfer function). Furthermore, we need to perform the integration of the product of the two functions quickly.

Rendering Equation – Rewrite



- Math:

$$L_p^{out}(\vec{v}) = \int L^{in}(\vec{s})V(\vec{s})\max(\vec{s} \cdot \vec{n}_p, 0)d\vec{s}$$

- Rewrite with $T(\vec{s}) = V(\vec{s})\max(\vec{s} \cdot \vec{n}_p, 0)$ 
- This is the **transfer function**
 - Encodes:
 - Visibility
 - Shading
 - Implicitly: normal as well (no need to store it)

Using some more math again, we get the transfer function $T(s)$.

Note, that this function is defined over the full sphere. It also implicitly encodes the normal at the point p ! So, for rendering no explicit normal will be needed.

Rendering Equation – Rewrite



- Math:

$$L_p^{out}(\vec{v}) = \int L^{in}(\vec{s}) V(\vec{s}) \max(\vec{s} \cdot \vec{n}_p, 0) d\vec{s}$$

- Plug new $T(\vec{s})$ into Equation:

$$L_p^{out}(\vec{v}) = \int \underbrace{L^{in}(\vec{s})}_{\text{into SH}} \underbrace{T(\vec{s})}_{\text{into SH}} d\vec{s}$$



light function: L_t^{in}



transfer: T

⇒ project *lighting* and *transfer* into SH

Now, when we plug the new $T(s)$ into the rendering equation, we see that we have an integral of a product of two functions. We remember, that this special case boils down to a dot-product of coefficient vectors, when the two functions are represented in SH.

This is exactly, what we will do. We project the incident lighting and the transfer function into SH.

Evaluating the Integral



- The integral

$$L_p^{out}(\vec{v}) = \int L_i^{in}(\vec{s}) T(\vec{s}) d\vec{s}$$

becomes

$$L_p^{out}(\vec{v}) = \sum_i^n L_i^{in} T_i$$

"light vector"
"transfer vector"

A *simple* dot-product!!!!

(All examples use n=25 coefficients)

Then the expensive integral becomes a simple product between two coefficient vectors.

What does this mean?



- **Positive:**

- Shadow computation is *independent* of **number** or **size** of light sources!
- Soft shadows are *cheaper* than hard shadows
- Transfer vectors need to be computed (can be done offline)
- Lighting coefficients computed at run-time (3ms)

This has a number of implications:

Shadow computation/shading is independent of the number or the size of the light sources!
All the lighting is encoded in the lighting vector, which is independent of that.

Rendering this kind of shadows is extremely cheap. It is in fact cheaper than rendering hard shadows!

The transfer vectors can be computed off-line, thus incurring no performance penalty at run-time.

The lighting vector for the incident light can be computed at run-time (fast enough, takes a few milliseconds).

What does this mean?



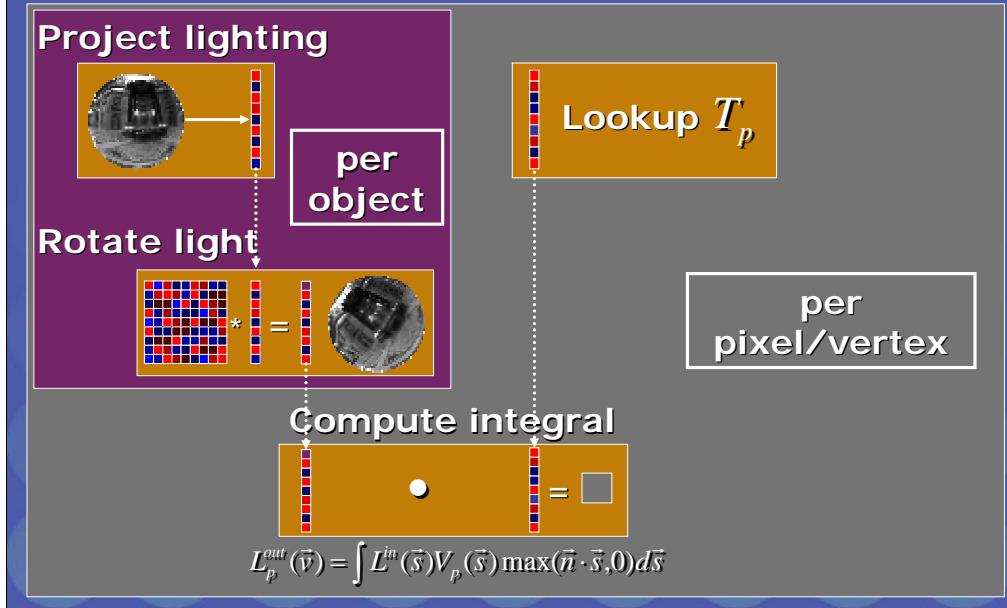
- **Negative:**

- Models are assumed to be ***static***
- Assumes all points on surface have ***same incident illumination*** (no shadows over half the object)

The precomputation of transfer coefficients means that the models have to be static!

Also, there is an implicit assumption, that all points on the surface receive the same incident illumination (environment map assumption). This implies that no half-shadow can be cast over the object (unless, it's part of the object preprocess).

Precomputed Radiance Transfer



This shows the rendering process.

We project the lighting into SH (integral against basis functions). If the object is rotated wrt to the lighting, we need to apply the inverse rotation to the lighting vector (using the SH rotation matrix).

At run-time, we need to lookup the transfer vector at every pixel (or vertex, depending on implementation). A (vertex/pixel)-shader then computes the dot-product between the coefficient vectors. The result of this computation is the exitant radiance at that point.

PRT Results



Unshadowed



Shadowed

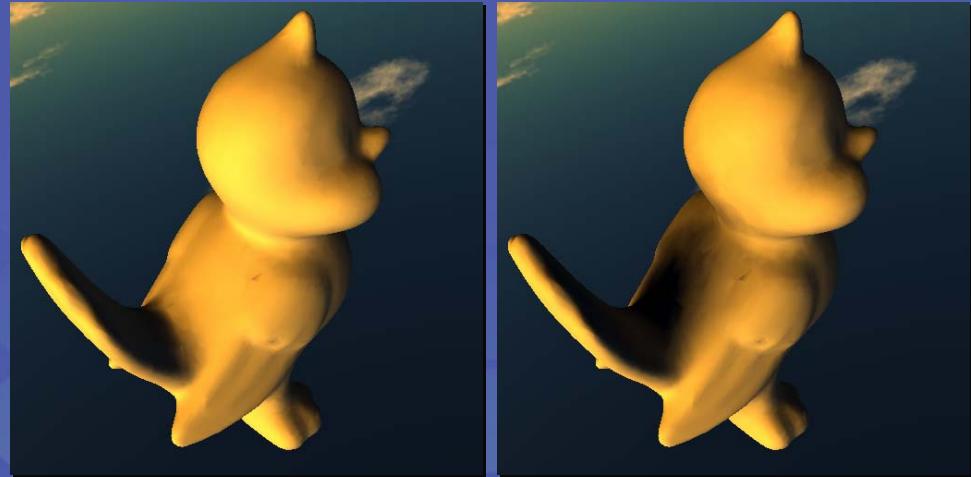
PRT Results



Unshadowed

Shadowed

PRT Results



Unshadowed

Shadowed

PRT Results



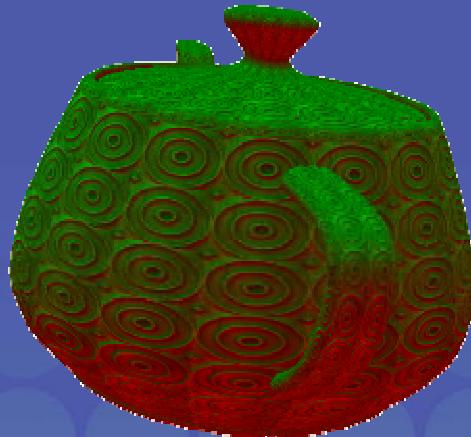
PRT Results



- Diffuse volume: 32x32x32 grid
- Runs 40fps on 2.2Ghz P4, ATI 8500
- Here: dynamic illumination

The same technique can be applied to diffuse volumes. Here we have a transfer vector at each voxel, instead of each pixel/vertex.

PRT Results



- Bump Map: 128x128 texel
- 50fps on 1.0Ghz Athlon, ATI 8500

Since, the normal is implicitly encoded in the transfer function, we can easily do bump mapping as well...

More Details



- How is precomputation done (for transfer vectors)?
- How is rendering done exactly?
- How can we reduce storage requirements?

Some detail has been lacking so far, which we will explain in the following.

Precomputation



- We need to ***project*** the transfer function $T_p(\vec{s})$ into SH (at every point p):

$$T_i = \int T_p(\vec{s}) y_i(\vec{s}) d\vec{s}$$

$$\text{with: } T_p(\vec{s}) = \frac{\rho}{\pi} V_p(\vec{s}) \max(\vec{n}_p \cdot \vec{s}, 0)$$

albedo visibility cosine

- Boils down to: Integral against basis functions

As we've seen before, both the lighting and the transfer function need to be projected into SH. Here we will talk about the projection of the transfer function.

As introduced in the Background-Section, projecting a function into SH boils down to integrating that function against the SH basis functions. This results in a vector of coefficients.

As a reminder: the transfer function is the visibility times the dot-product between the normal n and the sample direction s (multiplied by the albedo, which says how reflective the surface is). \

Precomputation



- Integral

$$T_i = \frac{\rho}{\pi} \int V_p(\vec{s}) \max(\vec{n}_p \cdot \vec{s}, 0) y_i(\vec{s}) d\vec{s}$$

evaluated numerically with ray-tracing:

$$T_i = \frac{4\pi}{N} \frac{\rho}{\pi} \sum_{j=0}^{N-1} V_p(\vec{s}_j) \max(\vec{n}_p \cdot \vec{s}_j, 0) y_i(\vec{s}_j)$$

- Directions \vec{s}_j need to be uniformly distributed (e.g. random)
- Visibility V_p is determined with ray-tracing

The main question is how to evaluate the integral. We will evaluate it numerically using Monte-Carlo integration. This basically means, that we generate a random (and uniform) set of directions s_j , which we use to sample the integrand. All the contributions are then summed up and weighted by $4*\pi/(\#samples)$.

The visibility $V_p()$ needs to be computed at every point. The easiest way to do this, is to use ray-tracing.

Aisde: uniform random directions can be generated the following way.

- 1) Generate random points in the 2D unit square (x,y)
- 2) These are mapped onto the sphere with:

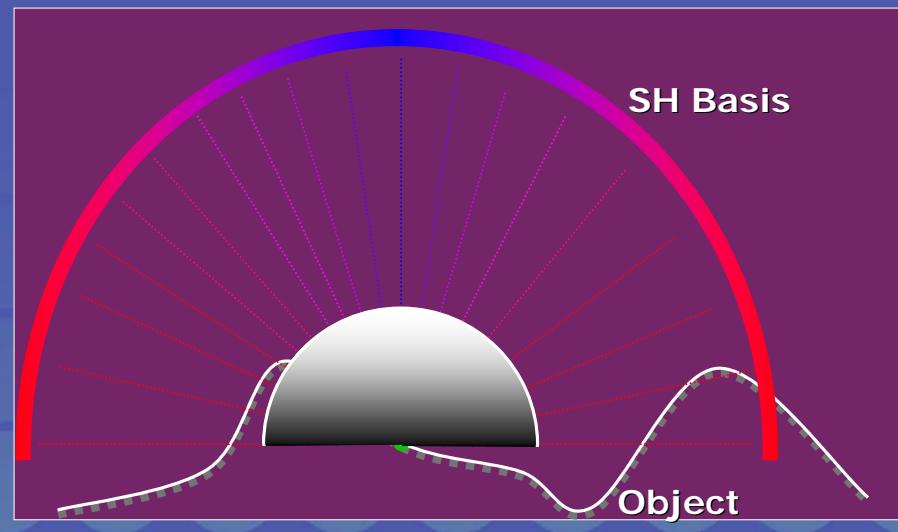
$$\text{theta} = 2 \arccos(\sqrt{1-x})$$

$$\text{phi} = 2y*\pi$$

Precomputation – Visually

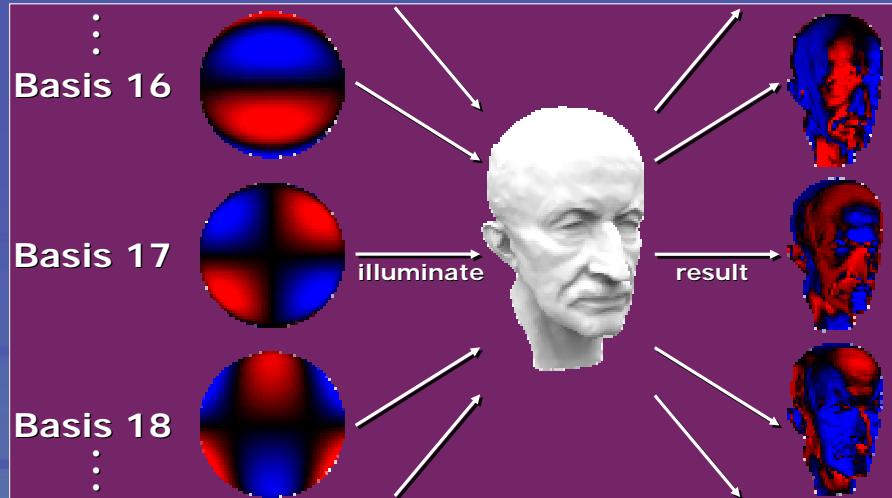


- Integrate $V_p(\vec{s}) \cdot \max(\vec{n}_p \cdot \vec{s}, 0) \cdot y_i(\vec{s})$



Visual explanation 1).

Precomputation – Visually



Visual explanation 2):

This slide illustrates the precomputation for direct lighting. Each image on the right is generated by placing the head model into a lighting environment that simply consists of the corresponding basis function (SH basis in this case illustrated on the left.) This just requires rendering software that can deal with negative lights.

The result is a spatially varying set of transfer coefficients shown on the right.

To reconstruct reflected radiance just compute a linear combination of the transfer coefficient images scaled by the corresponding coefficient for the lighting environment.

Precomputation – Code



```
// p: current vertex/pixel position
// normal: normal at current position
// sample[j]: sample direction #j (uniformly distributed)
// sample[j].dir: direction
// sample[j].SHcoeff[i]: SH coefficient for basis #i and dir #j

for(j=0; j<numberSamples; ++j) {
    double csn = dotProduct(sample[j].dir, normal);
    if(csn > 0.0f) {
        if(!selfShadow(p, sample[j].dir)) {      // are we self-shadowing?
            for(i=0; i<numberCoeff; ++i) {
                value = csn * sample[j].SHcoeff[i]; // multiply with SH coeff.
                result[i] += albedo * value;          //           and albedo
            }
        }
    }
}
const double factor = 4.0*PI / numberSamples; // ds (for uniform dirs)
for(i=0; i<numberCoeff; ++i)
    Tcoeff[i] = result[i] * factor;           // resulting transfer vec.
```

Pseudo-code for the precomputation.

The function `selfShadow(p, sample[j].dir)` traces a ray from position `p` in direction `sample[j].dir`. It returns true if there it hits the object, and false otherwise.

Precomputation – Bump Map



- Basically: no difference!
- We use per-pixel normal instead of interpolated normal...
- Self-shadowed bump maps are possible:
 - selfShadow() needs to account for that
- Again: run-time does *not* change

Bump mapping is really easy to incorporate (if transfer vectors are stored in a texture).

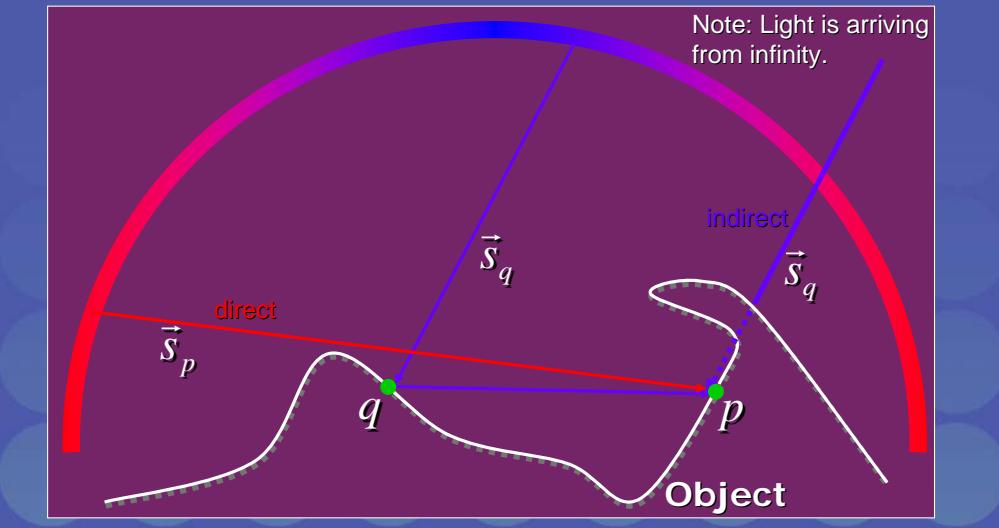
The precomputation algorithm from the previous slide remains the same. Only the normal needs to be looked up from a bump map in the precomputation phase!

If the bump map is too incorporate shadows from the bumps, then the selfShadow() function needs to be augmented to test for intersections with the bump map (e.g. convert to a heightfield for intersection tests).

Precomputation – Interreflections



- Light can interreflect from positions q onto p



Not only shadows can be included into PRT, but also interreflections.

Light arriving at a point q can be subsequently scattered onto a point p . I.e. light arriving from s_q can arrive at p , although there may be no direct path (along s_q) to p (as in this example).

Note, that light is arriving from infinity, so both shown direction s_q originate from the same point in infinity.

Precomputation – Interreflections



- Light can interreflect from positions q , where there is self-shadowing (no direct visibility of sphere!):

$$L_p^{out}(\vec{v}) = L^{DS} + \int L_q^{out}(\vec{s})(1 - V_p(\vec{s})) \max(\vec{s} \cdot \vec{n}_p, 0) d\vec{s}$$

↓
 direct illumination ↓
 light leaving from q
 towards p ↓
 inverse visibility ↓
 cosine

More formally, we do not only have direct illumination L^{DS} (Direct Shadowed), but also light arriving from directions s , where there is self-shadowing (i.e. $1 - V_p(s)$). The light arrives from positions q , which are the first hit along s .

Precomputation – Interreflections



- Precomputation of transfer vector has to be changed
- An additional bounce b is computed with

$$T_{p,i}^b = \frac{\rho_p}{\pi} \int T_{q,i}^{b-1} (1 - V_p(\vec{s})) \max(\vec{n}_p \cdot \vec{s}, 0) d\vec{s}$$

where $T_{p,i}^0$ is from the pure shadow pass

- Final transfer vector: $T_{p,i} = \sum_{b=0}^{B-1} T_{p,i}^b$

Run-time remains the same!

To account for interreflections, the precomputation has to be changed again.

Each additional bounce b generates a vector $T^b_{\{p,i\}}$, which is computed as shown on the slide. Each of these additional transfer vectors is for a certain bounce.

To get the final transfer vector, they have to be added. Again, the run-time remains the same!

Interreflections – Results



This set of images shows the buddha model lit in the same lighting environment, without shadows, with shadows and with shadows and inter reflections.

Rendering



- Reminder:

$$L_p^{out}(\vec{v}) = \sum_i^n L_i^{in} T_i$$

- Need lighting coefficient vector:

$$L_i = \int L^{in}(\vec{s}) y_i(\vec{s}) d\vec{s}$$

- Compute every frame (if lighting changes)
- Projection can e.g. be done using Monte-Carlo integration (see before)

Rendering is just the dot-product between the coefficient vectors of the light and the transfer.

The lighting coefficient vector is computed as the integral of the lighting against the basis functions (see slides about transfer coefficient computation).

Rendering



- Work that has to be done per-vertex is very simple:

```
// No color bleeding, i.e. transfer vector is valid for all 3 channels

for(j=0; j<numberVertices; ++j) {    // for each vertex
    for(i=0; i<numberCoeff; ++i) {
        vertex[j].red   += Tcoeff[i] * lightingR[i]; // multiply transfer
        vertex[j].green += Tcoeff[i] * lightingG[i]; // coefficients with
        vertex[j].blue  += Tcoeff[i] * lightingB[i]; // lighting coeffs.
    }
}
```

- Only shadows: independent of color channels \Rightarrow single transfer vector
- Interreflections: color bleeding \Rightarrow 3 vectors

Sofar, the transfer coefficient could be single-channel only (given that the 3-channel albedo is multiplied onto the result later on). If there are interreflections, color bleeding will happen and the albedo cannot be factored outside the precomputation. This makes 3-channel transfer vectors necessary, see next slide.

Rendering



- In case of interreflections (and color bleeding):

```
// Color bleeding, need 3 transfer vectors

for(j=0; j<numberVertices; ++j) {    // for each vertex
    for(i=0; i<numberCoeff; ++i) {
        vertex[j].red   += TcoeffR[i] * lightingR[i]; // multiply transfer
        vertex[j].green += TcoeffG[i] * lightingG[i]; // coefficients with
        vertex[j].blue  += TcoeffB[i] * lightingB[i]; // lighting coeffs.
    }
}
```

Extensions



- Reducing Storage Cost
- Animated Objects
- Other Materials
- Other Basis Functions (not SH)

In the following we talk about 4 extensions.

- 1) How to reduce the storage cost
- 2) How to incorporate animated objects
- 3) Other materials (other than diffuse)
- 4) Use of other basis functions

Reducing Storage Cost



- Original method:
 - Store transfer vector at each pixel/vertex
 - Each transfer vector has 25 coefficients (can be 8bit if scaled/biased appropriately)
 - Original paper suggest to just store 24 coeffs.
Pack into 6 textures
 - For a 256x256 texture, that's 1.5MB
- Reduce storage somehow

The original storage cost is fairly high (25 coefficients per texel).

Reducing Storage Cost



- Reduction:
 - Don't need high-resolution texture – shadows are very smooth (comparable to light maps)
 - Use standard texture compression
- Better Reduction:
 - Use Principle Component Analysis (PCA)
 - Use Vector Quantization (VQ)
 - Use combination (Clustered PCA)

Standard texture compression can work, but there are better techniques.

Reducing Storage Cost – PCA



- PCA:
 - Take all T_p and run PCA on it
 - Result: $T_p \approx T^0 + \sum_{k=1}^K w_p^k T^k$
 - Where: w_p^k are weights that change per pixel
 - And: T^k are basis vectors
 - Since original transfer vectors T_p can be very different, a high K is needed for good quality

PCA (Principle Component Analysis):

Plug all the vectors T_p (for all p) into PCA, and you will get above result/approximation.

The quality of the approximation depends on the number of basis vectors. If the variation in the T_p is high, a high number of basis vectors is needed.

The beauty of this is, that now we only need to store weights per pixel (hopefully far less than there are original coefficients). Unfortunately, this might not be the case, rendering pure PCA not very useful (or only limited to certain objects, where T_p is well-approximated with less than 25 basis vectors).

Reducing Storage Cost – VQ



- VQ:
 - Take all T_p and run VQ on it
 - Result:
 - Each T_p gets a unique T^k associated
 - Store k at each pixel, instead of actual vector
 - Number of T^k is much smaller than the number of T_p
- Problem:
 - Indices are stored in textures
 - Cannot do mip-mapping on them!
 - Visible quantization artifacts

VQ:

We plug all T_p into VQ and get a set of representative T^k back. Each T_p is uniquely associated with a T^k , but different T_p may map to the same T^k (quantization).

Now we only need to store the index at each pixel, instead of the actual vector.

Storage is hereby reduced, as the codebook (containing the T^k) is usually much smaller than the number of original vectors T^p .

Unfortunately, quantization artifacts may be visible and mip-mapping is not possible anymore.

Reducing Storage Cost – VQ+PCA



- VQ+PCA:
 - Take all T_p and run VQ on it
 - For each VQ transfer vector T^k
 - Find all T_p that map to it (cluster)
 - Do PCA only on those
 - Works much better than before, since the vectors in a cluster are very similar (because of VQ)
 - Works well, but complicates rendering a bit
 - But also makes it faster (few needed dot-products can be computed on CPU, GPU only does weighted sum for PCA)

The best compression technique combines both algorithms.

First run VQ. All the T_p that map to the same T^k are considered a cluster. Within that cluster PCA is used, which now works much better, because all T_p of a cluster are similar (due to VQ).

Rendering is slightly different now. The dot-products between T_p and lighting vector L is now: $T_p \cdot L = \text{SUM}(w_k T_k) \cdot L = \text{SUM}(w_k (T^k \cdot L)) = \text{SUM}(w_k R^k)$

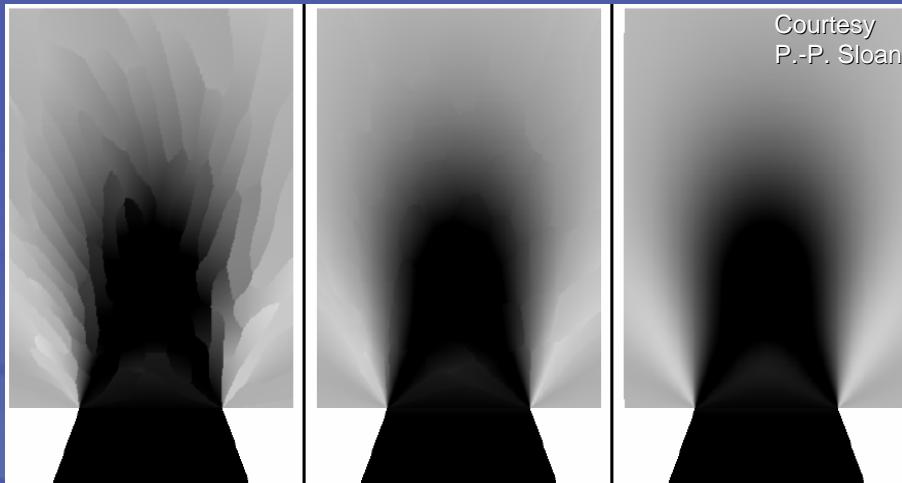
R^k can be computed on the CPU, since the T^k don't change per-pixel, but only the weights. Only the $\text{SUM}(w_k R^k)$ is then done on the GPU.

Attention has to be paid to the different areas of an object, that belong to different clusters (they have different basis vectors).

Compression – Comparison



Courtesy
P.-P. Sloan



SH Order 10, VQ+PCA
(2 PCA Vectors, 64 clus.)

SH Order 10, VQ+PCA
(3 PCA vectors, 64 clus.)

SH Order 10, VQ+PCA
(4 PCA vectors, 64 clus.)

Left: VQ producing 64 clusters. 2 PCA Vectors (mean + first) - Quantization artifacts are still visible (varies only linearly within a cluster)

Middle: VQ producing 64 clusters. 3 PCA Vectors (mean + first, second) - Artifacts almost gone (variation is now bilinear)

Right: VQ producing 64 clusters. 4 PCA Vectors (mean + first, second, third) - Visually as good as original (variation is trilinear within cluster)

Animated Objects



- So far, objects were assumed ***static***
- This is ok for e.g. architectural walkthroughs, but not for games
- Need to extend PRT to animated models
- Two kinds of models:
 - Key-framed
 - Fully dynamic

Assumption so far was: static models (because of precomputation).

Animated Objects



- Key-framed animations:
 - Compute PRT solution for (sub-set) of key-frames
 - Interpolate in-between (coefficients can be interpolated)
- Problems:
 - Very high storage costs
 - Solution: Do VQ+PCA on all precomputed frames
 - High computation cost (PRT for each frame)
 - Solution: None really

Key-framed objects can be easily used with PRT. Just precompute transfer coefficient for every key-frame and interpolate coefficients within key-frames (as the rest is interpolated as well).

Storage cost can be decrease by using the same compression technique as before.

The precomputation becomes a lot more expensive... (Nothing really to prevent this).

Animated Objects

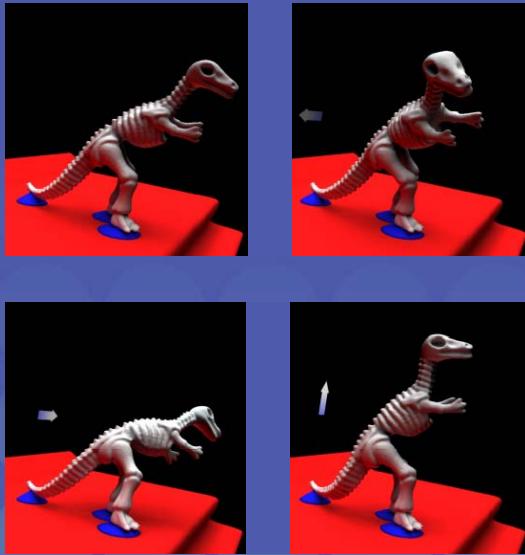


- Fully dynamic models:
 - No individual frames, for which PRT could be precomputed
 - Would need to do everything on-the-fly
 - **No solution yet!**

Fully dynamic models:

No solution in sight. Visibility changes radically (and in unknown ways). Computation of transfer vectors would need to be done on-the-fly.

Example: Animation with PRT



Courtesy
Doug James

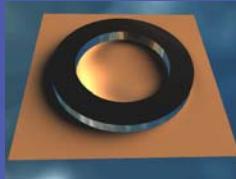
Example of a key-framed animation.

Other Effects / Materials



- The following effects can be incorporated easily:

- Caustics:



- Subsurface-Scattering:



Courtesy
P.-P. Sloan

- View-dependent effects require transfer matrices (see original paper)
 - E.g., glossy materials

Other effects that can be incorporated with an enhance pre-process: caustics and subsurface-scattering.

View-dependent effects require transfer matrices and not vectors (see original PRT paper).

PRT with glossy BRDFs

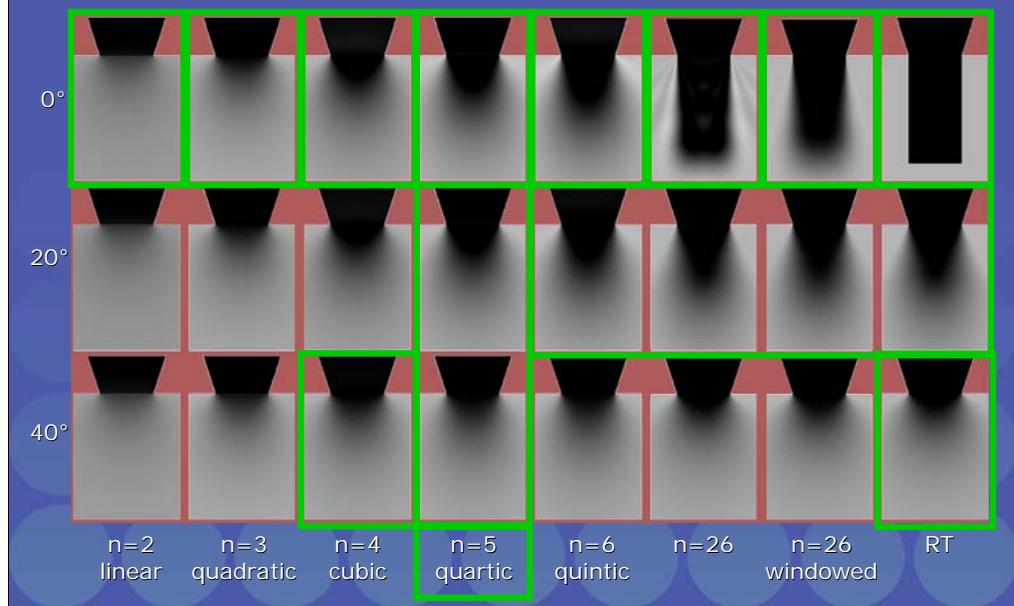


Phong



Measured Vinyl

PRT Quality – Is SH basis well-suited?



Quality of SH solution.

0 degree (point light) source, 20 degree light source, 40 degree light-source.

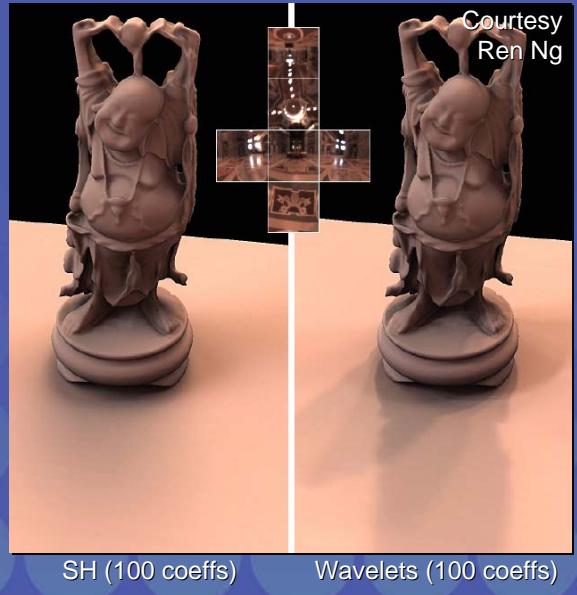
Light is blocked by a blocker casting a shadow onto the receiver plane. Different order of SH is shown (order² = number of basis functions). Very right: exact solution.

As stated before, lighting is assumed low-frequency, i.e. point light doesn't work well, but large area lights do!

Other Basis Functions



- SH produce very soft shadows
- Alternative: Haar Wavelets [Ng et al. 02]
- Quality improves quite a bit
- Idea remains the same:
 - Per-pixel dot-product to compute shading



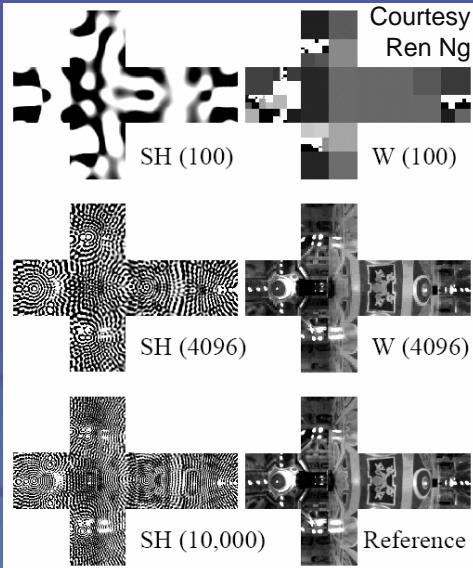
As we've noticed SH make shadows smooth (few coefficients are used, hence the low-frequency lighting assumption). Even if more are used (say 100), still low-frequency.

Alternative: Use Haar Wavelets. Quality improves quite a bit. Basic algorithm remains the same.

PRT with Haar Wavelets



- Main difference to SH:
 - Haar needs to precompute **all** lighting/transfer coefficients!
 - Decide depending on lighting, which ones to use! (see right)
 - Implies (compressed) storage of all transport coefficients ($64 \times 64 \times 6$)
 - Not well-suited to hardware rendering



As shown in the comparison on the right, with more coefficients, wavelets do much better represent the lighting than the SH (which show a lot of ringing artifacts).

There are a few differences when using Haar instead of SH:

- 1) All transfer coefficients need to be computed!
- 2) Because the actual N coefficients used, is decided at run-time based on the lighting's most important N coefficients ($N=100$ seems sufficient).
- 3) This requires all transfer coefficients to be stored as well (can be compressed well, like lossy wavelet compressed images).
- 4) Since the coefficients to be used change at run-time, this is not well-suited to a GPU implementation (but works fine on CPU)

Conclusions



Pros:

- Fast, arbitrary dynamic lighting
- PRT: includes shadows and interreflections

Cons:

- Works only well for low-frequency lighting
- Animated models are difficult to handle

- Thanks to:
 - ATI & NVIDIA for hardware donations
 - Paul Debevec for HDR environments

Thank you!