Robust Stenciled Shadow Volumes
Cass Everitt & Mark J. Kilgard
Graphics/Visualization Seminar to the Center for Computational Visualization
University of Texas
March 24, 2003
Stenciled Shadow Volumes in Practice

Notice the proper self-shadowing!
Completely Unrelated

Sarah Tiffany Kilgard
A shadow volume is simply the half-space defined by a light source and a shadowing object.
Shadow Volume Basics (2)

Simple rule: samples within a shadow volume are in shadow.

Surface outside shadow volume (illuminated)

Surface inside shadow volume (shadowed)

Partially shadowed object
Visualizing Shadow Volumes in 3D

- **Occluders and light source** cast out a shadow volume
- **Objects within the volume** should be shadowed

Scene with shadows from an NVIDIA logo casting a shadow volume

Visualization of the shadow volume
Shadow Volume Advantages

- Omni-directional approach
  - Not just spotlight frustums as with shadow maps
- Automatic self-shadowing
  - Everything can shadow everything, including self
  - Without *shadow acne* artifacts as with shadow maps
- Window-space shadow determination
  - Shadows accurate to a pixel
  - Or sub-pixel if multisampling is available
- Required stencil buffer broadly supported today
  - OpenGL support since version 1.0 (1991)
  - Direct3D support since DX6 (1998)
Shadow Volume Disadvantages

- Ideal light sources only
  - Limited to local point and directional lights
  - No area light sources for soft shadows
- Requires polygonal models with connectivity
  - Models must be closed (2-manifold)
  - Models must be free of non-planar polygons
- Silhouette computations are required
  - Can burden CPU
  - Particularly for dynamic scenes
- Inherently multi-pass algorithm
- Consumes lots of GPU fill rate
Counting Enter/Leaves With a Stencil Buffer (Zpass approach)

- Render scene to initialize depth buffer
  - Depth values indicate the closest visible fragments
- Use a stencil enter/leave counting approach
  - Draw shadow volume twice using face culling
    - 1st pass: render *front* faces and *increment* when depth test passes
    - 2nd pass: render *back* faces and *decrement* when depth test passes
  - Don’t update depth or color
- Afterward, pixel’s stencil is non-zero if pixel in shadow, and zero if illuminated
Visualizing the Stencil Buffer Counts

Shadowed scene

Stencil buffer contents

red = stencil value of 1
green = stencil value of 0

Stencil counts beyond 1 are possible for multiple or complex occluders.

GLUT shadowvol example credit: Tom McReynolds
Why Eye-to-Object Stencil Counting Approach Works

Light source → Shadowing object

Eye position

+1 +2 +2 +3

+1 +2 +2 +3

zero zero
Illuminated, Behind Shadow Volumes (Zpass)

Light source

Eye position

Shadowing object

Unshadowed object

Shadow Volume Count = +1+1+1-1-1-1 = 0
Shadowed, Nested in Shadow Volumes (Zpass)

Shadow Volume Count = +1+1+1-1 = 2
Illuminated, In Front of Shadow Volumes (Zpass)

Shadow Volume Count = 0 (no depth tests pass)
Nested Shadow Volumes
Stencil Counts Beyond One

Shadowed scene
Stencil buffer contents

green = stencil value of 0
red = stencil value of 1
darker reds = stencil value > 1
Animation of Stencil Buffer Updates for a Single Light’s Shadow Volumes

Every frame is 5 additional stencil shadow volume polygon updates. Note how various intermediate stencil values do not reflect the final state.
Problem Created by Near Clip Plane (Zpass)

Missed shadow volume intersection due to near clip plane clipping; leads to mistaken count
Alternative Approach: Zfail

- Render scene to initialize depth buffer
  - Depth values indicate the closest visible fragments
- Use a stencil enter/leave counting approach
  - Draw shadow volume twice using face culling
    - 1st pass: render back faces and increment when depth test fails
    - 2nd pass: render front faces and decrement when depth test fails
  - Don’t update depth or color
- Afterward, pixel’s stencil is non-zero if pixel in shadow, and zero if illuminated
Illuminated, Behind Shadow Volumes (Zfail)

Shadow Volume Count = 0 (zero depth tests fail)
Shadowed, Nested in Shadow Volumes (Zfail)

Shadow Volume Count = +1 + 1 = 2
Illuminated, In Front of Shadow Volumes (Zfail)

Light source

Shadowing object

Eye position

Shadowed object

Shadow Volume Count = -1-1-1+1+1+1 = 0
Problem Created by Far Clip Plane (\textit{Zfail})

Missed shadow volume intersection due to far clip plane clipping; leads to mistaken count

- Near clip plane
- Far clip plane

\begin{itemize}
  \item zero
  \item +1
  \item +2
  \item +3
  \item +2
\end{itemize}
Problem Solved by Eliminating Far Clip

Near clip plane
Avoiding Far Plane Clipping

Usual practice for perspective GL projection matrix
- Use *glFrustum* (or *gluPerspective*)
- Requires two values for near & far clip planes
  - Near plane’s distance from the eye
  - Far plane’s distance from the eye
- Assumes a *finite* far plane distance

Alternative projection matrix
- Still requires near plane’s distance from the eye
- But assume far plane is at *infinity*

What is the limit of the projection matrix when the far plane distance goes to infinity?
Standard \textit{glFrustum} Projection Matrix

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

Only third row depends on \text{Far} and \text{Near}
Limit of \( glFrustum \) Matrix as Far Plane is Moved to Infinity

\[
\lim_{\text{Far} \to \infty} P = P_{\text{inf}} = \begin{bmatrix}
\frac{2 \times \text{Near}}{\text{Right} - \text{Left}} & 0 & \frac{\text{Right} + \text{Left}}{\text{Right} - \text{Left}} & 0 \\
0 & \frac{2 \times \text{Near}}{\text{Top} - \text{Bottom}} & \\n0 & 0 & \frac{\text{Top} + \text{Bottom}}{\text{Top} - \text{Bottom}} & 0 \\
0 & 0 & -1 & -2 \times \text{Near}
\end{bmatrix}
\]

- First, second, and fourth rows are the same as in \( P \)
- But third row no longer depends on Far
- Effectively, Far equals \( \infty \)
What is the most distant possible vertex in front of the eye?

- Ok to use homogeneous coordinates
- OpenGL convention looks down the negative Z axis
- So most distant vertex is \((0,0,-D,0)\) where \(D>0\)

Transform \((0,0,-D,0)\) to window space

- Is such a vertex clipped by \(P_{\text{inf}}\)?
- No, it is not clipped, as explained on the next slide
Verifying $P_{\text{inf}}$ Will Not Clip Infinitely Far Away Vertices (2)

Transform eye-space $(0,0,-D,0)$ to clip-space

\[
\begin{bmatrix}
  x_c \\
  y_c \\
  -D \\
  -D 
\end{bmatrix}
= \begin{bmatrix}
  x_c \\
  y_c \\
  z_c \\
  w_c
\end{bmatrix}
= \begin{bmatrix}
  \frac{2 \times \text{Near}}{\text{Right} - \text{Left}} & 0 & \frac{\text{Right} + \text{Left}}{\text{Top} + \text{Bottom}} & 0 \\
  0 & \frac{2 \times \text{Near}}{\text{Top} - \text{Bottom}} & 0 & 0 \\
  0 & 0 & -1 & -2 \times \text{Near} \\
  0 & 0 & -1 & 0
\end{bmatrix}
\begin{bmatrix}
  0 \\
  0 \\
  -D \\
  0
\end{bmatrix}
\]

Then, assuming \textit{glDepthRange}(0,1), transform clip-space position to window-space position

\[z_w = 0.5 \times \frac{z_c}{w_c} + 0.5 = 0.5 \times \frac{-D}{-D} + 0.5 = 1\]

So $\infty$ in front of eye transforms to the maximum window-space Z value, but is still within the valid depth range (i.e., not clipped)
Robust Shadow Volumes *sans* Near (or Far) Plane Capping

- Use *Zfail* Stenciling Approach
  - Must render geometry to close shadow volume extrusion on the model and at infinity (explained later)
- Use the $P_{inf}$ Projection Matrix
  - No worries about far plane clipping
  - Losses some depth buffer precision (but not much)
- Draw the infinite vertices of the shadow volume using homogeneous coordinates ($w=0$)
Rendering Closed, but Infinite, Shadow Volumes

To be robust, the shadow volume geometry must be closed, even at infinity

Three sets of polygons close the shadow volume

1. Possible silhouette edges extruded to infinity away from the light
2. All of the occluder’s back-facing (w.r.t. the light) triangles projected away from the light to infinity
3. All of the occluder’s front-facing (w.r.t. the light) triangles

We assume the object vertices and light position are homogeneous coordinates, i.e. \((x,y,z,w)\)

Where \(w \geq 0\)
1st Set of Shadow Volume Polygons

Assuming
- A and B are vertices of an occluder model’s possible silhouette edge
- And L is the light position

For all A and B on silhouette edges of the occluder model, render the quad

\[
\begin{align*}
&\langle B_x, B_y, B_z, B_w \rangle \\
&\langle A_x, A_y, A_z, A_w \rangle \\
&\langle A_xL_w - L_xA_w, A_yL_w - L_yA_w, A_zL_w - L_zA_w, 0 \rangle \\
&\langle B_xL_w - L_xB_w, B_yL_w - L_yB_w, B_zL_w - L_zB_w, 0 \rangle
\end{align*}
\]

What is a possible silhouette edge?
- One polygon sharing an edge faces toward L
- Other faces away from L
Examples of Possible Silhouette Edges for Quake2 Models

An object viewed from the same basic direction that the light is shining on the object has an identifiable light-view silhouette.

An object’s light-view silhouette appears quite jumbled when viewed form a point-of-view that does not correspond well with the light’s point-of-view.
2\textsuperscript{nd} and 3\textsuperscript{rd} Set of Shadow Volume Polygons

\textbf{2\textsuperscript{nd} set of polygons}

- Assuming A, B, and C are each vertices of occluder model’s back-facing triangles w.r.t. light position L
- These vertices are effectively directions (w=0)

\[
\begin{align*}
\langle A_x L_w - L_x A_w, A_y L_w - L_y A_w, A_z L_w - L_z A_w, 0 \rangle \\
\langle B_x L_w - L_x B_w, B_y L_w - L_y B_w, B_z L_w - L_z B_w, 0 \rangle \\
\langle C_x L_w - L_x C_w, C_y L_w - L_y C_w, C_z L_w - L_z C_w, 0 \rangle
\end{align*}
\]

\textbf{3\textsuperscript{rd} set of polygons}

- Assuming A, B, and C are each vertices of occluder model’s front-facing triangles w.r.t. light position L

\[
\begin{align*}
\langle A_x, A_y, A_z, A_w \rangle \\
\langle B_x, B_y, B_z, B_w \rangle \\
\langle C_x, C_y, C_z, C_w \rangle
\end{align*}
\]
Complete Stenciled Shadow Volume Rendering Technique

See our paper “Practical and Robust Stenciled Shadow Volumes for Hardware-Accelerated Rendering”
  - In the accompanying course notes
  - And on-line at developer.nvidia.com

Paper has pseudo-code for rendering procedure
  - OpenGL state settings & rendering commands
  - Supports multiple per-vertex lights
  - Assumes application computes object-space determination of occluder model’s polygons orientation w.r.t. each light
Requirements for Our Stenciled Shadow Volume Technique (1)

1. Models must be composed of triangles only (avoiding non-planar polygons)
2. Models must be closed (2-manifold) and have a consistent winding order
   - Bergeron [’86] approach could be used to handle “open” models if necessary
3. Homogeneous object coordinates are permitted, assuming $w \geq 0$
   - If not, $(x, y, z, -1) = (-x, -y, -z, 1)$
4. Ideal light sources only
   - Directional or positional, assuming $w \geq 0$
5. Connectivity information for occluding models must be available
   - So silhouette edges w.r.t. light positions can be determined at shadow volume construction time

6. Projection matrix must be perspective
   - Not orthographic
   - NV_depth_clamp extension provides orthographic support (more later)

7. Render must guarantee “watertight” rasterization
   - No double hitting pixels at shared polygon edges
   - No missed pixels at shared polygon edges
Requirements for Our Stenciled Shadow Volume Technique (3)

8. Enough stencil bits
   - $N$ stencil bits where $2^N$ is greater than the maximum shadow depth count ever encountered
   - Scene dependent
   - 8-bits is usually quite adequate & what all recent stencil hardware provides
   - Wrapping stencil increment/decrement operations (i.e. OpenGL’s EXT_stencil_wrap) permit deeper shadow counts, modulo aliasing with zero
   - Realize that shadow depths > 255 imply too much fill rate for interactive applications
Requirements for Our Stenciled Shadow Volume Technique (4)

9. Rendering features provided by OpenGL 1.0 or DirectX 6 (or subsequent versions)
   - Transformation & clipping of homogenous positions
   - Front- and back-face culling
   - Masking color and depth buffer writes
   - Depth buffering (i.e. conventional Z-buffering)
   - Stencil-testing support

*In practice, these are quite reasonable requirements for nearly any polygonal-based 3D game or application*
Our Approach in Practice (1)

Scene with shadows. Yellow light is embedded in the green three-holed object. $P_{\text{inf}}$ is used for all the following scenes.

Same scene visualizing the shadow volumes.
Our Approach in Practice (2)

Details worth noting . . .

**Fine details:** Shadows of the A, N, and T letters on the knight’s armor and shield.

**Hard case:** The shadow volume from the front-facing hole would definitely intersect the near clip plane.
Our Approach in Practice (3)

Alternate view of same scene with shadows. Yellow lines indicate previous view’s view frustum boundary. Recall shadows are view-independent.

Shadow volumes from the alternate view.
Our Approach in Practice (4)

Clip-space view. Original view’s scene seen from clip space. The back plane is “at infinity” with very little effective depth precision near infinity.

Clip-space view of shadow volumes. Back-facing triangles w.r.t. light are seen projected onto far plane at infinity.
Another Example (1)

**Original eye’s view.** Again, yellow light is embedded in the green three-holed object. $P_{\text{inf}}$ is used for all the following scenes.

**Eye-space view of previous eye’s view.** Clipped to the previous eye’s $P_{\text{inf}}$ view frustum. Shows knight’s projection to infinity.
Another Example (2)

Clip-space view of previous eye’s view. Shows shadow volume closed at infinity and other shadow volume’s intersection with the near clip plane.

Original eye’s **far** clip plane

Original eye’s **near** clip plane
Stenciled Shadow Volumes & Multiple Lights

Three colored lights. Diffuse/specular bump mapped animated characters with shadows. 34 fps on GeForce4 Ti 4600; 80+ fps for one light.
Stenciled Shadow Volumes for Simulating Soft Shadows

Cluster of 12 dim lights approximating an area light source. Generates a soft shadow effect; careful about banding. 8 fps on GeForce4 Ti 4600.

The cluster of point lights.
Shadows in a Real Game Scene

Abducted game images courtesy Joe Riedel at Contraband Entertainment
Scene’s Visible Geometric Complexity

Wireframe shows geometric complexity of visible geometry

Primary light source location
Blow-up of Shadow Detail

Notice cable shadows on player model

Notice player’s own shadow on floor
Scene’s Shadow Volume
Geometric Complexity

Wireframe shows geometric complexity of shadow volume geometry

Shadow volume geometry projects away from the light source
Visible Geometry vs. Shadow Volume Geometry

Typically, shadow volumes generate considerably more pixel updates than visible geometry.
Other Example Scenes (1 of 2)

Dramatic chase scene with shadows

Abducted game images courtesy
Joe Riedel at Contraband Entertainment

Visible geometry

Shadow volume geometry
Other Example Scenes (2 of 2)

Scene with multiple light sources

*Abducted* game images courtesy Joe Riedel at Contraband Entertainment

Visible geometry

Shadow volume geometry
Situations When Shadow Volumes Are Too Expensive

Chain-link fence’s shadow appears on truck & ground with shadow maps.

Chain-link fence is shadow volume nightmare!

Fuel game image courtesy Nathan d’Obrenan at Firetoad Software
Shadow Volumes vs. Shadow Maps

- Shadow mapping via projective texturing
  - The other prominent hardware-accelerated shadow technique
  - Standard part of OpenGL 1.4
- Shadow mapping advantages
  - Requires no explicit knowledge of object geometry
  - No 2-manifold requirements, etc.
  - View independent
- Shadow mapping disadvantages
  - Sampling artifacts
  - Not omni-directional
Stenciled Shadow Volumes Optimizations

- Fill Rate Optimizations
  - Dynamic Zpass vs. Zfail determination
  - Bounds
- Culling Optimizations
  - Portal-based culling
  - Occluder and cap culling
- Silhouette Determination Optimizations
  - Efficient data structures for static occluder & dynamic light
  - Cache shadow volumes, update only when necessary
  - Simplified occluder geometry
- Shadow Volume Rendering Optimizations
  - Vertex programs for shadow volume rendering
  - Two-sided stencil testing
  - Directional lights
Shadow Volume History (1)

- Invented by Frank Crow ['77]
  - Software rendering scan-line approach
- Brotman and Badler ['84]
  - Software-based depth-buffered approach
  - Used lots of point lights to simulate soft shadows
- Pixel-Planes [Fuchs, et. al. ’85] hardware
  - First hardware approach
  - Point within a volume, rather than ray intersection
- Bergeron ['96] generalizations
  - Explains how to handle open models
  - And non-planar polygons
Fournier & Fussell ['88] theory
  Provides theory for shadow volume counting approach within a frame buffer
Akeley & Foran invent the stencil buffer
  IRIS GL functionality, later made part of OpenGL 1.0
  Patent filed in '92
Heidmann [IRIS Universe article, '91]
  IRIS GL stencil buffer-based approach
Deifenbach’s thesis ['96]
  Used stenciled volumes in multi-pass framework
Shadow Volume History (3)

- Dietrich slides [March ’99] at GDC
  - Proposes $Z_{fail}$ based stenciled shadow volumes
- Kilgard whitepaper [March ’99] at GDC
  - Invert approach for planar cut-outs
- Bilodeau slides [May ’99] at Creative seminar
  - Proposes way around near plane clipping problems
  - Reverses depth test function to reverse stencil volume ray intersection sense
- Carmack [unpublished, early 2000]
  - First detailed discussion of the equivalence of $Z_{pass}$ and $Z_{fail}$ stenciled shadow volume methods
Kilgard [2001] at GDC and CEDEC Japan

- **Proposes Zpass capping scheme**
  - Project back-facing (w.r.t. light) geometry to the near clip plane for capping
  - Establishes *near plane ledge* for crack-free near plane capping

- **Applies homogeneous coordinates (w=0) for rendering infinite shadow volume geometry**
- **Requires much CPU effort for capping**
- **Not totally robust because CPU and GPU computations will not match exactly, resulting in cracks**
Everitt & Kilgard [2002] Integrate Multiple Ideas into a robust solution
- Dietrich, Bilodeau, and Carmack’s Zfail approach
- Kilgard’s homogeneous coordinates (w=0) for rendering infinite shadow volume geometry
- Somewhat-obscure [Blinn ’93] infinite far plane projection matrix formulation
- DirectX 6’s wrapping stencil increment & decrement
  - OpenGL’s EXT_stencil_wrap extension

NVIDIA Hardware Enhancements
- Depth clamping [2001]: better depth precision
- Two-sided stencil testing [2002]: performance
- Depth/stencil-only hyper-rasterization: performance
- Depth bounds test [2003]: culling support
The Future

- Expect many games to have dynamic “everything shadows everything” shadows
- Expect hardware vendors to optimize hardware for this technique
  - The “Doom3 effect”
- Expect research to optimize shadow volume culling/rendering/etc.

Other applications
- Hardware-accelerated collision detection
- Computational Geometry
  - Problems related to the visibility problem
Supporting Slides

- Material not covered in presentation
- Mostly on optimization topics
Fill Rate Optimizations

- Dynamic Zpass vs. Zfail determination
- Bounds
  - CSG-style intersections
  - Window-space bounds
    - Scissoring
    - Depth bounds test (sort of a z-based scissor)
Zpass vs. Zfail (1)

- Zpass counting requires no caps
- Zpass is often more efficient (due to occlusion culling hw)
When to *not* use zpass

clipping throws the shadow count off for this whole region
Zpass vs Zfail (3)

- We can decide zpass vs zfail on a per-occluder basis
- do zfail when we must
- how do we decide?
Zpass vs Zfail (4)

How do we decide?

- one way: near plane – light pyramid

This diagram shows three planes in the “near plane - light pyramid”.

Any object that is completely outside one of the planes can be rendered with zpass.

Use object bounding volume for speed.
Zpass vs Zfail (5)

- closer near plane for more zpass

- bad for depth resolution
- NV_depth_clamp can help

objects completely outside plane “c” (zpass is ok)
Exploit Bounds

- Infinite shadow volumes can cover a lot of pixels
- With some knowledge of bounds, we can reduce fill
  - compute window space 3D axis-aligned bounding box of “possible shadow”
  - (x,y) bounds are scissor rect
  - What about z? New depth bounds test.
- Another strategy is to truncate the shadow volume using bounds
- Example bounds
  - lights
    - attenuation
    - environment (walls/floor)
  - shadow volumes
    - depth range of shadow
    - redundant shadows
For attenuated lights, we can use the scissor test to constrain the shadow volume.
- Works great for occluders beside the light.
- This is the window-space bounding box (x,y) of “possible shadow”.
Attenuated Light Bounds (2)

- Per-light scissor rect does not always very helpful
  - Cap the shadow volume at light bounds?
    - expensive/complicated
    - reasonable for static shadow volumes
  - Per-occluder scissor?
**Attenuated Light Bounds (3)**

- **Per-occluder scissor rect is nice**
  - simple
  - allows effective truncation of SV without altering SV geometry
  - good for dynamic SVs
Environmental Bounds

Some environments offer big opportunities for bounding optimizations.
Further savings from per-occluder info
intersecting scissor rects is easy

very small scissor rect!

fill savings

even more fill savings
Depth Bounds

Some depth values cannot be in shadow, so why bother to INCR and DECR their stencil index?

This is just the Z component of the window-space bounding box for the region of “possible shadow”!
Depth Bounds

- Discard fragment if **pixel** depth is out of bounds
  - not a clip plane!
  - sort of a scissor for z
- See **NV_depth_bounds_test** extension
- Simple API
  - `glDepthBoundsNV( GLclampd zmin, GLclampd zmax );`
  - `glEnable( GL_DEPTH_BOUNDS_TEST_NV );`
  - `glDisable( GL_DEPTH_BOUNDS_TEST_NV );`

- Depths in window space
  - same [0,1] range as `glDepthRange()`
Computing Window Space Bounds

Shadow volumes are infinite, but the depth bounds for possible shadows are constrained

SV constrained by frustum

depth bounds

zmin → zmax
Computing Window Space Bounds

SV constrained by light bounds

Light bounds (scissor and depth) can be used as conservative bounds. Usually too conservative.
Computing Window Space Bounds

SV constrained by environment

depth bounds

zmin  zmax

environment (floor)

scissor
Computing Window Space Bounds

- scissor rect and depth bounds really help constrain fill – use them aggressively
- use occluder bounding volume
  - conservative
  - inexpensive
Window Space Bounds

- Scissor + depth bounds
  - really just a window-space bounding box
- Beware points behind eye!
  - clip space w < 0
  - depth bounds: zmin == 0
  - scissor rect tricky

See “Jim Blinn’s Corner: Calculating Screen Coverage”
  - In Notation, Notation, Notation, and
  - also May ’96 issue of IEEE CG&A
Culling Optimizations

- **Shadow volume culling**
  - Culling the shadow volume extrusion
  - Specialized shadow volume culling for Zfail
    - Culling the near caps
    - Culling the far caps

- **Portal-based culling**
Shadow Volume Culling

Conventional view frustum culling
- Simple idea: If object is not within view frustum, don’t draw object
- Use conservative bounding geometry to efficiently determine when object is outside the view frustum
  - Often done hierarchically

Shadow volume culling
- Just because an object is not within the view frustum doesn’t mean it can’t cast a shadow that is visible within the view frustum!
Shadow Volume Culling Example

Light and occluder both outside the view frustum.

But occluder still casts shadow *into* the view frustum.

**Must** consider shadow volume of the occluder even though the occluder could be itself view frustum culled!
Shadow Volume Culling Easy Case

- If the light source is within the view frustum and the occluder is outside the view frustum, there’s no occluder’s shadow is within the view frustum. Culling (not rendering) the shadow volume is correct in this case.

Example:

Shadow volume never intersects the view frustum!
Generalized Shadow Volume Culling Rule

Form the convex hull of the light position and the view frustum

- For an infinite light, this could be an infinite convex hull
- If your far plane is at infinity due to using a Pinf projection matrix, the view frustum is infinite
  The convex hull including the light (even if local) is also infinite
- If an occluder is within this hull, you must draw the shadow volume for an occluder
Light and occluder both outside the view frustum.

But occluders still within convex hull formed by the light and view frustum

**Must** consider shadow volume of the purple occluder even though the purple occluder could be itself view frustum culled!
Light and occluder both outside the view frustum.

But purple occluders still **also** outside convex hull formed by the light and view frustum.

No need to render the purple occluders or their shadow volumes!

Green occluder must be considered however!
A note about “frustum”

- Effectively the view frustum is confined by
  - scissor rect
  - depth bounds
- Be sure to use this smaller frustum for culling
Infinite Cap Culling

When the infinite cap falls outside the scissor rect, don’t render it
- rendering a lot of trivially culled geometry -> pipeline bubbles
- can be significant optimization for per-occluder scissor
Portal-based Shadow Volume Optimizations

- Portal systems are convenient for
  - quickly identifying visible things
  - quickly eliminating hidden things
- For bounded lights, we can treat the light bounds as the “visible object” we’re testing for
  - If light bounds are visible, we need to process the light
  - If light bounds are invisible, we can safely skip the light
Silhouette Determination Optimizations

- Caching shadow volumes that are static
- Efficient data structures for static occluder & dynamic light
- Simplified occluder geometry
Cache Shadow Volumes, Update As Necessary

- Shadows are view-independent
- If occluder & light are static with respect to each other, shadow volumes can be re-used
  - Example: Headlights on a car have a static shadow volume w.r.t. the car itself as an occluder
- Precomputation of shadow volumes can include bounding shadow volumes to the light bounds and other optimizations
Efficient Data Structures for Static Occluder and Dynamic Light Interactions

- Brute force algorithm for possible silhouette detection inspects every edge to see if triangles joining form a possible silhouette edge.
  - Always works, but expensive for large models.
  - Perhaps the best practical approach for dynamic models.
- But static occluders could exploit precomputation.
    - Check out the “Fast Silhouette Extraction” section.
    - Data structure is useful for fast shadow volume possible silhouette edge determination.
Simplified Occluder Geometry

- More geometrically complex models can generate lots more possible silhouette edges
- More triangles in model means
  - More time spent searching for possible silhouette edges
- More possible silhouette edges means
  - More fill rate consumed by shadow volumes
- Consider substituting simplified model for shadow volume construction
  - Simplified substitute must “fit within” the original model
    - or you must give up self-shadowing
Shadow Volume Rendering Optimizations

- Compute Silhouette Loops on CPU
  - Only render silhouette for zpass
  - Vertex transform sharing
- Extrude Triangles instead of quads
  - For all directional lights
  - For zpass SVs of local lights
- Wrapping stencil to avoid overflows
- Two-sided stencil testing
- Vertex programs for shadow volume rendering
Avoid Transforming Shadow Volume Vertices Redundantly

- Use `GL_QUAD_STRIP` rather than `GL_QUADS` to render extruded shadow volume quads
  - Requires determining possible silhouette loop connectivity
  - Once you find an edge, look for connected possible silhouette edge loops until you form a loop
Shadow Volume Extrusion using Triangles or Triangle Fans

- Extrusion can be rendered with a GL_TRIANGLES_FAN
  - Directional light’s shadow volume always projects to a single point at infinity
  - Same is true for local lights – except the point is the “external” version of the light’s position (caveats)

Scene with directional light.

Clip-space view of shadow volume
Shadow Volume Extrusion using Triangle Fans

- For Local Lights, triangle fans can be used to render zpass (uncapped) shadow volumes.
  - If the light position is \((L_x, L_y, L_z, 1)\), the center of the triangle fan is the “external” point \((-L_x, -L_y, -L_z, -1)\).
  - This is simpler in terms of the geometry.
  - Can offer some fill efficiencies as well.

*What is an external triangle?*  
A triangle where one or two vertices has \(w<0\).

red vertex is **internal**  
red vertex is **external**
Hardware Enhancements: Wrapping Stencil Operations

- Conventional OpenGL 1.0 stencil operations
  - `GL_INCR` increments and clamps to $2^{N-1}$
  - `GL_DECR` decrements and clamps to zero
- DirectX 6 introduced “wrapping” stencil operations
  - Exposed by OpenGL’s `EXT_stencil_wrap` extension
    - `GL_INCR_WRAP_EXT` increments modulo $2^N$
    - `GL_DECR_WRAP_EXT` decrements modulo $2^N$
- Avoids saturation throwing off the shadow volume depth count
  - Still possible, though very rare, that $2^N$, $2 \times 2^N$, $3 \times 2^N$, etc. can alias to zero
Hardware Enhancements:
Two-sided Stencil Testing (1)

- Current stenciled shadow volumes required rendering shadow volume geometry twice
  - First, rasterizing **front**-facing geometry
  - Second, rasterizing **back**-facing geometry
- Two-sided stencil testing requires only one pass
  - Two sets of stencil state: front- and back-facing
  - Boolean enable for two-sided stencil testing
  - When enabled, back-facing stencil state is used for stencil testing back-facing polygons
  - Otherwise, front-facing stencil state is used
  - Rasterizes just as many fragments, but more efficient for CPU & GPU
Vertex Programs for Shadow Volumes

Three techniques to consider

- Fully automatic shadow volume extrusion
  - Everything off-loaded to GPU
  - Quite inefficient, not recommended
- Vertex normal-based extrusion
  - Prone to tessellation anomalies
- Semi-automatic shadow volume extrusion
  - CPU performs possible silhouette edge detection for each light
  - GPU projects out quads from single set of vertex data based on light position parameter
  - Doom3’s approach
Fully Automatic Shadow Volume Extrusion (1)

- 3 unique vertexes for each triangle of the original triangle mesh
- Insert degenerate “edge quads” at each & every edge of the original triangle mesh
  - This needs a LOT of extra vertices
    - Primary reason this technique is impractical
    - Too much transform & triangle setup overhead
- No way to do zpass cap optimizations
- No way to do triangle extrusion optimizations
Bloating the original triangle mesh

Original triangle mesh
6 vertexes
4 triangles

Bloated triangle mesh
12 vertexes
10 triangles

A lot of extra geometry!

Formula for geometry:

\[ v_{\text{bloat}} = 3 \times t_{\text{orig}} \]
\[ t_{\text{bloat}} = t_{\text{orig}} + 2 \times e_{\text{orig}} \]

Bloated geometry based only on number of triangles and edges of original geometry.
Vertex Normal-based Extrusion (1)

- Use per-vertex normals
  - If $\mathbf{N} \cdot \mathbf{L}$ is greater or equal to zero, leave vertex alone
  - If $\mathbf{N} \cdot \mathbf{L}$ is less than zero, project vertex away from light

Advantages
- Simple, requires only per-vertex computations
- Maps well to vertex program

Disadvantages
- Too simple – only per-vertex
- Best when normals approximate facet normals well
- Worst on faceted, jaggy, or low-polygon count geometry
- No way to do zpass cap optimizations
- No way to do triangle extrusion optimizations
Vertex Normal-based Extrusion (2)

- Example breakdown case

- Facet-based SV Extrusion (correct)
- Vertex Normal-based SV Extrusion (incorrect)
Vertex Normal-based Extrusion (3)

Example breakdown case

- Facet-based SV Extrusion (correct)
- Vertex Normal-based SV Extrusion (incorrect)
This shadow volume would collapse with a directional light source...

- **Facet-based SV Extrusion (correct)**
- **Vertex Normal-based SV Extrusion (incorrect)**
- **Incorrectly unshadowed region (incorrect)**
Vertex Normal-based Extrusion (5)

- Per-vertex normals are typically intended for lighting computations, not shadowing
  - Vertex normals convey curvature well, but not orientation
- Shadowing really should be based on facet orientations
  - That is, the normal for the triangle, not a vertex
- Hack compromise
  - Insert duplicate vertices at the same position, but with different normals
Semi-automatic Shadow Volume Extrusion (1)

- Possible silhouette edge detection done by CPU
  - For each light
- Want to only have a single set of vertices for each model
  - As opposed to a unique set of possible silhouette edge of vertices per light per model
- Separate index list per shadow volume
- Vertex program projects a single set of vertices appropriate for each possible silhouette edge for any particular light
Semi-automatic Shadow Volume Extrusion (2)

- Make two copies of every vertex, each with 4 components \((x,y,z,w)\)
  1. \(v = (x,y,z,1)\)
  2. \(v' = (x,y,z,0)\)
- For a possible silhouette edge with vertices \(p\) & \(q\)
  - Assume light position \(L\)
  - Render a quad using vertices \(p, q, p', q'\)
  - Vertex program computes
    \[
    \text{pos} = \text{pos} \times \text{pos}.w + (\text{pos} \times L.w - L \times \text{pos}.w) \times (1 - \text{pos}.w)
    \]
  - \(\text{pos}.w\) selects between projected and local vertex position
- Straightforward to handle caps too
Semi-automatic Shadow Volume Extrusion (3)

- Vertex array memory required = 8 floats / vertex
  - Independent of number of lights
  - Vertex program required is very short
  - MUCH less vertex array memory than fully automatic approach
Special Case: zpass

- Zpass requires no projection
  - 2 local points on silhouette
  - 1 point for extrusion
    - at infinity for infinite lights
    - external point for local lights
  - perfect for TRIANGLE_FAN
- Vertex memory = 4 floats / vertex
  - no projection per-vertex required!
- incompatible with automatic projection
Shadow Volume Polygon Rendering Order (1)

Naïve approach
- Simply render cap & projection shadow volume polygons in “memory order” of the edges and polygons in CPU memory

Disadvantages
- Potentially poor rasterization locality for stencil buffer updates
- Typically sub-par vertex re-use

Advantages
- Friendly to memory pre-fetching
- Obvious, easy to implement approach
Shadow Volume Polygon Rendering Order (2)

- GPU Optimized Approach
  - Possible silhouette edges form loops
    - Render the projected edge shadow volume quads in “loop order”
    - Once you find a possible silhouette edge, search for another edge sharing a vertex – continue until you get back to your original edge
  - When you must render finite and infinite caps
    - Greedily search for adjacent cap polygons
    - Continue until the cap polygons bump into possible silhouette edge loops – then look for more un-rendered capping polygons
Why use the GPU Optimized Approach?

- Tends to maximize vertex re-use
  - Avoids retransforming vertices multiple times due to poor locality of reference
- Maximizes stencil update efficiency
  - Adjacent polygons make better use of memory b/w
- Convenient for optimizations
  - zpass cap elimination
  - zfail infinite cap elimination
  - triangle instead of quad extrusion
- Easy to implement
  - Once you locate a possible silhouette edge, it’s easy to follow the loop
  - Easy to greedily search for adjacent finite & infinite cap polygons
Other Shadow Volume Issues

- Primarily an optimization talk, but...
- Some other important issues that need coverage
  - quality issue with smooth shading + shadow volumes
  - closed surface / 2-manifold issues and the art pipeline
  - portability to consoles
Smooth shading
- interpolate vertex quantities
- creates the illusion of denser geometry

coarse circle approximation

fine circle approximation

vertex normal
smooth normal (interpolated)
Shading “silhouette”

- Shading silhouette smoothly reaches 0
- animates nicely
Geometric silhouette

- Geometric silhouette abruptly cuts off intensity
- Polygons pop into shadow!

Incompatible lighting models...
How to resolve?

- Sometimes stencil should darken
- Sometimes shading should darken

Old Rule:
- Shade when unshadowed

New Rules:
- Shade when unshadowed
- Shade when trivially self-shadowed
Trivial Self-Shadowing

Pixel is trivially self shadowing when

- its polygon faces away from light
- it is in exactly one shadow volume (its own)
Implementation Alternatives

- No alternative is **fully** robust
  - live with popping
    - sometimes this is ok, but not usually
  - shade trivially self-shadowed polygons
    - back facing light and stencil == 1
    - some problematic shapes
  - don’t self shadow
    - inconsistent shadowing
  - inset shadow volume
    - problems knowing how much to inset...

You should evaluate which has right quality/performance tradeoff
2-Manifold issues

- Shadow volumes must be closed
  - convenient if occluder geometry closed
  - not always easy
    - many existing models are not closed
    - adds additional constraints on modelers
  - can just use front faces
    - “open” edges always on silhouette
Portability to Consoles

Stencil is supported directly on XBox and GameCube.

PS2 does not support stencil directly, but:
- use the color buffer with additive/subtractive blend
  - copy as texture
- can’t do zfail updates
  - must invert depth buffer
  - must invert depth of SV geometry
  - makes dynamically switching zpass/zfail impractical

Thanks to Pål-Kristian Engstad of Naughty Dog for PS2 info.