Shadows with rasterisation

Computer Graphics (DT3025)

Martin Magnusson
November 14, 2016

Doom 3, id Software

Assassin's Creed, Ubisoft

Last time

- Mapping 2D image data to 3D objects
- Interpolating over triangles (barycentric coordinates)
- Mapping applications
  - Texture maps
  - Bump maps
  - Displacement maps
  - Environment maps
- Aliasing, anti-aliasing

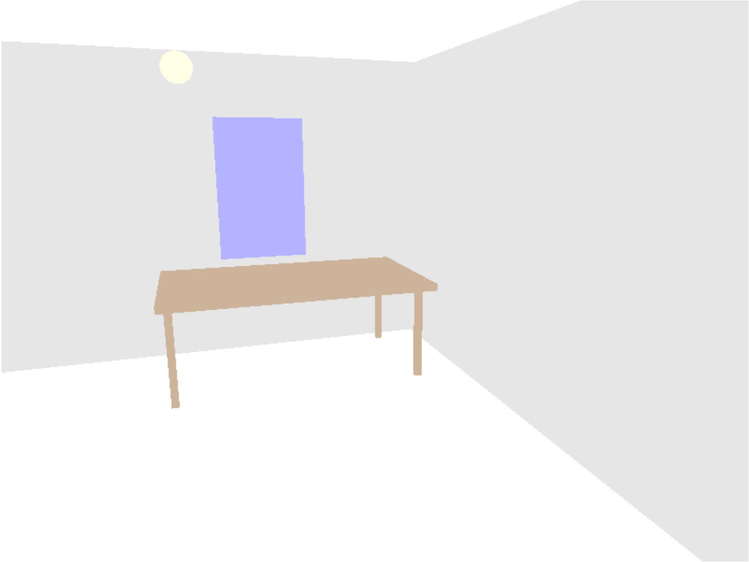
Today

- Tricks for casting shadows with rasterisation graphics
  - Shadow volumes
  - Shadow mapping

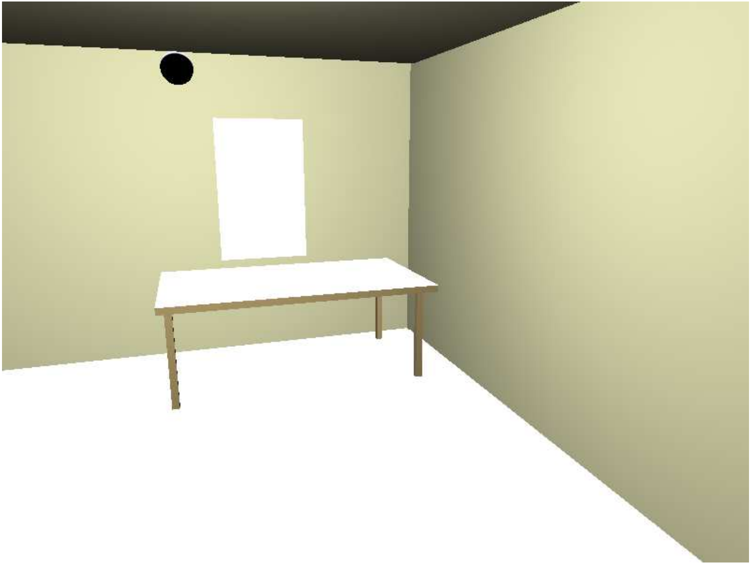
Reading material

- EuroGraphics'10 course notes:
  - Chapters 1, 2, 7 (and some of 3).
  - But mostly Chapter 2!

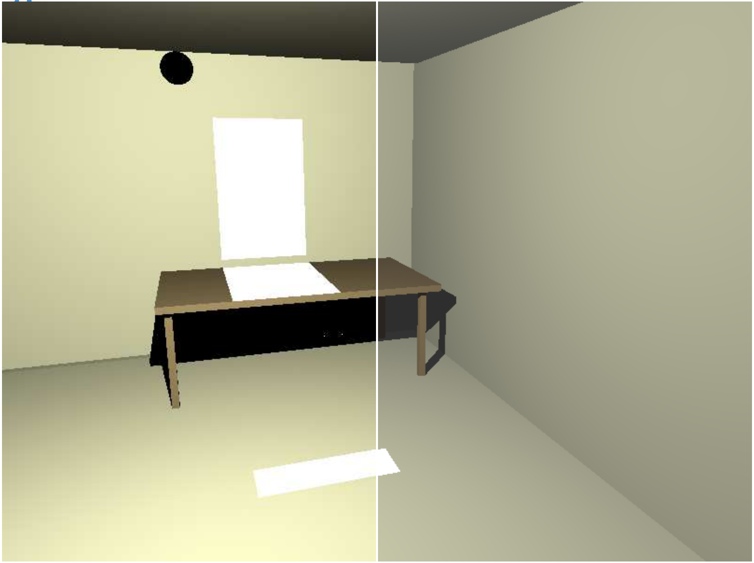
Objects/transformations/clipping



Basic direct lighting

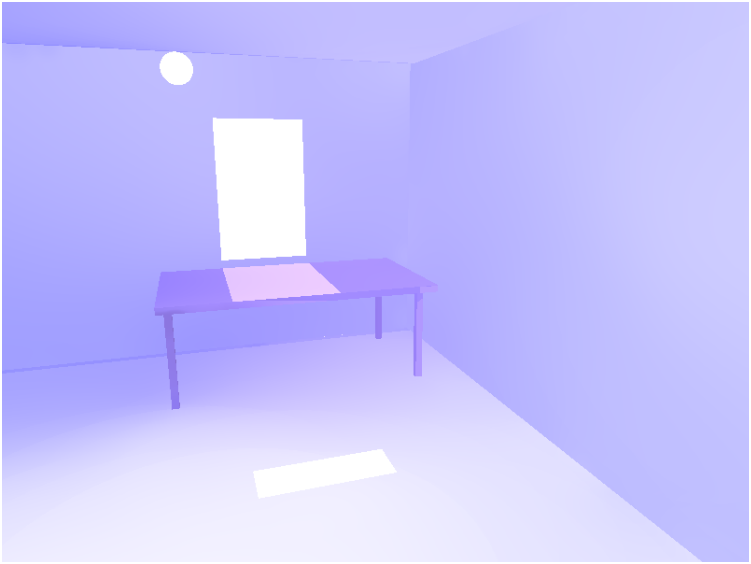


Casting shadows

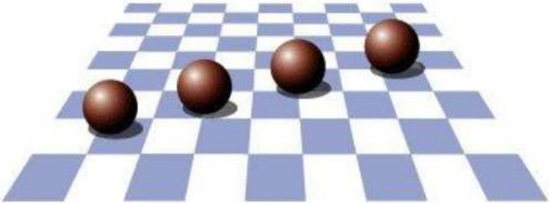


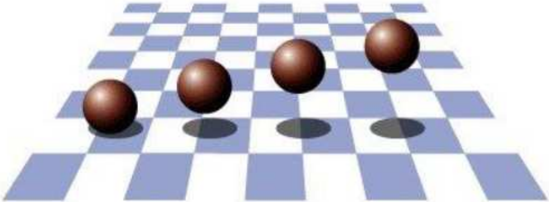
no ambient with ambient

Indirect illumination (and transparency, etc)



Light and shadow

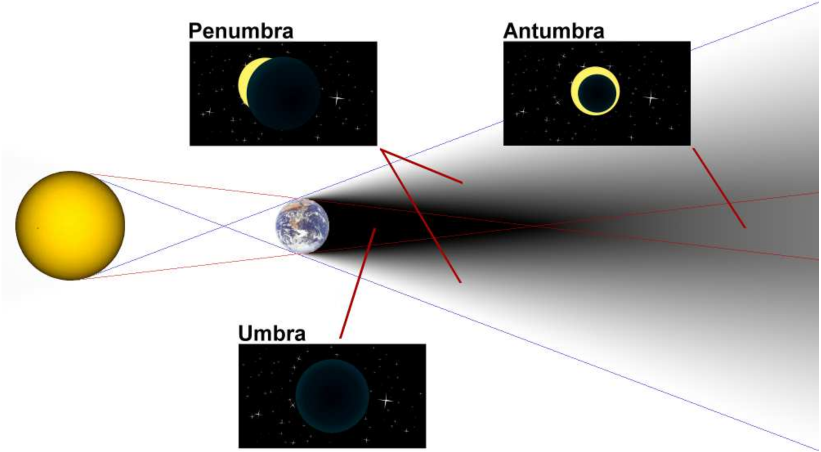




Shadows give important depth cues.

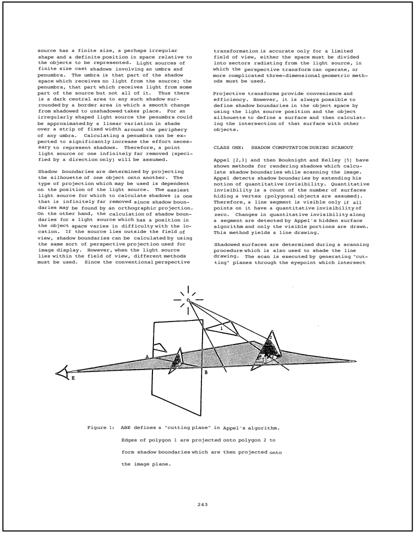
Shadow components

Occluder creates shadows
Receiver is shadowed



Qarnos, via Wikimedia Commons

Canonical shadows



source has a finite size, a perhaps irregular shape and a definite position in space relative to the objects to be represented. Light sources of finite size cast shadows involving an umbra and penumbra. The umbra is that part of the shadow space which receives no light from the source; the penumbra, that part which receives light from some part of the source but not all of it. Thus there is a dark central area to any such shadow surrounded by a border area in which a smooth change from shadowed to unshadowed takes place. For an irregularly shaped light source the penumbra could be approximated by a linear variation in shade over a strip of fixed width around the periphery of any umbra. Calculating a penumbra can be expected to significantly increase the effort necessary to represent shadows. Therefore, a point light source or one infinitely far removed (specified by a direction only) will be assumed.

Shadow boundaries are determined by projecting the silhouette of one object onto another. The type of projection which may be used is dependent on the position of the light source. The easiest light source for which to calculate shadows is one that is infinitely far removed since shadow boundaries may be found by an orthographic projection. On the other hand, the calculation of shadow boundaries for a light source which has a position in the object space varies in difficulty with the location. If the source lies outside the field of view, shadow boundaries can be calculated by using the same sort of perspective projection used for image display. However, when the light source lies within the field of view, different methods must be used. Since the conventional perspective transformation is accurate only for a limited field of view, either the space must be divided into sectors radiating from the light source, in which the perspective transform can operate, or more complicated three-dimensional geometric methods must be used.

Projective transforms provide convenience and efficiency. However, it is always possible to define shadow boundaries in the object space by using the light source position and the object silhouette to define a surface and then calculating the intersection of that surface with other objects.

CLASS ONE: SHADOW COMPUTATION DURING SCANOUT

Appel [2,3] and then Bouknight and Kelley [5] have shown methods for rendering shadows which calculate shadow boundaries while scanning the image. Appel detects shadow boundaries by extending his notion of quantitative invisibility. Quantitative invisibility is a count of the number of surfaces hiding a vertex (polygonal objects are assumed). Therefore, a line segment is visible only if all points on it have a quantitative invisibility of zero. Changes in quantitative invisibility along a segment are detected by Appel's hidden surface algorithm and only the visible portions are drawn. This method yields a line drawing.

Shadowed surfaces are determined during a scanning procedure which is also used to shade the line drawing. The scan is executed by generating "cutting" planes through the eyepoint which intersect

Figure 1: ABE defines a "cutting plane" in Appel's algorithm. Edges of polygon 1 are projected onto polygon 2 to form shadow boundaries which are then projected onto the image plane.

243

Shadow volumes, Crow (1977)

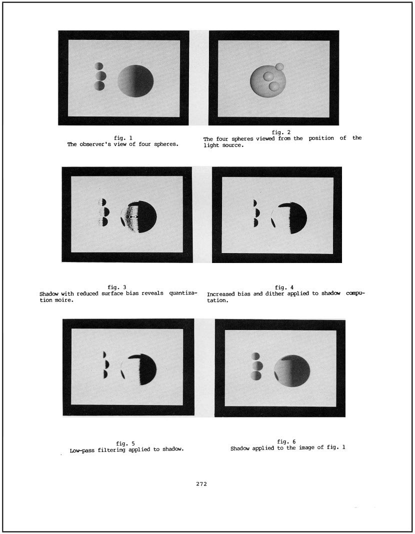


fig. 1
The observer's view of four spheres.

fig. 2
The four spheres viewed from the position of the light source.

fig. 3
Shadow with reduced surface bias reveals quantization moire.

fig. 4
Increased bias and dither applied to shadow computation.

fig. 5
Low-pass filtering applied to shadow.

fig. 6
Shadow applied to the image of fig. 1

272

Shadow maps, Williams (1978)

Microscopic shadows

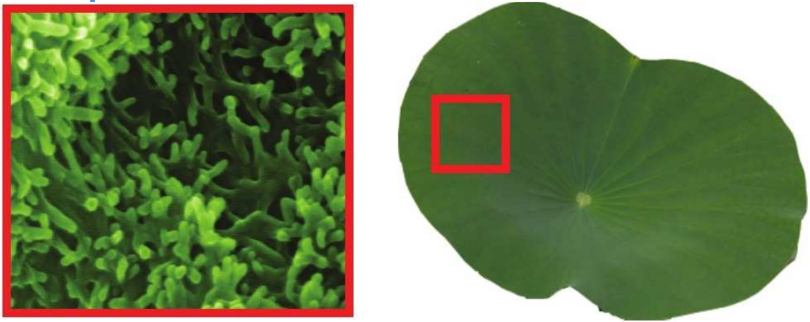
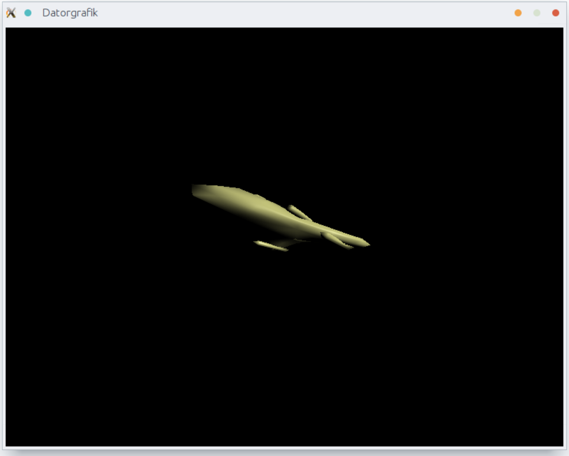


Figure 1.2 What we define as *shadow* depends upon the scale at which we look at objects. In the real world, the definition is thus very ambiguous; in a virtual world, described by a mathematically accurate framework, precise definitions are possible and meaningful. Left: Courtesy of Prof. U. Hartmann, Nanostructure Research and Nanotechnology, Saarland University. Right: Courtesy of [flickrPrince, 2007]

Eisemann et al. 2010

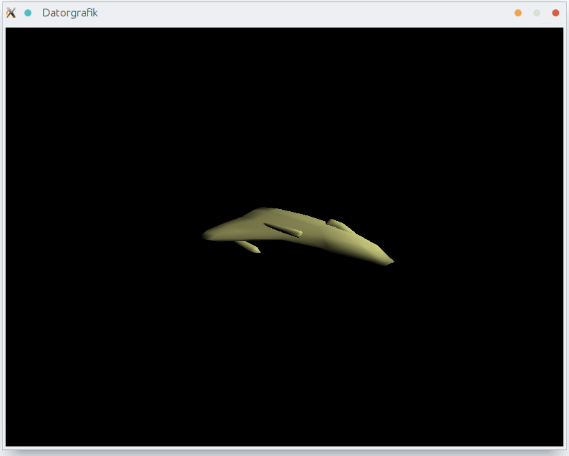
- Shadows vs material properties.
- Compare with Cook–Torrance reflectance.

Types of shadows



- Dark side at the back of an object
    - OK, we can already deal with this — but remember to check if dot product is positive!

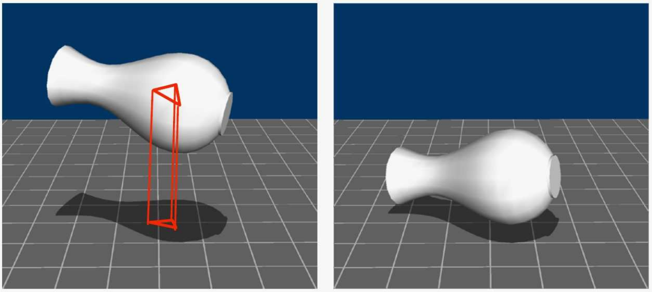
Types of shadows



- *Casting* shadows on an object
  - we'll deal with that today

Casting shadows on planar surfaces

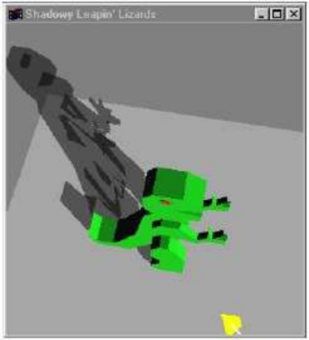
- Draw the object a second time:
  - coloured in as RGBA $= (0, 0, 0, 0.5)$ (for example),
  - and transformed with an extra matrix that projects vertices to ground plane.
  - What does such a projection matrix look like?
  - Any limitations?

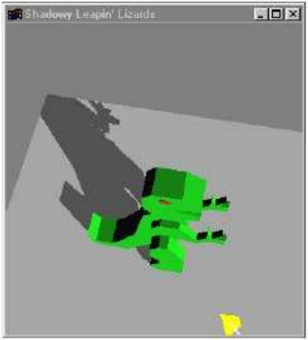


Jaakko Lehtinen

Limitations of planar shadows

- Much more difficult with curved surfaces.
- No self-shadowing.
- Shadow polygons falling outside receiver.

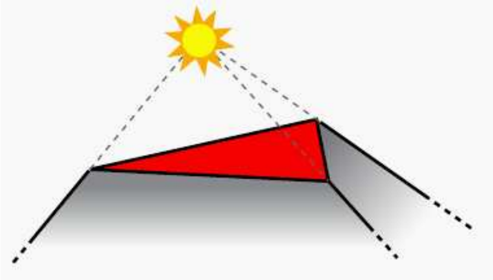




Kilgard 1999

Shadow volumes

- Essentially, we create a new 3D shape representing the *shadow volume* of an object.
- Shadow volume of triangle: capped triangular pyramid with point light at apex.



Eisemann et al. 2010

Shadow volume test

Conceptual pseudo code (“is the point inside the shadow volume of any light and polygon?”)

```
bool in_shadow( point P )
{
  for T in all triangles
  {
    for L in all lights
    {
       if (inside_shadow_volume( P, L, T ))
         return true;
    }
  }
  return false;
}
```

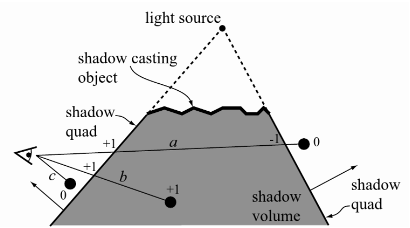
Shadow volume test

- Similar to the in-triangle test from lecture 1.
- Check if each fragment is inside all the half-spaces of the shadow-colume boundaries.
- Cost of naive implementation: $O($ #polygons $\times$ #lights $)$

A slightly better shadow volume test

- Is **p** in shadow?

1. Shoot ray from view point to **p**. Set counter $c = 0$.
2. Increment c each time ray enters a shadow volume.
3. Decrement c when ray exits shadow volume.
4. $c = 0$: **p** is lit.
5. $c \neq 0$: **p** is in shadow.



Eisemann et al. 2010

Smarter shadow volumes

1. We don't need shadow volume for *each primitive*.
   - $\Longrightarrow$ Compute shadow volumes only for *silhouette edges*.

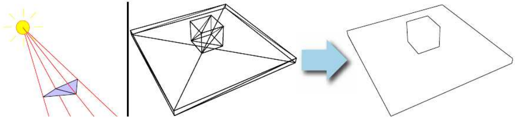


Figure 2.5 Left: Interior edge makes two quads which cancel out. Right: Finding the silhouette edges gets rid of many useless shadow volume quads.

Eisemann et al. 2010

2. Use the *stencil buffer* for more efficient intersection lookups.

The stencil buffer

- An additional buffer (same dimensions as framebuffer and depth buffer.)
- Typically used to mask out some pixels.

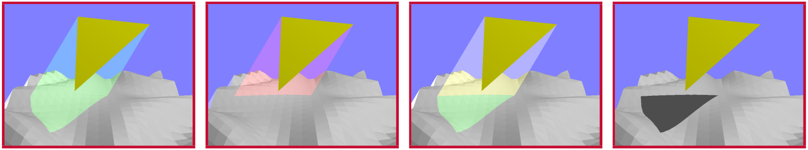
```
// Stencil test: write to depth/frame buffers only if test passes.
// This case: pass stencil test when stencilValue == 1.
glStencilFunc(GL_EQUAL, 1, 0xFF);
```

- Has operation such as increment or decrement based on result of stencil and depth tests.

```
glStencilOp(sfail,dpfail,dppass);
```

Stencil shadow volume algorithm

1. Draw scene with only ambient light.
2. Turn off frame and depth buffer.
3. Generate shadow volumes from triangle edges.
4. “Draw” *forward-facing* volumes, incrementing stencil buffer $+1$.
5. “Draw” *backward-facing* volumes, decreasing stencil buffer -1.
   Where stencil $\neq 0$ we now have a shadow.
6. Turn on frame and depth buffer.
7. Draw scene again, diffuse/specular light where stencil $= 0$.



Mathias Broxvall

Note that these images show diffuse+specular also in intermidiate stages.

Example: Doom 3

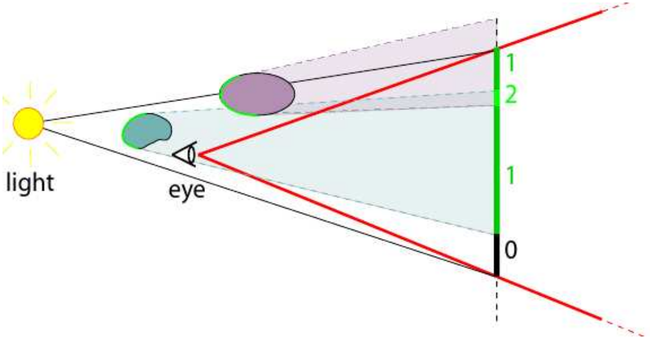
Doom 3, id Software

z-fail

- What if the camera is also in shadow?
- Count *c* will be wrong.
- Solution 1: initiate *c* to "# shadow vols that camera is in".
  - Need extra geometry tests on CPU.
- Solution 2: invert the depth test (known as "z-fail")
  - Count shadow-volume fragments *behind* **p**.
    - I e, "shoot ray from **p** *away from* view point."
  - Rationale: point at infinity is always in the light.
  - "z-fail" because the counted shadow-volume fragments fail the depth test (they are not visible).
  - More robust, but slower (must update stencil buffer also for many occluded fragments)

ZP+

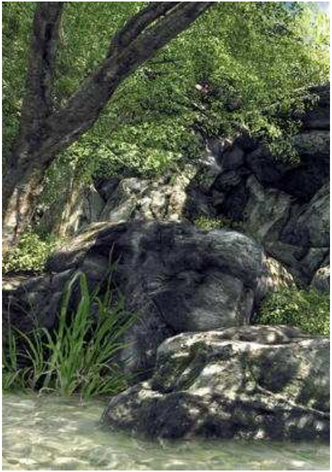
- Quick way to initate the stencil buffer (so that we can apply z-pass).
- Render scene from the light onto the camera's near plane.
- Initialize stencil buffer with correct values.



Eisemann et al. 2010

The limitations of shadow volumes

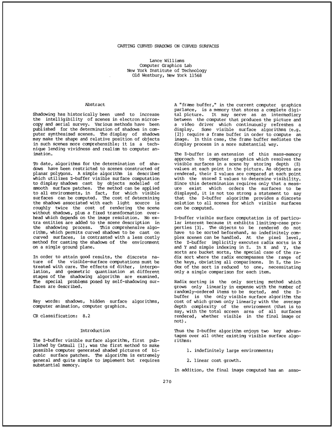
- Shadow volumes give exact shadows, but…
- Extracting silhouette edges and rasterising shadow polygons becomes *infeasible* for more complex scenes.



Crysis Warhead, CryTek

Shadow mapping

- So, shadow volumes are too heavy for complex scenes. What to do?
- Shadow mapping.
- Pros:
  - No explicit processing of scene geometry required!
  - Fewer rendering passes than shadow volumes for a single light source.
  - (No need to use stencil buffer.)
- Cons:
  - Aliasing artifacts may occur.
  - Needs an extra rendering pass for each additional light.



CASTING CURVED SHADOWS ON CURVED SURFACES

Lance Williams
Computer Graphics Lab
New York Institute of Technology
Old Westbury, New York 11568

Abstract

Shadowing has historically been used to increase the intelligibility of scenes in electron microscopy and aerial survey. Various methods have been published for the determination of shadows in computer synthesized scenes. The display of shadows may make the shape and relative position of objects in such scenes more comprehensible; it is a technique lending vividness and realism to computer animation.

To date, algorithms for the determination of shadows have been restricted to scenes constructed of planar polygons. A simple algorithm is described which utilizes Z-buffer visible surface computation to display shadows cast by objects modelled of smooth surface patches. The method can be applied to all environments, in fact, for which visible surfaces can be computed. The cost of determining the shadows associated with each light source is roughly twice the cost of rendering the scene without shadows, plus a fixed transformation overhead which depends on the image resolution. No extra entities are added to the scene description in the shadowing process. This comprehensive algorithm, which permits curved shadows to be cast on curved surfaces, is contrasted with a less costly method for casting the shadows of the environment on a single ground plane.

In order to attain good results, the discrete nature of the visible-surface computations must be treated with care. The effects of dither, interpolation, and geometric quantization at different stages of the shadowing algorithm are examined. The special problems posed by self-shadowing surfaces are described.

Key words: shadows, hidden surface algorithms, computer animation, computer graphics.

CR classification: 8.2

Introduction

The Z-buffer visible surface algorithm, first published by Catmull [1], was the first method to make possible computer generated shaded pictures of bicubic surface patches. The algorithm is extremely general and quite simple to implement but requires substantial memory.

A "frame buffer," in the current computer graphics parlance, is a memory that stores a complete digital picture. It may serve as an intermediary between the computer that produces the picture and a video driver which continuously refreshes a display. Some visible surface algorithms (e.g. [2]) require a frame buffer in order to compute an image. In this case, the frame buffer mediates the display process in a more substantial way.

The Z-buffer is an extension of this mass-memory approach to computer graphics which resolves the visible surfaces in a scene by storing depth (Z) values at each point in the picture. As objects are rendered, their Z values are compared at each point with the stored Z values to determine visibility. Since this determination requires only that a measure exist which orders the surfaces to be displayed, it is not too strong a statement to say that the Z-buffer algorithm provides a discrete solution to all scenes for which visible surfaces can be computed.

Z-buffer visible surface computation is of particular interest because it exhibits limiting-case properties [3]. The objects to be rendered do not have to be sorted beforehand, so indefinitely complex scenes can be handled. At the pixel level, the Z-buffer implicitly executes radix sorts in X and Y and simple indexing in Z. In X and Y, the sorts are bucket sorts, the special case of the radix sort where the radix encompasses the range of the keys, obviating all comparisons. In Z, the index of the sort is reduced to one, necessitating only a single comparison for each item.

Radix sorting is the only sorting method which grows only linearly in expense with the number of randomly-ordered items to be sorted, and the Z-buffer is the only visible surface algorithm the cost of which grows only linearly with the average depth complexity of the environment (that is to say, with the total screen area of all surfaces rendered, whether visible in the final image or not).

Thus the Z-buffer algorithm enjoys two key advantages over all other existing visible surface algorithms:

1. indefinitely large environments;

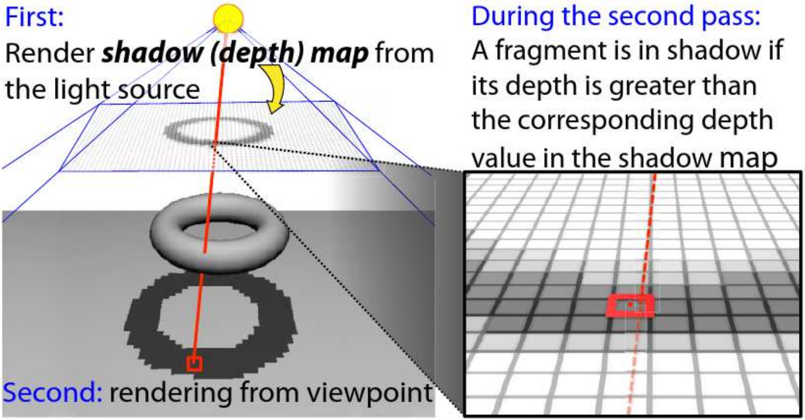
2. linear cost growth.

In addition, the final image computed has an asso-

270

Williams 1978

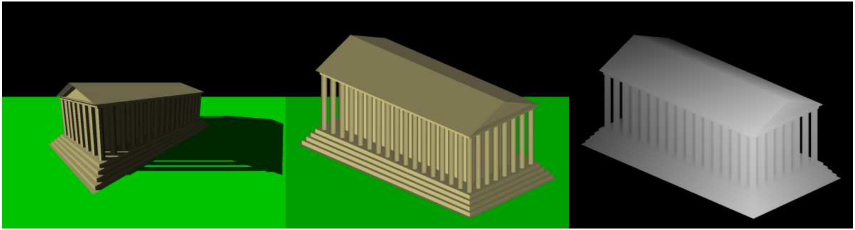
Shadow map algorithm in a nutshell



Eisemann et al. 2010

Shadow mapping algorithm (1/3)

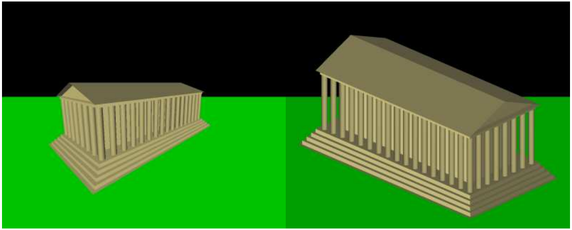
- All lit parts can be "seen" by the (point) light source.
- We have efficient implementations of how to determine which parts of objects are seen from a point:
- render an image from the light source's position!
- The *depth buffer* from this point is the *shadow map*.
- (Store it as a texture.)



Praetor alpha, via Wikipedia

Shadow mapping algorithm (2/3)

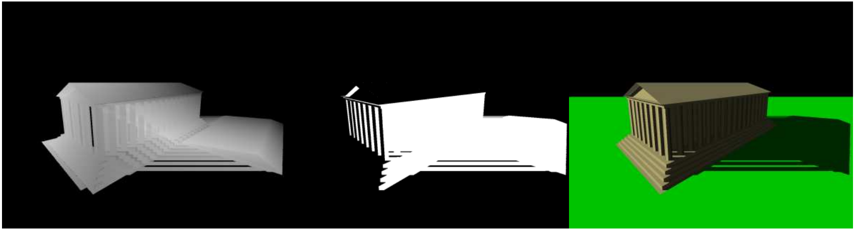
- Render the scene from the camera view as usual.
- Transform each fragment's *xyz* coordinates into *light space*.
- How?
  - We have $\mathbf{p} = (x, y, z)$ in view space; **V**: the view matrix (brings point in world space to *view* space); **L**: (brings point in world space into *light* space).
  - (For shadow mapping, our light source needs an orientation in addition to position.)
  - $\mathbf{L}\mathbf{V}^{-1}\mathbf{p} = \mathbf{p}' = (x_s, y_s, z_s)$ fragment's position in light space.



Praetor alpha, via Wikipedia

Shadow mapping algorithm (3/3)

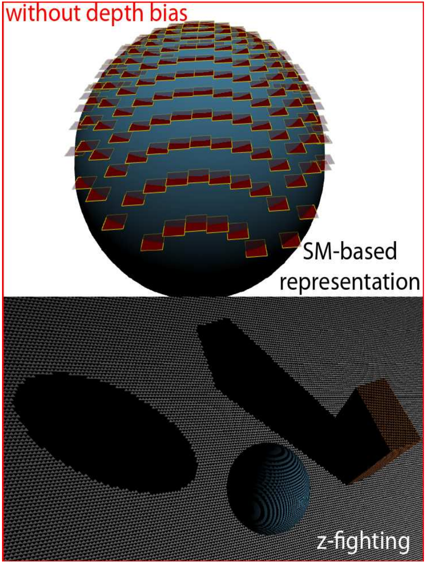
- Now, (x_s, y_s) correspond to (u, v) in the shadow map.
- z_s is the distance from the light to this fragment.
- How to determine if this fragment is occluded or visible from light source?
- Compare z_s and the depth map value d at position (x_s, y_s).
- $z_s > d \implies$ fragment is in shadow.
- $z_s = d \implies$ fragment is lit.



Praetor alpha, via Wikipedia

Issue: Z fighting

- For points that are visible from both camera and light source: shadow_map$(x_s, y_s) \approx z_s$.
- Rounding errors result in random noise.
- Z-fighting, a.k.a. shadow acne.
- (Top figure shows low-resolution shadow map $\rightarrow$ aliasing.)



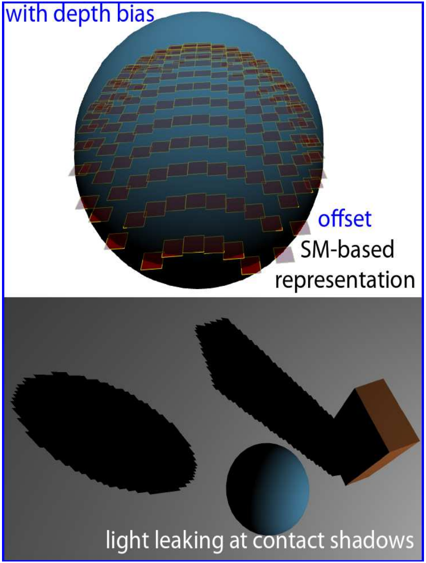
Eisemann et al. 2010

Workaround: bias

- Add bias b:

  $\text{shadow\_map}(x_s, y_s) + b < z_s$

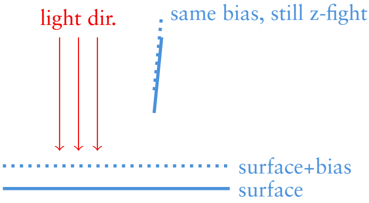
- How much bias is good?
- Too little: shadow acne.
- Too much: *light leak*.



Eisemann et al. 2010

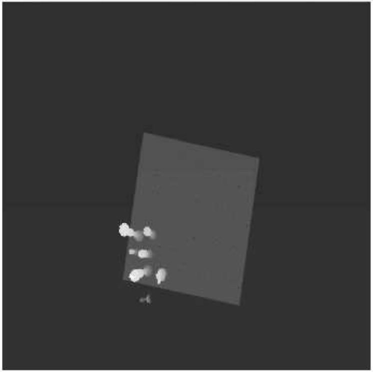
Depth bias: caveats

- If a face mostly aligns with the light's view, a much larger bias can be necessary.
- Usually, two parameters are available, a constant offset and an offset that depends on the alignment of the triangle with the light's view rays.
- Needs to be hand-adjusted.
- E g, for a very short triangle, too much offset $\implies$ shadow depth does not correlate to geometry.



Issue: Shadow map aliasing

- Shadow map has limited resolution.
- Shadow-map pixels (texels) do not correspond 1-to-1 with screen pixels.
- Workaround 1: high-res shadow maps. . .

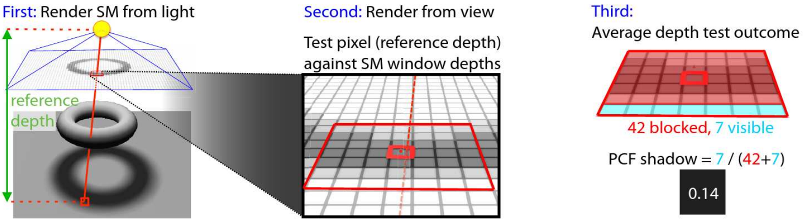


$\Longrightarrow$

Stamminger et al. 2002

Workaround 2: Percentage-closer filtering

- Countering aliasing: interpolation?
- Does not work for depth!
- Compute average of surrounding depth tests instead.

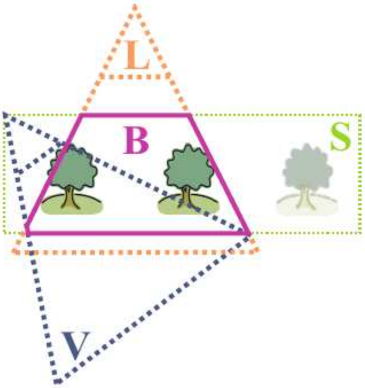


Eisemann et al. 2010

Read more in Eisemann et al. 2010 if you are interested in the details about these workarounds.

Workaround 3: Fitting

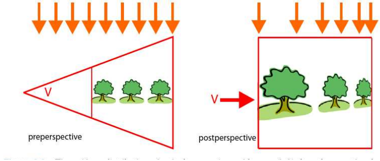
- Make sure shadow map is fitted to view frustum.
- (Making sure no shadow texels are wasted.)
- Can cause *temporal aliasing* because fitting is recomputed each frame.



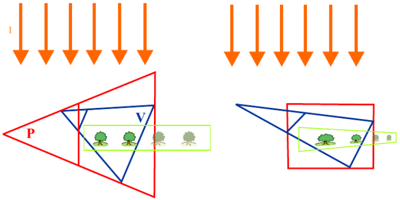
Eisemann et al. 2010

Workaround 4: Warping

- Perspective shadow maps.
- Transform scene (log along z axis) before projecting into shadow map.
- Leads to more uniform samples.



Without warping, fewer shadow map texels near camera.

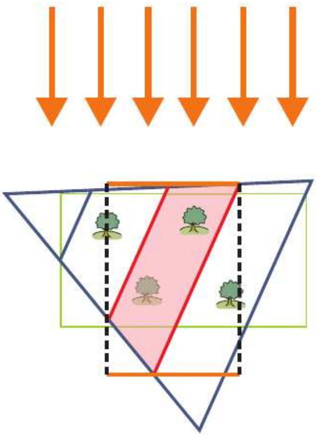


Warp scene with $\log z$ before creating shadow map.

Eisemann et al. 2010

Workaround 5: Partitioning

- Use separate shadow maps, partitioned along view direction.



Eisemann et al. 2010

Issue: Omnidirectional light sources

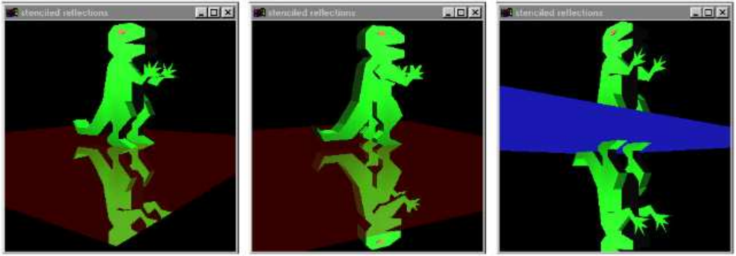
- Shadow map is computed with standard rendering.
- So we are confined to a “light frustum”.

Workaround: multi-pass shadow mapping

- Need to render several (6) shadow maps.

(Planar reflections + stencil buffer)

- The stencil buffer can also be used to work around some of the limitations of planar shadows and reflections:



Kilgard 1999

Left: proper stenciling. Middle: head peeks beyond surface. Right: showing that reflection is just a mirrored object.

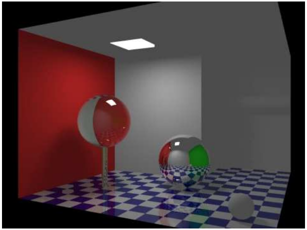
Shadow maps or shadow volumes?

- Shadow maps are generally faster (governed by cost of rendering the image for the viewpoint)
- and can generate shadows from any rasterizable geometry (not only polygonal).
- But shadow maps have biasing issues, undersampling artifacts (jaggy edges),
- and are limited to a single frustum: so omnidirectional lights (as opposed to directional spotlights) require 6 shadow maps.
- Shadow volumes produce perfectly sharp shadows, but require 3 render passes per light, additional geometry processing, and the elongated shadow quads cause a high *fill-rate*.

Summary

- Casting shadows on other objects (with rasterisation)
- Shadow volumes
  - Gives exact hard shadows
  - Needs polygonal geometry
  - Doesn’t scale well with complex geometry
- Shadow maps
  - Less sensitive to complex geometry
  - Useful also for non-polygon geometry
  - Issues with z-fighting and aliasing

Next lecture: ray tracing



- Fri Nov 18, 13.15–15.00
- T-211
- Hughes et al:
  - 15.1–15.2.4, 15.4–15.4.1, 15.4.3
  - 7.8
  - 29

References

- Franklin C Crow (1977). "Shadow Algorithms for Computer Graphics". In: *SIGGRAPH '77*, pp. 242–248.
- Elmar Eisemann et al. (2010). "Shadow Algorithms for Real-time Rendering: EuroGraphics 2010 Tutorial Notes". In: *EuroGraphics*.
- Mark J. Kilgard (1999). "Advanced OpenGL Game Development". In: Nvidia. Chap. Improving shadows and reflections via the stencil buffer, pp. 204–253. URL: https://developer.nvidia.com/sites/default/files/akamai/gamedev/docs/stencil.pdf.
- Marc Stamminger and George Drettakis (2002). "Perspective Shadow Maps". In: *SIGGRAPH '02*. San Antonio, Texas: ACM, pp. 557–562. ISBN: 1-58113-521-1. DOI: 10.1145/566570.566616. URL: http://doi.acm.org/10.1145/566570.566616.
- Lance Williams (1978). "Casting curved shadows on curved surfaces". In: *SIGGRAPH '78*, pp. 270–274.