Shadow Algorithms

CPSC 314

© Wolfgang Heidrich

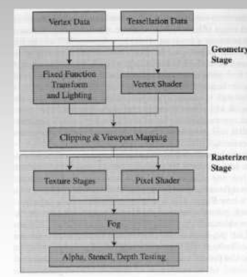
Last Lecture: Modern GPU Architectures

Vertex shader

- Replaces model/view, lighting, and perspective
- Have to implement these yourself
- But can also implement much more

Fragment/pixel shader

- Replaces texture mapping
- Fragment shader must do texturing
- But can do other things



© Wolfgang Heidrich

Vertex Program Properties

Run for every vertex, independently

- Access to all per-vertex properties
  - *Position, color, normal, texture coords, other custom properties*
- Access to read/write registers for temporary results
  - *Value is reset for every vertex*
  - *I.e. cannot pass information from one vertex to the next*
- Access to read-only registers
  - *Global variables, like ligt position, transformation matrices*
- Write output to a specific register for the resulting color

© Wolfgang Heidrich

Vertex Shaders/Programs

Concept:

- Programmable pipeline stage
  - *Floating-point operations on 4 vectors*
    - Points, vectors, and colors!
- Replace all of
  - *Model/View Transformation*
  - *Lighting*
  - *Perspective projection*

© Wolfgang Heidrich

Vertex Programs – Instruction Set

Arithmetic Operations on 4-vectors:

- ADD, MUL, MAD, MIN, MAX, DP3, DP4

Operations on Scalars

- RCP (1/x), RSQ (1/√x), EXP, LOG

Specialty Instructions

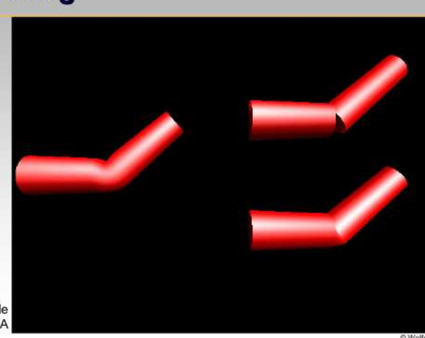
- DST (distance: computes length of vector)
- LIT (quadratic falloff term for lighting)

Very latest generation:

- Loops and conditional jumps

© Wolfgang Heidrich

Skinning



Example by NVIDIA

© Wolfgang Heidrich

Fragment Shaders

- *Fragment shaders* operate on fragments in place of the texturing hardware
  - *After rasterization, before any fragment tests or blending*
- Input: The fragment, with screen position, depth, color, and a set of texture coordinates
- Access to textures and some constant data and registers
- Compute RGBA values for the fragment, and depth
  - *Can also "kill "a fragment, that is throw it away*
- Two types of fragment shaders: register combiners (GeForce4) and fully programmable (GeForceFX, Radeon 9700)

© Wolfgang Heidrich

High Level Shading Languages e.g. Cg

Cg is a high-level language developed by NVIDIA

- It looks like C or C++
- Actually a language and a runtime environment
  - *Can compile ahead of time, or compile on the fly*
  - *Why compile on the fly?*
- What it can do is tightly tied to the hardware
  - *How does it know which hardware, and how to use it?*

© Wolfgang Heidrich

Vertex Program Example

```
void C5E2v_fragmentLighting(float4 position : POSITION,
                            float3 normal   : NORMAL,

                        out float4 oPosition : POSITION,
                        out float3 objectPos : TEXCOORD0,
                        out float3 oNormal   : TEXCOORD1,

                    uniform float4x4 modelViewProj)
{
  oPosition = mul(modelViewProj, position);
  objectPos = position.xyz;
  oNormal = normal;
}
```

© Wolfgang Heidrich

Pixel Program Example

```
void C5E3f_basicLight(float4 position : TEXCOORD0,
                      float3 normal   : TEXCOORD1,

                  out float4 color    : COLOR,

              uniform float3 globalAmbient,
              uniform float3 lightColor,
              uniform float3 lightPosition,
              uniform float3 eyePosition,
              uniform float3 Ke,
              uniform float3 Ka,
              uniform float3 Kd,
              uniform float3 Ks,
              uniform float  shininess)
{
  float3 P = position.xyz;
  float3 N = normalize(normal);

  // Compute the emissive term
  float3 emissive = Ke;

  // Compute the ambient term
  float3 ambient = Ka * globalAmbient;

  // Compute the diffuse term
  float3 L = normalize(lightPosition - P);
  float diffuseLight = max(dot(N, L), 0);
  float3 diffuse = Kd * lightColor * diffuseLight;

  // Compute the specular term
  float3 V = normalize(eyePosition - P);
  float3 H = normalize(L + V);
  float specularLight = pow(max(dot(N, H), 0),
                            shininess);
  if (diffuseLight <= 0) specularLight = 0;
  float3 specular = Ks * lightColor * specularLight;

  color.xyz = emissive + ambient + diffuse + specular;
  color.w = 1;
}
```

© Wolfgang Heidrich

Shadow Algorithms

CPSC 314

© Wolfgang Heidrich

Shadows

Types of light sources

- Point, directional
- Area lights and generally shaped lights
  - *Not considered here*
  - *Later: ray-tracing for such light sources*

Problem statement

- A shadow algorithm for point and directional lights determines which scene points are
  - *Visible from the light source (I.e. illuminated)*
  - *Invisible from the light source (I.e. in shadow)*
- Thus: shadow casting is a visibility problem!

© Wolfgang Heidrich

Types of Shadow Algorithms

Object Space

- Like object space visibility algorithms, the method computes in object space which polygon parts that are illuminated and which are in shadow
  - *Individuyal parts are then drawn with different intensity*
- Typically slow, O(n^2), not for dynamic scenes

Image Space

- Determine visibility per pixel in the final image
  - *Sort of like depth buffer*
  - *Shadow maps*
  - *Shadow volumes*

© Wolfgang Heidrich

Credits

- The following shadow mapping slides are taken from Mark Kilgard's OpenGL course at Siggraph 2002.

© Wolfgang Heidrich

Shadow Mapping Concept (1)

Depth testing from the light's point-of-view

- Two pass algorithm
- First, render depth buffer from the light's point-of-view
  - The result is a "depth map" or "shadow map"
  - Essentially a 2D function indicating the depth of the closest pixels to the light
- This depth map is used in the second pass

© Wolfgang Heidrich

Shadow Mapping Concept (2)

Shadow determination with the depth map

- Second, render scene from the eye's point-of-view
- For each rasterized fragment
  - Determine fragment's XYZ position relative to the light
  - This light position should be setup to match the frustum used to create the depth map
  - Compare the depth value at light position XY in the depth map to fragment's light position Z

© Wolfgang Heidrich

The Shadow Mapping Concept (3)

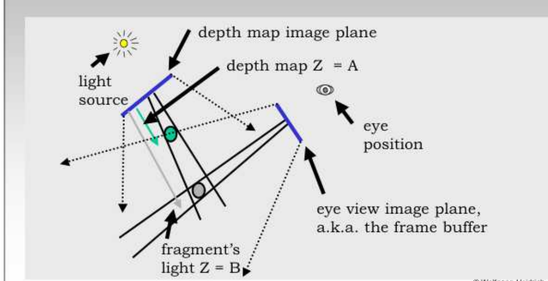
The Shadow Map Comparison

- Two values
  - A = Z value from depth map at fragment's light XY position
  - B = Z value of fragment's XYZ light position
- If B is greater than A, then there must be something closer to the light than the fragment
  - Then the fragment is shadowed
- If A and B are approximately equal, the fragment is lit

© Wolfgang Heidrich

Shadow Mapping with a Picture in 2D (1)

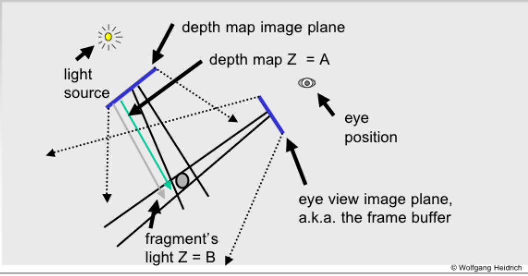
The A < B shadowed fragment case



© Wolfgang Heidrich

Shadow Mapping with a Picture in 2D (2)

The A ≈ B unshadowed fragment case

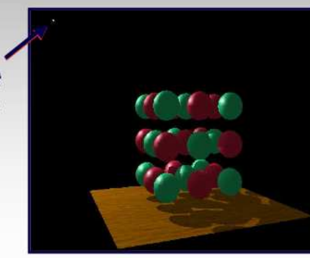


© Wolfgang Heidrich

Visualizing the Shadow Mapping Technique (1)

A scene with fairly complex shadows

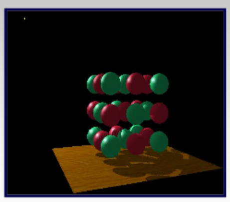
the point light source



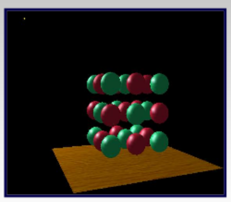
© Wolfgang Heidrich

Visualizing the Shadow Mapping Technique (2)

Compare with and without shadows



with shadows

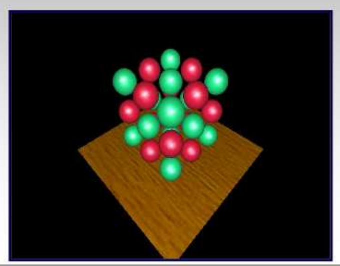


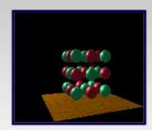
without shadows

© Wolfgang Heidrich

Visualizing the Shadow Mapping Technique (3)

The scene from the light's point-of-view



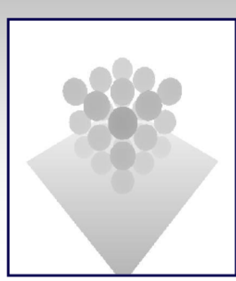


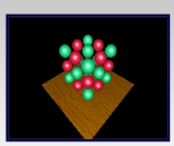
FYI: from the eye's point-of-view again

© Wolfgang Heidrich

Visualizing the Shadow Mapping Technique (4)

The depth buffer from the light's point-of-view



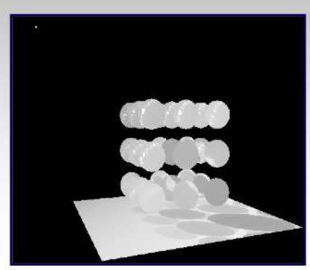


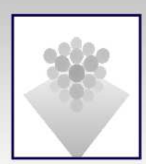
FYI: from the light's point-of-view again

© Wolfgang Heidrich

Visualizing the Shadow Mapping Technique (5)

Projecting the depth map onto the eye's view



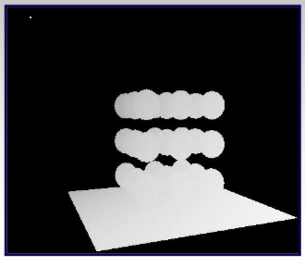


FYI: depth map for light's point-of-view again

© Wolfgang Heidrich

Visualizing the Shadow Mapping Technique (6)

Projecting light's planar distance onto eye's view



© Wolfgang Heidrich

Visualizing the Shadow Mapping Technique (6)

Comparing light distance to light depth map

Green is where the light planar distance and the light depth map are approximately equal

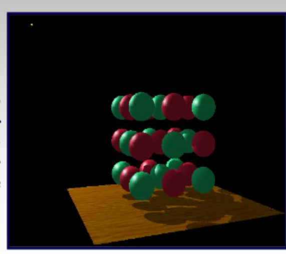
Non-green is where shadows should be

© Wolfgang Heidrich

Visualizing the Shadow Mapping Technique (7)

Complete scene with shadows

Notice how specular highlights never appear in shadows

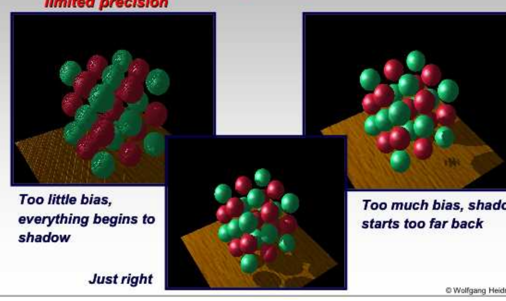


Notice how curved surfaces cast shadows on each other

© Wolfgang Heidrich

In Practice: Depth Map Precision Issues

Have to add a little offset to depth map values to account for limited precision



Too little bias, everything begins to shadow

Just right

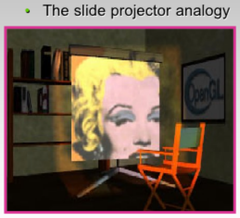
Too much bias, shadow starts too far back

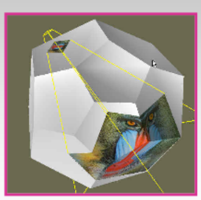
© Wolfgang Heidrich

What is Projective Texturing?

An intuition for projective texturing

- The slide projector analogy





© Wolfgang Heidrich

About Projective Texturing (1)

First, what is perspective-correct texturing?

- Normal 2D texture mapping uses (s, t) coordinates
- 2D perspective-correct texture mapping
  - Means (s, t) should be interpolated linearly in eye-space
  - So compute per-vertex s/w, t/w, and 1/w
  - Linearly interpolated these three parameters over polygon
  - Per-fragment compute s' = (s/w) / (1/w) and t' = (t/w) / (1/w)
  - Results in per-fragment perspective correct (s', t')

© Wolfgang Heidrich

About Projective Texturing (2)

So what is projective texturing?

- Now consider homogeneous texture coordinates
  - (s, t, r, q) --> (s/q, t/q, r/q)
  - Similar to homogeneous clip coordinates where (x, y, z, w) = (x/w, y/w, z/w)
- Idea is to have (s/q, t/q, r/q) be projected per-fragment

© Wolfgang Heidrich

Back to the Shadow Mapping Discussion . . .

Assign light-space texture coordinates to polygon vertices

- Transform eye-space (x, y, z, w) coordinates to the light's view frustum (match how the light's depth map is generated)
- Further transform these coordinates to map directly into the light view's depth map
  - *Expressible as a projective transform*
- (s/q, t/q) will map to light's depth map texture

© Wolfgang Heidrich

Shadow Map Operation

Next Step:

- Compare depth map value to distance of fragment from light source
- Different GPU generations support different means of implementing this
  - *Today's GPUs: pixel shader!*
  - *Earlier: special hardware for implmenting this feature (e.g. SGI), or just using alpha blending [Heidrich'99]*

© Wolfgang Heidrich

Issues with Shadow Mapping (1)

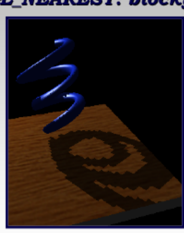
Not without its problems

- Prone to aliasing artifacts
  - "percentage closer" filtering helps this
  - normal color filtering does **not** work well
- Depth bias is not completely foolproof
- Requires extra shadow map rendering pass and texture loading
- Higher resolution shadow map reduces blockiness
  - but also increase texture copying expense

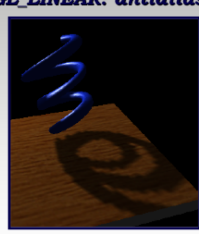
© Wolfgang Heidrich

Hardware Shadow Map Filtering Example

GL_NEAREST: blocky



GL_LINEAR: antialiased edges



Low shadow map resolution used to heightens filtering artifacts

© Wolfgang Heidrich

Issues with Shadow Mapping (2)

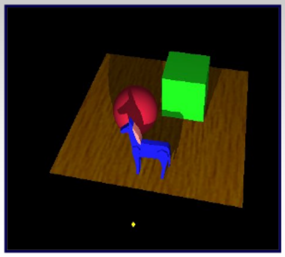
Not without its problems

- Shadows are limited to view frustums
  - could use six view frustums for omni-directional light
- Objects outside or crossing the near and far clip planes are not properly accounted for by shadowing
  - move near plane in as close as possible
  - but too close throws away valuable depth map precision when using a projective frustum

© Wolfgang Heidrich

More Examples

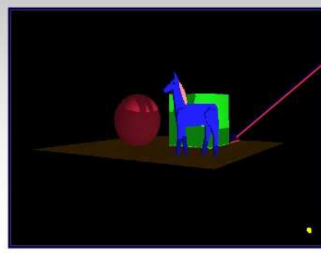
Complex objects all shadow



© Wolfgang Heidrich

More Examples

Even the floor casts shadow



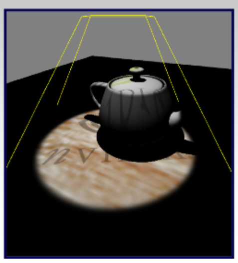
Note shadow leakage due to infinitely thin floor

Could be fixed by giving floor thickness

© Wolfgang Heidrich

Combining Projective Texturing for Spotlights

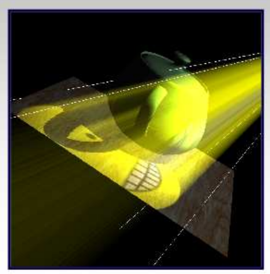
Use a spotlight-style projected texture to give shadow maps a spotlight falloff



© Wolfgang Heidrich

Combining Shadows with Atmospherics

Shadows in a dusty room



Simulate atmospheric effects such as suspended dust

1) *Construct shadow map*
2) *Draw scene with shadow map*
3) *Modulate projected texture image with projected shadow map*
4) *Blend back-to-front shadowed slicing planes also modulated by projected texture image*

Credit: Cass Everitt

© Wolfgang Heidrich

Shadow Maps

Approach for shadows from point light sources

- Surface point is in shadow if it is not visible from the light source
- Use depth buffer to test visibility:
  - *Render scene from the point light source*
  - *Store resulting depth buffer as texture map*
  - *For every fragment generated while rendering from the camera position, project the fragment into the depth texture taken from the camera, and check if it passes the depth test.*

© Wolfgang Heidrich

Shadow Volumes

Use new buffer: stencil buffer

- Just another channel of the framebuffer
- Can count how often a pixel is drawn

Algorithm (1):

- Generate silhouette polygons for all objects
  - *Polygons starting at silhouette edges of object*
  - *Extending away from light source towards infinity*
  - *These can be computed in vertex programs*

© Wolfgang Heidrich

Shadow Volumes

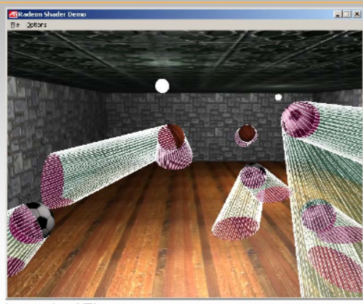


Image by ATI

© Wolfgang Heidrich

Shadow Volumes

Algorithm (2):

- Render all original geometry into the depth buffer
  - *I.e. do not draw any colors (or only draw ambient illumination term)*
- Render <u>front-facing</u> silhouette polygons while <u>incrementing</u> the stencil buffer for every rendered fragment
- Render <u>back-facing</u> silhouette polygons while <u>decrementing</u> the stencil buffer for every rendered fragment
- Draw illuminated geometry where the stencil buffer is 0, shadow elsewhere

© Wolfgang Heidrich

Shadow Volumes

Image by ATI

© Wolfgang Heidrich

Shadow Volumes

Discussion:

- Object space method therefore no precision issues
- Lots of large polygons: can be slow
  - *High geometry count*
  - *Large number of pixels rendered*

© Wolfgang Heidrich

Coming Up…

Tuesday:

- Color

Thursday:

- Ray-tracing

© Wolfgang Heidrich