ME 290-R:

General Purpose Computation (CAD/CAM/CAE) on the GPU

(a.k.a. Topics in Manufacturing)

Sara McMains

Spring 2009

Lecture 9

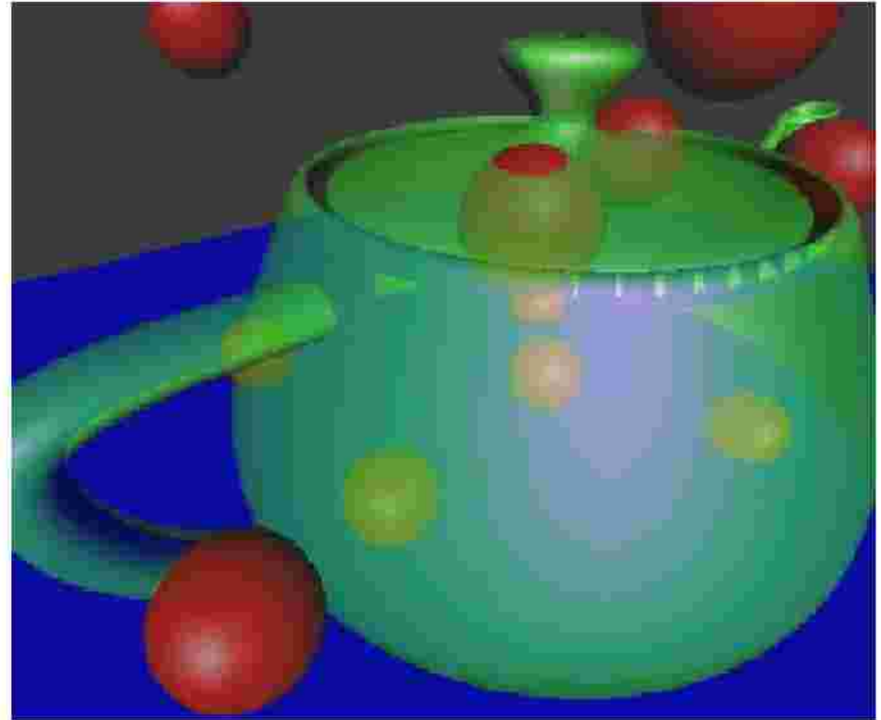
Outline

- Visibility
  - Depth peeling
  - Projective texturing
  - Shadow mapping
- Efficient Data Parallel Computing

Depth Peeling

- Fragment-level depth sorting technique
  - each pass peels off a “depth-layer”
    - each additional layer requires 1 more pass
    - n layers in n passes
  - primitive-order-independent
- “Interactive Order-Independent Transparency” by Cass Everitt, NVIDIA
  - Mammen ‘89 “virtual pixel maps”
  - Diefenbach ‘96 “dual depth buffers”

Order-Independent Transparency

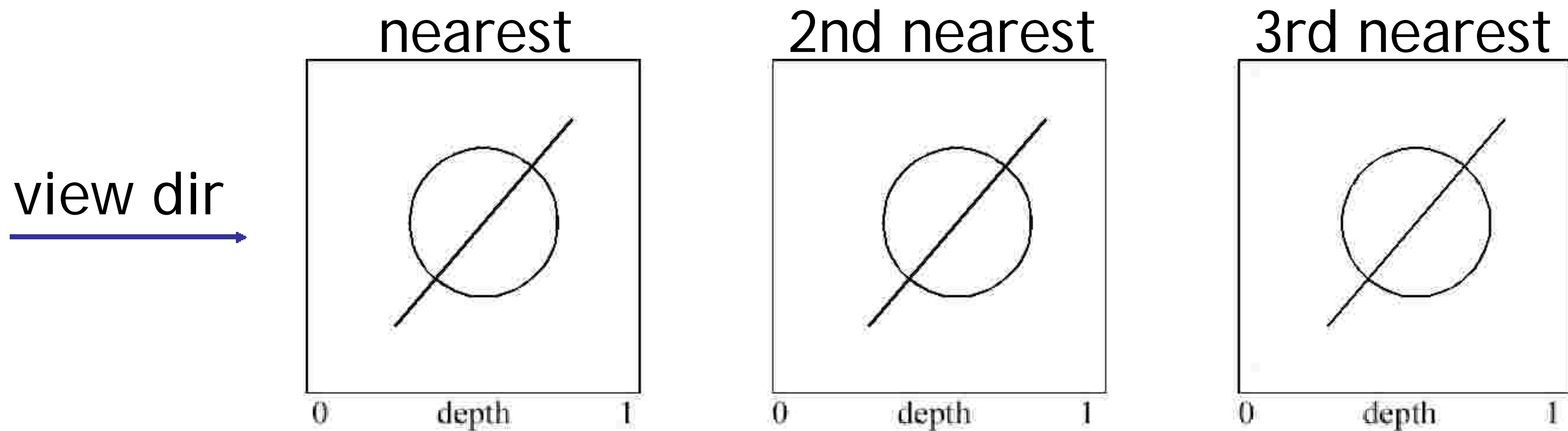


Order-Independent　　Order-Dependent

Credit: Cass Everitt

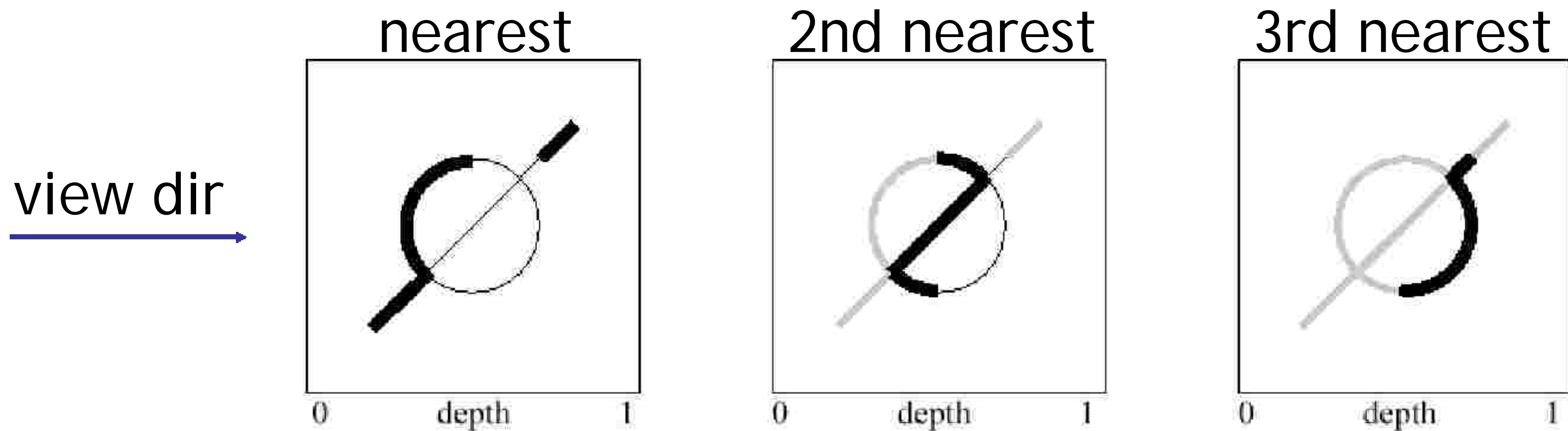
Depth Peeling

- Standard z-buffering returns nearest fragment
  - alternately, farthest
  - no way to get 2nd nearest, n-th nearest



Depth Peeling

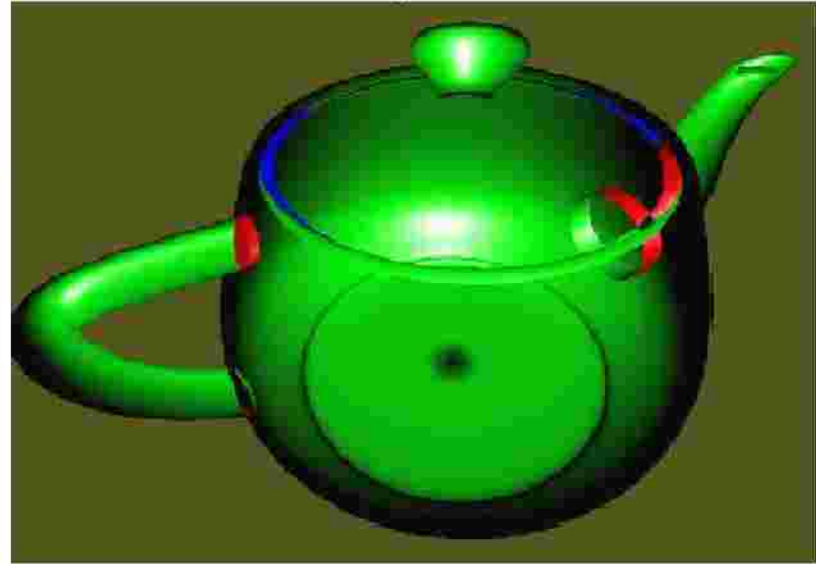
- Standard z-buffering returns nearest fragment
  - alternately, farthest
  - no way to get 2nd nearest, n-th nearest



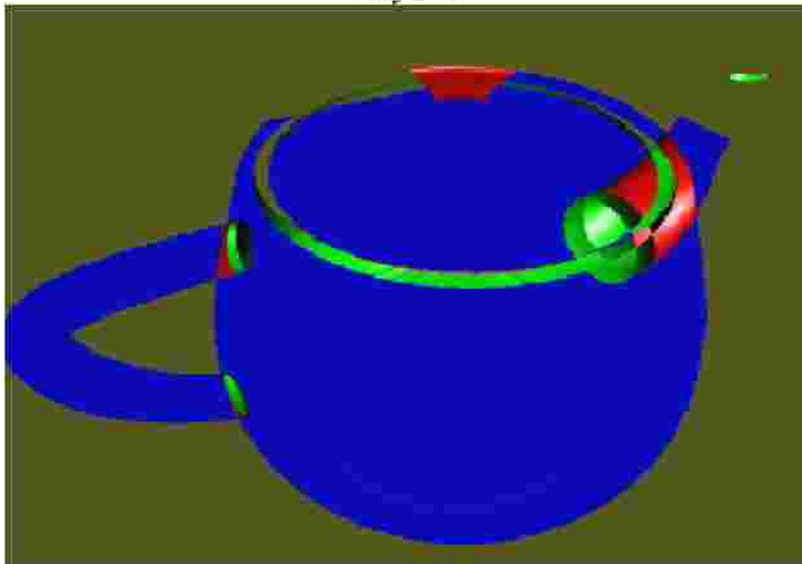
Example (red exterior, green interior)

Layer 0

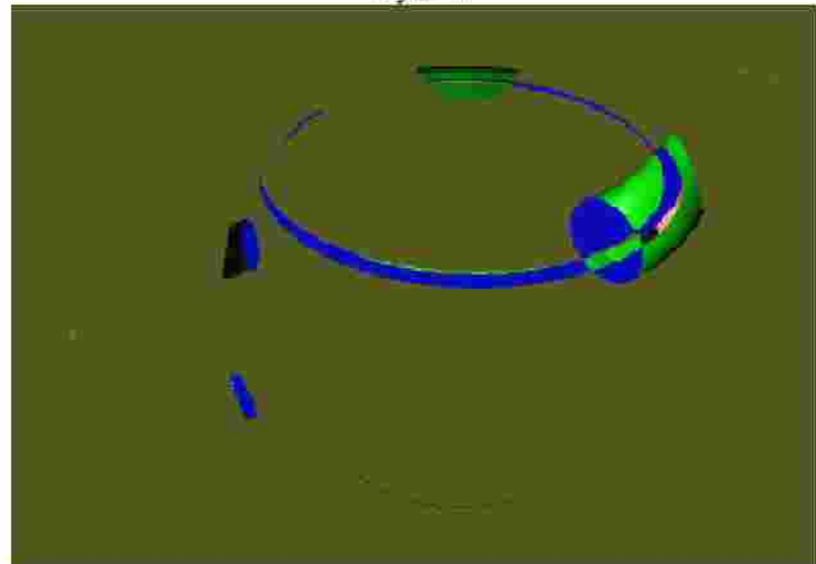
Layer 1



Layer 2



Layer 3



Depth Peeling

- Layer 0 uses standard z-buffer
  - nearest fragment
  - store the final depth buffer
- Next layer
  - peel away fragments w/ depth <= nearest depth stored in previous pass
  - keep the nearest remaining
  - a "two-sided" depth test
- Use depth buffer from 2nd layer to compute 3rd, etc.

Dual depth buffer intuition

- Pretend we have multiple “depth units” with their own depth buffers
  - executed sequentially
  - first to fail discards fragment
- For two units
  - unit 0 is “peel unit”
    - discards but doesn’t write
  - unit 1 is the regular unit that does depth writes

Pseudocode

```
for (i=0; i<num_passes; i++)
   {
        clear color buffer
        A = i % 2
        B = (i+1) % 2
        depth unit 0:
              if(i == 0)
                    disable depth test
              else
                    enable depth test
              bind buffer A
              disable depth writes
              set depth func to GREATER
        depth unit 1:
              bind buffer B
              clear depth buffer
              enable depth writes
              enable depth test
              set depth func to LESS
        render scene
        save color buffer RGBA as layer i
   }
```

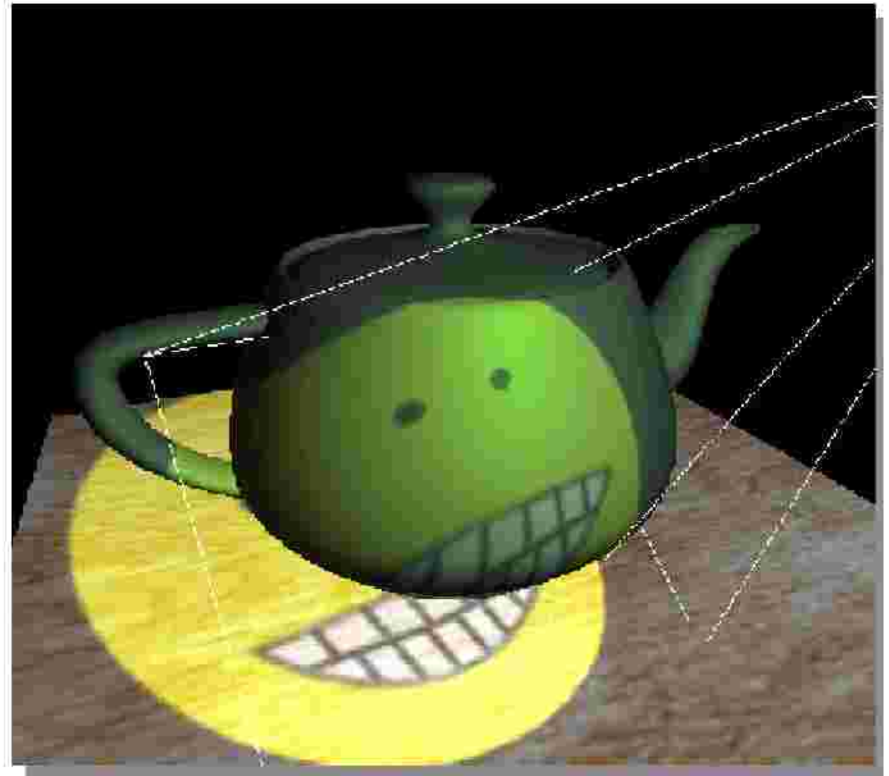
further back than stored depth, or discarded

closest depth (not discarded previously)

Outline

- Visibility
  - Depth peeling
  - Projective texturing
  - Shadow mapping
- Efficient Data Parallel Computing

Projective Texturing

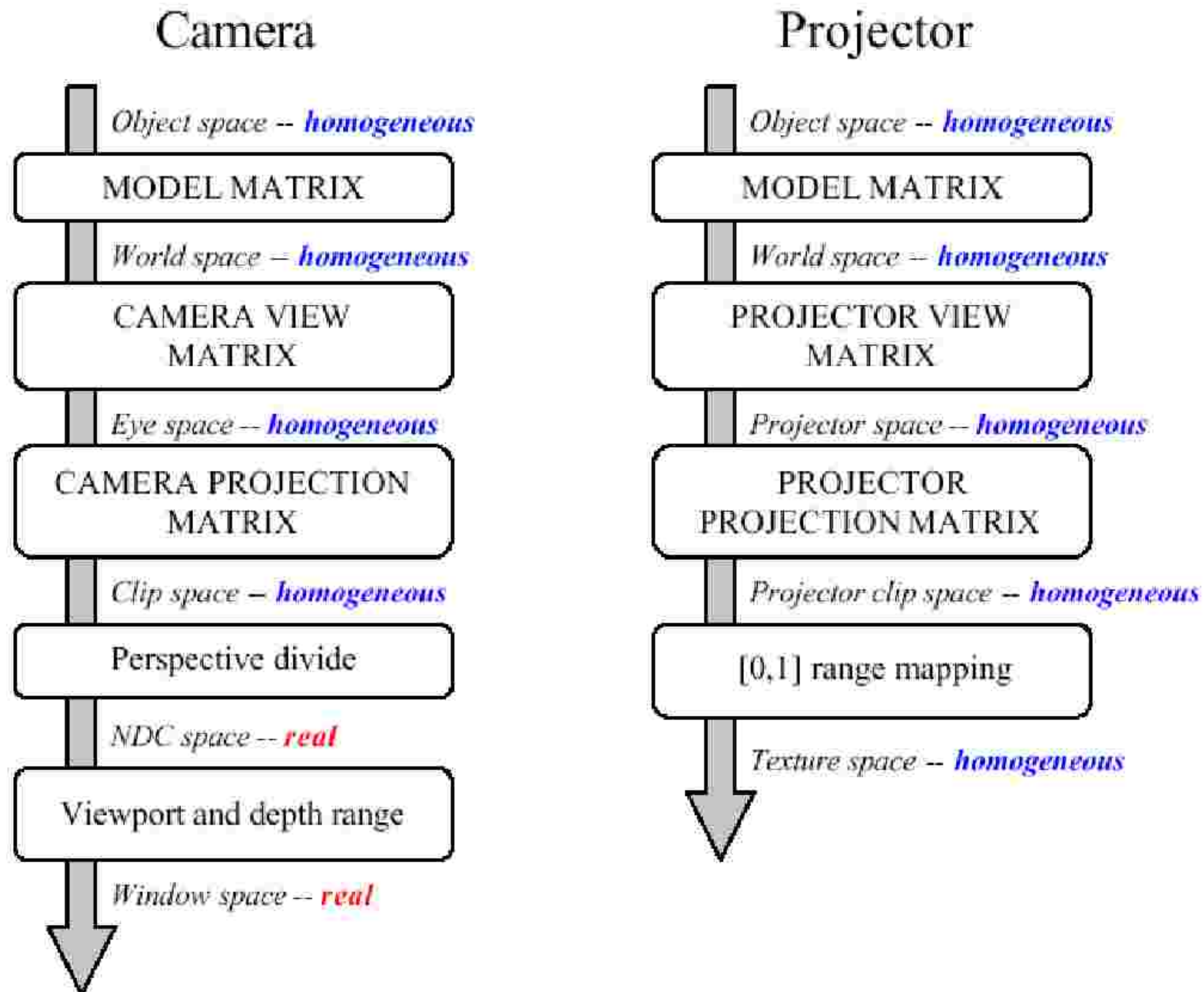


Projective Texturing

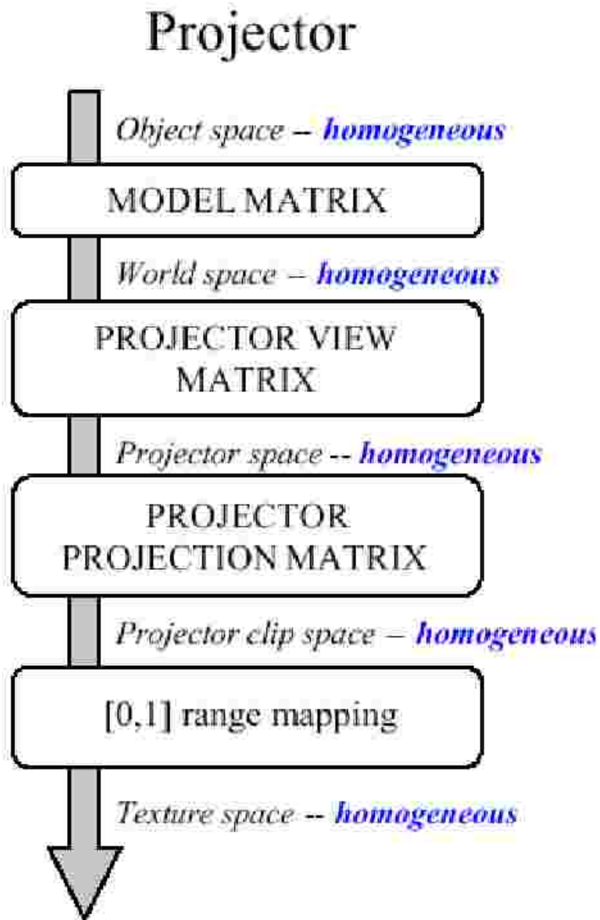
- As if texture map is a transparency, COP is light
  - but no occlusion checks,
  - no shadowing

- Standard texturing
  - vertices have tex. coords
- Projective texturing
  - texture coords must be calculated for each vtx
  - imagine ray from vtx to light
    - its intersection w/ texture tells you the coord to assign

Projective Texturing

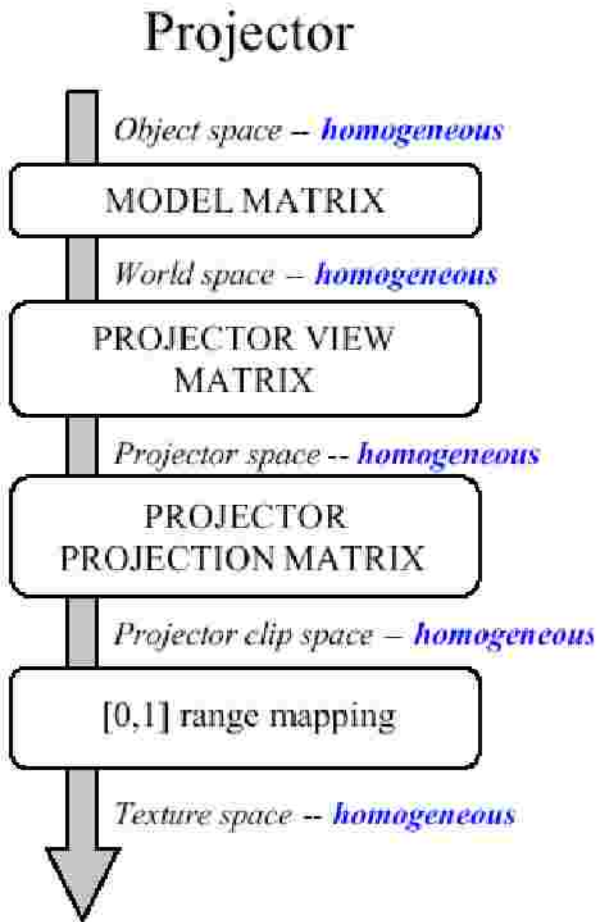


Projective Texturing



$$\begin{bmatrix} s \\ t \\ r \\ q \end{bmatrix} = \mathbf{T}_o \begin{bmatrix} v_x \\ v_y \\ v_z \\ v_w \end{bmatrix}_{object}$$

Projective Texturing



$$\begin{bmatrix} s \\ t \\ r \\ q \end{bmatrix} = \mathbf{T}_o \begin{bmatrix} v_x \\ v_y \\ v_z \\ v_w \end{bmatrix}_{object}$$

$$\mathbf{T}_o = \begin{bmatrix} \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{P}_p \mathbf{V}_p \mathbf{M}$$

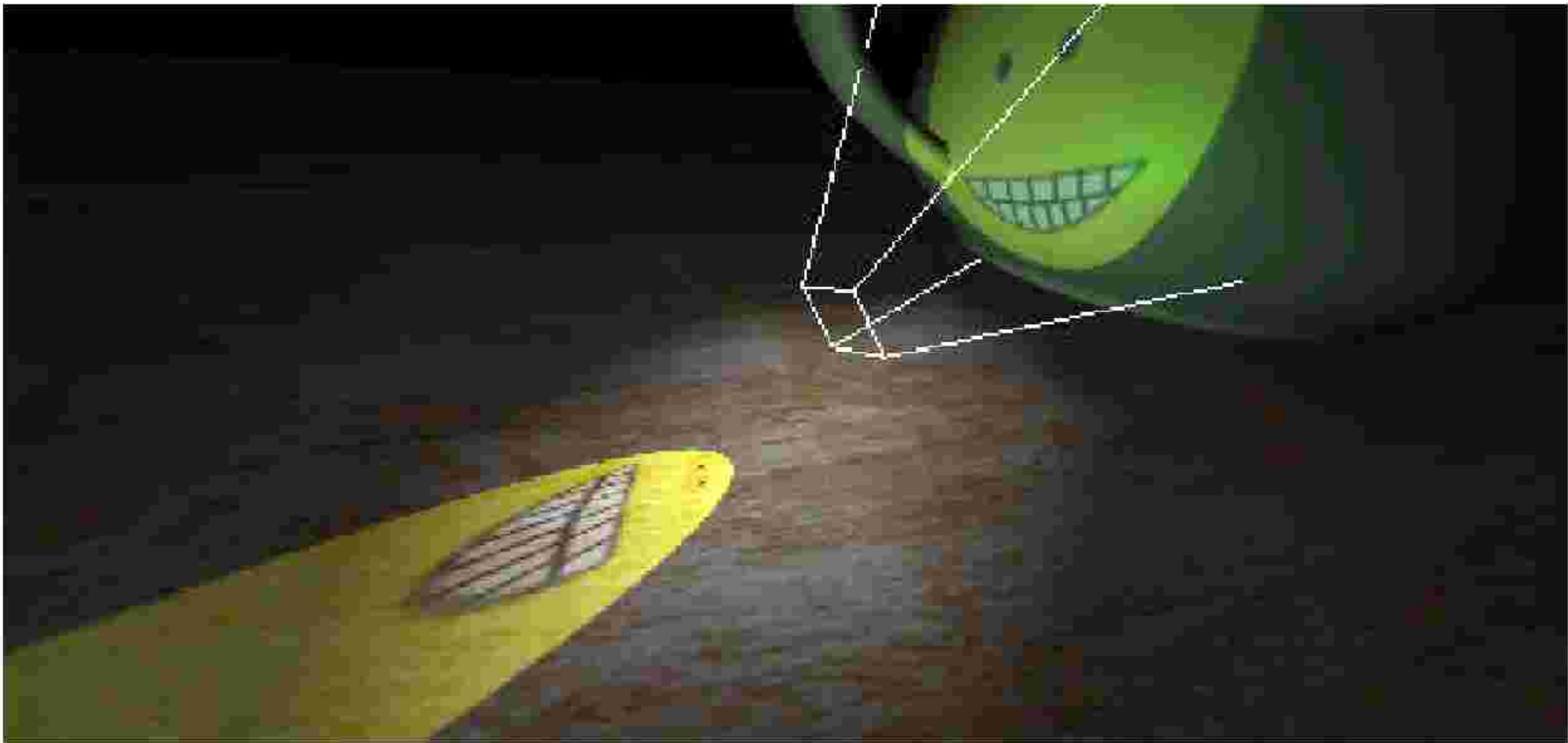
Projective Texturing

- Fragment program does perspective divide
  - (s/q, t/q)

$$\begin{bmatrix} s \\ t \\ r \\ q \end{bmatrix} = \mathbf{T}_o \begin{bmatrix} v_x \\ v_y \\ v_z \\ v_w \end{bmatrix}_{object}$$

Projective Texturing

- Fragment program does perspective divide
  - (s/q, t/q)
  - watch for negative q values!



Projective Texturing

- Watch for negative q values!
  - clip or cull geometry behind light source
  - OR, check for negative q in fragment program
    - then ignore texture if q negative
    - requires branching

Outline

- Visibility
  - Depth peeling
  - Projective texturing
  - Shadow mapping
- Efficient Data Parallel Computing

Shadow Mapping

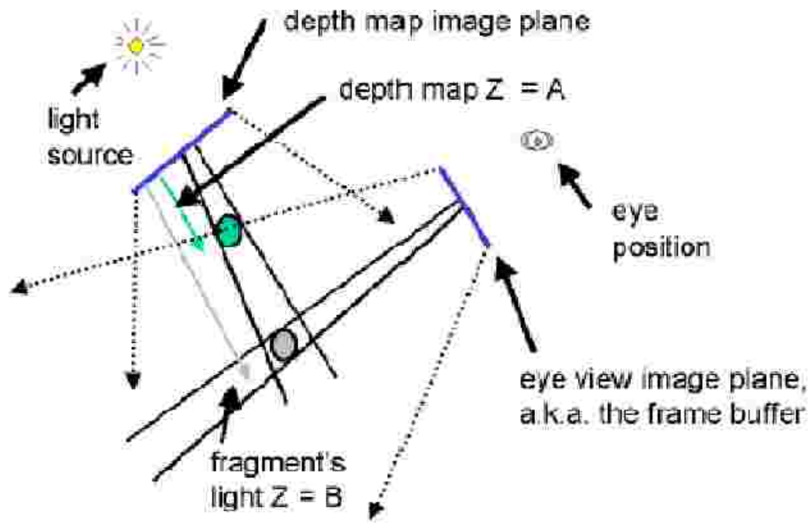
- Multi-pass shadow rendering technique
  - Williams '78
- Pass 1
  - render scene from light POV
  - depth buffer to depth texture (shadow map)
- Pass 2
  - "project" shadow map onto the scene
  - projective texture mapping
  - (x,y,z) from light's reference frame becomes (s,t,r) 3-component texture coordinate

Shadow Mapping

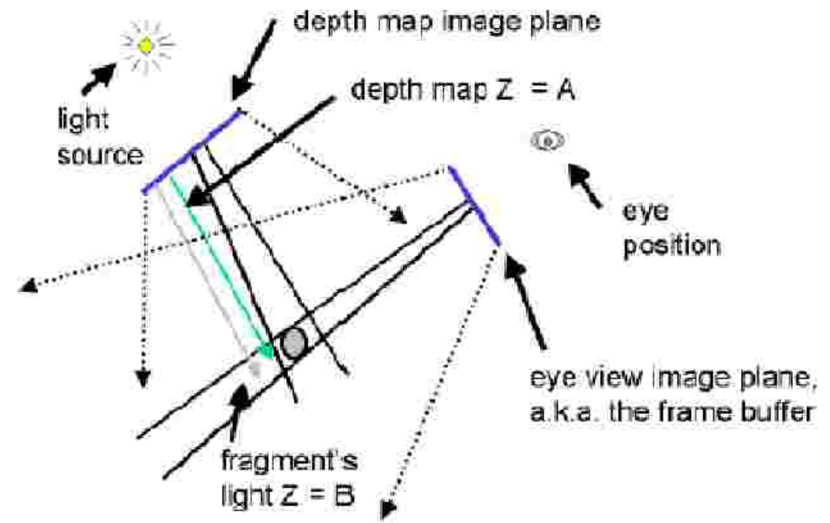
- (x,y,z) from light's reference frame becomes (s,t,r) 3-component texture coordinate

- (s,t) to index into shadow map
- distance stored there is distance to closest surface to light at that pixel
- if r <= lookup(s,t) then fragment not in shadow

The A < B shadowed fragment case



The A ≅ B unshadowed fragment case



Outline

- Visibility
  - Depth peeling
  - Projective texturing
  - Shadow mapping
- Efficient Data Parallel Computing
  - Instruction-Level Parallelism
  - Data-Level Parallelism

A really naïve shader

```
frag2frame Smooth(vert2frag IN, uniform samplerRECT Source : texunit0, uniform samplerRECT Operator : texunit1,
                  uniform samplerRECT Boundary : texunit2, uniform float4 params)
{
    frag2frame OUT;

    float2 center = IN.TexCoord0.xy;
    float4 U = f4texRECT(Source, center);

    // Calculate Red-Black (odd-even) masks
    float2 intpart;
    float2 place = floor(1.0f - modf(round(center + float2(0.5f, 0.5f)) / 2.0f, intpart));
    float2 mask = float2((1.0f-place.x) * (1.0f-place.y), place.x * place.y);

    if (((mask.x + mask.y) && params.y) || (!(mask.x + mask.y) && !params.y))
    {
        float2 offset = float2(params.x*center.x - 0.5f*(params.x-1.0f), params.x*center.y - 0.5f*(params.x-1.0f));
        ...
        float4 neighbor = float4(center.x - 1.0f, center.x + 1.0f, center.y - 1.0f, center.y + 1.0f);
        float central = -2.0f*(O.x + O.y);

        float poisson = ((params.x*params.x)*U.z + (-O.x * f1texRECT(Source, float2(neighbor.x, center.y)) +
                                                    -O.x * f1texRECT(Source, float2(neighbor.y, center.y)) +
                                                    -O.y * f1texRECT(Source, float2(center.x, neighbor.z)) +
                                                    -O.z * f1texRECT(Source, float2(center.x, neighbor.w)))) / O.w;
        OUT.COL.x = poisson;
    }
    ...
    return OUT;
}
```

Credit: Cliff Woolley

Instruction-Level Parallelism

```
float2 offset = float2(params.x*center.x - 0.5f*(params.x-1.0f),
                       params.x*center.y - 0.5f*(params.x-1.0f));

float4 neighbor = float4(center.x - 1.0f,
                         center.x + 1.0f,
                         center.y - 1.0f,
                         center.y + 1.0f);
```

Credit: Cliff Woolley

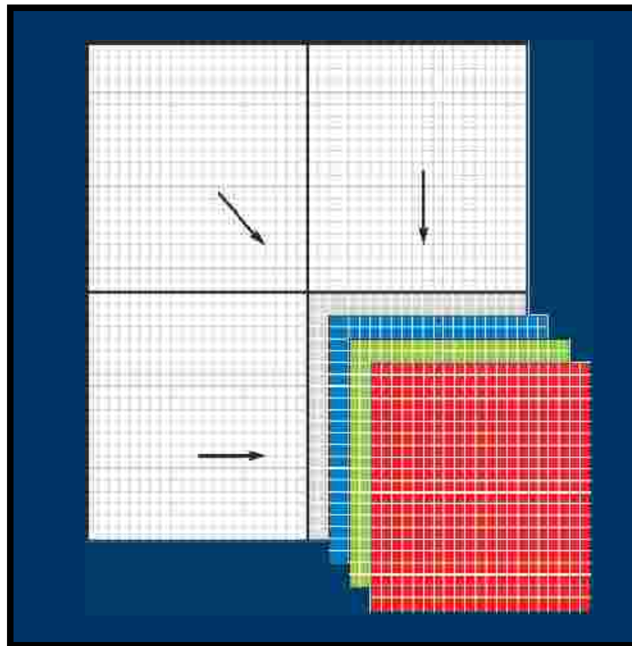
Instruction-Level Parallelism

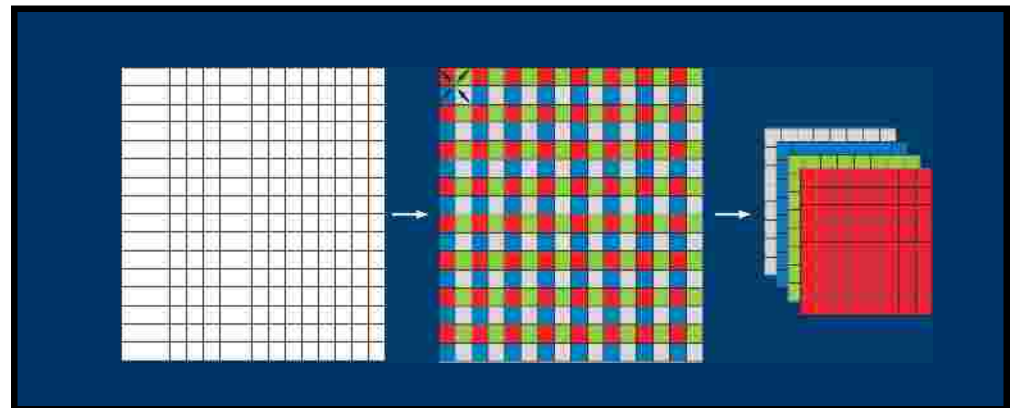
```
float2 offset = center.xy - 0.5f;
offset = offset * params.xx + 0.5f;  // MADR is cool too – one
   cycle, two flops

float4 neighbor = center.xxyy + float4(-1.0f,1.0f,-1.0f,1.0f);
```

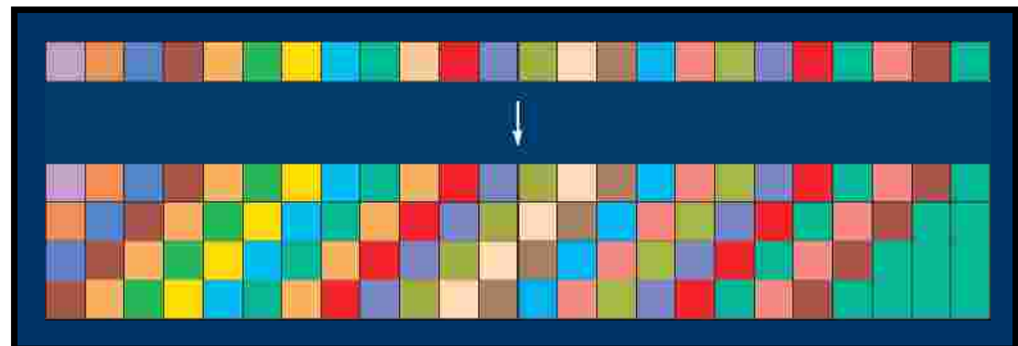
Credit: Cliff Woolley

Data-Level Parallelism

- Pack scalar data into RGBA in texture memory







Credit: Cliff Woolley

// Acknowledgements

- Cass Everitt
- Mark Kilgard
- Youngung Shon
- Cliff Woolley

Paper for next class

- "Blister:GPU-based rendering of Boolean combinations of free-form triangulated shapes"
- John Hable, Jarek Rossignac