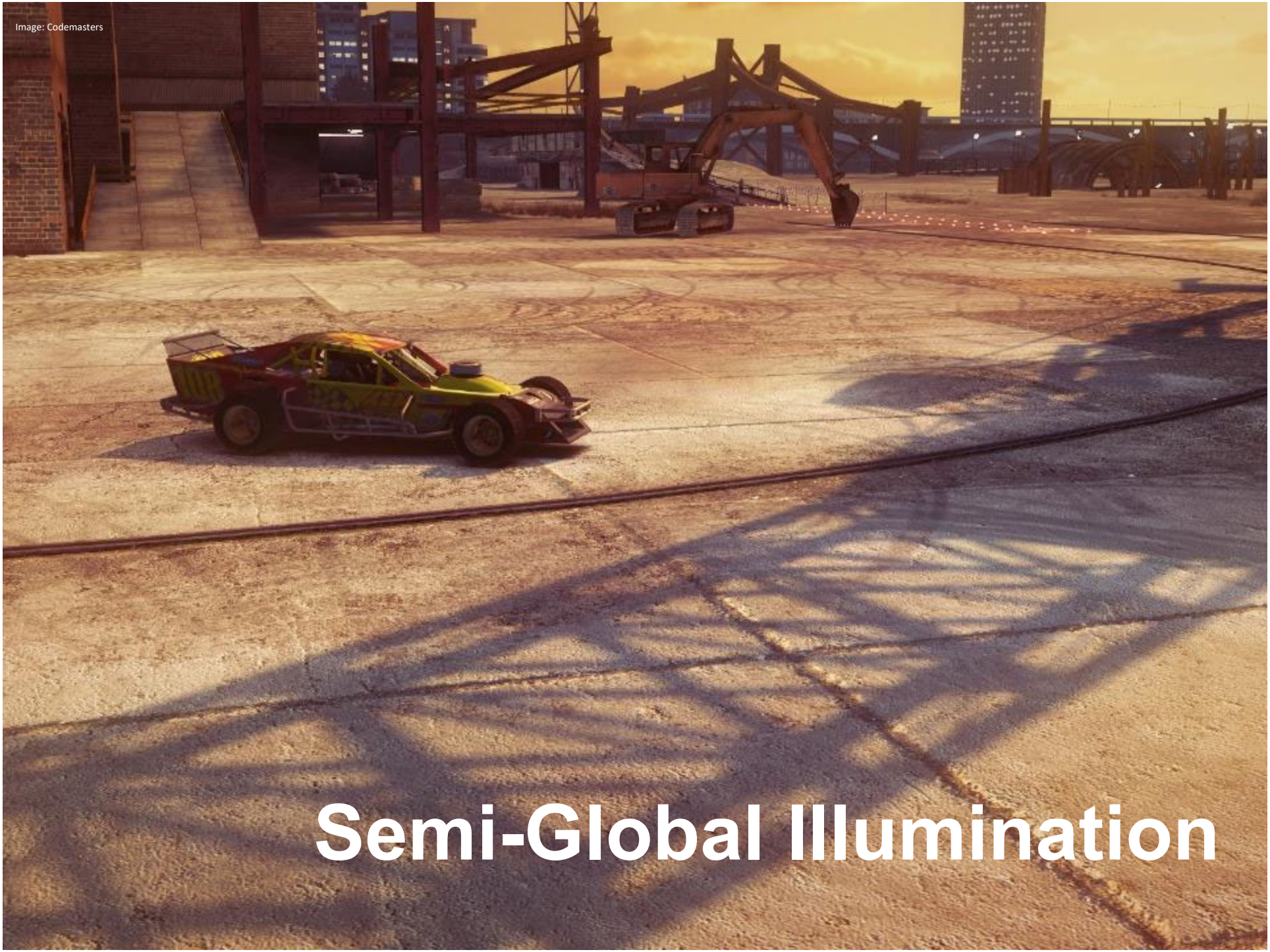
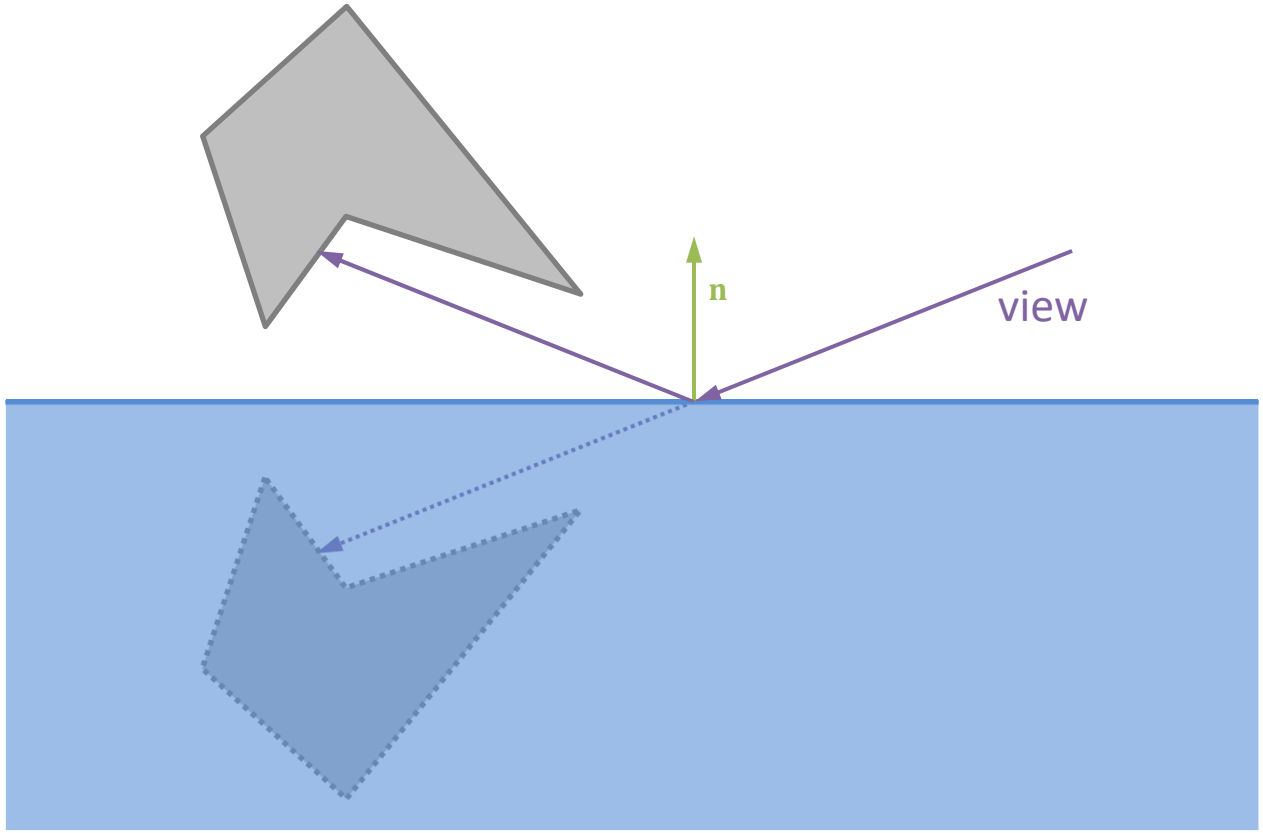


Image: Codemasters
Semi-Global Illumination

Introduction

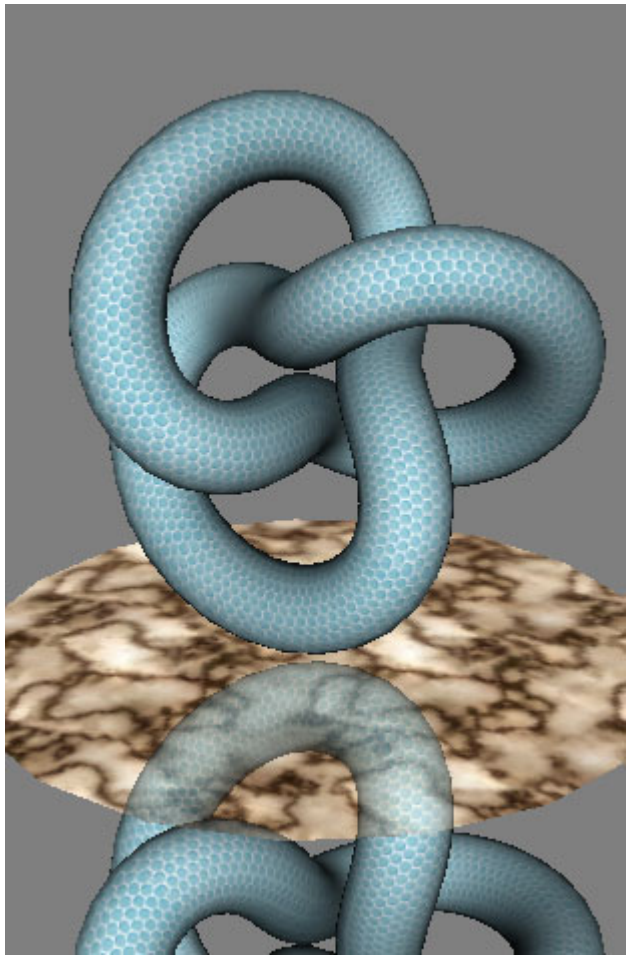
- Full global illumination is expensive
- Many subtle lighting effects may not be visible
- For real time rendering
  - Compute local illumination *everywhere*
  - Compute global illumination only *selectively*
    - Only certain types of light transport
    - Only for objects where it is visually important
- Important categories: reflections, transparency, shadows, ambient occlusion, precomputed radiance

Planar Reflections

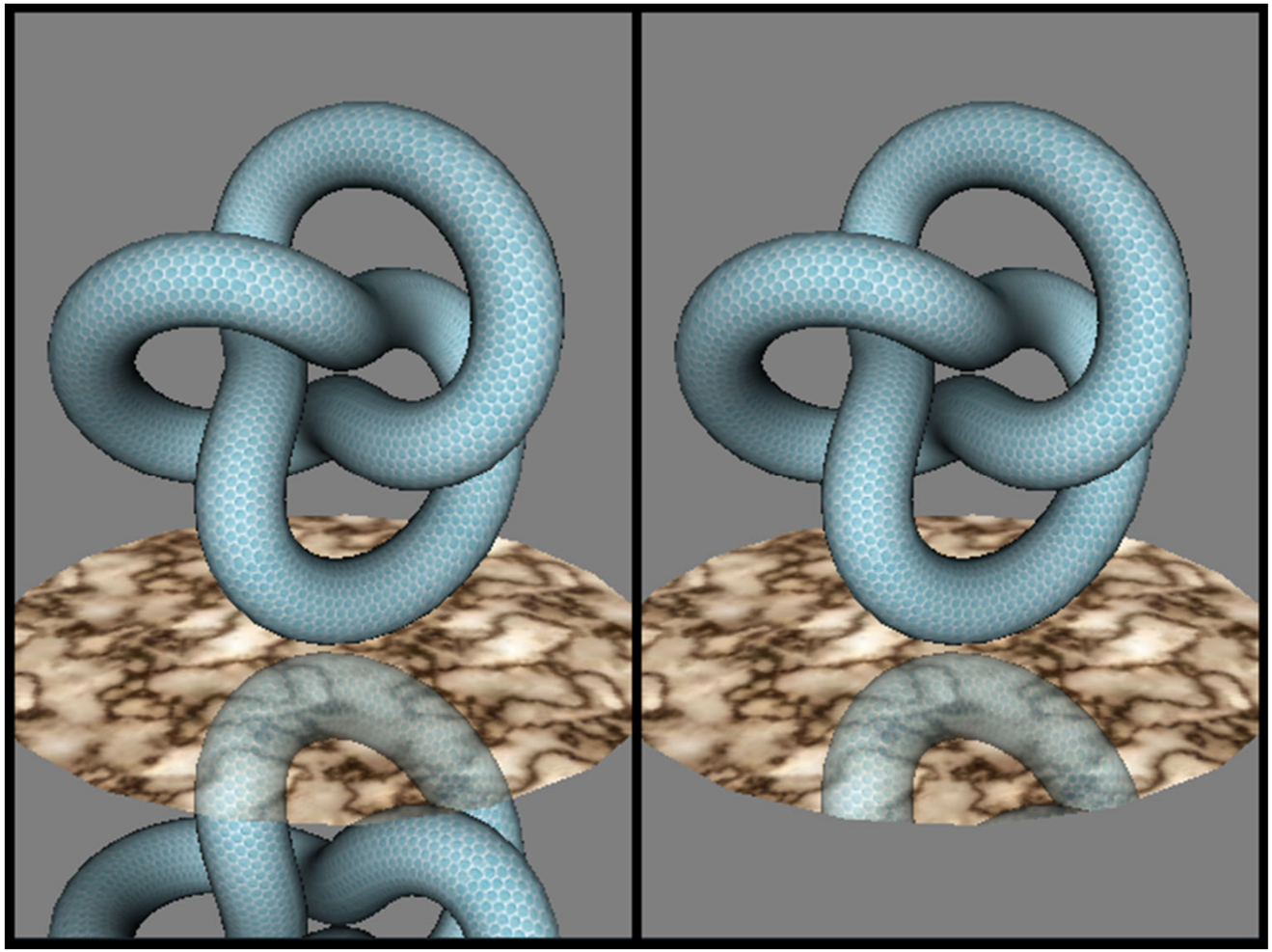


Planar Reflections in OpenGL

- Define a plane, e.g., z-plane
  - Scale(1, 1,-1)
- Adapt culling to mirroring
  - Cull front faces
  - Do *not* cull back faces
- Draw the object (mirrored)
- Draw the mirror (transparent)
- Draw the object (normal)

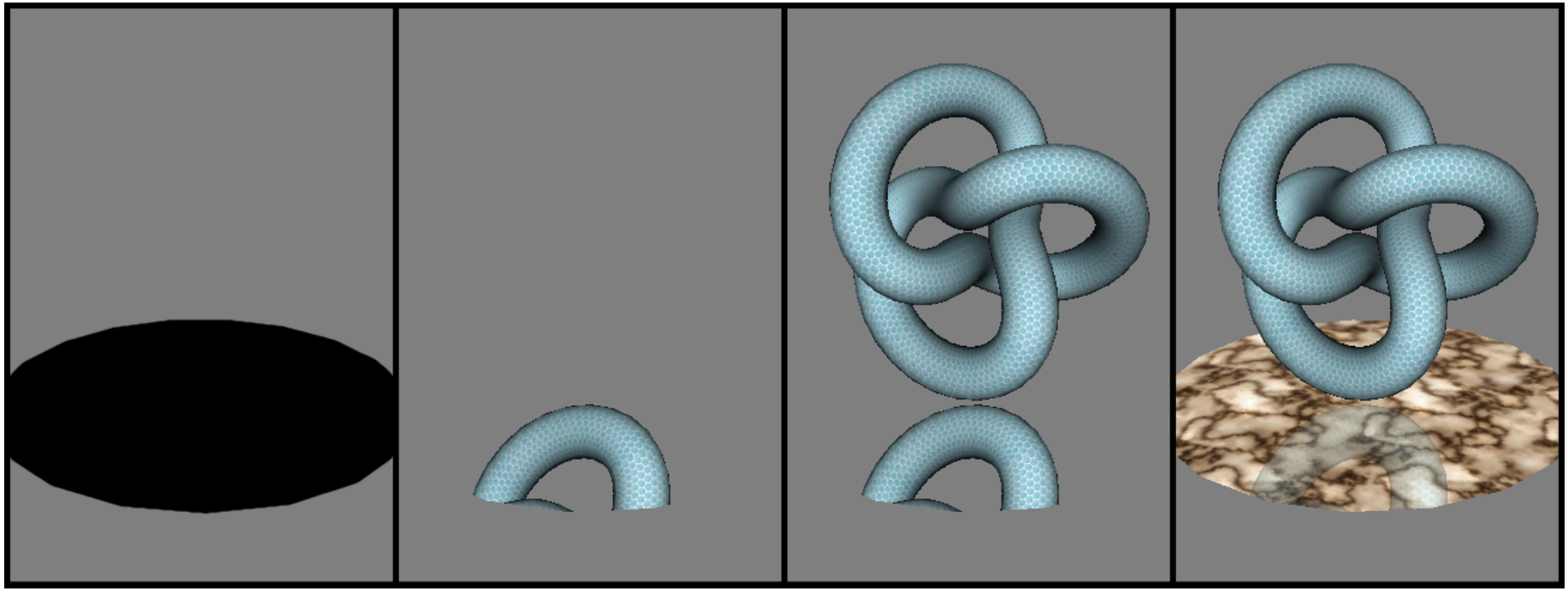


Problem



Object not restricted to mirror Solution: Stencil buffer

Reflection with Stencil Buffering



Stencil mask

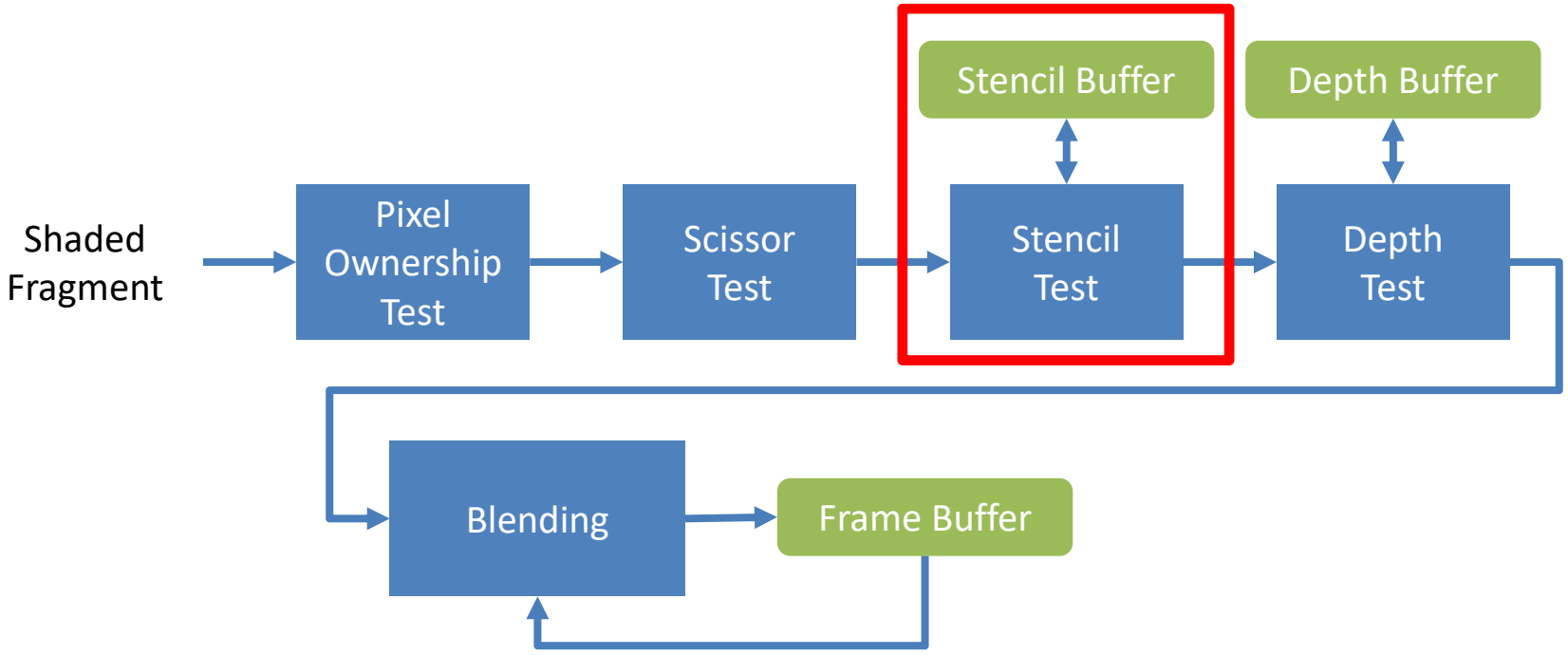
Reflected object clipped to stencil

Reflection + object

Result

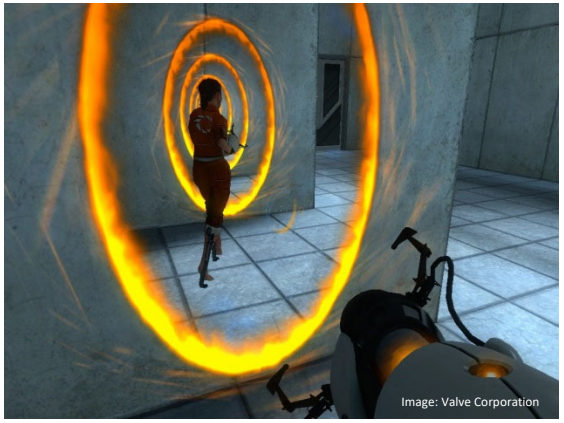
Fixed-Function Raster Operations

- When does the stencil test happen?

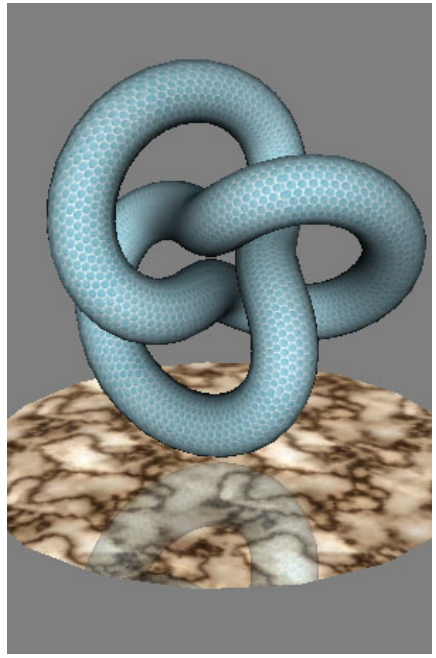


Stencil Buffer

- Allows to mask parts of the framebuffer
- 1 Bit Stencil Buffer ($\in \{0,1\}$)
- 8 Bit Stencil Buffer (256 states)
- Applications
  - Reflections
  - CSG (constructive solid geometry)
  - Shadow volumes
- Today:
  - Most applications do stenciling in shader code
  - Hardware stencil still very fast – example: Portal



Stencil Buffer Reflections



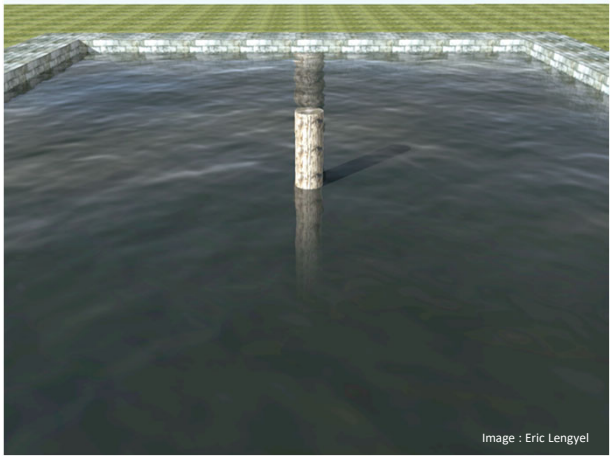
- Reset stencil buffer
  - Clear stencil buffer to zero
- Render mirror into stencil buffer
  - Enable stencil test
  - Set stencil operation to always write (on stencil-fail, depth-fail, depth-pass)
- Render mirrored object only in mirror
  - Set stencil operation to write if stencil bit set
- Render object normally

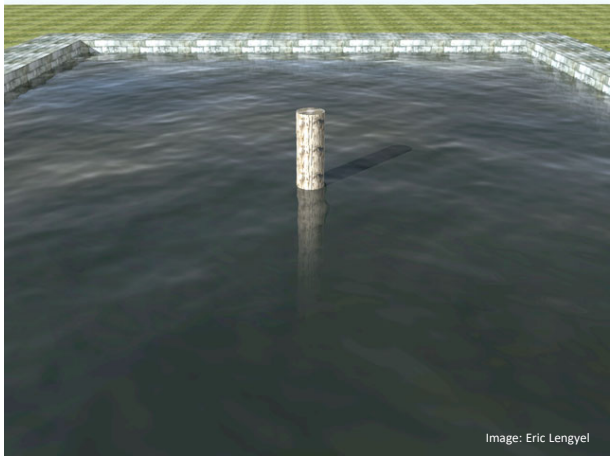
Advanced Reflections with Shaders

- Fade object color with increasing distance to reflector
- Fuzzy reflections: compute distorted texture coordinates

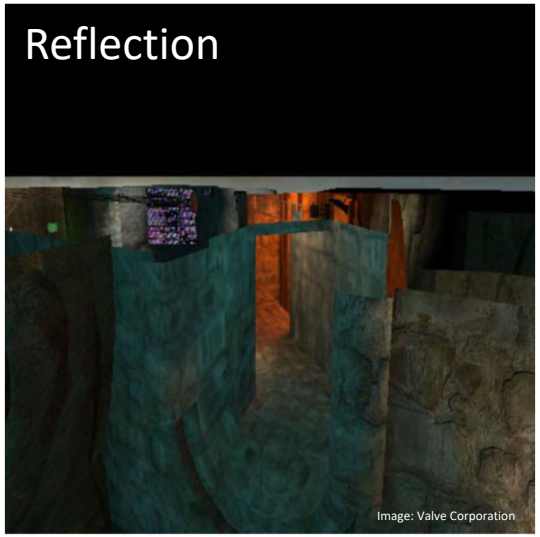
Caveats

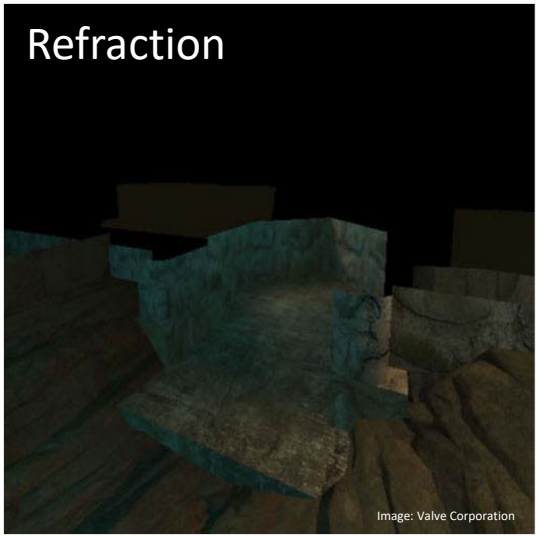
- Reflection changes handedness
  - Adjust winding order of front faces when rendering reflected geometry
- If scene intersects mirror plane
  - Need to clip against plane to avoid artifacts





Shader-Based Approach





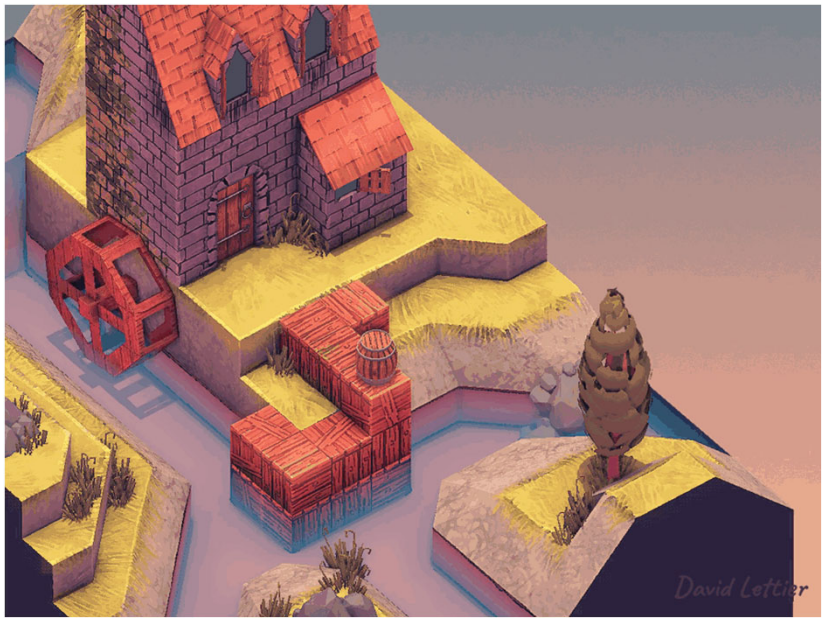
- Can also do refractions using shaders
- Render parts of the scene below the plane into additional texture
- Use fog to simulate scattering in water

Screen Space Reflections

Picture: Killing Floor 2 (Tripwire Interactive)

Screen Space Reflections (SSR)

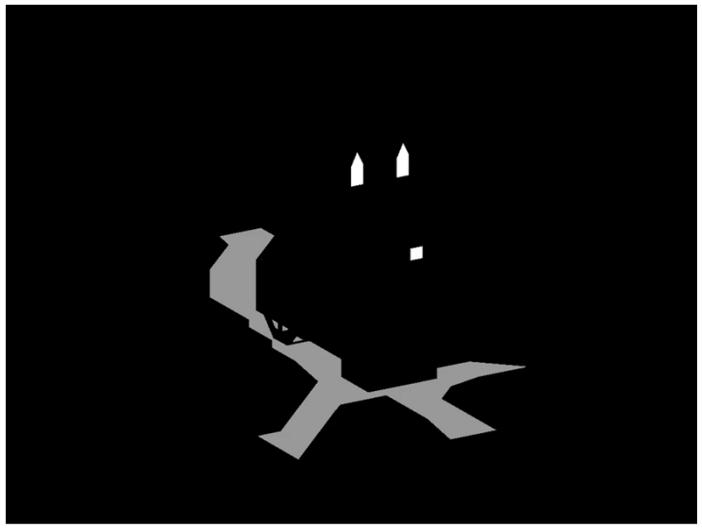
- Fully dynamic raytraced reflections are very expensive (even with RTX)
- Idea:
  - Reflect view ray across normal
  - Find depth buffer intersection in screen space
- Result: cheap dynamic approximation of reflections



https://lettier.github.io/3d-game-shaders-for-beginners/screen-space-reflection.html

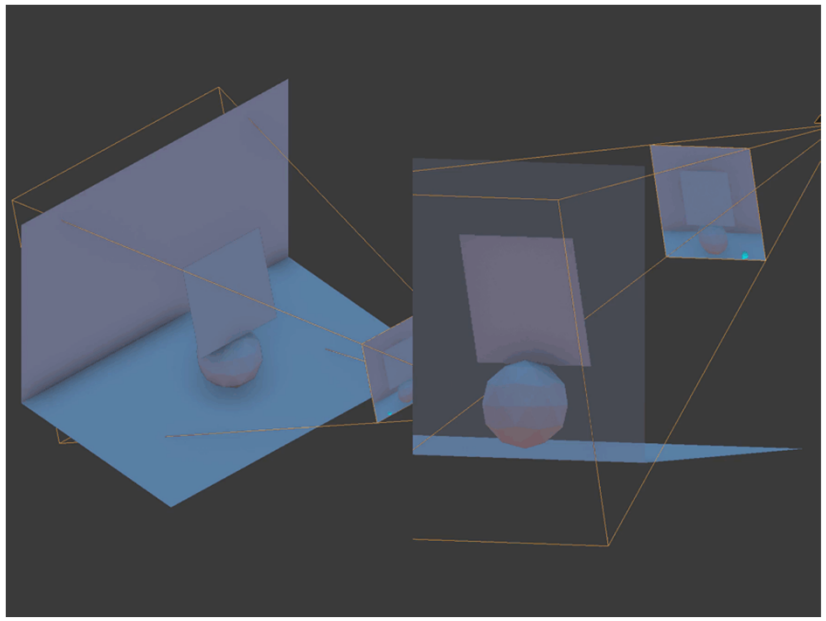
SSR: Reflection Map

- Compute a reflection map
  - Controls amount of reflection per fragment in screen space, depending on material
- Black pixels do not show SSR
- White pixels show full SSR



SSR: View Ray

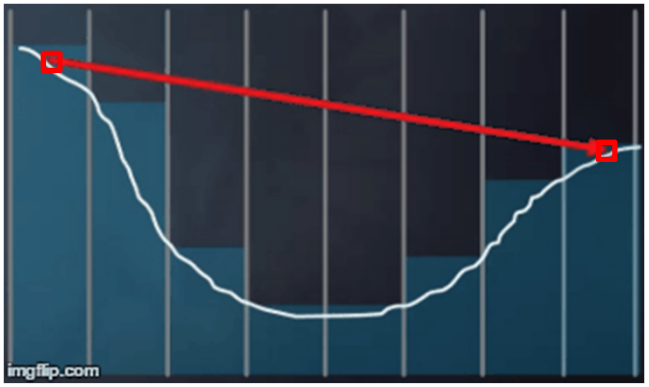
- For each non-zero pixel in reflection map, reflect view ray across surface normal
- Along reflection vector
  - Choose a point
  - Project it
  - Compute reflection vector in screen space



https://lettier.github.io/3d-game-shaders-for-beginners/screen-space-reflection.html

SSR: Ray Marching

- From initial screen space position, **ray march** towards reflected screen space position
- If ray **intersects depth buffer**, we found our reflection point



https://sakibsaikia.github.io/graphics/2016/12/26/Screen-Space-Reflection-in-Killing-Floor-2.html

SSR: Ray Marching Code

- From the initial screen space position, we then **ray march** along towards the reflected screen space position
- If the ray **intersects the depth buffer**, we found our reflection point

```
#define MAX_REFLECTION_RAY_MARCH_STEP 0.02f
#define NUM_RAY_MARCH_SAMPLES 16

bool GetReflection(
    vec3 ScreenSpaceReflectionVec,
    vec3 ScreenSpacePos,
    out vec3 ReflectionColor)
{
    // Raymarch in the direction of the ScreenSpaceReflectionVec until
    // you get an intersection with your z buffer
    for (int RayStepIdx = 0; RayStepIdx<NUM_RAY_MARCH_SAMPLES; RayStepIdx++)
    {
        vec3 RaySample = (RayStepIdx * ReflectionRayMarchStep)
                       * ScreenSpaceReflectionVec + ScreenSpacePos;

        float ZBufferVal = texture(DepthSampler, RaySample.xy).r;

        if (RaySample.z > ZBufferVal)
        {
            ReflectionColor = texture(SceneColorSampler, RaySample.xy).rgb;
            return true;
        }
    }

    return false;
}
```

https://sakibsaikia.github.io/graphics/2016/12/26/Screen-Space-Reflection-in-Killing-Floor-2.html

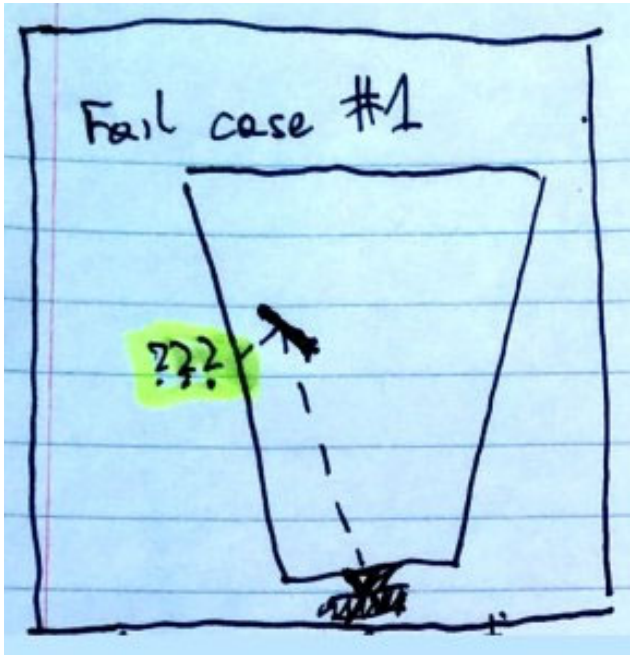
SSR: Intersection Code

- Fixed step size can lead to inaccurate results
- Use binary search between points just before and after depth intersection

```
#define NUM_BINARY_SEARCH_SAMPLES 6
// PrevRaySample: Sample just before the depth buffer intersection
// RaySample: Sample just after the depth buffer intersection
// DepthSampler: sampler2D to access the rendered depth texture

if (bFoundIntersection)
{
    vec4 MinRaySample = PrevRaySample;
    vec4 MaxRaySample = RaySample;
    vec4 MidRaySample;

    for (int i = 0; i < NUM_BINARY_SEARCH_SAMPLES; i++)
    {
        MidRaySample = mix(MinRaySample, MaxRaySample, 0.5);
        float ZBufferVal = texture(DepthSampler, MidRaySample.xy).r;

        if (MidSample.z > ZBufferVal)
            MaxRaySample = MidSample;
        else
            MinRaySample = MidSample;
    }
}
```

SSR: Limitations 1

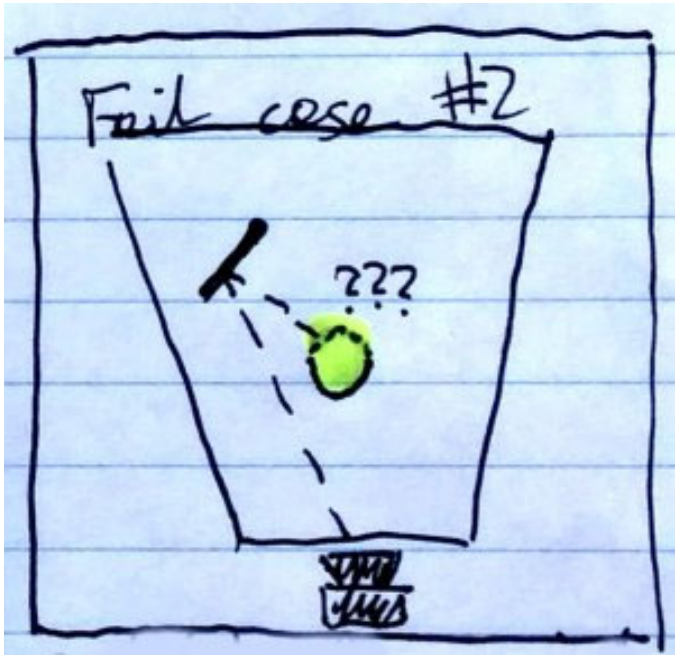
- Viewer facing reflections
- Reflections outside viewport



https://bartwronski.com/2014/01/25/the-future-of-screenspace-reflections/

SSR: Limitations 2

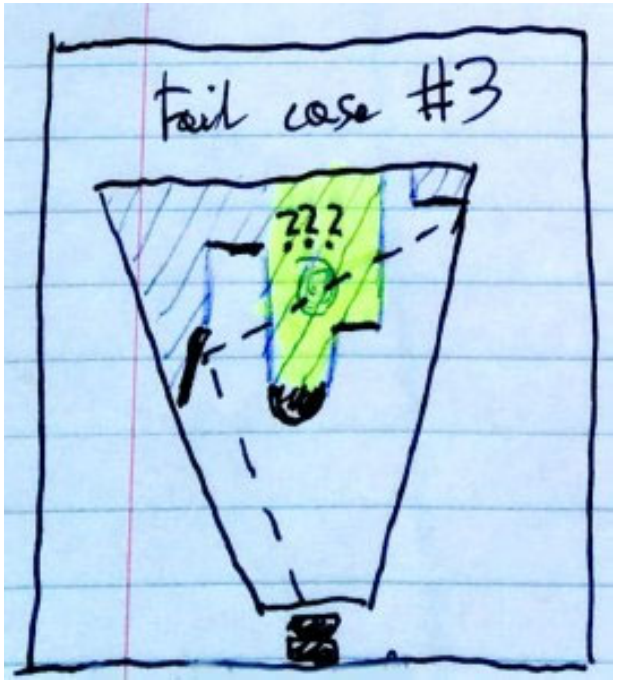
- Reflections of back faces



https://bartwronski.com/2014/01/25/the-future-of-screenspace-reflections/

SSR: Limitations 3

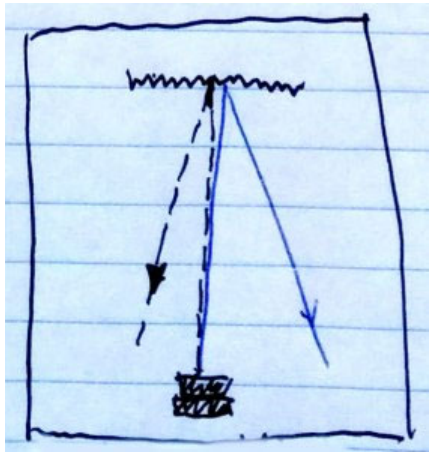
- Depth complexity

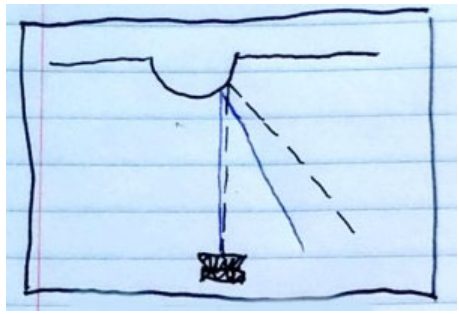


https://bartwronski.com/2014/01/25/the-future-of-screenspace-reflections/

SSR: Limitations 4

- Flickering
- Holes
- Temporal artifacts





https://bartwronski.com/2014/01/25/the-future-of-screenspace-reflections/

SSR: Further Improvements

- Blend with cubemaps or baked reflections
- Use temporal filtering to smoothen results
  - TAA is often exploited for this
- Use spatial filtering or blurring for rough surfaces
- Limit ray range in world space
  - For efficiency
  - To prevent far-away objects from flickering

Transparency

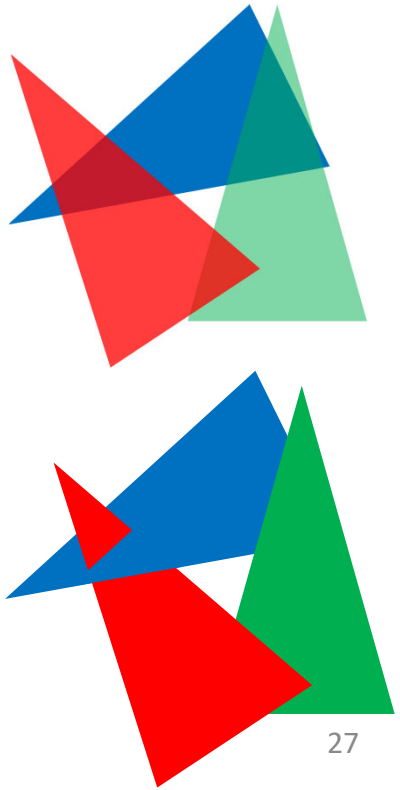
- Transparency blending is order dependent
- Blending must be back to front
- Need to sort scene elements by depth

Methods for Sorting Fragments

- Explicit depth sorting
  - Writed sorted order into draw buffer
- Depth peeling
  - Multi-pass rendering, once per *depth layer*
- Deep frame buffer
  - Write per-pixel linked lists
- Weighted blended order-independent rendering
  - Use depth as implicit weight (heuristic)

Explicit Depth Sorting

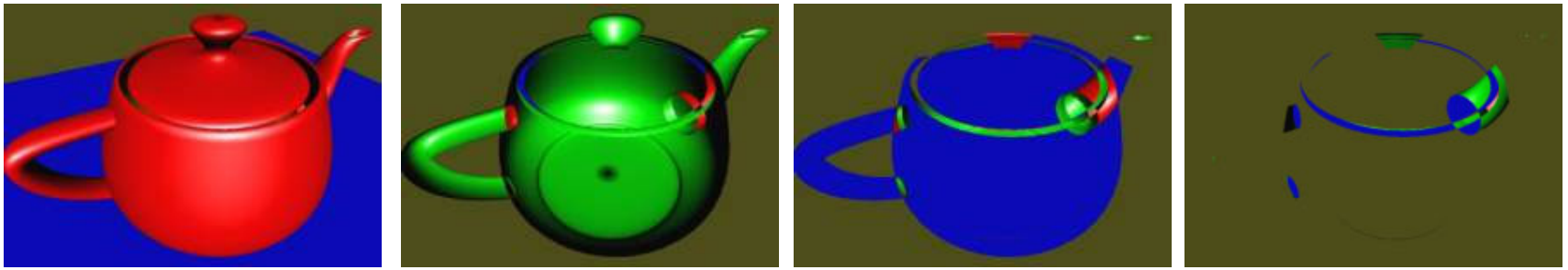
- Explicit depth sorting into draw buffer
- (+) Rendering with standard GPU primitive order
- (-) Writing sorted order to memory is expensive
- (-) Not always good enough
  - Cyclic overlap
  - Intersecting triangles
  - Can either split triangles (ugly, brittle), or
  - Can sort per fragments instead



Depth Peeling

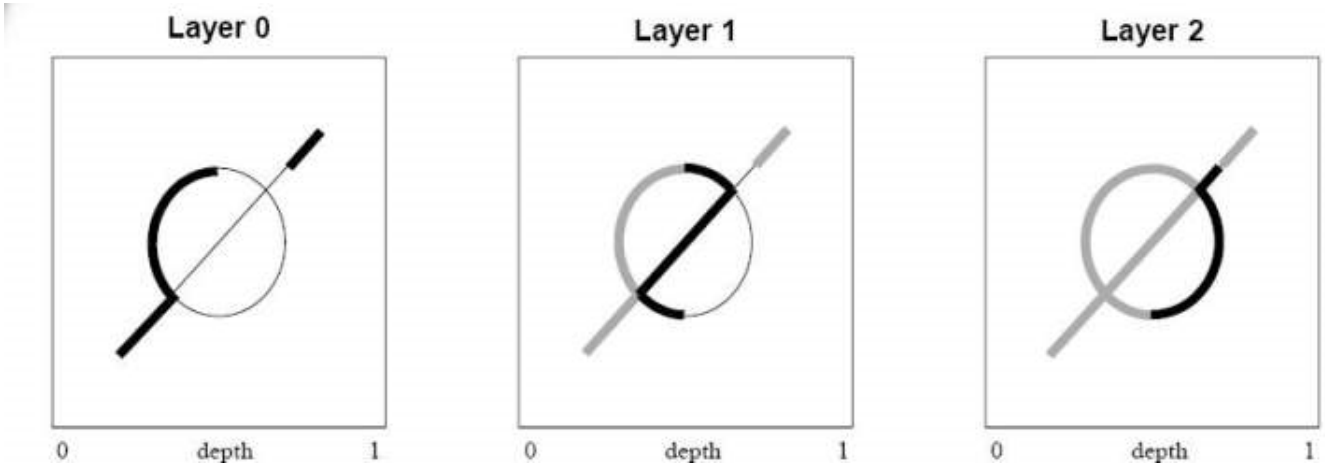
Interactive order independent transparency [Everitt'01]

- Multiple passes
  - First pass - find the front-most depth/color
  - Each successive pass - finds the depth/color for the next nearest fragment on a per-pixel basis
- Compare previous layer and current layer



Rendering Depth Peeling

- Render the scene *n* times
- Save the results in texture using render-to-texture/FBO
- Each pass forms a layer
- Starting from the second pass, use a shader for comparison with previous layer
- Compose *n* layers

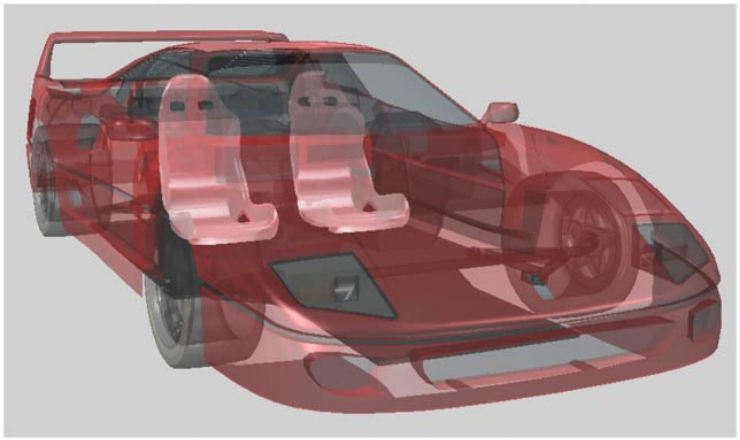


Dual Depth Peeling

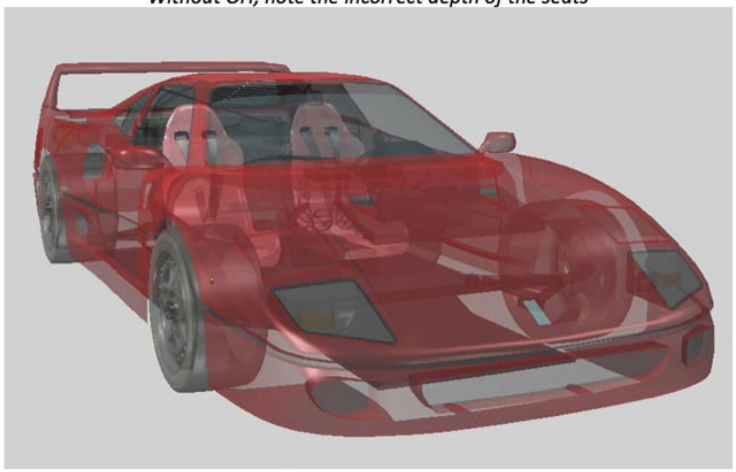
- Peel front and back in one pass
- Keep track of min/max depth in two textures
- Compare current depth with min/max values
  - If < min write to front layer
  - If > max write to back layer
- Use previous front/back layer depth values in current pass
- Decreases number of passes from *n* to *n*/2+1

Deep Framebuffer

- Depth peeling creates too much geometry overhead
  - O(triangles*overdraw)
- Use pixel operations instead
  - Write every fragment to a per-pixel linked list
- Sort and blend the lists
- Historically called *accumulation buffer* (*A-buffer*)



Without OIT, note the incorrect depth of the seats



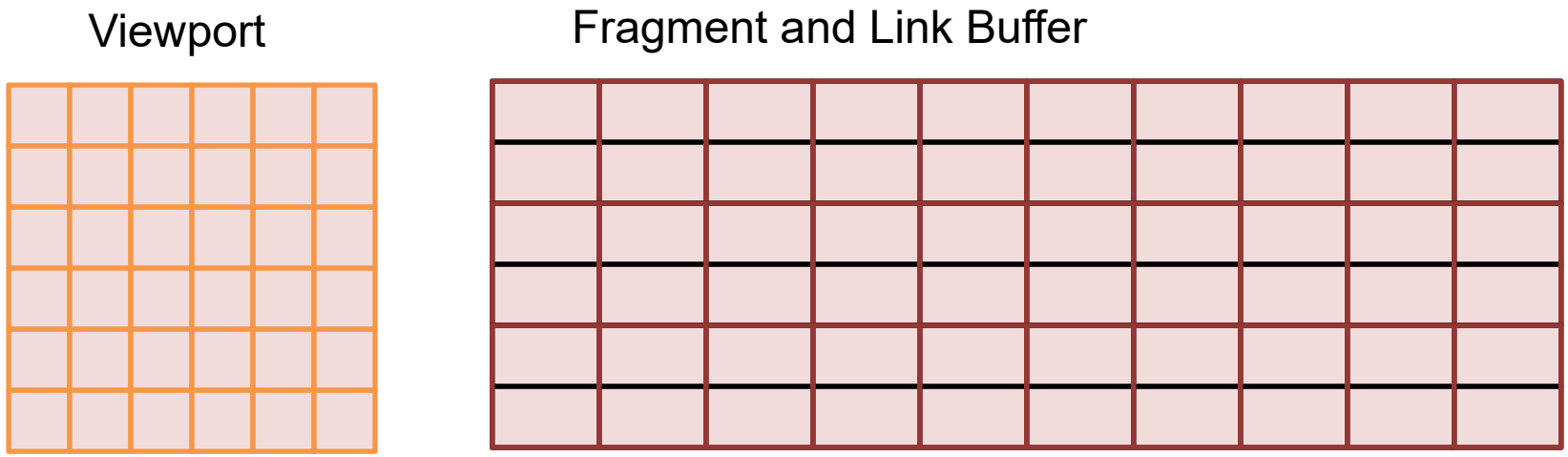
With OIT applied

Per-Pixel Linked Lists

- Algorithm
  - Render scene and generate linked list per pixel
  - Sort list and compose fragments
- Requires read/write buffer
- Need two additional buffers
  - Fragment and link buffer
  - Start offset buffer

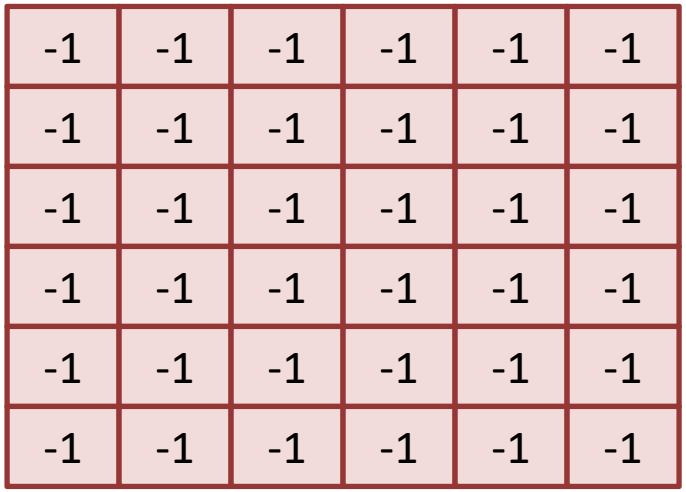
Fragment and Link Buffer

- Contains all fragment data produced during rasterization
- Must be large enough to store all fragments



Start Offset Buffer

- Contains the offset of the last fragment written at every pixel location
- Screen-sized: `(width * height * sizeof(UINT32))`
- Initialized to magic value
  - Magic value indicates end of list (no more fragments stored)



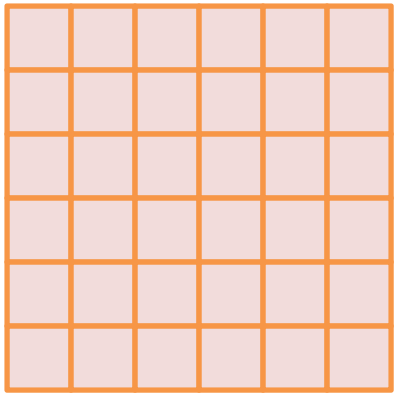
Start Offset Buffer

Linked List Creation 1

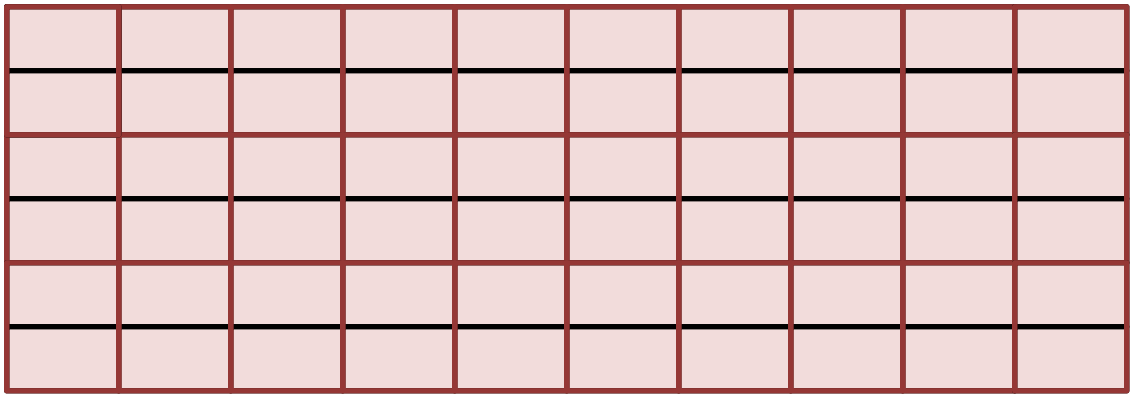
Start Offset Buffer

-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1

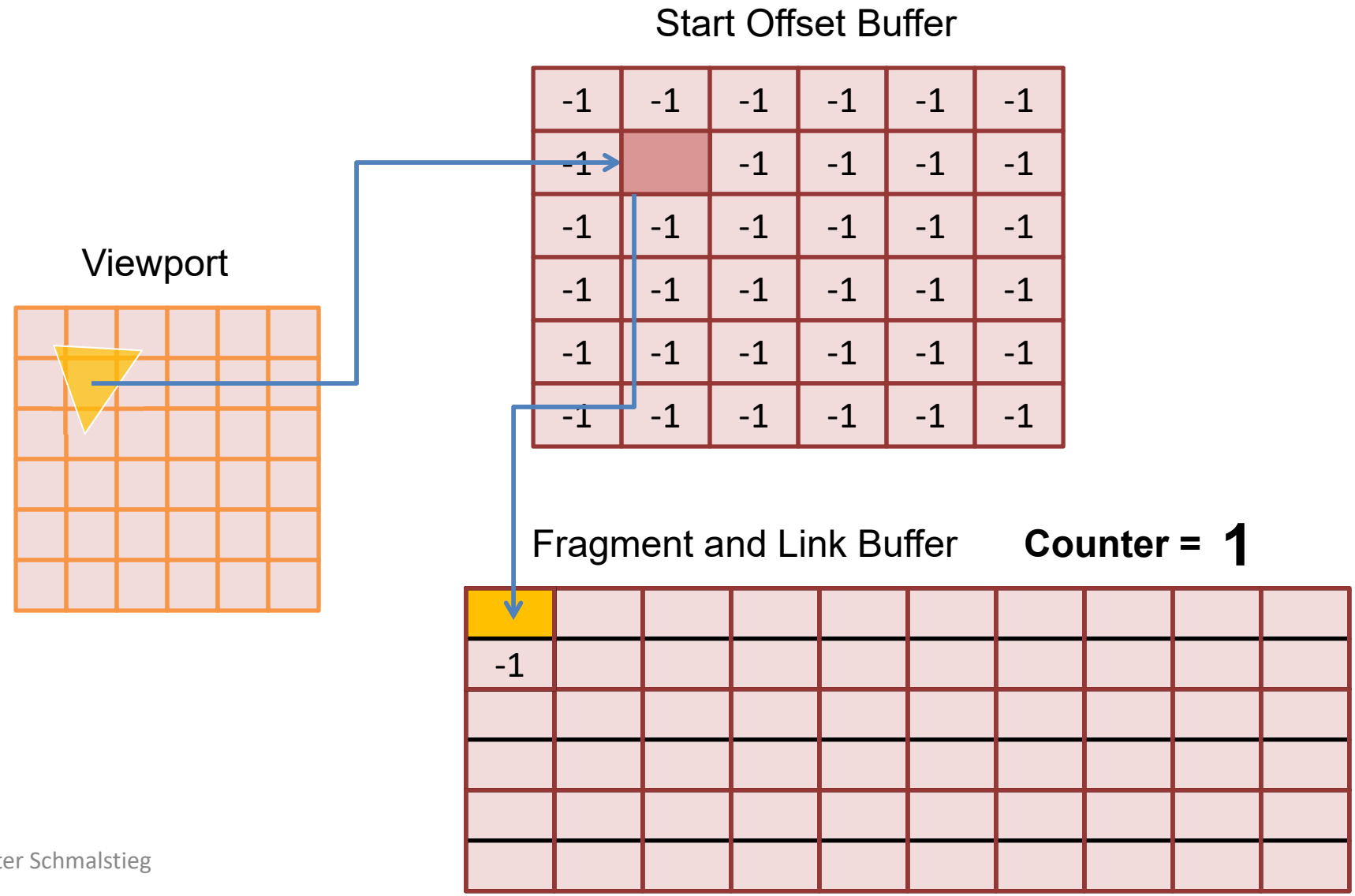
Viewport



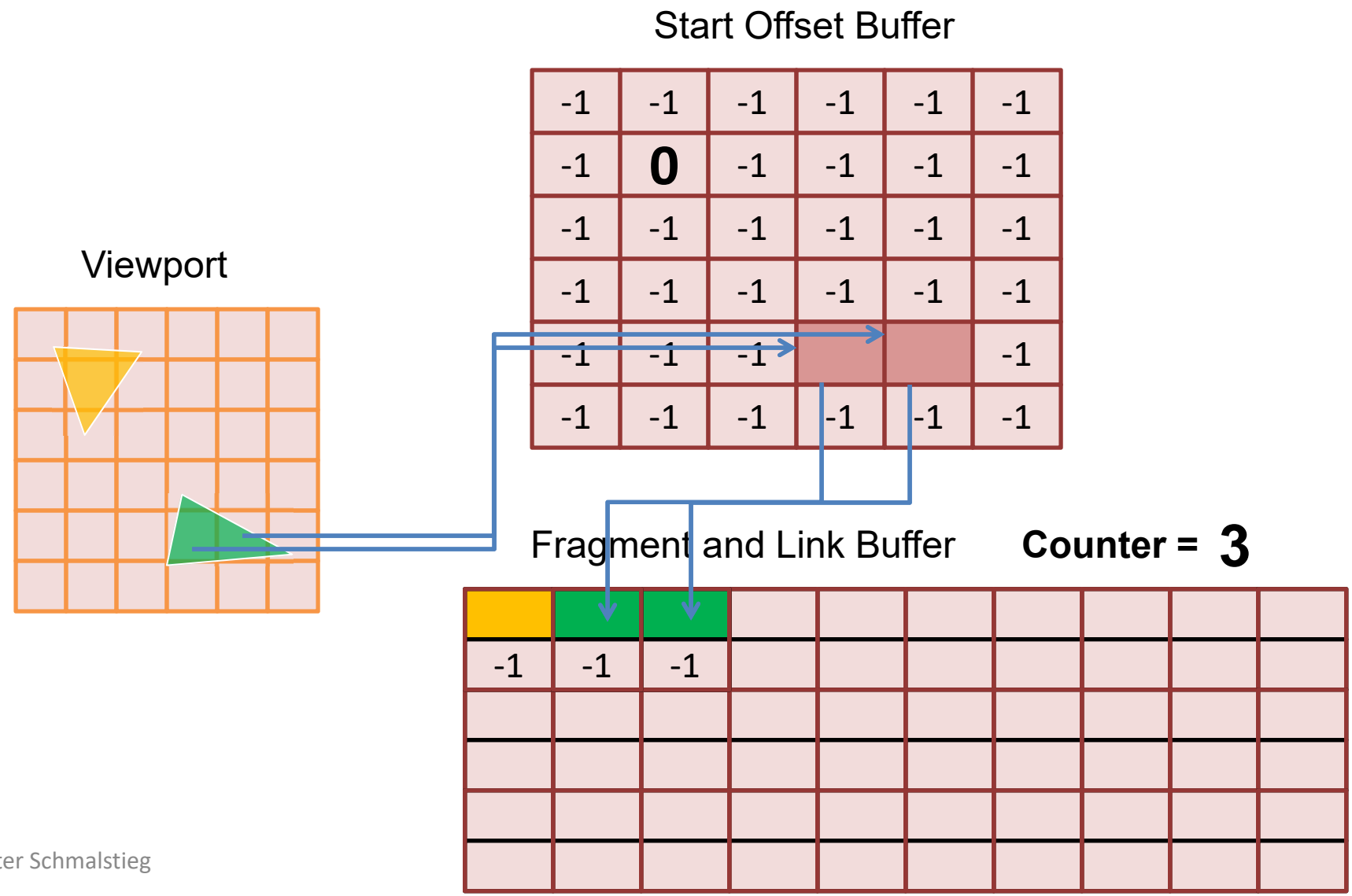
Fragment and Link Buffer **Counter = 1**



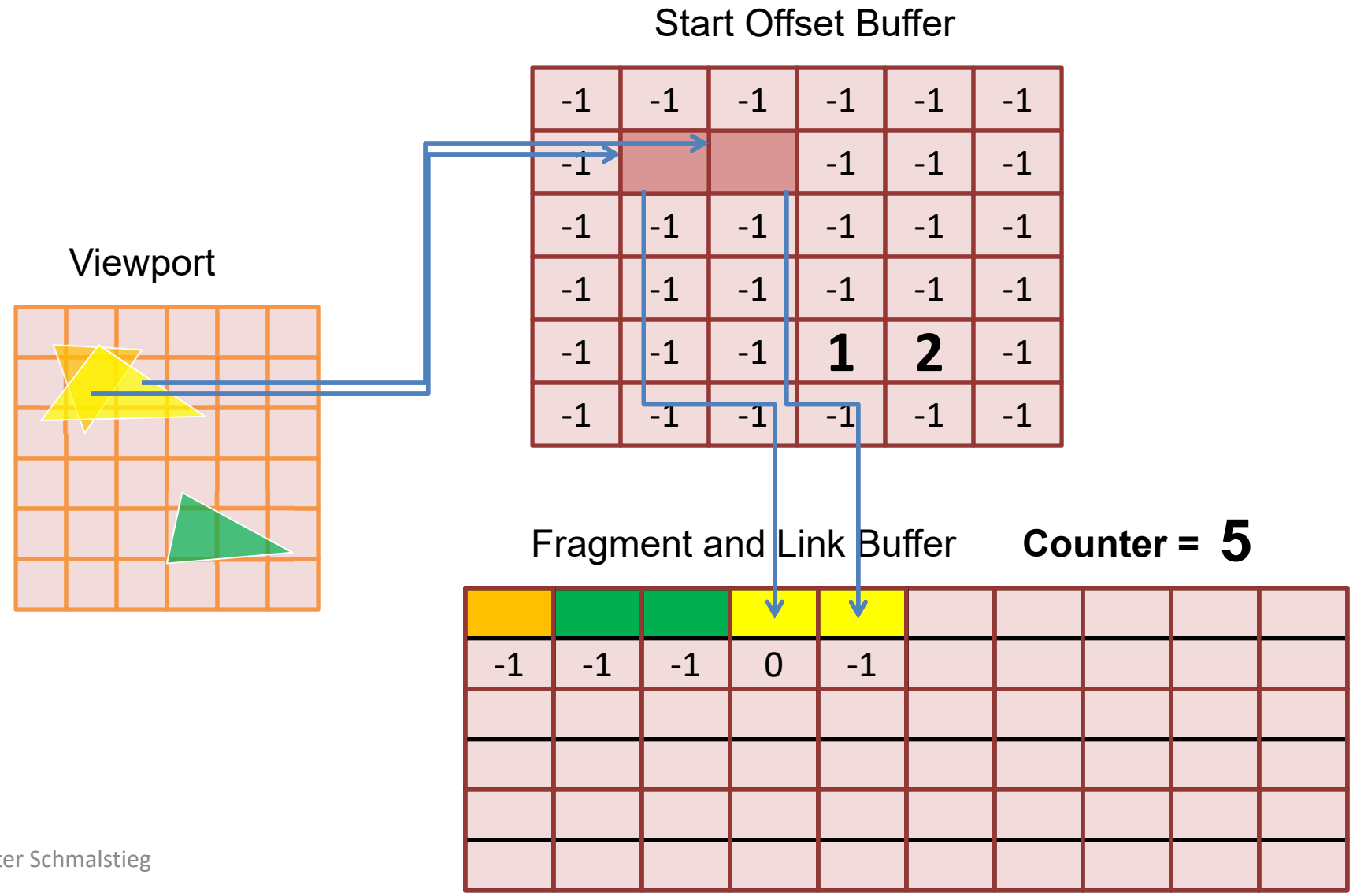
Linked List Creation 2



Linked List Creation 3



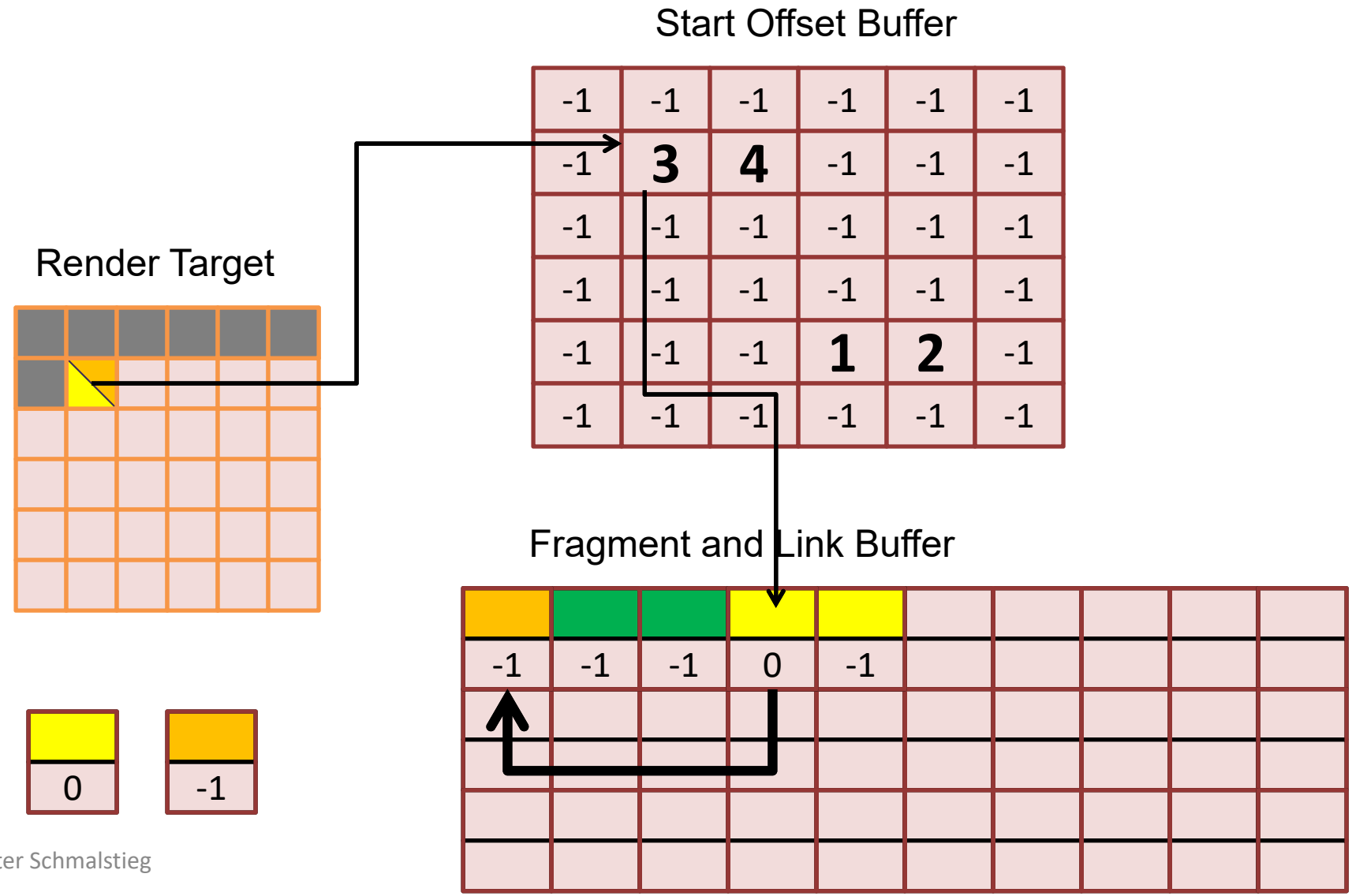
Linked List Creation 4



Linked List Traversal

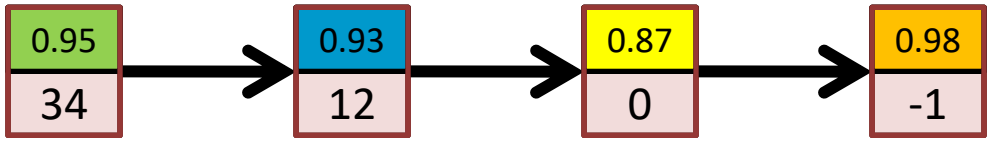
- Render a fullscreen quad (or compute shader)
- For each pixel
  - Parse the linked list
  - Retrieve fragments for this screen position
- Process list of fragments (Sort and Blend)

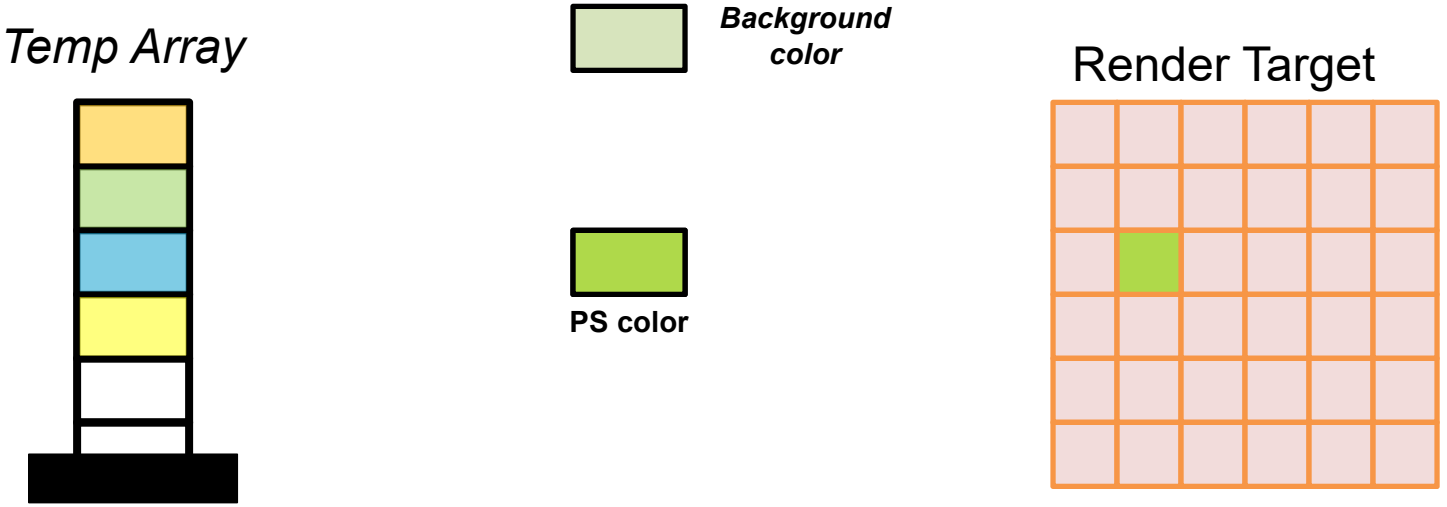
Rendering from Linked List



Sorting and Blending

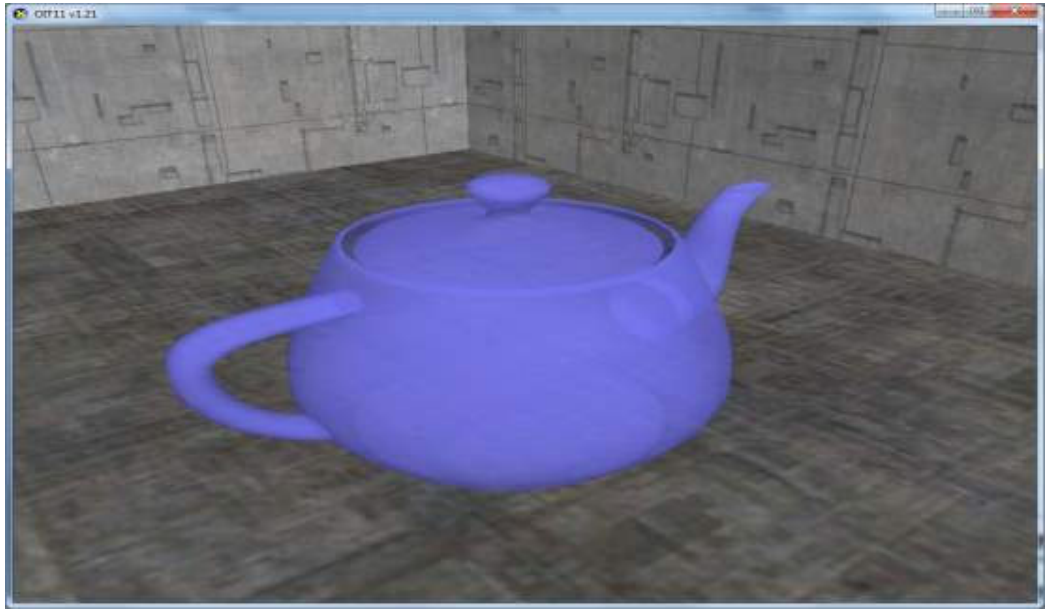
- Blend fragments back to front in PS
- Must sort by depth value in temp.array





Performance Comparison

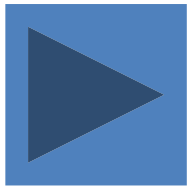
	Teapot	Dragon
Linked List	743 fps	338 fps
Depth Peeling	579 fps	45 fps
Dual Depth Peeling	---	94 fps



Result

- 602K scene triangles
  - 254K transparent triangles

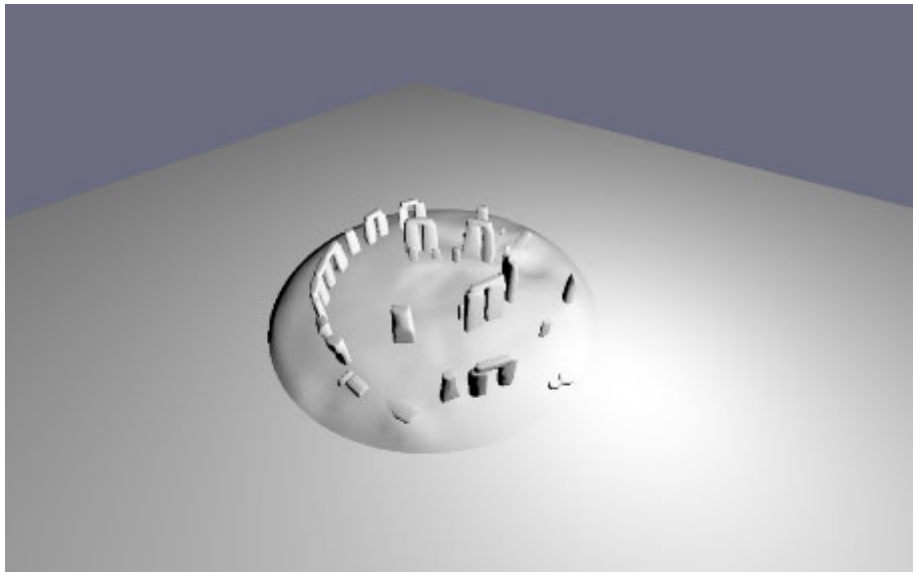
VIDEO

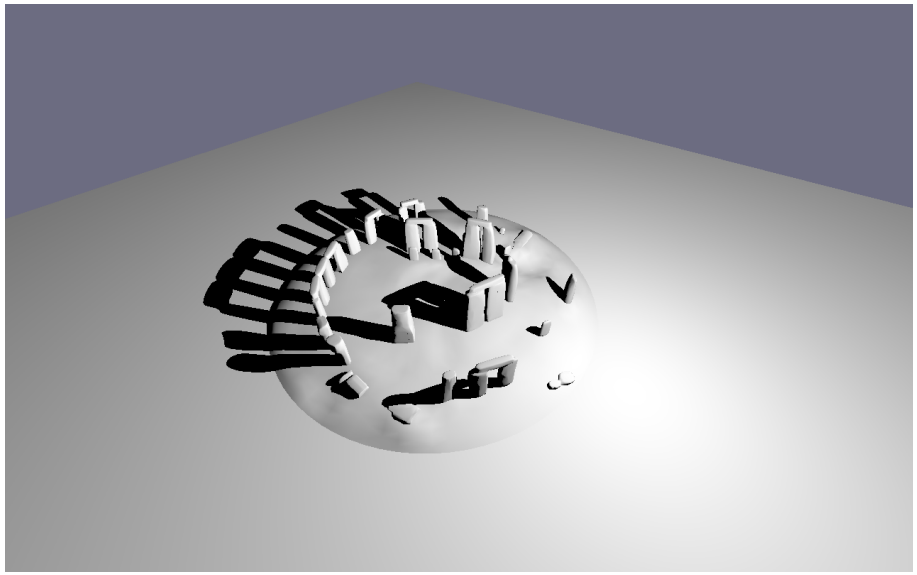


https://youtu.be/SYrHi4jF4dU

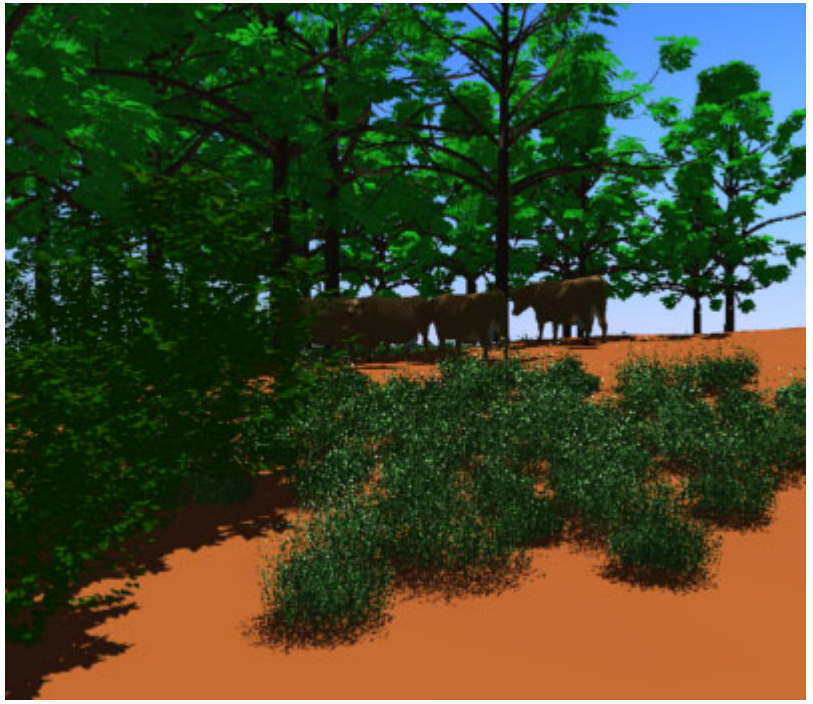
Shadows

- Light blocked by other objects
- Important visual cue
  - Relative location
  - Position of light





Shadows for Visual Realism

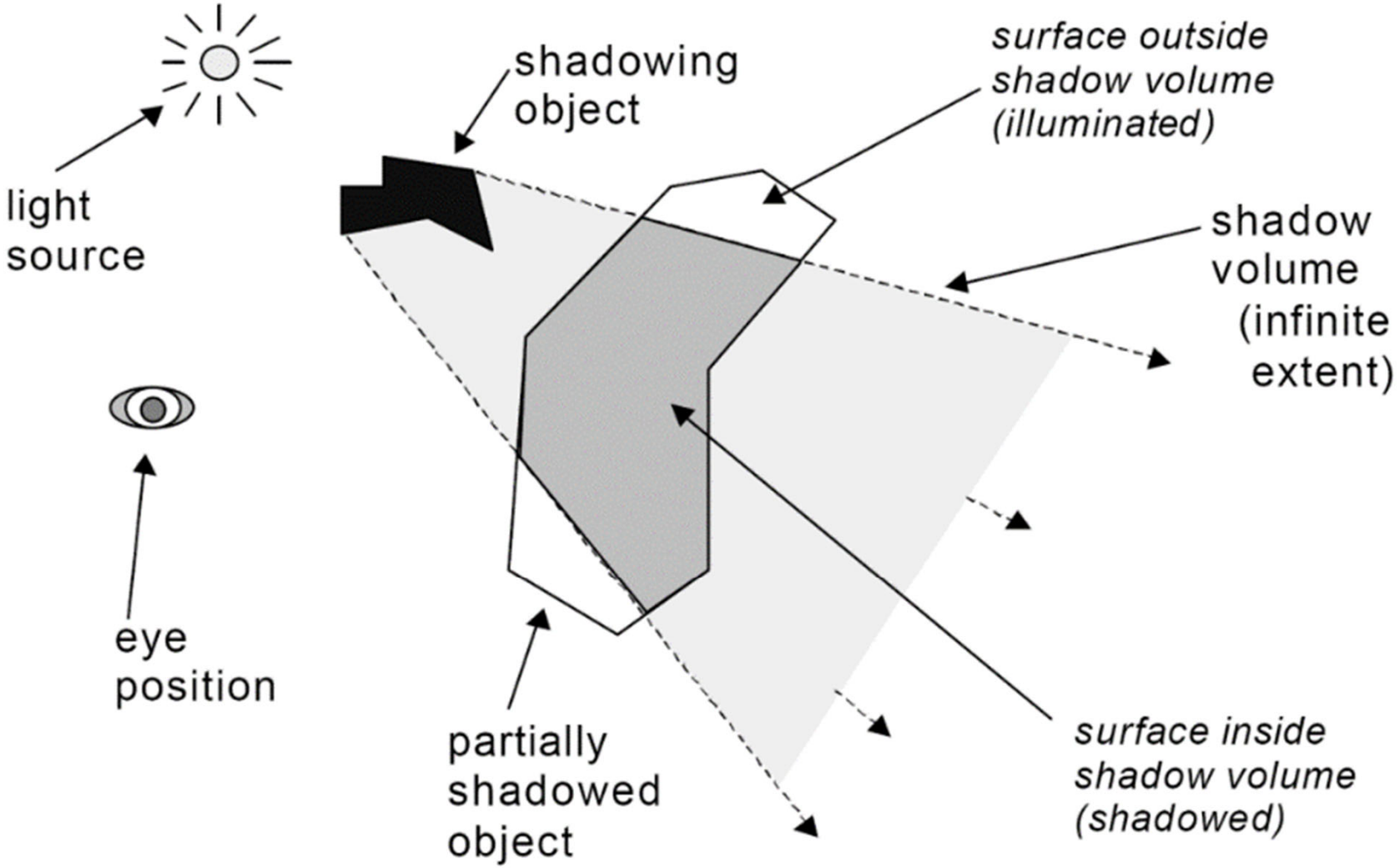


Objects look like they are “floating” → Shadows can fix that!

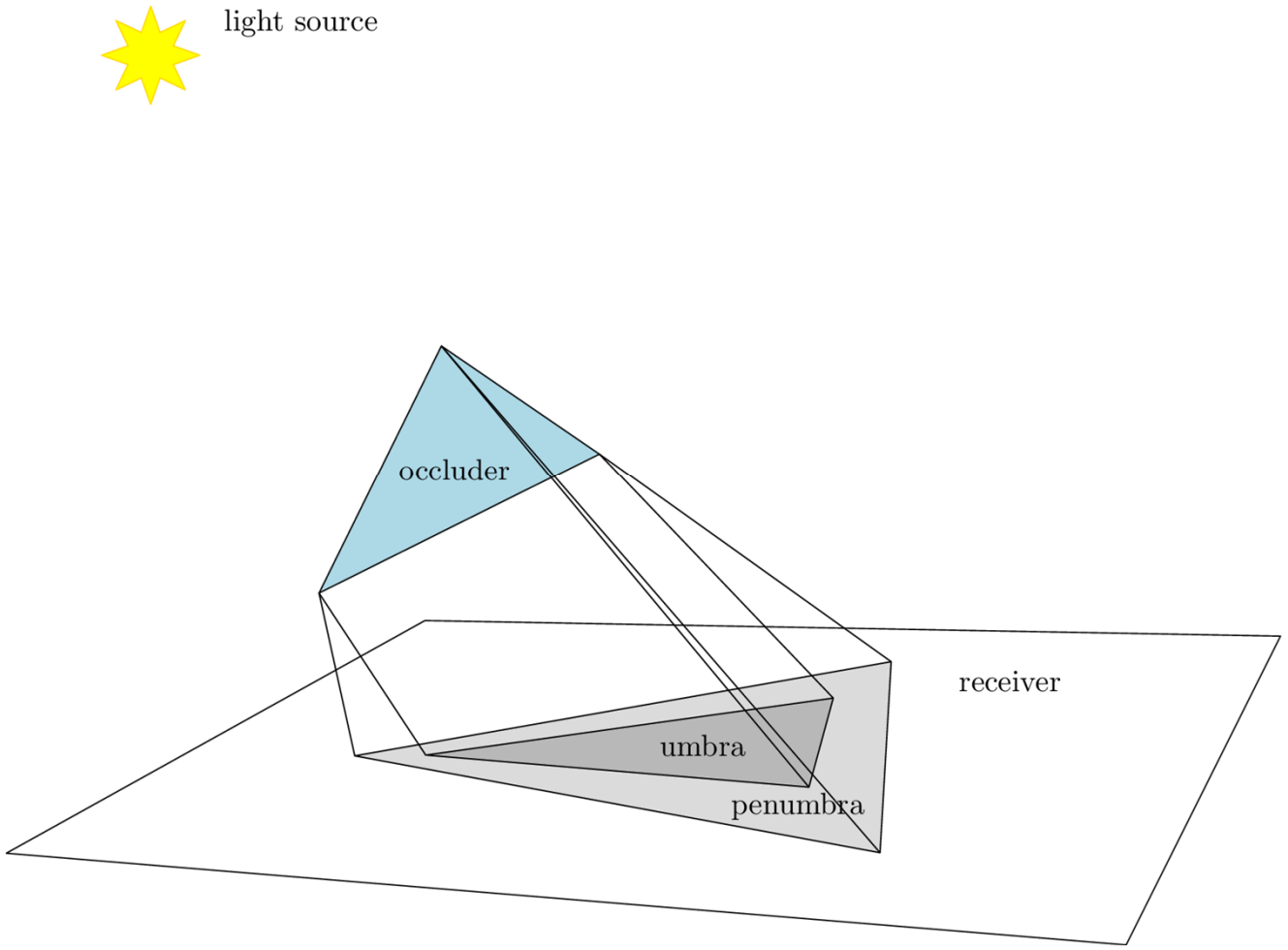
Shadows are Non-Local

- Shadows are a non-local effect between
  - Light source
  - Object currently being rendered (“receiver”)
  - Object blocking light from reaching rendered object (“blocker”, “occluder”, “caster”)
- Shading can be seen as pseudo self-shadowing

Shadow Geometry



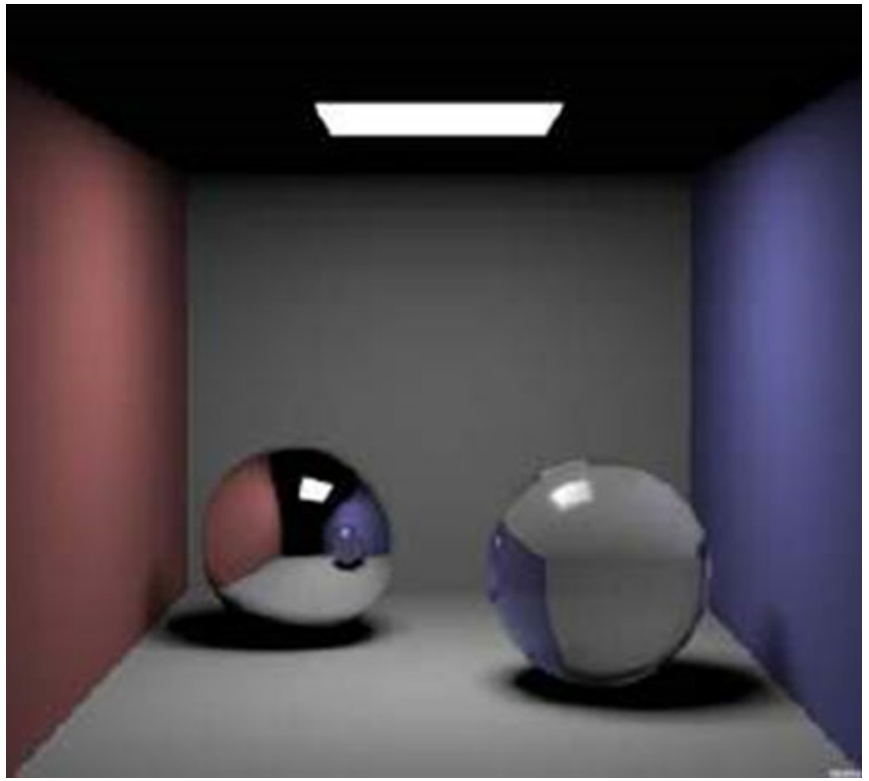
Umbra and Penumbra



Global Illumination

Shadows as byproduct of global light transport

Radiosity



Raytracing

The Problem

- Shadows are a *global* illumination (GI) problem
- GI is hard, especially in online rendering
  - Global information not available during rendering
- Address problem with two-pass techniques
  1. Compute and store (global) shadow information
  2. Render scene using that information

Real-Time Shadows

- Static shadows
  - Light maps - see also previous lectures
  - Updating in real-time is problematic
- Approximate shadows
- Planar projected shadows
- Shadow Maps
- Stencil Shadows

Static Shadows

- Glue to surface whatever we want
- Idea: incorporate shadows into light maps
  - For each texel, cast ray to each light source
- Bake soft shadows in light maps
  - Not by texture filtering alone
  - Must also sample area light sources

Static Soft Shadow Example

	no filtering	filtering
1 sample		
n samples		

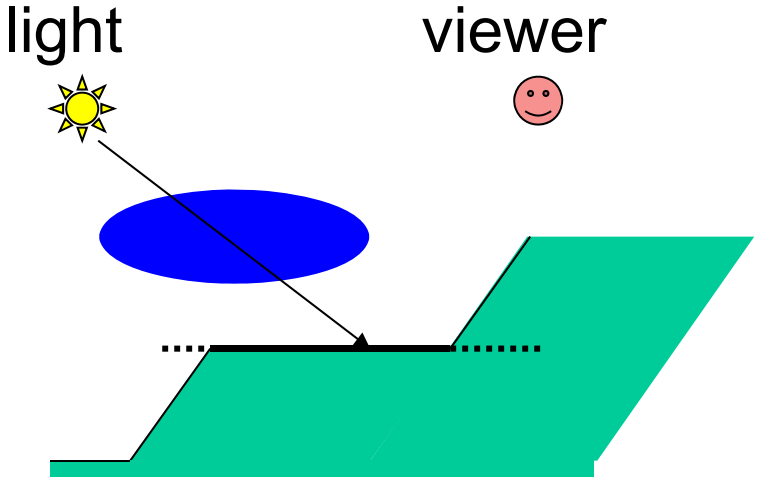
Approximate Shadows 1

- Hand-drawn approximate geometry
  - Perceptual studies suggest: shape not so important
  - Minimal cost

You can tell how young Lara was by counting the vertices of her shadow…

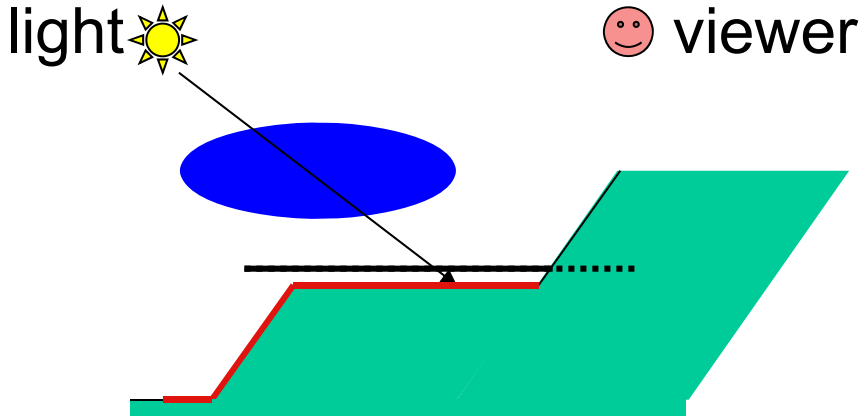
Approximate Shadows 2

- Dark polygon (maybe with texture)
  - Cast ray from light source through object center
  - Blend polygon into frame buffer at location of hit
  - May apply additional rotation/scale/translation to incorporate distance and receiver orientation
- Problem with z-quantization

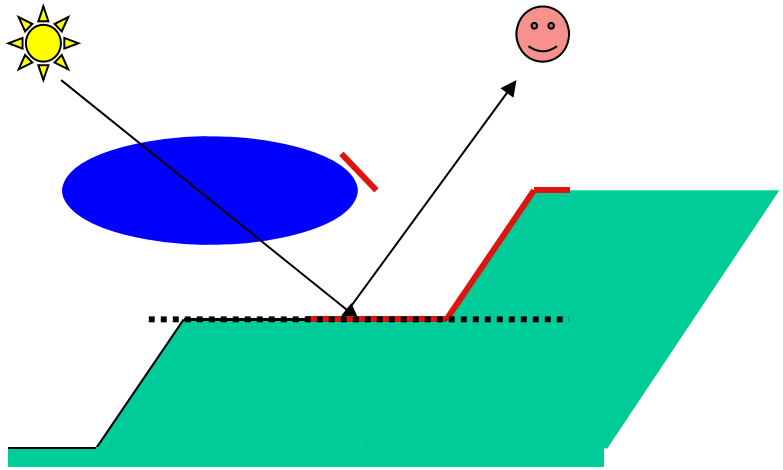


Blend at hit polygon + z-test equal → z-buffer quantization errors!

Approximate Shadows 3



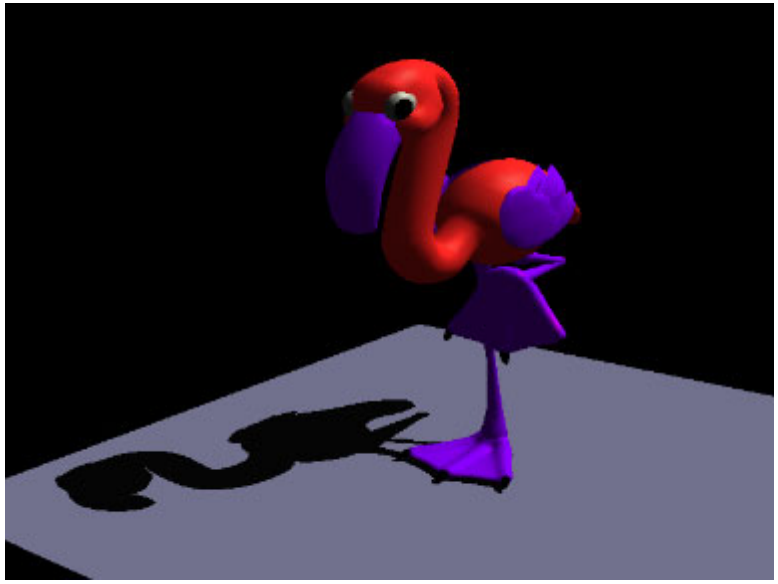
Elevate above hit polygon + Z-test less or equal
→ shadow too big → may appear floating



No z-test, only one eye ray → shadow too big, maybe in wrong place

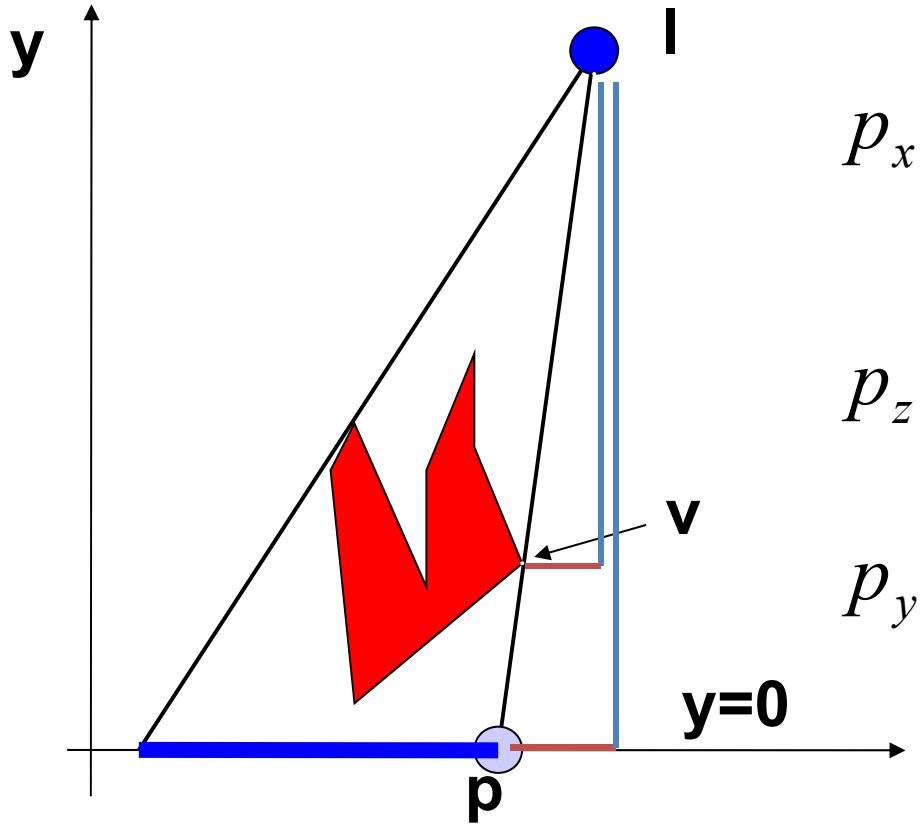
Planar Projected Shadows

- Very simple trick [Jim Blinn, 1988]
- Supports arbitrary shadow caster
- But only planar shadow receiver
- Method
  - ModelView matrix transforms object into plane
  - Object is drawn “flat”
  - Darken surface underneath using framebuffer blend



Projection Onto XZ-Plane

- **p = M v**
- Similar triangles



$$\frac{p_x - l_x}{v_x - l_x} = \frac{l_y}{l_y - v_y}$$

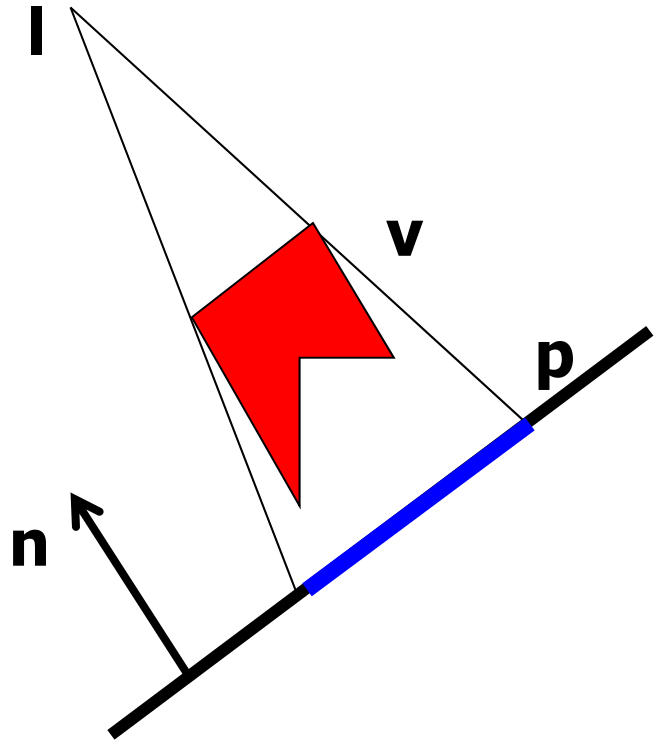
$$p_x = \frac{l_y v_x - l_x v_y}{l_y - v_y}$$

$$p_z = \frac{l_y v_z - l_z v_y}{l_y - v_y}$$

$$p_y = 0$$

$$\mathbf{M} = \begin{pmatrix} l_y & -l_x & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -l_z & l_y & 0 \\ 0 & -1 & 0 & l_y \end{pmatrix}$$

Projection Onto General Plane



ray: $\mathbf{p} = \mathbf{l} + \alpha(\mathbf{v} - \mathbf{l})$ .....α=(p-l)/(v-l)

plane: $\mathbf{n} \cdot \mathbf{p} + d = 0$ .....p=-d/n

α=(-d/n-l)/(v-l)= -d/(n(v-l)) - l/(v-l)

$$\mathbf{p} = \mathbf{l} - \frac{d + \mathbf{n} \cdot \mathbf{l}}{\mathbf{n} \cdot (\mathbf{v} - \mathbf{l})}(\mathbf{v} - \mathbf{l})$$

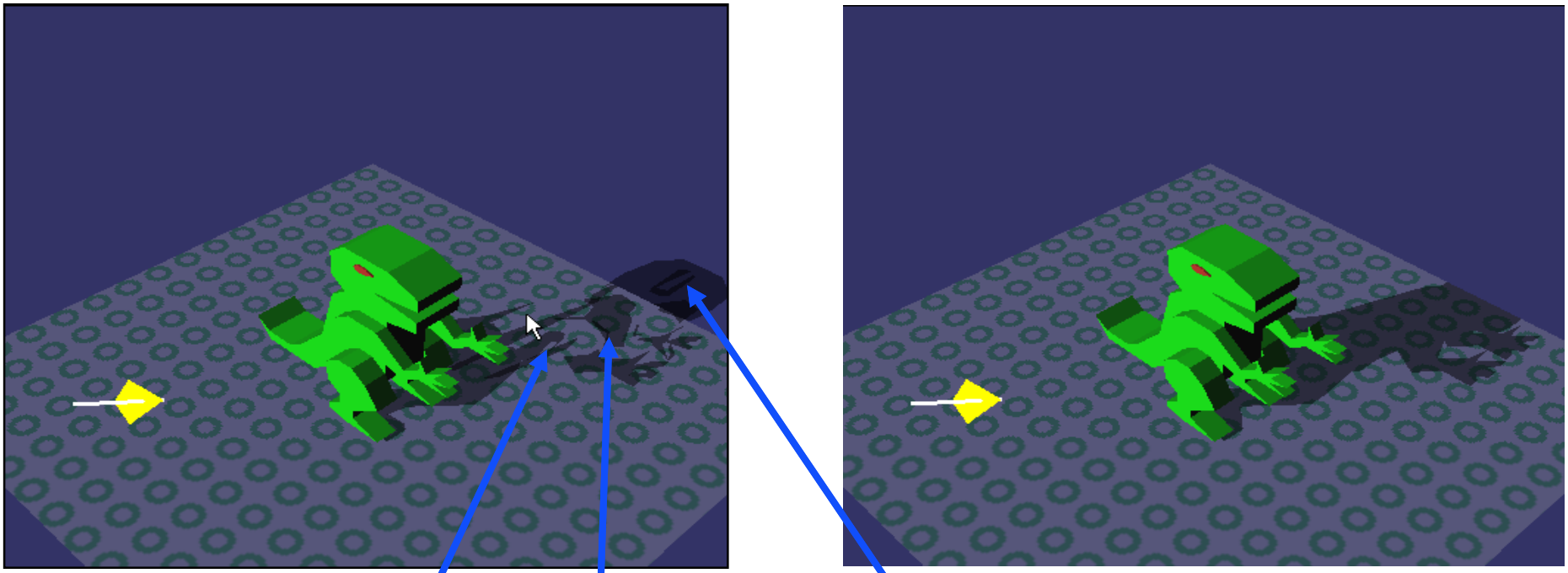
p = M v →

$$\mathbf{M} = \begin{pmatrix} \mathbf{n} \cdot \mathbf{l} + d - l_x n_x & -l_x n_y & -l_x n_z & -l_x d \\ -l_y n_x & \mathbf{n} \cdot \mathbf{l} + d - l_y n_y & -l_y n_z & -l_y d \\ -l_z n_x & -l_z n_y & \mathbf{n} \cdot \mathbf{l} + d - l_z n_z & -l_z d \\ -n_x & -n_y & -n_z & \mathbf{n} \cdot \mathbf{l} \end{pmatrix}$$

Projected Shadows - Artifacts

Bad

Good



Z-fighting

double blending

extends off ground

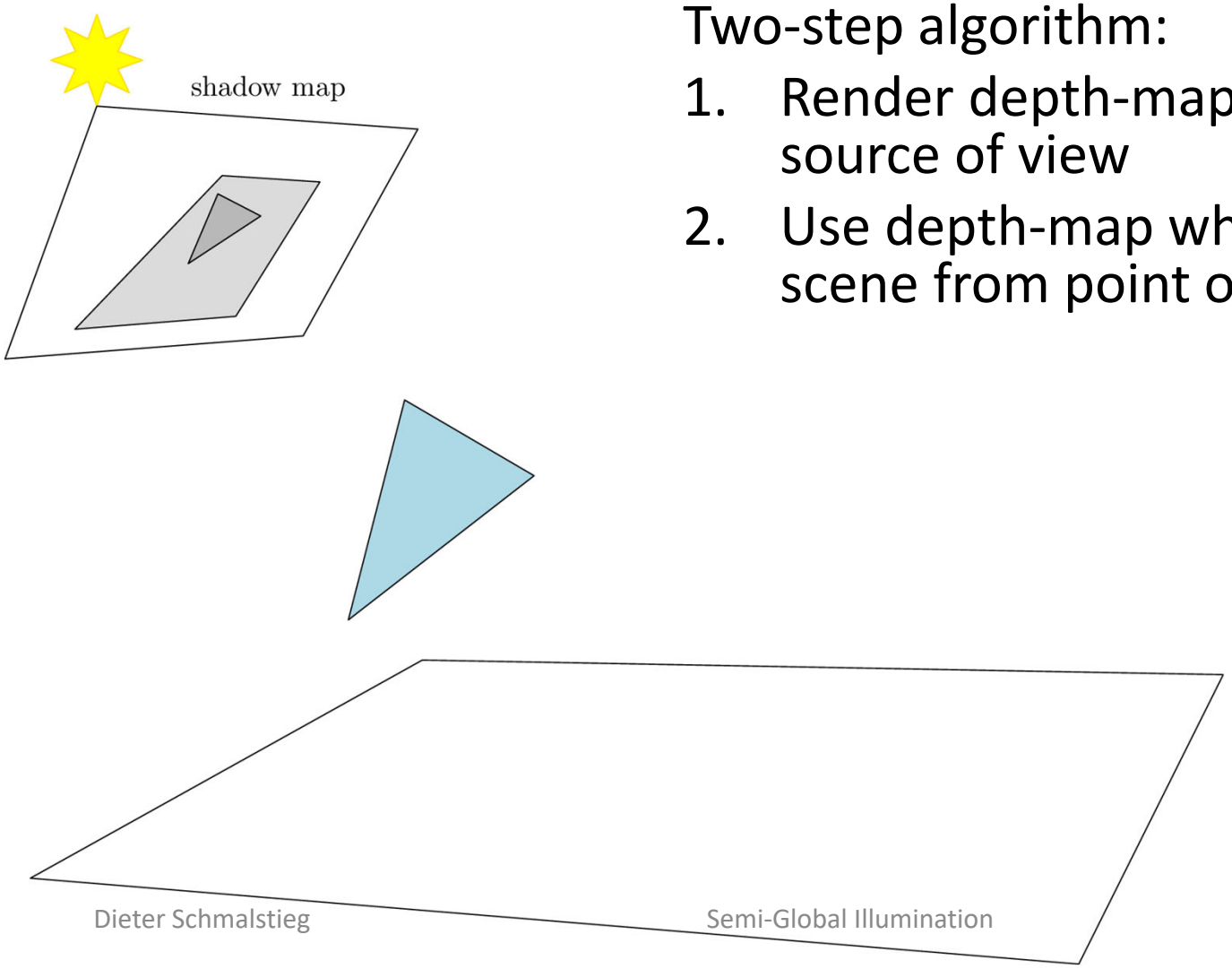
Projected Shadows with Stencil

- Clear stencil buffer to 0
- Draw receiver plane
  - Write 1 to stencil buffer
- Draw 3D object
- Draw shadow
  - Where stencil is 1
  - Write 0 to stencil buffer
  - No z-fighting problem

Projected Shadows - Summary

- Easy to implement
  - GLQuake first game to implement it
- Only practical for very few, large receivers
- No self-shadowing
- Possible remaining artifacts: wrong shadows
  - Objects behind light source
  - Objects behind receiver

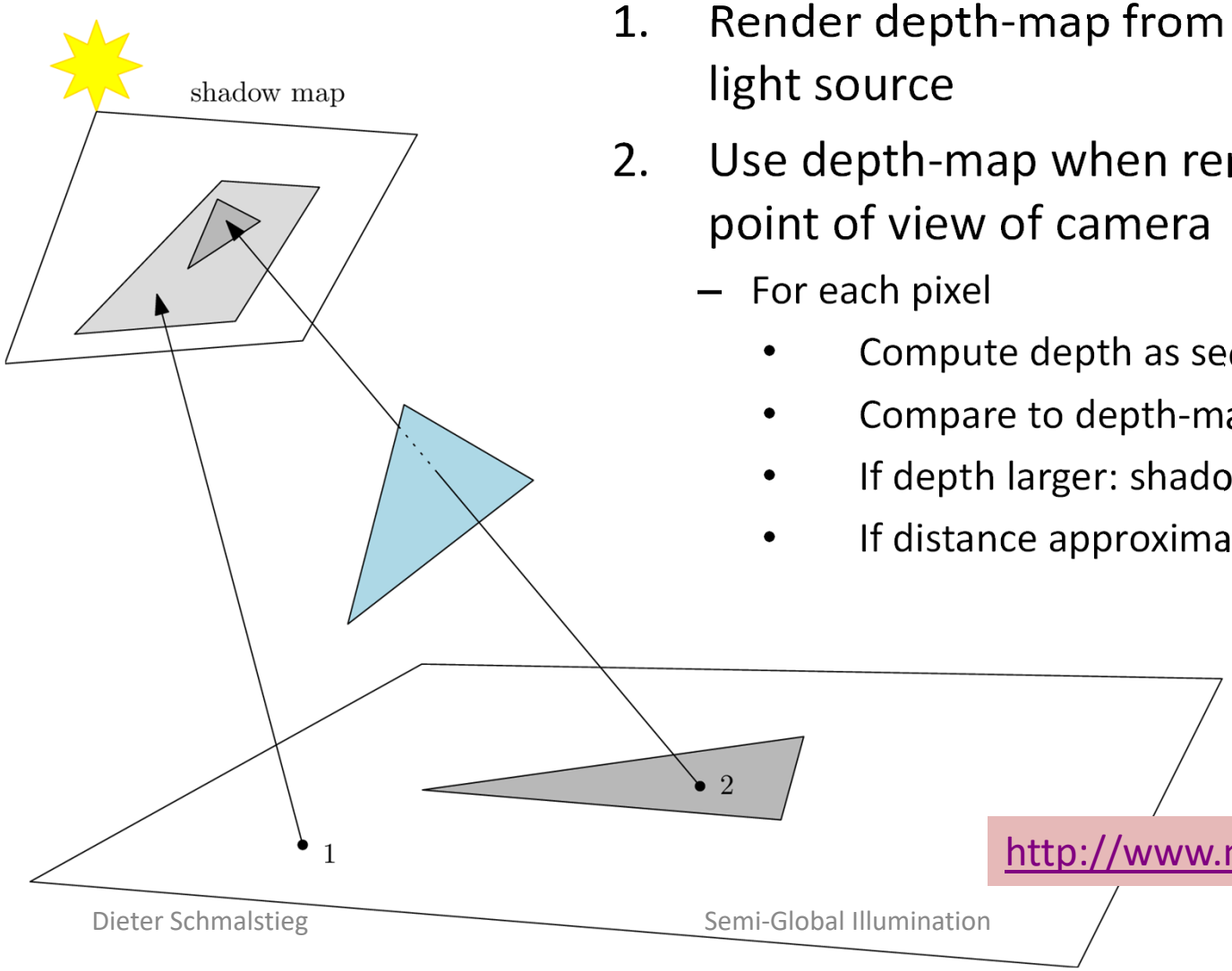
Shadow Mapping Algorithm 1



Two-step algorithm:

1. Render depth-map from point light source of view
2. Use depth-map when rendering scene from point of view of camera

Shadow Mapping Algorithm 2

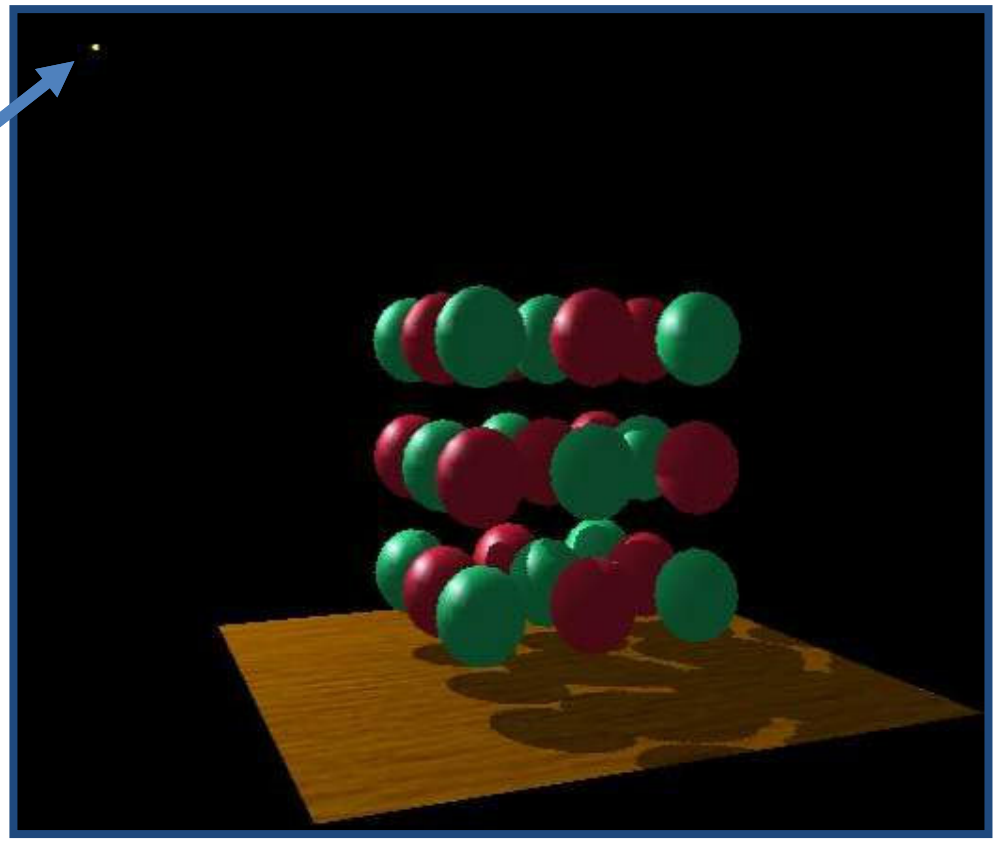


1. Render depth-map from point of view of the light source
2. Use depth-map when rendering scene from point of view of camera
   - For each pixel
     - Compute depth as seen from light source
     - Compare to depth-map
     - If depth larger: shadow
     - If distance approximately equal: light

http://www.nutty.ca/webgl/shadows/

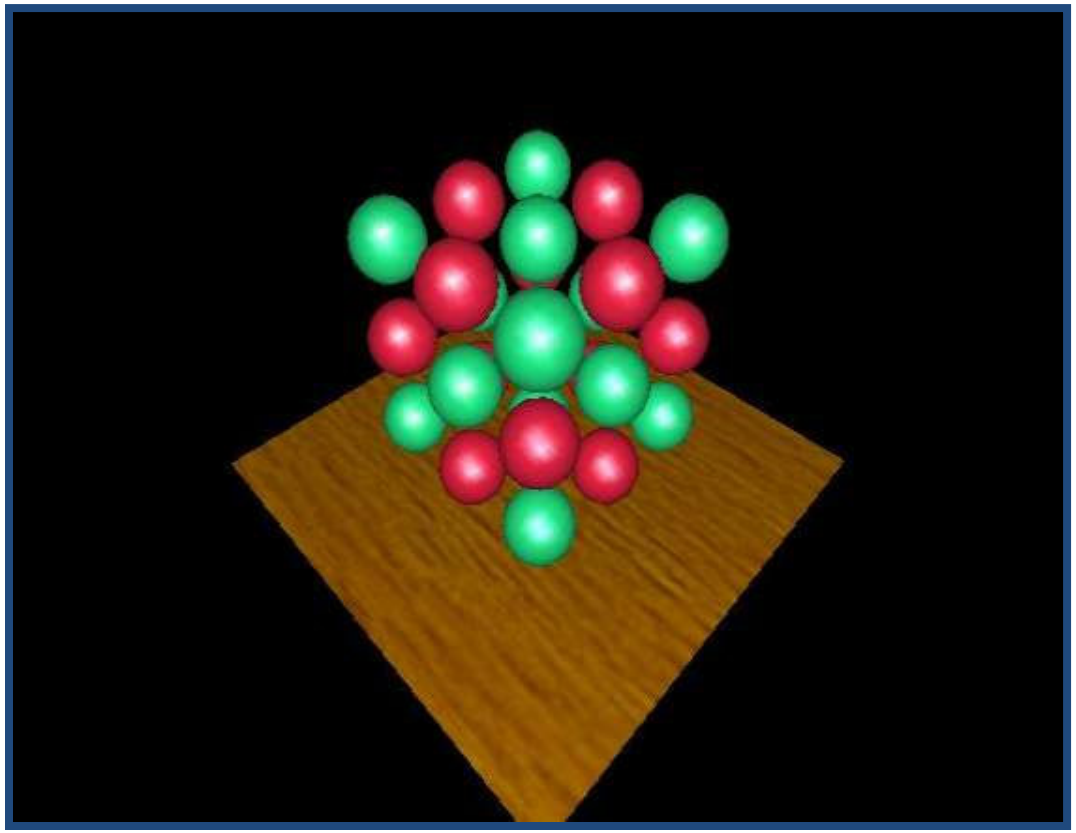
Shadow Mapping - Point Light

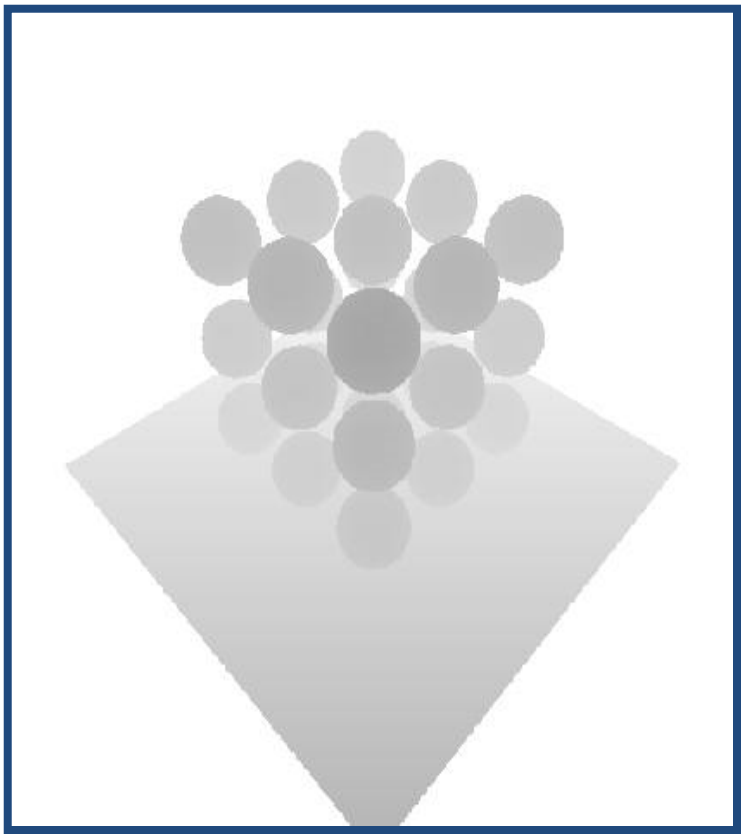
Our point light source



Shadow Map as Depth Map

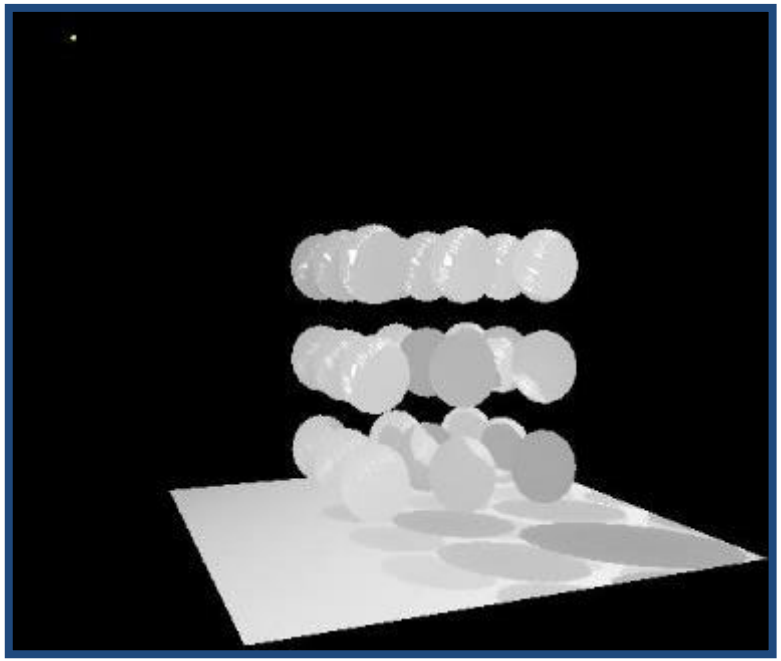
- Rendering the Depth-Map

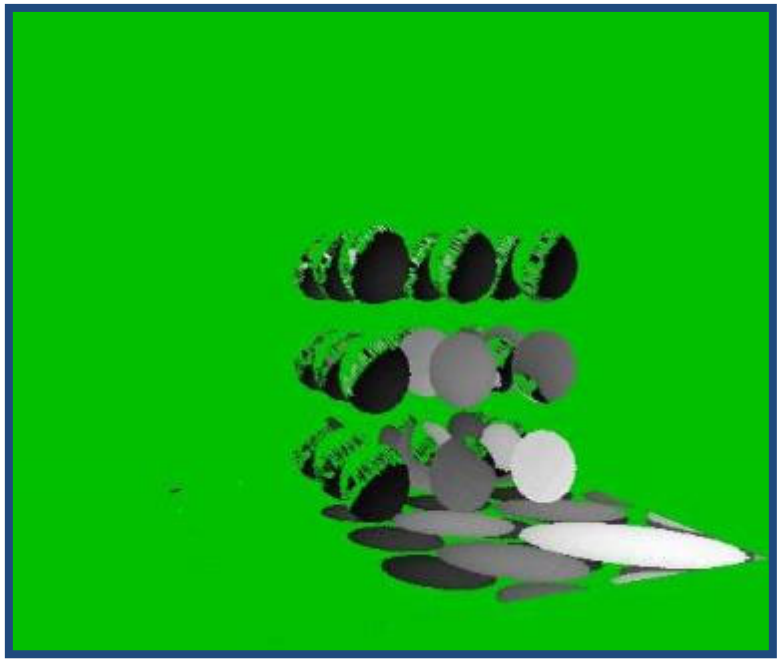




Shadow Mapping - Depth Test

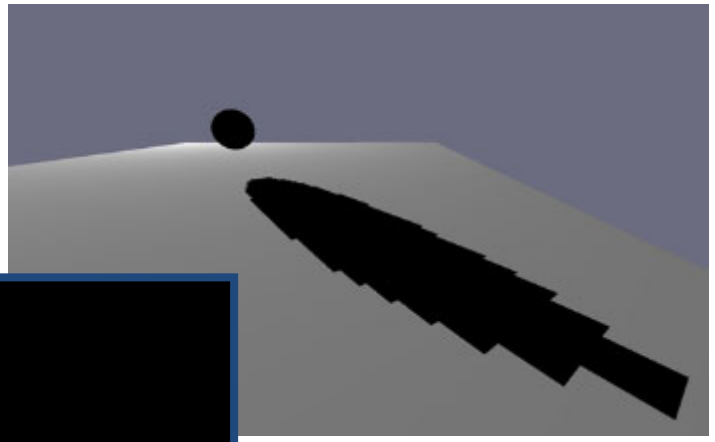
- Green: depth approximately equal
- Non-green: shadow

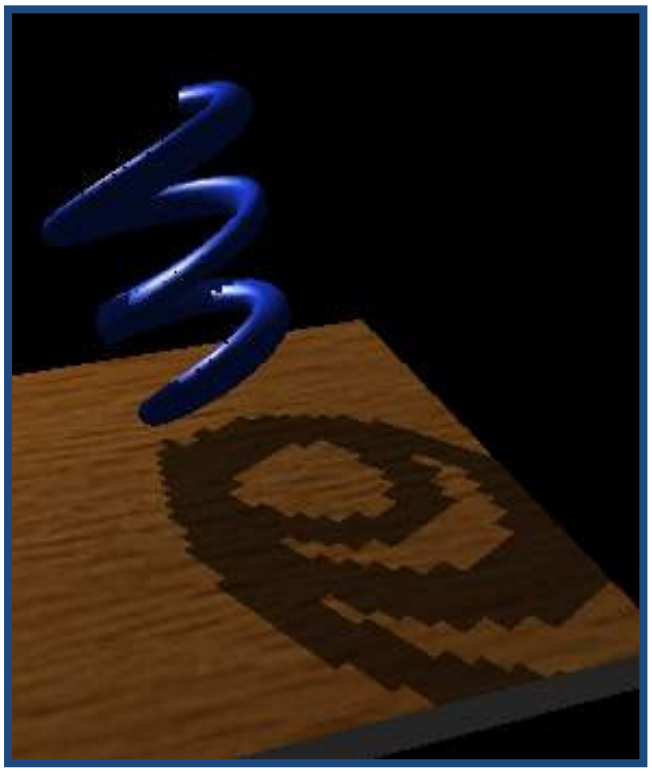




Aliasing

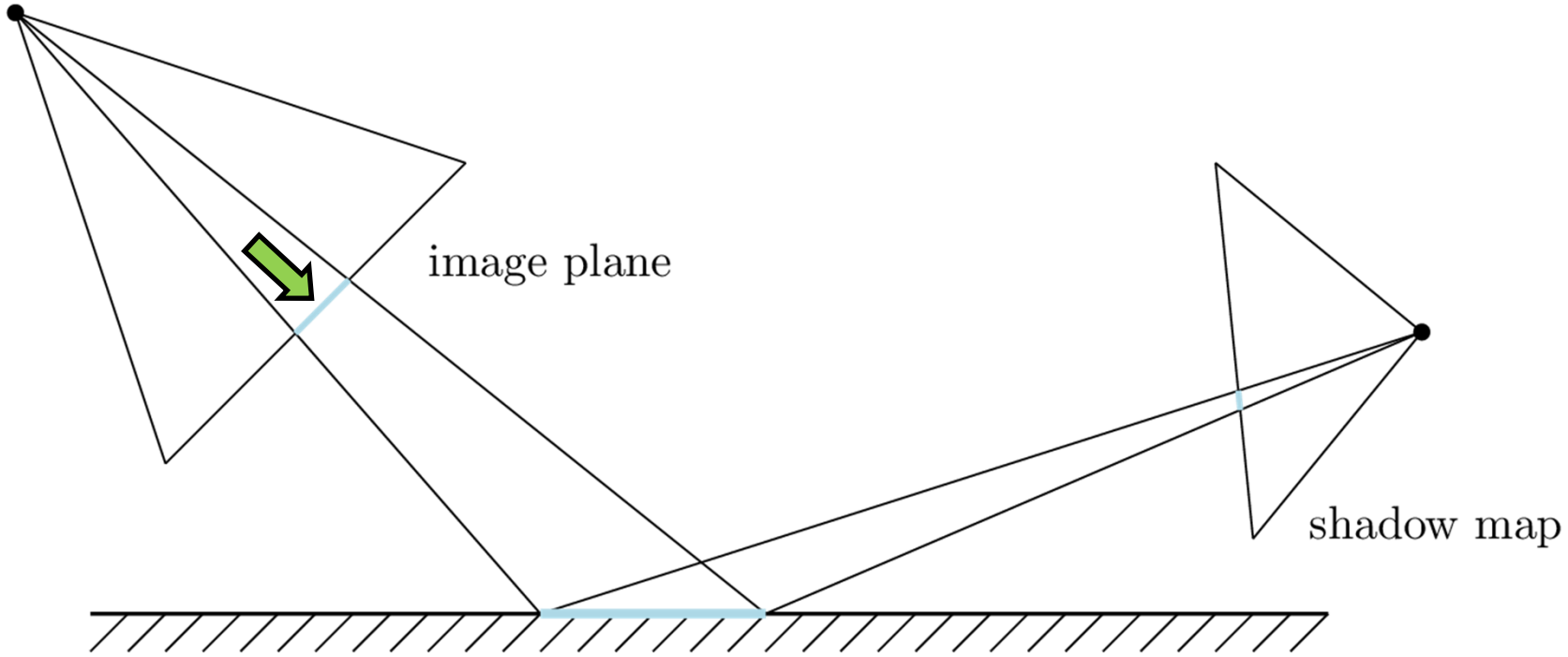
- Two components
  - Perspective aliasing
  - Projection aliasing





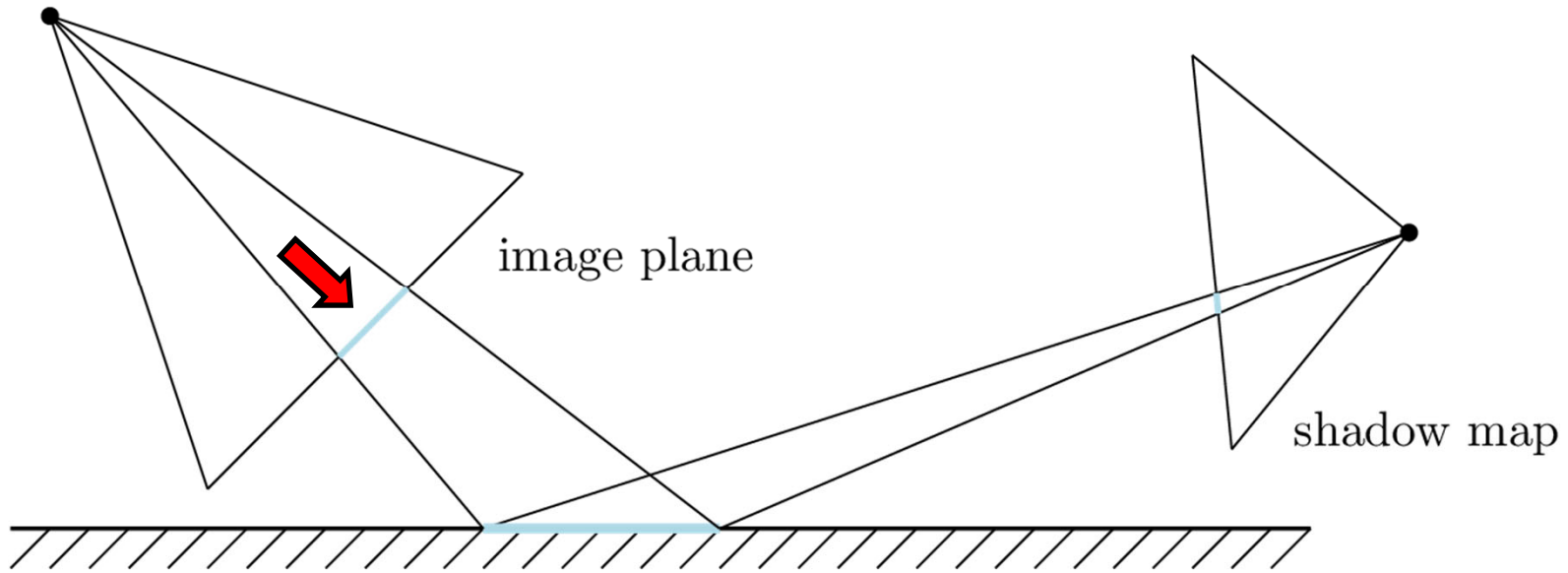
Perspective Aliasing 1

- Texel resolution in image plane is okay



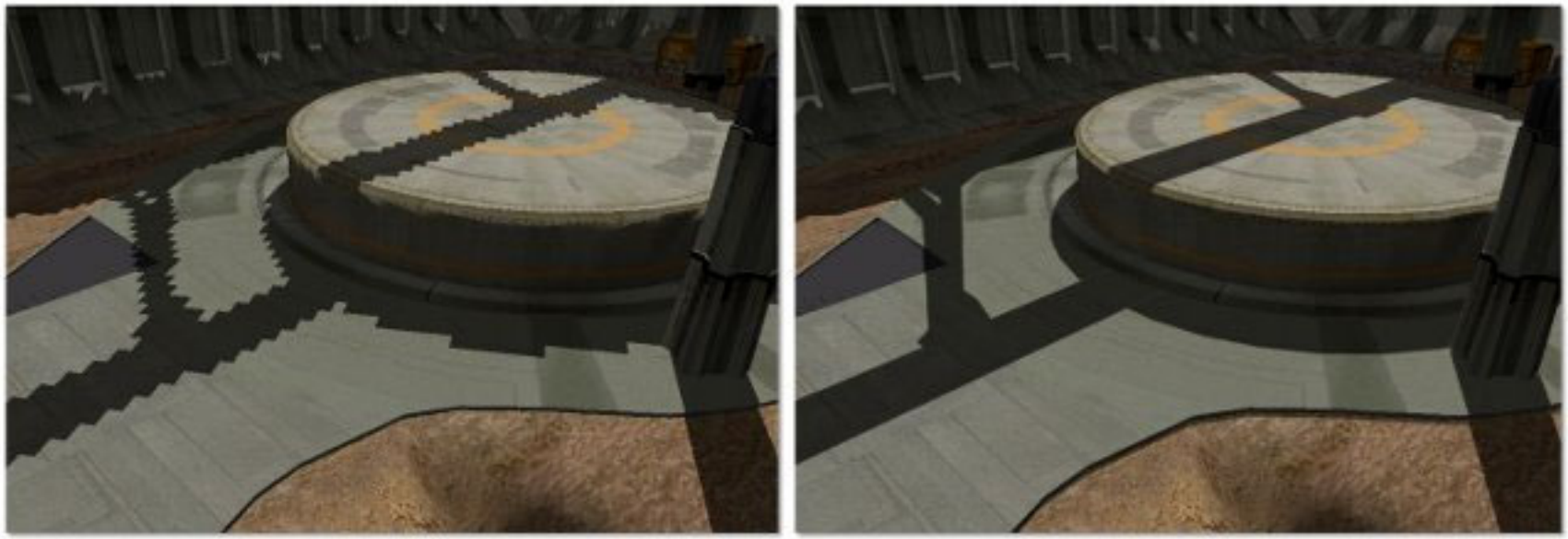
Perspective Aliasing 2

- If camera moves closer,
  texel resolution in image space gets worse



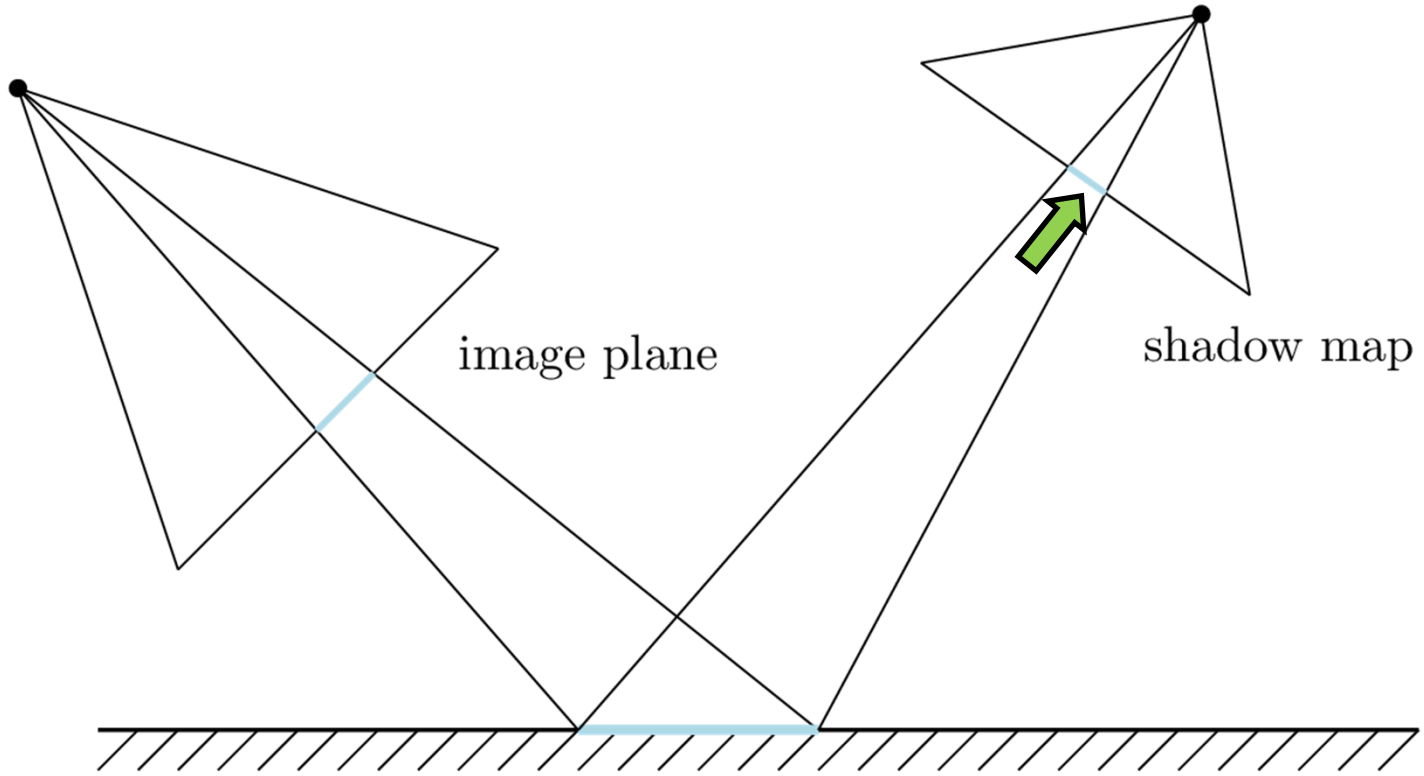
Example: Perspective Aliasing

Shadow texels large and blocky



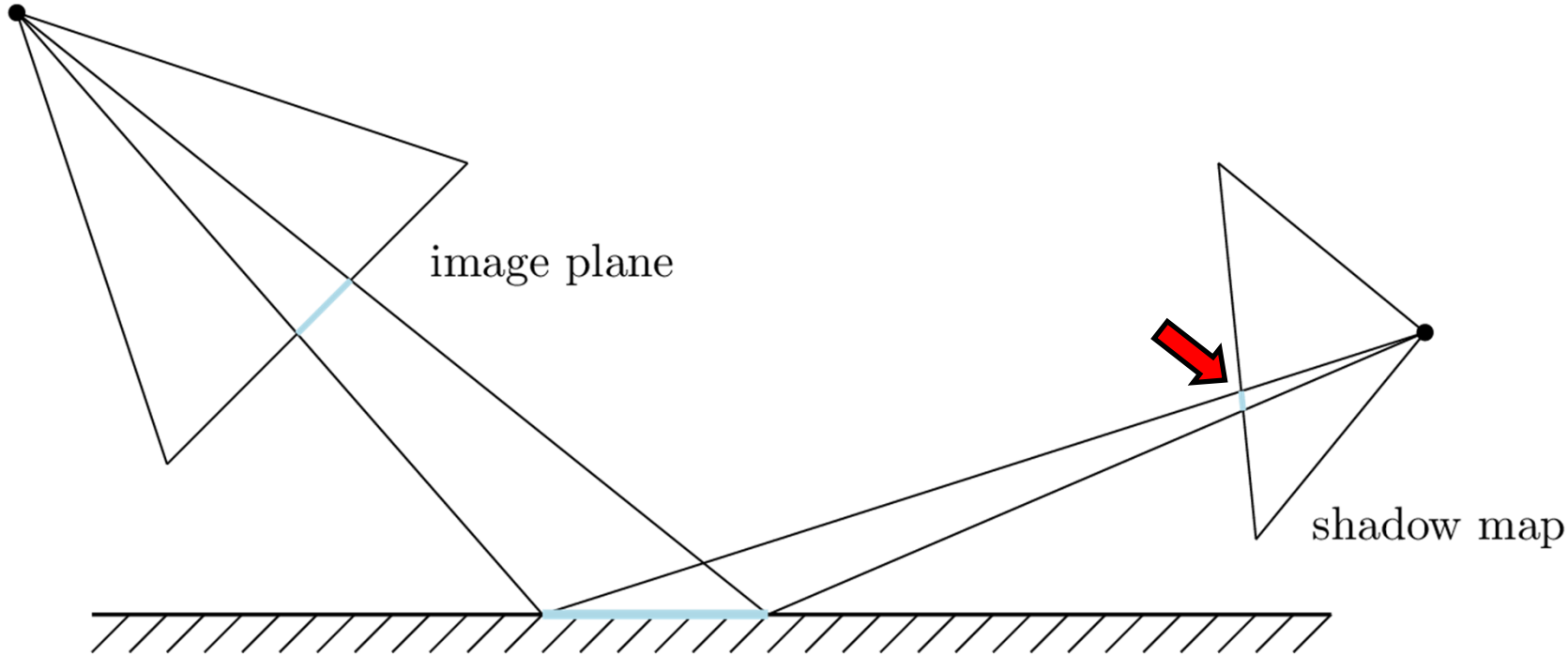
Projection Aliasing 1

- Shadow map resolution okay



Projection Aliasing 2

- If angle to light source gets larger, shadow map resolution gets worse



Example: Projection Aliasing

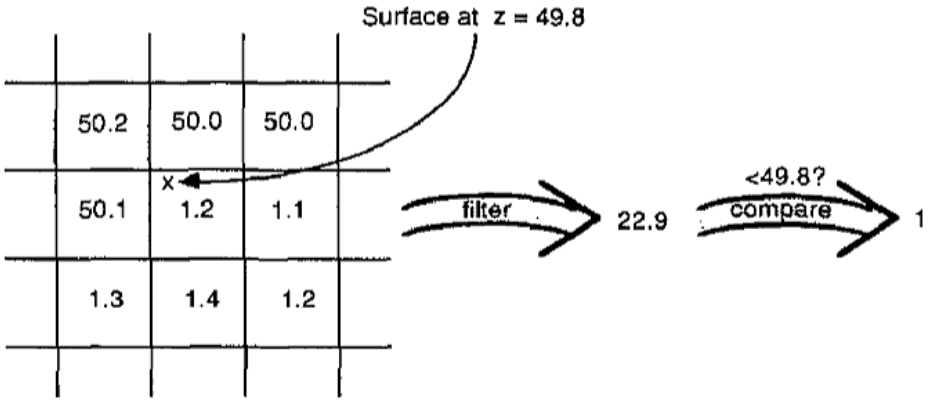
Shadow texels stretched

How to Reduce Aliasing?

- Increase shadow map resolution
  - Not practical in general
- Practical approaches
  - Anti-aliasing by filtering
  - Optimize sample distribution in shadow map

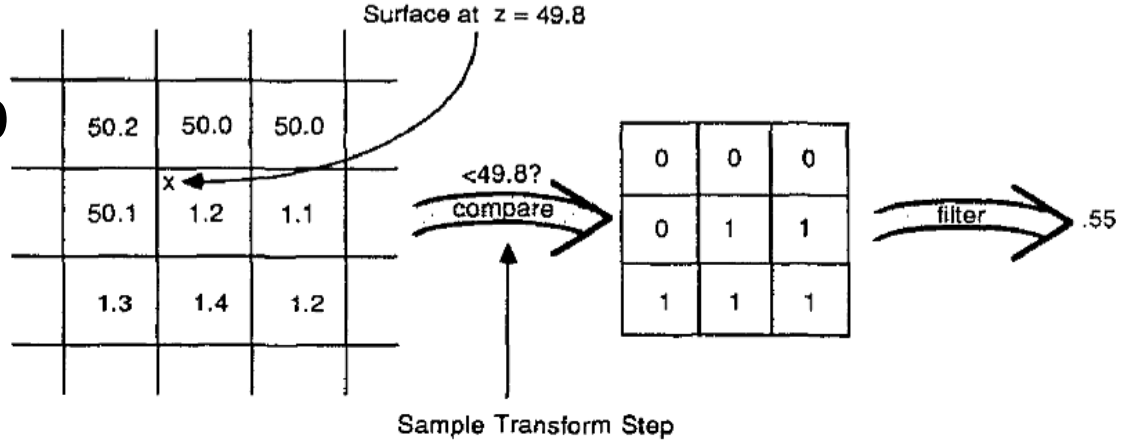
Percentage Closer Filtering

- Normal bilinear filtering cannot be used on depth



a) Ordinary texture map filtering. Does not work for depth maps.

- Must filter lookup result, not depth!



b) Percentage closer filtering.

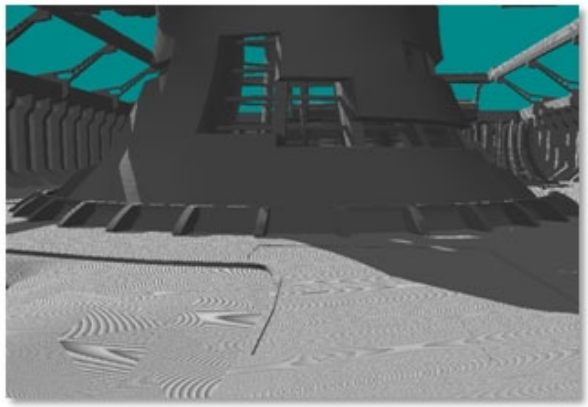
PCF Example

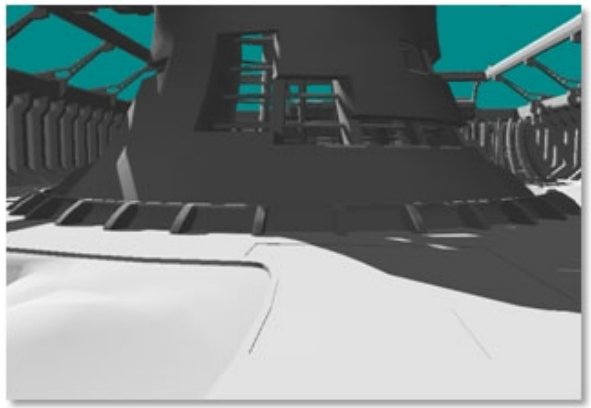
- https://www.youtube.com/watch?v=qA920lufMxU

(requires multi-pass) (single-pass feature)

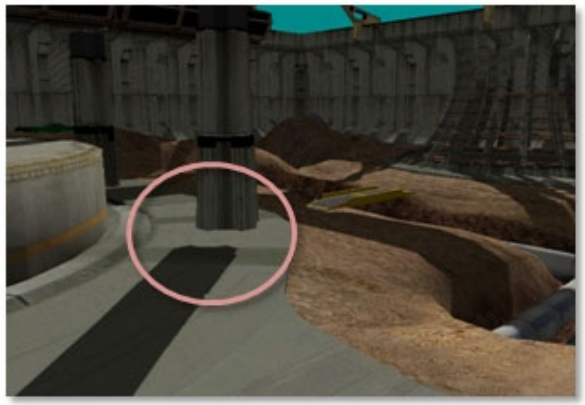
Depth Buffer Precision Problems

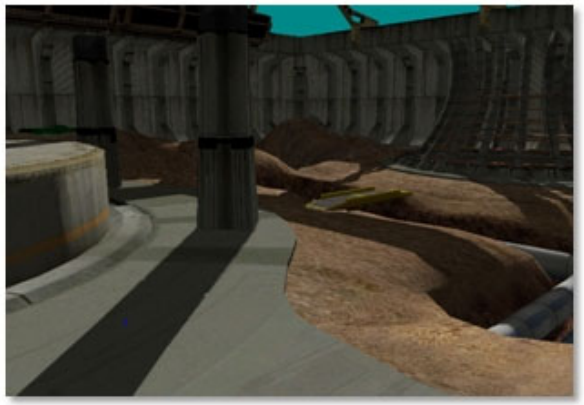
- Shadow Acne
  - Depth offset too small





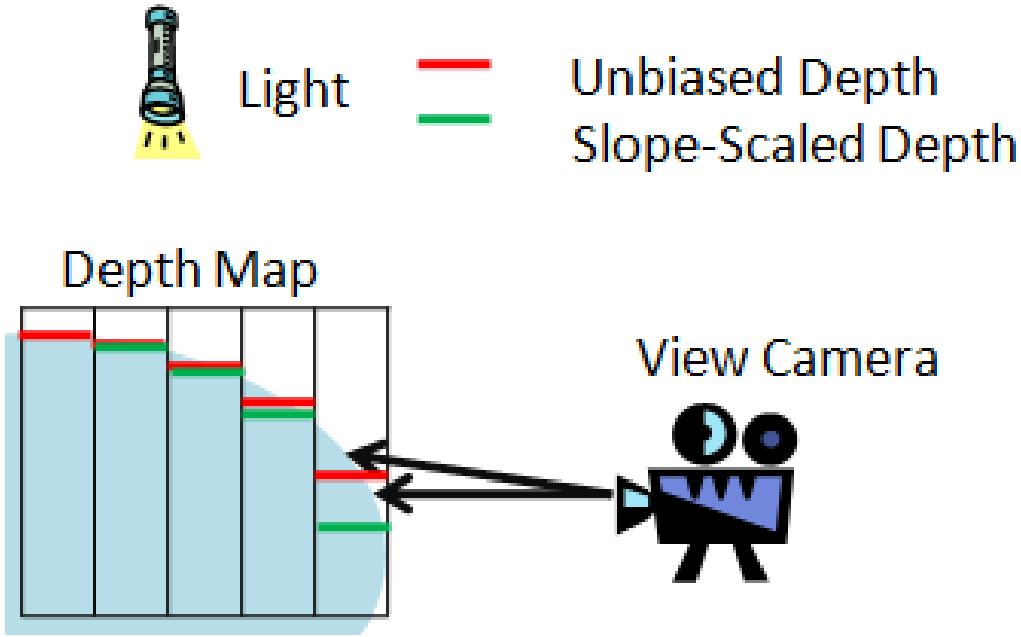
- Peter Panning
  - Shadow offset too large





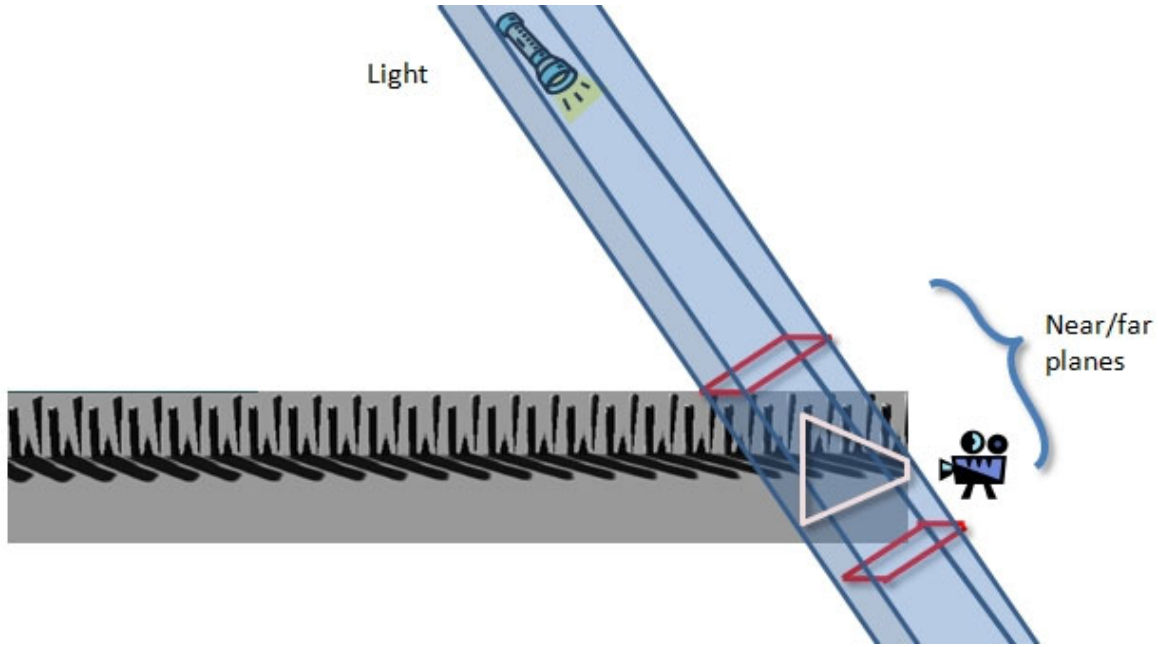
Slope Scale Bias

- Depth offset is modified by adding a bias
- Bias a polygon based on its slope with respect to the view direction

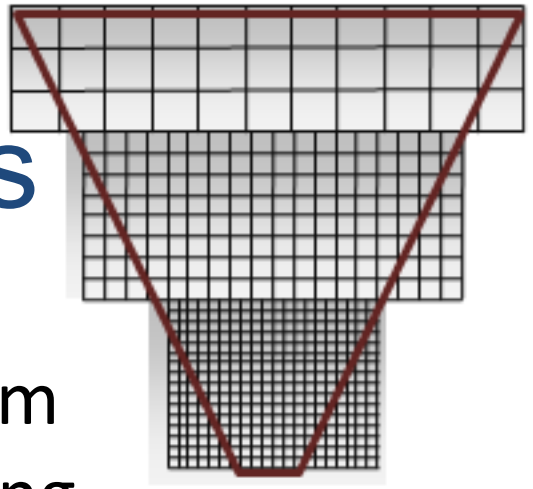


Optimize Shadow Frustum

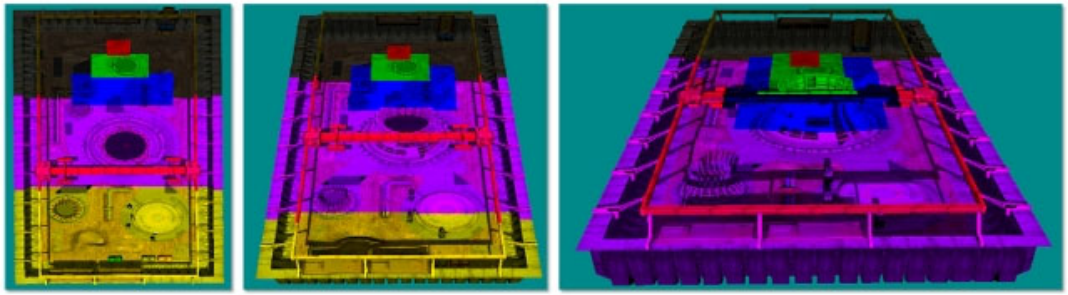
- Optimal use of depth precision: far-near→min
- Choose near/far planes based on intersection of light frustum and scene bbox



Cascaded Shadow Maps

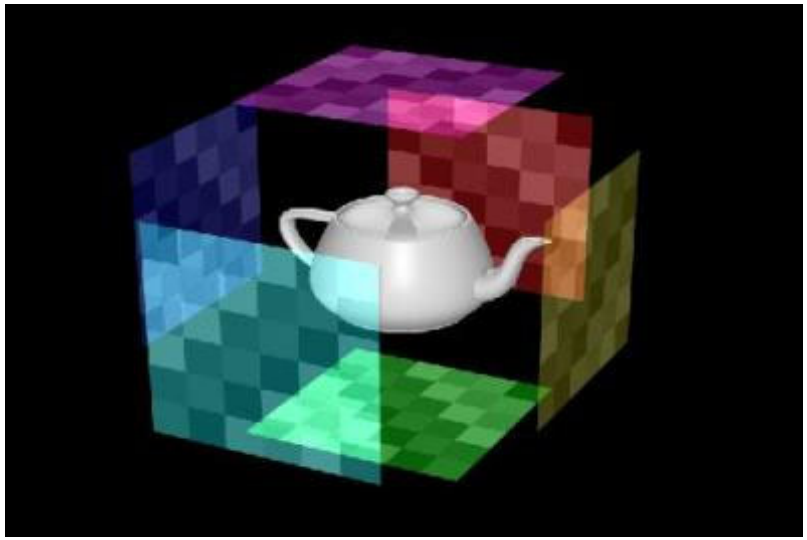


- Partition view frustrum
- Generate one cascade per sub-frustum
- Simultaneous multi-viewport rendering
  - One viewport created per cascade
  - Each viewport renders to separate portion of the depth buffer
- Cascades must overlap (for blending)
- Cascade overlap increases as light direction becomes parallel with camera direction

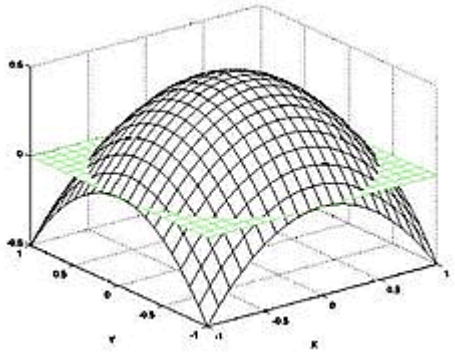


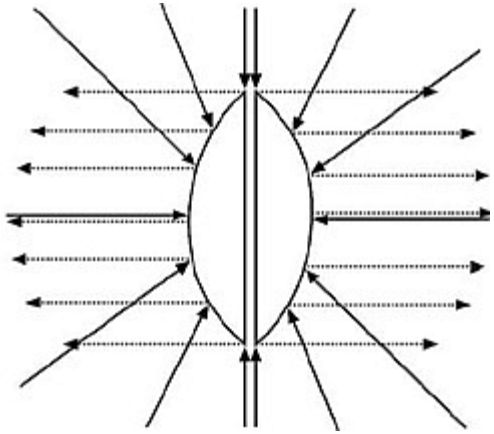
Omni-directional Shadow Maps

- Shadow maps limited to spot-light
- Cannot have a 360° “viewing frustum”
- Use six shadow maps for omni-directional light source
- Expensive!



Dual Paraboloid Shadow Map





- Paraboloid maps directions inside one half-sphere to a disc
- Arrange two such discs inside a shadow map (or environment map)
- Supported as native texture coordinate mode
- Can be generated in two passes (instead of six)

https://www.youtube.com/watch?v=xoTMdEoMlhQ

Shadow Maps Summary

- Advantages
  - Fast – only one additional pass
  - Independent of scene complexity (no additional shadow polygons!)
  - Self shadowing (but beware bias)
  - Can sometimes reuse depth map
- Disadvantages
  - Problematic for omnidirectional lights
  - Biasing tweak (light leaks, surface acne)
  - Jagged edges (aliasing)

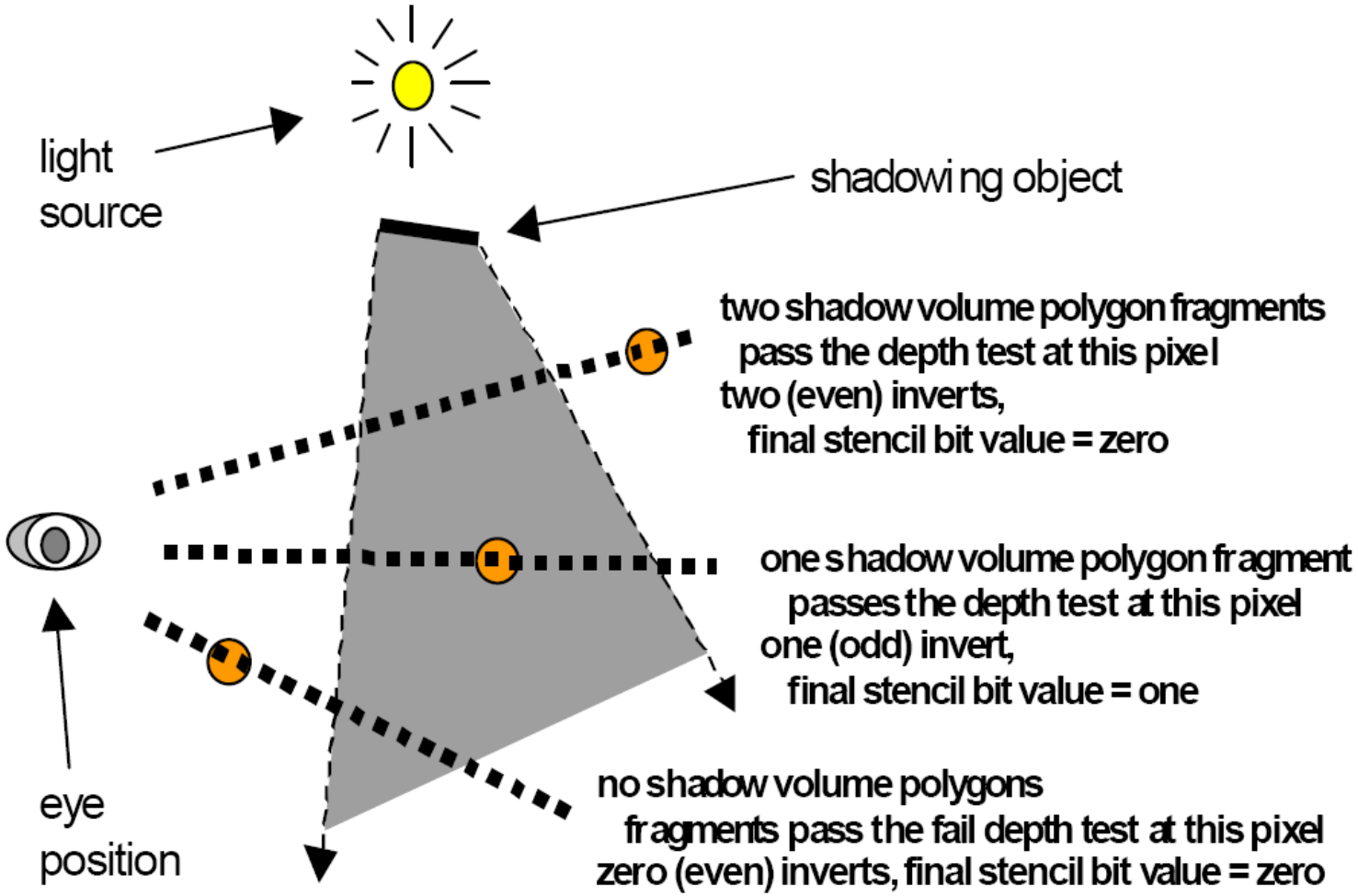
Shadow Volumes

[Frank Crow, 1977]

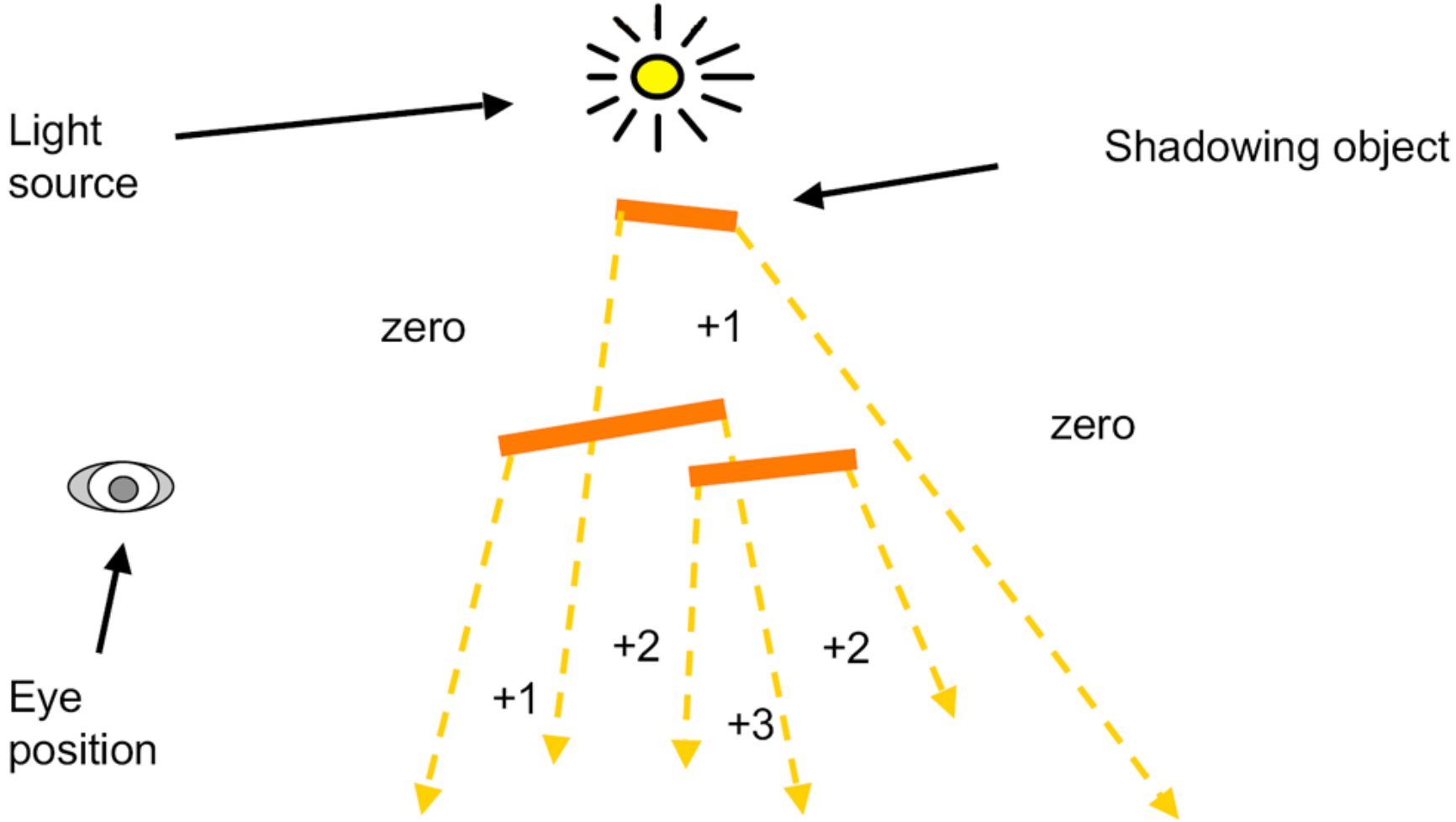
- Complex, but can be implemented efficiently using stencil buffer
- No aliasing
- Method
  - Intersect view rays with shadow volume
  - Count number of intersections, until receiver is hit

Example: Doom 3

Shadow Volumes

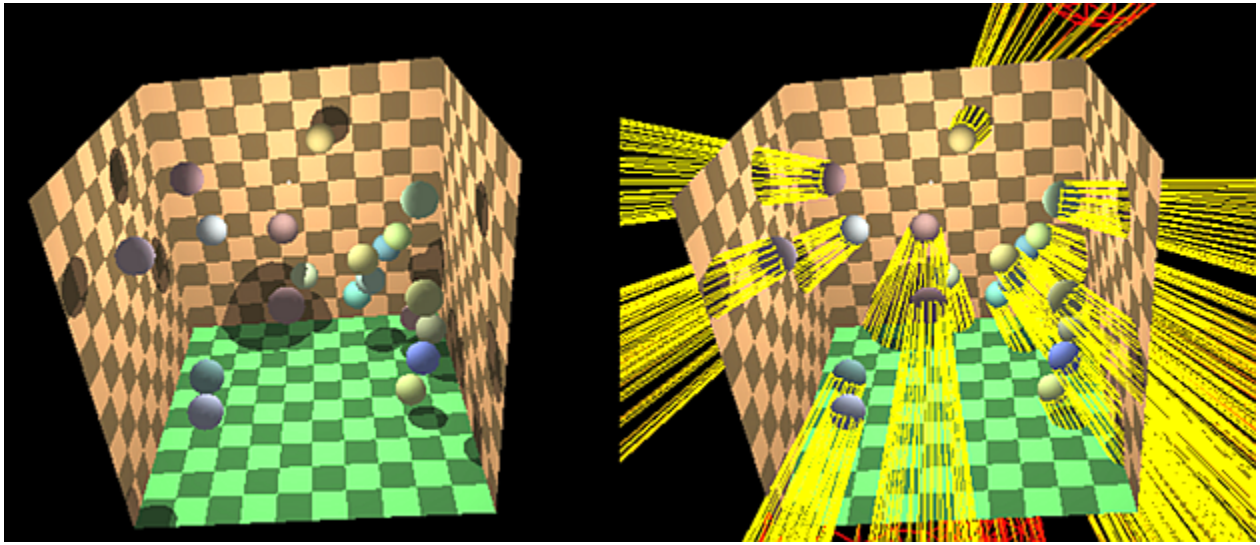


Counting Scheme



Shadow Volume Problems

- Camera inside shadow volume
  - Treat as special case or use “depth-fail” method
- Shadow volume intersects near-plane
  - Solution: render “front-caps”
- Objects must be manifold
- High amount of overdraw
- Fill-rate bound

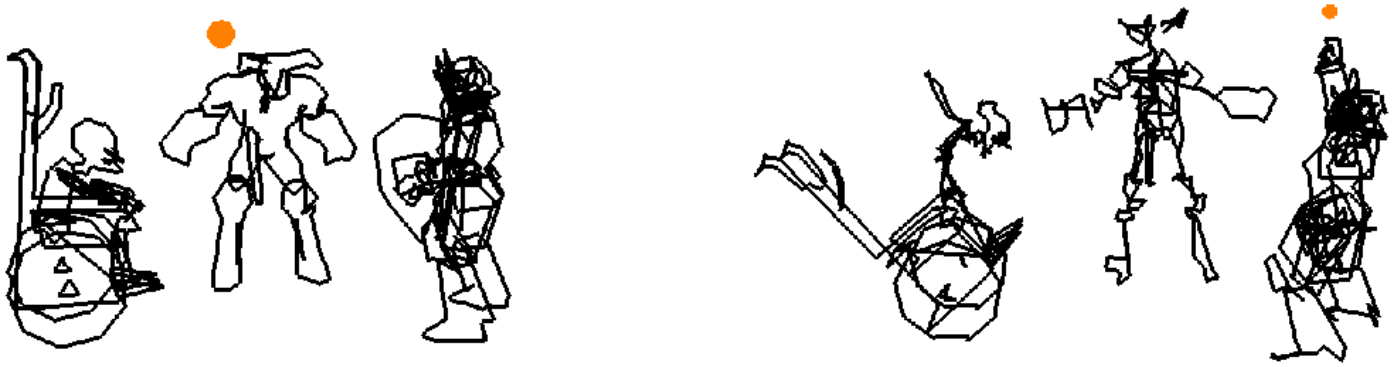


Shadow Volume Geometry

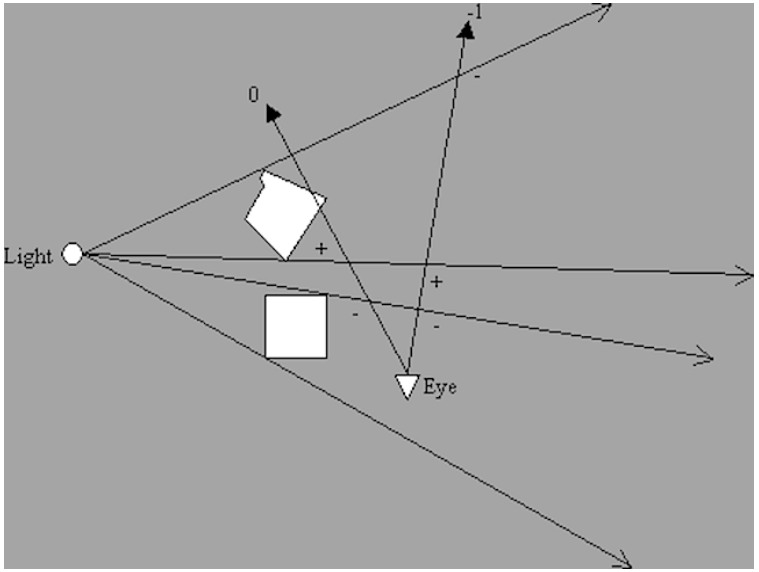
- Closed polyhedron with 3 sets of polygons
  - Light cap
    - Object polygons facing the light
  - Dark cap
    - Object polygons facing away from the light
    - Projected to infinity (with w=0)
  - Sides
    - Actual extruded object edges
    - Which edges?

Silhouette Detection

- Classify polygons into frontfaces and backfaces
- Edge shared by front + back face is on silhouette
- Extrude silhouette edges away from light source
  - In vertex shader
- If necessary: add front-cap and back-cap



Stencil Shadow Volumes Algorithm



- Fill depth-buffer with scene
- Disable depth writes
- Render front-faces of stencil volumes with stencil increment on depth test pass
- Counts shadows in front of objects
- Render back-faces of stencil volumes with stencil decrement on depth test pass
- All lit surfaces have stencil value of 0
- Incorrect if the camera is inside a shadow volume!

Shadow Algorithms Compared

Shadow Volumes

- Needs geometry
  - Closed manifolds 😐
- Cost hard to predict
  - Geometric complexity of shadow volume ☹
- Shadow rendering is bandwidth intensive ☹
  - Stencil buffer techniques
  - Additional pass
- No sampling artifacts ☺

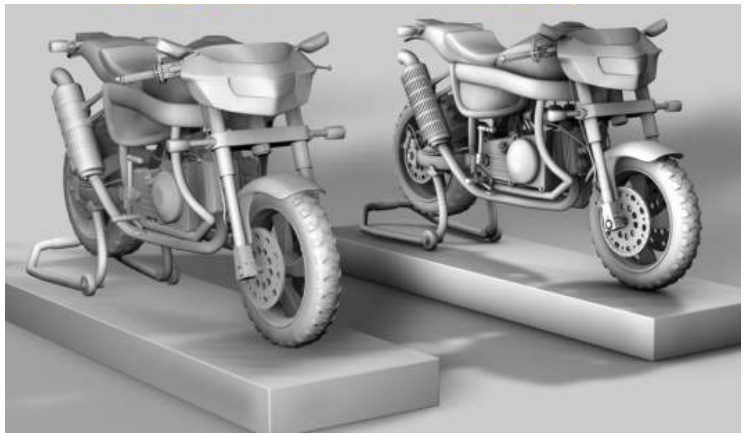
Shadow Mapping

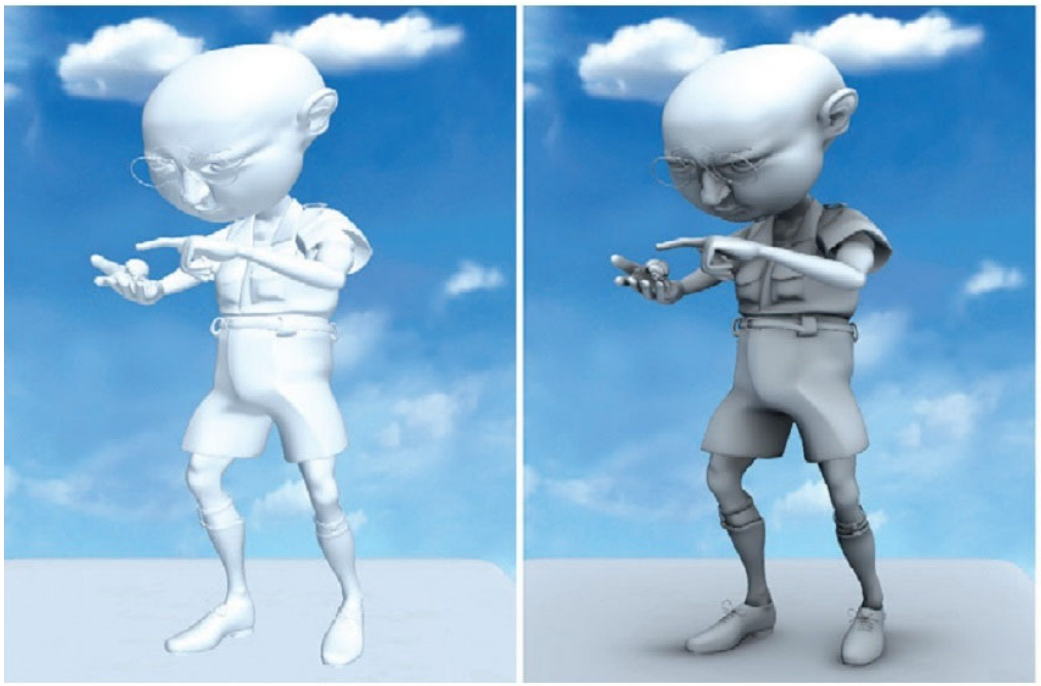
- Only depth needed
  - Everything one can render can cast a shadow ☺
- Predictable cost
  - Rendering the scene once ☺
- Shadow rendering is simple texture lookup ☺
  - Hardware Support
- Sampling artifacts ☹

Shadowing with Environmental Lighting

- So far: shading with known light position
- Now: shadowing by ambient light
- Independent of light/object orientation
- Darken surfaces which are partially visible in the environment
- Adds depth and contrast

without AO with AO

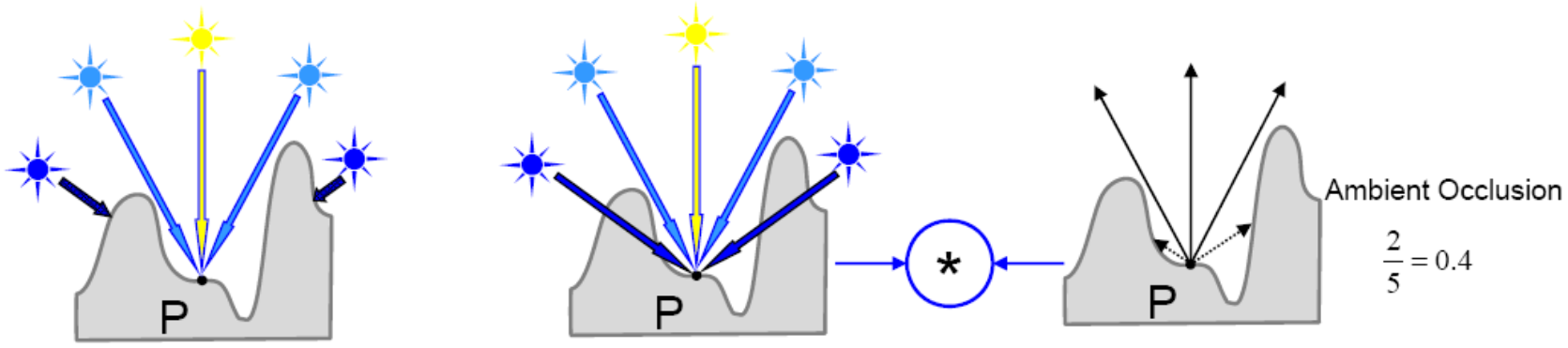




Ambient Occlusion

- Compute (per vertex)
  - Mean visibility $V_{AO} = 1 - AO$
  - Bent normal vector pointing into direction of average visibility
  - Used for advanced lighting instead of surface normal
- Weigh ordinary shading using mean visibility

$$I = (1 - AO)(k_a I_a + k_d I_d \max(\mathbf{n} \cdot \mathbf{l}, 0))$$



Mean Visibility

- Pre-computation = raycasting to find self-occlusions
- For each vertex
  - Cast *n* rays into the half sphere oriented by the normal
  - Count blocked rays *m*
  - Mean visibility $V_{AO} = 1 - m/n$
- Account for Lambert's law
  - Apply cosine distribution around normal vector
- To account for other nearby objects
  - Use ordinary raycasting with maximal distance
- Problem: raycasting from every point is expensive

Ambient Occlusion

(a) Gouraud shading

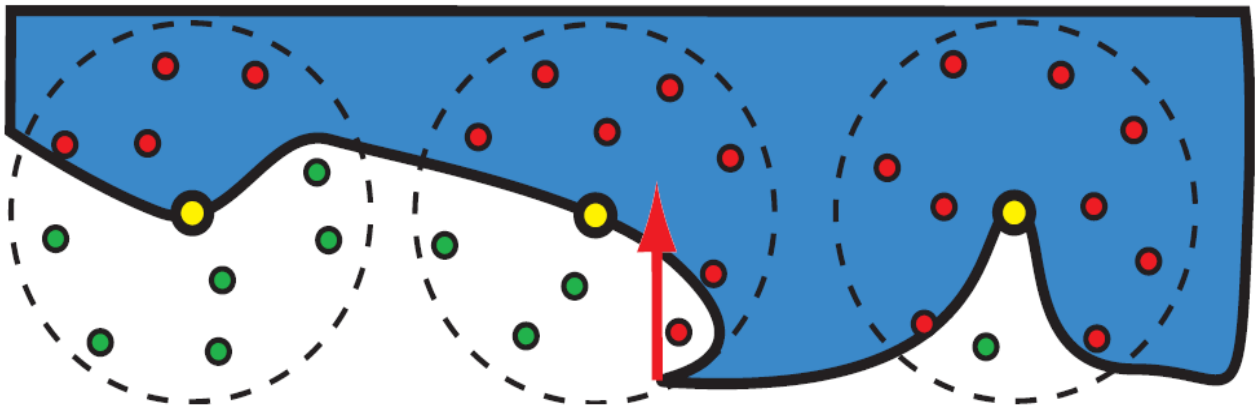
(b) Ambient Occlusion

(c) Ground truth

https://www.youtube.com/watch?v=9Fe1nYnvmiA

Screen Space Ambient Occlusion

- Z-buffer = approximation of surrounding
- Compute ambient occlusion from neighboring pixels
- Use random sampling of z-buffer
  - V_{AO} = Ratio occluded/unoccluded samples
  - L_{in} = incoming radiance
  - ω_i = incoming (=light source) direction
  - θ_i = angle to normal



Questions?