Rendu différé

Source cours de Steve Marschner

Light reflection physics

Radiometry redux

Power

Intensity power per unit solid angle

Irradiance power per unit area

Radiance power per unit (solid angle × area)

Sources of light

Point sources

- intensity
- can be directionally varying—spotlights

Area sources

- radiance
- can be spatially varying

Directional sources

- irradiance (normal irradiance)

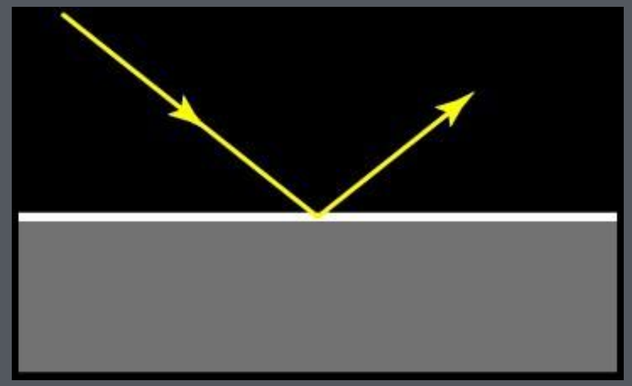
Environment lighting

- radiance (usually spatially varying)
- sun-sky models

Simple kinds of scattering

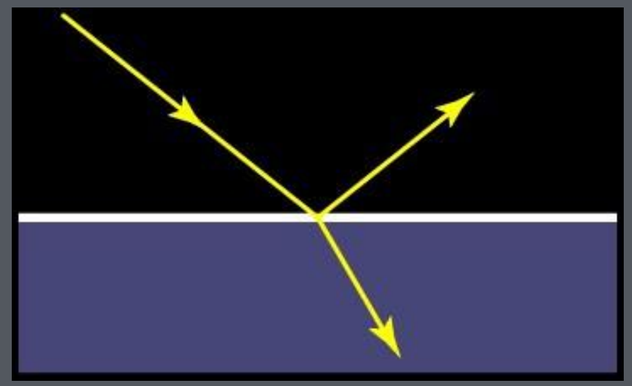
Ideal specular reflection

- incoming ray reflected to a single direction
- mirror-like behavior
- arises at smooth surfaces



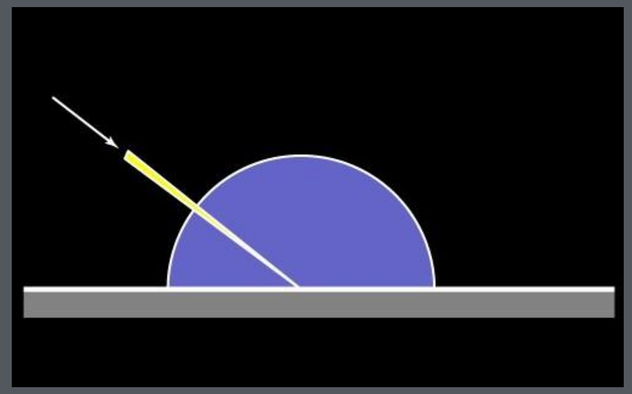
Ideal specular transmission

- incoming ray refracted to a single direction
- glass-like behavior
- arises at smooth dielectric (nonmetal) surfaces

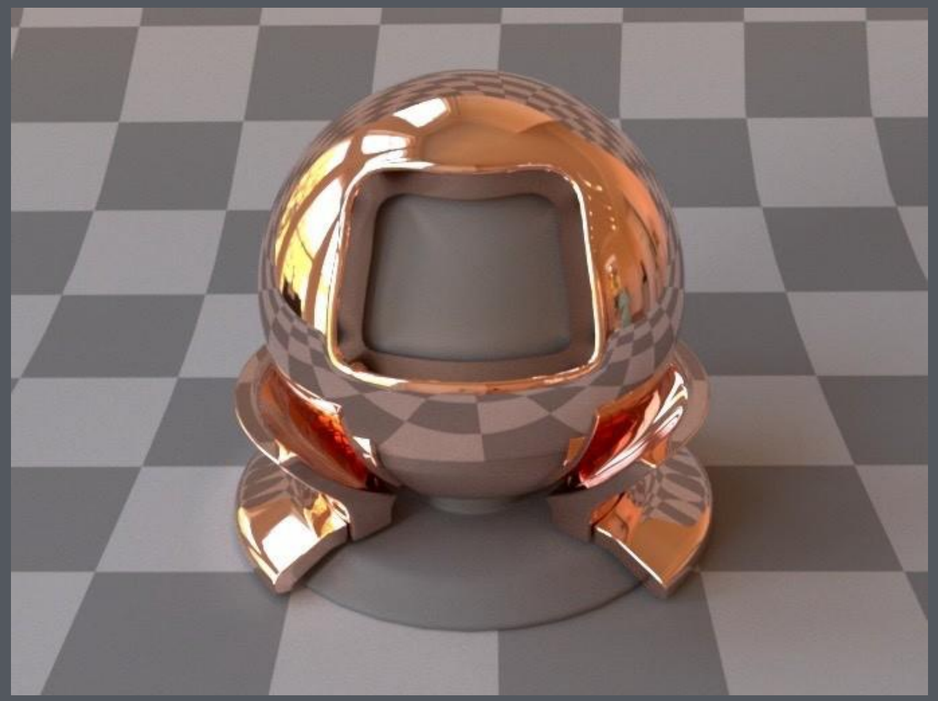


Ideal diffuse reflection or transmission

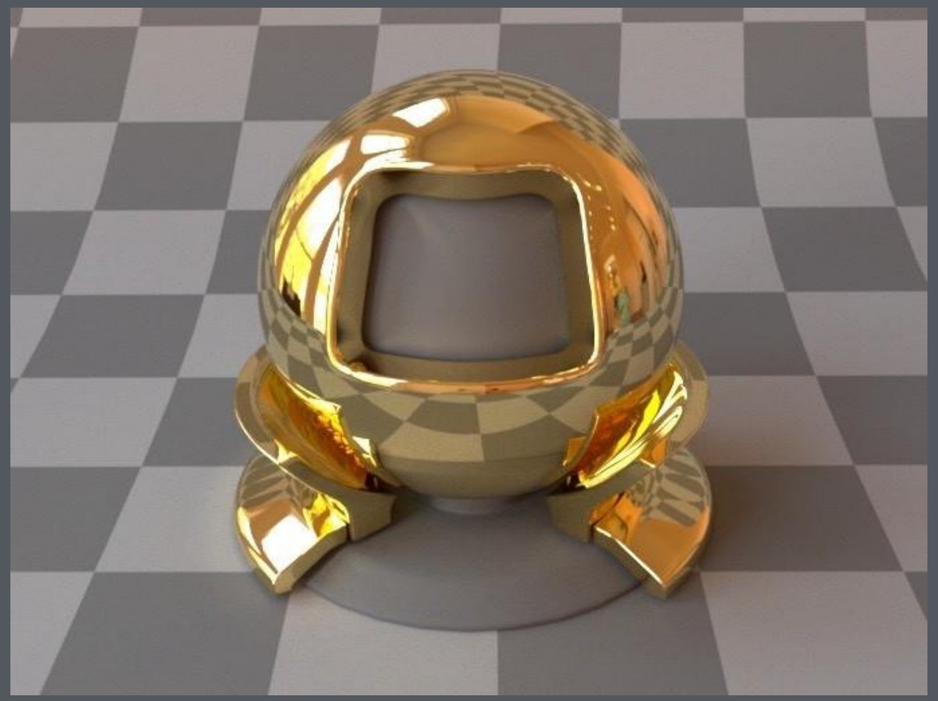
- outgoing radiance independent of direction
- arises from subsurface multiple scattering



Ideal specular reflection from metals



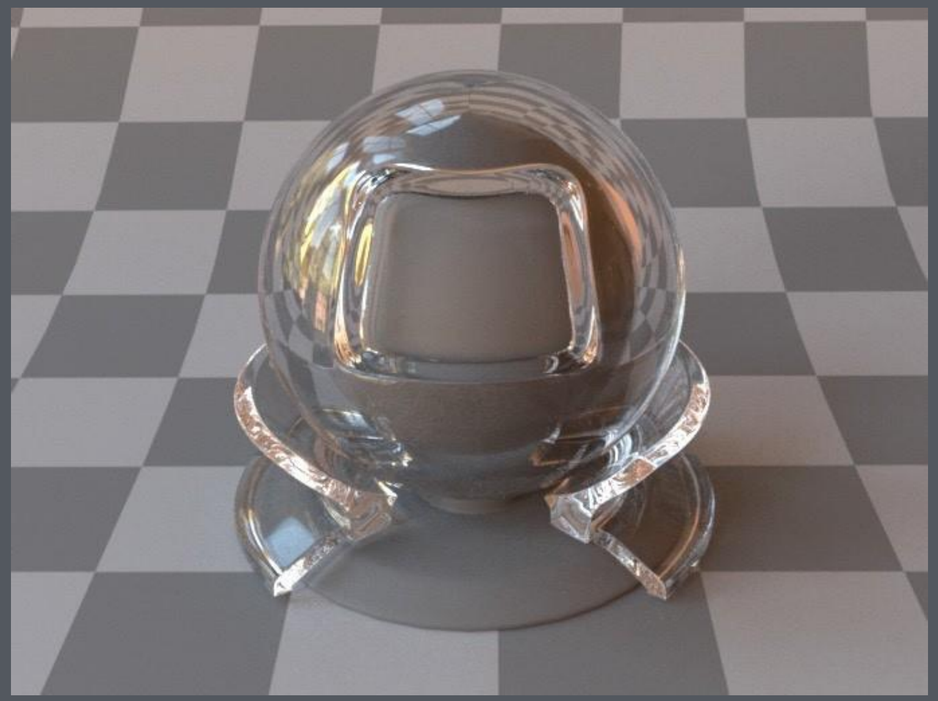
Cu



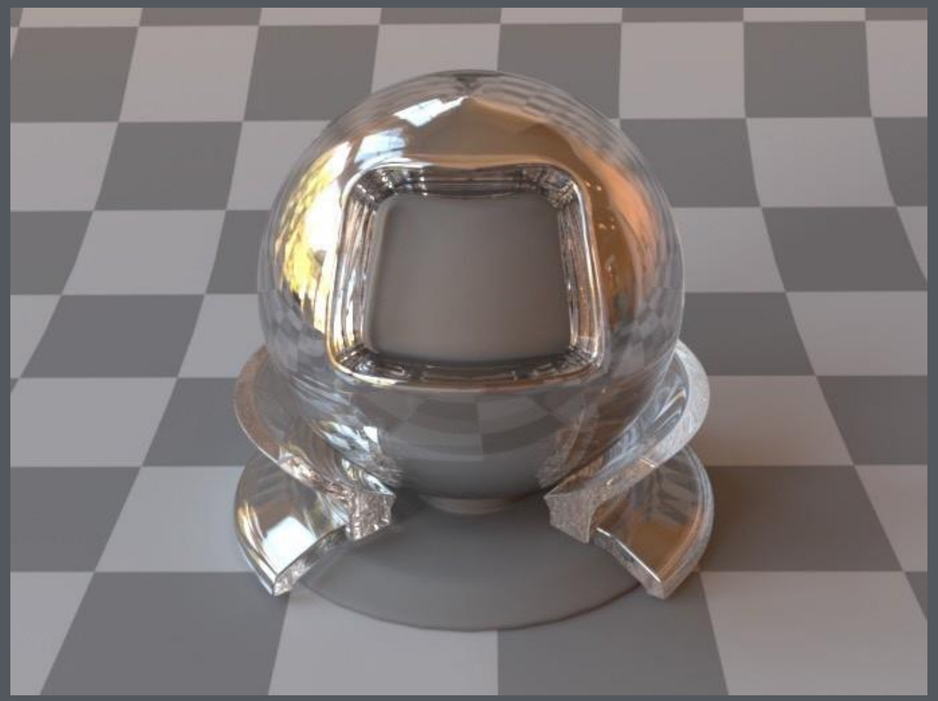
Au

Wenzel Jakob / Mistuba

Ideal reflection and transmission from smooth dielectrics



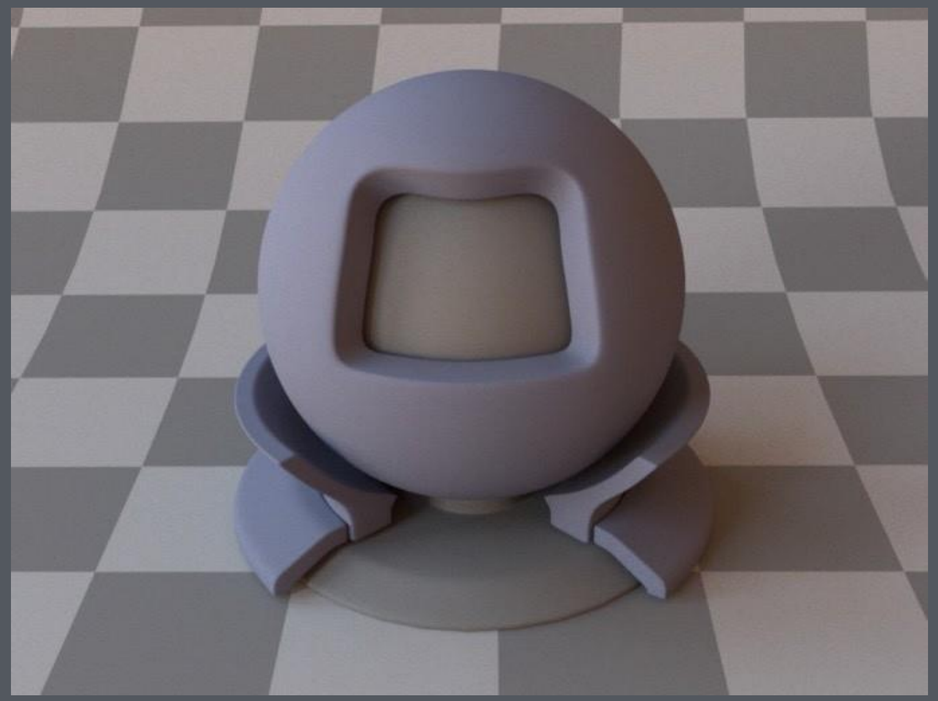
Water (ior = 1.33)



Diamond (ior = 2.4)

Wenzel Jakob / Mistuba

Two diffuse surfaces

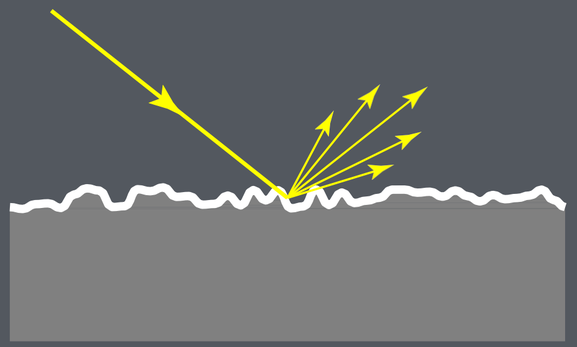


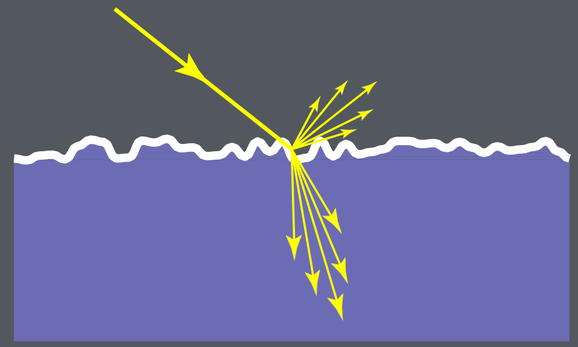
Wenzel Jakob / Mistuba

More complex scattering

Rough interfaces

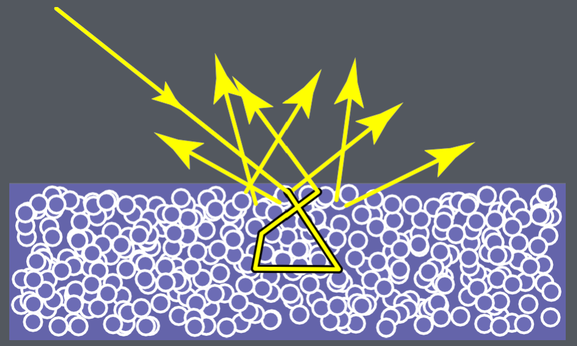
- metal interfaces: blurred reflection
- dielectric interfaces: blurred transmission

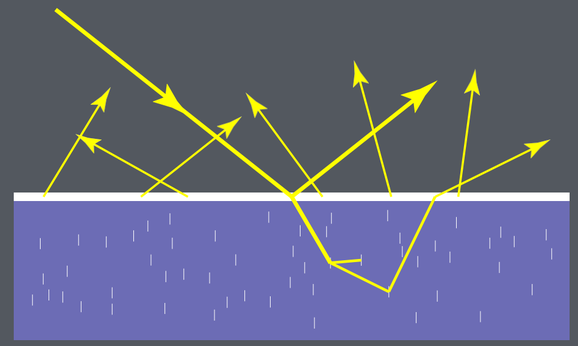




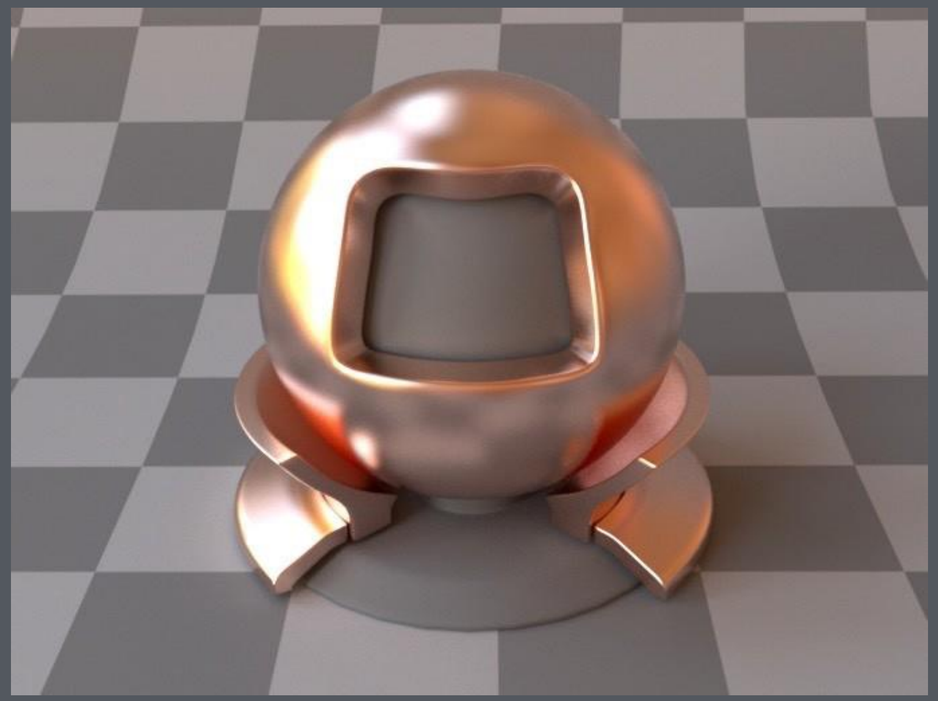
Subsurface scattering

- liquids—milk, juice, beer, …
- coatings—paint, glaze, varnish, …
- natural materials—wood, marble, …
- biological materials—skin, plants, …
- low optical density leads to *translucency*

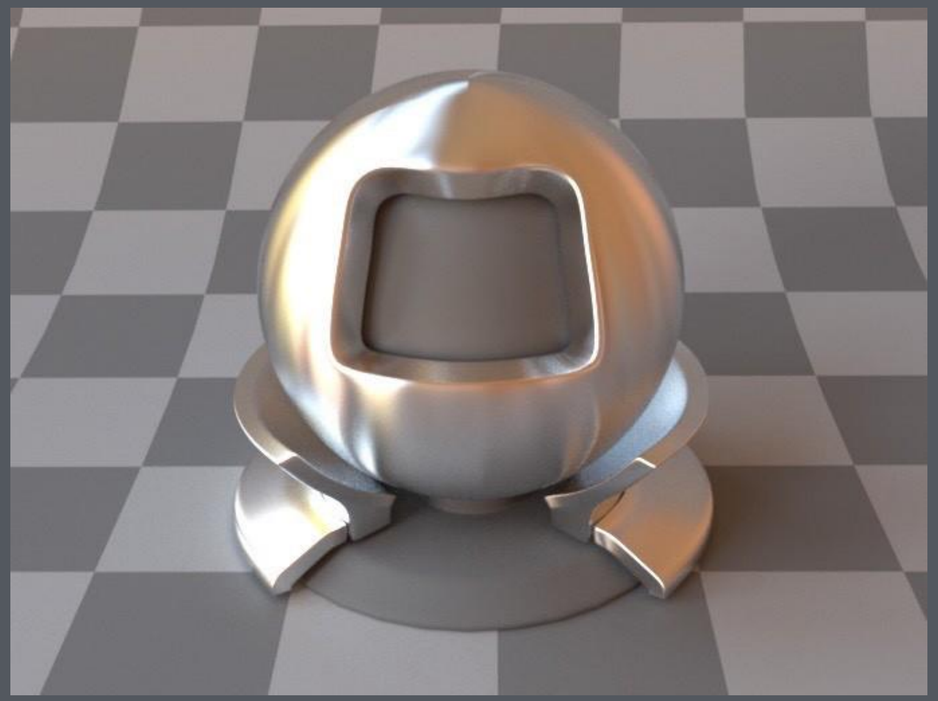




Reflection from rough metal interfaces



Cu (α = 0.1)



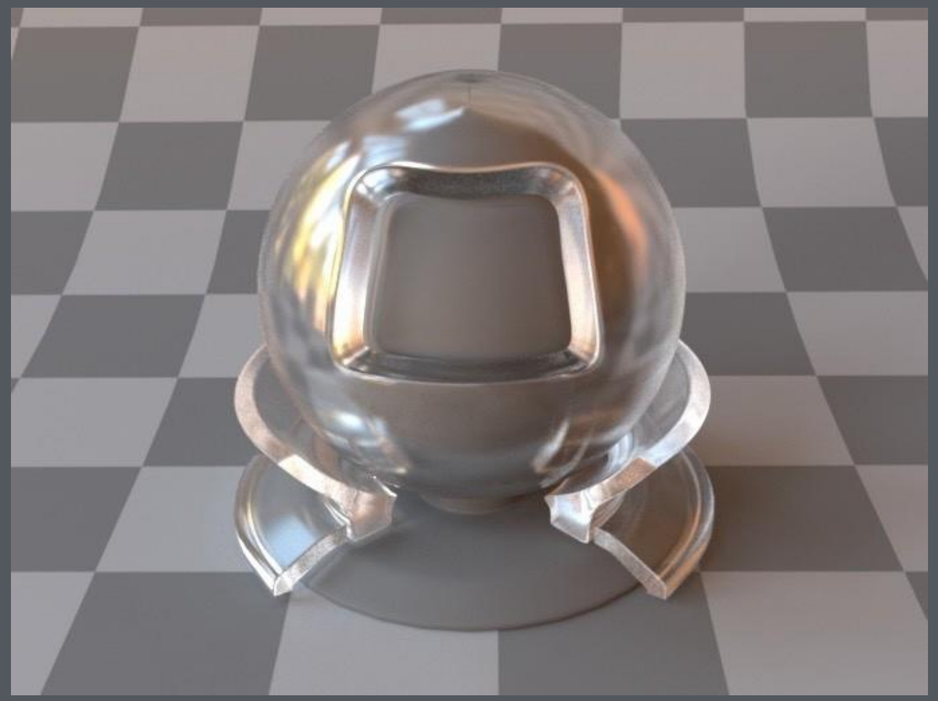
Al (anisotropic)

Wenzel Jakob / Mistuba

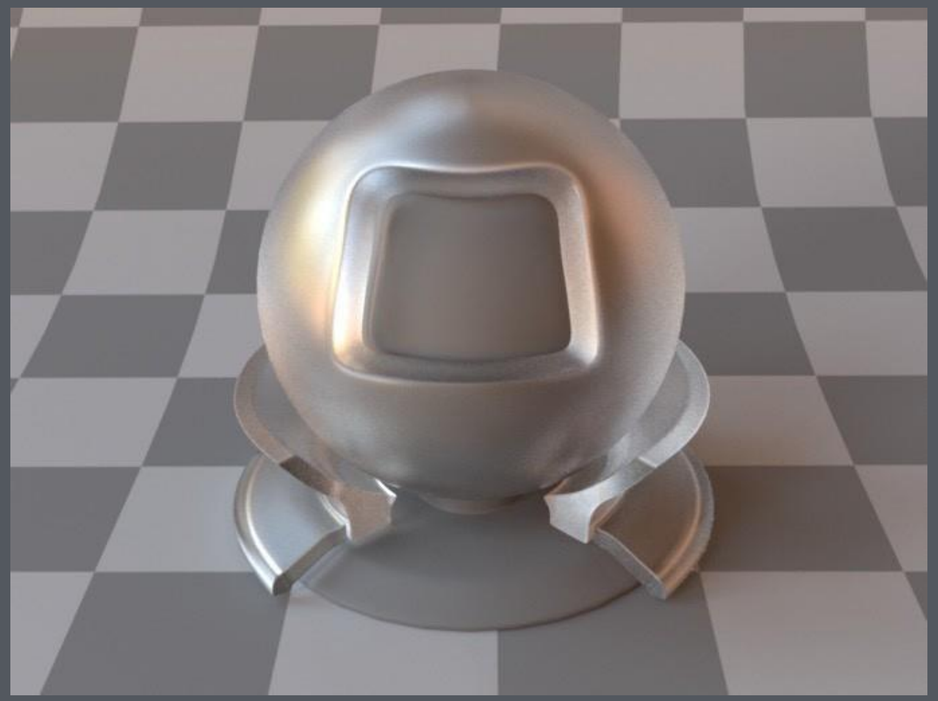
Ling-qi Yan & Milos Hasan | work in progress

Ling-qi Yan & Milos Hasan | work in progress

Reflection and refraction at rough dielectric interfaces



Anti-glare glass (α = 0.02)



Etched glass (α = 0.1)

Wenzel Jakob / Mistuba

Translucent materials

Wenzel Jakob / Mistuba

“skim milk”

low optical density

Wenzel Jakob / Mistuba

high optical density

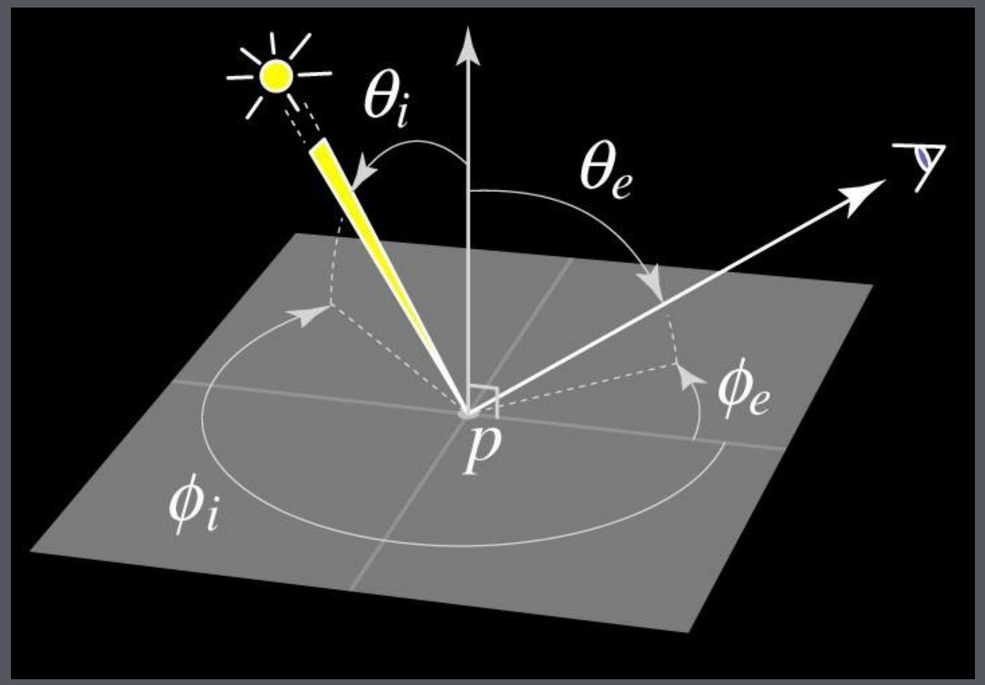
Modeling complex scattering

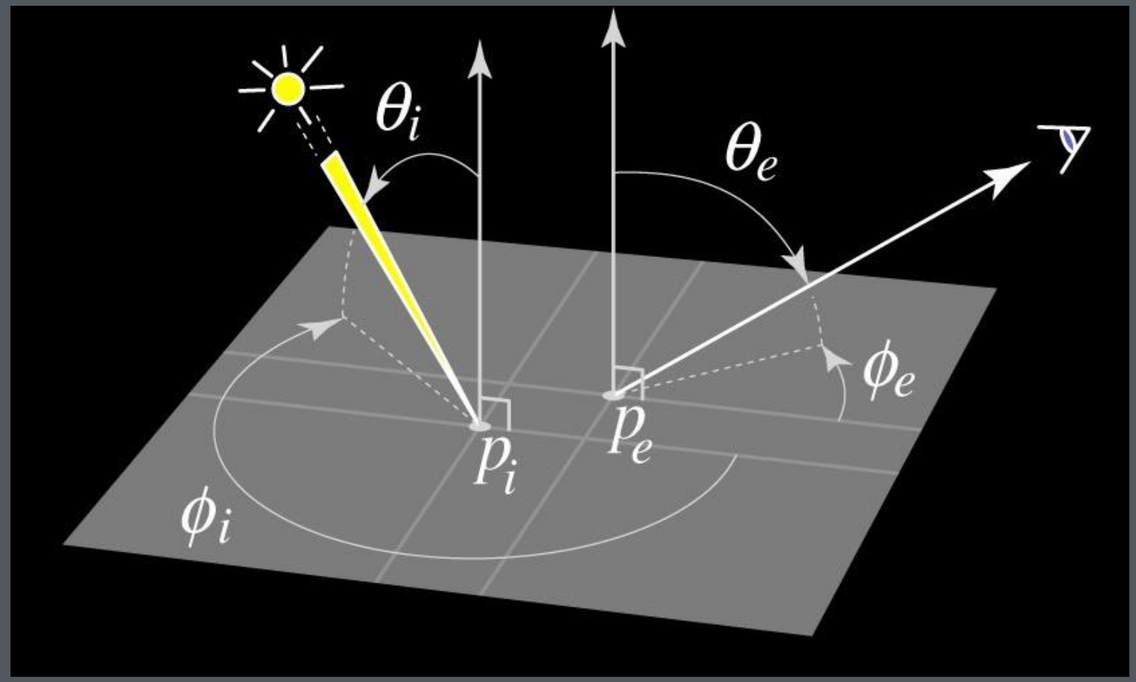
Opaque materials

- reflection: bidirectional reflectance distribution function (BRDF)
- transmission: bidirectional transmittance distribution function (BTDF)
- both: bidirectional scattering distribution function (BSDF)

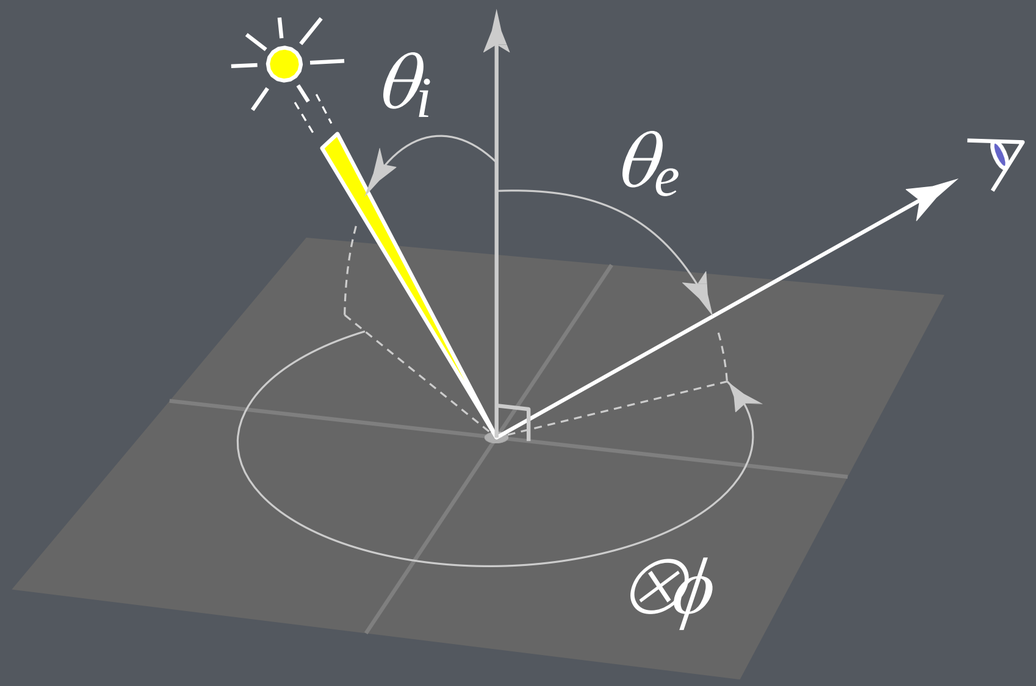
Translucent materials

- bidirectional subsurface scattering reflectance distribution function (BSSRDF)

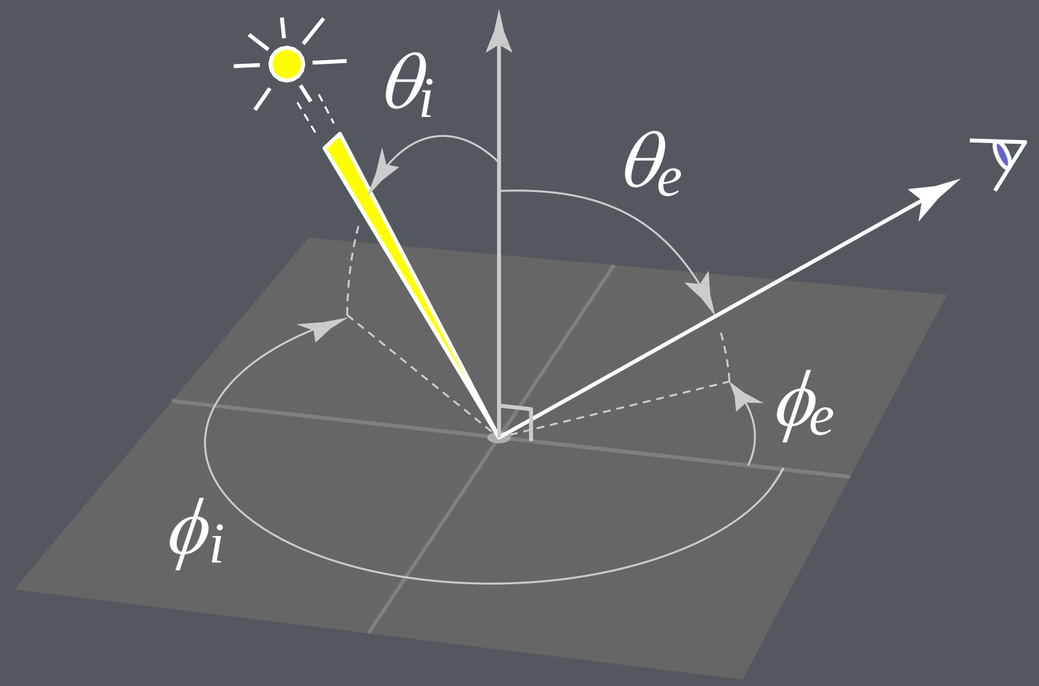




Isotropy vs. anisotropy

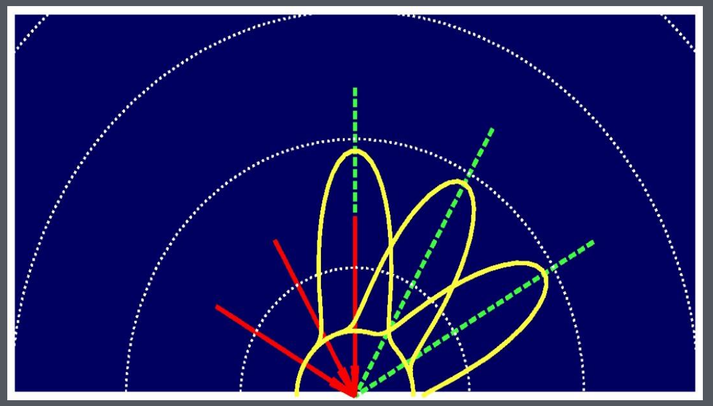


isotropic



anisotropic

Types of BRDF/BSDF models

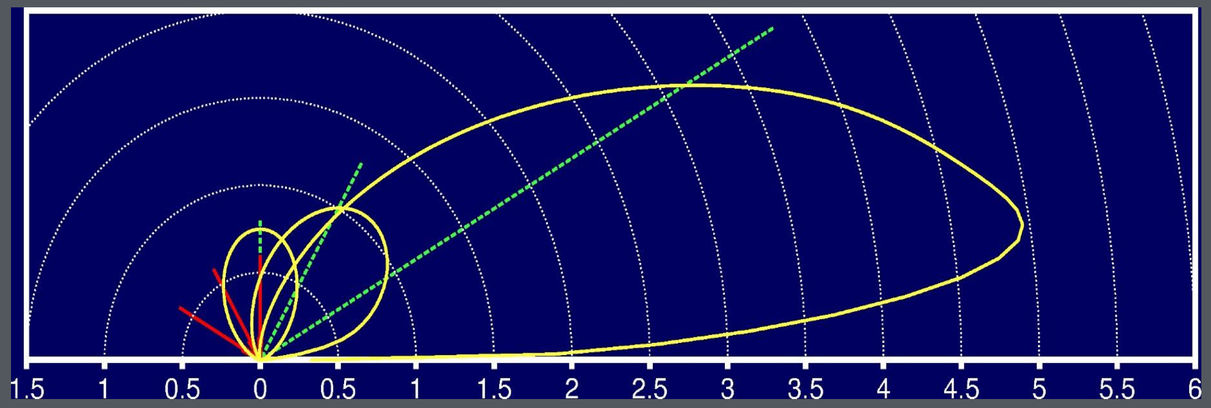


Ad hoc formulas

- e.g. Blinn-Phong

Physics-based analytical models

- Lambertian
- Microfacet-based models
- Kirchhoff-based models



Measured data

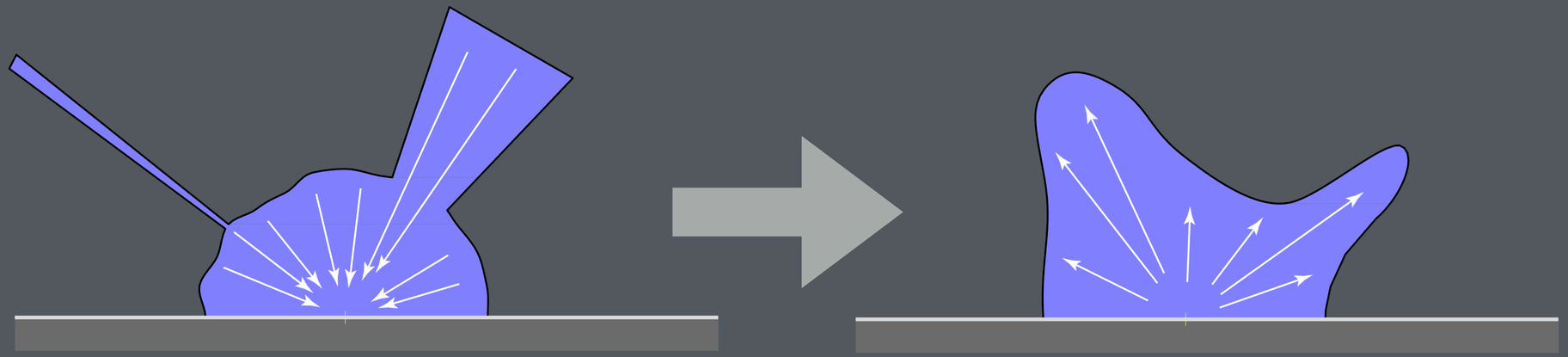
- tables of data from pointwise BRDF measurements
- image-based BRDF measurements

Light reflection in shaders

Light reflection: full picture

all types of reflection reflect all types of illumination

- diffuse, glossy, mirror reflection
- environment, area, point illumination



incident distribution
(function of direction)

reflected distribution
(function of direction)

Categories of illumination

	diffuse	glossy	mirror
indirect	soft indirect illumination	blurry reflections of other objects	reflected images of other objects
environment	soft shadows	blurry reflection of environment	reflected image of environment
area	soft shadows	shaped specular highlight	reflected image of source
point/directional	hard shadows	simple specular highlight	point reflections

= easy to compute using standard shaders

How to compute shading

Basic case: point or directional lights; diffuse or glossy BRDF

Type in BRDF model, plug in illumination and view direction

- can write down model in world space, use world-space vectors
- can write down model in surface frame, transform vectors
- really not different

Subtleties are all about what frame to use for shading

Interpolated shading

Coarse triangle meshes are fast

Discontinuities are bad

Therefore: interpolate geometric quantities across triangles

- goal: shading is smooth across edges

What do we interpolate?

- what do we need to compute shading?

Shading frames

When we carry around a normal, we are defining a tangent plane

- interpolated normal defines an approximate, smoothly varying tangent plane

For some purposes, the tangent plane is enough

- e.g. computing shading for isotropic BRDFs
- any coordinate system conforming to the normal is equally good

In other cases, need a complete frame

- whenever directions within the plane are inequivalent
- e.g. anisotropic BRDFs
- e.g. tangent-frame normal maps

What to interpolate

Need plane: can just interpolate a normal

Need frame: interpolate enough data to define a tangent frame

One and a half vectors rounds up to two

- normal and one tangent vector
- two tangent vectors

Rebuilding a frame from the vectors

- worry about handedness matching texture coordinates (or not)
- orthonormality gets broken by interpolation (when does that matter?)

What you need for shading

When/why you need full frames

- when you care (or not) what the orientation is
- when you care (or not) about orthonormality

What to interpolate

- underlying math question: representation of frames
- representations that behave well under interpolation

How to author orientation

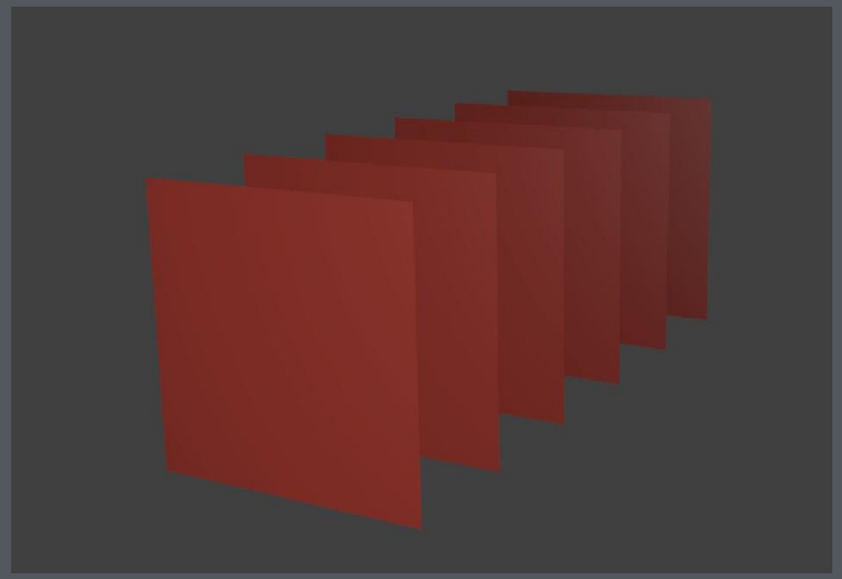
- with maps
- by following a parameterization

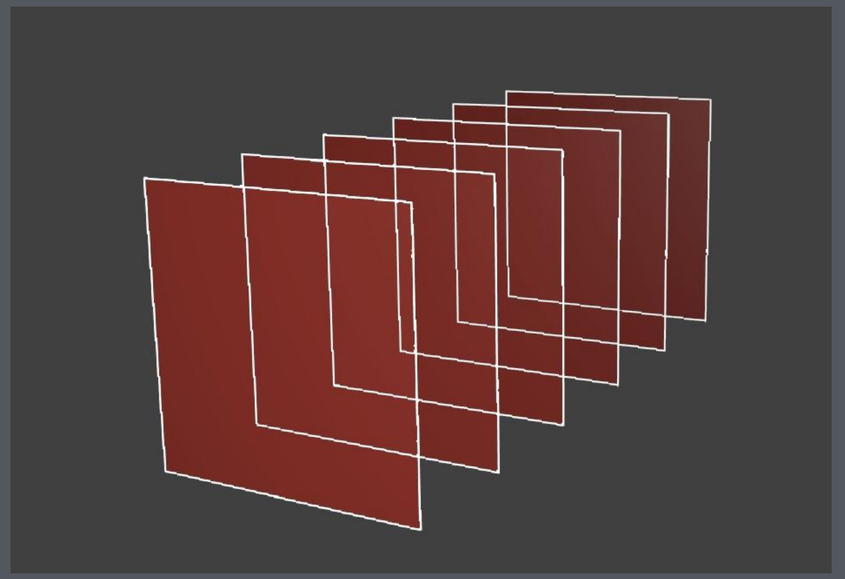
How to deal with corner cases

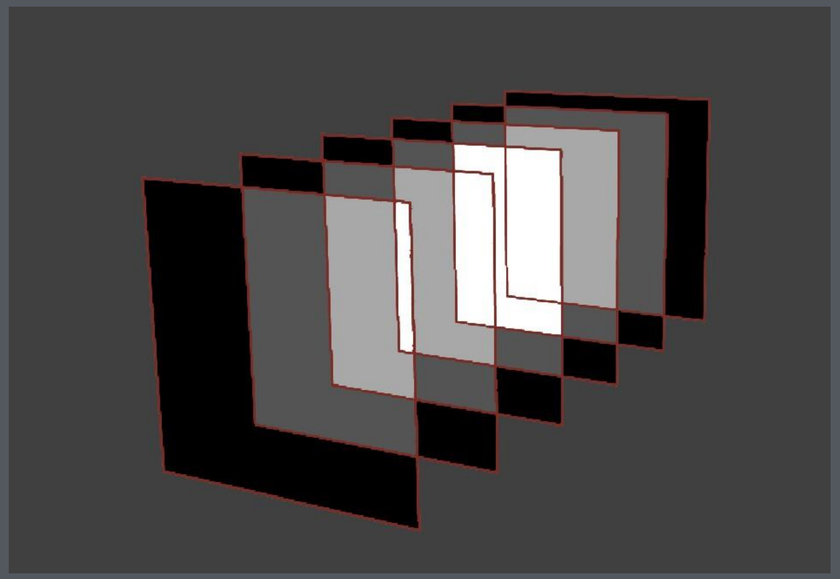
Rendering: forward shading

```
for each object in the scene
   for each triangle in this object
      for each fragment f in this triangle

         gl_FragColor = shade(f)

         if (depth of f < depthbuffer[x, y])
            framebuffer[x, y] = gl_FragColor
            depthbuffer[x, y] = depth of f
         end if

      end for
   end for
end for
```

Problem: Overdraw







Problem: lighting complexity

Just Cause 3 | Avalanche Studios

Missed opportunity: spatial processing

Fragments cannot talk to each other

- a fundamental constraint for performance

Many interesting effects depend on neighborhood and geometry

- bloom
- ambient occlusion
- motion blur
- depth of field
- edge-related rendering effects

Deferred shading approach

First render pass

- draw all geometry
- compute material- and geometry-related inputs to lighting model
- don’t compute lighting
- write shading inputs to intermediate buffer

Second render pass

- don’t draw geometry
- use stored inputs to compute shading
- write to output

Post-processing pass (optional, can also be used with fwd. rendering)

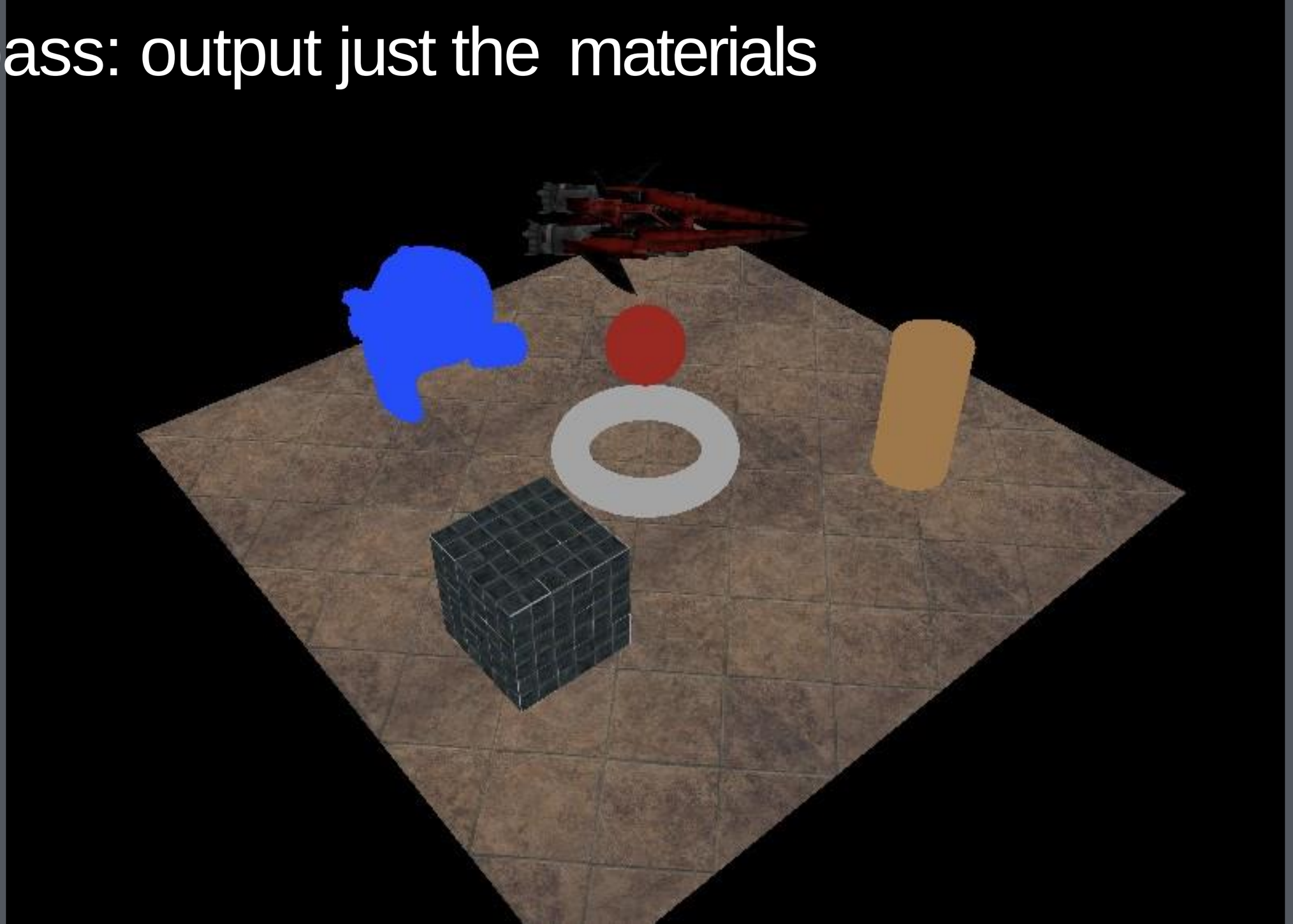
- process final image to produce output pixels

Deferred shading step 1

```
for each object in the scene
   for each triangle in this object
      for each fragment f in this triangle

         gl_FragData[…] = material properties of f
         if (depth of f < depthbuffer[x, y])
            gbuffer[…][x, y] = gl_FragData[…]
            depthbuffer[x, y] = depth of f
         end if

      end for
   end for
end for
```

Here we're making use of an OpenGL feature called "Multiple Render Targets" in which the familiar gl_FragColor is replaced by an array of values, each of which is written to a different buffer.

First pass: output just the materials

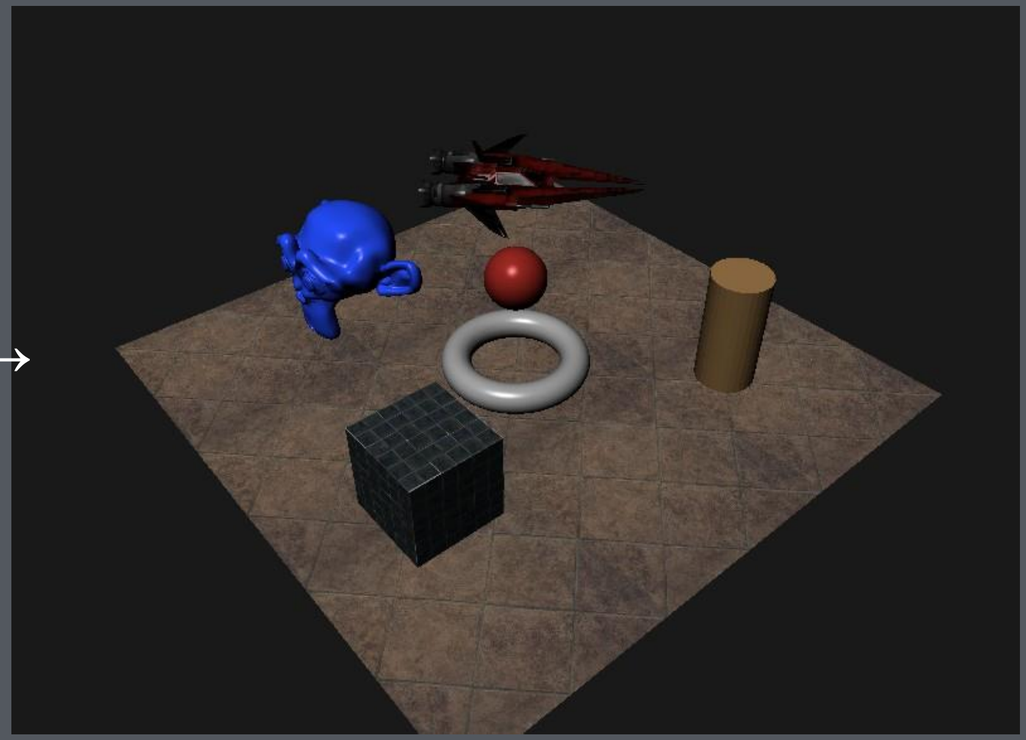


Deferred Shading Step 2

```
for each fragment f in the gbuffer
   framebuffer[x, y] = shade (f)
end for
```

Key improvement: shade (f) only executed for visible fragments.

Output is the same →

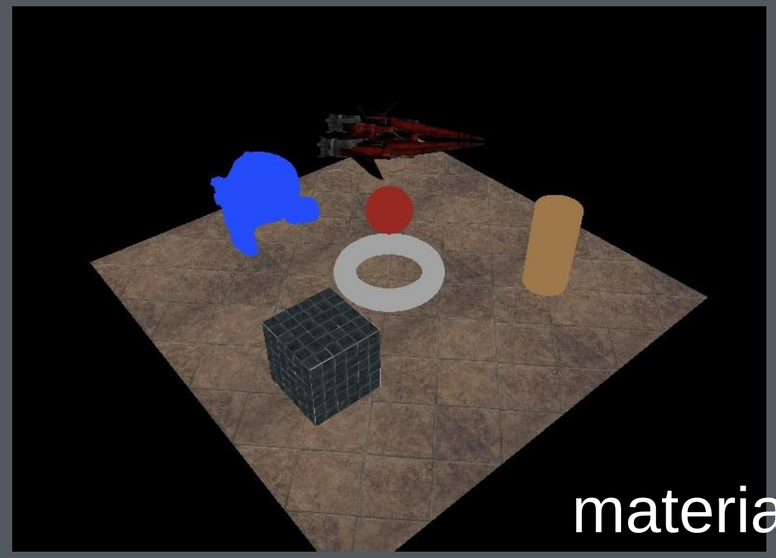


```
for each object in the scene
   for each triangle in this object
      for each fragment f in this triangle

         gl_FragData[…] = material properties of f
         if (depth of f < depthbuffer[x, y])
            gbuffer[…][x, y] = gl_FragData[…]
            depthbuffer[x, y] = depth of f
         end if

      end for
   end for
end for
```

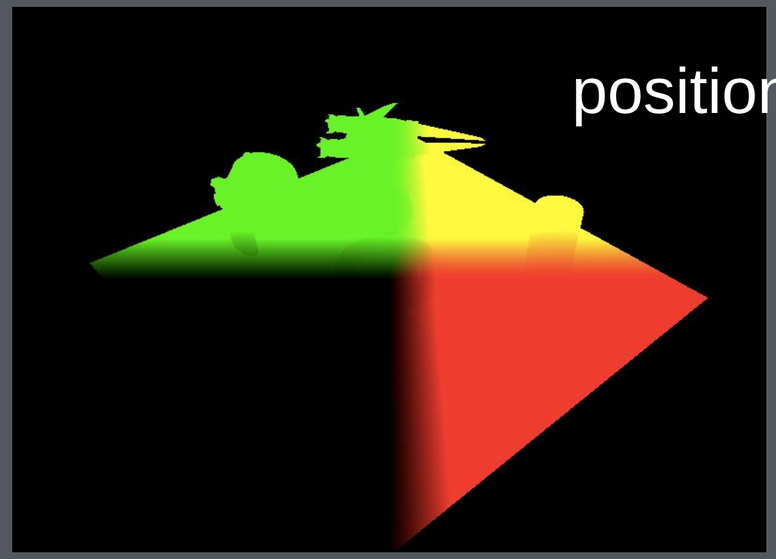
```
for each fragment f in the gbuffer
   framebuffer[x, y] = shade (f)
end for
```

Here we're making use of an OpenGL feature called "Multiple Render Targets" in which the familiar gl_FragColor is replaced by an array of values, each of which is written to a different buffer.

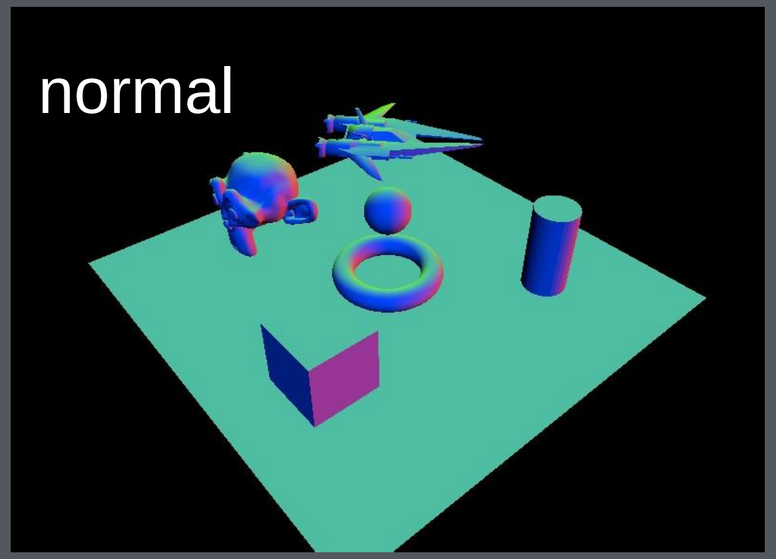
G-buffer: multiple textures



material properties



position



normal

The übershader

Shader which computes lighting based on g-buffer: has code for all material/ lighting models in a single huge shader.

```
shade (f) {
    result = 0;
    if (f is Lambertian) {
        for each light
            result += (n . l) * diffuse;
        end for
    } else if (f is Blinn-Phong) {
        ...
    } else if (f is ...) {
        ...
    }
    return result;
}
```

Übershader inputs

Need access to all parameters of the material for the current fragment:

- Blinn-Phong: kd, ks, n
- Microfacet: kd, ks, alpha
- etc.

Also need fragment position and surface normal

Solution: write all that out from the material shaders:

```
{outputs} = {f.material, f.position, f.normal}
if (depth of f < depthbuffer[x, y])
   gbuffer[x, y] = {outputs}
   depthbuffer[x, y] = depth of f
end if
```

Deferred lighting

Single-pass render has to consider all lights for every fragment

- much wasted effort since only a few lights probably contribute
- batching geometry by which lights affect it is awkward
- straight 2-pass deferred has same problem

With deferred shading, fragments can be visited in subsets

- for each light, draw bounds of (significantly) affected volume
- only compute shading for fragments covered by that
- with depth/stencil games, can only shade fragments inside the volume

Power of Deferred Shading

Can do any image processing between step 1 and step 2!

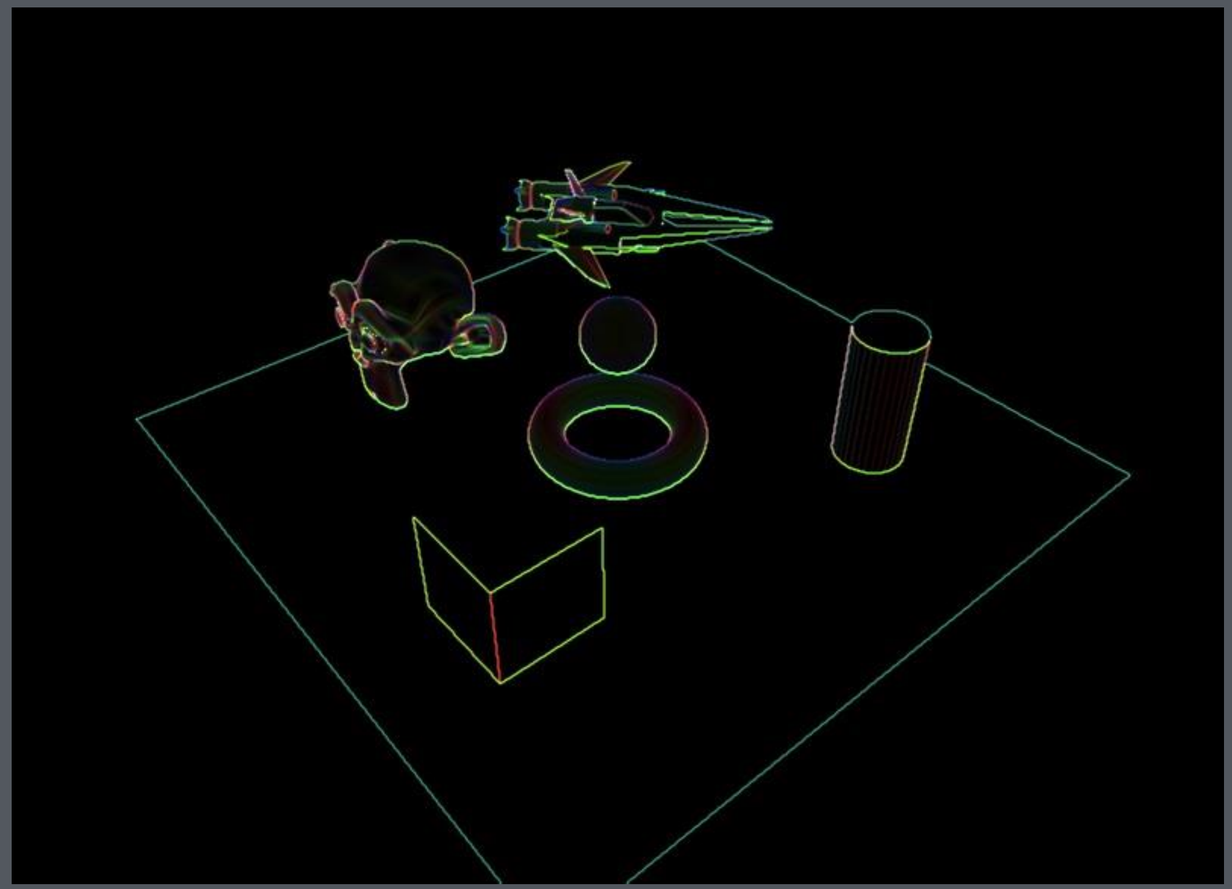
- Recall: step 1 = fill g-buffer, step 2 = light/shade
- Could add a step 1.5 to filter the g-buffer

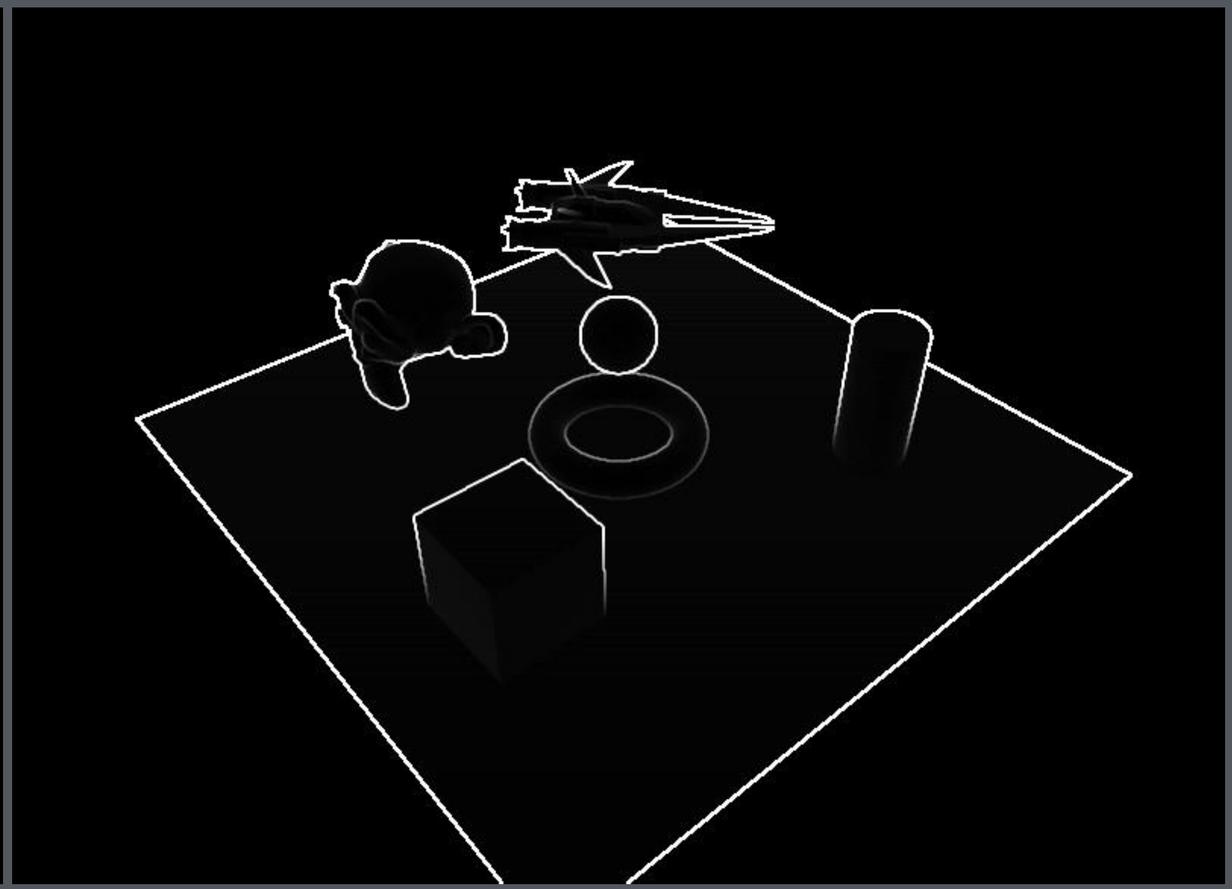
Examples:

- silhouette detection for artistic rendering
- screen-space ambient occlusion
- denoising based on bilateral filter using geometric info

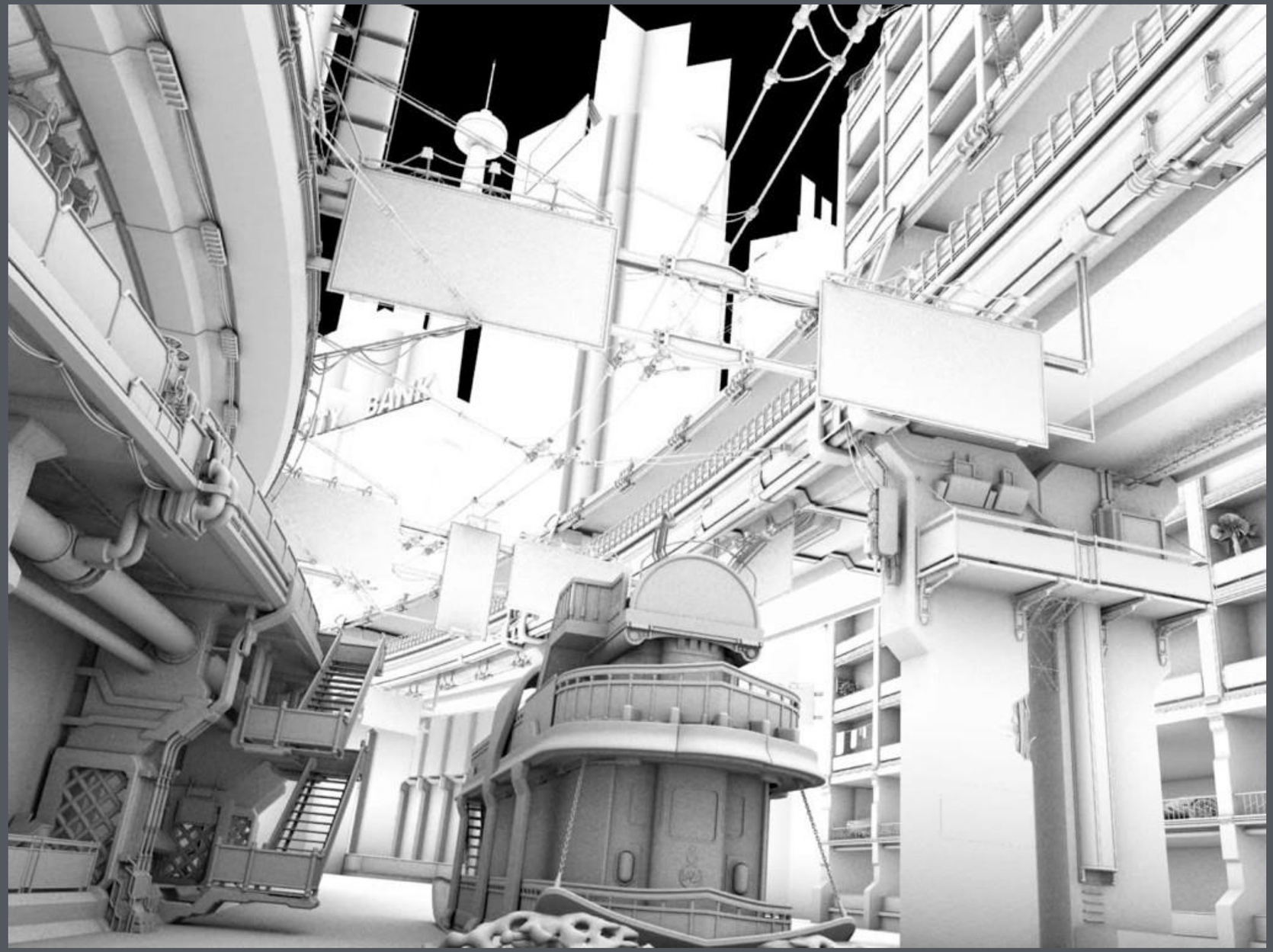
Silhouette detection

By differentiating the depth buffer, can locate silhouettes and creases





Amient occlusion



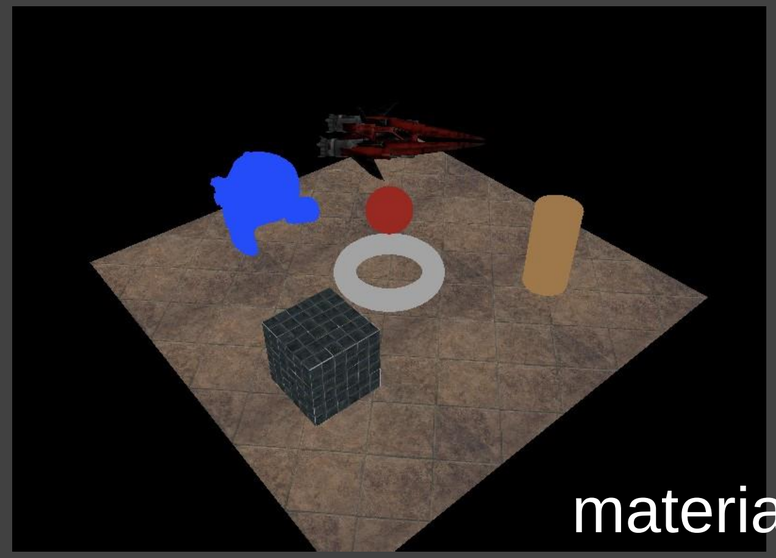
Denoising

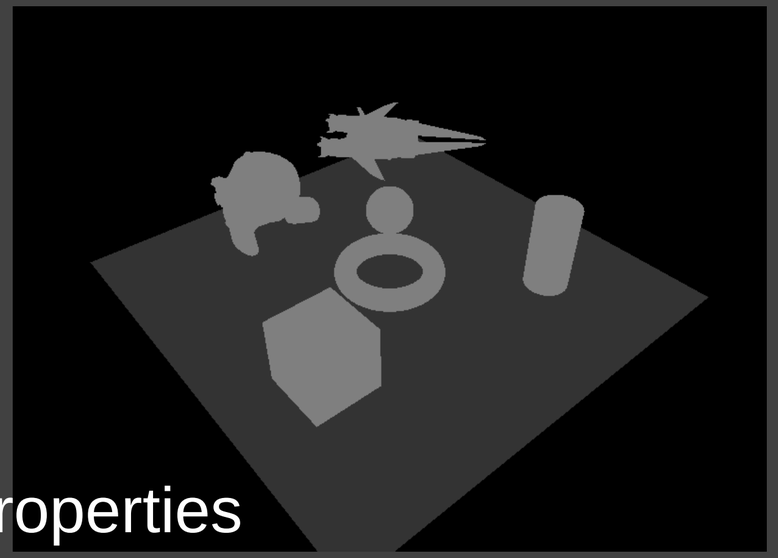
[Mara et al. HPG 2017]

Denoising

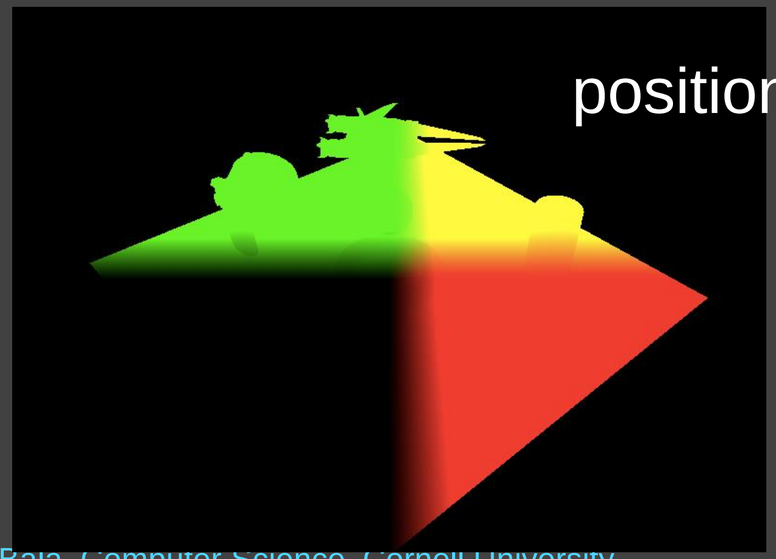
[Mara et al. HPG 2017]

How to Fill the Buffers?

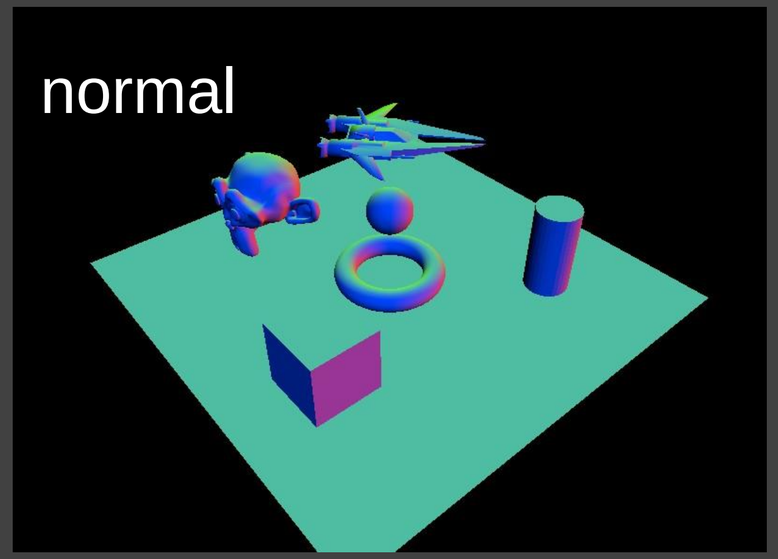




material properties



position



normal

Limitations of Deferred Shading

Each pixel in the g-buffer can only store material and surface info for a single surface.

- blending/transparency is difficult
- antialiasing is a different ball game

For transparency: a “hybrid” renderer

- deferred shading for opaque objects, forward shading for translucent objects
- allows translucent geometry to know about opaque geometry behind it

For antialiasing: smart blurring

- use what is in the g-buffers to blur along but not across edges

Hybrid Rendering

Image credit: random guy on internet: http://vimeo.com/14337919

Antialiasing

Single shading sample per pixel

Reconstruct by blending nearby samples

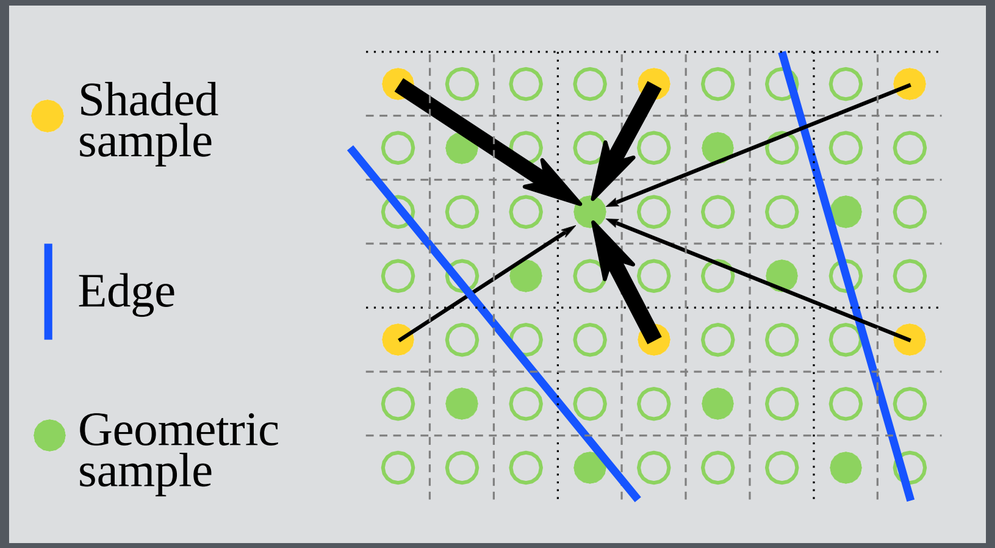
- select them by looking for edges
  (Morphological AA [Reshetov 09])
- learn about edges using multisample depth
  (Subpixel Reconstruction AA [Chajdas et al. 11])

MLAA

[Reshetov 09]

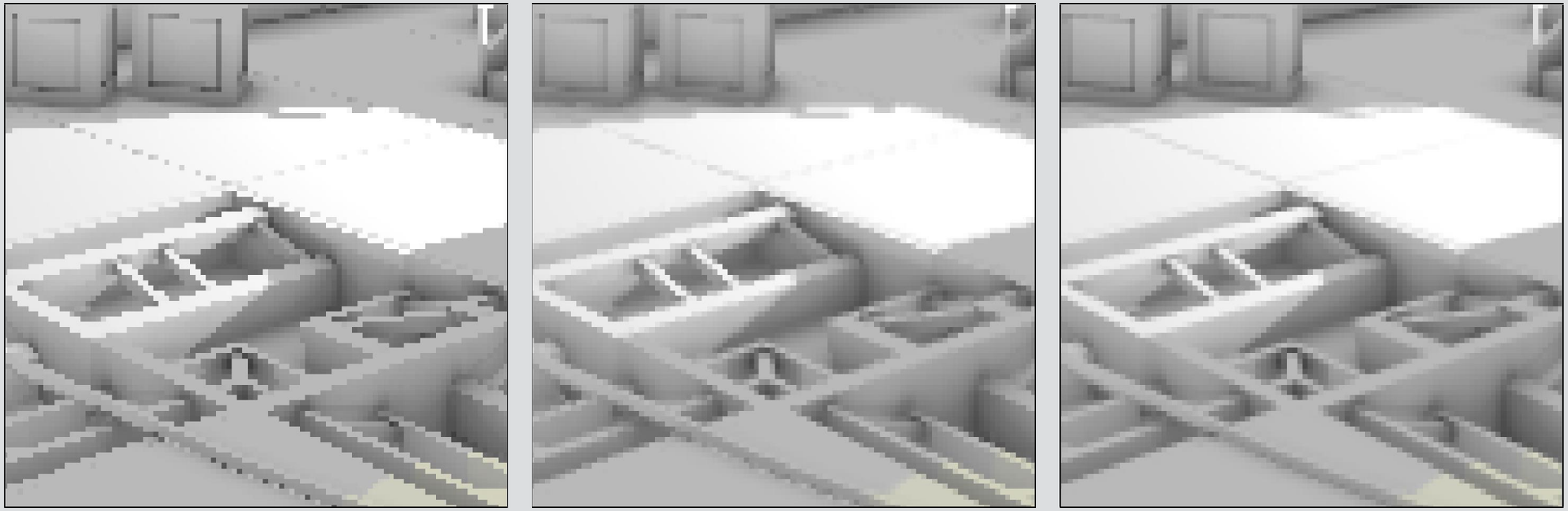
Detects borders in the resulting image and then finds specific patterns in these.
Anti-aliasing : blending pixels in these borders, according to the pattern and their position within the pattern

SRAA



Similar Time !

Similar Quality !



(a) 1⇥ Shading + Box (poor, fast)

(b) **NEW: 1⇥ Shading + SRAA (good, fast)**

(c) 16⇥ Shading + Box (good, slow)

[Chajdas et al. 11]

Summary: Deferred Shading

Pros

- Store everything you need in 1st pass
  - normals, diffuse, specular, positions,...
  - G-buffer
- After z-buffer, can shade only what is visible

Cons:

- transparency (only get one fragment per pixel)
- antialiasing (multisample AA not easy to adapt)

Standard game engines provide both forward and deferred paths

How to do all this in OpenGL

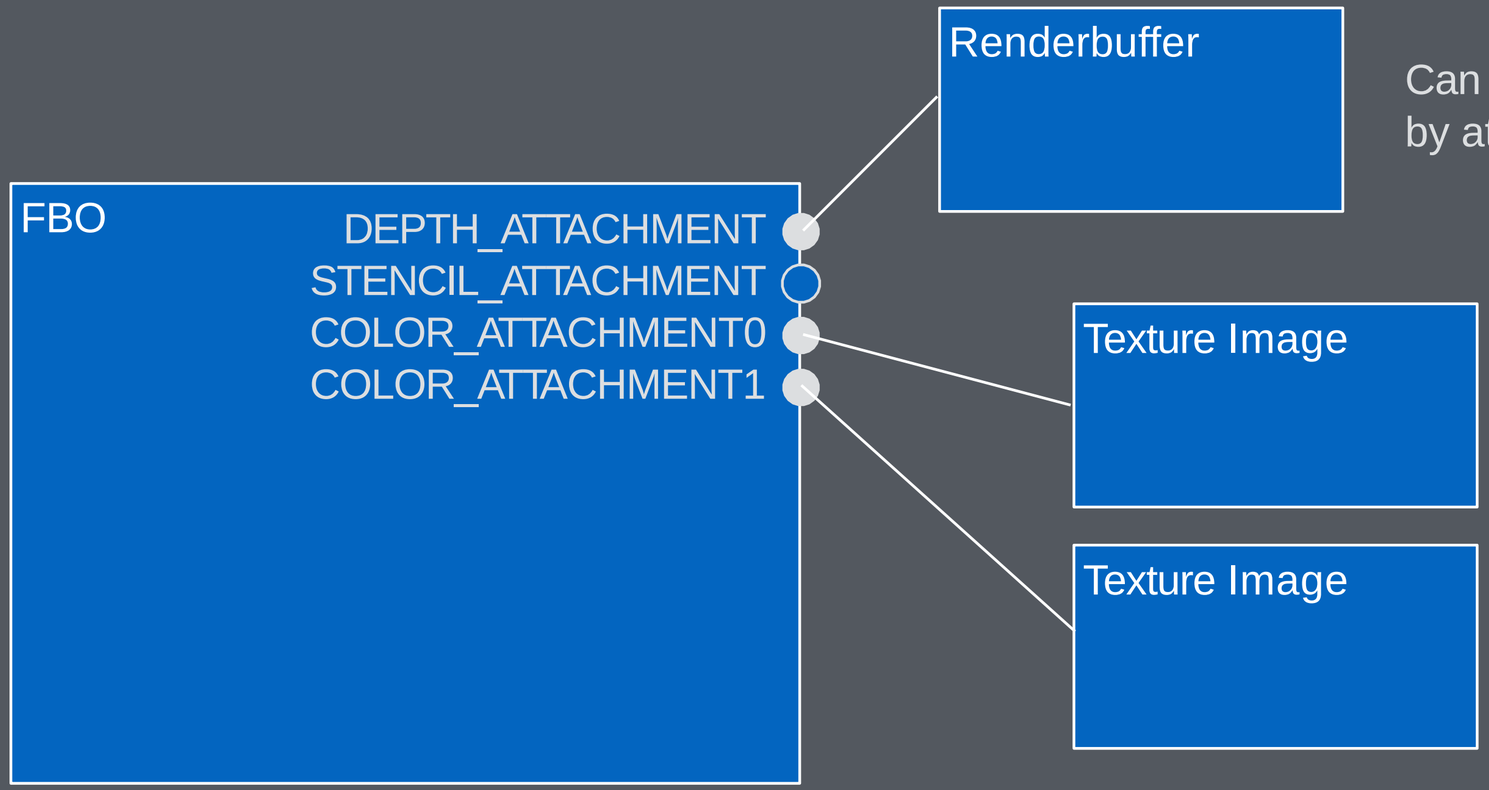
When you first fire up OpenGL, all fragment shader output is written to the framebuffer that shows up in your window

- this is the default framebuffer
- it actually can contain multiple buffers: front and back for double-buffering; left and right for stereo/HMD devices. You can control where fragment shader output goes using glDrawBuffer()

For deferred shading and other multi-pass methods, you instead create a (non-default) *Framebuffer Object* (FBO)

- you *attach* images to the FBO to receive fragment shader output
- color attachments (variable number) receive color data from gl_FragData[…] (gl_FragColor is just an alias for gl_FragData[0])
- a depth attachment is required for *z*-buffering to function; stencil attachments are also possible

Framebuffer Object

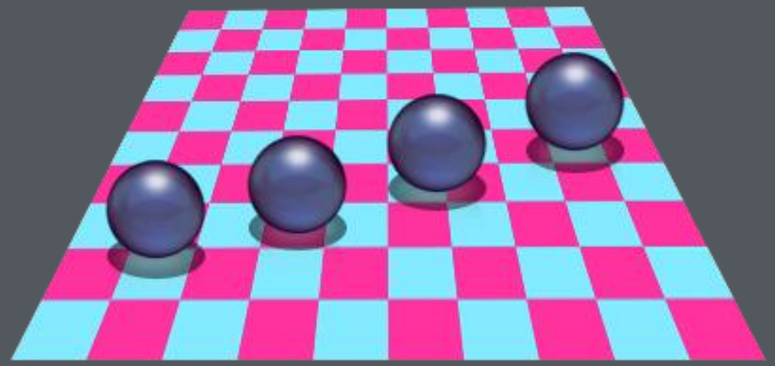


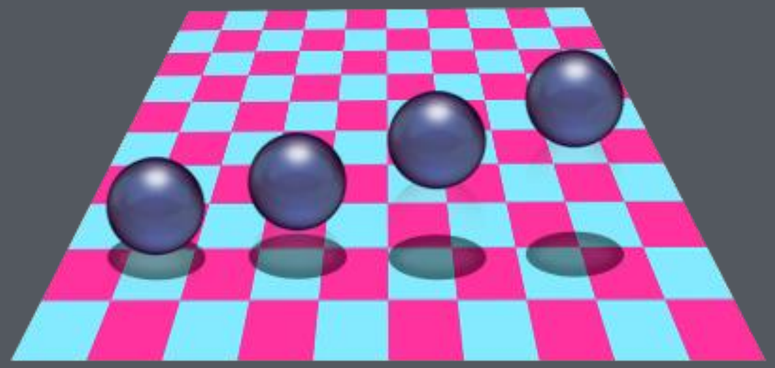
Can only be used by attaching to an FBO

Can be attached to an FBO, filled with data, and later sampled as a texture in another shading pass

All these images have to match in size!

Shadows as depth cue

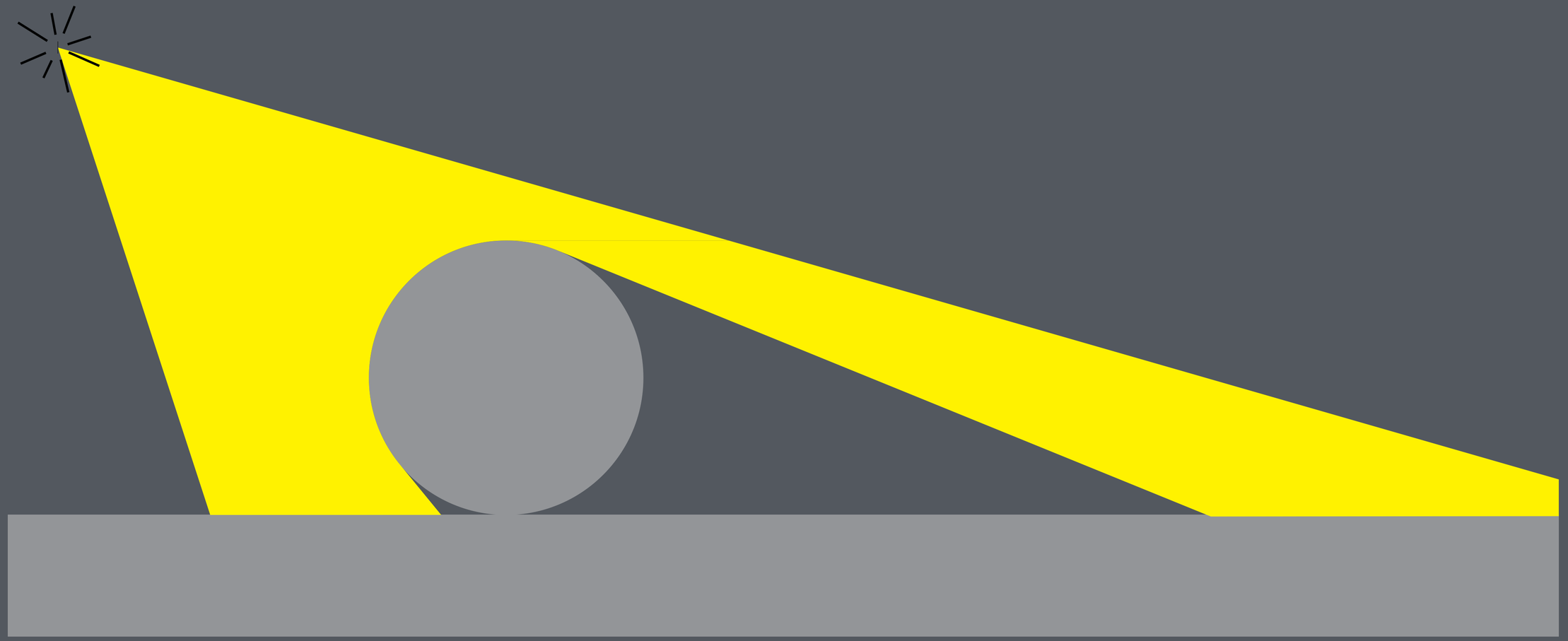


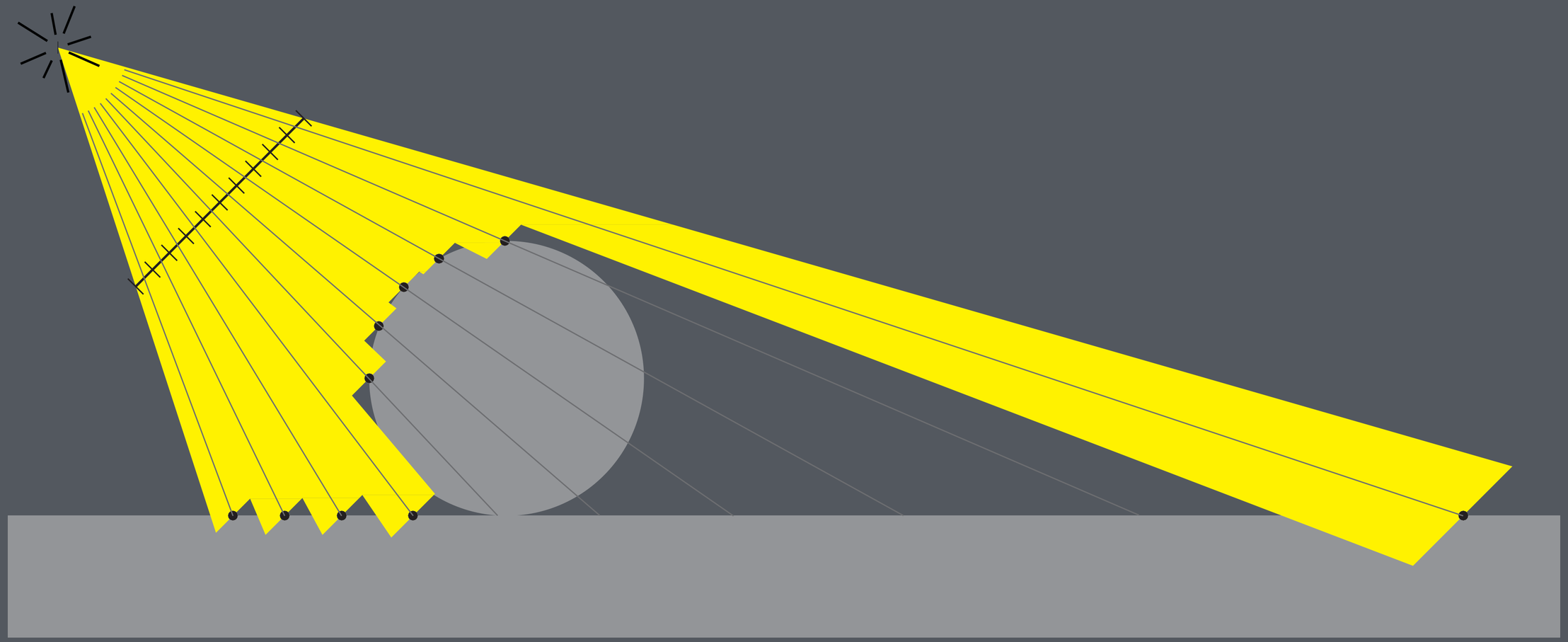


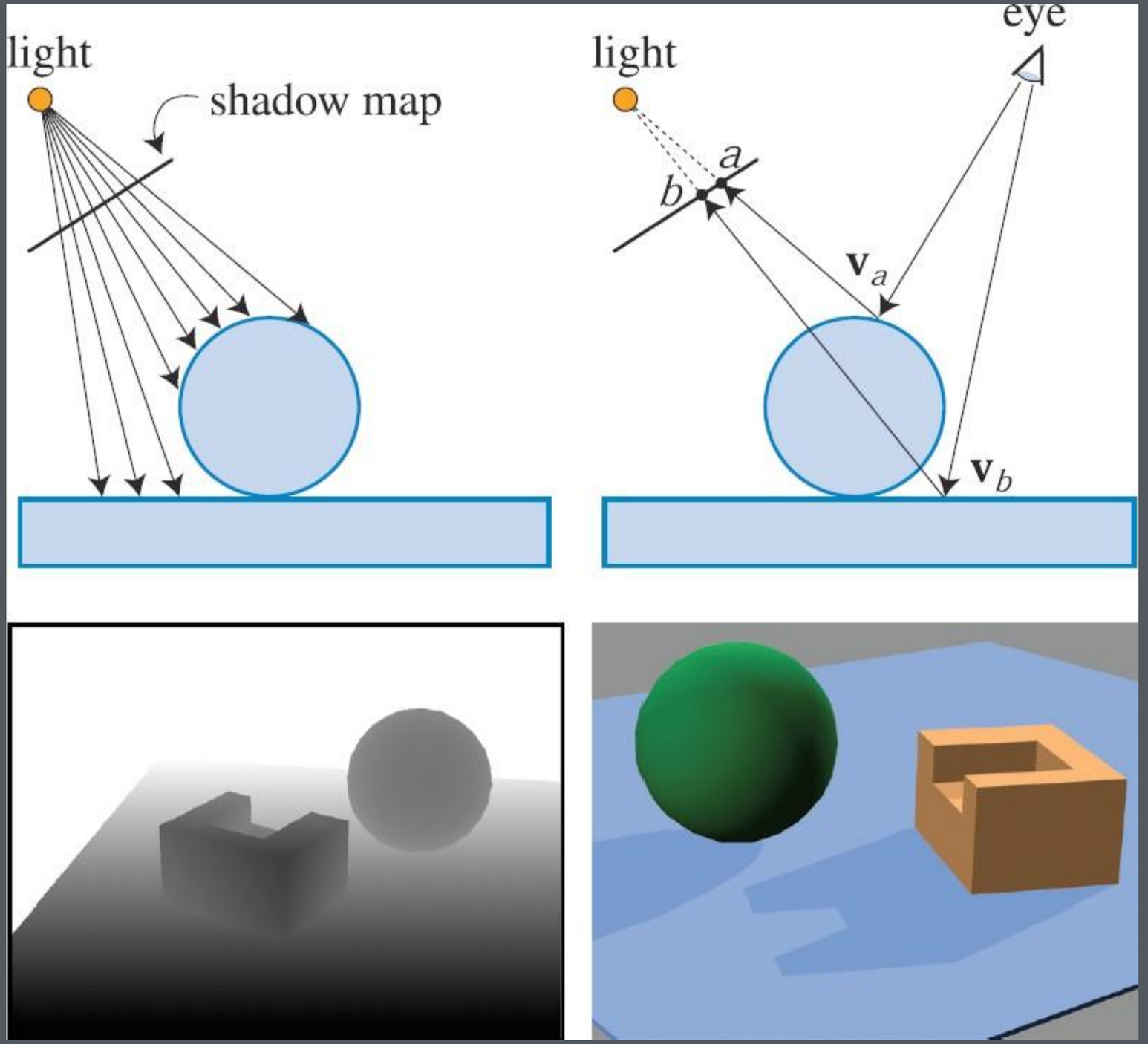
[tricks-and-illusions.com]

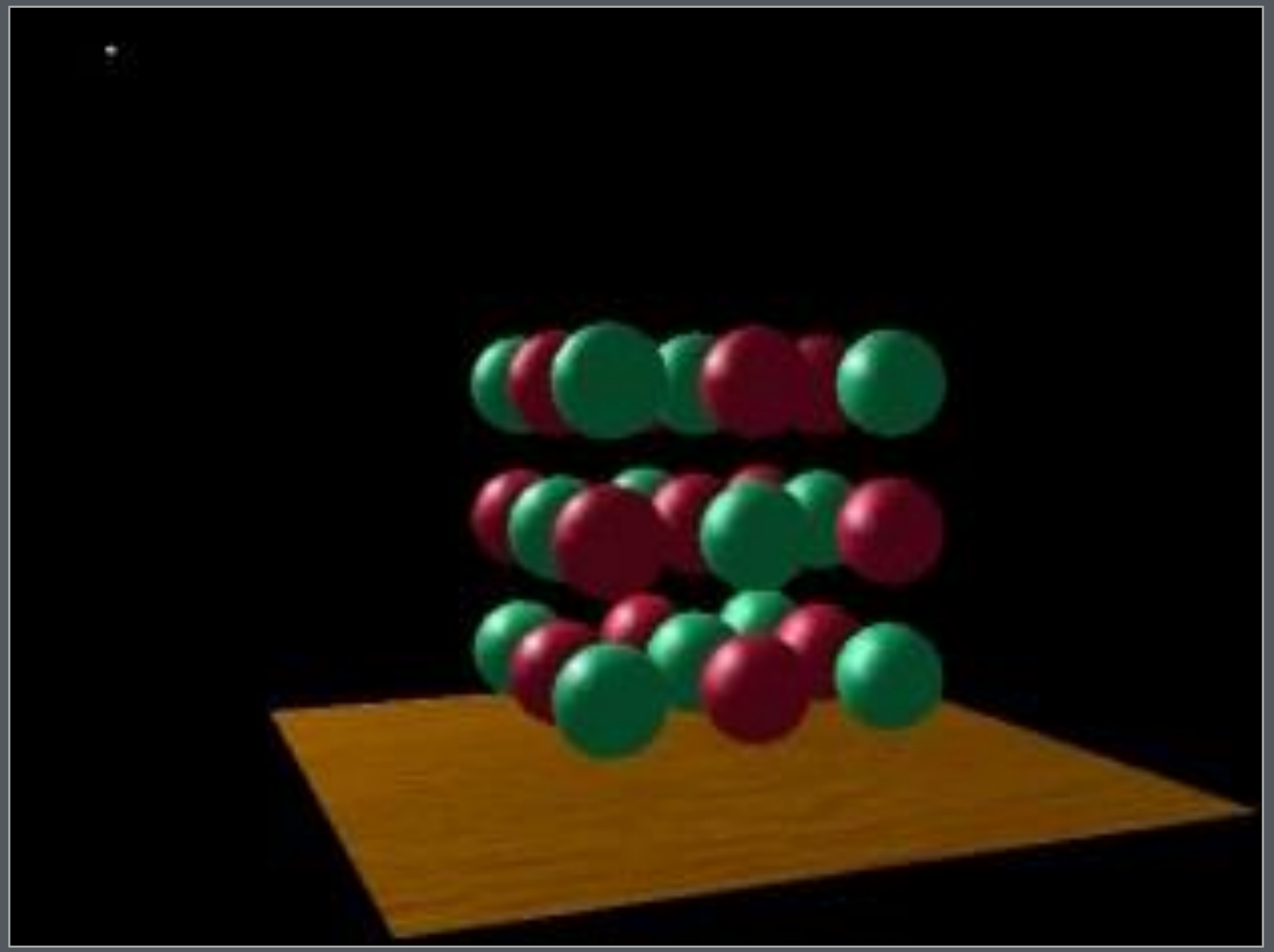
Shadows as anchors

Shadows as anchors

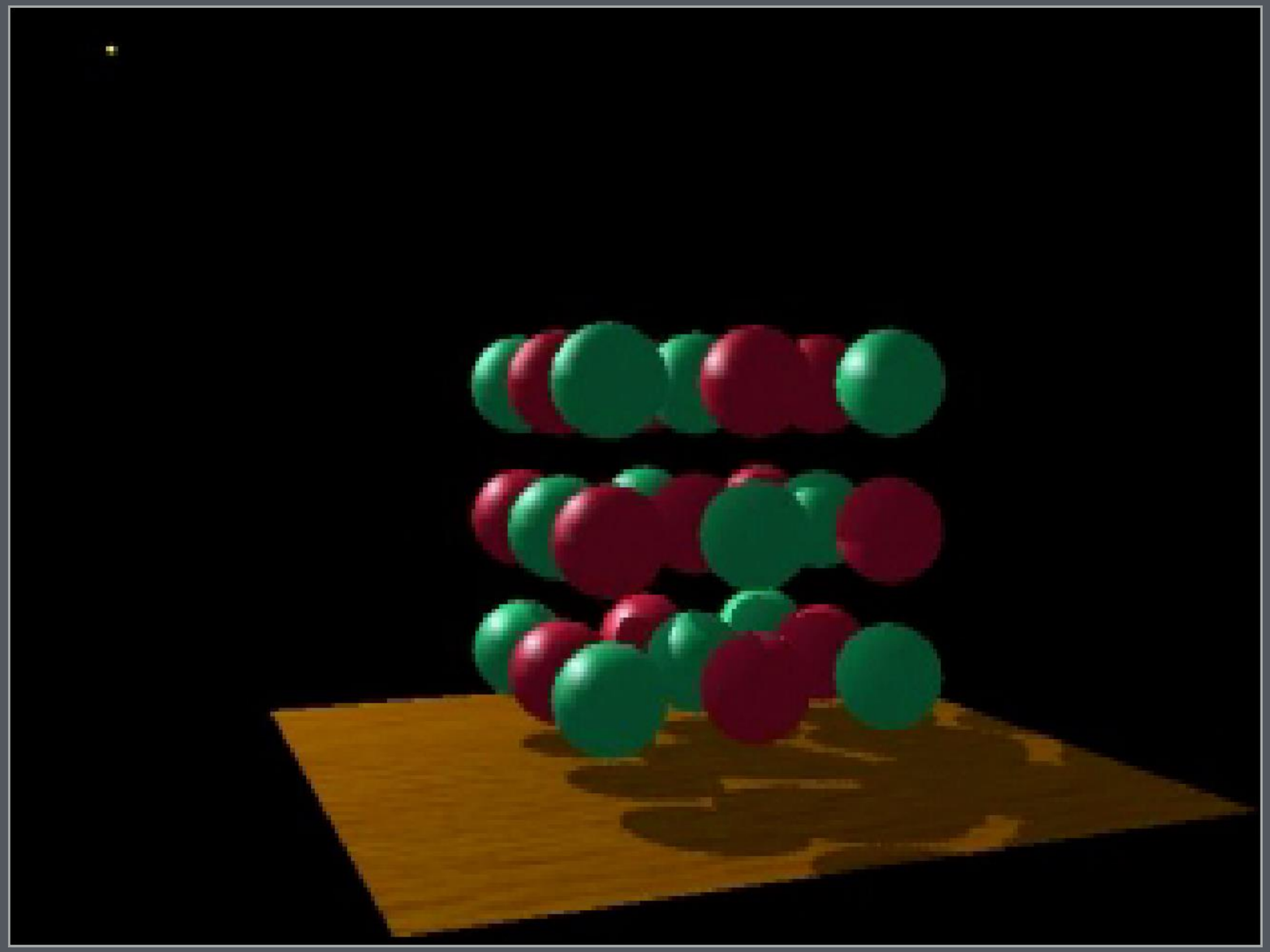




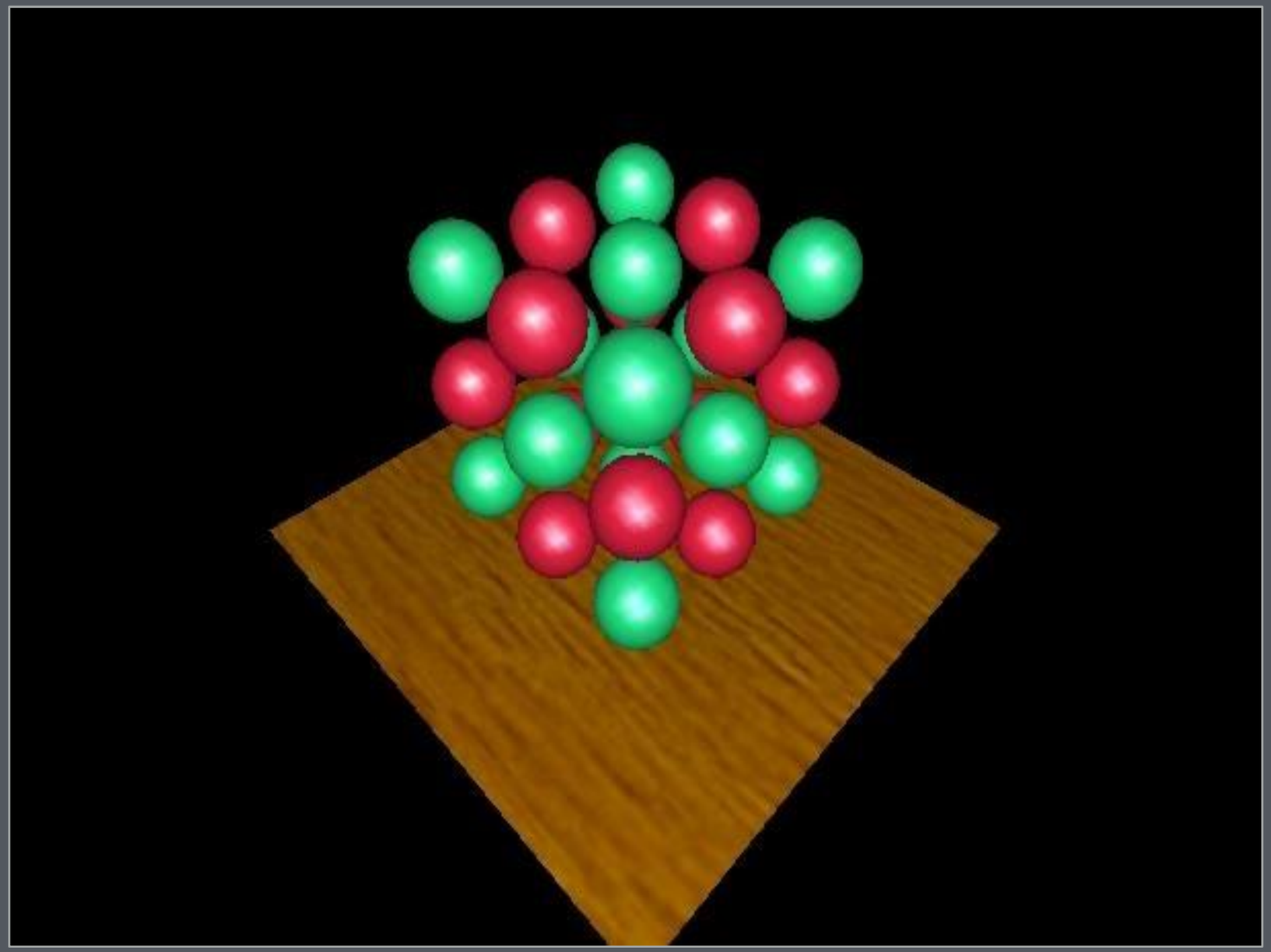

light
shadow map
light
eye
a
b
$\mathbf{v}_a$
$\mathbf{v}_b$



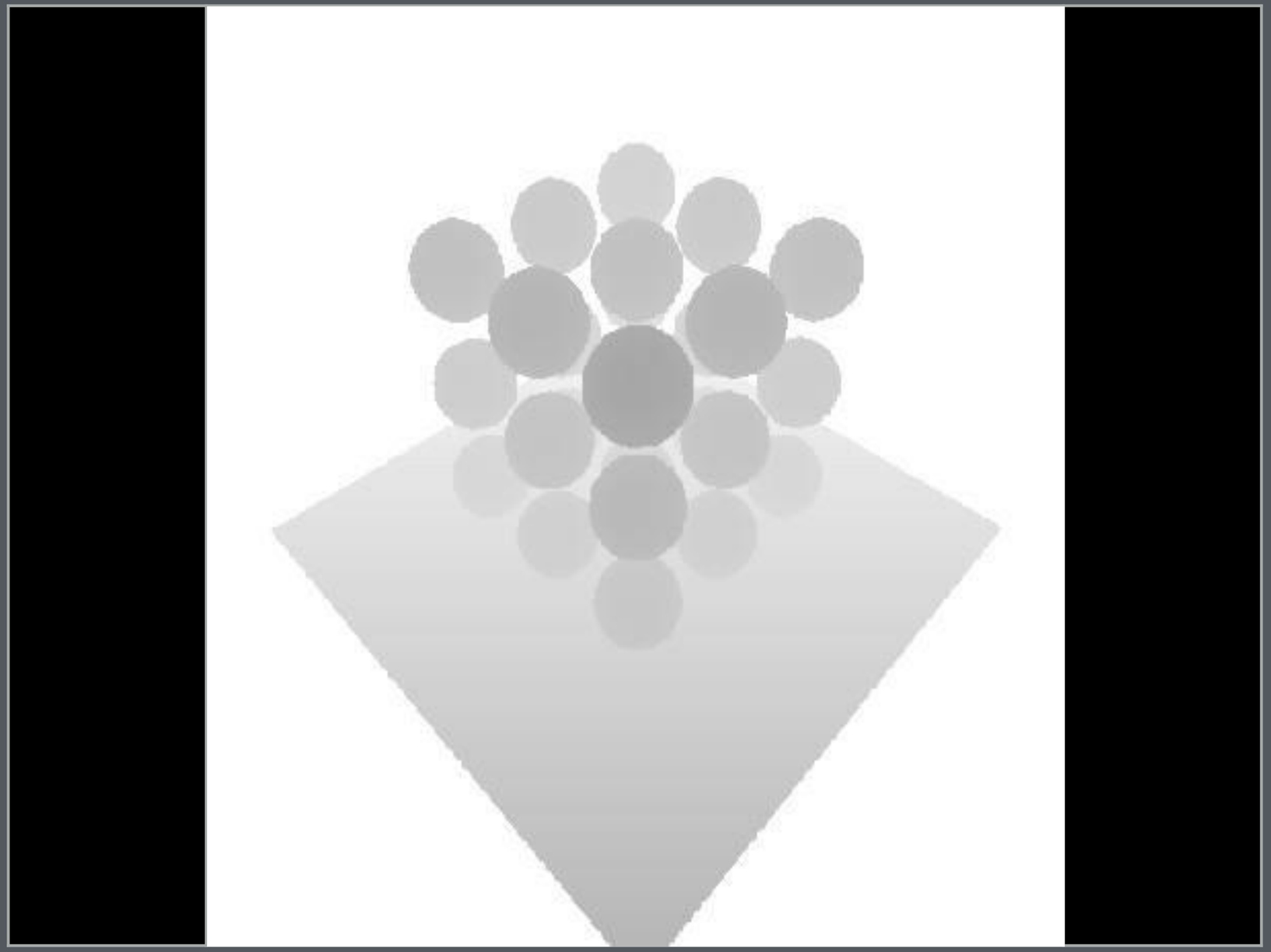
Mark Kilgard



Mark Kilgard



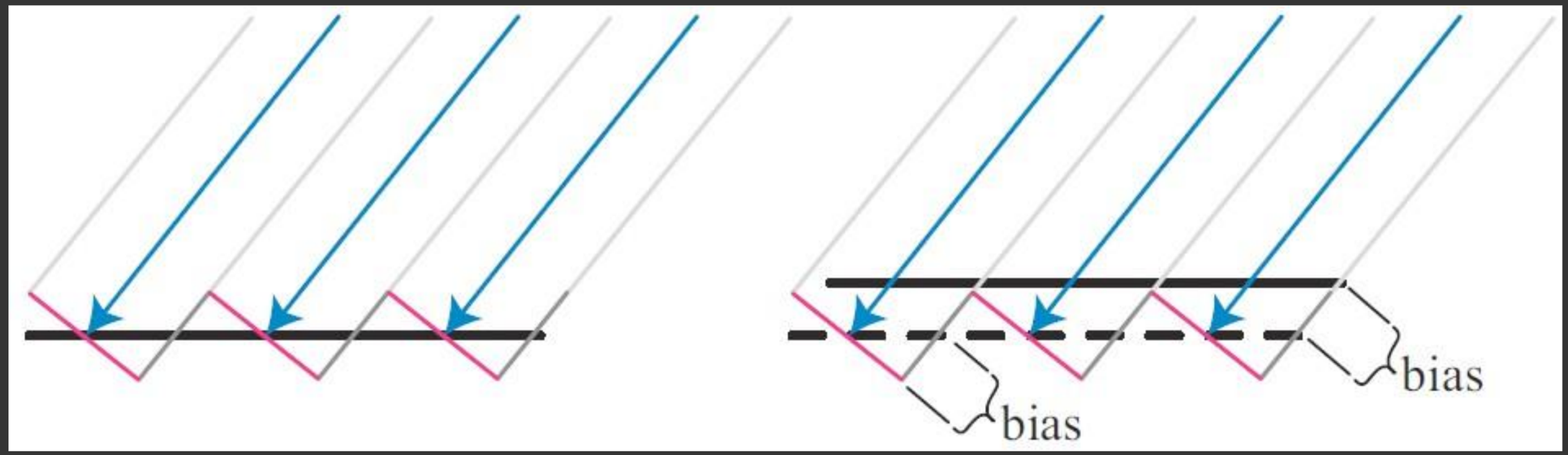
Mark Kilgard



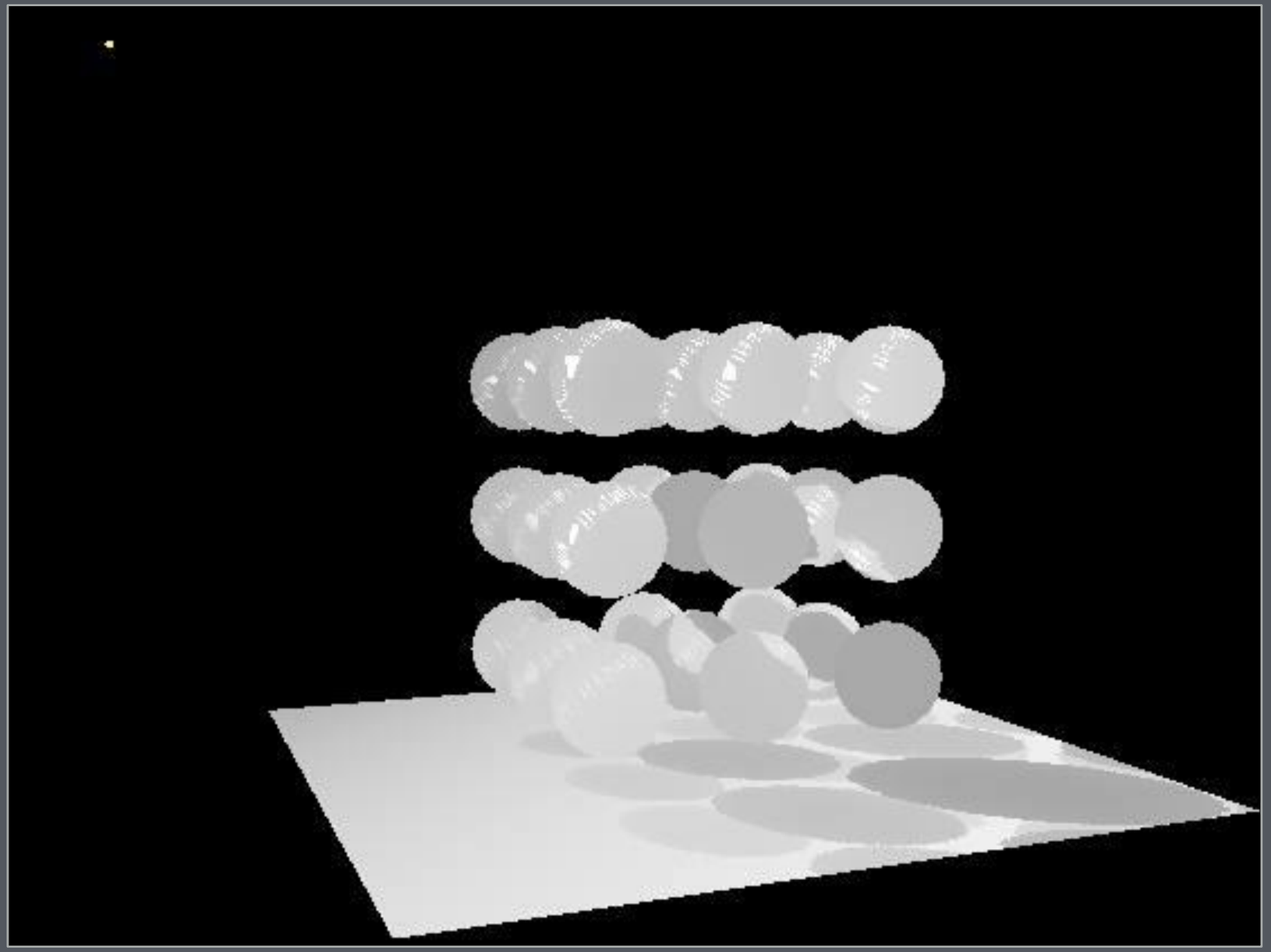
Mark Kilgard

Shadow Map Issues

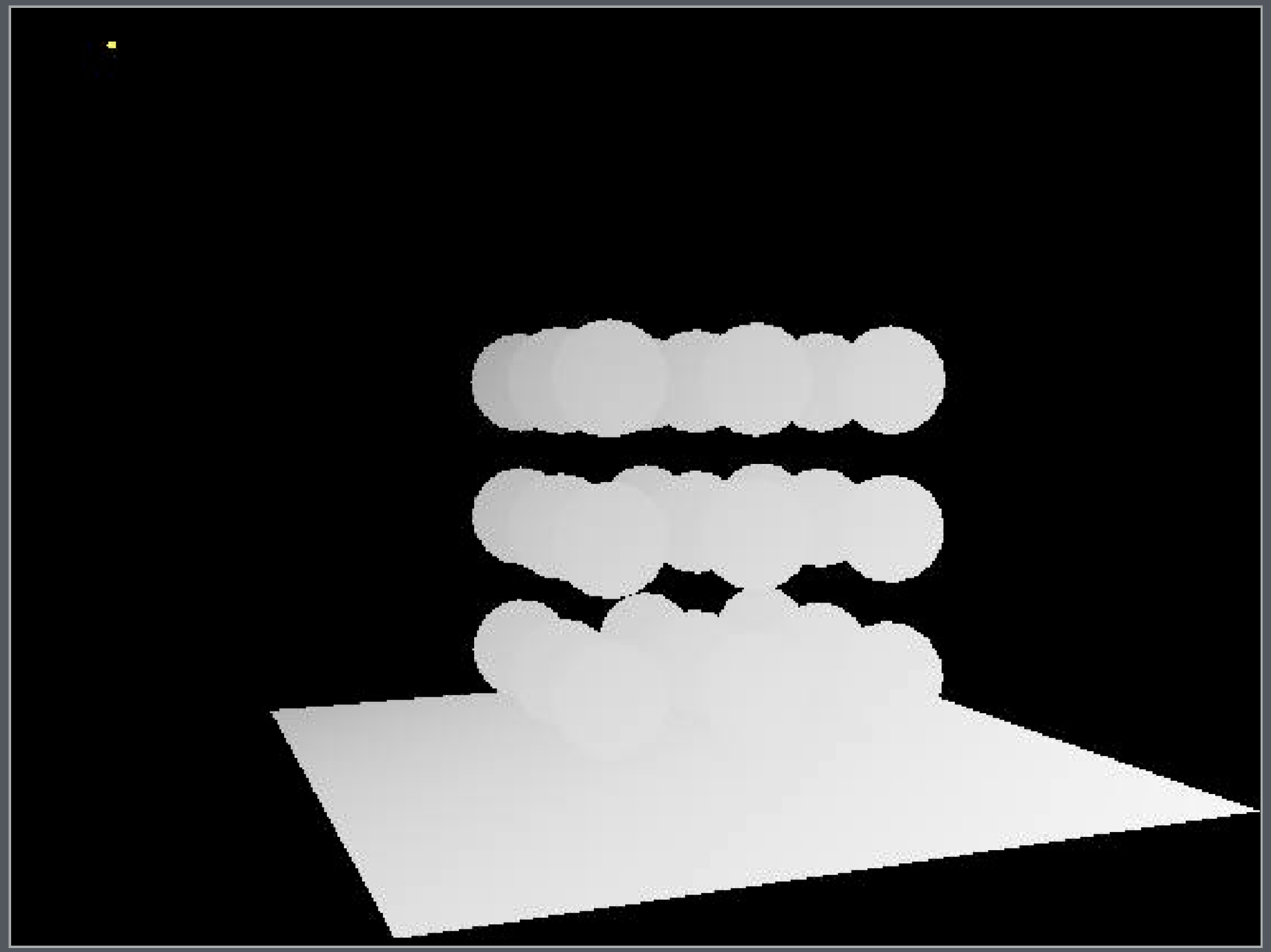
- if A and B are approximately equal?
- Speckling



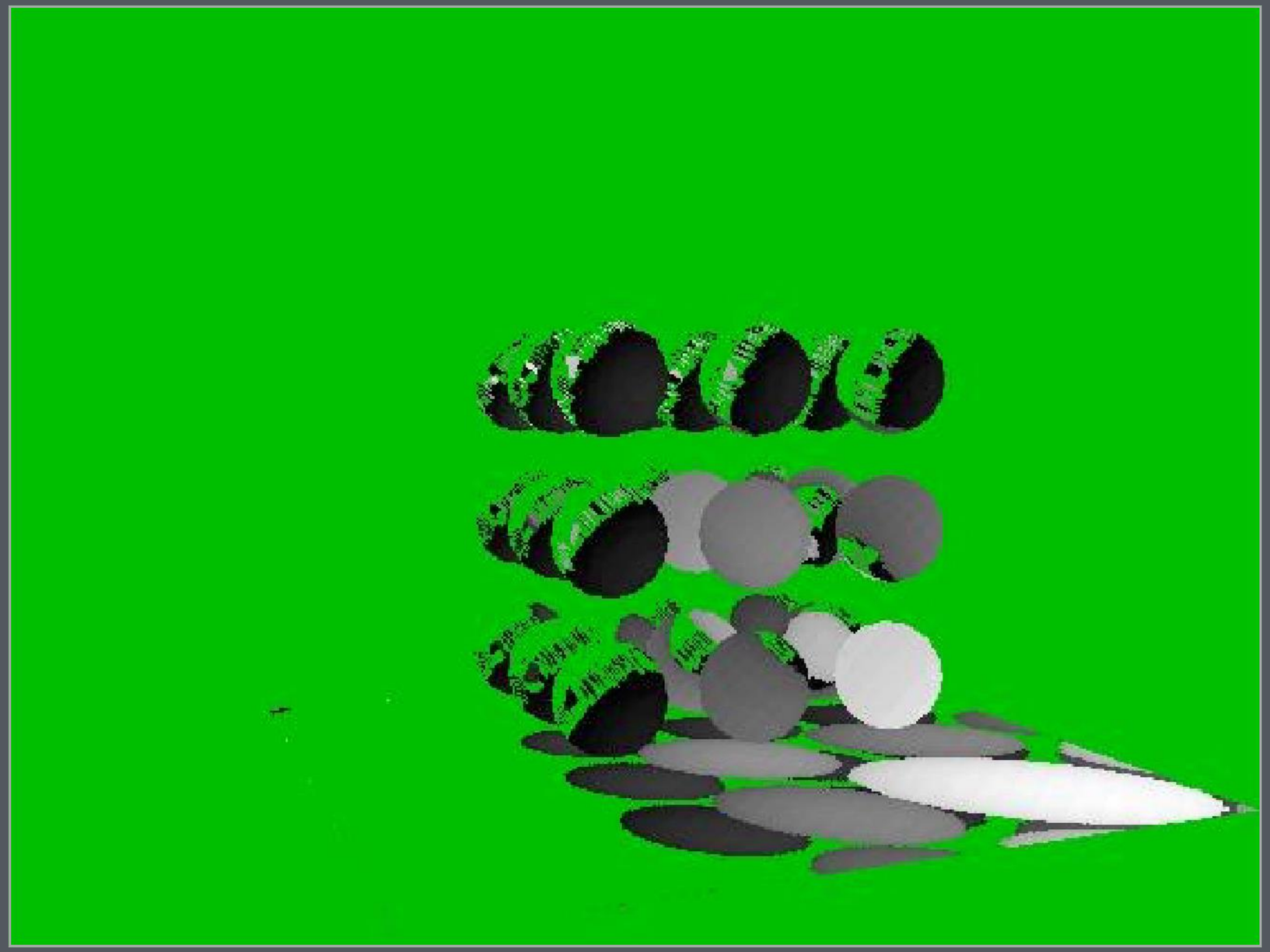
slide courtesy of Kavita Bala, Cornell University



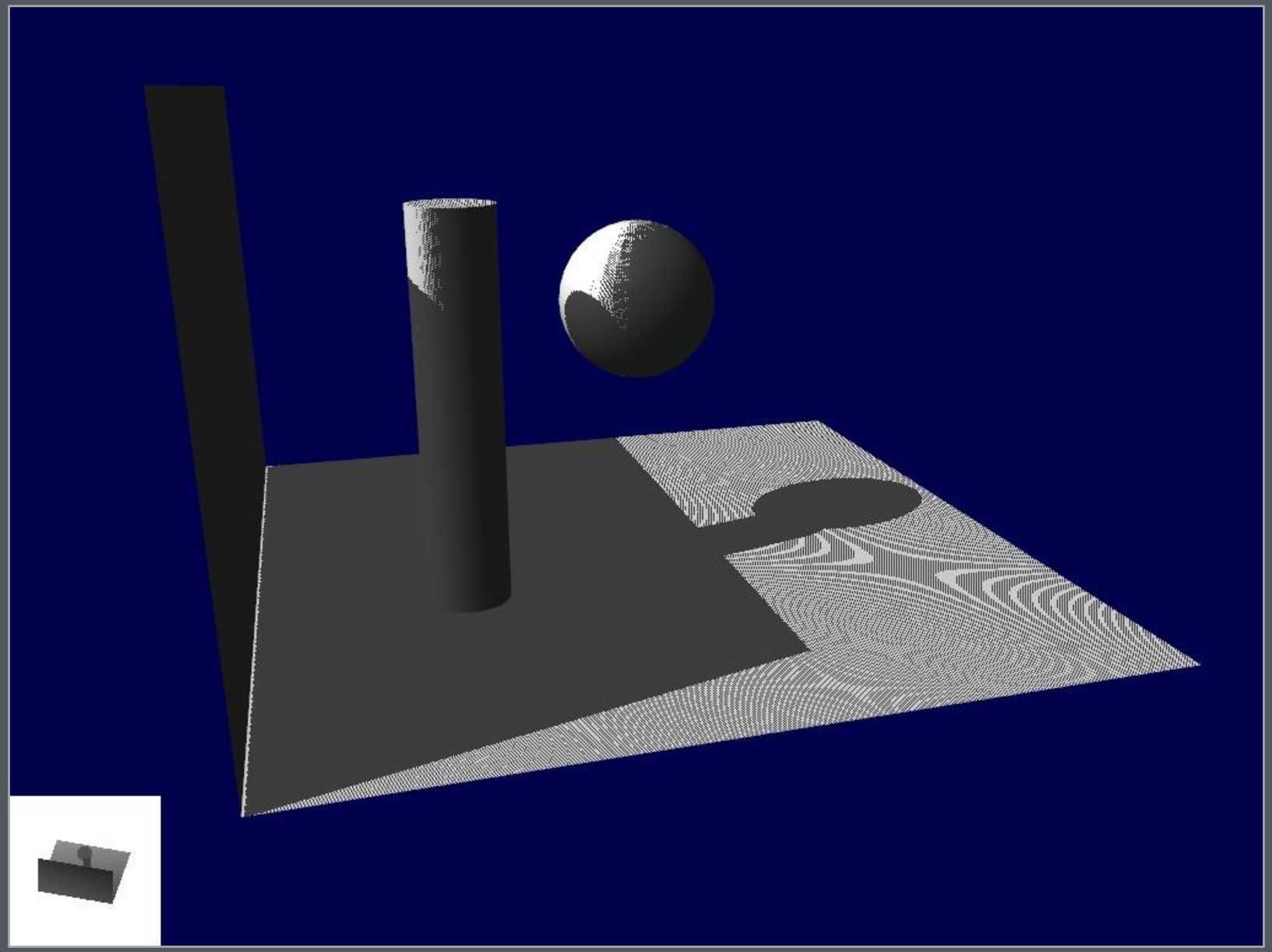
Mark Kilgard



Mark Kilgard

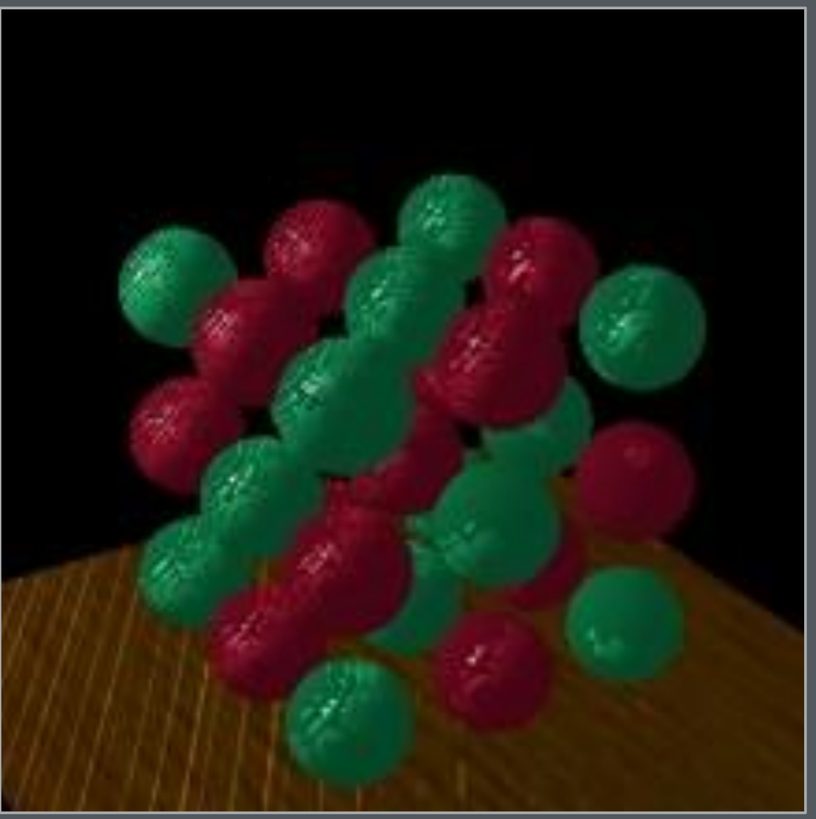


Mark Kilgard

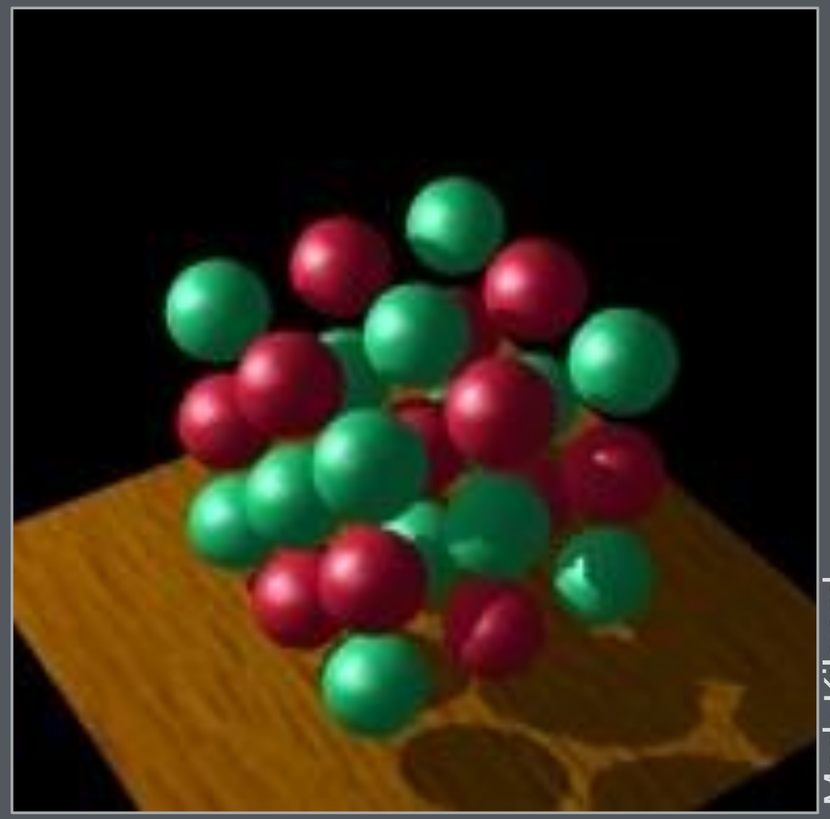


opengl-tutorial.org

first try at shadow mapping

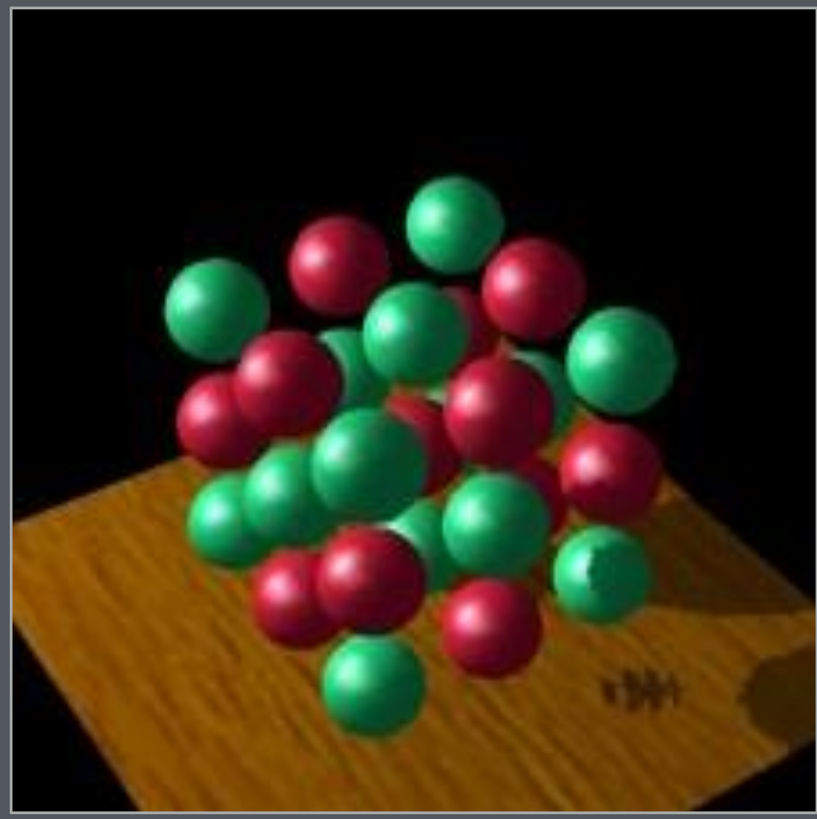


not enough shadow bias

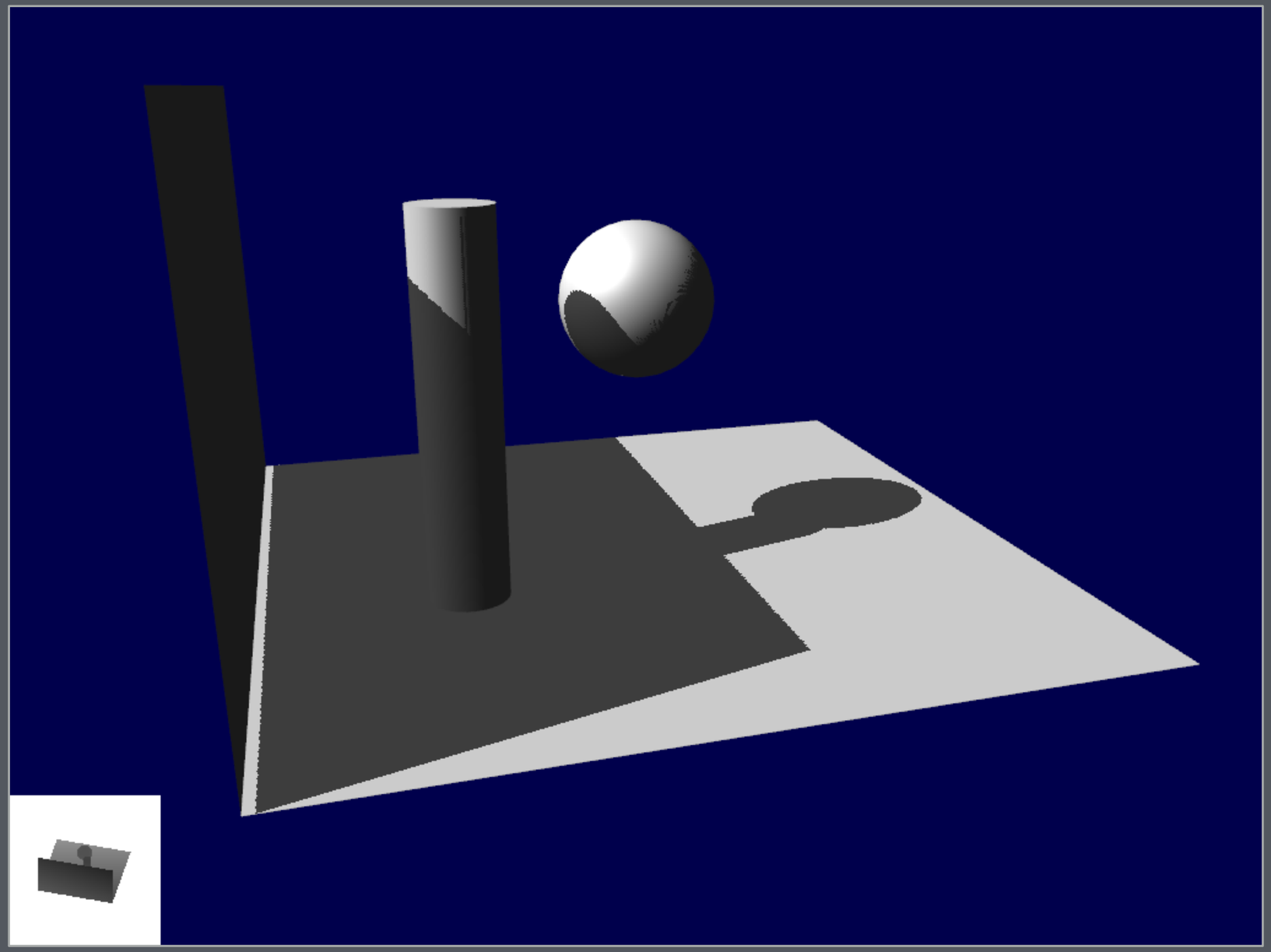


good shadow bias

Mark Kilgard

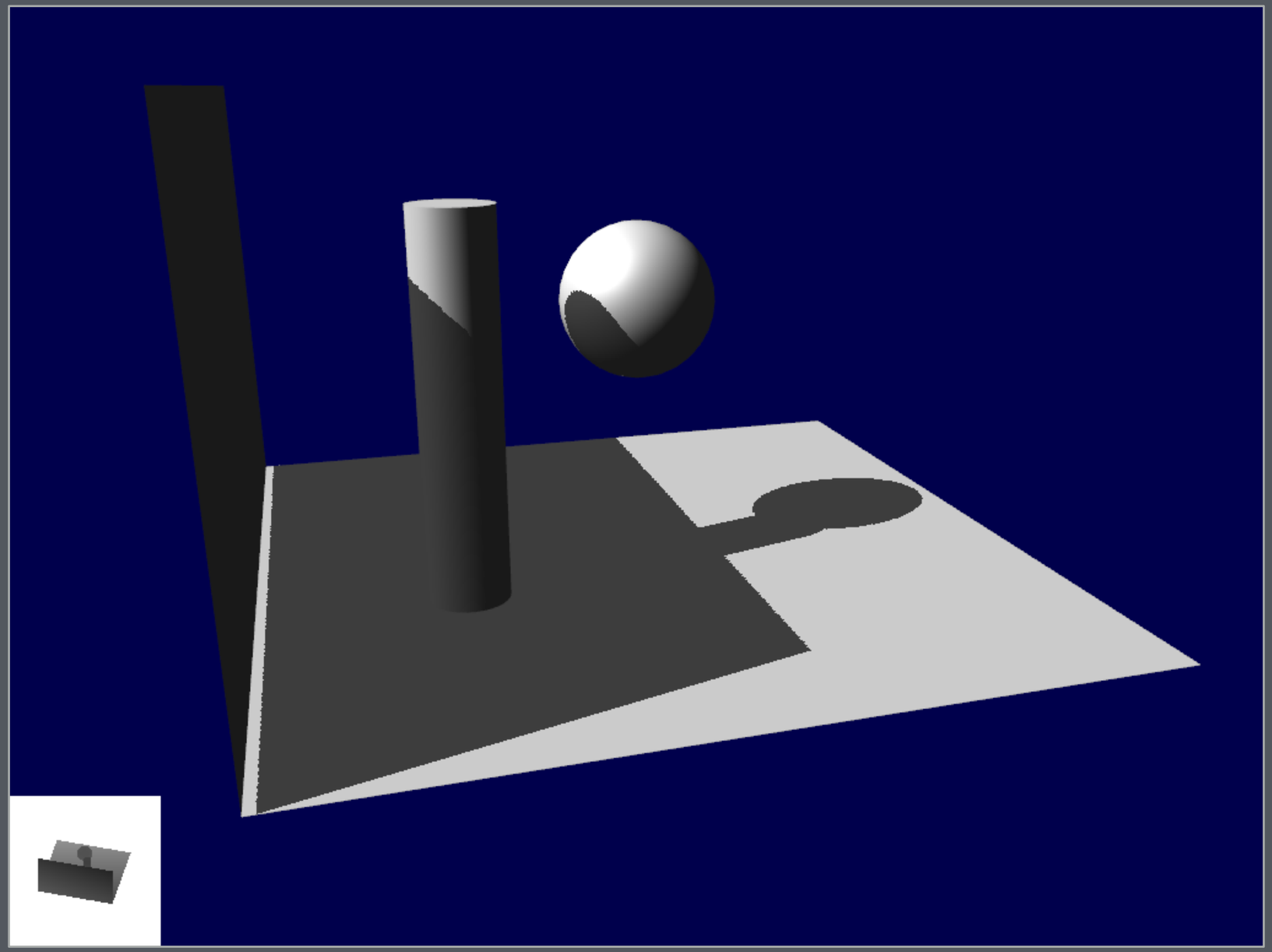


too much shadow bias



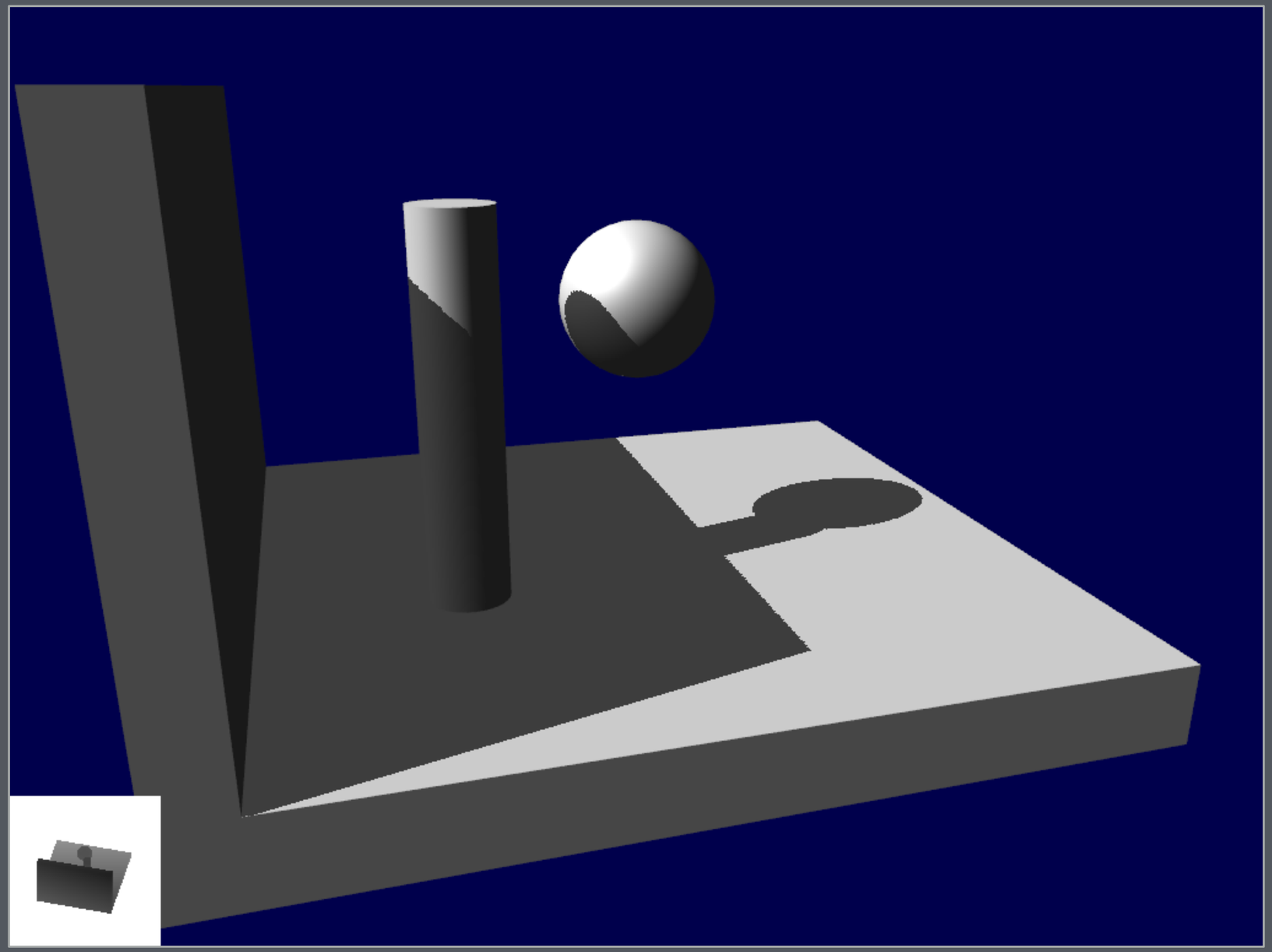
opengl-tutorial.org

shadow mapping with constant bias



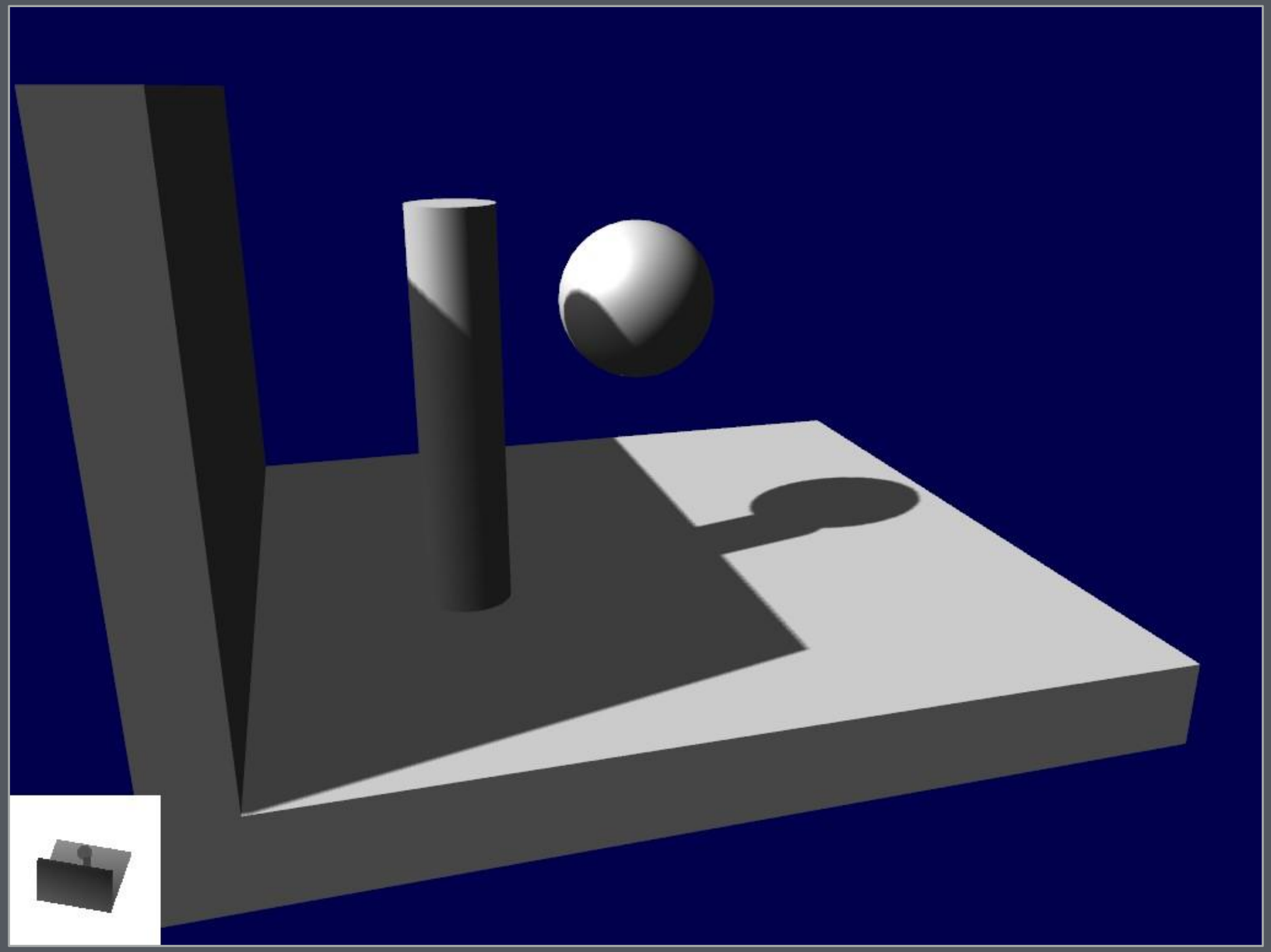
opengl-tutorial.org

shadow mapping with slope-dependent bias



opengl-tutorial.org

closed surfaces and slope-dependent bias



opengl-tutorial.org

adding percentage-closer filtering

Cascaded shadow maps (aka. parallel-split SM)

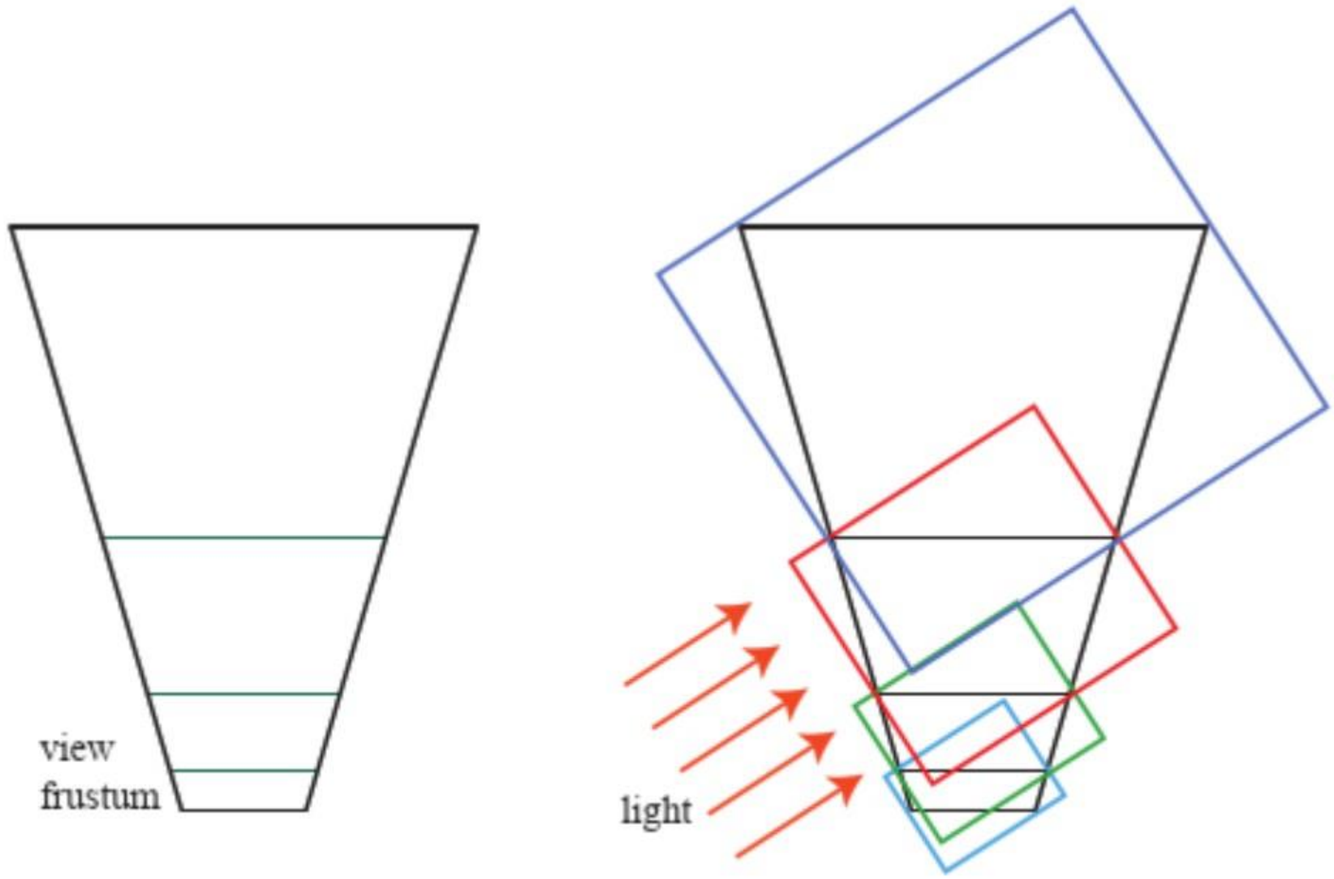


Figure 7.18. On the left, the view frustum from the eye is split into four volumes. On the right, bounding boxes are created for the volumes, which determine the volume rendered by each of the four shadow maps for the directional light. *(After Engel [430].)*

[Möller et al. *RTR*]

Single shadow map, 2048x2048

Four 1024x1024 shadow maps (equal memory)

Fan Zhang, Chinese U. Hong Kong

Filtering shadow maps

Shadow map lookups cause aliasing, need filtering

As with normal maps, pixel is a nonlinear function of the shadow depth

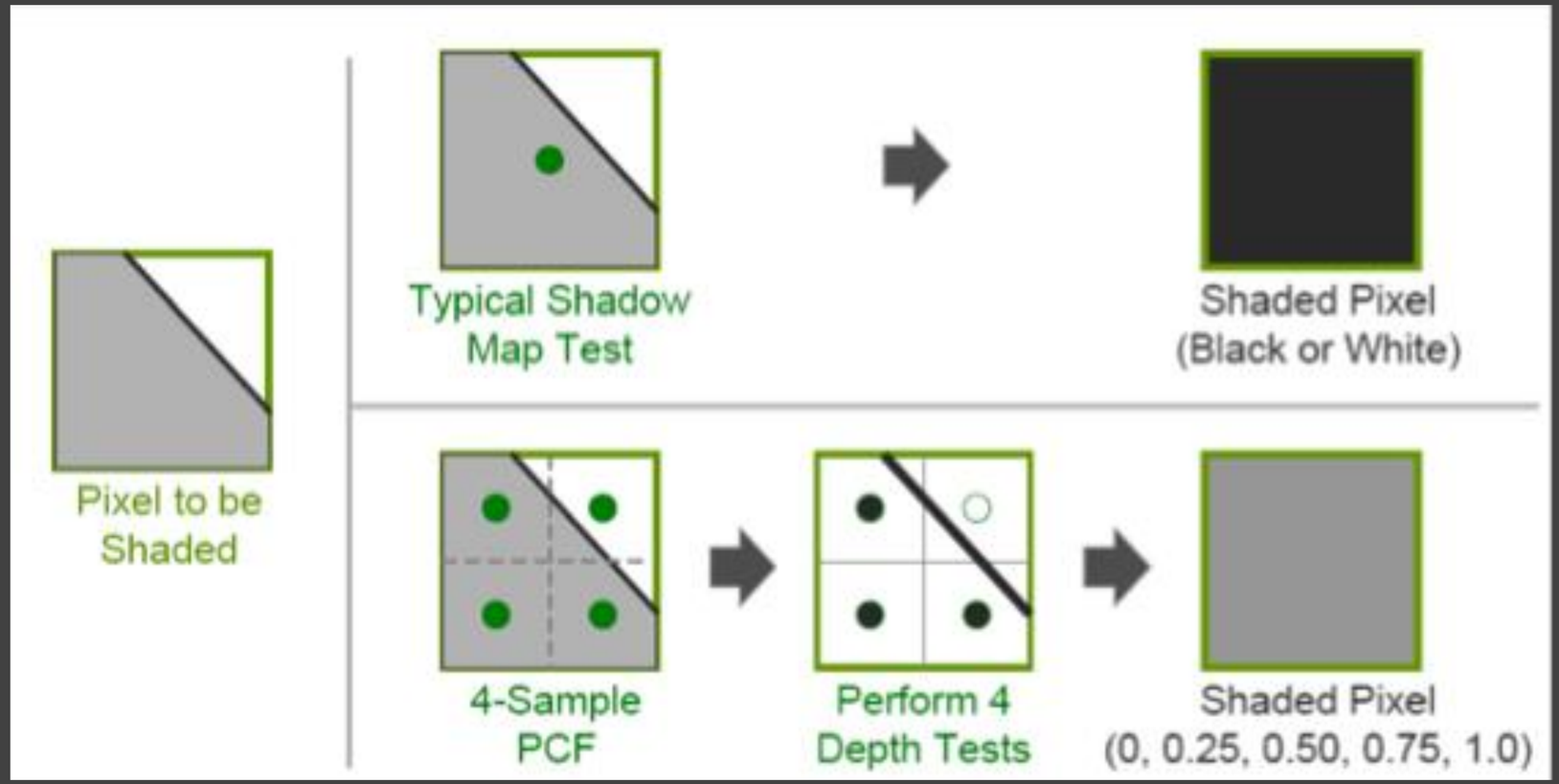
- this means applying a linear filter to the depth is wrong

We want to filter the output, not the input, of the shadow test

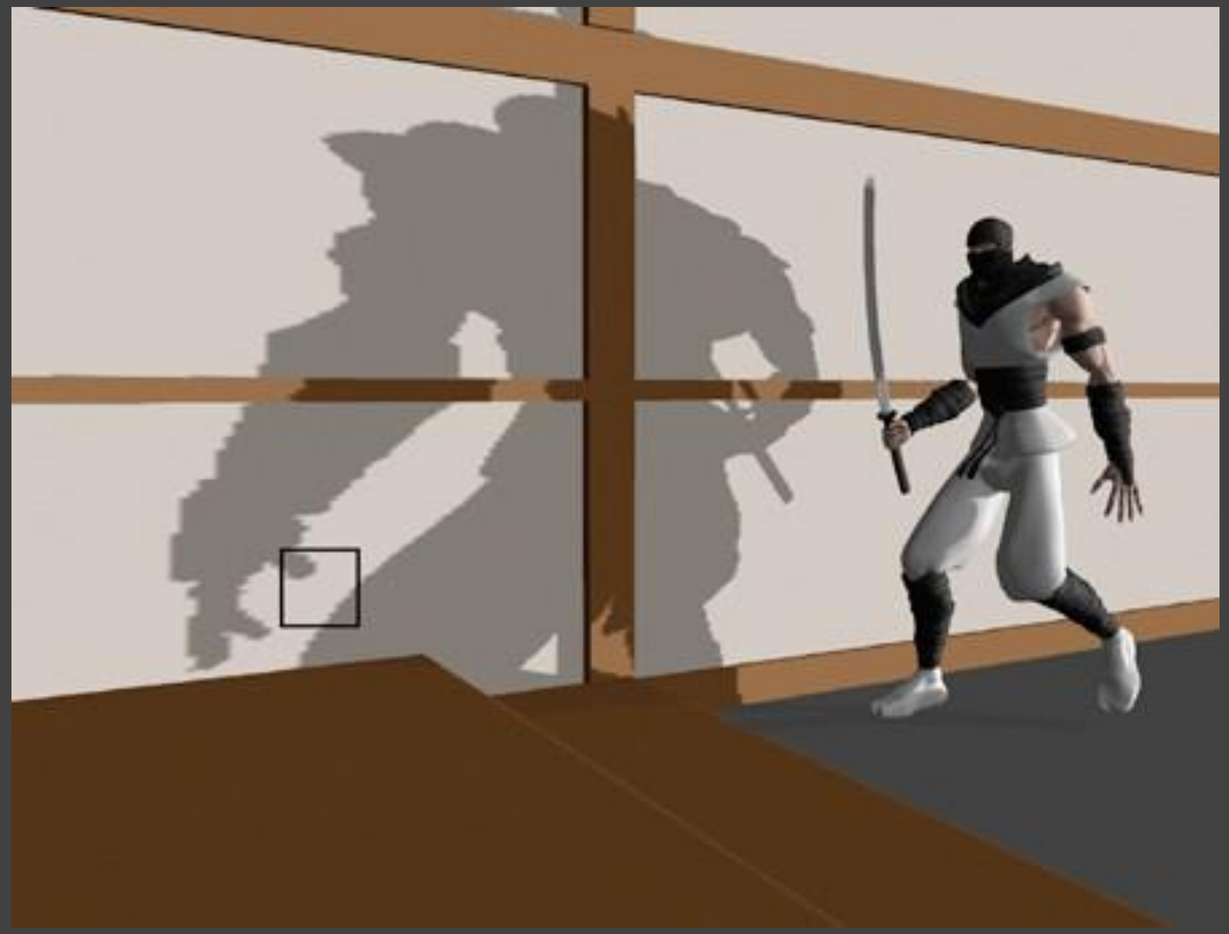
- what fraction of samples pass the test
- samples pass the test if they are closer than the shadow map depth
- therefore “percentage closer filtering” or PCF

Percentage Closer Filtering

- Soften the shadow to decrease aliasing
  - Reeves, Salesin, Cook 87
  - GPU Gems, Chapter 11

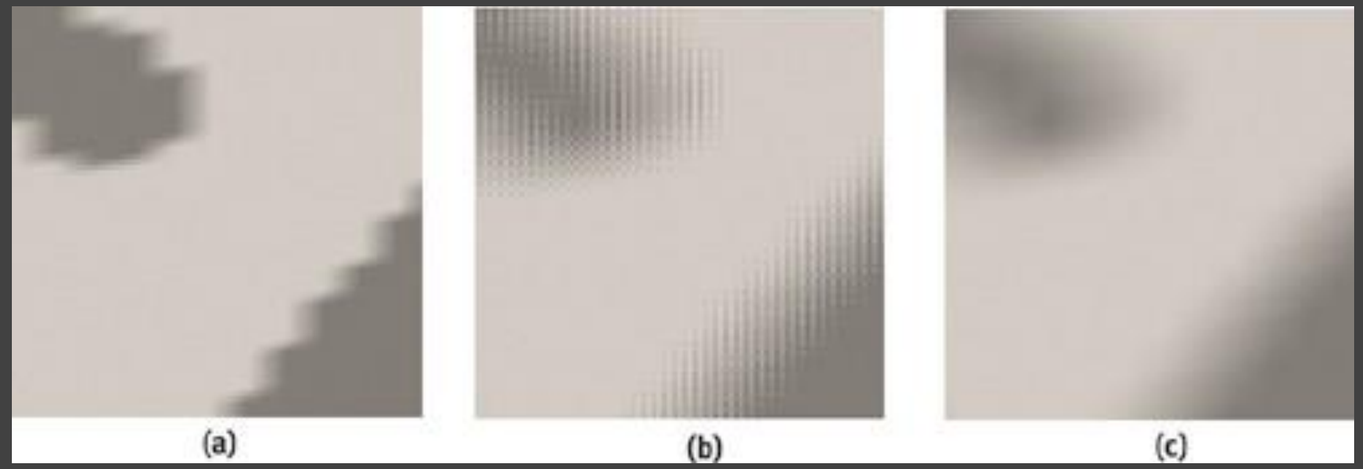


1 sample SM



4 sample PCF

16 sample PCF


(a)
(b)
(c)

Soft shadows from small sources

Main effect is to blur shadow boundaries

- PCF can do this
- …but how wide to make the filter?

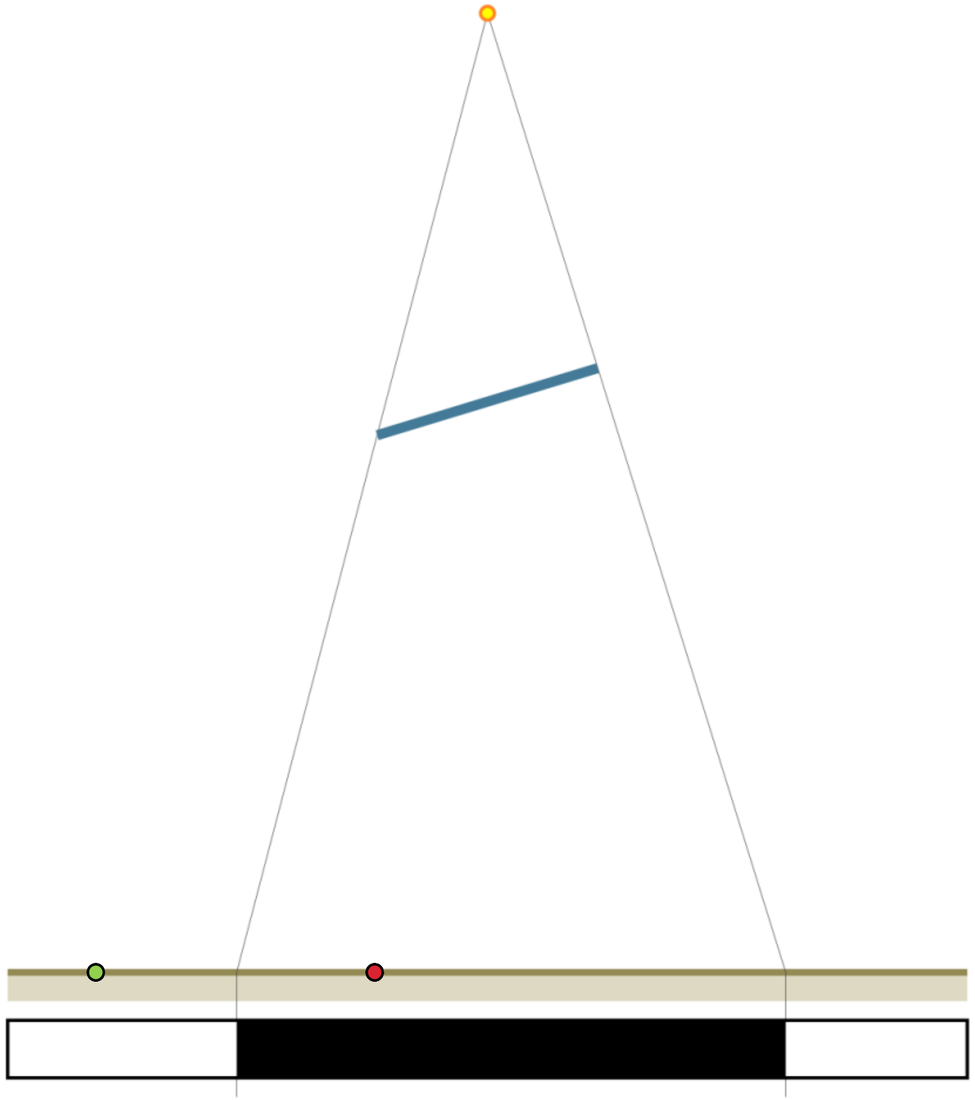
Real shadows depend on area of light visible from surface

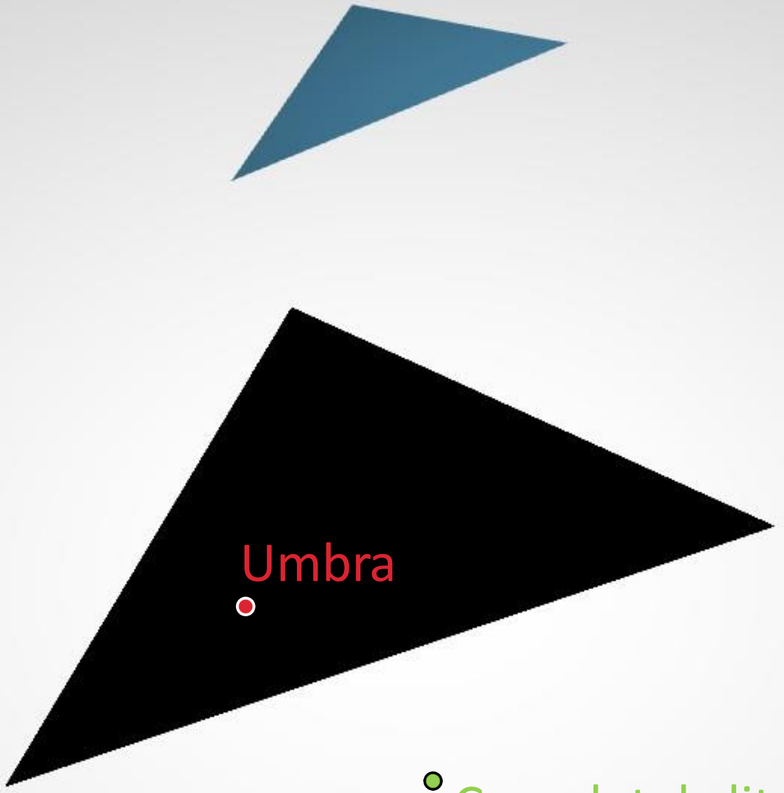
- this can vary in complex ways
- example: sun viewed through leafy trees

Useful approximation: convolution

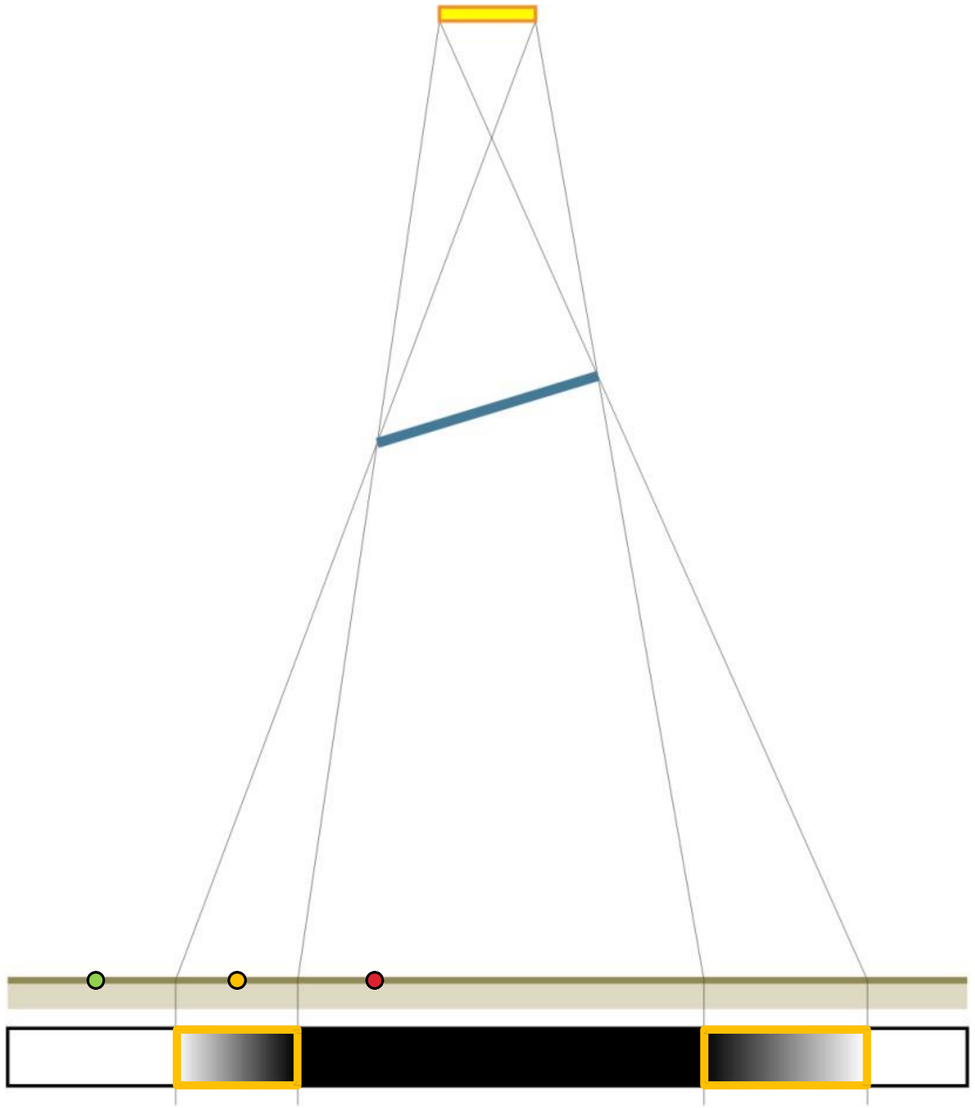
- shadows are convolutions when the blocker and source are parallel and planar
- occluder fusion: approximating some occluding geometry as a planar blocker

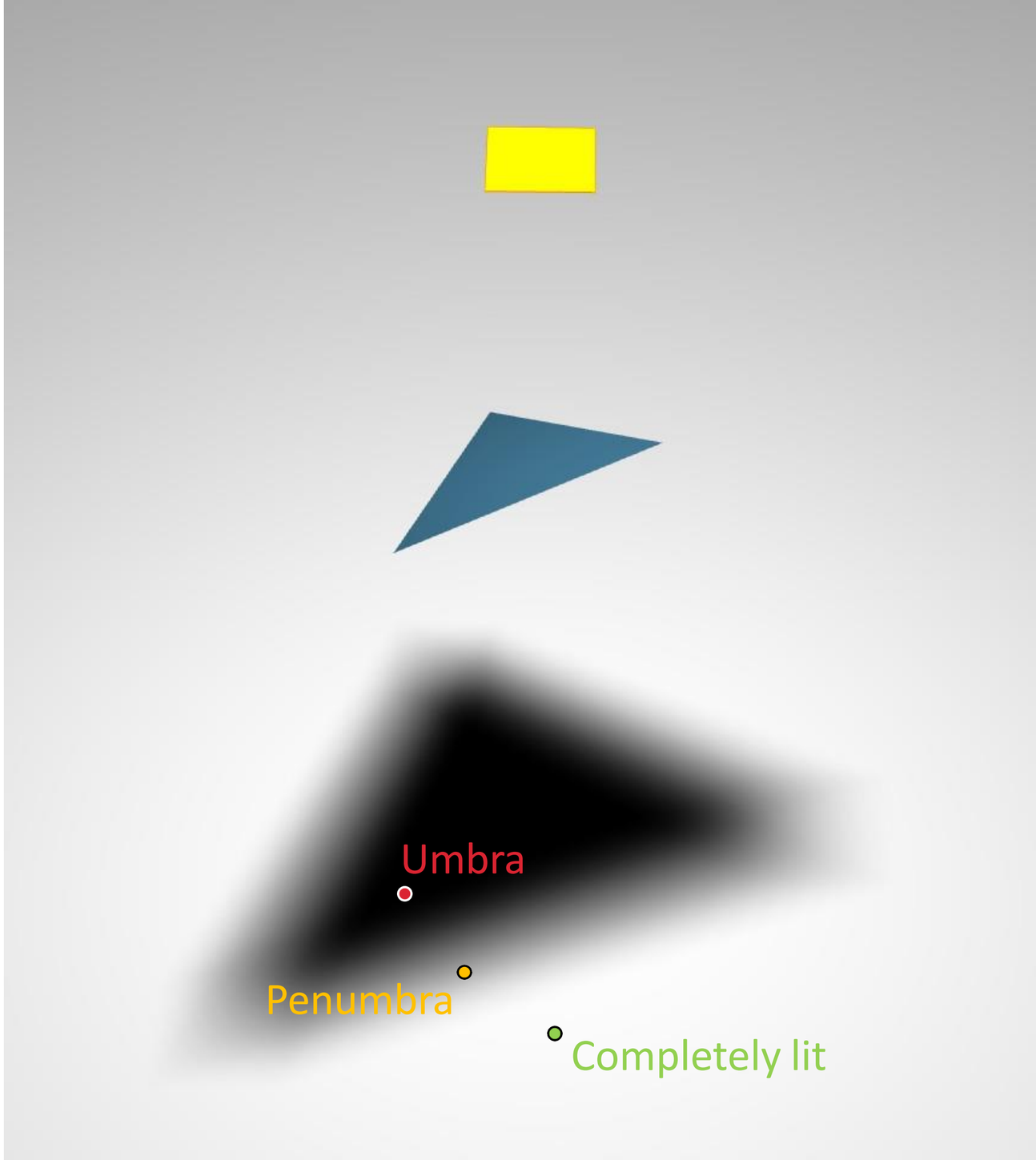
Hard Shadows





Soft Shadows

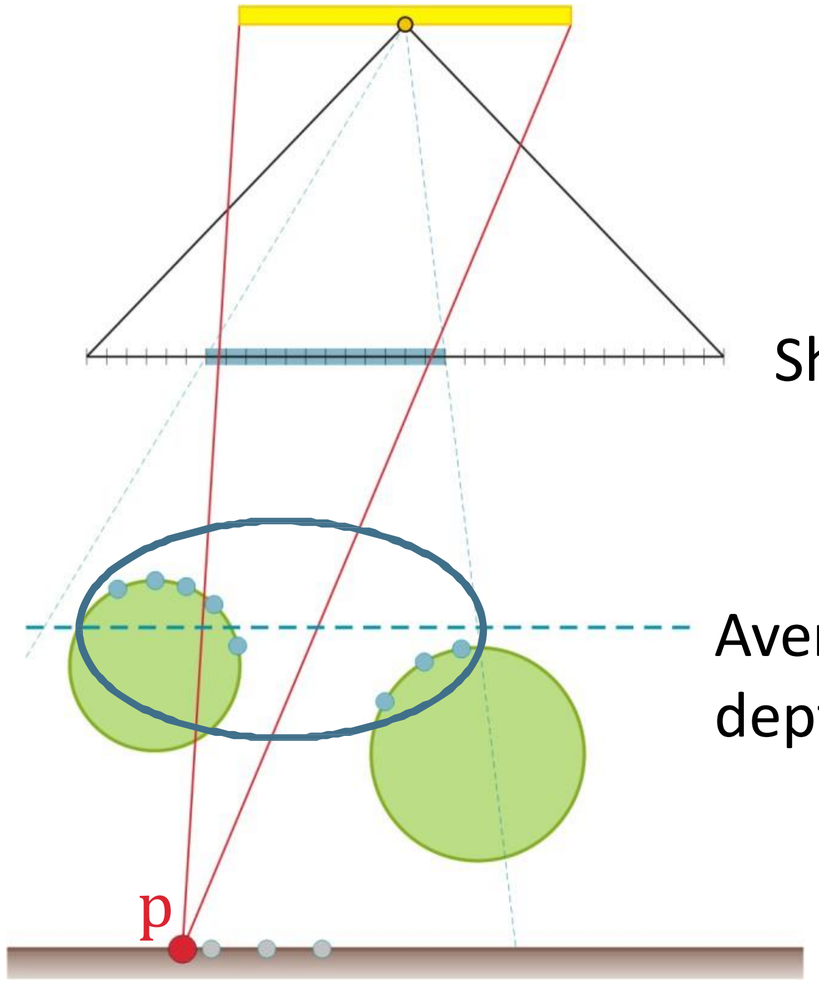




Shadow Hardening on Contact

Percentage-Closer Soft Shadows

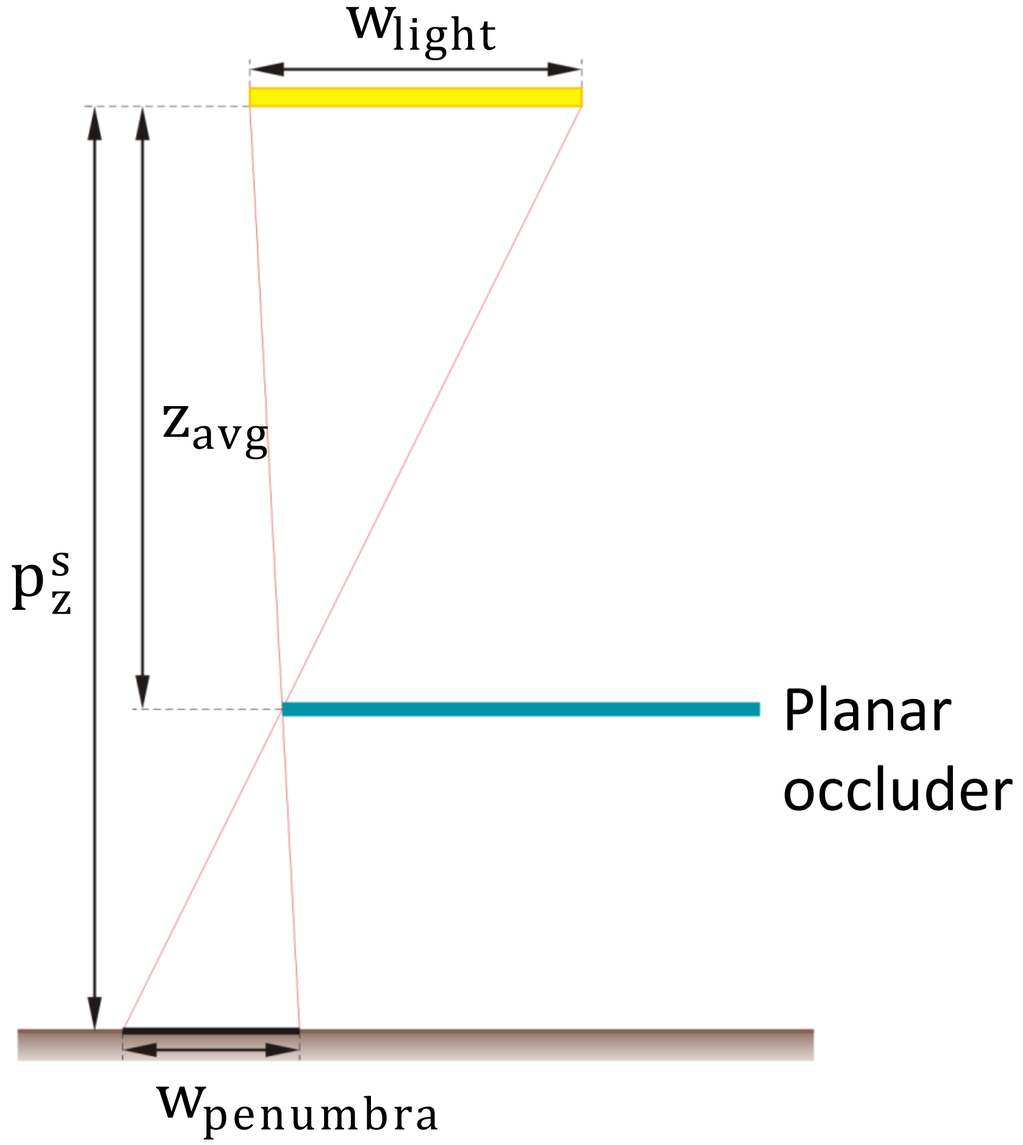
1. Blocker search



Percentage-Closer Soft Shadows

1. Blocker search
2. Penumbra width estimation

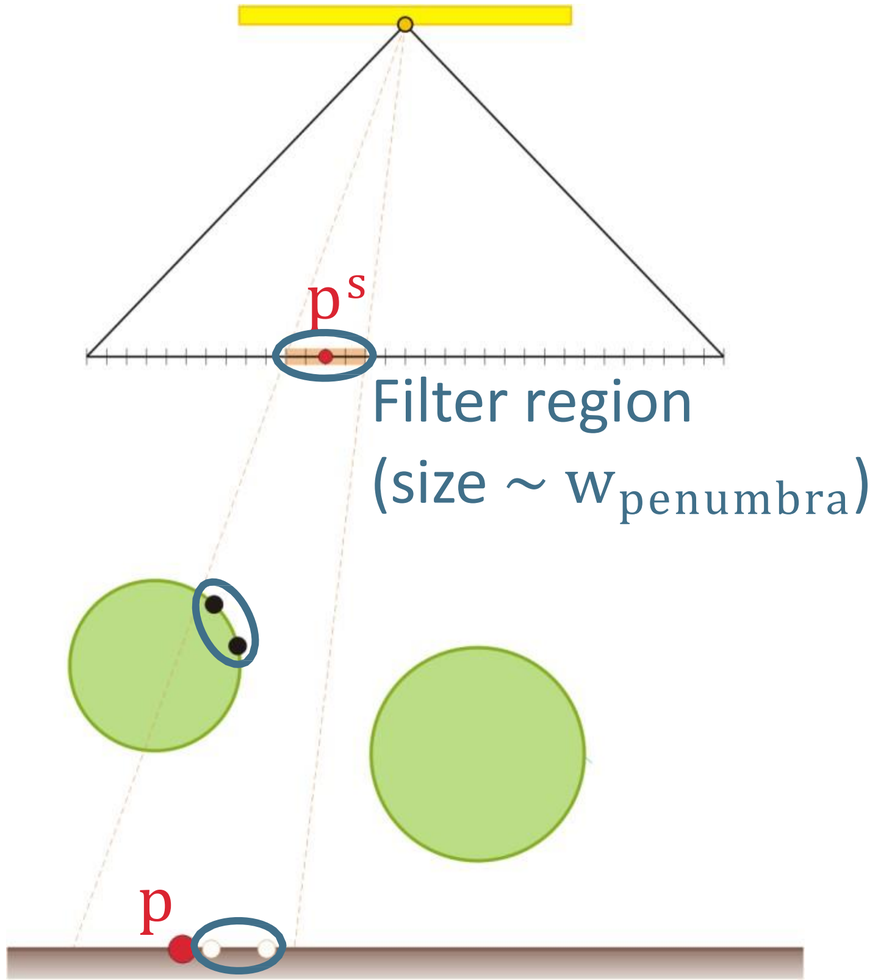
$$w_{penumbra} = \frac{p_z^s - z_{avg}}{z_{avg}} w_{light}$$



Percentage-Closer Soft Shadows

1. Blocker search
2. Penumbra width estimation
3. Filtering

50%



Percentage-closer soft shadows

Fernando, NVidia whitepaper ~2005

Detail map

Hierarchy of scales

macroscopic

mesoscopic

microscopic

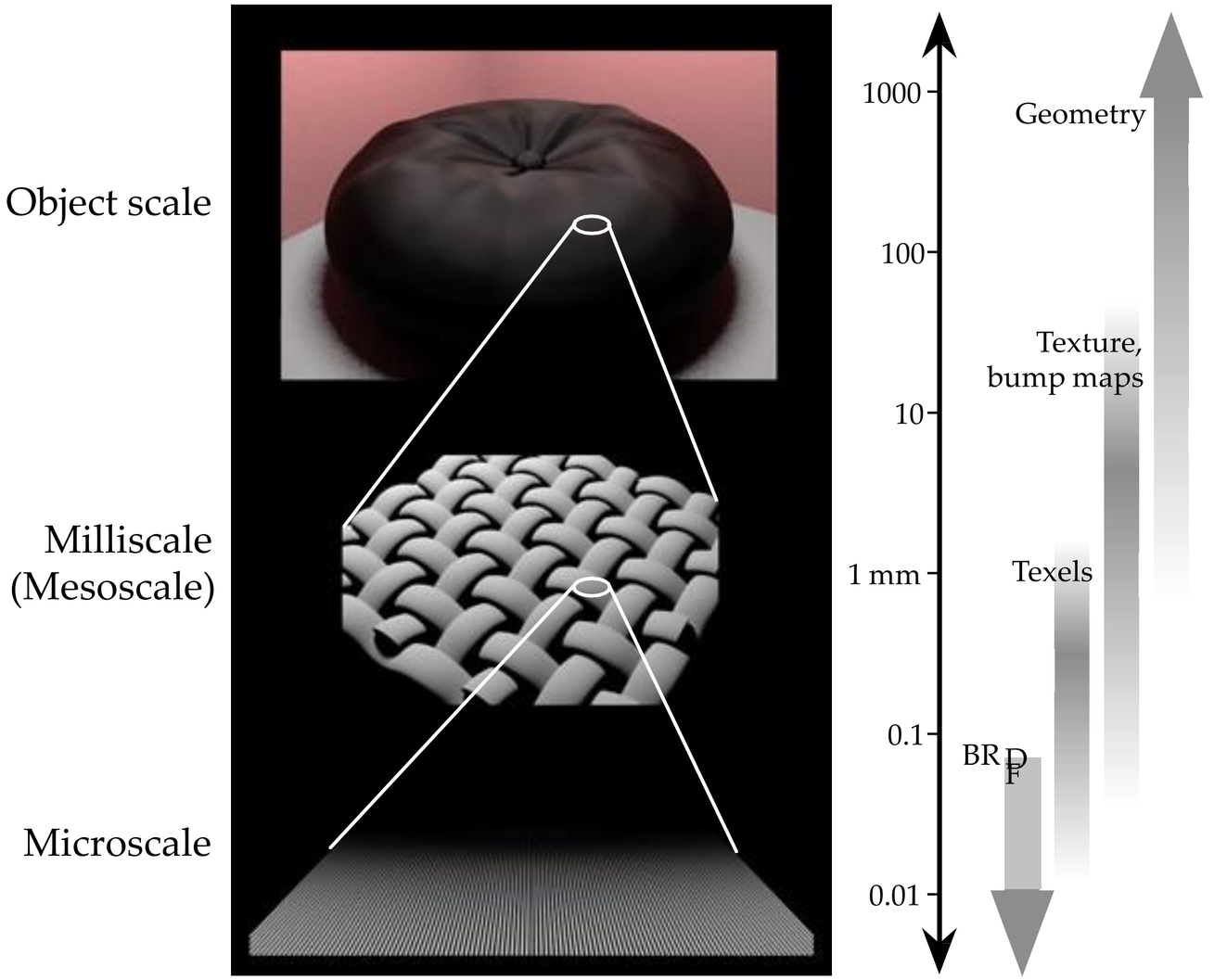
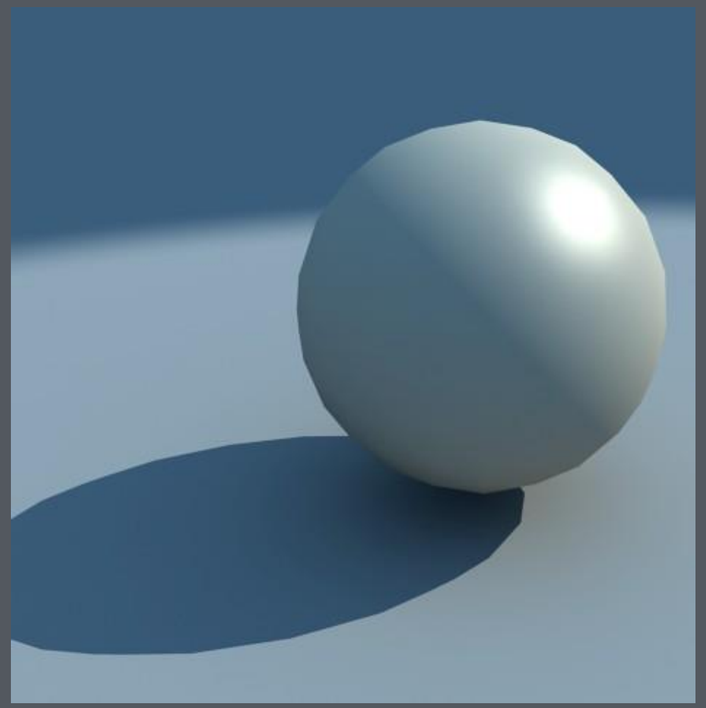


Figure 1: Applicability of Techniques

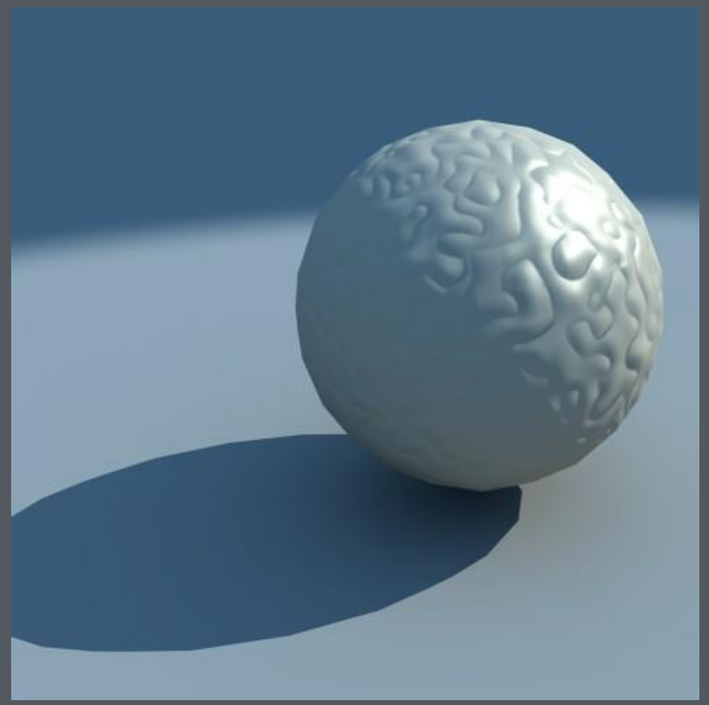
Displacement and Bump/Normal Mapping

Mimic effect of geometric detail/meso geometry

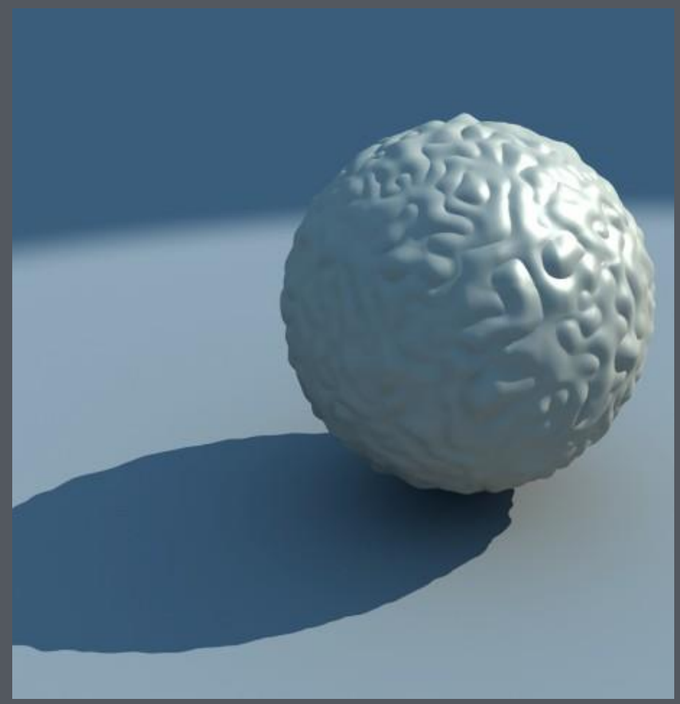
- Also detail mapping



Geometry



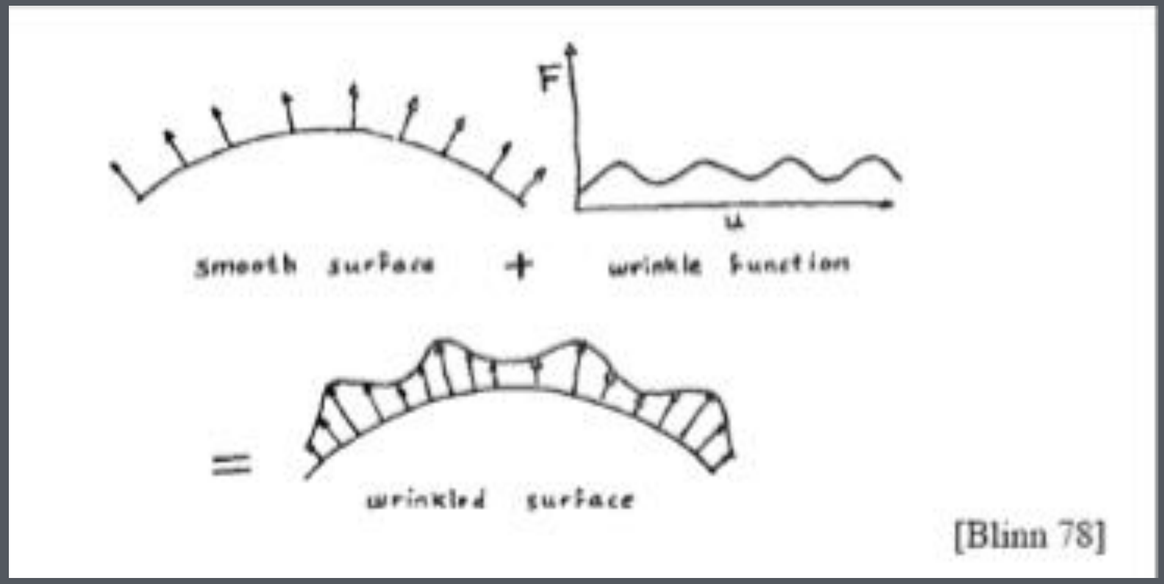
Bump mapping



Displacement mapping

Displacement Mapping

- `Pnew = Pold + DM(u)* N`



$$\mathbf{p}^0(u,v) = \mathbf{p}(u,v) + h(u,v)\mathbf{n}(u,v)$$

Displacement in vertex shader

Displacement Maps

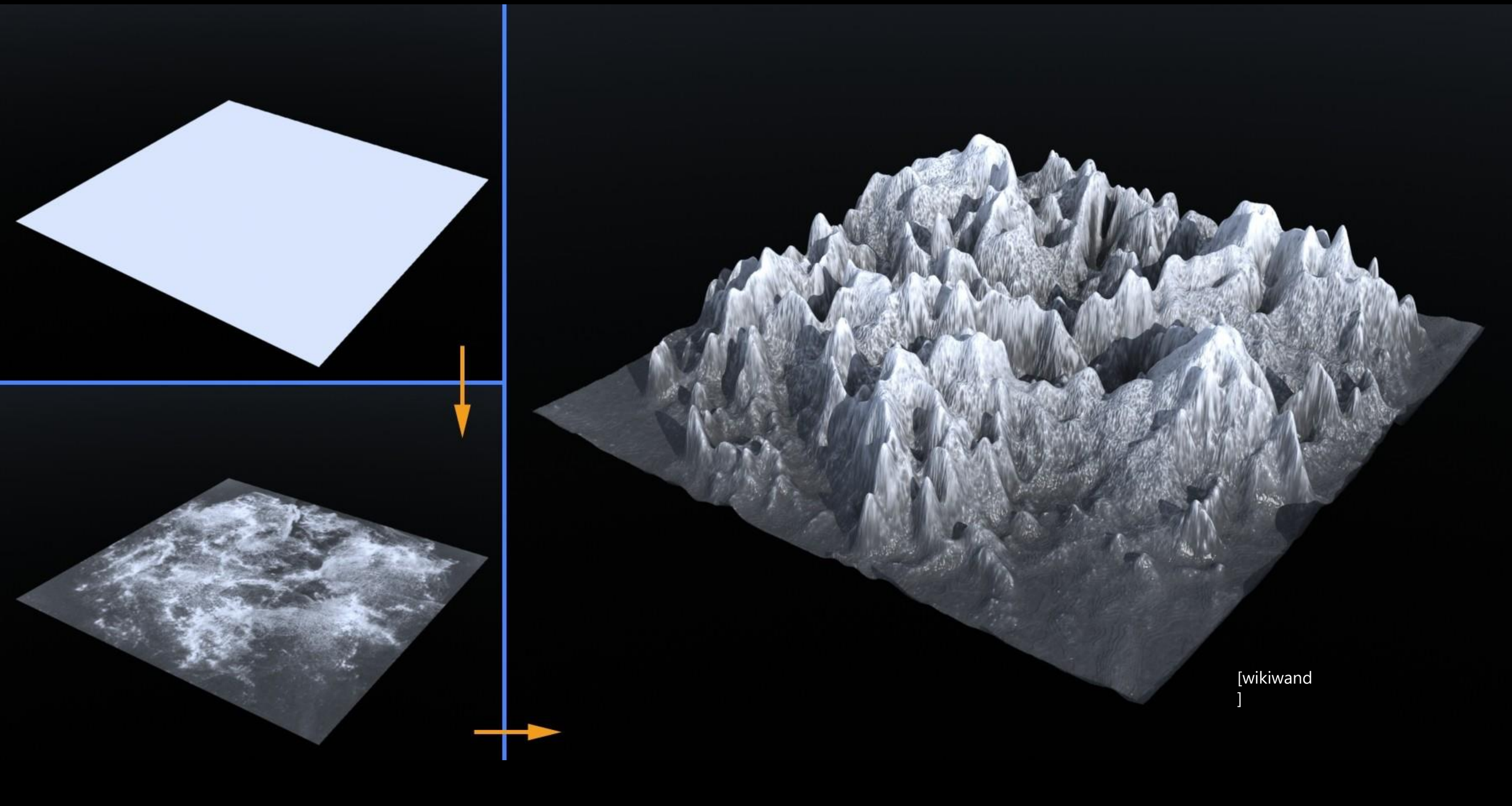
Pros

- Gives you very complex surfaces

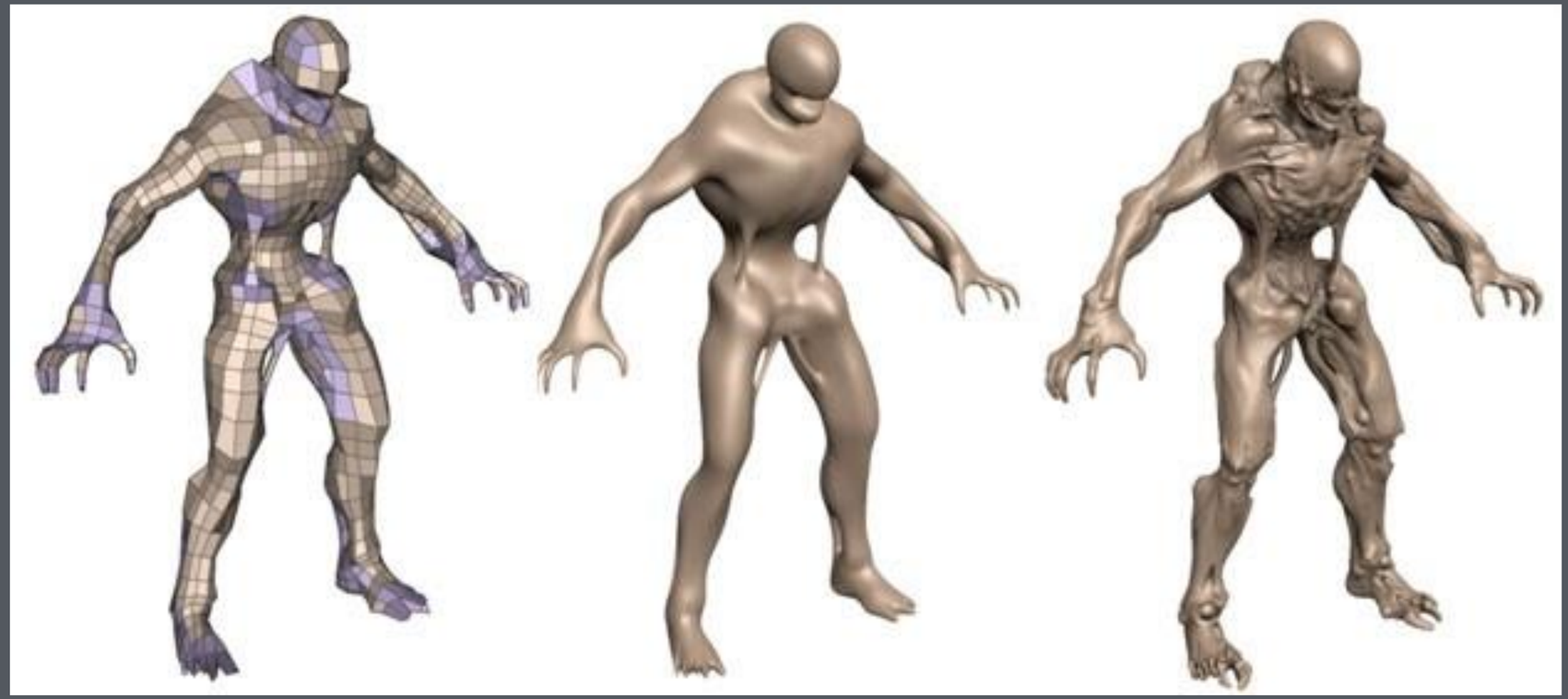
Cons

- Gives you very complex surfaces
- Or boring with small numbers of vertices

Relationship with tesselation shaders



[wikiwand
]



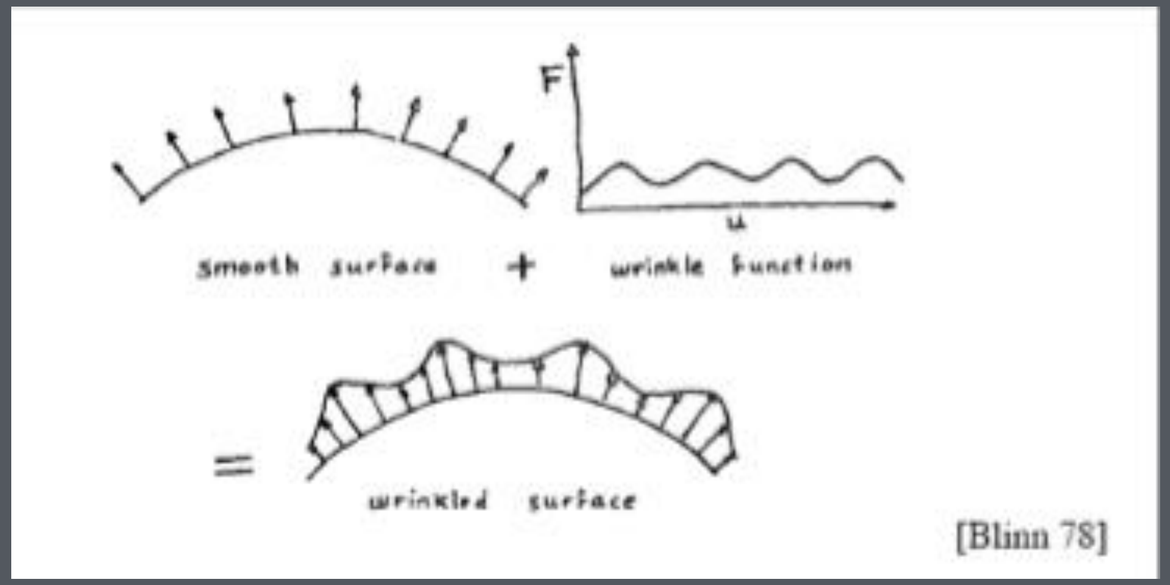
Original

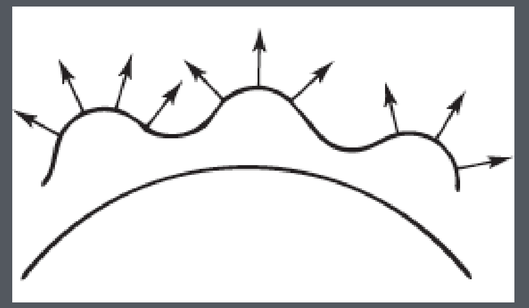
Tesselated

Displac
ement

Bump mapping

“Simulation of Wrinkled Surfaces” Blinn 78

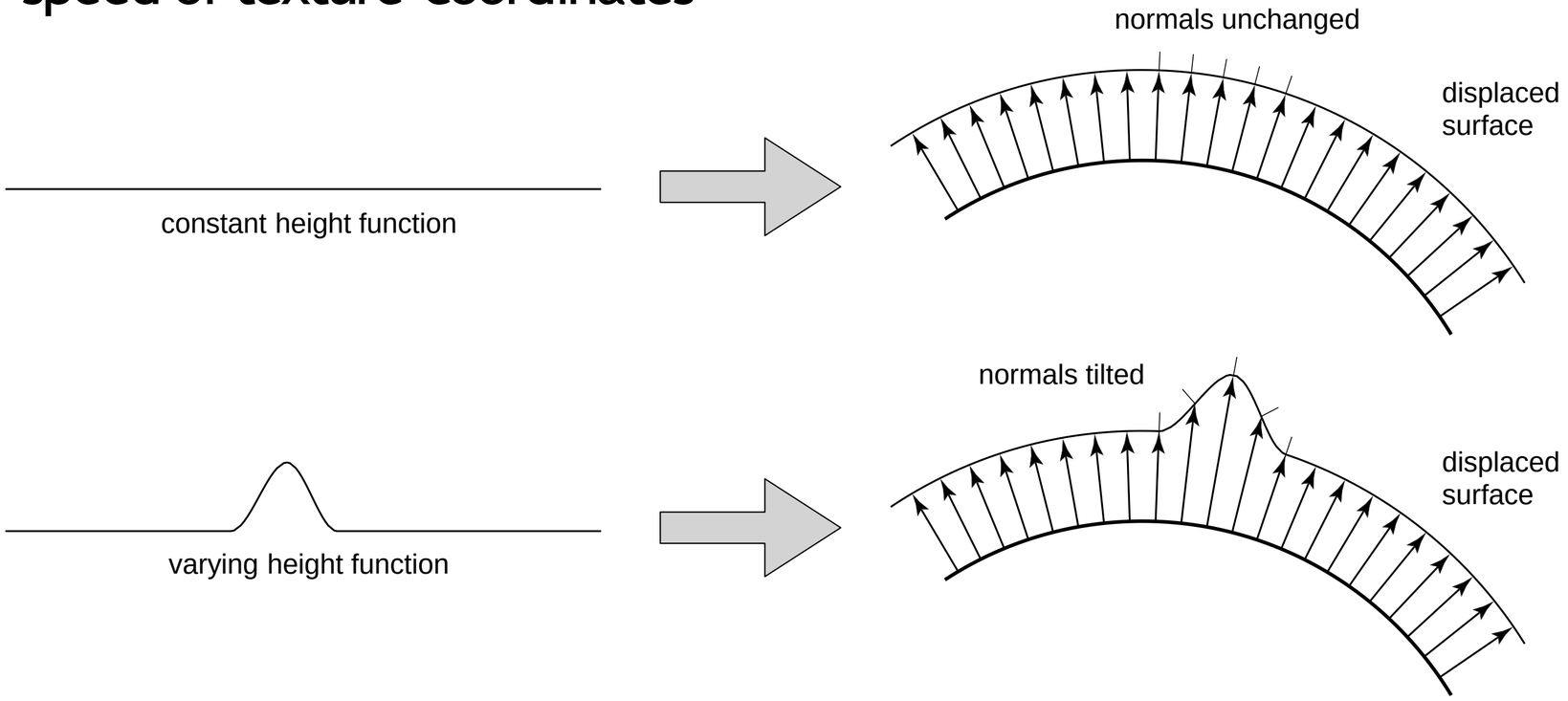




Blinn: keep surface, use new normals

Normals in displacement mapping

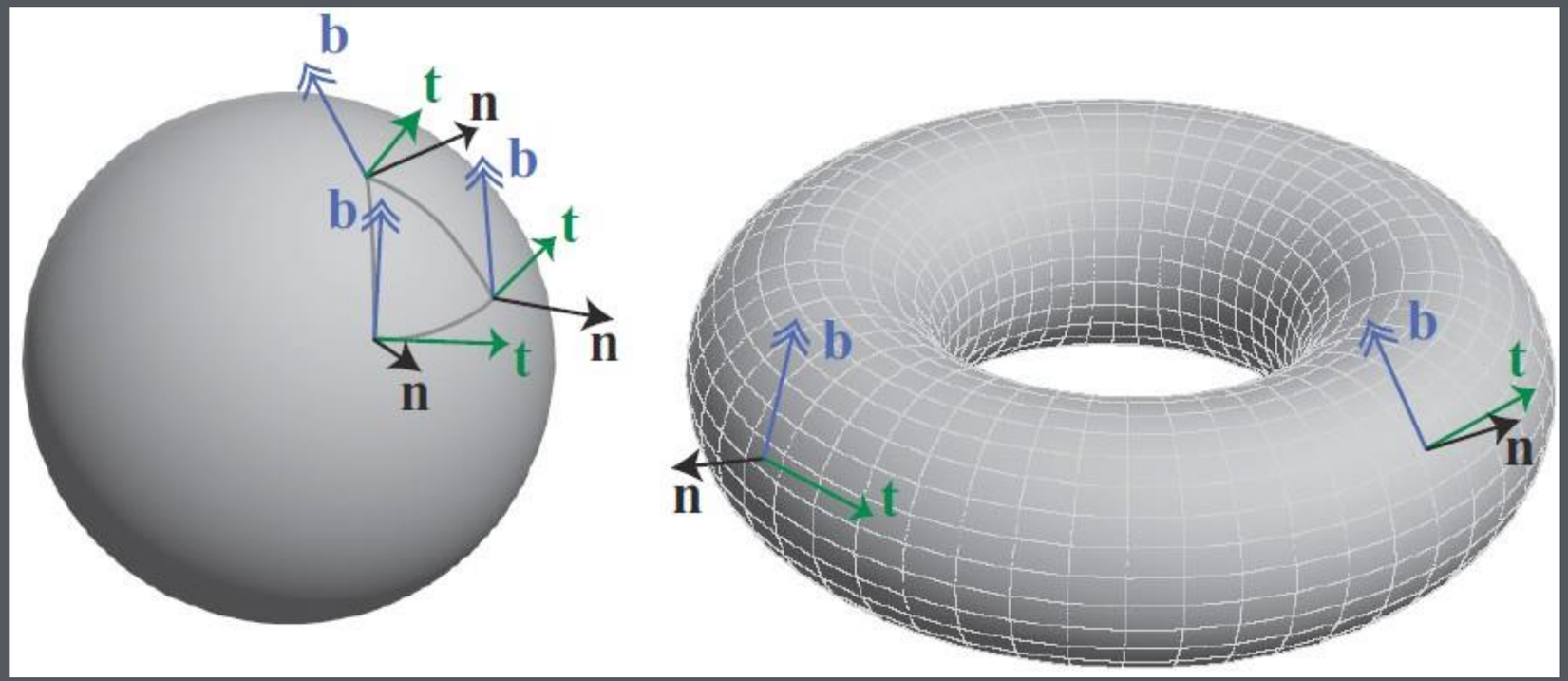
- **Displacement changes the surface normal, depending on:**
  - derivative of height function
  - orientation of texture coordinates
  - speed of texture coordinates



How to change the normal?

First, need some frame of reference

- Normal is modified with respect to that
- Have tangent space basis: t and b
- Normal, tangent and bitangent vectors



Geometry for displacement

- **geometric inputs**
  - *u* tangent (unnormalized) as vertex attribute
  - *v* tangent (unnormalized) as vertex attribute
  - height field as a texture
- **vertex stage**
  - compute displaced vertex position
    - look up displacement value from texture
  - compute normal to displaced surface
    - compute derivatives of height by finite differences
    - add offset to the base surface tangents
    - normalized cross product is the shading normal
- **fragment stage: just compute shading**

(or compute them ahead of time and store height and derivatives in a 3-channel texture)

Bump Mapping

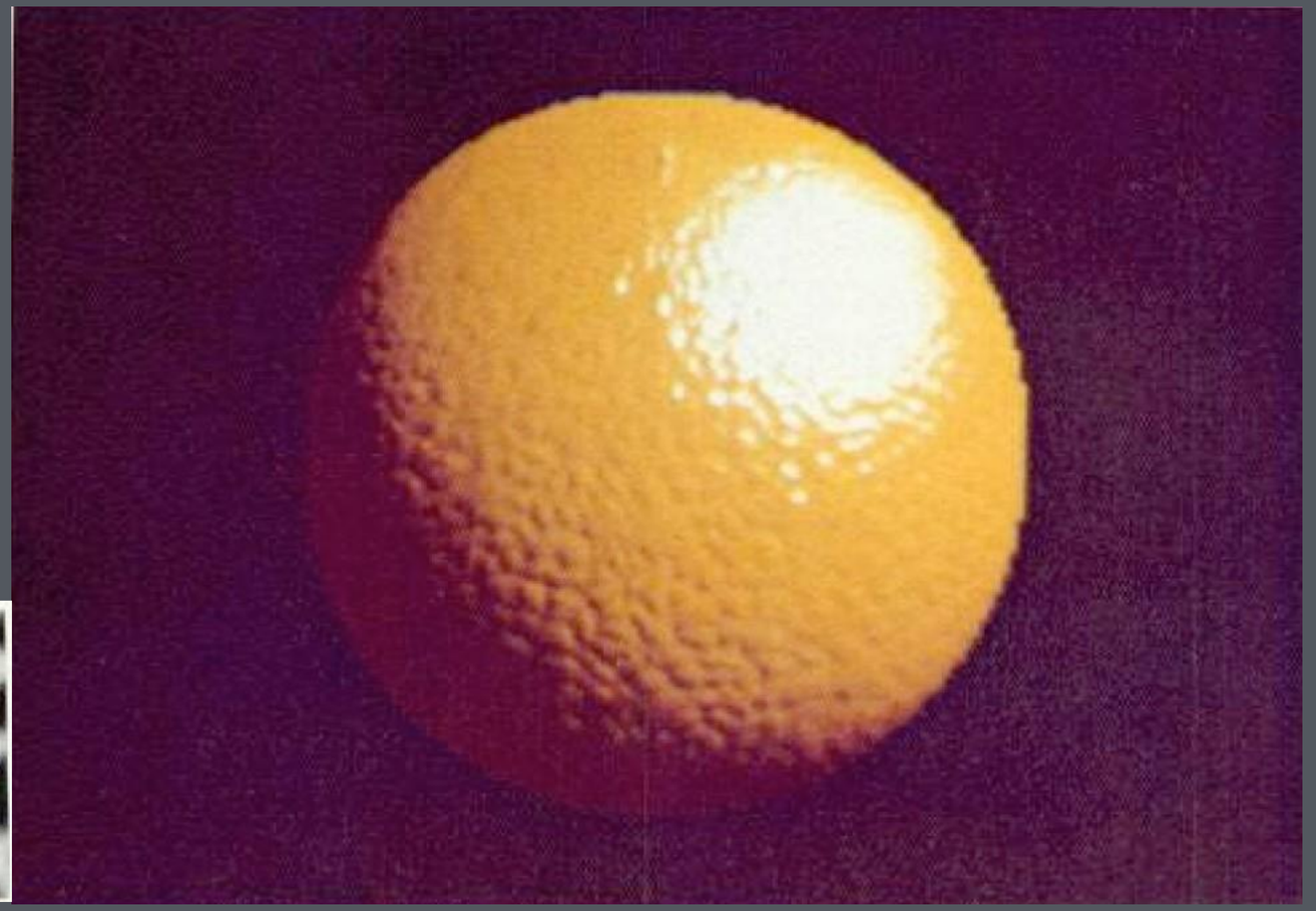
Alter normals of surface

- Only affects shading normals

Also, mimics effect of small scale geometry

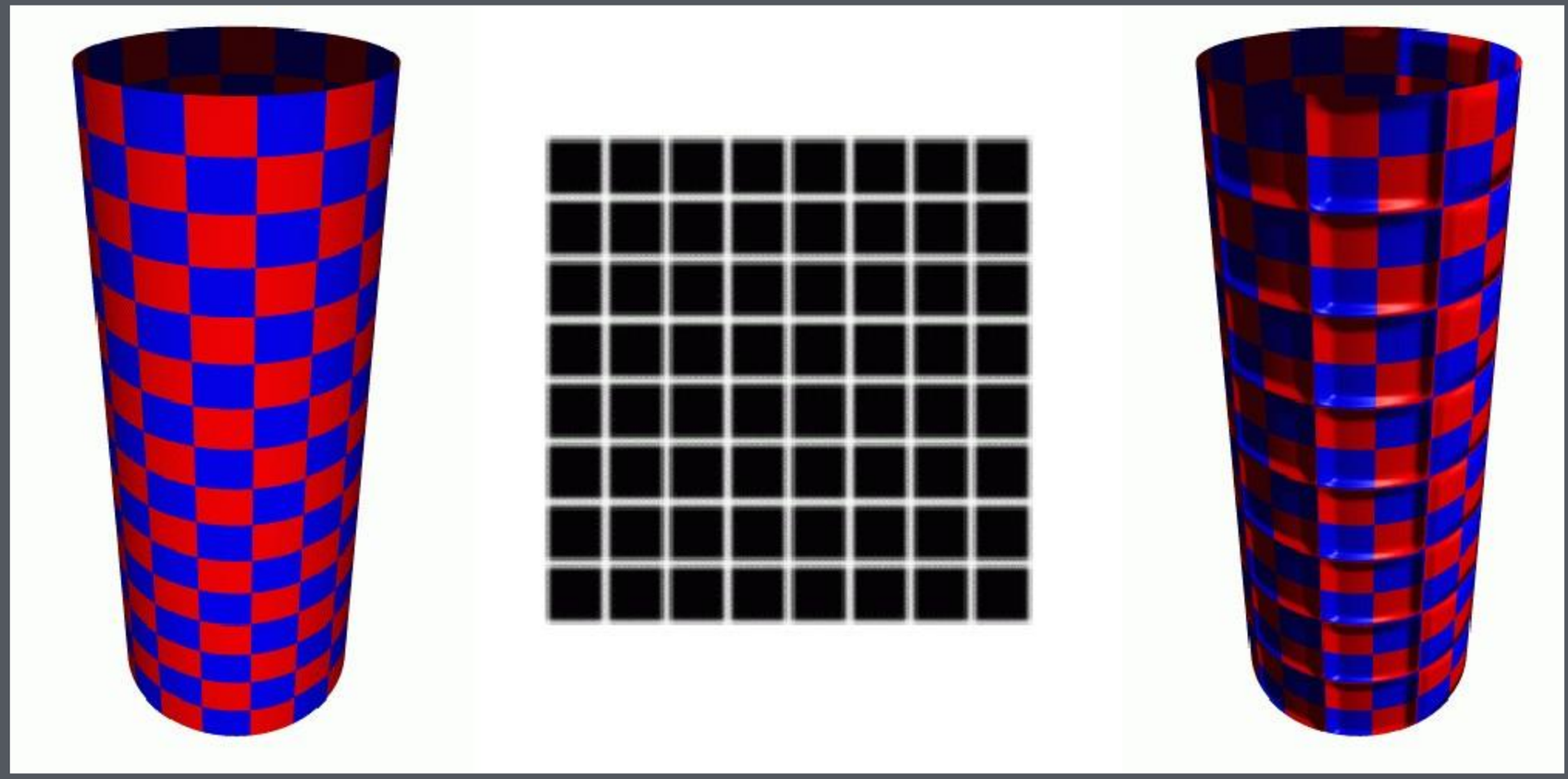
- Detail map
- Except at silhouette
- Adds perceived bumps, wrinkles

Bump mapping



[Blinn 1978]

Bump Mapping



Bump mapping

Displacement mapping is expensive

- requires densely tessellated geometry
- many triangles to rasterize

For small displacements, the most important effect is on the normal

- so just do that part; don’t displace the surface

Bump mapping is then a fragment operation

- doesn’t require dense tessellation
- doesn’t actually displace the surface
- gives shading that looks just like displaced surface

Bump Mapping

Bump mapping

Geometric inputs

- tangent vectors (unnormalized) as vertex attributes
- height field as a texture
- no dense triangulation needed

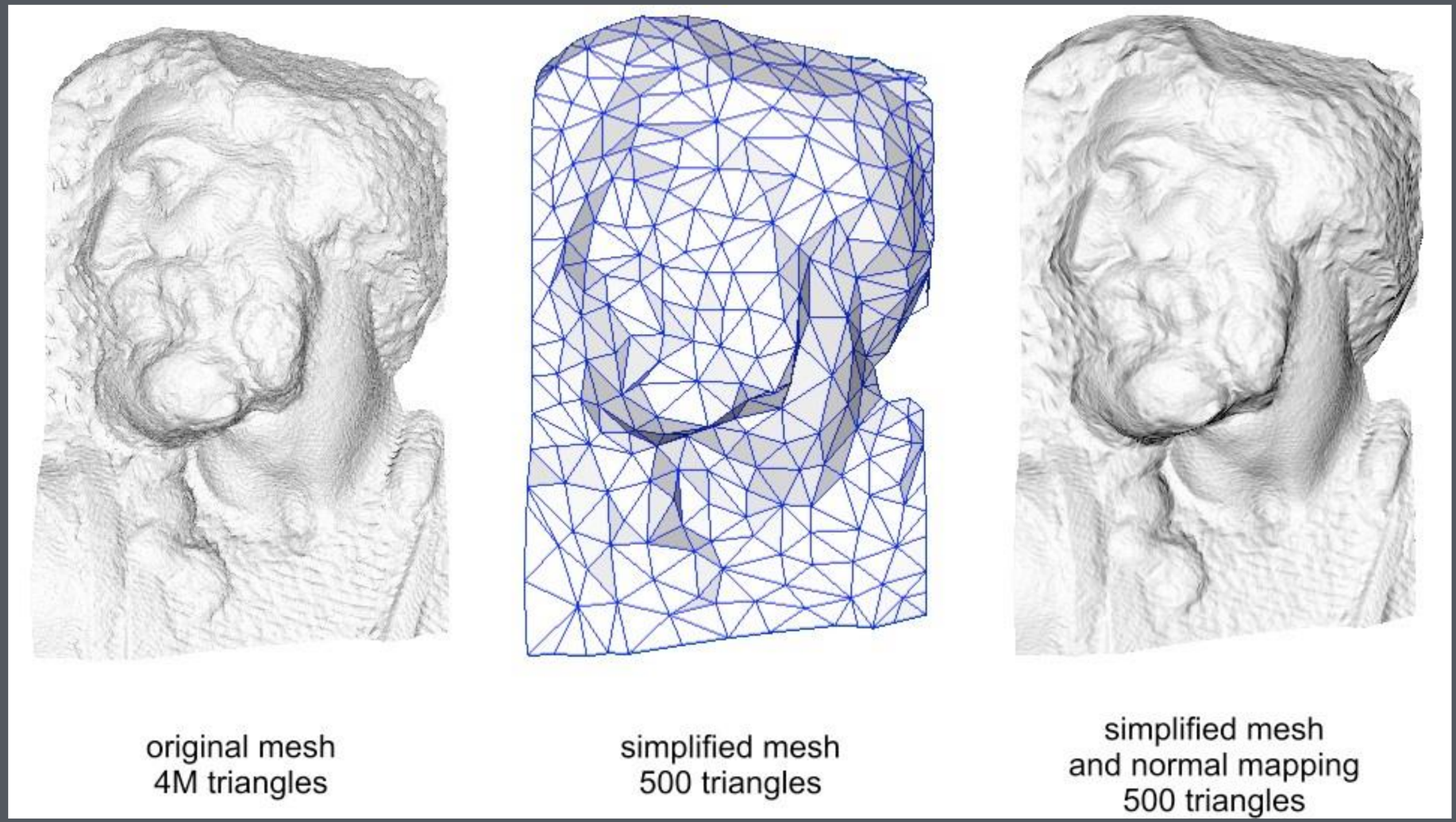
Vertex phase

- simply transform and pass through the position and tangents

Fragment phase

- compute normal to displaced surface
  - compute derivatives of height by finite differences
  - add offset to the base surface tangents; cross product is the shading normal
- compute shading using displaced normal

Normal mapping



[Paolo Cignoni]

Normal mapping

Geometric prerequisites

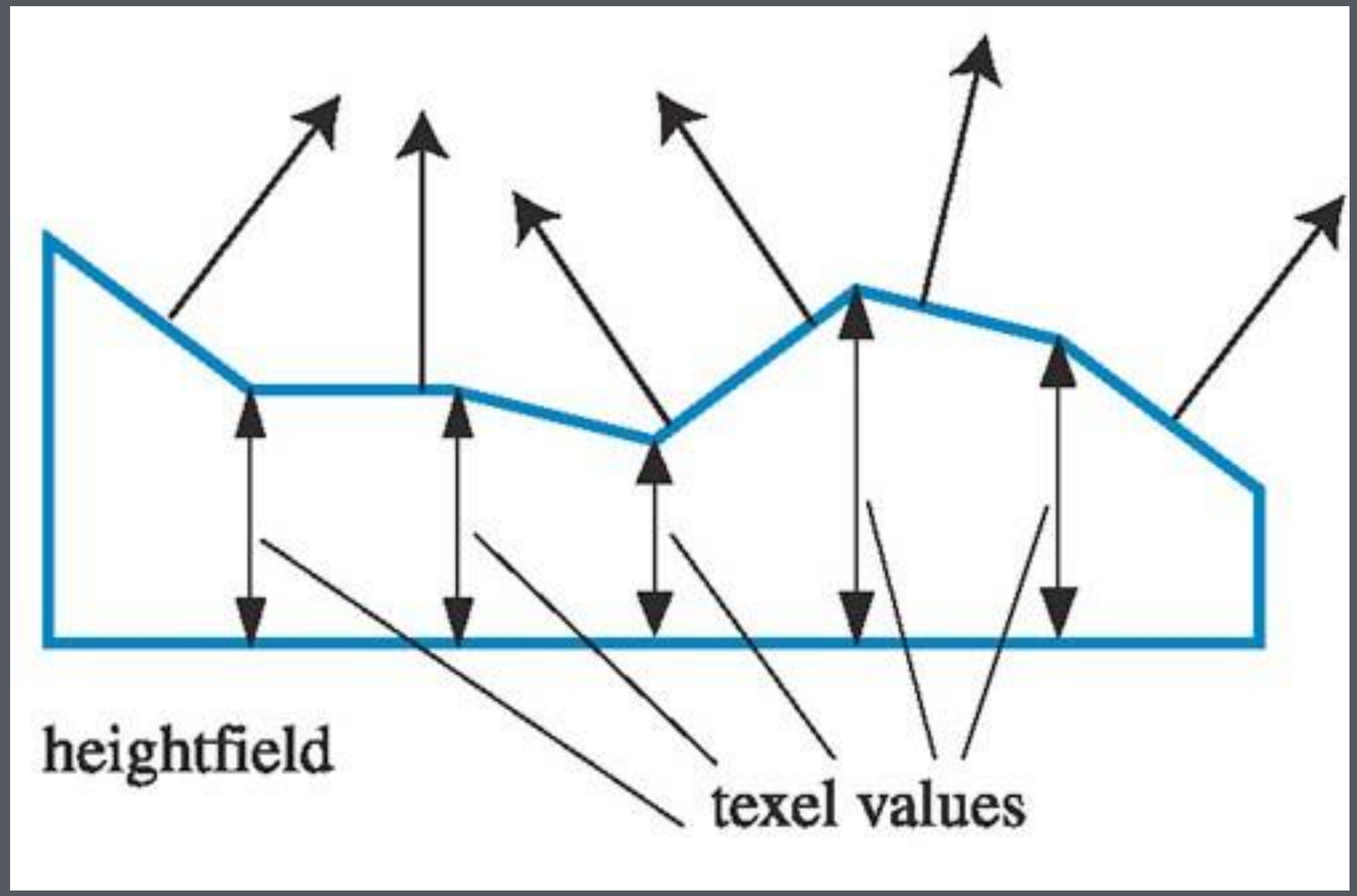
- Texture map (3 channels) representing normal field
  - single lookups into normal map required
- Smooth normals
- Unit tangent vectors
  - if you want to store normals in tangent space (and you do)
- No dense triangulation needed
- No finite differencing needed

Geometric logic

- look up normal from map
- transform into (tangent-u, tangent-v, normal) space

Heightfield: Blinn’s original idea

Single scalar, more computation to infer N’



Perturbed normal given height map

Normal is determined by partial derivatives of height

- in the local frame of the displacement map:

$$\mathbf{n}_{\text{disp}} = (h_u, h_v, 1)$$

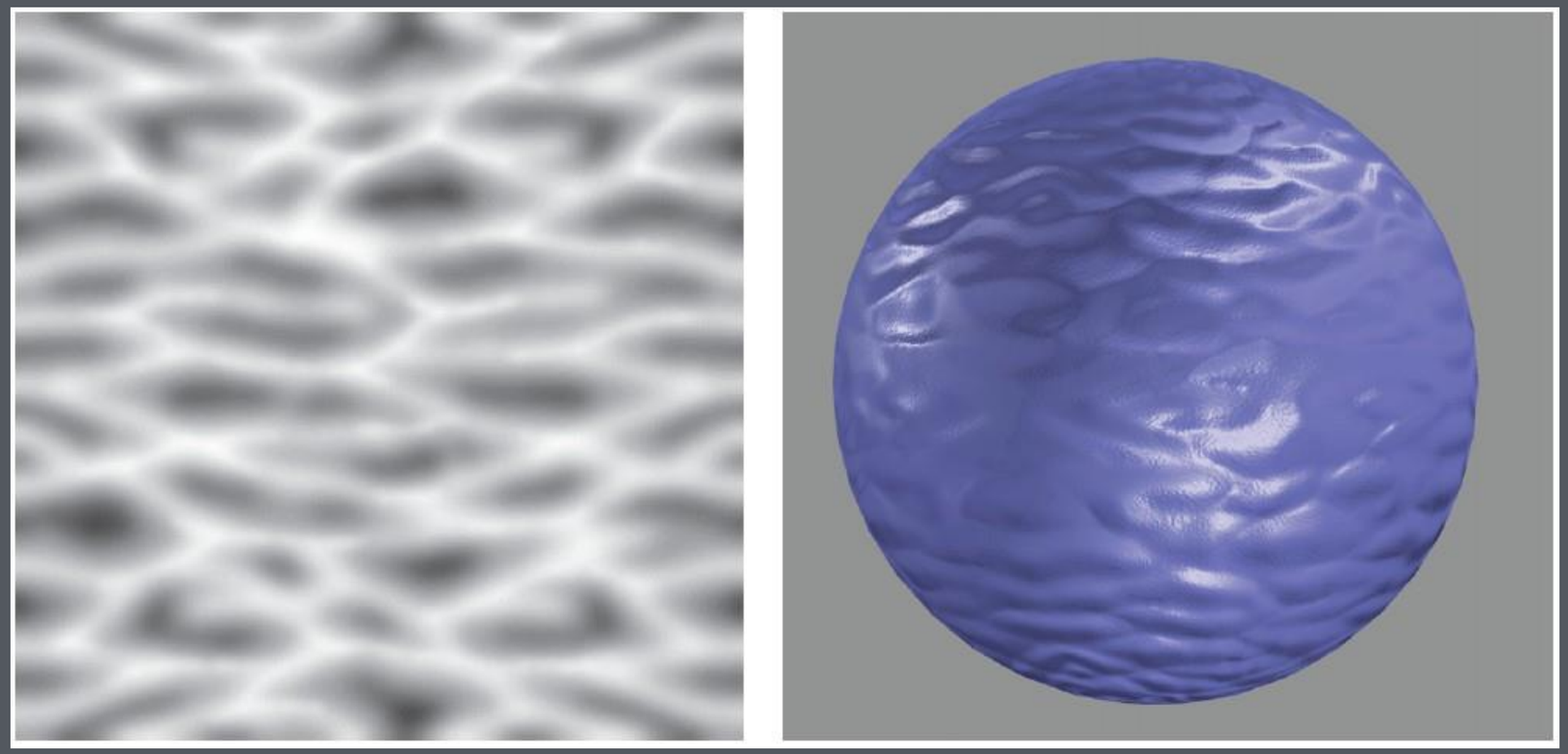
- approx: heights are small compared to radius of curvature (constant normal)
- then the displaced surface is locally a linear transformation of the height field
- normal transforms by the adjoint matrix (as normals always to)
- perform 4 lookups to get 4 neighboring height values
- subtract to obtain finite difference derivatives

Height Field Bump Maps

Older technique, less memory

Texture map value is a height

Gray scale value: light is +, dark is -

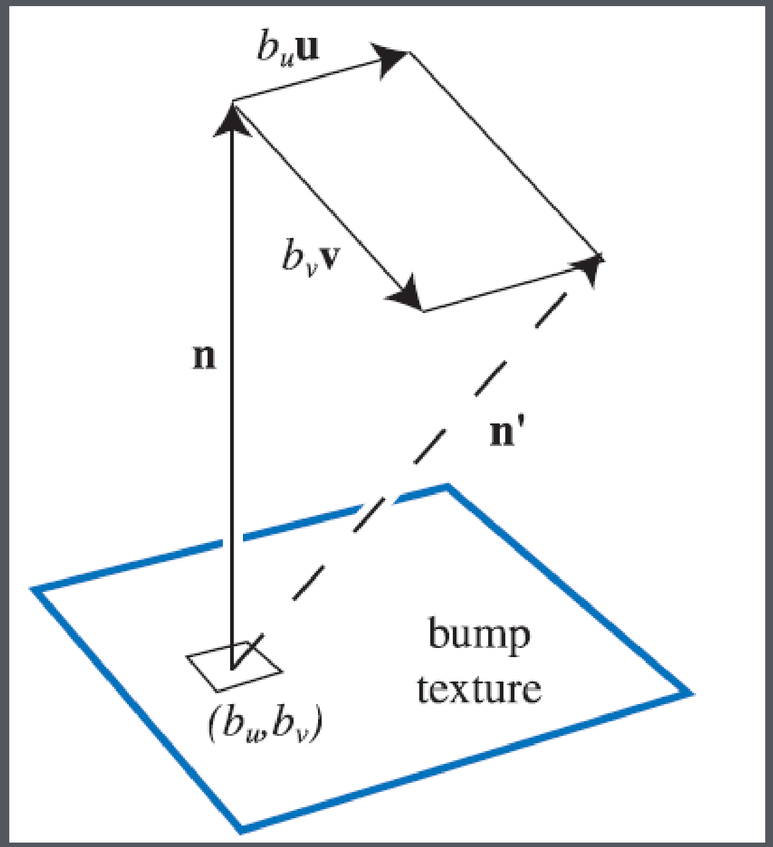


Bump Mapping

Look up bu and bv

N’ is not normalized

N’ = N + bu T + bv B



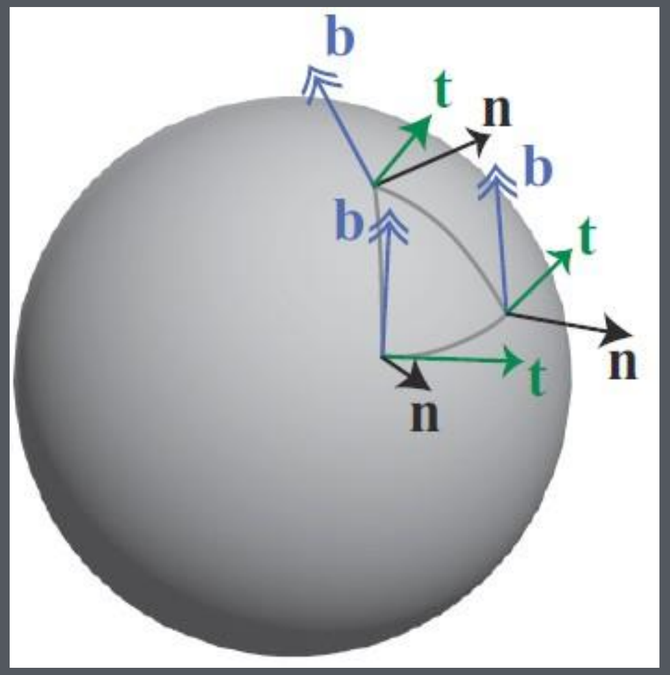
Rendering with Bump Maps

N’.L

Perturb N to get N’ using bump map

Transform L to tangent space of surface

- Have N, T (tangent), bitangent B = T x N



Normal Maps

Preferred technique for bump mapping for modern graphics cards

Store new normals in texture map

- Encodes (x, y, z) mapped to [-1, 1]

More memory but lower computation

Normal Map

Height Map

- Store

```
colorComponent = 0.5 * normalComponent + 0.5
```

-

- Use

```
normalComponent = 2* colorComponent -1
```

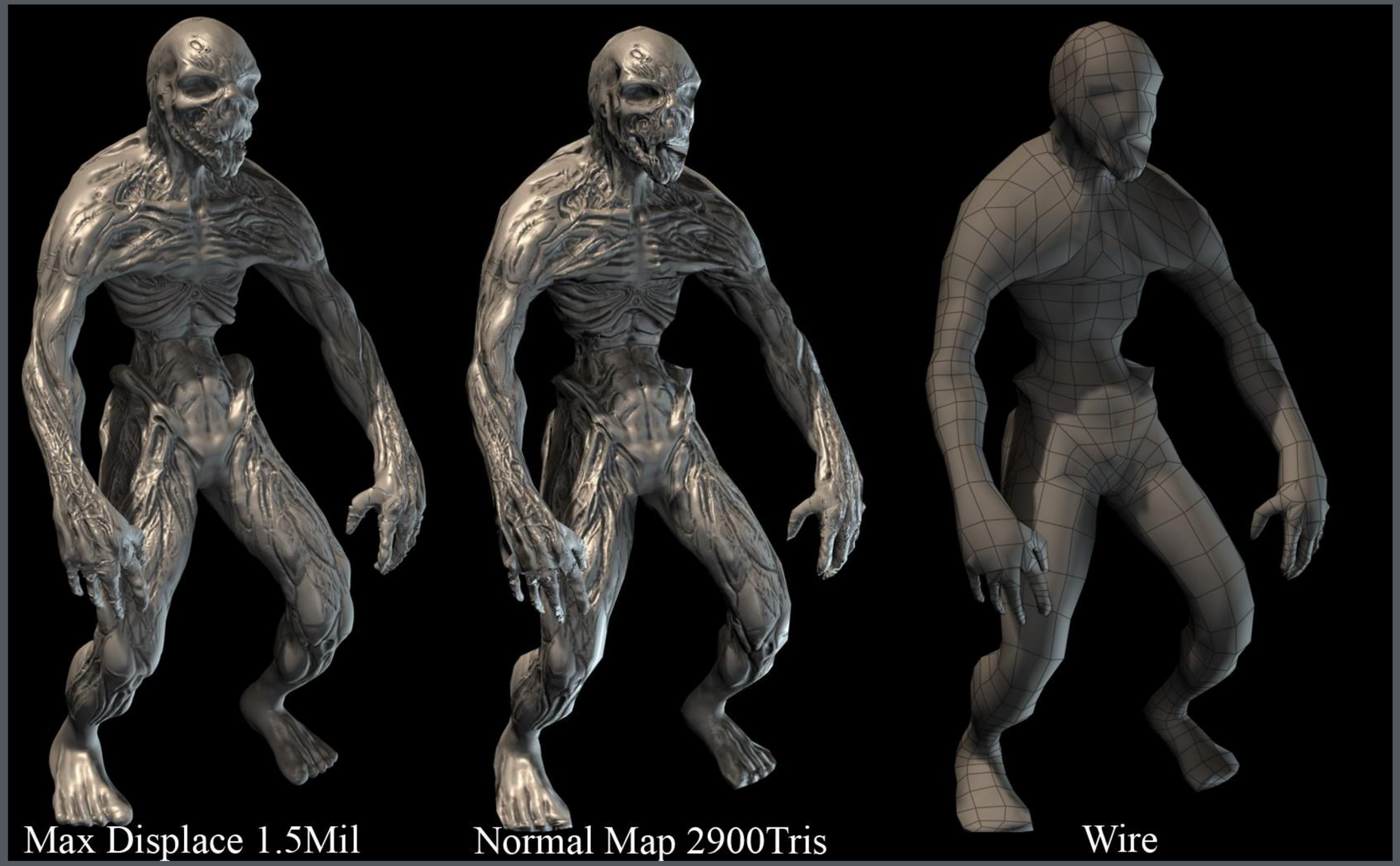
Normal Map

Creating Normal Maps

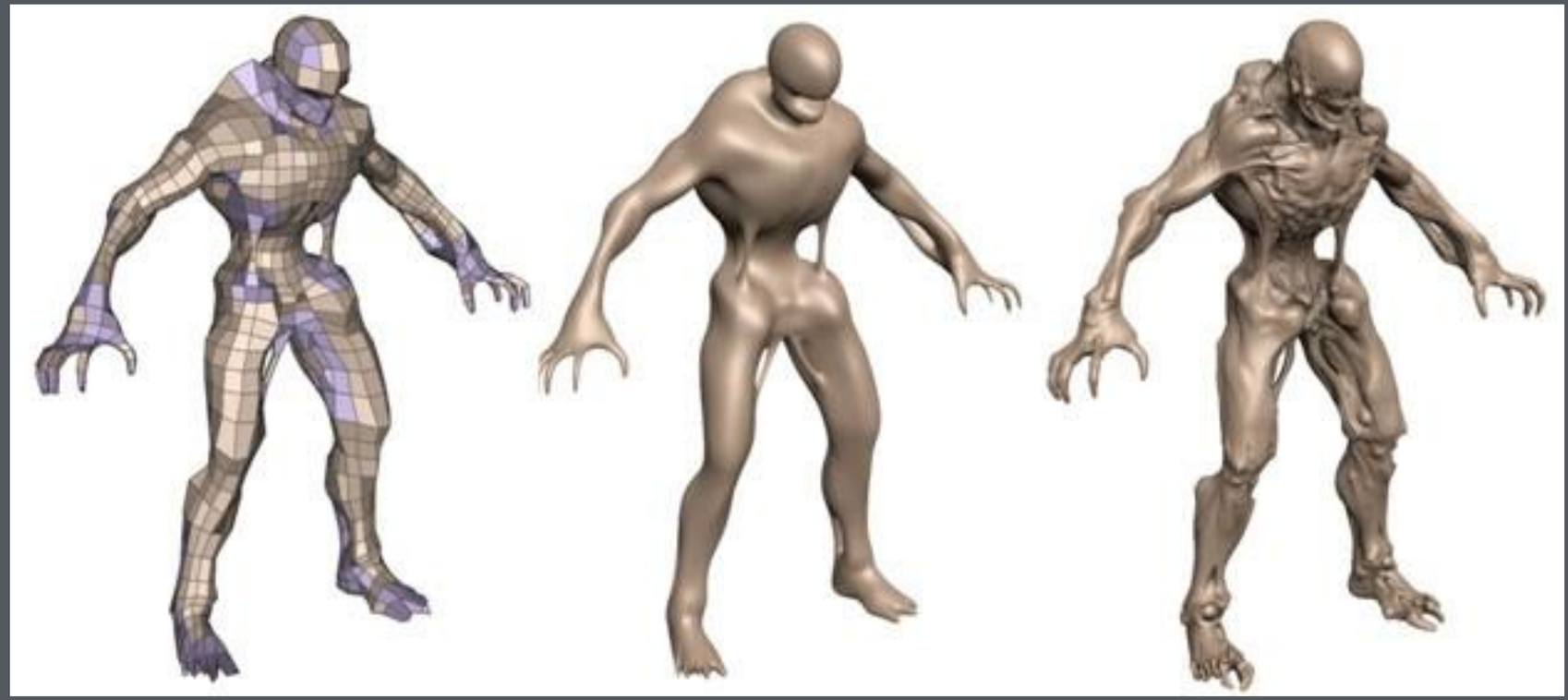
First create complex geometry

Simplify (in modeling time) to simple mesh with normal map

Displacement Maps vs. Normal Maps



Compare with the opposite view

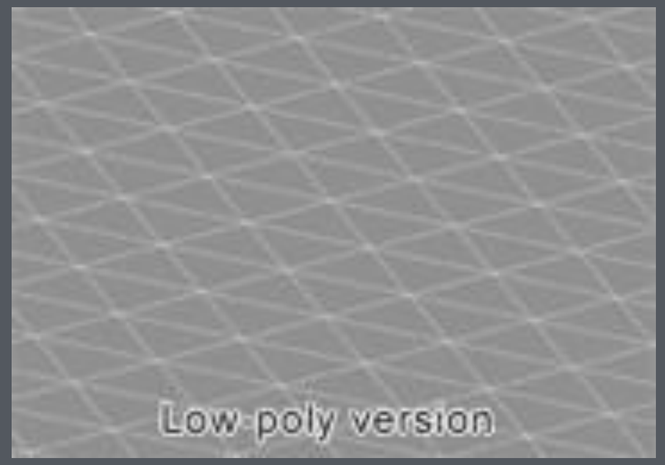


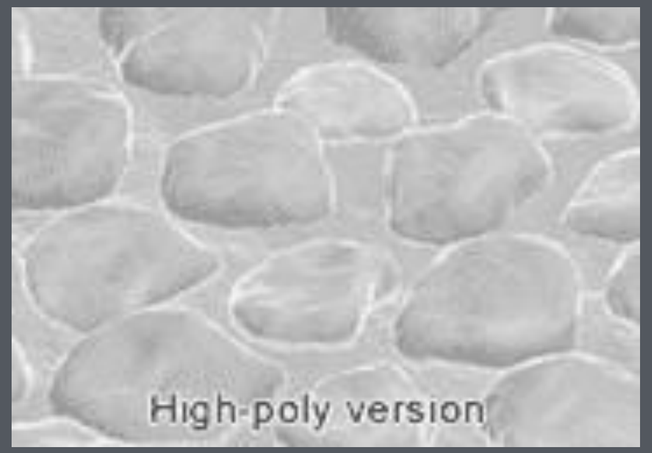
Original

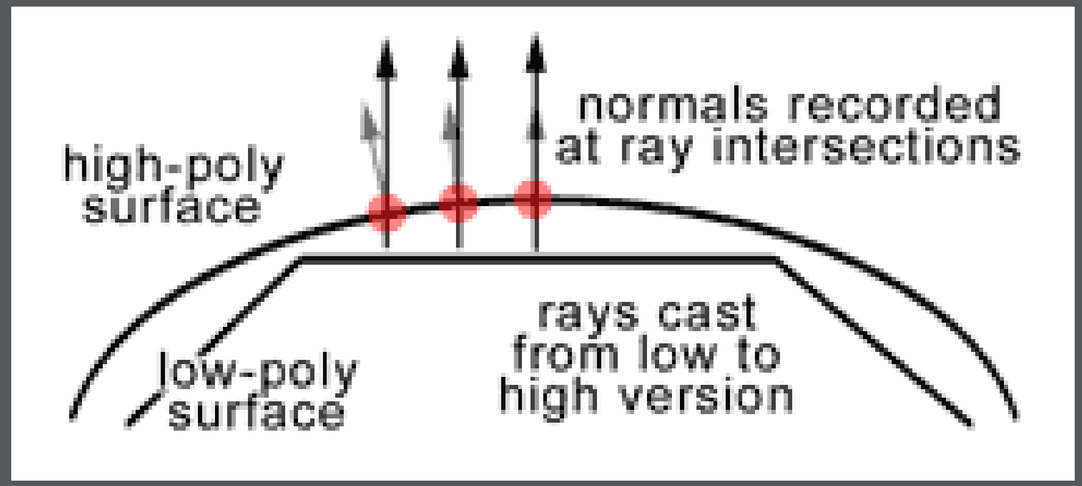
Tesselated

Displacement Mapped

Creating Normal Maps







Which space is normal map in?

World space

- Easy computation, but can’t use the same normal map for…
  - two walls
  - A rotating object

Object space

- Better, but cannot be reused for…
  - deforming objects
  - different objects with similar material

Tangent space

- Can reuse for deforming surfaces
- Transform lighting to this space and shade

Parallax Mapping

Problem with normal mapping

- No self-occlusion
- Supposed to be a height field but never see this occlusion across different viewing angles

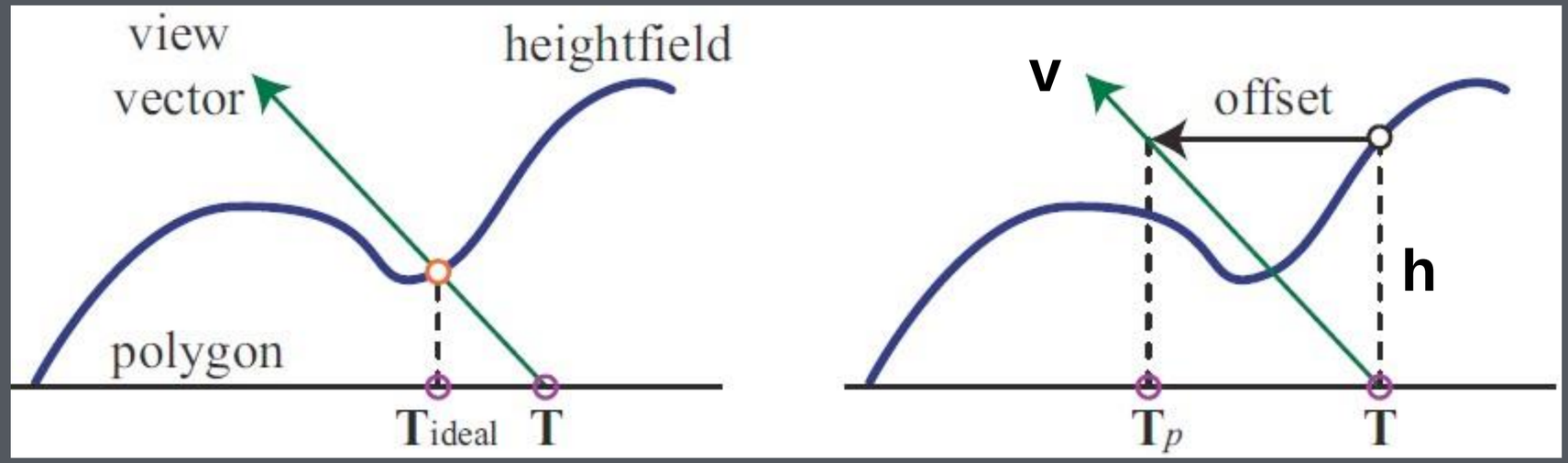
Parallax mapping

- Positions of objects move relative to one other as viewpoint changes

Parallax Mapping

Want T_{ideal}

Use T_p to approximate it



$$\mathbf{T}_p = \mathbf{T} + h\frac{\mathbf{v}_{xy}}{\mathbf{v}_z}$$

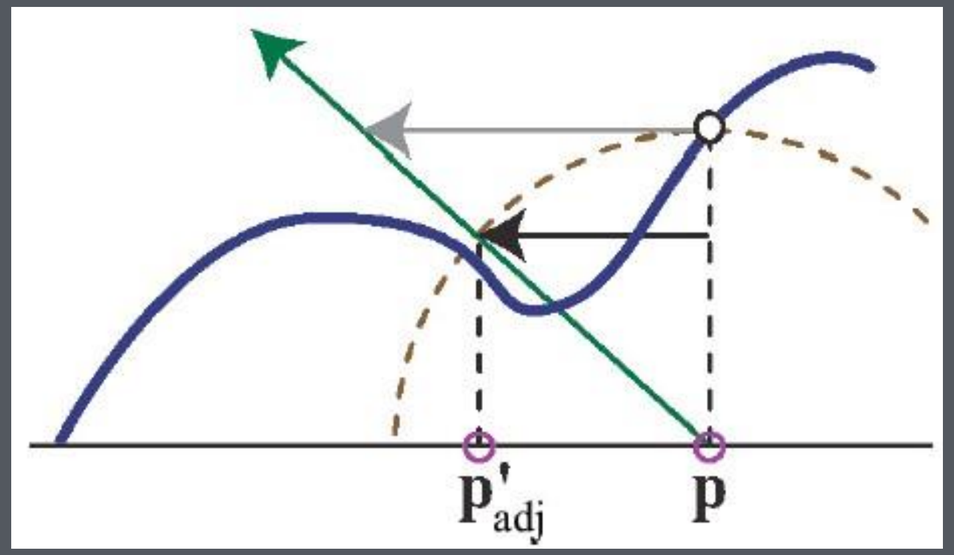
Parallax Offset Limiting

Problem: at steep viewing, can offset too much

Limit offset

- results in a simpler formula

$$\mathbf{p}_{\text{adj}} = \mathbf{p} + h\mathbf{v}_{xy}$$



Parallax Offset Limiting

Widely used in games

- the standard in simple bump mapping

Normal Mapping

Parallax Mapping Offset Limiting

1,100 polygon object w/
parallax occlusion mapping

1.5 million polygon

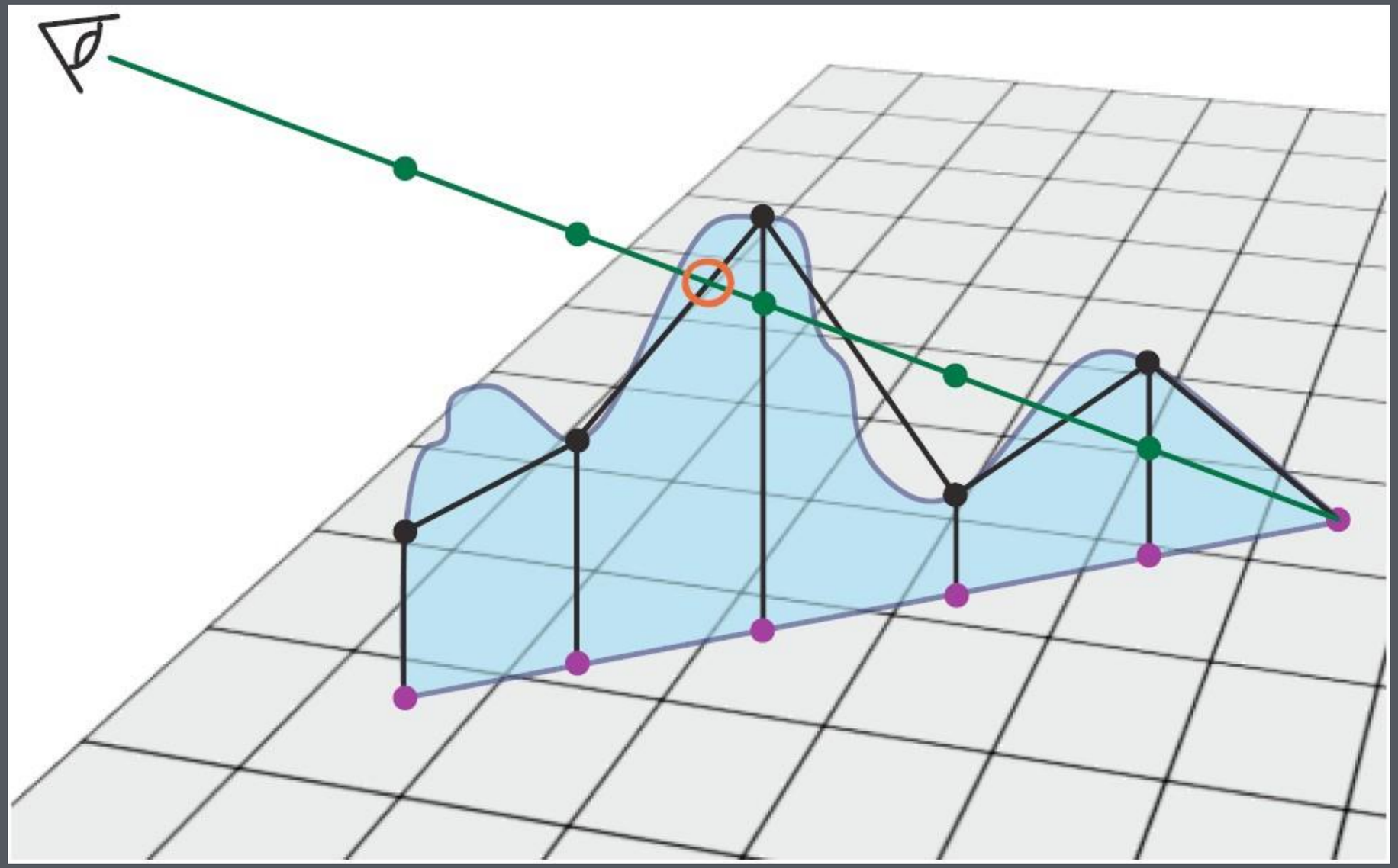
Relief Mapping

Aka Parallax occlusion mapping, relief mapping, steep parallax mapping

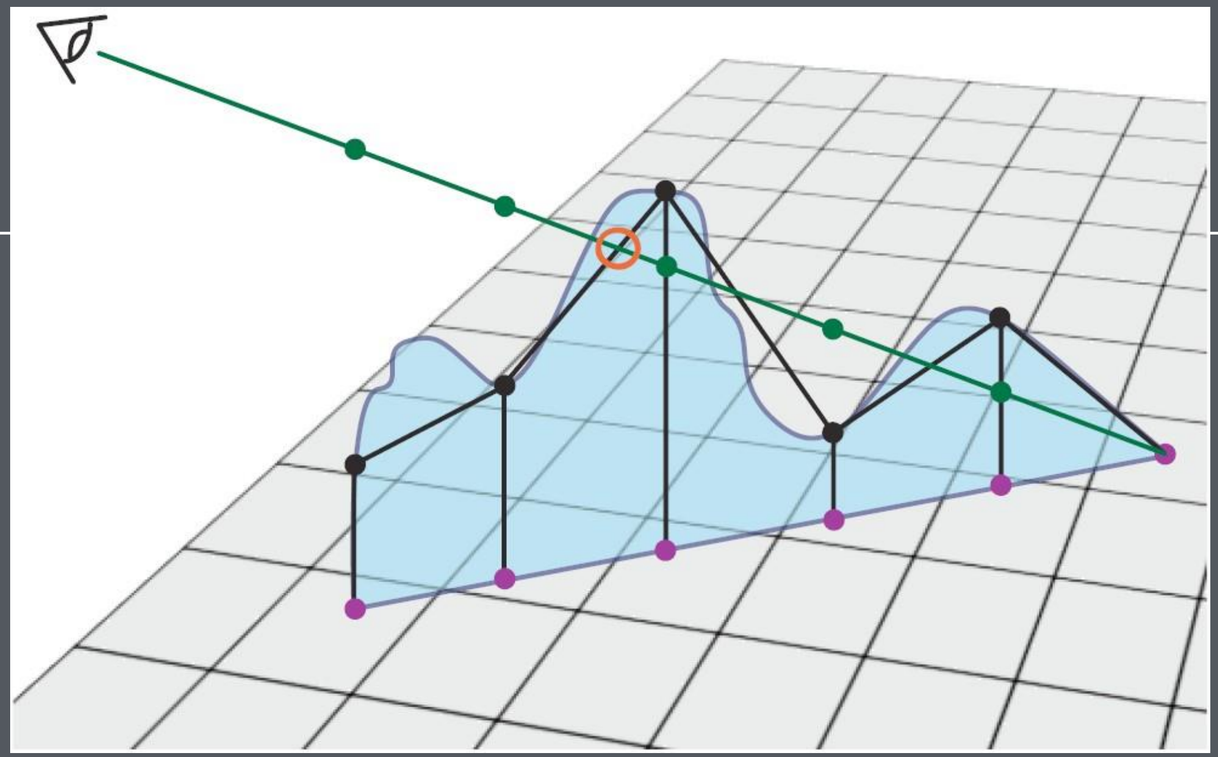
Tries to find where the view ray intersects the height field

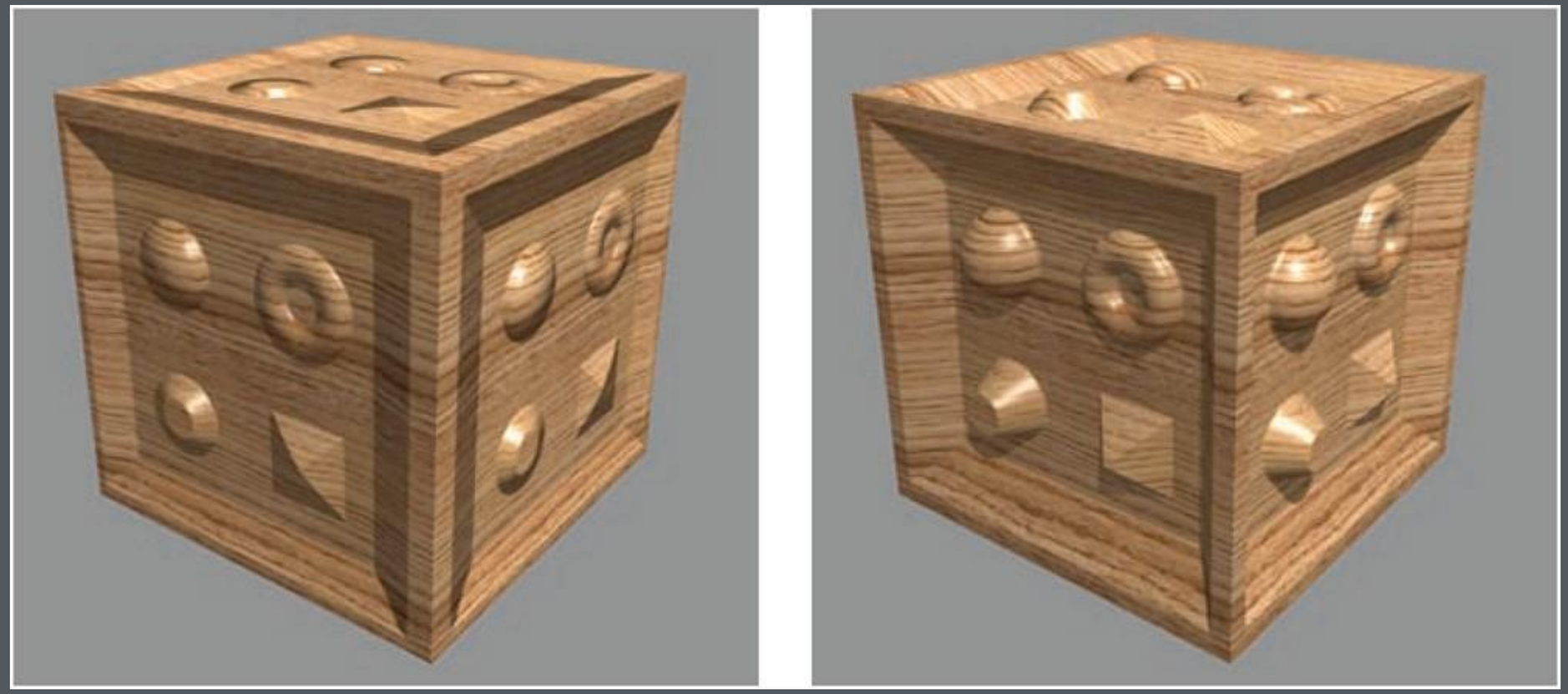
- Kinda

Relief Mapping



```
Sample along ray (green points)
Lookup violet points (texture values)
   /* Inferthe black line shape */
Compare green points with black points
Find intersect between two conditions
   prev: green above black
   next: green below black
```





Parallax Mapping

Relief Mapping

http://www.youtube.com/watch?v=5gorm90TXJM

Crysis, Crytek