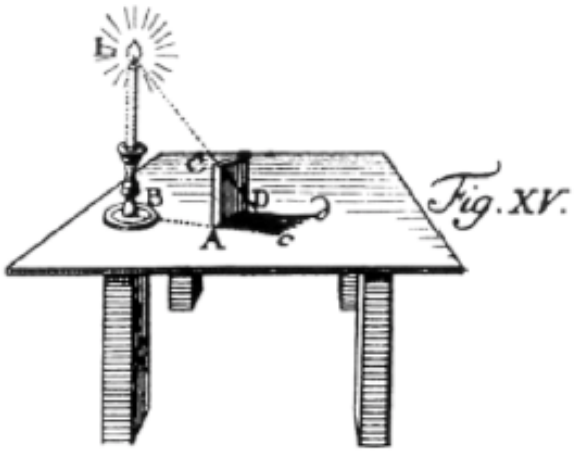
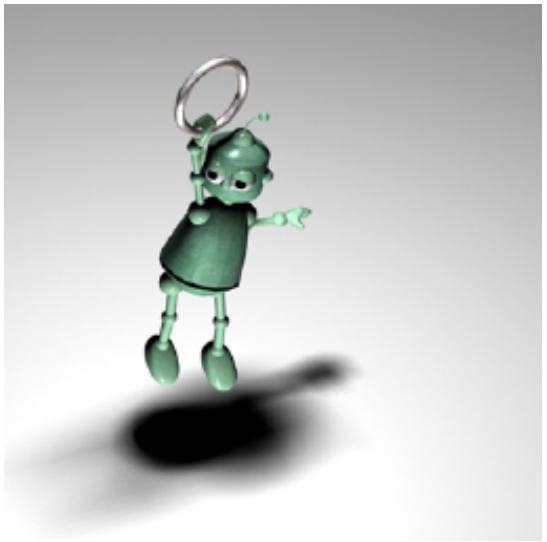
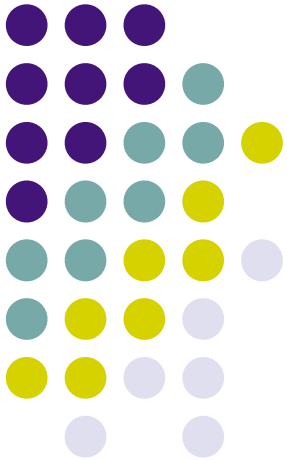
Shadow Algorithms

CSE 781 Winter 2010

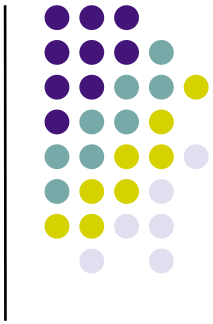




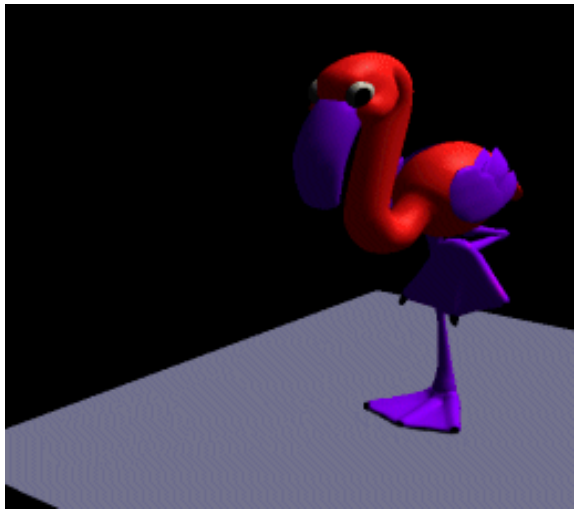


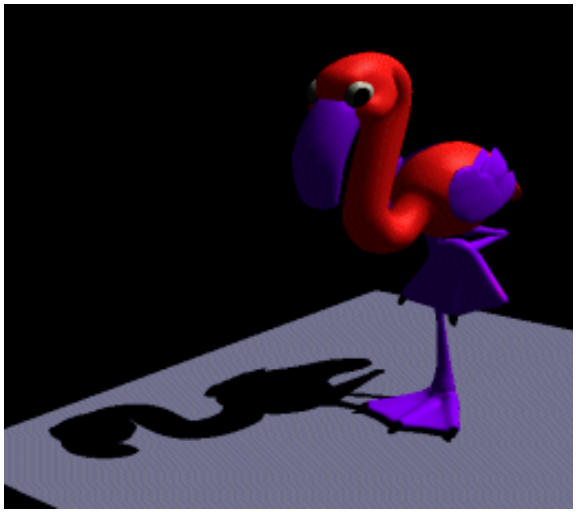
Han-Wei Shen

Why Shadows?



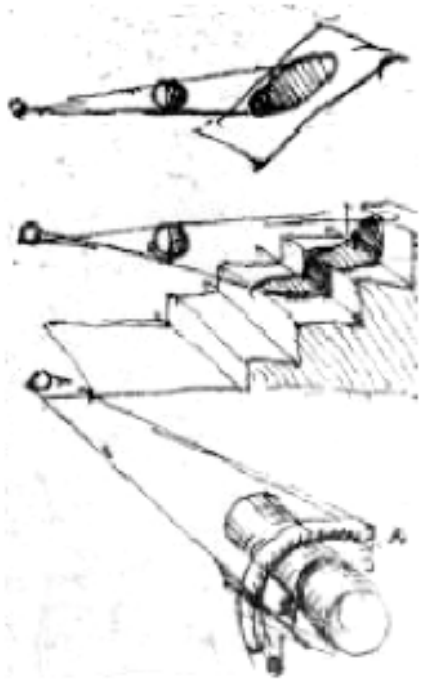
- Makes 3D Graphics more believable
- Provides additional cues for the shapes and relative positions of objects in 3D



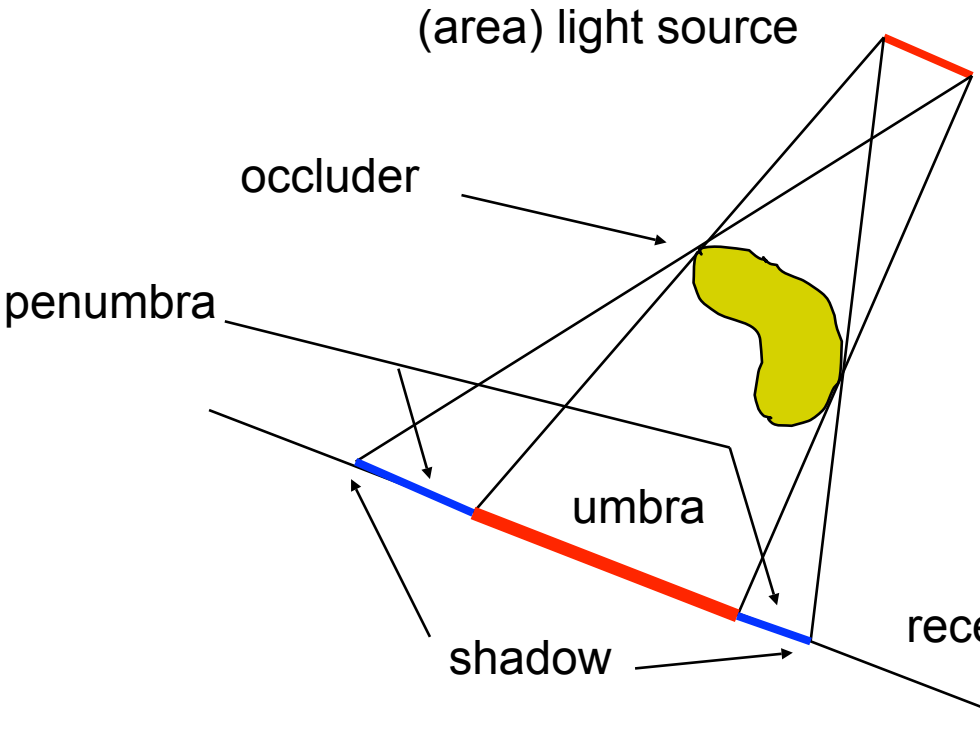


What is shadow?

- Shadow: comparative darkness given by shelter from direct light; patch of shade projected by a body intercepting light



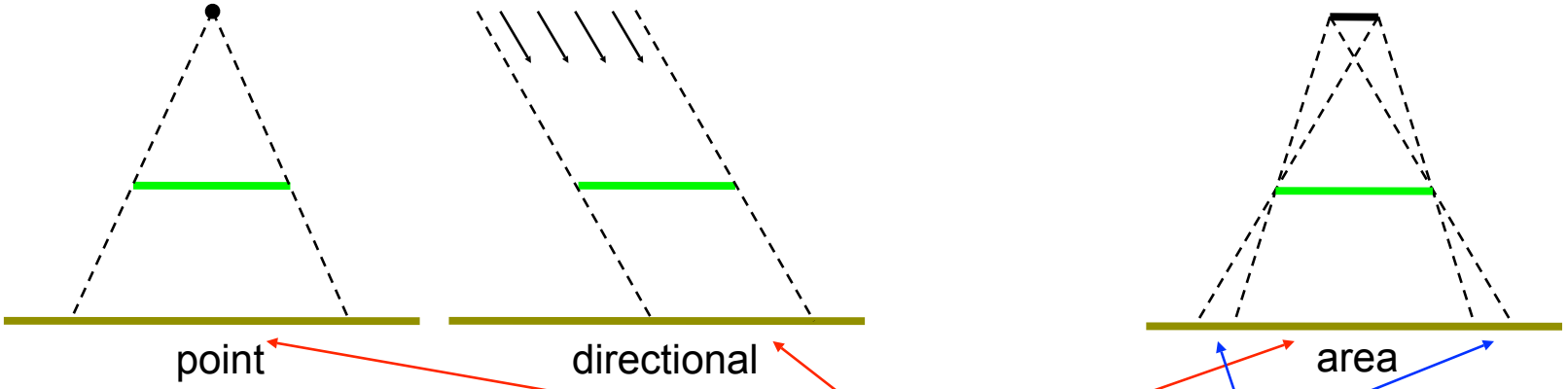
Terminology



- **umbra** – fully shadowed region
- **penumbra** – partially shadowed region

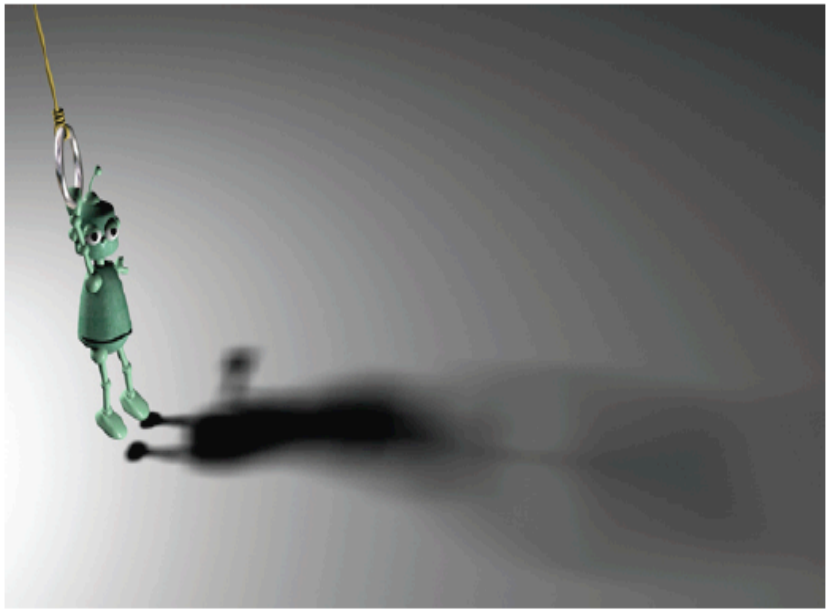
“Hard” and “Soft” Shadows

- Depends on the type of light sources
  - Point or Directional (“Hard Shadows”, umbra)



  - Area (“Soft Shadows”, *umbra, penumbra*), more difficult problem

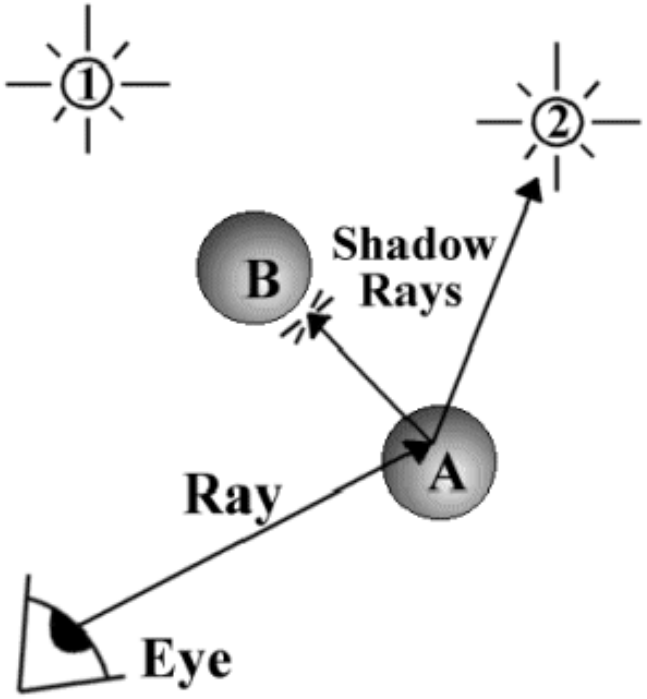
“Hard” and “Soft” Shadows



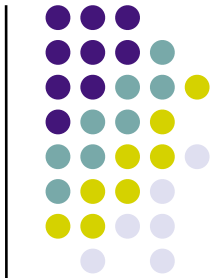
- Hard shadow
  - *point* light source
- Soft shadow
  - *area* light source

Simple Approach: Ray tracing

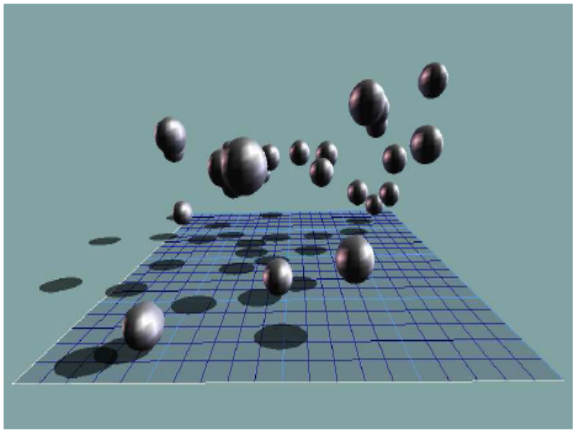
- Cast ray to light (*shadow rays*)
- Surface point in shadow if the shadow rays hits an occluder object.
- Ray tracing is slow, can we do better? (perhaps at the cost of quality)

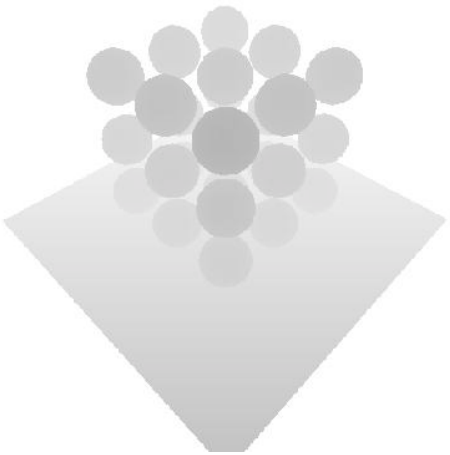


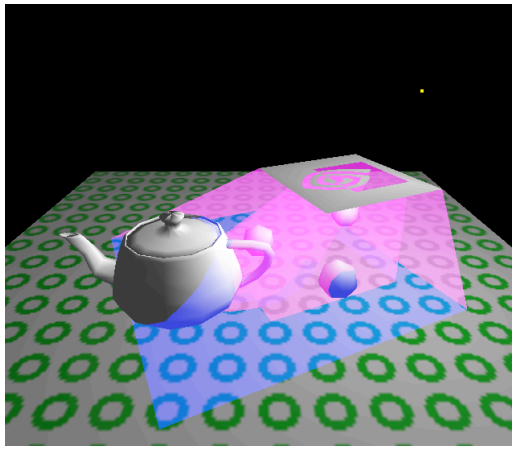
Shadow Algorithms



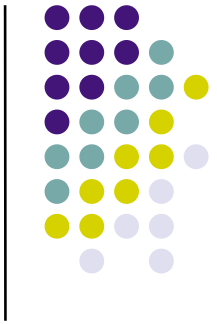
- We will first focus on hard shadows
  - Planar Shadows
  - Shadow Maps
  - Shadow Volume



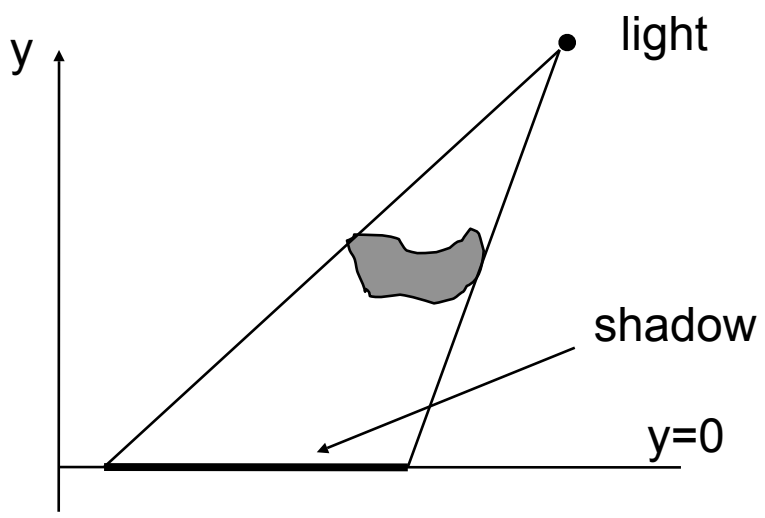


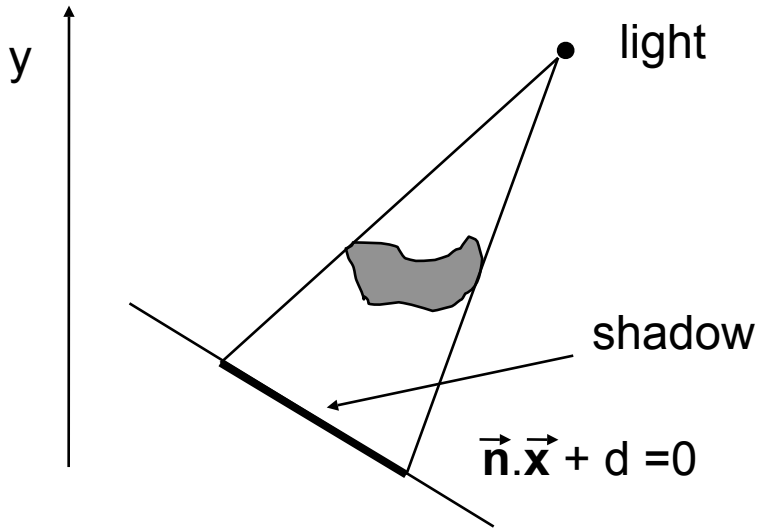


Planar Shadows

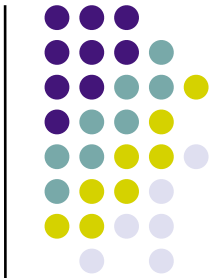


- The simplest algorithm – shadowing occurs when objects cast shadows on planar surfaces (projection shadows)

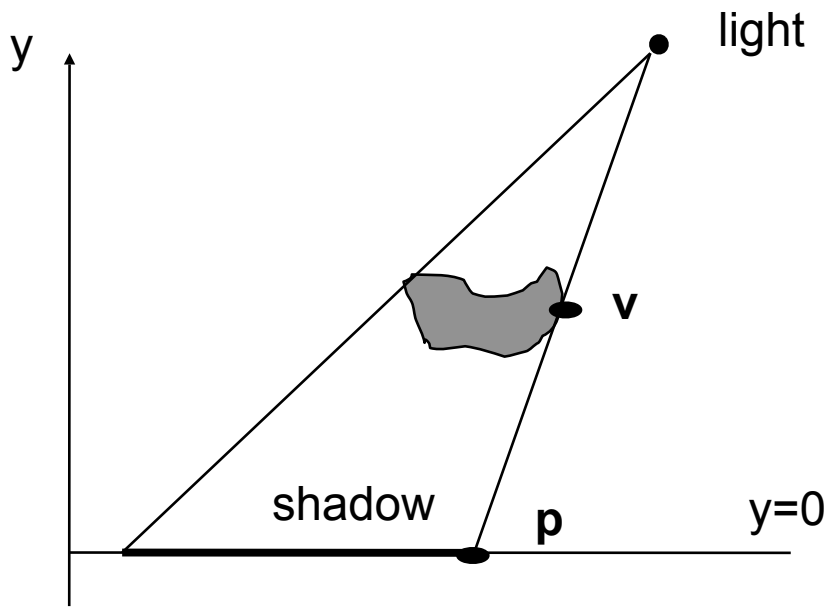




Planar Shadows



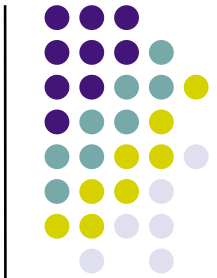
- Special case: the shadow receiver is an axis plane
  - Just project all the polygon vertices to that plane and form shadow polygons



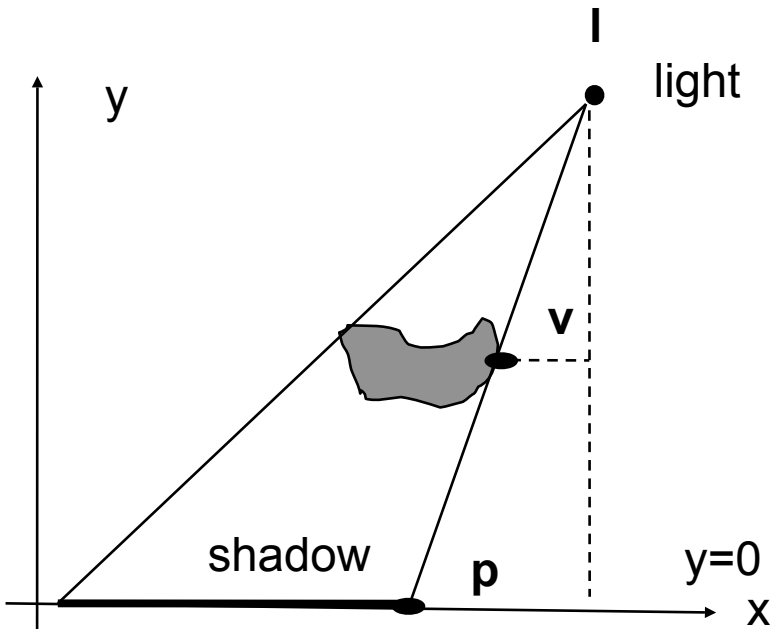
Given:
- Light position **l**
- Plane position y = 0
- Vertex position v

Calculate: **p**

Planar Shadows



- We can use similar triangles to solve **p**

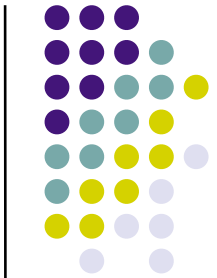


$$\frac{\mathbf{p}_x - \mathbf{l}_x}{\mathbf{v}_x - \mathbf{l}_x} = \frac{\mathbf{l}_y}{\mathbf{l}_y - \mathbf{v}_y}$$

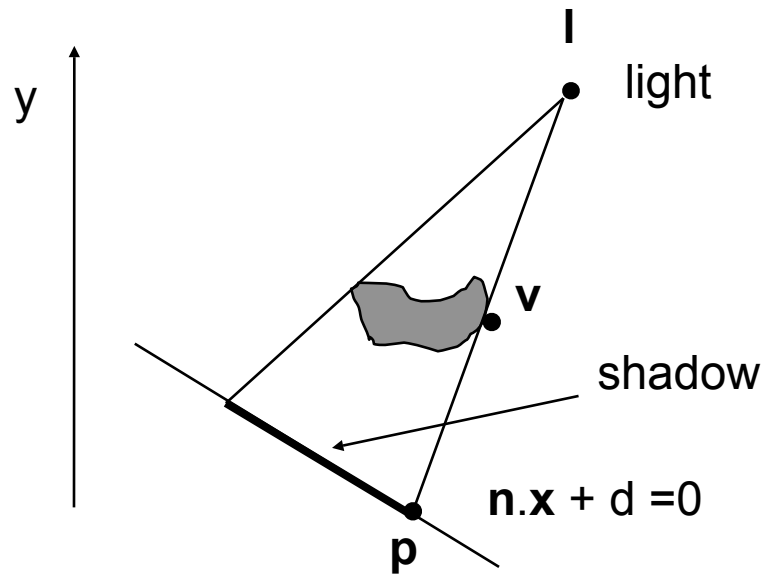
$$\mathbf{p}_x = \frac{\mathbf{l}_y \mathbf{v}_x - \mathbf{l}_x \mathbf{v}_y}{\mathbf{l}_y - \mathbf{v}_y}$$

- Same principle applied to different axis planes

Planar Shadows



- How about arbitrary plane as the shadow receiver?



Plane equation: **n**.**x** + d = 0 or

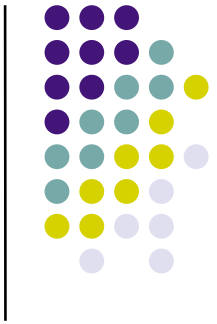
ax + by + cz + d = 0

where **n** = (a,b,c) - plane normal

Again, given light position **l** , **v**

Find **p**

Planar Shadows

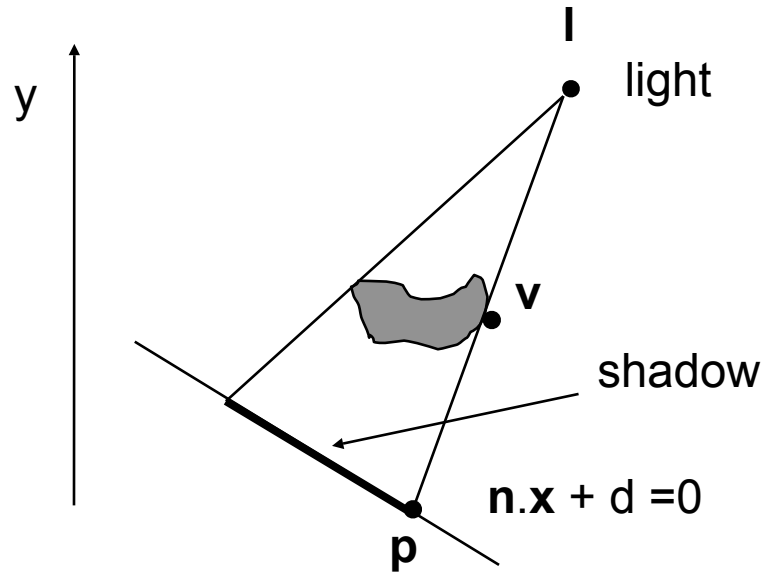


- Finding **p**

We know: **p** = **l** + (**v**-**l**) x α

Also we know: **n**.**p** + d = 0
Because **p** is on the plane

We can solve α easily:

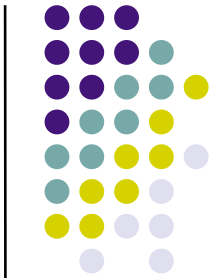


$$\mathbf{p} = \mathbf{l} - \frac{d + \mathbf{n}.\mathbf{l}}{\mathbf{n}.(\mathbf{v} - \mathbf{l})} (\mathbf{v} - \mathbf{l})$$

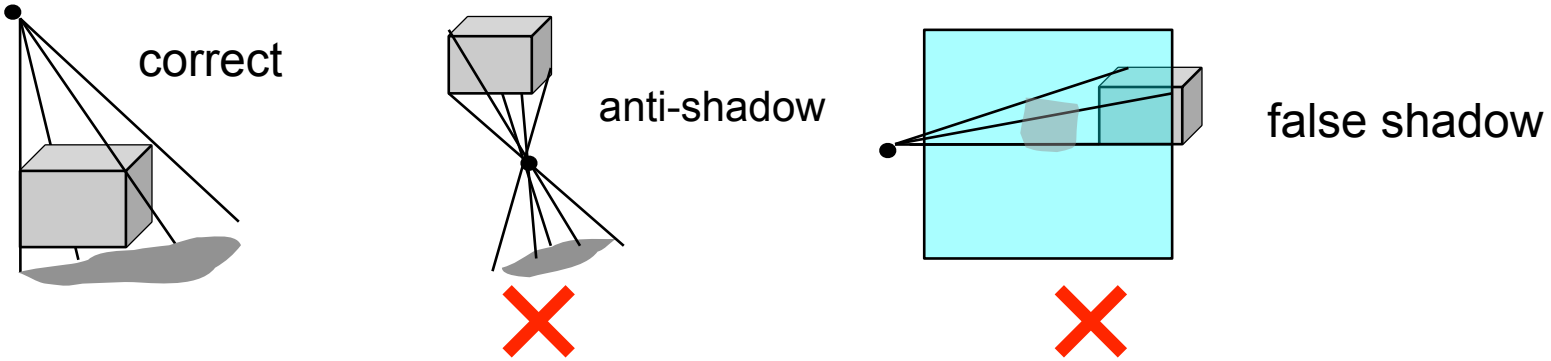
Issues about planar shadows

- Shadow polygon generation (z fighting)
  - Add an offset to the shadow polygons (glPolygonOffset)
  - Draw receivers first, turn z-test off, then draw the shadow polygons. After this, draw the rest of the scene.
- Shadow polygons fall outside the receiver
  - Using stencil buffer – draw receiver and update the stencil buffer
- Shadows have to be rendered at each frame
  - Render into texture
- Restrictive to planar objects

Issues about planar shadows

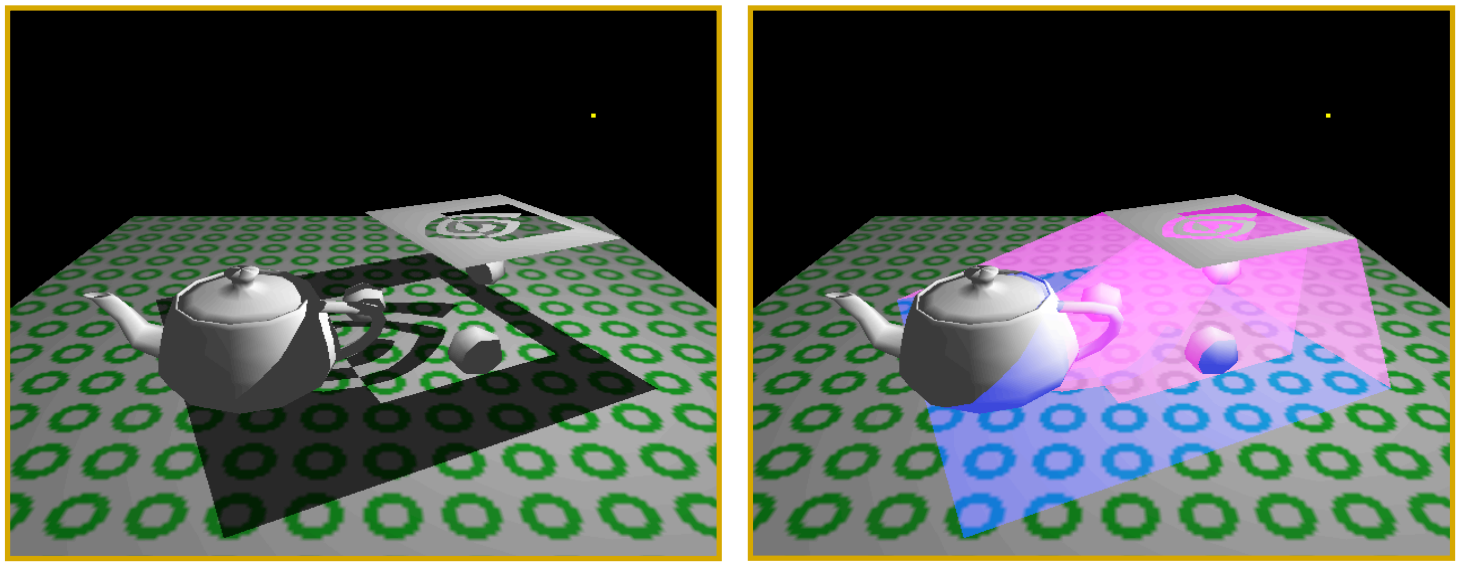


- Anti-shadows and false shadows



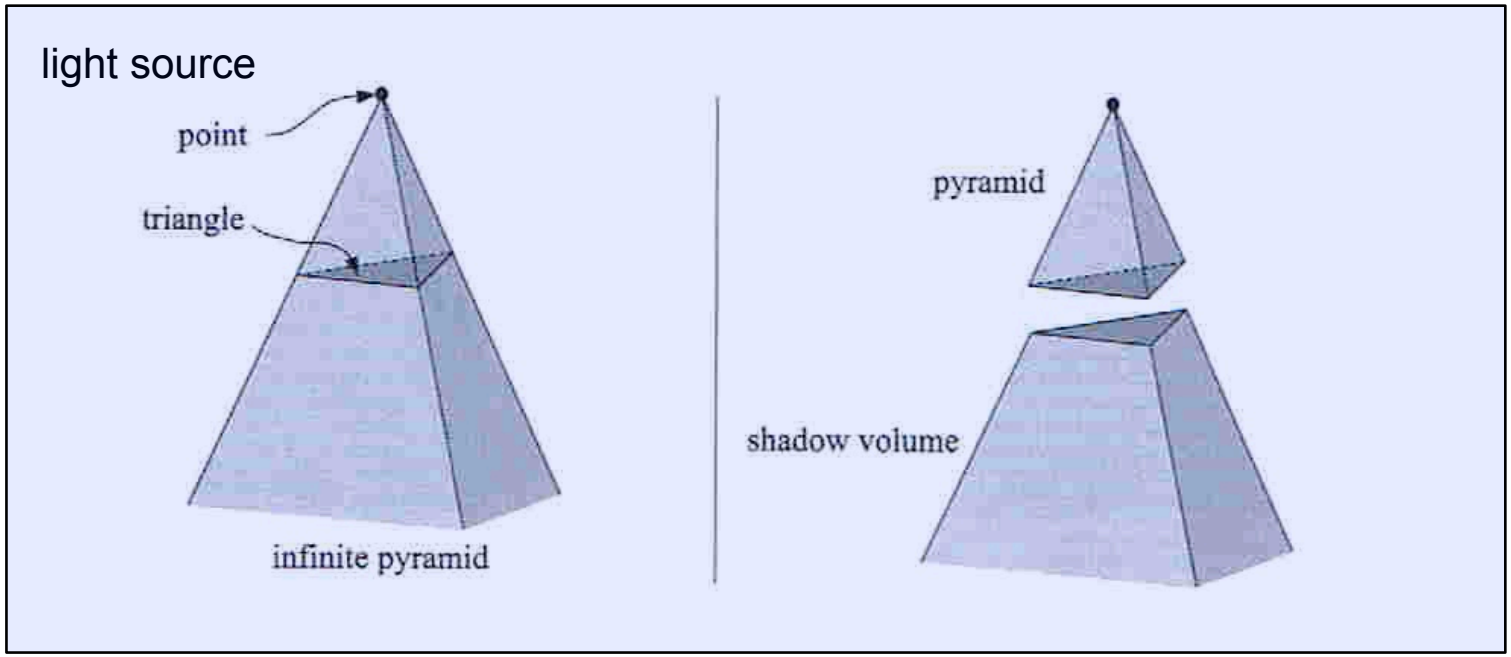
Shadow Volumes

- A more general approach for receivers that have arbitrary shapes

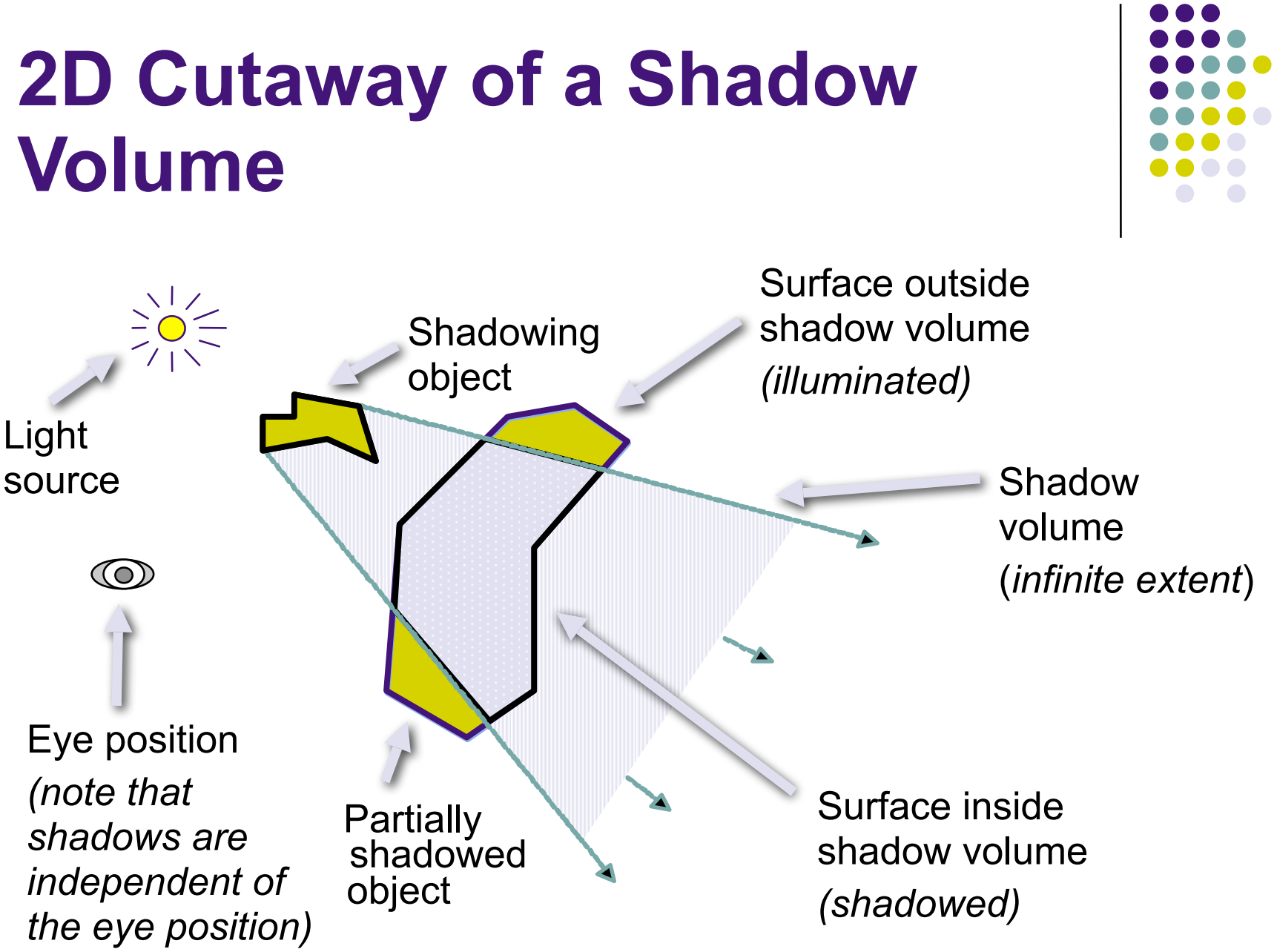


What is shadow volume?

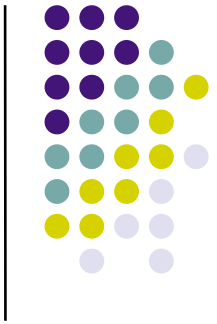
- A volume of space formed by an occluder
- Bounded by the edges of the occluder
- Any object inside the shadow volume is in shadow



2D Cutaway of a Shadow Volume

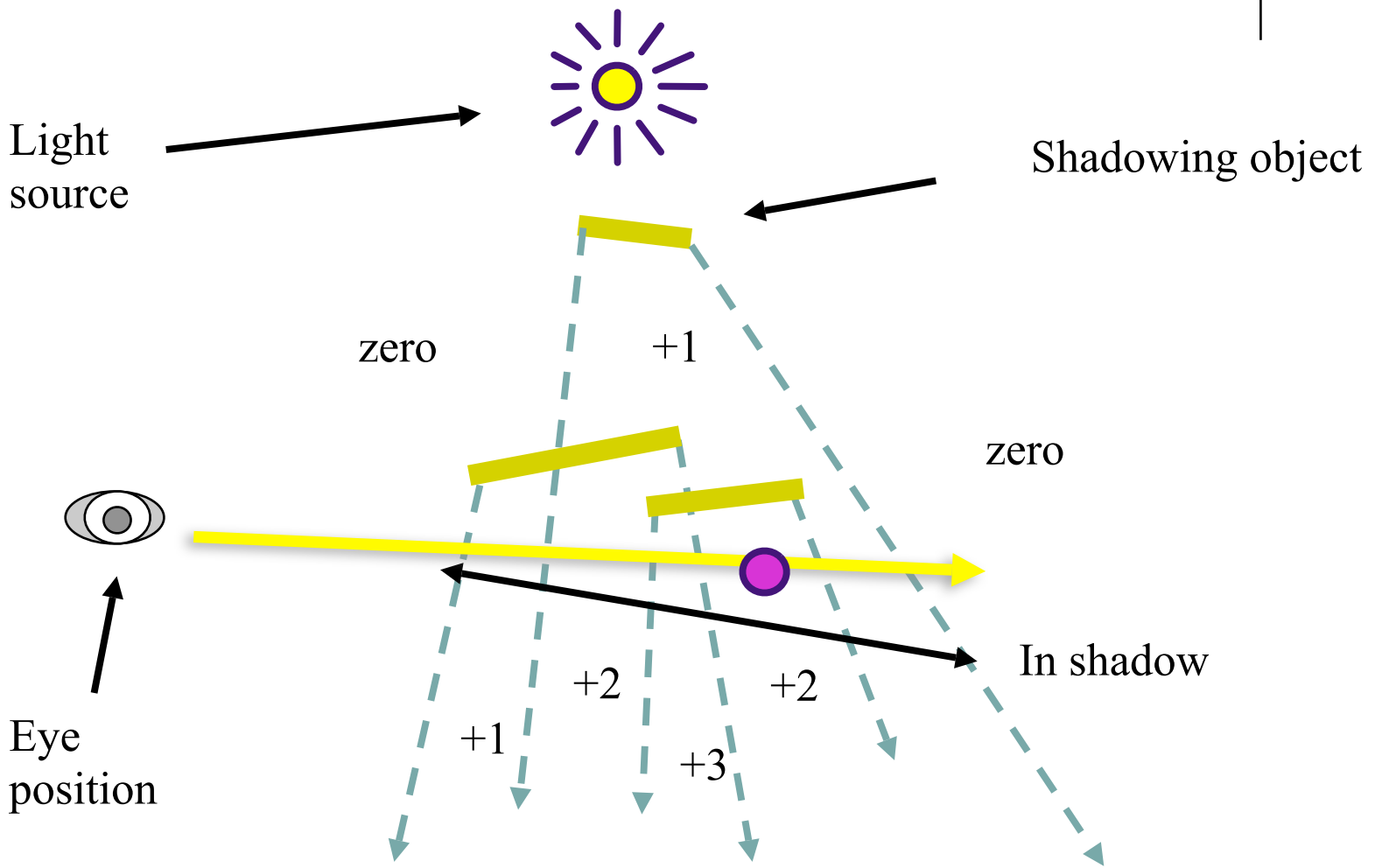


In Shadow or not?



- Use shadow volume to perform such a test
- How do we know an object is inside the shadow volume?
  1. Allocate a counter
  2. Cast a ray into the scene
  3. Increment the counter when the ray enter a front-facing polygon of the shadow volume (enter the shadow volume)
  4. Decrement the counter when the ray crosses a back-facing polygon of the shadow volume (leave the shadow volume)
  5. When we hit the object, check the counter.
     - If counter >0 ; in shadow
     - Otherwise - not in shadow

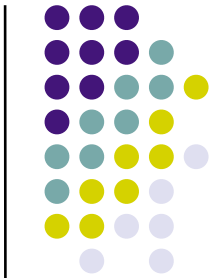
Counter for Shadow Volume



Real time shadow volume

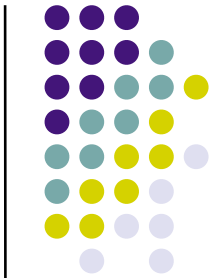
- How can we render the idea of shadow volume in real time?
  - Use OpenGL Stencil buffer as the counter
- Stencil buffer?
  - Similar to color or depth buffer, except it's meaning is controlled by application (and not visible)
  - Part of OpenGL fragment operation – after alpha test before depth test
  - Control whether a fragment is discarded or not
    - Stencil function (Stencil test) - used to decide whether to discard a fragment
    - Stencil operation – decide how the stencil buffer is updated as the result of the test

Stencil Function



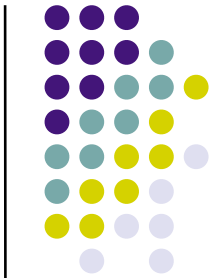
- Comparison test between reference and stencil value
  - GL_NEVER always fails
  - GL_ALWAYS always passes
  - GL_LESS passes if reference value is less than stencil buffer
  - GL_LEQUAL passes if reference value is less than or equal to stencil buffer
  - GL_EQUAL passes if reference value is equal to stencil buffer
  - GL_GEQUAL passes if reference value is greater than or equal to stencil buffer
  - GL_GREATER passes if reference value is greater than stencil buffer
  - GL_NOTEQUAL passes if reference value is not equal to stencil buffer
- **If the stencil test fails**, the fragment is discarded and the stencil operations associated with the stencil test failing is applied to the stencil value
- **If the stencil test passes**, the depth test is applied
  - **If the depth test passes**, the fragment continue through the graphics pipeline, and the stencil operation for stencil and depth test passing is applied
  - **If the depth test fails**, the stencil operation for stencil passing but depth

Stencil Operation



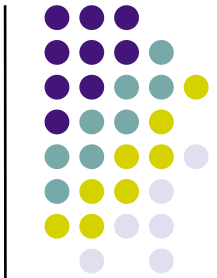
- Stencil Operation: Results of Operation on Stencil Values
  - GL_KEEP stencil value unchanged
  - GL_ZERO stencil value set to zero
  - GL_REPLACE stencil value replaced by stencil reference value
  - GL_INCR stencil value incremented
  - GL_DECR stencil value decremented GL_INVERT stencil value bitwise inverted
- Remember you can set different operations for
  - Stencil fails
  - Stencil passes, depth fails
  - Stencil passes, depth passes

OpenGL for shadow volumes



- Z-pass approach
- Z-fail approach
- Ideas used by both of the algorithms are similar
  - Z-pass: see whether the number of visible front-facing shadow volume polygons and the number of visible back-facing polygons are equal. If yes – objects are not in shadow
  - Z-fail: see whether the number of invisible back-facing shadow volume polygons and the number of invisible front-facing polygons are equal. If yes – objects are not in shadow

Z-pass approach

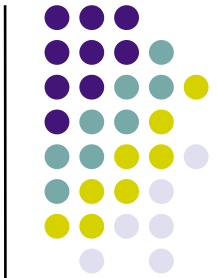


- Render visible scene with only ambient and emssion and update depth buffer
- Turn off depth and color write, turn on stencil (but still keep the depth test on)
- Init. stencil buffer
- Draw shadow volume twice using face culling
  - 1st pass: render *front* faces and *increment* stencil buffer when depth test passes
  - 2nd pass: render *back* faces and *decrement* when depth test passes
- stencil pixels != 0 in shadow, = 0 are lit
- Render the scene again with diffuse and specular when stencil pixels =0

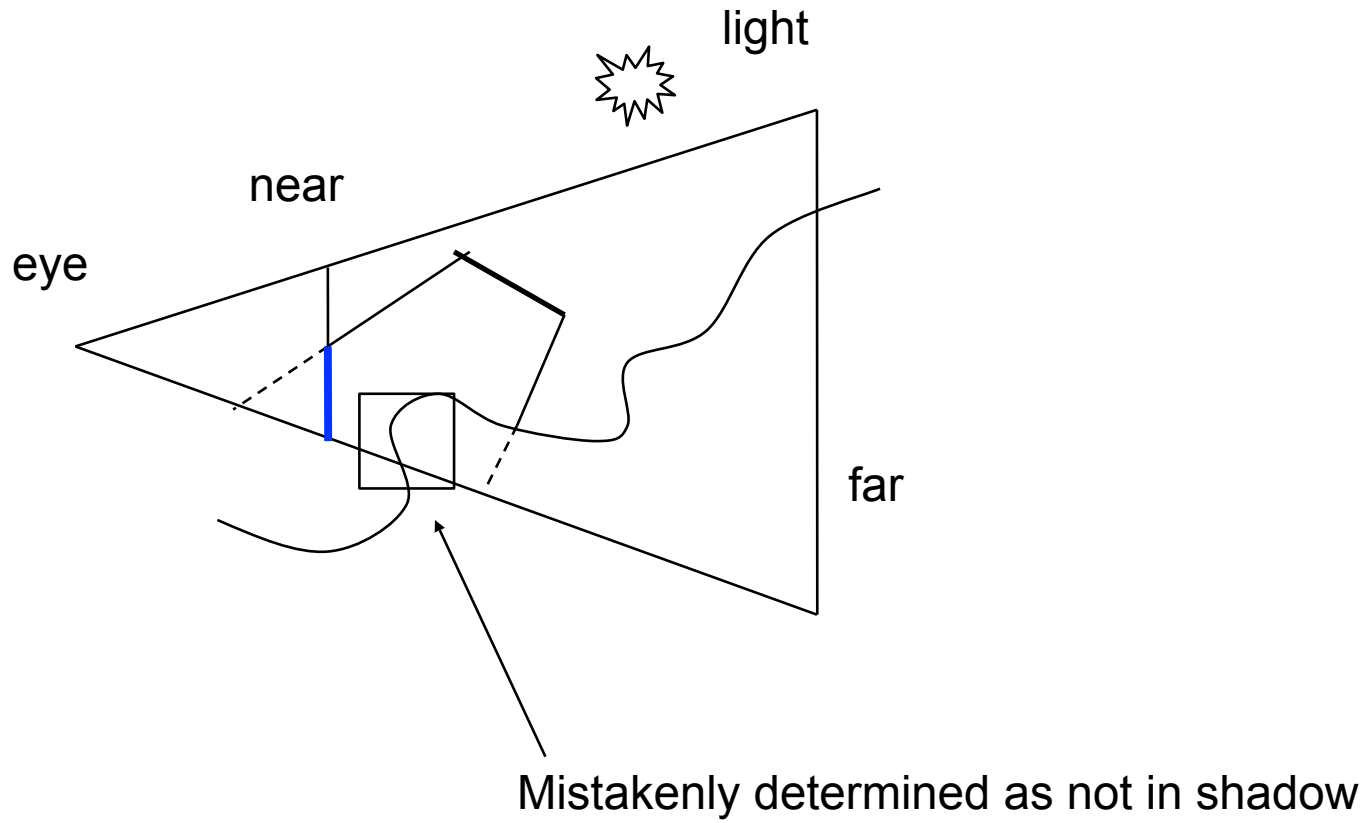
Problems of Z-pass algorithm

1. When the eye is in the shadow volume
   - Counter= 0 does not imply out of shadow anymore
   - In this case, the stencil buffer should be init. with the number of shadow volumes in which the eye is in (instead 0)
2. When the near plane intersects with the shadow volume faces (and thus will clip the faces)

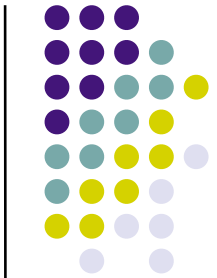
Z-pass algorithm problem



- illustration

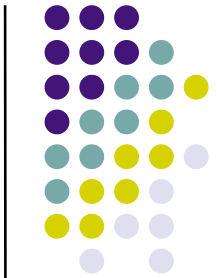


Z-fail approach

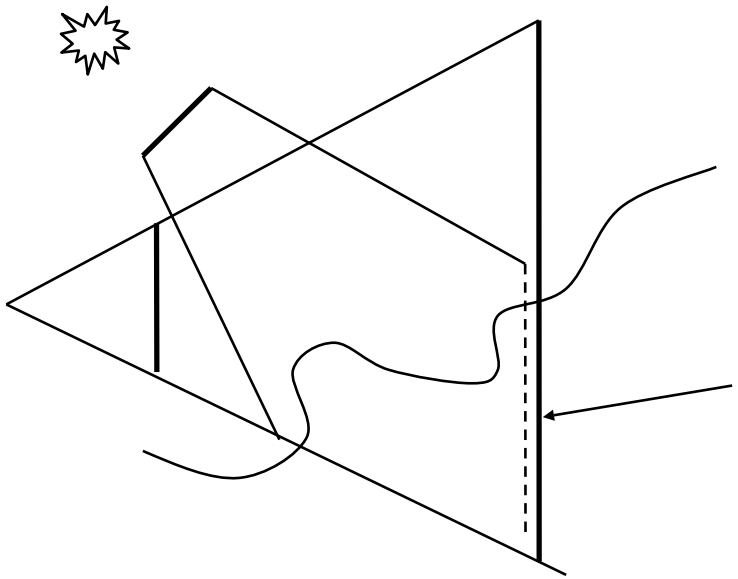


- Render visible scene to depth buffer
- Turn off depth and color, turn on stencil
- Init. stencil buffer given viewpoint
- Draw shadow volume twice using face culling
  - 1st pass: render *back* faces and *increment* when depth test fails
  - 2nd pass: render *front* faces and *decrement* when depth test fails
- stencil pixels != 0 in shadow, = 0 are lit

Problem of z-fail algorithm

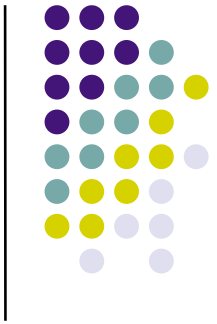


- Shadow volume can penetrate the far plane



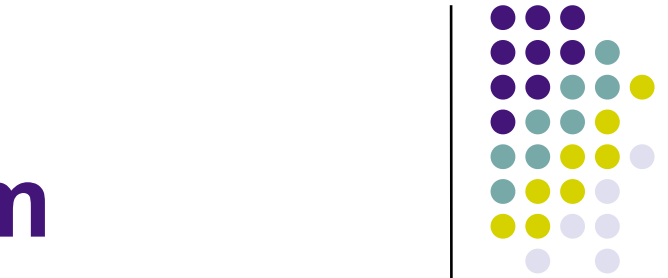
- Solution: depth clamping - close up the shadow volume

Shadow Map

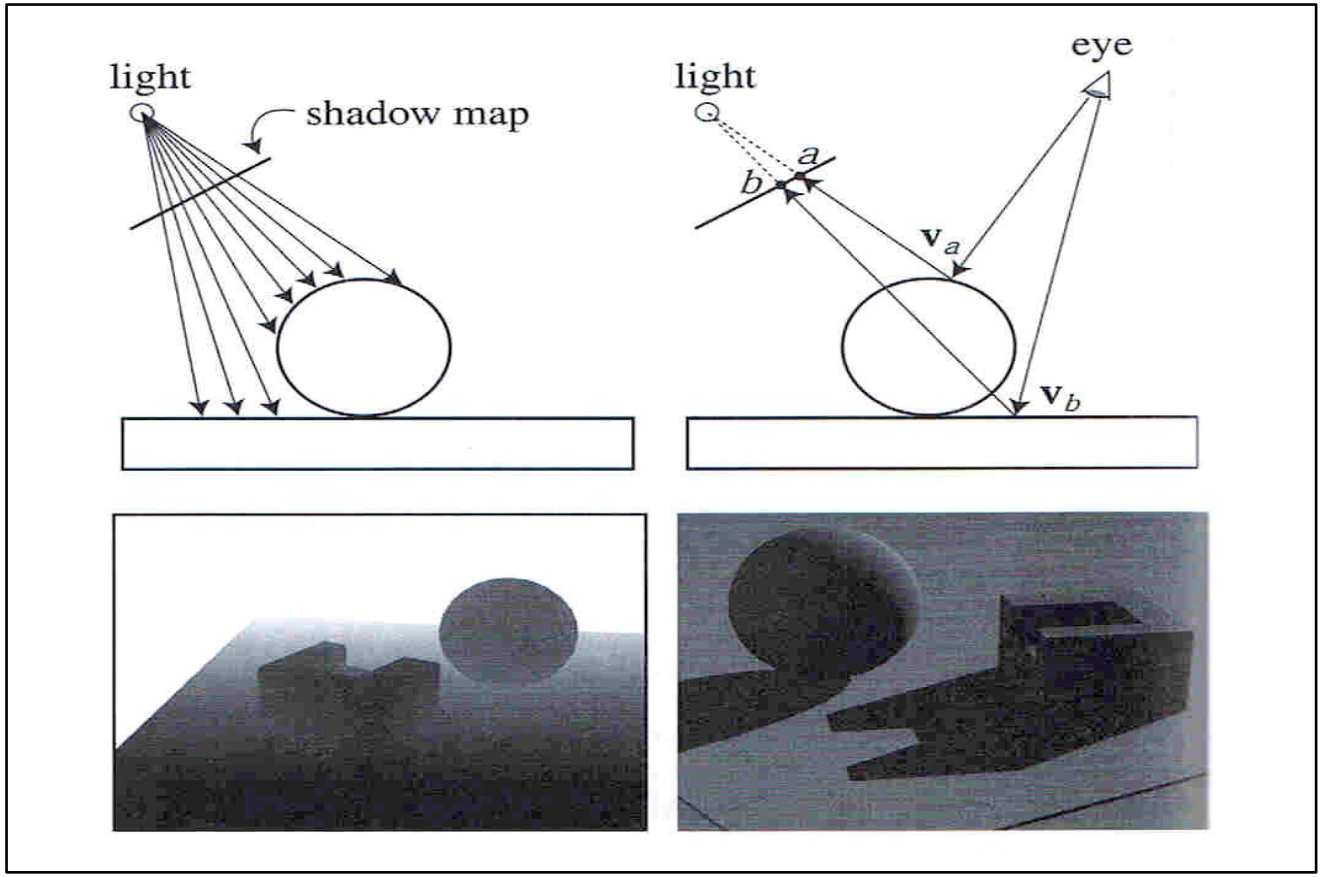


- Basic idea: objects that are not visible to the light are in shadow
- How to determine whether an object are visible to the eye?
  - Use z-buffer algorithm, but now the “eye” is light, i.e., the scene is rendered from light’s point of view
  - This particular z-buffer for the eye is called *shadow map*

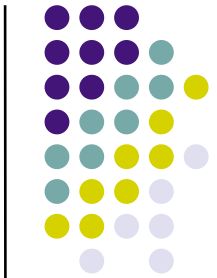
Shadow Map Algorithm



- illustration

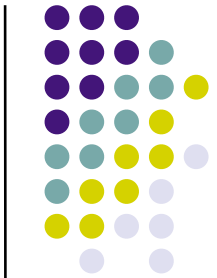


Shadow Map Algorithm



1. Render the scene using the light as the camera and perform z-buffering
2. Generate a light z buffer (called shadow map)
3. Render the scene using the regular camera, perform z-buffering, and run the following steps: (next slide)

Shadow Map Algorithm (cont'd)

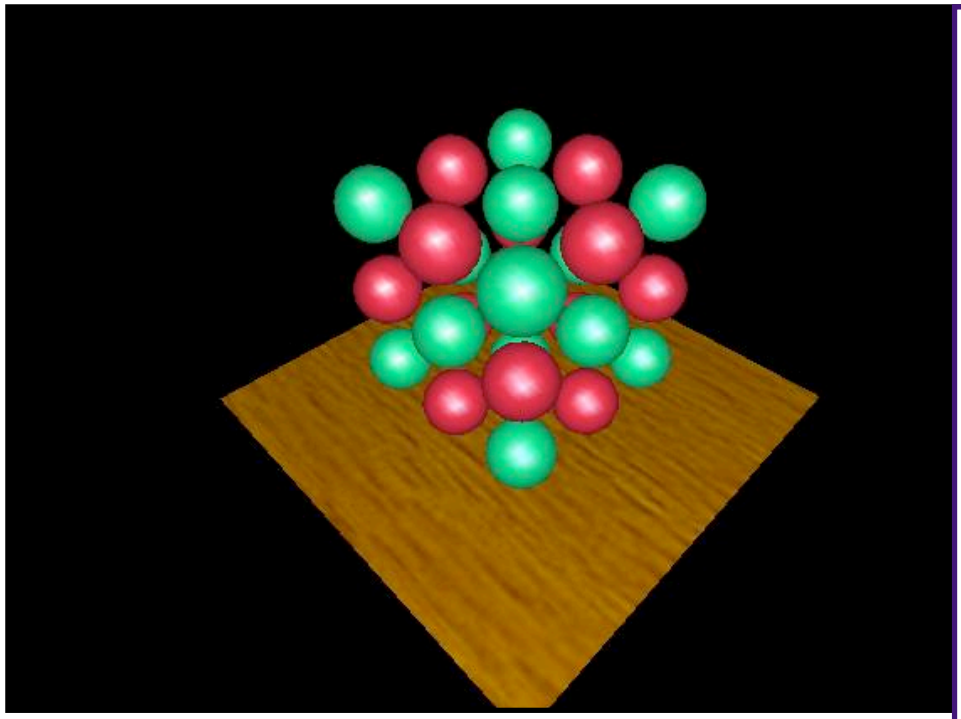


3.1 For each visible pixel with [x,y,z] in world space, perform a transformation to the light space (light as the eye) [x1,y1,z1]

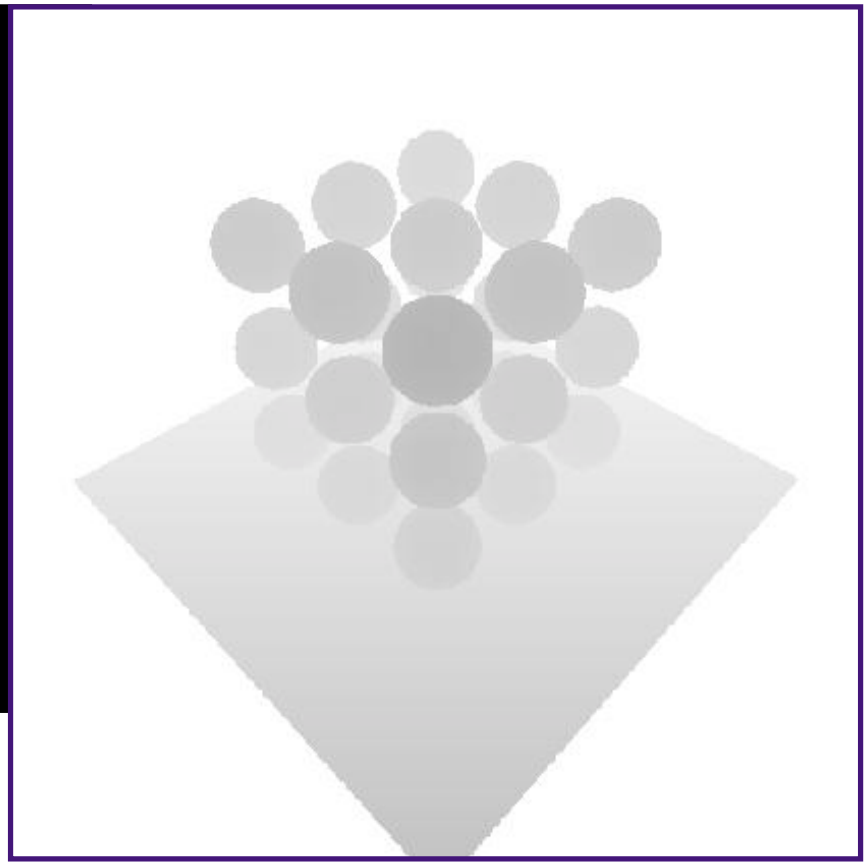
3.2 Compare z1 with z = shadow_map[x1,y1]

- If z1 <=z (closer to light), then the pixel in question is not in shadow; otherwise the pixel is shadowed

1st Pass

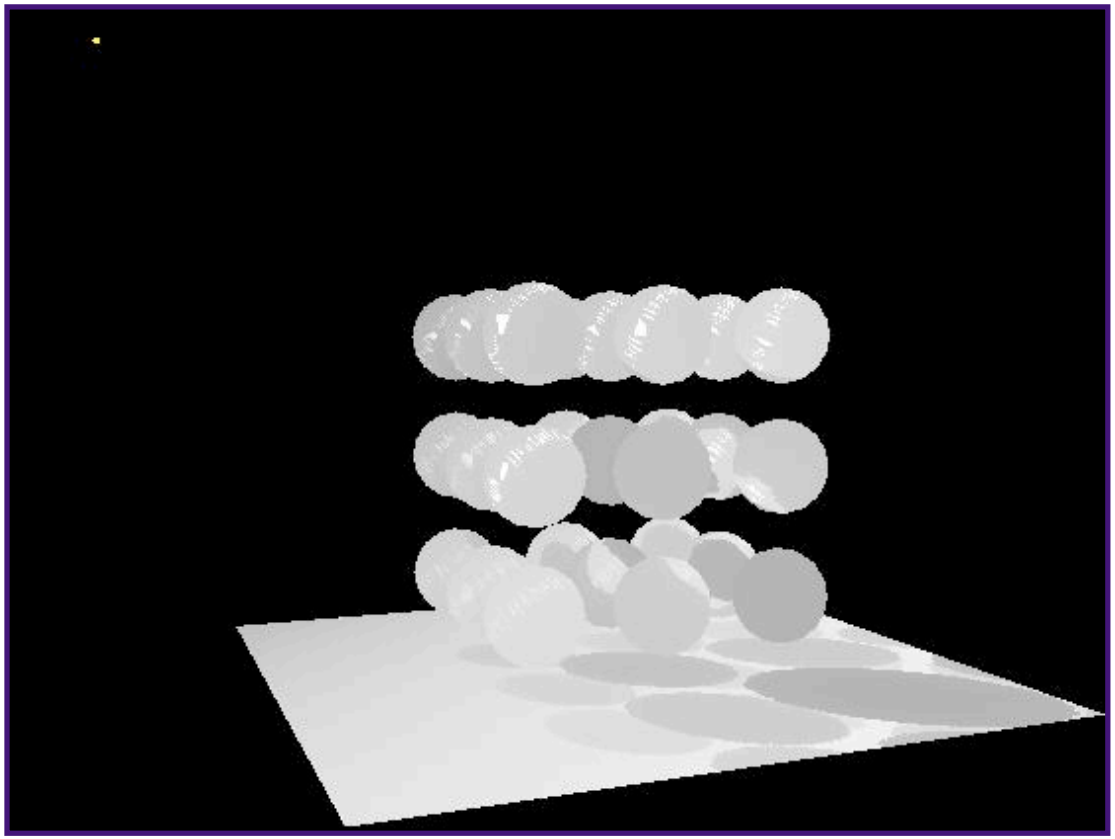


View from light



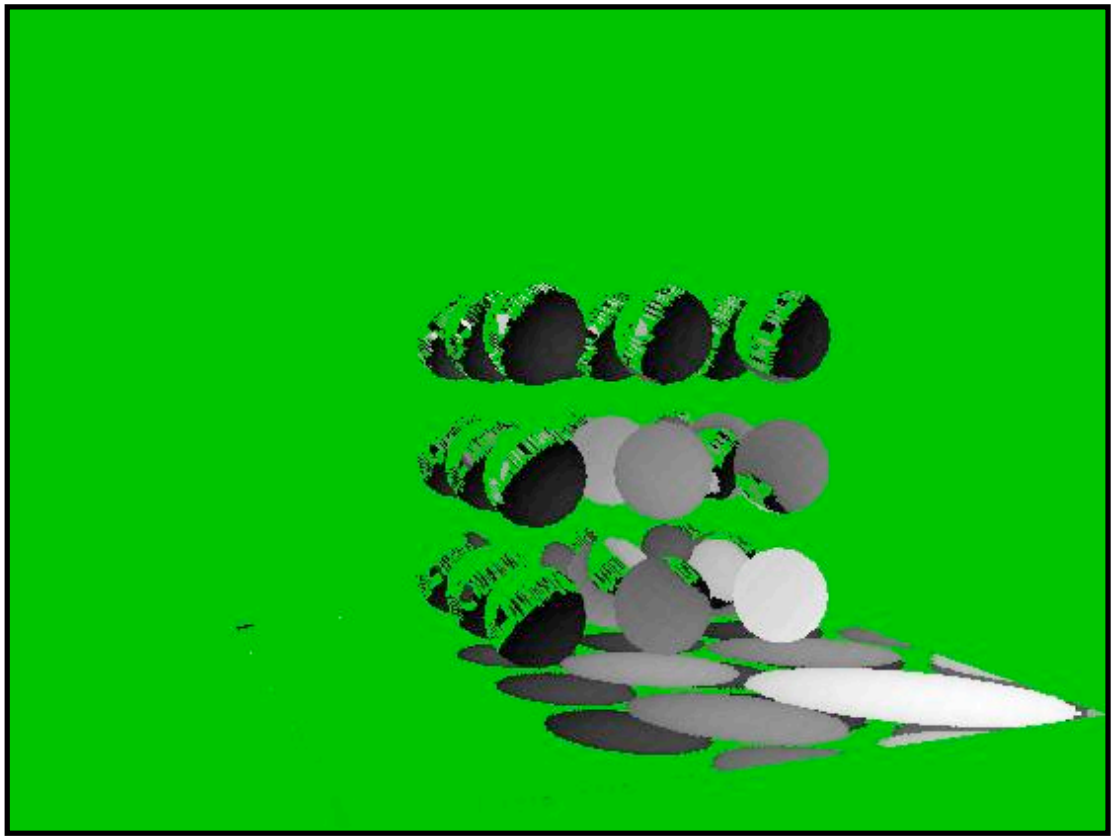
Depth Buffer (shadow map)

2nd Pass

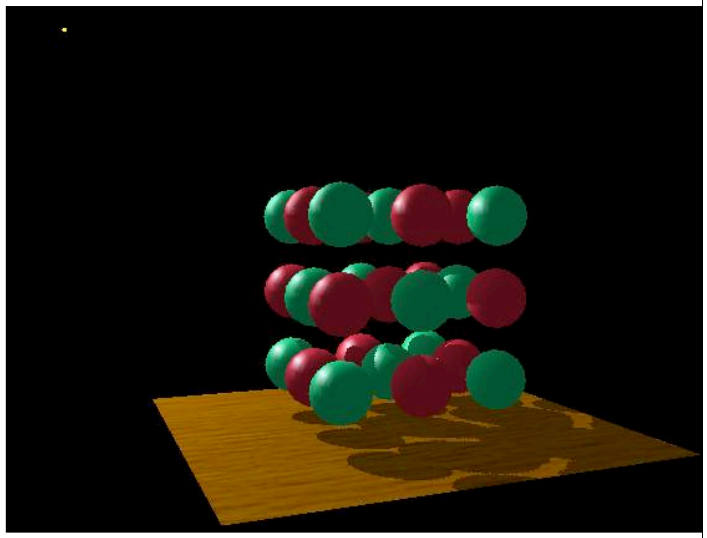


Visible surface depth

2nd Pass

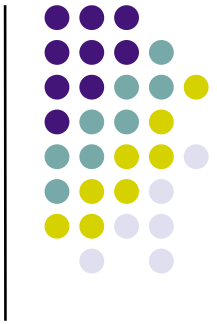


Non-green in shadow



Final Image

Shadow map issues

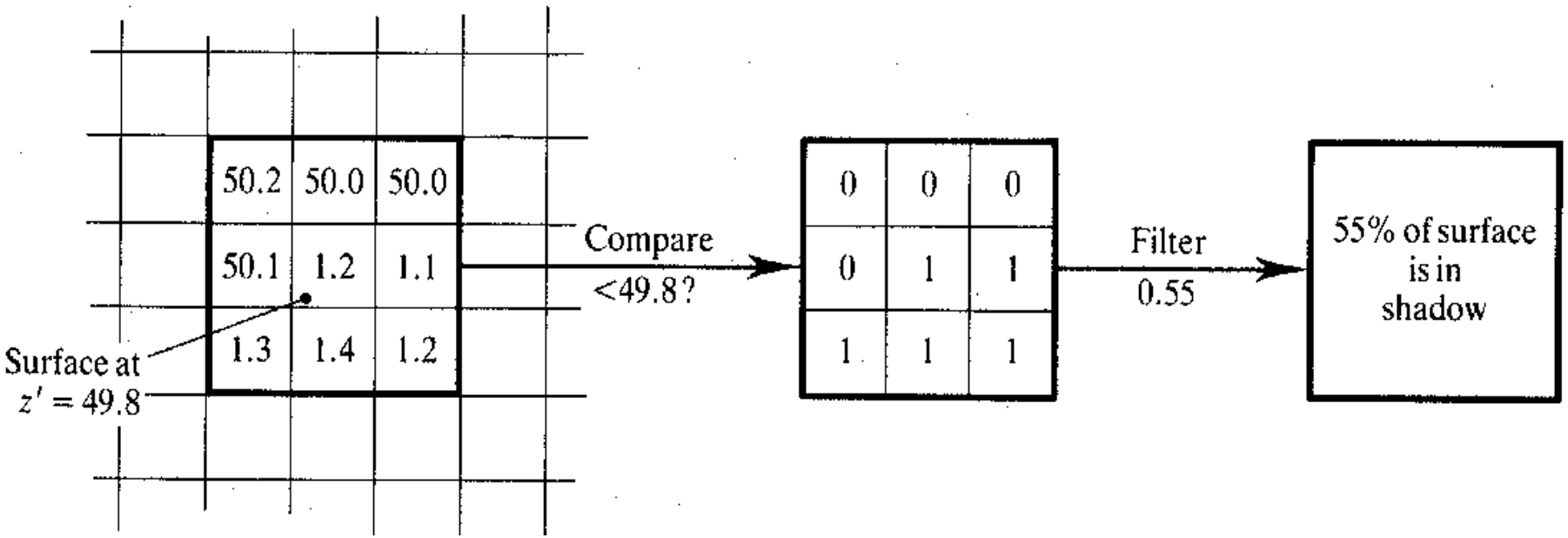


- Shadow quality depends on
  - Shadow map resolution – aliasing problem
  - Z resolution – the shadow map is often stored in one channel of texture, which typically has only 8 bits
    - Some hardware has dedicated shadow map support, such as Xbox and GeForce3
  - Self-shadow aliasing – caused by different sample positions in the shadow map and the screen

Shadow map aliasing problem

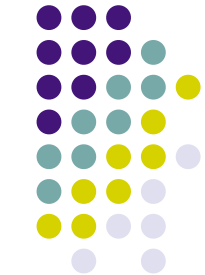
- The shadow looks blocky – when one single shadow map pixel covers several screen pixels
- This is a similar problem to texture magnification
  - Where bi-linear interpolation or nearest neighbor are used in OpenGL

Percentage Closer Filtering



- Could average binary results of all depth map pixels covered
- Soft anti-aliased shadows

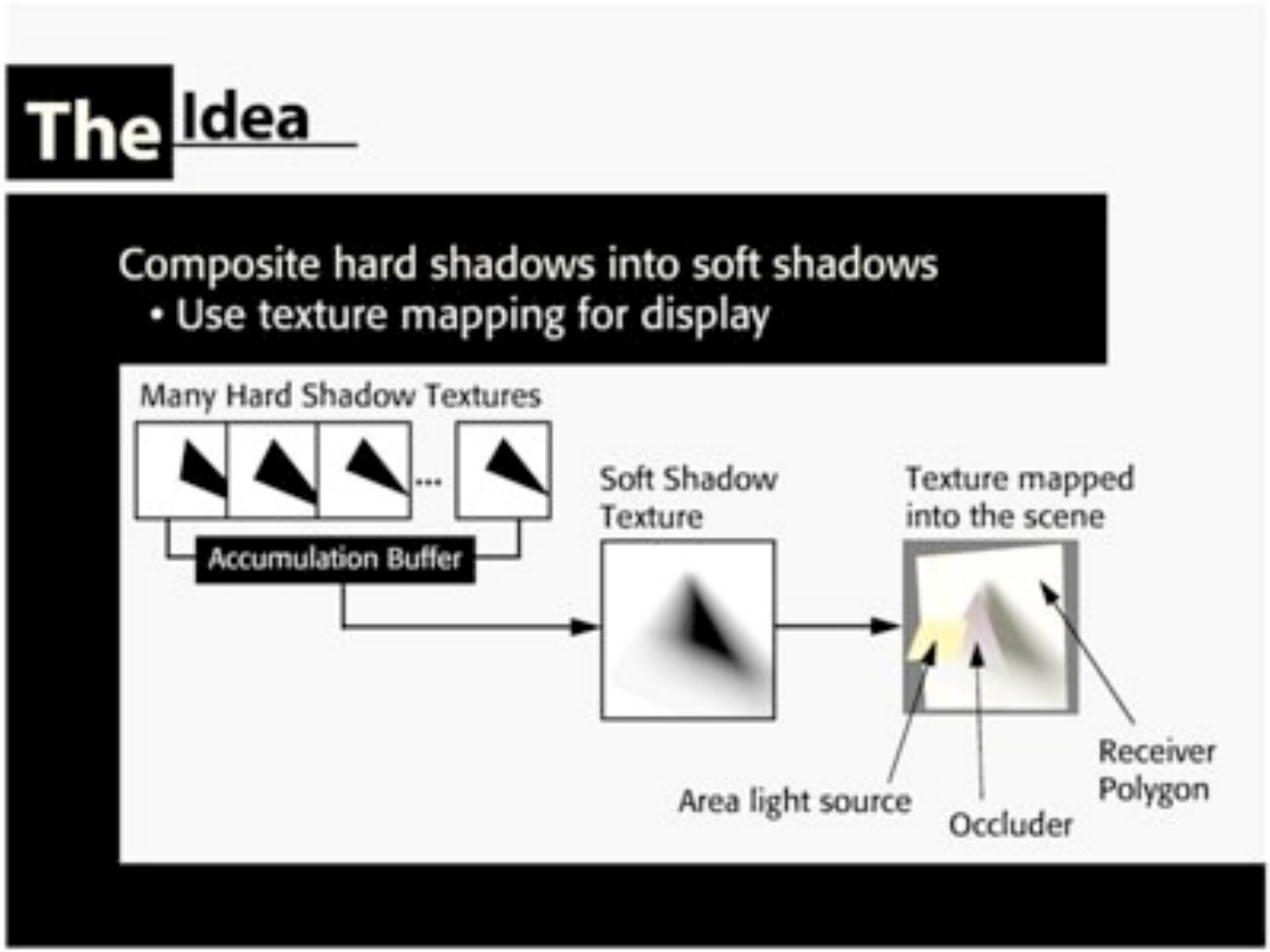
Shadow Maps With Graphics Hardware



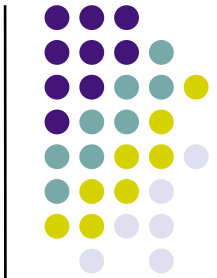
- Render scene using the light as a camera
- Read depth buffer out and copy into a 2D texture.
  - We now have a depth texture.
- Fragment's position in light space can be generated using eye-linear texture coordinate generation
  - specifically OpenGL's GL_EYE_LINEAR texgen
  - Using light-space (x, y, z, w) to generate homogenous (s, t, r, q) texture coordinates
  - Transform the texture coodrinates

Real time soft shadowing

- Only area light source can generate soft shadows
- We can sample the area light source at different points and then average the results
- Many research done
- Example: Herf and Heckbert's algorithm


The Idea
Composite hard shadows into soft shadows
• Use texture mapping for display
Many Hard Shadow Textures
Accumulation Buffer
Soft Shadow Texture
Texture mapped into the scene
Area light source
Occluder
Receiver Polygon

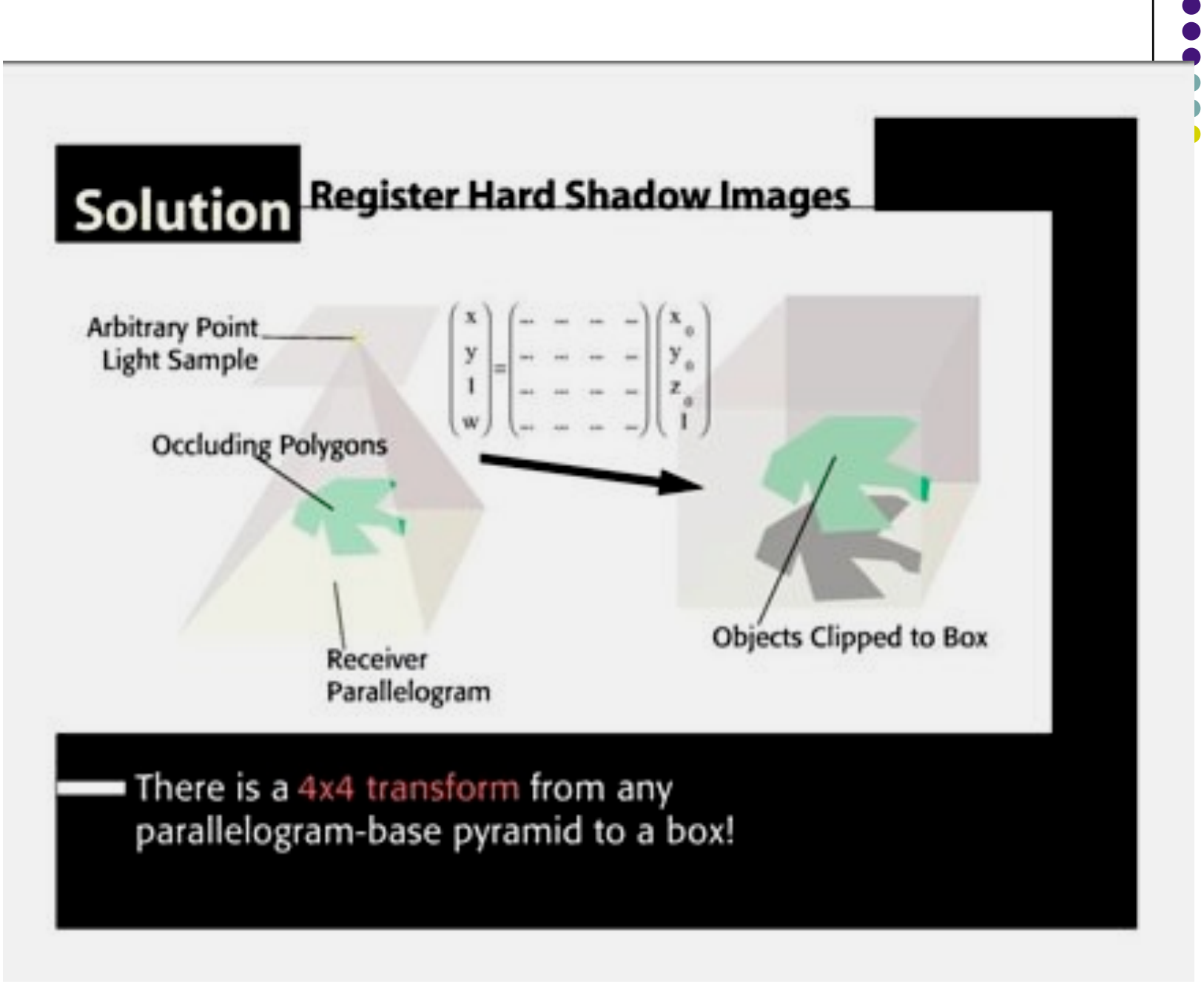
Herf and Heckbert's Basic Ideda



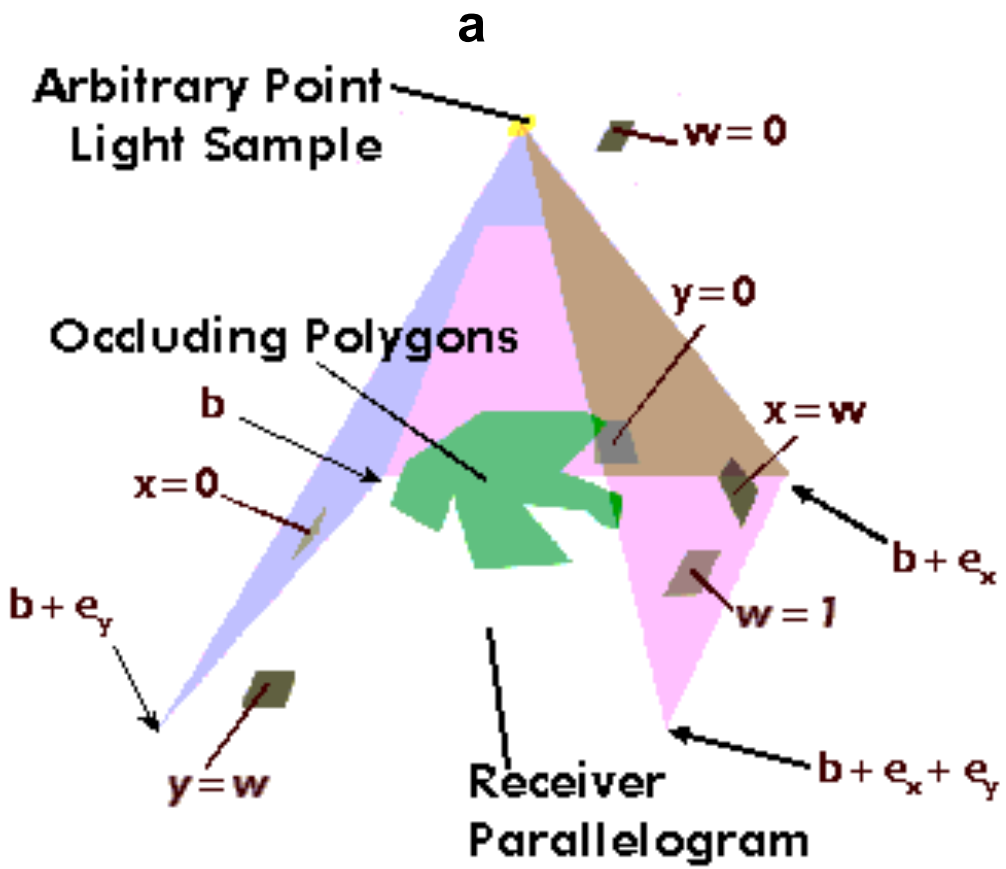
- Sample multiple points from the area light
- For each shadow receiver
  - For each point light source
    - Draw the receiver, shaded
    - Project the environment onto the shadow receiver (draw in black)
    - Draw the projection result into the OpenGL accumulation buffer
  - Save the accumulation buffer content into a texture
  - Draw the shadow receiver using the texture

Project onto the shadow receiver

- Transform the pyramid formed by the light and the receiver into a parallelepiped
- The parallelepiped has x, y ranged from (0,0) to (1,1) and z from 1 to infinity
- Essentially the light is viewing the receiver, and a perspective projection is performed
- Use OpenGL to render the environment between (0,0,1) and (1,1,infinity) in black


Solution
Register Hard Shadow Images
Arbitrary Point
Light Sample
Occluding Polygons
Receiver
Parallelogram
Objects Clipped to Box
There is a 4x4 transform from any
parallelogram-base pyramid to a box!

The projection setup



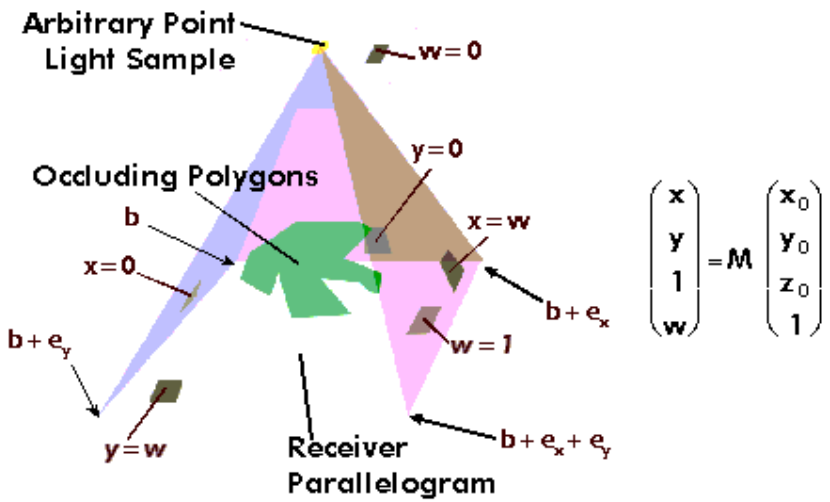
$$\begin{pmatrix} x \\ y \\ 1 \\ w \end{pmatrix} = M \begin{pmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{pmatrix}$$

The Projection Matrix

$$\mathbf{M} = \begin{pmatrix} \alpha_x n_{xx} & \alpha_x n_{xy} & \alpha_x n_{xz} & -\alpha_x \mathbf{n}_x \cdot \mathbf{b} \\ \alpha_y n_{yx} & \alpha_y n_{yy} & \alpha_y n_{yz} & -\alpha_y \mathbf{n}_y \cdot \mathbf{b} \\ 0 & 0 & 0 & 1 \\ \alpha_w n_{wx} & \alpha_w n_{wy} & \alpha_w n_{wz} & -\alpha_w \mathbf{n}_w \cdot \mathbf{a} \end{pmatrix}$$

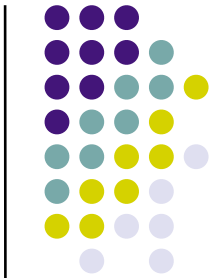
where

$$\mathbf{n}_x = \mathbf{e}_w \times \mathbf{e}_y \qquad \alpha_x = 1/\mathbf{n}_x \cdot \mathbf{e}_x$$
$$\mathbf{n}_y = \mathbf{e}_x \times \mathbf{e}_w \qquad \alpha_y = 1/\mathbf{n}_y \cdot \mathbf{e}_y$$
$$\mathbf{n}_w = \mathbf{e}_y \times \mathbf{e}_x \qquad \alpha_w = 1/\mathbf{n}_w \cdot \mathbf{e}_w$$



Remember **a** is the point light source, **b** is one corner of the receiver parallelogram
And **ex ey** are the two vectors for the parallelogram

Orthographic rendering



```
make_projmatrix(M);  // see the previous slide
// OpenGL stuff
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(0.0, 1.0, 0.0, 1.0, 0.999, 1000);
glMultMatrixf(M);
glMatrixMode(GL_MODELVIEW);
   …// render the scene
```