Full-time wrapup

Advanced Computer Graphics Seminar Course

Spring, sp3+4, 7.5p, Optional CTH-course

Email uffe@chalmers.se

- Seminars
  - 2 hour each week or
  - 2 hours every 2:nd week
- A smaller project of your choice. E.g.:
  - Realistic explosions, clouds or fractal mountains
  - CUDA – program (general parallel problem)
  - Medical (volume) visualization
  - Game
  - Modeling animated scene with physics using Maya / Blender/3DSMax etc

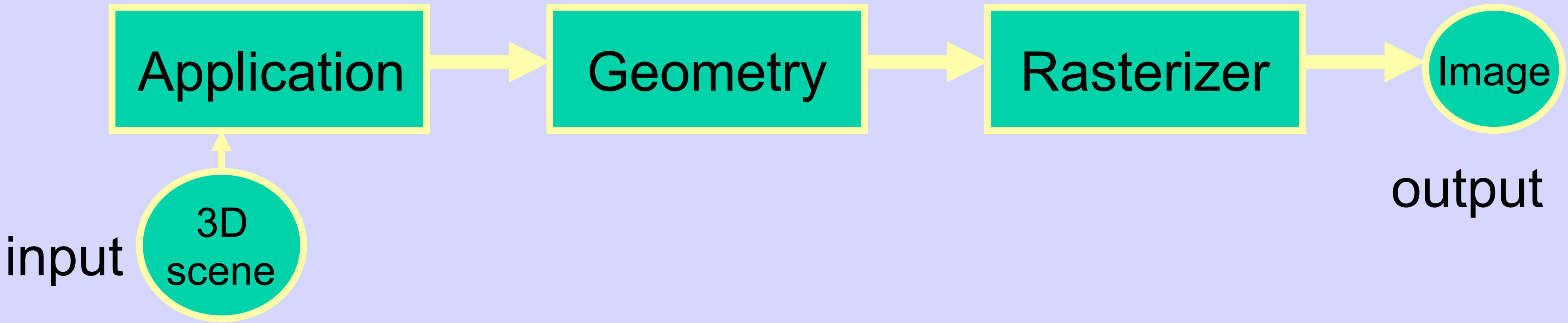
Exam / Tenta

- 16 Dec 2011, 14:00-18:00, “Väg och vatten”-salar, Johanneberg, Duration: 4 Hours,
- Re-exam:11 Apr 2012, 14:00-18:00, “Väg och vatten”, Johanneberg, Duration: 4 Hours

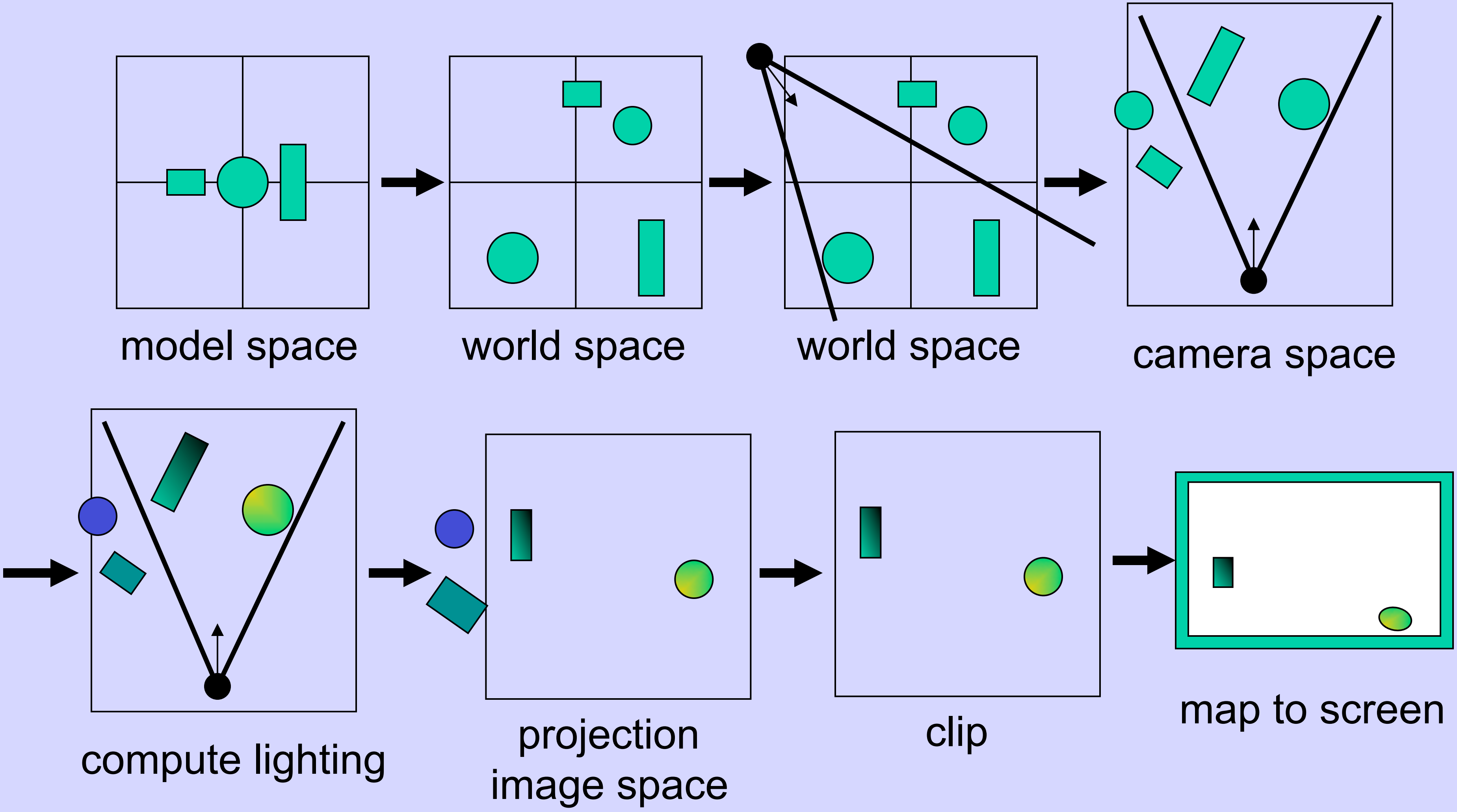
- This lecture – repetition and what I think is most important.
  - Repetition of all lectures – good chance for you to ask about anything you wonder
  - Ask anything you want about the exam

Lecture 1: Real-time Rendering
The Graphics Rendering Pipeline

- Three conceptual stages of the pipeline:
  - Application (executed on the CPU)
    - collision detection, speed-up techniques, animation
  - Geometry
    - Executing vertex and geometry shader
    - Vertex shader:
      - lighting computations per triangle vertex
      - Project onto screen (3D to 2D)
  - Rasterizer
    - Executing fragment shader
    - Interpolation of parameters (colors, texcoords etc) over triangle
    - Z-buffering, fragment merge (i.e., blending), stencil tests…



GEOMETRY – transformation summary



Lecture 2: Transforms

- Scaling, rotations, translations
- Cannot use same matrix to transform normals

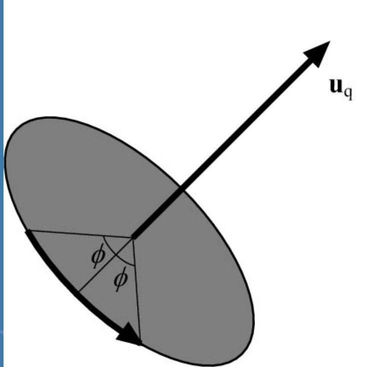
$$\text{Use: } \mathbf{N} = \left(\mathbf{M}^{-1}\right)^{T} \quad \text{instead of } \mathbf{M}$$

Identical if rigid-body transform

- Homogeneous notation
- Projections
- Quaternions $\hat{\mathbf{q}} = (\sin\phi\,\mathbf{u}_q, \cos\phi)$
  - Know what they are good for. Not knowing the mathematical rules.

$$\hat{\mathbf{q}}\hat{\mathbf{p}}\hat{\mathbf{q}}^{-1}$$

- …represents a rotation of 2ϕ radians around axis $\mathbf{u}_q$ of point **p**



Homogeneous notation

- A point: $\mathbf{p} = \begin{pmatrix} p_x & p_y & p_z & 1 \end{pmatrix}^T$
- Translation becomes:

$$\underbrace{\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\mathbf{T}(\mathbf{t})} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x + t_x \\ p_y + t_y \\ p_z + t_z \\ 1 \end{pmatrix}$$

Rotation part

Translation part

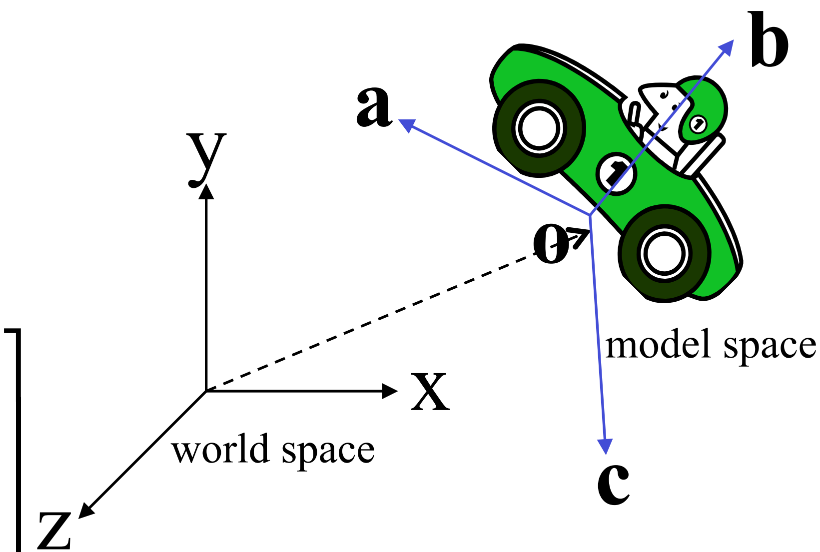
- A vector (direction): $\mathbf{d} = \begin{pmatrix} d_x & d_y & d_z & 0 \end{pmatrix}^T$
- Translation of vector: $\mathbf{Td} = \mathbf{d}$
- Also allows for projections (later)

Change of Frames

(0,5,0)

- $M_{\text{model-to-world}}$:

The basis vectors **a,b,c** are expressed in the world coordinate system



$$M_{\text{model-to-world}} = \begin{bmatrix} a_x & b_x & c_x & o_x \\ a_y & b_y & c_y & o_y \\ a_z & b_z & c_z & o_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

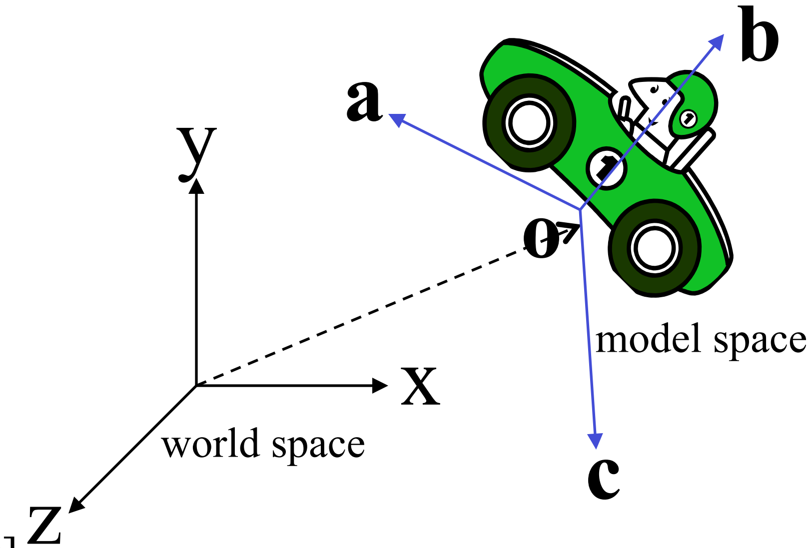
(Both coordinate systems are supposed to be right handed)

E.g.: $\mathbf{p}_{\text{world}} = M_{m\to w}\,\mathbf{p}_{\text{model}} = M_{m\to w}\,(0,5,0)^T = 5\,\mathbf{b}\;\;(+\,\mathbf{o})$

Change of Frames

$\mathbf{p}_{\text{modelspace}} = (\mathbf{p}_x, \mathbf{p}_y, \mathbf{p}_z)$

$$M_{\text{model-to-world}} = \begin{bmatrix} a_x & b_x & c_x & o_x \\ a_y & b_y & c_y & o_y \\ a_z & b_z & c_z & o_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Let's initially disregard the translation **o**. I.e., **o**=[0,0,0]

X: One step along **a** results in $\mathbf{a}_x$ steps along world space axis x.

One step along **b** results in $\mathbf{b}_x$ steps along world space axis x.

One step along **c** results in $\mathbf{c}_x$ steps along world space axis x.

The x-coord for **p** in world space is thus $[a_x\ b_x\ c_x]\mathbf{p}$.

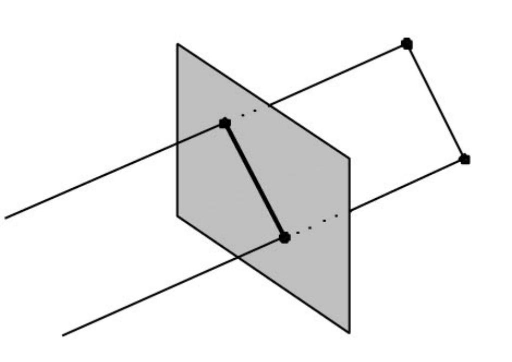
The y-coord for **p** in world space is thus $[a_y\ b_y\ c_y]\mathbf{p}$.

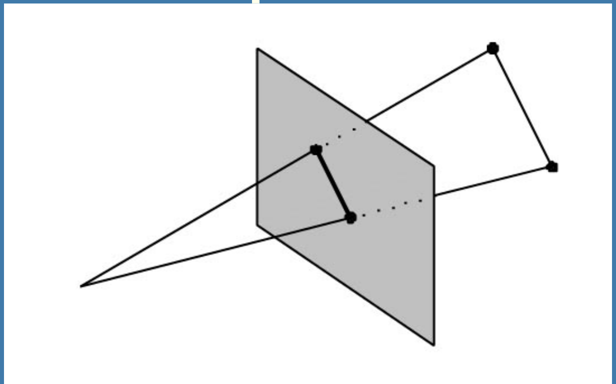
The z-coord for **p** in world space is thus $[a_z\ b_z\ c_z]\mathbf{p}$.

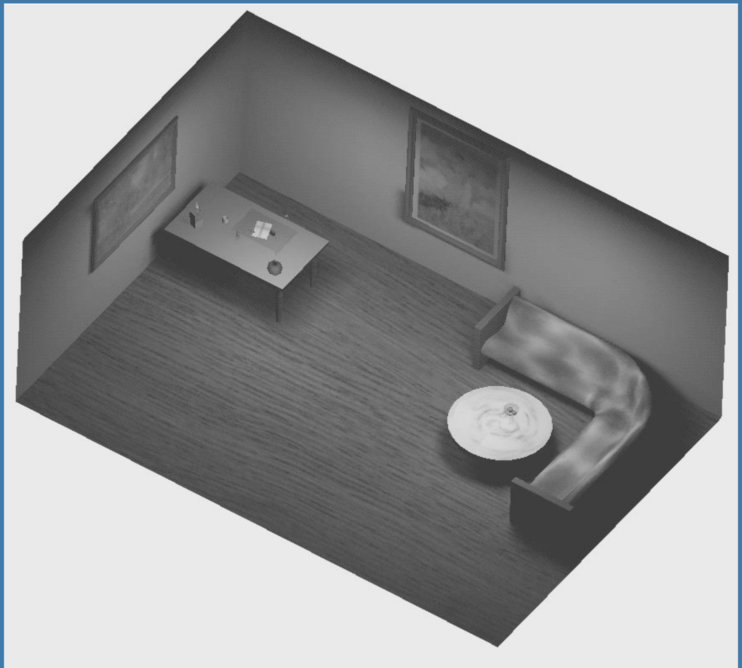
With the translation **o** we get $\mathbf{p}_{\text{worldspace}} = M_{\text{model-to-world}}\ \mathbf{p}_{\text{modelspace}}$

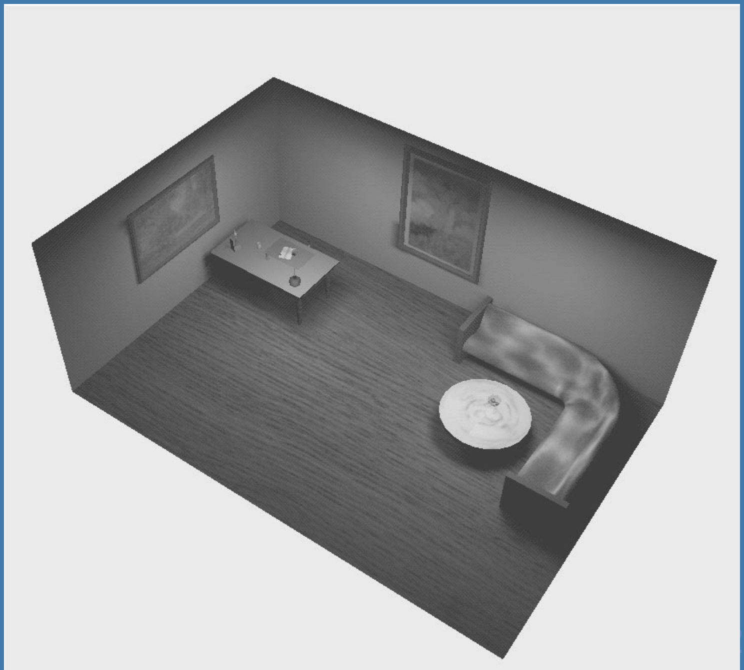
Projections

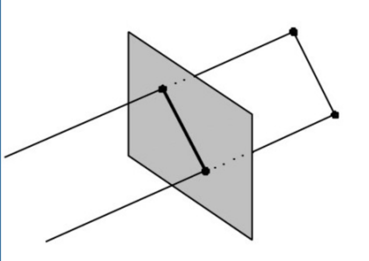
- Orthogonal (parallel) and Perspective







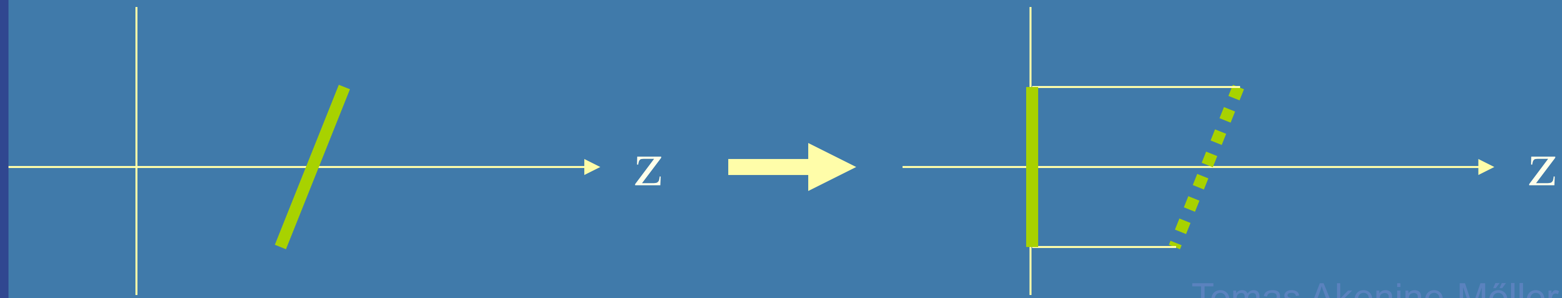




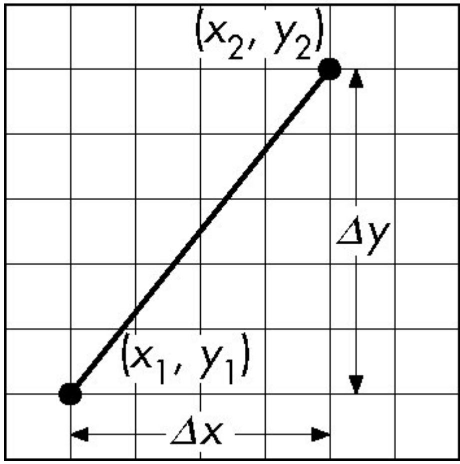
Orthogonal projection

- Simple, just skip one coordinate
  - Say, we're looking along the z-axis
  - Then drop z, and render

$$\mathbf{M}_{ortho} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \Rightarrow \mathbf{M}_{ortho} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x \\ p_y \\ 0 \\ 1 \end{pmatrix}$$



DDA Algorithm



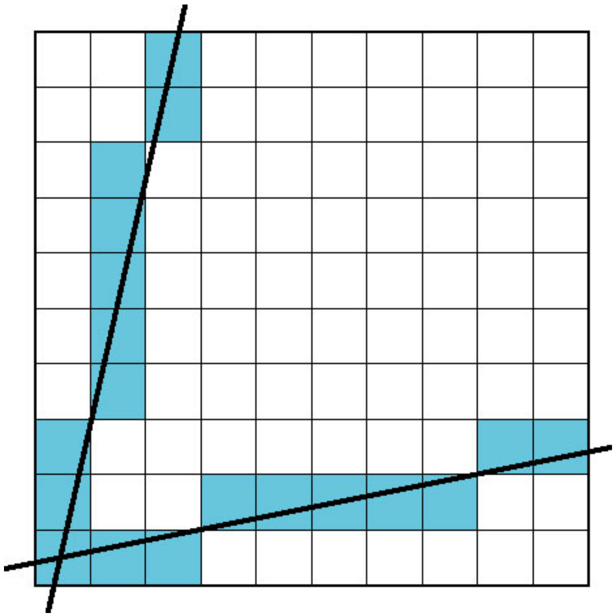
- Digital Differential Analyzer
  - DDA was a mechanical device for numerical solution of differential equations
  - Line y=kx+ m satisfies differential equation

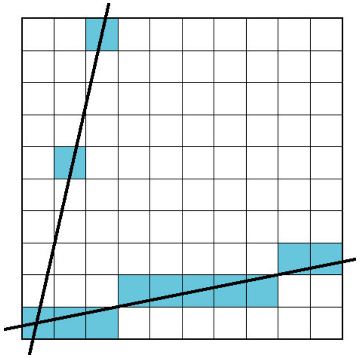
    $dy/dx = k = \Delta y/\Delta x = y_2\text{-}y_1/x_2\text{-}x_1$
- Along scan line $\Delta x = 1$

```
y=y1;
For(x=x1; x<=x2,ix++) {
  write_pixel(x, round(y), line_color)
  y+=k;
}
```

Using Symmetry

- Use for 1 ≥ k ≥ 0
- For k > 1, swap role of x and y
  - –For each y, plot closest x





Otherwise we get problem for steep slopes

02. Rasterization, Depth Sorting and Culling:

Very Important!

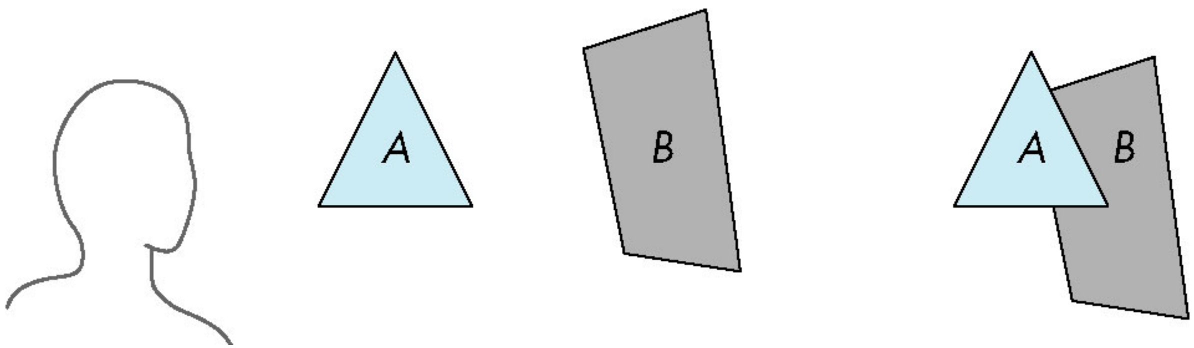
- The problem with DDA is that it uses floats which was slow in the old days
- Bresenhams algorithm only uses integers

You do not need to know Bresenham's algorithm by heart. It is enough that you **understand** it if you see it.

Said on the lecture:

Painter's Algorithm

- Render polygons a back to front order so that polygons behind others are simply painted over



B behind A as seen by viewer

Fill B then A

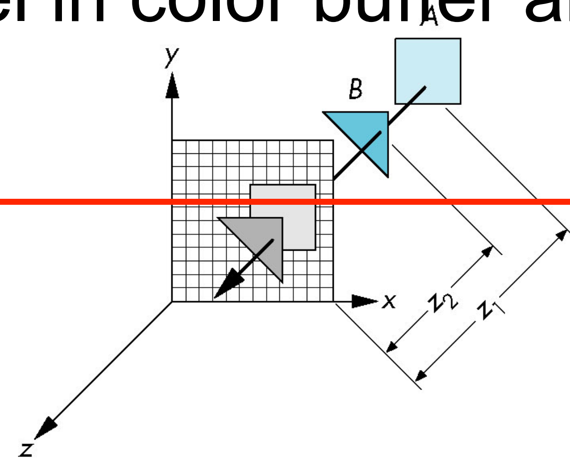
- Requires ordering of polygons first
  - O(n log n) calculation for ordering
  - Not every polygon is either in front or behind all other polygons

I.e., : Sort all triangles and render them back-to-front.

Said on the lecture:

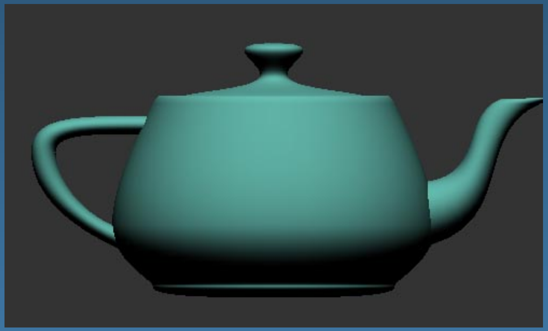
z-Buffer Algorithm

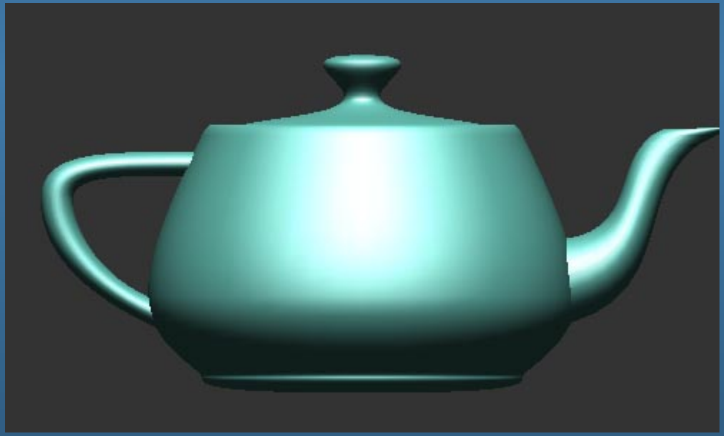
- Use a buffer called the z or depth buffer to store the depth of the closest object at each pixel found so far
- As we render each polygon, compare the depth of each pixel to depth in z buffer
- If less, place shade of pixel in color buffer and update z buffer



Lighting

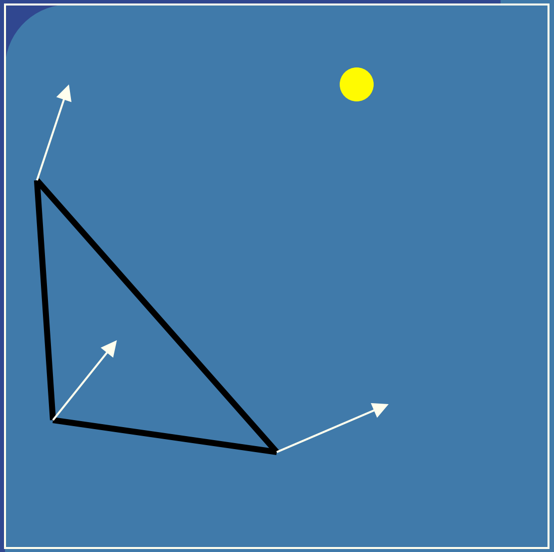
$$i = i_{amb} + i_{diff} + i_{spec} + i_{emission}$$


+

+

=


Know how to compute components.
Also, Blinns and Phongs highlight model

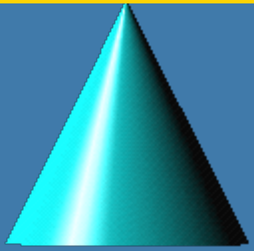
Lighting



Light:

- •Ambient (r,g,b,a)
- •Diffuse (r,g,b,a)
- •Specular (r,g,b,a)

DIFFUSE	Base color
SPECULAR	Highlight Color
AMBIENT	Low-light Color
EMISSION	Glow Color
SHININESS	Surface Smoothness



Material:

- •Ambient (r,g,b,a)
- •Diffuse (r,g,b,a)
- •Specular (r,g,b,a)
- •Emission (r,g,b,a) =”självlysande färg”

Lighting
$i = i_{amb} + i_{diff} + i_{spec} + i_{emission}$

I.e.:

$i = i_{amb} + i_{diff} + i_{spec} + i_{emission}$

$$\mathbf{i}_{amb} = \mathbf{m}_{amb} \otimes \mathbf{s}_{amb}$$

$$\mathbf{i}_{diff} = (\mathbf{n} \cdot \mathbf{l})\mathbf{m}_{diff} \otimes \mathbf{s}_{diff}$$

$$((\mathbf{n} \cdot \mathbf{l}) < 0) \Rightarrow \mathbf{i}_{spec/diff} = 0$$

Phong's reflection model:

$$\mathbf{i}_{spec} = \max(0, (\mathbf{r} \cdot \mathbf{v}))^{m_{shi}} \mathbf{m}_{spec} \otimes \mathbf{s}_{spec}$$

Blinn's reflection model:

$$\mathbf{i}_{spec} = \max(0, (\mathbf{h} \cdot \mathbf{n}))^{m_{shi}} \mathbf{m}_{spec} \otimes \mathbf{s}_{spec}$$

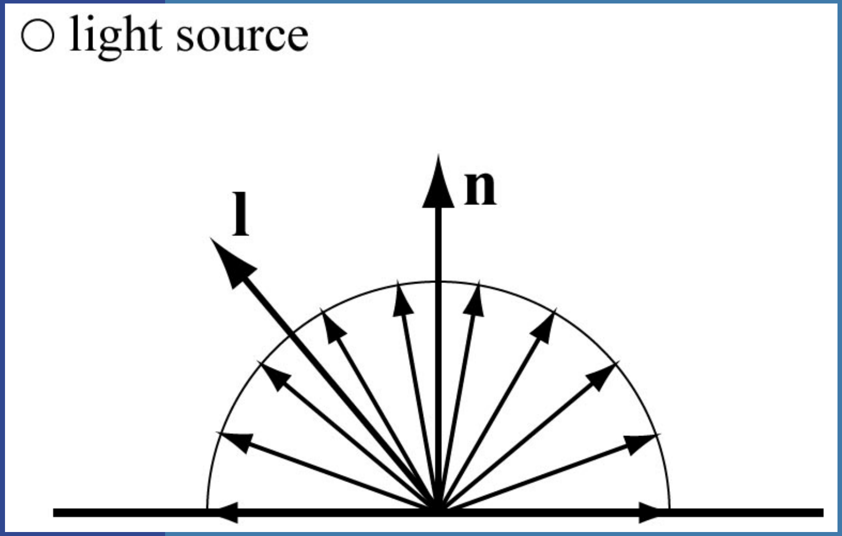
$$\mathbf{i}_{emission} = \mathbf{m}_{emission}$$

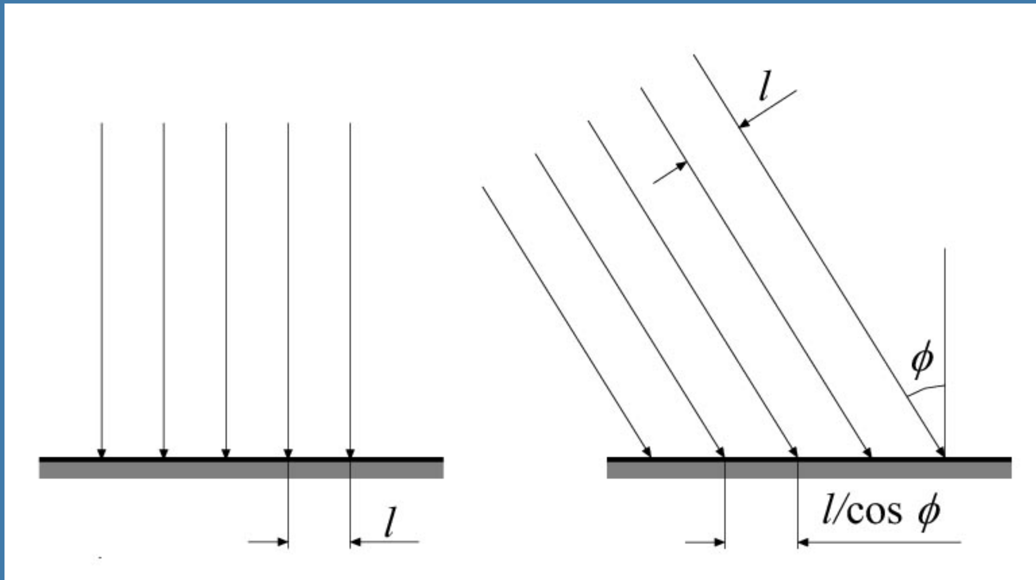
Diffuse component : $\mathbf{i}_{diff}$

- $\mathbf{i}=\mathbf{i}_{amb}+\mathbf{i}_{diff}+\mathbf{i}_{spec}+\mathbf{i}_{emission}$
- Diffuse is Lambert's law: $i_{diff} = \mathbf{n}\cdot\mathbf{l} = \cos\phi$
- Photons are scattered equally in all directions

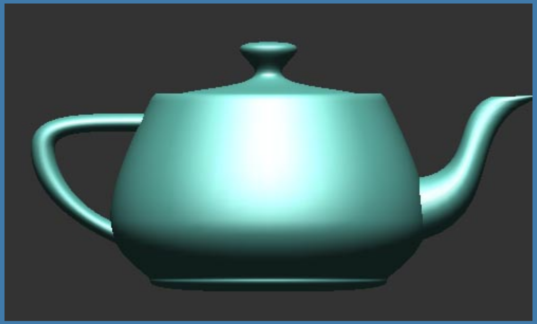
$$\mathbf{i}_{diff} = (\mathbf{n}\cdot\mathbf{l})\mathbf{m}_{diff} \otimes \mathbf{s}_{diff}$$

(Note that **n** and **l** need to be normalized)

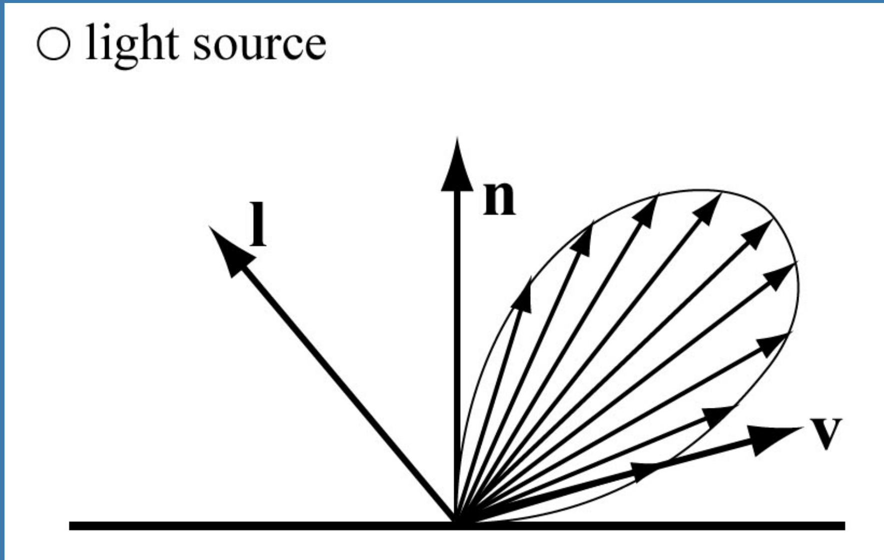




Lighting
Specular component : i_{spec}



- Diffuse is dull (left)
- Specular: simulates a highlight



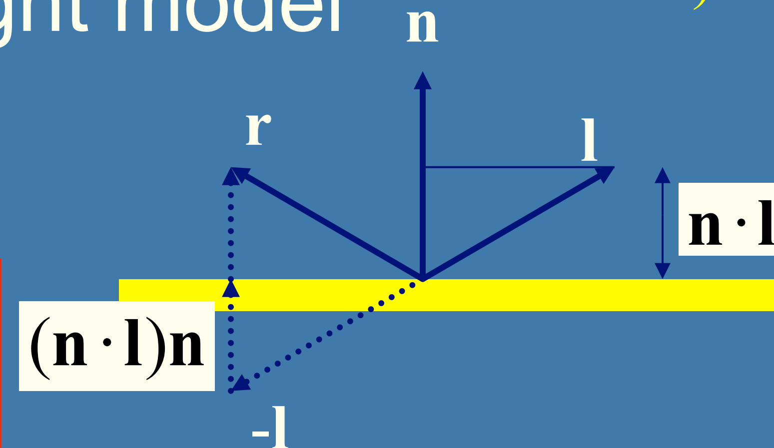
Specular component: Phong

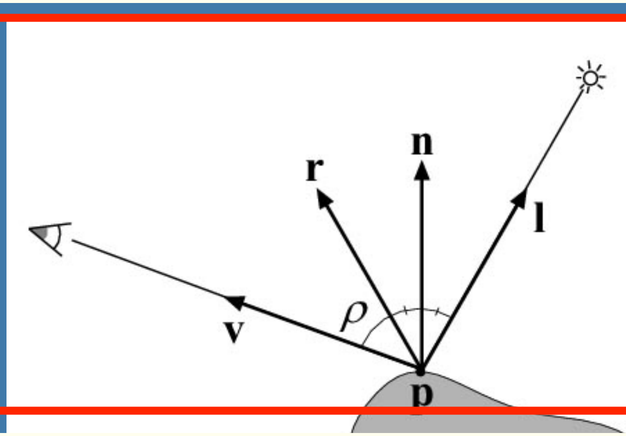
(**n** needs to be normalized)

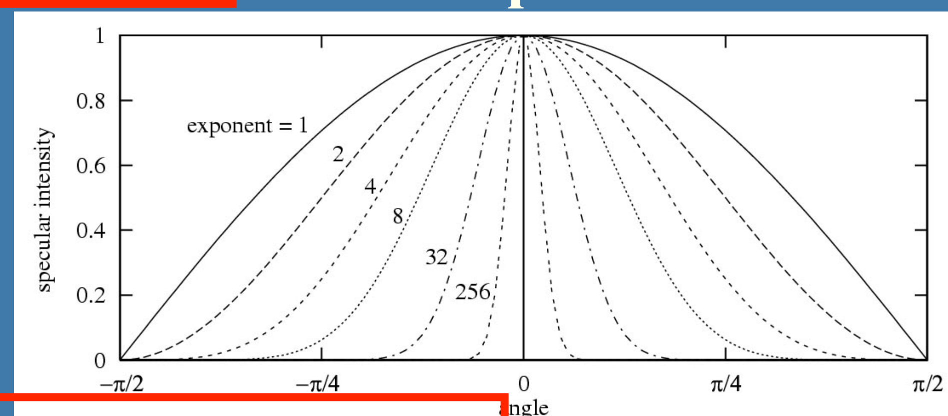
- Phong specular highlight model
- Reflect **l** around **n**:

$$\mathbf{r} = -\mathbf{l} + 2(\mathbf{n} \cdot \mathbf{l})\mathbf{n}$$

$$i_{spec} = (\mathbf{r} \cdot \mathbf{v})^{m_{shi}} = (\cos\rho)^{m_{shi}}$$







$$\mathbf{i}_{spec} = \max(0, (\mathbf{r} \cdot \mathbf{v}))^{m_{shi}} \mathbf{m}_{spec} \otimes \mathbf{s}_{spec}$$

- Next: Blinns highlight formula: $(\mathrm{n} \cdot \mathrm{h})^{\mathrm{m}}$

The University of New Mexico

Halfway Vector

Blinn proposed replacing $\mathbf{v \cdot r}$ by $\mathbf{n \cdot h}$ where

$\mathbf{h} = (\mathbf{l}+\mathbf{v})/|\mathbf{l} + \mathbf{v}|$

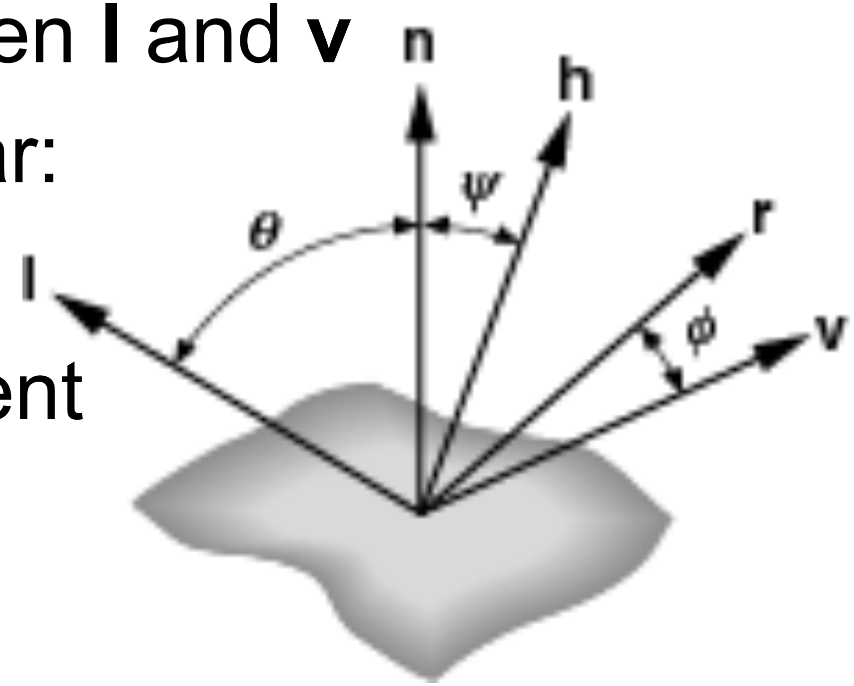
$(\mathbf{l}+\mathbf{v})/2$ is halfway between **l** and **v**

If **n**, **l**, and **v** are coplanar:

$$\psi = \phi/2$$

Must then adjust exponent so that $(\mathbf{n \cdot h})^{e'} \approx (\mathbf{r \cdot v})^{e}$

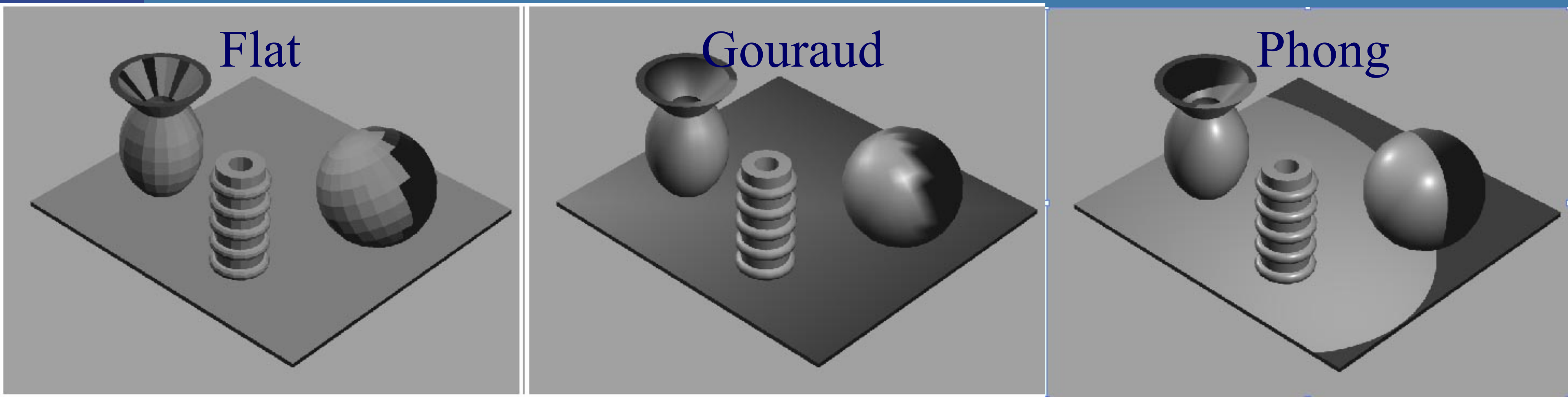
$(e' \approx 4e)$



$$\mathbf{i}_{spec} = \max(0, (\mathbf{h \cdot n})^{m_{shi}})\mathbf{m}_{spec} \otimes \mathbf{s}_{spec}$$

Shading

- Three common types of shading:
  - Flat, Goraud, and Phong
- In standard Gouraud shading the lighting is computed per triangle vertex and for each pixel, the color is interpolated from the colors at the vertices.
- In Phong Shading the lighting is **not** per vertex. Instead the normal is interpolated per pixel from the normals defined at the vertices and full lighting is computed per pixel using this normal. This is of course more expensive but looks better.



Transparency and alpha

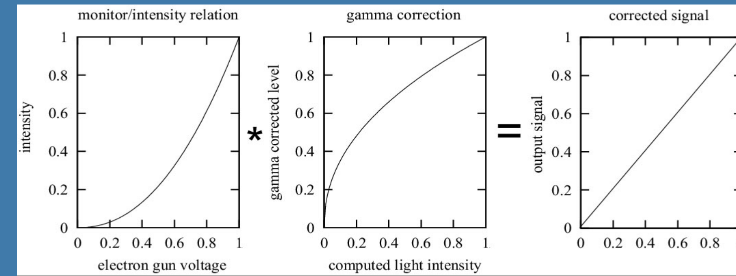
- Transparency
  - Very simple in real-time contexts
- The tool: alpha blending (mix two colors)
- Alpha (α) is another component in the frame buffer, or on triangle
  - Represents the opacity
  - 1.0 is totally opaque
  - 0.0 is totally transparent
- The over operator: $\mathbf{c}_o = \alpha \mathbf{c}_s + (1 - \alpha)\mathbf{c}_d$

Rendered object

Transparency

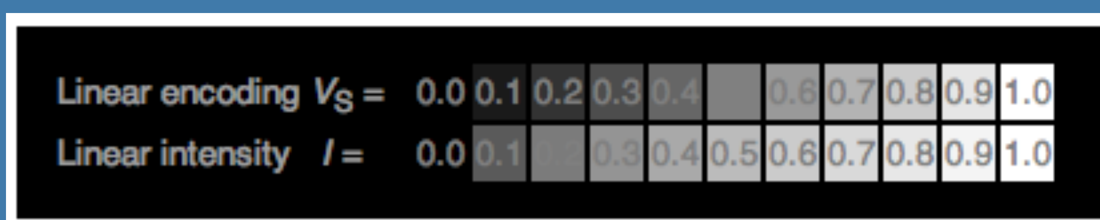
- Need to sort the transparent objects
  - First, render all non-transparent triangles as usual.
  - Then, sort all transparent triangles and render back-to-front with blending enabled. (and using standard depth test)
    - The reason is to avoid problems with the depth test and because the blending operation (i.e., over operator) is order dependent.

Gamma correction



- (Standard is 2.2)

1. Screen has non-linear color intensity
    1. We often want linear output (e.g. antialiasing)
2. Happens to give more efficient color space when compressing intensity from 32-bit floats to 8-bits. Thus, often desired when storing textures.

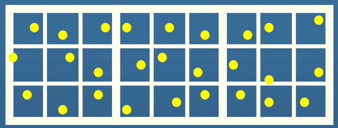


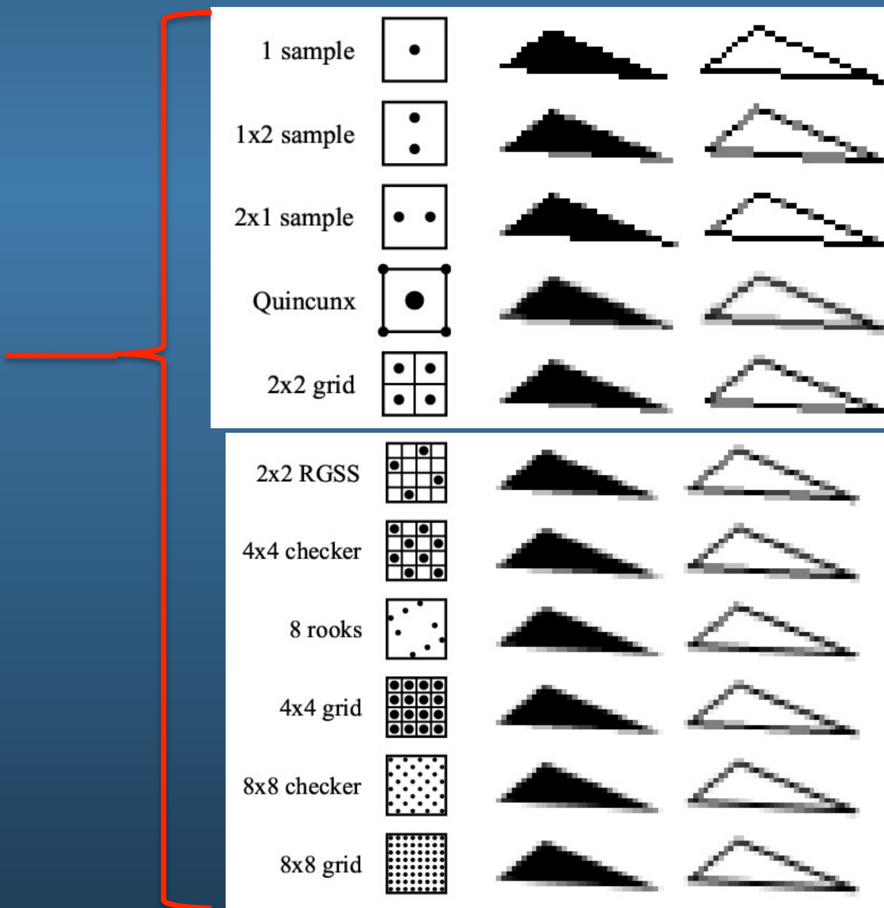
Gamma of 2.2

On most displays (those with gamma of about 2.2), one can observe that the linear-intensity scale has a large jump in perceived brightness between the intensity values 0.0 and 0.1, while the steps at the higher end of the scale are hardly perceptible. The linearly-encoded scale, which has a nonlinearly-increasing intensity, will show much more even steps in perceived brightness.

Leture 3.2: Sampling, filtrering, and Antialiasing

- When does it occur?
  - In 1) pixels, 2) time, 3) texturing
- Supersampling schemes
- Jittered sampling
  - Why is it good?





04. Texturing

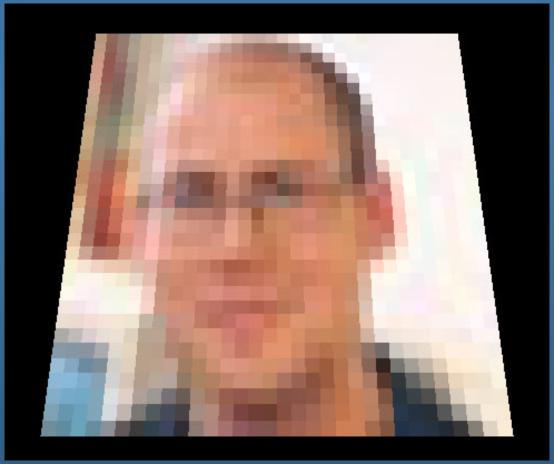
Most important:

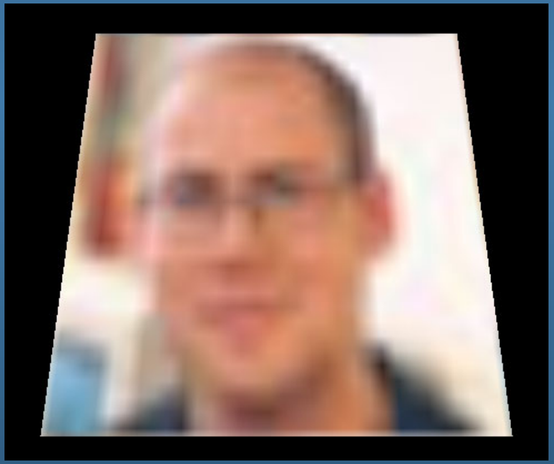
- Texturing
  - Mipmapping: bi/tri-linear filtering, anisotropic filtering
- Environment mapping
  - Cube mapping
- Bump mapping
- 3D-textures,
- Particle systems
- Sprites and billboards

Filtering

FILTERING:

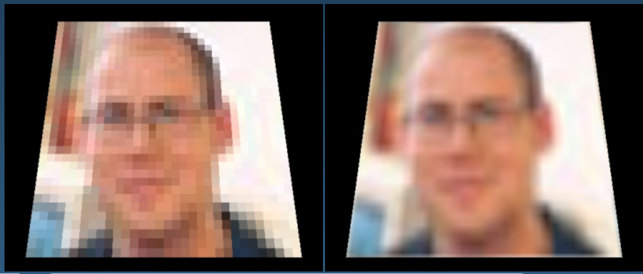
- For magnification: Nearest or Linear (box vs Tent filter)



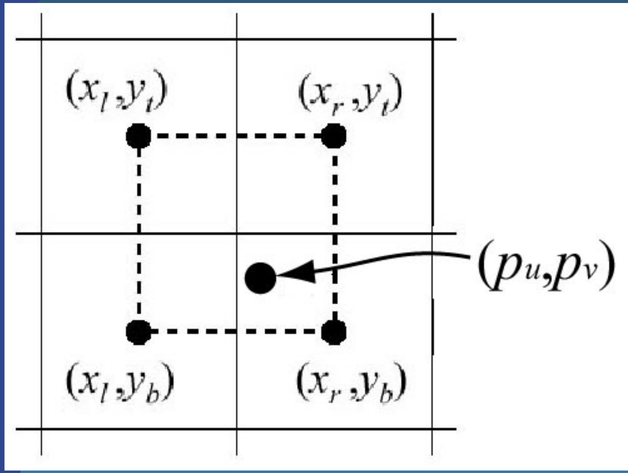


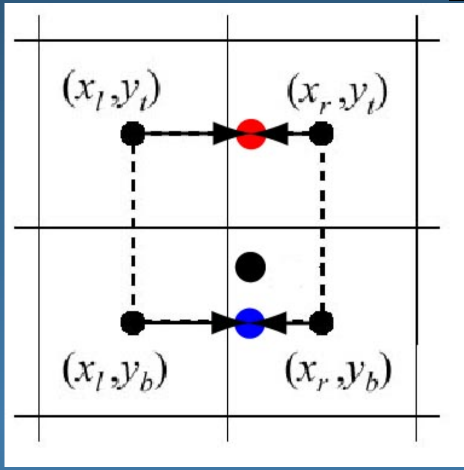
- For minification: nearest, linear and…
  - Bilinear – using mipmapping
  - Trilinear – using mipmapping
  - Anisotropic – up to 16 mipmap lookups along line of anisotropy

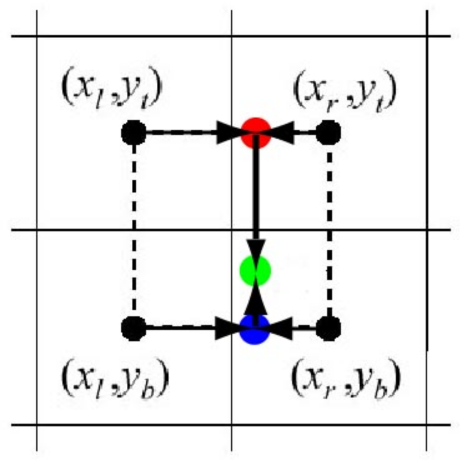
Interpolation



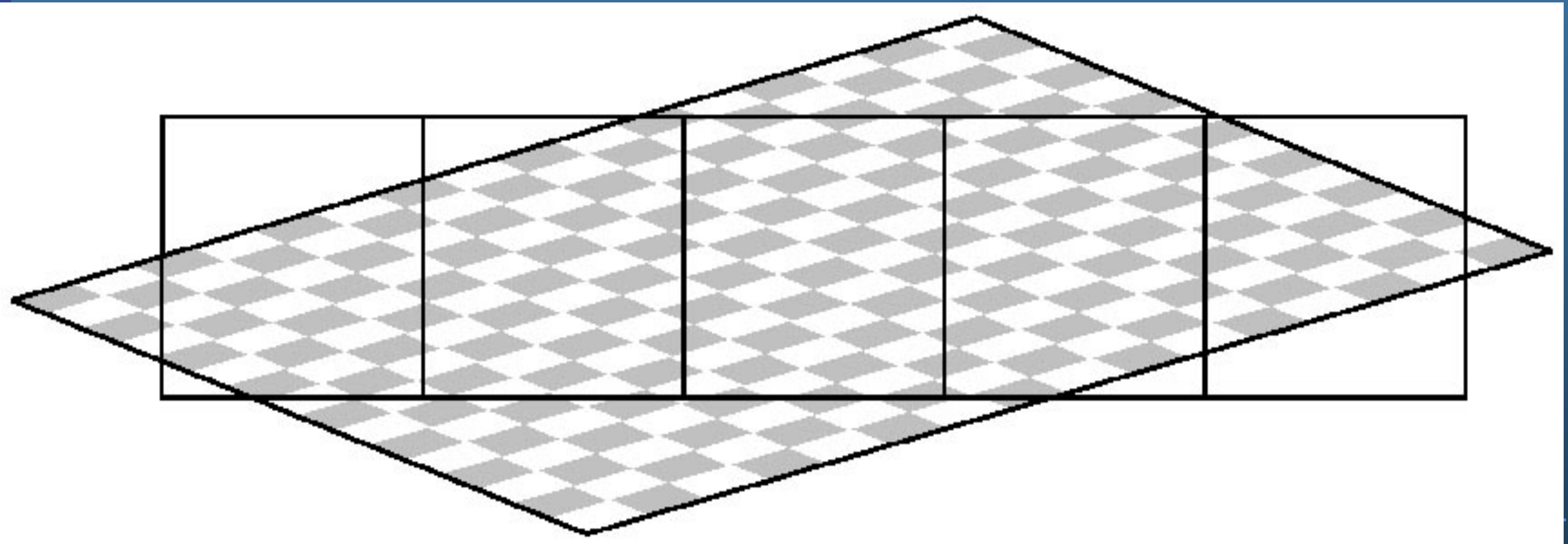
Magnification



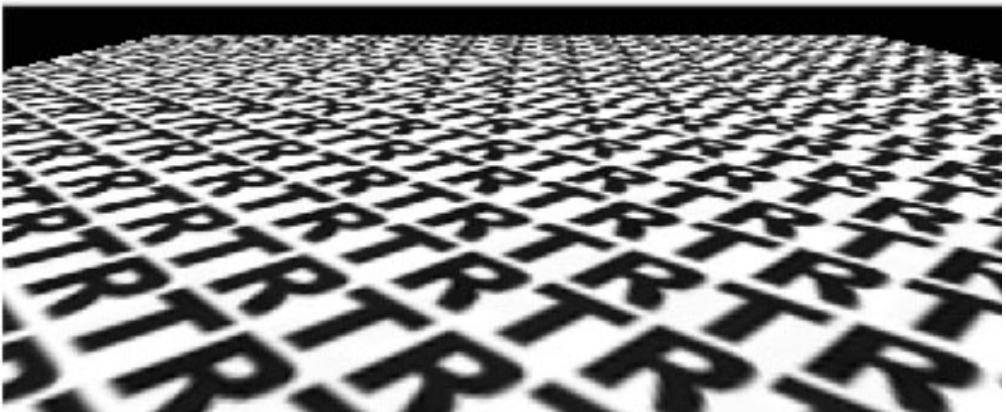


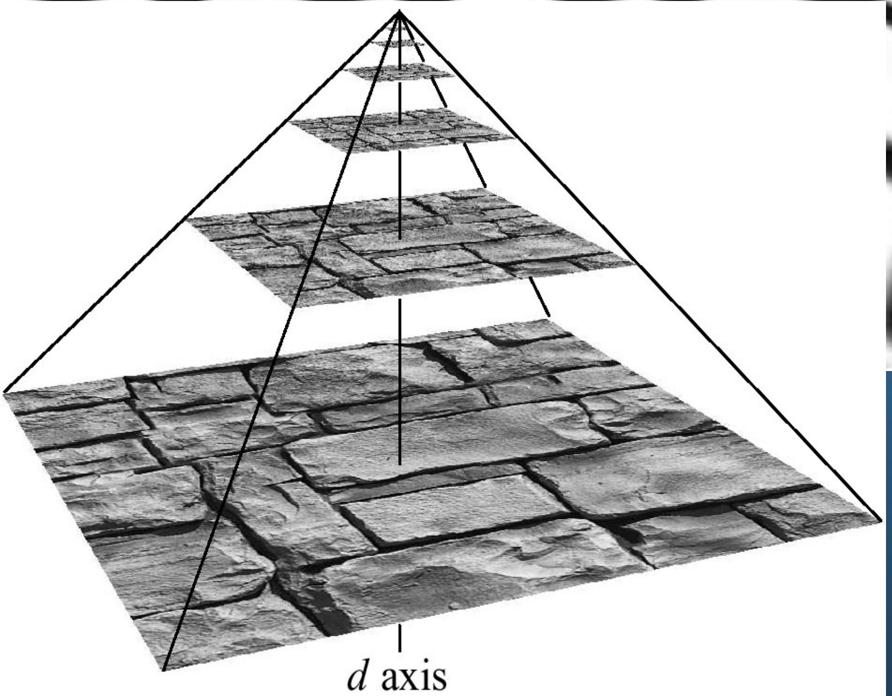


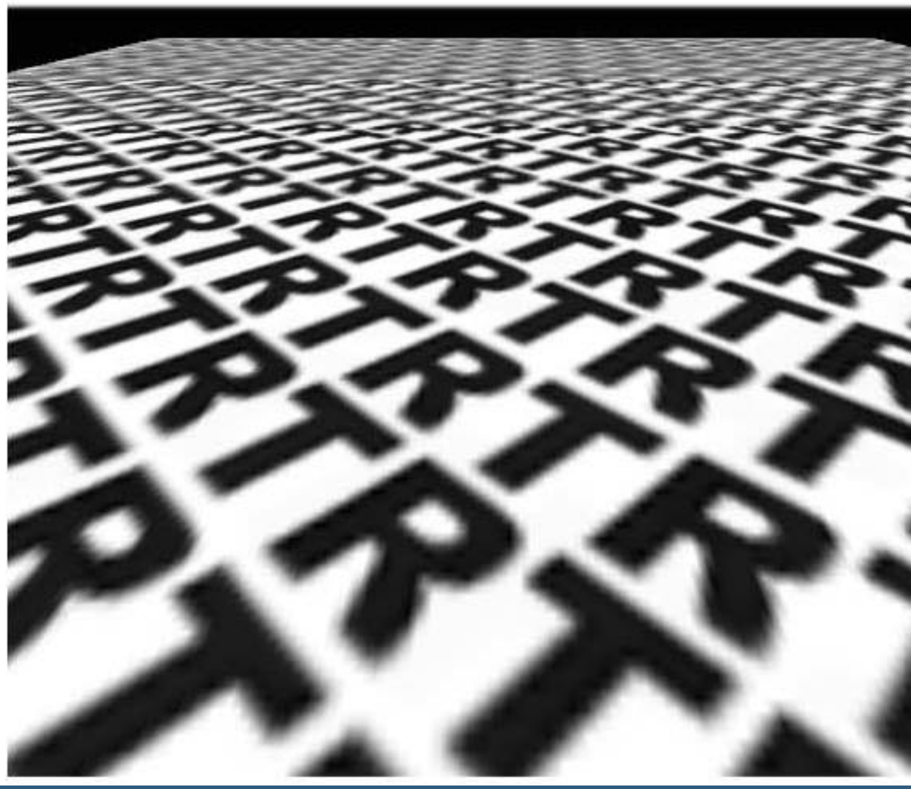
Minification



Bilinear filtering using Mipmapping

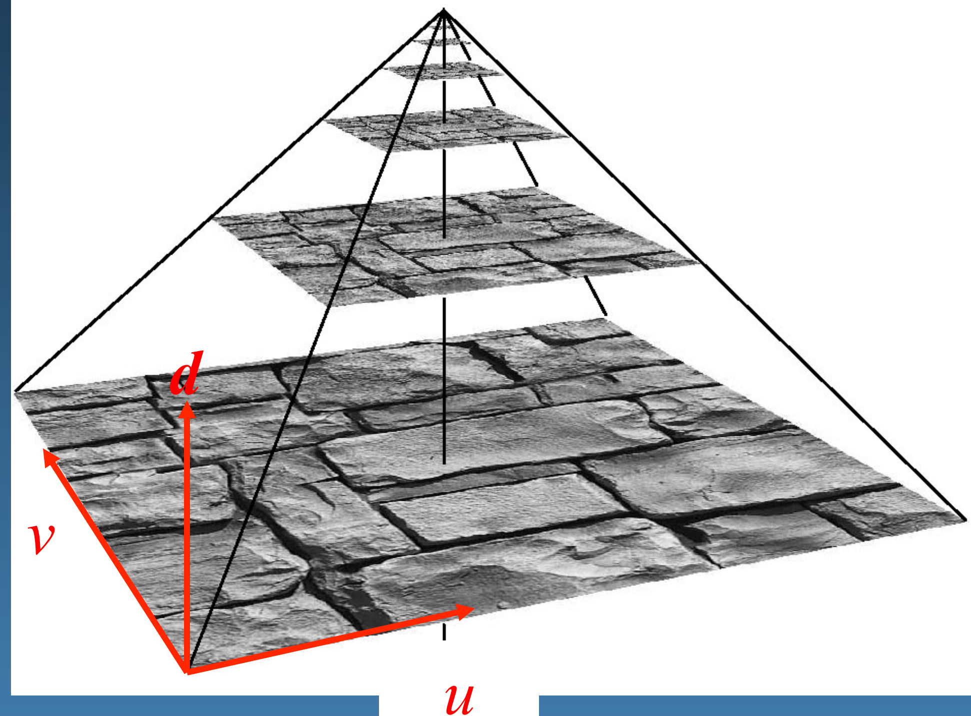






Mipmapping

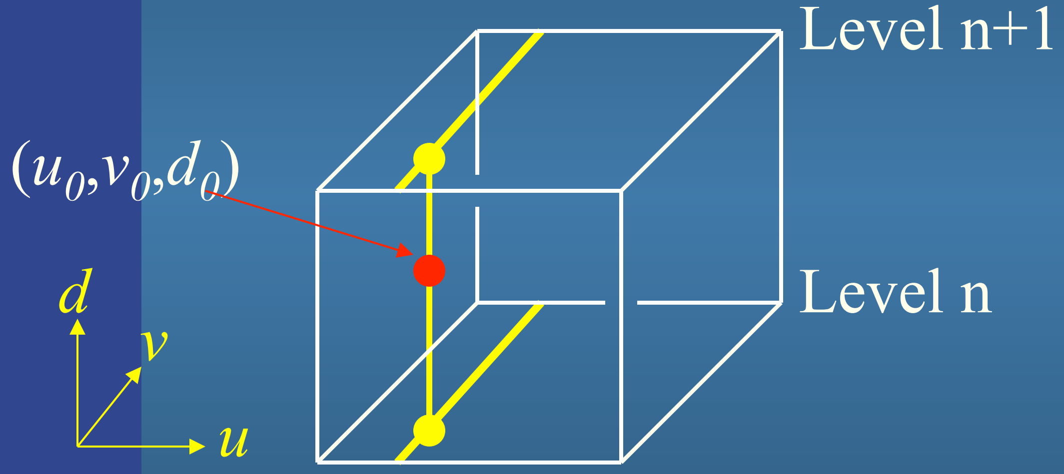
- Image pyramid
- Half width and height when going upwards



- Average over 4 ”parent texels” to form ”child texel”
- Depending on amount of minification, determine which image to fetch from
- Compute *d* first, gives two images
  - Bilinear interpolation in each

Mipmapping

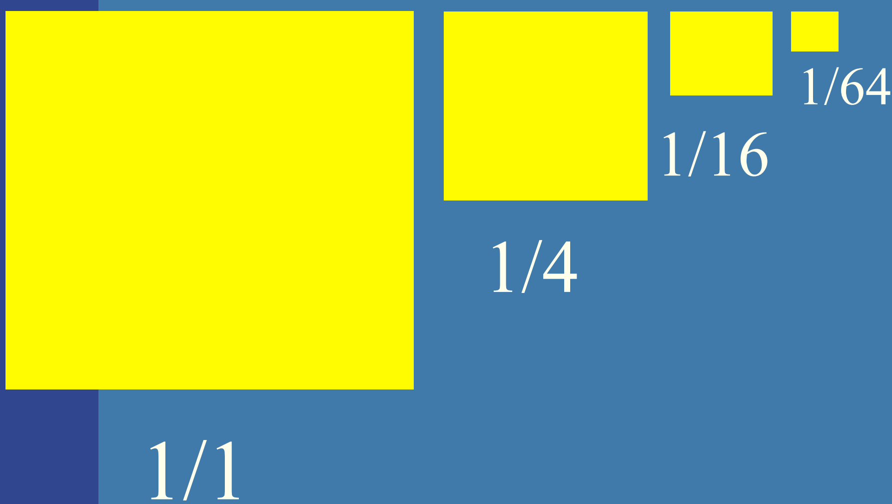
- Interpolate between those bilinear values
  - Gives trilinear interpolation

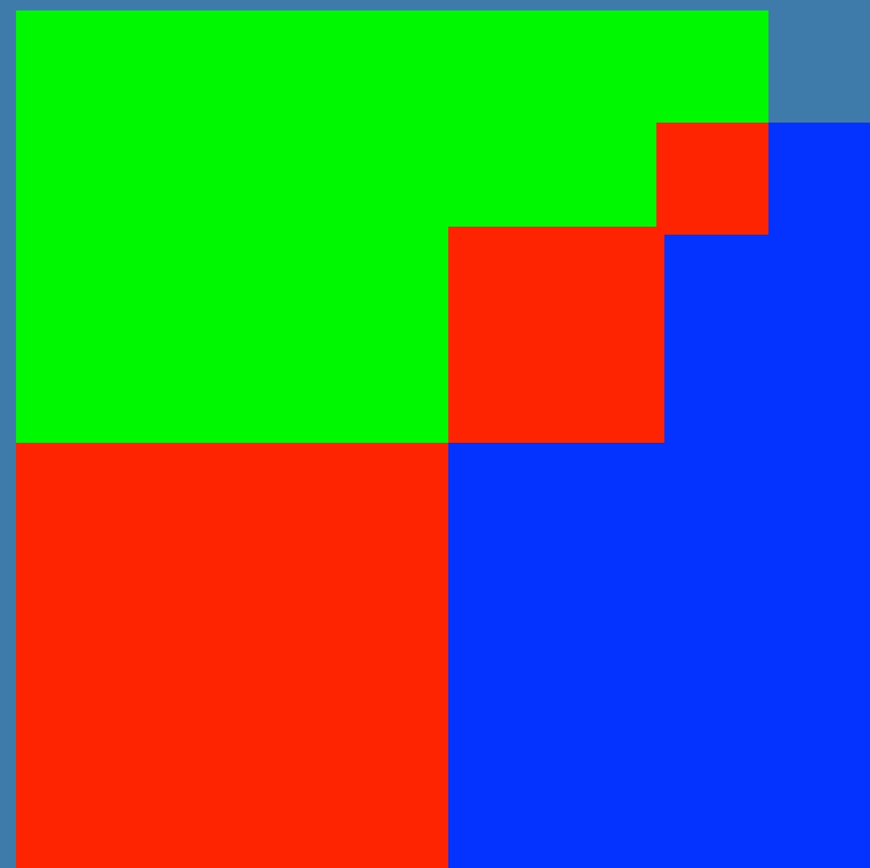


- Constant time filtering: 8 texel accesses

Mipmapping: Memory requirements

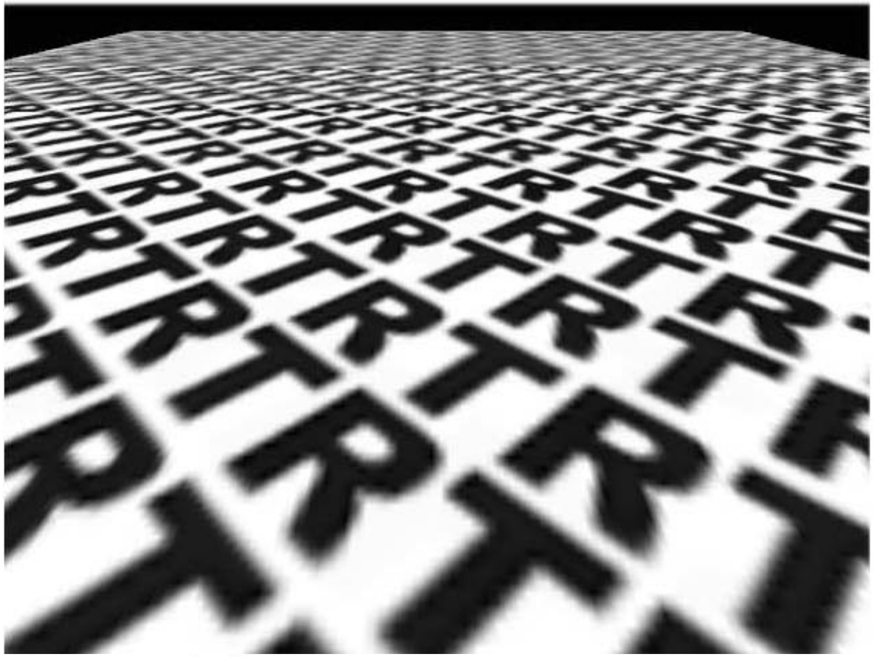
- Not twice the number of bytes…!



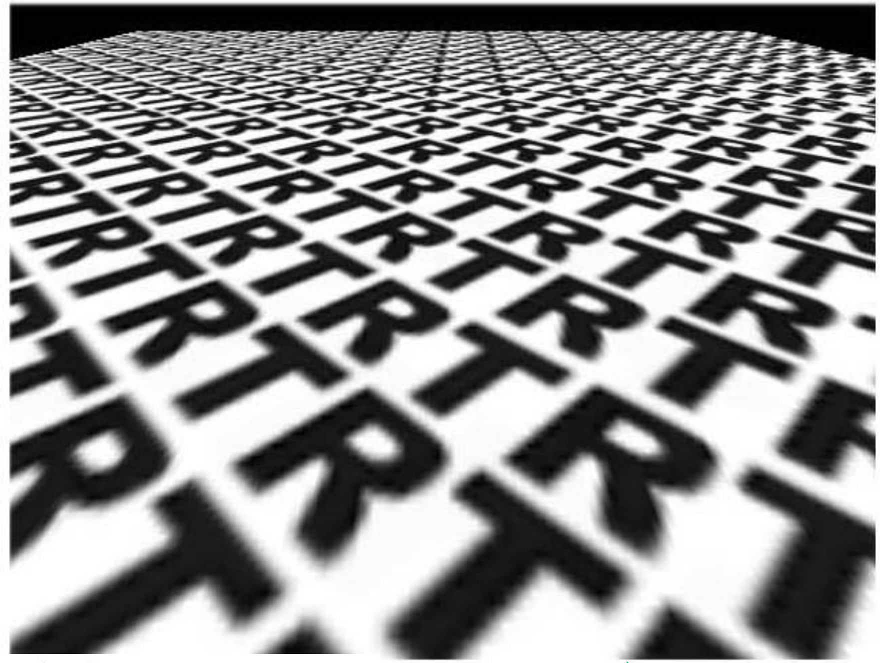


- Rather 33% more – not that much

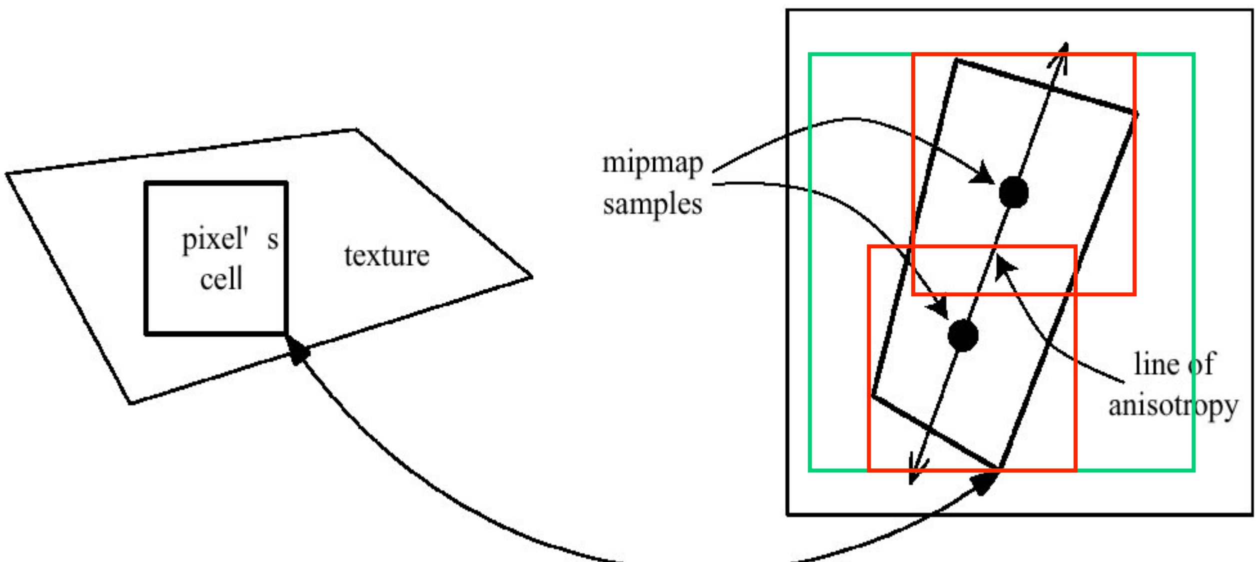
Anisotropic texture filtering



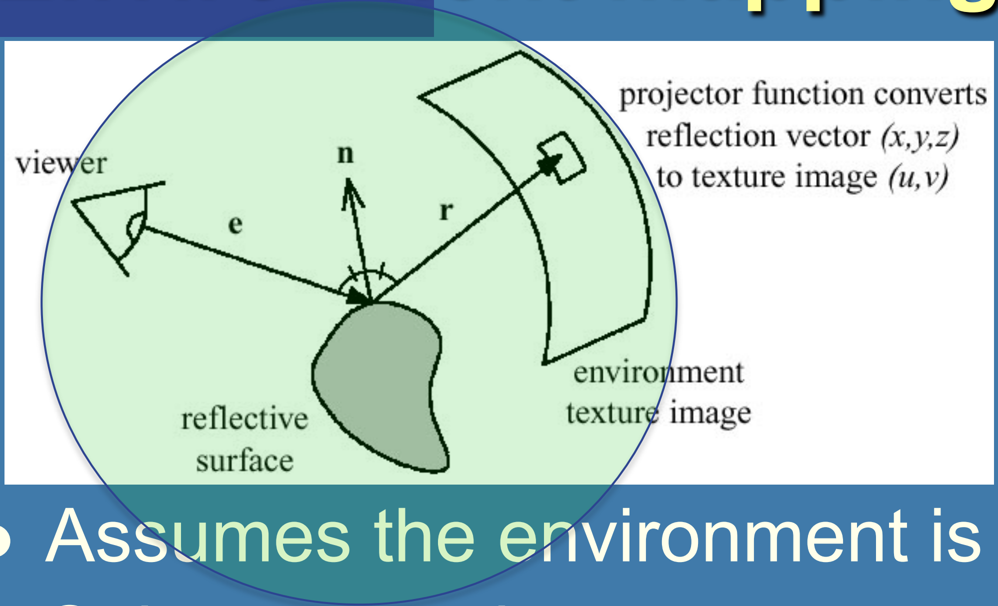
pixel space



texture space

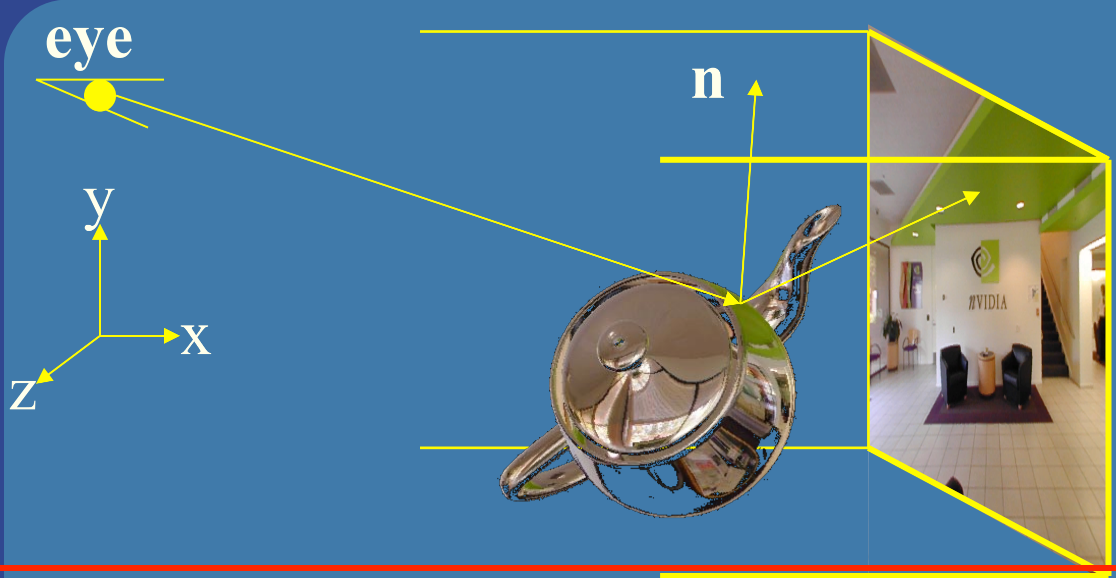


Environment mapping



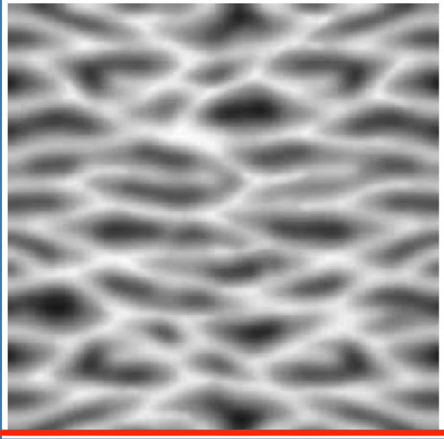
- Assumes the environment is infinitely far away
- Sphere mapping
- Cube mapping is the norm nowadays
  - Advantages: no singularities as in sphere map
  - Much less distortion
  - Gives better result
  - Not dependent on a view position

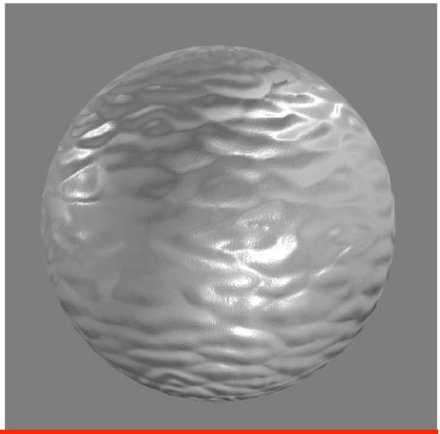
Cube mapping



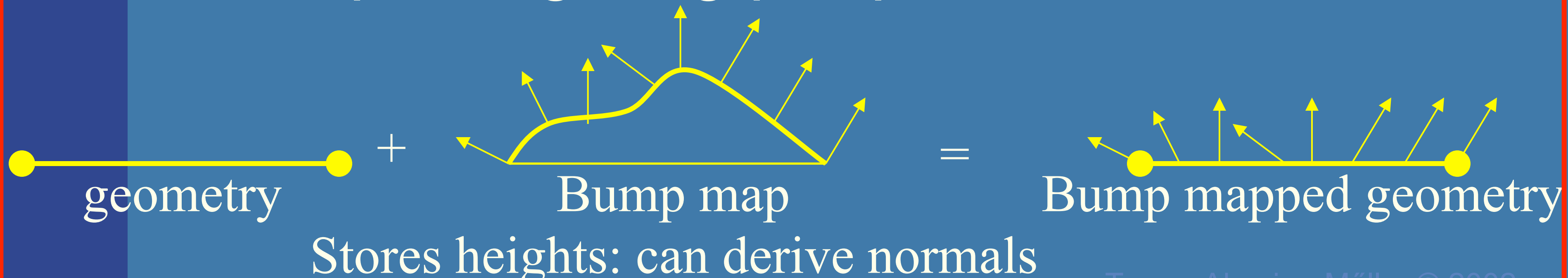
- Simple math: compute reflection vector, **r**
- Largest abs-value of component, determines which cube face.
  - Example: **r**=(5,-1,2) gives POS_X face
- Divide **r** by abs(5) gives (*u*,*v*)=(-1/5,2/5)
- Also map from [-1,1] to [0,1] by (u,v) = ((u,v)+vec2(1,1))*0.5;
- Your hardware does all the work for you

Bump mapping

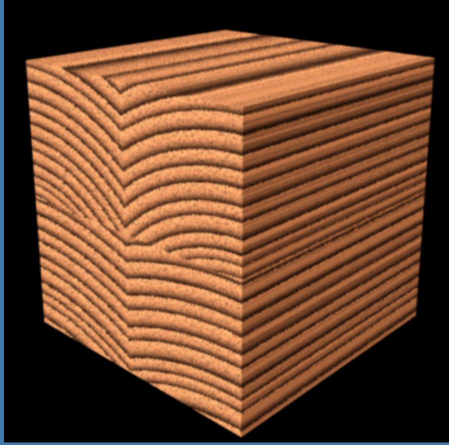




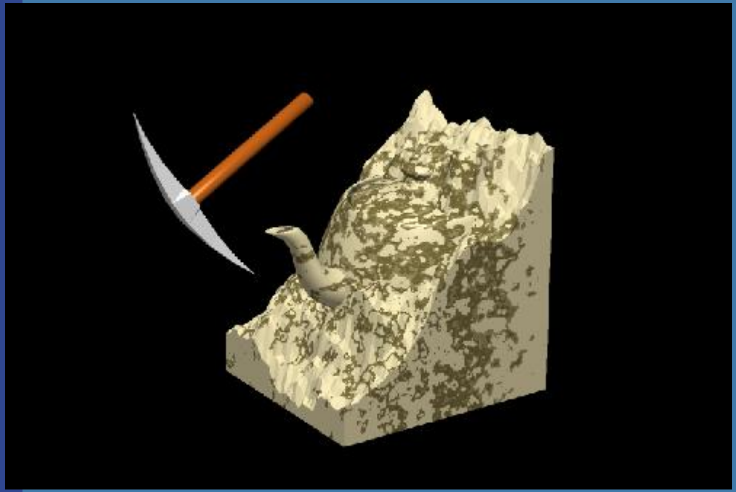
- by Blinn in 1978
- Inexpensive way of simulating wrinkles and bumps on geometry
  - Too expensive to model these geometrically
- Instead let a texture modify the normal at each pixel, and then use this normal to compute lighting per pixel

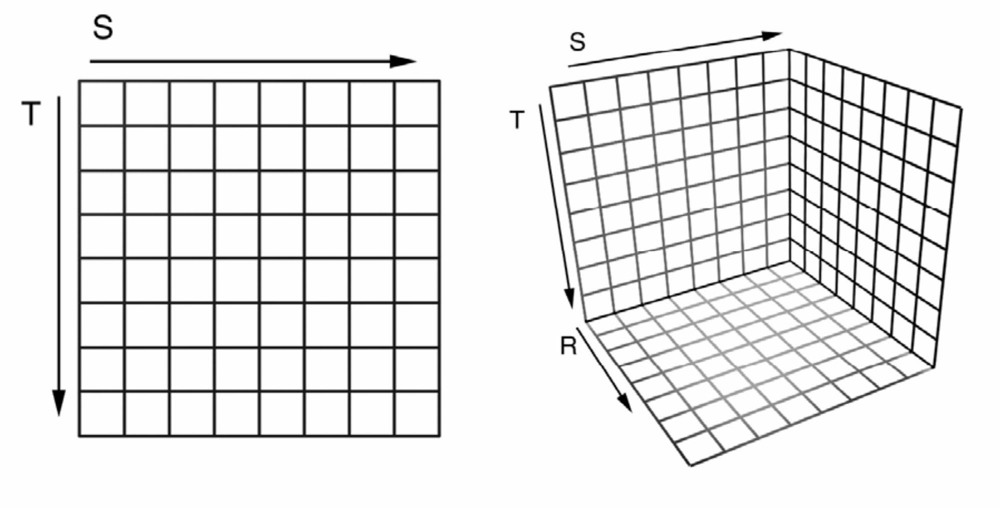


3D Textures



- 3D textures:
  - Feasible on modern hardware as well
  - Texture filtering is no longer trilinear
  - Rather quadlinear (linear interpolation 4 times)
  - Enables new possibilities
    - Can store light in a room, for example



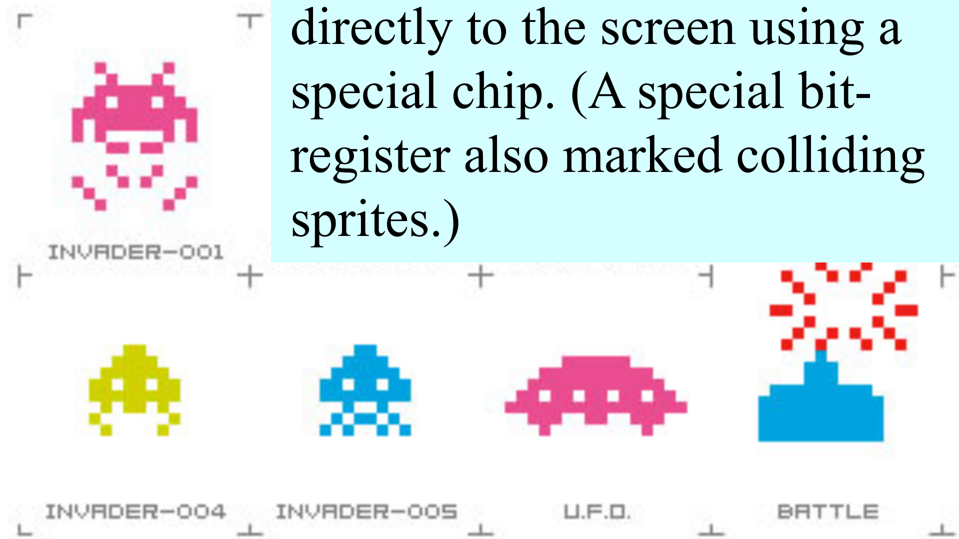


Just know what "sprites" is (i.e., similar to a billboard)

Sprites

Sprites (=älvor) was a technique on older home computers, e.g. VIC64. As opposed to billboards sprites does not use the frame buffer. They are rasterized directly to the screen using a special chip. (A special bit-register also marked colliding sprites.)

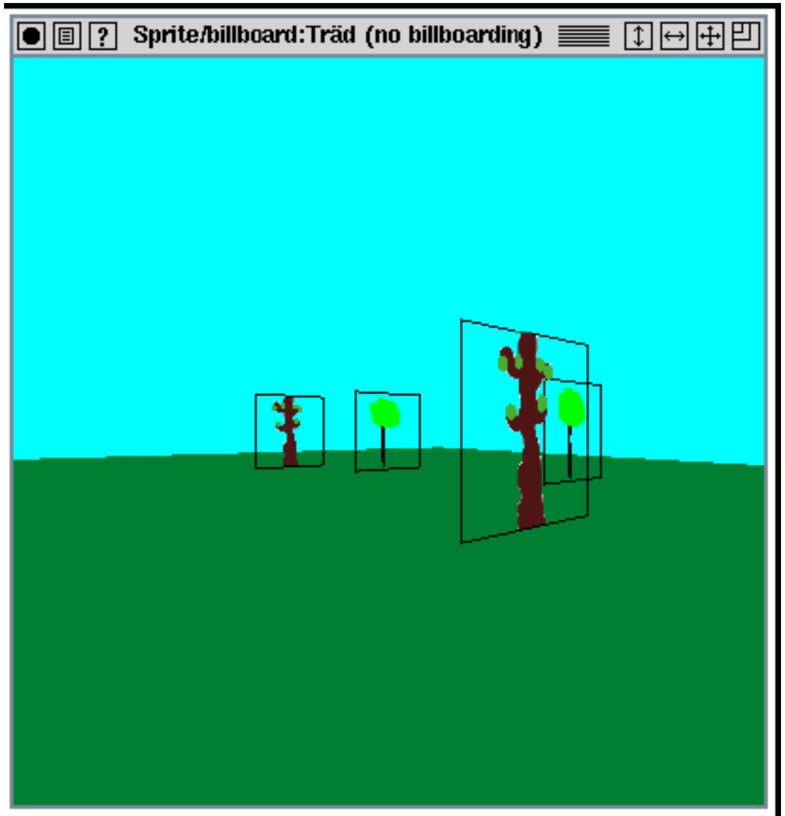
```
GLbyte M[64]=
{   127,0,0,127, 127,0,0,127,
    127,0,0,127, 127,0,0,127,
    0,127,0,0, 0,127,0,127,
    0,127,0,127, 0,127,0,0,
    0,0,127,0, 0,0,127,127,
    0,0,127,127, 0,0,127,0,
    127,127,0,0, 127,127,0,127,
    127,127,0,127, 127,127,0,0};

void display(void) {
    glClearColor(0.0,1.0,1.0,1.0);
    glClear(GL_COLOR_BUFFER_BIT);
    glEnable (GL_BLEND);
    glBlendFunc (GL_SRC_ALPHA,
          GL_ONE_MINUS_SRC_ALPHA);
    glRasterPos2d(xpos1,ypos1);
    glPixelZoom(8.0,8.0);
    glDrawPixels(width,height,
          GL_RGBA, GL_BYTE, M);

    glPixelZoom(1.0,1.0);
    glutSwapBuffers();
}
```

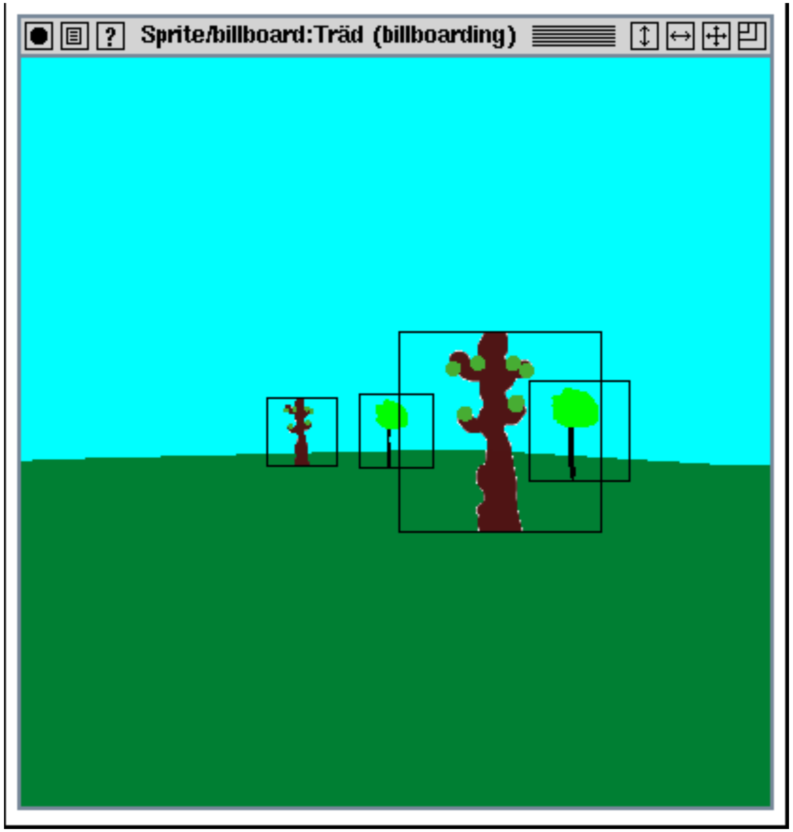


Billboards

- 2D images used in 3D environments
  - Common for trees, explosions, clouds, lens flares

Billboards





- Rotate them towards viewer
  - Either by rotation matrix or
  - by orthographic projection

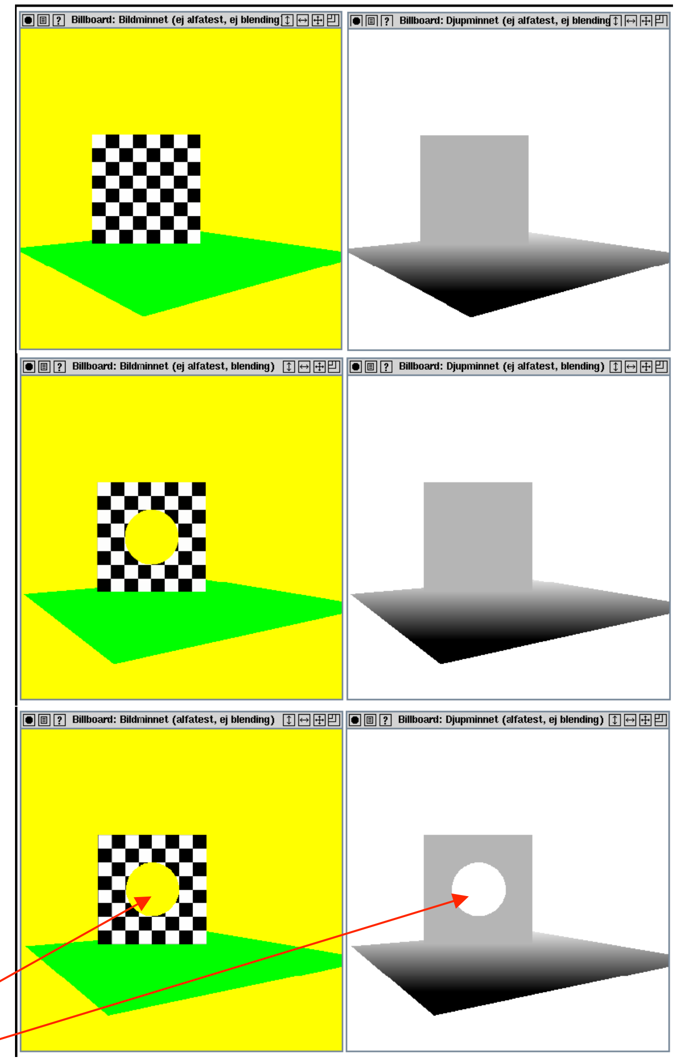
Billboards

- Fix correct transparency by blending AND using alpha-test
  - In fragment shader:

```
if (color.a < 0.1) discard;
```

If alpha value in texture is lower than this threshold value, the pixel is not rendered to. I.e., neither frame buffer nor z-buffer is updated. Which is what we want to achieve.

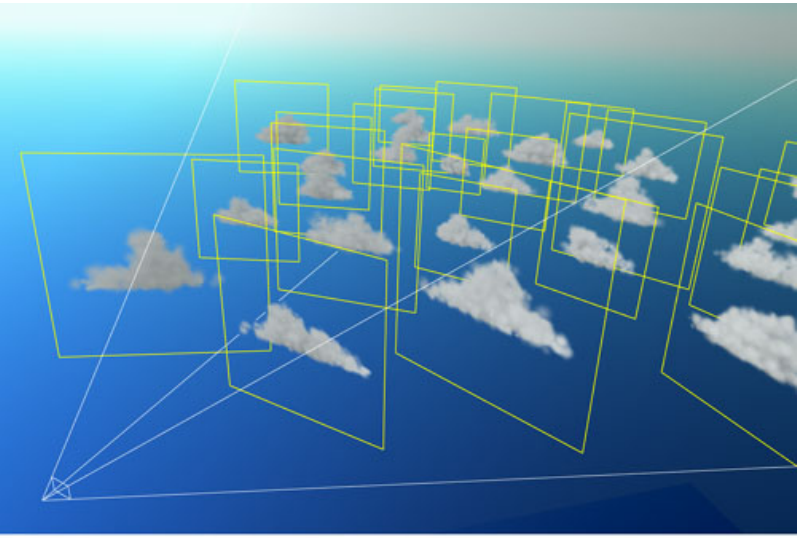
Color Buffer   Depth Buffer



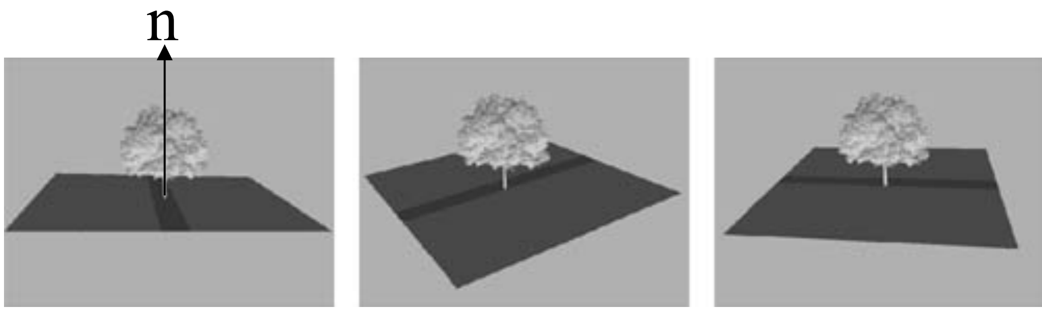
With blending

With alpha test

E.g. here: so that objects behind show through the hole



(Also called *Impostors)*



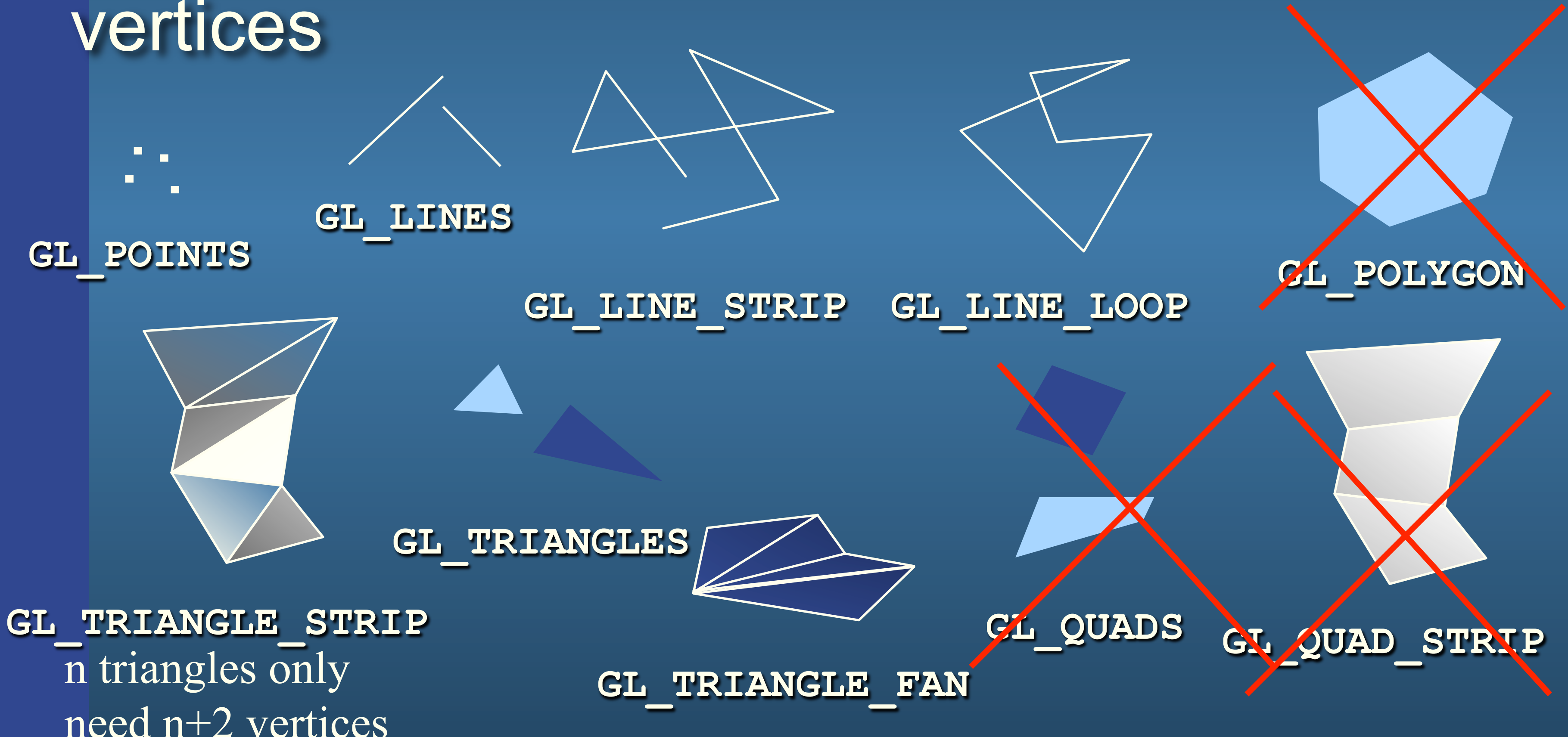
axial billboarding
The rotation axis is fixed and disregarding the view position

Lecture 5: OpenGL

- Uses OpenGL (or DirectX)
  - Will not ask about syntax. Know how to use.
    - I.e. functionality
  - E.g. how to achieve
    - Transparency
    - Fog(start, stop, linear/exp/exp-squared)
    - Specify a material, a triangle, how to translate or rotate an object.

OpenGL Geometric Primitives

- All geometric primitives are specified by vertices



Coordinate transformations

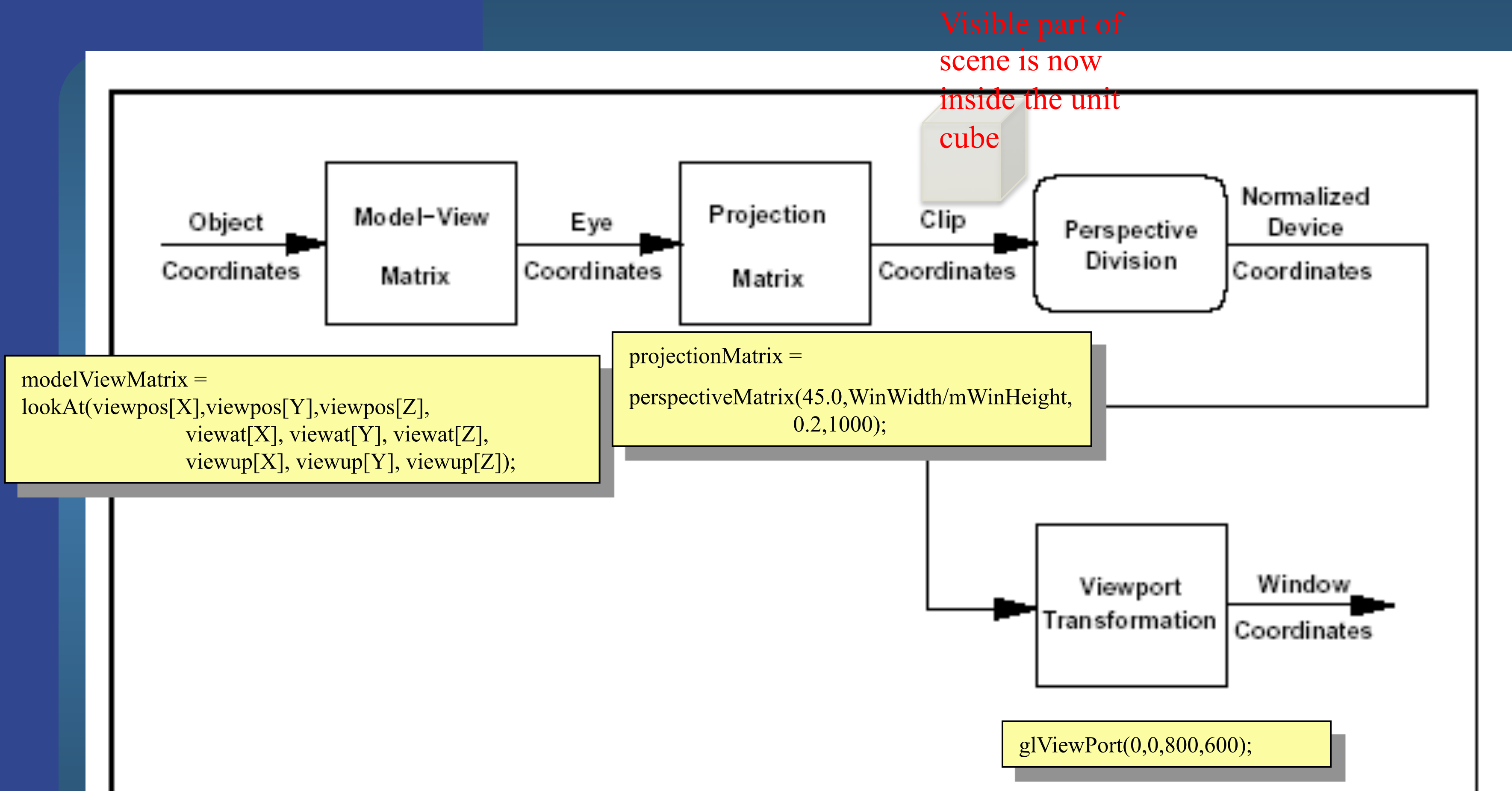


Figure 2.6. Vertex transformation sequence.

Reflections with environment mapping

- Uses the active texture as an environment map

VERTEX SHADER

```
in vec3        vertex;
in  vec3       normalIn;      // The normal
out vec3       normal;
out vec3       eyeVector;
uniform mat4 normalMatrix;
uniform mat4 modelViewMatrix;
uniform mat4 modelViewProjectionMatrix;

void main()
{
    gl_Position = modelViewProjectionMatrix *vec4(vertex,1);
    normal = (normalMatrix * vec4(normalIn,0.0)).xyz;
    eyeVector = (modelViewMatrix * vec4(vertex, 1)).xyz;
}
```

FRAGMENT SHADER

```
in vec3 normal;
in vec3 eyeVector;
uniform samplerCube tex1;
out vec4 fragmentColor;

void main()
{
    vec3 reflectionVector = normalize(reflect(normalize(eyeVector),
                                              normalize(normal)));
    fragmentColor = texture(tex1, reflectionVector);
}
```

Buffers

- Frame buffer
  - Back/front/left/right – **glDrawBuffers()**
- Depth buffer (z-buffer)
  - For correct depth sorting
  - Instead of BSP-algorithm, painters algorithm…
  - **glDepthFunc(), glDepthMask**
- Stencil buffer
  - Shadow volumes,
  - **glStencilFunc(), glStencilMask, glStencilMaskSeparate, glStencilOp**
- General commands:
  - **glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT)**
  - Specify clearing value:, **glClearStencil(), glClearColor()**

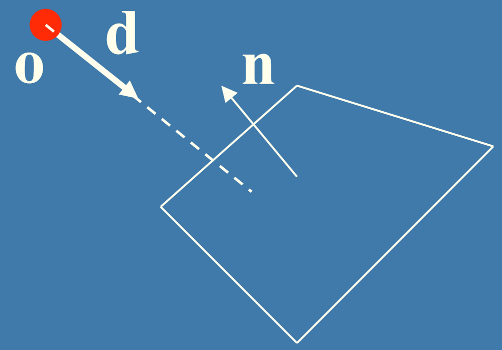
Lecture 6: Intersection Tests

- 4 techniques to compute intersections:
  - Analytically
  - Geometrically – e.g. ray vs box (3 slabs)
  - SAT (Separating Axis Theorem) for convex polyhedra

    Test:

    1. axes orthogonal to side of A,
    2. axes orthogonal to side of B
    3. All different axes formed by crossprod of one edge of A and one of B
  - Dynamic tests – know what it means.
- E.g., describe an algorithm for intersection between a **ray** and a
  - Polygon, triangle, sphere or plane.
- Know equations for ray, sphere, cylinder, plane, triangle

Analytical: Ray/plane intersection



- Ray: $\mathbf{r}(t)=\mathbf{o}+t\mathbf{d}$
- Plane formula: $\mathbf{n}\bullet\mathbf{p} + d = 0$

- Replace **p** by **r**(*t*) and solve for t:

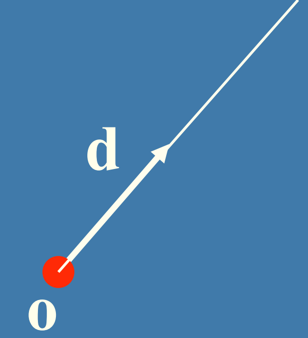
$\mathbf{n}\bullet(\mathbf{o}+t\mathbf{d}) + d = 0$

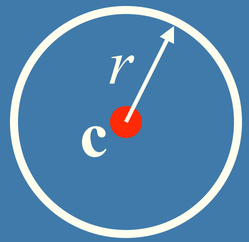
$\mathbf{n}\bullet\mathbf{o}+t\mathbf{n}\bullet\mathbf{d} + d = 0$

$t = (-d - \mathbf{n}\bullet\mathbf{o}) / (\mathbf{n}\bullet\mathbf{d})$

Here, one scalar equation and one unknown -> just solve for t.

Analytical: Ray/sphere test





- Sphere center: **c**, and radius *r*
- Ray: **r**(*t*)=**o**+*t***d**
- Sphere formula: ||**p**-**c**||=*r*
- Replace **p** by **r**(*t*): ||**r**(*t*)-**c**||=*r*

$$(\mathbf{r}(t)-\mathbf{c})\cdot(\mathbf{r}(t)-\mathbf{c})-r^2=0$$

$$(\mathbf{o}+t\mathbf{d}-\mathbf{c})\cdot(\mathbf{o}+t\mathbf{d}-\mathbf{c})-r^2=0$$

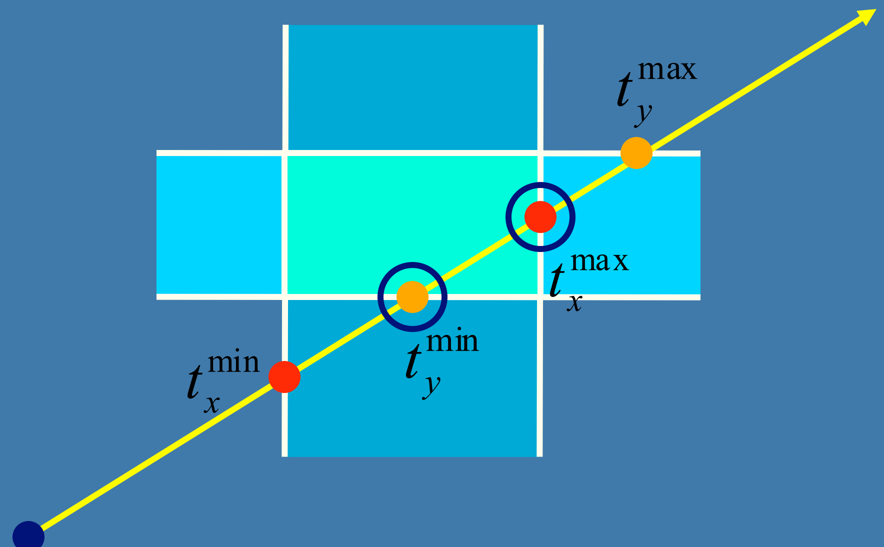
$$(\mathbf{d}\cdot\mathbf{d})t^2+2((\mathbf{o}-\mathbf{c})\cdot\mathbf{d})t+(\mathbf{o}-\mathbf{c})\cdot(\mathbf{o}-\mathbf{c})-r^2=0$$

$$t^2+2((\mathbf{o}-\mathbf{c})\cdot\mathbf{d})t+(\mathbf{o}-\mathbf{c})\cdot(\mathbf{o}-\mathbf{c})-r^2=0 \qquad \|\mathbf{d}\|=1$$

This is a standard quadratic equation. Solve for t.

Geometrical: Ray/Box Intersection (2)

- Intersect the 2 planes of each slab with the ray



- Keep max of t^{min} and min of t^{max}
- If $t^{min} < t^{max}$ then we got an intersection
- Special case when ray parallell to slab

Point/Plane

Plane: $\pi : \mathbf{n} \cdot \mathbf{p} + d = 0$

- Insert a point $\mathbf{x}$ into plane equation:

$$f(\mathbf{x}) = \mathbf{n} \cdot \mathbf{x} + d = ?$$

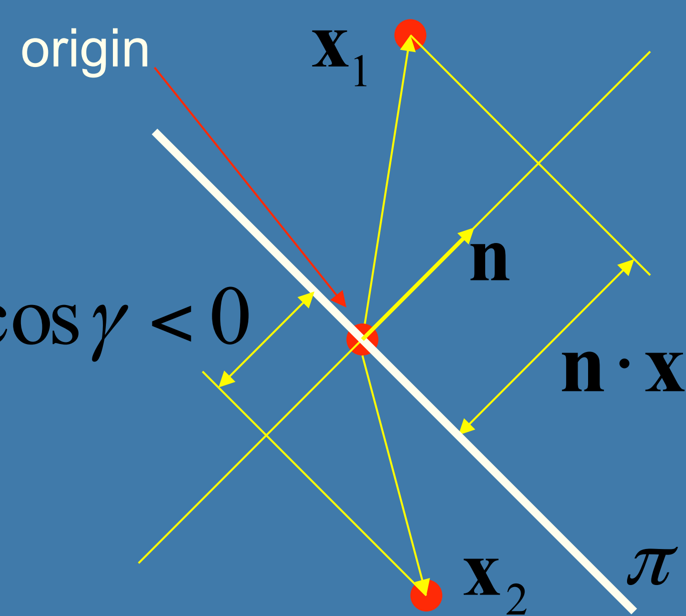
$$f(\mathbf{x}) = \mathbf{n} \cdot \mathbf{x} + d = 0 \quad \text{for } \mathbf{x}\text{'s on the plane}$$

Negative half space

$$f(\mathbf{x}) = \mathbf{n} \cdot \mathbf{x} + d < 0 \quad \text{for } \mathbf{x}\text{'s on one side of the plane}$$

Positive half space

$$f(\mathbf{x}) = \mathbf{n} \cdot \mathbf{x} + d > 0 \quad \text{for } \mathbf{x}\text{'s on the other side}$$



$$\mathbf{n} \cdot \mathbf{x}_2 = \| \mathbf{x}_2 \| \cos \gamma < 0$$

$$\mathbf{n} \cdot \mathbf{x}_1 = \| \mathbf{x}_1 \| \cos \phi > 0$$

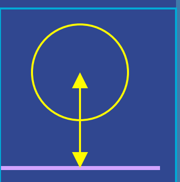
Sphere/Plane
AABB/Plane

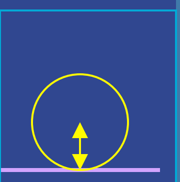
Plane: $\pi : \mathbf{n}\cdot\mathbf{p} + d = 0$

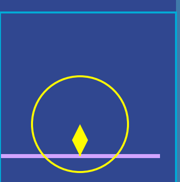
Sphere: $\mathbf{c}$ $\quad r$

Box: $\mathbf{b}^{\min}$ $\quad \mathbf{b}^{\max}$

- Sphere: compute $f(\mathbf{c}) = \mathbf{n}\cdot\mathbf{c} + d$
- $f(\mathbf{c})$ is the signed distance ($\mathbf{n}$ normalized)
- $\mathrm{abs}(f(\mathbf{c})) > \mathrm{r}$ no collision
- $\mathrm{abs}(f(\mathbf{c})) = \mathrm{r}$ sphere touches the plane
- $\mathrm{abs}(f(\mathbf{c})) < \mathrm{r}$ sphere intersects plane

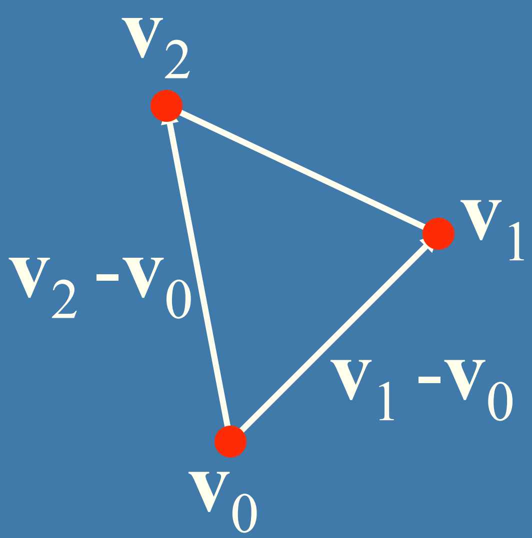






- Box: insert all 8 corners
- If all f's have the same sign, then all points are on the same side, and no collision

Another analytical example: Ray/Triangle in detail

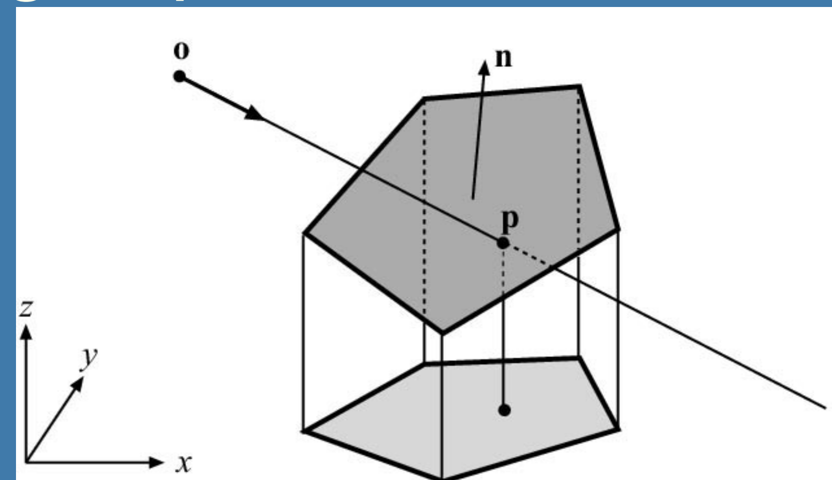


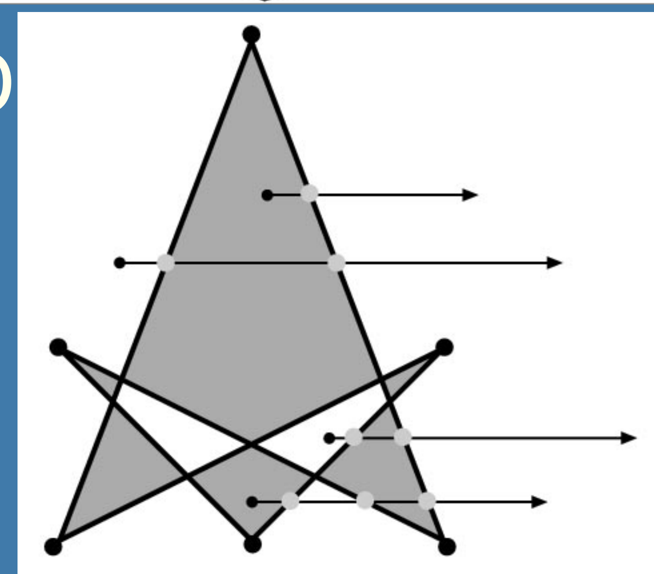
- Ray: $\mathbf{r}(t)=\mathbf{o}+t\mathbf{d}$
- Triangle vertices: $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2$
- A point in the triangle:
- $\mathbf{t}(u,v)=\mathbf{v}_0+u(\mathbf{v}_1-\mathbf{v}_0)+v(\mathbf{v}_2-\mathbf{v}_0)=$
  $=(1-u-v)\mathbf{v}_0+u\mathbf{v}_1+v\mathbf{v}_2 \quad [u,v>=0,\ u+v<=1]$
- Set $\mathbf{t}(u,v)=\mathbf{r}(t)$, and solve!

$$\begin{pmatrix} | & | & | \\ -\mathbf{d} & \mathbf{v}_1-\mathbf{v}_0 & \mathbf{v}_2-\mathbf{v}_0 \\ | & | & | \end{pmatrix}\begin{pmatrix} t \\ u \\ v \end{pmatrix} = \begin{pmatrix} | \\ \mathbf{o}-\mathbf{v}_0 \\ | \end{pmatrix}$$

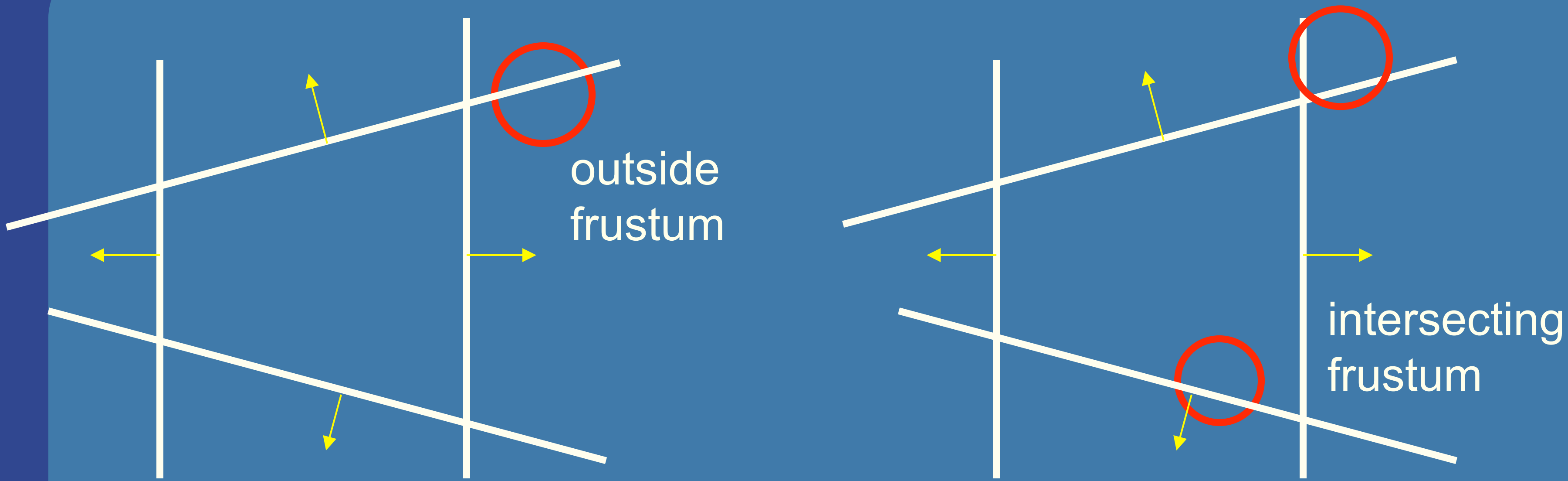
Ray/Polygon: very briefly

- Intersect ray with polygon plane
- Project from 3D to 2D
- How?
- Find $\max(|n_x|,|n_y|,|n_z|)$
- Skip that coordinate!
- Then, count crossing in 2D





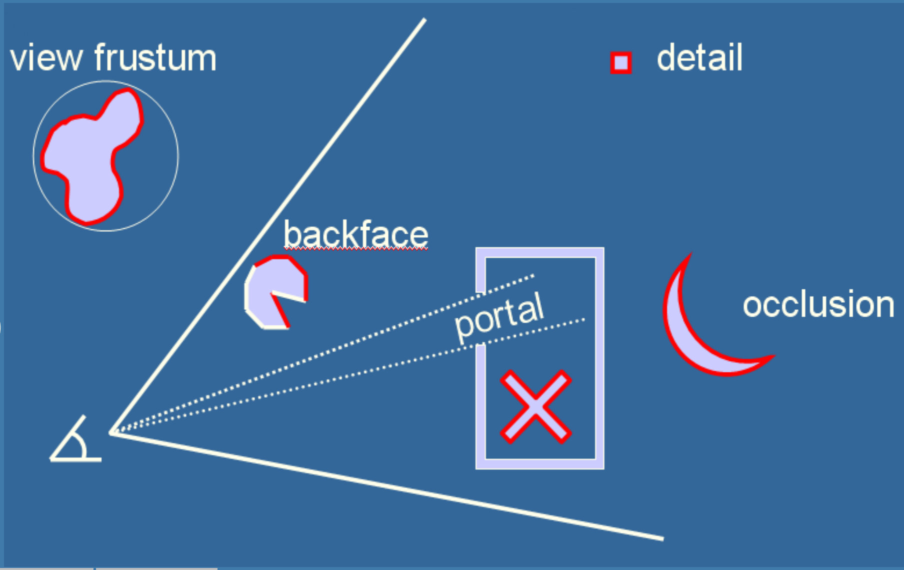
View frustum testing example



- Algo:
  - if sphere is outside any of the 6 frustum planes -> report ”outside”.
  - Else report intersect.
- Not exact test, but not incorrect
  - A sphere that is reported to be inside, can be outside
  - Not vice versa, so test is conservative

Lecture 7.1: Spatial Data Structures and Speed-Up Techniques

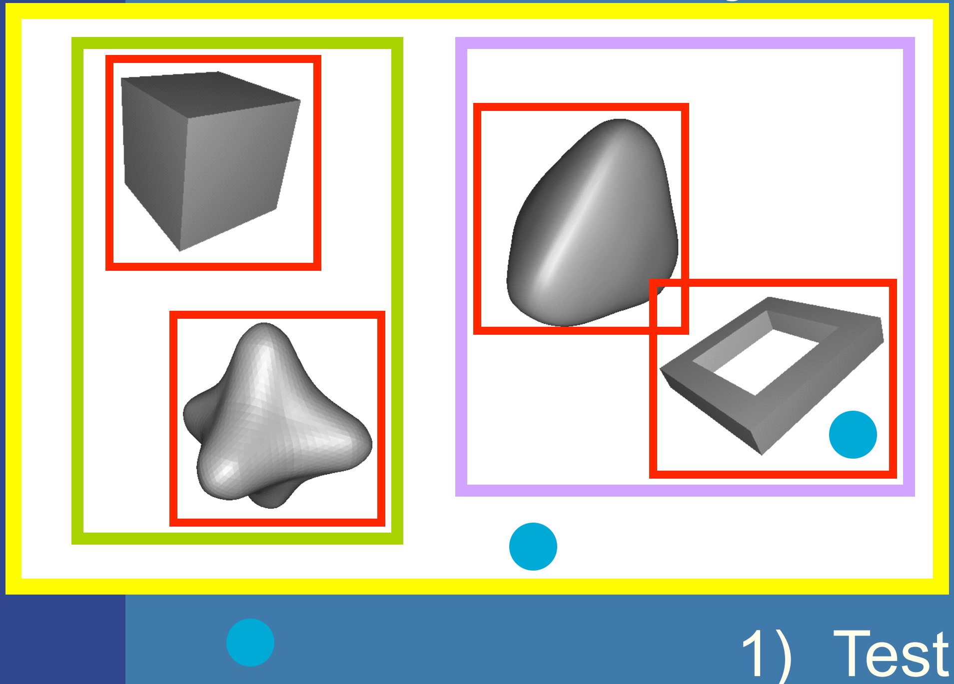
- Speed-up techniques
  - Culling
    - Backface
    - View frustum (hierarchical)
    - Portal
    - Occlusion Culling
    - Detail
  - Levels-of-detail:

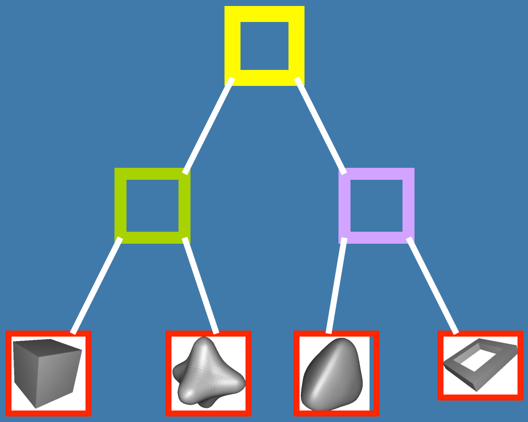


- How to construct and use the spatial data structures
    - BVH, BSP-trees (polygon aligned + axis aligned)

Axis Aligned Bounding Box Hierarchy - an example

- Assume we click on screen, and want to find which object we clicked on





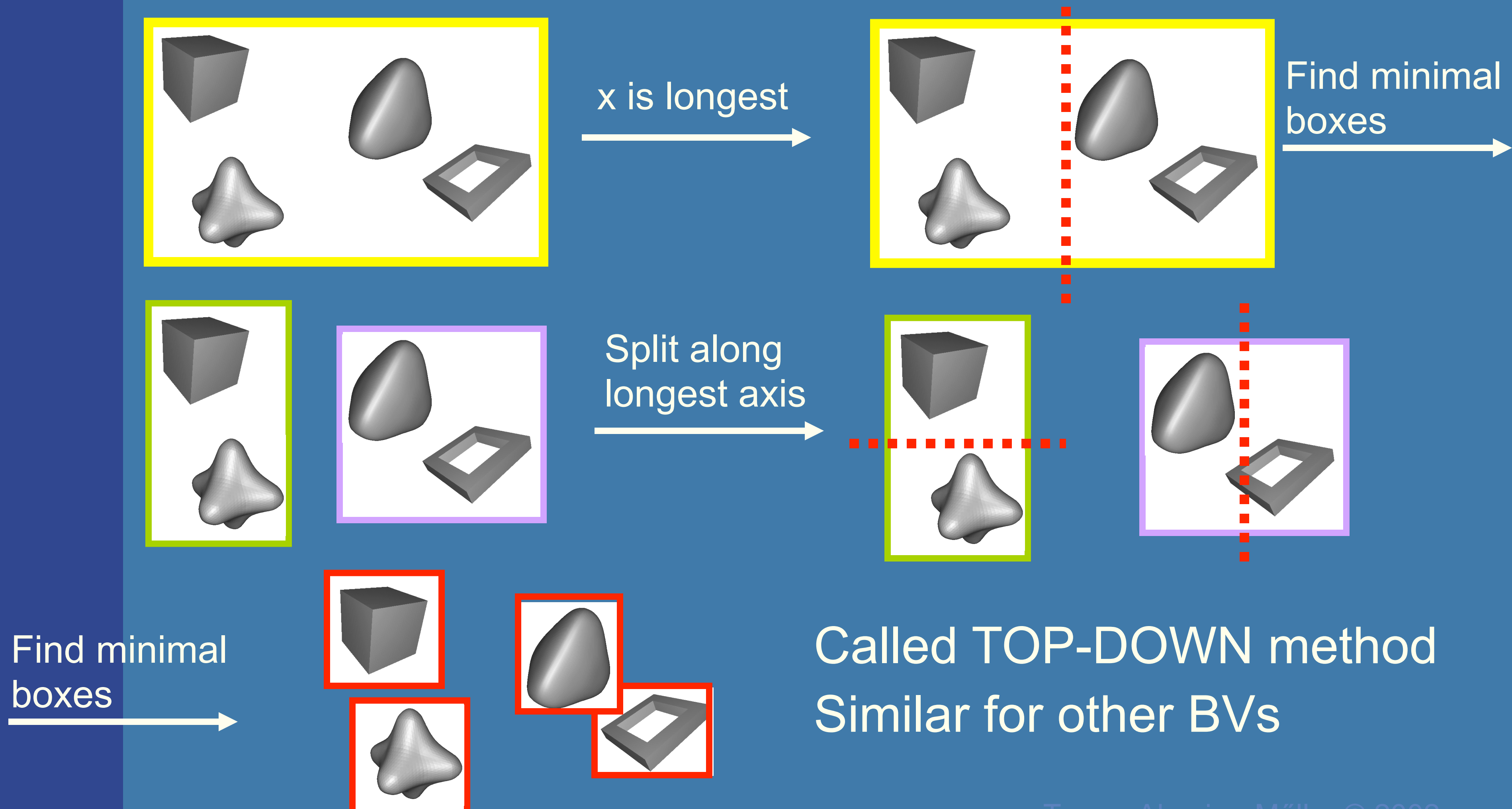
1) Test the root first
2) Descend recursively as needed
3) Terminate traversal when possible

In general: get O(log n) instead of O(n)

How to create a BVH?
Example: using AABBs

AABB = Axis Aligned Bounding Box
BVH = Bounding Volume Hierarchy

- Find minimal box, then split along longest axis

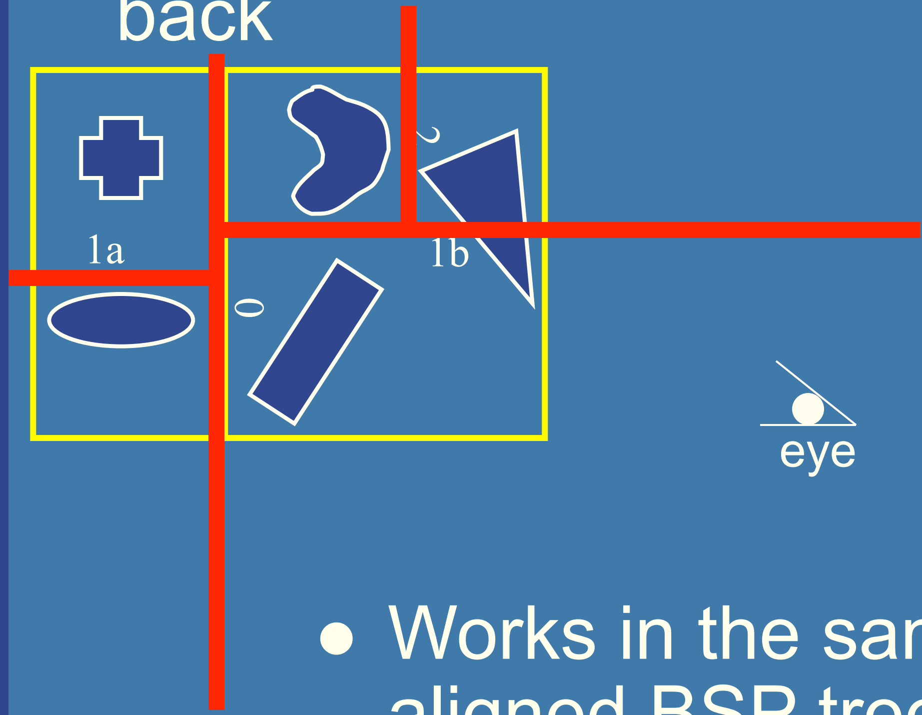


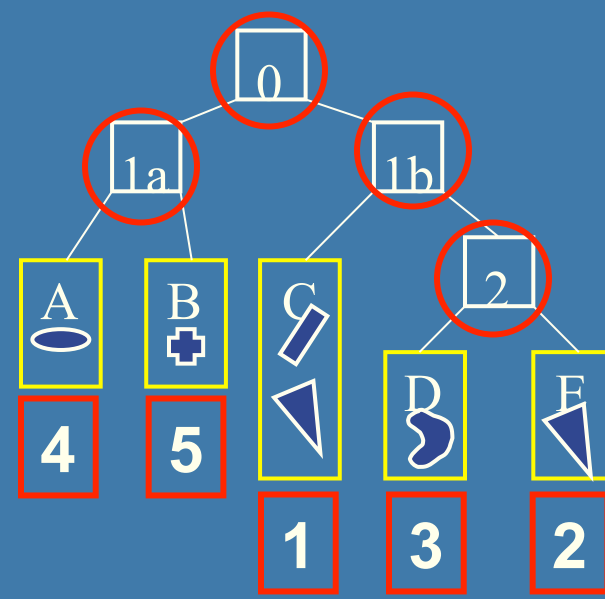
Called TOP-DOWN method

Similar for other BVs

Axis-aligned BSP tree
Rough sorting

- Test the planes against the point of view
- Test recursively from root
- Continue on the ”hither” side to sort front to back

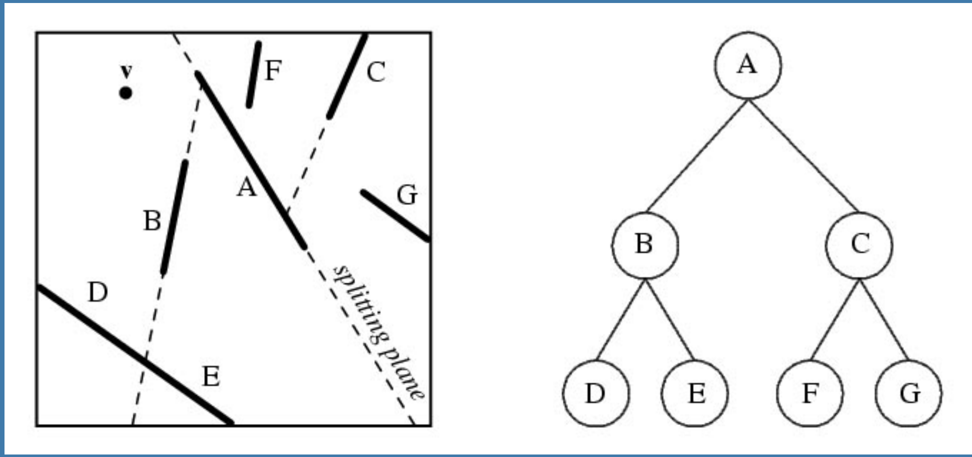




- Works in the same way for polygon-aligned BSP trees --- but that gives exact sorting

Polygon-aligned BSP tree

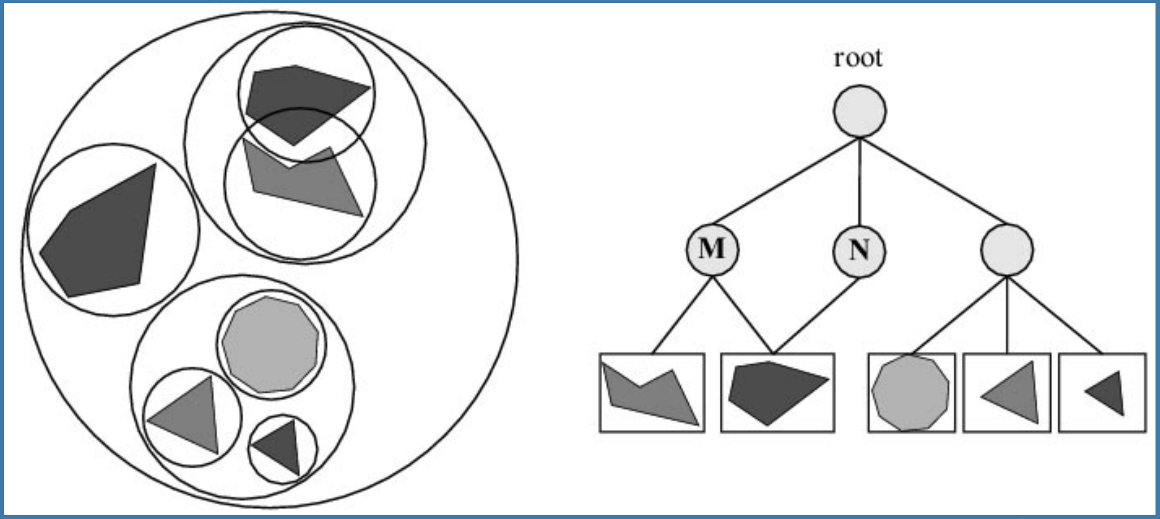
- Allows exact sorting
- Very similar to axis-aligned BSP tree
  - But the splitting plane are now located in the planes of the triangles



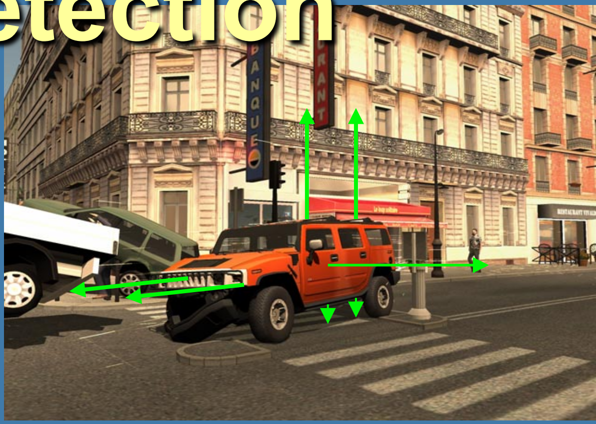
Know how to build it and how to traverse back-to-front or front-to-back with respect to the eye position (here: **v**)

Scene graphs

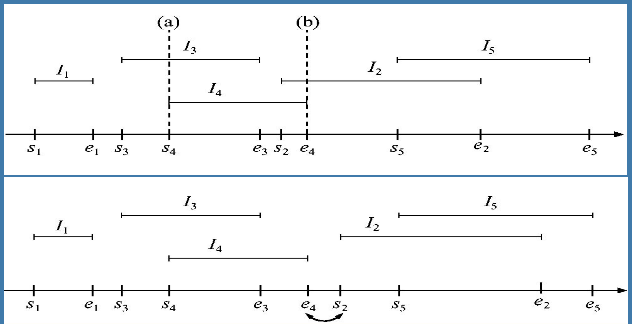
- BVH is the data structure that is used most often
  - Simple to understand
  - Simple code
- However, BVH stores just geometry
  - Rendering is more than geometry
- The scene graph is an extended BVH with:
  - Lights
  - Materials
  - Transforms
  - several connections to a node
  - And more



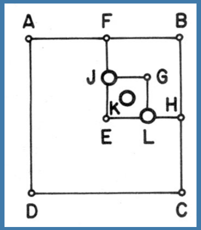
Lecture 7.2: Collision Detection

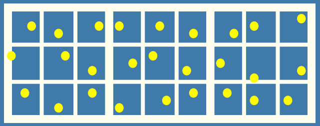


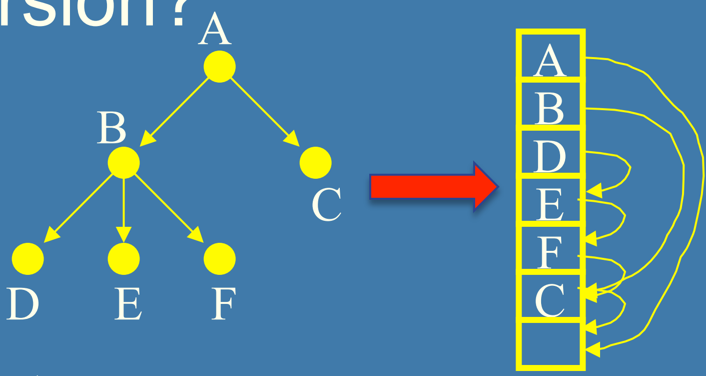
- 3 types of algorithms:
  - With rays
    - Fast but not exact
  - With BVH
    - You should be able to write pseudo code for BVH/BVH test, for coll det between two objects.
    - Slower but exact
  - For many many objects.
    - why? Course pruning of ”obviously” non-colliding objects
    - Sweep-and-prune



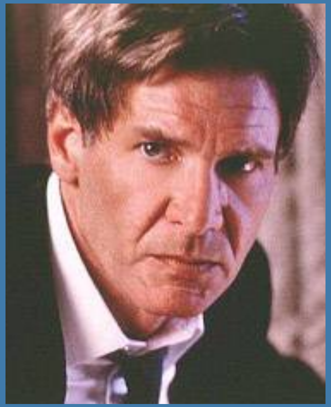
Lecture 8: Ray tracing



- **Adaptive Super Sampling**
- **Jittering** 
- How to stop ray tracing recursion?

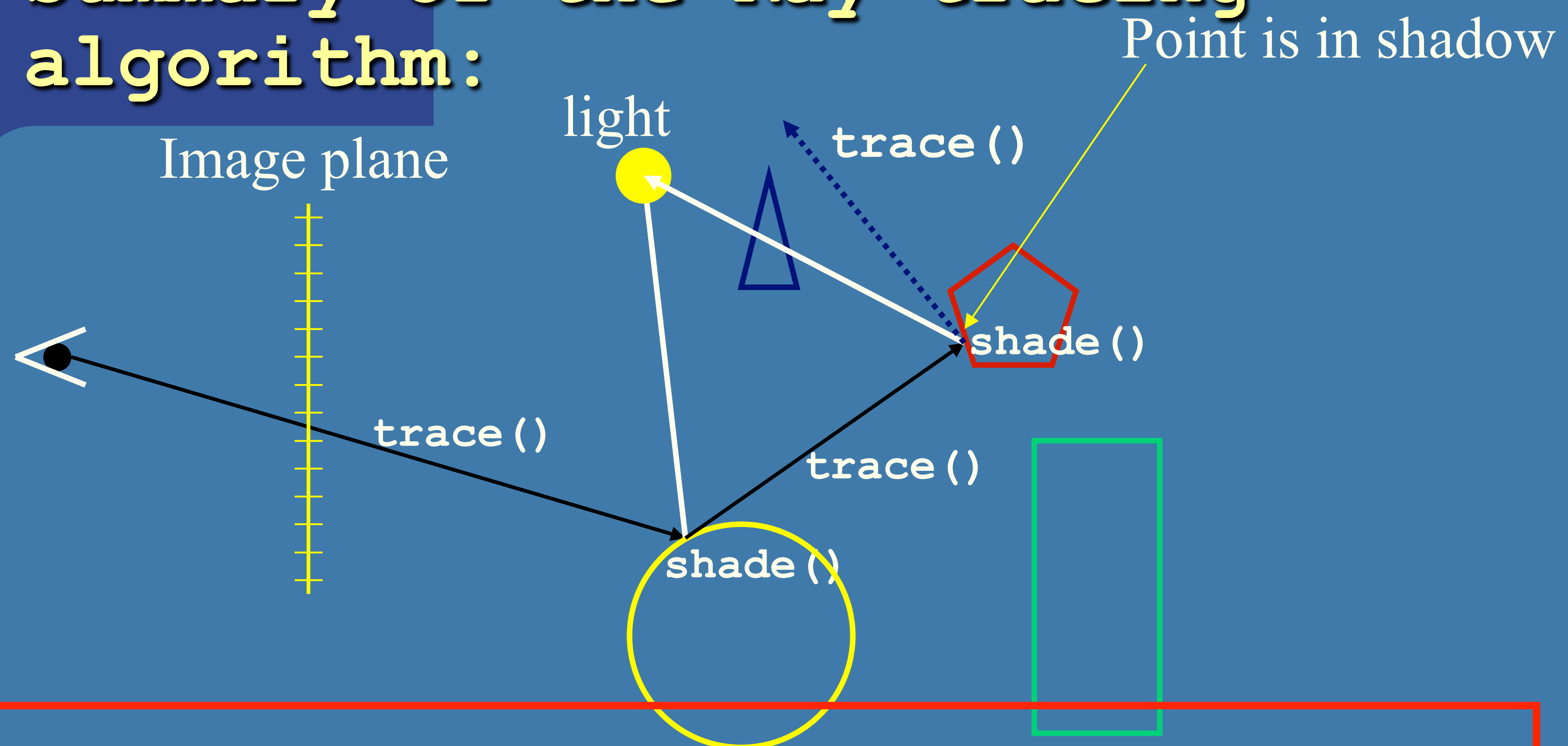


- Speedup techniques
  - Spatial data structures
    - Optimizations for BVHs – skippointer tree
    - BVH-traversal
    - (You do not need to learn the **ray traversal** algorithms for Grids nor AA-BSP trees)
  - Shadow cache



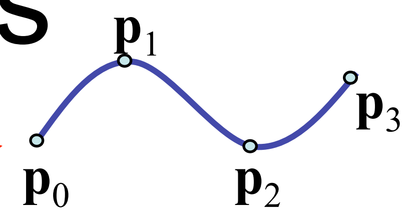
- Material (Fresnel: metall, dielectrics)
- Constructive Solid Geometry – how to implement

Summary of the Ray tracing-algorithm:

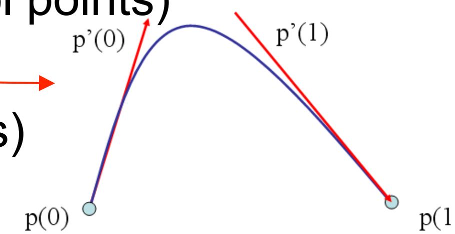


- **main()-calls trace() for each pixel**
- **trace(): should return color of closest hit point along ray.**
  1. **calls findClosestIntersection()**
  2. **If any object intersected → call shade().**
- **Shade(): should compute color at hit point**
  1. **For each light source, shoot shadow ray to determine if light source is visible**
     If not in shadow, compute diffuse + specular contribution.
  2. **Compute ambient contribution**
  3. **Call trace() recursively for the reflection- and refraction ray.**

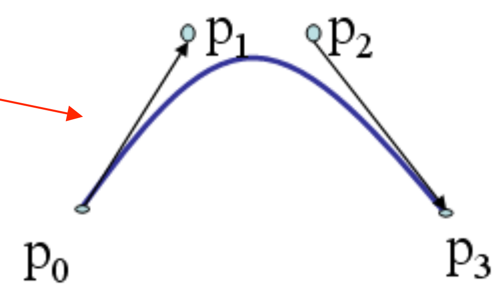
Types of Curves



- The types of curves:
  - Interpolating
    - Blending polynomials (or cubic parametric polynomials) for interpolation of 4 control points (fit curve to 4 control points)
  - Hermite
    - fit curve to 2 control points + 2 derivatives (tangents)



  - Bezier
    - 2 interpolating control points + 2 intermediate points to define the tangents



  - B-spline
    - To get C^2 continuity
  - NURBS
    - Different weights of the control points and
    - The control points can be at non-uniform intervalls

Goods and bads with these curves.

Splines and Basis

- If we examine the cubic B-spline from the perspective of each control (data) point, each interior point contributes (through the blending functions) to four segments
- We can rewrite p(u) in terms of the data points as

$$p(u) = \sum B_i(u)\, p_i$$

defining the basis functions $\{B_i(u)\}$

B-Splines

These are our control points, p_0-p_8, to which we want to approximate a curve

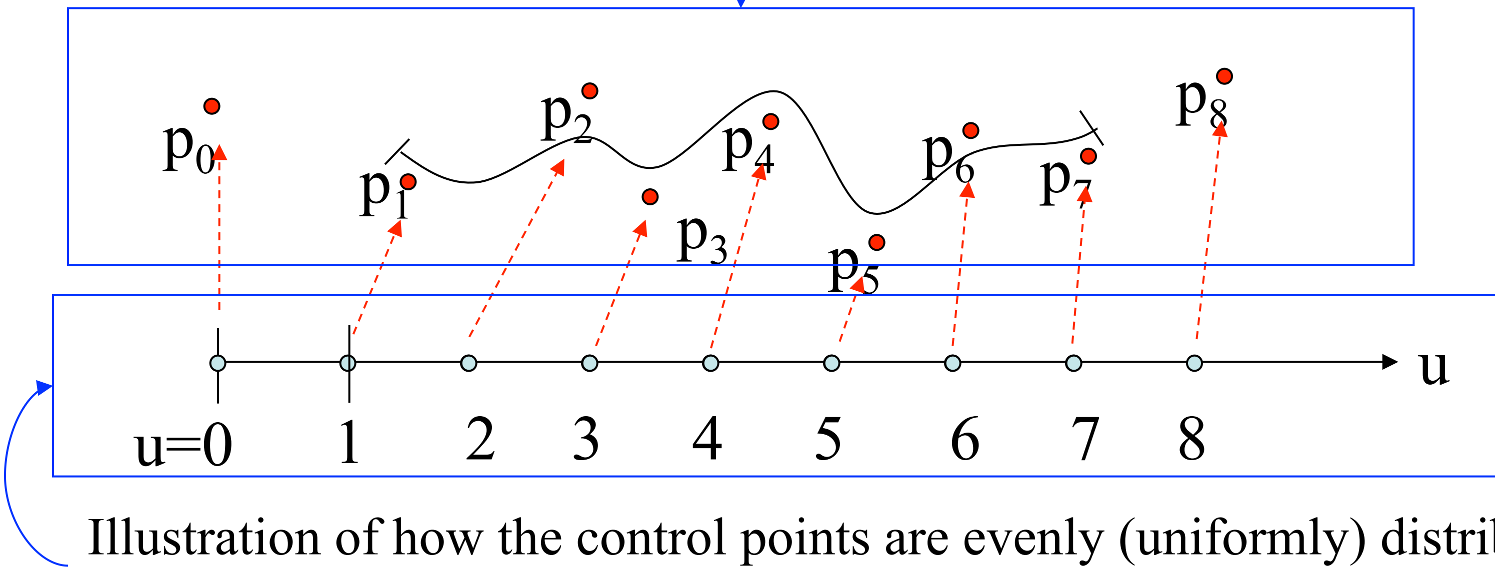
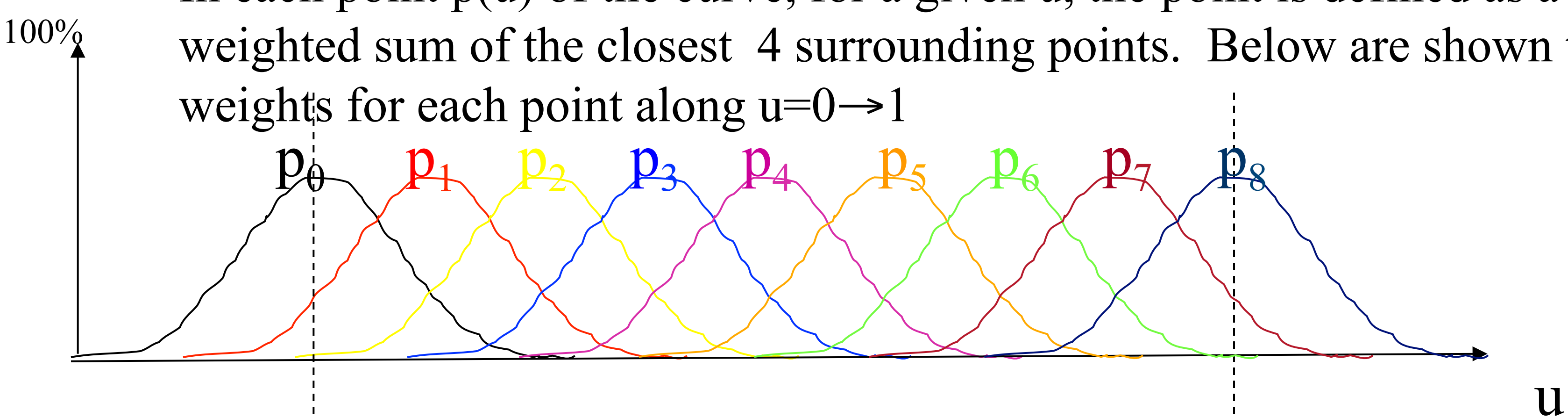


Illustration of how the control points are evenly (uniformly) distributed along the parameterisation u of the curve p(u).

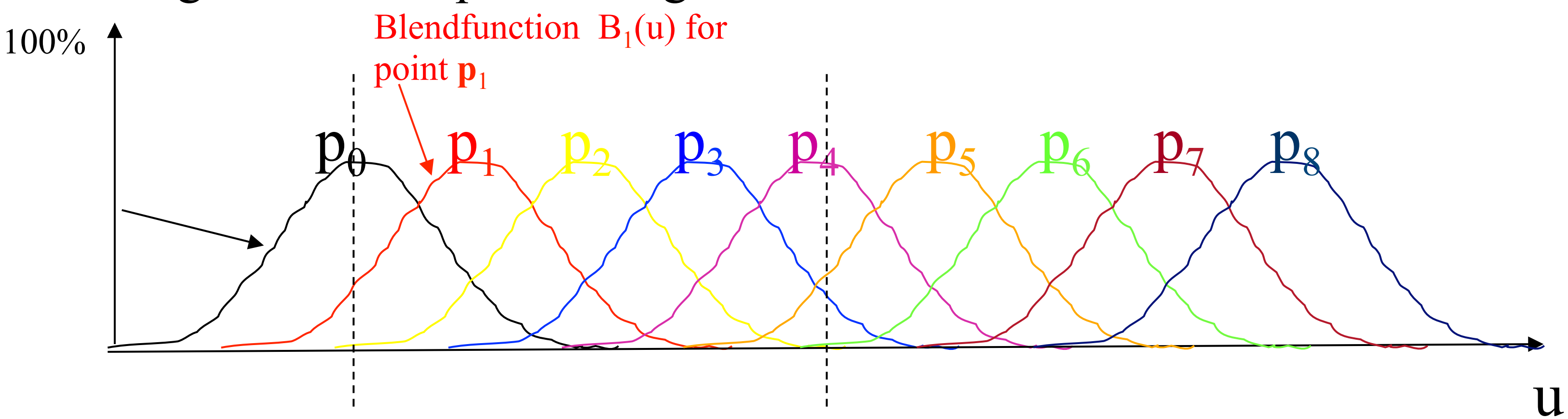
In each point p(u) of the curve, for a given u, the point is defined as a weighted sum of the closest 4 surrounding points. Below are shown the weights for each point along u=0→1



SUMMARY

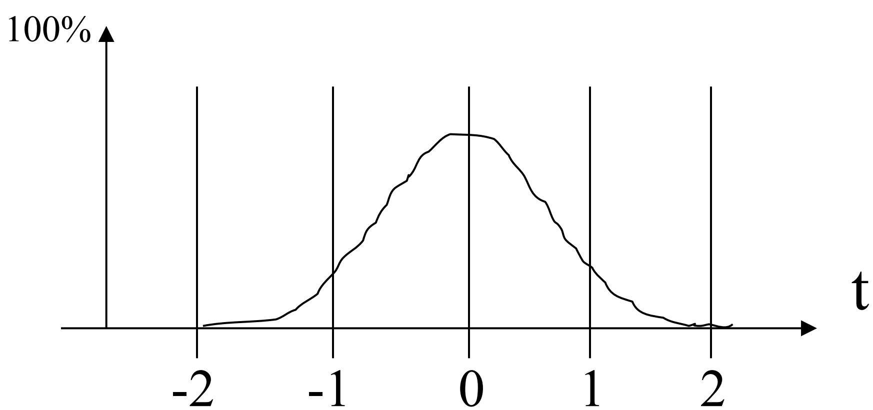
B-Splines

In each point p(u) of the curve, for a given u, the point is defined as a weighted sum of the closest 4 surrounding points. Below are shown the weights for each point along $u=0 \rightarrow 1$



The weight function (blend function) $B_{\mathbf{pi}}(u)$ for a point $\mathbf{p}_i$ can thus be written as a translation of a basis function B(t). $B_{\mathbf{pi}}(u) = B(u-i)$

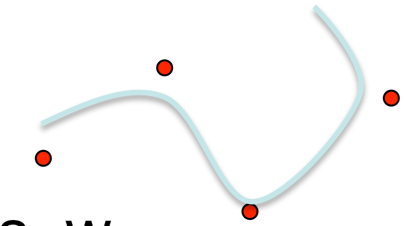
B(t):



Our complete B-spline curve p(u) can thus be written as:

$$p(u) = \sum B_i(u)\, p_i$$

NURBS



NURBS is similar to B-Splines except that:

1. The control points can have different weights, w_i, (heigher weight makes the curve go closer to that control point)
2. The control points do not have to be at uniform distances (u=0,1,2,3...) along the parameterisa-tion u. E.g.: u=0, 0.5, 0.9, 4, 14,…

NURBS = Non-Uniform Rational B-Splines

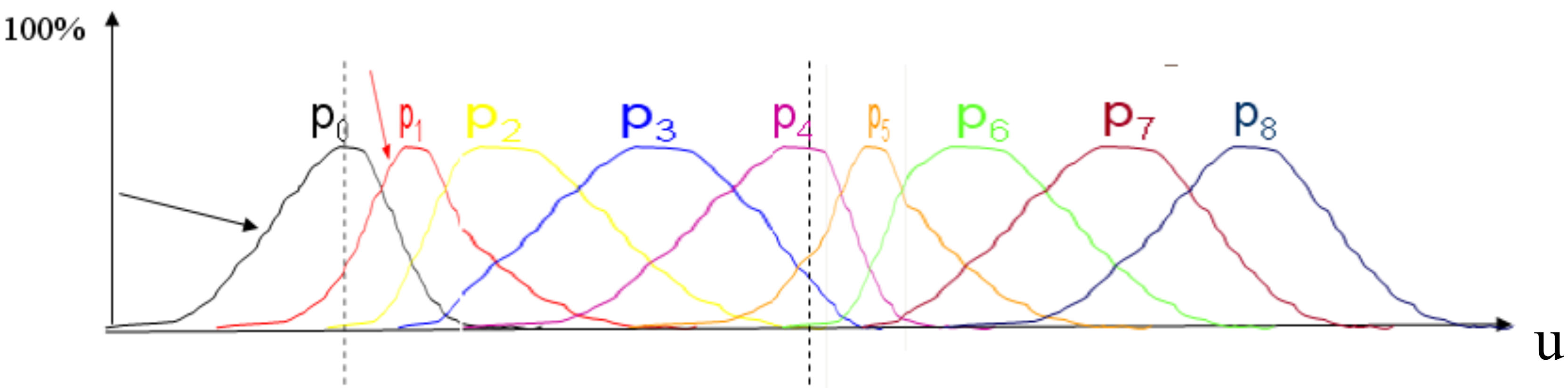
The NURBS-curve is thus defined as:

$$\mathbf{p}(u) = \frac{\sum_{i=0}^{n} B_i(u) w_i \mathbf{p}(i)}{\sum_{i=0}^{n} B_i(u) w_i}$$

Division with the sum of the weights, to make the combined weights sum up to 1, at each position along the curve. Otherwise, a translation of the curve is introduced (which is not desirable)

NURBS

- Allowing control points at non-uniform distances means that the basis functions $B_{pi}()$ are being streched and non-uniformly located.
- E.g.:



Each curve $B_{pi}()$ should of course look smooth and C^2 –continuous. But it is not so easy to draw smoothly by hand…(The sum of the weights are still =1 due to the division in previous slide)

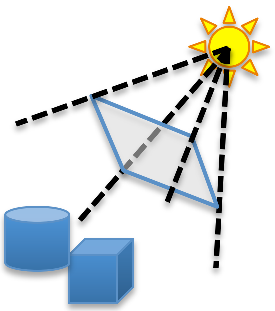
Lecture 11: Shadows + Reflection

- Point light / Area light
- Three ways of thinking about shadows
  - The basis for different algorithms.
- Shadow mapping
  - Be able to describe the algorithm
- Shadow volumes
  - Be able to describe the algorithm
  - Stencil buffer, 4-pass algorithm, Z-pass, Z-fail,
  - Creating quads from the silhouette edges as seen from the light source, etc
- Pros and cons of shadow volumes vs shadow maps
- Planar reflections – how to do. Why not using environment mapping?

Ways of thinking about shadows

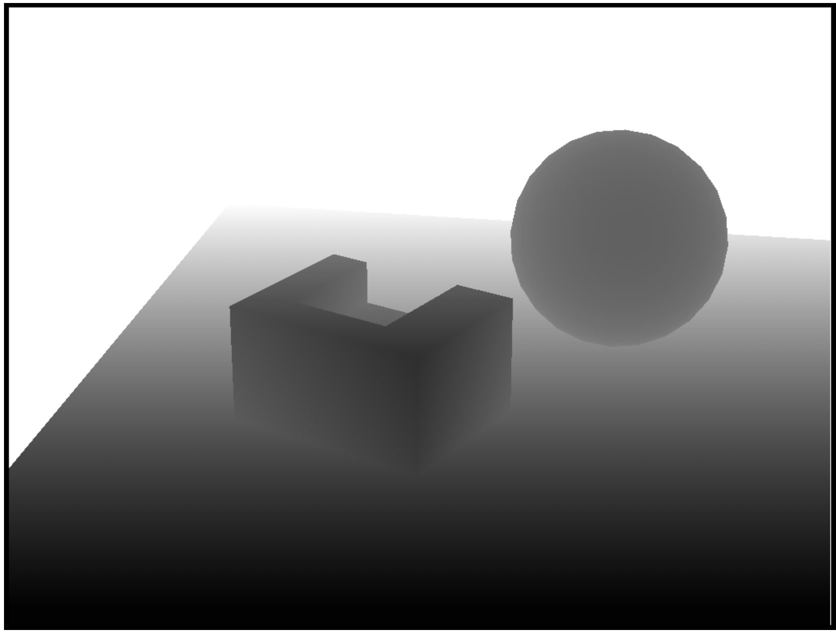
- As separate objects (like Peter Pan's shadow) **This corresponds to planar shadows**
- As volumes of space that are dark
  - **This corresponds to shadow volumes**
- As places not seen from a light source looking at the scene. **This corresponds to shadow maps**
- Note that we already "have shadows" for objects facing away from light

Shadow Maps - Summary



Shadow Map Algorithm:

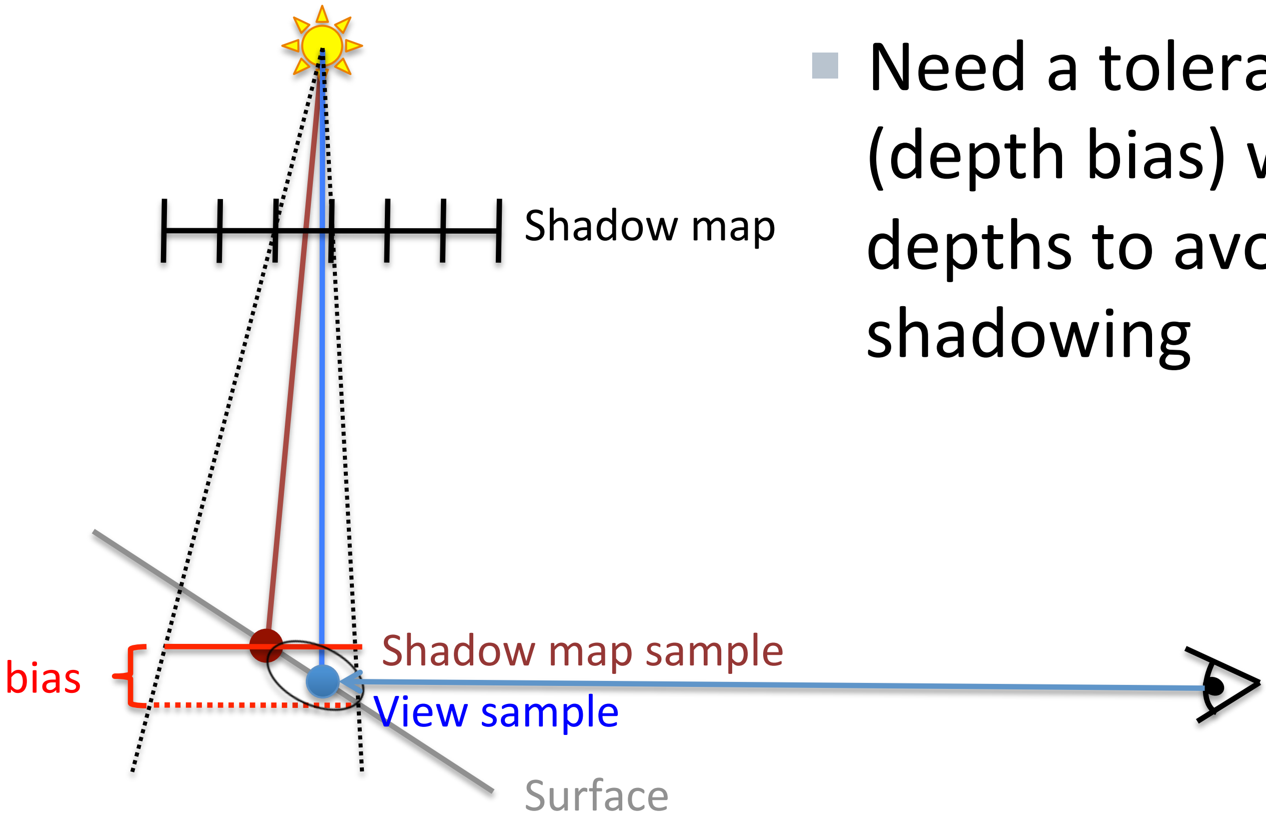
- Render a z-buffer from the light source
  - Represents geometry in light
- Render from camera
  - For every fragment:
    - Transform(warp) its 3D-pos (x,y,z) into shadow map (i.e. light space) and compare depth with the stored depth value in the shadow map
    - If depth greater-> point in shadow
    - Else -> point in light
    - Use a bias at the comparison



Shadow Map (=depth buffer)

Understand z-fighting and light leaks

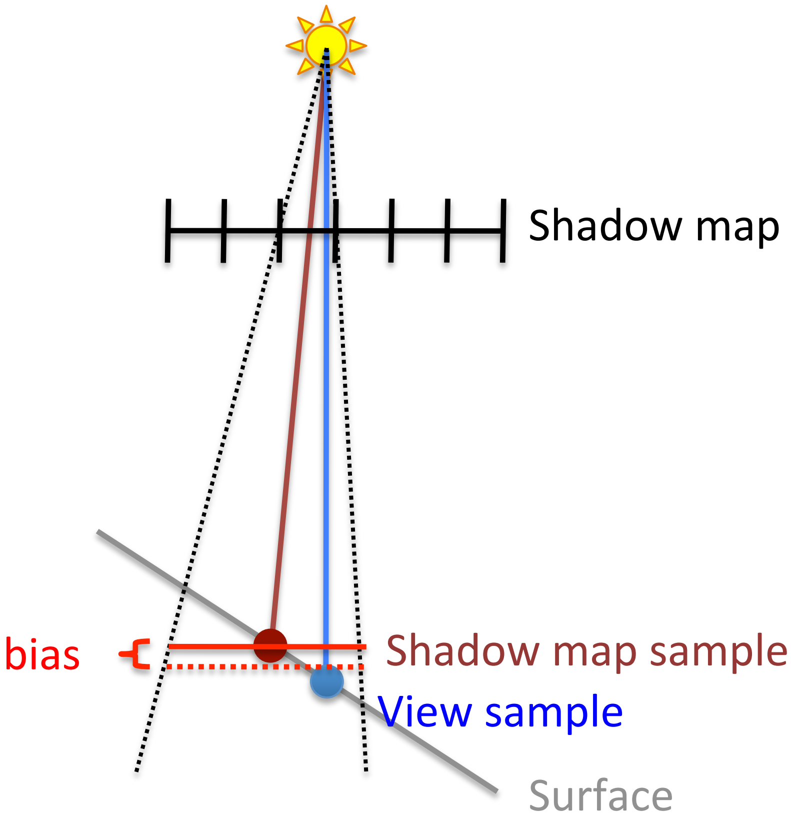
Bias

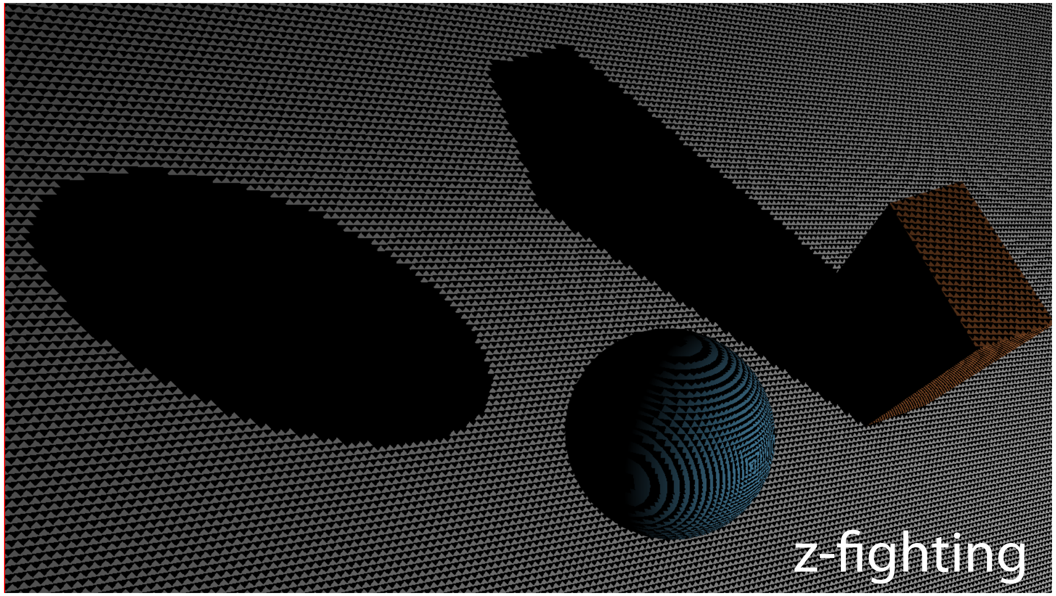


- Need a tolerance threshold (depth bias) when comparing depths to avoid surface self shadowing

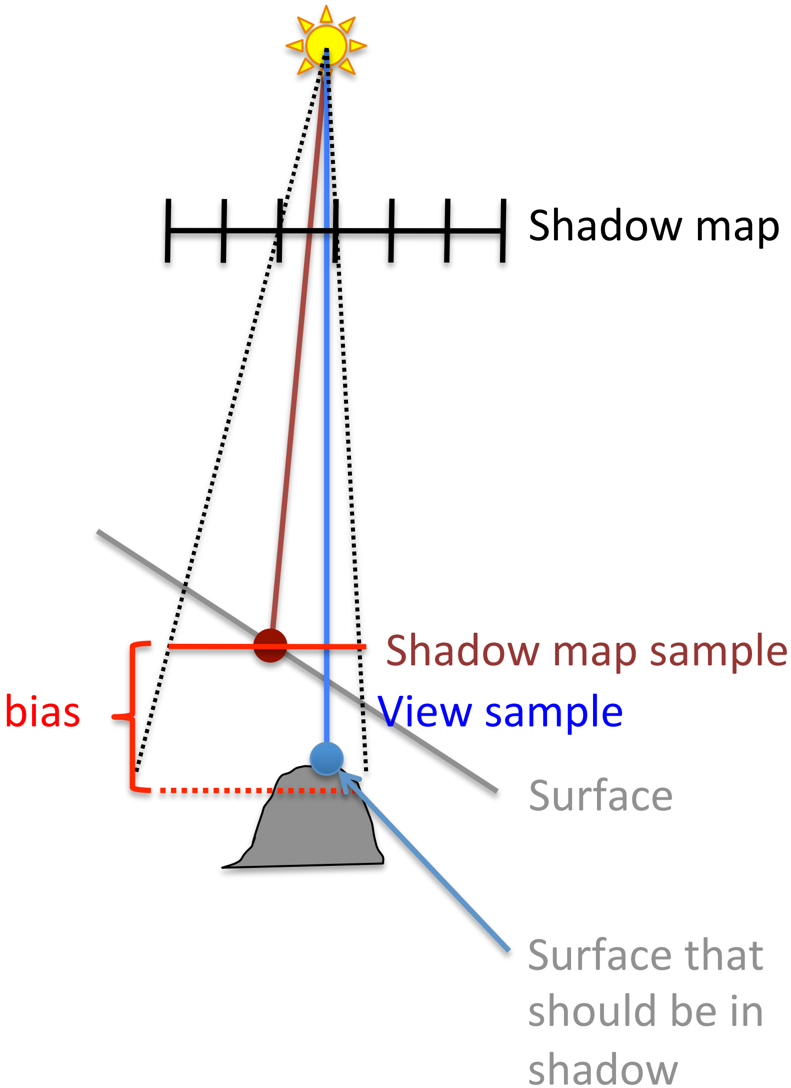
Bias

- Need a tolerance threshold (depth bias) when comparing depths to avoid surface self shadowing

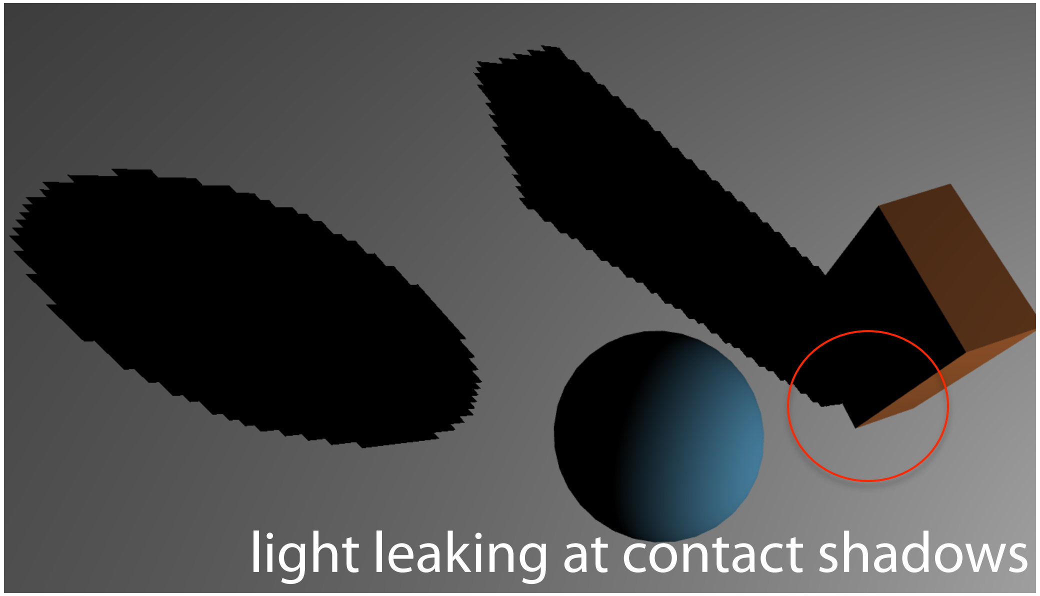




Bias

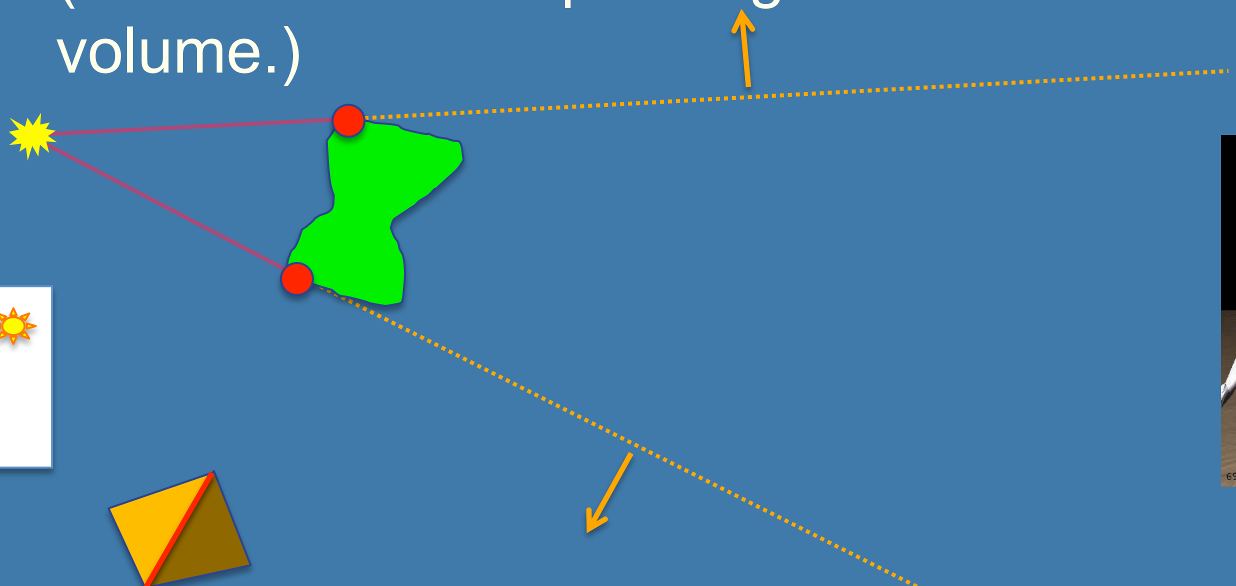


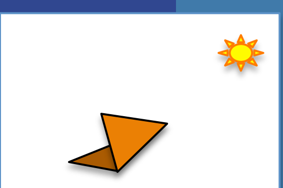
- Need a tolerance threshold (depth bias) when comparing depths to avoid surface self shadowing

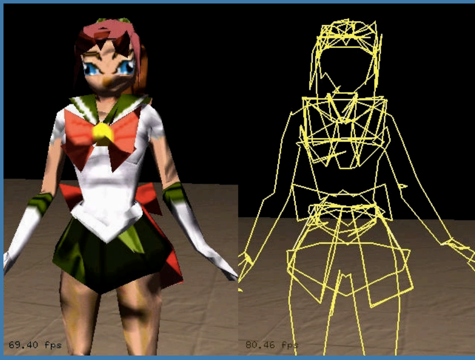


Shadow volumes

Create shadow quads for all silhouette edges (as seen from the light source).
(The normals are pointing outwards from the shadow volume.)







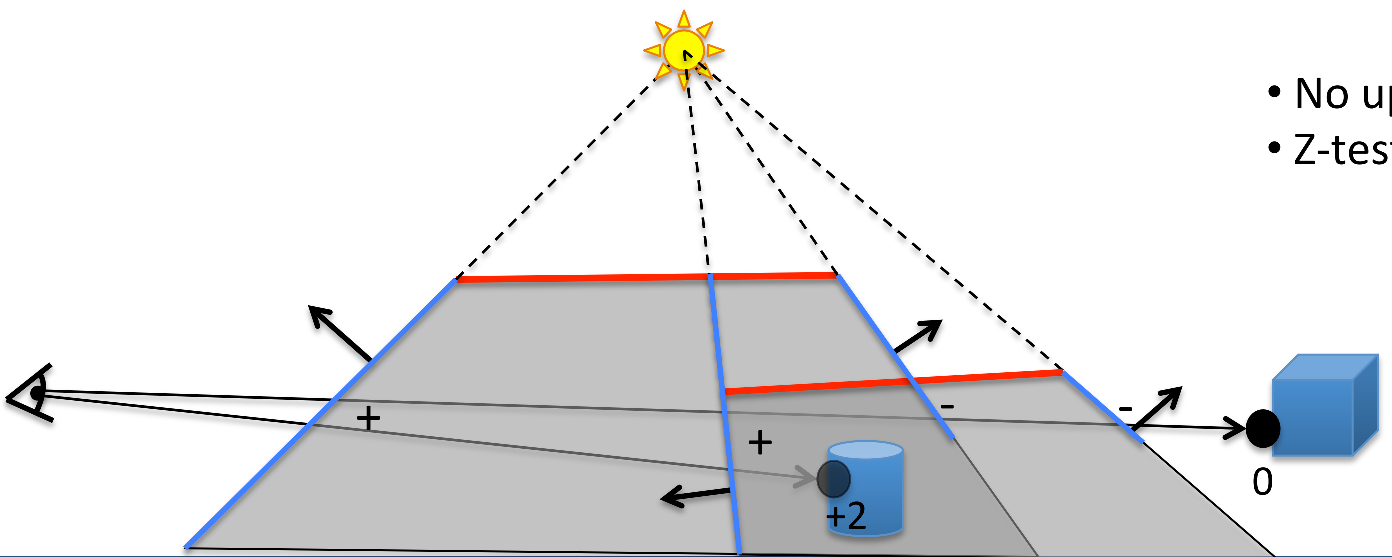
Edges between one triangle front facing the light source and one triangle back facing the light source are considered silhouette edges.

Then…

Example of silhouettes from light position

Shadow Volumes - concept

- Perform counting with the stencil buffer
  - Render front facing shadow quads to the stencil buffer
    - Inc stencil value, since those represents entering shadow volume
  - Render back facing shadow quads to the stencil buffer
    - Dec stencil value, since those represents exiting shadow volume



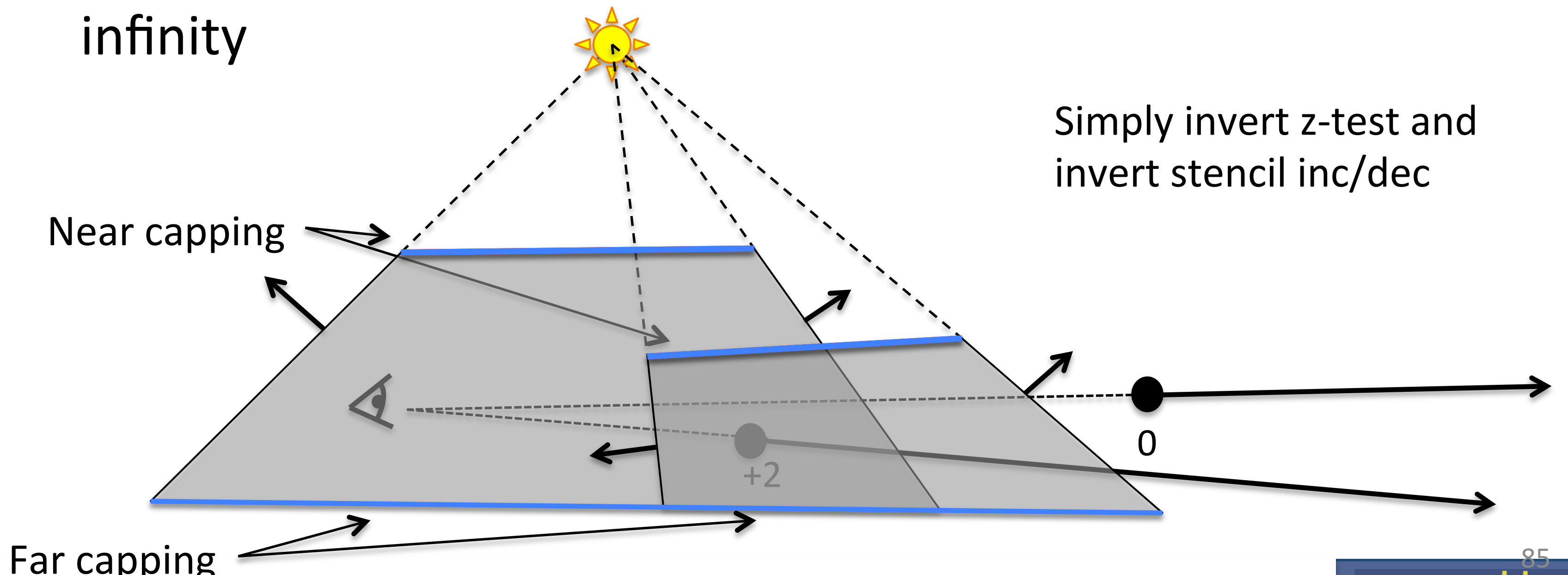
- No updating of z-buffer
- Z-test is enabled as usual

Shadow Volumes with the Stencil Buffer

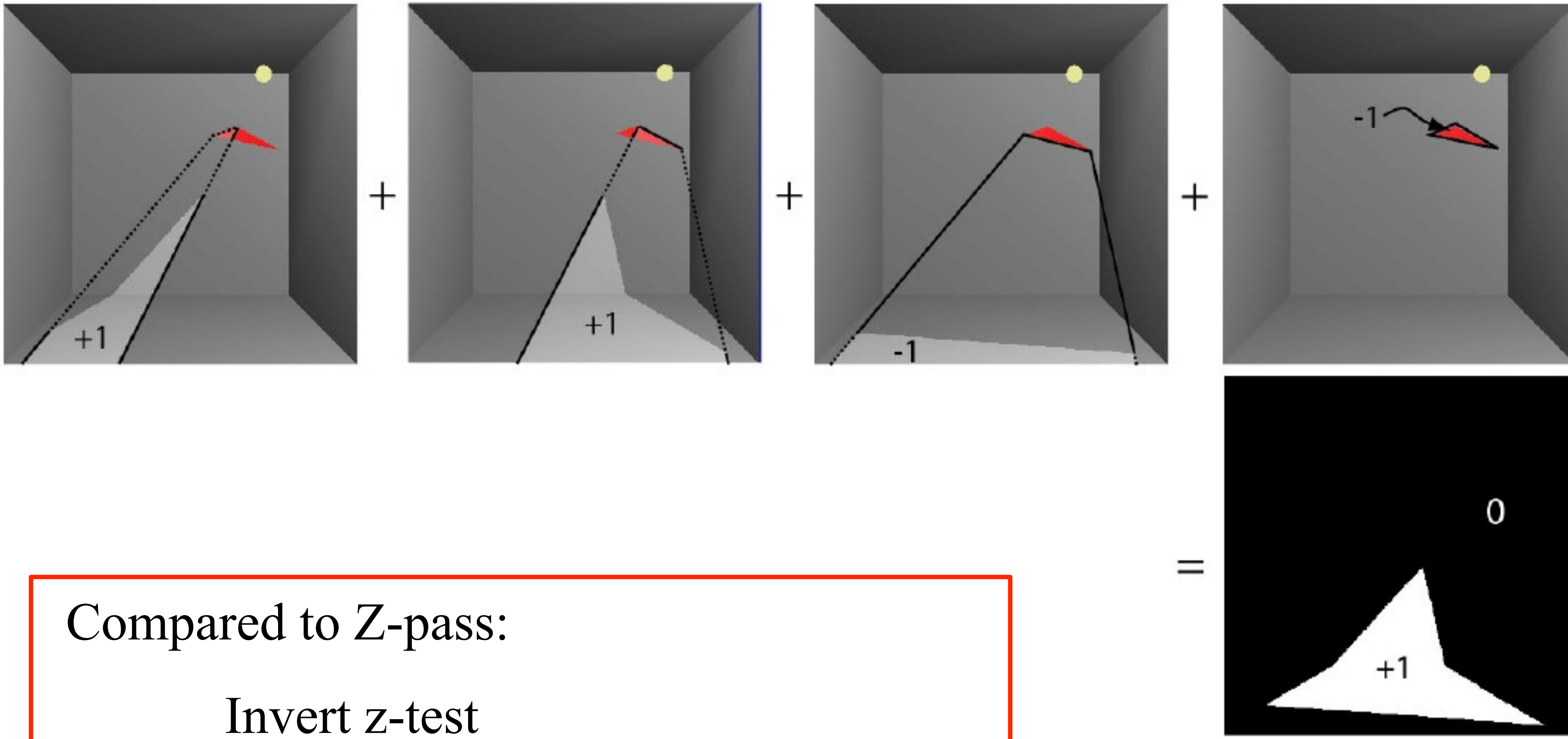
- A four pass process [Heidmann91]
  - **1st pass:** Render *ambient* lighting
  - Draw to stencil buffer only
    - Turn off updating of z-buffer and writing to color buffer but still use standard depth test
  - **2nd pass:**
    - Render *frontfacing* shadow volume quads: *incrementing* stencil buffer count
  - **3rd pass:**
    - Render *backfacing* shadow volume quads: *decrementing* stencil buffer count
  - **4th pass:** Render *diffuse and specular* where stencil buffer is 0.

The Z-fail Algorithm

- Z-pass must offset the stencil buffer with the number of shadow volumes that the eye is inside. Problematic.
- Count to infinity instead of to the eye
  - We can choose any reference location for the counting
  - A point in light avoids any offset
  - Infinity is always in light – if we cap the shadow volumes at infinity



Z-fail by example



Compared to Z-pass:

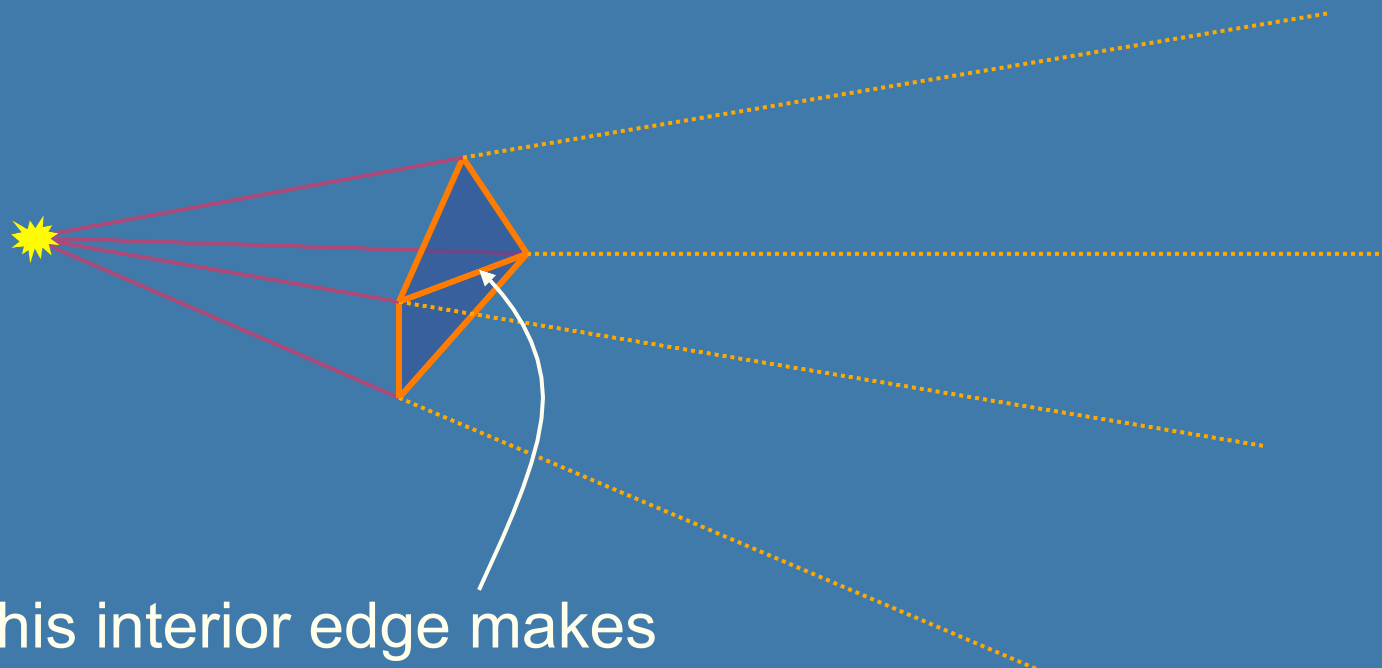
Invert z-test

Invert stencil inc/dec

I.e., count to infinity instead of from eye.

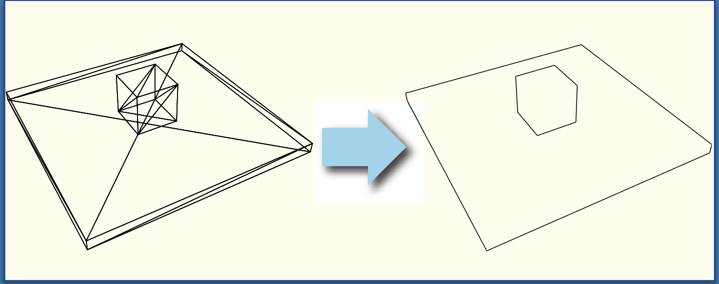
Merging Volumes

- Edge shared by two polygons facing the light creates front and backfacing quad.

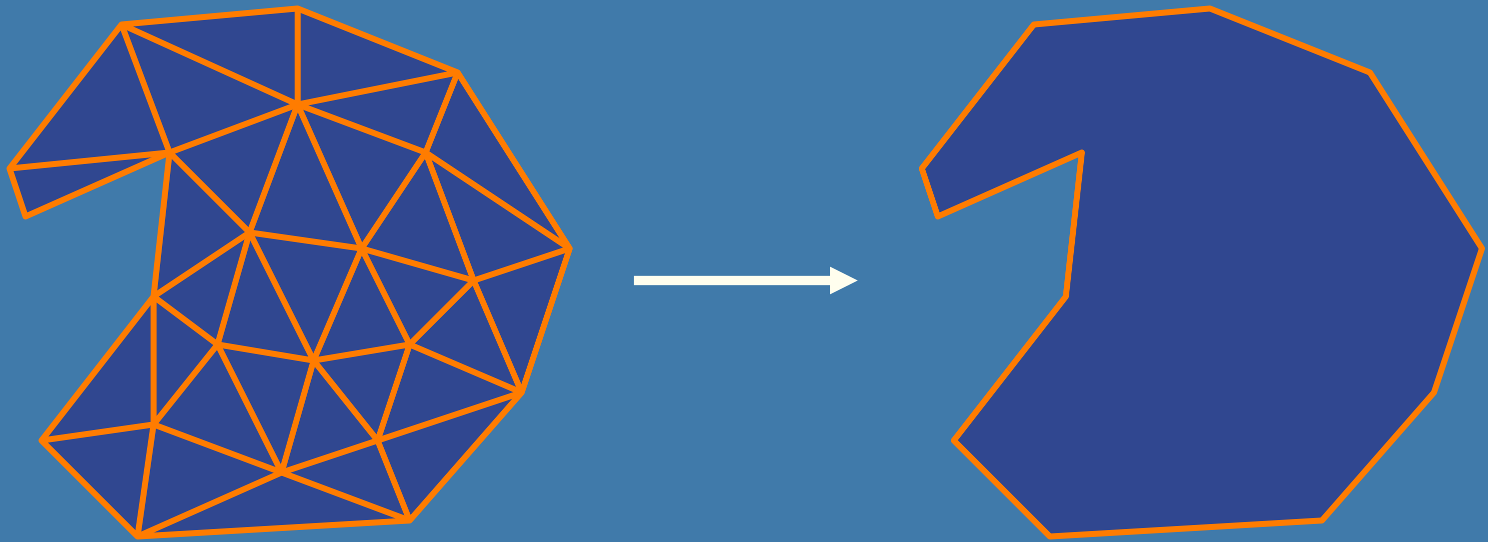


This interior edge makes two quads, which cancel out

Silhouette Edges

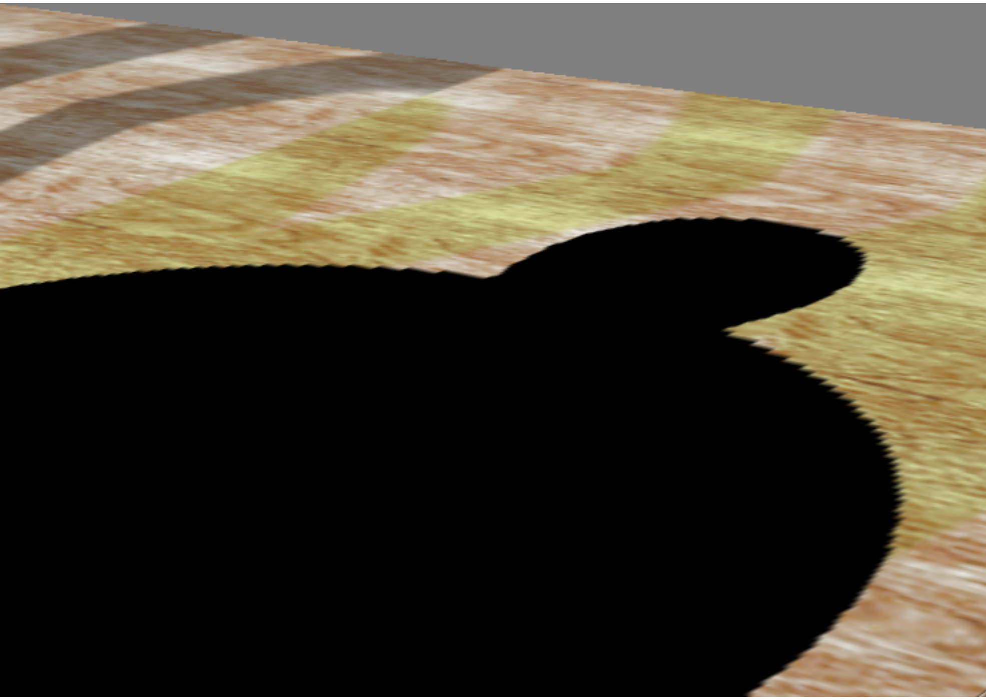


From the light's view, caster interior edges do not contribute to the shadow volume.



Finding the silhouette edge gets rid of many useless shadow volume polygons.

Shadow Maps vs Shadow Volumes



Shadow Maps

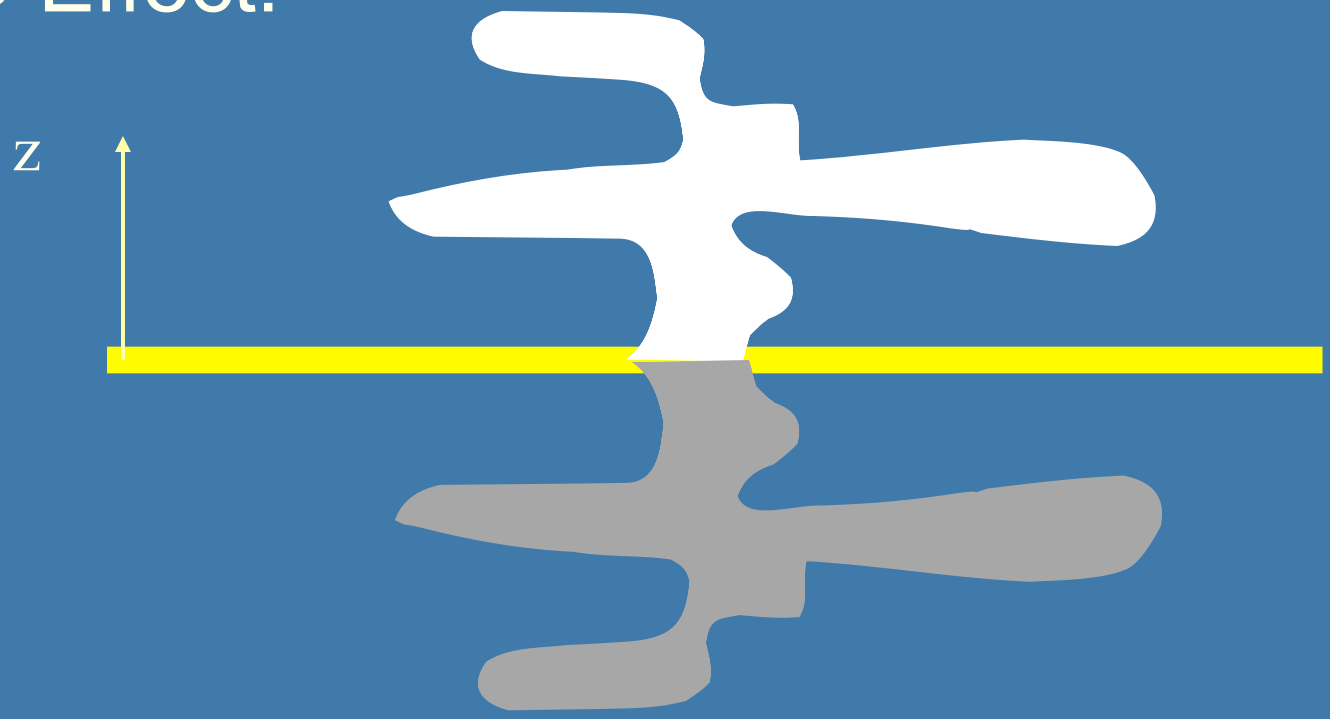
- *Good*: Handles any rasterizable geometry, **constant cost** regardless of complexity, map can sometimes be reused. **Very fast**.
- *Bad*: Frustum limited. **Jagged shadows** if res too low, **biasing** headaches.

Shadow Volumes

- *Good*: shadows are **sharp**. Handles omni-directional lights.
- *Bad*: **3 or 4 passes**, shadow polygons must be generated and rendered → lots of polygons & **fill**

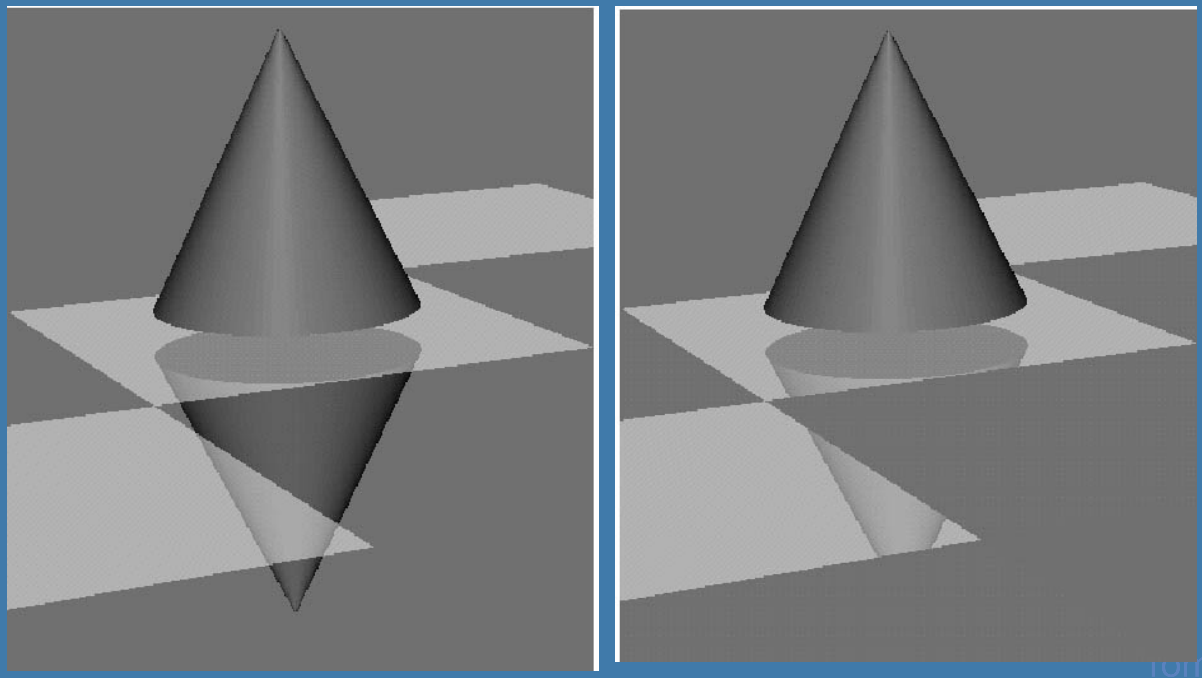
Planar reflections

- Assume plane is z=0
- Then apply `glScalef(1,1,-1);`
- Effect:

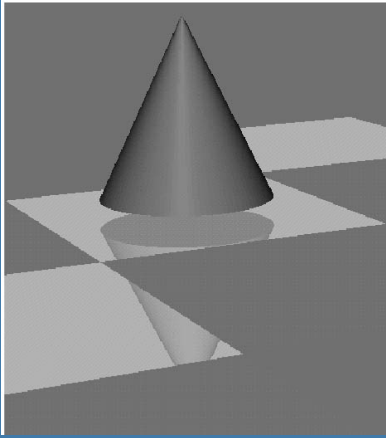


Planar reflections

- Backfacing becomes front facing!
- Lights should be reflected as well
- Need to clip (using stencil buffer)
- See example on clipping:



Planar reflections



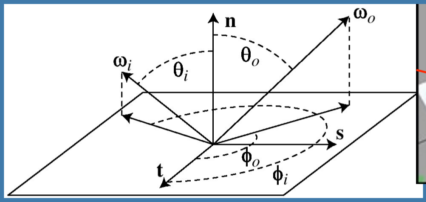
- How should you render?
- 1) the ground plan polygon into the stencil buffer
- 2) the scaled (1,1,-1) model, but mask with stencil buffer
  - Reflect light pos as well
  - Use front face culling
- 3) the ground plane (semi-transparent)
- 4) the unscaled model

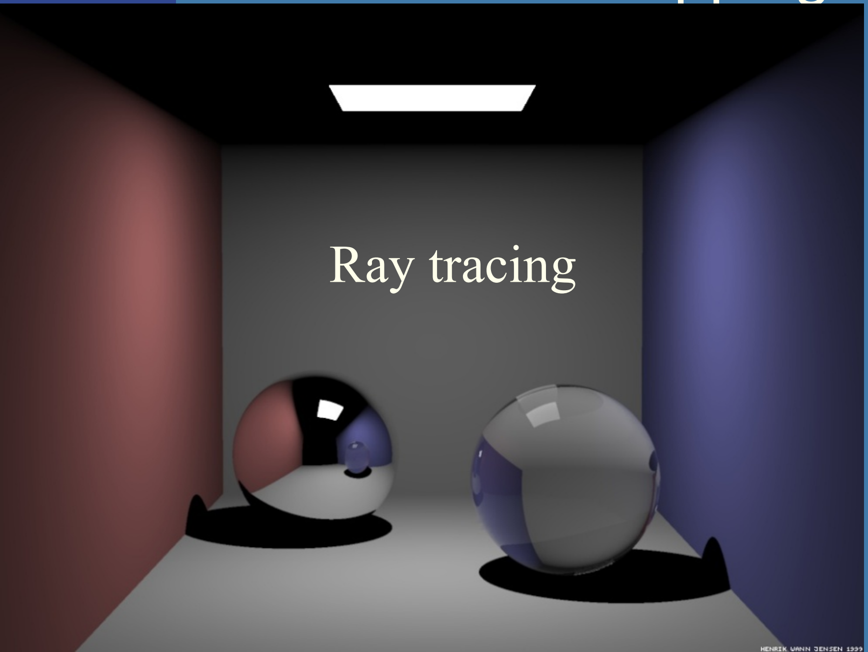
Lecture 12 – Global Illumination

- Global illumination:

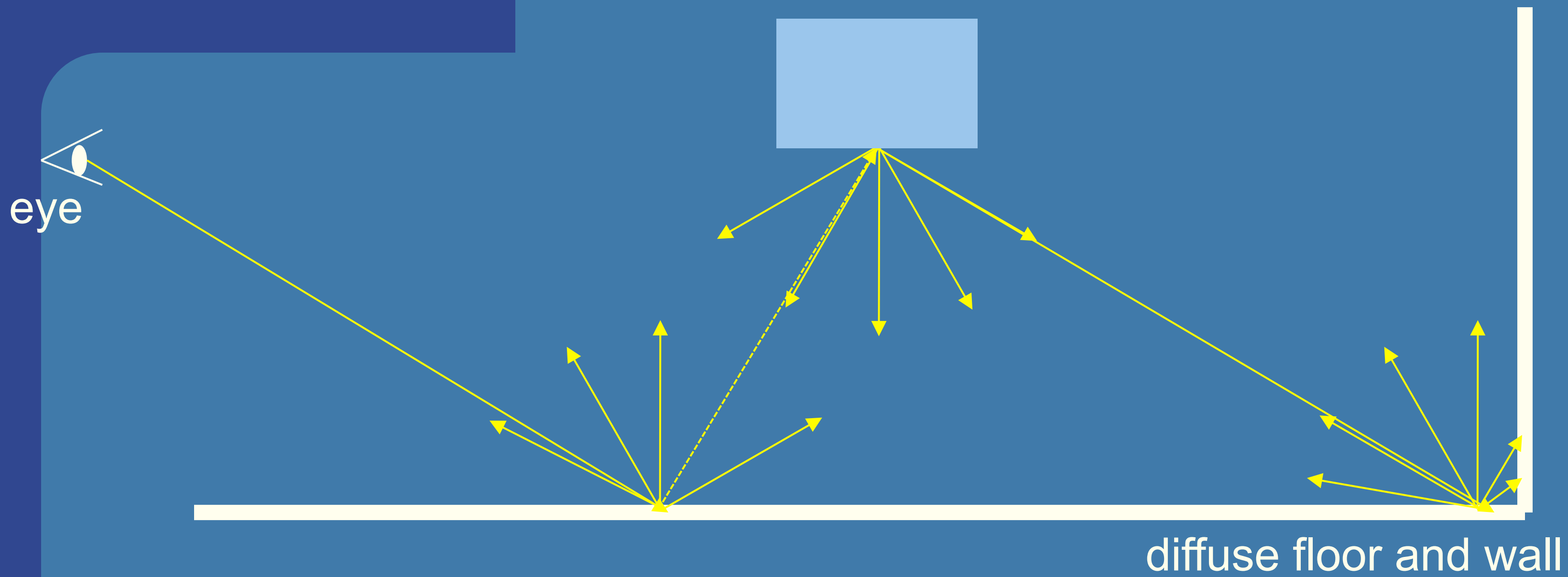
$$L_o = L_e + \int_{\Omega} f_r(\mathbf{x}, \omega, \omega')L_i(\mathbf{x}, \omega')(\omega' \cdot \mathbf{n})d\omega'$$

  - Why is not standard ray tracing enough?
  - rendering eq., BRDFs
  - Monte Carlo Ray Tracing
  - Path tracing
  - Photon mapping





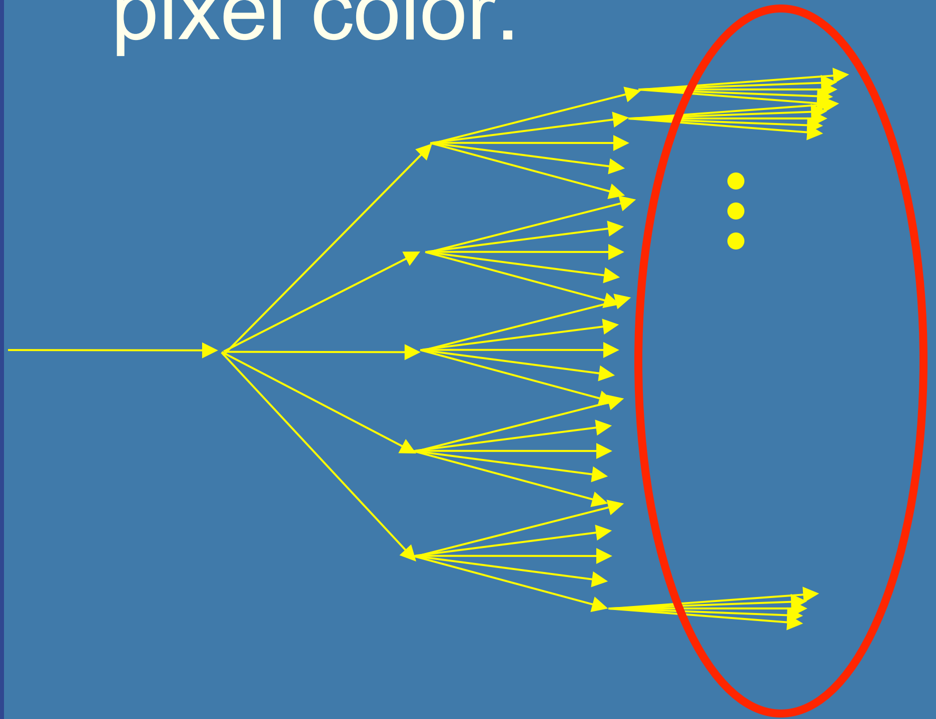
Monte Carlo Ray Tracing



- Sample indirect illumination by shooting sample rays over the hemisphere, at each hit.
  - At some recursion depth, stop and compute standard local lighting (i.e., without indirect illumination)

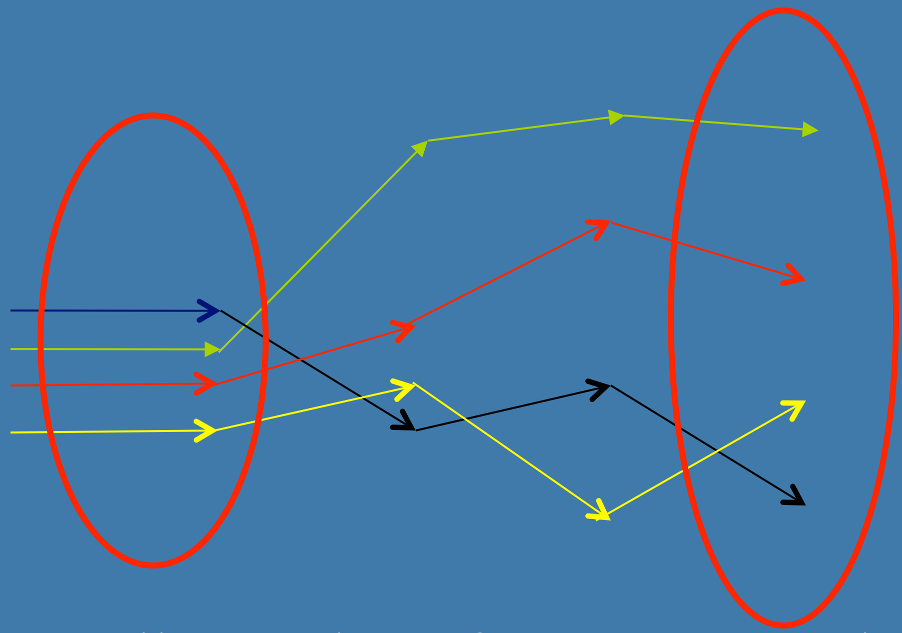
Monte Carlo Ray Tracing

- This gives a ray tree with most rays at the bottom level. This is bad since these rays have the lowest influence on the pixel color.



PathTracing

- Path Tracing instead only traces one of the possible ray paths at a time. This is done by randomly selecting only one sample direction at a bounce. Hundreds of paths per pixel are traced.



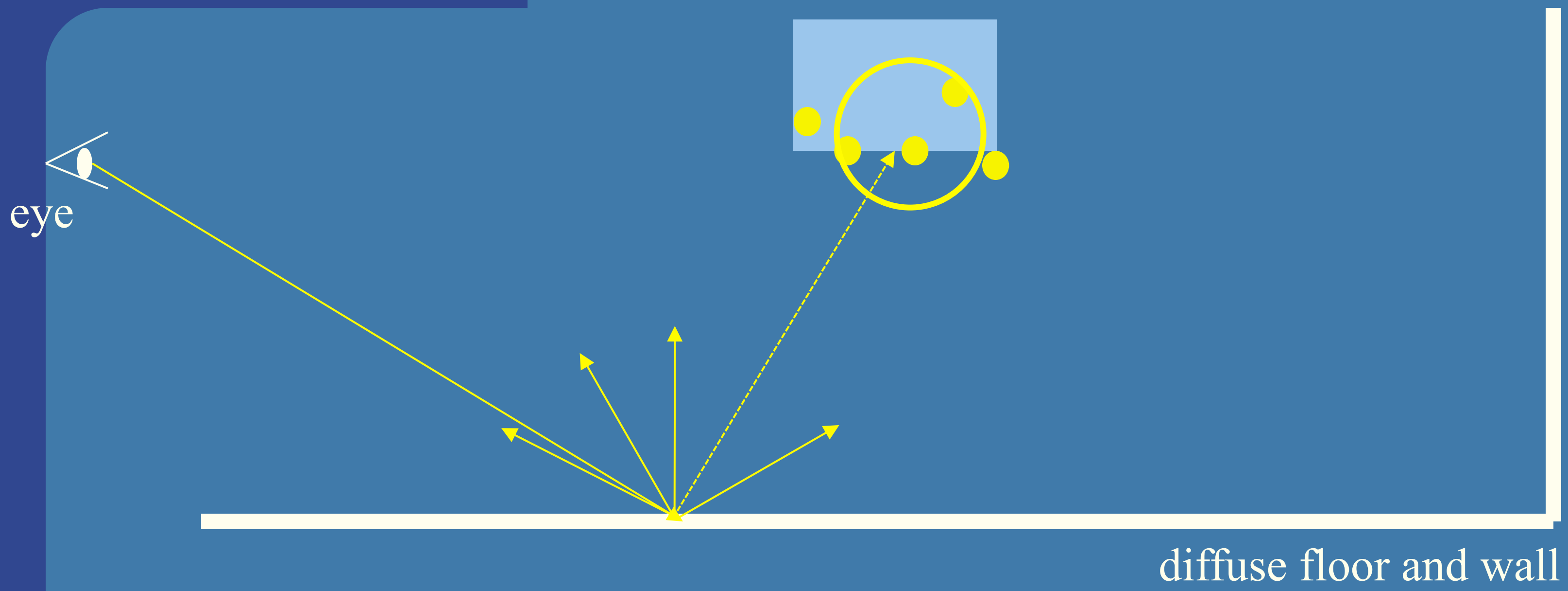
Equally number of rays are traced at each level

Photon Mapping

- Creating Photon Maps: Trace photons (~100K-1M) from light source. Store them in kd-tree when they hit diffuse surface. Then, use russian roulette to decide if the photon should be absorbed or specularly or diffusively reflected. Create both global map and caustics map. Caustics map sends photons from light only in directions of reflective/refractive objects.
- Ray trace from eye: At an intersection point **p**, compute direct illumination and shoot reflection/refraction rays. Also, grow sphere around **p** in caustics map to get caustics contribution. Also sample indirect slow varying light around **p** by sampling the hemisphere with ~1000 rays (=final gathering) and use the global photon map where those rays hit a surface.

- Growing sphere: Uses the kd-tree to expand a sphere around **p** until a fixed amount (e.g. 50) photons are inside the sphere. The radius is an inverse measure of the intensity of indirect light at **p**. The BRDF at **p** could also be used to get a more accurate color and intensity value by considering the photons incoming directions.

A modification for indirect Illumination – Final Gather



- Too noicy to use the global map for direct rays
- Use global map for indirect rays (shoot 100-1000 indirect rays per pixel)

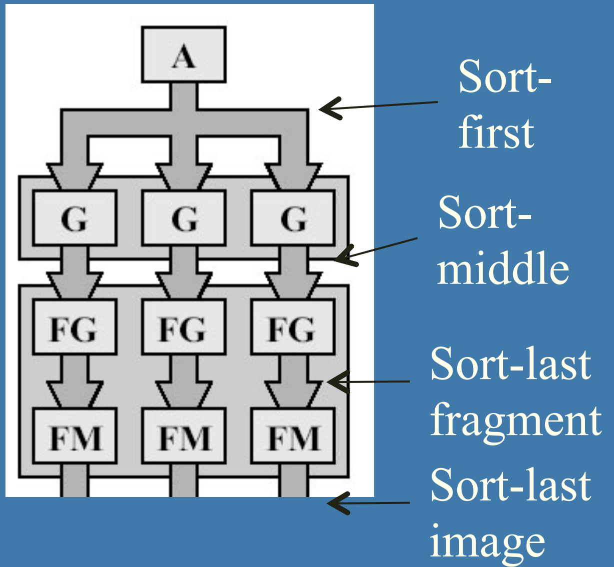
Lecture 13:

Linearly interpolate $(u_i/w_i, v_i/w_i, 1/w_i)$ in screenspace from each triangle vertex i.
Then at each pixel:

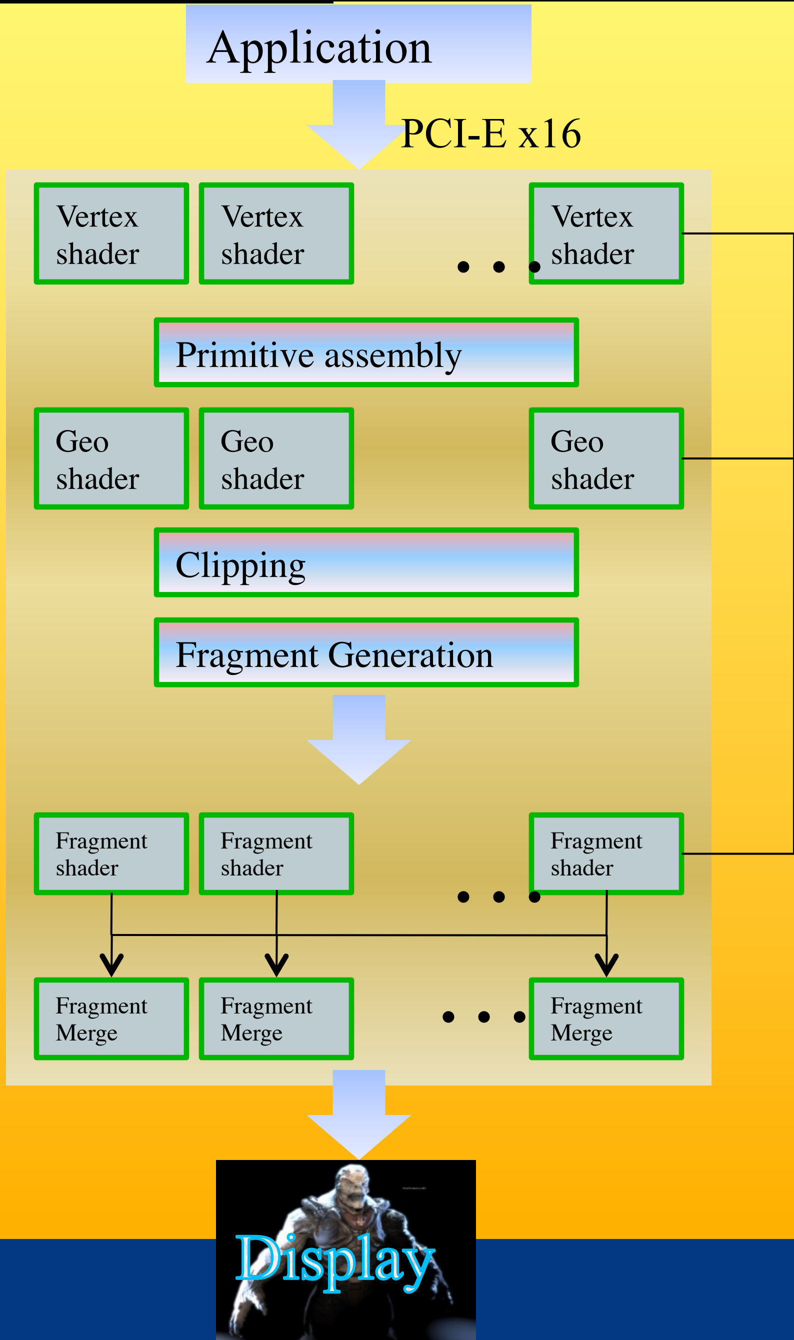
$$u_{ip} = (u_{ip}/w_{ip}) / (1/w_{ip})$$
$$v_{ip} = (v_{ip}/w_{ip}) / (1/w_{ip})$$

where ip = screen-space interpolated value from the triangle vertices.

- Perspective correct interpolation (e.g. for textures)
- Taxonomy:
  - Sort first
  - sort middle
  - sort last fragment
  - sort last image



- Bandwidth
  - Why it is a problem
  - How to ”solve” it
- Be able to sketch the architecture of a moder graphics card



On NVIDIA 8000-series:
Vertex-, Geometry- and Fragment shaders allocated from a pool of 128 processors