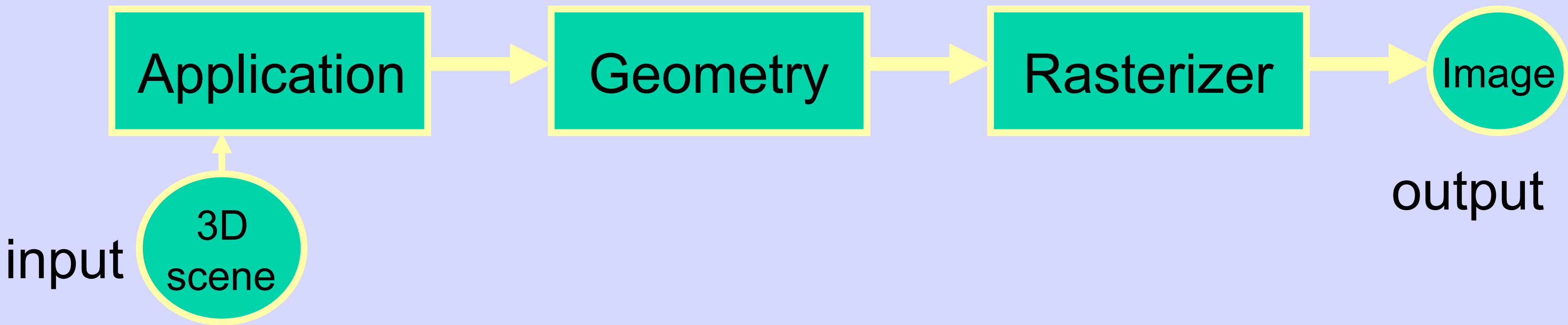
TDA361 - Computer Graphics

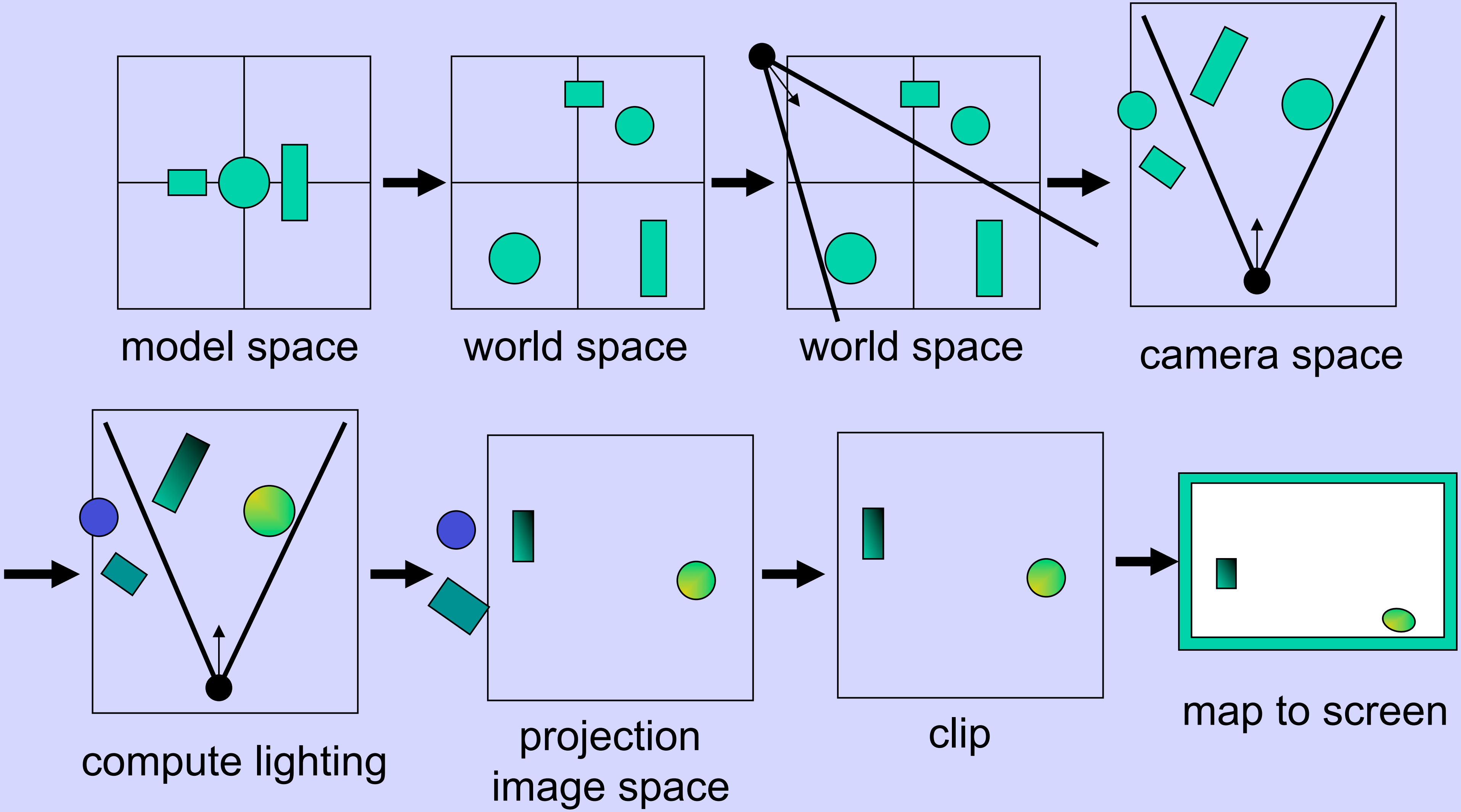
Ulf Assarsson
Department of Computer Engineering
Chalmers University of Technology

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
    - Compute lighting at vertices of triangle
    - Project onto screen (3D to 2D)
  - Rasterizer
    - Texturing
    - Interpolation over triangle
    - Z-buffering



GEOMETRY - Summary



Lecture 2: Transforms

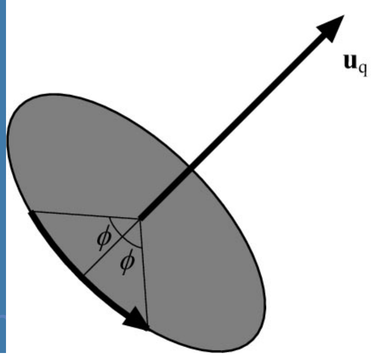
- Cannot use same matrix to transform normals

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

- 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 $\mathbf{p}$



Homogeneous notation

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

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

rotations-
delen

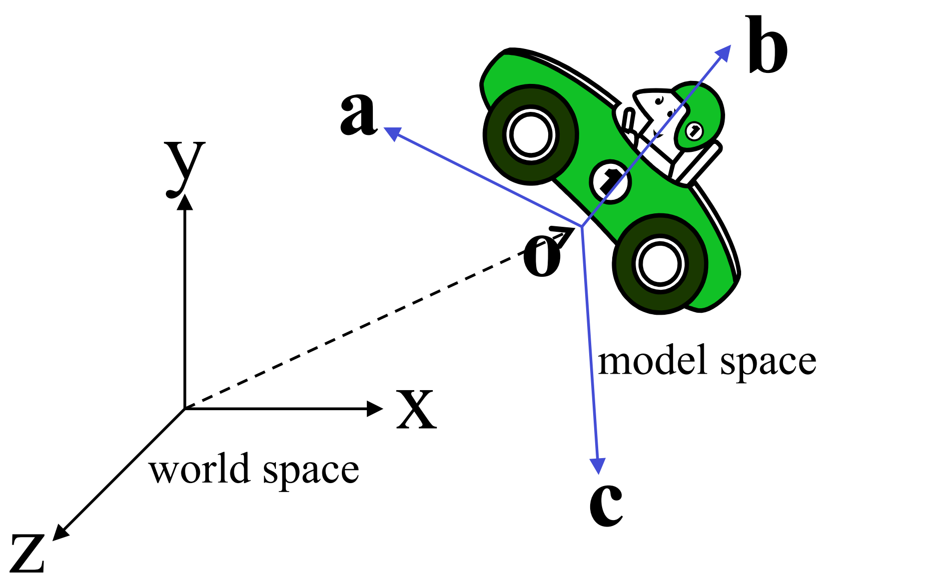
Translationsdelen

- 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)

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

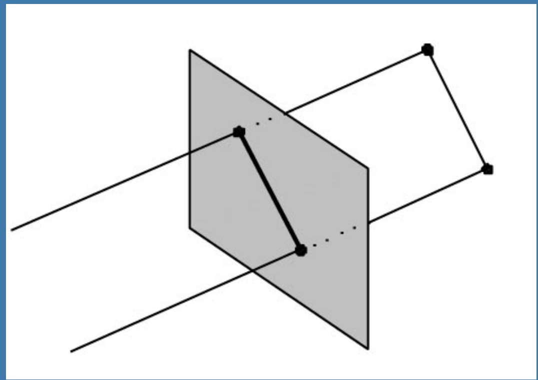


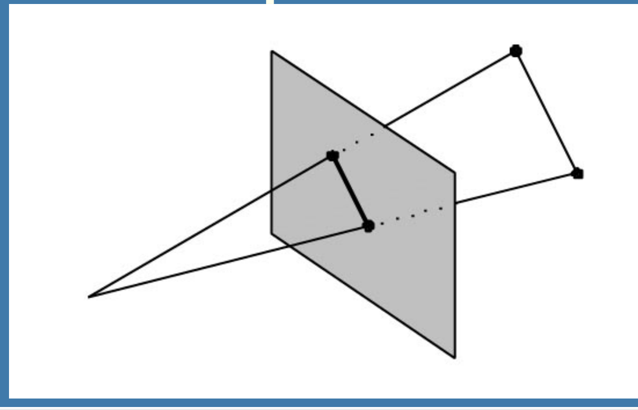
$$M_{\text{model-to-world}} = \begin{bmatrix} a_x & b_x & c_x & o_x \\ a_y & b_y & c_y & o_y \\ a_z & b_z & c_z & o_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

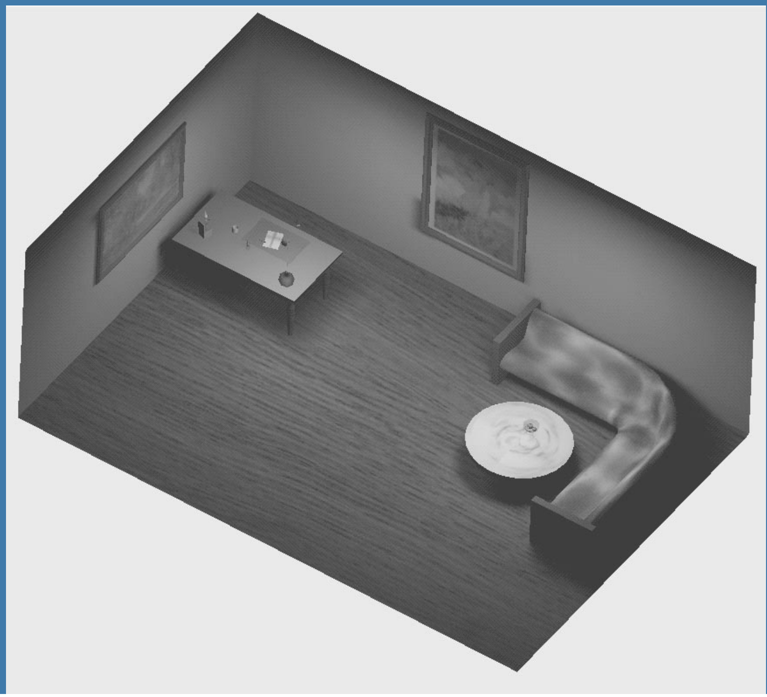
E.g.: $\mathbf{p}_{\text{world}} = M_{m\to w}\,\mathbf{p}_{\text{model}} = M_{m\to w}\,(0,5,0)^T = 5\,\mathbf{b}\;\;(+\,\mathbf{o})$

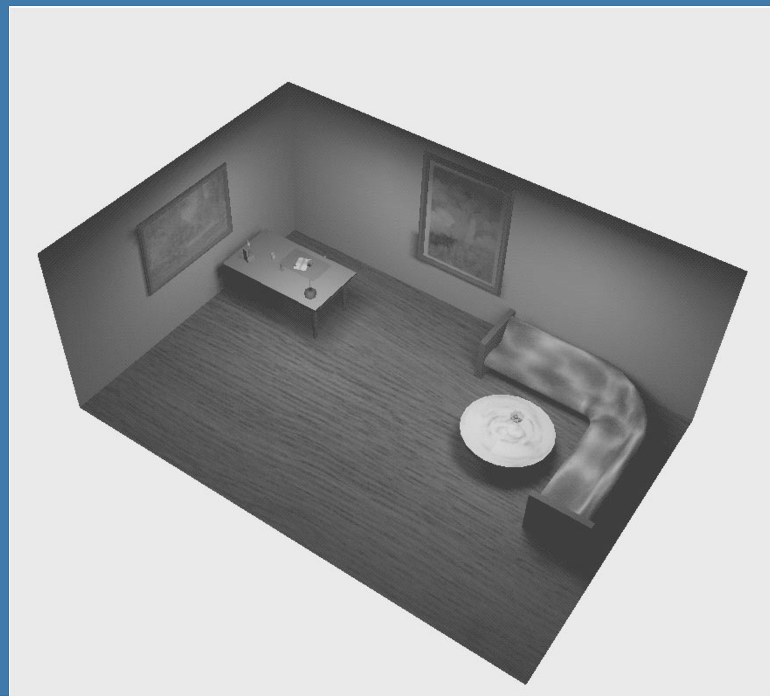
Projections

- Orthogonal (parallel) and Perspective

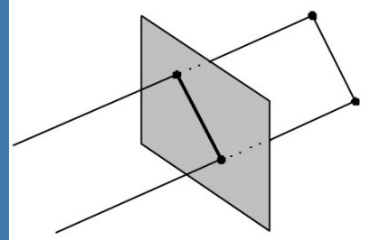






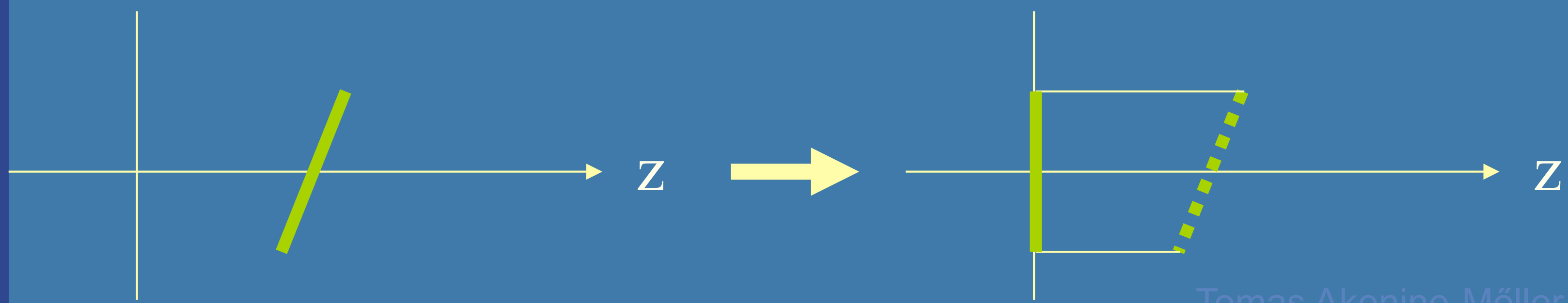


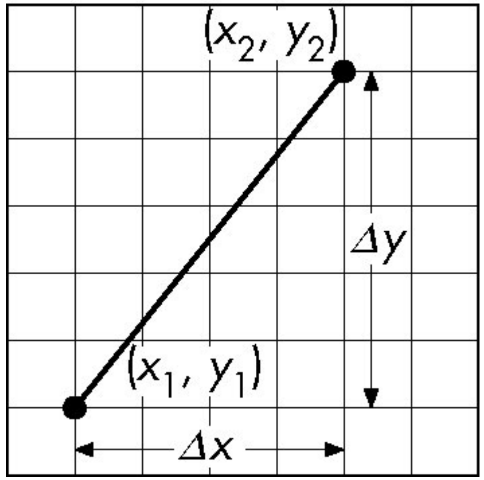
Orthogonal projection



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

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





DDA Algorithm

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

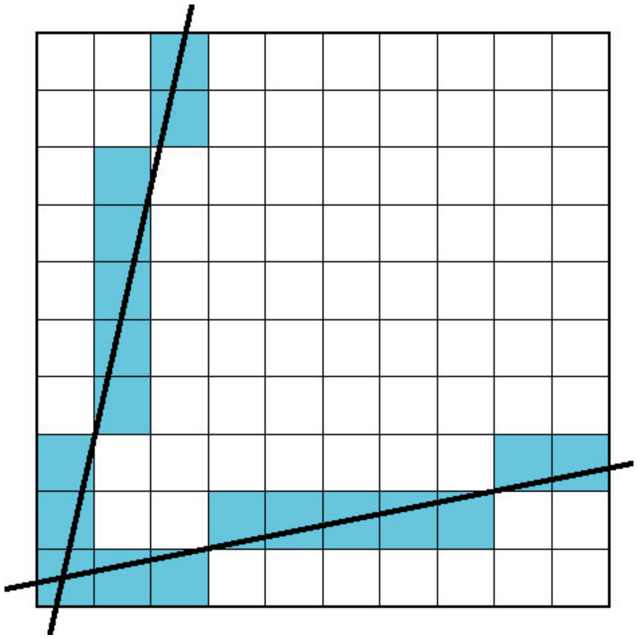
$$dy/dx = k = \Delta y/\Delta x = y_2 - y_1/x_2 - x_1$$

- 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 $1 \geq k \geq 0$
- For $k > 1$, swap role of x and y
  - For each y, plot closest x



02. Rasterization, Depth Sorting and Culling:

Very Important!

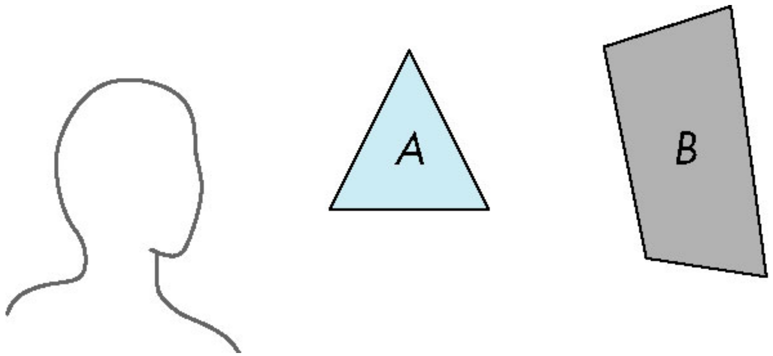
- The problem with DDA is that it uses floats which was slow in the old days
- Bresenhams algorithm only uses integers

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

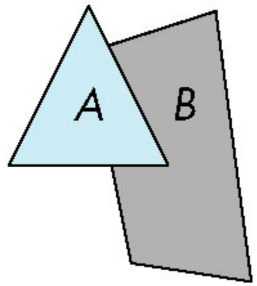
Said on the lecture:

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

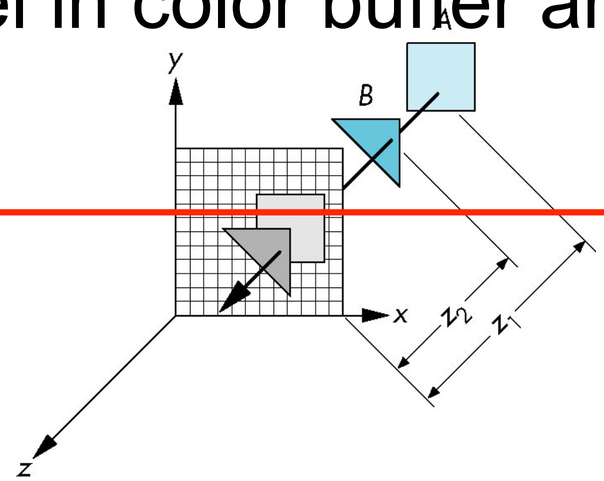
- Requires ordering of polygons first
  - O(n log n) calculation for ordering
  - Not every polygon is either in front or behind all other polygons

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

Said on the lecture:

z-Buffer Algorithm

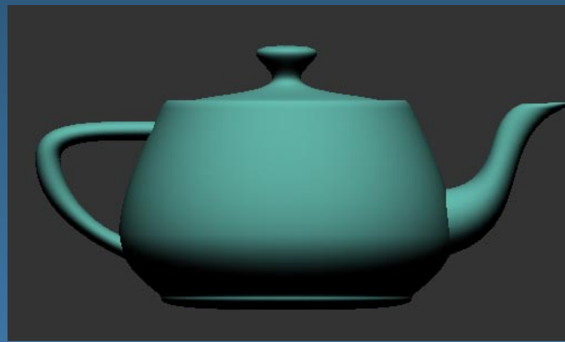
- Use a buffer called the z or depth buffer to store the depth of the closest object at each pixel found so far
- As we render each polygon, compare the depth of each pixel to depth in z buffer
- If less, place shade of pixel in color buffer and update z buffer



Lighting

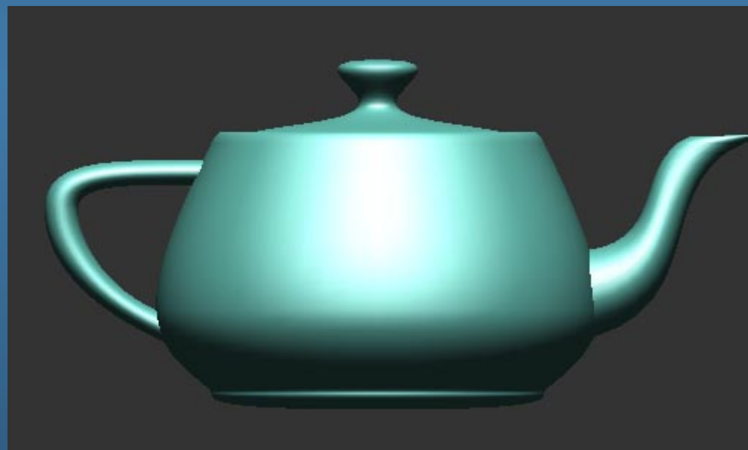
$i = i_{amb} + i_{diff} + i_{spec} + i_{emission}$

+



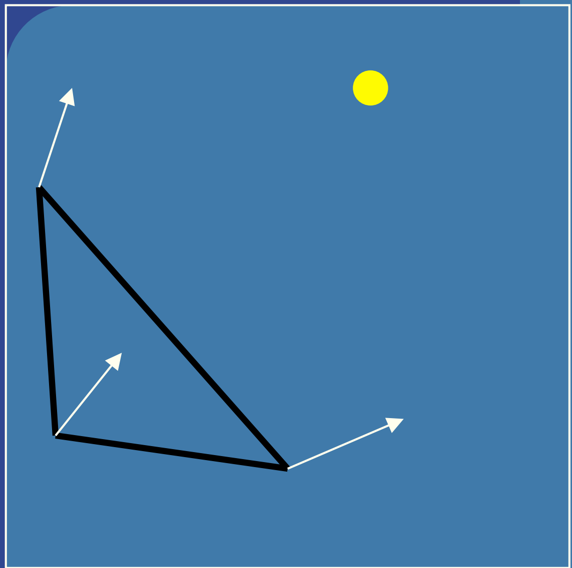
+

=



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

Lighting



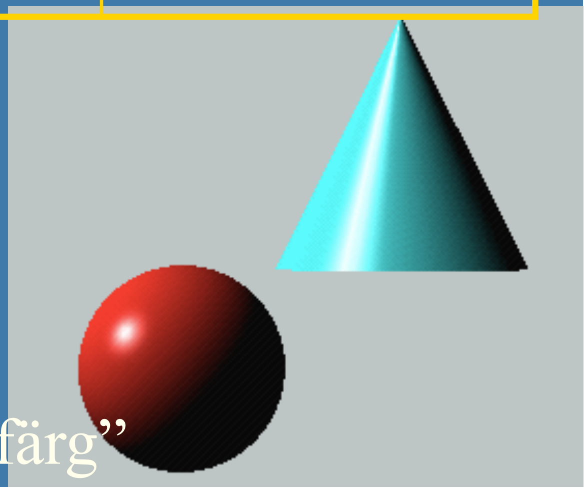
Light:

- •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.:

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

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

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

zero if $\mathbf{n} \bullet \mathbf{l} < 0$

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

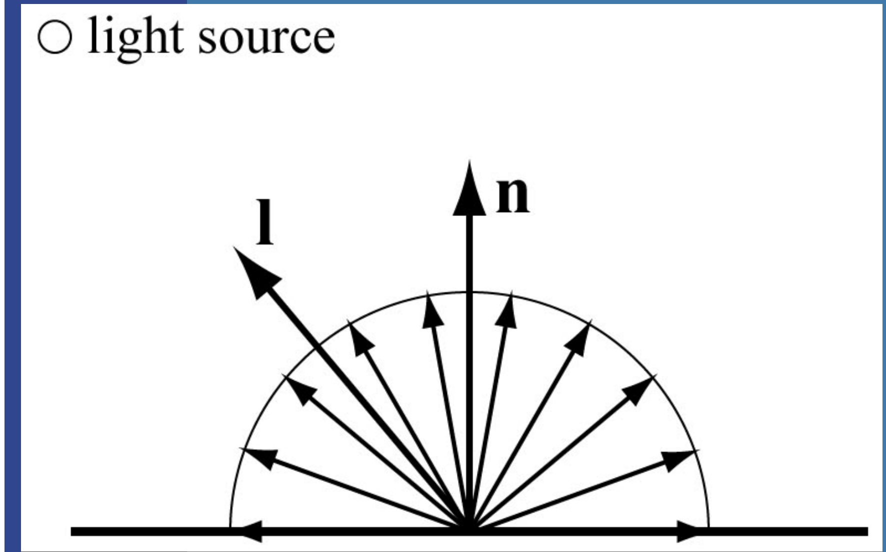
Blinn's reflection model: $$\mathbf{i}_{spec} = \max(0, (\mathbf{h} \cdot \mathbf{n}))^{m_{shi}} \mathbf{m}_{spec} \otimes \mathbf{s}_{spec}$$

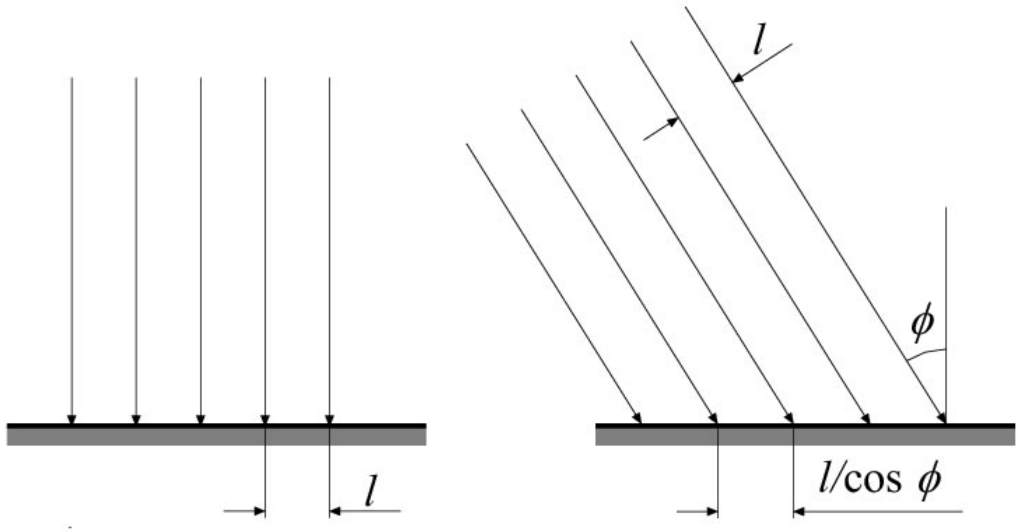
$$\mathbf{i}_{emission} = \mathbf{m}_{emission}$$

Diffuse component : i_{diff}

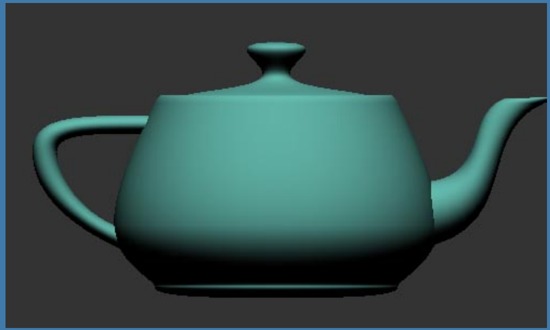
- $i = i_{amb} + i_{diff} + i_{spec} + i_{emission}$
- Diffuse is Lambert's law: $i_{diff} = \mathbf{n} \cdot \mathbf{l} = \cos\phi$
- Photons are scattered equally in all directions

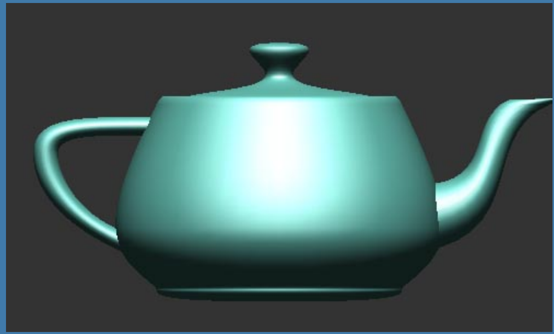
$$\mathbf{i}_{diff} = (\mathbf{n} \cdot \mathbf{l})\mathbf{m}_{diff} \otimes \mathbf{s}_{diff}$$



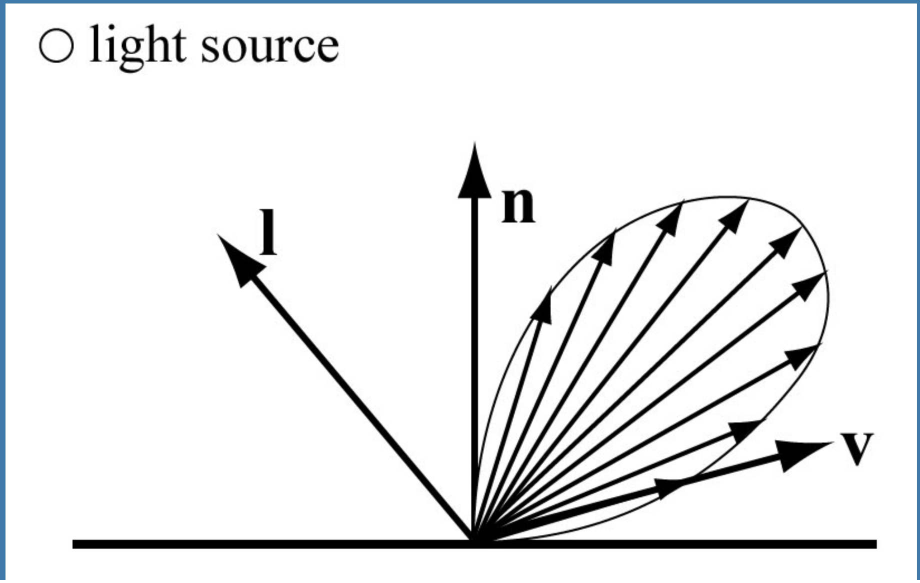


Lighting
Specular component : i_{spec}





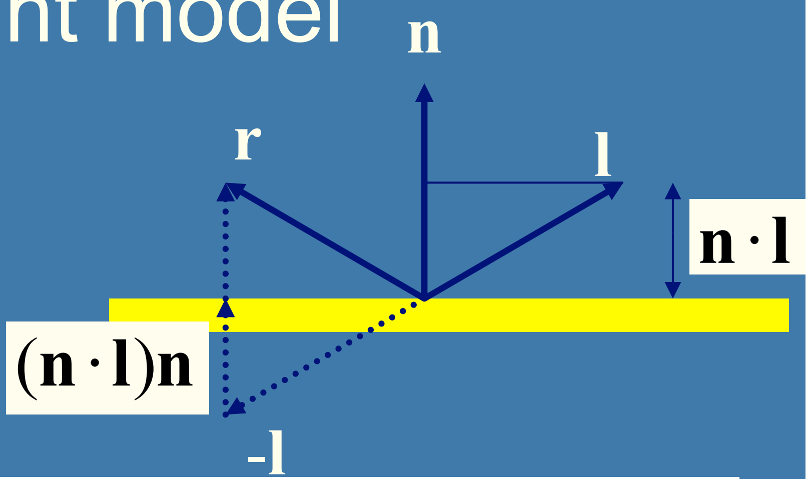
- Diffuse is dull (left)
- Specular: simulates a highlight



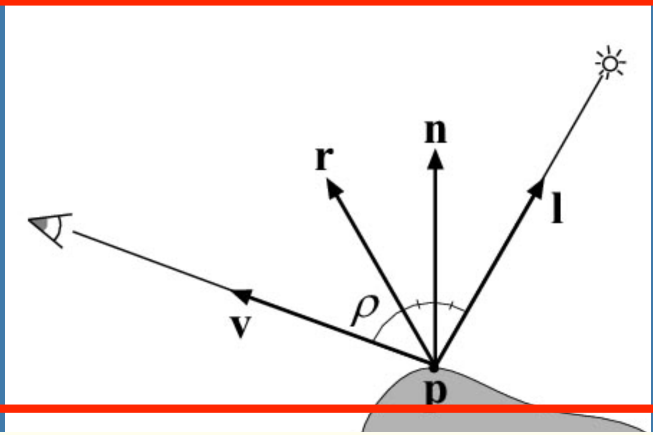
Specular component: Phong

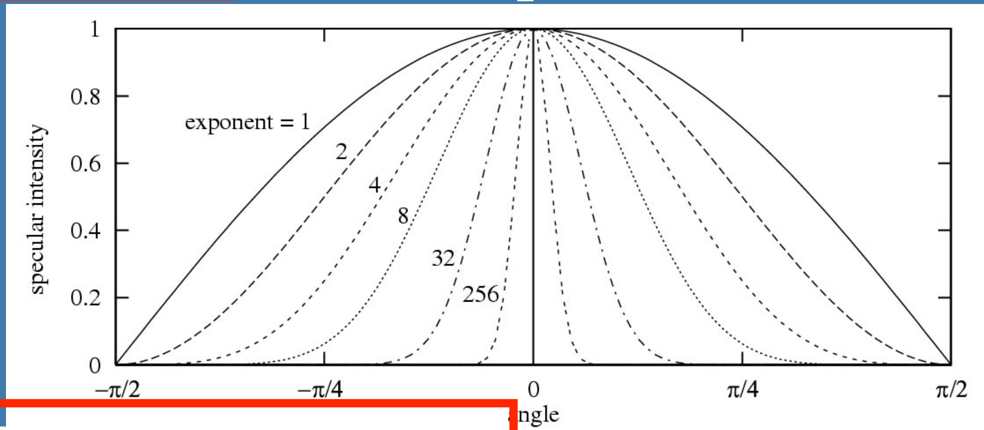
- Phong specular highlight model
- Reflect $\mathbf{l}$ around $\mathbf{n}$:

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



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





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

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

The University of New Mexico

Halfway Vector

Blinn proposed replacing **v·r** by **n·h** where

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

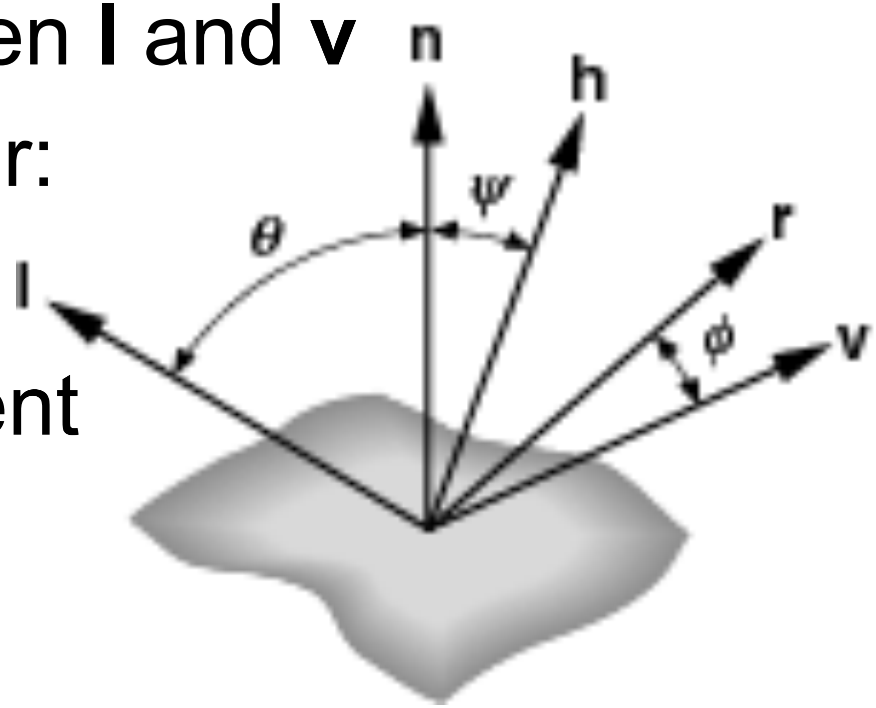
$(\mathbf{l}+\mathbf{v})/2$ is halfway between **l** and **v**

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

$\psi = \phi/2$

Must then adjust exponent

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

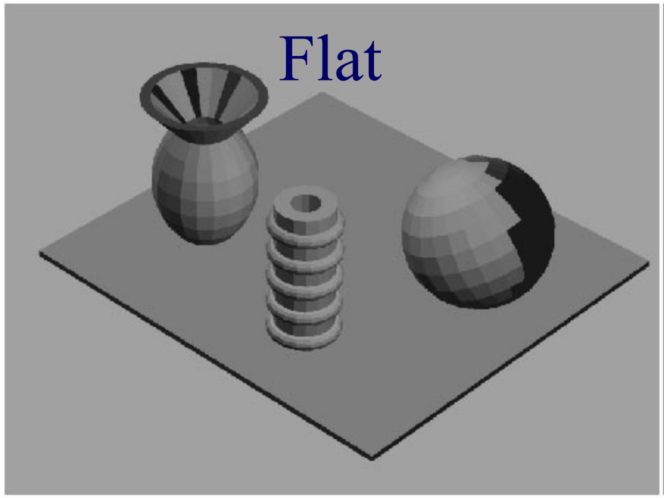


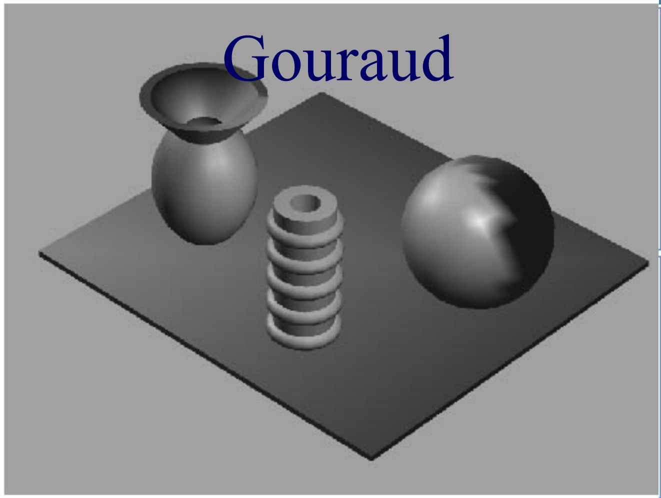
$(e' \approx 4e)$

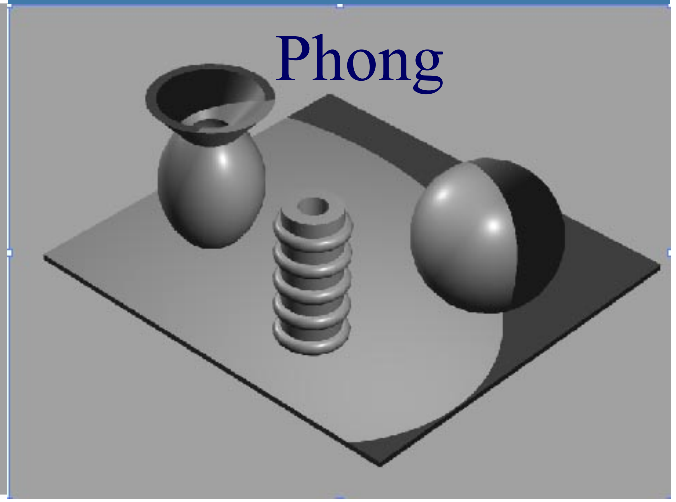
$$\boxed{\mathbf{i}_{spec} = \max(0,(\mathbf{h}\cdot\mathbf{n}))^{m_{shi}}\,\mathbf{m}_{spec} \otimes \mathbf{s}_{spec}}$$

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 blend operator: $\mathbf{c}_o = \alpha \mathbf{c}_s + (1-\alpha)\mathbf{c}_d$

Color already in the frame buffer at the corresponding position

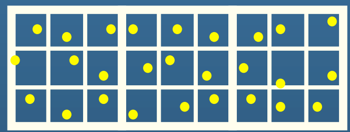
Rendered object

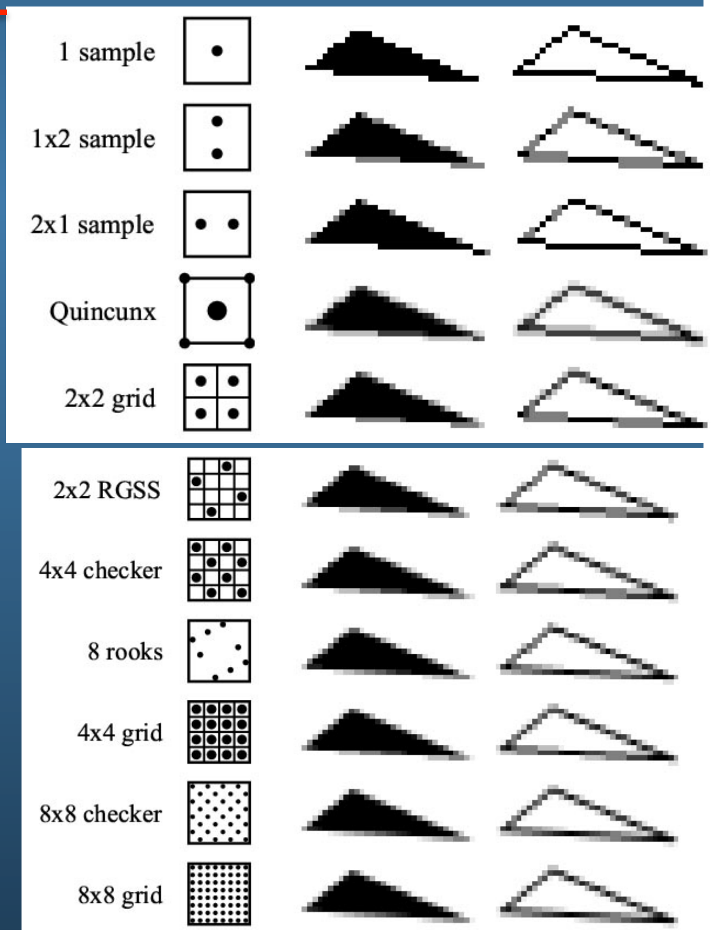
Transparency

- 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 is order dependent.**

Leture 3.2: Sampling, filtrering, and Antialiasing

- When does it occur?
  - In 1) pixels, 2) time, 3) texturing
  - Nyquist
- Filters
- Supersampling schemes
- Jittered sampling





04. Texturing

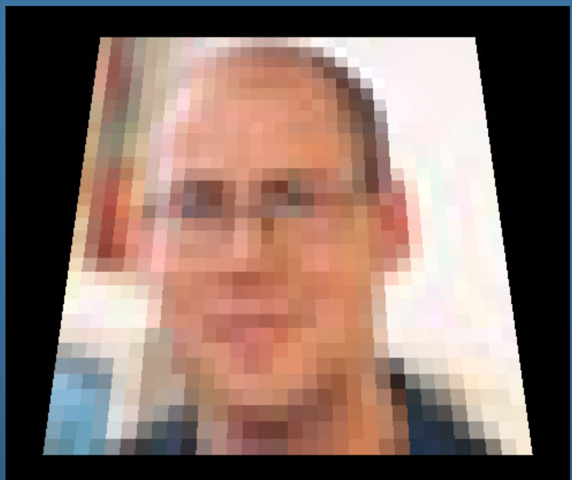
Most important:

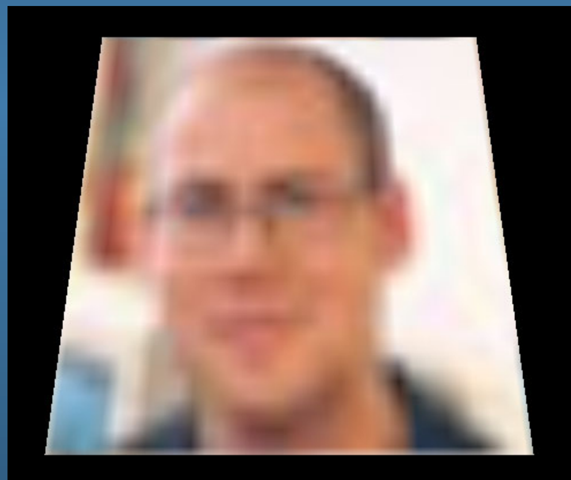
- Texturing, environment mapping
- Bump mapping
- 3D-textures,
- Particle systems
- Sprites and billboards

Filtering

FILTERING:

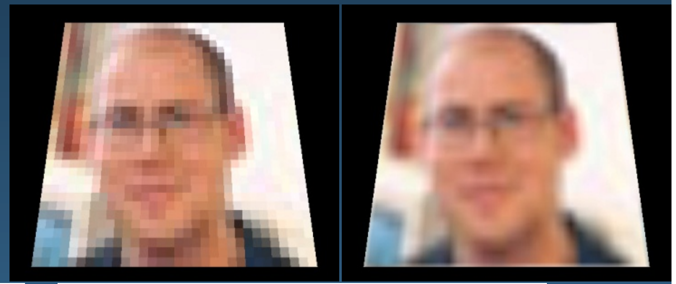
- For magnification: Nearest or Linear (box vs Tent filter)



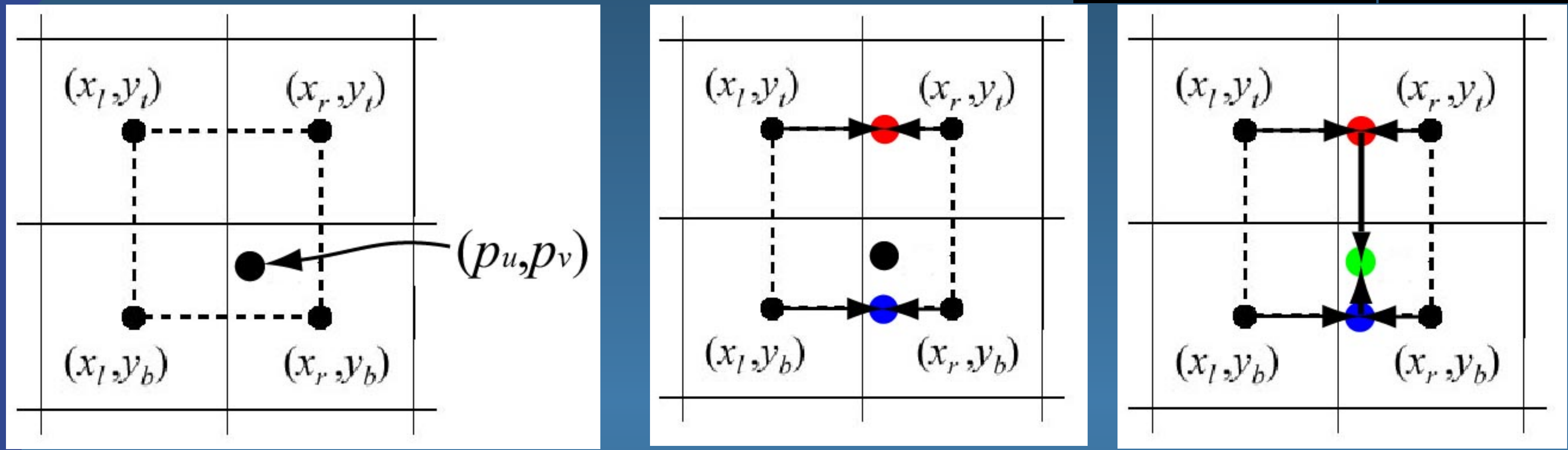


- For minification:
  - Bilinear – using mipmapping
  - Trilinear – using mipmapping
  - Anisotropic – up to 16 mipmap lookups along line of anisotropy

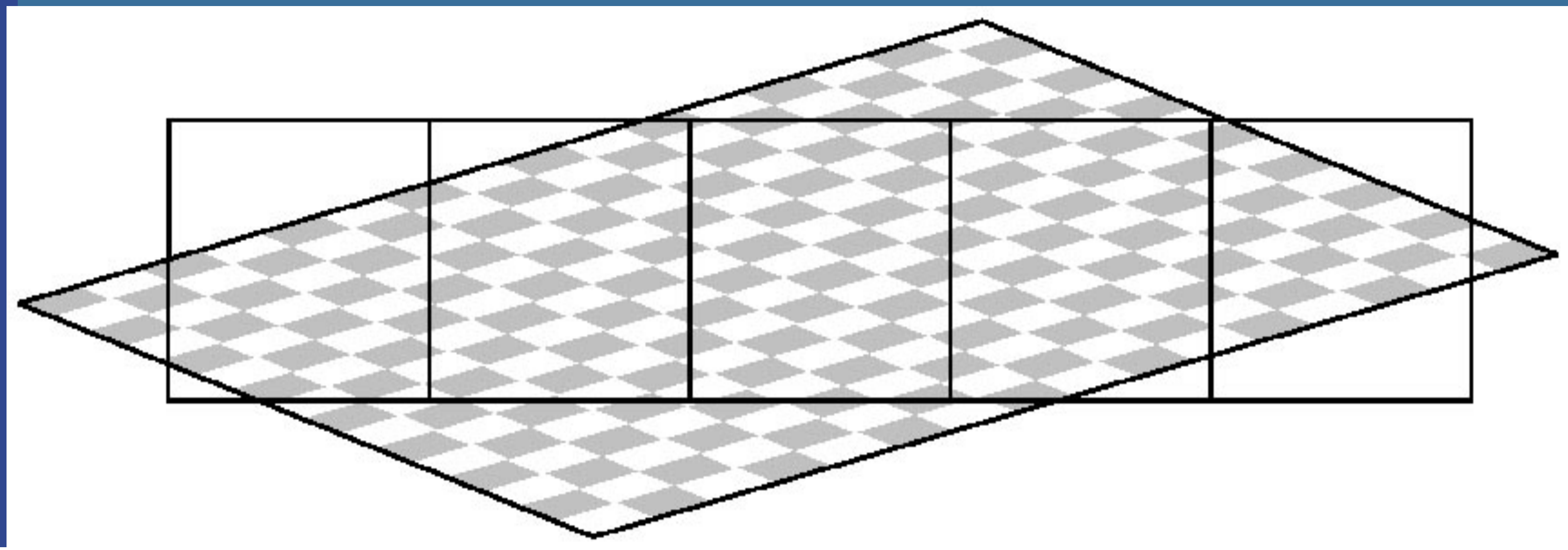
Interpolation



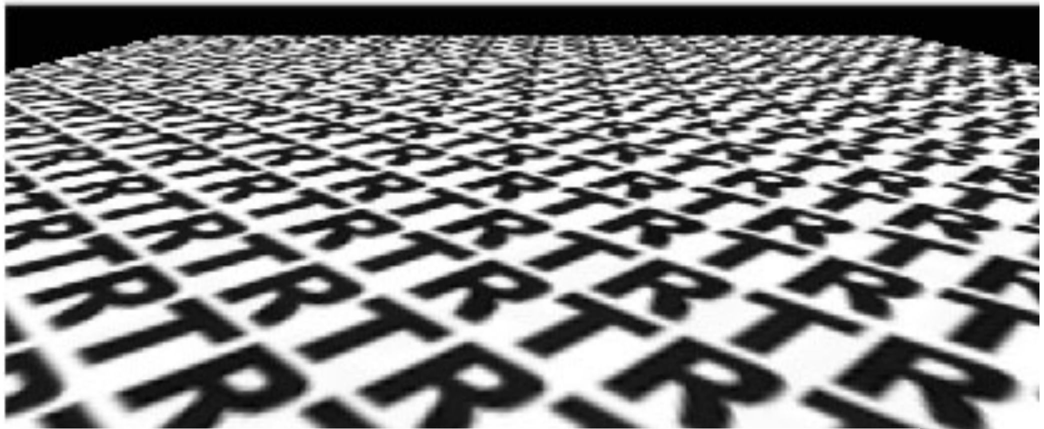
Magnification

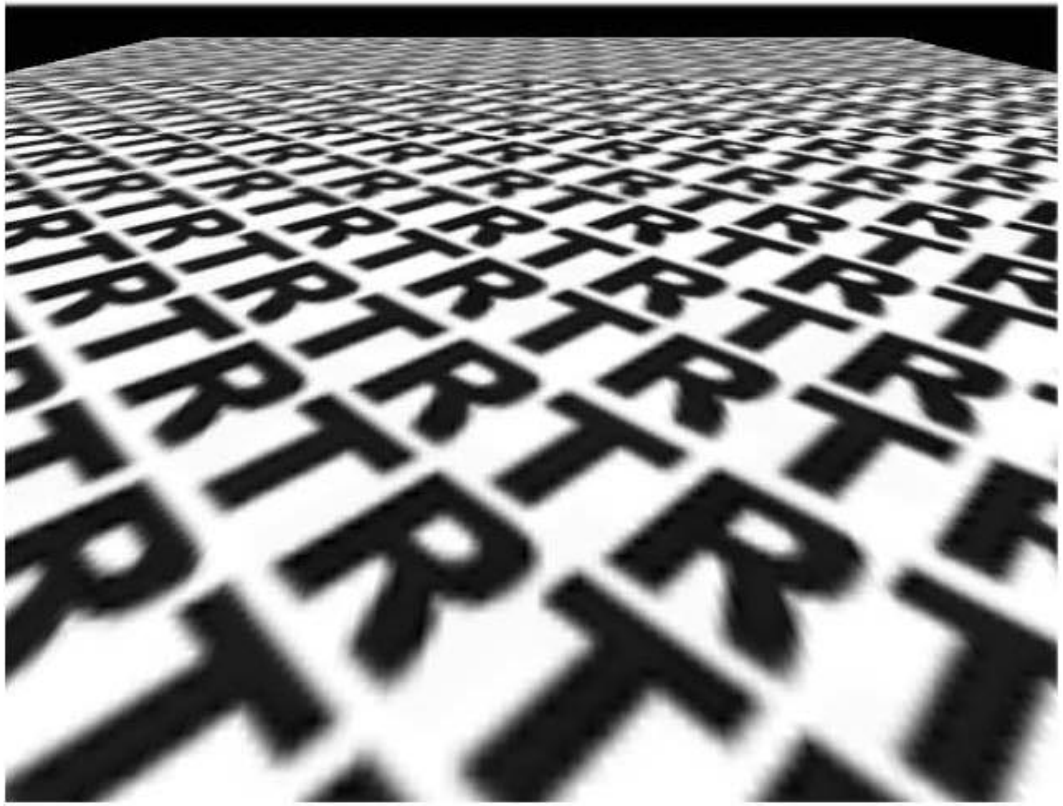


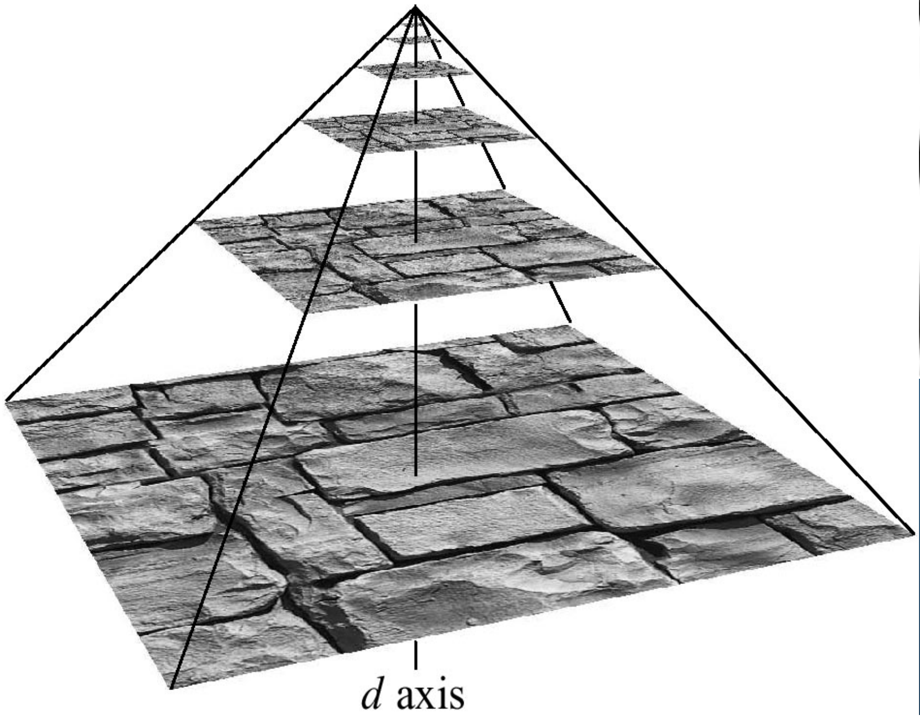
Minification



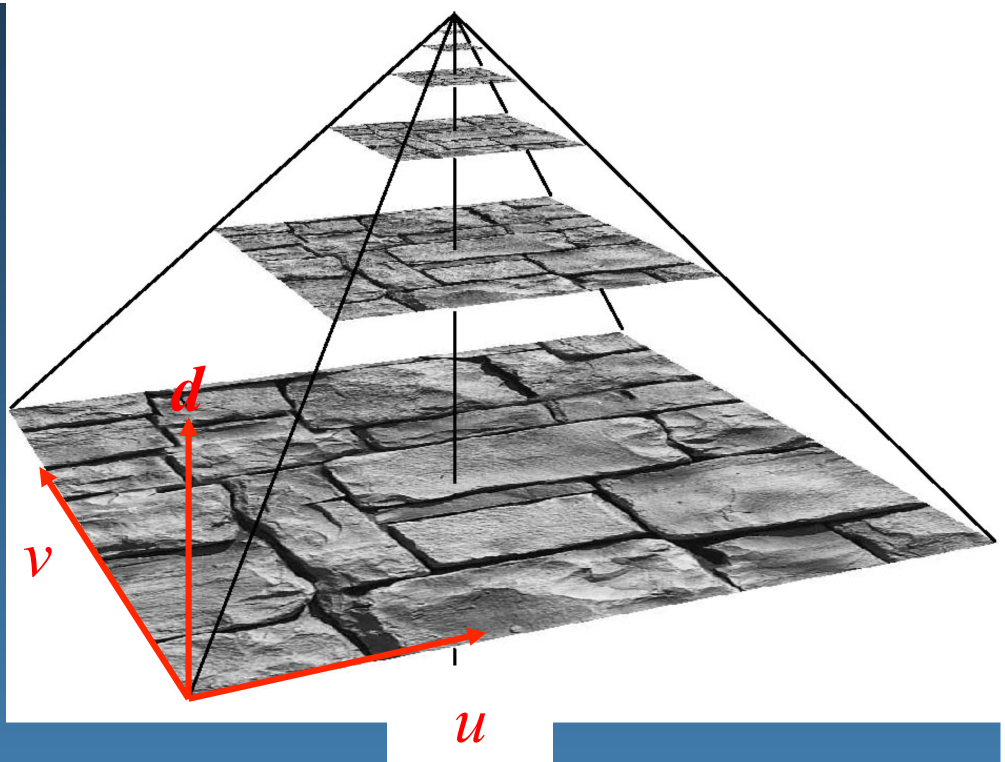
Bilinear filtering using Mipmapping







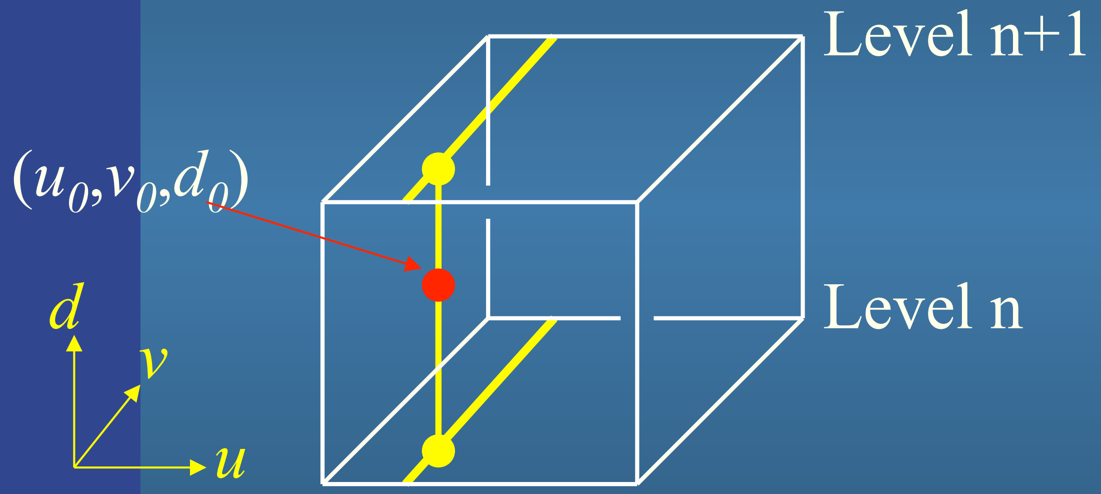
Mipmapping



- 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

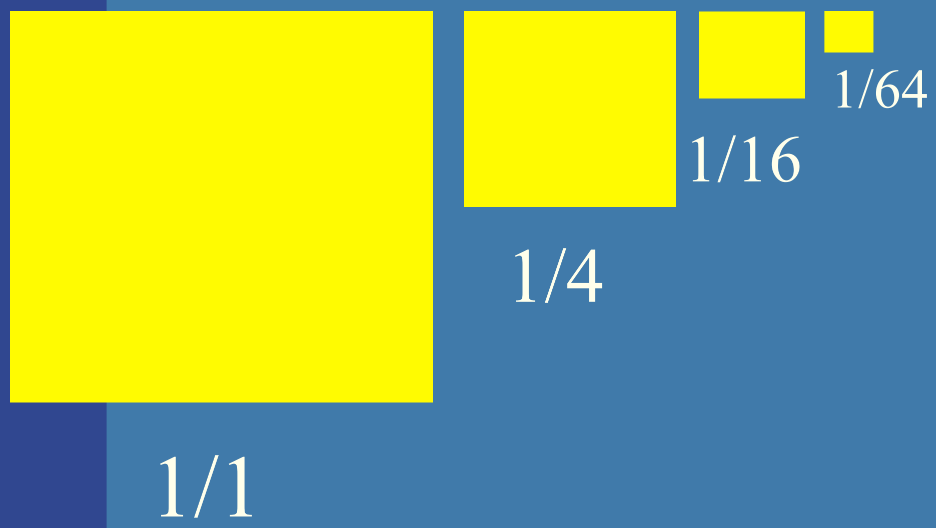
- Interpolate between those bilinear values
  - Gives trilinear interpolation

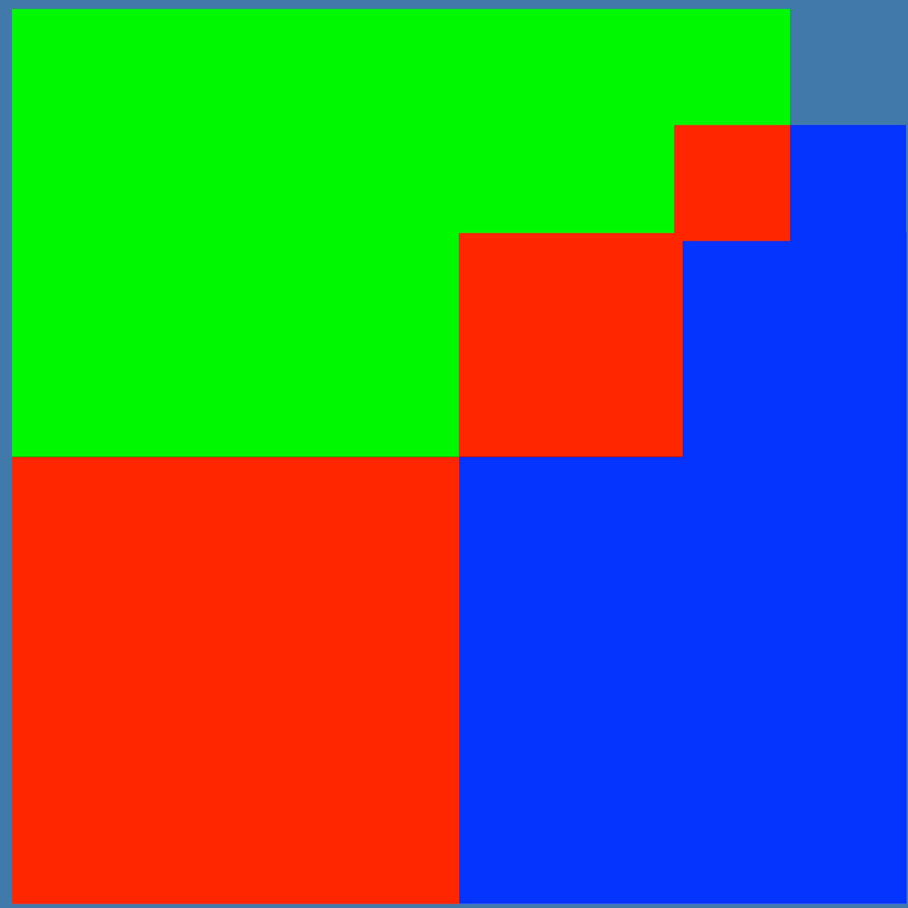


- Constant time filtering: 8 texel accesses

Mipmapping: Memory requirements

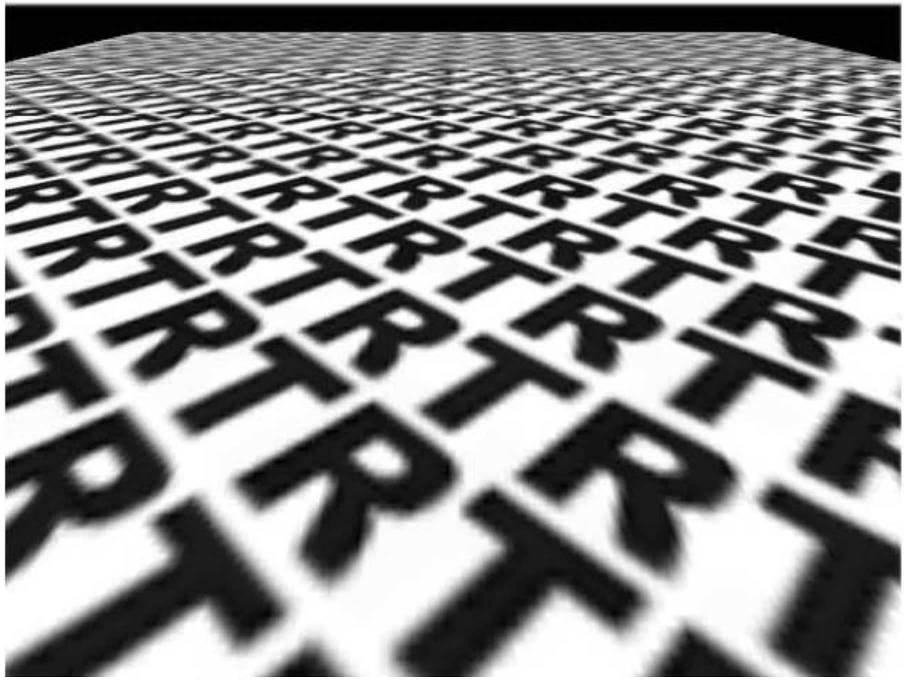
- Not twice the number of bytes…!



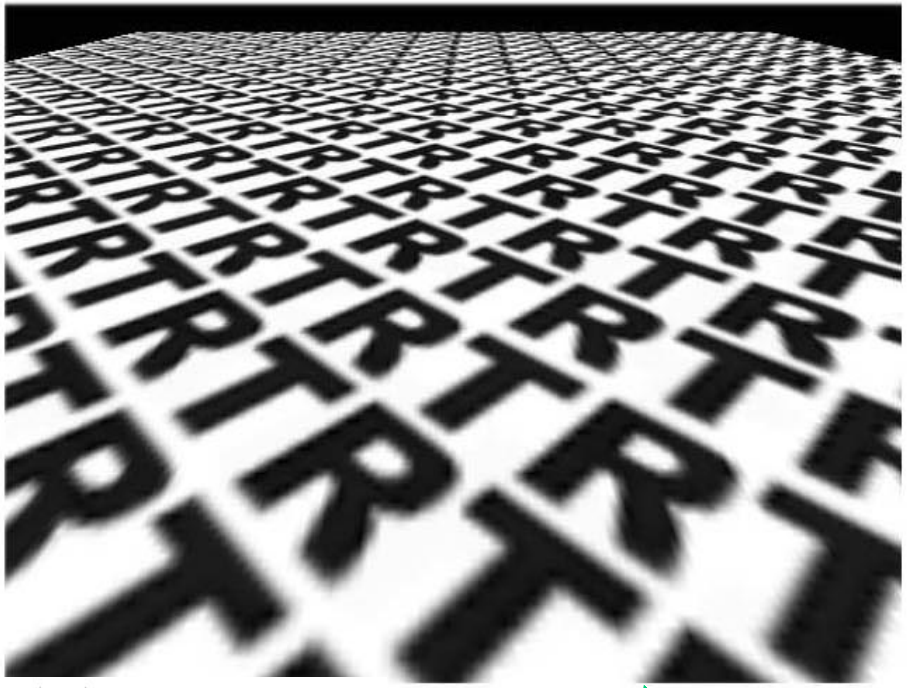


- Rather 33% more – not that much

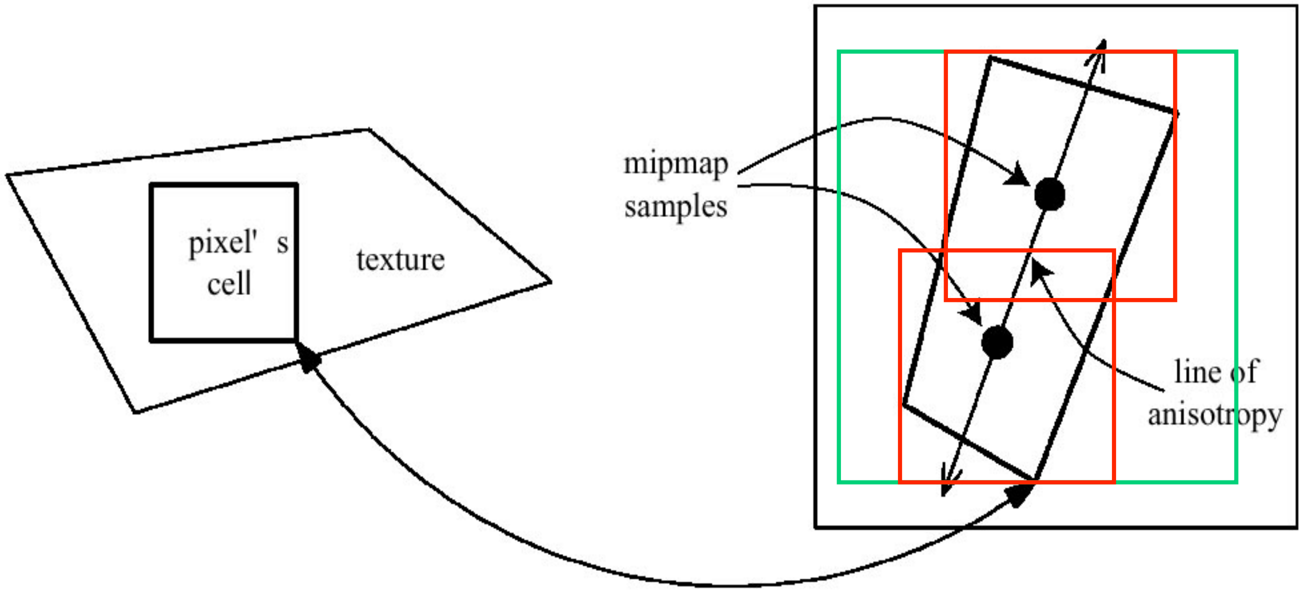
Anisotropic texture filtering



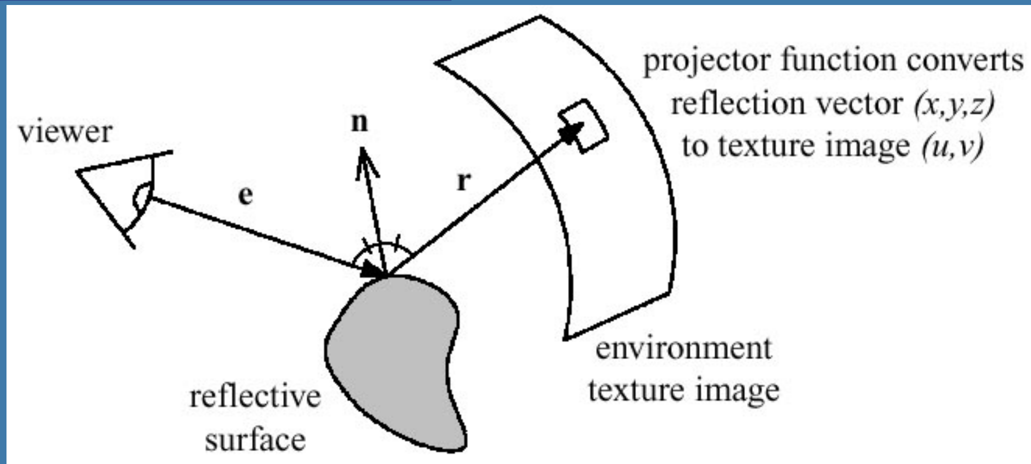
pixel space



texture space

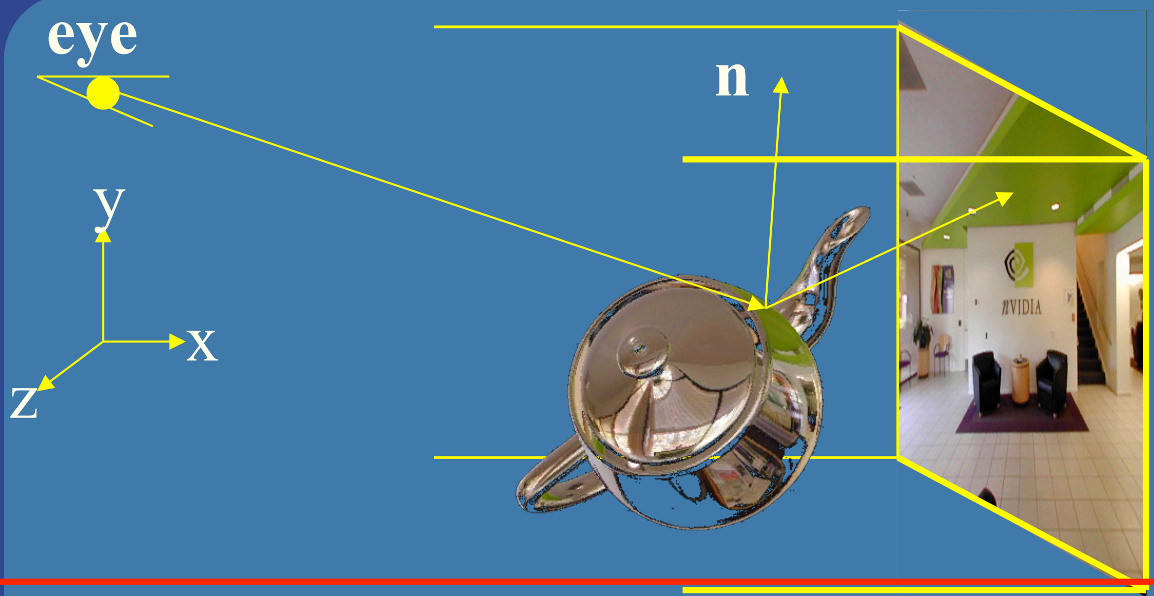


Environment mapping



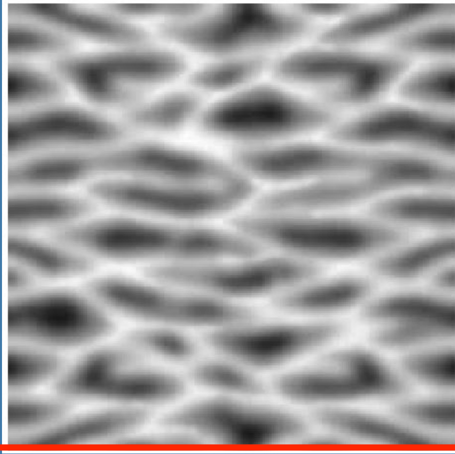
- Assumes the environment is infinitely far away
- Sphere mapping
- Cube mapping is the norm nowadays
  - Advantages: no singularities as in sphere map
  - Much less distortion
  - Gives better result
  - Not dependent on a view position

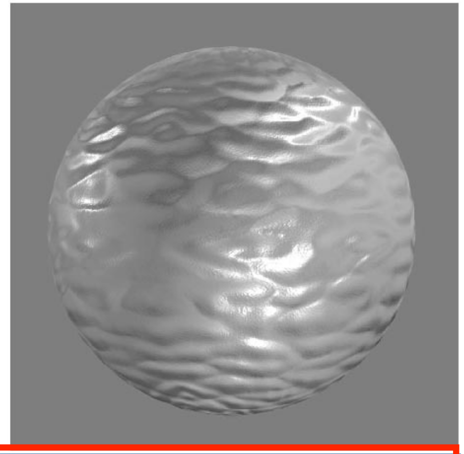
Cube mapping



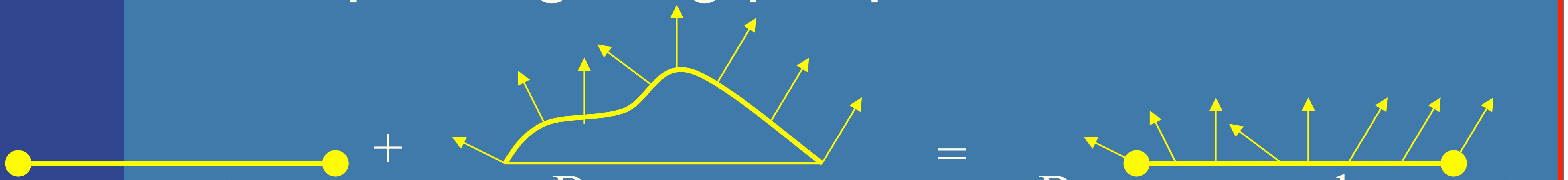
- Simple math: compute reflection vector, **r**
- Largest abs-value of component, determines which cube face.
  - Example: **r**=(5,-1,2) gives POS_X face
- Divide **r** by abs(5) gives (u,v)=(-1/5,2/5)
- If your hardware has this feature, then it does all the work

Bump mapping





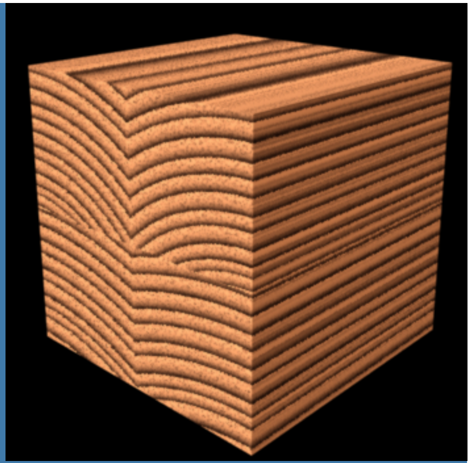
- by Blinn in 1978
- Inexpensive way of simulating wrinkles and bumps on geometry
  - Too expensive to model these geometrically
- Instead let a texture modify the normal at each pixel, and then use this normal to compute lighting per pixel

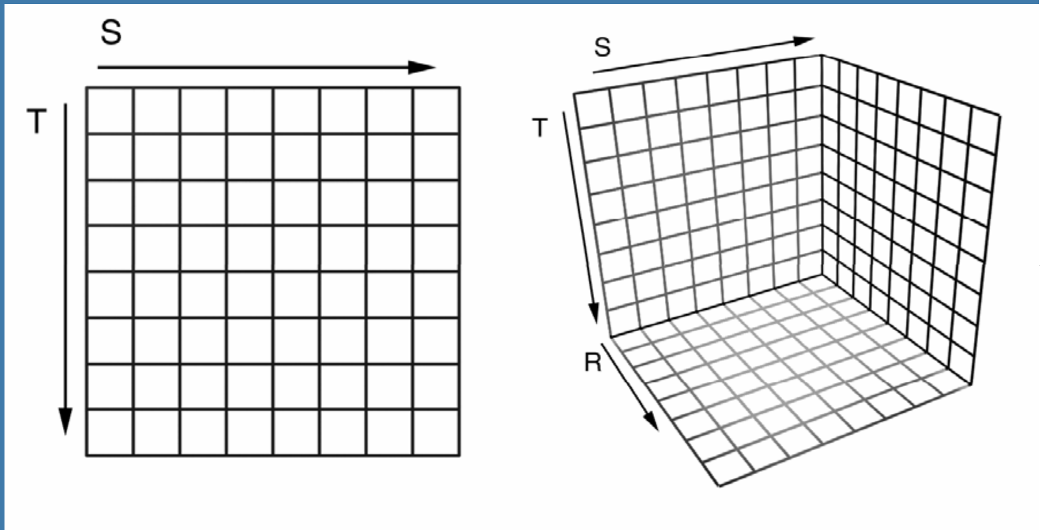


Stores heights: can derive normals

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





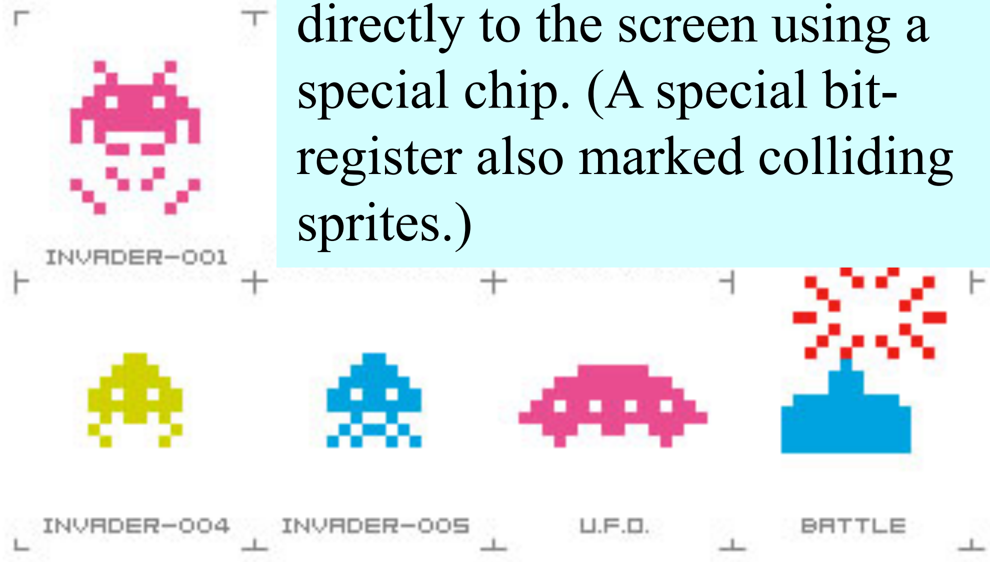
05. Texturing:

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.)

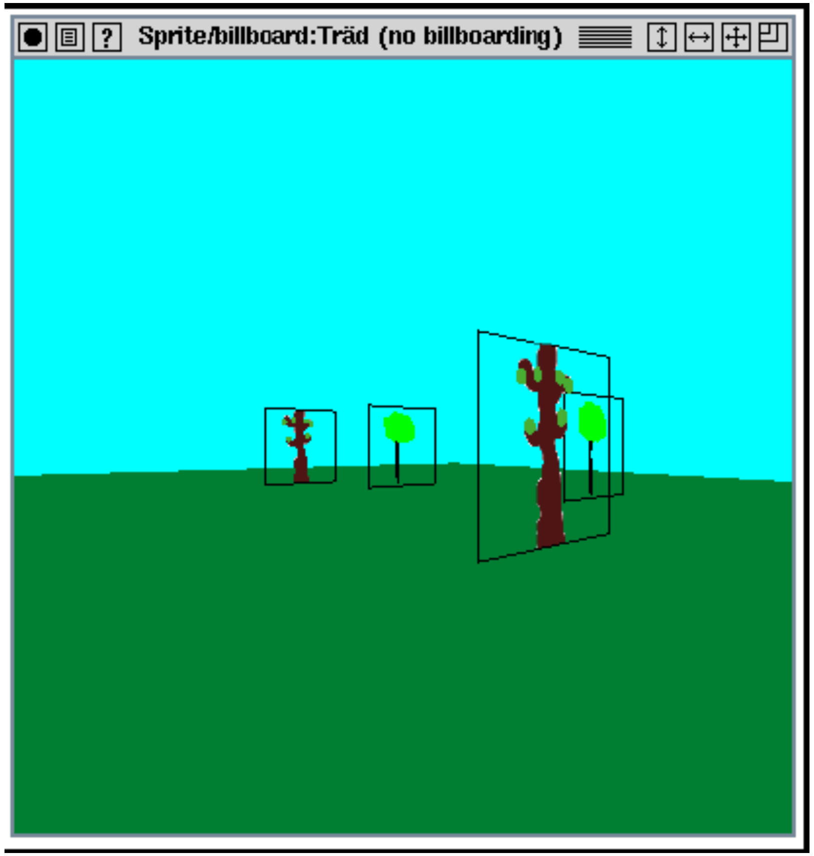
```
GLbyte M[64]=
{   127,0,0,127, 127,0,0,127,
    127,0,0,127, 127,0,0,127,
    0,127,0,0, 0,127,0,127,
    0,127,0,127, 0,127,0,0,
    0,0,127,0, 0,0,127,127,
    0,0,127,127, 0,0,127,0,
    127,127,0,0, 127,127,0,127,
    127,127,0,127, 127,127,0,0};

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();
}
```

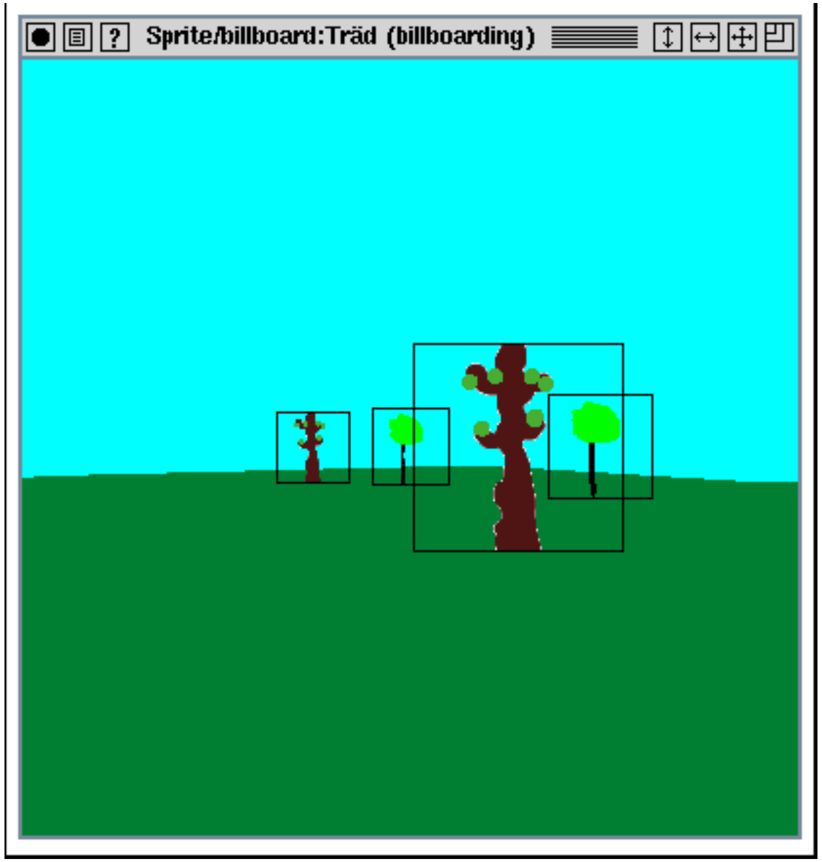


Billboards

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

Billboards





- Rotate them towards viewer
  - Either $\boxed{\text{by rotation matrix or}}$
  - by orthographic projection

Billboards

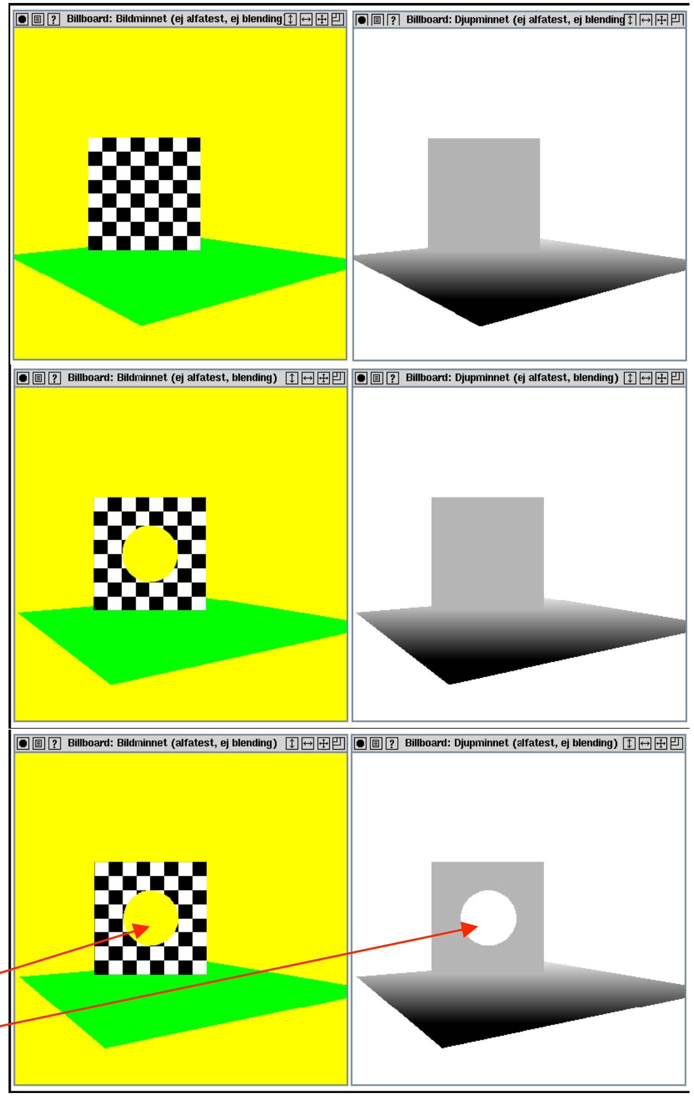
- Fix correct transparency by blending AND using alpha-test
  - In fragment shader:

    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.

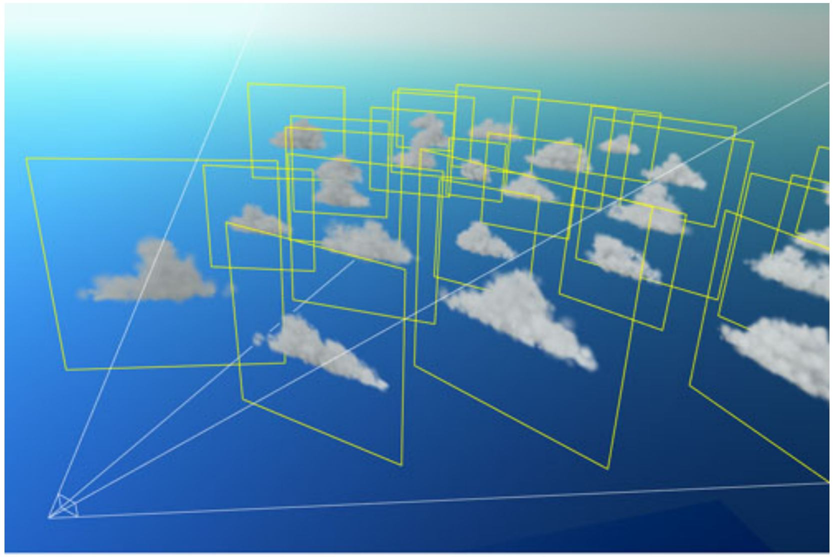
E.g. here: so that objects behind show through the hole

Color Buffer | Depth Buffer

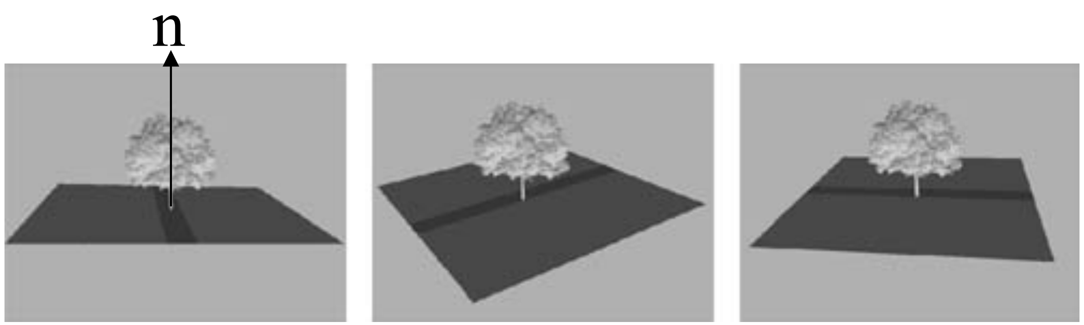


With blending

With alpha test



(Also called *Impostors*)



axial billboarding

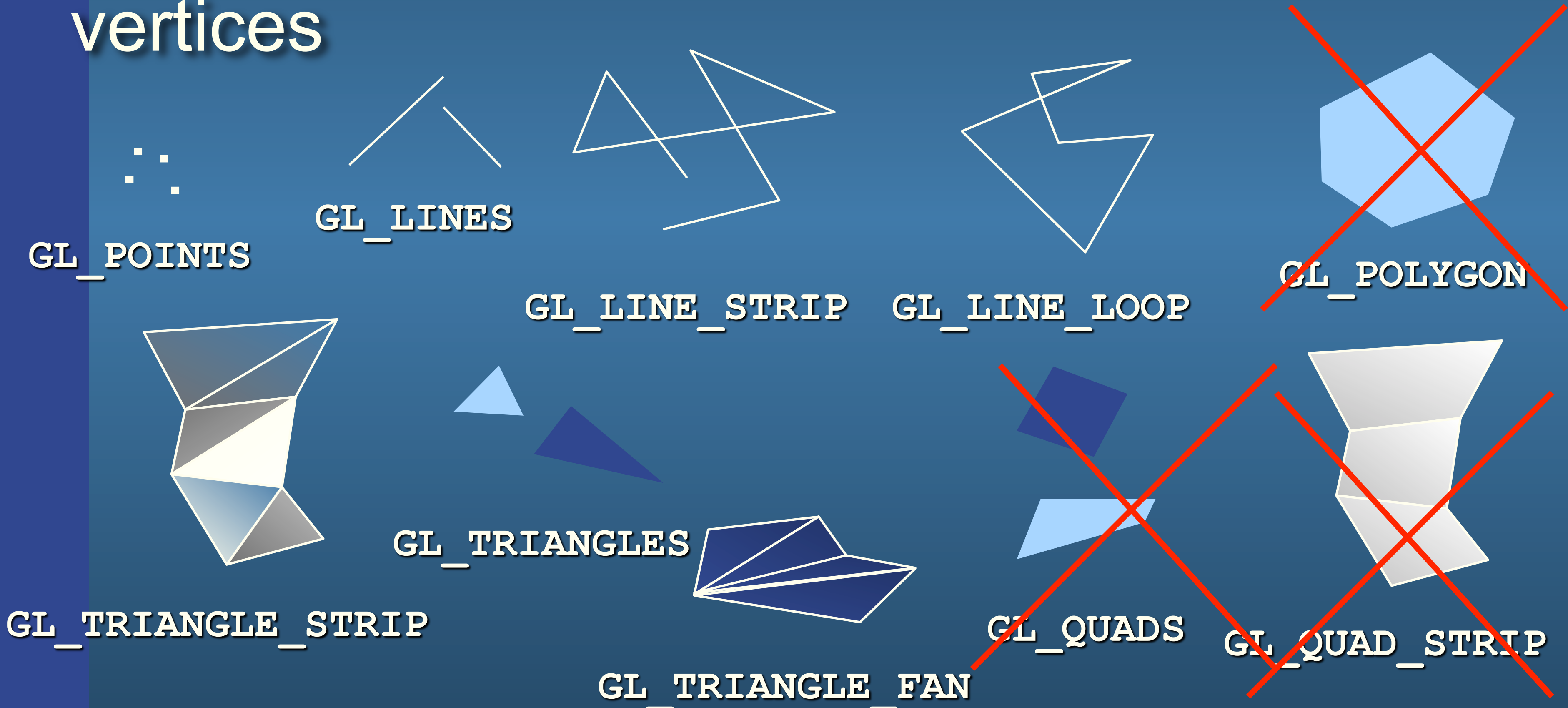
The rotation axis is fixed and disregarding the view position

Lecture 5: OpenGL

- Uses OpenGL (or DirectX)
  - Will not ask about syntax. Know how to use.
  - 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

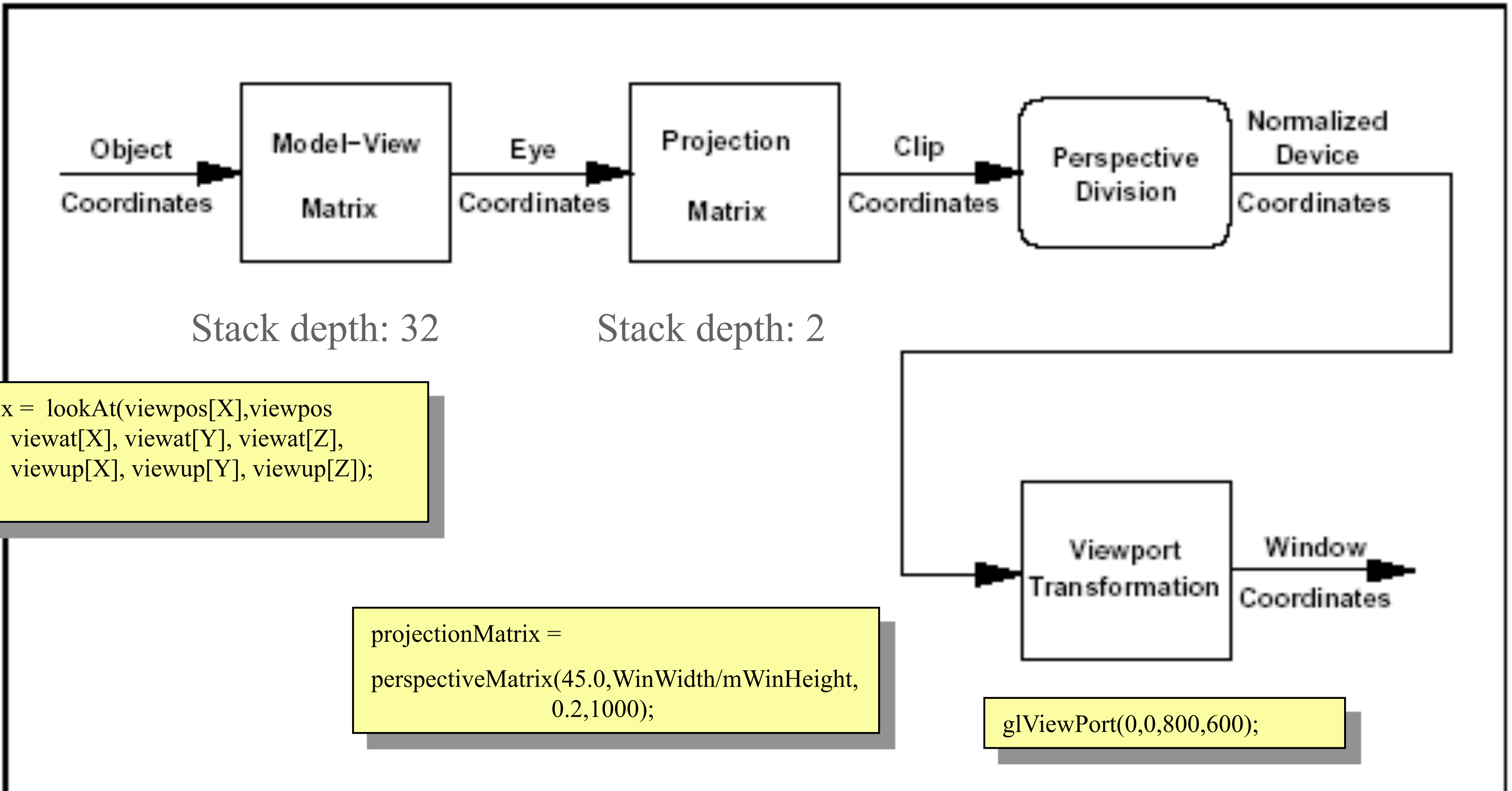


Figure 2.6. Vertex transformation sequence.

```
modelViewMatrix = lookAt(viewpos[X],viewpos[Y],viewpos[Z], viewat[X], viewat[Y], viewat[Z],
                  viewup[X], viewup[Y], viewup[Z]);
```

```
projectionMatrix =
perspectiveMatrix(45.0,WinWidth/mWinHeight,
                  0.2,1000);
```

```
glViewPort(0,0,800,600);
```

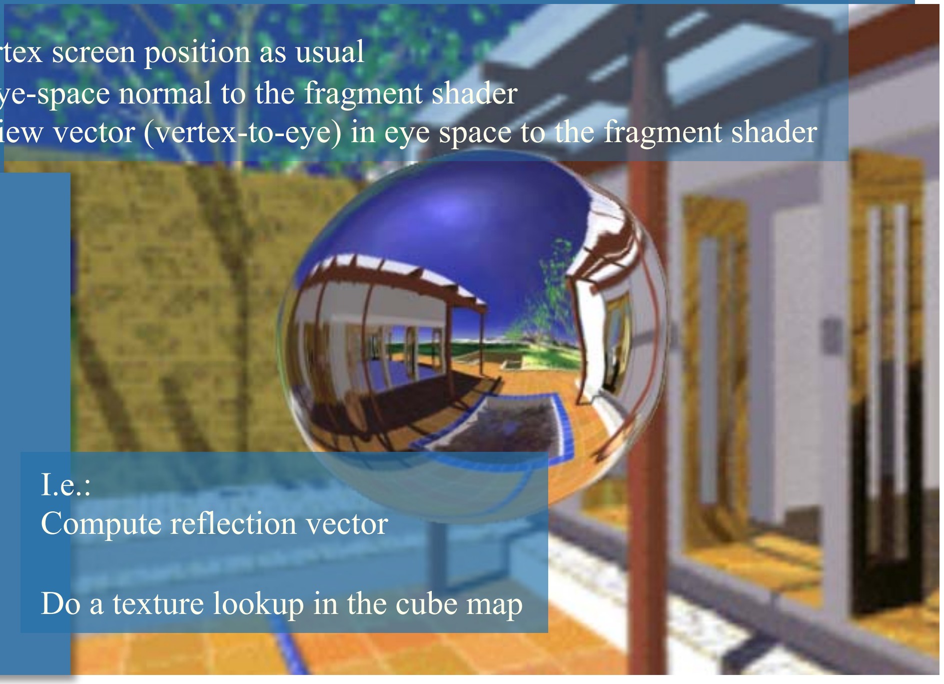
Reflections with environment mapping

- 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;
}
```

I.e.:
Compute vertex screen position as usual
Output the eye-space normal to the fragment shader
Output the view vector (vertex-to-eye) in eye space to the fragment shader



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);
}
```

I.e.:
Compute reflection vector

Do a texture lookup in the cube map

Buffers

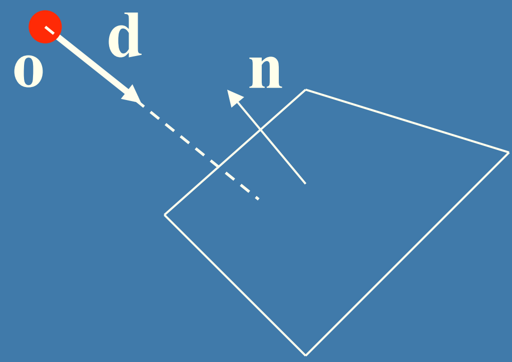
- Frame buffer
  - Back/front/left/right – **glDrawBuffers()**
- Depth buffer (z-buffer)
  - For correct depth sorting
  - Instead of BSP-algorithm, painters algorithm…
  - **glDepthFunc(), glDepthMask**
- Stencil buffer
  - Shadow volumes,
  - **glStencilFunc(), glStencilMask, glStencilMaskSeparate, 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)

    Test:
    1. axes orthogonal to side of A,
    2. axes orthogonal to side of B
    3. crossprod of edges of A and B
  - Dynamic tests – know what it means.
- E.g., describe an algorithm for intersection between a **ray** and a
  - polygon or sphere or plane.
- Know equations for ray, sphere, cylinder, plane

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 $\mathbf{r}(t)$ and solve for t:

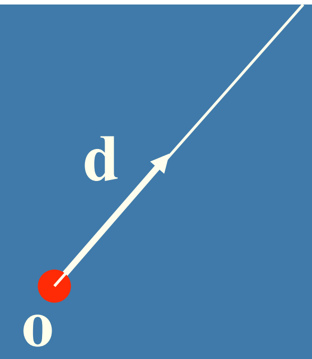
$\mathbf{n}\bullet(\mathbf{o}+t\mathbf{d}) + d = 0$

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

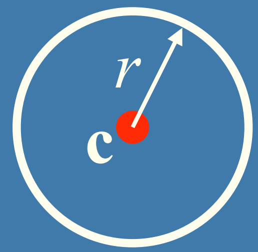
$t = (-d - \mathbf{n}\bullet\mathbf{o}) \,/\, (\mathbf{n}\bullet\mathbf{d})$

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*



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

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

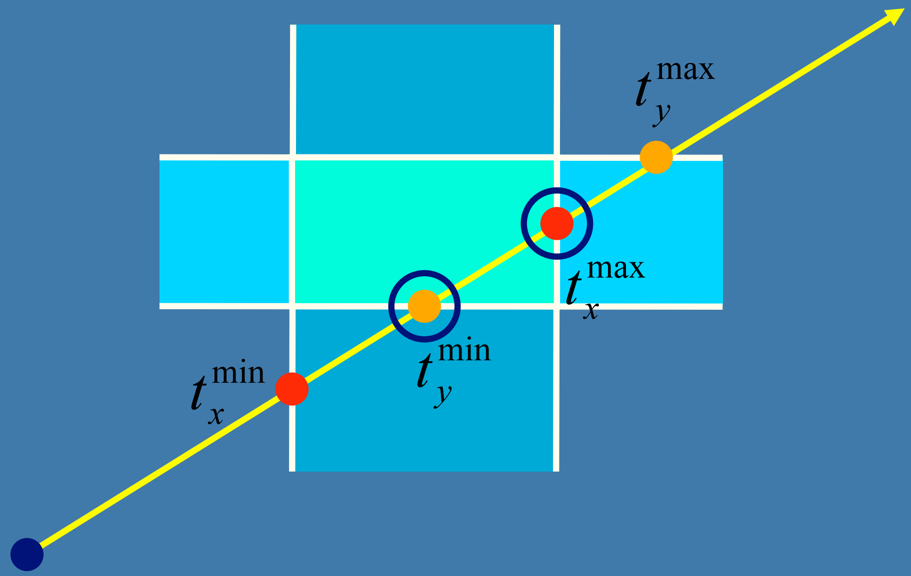
$$(\mathbf{d}\cdot\mathbf{d})t^2+2((\mathbf{o}-\mathbf{c})\cdot\mathbf{d})t+(\mathbf{o}-\mathbf{c})\cdot(\mathbf{o}-\mathbf{c})-r^2=0$$

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

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

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

Point/Plane

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

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

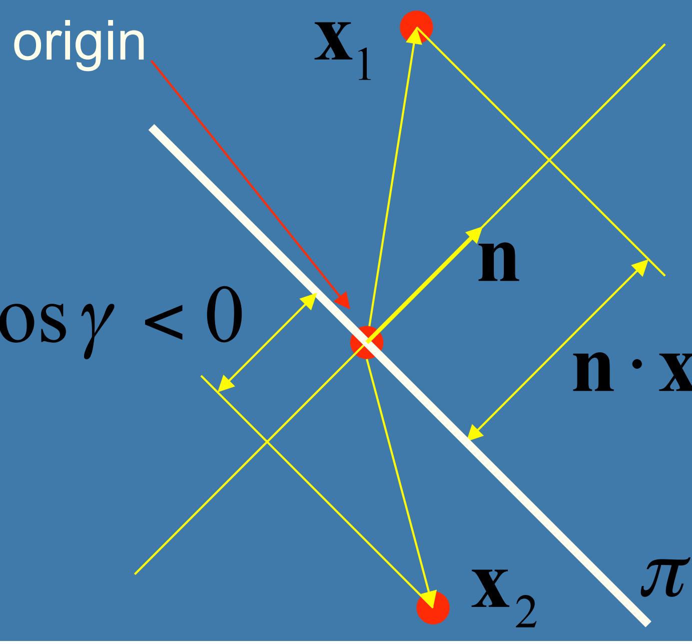
$$f(\mathbf{x}) = \mathbf{n}\cdot\mathbf{x} + d = 0 \qquad \text{for } \mathbf{x}\text{'s on the plane}$$

Negative half space

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

Positive half space

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

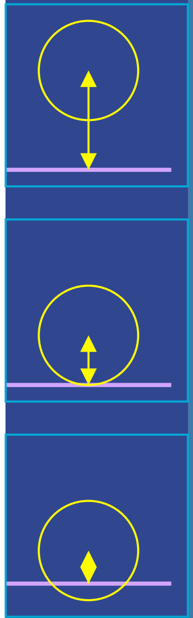


$$\mathbf{n}\cdot\mathbf{x}_2 = \|\mathbf{x}_2\|\cos\gamma < 0$$

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

Sphere/Plane
AABB/Plane

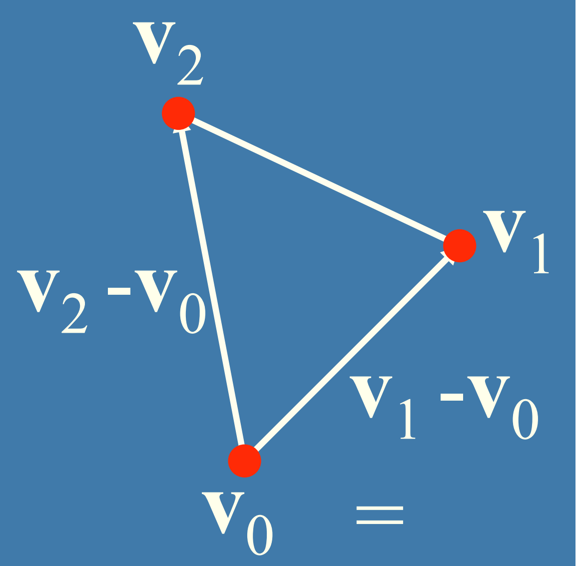
Plane: $\pi : \mathbf{n} \cdot \mathbf{p} + d = 0$
Sphere: $\mathbf{c}$ $\quad r$
Box: $\mathbf{b}^{\min}$ $\quad \mathbf{b}^{\max}$



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

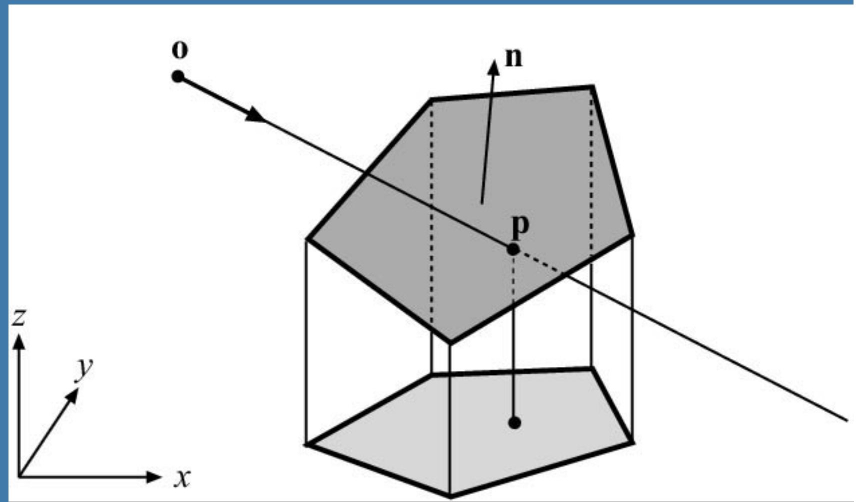


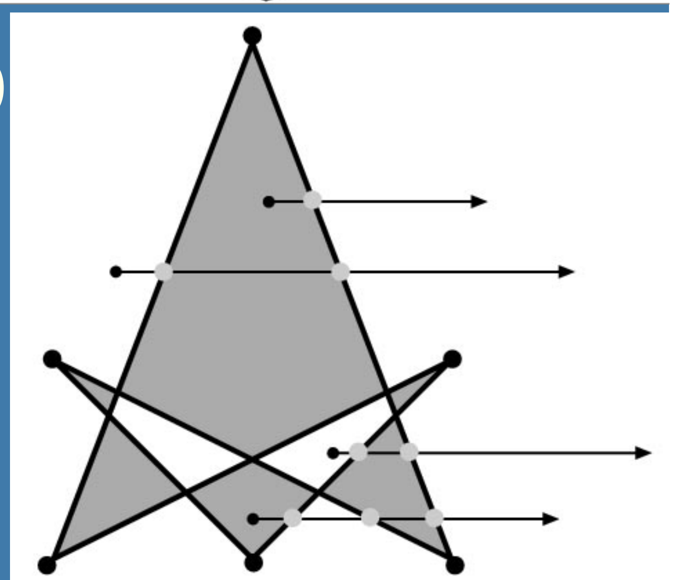
- Ray: $\mathbf{r}(t)=\mathbf{o}+t\mathbf{d}$
- Triangle vertices: $\mathbf{v}_0$, $\mathbf{v}_1$, $\mathbf{v}_2$
- A point in the triangle:
- $\mathbf{t}(u,v)=\mathbf{v}_0+u(\mathbf{v}_1-\mathbf{v}_0)+v(\mathbf{v}_2-\mathbf{v}_0)=$ $=$
  $(1-u-v)\mathbf{v}_0+u\mathbf{v}_1+v\mathbf{v}_2 \quad [u,v>=0,\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

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

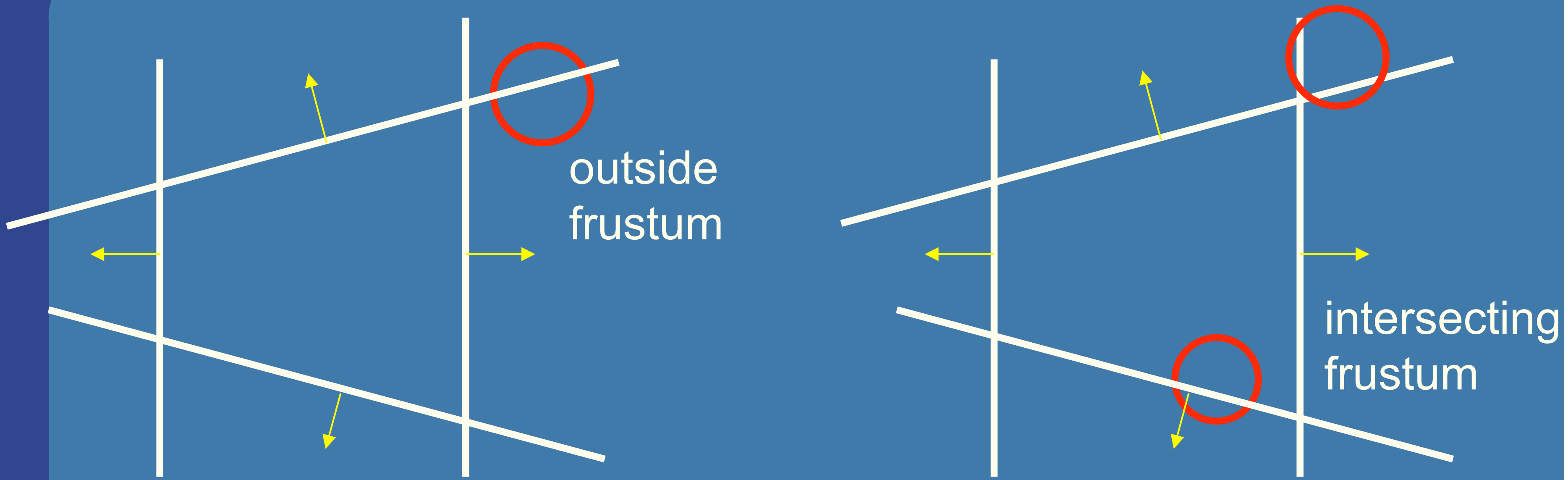
Ray/Polygon: very briefly

- 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





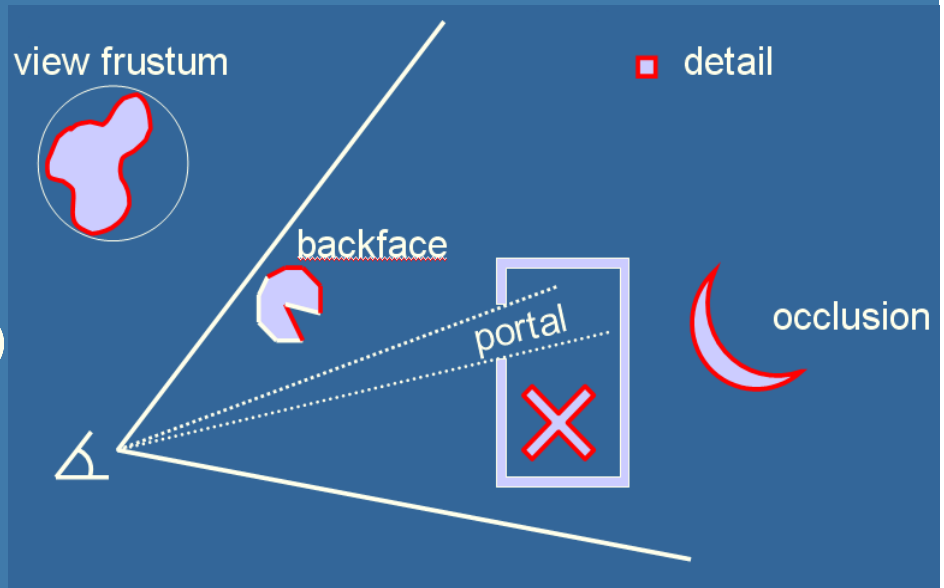
View frustum testing example



- 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

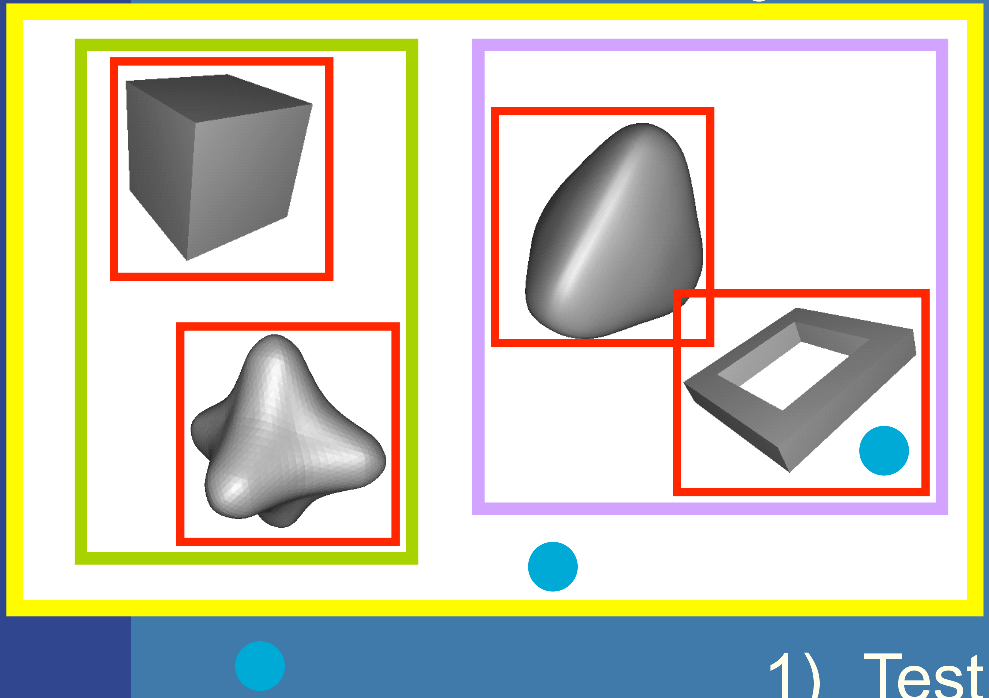


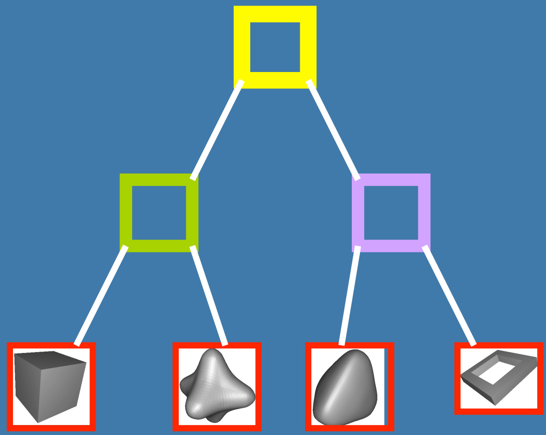
  - Levels-of-detail:

- 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





click!

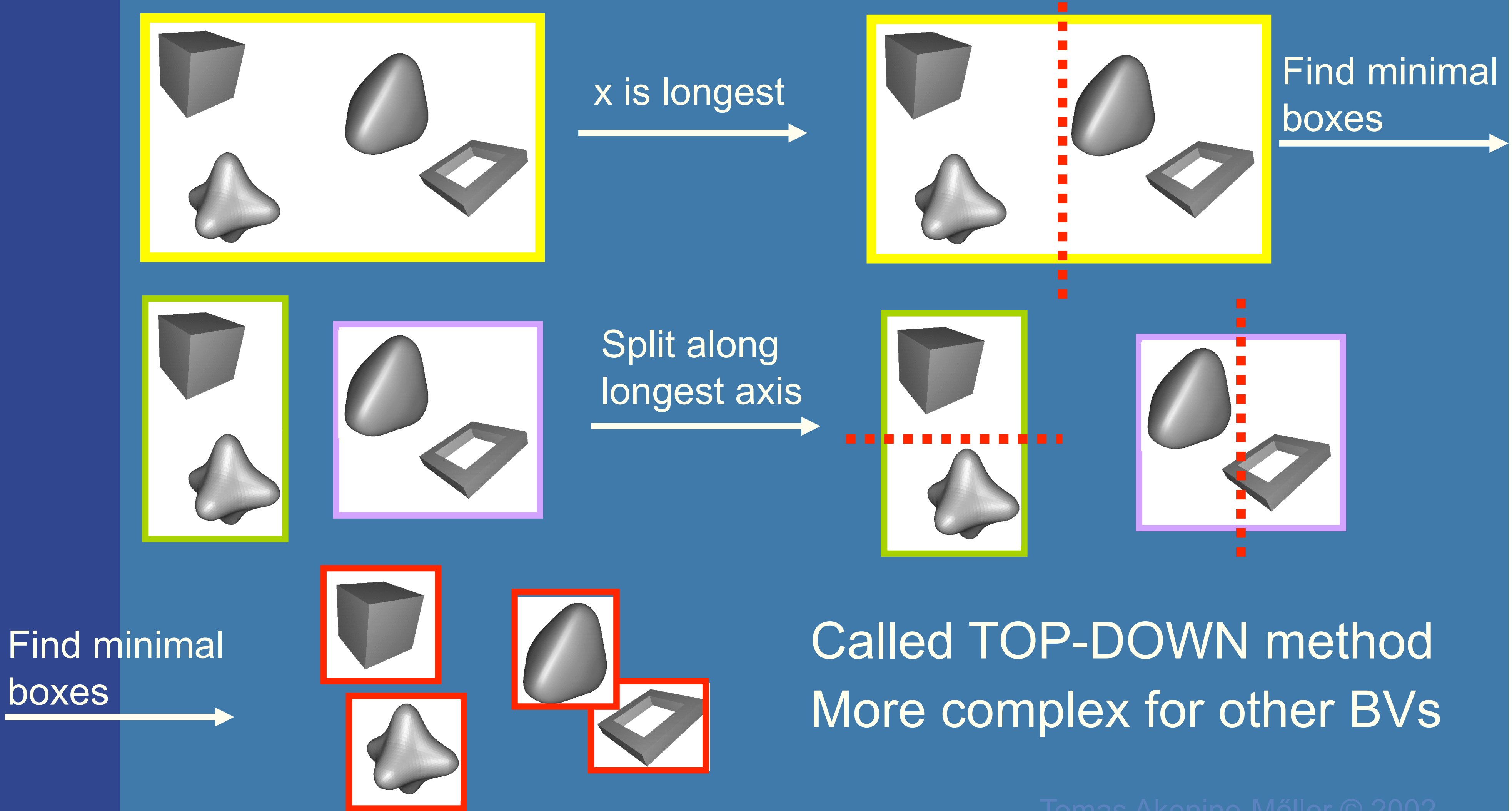
1) Test the root first
2) Descend recursively as needed
3) Terminate traversal when possible

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

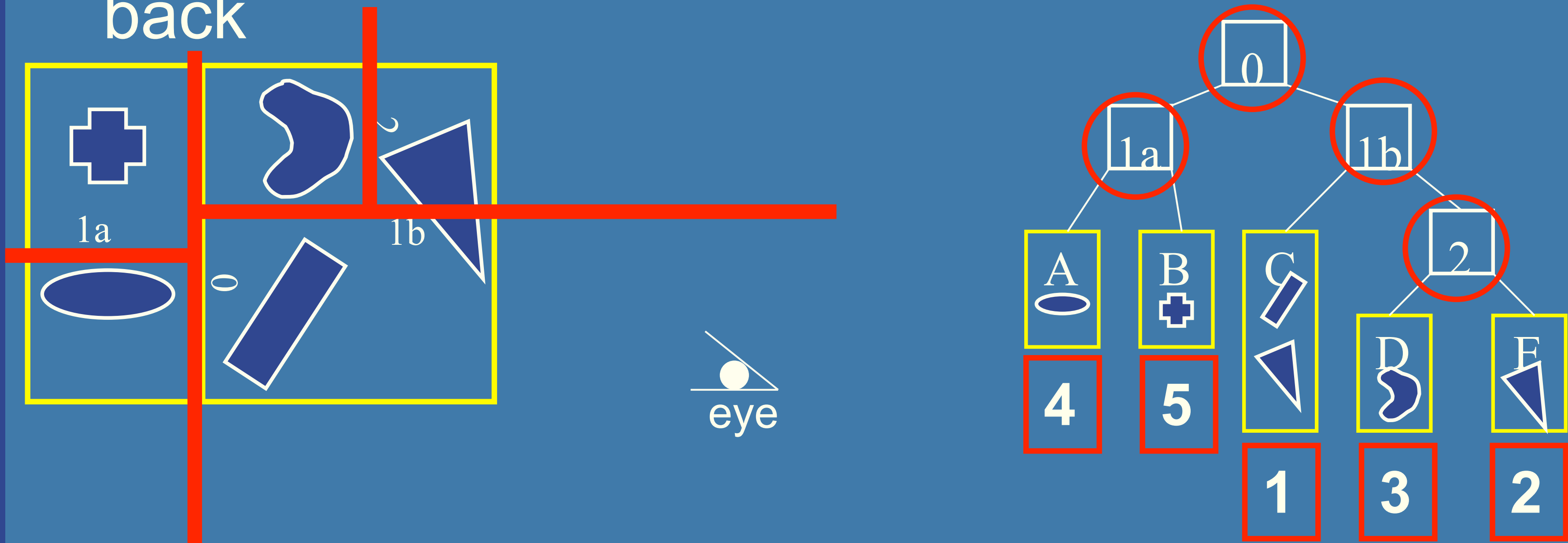


Called TOP-DOWN method

More complex for other BVs

Axis-aligned BSP tree
Rough sorting

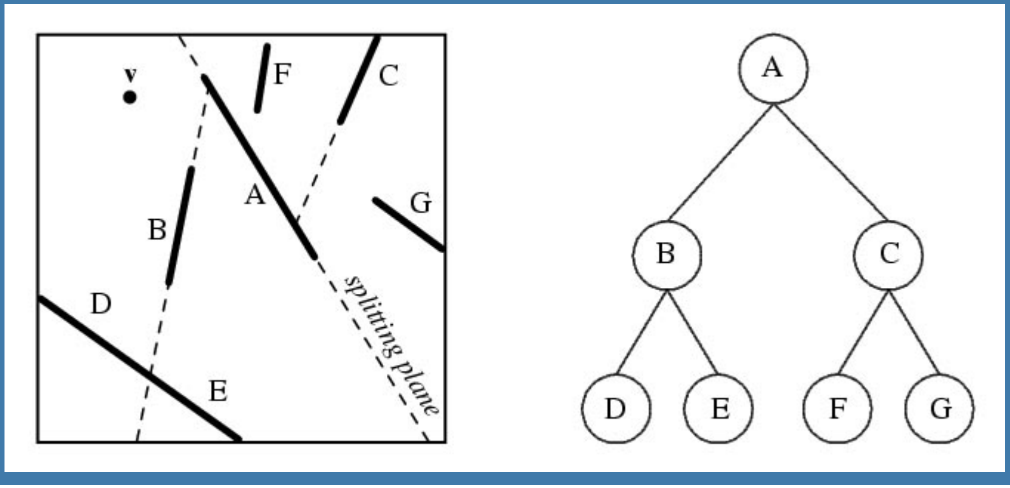
- Test the planes against the point of view
- Test recursively from root
- Continue on the ”hither” side to sort front to back



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

Polygon-aligned BSP tree

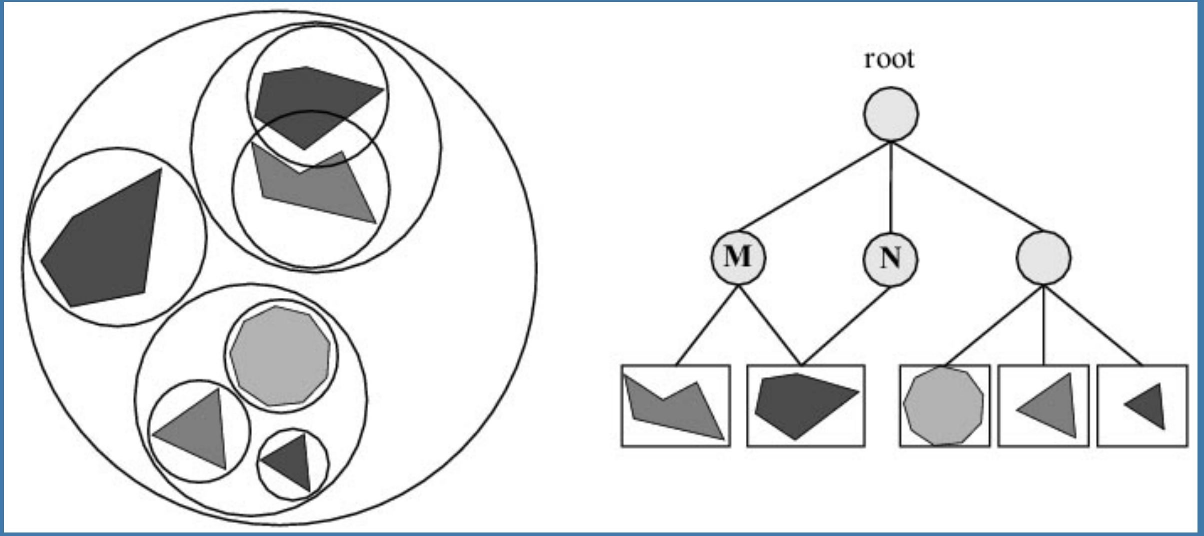
- Allows exact sorting
- Very similar to axis-aligned BSP tree
  - But the splitting plane are now located in the planes of the triangles



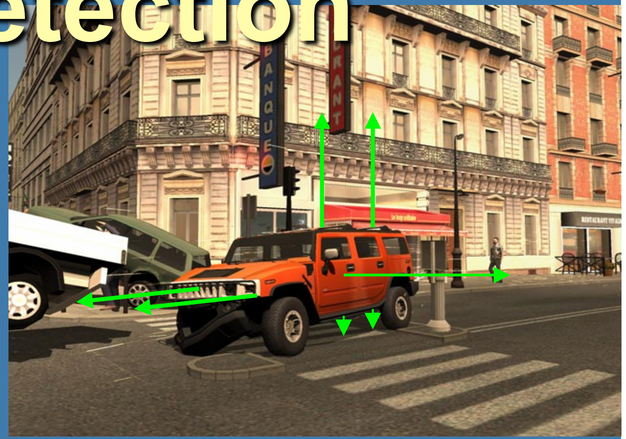
Know how to build it and how to traverse back-to-front or front-to-back

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
  - Textures
  - Transforms
  - 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