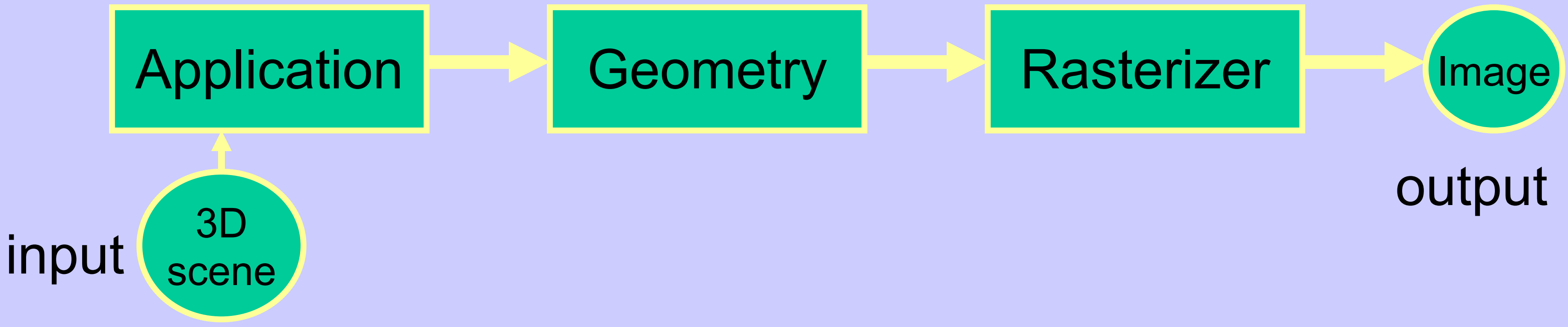
Half Time Wrapup Slides

Lecture 1

- Real-time Graphics pipeline
- Application-, geometry-, rasterization stage
- Modelspace, worldspace, viewspace, clip space, screen space
- Z-buffer
- Double buffering
- Screen tearing

Lecture 1: Real-time Rendering
The Graphics Rendering Pipeline

- Three conceptual stages of the pipeline:
  - Application (executed on the CPU)
    - logic, speed-up techniques, animation, etc…
  - 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 per-vertex parameters (colors, texcoords etc) over triangle
    - Z-buffering, fragment merge (i.e., blending), stencil tests…



Rendering Pipeline and Hardware

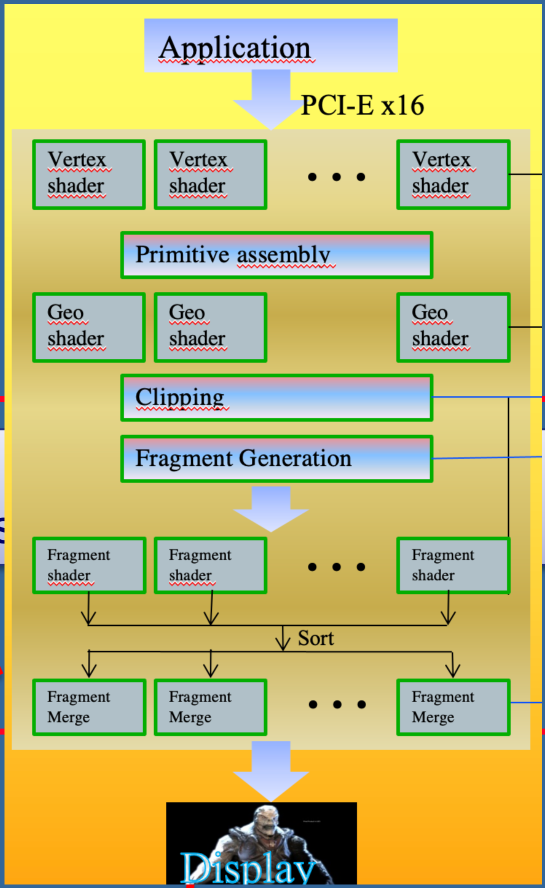
CPU

GPU

Appli-
cation
Stage

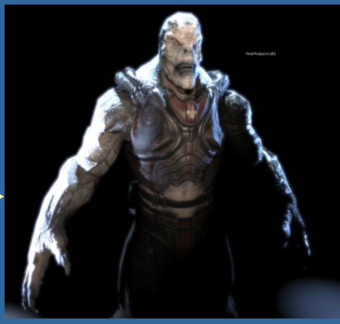
Geometry Stage

Ras



HARDWARE

Vertex shader → Geometry shader → Clipping → Screen mapping → Triangle Setup → Triangle Traversal → Pixel shader → Merger →



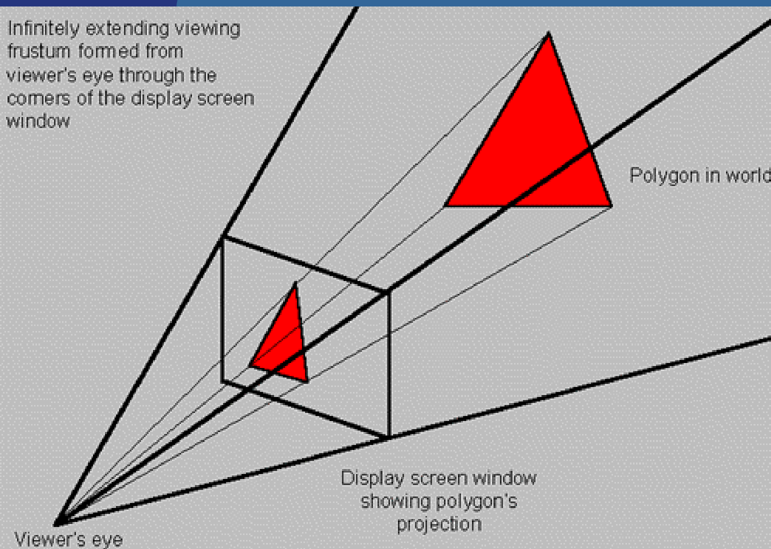
Display

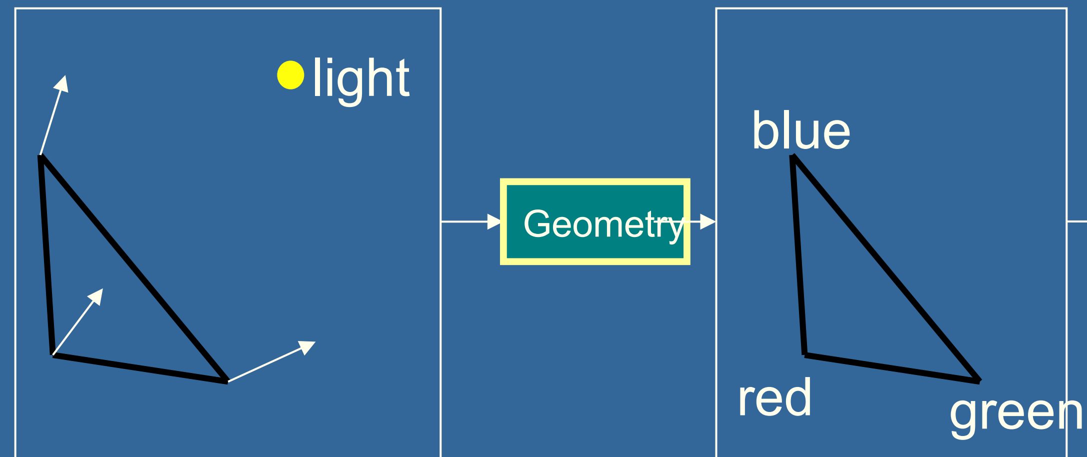
Hardware design

Geometry Stage

Vertex shader:

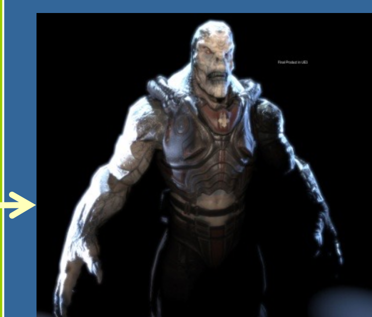
- Lighting (colors)
- Screen space positions





HARDWARE

Vertex shader → Geometry shader → Clipping → Screen mapping → Triangle Setup → Triangle Traversal → Pixel shader → Merger → Display

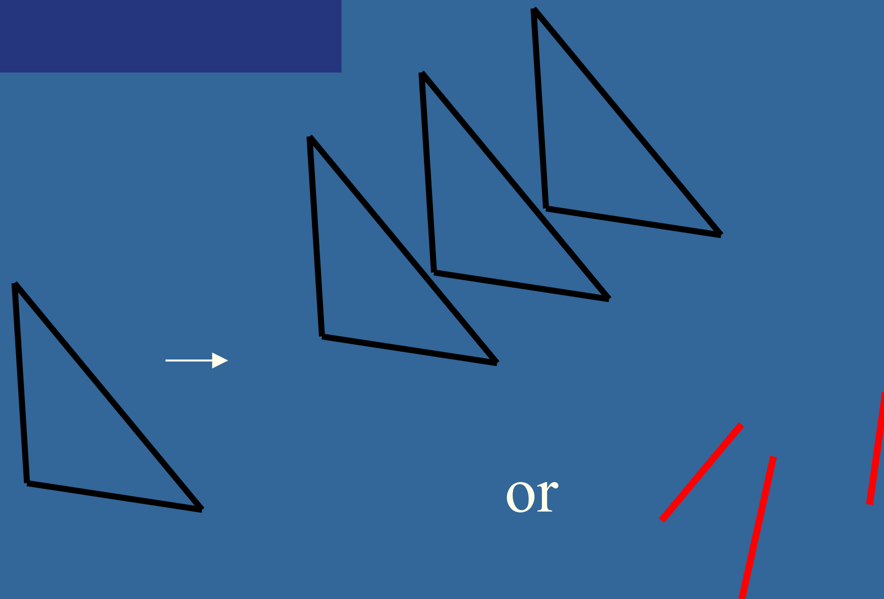


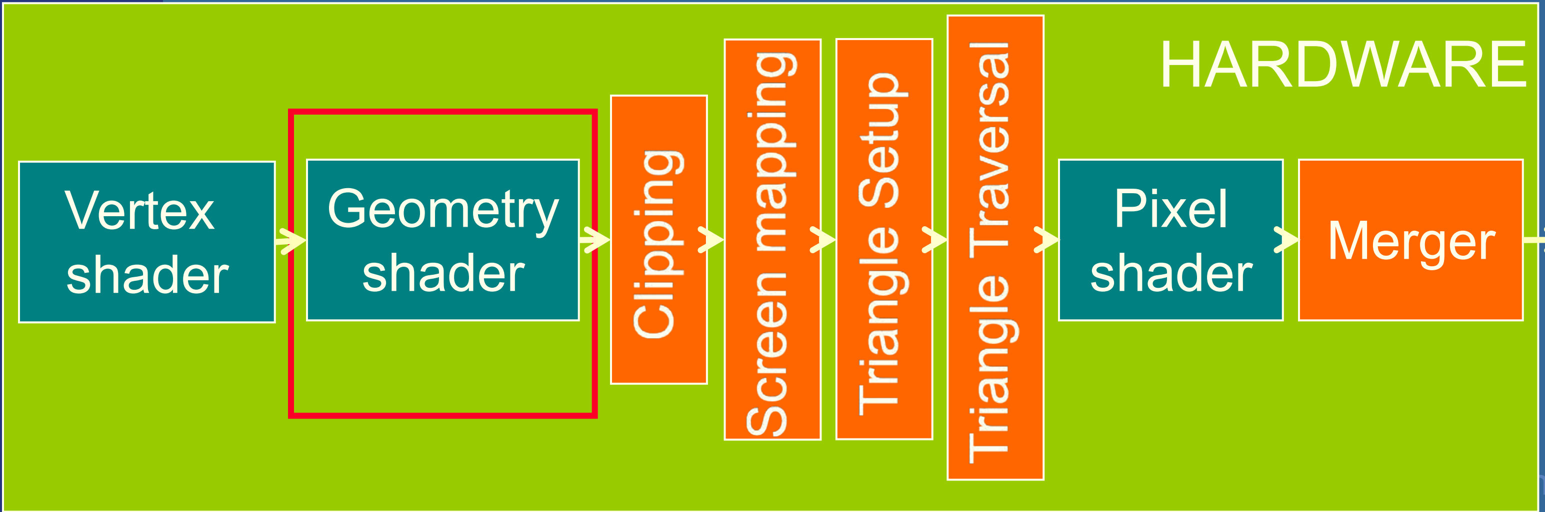
Hardware design

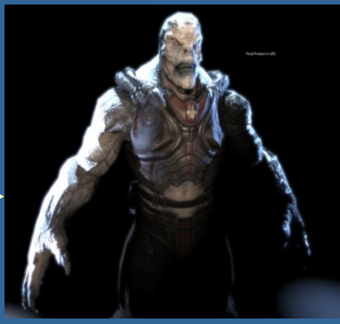
Geometry Stage

Geometry shader:

- One input primitive
- Many output primitives





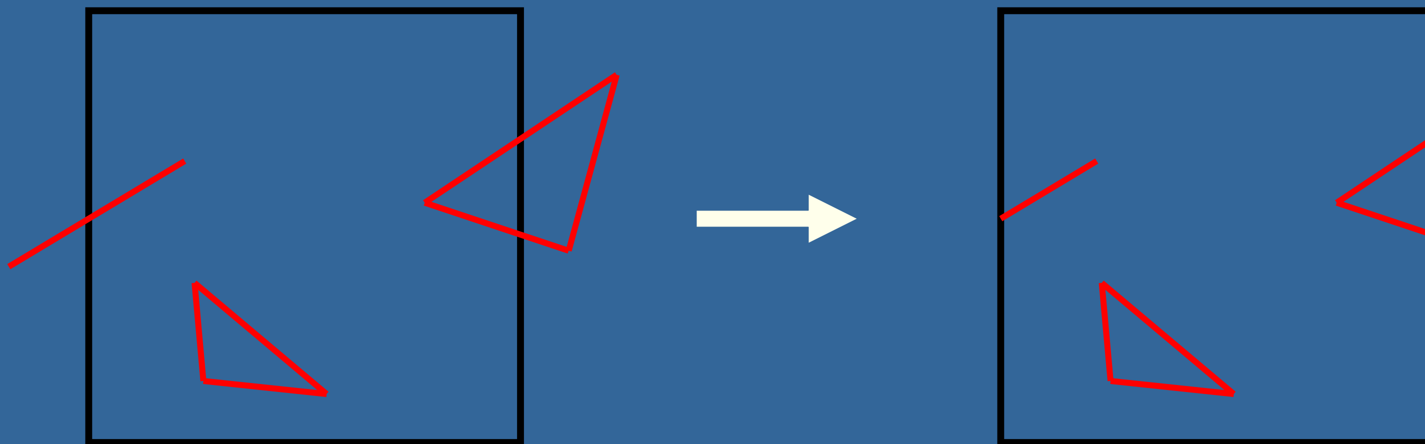


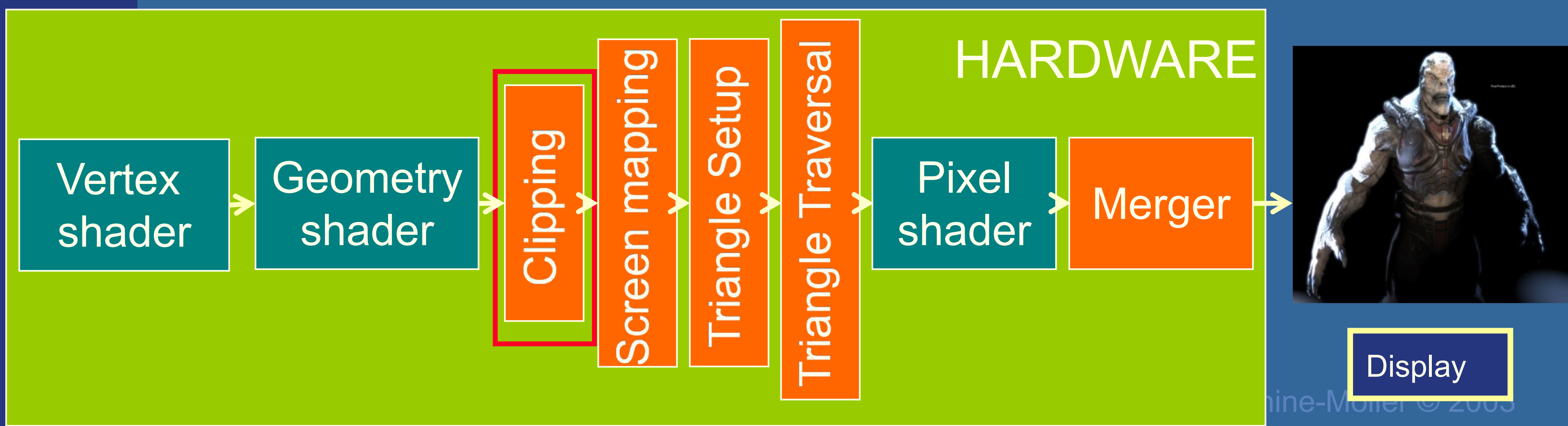
Display

Hardware design

Geometry Stage

Clips triangles against the unit cube (i.e., ”screen borders”)



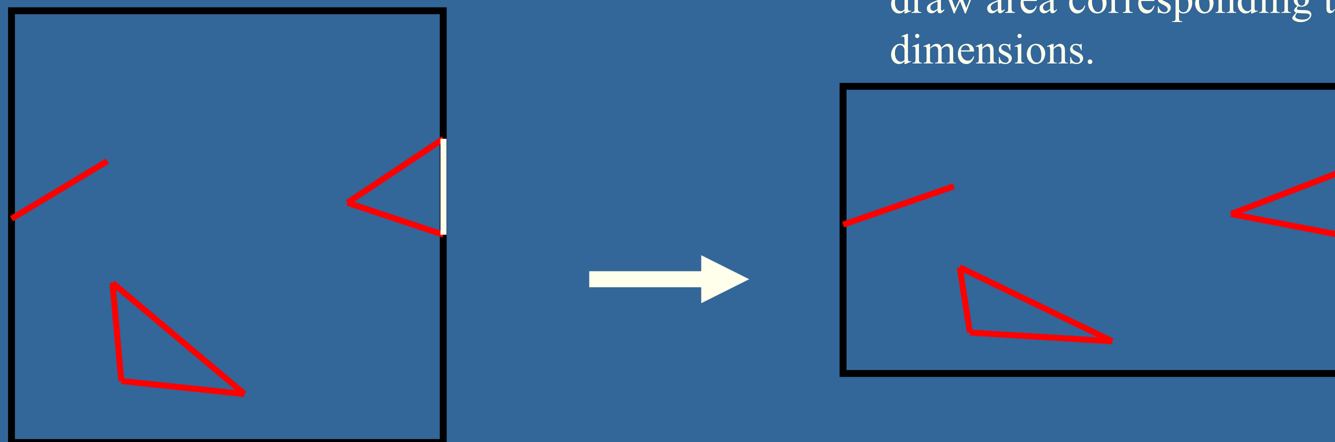


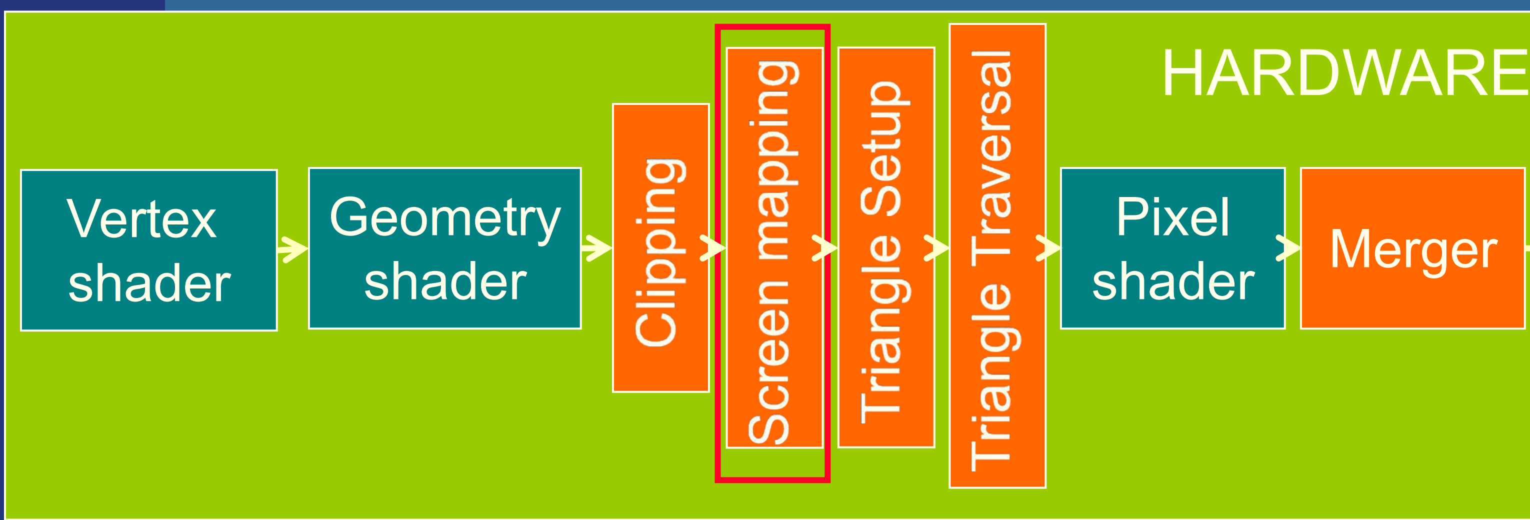
Hardware design

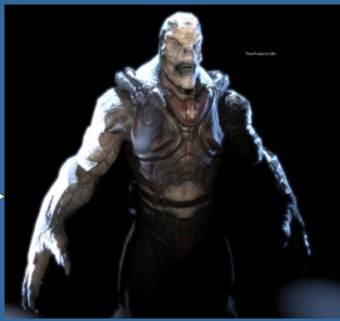
Rasterizer Stage

Maps window size to unit cube

Geometry stage always operates inside a unit cube [-1,-1,-1]-[1,1,1]
Next, the rasterization is made against a draw area corresponding to window dimensions.





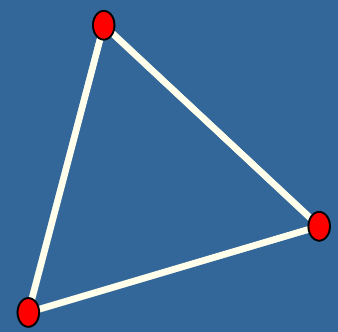


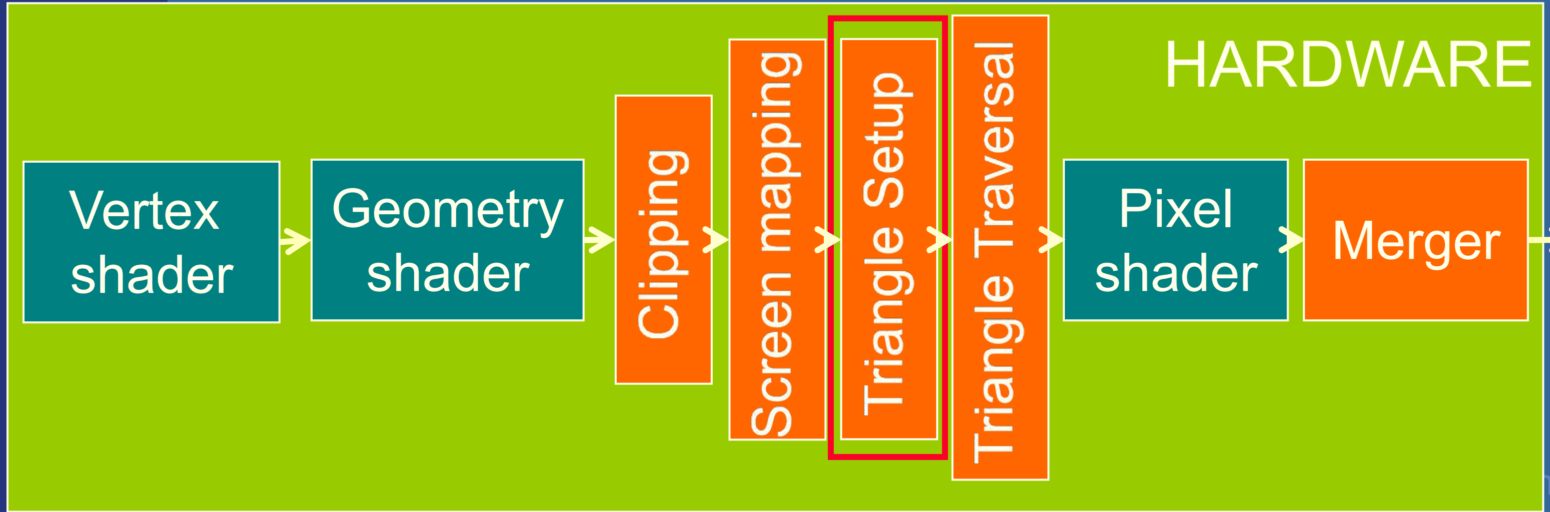
Display

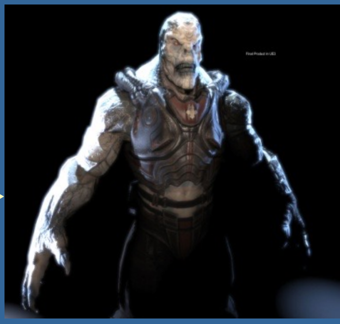
Hardware design

Rasterizer Stage

Collects three vertices into one triangle





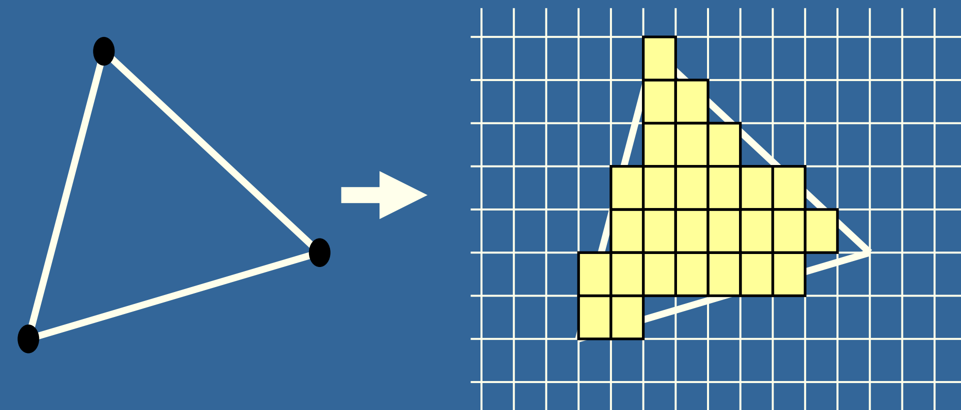


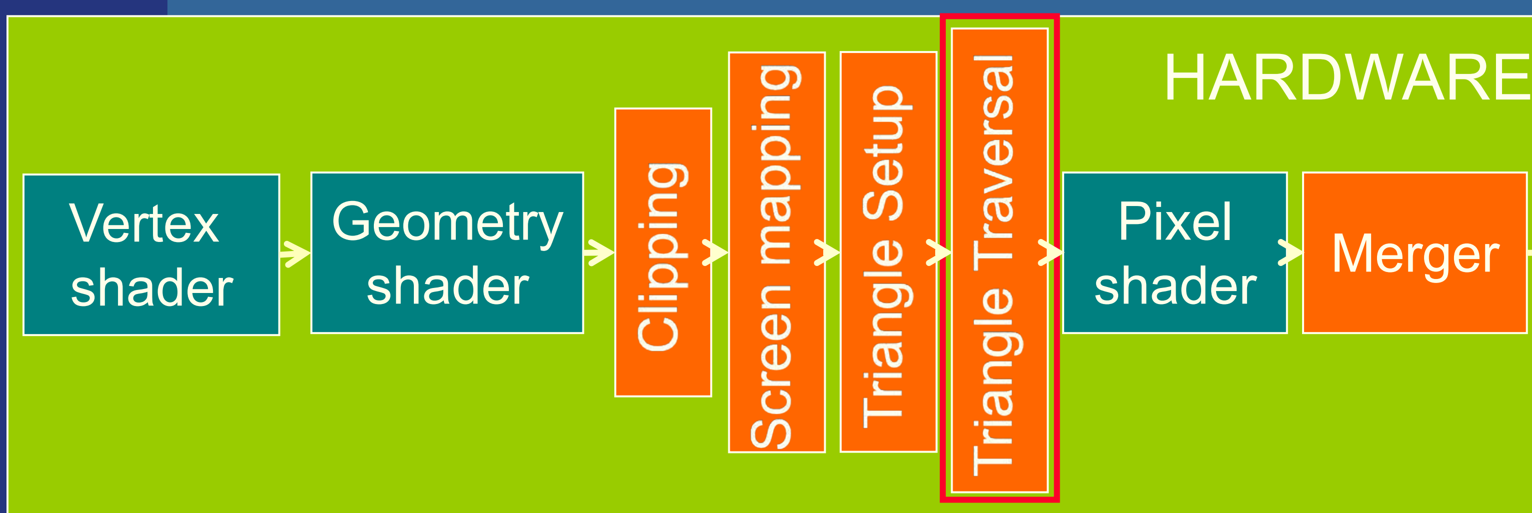
Display

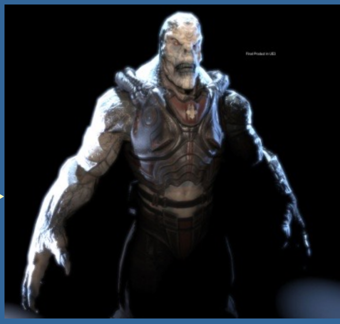
Hardware design

Rasterizer Stage

Creates the fragments/pixels for the triangle



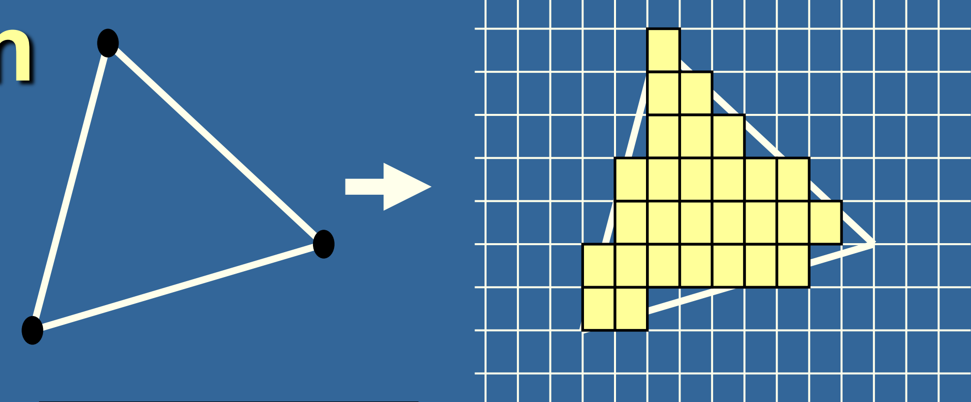


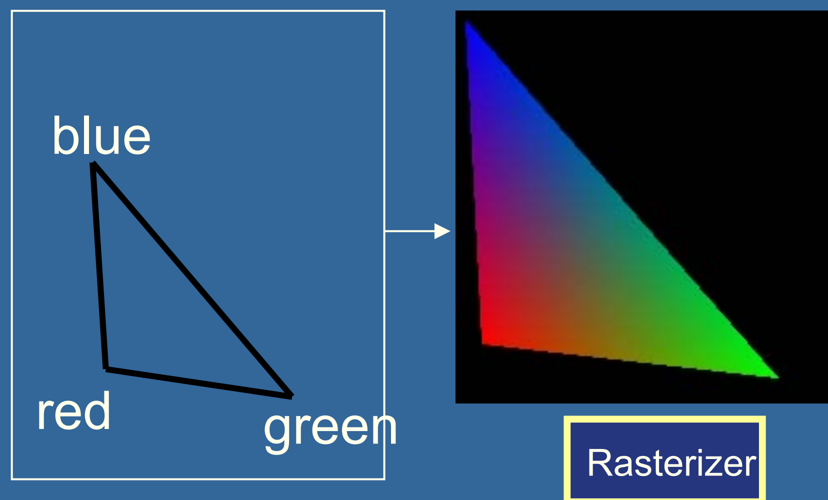


Display

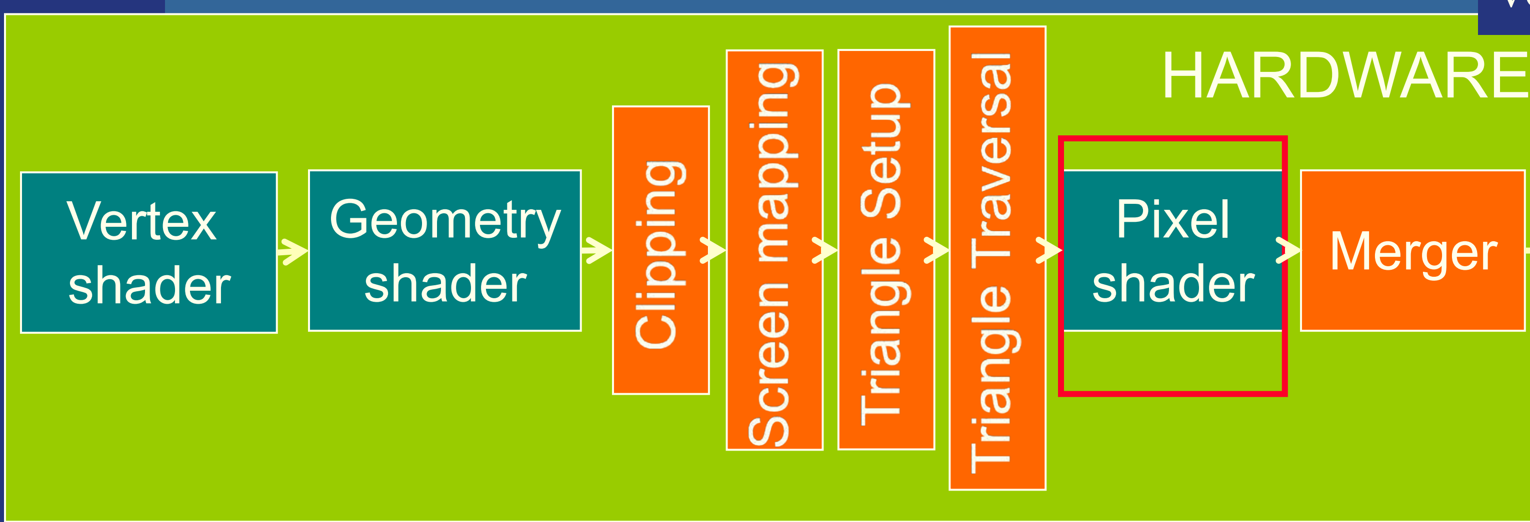
Hardware design

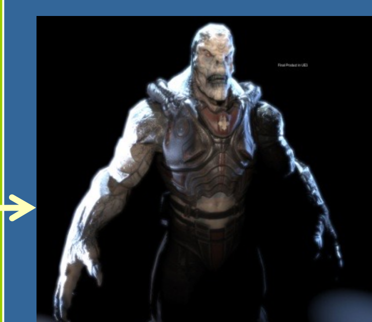
Rasterizer Stage





Pixel Shader:
Compute color using:
- Textures
- Interpolated data (e.g. Colors + normals) from vertex shader





Display

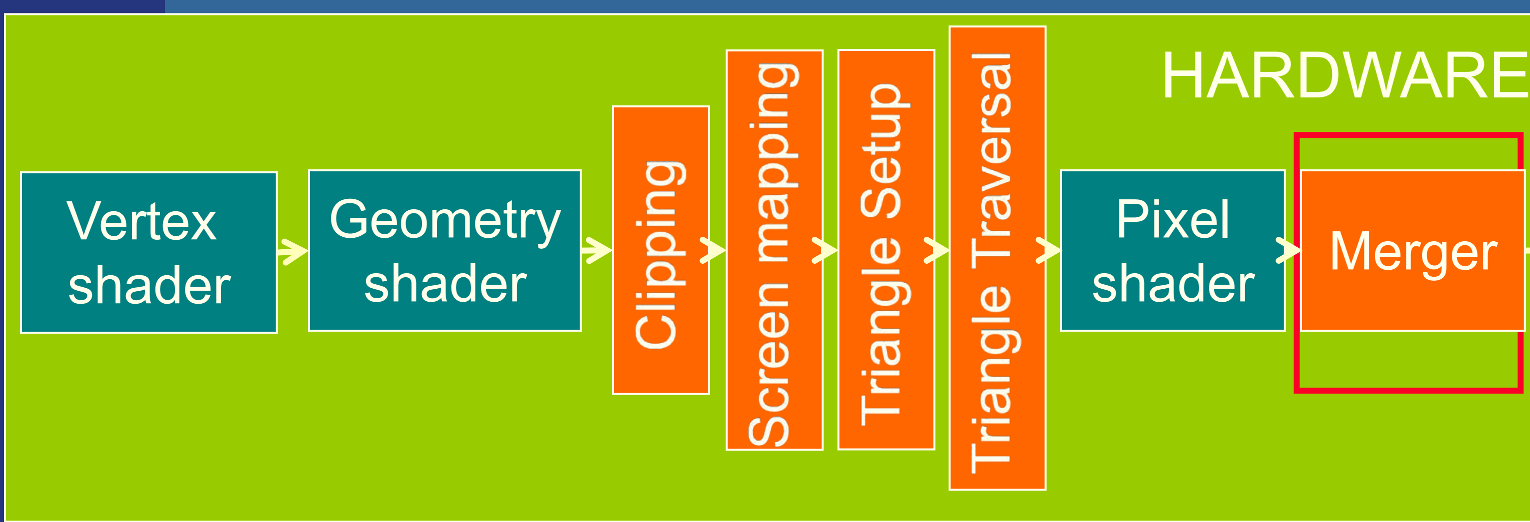
Hardware design

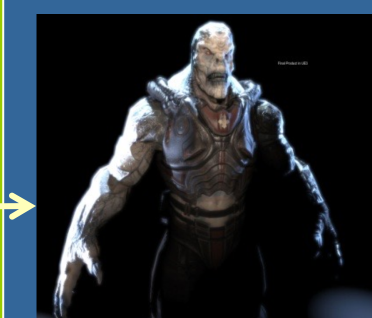
Rasterizer Stage

The merge units update the frame buffer with the pixel’s color

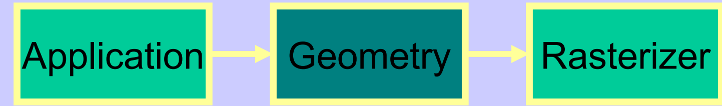
Frame buffer:

- Color buffers
- Depth buffer
- Stencil buffer

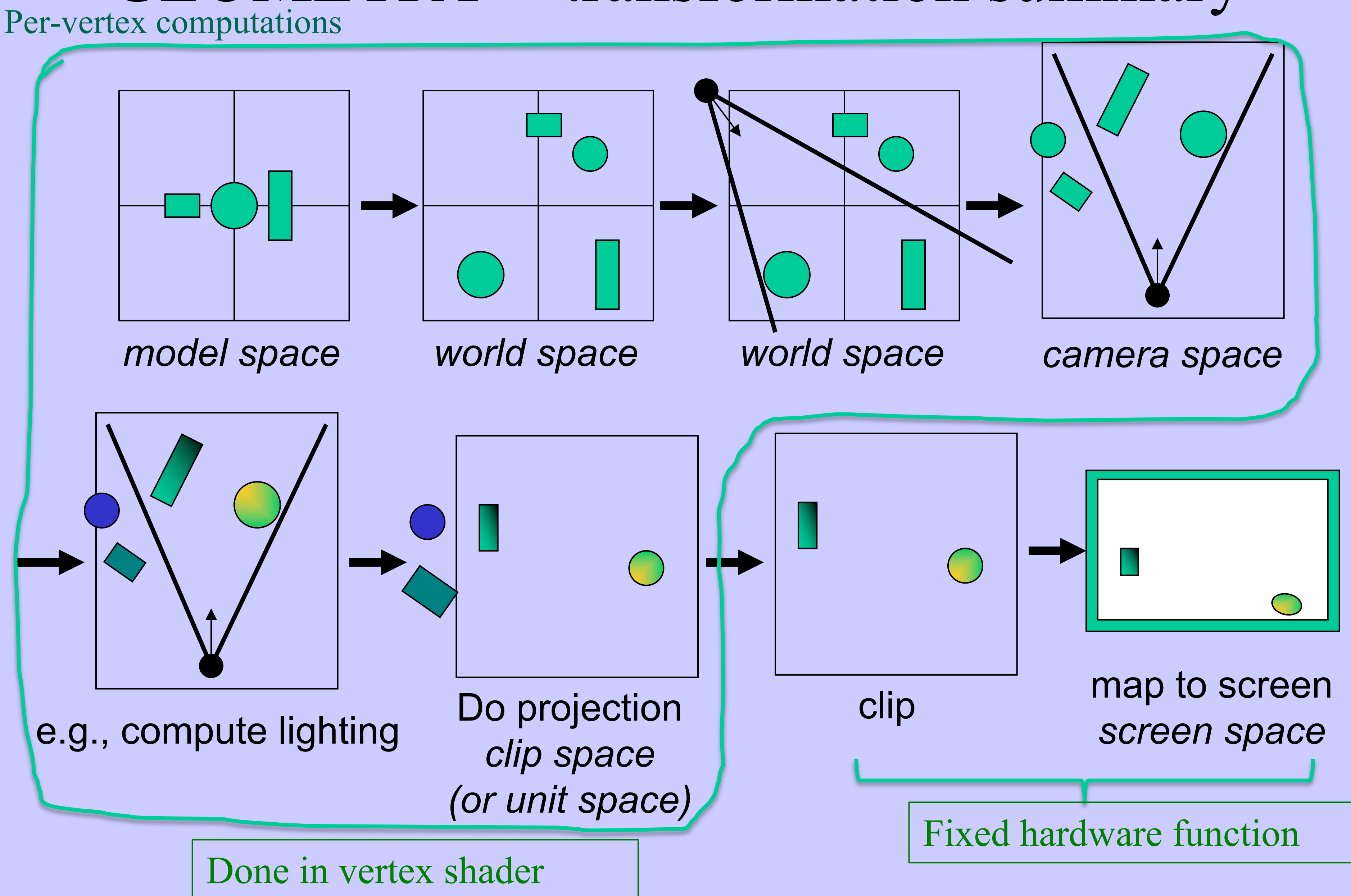




Display

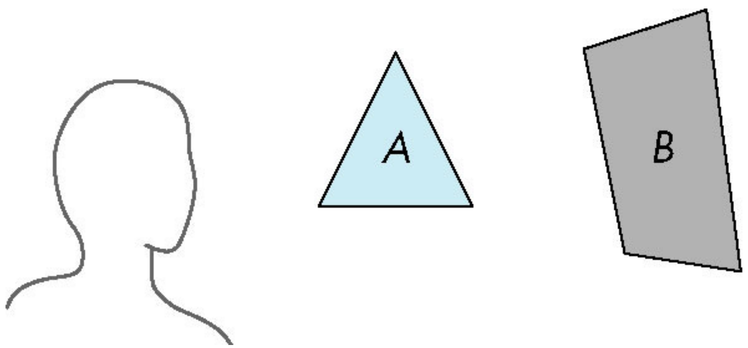


GEOMETRY – transformation summary

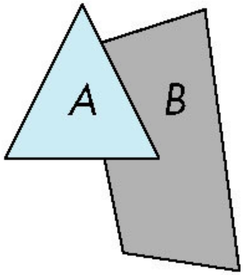


Painter's Algorithm

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



B behind A as seen by viewer



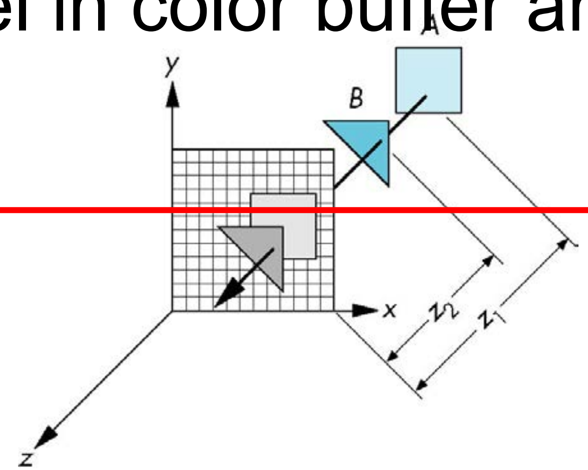
Fill B then A

- 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.

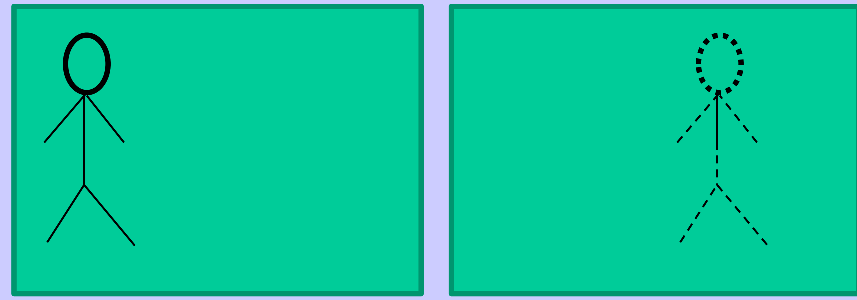
z-Buffer Algorithm

- 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



Also know double buffering!

The RASTERIZER double-buffering



Front buffer (rgb color buffer)

Last fully finished drawn frame.

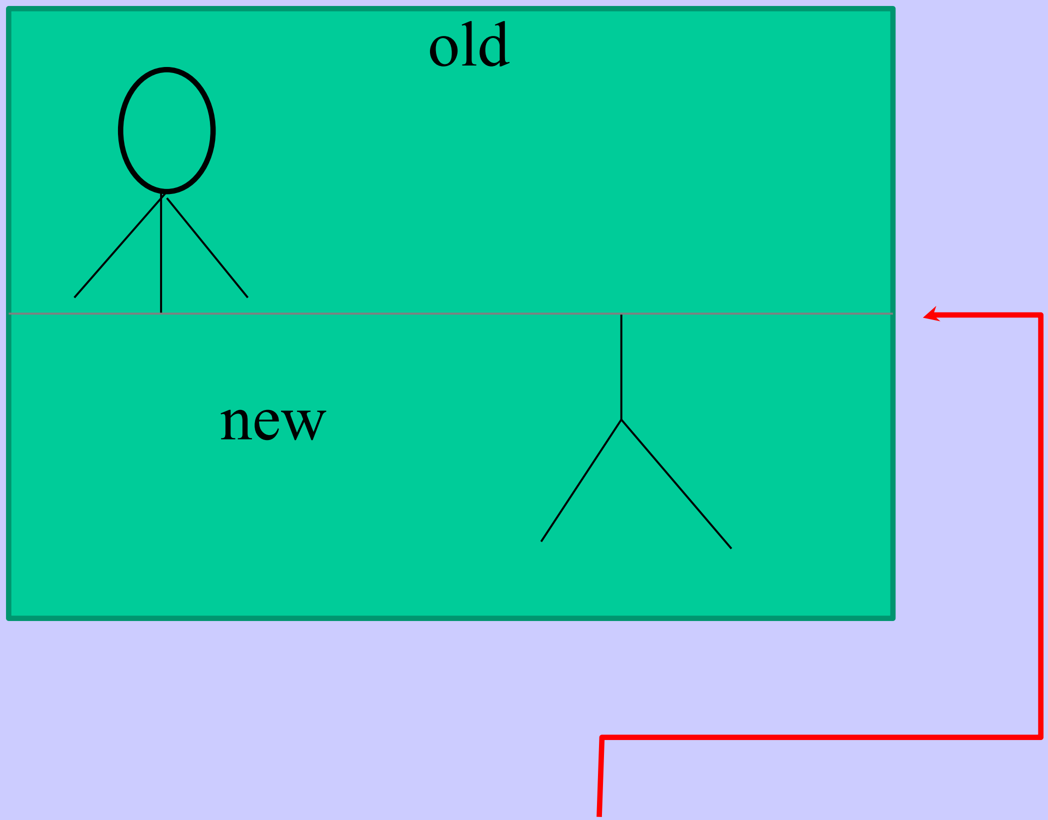
Back buffer (rgb color buffer)

Color buffer we draw to. Not displayed yet.

- We do not want to show the image until its drawing is finished.
- The front buffer is displayed
- The back buffer is rendered to
- When new image has been created in back buffer, swap the Front-/Back-buffer pointers.
- Use vsynch or screen tearing will occur…
  i.e., when the swap happens in the middle of the screen with respect to the screen refresh rate.

The RASTERIZER double-buffering – screen tearing

Application → Geometry → Rasterizer



Example if the swap happens here (w.r.t the screen refresh rate).

Screen Tearing

Swapping back/front buffers

vblank

Screen tearing is solved by using V-Sync.

V-Sync: swap front/back buffers during vertical blank (vblank) instead.

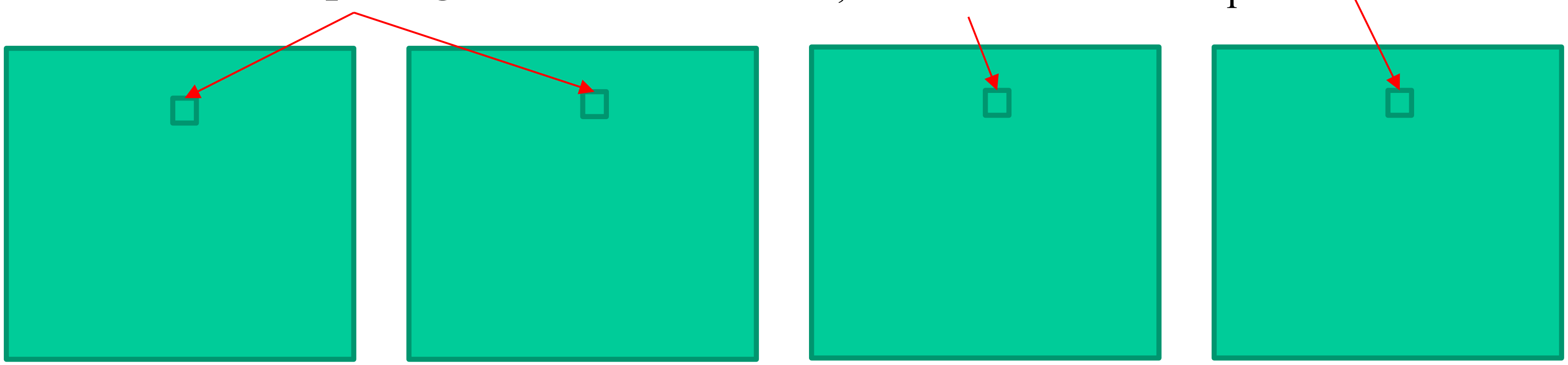
The default frame buffer:

Typically: Front + Back color buffers + Z buffer + (Stencil buffer)

Stores rgb(a) value per pixel.
Default: 8 bits per r,g,b channel.

Stores fragment's depth value per pixel, typically: (16), 24, or 32 bits.

Stencil buffer can be asked for. 8-bits per pixel.



Front buffer (rgb color buffer)

Last fully finished drawn frame.
Is displayed.

Back buffer (rgb color buffer)

Color buffer we draw to.
Not displayed yet.

Z buffer (depth)

To resolve visibility

Stencil buffer

Used for masking rendering to only where pixel's stencil value = some specific value.

Lecture 2: Transforms

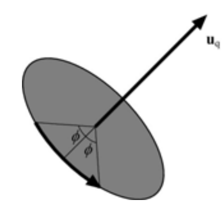
- Transformation pipeline: ModelViewProjection matrix
- Scaling, rotations, translations, projection
- Cannot use same matrix to transform normals

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

$(M^{-1})^T=M$ if rigid-body transform

- Homogeneous notation
- Rigid-body transform, Euler rotation (head,pitch,roll)
- Change of frames
- 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 $\mathbf{p}$

- Understand the simple DDA algorithm
- Bresenhams line-drawing algorithm

Transformation Pipeline

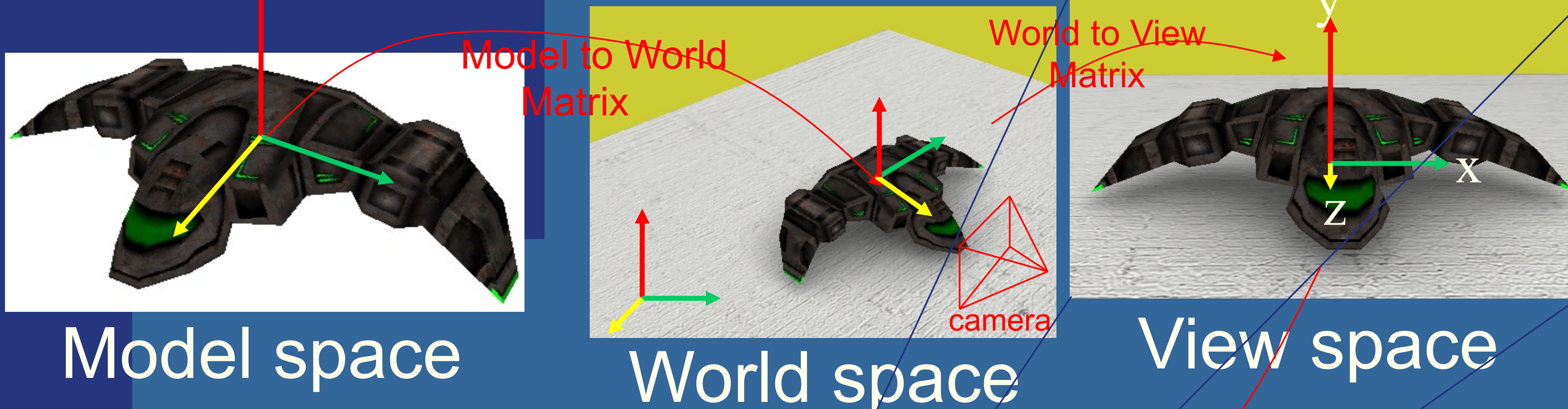
Clip space: clipping is nowadays typically done in homogeneous space. However, it used to be done in unit-cube space. Both terminologies are still used.

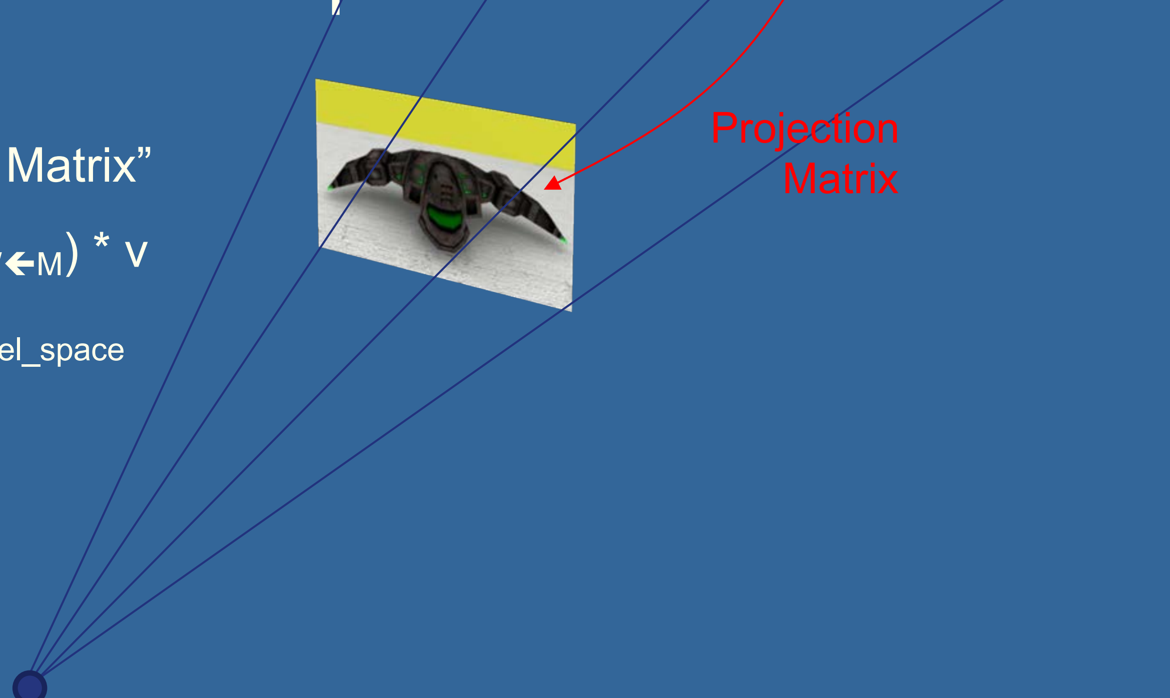
object space → **vertex** → *Modelview Matrix* → *View/Eye/Camera space* → *Projection Matrix* → *Homogeneous coord. space* → *Perspective Division* → *Unit-cube space / Normalized device coords* → *Viewport Transform* → *Window coords. Screen space*

Done by the vertex shader:

```
gl_Position = modelViewProjectionMatrix*vec4(vertex,1);
```

OpenGL | Geometry stage | done on GPU





ModelViewMtx = ”Model to View Matrix”

$\text{ModelViewMtx} * v = (M_{V\leftarrow W} * M_{W\leftarrow M}) * v$

$v_{view\_space} = \text{ModelViewMtx} * v_{model\_space}$

Full projection:

$V_{clip\_space} = \text{projectionMatrix} * \text{ModelViewMatrix} * v_{model\_space}$

Or simply: $v_{clip\_space} = M_{ModelViewProjection} * v$, where $M_{ModelViewProjection} = \text{projectionMatrix} * \text{ModelViewMatrix}$

Homogeneous notation

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

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

Rotation part

Translation part

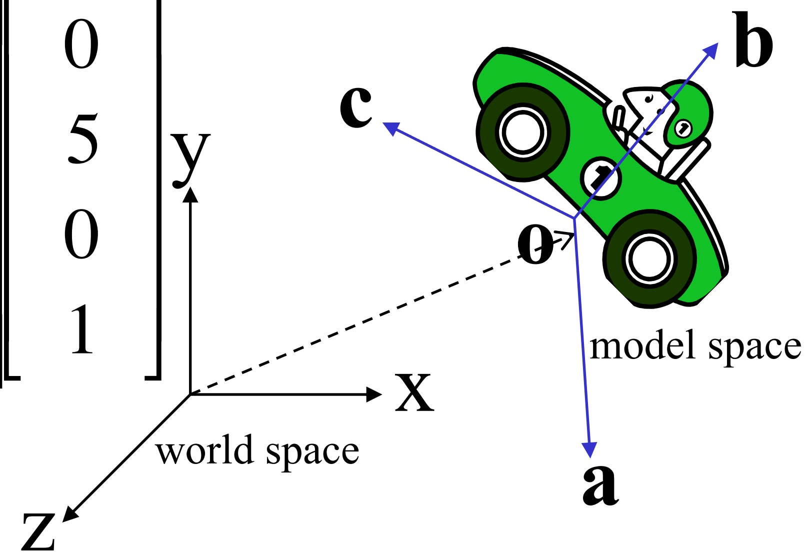
- A vector (direction): $\mathbf{d} = \begin{pmatrix} d_x & d_y & d_z & 0 \end{pmatrix}^T$
- Translation of vector: $\mathbf{Td} = \mathbf{d}$

Change of Frames

- How to get the $M_{\text{model-to-world}}$ matrix:

$\mathbf{P} = (0,5,0,1)$

$$M_{\text{model-to-world}} = \begin{bmatrix} a_x & b_x & c_x & o_x \\ a_y & b_y & c_y & o_y \\ a_z & b_z & c_z & o_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 5 \\ 0 \\ 1 \end{bmatrix}$$



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

(Both coordinate systems are right-handed)

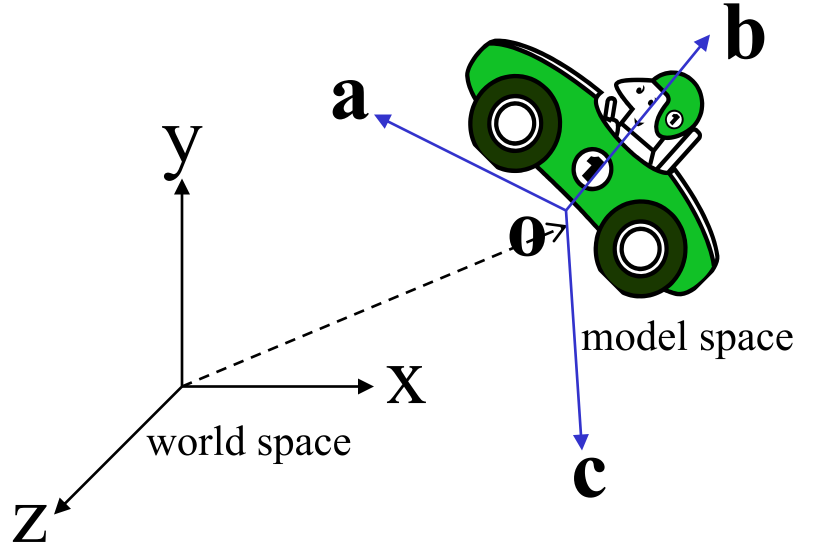
E.g.: $\mathbf{p}_{\text{world}} = M_{m\to w}\,\mathbf{p}_{\text{model}} = M_{m\to w}\,(0,5,0,1)^T = 5\,\mathbf{b}\;\; (+\,\mathbf{o})$

Same example, just explained differently:

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* (instead of modelspace) is thus $[a_x\ b_x\ c_x]\mathbf{p}$.

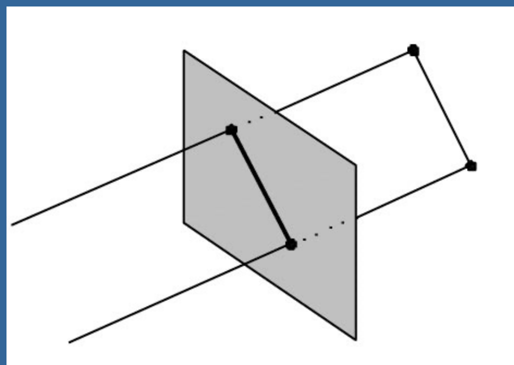
The y-coord for **p** in world space is thus $[a_y\ b_y\ c_y]\mathbf{p}$.

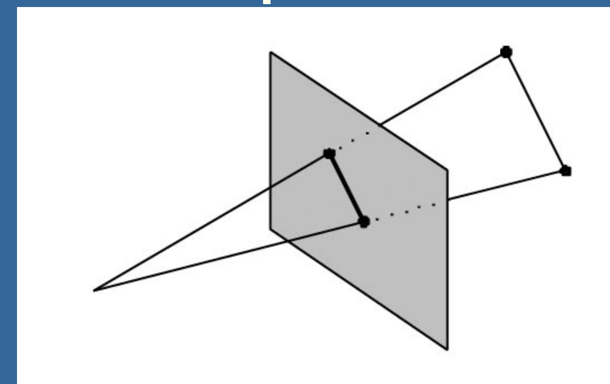
The z-coord for **p** in world space is thus $[a_z\ b_z\ c_z]\mathbf{p}$.

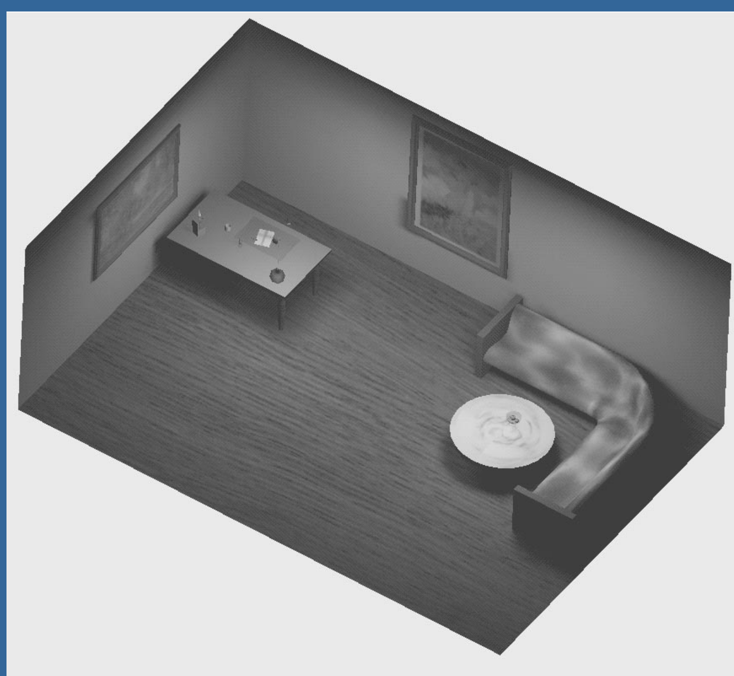
With the translation **o** we get $\mathbf{p}_{\text{worldspace}} = M_{\text{model-to-world}}\, \mathbf{p}_{\text{modelspace}}$

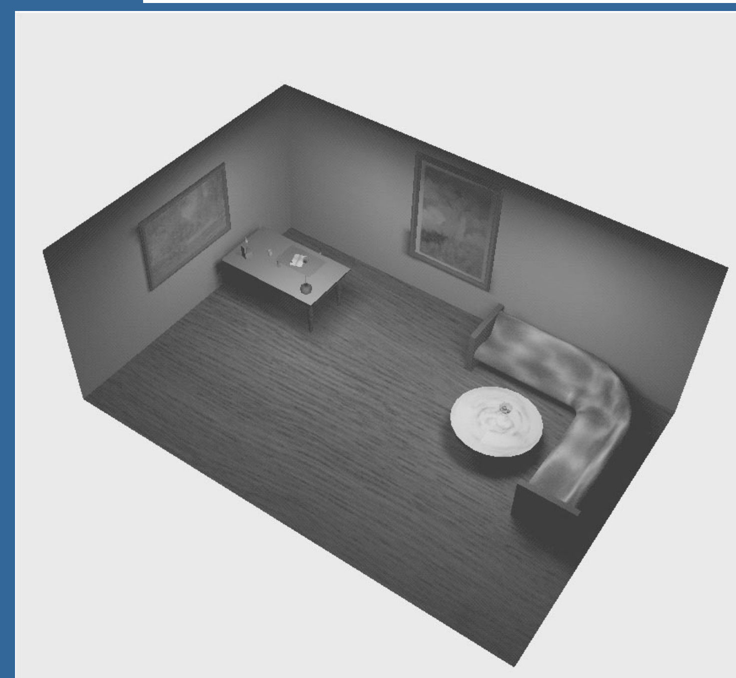
Projections

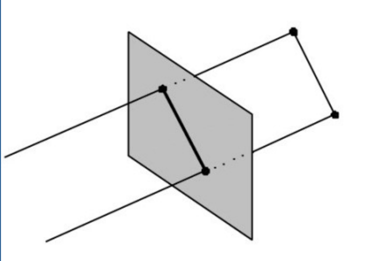
- Orthogonal (parallel) and Perspective







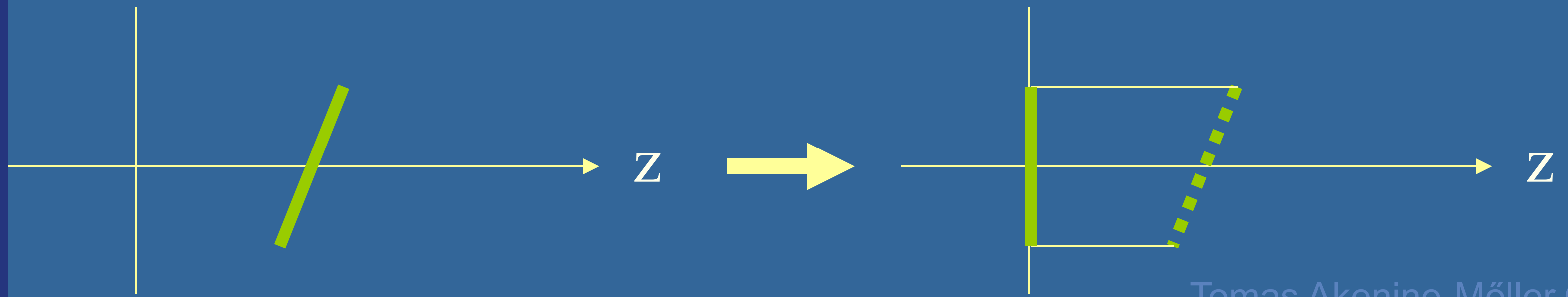




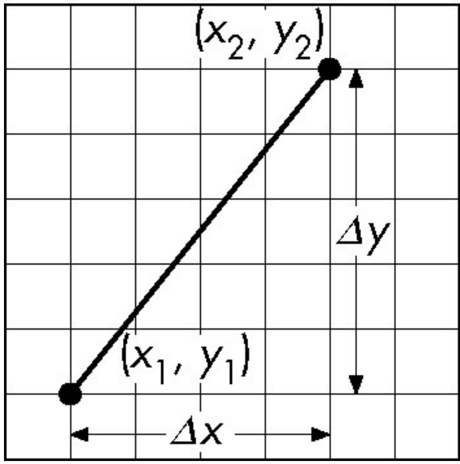
Orthogonal projection

- 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

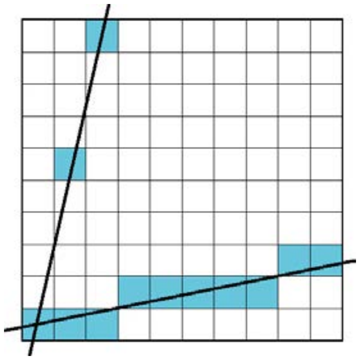


- Digital Differential Analyzer
  - DDA was a mechanical device for numerical solution of differential equations
  - Line y=kx+ m satisfies differential equation

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

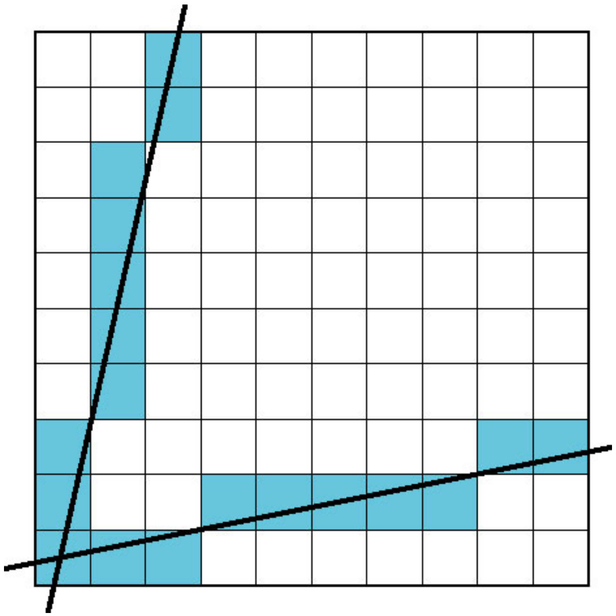
```
y=y1;
For(x=x1; x<=x2,ix++) {
  write_pixel(x, round(y), line_color)
  y+=k;
}
```

Using Symmetry



Otherwise we get problem for steep slopes

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

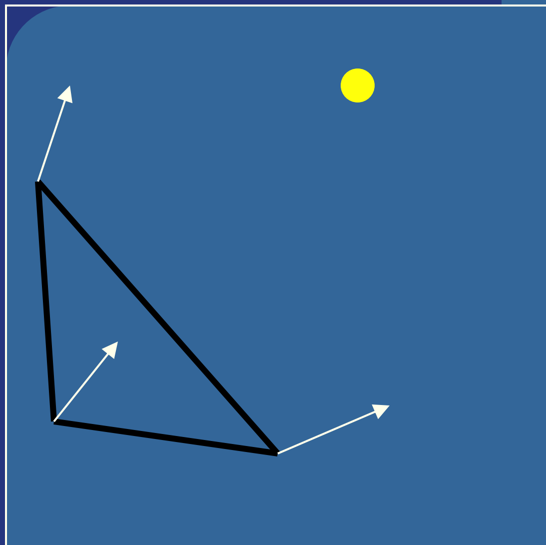


Very Important!

- 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.

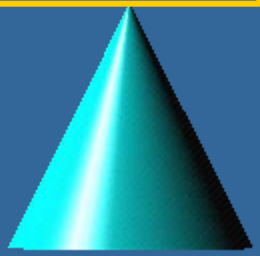
Lighting



Light:

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

DIFFUSE SPECULAR	Base color 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”

The ambient/diffuse/specular/emission model

- Summary of formulas:

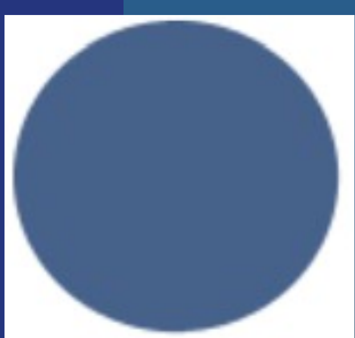
Ambient: $\mathbf{i}_{\mathrm{amb}} = \mathbf{m}_{\mathrm{amb}}\, \mathbf{l}_{\mathrm{amb}}$

Diffuse: $(\boldsymbol{n} \cdot \boldsymbol{l})\, \mathbf{m}_{\mathrm{diff}}\, \mathbf{l}_{\mathrm{diff}}$

Specular:

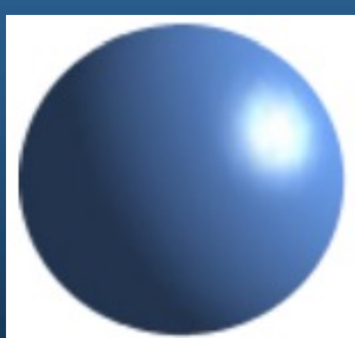
- Phong: $(\boldsymbol{r} \cdot \boldsymbol{v})^{shininess}\, \mathbf{m}_{\mathrm{spec}}\, \mathbf{l}_{\mathrm{spec}}$
- Blinn: $(\boldsymbol{n} \cdot \boldsymbol{h})^{shininess}\, \mathbf{m}_{\mathrm{spec}}\, \mathbf{l}_{\mathrm{spec}}$

Emission: $\mathbf{m}_{\mathrm{emission}}$

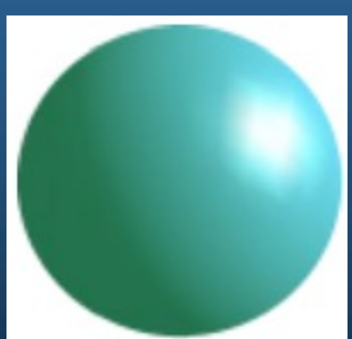


Ambient

Amb + Diff



Amb + Diff + Spec



Amb + Diff + Spec + Em

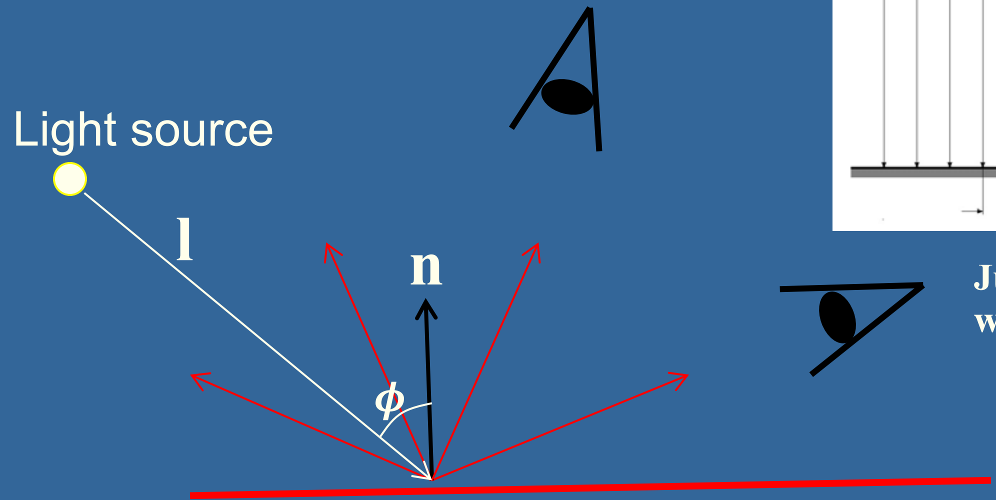
The ambient/diffuse/specular/emission model

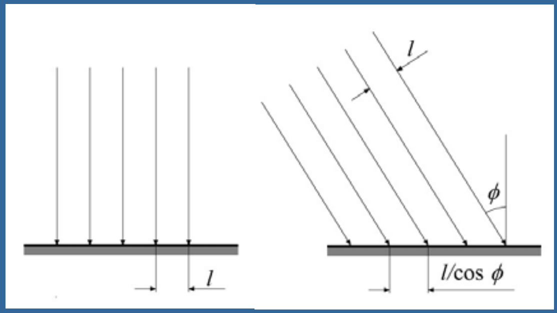
- The most basic real-time model:
- Light interacts with material and change color at bounces:

$$\mathbf{outColor}_{rgb} \sim \mathbf{material}_{rgb} \otimes \mathbf{lightColor}_{rgb}$$

- Ambient light: incoming background light from all directions and spreads in all directions (view-independent and light-position independent color)
- **Diffuse** light: the part that spreads equally in **all** directions (view independent) due to that the surface is very **rough** on microscopic level

Amb + Diff





Just scale light intensity with incoming angle

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

$$(\boldsymbol{n}\cdot\boldsymbol{l}) = \boldsymbol{\cos\phi}$$

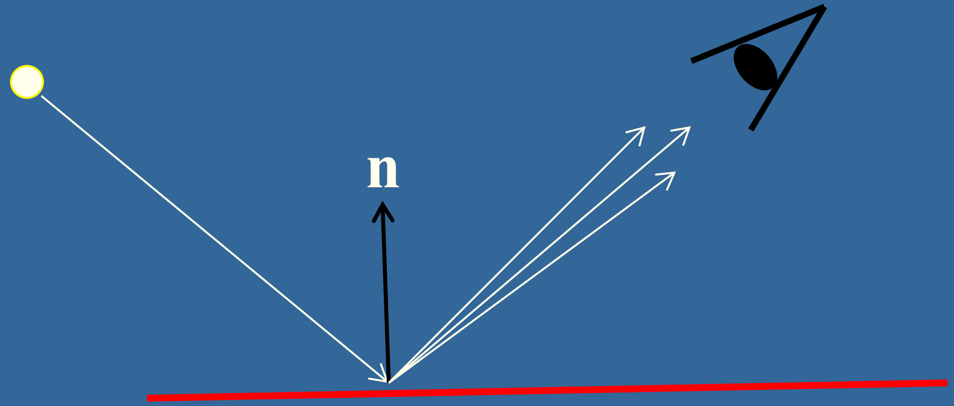
The ambient/diffuse/specular/emission model

- The most basic real-time model:
- Light interacts with material and change color at bounces:

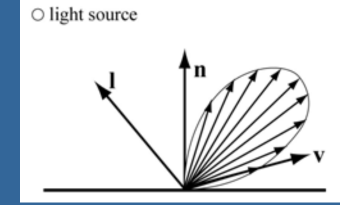
$$\mathbf{outColor}_{rgb} \sim \mathbf{material}_{rgb} \otimes \mathbf{lightColor}_{rgb}$$

- Ambient light: incoming background light from all directions and spreads in all directions (view-independent and light-position independent color)
- Diffuse light: the part that spreads equally in **all** directions (view independent) due to that the surface is very **rough** on microscopic level
- **Specular** light: the part that spreads mostly in the reflection direction (often same color as light source)

Amb + Diff + Spec



Specular: Phong's model

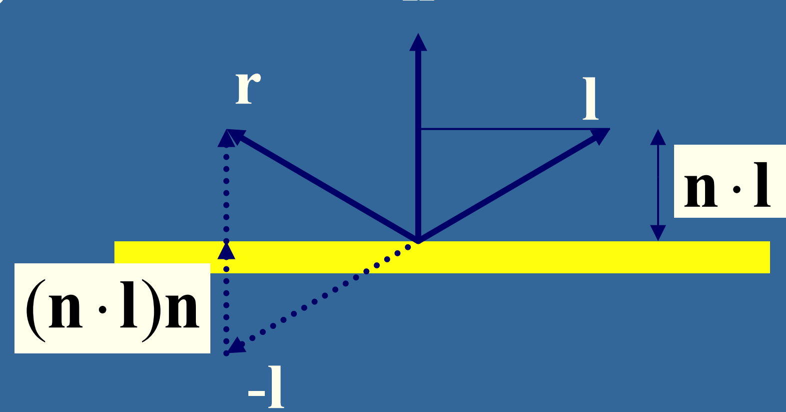


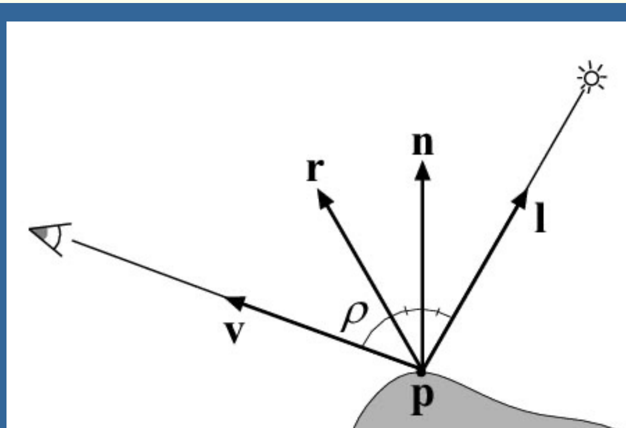
- Phong specular highlight model
- Reflect **l** around **n**:

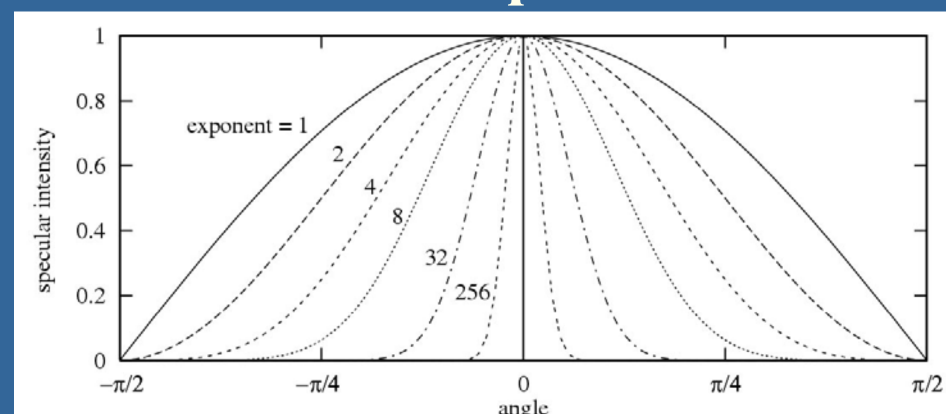
$$\mathbf{r} = -\mathbf{l} + 2(\mathbf{n}\cdot\mathbf{l})\mathbf{n}$$

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

n must be unit vector







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

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

Specular: Blinn's specular highlight model

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

h = (**l**+**v**)/|**l** + **v**|

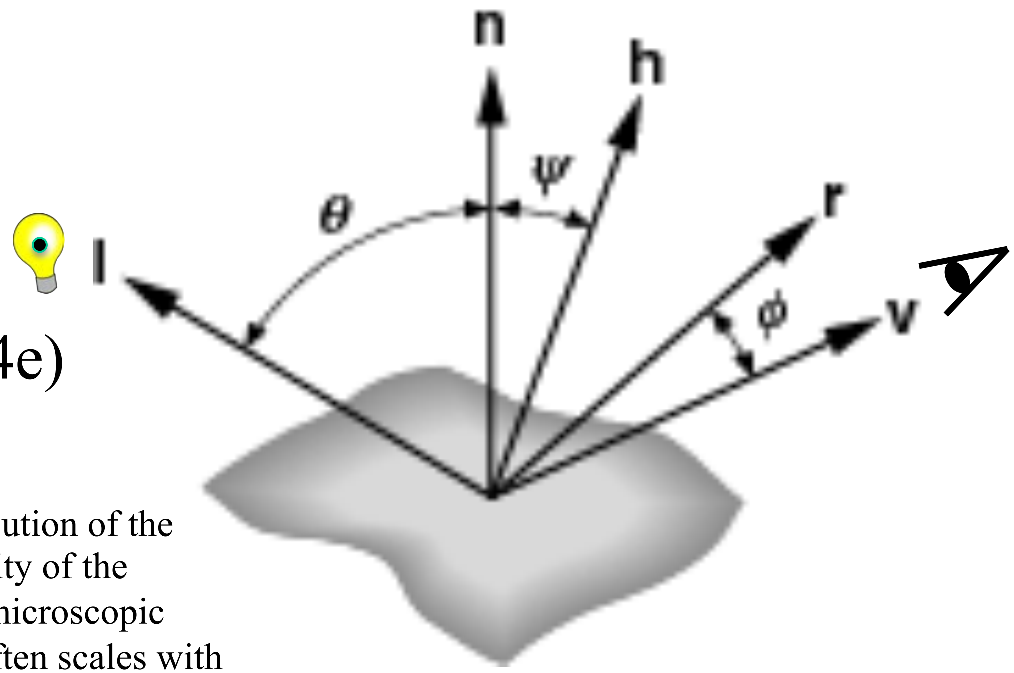
h is halfway between **l** and **v**

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

$\psi = \phi/2$

Must then adjust exponent

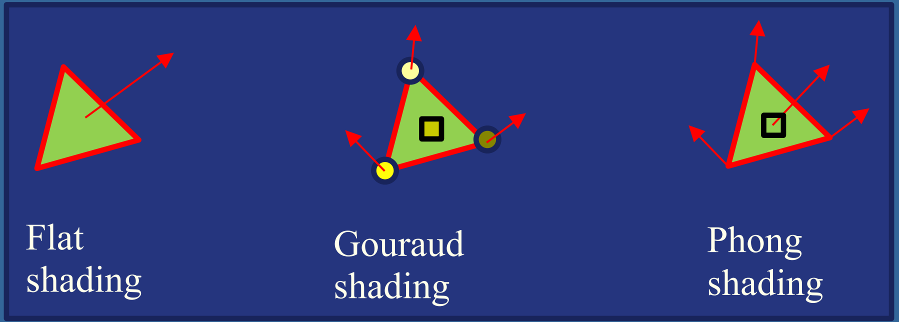
so that $(\mathbf{n}\cdot\mathbf{h})^{e'} \approx (\mathbf{r}\cdot\mathbf{v})^{e}$, ($e' \approx 4e$)



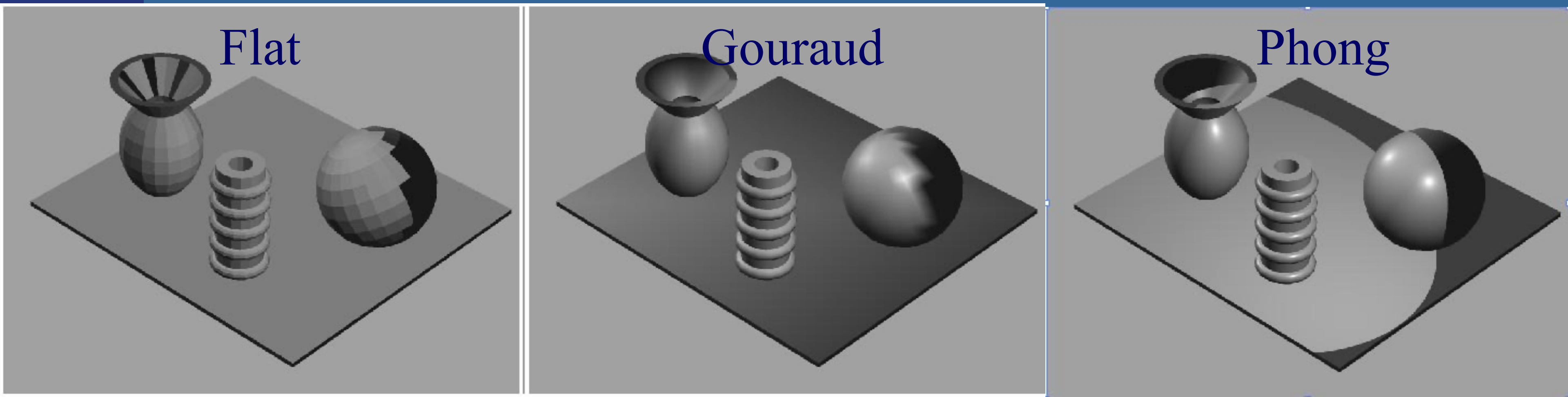
If the surface is rough, there is a probability distribution of the microscopic normals **n**. This means that the intensity of the reflection is decided by how many percent of the microscopic normals are aligned with **h**. And that probability often scales with how close **h** is to the macroscopic surface normal **n**.

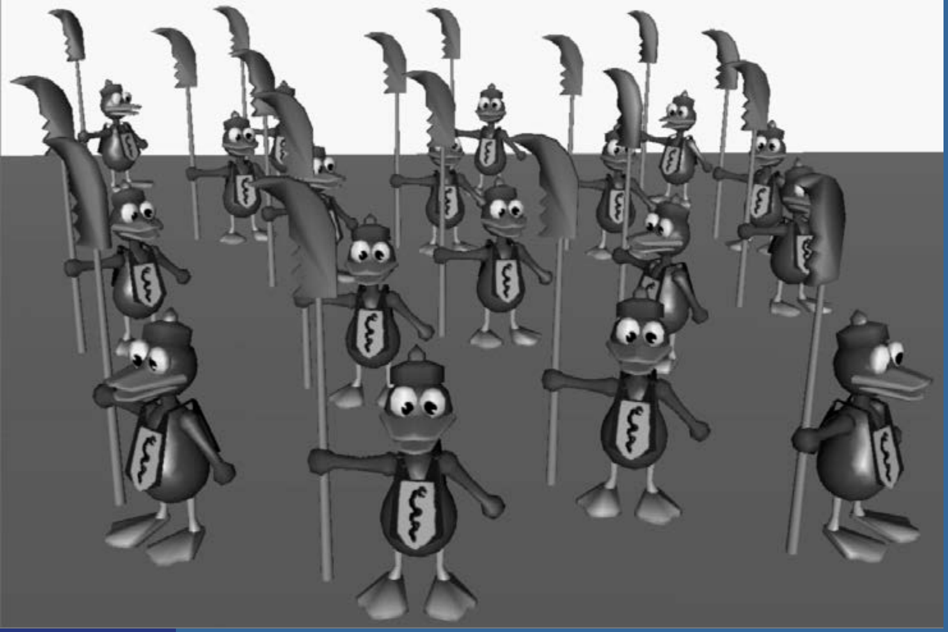
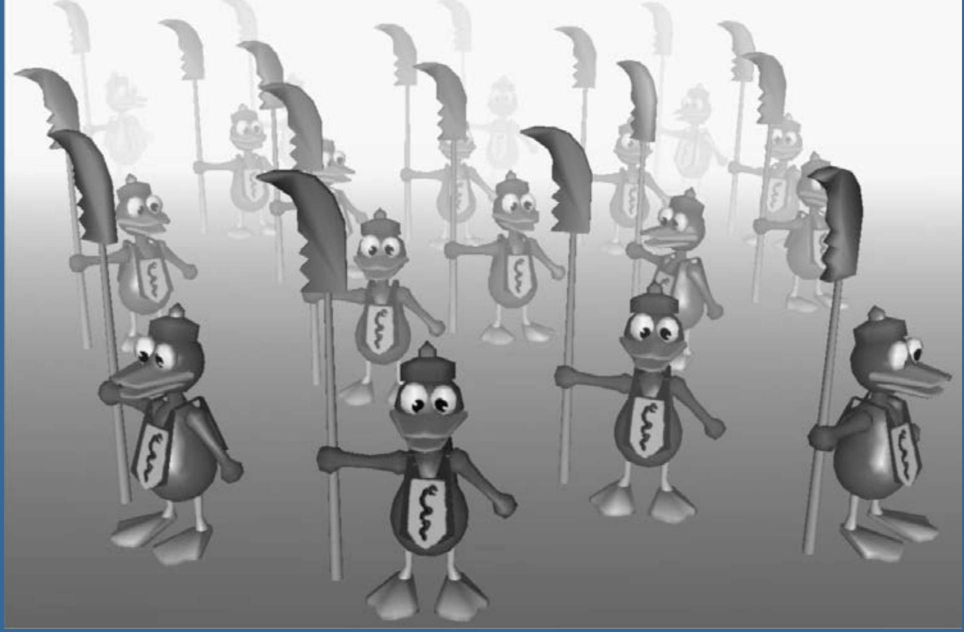
$$\mathbf{i}_{spec} = \max(0, (\mathbf{h}\cdot\mathbf{n})^{m_{shi}})\mathbf{m}_{spec} \otimes \mathbf{s}_{spec}$$

Shading



- Flat, Goraud, and Phong shading:
  - Flat shading: one normal per triangle. Lighting computed once for the whole triangle.
  - Gouraud shading: the lighting is computed per triangle vertex and for each pixel, the **color is interpolated** from the colors at the vertices.
  - Phong Shading: the lighting is **not** computed 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.



- Color of fog: $\mathbf{c}_f$ color of surface: $\mathbf{c}_s$

$$\mathbf{c}_p = f\mathbf{c}_s + (1-f)\mathbf{c}_f \qquad f \in [0,1]$$

- How to compute f?
- E.g., linearly:

$$f = \frac{z_{end} - z_p}{z_{end} - z_{start}}$$

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:

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

(Blending)

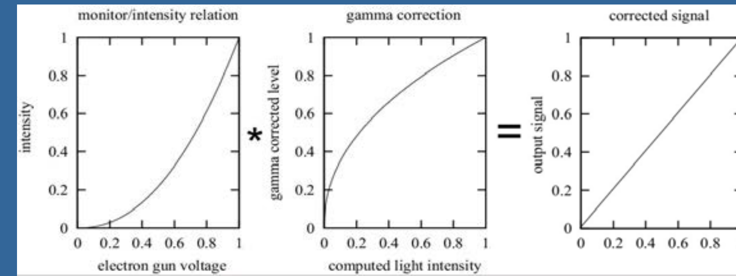
Rendered object

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.**

If we have high frame-to-frame coherency regarding the objects to be sorted per frame, then Bubble-sort (or Insertion sort) are really good! Superior to Quicksort.
Because, they have expected runtime of resorting already almost sorted input in O(n) instead of O(n log n), where n is number of elements.

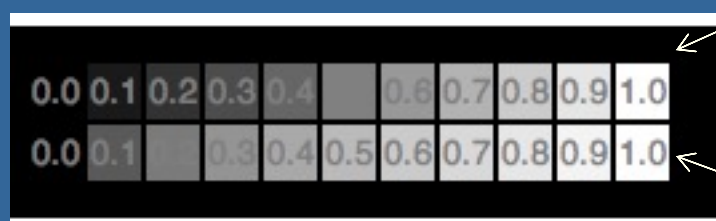
$$c = c_i^{(1/\gamma)}$$



Gamma correction

- Reasons for wanting gamma correction (standard is 2.2):

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

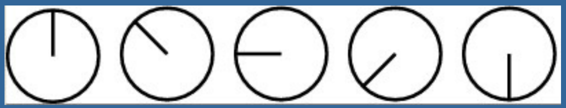


Gamma of 2.2. Better distribution for humans. Perceived as linear.

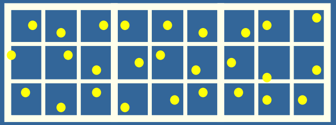
Truly linear intensity increase.

A linear intensity output (bottom) 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.
A nonlinearly-increasing intensity (upper), will show much more even steps in perceived brightness.

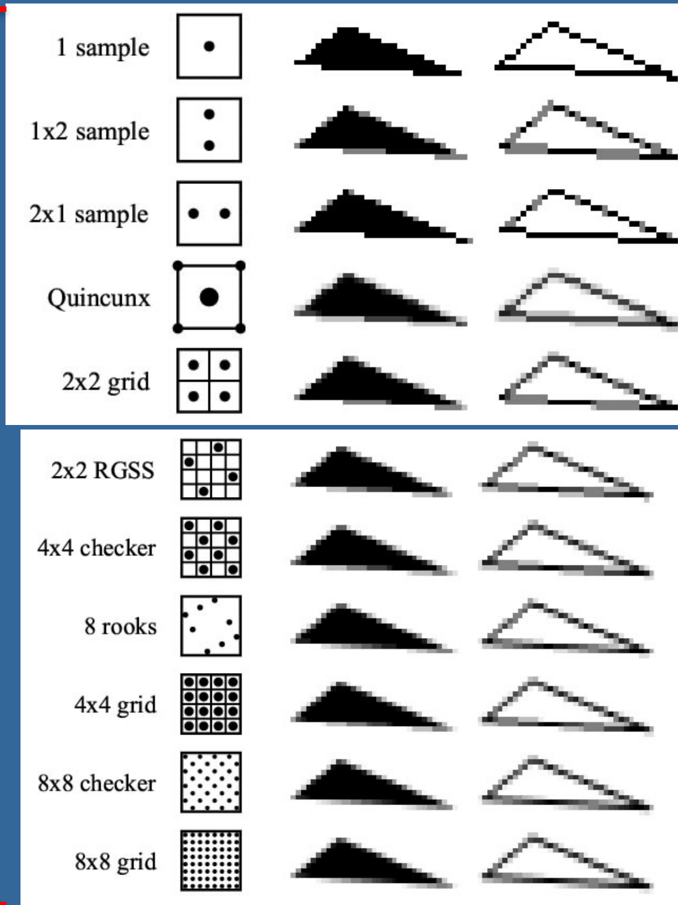
Leture 3.2: Sampling, filtrering, and Antialiasing



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



- Supersampling vs multisampling vs coverage sampling



04. Texturing

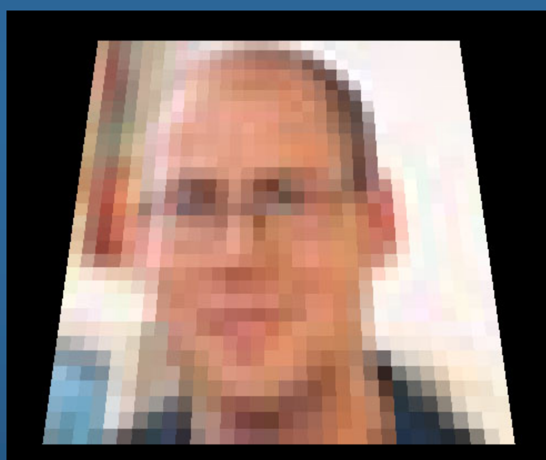
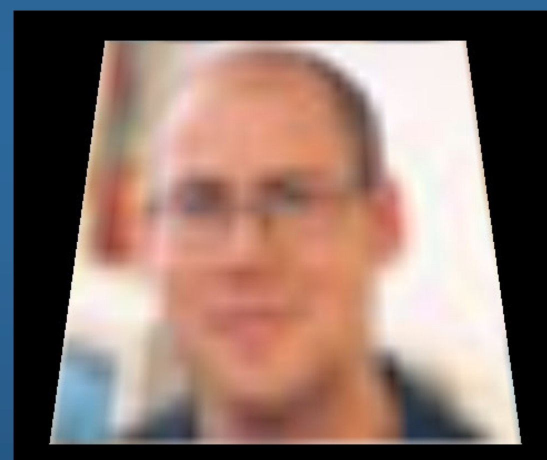
What is most important:

- Filtering: magnification, minification
  - Mipmaps + their memory cost
  - How compute bilinear/trilinear filtering
  - Number of texel accesses for trilinear filtering
  - Anisotropic filtering
- Environment mapping – cube maps, how compute lookup.
- Bump mapping
- 3D-textures – what is it?
- Sprites
- Billboards/Impostors, viewplane vs viewpoint oriented, axial billboards, how to handle depth buffer for fully transparent texels.
- Particle systems

Filtering

FILTERING:

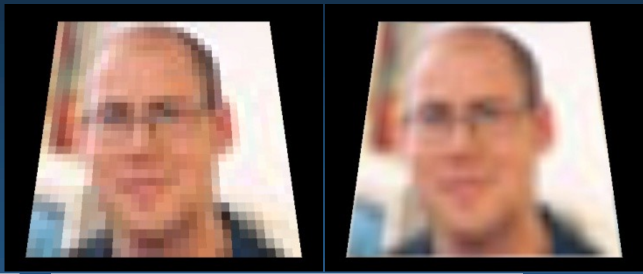
- For magnification: Nearest or Linear (box vs Tent filter)

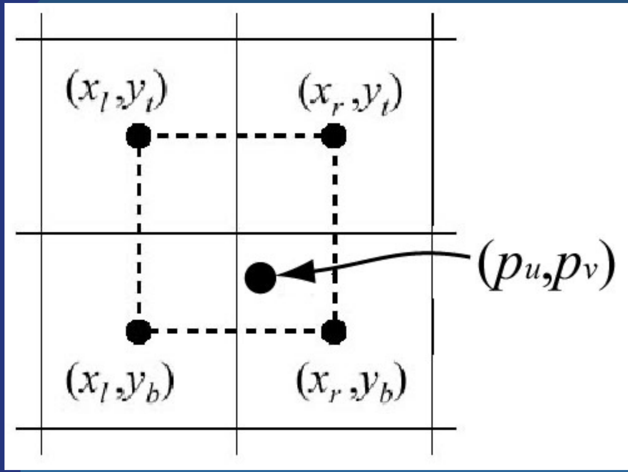
Sinc filter not usable in real time. Why?...

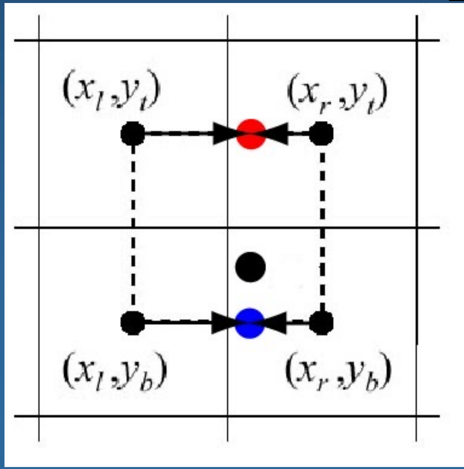
- For minification: nearest, linear and…
  - Bilinear – using mipmapping
  - Trilinear – using mipmapping
  - Anisotropic – up to 16 mipmap lookups along line of anisotropy

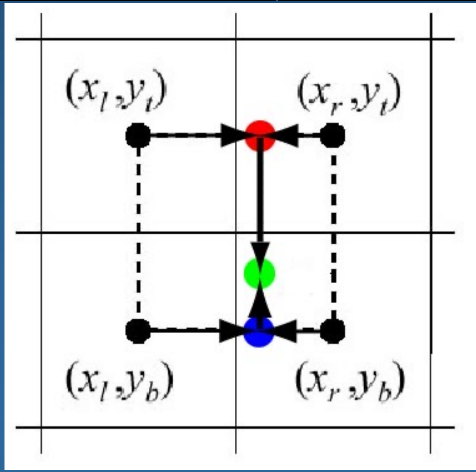
Interpolation



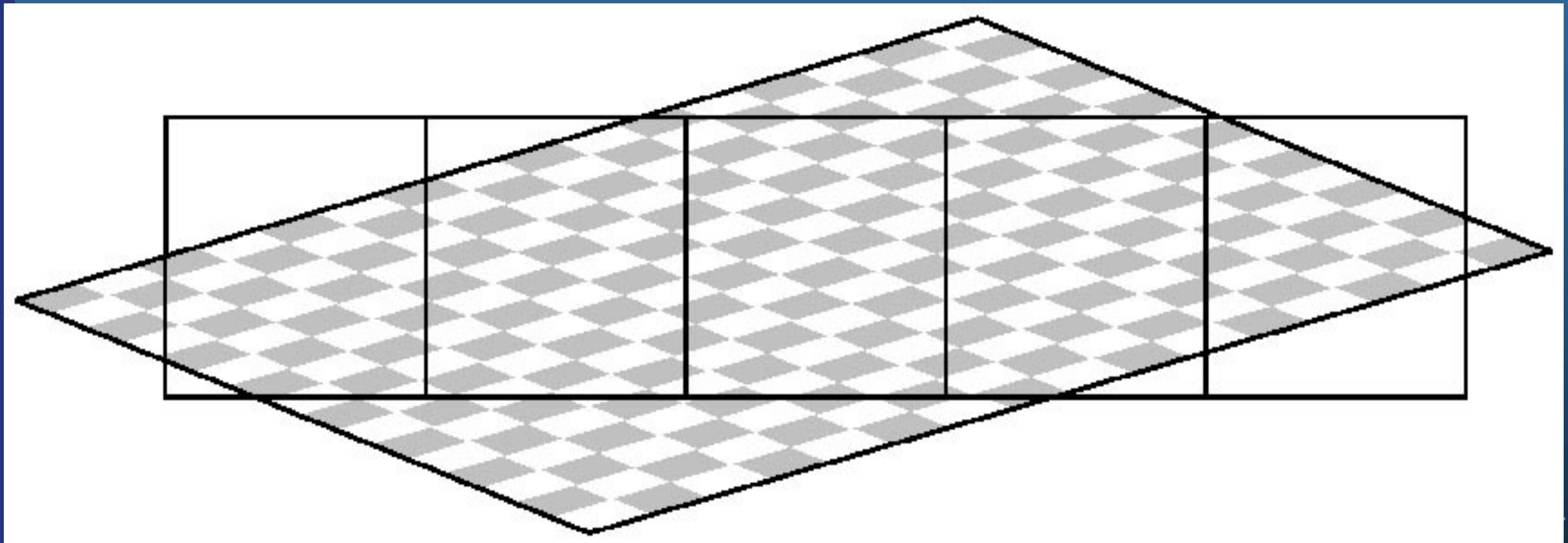
Magnification



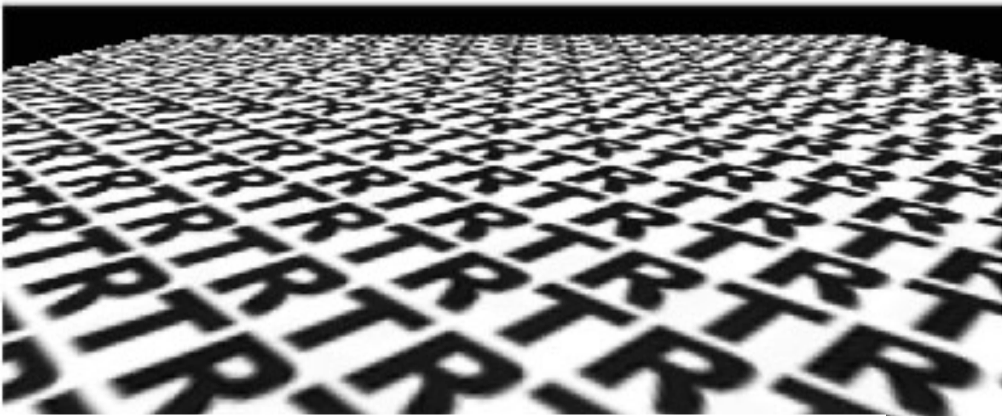


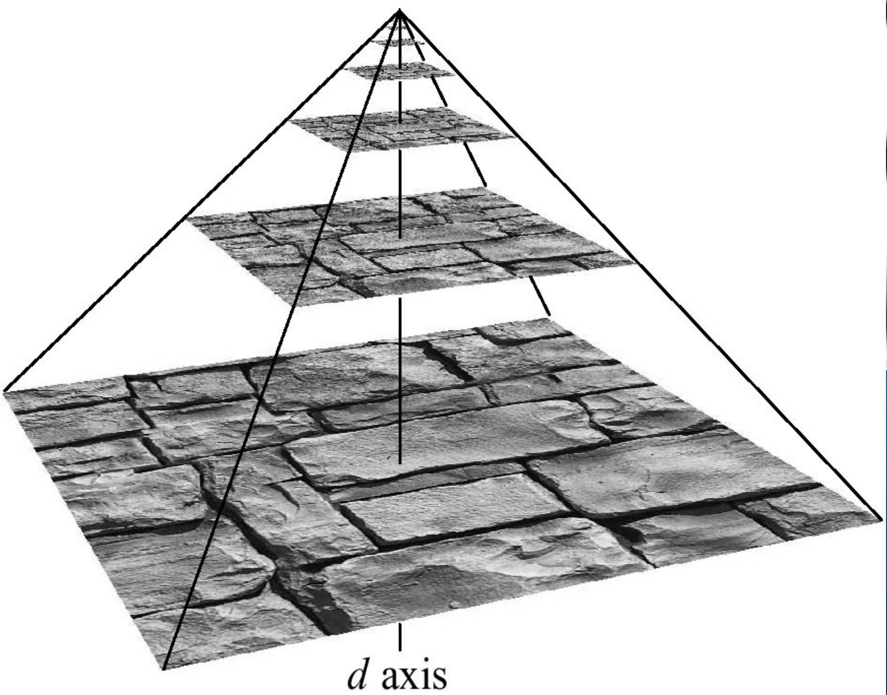


Minification



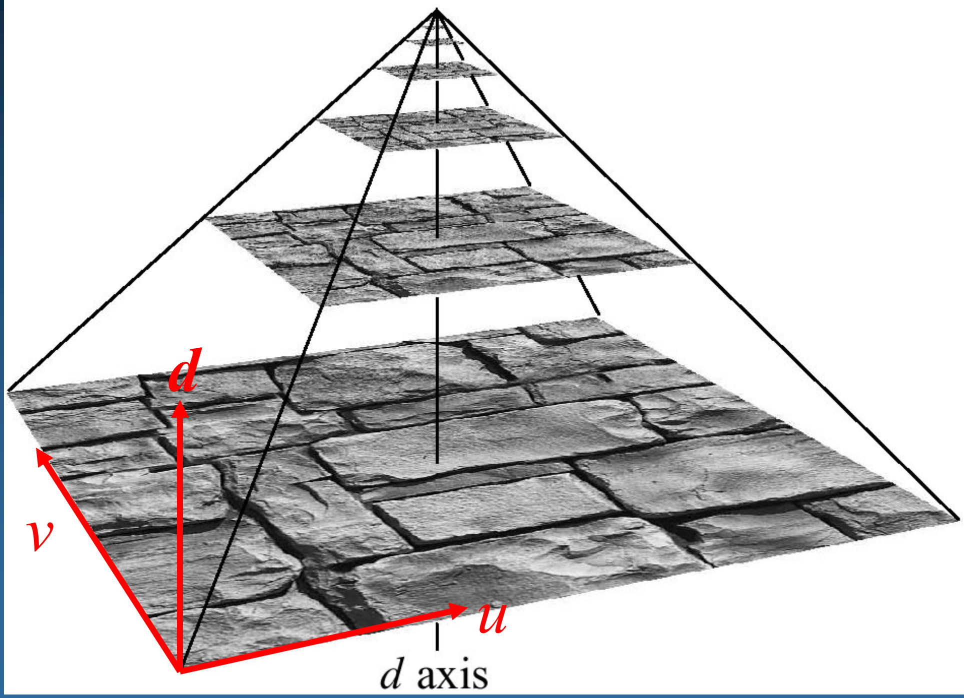
Bilinear filtering using Mipmapping





Mipmapping

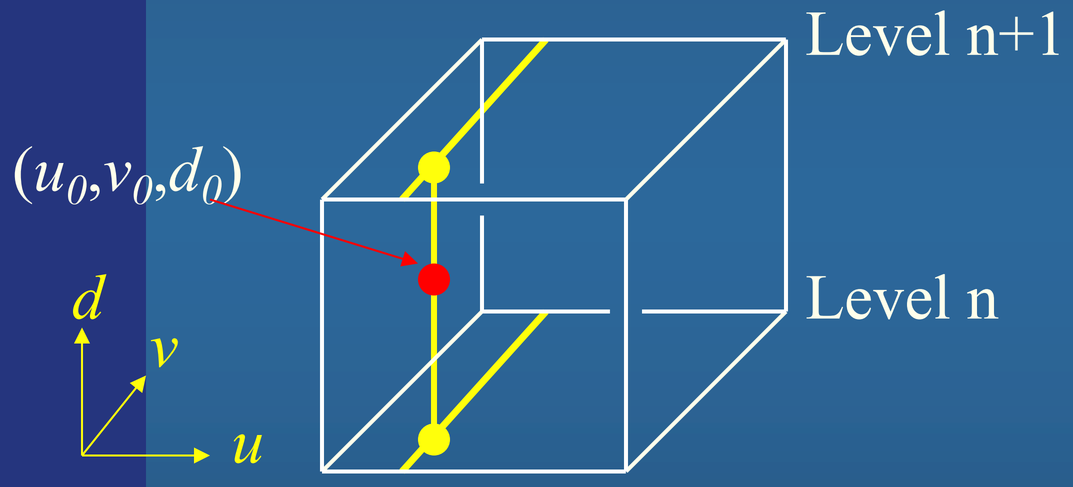
- Image pyramid
- Half width and height when going upwards



- 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

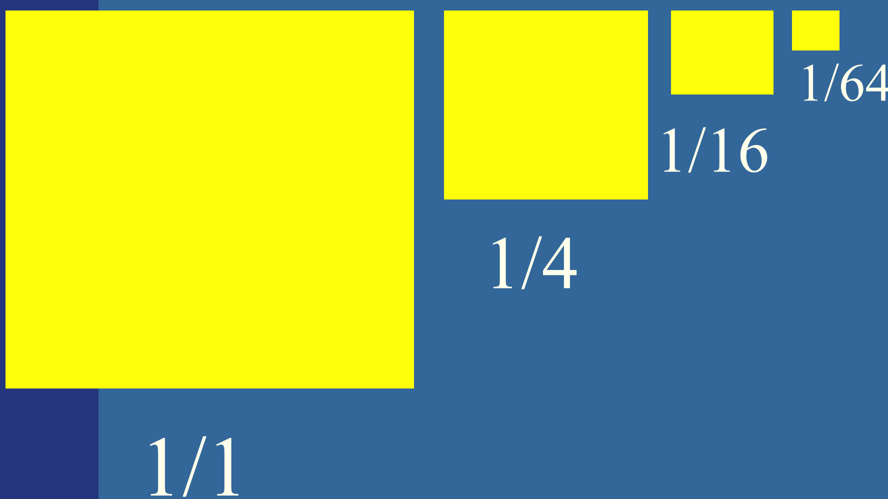
- Interpolate between those bilinear values
  - Gives trilinear interpolation

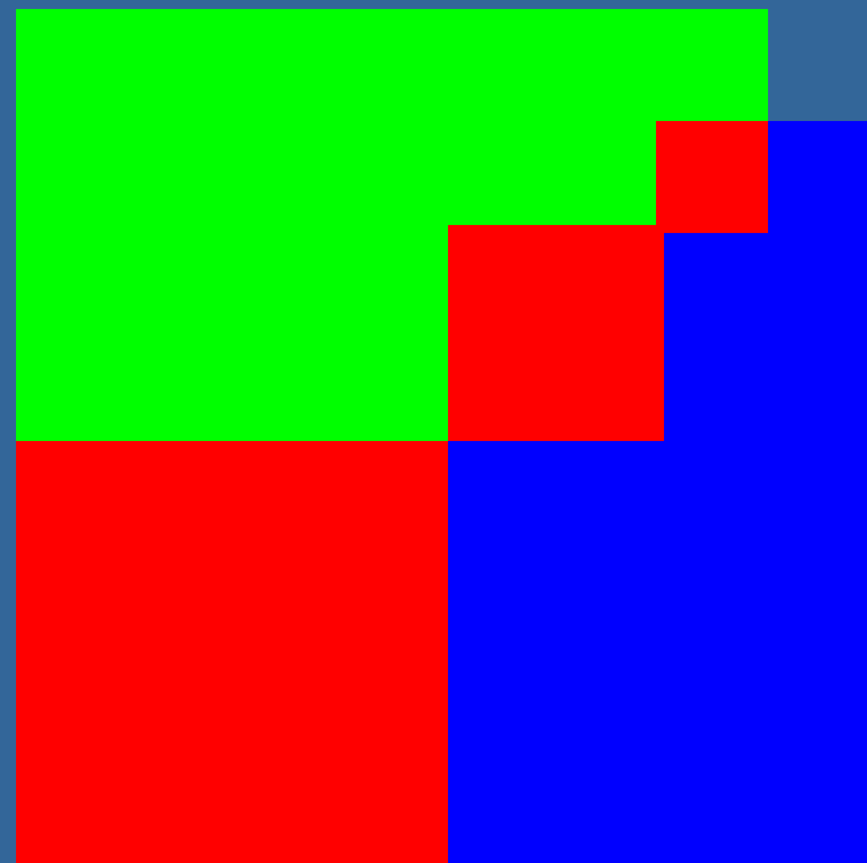


- Constant time filtering: 8 texel accesses

Mipmapping: Memory requirements

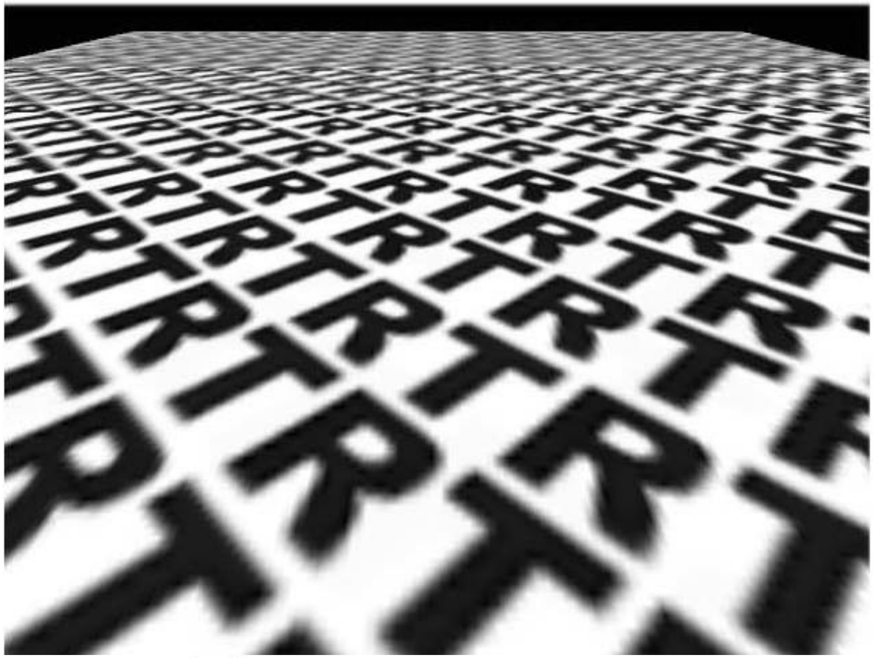
- Not twice the number of bytes…!



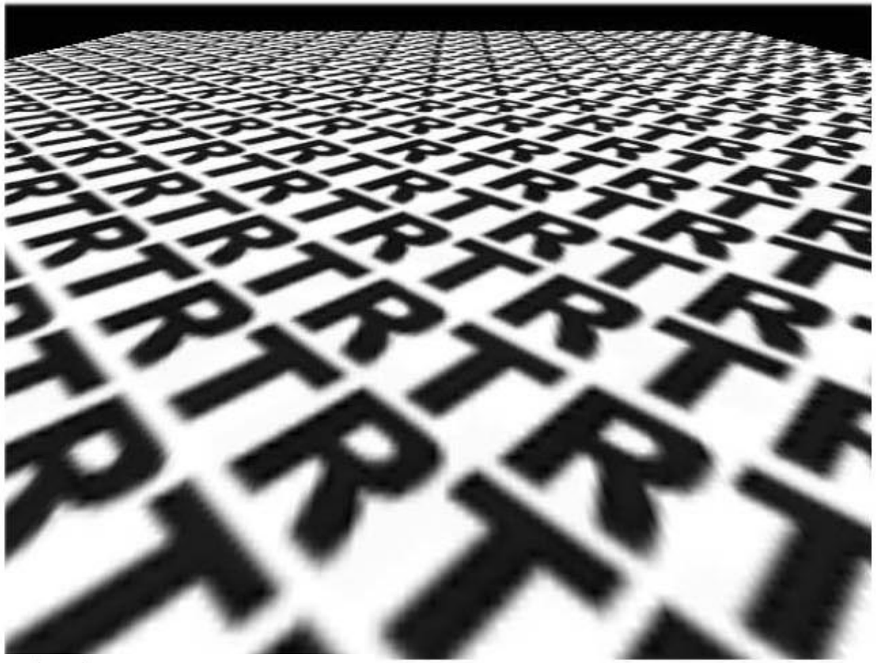


- Rather 33% more – not that much

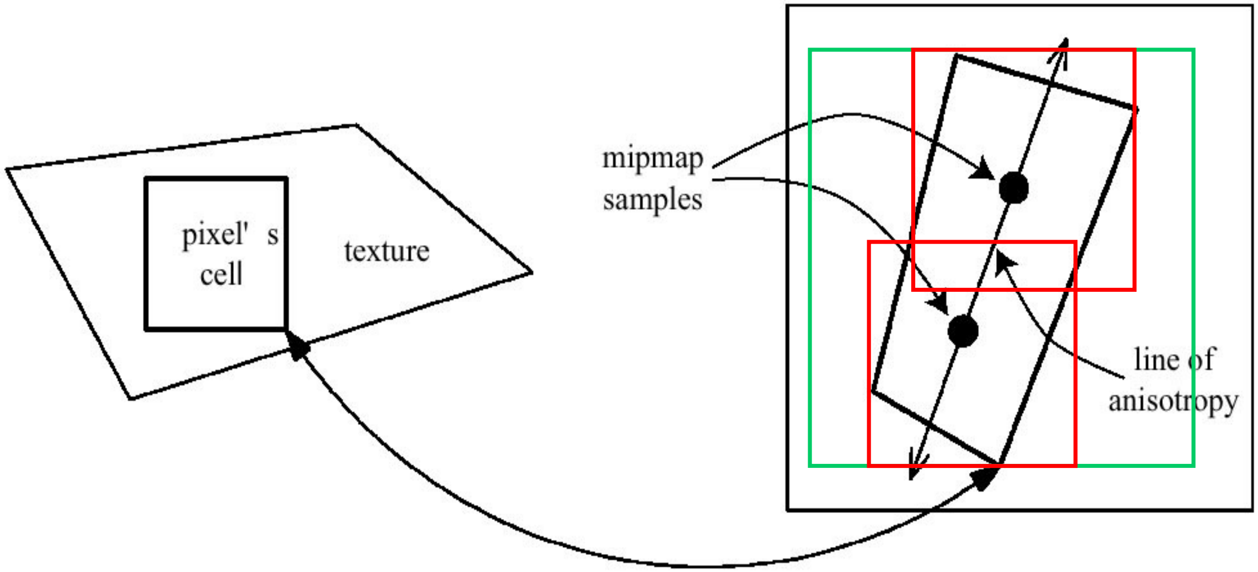
Anisotropic texture filtering



pixel space

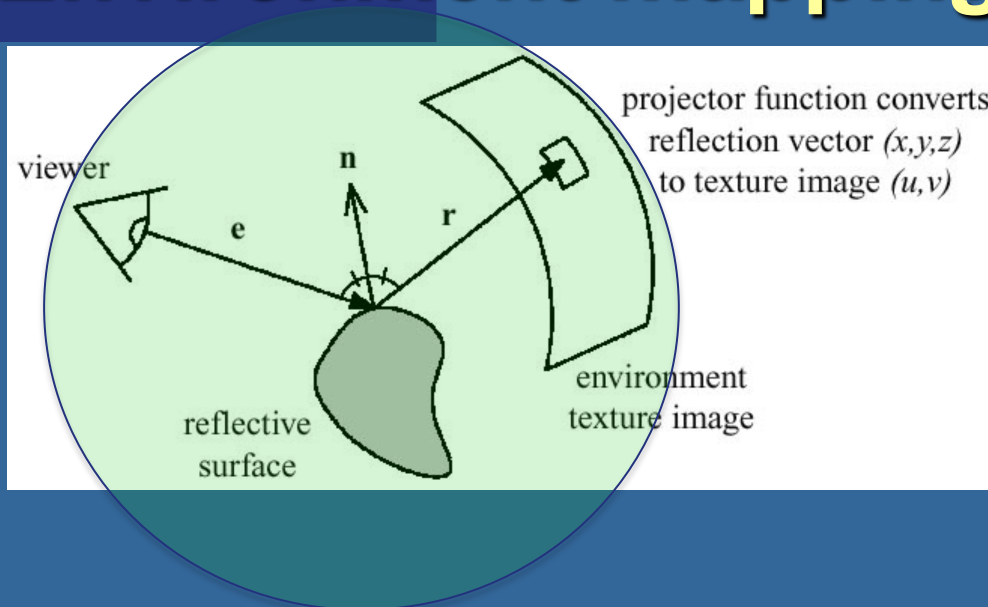


texture space



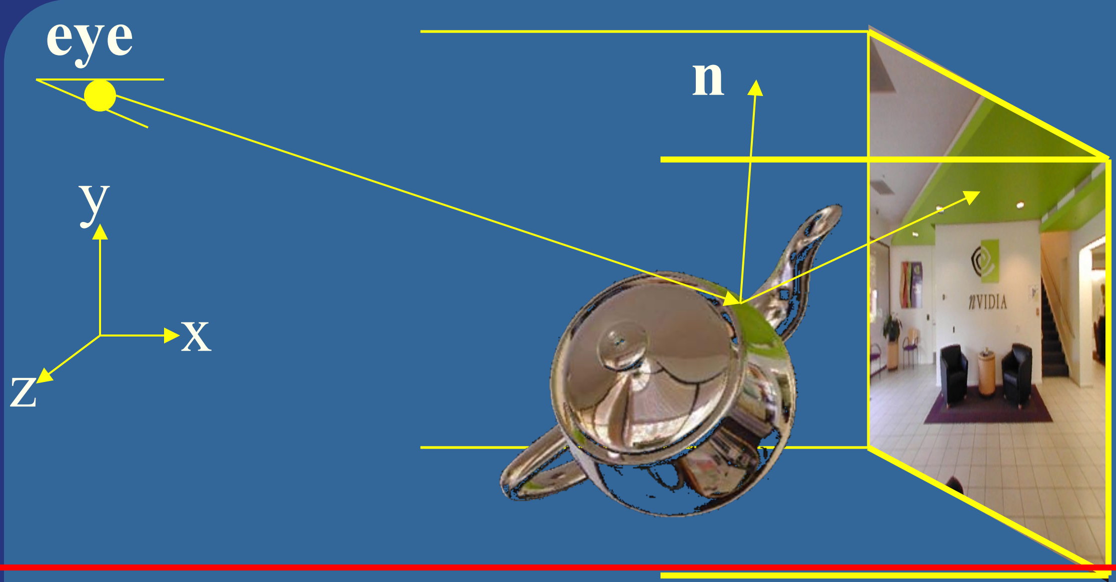
See page 187-188

Environment mapping



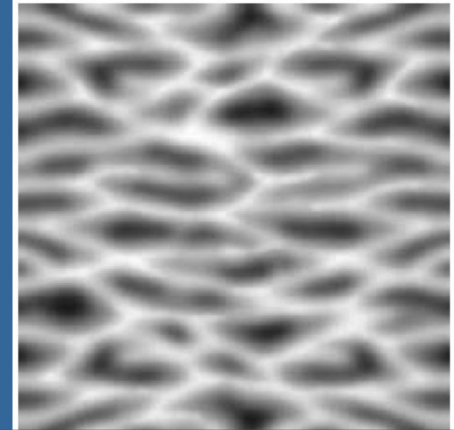
- Assumes the environment is infinitely far away
- E.g., sphere mapping, or cube mapping
- Cube mapping is the norm nowadays

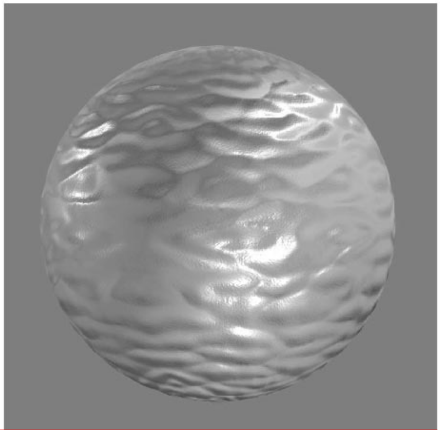
Cube mapping



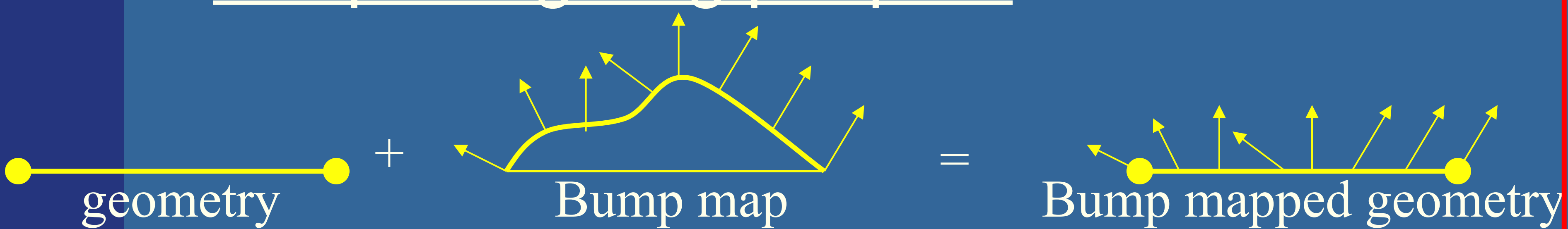
- Simple math: compute reflection vector, **r**
- Largest abs-value of component, determines which cube face.
  - Example: **r**=(5,-1,2) gives POS_X face
- Divide **r** by abs(5) gives (*u*,*v*)=(-1/5,2/5)
- Also remap from [-1,1] to [0,1] by (u,v) = ((u,v)+vec2(1,1))*0.5;
- Your hardware does all the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

Bump mapping



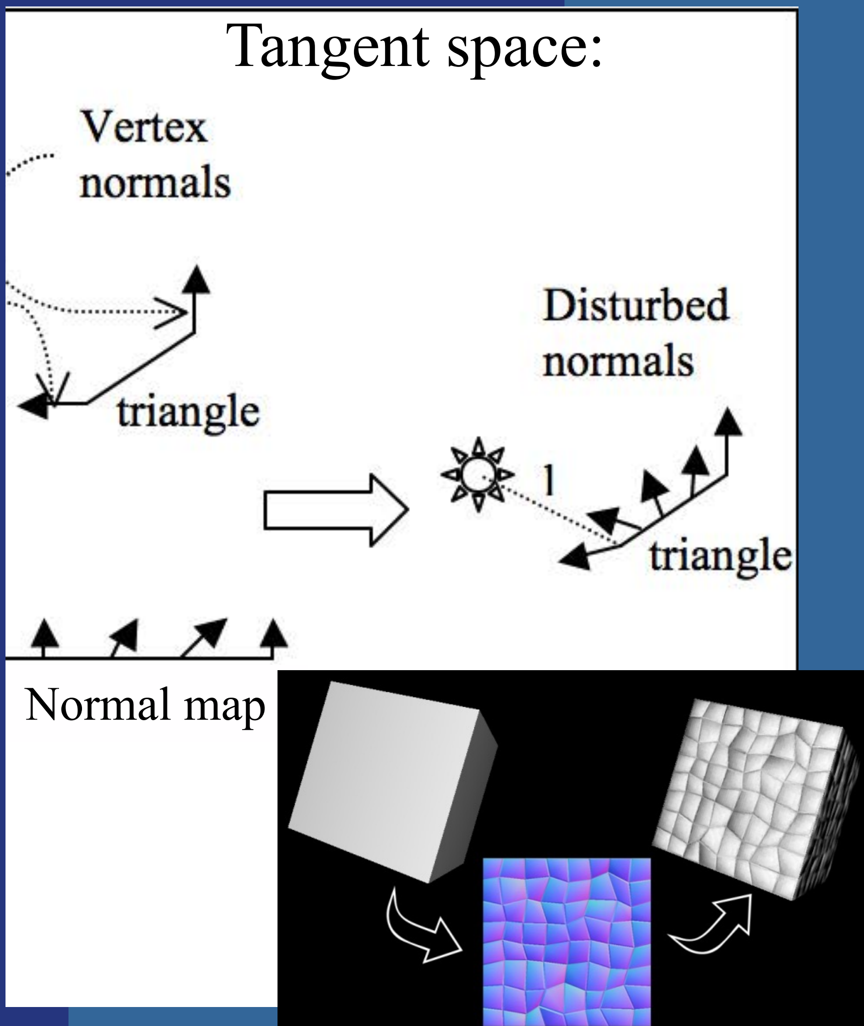


- by Blinn in 1978
- Inexpensive way of simulating wrinkles and bumps on geometry
  - Expensive to model these geometrically
- Instead <u>let a texture modify the normal at each pixel, and then use this normal to compute lighting per pixel</u>



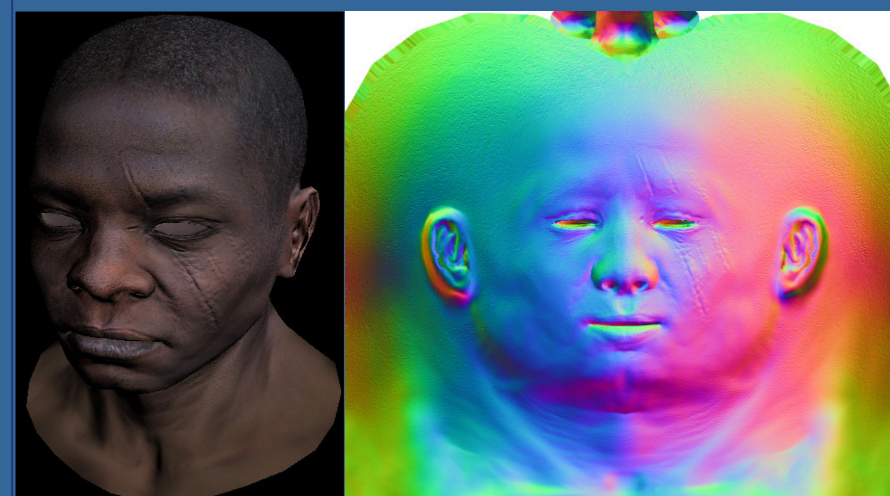
Stores heights: can derive normals

Normal mapping in tangent vs object space



Object space:

•Normals are stored directly in model space. I.e., as including both face orientation plus distorsion.

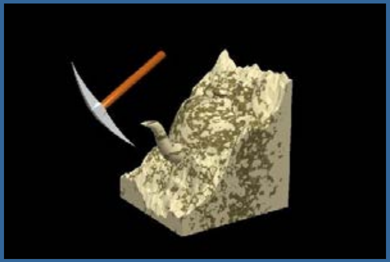


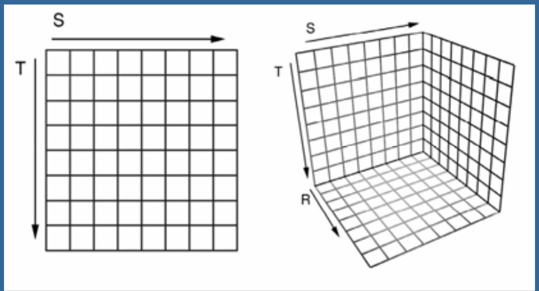
Tangent space:

●Normals are stored as distorsion of face orientation. The same bump map can be tiled/repeated and reused for many faces with different orientation

More...

- 3D textures:





2D 3D

  - Texture filtering is no longer trilinear
  - Rather quadlinear
    - (trilinear interpolation in both 3D-mipmap levels + between mipmap levels)
  - Enables new possibilities
    - Can store light in a room, for example

- Displacement Mapping
  - Like bump/normal maps but truly offsets the surface geometry (not just the lighting).
  - Gfx hardware cannot offset the fragment's position
    - Offsetting per vertex is easy in vertex shader but requires a highly tessellated surface.
    - Tesselation shaders are created to increase the tessellation of a triangle into many triangles over its surface. Highly efficient.
    - (Can also be done using Geometry Shader (e.g. Direct3D 10) by ray casting in the displacement map, but tessellation shaders are generally more efficient for this.)

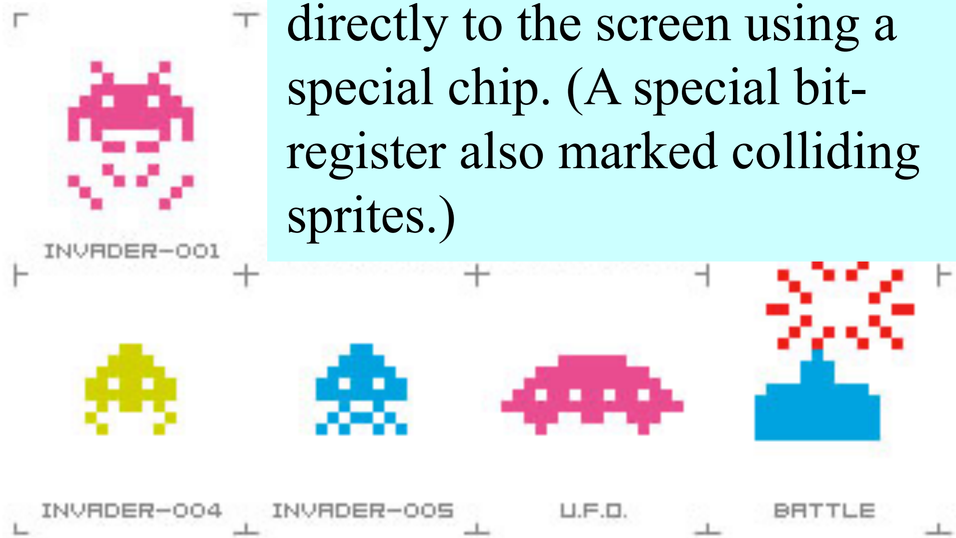
05. Texturing:

Just know what “sprites” is and that they are very 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.)

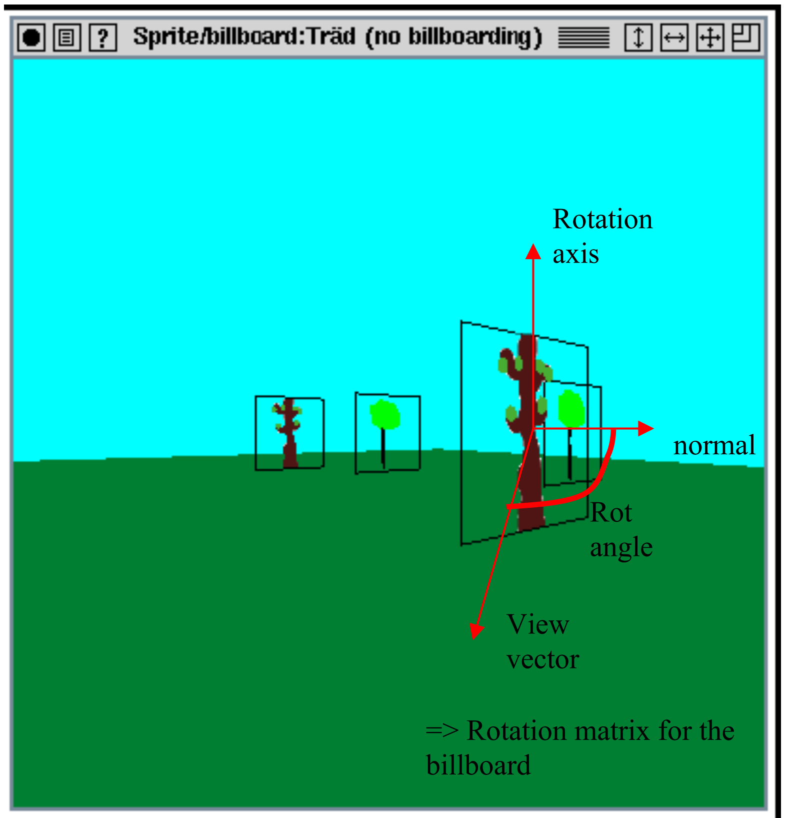
```
GLbyte M[64]=
{   127,0,0,127, 127,0,0,127,
    127,0,0,127, 127,0,0,127,
    0,127,0,0, 0,127,0,127,
    0,127,0,127, 0,127,0,0,
    0,0,127,0, 0,0,127,127,
    0,0,127,127, 0,0,127,0,
    127,127,0,0, 127,127,0,127,
    127,127,0,127, 127,127,0,0};

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();
}
```

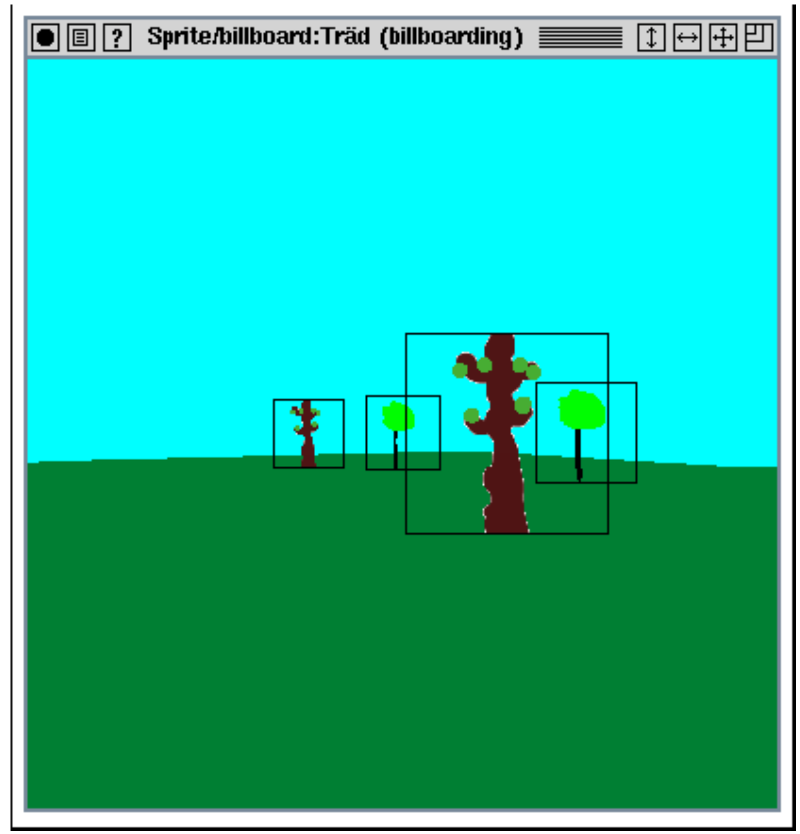


Billboards

- 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

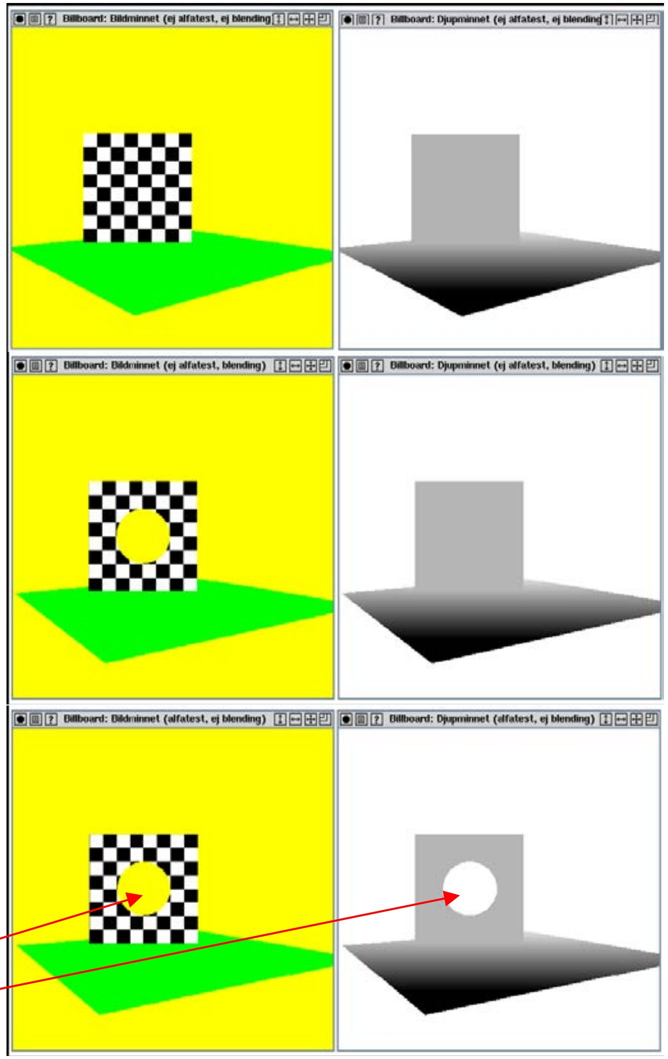
- Fix correct transparency by blending AND using alpha-test
  - In fragment shader:

    if (color.a < 0.1) discard;

If alpha value in texture is lower than some small 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.

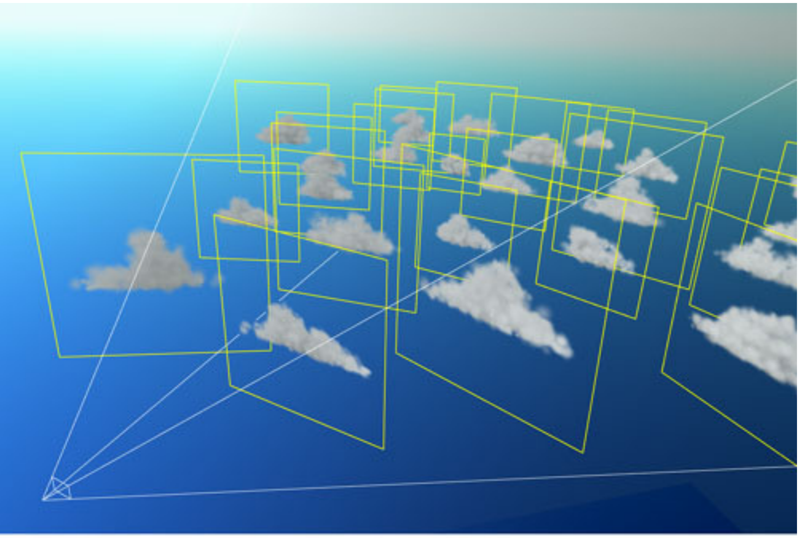
E.g. here, so that objects behind are visible through the hole

Color Buffer   Depth Buffer

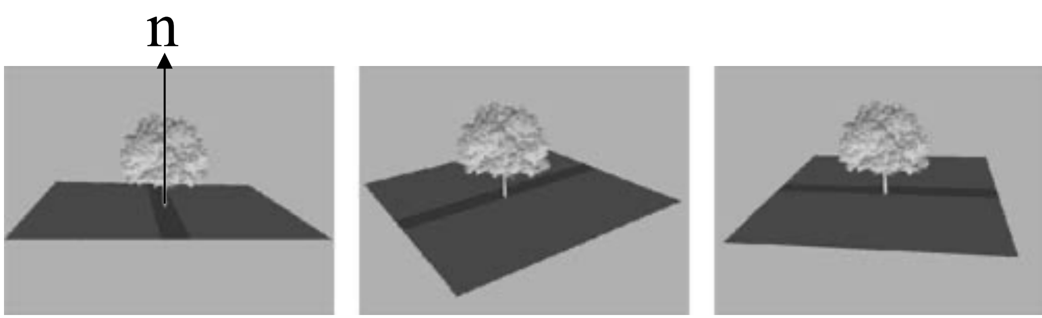


With blending

With alpha test



(Also called *Impostors)*



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

Lecture 5: OpenGL

- How to use OpenGL (or DirectX)
  - Will not ask about syntax. Know how to use.
    - I.e. functionality
  - E.g. how to achieve
    - Blending and transparency
    - Fog – how would you implement in a fragment shader?
      - pseudo code is enough
    - Specify a material, a triangle, how to translate or rotate an object.
    - Triangle – vertex order and facing

Buffers

- Frame buffer
  - Back/front/left/right – **glDrawBuffers()**
  - Offscreen buffers (e.g., framebuffer objects, auxiliary buffers)

Frame buffers can consist of:

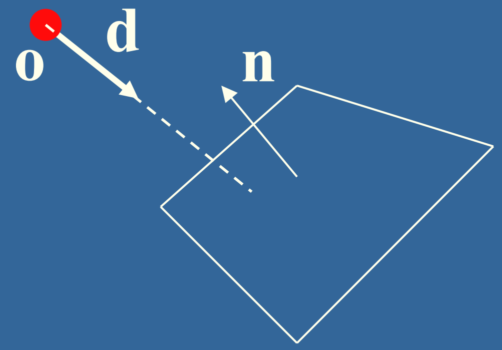
- Color buffer - rgb(a)
- Depth buffer (z-buffer)
  - For correct depth sorting
  - Instead of BSP-algorithm or painters algorithm…
- Stencil buffer
  - E.g., for shadow volumes or only render to frame buffer where stencil = certain value (e.g., for masking).

Lecture 6: Intersection Tests

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

    Test:
    1. face normals of A,
    2. face normals 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 and plane.
- Know equations for ray, sphere, cylinder, plane, triangle

Analytical: Ray/plane intersection



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

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

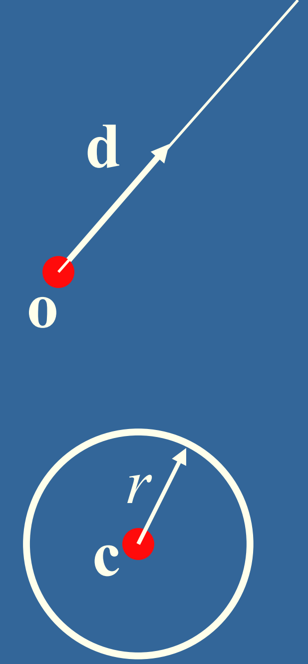
$\mathbf{n}\cdot(\mathbf{o}+t\mathbf{d}) + d = \mathbf{0}$

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

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

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*

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

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

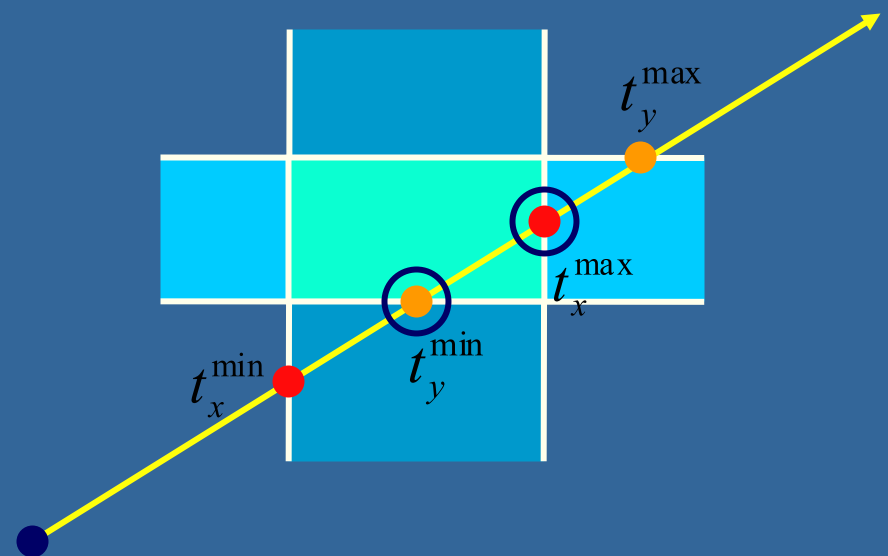
$$(\mathbf{d}\cdot\mathbf{d})t^2+2((\mathbf{o}-\mathbf{c})\cdot\mathbf{d})t+(\mathbf{o}-\mathbf{c})\cdot(\mathbf{o}-\mathbf{c})-r^2=0$$

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

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

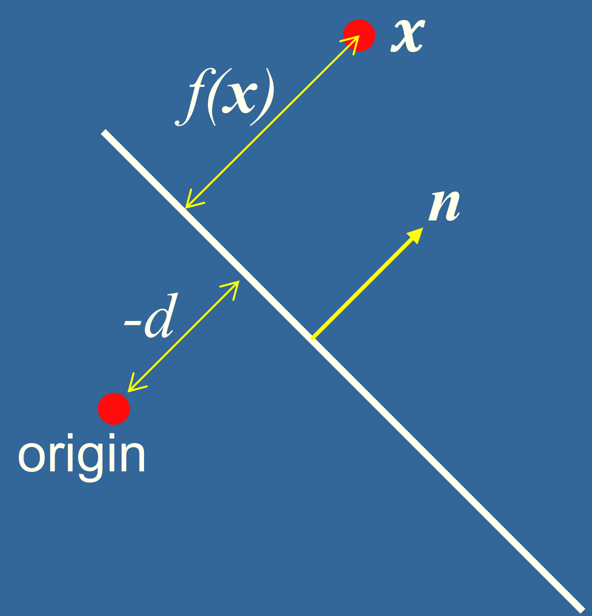
The Plane Equation

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

If $\boldsymbol{n} \cdot \boldsymbol{x} + d = 0$, then $\boldsymbol{x}$ lies in the plane.

The function $f(\mathbf{x}) = \mathbf{n} \cdot \mathbf{x} + d$ gives the signed distance of $\mathbf{x}$ from the plane. ($\mathbf{n}$ should be normalized.)

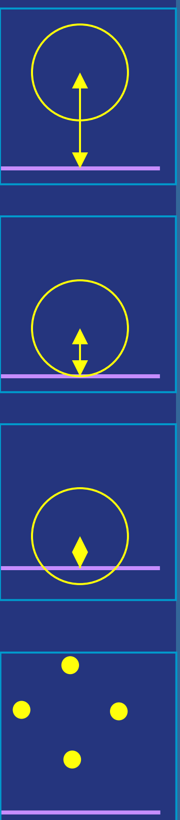
- $f(\boldsymbol{x})>0$ means above the plane
- $f(\boldsymbol{x})<0$ means below the plane



-d is how far the origin is behind the plane

Sphere/Plane
Box/Plane

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



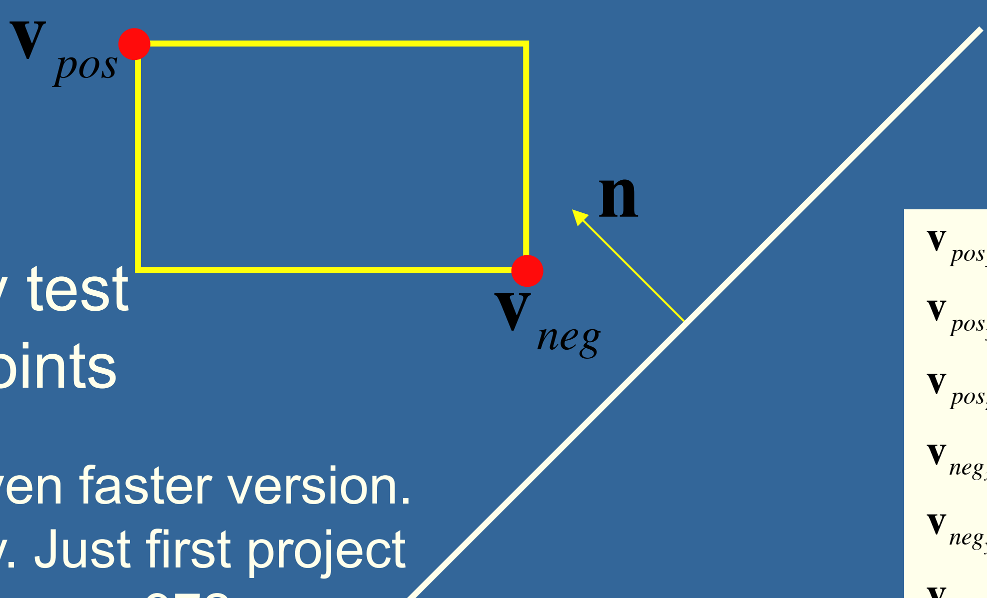
- Sphere: compute $\boxed{f(\mathbf{c}) = \mathbf{n}\cdot\mathbf{c} + d}$
- $f(\mathbf{c})$ is the signed distance ($\mathbf{n}$ normalized)
- $\mathrm{abs}(f(\mathbf{c})) > \mathrm{r}$ no collision
- $\mathrm{abs}(f(\mathbf{c})) = \mathrm{r}$ sphere touches the plane
- $\mathrm{abs}(f(\mathbf{c})) 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

AABB/plane

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

- The smart way (shown in 2D)
- Find the two vertices that have the most positive and most negative value when tested againt the plane

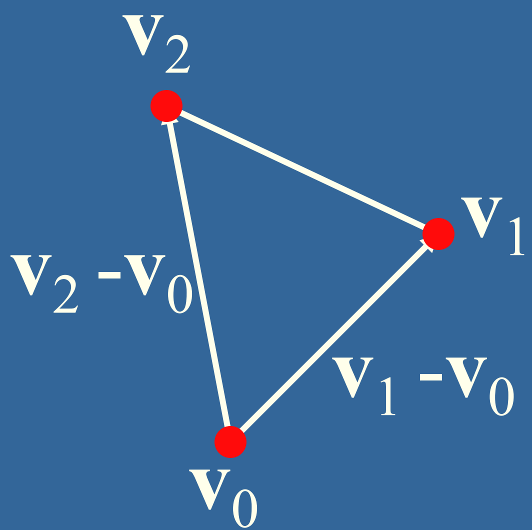


Need only test the red points

$$\mathbf{v}_{pos_x} = (\mathbf{n}_x > 0) ? \mathbf{b}_{\max_x} : \mathbf{b}_{\min_x}$$
$$\mathbf{v}_{pos_y} = (\mathbf{n}_y > 0) ? \mathbf{b}_{\max_y} : \mathbf{b}_{\min_y}$$
$$\mathbf{v}_{pos_z} = (\mathbf{n}_z > 0) ? \mathbf{b}_{\max z} : \mathbf{b}_{\min_z}$$
$$\mathbf{v}_{neg_x} = (\mathbf{n}_x < 0) ? \mathbf{b}_{\max_x} : \mathbf{b}_{\min_x}$$
$$\mathbf{v}_{neg_y} = (\mathbf{n}_y < 0) ? \mathbf{b}_{\max_y} : \mathbf{b}_{\min_y}$$
$$\mathbf{v}_{neg_z} = (\mathbf{n}_z < 0) ? \mathbf{b}_{\max z} : \mathbf{b}_{\min_z}$$

See page 970 for even faster version.
OBB almost as easy. Just first project **n** on OBB's axes – see p: 972

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:

$$\mathbf{t}(u,v) = \mathbf{v}_0 + u(\mathbf{v}_1 - \mathbf{v}_0) + v(\mathbf{v}_2 - \mathbf{v}_0)$$

where $[u,v>=0,\ u+v<=1]$ is inside triangle

- Set $\mathbf{t}(u,v)=\mathbf{r}(t)$, and solve for t, u, v:

$$\mathbf{v}_0+u(\mathbf{v}_1 - \mathbf{v}_0) + v(\mathbf{v}_2 - \mathbf{v}_0) = \mathbf{o}+t\mathbf{d}$$

$$=> -t\mathbf{d} + u(\mathbf{v}_1 - \mathbf{v}_0) + v(\mathbf{v}_2 - \mathbf{v}_0) = \mathbf{o}-\mathbf{v}_0$$

$$=> [-\mathbf{d}, (\mathbf{v}_1 - \mathbf{v}_0), (\mathbf{v}_2 - \mathbf{v}_0)]\ [t, u, v]^T = \mathbf{o}-\mathbf{v}_0$$

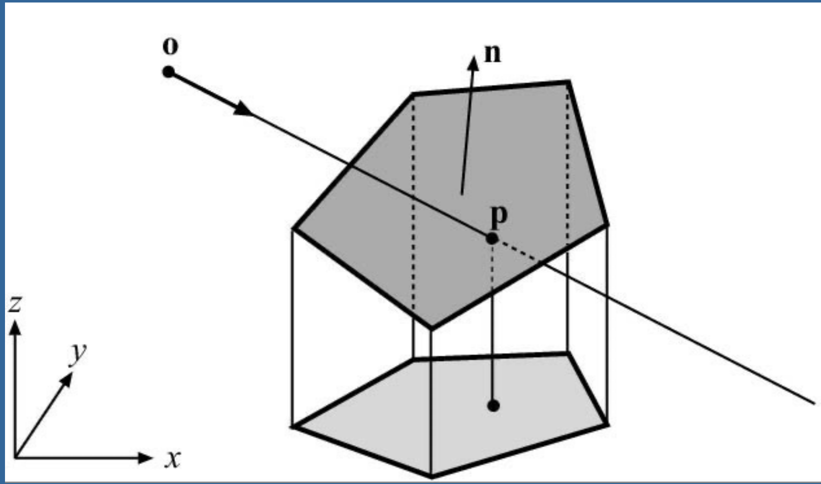
$$\begin{pmatrix} | & | & | \\ -\mathbf{d} & \mathbf{v}_1 - \mathbf{v}_0 & \mathbf{v}_2 - \mathbf{v}_0 \\ | & | & | \end{pmatrix} \begin{pmatrix} t \\ u \\ v \end{pmatrix} = \begin{pmatrix} | \\ \mathbf{o} - \mathbf{v}_0 \\ | \end{pmatrix}$$

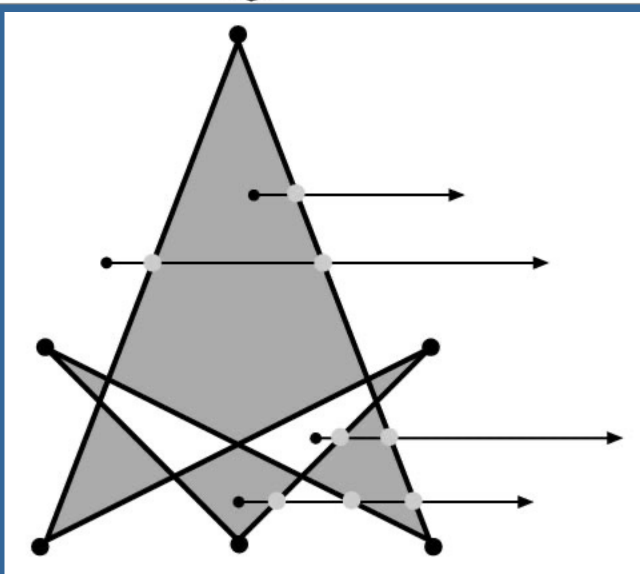
Ax=b

$\mathbf{x}=\mathbf{A}^{-1}\mathbf{b}$

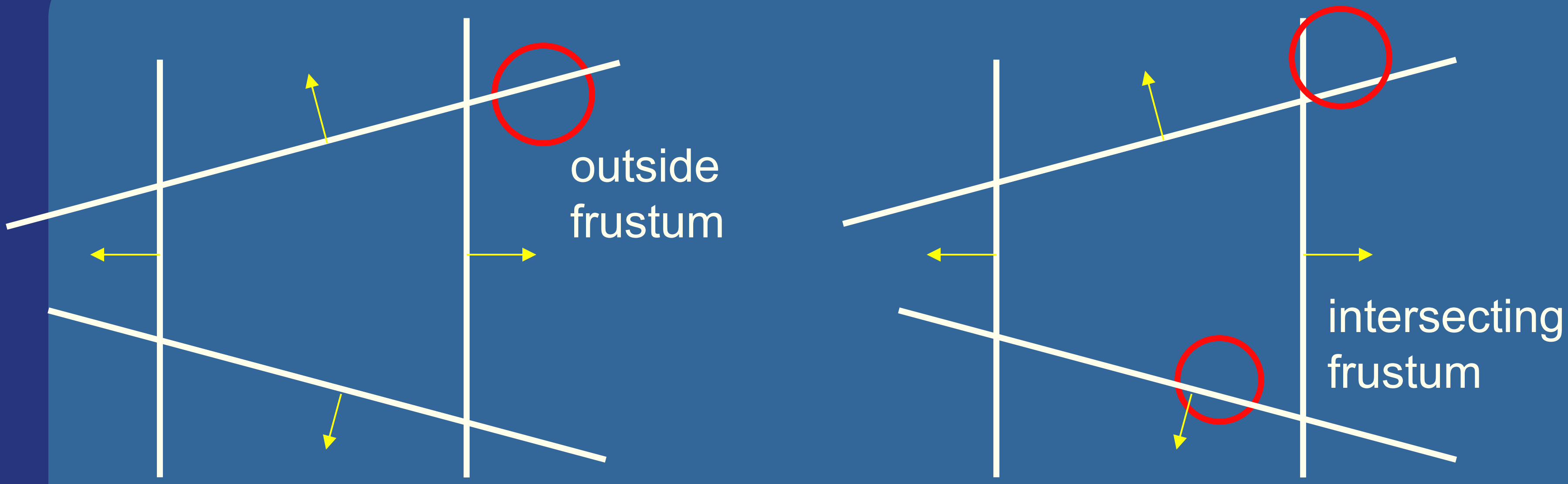
Ray/Polygon: very briefly

- 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





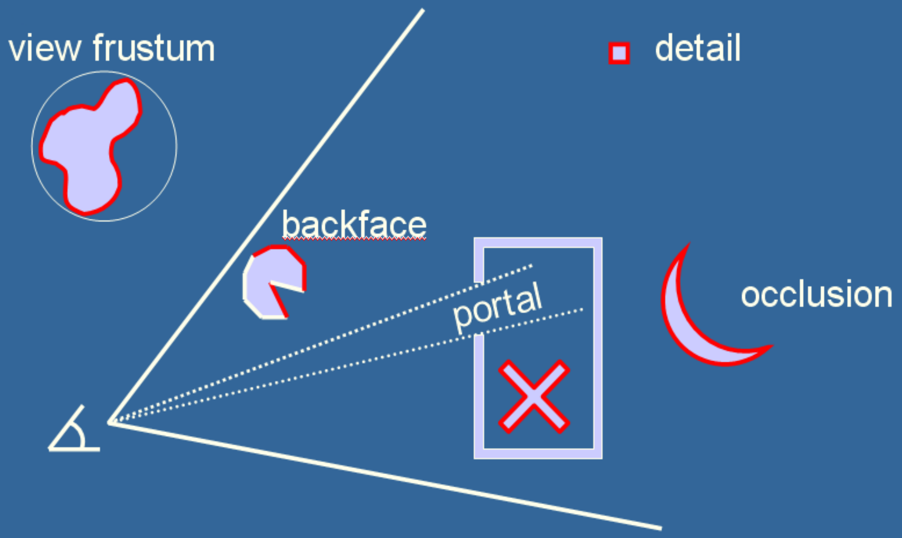
View frustum testing example



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

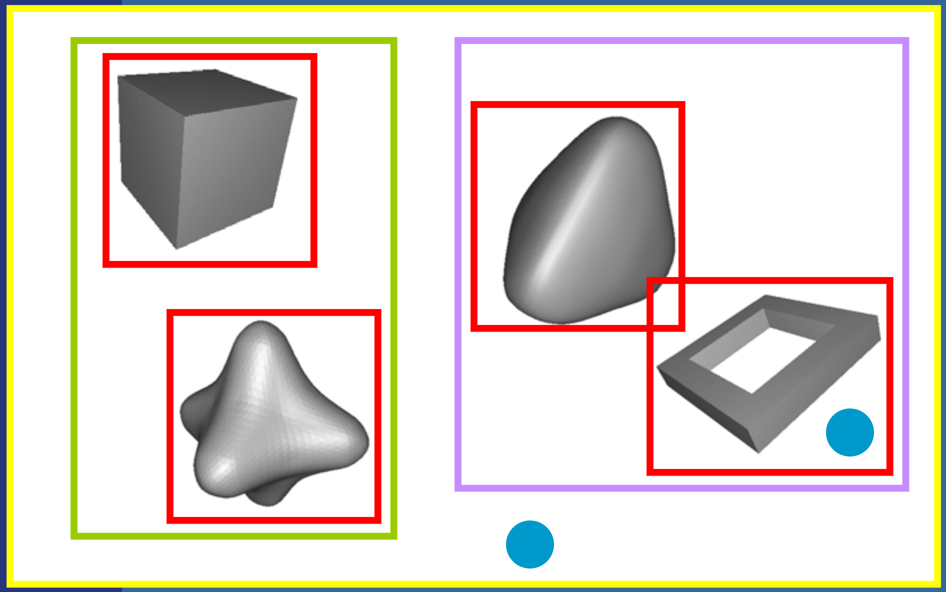
Lecture 7.1: Spatial Data Structures and Speed-Up Techniques

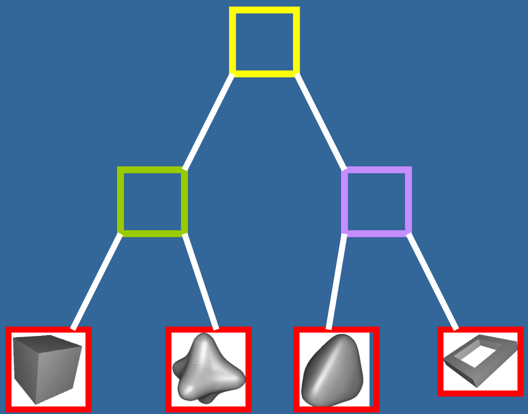
- Speed-up techniques
  - Culling
    - Backface
    - View frustum (hierarchical)
    - Portal
    - Occlusion Culling
    - Detail
  - Levels-of-detail: 
- How to construct and use the spatial data structures
    - BVH, BSP-trees (polygon aligned + axis aligned)



Axis Aligned Bounding Box Hierarchy - an example

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





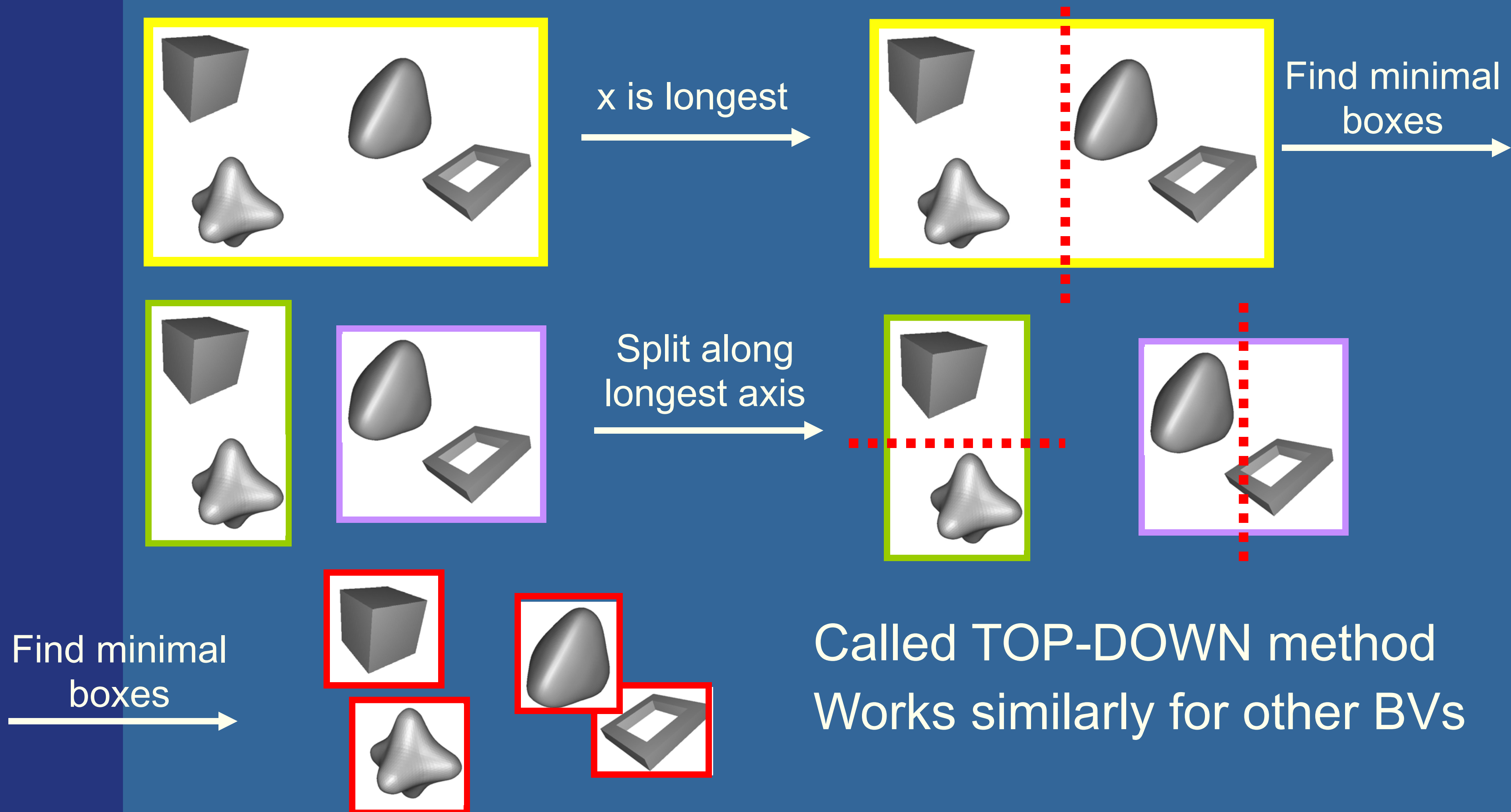
click!

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)

Bounding-Volume Hierarchy
– TOP-DOWN construction:

- Find minimal box, then split along longest axis

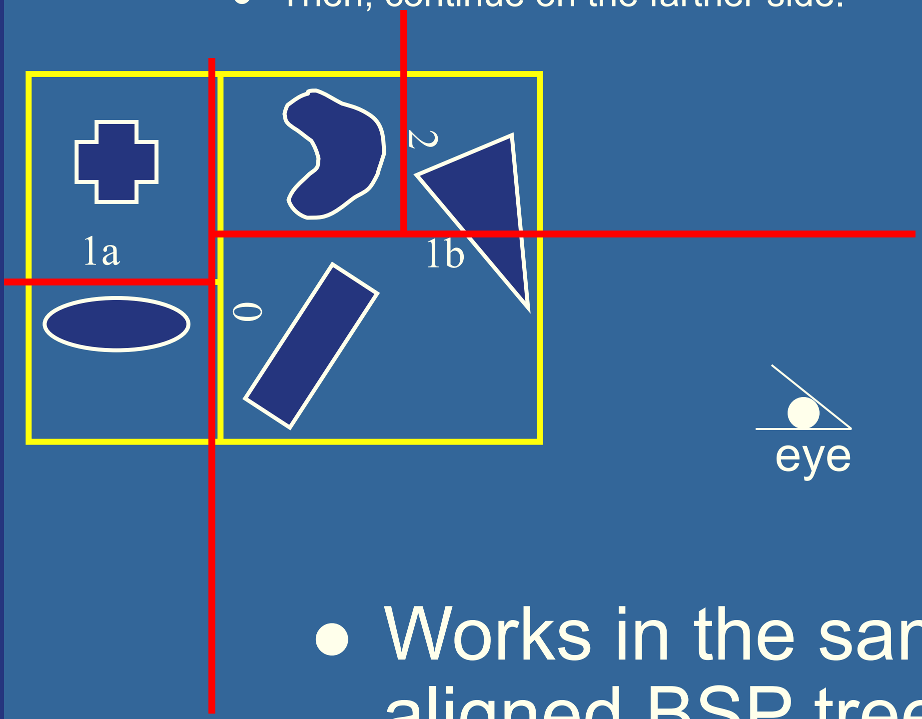


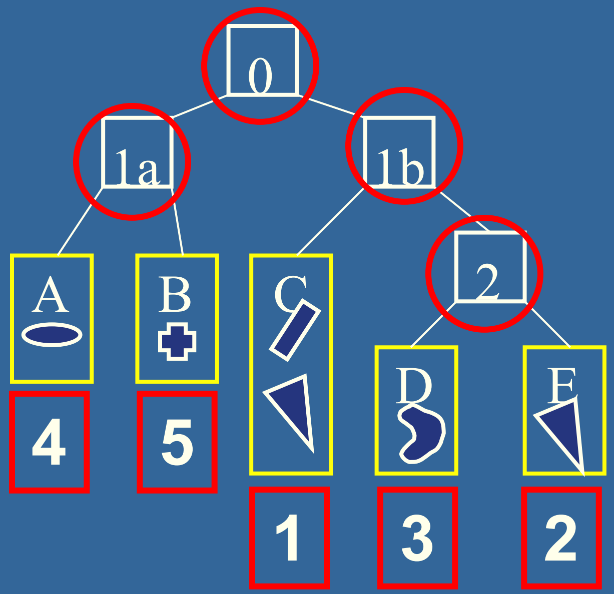
Called TOP-DOWN method

Works similarly for other BVs

Axis-aligned BSP tree
Rough sorting

- Test the planes, recursively from root, against the point of view. For each traversed node:
  - If node is leaf, draw the node's geometry
  - else
    - Continue traversal on the "hither" side with respect to the eye to sort front to back
    - Then, continue on the farther side.



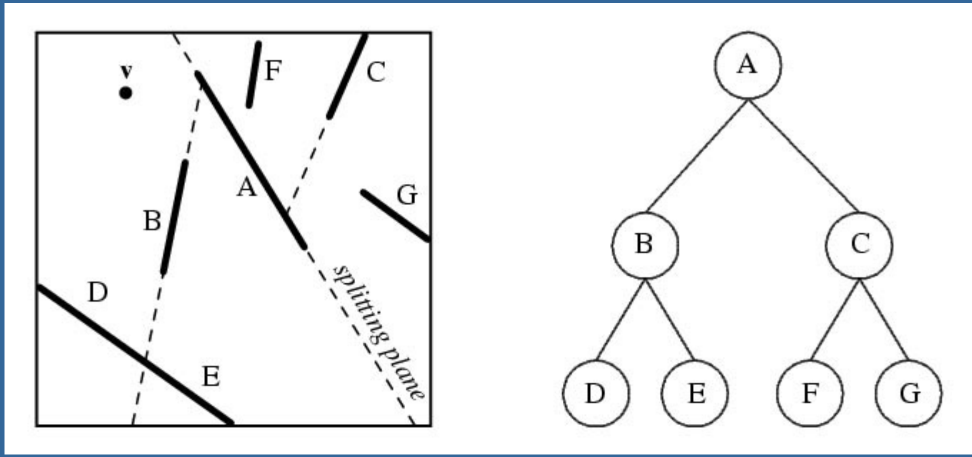


- Works in the same way for polygon-aligned BSP trees --- but that gives exact sorting

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

Drawing Back-to-Front {
```
     recurse on farther side of P;
     Draw P;
     Recurse on hither side of P;
}// farther/hither is with respect to eye pos.
```



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

Lecture 7.2: Collision Detection

- 3 types of algorithms:
  - With rays
    - Fast but not exact
  - With BVH
    - Slower but exact
    - You should be able to write pseudo code for BVH/BVH test for coll det between two objects.
  - For many many objects.
    - Course pruning of ”obviously” non-colliding objects
    - E.g., Use a grid with an object list per cell, storing the objects that intersect that cell. For each cell with list length > 1, test those against each other with a more exact method.