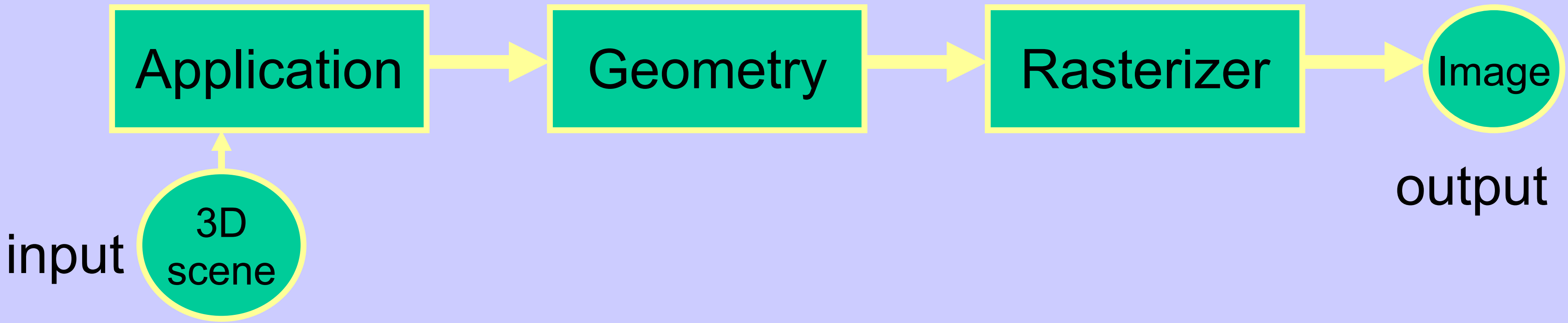
Full-time wrapup

Lecture 1

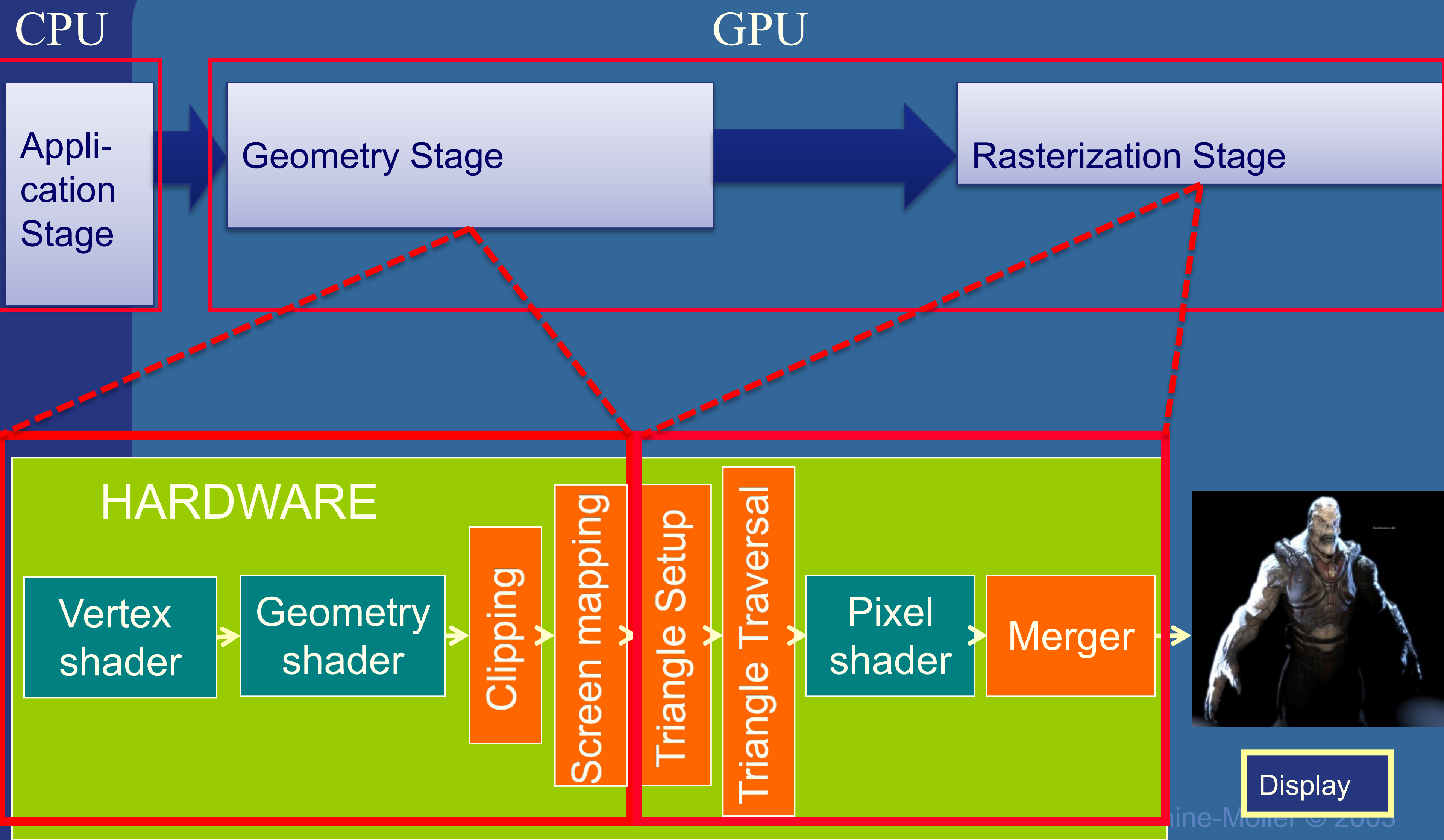
- Application-, geometry-, rasterization stage
- Real-time Graphics pipeline
- Modelspace, worldspace, viewspace, clip space, screen space
- Z-buffer
- Double buffering
- Screen tearing

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

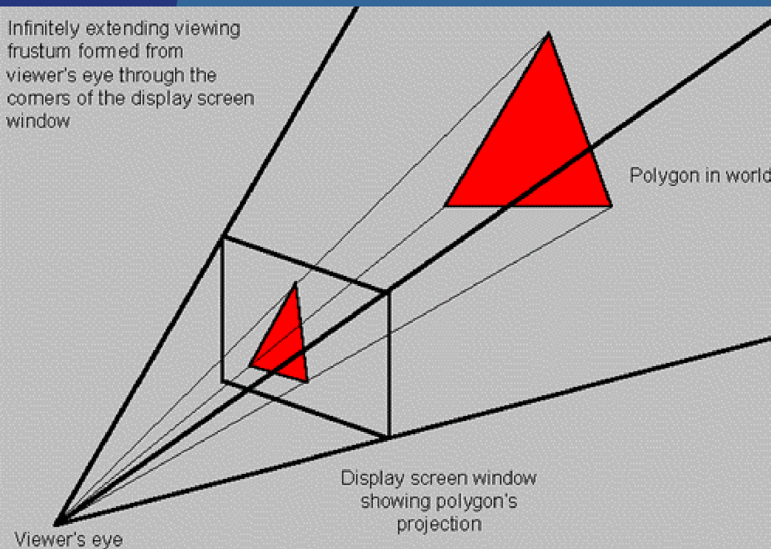


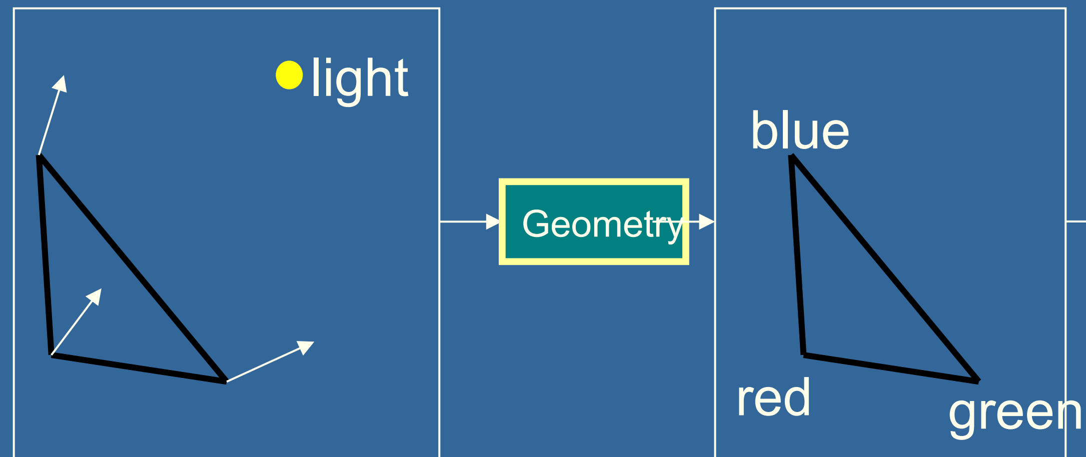
Hardware design

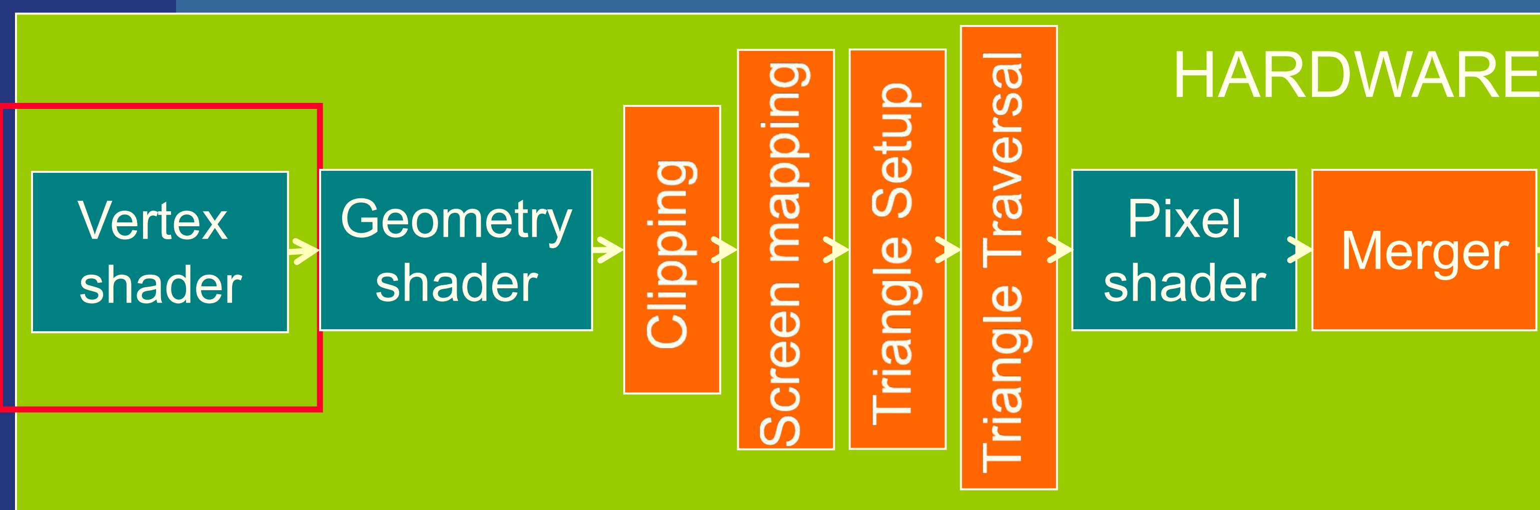
Geometry Stage

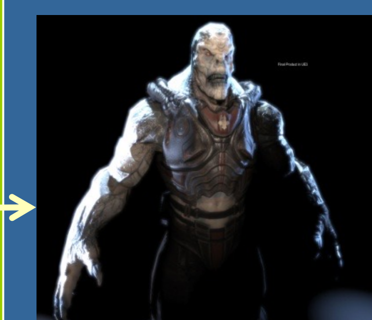
Vertex shader:

- Lighting (colors)
- Screen space positions









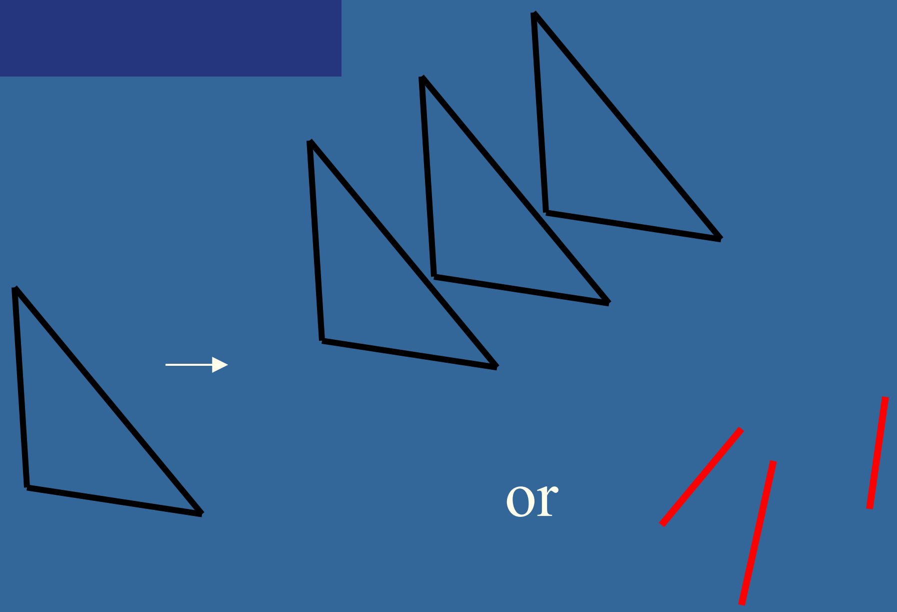
Display

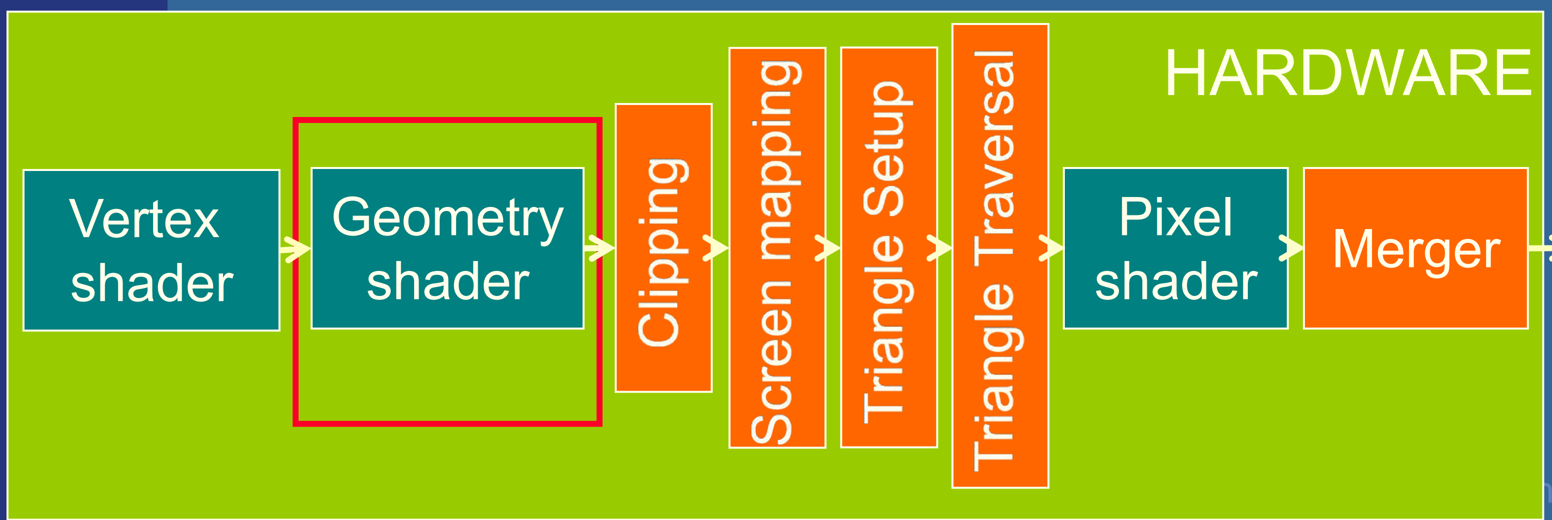
Hardware design

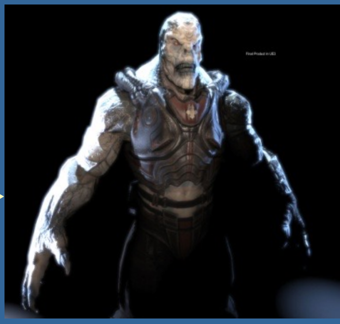
Geometry Stage

Geometry shader:

- One input primitive
- Many output primitives





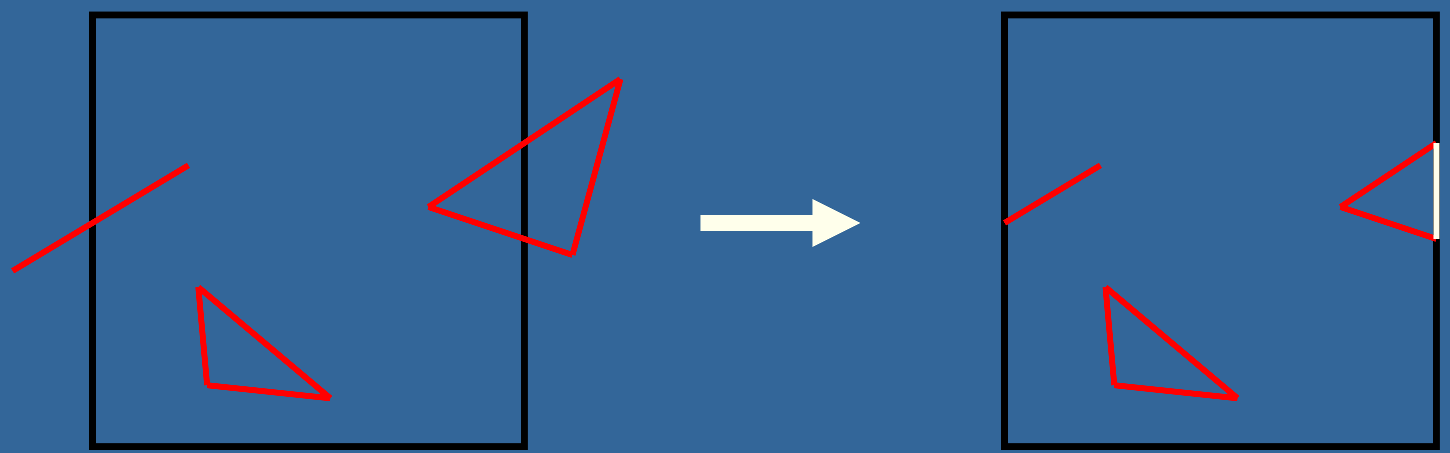


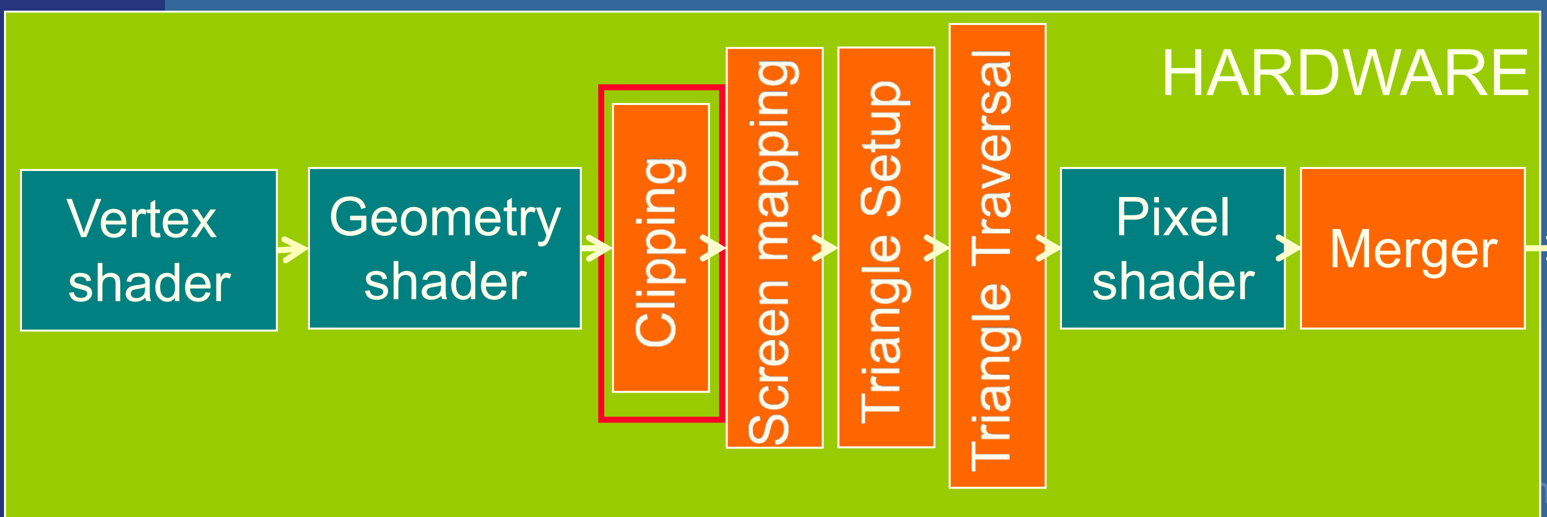
Display

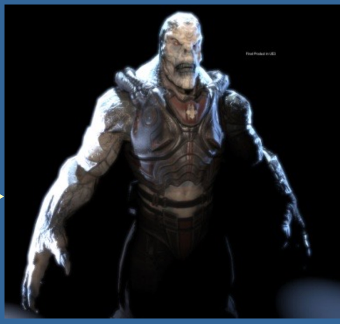
Hardware design

Geometry Stage

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







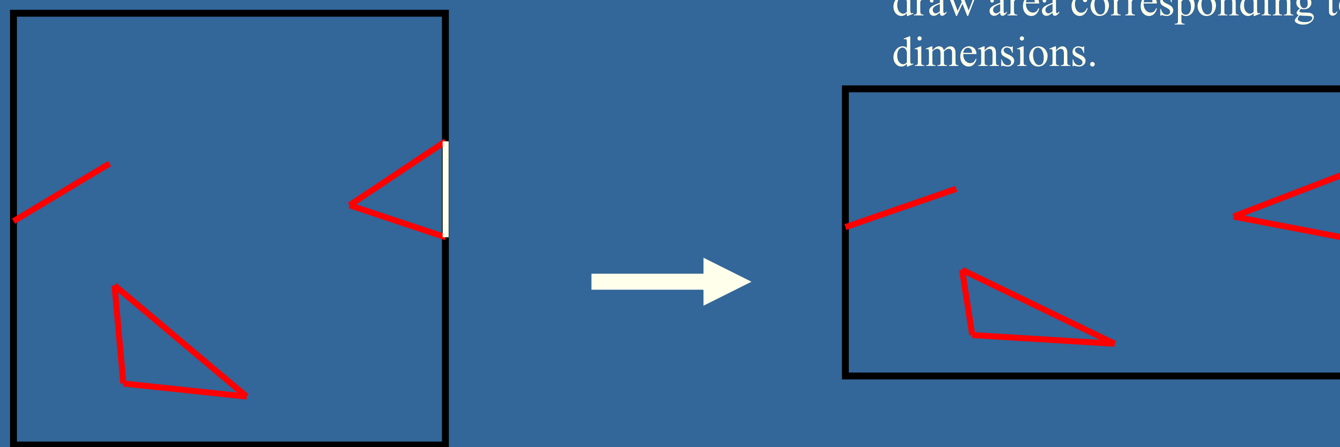
Display

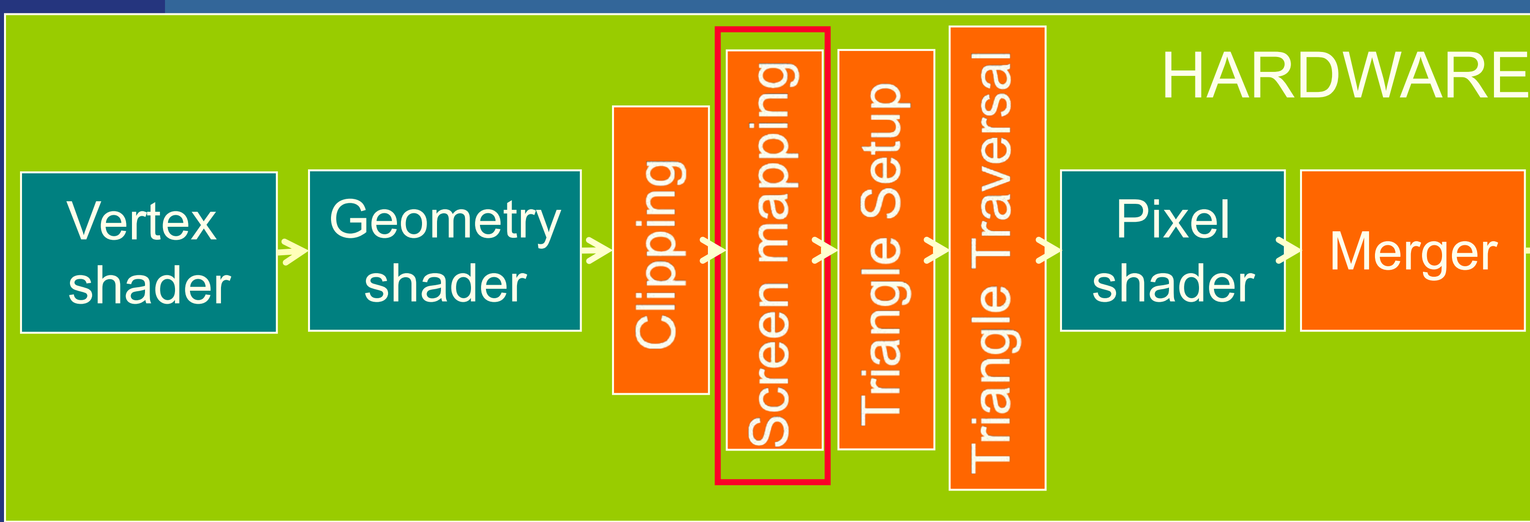
Hardware design

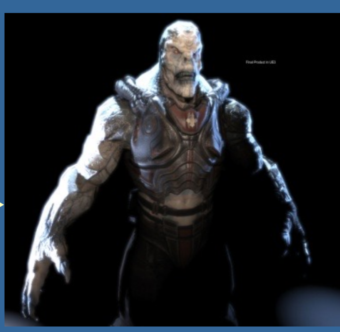
Rasterizer Stage

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.





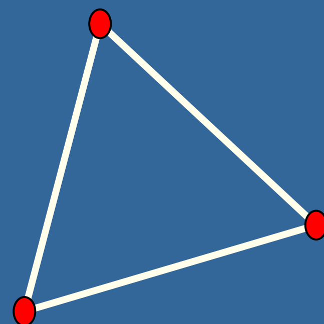


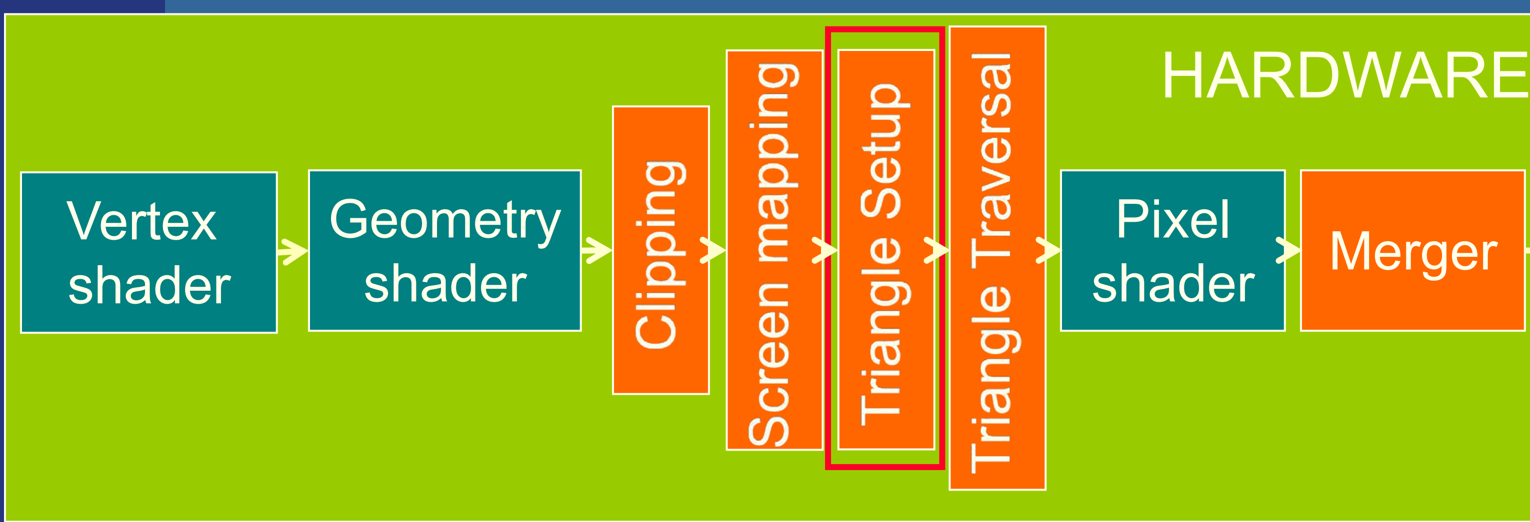
Display

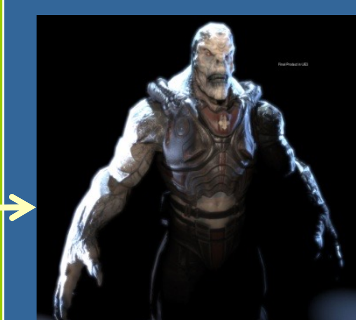
Hardware design

Rasterizer Stage

Collects three vertices into one triangle





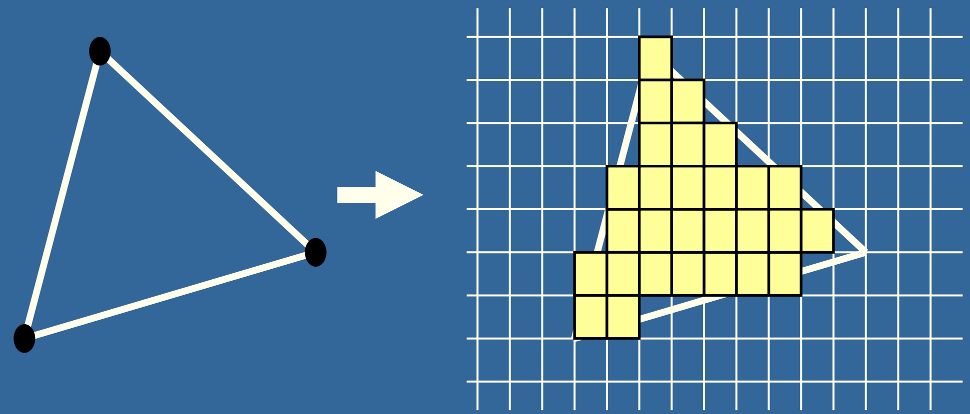


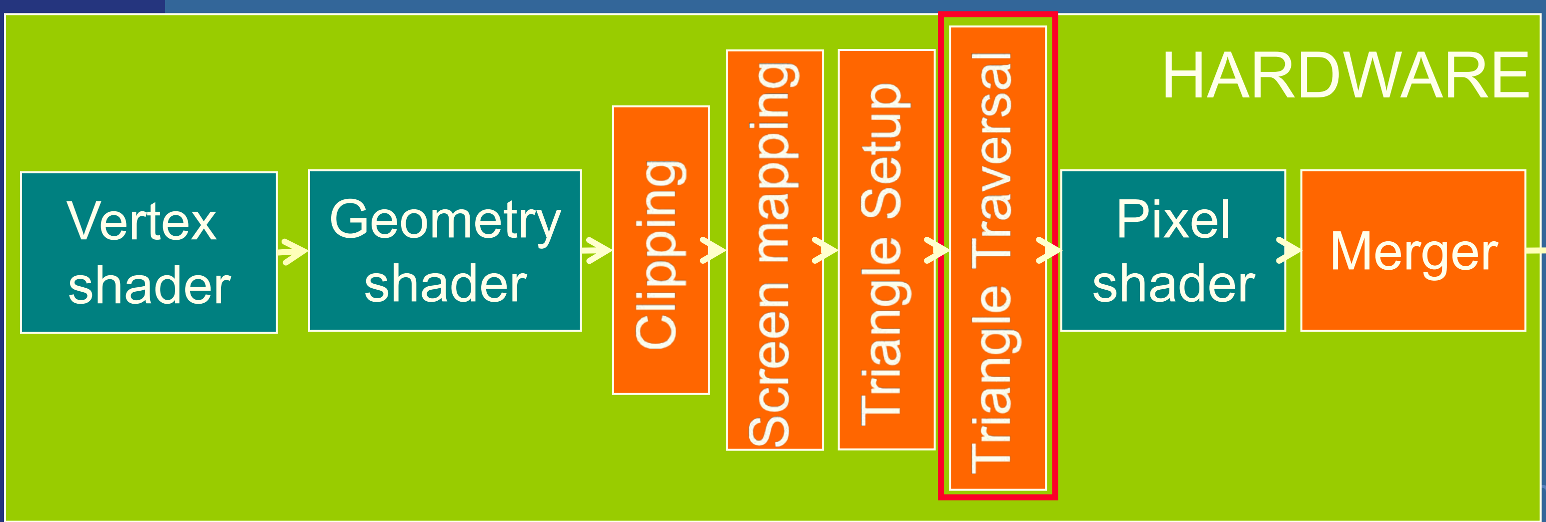
Display

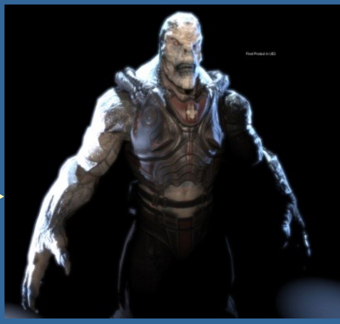
Hardware design

Rasterizer Stage

Creates the fragments/pixels for the triangle



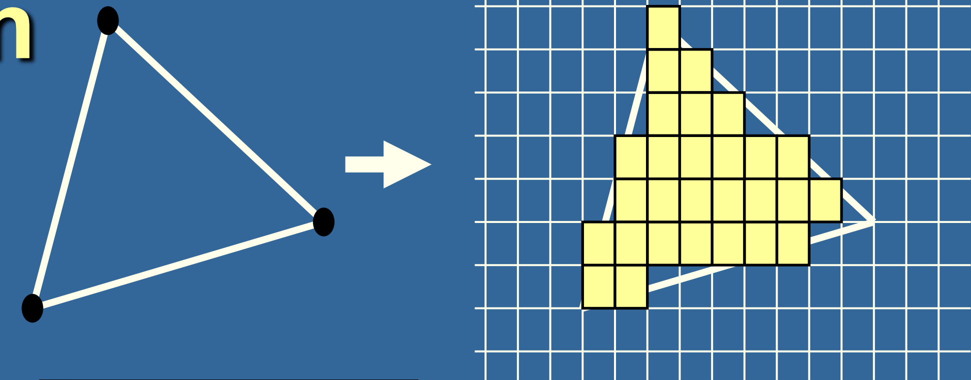


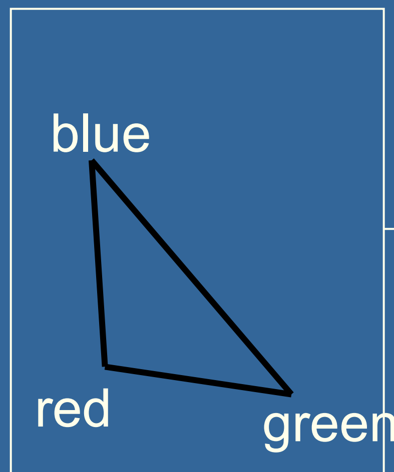


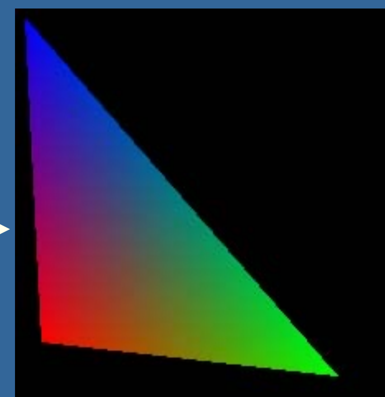
Display

Hardware design

Rasterizer Stage

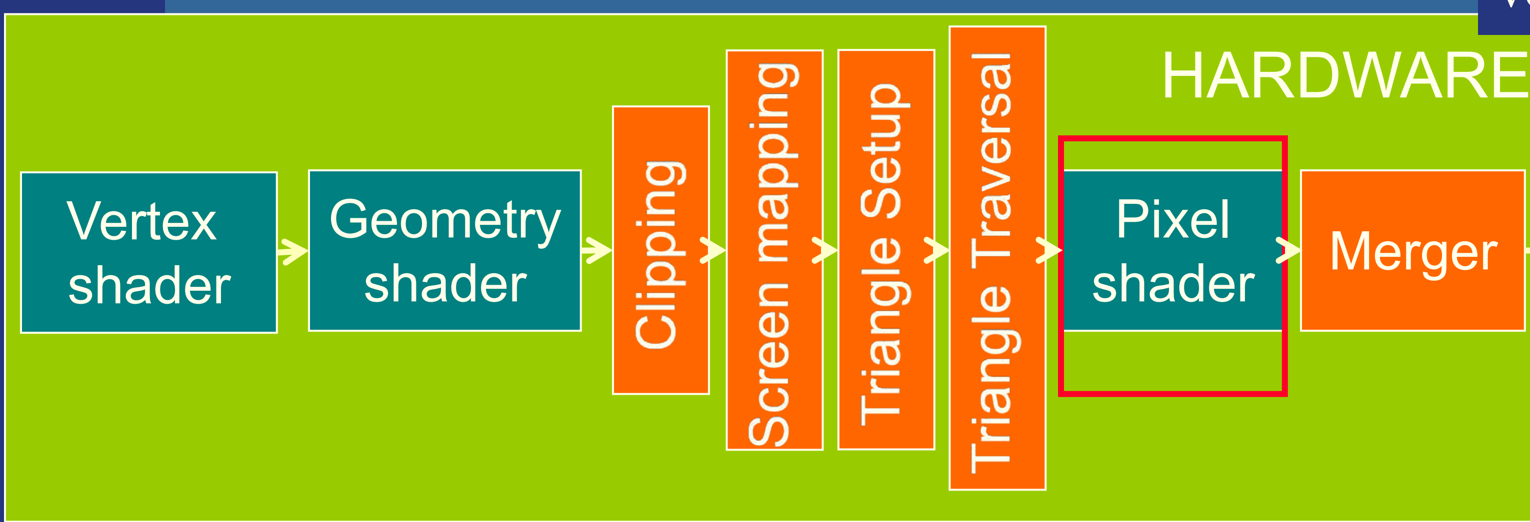


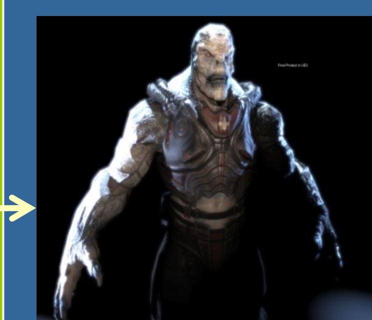




Rasterizer

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





Display

Hardware design

Rasterizer Stage

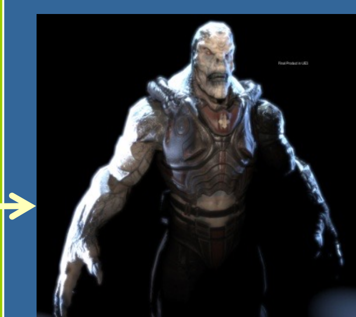
The merge units update the frame buffer with the pixel's color

Frame buffer:

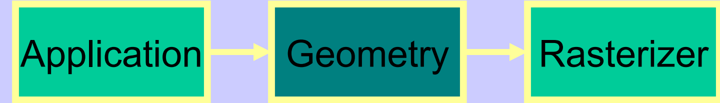
- Color buffers
- Depth buffer
- Stencil buffer

HARDWARE

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

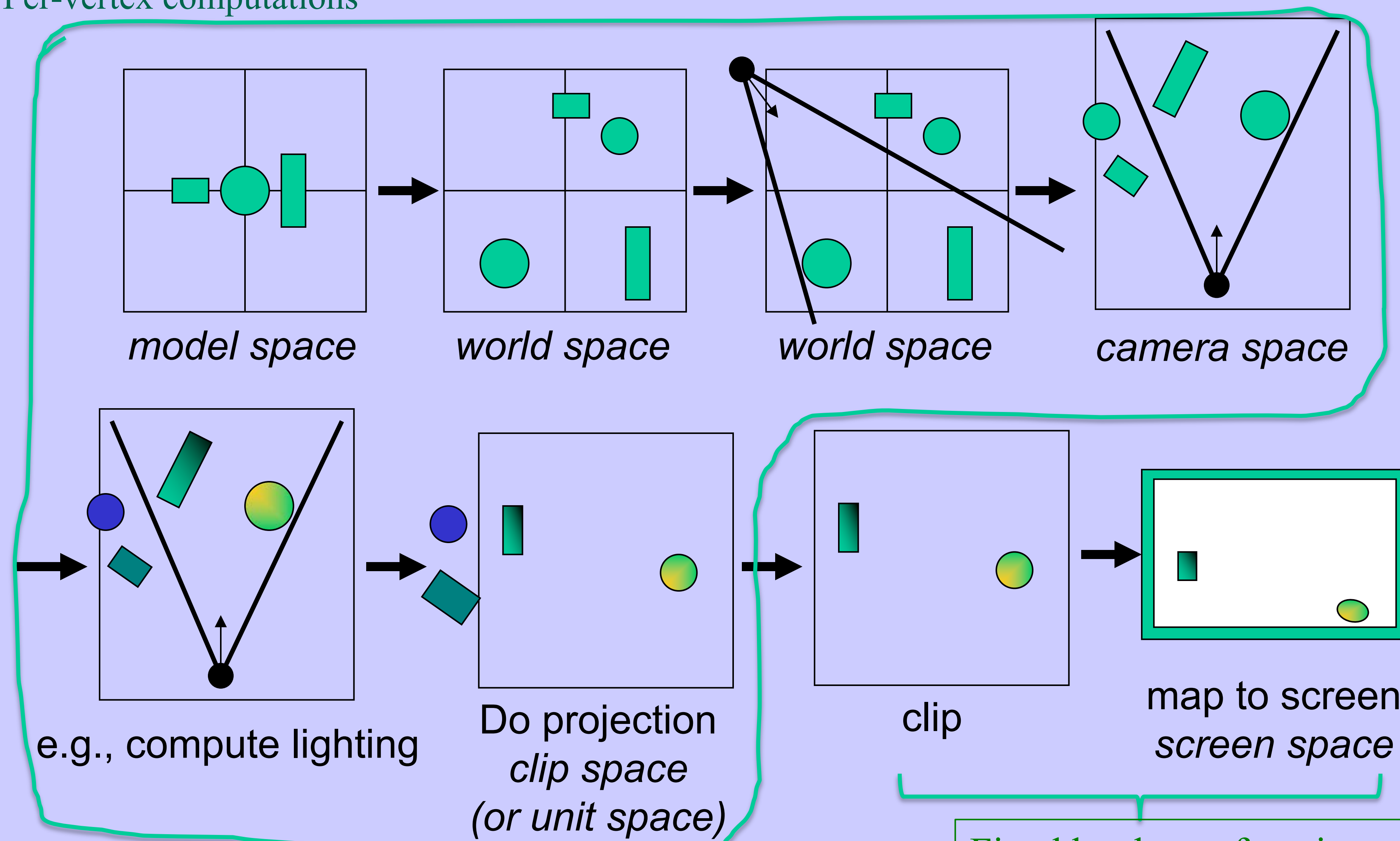


Display



GEOMETRY – transformation summary

Per-vertex computations

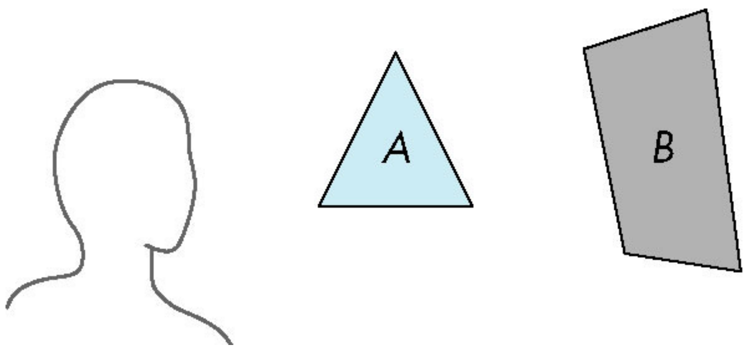


Fixed hardware function

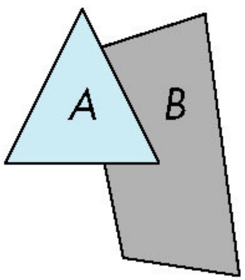
Done in vertex shader

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



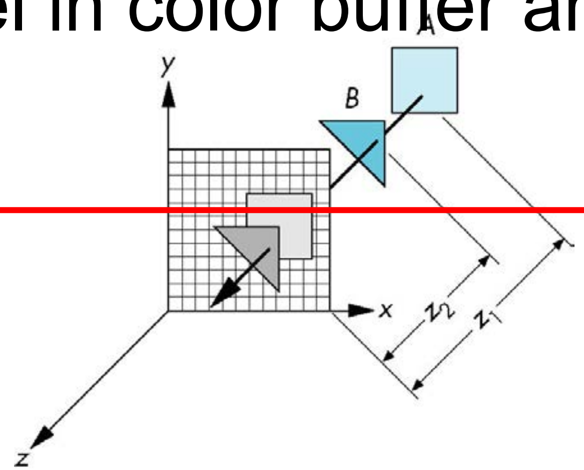
Fill B then A

- 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.

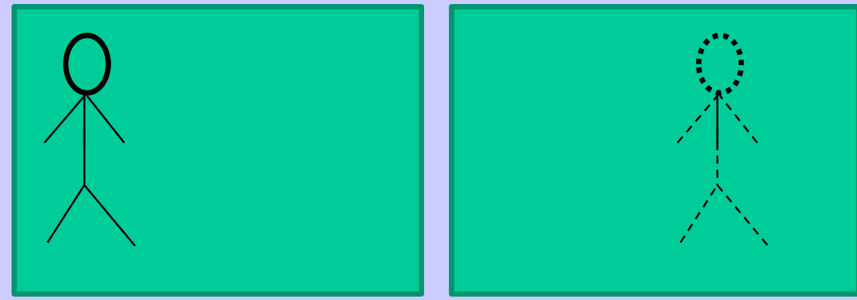
z-Buffer Algorithm

- 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)

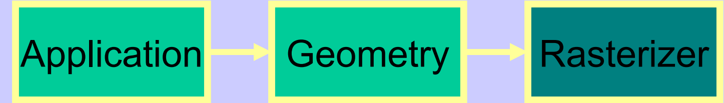
Last fully finished
drawn frame.

Back buffer
(rgb color buffer)

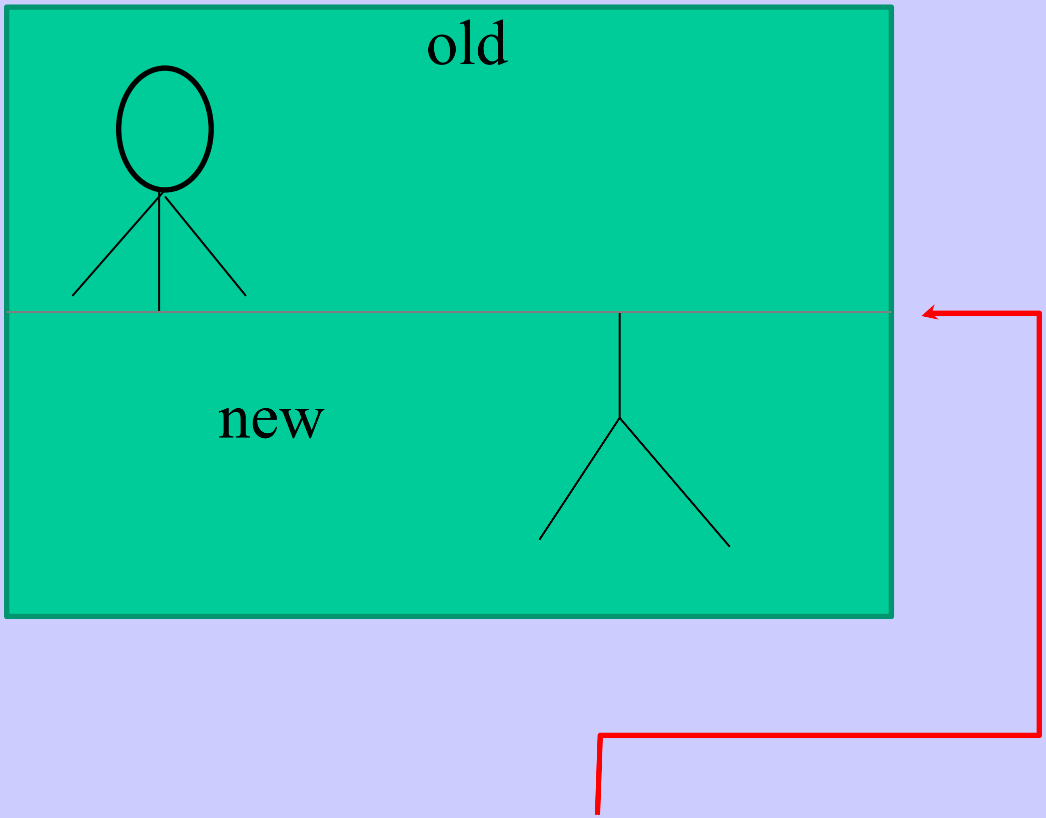
Color buffer we draw to.
Not displayed yet.

- 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



Example if the swap happens here (w.r.t the screen refresh rate). Solution: use vsynch to swap buffers after monitor has ”updated” the screen. See page 1011-1012.

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.

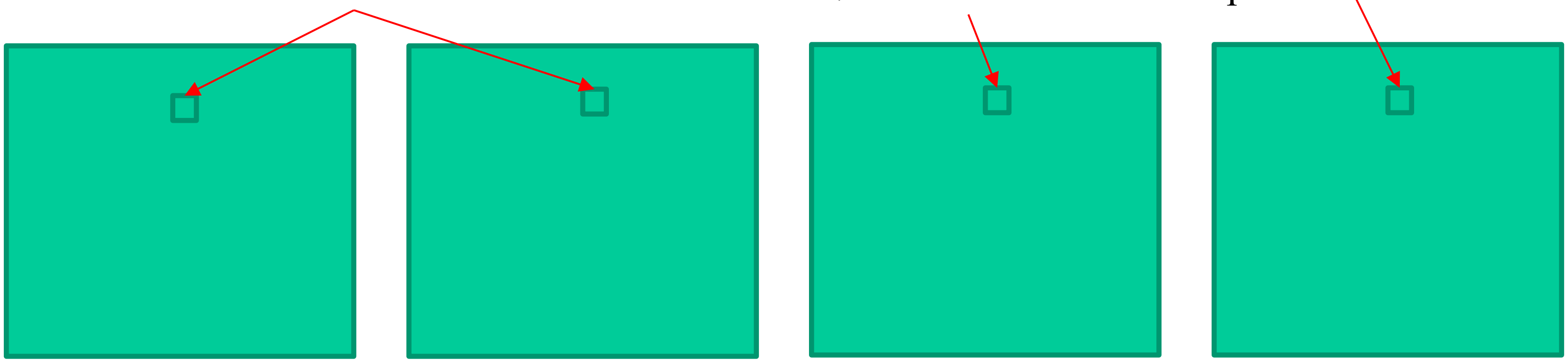
The default frame buffer:

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

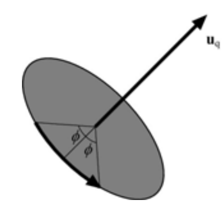
- Transformation pipeline: ModelViewProjection matrix
- Scaling, rotations, translations, projection
- Cannot use same matrix to transform normals

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

$(M^{-1})^T=M$ 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.

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

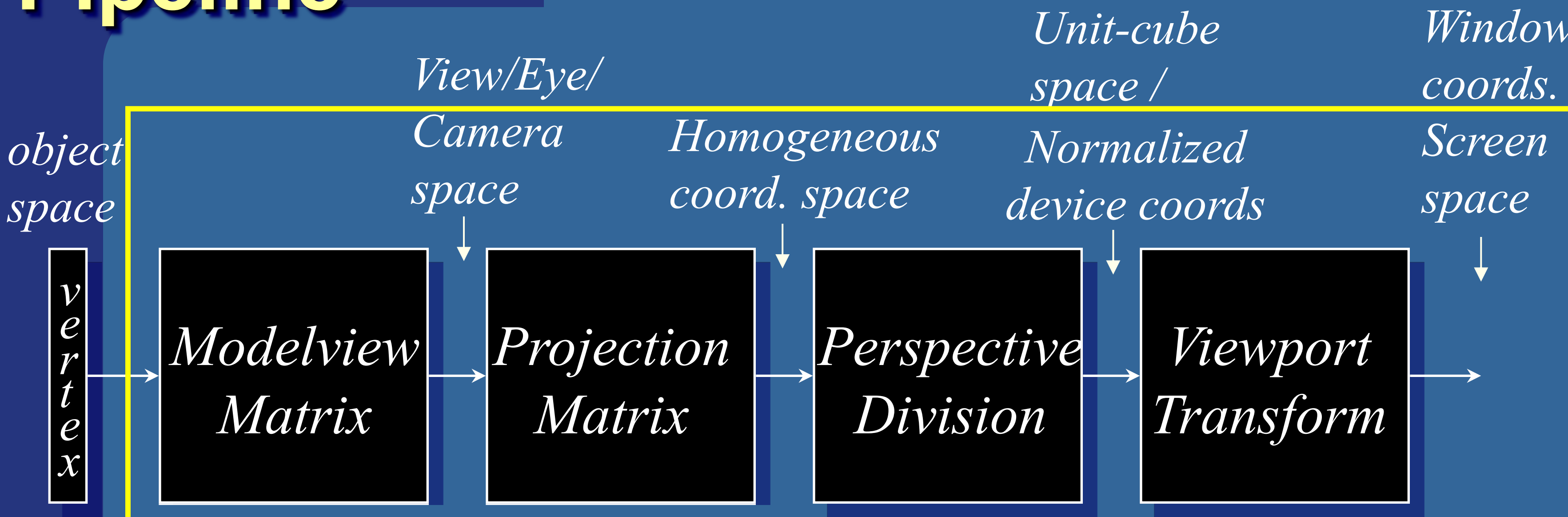


  - …represents a rotation of 2ϕ radians around axis $\mathbf{u}_q$ of point $\mathbf{p}$
- Understand the simple DDA algorithm
- Bresenhams line-drawing algorithm

Lecture 2:

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.

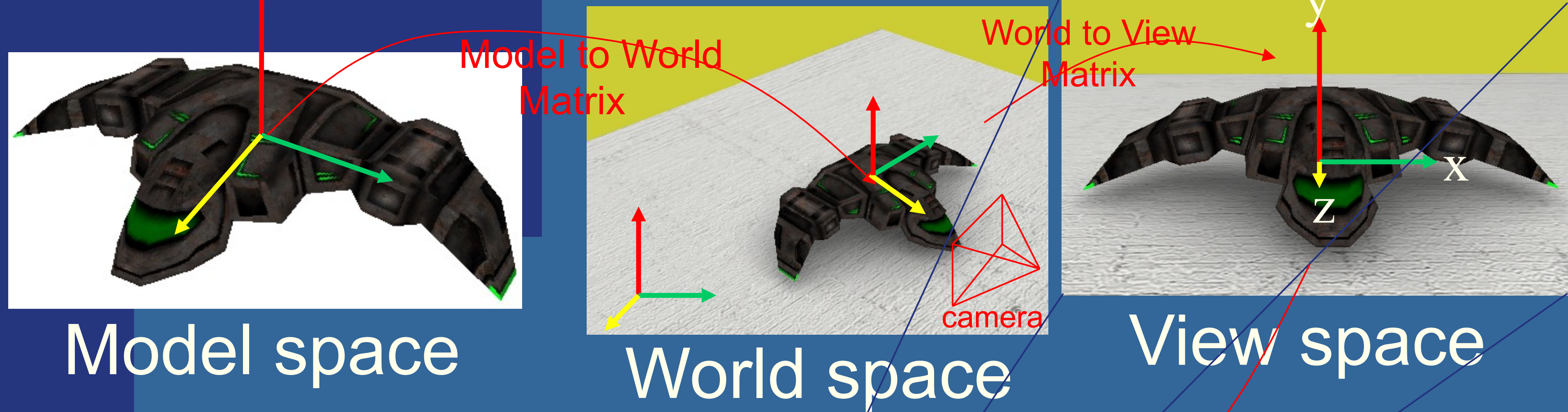


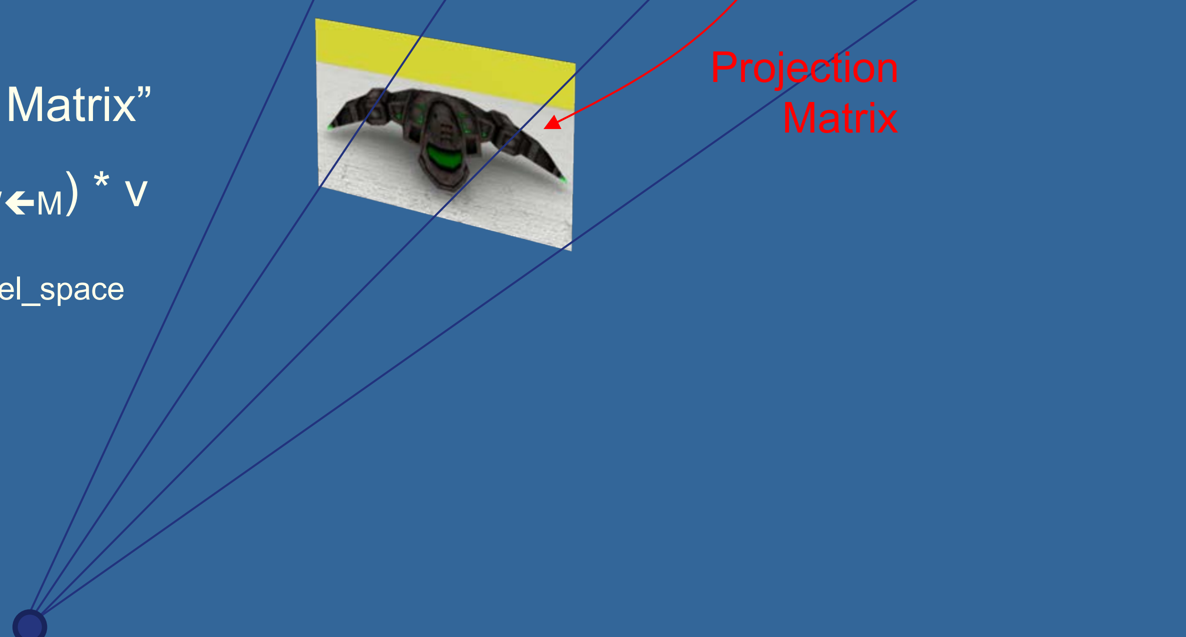
What we do in the vertex shader:

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

The perspective division is then done automatically by the GPU before the GPU does clipping and screen mapping.

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:

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

Translation part

Rotation part

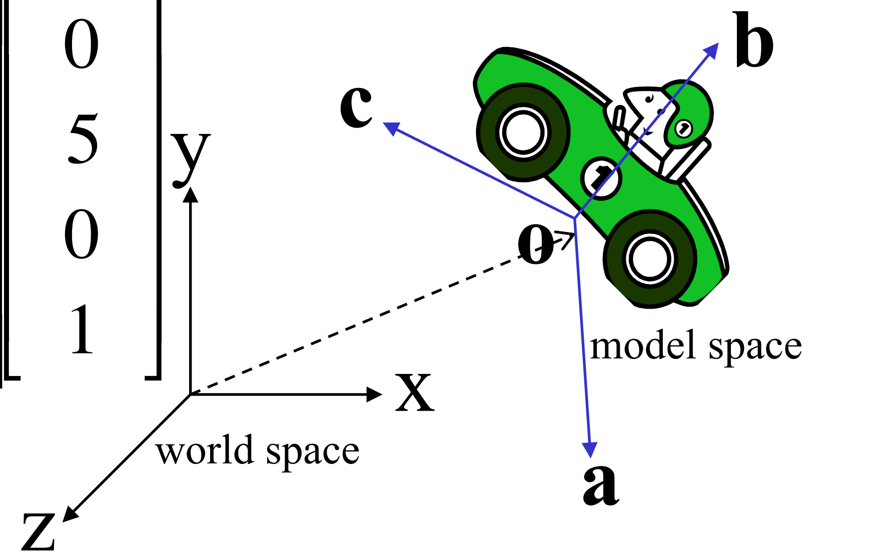
- A vector (direction): $\mathbf{d} = \begin{pmatrix} d_x & d_y & d_z & 0 \end{pmatrix}^T$
- Translation of vector: $\mathbf{Td} = \mathbf{d}$

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)

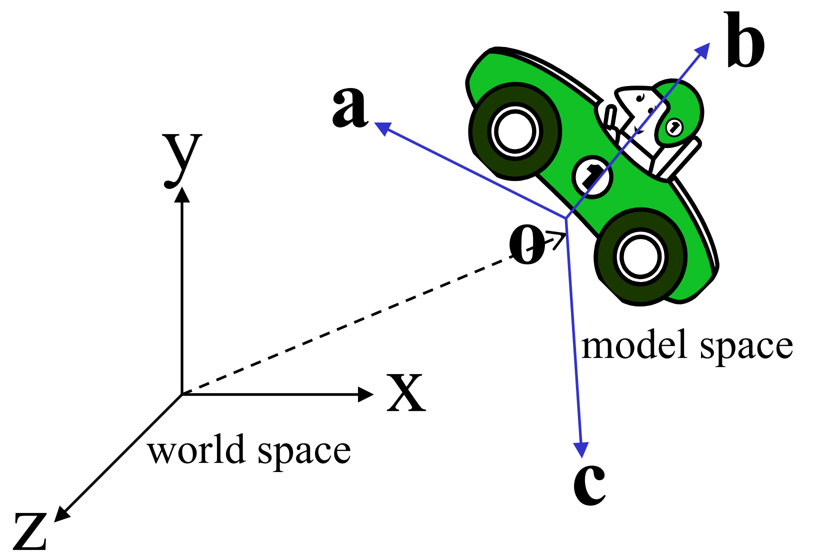
E.g.: $\mathbf{p}_{\text{world}} = M_{m\to w}\, \mathbf{p}_{\text{model}} = M_{m\to w}\,(0,5,0,1)^T = 5\,\mathbf{b}\ \ (+\,\mathbf{o})$

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]

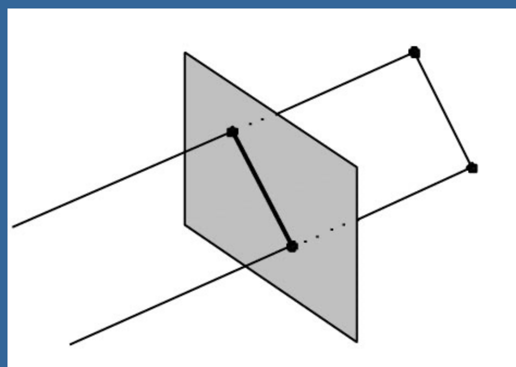
X: One step along **a** results in $\mathbf{a}_x$ steps along world space axis x.
One step along **b** results in $\mathbf{b}_x$ steps along world space axis x.
One step along **c** results in $\mathbf{c}_x$ steps along world space axis x.

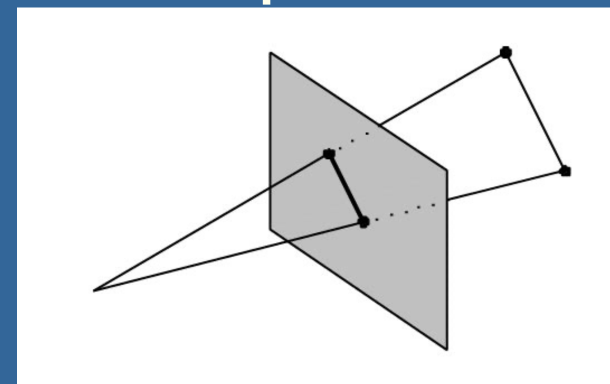
The x-coord for **p** in *world space* (instead of modelspace) is thus $[a_x\ b_x\ c_x]\mathbf{p}$.
The y-coord for **p** in world space is thus $[a_y\ b_y\ c_y]\mathbf{p}$.
The z-coord for **p** in world space is thus $[a_z\ b_z\ c_z]\mathbf{p}$.

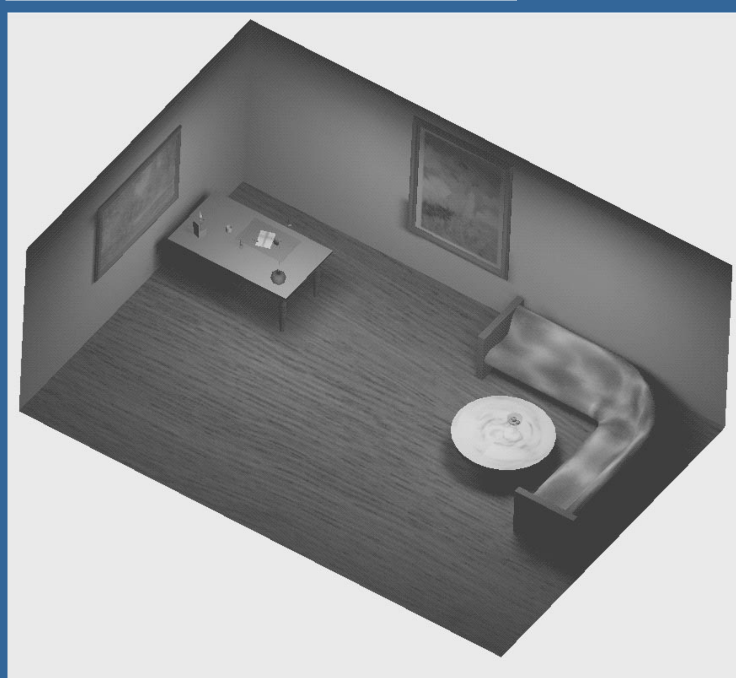
With the translation **o** we get $\mathbf{p}_{\text{worldspace}} = M_{\text{model-to-world}}\ \mathbf{p}_{\text{modelspace}}$

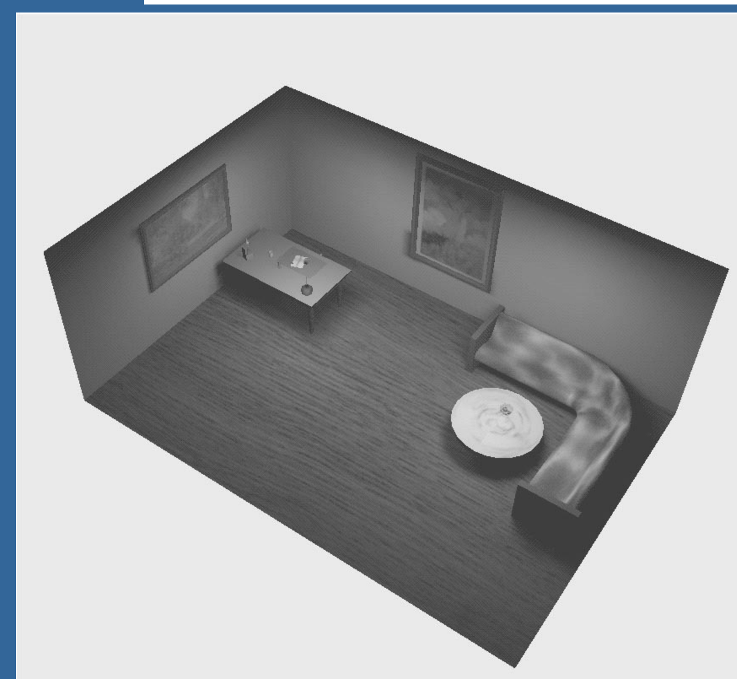
Projections

- Orthogonal (parallel) and Perspective

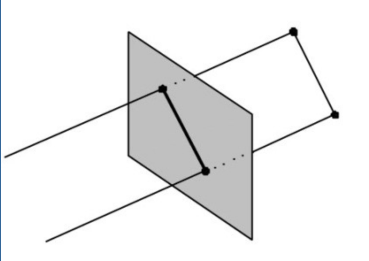






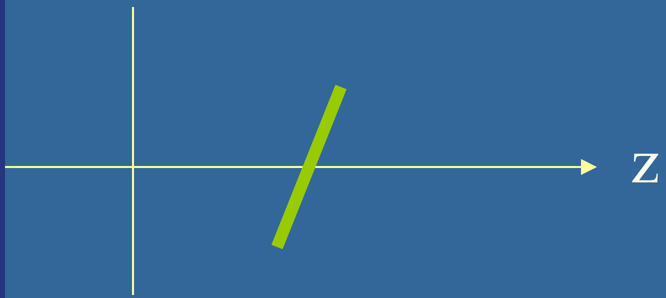


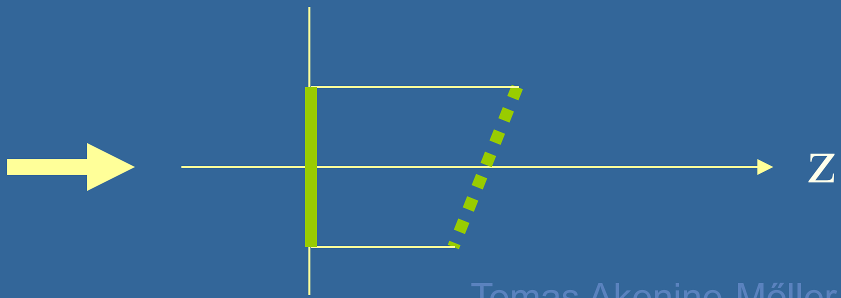
Orthogonal projection



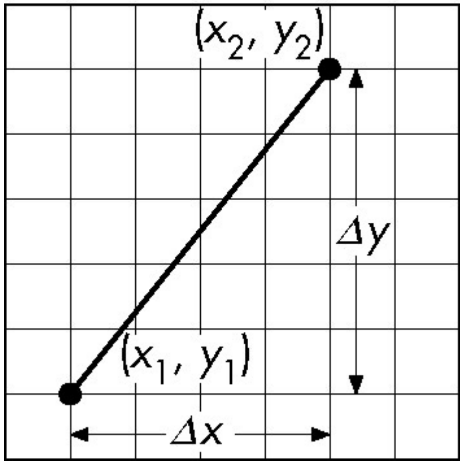
- Simple, just skip one coordinate
  - Say, we're looking along the z-axis
  - Then drop z, and render

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





DDA Algorithm



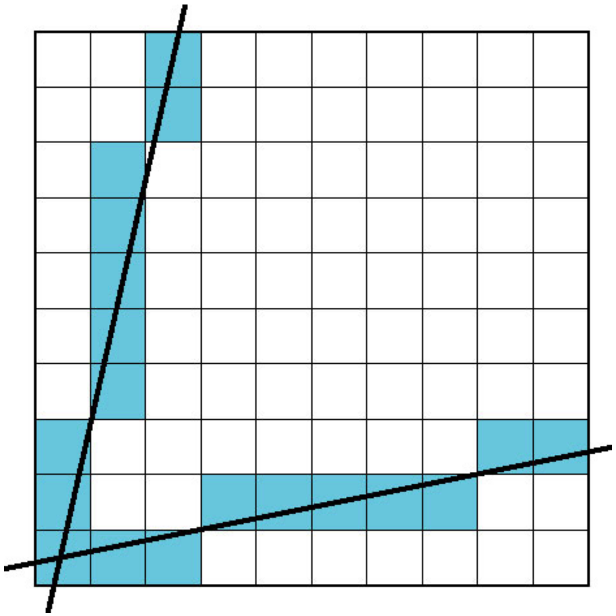
- Digital Differential Analyzer
  - DDA was a mechanical device for numerical solution of differential equations
  - Line y=kx+ m satisfies differential equation

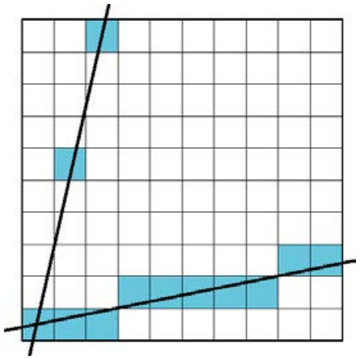
    $dy/dx = k = \Delta y/\Delta x = y_2-y_1/x_2-x_1$
- Along scan line $\Delta x = 1$

```
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





Otherwise we get problem for steep slopes

02. Vectors and Transforms

Very Important!

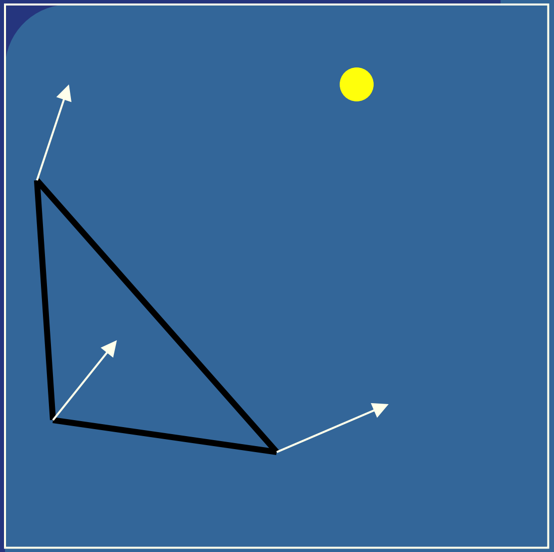
- The problem with DDA is that it uses floats which was slow in the old days
- Bresenhams algorithm only uses integers

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

Lecture 3.1: Shading

- Ambient, diffuse, specular, emission
  - Formulas,
  - Phongs vs Blinns highlight model.
- Half vector: $\boldsymbol{h} = \frac{\boldsymbol{l}+\boldsymbol{v}}{||\boldsymbol{l}+\boldsymbol{v}||}$
- Flat, Goraud, and Phong shading
- Fog
- Transparency
- Gamma correction

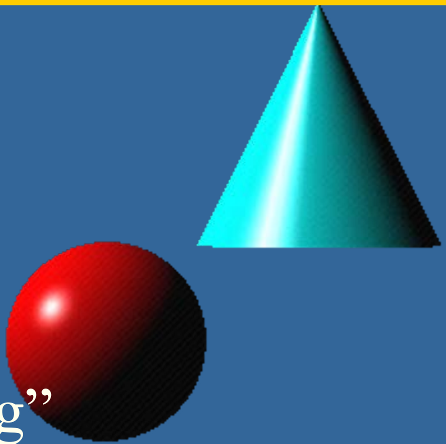
Lighting



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:

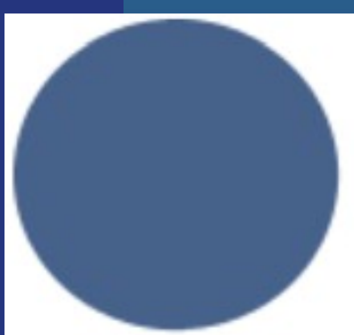
Ambient: $\mathbf{i}_{amb} = \mathbf{m}_{amb}\, \mathbf{l}_{amb}$

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

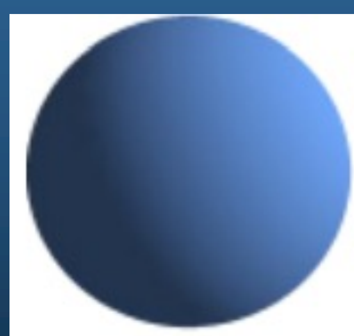
Specular:

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

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

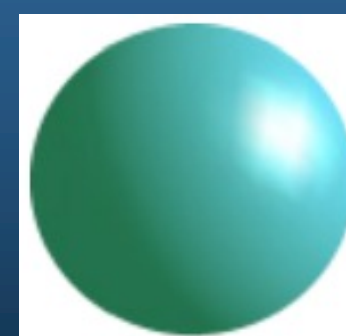


Ambient



Amb + Diff

Amb + Diff + Spec



Amb + Diff + Spec + Em

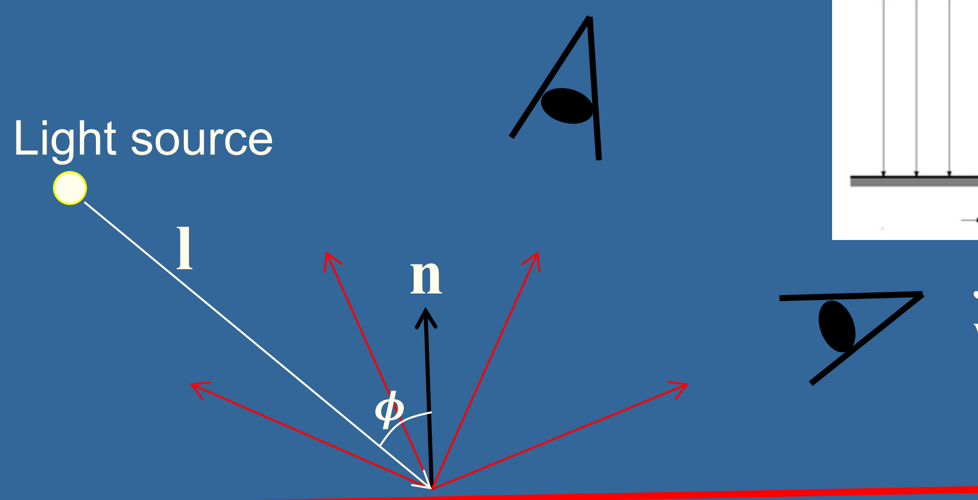
The ambient/diffuse/specular/emission model

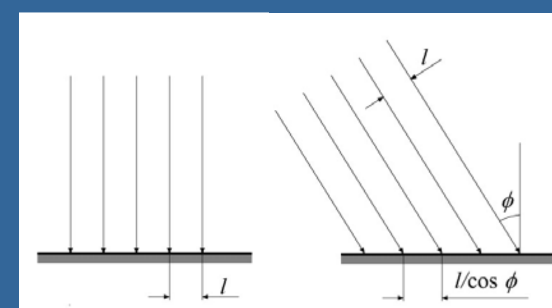
- The most basic real-time model:
- Light interacts with material and change color at bounces:

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

- 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

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

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

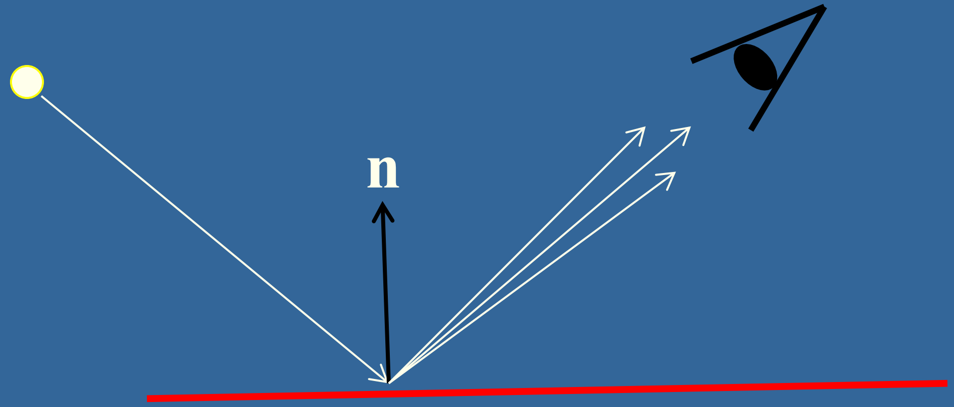
The ambient/diffuse/specular/emission model

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

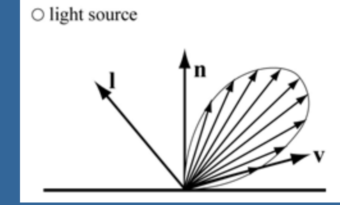
$$\mathbf{outColor}_{rgb} \sim \mathbf{material}_{rgb} \otimes \mathbf{lightColor}_{rgb}$$

- 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

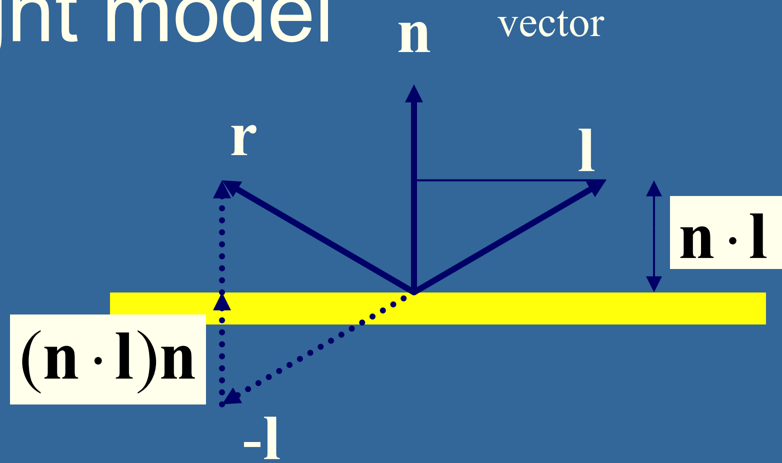


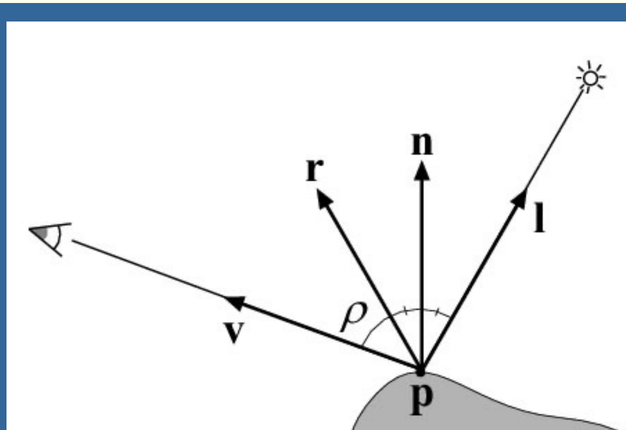
- Phong specular highlight model
- Reflect $\mathbf{l}$ around $\mathbf{n}$:

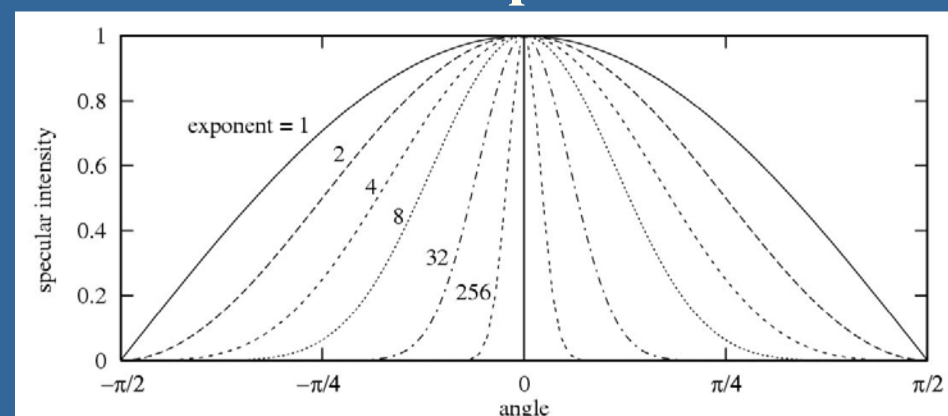
$$\mathbf{r} = -\mathbf{l} + 2(\mathbf{n}\cdot\mathbf{l})\mathbf{n}$$

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

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: $(\mathbf{n}\cdot\mathbf{h})^m$

Specular: Blinn's specular highlight model

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

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

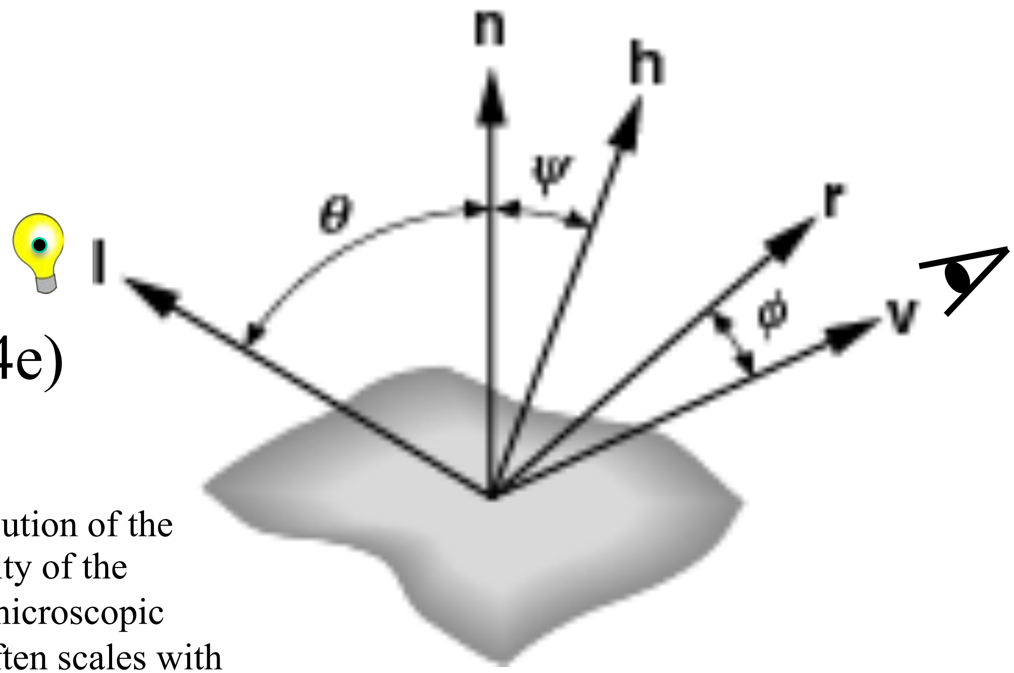
h is halfway between **l** and **v**

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

$\psi = \phi/2$

Must then adjust exponent

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

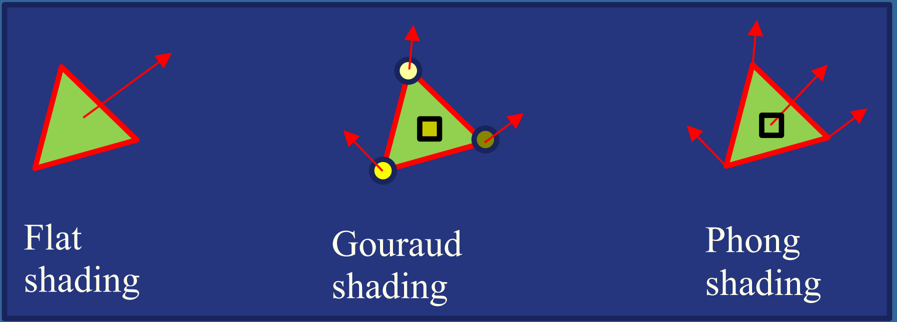


If the surface is rough, there is a probability distribution of the microscopic normals **n**. This means that the intensity of the reflection is decided by how many percent of the microscopic normals are aligned with **h**. And that probability often scales with how close **h** is to the macroscopic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

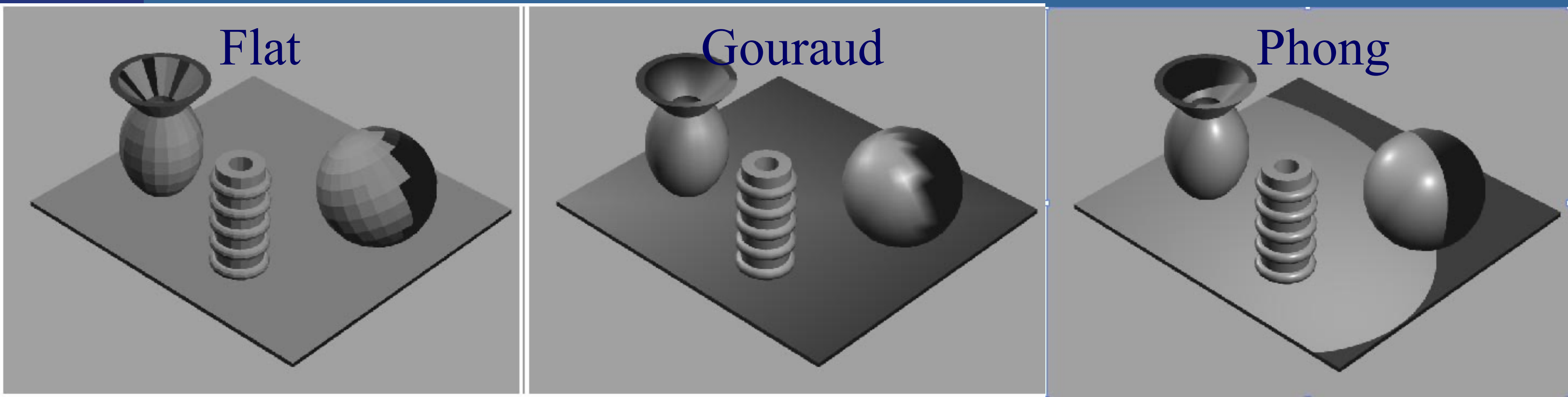
$$\mathbf{i}_{spec} = \max(0, (\mathbf{h}\cdot\mathbf{n})^{m_{shi}})\mathbf{m}_{spec} \otimes \mathbf{s}_{spec}$$

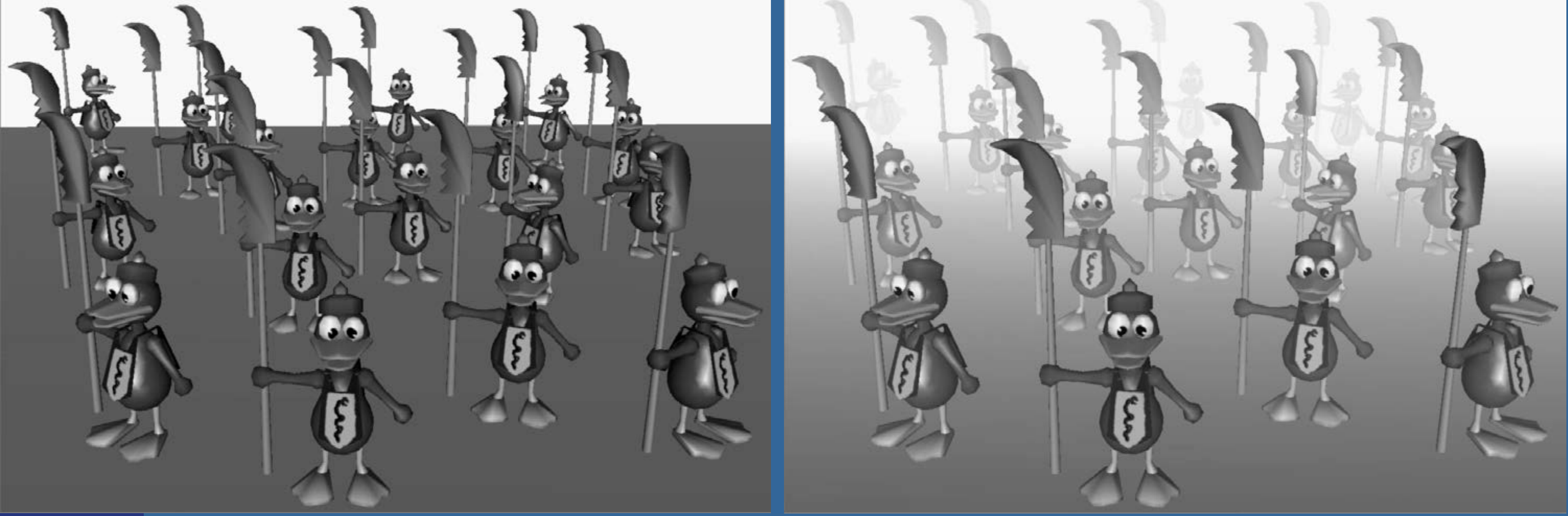
03. Shading:

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)

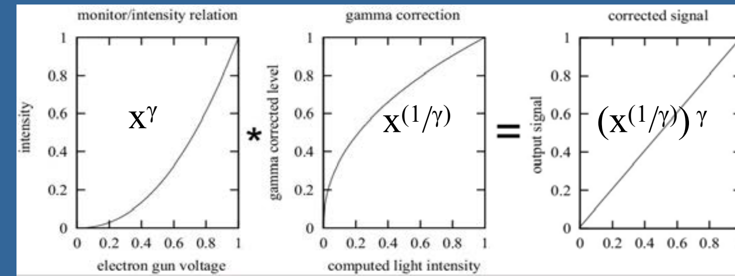
Rendered object

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.

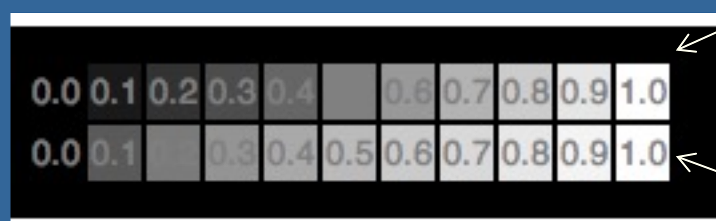
$$c = c_i^{(1/\gamma)}$$



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.



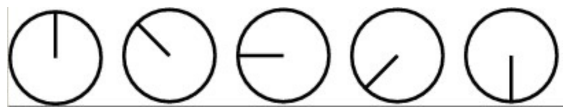
Gamma of 2.2. Better distribution for humans. Perceived as linear.

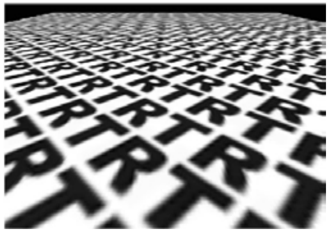
Truly linear intensity increase.

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.

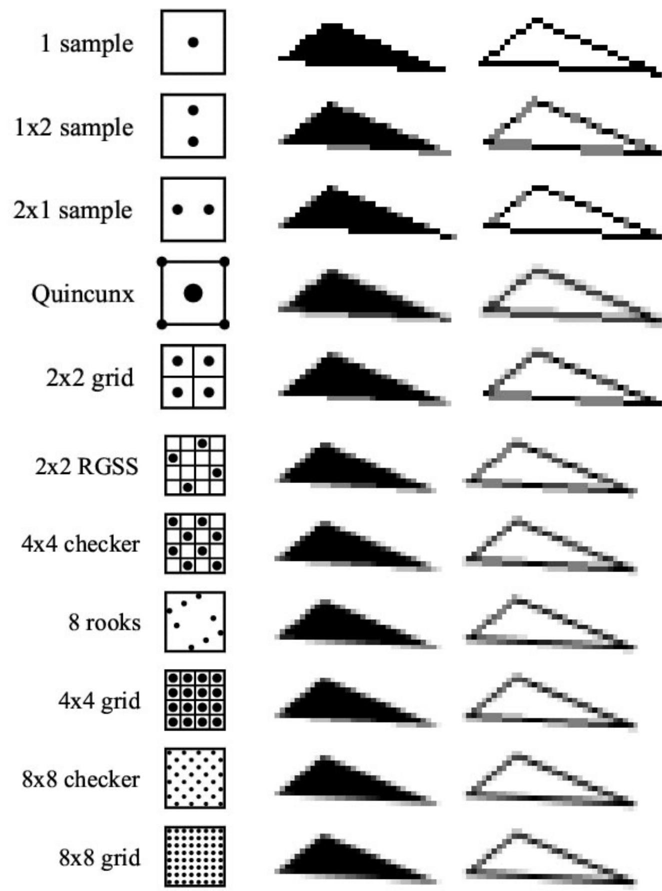
Leture 3.2: Sampling, filtrering, and Antialiasing

- When does it occur?
  - In 1) pixels, 2) time, 3) texturing

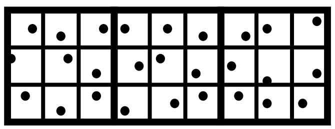




- Supersampling schemes:
- Quincunx + weights

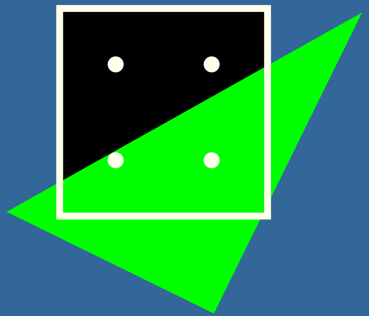


- Jittered sampling
  - Why is it good?

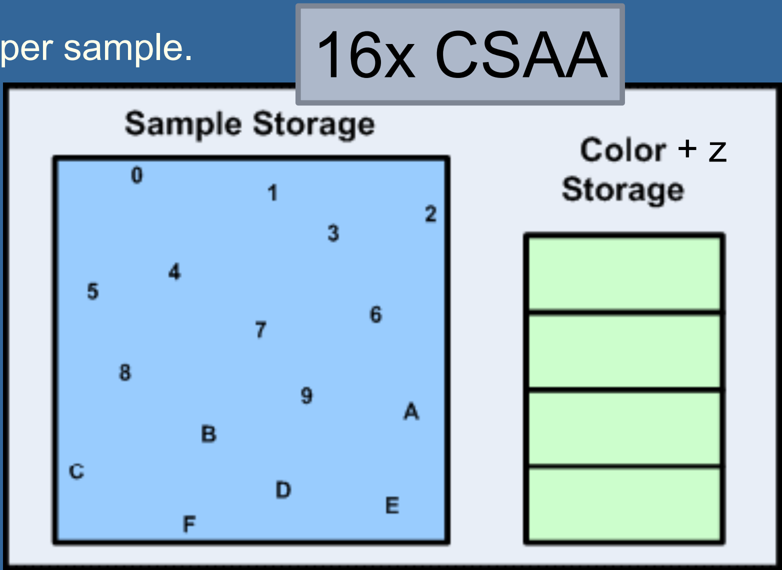


- Supersampling vs multisampling vs coverage sampling

SSAA, MSAA and CSAA



- Super Sampling Anti Aliasing
  - Stores duplicate information (color, depth, stencil) for each sample and fragment shader is run for each sample.
  - Corresponds to rendering to an oversized buffer and downfiltering.
- Multi Sampling Anti Aliasing
  - Shares some information between samples. E.g:
    - **Result of Frament shader – Frag. shader is only run once per** rasterized **fragment**.
    - But stores a color per sample and typically also a stencil and depth-value per sample
- Coverage Sampling Anti Aliasing
  - Idea: Don't even store **unique** color and depth per sample. In each subsample, store index into a per-pixel list of 4-8 colors+depths.
  - I.e., for 4-8 polygons, store their coverage.
  - Fragment shader executed once per rasterized fragment
  - E.g., Each sample holds a 2-bit index into a table (a storage of up to four colors per pixel)



04. Texturing

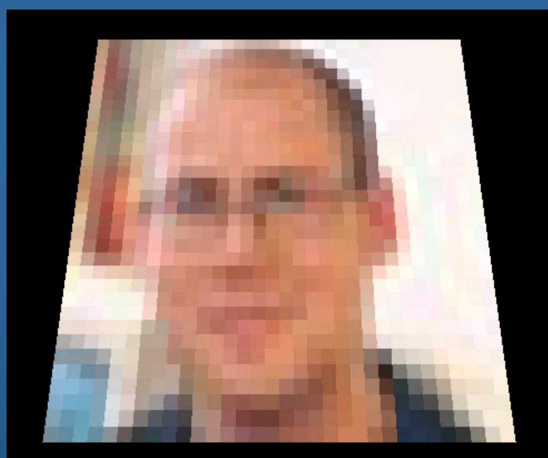
Texturing:

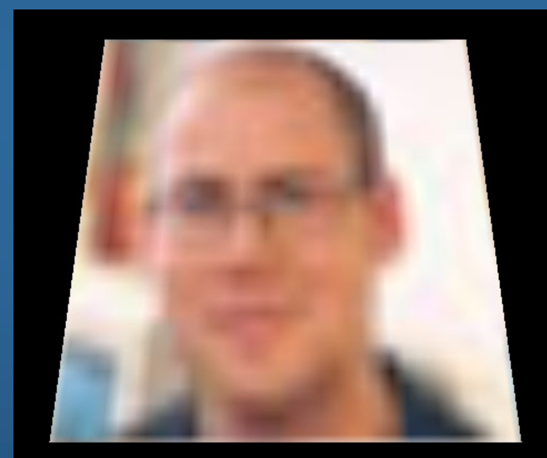
- Real-time Filtering:
  - Magnification – nearest neightbor, linear
  - Minification – nearest neighbor, bilinear, bilinear mipmap filtering & trilinear-filtered mipmap lookup.
  - Why not sinc filter as real-time filter?
  - Mipmaps + their memory cost
  - How compute bilinear/trilinear filtering
  - Number of texel accesses for trilinear filtering
  - Anisotropic filtering – take several trilinear-filtered mipmap lookups along the line of anisotropy (e.g., up to 16 lookups)
- 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)

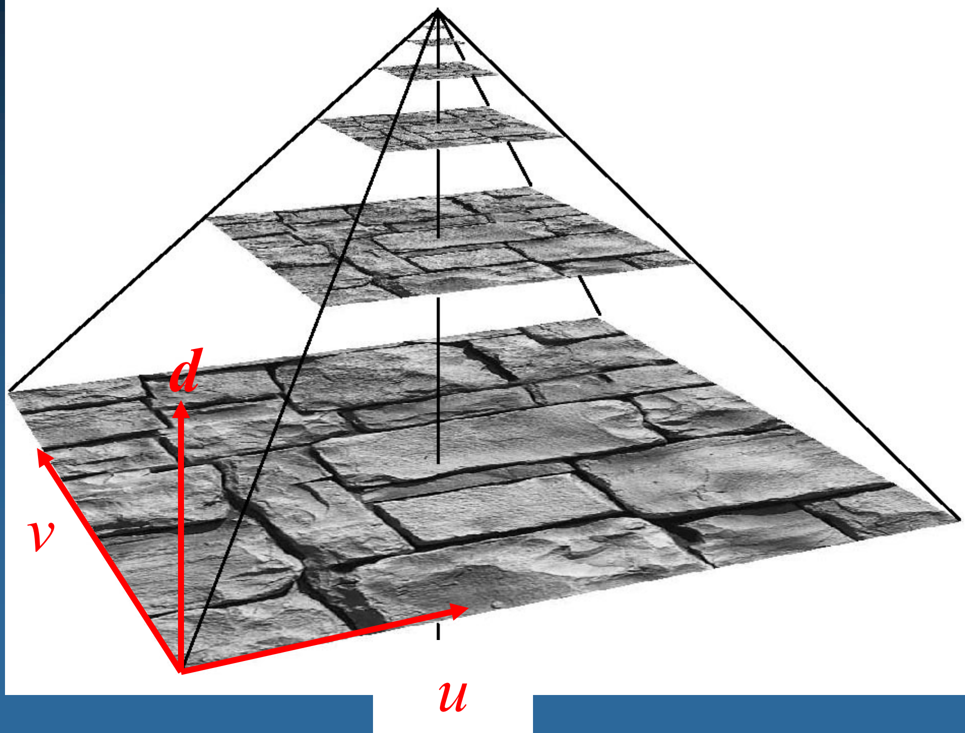




- For minification: nearest, linear and…
  - Bilinear – using mipmapping
  - Trilinear – using mipmapping
  - Anisotropic – up to 16 mipmap lookups along line of anisotropy

Mipmapping

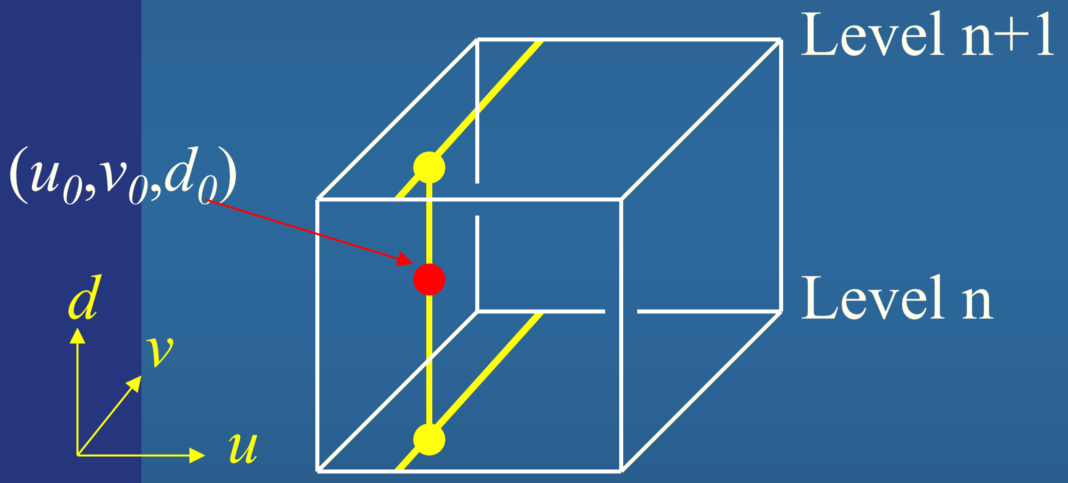
- Image pyramid
- Half width and height when going upwards



- 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

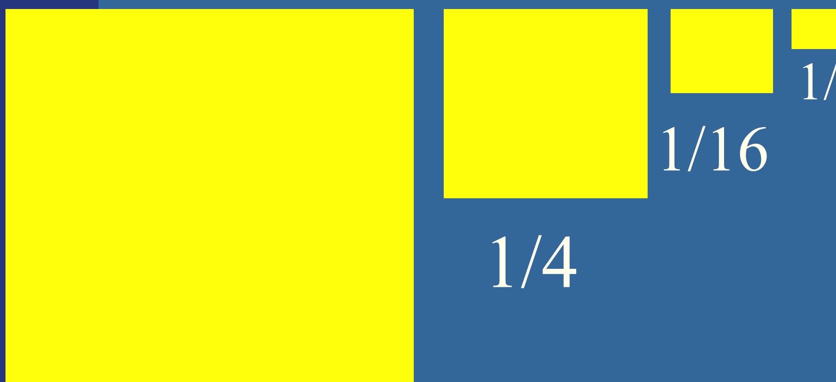
- Interpolate between those bilinear values
  - Gives trilinear interpolation



- Constant time filtering: 8 texel accesses

Mipmapping: Memory requirements

- Not twice the number of bytes…!

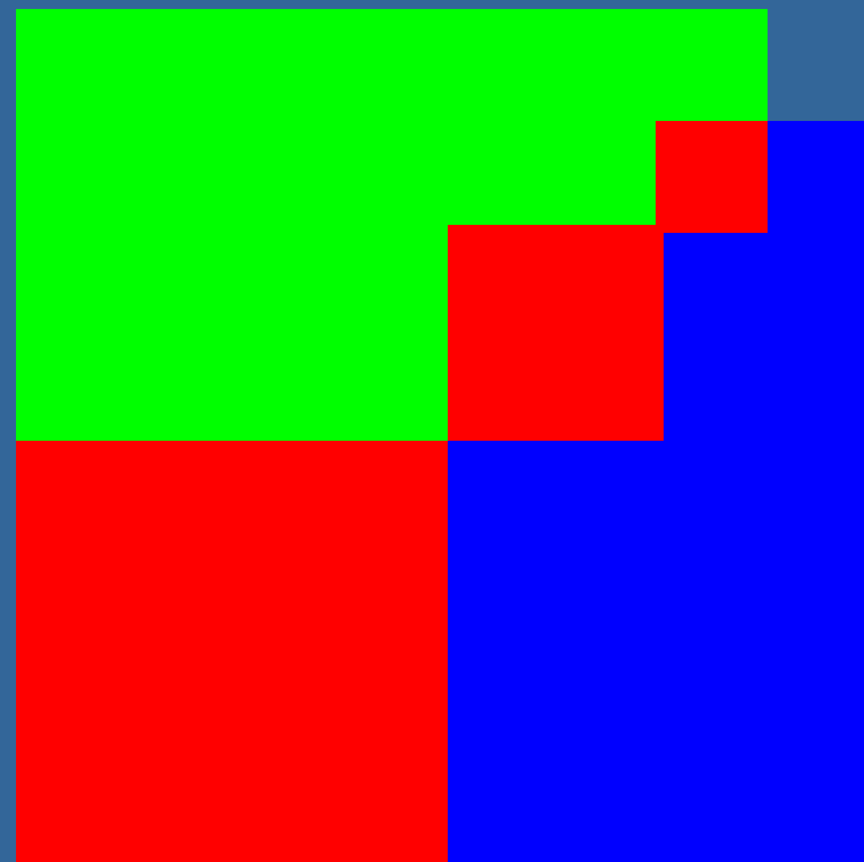


1/64

1/16

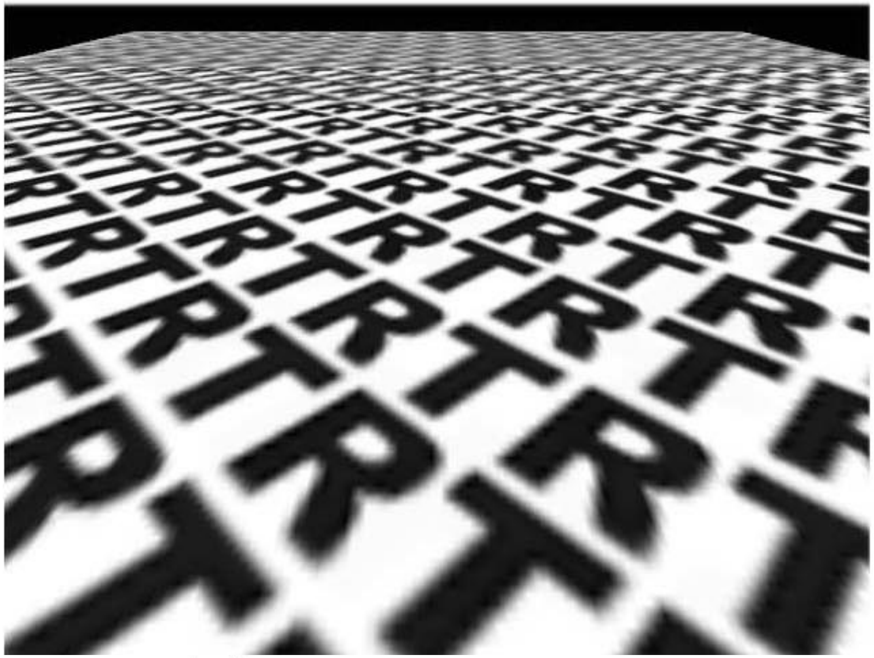
1/4

1/1

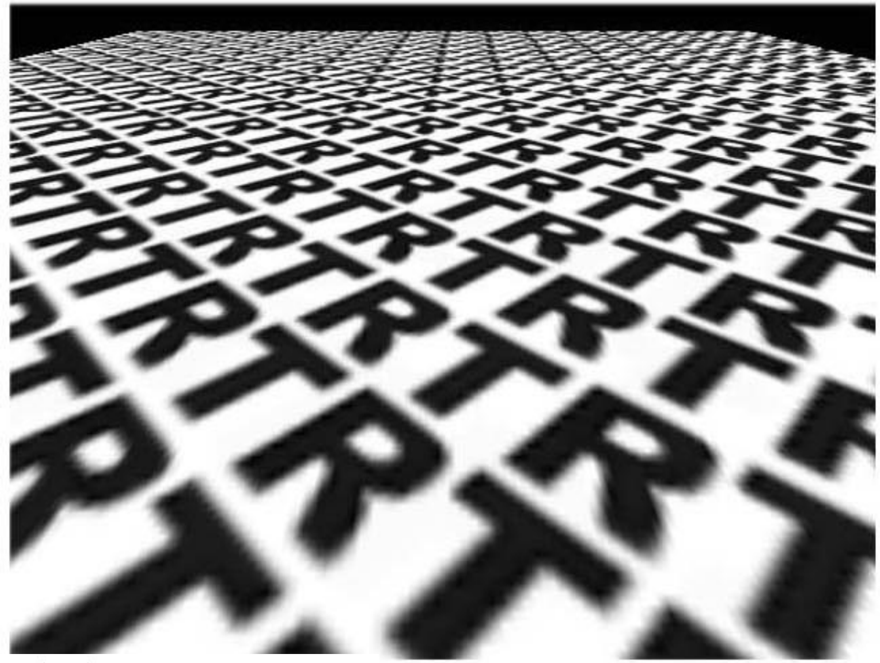


- Rather 33% more – not that much

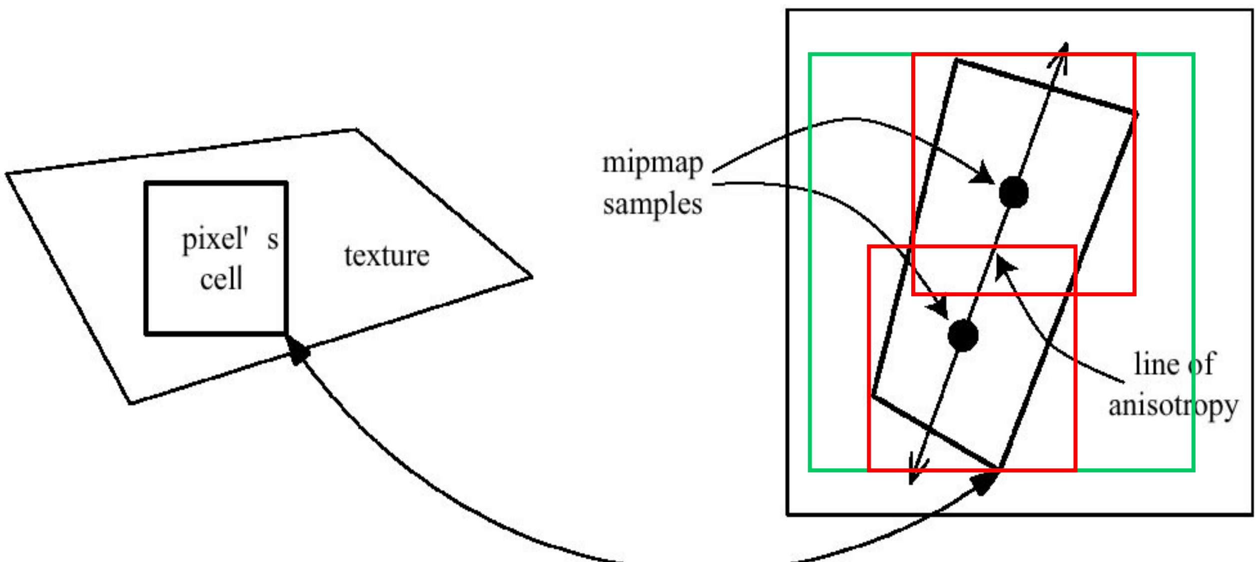
Anisotropic texture filtering



pixel space

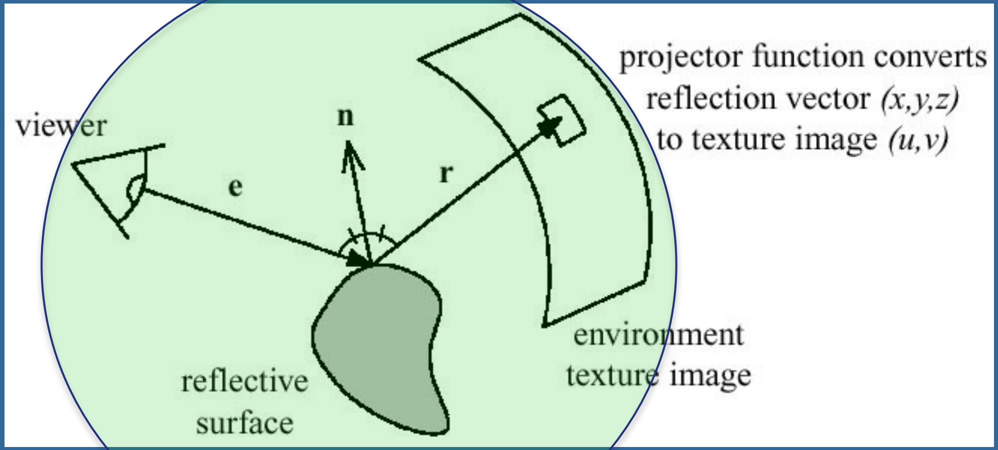


texture space



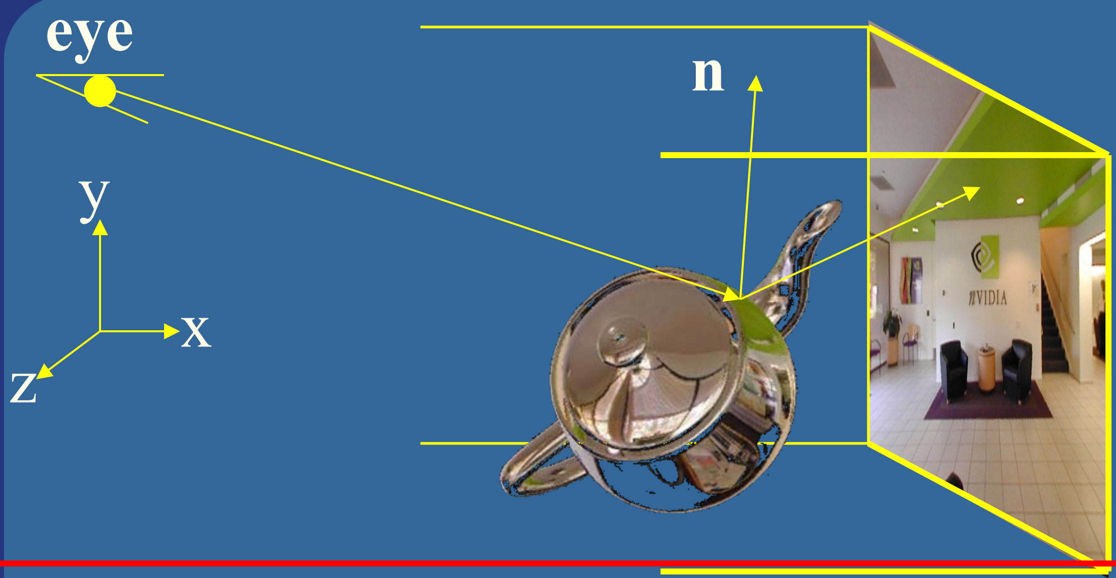
(See page 188-189)

Environment mapping



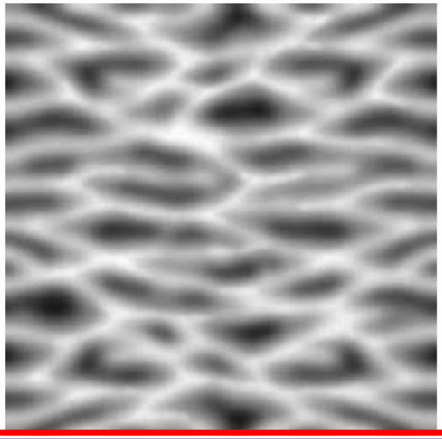
- Assumes the environment is infinitely far away
- Cube mapping is the norm nowadays

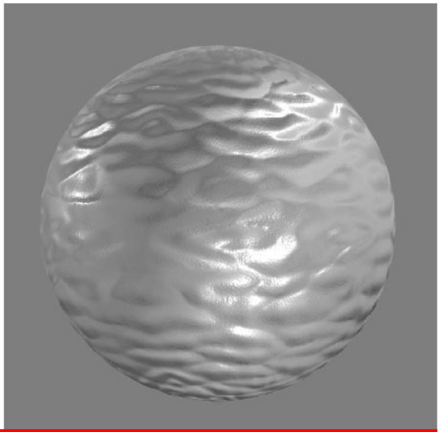
Cube mapping



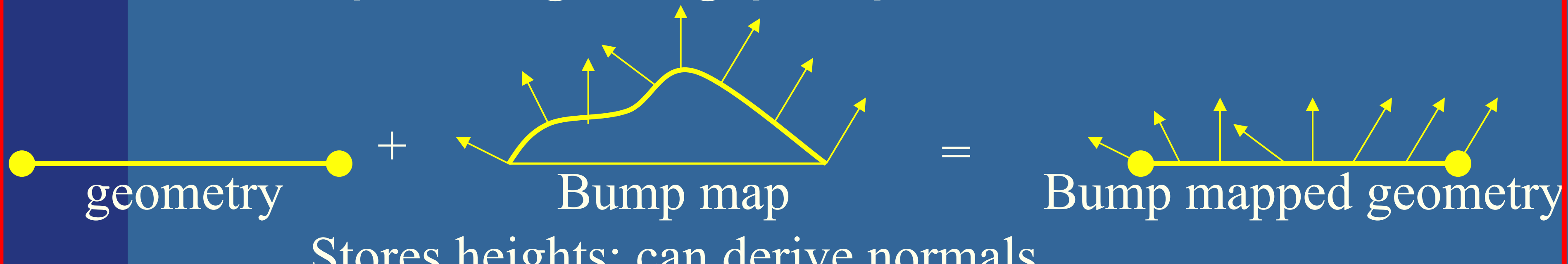
- Simple math: compute reflection vector, **r**
- Largest abs-value of component, determines which cube face.
  - Example: **r**=(5,-1,2) gives POS_X face
- Divide **r** by abs(5) gives (*u*,*v*)=(-1/5,2/5)
- Also remap from [-1,1] to [0,1] by (u,v) = ((u,v)+vec2(1,1))*0.5;
- Your hardware does all the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

Bump mapping

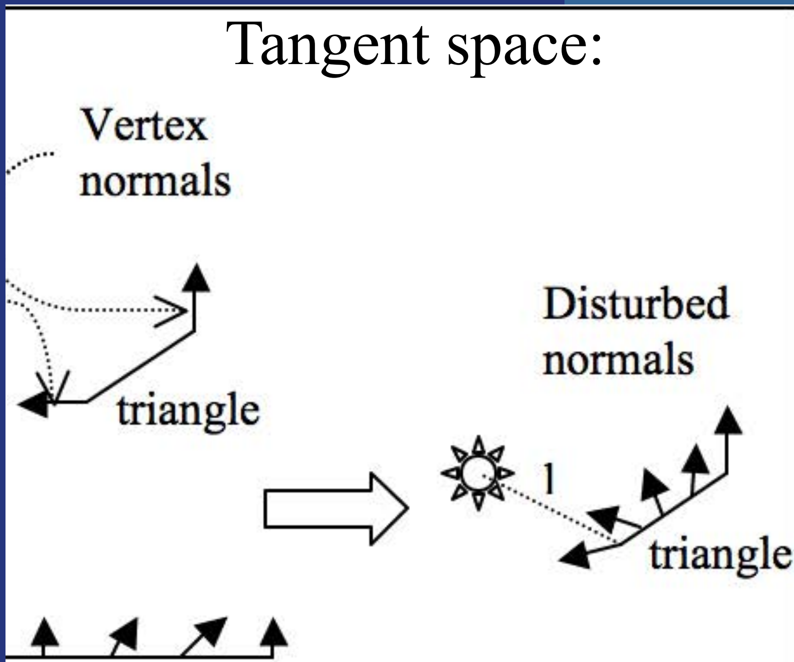


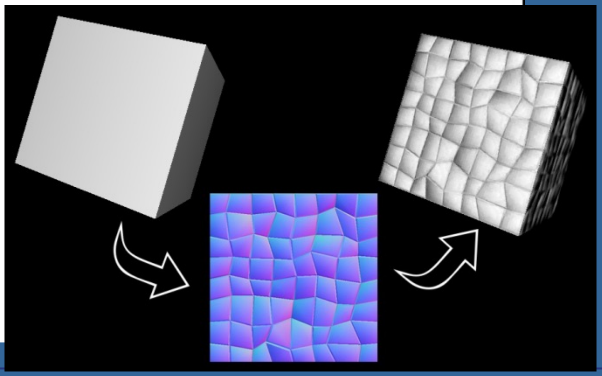


- by Blinn in 1978
- Inexpensive way of simulating wrinkles and bumps on geometry
  - Expensive to model these geometrically
- Instead <u>let a texture modify the normal at each pixel, and then use this normal to compute lighting per pixel</u>



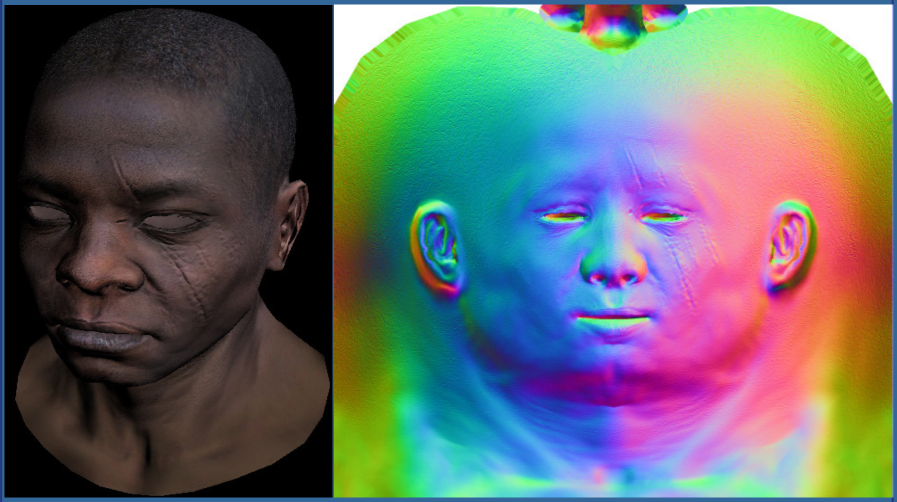
Normal mapping in tangent vs object space





Object space:

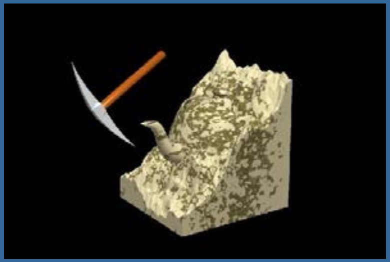
- Normals are stored directly in model space. I.e., as including both face orientation plus distorsion.

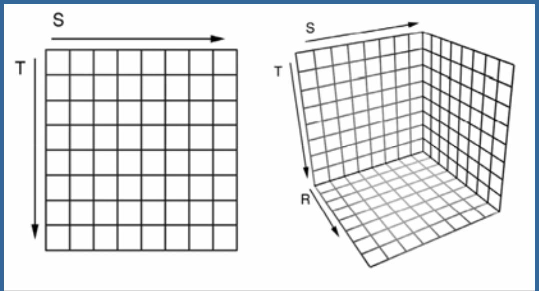


Tangent space:

- 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...





2D 3D

- 3D textures:
  - 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

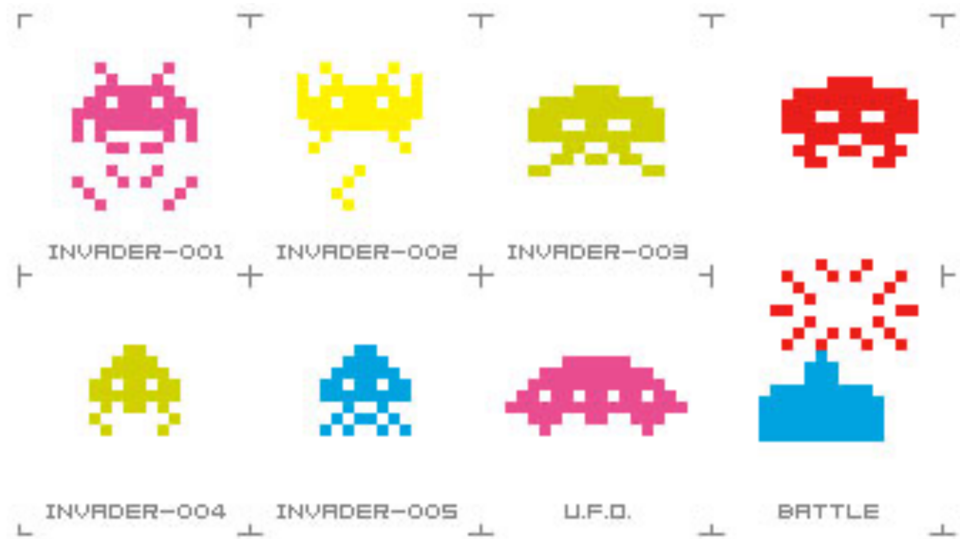
- Displacement Mapping
  - Like bump/normal maps but truly offsets the surface geometry (not just the lighting).
  - Gfx hardware cannot offset the fragment's position
    - Offsetting per vertex is easy in vertex shader but requires a highly tessellated surface.
    - Tesselation shaders are created 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.)

Sprites

Just know what “sprites” are and that they are very similar to a billboard

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

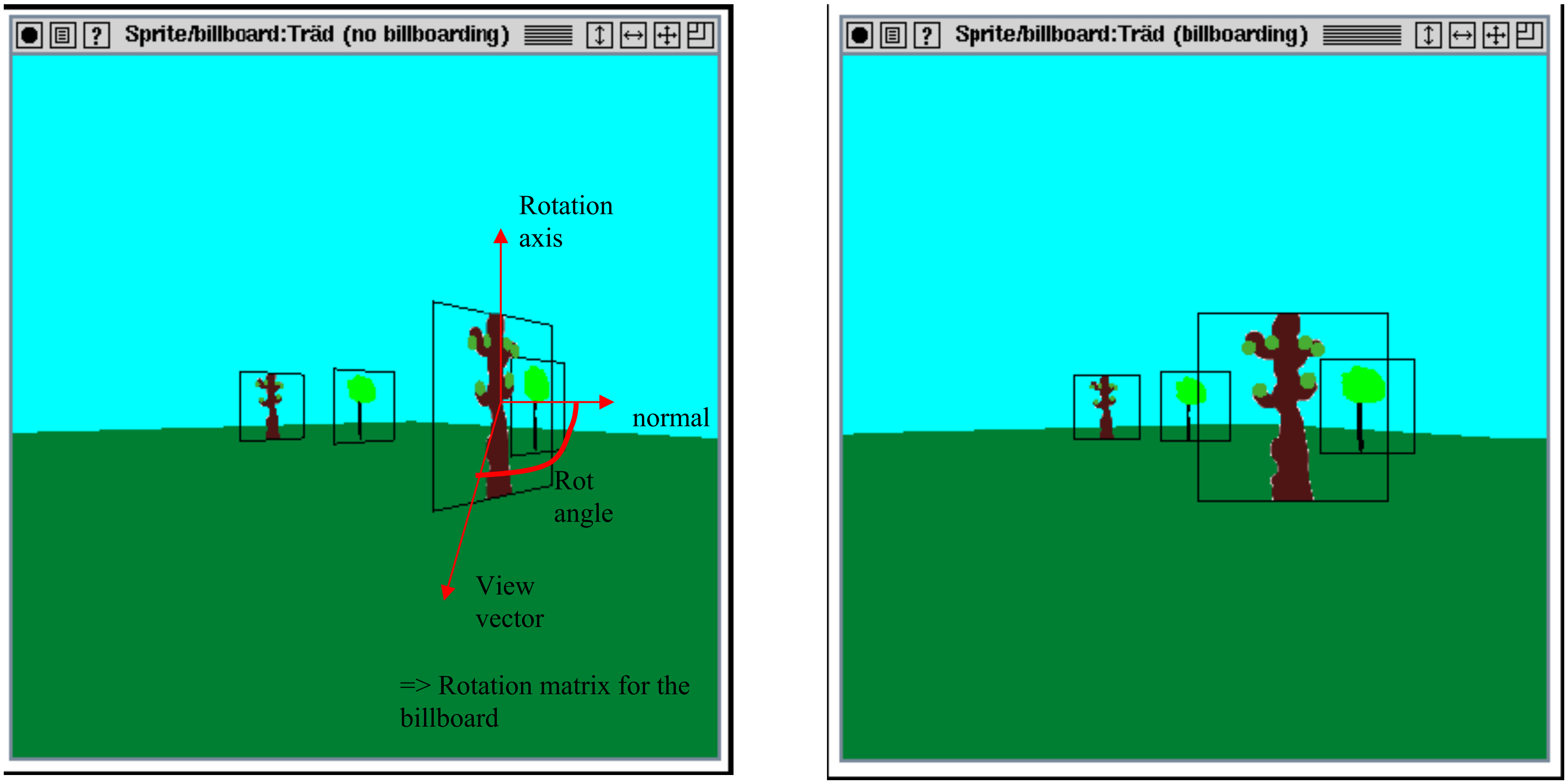
```
GLbyte M[64]=
{   127,0,0,127, 127,0,0,127,
    127,0,0,127, 127,0,0,127,
    0,127,0,0,   0,127,0,127,
    0,127,0,127, 0,127,0,0,
    0,0,127,0,   0,0,127,127,
    0,0,127,127, 0,0,127,0,
    127,127,0,0, 127,127,0,127,
    127,127,0,127, 127,127,0,0};

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);
    SDL_GL_SwapWindow //"Swap buffers"
}
```



Billboards

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

Billboards



- Rotate them towards viewer
  - Either by rotation matrix (see OH 288), or
  - by orthographic projection

Billboards

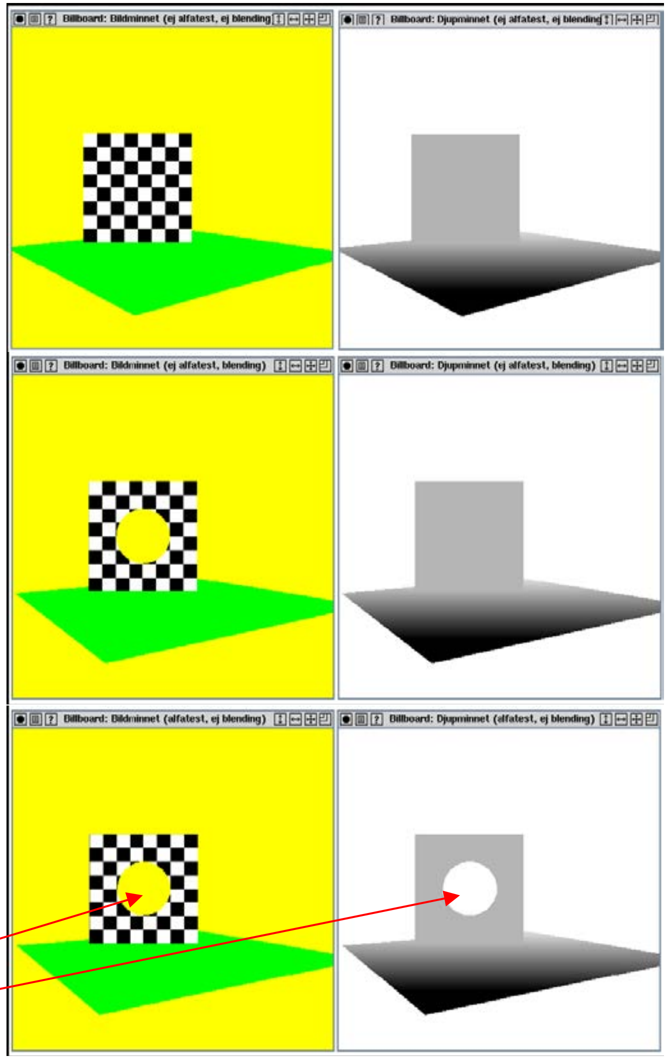
- Fix correct transparency by blending AND using alpha-test
  - In fragment shader:

    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.

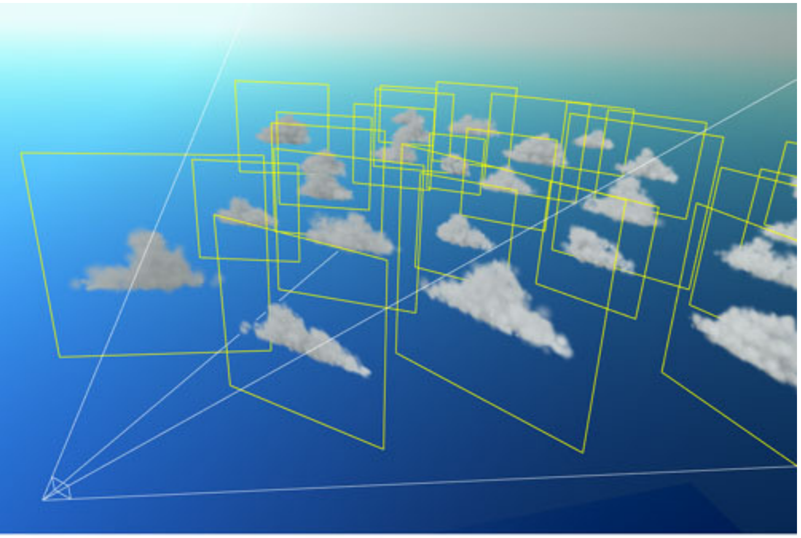
E.g. here: so that objects behind is visible through the hole

Color Buffer Depth Buffer

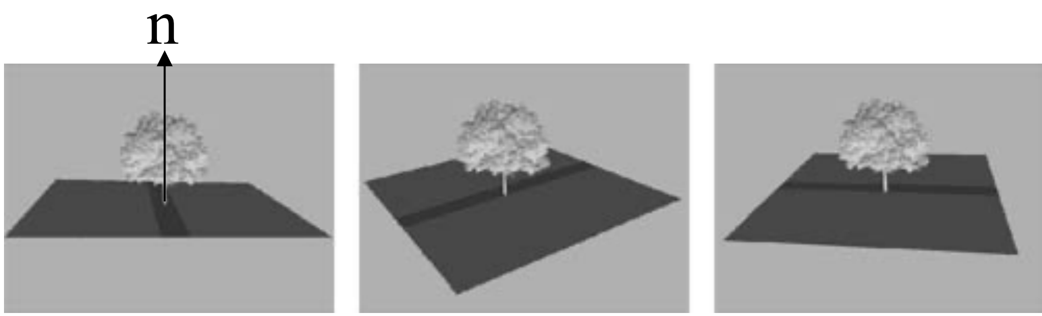


With blending

With alpha test



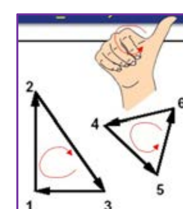
(Also called *Impostors)*



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)

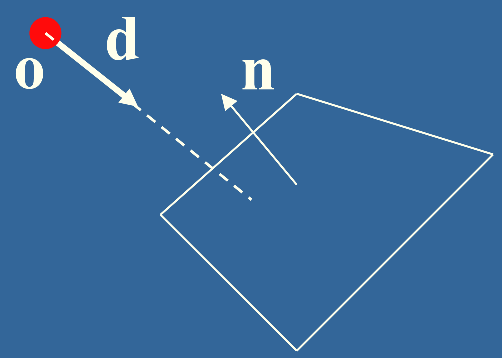
Frame buffers can consist of:

- 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

- Analytic test:
  - Be able to compute ray vs sphere or other similar formula
  - Ray/triangle, ray/plane
  - Point/plane, Sphere/plane,
  - Know expressions for ray, sphere, cylinder, plane, triangle
- Geometrical tests
  - Ray/box with slab-test
  - Ray/polygon (3D->2D)
  - box/plane
  - AABB/AABB
  - View frustum vs spheres/AABB:s/BVHs.
  - Separating Axis Theorem (SAT)
- Know what a dynamic test is

Analytical: Ray/plane intersection



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

- Replace **p** by $\mathbf{r}(t)$ and solve for t:

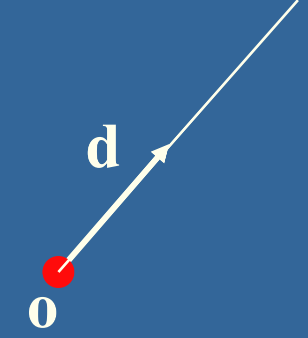
  $\mathbf{n}\cdot(\mathbf{o}+t\mathbf{d}) + d = 0$

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

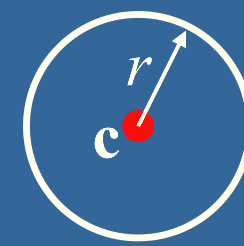
  $t = (-d - \mathbf{n}\cdot\mathbf{o}) / (\mathbf{n}\cdot\mathbf{d})$

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*



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

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

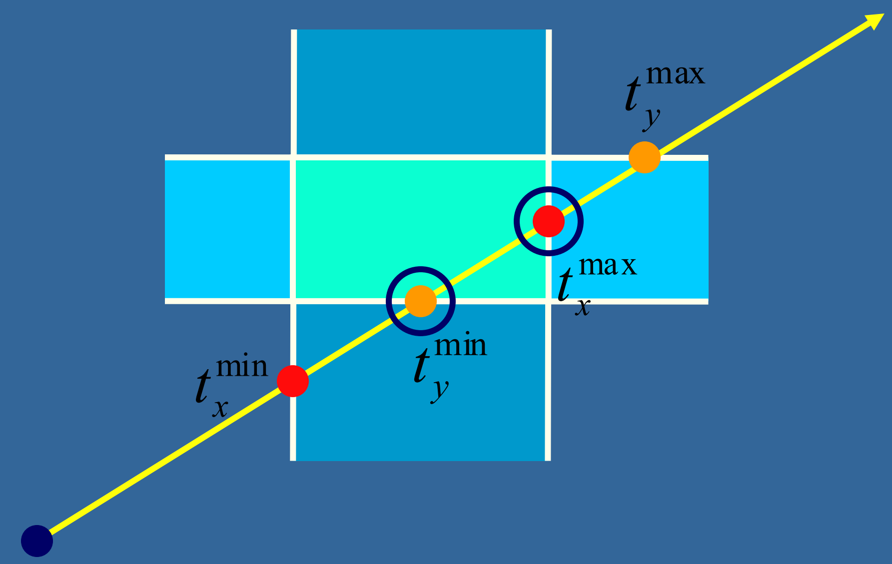
$$(\mathbf{d}\cdot\mathbf{d})t^2+2((\mathbf{o}-\mathbf{c})\cdot\mathbf{d})t+(\mathbf{o}-\mathbf{c})\cdot(\mathbf{o}-\mathbf{c})-r^2=0$$

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

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

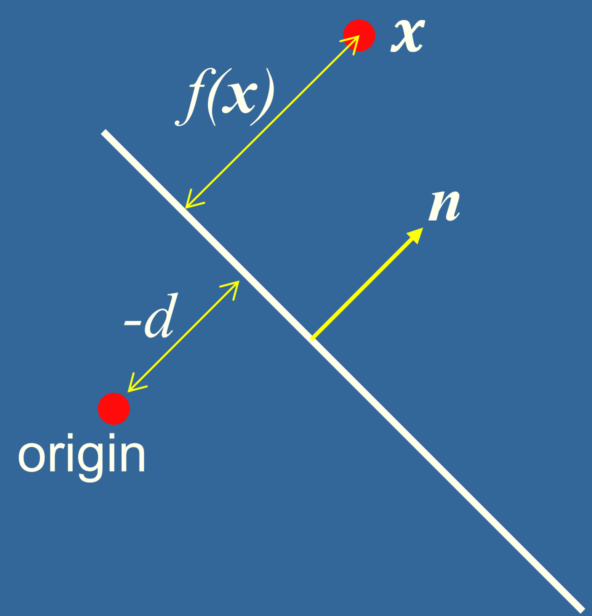
The Plane Equation

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

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.)

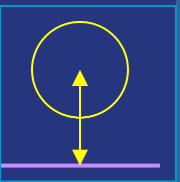
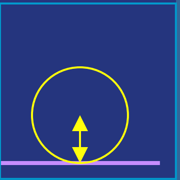
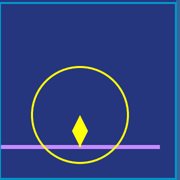
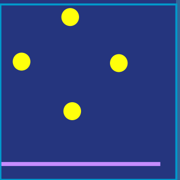
- $f(x)>0$ means above the plane
- $f(x)<0$ means below the plane



$-d$ is how far the origin is behind the plane

Sphere/Plane
Box/Plane

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

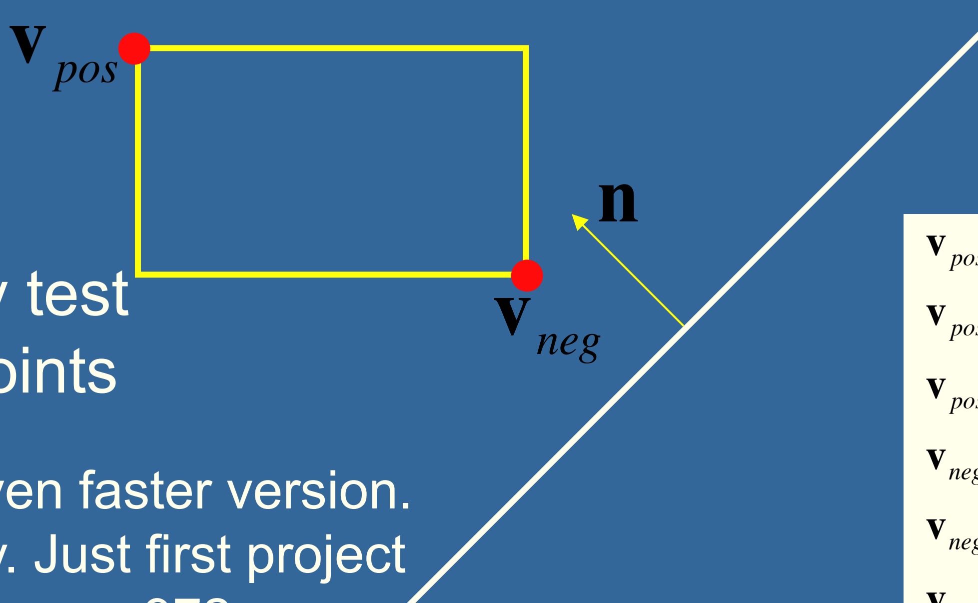





- Sphere: compute $\boxed{f(\mathbf{c}) = \mathbf{n}\cdot\mathbf{c} + d}$
- $f(\mathbf{c})$ is the signed distance ($\mathbf{n}$ normalized)
- $\text{abs}(f(\mathbf{c})) > \text{r}$ no collision
- $\text{abs}(f(\mathbf{c})) = \text{r}$ sphere touches the plane
- $\text{abs}(f(\mathbf{c})) < \text{r}$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

AABB/plane

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

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

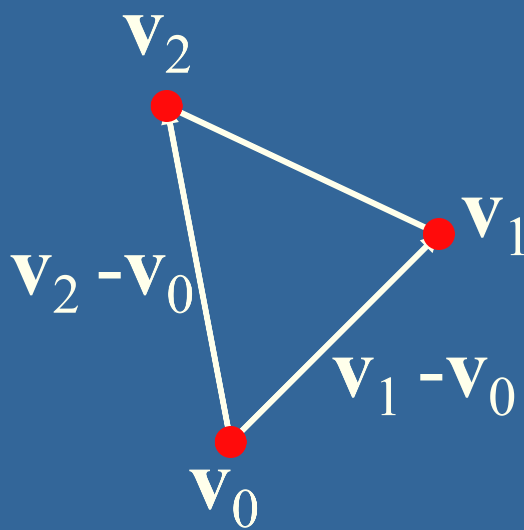


Need only test the red points

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

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:

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

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

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

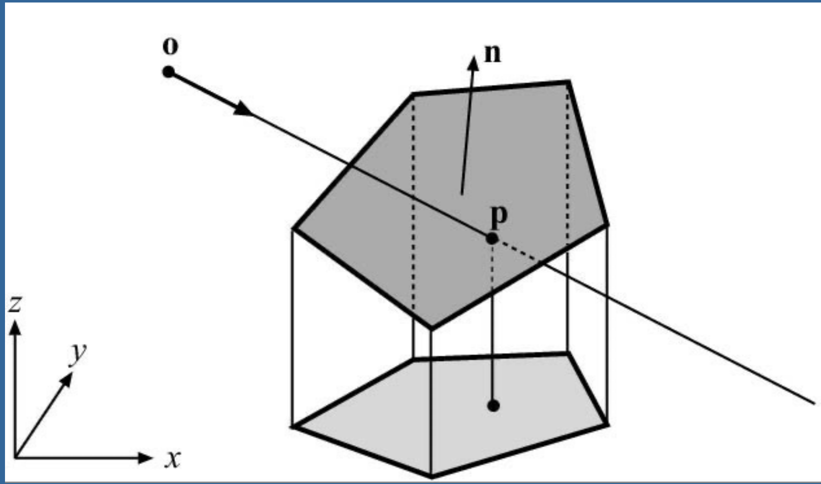
$$\begin{pmatrix} | & | & | \\ -\mathbf{d} & \mathbf{v}_1 - \mathbf{v}_0 & \mathbf{v}_2 - \mathbf{v}_0 \\ | & | & | \end{pmatrix} \begin{pmatrix} t \\ u \\ v \end{pmatrix} = \begin{pmatrix} | \\ \mathbf{o} - \mathbf{v}_0 \\ | \end{pmatrix}$$

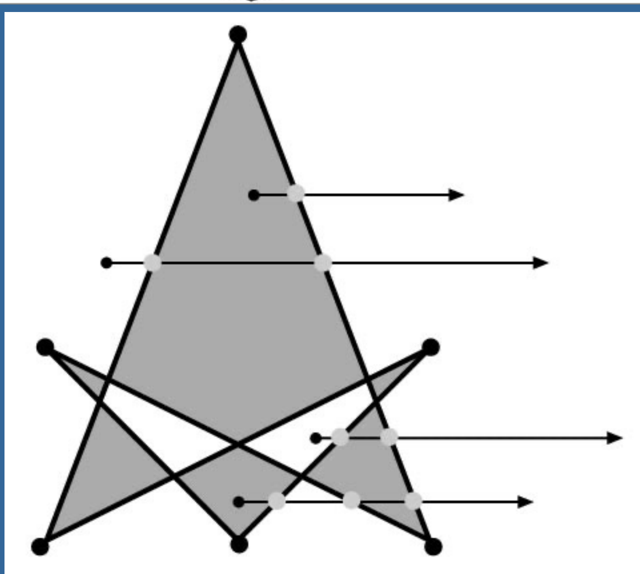
$\mathbf{Ax=b}$

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

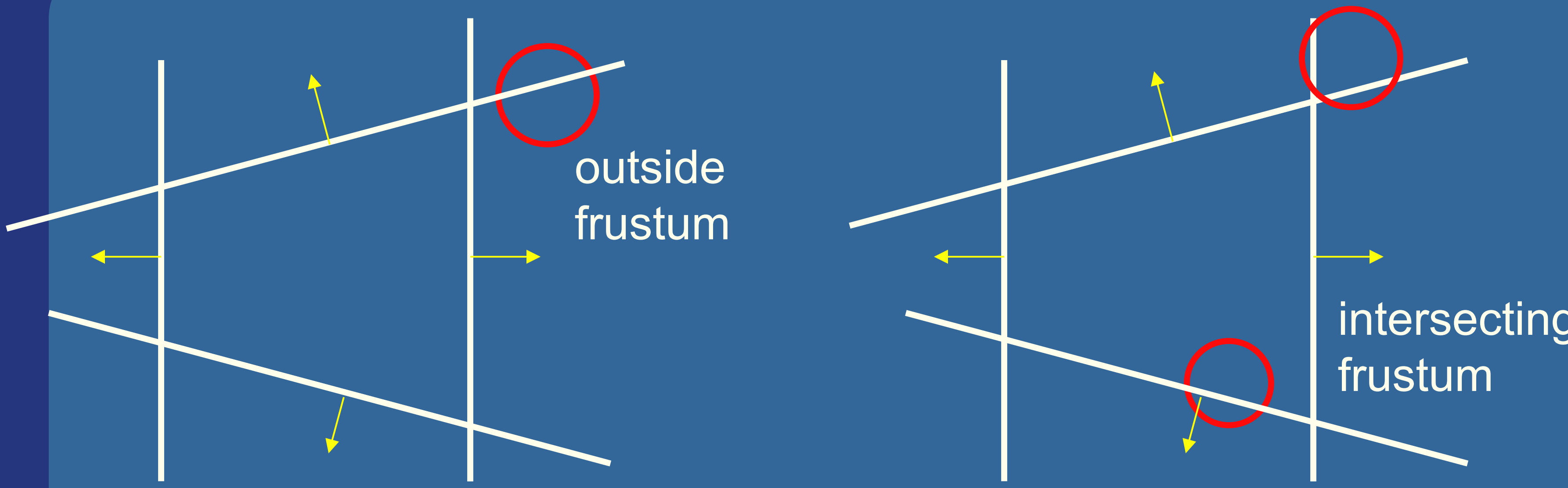
Ray/Polygon: very briefly

- 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



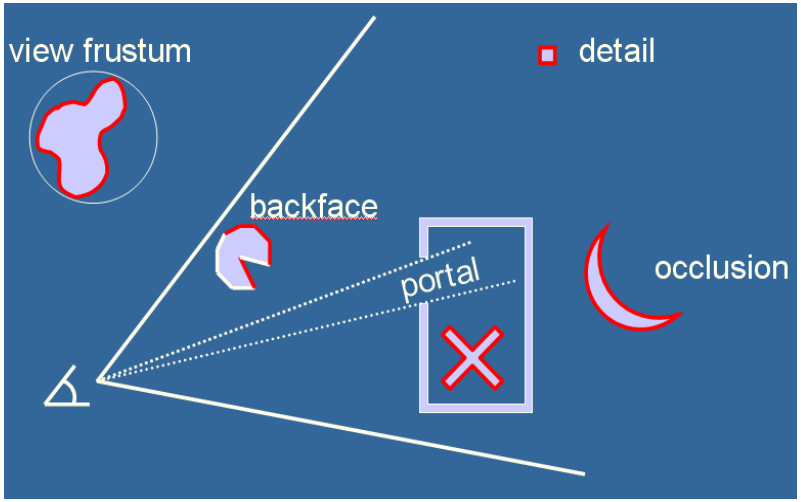


View frustum testing example



- 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

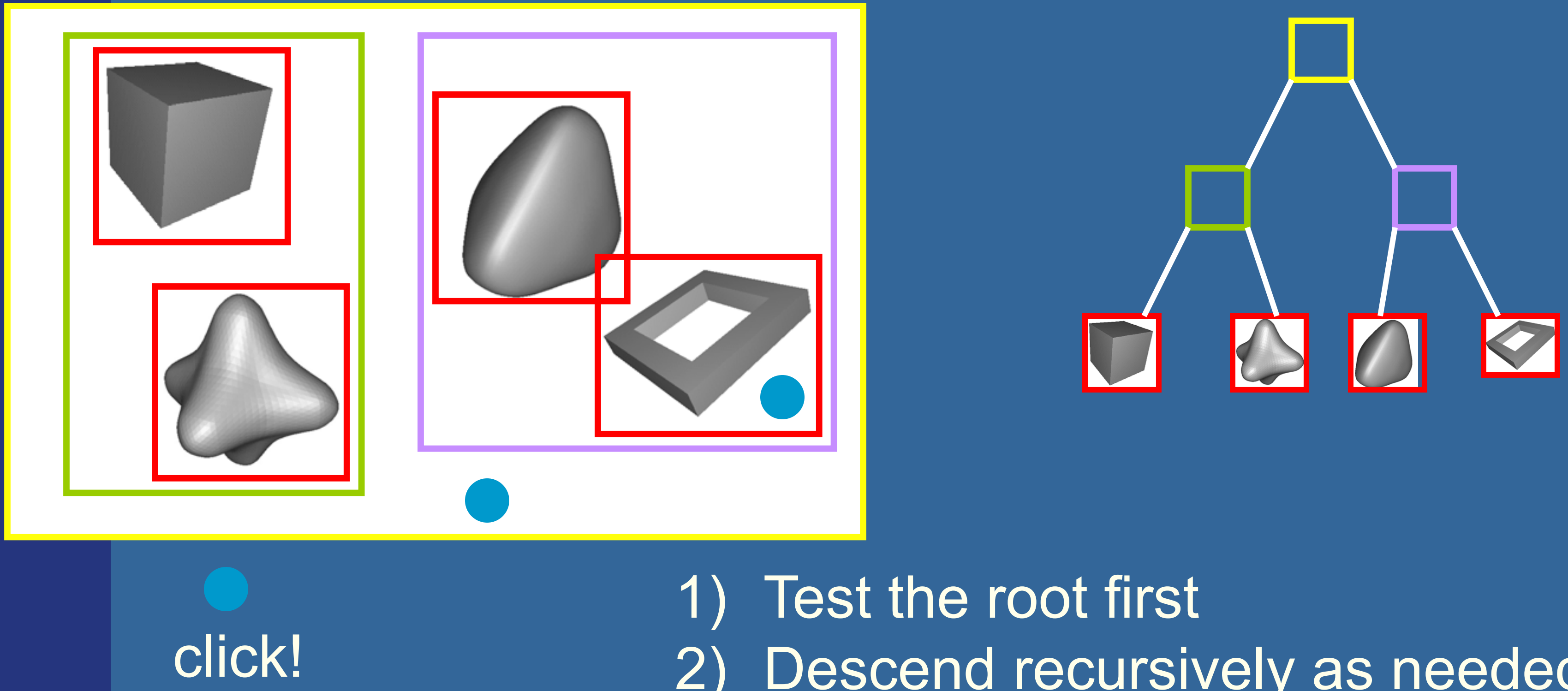
Lecture 7.1: Spatial Data Structures and Speed-Up Techniques



- 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), quadtree/octree

Axis Aligned Bounding Box Hierarchy - an example

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



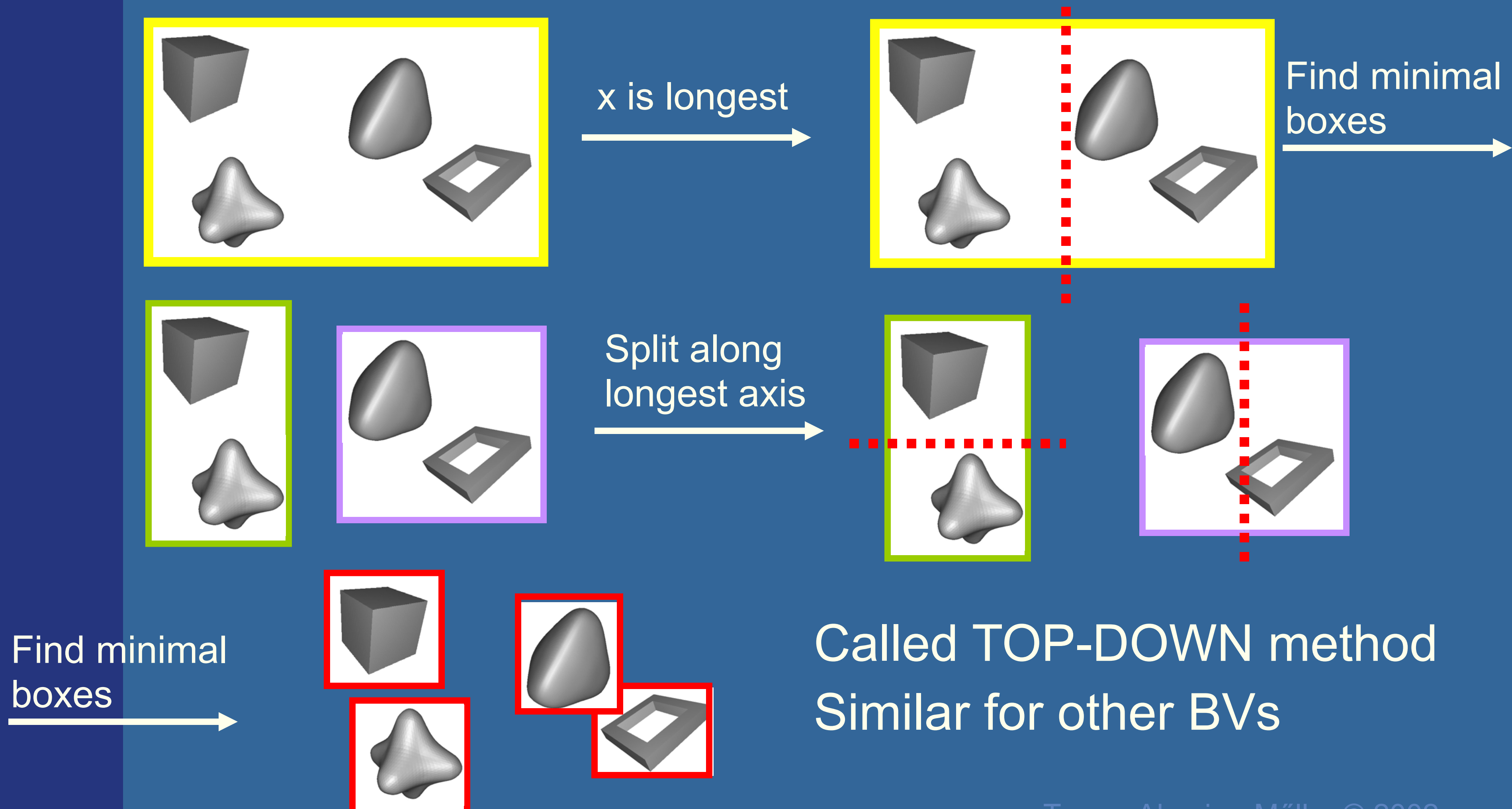
1) Test the root first
2) Descend recursively as needed
3) Terminate traversal when possible

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

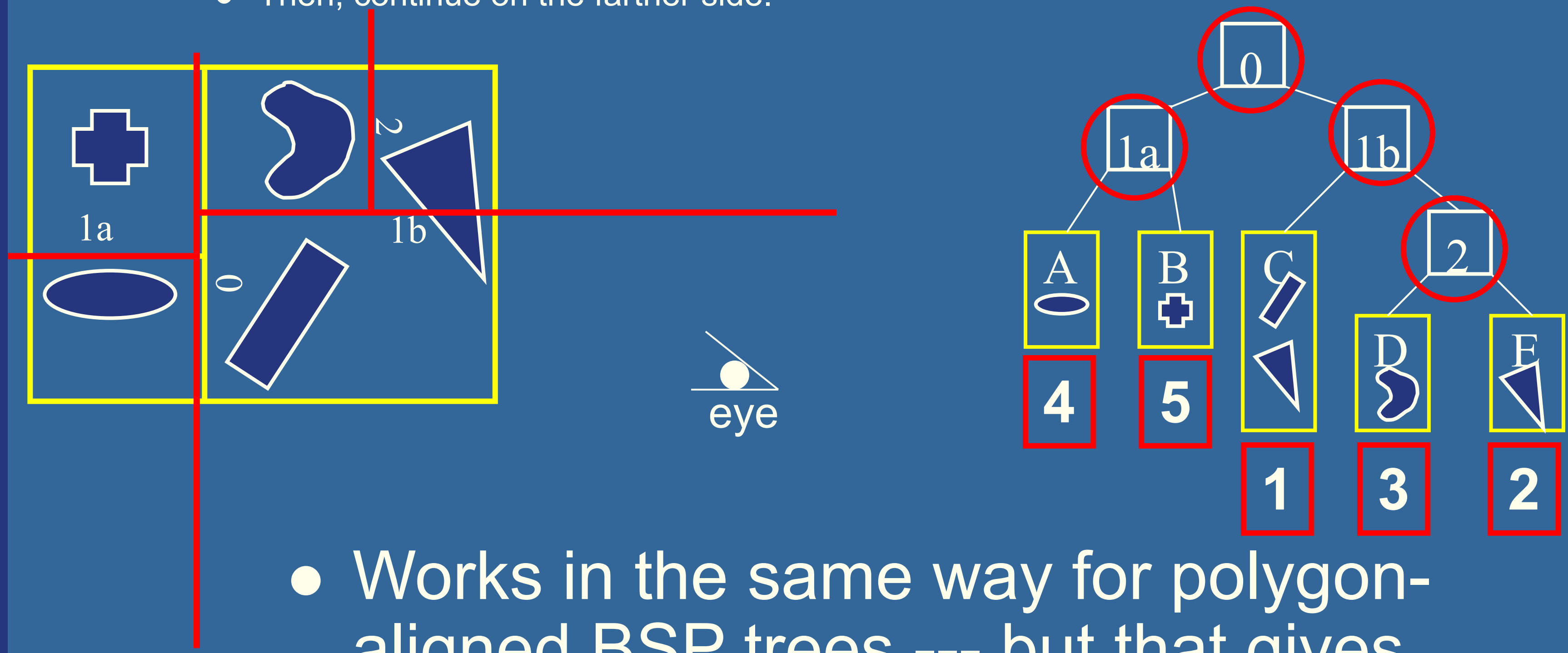


Called TOP-DOWN method

Similar for other BVs

Axis-aligned BSP tree
Rough sorting

- Test the planes, 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.

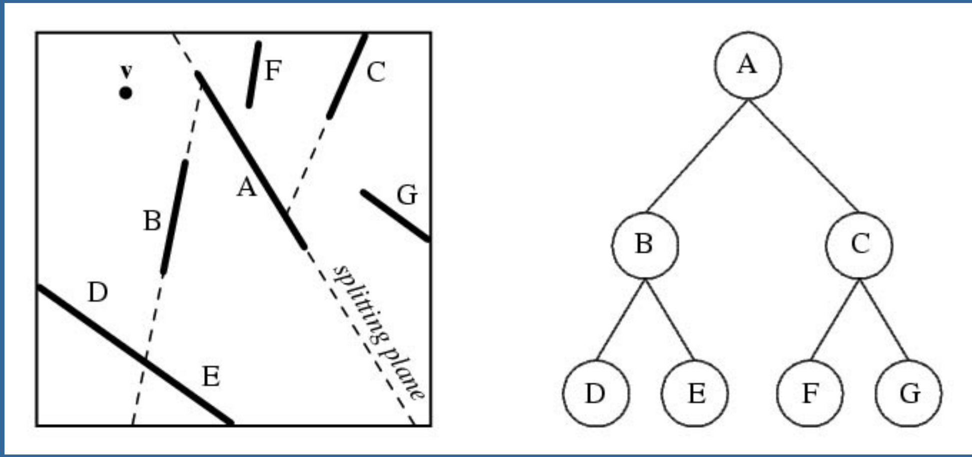


- Works in the same way for polygon-aligned BSP trees --- but that gives exact sorting

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.
```

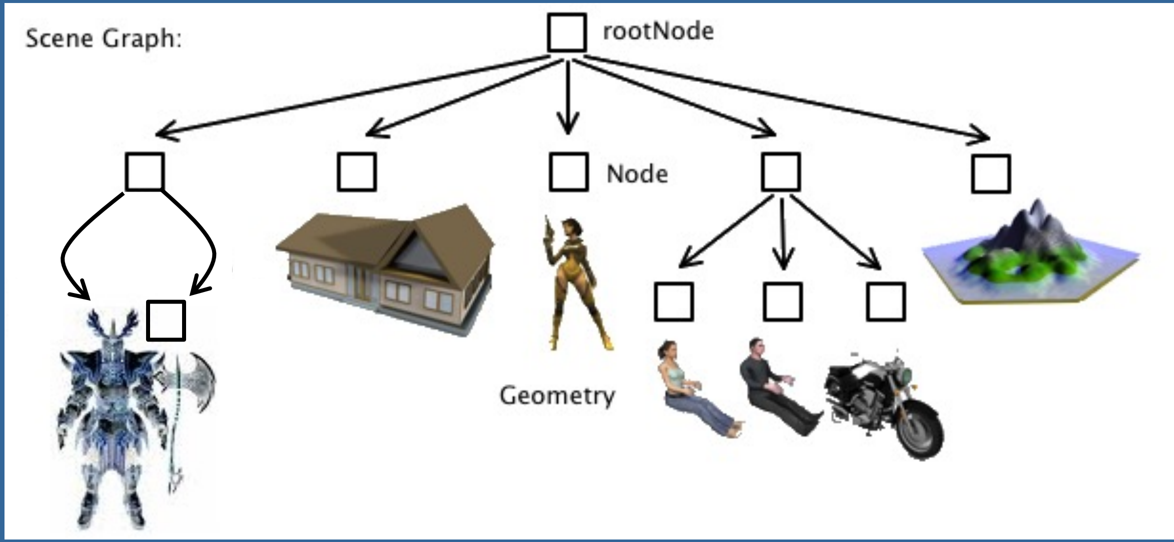


Know how to build it and how to traverse back-to-front or front-to-back with respect to the eye position (here: **v**)

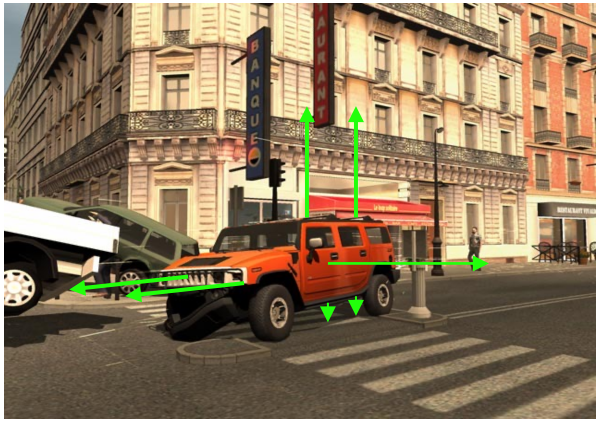
A Scene Graph is a hierarchical scene description – more typically a **logical** hierarchy (than e.g. **spatial**)

Scene graphs
– a node hierarchy

- A scene graph is a node hierarchy, which often reflects a logical hierarchical scene description
  - often in combination with a BVH such that each node has a BV.
- Common hierarchical features include:
  - Lights
  - Materials
  - Transforms
  - Transparency
  - Selection



Lecture 7.2: Collision Detection



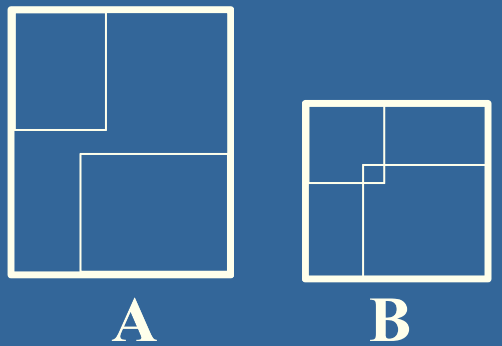
- 3 types of algorithms:
  - With rays
    - Fast but not exact
  - With BVH
    - Slower but exact
    - You should be able to write pseudo code for BVH/BVH test for coll det between two objects.
  - For many many objects.
    - Course pruning of ”obviously” non-colliding objects
    - E.g., Use a grid 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.
    - (Sweep-and-prune is interesting but you can skip it.)

Pseudo code for BVH against BVH

```
FindFirstHitCD(A, B)
if(not overlap(A, B)) return false;
if(isLeaf(A) and isLeaf(B))
   for each triangle pair T_A ∈ A_c and T_B ∈ B_c
      if(overlap(T_A, T_B)) return TRUE;
else if(isNotLeaf(A) and isNotLeaf(B))
   if(Volume(A) > Volume(B))
      for each child C_A ∈ A_c
       if FindFirstHitCD(C_A, B) return true;
   else
      for each child C_B ∈ B_c
       if FindFirstHitCD(A, C_B) return true;
else if(isLeaf(A) and isNotLeaf(B))
   for each child C_B ∈ B_c
    if FindFirstHitCD(C_B, A) return true;
else
   for each child C_A ∈ A_c
    if FindFirstHitCD(C_A, B) return true;
return FALSE;
```

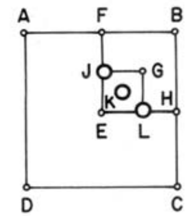
Pseudocode deals with 4 cases:

1) Leaf against leaf node
2) Internal node against internal node
3) Internal against leaf
4) Leaf against internal

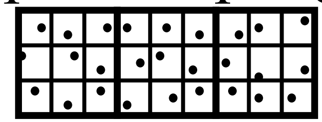


Lecture 8+9: Ray tracing

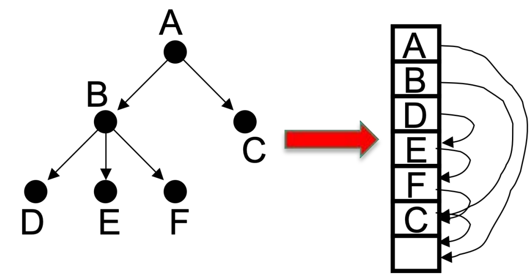
- Compute reflection ray
- Adaptive Super Sampling scheme:



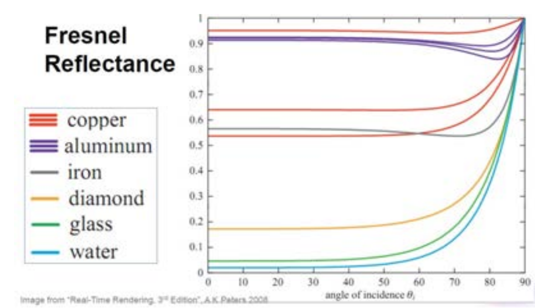
- Jittering:



- How to stop ray tracing recursion? Send weight…
- Spatial data structures - super important:
  – **Draw**: BVH: AABB/OBB/sphere. BSP-trees: polygon-aligned + AABSP=kd-tree. Octree/quadtree. Grids, hierarchical/recursive grids.
- Speedup techniques



  - Optimizations for BVHs: skippointer tree
  - Ray BVH-traversal
  - Grids: mailboxing – purpose and how it works.
  - (You do not need to learn the <u>**ray**</u> **traversal** algorithms for Grids nor AA-BSP trees)
  - Shadow cache
- Material: Metall: rgb-dependent Fresnel effect
  Dielectrics: not rgb-dependent.



- Constructive Solid Geometry – how to implement

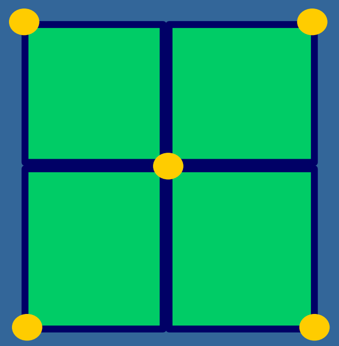
Adaptive Supersampling

Pseudo code:

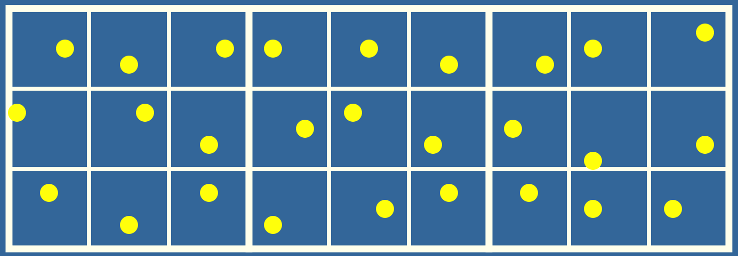
Color AdaptiveSuperSampling() {

- Make sure all 5 samples exist
  - (Shoot new rays along diagonal if necessary)
- Color col = black;
- For each quad i
  - If the colors of the 2 samples are fairly similar
    - col += (1/4)*(average of the two colors)
  - Else
    - col +=(1/4)* adaptiveSuperSampling(quad[i])
- return col;

}

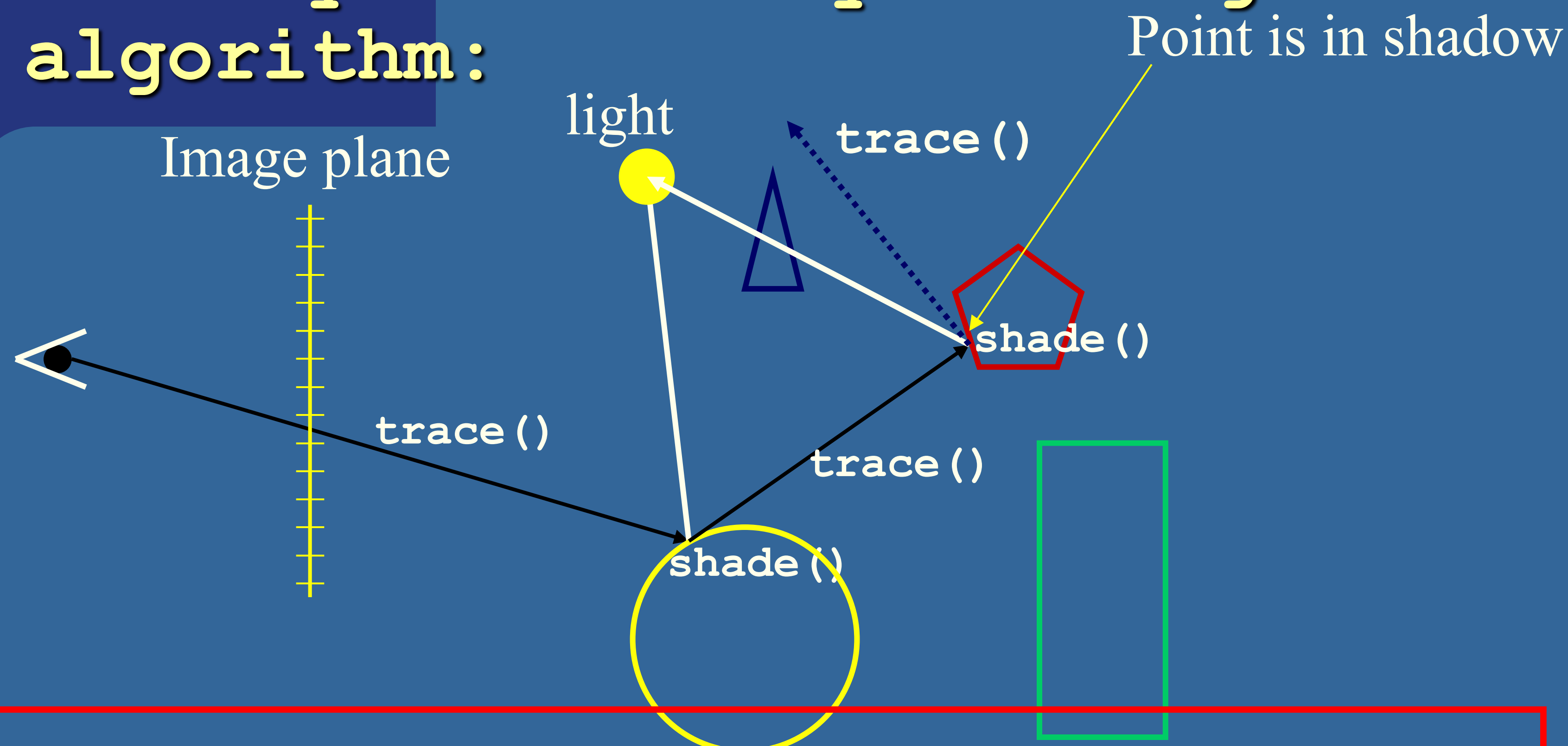


Jittered sampling



- Works as before
  - Replaces aliasing with noise
  - Our visual system likes that better
- This is often a preferred solution
- Can use adaptive strategies as well

Summary of the Ray tracing-algorithm:

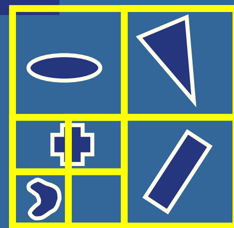


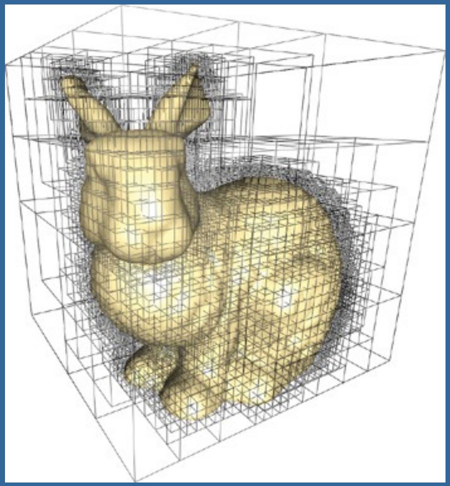
- **main()-calls trace() for each pixel**
- **trace(): should return color of closest hit point along ray.**
  1. **calls findClosestIntersection()**
  2. **If any object intersected → call shade().**
- **Shade(): should compute color at hit point**
  1. **For each light source, shoot shadow ray to determine 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.**

One of the most important slides in the whole course:

Data structures

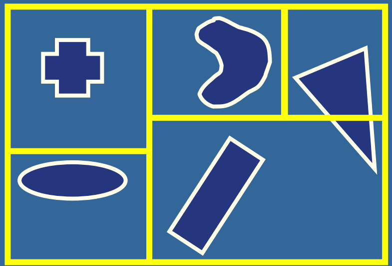
- Octree



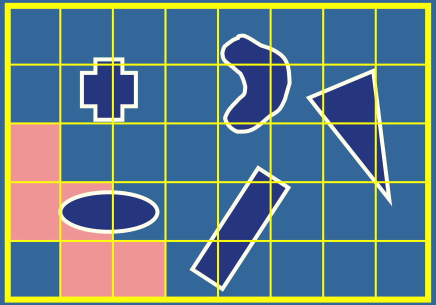


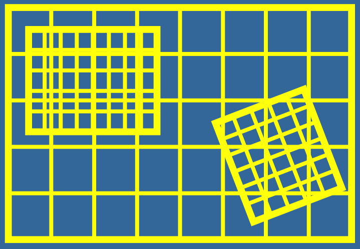
Kd-tree = Axis-Aligned BSP tree with fixed recursive split plane order (e.g. x,y,z,x,y,z…)

- Kd tree

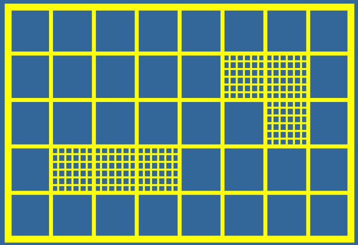


- Grids
  Including mail boxing



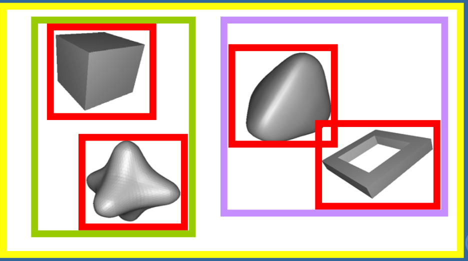


Hierarchical grid



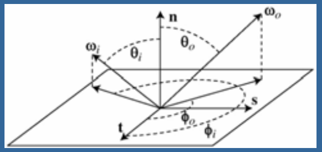
Recursive grid

- Bounding box hierarchies



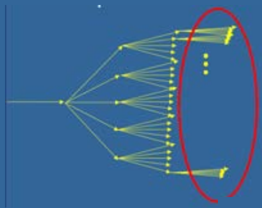
Lecture 10 – Global Illumination

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

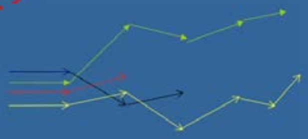


- Be able to explain all its components

– Monte Carlo sampling:



- The naïve way (an exponentially growing ray tree)
- Path tracing
  - – Why it is good, compared to naive monte-carlo sampling
  - – The overall algorithm (on a high level as in these slides).



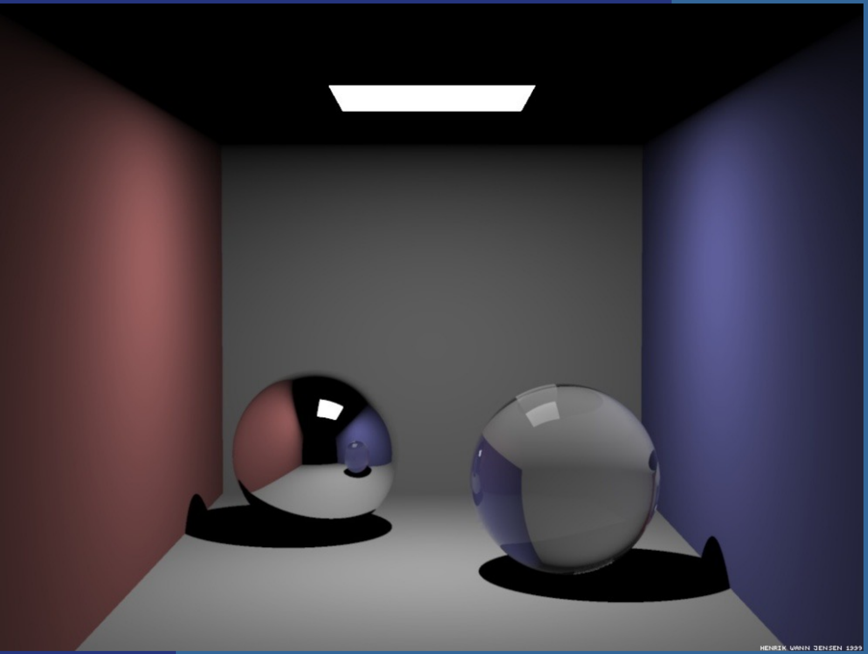
- Photon Mapping:
  1. Shoot photons from light source, and let them bounce around in the scene, and store them where they land (e.g. in a kD-tree).
  2. Ray-tracing pass from the eye. Estimate photon density at each ray hit, by growing a sphere (at the hit point in the kD-tree) until it contains a predetermined #photons. Sphere radius is then the inverse measure of the light intensity at the point.
- Bidirectional Path Tracing, Metropolis Light Transport
  - – Just their names. Don't need to know the algorithms.

– Denoising by Final Gather or AI

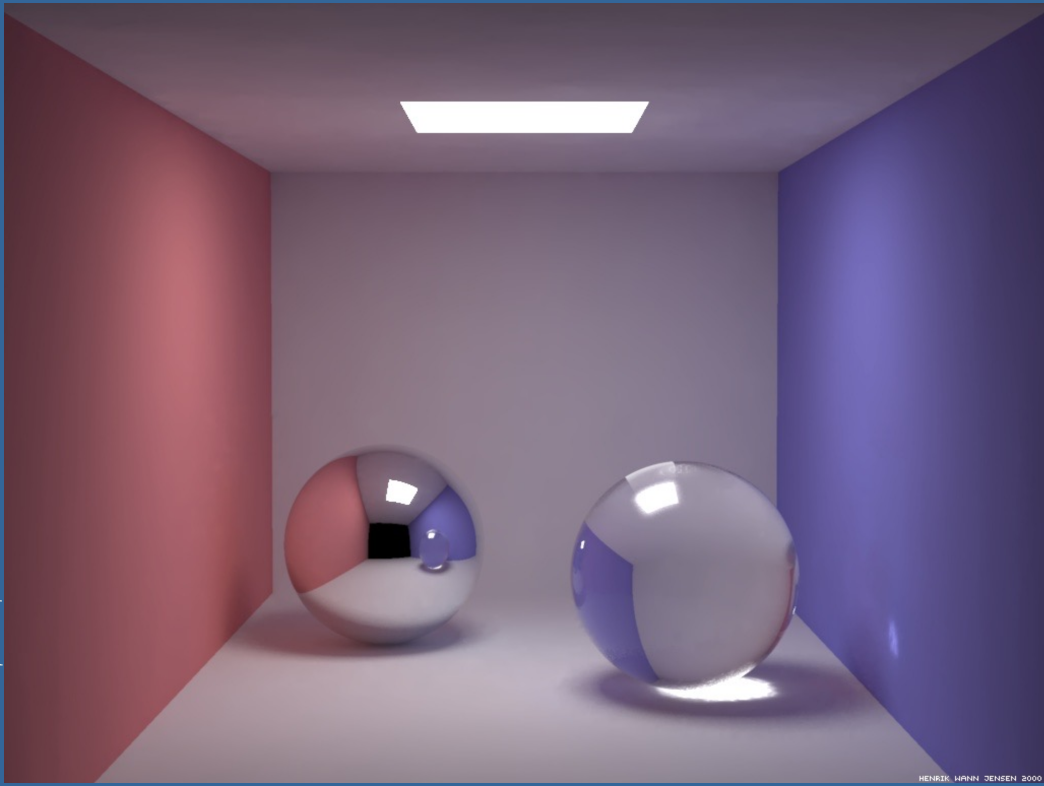
- – Final Gather – sample indirect illumination carefully at some positions in the world (final-gather points). At each ray hit, estimate indirect illumination by interpolation from nearby final-gather points.
- – AI: use some existing Deep Neural Network solution that denoises your images for your kind of scenes.

Isn't ray tracing enough?

Effects to note in Global Illumination image:
1) Indirect lighting (light reaches the roof)
2) Soft shadows (light source has area)
3) Color bleeding (example: roof is red near red wall) (same as 1)
4) Caustics (concentration of refracted light through glass ball)
5) Materials have no ambient component



Ray tracing



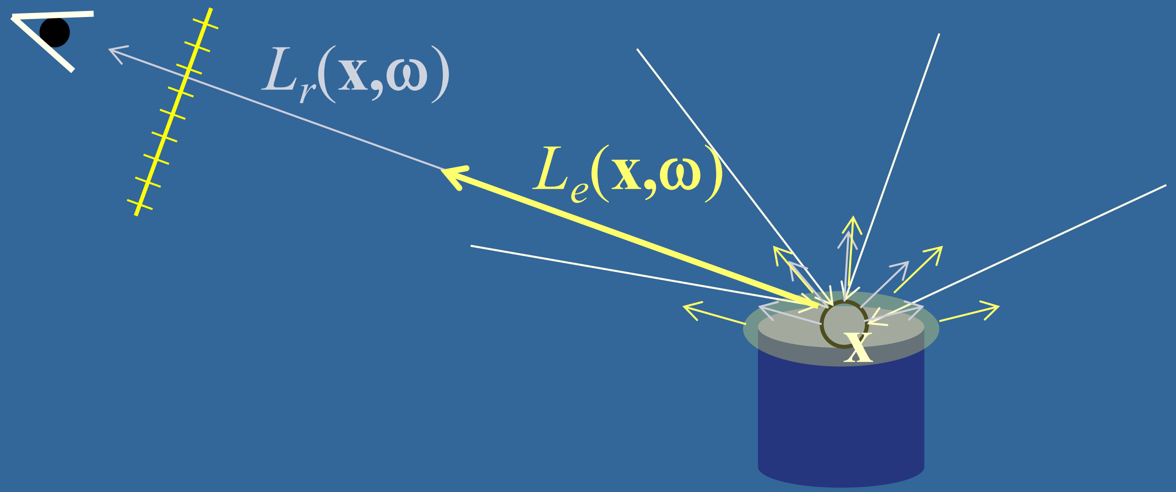
Global Illumination

Which are the differences?

Images courtesy of Henrik Wann Jensen

The rendering equation

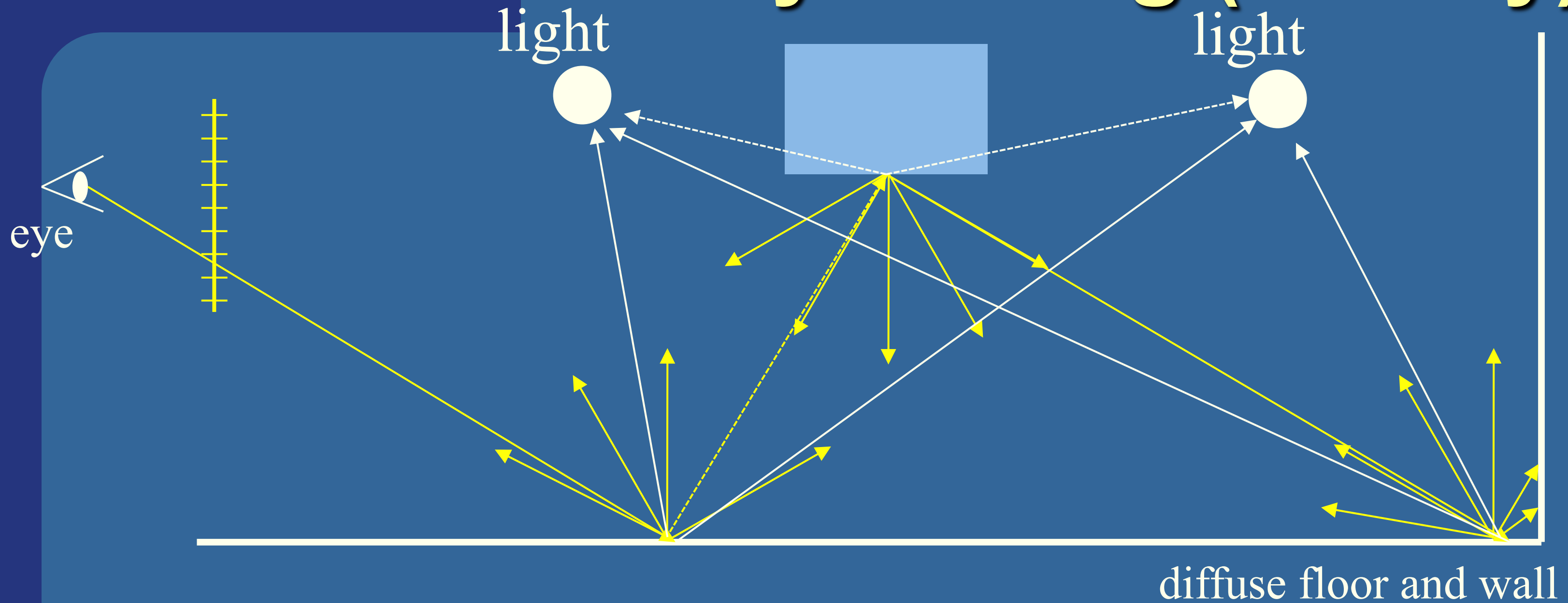
- Paper by Kajiya, 1986.
- Is the basis for all global illumination algorithms
- $L_o(\mathbf{x},\omega)=L_e(\mathbf{x},\omega)+L_r(\mathbf{x},\omega)$
  - outgoing=emitted+reflected radiance



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

- f_r is the BRDF, ω' is incoming direction, $\mathbf{n}$ is normal at point $\mathbf{x}$, Ω is hemisphere "around" $\mathbf{x}$ and $\mathbf{n}$, L_i is incoming radiance

Monte Carlo Ray Tracing (naïvely)

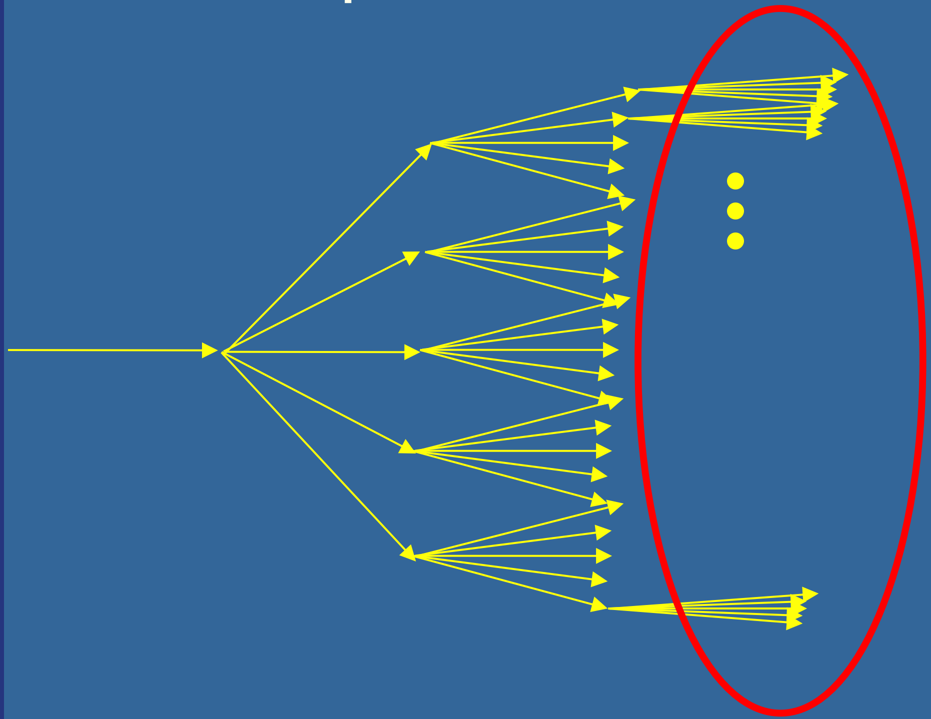


- (Compute local lighting as usual, with a shadow ray per light.)
- Sample indirect illumination by shooting sample rays over the hemisphere, at each hit.

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

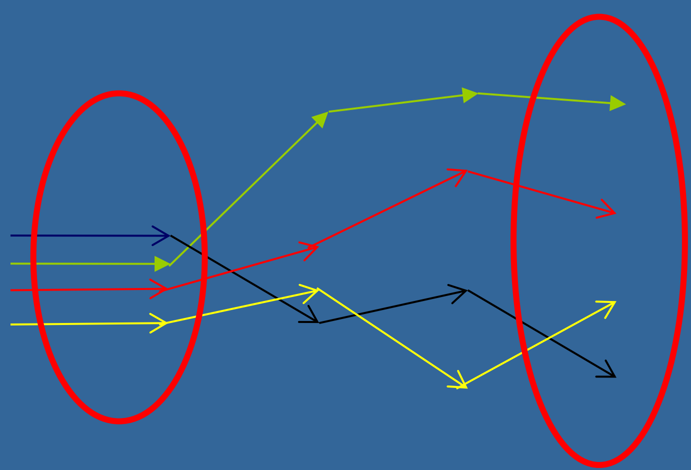
Monte Carlo Ray Tracing (naïvely)

- The indirect-illumination sampling 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.



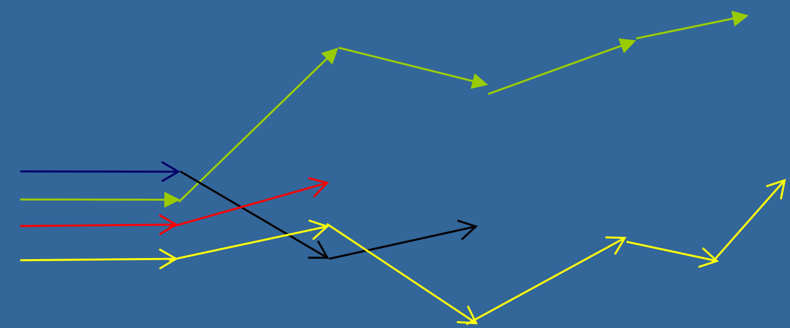
PathTracing
– one efficient Monte-Carlo Ray-Tracing solution

- 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.



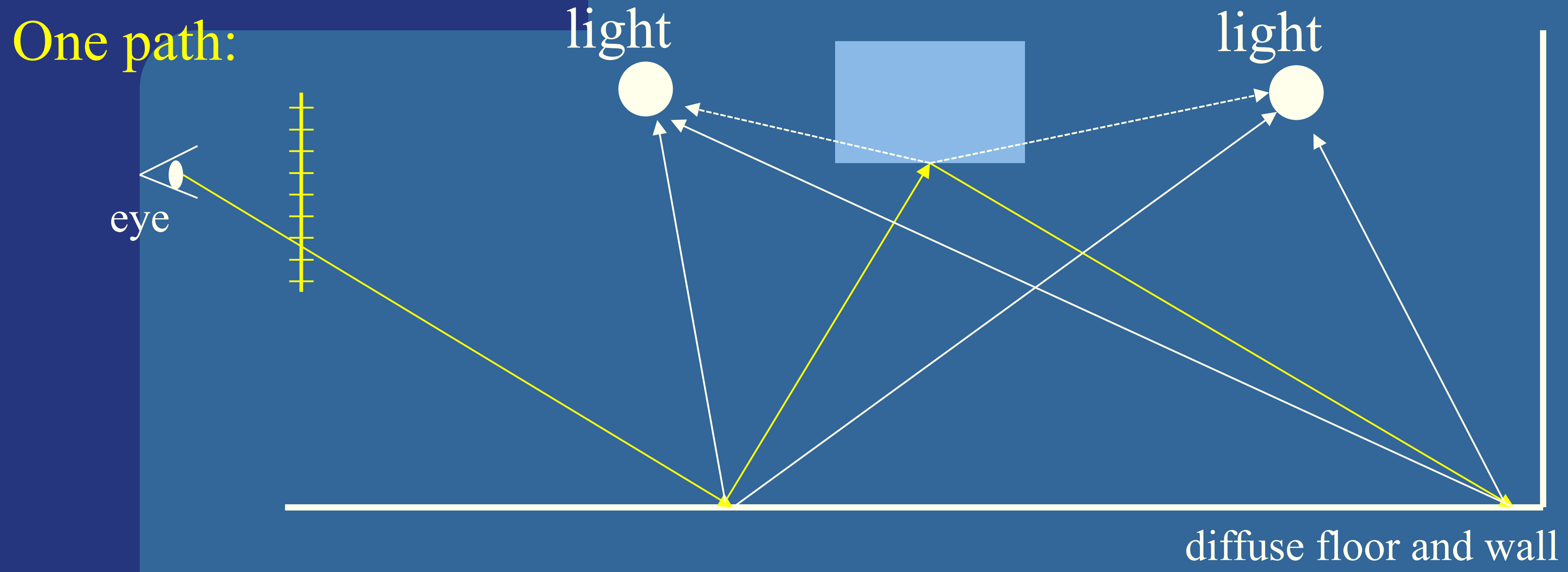
Equally number of rays are traced at each level

Or:



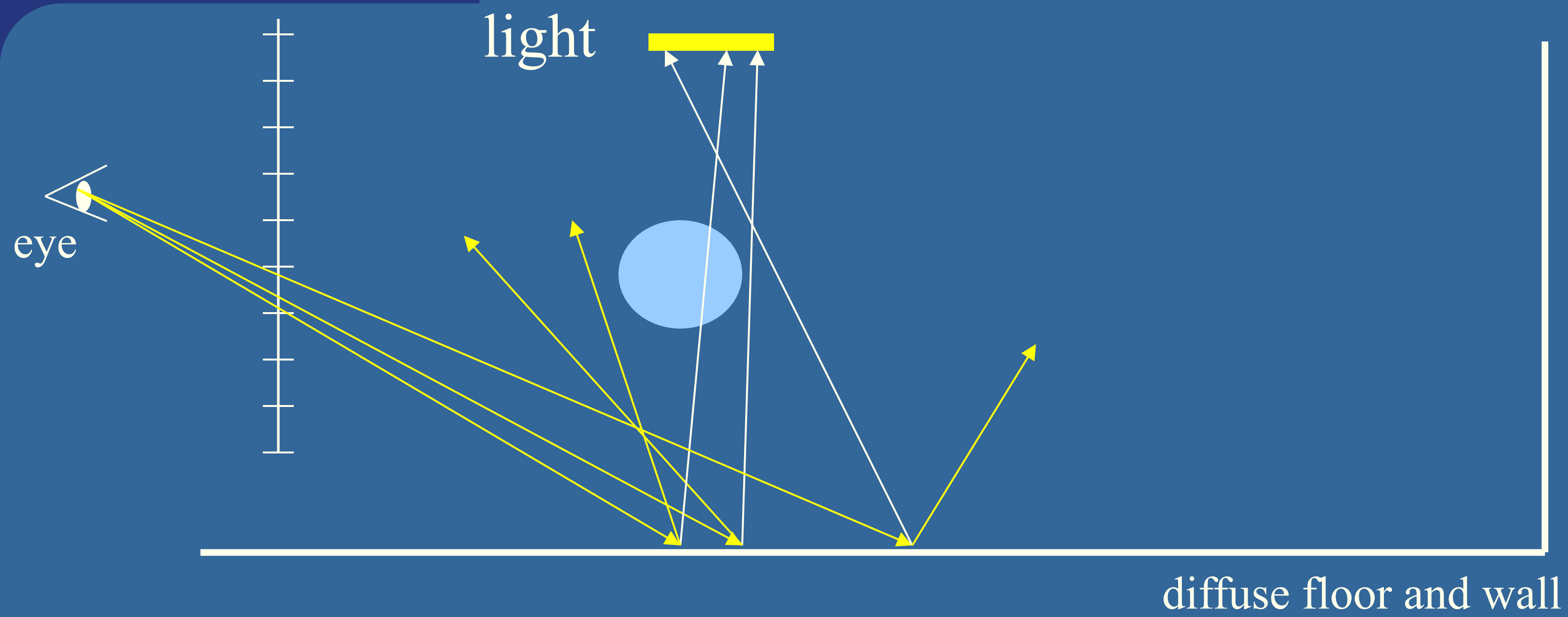
Even smarter: terminate path with some probablility after each level, since they have decreasing importance to final pixel color.

Path Tracing – indirect + direct illumination.



- Shoot many paths per pixel (the image just shows one light path).
  - At each intersection,
    - Shoot one shadow ray per light source
      - at random position on light, for area/volumetric light sources
    - and randomly select one new ray direction.

Path Tracing and area lights



- For area light sources, shoot the shadow ray to one random position on the area light. This gives soft shadows when many paths are averaged for the pixel.
- Example: Three paths for one pixel
  - At each ray intersection,
    - Pick ***one*** random position on light source
    - Send one random ray bounce to continue the path...

Path tracing: Summary

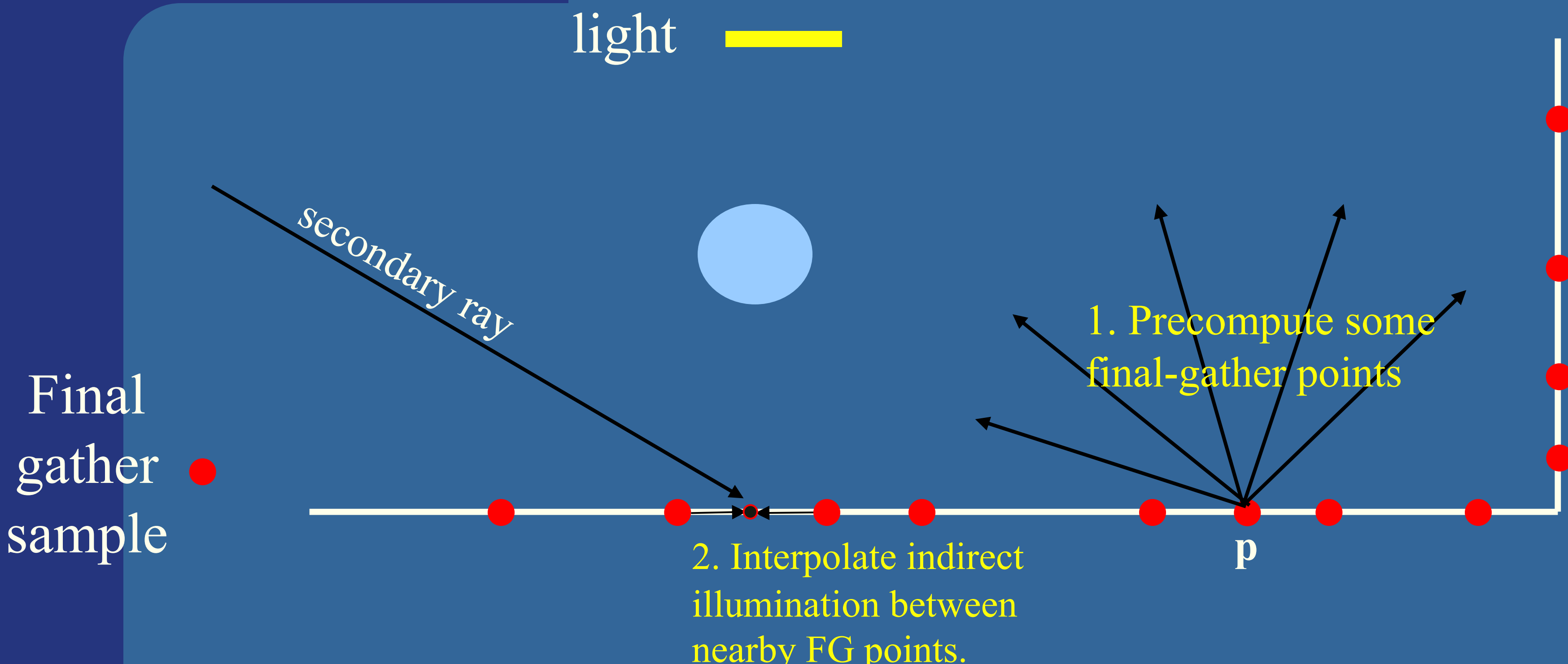
- Uses Monte Carlo sampling to solve integration:
  - by shooting many random ray ***paths*** over the integral domain.
  - Algorithm:
    - For each pixel, // we will shoot a number of paths:
      - For each path, generate the primary ray:
      - Repeat {
        1. Trace the ray. At hitpoint:
        2. Shoot one shadow ray and compute local lighting.
        3. Sample indirect illumination randomly over the possible reflection/refraction directions by generating **one** such new ray.
      - } until the path is randomly terminated (or the ray does not hit anything).
- Shorter summary: shoot many paths per pixel, by randomly choosing **one** new ray at each interaction with surface + **one** shadow ray per light. Terminate the path with a random probability

Final Gather

Popular for ray tracing and photon mapping but not path tracing

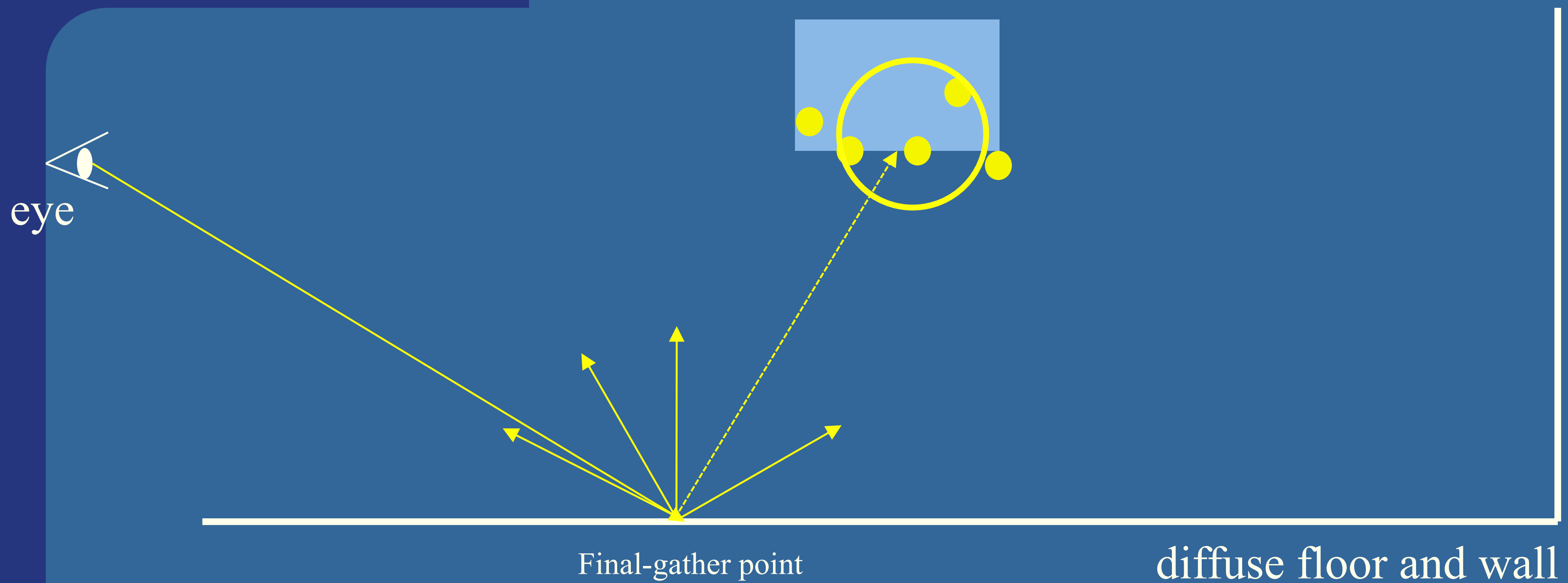
Idea and good answer:

- Compute indirect illumination somehow, but only at a few positions (final gather points) in the scene.
- Estimate indirect illumination for other positions by interpolation from nearby final-gather points



- Many versions of Final Gathering exist.
- E.g., to compute final-gather point **p**:
  - Send thousand(s) random rays out from **p** to sample indirect illumination
- To use during ray tracing: interpolate global illumination between nearby Final Gather points, to estimate incoming radiance at the ray's intersection point.
- Does not matter much if indirect illumination is blotchy for secondary rays.

Final Gather with Photon Mapping



- Too noicy to use the <u>global</u> map for direct visualization
- Remember: eye rays are recursively traced (via reflections/refractions) until a diffuse hit, **p**. There, we want to estimate slow-varying indirect illumination.
  - Instead of growing sphere in global map at **p**, Final Gather shoots 100-1000 indirect rays from **p** and grows sphere in the global map and also caustics map, where each of those rays end at a diffuse surface. Or interpolate from nearby already computed final-gather points.

Photon Mapping - Summary

- Creating Photon Maps:
  - Trace photons (~100K-1M) from light source. Store them in kd-tree when they hit diffuse enough surface (e.g., not 100% specular). Then, use russian roulette to decide if the photon should be absorbed or specularly or diffusively reflected. Create both global map and caustics map. For the Caustics map, we send more of the photons towards reflective/refractive objects.
- Ray trace from eye:
  - As usual: I.e., shooting primary rays and recursively shooting reflection/refraction rays, and at each intersection point **p**, compute direct illumination (shadow rays + local shading).
  - Also grow sphere around each **p** in caustics map to get caustics contribution and in global map to get slow-varying indirect illumination.
  - If final gather is used: At the first diffuse hit, instead of using global map directly, sample the indirect illumination around **p** by sampling the hemisphere with ~100–1000 rays and **then** use the two photon maps where those rays hit a surface.
- Growing sphere:
  - Uses the kd-tree to grow a sphere around **p** until a fixed amount (e.g. 50) photons are inside the sphere. Estimate outgoing radiance by using the material's brdf and the photons' powers and incoming directions.

Or shorter summary:

1. Shoot photons from light source, and let them bounce around in the scene, and store them where they land (e.g. in a kD-tree).
2. Ray-tracing pass from the eye. Estimate radiance at each ray hit, by growing a sphere (at the hit point in the kD-tree) until it contains a predetermined #photons. Use a caustics map and a global map.

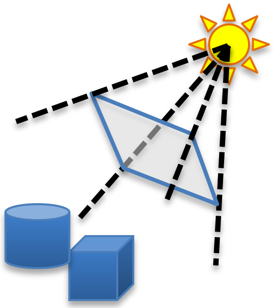
Lecture 11: Shadows + Reflection

- Point light / Area light
- Three ways of thinking about shadows
  - The basis for different algorithms.
- Shadow mapping
  - Be able to describe the algorithm
  - Percentage closer filtering
  - Cascaded shadow maps
- Shadow volumes
  - Be able to describe the algorithm
  - Stencil buffer, 3-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

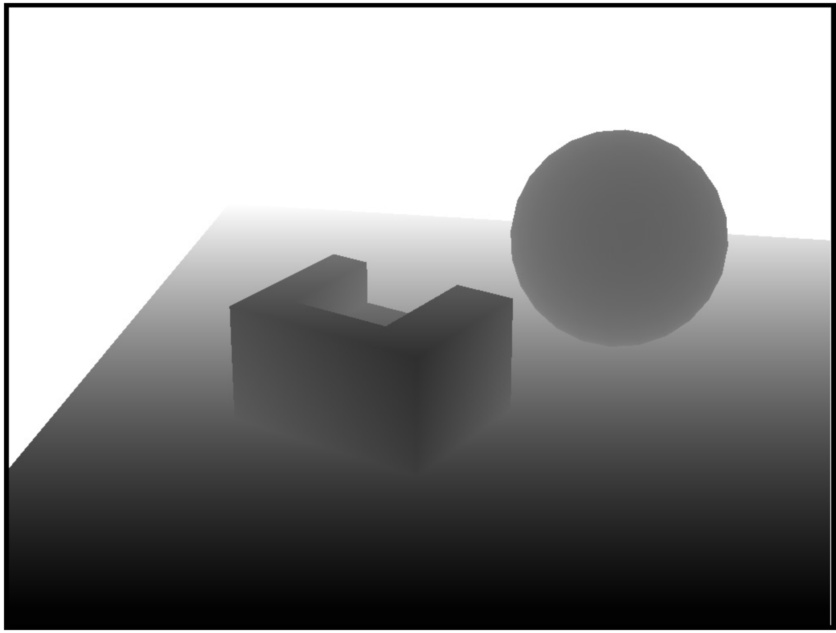
Shadow Maps - Summary



Shadow Map Algorithm:

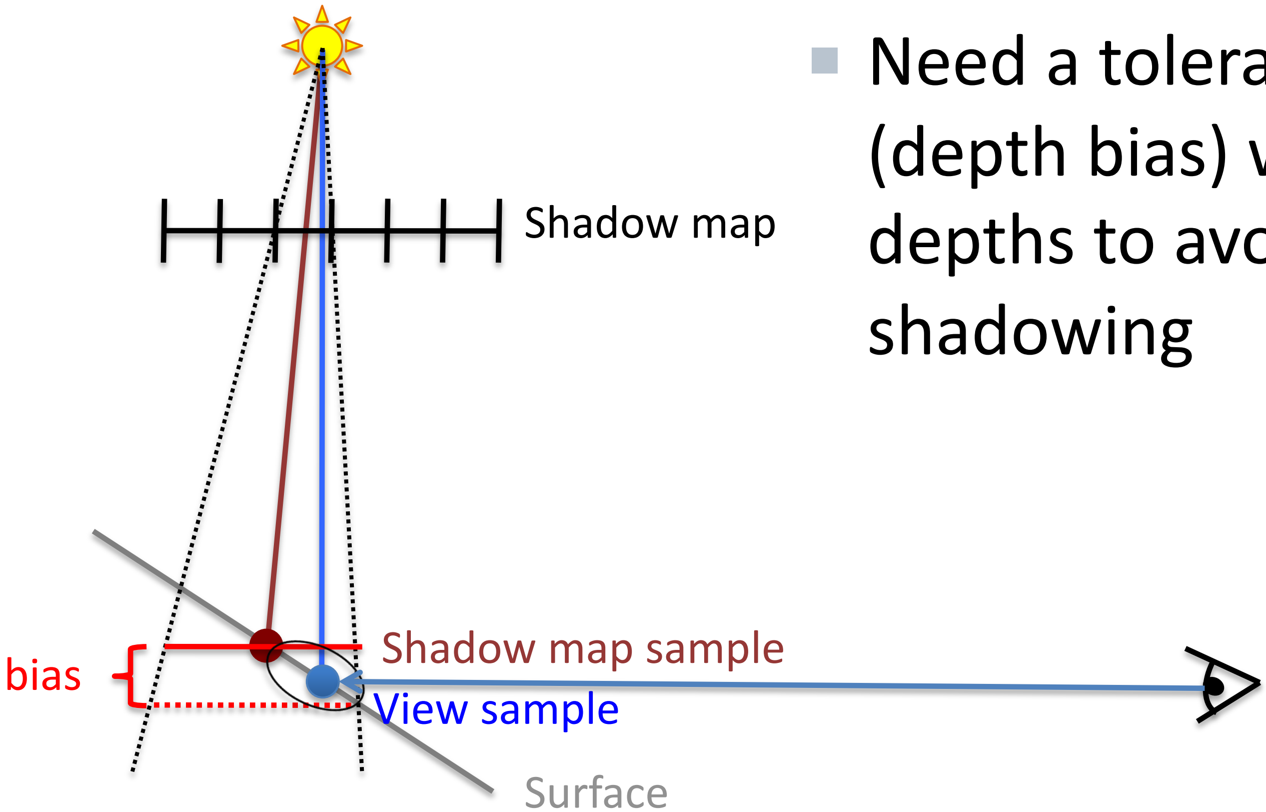
- Render a z-buffer from the light source
  - Represents geometry in light
- Render from camera
  - For every fragment:
    - Transform(warp) its 3D-pos (x,y,z) into shadow map (i.e. light space) and compare depth with the stored depth value in the shadow map
    - If depth greater-> point in shadow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

Understand z-fighting and light leaks



Shadow Map (=depth buffer)

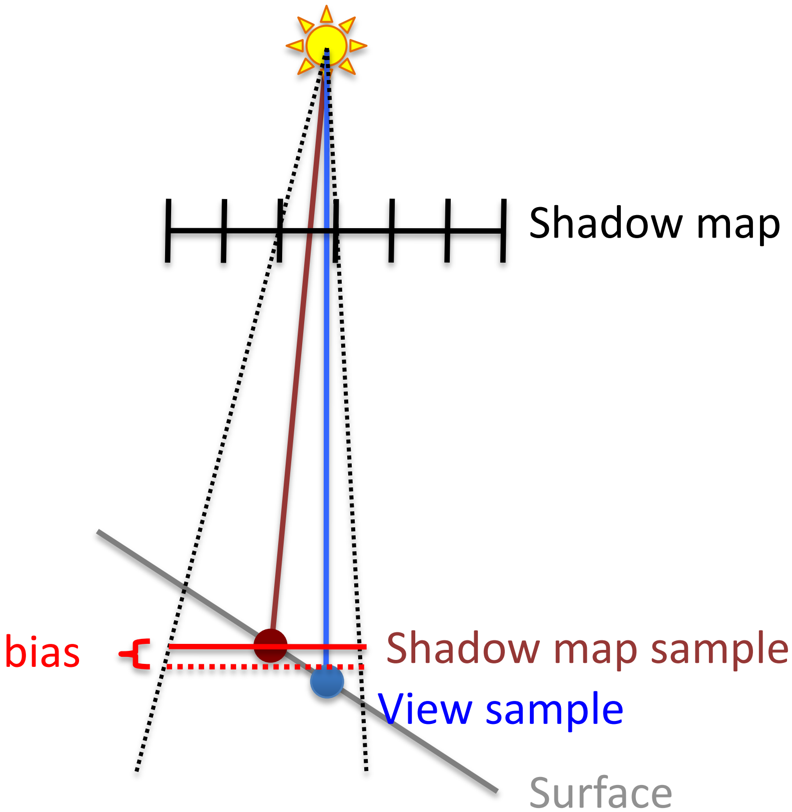
Bias

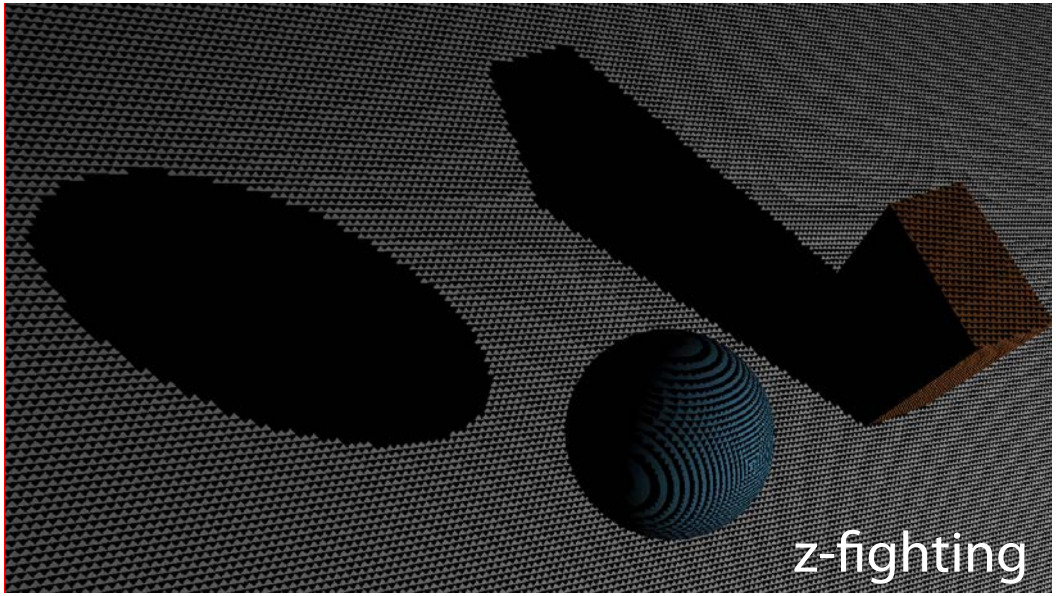


- Need a tolerance threshold (depth bias) when comparing depths to avoid surface self shadowing

Bias

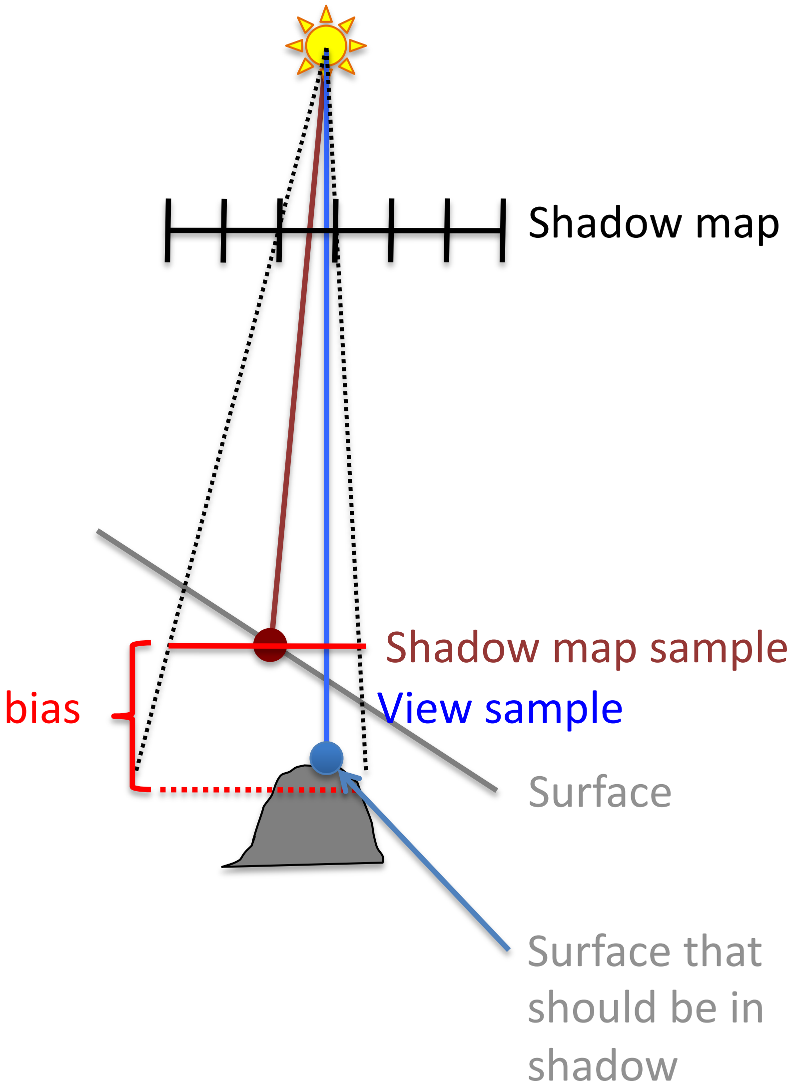
- Need a tolerance threshold (depth bias) when comparing depths to avoid surface self shadowing

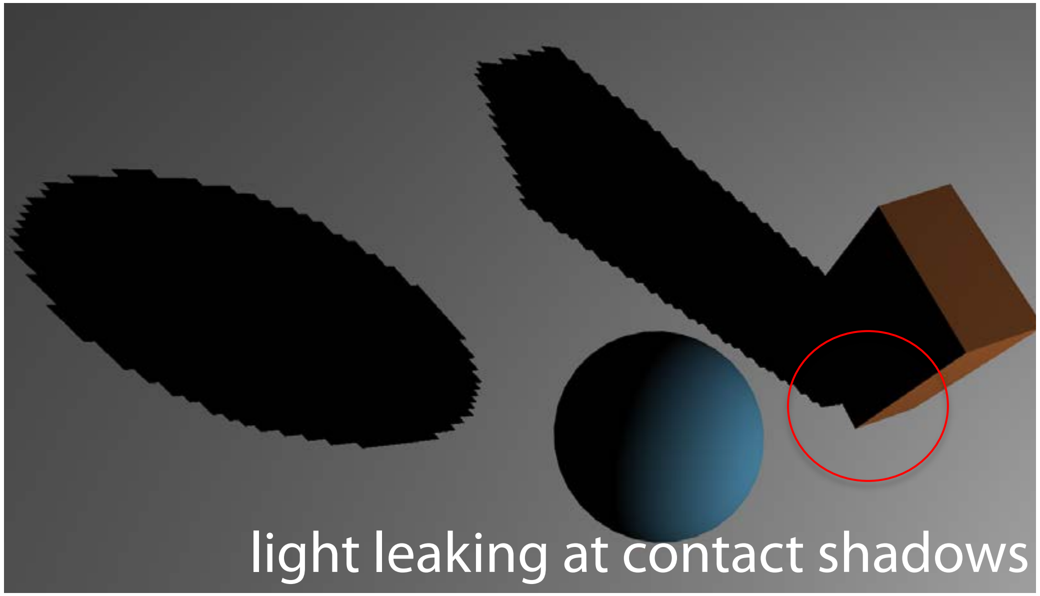




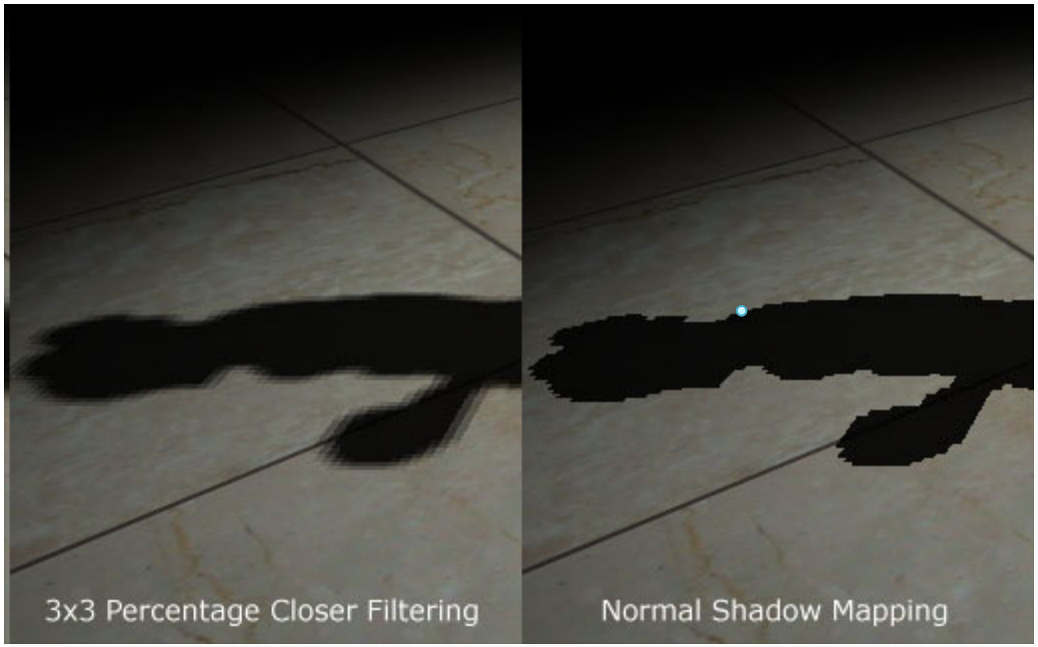
Bias

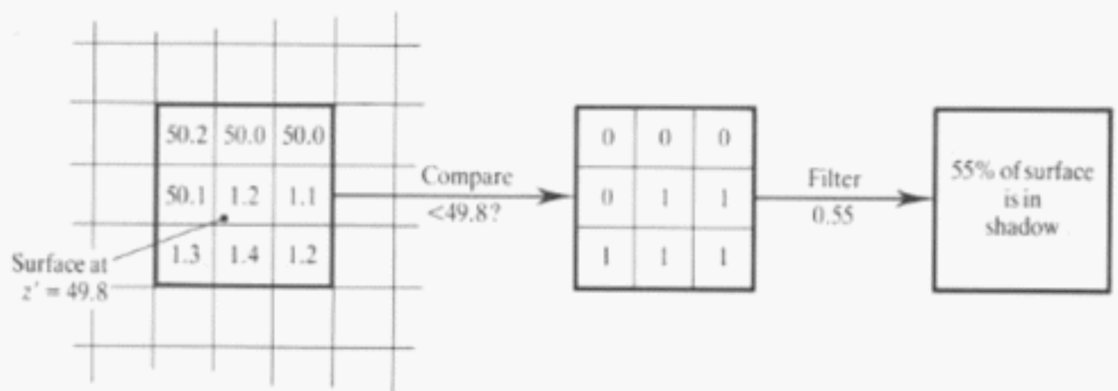
- Need a tolerance threshold (depth bias) when comparing depths to avoid surface self shadowing





Percentage Closer Filtering





(b)

Use a neighborhood of the SM pixel (e.g., 3x3 region) to compute an averaged shadow result of this region.

Cascaded Shadow Maps

- You need high SM resolution close to the camera and can use lower further away. So create a separate SMs per depth region of the view frustum, with higher spatial resolution closer to camera.

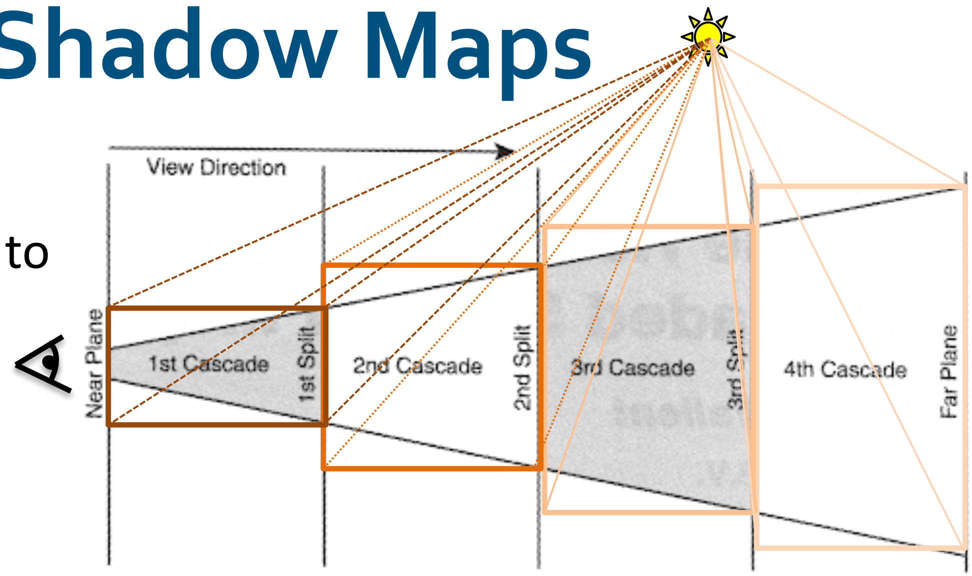
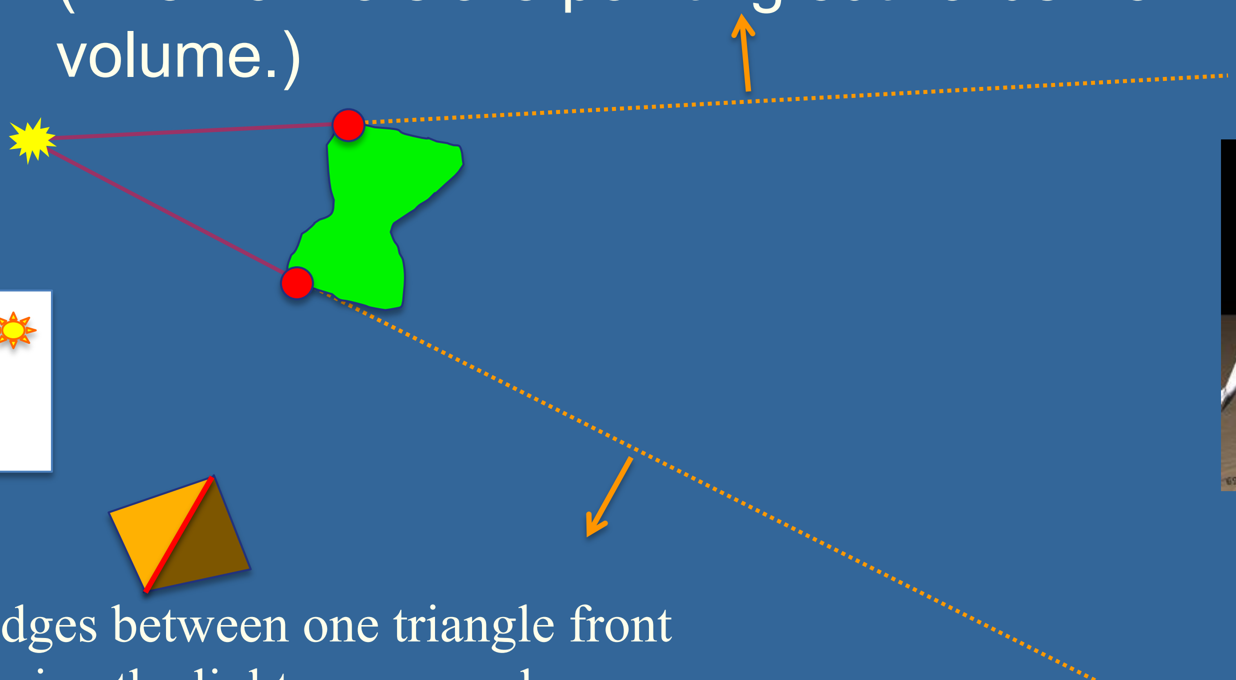
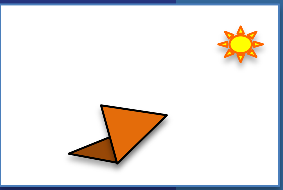


FIGURE 4.1.1 2D visualization of view frustum split (uniformly) into separate cascade frustums.

Shadow volumes

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



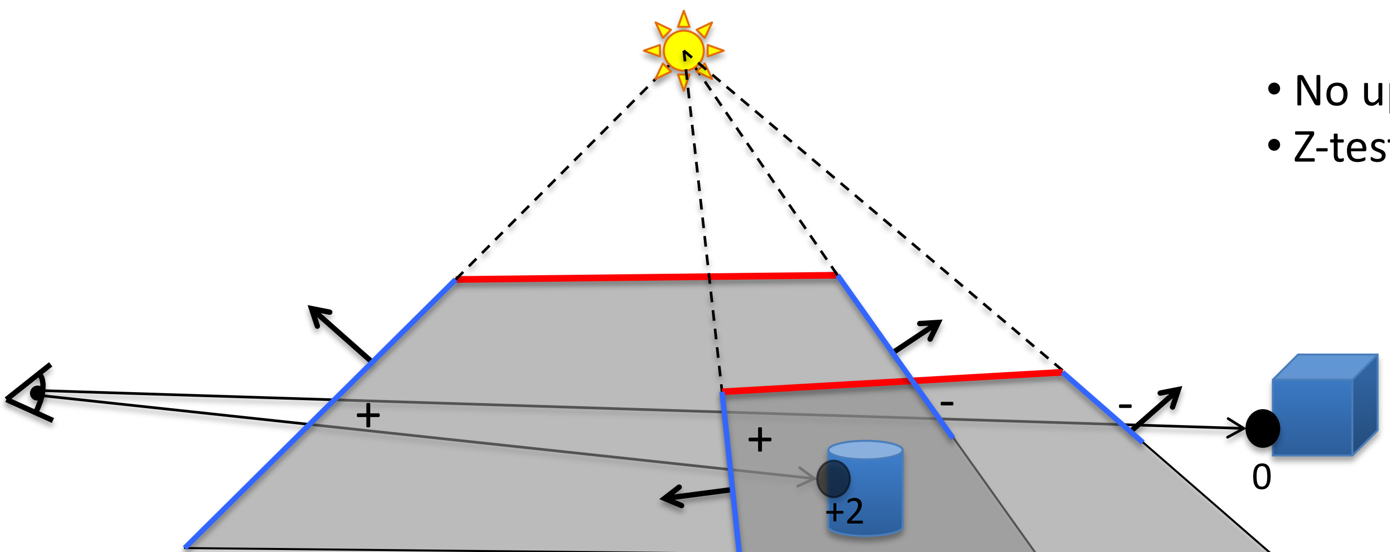


Edges between one triangle front facing the light source and one triangle back facing the light source are considered silhouette edges.

Then…

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



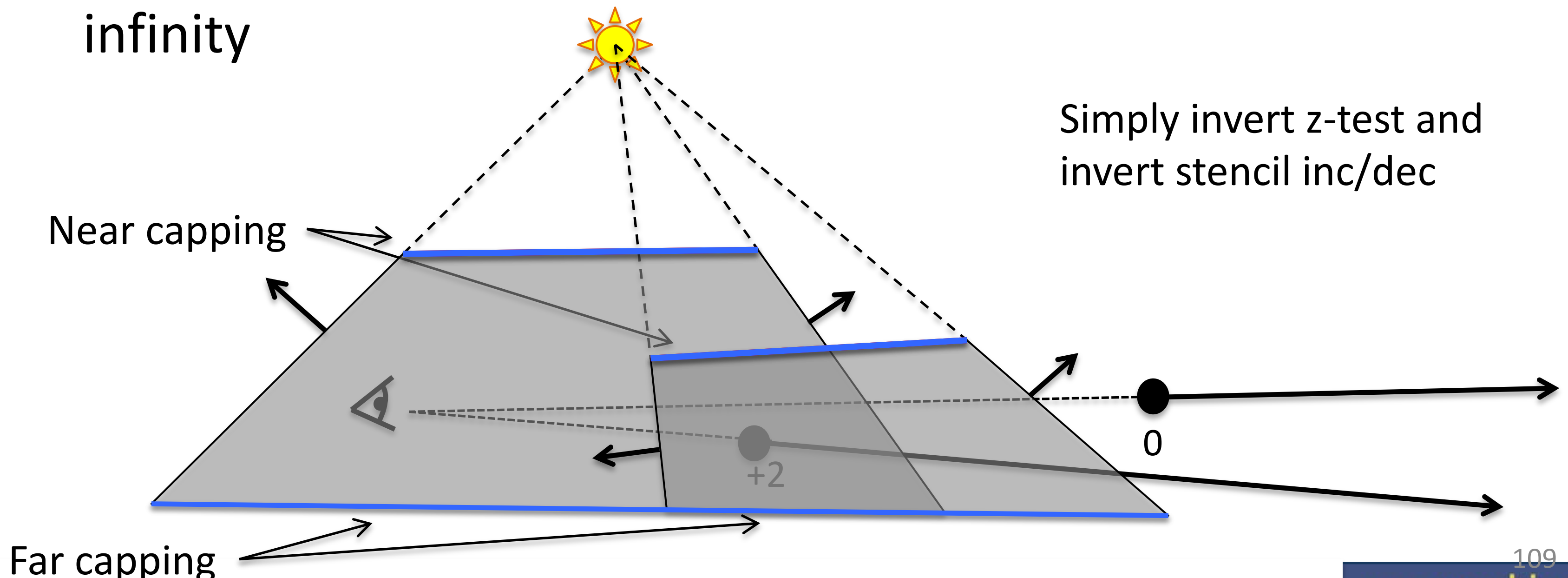
- No updating of z-buffer
- Z-test is enabled as usual

Shadow Volumes with the Stencil Buffer

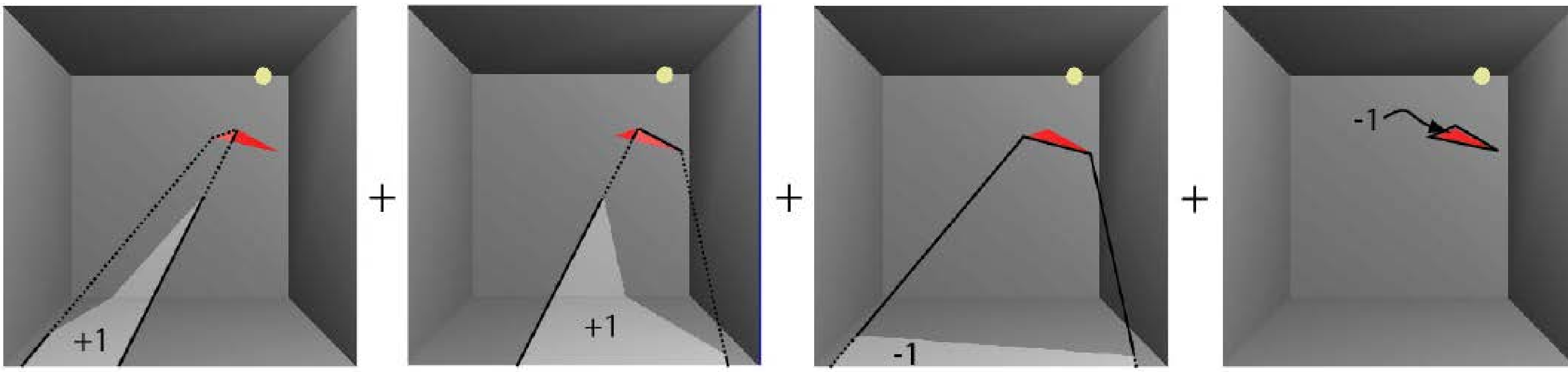
- A three pass process:
  - **1st pass:** Render *ambient* lighting
  - **2nd pass:**
    - Draw to stencil buffer only
      - Turn off updating of z-buffer and writing to color buffer but still use standard depth test
      - Set stencil operation to
        - » *incrementing* stencil buffer count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

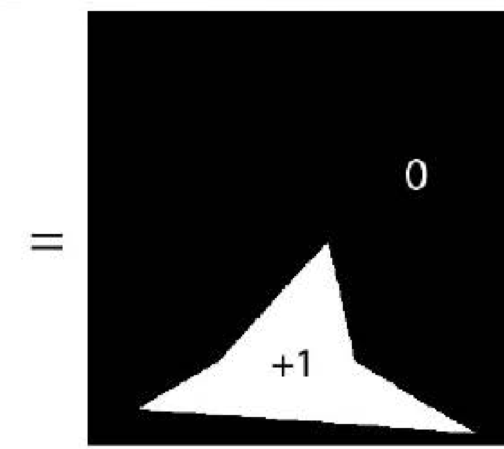
The Z-fail Algorithm

- 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





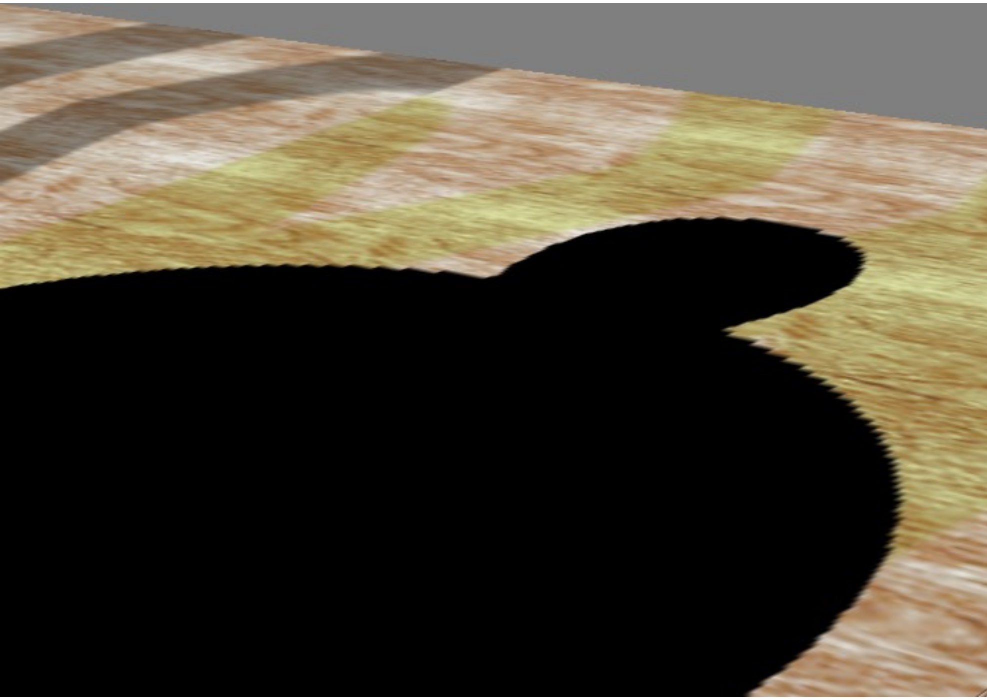
Compared to Z-pass:

Invert z-test

Invert stencil inc/dec

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

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.
  - Solution:
  - 6 SM (cube map), high res., use filtering (huge topic)

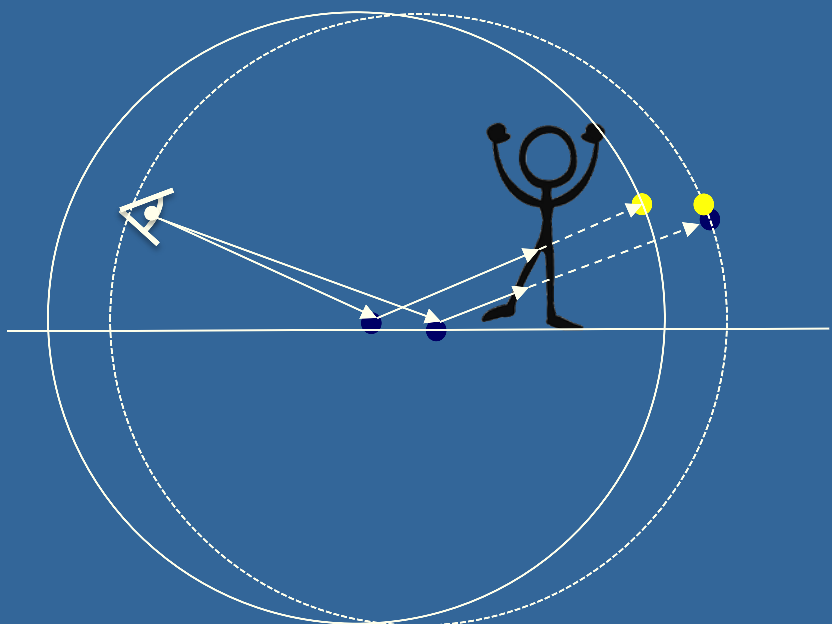
Shadow Volumes

- *Good*: shadows are **sharp**. Handles omni-directional lights.
- *Bad*: **3 passes**, shadow polygons must be generated and rendered $\rightarrow$ lots of polygons & **fill**
  - Solution: culling & clamping

Planar reflections

- We’ve already done reflections in curved surfaces with environment mapping. But the env.map is assumed to have an infinite radius, such that only the reflection ray’s direction (not origin) matters. Hence…
- …Environment maps does not work well for reflections in planar surfaces:

For two adjacent screen pixels, the cube map returns a too small uv change. Hence the reflection will be smeared out.

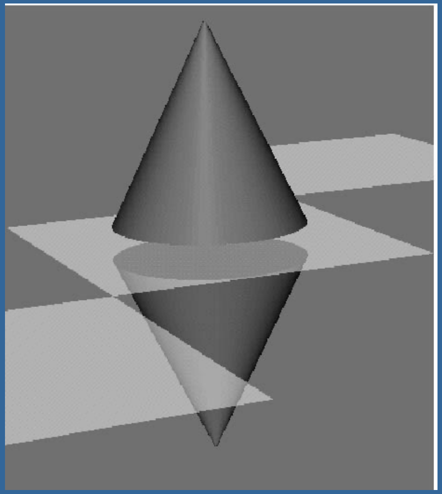


Standard cube map
(smear in xy)

Parallax corrected
(no smear)

- Parallax corrected cube maps fix this, but has its own problems. Ray tracing solves all but is slower. Purely planar reflections are actually easy to get by reflecting the geometry or camera as we will see on the next slide…

Planar reflections



Two methods:

1. Reflecting the object:
   - If reflection plane is z=0 (else somewhat more complicated – see page 504)
     - Apply `glScalef(1,1,-1);`
   - Backfacing becomes front facing!
     - i.e., use frontface culling instead of backface culling
   - Lights should be reflected as well

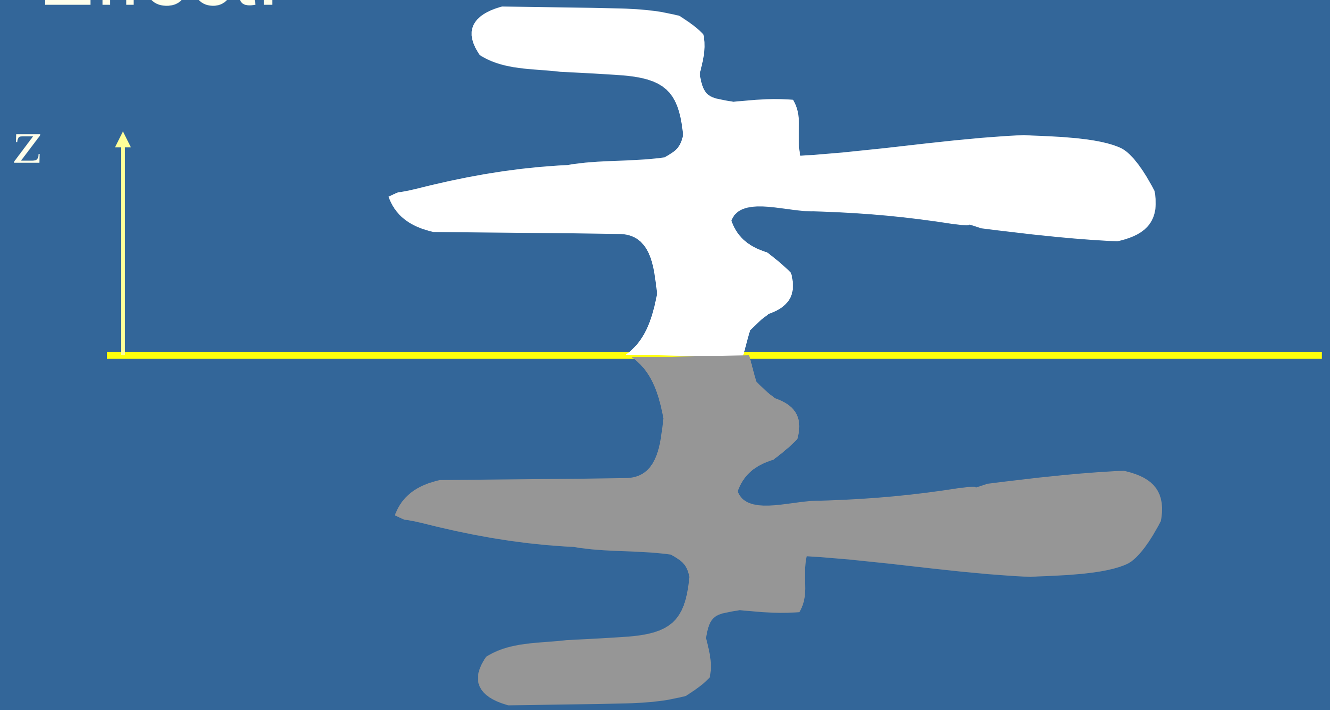
2. Reflecting the camera in the reflection plane

Planar reflections

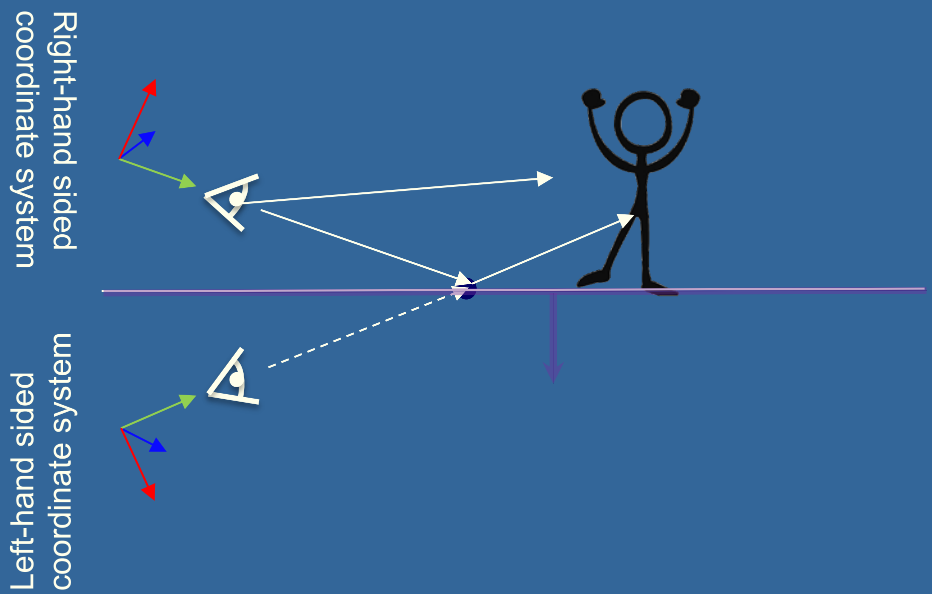
Important:

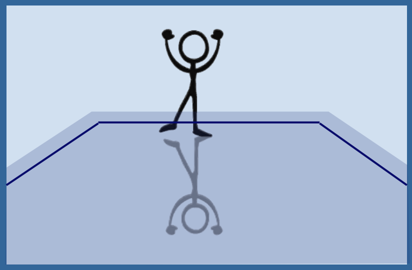
- render scaled (1,1,-1) model
- with reflected light pos.
- using front face culling

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



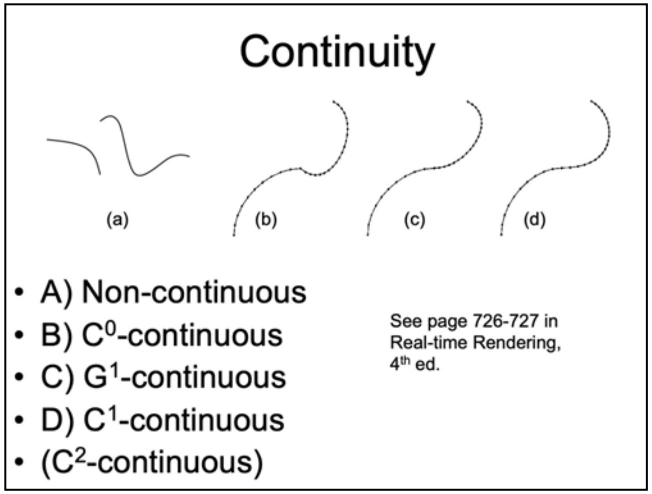
Or reflect camera position instead of the object:

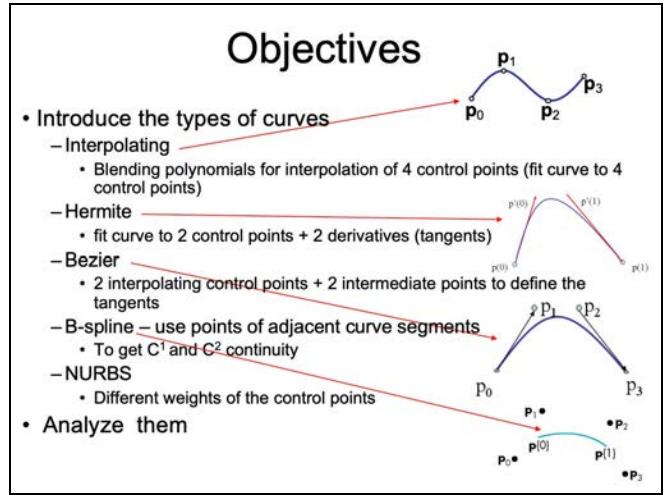


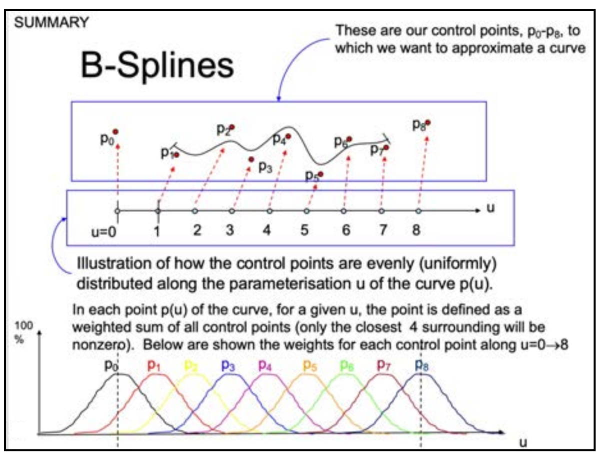


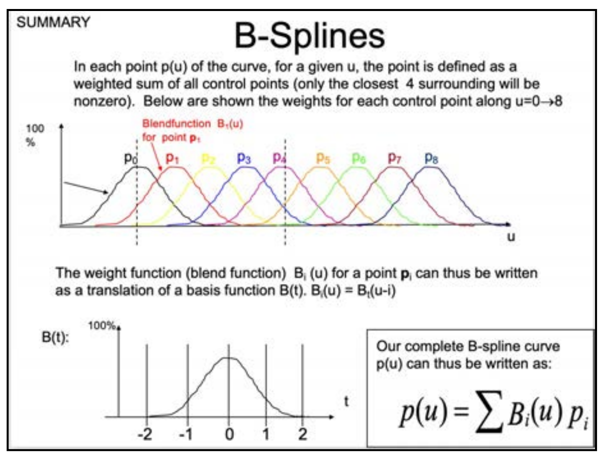
- Render reflection:
  1. Render reflective plane to stencil buffer
  2. Reflect camera including camera axes ← The important part!
  3. Set user clip plane in mirror plane to cull anything between mirror and reflected camera
  4. Render scene from reflected camera.
- Render scene as normal from original camera

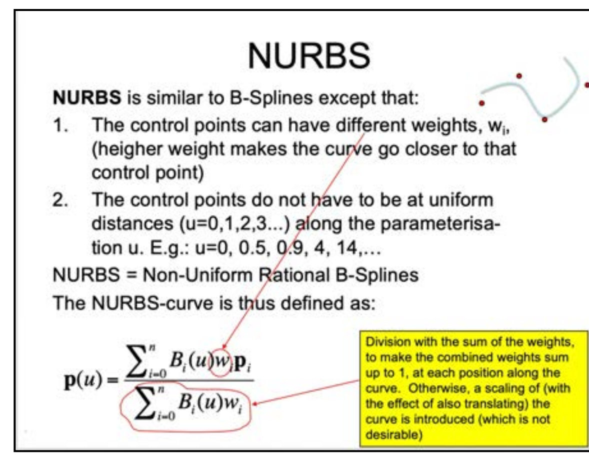
12. Curves and Surfaces – what you need to know:









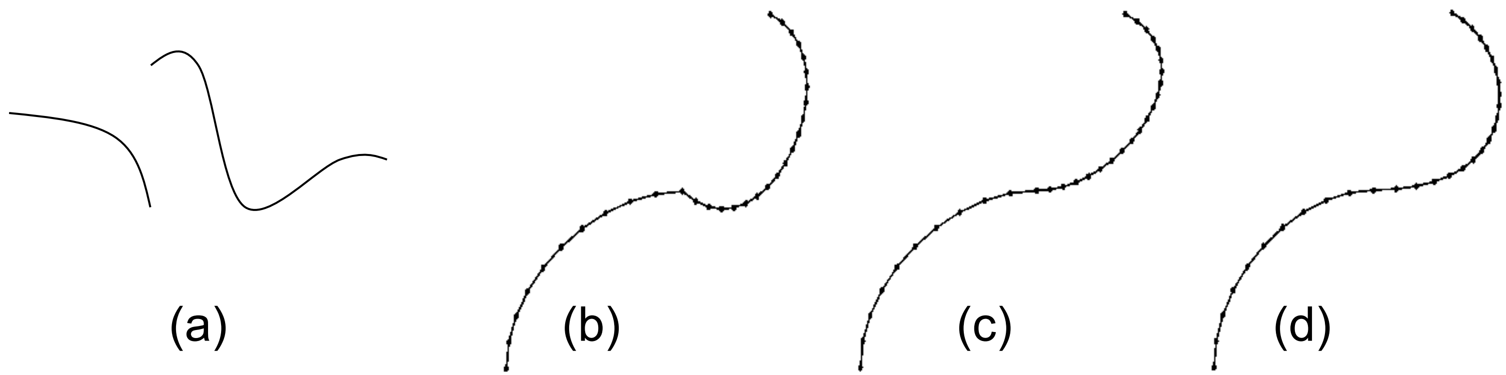


Curves and Surfaces - outline

Goal is to explain NURBS curves/surfaces…

- Introduce types of curves and surfaces
  - – Explicit – not general, easy to compute.
  - – Implicit – general, non-easy to compute.
  - – Parametric - general + simple to compute. We choose this.
- A complete curve is split into curve segments, each defined by a cubical polynomial.
  - – Introducing Interpolating/Hermite/Bezier curves.
- Adjacent segments should have C^2 continuity.
  - – Leads to B-Splines with a blending function (a spline) per control point
    - Each spline consists of 4 cubical polynomials, forming a bell shape translated along *u*. 
    - (Also, four bells will overlap at each point on the complete curve.)
- NURBS – a generalization of B-Splines:
  - – Control points at non-uniform locations along parameter *u*.
  - – Individual weights (i.e., importance) per control point

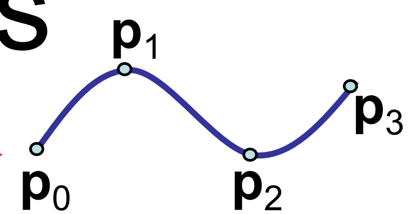
Continuity



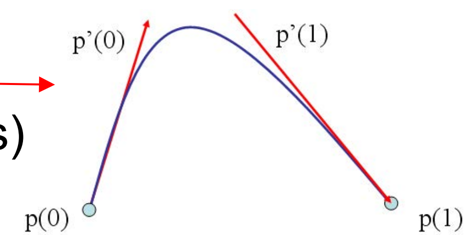
- A) Non-continuous
- B) C^0-continuous
- C) G^1-continuous
- D) C^1-continuous
- (C^2-continuous)

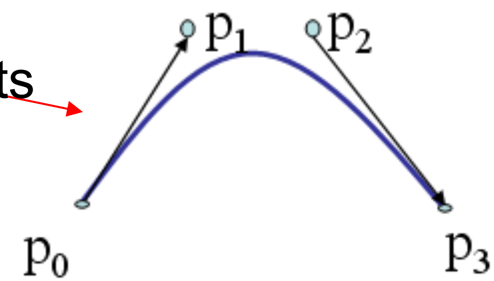
See page 726-727 in Real-time Rendering, 4th ed.

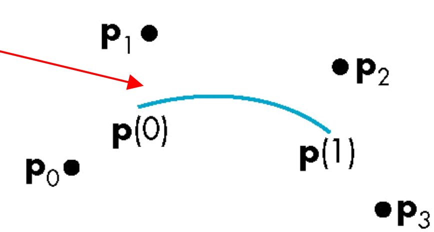
Types of Curves



- Introduce the types of curves
  - Interpolating
    - Blending 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 – use points of adjacent curve segments
    - To get C^1 and C^2 continuity
  - NURBS
    - Different weights of the control points
    - The control points can be at non-uniform intervalls







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 $\{B_i(u)\}$

12. Curves and Surfaces:

B-Splines

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

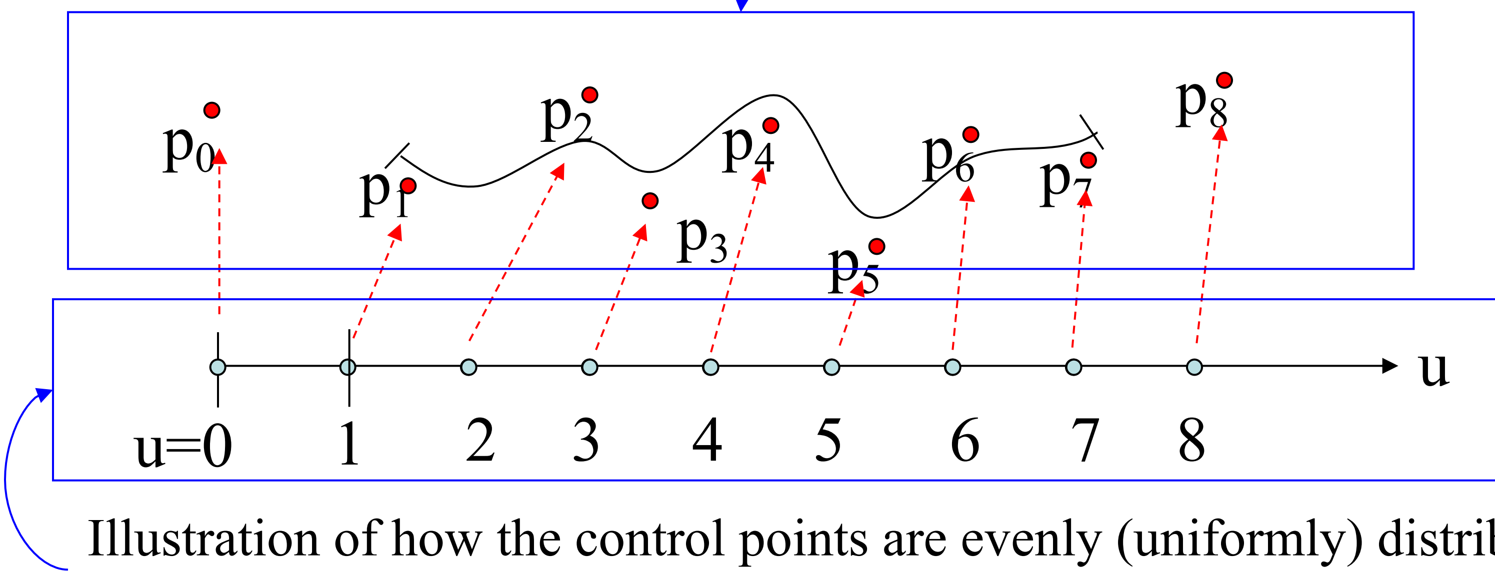
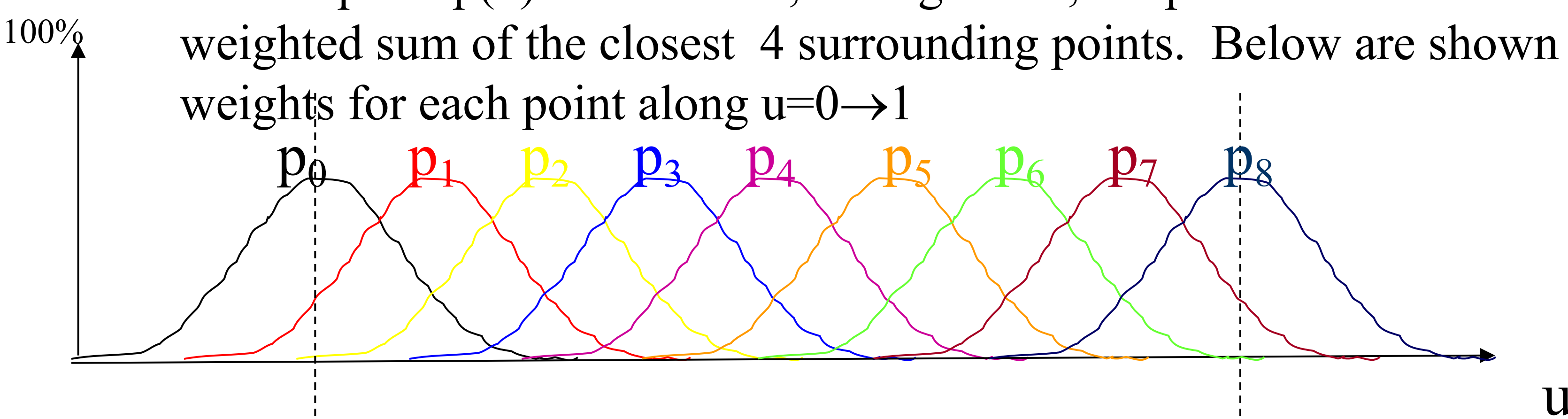


Illustration of how the control points are evenly (uniformly) distributed along the parameterisation u of the curve p(u).

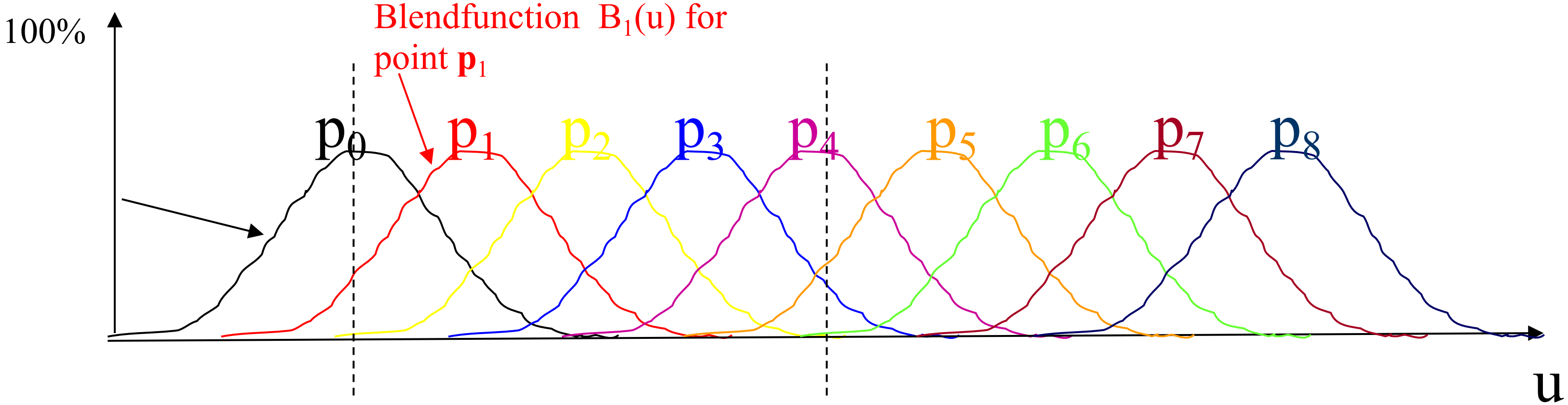
In each point p(u) of the curve, for a given u, the point is defined as a weighted sum of the closest 4 surrounding points. Below are shown the weights for each point along u=0→1



SUMMARY

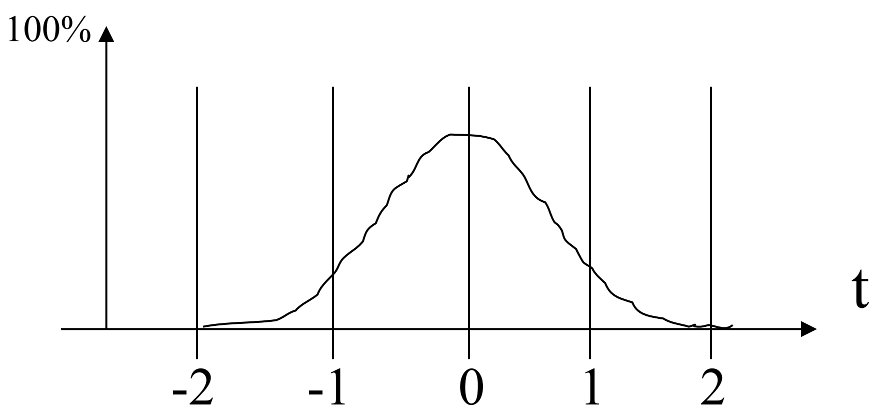
B-Splines

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→1



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

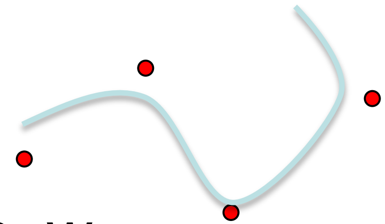
B(t):



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

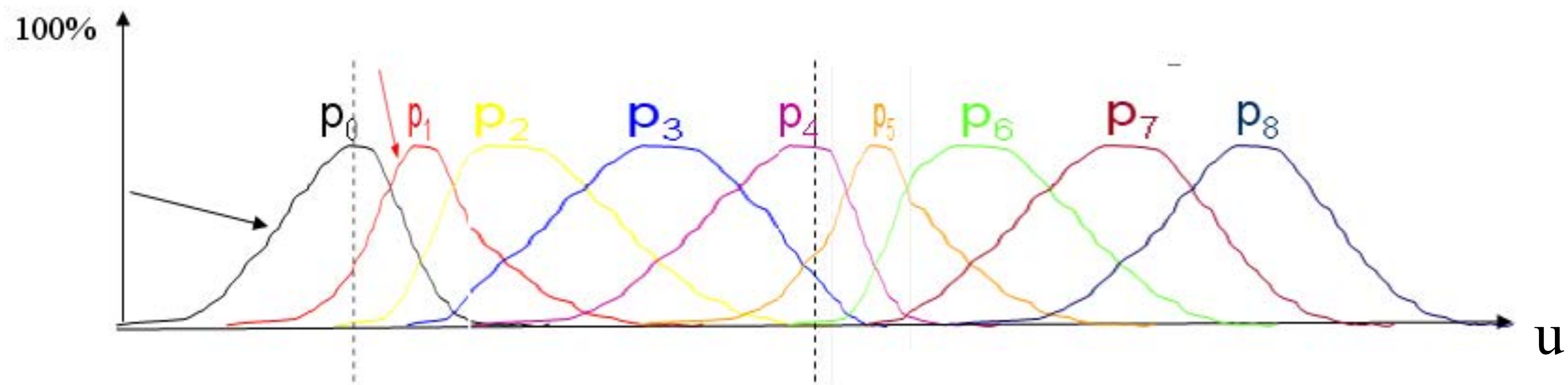
The NURBS-curve is thus defined as:

$$\mathbf{p}(u) = \frac{\sum_{i=0}^{n-1} B_i(u) w_i \mathbf{p}(i)}{\sum_{i=0}^{n-1} B_i(u) w_i}$$

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)

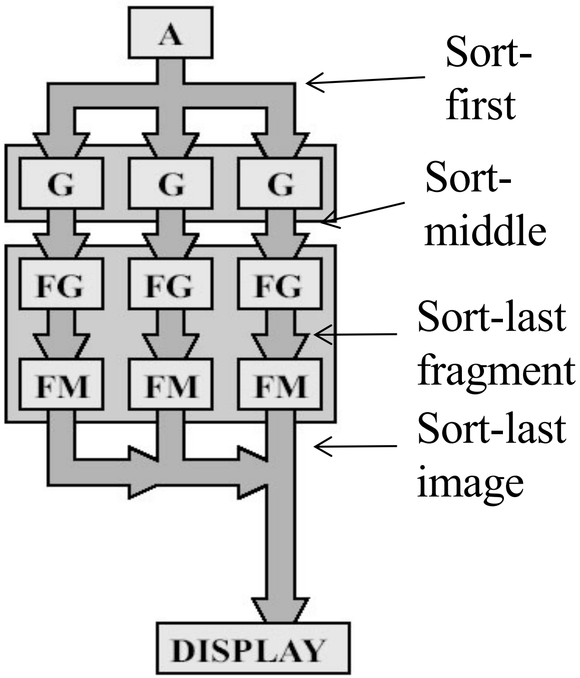
Lecture 13:

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

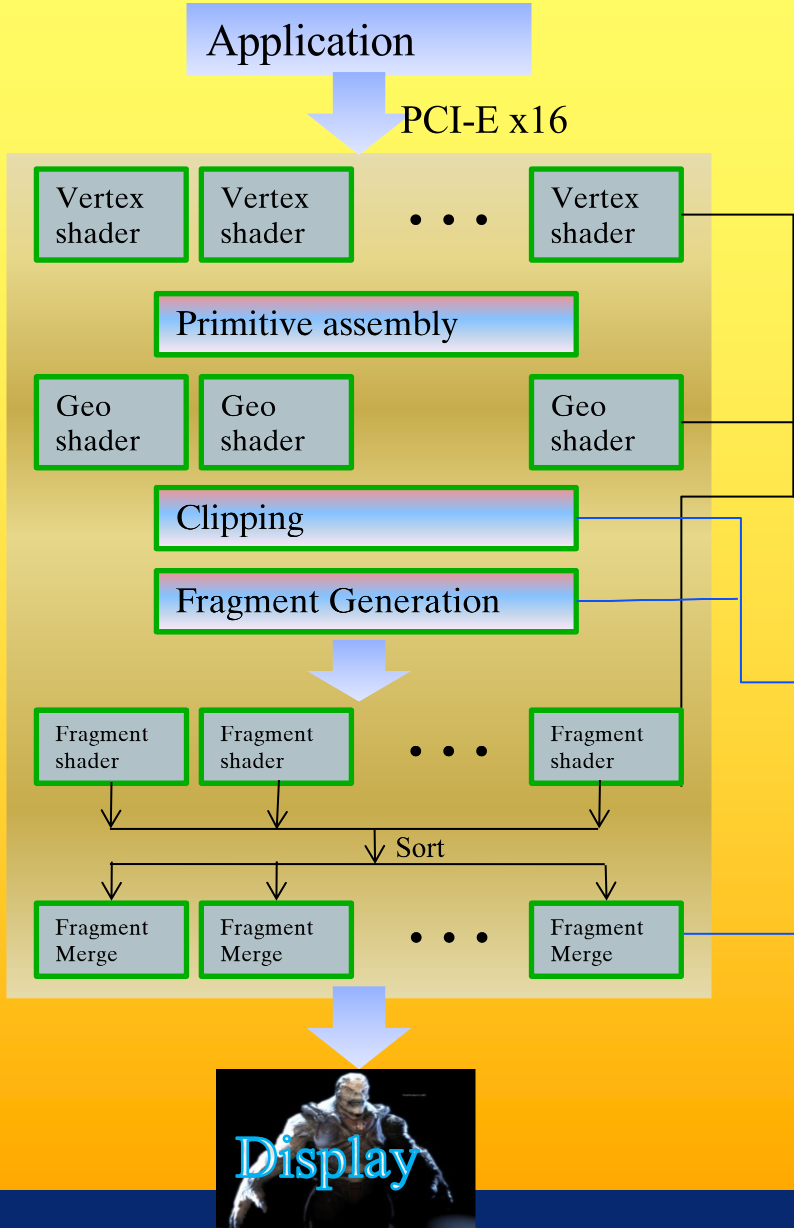
$$u_{ip} = (u/w)_{ip} / (1/w)_{ip}$$
$$v_{ip} = (v/w)_{ip} / (1/w)_{ip}$$

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 and how to ”solve” it
    - L1 / L2 caches
    - Texture caching with prefetching, (warp switching)
    - Texture compression, Z-compression, Z-occlusion testing (HyperZ)
- Be able to sketch the functional blocks and relation to hardware for a modern graphics card (next slide→)



The graphics-pipeline's funcional blocks and their relation to hardware



Vertex-, Geometry- and Fragment shaders exectued on a pool of thousands of ALUs

Fixed function hardware

Fixed function hardware