Repetition
av
TDA360 / INN220

Ulf Assarsson

Labtider lv 8

- Labbet är ledigt i princip 100% hela veckan.
- Handledare finns på plats:

  Tisd:

  – Lab3: 13.00-17.00, Johan

  – Multiplayerlab: 17.00 - 20.00, Ulf
- Onsd: 18.00 $\rightarrow$ ~21.00
- Torsd: 13.00 $\rightarrow$ 18.00

- Dessa slides är en sammanfattning av allt jag anser viktigt att kunna inför tentan

- Tentamensdatum lördag 28 Oktober em VV-salar
- 01. Introduktion Ingenting på tentan
- 02. Pipeline and OpenGL Ingenting på tentan
- 03. Vectors and Transforms
- 04. OpenGL
- 05. Shading
- 06. Rasterization, Depth Sorting and Culling
- 07. Texturing
- 08. Ray Tracing and Radiosity
- 09. Curves and Surfaces
- 10. Vertex and Fragment Shaders
- 11. Shadows and Reflections

Homogeneous notation

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

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

rotations-delen

Translationsdelen

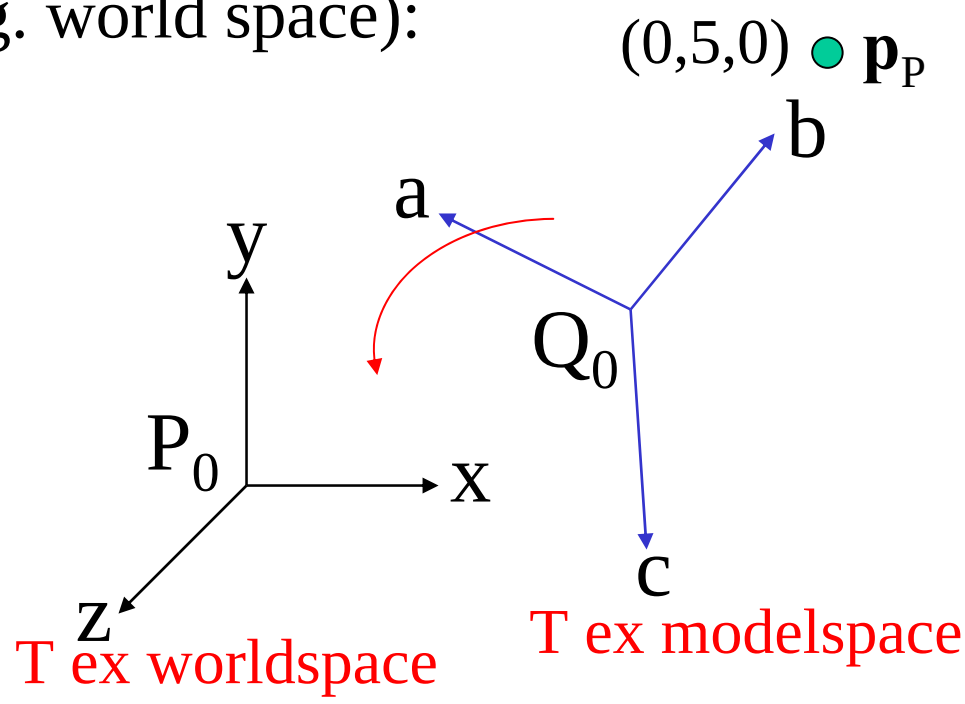
- A vector (direction): $\mathbf{d} = \begin{pmatrix} d_x & d_y & d_z & 0 \end{pmatrix}^T$
- Translation of vector: $\mathbf{T}\mathbf{d} = \mathbf{d}$
- Also allows for projections (later)

Change of Frames

Computing the matrix $M_{Q\to P}$ that transforms a vertex from coordinate system Q (e.g. model space) into coordinate system P (e.g. world space):

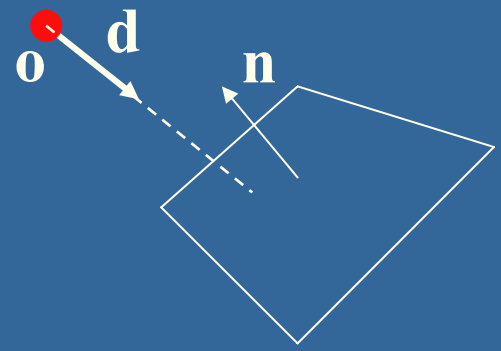
Basvektorerna **a**,**b**,**c** är uttryckta i världskoordinatsystemet

$$M_{Q\to P} = \begin{bmatrix} a_x & b_x & c_x & 0 \\ a_y & b_y & c_y & 0 \\ a_z & b_z & c_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Ex: $\mathbf{p}_P = M_{Q\to P}\,\mathbf{p}_Q = M_{Q\to P}\,(0,5,0)^T = 5\,\mathbf{b}$

Analytical: Ray/plane intersection



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

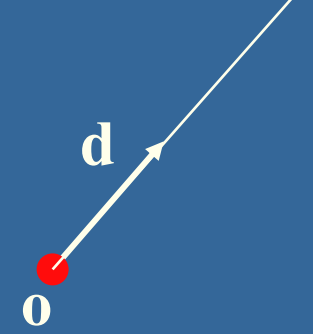
- Replace $\mathbf{p}$ by $\mathbf{r}(t)$ and solve for t:

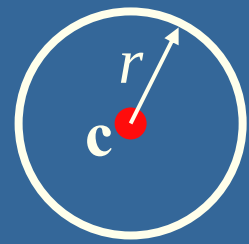
  $\mathbf{n}\cdot(\mathbf{o}+t\mathbf{d}) + d = 0$

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

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

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*

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

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

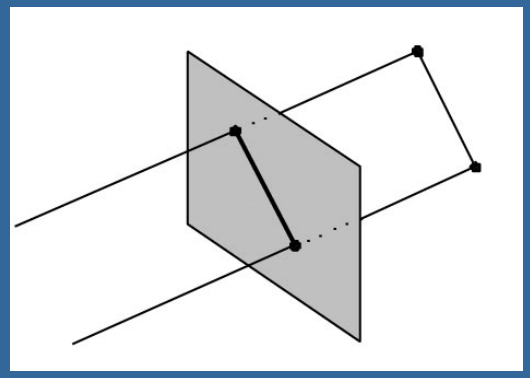
$$(\mathbf{d}\cdot\mathbf{d})t^2+2((\mathbf{o}-\mathbf{c})\cdot\mathbf{d})t+(\mathbf{o}-\mathbf{c})\cdot(\mathbf{o}-\mathbf{c})-r^2=0$$

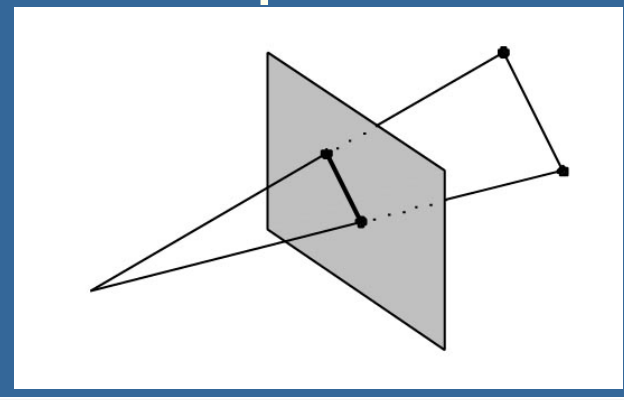
$$t^2+2((\mathbf{o}-\mathbf{c})\cdot\mathbf{d})t+(\mathbf{o}-\mathbf{c})\cdot(\mathbf{o}-\mathbf{c})-r^2=0 \qquad \|\mathbf{d}\|=1$$

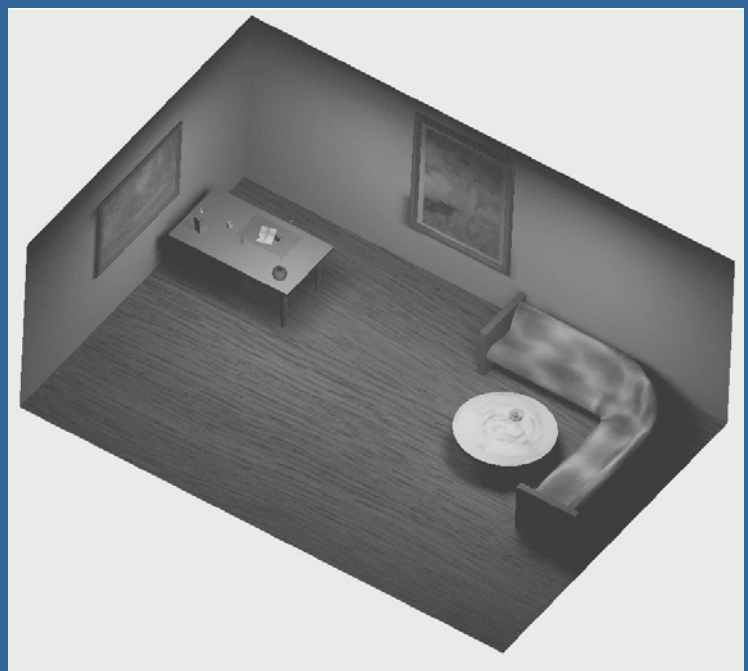
Detta är en standard andragradsekvation. Lös ut t.

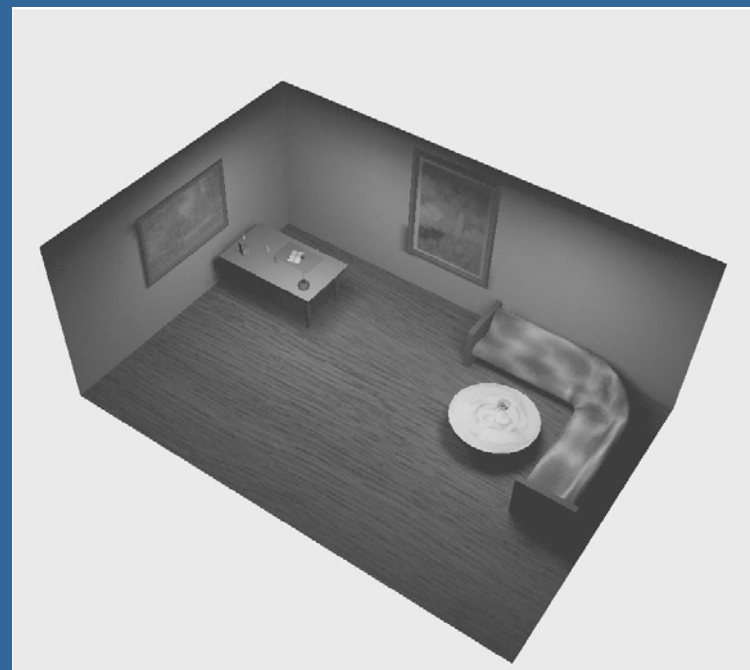
Projections

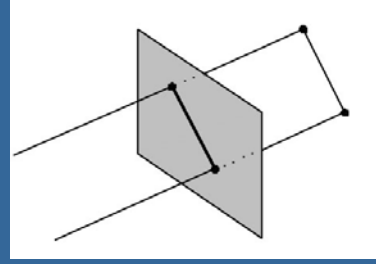
- Orthogonal (parallel) and Perspective







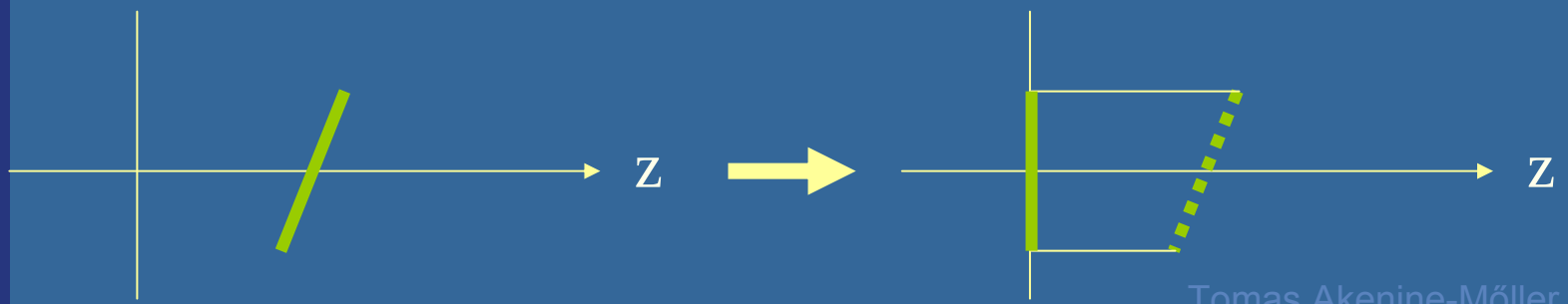




Orthogonal projection

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

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



Specifying vertices and polygons

- OpenGL is a state machine. Commands typically change the current state
- Multiple calling formats for the commands: `void` **glVertex{234}{sifd}**(`T` *coords*);
- glBegin()/glEnd(). (Slow)

```
glBegin(GL_TRIANGLE)
        glVertex3f(0,0,0)
        glVertex3f(0,1,0);
        glVertex3f(1,1,0);
glEnd();
```

  Optional: Can specify for instance glColor3f(r,g,b), glTexCoord2f(s,t), glNormal3f(x,y,z) - typically per vertex or per primitive.

- **Display lists** are created with surrounding glNewList() and glEndList(). (Fast)

```
int dlist=glGenLists(1);                    // generate one display list number
glNewList(dlist,GL_COMPILE);                // indicates start of display list
    glutSolidSphere(radius, 20, 20);        // or any object specification
glEndList();                                // indicates end of display list
```

  Display lists are then drawn with: glCallList(dlist);

Idea: Objects are stored internally in the most efficient format

- **Vertex Arrays (Fast):**

`void` **glTexCoordPointer**(`int` *size*, `enum` *type*, `sizei` *stride*, `void` **pointer*); //Defines an array of texture coordinates

`void` **glColorPointer**(`int` *size*, `enum` *type*, `sizei` *stride*,`void` **pointer*); //Defines an array of colors

(`void` **glIndexPointer**(`enum` *type*, `sizei` *stride*, `void` **pointer*); //Defines an array of color indices)

`void` **glNormalPointer**(`enum` *type*, `sizei` *stride*,`void` **pointer*); //Defines an array of normals

`void` **glVertexPointer**(`int` *size*, `enum` *type*, `sizei` *stride*,`void` **pointer);* //Defines an array of vertices

`void` **glEnableClientState**(`enum` *array*); `void` **glDisableClientState**(`enum` *array*); with *array* set to `TEXTURE_COORD_ARRAY`, `COLOR_ARRAY`, `INDEX_ARRAY`, `NORMAL_ARRAY`, or `VERTEX_ARRAY`,

`void` **glArrayElement**(int *i*); // Transfers the ith element of every enabled array to OpenGL

`void` **glDrawArrays**(mode; first; count); // Renders multiple primitives from the enabled arrays

`void` **glDrawElements**(mode, count, type,*indices); // Same as glDrawArrays, but takes array elements pointed to by indices.

`void` **glInterleavedArrays**(format; stride; pointer); // One array with vertices, normals, textures and/colors interleaved

Coordinate transformations

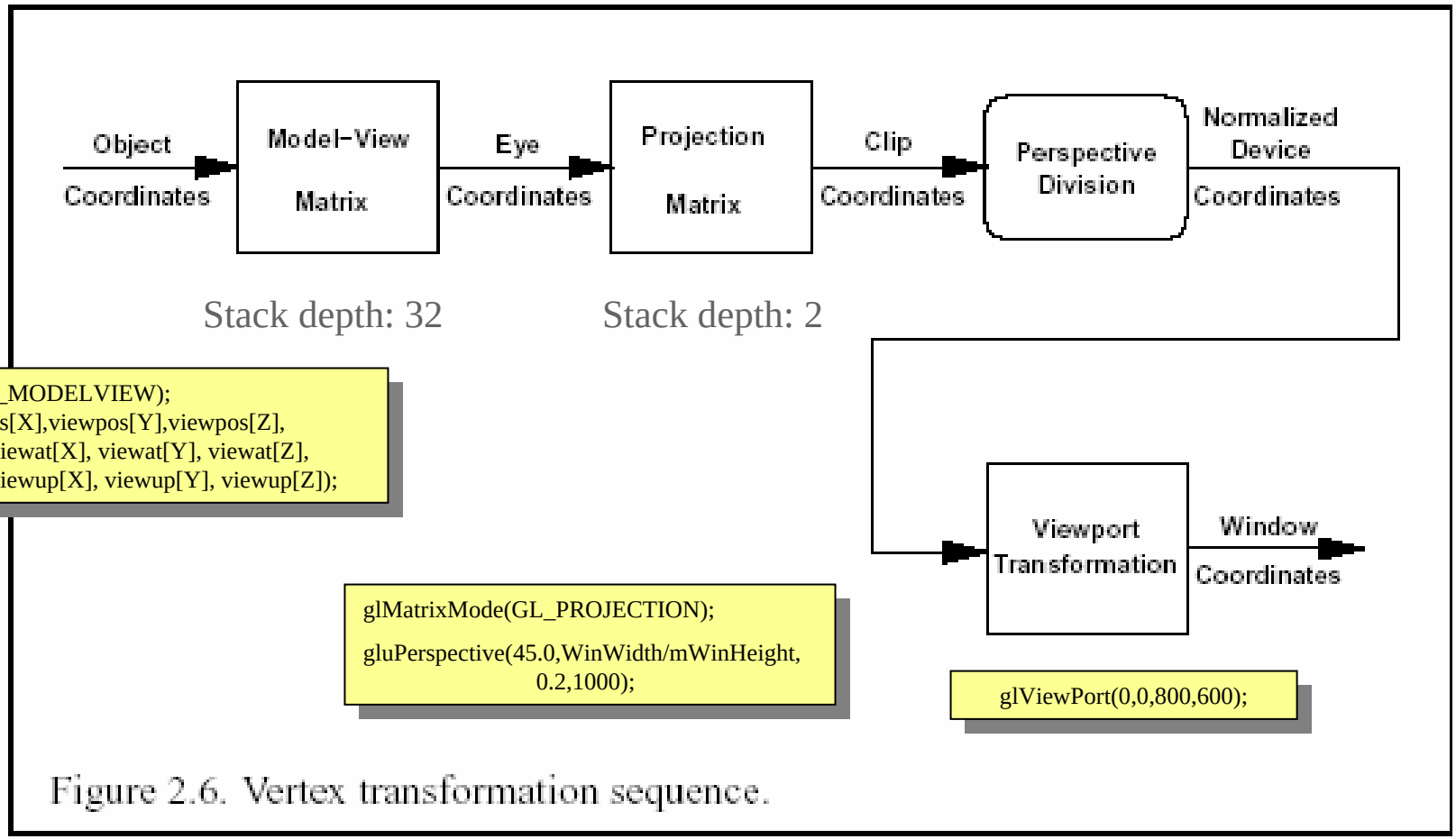


Figure 2.6. Vertex transformation sequence.

Matrix and Stack Operations

- Matrix Operations:
  - **glMatrixMode(*GL_MODELVIEW* or *GL_PROJECTION*)**
  - **glLoadIdentity(), glMultMatrix()**
  - **glRotate(),glTranslate(), glScale(), (glFrustum(), glOrtho())**
  - **GLU Helper functions: gluPerspective(), gluLookAt, gluOrtho2D()**(good for 2D rendering like text)
- Stack Operations:
  - **glPushMatrix(), glPopMatrix()**
  - **glPushAttrib(), glPopAttrib()**

Lighting and Colors

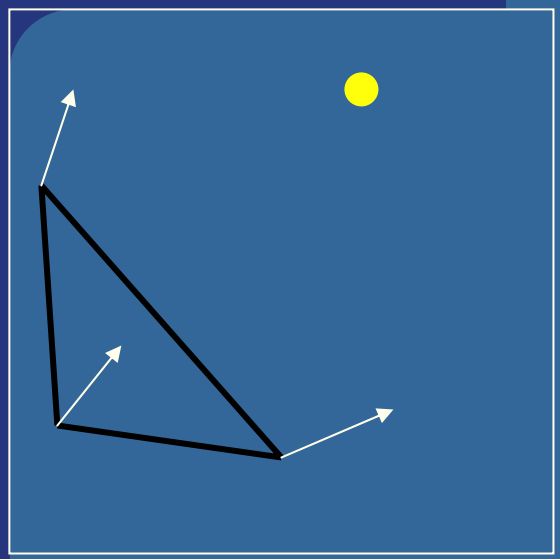
- glColor4f(r,g,b,a), glColor3f(r,g,b)
  - Used when lighting is disabled:
  - Disable with `glDisable( GL_LIGHTING );`
  - Could be changed for instance per vertex or per object. Can also be specified with glColorPointer() as described previously
- glMaterialfv()
  - Used when lighting is enabled.
  - Enable with `glEnable( GL_LIGHTING );`
  - Must also enable lights: `glEnable( GL_LIGHTn );`
  - Example:
    - glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, float rgba[4])
    - glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, float rgba[4])
    - glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, float rgba[4])
    - glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, float rgba[4])
    - glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, 30)

Buffers

- **Frame buffer**
  - Back/front/left/right – **glDrawBuffers()**
  - Alpha channel: **glAlphaFunc()**
- **Depth buffer (z-buffer)**
  - For correct depth sorting
  - Instead of BSP-algorithm, painters algorithm…
  - **glDepthFunc(), glDepthMask, glDepthRange()**
- **Stencil buffer**
  - Shadow volumes,
  - **glStencilFunc(), glStencilMask, glStencilOp() –see www.msdn.com**
- **Accumulation buffer**
  - **glAccum(),** GL_LOAD, GL_ACCUM, GL_ADD, GL_MULT, GL_RETURN
- General commands:
  - **glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_ACCUM_BUFFER_BIT | GL_STENCIL_BUFFER_BIT)**
  - Specify clearing value: **glClearAccum(), glClearStencil(), glClearColor()**

Lighting

Ni skall kunna totala ljusmodellen



Light:

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

Material:

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

Material Properties

- Material properties are also part of the OpenGL state
- Set by **`glMaterialv()`**

```
GLfloat ambient[] = {0.2, 0.2, 0.2, 1.0};
GLfloat diffuse[] = {1.0, 0.8, 0.0, 1.0};
GLfloat specular[] = {1.0, 1.0, 1.0, 1.0};
GLfloat emission[] = {0.0, 0.0, 0.0, 0.0};
GLfloat shine = 100.0
glMaterialf(GL_FRONT, GL_AMBIENT, ambient);
glMaterialf(GL_FRONT, GL_DIFFUSE, diffuse);
glMaterialf(GL_FRONT, GL_SPECULAR, specular);
glMaterialf(GL_FRONT, GL_EMISSION, emission);
glMaterialf(GL_FRONT, GL_SHININESS, shine);
```

Defining a Point Light Source

- For each light source, we can set an RGBA for the diffuse, specular, and ambient components, and for the position

```
GL float diffuse0[]={1.0, 0.0, 0.0, 1.0};
GL float ambient0[]={1.0, 0.0, 0.0, 1.0};
GL float specular0[]={1.0, 0.0, 0.0, 1.0};
Glfloat light0_pos[]={1.0, 2.0, 3,0, 1.0};

glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glLightv(GL_LIGHT0, GL_POSITION, light0_pos);
glLightv(GL_LIGHT0, GL_AMBIENT, ambient0);
glLightv(GL_LIGHT0, GL_DIFFUSE, diffuse0);
glLightv(GL_LIGHT0, GL_SPECULAR, specular0);
```

Ambient component: i_{amb}

- Ad-hoc – tries to account for light coming from other surfaces
- Just add a constant color:

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

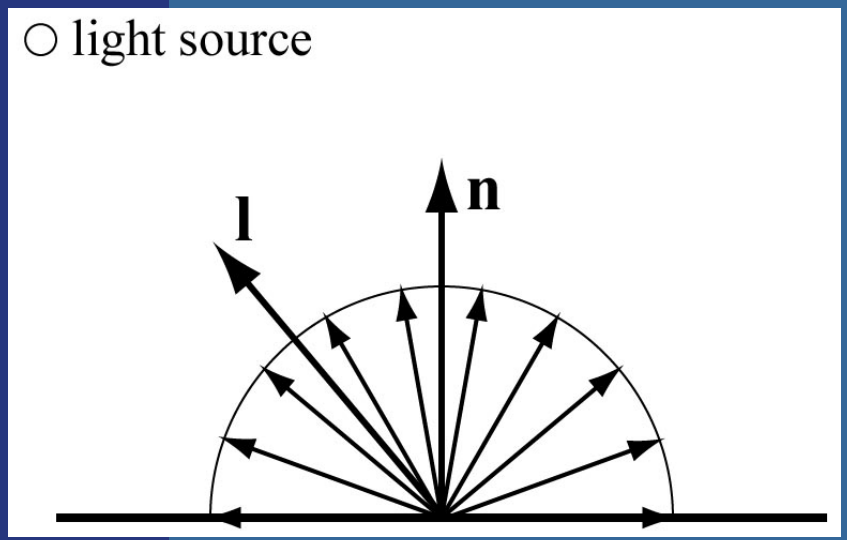
i.e., $(i_r, i_g, i_b, i_a) = (m_r, m_g, m_b, m_a)\ (l_r, l_g, l_b, l_a)$

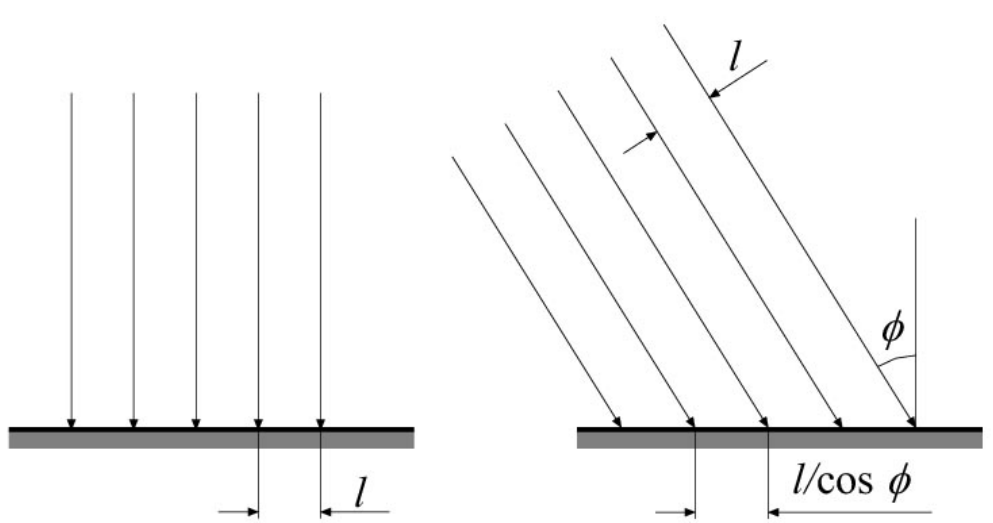
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, global_ambient)

Diffuse component : i_{diff}

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

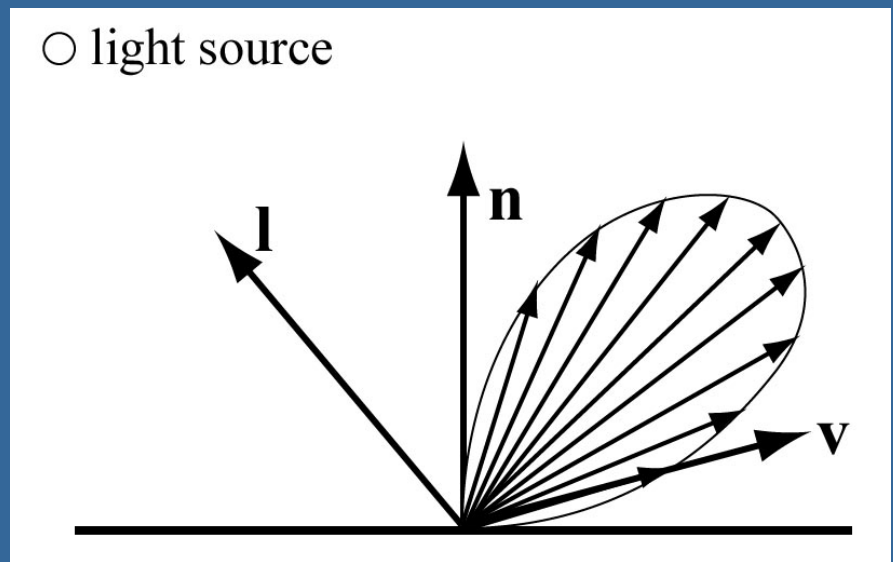
$$\mathbf{i}_{diff} = (\mathbf{n} \cdot \mathbf{l})\mathbf{m}_{diff} \otimes \mathbf{s}_{diff}$$





Lighting
Specular component : i_{spec}

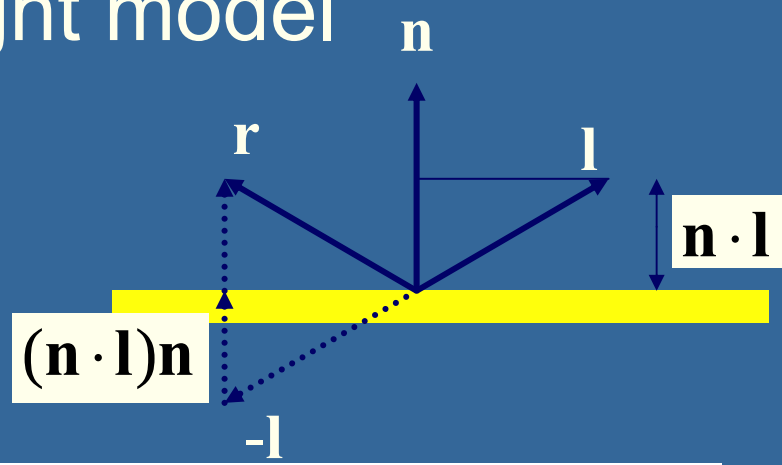

- Diffuse is dull (left)
- Specular: simulates a highlight



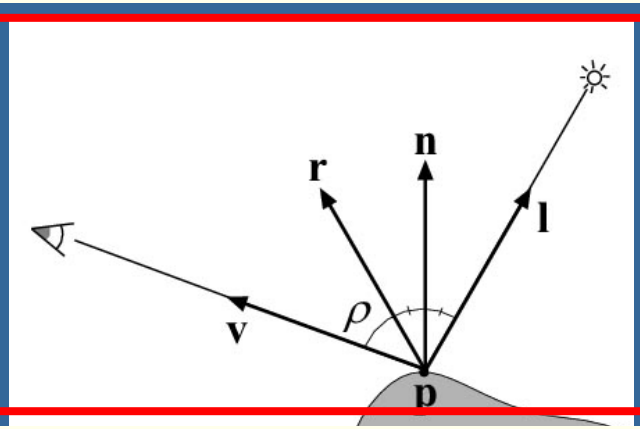
Specular component: Phong

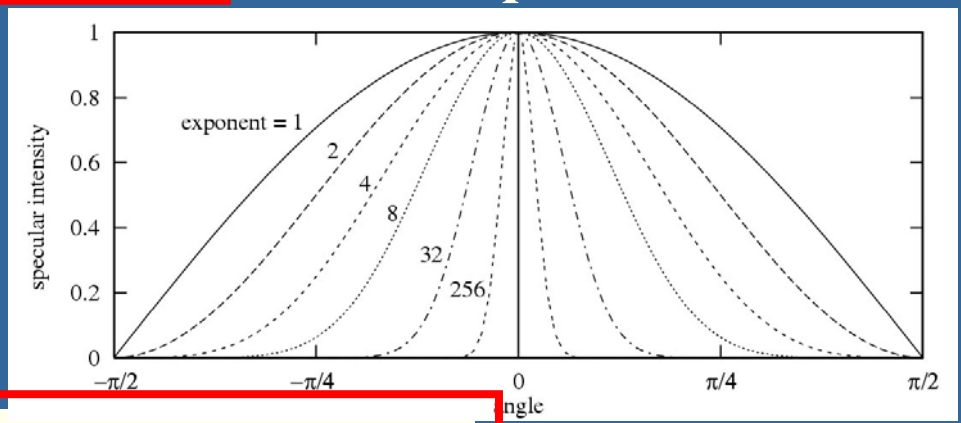
- Phong specular highlight model
- Reflect $\mathbf{l}$ around $\mathbf{n}$:

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



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





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

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

Halfway Vector

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

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

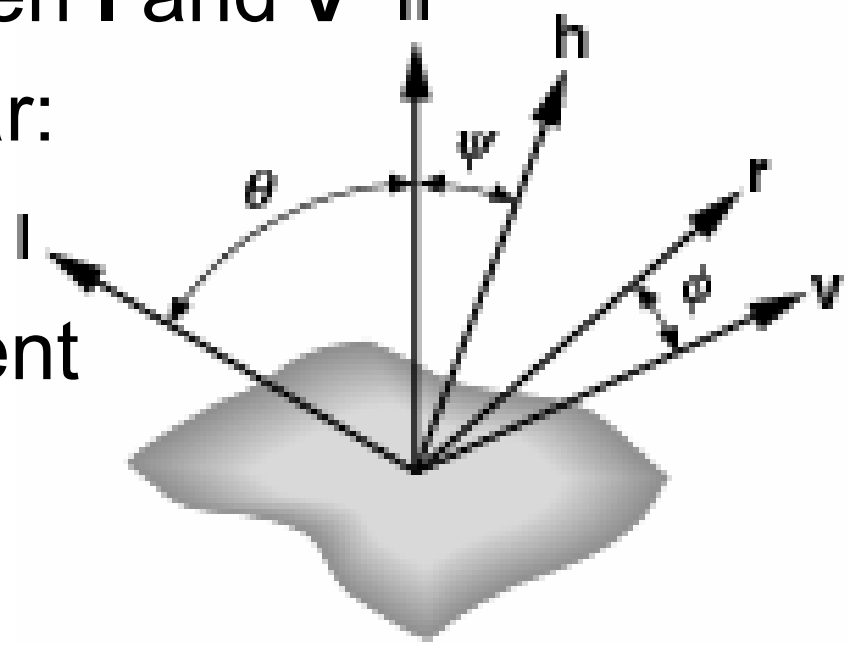
(**l**+**v**)/2 is halfway between **l** and **v**

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

$$\psi = \phi/2$$

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

$(e' \approx 4e)$



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

Lighting

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

D v s

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

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

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

Phongs reflektionsmodel:

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

Blinns reflektionsmodel:

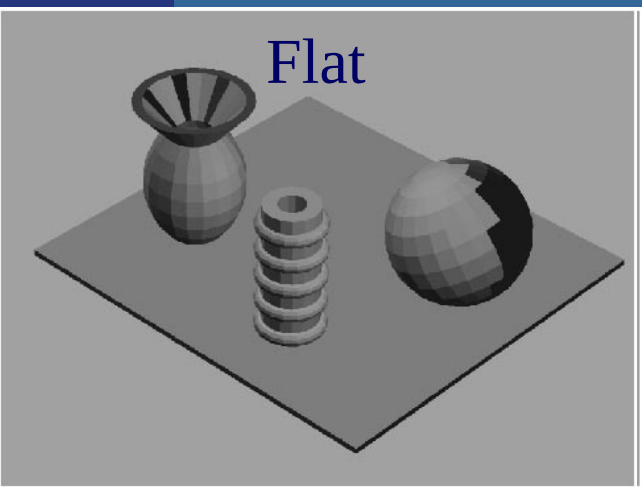
$$\mathbf{i}_{spec} = \max(0, (\mathbf{h} \cdot \mathbf{n})^{m_{shi}})\mathbf{m}_{spec} \otimes \mathbf{s}_{spec}$$

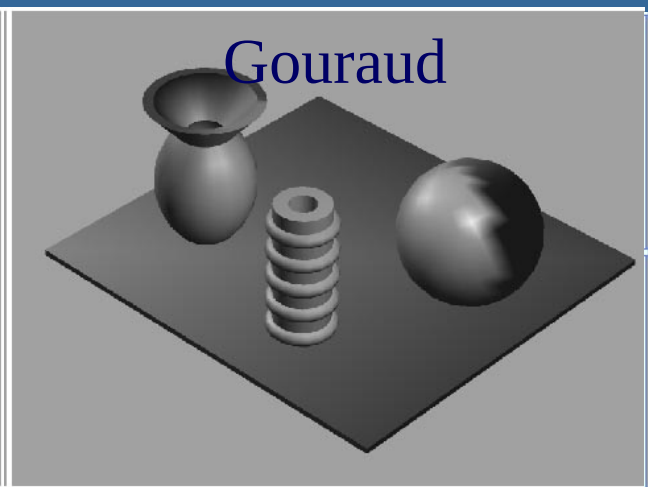
$$\mathbf{i}_{emission} = \mathbf{m}_{emission}$$

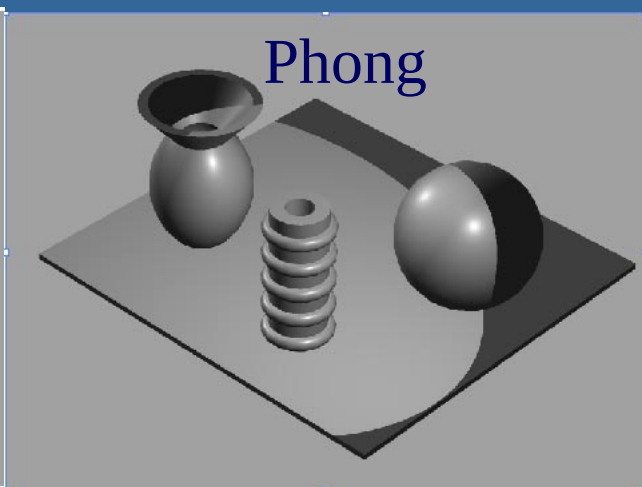
Strunta i Cook-Torrance shading

Shading

- Three common types of shading:
  - Flat, Goraud, and Phong
- I standard Gouraud shading beräknas belysning per triangelhörn och för varje pixel interpoleras färgen från triangelhörnen.
- Vid Phong Shading beräknas **inte** belysning per triangelhörn. För varje pixel interpoleras istället normalen fram från normalerna i triangelhörnen och man gör **full ljusberäkning per pixel med den interpolerade normalen**. Detta är såklart mycket dyrare, men ser mycket bättre ut.







Transparency and alpha

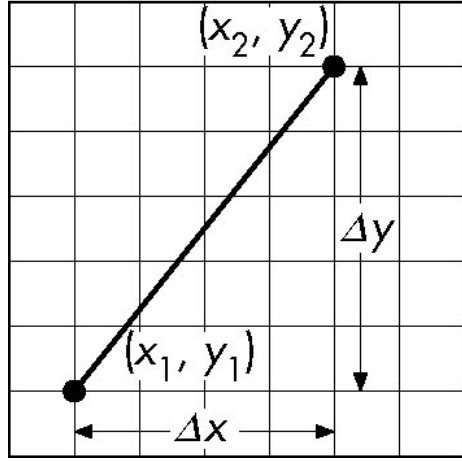
- Transparency
  - Very simple in real-time contexts
- The tool: alpha blending (mix two colors)
- Alpha (α) is another component in the frame buffer, or on triangle
  - Represents the opacity
  - 1.0 is totally opaque
  - 0.0 is totally transparent
- The over operator: $\mathbf{c}_o = \alpha\mathbf{c}_s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

Rendered object

Transparency

- Need to sort the transparent objects
  - Render back to front
  - **Man ritar först alla ogenomskinliga trianglar som vanligt.**
  - **Därefter sorterar man alla genomskinliga trianglar och renderar dem bakifrån-o-fram med blending påslaget (med djuptest som vanligt)**
  - **Dels för att slippa trixa med z-buffern och dels för att resultatet av blending ej är ordningsoberoende.**

DDA Algorithm



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

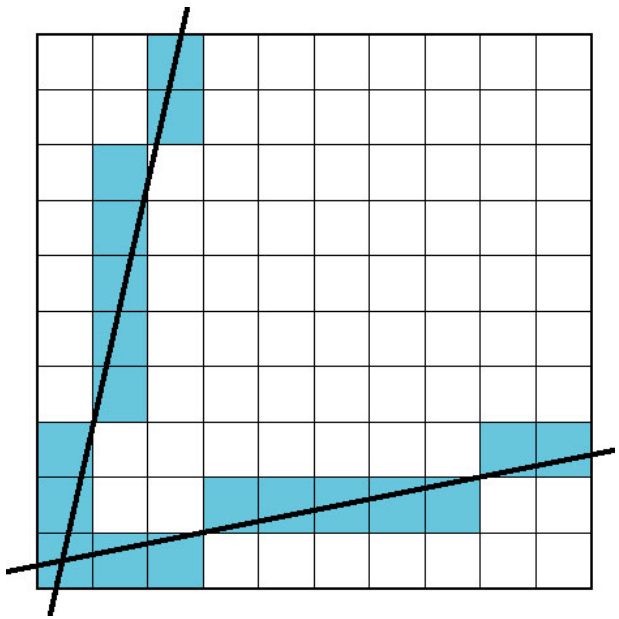
$$dy/dx = k = \Delta y/\Delta x = y_2\text{-}y_1/x_2\text{-}x_1$$

- Along scan line $\Delta x = 1$

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

Using Symmetry

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



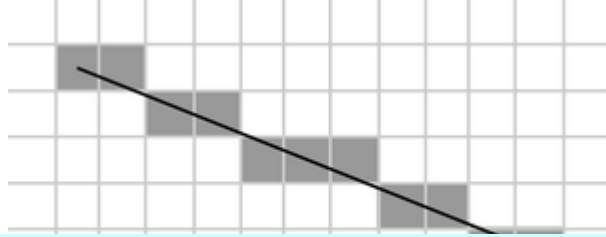
06. Rasterization, Depth Sorting and Culling:

Mycket viktigt!

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

Ni behöver inte kunna Bresenhams algoritm utantill. Det räcker att ni **förstår** följande 2 slides.

Bresenham's line drawing algorithm



- The line is drawn between two points (x_0, y_0) and (x_1, y_1)
- Slope $k = \dfrac{(y_1 - y_0)}{(x_1 - x_0)}$ $(y = kx + m)$

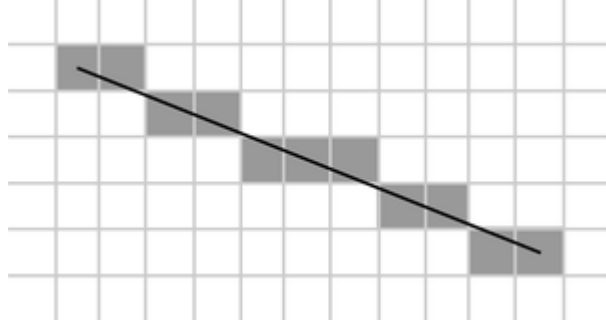
Detta är anledningen till att DDA behöver floats. Så Bresenhams lösning är att multiplicera alla relevanta tal med $(x_1\text{-}x_0)$ för att få integers.

- Each time we step 1 in x-direction, we should increment *y* with *k.* Otherwise the error in y increases with *k.*
- If the error surpasses 0.5, the line has become closer to the next *y*-value, so we add 1 to *y* simultaneously decreasing the error by 1

```
function line(x0, x1, y0, y1)
    int deltax := abs(x1 - x0)
    int deltay := abs(y1 - y0)
    real error := 0
    real deltaerr := deltay / deltax
    int y := y0
    for x from x0 to x1
        plot(x,y)
        error := error + deltaerr
        if error ≥ 0.5
            y := y + 1
            error := error - 1.0
```

See also
http://en.wikipedia.org/wiki/Bresenham's_line_algorithm

Bresenham's line drawing algorithm



- Now, convert algorithm to only using integer computations'
- The trick we use is to multiply all the fractional numbers above by (x_1, x_0), which enables us to express them as integers.
- The only problem remaining is the constant 0.5—to deal with this, we multiply both sides of the inequality by 2

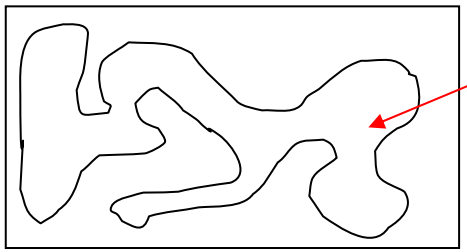
Old float version:

```
function line(x0, x1, y0, y1)
    int deltax := abs(x1 - x0)
    int deltay := abs(y1 - y0)
    real error := 0
    real deltaerr := deltay / deltax
    int y := y0
    for x from x0 to x1
        plot(x,y)
        error := error + deltaerr
        if error ≥ 0.5
            y := y + 1
            error := error - 1.0
```

New integer version:

```
function line(x0, x1, y0, y1)
    int deltax := abs(x1 - x0)
    int deltay := abs(y1 - y0)
    real error := 0
    real deltaerr := deltay
    int y := y0
    for x from x0 to x1
        plot(x,y)
        error := error + deltaerr
        if 2*error ≥ deltax
            y := y + 1
            error := error - deltax
```

Bresenhams alg.



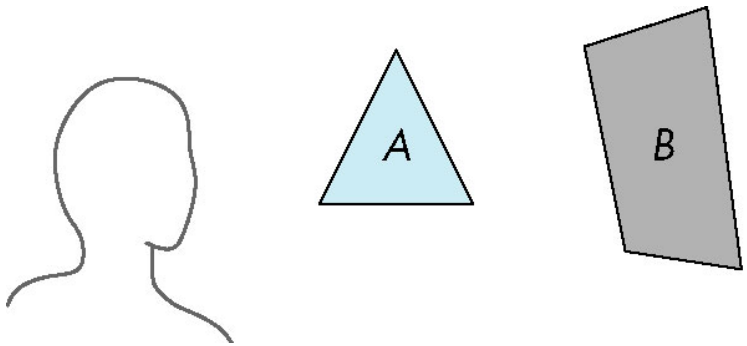
Flood Fill

- Fill can be done recursively if we know a seed point located inside (WHITE)
- Scan convert edges into buffer in edge/inside color (BLACK)

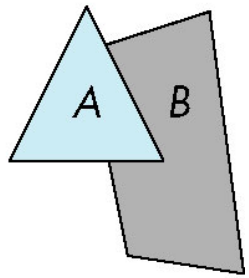
```
flood_fill(int x, int y) {
    if(read_pixel(x,y)= = WHITE) {
       write_pixel(x,y,BLACK);
       flood_fill(x-1, y);
       flood_fill(x+1, y);
       flood_fill(x, y+1);
       flood_fill(x, y-1);
}   }
```

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



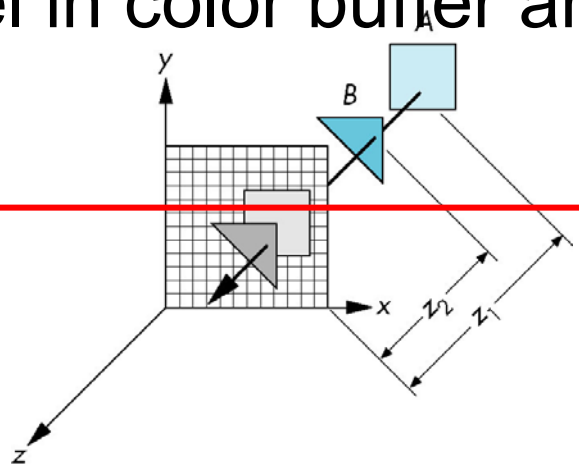
Fill B then A

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

D v s: Sortera alla trianglar och rendera dem bakifrån o fram.

z-Buffer Algorithm

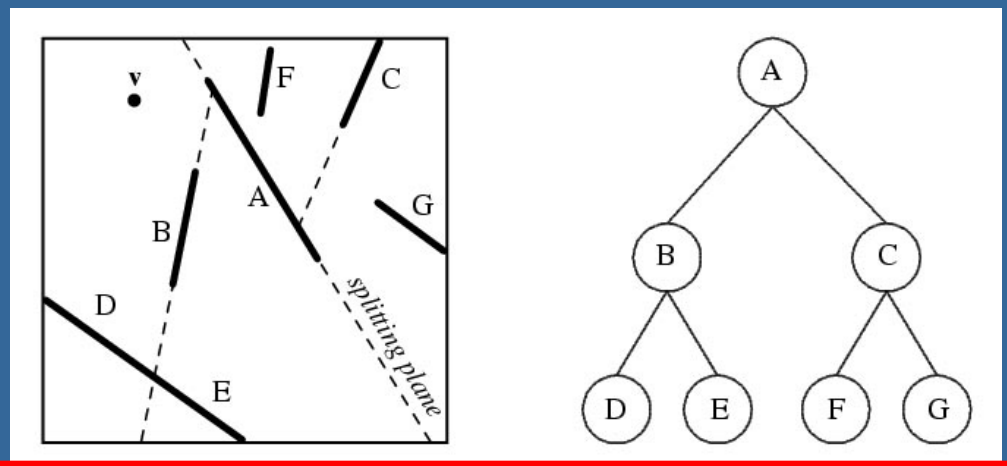
- Use a buffer called the z or depth buffer to store the depth of the closest object at each pixel found so far
- As we render each polygon, compare the depth of each pixel to depth in z buffer
- If less, place shade of pixel in color buffer and update z buffer



Polygon-aligned BSP tree

Mycket viktigt

- Used for visibility and occlusion/depth testing (BSP=Binary Space Partitioning)
- Allows exact sorting
  - Each node stores:
    - Polygon (triangle)
    - The splitting plane (defined by the triangle)
    - Front and back subtree



Algorithm for BSP trees

Mycket viktigt

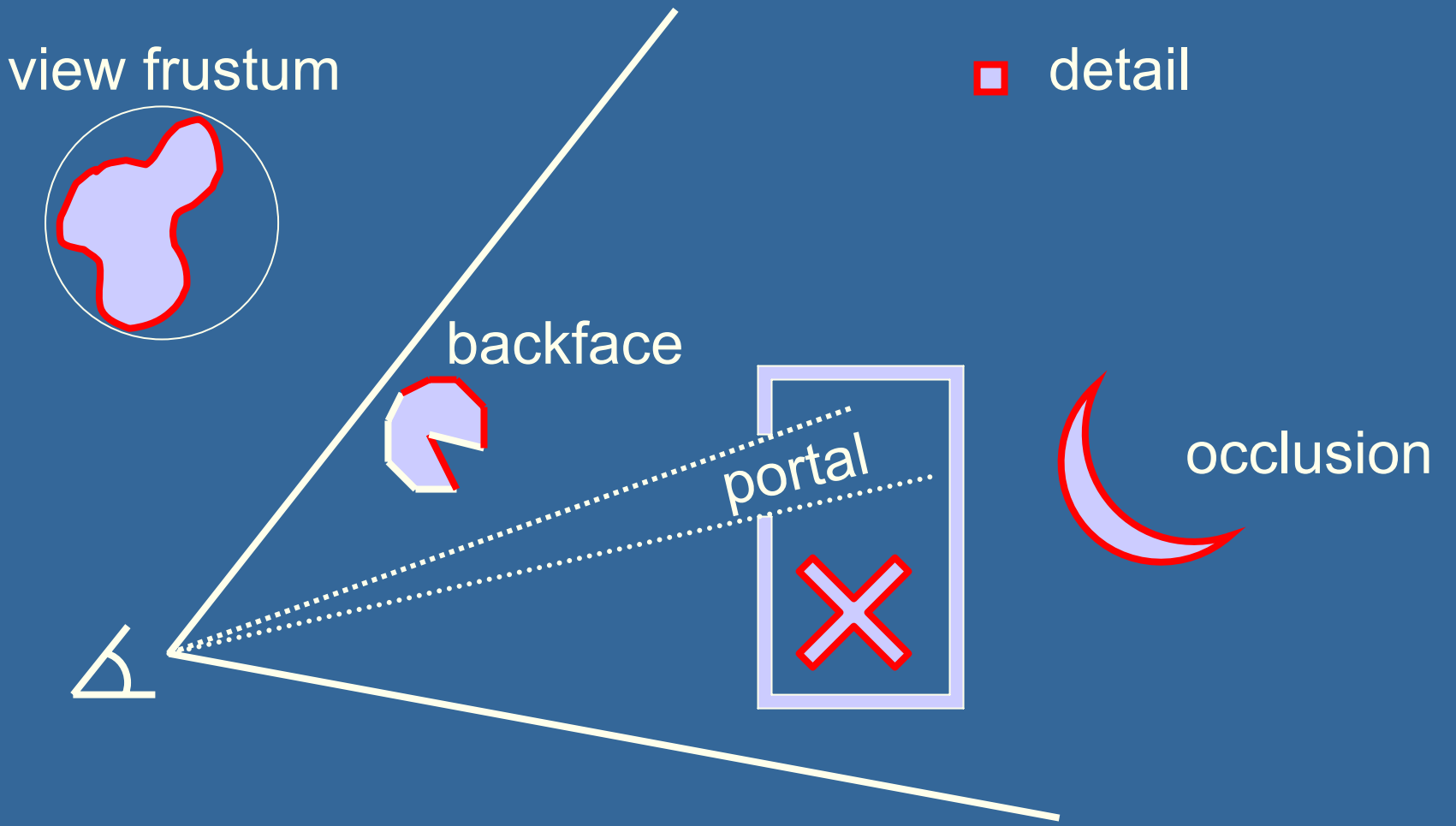
```
Tree SkapaBSP(PolygonLista L) {
    Om L tom returnera ett tomt träd;
    Annars: Välj en polygon P i listan.
    Bilda en lista B med de polygoner som ligger bakom
    P och en annan H med övriga. Returnera ett träd med
    P som rot och SkapaBSP(B) och SkapaBSP(H) som
    vänsterbarn respektive högerbarn.
}
```

Uppritningsteget (kolla även om trädet tomt! Fick ej plats i koden):

```
void RitaBSP(Tree t) {
    Om observatören hitom roten i t:
        RitaBSP(t:s vänsterbarn);
        Rita polygonen i t:s rot;
        RitaBSP(t:s högerbarn);
    Annars:
        RitaBSP(t:s högerbarn);
        Rita polygonen i t:s rot;
        RitaBSP(t:s vänsterbarn);
}
```

Different culling techniques

(red objects are skipped)



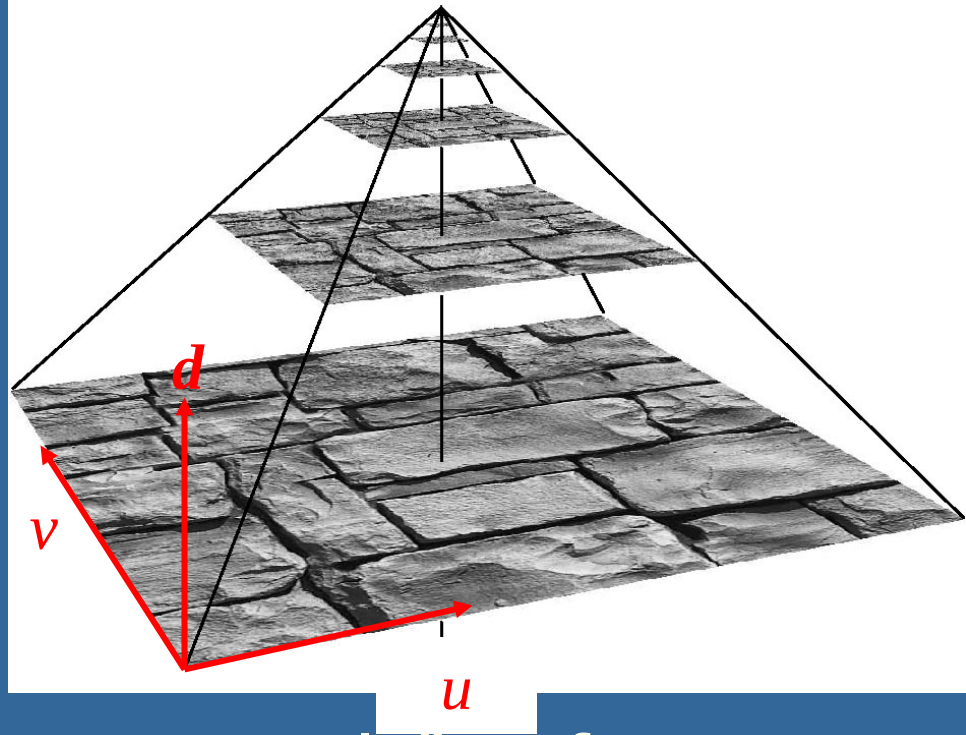
View Frustum Culling (VFC) och Backface culling är viktigt!

07. Texturing

Viktigast:

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

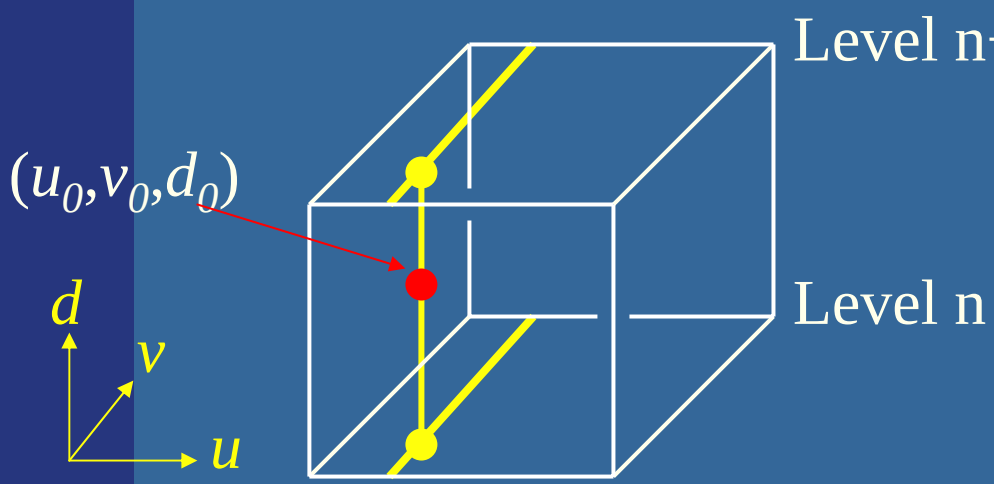
Mipmapping



- 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

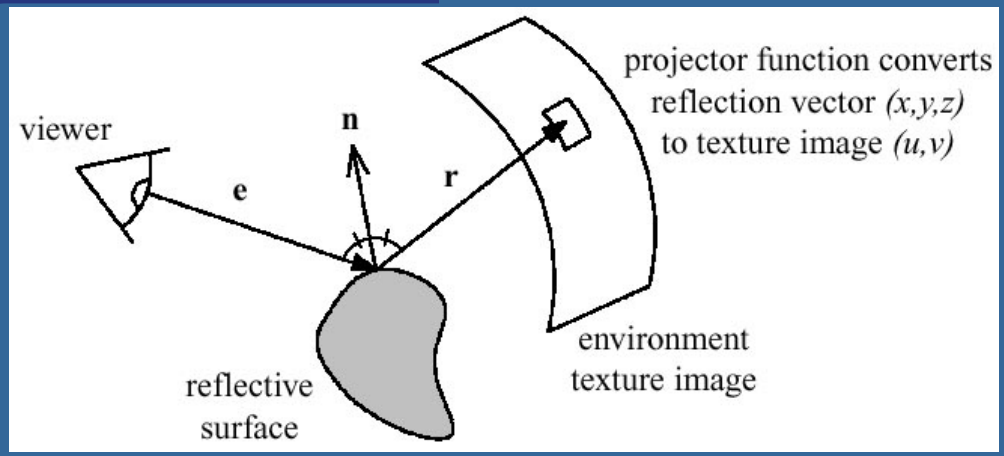
- Interpolate between those bilinear values
  - Gives trilinear interpolation



- Constant time filtering: 8 texel accesses

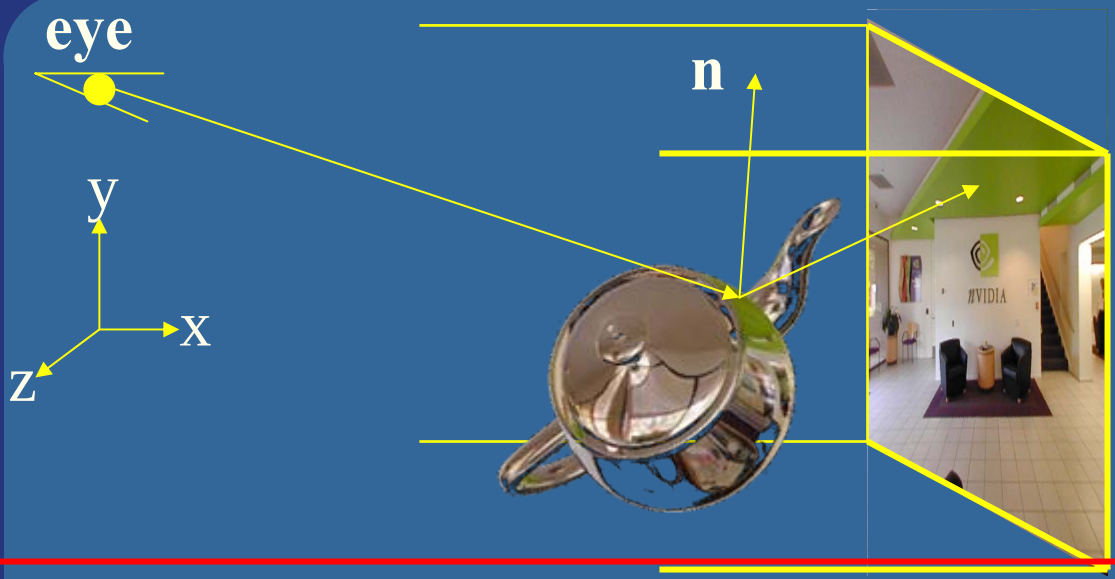
- How to compute d?

Environment mapping



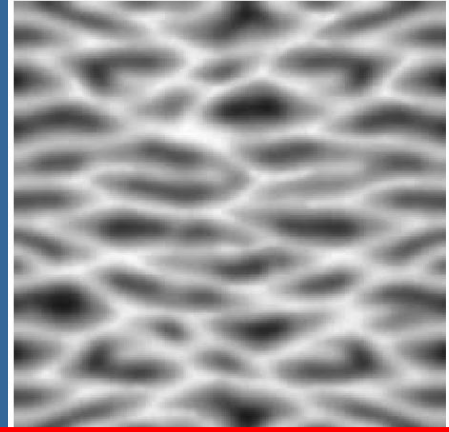
- Assumes the environment is infinitely far away
- Sphere mapping
  - For details, see OH 166-169
- Cube mapping is the norm nowadays
  - Advantages: no singularities as in sphere map
  - Much less distortion
  - Gives better result
  - Not dependent on a view position

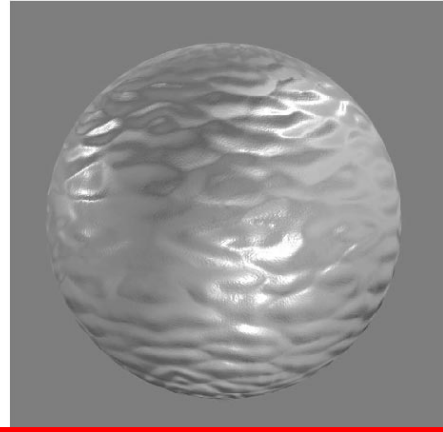
Cube mapping



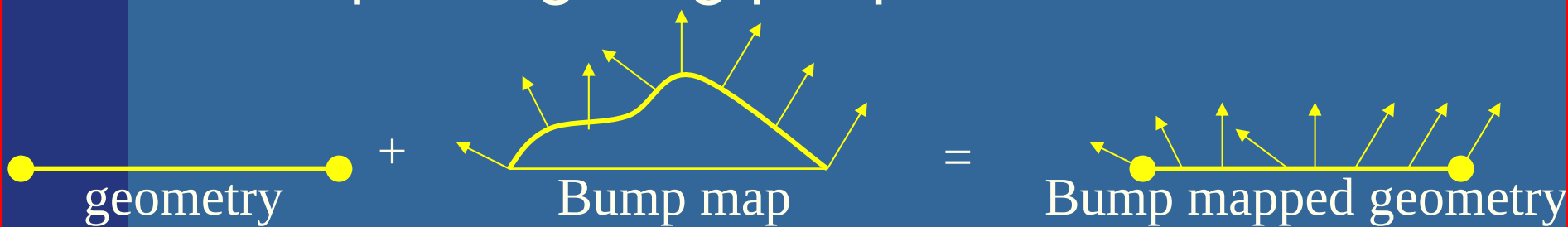
- Simple math: compute reflection vector, **r**
- Largest abs-value of component, determines which cube face.
  - Example: **r**=(5,-1,2) gives POS_X face
- Divide **r** by 5 gives (*u*,*v*)=(-1/5,2/5)
- If your hardware has this feature, then it does all the work

Bump mapping





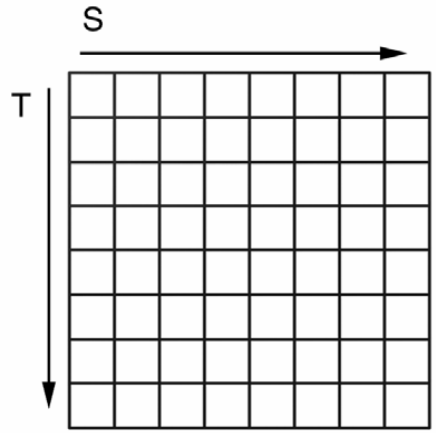
- by Blinn in 1978
- Inexpensive way of simulating wrinkles and bumps on geometry
  - Too expensive to model these geometrically
- Instead let a texture modify the normal at each pixel, and then use this normal to compute lighting per pixel

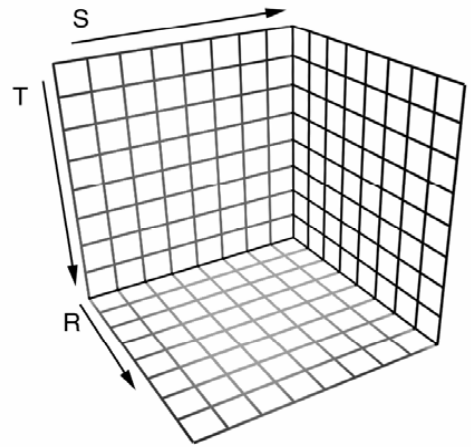


Stores heights: can derive normals

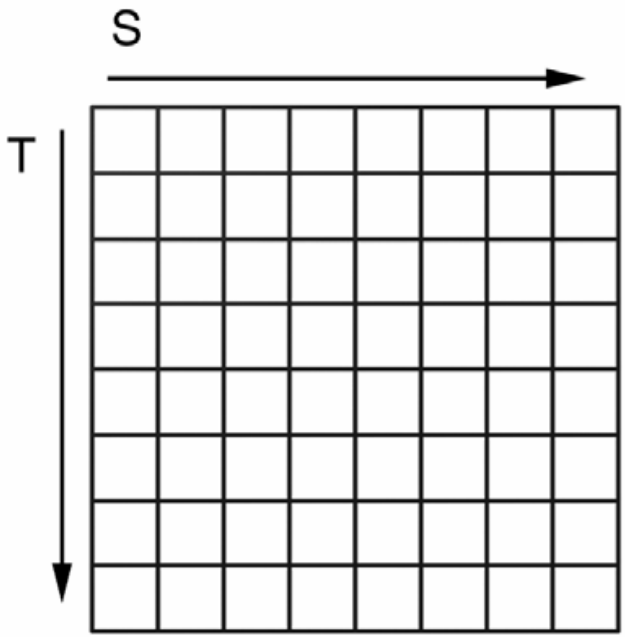
3D Textures

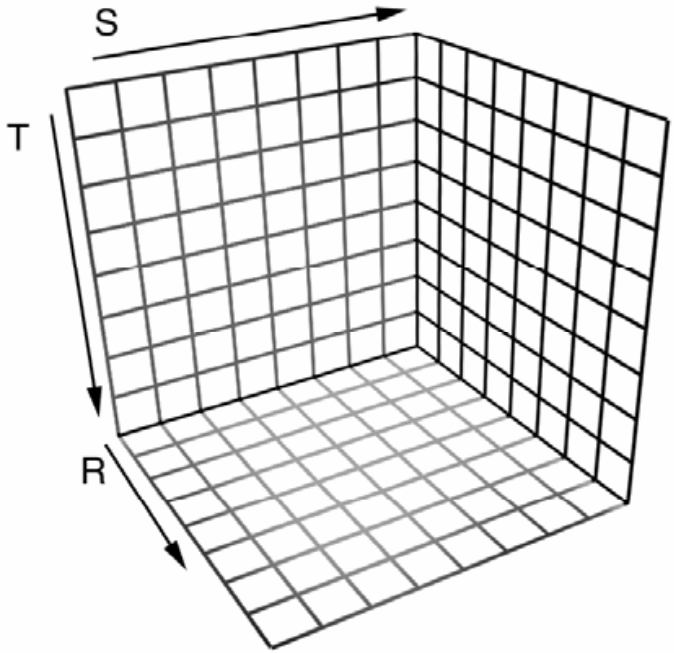
- 3D textures:
  - Feasible on modern hardware as well
  - Texture filtering is no longer trilinear
  - Rather quadlinear (linear interpolation 4 times)
  - Enables new possibilities
    - Can store light in a room, for example





2D texture vs 3D texture





Sprites

Läs OH284-291

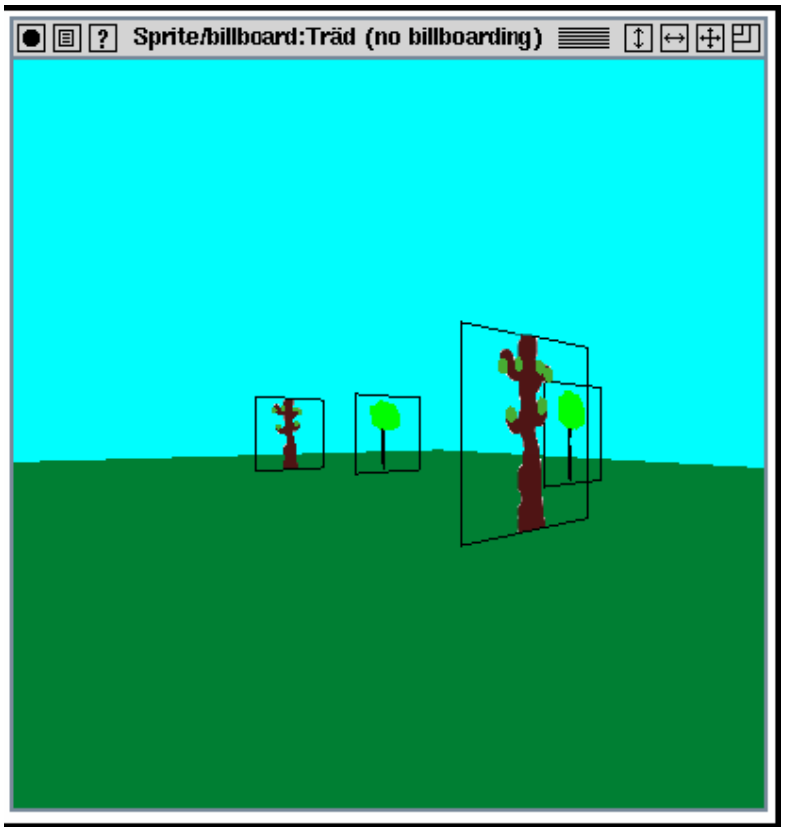
Sprites (=älvor) var en teknik på speldatorer, som t ex VIC64. Till skillnad från billboards använder sprites ej framebuffern. De rastreras direkt mot TV-skärmen. Ett speciellt bit-register markerade vilka sprites som kolliderade med varandra på skärmen.

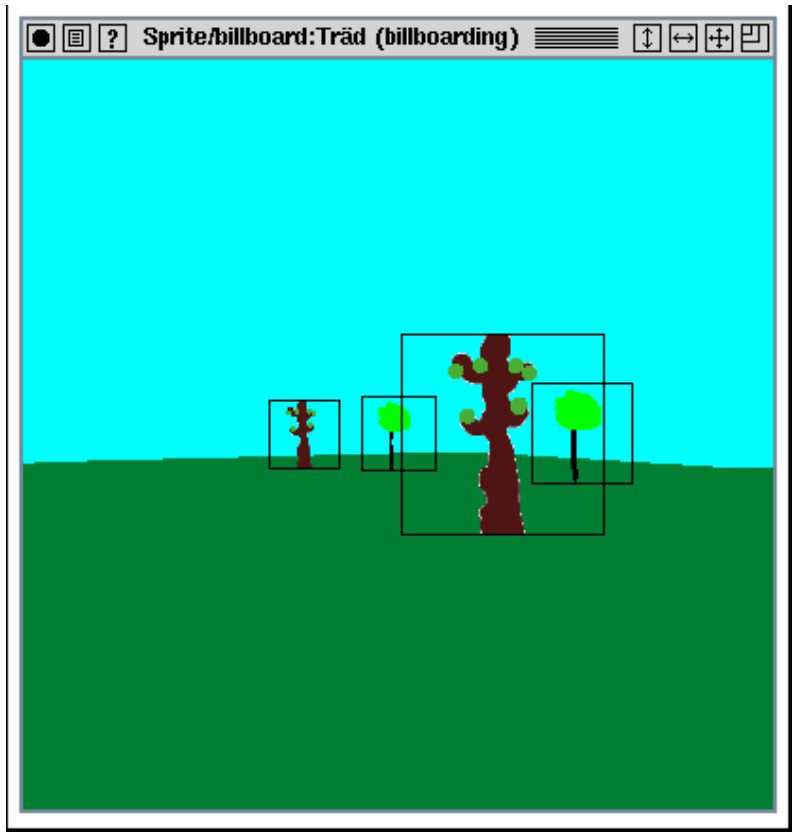
```
GLbyte M[64]=
{   127,0,0,127, 127,0,0,127,
    127,0,0,127, 127,0,0,127,
    0,127,0,0, 0,127,0,127,
    0,127,0,127, 0,127,0,0,
    0,0,127,0, 0,0,127,127,
    0,0,127,127, 0,0,127,0,
    127,127,0,0, 127,127,0,127,
    127,127,0,127, 127,127,0,0};

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



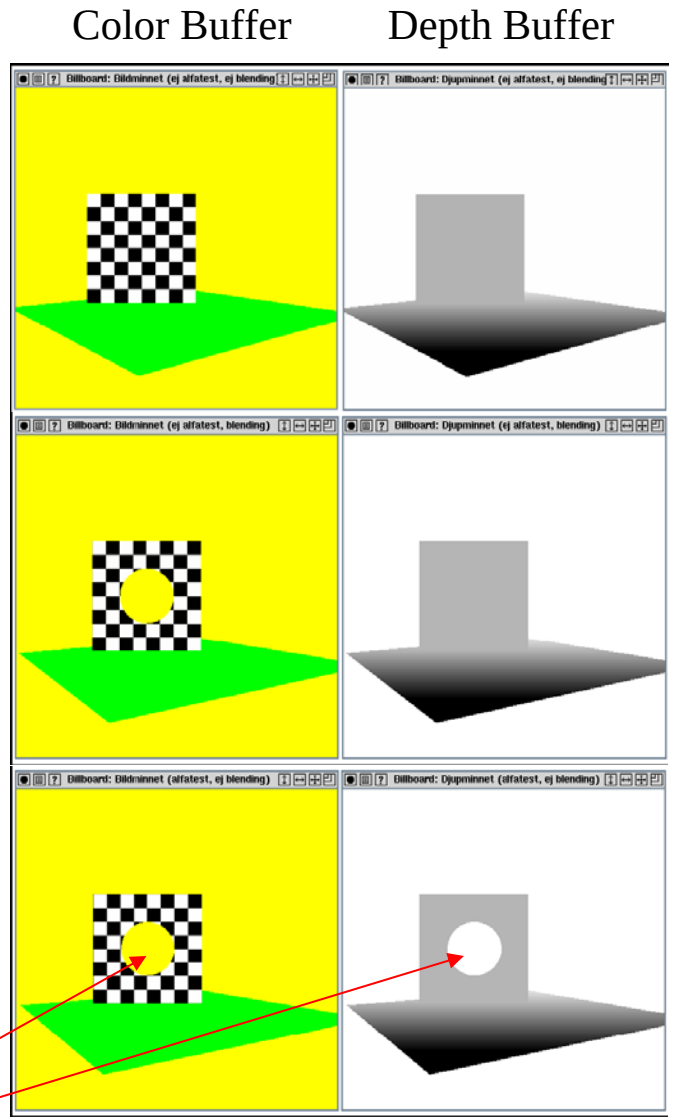


- Rotate them towards viewer
  - Either by rotation matrix (see OH 288), or
  - by orthographic projection

Billboards

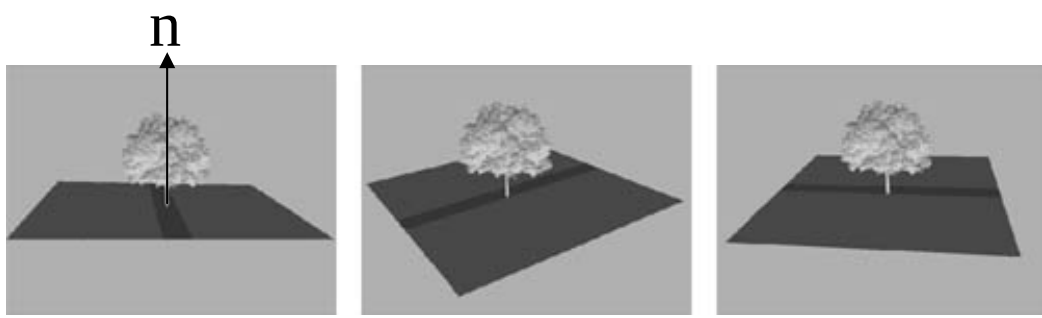
- Fix correct transparency by blending AND using alpha-test
  - glEnable(GL_ALPHA_TEST);
  - glAlphaFunc(GL_GREATER, 0.1);
- Se OH 289-291

Tröskelvärde > 0. Om alfavärdet i texturen är lägre än detta tröskelvärde renderas inte pixeln till framebuffern och z-buffern uppdateras ej. Vilket är vad man vill åstadkomma. T ex här:



07. Texturing:

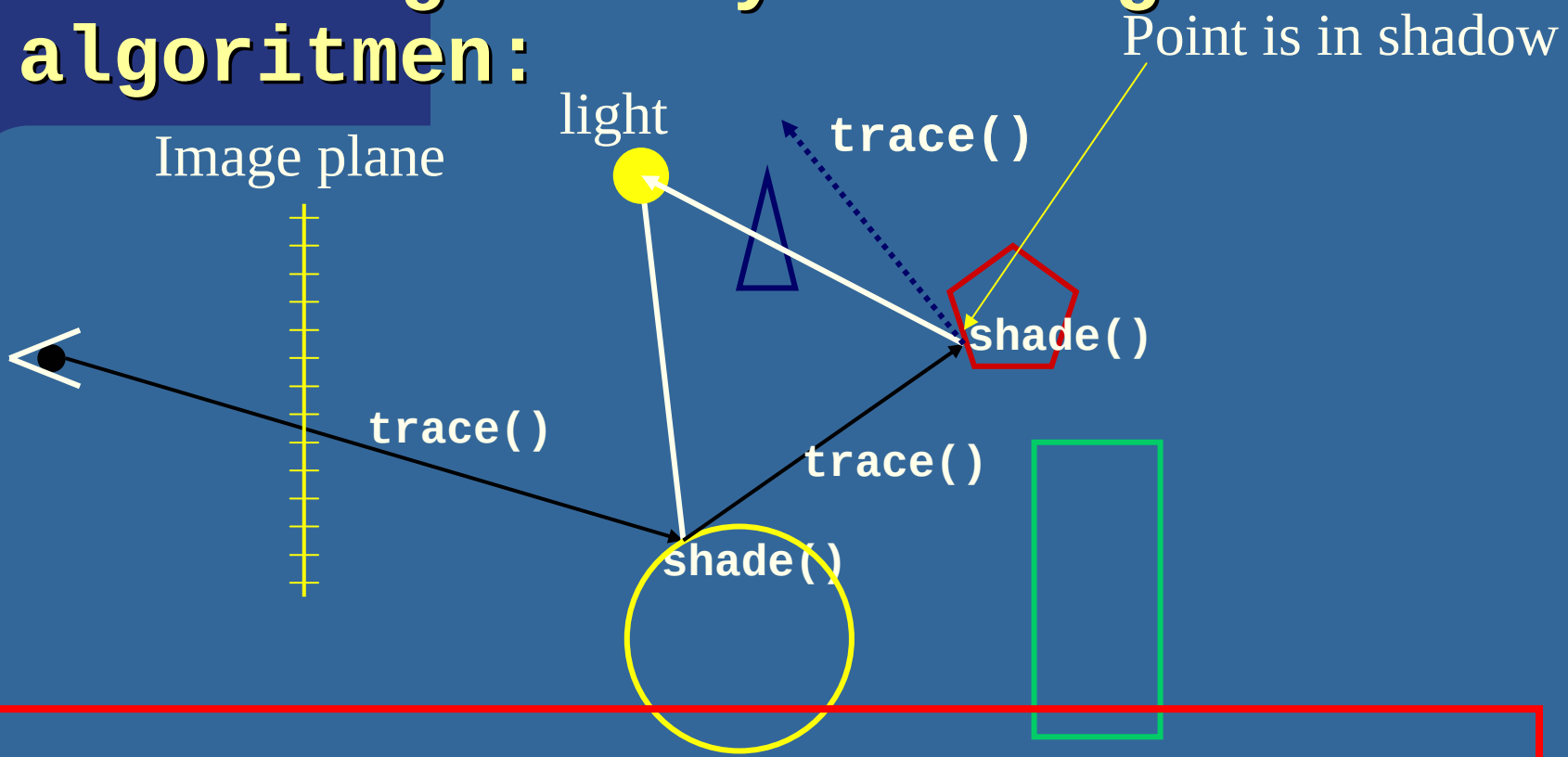
Also called *Impostors*



axial billboarding

The rotation axis is fixed and disregarding the view position

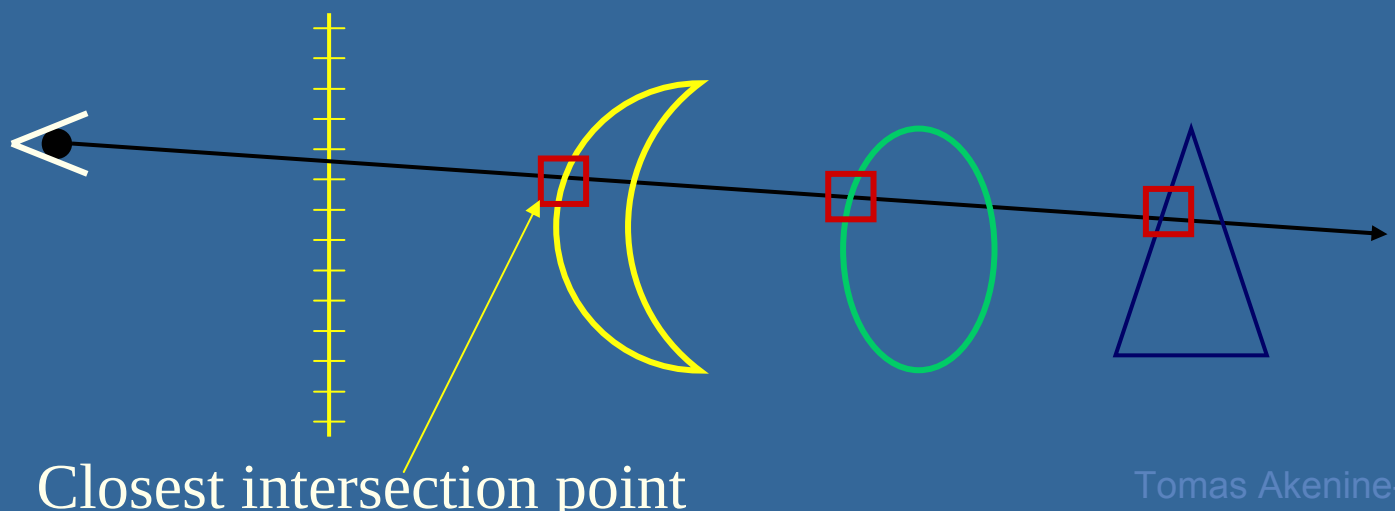
Summering av Ray tracing-algoritmen:



- main()-anropar trace() för varje pixel
- trace(): Har som uppgift att returnera färg hos första träffade punkt.
  1. anropar findClosestIntersection()
  2. Om något objekt träffades → anropa shade().
- Shade(): Har som uppgift att beräkna färgen i punkten
  1. För varje ljuskälla skjut shadow ray för att avgöra om ljuskällan syns
     Om ej i skugga, beräkna diffust + spekulärt ljusbidrag.
  2. Beräkna ambient ljusbidrag
  3. Anropa trace() rekursivt för reflektions- och refraktionsstråle.

Follow photons backwards from the eye: treat one pixel at a time

- Rationale: find photons that arrive at each pixel
- How do one find the visible object at a pixel?
- With intersection testing
  - Ray, $\mathbf{r}(t)=\mathbf{o}+t\mathbf{d}$, against geometrical objects
  - Use object that is closest to camera!
  - Valid intersections have $t > 0$
  - t is a signed distance



Finding closest point of intersection

- Naively: test all geometrical objects in the scene against each ray, use closest point
  - Very very slow!
- Be smarter:
  - Use spatial data structures, e.g.:
  - Bounding volume hierarchies
  - Octrees
  - BSP trees
  - Grids (not yet treated)
  - Or a combination (hierarchies) of those
- See Advanced Computer Graphics, EDA425, for more

`trace()` and `shade()`

- We now know how to find the visible object at a pixel
- How about the finding the color of the pixel?
- Basic ray tracing is essentially only two functions that recursively call each other
  - **`trace()`** and **`shade()`**
- **`trace()`**: finds the first intersection with an object and calls shade() for the hit point
- **`shade()`**: computes the lighting at that intersection point

trace() in detail

```
Color trace(Ray R)
{
      float t;    bool hit;
      Object O;
      Color col;
      Vector P,N; // point & normal at intersection point
      hit=findClosestIntersection(R,&t,&O);
      if(hit)
      {
            P=R.origin() + t*R.direction();
            N=computeNormal(P,O);
            // flip normal if pointing in wrong dir.
            if(dot(N,R.direction()) > 0.0) N=-N;
            col=shade(t,O,R,P,N);
      }
      else col=background_color;
      return col;
}
```

shade( ) computes lighting

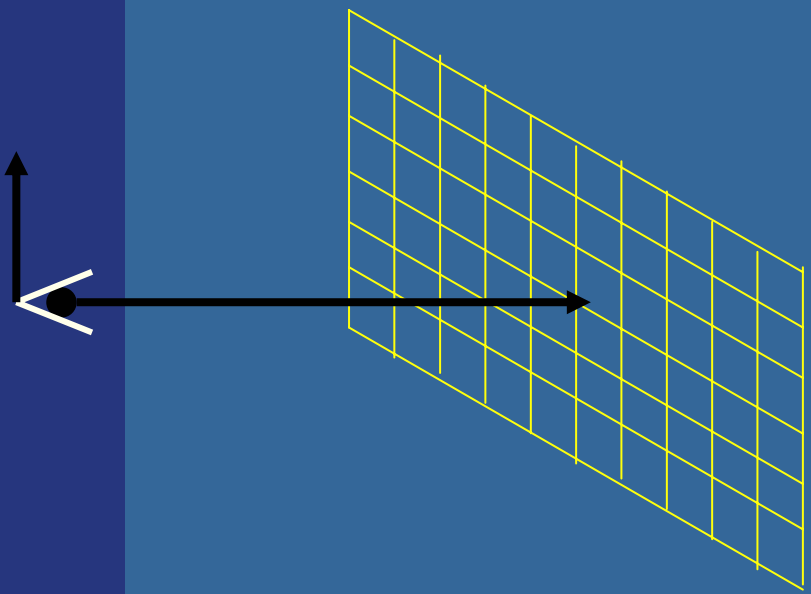
- For now, we will use the simple standard lighting equation that we used so far
- Ambient+Diffuse+Specular
- However, we also spawn new rays in the:
  - Reflection and
  - Refraction direction
- Can use more advanced models
  - Simple to exchange --- a strength of ray tracing

shade( ) in detail

```
Color shade(Ray R, Mtrl &m, Vector P,N)
{
      Color col;
      Vector N,P,refl,refr;
      for each light L
      {
            if(not inShadow(L,P))
                  col+=DiffuseAndSpecular();
      }
      col+=AmbientTerm();
      if(recursed_too_many_times()) return col;
      refl=reflectionVector(R,N);
      col+=m.specular_color()*trace(refl);
      refr=computeRefractionVector(R,N,m);
      col+=m.transmission_color()*trace(refr);
      return col;
}
```

Who calls `trace()` or `shade()`?

- Someone need to spawn rays
  - One or more per pixel
  - A simple routine, **`raytraceImage()`**, computes rays, and calls **`trace()`** for each pixel.



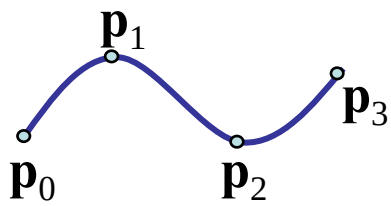
- Use camera parameters to compute rays
  - Resolution, fov, camera direction & position & up

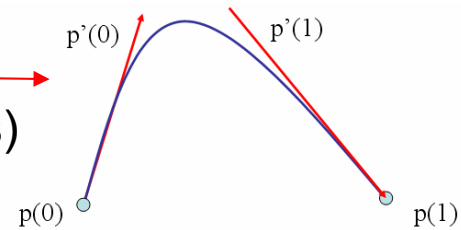
When does recursion stop?

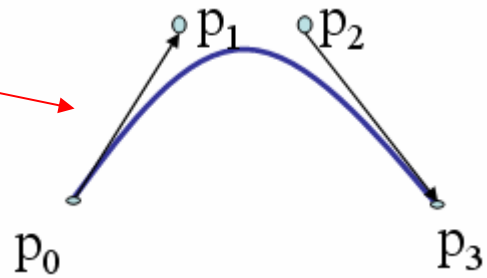
- Recurse until ray does not hit something?
  - Does not work for closed models
- One solution is to allow for max N levels of recursion
  - N=3 is often sufficient (sometimes 10 is sufficient)
- Another is to look at material parameters
  - E.g., if specular material color is (0,0,0), then the object is not reflective, and we don't need to spawn a reflection ray
  - More systematic: send a weight, w, with recursion
  - Initially w=1, and after each bounce, w*=O.specular_color(); and so on.
  - Will give faster rendering, if we terminate recursion when weight is too small (say <0.05)

Objectives

- 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
    - To get C^2 continuity
  - NURBS
    - Different weights of the control points







Splines and Basis

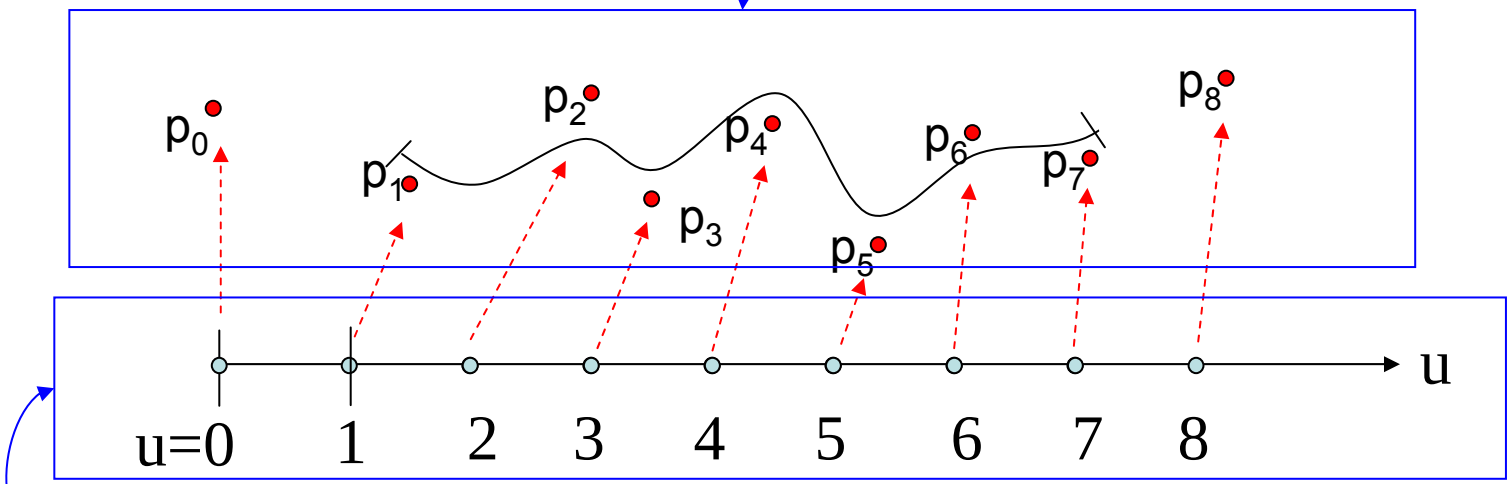
- If we examine the cubic B-spline from the perspective of each control (data) point, each interior point contributes (through the blending functions) to four segments
- We can rewrite p(u) in terms of the data points as

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

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

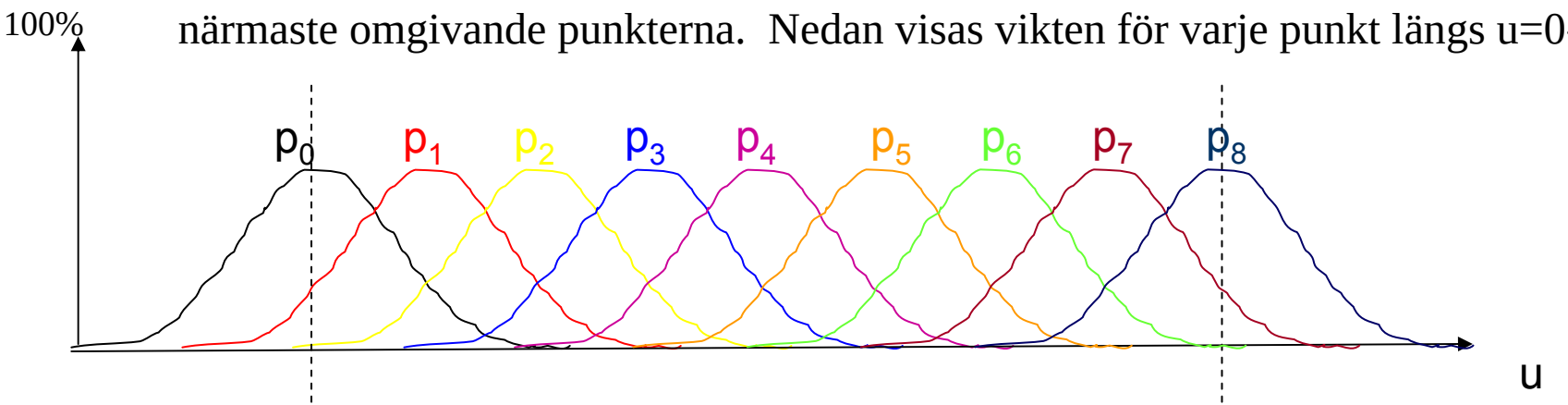
B-Splines

Detta är våra kontrollpunkter, p_0-p_8, till vilka vi vill approximera en kurva



Detta visar hur kontrollpunkterna ligger fördelade (jämnt) utefter parametriseringen u av kurvan p(u).

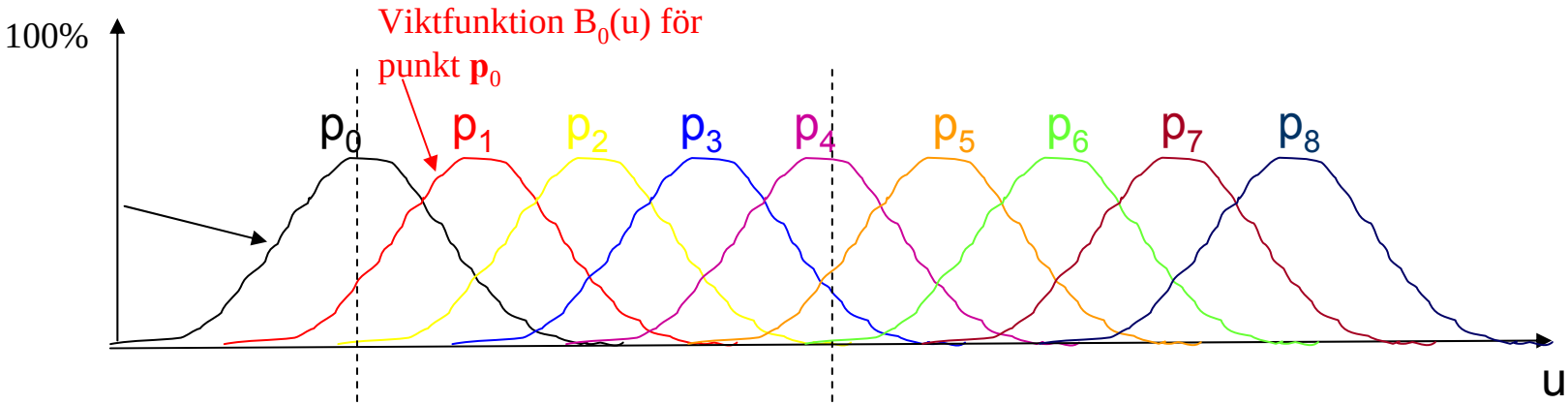
I varje punkt p(u) på kurvan, för givet u, är punkten definierad som en vikt av de 4 närmaste omgivande punkterna. Nedan visas vikten för varje punkt längs u=0→1



09. Curves and Surfaces:

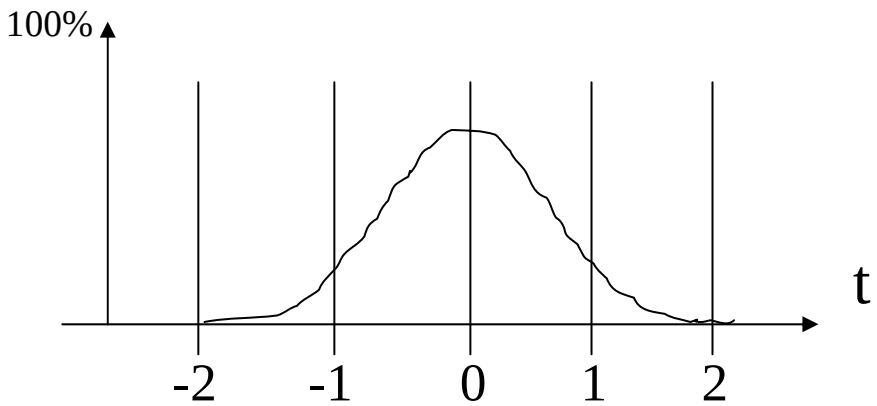
B-Splines

I varje punkt p(u) på kurvan, för givet u, är punkten definierad som en vikt av de 4 närmaste omgivande punkterna. Viktsumman =1 överallt längs u=0→1. Nedan visas vikten för varje punkt längs u=0→1



Viktfunktionen $B_{\mathbf{p}i}$ (u) för en punkt $\mathbf{p}_i$ kan alltså skrivas som en translation av en basfunktion B(t). $B_{\mathbf{p}i}(u) = B(u-i)$

B(t):



Hela vår B-splinekurva p(u) kan alltså skrivas:

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

NURBS

NURBS är samma sak som B-Splines förutom att:

1. Kontrollpunkterna kan ha olika vikt (högre vikt gör att kurvan måste gå närmare kontrollpunkten)
2. Kontrollpunkterna behöver inte ligga på jämnt avstånd (u=0,1,2,3...) över parametriseringen u. T ex kan de ligga på: u=0,0.5, 0.9, 4, 14,

NURBS = Non-Uniform Rational B-Splines

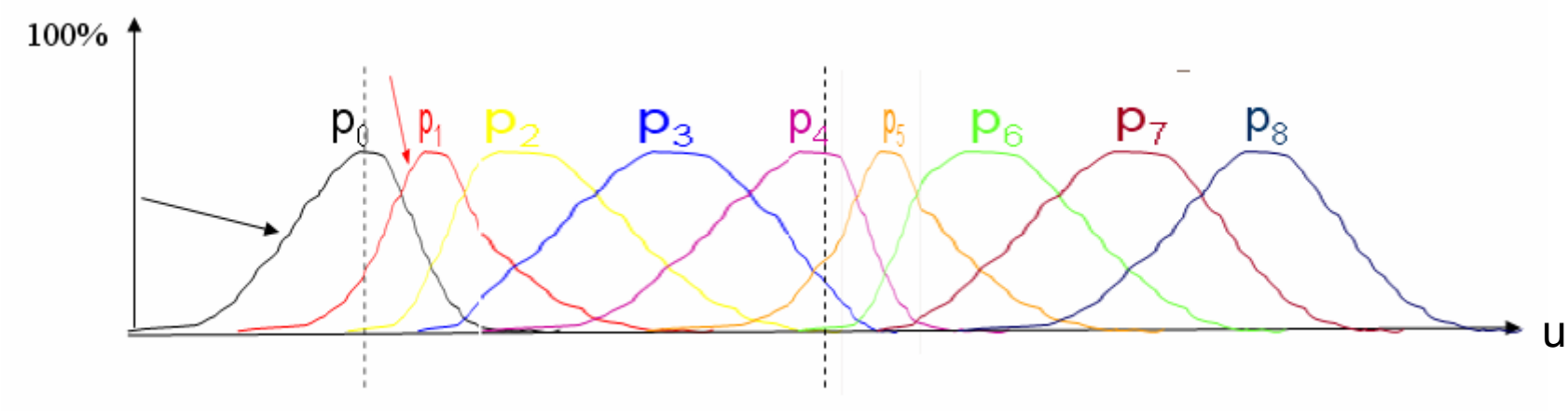
NURBS-kurvan definieras alltså nu som:

$$\boldsymbol{P}_i(t) = \frac{\sum_{j=i-1}^{i+2} w_j B_j(t) \boldsymbol{P}_j}{\sum_{j=i-1}^{i+2} w_j B_j(t)}$$

Man dividerar med summan av vikterna för att genomsnittsvikten hos punkterna skall bli 1. Annars inför man nämligen en förskjutning av kurvan.

NURBS

- Att kontrollpunkterna kan ligga på ojämnt avstånd innebär bara att basfunktionerna $B_{pi}()$ blir stretchade och ligger ojämnt placerade.
- Ex:



Varje kurva $B_{pi}()$ skall förstås föreställa mjuk och C^2 –kontinuerlig. Men det är inte så lätt att rita...

(Viktsumman blir fortfarande =1 pga divisionen i föregående slide)

•10. Vertex and Fragment Shaders:

Vertex- and Fragment shaders

- Endast kunna *förstå* de shaders som är på följande slides. De är tagna från slide 53-60 ur de föreläsningsslides som ligger på hemsidan under lektion 10-Vertex and Fragment Shaders. (Ni skall absolut ej kunna dessa utantill på något sätt.)

•10. Vertex and Fragment Shaders:

Wave Motion Vertex Shader

```
uniform float time;
uniform float xs, zs;
void main()
{
float s;
s = 1.0 + 0.1*sin(xs*time)*sin(zs*time);
gl_Vertex.y = s*gl_Vertex.y;
gl_Position =
   gl_ModelViewProjectionMatrix*gl_Vertex;
}
```

•10. Vertex and Fragment Shaders:

Particle System

```
uniform vec3 init_vel;
uniform float g, m, t;
void main()
{
     vec3 object_pos;
     object_pos.x = gl_Vertex.x + vel.x*t;
     object_pos.y = gl_Vertex.y + vel.y*t
       - g/(2.0*m)*t*t;
     object_pos.z = gl_Vertex.z + vel.z*t;
     gl_Position =
          gl_ModelViewProjectionMatrix*
          vec4(object_pos,1);
}
```

•10. Vertex and Fragment Shaders:

Vertex Shader for per Fragment Lighting

```
/* vertex shader for per-fragment Phong shading */
varying vec3 normale;
varying vec4 positione;
void main()
{
   normale = gl_NormalMatrixMatrix*gl_Normal;
   positione = gl_ModelViewMatrix*gl_Vertex;
   gl_Position = gl_ModelViewProjectionMatrix*gl_Vertex;
}
```

•10. Vertex and Fragment Shaders:

Fragment Shader for per Fragment Lighting

```
varying vec3 normale;
varying vec4 positione;
void main()
{
vec3 norm = normalize(normale);
vec3 lightv = normalize(gl_LightSource[0].position-positione.xyz);
vec3 viewv = normalize(positione);
vec3 halfv = normalize(lightv + viewv);
vec4 diffuse = max(0, dot(lightv, viewv))
   *gl_FrontMaterial.diffuse*gl_LightSource[0].diffuse;
vec4 ambient = gl_FrontMaterial.ambient*gl_LightSource[0].ambient;
```

•10. Vertex and Fragment Shaders:

Fragment Shader for per Fragment Lighting

```
int f;
if(dot(lightv, viewv)> 0.0) f =1.0);
else f = 0.0;
vec3 specular = f*pow(max(0, dot(norm, halfv),
        gl_FrontMaterial.shininess)
       *gl_FrontMaterial.specular*gl_LightSource[0].specular);
vec3 color = vec3(ambient + diffuse + specular);
gl_FragColor = vec4(color, 1.0);
}
```

Shadows and Reflections

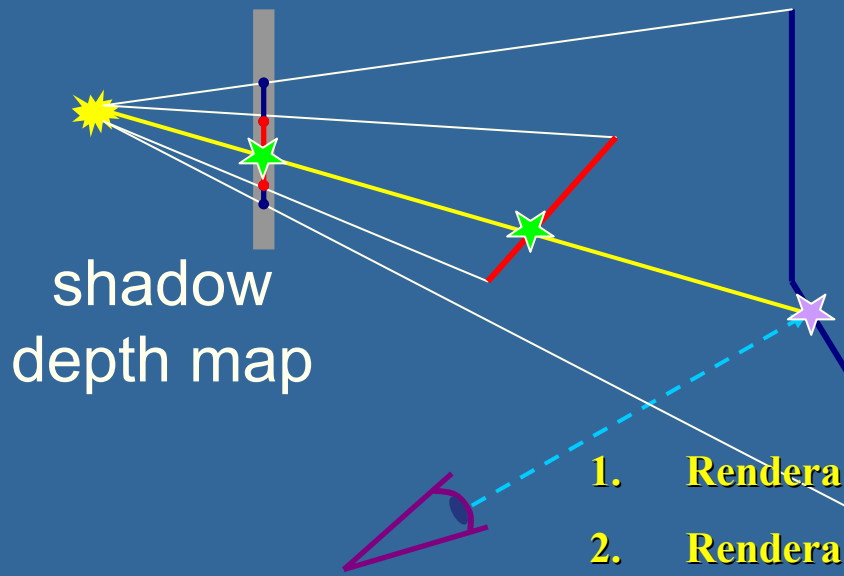
- Kunna algoritmen för
  - Shadow Mapping
  - Shadow Volumes
  - Plana reflektioner (reflektion i xy-planet genom negativ skalning i z-led)

Ways of thinking about shadows

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

Using the Shadow Map

- When scene is viewed, check viewed location in light's shadow buffer
  - If point's depth is (epsilon) greater than shadow depth, object is in shadow.



shadow depth map

For each pixel, compare distance to light ✦ with the depth ✦ stored in the shadow map

1. **Rendera en shadow depth map från ljuskällan.**
2. **Rendera från ögat. För varje genererad pixel warpar man x,y,z-koordinaten till light space och jämför djupet med det lagrade värdet i shadow mappen.**

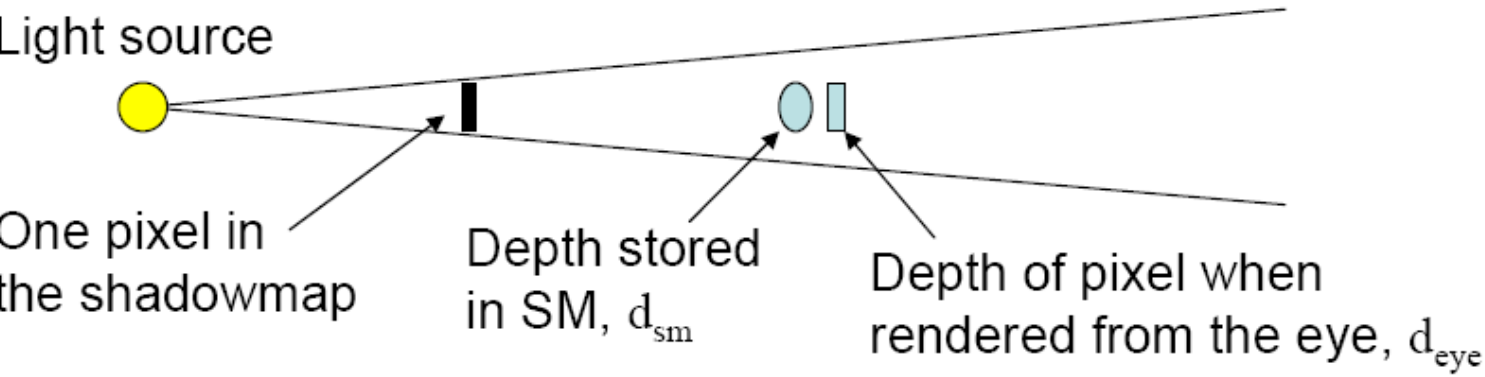
Om större → punkten ligger i skugga

Annars → punkten ligger ej i skugga

Problem: bias, offset behövs

Shadow mapping problems (2)

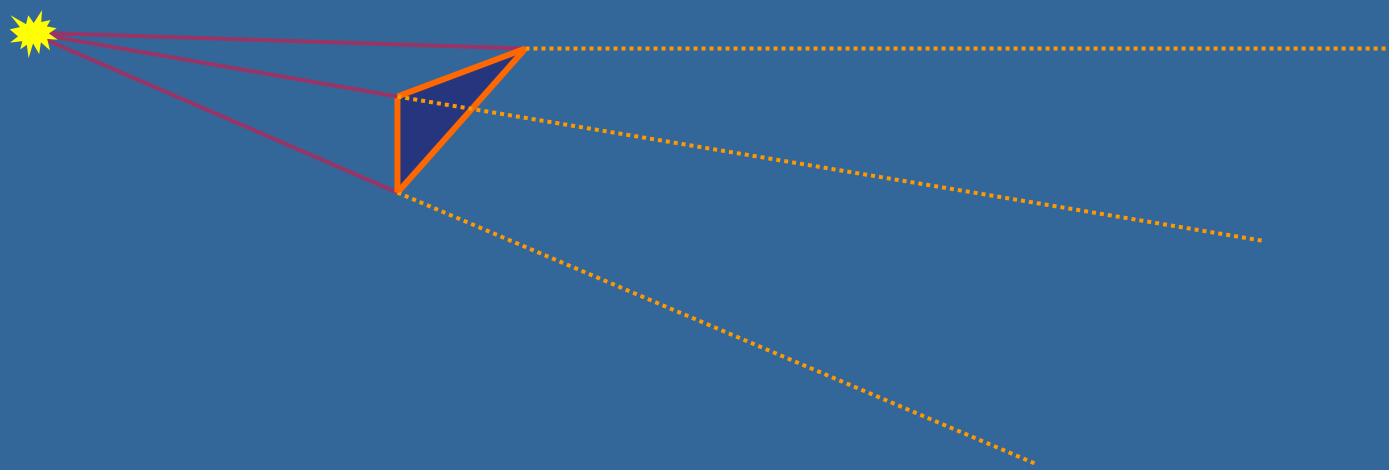
- Choosing bias (epsilon) is not trivial!



- Assume that the ellipse and rectangle is the same surface
  - We do not want incorrect self-shadowing
- Solution: add bias
  - $d_{sm} + \text{bias} < d_{eye}$ → shadow

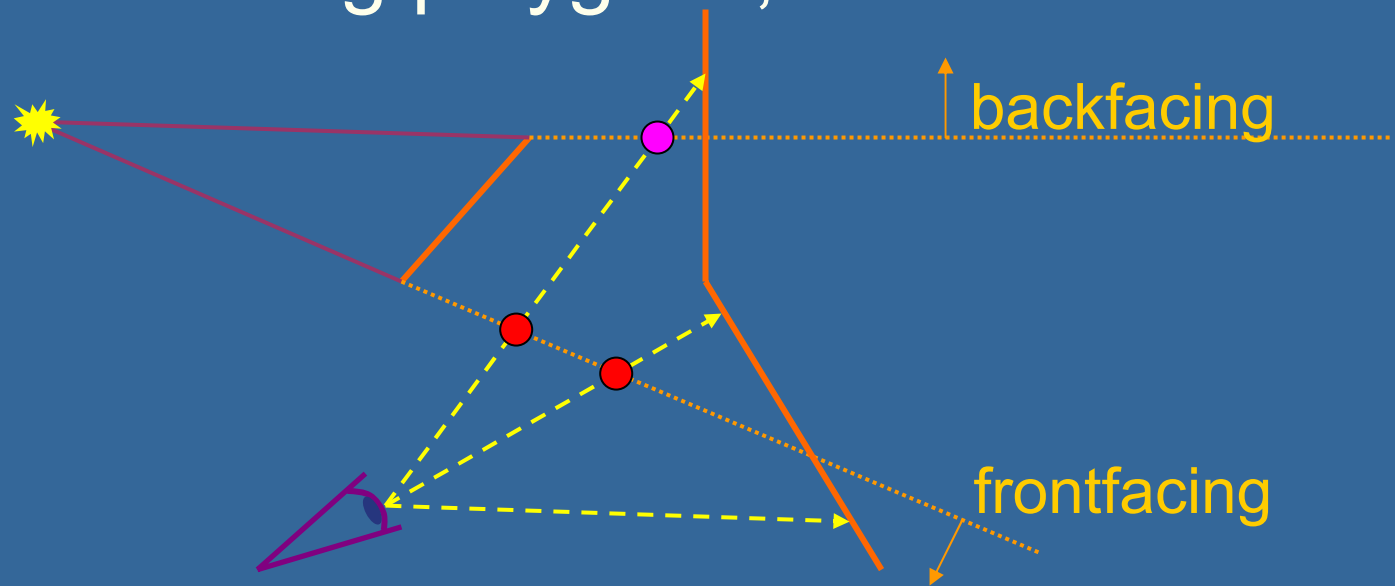
Shadow volumes

- Shadow volume concept
- Create volumes of space in shadow from each polygon in light.
- Each triangle creates 3 projecting quads



Using the Volume

- To test a point, count the number of polygons between it and the eye.
- If we look through more frontfacing than backfacing polygons, then in shadow.

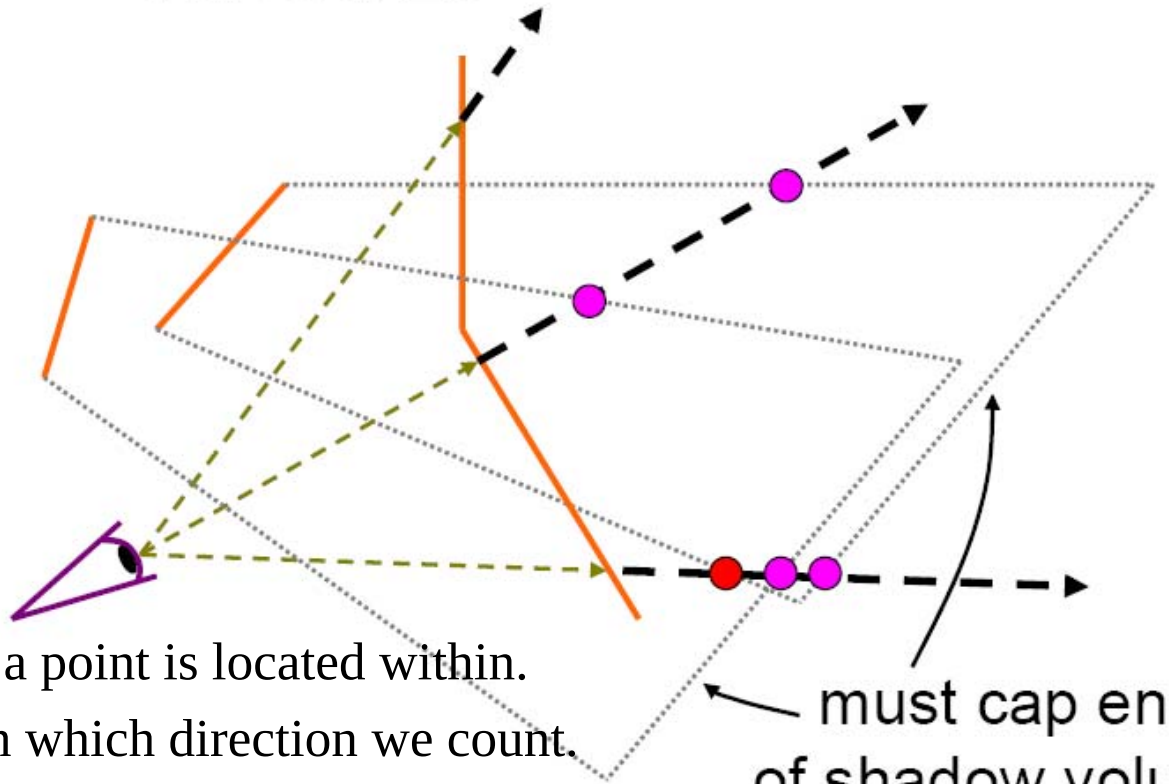


How to implement shadow volumes with stencil buffer (Z-pass)

- A four pass process [Heidmann91]:
  - **1st Pass**: render the scene with just ambient lighting.
  - Turn off updating Z-buffer and writing to color buffer (i.e. Z-compare, draw to stencil only).
  - **2nd pass**: render front facing shadow volume polygons to stencil buffer, incrementing count.
  - **3rd pass**: render backfacing shadow volume polygons to stencil, decrementing.
  - **4th pass**: render diffuse and specular where stencil buffer is 0.

Solution: Count Beyond Surface "Z-fail-algorithm"

- Render to stencil only when shadow volume Z >= stored Z!

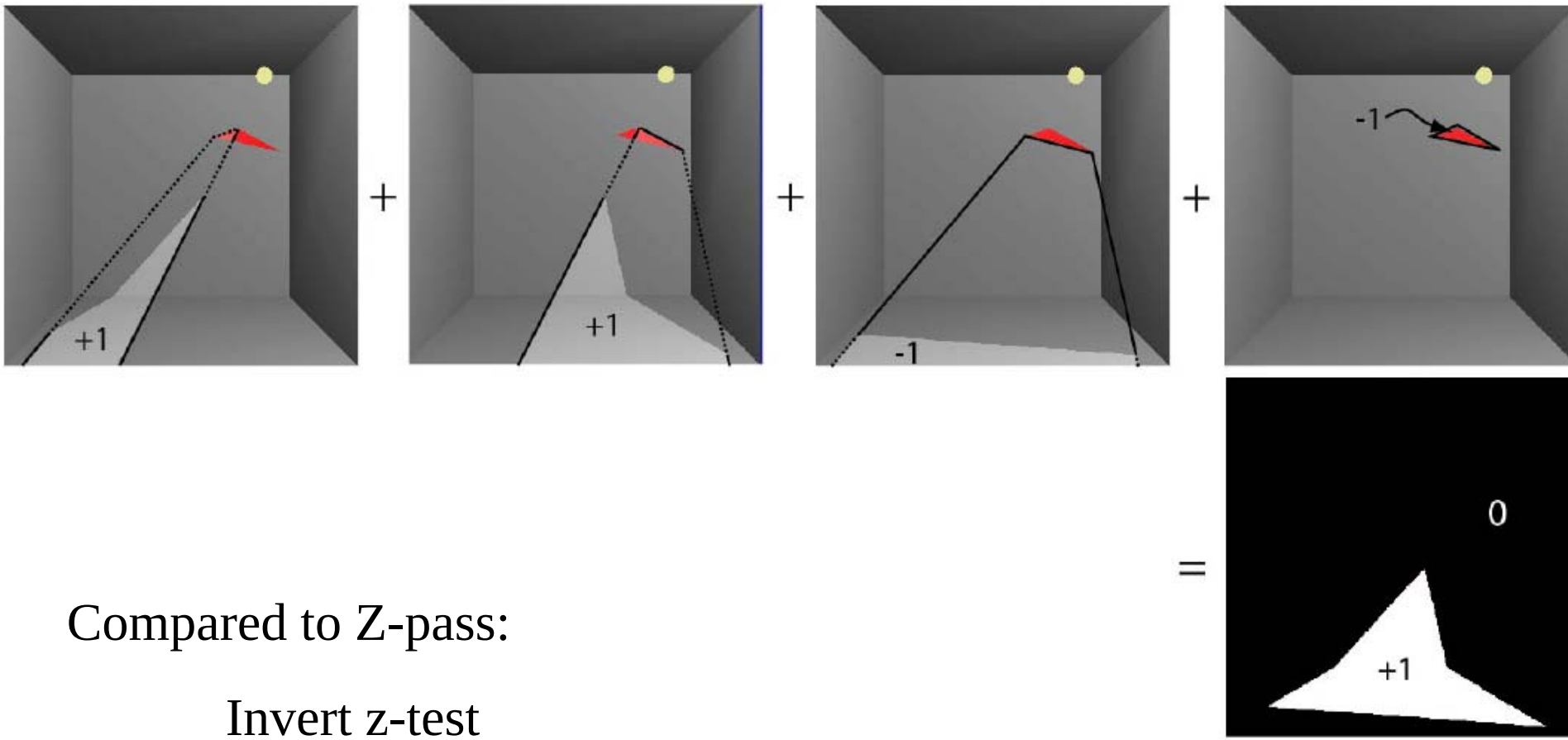


a=#shadow volumes a point is located within.

a is independent of in which direction we count.

Choose point at infinity. That point is always outside shadow, due to the caping of the shadow volumes

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 Volumes

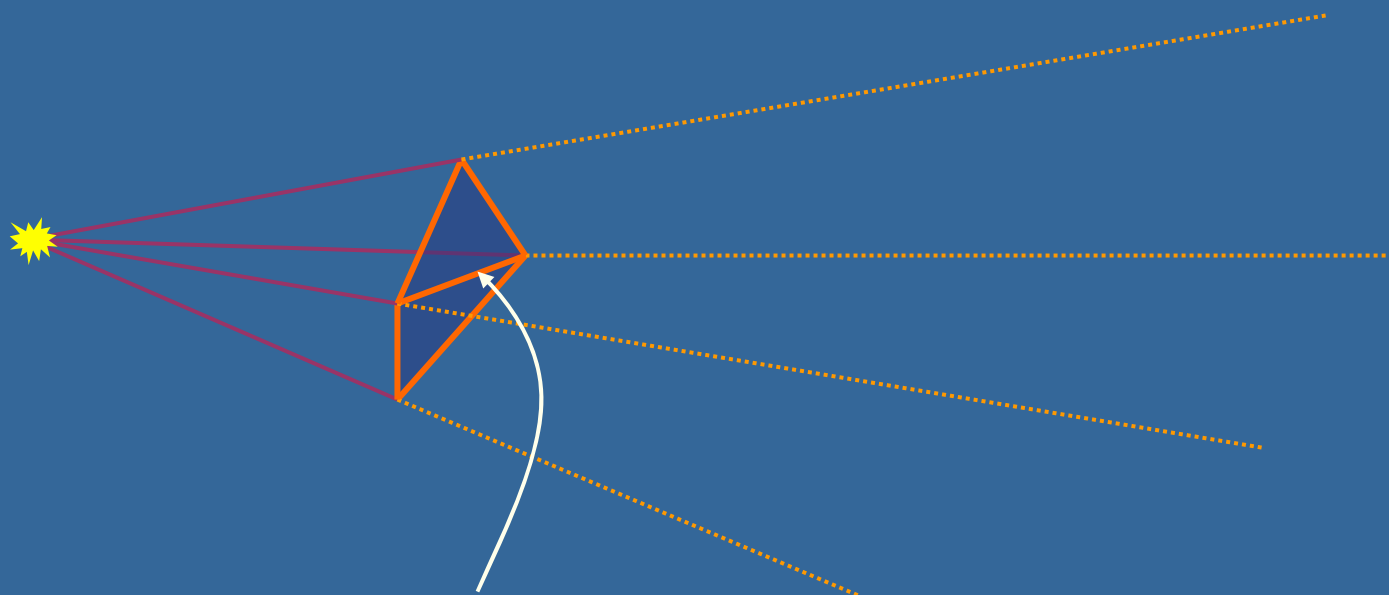
- *Good*: Anything can shadow anything, including self-shadowing, and the shadows are **sharp**.
- *Bad*: **3 or 4 passes**, shadow polygons must be generated and rendered → lots of polygons & **fill**,
- Z-Fail: Near-capping polygons cannot be rendered with tristrips .
- Z-Pass: counting problems, ZP+: more complex.

Shadow Maps

- *Good*: Anything to anything, **constant cost** regardless of complexity, map can sometimes be reused.
- *Bad*: Frustum limited. **Jagged shadows** if res too low, **biasing** headaches.

Merging Volumes

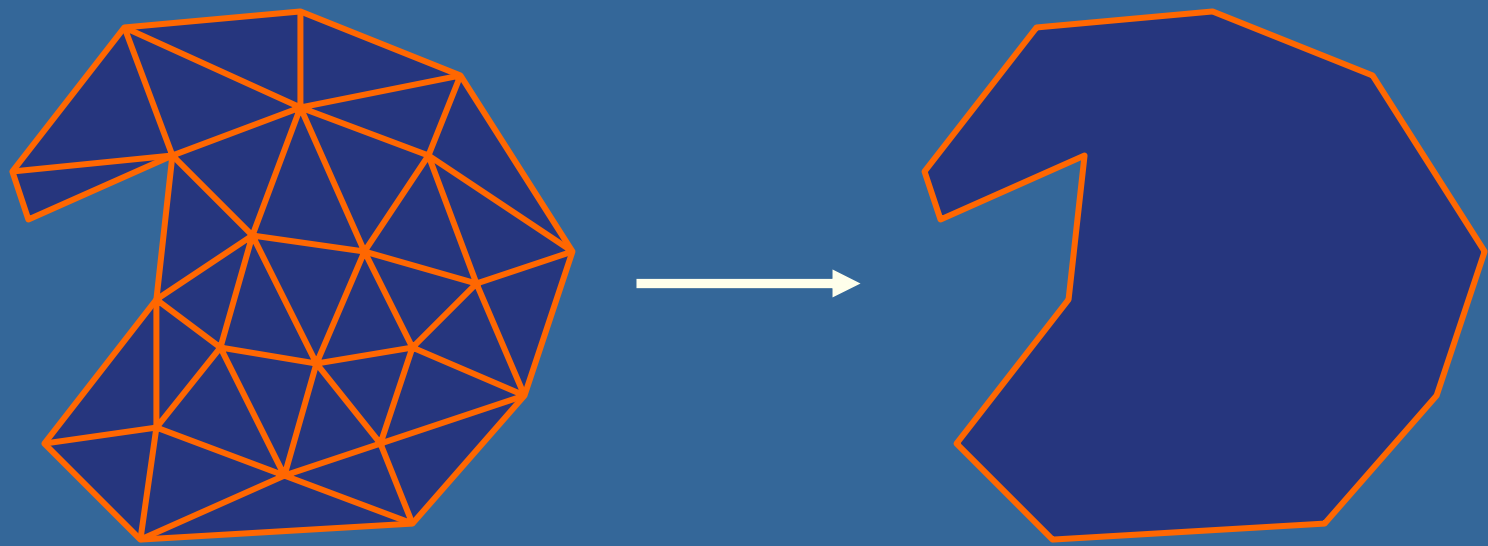
- Edge shared by two polygons facing the light creates front and backfacing quad.



This interior edge makes two quads, which cancel out

Silhouette Edges

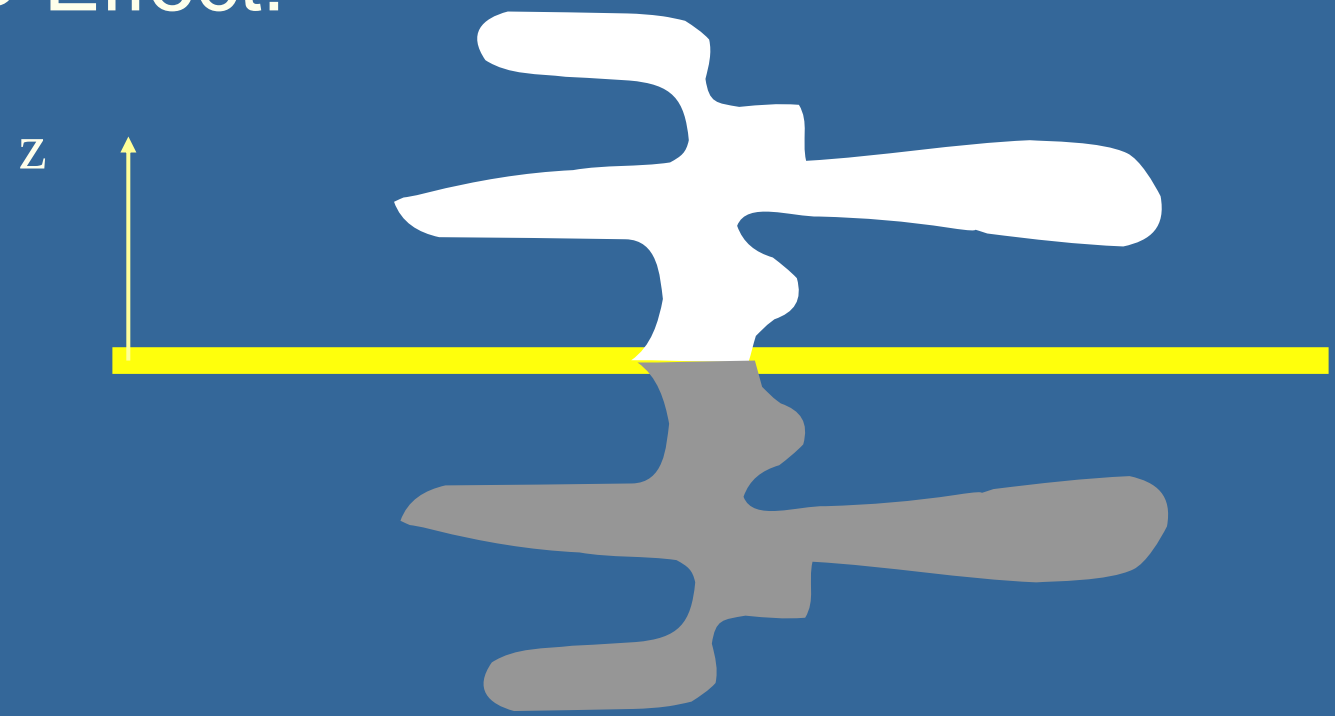
From the light's view, caster interior edges do not contribute to the shadow volume.



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

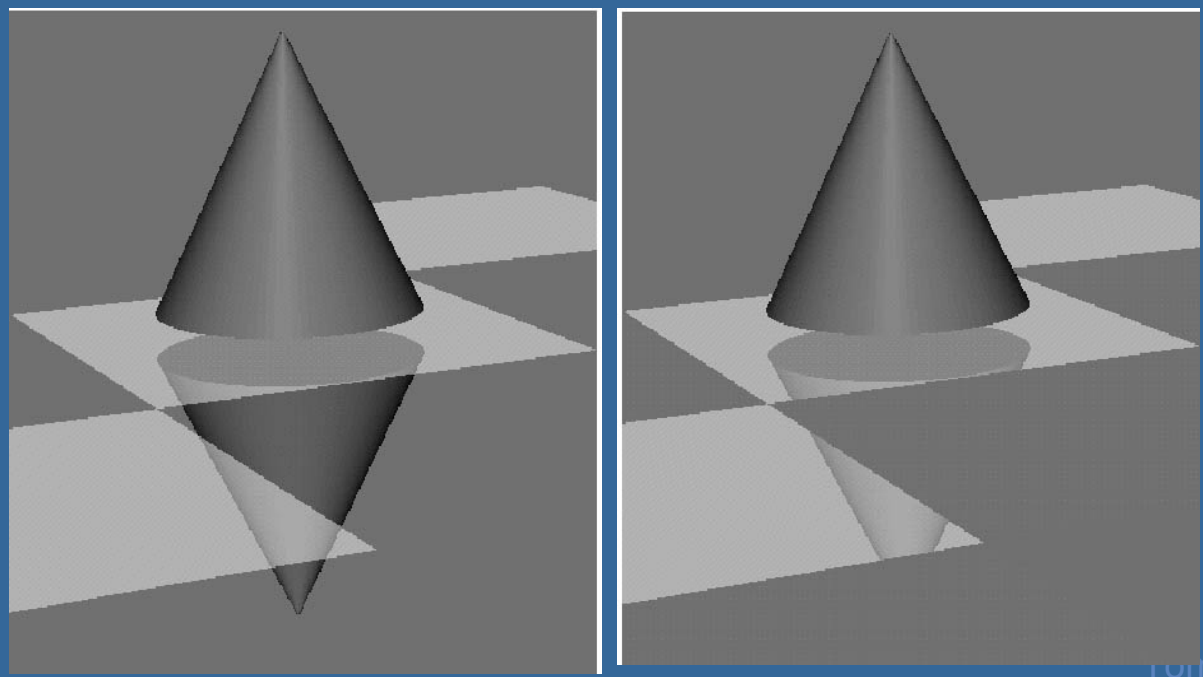
Planar reflections

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



Planar reflections

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



Planar reflections

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

Bildbehandling

- Fokusera på algoritmerna:
  - Brusreducering s:6-7
    - Utjämnande filter
    - Laplace filter
  - Och för att räkna #svarta objekt på vit bakgrund s:8

```
void Fill (int x,int y)
{
   if (GetPixel(x, y)) {
       MoveTo(x,y); LineTo(x,y);
       Fill(x + 1, y);Fill(x - 1, y);
       Fill(x, y + 1); Fill(x, y - 1);
    }
}

void main()
{
  /* Deklarationer*/
  count = 0;
  for (y = r.top; y <=r.bottom; y++) {
       for (x = r.left; x <= r.right; x++) {
           if (GetPixel(x, y)) {
              /*Objekt hittat; radera alla punkter i objektet */
              count = count + 1;
              PenPat(White); // Rita med vitt
              printf("Found object %d\n", count);
              Fill(x, y);
              PenPat(Black);  // Rita med svart
            }
       }
   }
   printf("Antal objekt= %d \n", count);
}
```

Eller ”plot(x,y)”

”Sätt färg till vitt”