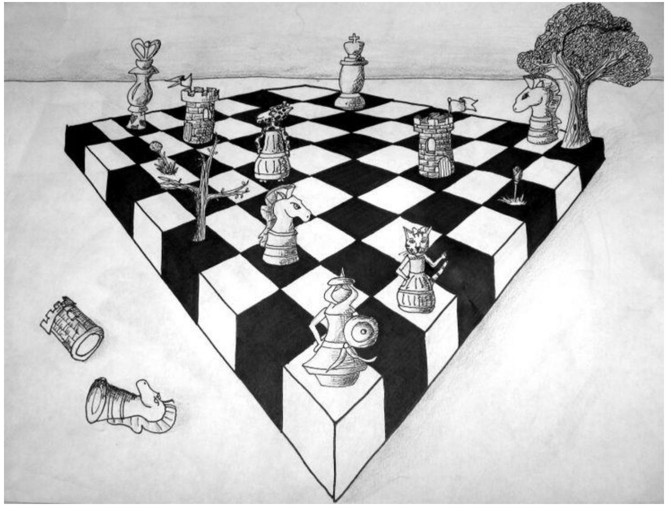
GRK 4

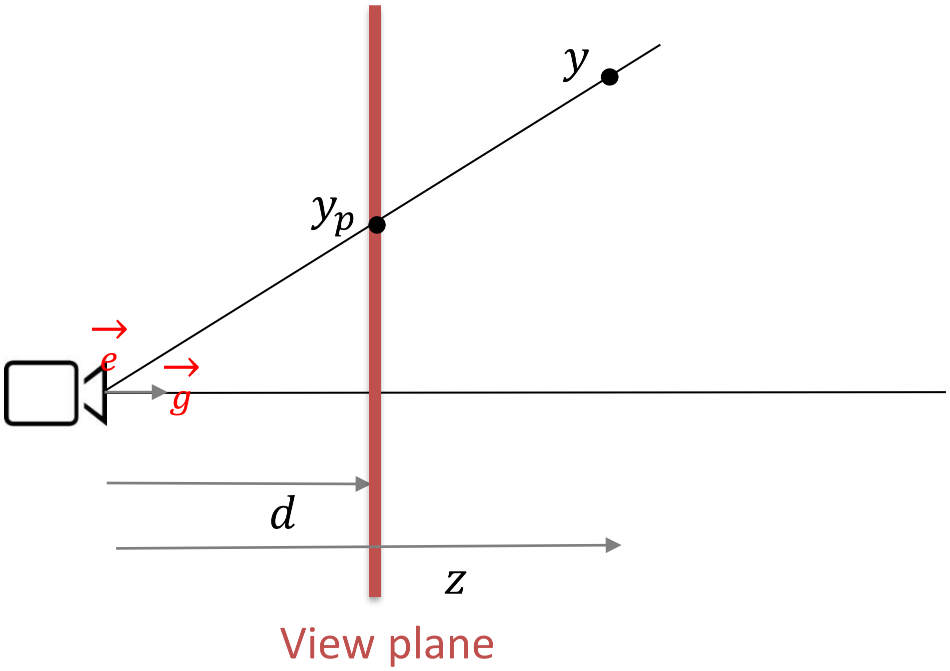
Dr W Palubicki

Perspective projection

- What is perspective?
- The size of an object is proportional to its distance from the viewpoint.



Perspective Math



$$M\begin{bmatrix}x\\y\\z\\1\end{bmatrix}=\begin{bmatrix}x_p\\y_p\\z\\1\end{bmatrix}=\begin{bmatrix}\frac{nx}{z}\\\frac{ny}{z}\\z\\1\end{bmatrix}$$

Homogeneous coordinates

- In homogeneous coordinates

(x, y, z, 1) represents the point **(x, y, z)**

Additionally, we now allow other points

(x, y, z, w) which specify the points $\left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right)$

homogeneous

cartesian

Matrix P

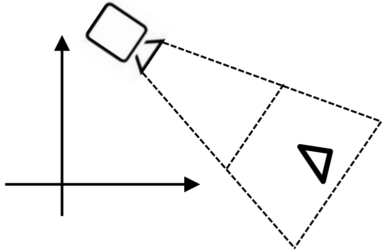
$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{nx}{z} \\ \frac{ny}{z} \\ n+f-\frac{nf}{z} \\ 1 \end{bmatrix}$$

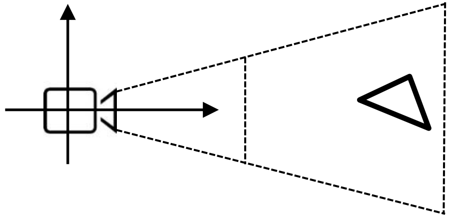
Perspective transformation matrix

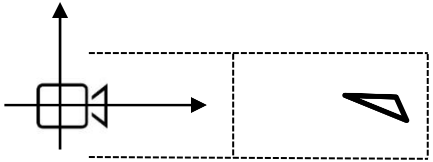
- Our final perspective transformation matrix M_{pers} is then

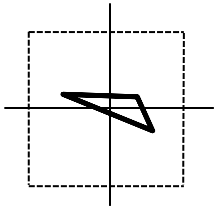
- $$M_{per} = M_{orth}P = M_{orth}\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & n+f & -nf \\ 0 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{l+r}{l-r} & 0 \\ 0 & \frac{2n}{t-b} & \frac{b+t}{b-t} & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

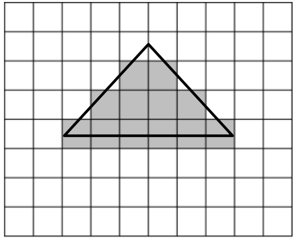
Rendering pipeline











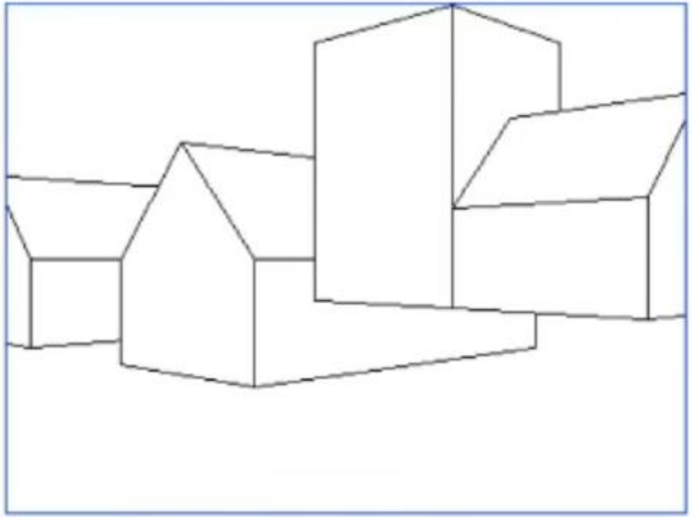
- Rendering pipeline with **orthographic** projection:
- $M_{vp}M_{ort}M_{cam}\begin{bmatrix}x\\y\\z\\1\end{bmatrix}$
- Rendering pipeline with **perspective** projection:
- $M_{vp}M_{per}M_{cam}\begin{bmatrix}x\\y\\z\\1\end{bmatrix}$

Drawing on the Display

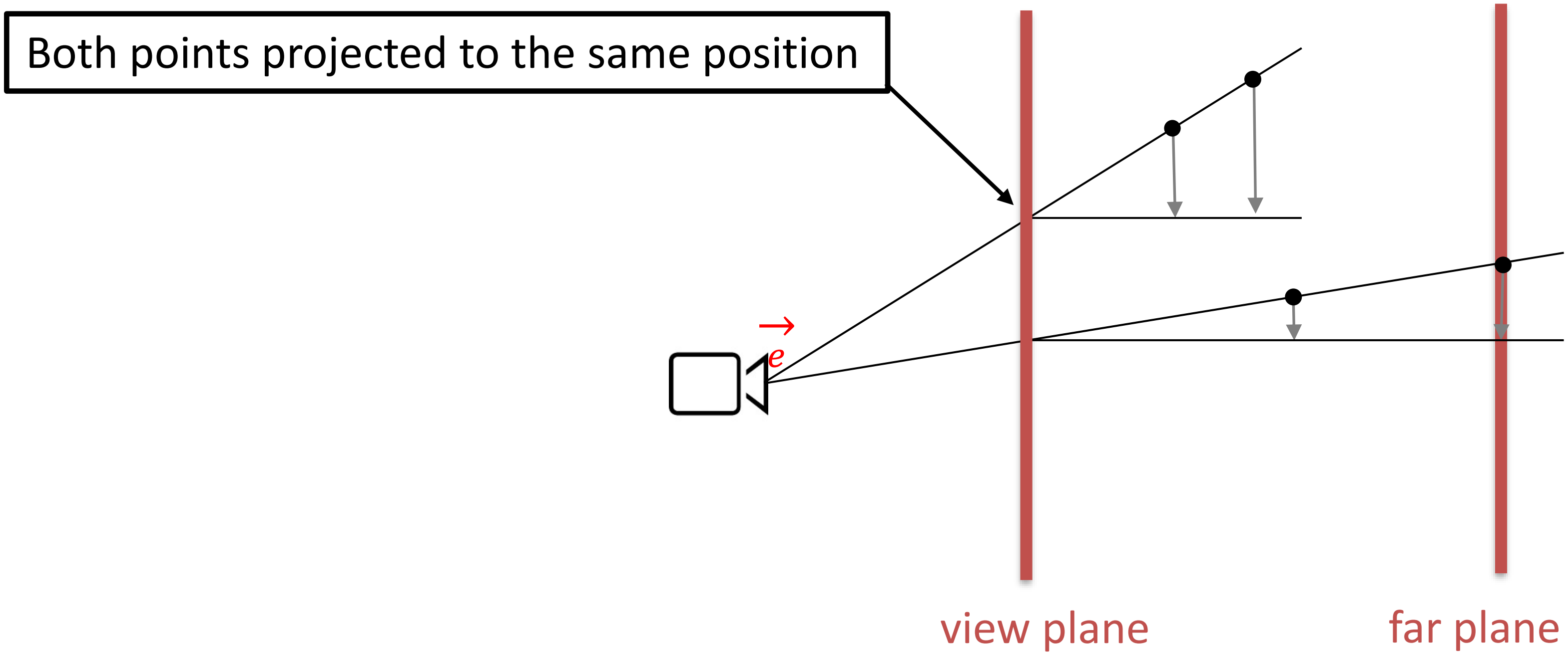
- The following pseudo-code illustrates how to draw a line between two points a and b

```
compute M_vp
compute M_per
compute M_cam
M = M_vp M_per M_cam

for each line segment (a, b) do
     p = M a
     q = M b
     drawline(p_x/p_w, p_y/p_w, q_x/q_w, q_y/q_w)
```

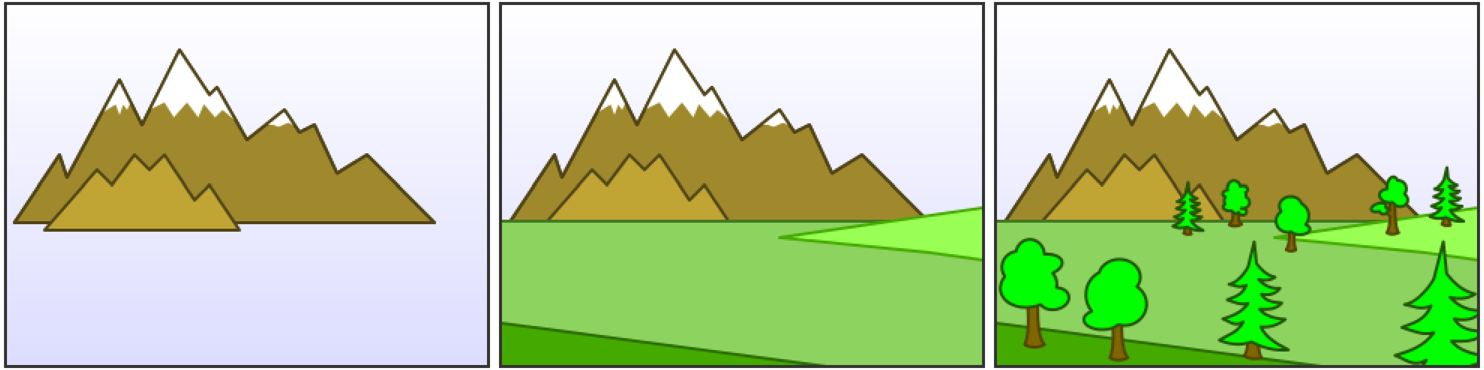
Drawing Order Problem



Why?


Both points projected to the same position
$\vec{e}$
view plane
far plane

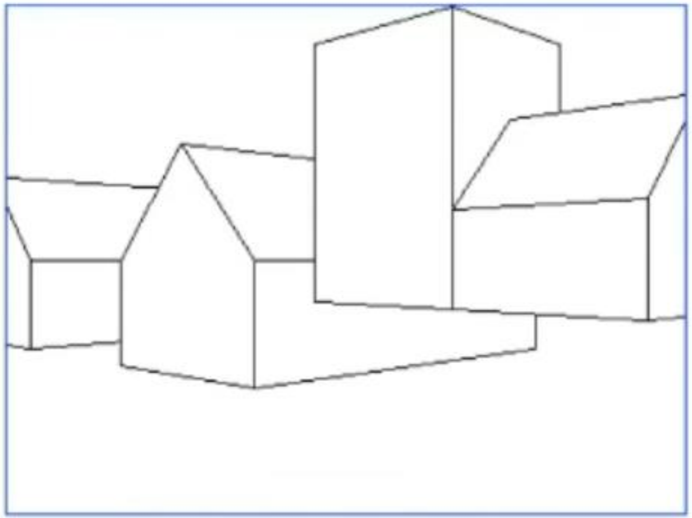
Painter’s algorithm

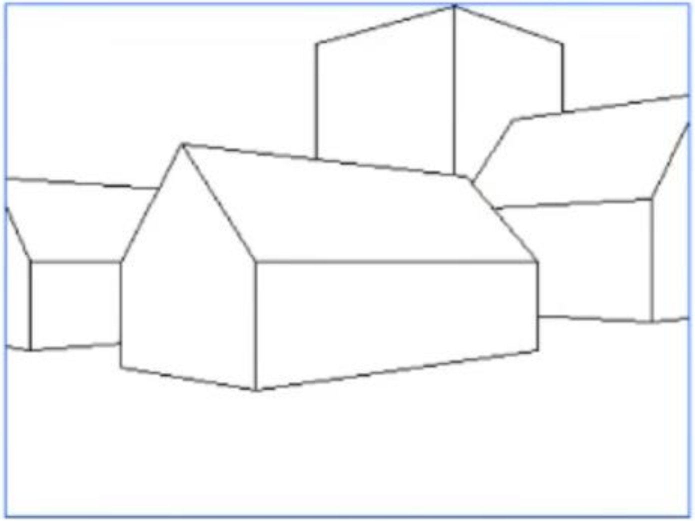


Painter’s algorithm

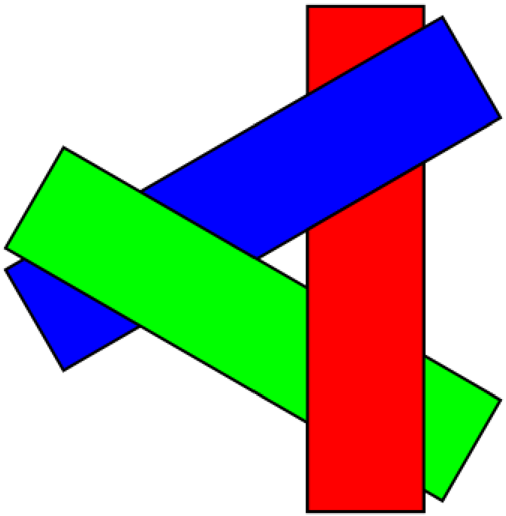
- Algorithm draws polygons in relation to distance from camera
- Smallest coordinate z_s of all polygons is used to determine the distance
- Closer polygons are drawn on top of further ones

Example





Partial polygon overlay



z-buffer / depth buffer

- Z-buffer algorithm is similar to painter’s algorithm but at the scale of pixels
- Two buffers (arrays) are created where each pixel corresponds to one lement of the array
- **depth buffer** stores distance **z** from the nearest surface in world space for each pixel
- **frame buffer** stores indices of polygons (to later on select corresponding colors)

z-buffer / depth buffer

Algorithm 1 Z buffer

Require: a set of polygons P, a depth buffer array Z and a frame buffer array F
  initialise Z to $z_{\max}$
  for all polygons in P **do**
    for all pixels in the current polygon **do**
      calculate the z co-ordinate of the point corresponding to the current pixel
      if $z < Z(x, y)$ **then**
        Replace $Z(x, y)$ with z
        Replace $F(x, y)$ with the colour of the current polygon
      end if
    end for
  end for
  Display F on screen

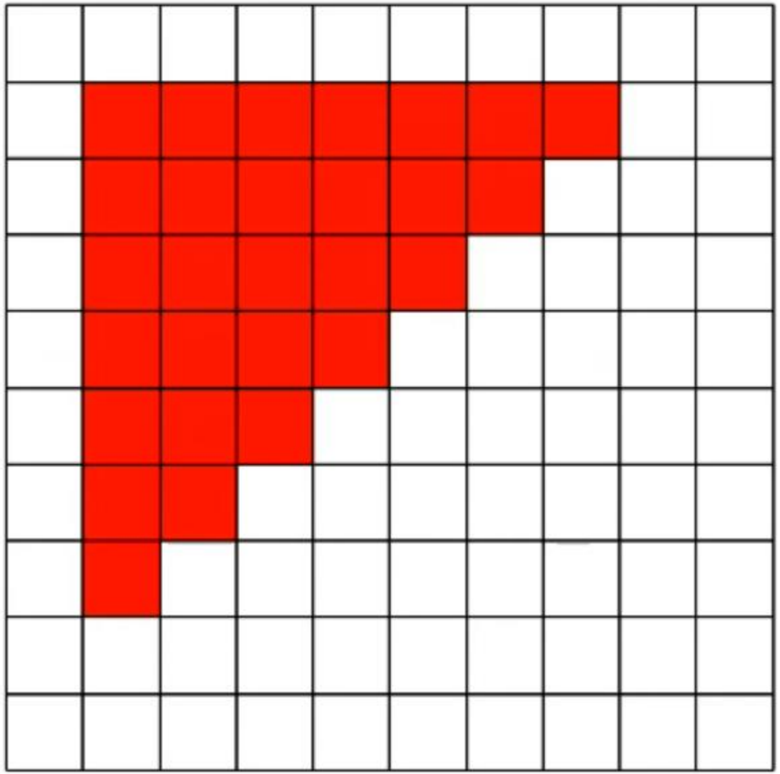
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

Z buffer

Frame buffer

∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	1	2	3	4	5	6	7	∞	∞
∞	1	2	3	4	5	6	∞	∞	∞
∞	1	2	3	4	5	∞	∞	∞	∞
∞	1	2	3	4	∞	∞	∞	∞	∞
∞	1	2	3	∞	∞	∞	∞	∞	∞
∞	1	2	∞	∞	∞	∞	∞	∞	∞
∞	1	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

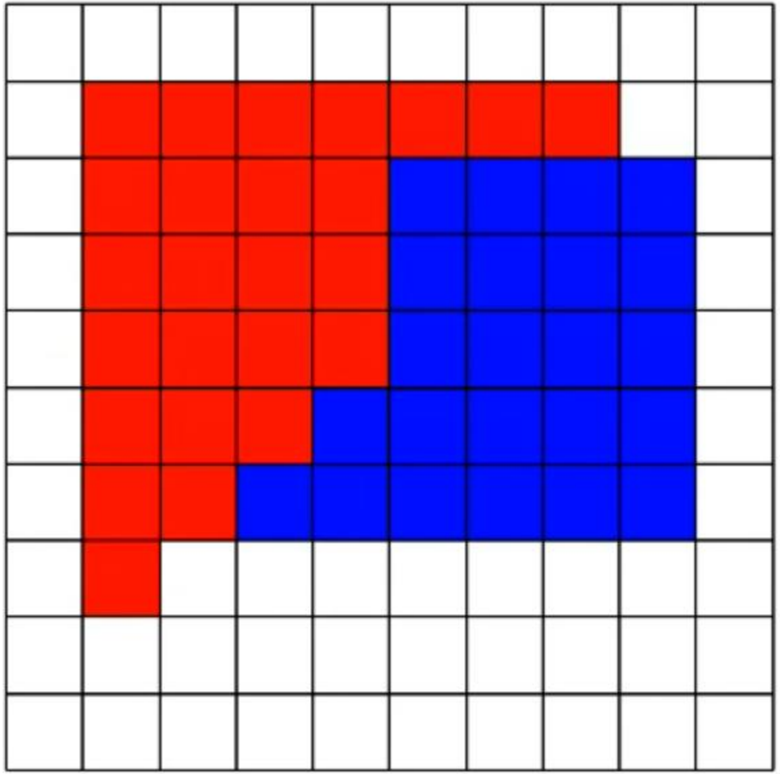
Z buffer



Frame buffer

∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	1	2	3	4	5	6	7	∞	∞
∞	1	2	3	4	4	3	2	1	∞
∞	1	2	3	4	4	3	2	1	∞
∞	1	2	3	4	4	3	2	1	∞
∞	1	2	3	5	4	3	2	1	∞
∞	1	2	6	5	4	3	2	1	∞
∞	1	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
∞	∞	∞	∞	∞	∞	∞	∞	∞	∞

Z buffer

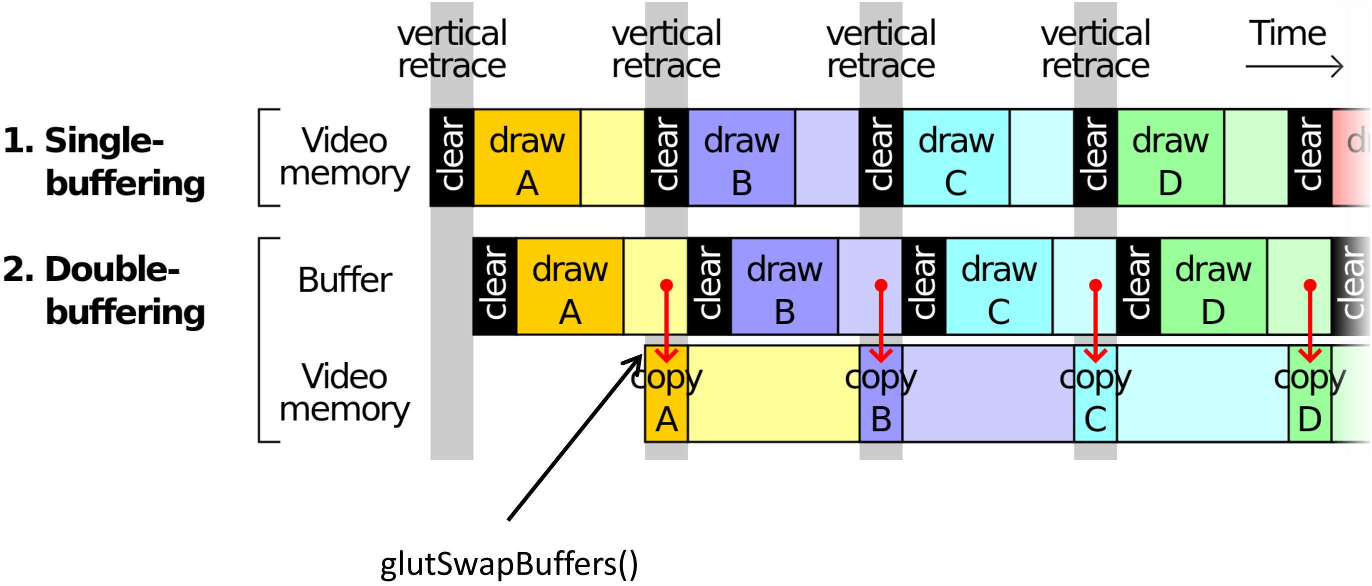


Frame buffer

GLUT Callbacks

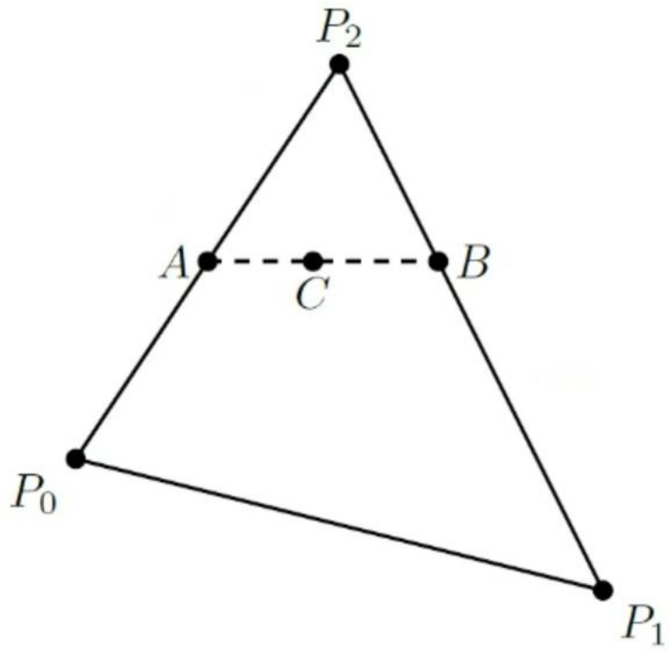
```
void display( void )
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    glDrawArrays( GL_TRIANGLES, 0, NumVertices );
    glutSwapBuffers();
}

void keyboard( unsigned char key, int x, int y )
{
    switch( key ) {
        case 033: case 'q': case 'Q':
            exit( EXIT_SUCCESS );
            break;
    }
}
```

Double Buffering



How to compute the inner points

- Go through all horizontal pixel lines (scan lines) from top to bottom
- Calculate pixel positions on the right and left ends of each scan line by interpolating between vertices P
- Calculate pixel positions on the scan line C by interpolating between A and B

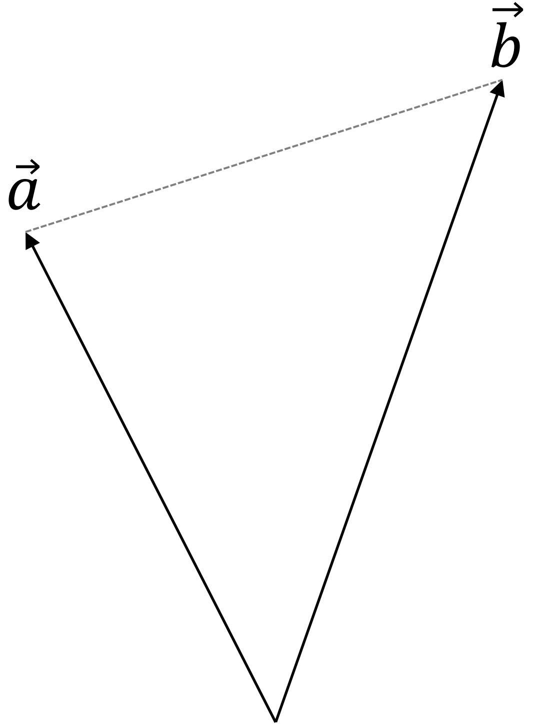


Linear Interpolation

- Given two vectors $\vec{a}, \vec{b}$, linear interpolation is defined by:

$$\vec{p}(t) = (1 - t)\vec{a} + t\vec{b}$$

where $t \in [0, 1]$

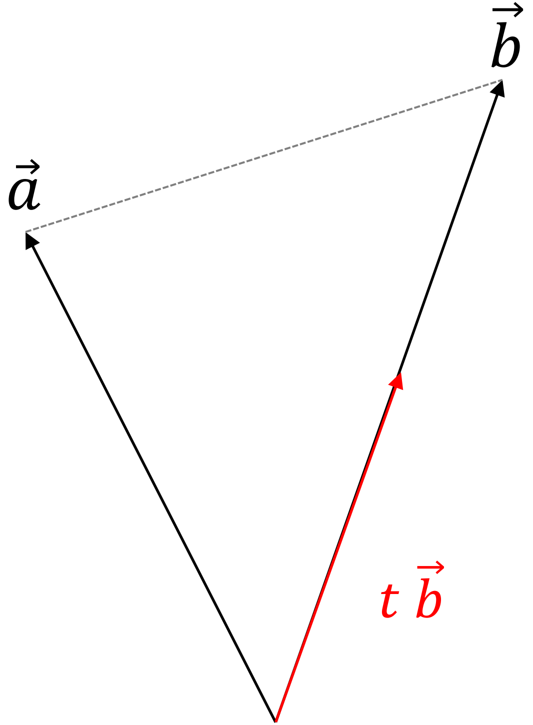


Linear Interpolation

- Given two vectors $\vec{a}, \vec{b}$, linear interpolation is defined by:

$$\vec{p}(t) = (1 - t)\vec{a} + t\vec{b}$$

where $t \in [0, 1]$

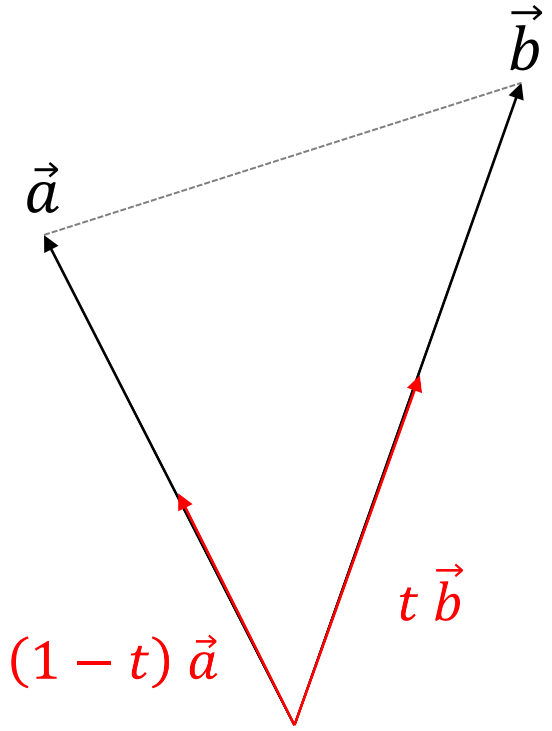


Linear Interpolation

- Given two vectors $\vec{a}, \vec{b}$, linear interpolation is defined by:

$$\vec{p}(t) = (1 - t)\vec{a} + t\vec{b}$$

where $t \in [0, 1]$

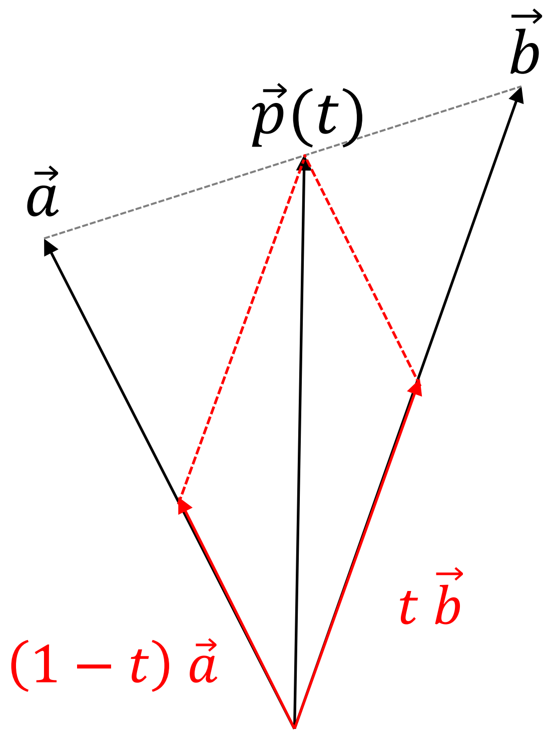


Linear Interpolation

- Given two vectors $\vec{a}$, $\vec{b}$, linear interpolation is defined by:

$$\vec{p}(t) = (1 - t)\vec{a} + t\vec{b}$$

where $t \in [0, 1]$

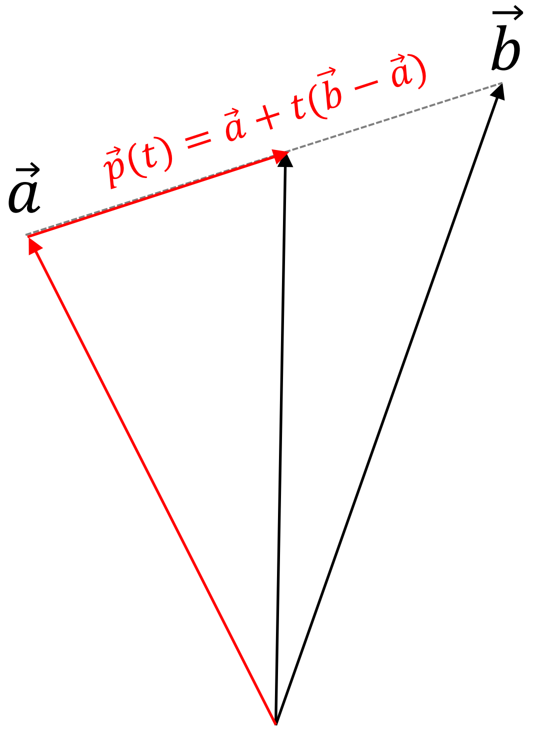


Linear Interpolation

- Geometric interpretation:

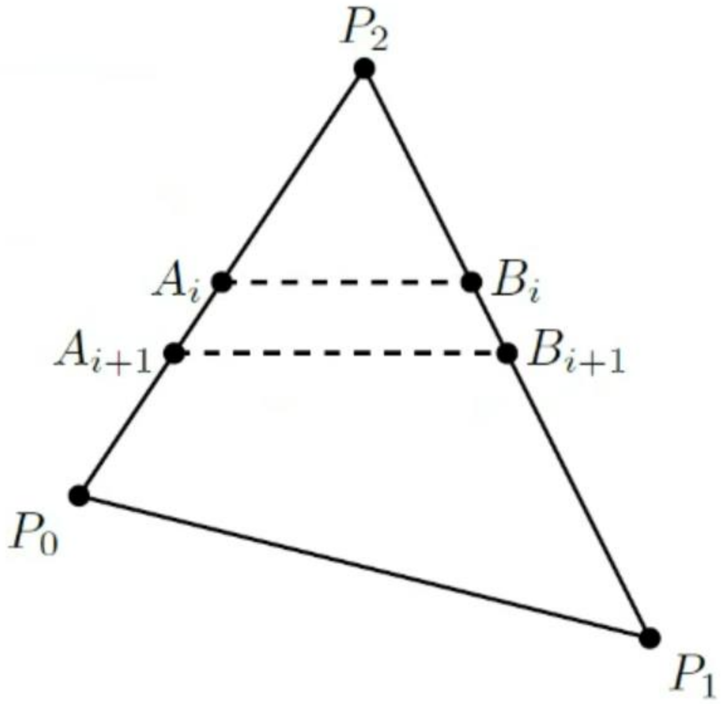
$$\vec{p}(t) = (1-t)\vec{a} + t\vec{b} = \vec{a} - t\vec{a} + t\vec{b}$$
$$= \vec{a} + t(\vec{b} - \vec{a})$$

- For $0 \leq t \leq 1$ this gives us all possible positions on a line between $\vec{a}$ and $\vec{b}$



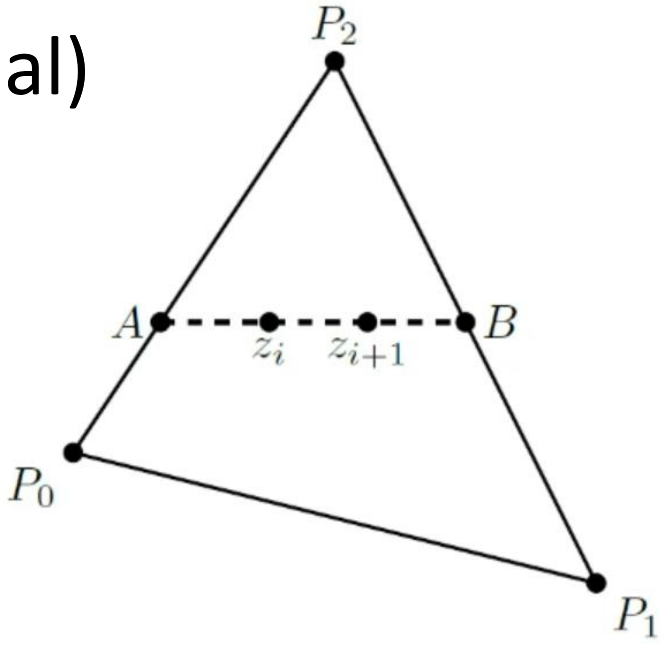
Interpolating between vertices P

- $x_{A,\,i+1} = x_{A,\,i} - \Delta x_A$
- $x_{B,\,i+1} = x_{B,\,i} - \Delta x_B$
- $y_{A,\,i+1} = y_{A,\,i} - 1$
- $y_{B,\,i+1} = y_{B,\,i} - 1$
- Where
- $\Delta x_A = \frac{y_A - y_0}{y_2 - y_0}$
- $\Delta x_B = \frac{y_B - y_1}{y_2 - y_1}$



Calculating z

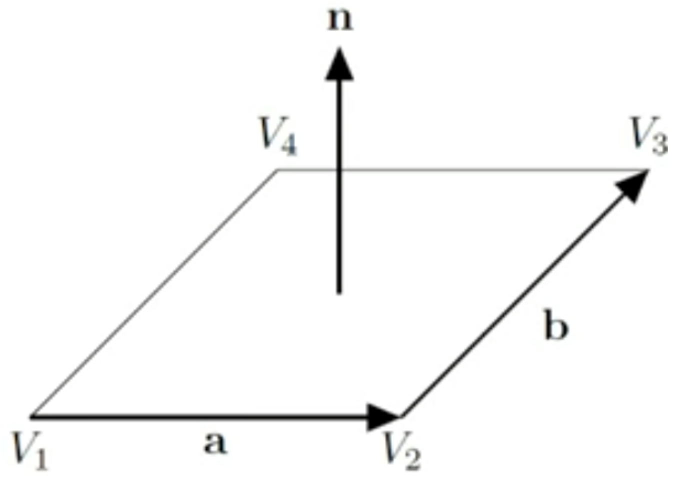
- Using the vector formula for planes:
- N r = s (N denotes plane normal)
- Value z_{i+1} is $z_{i+1} = z_i - \frac{n_x}{n_z}$

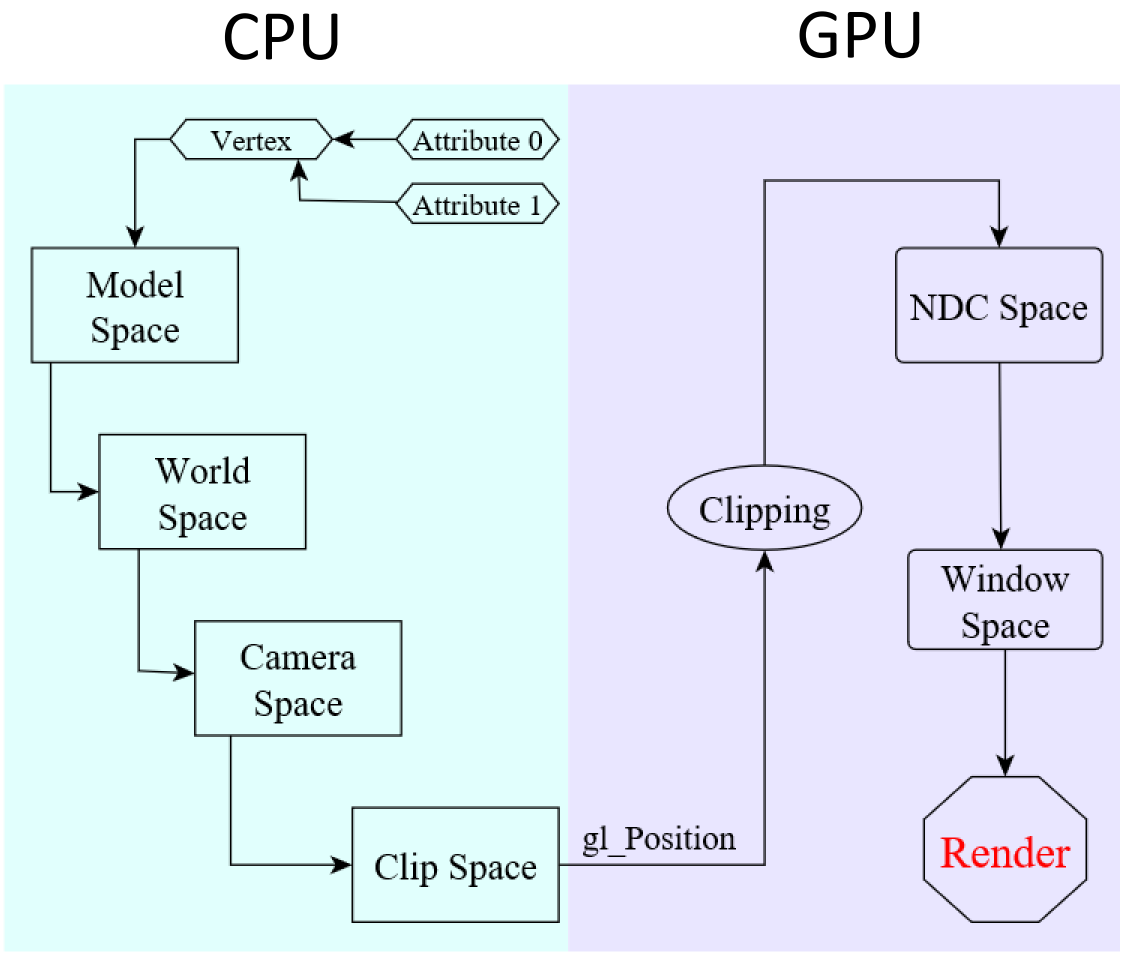


- Where $\frac{n_x}{n_z}$ is constant

Normal vectors

- **Normal vectors** are oriented **orthogonal** to a plane
- We can use the **cross product** to calculate the value of the normal vector **n**
- $n = a \times b$
- Convention is to give the vectors in opposite clock direction
- $n = (V_2 - V_1) \times (V_3 - V_2)$

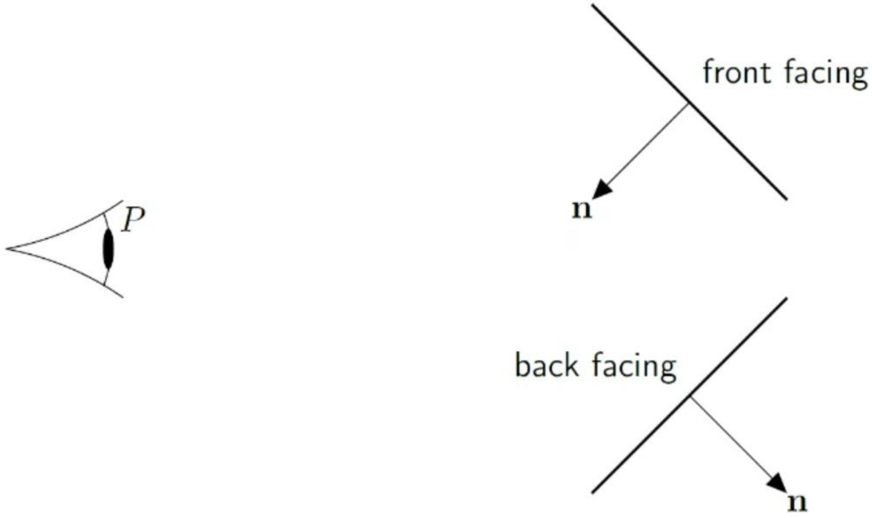




CPU
GPU
Vertex
Attribute 0
Attribute 1
Model Space
World Space
Camera Space
Clip Space
gl_Position
Clipping
NDC Space
Window Space
Render

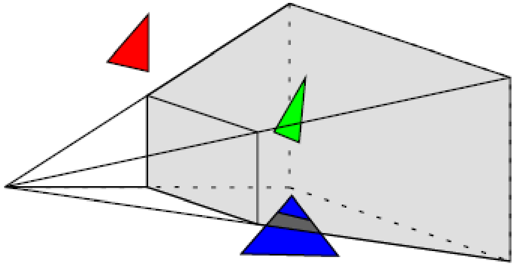
Polygon facing

- A polygon is **front facing** if the normal of a plane is oriented towards the viewpoint, otherwise it is **back facing**



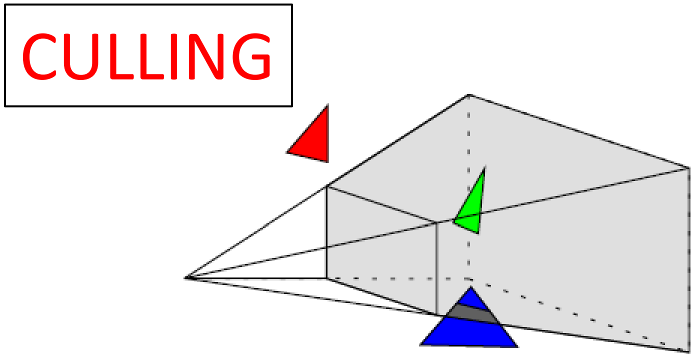
Which triangles should be projected?

- Triangles that lie (partly) outside of the view frustum don’t have to be projected and are culled (**clipping/culling**)



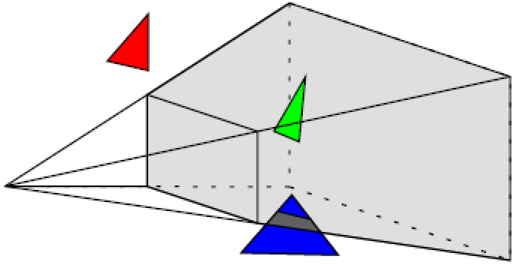
Which triangles should be projected?

- Triangles that lie (partly) outside of the view frustum don't have to be projected and are culled (**clipping/culling**)



Which triangles should be projected?

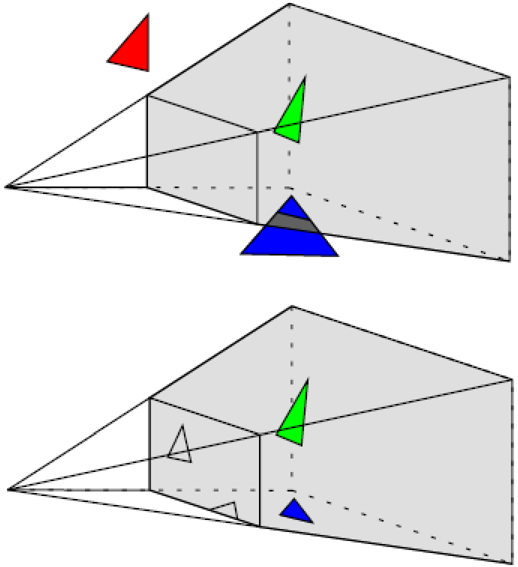
- Triangles that lie (partly) outside of the view frustum don't have to be projected and are culled (**clipping/culling**)



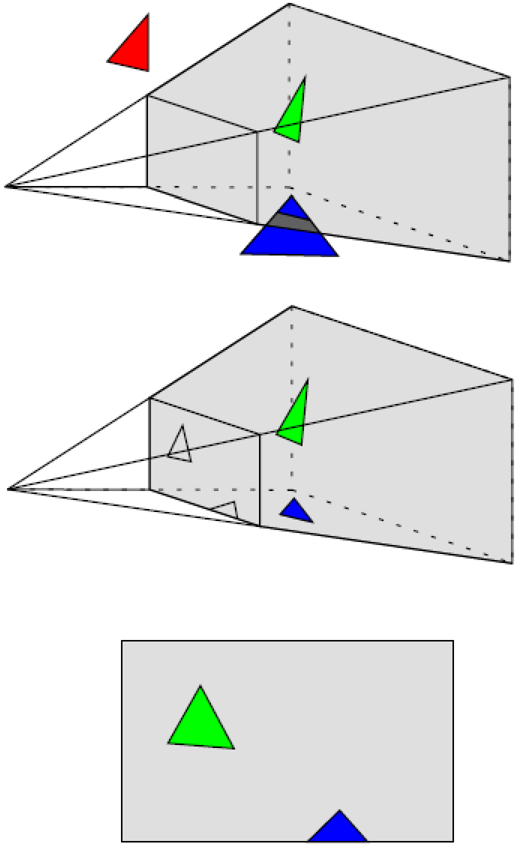
CLIPPING

Which triangles should be projected?

- Triangles that lie (partly) outside of the view frustum don’t have to be projected and are culled (**clipping/culling**)
- The remaining triangles are projected on the view plane

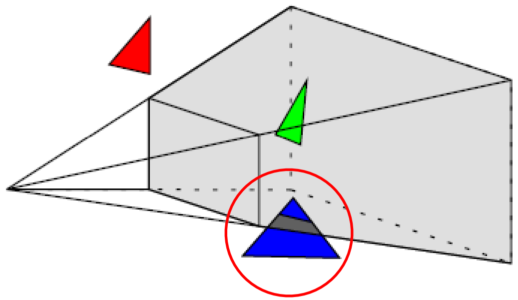


- Triangles that lie (partly) outside of the view frustum don't have to be projected and are culled (**clipping/culling**)
- The remaining triangles are projected on the view plane



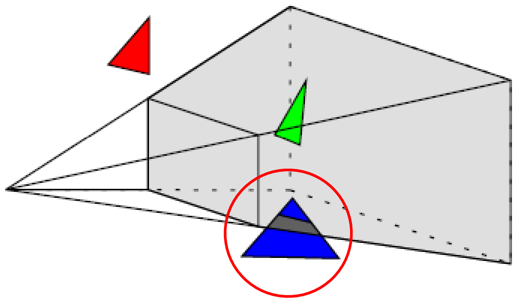
Clipping

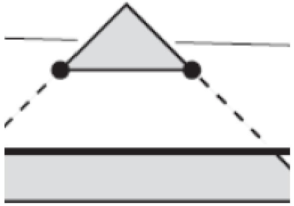
- To decide whether to clip a triangle we have to:
  - Test whether it intersects the **hyperplane**



Clipping

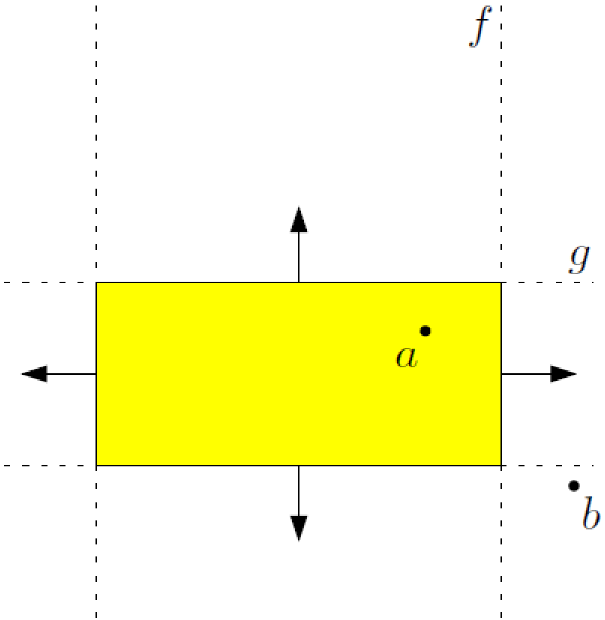
- To decide whether to clip a triangle we have to:
  - Test whether it intersects the **hyperplane**
  - Create **new** triangle(s)





Intersection test

- The hyperplane equation through a point q and normal n is given by:
- $f(p) = n \cdot (p - q) = 0$



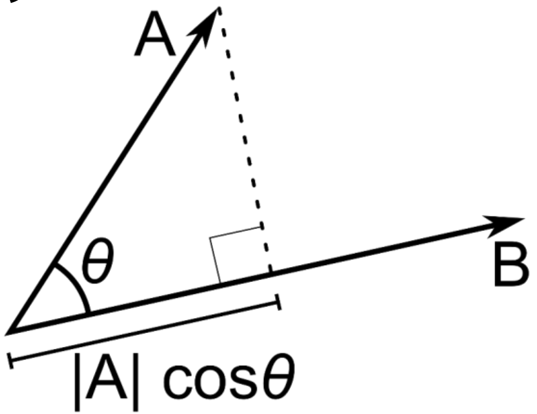
3D – dot product $a \cdot b$

Definition 1: $a \cdot b = a_1 b_1 + a_2 b_2 + a_3 b_3$

3D – dot product $a \cdot b$

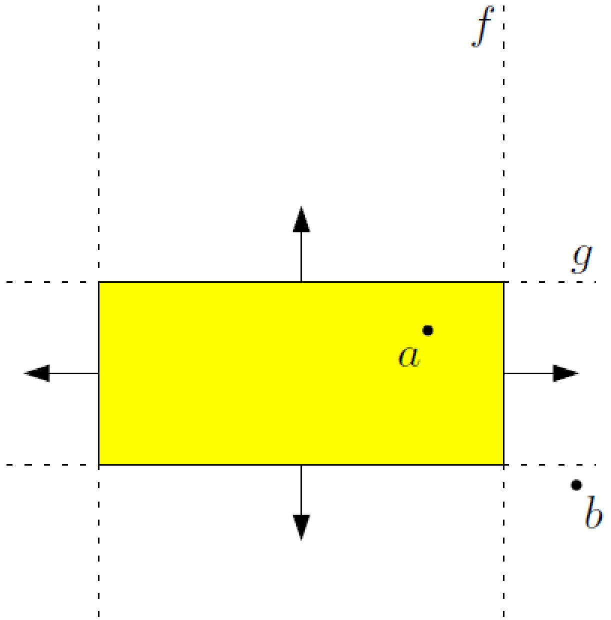
Definition 1: $a \cdot b = a_1 b_1 + a_2 b_2 + a_3 b_3$

Definition 2: $a \cdot b = |a|\,|b| \cos(\theta)$



Intersection test

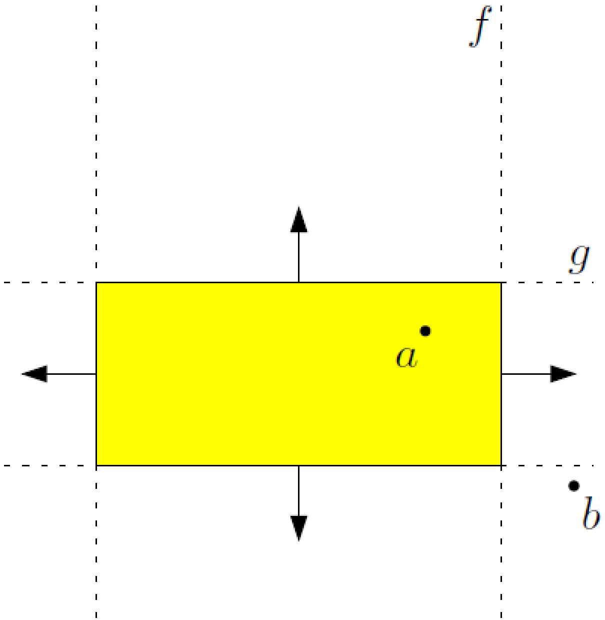
- The hyperplane equation through a point q and normal n is given by:
- $f(p) = n \cdot (p - q) = 0$



Intersection test

- The hyperplane equation through a point q and normal n is given by:
- $f(p) = n \cdot (p - q) = 0$

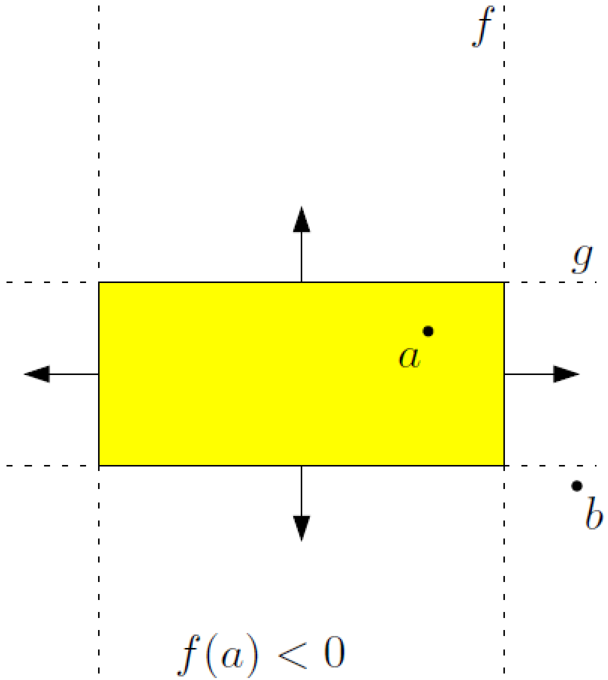
- Convention: hyperplane normals are oriented **outwards** (view frustum), so if $f(p) < 0$ then p is **inside**, and if $f(p) > 0$ then p is **outside** the plane.



Intersection test

- The hyperplane equation through a point q and normal n is given by:
- $f(p) = n \cdot (p - q) = 0$

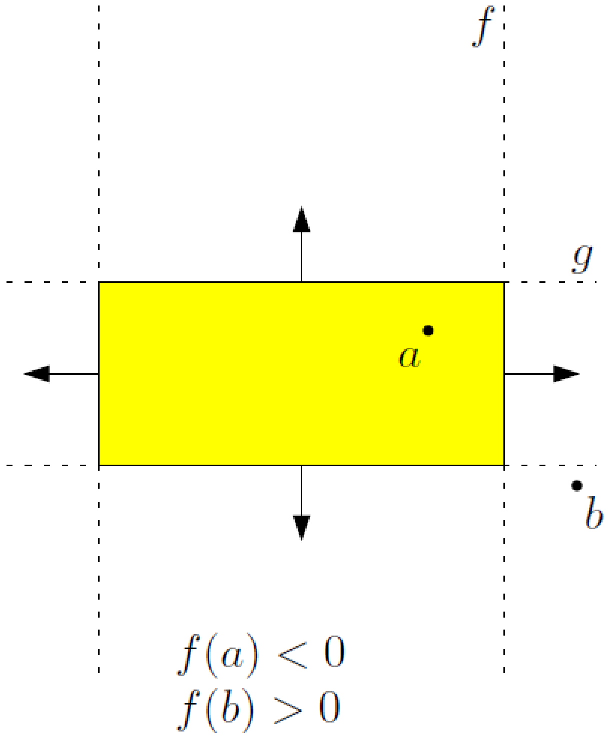
- Convention: hyperplane normals are oriented **outwards** (view frustum), so if $f(p) < 0$ then p is **inside**, and if $f(p) > 0$ then p is **outside** the plane.



Intersection test

- The hyperplane equation through a point q and normal n is given by:
- $f(p) = n \cdot (p - q) = 0$

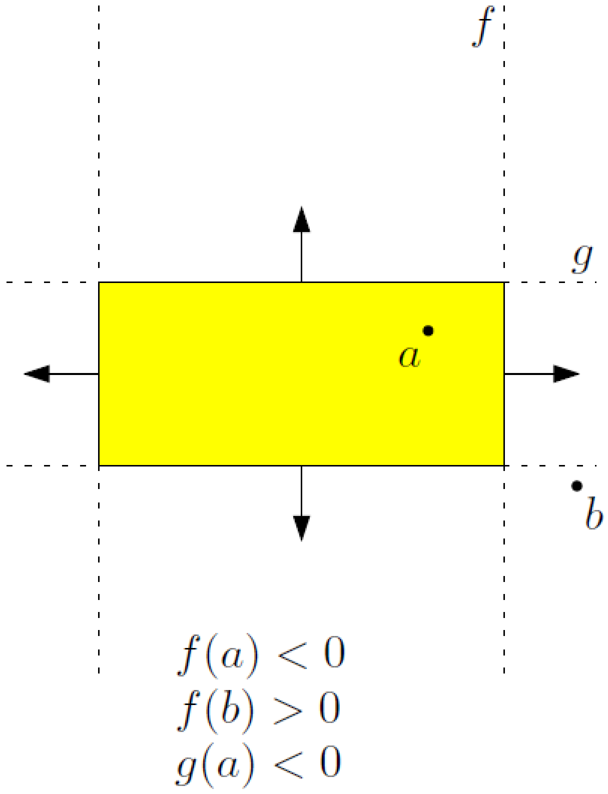
- Convention: hyperplane normals are oriented **outwards** (view frustum), so if $f(p) < 0$ then p is **inside**, and if $f(p) > 0$ then p is **outside** the plane.



Intersection test

- The hyperplane equation through a point q and normal n is given by:
- $f(p) = n \cdot (p - q) = 0$

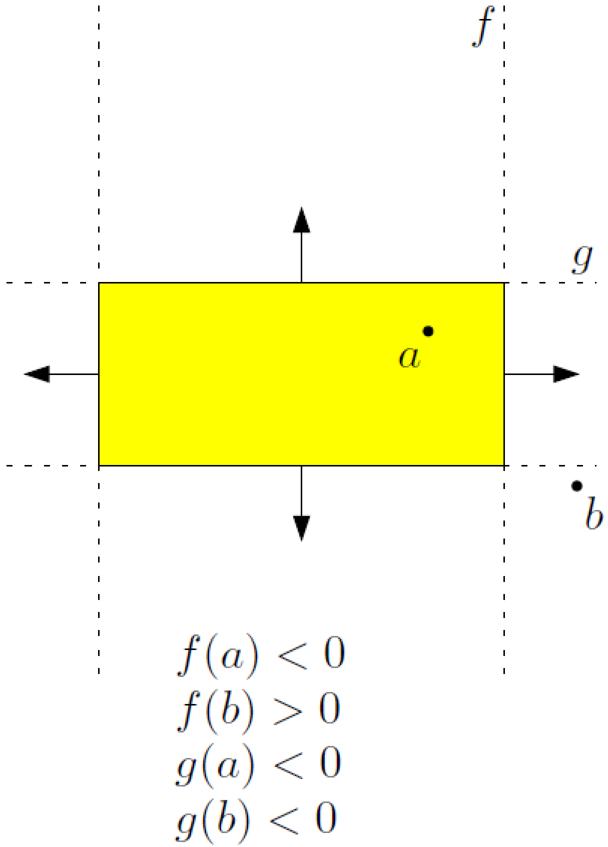
- Convention: hyperplane normals are oriented **outwards** (view frustum), so if $f(p) < 0$ then p is **inside**, and if $f(p) > 0$ then p is **outside** the plane.



Intersection test

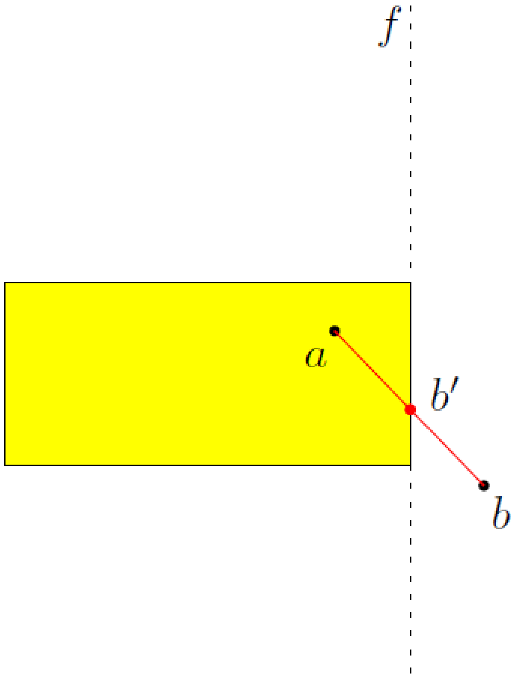
- The hyperplane equation through a point q and normal n is given by:
- $f(p) = n \cdot (p - q) = 0$

- Convention: hyperplane normals are oriented **outwards** (view frustum), so if $f(p) < 0$ then p is **inside**, and if $f(p) > 0$ then p is **outside** the plane.



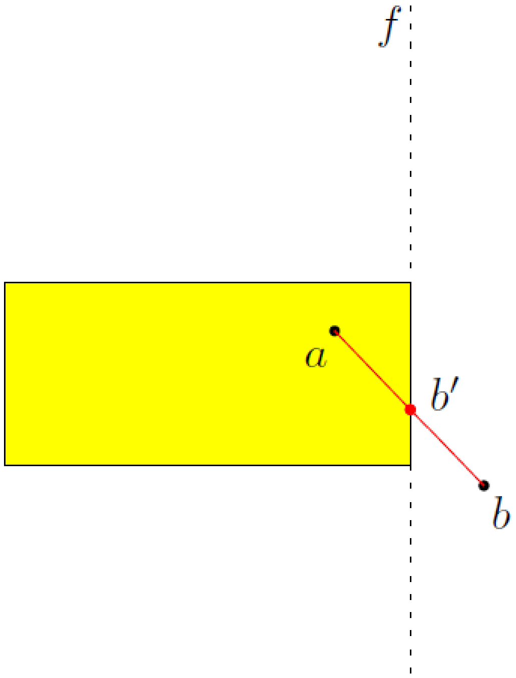
Calculating intersection points

- If two points $\vec{a}$ and $\vec{b}$ are on different sides of a hyperplane we define the line that passes through both points:
- $\vec{p}(t) = \vec{a} + t\left(\vec{b} - \vec{a}\right)$



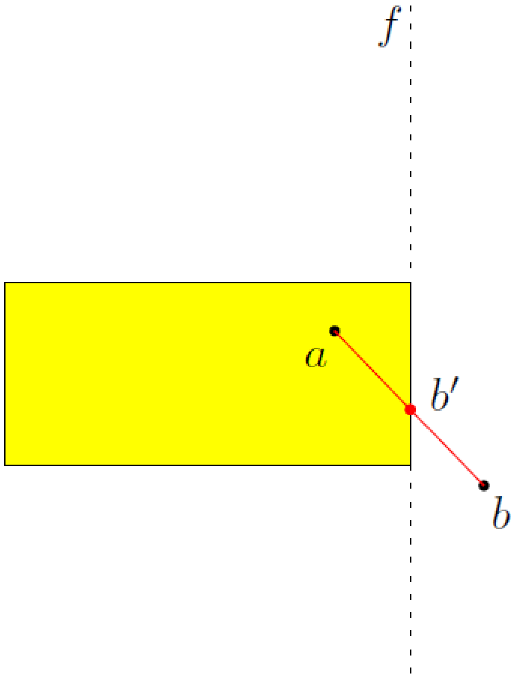
Calculating intersection points

- If two points $\vec{a}$ and $\vec{b}$ are on different sides of a hyperplane we define the line that passes through both points:
- $\vec{p}(t) = \vec{a} + t\left(\vec{b} - \vec{a}\right)$
- Substituting:
- $\vec{n} \cdot (\vec{p} - \vec{q}) = 0$



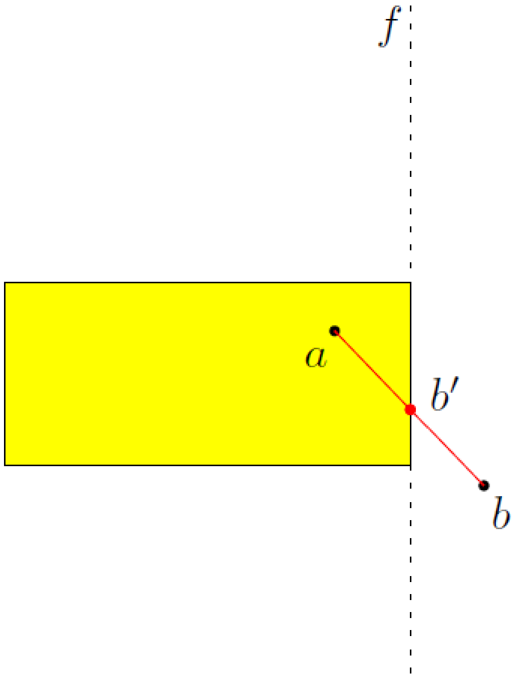
Calculating intersection points

- If two points $\vec{a}$ and $\vec{b}$ are on different sides of a hyperplane we define the line that passes through both points:
- $\vec{p}(t) = \vec{a} + t\left(\vec{b} - \vec{a}\right)$
- Substituting:
- $\vec{n} \cdot (\vec{p} - \vec{q}) = 0$
- $\vec{n} \cdot \left(\vec{a} + t\left(\vec{b} - \vec{a}\right) \; - \vec{q}\right) = 0$



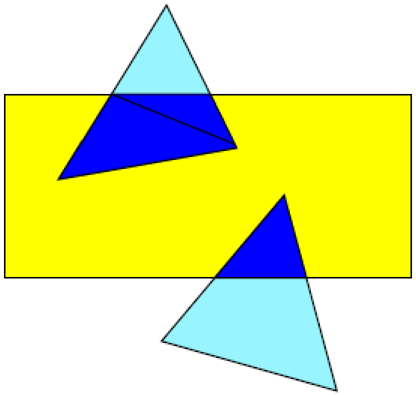
Calculating intersection points

- If two points $\vec{a}$ and $\vec{b}$ are on different sides of a hyperplane we define the line that passes through both points:
- $\vec{p}(t) = \vec{a} + t\left(\vec{b} - \vec{a}\right)$
- Substituting:
- $\vec{n} \cdot (\vec{p} - \vec{q}) = 0$
- $\vec{n} \cdot \left(\vec{a} + t\left(\vec{b} - \vec{a}\right) \ - \vec{q}\right) = 0$
- $t = \frac{\vec{n}\cdot\vec{a}+\vec{n}\cdot\vec{q}}{\vec{n}\cdot(\vec{a}-\vec{b})}$



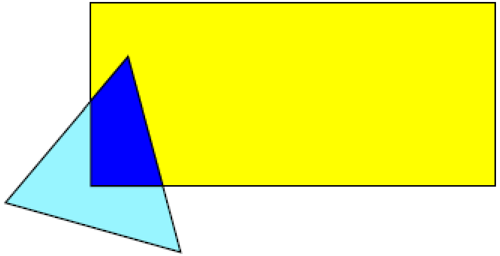
Creating new triangles

- With **two intersection points** we can clip triangles using a hyperplane:
  - If **two vertices** are outside the hyperplane we create a **new triangle**
  - If **one vertex** is outside of the hyperplane we create **two new triangles**



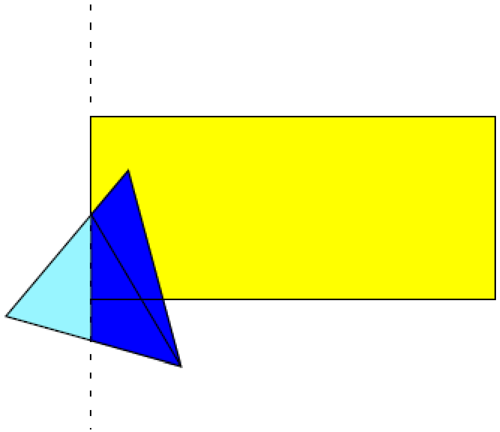
Creating new triangles

- But what if a triangle is clipped by two hyperplanes?



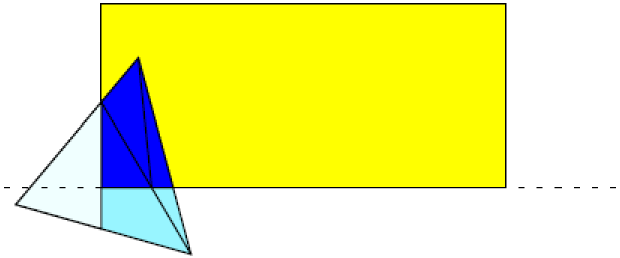
Creating new triangles

- First we clip according to the first hyperplane



Creating new triangles

- Then the second hyperplane



$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

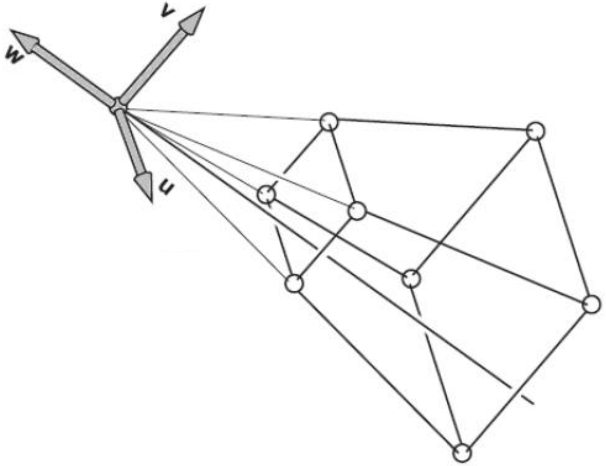
Perspective transform

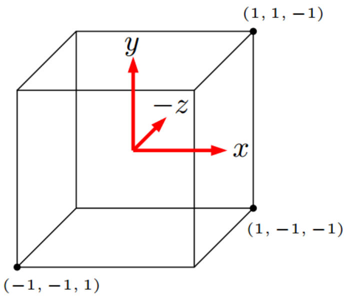
$$\begin{bmatrix} x' \\ y' \\ z' \\ w \end{bmatrix}$$

Homogenization

$$\begin{bmatrix} x'/w \\ y'/w \\ z'/w \\ 1 \end{bmatrix}$$

Rasterization

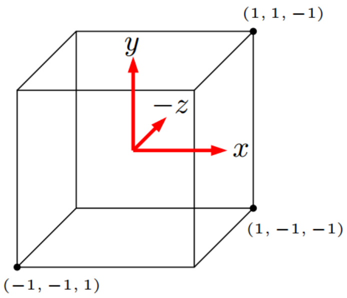




Clipping after homogenization

- Simple equations for the hyperplanes:

$$
\begin{aligned}
-x + l &= 0 \\
x - r &= 0 \\
-y + b &= 0 \\
y - t &= 0 \\
-z + n &= 0 \\
z - f &= 0
\end{aligned}
$$

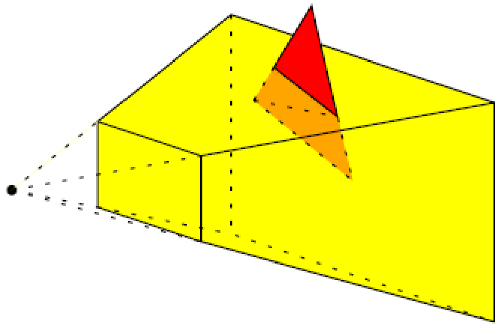


Problem with the XY plane

$$z_p = n + f - \frac{fn}{z} \quad \rightarrow \quad z_p \sim \frac{1}{z}$$

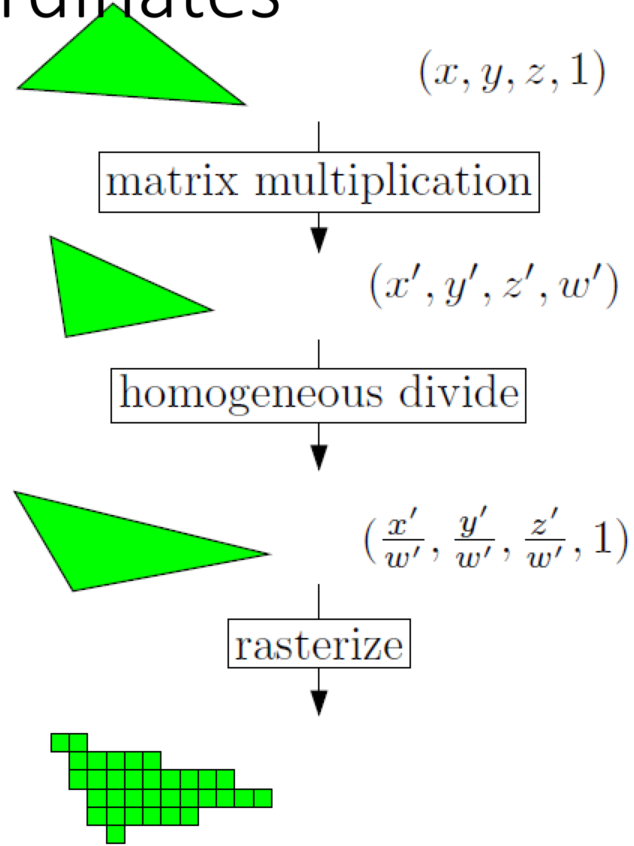
Clipping before homogenization

- Vertices of the view frustum can be obtained from the transformation matrix M_{per}^{-1}
- Then we can deduce the equations of the hyperplanes



Clipping in homogeneous coordinates

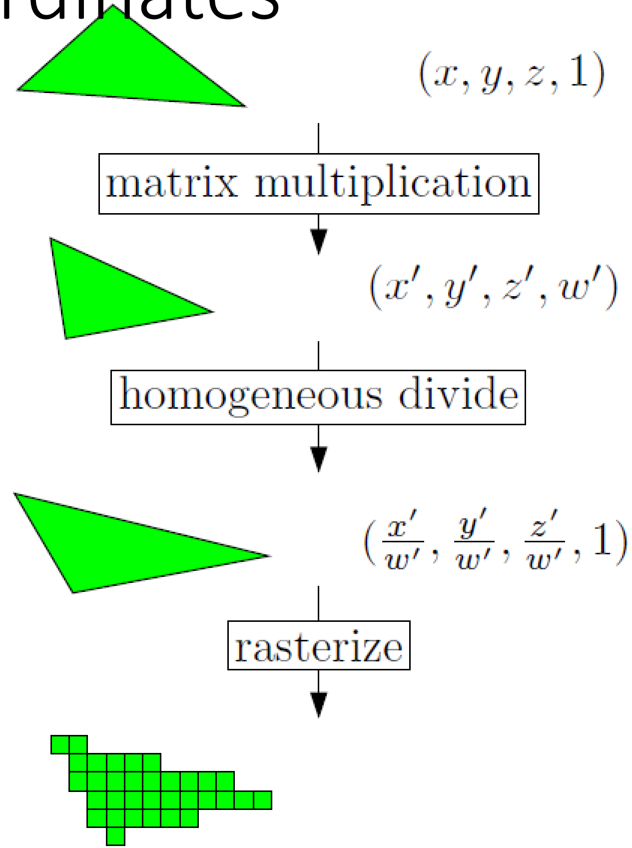
- It turns out that **clipping** is most convenient in **homogeneous coordinates**. This means: we clip triangles in **4 dimensions** using **3D hyperplanes**.

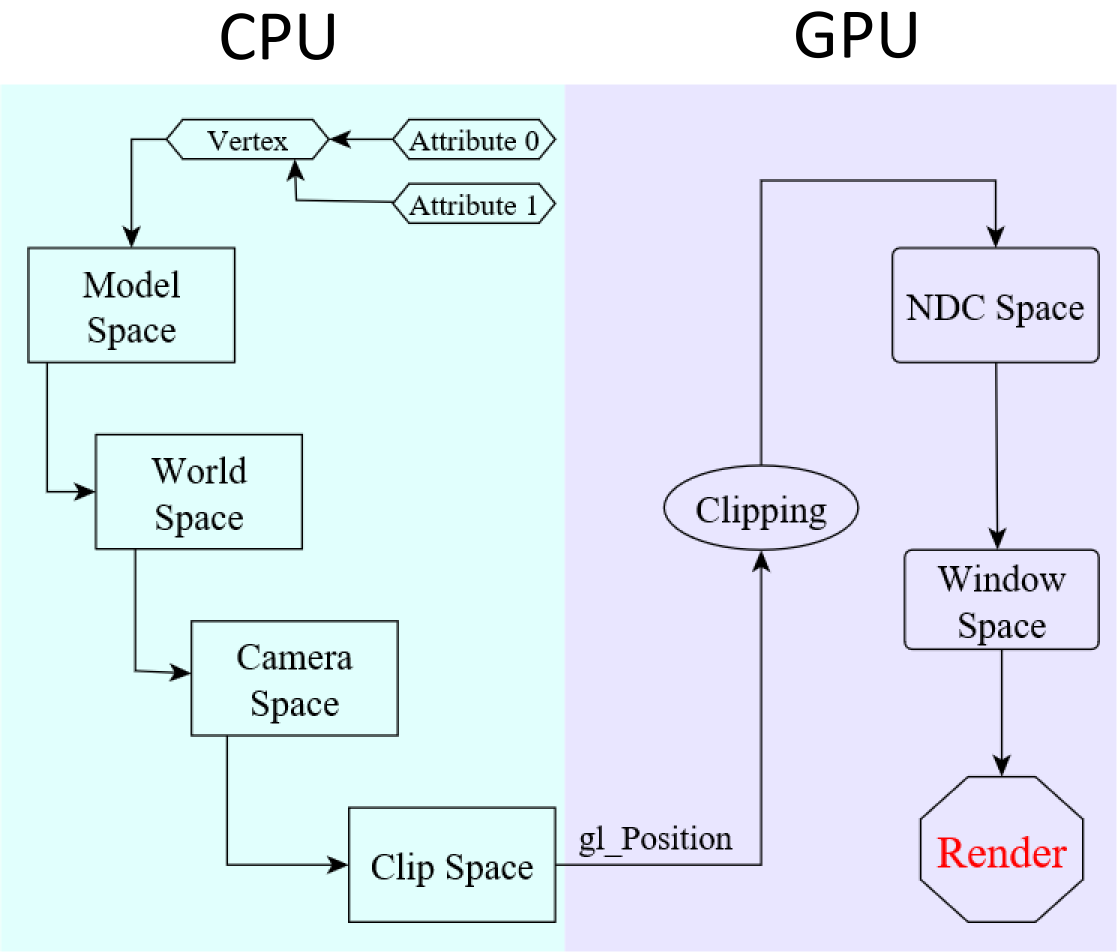


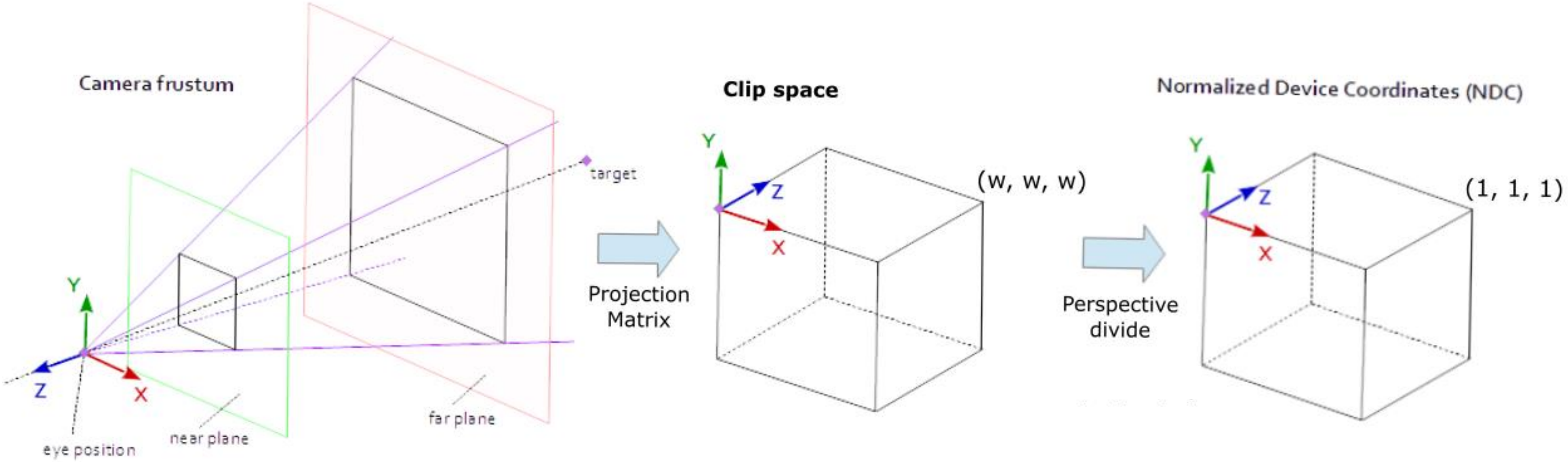
Clipping in homogeneous coordinates

- It turns out that **clipping** is most convenient in **homogeneous coordinates**. This means: we clip triangles in **4 dimensions** using **3D hyperplanes**.

$$
\begin{aligned}
-x' + lw' &= 0 \\
x' - rw' &= 0 \\
-y' + bw' &= 0 \\
y' - tw' &= 0 \\
-z' + nw' &= 0 \\
z' - fw' &= 0
\end{aligned}
$$

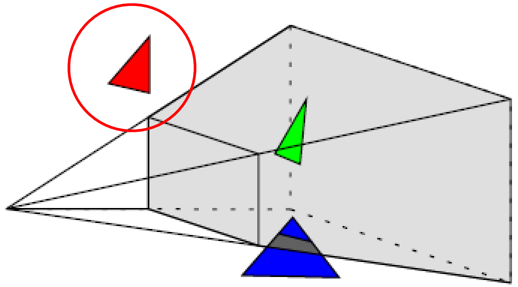



CPU
GPU
Vertex
Attribute 0
Attribute 1
Model Space
World Space
Camera Space
Clip Space
gl_Position
Clipping
NDC Space
Window Space
Render


Camera frustum
Y
Z
X
target
eye position
near plane
far plane
Projection
Matrix
Clip space
Y
Z
X
(w, w, w)
Perspective
divide
Normalized Device Coordinates (NDC)
Y
Z
X
(1, 1, 1)

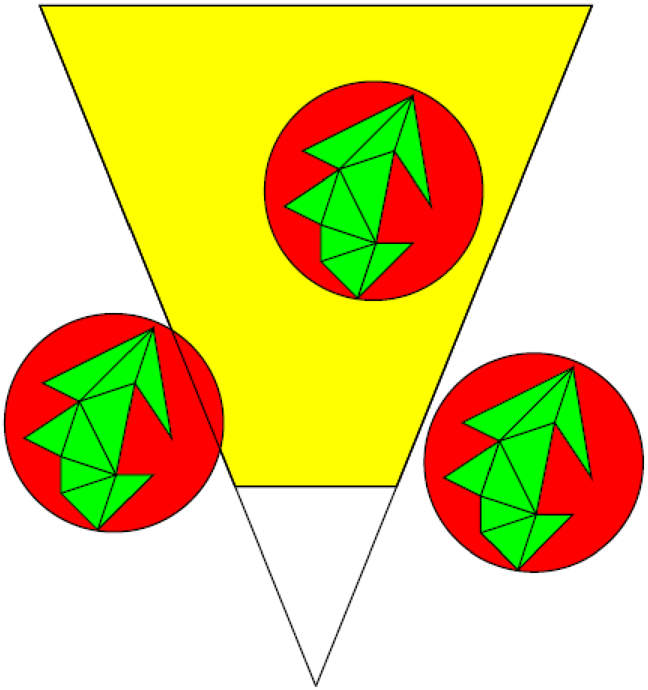
Culling

- If a triangle lies outside of the view frustum we remove it completely
- Testing vertices is costly…



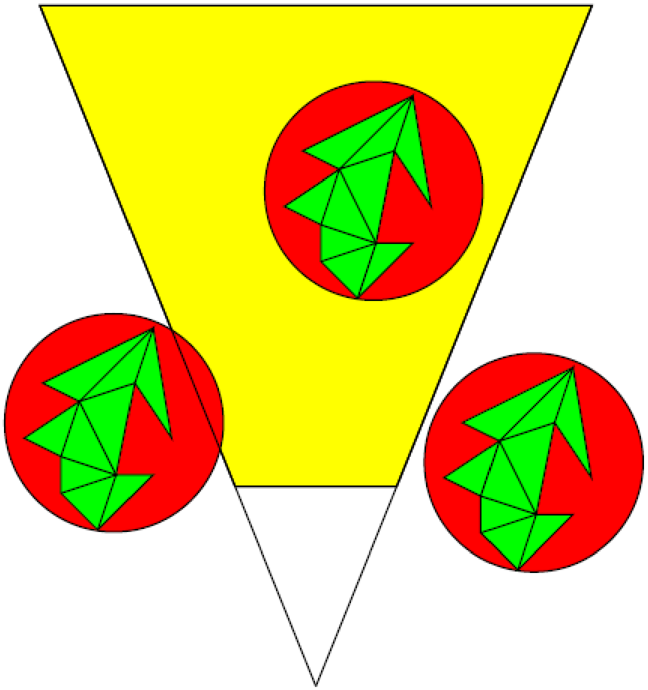
Bounding volumes

- Using bounding volumes for complex geometric objects accelerates the rendering pipeline



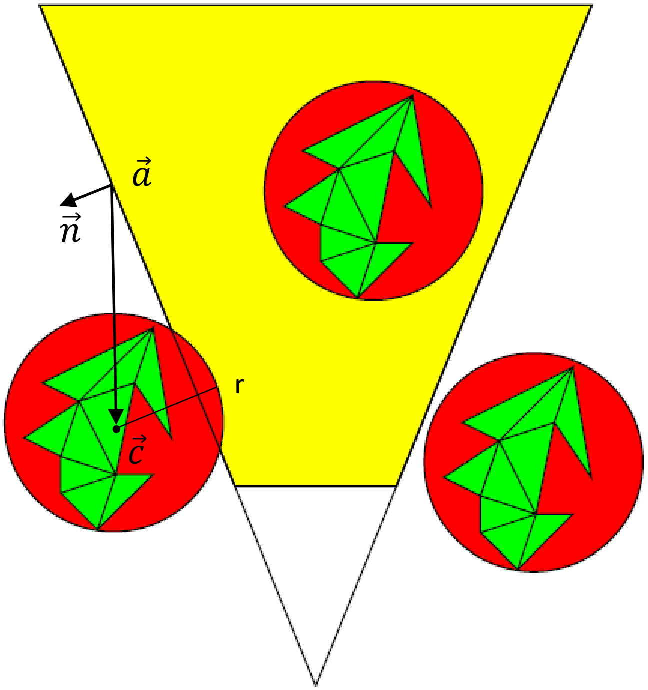
Bounding volumes

- Spheres are often used BV
- If a plane is given by
- $(\vec{p} - \vec{a}) \cdot \vec{n} = 0$
- And the sphere has center $\vec{c}$ and radius r, we test the inequality
- $\frac{(\vec{c}-\vec{a})\cdot\vec{n}}{\|n\|} > r$



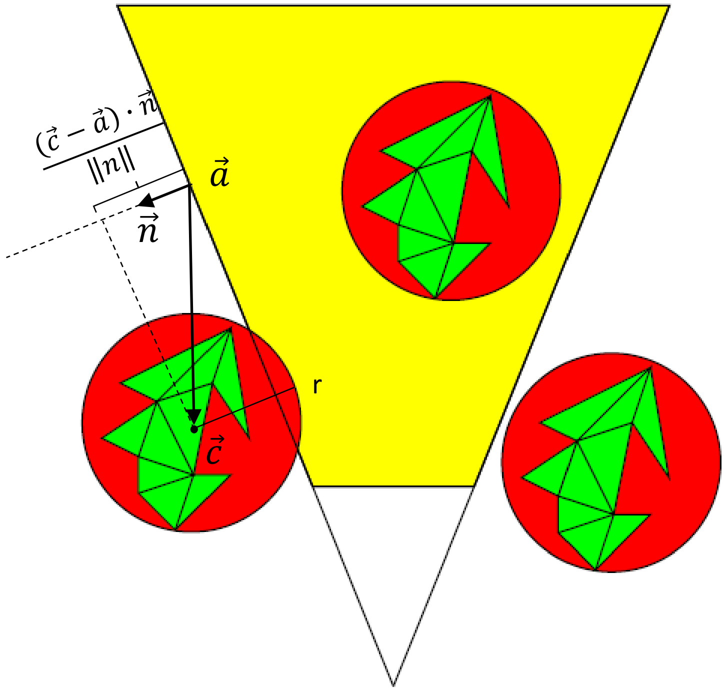
Bounding volumes

- Spheres are often used BV
- If a plane is given by
- $(\vec{p} - \vec{a}) \cdot \vec{n} = 0$
- And the sphere has center $\vec{c}$ and radius r, we test the inequality
- $\frac{(\vec{c} - \vec{a}) \cdot \vec{n}}{\|n\|} > r$



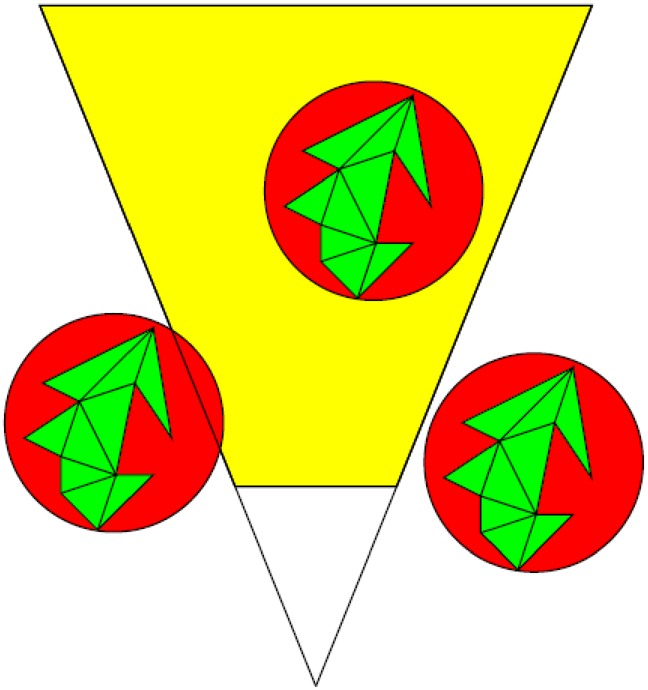
Bounding volumes

- Spheres are often used BV
- If a plane is given by
- $(\vec{p} - \vec{a}) \cdot \vec{n} = 0$
- And the sphere has center $\vec{c}$ and radius r, we test the inequality
- $\frac{(\vec{c}-\vec{a})\cdot\vec{n}}{\|n\|} > r$



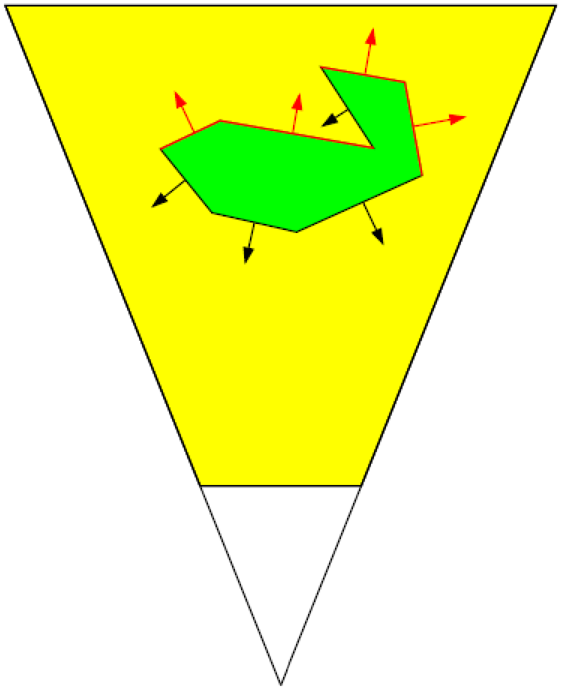
Culling

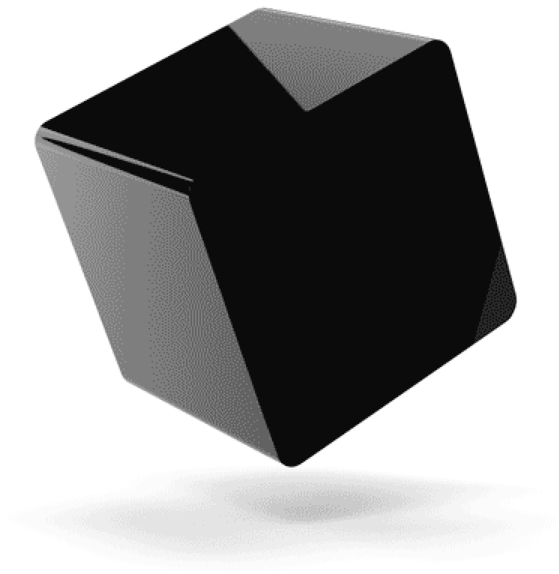
- **Frustum culling**
  removing triangle outside of the view frustum
- **Backface culling**
  removing triangles oriented away from the camera

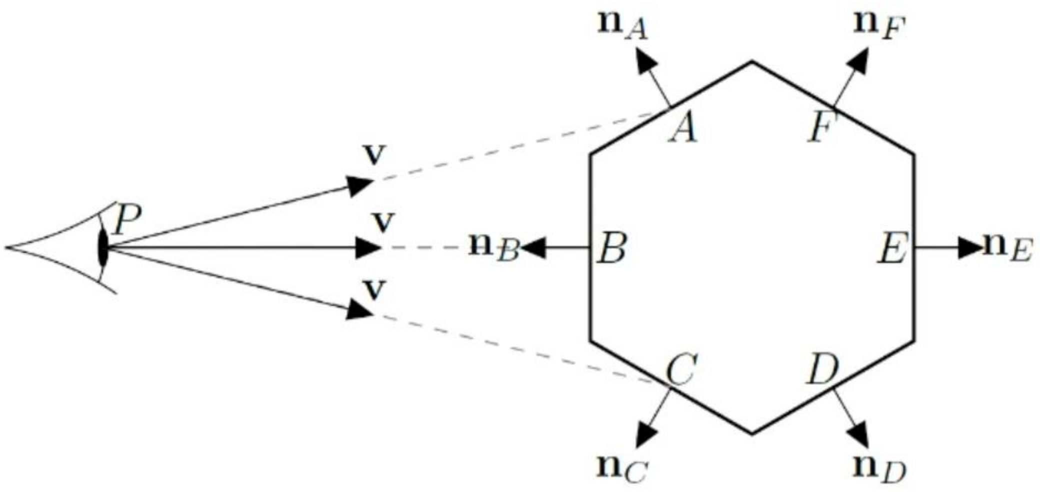


Backface culling

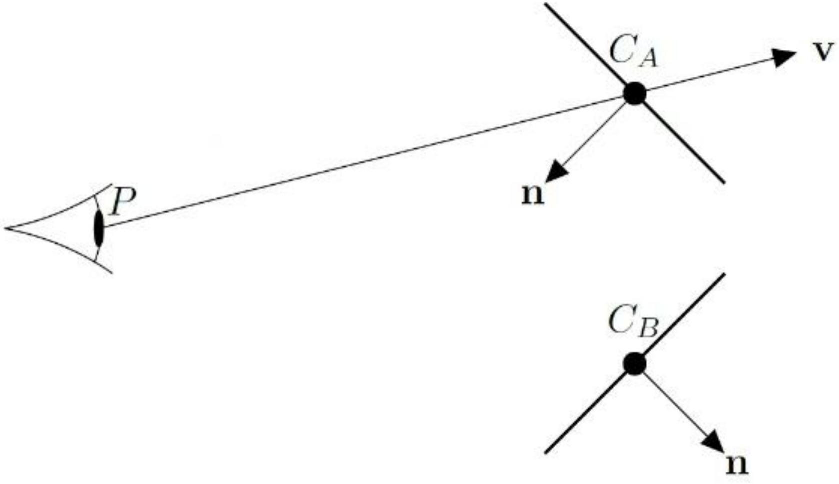
- If we model geometric objects with triangles the **normals** are directed **outside** of the object
- Removing triangles whose normals is directed away **from the view point** we call **backface culling**



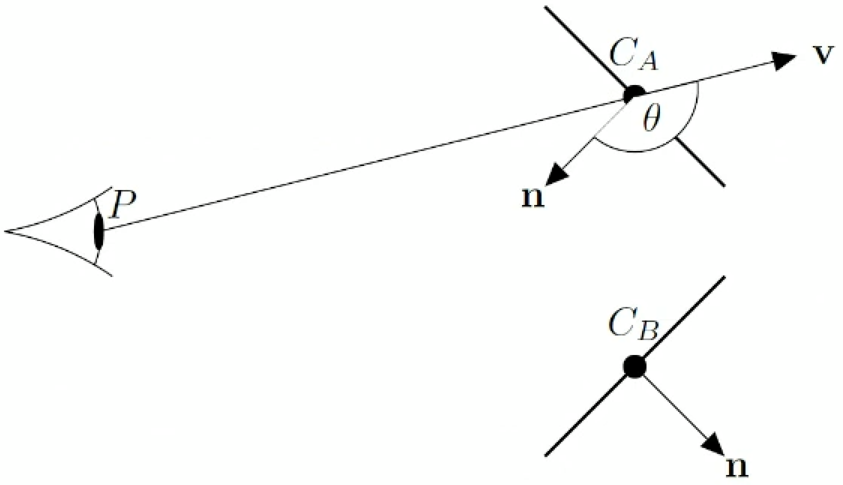


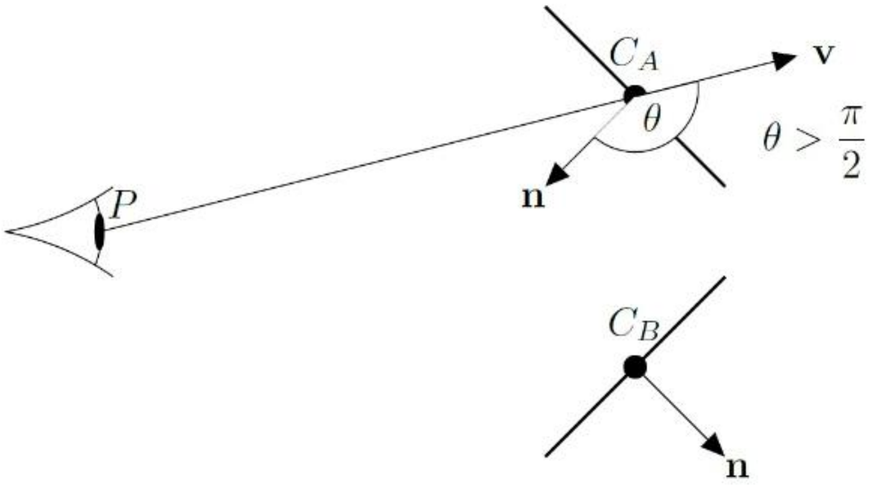


$\mathbf{n}_A$
$\mathbf{n}_F$
A
F
$\mathbf{v}$
P
$\mathbf{v}$
$\mathbf{n}_B$
B
E
$\mathbf{n}_E$
$\mathbf{v}$
C
D
$\mathbf{n}_C$
$\mathbf{n}_D$

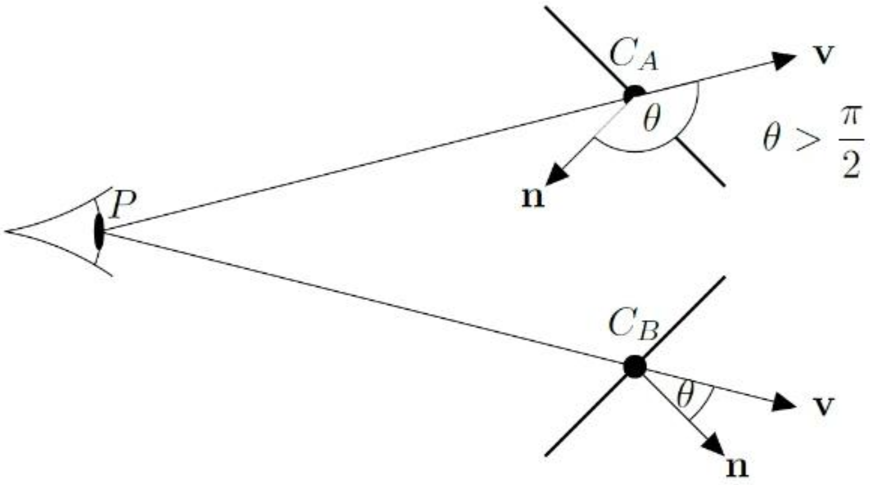


C_A
v
n
P
C_B
n

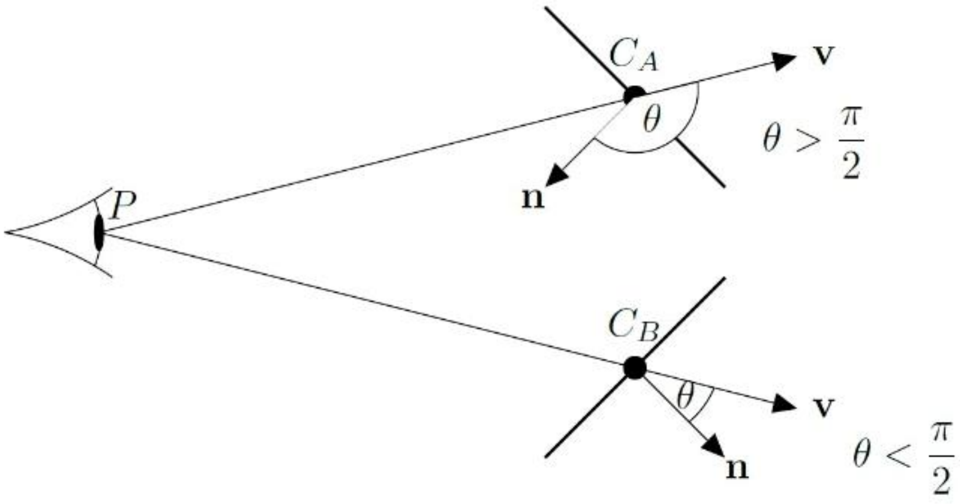

C_A
$\mathbf{v}$
θ
$\mathbf{n}$
P
C_B
$\mathbf{n}$

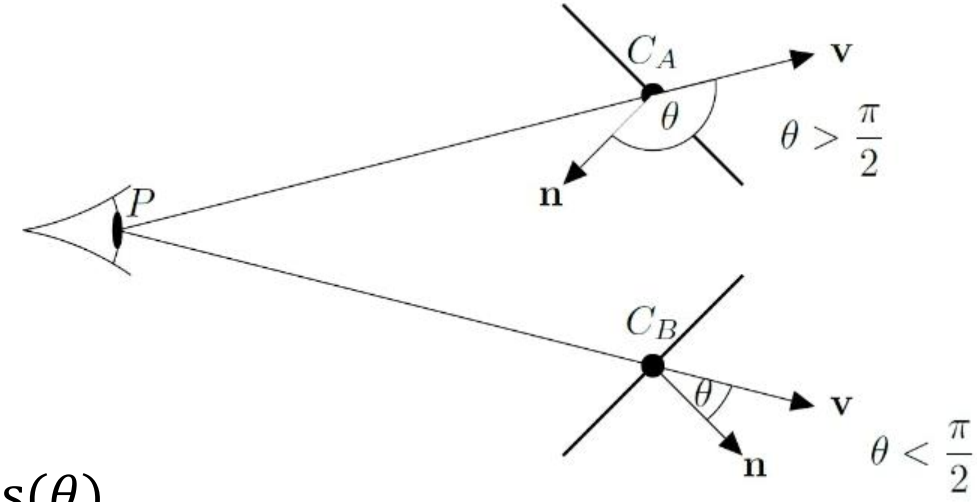


C_A
v
θ
$\theta > \frac{\pi}{2}$
n
P
C_B
n

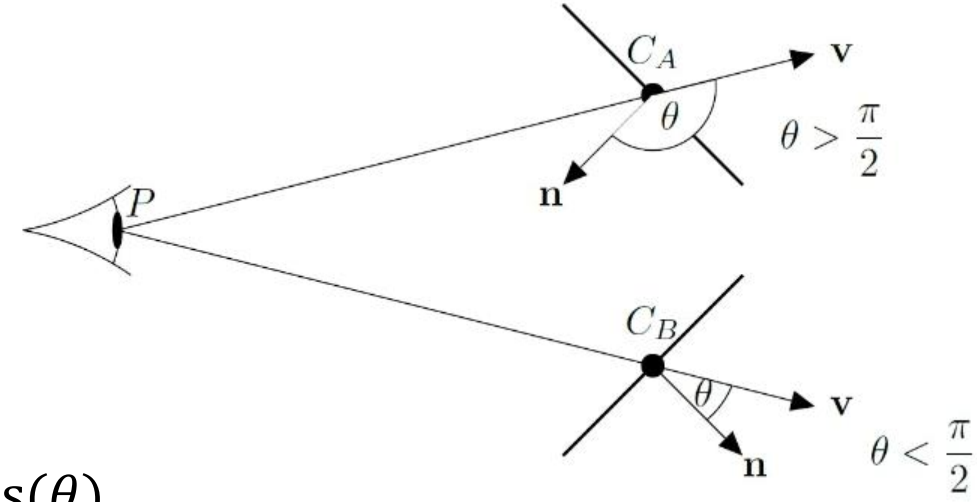


C_A
v
θ
$\theta > \frac{\pi}{2}$
n
P
C_B
θ
v
n


C_A
$\mathbf{v}$
θ
$\theta > \frac{\pi}{2}$
$\mathbf{n}$
P
C_B
θ
$\mathbf{v}$
$\theta < \frac{\pi}{2}$
$\mathbf{n}$



- $\boldsymbol{v} \cdot \boldsymbol{n} = |\boldsymbol{v}||\boldsymbol{n}|\cos(\theta)$



- $\boldsymbol{v} \cdot \boldsymbol{n} = |\boldsymbol{v}||\boldsymbol{n}|\cos(\theta)$

- Front facing if $\theta > \frac{\pi}{2}$, or $\cos(\theta) < 0$ and
  $\boldsymbol{v} \cdot \boldsymbol{n} < 0$

Algorithm Back face culling

Require: Vertex co-ordinates of polygons and an viewpoint P

for all polygons in the virtual world **do**

  calculate the normal vector $\mathbf{n}$ of the current polygon

  calculate the centre C of the current polygon

  calculate the viewing vector $\mathbf{v} = C - P$

  if $\mathbf{v} \cdot \mathbf{n} < 0$ **then**

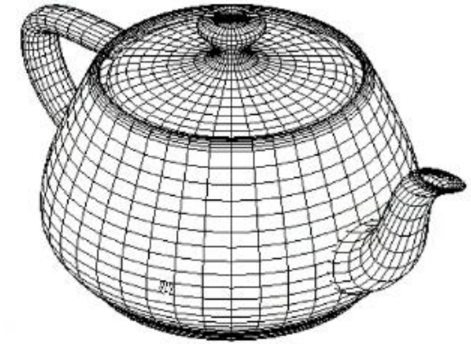
    render current polygon

  end if

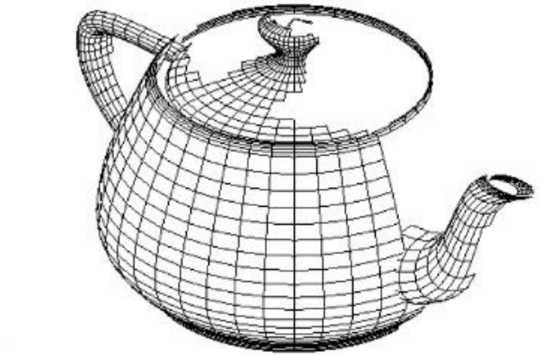
end for

All polygons

backface culling



All polygons



backface culling

Przykład

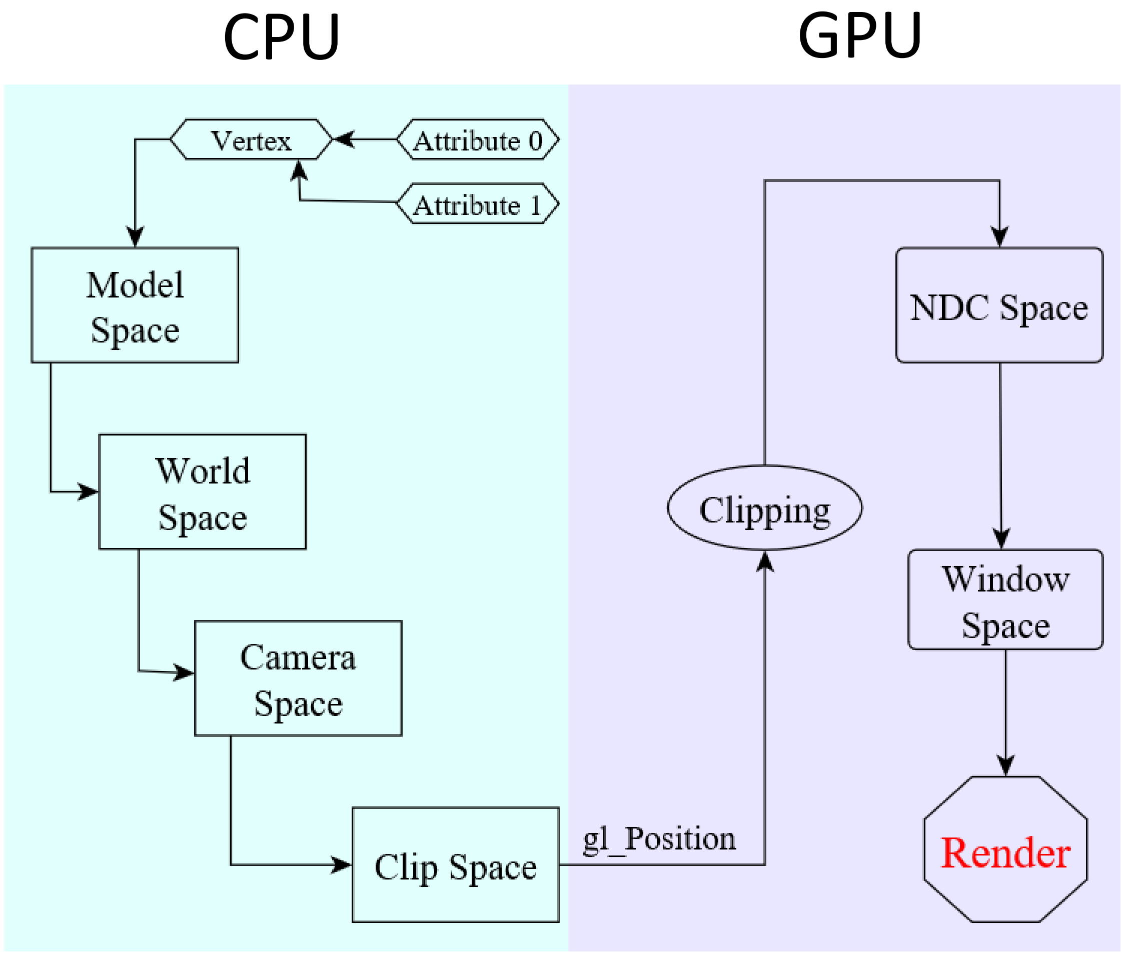
No backface culling Backface culling

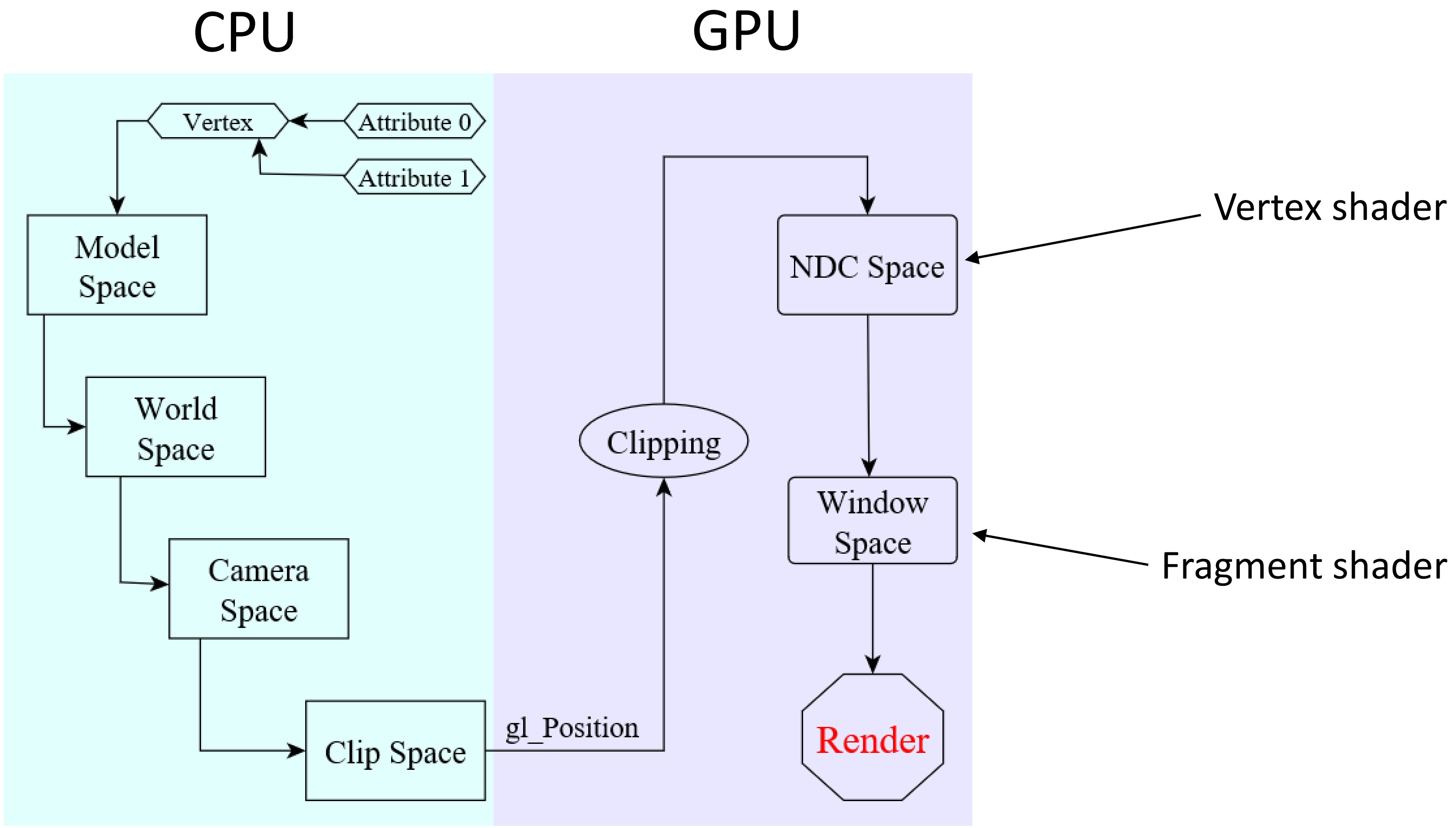
OpenGL culling

```
void glCullFace(GLenum mode);
```

```
GL_FRONT, GL_BACK
```

By default turned off.


CPU
GPU
Vertex
Attribute 0
Attribute 1
Model Space
World Space
Camera Space
Clip Space
gl_Position
Clipping
NDC Space
Window Space
Render


CPU
GPU
Vertex
Attribute 0
Attribute 1
Model Space
World Space
Camera Space
Clip Space
gl_Position
Clipping
NDC Space
Window Space
Render
Vertex shader
Fragment shader