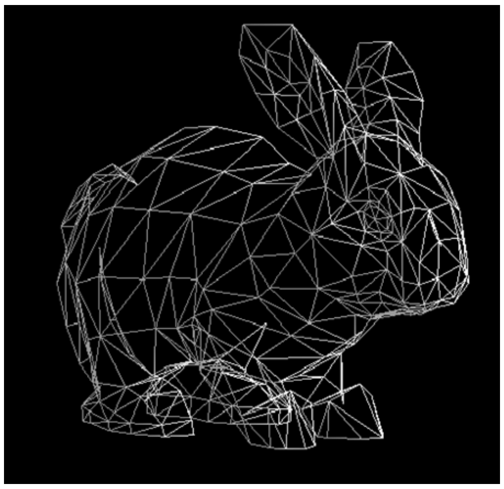
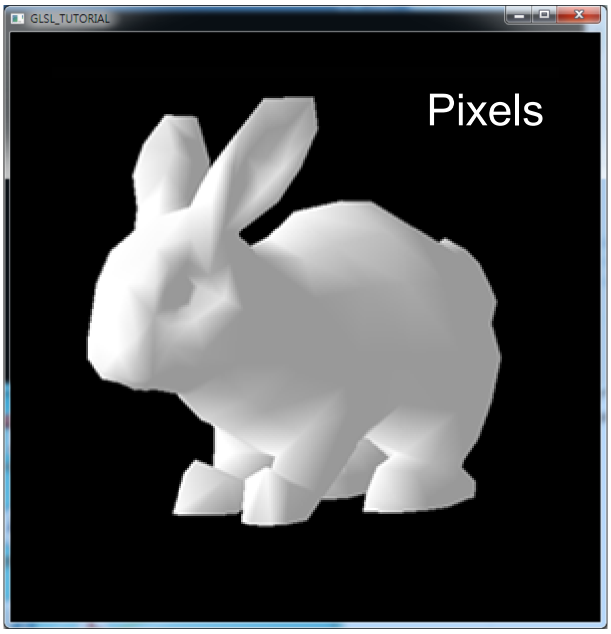
From Vertex to Pixel

Chapter 12

Chapter 1 (Lecture 4)

- Chapter 1 (Lecture 4)
  - Vertices

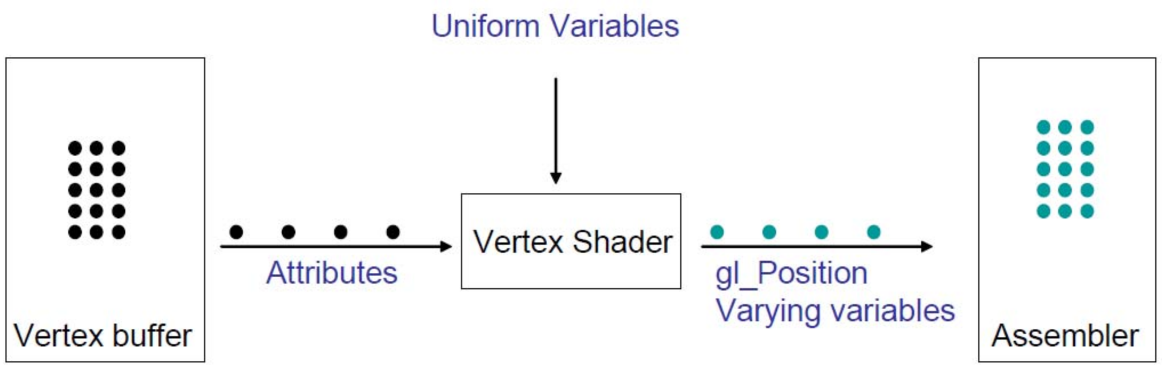




Vertex Shader

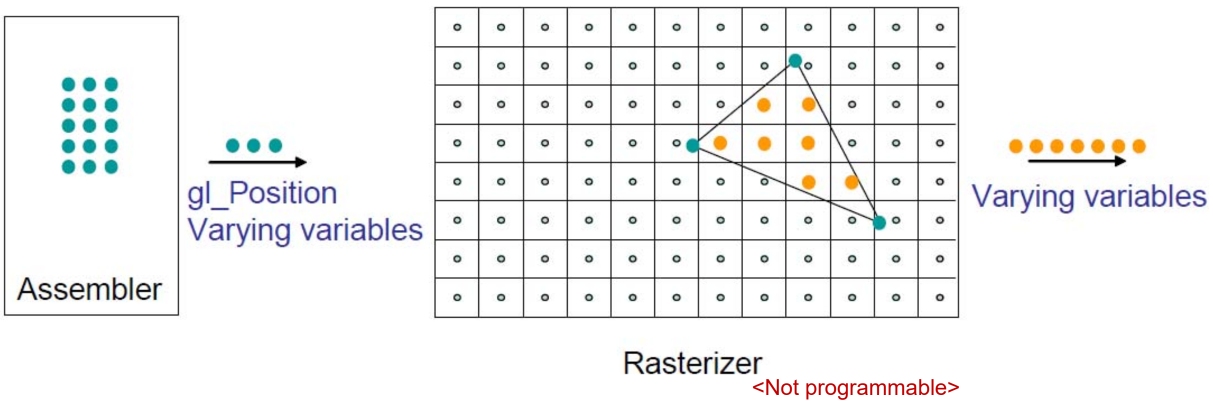
- Vertices are stored in a vertex buffer.
- When a draw call is issued, each of the vertices passes through the vertex shader.
- On input to the vertex shader, each vertex (black) has associated attributes.
- On output, each vertex (cyan) has a value for gl_Position and for its varying variables.

Uniform variables are set by your program, but you can only set them in between OpenGL draw calls and not per vertex.
(e.g., virtual camera parameters)



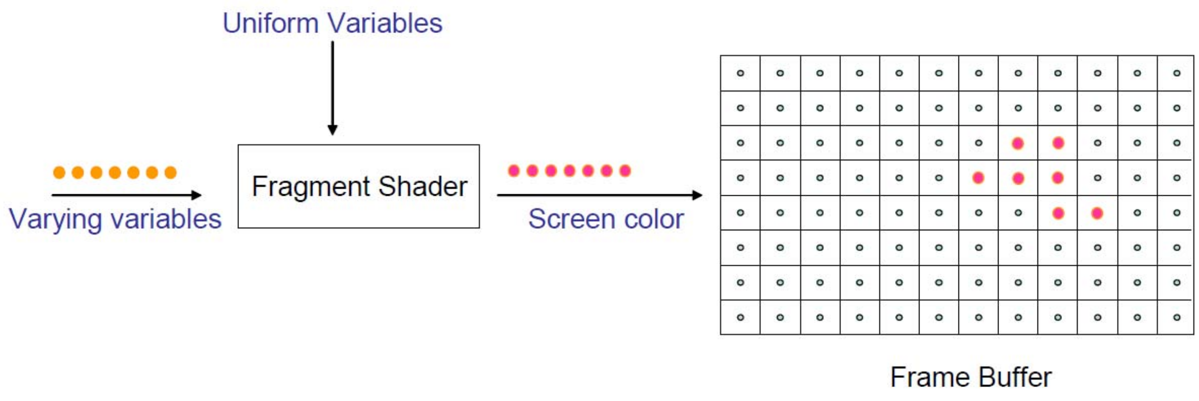
In the pipeline: Rasterization

- The data in gl_Position is used to place the three vertices of the triangle on a virtual screen.
- The rasterizer figures out which pixels (orange) are *inside* the triangle and interpolates the varying variables from the vertices to each of three pixels.



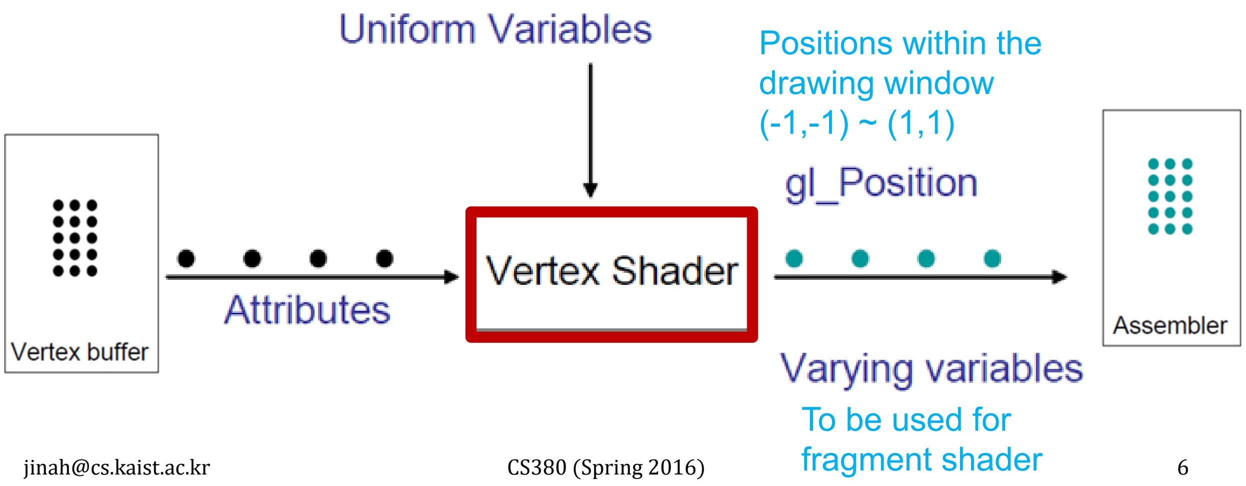
Fragment Shader

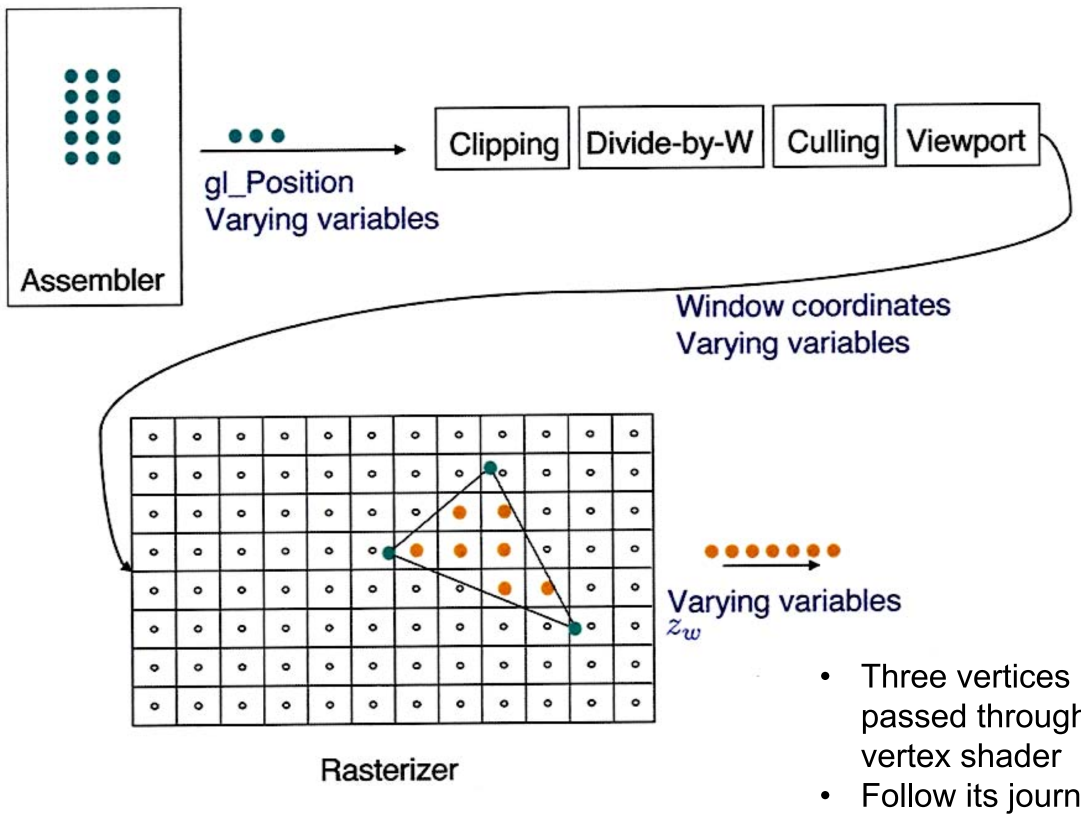
- Each pixel (orange) is passed through the fragment shader, which computes the final color of the pixel (pink).
- The pixel is then placed in the frame buffer for display.



Chapter 2~6, 10

- Transformation
- Virtual camera
  - Mapping 3D coordinates to the actual 2D screen





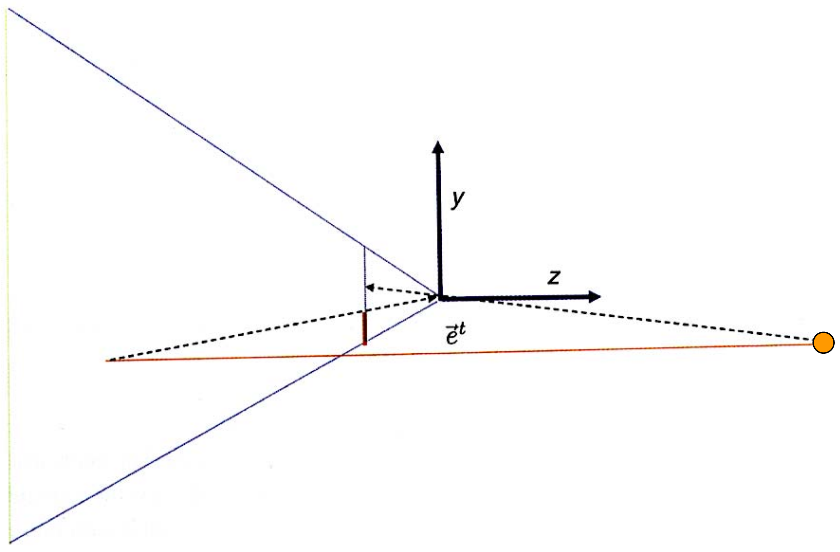
- Three vertices have passed through the vertex shader
- Follow its journey to be a bunch of pixels

Varying Variables

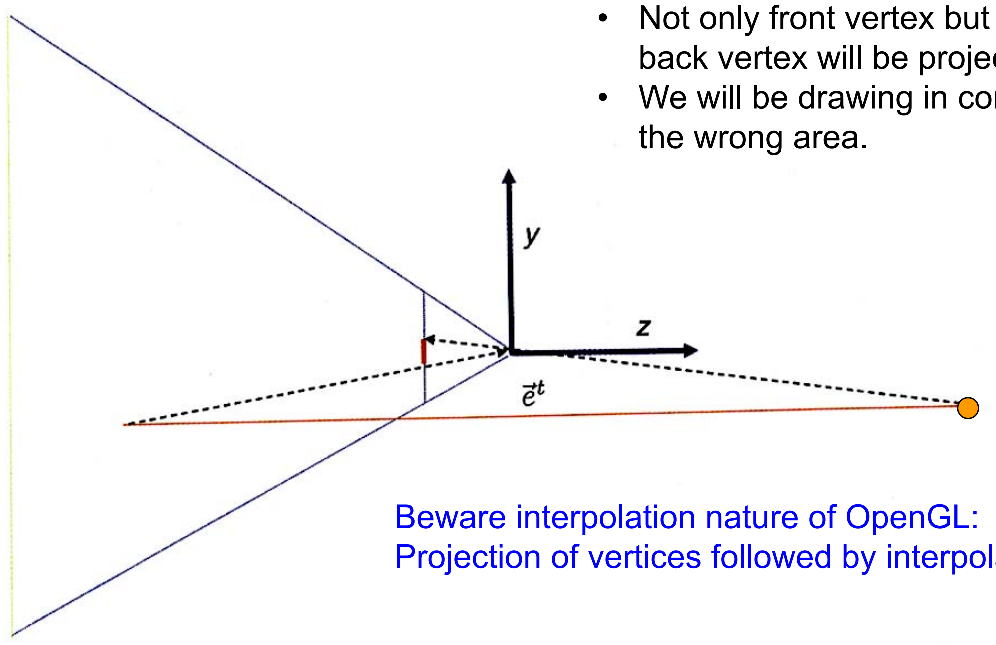
- Varying variables provides an interface between the vertex and fragment shader.
- When the primitives are assembled and fragments computed, for each fragment there is a set of variables that are interpolated automatically and provided to the fragment shader.
- An example is the color of the fragment. The color that arrives at the fragment shader is the result of the interpolation of the colors of the vertices that make up the primitive.
  - varying float intensity;

Clipping

- What if there is a vertex behind you?



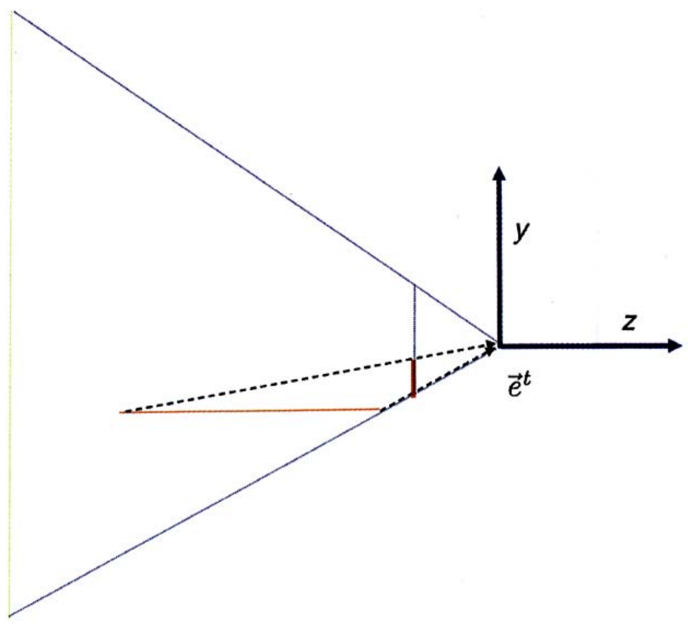
Clipping



- Not only front vertex but also back vertex will be projected
- We will be drawing in completely the wrong area.

Beware interpolation nature of OpenGL: Projection of vertices followed by interpolation

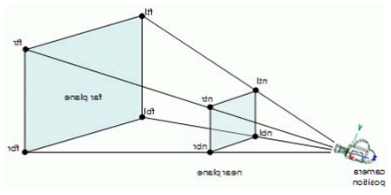
Clipping

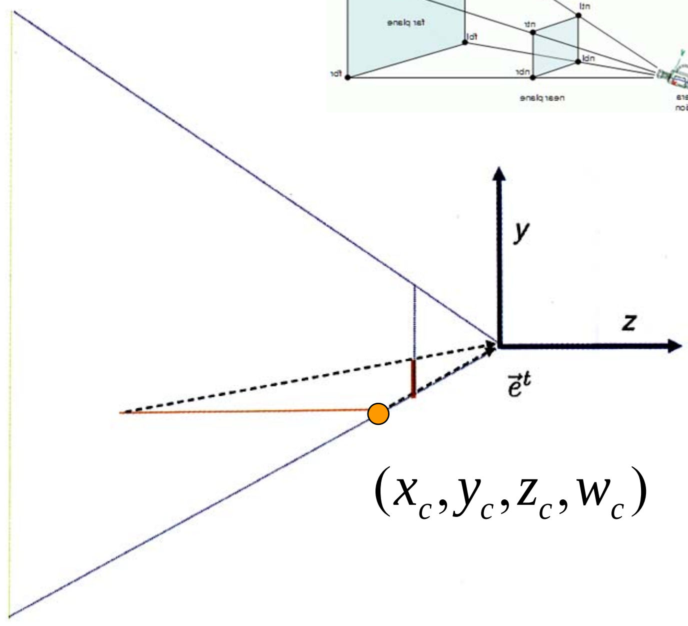


- Processing for triangles that are fully or partially out of view
- We don't want to see behind us
- We want to minimize processing

- The tricky part will be to deal with eye-spanning triangles.

Eye spanners





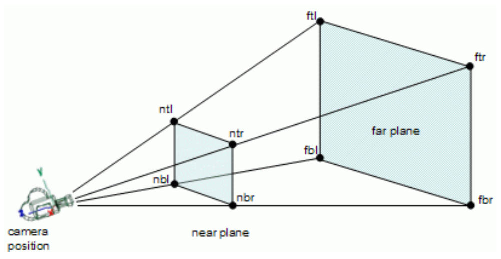
(x_c, y_c, z_c, w_c)

- Back vertex projects higher up in the image
- Filling in the in-between pixels will fill in the wrong region.
- Solution: slice up the geometry by the six faces of the view frustum
- → Clipping

Clipping coordinates

- Eye coordinates (projected) → clip coordinates → normalized device coordinates (NDCs)
- (reminder) Dividing clip coordinates (x_c, y_c, z_c, w_c) by the $w_c (w_c = w_n)$ component (the fourth component in the homogeneous coordinates) yields normalized device coordinates (NDCs).

$$\begin{bmatrix} x_n w_n \\ y_n w_n \\ z_n w_n \\ w_n \end{bmatrix} = \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix} = \begin{bmatrix} s_x & 0 & -c_x & 0 \\ 0 & s_y & -c_y & 0 \\ 0 & 0 & \frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix}$$



recap

- Point $\tilde{p}$ in eye coordinates $[x_e, y_e, z_e]^t$
- After projection (pinhole camera) $[x_e, y_e, z_e]^t = -z_e[x_n, y_n, -1]^t$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ - & - & - & - \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ 1 \end{bmatrix} = \begin{bmatrix} x_c \\ y_c \\ - \\ w_c \end{bmatrix} = \begin{bmatrix} x_n w_n \\ y_n w_n \\ - \\ w_n \end{bmatrix}$$

- $[x_c, y_c, -, w_c]^t$ clip coordinates of $\tilde{p}$
- $w_n = w_c$: a new variable called the w-coordinate
- Divide by w. (n refers to 'normalized device coordinates)

$$x_n = \frac{x_n w_n}{w_n}$$

- $-1 \leq x_n \leq +1, \ -1 \leq y_n \leq +1$ *canonical square*

→ will be mapped onto a window on the screen.

Clipping coordinates

- Suppose that x_c / w_c is the same as $-x_c / -w_c$.
- If you wait for normalized device coordinates (NDCs) where the vertex has flipped, and it's too late to do the clipping.
- We could do in the eye space, but then would need to use the camera parameters
- The solution is to <u>use clip coordinates</u>: post-matrix-multiply but pre-divide.
- No divide = no flipping

Clipping coordinates

- Recall that we want points in the range:

$$-1 < x_n < 1$$
$$-1 < y_n < 1$$
$$-1 < z_n < 1$$

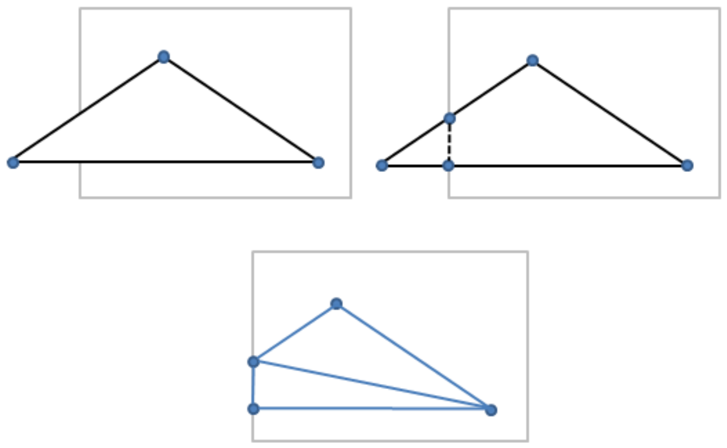
- In clip coordinates this is:

$$-w_c < x_c < w_c$$
$$-w_c < y_c < w_c$$
$$-w_c < z_c < w_c$$

$$\begin{bmatrix} x_n w_n \\ y_n w_n \\ z_n w_n \\ w_n \end{bmatrix} = \begin{bmatrix} x_c \\ y_c \\ z_c \\ w_c \end{bmatrix}$$

Clipping coordinates

- Primitives totally inside the clipping volume are not altered. Primitives outside the viewing volume are discarded. Primitives whose edges intersect the boundaries of the clipping volume are clipped.



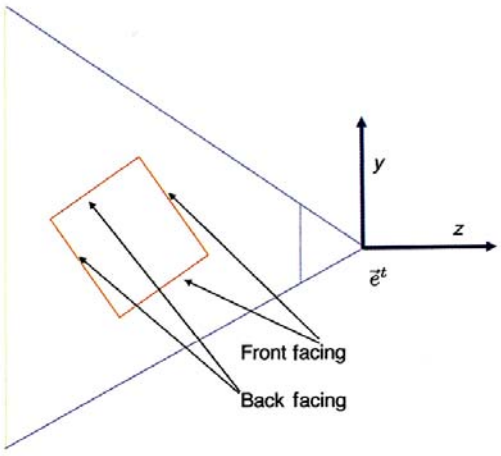
Clipping coordinates → NDCs

- Clipping is done,
  we can now divide by w_c
  to obtain normalize device coordinates.

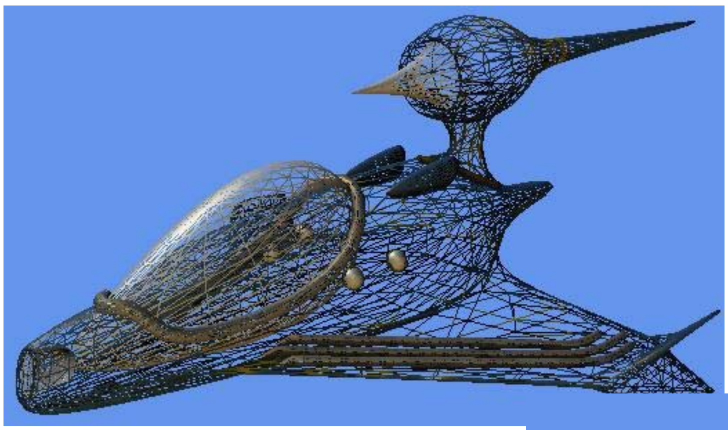
$$\begin{bmatrix} x_n \\ y_n \\ z_n \\ 1 \end{bmatrix} = \begin{bmatrix} x_c / w_c \\ y_c / w_c \\ z_c / w_c \\ w_c / w_c \end{bmatrix}$$

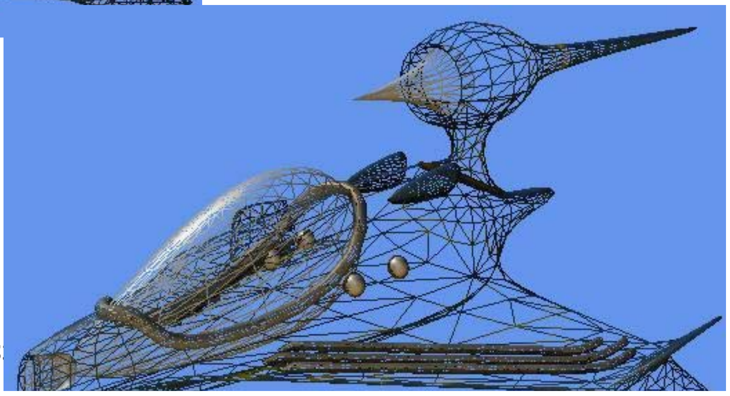
Backface culling

- When drawing a closed solid object, we will only ever se e one ‘front’ side of each triangle.
- For efficiency we can drop these from the processing.



Backface culling





Backface culling

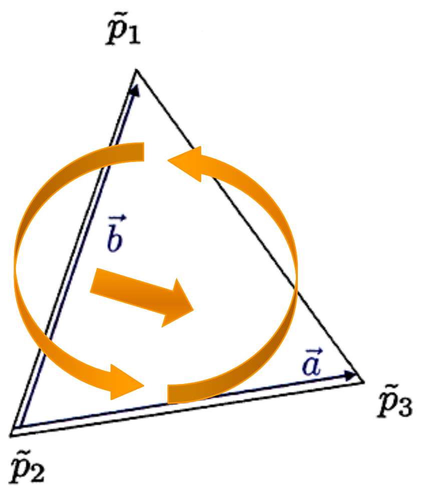
- To do this, in OpenGL, we use the convention of ordering the three vertices in the draw call so that they are counterclockwise (CCW) when looking at its front side.
- During setup, we call glEnable(GL_CULL_FACE)
- To implement culling, OpenGL does the following…

Math of Backface Culling

- Let $\tilde{p}_1, \tilde{p}_2, \text{ and } \tilde{p}_3$ be the three vertices of the triangle projected down to the $(x_n, y_n, 0)$ plane.
- Define the vector $\vec{a} = \tilde{p}_3 - \tilde{p}_2$ and $\vec{b} = \tilde{p}_1 - \tilde{p}_2$
- Next compute the cross product

$$\vec{c} = \vec{a} \times \vec{b}$$

- If the three vertices are counterclockwise in the plane, then $+z_n$ will be in the $\vec{c}$ direction.



- So the area of the triangle is:

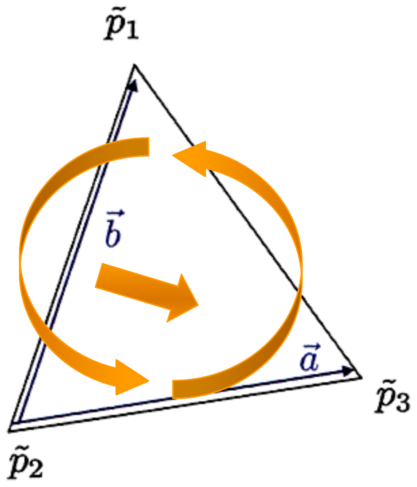
$$Area = \frac{1}{2}\left\|\vec{a} \times \vec{b}\right\|$$

Math of Backface Culling

- Taking account of the direction, we could calculate the direction of the cross product of these two vectors.

  $(x_n^3 - x_n^2)(y_n^1 - y_n^2) - (y_n^3 - y_n^2)(x_n^1 - x_n^2)$

- If this is negative, the polygon looks backward.

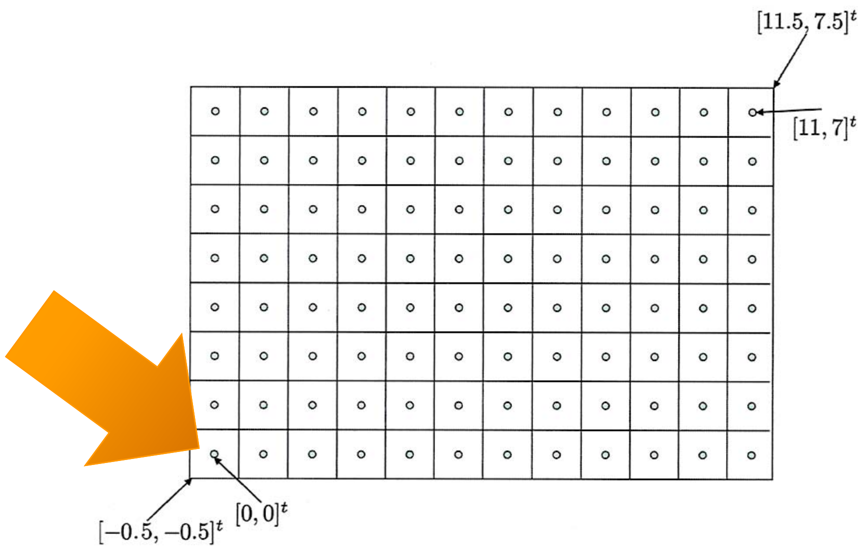


Viewport

- Now we want to position the vertices in the window. So it is time to move the NDCs to window coordinates.
  - in NDCs, lower left corner is $[-1, -1]^t$ and upper right corner is $[1,1]^t$.
- Each pixel center has an integer coordinate.
  - This will make subsequent pixel computations more natural.
- We want the lower left pixel center to have 2D window coordinates of $[0,0]^t$ and the upper right pixel center to have coordinates $[W-1, H-1]^t$

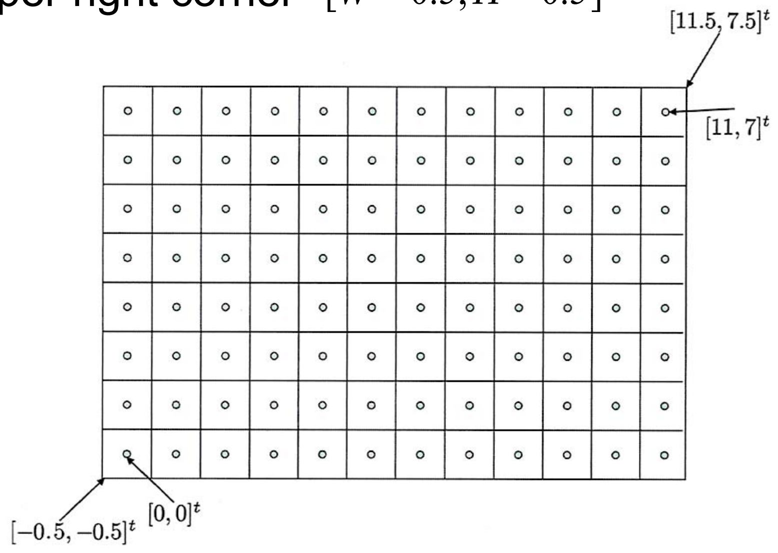
Viewport

- We think of each pixel as owning the real estate which extends 0.5 pixel units in the positive and negative, horizontal and vertical directions from the pixel center.



Viewport

- Thus the extent of 2D window rectangle covered by the union of all our pixels is the rectangle in window coordinates with lower left corner $[-0.5,-0.5]^t$ and upper right corner $[W-0.5, H-0.5]^t$



Viewport matrix

- We need a transform that maps the lower left corner to $[-0.5,-0.5]^t$ and upper right corner to $[W-0.5,H-0.5]^t$

- The appropriate scale and shift can be done using the viewport matrix:

$$\begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} = \begin{bmatrix} W/2 & 0 & 0 & (W-1)/2 \\ 0 & H/2 & 0 & (H-1)/2 \\ 0 & 0 & 1/2 & 1/2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_n \\ y_n \\ z_n \\ 1 \end{bmatrix}$$

Viewport matrix

- This does a scale and shift in both *x* and *y*
- You can verify that it maps the corners appropriately
- In OpenGL, we set up this viewport matrix with the call glViewport(0,0,W,H)
- The third row of this matrix is used to map the $[-1,-1]$ range of z_n values to the more convenient $[0...1]$ range.
- So now (in our conventions), $z_w = 0$ is far and $z_w = 1$ is near.

Viewport matrix

- So we must also tell OpenGL that when we clear the z-buffer, we should set it to 0;
  we do this with the call glClearDepth(0.0)

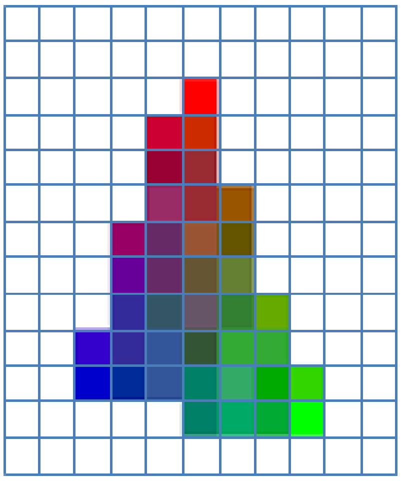
Texture Viewport

- The abstract domain for textures is not the canonical square, but instead is the unit square.
- Its lower left corner is $[0,0]^t$ and upper right corner is $[1,1]^t$.
- In this case the coordinate transformation matrix is:

$$\begin{bmatrix} x_w \\ y_w \\ - \\ 1 \end{bmatrix} = \begin{bmatrix} W & 0 & 0 & -1/2 \\ 0 & H & 0 & -1/2 \\ - & - & - & - \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_t \\ y_t \\ - \\ 1 \end{bmatrix}$$

Rasterization

- Starting from the window coordinates for the three vertices, the rasterizer needs to figure out which pixel centers are inside of the triangle.
- Each triangle on the screen can be defined as the intersection of three half-spaces.

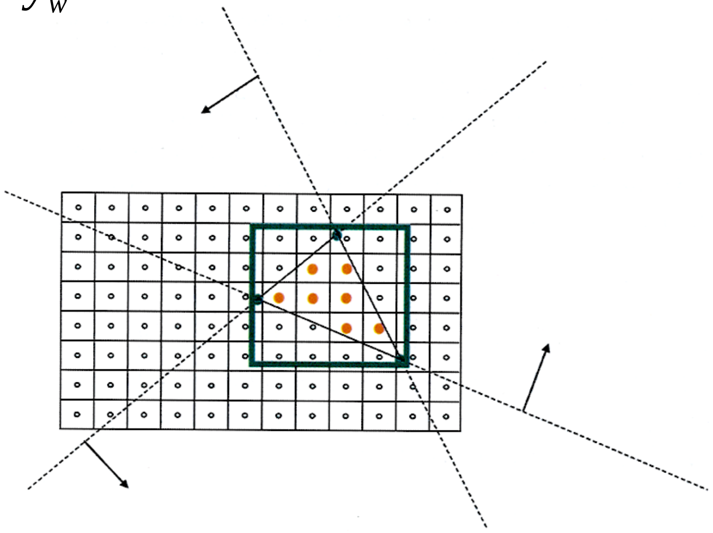


Math of rasterization

- Each such halfspace is defined by a line that coincides with one of the edges of the triangle, and can be tested using an 'edge function' of the form:

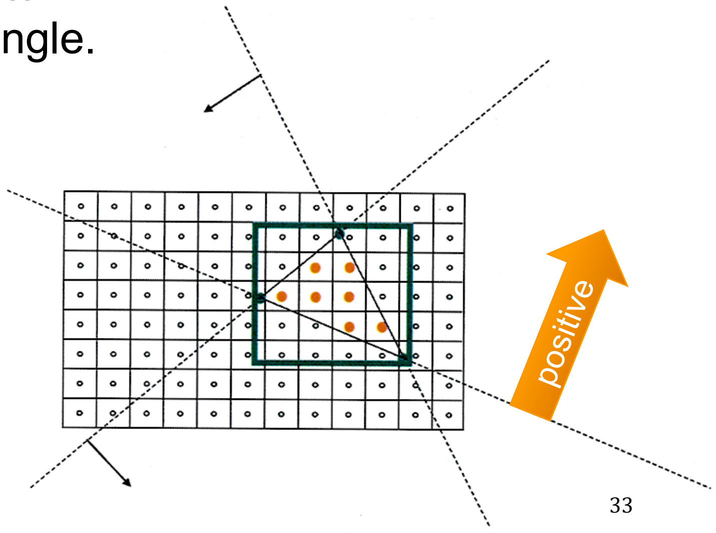
$$edge = ax_w + by_w + c$$

where the (a,b,c) are constants that depend on the geometry of the edge.



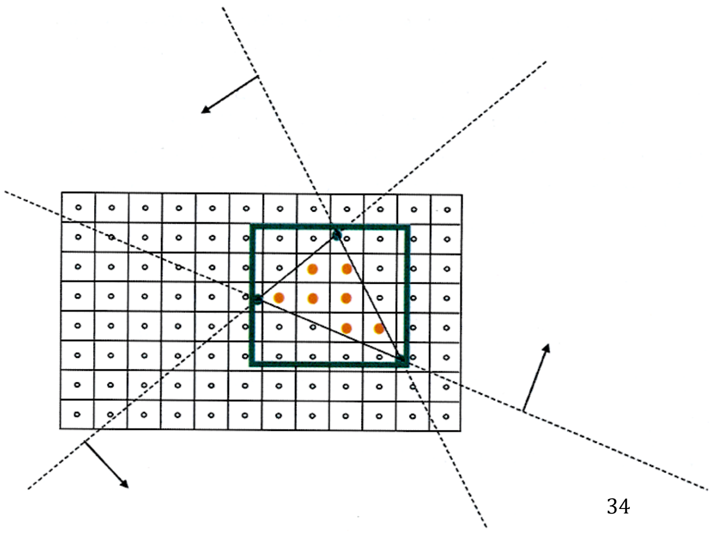
Math of rasterization

- A positive value of this function at a pixel with coordinates $[x_w, y_w]^t$ means that the pixel is inside the specified halfspace (on the left-hand side).
- If all three tests pass, then the pixel is inside the triangle.



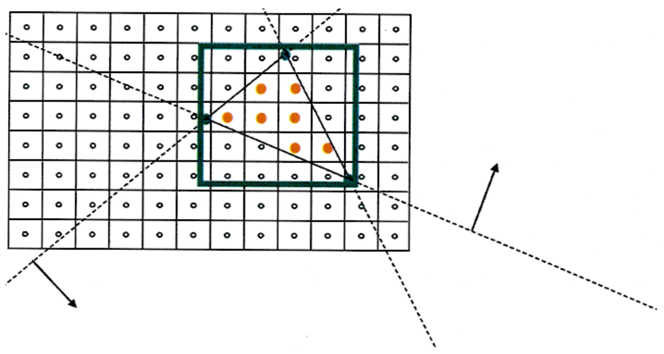
Speed up

- Only look at pixels in the <u>bounding box</u> of the triangle
- Test if a pixel block is entirely outside of triangle
- Use incremental calculations along a scanline.



Interpolation

- As input to rasterization, each vertex also has some auxiliary data associated with it.
  - This data includes a z_w value,
  - As well as other data that is related, but not identical to the varying variables.
- It is also the job of the rasterizer to linearly interpolate this data over the triangle.



Math of interpolation

- Each such value v to be linearly interpolated can be represented as an affine function over screen space with the form: $\text{1D}: \; v_x = t x_w + (1-t) x'_w$

$$v = a x_w + b y_w + c$$

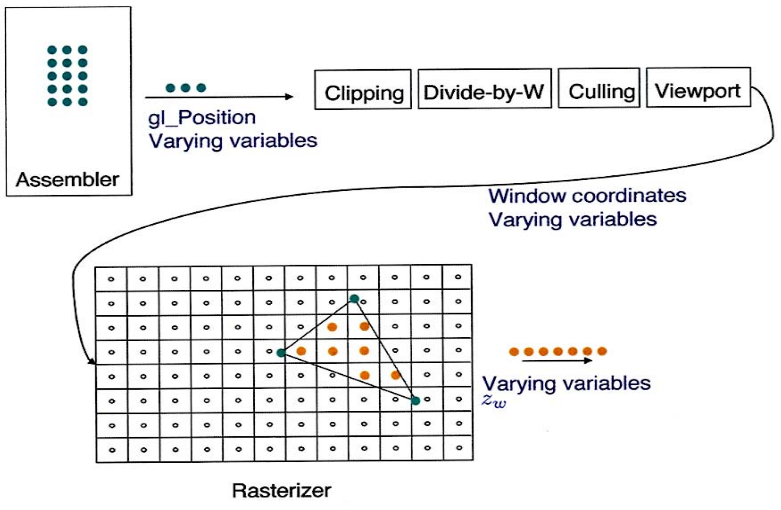
- An affine function can be easily evaluated at each pixel by the rasterizer.
- Indeed, this is no different from evaluating the edge test functions just described.

Boundaries

- For pixel on edge or vertex it should be rendered exactly once.
- Need special care in the implementation to avoid duty edge representation.

Next Lecture Reading

- Appendix B: Affine Functions
- Chapter 13: Varying Variables


Assembler
gl_Position
Varying variables
Clipping
Divide-by-W
Culling
Viewport
Window coordinates
Varying variables
Varying variables
z_w
Rasterizer