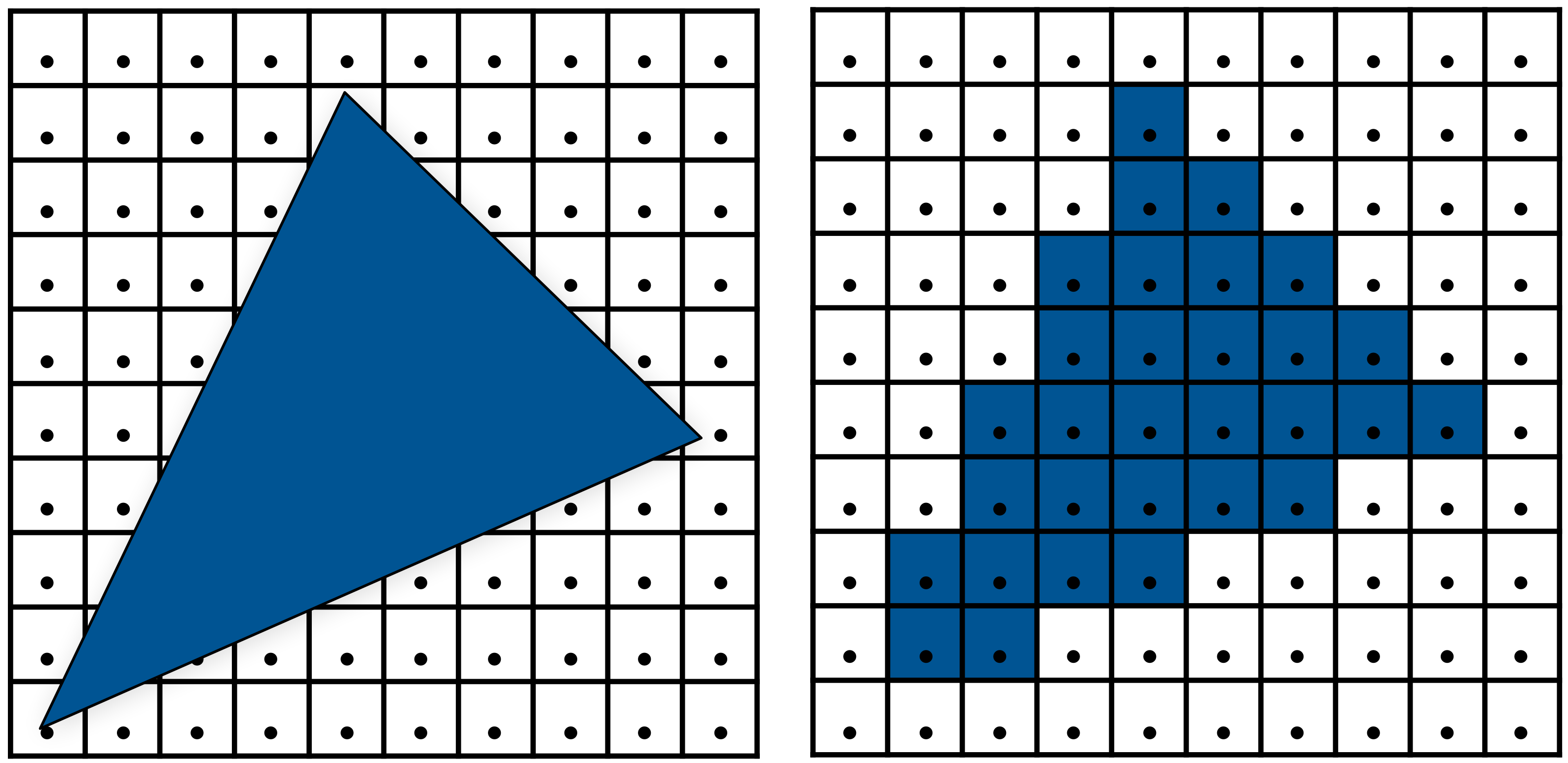
ECSE 446/546

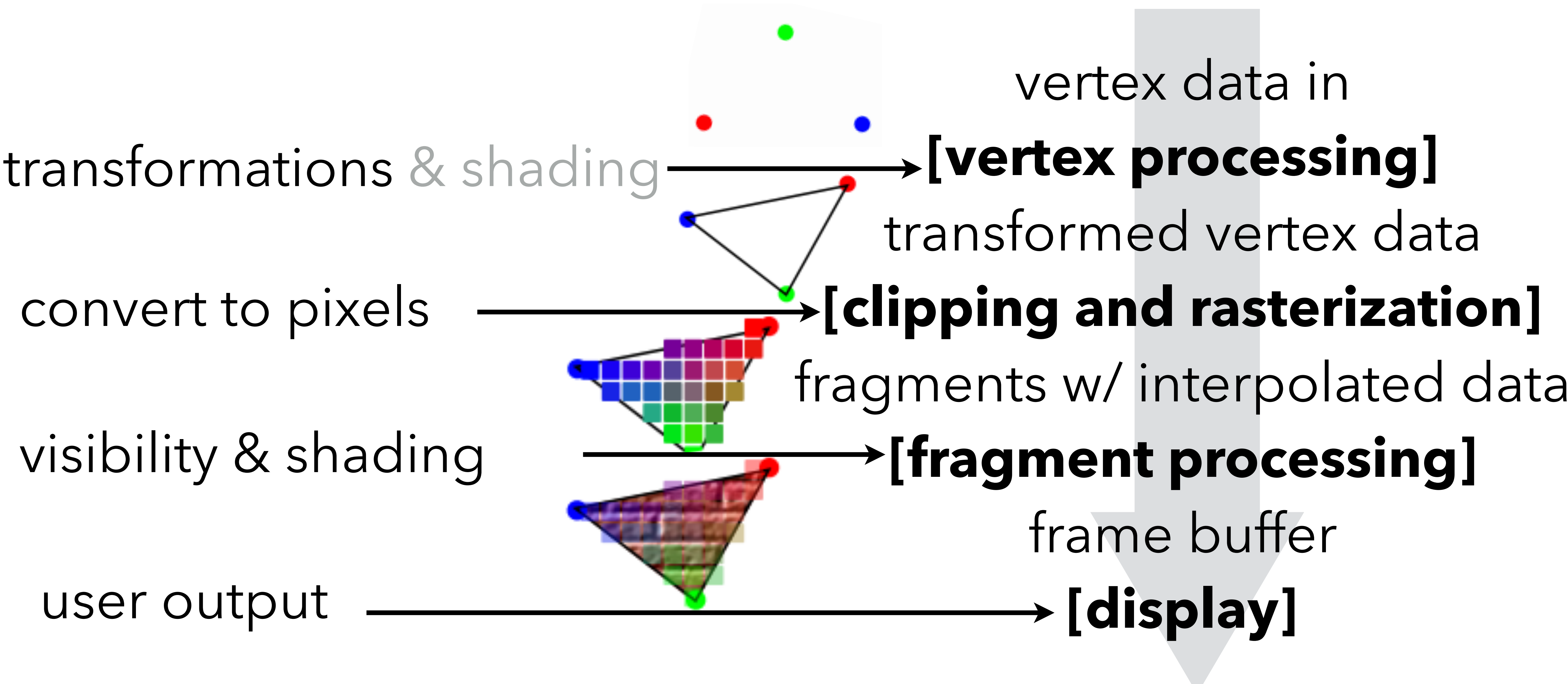
Image Synthesis



Systems 1 – Rasterization

Prof. Derek Nowrouzezahrai
derek@cim.mcgill.ca

Programmable Rasterization Pipeline™



After a slide by Steve Marschner

Programmable Rasterization Pipeline™

vertex data in

[vertex processing]

transformed vertex data

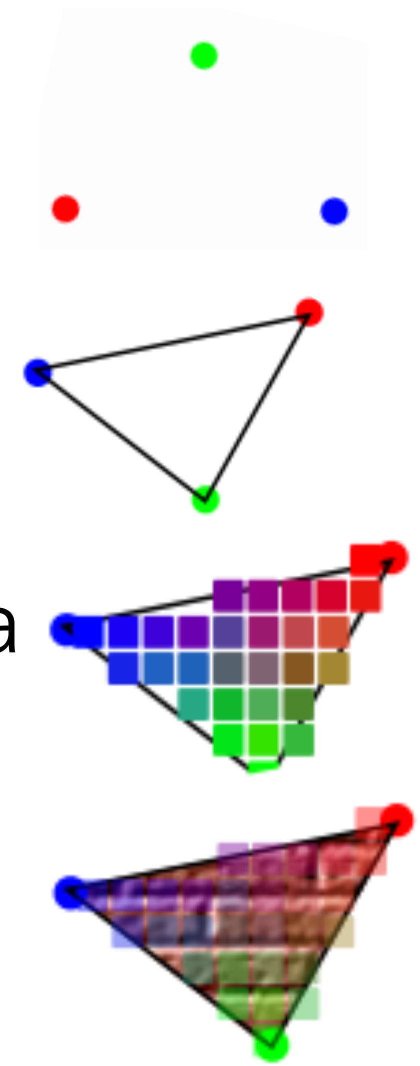
[clipping and rasterization]

fragments w/ interpolated data

[fragment processing]

frame buffer

[display]



```
for(each triangle)
    transform vertices into eye space
    project vertices to image space
    for(each pixel x,y)
        if(x,y in triangle)
```

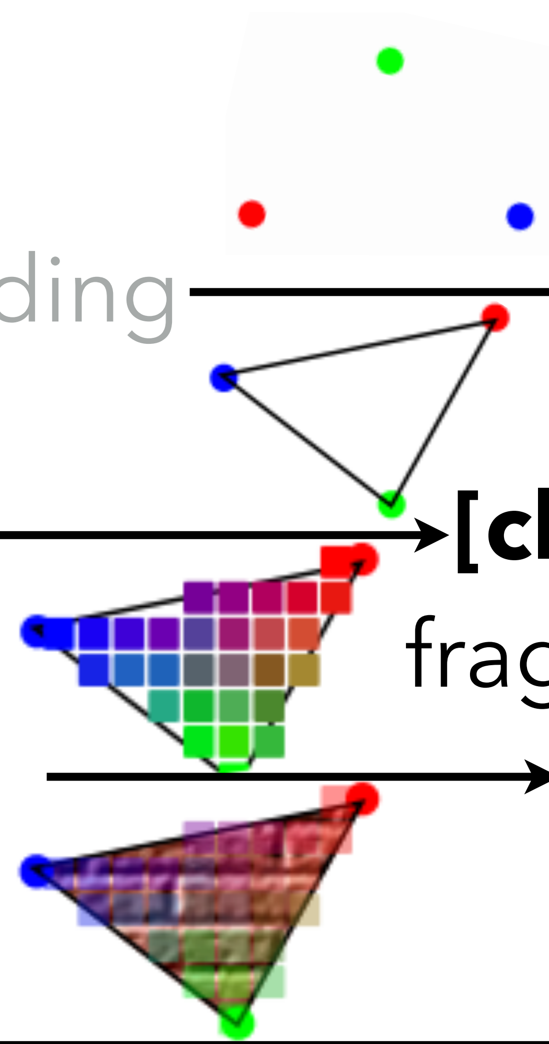
Programmable Rasterization Pipeline™

```
for(each triangle)
    transform vertices into eye space
    project vertices to image space
    for(each pixel x,y)
        if(x,y in triangle)
            compute z
            if(z < zbuffer[x,y])
                zbuffer[x,y] = z
                framebuffer[x,y] = shade()
```

Programmable Rasterization Pipeline™

```
for(each triangle)
    transform vertices into eye space
    project vertices to image space
    for(each pixel x,y)
        if(x,y in triangle)
            compute z
            if(z < zbuffer[x,y])
                zbuffer[x,y] = z
                framebuffer[x,y] = shade()
```

Programmable Rasterization Pipeline™



vertex data in

transformations & shading → **[vertex processing]**

transformed vertex data

convert to pixels → **[clipping and rasterization]**

fragments w/ interpolated data

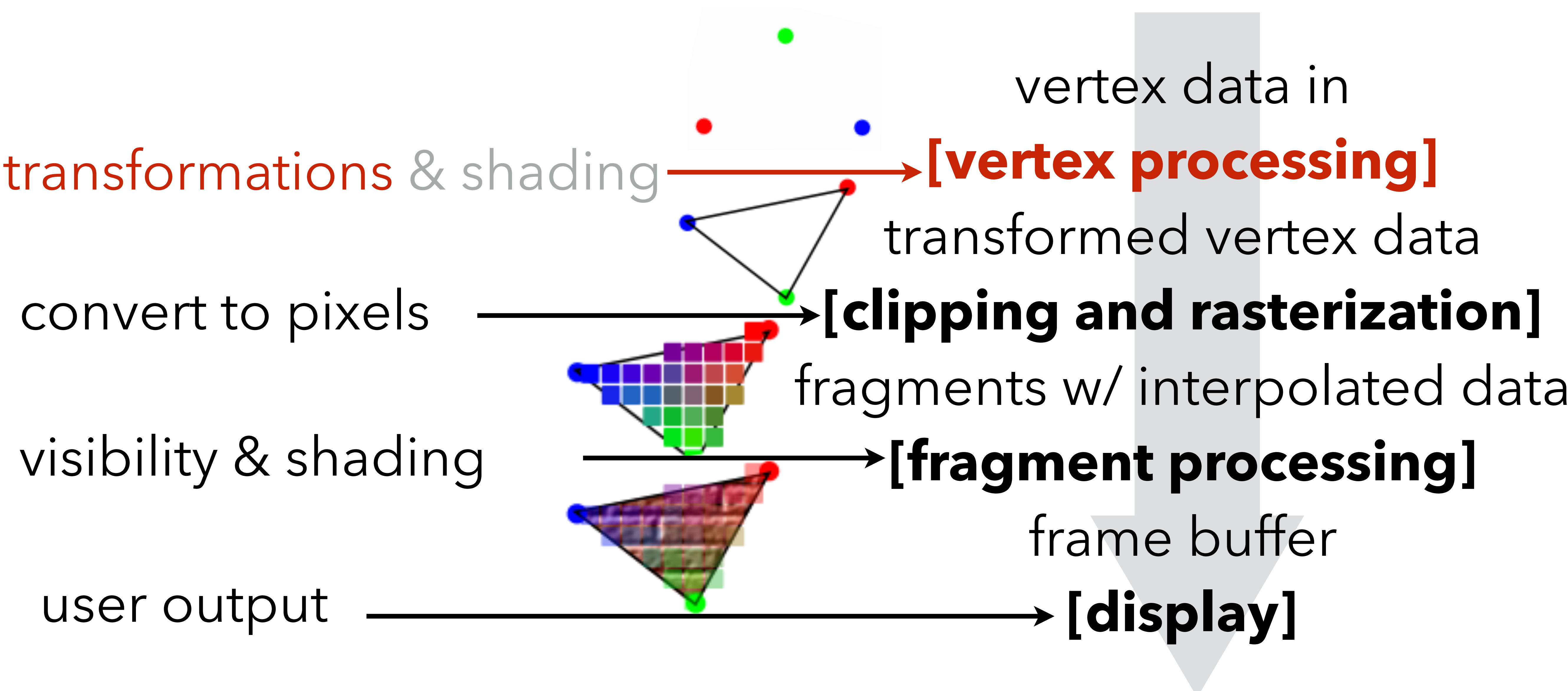
visibility & shading → **[fragment processing]**

frame buffer

user output → **[display]**

After a slide by Steve Marschner

Programmable Rasterization Pipeline™



After a slide by Steve Marschner

Programmable Rasterization Pipeline™

```
for(each triangle)
    transform vertices into eye space
    project vertices to image space
    for(each pixel x,y)
        if(x,y in triangle)
            compute z
            if(z < zbuffer[x,y])
                zbuffer[x,y] = z
                framebuffer[x,y] = shade()
```

Programmable Rasterization Pipeline™

```
for(each triangle)
    transform vertices into eye space
    project vertices to image space
    for(each pixel x,y)
        if(x,y in triangle)
            compute z
            if(z < zbuffer[x,y])
                zbuffer[x,y] = z
                framebuffer[x,y] = shade()
```

Model/View/Projection (MVP) Transformations

Programmable Rasterization Pipeline™

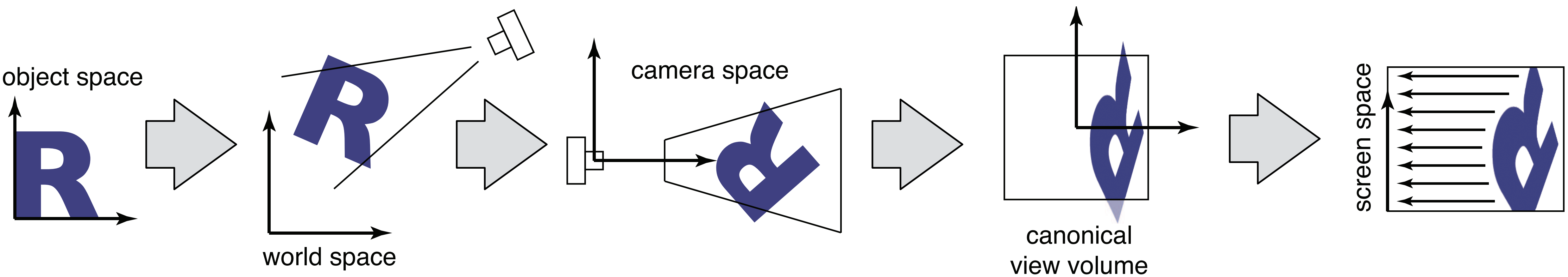
```
for(each triangle)
    transform vertices into eye space
    project vertices to image space
    for(each pixel x,y)
        if(x,y in triangle)
            compute z
            if(z < zbuffer[x,y])
                zbuffer[x,y] = z
                framebuffer[x,y] = shade()
```

Rasterization

Goal: convert from object point to image plane

- start with a 3D object point
- apply a sequence of transforms
  - transforms can be specified by a matrix
- determine the 2D image plane point it projects to

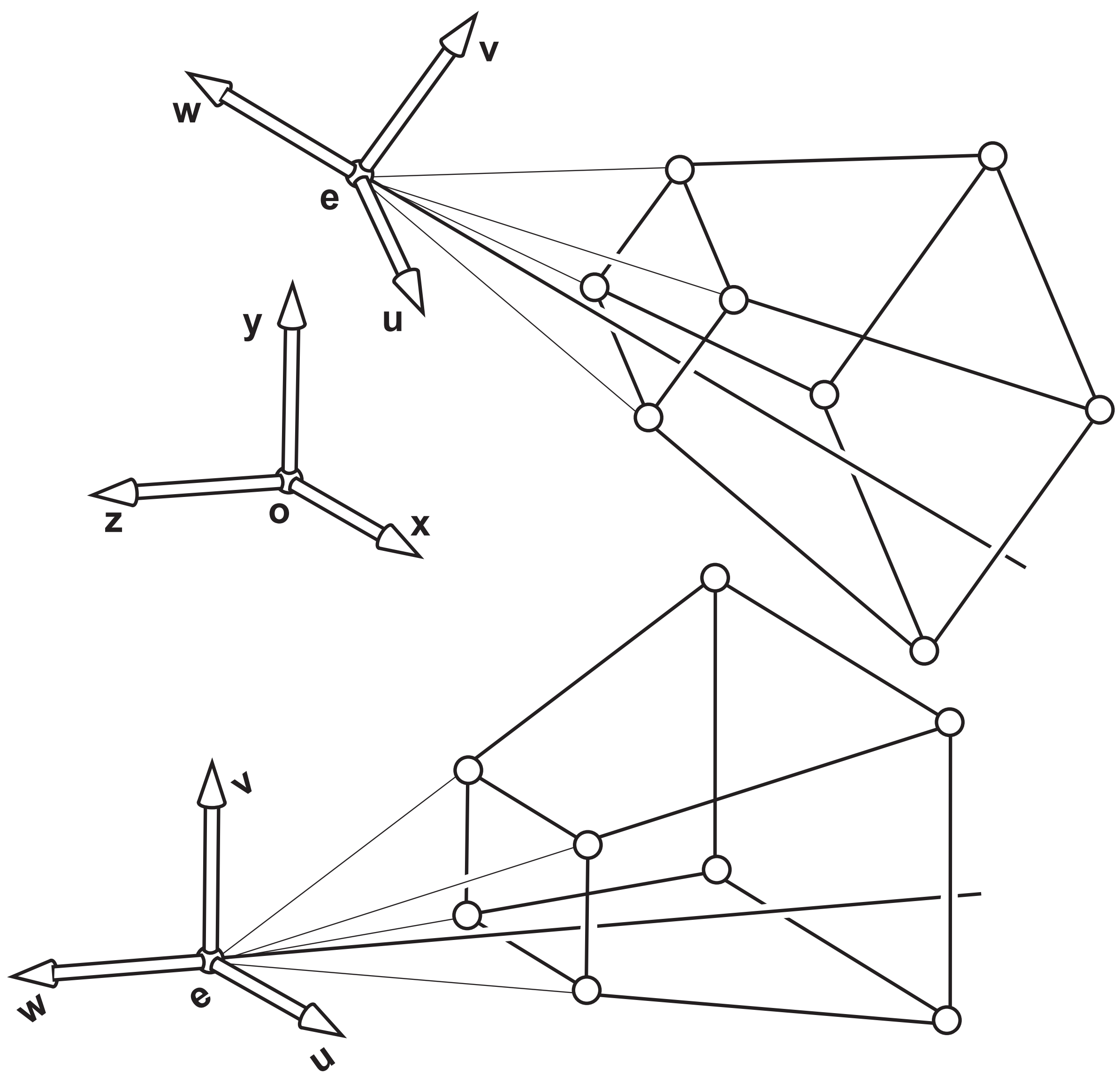
Pipeline of transformations

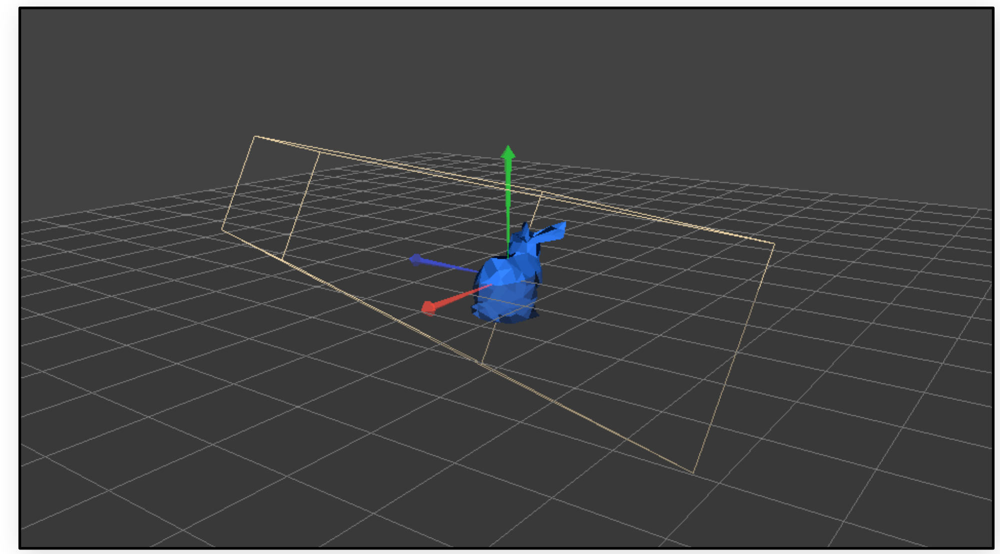


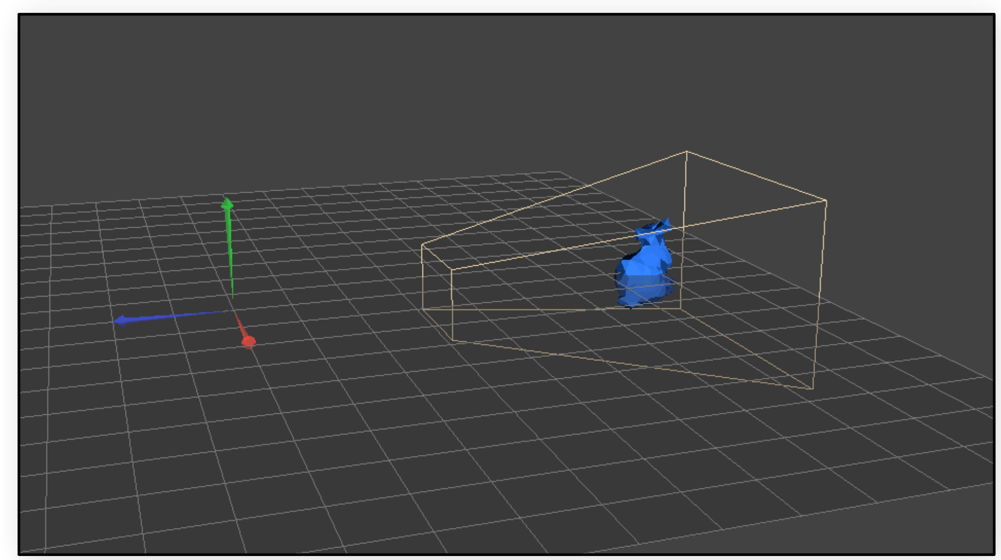
1. Model	2. Viewing	3. Projection	4. Viewport
map local object coords to world coords	map world coords to camera coords	map camera coords to canonical view volume	map canonical view volume to screen space

These two stages perform the actual 3D-to-2D projection

2. Viewing transform







The <u>view matrix</u> transforms all coordinates into “eye space”

2. Viewing transform

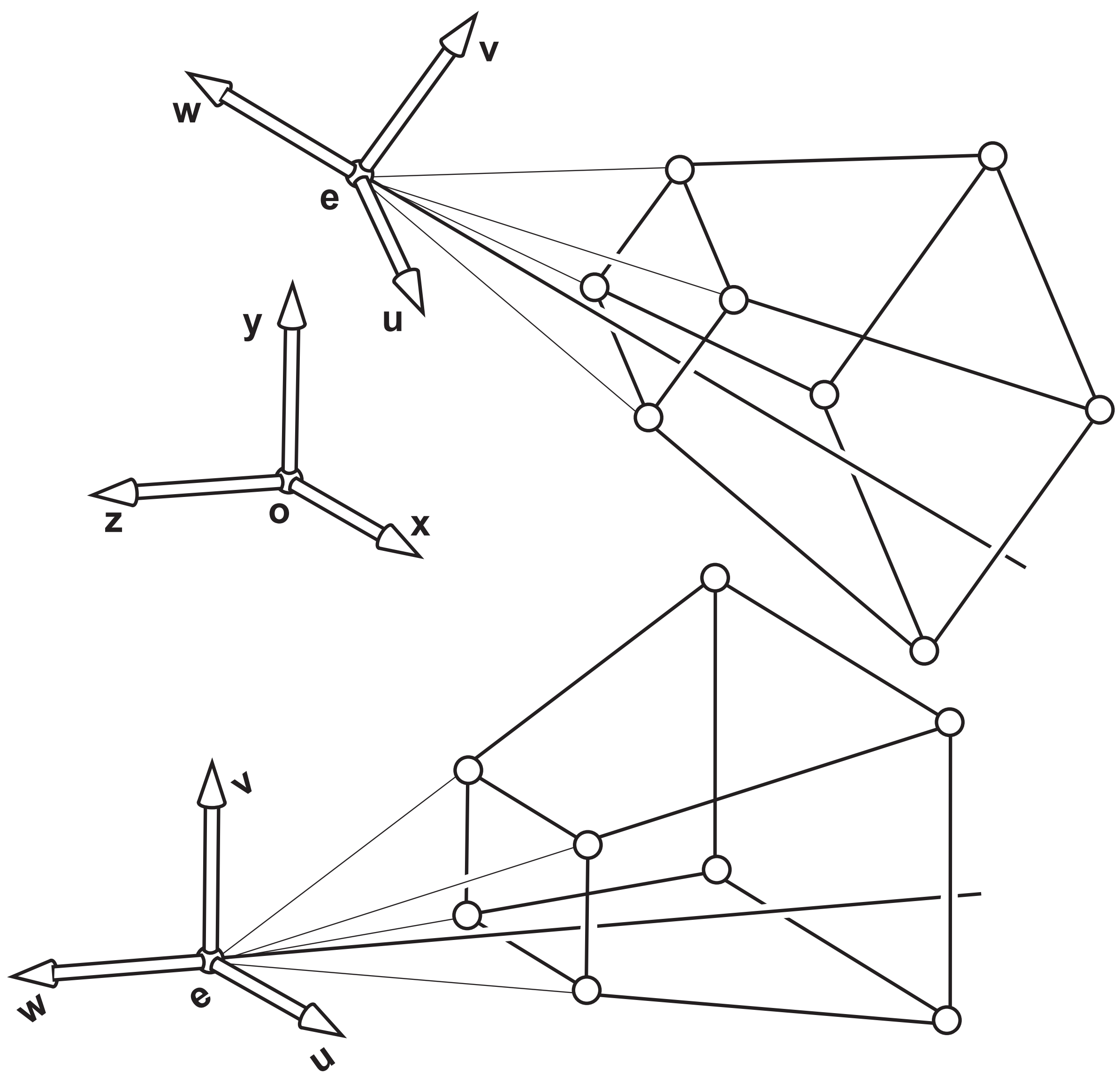
Many ways to construct a view transform (i.e., matrix)

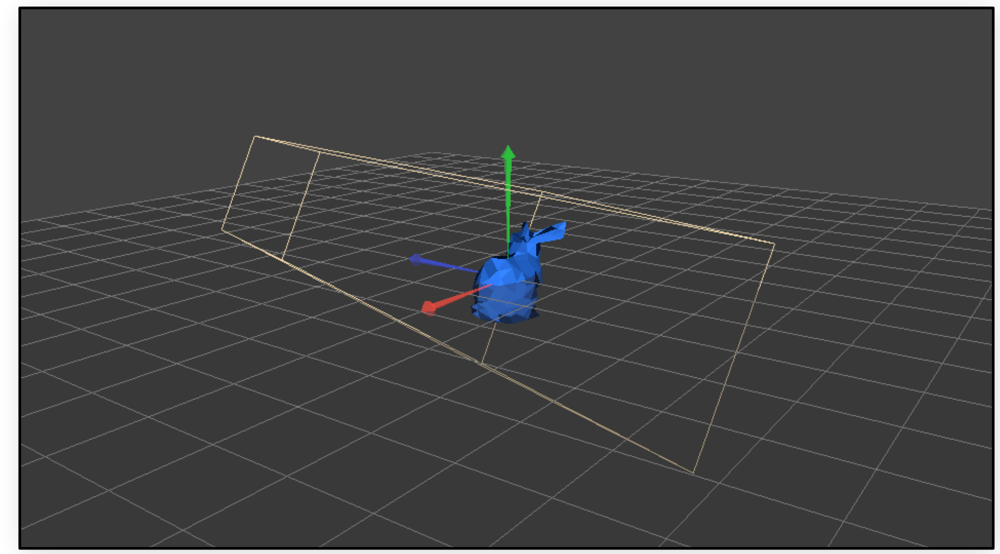
For example, can specify an orthonormalized frame with the:

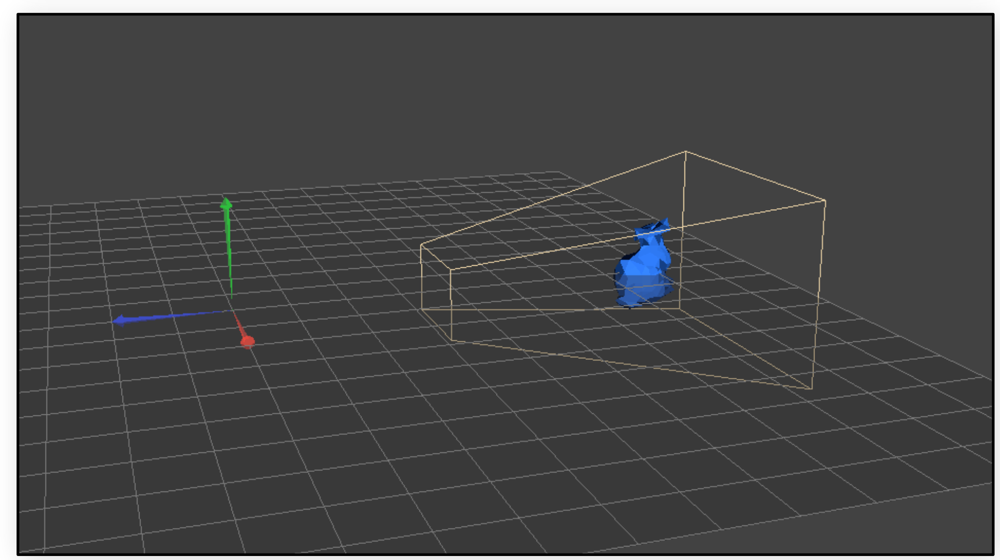
- **eye** point
- **up**-vector, and
- **look-at** point

After a slide by Fabio Pellacini

2. Viewing transform







Programmable Rasterization Pipeline™

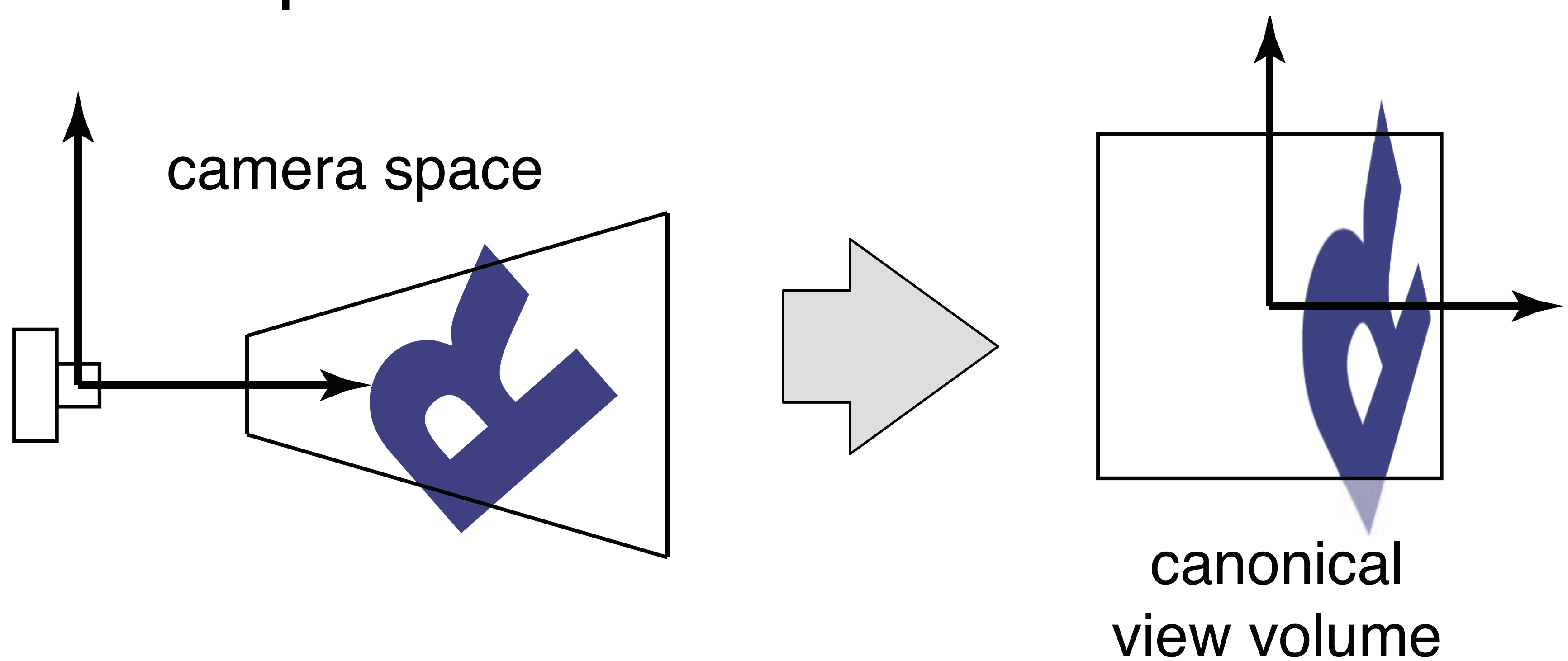
```
for(each triangle)
    transform vertices into eye space
    project vertices to image space
    for(each pixel x,y)
        if(x,y in triangle)
            compute z
            if(z < zbuffer[x,y])
                zbuffer[x,y] = z
                framebuffer[x,y] = shade()
```

3. Projection

Generally, a function that transforms points from m- to n-space where $m > n$

In graphics, map 3D points to 2D image coordinates

- except we will keep around the third coordinate



After a slide by Fabio Pellacini

Mathematics of Projection

Always work in eye coords

- assume eye point at **0** and plane perpendicular to z

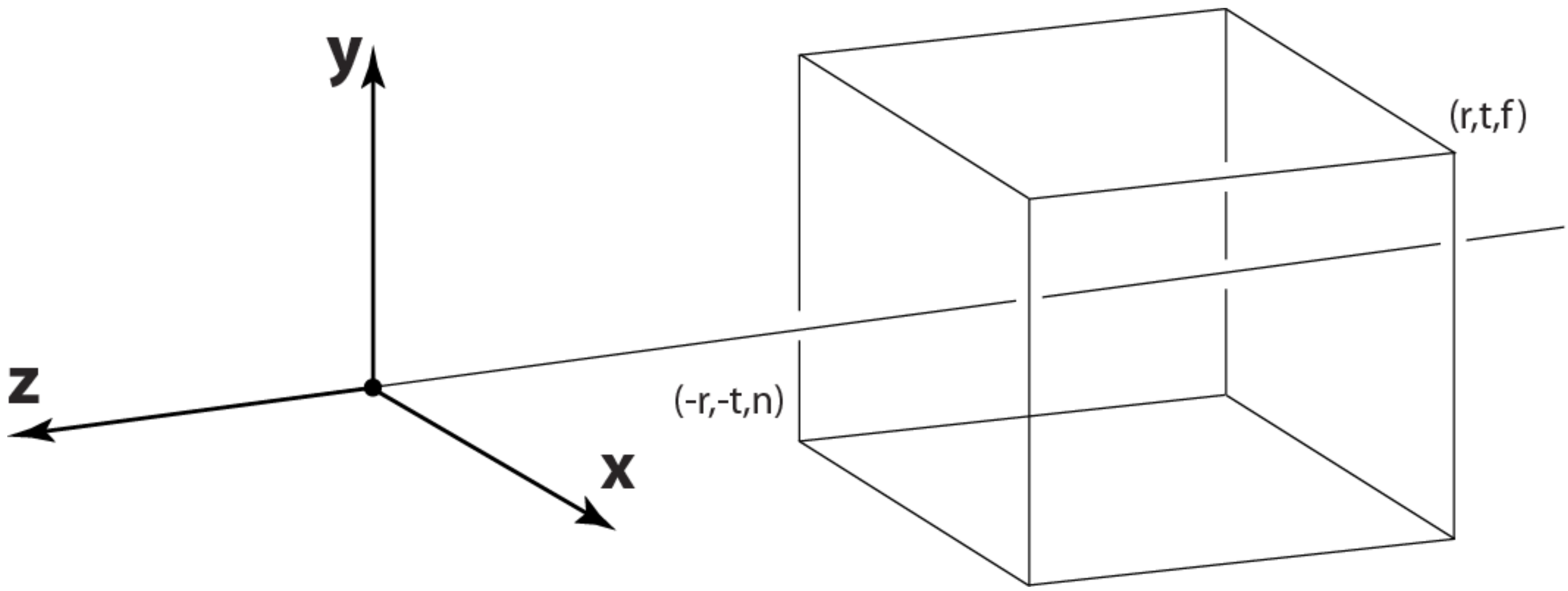
Orthographic case: just toss out the z-coord

Perspective case: scale diminishes with z

After a slide by Steve Marschner

Orthographic projection

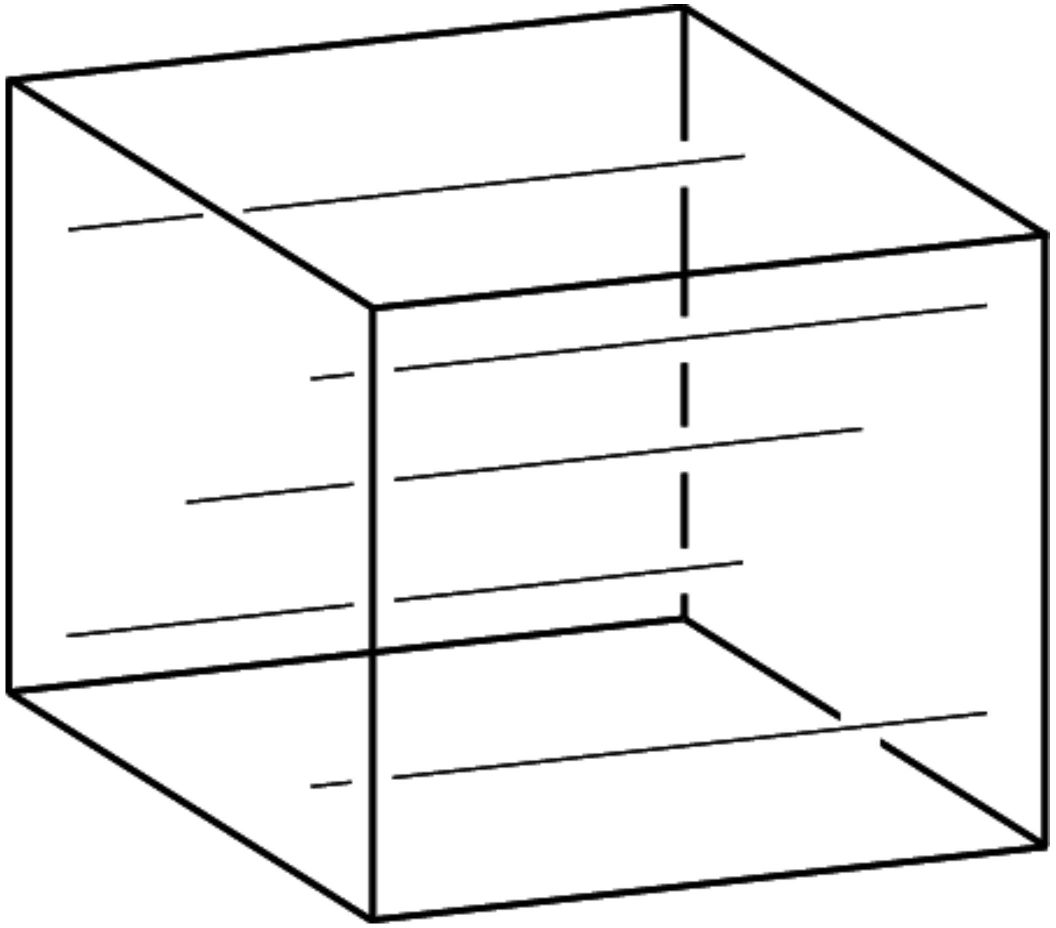
Box view volume



After a slide by Fabio Pellacini

Orthographic projection

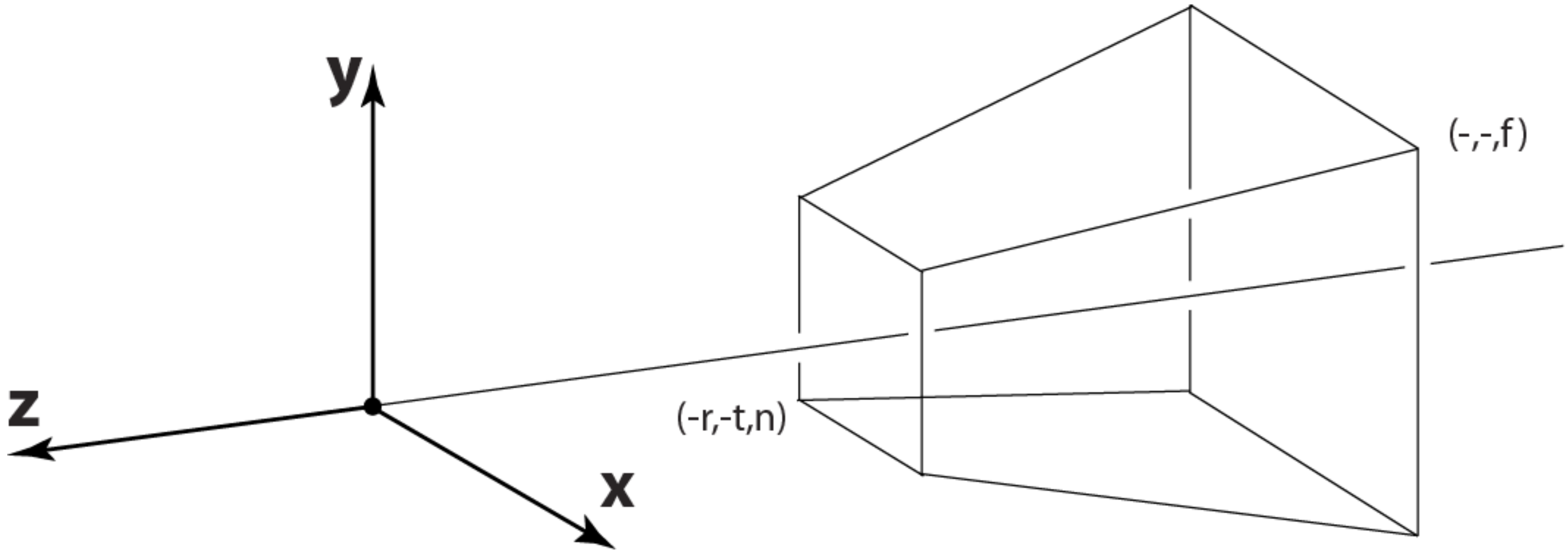
Viewing rays are parallel



After a slide by Fabio Pellacini

Perspective projection

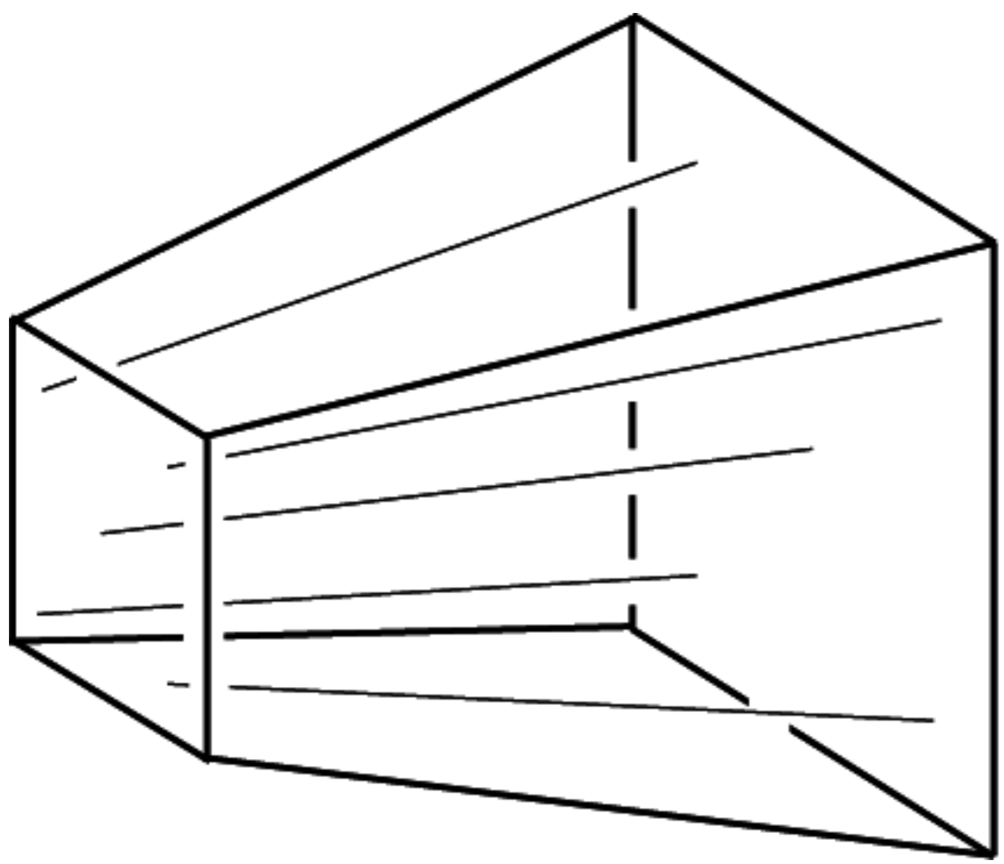
Truncated pyramid view volume



After a slide by Fabio Pellacini

Perspective projection

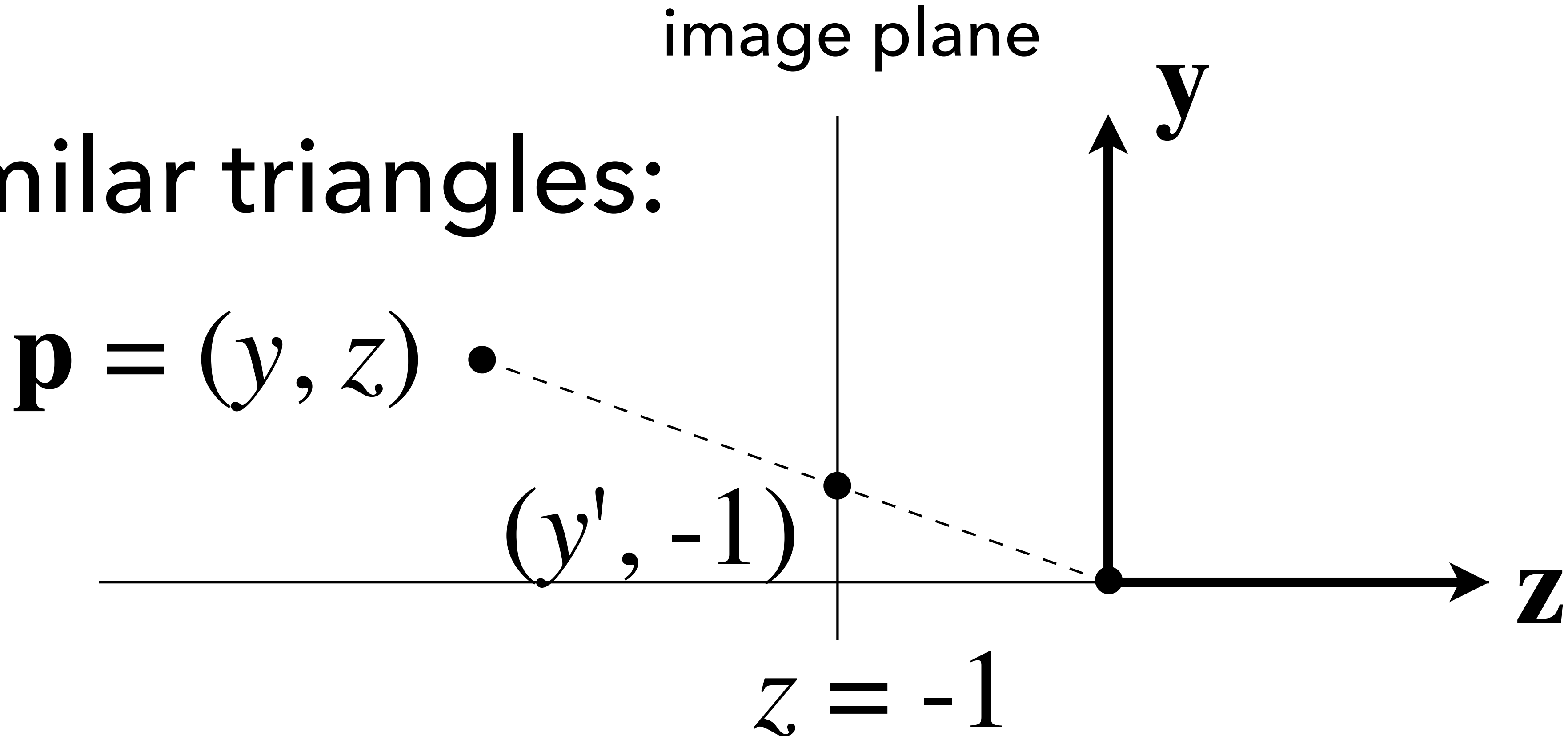
Viewing rays converge to a point



After a slide by Fabio Pellacini

Perspective projection

Use similar triangles:



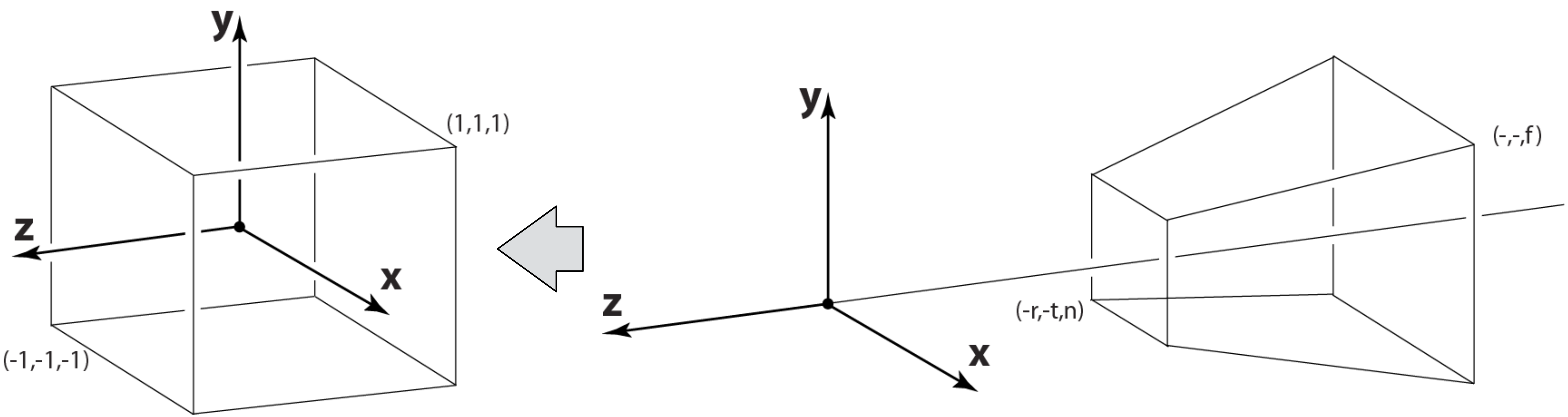
$$\frac{y'}{-1} = \frac{y}{z}$$

$$y' = -\frac{y}{z}$$

After a slide by Steve Marschner

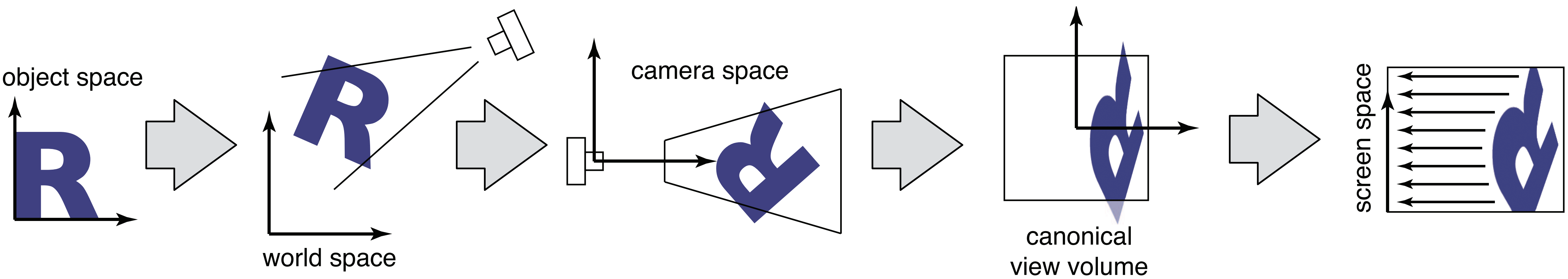
Remapping the view frustum

We also want the projection matrix to remap the space between **near** and **far** planes to a canonical view volume



After a slide by Fabio Pellacini

Pipeline of transformations



1. **Model**

map local object coords to world coords

2. **Viewing**

map world coords to camera coords

3. **Projection**

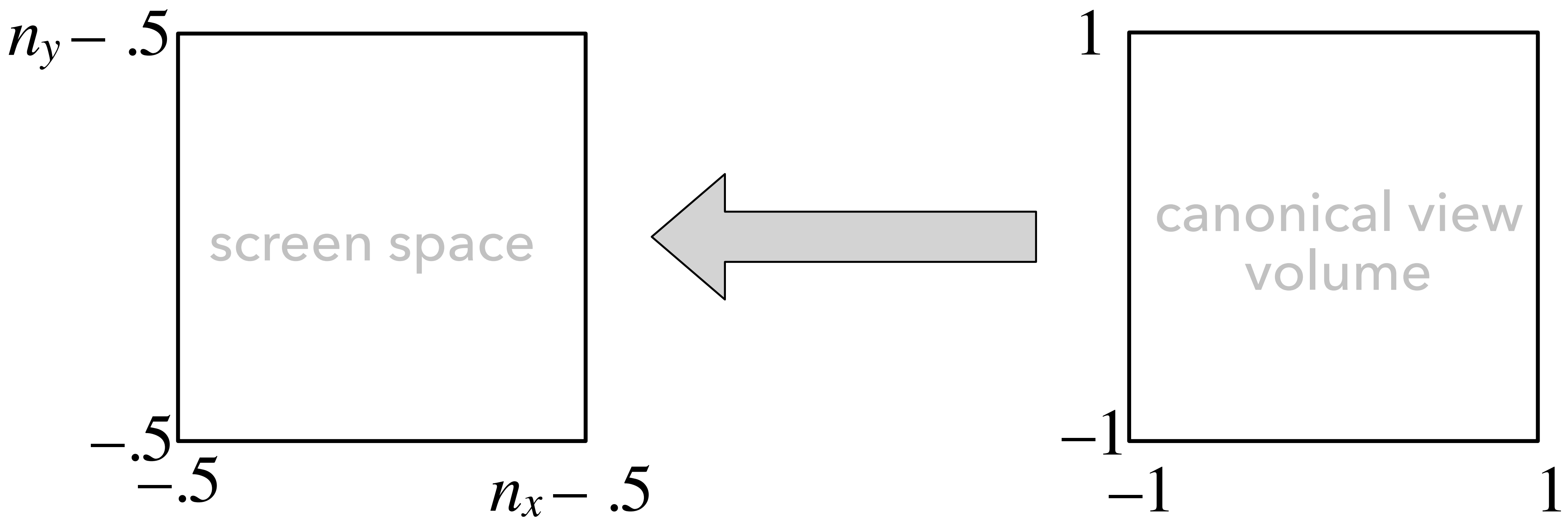
map camera coords to canonical view volume

4. **Viewport**

map canonical view volume to screen space

These two stages perform the actual 3D-to-2D projection

4. Viewport transformation



$$\begin{bmatrix} x_{\text{screen}} \\ y_{\text{screen}} \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{n_x}{2} & 0 & \frac{n_x - 1}{2} \\ 0 & \frac{n_y}{2} & \frac{n_y - 1}{2} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{\text{canonical}} \\ y_{\text{canonical}} \\ 1 \end{bmatrix}$$

After a slide by Steve Marschner

Orthographic Transformation Chain

Start with coordinates in object's local coordinates

Transform into world coords (modeling transform, $\mathbf{M}_\mathrm{m}$)

Transform into eye coords (camera xf., $\mathbf{M}_\mathrm{cam} = \mathrm{F_c}^{-1}$)

Orthographic projection, $\mathbf{M}_\mathrm{orth}$

Viewport transform, $\mathbf{M}_\mathrm{vp}$

$$\mathbf{p}_s = \mathbf{M}_\mathrm{vp}\mathbf{M}_\mathrm{orth}\mathbf{M}_\mathrm{cam}\mathbf{M}_\mathrm{m}\mathbf{p}_o$$

$$\begin{bmatrix} x_s \\ y_s \\ z_c \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x-1}{2} \\ 0 & \frac{n_y}{2} & 0 & \frac{n_y-1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{n-f} & -\frac{n+f}{n-f} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{e} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \mathbf{M}_\mathrm{m} \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix}$$

After a slide by Steve Marschner

Perspective Transformation Chain

Start with coordinates in object's local coordinates

Transform into world coords (modelling transform, $\mathbf{M}_{\mathrm{m}}$)

Transform into eye coords (camera xf., $\mathbf{M}_{\mathrm{cam}} = \mathrm{F}_{\mathrm{c}}^{-1}$)

Perspective projection, $\mathbf{M}_{\mathrm{persp}}$

Viewport transform, $\mathbf{M}_{\mathrm{vp}}$

$$\mathbf{p}_s = \mathbf{M}_{\mathrm{vp}}\mathbf{M}_{\mathrm{persp}}\mathbf{M}_{\mathrm{cam}}\mathbf{M}_{\mathrm{m}}\mathbf{p}_o$$

$$\begin{bmatrix} x_s \\ y_s \\ z_c \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{n_x}{2} & 0 & 0 & \frac{n_x-1}{2} \\ 0 & \frac{n_y}{2} & 0 & \frac{n_y-1}{2} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{l+r}{l-r} & 0 \\ 0 & \frac{2n}{t-b} & \frac{b+t}{b-t} & 0 \\ 0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{u} & \mathbf{v} & \mathbf{w} & \mathbf{e} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \mathbf{M}_{\mathrm{m}} \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix}$$

After a slide by Steve Marschner

Recap: Transformation Pipeline

Perform rotation/translation/other transforms to put viewpoint at origin and view direction along z axis

- combination of model and view matrix called "modelview" matrix (e.g., in OpenGL)

Combine with projection matrix (perspective or orthographic)

- this is the OpenGL "projection" matrix

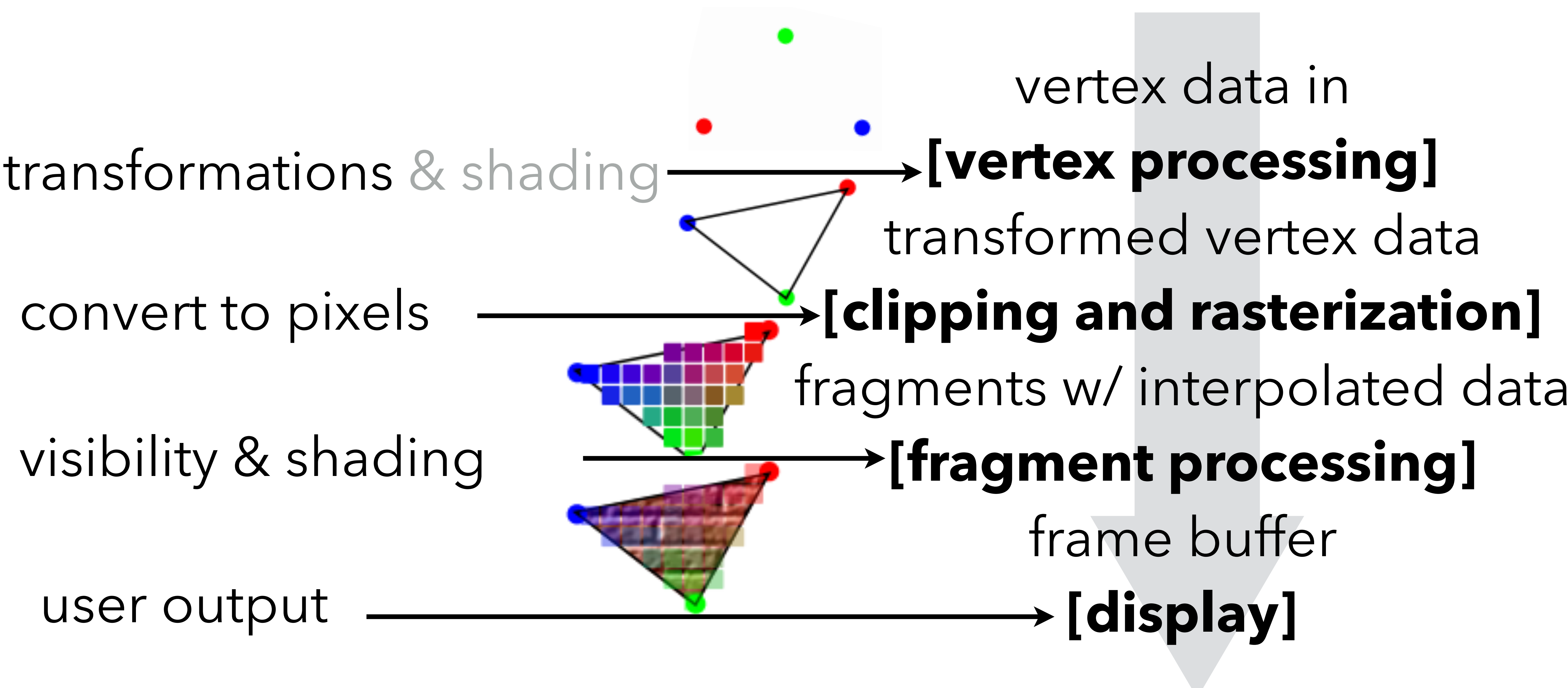
Convert canonical view volume $[-1,1]^3$ to screen space

Corollary: The entire transformation from local object space to screen space is a single 4x4 matrix!

After a slide by Frédo Durand

Questions?

Programmable Rasterization Pipeline™



After a slide by Steve Marschner

Programmable Rasterization Pipeline™

vertex data in

transformations & shading → **[vertex processing]**

transformed vertex data

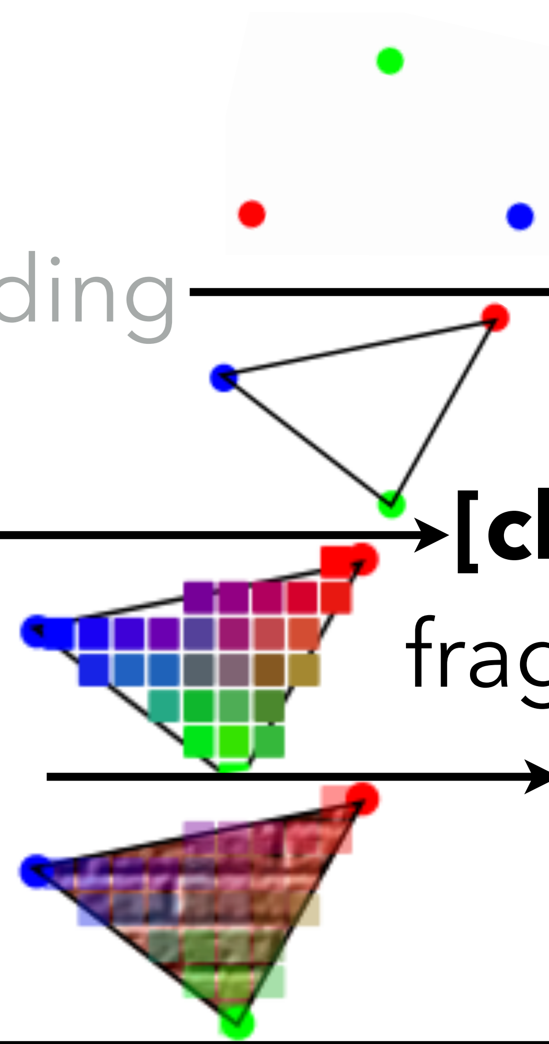
convert to pixels → **[clipping and rasterization]**

fragments w/ interpolated data

visibility & shading → **[fragment processing]**

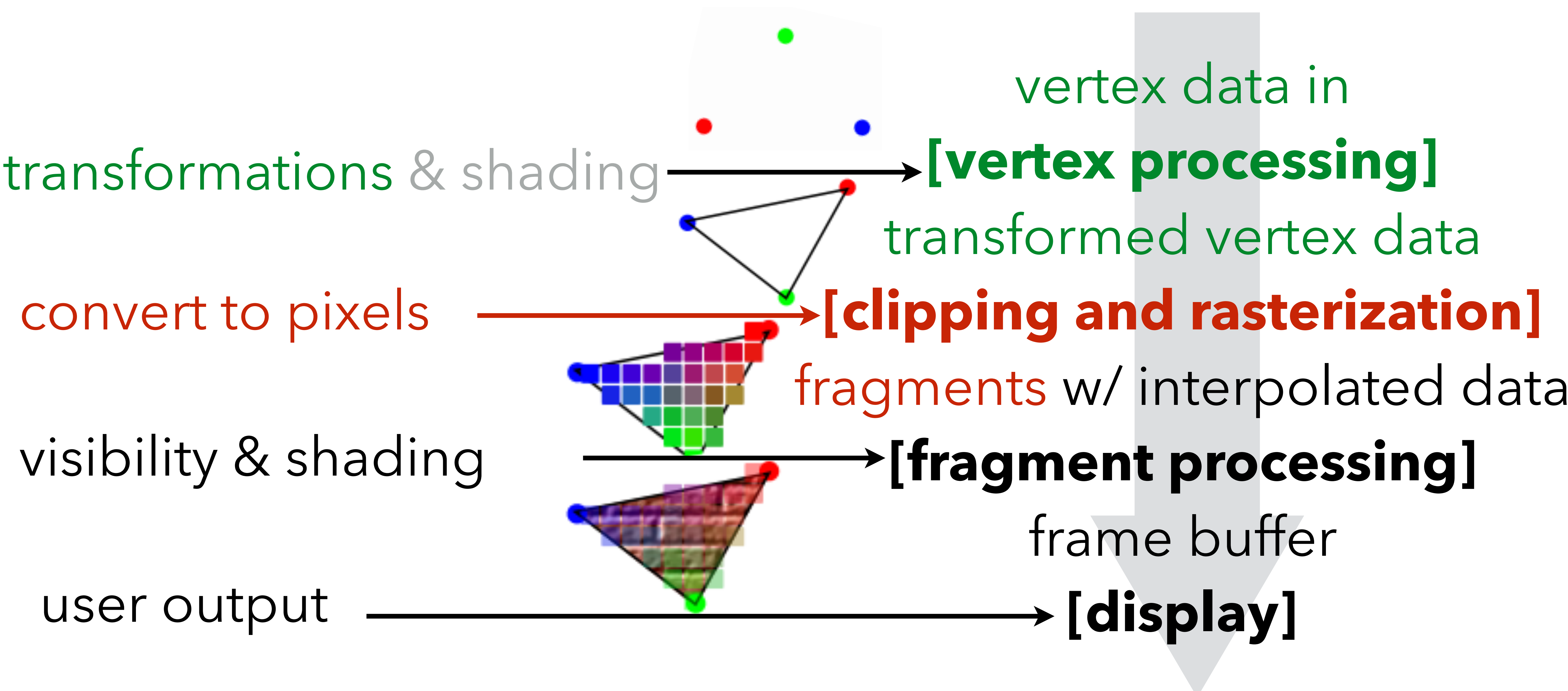
frame buffer

user output → **[display]**



After a slide by Steve Marschner

Programmable Rasterization Pipeline™



After a slide by Steve Marschner

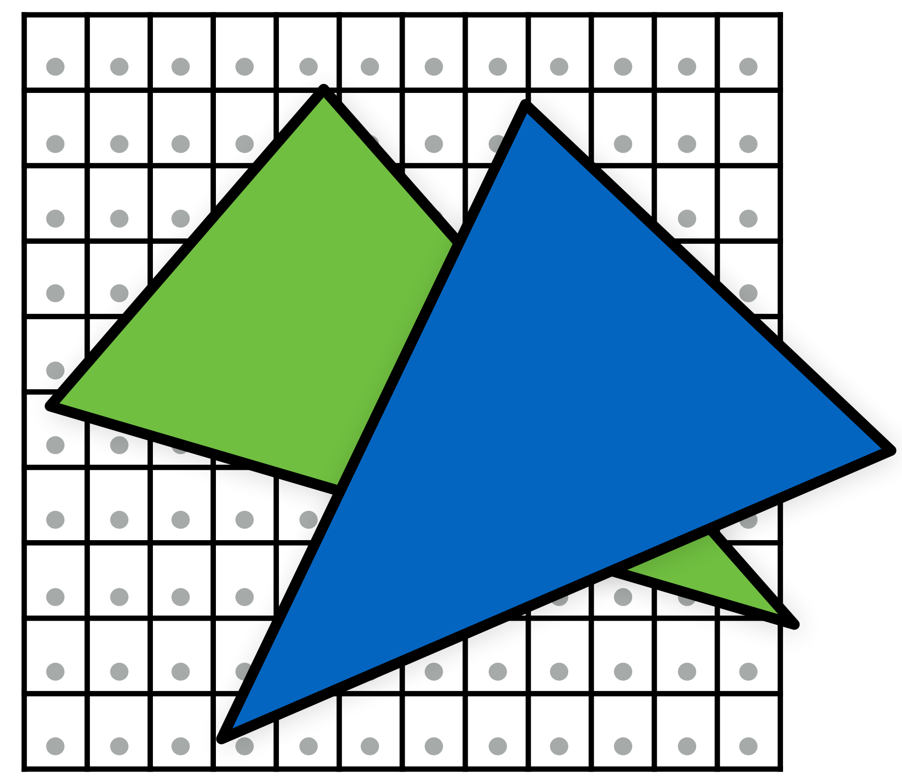
Programmable Rasterization Pipeline™

```
for(each triangle)
    transform vertices into eye space
    project vertices to image space
    for(each pixel x,y)
        if(x,y in triangle)
            compute z
            if(z < zbuffer[x,y])
                zbuffer[x,y] = z
                framebuffer[x,y] = shade()
```

Triangle Rasterization

Triangle Rasterization

Primitives are "continuous" geometric objects; screen is discrete (pixels)

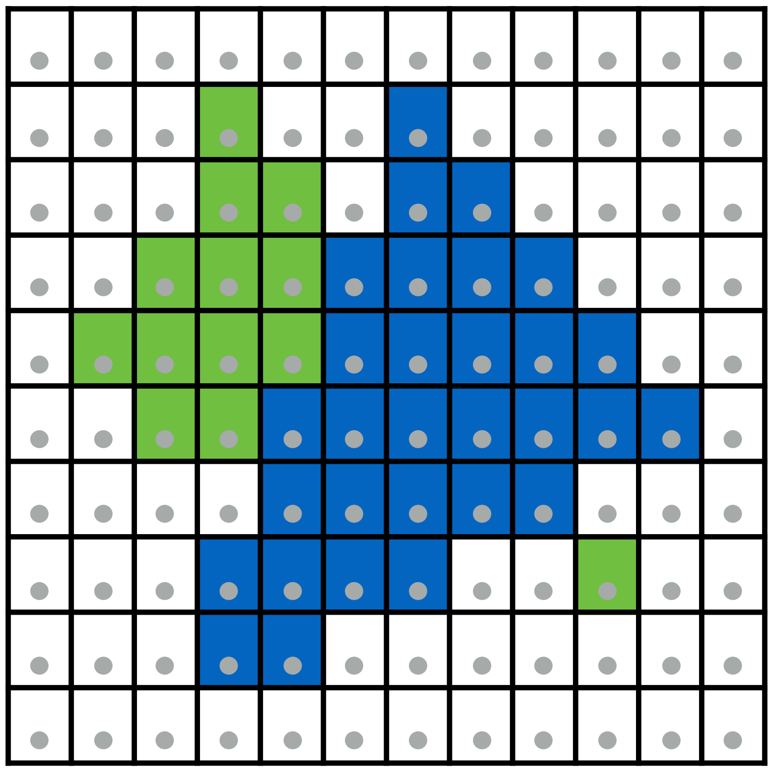


After a slide by Frédo Durand

Rasterization

Primitives are "continuous" geometric objects; screen is discrete (pixels)

Rasterization computes a discrete approximation in terms of pixels

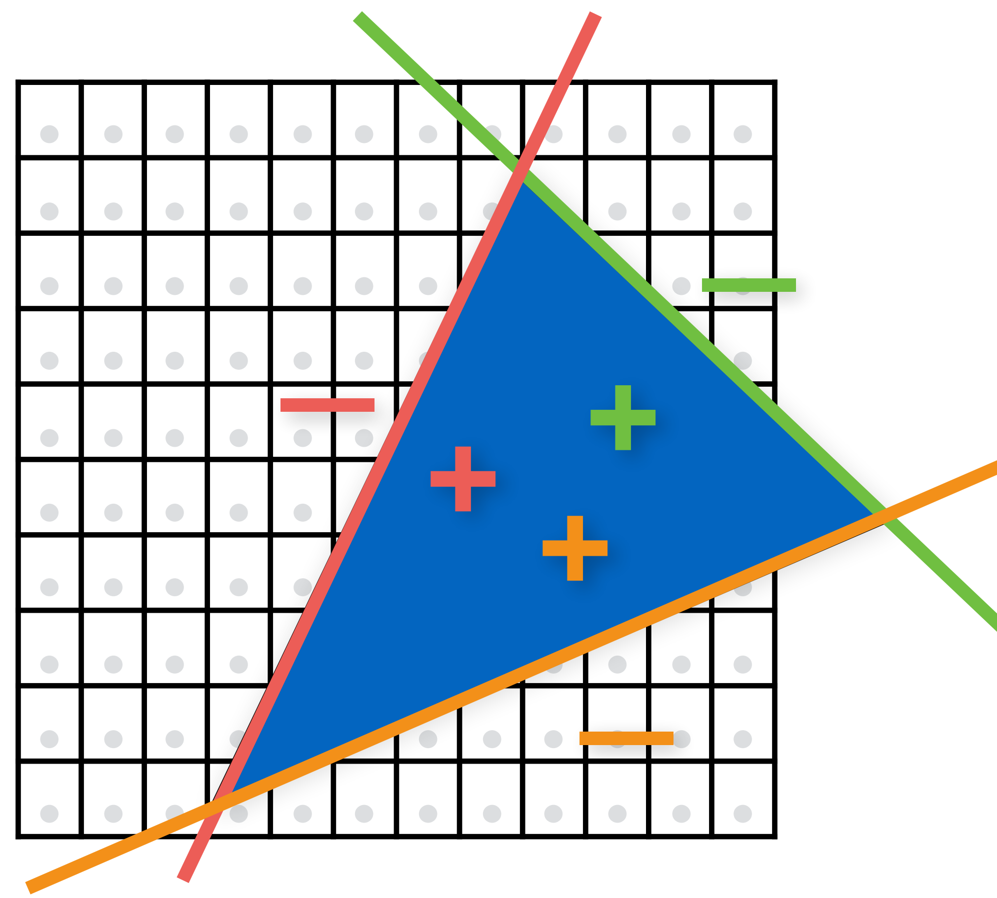


After a slide by Frédo Durand

Edge Functions

A triangle's 3D edges project to line segments in the image (thanks to planar perspective)

- lines map to lines, not curves



After a slide by Frédo Durand

Edge Functions

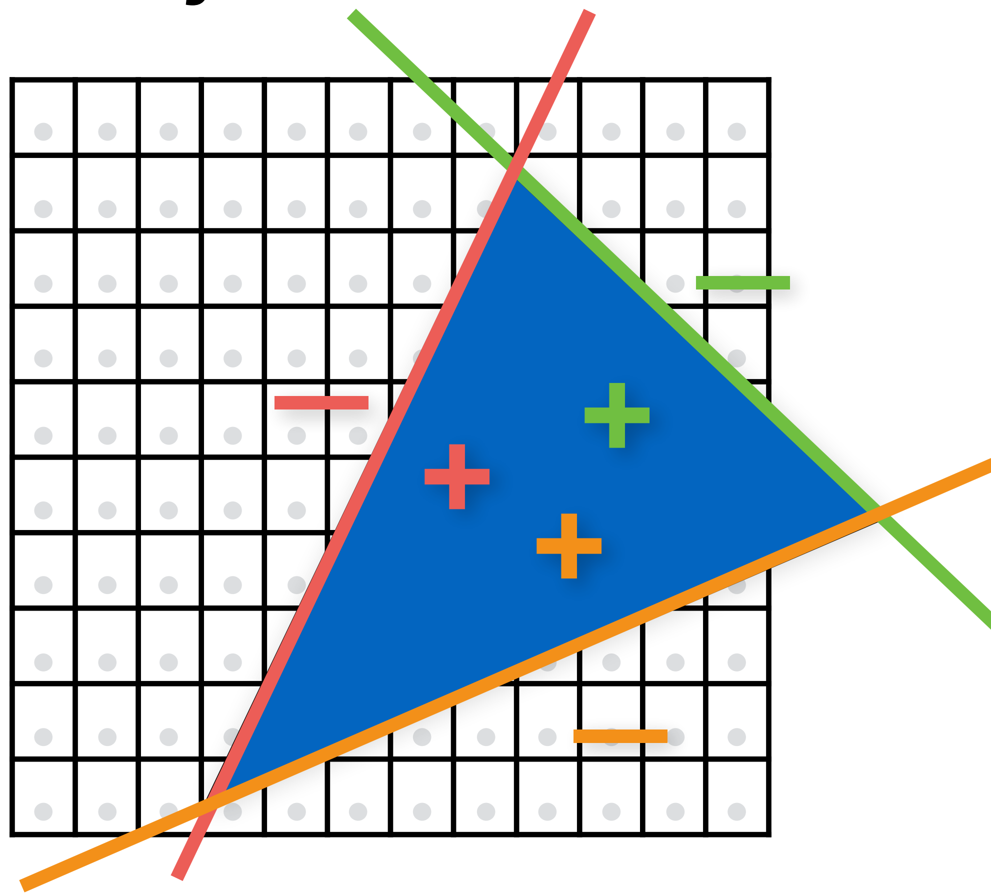
A triangle's 3D edges project to line segments in the image (thanks to planar perspective)

- interior of the triangle is the set of points lie inside **all** 3 half-spaces defined by these lines

After a slide by Frédo Durand

$$E_i(x, y) = a_i x + b_i y + c_i$$

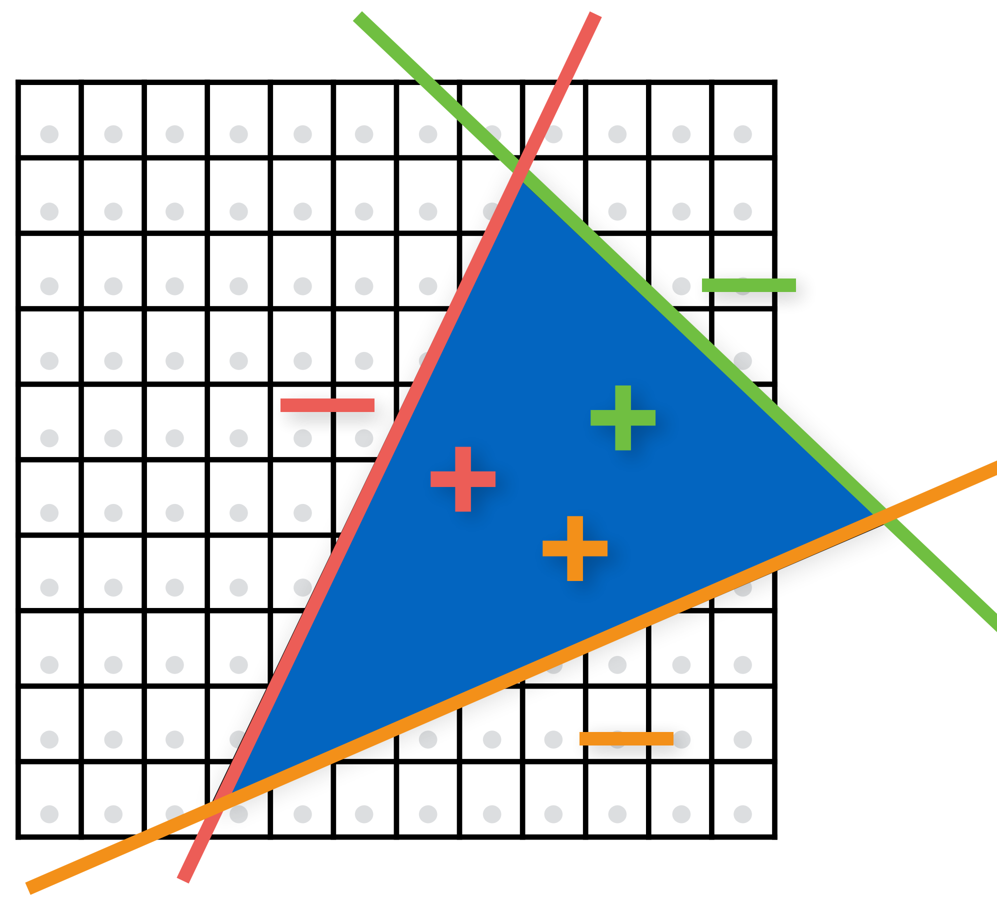
$$(x, y) \text{ within triangle iff}$$
$$E_i(x, y) \geq 0, \forall i = 1, 2, 3$$



Brute Force Triangle Rasterizer

Compute E_1, E_2, E_3 coefficients from projected vertices

- called "triangle setup": yields a_i, b_i, c_i for $i = 1,2,3$



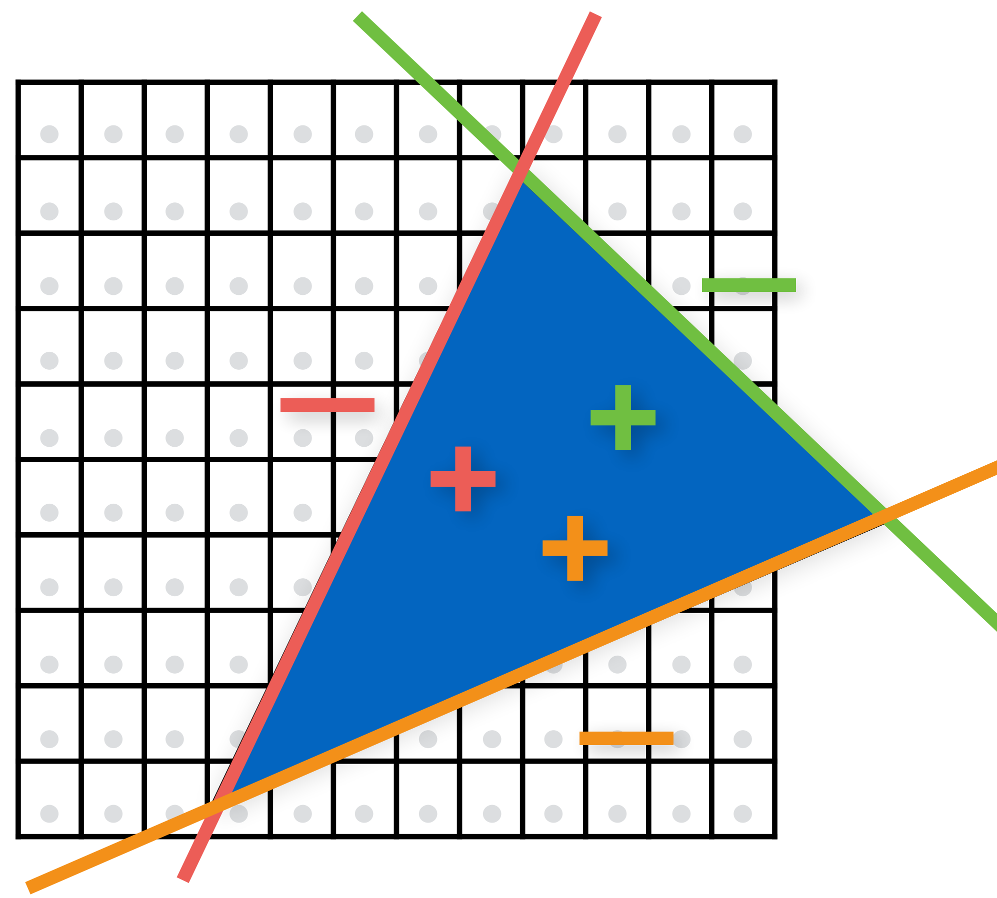
After a slide by Frédo Durand

Brute Force Triangle Rasterizer

```
for each pixel (x,y)
    evaluate edge functions at pixel center
    if all non-negative, pixel is in
```

After a slide by Frédo Durand

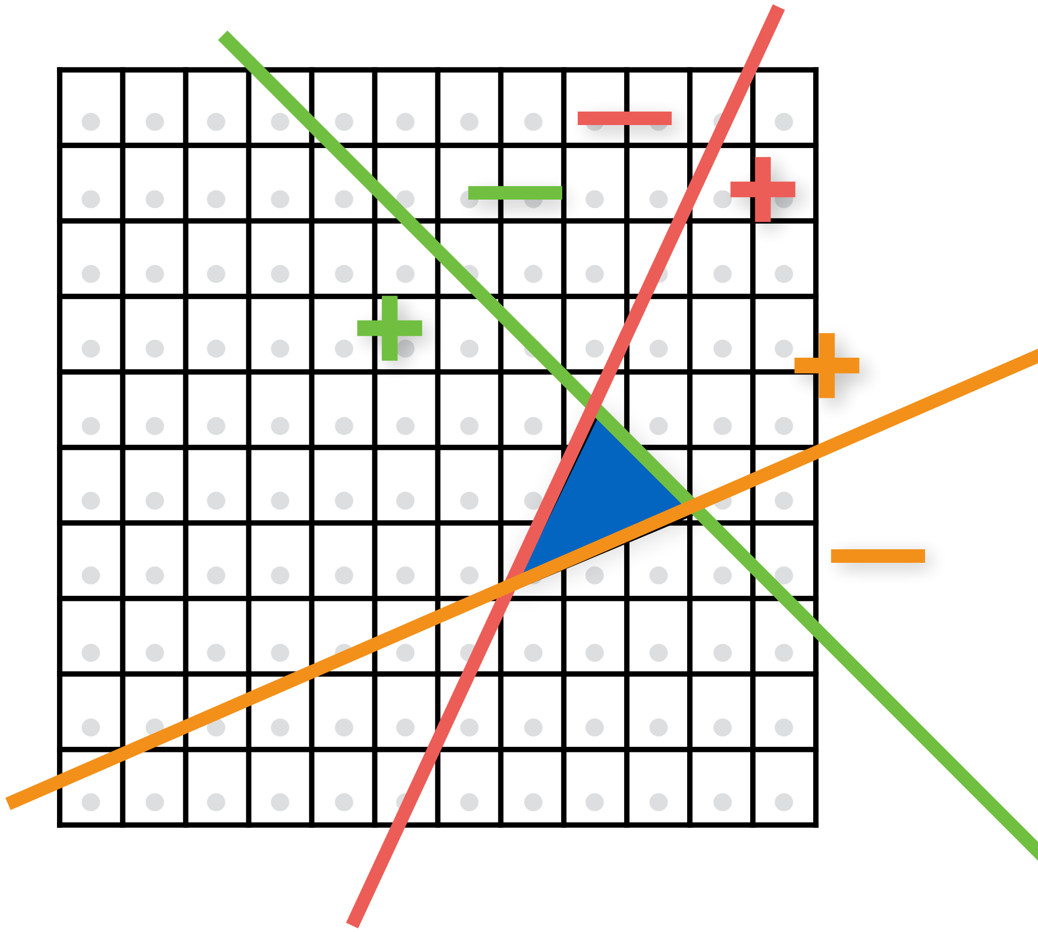
Problem?



Brute Force Triangle Rasterizer

```
for each pixel (x,y)
   evaluate edge functions at pixel center
   if all non-negative, pixel is in
```

If the triangle is small, lots of useless computation (if we really test all pixels)

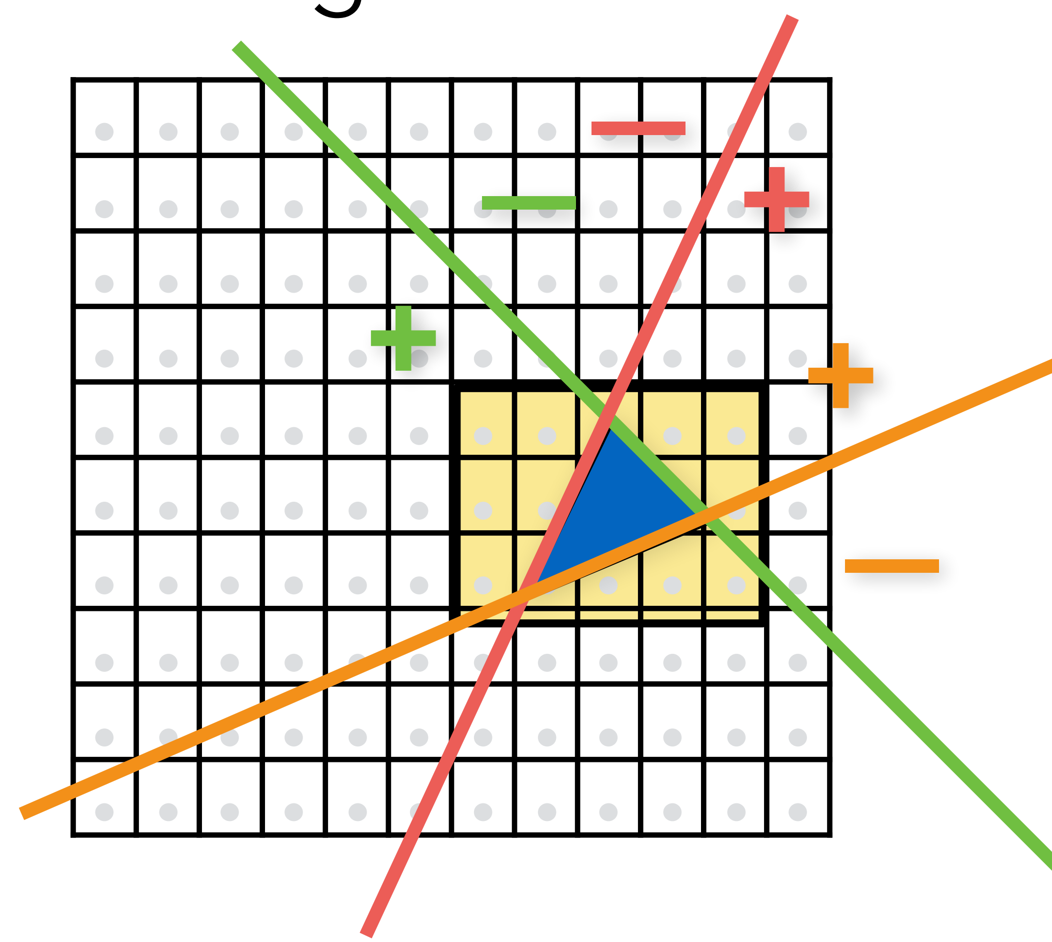


After a slide by Frédo Durand

Easy Optimization

Improvement: only scan over pixels overlapping the **screen bounding box** (BBox) of the triangle

- how do we get such a bounding box?
  - $x_{min}, x_{max}, y_{min}, y_{max}$ of projected triangle vertices



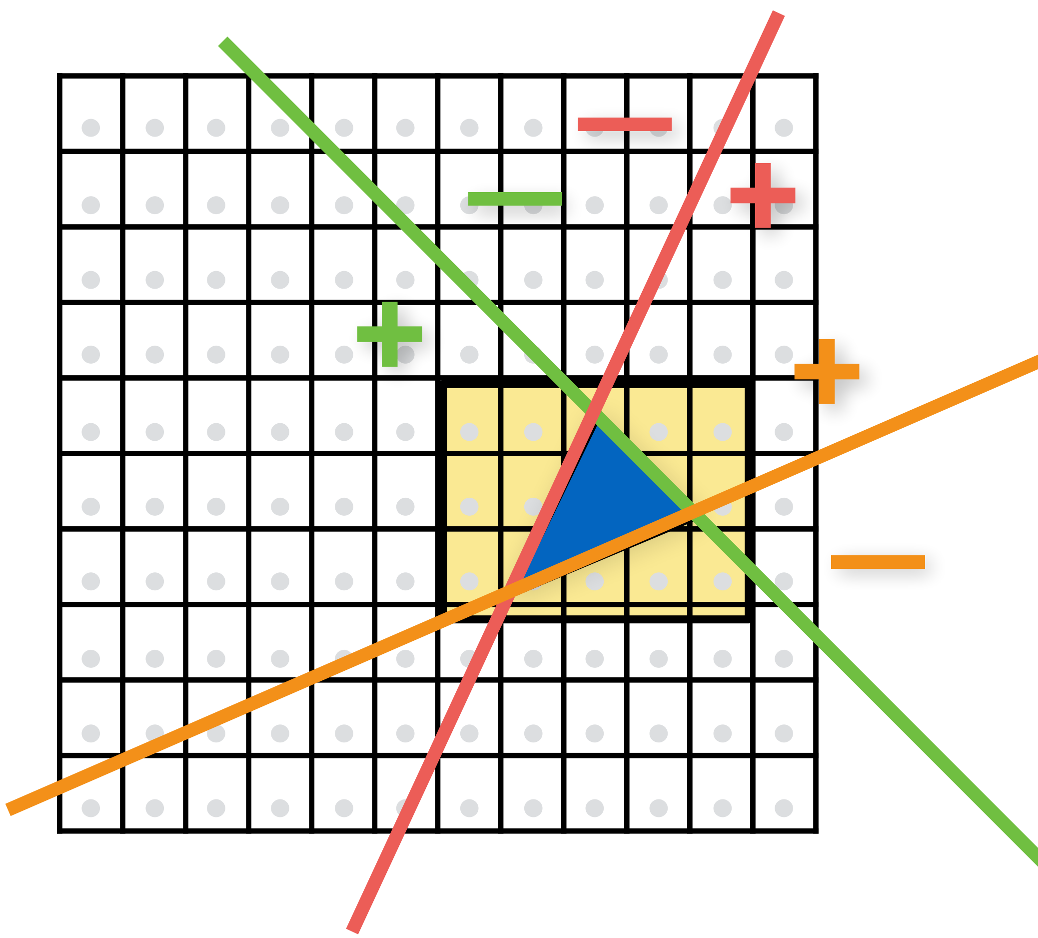
After a slide by Frédo Durand

Triangle Rasterization Pseudocode

```
for every triangle
    project vertices, compute the Ei
    compute bbox, clip box to screen limits
    for all pixels in box
        evaluate Ei functions
        if all > 0
            framebuffer[x,y] = c;
```

After a slide by Frédo Durand

Bounding box clipping is easy, just clamp the coordinates to the screen rectangle



Note: no visibility!

Triangle Rasterization Pseudocode

Jaakko Lehtinen
@jaakkolehtinen

Follow

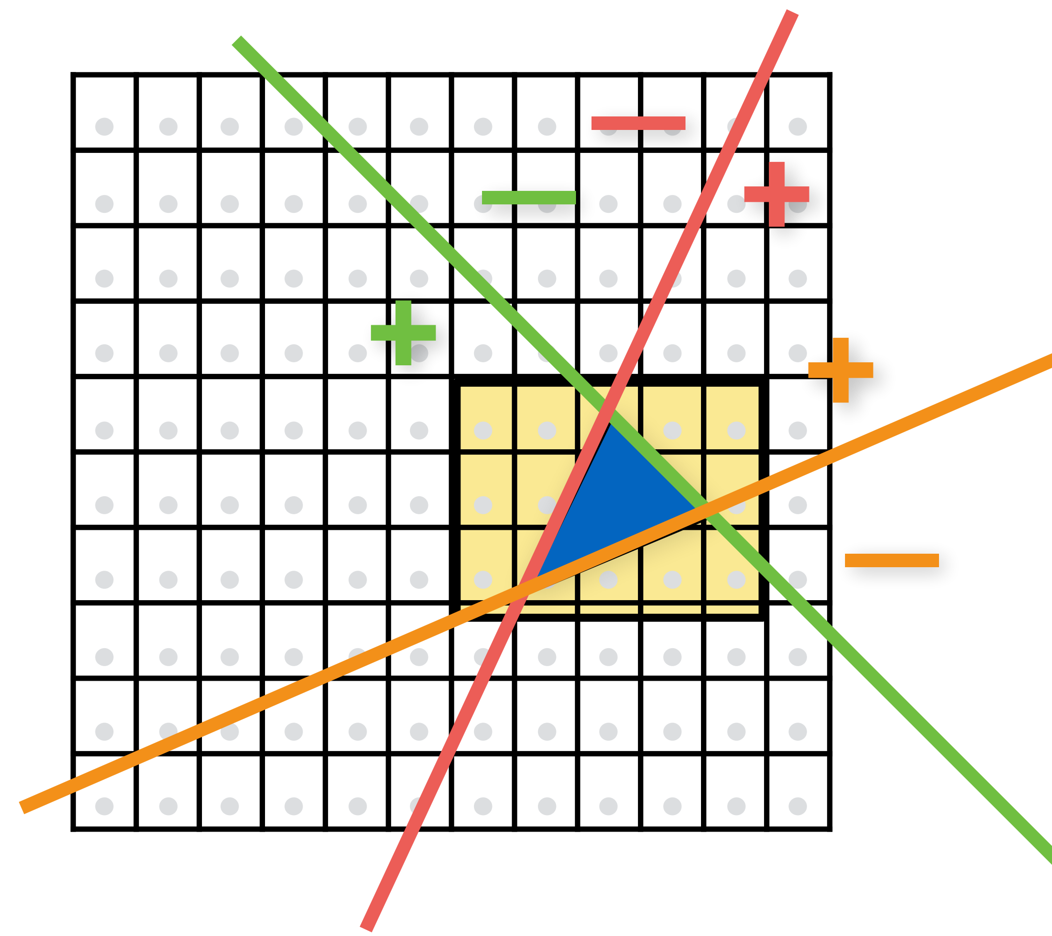
My most succinct rasterizer yet. Still kind of readable.

```
% triangle vertices in [-1,1]^2
P = [-1 -0.51; -0.3 0.77; -0.1 -0.32];
% pixel samples 0.1 units apart
[px,py] = meshgrid(-1+0.1/2:0.1:1, -1+0.1/2:0.1:1);
% form edge functions
E = [P([2 3 1],2), P([1 2 3],1)] - [P([1 2 3],2), P([2 3 1],1)];
E(:,3) = -diag(E*P');
% evaluate edge functions at sample points
res = E*[px(:)'; py(:)'; ones(1,length(px(:)))];
% which ones are in?
inside = min(res,[],1) > 0;
```

7:32 AM - 13 Nov 2015

7 Retweets **23** Likes

7 23



Options, Optimizations and Caveats

Could use barycentric coordinates instead of edge functions

Can operate on blocks of pixels before going per-pixel

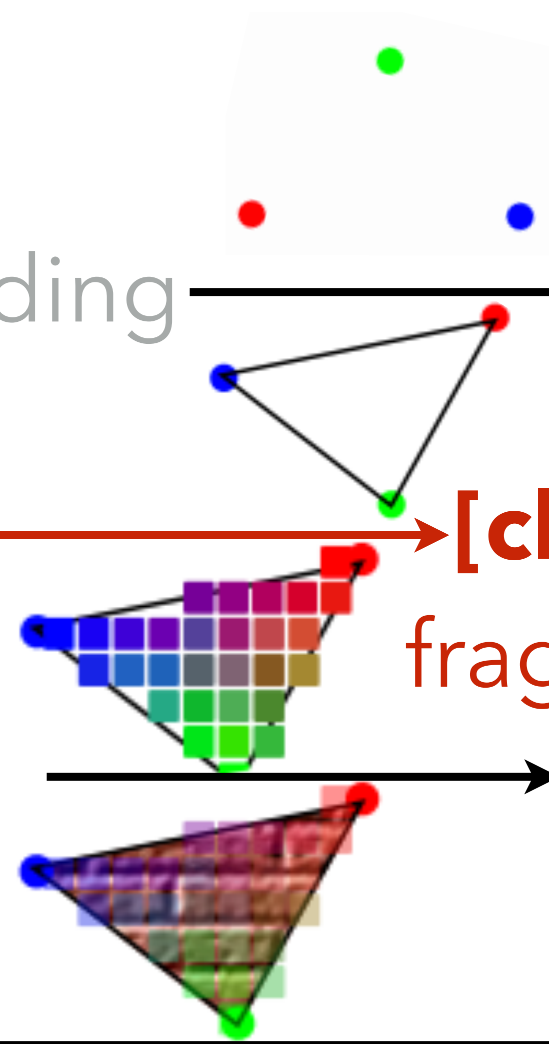
Incremental computation possible, but...

- ... parallelism most important for hardware

Be careful about pixels on triangle edges!

- pixels should only be drawn once, but no holes between adjacent triangles

Programmable Rasterization Pipeline™



vertex data in

transformations & shading → **[vertex processing]**

transformed vertex data

convert to pixels → **[clipping and rasterization]**

fragments w/ interpolated data

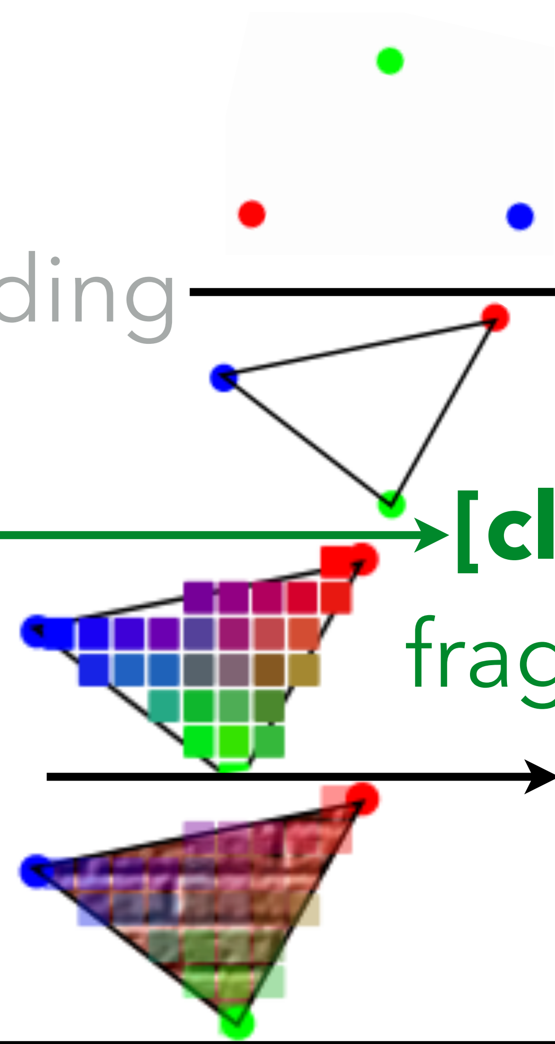
visibility & shading → **[fragment processing]**

frame buffer

user output → **[display]**

After a slide by Steve Marschner

Programmable Rasterization Pipeline™



vertex data in

transformations & shading → **[vertex processing]**

transformed vertex data

convert to pixels → **[clipping and rasterization]**

fragments w/ interpolated data

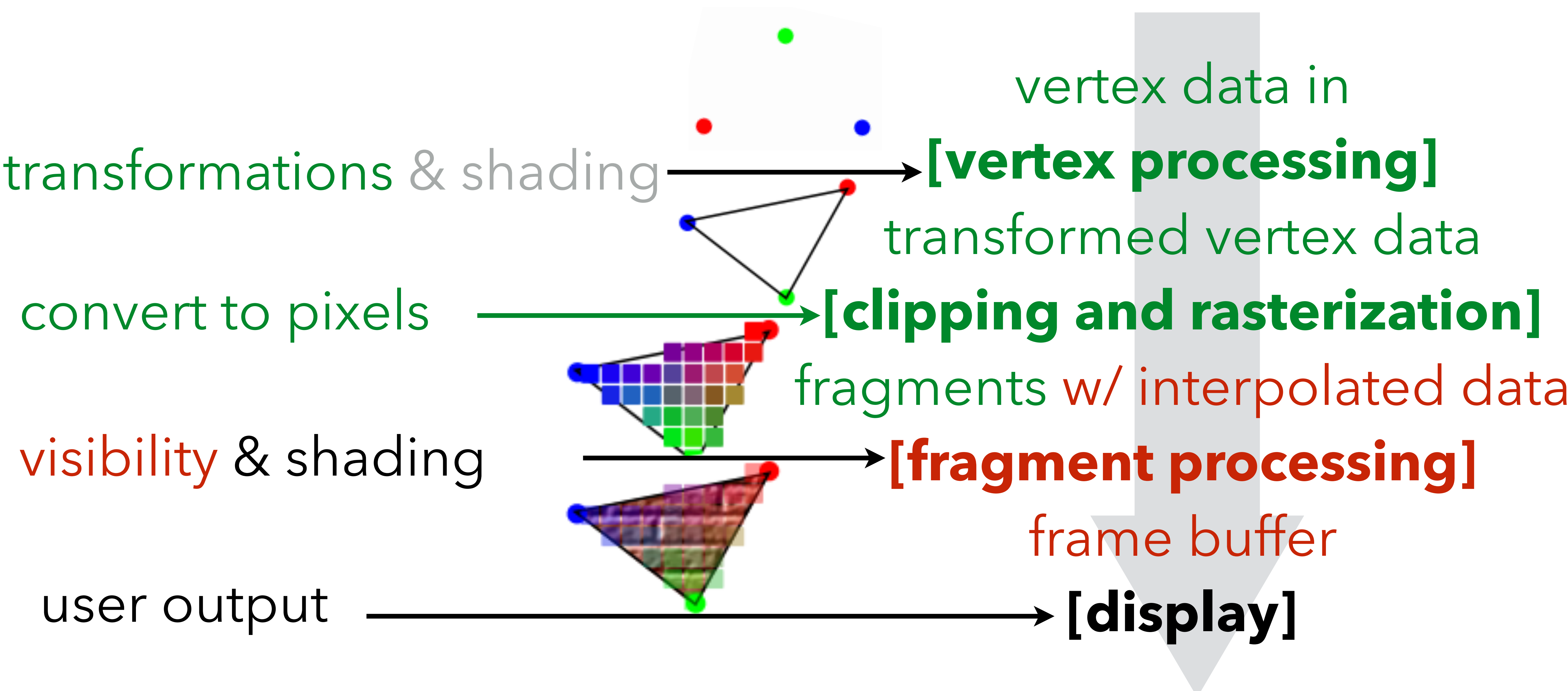
visibility & shading → **[fragment processing]**

frame buffer

user output → **[display]**

After a slide by Steve Marschner

Programmable Rasterization Pipeline™



After a slide by Steve Marschner

Programmable Rasterization Pipeline™

```
for(each triangle)
    transform vertices into eye space
    project vertices to image space
    for(each pixel x,y)
        if(x,y in triangle)
            compute z
            if(z < zbuffer[x,y])
                zbuffer[x,y] = z
                framebuffer[x,y] = shade()
```

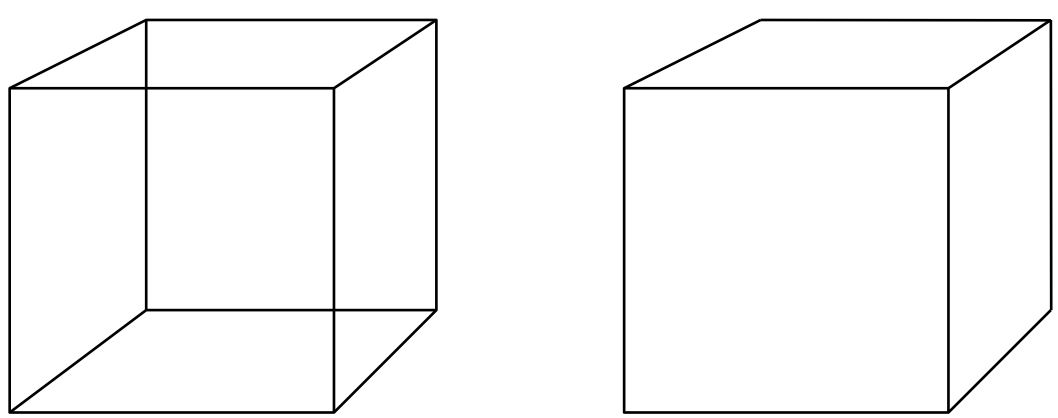
Hidden Surface Elimination

Visible Surface Determination

Hidden Surface Elimination

We have discussed how to map primitives to image space

- projection and perspective are depth cues
- occlusion is another very important cue

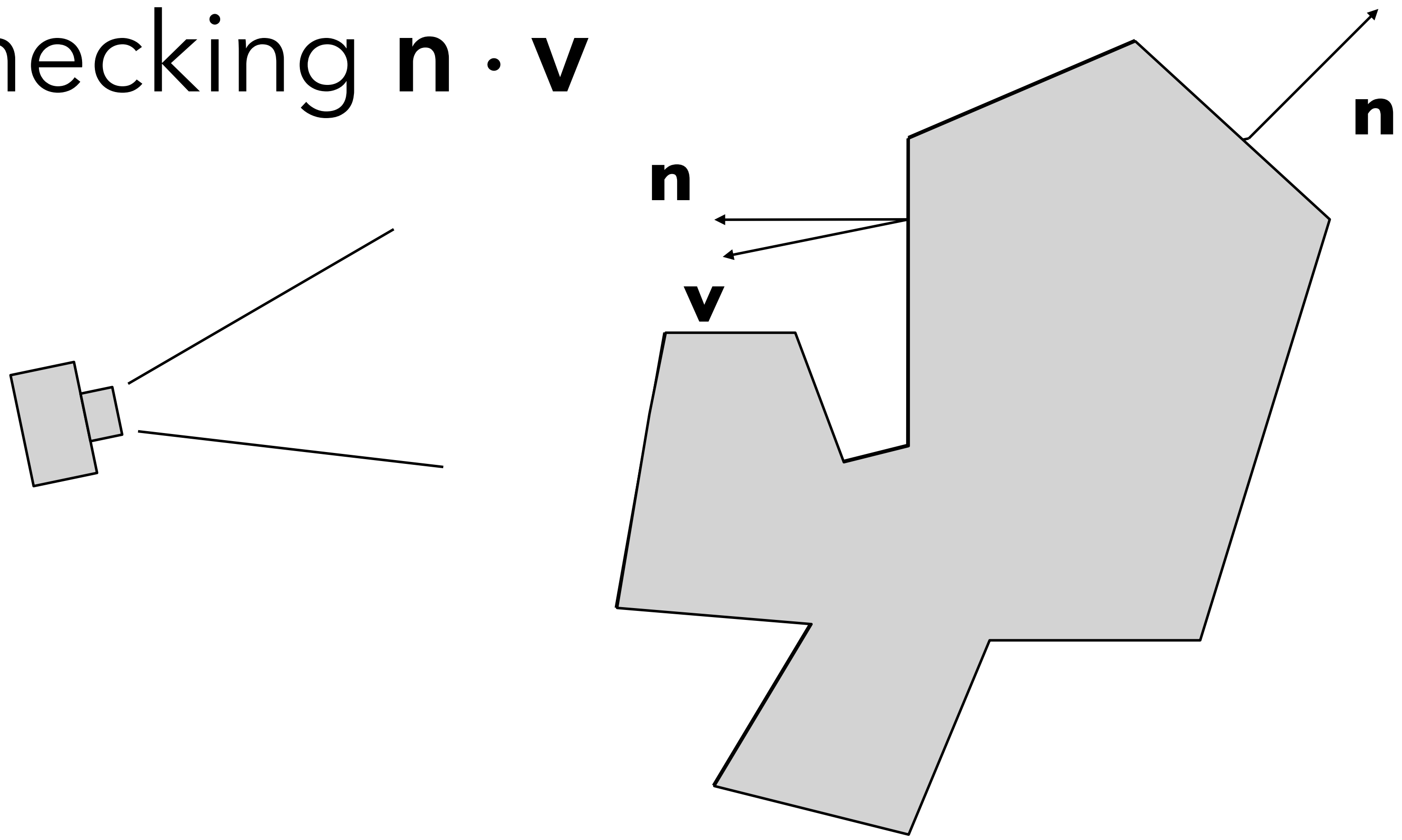


After a slide by Steve Marschner

Backface Culling

For closed shapes you will never see the inside of the shape

- therefore, only draw surfaces that face the camera
- implement by checking $\mathbf{n} \cdot \mathbf{v}$

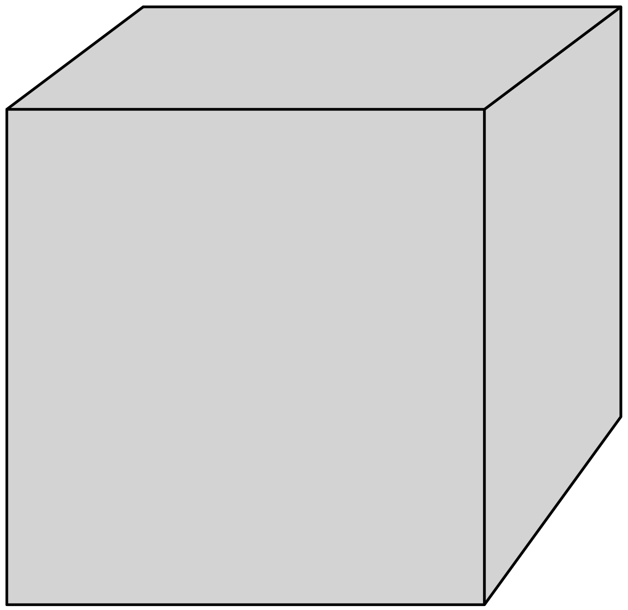


After a slide by Steve Marschner

Painter's Algorithm

Simplest way to treat hidden surfaces:

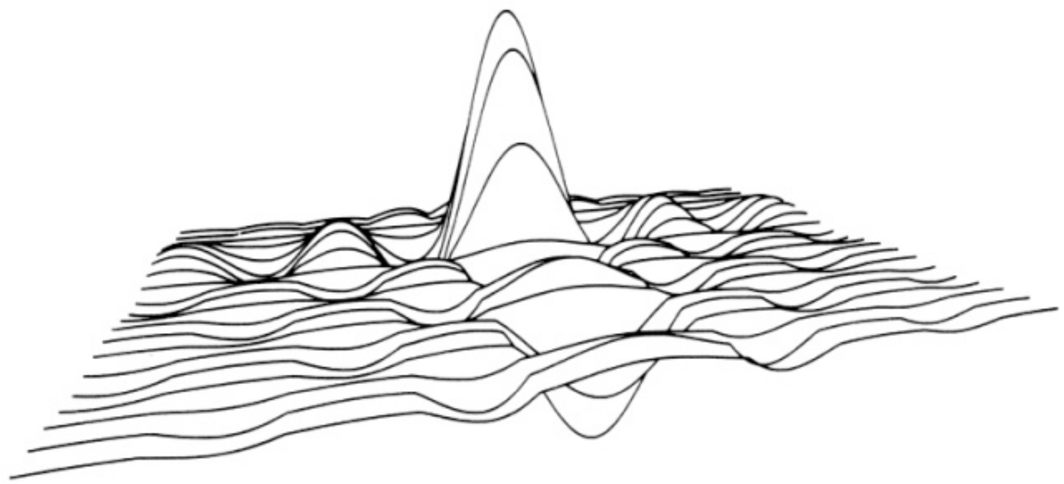
- draw from back to front, overwriting the framebuffer along the way

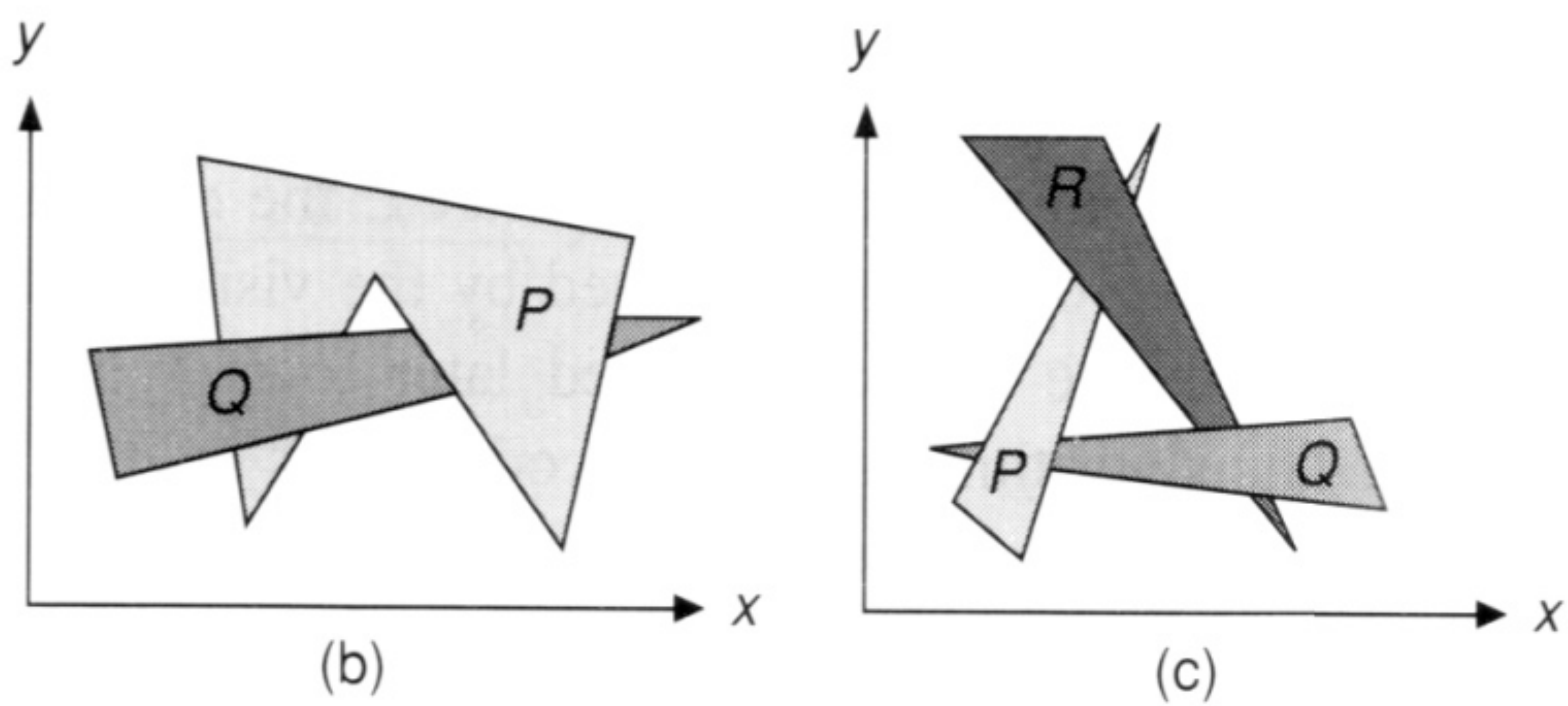


After a slide by Steve Marschner

Painter's Algorithm

Useful when a valid order is easy to come by





[Foley et al.]

After a slide by Steve Marschner

The z-buffer (a.k.a. depth buffer)

In many/most applications, maintaining a z sort is too expensive

- changes all the time as the view changes
- many data structures exist, but complex

Solution: draw in any order, keep track of closest

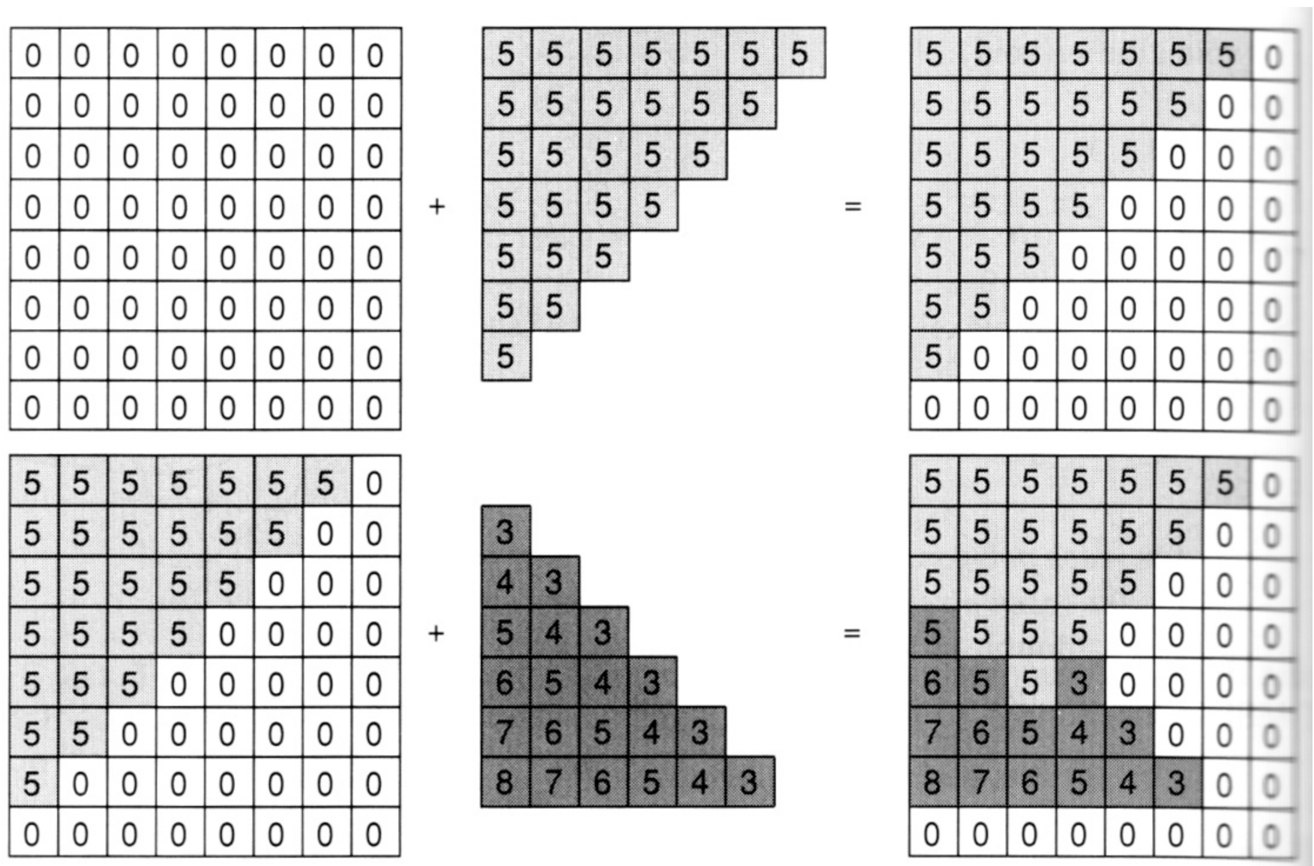
- allocate extra channel per pixel to keep track of **closest depth so far**
- when drawing, compare object's depth to current closest depth, and discard if greater*
- works just like any other compositing operation

After a slide by Steve Marschner

The z-buffer

Another example of a memory-intensive, brute force approach that works and has become the standard

After a slide by Steve Marschner

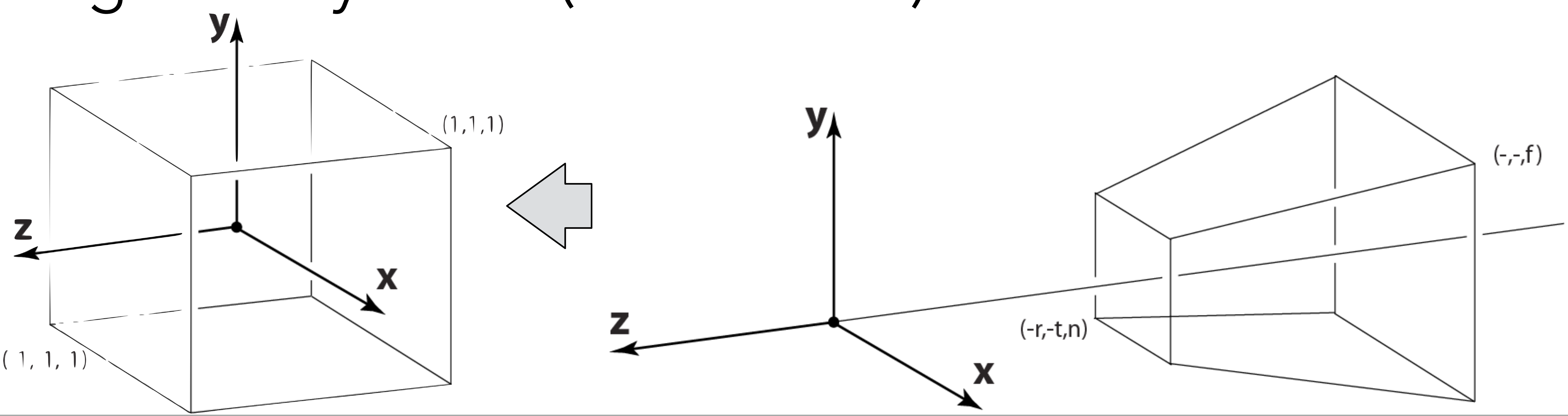


[Foley et al.]

Precision in the z-buffer

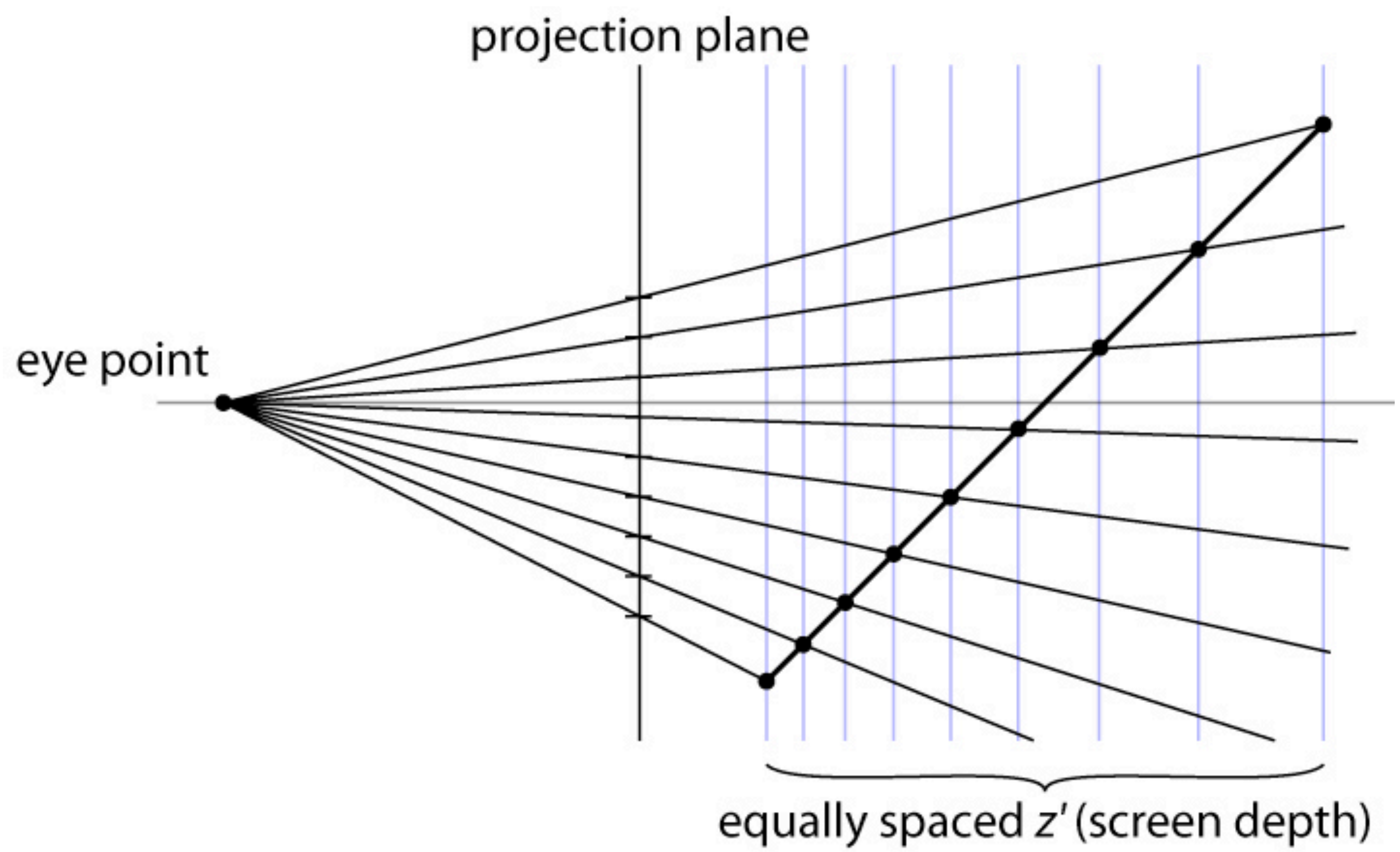
The precision is distributed between the near and far clipping planes

- this is why these planes have to exist
- also why you can't always just set them to very small and very large distances
- generally use z′ (not world z) in z buffer



After a slide by Steve Marschner

Interpolating in Projection



linear interp. in screen space ≠ linear interp. in world (eye) space

After a slide by Steve Marschner

Pipeline for Basic z-buffer

Vertex stage (input: position / vtx; color / tri)

- transform position (object to screen space)
- pass through color

Rasterizer

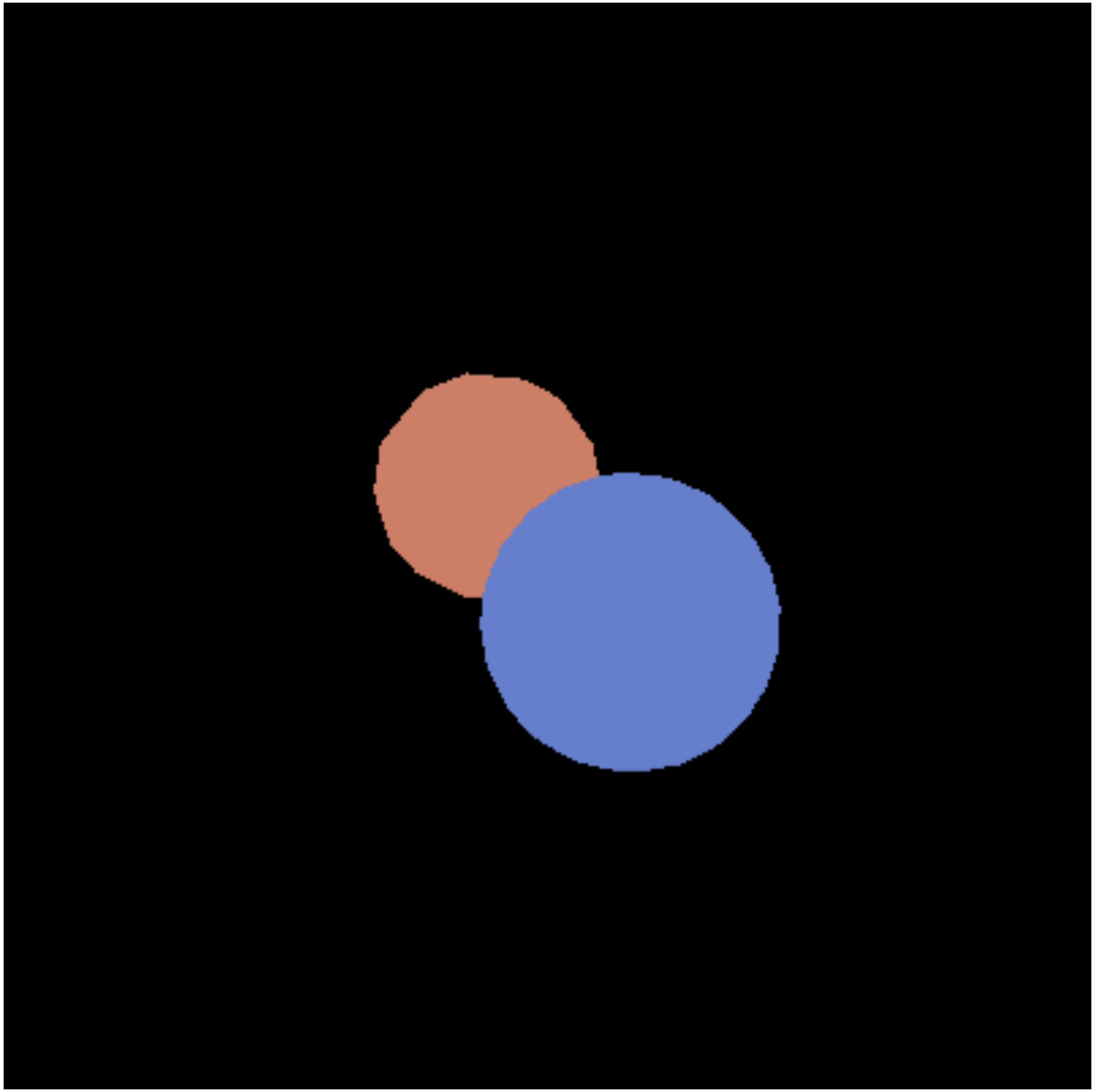
- interpolated parameter: z′ (screen z)
- pass through color

Fragment stage (output: color, z′)

- write to frame buffer only if interpolated z′ < current z′

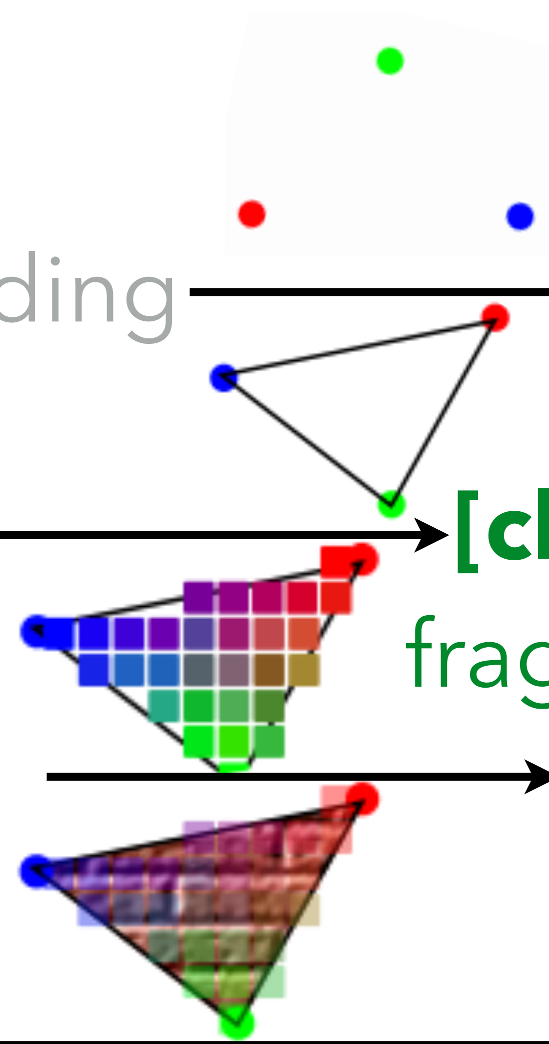
After a slide by Steve Marschner

Result of Basic z-buffer Pipeline



After a slide by Steve Marschner

Programmable Rasterization Pipeline™



vertex data in

transformations & shading → **[vertex processing]**

transformed vertex data

convert to pixels → **[clipping and rasterization]**

fragments w/ interpolated data

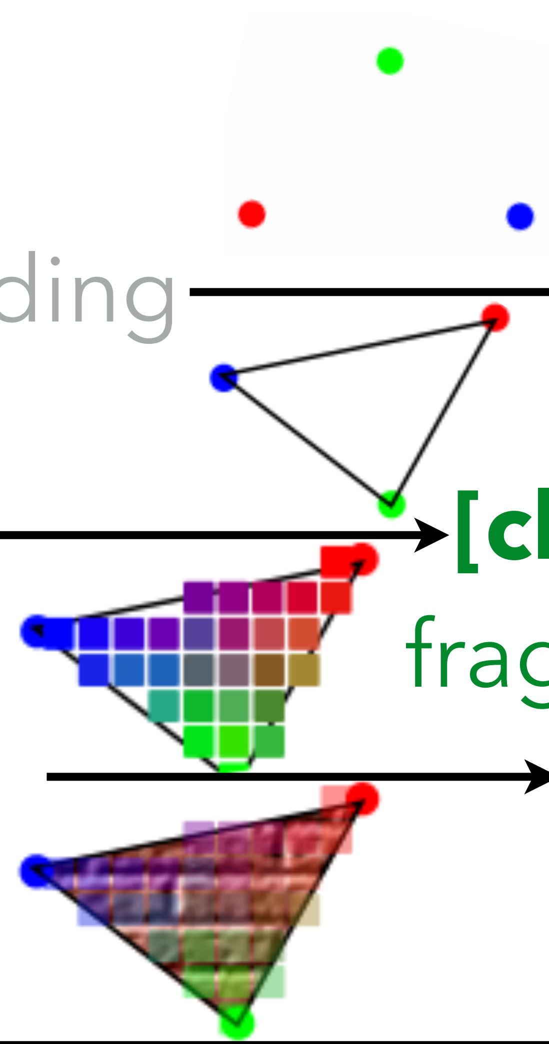
visibility & shading → **[fragment processing]**

frame buffer

user output → **[display]**

After a slide by Steve Marschner

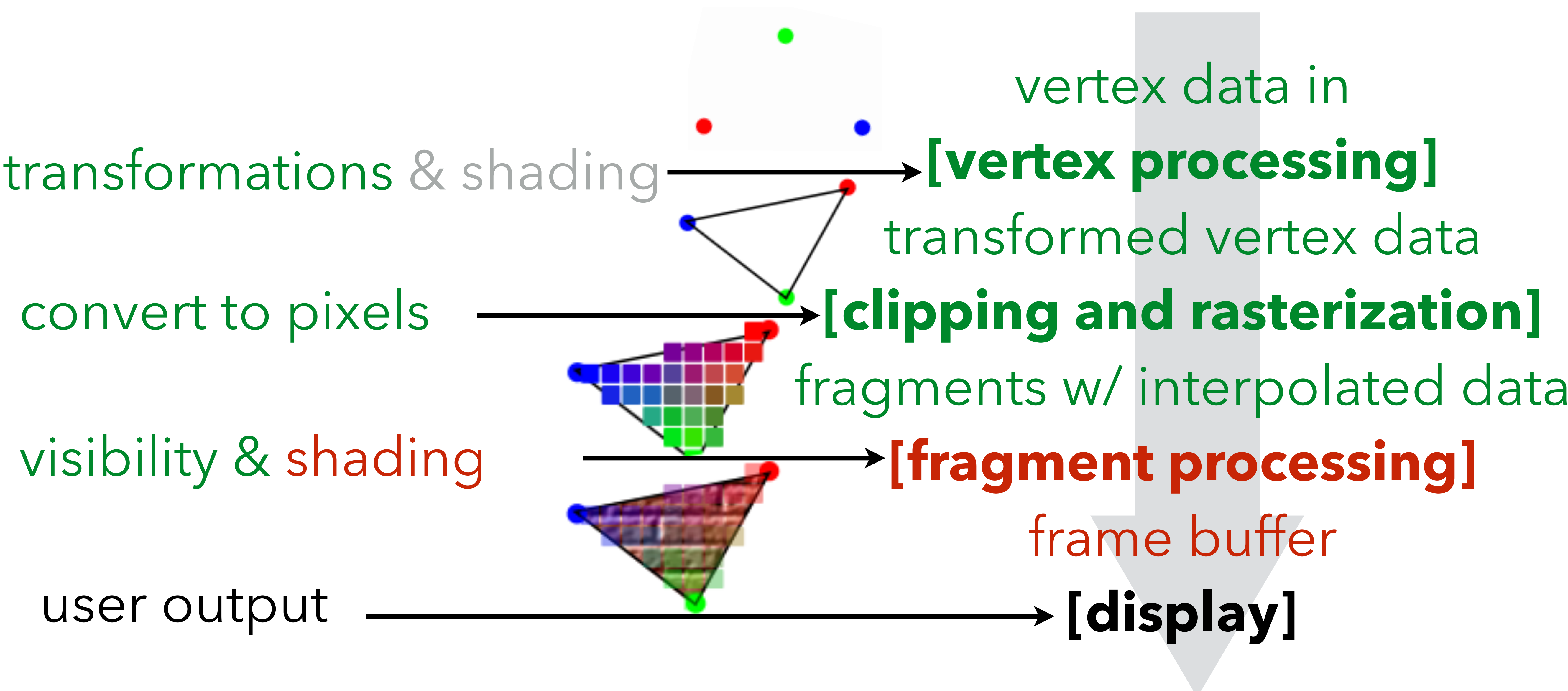
Programmable Rasterization Pipeline™



vertex data in
transformations & shading → **[vertex processing]**
transformed vertex data
convert to pixels → **[clipping and rasterization]**
fragments w/ interpolated data
visibility & shading → **[fragment processing]**
frame buffer
user output → **[display]**

After a slide by Steve Marschner

Programmable Rasterization Pipeline™



After a slide by Steve Marschner

Programmable Rasterization Pipeline™

```
for(each triangle)
    transform vertices into eye space
    project vertices to image space
    for(each pixel x,y)
        if(x,y in triangle)
            compute z
            if(z < zbuffer[x,y])
                zbuffer[x,y] = z
                framebuffer[x,y] = shade()
```