Lecture #15

The Rendering Pipeline

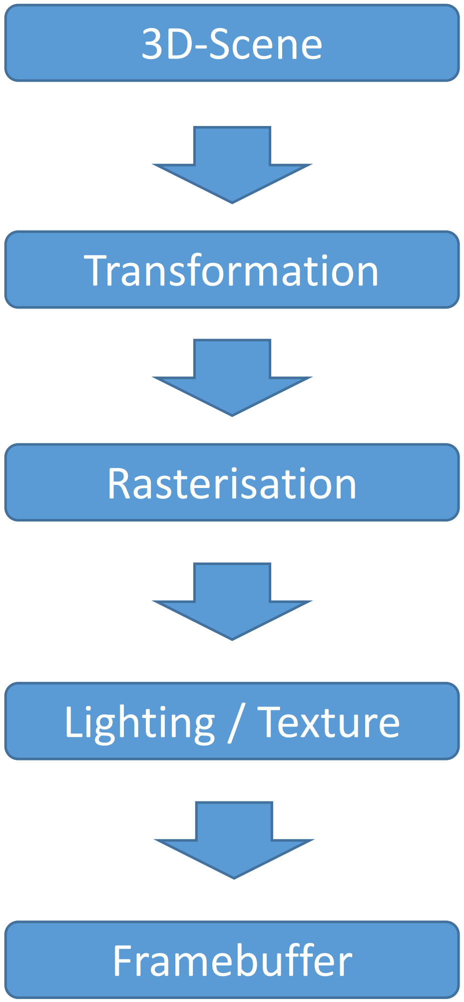
Computer Graphics

Winter Term 2016/17

Marc Stamminger / Roberto Grosso

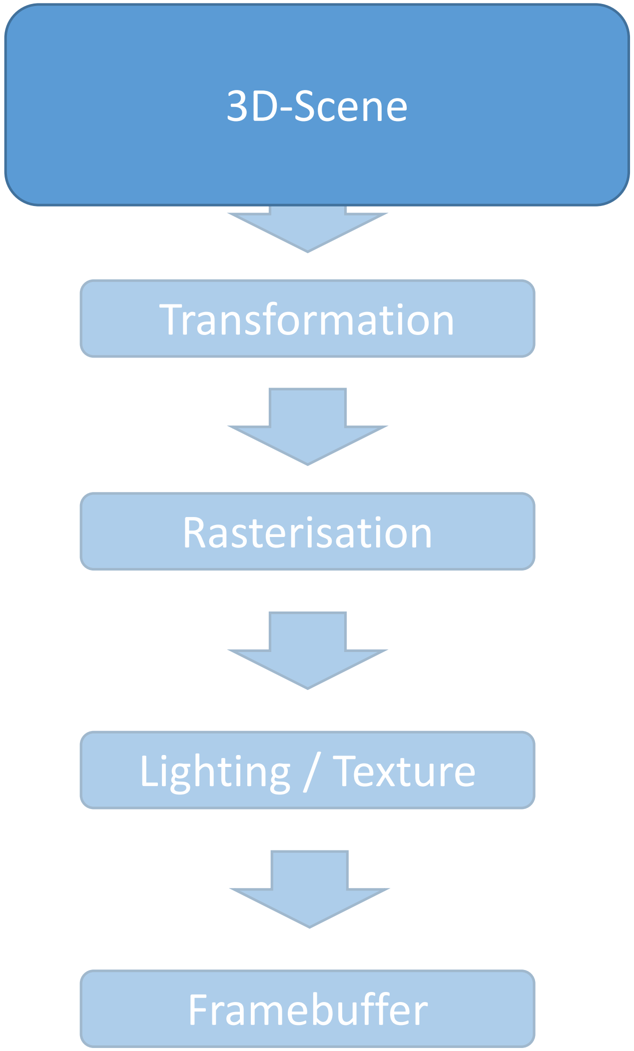
The Rendering Pipeline

- Coarse Version



The Rendering Pipeline

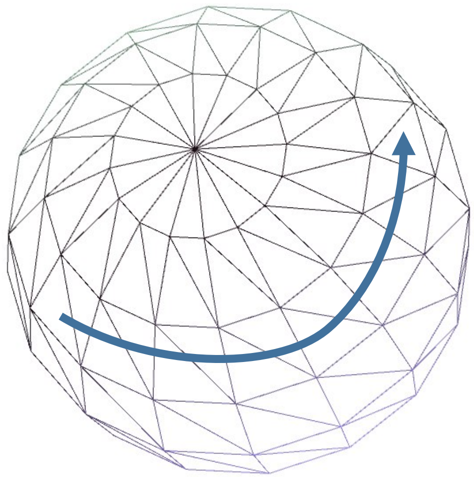
- Coarse Version



3D Objects

- From simple shapes, e.g. a sphere

- Generate using Quad or Triangle Strips
  - enumerate vertices in proper order
  - most vertices appear twice
  - Triangle Fan needed for caps
    - degenerate Strips also possible,
      where ring at pole collapses to point
  - no index buffer needed



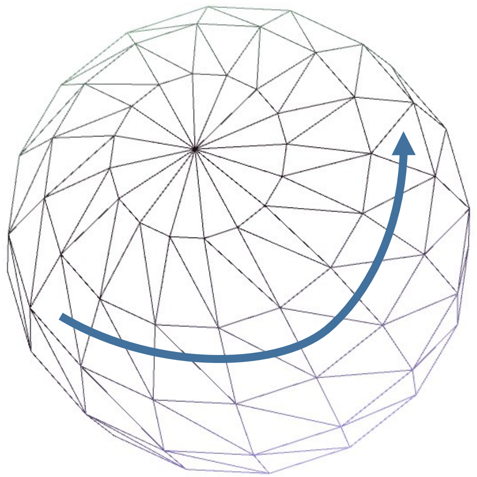
By MaxDZ8 (Snapshot from a program I’ve written.) [Public domain], via Wikimedia Commons

```
var v = [...];
var vbo = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vbo);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(v), gl.STATIC_DRAW);
gl.drawArrays(gl.QUAD_STRIP,0,v.length);
```

3D Objects

- From simple shapes, e.g. a sphere

- Generate using an Indexed Face Set
  - two buffers needed (vertices and indices)
  - every vertex appears only once



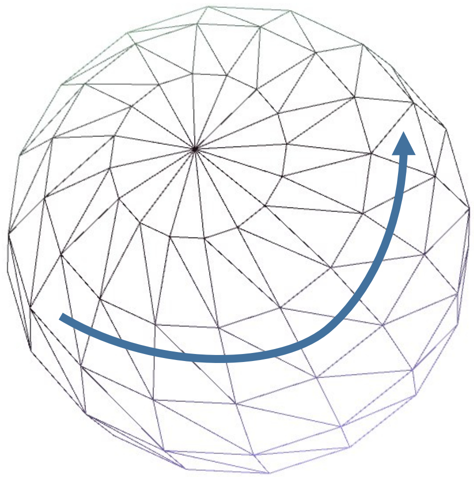
[By MaxDZ8 (Snapshot from a program I've written.) [Public domain], via Wikimedia Commons]

```
var v = [...];
var i = [...];

var vbo = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vbo);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(v), gl.STATIC_DRAW);

var ibo = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, ibo);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(i), gl.STATIC_DRAW);

gl.drawElements(gl.TRIANGLES, 6, gl.UNSIGNED_SHORT, 0);
```

3D Objects

- From simple shapes, e.g. a sphere

- Generate using an Indexed Face Set **and Strips**
  - two buffers needed (vertices and indices)
  - every vertex appears only once
  - smaller index buffer by using strips
  - strips restart using degenerate triangles: what does strip ABCDEEFFGHI generate?



By MaxDZ8 (Snapshot from a program I've written.) [Public domain], via Wikimedia Commons

```
var v = [...];
var i = [...];

var vbo = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, vbo);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(v), gl.STATIC_DRAW);

var ibo = gl.createBuffer();
gl.bindBuffer(gl.ELEMENT_ARRAY_BUFFER, ibo);
gl.bufferData(gl.ELEMENT_ARRAY_BUFFER, new Uint16Array(i), gl.STATIC_DRAW);

gl.drawElements(gl.TRIANGLE_STRIP, 6, gl.UNSIGNED_SHORT, 0);
```

3D Objects

- From files, e.g. OBJ:

```
v 1.000000 -1.000000 -1.000000
v 1.000000 -1.000000 1.000000
v -1.000000 -1.000000 1.000000
v -1.000000 -1.000000 -1.000000
v 1.000000 1.000000 -1.000000
v 0.999999 1.000000 1.000001
v -1.000000 1.000000 1.000000
v -1.000000 1.000000 -1.000000
vn -0.000000 -1.000000 0.000000
vn 0.000000 1.000000 -0.000000
vn 1.000000 0.000000 0.000000
vn -0.000000 -0.000000 1.000000
vn -1.000000 -0.000000 -0.000000
vn 0.000000 0.000000 -1.000000
f 1//1 2//1 3//1 4//1
f 5//2 8//2 7//2 6//2
f 1//3 5//3 6//3 2//3
f 2//4 6//4 7//4 3//4
f 3//5 7//5 8//5 4//5
f 5//6 1//6 4//6 8//6
```

Vertex positions

Vertex normals

Topology 3//1 means: vertex with 3^{rd} position and 1^{st} normal

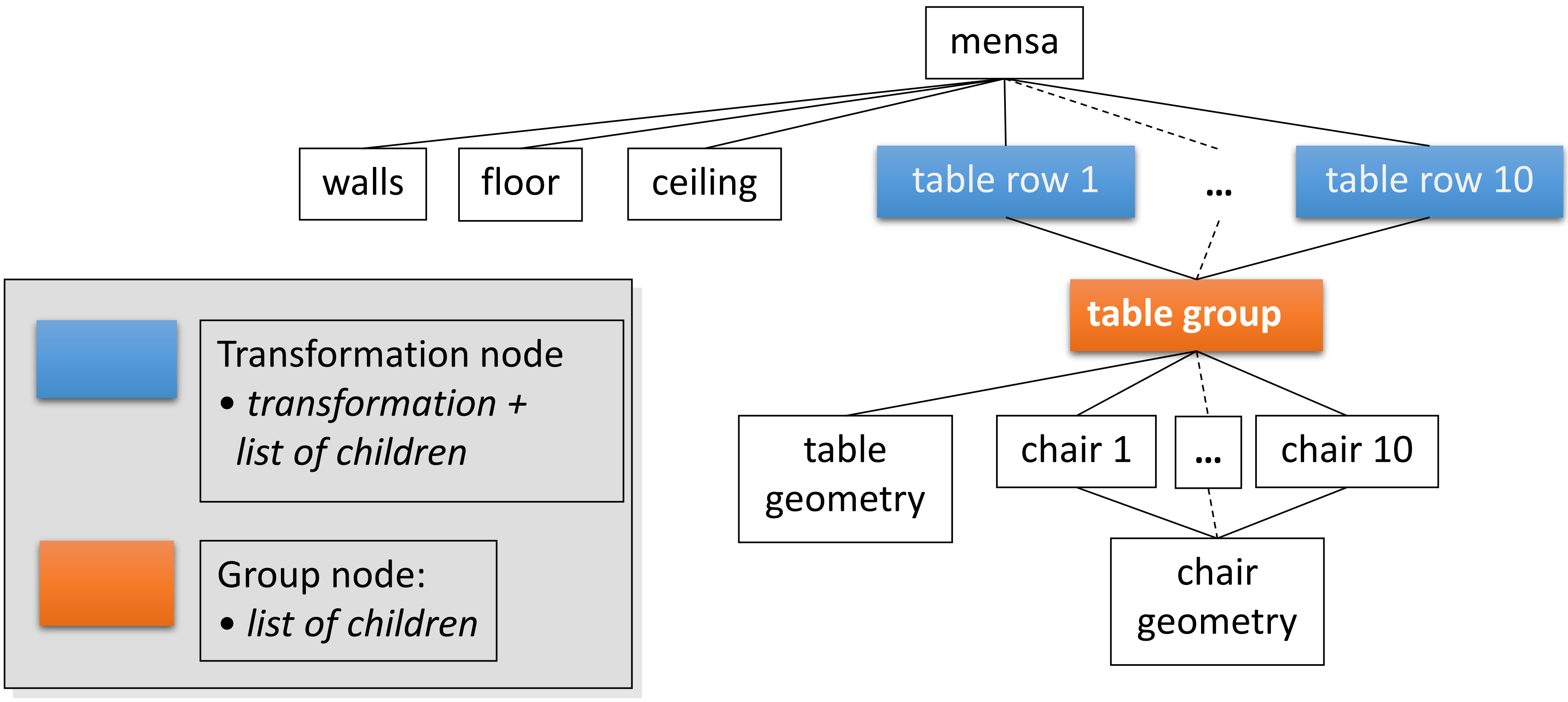
OBJ file format

3D Scenes

- Combine many objects to a 3D Scene
- Each object has
  - material properties
  - a modeling transformation that positions the object in the scene
- **Instancing:**
  - position copies of an object under various transformations
  - supported by OpenGL
- **Scene Graph:**
  - Store the material and transformations in a hierarchy
  - instancing by multiple references to single objects

3D Scenes

- Scene Graph



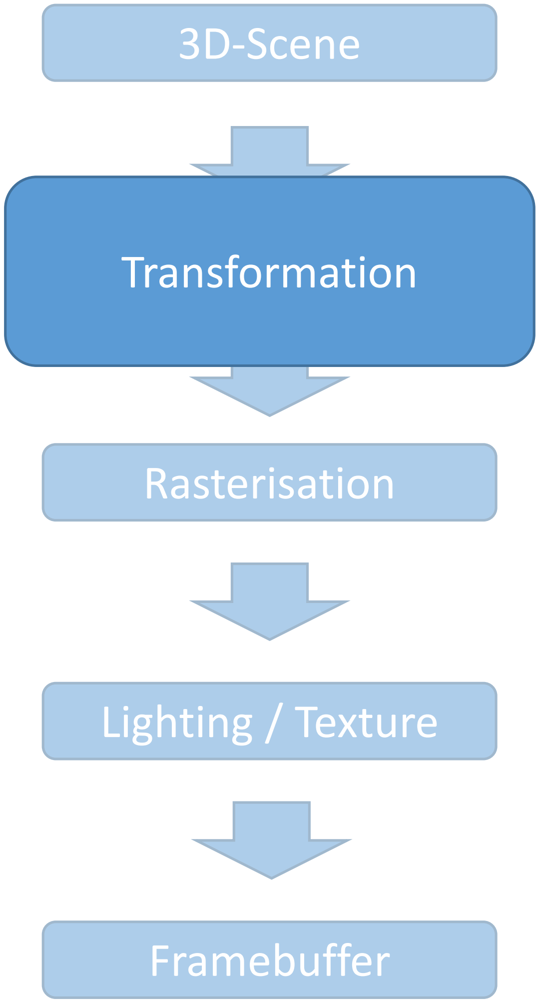
3D Scenes

- Scene Graph traversal:
  - to render a scene graph, we do a depth traversal, always down to the leafes
  - on the way, we gather transformations using a matrix stack (or similar)
  - at each leaf, we have
    - an object, usually as triangle mesh → bind corresponding buffer
    - its modeling transformation → set as model matrix for shader
    - and material parameters → set shader plus its parameters

```
class Node { ... }

class TransformNode extends Node {
    Transformation t;
    void render(MatrixStack stack) {
    stack.push(t);
    for (each child i)
         i.render(stack);
    stack.pop();
}   }

class Object extends Node {
    void render(MatrixStack stack) {
         render Object with modeling matrix stack.top();
}   }
```

The Rendering Pipeline

- Coarse Version

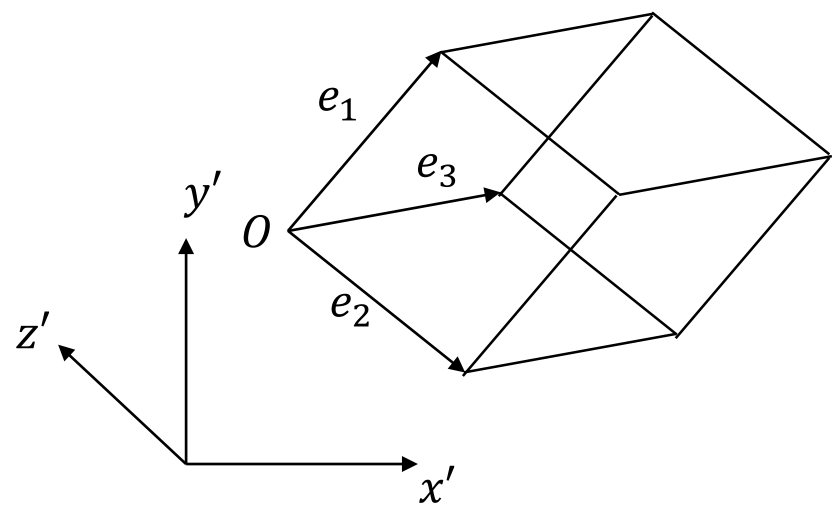


Transformations

- **Affine Transformations**:
  Described using homogeneous coordinates and a matrix

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} \vdots & \vdots & \vdots & O_1 \\ e_1 & e_2 & e_3 & O_2 \\ \vdots & \vdots & \vdots & O_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

- used as
  - modeling matrices
  - viewing matrix



Transformations

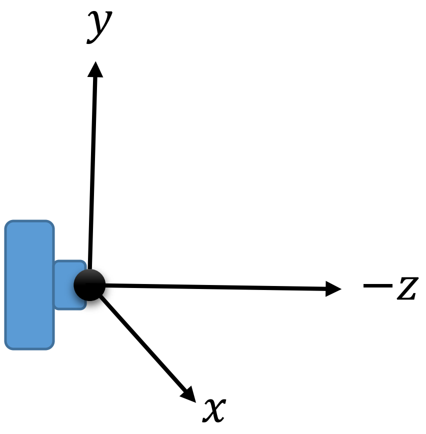
- **Affine Transformations**:
  Special case: Rotations

- Representations:
  - Orthogonal matrix
  - Euler angles (e.g.: yaw, pitch, and roll)
    $$R = R_z(\alpha)R_y(\beta)R_z(\gamma)$$
  - axis + angle
    - see slides #6
  - or quaternions
    - best for interpolation
    - see slides #6

- Modeling Transformation is usually translation + rotation + scale
- Viewing transformation is a translation + rotation

Transformations

- **Viewing:**
  coordinate transformation to coordinate system aligned with camera
- sets the “extrinsic” camera parameters
- usually defined by
  - camera position (eye)
  - view direction (gaze)
  - up-vector (up)
- or by
  - eye
  - look-at (at)
  - up
  - in this case: gaze = at-eye



Transformations

- Projective Transformations: arbitrary homogeneous matrix

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} \ddots & & & \\ & \ddots & & \\ & & \ddots & \\ & & & \ddots \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = M \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

- Interpretation:
  - Points with $w \neq 0$: points in 3D
  - Points with $w = 0$: points at infinity = directions = vectors
  - → first column of M: image of $(1,0,0,0)^T$, second: …
  - $(1,0,0,0)^T$ is intersection of lines parallel to x-axis
  - if $M_{30} \neq 0$, these parallel lines will intersect in finite space
    → parallel lines don't remain parallel
  - columns of M correspond to vanishing points in perspective

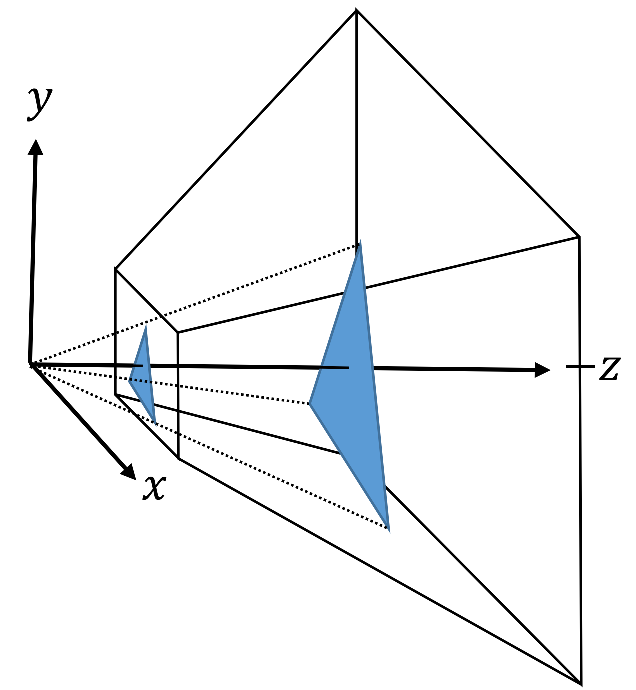
Transformations

- very simple perspective = division by z
- as a matrix:

$$M_{perspective} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

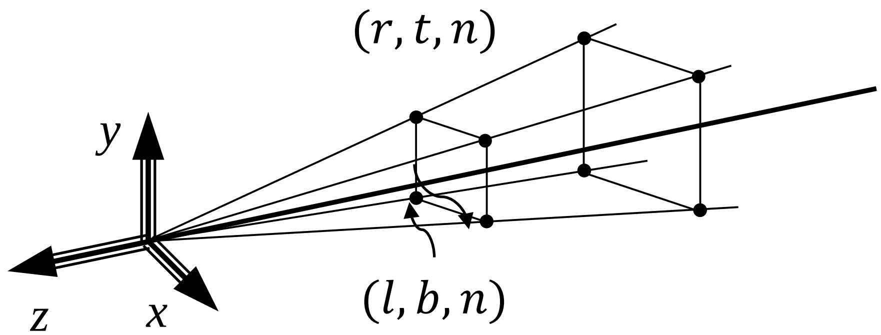
- generates non-linear z: $z \rightarrow 1 - \frac{1}{z}$

perspective projection



Transformations

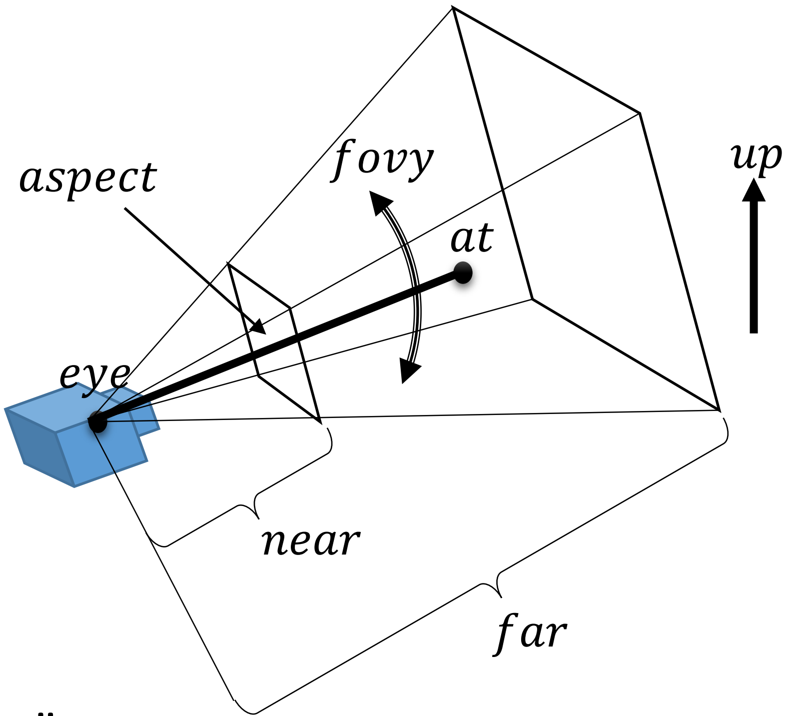
- Plus selection of window on image plane, plus selection of z-range:



- $M_{perspective}(l, r, b, t, n, f) = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{l+r}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{b+t}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$

Transformations

- Perspective matrix defines “intrinsic” camera parameters:
  - field of view (fovy)
    → wide angle lens / tele lens
  - aspect ratio (ratio)
    → width over height of image
  - near and far plane
    → z-range mapped to [-1,1]



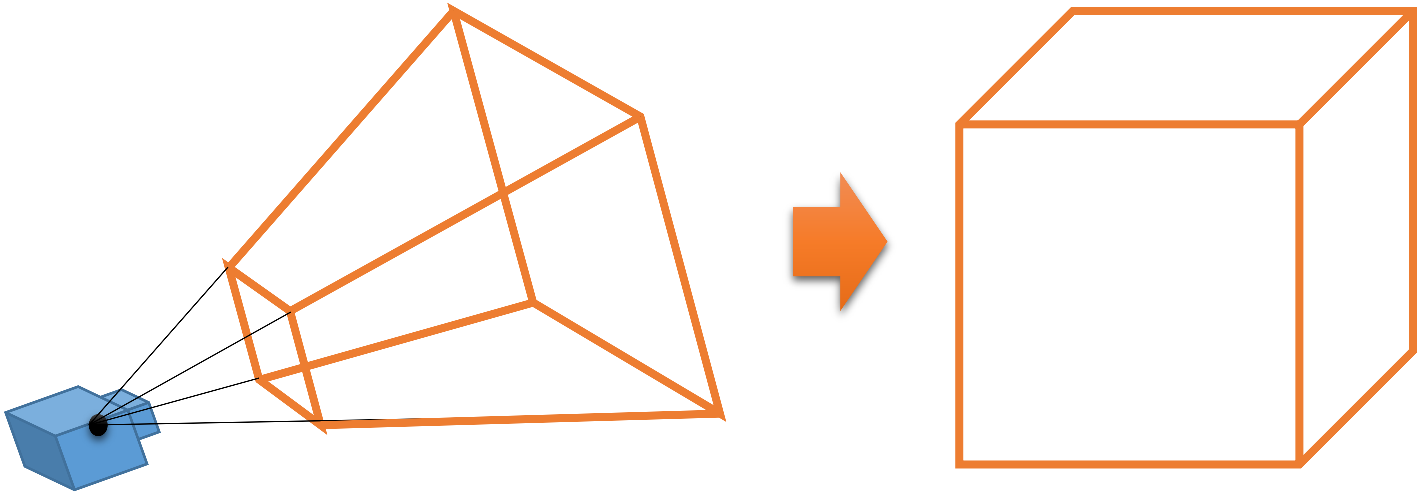
- “fovy” defines “top” and “bottom”
- then “aspect” defines “left” and “right”

Transformations

- These three transformations have to be applied in right order:

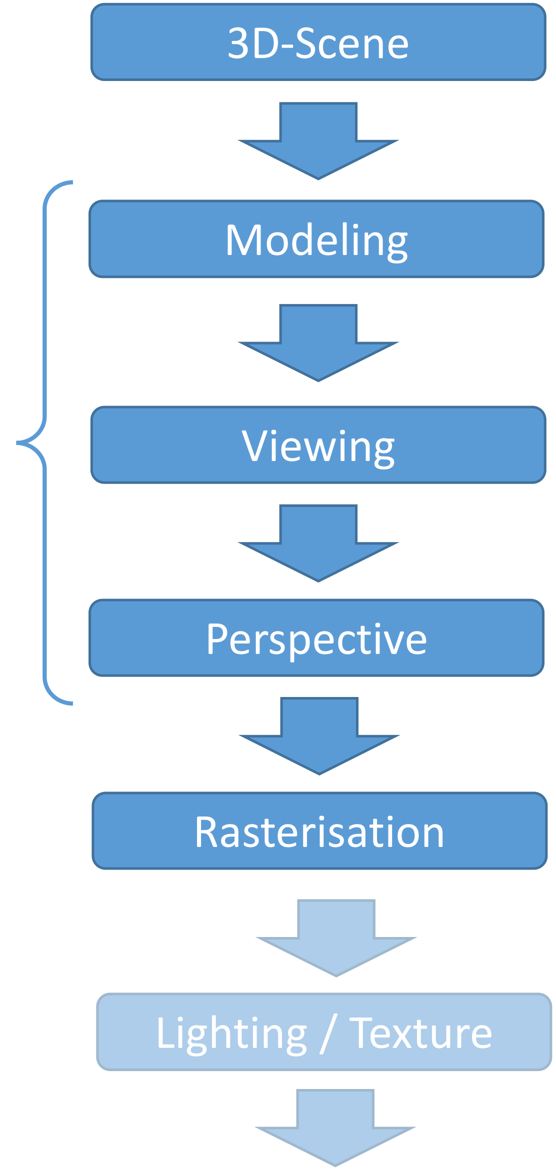
$$P\ V\ M \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

- **M**odeling, **V**iewing, **P**erspective
- Note: right-most matrix is applied **first!**
- M is the first to “see” the point
- Matrix PVM directly transforms a point to the final “canonical view volume”



Transformations

- Transformations are a pipeline on its own:

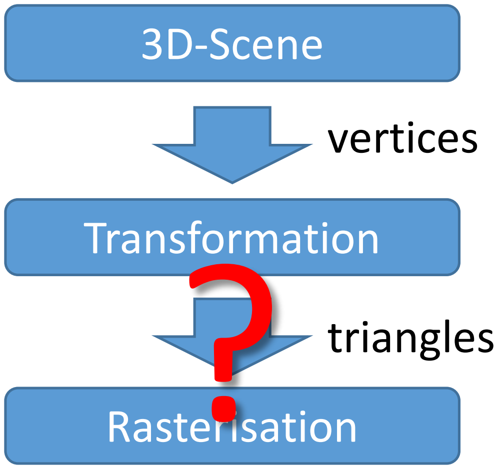


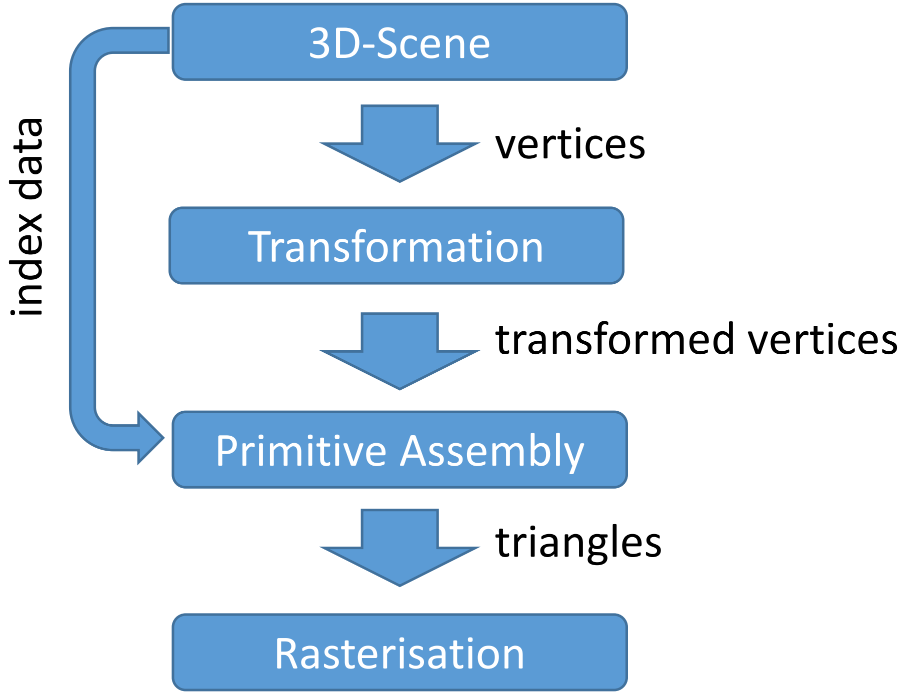
- in OpenGL, these transformations happen in the vertex shader → later
- matrices are passed as **uniforms**

```
// vertex shader – simplest version
attribute vec4 pos;
uniform mat4 PVM;

void main(void) {
    gl_Position = PVM * pos;
}
```

Primitive Assembly

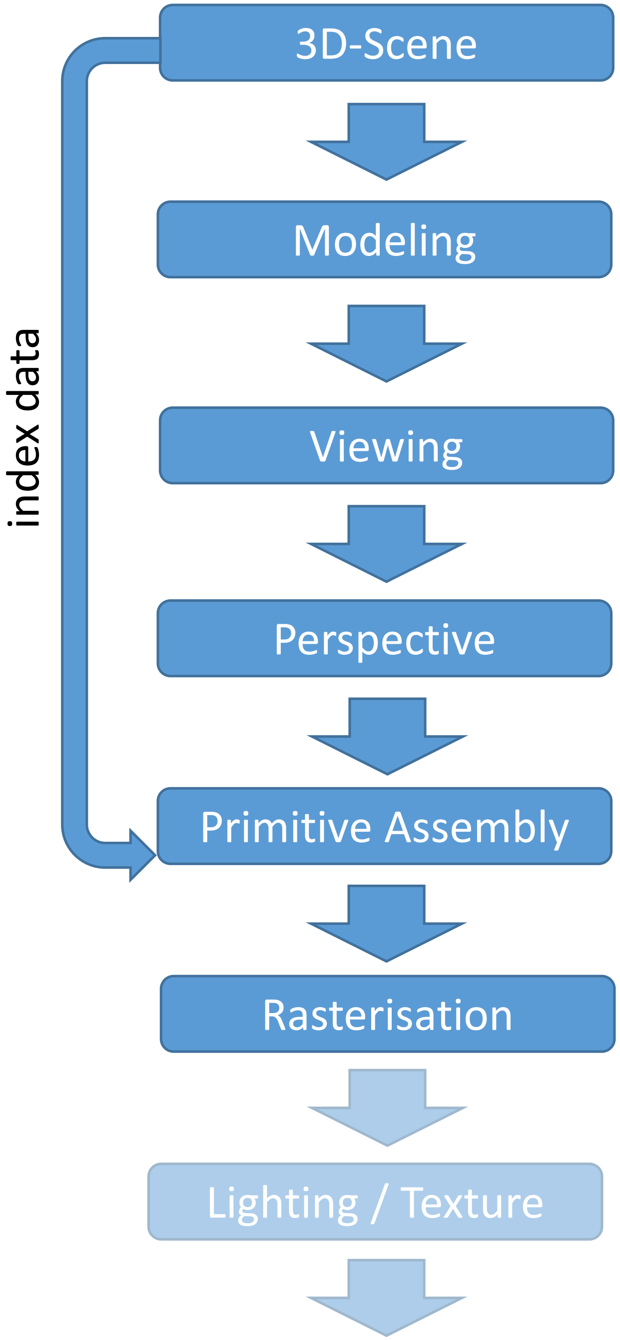
- there is an m:n relation between vertices and triangles
- transformation works on vertices
- rasterization works on triangles
- conversion of vertex stream to triangle stream is done by **primitive assembly**
- This is done either using an implicit topology (triangle strips, fans, …)
- or using an index buffer





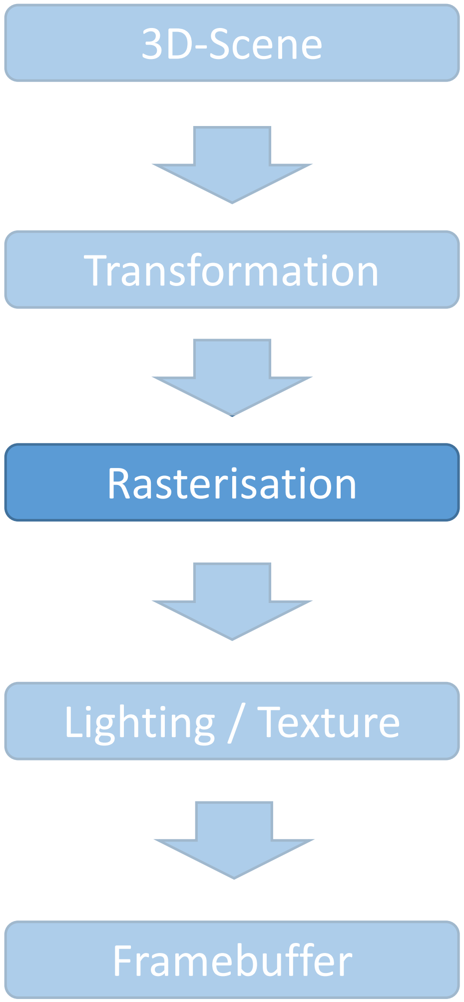
Rendering Pipeline

- so now we have:



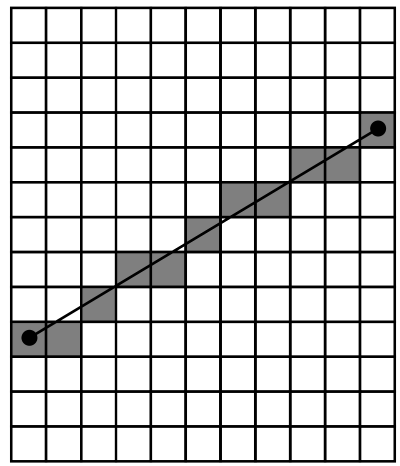
Rendering Pipeline

- Coarse Version

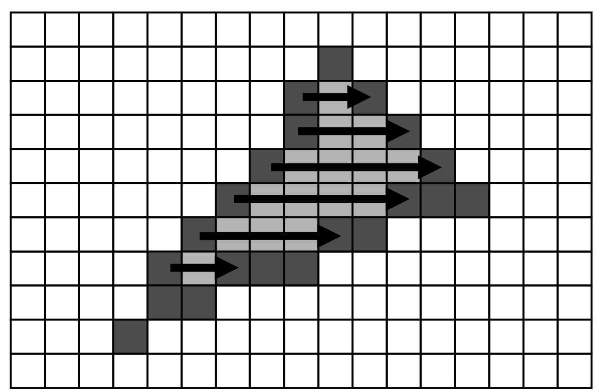


Rasterization

- Lines
  - Bresenham



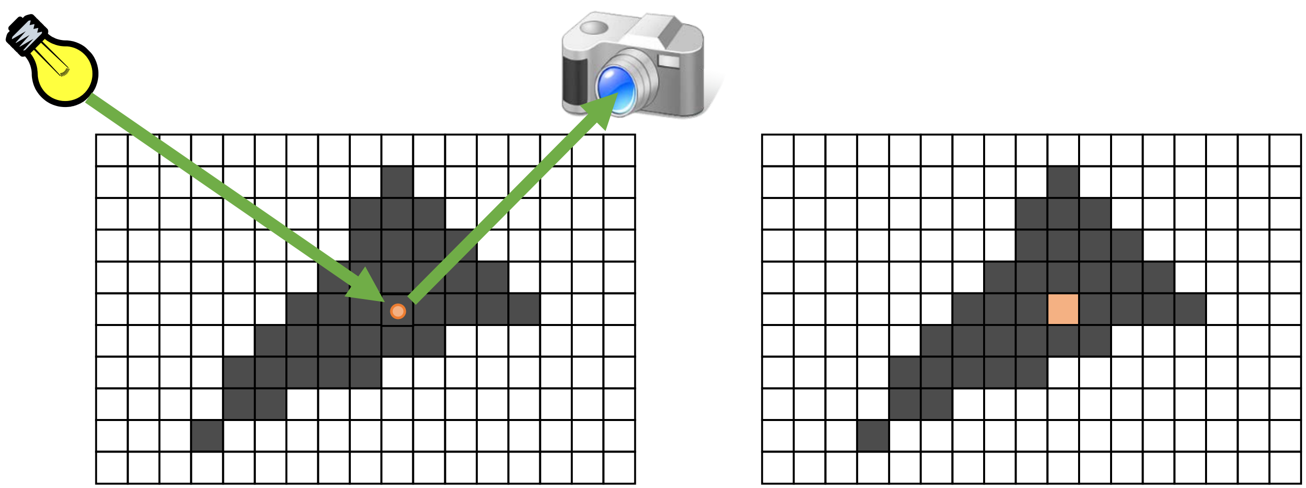
- Polygons
  - Scanline



- Where to get the colors of the set pixels from ?
  → **Lighting**

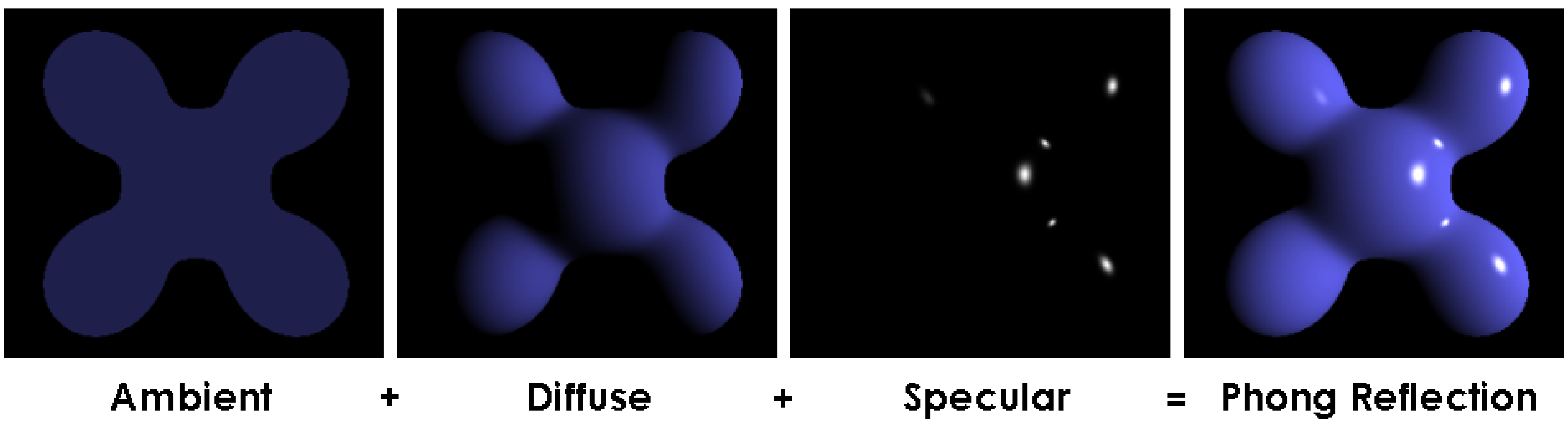
Lighting

- Given:
  - a number of light sources (point light, parallel light, spot light)
  - a surface point (position and surface normal)
  - and material parameters
  - and the current viewer position
- Compute:
  - color of surface point as seen from the current camera



Lighting

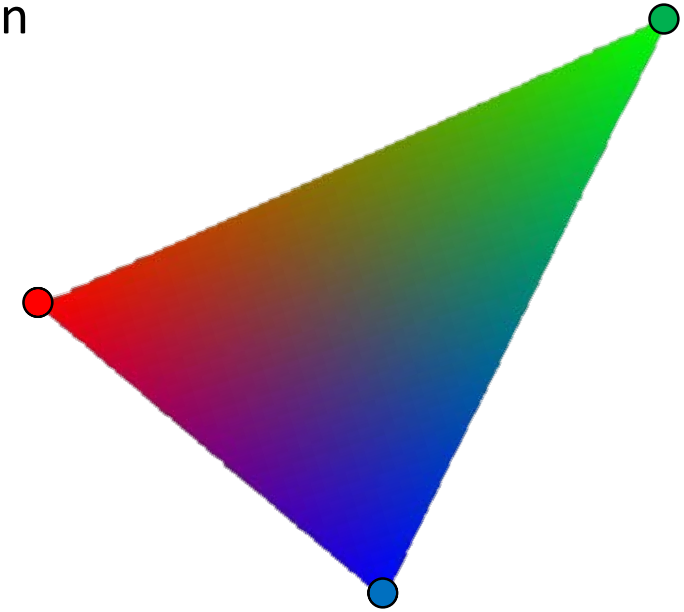
- **Phong Model**
  - ambient
  - diffuse
  - specular



Shading

- How to integrate lighting into rasterization ?

- All vertices contain a number of attributes
  - at least: vertex position
  - possibly: normal, color, texture coordinates, …
- all these are interpolated during rasterization
- linear interpolation in screen space
  != linear interpolation in object space
- → **perspective correct interpolation**



Shading

- How to integrate lighting into rasterization ?

- Two important approaches:

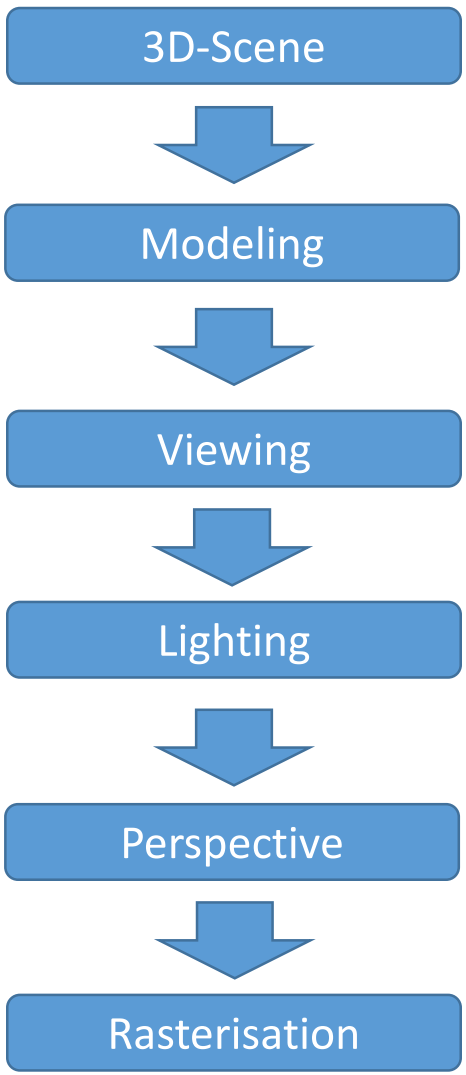
- **Gouraud Shading = Vertex Lighting**
  Do lighting computation at vertices, then interpolate color

- **Phong Shading = Pixel Lighting**
  Interpolate attributes needed for lighting, then compute lighting per pixel
  → better quality, usually more lighting computations → more expensive

- Let’s play – Gouraud Shading / Phong Shading / Phong Lighting

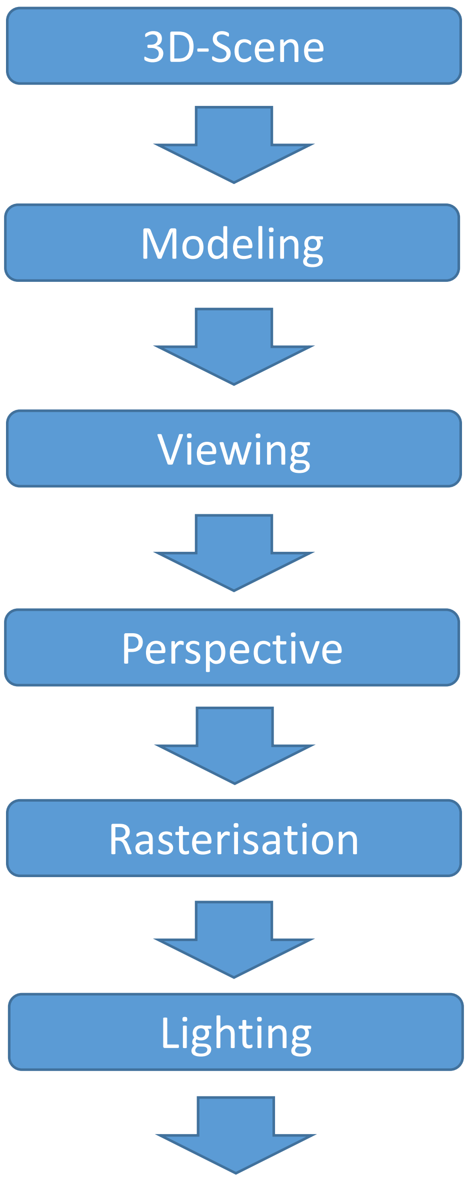
Shading

- Gouraud Shading:
  - Lighting happens per Vertex
  - usually in camera coordinates
  - can also happen in world coordinates (after modeling transformation)



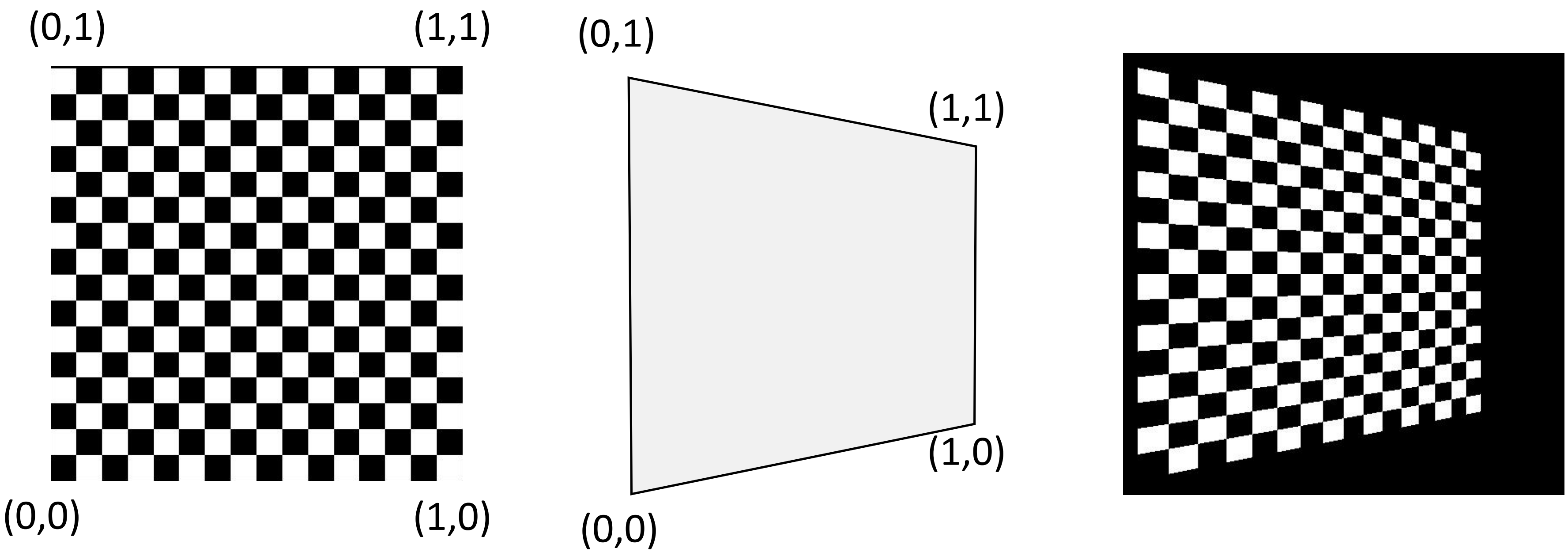
Shading

- Phong Shading
  - Lighting happens after rasterization



Texturing

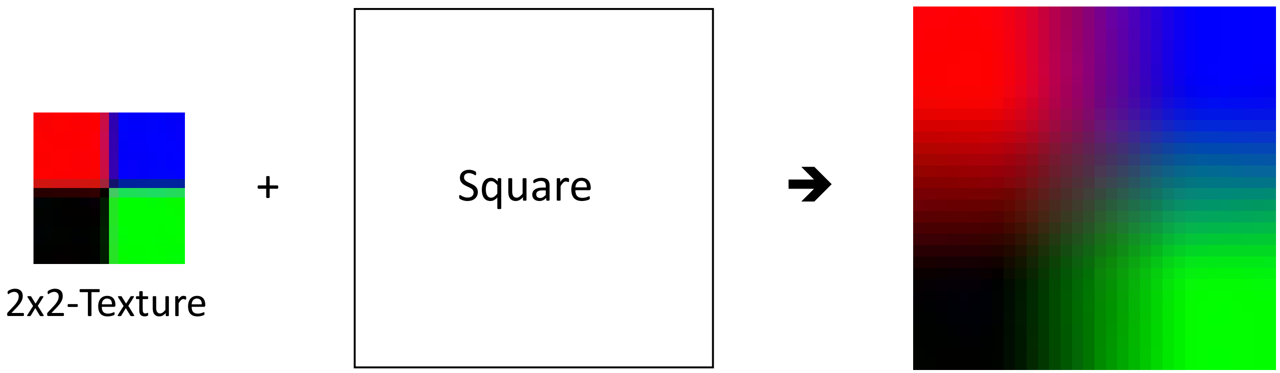
- Glue image onto objects



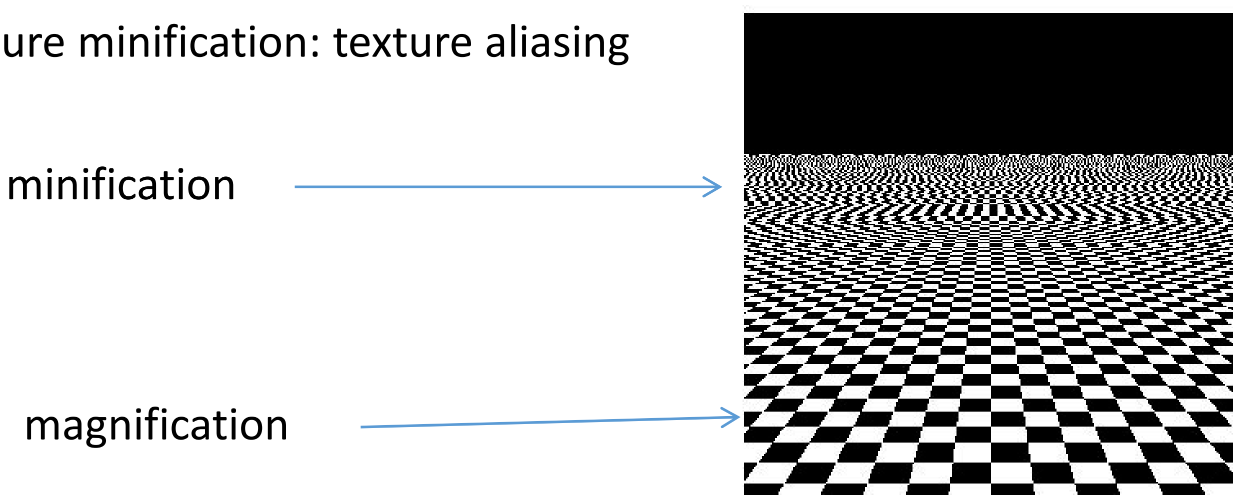
- Which part of the image to glue: texture coordinates
- texture coordinates are ordinary attribute that is interpolated during rasterization
  → **perspective correct interpolation!**

Texturing

- For each pixel, the texture value is fetched using the interpolated texture coordinates
  - texture magnification: nearest neighbor or bilinear interpolation (see below)

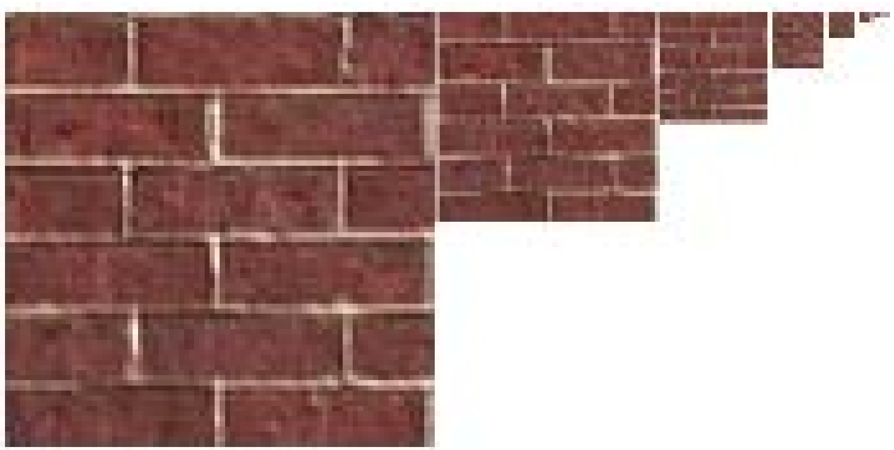


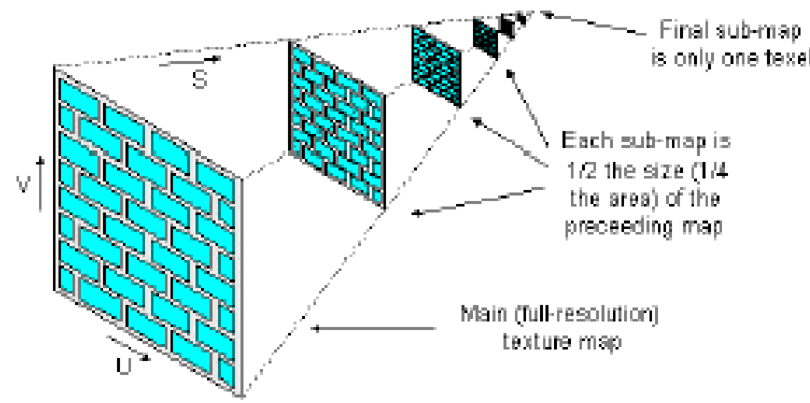
  - texture minification: texture aliasing



Texturing

- Solution / reduction of texture aliasing: MIPmaps





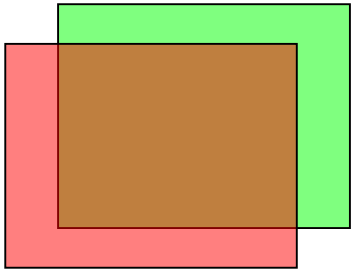
- Let’s play

Depth Buffer

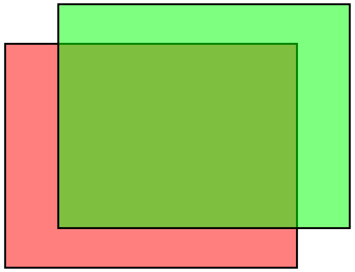
- In addition to attributes, also depth value is interpolated
- Depth buffer used to solve visibility:

```
setpixel(x,y,depth,color)
       if(zBuffer(x,y) > depth)
              screen(x,y) := color
              zBuffer(x,y) := depth
       endif
```

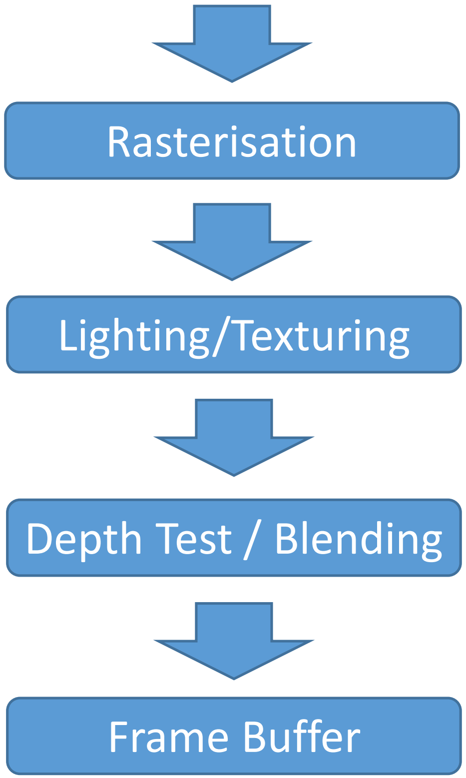
- Blending: combine new pixels with old ones
  → transparency effects



50% red over
50% green over
100% white



50% green over
50% red over
100% white



OpenGL - Shaders

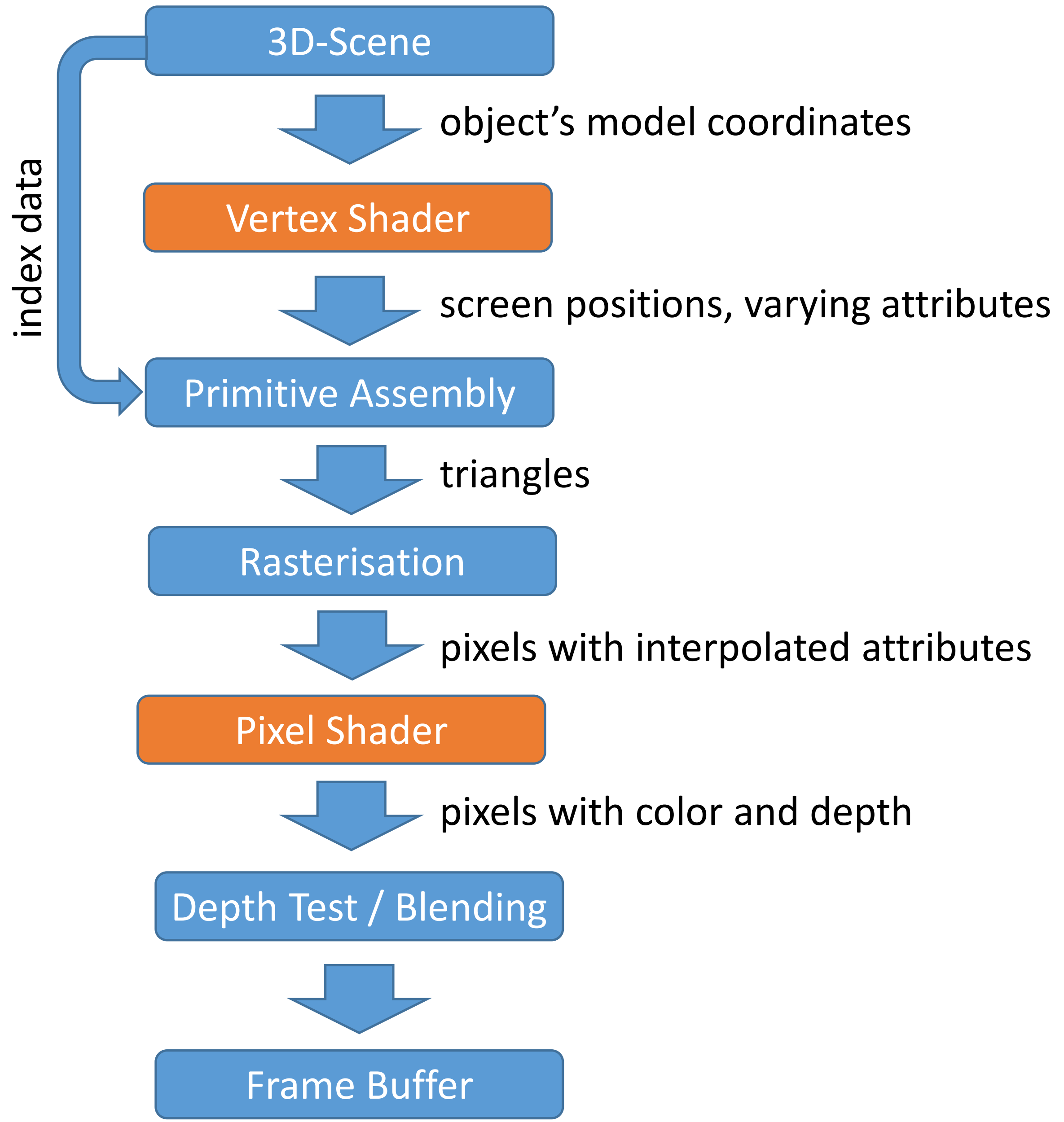
- In OpenGL, some parts of the pipeline are computed using **Shaders**
- Small, C-like programs executed on the GPU
- usually pairs of a vertex and a pixel shader

```
// vertex shader
attribute vec2 pos;
attribute vec3 col;
varying vec3 c;

void main(void) {
    gl_Position = vec4((pos+offset)*zoom, 0.0, 1.0);
    c = col;
}
```

```
// fragment shader
precision highp float;
varying vec3 c;

void main(void) {
    gl_FragColor = vec4(c,1);
}
```

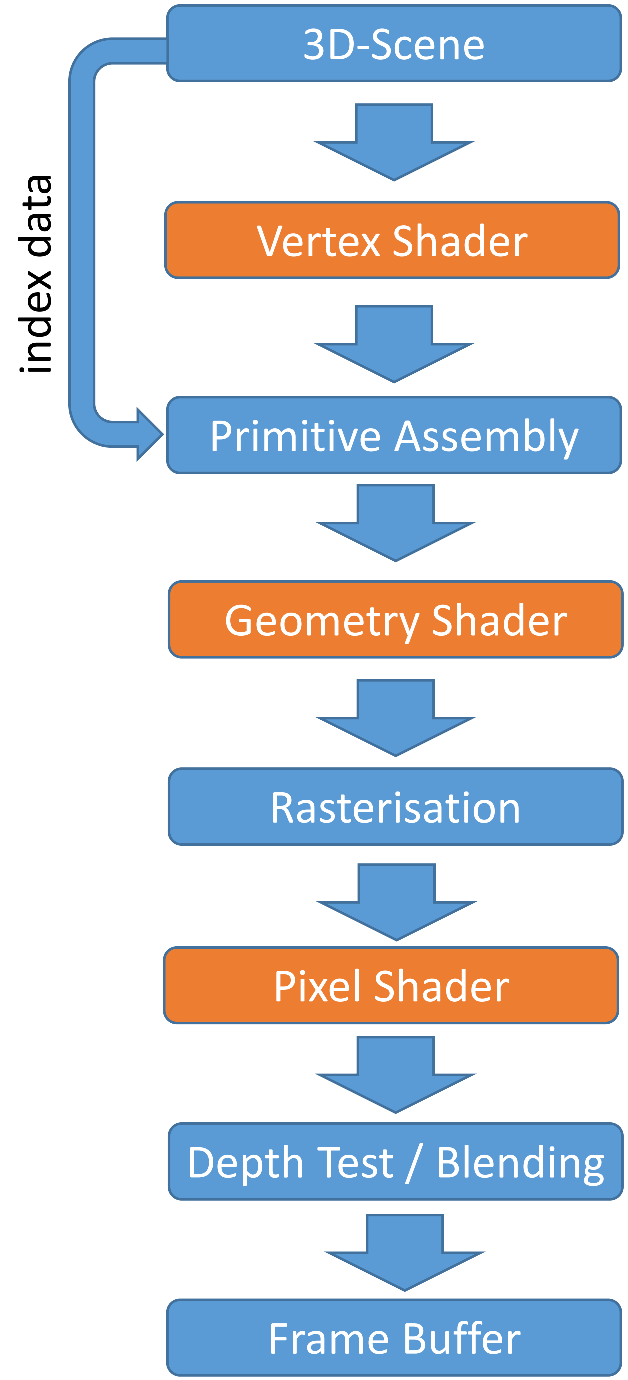
OpenGL - Shaders

- OpenGL pipeline



OpenGL - Shaders

- with geometry shader
  (handled in advanced exercises)



Merry Christmas !