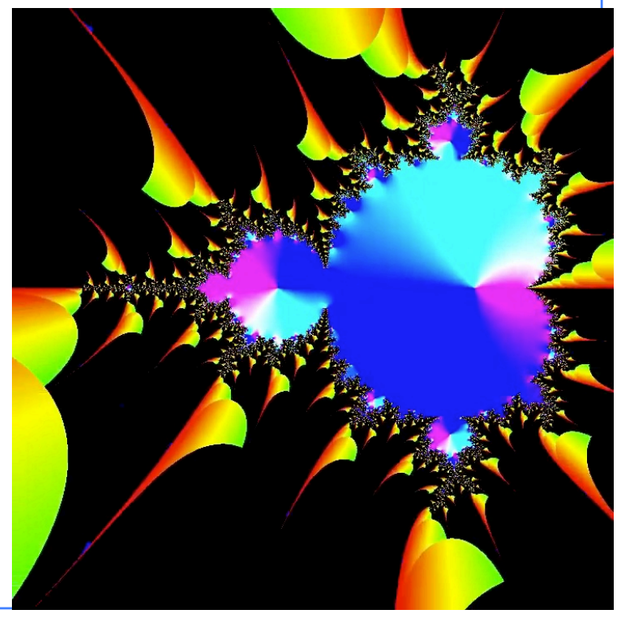
TSBK07/TSBK11/ETE378

Computer Graphics

Ingemar Ragnemalm, ISY





Lecture 4
3D graphics part 2

Today's topics:

Graphics pipeline and shaders
GLSL

Visible surface detection (intro):
Z-buffer
Back-face culling

Object representation (intro)

Lecture questions

1. What happens after the fragment shader stage?

2. Why is glDrawElements better than glDrawBuffers?

3. What is an attribute variable, and a varying?

4. How can OpenGL tell the facing of a polygon?

5. What operation is possible due to the projection preserving the Z value?

Lab material:

A bunch of small utility files:

GL_utilities
MicroGlut
LittleOBJLoader
LoadTGA
VectorUtils3/4

Note on the lab material:

Why are we not using big libraries instead of these small ones?

Answers:

Transparency!
Portability!
Ease of compilation!
Harder to cheat!

Dugga results:

Varied literally from 0 to 10!

Tree students nailed it!

TSBK11 students make 7 at best.

TSBK07: You need 4 or more on average to pass.

TSBK11: You need a bit more than 3 on average to pass.

Transformations in 2D and 3D with homogenous coordinates

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

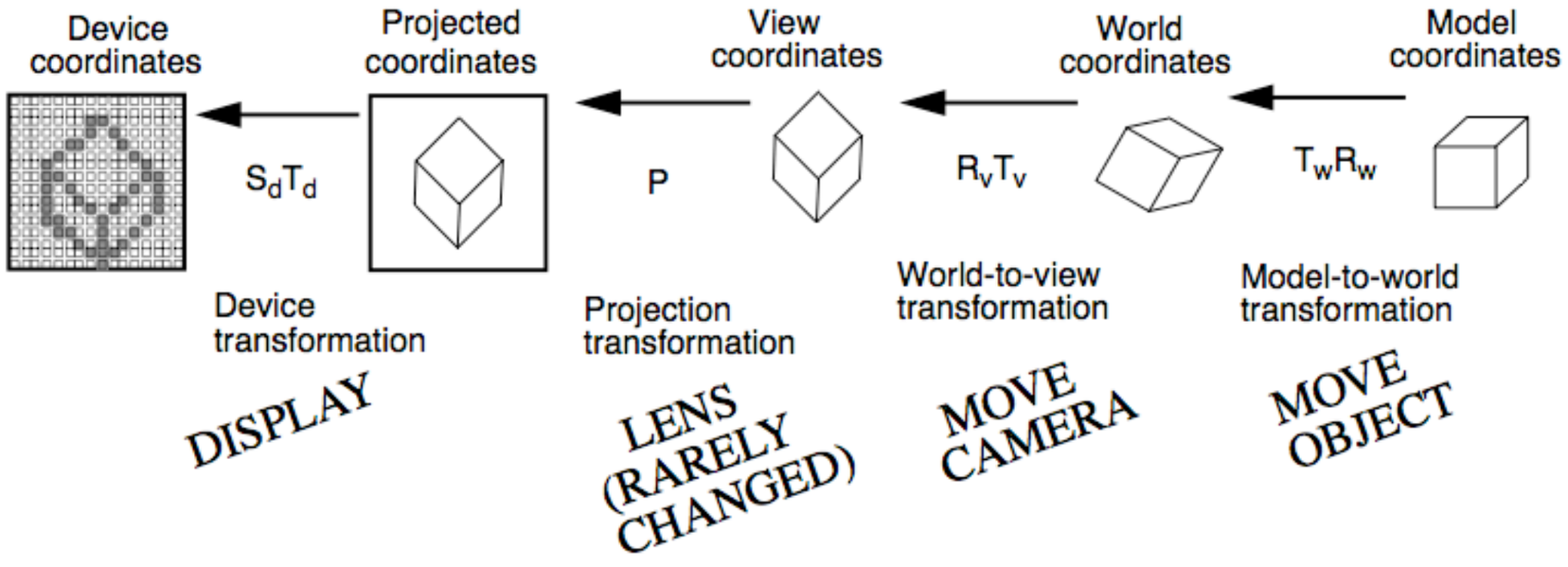
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Transformation pipeline

Model coordinates
World coordinates
View coordinates
Projected coordinates
Device coordinates



Projection by using the bottom row with homogenous coordinates

$$\begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

f = focal distance

Assuming $z_{prp} = 0$

z not interesting here

Straight projection - no normalization

Projection by using the bottom row with homogenous coordinates

$$\begin{bmatrix} 2n/(r-l) & 0 & A & 0 \\ 0 & 2n/(t-b) & B & 0 \\ 0 & 0 & C & D \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

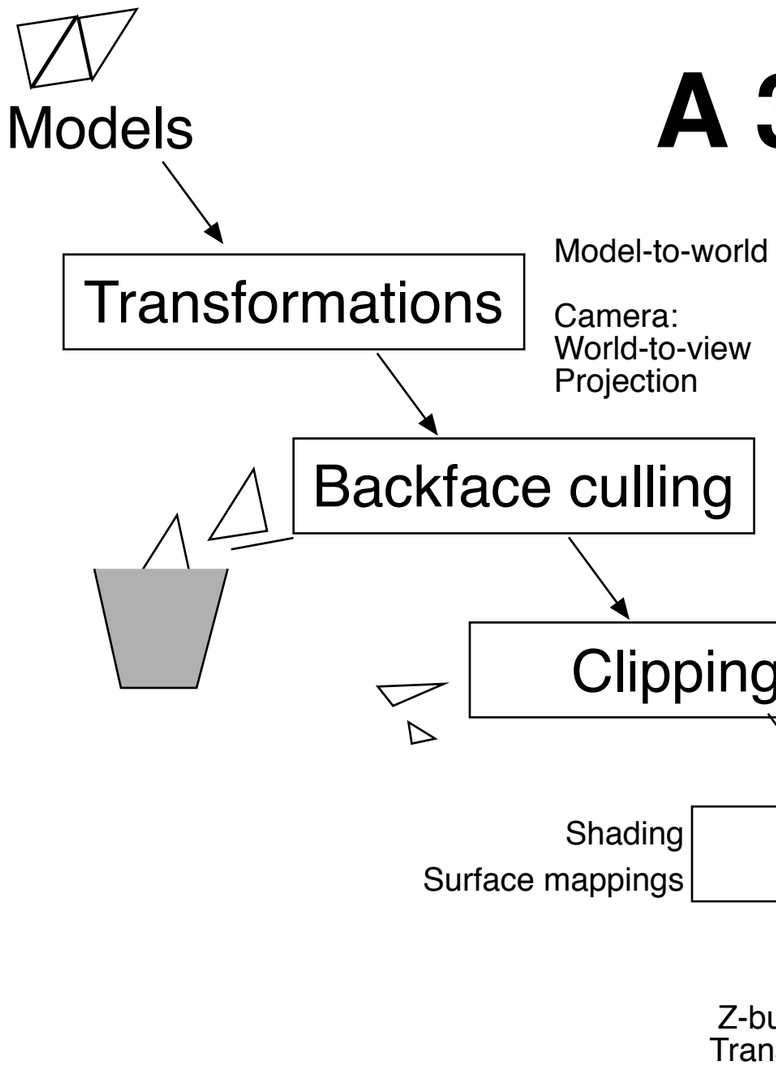
A, B usually zero

C = -(f+n)/(f-n)

D = -2f*n/(f-n)

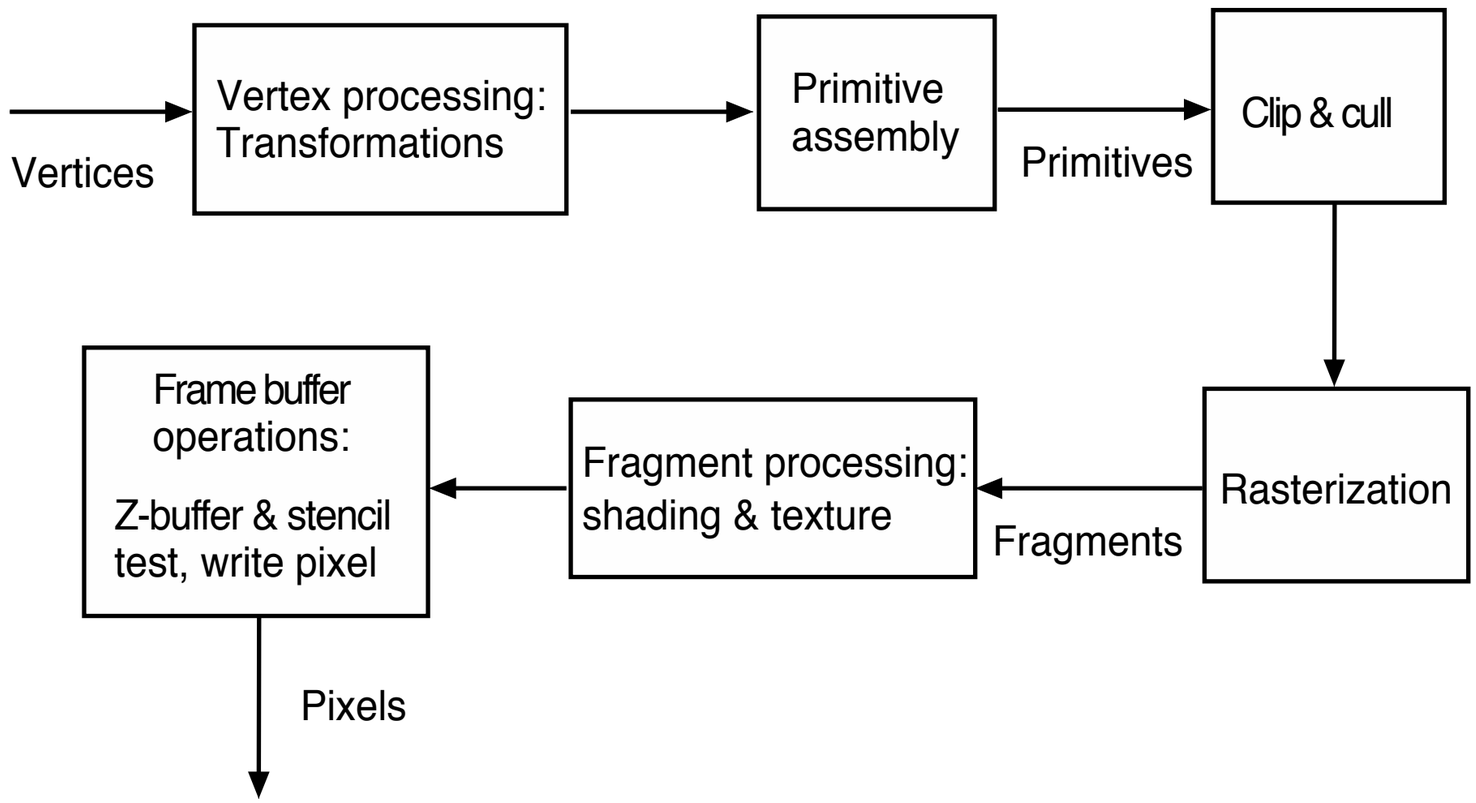
Normalized coordinates and viewing frustum

A 3D engine



(Not complete or general structure.)

The OpenGL pipeline



So what is a shader?

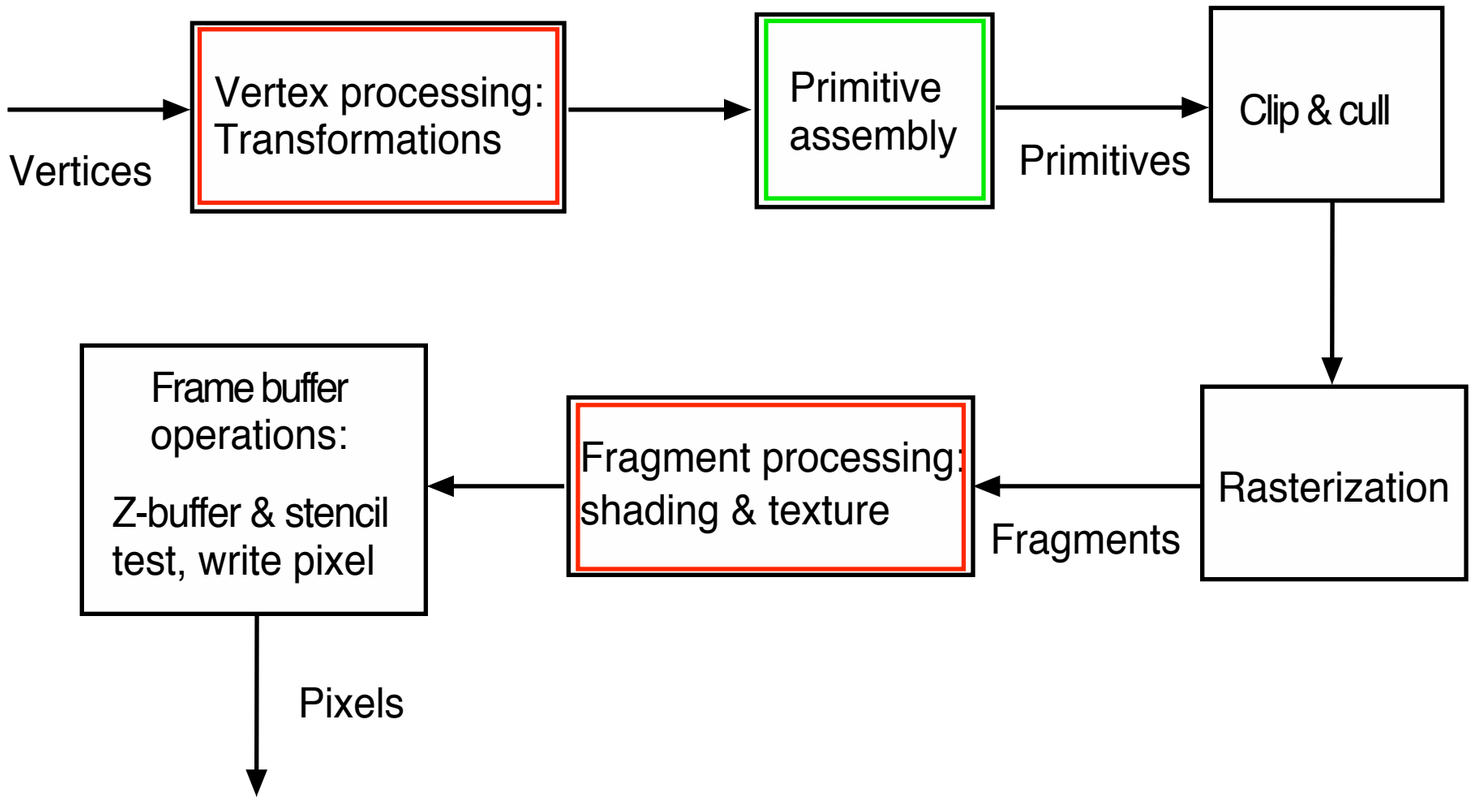
- Small program (computing kernels)

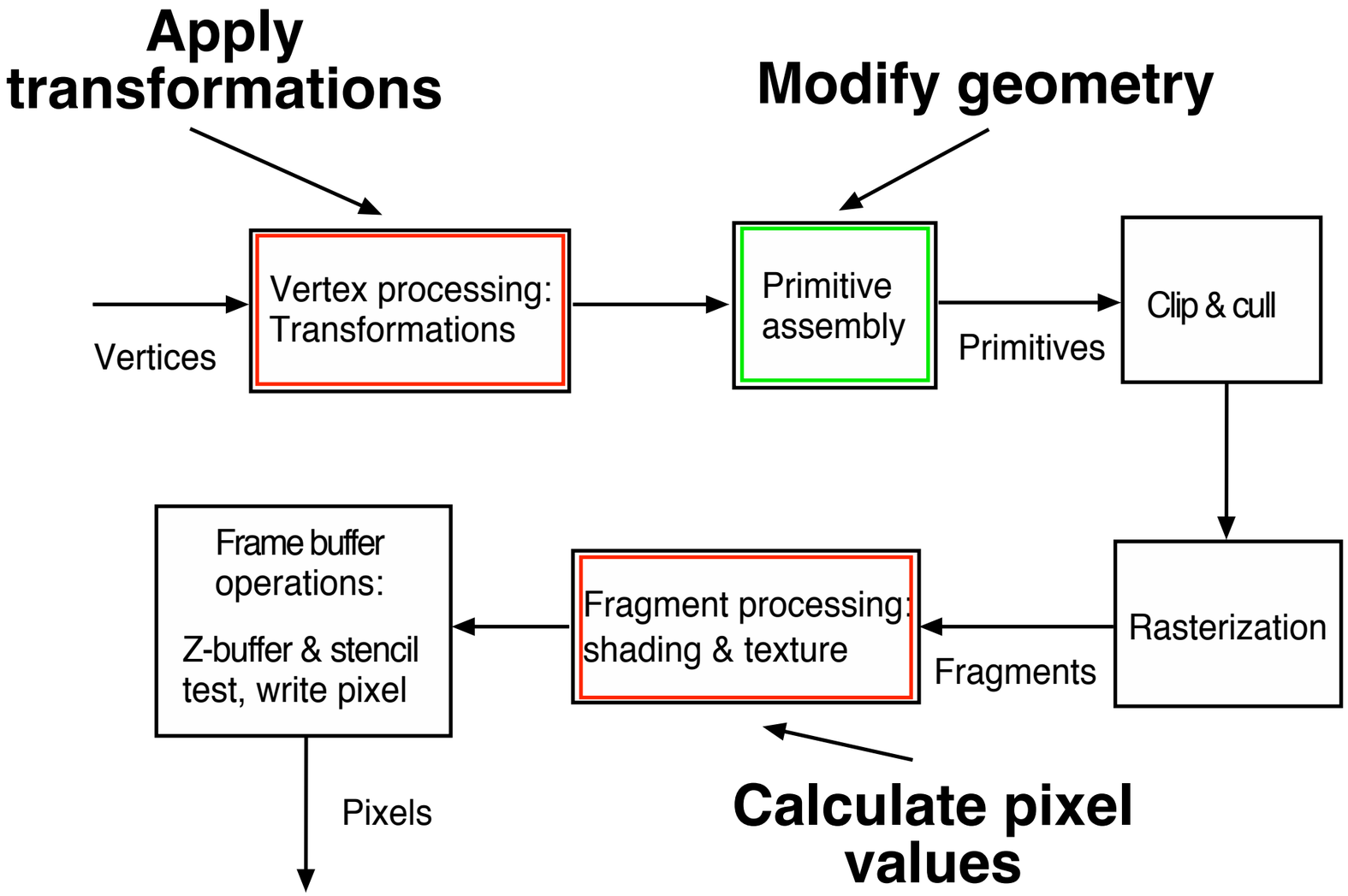
- Performs calculations for a specific stage in the graphics pipeline

- Runs natively on the GPU

- CPU orders rendering -> active shaders are invoked as part of the rendering

Stages that may or must have shaders




Apply transformations
Modify geometry
Vertex processing: Transformations
Vertices
Primitive assembly
Primitives
Clip & cull
Frame buffer operations:
Z-buffer & stencil test, write pixel
Fragment processing: shading & texture
Fragments
Rasterization
Pixels
Calculate pixel values

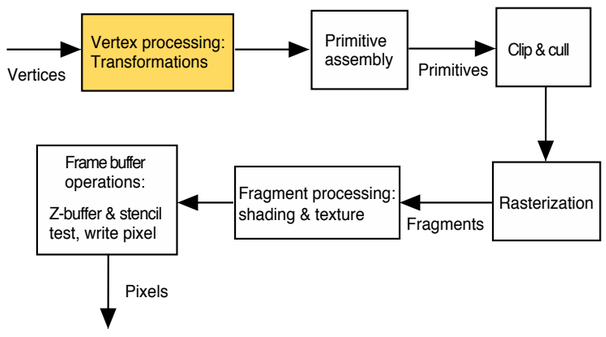
More on shaders

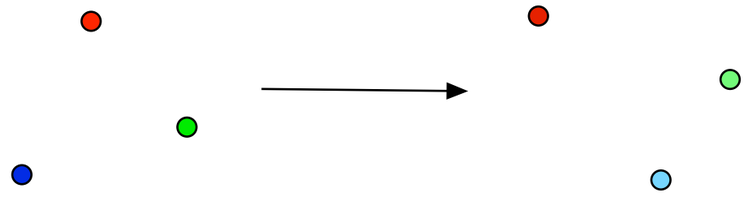
- Load from source and compile at runtime (e.g. at program launch or scene changes)
- Written in a language, e.g. GLSL (OpenGL Shader Language)
- Runs in parallel, many runs, for many data items at once
- Fast and flexible!

The vertex stage/vertex shader

The vertex stage handles the following tasks:

- Vertex transformation (model to projected)
- Transformation of normal vectors
- Generation/transformation of texture coordinates
- Lighting calculations (if not in fragment shader)
- Material parameters
- Send interpolated data to fragment shader





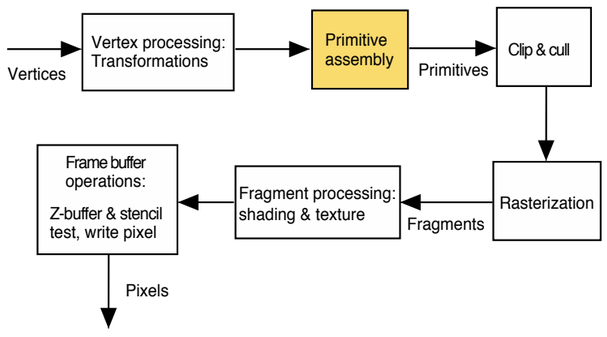
Primitive assembly/Geometry stage

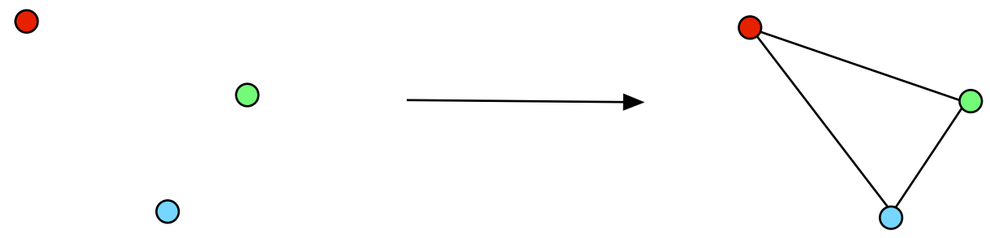
Assembly of primitives

“Primitive” not as in simple but as in geometrical primitives

Transformed coordinates are collected into structures for each triangle, line etc.

Geometry & tesselation shaders can alter geometry. (Optional.)

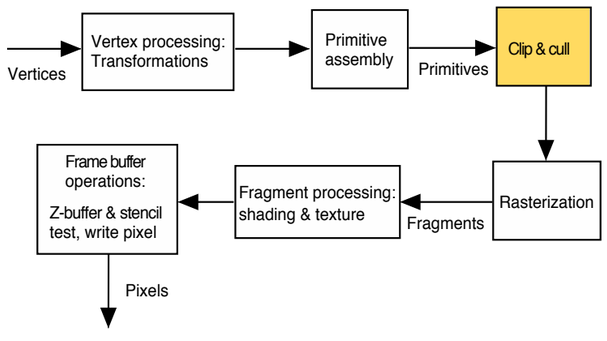


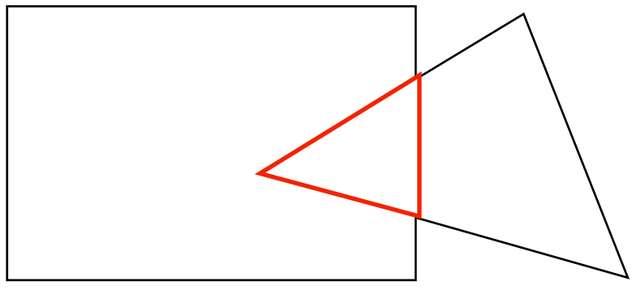


Clipping and culling

Primitives are clipped to screen borders.
Backface culling is performed.

Note that texture coordinates also needs clipping (as well as any other data that is interpolated between vertices).

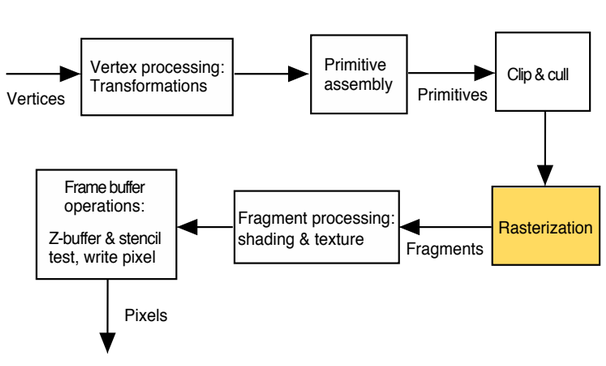


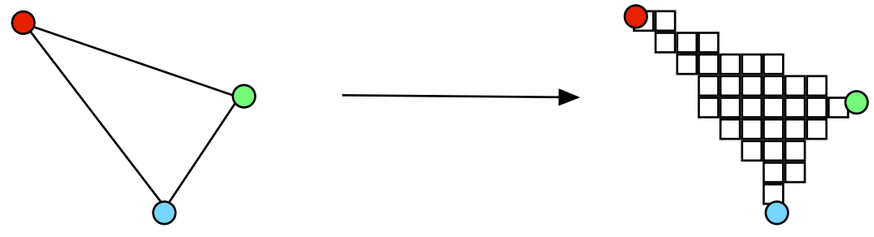


Raster conversion

Polygon rendering, convert polygons to pixel coordinates

Creates “fragments”. Note that they do not have any colors yet!

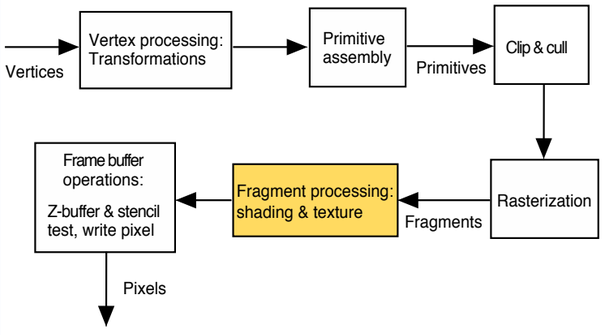


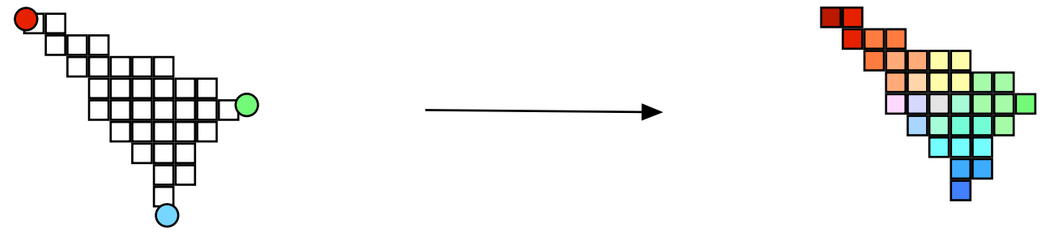


The fragment stage/fragment shader

From pixel coordinates and interpolated data for color, texture etc, calculate a color for the fragment.

- Shading, light calculations
- Texturing
- Fog
- Color calculations

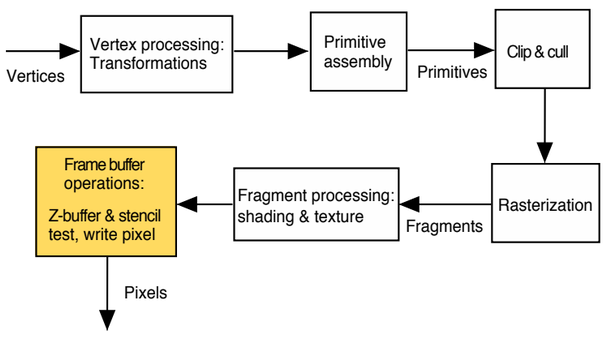




Frame buffer operations

Final operations before the fragment is written to a frame buffer pixel

- Stencil test
- Z-buffer test
- The blend function (glBlendFunc mm)



GLSL basics

A tour of the language (with some examples)

- Character set
- Preprocessor directives
- Comments
- Identifiers
- Types
- Modifiers
- Constructors
- Operators
- Built-in functions and variables
- Activating shaders from OpenGL
- Communication with OpenGL

Character set

Alphanumerical characters: a-z, A-Z, _, 0-9

. + - / * % < > [ ] { } ^| & ~ = ! : ; ?

for preprocessor directives (!)

space, tab, FF, CR, FL

Note! Tolerates both CR, LF och CRLF! ☺

Case sensitive

BUT

Characters and strings do not exist! ‘a’, “Hej” mm

The preprocessor

#define #undef #if etc

VERSION is useful for handling version differences. It will hardly be possible to avoid in the long run.

150 = OpenGL 3.2
330 = OpenGL 3.3

#include does not exist! ☺

Comments

/* This is a comment
that spans more than one line */

// but personally I prefer the one-line version

Just like we are used to! ☺

So litter your code with comments!

Identifiers

Just like C: alphanumerical characters, first non-digit

BUT

Reserved identifiers, predefined variables, have the prefix gl_! (I.e. gl_Position.)

It is not allowed to declare your own variables with the gl_ prefix!

Types

There are some well-known scalar types:

void: return value for procedures
bool: Boolean variable, that is a flag
int: integer value
float: floating-point value
double: double precision floating-point value

More types

Vector and matrix types:

vec2, vec3, vec4: Floating-point vectors with 2, 3 or 4 components

bvec2, bvec3, bvec4: Boolean vectors

ivec2, ivec3, ivec4: Integer vectors

mat2, mat3, mat4: Floating-point matrices of size 2x2, 3x3, 4x4

Most common: vec2, vec3, vec4, mat3, mat4!

Swizzling

Indexing vectors:

v.xyzw

v.rgba

v.stpq

Change order as desired: xxy, gbr...

Don't mix! No xgp!

Important!

Modifiers

Variable usage is declared with modifiers:

const

in (attribute)

uniform

in/out (varying)

out (output)

If none of these are used, the variable is “local” in its scope and can be read and written as you please.

const

constant, assigned at compile time, can not be changed

attribute and uniform

an attribute (declared "in" in the vertex shader) is argument from OpenGL, per-vertex-data

uniform is argument from OpenGL, per primitive (in practice, per drawing call). Can not be changed within a drawing call.

varying (”in”, ”out”)

data that should be interpolated between vertices

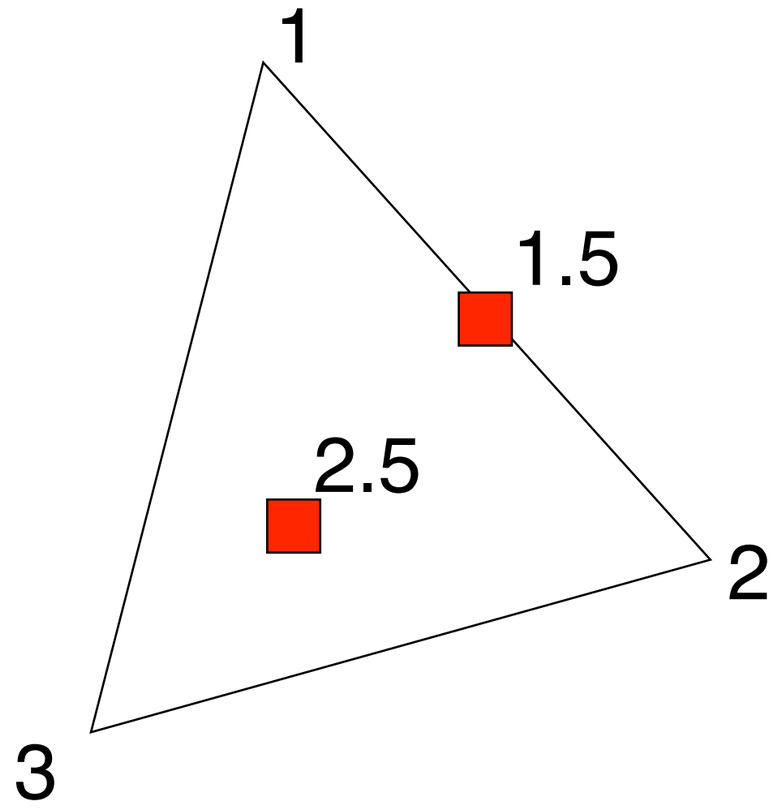
Written in vertex shader

Read (only) by fragment shaders

Declared "out" in vertex, "in" in fragment shader. In the fragment shader, they are read only.

Examples: texture coordinates, normal vectors for Phong shading, vertex color, light value for Gouraud shading

varying (”in”, ”out”)



varying means varies between vertices, interpolated!

”varying” or ”in/out”?

”varying” was a keyword in older GLSL, replaced by ”in/out” in newer (somewhat more intuitive)

I will use "varying" as a term denoting this kind of interpolated variables.

Compilation and execution

Done in two steps:

1) Initialization, compilation

- Create a “program object”
- Create a “shader object” and pass source code to it
- Compile the shader programs

2) Activation

- Activate the program object for rendering

The entire initialization in code

```
PROG = glCreateProgram();

VERT = glCreateShader(GL_VERTEX_SHADER);
  text = readTextFile("shader.vert");
glShaderSource(VERT, 1, text, NULL);
      glCompileShader(VERT);
```

Same for fragment shader

```
glAttachShader(PROG, VERT);
glAttachShader(PROG, FRAG);

    glLinkProgram(PROG);
```

Activate the program for rendering

With an installed and compiled shader program:

```
GLuint PROG;
```

activate:

```
glUseProgram(PROG);
```

Important!

With more than one shader, make sure that the right one is active!
(Simplified if you use the calls in VectorUtils.)

Shader input/output

From host to vertex shader

From vertex shader to fragment shader

From fragment shader to frame buffer

From host to vertex shader

Two variants:

- Uniform
- Attribute

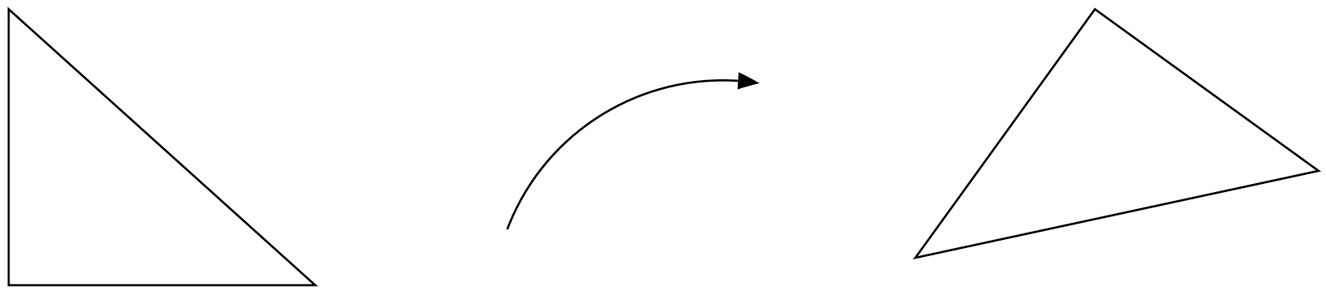
Uniform

Same for all in a primitive

Typical usage:

- Transformation matrices
- Texture units
- Time variable

Uniform



Same rotation
for all vertices

Applied for entire primitive (e.g. model)

Declare as "uniform" in the shaders

Attributes

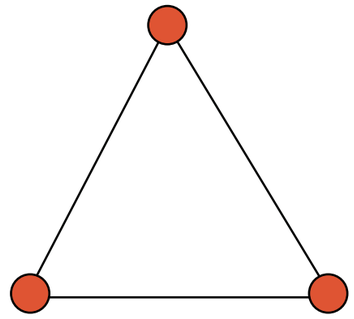
Different for every vertex

Passed as arrays, as VBOs

Typical usage:

- Vertices
- Normal vectors
- Texture coordinates

Attributes



Every vertex is an attribute.

Delivered in VBOs, vertex array buffers

Declare as "in" in the vertex shader

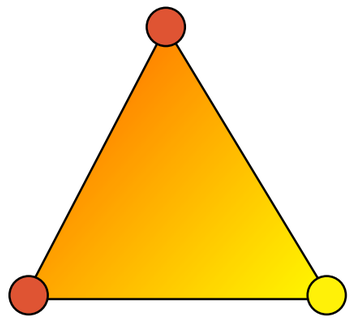
Varying

"out" from vertex shader

"in" into fragment shader

Interpolated between vertices!

Varying



Values sent from vertex shaders are interpolated and sent to fragments

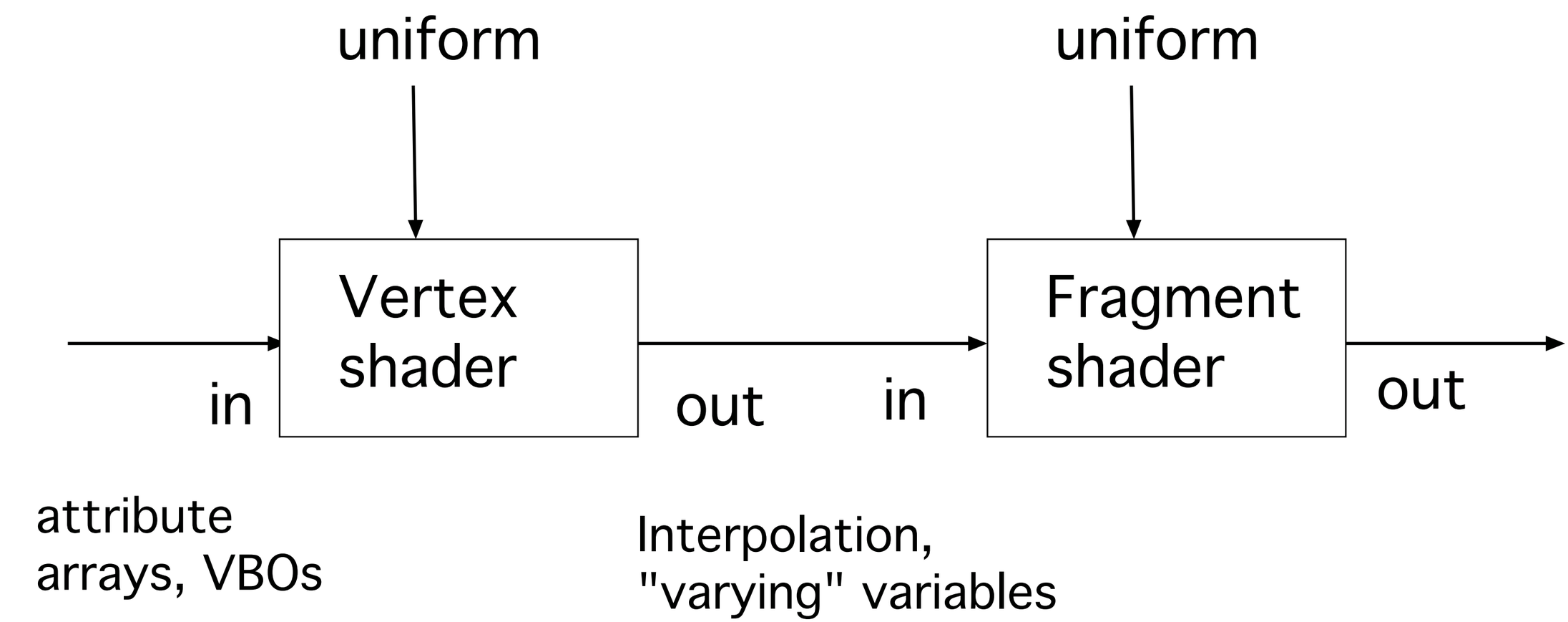
Simple usage: Set color in each vertex to get a gradient over the polygon

Output from fragment shader

Declared "out"

Typically a single output, to the frame buffer

vec3 or vec4


uniform
uniform
Vertex shader
Fragment shader
in
out
in
out
attribute
arrays, VBOs
Interpolation,
"varying" variables

Pass-through vertex shader

```
#version 150

in  vec3 in_Position;

void main(void)
{
    gl_Position = vec4(in_Position, 1.0);
}
```

Attribute!

Pass-through fragment shader

```
#version 150

out vec4 out_Color;

void main(void)
{
    out_Color = vec4(1.0, 1.0, 1.0 ,1.0);
}
```

Final output!

More typical vertex shader

```
#version 150

in  vec3 in_Position;
in  vec3 in_Normal;
in  vec2 in_TexCoord;
uniform mat4 mvMatrix;
uniform mat4 projMatrix;
out vec3 exNormal;
out vec2 exTexCoord;

void main(void)
{
    gl_Position = projMatrix * mvMatrix * vec4(in_Position, 1.0);
    exNormal = mat3(mvMatrix) * in_Normal;
    exTexCoord = in_TexCoord;
}
```

Attributes!

Uniforms!

Varyings!

More typical fragment shader

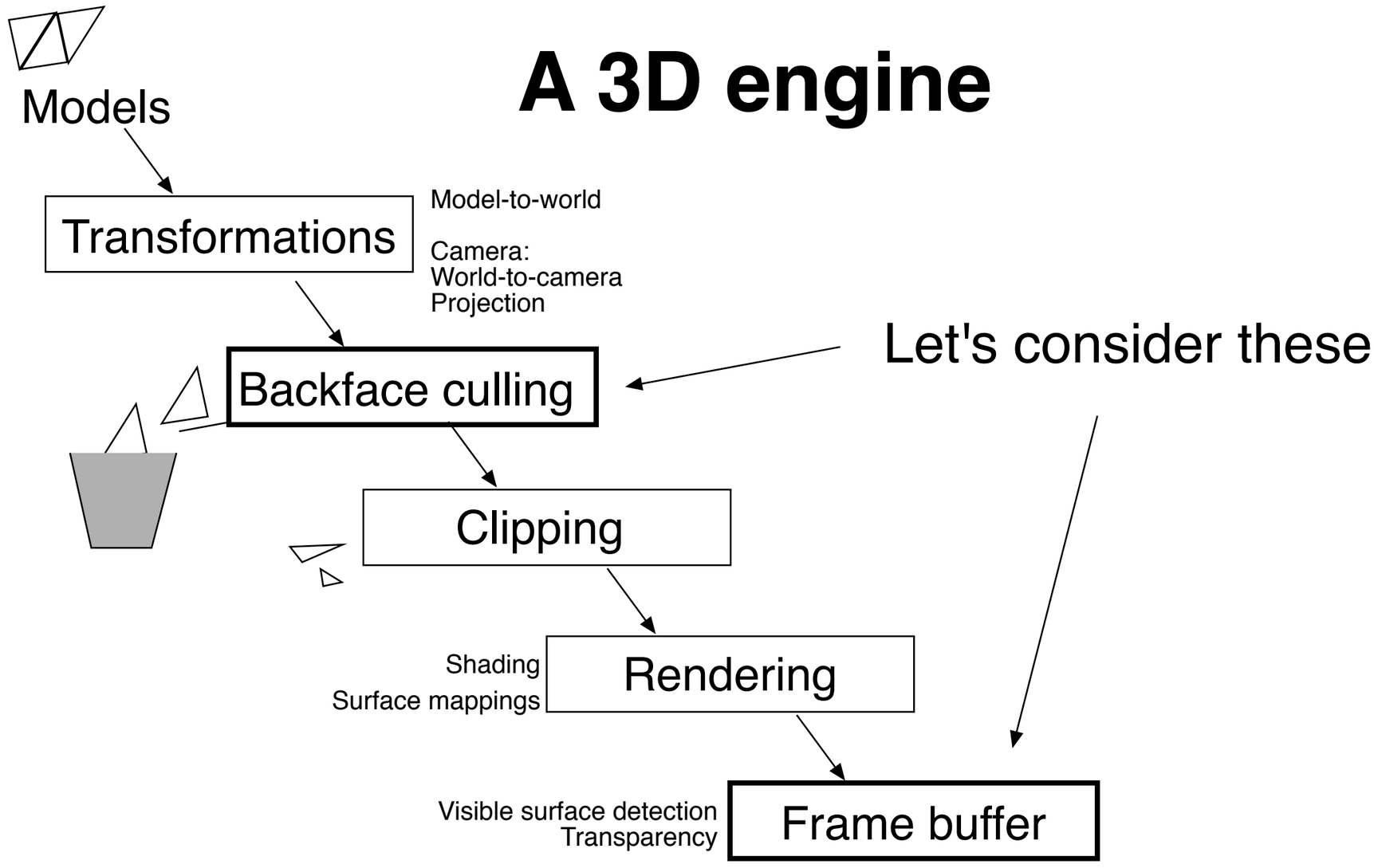
```
#version 150

in vec2 exTexCoord;
in vec3 exNormal;
out vec4 out_Color;

void main(void)
{
// Texture lookups, light calculations...
   out_Color = ...;
}
```

Varyings!

Questions on shaders?

Very important concept,
not worth leaving unclear!

A 3D engine


Models
Transformations
Model-to-world
Camera:
World-to-camera
Projection
Backface culling
Clipping
Shading
Surface mappings
Rendering
Visible surface detection
Transparency
Frame buffer
Let's consider these

Visible surface detection

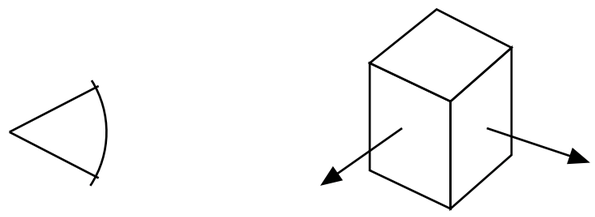
Backface culling
Z-buffer
Painter’s algorithm
BSP trees
Scan-line method

Ray-casting
A-buffer
Area subdivision
Octrees
Portals

Backface culling

Object space method

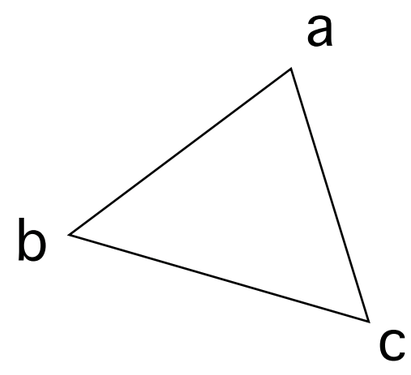
Removes all polygons that are "looking away" from the camera.



Removes ≈50% of all polygons that would otherwise be in view!

Backface culling

Based on vertex order of the *projected* polygon.



Inspect sign of z component of ab x ac!

Back-face culling can be configured to cull clockwise or counter-clockwise. Default is counter clockwise!

Z-buffer

Depth-buffer method

Image-space method

“Z” since we usually look along Z axis.

An extra “depth image” buffer is used, the Z buffer.

The Z buffer holds a Z value for every pixel in the image. The Z value corresponds to the nearest object written so far, that has touched that pixel.

Z-buffer algorithm

Initialize Z-buffer to infinite distance and the image buffer to background.

```
for each polygon
   for each pixel (x,y) in the polygon
      calculate z value
      if z closer than the current z-buffer value Z(x,y)
         write pixel to image (x,y)
         write z to z-buffer Z(x,y)
```

The need for Z

Z values must survive the projection - as specified in previous lecture

Z values are calculated for each vertex and interpolated over the surface

Culling and Z-buffer in OpenGL

```
glEnable(GL_CULL_FACE);

glCullFace(GL_BACK);
glCullFace(GL_FRONT);
```

Rarely used:

```
glFrontFace(GL_CW);
glFrontFace(GL_CCW);
```

Initialize context with depth buffer, e.g. GLUT_DEPTH

```
glEnable(GL_DEPTH_TEST);

glClear(GL_DEPTH_BUFFER_BIT);
```

Performance

Back-face culling removes polygons before fragment shader. Significant speedup!

Z-buffer test done after fragment shader. Limited performance improvement if any (saves some writes to the frame buffer, costs reads from the Z buffer.

When will culling + Z-buffering not be enough?

- Can not handle transparency

- High-level methods are needed to reduce the amount of data, to avoid passing unseen surfaces to the OpenGL pipeline

3D object representation

Most common for real-time:
Polygon surfaces

- Supported by graphics processors
- Easy to handle
- Not very space efficient
- Often triangle-based

How do we store that?

First try: A simple format

Vertex = (x, y, z)

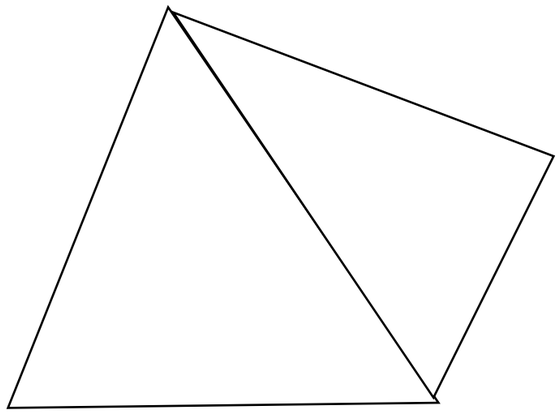
Triangle = array of Vertex

3DObject = array of Triangle

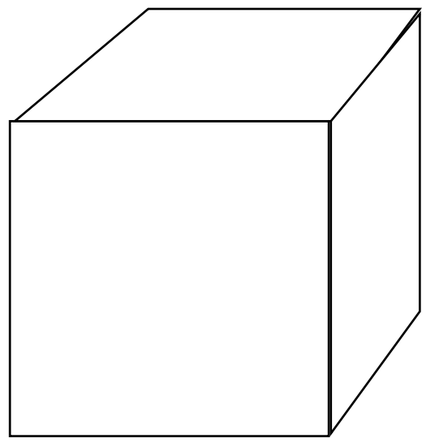
Example

Triangle[1]: (10, 10, 10), (10, 20, 10) (10, 20, 20)
Triangle[2]: (10, 10, 10), (10, 20, 10), (10, 10, 20)
Triangle[3]: (. . .), (. . .), (. . .)
. . .

…but why is this not quite good?



Tetraeder:
4 triangles
4 corners
Store as 12 vertices!



Cube:
6 squares or 12 triangles
8 corners
Store as 24 or 36 vertices!

Several times too many vertices to process!

A better format

Vertex = (x, y, z)

Vertex table = array of Vertex

Triangle = array of integers

Triangle table = array of Triangles

3DObject = Vertex table + Triangle table

Example

Vertex table:
(10, 10, 10)
(10, 20, 10)
(10, 20, 20)
(10, 10, 20)
. . .

All vertices in the object in one list

Triangle table:
(1, 2, 3)
(1, 2, 4)
. . .

Indexes to the vertex table

Modelling in OpenGL

```
GLfloat vertices[8*3] = { -1,-1,-1,  1,-1,-1, 1,1,-1, -1,1,-1,
                          -1,-1,1, 1,-1,1, 1,1,1, -1,1,1};

GLubyte cubeIndices[36] = {0,3,2, 0,2,1,   2,3,7, 2,7,6,
                           0,4,7, 0,7,3,   1,2,6, 1,6,5,
                           4,5,6, 4,6,7,   0,1,5, 0,5,4};
```

glDrawElements can draw the entire shape with one call (almost)!

Models on disk

Wavefront .obj format. Simple, text-based mesh format. Example: A cube:

```
# Exported from Wings 3D 0.98.26b
mtllib cube.mtl
o cube1
#8 vertices, 6 faces
v -1.00000000 -1.00000000 1.00000000
v -1.00000000 1.00000000 1.00000000
v 1.00000000 1.00000000 1.00000000
v 1.00000000 -1.00000000 1.00000000
v -1.00000000 -1.00000000 -1.00000000
v -1.00000000 1.00000000 -1.00000000
v 1.00000000 1.00000000 -1.00000000
v 1.00000000 -1.00000000 -1.00000000
```

Vertex list

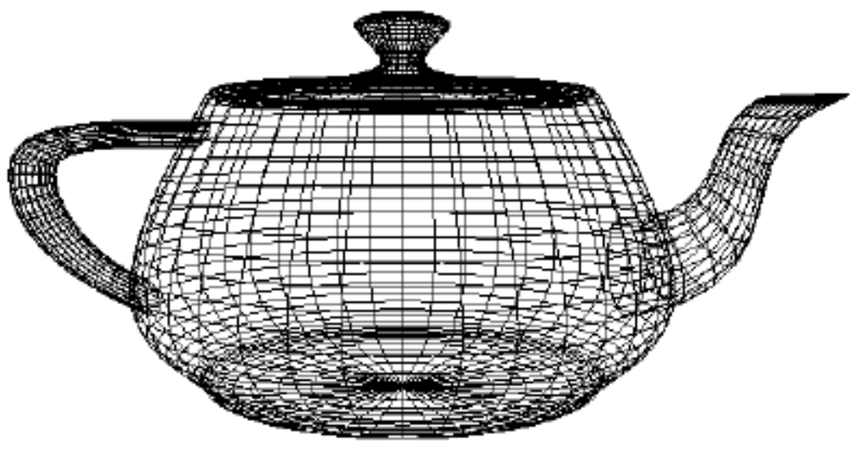
```
vn -0.57735027 -0.57735027 0.57735027
vn -0.57735027 0.57735027 0.57735027
vn 0.57735027 0.57735027 0.57735027
vn 0.57735027 -0.57735027 0.57735027
vn -0.57735027 -0.57735027 -0.57735027
vn -0.57735027 0.57735027 -0.57735027
vn 0.57735027 0.57735027 -0.57735027
vn 0.57735027 -0.57735027 -0.57735027
```

Normal vectors list

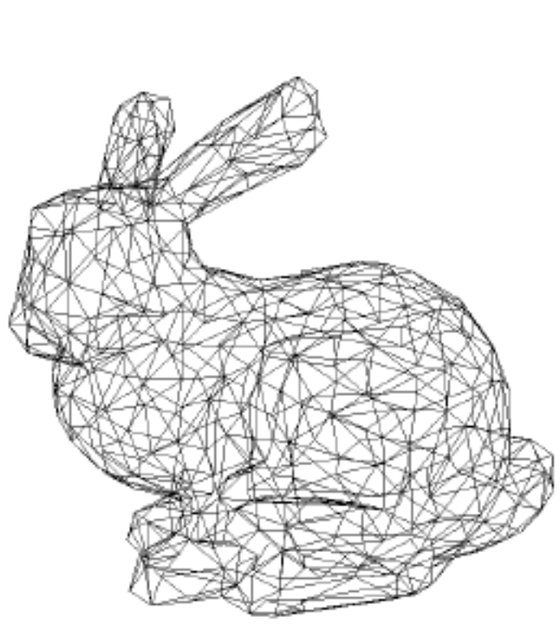
```
g cube1_default
usemtl default
f 3//3 2//2 1//1 4//4
f 5//5 1//1 2//2 6//6
f 6//6 2//2 3//3 7//7
f 7//7 3//3 4//4 8//8
f 8//8 4//4 1//1 5//5
f 8//8 5//5 6//6 7//7
```

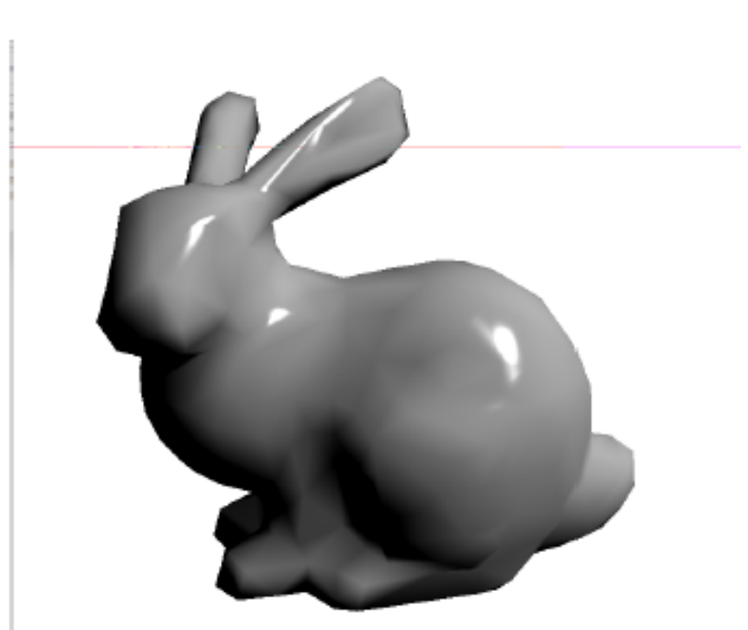
Polygon list

Common standard model: Utah Teapot

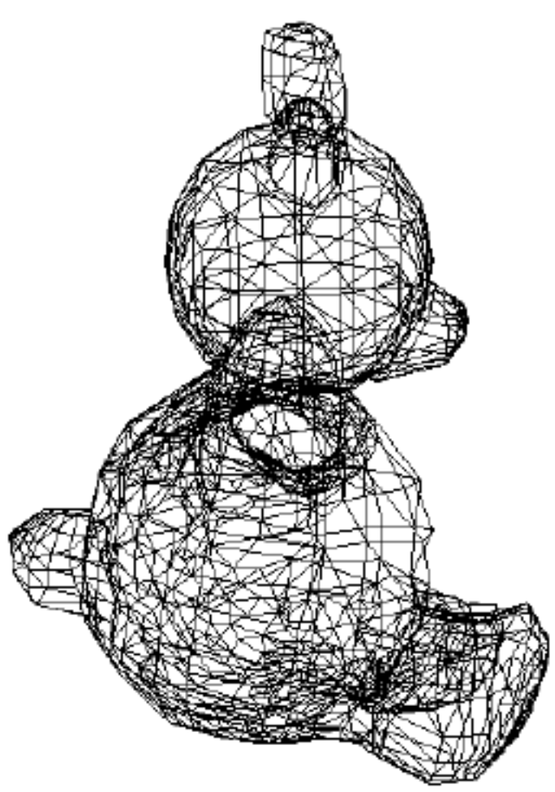


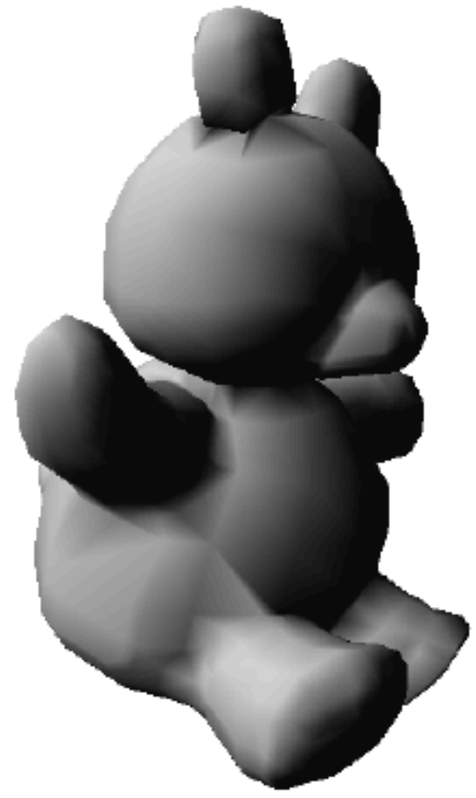
Common standard model: Stanford Bunny



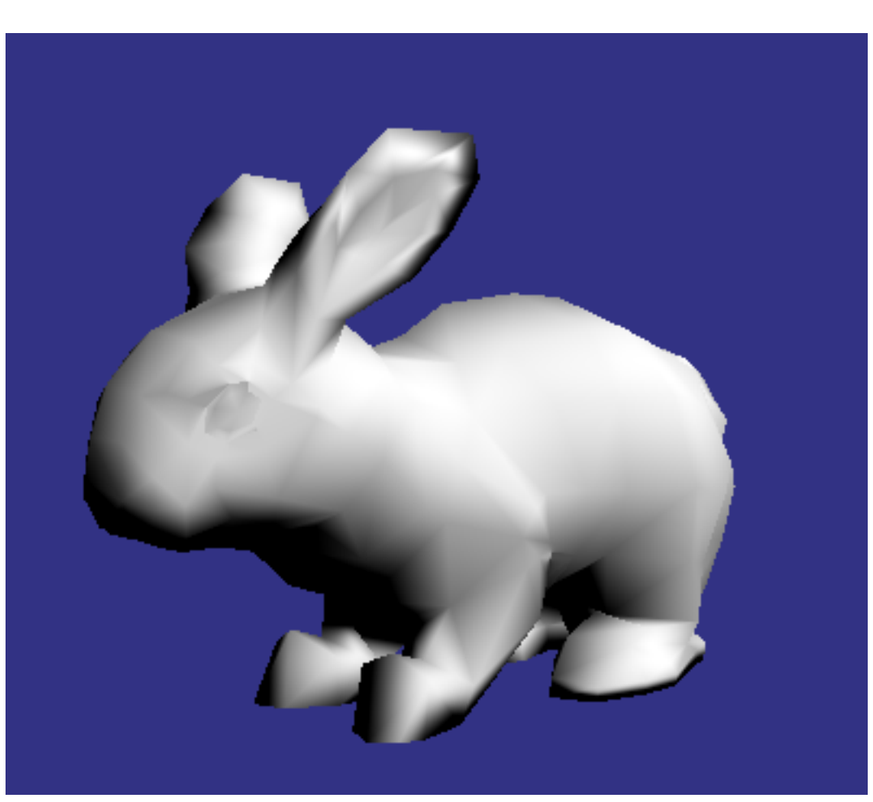


Not so common standard model: Teddy





Not so common standard model: That other bunny



Loading models into OpenGL:

LittleOBJLoader.h: our OBJ loader (header only)

TinyOBJLoader: Not as small, needs some of the calls in loadobj/LOL to load. Demo available.

Assimp: Big library, 7M, >400 files!

More models:

turbosquid.com

LittleOBJLoader can load most of these models if saved as OBJ.

Free and useful 3D modelling software:

Wings 3D - simple, quick learning curve

TinkerCAD - online, some limitations but much easier to work with than Wings

Blender - more powerful! Ask Susanne about this!

Summary

- GLSL
- Vertex and fragment shaders, varyings and uniforms
- Basic visible surface detection:
  back-face cull and Z buffer
- 3D object representation: The need for index table

Next lecture:

Illumination

Shading

Texture mapping