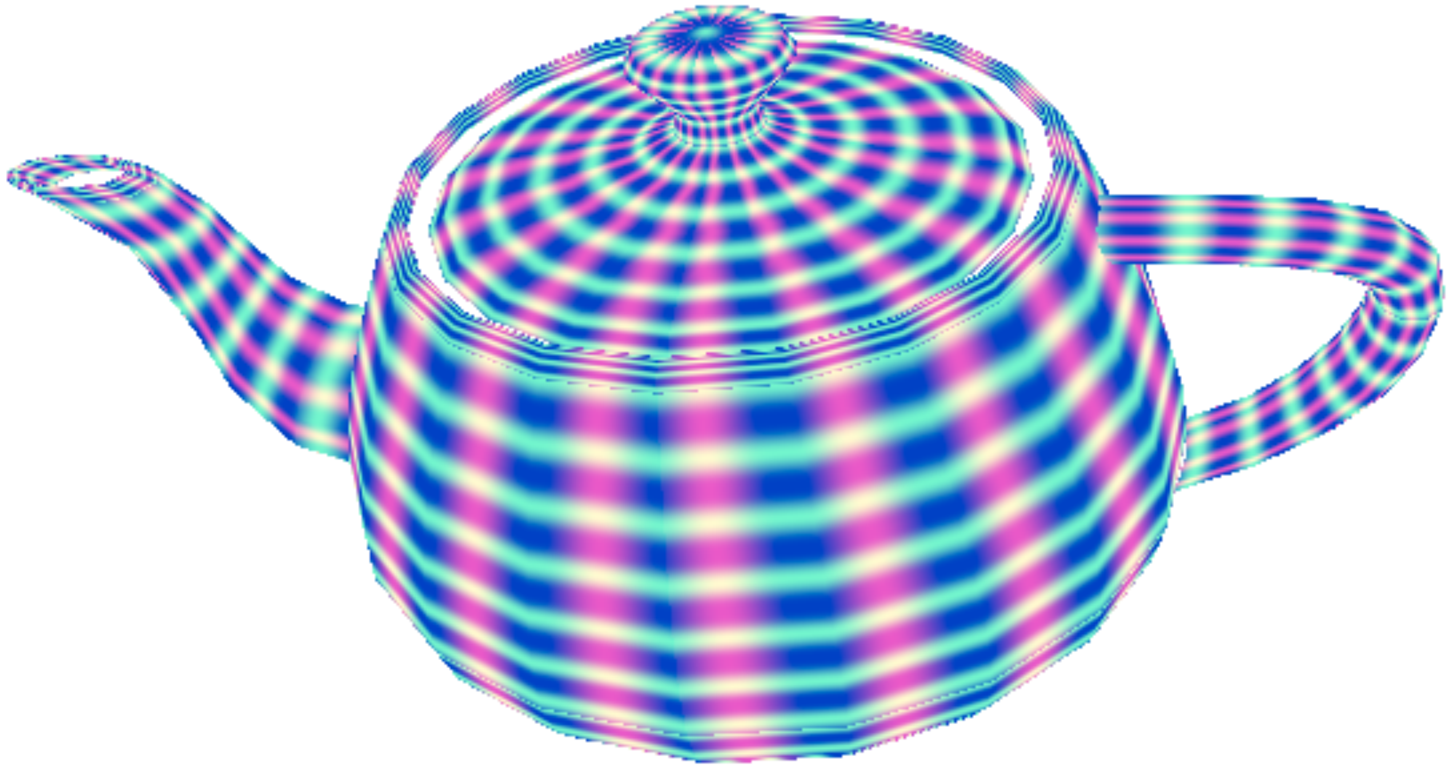
TNM084
Procedural images

Ingemar Ragnemalm, ISY



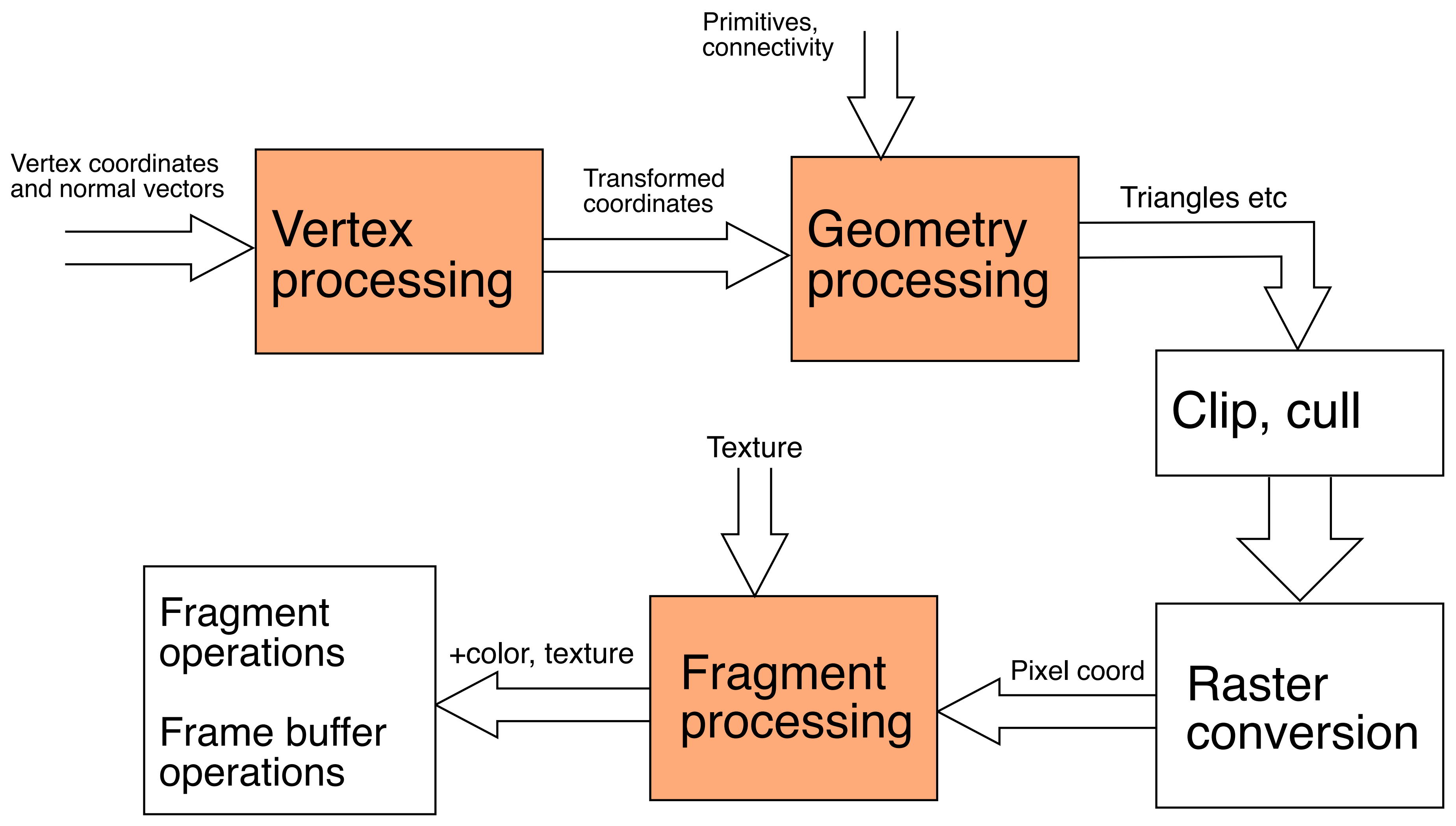
OpenGL and GLSL

OpenGL = Open Graphic Library

GLSL = OpenGL Shading Language

Open *specification*

OpenGL pipeline



Transformations in 2D and 3D with homogenous coordinates

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

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

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

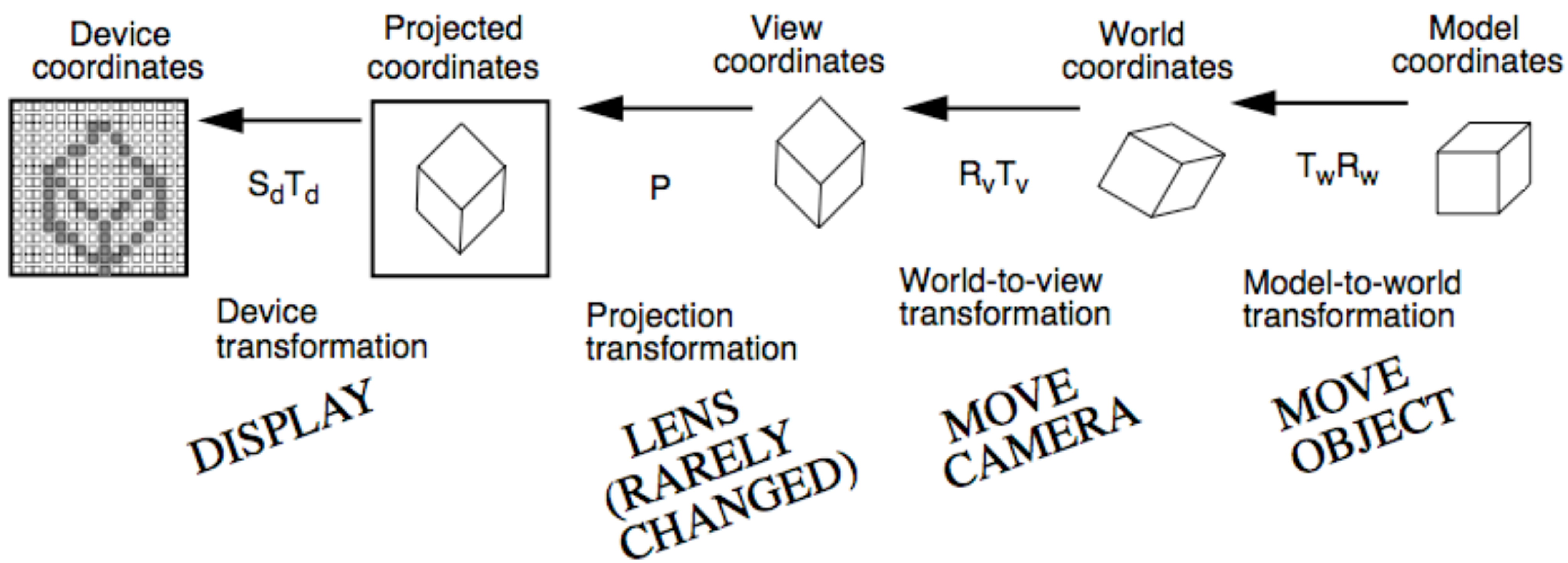
$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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

Passed to the vertex shader to compute transformations

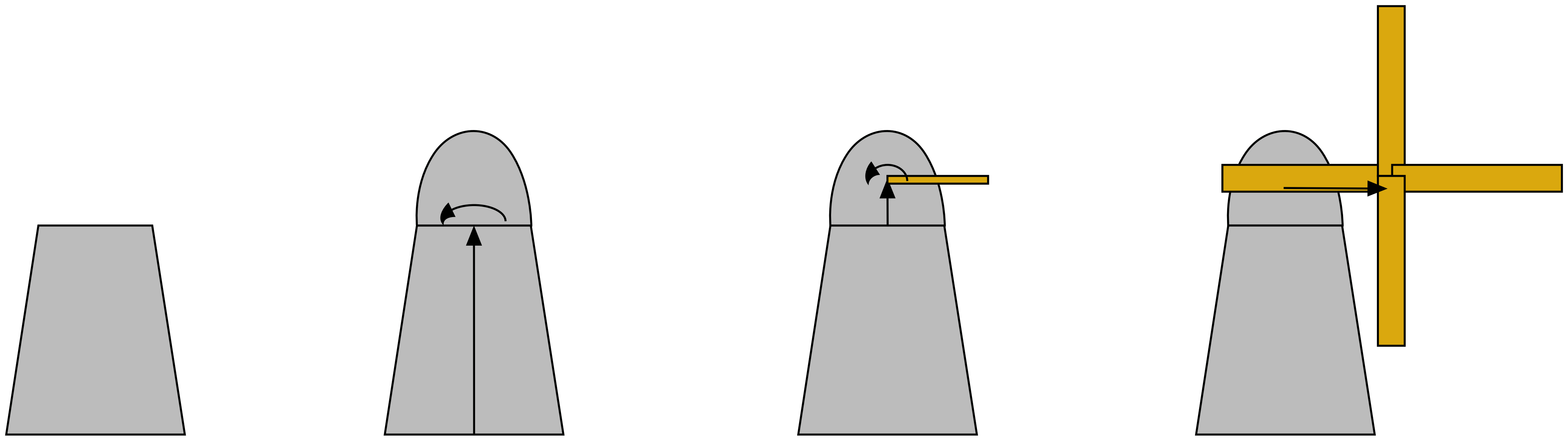
Transformation pipeline

Model coordinates
World coordinates
View coordinates
Projected coordinates
Device coordinates



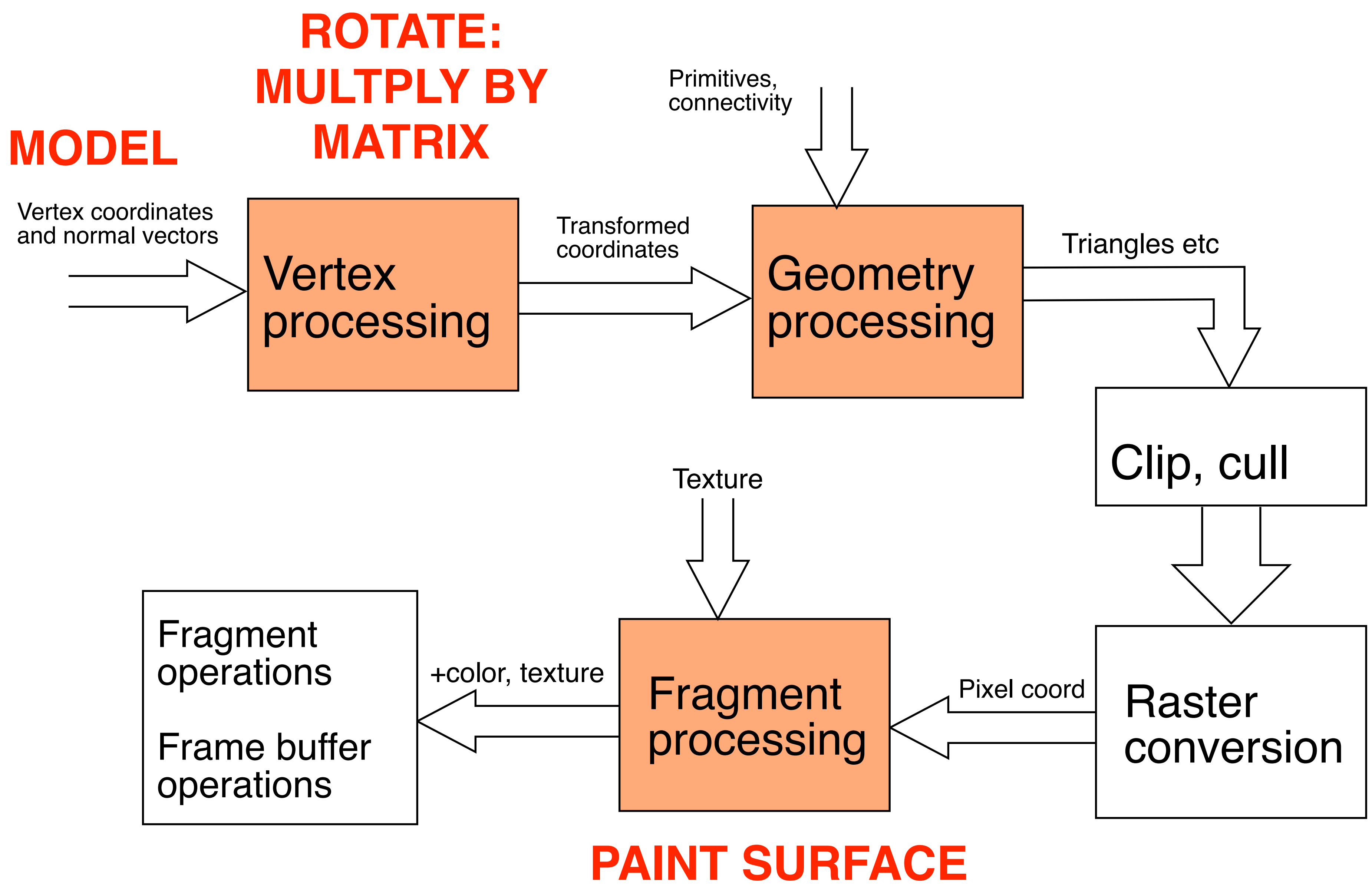
Transformations to sub-systems under model coordinates

Used for dependencies in hierarchical models



Model coordinates	Top of windmill	Axis for blades	Blade
Body of windmill	with rotation for top (around y)	The axis can rotate (around x or z)	Blades rotate by following the rotation of the axis

Example: Render a rotating model



The 3D graphic issues are not in focus to begin with.

We will initially focus on fragment shaders, shaders on pixel level

What is a shader?

Small program kernel that (in OpenGL) runs on the GPU!

Gives you great freedom to specify certain operations.

Run in parallel on multiple cores (hundreds or even thousands!) in the GPU! Extremely efficient!

But note that this description does not hold for OSL (later).

Vertex shader

Specifies transformations on each vertex

Translations, rotations...

Short program with data sent from the main program

In the example: Pass-through

Fragment shader

Specifies color of each pixel

Short program with data sent from the main program or vertex shader.

In the example: Set-to-white

Sample shaders

Minimal, doing pretty much nothing

Vertex shader:
Specifying positioning
(pass-through)

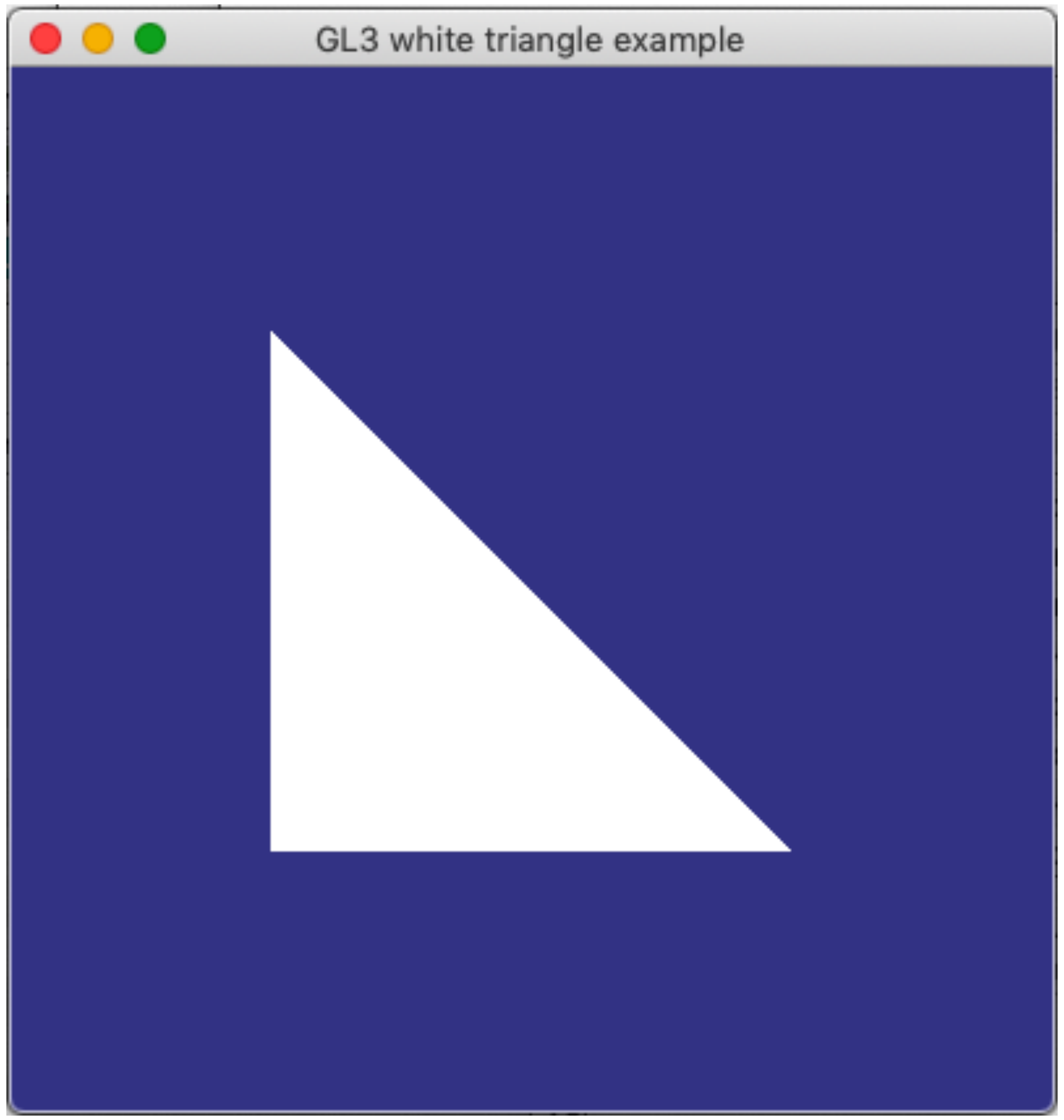
```
#version 150

in  vec3 in_Position;

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

Fragment shader:
Specifying pixel colors
(set-to-white)

```
#version 150

out vec4 out_Color;

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

Minimal shaders

Applied on a triangle




```
#version 150

in  vec3 in_Position;

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

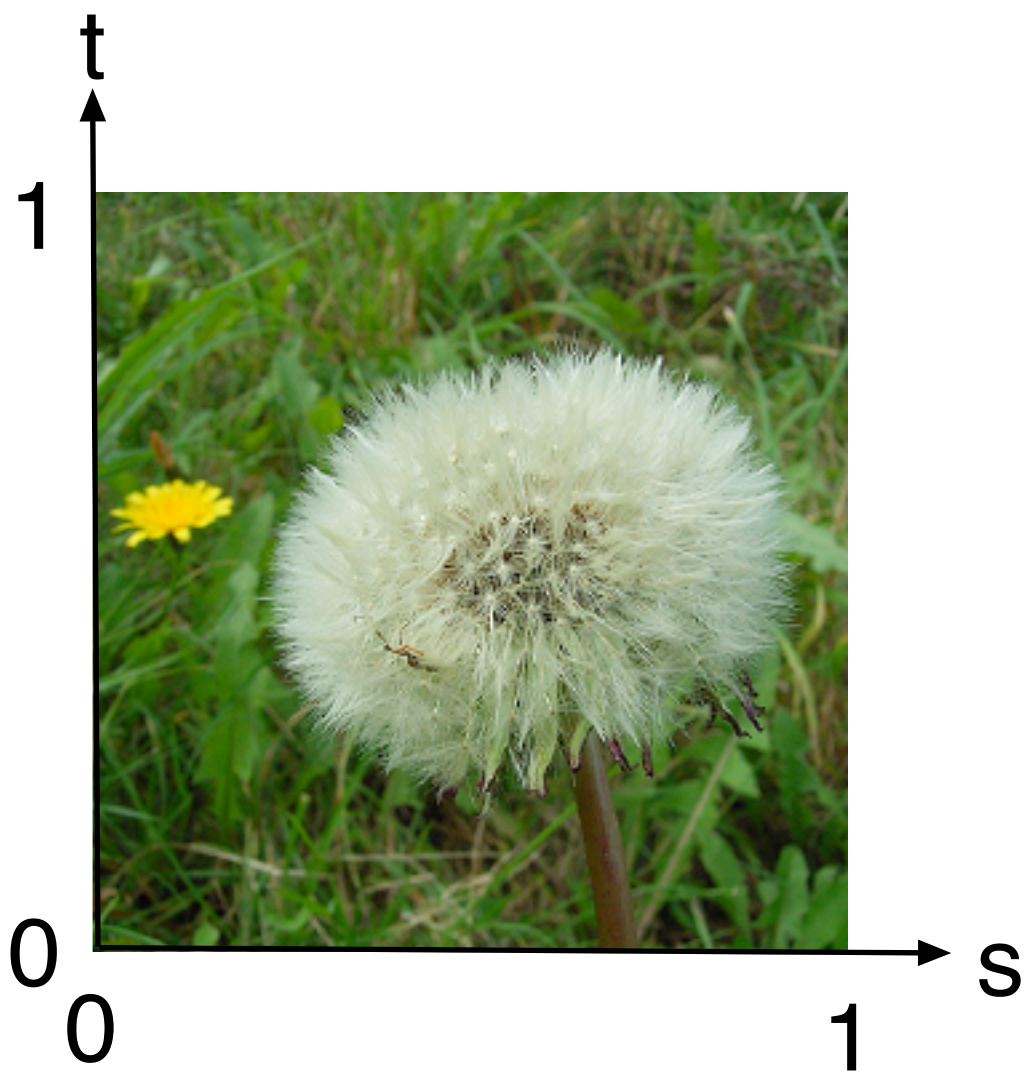
```
#version 150

out vec4 out_Color;

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

Texture mapping

“Wrap” a specified part of “texture space” onto an object

Consider the texture to be an elastic wrapping

Texture space



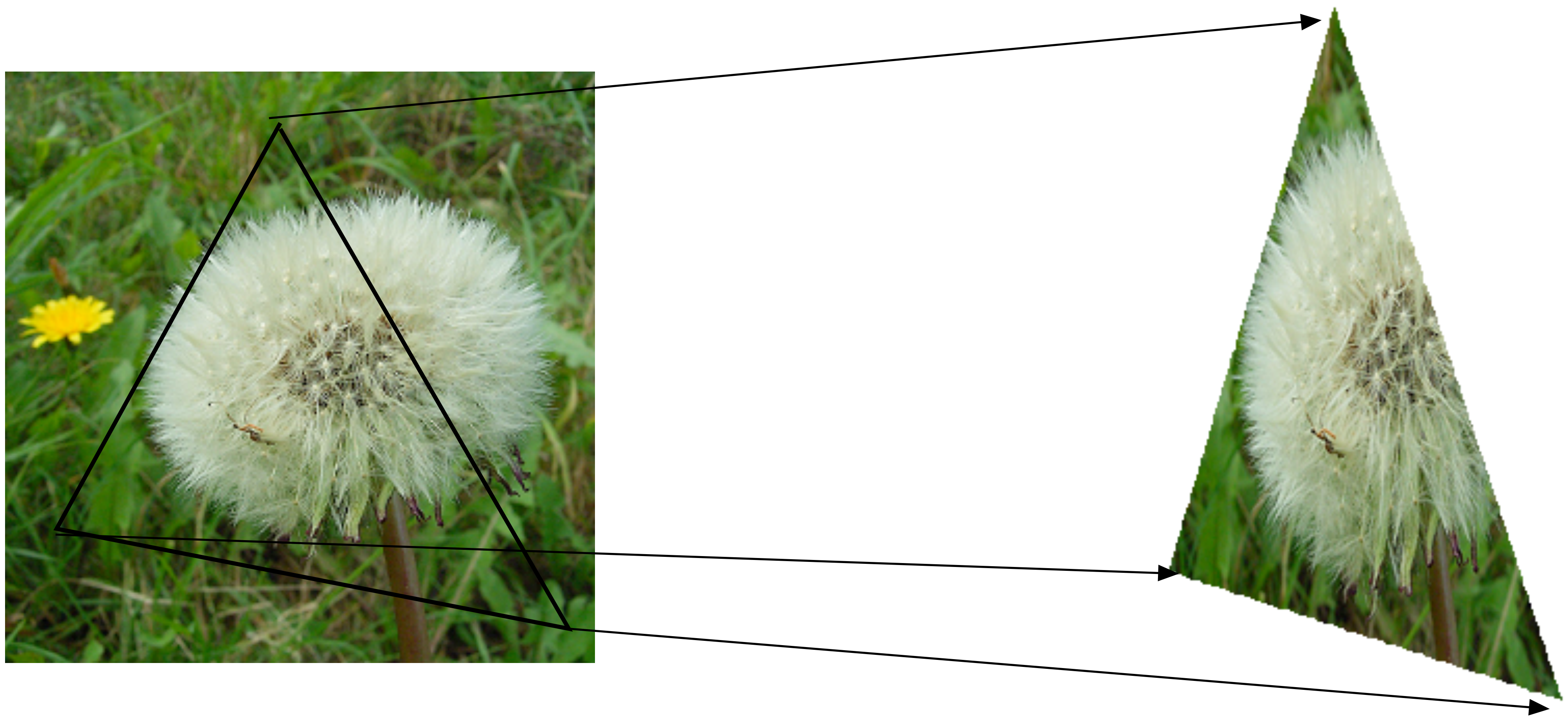
Texture = image used for texture mapping

Built from ”texels”

Texture space is usually 2-dimensional, (s, t), with textures defined in [0, 1]

Mapping from texture to surface

Each vertex has a texture coordinate; interpolate between, look up texture with interpolated coordinates.



Texture objects

Referring to already loaded textures

glGenTextures(...);
reserves texture numbers, making them available to use

glBindTexture(...);
makes a texture the current one

glTexImage2D(...);
loads a texture for the current texture number

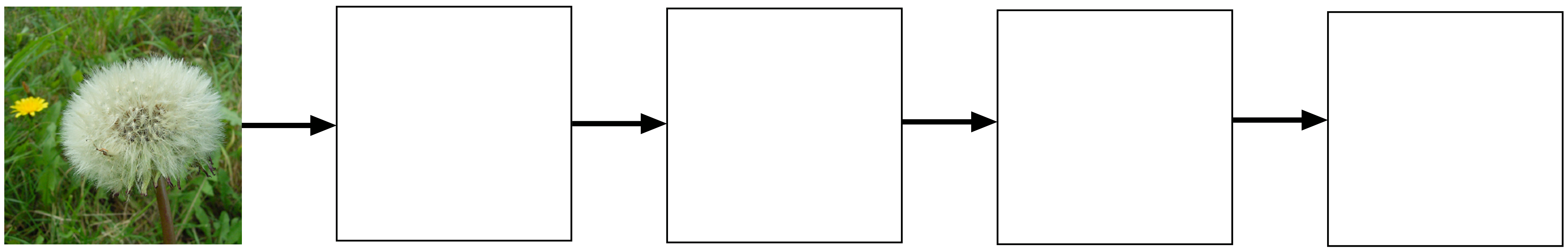
Important: Why do we have texture units? See for example "Polygons feel no pain" page 137.

Texture units

Texture access is complex

• Load from disc
• Bind to texture object
• Bind that to a texture unit
• Inform shader of texture unit
• Access texture unit from shader

but we are working procedurally and don't have to bother! :)



Texture coordinates

Texture coordinates often included in models.

LittleOBJLoaderX.h supports texture coordinates.

Pass as attribute array to vertex shader.

Interpolate from vertex to fragment shader.

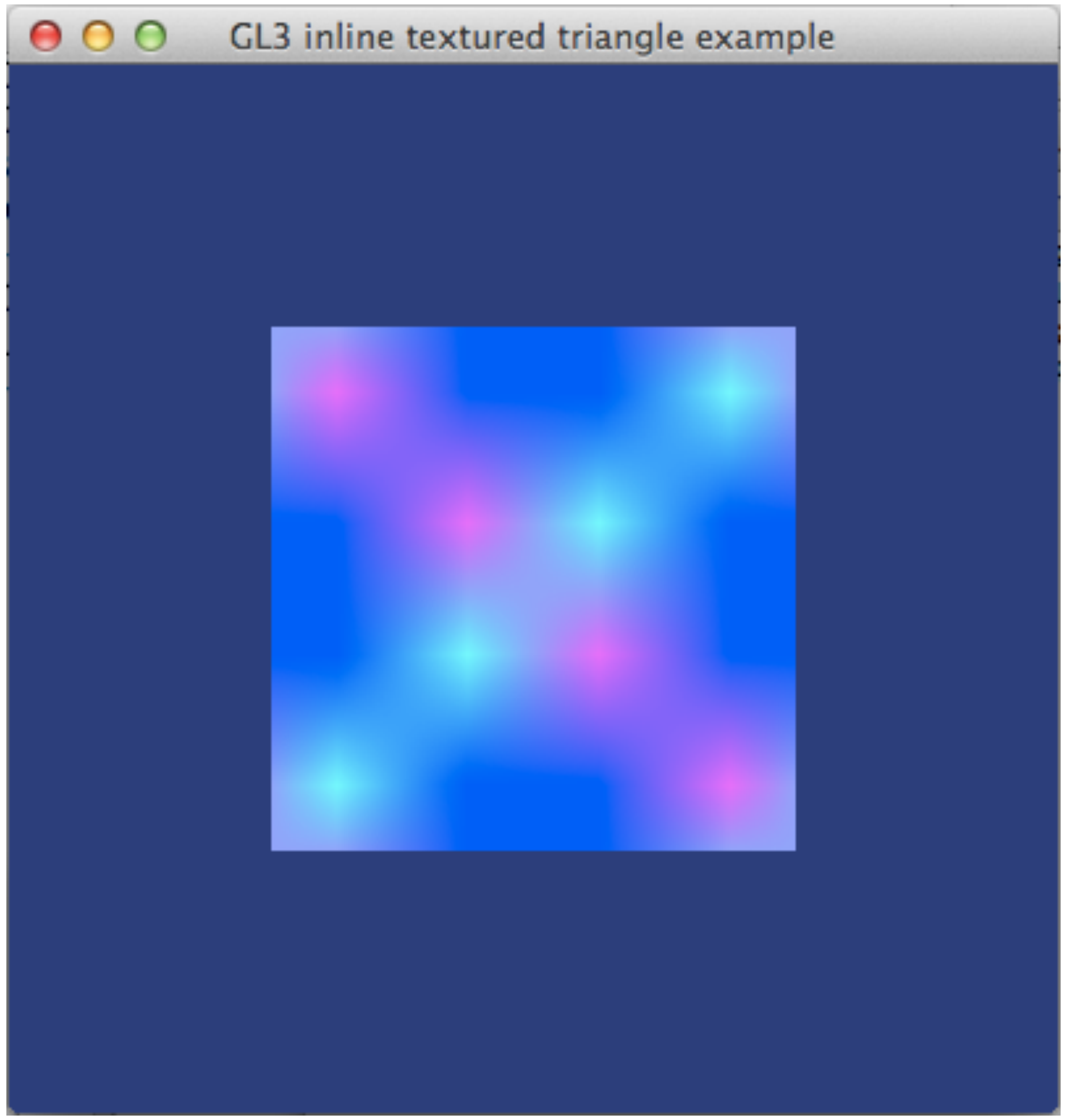
In-line textures from CPU

Simple textures can be given as arrays on the CPU.

It can also be generated by procedural algorithms!

```
// My "classic" inline texture (I have been using them for many
older demos)
  GLubyte minitex[4][4][3] =
  {
    { {255, 50,255}, { 50, 50,255}, { 50, 50,255}, { 50,255,255}},
    { { 50, 50,255}, {255, 50,255}, { 50,255,255}, { 50, 50,255}},
    { { 50, 50,255}, { 50,255,255}, {255, 50,255}, { 50, 50,255}},
    { { 50,255,255}, { 50, 50,255}, { 50, 50,255}, {255, 50,255}},
  };
```

In-line textures, demo



Procedural textures on quad

For lab 1, we will work on the simplest geometry: A quad over the whole viewport!

Texture from CPU

Texture on GPU

Example: Lab shell with CPU and GPU base

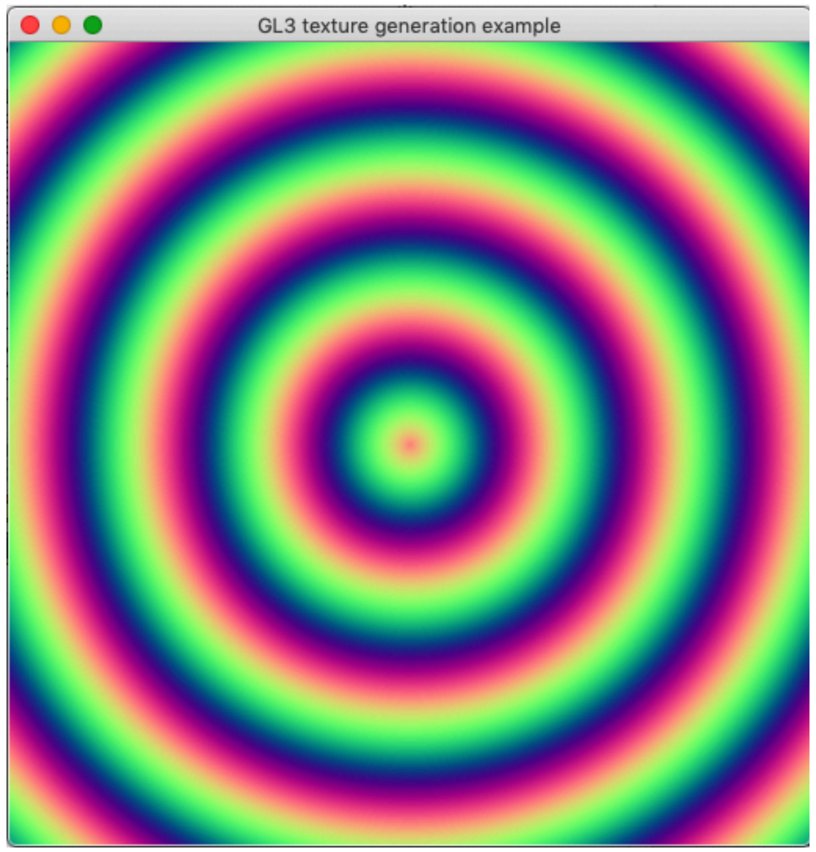
You do the first exercises on one of these at a time.

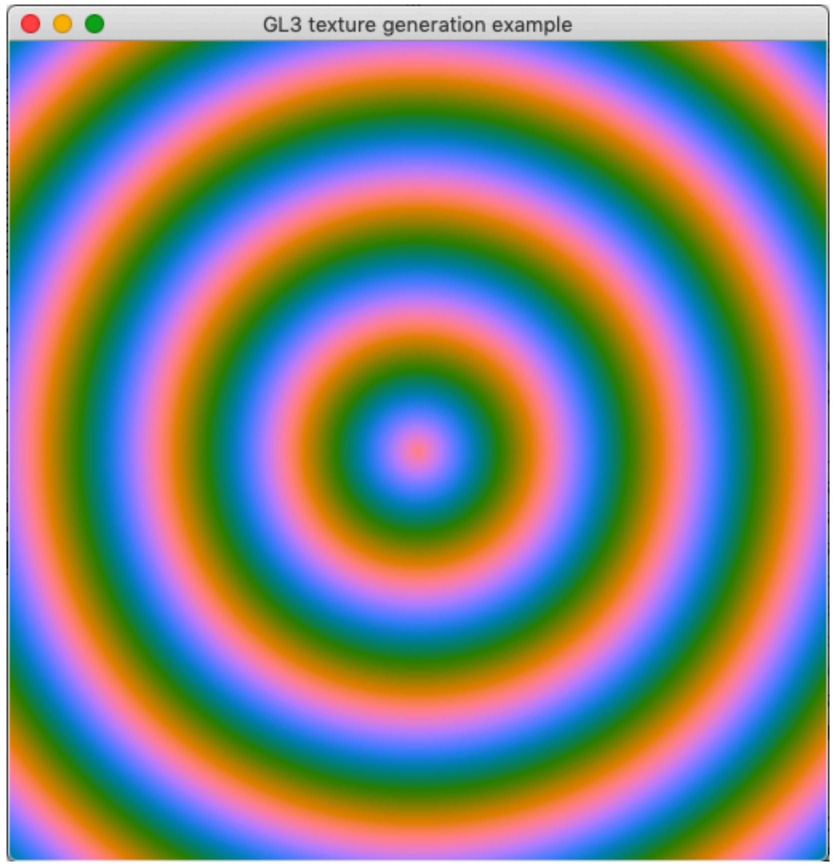
For later parts of the lab, you choose either.

Parallel patterns, GPU vs CPU

Lab material for lab 1

Initial: Trivial patterns for both CPU and GPU





Example:
Procedural texture

Texture generated by fragment shader!

- Vertex shader passes on texture coordinates
  - Texture coordinates are used in a texture generating function in the fragment shader

Simpler than you might think!

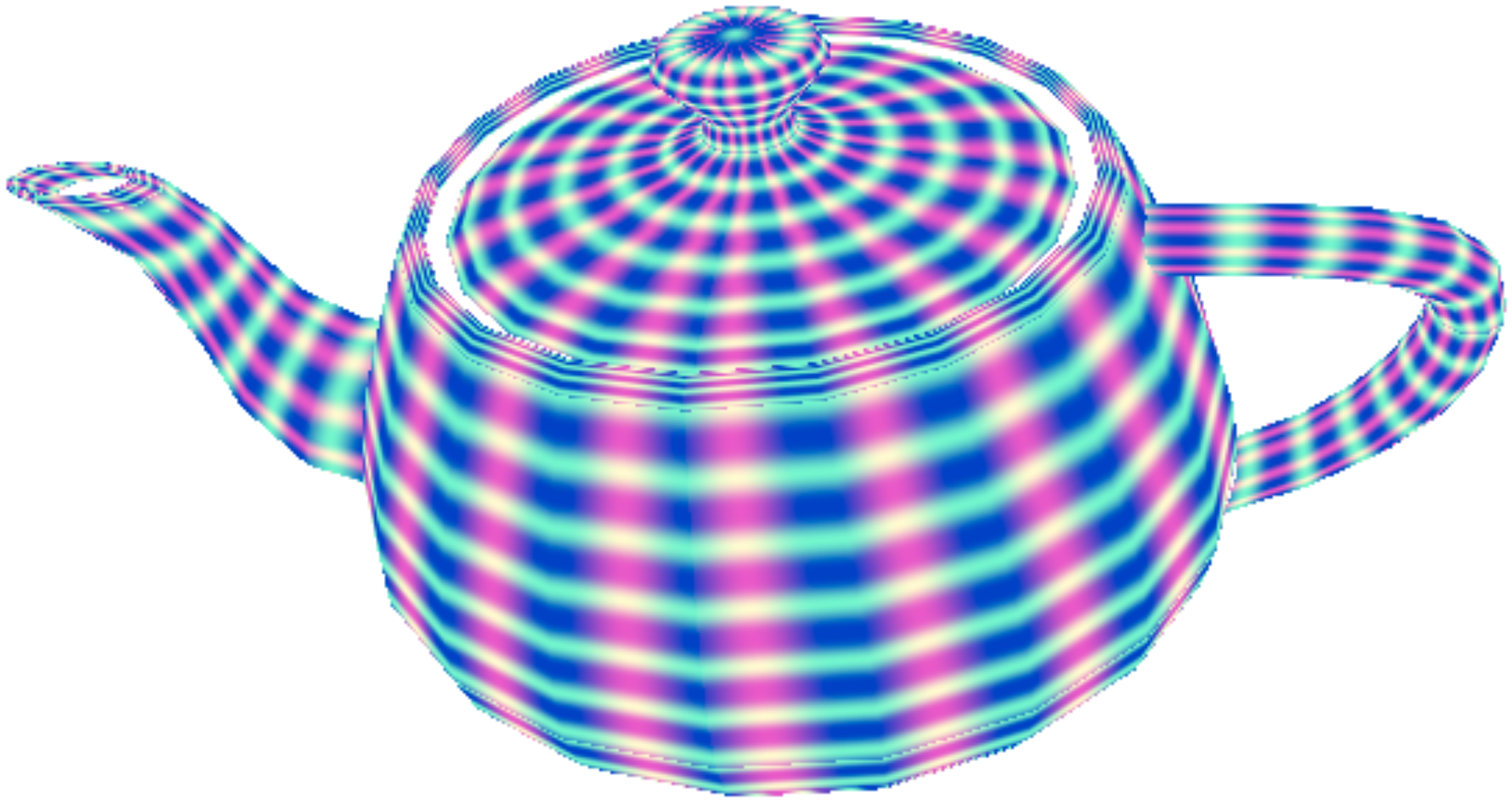
Procedural texture, Vertex shader

```
uniform mat4 proj;
uniform mat4 view;
out vec2 texCoord;
in vec2 inTexCoord;

void main()
{
   gl_Position = proj * view * gl_Vertex;
   texCoord = inTexCoord;
}
```

Procedural texture, Fragment shader

```
in vec2 texCoord;
out outColor;

void main()
{
   float a = sin(texCoord.s*30)/2+0.5;
   float b = sin(texCoord.t*30)/2+0.5;
   outColor = vec4(a, b, 1.0, 0.0);
}
```

Result: Ingemar's Psychedelic Teapot!

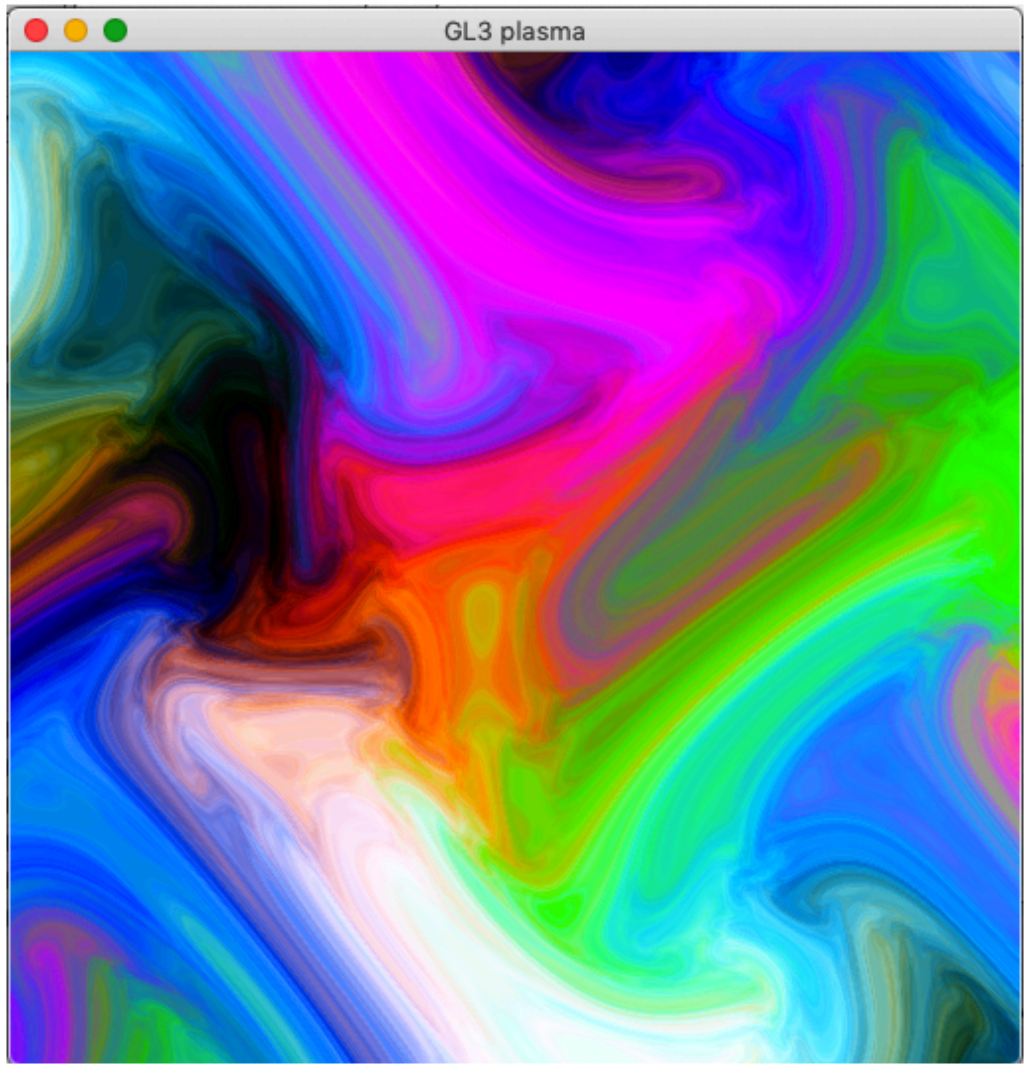


...but actually, I also add a time dependency:

```
#version 150

out vec4 outColor;
in vec2 texCoord;
uniform float t;

void main(void)
{
  float a = sin(texCoord.s * 30.0 + t)/2.0 + 0.5;
  float b = sin(texCoord.t * 30.0 * (1.0+sin(t/4.0)))/2.0 + 0.5;
  outColor = vec4(a, b, 0.8, 1.0);
}
```

"Plasma" demo

This is just a mix of multiple sin functions + time!



Topics ignored for now

Light calculation

Multi-texturing

Mip-mapping

Models on disc

...

because our focus now is procedural textures!

(More on models later.)

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.

#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 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!

Important!

Modifiers

Variable usage is declared with modifiers:

const

attribute (in)

uniform

varying (in/out)

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

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

uniform is argument from OpenGL, per primitive. Can not be changed within a primitive.

varying (”in”, ”out”)

data that should be interpolated between vertices

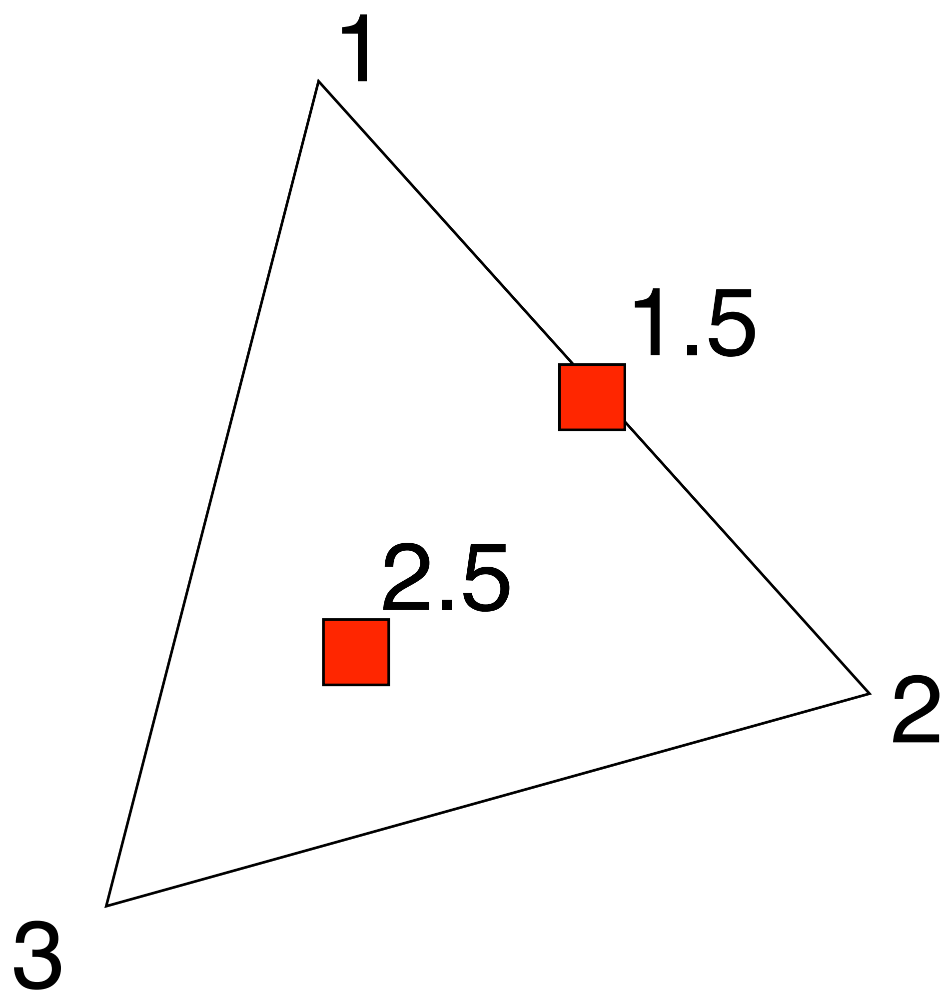
Written in vertex shader

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” or ”in/out”?

”varying” is 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.

In WebGL, you will still find the "varying" keyword

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;
```

we activate with:

```
glUseProgram(PROG);
```

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

Passing uniforms from the host

Often sent directly by main program

Most common: Scalars and matrices

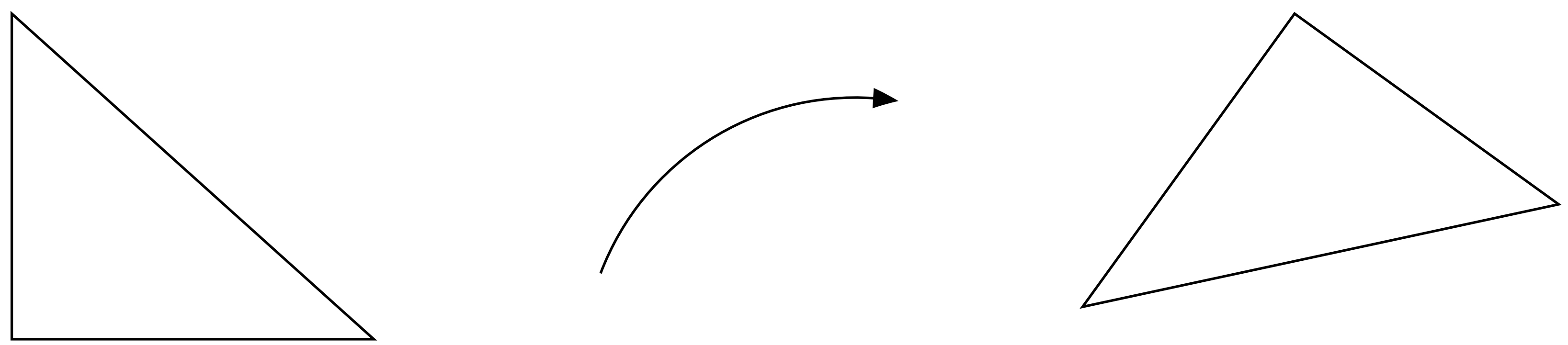
glUniform1f

glUniform1i

glUniformMatrix4fv

```
glUniformMatrix4fv(glGetUniformLocation(program, "mdlMatrix"), 1, GL_TRUE, total.m);
glUniform1f(glGetUniformLocation(program, "t"), t);
```

Uniform



Same rotation
for all vertices

Applied for entire primitive (e.g. model)

Declare as "uniform" in the shaders

Attributes

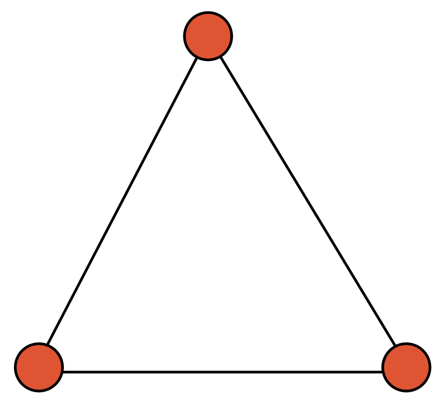
Different for every vertex

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

Passing attributes from the host

Sent as array buffers, VBO (vertex buffer object), grouped in VAOs (vertex array objects)

Several steps, usually packed "out of sight"

```
glGenVertexArrays(1, &vertexArrayObjID[i]);
glBindVertexArray(vertexArrayObjID[i]);
glGenBuffers(1, &vertexBufferObjID[i]);

glBindBuffer(GL_ARRAY_BUFFER, vertexBufferObjID[i]);
glBufferData(GL_ARRAY_BUFFER, 9*sizeof(GLfloat), vertices,
    GL_STATIC_DRAW);
glVertexAttribPointer(glGetAttribLocation(program, "in_Position"), 3, GL_FLOAT,
    GL_FALSE, 0, 0);
glEnableVertexAttribArray(glGetAttribLocation(program, "in_Position"));
```

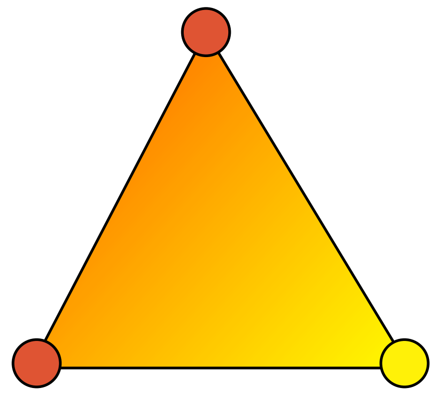
Varying

"out" from vertex shader

"in" into fragment shader

Interpolated between vertices!

Varying



Values sent from vertex shaders are interpolated and sent to fragments

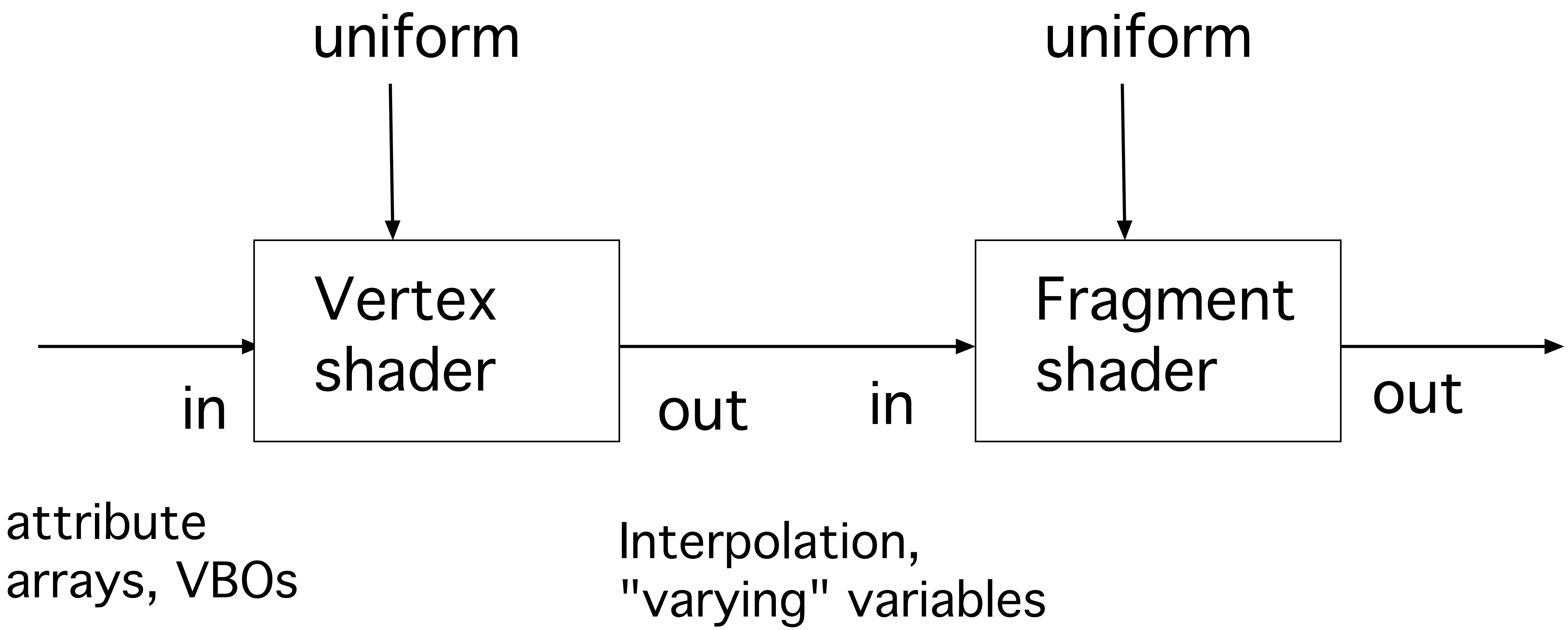
Simple usage: Set color in each vertex to get a gradient over the polygon

Output from fragment shader

Declared "out"

Typicaly 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);
}
```

Pass-through fragment shader

```
#version 150

out vec4 out_Color;

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

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;
}
```

More typical fragment shader

```
#version 150

in exTexCoord;
in exNormal;
// also textures, light sources...
out vec4 out_Color;

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

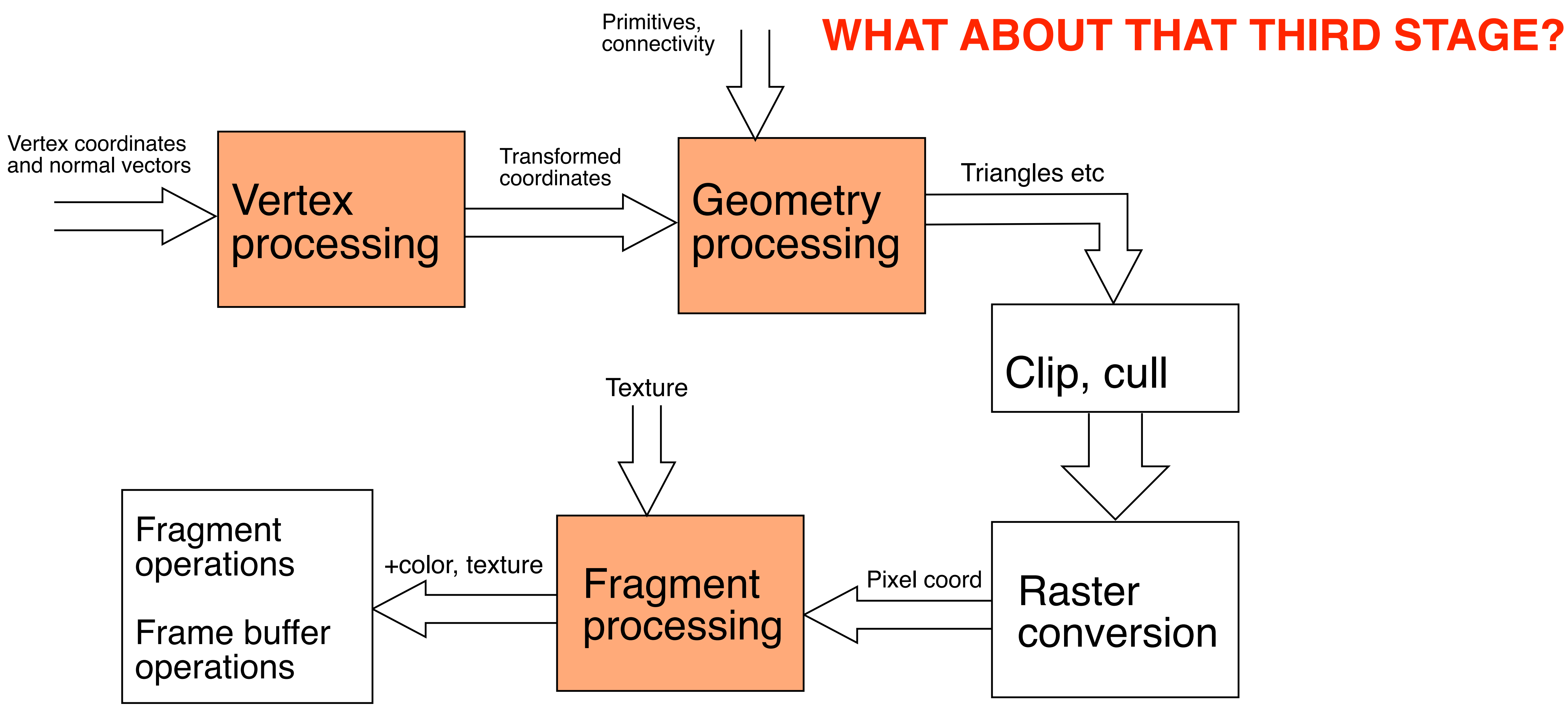
Fragment shaders for procedural texturing

- Need random numbers

- Numerical tricks for visual effects

- Need to define subroutines for standard noise generators

But there is also the geometry stage



More about this on a later lecture

Usage for each shader stage

Vertex shader: Feeds the fragment shader with interpolated data like texture coordinates and normal vectors

Geometry stage: Can modify and add geometry! Very useful for procedural methods!

Fragment stage: Obviously the place for procedural textures!

Questions on shaders?

Very important concept,
not worth leaving unclear!

Next week

Lab 1

Lecture 4: More noise

Lecture 5: OSL