centre adscrit a:

UNIVERSITAT DE BARCELONA

ESCOLA DE NOVES TECNOLOGIES INTERACTIVES

# Computer Graphics

## 3. My first shader

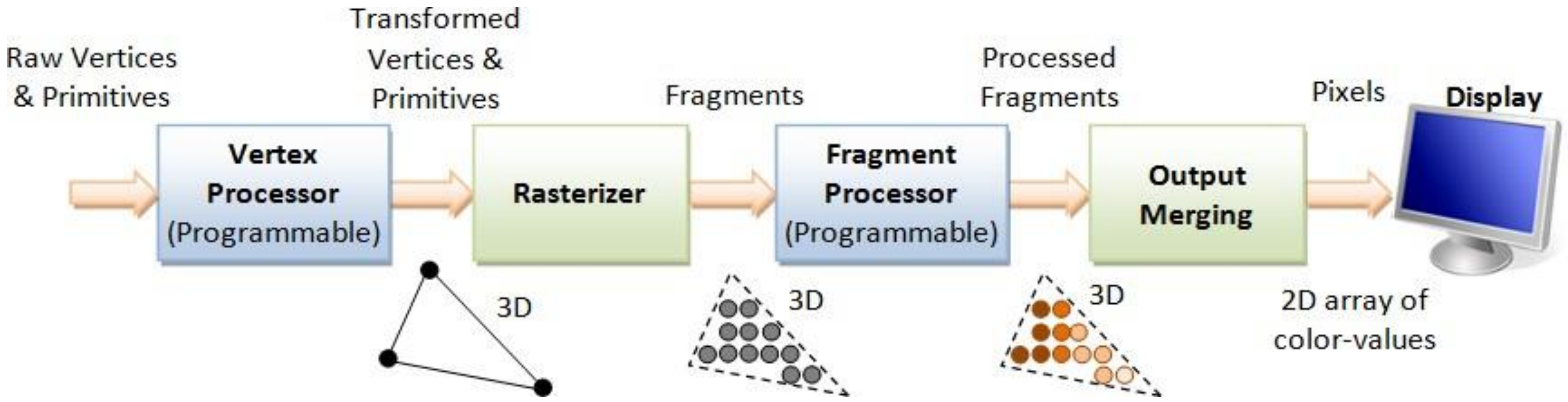
Dr Joan Llobera – joanllobera@enti.cat

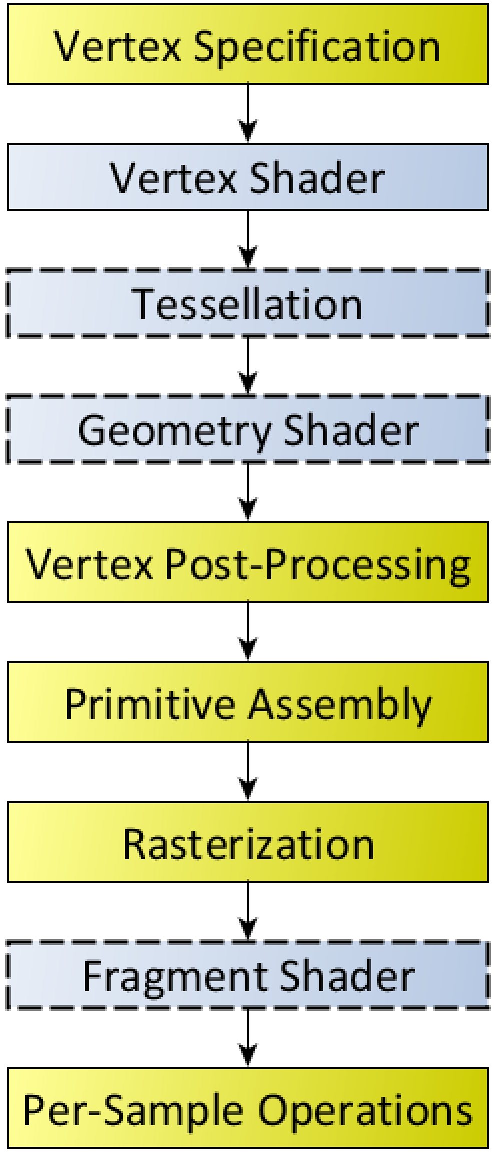
Spring 2019

# Outline

1. The programmable stages (reminder)
2. Draw a colour
3. Draw a vertex
4. Draw a triangle
5. Transform operations, a bit closer
6. Homework!

# 1. Programmable stages in OpenGL



# EX1.a Draw Color

```
static const GLfloat red[] = { 1.0f,0.0f,0.0f,1.0f };
glClearBufferfv(GL_COLOR, 0, red);
```

```
Prototype:
void glClearBufferfv(GL_ENUM buffer,GLint drawBuffer, const GLfloat * value);

buffer                  (kind of buffer)
drawBuffer              (0 if only using 1)
value                   (float or vector)
```

# EX1b Draw Changing Color

# EX1.b Draw Changing color. Solution

```
const GLfloat color[] = { (float)sin(currentTime) * 0.5f + 0.5f,
(float)cos(currentTime)*0.5f + 0.5f, 0.0f, 1.0f };
glClearBufferfv(GL_COLOR, 0, color);
```

# EX1.c A basic Shader (1/6)

**A vertex shader:**

```
static const GLchar * vertex_shader_source[] =
{
"#version 330\n\
\n\
void main() {\n\
gl_Position = vec4(0.0,0.0,0.5, 1.0);\n\
}"
};
```

# EX1.c A basic shader (2/6)

**A fragment shader:**

```
static const GLchar * fragment_shader_source[] =
{
"#version 330\n\
\n\
out vec4 color;\n\
\n\
void main() {\n\
 color = vec4(0.0,0.8,1.0,1.0);\n\
}"
};
```

# EX1.c A basic shader (3/6)

**We create and compile the two shaders:**

```
GLuint compile_shaders(void){

GLuint vertex_shader;
GLuint fragment_shader;
GLuint program;

vertex_shader =
glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertex_shader, 1,
vertex_shader_source, NULL);
glCompileShader(vertex_shader);
```

```
fragment_shader =
glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fragment_shader, 1,
fragment_shader_source, NULL);
glCompileShader(fragment_shader);

program = glCreateProgram();
glAttachShader(program, vertex_shader);
glAttachShader(program, fragment_shader);
glLinkProgram(program);

glDeleteShader(vertex_shader);
glDeleteShader(fragment_shader);
return program;
}
```

# EX1.c A basic shader (4/6)

**glCreateShader** creates an empty shader object

**glShaderSource** hands shader source code to the shader object

**glCompileShader** compiles whatever source code is contained in the shader object

**glCreateProgram** creates a program to which attach shaders

**glAttachShader**

**glLinkProgram**

**glDeleteShader**

# EX1.c A basic shader (5/6)

**A Vertex Array Object (VAO)**

Maintains all of the state related to input of the OpenGL pipeline.

**Two more primitives:**

```
void glCreateVertexArrays(GLsizei n,
GLuint * arrays);
```

Creates the VAO (in glinit)

```
void glBindVertexArray(GLuint array);
```

Binds the VAO to the current context (in glrender)

# EX1.c A basic shader (6/6)

**The Rendering code**

Draws everything together

**Two more primitives:**

```
glUseProgram(myRenderProgram);
glDrawArrays(GL_POINTS, 0, 1);
```

**In addition to:**

```
glPointSize(40.0f);
```

- In GlInit()

```
glGenVertexArrays(1, &myVao);
myRenderProgram = com
```

- In GlInit()

```
glPointSize(40.0f);
glBindVertexArray(myVao);
glUseProgram(myRenderProgram);
glDrawArrays(GL_POINTS, 0, 1);
```

# EX2. Hello World! (or rather… Hello Triangle)

```
static const GLchar *
vertex_shader_source[] =
{
"#version 330

void main() {
const vec4 vertices[3] = vec4[3](
vec4( 0.25, -0.25, 0.5, 1.0),
vec4(-0.25, -0.25, 0.5, 1.0),
vec4( 0.25,  0.25, 0.5, 1.0));
gl_Position =
vertices[gl_VertexID];
}"
};
```

```
static const GLchar *
fragment_shader_source[] =
{
"#version 330\n\
\n\
out vec4 color;\n\
\n\
void main() {\n\
color = vec4(0.0,0.8,1.0,1.0);\n\
}"
};
```

# EX2. Hello Triangle

**The Rendering code**

Draws everything together

**Two more primitives:**

`glDrawArrays(GL_TRIANGLES, 0, 3);`

Will draw triangles instead of points.

`GL_LINE_LOOP` will draw a triangle outline

`GL_LINES` will only draw a line with the first pair of vertexes

In vertex shader,

`gl_VertexID`

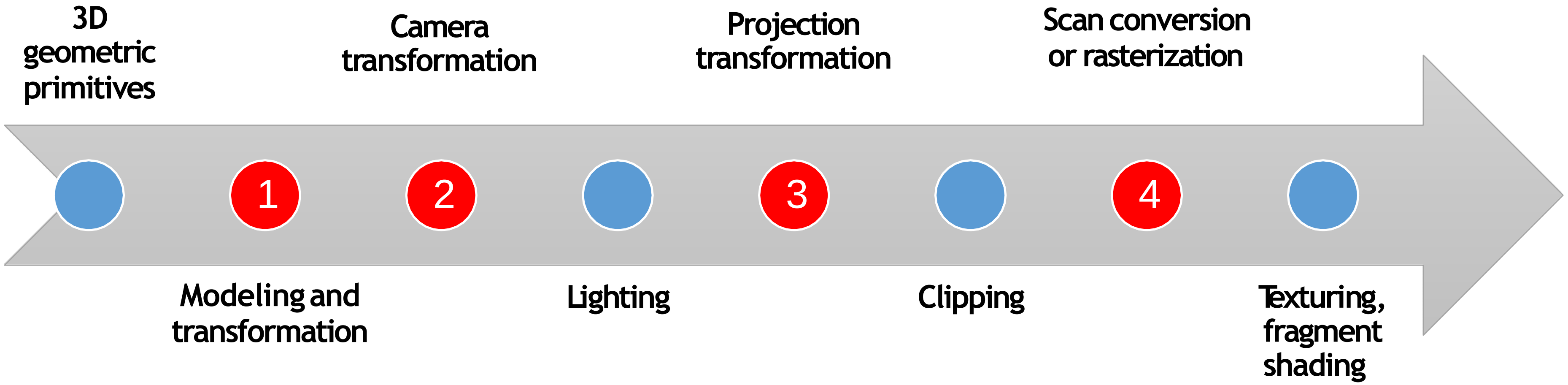
Starts counting from the value given in first `glDrawArrays`

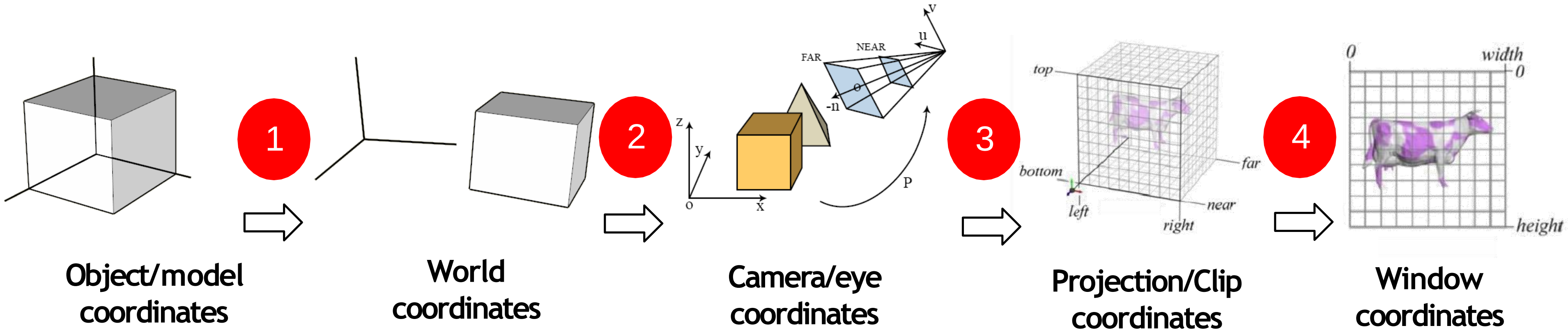
Note: Since OpenGL follows the anti-hour convention for normal, you will need to make sure that your arrays define a normal that goes toward the camera. Place the vertex in space, and use your right hand to check how the rotation generates a normal that points to you. Therefore, to draw the triangle with `GL_TRIANGLES`, the shader actually needs to look like this:

```
static const GLchar * vertex_shader_source[] =
{
"#version 330

void main() {
const vec4 vertices[3] = vec4[3](
vec4( 0.25, -0.25, 0.5, 1.0),
vec4( 0.25,  0.25, 0.5, 1.0),
vec4(-0.25, -0.25, 0.5, 1.0));
gl_Position = vertices[gl_VertexID];
}"
};
```

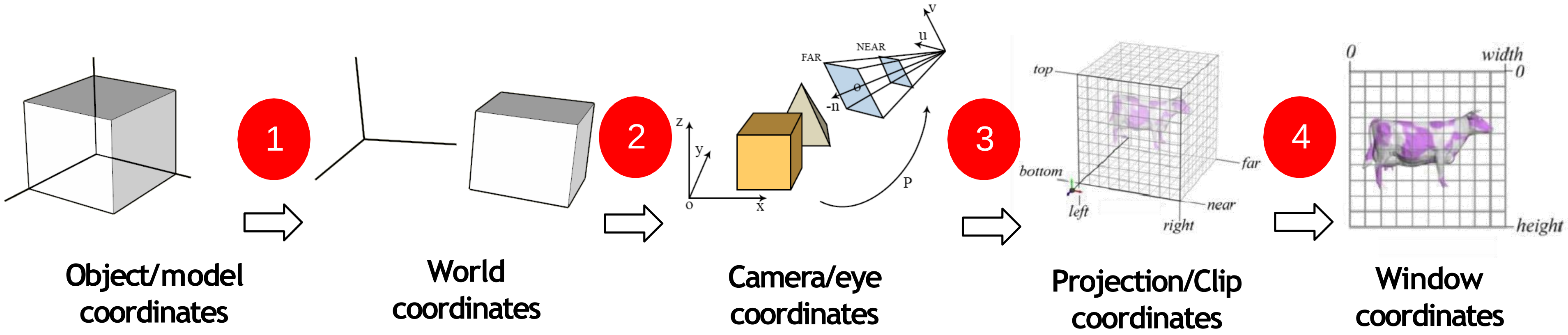
# Reminder. Transformation operations



# Reminder. Transformation operations

1. **Model transformation.** It transforms a position in a model to the position in the world
2. **View transformation.** The camera in OpenGL cannot move and is defined to be located at (0,0,0) facing the negative Z direction. That means that instead of moving and rotating the camera, the world is moved and rotated around the camera to construct the appropriate view
3. **Projection transformation.** It projects the information to clip coordinates that are in normalized devices coordinates
4. **Viewport transformation.** It adapts the image to the window resolution



# 5. Translate and rotation operations

```
RV::_modelView = glm::mat4(1.f);
RV::_modelView = glm::translate(RV::_modelView,
glm::vec3(RV::panv[0], RV::panv[1], RV::panv[2]));
RV::_modelView = glm::rotate(RV::_modelView,
RV::rota[1], glm::vec3(1.f, 0.f, 0.f));
RV::_modelView = glm::rotate(RV::_modelView,
RV::rota[0], glm::vec3(0.f, 1.f, 0.f));

RV::_MVP = RV::_projection * RV::_modelView;
```

```cpp
void GLmousecb(MouseEvent ev) {
if(RV::prevMouse.waspressed && RV::prevMouse.button == ev.button) {
float diffx = ev.posx - RV::prevMouse.lastx;
float diffy = ev.posy - RV::prevMouse.lasty;
switch(ev.button) {
case MouseEvent::Button::Left: // ROTATE
RV::rota[0] += diffx * 0.005f;
RV::rota[1] += diffy * 0.005f;
break;
case MouseEvent::Button::Right: // MOVE XY
RV::panv[0] += diffx * 0.03f;
RV::panv[1] -= diffy * 0.03f;
break;
case MouseEvent::Button::Middle: // MOVE Z
RV::panv[2] += diffy * 0.05f;
break;
default: break;
}
} else {
RV::prevMouse.button = ev.button;
RV::prevMouse.waspressed = true;
}
RV::prevMouse.lastx = ev.posx;
RV::prevMouse.lasty = ev.posy;
}
```

# 6.Homework!

https://www.youtube.com/watch?v=Amj6RiGiTOE&t=283s

# 6.Homework!

**Goal:** The goal of practice 1 will require you to implement camera and world transformations showing a basic understanding about how the camera, world and object coordinates work in openGL

**Tasks:** Watch the video and think how you would implement such camera movements.



https://www.youtube.com/watch?v=Amj6RiGiTOE&t=283s

# Resources

- [Sellers2016] Graham Sellers, Richard S. Wright, Jr. Nicholas Haemel (2016) *OpenGL SuperBible*, 6th Edition. Pearson education (chapters 2 and 3)