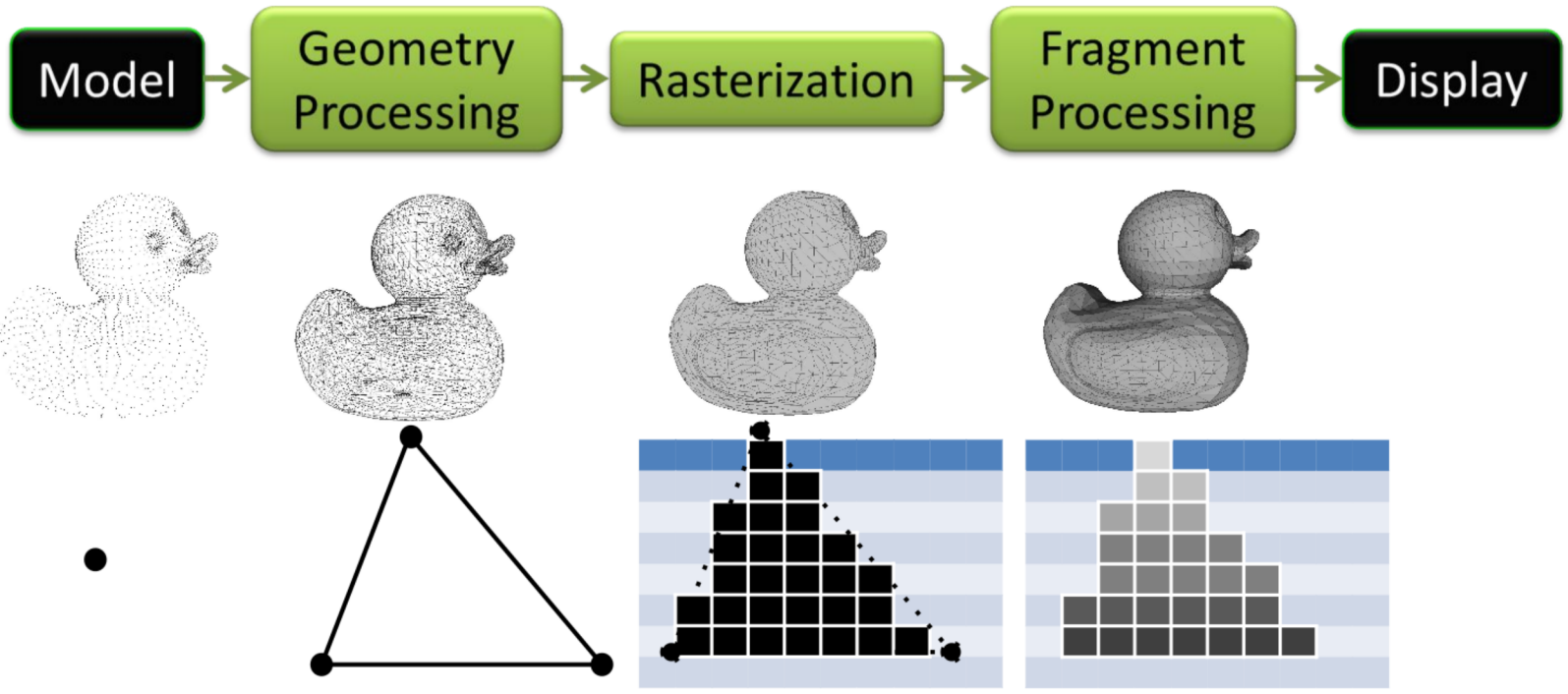
Computer Graphics Today: GPUs and Graphics API

Prof. PhD Rafael P. Torchelsen
rafael.torchelsen@inf.ufpel.edu.br

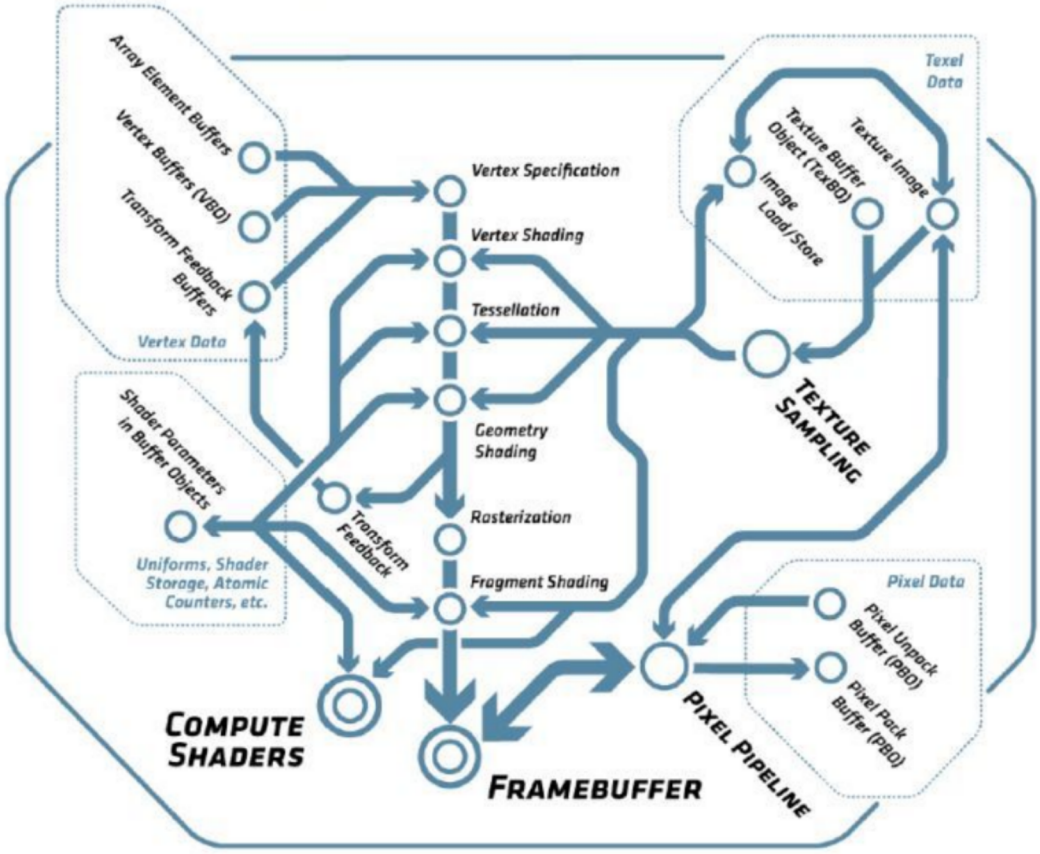
Raster Pipeline Overview

computação
Universidade Federal de Pelotas



OpenGL 4.5

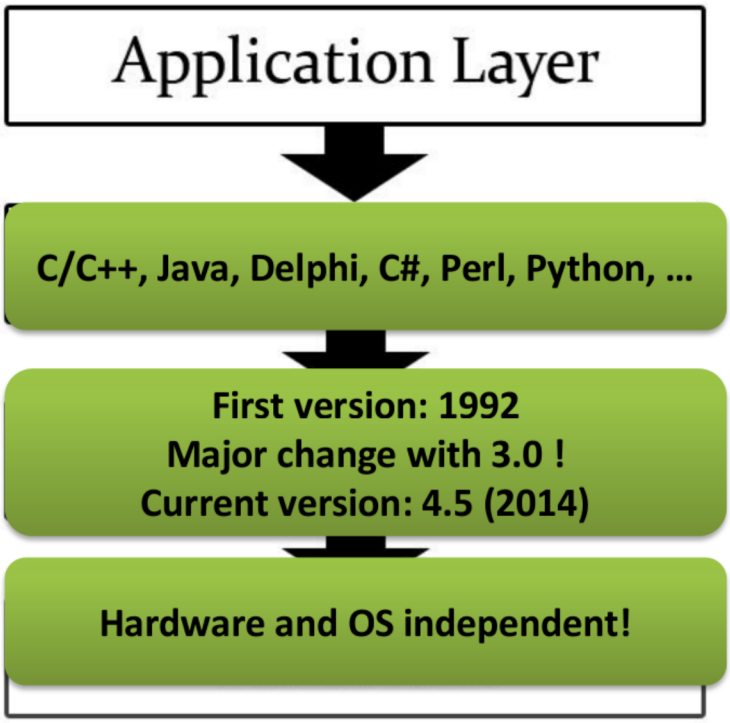
CORE PROFILE



OpenGL

computação
Universidade Federal de Pelotas

OpenGL is a software **interface** that allows a programmer to communicate with **graphics hardware**



http://www.opengl.org/

Not the only one

computação
Universidade Federal de Pelotas

Family

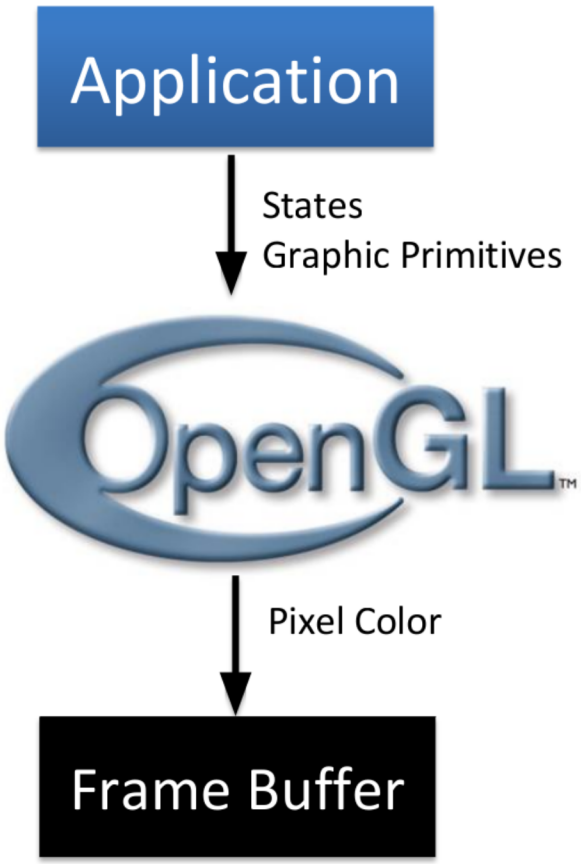
OpenGL ES and WebGL

computação
Universidade Federal de Pelotas

- OpenGL ES 2.0
  - Designed for embedded and hand-held devices such as cell phones
  - Based on OpenGL 3.1
  - Shader based
- WebGL
  - JavaScript implementation of ES 2.0
  - Runs on most recent browsers

OpenGL = State Machine

computação
Universidade Federal de Pelotas



Change states (pipeline) and send **primitives** thought the state machine (pipeline)

The pipeline format is defined by the **current state of the machine**

It´s not fixed!

User Input, Window creation, sound, etc

- We need **another library** for that: GLUT, GLFW, SDL, etc…

- We will use **GLFW** for input and window handling
  - http://www.glfw.org/

OpenGL Implementations

computação
Universidade Federal de Pelotas

- Official OpenGL Version: 1.0, 2.0, 3.0, 4.0, ...
- **Besides** the **official** features there are some features available only on specific products, those features are called **extensions**

GLEW

http://glew.sourceforge.net

OpenGL Mathematics (GLM)

computação
Universidade Federal de Pelotas

http://glm.g-truc.net/

OpenGL Mathematics (GLM)

computação
Universidade Federal de Pelotas

```
void Test()
{
  glm::vec4 Position = glm::vec4(glm:: vec3(0.0f), 1.0f);
  glm::mat4 Model = glm::translate(glm::mat4(1.0f), glm::vec3(1.0f));
  glm::vec4 Transformed = Model * Position;
}
```

```
void GlmToOpengl()
{
  glm::vec4 v(0.0f);
  glm::mat4 m(1.0f);
  ...
  glVertex3fv(&v[0]);  //OpenGL interface
  glLoadMatrixfv(&m[0][0]);
}
```

Shading Language

computação
Universidade Federal de Pelotas

A shading language is a **graphics programming language**

Data structures: color, normal, etc…

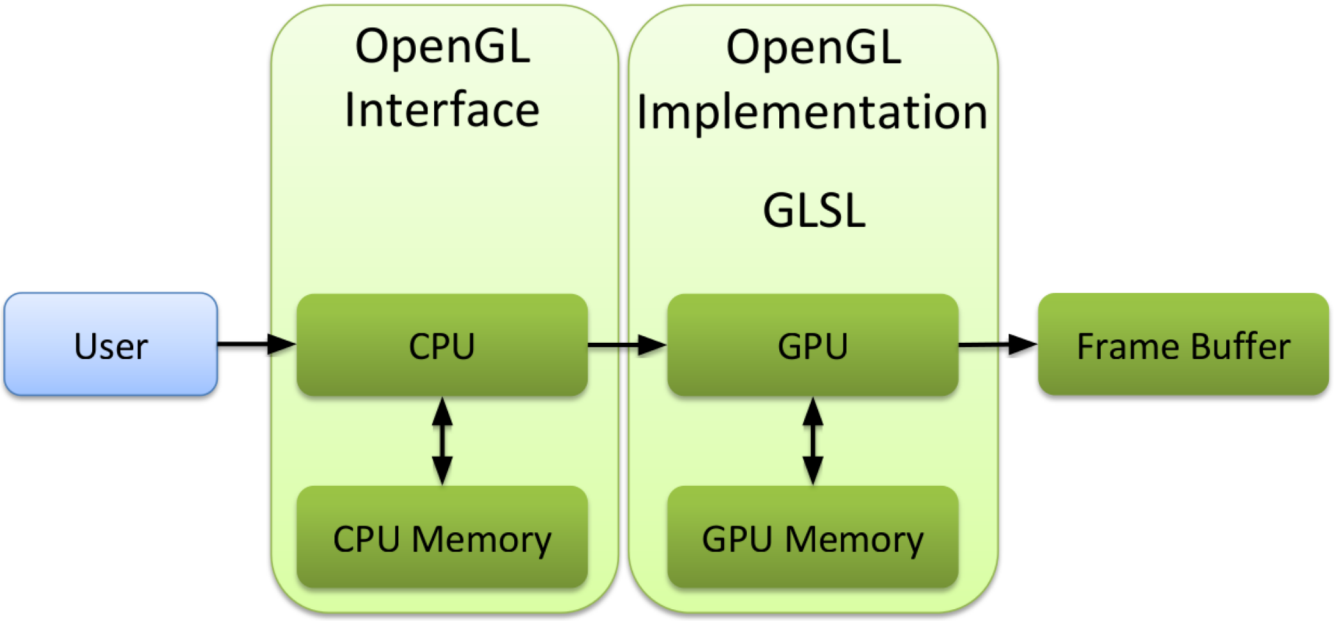
Operations: dot, cross product, etc...

GLSL, HLSL, Cg

Usually highly **parallel**

Information Pipeline

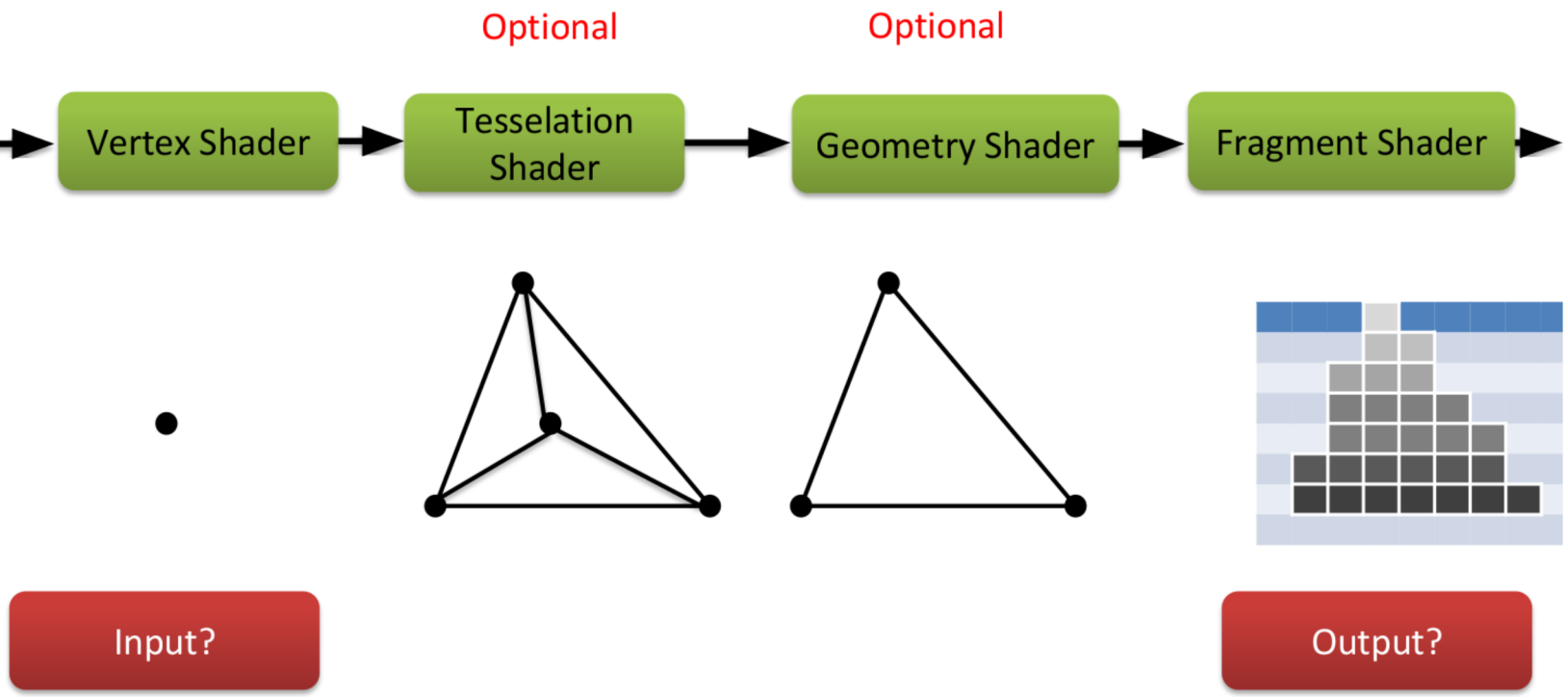
computação
Universidade Federal de Pelotas



Shading Language: GLSL

computação
Universidade Federal de Pelotas

- OpenGL Shading Language
- GPU = SIMD (Single Instruction, Multiple Dada)

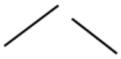


Input: Geometric Primitive

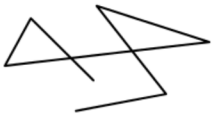
computação
Universidade Federal de Pelotas

All primitives are specified by **vertices**

GL_POINTS

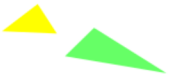


GL_LINES



GL_LINE_STRIP

GL_LINE_LOOP



GL_TRIANGLES

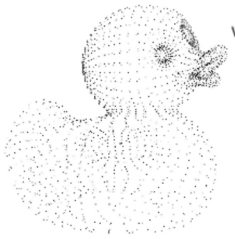
GL_TRIANGLE_STRIP

GL_TRIANGLE_FAN

Current GPU **Graphics** Pipeline

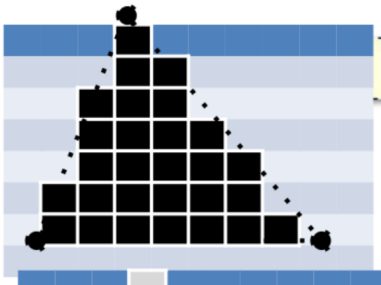
computação
Universidade Federal de Pelotas

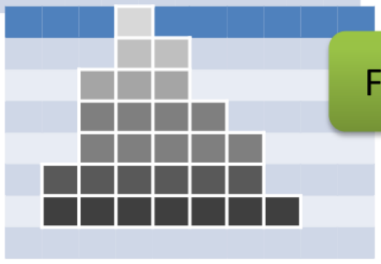
Geometric Primitive



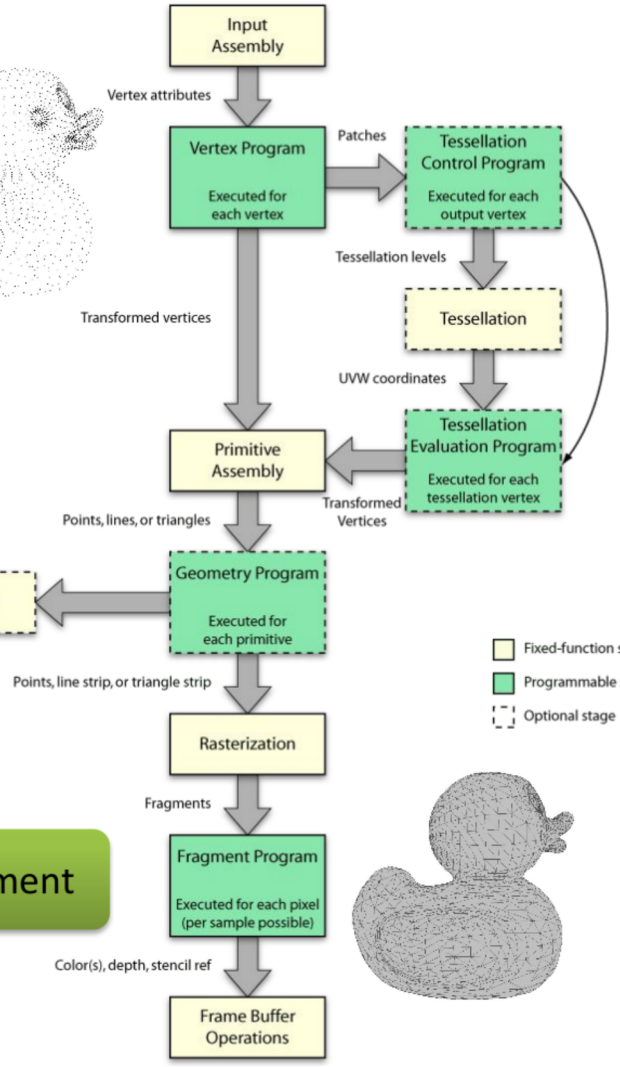
One thread for each vertex!

Raster Pipeline

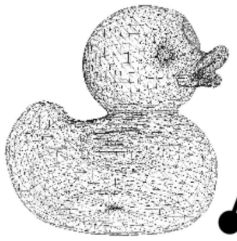


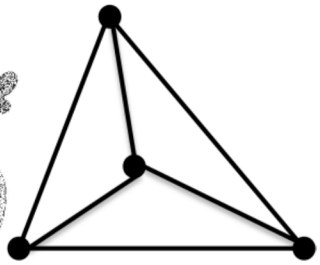


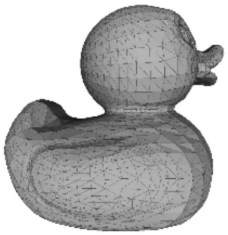
Fragment

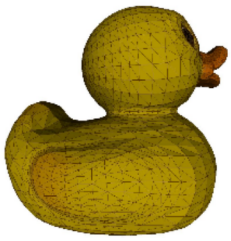


Geometric Primitive









GLSL Data Types

Scalar types: float, int, bool

Vector types: vec2, vec3, vec4
ivec2, ivec3, ivec4
bvec2, bvec3, bvec4

Matrix types: mat2, mat3, mat4

Texture sampling: sampler1D, sampler2D, sampler3D, samplerCube

C++ Style Constructors `vec3 a = vec3(1.0, 2.0, 3.0);`

Operators

computação
Universidade Federal de Pelotas

- **Standard** C/C++ arithmetic and logic operators
- **Operators overloaded** for matrix and vector operations

```
Mat4 m;
Vec4 a, b, c;

b = a*m;
c = m*a;
```

Components and Swizzling

For **vectors** can use [ ], xyzw, rgba or strq

```
vec3 v;
```

`v[1], v.y, v.g, v.t` all refer to the **same element**

Swizzling:

```
vec3 a, b;
a.xy = b.yx;
```

Qualifiers

computação
Universidade Federal de Pelotas

- in, out, inout
  - Copy vertex attributes and other variable to/ from shaders

```
in vec2 tex_coord;
out vec4 color;
```

One for each **primitive**!

- **Uniform**: variable from application

```
uniform float time;
uniform vec4 rotation;
```

One for each **render call**!

Flow Control

- if
- if else
- expression ? true-expression : false-expression
- while, do while
- for

Functions

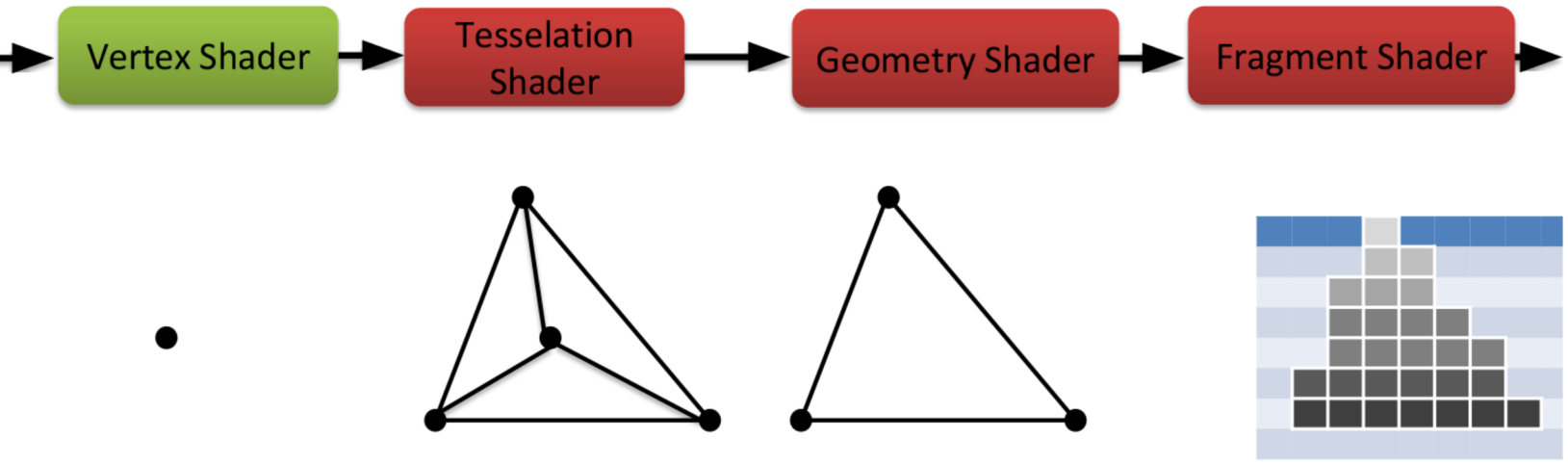
- Built in
  - Arithmetic: sqrt, power, abs
  - Trigonometric: sin, asin
  - Graphical: length, reflect
- User defined

Built-in Variables

- `gl_Position`: output position from vertex shader
- `gl_FragColor`: output color from fragment shader

Let´s define the pipeline

computação
Universidade Federal de Pelotas



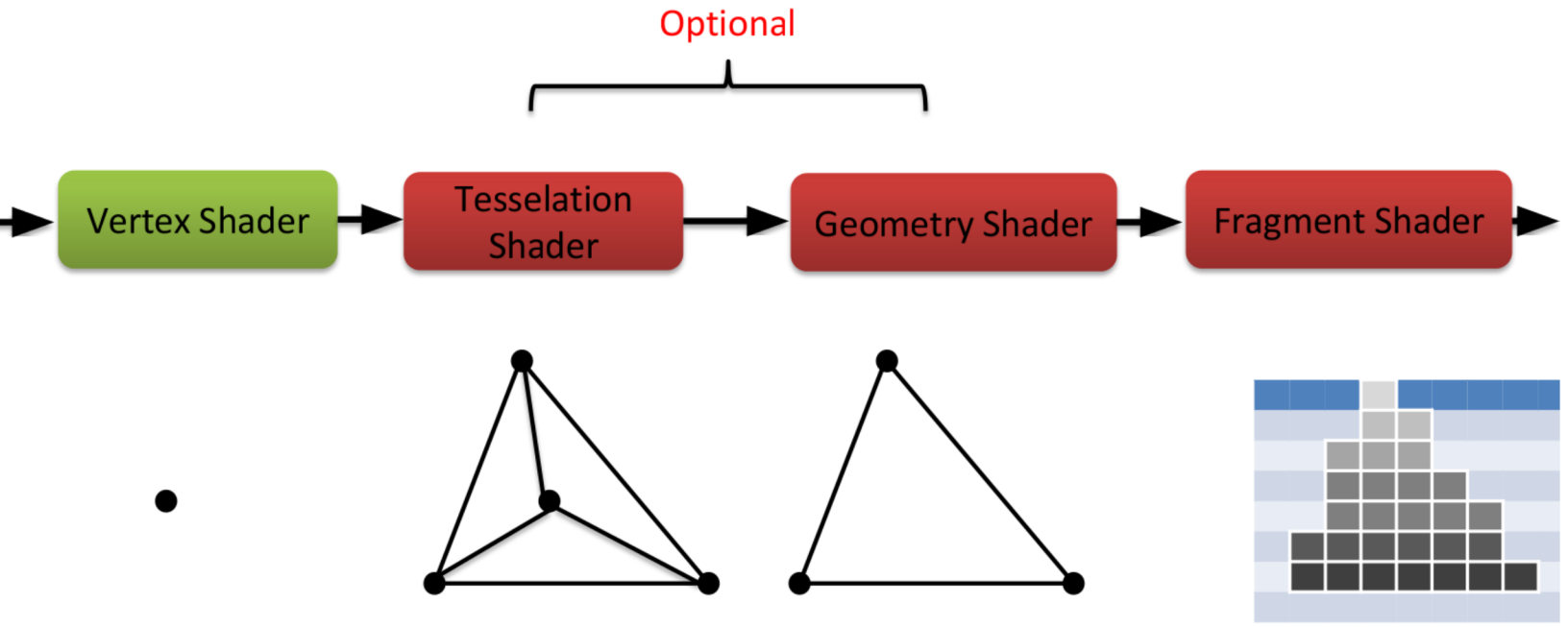
Vertex Shader

computação
Universidade Federal de Pelotas

```
in vec4 vPosition;
in vec4 vColor;
out vec4 color;

void main()
{
    color = vColor;
    gl_Position = vertex;
}
```

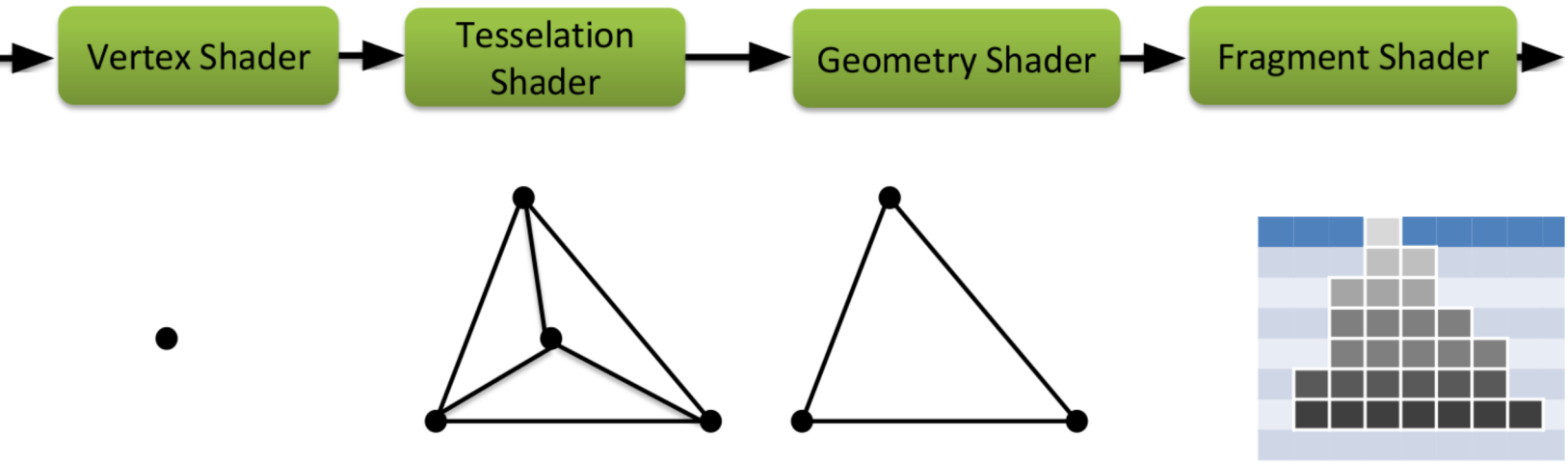
Let´s define the pipeline

computação
Universidade Federal de Pelotas



Let´s define the pipeline

computação
Universidade Federal de Pelotas

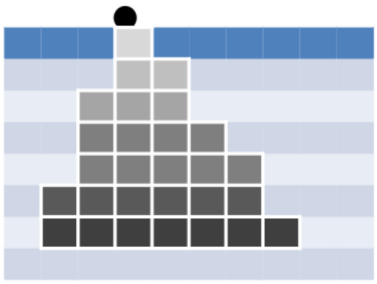


Fragment Shader

computação
Universidade Federal de Pelotas

```
in vec4 color;

void main()
{
    gl_FragColor = color;
}
```

Interpolated!



Pipeline

computação
Universidade Federal de Pelotas

Note that what is **in** is **out** in the next stage

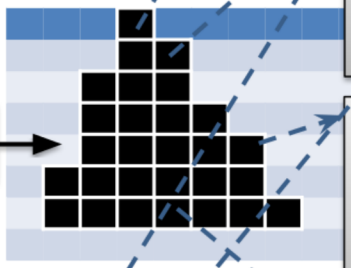
```
in vec4 vPosition;
in vec4 vColor;
out vec4 color;

void main()
{
    color = vColor;
    gl_Position = vertex;
}
```

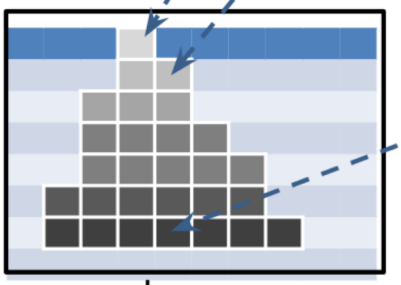
```
in vec4 vPosition;
in vec4 vColor;
out vec4 color;

void main()
{
    color = vColor;
    gl_Position = vertex;
}
```

```
in vec4 vPosition;
in vec4 vColor;
out vec4 color;

void main()
{
    color = vColor;
    gl_Position = vertex;
}
```

...
N

Raster





Image

```
in vec4 color;

void main()
{
    gl_FragColor = color;
}
```

```
in vec4 color;

void main()
{
    gl_FragColor = color;
}
```

```
in vec4 color;

void main()
{
    gl_FragColor = color;
}
```

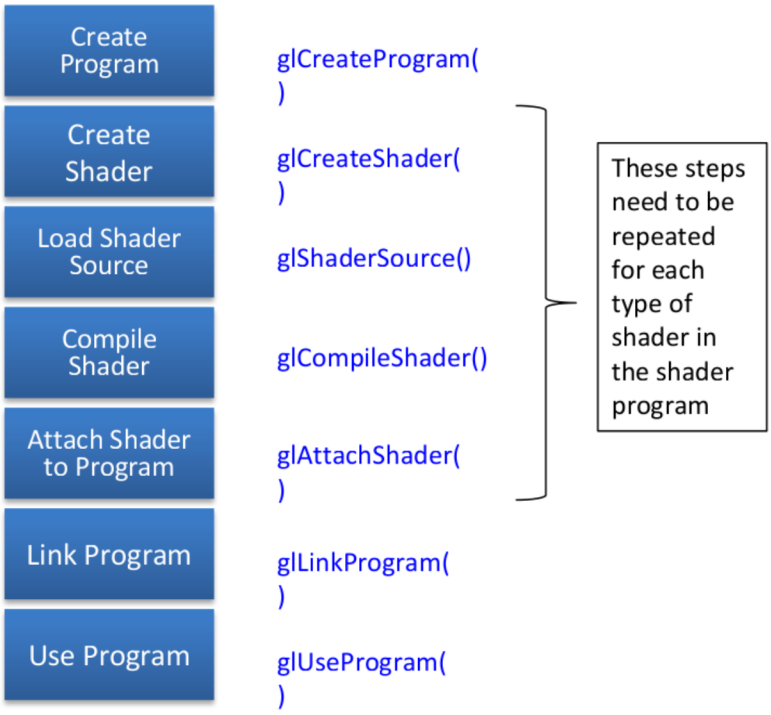
```
in vec4 color;

void main()
{
    gl_FragColor = color;
}
```

...
N

Getting Your Shaders into OpenGL

computação
Universidade Federal de Pelotas

- **Shaders** need to be **compiled and linked** to form an executable shader program
- OpenGL **provides** the compiler and linker
- A program must contain
  - vertex and fragment shaders
  - other shaders are optional



Associating Shader Variables and Data

- Need to **associate** a shader **variable** with an OpenGL data source
  - vertex shader attributes → app vertex attributes
  - shader uniforms → app provided uniform values
- OpenGL relates shader **variables** to **indices** for the app to set
- Two methods for determining variable/index association
  - specify association **before** program linkage
  - **query** association after program linkage

Determining Locations After Linking

Assumes you already know the variables name

```
GLint  idx = glGetAttribLocation( program, "name" );

GLint  idx = glGetUniformLocation( program, "name" );
```

Uniform Qualified

computação
Universidade Federal de Pelotas

- Variables that are **constant** for an **entire** primitive
- Can be changed in application and sent to shaders
- **Cannot be changed in shader**
- Used to pass information to shader such as the bounding box of a primitive

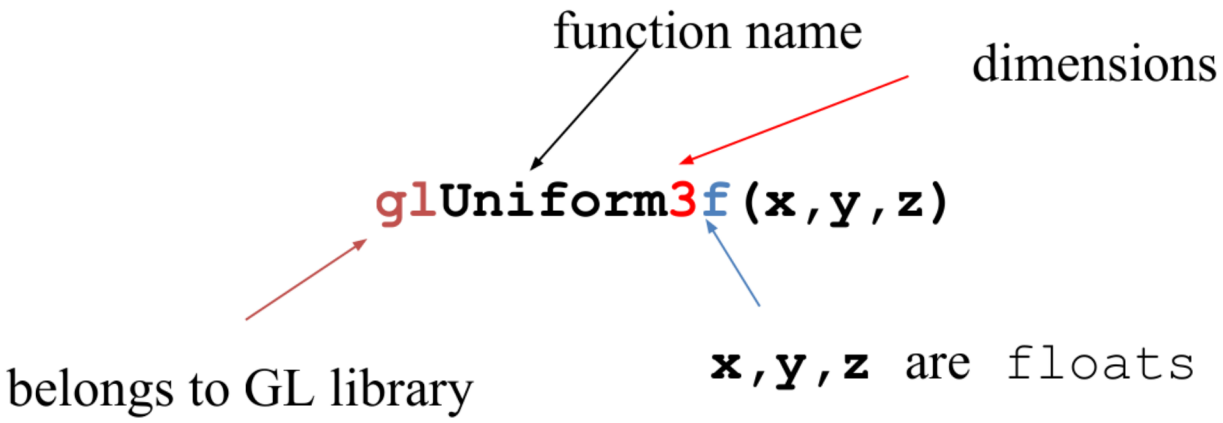
Initializing Uniform Variable Values

- Uniform Variables

```
glUniform4f( index, x, y, z, w );
```

```
GLboolean  transpose = GL_TRUE;
GLfloat  mat[3][4][4] = { … };

glUniformMatrix4fv( index, 3, transpose, mat );
```

OpenGL function format




```
glUniform3fv(p)
```

p is a pointer to an array

Entering the pipeline

computação
Universidade Federal de Pelotas

Overview

- Put geometric data in an array

```
vec3 points[3];
points[0] = vec3(0.0, 0.0, 0.0);
points[1] = vec3(0.0, 1.0, 0.0);
points[2] = vec3(0.0, 0.0, 1.0);
```

- Send array to GPU
- Tell GPU to render as triangle

Let´s render a cube

computação
Universidade Federal de Pelotas

- We'll **render a cube** with colors at each vertex
- Our example demonstrates:
  - initializing vertex data
  - organizing data for rendering
  - simple object modeling
    - building up **3D** objects **from geometric primitives**
    - building **geometric** primitives **from vertices**

Initializing the Cube's Data

- We'll build each cube face from individual triangles
- Need to determine how must storage is required
  - (6 faces)(2 triangles/face)(3 vertices/triangle)

```
const int NumVertices = 36;
```

- To simplify communicating with GLSL, we'll use a vec4 class (implemented in C++) similar to GLSL's vec4 type
  - we'll also typedef it to add logical meaning

```
typedef  vec4  point4;
typedef  vec4  color4;
```

Initializing the Cube's Data (cont'd)

- Before we can initialize our Vertex **Buffer** Object (**VBO**), we need to stage the data
- Our cube has two attributes per vertex
  - position
  - color
- We create two arrays to hold the VBO data

```
point4  points[NumVertices];
color4  colors[NumVertices];
```

Cube Data

computação
Universidade Federal de Pelotas

```
// Vertices of a unit cube centered at
  origin, sides aligned with axes
point4 vertex_positions[8] = {
    point4( -0.5, -0.5,  0.5, 1.0 ),
    point4( -0.5,  0.5,  0.5, 1.0 ),
    point4(  0.5,  0.5,  0.5, 1.0 ),
    point4(  0.5, -0.5,  0.5, 1.0 ),
    point4( -0.5, -0.5, -0.5, 1.0 ),
    point4( -0.5,  0.5, -0.5, 1.0 ),
    point4(  0.5,  0.5, -0.5, 1.0 ),
    point4(  0.5, -0.5, -0.5, 1.0 )
};
```

There are triangles here?

Just a point cloud!

Cube Data

computação
Universidade Federal de Pelotas

```
// RGBA colors
color4 vertex_colors[8] = {
    color4( 0.0, 0.0, 0.0, 1.0 ),  // black
    color4( 1.0, 0.0, 0.0, 1.0 ),  // red
    color4( 1.0, 1.0, 0.0, 1.0 ),  // yellow
    color4( 0.0, 1.0, 0.0, 1.0 ),  // green
    color4( 0.0, 0.0, 1.0, 1.0 ),  // blue
    color4( 1.0, 0.0, 1.0, 1.0 ),  // magenta
    color4( 1.0, 1.0, 1.0, 1.0 ),  // white
    color4( 0.0, 1.0, 1.0, 1.0 )   // cyan
};
```

One color of each vertice

Generating a Cube Face from Vertices

computação
Universidade Federal de Pelotas

```
// quad() generates two triangles for each face and assigns colors to the vertices
int Index = 0;  // global variable indexing into VBO arrays

void
quad( int a, int b, int c, int d )
{
    colors[Index] = vertex_colors[a]; points[Index] = vertex_positions[a]; Index++;
    colors[Index] = vertex_colors[b]; points[Index] = vertex_positions[b]; Index++;
    colors[Index] = vertex_colors[c]; points[Index] = vertex_positions[c]; Index++;
    colors[Index] = vertex_colors[a]; points[Index] = vertex_positions[a]; Index++;
    colors[Index] = vertex_colors[c]; points[Index] = vertex_positions[c]; Index++;
    colors[Index] = vertex_colors[d]; points[Index] = vertex_positions[d]; Index++;
}
```

Is the colors and tesselation right? Let´s see the next slide and than we come back

Generating the Cube from Faces


```
// generate 12 triangles: 36 vertices and
  36 colors
void
colorcube()
{
    quad( 1, 0, 3, 2 );
    quad( 2, 3, 7, 6 );
    quad( 3, 0, 4, 7 );
    quad( 6, 5, 1, 2 );
    quad( 4, 5, 6, 7 );
    quad( 5, 4, 0, 1 );
}
```

Vertex Array Objects (VAOs)

- VAOs store the data of an geometric object
- Steps in using a VAO
  1. generate VAO names by calling `glGenVertexArrays()`
  2. bind a specific VAO for initialization by calling `glBindVertexArray()`
  3. update VBOs associated with this VAO
  4. bind VAO for use in rendering
- This approach allows a single function call to specify all the data for an objects

VAOs in Code

computação
Universidade Federal de Pelotas

```
// Create a vertex array object
GLuint vao;
glGenVertexArrays( 1, &vao );
glBindVertexArray( vao );
```

Storing Vertex Attributes

computação
Universidade Federal de Pelotas

- Vertex data must **be stored in a VBO**, and **associated with a VAO**
- The code-flow is similar to configuring a VAO
  1. generate VBO names by calling `glGenBuffers()`
  2. bind a specific VBO for initialization by calling `glBindBuffer( GL_ARRAY_BUFFER, … )`
  3. load data into VBO using `glBufferData( GL_ARRAY_BUFFER, … )`
  4. bind VAO for use in rendering `glBindVertexArray()`

VBOs in Code


```
// Create and initialize a buffer object
GLuint buffer;
glGenBuffers( 1, &buffer );
glBindBuffer( GL_ARRAY_BUFFER, buffer );
glBufferData( GL_ARRAY_BUFFER, sizeof(points) +
              sizeof(colors), NULL, GL_STATIC_DRAW );
glBufferSubData( GL_ARRAY_BUFFER, 0,
                 sizeof(points), points );
glBufferSubData( GL_ARRAY_BUFFER, sizeof(points),
                 sizeof(colors), colors );
```

Connecting Vertex Shaders with Geometric Data

- Application **vertex** data **enters** the OpenGL **pipeline** through the vertex shader
- Need to connect vertex data to shader variables
  - requires knowing the attribute location
- Attribute location can either be queried by calling `glGetVertexAttribLocation()`

Vertex Array Code


```
// set up vertex arrays (after shaders are loaded)
GLuint vPosition = glGetAttribLocation( program, "vPosition" );
glEnableVertexAttribArray( vPosition );
glVertexAttribPointer( vPosition, 4, GL_FLOAT, GL_FALSE, 0,
                       BUFFER_OFFSET(0) );

GLuint vColor = glGetAttribLocation( program, "vColor" );
glEnableVertexAttribArray( vColor );
glVertexAttribPointer( vColor, 4, GL_FLOAT, GL_FALSE, 0,
                      BUFFER_OFFSET(sizeof(points)) );
```

```
in vec4 vPosition;
in vec4 vColor;
out vec4 color;

void main()
{
    color = vColor;
    gl_Position = vertex;
}
```

Drawing Geometric Primitives

- For contiguous groups of vertices

```
glDrawArrays( GL_TRIANGLES, 0, NumVertices );
```

- Initiates vertex shader

Not the only way to send geometric primitives through the pipeline!

Tutorial

computação
Universidade Federal de Pelotas

- https://learnopengl.com/Getting-started/Hello-Triangle
- https://learnopengl.com/Getting-started/Shaders
- https://learnopengl.com/Getting-started/Transformations
  - See the exercises at the end!

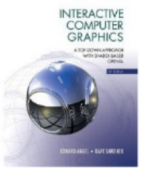
Links

computação
Universidade Federal de Pelotas

- http://www.realtimerendering.com/
- http://www.opengl-tutorial.org/
  - http://www.opengl-tutorial.org/miscellaneous/useful-tools-links/
- http://openglbook.com/

- http://blog.icare3d.org/2011/06/free-3d-meshes-links.html
- http://nehe.gamedev.net/
- http://www.gamedev.net/
- http://arcsynthesis.org/gltut/
- http://ogldev.atspace.co.uk/
- http://devmaster.net/
- http://www.flipcode.com/

- Thanks Ed Angel and Dave Shreiner: http://www.cs.unm.edu/~angel/
- http://www.cs.unm.edu/~angel/BOOK/INTERACTIVE_COMPUTER_GRAPHICS/SIXTH_EDITION/



STL

computação
Universidade Federal de Pelotas

- http://pt.wikipedia.org/wiki/Standard_Template_Library

- Highly recommended!

DevIL

computação
Universidade Federal de Pelotas

- http://openil.sourceforge.net/

Assimp

- http://assimp.sourceforge.net

FMOD

- http://www.fmod.org