ARLAB

ME/CprE/ComS 557

# Computer Graphics and Geometric Modeling

## Introduction to GPU Programming

September 3rd, 2015

Rafael Radkowski

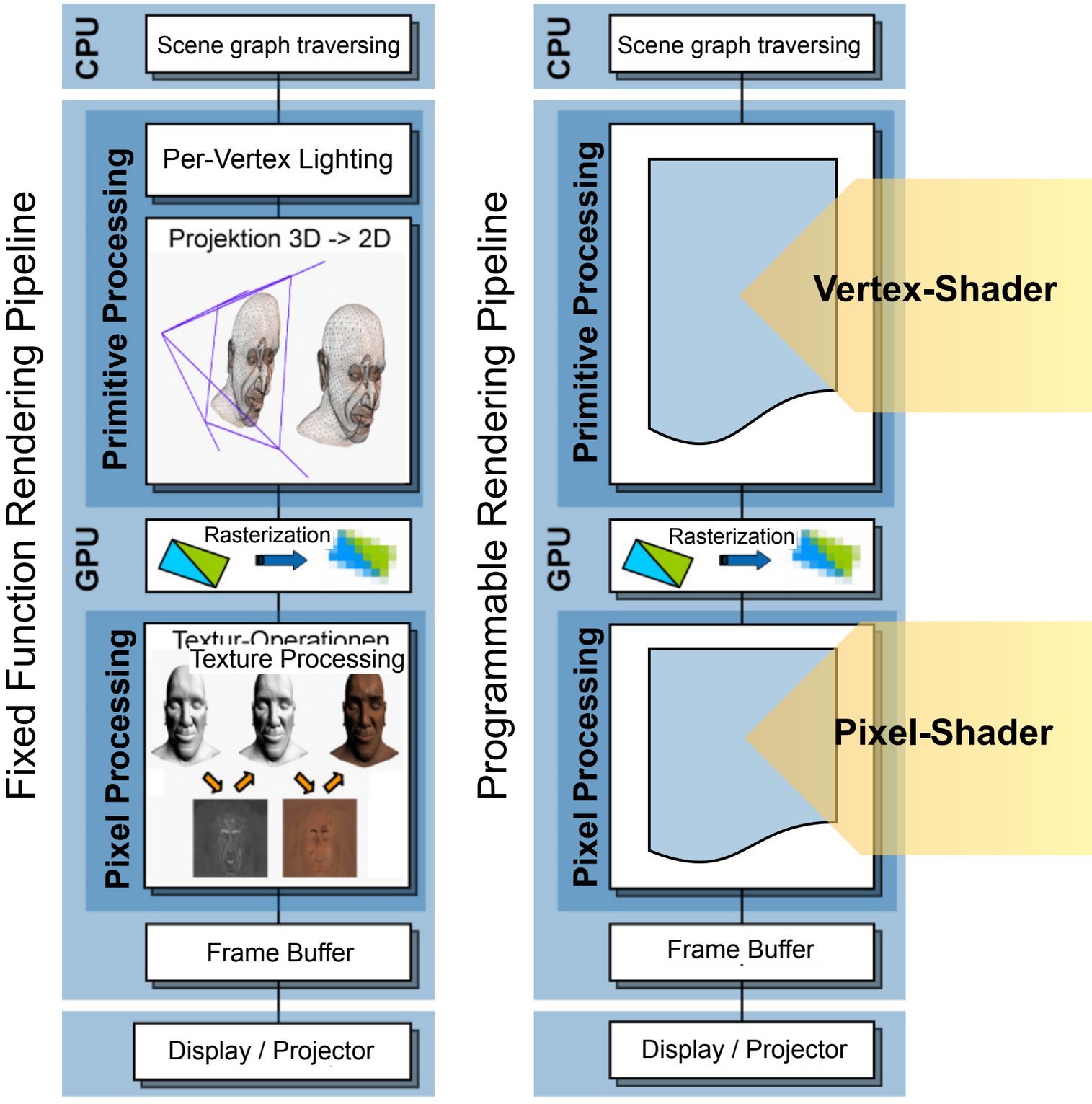
VRAC|HCI

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY

# Topics

- Concepts of GPU-Programming
- Vertex and Pixel (Fragment) Shader
- Functions and Built-in variables
- Shader Compiler and Linker
- Program Structure

# Concept of GPUs



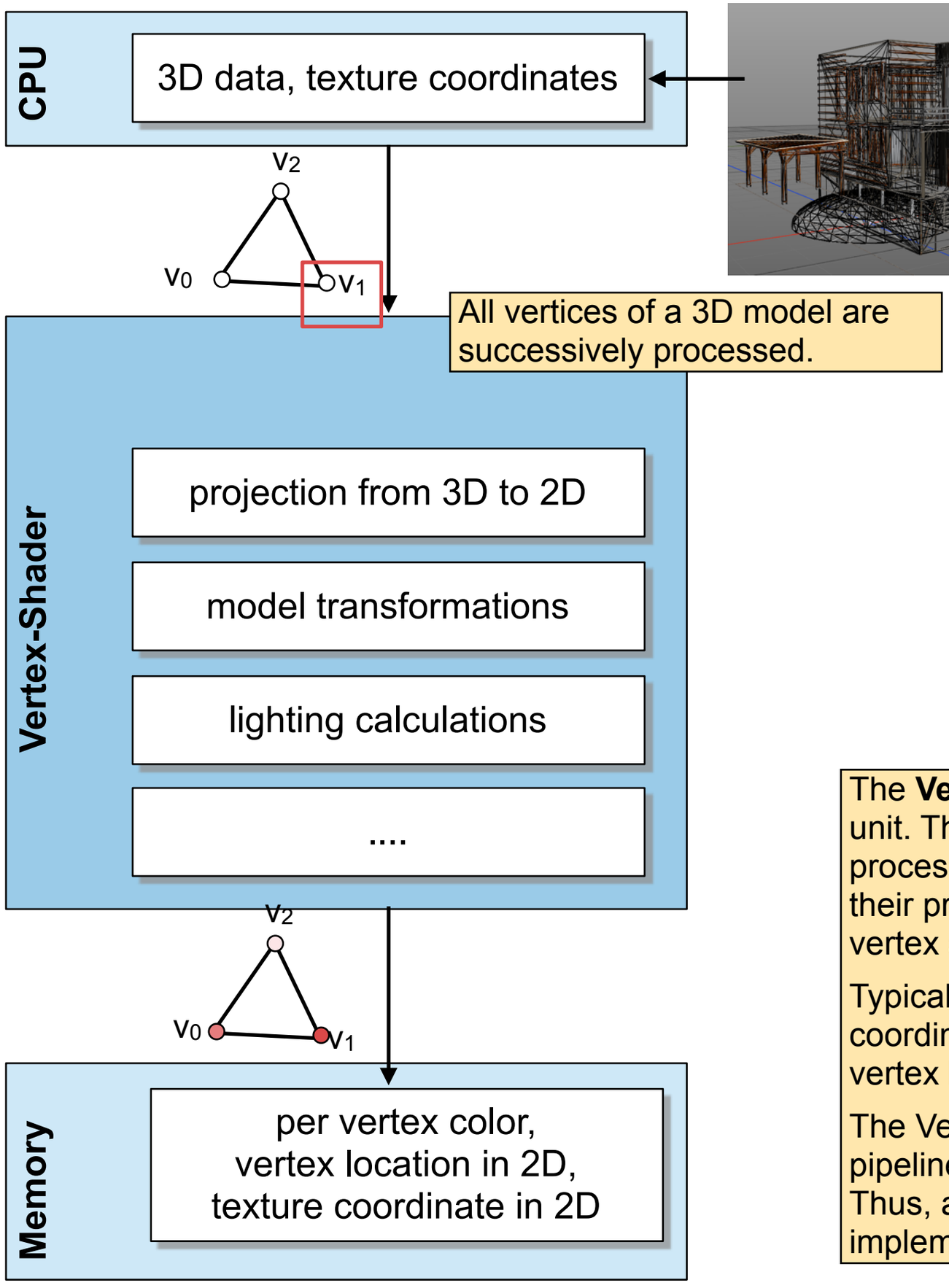

The **fixed function rendering pipeline** implements all necessary calculations (transformation & lighting, rasterization, shading, etc.) via hardware circuits to generate 3D graphic on display.

The **programmable rendering pipeline** uses free-programmable logic processors. Thus, they can be used to implement a vast amount of visual effects which goes beyond the capabilities of the fixed function rendering pipeline. The programmer can decide on his/her own, which function need to be implement to realize a distinct effect.

The **Vertex-Shader** is used to manipulate the vertices of a 3D model and to carry out per-vertex lighting calculations.

The **Pixel (Fragment)-Shader** facilitates the manipulation of single pixels.

The **Geometry-Shader** enables the programmer to create new vertices and



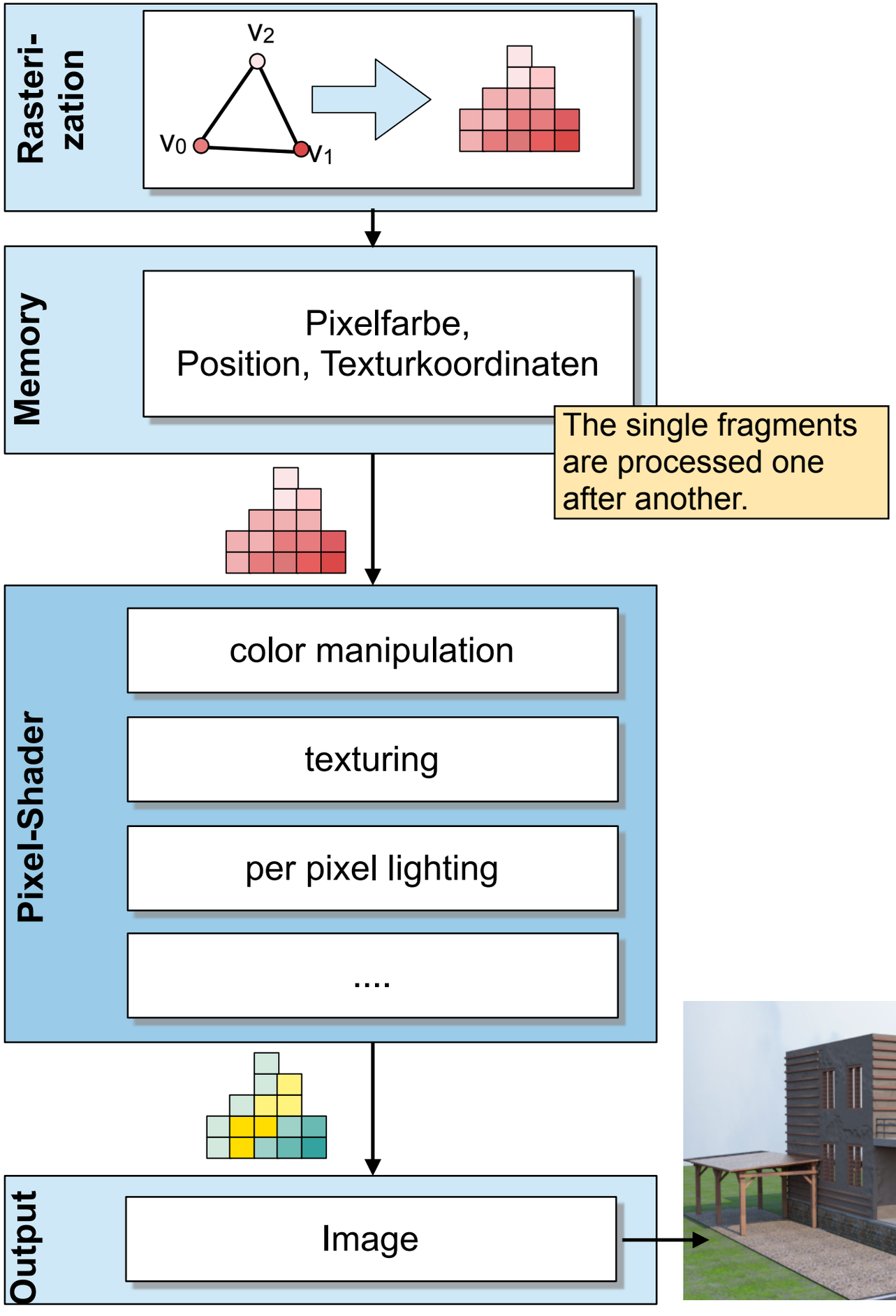



# Vertex-Shader

The **Vertex-Shader** serves as a geometry manipulation unit. The input data are all vertices of a 3D model. The are processed one after another and are combined according their primitives. Thus, a Vertex-Shader processes one vertex at once.

Typical calculations are the projection from 3D to 2D coordinate system, transformation calculations, and per-vertex lighting calculations.

The Vertex-Shader replaces the fixed function rendering pipeline especially the Transformation & Lighting Unit. Thus, all the functions provided by this unit must be implemented manually.

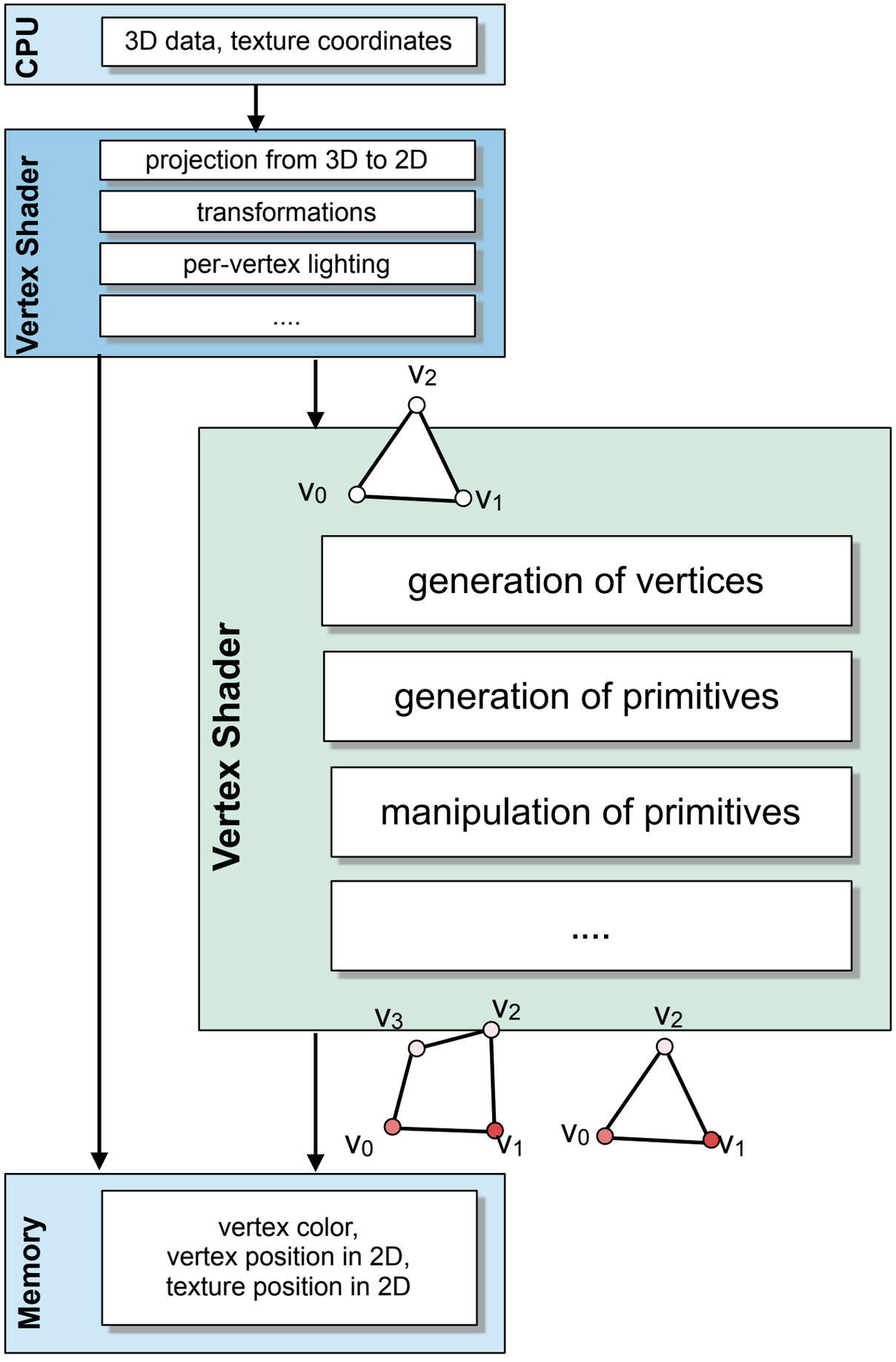



# Pixel (Fragment)-Shader



The **Pixel-Shader** (or Fragment-Shader) is used to manipulate the color data of each primitive's fragment that appears on screen. It replace the **Texture Unit** of the fixed function pipeline. Nevertheless, is can be used for a vast amount of additional functions that go beyond the capabilities of the fixed function pipeline.

VRAC|HCI
IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY




# Geometry-Shader

The **Geometry-Shader** is used for the generation of new vertices and primitives as well as for the manipulation (e.g., Quad to Polygon) of already existing primitives. It is invoked after the Vertex-Shader processing.

Common usage includes i.e., shadow processing and morphing.

Today's realizations are still not performant. Thus, it can only be used to create and manipulate a limited number of vertices and primitives.

# Shader Programming Languages

ARLAB

OpenGL Shading Language

**OpenGL Shanding Language** (GLSL or glSlang) is a system-independent shader programming language for programmable rendering pipelines. It is similar to C and OpenGL.

The latest version is GLSL 4.3

Cg

**C for Graphics (Cg)** is a high-level shader programming language. The language works independently from the underlying graphics subsystem; it supports Direct X as well as OpenGL. It has be published by NVIDIA, nevertheless, it also works on ATI Graphics Boards.

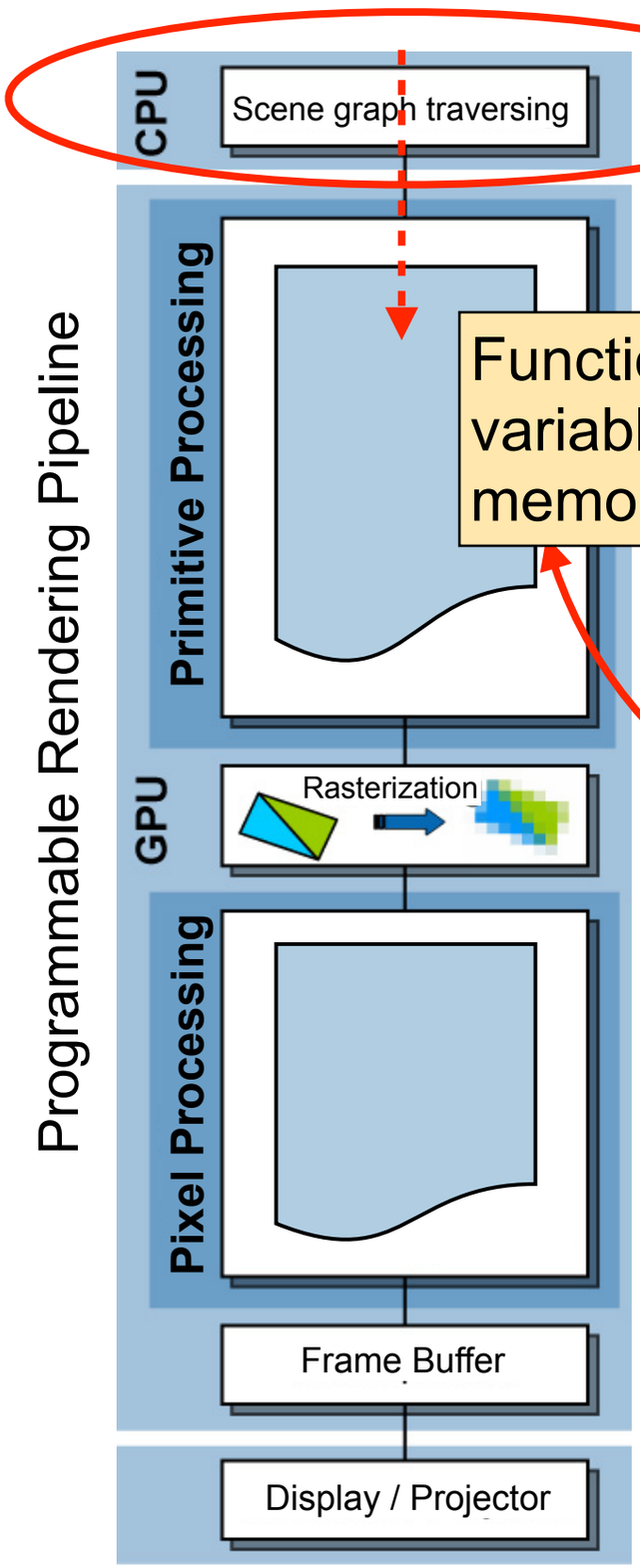
NVIDIA offers the Cg Toolkit for a convenient shader development.

Microsoft® DirectX®

**High Level Shading Language (HLSL)** is a high-level shader language, published by Microsoft. It is a part of DirectX and conceptually integrated into the Microsoft DirectX programming pattern. Thus, it can only be used within DirectX graphics applications.

ARLAB

# The host OpenGL program




Functions to read variables from this memory

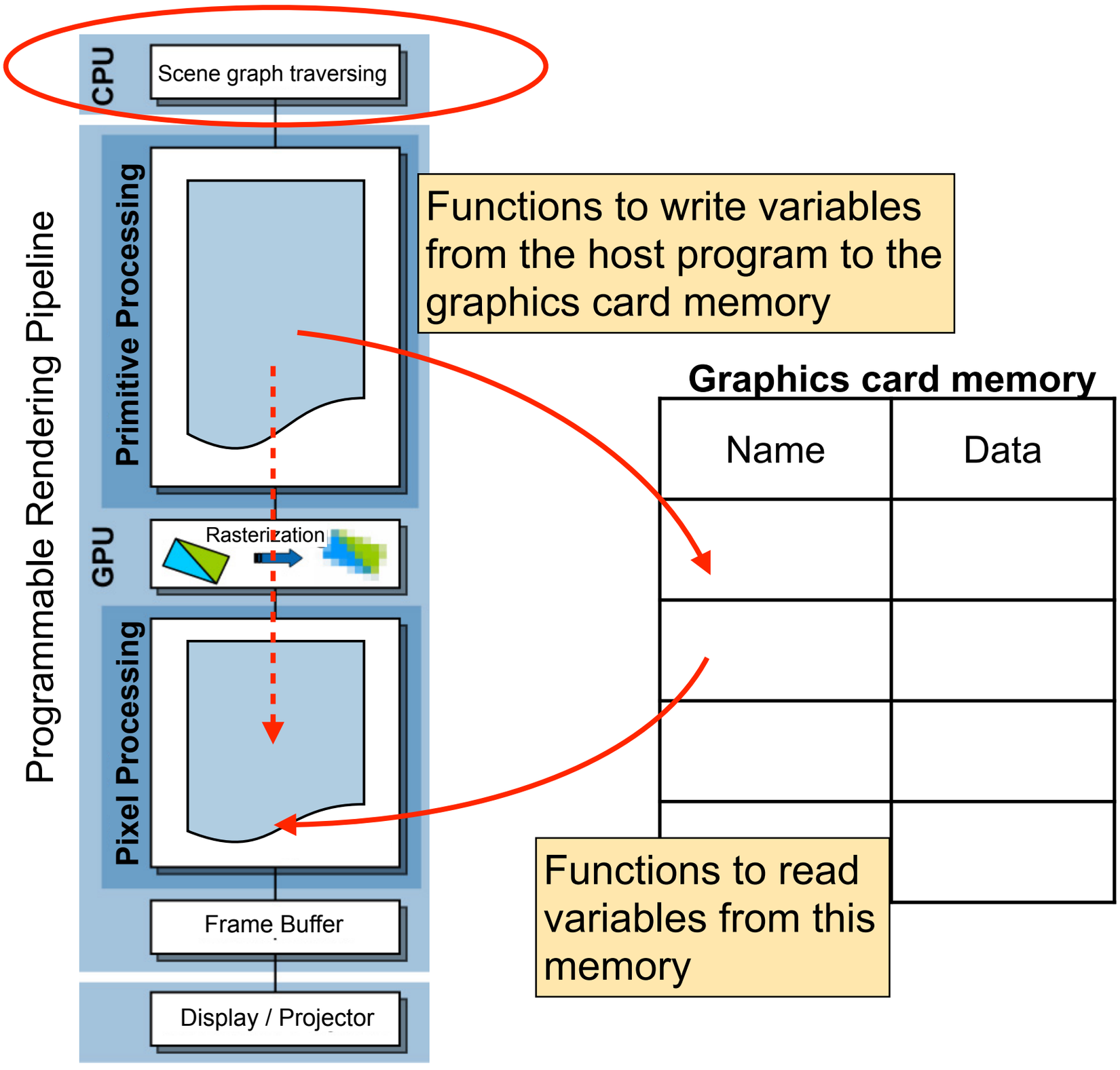
Functions to write variables from the host program to the graphics card memory

**Graphics card memory**

| Name | Data |
|---|---|
| | |
| | |
| | |
| | |

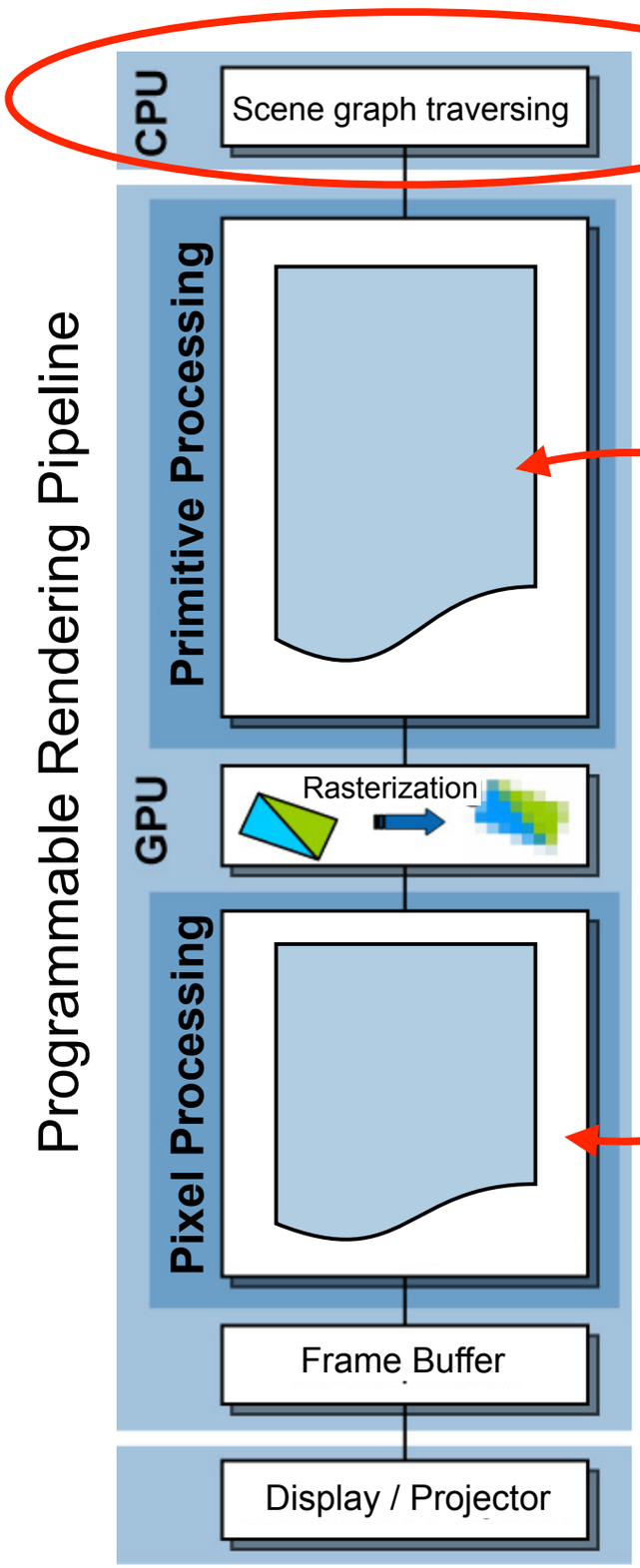
## Types of Functions / Variables

# The host OpenGL program




## Types of Functions / Variables

# The host OpenGL program




**Graphics card memory**

| Name | Data |
|---|---|
| predefined | |
| predefined | |
| predefined | |
| predefined | |

# State Machine

OpenGL uses the computational concept of a state machine:

- The graphics card can be set to a certain state (color, transformation, etc. )
- All functions and parameters that are active in a state are used to modify the vertices.

ARLAB

**There is not one way**
**due to the development in this area!!!!!**

VRAC|HCI

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY

# GL Shader Languange


```
uniform vec4 VariableA;
float VariableB;
vec3  VariableC;
const float ConstantA = 256.0;

float MyFunction(vec4 ArgumentA)
{
    float FunctionsVariableA = float(5.0);

    return float(ArgumentA * (FunctionsVariableA + ConstantA));
}



// I am a comment
/* Me too */
void main(void)
{
    gl_Position     = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_TexCoord[0]  = gl_MultiTexCoord0;
}
```

Declaration of a variable

A function definition looks similar to C/ C++ function definitions.

The entry-point for every vertex and fragement shader program is the function called main.

All variables and functions starting with **gl_*** are so-called built-in variables and functions. Using this variables, a programmer gains access to the data of the rendering pipeline.

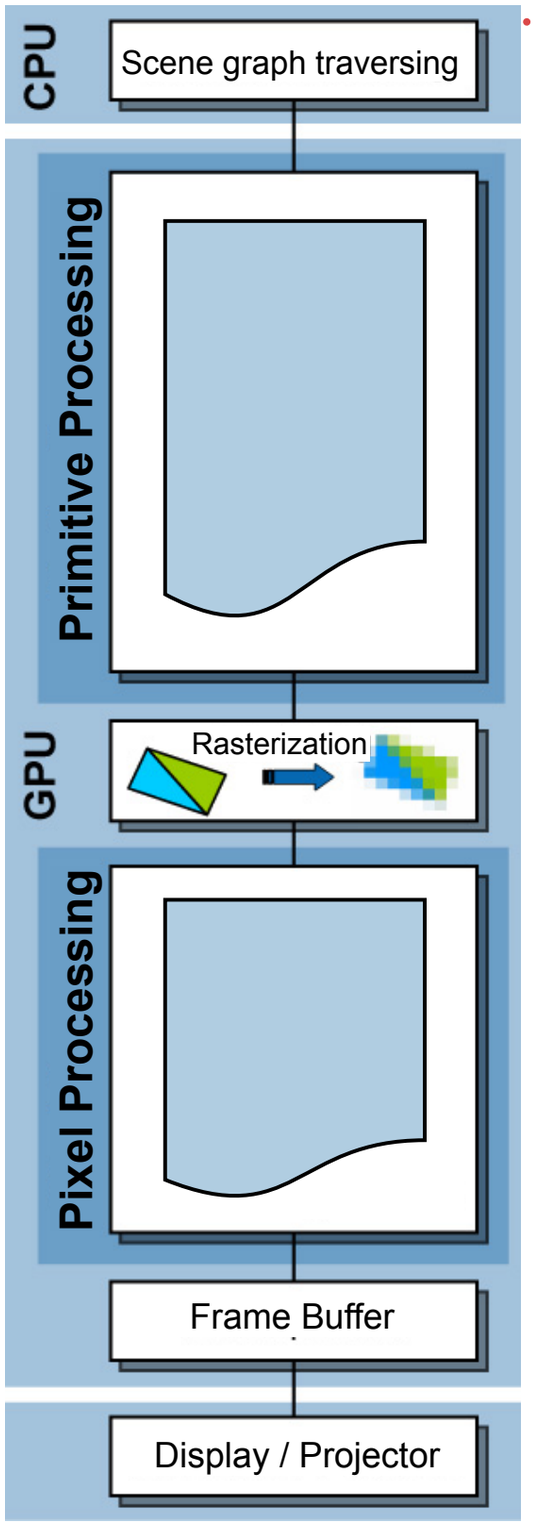
# Data Types

ARLAB

| Datentyp | Erklärung |
|---|---|
| void | For functions, which do not return a value. |
| bool | conditional type that can be true or false. |
| int | signed integer |
| float | floating point value with single precision (32 Bit) |
| vec2 | 2-components floating point vector |
| vec3 | 3 -components floating point vector |
| vec4 | 4 -components floating point vector |
| bvec2 | 2-components boolean vector |
| bvec3 | 3 -components boolean vector |
| bvec4 | 4 -components boolean vector |
| ivec2 | 2 -components integer vector |
| ivec3 | 3 -components integer vector |
| ivec4 | 4 -components integer vector |
| mat2 | 2x2 floating point value matrix |
| mat3 | 3x3 floating point value matrix |
| mat4 | 4x4 floating point value matrix |

| Datentyp | Erklärung |
|---|---|
| sampler1D | A 1D data array (a 1D texture). |
| sampler2D | A 2D data array (a 2D texture). |
| sampler3D | A 3D data array (a 3D texture). |
| samplerCube | A cube map texture. |
| sampler2DRect | Texture which size is not a power of 2 (2^n * 2^n) |
| sampler1DShadow | A 1D data array (a 1D texture) with an additional comparison operator for shadow textures. |
| sampler2DShadow | A 2D data array (a 2D texture) with an additional comparison operator for shadow textures. |
| samplerCubeShadow | A cube map texture with an additional comparison operator for shadow textures (e.g., for omni-directional |
| sampler2DRectShadow | A 2D shadow texture for 2D-non-power-of-two (NPOT)-textures |
| sampler1DArray | An array of 1D textures |
| sampler2DArray | An array of 2D textures |
| sampler1DArrayShadow | A array of 1D data (a 1D texture) with an additional comparison operator for shadow textures. |
| sampler2DArrayShadow | A array of 2D data (a 2D texture) with an additional comparison operator for shadow textures. |
| samplerBuffer | An 1D temporary buffer that can be used as buffer storage |
| sampler2DMS | A 2D data array (a 2D texture) eligible for multi texturing. |
| sampler2DMSArray | An arran of 2D data arrays (a 2D textures) eligible for multi texturing. |

# Type Qualifiers (1/2)
# uniform and varying




The data type **uniform** allows to pass data from a 3D application to the vertex and fragment shader program.

```
uniform vec4 VariableA;
varying vec3  VariableB;
const float ConstantA = 256.0;

/* Vertex Program */
void main(void)
{
   VariableB = vec3(12.0, 13.0, 14.0);
   gl_Position     = gl_ModelViewProjectionMatrix * gl_Vertex;
   gl_TexCoord[0]  = gl_MultiTexCoord0 + VariableA;
}
```

The data type **varying** allows push data from vertex shader to the fragment shader program.

```
varying vec3  VariableB;
uniform vec4 VariableA;
const float ConstantB = 63.0;

/* Fragment Program */
void main(void)
{
   gl_FragColor  = gl_FrontLightModelProduct.sceneColor;
}
```

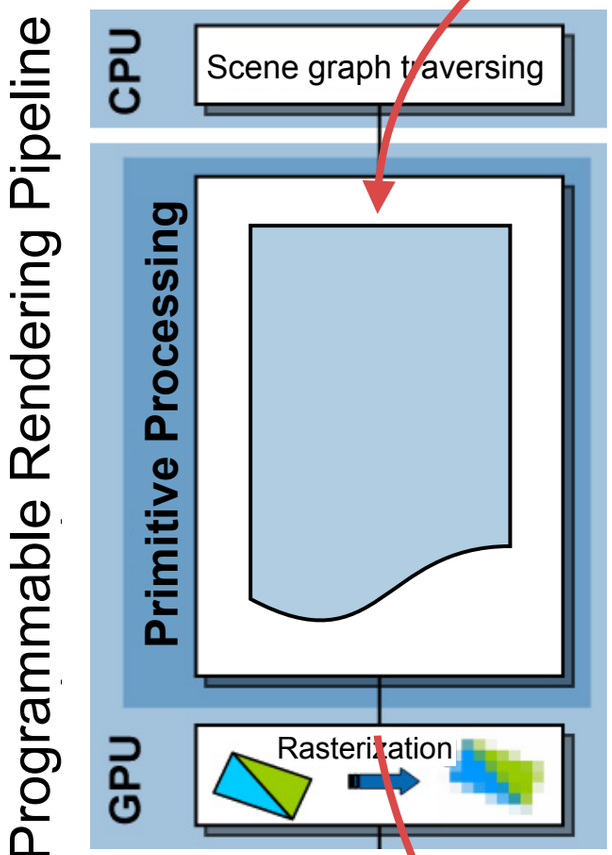
# Type Qualifiers (1/2)
# uniform and varying

ARLAB

In addition to a variable's type, it can bear a Type Qualifier that describes the accessibility of the data.

- **const**
  Fixed (read-only) constant values, available only in the program in which the variable is declared.
- **uniform**
  A variable that poses an interface between the graphics application and the shader program. It can be written and altered within the graphics application and passed to the shader program. The value of the variable can only be changed once at each rendering pass.
- **attribute**
  Read-only, and built-in data that grants access to data of the fixed function rendering pipeline (vertices, color, depth). They pose a pre-defined interface between the graphics application and the shader program.
- **varying**
  Data of type varying are for data exchange between vertex shader program and fragment shader program. They are writeable inside the vertex shader program and read-only in the fragment shader program.
- **in**
  Function variable: an input variable for a distinct function.
- **out**
  Function variable: an output variable for a distinct function.
- **inout**
  Function variable: an input and output variable for a distinct function. It works like a C++ pointer or reference.

# Built-in variables and attributes (1/2)




The rendering pipeline automatically provides these attributes; they are only readable; write access is not permitted.

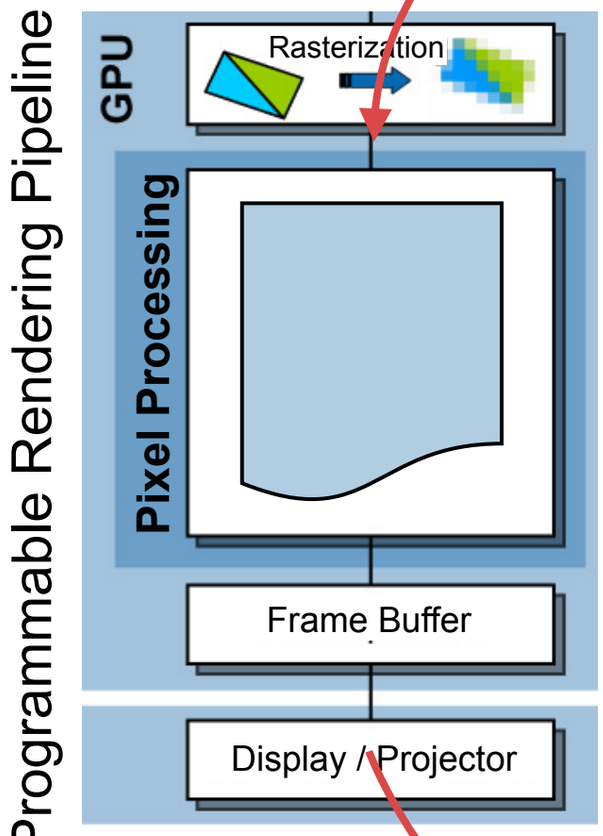
- vec4 gl_Color: Vertex color of this vertex, if provided.
- vec4 gl_SecondaryColor: Secondary certex color of this vertex, if provided.
- vec4 gl_Normal: Vertex normal vector
- vec4 gl_Vertex: Position of the vertex in 3D object coordinate system.
- float gl_FogCoord Fog coordinates

```
uniform vec4 VariableA;
varying vec3  VariableB;
const float ConstantA = 256.0;

/* Vertex Program */
void main(void)
{
   VariableB = vec3(12.0, 13.0, 14.0);
    gl_Position     = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_TexCoord[0]  = gl_MultiTexCoord0;
}
```

- **vec4 gl_Position** must be provided by the programmer.
  This variable stores the position of a vertex in 3D space. The subsequent rasterization unit requires this value to be able to determine the fragments that are covered by a distinct primitive.
- **vec3 gl_TexCoord[x]**
  This variable stores the texture coordinates associated with a distinct vertex. They must be provided if textures available. x is the texture unit.

# Built-in variables and attributes (2/2)

The input values of a fragment shader program are all variable that have been defined in the vertex shader program as well as built-in variables of the graphics pipeline.




```
varying vec3  VariableB;
uniform vec4 VariableA;
const float ConstantB = 63.0;

/* Fragment Program */
void main(void)
{
    gl_FragColor  = gl_FrontLightModelProduct.sceneColor;
}
```

The following out variables are built-in variables; the pose the output data of the fragment shader program:

- **vec4 gl_FragColor**
  The variable bears the output color of each fragment. The fragment will disappear on screen, if this variable is not specified.
- **vec4 gl_FragData[0..15]**
  Replaces the gl_FragColor variable, if multipass-rendering is used.
- **float gl_FragDepth**
  The variable stores the depth value that is associated with the FragColor of the current primitive.

At least, a shader program writer must provide a gl_FragColor to get an output on screen.

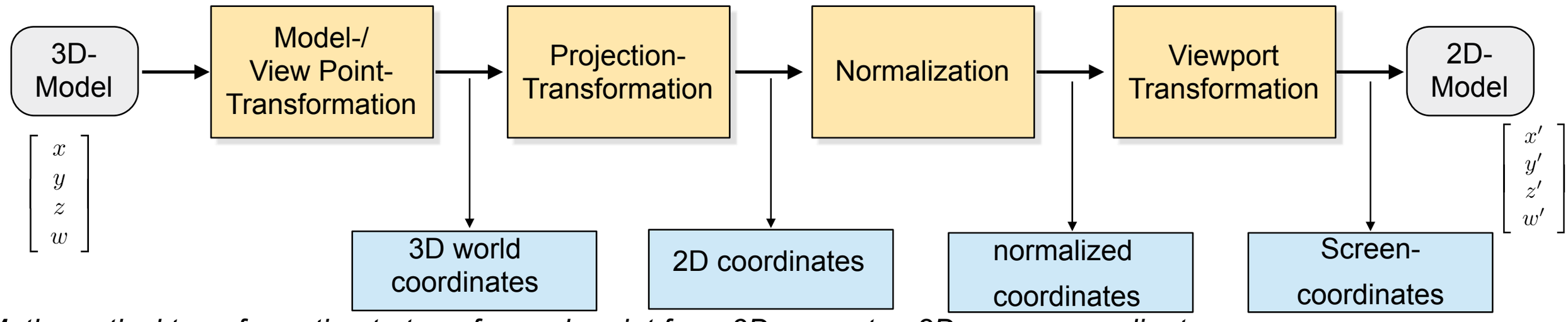
# Built-in varyings

All variables of type varying pose a interface between a vertex shader program and a fragment shader program. They are writeable inside a vertex shader and readable inside the fragment shader. The following varyings are built-in varyings provided by OpenGL and GLSL. The can be accessed without being declared manually.

- **vec4 gl_FrontColor, gl_Color**
  Color of the vertex's front
- **vec4 gl_BackColor**
  Color of the vertex's back
- **vec4 gl_FrontSecondaryColor, gl_SecondaryColor**
  Secondary vertex front color
- **vec4 gl_BackSecondaryColor**
  Secondary vertex back color
- **vec4 gl_TexCoord[x]**
  Texture coordinates of the vertex that are stored in texture unit **x**.
- **float gl_FogFragCoord**
  Fog coordinate of that vertex.

Within the fragment shader program, the varying variables `gl_FrontColor`, `gl_FrontSecondaryColor`, `gl_BackColor`, and `gl_BackSecondaryColor` are only accessible via the alias **gl_Color** and **gl_SecondaryColor**, due to the absence of a back color on screen.

# Built-in uniforms




*Mathematical transformation to transfer each point from 3D space too 2D screen coordinates.*

GLSL grants access to typical OpenGL variables commonly used in OpenGL graphics applications.

- **mat4 gl_ModelViewMatrix**
  The model-view-matrix, which describes the transformation from local to global model space.
- **mat4 gl_ProjectionMatrix**
  The projection matrix that poses the projection of the virtual camera
- **mat4 gl_ModelViewProjectionMatrix**
  The product of model-view matrix and projection matrix.
- **mat3 gl_NormalMatrix**
  The normalized matrix, which scales all vertices of a 3D model to a range of [-1,1] and [0,1].
- **mat4 gl_TextureMatrix[gl_MaxTextureCoordsARB]**
  Textures int the texture memory.
- **float gl_NormalScale**
  The OpenGL factor that describes the scaling of surface normal vectors. It helps to determine the scale factor of a 3D model.

# Example

ARLAB

## Vertex-Shader Program

```
uniform vec4 GlobalColor;


void main(void)
{
    gl_Position     = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_FrontColor   = gl_Color * GlobalColor;
    gl_TexCoord[0]  = gl_MultiTexCoord0;
}
```

This example shows the most smallest programs necessary to get an output on screen.

## Fragment Shader-Program

```
uniform sampler2D Texture0;


void main(void)
{
    vec2 TexCoord = vec2( gl_TexCoord[0] );
    vec4 RGB      = texture2D( Texture0, TexCoord );


    gl_FragColor = RGB + gl_Color;
}
```

ARLAB

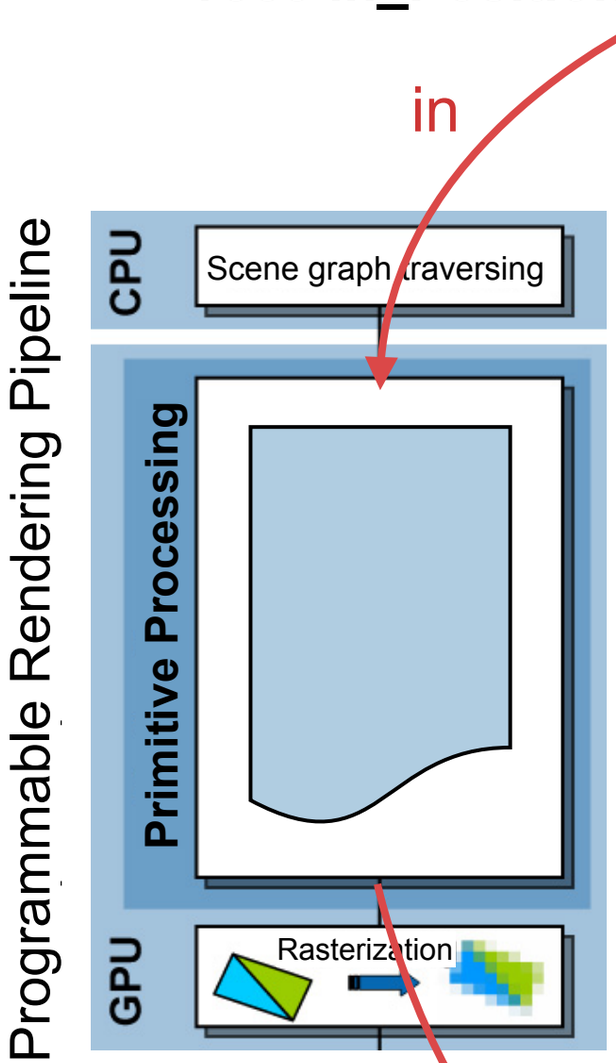
# There is a different way !!

VRAC|HCI

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY

# Type Qualifier in, out, inout

The type qualifiers **in**, **out**, **and** inout allow us to set the set the direction of variables and values.



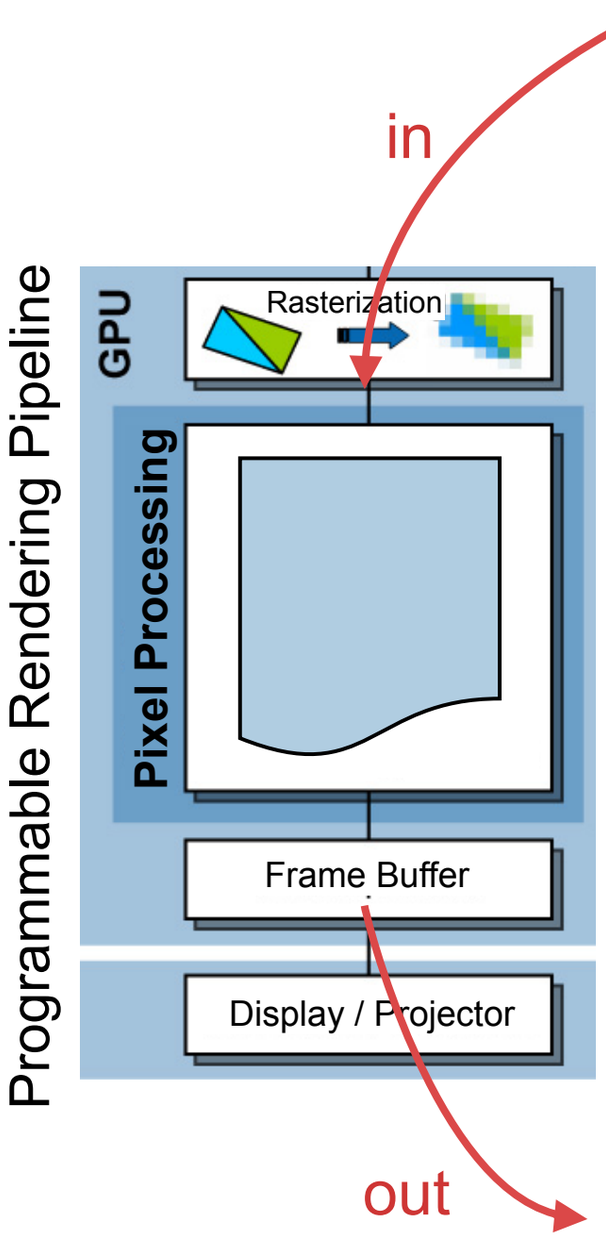

```
#version 410 core

uniform mat4 projectionMatrix;
uniform mat4 viewMatrix;
uniform mat4 modelMatrix;

in vec3 in_Position;

in vec3 in_Color;
out vec3 pass_Color;

void main(void)
{
    gl_Position = projectionMatrix * viewMatrix * modelMatrix *
                  vec4(in_Position, 1.0);

    pass_Color = in_Color;

}
```

The names must be equal

The programer decides

- which variable type he/she wants to pass into the shader program.
- the name of the variable.

# Type Qualifier in, out, inout




The type qualifiers **in**, **out**, **and** inout allow us to set the set the direction of variables and values.

```
#version 410 core
in vec3 pass_Color;
out vec4 color;

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

The names must be equal.
Here, it must be an "in" qualifier

Here: the compiler notifies that the only color that goes out. Thus, the compiler will find the correct color information.

# GLSL Program Structure and Process

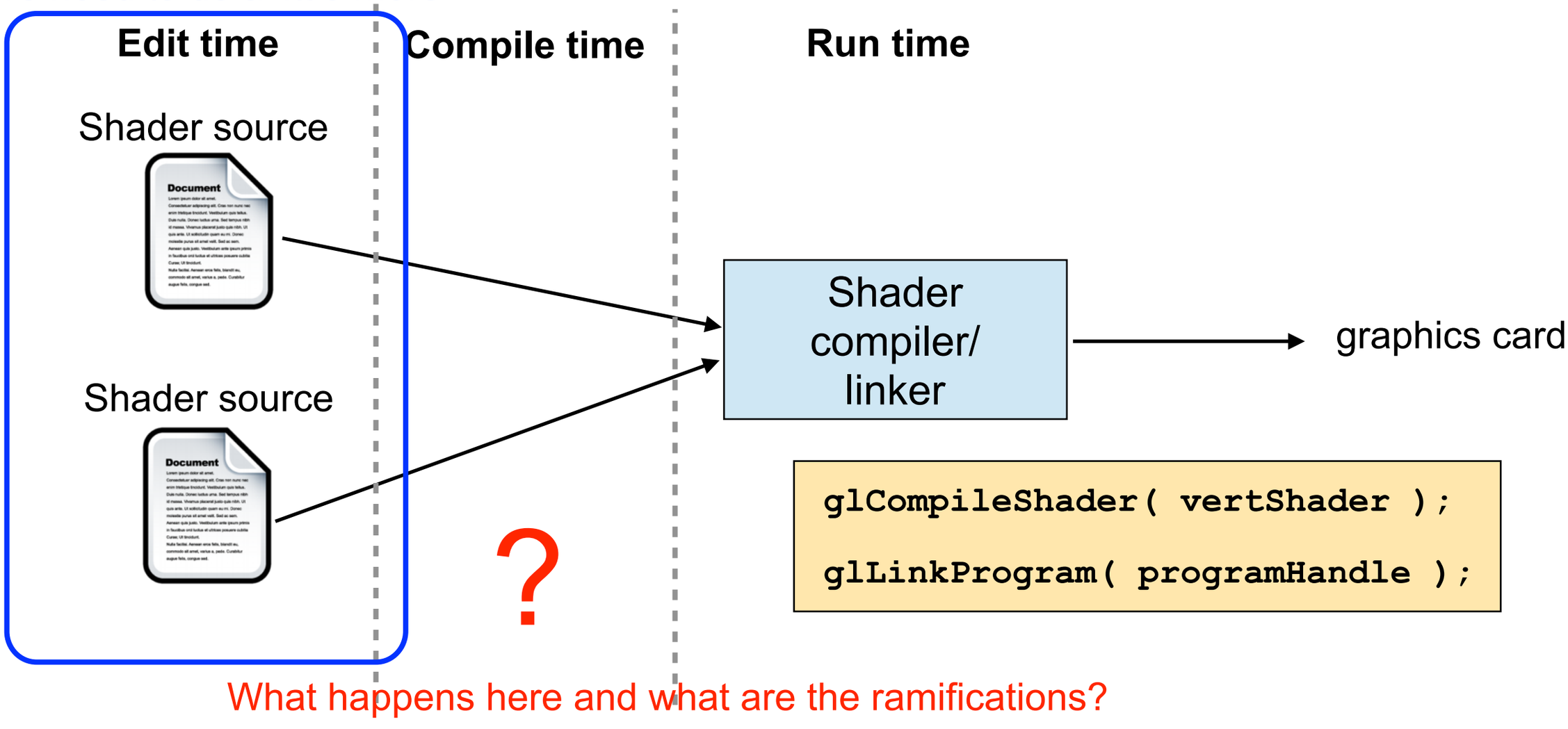
VRAC|HCI
IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY

# GLSL Shader Compiler

The GLSL compiler is built into the OpenGL library, and shaders can only be compiled within the context of a running OpenGL program.

Compiling a shader involves creating a shader object, providing the source code (as a string or set of strings) to the shader object, and asking the shader object to compile the code.

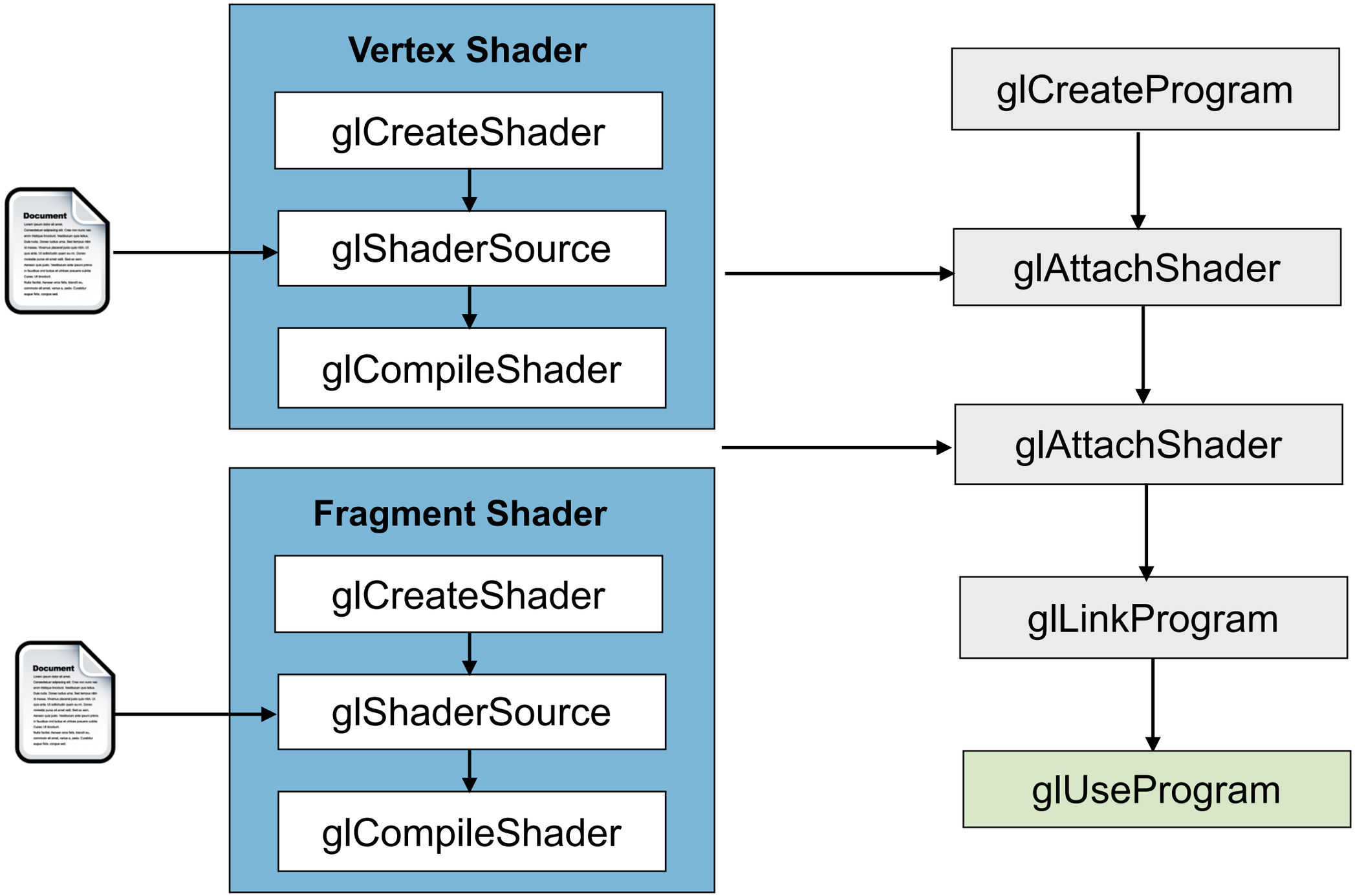
We need to do a little more




What happens here and what are the ramifications?

# Programming Language Process

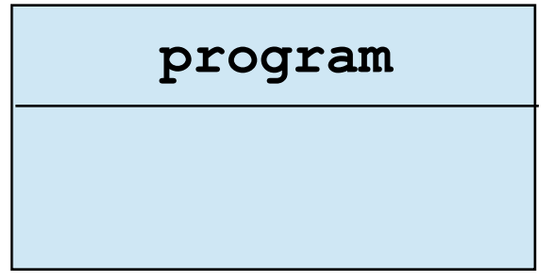
Process to create one shader = one type of appearance.



# Shader Program

**glCreateProgram** — create a program object

A program object is an object to which shader objects can be attached. This provides a mechanism to specify the shader objects that will be linked to create a program.




Example:

```
GLuint program = glCreateProgram();
```

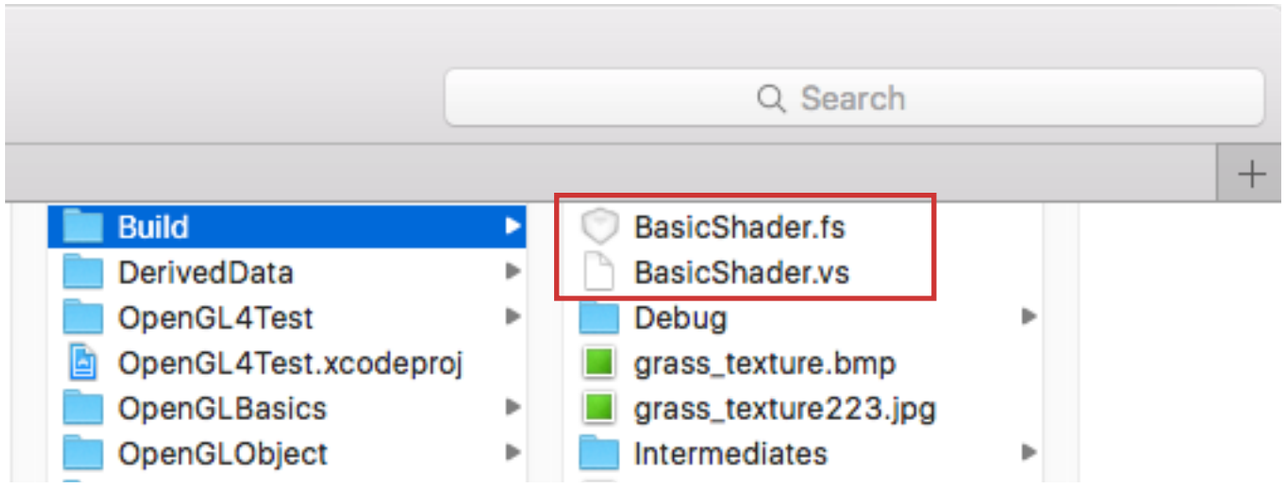
glCreateProgram creates an empty program object and returns a non-zero value by which it can be referenced.

# Shader Code

```
11
12
13
14  static const string vs_string =
15  {
16      "#version 410 core                                                  \n"
17      "                                                                    \n"
18      "uniform mat4 projectionMatrix;                                     \n"
19      "uniform mat4 viewMatrix;                                            \n"
20      "uniform mat4 modelMatrix;                                           \n"
21      "in vec3 in_Position;                                                \n"
22      "                                                                    \n"
23      "in vec3 in_Color;                                                   \n"
24      "out vec3 pass_Color;                                                \n"
25      "                                                                   \n"
26      "void main(void)                                                    \n"
27      "{                                                                  \n"
28      "    gl_Position = projectionMatrix * viewMatrix * modelMatrix * vec4(in_Position, 1.0);  \n"
29      "    pass_Color = in_Color;                                          \n"
30      "}                                                                  \n"
31  };
32
33  // Fragment shader source code. This determines the colors in the fragment generated in the shader p
        our vertex shader.
34  static const string fs_string  =
35  {
36      "#version 410 core                                                  \n"
37      "                                                                   \n"
38      "in vec3 pass_Color;                                                  \n"
39      "out vec4 color;                                                                \n"
40      "void main(void)                                                    \n"
41      "{                                                                  \n"
42      "    color = vec4(pass_Color, 1.0);                                \n"
43      "}                                                                  \n"
44  };
45
```

Shader code can be part of the C++ program.

- Pro: the code is always with your program.
- Con: you need to recompile you code every time when you change the program.




Shader code can be written in separate files.

- Pro: you can change the shader code without recompiling you program
- Con: additional complexity. You have to make sure that you program finds those files and can load them.

*We skip the loading part this time!*

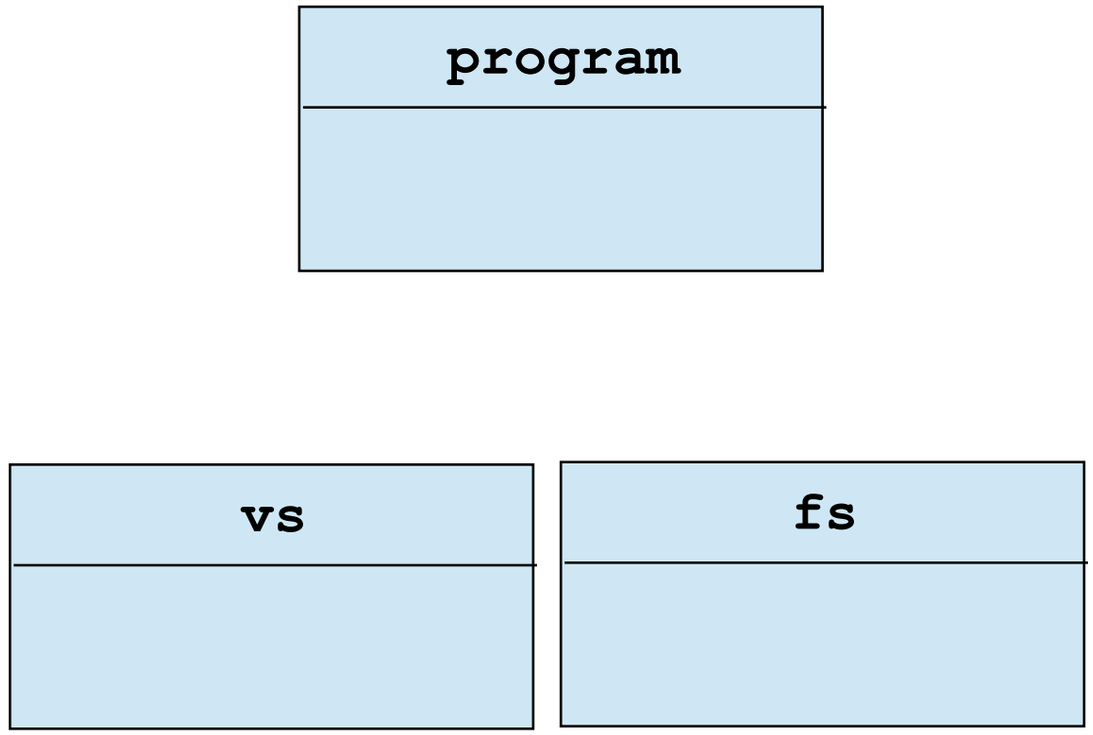
# Shader Object

ARLAB

```
GLuint glCreateShader( GLenum shaderType );
```

A shader object is used to maintain the source code strings that define a shader.

- shaderType: Specifies the type of shader to be created. Must be GL_VERTEX_SHADER or GL_FRAGMENT_SHADER.




Example:

```
GLuint fs = glCreateShader(GL_FRAGMENT_SHADER);

                [.....]

GLuint vs = glCreateShader(GL_VERTEX_SHADER);

                [.....]
```

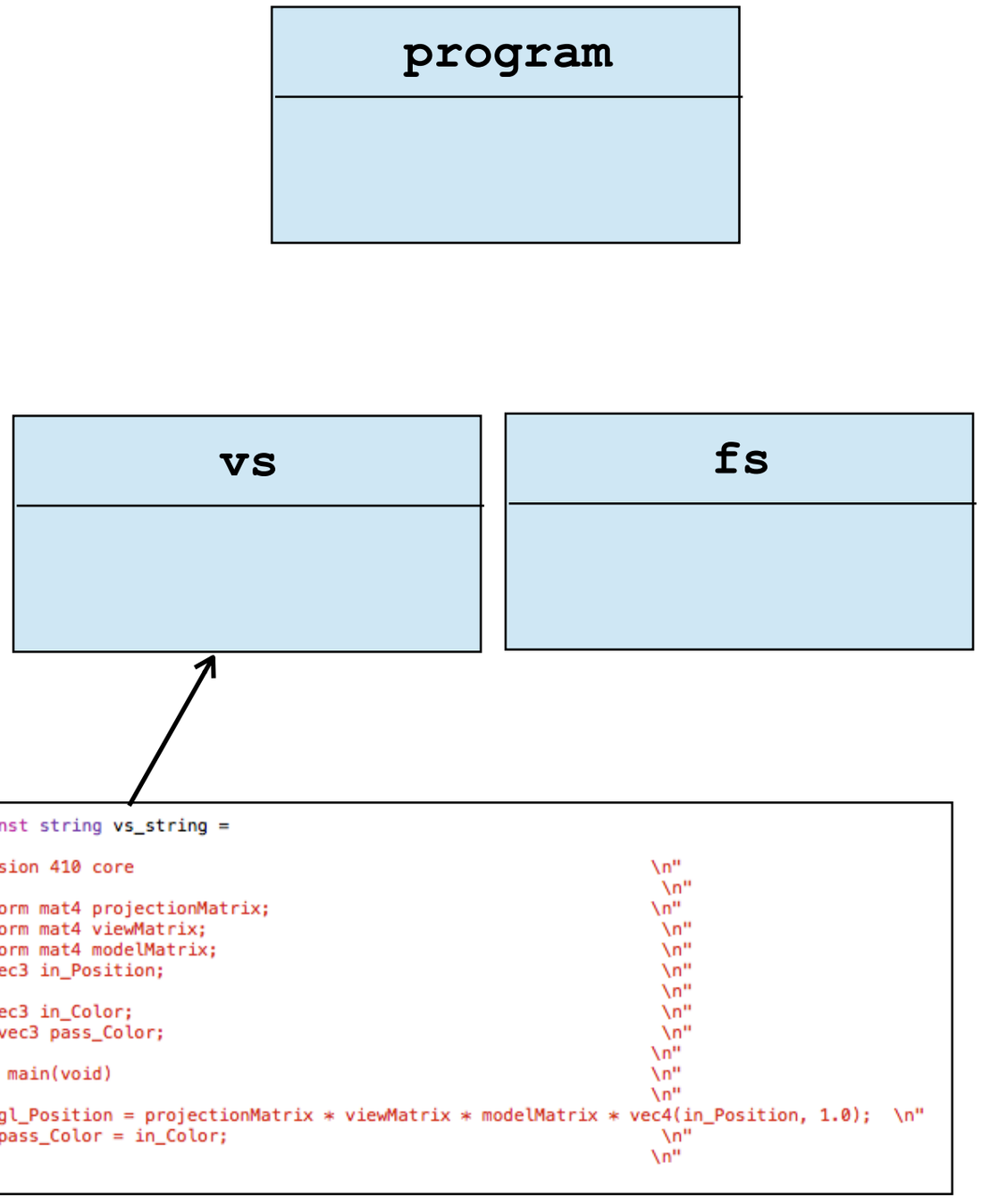
# Add Source Code

```
void glShaderSource(
GLuint shader,
GLsizei count,
const GLchar **string,
const GLint *length);
```

Attach source code to a shader object

Parameters

- shader: Specifies the handle of the shader object whose source code is to be replaced.
- count: Specifies the number of elements in the string and length arrays.
- string: Specifies an array of pointers to strings containing the source code to be loaded into the shader.
- length: Specifies an array of string lengths.




Shader code is a string!

# Example

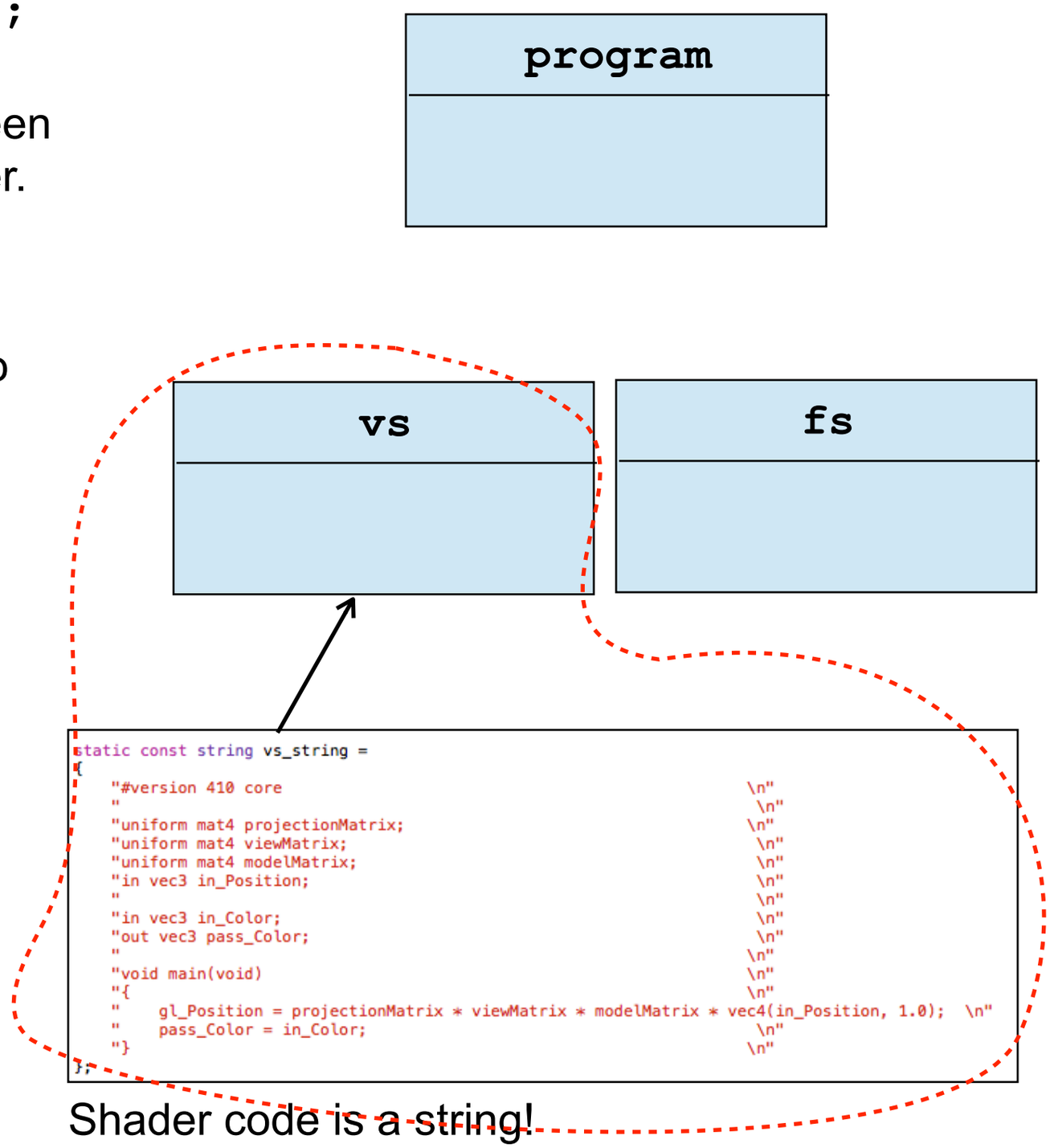
```
GLuint fs = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fs, 1, &fs_source, NULL);
                    [.....]

GLuint vs = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vs, 1, &vs_source, NULL);
                    [.....]
```

# Shader Compiler

```
void glCompileShader(GLuint shader);
```

compiles the source code strings that have been stored in the shader object specified by shader.

- The shader source string must have been already attached.
- If you change the source string, you have to recompile.
- This step is invoked when your program is executed, after start.




Shader code is a string!

# Example

```
GLuint fs = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fs, 1, &fs_source, NULL);
glCompileShader(fs);
```

```
GLuint vs = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vs, 1, &vs_source, NULL);
glCompileShader(vs);
```

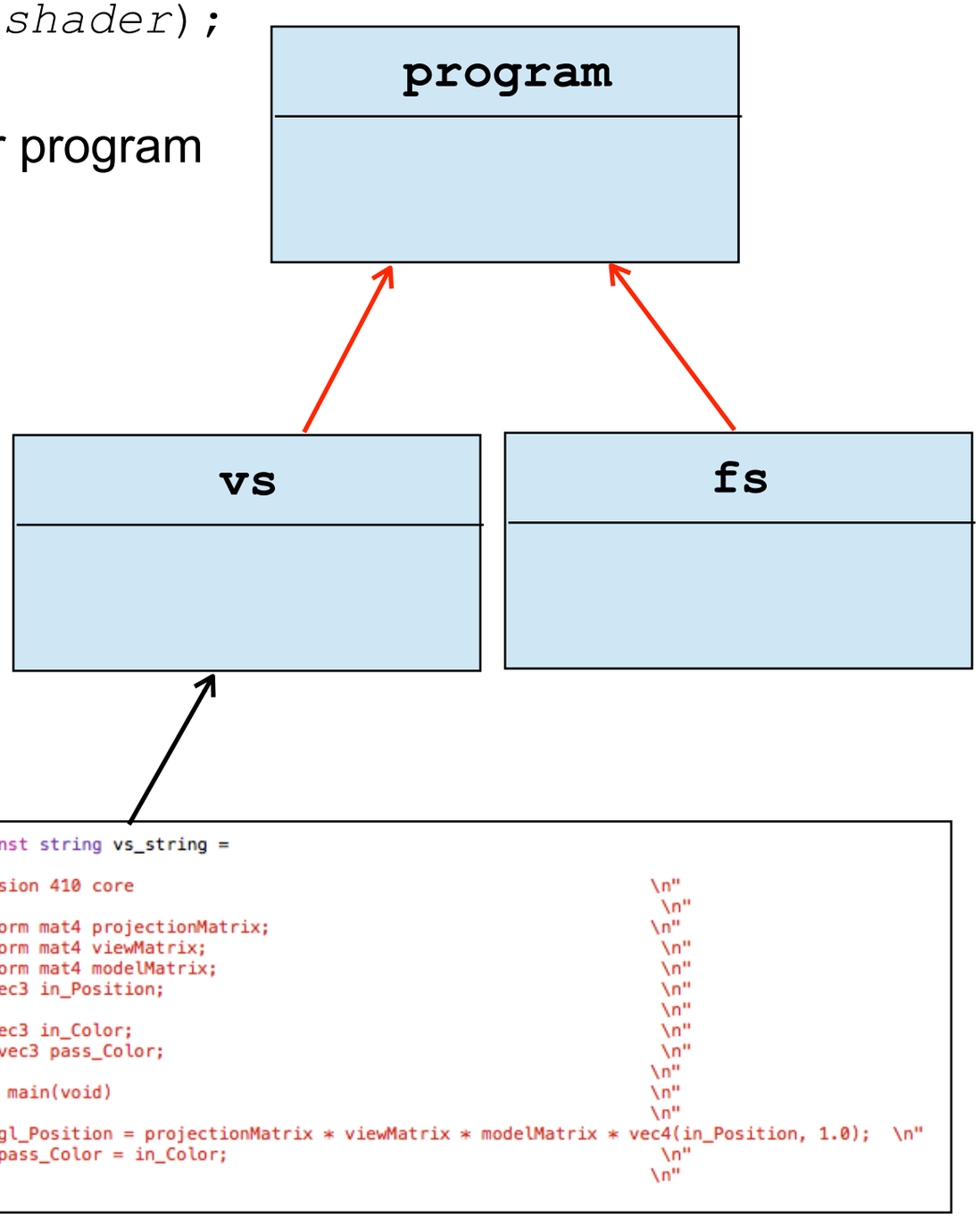
# Attach the Shader to your Program


```
void glAttachShader(GLuint program, GLuint shader);
```

Allows you to add the compiled shader code to a shader program

Parameters

- program: Specifies the program object to which a shader object will be attached.
- shader: Specifies the shader object that is to be attached.



# Example

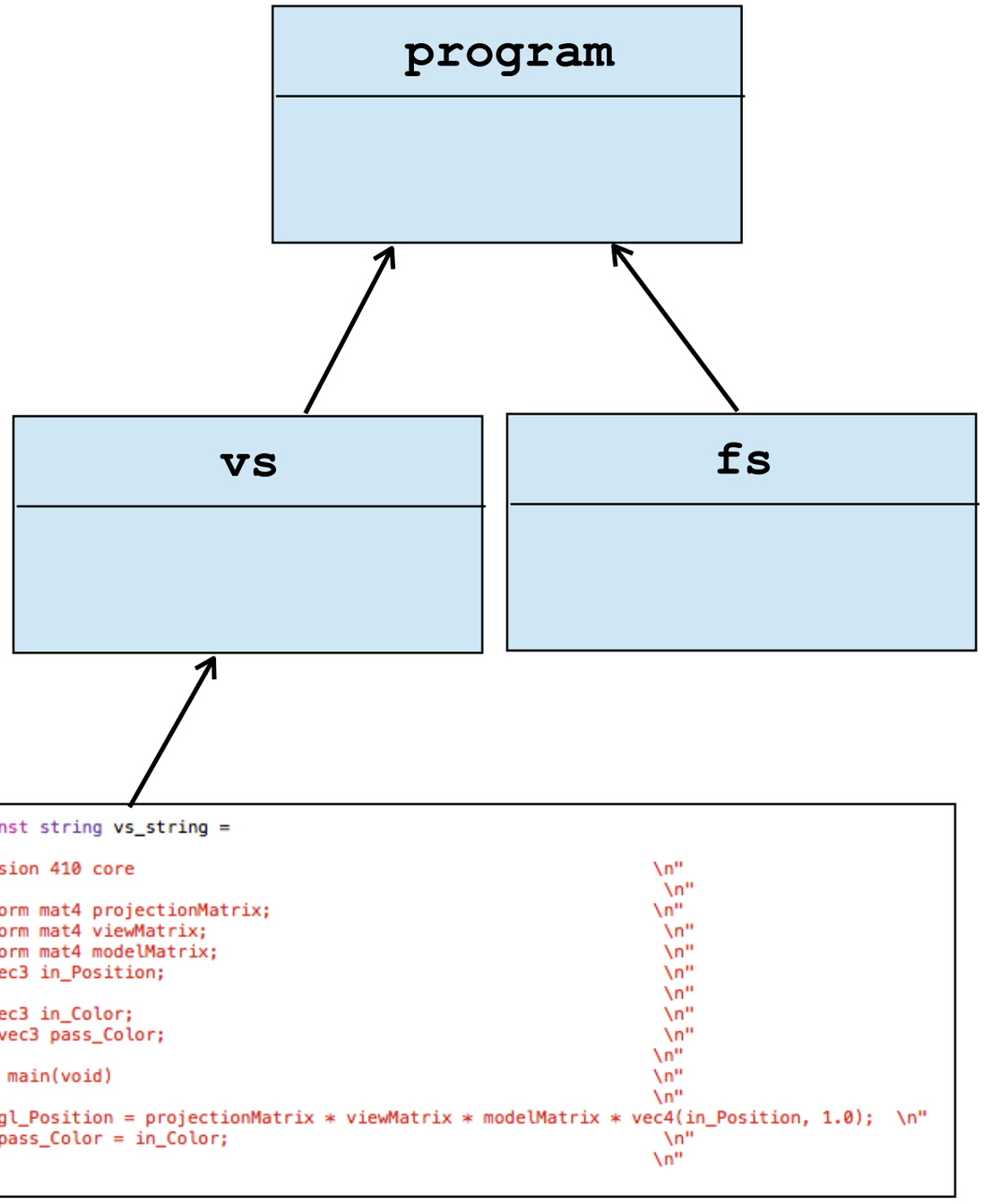
```
GLuint program = glCreateProgram();

GLuint fs = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fs, 1, &fs_source, NULL);
glCompileShader(fs);

GLuint vs = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vs, 1, &vs_source, NULL);
glCompileShader(vs);

// We'll attach our two compiled shaders to the OpenGL program.
glAttachShader(program, vs);
glAttachShader(program, fs);
```

# Linker

```
void glLinkProgram( GLuint program);
```

links the program object with the GLSL libraries and all source code files.

The result is a binary program in machine code that is executed on the graphics processor.




Parameters

- program: Specifies the handle of the program object to be linked.

# Example

```
GLuint program = glCreateProgram();

GLuint fs = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fs, 1, &fs_source, NULL);
glCompileShader(fs);

GLuint vs = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vs, 1, &vs_source, NULL);
glCompileShader(vs);

// We'll attach our two compiled shaders to the OpenGL program.
glAttachShader(program, vs);
glAttachShader(program, fs);

glLinkProgram(program);
```

# Use the Shader

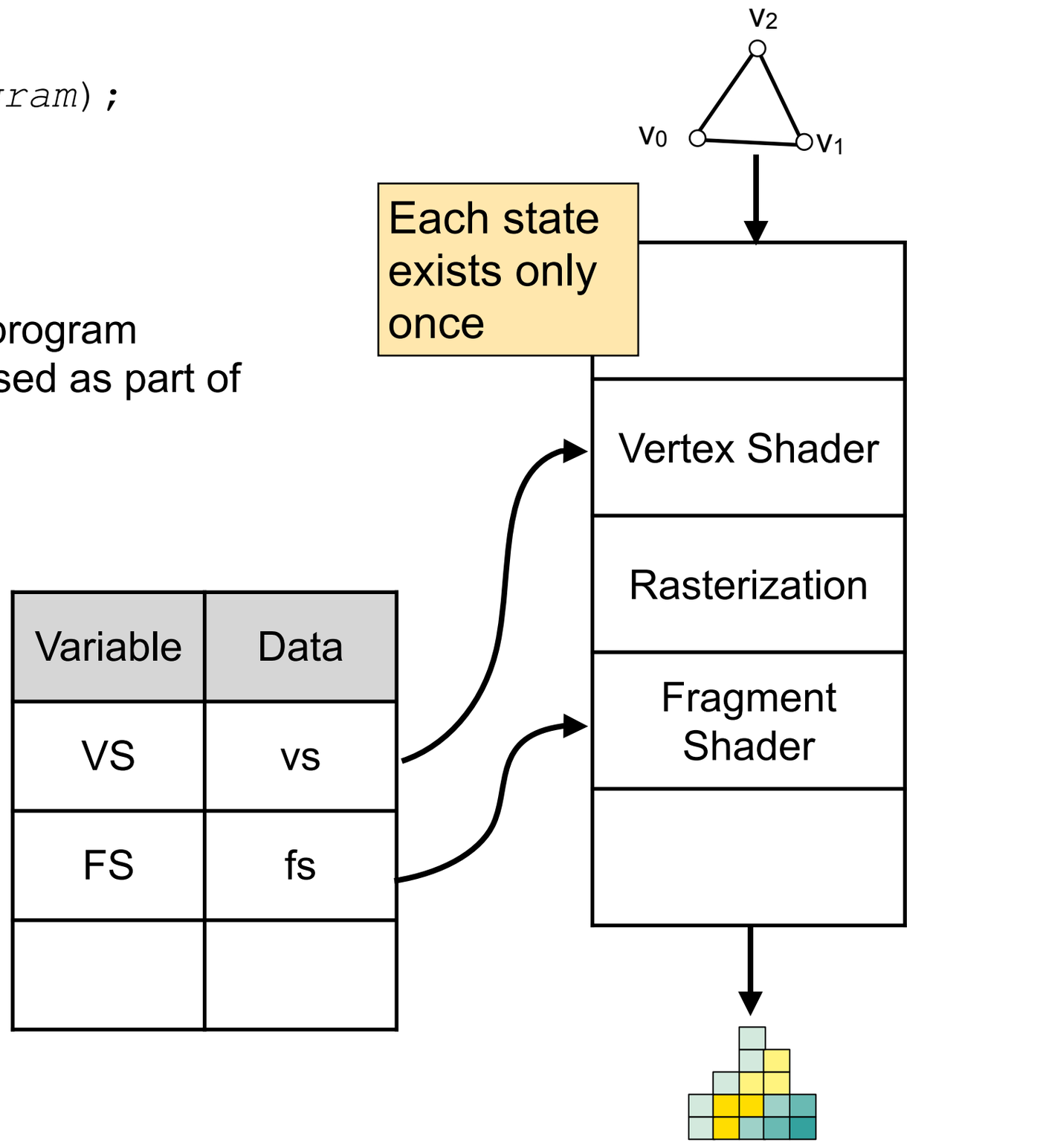
ARLAB

`void` **`glUseProgram`**`( GLuint` *`program`*`);`

Turn on the program

Parameters

- program: Specifies the handle of the program object whose executables are to be used as part of current rendering state.



# Example

Init

```
GLuint program = glCreateProgram();

GLuint fs = glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(fs, 1, &fs_source, NULL);
glCompileShader(fs);


GLuint vs = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vs, 1, &vs_source, NULL);
glCompileShader(vs);


// We'll attach our two compiled shaders to the OpenGL program.
glAttachShader(program, vs);
glAttachShader(program, fs);

glLinkProgram(program);
```

Runtime

```
glUseProgram(program);
```

# Program Structure

```
int main(int argc, const char * argv[])
{
        [....]
```

Program init

**Init shader 1**

```
        [....]
```

**Init shader 2**

```
        [....]
```

```
while
```

main loop

**Use shader 1**

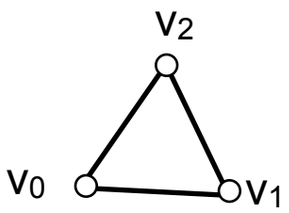
```
        [....]
```

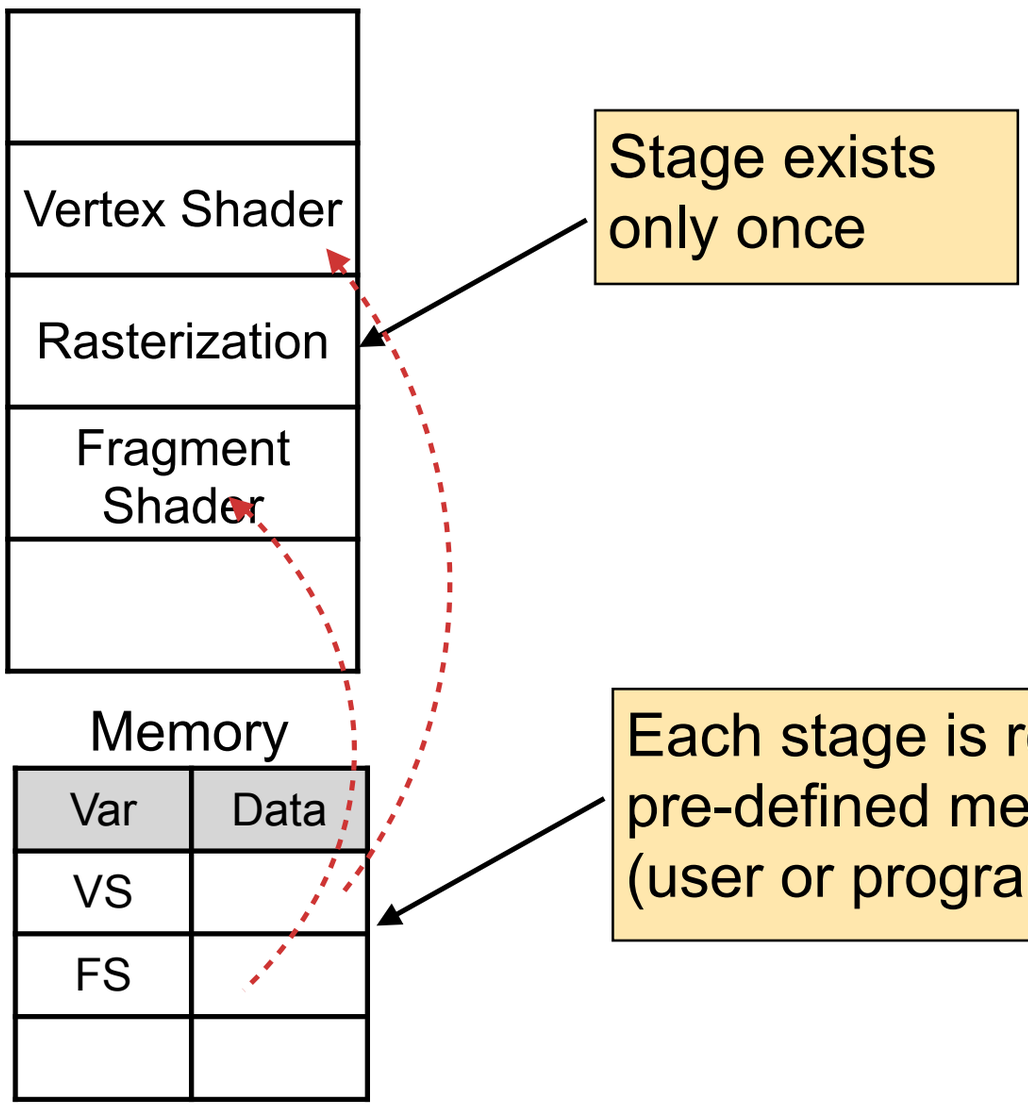
**Use shader 2**

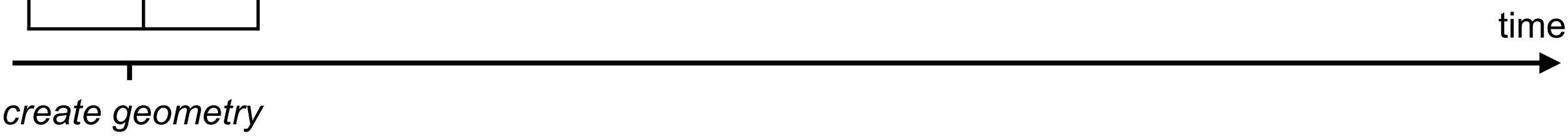
```
        [....]
}
```

**Runtime**

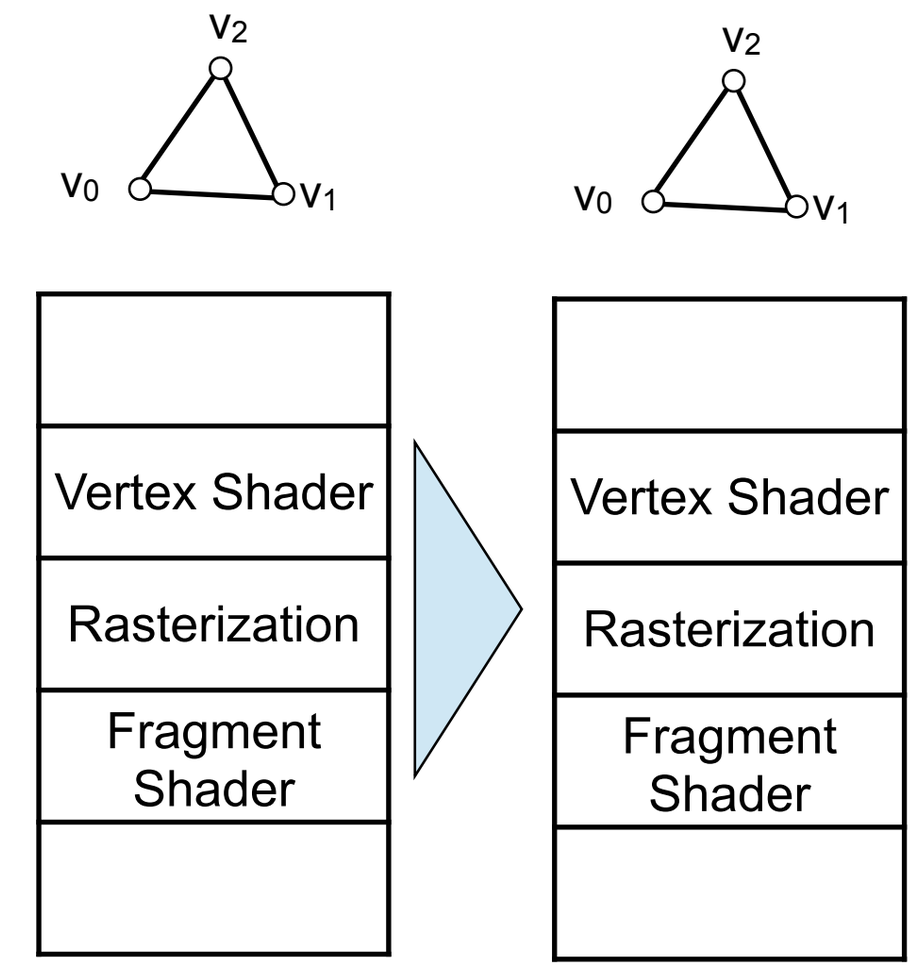
# Process


v2
v0
v1

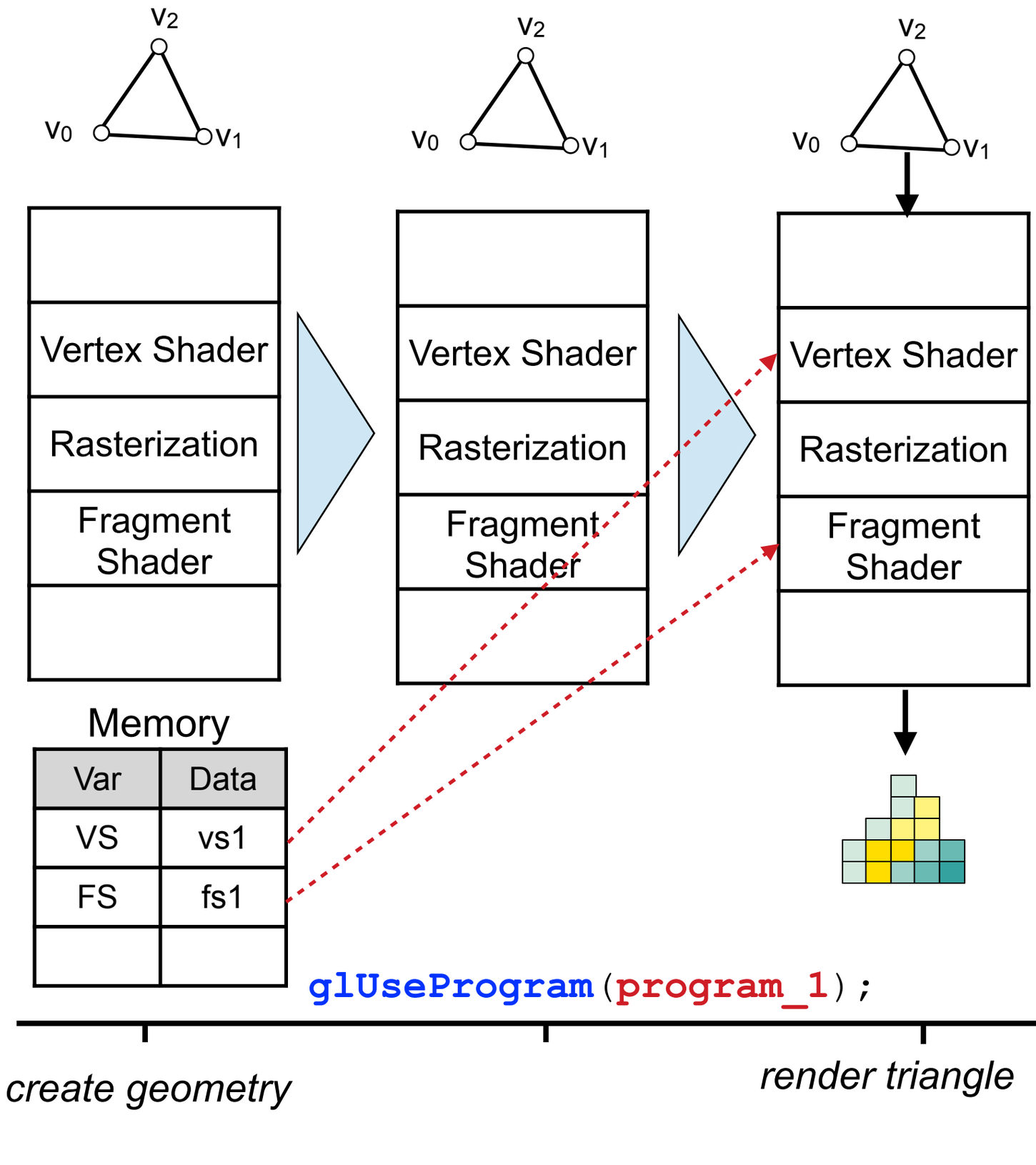

Vertex Shader
Rasterization
Fragment Shader
Stage exists only once
Memory
Var
Data
VS
FS
Each stage is related to pre-defined memory field (user or program defined)


time
create geometry

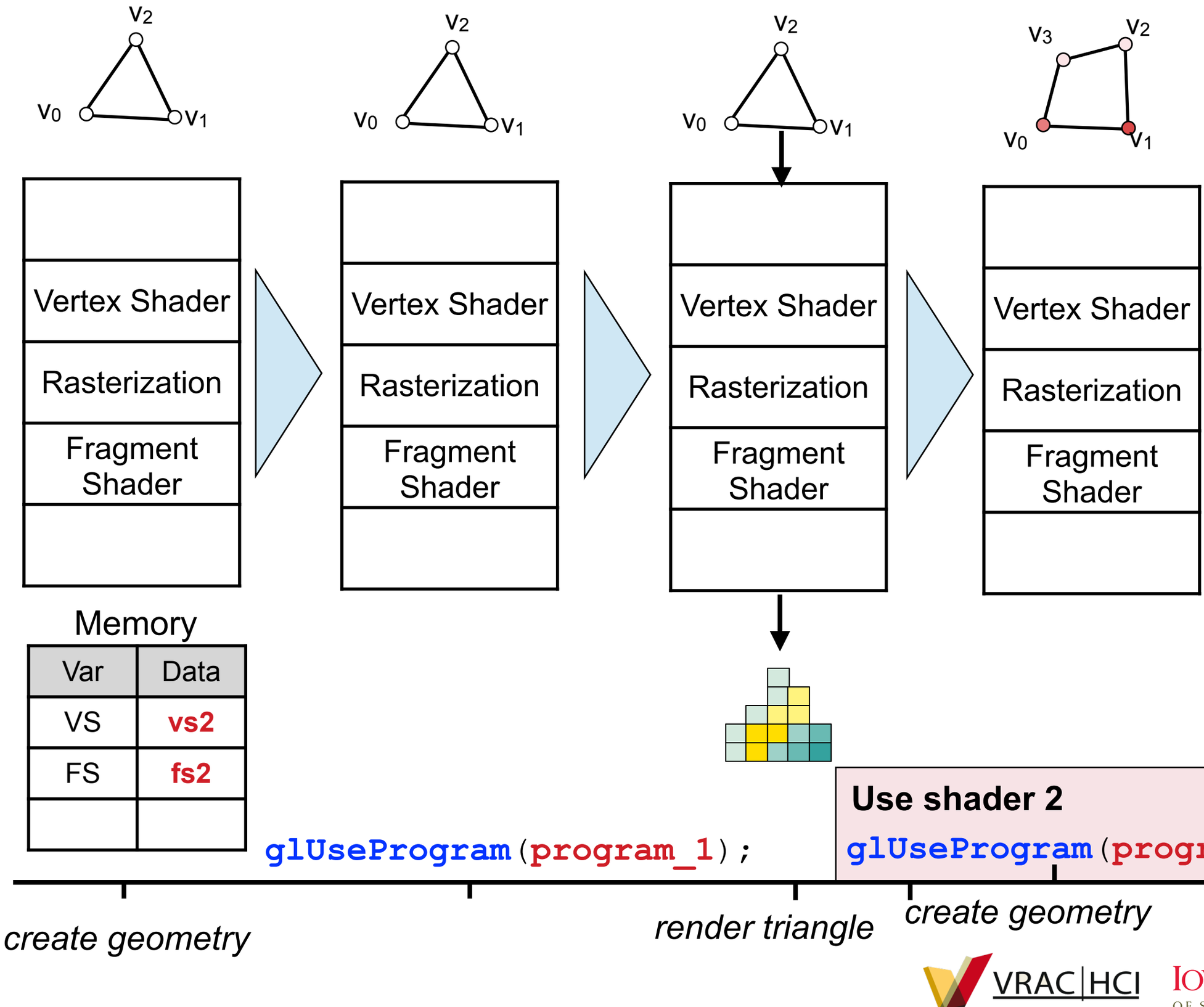
# Process


v2
v0
v1
v2
v0
v1
Vertex Shader
Rasterization
Fragment Shader
Vertex Shader
Rasterization
Fragment Shader
Memory
Var
Data
VS
vs1
FS
fs1
Use shader 1
glUseProgram(program_1);
time
create geometry

# Process

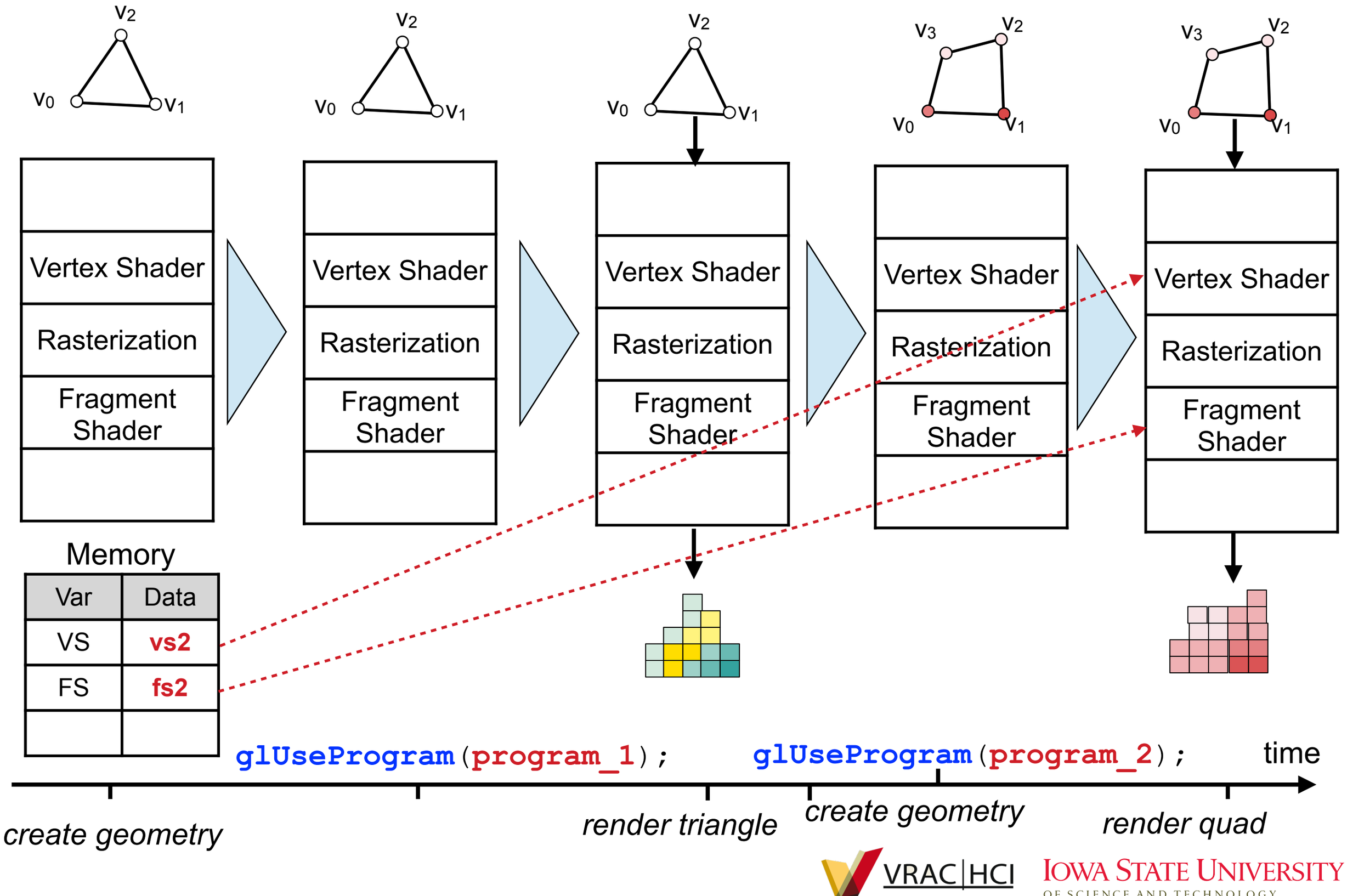

v2
v0
v1
Vertex Shader
Rasterization
Fragment Shader
Memory
Var
Data
VS
vs1
FS
fs1
glUseProgram(program_1);
time
create geometry
render triangle

# Process


v2
v0
v1
Vertex Shader
Rasterization
Fragment Shader
v3
Memory
Var
Data
VS
vs2
FS
fs2
glUseProgram(program_1);
Use shader 2
glUseProgram(program_2);
time
create geometry
render triangle
create geometry

# Process

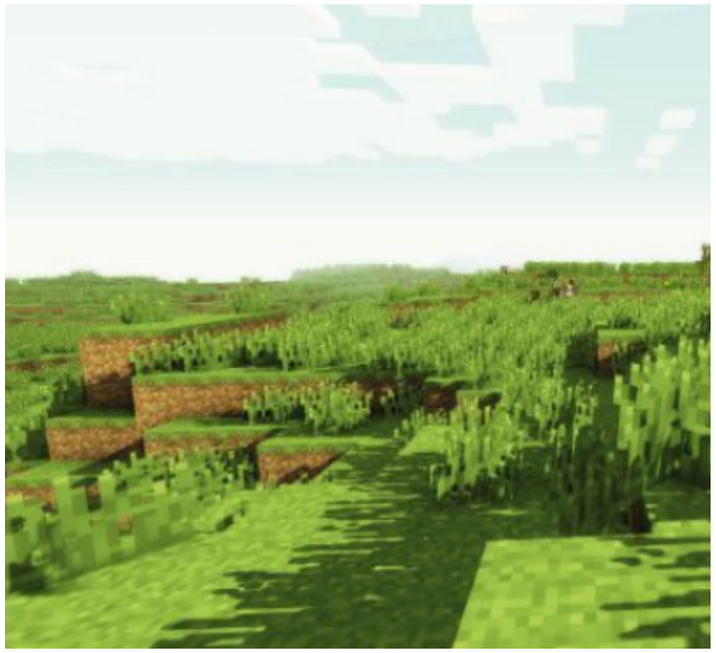
ARLAB



ARLAB

# When do we create a new shader?
# or
# how many shaders do we need?

# Appearance Counts



Shadow

Glare

Cartoon

Reflectance

Transparency

Shader programs allow us to define the visual appearance of an object. We can combine many different shader programs in one application.

However, this is (almost) a basic course. We will not deal with advanced graphic effects

ARLAB

# Questions ?

VRAC|HCI

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY

# Thank you!

# Questions

Rafael Radkowski, Ph.D.
Iowa State University
Virtual Reality Applications Center
1620 Howe Hall
Ames, Iowa 50011, USA
+1 515.294.5580

rafael@iastate.edu
http://arlab.me.iastate.edu

www.linkedin.com/in/rradkowski

ARLAB

VRAC|HCI

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY