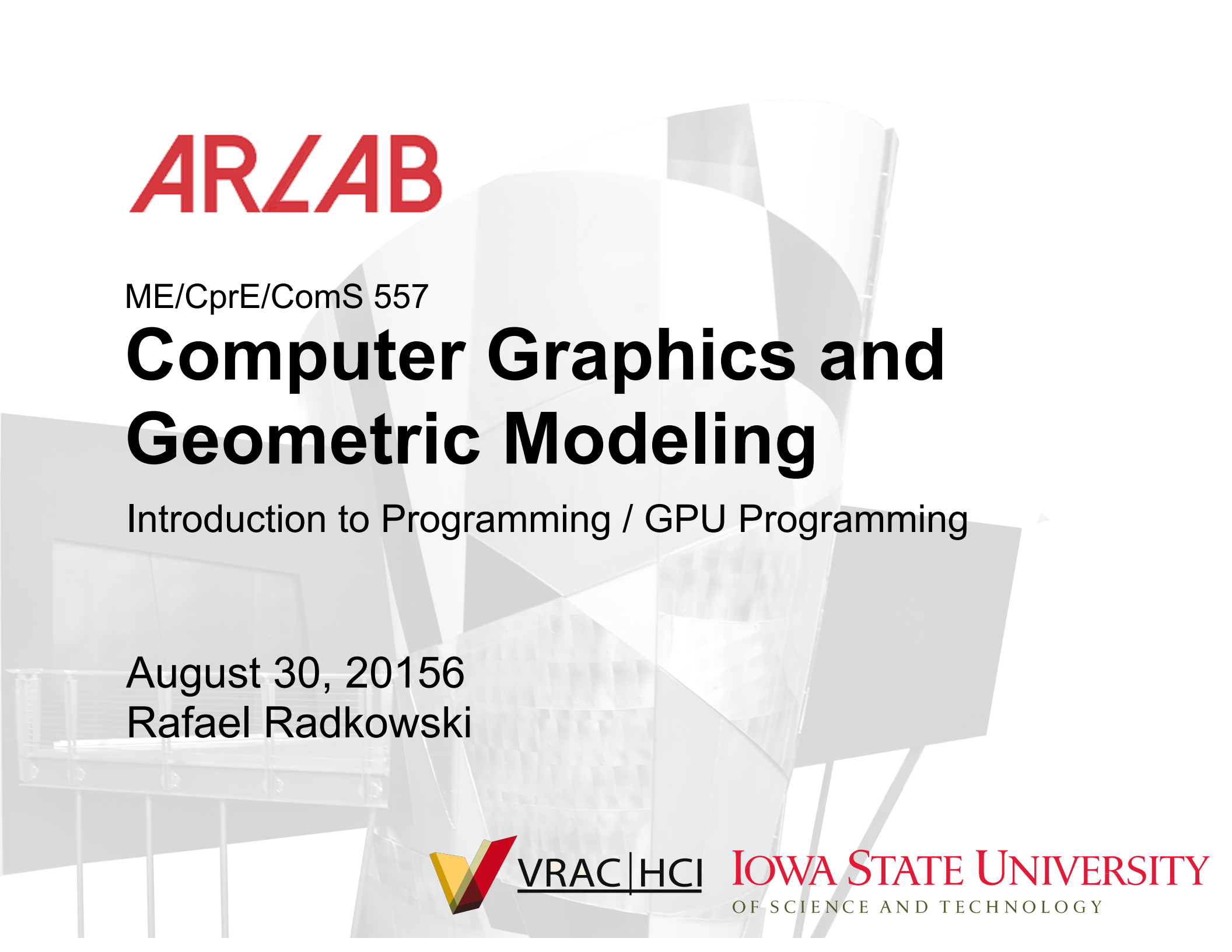


ME/CprE/ComS 557

Computer Graphics and Geometric Modeling

Introduction to Programming / GPU Programming

August 30, 20156

Rafael Radkowski

IOWA STATE UNIVERSITY
OF SCIENCE AND TECHNOLOGY

Content

ARLAB

- C++ Introduction
  - Datatypes
  - Operations
  - Functions
- GLSL Introduction

C++

ARLAB

C++ is a high-level, general-purpose programming language.

C++ is standardized by the International Organization for Standardization (ISO). The current standard version (December 2014) is ISO/IEC 14882:2014 (also known as C++14).

History

Bjarne Stroustrup, a Danish computer scientist, began his work on C++'s predecessor "C with Classes" in 1979.

The first edition was was ready and released in 1985.

Philosophy

- Programmers should be able to program in their own style
- No implicit violations of the type system but allows explicit violations;
- User-created types need to have the same support and performance as built-in types.
- .....
- The programmer has a lot of freedom BUT must know what he or she does !!!

C++ Code

ARLAB

C++ code example

```
147
148
149 int main(int argc, const char * argv[])
150 {
151     // Initialize GLFW, and if it fails to initialize for any reason, print it out to STDERR.
152     if (!glfwInit()) {
153         fprintf(stderr, "Failed initialize GLFW.");
154         exit(EXIT_FAILURE);
155     }
156
157     // Set the error callback, as mentioned above.
158     glfwSetErrorCallback(error_callback);
159
160     // Set up OpenGL options.
161     // Use OpenGL verion 4.1,
162     glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
163     glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 1);
164     // GLFW_OPENGL_FORWARD_COMPAT specifies whether the OpenGL context should be forward-compatible, i.e. one where all functionality
165     glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
166     // Indicate we only want the newest core profile, rather than using backwards compatible and deprecated features.
167     glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
168     // Make the window resize-able.
169     glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);
170
171     // Create a window to put our stuff in.
172     GLFWwindow* window = glfwCreateWindow(800, 600, "Hello OpenGL", NULL, NULL);
173
174     // If the window fails to be created, print out the error, clean up GLFW and exit the program.
175     if(!window) {
176         fprintf(stderr, "Failed to create GLFW window.");
177         glfwTerminate();
178         exit(EXIT_FAILURE);
179     }
180
181     // Use the window as the current context (everything that's drawn will be place in this window).
182     glfwMakeContextCurrent(window);
183
184     // Set the keyboard callback so that when we press ESC, it knows what to do.
185     glfwSetKeyCallback(window, key_callback);
186
187     printf("OpenGL version supported by this platform (%s): \n", glGetString(GL_VERSION));
188
189     // Makes sure all extensions will be exposed in GLEW and initialize GLEW.
190     glewExperimental = GL_TRUE;
191     glewInit();
192
193     // Shaders is the next part of our program. Notice that we use version 410 core. This has to match our version of OpenGL we are u
```

Structure of a Program

ARLAB

```
// my first program in C++
#include <iostream>

int main()
{
  std::cout << "Hello World!";


}
```

Every program starts at the entry point "main".

Every program can only have ONE main - entry point

Output on screen:

Hello World!

http://www.cplusplus.com/doc/tutorial/program_structure/

C++ Program

ARLAB

```
/ operating with variables

#include <iostream>
using namespace std;

int main ()
{
  // declaring variables:
  int a, b;
  int result;

  // process:
  a = 5;
  b = 2;
  a = a + 1;
  result = a - b;

  // print out the result:
  cout << result;

  // terminate the program:
  return 0;
}
```

Datatypes

C++ Program

ARLAB

```
/ operating with variables

#include <iostream>
using namespace std;

int main ()
{
  // declaring variables:
  int a, b;
  int result;

  // process:
  a = 5;
  b = 2;
  a = a + 1;
  result = a - b;

  // print out the result:
  cout << result;

  // terminate the program:
  return 0;
}
```

Datatypes (Declaration)

Datatypes:

The values of variables are stored somewhere in the memory of your computer. The program does not need to know where.

The program needs to know what kind of data will be stored and how much storage is required, and how the programmer wants to refer to this storage:

type *[name]*

- type: what data should be stored
- name: the name to link the storage with your program

You **must** declare every variable before you use (define) and process it.

Datatypes

ARLAB

Fundamental datatypes:

Basic datatype which are "implemented" as part of your compiler

```
int a, b;

double dA, dB;

bool bA, bB;

char cA, cB;

string strA, strB;
```

:is a STL (Standard Template Library) datatype

API datatypes

Datatypes, provided by a programming API such as OpenGL

```
GLuint a, b;

glm::mat4 a_matrix, b_matrix;
```

Datatypes

ARLAB

Here is the complete list of fundamental types in C++:

Group	Type names*	Notes on size / precision
Character types	**char**	Exactly one byte in size. At least 8 bits.
	char16_t	Not smaller than char. At least 16 bits.
	char32_t	Not smaller than char16_t. At least 32 bits.
	wchar_t	Can represent the largest supported character set.
Integer types (signed)	**signed char**	Same size as char. At least 8 bits.
	signed **short** *int*	Not smaller than char. At least 16 bits.
	signed **int**	Not smaller than short. At least 16 bits.
	signed **long** *int*	Not smaller than int. At least 32 bits.
	signed **long long** *int*	Not smaller than long. At least 64 bits.
Integer types (unsigned)	**unsigned char**	(same size as their signed counterparts)
	unsigned short *int*	
	unsigned *int*	
	unsigned long *int*	
	unsigned long long *int*	
Floating-point types	**float**	
	double	Precision not less than float
	long double	Precision not less than double
Boolean type	**bool**	
Void type	**void**	no storage
Null pointer	**decltype(nullptr)**	

http://www.cplusplus.com/doc/tutorial/variables/

C++ Program

ARLAB

```
/ operating with variables

#include <iostream>
using namespace std;

int main ()
{
  // declaring variables:
  int a, b;
  int result;

  // process:
  a = 5;
  b = 2;
  a = a + 1;
  result = a - b;

  // print out the result:
  cout << result;

  // terminate the program:
  return 0;
}
```

Basic Operations (Definition)

- Every line is terminated with a semicolon ;
- Declared variables can be combined with basic operators

Namespaces

ARLAB

```
#include <iostream>

int main()
{
  int age = 9;
  int zipcode = 50011;
   std::cout << "Hello World!"
   std::cout << "I am " << age << "
          years old and my zipcode is " << zipcode << std::endl;
}
```

```
#include <iostream>
using namespace std;

int main()
{
  int age = 9;
  int zipcode = 50011;
   cout << "Hello World!"
   cout << "I am " << age << "
          years old and my zipcode is " << zipcode << endl;
}
```

Namespace declaration

Statements and Control Flow

ARLAB

Please review them on http://www.cplusplus.com/doc/tutorial/control/

Selection statements: if and else

```
if (x == 100){
  cout << "x is 100";
}
else{
  cout << "x is not 100";
}
```

Selection statements: switch

```
switch (x) {
  case 1:
    cout << "x is 1";
    break;
  case 2:
    cout << "x is 2";
    break;
  default:
    cout << "value of x unknown";
  }
```

Statements and Control Flow

ARLAB

Please review them on http://www.cplusplus.com/doc/tutorial/control/

The for loop

```
for (int n=10; n>0; n--) {
    cout << n << ", ";
}
```

The while loop

```
int n = 10;

while (n>0) {
    cout << n << ", ";
    --n;
}
```

C++ Functions

A function is a group of statements that is executed when it is called from some point of the program. It allows to structure programs in segments of code to perform individual tasks.

The following is its format:

type name (parameter1, parameter2, ...) { statements }

where:

- type is the data type specifier of the data returned by the function.
- name is the identifier by which it will be possible to call the function.
- parameters (as many as needed): Each parameter consists of a data type specifier followed by an identifier, like any regular variable declaration (for example: int x) and which acts within the function as a regular local variable. They allow to pass arguments to the function when it is called. The different parameters are separated by commas.
- statements is the function's body. It is a block of statements surrounded by braces { }.

C++ Function Example (1/2)

ARLAB

cpp file:

```
// function example
#include <iostream>
using namespace std;

int addition (int a, int b)
{
  int r;
  r=a+b;
  return (r);
}

int main ()
{
  int z;
  z = addition (5,3);
  cout << "The result is " << z;
  return 0;
}
```

Includes "copy & paste" the content of a header file

Function

All variables that are defined inside a function are only valid inside this function.

Returns the value of r.

Function call: the function must be defined before it is called in a cpp file.

C++ Function Example (2/2)

ARLAB

cpp file:

```
// function example
#include <iostream>
using namespace std;

void addition (int a, int b, int* r);

int main ()
{
  int z;
  z = addition (5,3, &z);
  cout << "The result is " << z;
  return 0;
}

int addition (int a, int b, int* r)
{
  (*r)=a+b;
}
```

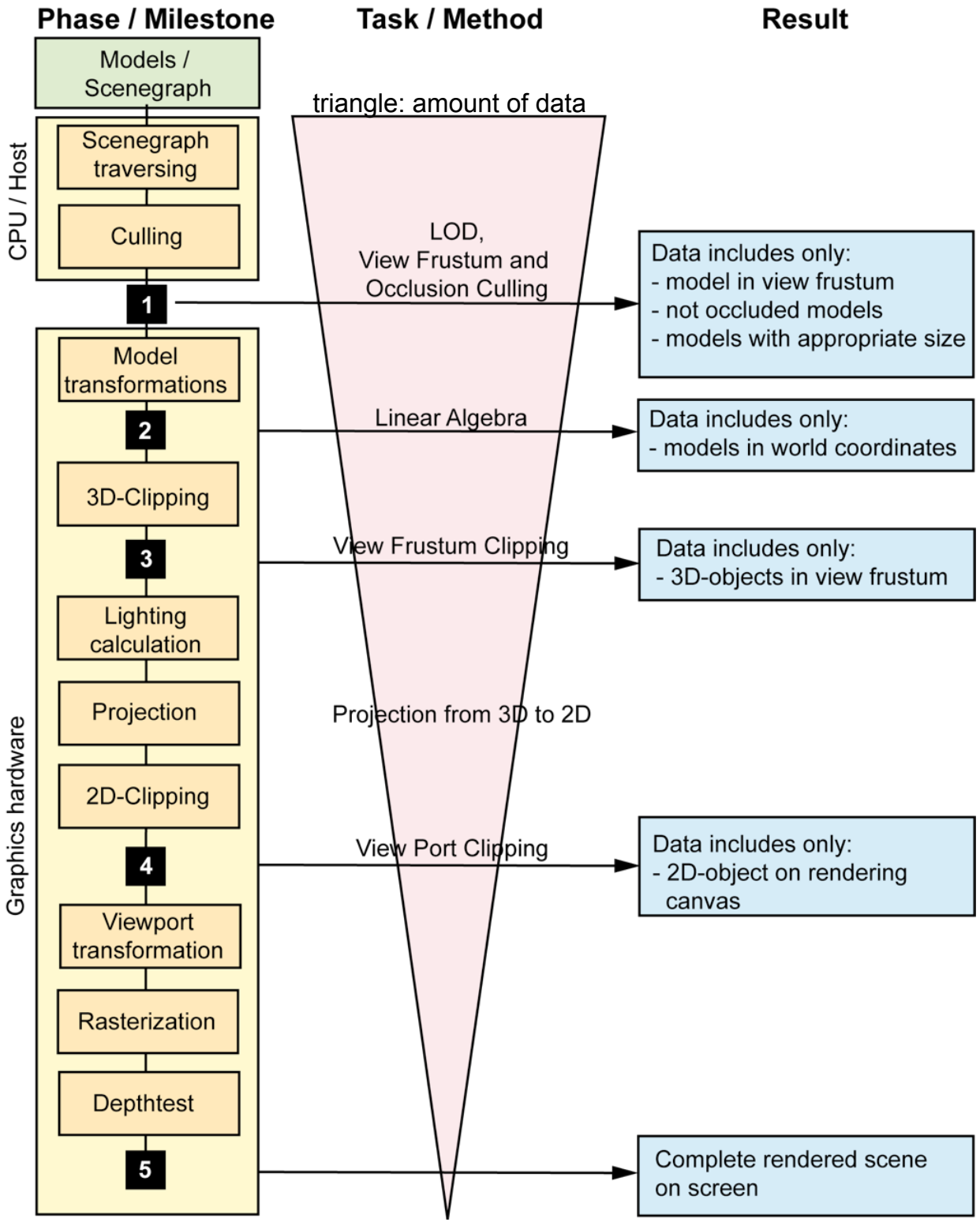
Function prototype; can also be specified in a header file. The header file must be included.

A third variable is added for the return value: **call-by-reference**.

Function implementation. Using a prototype, the implementation can be located after the function call in the main function. If a header file for the prototype is used, it is recommended to move the implementation into a related cpp-file.

The and symbol returns the address of the variable:

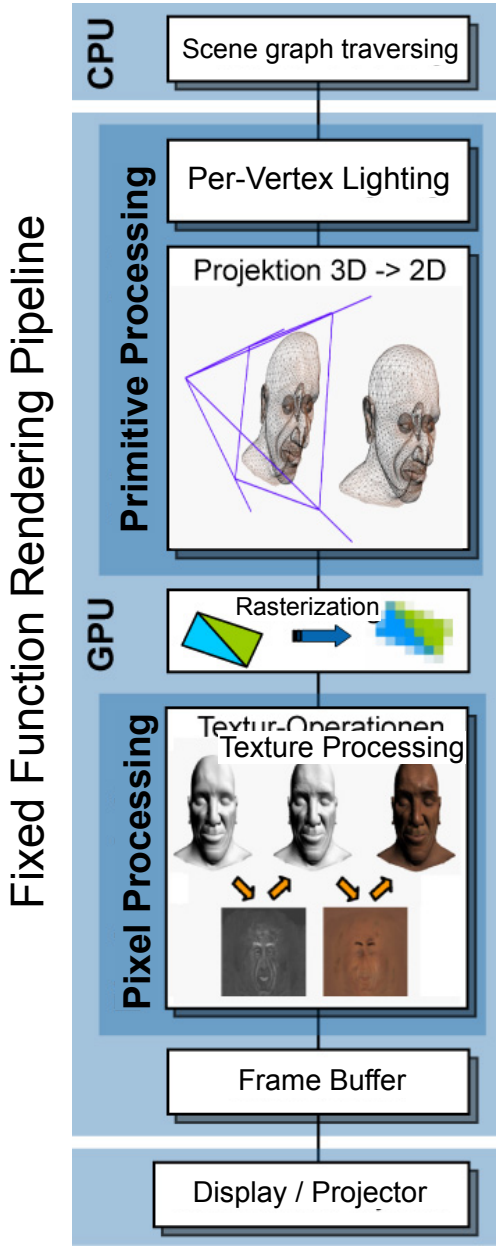
```
int z;
&z;
(&r); // gives access to the data
```

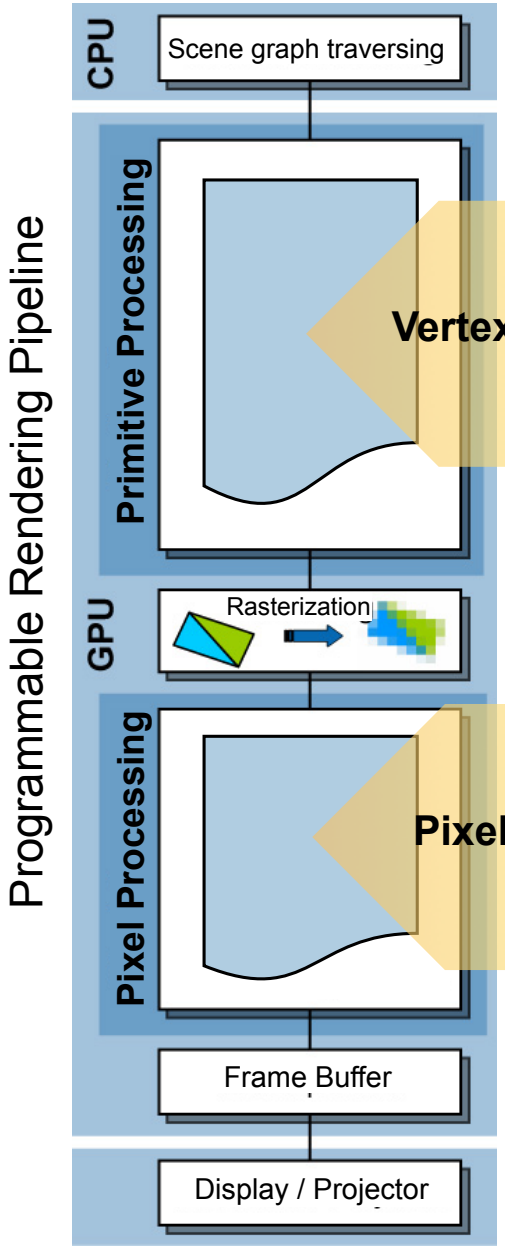


Rendering Pipeline

Concept of GPUs

ARLAB





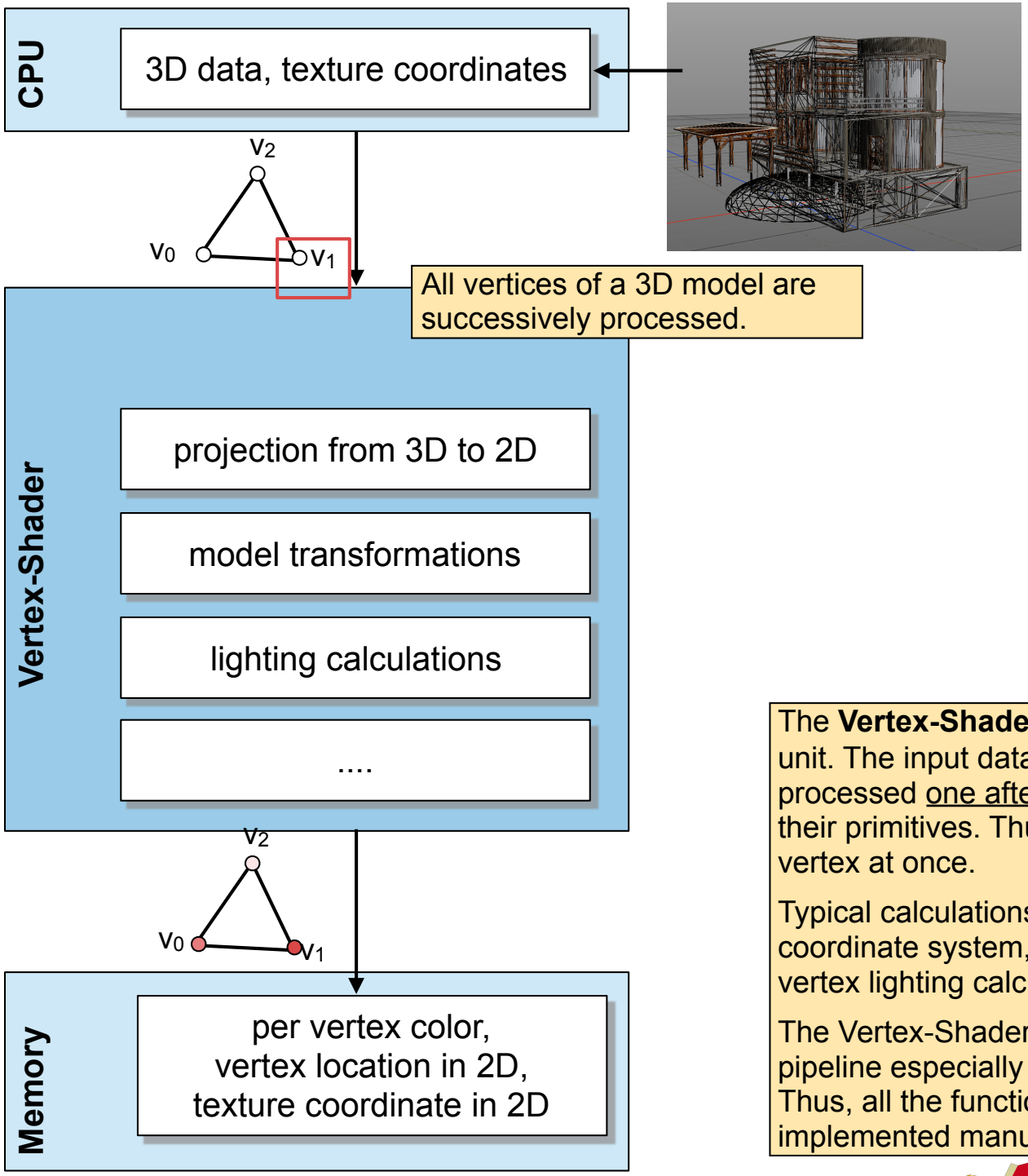
The **fixed function rendering pipeline** implements all necessary calculations (transformation & lighting, rasterization, shading, etc.) via hardware circuits to generate 3D graphic on display.

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

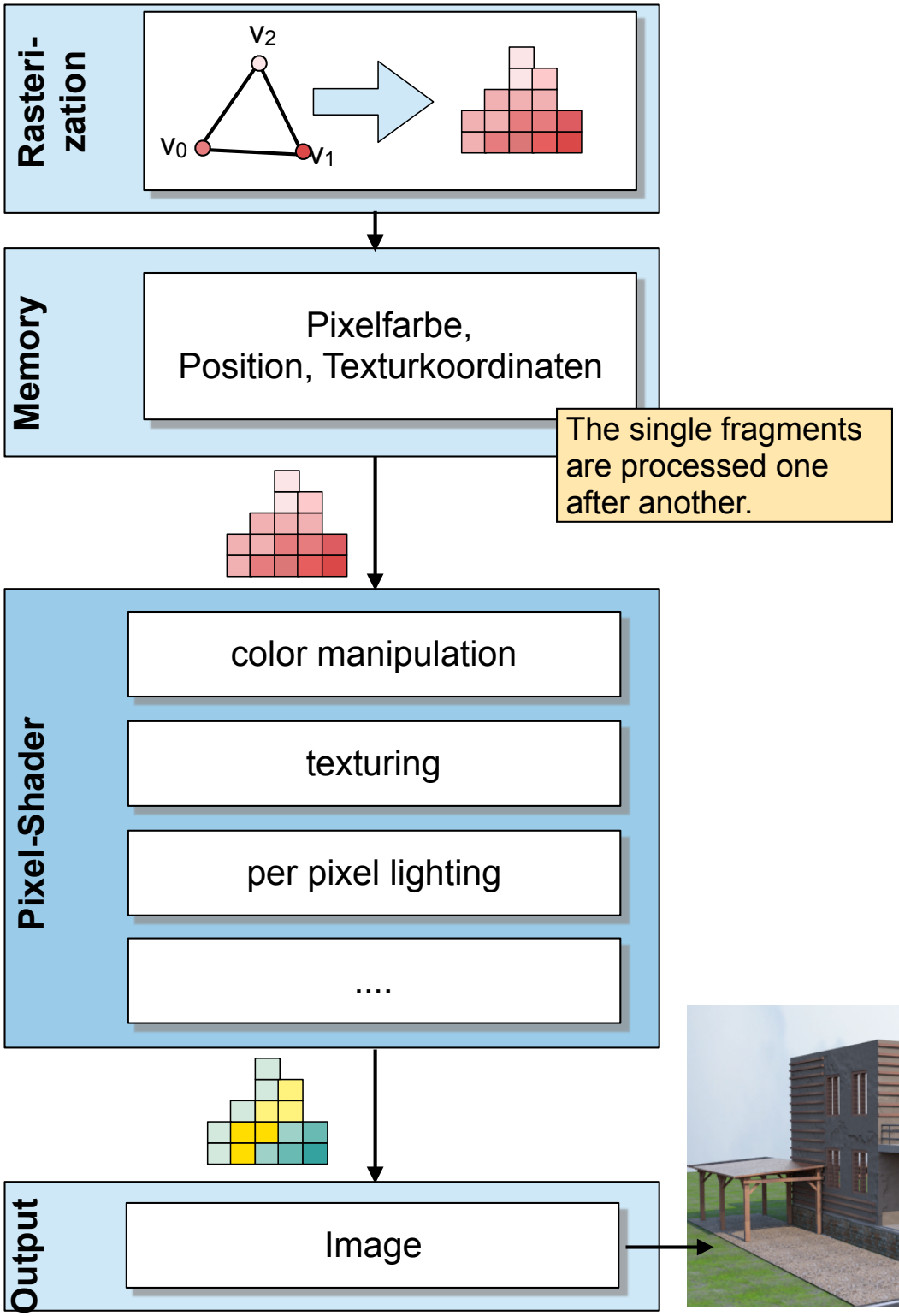


Vertex-Shader

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.



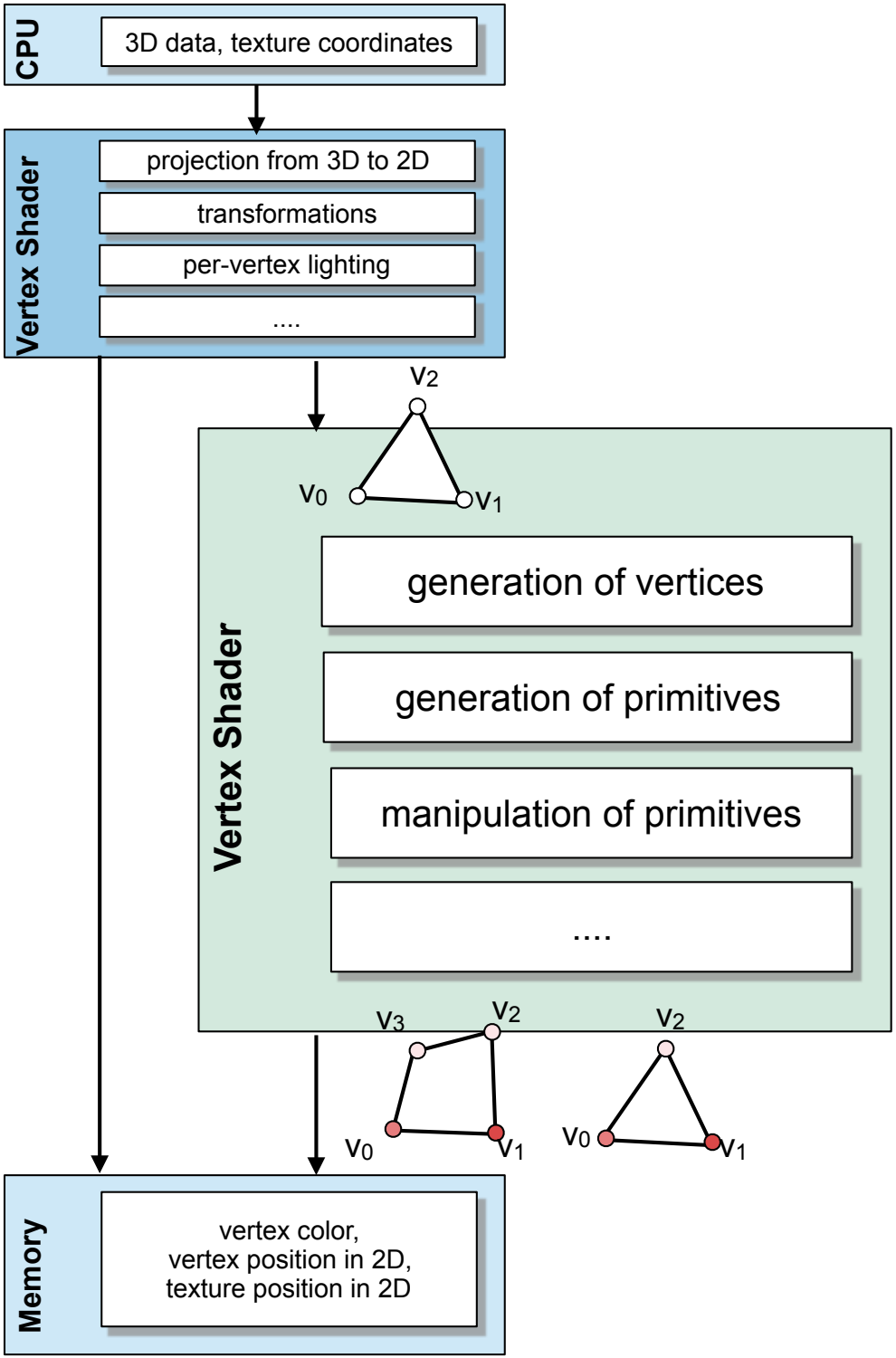
ARLAB

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



ARLAB

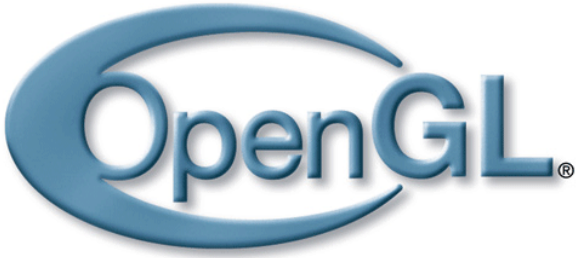
Geometry-Shader

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



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

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®

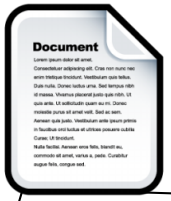
High Level Shading Language (HLSL) is a high-level shader language, published by Microsoft. It is a part of DirectX and conceptually integrated into the Microsoft DirectX programming pattern. Thus, it can only be used within DirectX graphics applications.

ARLAB

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

GL Shader Languange

ARLAB




```
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 fragment 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.

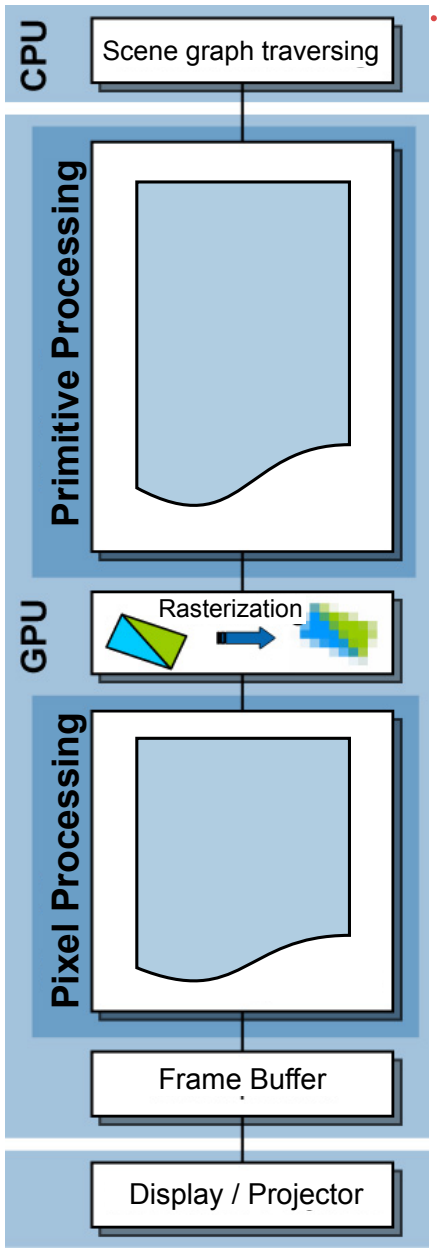
Type Qualifiers

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.

Type Qualifiers 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;
}
```

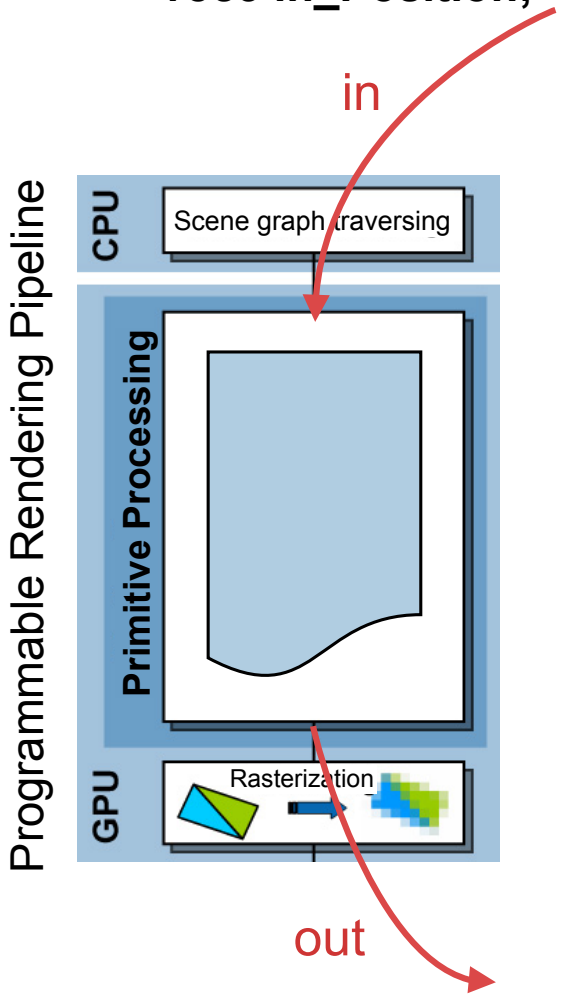
The data type **varying** allows push data from vertex shader to the fragment shader program.

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

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

Type Qualifier in, out, inout

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

vec3 in_Position;



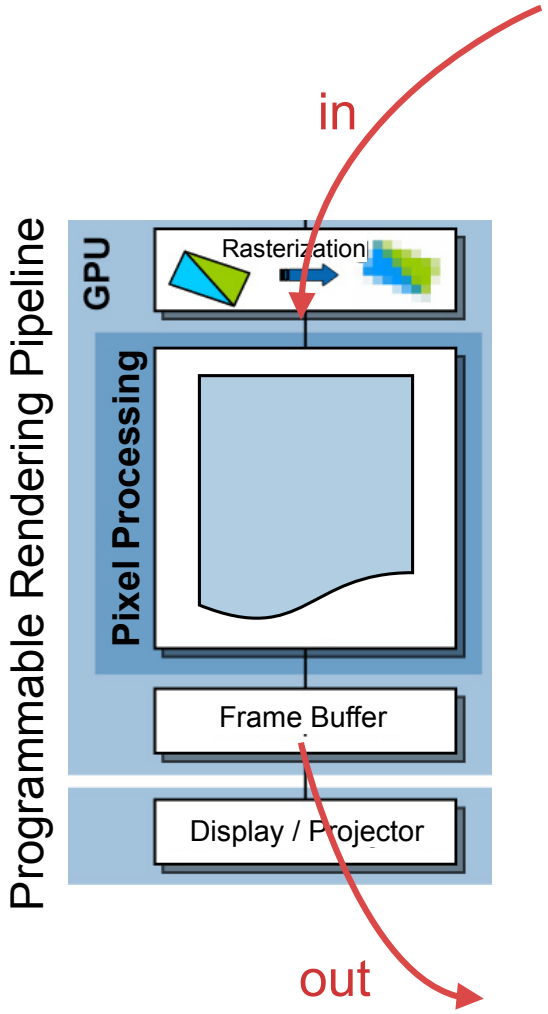

```
#version 410 core

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.

in vec3 in_Position;

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.

ARLAB

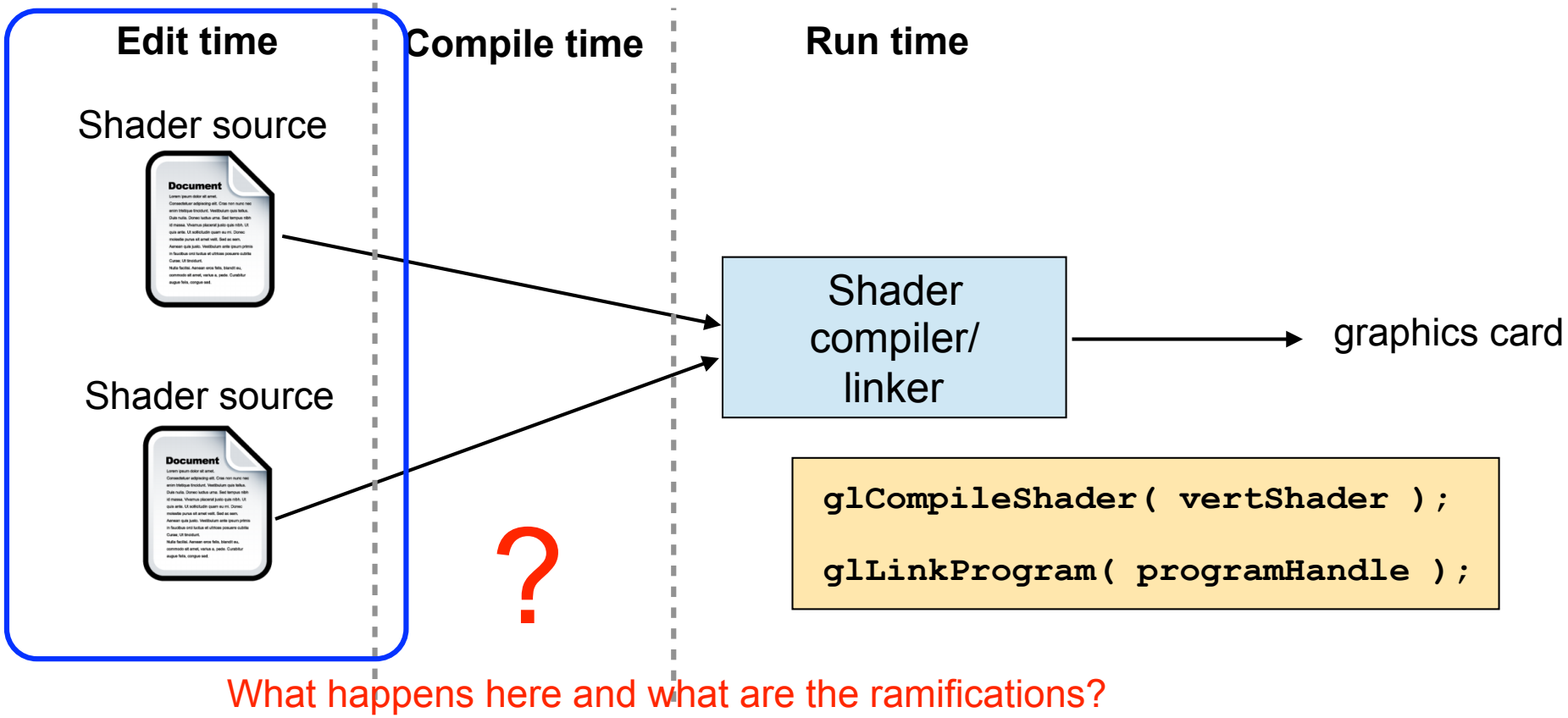
GLSL Program Structure and Process

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.

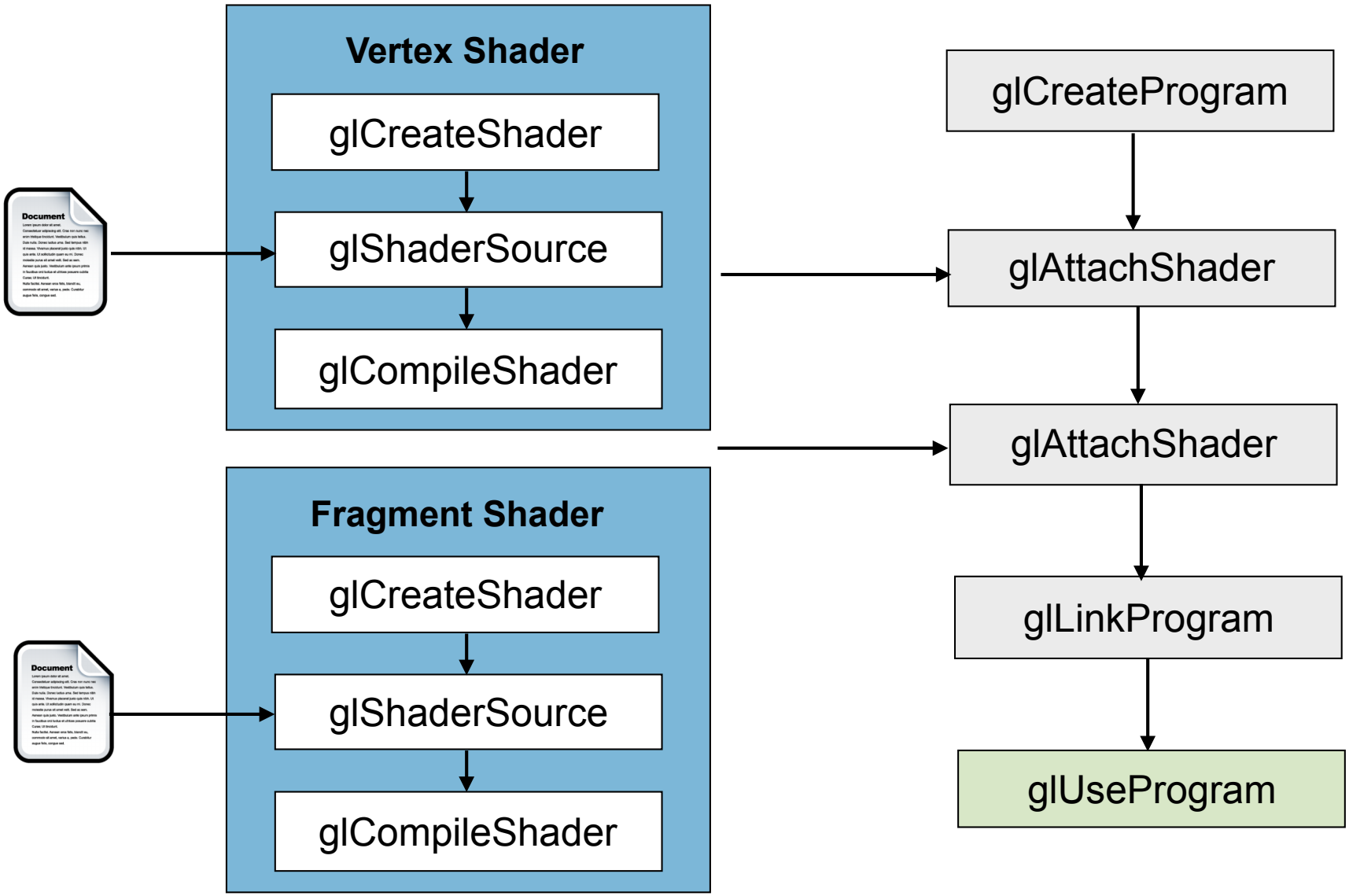
We need to do a little more



What happens here and what are the ramifications?

Programming Language Process

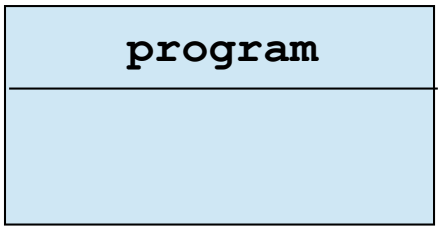
Process to create one shader = one type of appearance.



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();
```

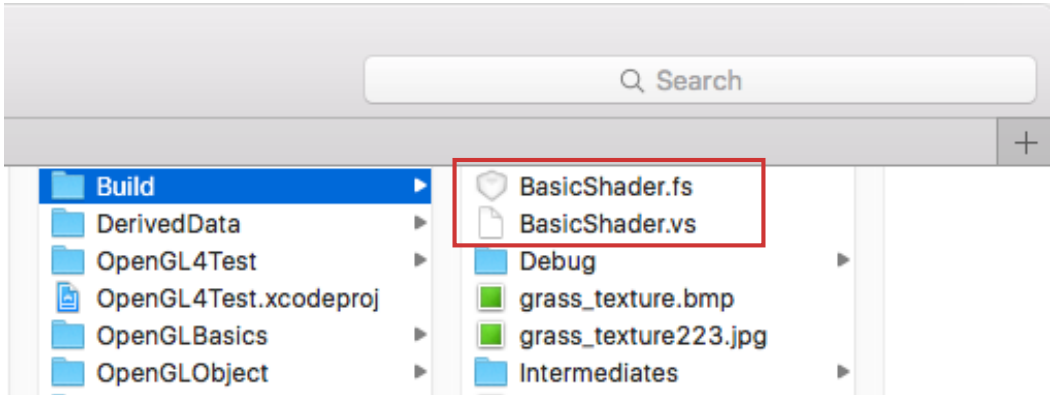
glCreateProgram creates an empty program object and returns a non-zero value by which it can be referenced.

Shader Code

```
static const string vs_string =
{
    "#version 410 core                                                 \n"
    "                                                                   \n"
    "uniform mat4 projectionMatrix;                                    \n"
    "uniform mat4 viewMatrix;                                          \n"
    "uniform mat4 modelMatrix;                                         \n"
    "in vec3 in_Position;                                              \n"
    "                                                                  \n"
    "in vec3 in_Color;                                                 \n"
    "out vec3 pass_Color;                                              \n"
    "                                                                  \n"
    "void main(void)                                                   \n"
    "{                                                                 \n"
    "    gl_Position = projectionMatrix * viewMatrix * modelMatrix * vec4(in_Position, 1.0);  \n"
    "    pass_Color = in_Color;                                        \n"
    "}                                                                 \n"
};

// Fragment shader source code. This determines the colors in the fragment generated in the shader p
    our vertex shader.
static const string fs_string  =
{
    "#version 410 core                                                 \n"
    "                                                                  \n"
    "in vec3 pass_Color;                                                 \n"
    "out vec4 color;                                                                \n"
    "void main(void)                                                   \n"
    "{                                                                 \n"
    "    color = vec4(pass_Color, 1.0);                               \n"
    "}                                                                 \n"
};
```

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!

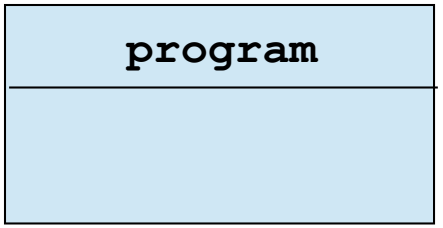
Shader Object

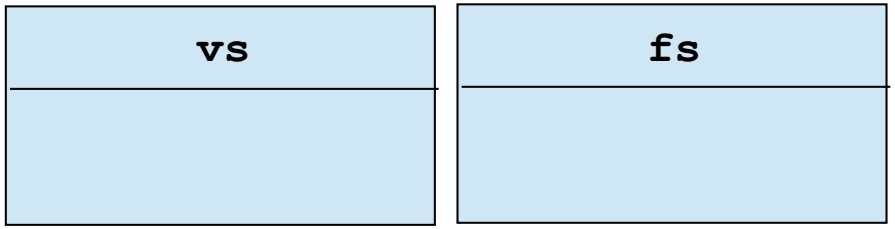
ARLAB

```
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

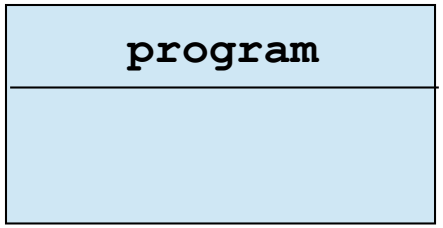
ARLAB

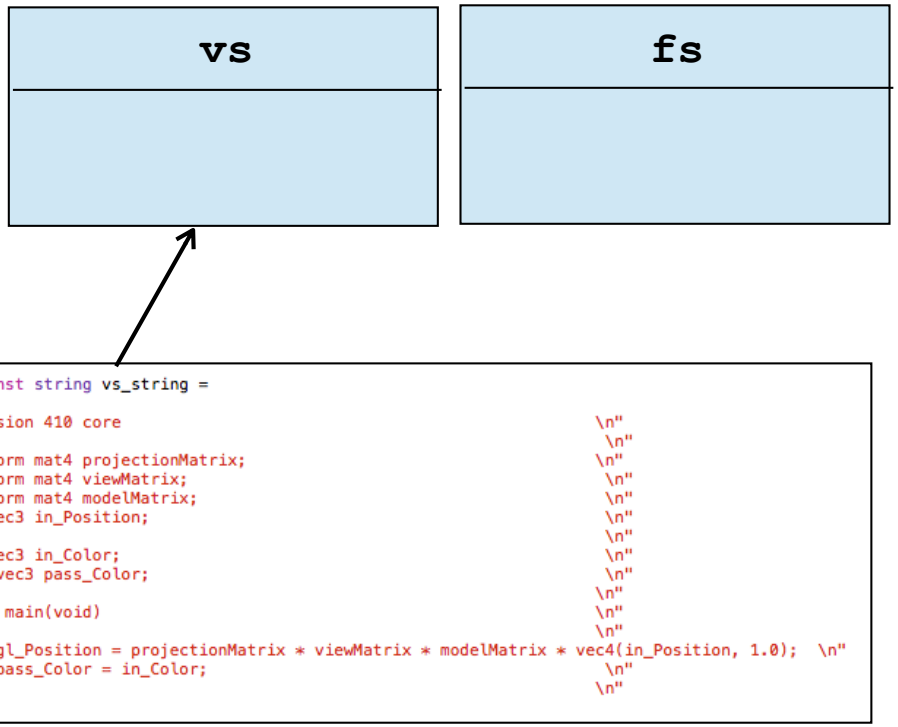
```
void glShaderSource(
GLuint shader,
GLsizei count,
const GLchar **string,
const GLint *length);
```

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);

                [.....]
```

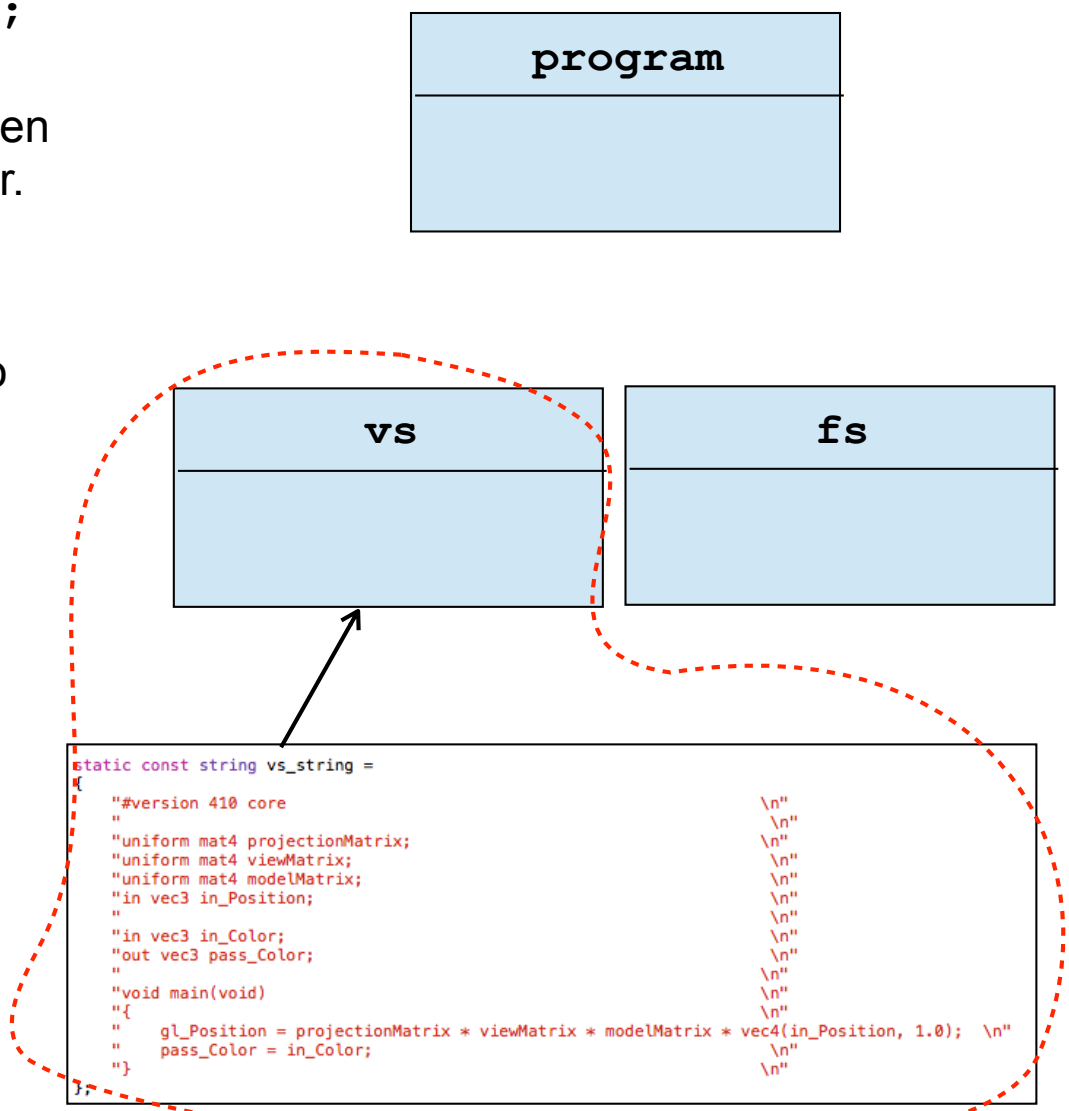
Shader Compiler

ARLAB

```
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

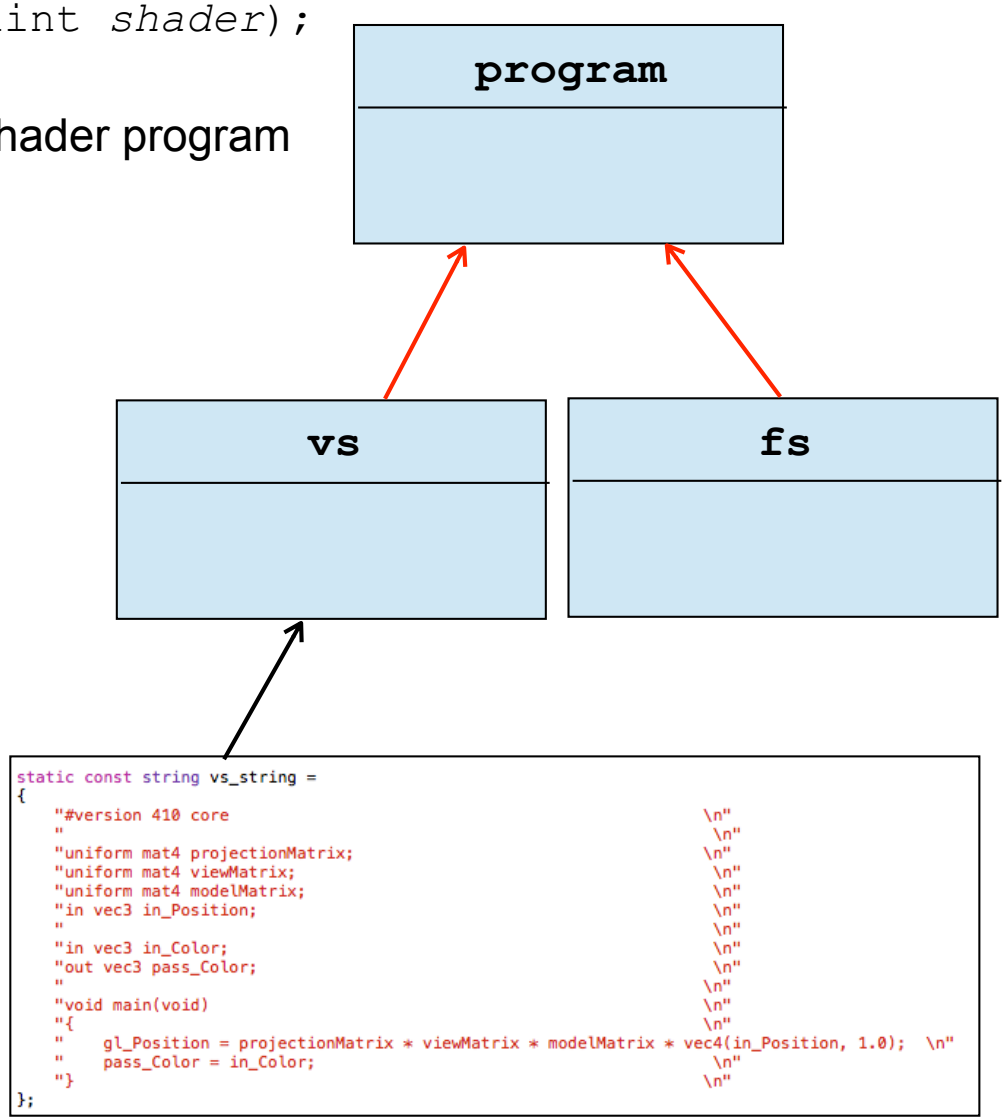
ARLAB

```
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

ARLAB

```
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

ARLAB

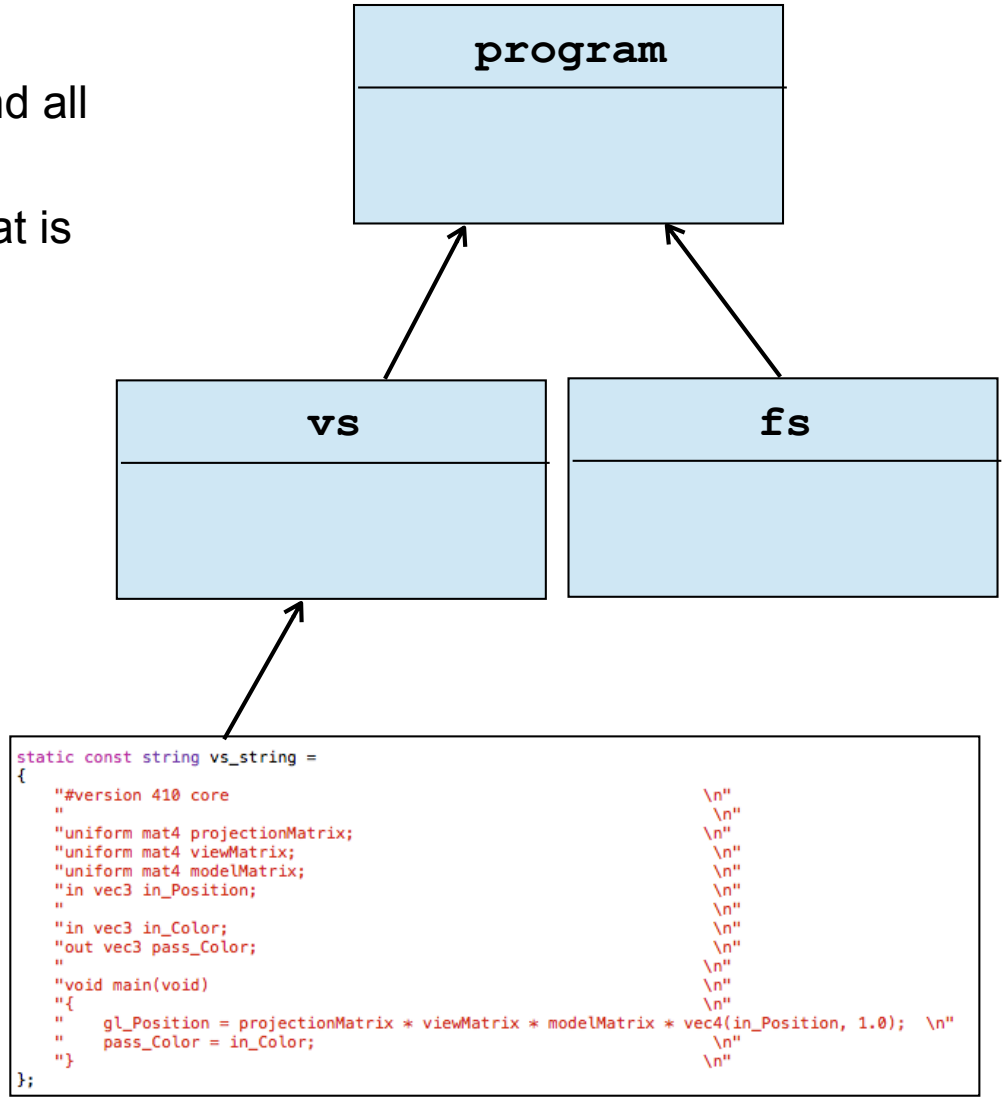
```
void glLinkProgram( GLuint program);
```

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

ARLAB

```
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

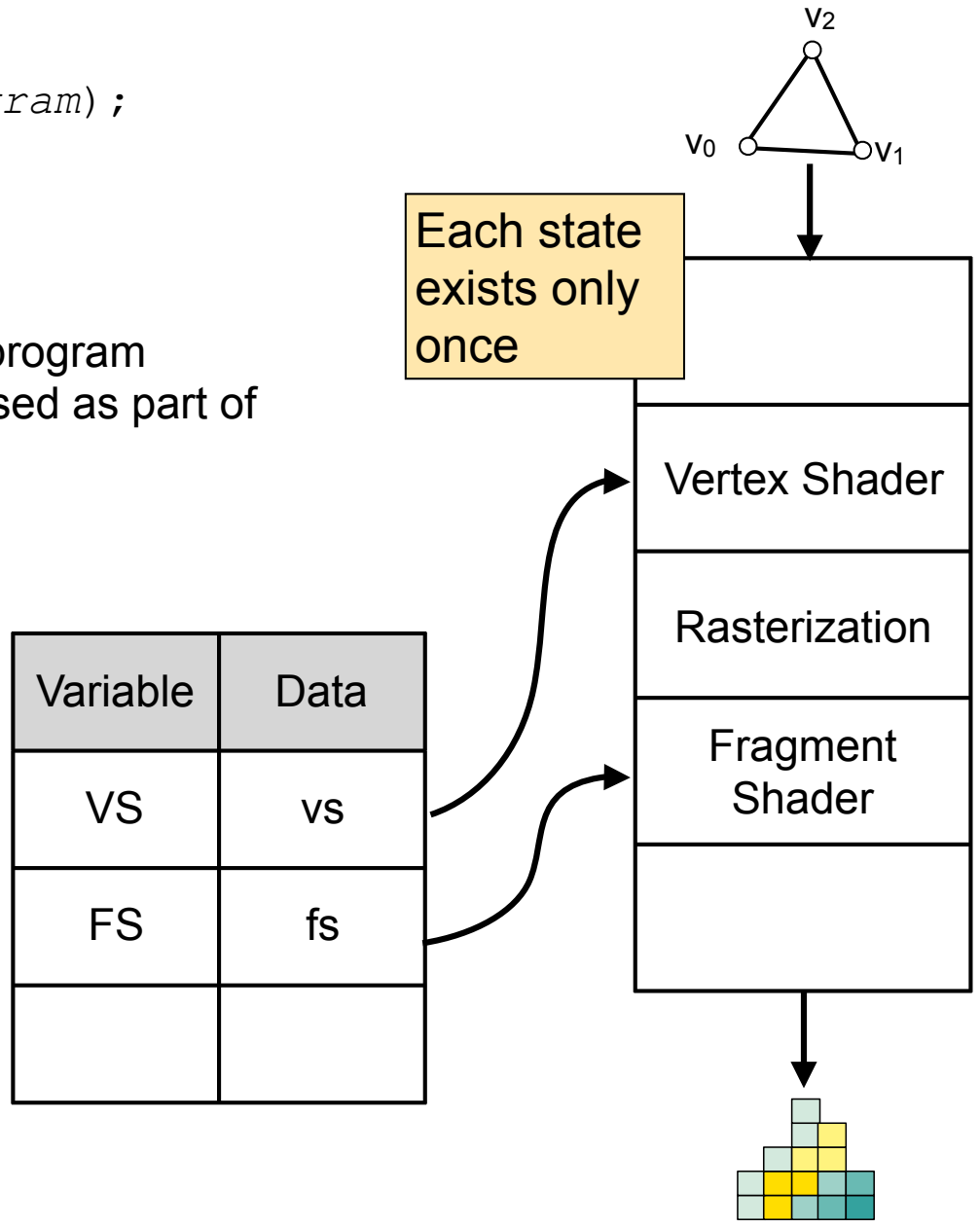
ARLAB

```
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

ARLAB

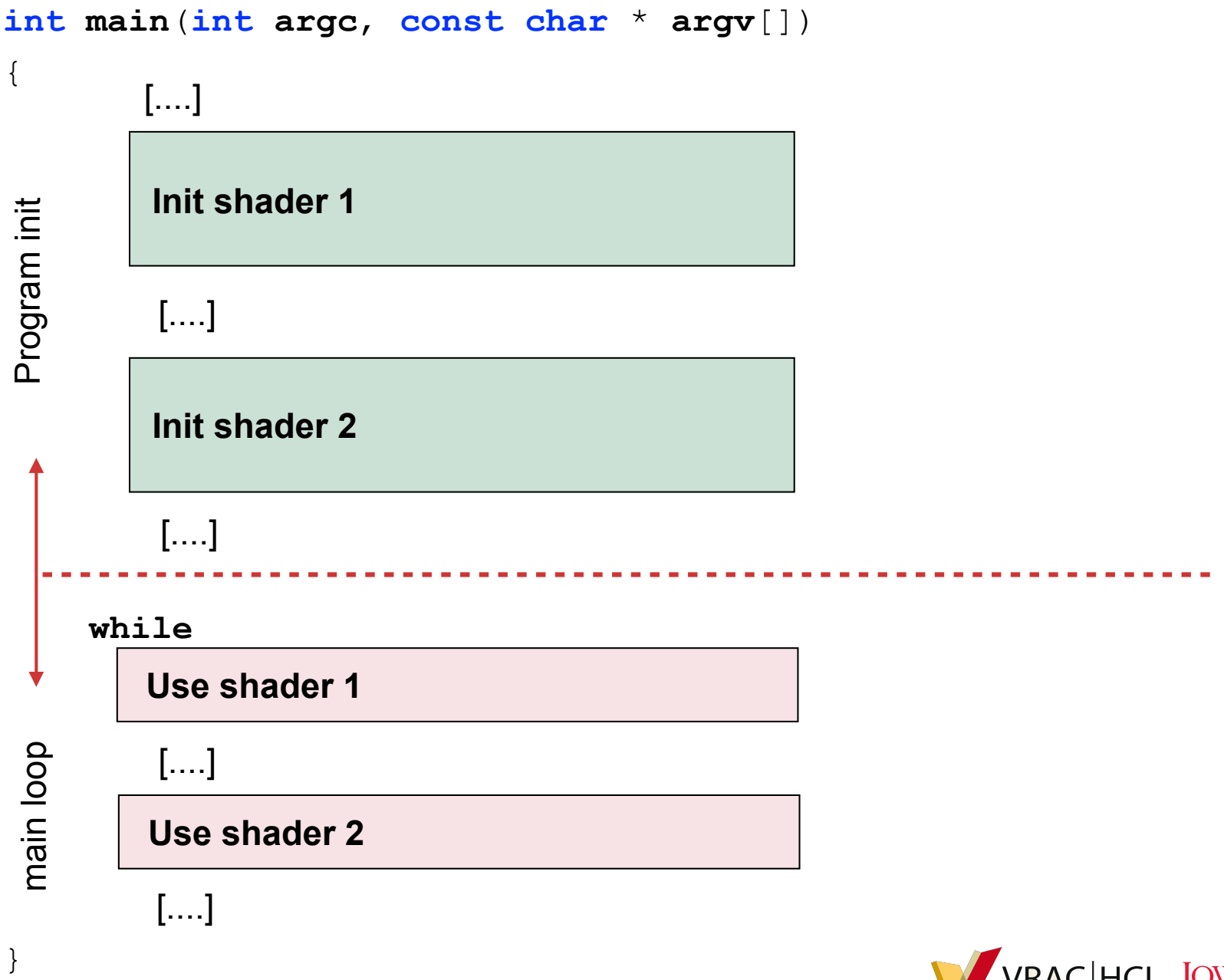
Init

```
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

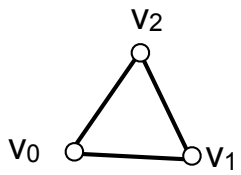
```
glUseProgram(program);
```

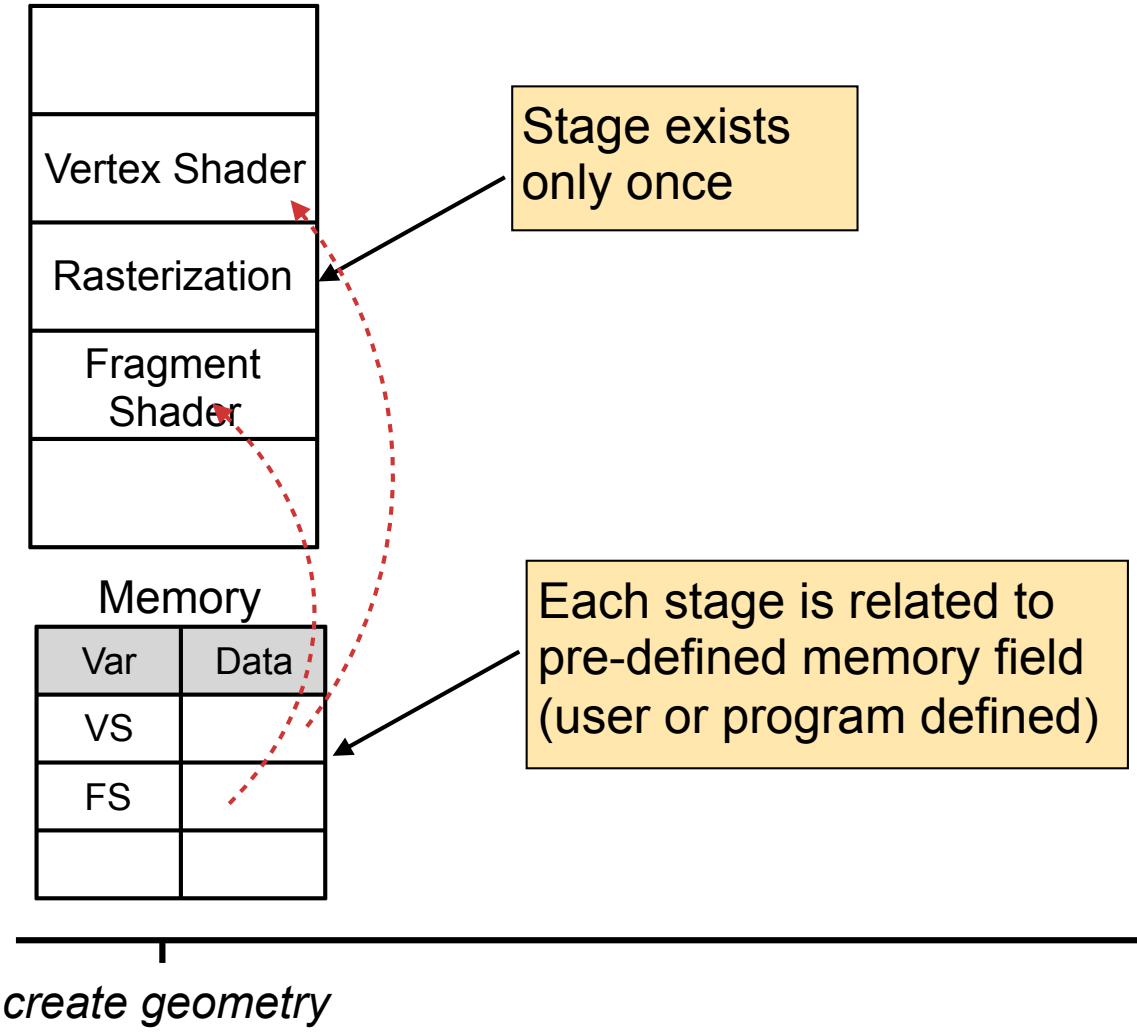
Program Structure



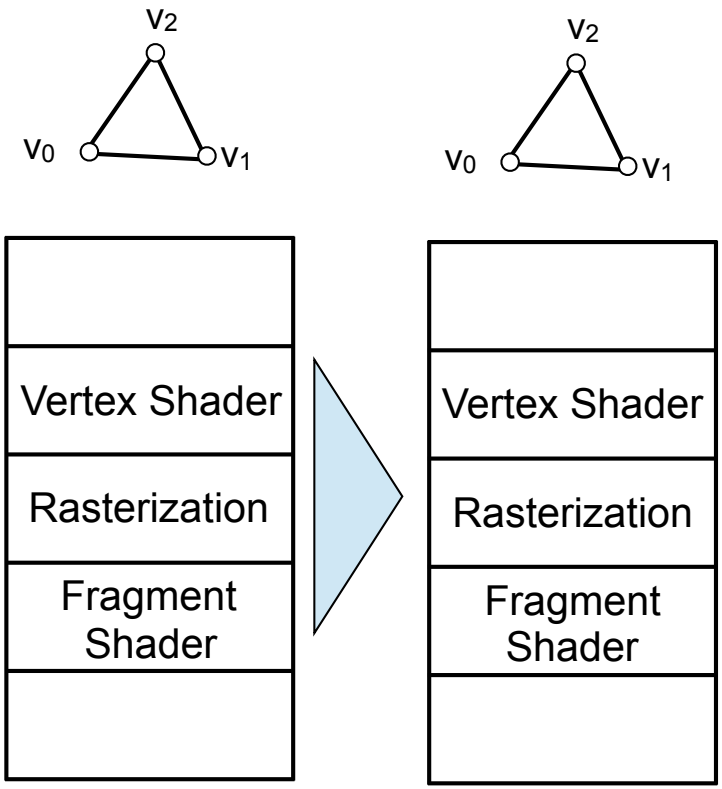
Runtime

Process

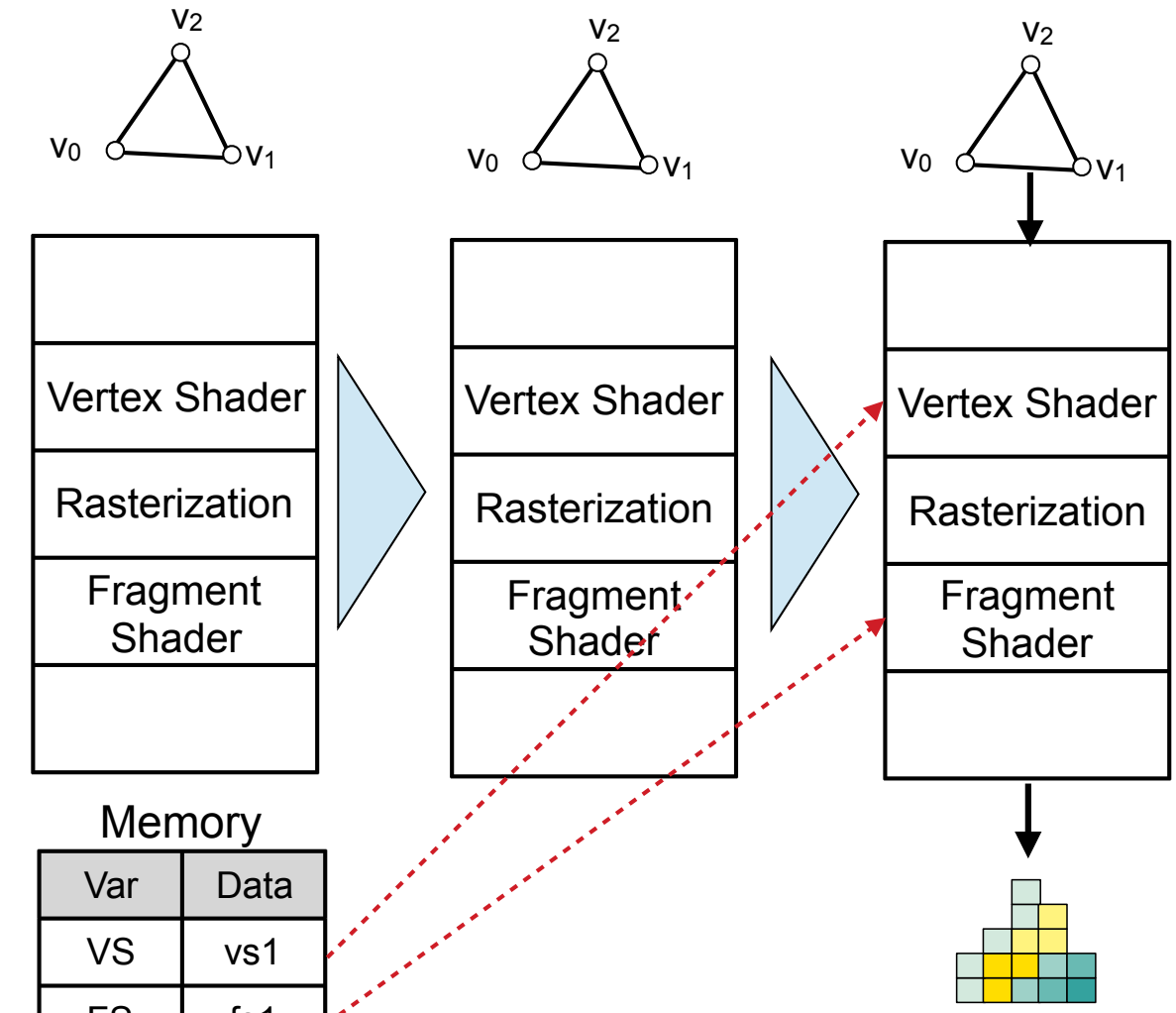

v2
v0
v1


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

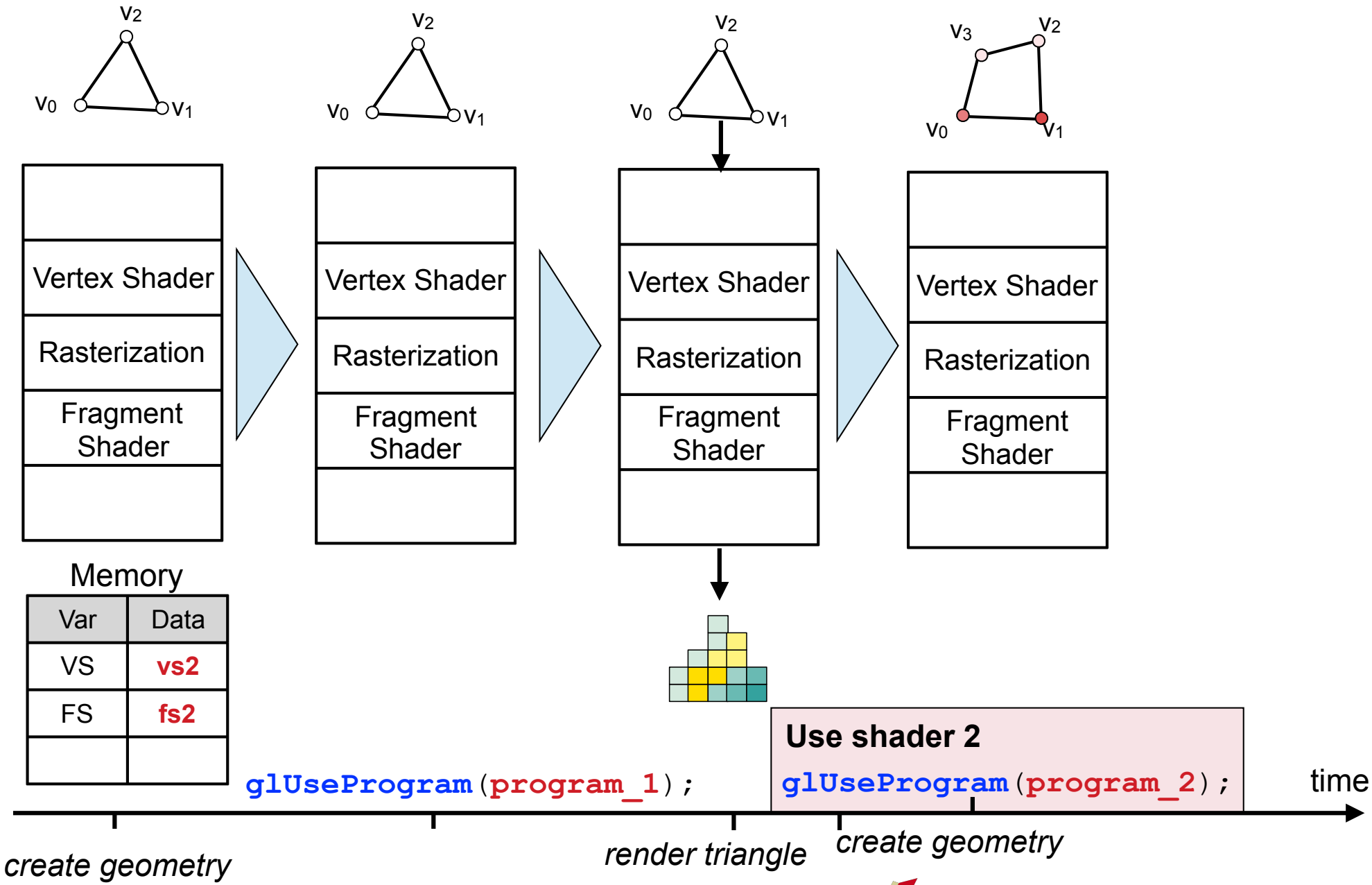
Process


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

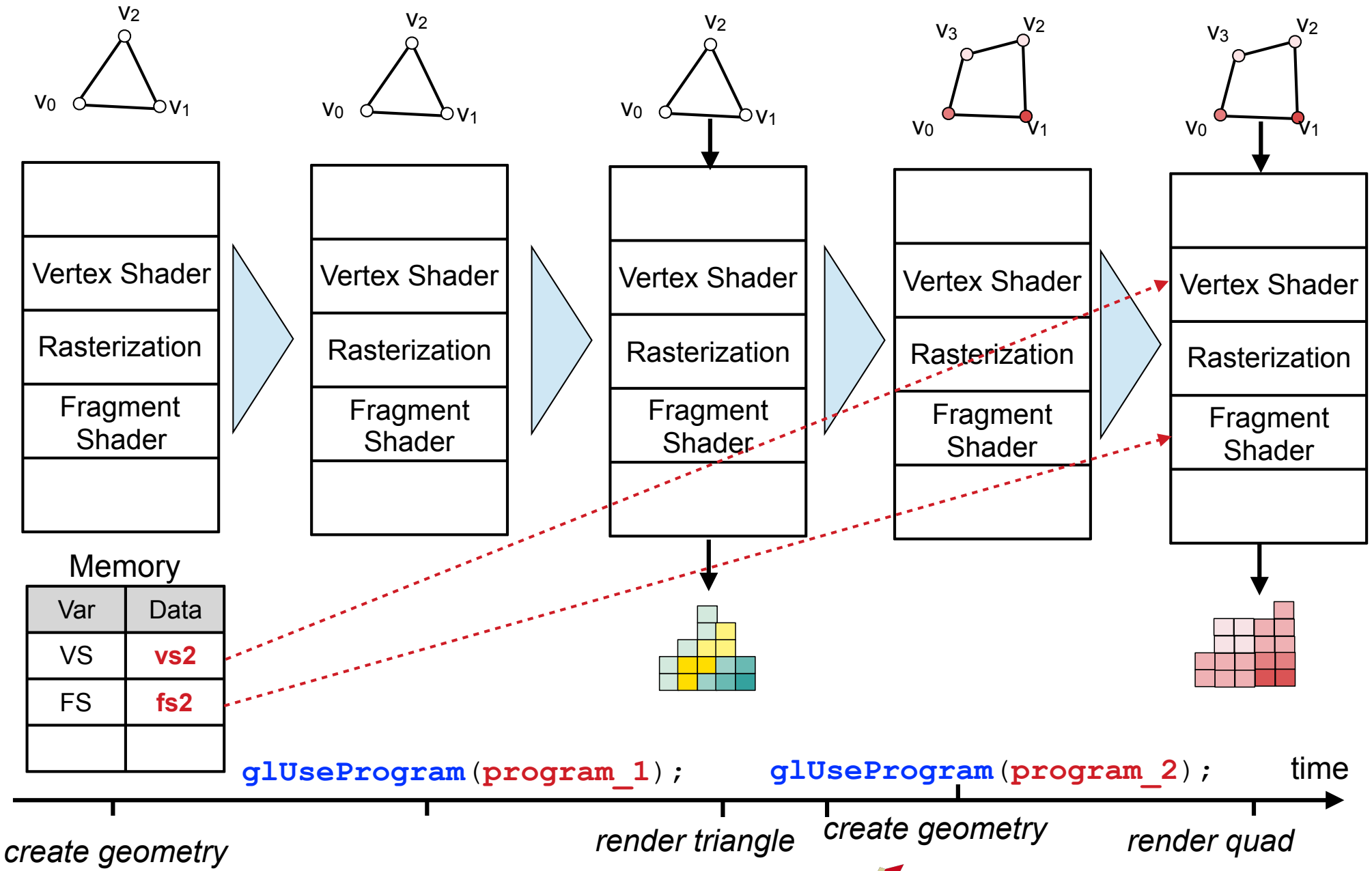
Process


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

Process

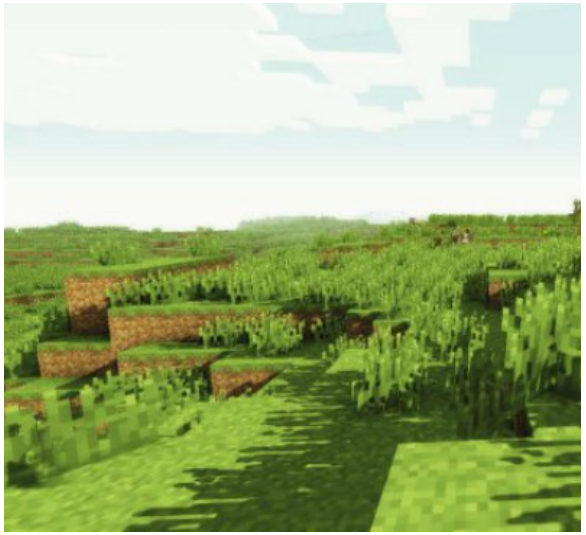

v2
v0
v1
v3
Vertex Shader
Rasterization
Fragment Shader
Memory
Var
Data
VS
vs2
FS
fs2
glUseProgram(program_1);
glUseProgram(program_2);
time
create geometry
render triangle
create geometry
render quad

ARLAB

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

Appearance Counts

ARLAB



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 ?

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