Programmable Rendering Pipeline

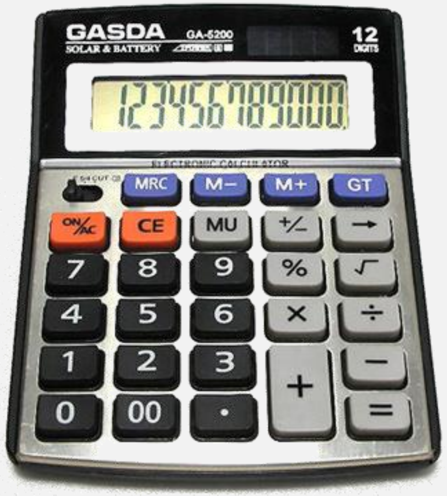
김준호
Visual Computing Lab.
국민대학교 소프트웨어학부

Programmable Rendering Pipeline (Part 1)

Programmable Rendering Pipeline

- What is the programmable rendering pipeline?

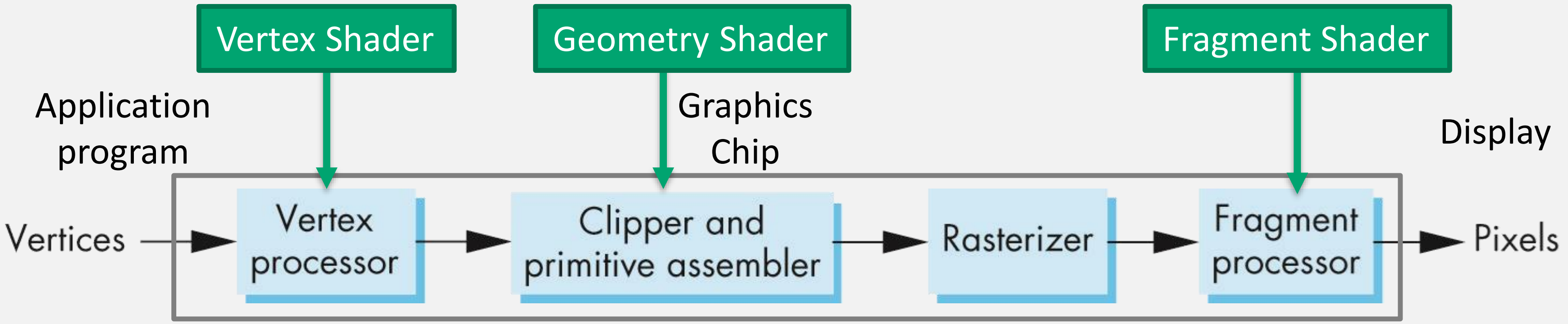
Fixed rendering pipeline : **Programmable** rendering pipeline

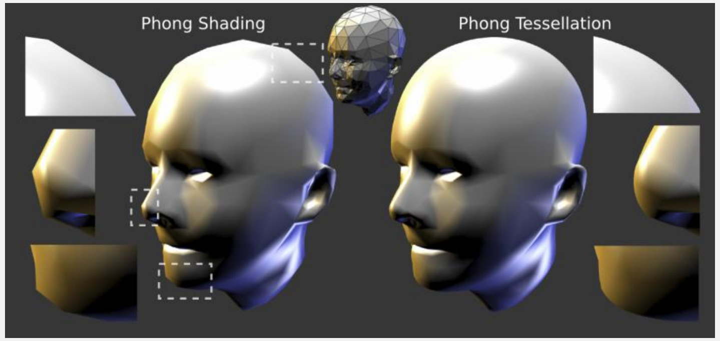
= 

: 

Programmable Rendering Pipeline

- Function units in rendering pipeline can be programmed with *shader* language
  - We can programming the functionality of rendering pipeline units





[Boubekeur and Alexa, Siggraph Asia 2008]

OpenGL Shading Language (GLSL)

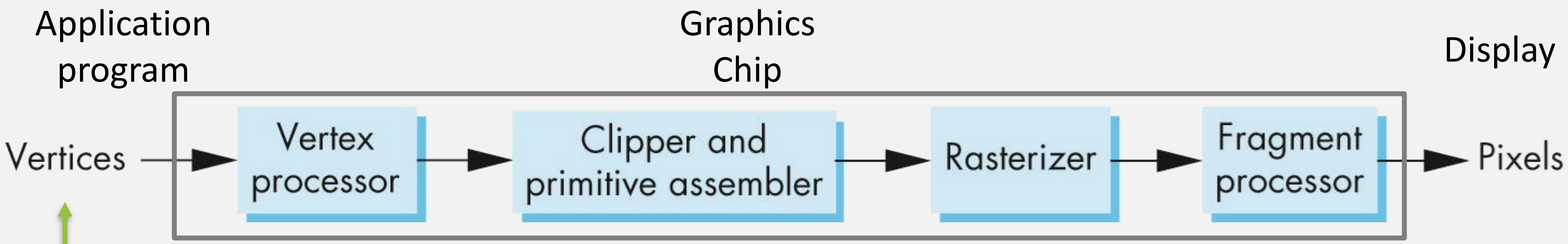
- OpenGL Shading Language
  - Part of OpenGL 2.0 or higher
  - High level C-like language
  - New data types
    - Matrices
    - Vectors
    - Samplers
  - OpenGL state available through built-in variables

GLSL Version	OpenGL Version
N/A	1.x
1.10.59	2.0
1.20.8	2.1
1.30.10	3.0
1.40.08	3.1
1.50.11	3.2
3.30.6	3.3
4.00.9	4.0
4.10.6	4.1
4.20.11	4.2
4.30.8	4.3
4.40	4.4

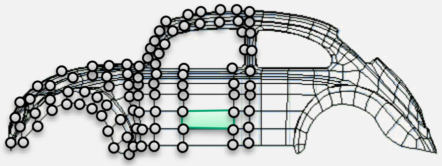
Understanding
Fixed Rendering Pipeline
– from the perspective of Programmable Rendering Pipeline

Overview of Rendering Pipeline

- Pipeline architecture
  – This is everything for interactive computer graphics!
    - First, we focus on the *fixed rendering pipeline*
  – Mechanism: a *state* machine
    - All information for image formations should be specified

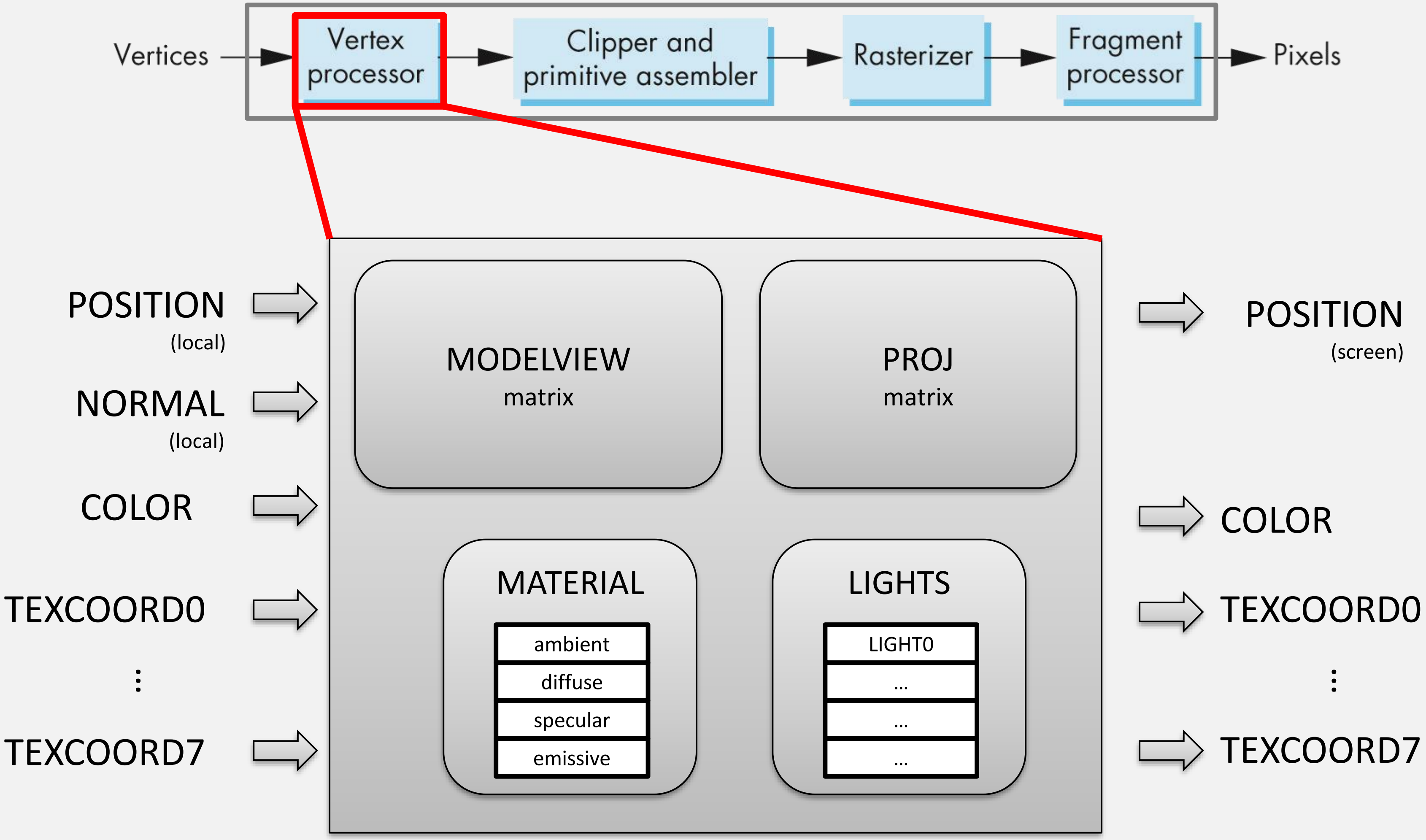


Positions: (x, y ,z), …
Normals: (nx, ny, nz), …
Texcoords: (s, t), …
Primitive type: Quads

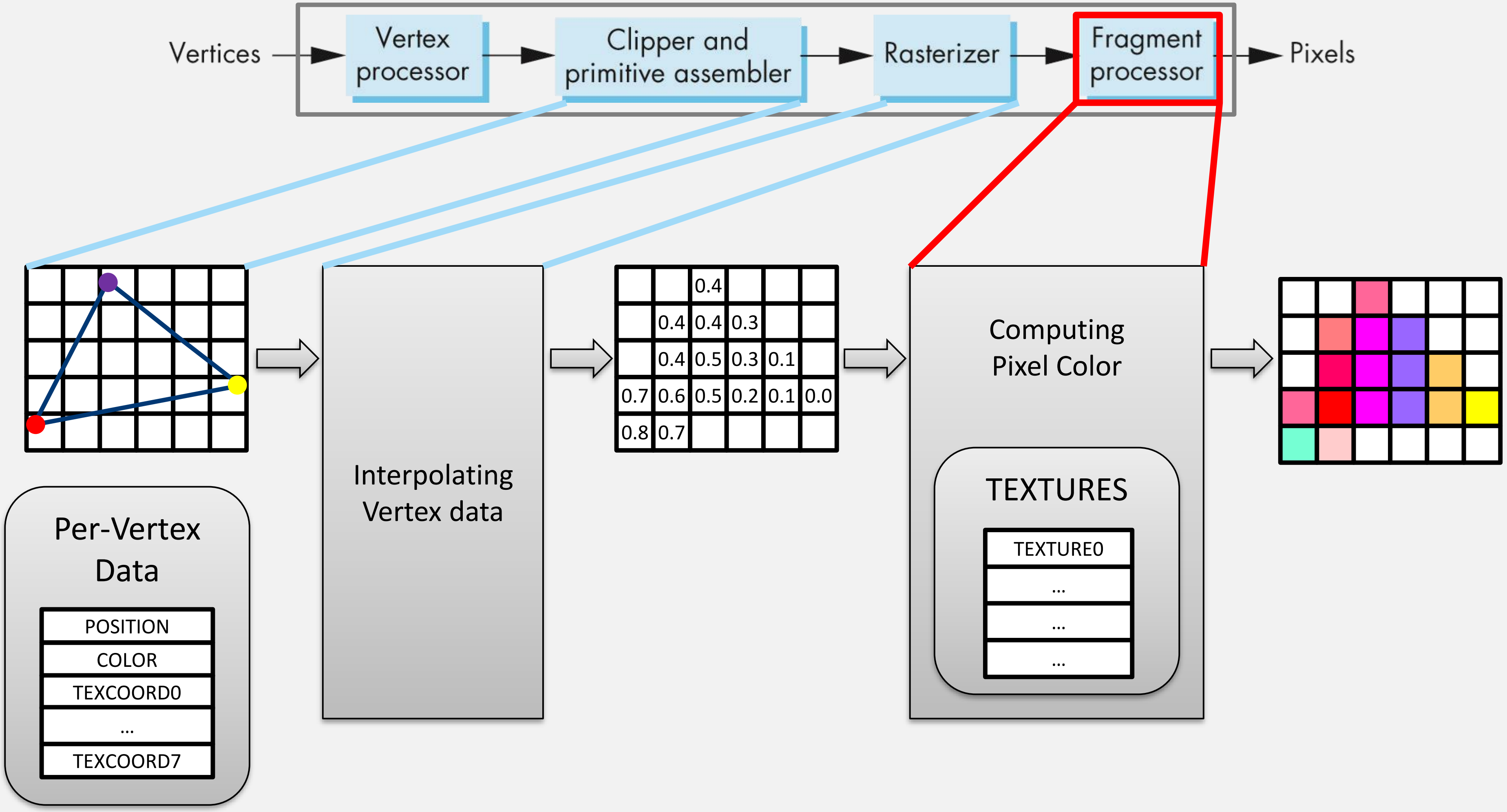


Our input:
In general, artists make it for programmers
(* Maya, 3D studio max, Blender, etc.)

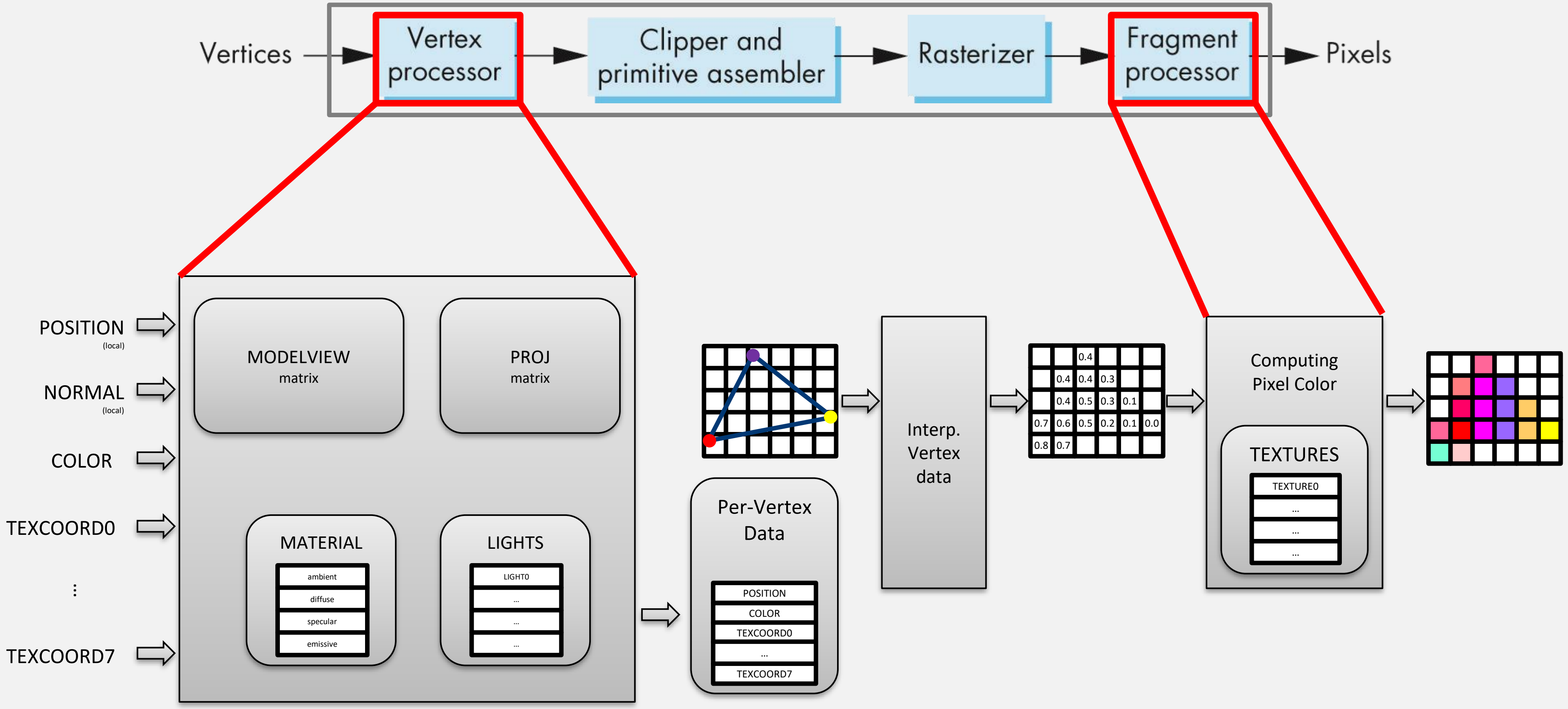
Vertex Processing w/ Fixed Rendering Pipeline



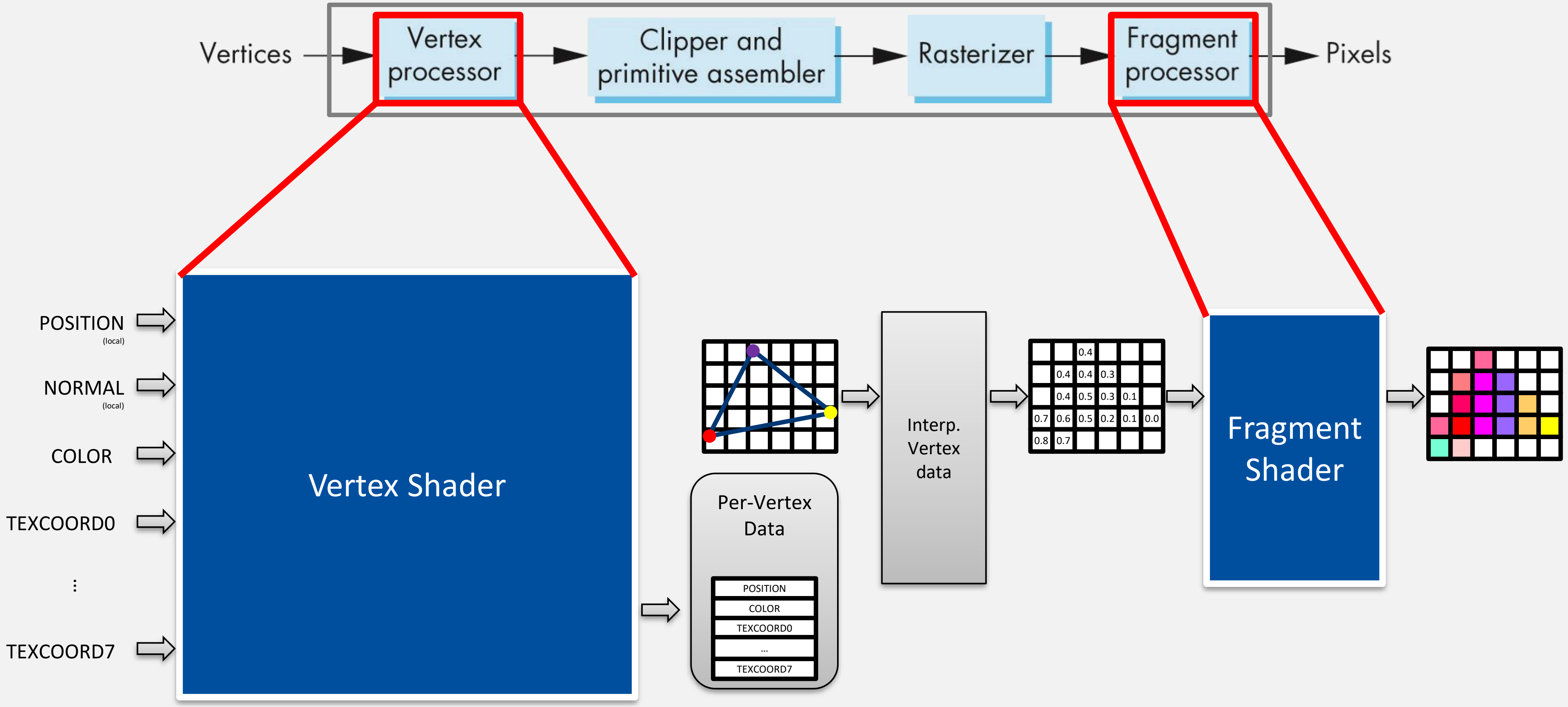
Fragment Processing w/ Fixed Rendering Pipeline



Fixed Rendering Pipeline



Programmable Rendering Pipeline



Limitation on Fixed Rendering Pipeline

Per-vertex Lighting v.s. Per-pixel Lighting

- Fixed rendering pipeline only supports per-vertex lighting
  - Computation of lighting is performed in vertex processor only
  - We may ignore specular effects from highlights, with coarse triangles
- Programmable rendering pipeline supports per-fragment lighting
  - We can program in a such a way that computation of lighting is performed in fragment processor
  - We can represent specular effects from highlights, with coarse triangles

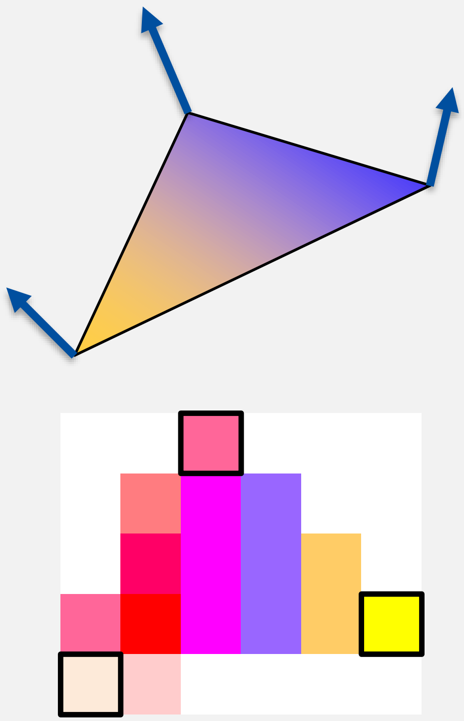
Per-vertex lighting

per-fragment lighting

Per-vertex Lighting v.s. Per-pixel Lighting

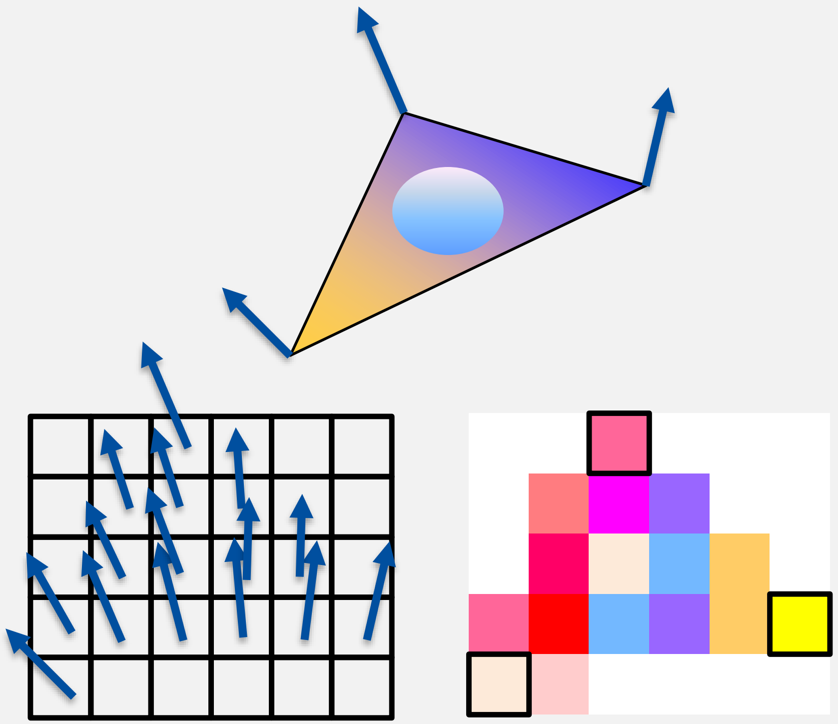
Per-vertex lighting

- Computation of lighting is performed in vertex processor only

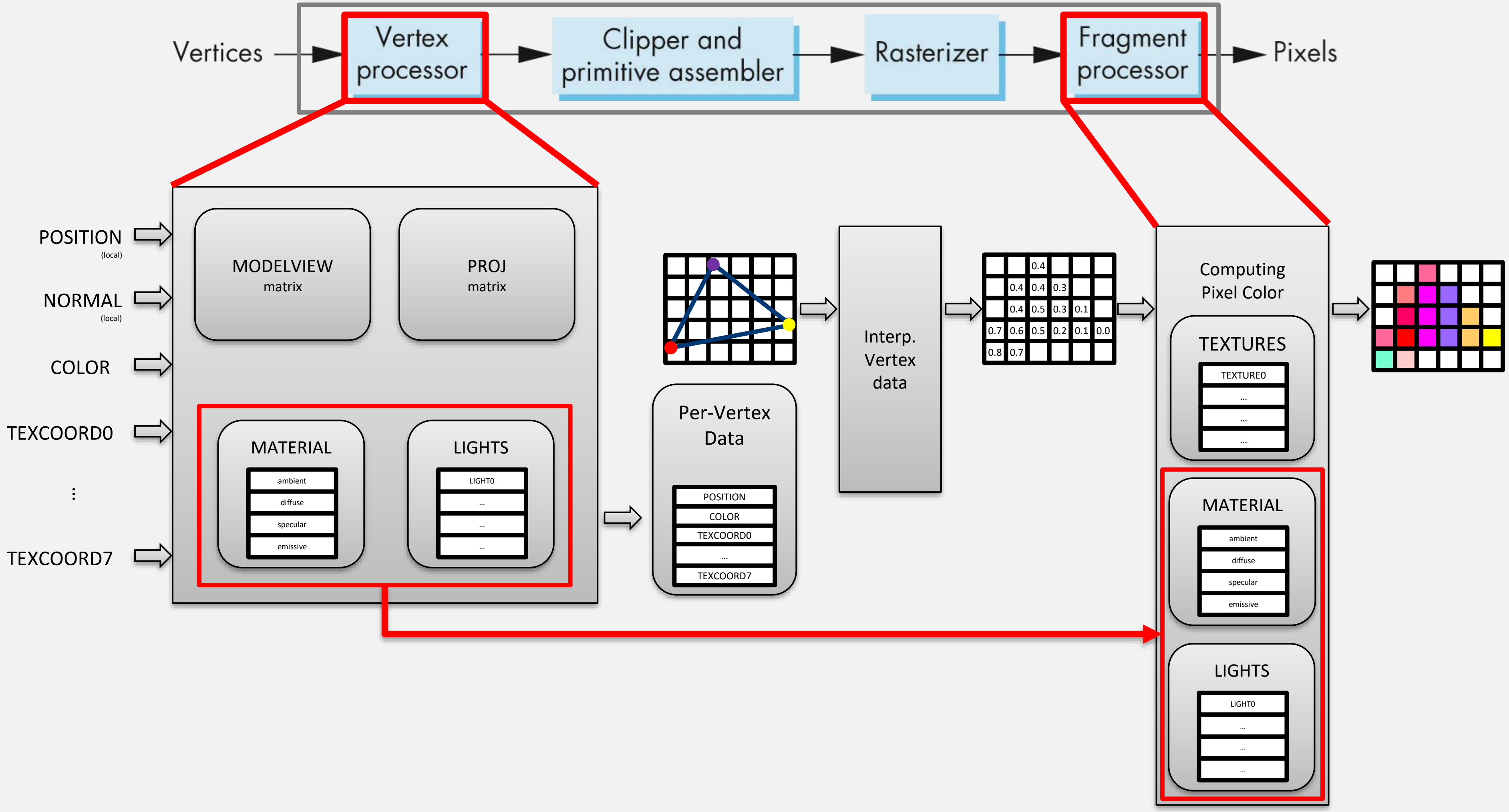


Per-fragment lighting

- Computation of lighting CAN be performed in fragment processor



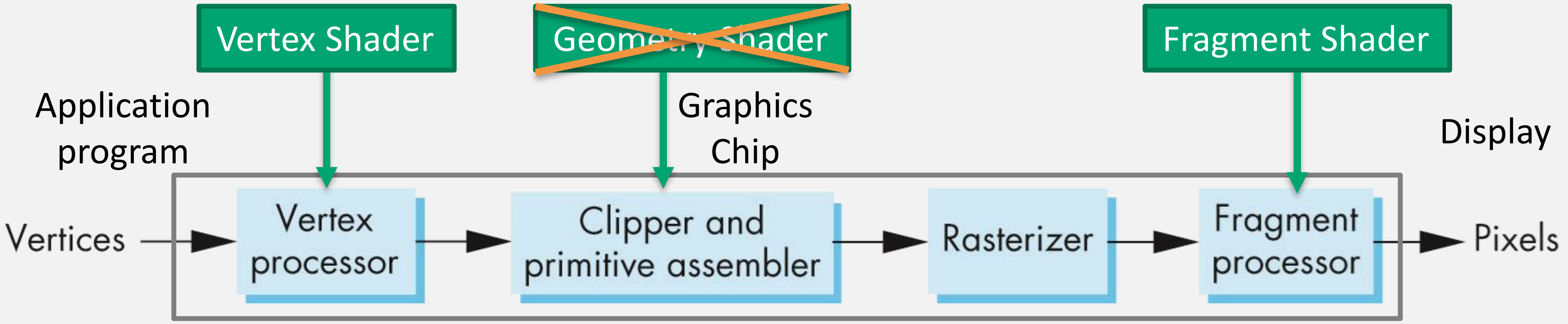
Per-pixel Lighting w/ Programmable Rendering Pipeline



Shader Programming at a Glance

Programmable Rendering Pipeline in OpenGL

- Function units in rendering pipeline can be programmed with *shader* language
  - We can programming the functionality of rendering pipeline units
- OpenGL w/ GLSL 1.2 only supports two types of shaders
  - Vertex shader & fragment shader



GLSL Programming at a glance

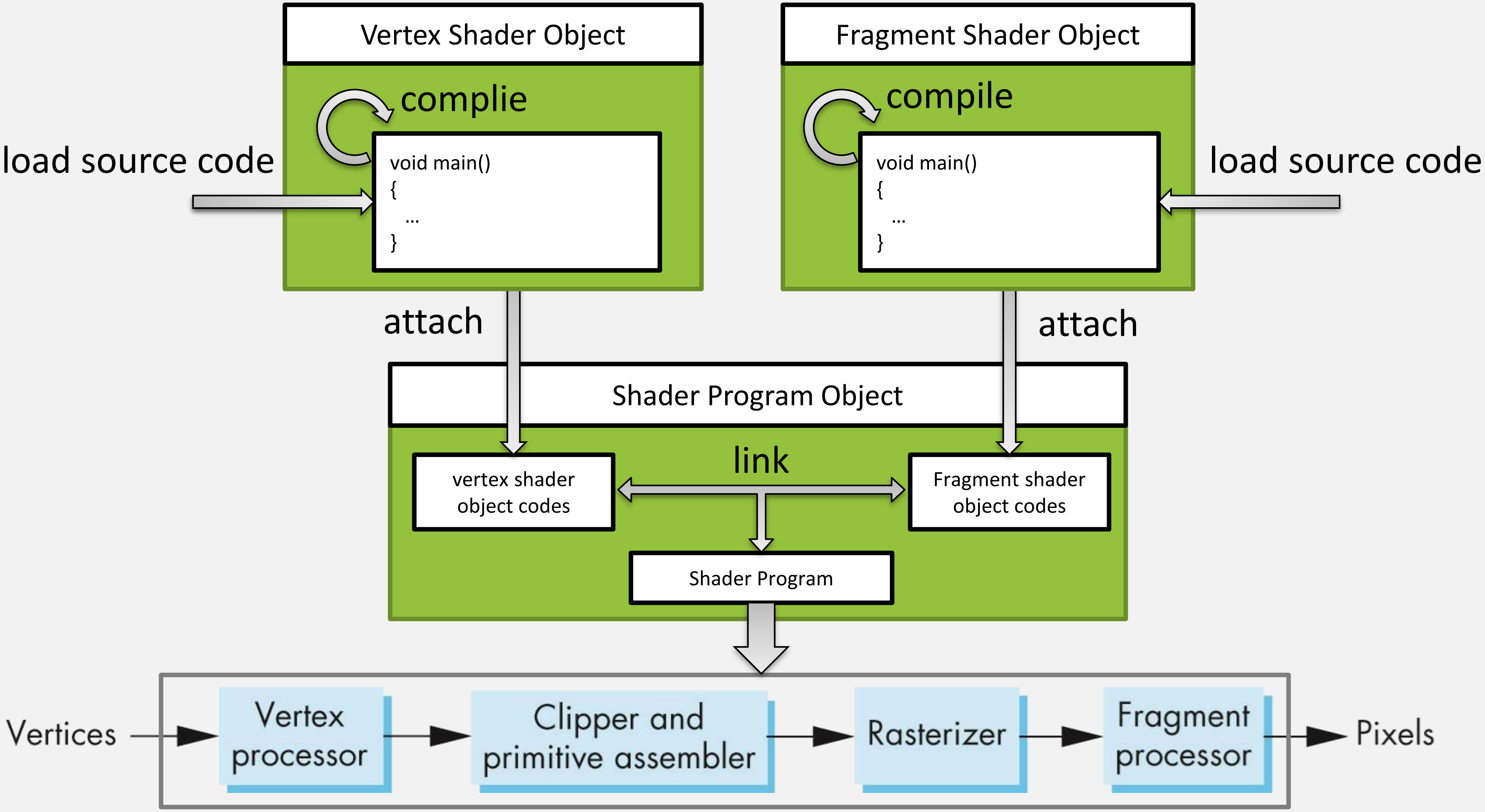
Vertex Shader

```glsl
#version 120                  // GLSL 1.20

uniform mat4 u_PVM;           // Proj * View * Model

attribute vec3 a_position;    // per-vertex position (per-vertex input)
attribute vec3 a_color;       // per-vertex color (per-vertex input)

varying   vec3 v_color;       // per-vertex color (per-vertex output)

void main()
{
  gl_Position = u_PVM * vec4(a_position, 1.0f);
  v_color = a_color;
}
```

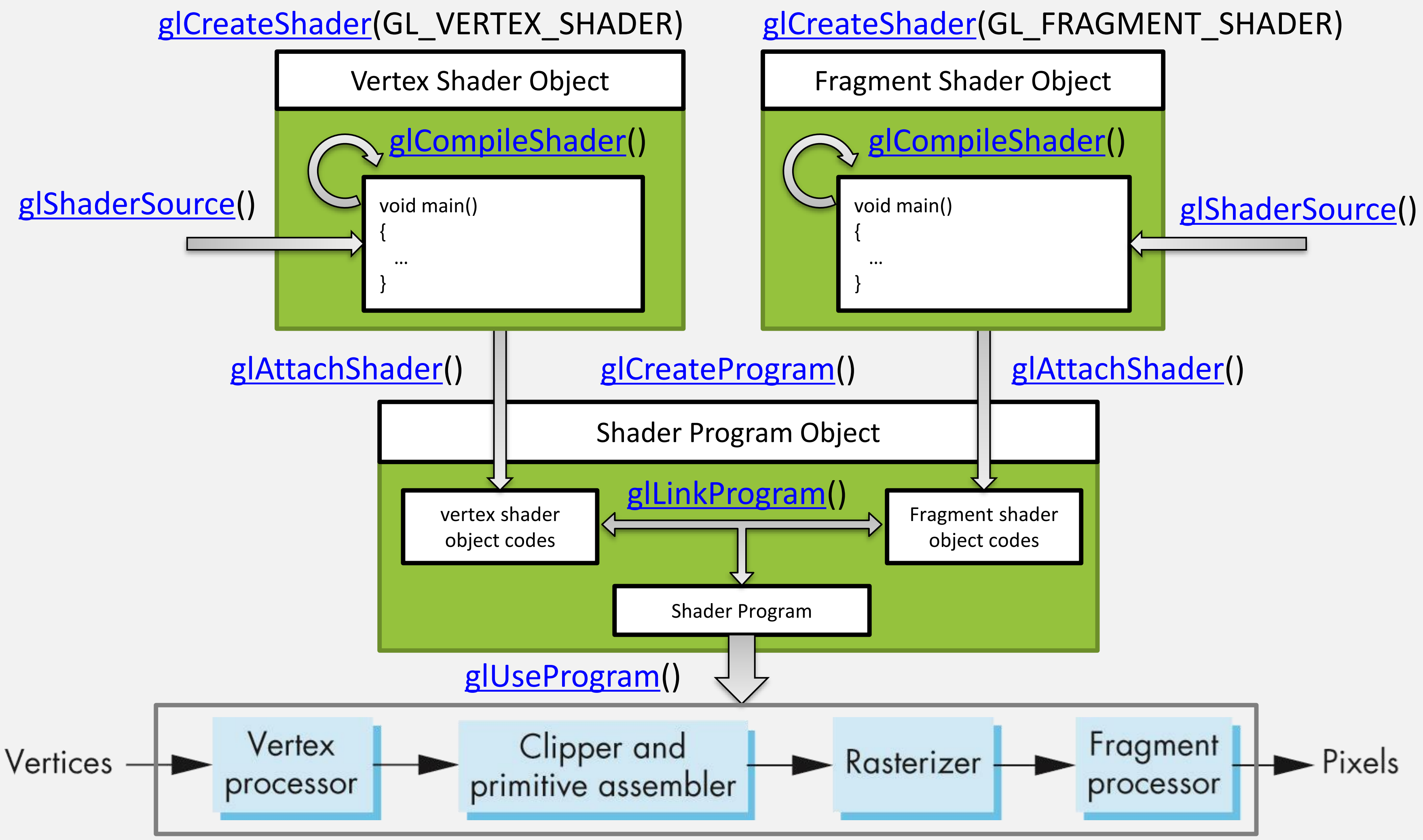
Fragment Shader

```glsl
#version 120                  // GLSL 1.20

varying vec3 v_color;         // per-fragment color (per-fragment input)

void main()
{
  gl_FragColor = vec4(v_color, 1.0f);
}
```

GLSL Programming at a glance



GLSL Programming at a glance



GLSL Programming at a glance

Initialization of Vertex Shader, Fragment Shader, and Program Object

```
void init_shader_program()
{
  GLuint vertex_shader
    = create_shader_from_file("./shader/vertex.glsl", GL_VERTEX_SHADER);

  std::cout << "vertex_shader id: " << vertex_shader << std::endl;
  assert(vertex_shader != 0);

  Gluint fragment_shader
    = create_shader_from_file("./shader/fragment.glsl", GL_FRAGMENT_SHADER);

  std::cout << "fragment_shader id: " << fragment_shader << std::endl;
  assert(fragment_shader != 0);

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

  std::cout << "program id: " << program << std::endl;
  assert(program != 0);

  loc_u_PVM = glGetUniformLocation(program, "u_PVM");

  loc_a_position = glGetAttribLocation(program, "a_position");
  loc_a_color = glGetAttribLocation(program, "a_color");
}
```

Rendering with Programmable Rendering Pipeline

```
void render_object()
{
  // Use a program
  glUseProgram(program);

  // Load uniforms (Here, for setting mat_PVM as Proj * View * Model)
  mat_PVM = mat_proj * mat_view * mat_model;
  glUniformMatrix4fv(loc_u_PVM, 1, GL_FALSE, mat_PVM);

  // Load attributes as per-vertex data (Here, we use VBO)
  glBindBuffer(…);
  glEnableVertexAttribArray(…);
  glVertexAttribPointer(…);

  // Draw with per-vertex data
  glDrawArrays(…);

  // …
}
```

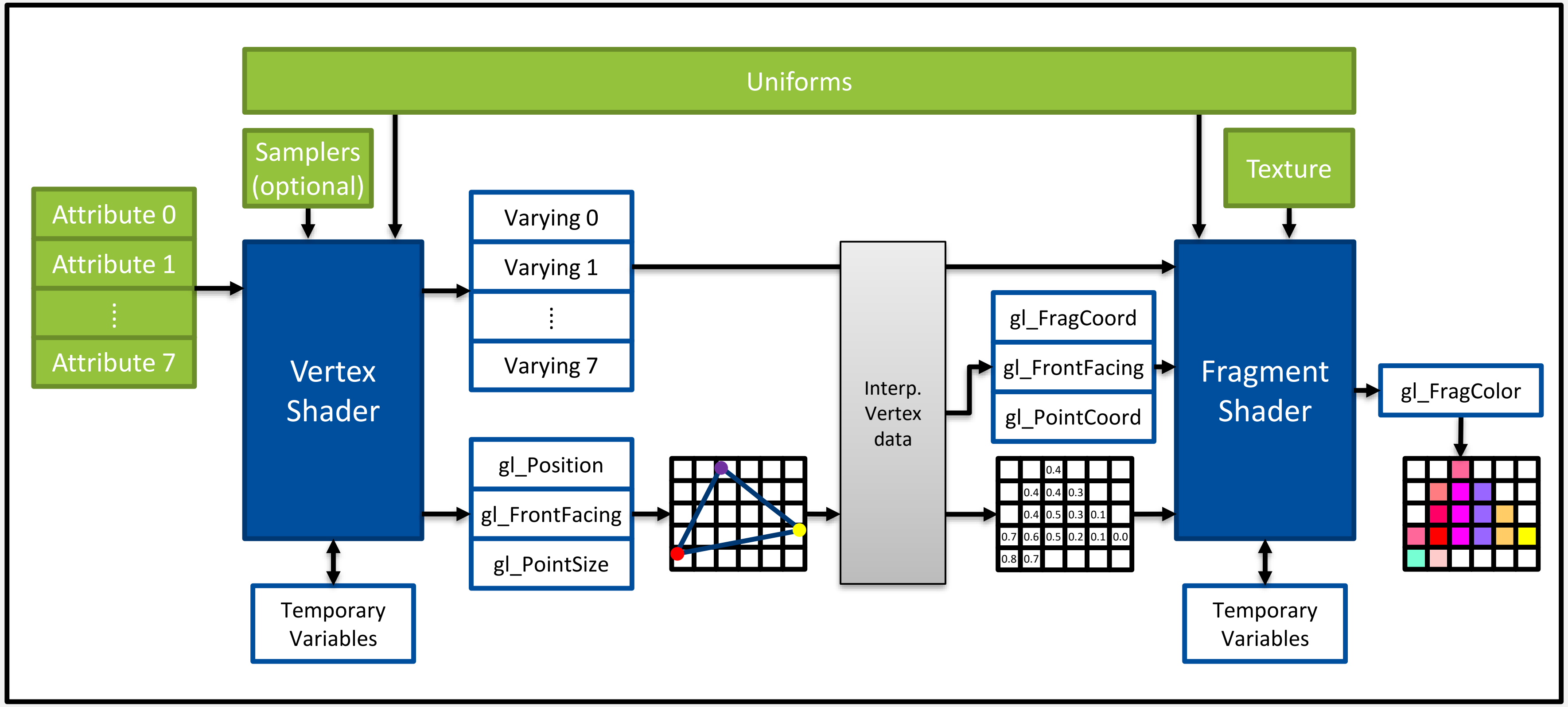
Summary of GLSL Programming at a Glance

- There are two shaders, and we can program the shader with GLSL 1.20 language
  - Vertex shader
  - Fragment shader
- Similar to developing typical programs, we have to
  - Compile vertex/fragment shader objects
  - Link them into a program object
- Similar to using typical programs, we can
  - Use the program object

- There are big changes in the rendering routine
  - Where is ~~glMatrixMode~~()?
    - You should handle the modelview/projection transformation in the vertex shader by yourself
  - Where is ~~glVertexPointer~~(), ~~glNormalPointer~~(), ~~glTexCoordPointer~~(), etc.?.
    - You should use glVertexAttribPointer() in Modern OpenGL
    - Every data associated with vertices is considered as one of vertex attributes in Modern OpenGL

More about Shader Programming

Programmable Rendering Pipeline – Modern OpenGL



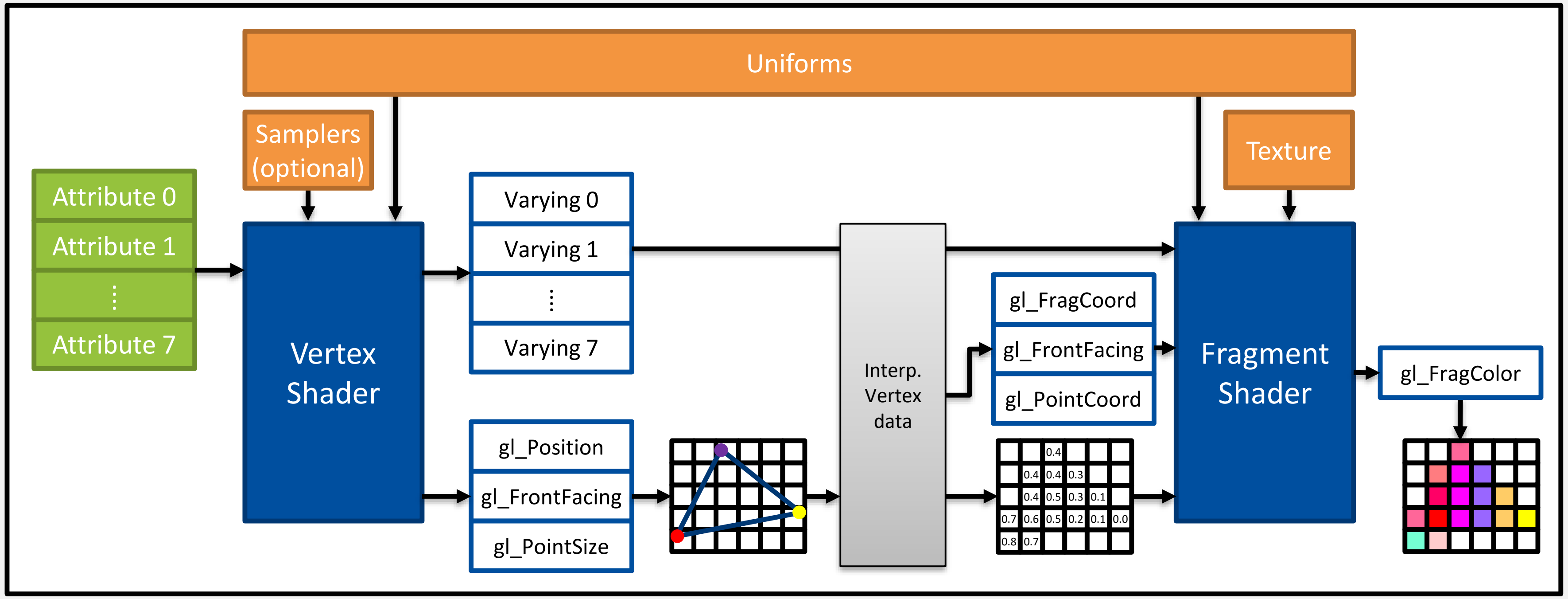
I/O Storage Qualifiers in GLSL 1.2

- Three types of I/O storage qualifiers in shader
  - Uniforms
    - Declare global variables whose values are the same across the entire shaders
  - Attributes
    - Declare variables that are passed to a vertex shader from OpenGL on a per-vertex basis
  - Varyings
    - Variables that provide the interface between the vertex shader, the fragment shader, and the fixed functionality between them

- Built-in variables and generic variables are available
  - Built-in type: OpenGL pre-defined constants and uniform state
  - Generic type: User-defined variables

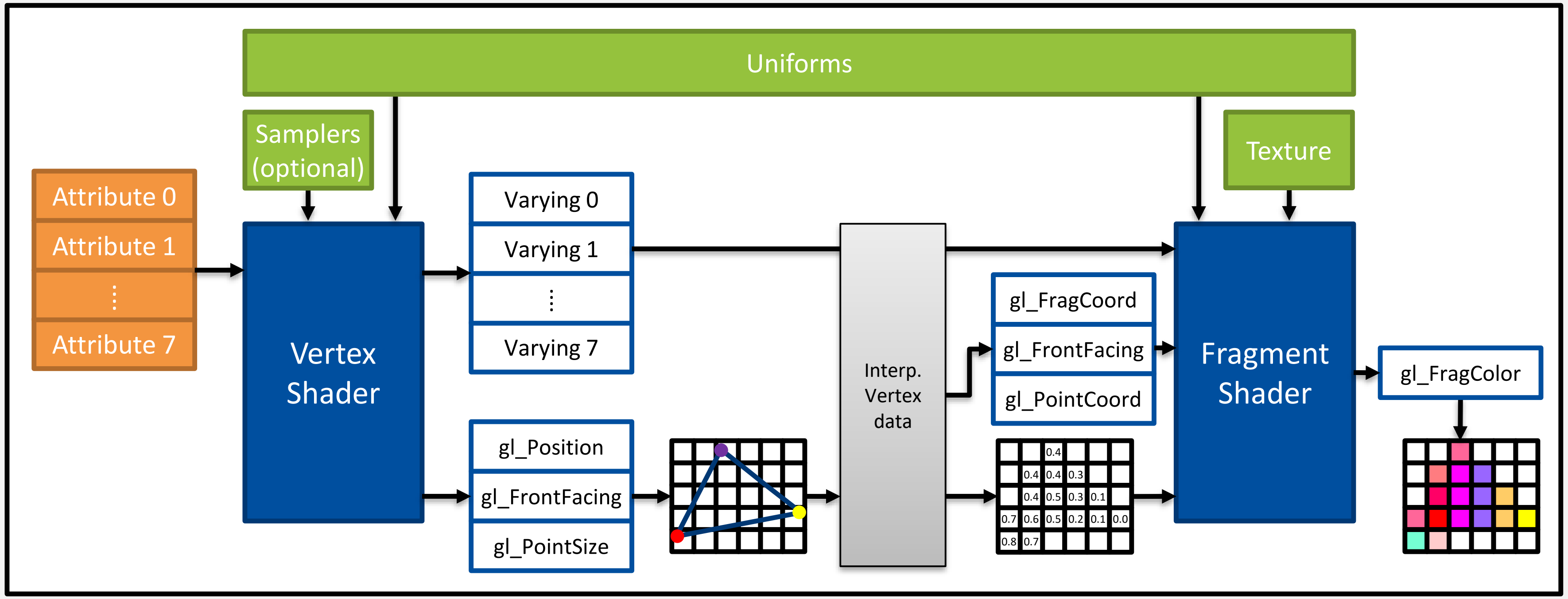
Uniform – I/O Storage Qualifiers

- Uniforms
  - Declare values, which do not change during a rendering
  - Available in both of vertex shader and fragment shader
  - Read-only
    - Initialized either directly by an appliction via API commands or indirectly by OpenGL



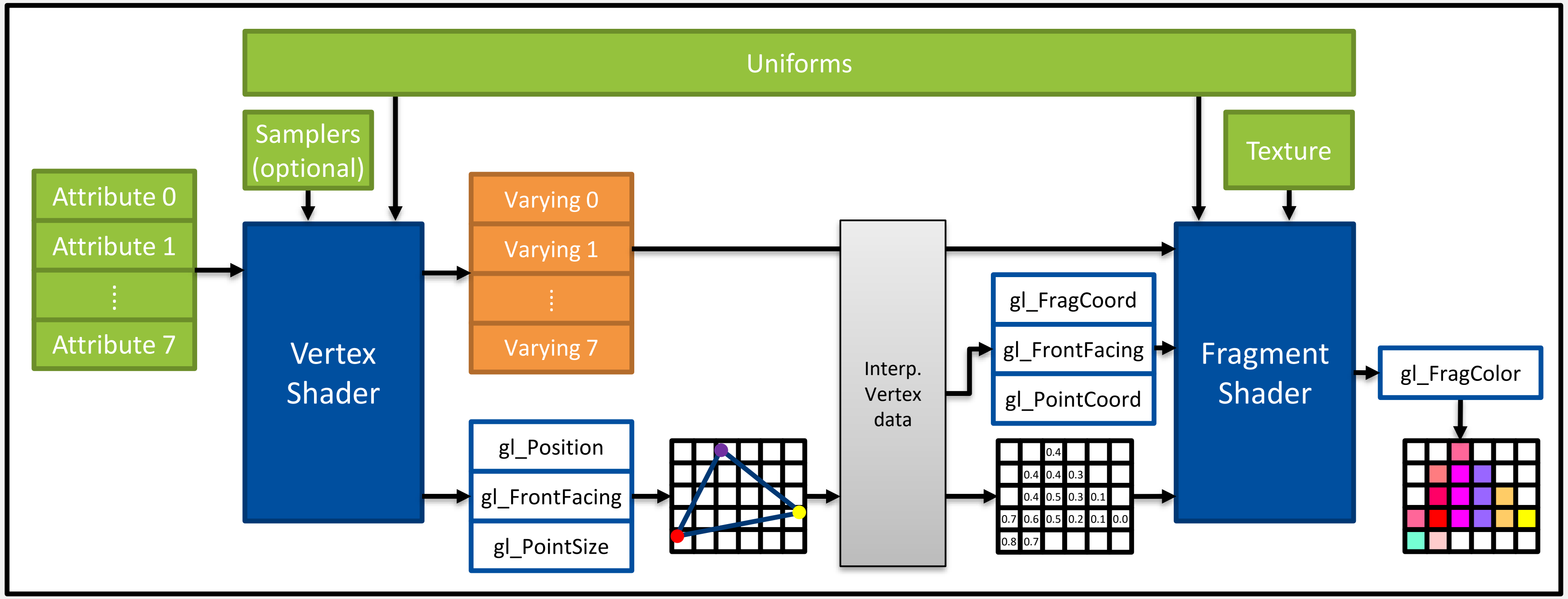
Attribute – I/O Storage Qualifiers

- Attributes
  - Declare values, which vary for per-vertex
  - Available in vertex shader only
  - Read only
    - Passed through the OpenGL vertex API or as part of a vertex array



Varying – I/O Storage Qualifiers

- Varyings
  - Used for passing data from vertex shader to fragment shader
  - Read/writable in vertex shader
  - Read-only in fragment shader

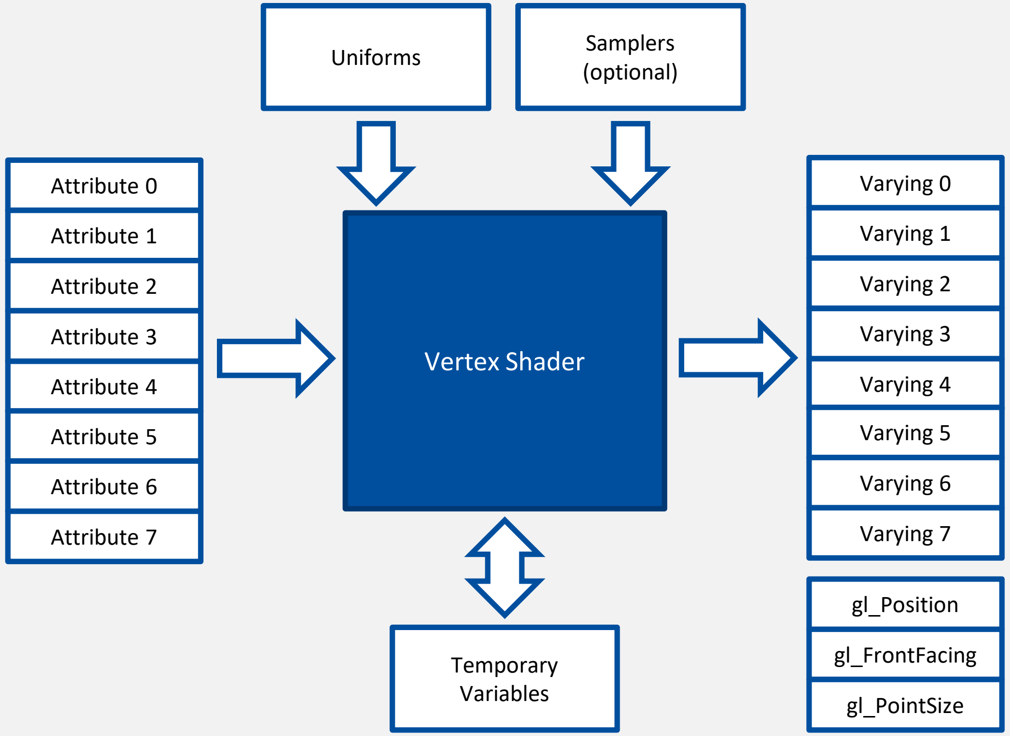


Data Types in OpenGL Shader

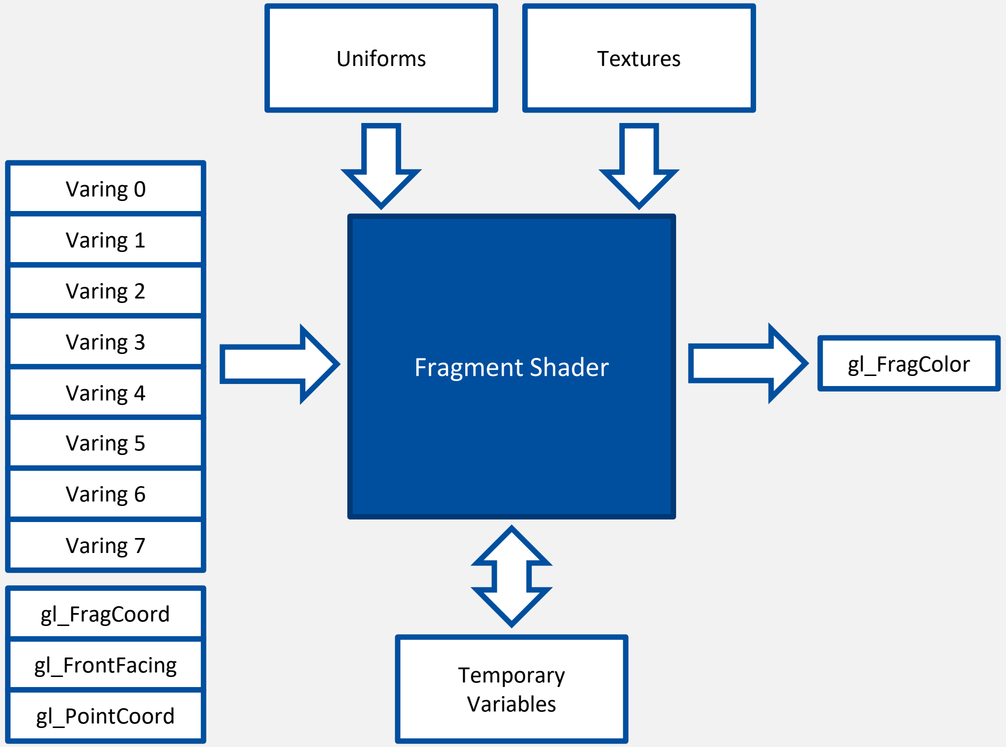
Variable Class	Types	Description
Scalars	**float**, **int**, **bool**	Scalar-based data types for floating-point, integer, and boolean values
Floating-point vectors	**float**, **vec2**, **vec3**, **vec4**	Floating-point-based vector types of one, two, three, or four components
Integer vector	**int**, **ivec2**, **ivec3**, **ivec4**	Integer-based vector types of one, two, three, or four components
Boolean vector	**bool**, **bvec2**, **bvec3**, **bvec4**	Boolean-based vector types of one, two, three, or four components
Matrices	**mat2**, **mat3**, **mat4**	Floating-point based matrices of size 2x2, 3x3, 4x4

I/O Types of Vertex/Fragment Shaders

I/O Types of Vertex Shader



I/O Types of Fragment Shader



Built-In Variables

I/O Types of Vertex Shader

- vec4 gl_Position
  - contains the position fo the current vertex
  - undefined after the vertex shading stages
- vec4 gl_FrontFacing
  - Indicates whether a primitive is front or back facing
- float gl_PointSize
  - contains size of rasterized points, in pixels
  - undefined after the vertex shading stages

I/O Types of Fragment Shader

- vec4 gl_FragCoord
  - contains the window-relative coordinates (x, y, z, 1/w) of the current framgment
- bool gl_FrontFacing
  - Indicates whether a primitive is front or back facing
- vec2 gl_PointCoord
  - contains the coordinates of a fragment within a point
  - ranges [0, 1]

Specifying Uniform / Vertex Attribute Data

Vertex / Fragment Shaders

Vertex shader

```
#version 120                  // GLSL 1.20

uniform mat4 u_PVM;           // Proj * View * Model

attribute vec3 a_position;    // per-vertex position (per-vertex input)
attribute vec3 a_color;       // per-vertex color (per-vertex input)

varying   vec3 v_color;       // per-vertex color (per-vertex output)

void main()
{
  gl_Position = u_PVM * vec4(a_position, 1.0f);
  v_color = a_color;
}
```

Fragment shader

```
#version 120                  // GLSL 1.20

varying vec3 v_color;         // per-fragment color (per-fragment input)

void main()
{
  gl_FragColor = vec4(v_color, 1.0f);
}
```

Modern OpenGL codes (C/C++)

```
GLint   loc_u_PVM, loc_a_position, loc_a_color;

void init_shader_program()
{
  // Create a shader program with a vertex shader and a fragment shader
  program = glCreateProgram();
  // …

  // Get uniform / vertex attribute locations
  loc_u_PVM = glGetUniformLocation(program, "u_PVM");
  loc_a_position = glGetAttribLocation(program, "a_position");
  loc_a_color = glGetAttribLocation(program, "a_color");
}

void render_object()
{
  // Use the progam
  glUseProgram(program);

  // Setting Proj * View * Model
  mat_PVM = mat_proj * mat_view * mat_model;
  glUniformMatrix4fv(loc_u_PVM, 1, GL_FALSE, mat_PVM);

  glEnableVertexAttribArray(loc_a_position);
  glVertexAttribPointer(loc_a_position, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);

  glEnableVertexAttribArray(loc_a_color);
  glVertexAttribPointer(loc_a_color, 3, GL_FLOAT, GL_FALSE, 0, (void*)0);

  // Draw triangles
  glDrawArrays(GL_TRIANGLES, 0, 3);
  // …
}
```

Specifying Uniform Data

- glGetUniformLocation() – return the location of a uniform variable

```
// program              specifies the program object to be queried
// name                 points to a null terminated string containing the name of the uniform variable
GLint glGetUniformLocation(GLuint program, const GLchar* name);
```

- glUniform() – specify the value of a uniform variable

```
// location specify the location of the uniform value to be modified
// v0, v1, v2, v3       specify the new values to be used for the specified uniform varialbe
void glUniform1f(GLint location, GLfloat v0);
void glUniform2f(GLint location, GLfloat v0, GLfloat v1);
void glUniform3f(GLint location, GLfloat v0, GLfloat v1, GLfloat v2);
void glUniform4f(GLint location, GLfloat v0, GLfloat v1, GLfloat v2, GLfloat v3);


// location specify the location of the uniform value to be modified
// count                specify the number of matrices that are to be modified (usually 1)
// transpose            specify whether it is transpose or not (MUST be GL_FLASE)
// value                specify a pointer to an array of count values
void glUniformMatrix2fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat* value);
void glUniformMatrix3fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat* value);
void glUniformMatrix4fv(GLint location, GLsizei count, GLboolean transpose, const GLfloat* value);
```

Specifying Vertex Attribute Data

- glGetAttribLocation() – return the location of an attribute variable

```
// program              specifies the program object to be queried
// name                 points to a null terminated string containing the name of the attribute variable
GLint glGetAttribLocation(GLuint program, const GLchar* name);
```

- glVertexAttribPointer() – define an array of generic vertex attribute data

```
// index                specifies the index of the generic vertex attribute to be modified
// size                 specifies the number of components per generic vertex attribute (Must be 1, 2, 3, or 4)
// type                 specifies the data type of each comp[onent in the array
                        (GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_FIXED, or GL_FLOAT)
// normalized           specifies wheter fixed-point data values should be normalized (GL_TRUE or GL_FALSE)
// stride               Specifies the byte offset between consecutive generic vertex attributes
// pointer              Specifies a pointer to the first component in the array
void glVertexAttribPointer(GLuint index, GLint size, GLenum type, GLboolean normalized,
                           GLsizei stride, const GLvoid* pointer);
```

- glEnableVertexAttribArray()/glDisableVertexAttribArray()

```
// index                specifies the index of the generic vertex attribute to be enabled or disabled
void glEnableVertexAttribArray(GLuint index);
void glDisableVertexAttribArray(GLuint index);
```