# Programmable Rendering Pipeline

김준호

Visual Computing Lab.

국민대학교 소프트웨어학부

# Programmable Rendering Pipeline (Part 2)

# Programmable Rendering Pipeline

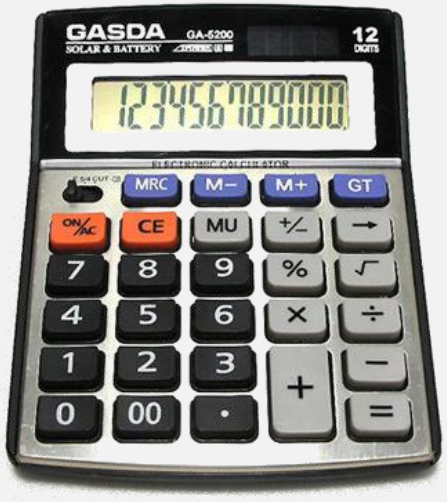
- What is the programmable rendering pipeline?

**Fixed**
rendering pipeline

:

**Programmable**
rendering pipeline

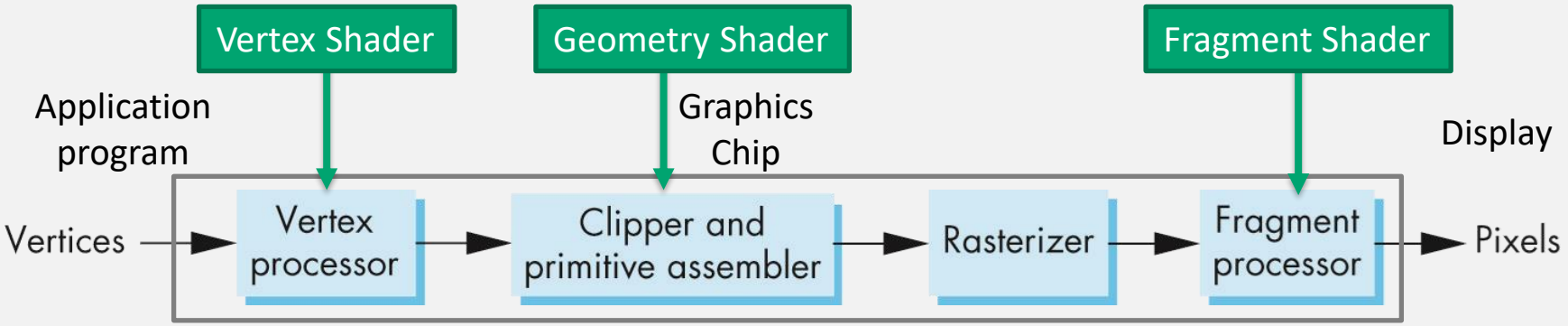
=

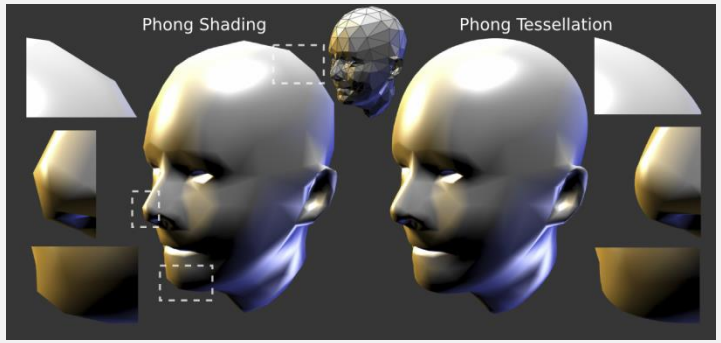



:

# 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)

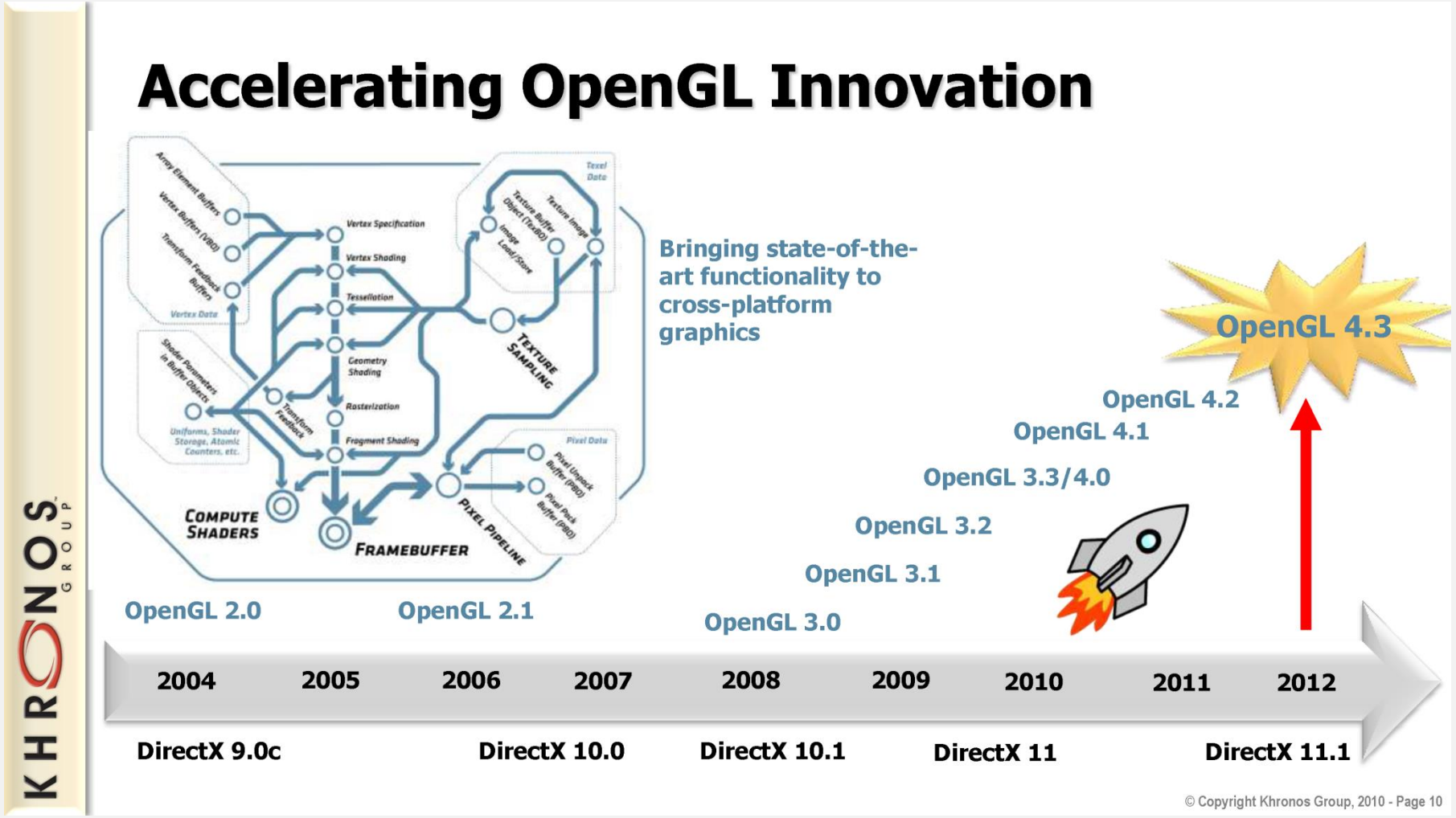
- OpenGL Shading Language
  - Part of OpenGL 2.0 or higher
  - High level C-like language
  - New data types
    - Matrices
    - Vectors
    - Samplers
  - OpenGL state available through built-in variables

| 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 |

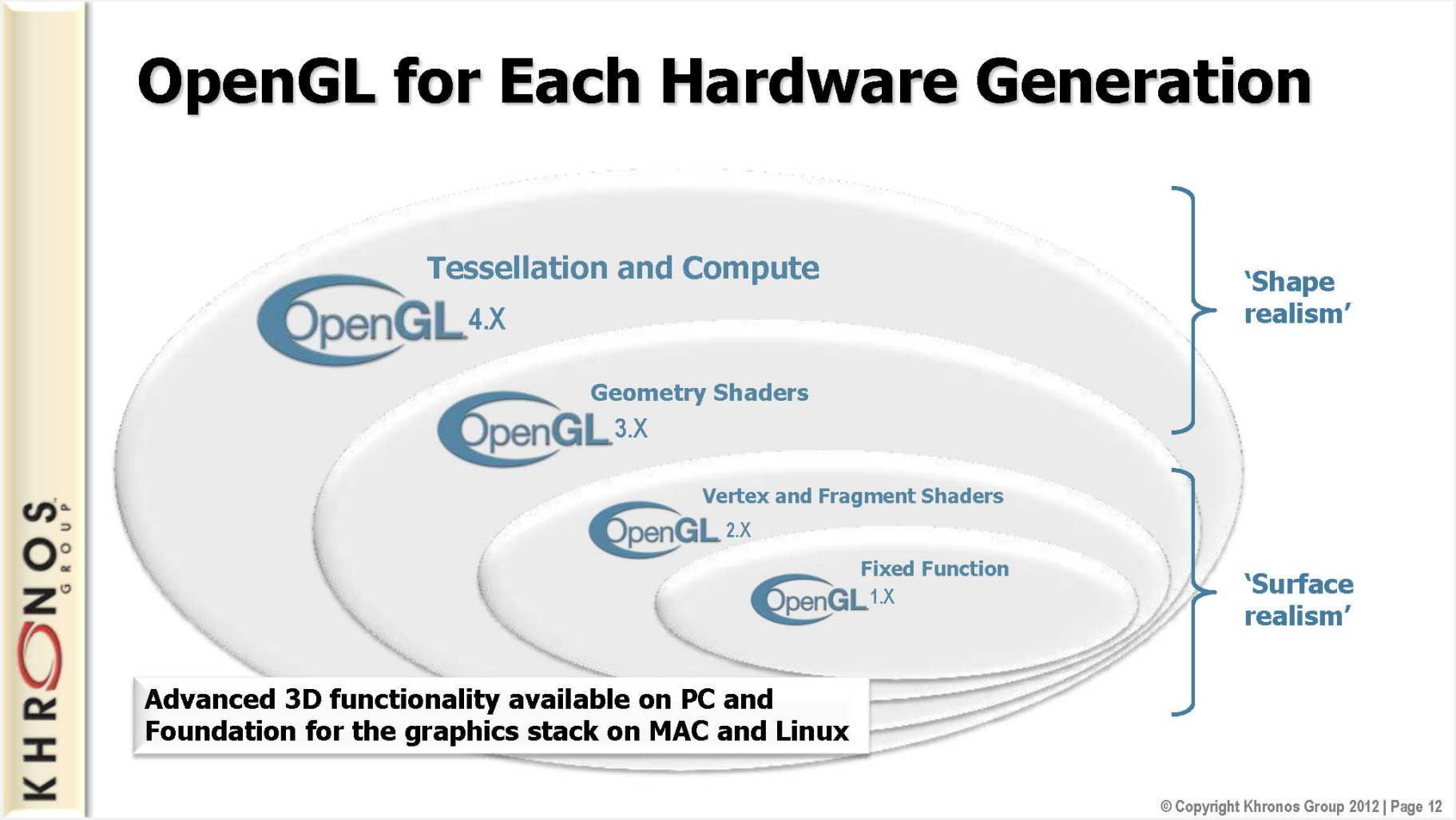
- **OpenGL**
- OpenGL ES

# OpenGL History at a Glance

# History of OpenGL



# History of OpenGL – Summary

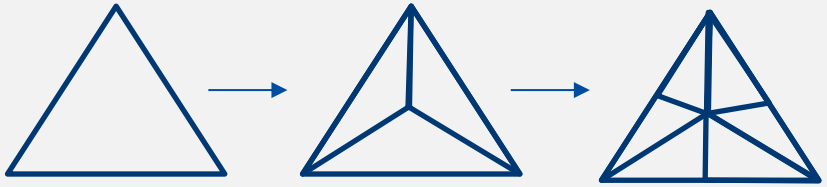


# Geometry Shader

- Geometry shader can create or destroy primitives on GPU
  - Supported in OpenGL 3.x
  - Not yet supported in OpenGL ES

# Tesselleration Shader

- Tesslleration shader performs subdivision rules in the OpenGL pipeline
  - Supported in OpenGL 4.x
  - Not yet supported in OpenGL ES



# Compute Shader

- Execute algorithmically general purpose GLSL shaders
  - Complementary to OpenCL
  - Supported in OpenGL 4.x

- OpenGL
- **OpenGL ES**

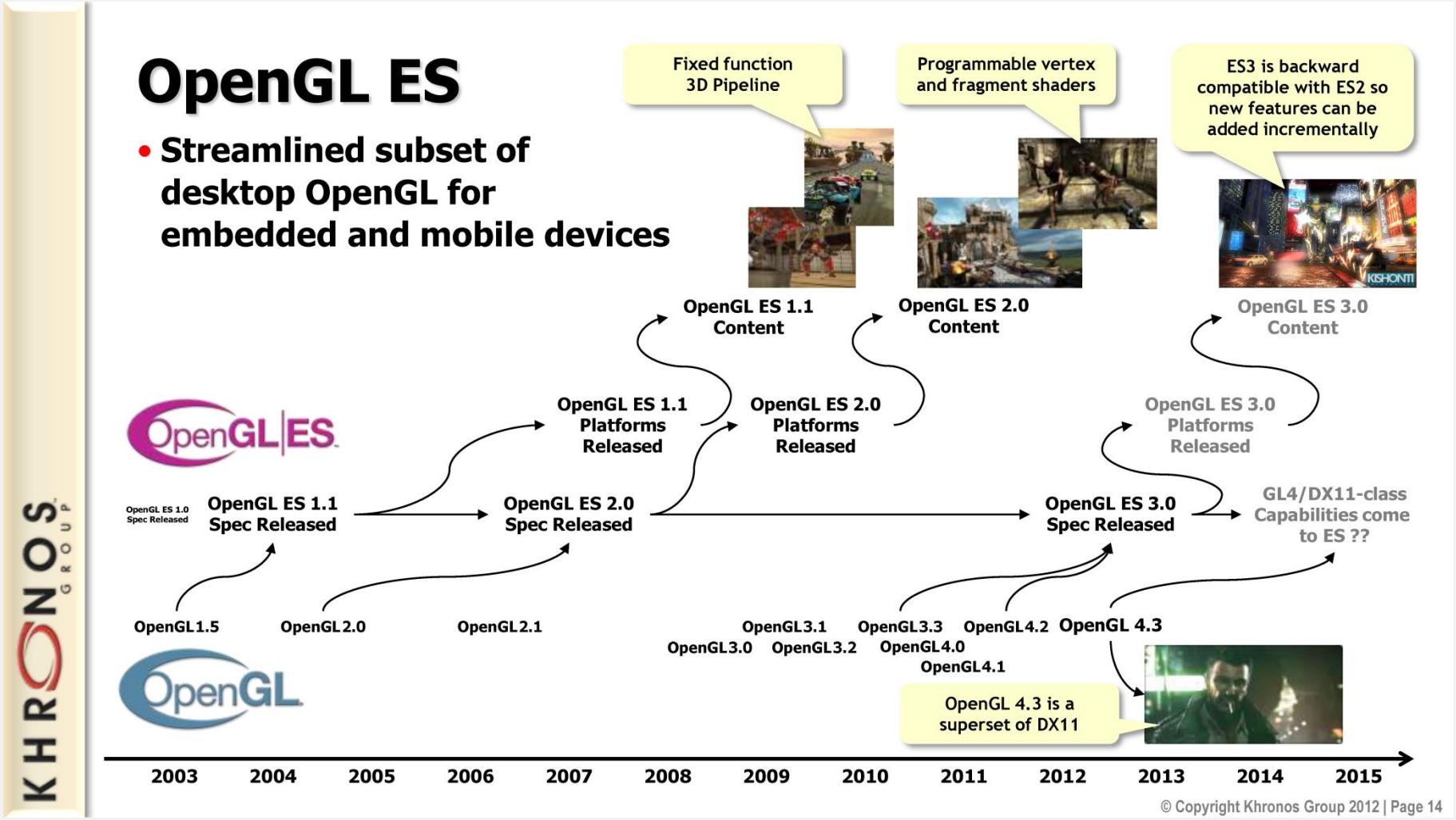
# Introduction

# OpenGL ES

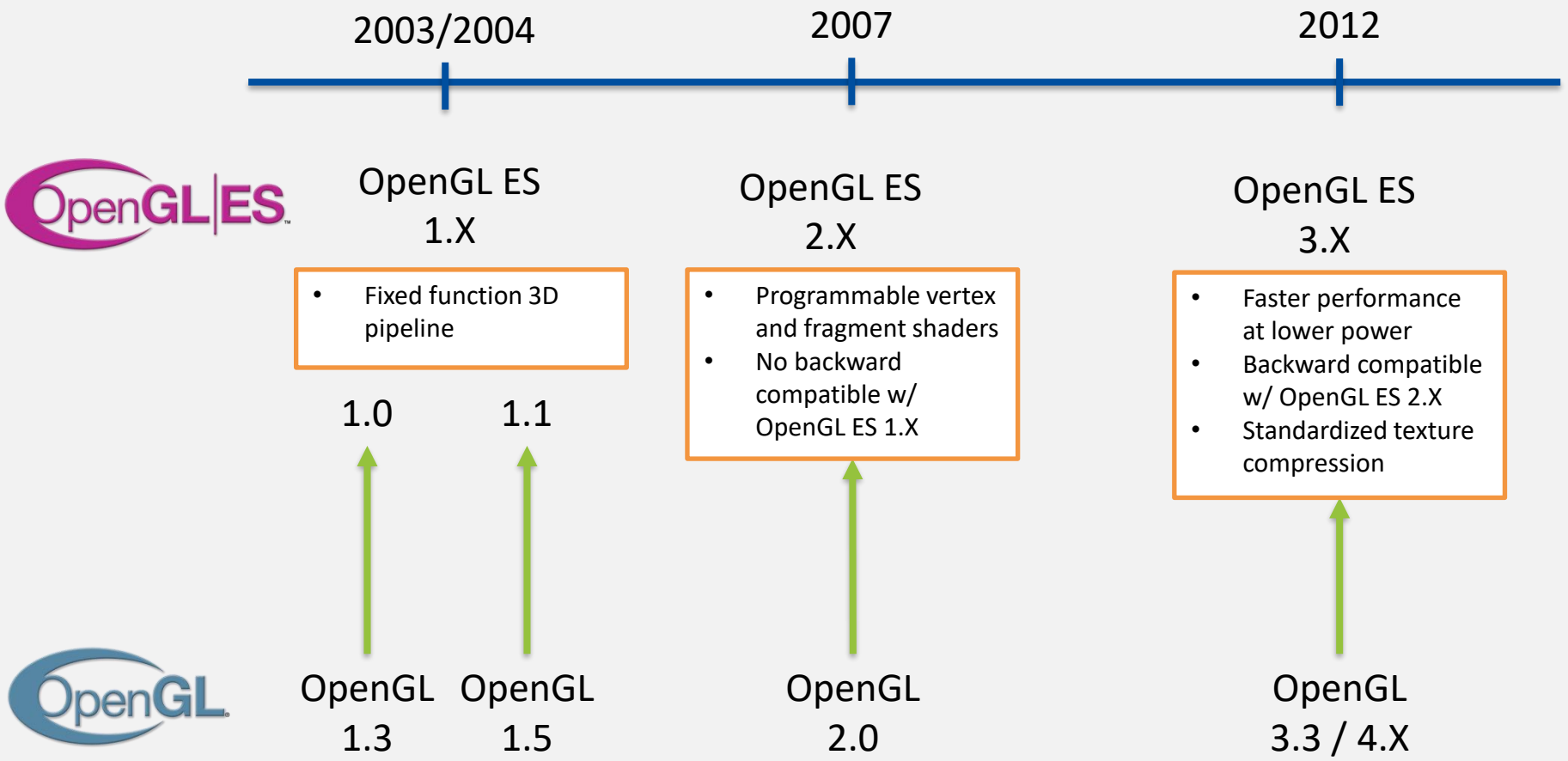
- OpenGL for Embedded Systems
  - A subset of OpenGL for embedded systems
    - Mobile phones, PDAs, video game consoles
  - Royalty-free, cross-platform API
  - Maintained by Khronos Group


- The leading 3D rendering API for mobile and embedded edvices
  - OpenGL ES adopted by every major handset OS
  - OpenGL ES has become the most widely deployed 3D API

# History of OpenGL ES



# History of OpenGL ES – Summary



# History of OpenGL ES – Comparison

| Feature | OpenGL ES 1.1 (2002) | OpenGL ES 2.0 (2007) | OpenGL ES 3.0 (2012) |
|---|---|---|---|
| Texture compression | N/A | N/A | Available |
| 3D textures | N/A | N/A | Available |
| 32bit floating point support | N/A | N/A | Available |
| Programmable shader pipeline | N/A | Available | Available |
| Fixed function pipeline | Available | N/A | N/A |

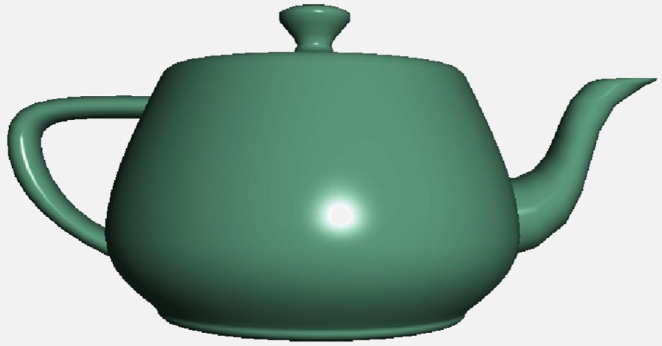
# History of OpenGL ES – Comparison

| Feature | OpenGL ES 1.1 (2002) | OpenGL ES 2.0 (2007) | OpenGL ES 3.0 (2012) |
|---|---|---|---|
| Texture compression | N/A | N/A | Available |
| 3D textures | N/A | N/A | Available |
| 32bit floating point support | N/A | N/A | Available |
| Programmable shader pipeline | N/A | Available | Available |
| Fixed function pipeline | Available | N/A | N/A |

OpenGL ES 1.1

Per-vertex lighting

OpenGL ES 2.0



Per-fragment lighting

# History of OpenGL ES – Comparison

| Feature | OpenGL ES 1.1 (2002) | OpenGL ES 2.0 (2007) | OpenGL ES 3.0 (2012) |
|---|---|---|---|
| Texture compression | N/A | N/A | Available |
| 3D textures | N/A | N/A | Available |
| 32bit floating point support | N/A | N/A | Available |
| Programmable shader pipeline | N/A | Available | Available |
| Fixed function pipeline | Available | N/A | N/A |

- Compressed textures on OpenGL ES 2.0 were a nightmare for developers
  - Compressed textures are supported as OpenGL ES extension APIs
  - Different mobile GPUs → Different compressed texture APIs
    - PowerVR's SGX series: PVRTC/PVRTC2 format
    - Qualcomm Adreno series: KTX/DDS format
    - ARM Mali series: ETC format

# History of OpenGL ES – Comparison

| Feature | OpenGL ES 1.1 (2002) | OpenGL ES 2.0 (2007) | OpenGL ES 3.0 (2012) |
|---|---|---|---|
| Texture compression | N/A | N/A | Available |
| 3D textures | N/A | N/A | Available |
| 32bit floating point support | N/A | N/A | Available |
| Programmable shader pipeline | N/A | Available | Available |
| Fixed function pipeline | Available | N/A | N/A |

- OpenGL ES 3.0 officially supports ETC2/EAC, providing great quality
  - 0.8 db higher quality than S3TC/DXTC
  - 1.0 db higher quality than ETC1
  - 1.8 db higher quality than ATC

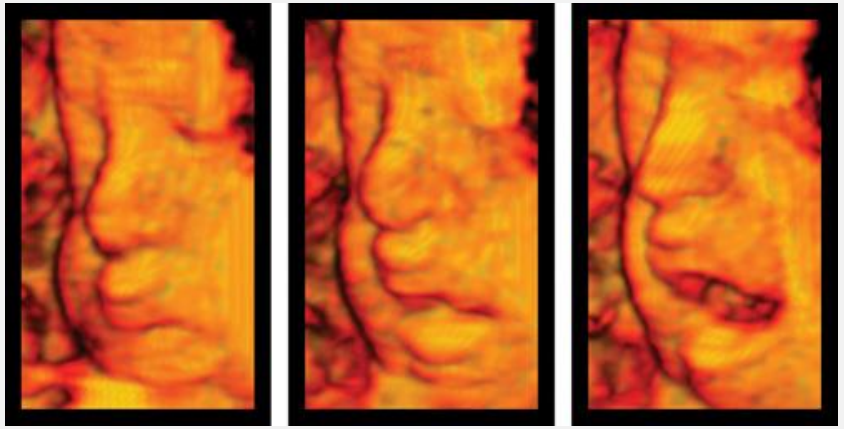
# History of OpenGL ES – Comparison

| Feature | OpenGL ES 1.1 (2002) | OpenGL ES 2.0 (2007) | OpenGL ES 3.0 (2012) |
|---|---|---|---|
| Texture compression | N/A | N/A | Available |
| 3D textures | N/A | N/A | Available |
| 32bit floating point support | N/A | N/A | Available |
| Programmable shader pipeline | N/A | Available | Available |
| Fixed function pipeline | Available | N/A | N/A |

- Now, Khronos group supports the KTX (**K**hronos **T**e**X**ture) format
  - KTX files contain all the parameters needed for texture loading
  - Image data can be stored in any of the compressed formats supported by OpenGL family APIs and extensions
    - ETC1, ETC2, EAC, ATITC, S3TC, BPTC, ASTC, etc.

# History of OpenGL ES – Comparison

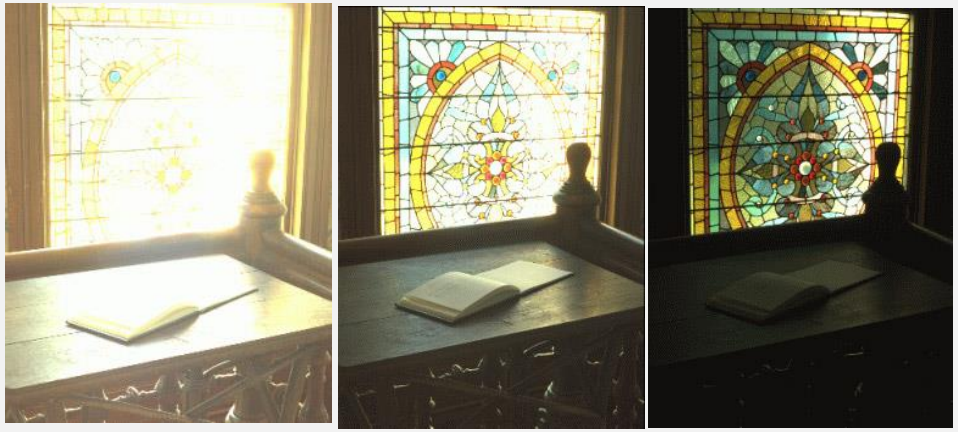
| Feature | OpenGL ES 1.1 (2002) | OpenGL ES 2.0 (2007) | OpenGL ES 3.0 (2012) |
|---|---|---|---|
| Texture compression | N/A | N/A | Available |
| 3D textures | N/A | N/A | Available |
| 32bit floating point support | N/A | N/A | Available |
| Programmable shader pipeline | N/A | Available | Available |
| Fixed function pipeline | Available | N/A | N/A |



# History of OpenGL ES – Comparison

| Feature | OpenGL ES 1.1 (2002) | OpenGL ES 2.0 (2007) | OpenGL ES 3.0 (2012) |
|---|---|---|---|
| Texture compression | N/A | N/A | Available |
| 3D textures | N/A | N/A | Available |
| 32bit floating point support | N/A | N/A | Available |
| Programmable shader pipeline | N/A | Available | Available |
| Fixed function pipeline | Available | N/A | N/A |

- A new version of GLSL ES supports 32bit floating point operations
  - Floating point (FP) texture can be manipulated
    - [0, 1] → [-inf, inf]
    - HDR rendering can be easily supported with shader

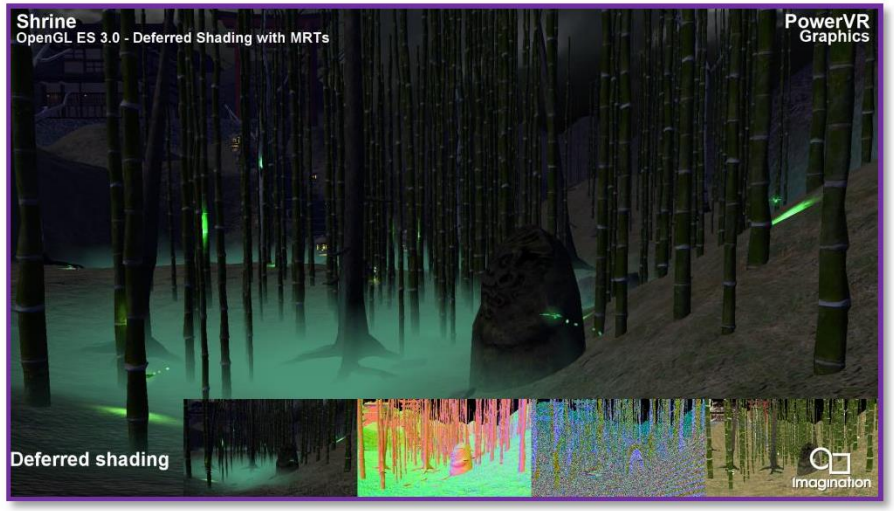


# OpenGL ES 3.0

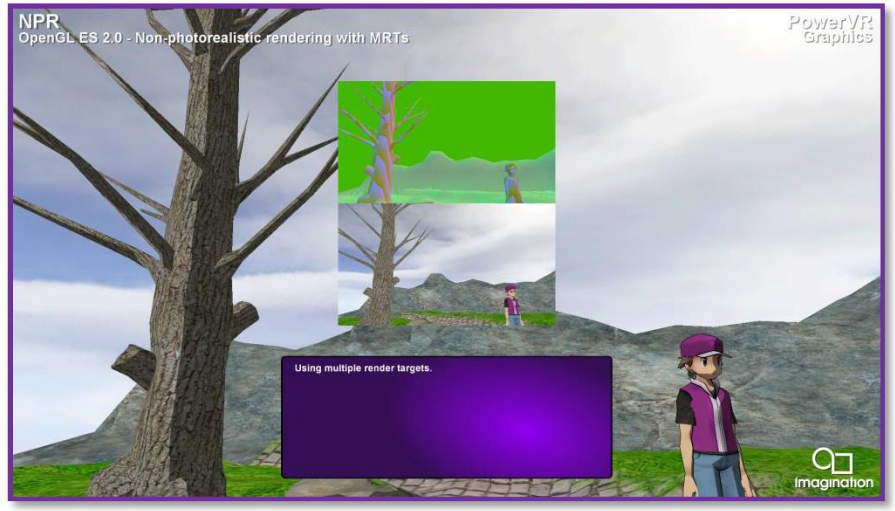
## OpenGL ES 3.0 – Power Efficiency Focus

*Multiple Render Targets*

- Single Geometry Submission fills “Multiple Render Targets”
  - Reduces bandwidth and geometry processing = lower power & higher efficiency




Deferred Shading using MRTs




Cartoon Rendering using MRTs

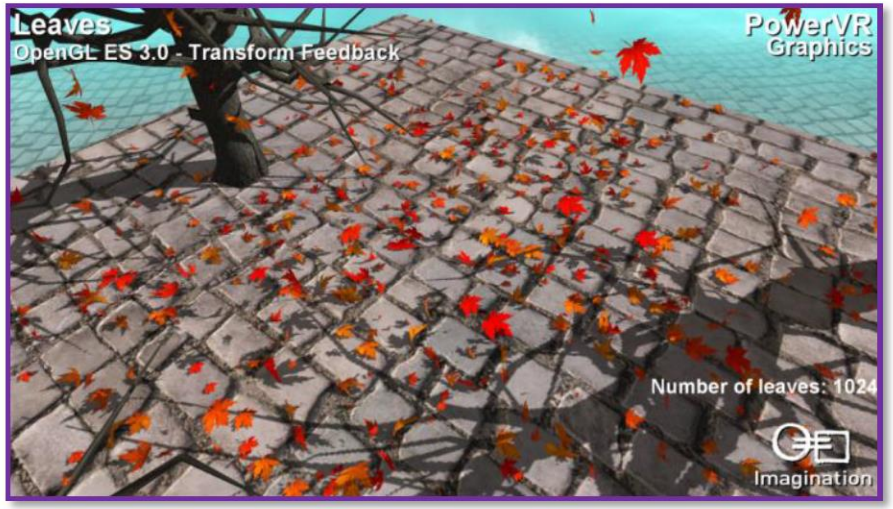
# OpenGL ES 3.0

## OpenGL ES 3.0 – Power Efficiency Focus

*Instancing*

- Instancing enables efficient drawing of many copies of the same object
  - Reduced CPU workload (API calls) resulting in lower power and higher efficiency


Instancing of Buildings for City Rendering




Instancing of Leaves

# OpenGL ES 3.0

## OpenGL ES 3.0 – Power Efficiency Focus

*Transform Feedback*

- Transform Feedback allows write out of vertex shader results to memory
  - Re-using “cached” results equates to higher efficiency by avoiding duplicate calculations


Transform feedback used to cache morphing and skinning animation

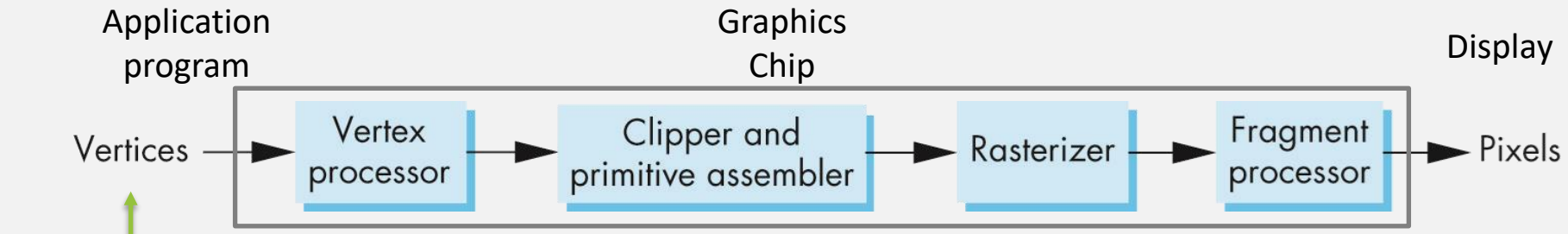

Transform feedback used to implement physics based animation

# Understanding Fixed Rendering Pipeline

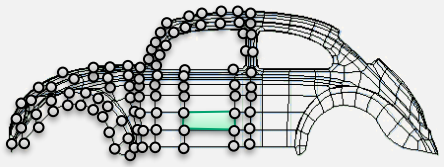
## – from the perspective of Programmable Rendering Pipeline

# 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

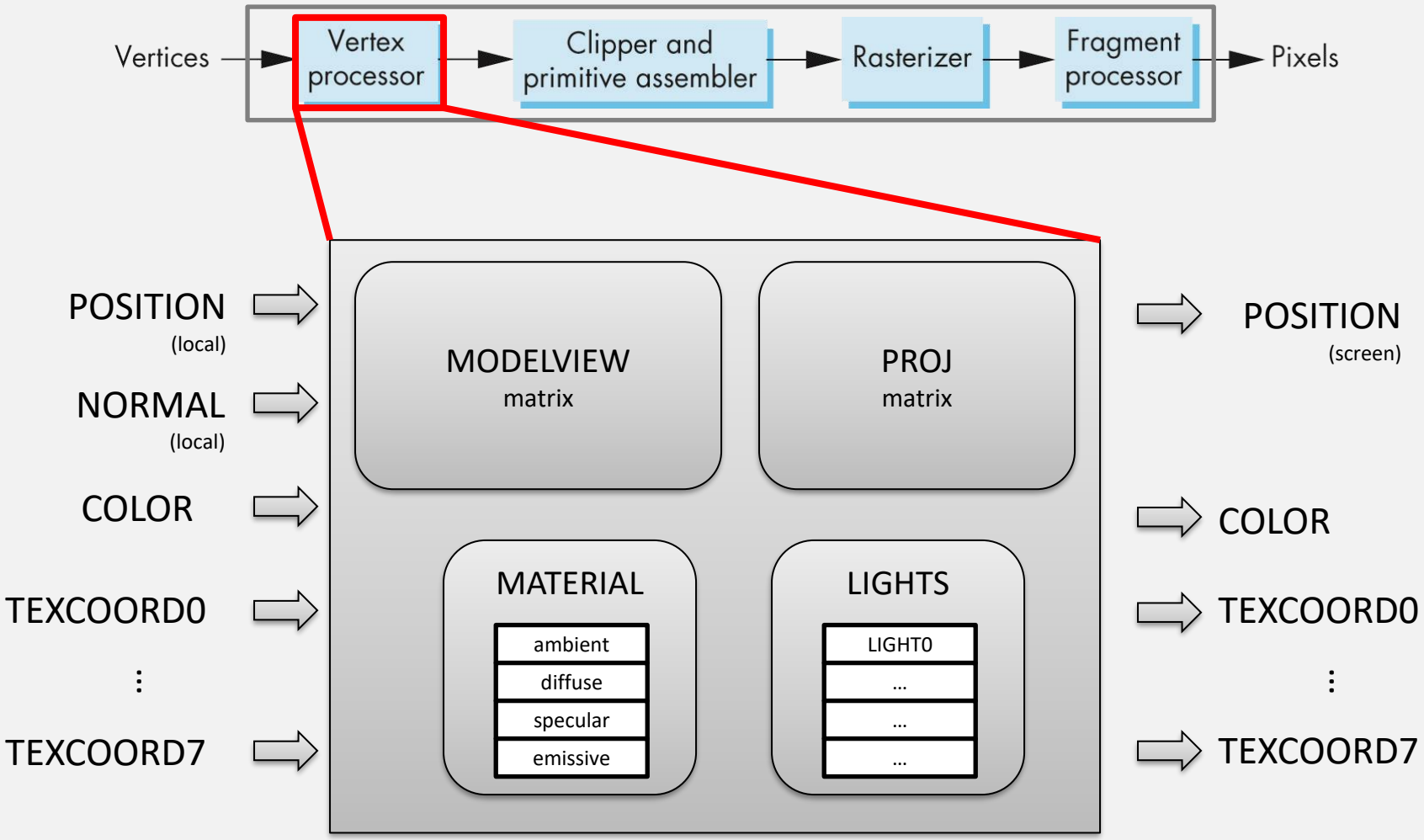



Positions: (x, y ,z), …
Normals: (nx, ny, nz), …
Texcoords: (s, t), …
Primitive type: Quads

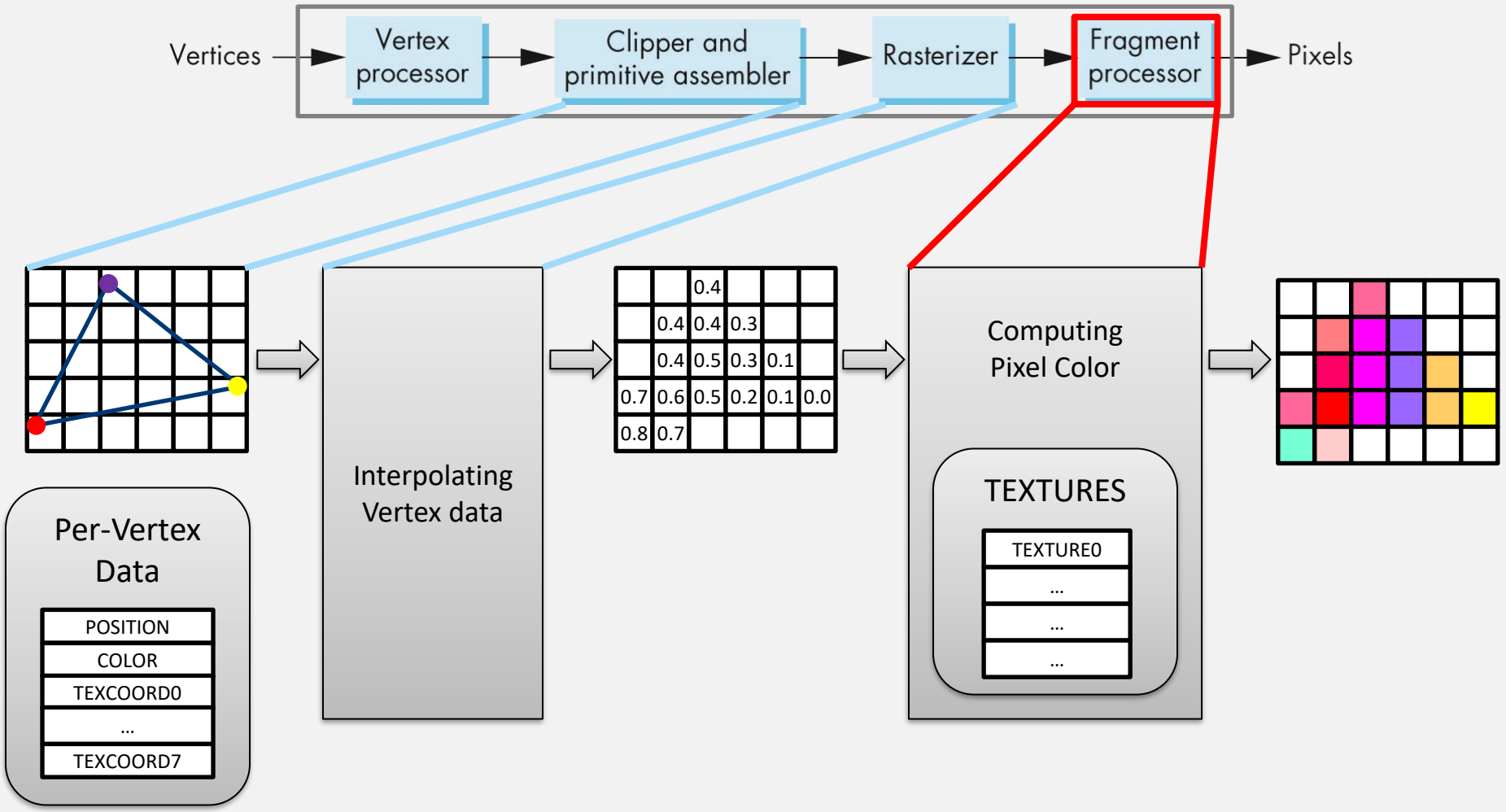


Our input:
In general, artists make it for programmers
(* Maya, 3D studio max, Blender, etc.)

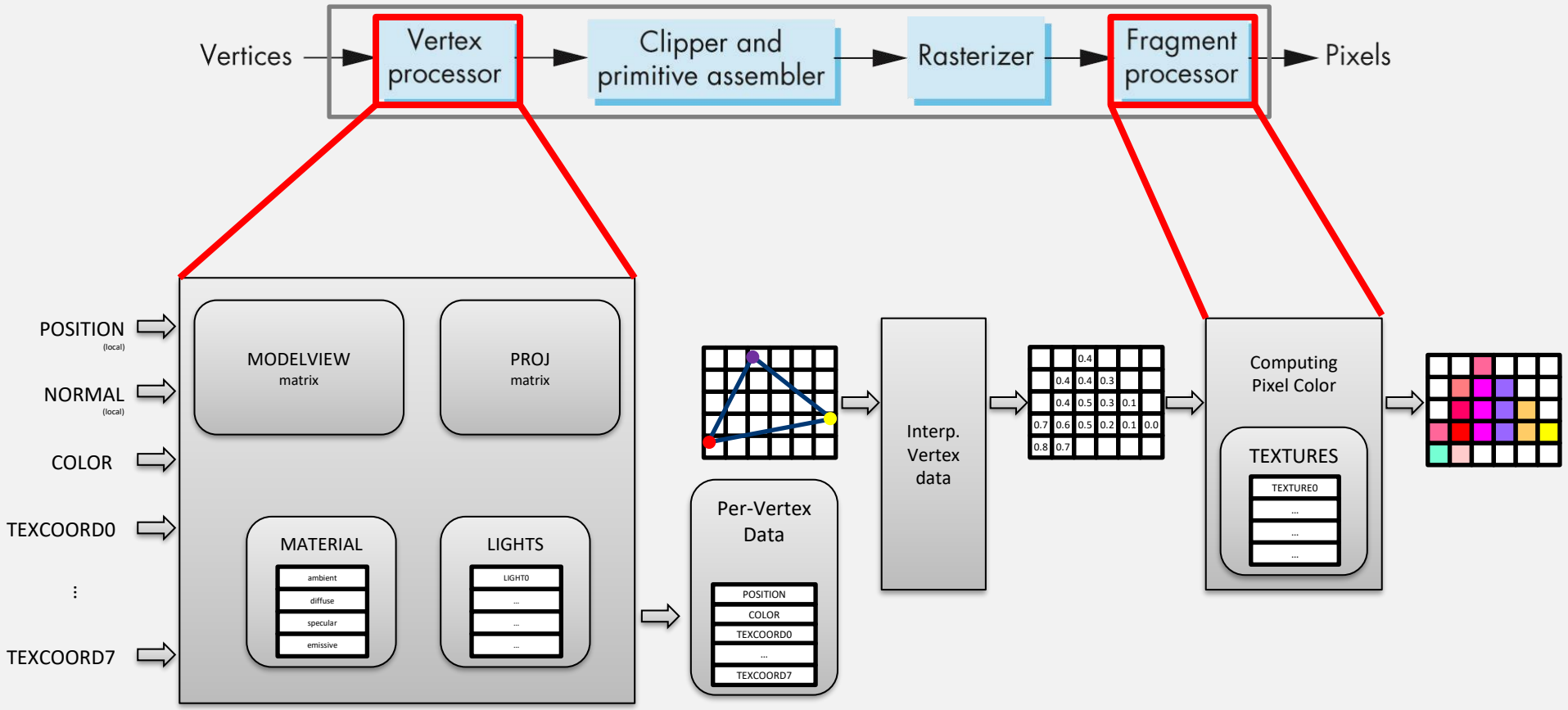
# Vertex Processing w/ Fixed Rendering Pipeline



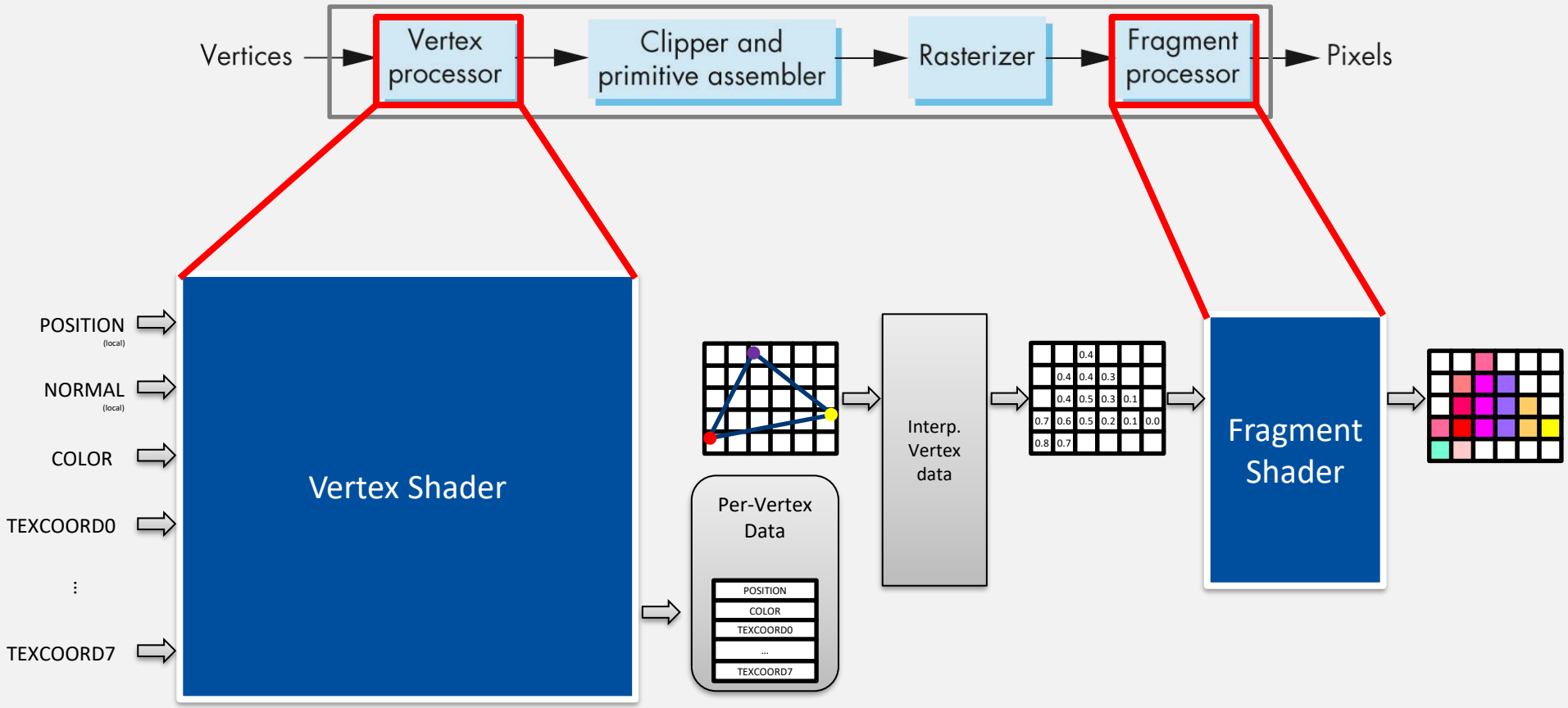
# Fragment Processing w/ Fixed Rendering Pipeline



# 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

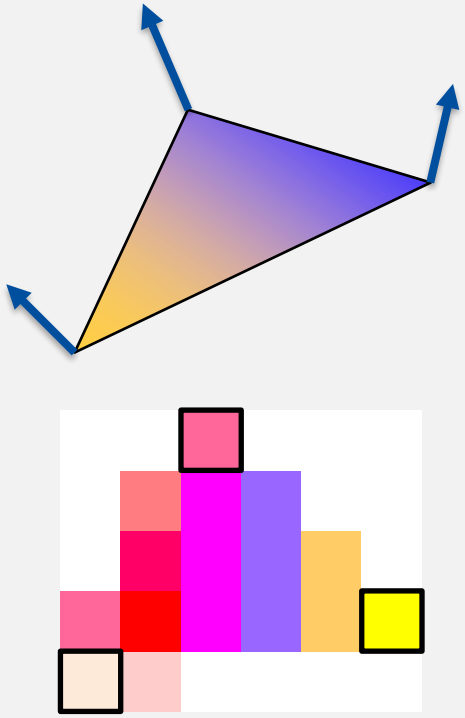
Per-vertex lighting

per-fragment lighting

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

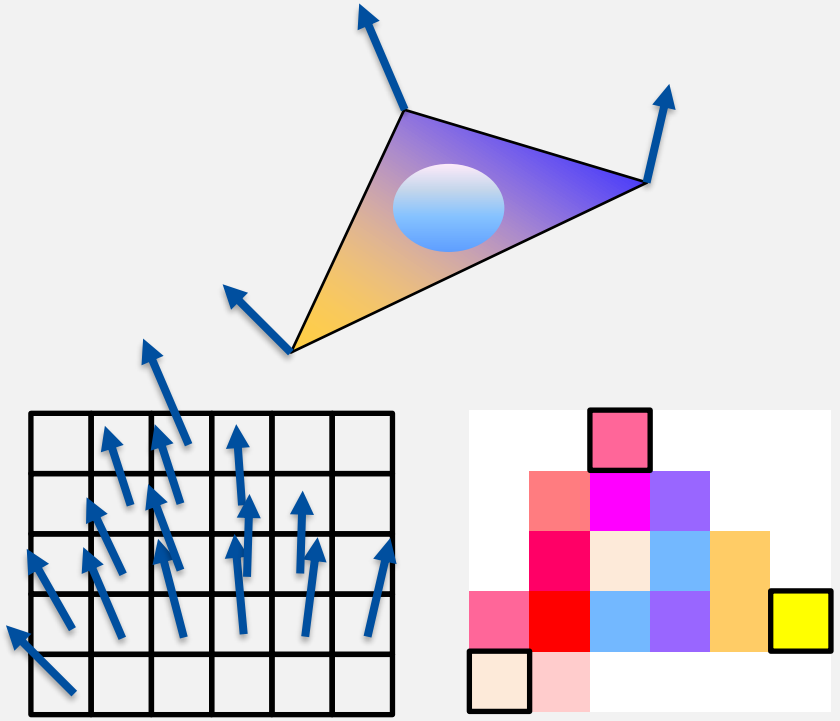
## Per-vertex lighting

- Computation of lighting is performed in vertex processor only

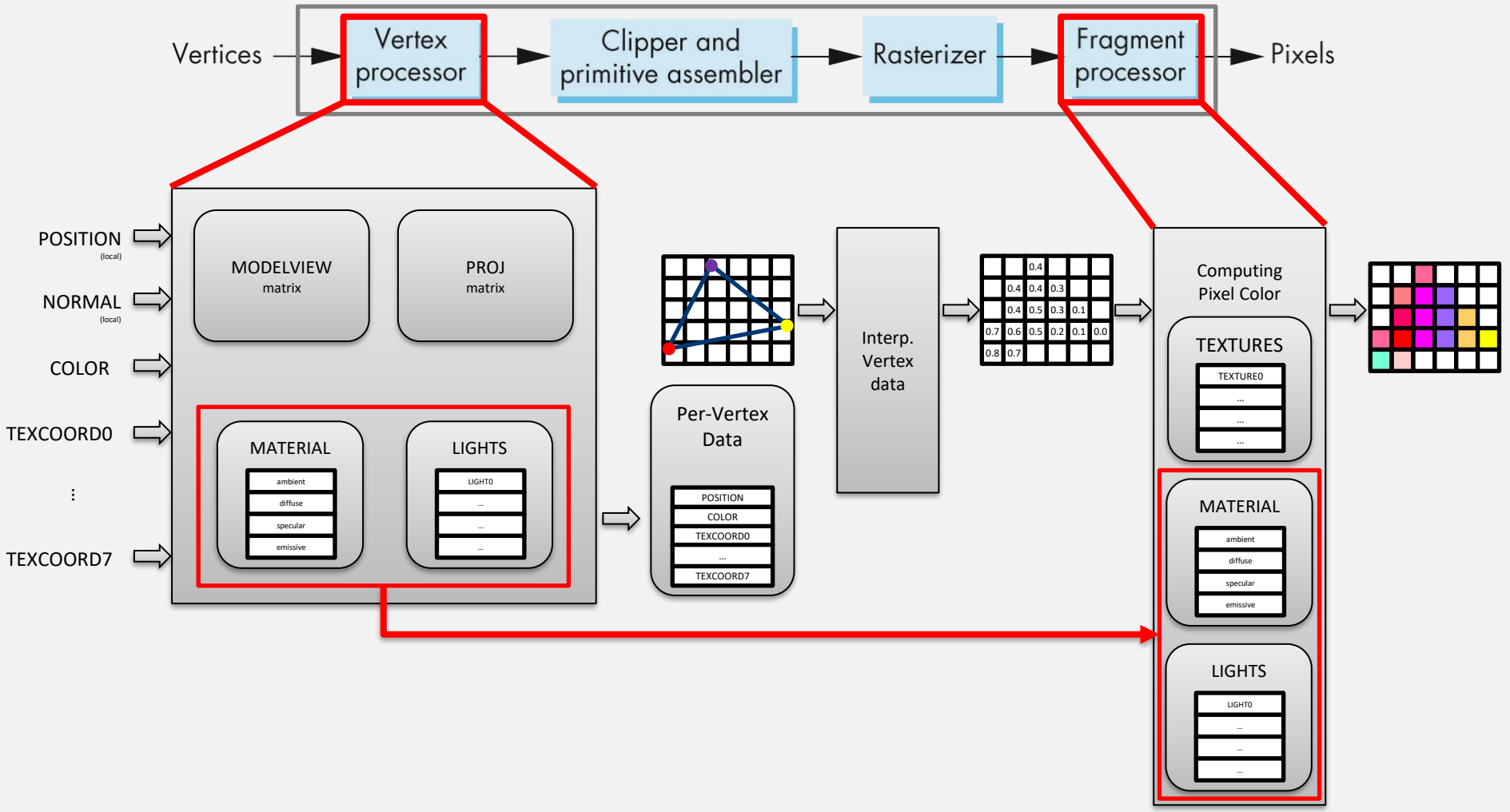


## Per-fragment lighting

- Computation of lighting CAN be performed in fragment processor



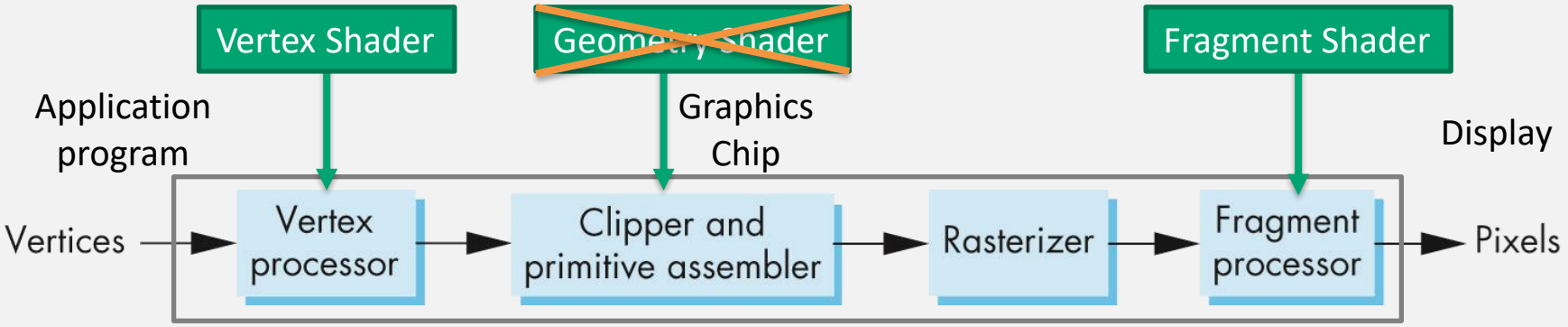
# Per-pixel Lighting w/ Programmable Rendering Pipeline



# Shader Programming at a Glance

# Programmable Rendering Pipeline in Modern OpenGL

- Function units in rendering pipeline can be programmed with *shader* language
  - We can programming the functionality of rendering pipeline units
- OpenGL 2.x only supports two types of shaders
  - Vertex shader & framgment shader



# 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 |

# GLSL Programming at a glance

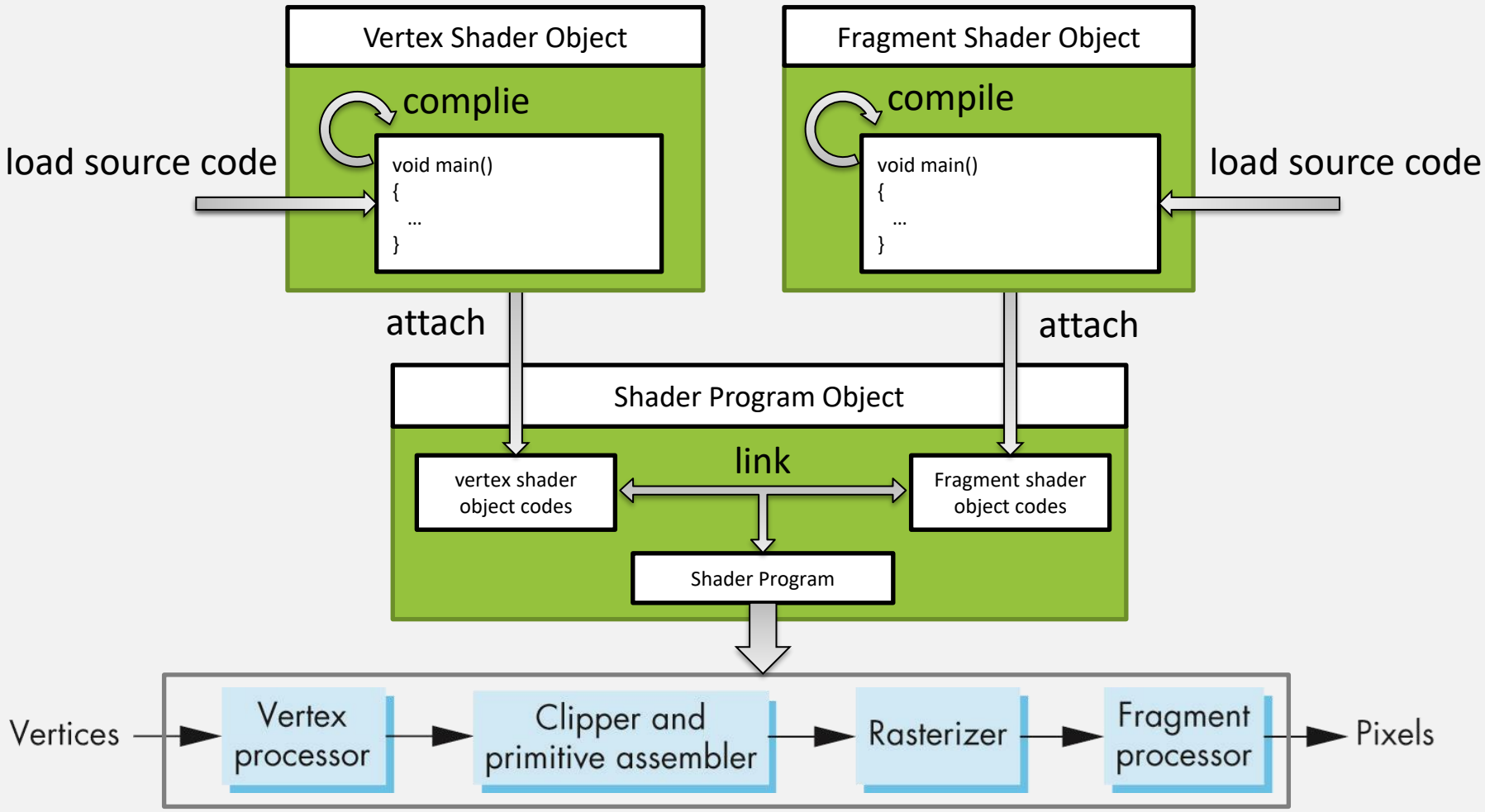
## Vertex Shader

```
////////////////////////////////
/// Vertex Shader             //
////////////////////////////////

#version 120                  // GLSL 1.20

// uniforms used by the vertex shader
uniform mat4 u_PVM;

// attributes input to the vertex shader
attribute vec3 a_position;
attribute vec3 a_color;    // input vertex color

// varying variables - input to the fragment shader
varying vec3 v_color;       // output vertex color

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

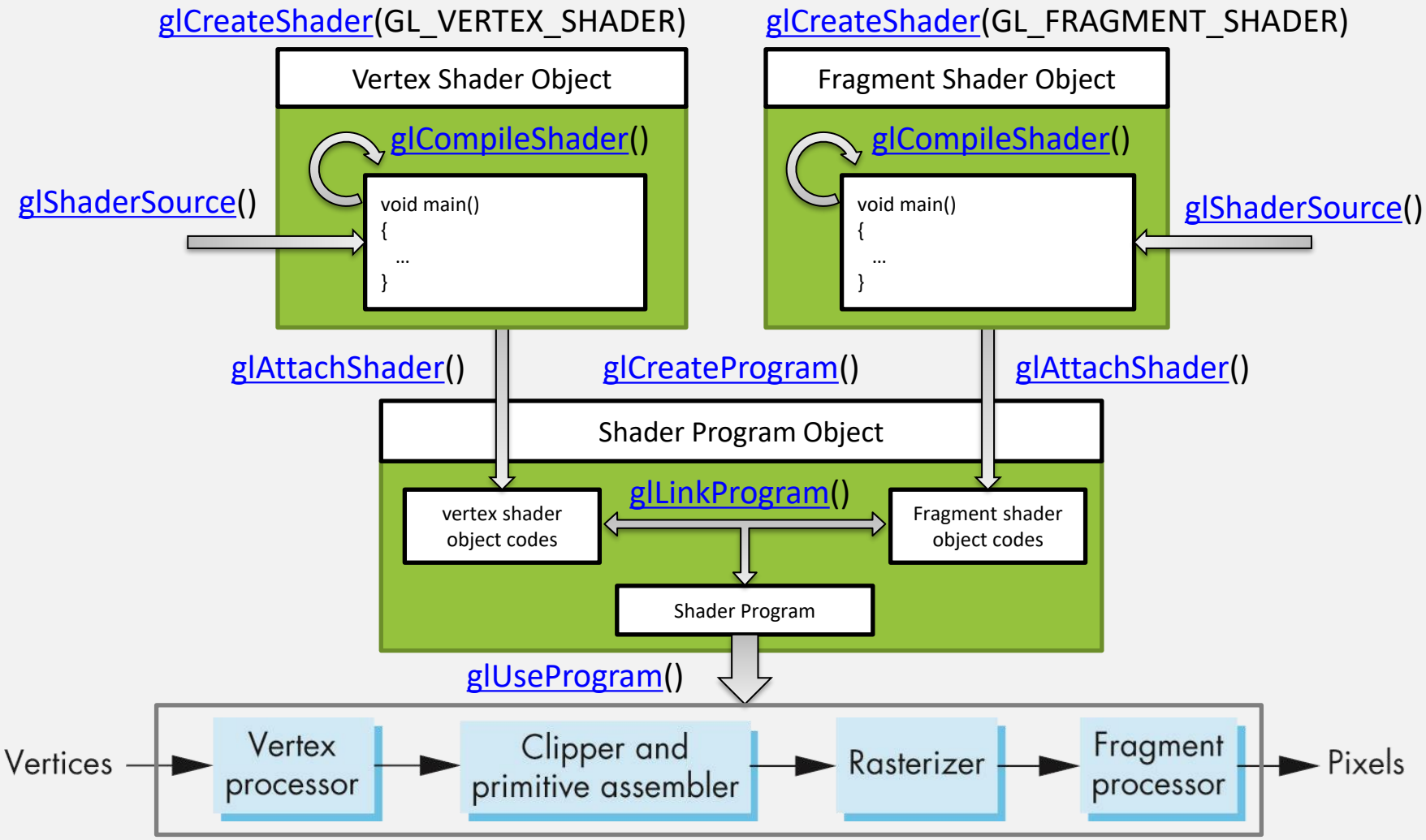
## Fragment Shader

```
////////////////////////////////
/// Fragment Shader            //
////////////////////////////////

#version 120                  // GLSL 1.20

varying vec3 v_color;

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

# GLSL Programming at a glance



# GLSL Programming at a glance



# GLSL Programming – OpenGL 2.x+

## Initialization of Vertex Shader, Fragment Shader, and Program Object

```
int init()
{
  GLbyte vShaderStr[] = { ... };
  GLbyte fShaderStr[] = { ... };

  // shader and program objects
  GLuint vertexShader, fragmentShader, programObject;

  // create and load vertex shader
  vertexShader = glCreateShader(GL_VERTEX_SHADER);
  glShaderSource(vertexShader, 1, &vShaderStr, NULL);
  glCompileShader(vertexShader);

  // create and load fragment shader
  fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
  glShaderSource(fragmentShader, 1, &vShaderStr, NULL);
  glCompileShader(fragmentShader);

  // create program object and attach shaders
  programObject = glCreateProgram();
  glAttachShader(programObject, vertexShader);
  glAttachShader(programObject, fragmentShader);

  // bind attributes 0 ~ 7
  glBindAttribLocation(programObject, 0, …);

  // link the program
  glLinkProgram(programObject);
}
```

## Rendering with Programmable Rendering Pipeline

```
int draw()
{
  // set viewport
  glViewport(…);

  // clear color/depth buffers
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  // use the program
  glUseProgram(programObject);

  // load the vertex data
  glVertexAttribPointer(…);
  glEnableVertexAttribArray(0);

  // draw with the vertex data
  glDrawArrays(…);

  // flush the framebuffer with double buffering
  eglSwapBuffers(…);
}
```

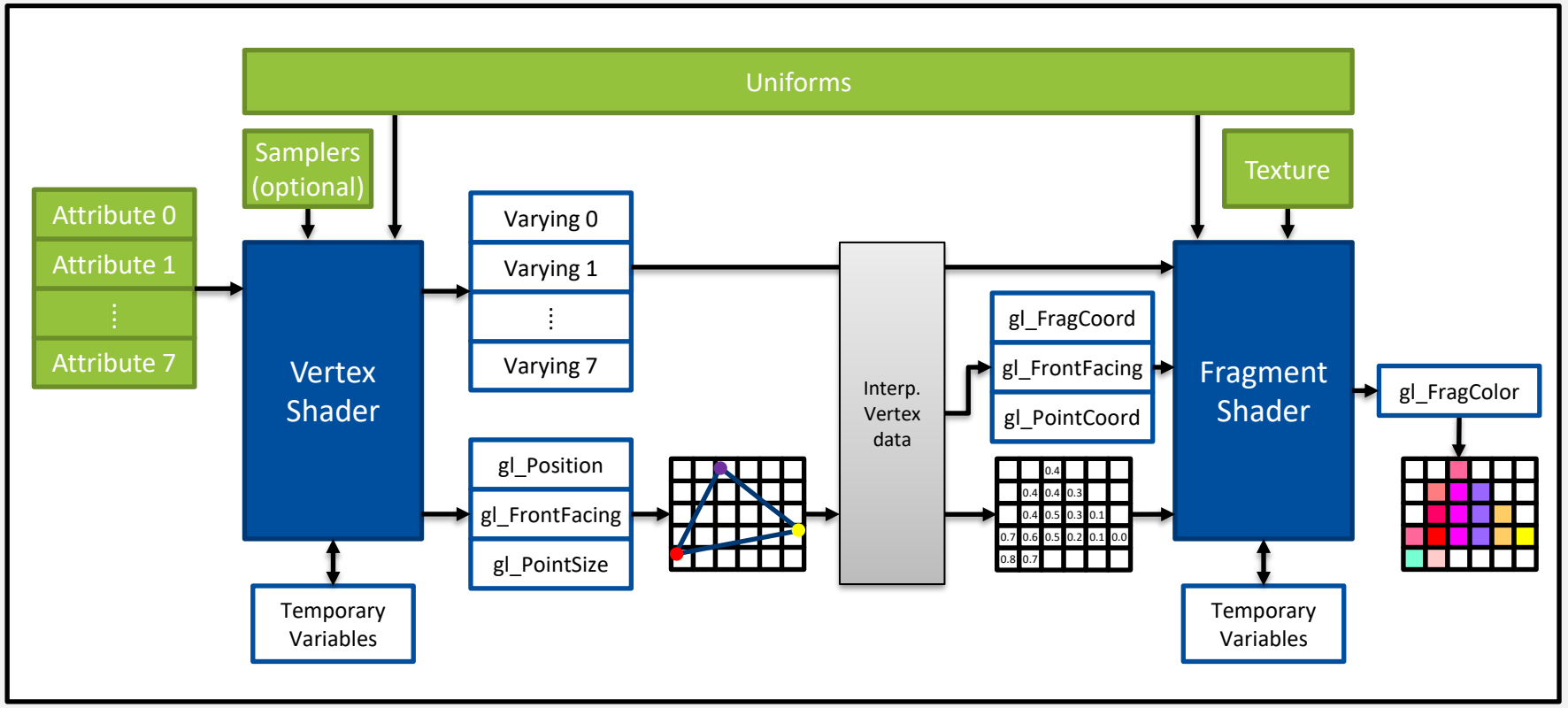
# Summary of GLSL Programming at a Glance

- There are two shaders, and we can program the shader with GLSL language
  - Vertex shader
  - Fragment shader
- Similar to developing typical programs, we have to
  - Complie vertex/fragment shader objects
  - Link them into a program object
- Similar to using typiscal 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 shoud use glVertexAttribPointer() in OpenGL 2.x+
    - Every data associated with vertices is considered as one of vertex attributes in OpenGL 2.x+

# More about Shader Programming

# Programmable Rendering Pipeline – OpenGL 2.x+

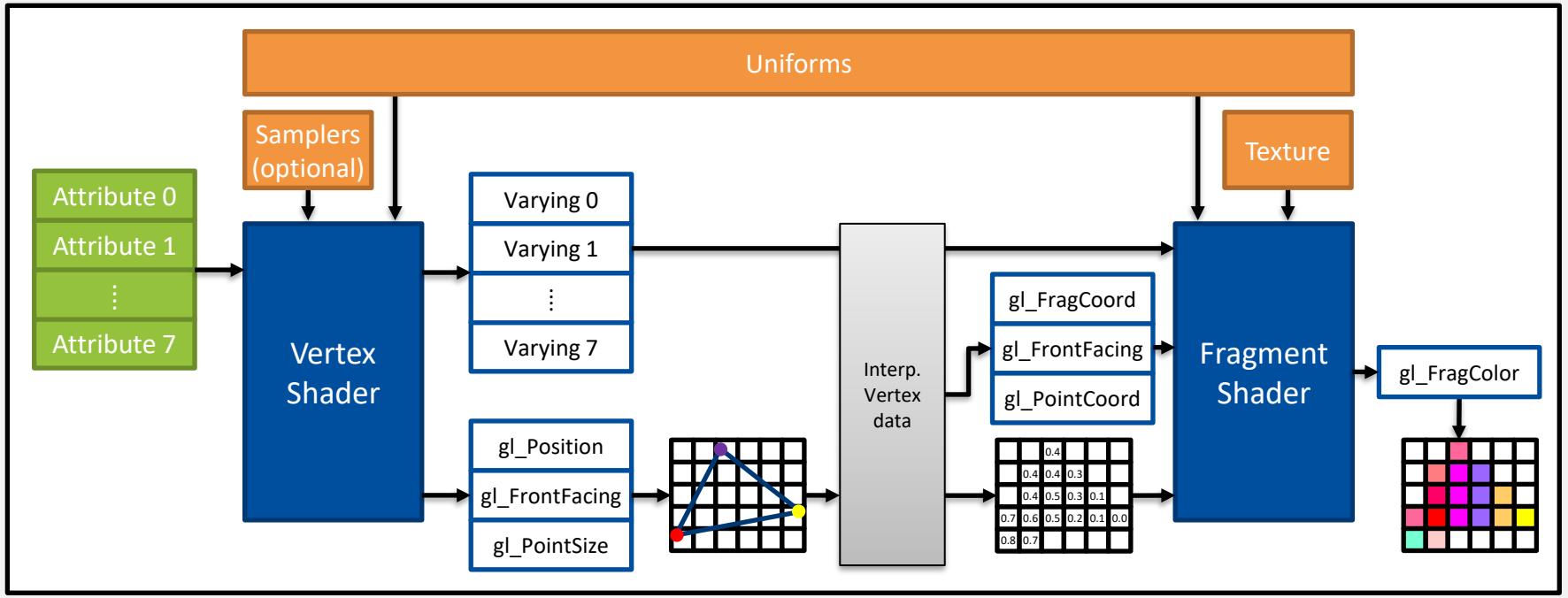


# I/O Storage Qualifiers in OpenGL 2.x+ Shader

- 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

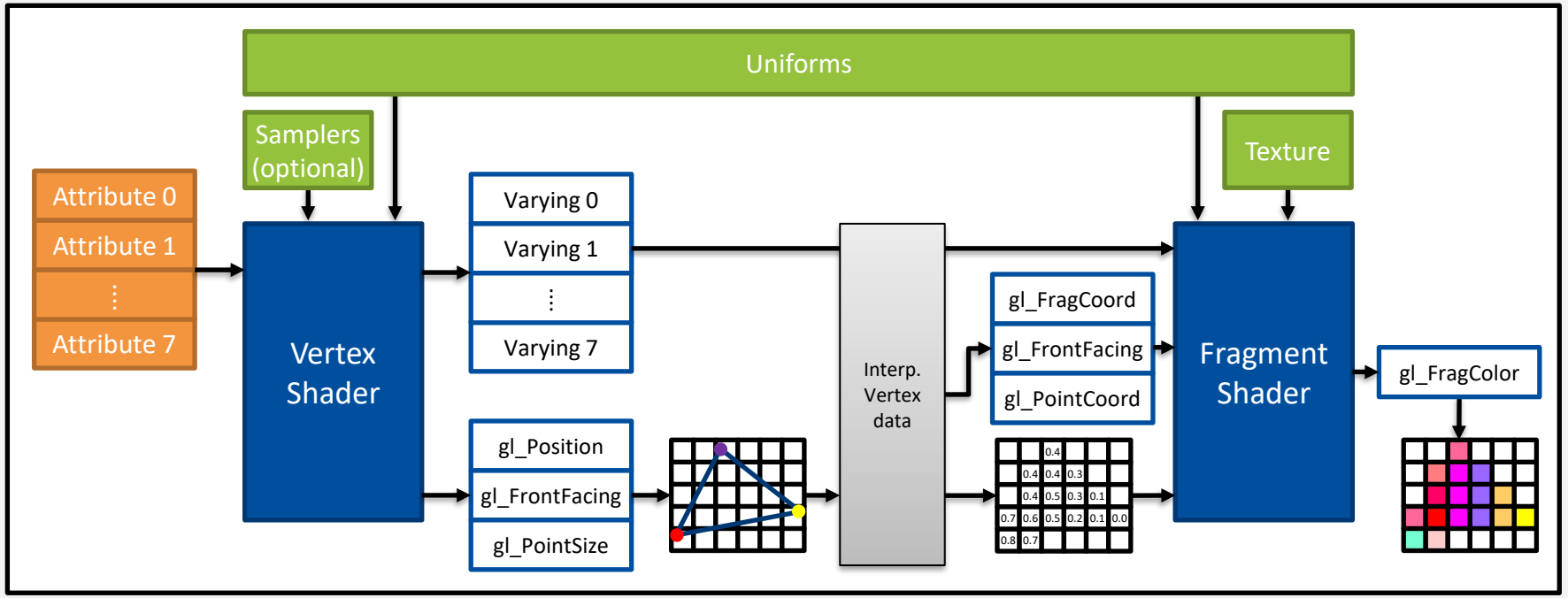
# Uniform – I/O Storage Qualifiers

- 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



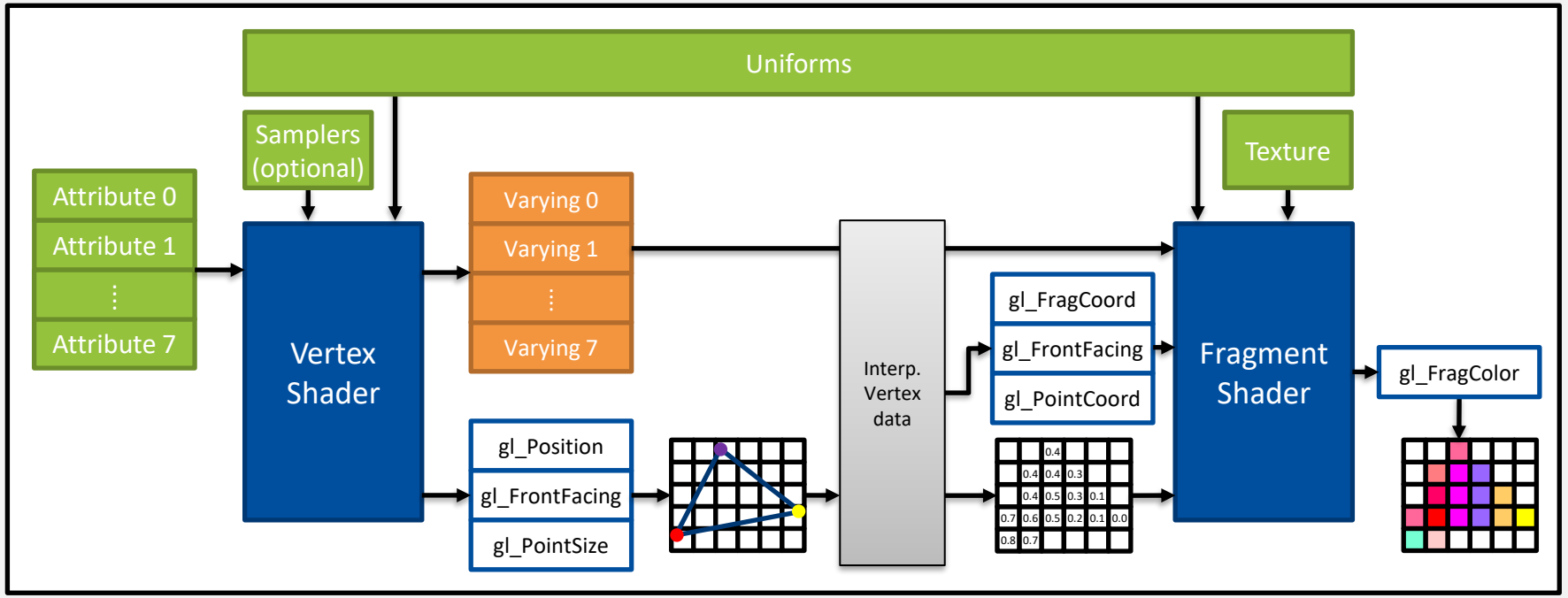
# Attribute – I/O Storage Qualifiers

- 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

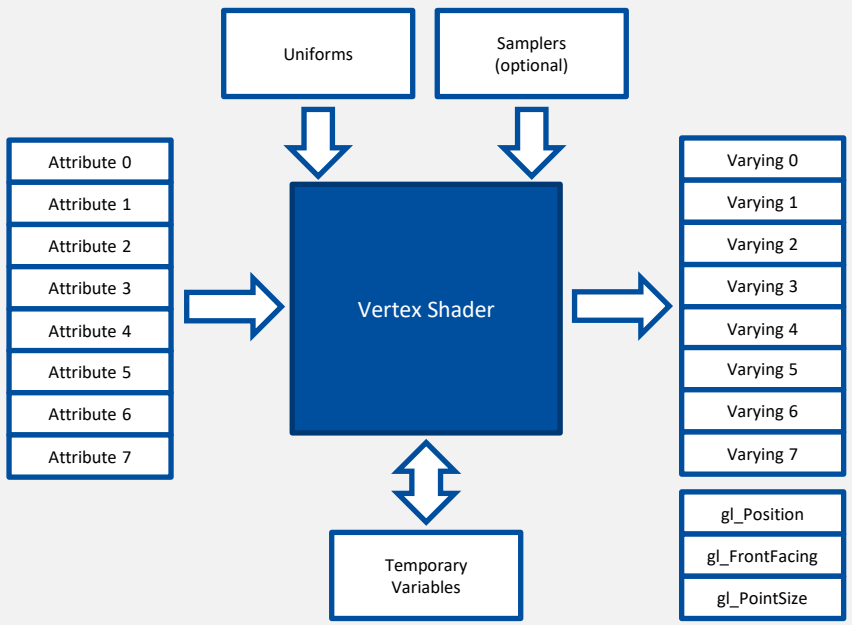


# Data Types in OpenGL Shader

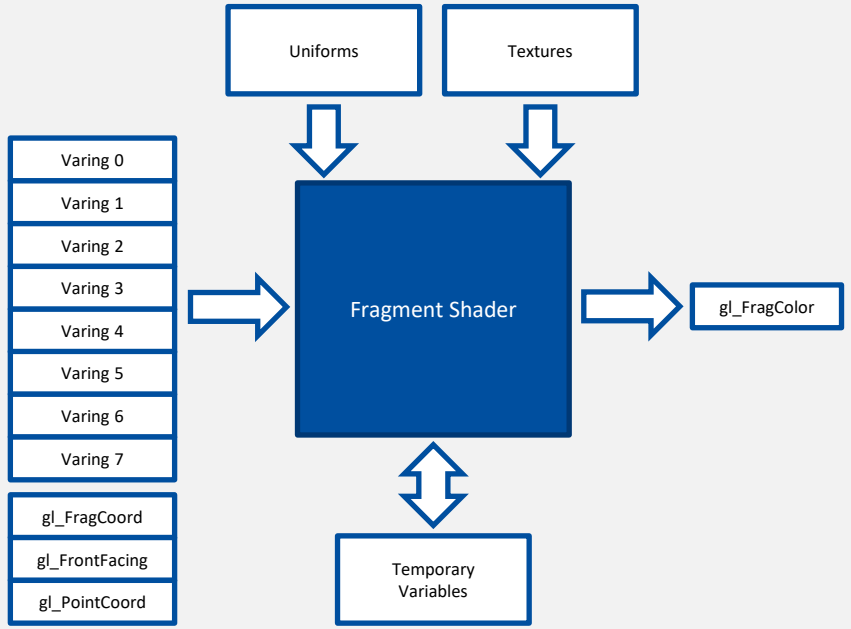
| Variable Class | Types | Description |
|---|---|---|
| Scalars | **float**, **int**, **bool** | Scalar-based data types for floating-point, integer, and boolean values |
| Floating-point vectors | **float**, **vec2**, **vec3**, **vec4** | Floating-point-based vector types of one, two, three, or four 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 glFrontFacing
  - 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

// uniforms used by the vertex shader
uniform mat4 u_PVM;

// attributes input to the vertex shader
attribute vec3 a_position;
attribute vec3 a_color;    // input vertex color

// varying variables - input to the fragment shader
varying vec3 v_color;       // output vertex color

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;

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

## OpenGL codes (C/C++)

```
GLuint  vertexShader, fragmentShader, programObject;
GLuint  loc_a_position, loc_a_color, loc_u_PVM;
GLuint  position_buffer, color_buffer;

void init()
{
 // Compile sahders and Link them into the program
  programObject = glCreateProgram();
  vertexShader   = glCreateShader(GL_VERTEX_SHADER);
  fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);

  // get uniform / vertex attribute locations
  loc_u_PVM      = glGetUniformLocation(programObject, "u_PVM");
  loc_a_position = glGetAttribLocation(programObject, "a_position");
  loc_a_color    = glGetAttribLocation(programObject, "a_color");

  glLinkProgram(programObject);
}

void draw()
{
  std::vector<glm::vec3>  tv_positions;    // per triangle-vertex 3D position
  std::vector<glm::vec3>  tv_colors;       // per triangle-vertex 3D color
  glm::mat4   mat_PMV;  // Proj * View * Model

  // You set tv_positions, tv_colors, mat_PVM, and then...
  glUniformMatrix4fv(loc_u_PVM, 1, GL_FLASE, glm::value_ptr(mat_PVM));

  glEnableVertexAttribArray(loc_a_position);
  glVertexAttribPointer(loc_a_position, 3, GL_FLOAT, GL_FALSE, 0, &tv_positions[0]);

  glEnableVertexAttribArray(loc_a_color);
  glVertexAttribPointer(loc_a_color, 3, GL_FLOAT, GL_FALSE, 0, &tv_colors[0]);

  glDrawArrays(...);

  glDisableVertexAttribArray(loc_a_position);
  glDisableVertexAttribArray(loc_a_color);
}
```

# 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);
```

# Parameter/Precision Qualifiers in OpenGL Shader

- Parameter Qualifiers

| Qualifier | Meaning |
|---|---|
| <none: default> | same as **in** |
| **in** | for function parameters passed into a function |
| **out** | for function parameters passed back out of a function, but not initialized for used when passed in |
| **inout** | for function parameters passed both into and out of a function |

- Precision Qualifiers

| Qualifier | Meaning |
|---|---|
| **highp** | Satisfies the minimum requirements for the vertex language described above. Optional in the fragment language |
| **mediump** | Satisfies the minimum requirements above for the fragment language. Its range and precision has to be greater than or the same as provided by **lowp** and less than or the same as provided by **highp** |
| **lowp** | Range and precision that can be less than midump, but still intended to represent all color values for any color channel |

# Setting Object & Viewer with Programmable Rendering Pipeline

# Change of Coordinates

Rendering pipeline
- Or, it can be emulated w/ gluProject())

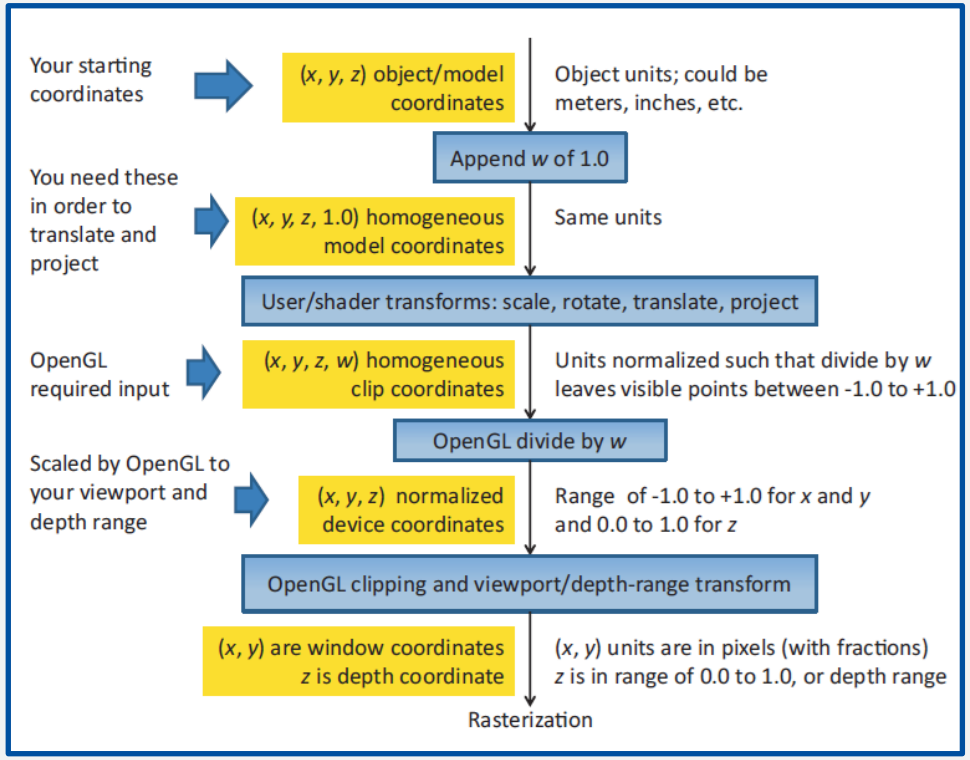
Emulation w/ gluUnProject()
- The most important part for UI designs

$\mathbf{x}_{obj}$

$\mathbf{V}^{-1}\mathbf{TRSx}_{obj} = \mathbf{x}_{camera}$

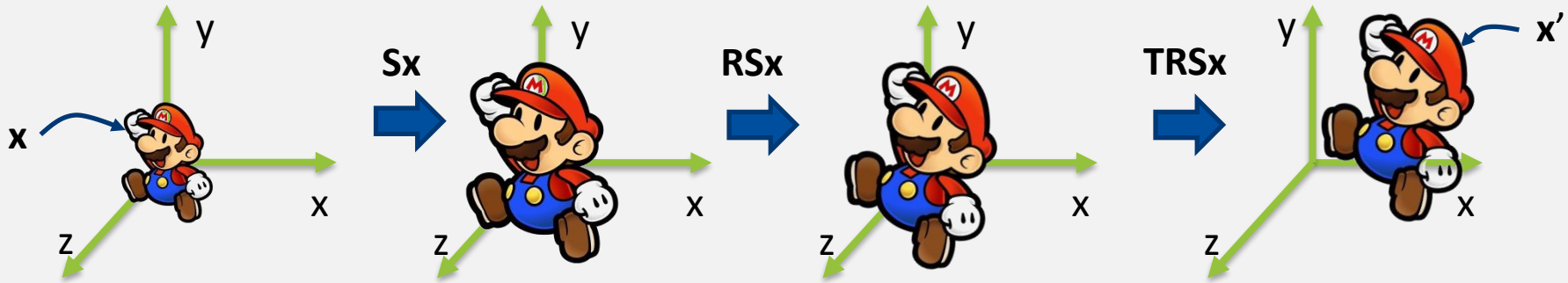
$\mathbf{PV}^{-1}\mathbf{TRSx}_{obj} = \mathbf{x}_{clip}$

$\mathbf{WPV}^{-1}\mathbf{Mx}_{obj} = \mathbf{x}_{screen}$



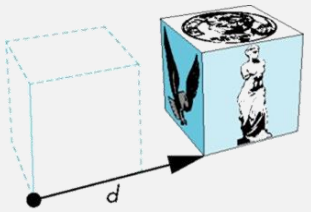
# Model Matrix

- We can composite transformation by multiplying matrices of rotation, translation, and scaling.
  - Example:
    1) Uniformly scale 2x: **S = S(2,2,2)**
    2) Rotate -30 degrees by +z axis: **R = $R_z$(-30)**
    3) Translate by (2, 1, 0): **T = T(2,1,0)**
    - **x'** = **T**(**R**(**Sp**)) = **TRSx**
      - **x**: original object
      - **x'**: transformed object



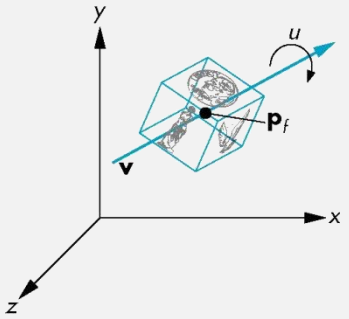
# Transformation Matrices

- Translation



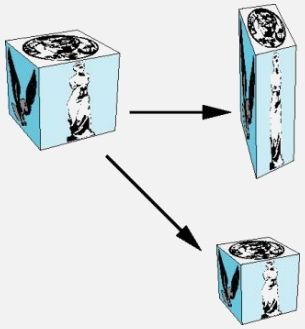

$$\begin{bmatrix} 1 & 0 & 0 & d_x \\ 0 & 1 & 0 & d_y \\ 0 & 0 & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Rotation




$$\begin{bmatrix} \cos\theta + u_x^2(1-\cos\theta) & u_x u_y(1-\cos\theta) - u_z \sin\theta & u_x u_z(1-\cos\theta) + u_y \sin\theta & 0 \\ u_y u_x(1-\cos\theta) + u_z \sin\theta & \cos\theta + u_y^2(1-\cos\theta) & u_y u_z(1-\cos\theta) - u_x \sin\theta & 0 \\ u_z u_x(1-\cos\theta) - u_y \sin\theta & u_z u_y(1-\cos\theta) + u_x \sin\theta & \cos\theta + u_z^2(1-\cos\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

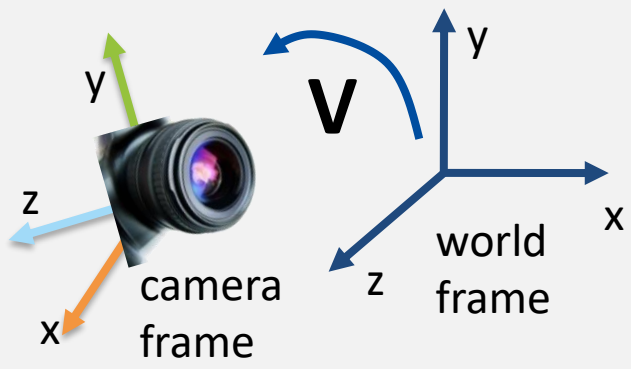
- Scaling



$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# View Matrix

- See the inside of gluLookAt() at this link
  - Rotate the camera frame on the world frame: $\mathbf{R}_c$
  - Translate the camera frame on the world frame: $\mathbf{T}_c$
  - Therefore, $\mathbf{V} = \mathbf{T}_c\mathbf{R}_c$ and gluLookAt() generates $\mathbf{V}^{-1} = \mathbf{R}_c^{-1}\mathbf{T}_c^{-1}$

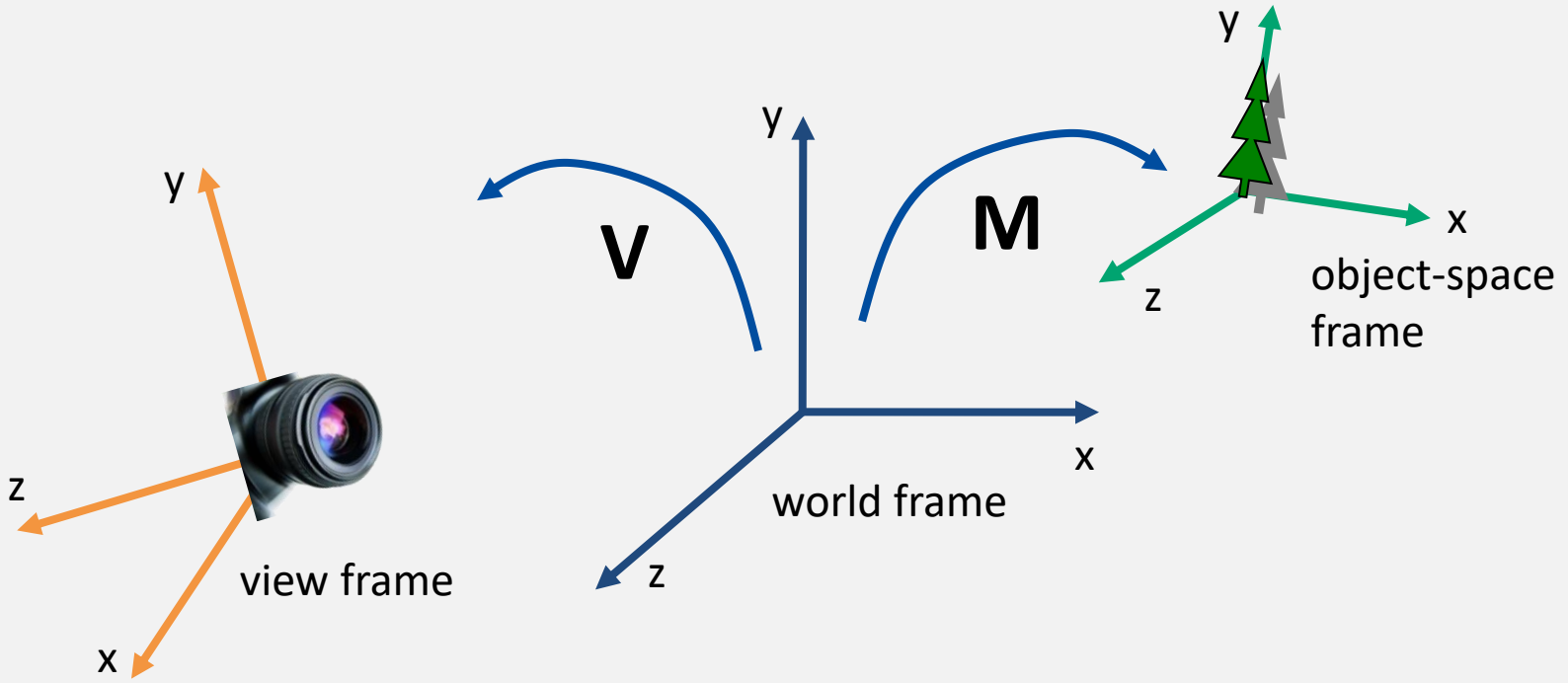



$$\mathbf{V}^{-1} = \mathbf{R}_c^{-1}\mathbf{T}_c^{-1} = \begin{bmatrix} & & & \\ & & & \\ & & & \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & & & -dx \\ & 1 & & -dy \\ & & 1 & -dz \\ & & & 1 \end{bmatrix}$$

# ModelView Matrix

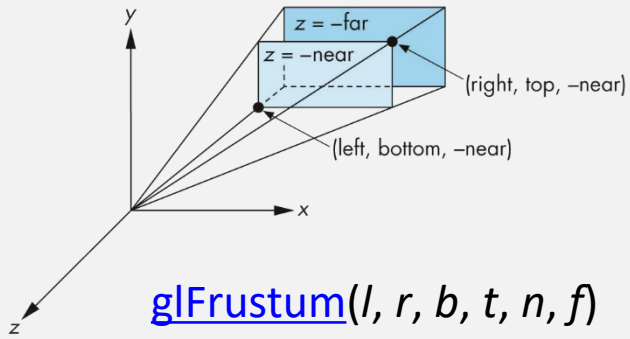
- Exactly same!
  - $\mathbf{x}_{world} = \mathbf{M}\mathbf{x}_{obj}$
  - $\mathbf{x}_{world} = \mathbf{V}\mathbf{x}_{view}$

$$\mathbf{x}_{view} = \mathbf{V}^{-1}\mathbf{M}\mathbf{x}_{obj}$$



# Projection Matrix

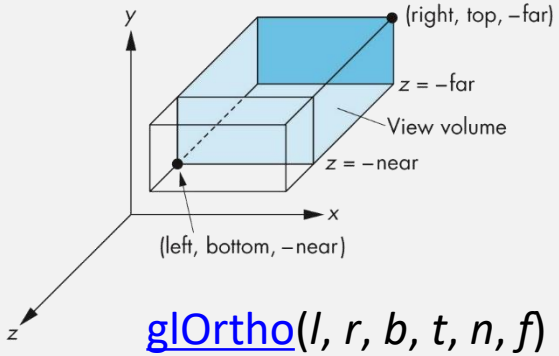
- Perspective projection




glFrustum(*l*, *r*, *b*, *t*, *n*, *f*)

$$\boldsymbol{P} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{r-l} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

- Orthographic projection

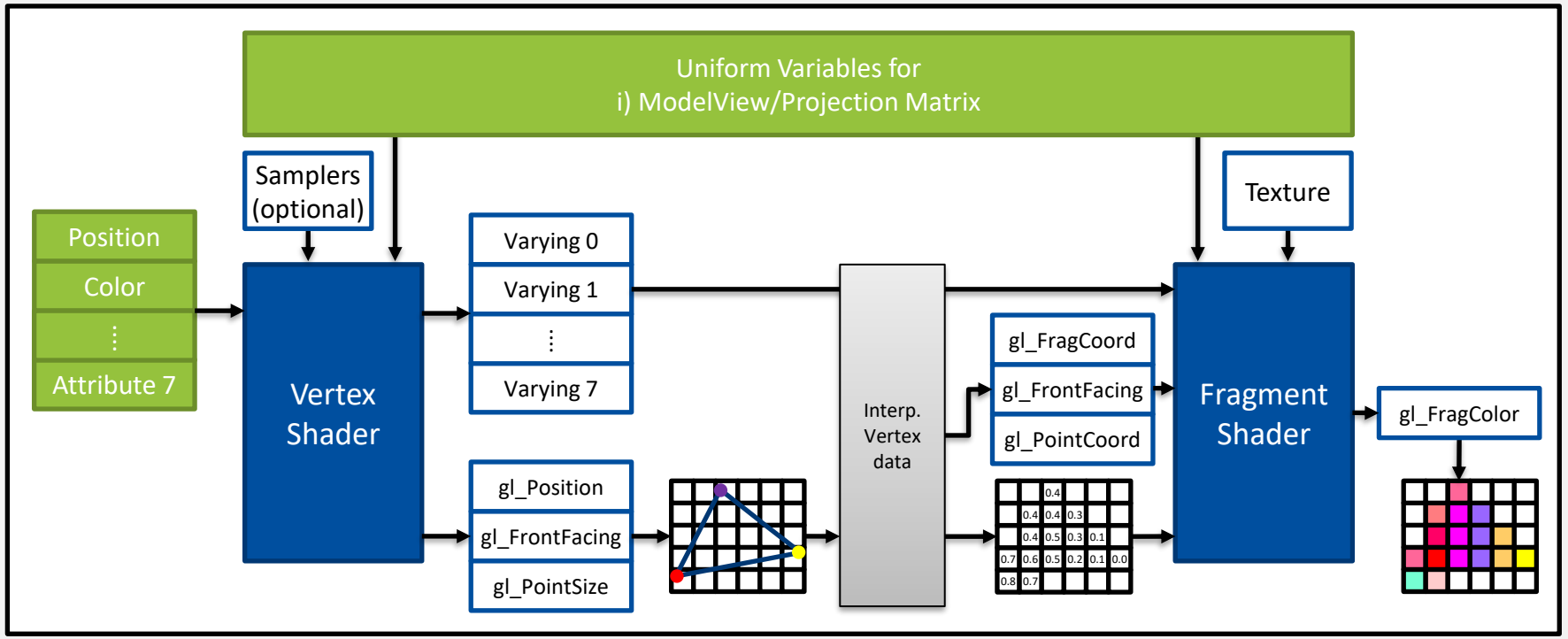



glOrtho(*l*, *r*, *b*, *t*, *n*, *f*)

$$\boldsymbol{P} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & \frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & \frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & \frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

# Simple Rendering w/ Objects and a Camera

- Strategy for programmable rendering pipeline
  - Vertex shader: computing change of coordinates
    - Uniforms: Matrix information for ModelView transformation and Projection
    - Attributes: Position, Color
  - Fragment shader: setting output (i.e., gl_FragColor) with interpolated per-vertex colors



# Practice – Simple Rendering in OpenGL 2.x+

- Modeling a triangular mesh
  - Per-vertex color
- Draw a mesh with OpenGL 2.x+ shaders & buffer objects
  - glGenBuffers()
  - glBindBuffer()
  - glBufferData()
  - glVertexAttribPointer()
  - glDrawArrays()

# Simple Rendering w/ Objects and a Camera

## Vertex / Fragment Shaders

```
/// Vertex Shader
#version 120                    // GLSL 1.20

// uniforms used by the vertex shader
uniform mat4 u_PVM;

// attributes input to the vertex shader
attribute vec3 a_position;
attribute vec3 a_color;    // input vertex color

// varying variables - input to the fragment shader
varying vec3 v_color;       // output vertex color

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;

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

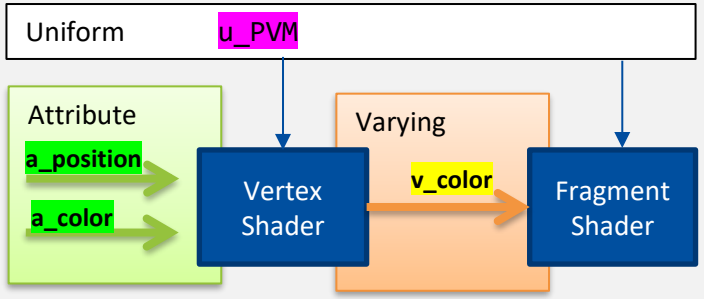
## OpenGL 2.x+ codes (C/C++)

```
// Texture object handle
GLuint  vertexShader, fragmentShader, programObject;
GLuint  loc_a_position, loc_a_color, loc_u_PVM;
GLuint  position_buffer, color_buffer;

void init()
{
 // Compile sahders and Link them into the program
  programObject = glCreateProgram();
  vertexShader   = glCreateShader(GL_VERTEX_SHADER);
  fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);

  // get uniform / vertex attribute locations
  loc_u_PVM      = glGetUniformLocation(programObject, "u_PVM");
  loc_a_position = glGetAttribLocation(programObject, "a_position");
  loc_a_color    = glGetAttribLocation(programObject, "a_color");

  glLinkProgram(programObject);
}

void set_gl_buffers ()
{
  std::vector<glm::vec3>  tv_positions;    // You set per triangle-vertex 3D position
  std::vector<glm::vec3>  tv_colors;       // You set per triangle-vertex 3D color

  glGenBuffers(1, &position_buffer);
  glBindBuffer(GL_ARRAY_BUFFER, position_buffer);
  glBufferData(GL_ARRAY_BUFFER, sizeof(glm::vec3)*tv_positions.size(), &tv_positions[0], GL_STATIC_DRAW);

  glGenBuffers(1, &color_buffer);
  glBindBuffer(GL_ARRAY_BUFFER, color_buffer);
  glBufferData(GL_ARRAY_BUFFER, sizeof(glm::vec3)*tv_colors.size(), &tv_colors[0], GL_STATIC_DRAW);
}

void render()
{
  glm::mat4   mat_PMV;  // You compute PVM matrix, for example, using glm

  glUniformMatrix4fv(loc_u_PVM, 1, GL_FLASE, glm::value_ptr(mat_PVM));

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

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

  glDrawArrays(...);

  glDisableVertexAttribArray(loc_a_position);
  glDisableVertexAttribArray(loc_a_color);
}
```

# Simple Shader

## Shader

- Uniform / attribute
  - model-view-projection matrix
  - vertex, color
- Vertex shader
  - Transform vertex
  - Send color
- Fragment shader
  - Set fragment color




## Vertex / Fragment Shaders

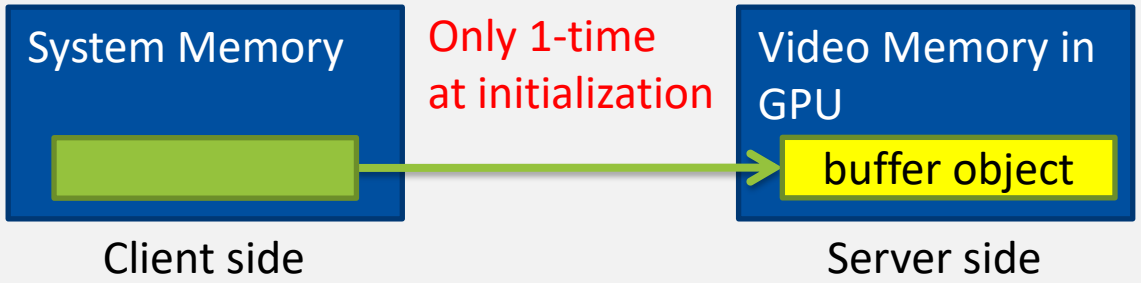
```
/// Vertex Shader
#version 120                    // GLSL 1.20

// uniforms used by the vertex shader
uniform mat4 u_PVM;

// attributes input to the vertex shader
attribute vec3 a_position;
attribute vec3 a_color;     // input vertex color

// varying variables - input to the fragment shader
varying vec3 v_color;        // output vertex color

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;

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

# Simple Shader

## Mesh data

- Vertex & color data



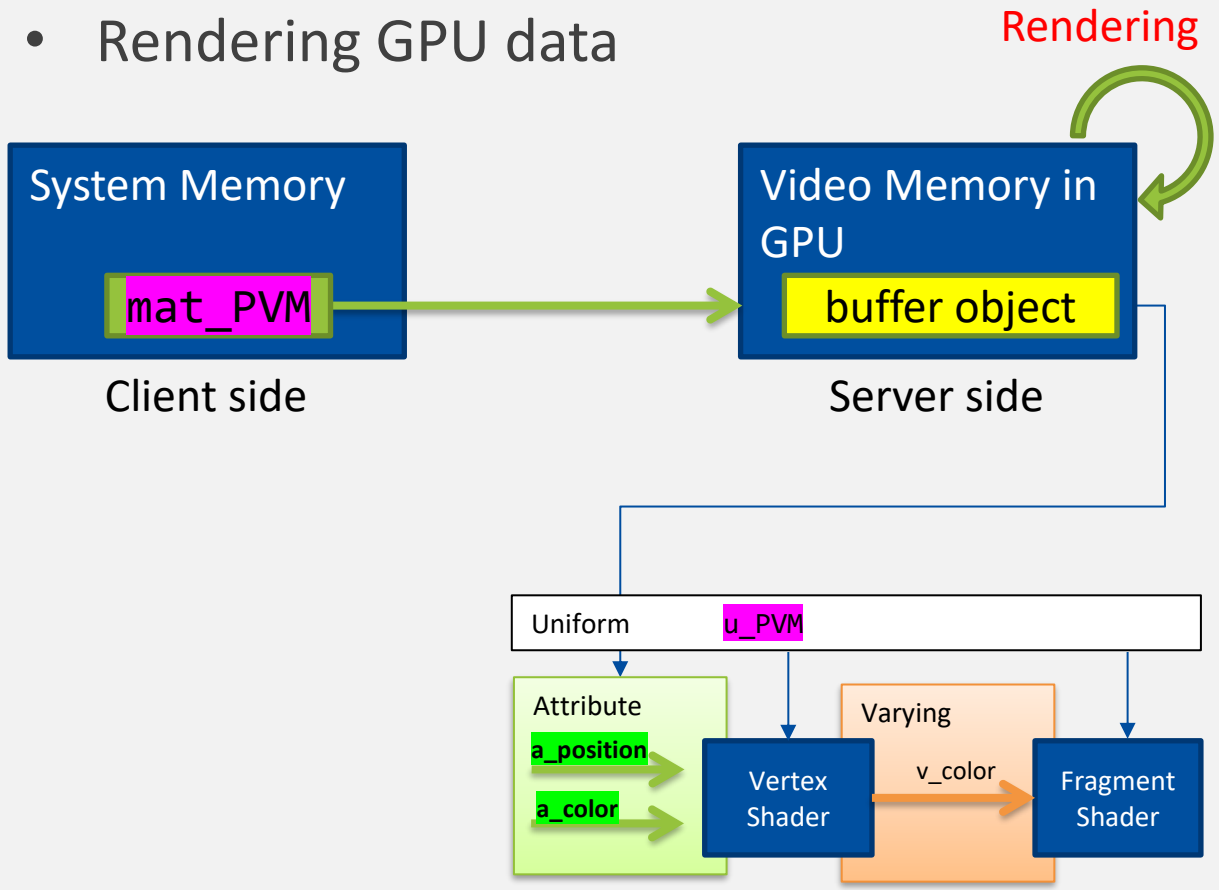

## C++ codes

```
GLuint  position_buffer, color_buffer;

void set_gl_buffers ()
{
  std::vector<glm::vec3>  tv_positions;
  std::vector<glm::vec3>  tv_colors;

  // You set per triangle-vertex 3D position and color
  // ......

  glGenBuffers(1, &position_buffer);
  glBindBuffer(GL_ARRAY_BUFFER, position_buffer);
  glBufferData(GL_ARRAY_BUFFER, sizeof(glm::vec3)*tv_positions.size(),
               &tv_positions[0], GL_STATIC_DRAW);

  glGenBuffers(1, &color_buffer);
  glBindBuffer(GL_ARRAY_BUFFER, color_buffer);
  glBufferData(GL_ARRAY_BUFFER, sizeof(glm::vec3)*tv_colors.size(),
               &tv_colors[0], GL_STATIC_DRAW);
}
```

# Simple Shader

## Mesh data

- Rendering GPU data




## C++ codes

```
GLuint  position_buffer, color_buffer;

void render()
{
  glm::mat4   mat_PMV;  // You compute PVM matrix, for example, using glm

  glUniformMatrix4fv(loc_u_PVM, 1, GL_FLASE, glm::value_ptr(mat_PVM));

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

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

  glDrawArrays(...);

  glDisableVertexAttribArray(loc_a_position);
  glDisableVertexAttribArray(loc_a_color);
}
```

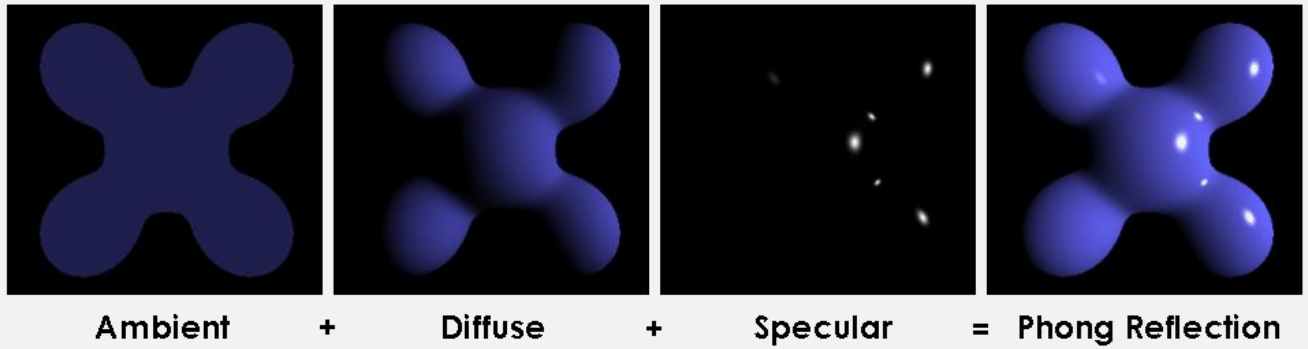
# Simple Shader

- Modeling a triangular mesh
  - Per-vertex color
- Draw a mesh with OpenGL 2.x+ shaders & buffer objects
  - glGenBuffers()
  - glBindBuffer()
  - glBufferData()
  - glVertexAttribPointer()
  - glDrawArrays()

# Per-fragment Lighting with Programmable Rendering Pipeline

# Per-fragment Lighting

- Phong reflection model

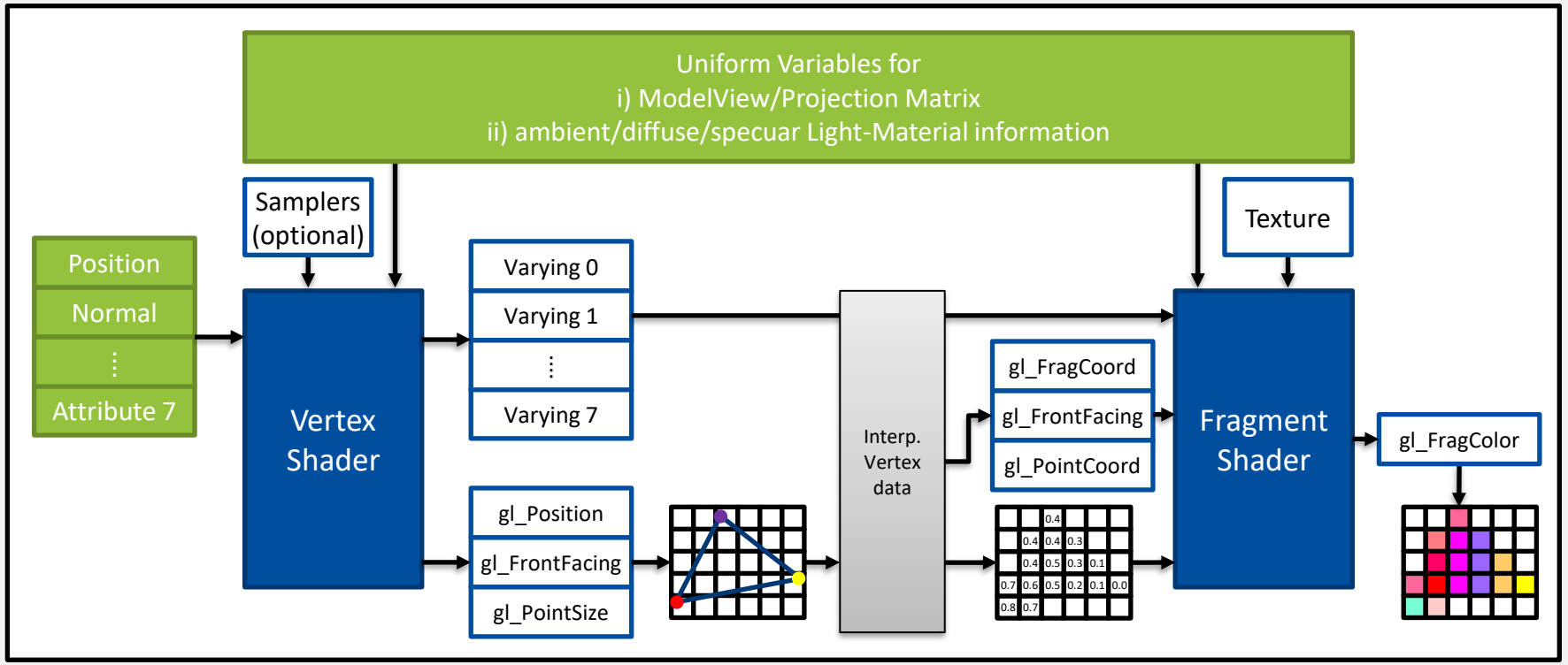



$$I = \kappa_a L_a + \left(\kappa_d \max\left((\boldsymbol{l} \cdot \boldsymbol{n}), 0\right) L_d \ + \kappa_s \max\left((\boldsymbol{r} \cdot \boldsymbol{v})^{\alpha}, 0\right) L_s\right)$$

- Per-fragment lighting w/ Phong reflection model
  - Material-light interaction should be computed in the fragment shader
    - cf) In fixed-rendering pipeline, material-light interaction is computed in the vertex processor

# Per-fragment Lighting

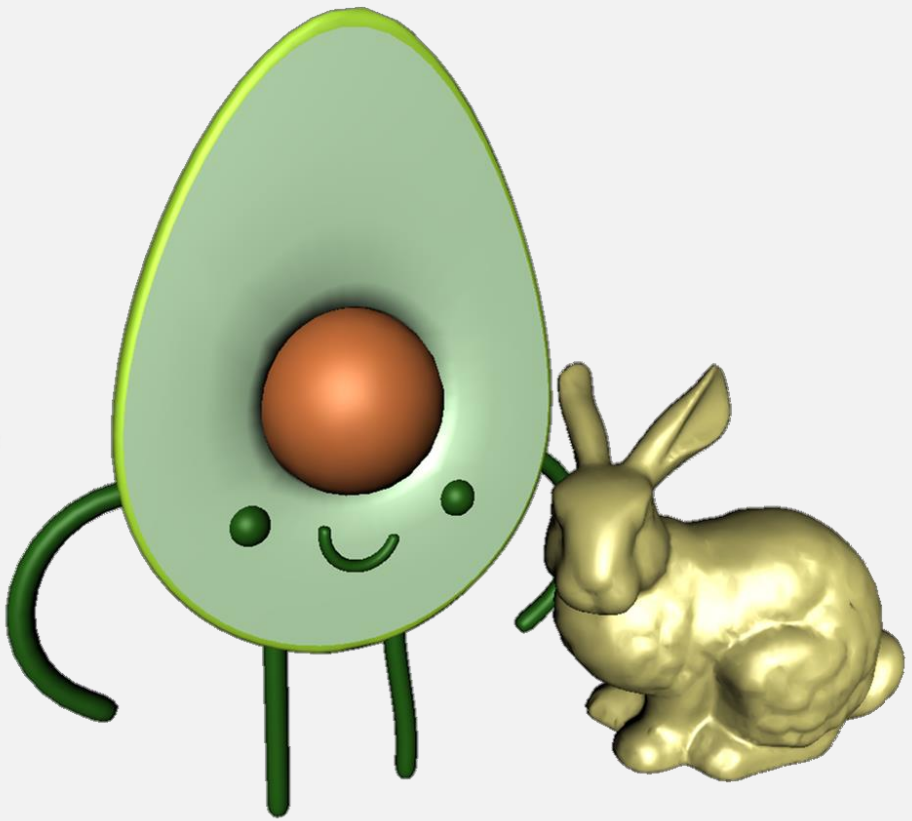
- Strategy for programmable rendering pipeline
  - Vertex shader: typical vertex processing
  - Fragment shader: computing Phong reflection model



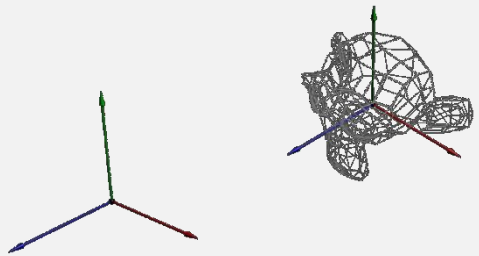
# Practice – Per-Fragment (Pixel) Lighting

- Based on Phong reflection model
  - Not Phong-Blinn model
  - Directional light
  - Ambient, diffuse, specular light & material

$$I = \frac{1}{a + bd + cd^2}\left(k_d \max\left((\mathbf{l}\bullet\mathbf{n}), 0\right) L_d + k_s \max\left((\mathbf{r}\bullet\mathbf{v})^{\alpha}, 0\right) L_s\right) + k_a L_a$$

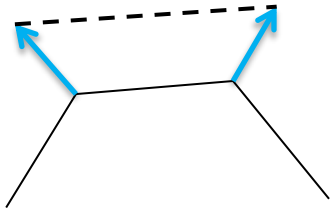


# Per-Fragment (Pixel) Lighting



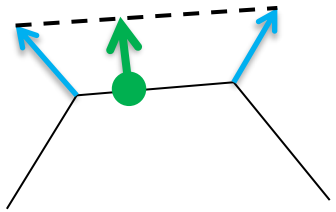
## Vertex shader

```
#version 120                        // GLSL 1.20

attribute vec3 a_position;     // per-vertex position
attribute vec3 a_normal;

uniform mat4 u_PVM;

// for phong shading
uniform mat4 u_model_matrix;
uniform mat3 u_normal_matrix;

varying vec3 v_position_wc; // world coordinate
varying vec3 v_normal_wc;    // world coordinate


void main()
{
  gl_Position = u_PVM * vec4(a_position, 1.0f);

  v_position_wc = (u_model_matrix * vec4(a_position, 1.0f)).xyz;
  v_normal_wc    = normalize(u_normal_matrix * a_normal);
}
```



## Fragment shader

```
#version 120                        // GLSL 1.20

varying   vec3 v_normal_wc;
varying   vec3 v_position_wc;

uniform vec3 u_light_position;
uniform vec3 u_light_ambient;
uniform vec3 u_light_diffuse;
uniform vec3 u_light_specular;

uniform vec3  u_obj_ambient;
uniform vec3  u_obj_diffuse;
uniform vec3  u_obj_specular;
uniform float u_obj_shininess;

uniform vec3 u_camera_position;
uniform mat4 u_view_matrix;

vec3 directional_light()
{
    vec3 color = vec3(0.0);
    vec3 normal_wc = normalize(v_normal_wc);
    vec3 light_dir = normalize(u_light_position);

    // ambient
    color += (u_light_ambient * u_obj_ambient);

    // diffuse
    float ndotl = max(dot(normal_wc, light_dir), 0.0);
    color += (ndotl * u_light_diffuse * u_obj_diffuse);

    // specular
    vec3 view_dir = normalize(u_camera_position - v_position_wc);
    vec3 reflect_dir = reflect(-light_dir, normal_wc);

    float rdotv = max(dot(view_dir, reflect_dir), 0.0);
    color += (pow(rdotv, u_obj_shininess) * u_light_specular * u_obj_specular);


    return color;
}
void main()
{
    vec3 result = directional_light();
    gl_FragColor = vec4(result, 1.0f);
}
```



# Per-Fragment (Pixel) Lighting

## Vertex shader

- Transform vertex to view space
  - Model-view (inverse) matrix
- Normal matrix
  - Transform normal vector
  - Rotation, uniform scale matrix
  - Upper left of model-view matrix

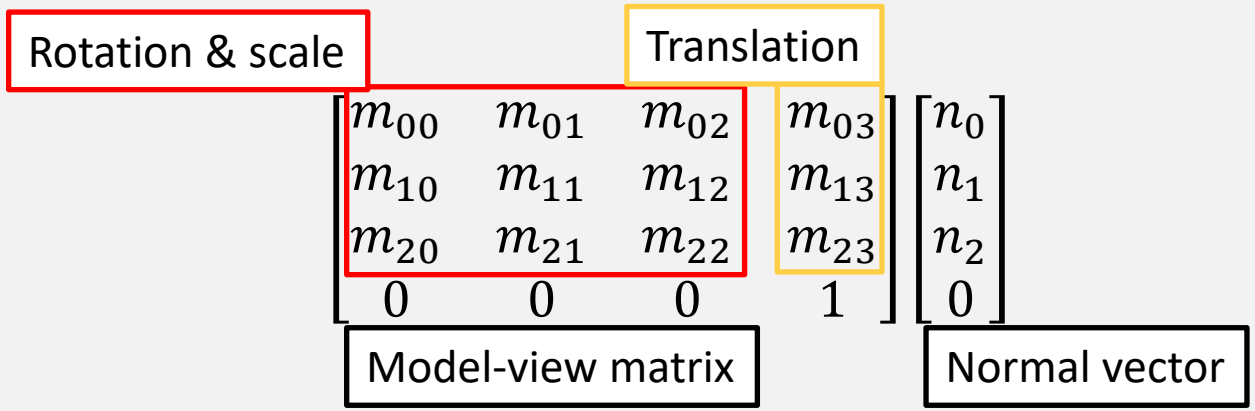
$$\begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

## Vertex shader

```
#version 120                    // GLSL 1.20

attribute vec3 a_position;      // per-vertex position
attribute vec3 a_normal;

uniform mat4 u_PVM;

// for phong shading
uniform mat4 u_model_matrix;
uniform mat3 u_normal_matrix;

varying vec3 v_position_wc; // world coordinate
varying vec3 v_normal_wc;   // world coordinate


void main()
{
  gl_Position = u_PVM * vec4(a_position, 1.0f);

  v_position_wc = (u_model_matrix * vec4(a_position, 1.0f)).xyz;
  v_normal_wc   = normalize(u_normal_matrix * a_normal);
}
```

# Normal Matrix

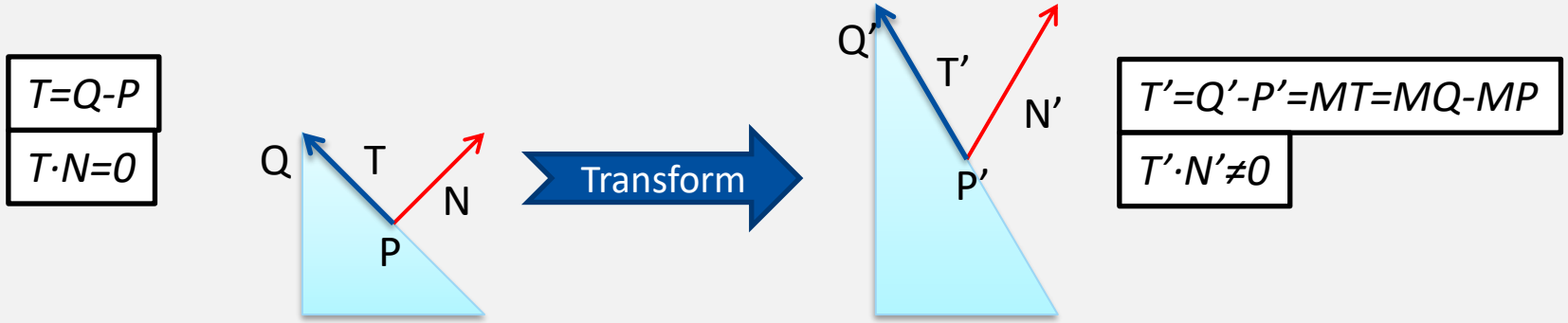
- Normal matrix
  - Transform a normal vector with a **model-view matrix**




  - Translation doesn't affect
  - Transform only orientation of a normal vector

# Normal Matrix

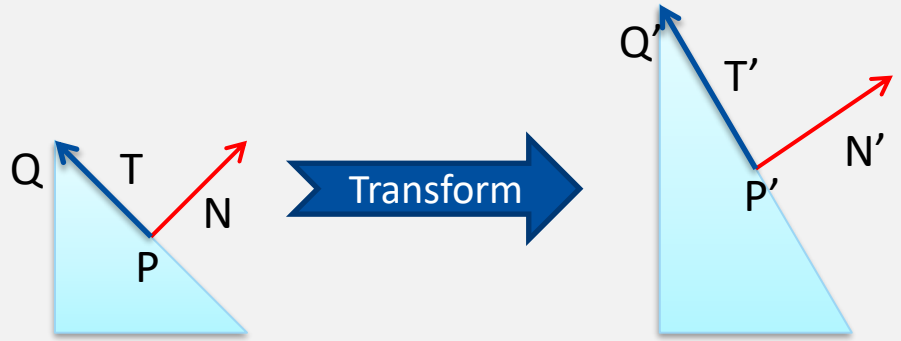
- Normal matrix
  - Transform a normal vector with a **model-view matrix**
  - But if model-view matrix contains non-uniform scale




  - Uniform scale can affect the length of the normal vector

# Normal Matrix

- Normal matrix




$$0 = N'^T \times T' = (R \times N^T) \times (M \times T) = (N^T \times R^T) \times (M \times T) = N^T \times (R^T \times M) \times T$$

$$N^T \times (R^T \times M) \times T = 0,\ N^T \times T = 0$$

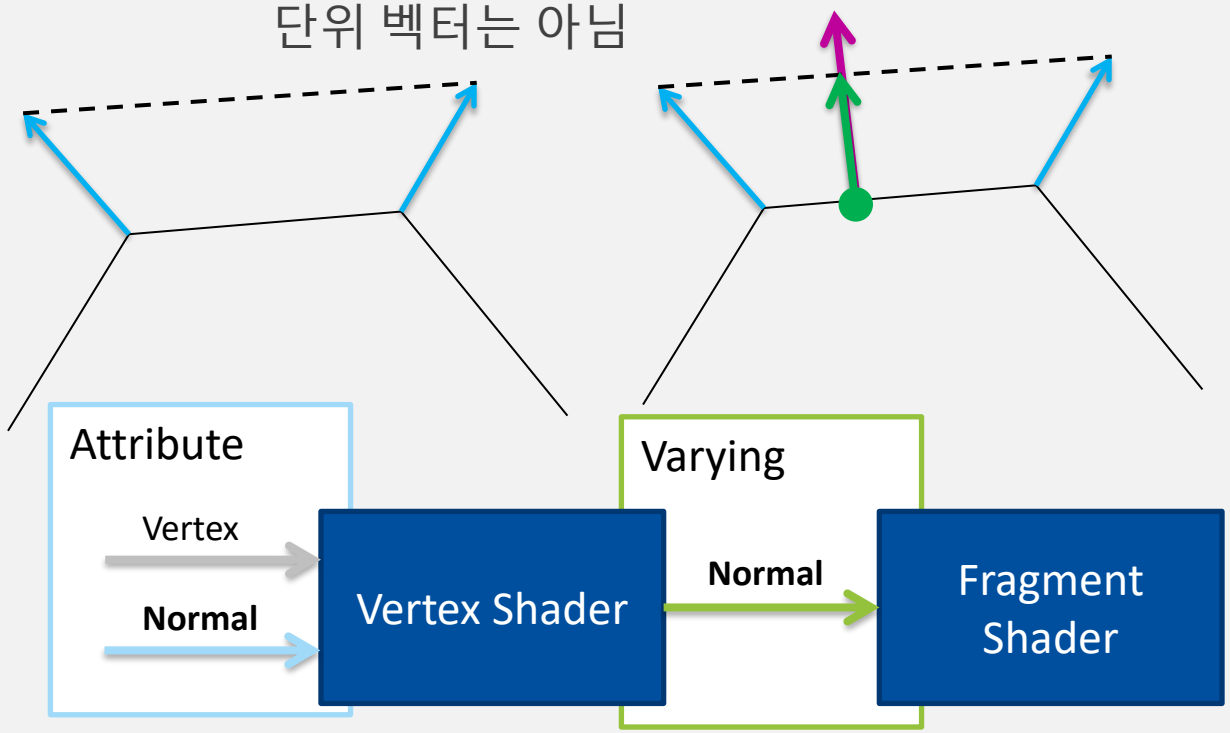
$$R^T \times M = I,\ R^T = M^{-1}$$

$$R = (M^{-1})^T = M$$

Should be orthonormal

# Normalization

- Vertex Shader에서 이미 정규화된 벡터를 fragment Shader에서 다시 정규화
  - 기준이 되는 법선 벡터는 정규화 해야함
  - 보간된 (interpolated) 벡터
    - 벡터가 올바른 방향을 갖는다 해도 단위 벡터는 아님




```glsl
#version 120                    // GLSL 1.20
varying   vec3 v_normal_wc;
varying   vec3 v_position_wc;

uniform vec3 u_light_position;
uniform vec3 u_light_ambient;
uniform vec3 u_light_diffuse;
uniform vec3 u_light_specular;

uniform vec3  u_obj_ambient;
uniform vec3  u_obj_diffuse;
uniform vec3  u_obj_specular;
uniform float u_obj_shininess;

uniform vec3 u_camera_position;
uniform mat4 u_view_matrix;

vec3 directional_light()
{
    vec3 color = vec3(0.0);
    vec3 normal_wc = normalize(v_normal_wc);
    vec3 light_dir = normalize(u_light_position);

    // ambient
    color += (u_light_ambient * u_obj_ambient);

    // diffuse
    float ndotl = max(dot(normal_wc, light_dir), 0.0);
    color += (ndotl * u_light_diffuse * u_obj_diffuse);

    // specular
    vec3 view_dir = normalize(u_camera_position - v_position_wc);
    vec3 reflect_dir = reflect(-light_dir, normal_wc);

    float rdotv = max(dot(view_dir, reflect_dir), 0.0);
    color += (pow(rdotv, u_obj_shininess) * u_light_specular * u_obj_specular);


    return color;
}
void main()
{
    vec3 result = directional_light();
    gl_FragColor = vec4(result, 1.0f);
}
```

# Per-Fragment (Pixel) Lighting

## Fragment shader

- Lighting
  - Directional light
  - Light position
  - Ambient, diffuse, specular light & material
  - Normal, light direction, reflection, view direction vectors

## Fragment shader

```
#version 120                    // GLSL 1.20

varying   vec3 v_normal_wc;
varying   vec3 v_position_wc;

uniform vec3 u_light_position;
uniform vec3 u_light_ambient;
uniform vec3 u_light_diffuse;
uniform vec3 u_light_specular;

uniform vec3  u_obj_ambient;
uniform vec3  u_obj_diffuse;
uniform vec3  u_obj_specular;
uniform float u_obj_shininess;

uniform vec3 u_camera_position;
uniform mat4 u_view_matrix;

vec3 directional_light()
{
    vec3 color = vec3(0.0);
    vec3 normal_wc = normalize(v_normal_wc);
    vec3 light_dir = normalize(u_light_position);

    // ambient
    color += (u_light_ambient * u_obj_ambient);

    // diffuse
    float ndotl = max(dot(normal_wc, light_dir), 0.0);
    color += (ndotl * u_light_diffuse * u_obj_diffuse);

    // specular
    vec3 view_dir = normalize(u_camera_position - v_position_wc);
    vec3 reflect_dir = reflect(-light_dir, normal_wc);

    float rdotv = max(dot(view_dir, reflect_dir), 0.0);
    color += (pow(rdotv, u_obj_shininess) * u_light_specular * u_obj_specular);

    return color;
}
void main()
{
    vec3 result = directional_light();
    gl_FragColor = vec4(result, 1.0f);
}
```

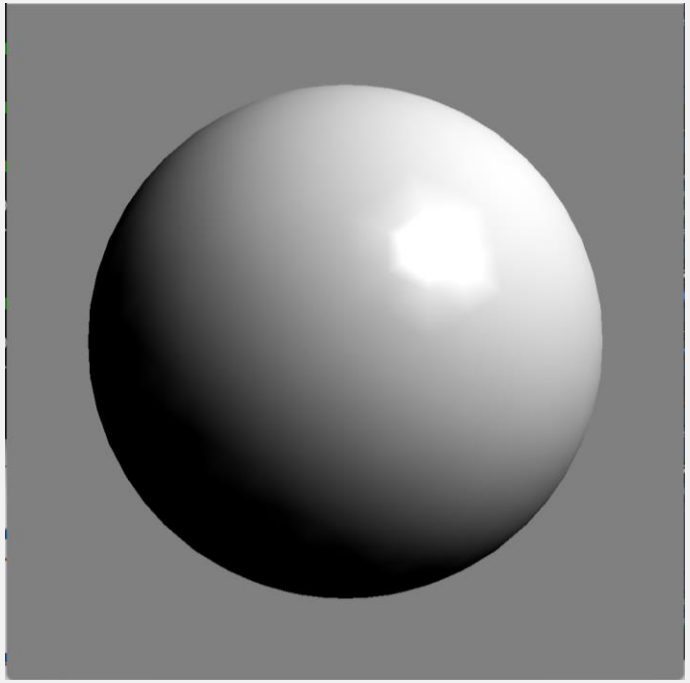
$\kappa_a L_a$

$\kappa_d \max\big((\boldsymbol{l} \cdot \boldsymbol{n}), 0\big) L_d$

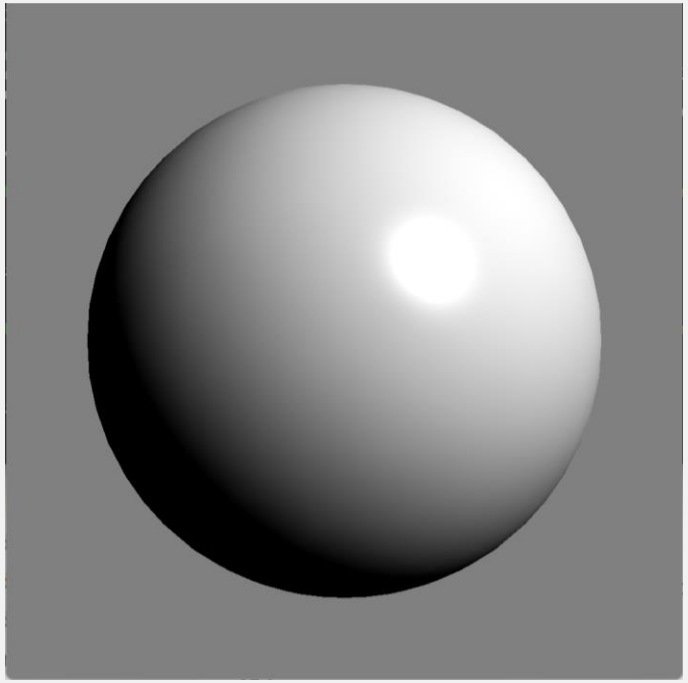
$\kappa_s \max\big((\boldsymbol{r} \cdot \boldsymbol{v})^{\alpha}, 0\big) L_s$

# Per-Fragment (Pixel) Lighting

- Per-vertex lighting vs. Per-Fragment (Pixel) Lighting for the same mesh



Per-vertex lighting

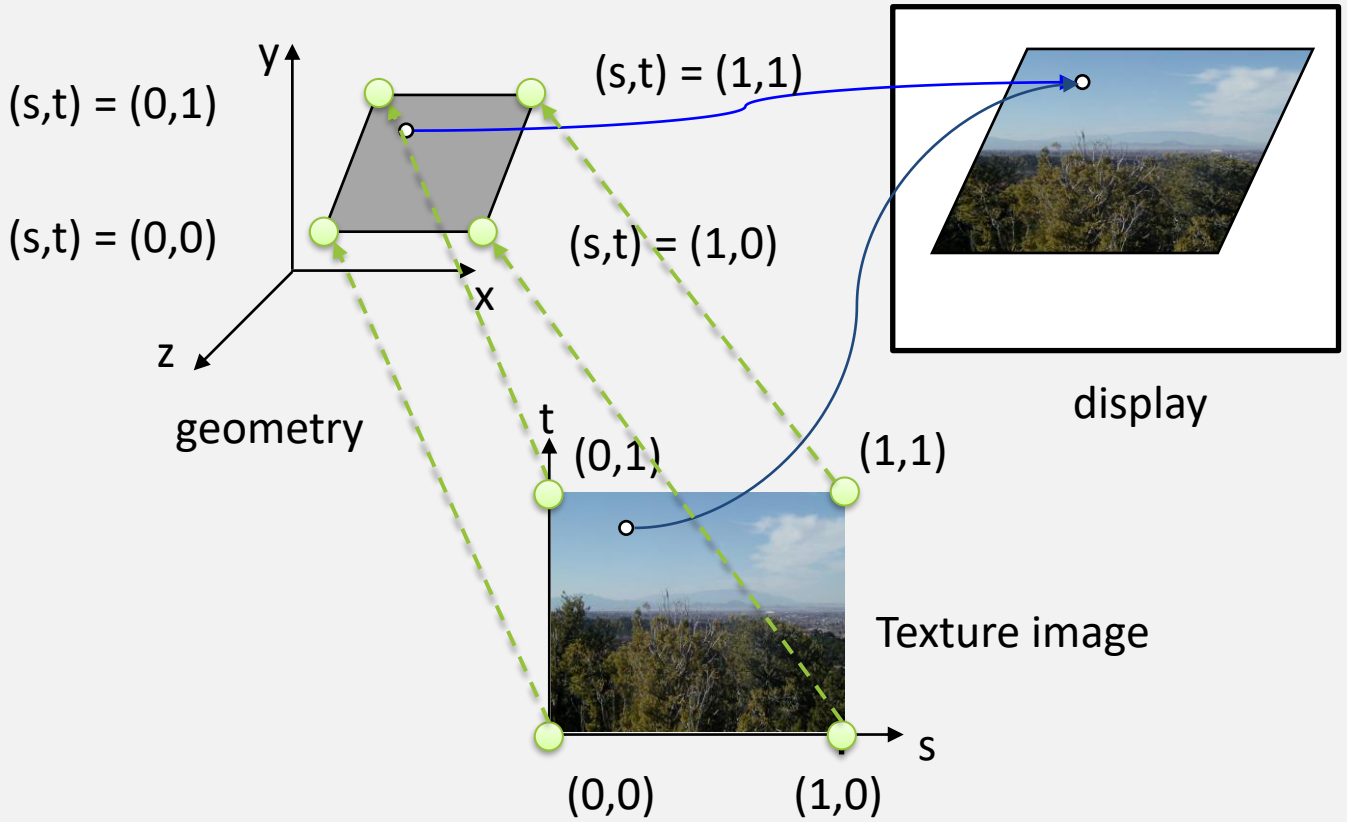


Per-fragment lighting

# Texture Mapping with Programmable Rendering Pipeline

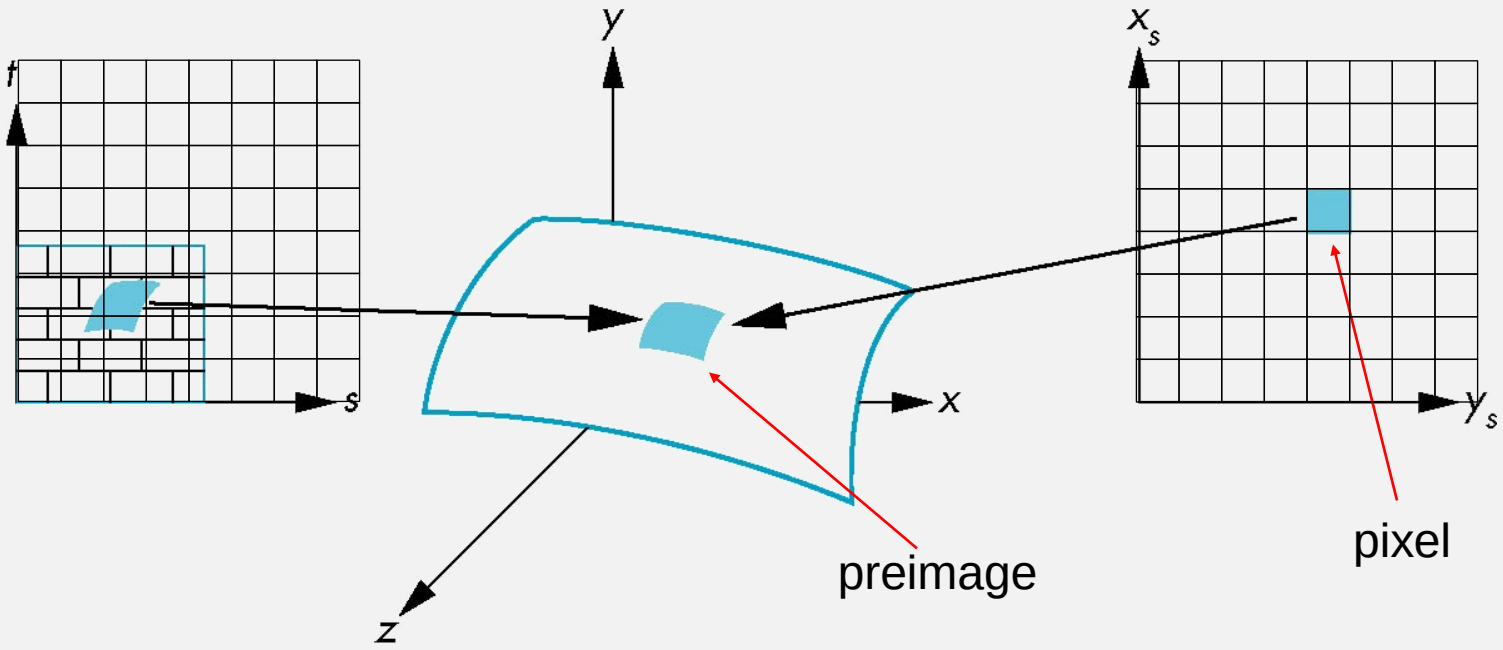
# Texture Mapping

- We need to specify per-vertex texture coordinates



# Sampling Problem

- A pixel must have one color value!!!
  - A pixel may correspond to several texels
  - A pixel may correspond to a small portion of a texel



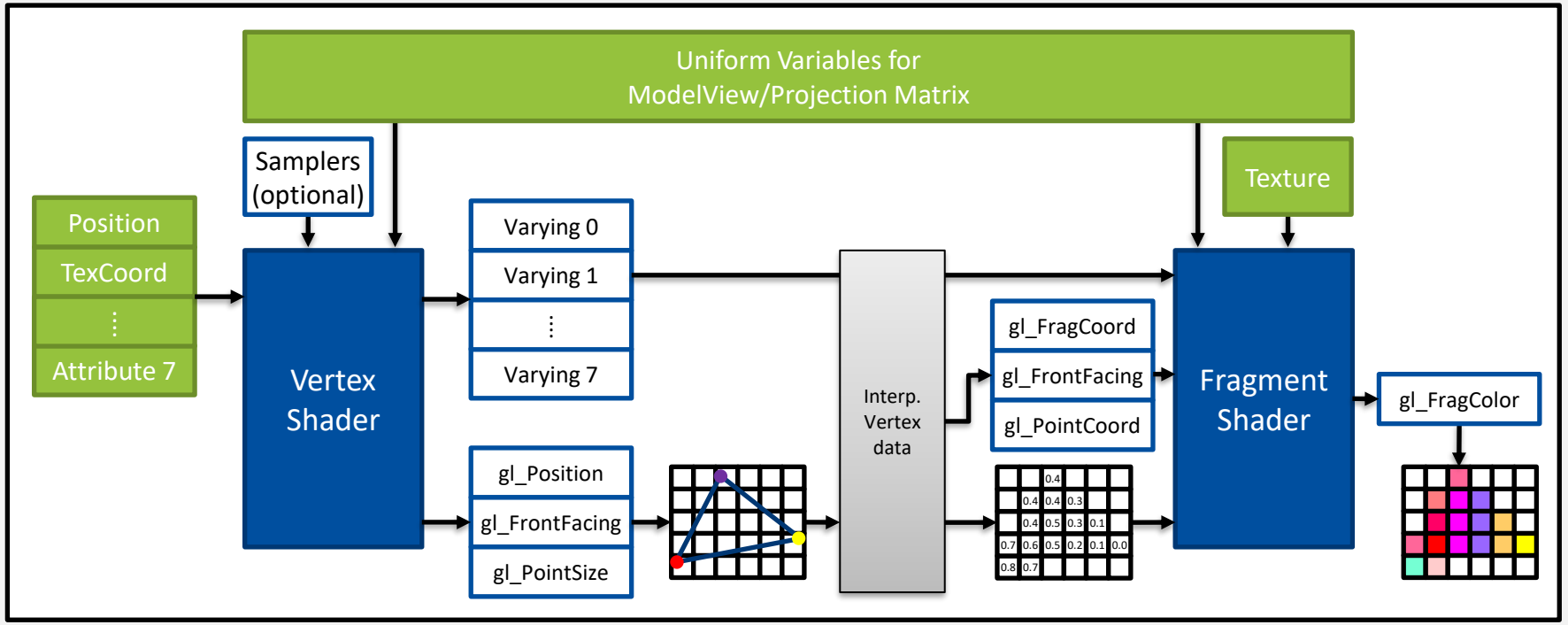
# Sampler

- A specific type of uniforms that represent a single texture of a particular texture type
  - It should be initialized with the OpenGL 2.x+ API
    - It also can be declared as function parameters in Shaders
  - Used with the built-in texture lookup functions
    - texture2D(), texture2DProj(), textureCube()

```
// Use the texture coordinate coord to do a texture lookup in the 2D texture currently bound to sampler
// sampler              the sampler to which the texture from which texels will be retrieved in bound
// coord                the texture coordinates at which texture will be sampled
// bias                 an optional parameter that provides a mipmap bias to be applied during LOD computation
vec4 texture2D(sampler2D sampler, vec2 coord, [float bias]);

// coord                the texture coordinates at which texture will be sampled
                        The (x,y) arguments are divided by (z) such that the fetch occurs at (x/z, y/z)
vec4 textureProj(samplerCube sampler, vec3 coord, [float bias]);


// Use the texture coordinate coord to do a texture lookup in the cubemap texture currently bound to sampler
vec4 textureCube(samplerCube sampler, vec3 coord, [float bias]);
```

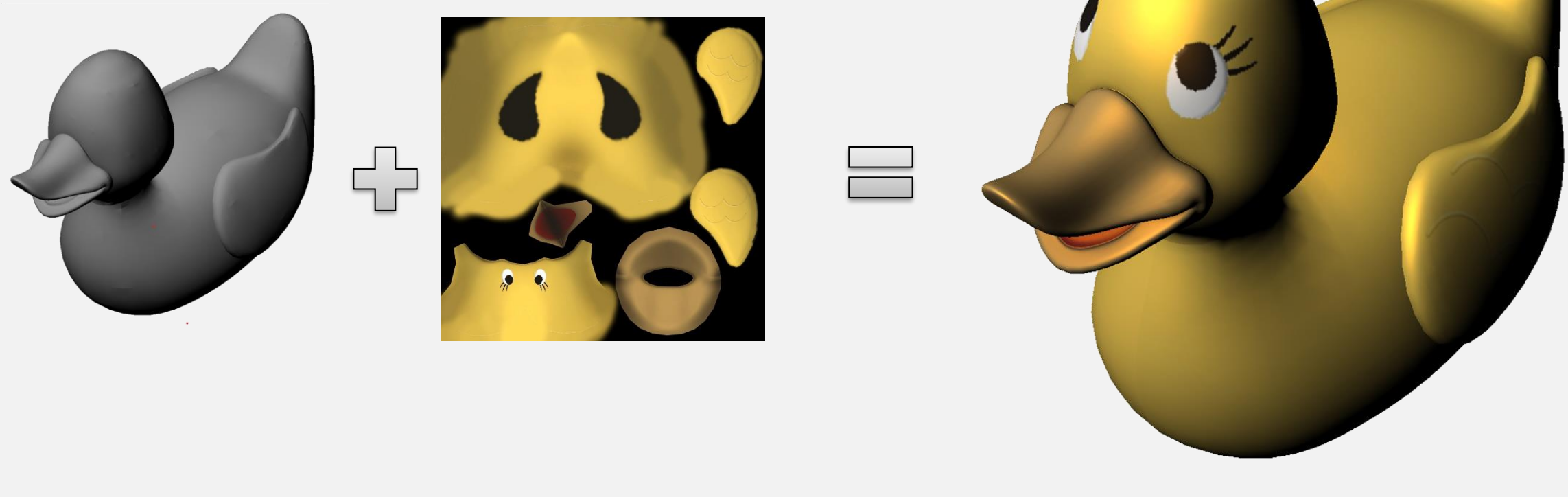
# Texture Mapping with Sampler

- Strategy for programmable rendering pipeline
  - Vertex shader: typical vertex processing
    - Attributes: Position, TexCoord
    - Varying: Passing through TexCoord to the fragment shader as varying variables
  - Fragment shader: computing per-fragment color with (uniform) sampler & (varying) TexCoord



# Practice – Texture Mapping with Sampler

- Texture mapping with OpenGL 2.x+



# Texture Mapping with Sampler

## Vertex / Fragment Shaders

```
#version 120                    // GLSL 1.20

uniform mat4 u_PVM;
attribute vec3 a_position;      // per-vertex position (per-vertex input)
attribute vec2 a_texcoord;

varying vec2 v_texcoord;

void main()
{
  gl_Position = u_PVM * vec4(a_position, 1.0f);

  v_texcoord = a_texcoord;
}
```

```
#version 120                    // GLSL 1.20

// texture mapping
uniform sampler2D s_texture;

varying   vec2 v_texcoord;
void main()
{
    gl_FragColor = texture2D(s_texture, v_texcoord);
}
```

## OpenGL 2.x codes (C/C++)

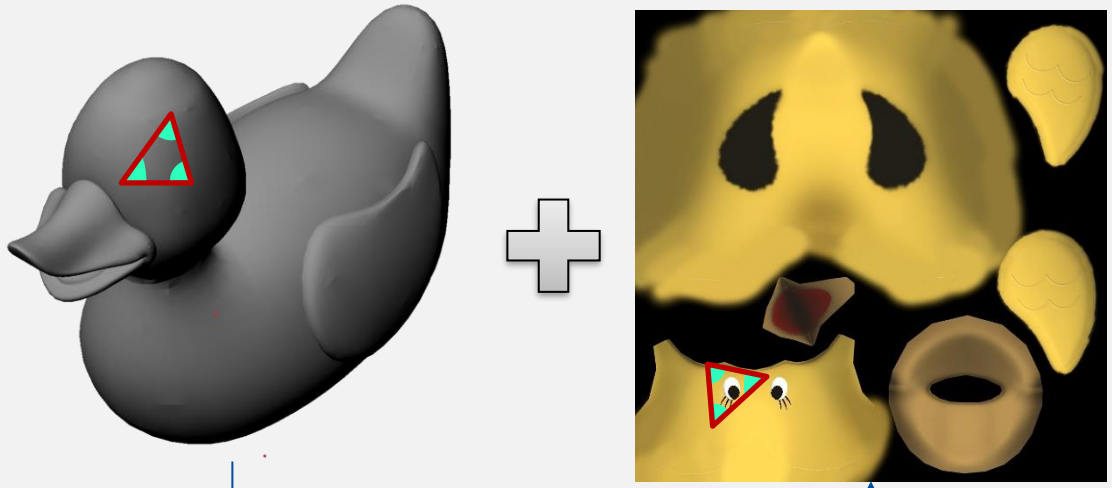
```
// Texture object handle
GLuint tex_id;
GLuint loc_a_position, loc_a_texcoord, loc_s_texture;

void init()
{
  // Compile sahders and Link them into the program
 loc_a_position = glGetAttribLocation(program, "a_position");
 loc_a_texcoord = glGetAttribLocation(program, "a_texcoord");
 loc_s_texture = glGetUniformLocation(program, "s_texture");
}

void set_texture2D()
{
  unsigned char* image;
  // Fill image contents
  image = stbi_load(img_filepath.c_str(), &width, &height, &channels, STBI_rgb);

  glGenTextures(1, &tex_id);
  glBindTexture(GL_TEXTURE_2D, tex_id);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, image);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
  glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
}

void render()
{
  // Set the sampler to texture unit 0
  glUniform1i(loc_s_texture, 0);
  glActiveTexture(GL_TEXTURE0);
  glBindTexture(GL_TEXTURE_2D, tex_id);

  // texcoord
  glBindBuffer(GL_ARRAY_BUFFER, texcoord_buffer_);
  glEnableVertexAttribArray(program.loc_a_texcoord);
  glVertexAttribPointer(program.loc_a_texcoord, 2, GL_FLOAT, GL_FALSE, 0, (void*)0);
}
```

# Practice – Texture Mapping with Sampler

- Texture mapping with OpenGL 2.x+

```
attribute vec2 a_texcoord;
```

```
glBindTexture(GL_TEXTURE_2D, tex_id);
```



```
uniform sampler2D s_texture;

glUniform1i(loc_s_texture, 0);
glActiveTexture(GL_TEXTURE0);
```