COSE436

Lecture 10: OpenGL Shading Language

Oct 17, 2024

Won-Ki Jeong
(wkjeong@korea.ac.kr)

KOREA UNIVERSITY

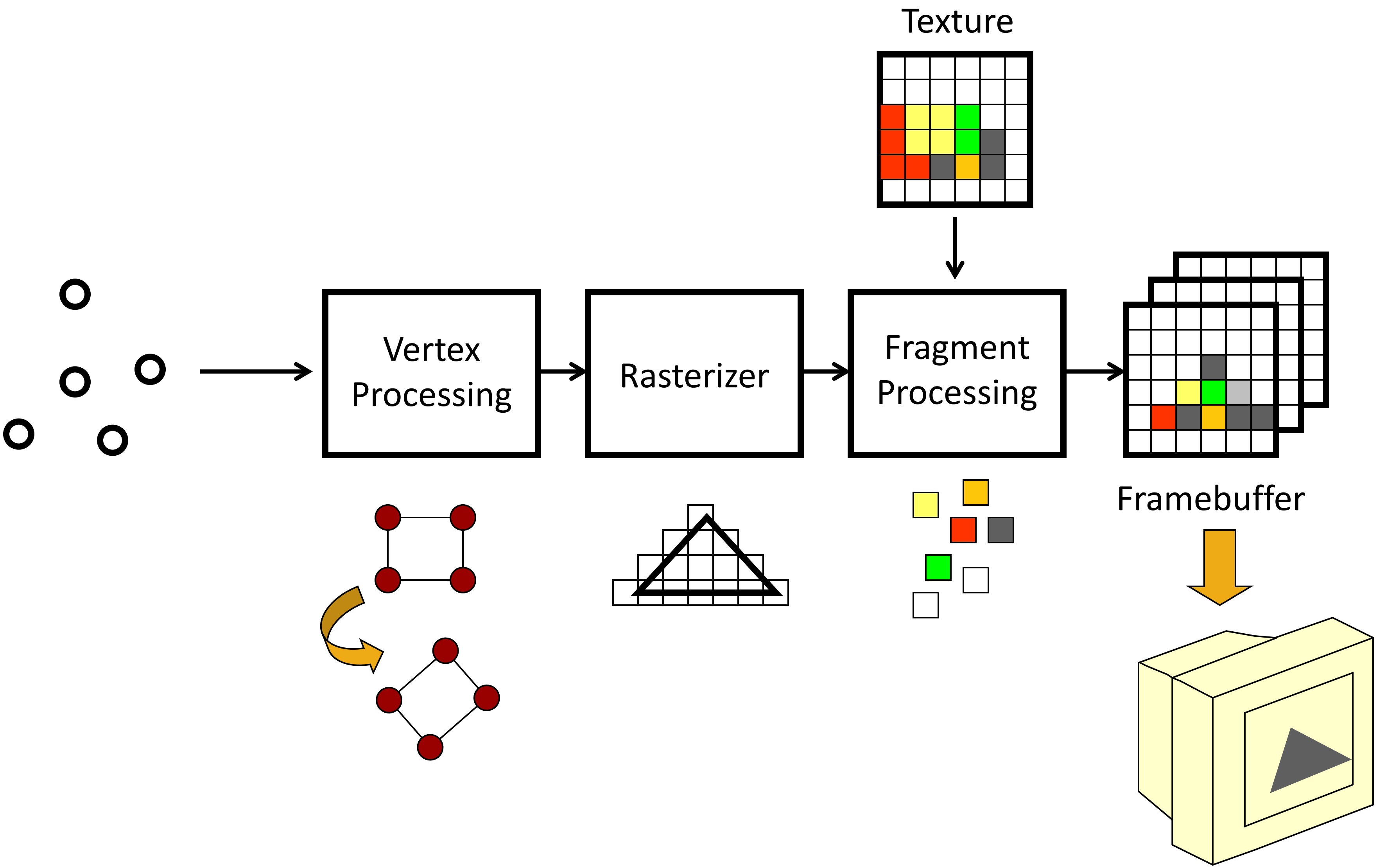
Outline

- Conventional vs. Programmable graphics pipeline
- GLSL basics
- GLSL examples

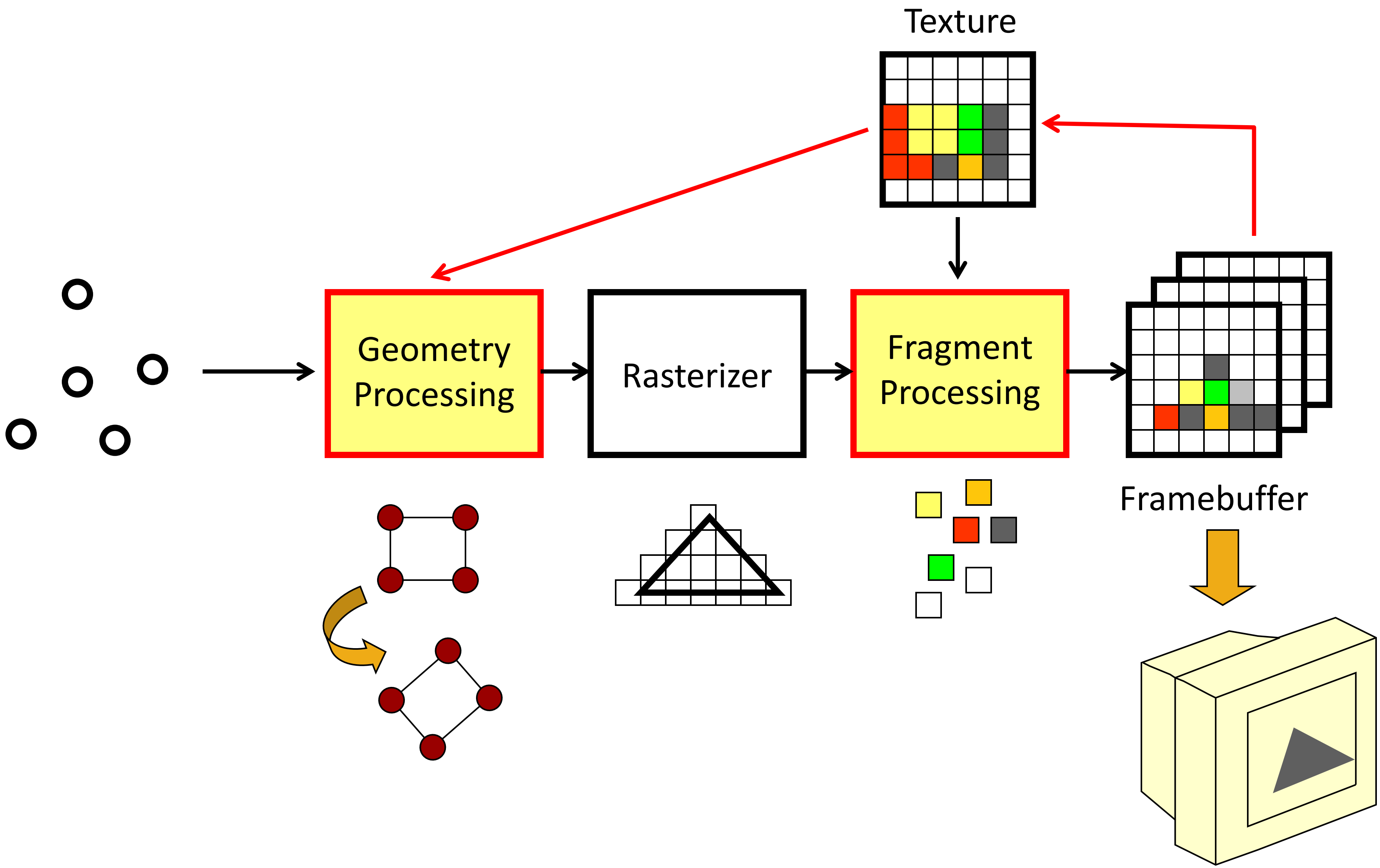
Outline

- Conventional vs. Programmable graphics pipeline
- GLSL basics
- GLSL examples

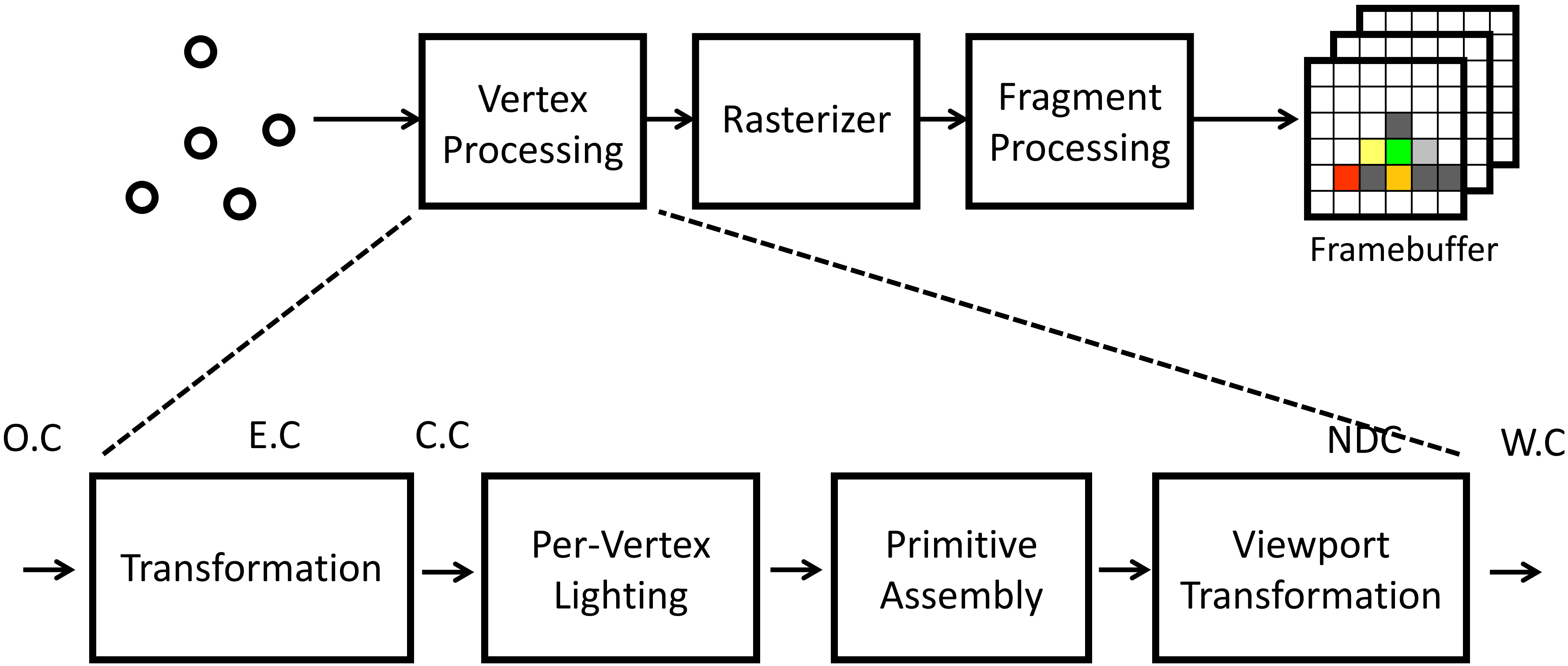
Conventional Graphics Pipeline



Programmable Graphics Pipeline



Vertex Processing (C.P)

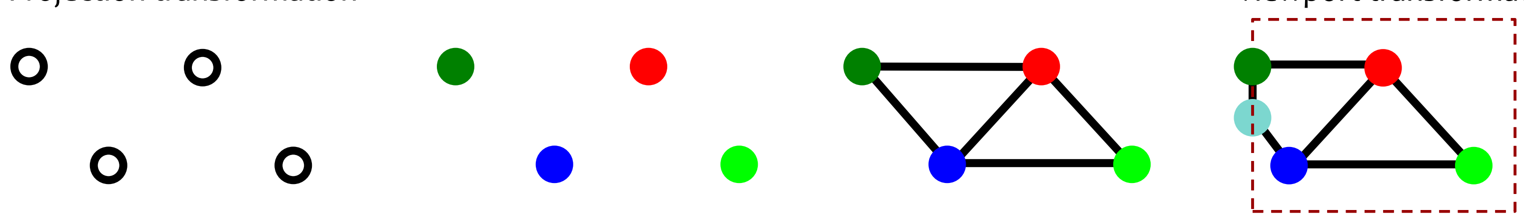


- Modeling transformation
- Viewing transformation
- Projection transformation

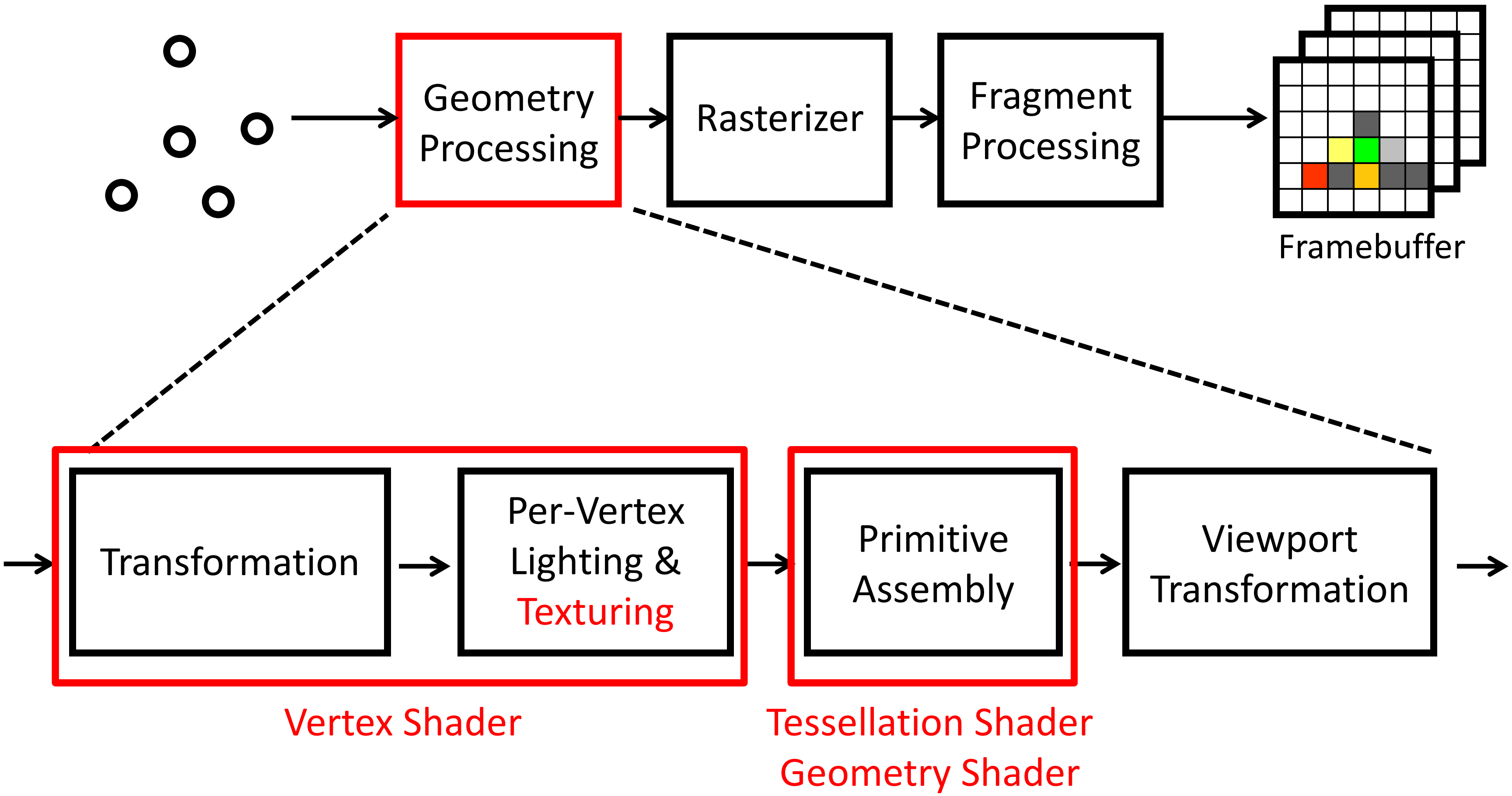
- Per-vertex shading (Blinn-Phong model)

- Geometric primitives (points, lines, triangles,..)

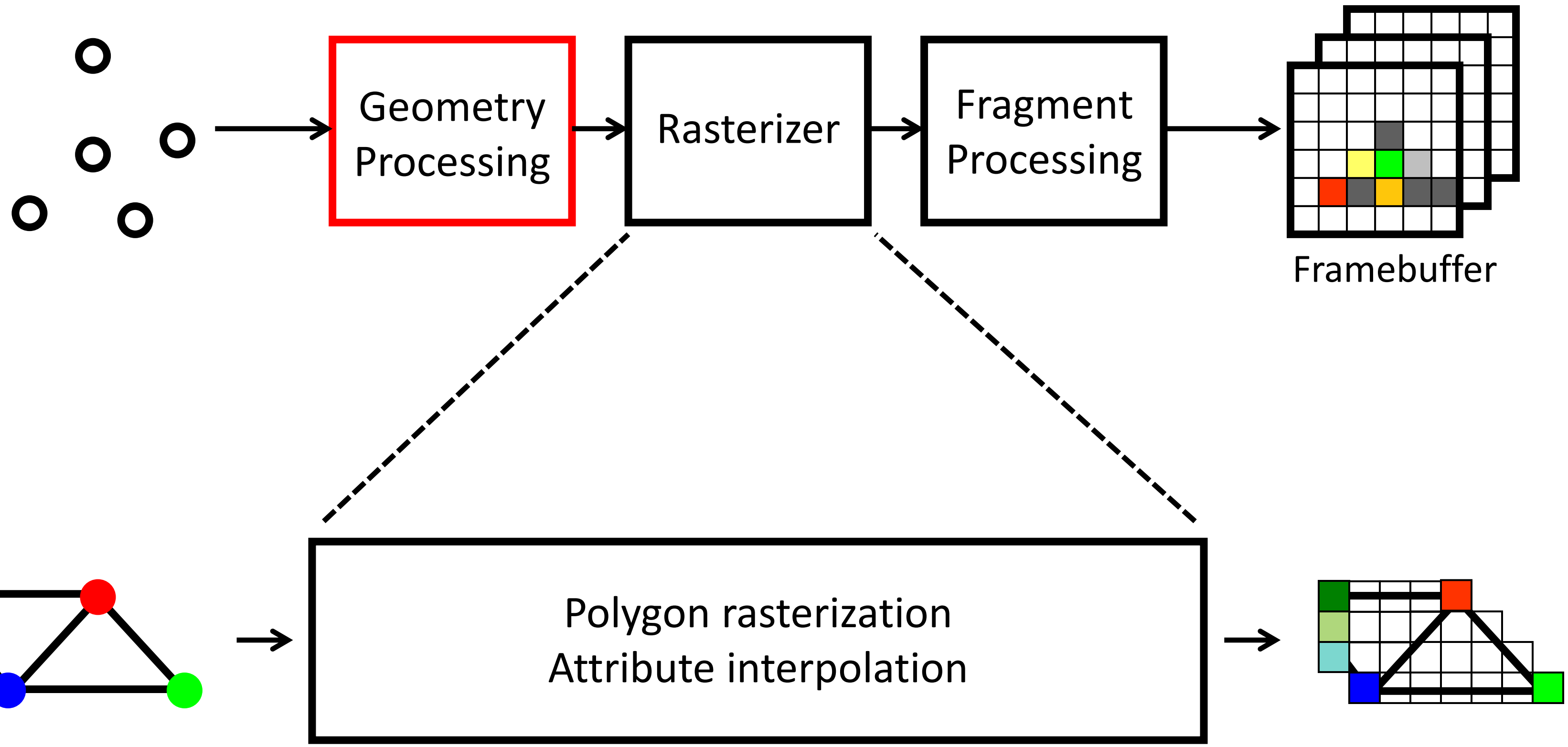
- Clipping
- Perspective division
- Viewport transformation



Geometry Processing (P.P)

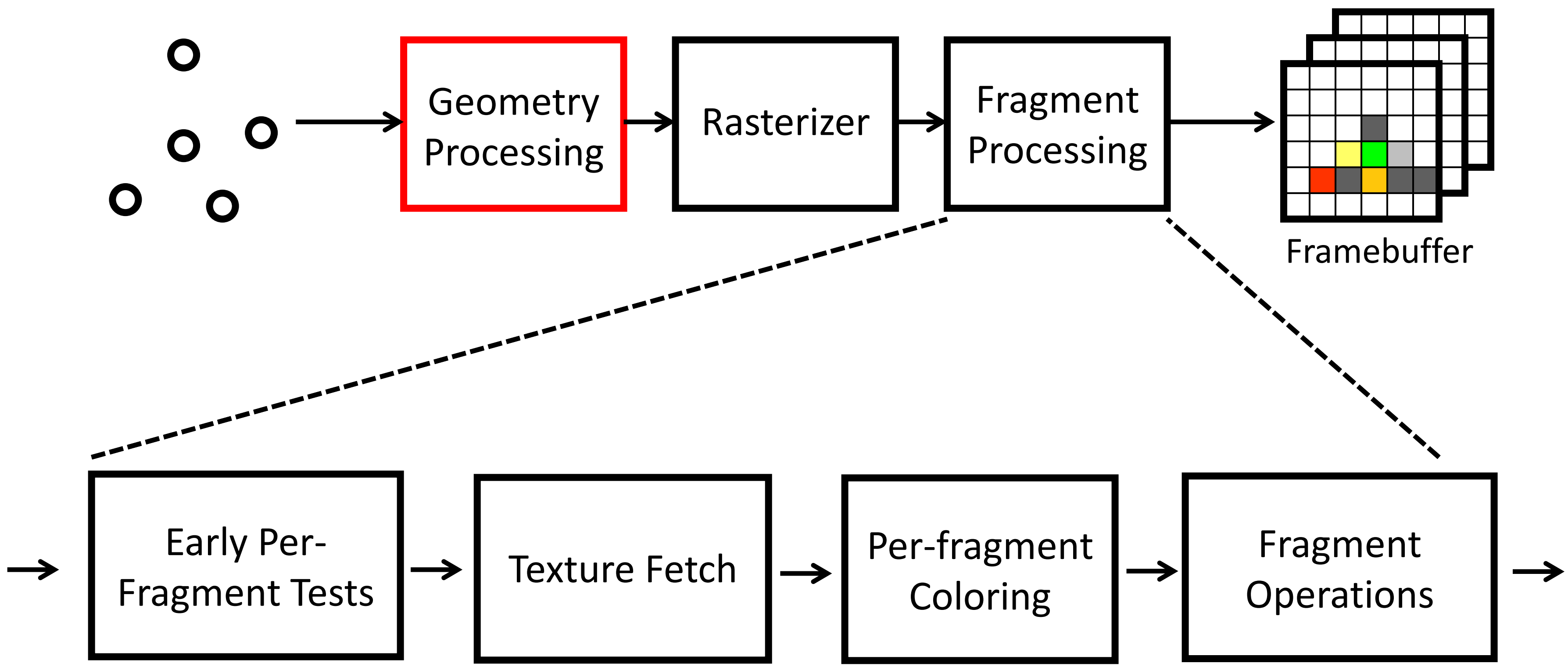


Rasterizer (C.P & P.P)



- Decomposition of primitives into fragments
- Interpolation of vertex colors
- Interpolation of texture coordinates
- Interpolation of vertex normals

Fragment Processing (C.P)



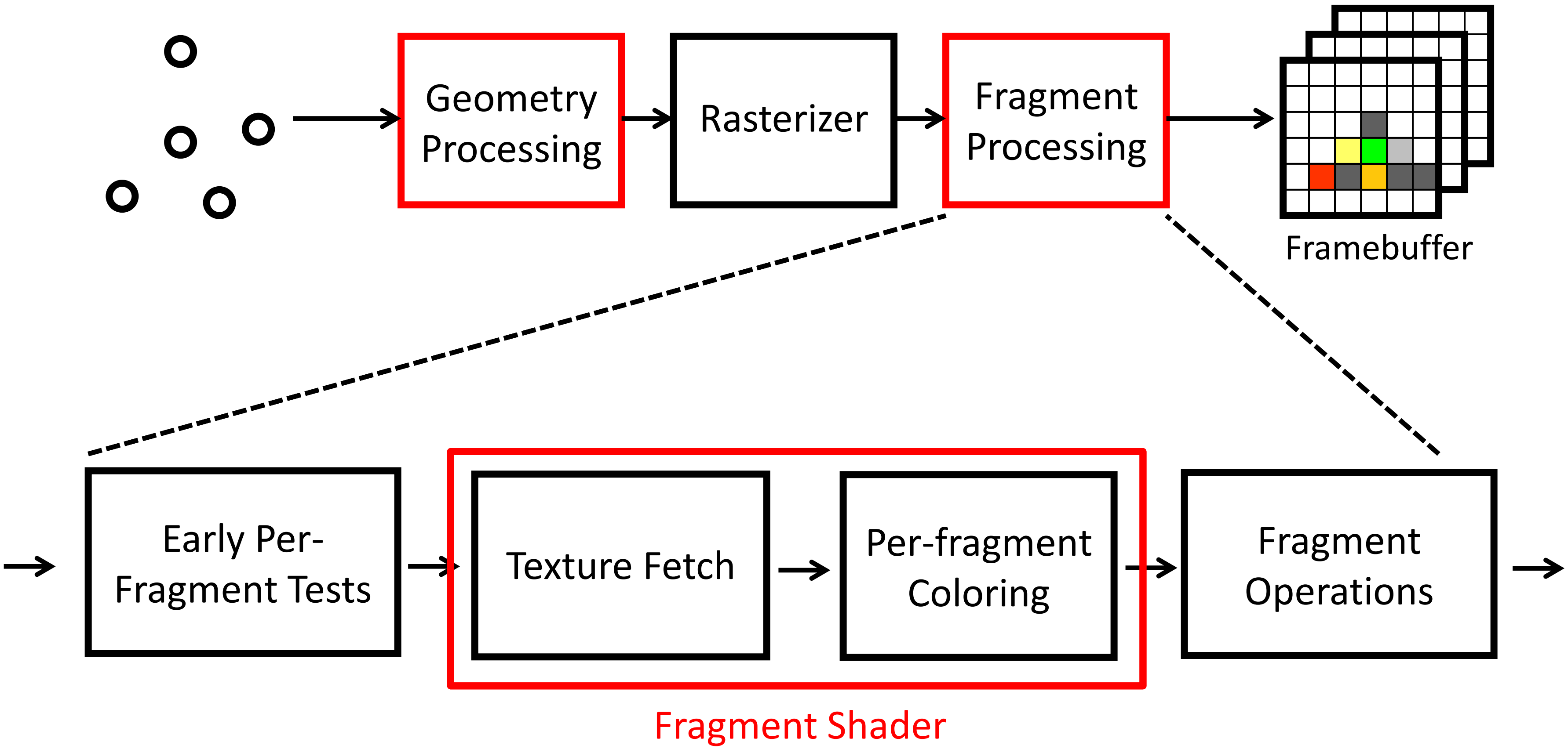
Early Per-Fragment Tests
- Pixel ownership test
- Scissor test
- Stencil test
- Depth buffer test
- Occlusion query counting

Texture Fetch
- Texture filtering
- Decompress

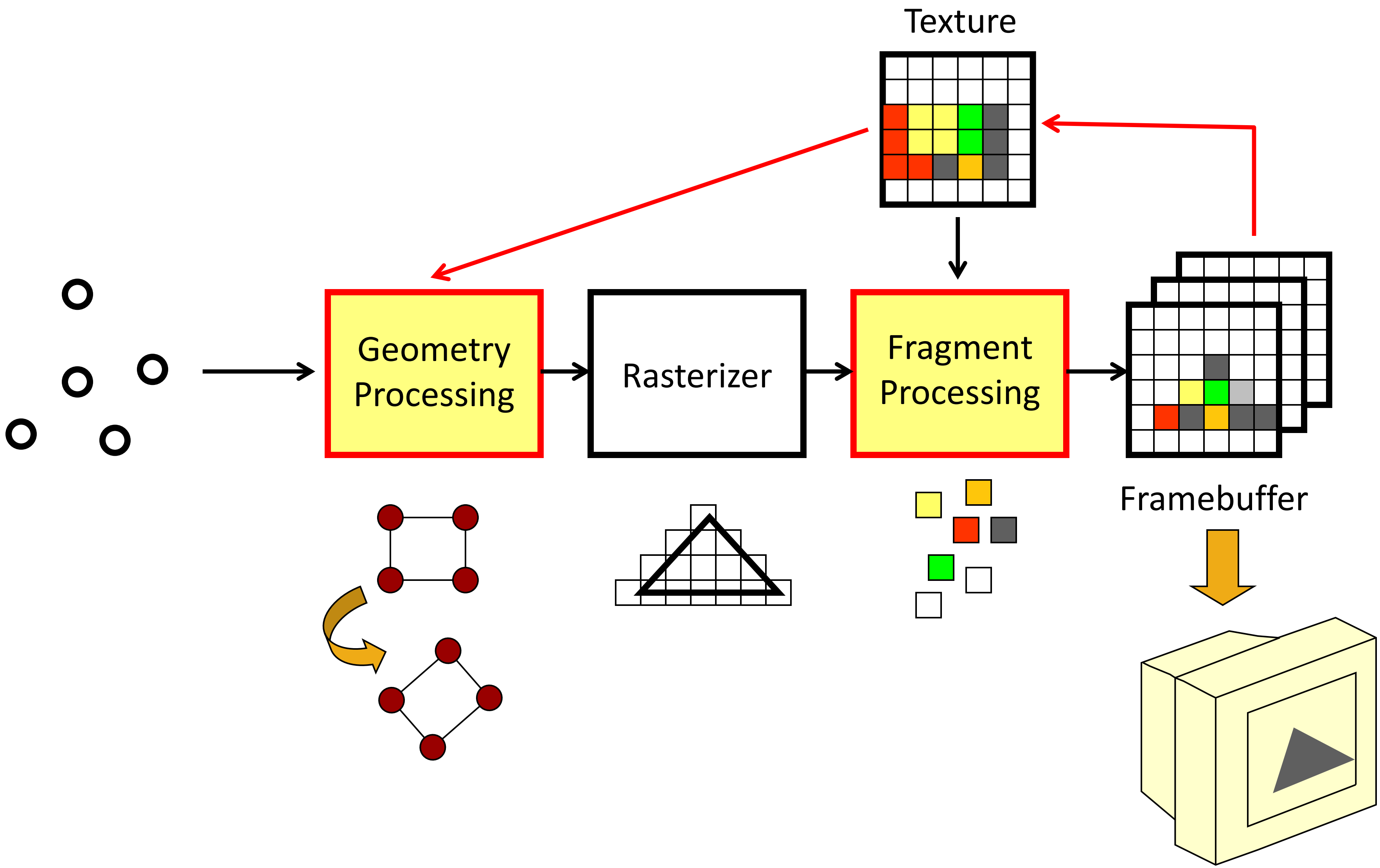
Per-fragment Coloring
- Applying texture to primary color

Fragment Operations
- Per-fragment test (if not done early)
- Alpha test
- Blending
- Dithering
- Logic Op

Fragment Processing (P.P)



Programmable Graphics Pipeline



Outline

- Conventional vs. Programmable graphics pipeline
- GLSL basics
- GLSL examples

Shading Languages

- Pixar's RenderMan
  - Offline
- ARB fragment / vertex program
- NVIDIA Cg (C for graphics)
- DirectX HLSL
- OpenGL Shading Language (GLSL)

GLSL

- OpenGL Shading Language
- Part of OpenGL 2.0 and up
- High level C-like language
- New data types
  - Matrices
  - Vectors
  - Samplers
- As of OpenGL 3.1, application must provide shaders

GLSL Versions

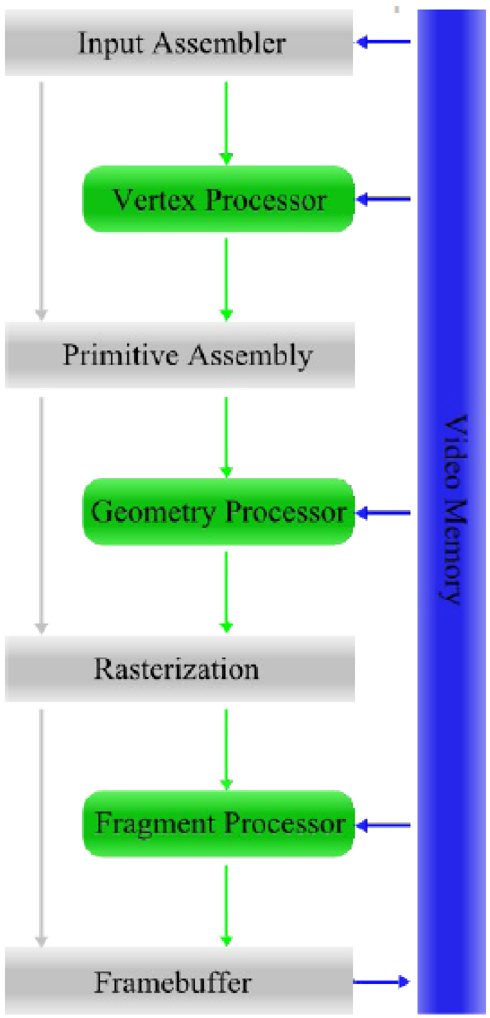
- Geometry shader becomes official in GLSL 1.5
- Older (deprecated) functions still available through compatibility extension

GLSL Version	OpenGL Version	Date	Shader Preprocessor
1.10.59 [1]	2.0	April 2004	#version 110
1.20.8[2]	2.1	September 2006	#version 120
1.30.10[3]	3.0	August 2008	#version 130
1.40.08[4]	3.1	March 2009	#version 140
1.50.11[5]	3.2	August 2009	#version 150
3.30.6[6]	3.3	February 2010	#version 330
4.00.9[7]	4.0	March 2010	#version 400
4.10.6[8]	4.1	July 2010	#version 410
4.20.11[9]	4.2	August 2011	#version 420
4.30.8[10]	4.3	August 2012	#version 430
4.40[11]	4.4	July 2013	#version 440
4.50[12]	4.5	August 2014	#version 450

Wikipedia

Programmable Stages

- GLSL 1.50 (OpenGL 3.2)
  - Vertex shaders
  - Geometry shaders
  - Fragment shaders
- All of these shaders are in reality executed on identical processing cores (unified shader) on current GPUs



Simple Vertex Shader

input from application

```
in vec4 vPosition;
void main(void)
{
     gl_Position = vPosition;
}
```

must link to variable in application

built in variable

Simple Fragment Shader

```
void main(void)
{
   gl_FragColor = vec4(1.0, 0.0,
 0.0, 1.0);
}
```

built in variable

Data Types

- C types: int, float, bool
- Vectors:
  - float vec2, vec3, vec4
  - Also int (ivec) and boolean (bvec)
- Matrices: mat2, mat3, mat4
  - Stored by columns
  - Standard referencing m[row][column]
- C++ style constructors
  - vec3 a = vec3(1.0, 2.0, 3.0)
  - vec2 b = vec2(a)

Pointers

- There are no pointers in GLSL
- We can use C structs which can be copied back from functions
- Because matrices and vectors are basic types they can be passed into and output from GLSL functions
  - mat3 func(mat3 a)

Storage Qualifiers

- `const, in, out, uniform`
- Variables can change
  - Once per primitive
  - Once per vertex
  - Once per fragment
  - At any time in the application
  - Cannot be changed after initialized
- Vertex attributes are interpolated by the rasterizer into fragment attributes

`Const` Qualifier

- Same as with C/C++
- Read only
- Cannot be modified in shaders
- Initialize when it is declared
  - `const float PI = 3.141592; // ok`
  - ```
    const float PI;
    PI = 3.141592;  // not ok
    ```
    ```

`Uniform` Qualifier

- Variables that are constant for an entire primitive
  - Think of this as read-only global variable
- Can be changed in application and sent to shaders
- Cannot be changed in shader
- Used to pass information to shader such as light location, transformation matrix, etc.

Set Uniform Variable

- glGetUniformLocation(program, *name)
  - Return index of the uniform variable name defined in shader program
- glUniform(location, value)
  - Set uniform variable at location with value

```
GLint timeLoc;
Glfloat timeValue;

timeLoc = glGetUniformLocation(program, "time");
glUniform1f(timeLoc, timeValue);
```

`In` Qualifier

- Input to a shader stage
- There are a few built-in variables such as gl_Position
- User defined vertex attribute variables
  - `in float temperature`
  - `in vec3 velocity`

Set Attribute Variables

- glGetAttribLocation(program, *name)
  - Return index of the attribute variable name defined in vertex shader program
- glEnableVertexAttribArray(location)
  - Enable attribute variable at location

Set Attribute Variables

- Vertex Shader

```
#version 140
in vec3 in_pos
void main(void)
{
   gl_Position = in_pos;
}
```

- Main code

```
// p : shader program
glBindBuffer(..)
vtxLoc = glGetAttribLocation(p, "in_pos");
glEnableVertexAttribArray(vtxLoc);
glVertexAttribPointer(vtxLoc, 3, GL_FLOAT, 0, 0, 0);
```

`Out` Qualifier

- Output from a shader stage
- Vertex -> Geometry -> Fragment shader
  - Vertex attributes are automatically interpolated by the rasterizer before fragment shader
- Use **out** in vertex shader and **in** in the fragment shader
  - `out vec4 color; // vertex shader`
  - `in vec4 color; // fragment shader`

KOREA UNIVERSITY

Operators and Functions

- Standard C functions
  - Trigonometric
  - Arithmetic
  - Normalize, reflect, length
- Overloading of vector and matrix types

```
mat4 a;
vec4 b, c, d;
c = b*a; // a column vector stored as a 1d array
d = a*b; // a row vector stored as a 1d array
```

Swizzling and Selection

- Can refer to array elements by element using [] or selection (.) operator with
  - x, y, z, w
  - r, g, b, a
  - s, t, p, q
  - `a[2], a.b, a.z, a.p` are the same
- **Swizzling** operator lets us manipulate components

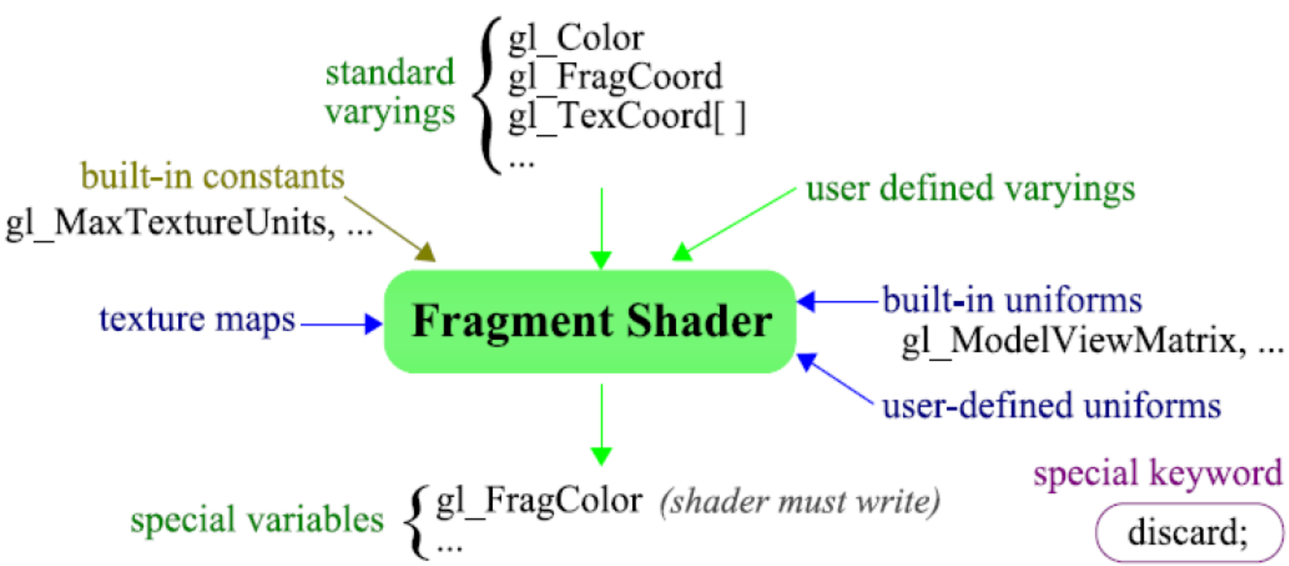
```
vec4 a;
a.yz = vec2(1.0, 2.0);
```

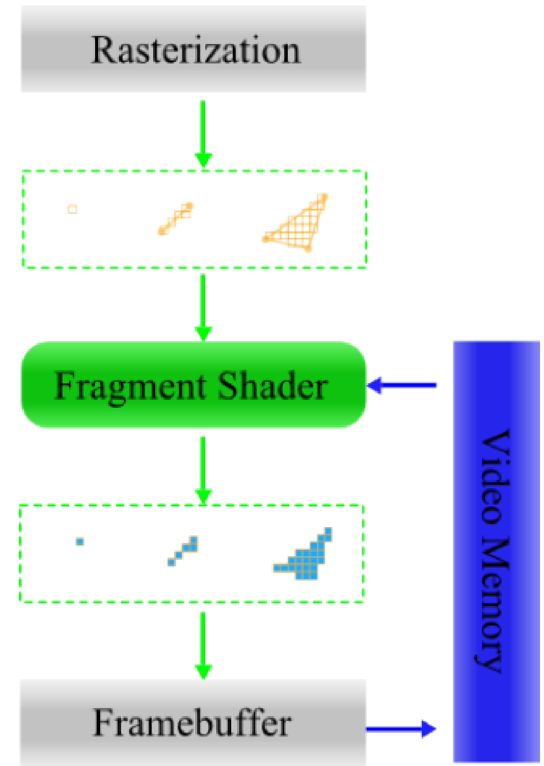
Vertex Shader

- Input
  - gl_Vertex, gl_normal, gl_Color, gl_TexCoord,...
  - User-defined vertex attributes (in)
  - User-defined uniform variables (uniform)
- Output
  - gl_Position, gl_FrontColor, gl_TexCoord[]...
  - User-defined varying variable (out)

Fragment Shader

- Process fragments
  - Write one or more output fragments
  - Use input color, texture coordinates, ...
  - Compute shading, sample textures, ...
  - Optionally discard fragment
  - MRT: multiple render targets





Setup GL Program for GLSL

```
int main(int argc, char **argv) {

    // init GLUT and create Window
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
    glutInitWindowPosition(100,100);
    glutInitWindowSize(320,320);
    glutCreateWindow("COSE436");

    // register callbacks
    glutDisplayFunc(renderScene);
    glutIdleFunc(renderScene);
    glutReshapeFunc(changeSize);

    glEnable(GL_DEPTH_TEST);
    glClearColor(1.0,1.0,1.0,1.0);

    glewInit();
    if (glewIsSupported("GL_VERSION_3_3"))
        printf("Ready for OpenGL 3.3\n");
    else {
        printf("OpenGL 3.3 is not supported\n");
        exit(1);
    }

    // create shader program
    setShaders();

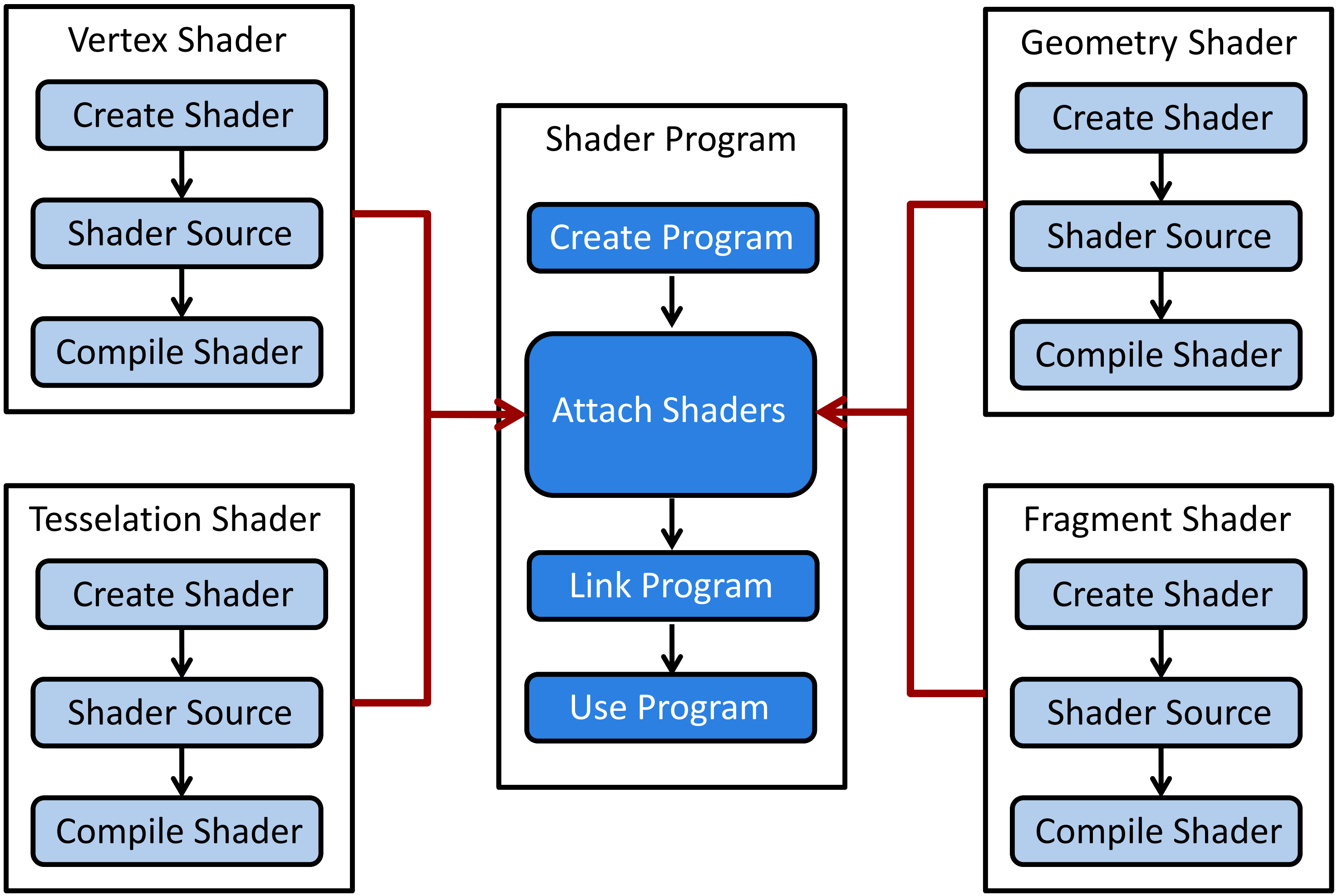
    // enter GLUT event processing cycle
    glutMainLoop();

    return 1;
}
```

Shader and Program

- A (shader) program is a collection of shaders
  - One program per render pass
  - Vertex, geometry, tessellation, fragment shaders
- A shader can be used in multiple programs
- Compiled shaders are linked in a program
- Your application can switch between different shader programs for different visual effect

Shaders and Program



Simple Shader

- Create shader based on the shader type
- Loading source file from text
- Compile shader

```
GLuint loadShader(GLenum shadertype, char *c)
{
    GLuint s = glCreateShader( shadertype );
    char *ss = textFileRead( c );
    const char *css = ss;
    glShaderSource(s, 1, &css, NULL);
    free( ss );
    glCompileShader( s );

    return s;
}
```

Creating a Shader

- `GLuint glCreateShader(GLenum shaderType)`
  - GL_VERTEX_SHADER
  - GL_GEOMETRY_SHADER
  - GL_TESS_CONTROL_SHADER
  - GL_TESS_EVALUATION_SHADER
  - GL_FRAGMENT_SHADER
- Return Value
  - the shader handler
- One main() per each shader

Loading Source Code

- ```
  void glShaderSource(GLuint shader,
                      int numOfStrings,
                      const char **strings,
                      int *lengthOfStrings)
  ```
  - shader: handle of the shader
  - numOfString: number of strings in the source code
  - strings: pointer to the array of source code
  - lengthOfStrings: array of the length of each string
- numOfStrings = 1, lengthOfStrings = NULL
  ```

Example

```
std::string v = "#version 150\n";
std::string c = ReadTextFile("common.glsl");
std::string s = ReadTextFile("mesh.vert");

GLchar const* files[] = { v.c_str(), c.c_str(), s.c_str() };
GLint lengths[]       = { v.size(),  c.size(),  s.size()  };

glShaderSource(id, 3, files, lengths);
```

Compile a Shader

- `void glCompileShader(GLuint shader)`
  - shader: handle to the shader
  - Compile shader code

Check Shader Status

- `void glGetShaderiv(`GLuint shader,
  GLenum pname,
  GLint * params)
  - GL_SHADER_TYPE
  - GL_DELETE_STATUS
  - GL_COMPILE_STATUS
  - GL_INFO_LOG_LENGTH
  - GL_SHADER_SOURCE_LENGTH

Validate a Shader

- `void glGetShaderInfoLog()`
  - Check log if shader is successfully compiled
  - Get the size of the log using `glGetShaderiv`

```
// validating shader after compilation
int status, sizeLog;
glGetShaderiv(shader, GL_COMPILE_STATUS, &status);
if(status == GL_FALSE)
{
    glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &sizeLog);
    char *log = new char[sizeLog + 1];
    glGetShaderInfoLog(shader, sizeLog, NULL, log);
    cout << "Compilation error: " << log << endl;
    delete [] log;
}
```

Simple Program

- Attach and link compiled shaders

```
void setShaders() {

    v = loadShader( GL_VERTEX_SHADER, "../ex.vert" );
    g = loadShader( GL_GEOMETRY_SHADER_EXT, "../ex.geom" );
    f = loadShader( GL_FRAGMENT_SHADER, "../ex.frag" );

    p = glCreateProgram();
    glAttachShader(p,f);
    glAttachShader(p,v);
    glAttachShader(p,g);

    glProgramParameteri(p, GL_GEOMETRY_INPUT_TYPE, GL_TRIANGLES);
    glProgramParameteri(p, GL_GEOMETRY_OUTPUT_TYPE, GL_TRIANGLE_STRIP);

    int temp;
    glGetIntegerv(GL_MAX_GEOMETRY_OUTPUT_VERTICES_EXT, &temp);
    glProgramParameteri(p,GL_GEOMETRY_VERTICES_OUT_EXT, temp);

    glLinkProgram(p);
    glUseProgram(p);
}
```

Create a Program

- glCreateProgram()
- glAttachShader()
- glLinkProgram()

Check Program Status

- `void glGetProgramiv(`GLuint shader,
  GLenum pname,
  GLint * params)
  - GL_DELETE_STATUS
  - GL_LINK_STATUS
  - GL_VALIDATE_STATUS
  - GL_INFO_LOG_LENGTH
  - …

Check Link Error

- Check if all attached shaders can be linked

```
// validating program after linking
GLint status;
glGetProgramiv (program, GL_LINK_STATUS, &status);
if (status == GL_FALSE)
{
   GLint sizeLog;
   glGetProgramiv(program, GL_INFO_LOG_LENGTH, &sizeLog);
   char *log= new char[sizeLog + 1];
   glGetProgramInfoLog(program, sizeLog, NULL, log);
   cout << "Linker error: " << log << endl;
   delete[] log;
}
```

Validate a Program

- `glValidateProgram()`
  - checks if program can execute given the current OpenGL state

```
// validating program after linking
glValidateProgram(program);
GLint status;
glGetProgramiv(program, GL_VALIDATE_STATUS, &status);
if (status == GL_FALSE)
{
   cerr << "Error validating program "
        << program << endl;
}
```

Outline

- Conventional vs. Programmable graphics pipeline
- GLSL basics
- GLSL examples

Simple Coloring Faces

- Set color in OpenGL application
  - Global GL attribute

```
// set color
glColor4f(1.0, 0.0, 0.0, 1.0);
...
Draw something
...
```

Shaders

```
// vertex shader
void main(void)
{
  gl_Position = gl_ModelViewProjectionMatrix*gl_Vertex;
  gl_FrontColor = gl_Color;
  gl_BackColor = vec4(0,0,0,0); // black for backface
}
```

```
// fragment shader
void main(void)
{
  gl_FragColor = gl_Color;
}
```

Diffuse Shading

```
// vertex shader
out vec3 normal;
void main(void)
{
  gl_Position = gl_ModelViewProjectionMatrix * (gl_Vertex);
  normal = normalize(gl_NormalMatrix * gl_Normal).xyz;
  gl_FrontColor = vec4(1,0,0,1); // Red
}
```

```
// fragment shader
in vec3 normal;
void main(void)
{
  vec3 n = normalize(normal);
  vec3 l = normalize(gl_LightSource[0].position.xyz);
  gl_FragColor = max(dot(l,n),0)*gl_Color;
}
```

KOREA UNIVERSITY

Scaling

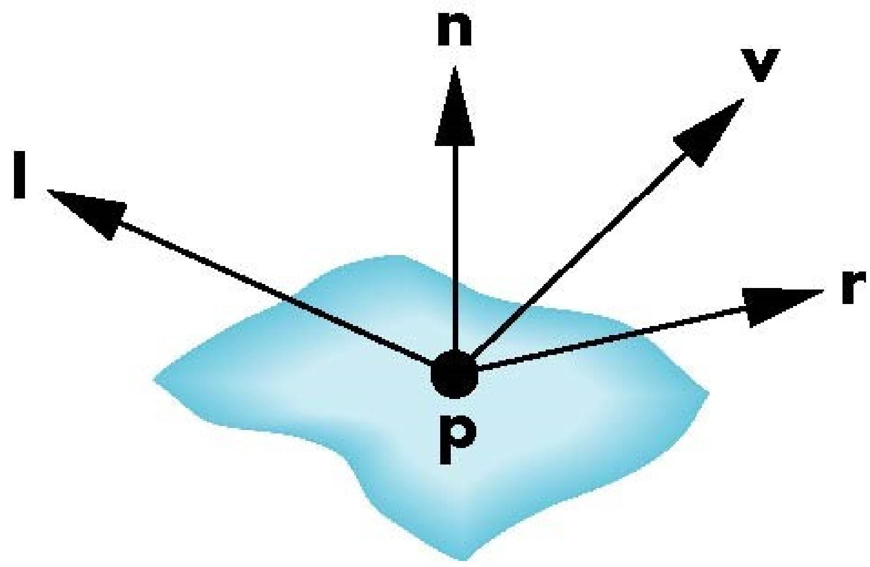
```
// main.cpp
Gluint scaleLocation = glGetUniformLocation(p_curr, "m_Scale");
if(scaleLocation != -1) glUniform1f(scaleLocation, currScale);
```

```
// vertex shader
uniform float m_Scale;
out vec3 normal;
void main(void)
{
   vec4 P_obj = gl_Vertex;
   P_obj.x = P_obj.x * m_Scale;
   P_obj.y = P_obj.y * m_Scale;
   P_obj.z = P_obj.z * m_Scale;

   // Clip Coordinate
   gl_Position = gl_ModelViewProjectionMatrix * (P_obj);
   normal = normalize(gl_NormalMatrix * gl_Normal).xyz;
   gl_FrontColor = vec4(1,0,0,1); // Red
}
```

Phong Reflection Model

- Three components
  - Diffuse
  - Specular
  - Ambient
- Uses four vectors
  - To source
  - To viewer
  - Normal
  - Perfect reflector

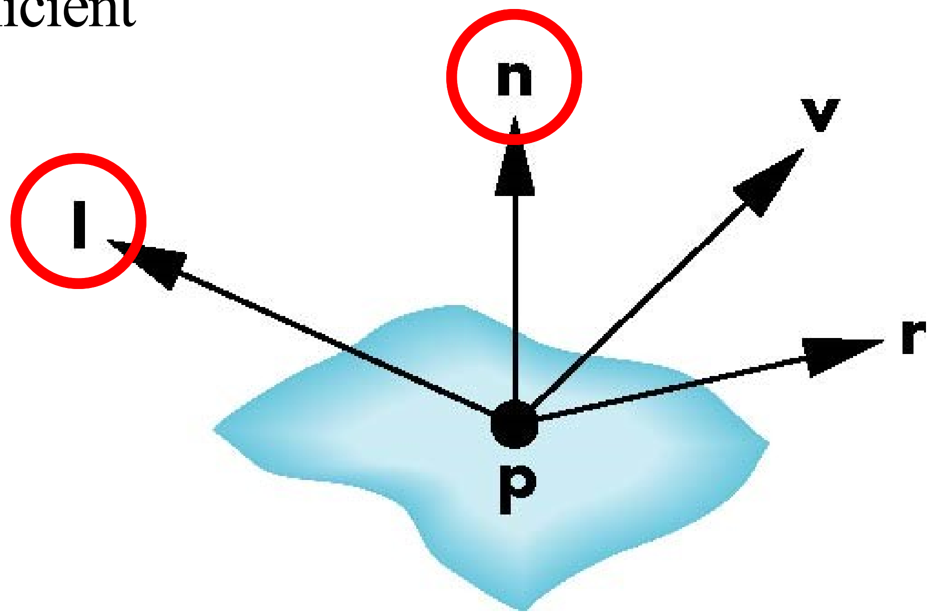


Diffuse Reflection

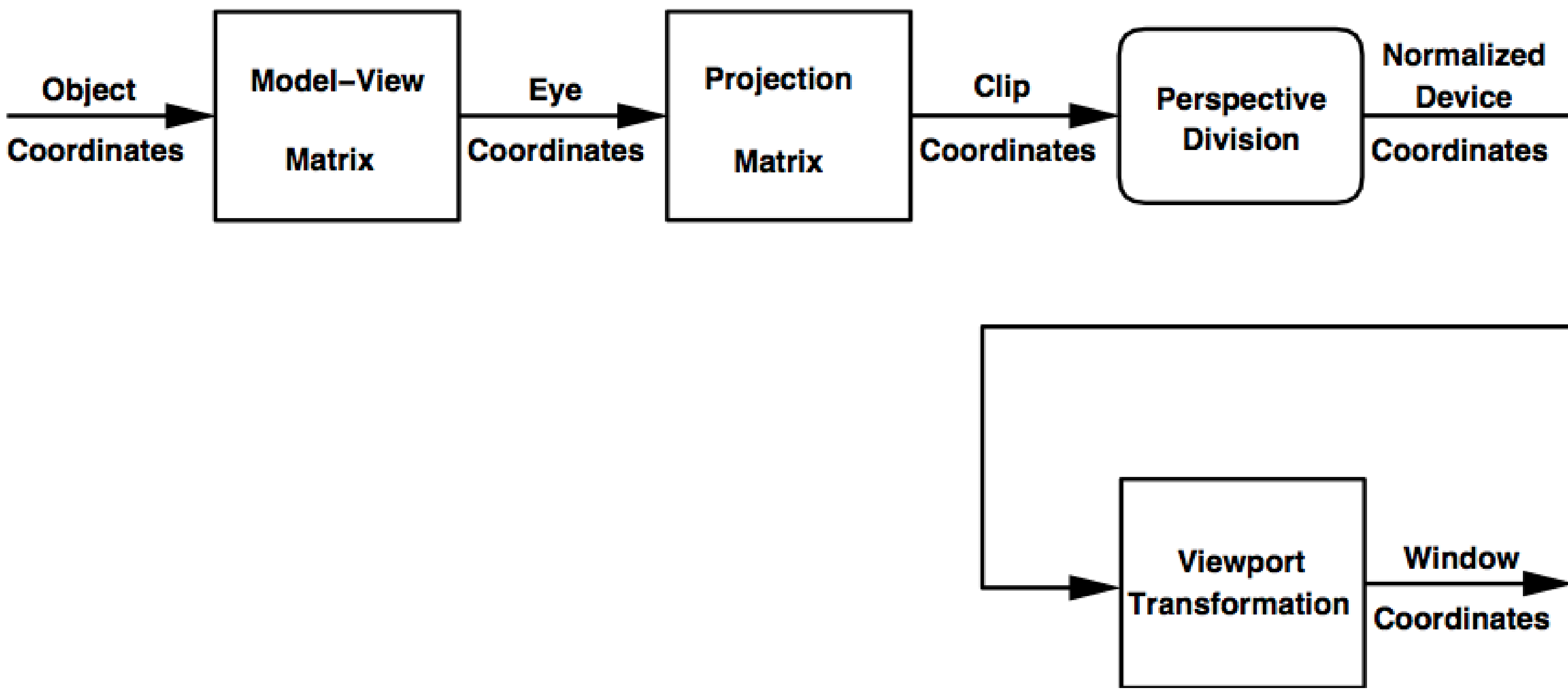
$$\cos\theta = \mathbf{l} \cdot \mathbf{n}$$

$$\boxed{\mathbf{I}_d = \mathbf{k}_d(\mathbf{l}\cdot\mathbf{n})\mathbf{L}_d}$$

$0 \le \mathbf{k}_d \le 1$: absorption coefficient



Coordinate Transformations



Eye Coordinate

- Global coordinate relative to the viewer
  - Multiplied by `ModelView` matrix
- Eye position is (0,0,0)
- Light position is in eye coordinate by default
- *l* = light pos
  - directional light
- Normal vector must be transformed to eye coordinate
  - Modelview transform is not sufficient

Per-fragment Diffuse Shader

Vertex Shader (note that vec,mat should be vec3/4 or mat3/4 in real code)

```
attribute vec vPosition;
attribute vec vNormal;
uniform mat modelViewMatrix;
uniform mat projectionMatrix;
uniform mat normalMatrix;
out vec normal;
void main(void)
{
  gl_Position = projectionMatrix*modelViewMatrix*vPosition;
  normal = normalMatrix*vNormal;
}
```

Per-fragment Diffuse Shader

Fragment Shader

```
uniform vec lightpos;
uniform vec Kd; // Material property
uniform vec Ld; // Light property
in vec normal;
void main()
{
  vec n = normalize(normal);
  Vec l = normalize(lightpos); // distance light
  vec diffuse = max(dot(l,n),0)* Kd * Ld;
  gl_FragColor = diffuse;
}
```

Normal Matrix

- Compensate anisotropic scaling
  - Modelview matrix

G: unknown here

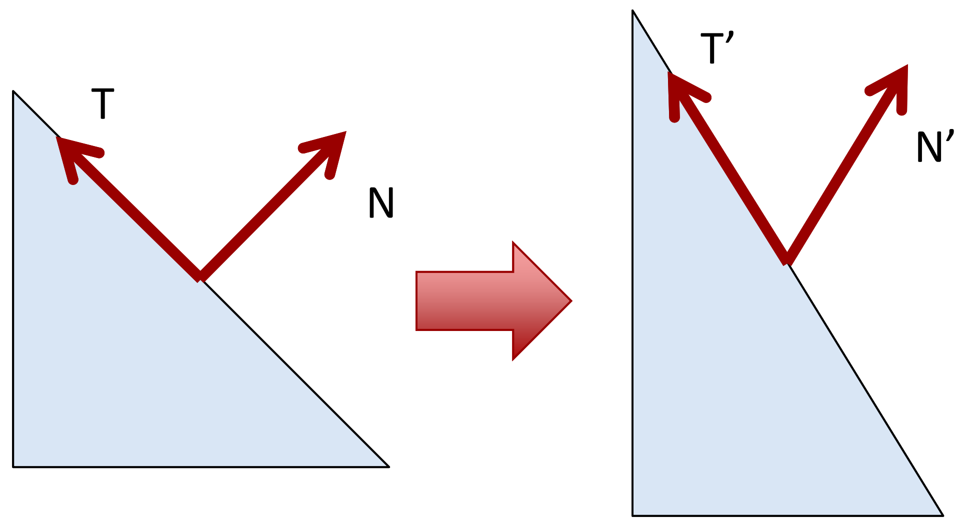
$$N' \cdot T' = (GN) \cdot (MT) = 0$$

$$(GN) \cdot (MT) = (GN)^T (MT)$$

$$(GN)^T (MT) = N^T G^T MT$$

$$G^T M = I$$

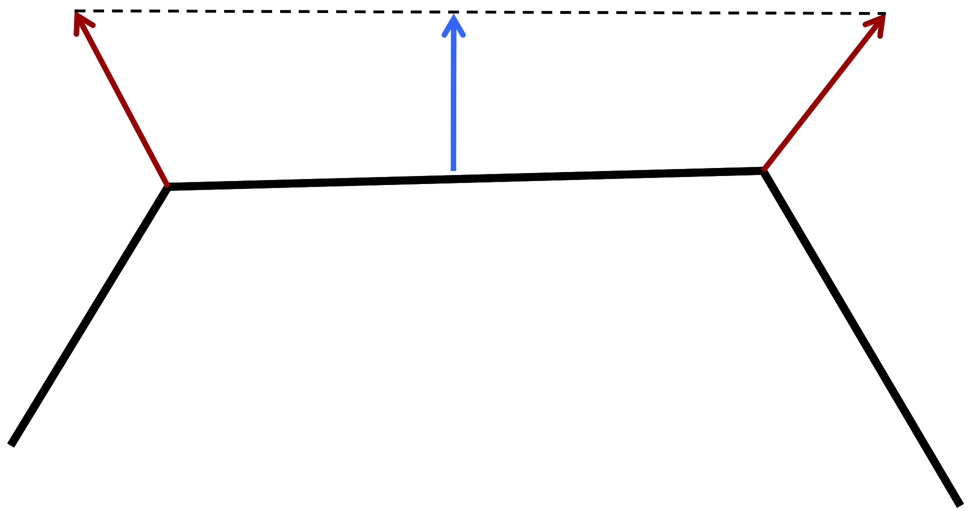
$$G = (M^{-1})^T$$



Normal Matrix

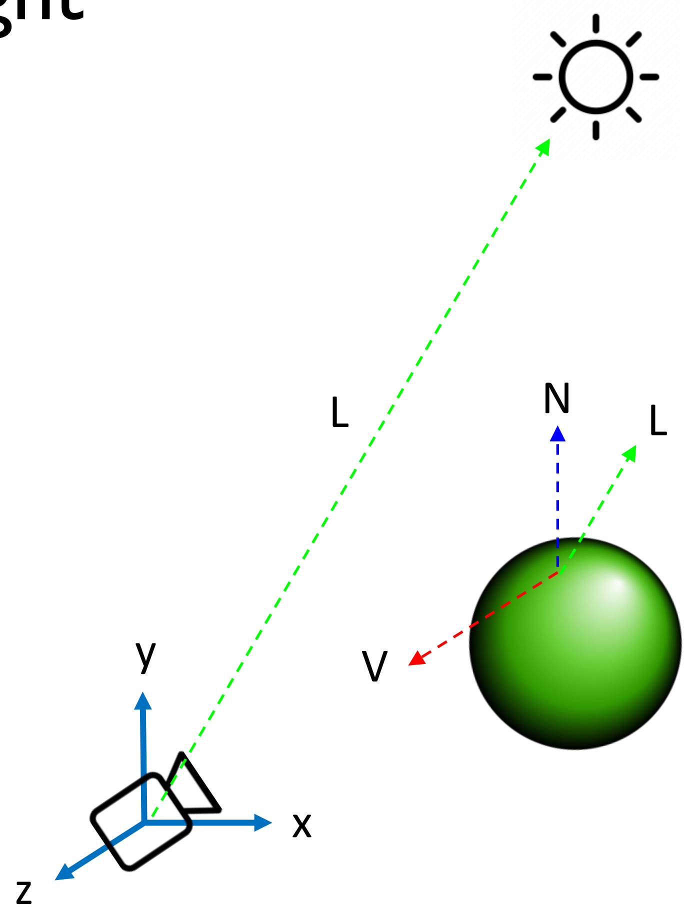
Normalize Normal Vector

- Linear interpolation does not preserve length



Light Direction

- Light position defined in eye coordinate
- Distant light



Ambient Reflection

- Same at every point on the surface
- Ambient reflection term

$$\boxed{\mathbf{I}_a = \mathbf{k}_a \mathbf{L}_a} \qquad 0 \leq k_a \leq 1$$

  – k: Amount reflected
    - Some is absorbed and some is reflected

- Ambient reflection term in rendering equation
  – <u>Individual</u> light sources, a <u>global</u> ambient term

Adding Ambient Term

Vertex Shader

```
out vec normal;
void main(void)
{
  gl_Position = projectionMatrix*modelViewMatrix*vPosition;
  normal = normalMatrix*vNormal;
}
```

Fragment Shader

```
uniform vec Kd, Ka;
uniform vec Ld, La;
in vec normal;
void main()
{
  vec n = normalize(normal);
  Vec l = normalize(lightpos); // distance light
  vec ambient = Ka * La;
  vec diffuse = max(dot(l,n),0)* Kd * Ld;
  gl_FragColor = diffuse + ambient;
}
```

Specular Reflection

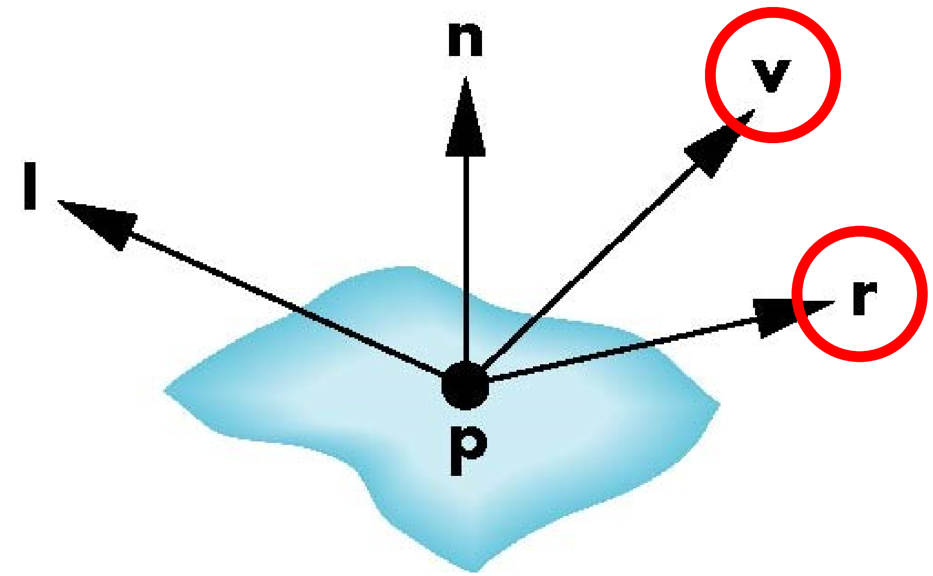
- Reflection vector

$$\mathbf{r} = 2(\mathbf{l} \square \mathbf{n})\mathbf{n} - \mathbf{l}$$

- Angle between ***r*** and ***v***

$$\mathbf{I}_s = \mathbf{k}_s(\mathbf{r} \cdot \mathbf{v})^{\alpha} \mathbf{L}_s$$

  - α → infinite : mirror
  - 100 < α < 200 : metal
  - 5< α < 10 : plastic

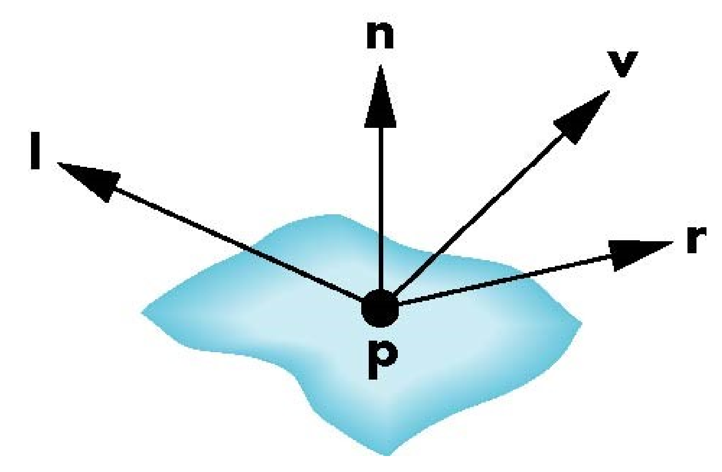


KOREA UNIVERSITY

Adding Specular Term

Vertex Shader

```
out vec p;
void main(void) {
  ...
  p = modelViewMatrix * vPosition;
}                          // eye coordinate
```




```
uniform vec Kd, Ka, Ks;
uniform vec Ld, La, Ls;
uniform float alpha; // shininess parameter
in vec normal;
in vec p;
void main() {
  vec v = normalize(-p);
  vec n = normalize(normal);
  vec l = normalize(lightpos - p); // point light
  vec r = normalize(reflect(-l,n));
  vec ambient = Ka * La;
  vec diffuse = max(dot(l,n),0) * Kd * Ld;
  vec specular = Ks * pow(max(dot(r,v),alpha) * Ls;
  gl_FragColor = diffuse + ambient + specular;
}
```

Fragment Shader

Attenuation

- The light from a point source that reaches a surface is inversely proportional to the square of the distance between them

- *d*: distance to the light source
  - a, b, c : attenuation factor

$$\mathbf{I} = \mathbf{k}_a \mathbf{L}_a + \frac{1}{a + bd + cd^2}\left(\mathbf{k}_d \left(\mathbf{l} \cdot \mathbf{n}\right)\mathbf{L}_d + \mathbf{k}_s \left(\mathbf{r} \cdot \mathbf{v}\right)^{\alpha} \mathbf{L}_s\right)$$

Computing Attenuation

- Distance to vertex is computed in vertex shader
- Distance to fragment is computed by rasterization
- Per-fragment attenuation is computed in fragment shader

Vertex Shader

```
uniform vec lightpos;
out vec normal;
out vec p;
out float dist;
void main(void)
{
  gl_Position = projectionMatrix*modelViewMatrix*vPosition;
  normal = normalMatrix*vNormal;
  p = modelViewMatrix * vPosition;
  dist = vec(lightpos - p);
  dist = length(dist);
}
```

Fragment Shader

```
uniform vec attenuation;
in vec normal;
in vec p;
in float dist;
void main() {
  vec3 v = normalize(-p);
  vec3 n = normalize(normal);
  vec3 l = normalize(lightpos - p);
  vec3 r = normalize(reflect(-l,n));
  float att = 1.0 / (attenuation.x +
                     attenuation.y * dist +
                     attenuation.z * dist * dist);
  vec ambient = ...
  vec diffuse = ...
  vec specular = ...
  gl_FragColor = ambient + att * (diffuse + specular);
}
```

Questions?

Per vertex lighting

Per fragment lighting