北京航空航天大學
人工智能研究院

Computer Graphics
Lecture 8: Graphics Programming (OpenGL Shadings Language, GLSL)

潘成伟 (Chengwei Pan)

Email: pancw@buaa.edu.cn

Office: Room B1021, New Main Building

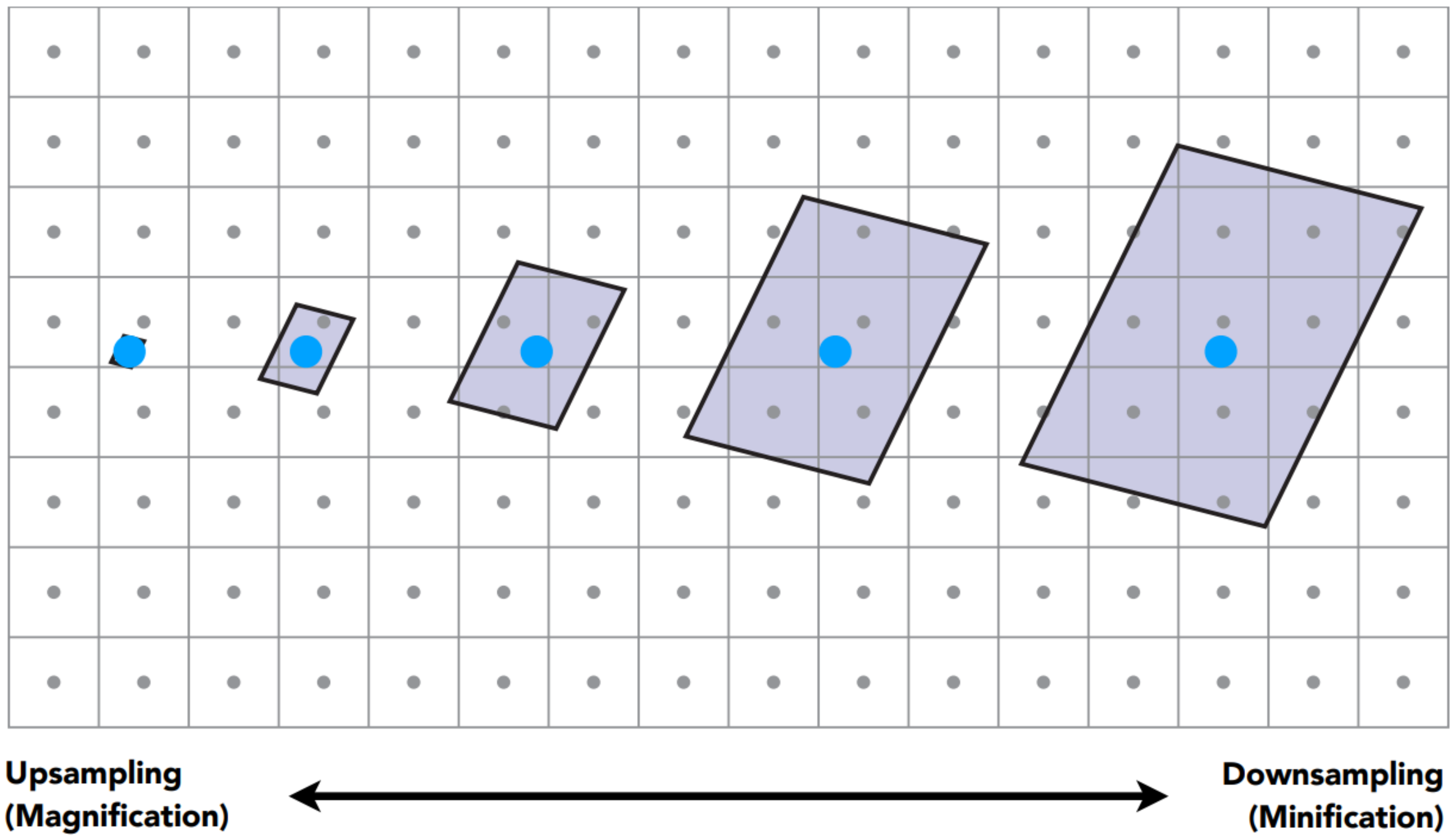
北京航空航天大学，人工智能研究院

Institute of Artificial Intelligence, Beihang University

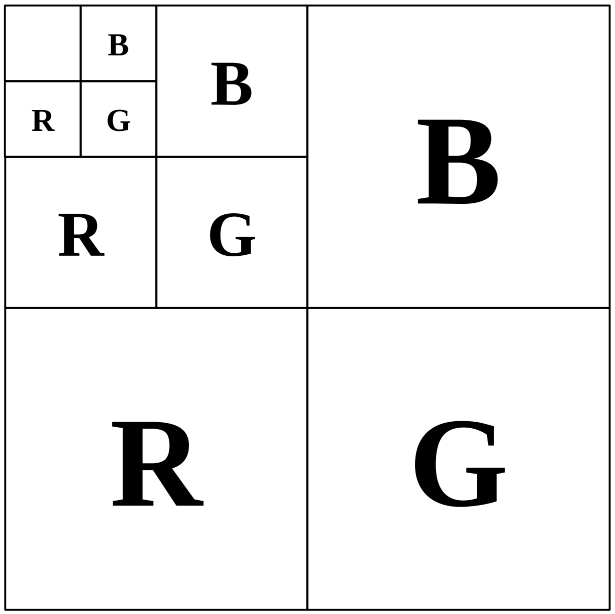
Last Lecture

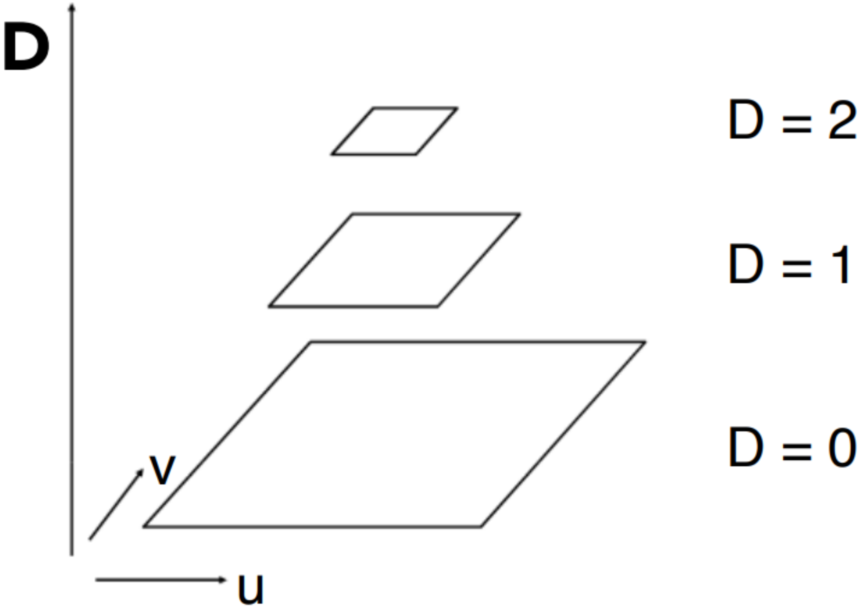
- Texture & Texture mapping
- Texture mapping stages
  - Parameterization
  - Mapping
  - Filtering
- Texture mapping applications
  - Modulation textures
  - Illumination mapping
  - Bump mapping
  - Environment mapping
  - …

Screen Pixel Footprint in Texture



Mipmap (L. Williams 83)



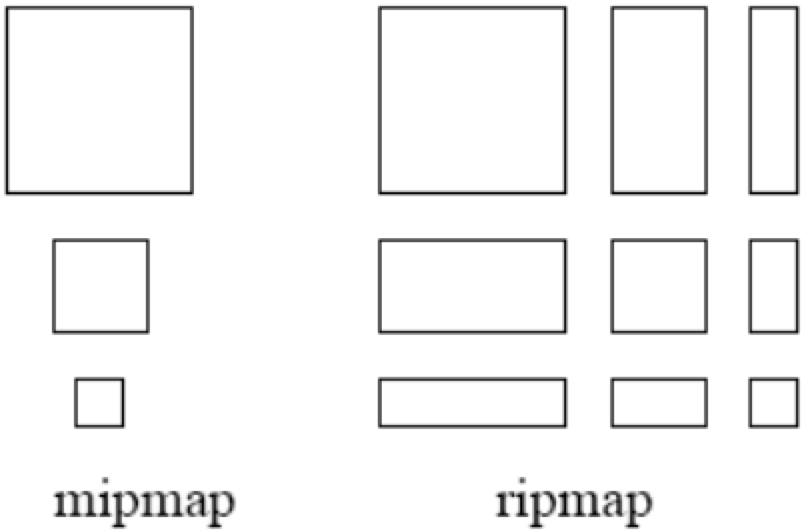


"Mip hierarchy"
level = D

- What is the storage overhead of a mipmap ?

Anisotropic Filtering

- Ripmaps and summed area tables
  - Can look up axis-aligned rectangular zones
  - Diagonal footprints still a problem

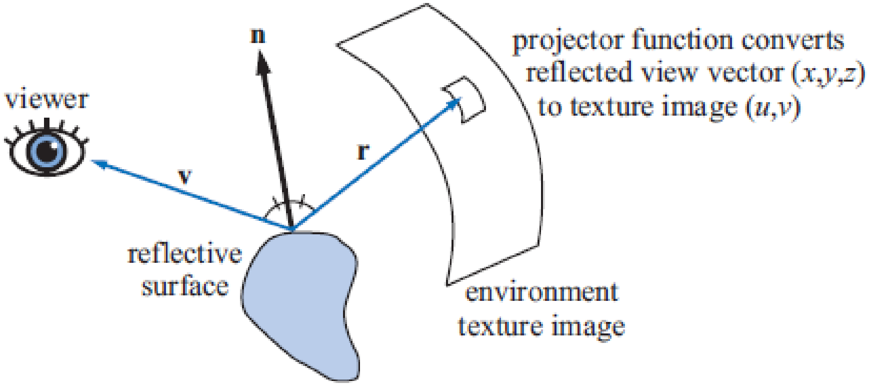


Environment Map

Light from the environment

[Blinn & Newell 1976]

Rendering with the environment



Spherical Environment Map

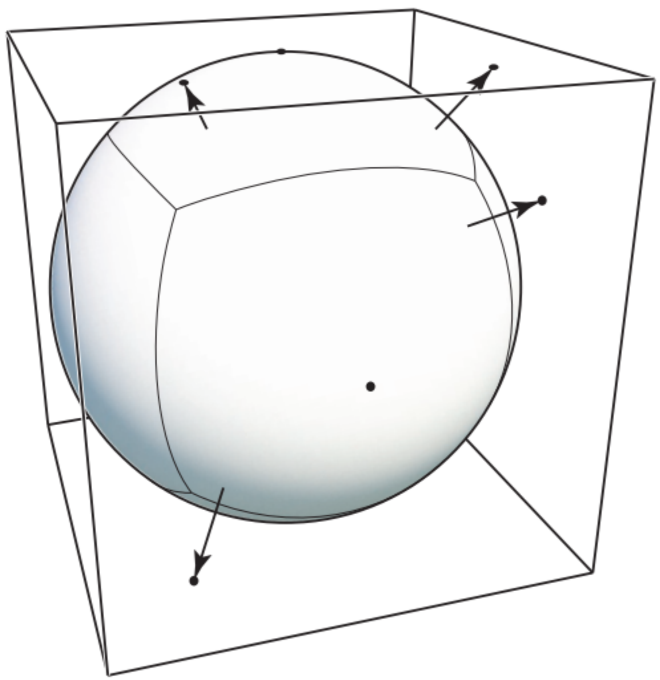
Hand with Reflecting Sphere. M. C. Escher, 1935. lithograph

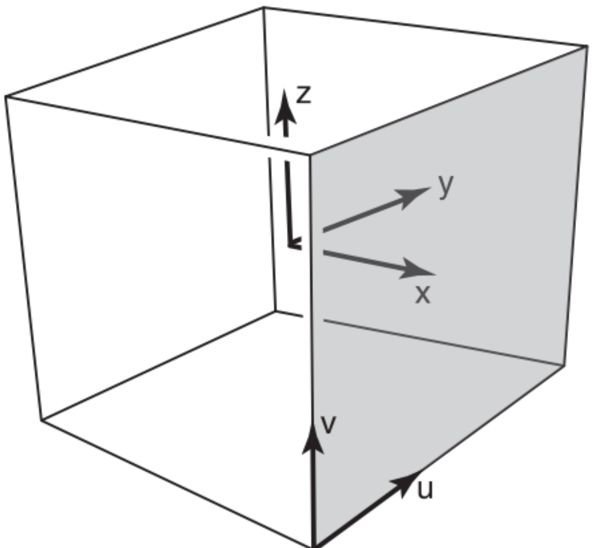
Light Probes, Paul Debevec

Spherical Map — Problem

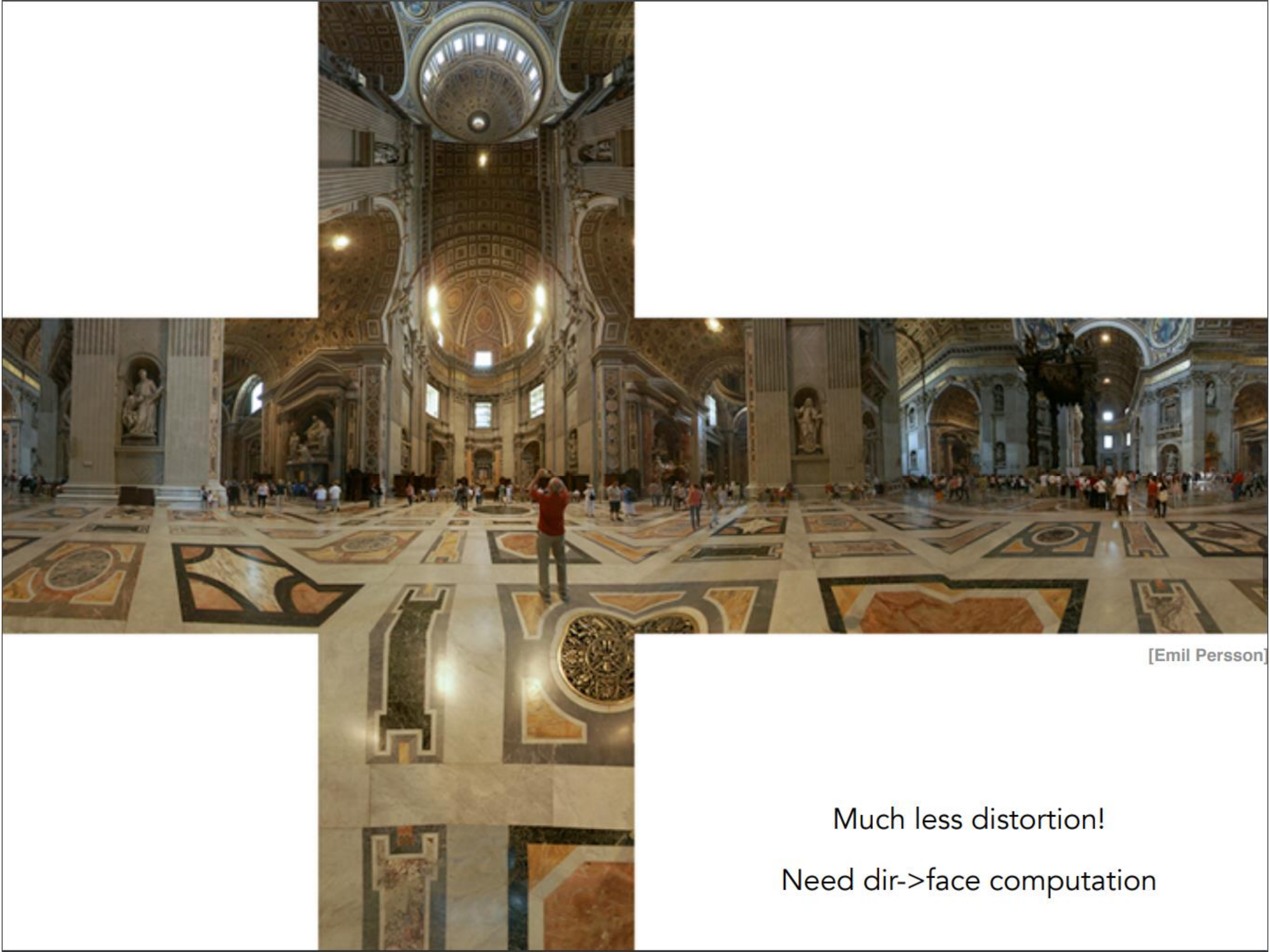
Prone to distortion (top and bottom parts)!

Cube Map





A vector maps to cube point along that direction.
The cube is textured with 6 square texture maps.



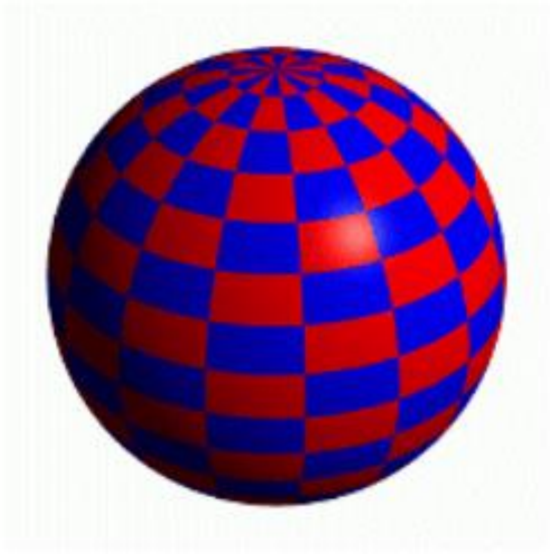
[Emil Persson]

Much less distortion!

Need dir->face computation

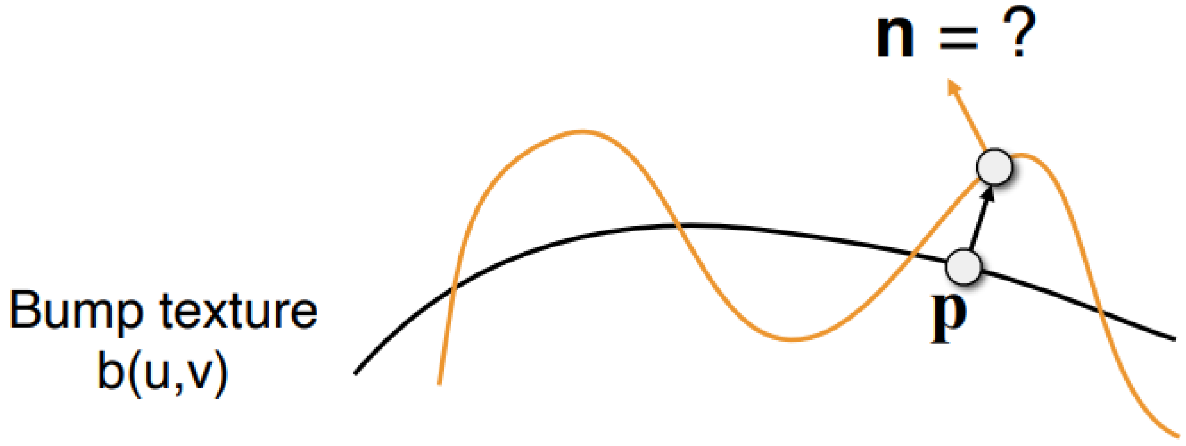
Textures can affect shading!

- Textures doesn’t have to only represent colors
  - What if it stores the height / normal?
  - Bump / normal mapping
  - Fake the detailed geometry



Bump Mapping

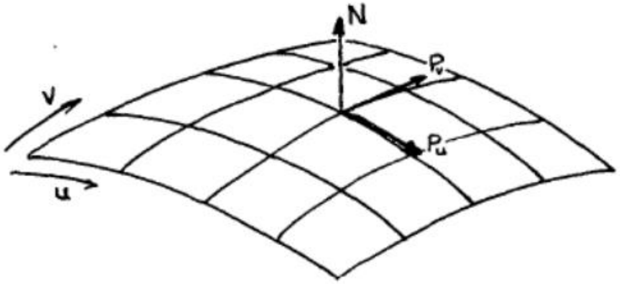
- Adding surface detail without adding more triangles
  - Perturb surface normal per pixel
    (for shading computations only)
  - “Height shift” per texel defined by a texture
  - How to modify normal vector?



Bump mapping

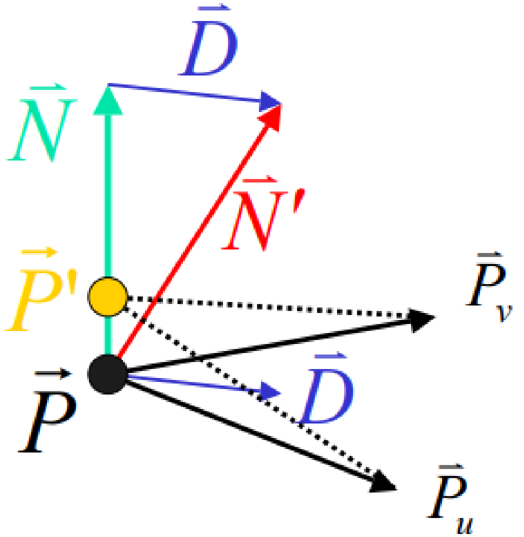
- Calculate finite differences from the bump map

$$B_u = \frac{dB}{du}, B_v = \frac{dB}{dv}$$



- Calculate a perturbed normal vector

$$\boldsymbol{n}' = \boldsymbol{n} + \frac{B_u(\boldsymbol{n} \times \boldsymbol{P_v}) - B_v(\boldsymbol{n} \times \boldsymbol{P_u})}{\|\boldsymbol{n}\|}$$

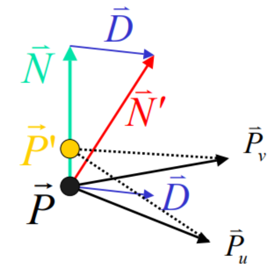


Bump mapping

$$\vec{P} = [x(u,v), y(u,v), z(u,v)]^T$$ Initial point

$$\vec{N} = \vec{P}_u \times \vec{P}_v$$ Normal

$$\vec{P}' = \vec{P} + B(u,v)\vec{N}$$ Simulated elevated point after bump



$$\vec{N}' \approx \vec{N} + \underbrace{B_u\vec{P}_u + B_v\vec{P}_v}_{\vec{D}}$$

Variation of normal in u direction

$$B_u = \frac{B(s-\Delta,t) - B(s+\Delta,t)}{2\Delta}$$

$$B_v = \frac{B(s,t-\Delta) - B(s,t+\Delta)}{2\Delta}$$

Variation of normal in v direction

$$\vec{P}' = \vec{P} + \frac{B(u,v)\vec{N}}{\|\vec{N}\|}$$

Assume B is very small...

$$\vec{P}'_u = \vec{P}_u + \frac{B_u\vec{N}}{\|\vec{N}\|} + \cancel{\frac{B\vec{N}_u}{\|\vec{N}\|}}^{\approx 0}$$

$$\vec{P}'_v = \vec{P}_v + \frac{B_v\vec{N}}{\|\vec{N}\|} + \cancel{\frac{B\vec{N}_v}{\|\vec{N}\|}}^{\approx 0}$$

$$\vec{N}' = \vec{P}'_u \times \vec{P}'_v$$

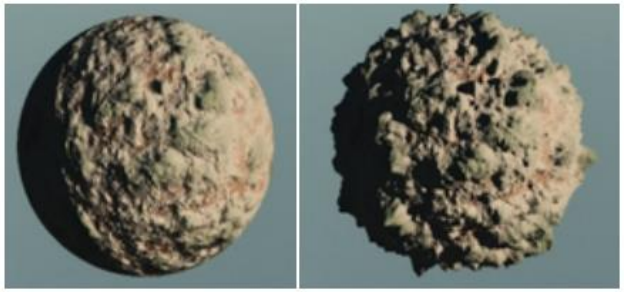
$$\vec{N}' \approx \vec{P}_u \times \vec{P}_v + \frac{B_u(\vec{N} \times \vec{P}_v)}{\|\vec{N}\|} + \frac{B_v(\vec{P}_u \times \vec{N})}{\|\vec{N}\|} + \frac{B_u B_v(\vec{N} \times \vec{N})}{\|\vec{N}\|^2}$$

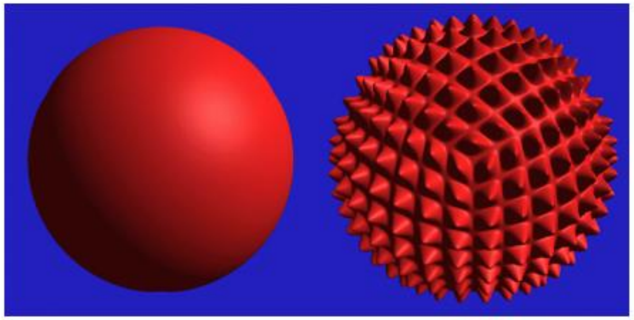
But $\vec{P}_u \times \vec{P}_v = \vec{N}$, $\vec{P}_u \times \mathrm{\vec{N}} = -\vec{N} \times \vec{P}_u$ and $\vec{N} \times \vec{N} = 0$ so

$$\vec{N}' \approx \vec{N} + \frac{B_u(\vec{N} \times \vec{P}_v)}{\|\vec{N}\|} - \frac{B_v(\vec{N} \times \vec{P}_u)}{\|\vec{N}\|}$$

Displacement mapping

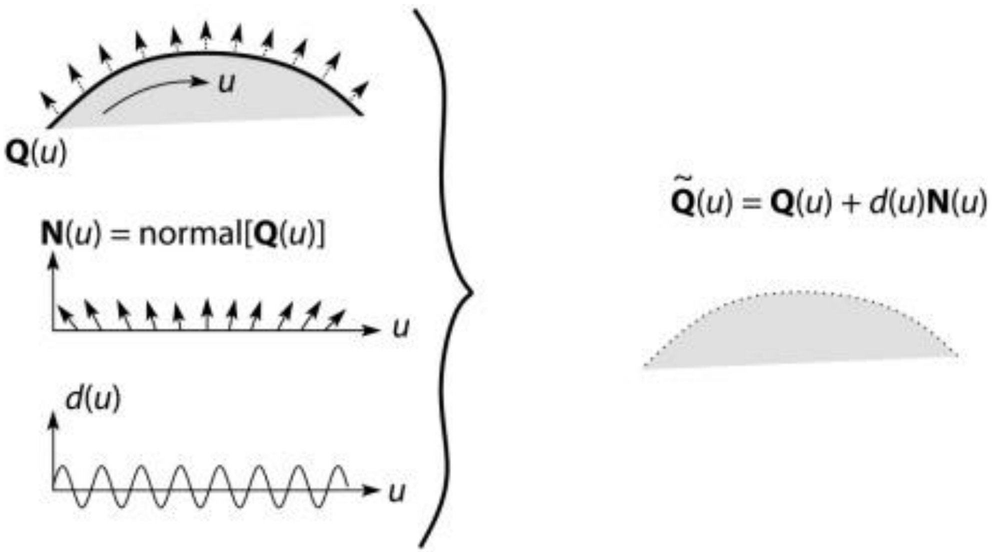
- Bump mapping only changes shading of the surface, not the actual geometry
- Displacement mapping alters the surface using the texture as a mesh defined in uv coordinates

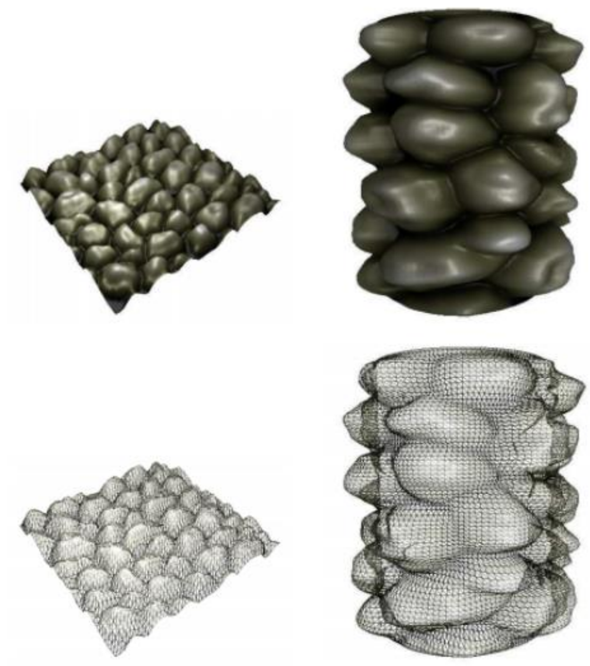




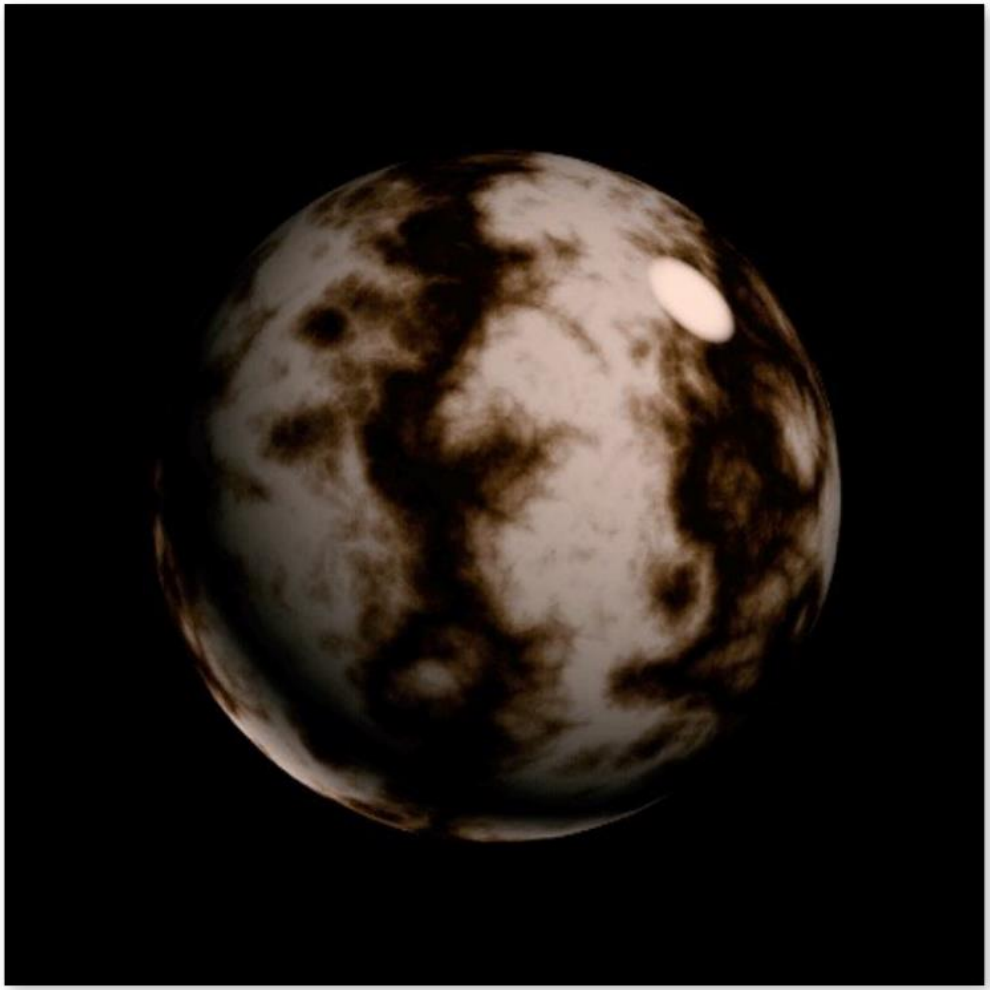
Displacement Mapping

- Subdivide the surface to resolution of texture
- Displace vertices in normal direction of surface by height in displacement map



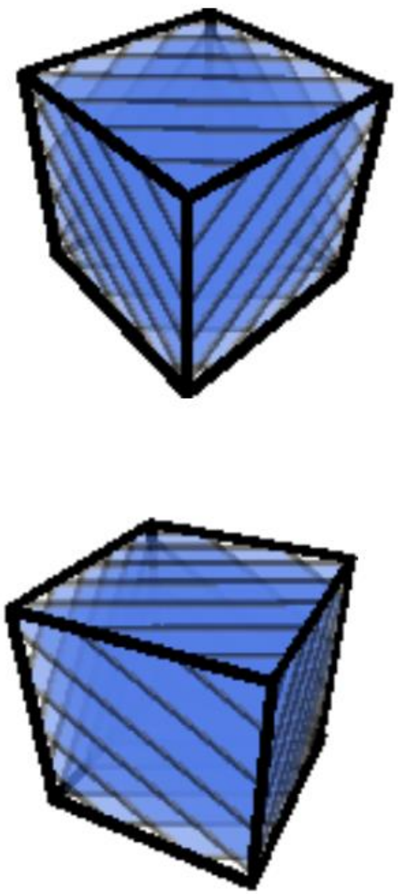


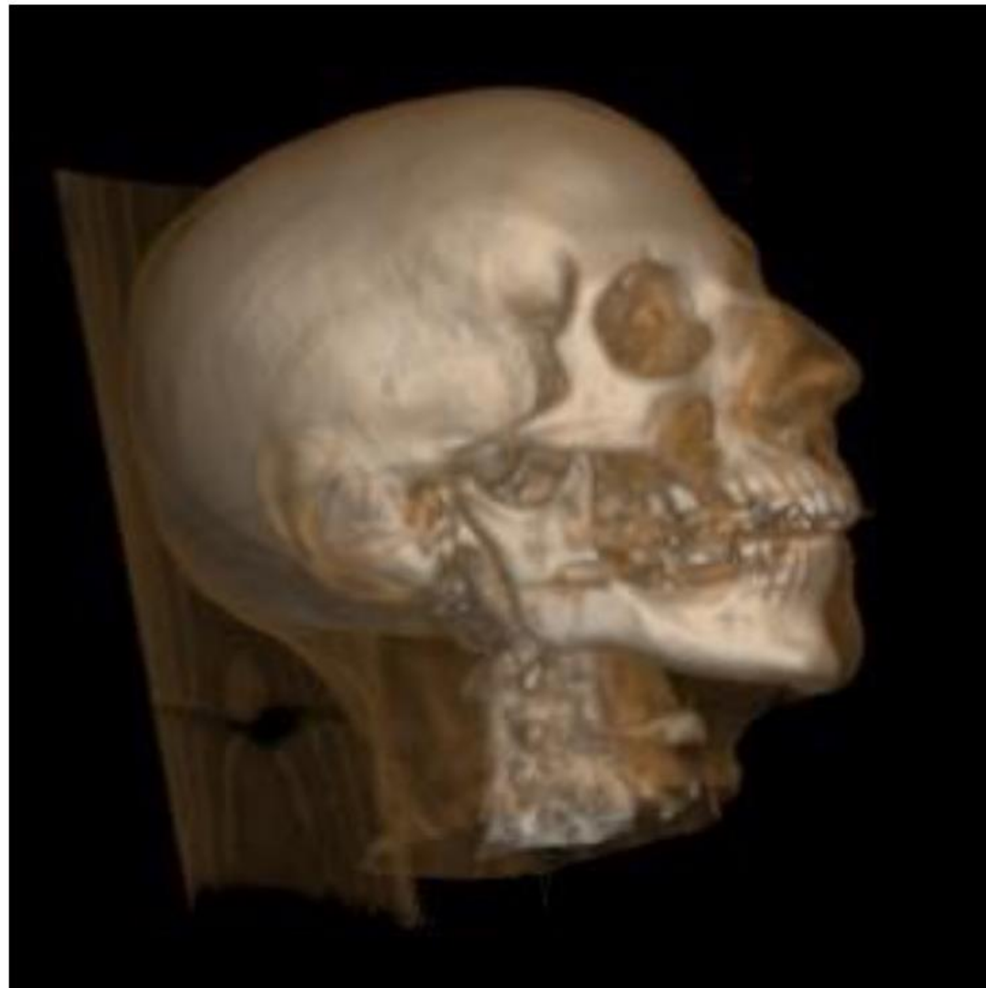
3D Procedural Noise + Solid Modeling



Perlin noise, Ken Perlin

3D Textures and Volume Rendering





Marc Levoy

This Lecture

- Graphics Pipeline
  - Real-time Rendering
  - Evolution of the OpenGL Pipeline
- Shaders
  - Vertex Shader
  - Fragment Shader
- OpenGL Shading Language

Graphics Pipeline

Application

Vertex Processing

Vertex Stream

Triangle Processing

Triangle Stream

Rasterization

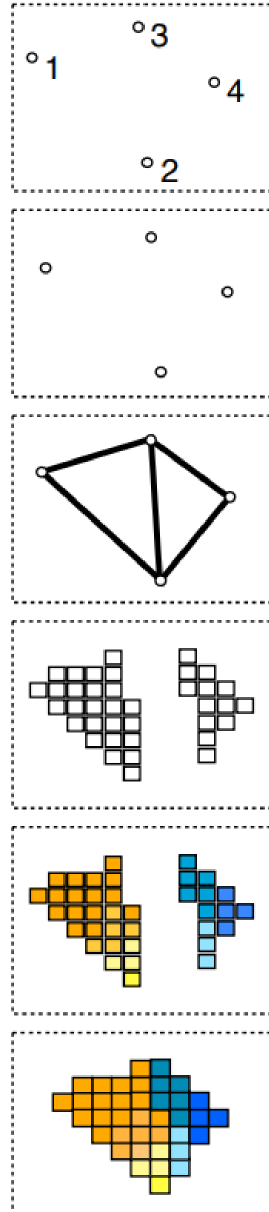
Fragment Stream

Fragment Processing

Shaded Fragments

Framebuffer Operations

Display



Input:
vertices in 3D space

Vertices positioned in screen space

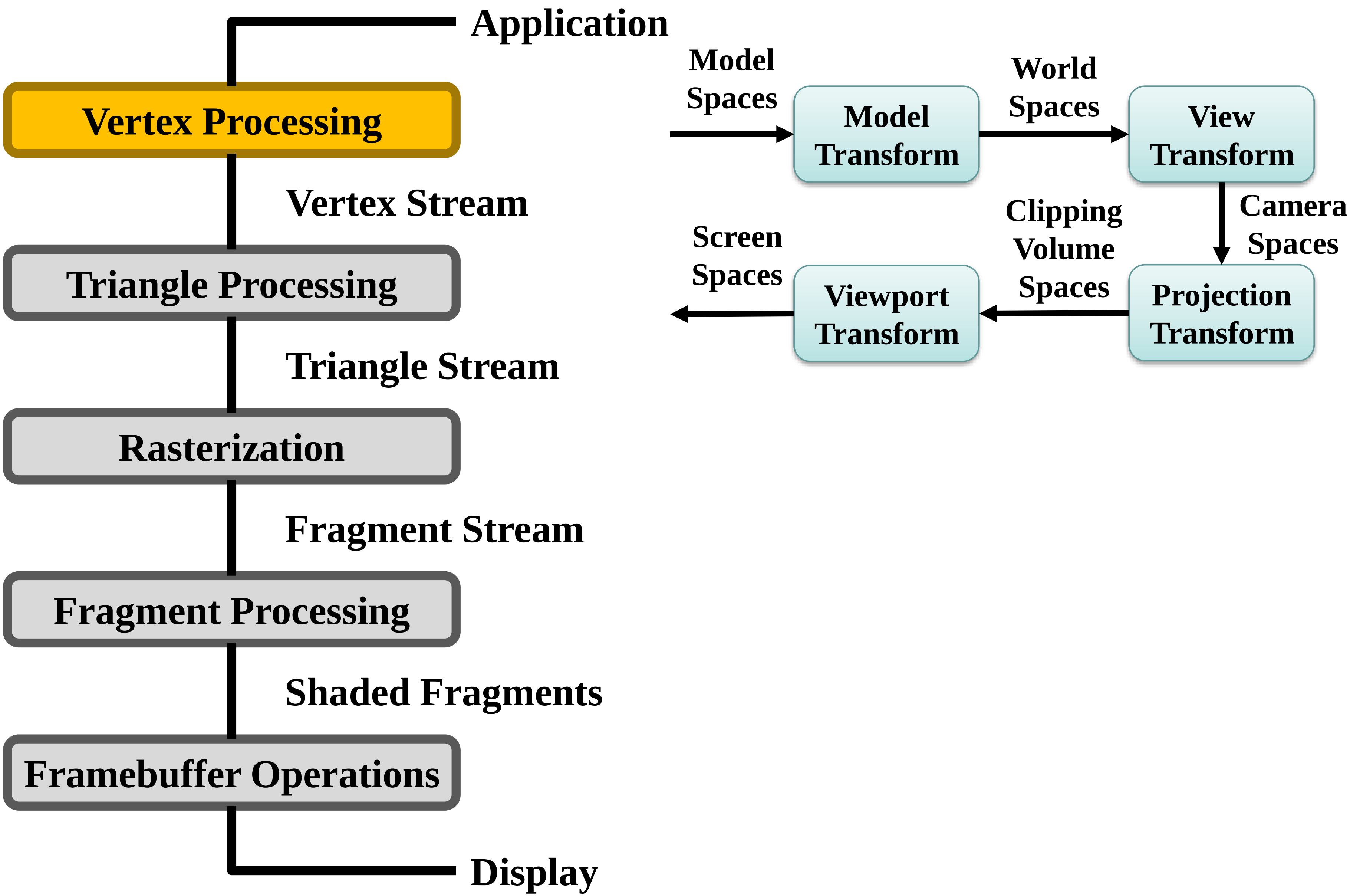
Triangles positioned in screen space

Fragments (one per covered sample)

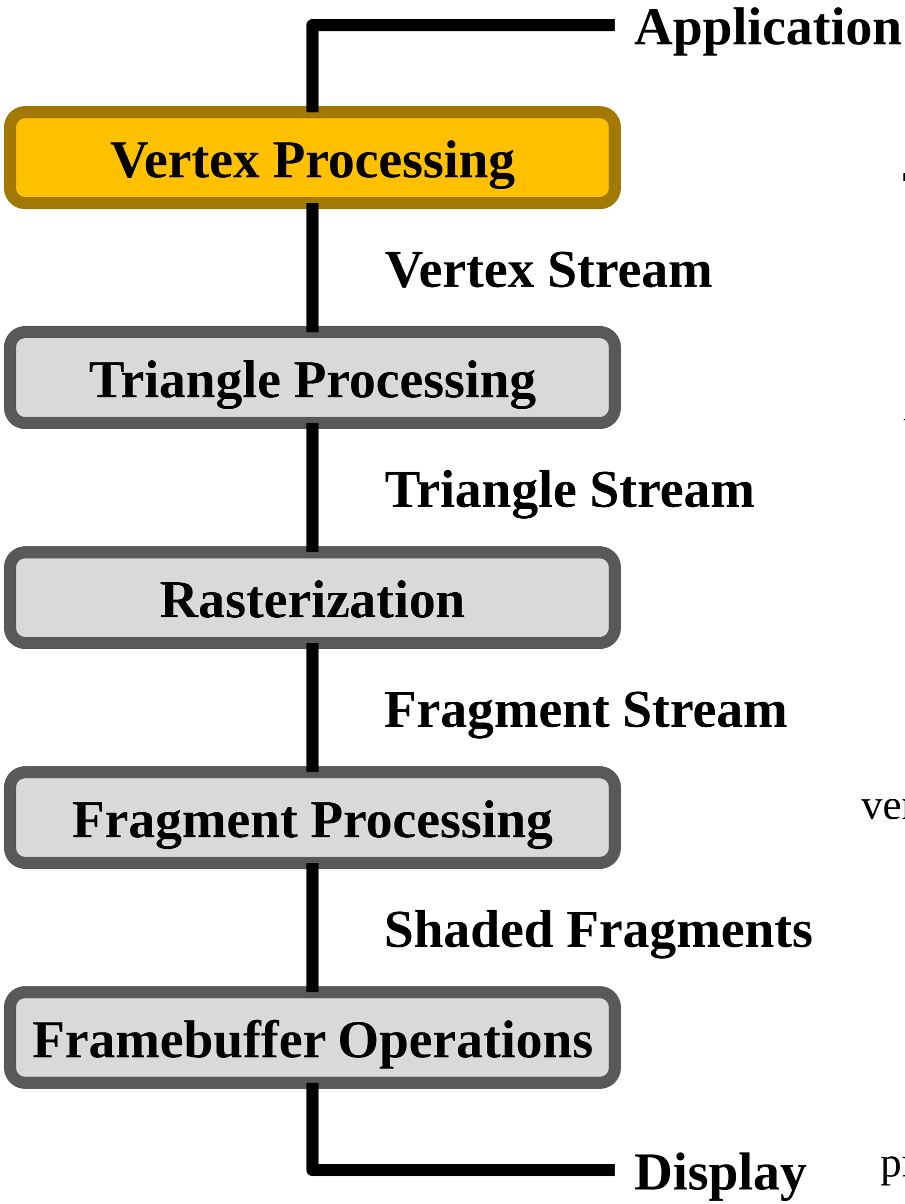
Shaded fragments

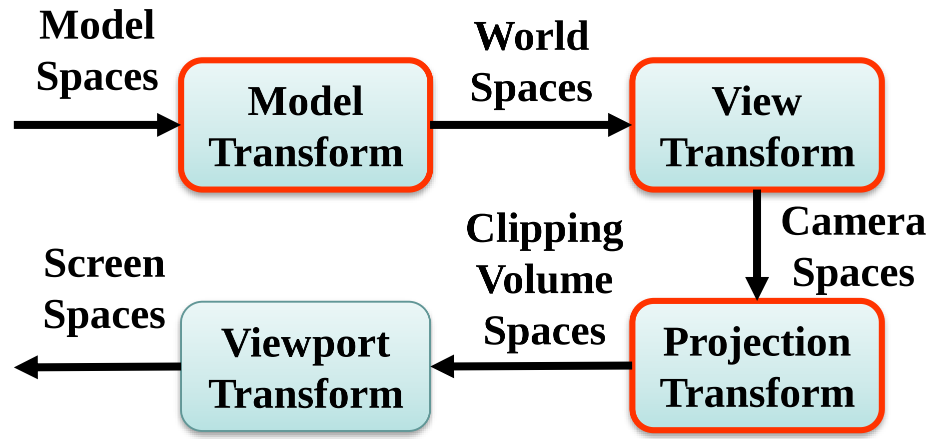
Output:
Image (pixels)

Graphics Pipeline



Graphics Pipeline





Modelview Projection Matrix

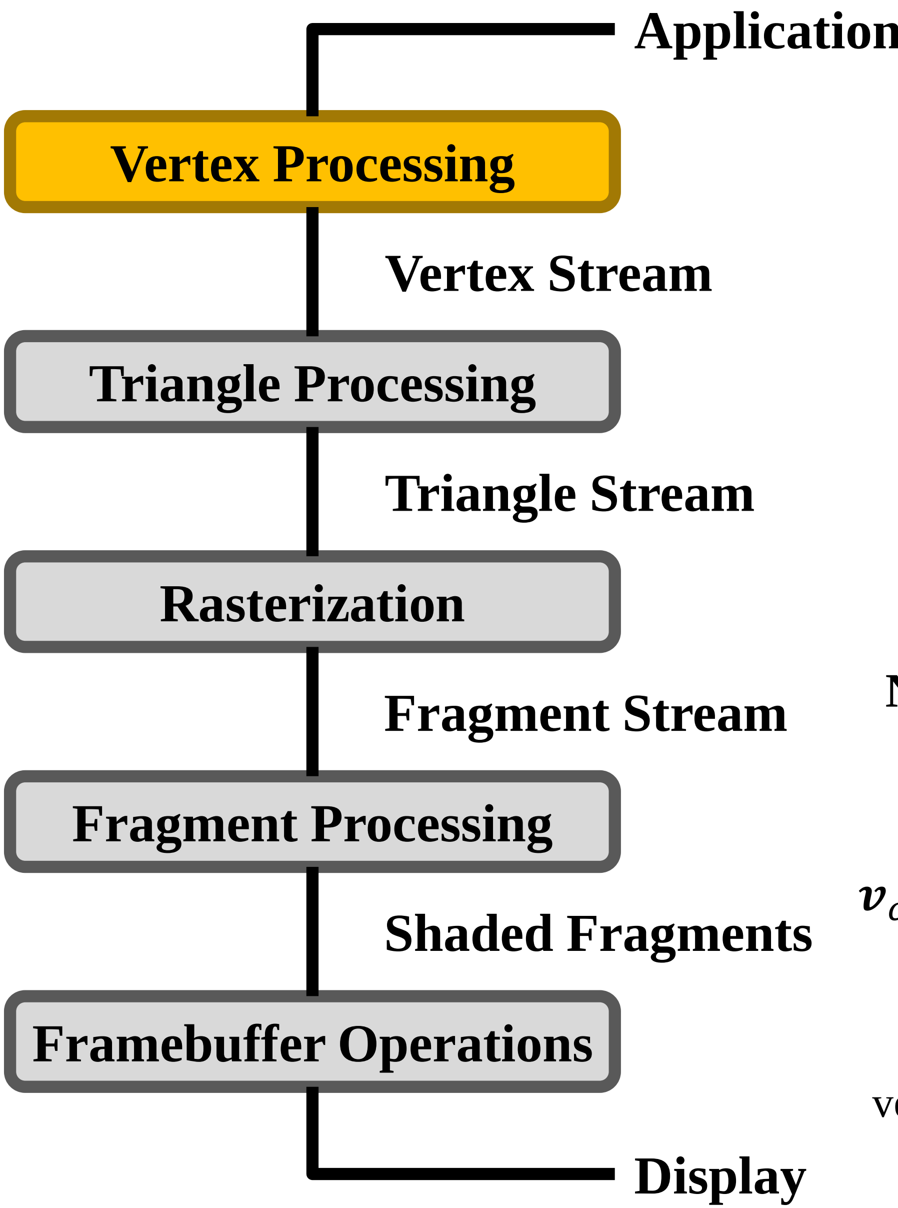
vertex in clip space

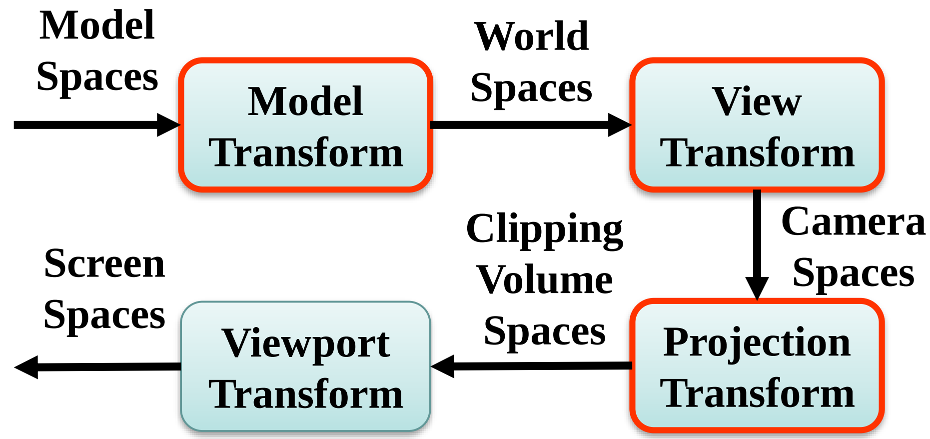
$$\boldsymbol{v}_{clip} = \boldsymbol{M}_{proj} \cdot \boldsymbol{M}_{view} \cdot \boldsymbol{M}_{model} \cdot \boldsymbol{v}$$

$$= \boldsymbol{M}_{proj} \cdot \boldsymbol{M}_{mv} \cdot \boldsymbol{v}$$

projection matrix modelview matrix

Graphics Pipeline



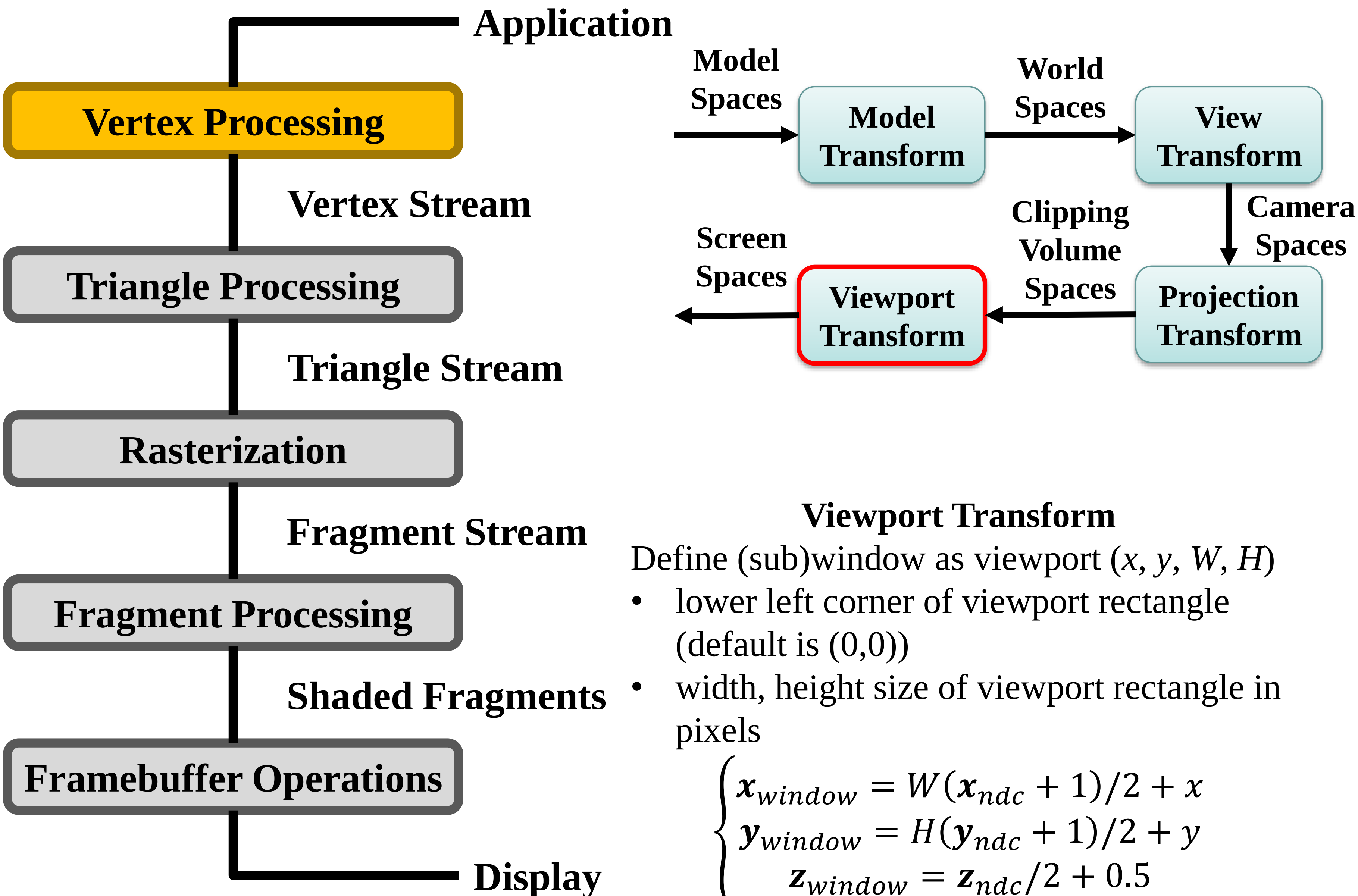


Normalized Device Coordinates (NDC)

$$\boldsymbol{v}_{clip} = \begin{pmatrix} \boldsymbol{x}_{clip} \\ \boldsymbol{y}_{clip} \\ \boldsymbol{z}_{clip} \\ \boldsymbol{w}_{clip} \end{pmatrix} \rightarrow \boldsymbol{v}_{ndc} = \begin{pmatrix} \boldsymbol{x}_{clip}/\boldsymbol{w}_{clip} \\ \boldsymbol{y}_{clip}/\boldsymbol{w}_{clip} \\ \boldsymbol{z}_{clip}/\boldsymbol{w}_{clip} \\ 1 \end{pmatrix}$$

vertex in clip space — vertex in NDC

Graphics Pipeline



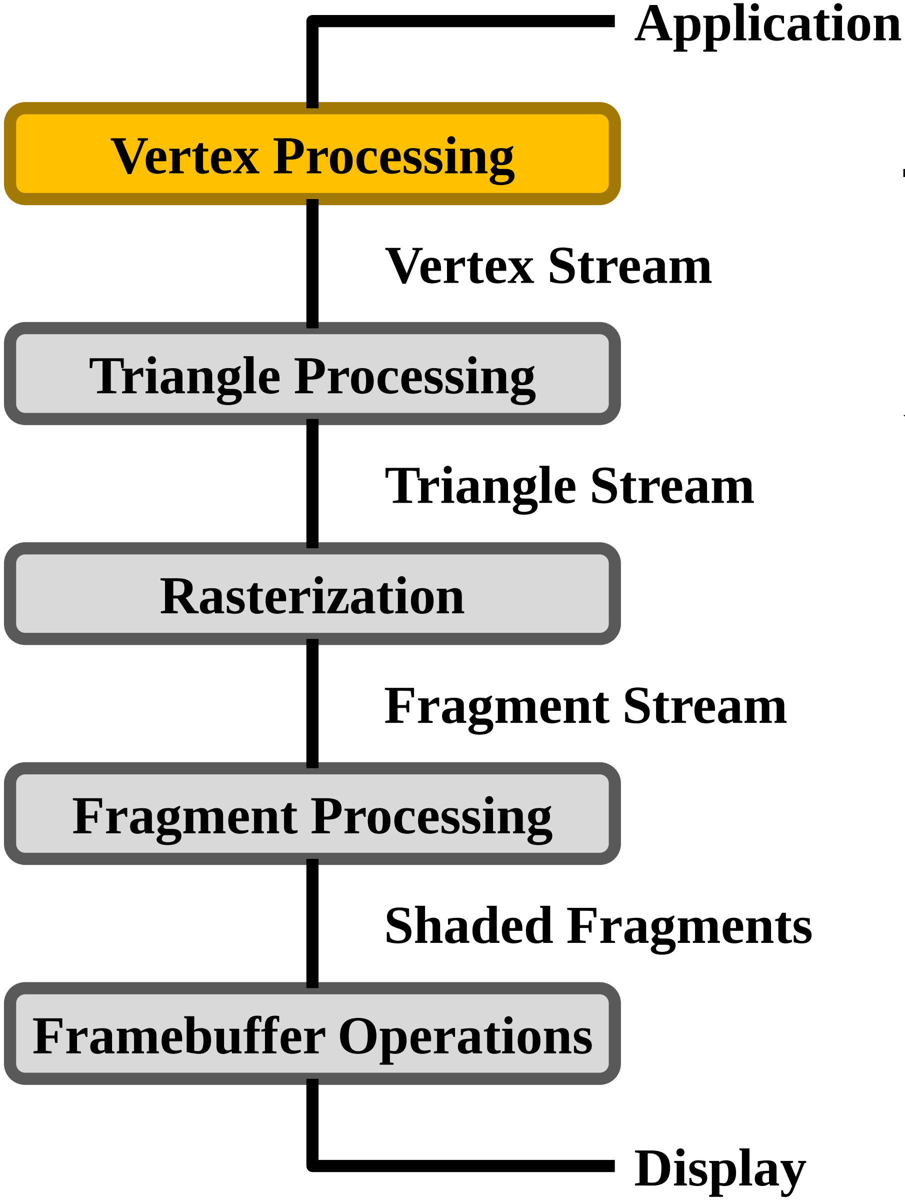
Viewport Transform

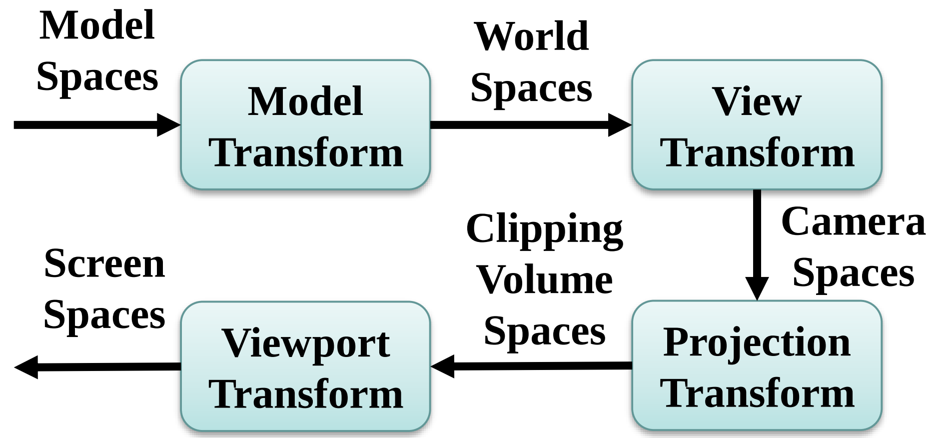
Define (sub)window as viewport (x, y, W, H)

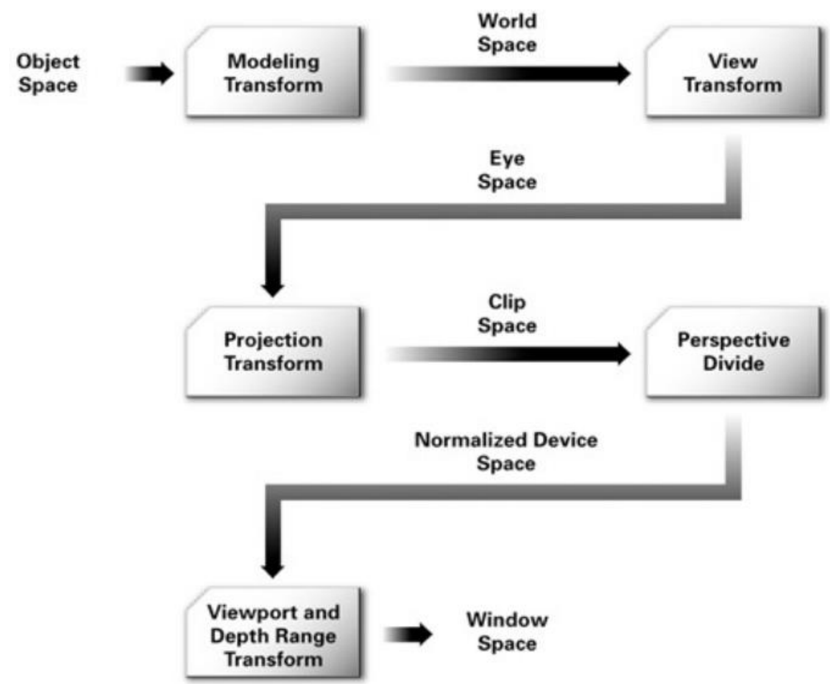
- lower left corner of viewport rectangle (default is (0,0))
- width, height size of viewport rectangle in pixels

$$\begin{cases} \boldsymbol{x}_{window} = W(\boldsymbol{x}_{ndc} + 1)/2 + x \\ \boldsymbol{y}_{window} = H(\boldsymbol{y}_{ndc} + 1)/2 + y \\ \boldsymbol{z}_{window} = \boldsymbol{z}_{ndc}/2 + 0.5 \end{cases}$$

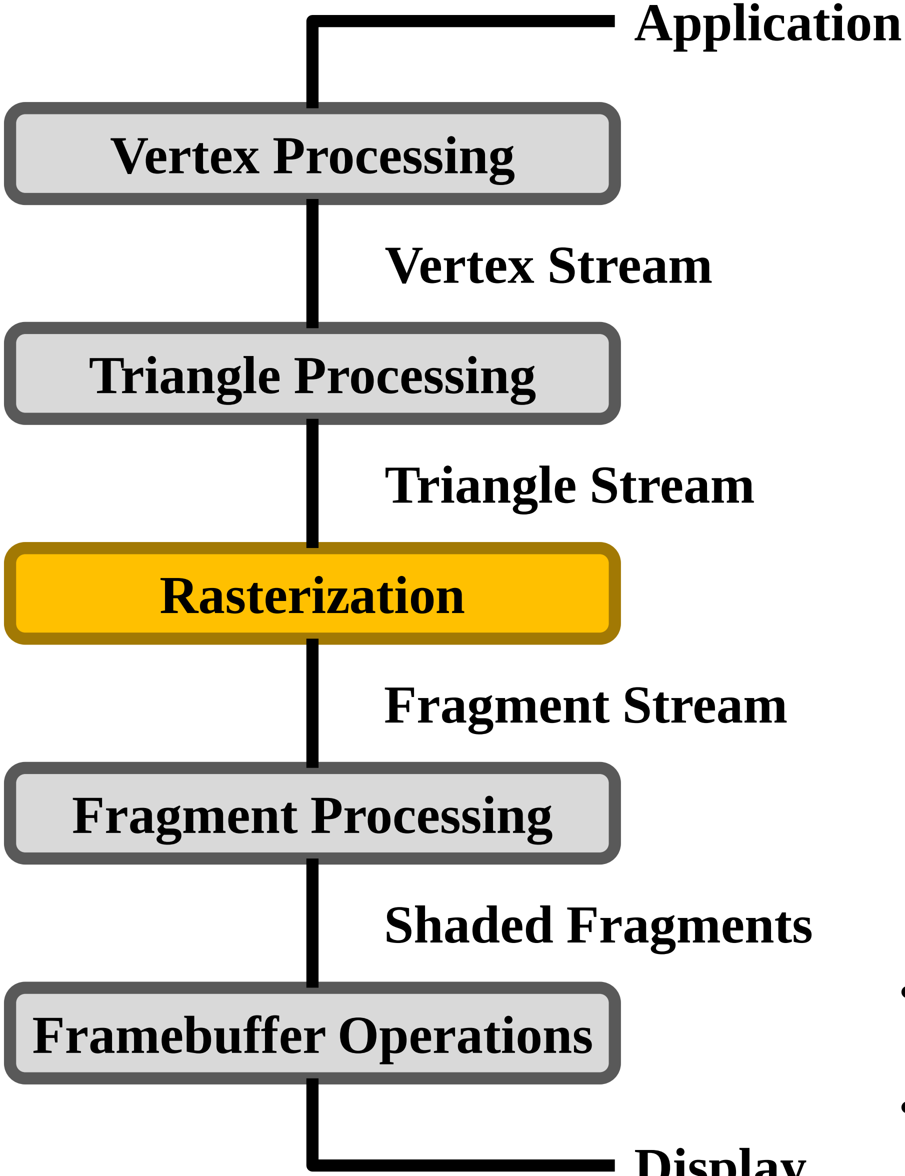
Graphics Pipeline



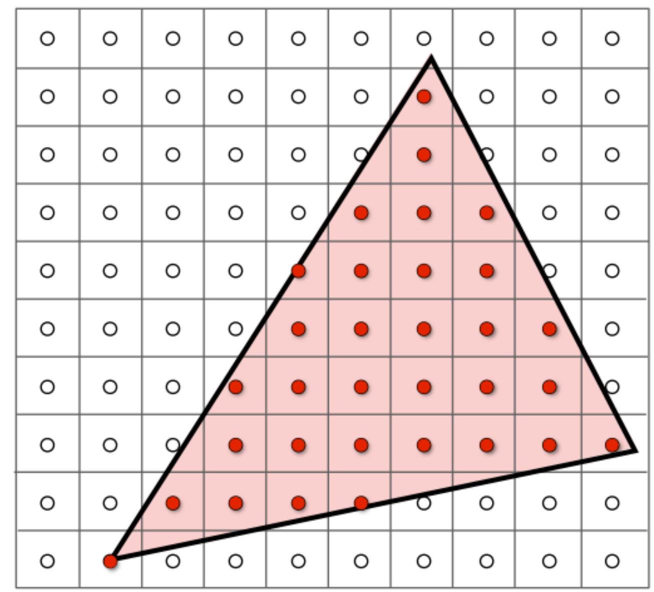




Graphics Pipeline



Sampling triangle coverage



Rasterization

- Determine which fragments are inside the triangles
- Interpolate vertex attributes (e.g color) to all fragments

Graphics Pipeline

Application

Vertex Processing

Vertex Stream

Triangle Processing

Triangle Stream

Rasterization

Fragment Stream

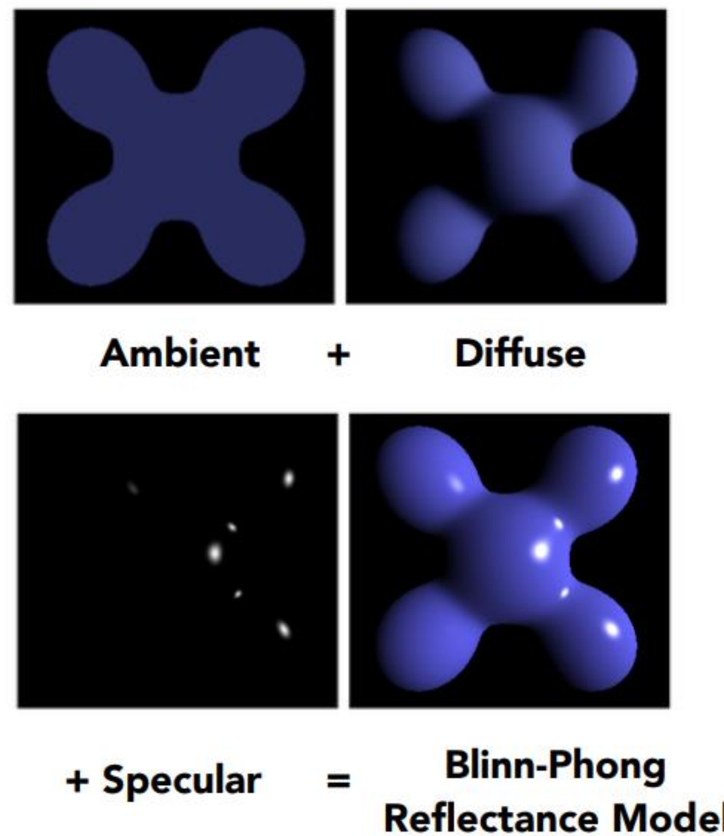
Fragment Processing

Shaded Fragments

Framebuffer Operations

Display

Shading



Graphics Pipeline

Application

Vertex Processing

Vertex Stream

Triangle Processing

Triangle Stream

Rasterization

Fragment Stream

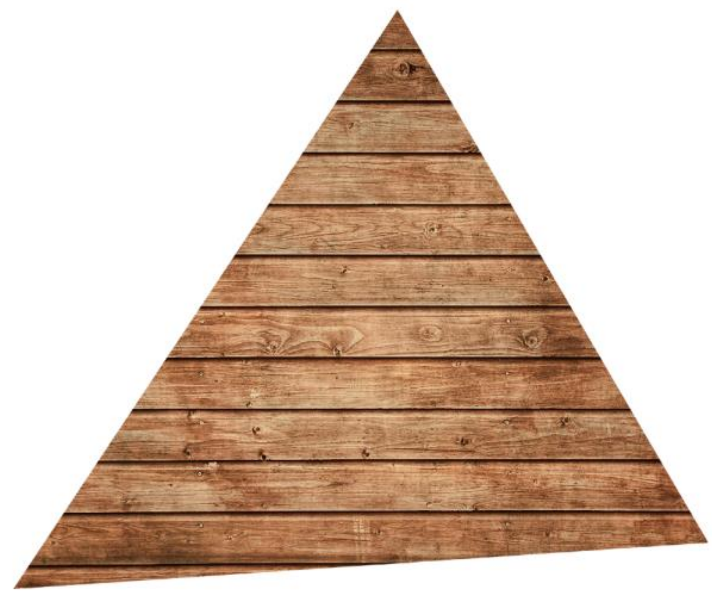
Fragment Processing

Shaded Fragments

Framebuffer Operations

Display

Texture mapping



Graphics Pipeline

Application

Vertex Processing

Vertex Stream

Triangle Processing

Triangle Stream

Rasterization

Fragment Stream

Fragment Processing

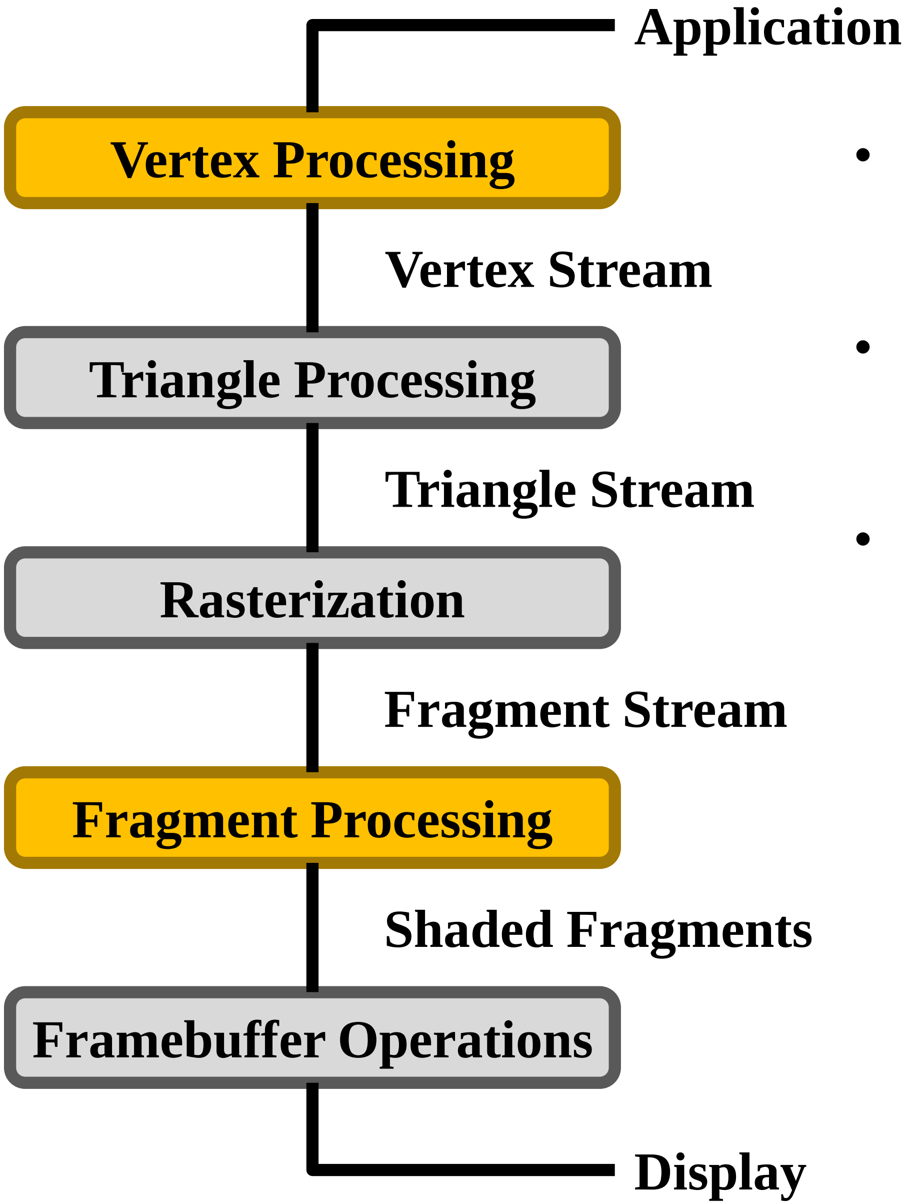
Shaded Fragments

Framebuffer Operations

Display

Z-Buffer Visibility Tests

Graphics Pipeline



- Functionality in parts of the GPU pipeline specified by user programs
- Called shaders, or shader programs, executed on GPU
- Not all functionality in the pipeline is programmable

This Lecture

- Graphics Pipeline
  - Real-time Rendering
  - Evolution of the OpenGL Pipeline

- Shaders
  - Vertex Shader
  - Fragment Shader

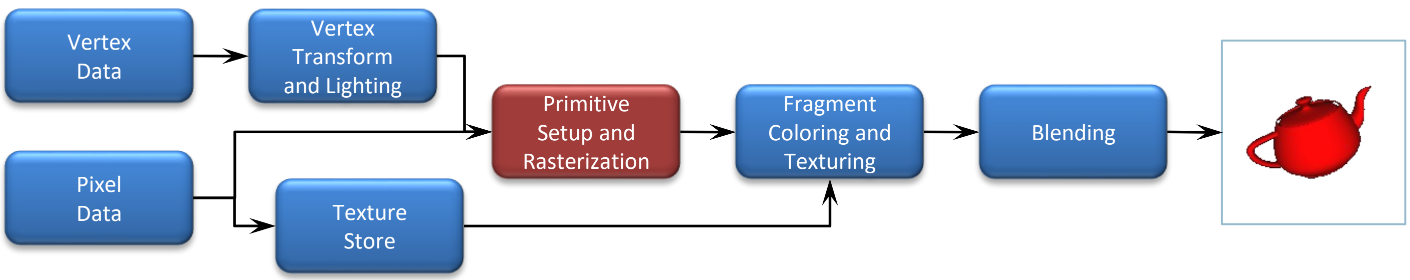
- OpenGL Shading Language

What Is OpenGL?

- OpenGL is a computer graphics rendering *application programming interface*, or API (for short)
  - With it, you can generate high-quality color images by rendering with geometric and image primitives
  - It forms the basis of many interactive applications that include 3D graphics
  - By using OpenGL, the graphics part of your application can be
    - operating system independent
    - window system independent

OpenGL 1.0

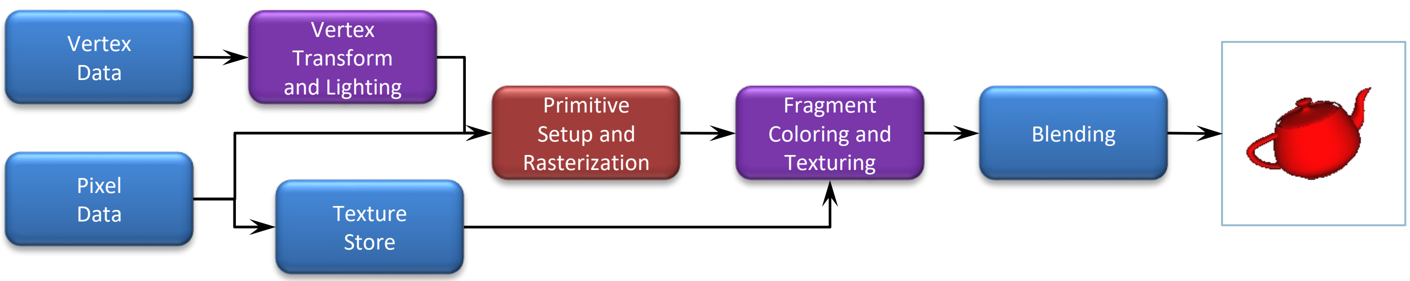
- OpenGL 1.0 was released on July 1st, 1994
- Its pipeline was entirely *fixed-function*
  - the only operations available were fixed by the implementation



- The pipeline evolved but remained based on fixed-function operation through OpenGL versions 1.1 through 2.0 (Sept. 2004)

OpenGL 2.0

- OpenGL 2.0 (officially) added programmable shaders
  - *vertex shading* augmented the fixed-function transform and lighting stage
  - *fragment shading* augmented the fragment coloring stage



- However, the fixed-function pipeline was still available

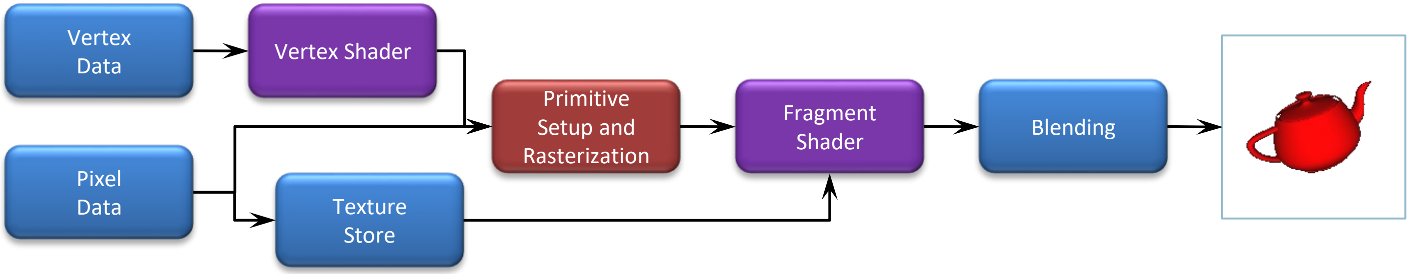
OpenGL 3.0

- OpenGL 3.0 introduced the *deprecation model*
  - the method used to remove features from OpenGL
- The pipeline remained the same until OpenGL 3.1 (released March 24th, 2009)
- Introduced a change in how OpenGL contexts are used

Context Type	Description
Full	Includes all features (including those marked deprecated) available in the current version of OpenGL
Forward Compatible	Includes all non-deprecated features (i.e., creates a context that would be similar to the next version of OpenGL)

OpenGL 3.1

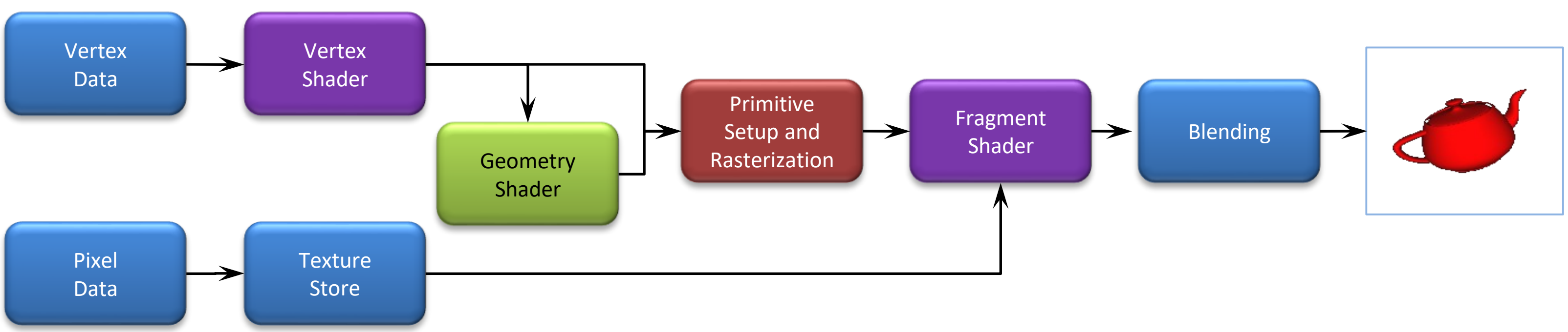
- OpenGL 3.1 removed the fixed-function pipeline
  - programs were required to use only shaders



- Additionally, almost all data is GPU-resident
  - all vertex data sent using buffer objects

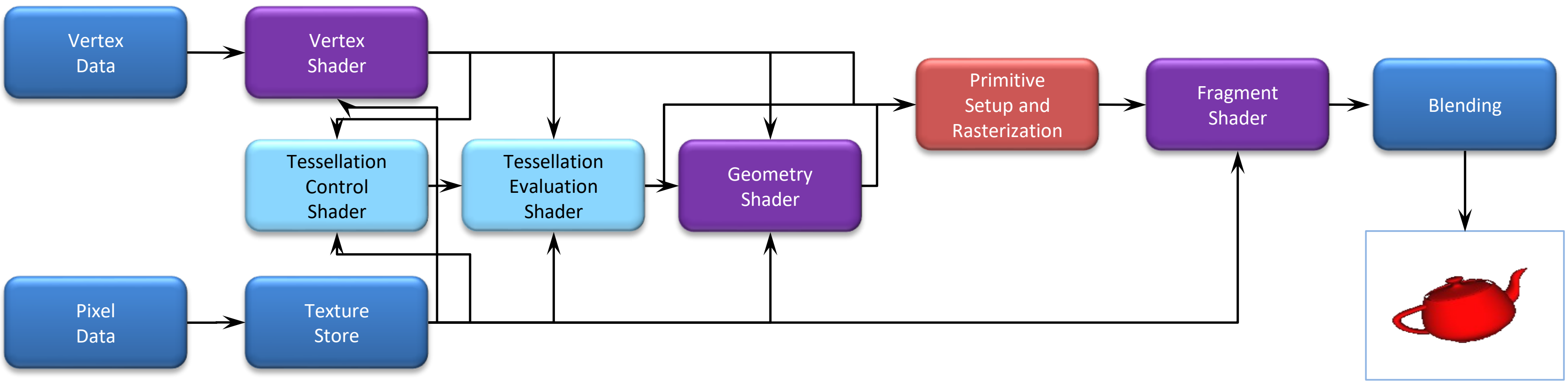
OpenGL 3.2

- OpenGL 3.2 (released August 3rd, 2009) added an additional shading stage – geometry shaders
  - modify geometric primitives within the graphics pipeline

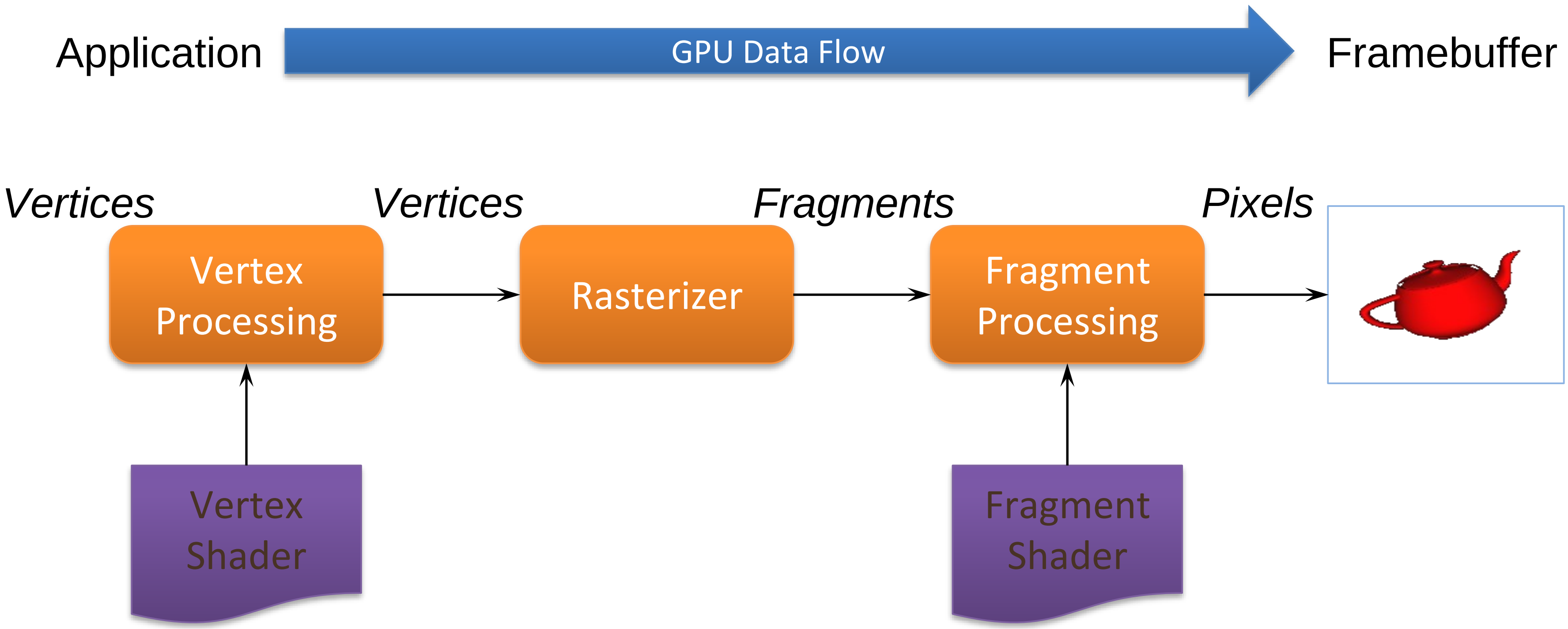


The Latest Pipelines

- OpenGL 4.1 (released July 25th, 2010) included additional shading stages – *tessellation-control* and *tessellation-evaluation* shaders
- Latest version is 4.6



A Simplified Pipeline Model



OpenGL Programming

- Modern OpenGL programs essentially do the following steps:
  - Create shader programs
  - Create buffer objects and load data into them
  - “Connect” data locations with shader variables
  - Render

A first program

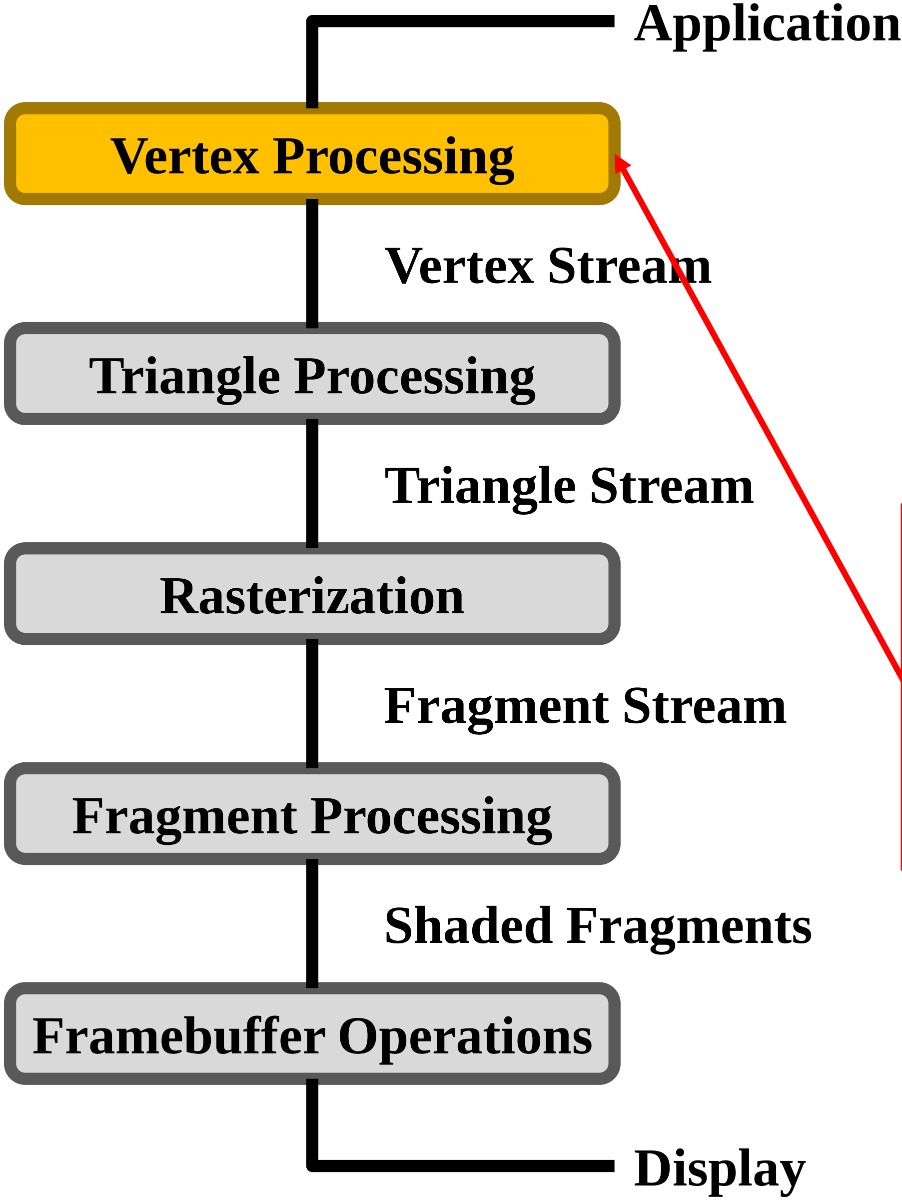
- Render a cube with colors at each vertex

- The example demonstrates:
  - initializing vertex data
  - organizing data for rendering
  - simple object modeling
    - building up 3D objects from geometric primitives
    - building geometric primitives from vertices

This Lecture

- Graphics Pipeline
  - Real-time Rendering
  - Evolution of the OpenGL Pipeline
- Shaders
  - Vertex Shader
  - Fragment Shader
- OpenGL Shading Language

Vertex Shader



Input

- Vertex position, normal, color, texture coordinates
- Modelview matrix, projection matrix, normal matrix
- …

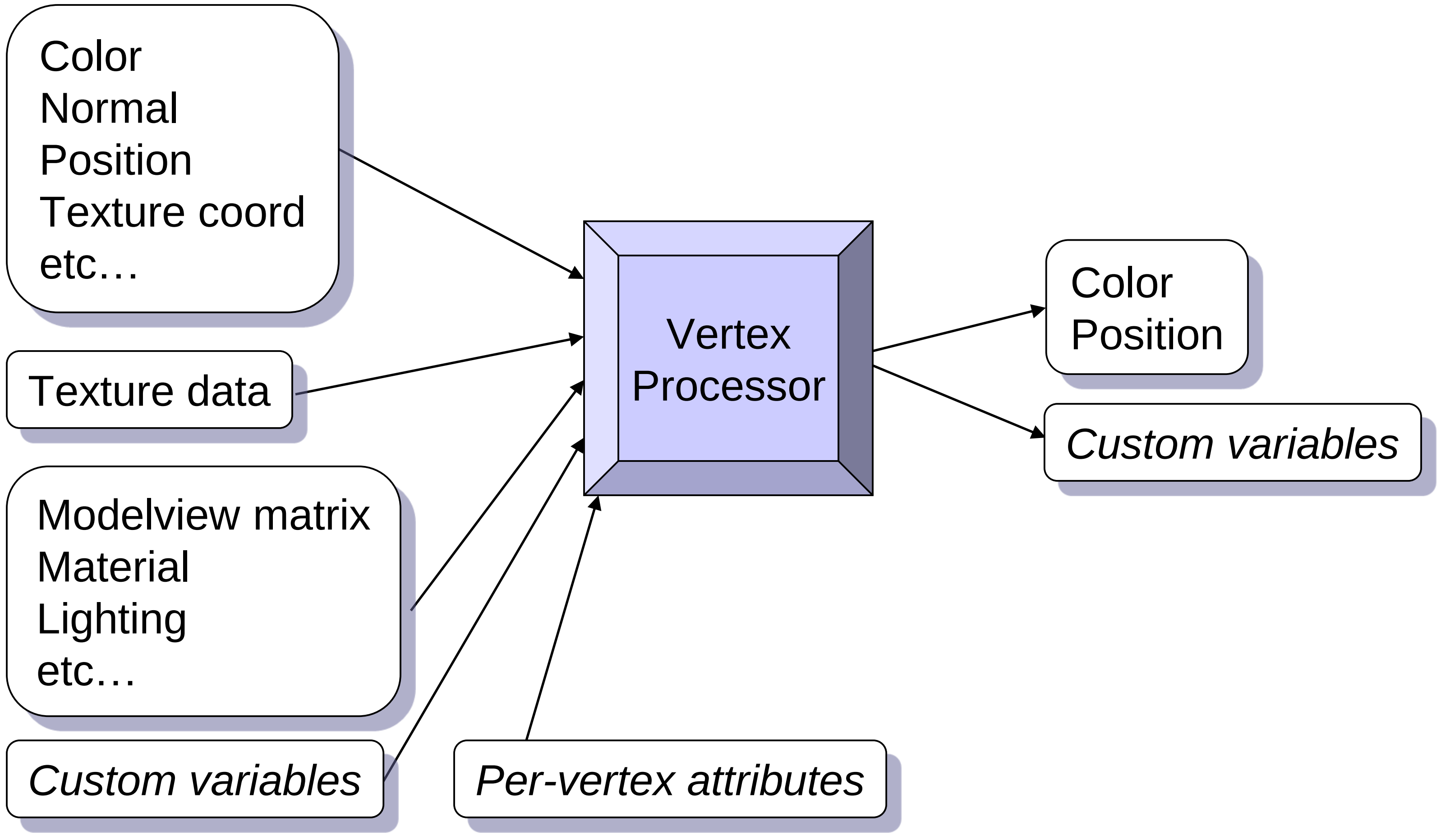
```
void main ()
{
  // do something here
  …

}
```

Output

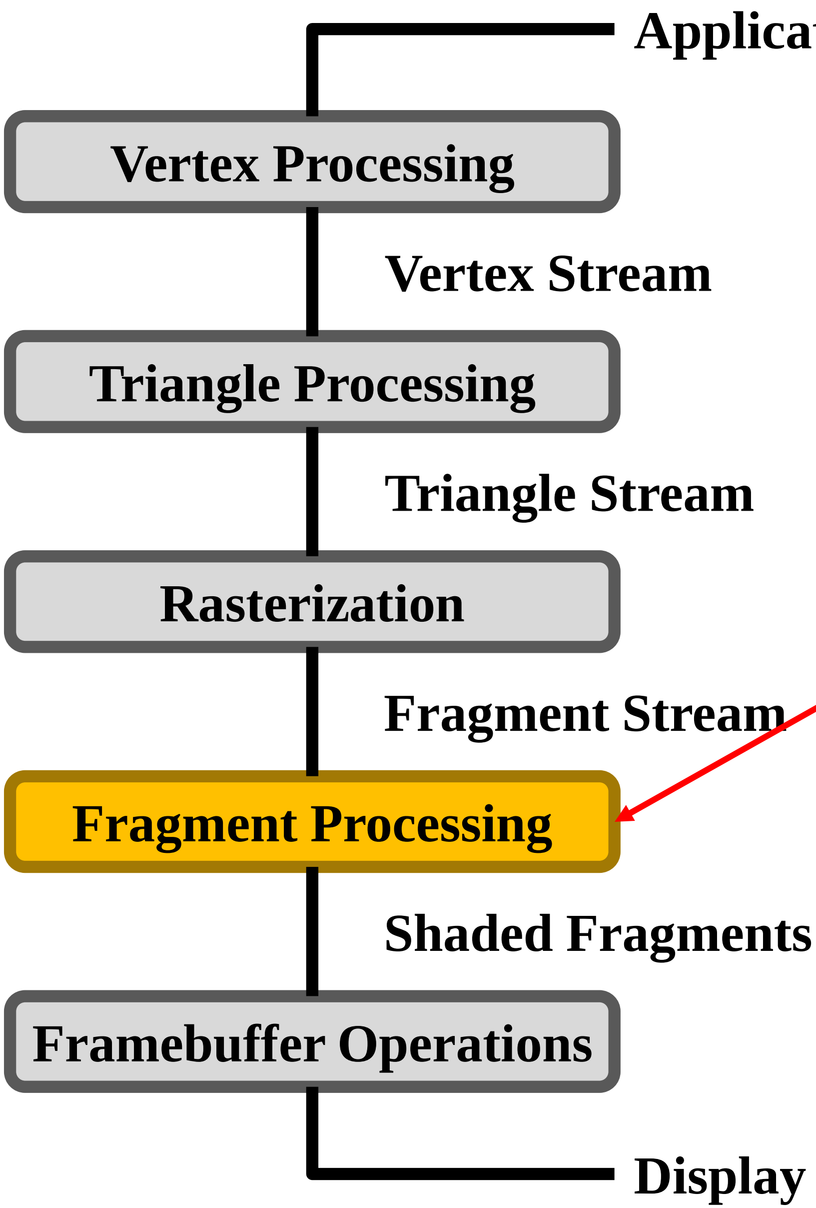
- Transformed vertex position(in clip coords), texture coordinates
- …

Vertex Shader

- Inputs and outputs



Fragment Shader



Input

- Vertex position in window cords, texture coordinates
- …

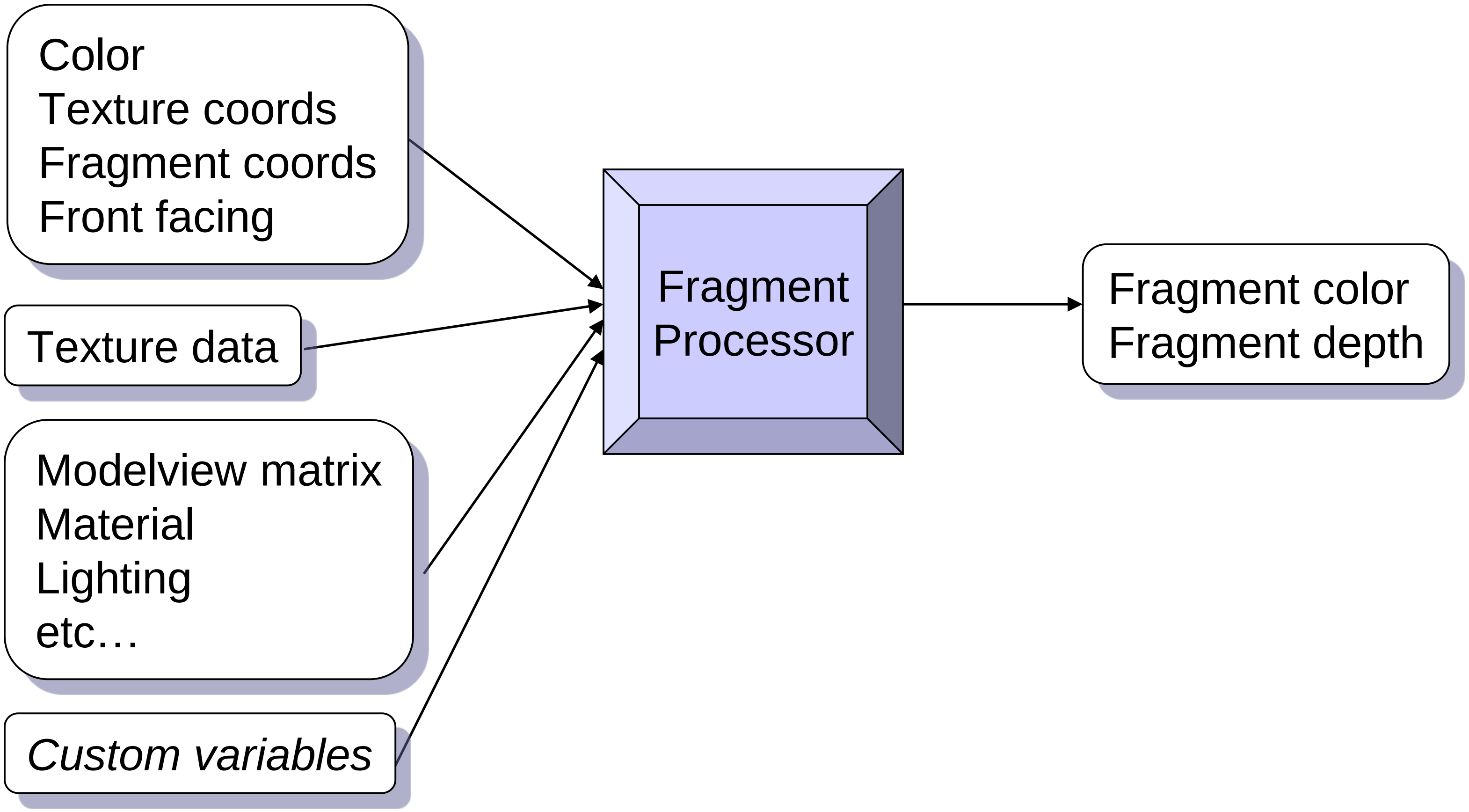
```
void main ()
{
  // do something here
  …

}
```

Output

- Fragment color
- Fragment depth
- …

Fragment Shader

- Inputs and outputs



Why do we need shaders?

- Massively parallel computing
- GPUs are designed to be parallel processors
- Vertex shaders are independently executed for each vertex on GPU (in parallel)
- Fragment shaders are independently executed for each fragment on GPU (in parallel)

This Lecture

- Graphics Pipeline
  - Real-time Rendering
- Shaders
  - Vertex Shader
  - Fragment Shader
- OpenGL Shading Language

Shader programs

- Written in a shading language
- Example
  - Cg, early shading language by Nvidia (deprecated)
  - OpenGL Shading Language (GLSL)
  - DirectX Shading Language HLSL (high level shading language)
  - All similar to C, with specialties
- Driven by more and more flexible GPUs

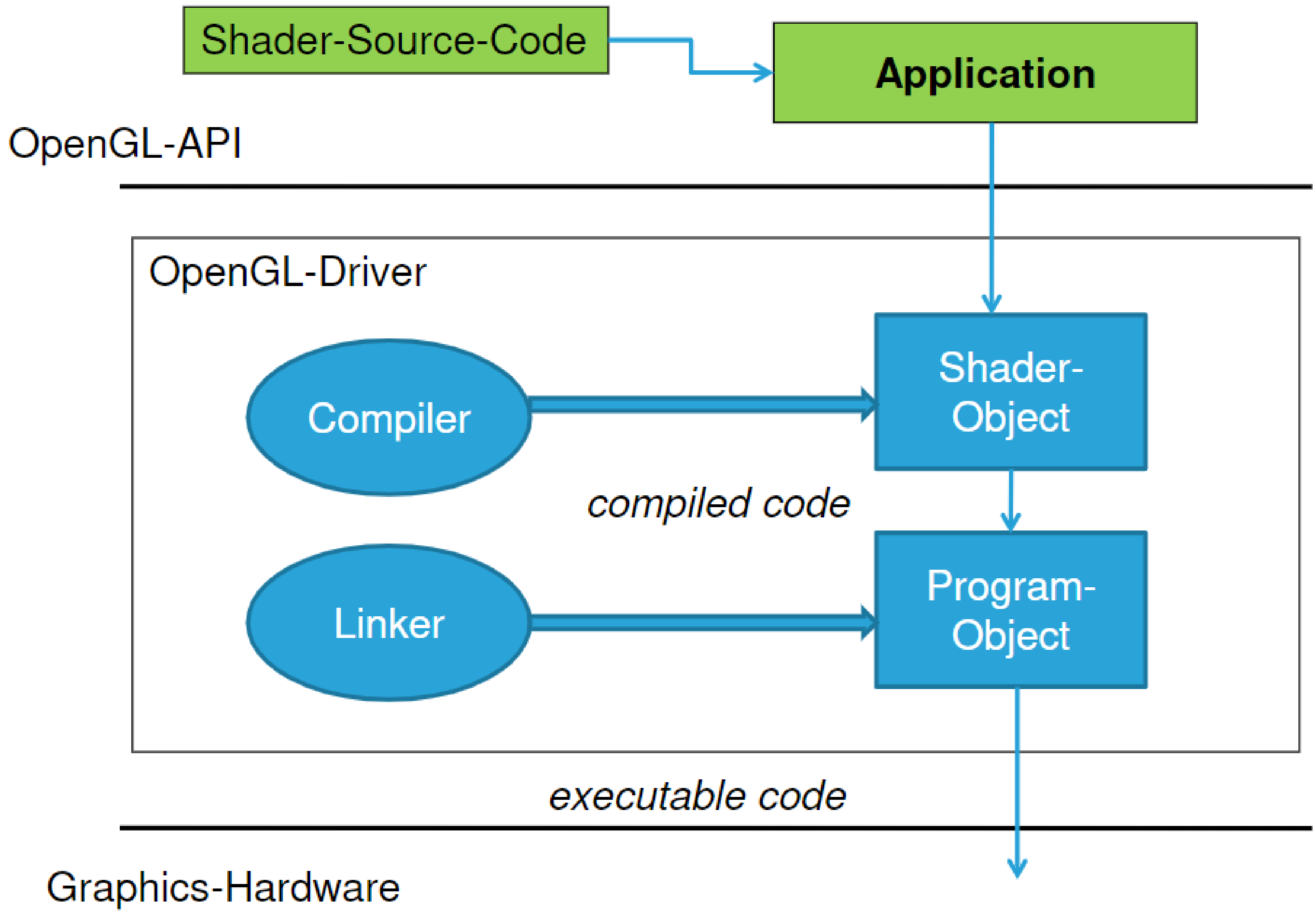
OpenGL Shading Language (GLSL)

- high-level programming language for shaders
- syntax similar to C (i.e. has main function and many other similarities)
- usually very short programs that are executed in parallel on GPU
- good introduction / tutorial:
  - https://www.opengl.org/sdk/docs/tutorials/TyphoonLabs/

OpenGL Shading Language (GLSL)

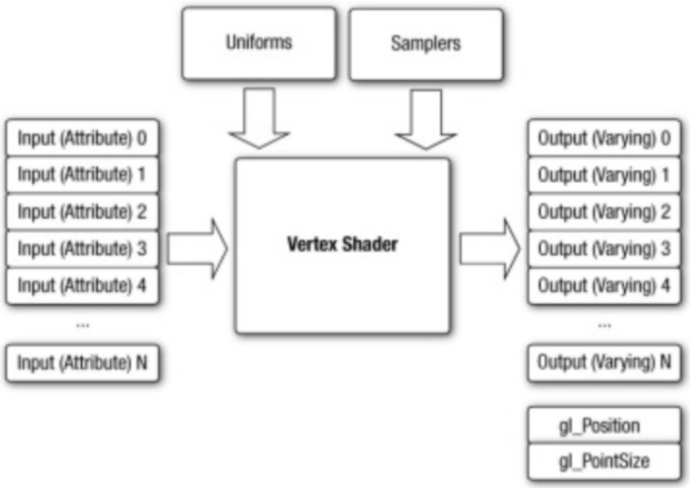
- Small C-like programs executed on the graphics-hardware
- Replace fixed function pipeline with shaders
- Shader-Types
  - Vertex Shader (VS): per vertex operation
  - Geometry Shader (GS): per primitive operation
  - Fragment Shader (FS): per fragment operation
- Used e.g. for transformations and lighting

Shader-Execution model



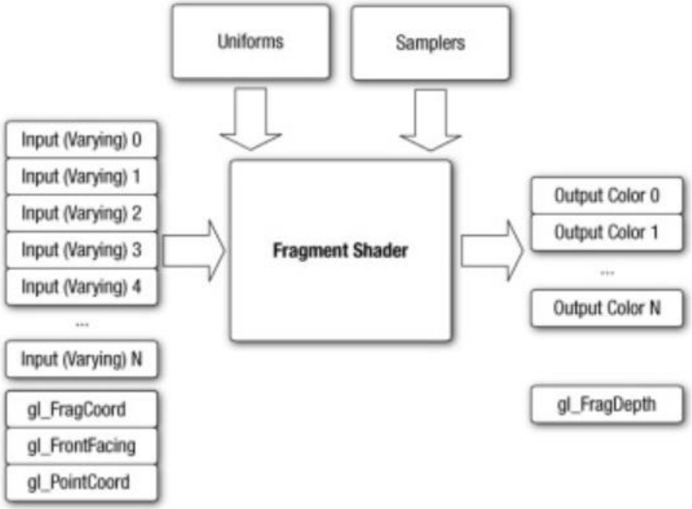
Vertex Shader

- Vertex Shader program
  - Source code or executable that describes the operations that will be performed on the vertex
- inputs (or attributes)
  - Per-vertex data supplied using vertex arrays
- Uniforms
  - Constant data used by the vertex (or fragment) shader.
- Samplers
  - Specific types of uniforms that represent textures used by the vertex shader



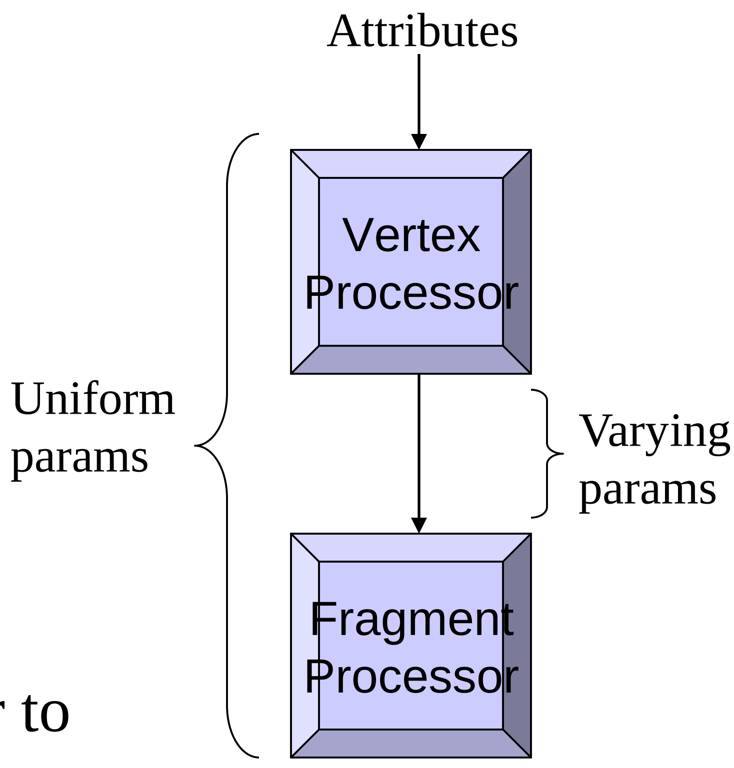
Fragment Shader

- Fragment Shader program
  - Source code or executable that describes the operations that will be performed on the fragment
- inputs
  - Outputs of the vertex shader that are generated by the rasterization unit for each fragment using interpolation
- Uniforms
  - Constant data used by the vertex (or fragment) shader.
- Samplers
  - Specific types of uniforms that represent textures used by the fragment shader



How do the shaders communicate?

- There are three types of shader parameter in GLSL:
  - *Uniform parameters*
    - Set throughout execution
    - Example: surface color
  - *Attribute parameters*
    - Set per vertex
    - Example: local tangent
  - *Varying parameters*
    - Passed from vertex processor to fragment processor
    - Example: transformed normal



GLSL Data Types

- Scalar types: `float, int, bool`
- Vector types: `vec2, vec3, vec4`
  `ivec2, ivec3, ivec4`
  `bvec2, bvec3, bvec4`
- Matrix types: `mat2, mat3, mat4`
- Texture sampling: `sampler1D, sampler2D, sampler3D, samplerCube`
- C++ Style Constructors

```
vec3 a = vec3(1.0, 2.0, 3.0);
```

Memory Layout and Matrices

- The OpenGL/WebGL/GLSL convention is layout matrices in column-major order

$$\begin{bmatrix} a & b & c & t_x \\ d & e & f & t_y \\ g & h & i & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- is laid out as 16 contiguous floating point numbers

$[a, d, g, 0, b, e, h, 0, c, f, i, 0, t_x, t_y, t_z, 1]$

Operators

- Standard C/C++ arithmetic and logic operators

- Overloaded operators for matrix and vector operations

```
mat4 m;
vec4 a, b, c;

b = a*m;
c = m*a;
```

Components and Swizzling

- Access vector components using either:
  - [ ] (c-style array indexing)
  - `xyzw`, `rgba` or `strq` (named components)

- For example:

```
vec3 v;
v[1], v.y, v.g, v.t
```
- all refer to the same element

- Component swizzling:

```
vec3 a, b;
a.xy = b.yx;
```

Qualifiers

- `in`, `out`
  - Copy vertex attributes and other variable into and out of shaders

```
in  vec2 texCoord;
out vec4 color;
```

- `uniform`
  - shader-constant variable from application

```
uniform float time;
uniform vec4 rotation;
```

Functions

- Built in
  - Arithmetic: `sqrt`, `power`, `abs`
  - Trigonometric: `sin`, `asin`
  - Graphical: `length`, `reflect`

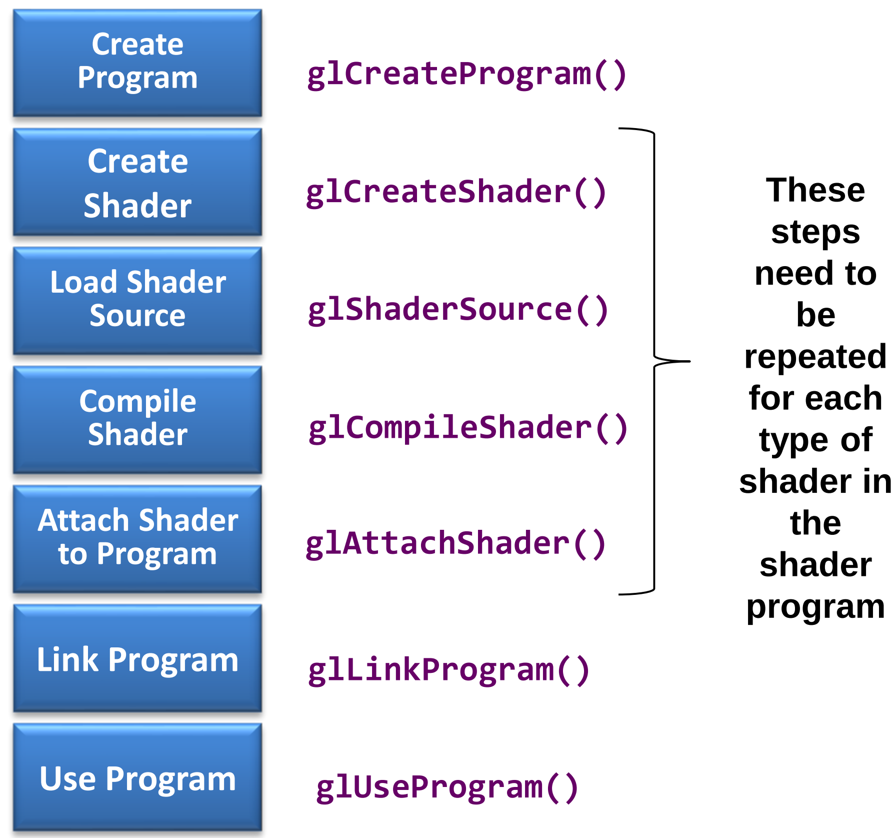
- User defined

Built-in Variables

- `gl_Position`
  - (required) output position from vertex shader
- `gl_FragCoord`
  - input fragment position
- `gl_FragDepth`
  - input depth value in fragment shader

Getting Your Shaders into OpenGL

- Shaders need to be compiled and linked to form an executable shader program
- OpenGL provides the compiler and linker
- A program must contain
  - vertex and fragment shaders
  - other shaders are optional



Associating Shader Variables and Data

- Need to associate a shader variable with an OpenGL data source
  - vertex shader attributes → app vertex attributes
  - shader uniforms → app provided uniform values
- OpenGL relates shader variables to indices for the app to set
- Two methods for determining variable/index association
  - specify association before program linkage
  - query association after program linkage

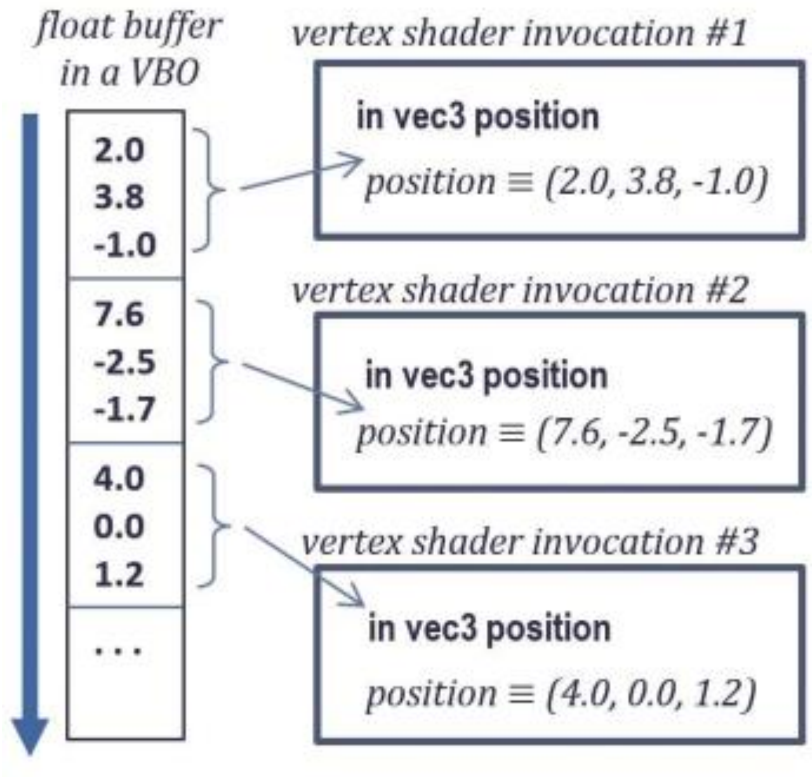
Buffers and Vertex Attributes

- Vertices of objects must be available to the vertex shader
  - in vec4 position (define the vertices inside the shader)
  - usually only once, done in initialization

- For every frame (rendering)
  - Enable the buffer containing the vertices
  - Associate the buffer with a vertex attribute
  - Enable the vertex attribute
  - Call glDrawArrays (…)

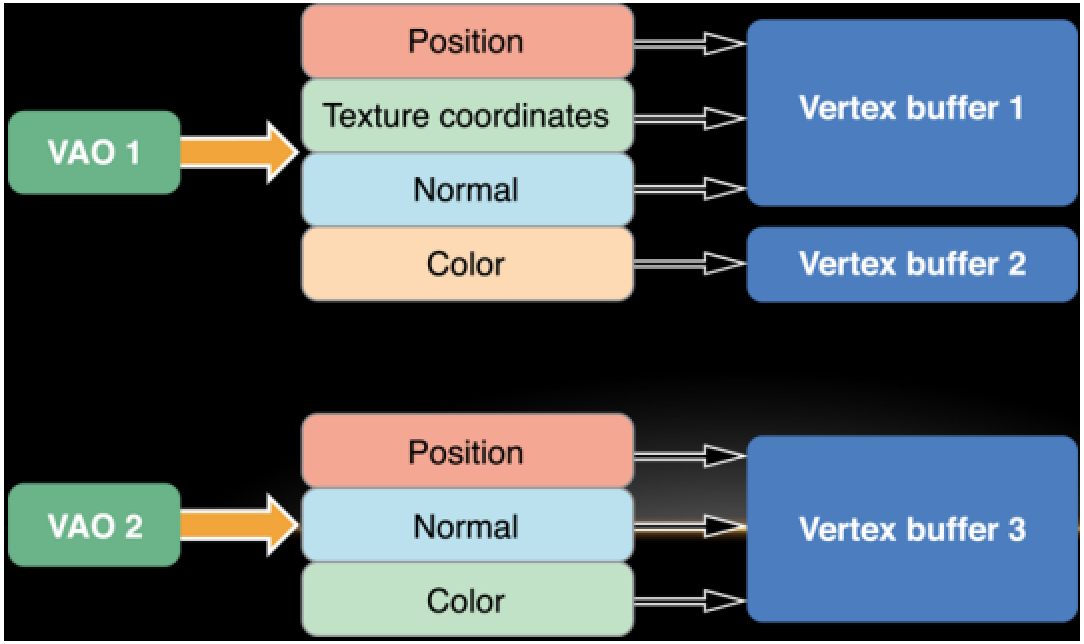
OpenGL Buffers

- Buffers are contained in a VBO – Vertex Buffer Object
- Most scenes are going to have several objects
  - Will need a VBO for each object



Vertex Array Objects

- OpenGL requires at least one vertex array object (VAO)
- VAO allows you to organize multiple buffer objects in one structure
  - Makes it easier to manipulate multiple objects in a scene



VAO and VBO

- `unsigned int VAO, VBO`
- `glGenVertexArrays(1, &VAO); // create one VAO`
- `glGenBuffers(1, &VBO); // creat one VBO`
- `glBindVertexArray(VAO); // active the VAO`
- `glBindBuffer(GL_ARRAY_BUFFER, VBO);`
- `glBufferData(GL_ARRAY_BUFFER, …)`

VAO and VBO in GLSL

- In the vertex shader
- `layout (location = 0) in vec3` position
  - `layout (location = 0)` describes how the vertex attribute and the buffer will be associated
  - `in` means this is an input variable (coming in form the OpenGL code)
  - `vec3` is a 3 element vector

```
//…
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, …);
```

Determining Locations After Linking

- Assumes you already know the variables' names

```
GLint loc = glGetAttribLocation( program,
“name” );

GLint loc = glGetUniformLocation( program,
“name” );
```

Initializing Uniform Variable Values

- Uniform Variables

```
glUniform4f( index, x, y, z, w );

GLboolean  transpose = GL_TRUE;

// Since we’re C programmers
GLfloat  mat[3][4][4] = { … };
glUniformMatrix4fv( index, 3, transpose, mat
);
```

Vertex Shader Examples

- A vertex shader is initiated by each vertex output by `glDrawArrays()`

- A vertex shader must output a position in clip coordinates to the rasterizer

- Basic uses of vertex shaders
  - Transformations
  - Lighting
  - Moving vertex positions

A first program

- Render a cube with colors at each vertex

- The example demonstrates:
  - initializing vertex data
  - organizing data for rendering
  - simple object modeling
    - building up 3D objects from geometric primitives
    - building geometric primitives from vertices

Initializing the Cube's Data

- Before we can initialize our VBO, we need to stage the data
- Our cube has two attributes per vertex
  - position
  - color
- We create two arrays to hold the VBO data

```
point4  vPositions[NumVertices];
color4  vColors[NumVertices];
```

Cube's Data

- Vertices of a unit cube centered at origin
  - sides aligned with axes

```
point4 positions[8] = {
    point4( -0.5, -0.5,  0.5, 1.0 ),
    point4( -0.5,  0.5,  0.5, 1.0 ),
    point4(  0.5,  0.5,  0.5, 1.0 ),
    point4(  0.5, -0.5,  0.5, 1.0 ),
    point4( -0.5, -0.5, -0.5, 1.0 ),
    point4( -0.5,  0.5, -0.5, 1.0 ),
    point4(  0.5,  0.5, -0.5, 1.0 ),
    point4(  0.5, -0.5, -0.5, 1.0 )
};
```

Cube data

- We’ll also set up an array of RGBA colors

```
color4 colors[8] = {
    color4( 0.0, 0.0, 0.0, 1.0 ),  // black
    color4( 1.0, 0.0, 0.0, 1.0 ),  // red
    color4( 1.0, 1.0, 0.0, 1.0 ),  // yellow
    color4( 0.0, 1.0, 0.0, 1.0 ),  // green
    color4( 0.0, 0.0, 1.0, 1.0 ),  // blue
    color4( 1.0, 0.0, 1.0, 1.0 ),  // magenta
    color4( 1.0, 1.0, 1.0, 1.0 ),  // white
    color4( 0.0, 1.0, 1.0, 1.0 )   // cyan
};
```

Generating a Cube Face from Vertices

- To simplify generating the geometry, we use a convenience function `quad()`
  - create two triangles for each face and assigns colors to the vertices

```
int Index = 0;  // global variable indexing into VBO arrays
void quad( int a, int b, int c, int d )
{
    vColors[Index] = colors[a]; vPositions[Index] = positions[a]; Index++;
    vColors[Index] = colors[b]; vPositions[Index] = positions[b]; Index++;
    vColors[Index] = colors[c]; vPositions[Index] = positions[c]; Index++;
    vColors[Index] = colors[a]; vPositions[Index] = positions[a]; Index++;
    vColors[Index] = colors[c]; vPositions[Index] = positions[c]; Index++;
    vColors[Index] = colors[d]; vPositions[Index] = positions[d]; Index++;
}
```

Generating the Cube from Faces

- Generate 12 triangles for the cube
  - 36 vertices with 36 colors

```
void colorcube()
{
    quad( 1, 0, 3, 2 );
    quad( 2, 3, 7, 6 );
    quad( 3, 0, 4, 7 );
    quad( 6, 5, 1, 2 );
    quad( 4, 5, 6, 7 );
    quad( 5, 4, 0, 1 );
}
```

Vertex Array Objects

- VAOs store the data of an geometric object
- Steps in using a VAO
  - generate VAO names by calling `glGenVertexArrays()`
  - bind a specific VAO for initialization by calling `glBindVertexArray()`
  - update VBOs associated with this VAO
  - bind VAO for use in rendering
- This approach allows a single function call to specify all the data for an objects
  - previously, you might have needed to make many calls to make all the data current

VAOs in code

- Create a vertex array object

```
GLuint vao;
glGenVertexArrays( 1, &vao );
glBindVertexArray( vao );
```

Storing Vertex Attributes

- Vertex data must be stored in a VBO, and associated with a VAO
- The code-flow is similar to configuring a VAO
  - generate VBO names by calling `glGenBuffers()`
  - bind a specific VBO for initialization by calling

    ```
    glBindBuffer( GL_ARRAY_BUFFER, … )
    ```

  - load data into VBO using

    ```
    glBufferData( GL_ARRAY_BUFFER, … )
    ```

  - bind VAO for use in rendering

    ```
    glBindVertexArray()
    ```

VBOs in Code

- Create and initialize a buffer object

```
GLuint buffer;
glGenBuffers( 1, &buffer );
glBindBuffer( GL_ARRAY_BUFFER, buffer );
glBufferData( GL_ARRAY_BUFFER,
              sizeof(vPositions) + sizeof(vColors),
              NULL, GL_STATIC_DRAW );
glBufferSubData( GL_ARRAY_BUFFER, 0,
                 sizeof(vPositions), vPositions );
glBufferSubData( GL_ARRAY_BUFFER, sizeof(vPositions),
                 sizeof(vColors), vColors );
```

Connecting Vertex Shaders with Geometric Data

- Application vertex data enters the OpenGL pipeline through the vertex shader
- Need to connect vertex data to shader variables
  - requires knowing the attribute location
- Attribute location can either be queried by calling `glGetVertexAttribLocation()`

Vertex Array Code

- Associate shader variables with vertex arrays
  - do this after shaders are loaded

```
GLuint vPosition =
   glGetAttribLocation( program, “vPosition" );
glEnableVertexAttribArray( vPosition );
glVertexAttribPointer( vPosition, 4, GL_FLOAT,
   GL_FALSE, 0,BUFFER_OFFSET(0) );

GLuint vColor =
   glGetAttribLocation( program,"vColor" );
glEnableVertexAttribArray( vColor );
glVertexAttribPointer( vColor, 4, GL_FLOAT,
  GL_FALSE, 0, BUFFER_OFFSET(sizeof(vPositions)) );
```

Drawing Geometric Primitives

- For contiguous groups of vertices

```
glDrawArrays( GL_TRIANGLES, 0, NumVertices );
```

- Usually invoked in display callback
- Initiates vertex shader

Finishing the Cube Program

```
int main( int argc, char **argv )
{
    glutInit( &argc, argv );
    glutInitDisplayMode( GLUT_RGBA | GLUT_DOUBLE |GLUT_DEPTH );
    glutInitWindowSize( 512, 512 );
    glutCreateWindow( "Color Cube" );

    glewInit();
    init();

    glutDisplayFunc( display );
    glutKeyboardFunc( keyboard );
    glutMainLoop();

    return 0;
}
```

Cube Program's GLUT Callbacks

```
void display( void )
{
    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT
);
    glDrawArrays( GL_TRIANGLES, 0, NumVertices );
    glutSwapBuffers();
}

void keyboard( unsigned char key, int x, int y )
{
    switch( key ) {
        case 033: case 'q': case 'Q':
            exit( EXIT_SUCCESS );
            break;
    }
}
```

Simple Vertex Shader for Cube Example

```
#version 430

in vec4 vPosition;
in vec4 vColor;

out vec4 color;

void main()
{
    color = vColor;
    gl_Position = vPosition;
}
```

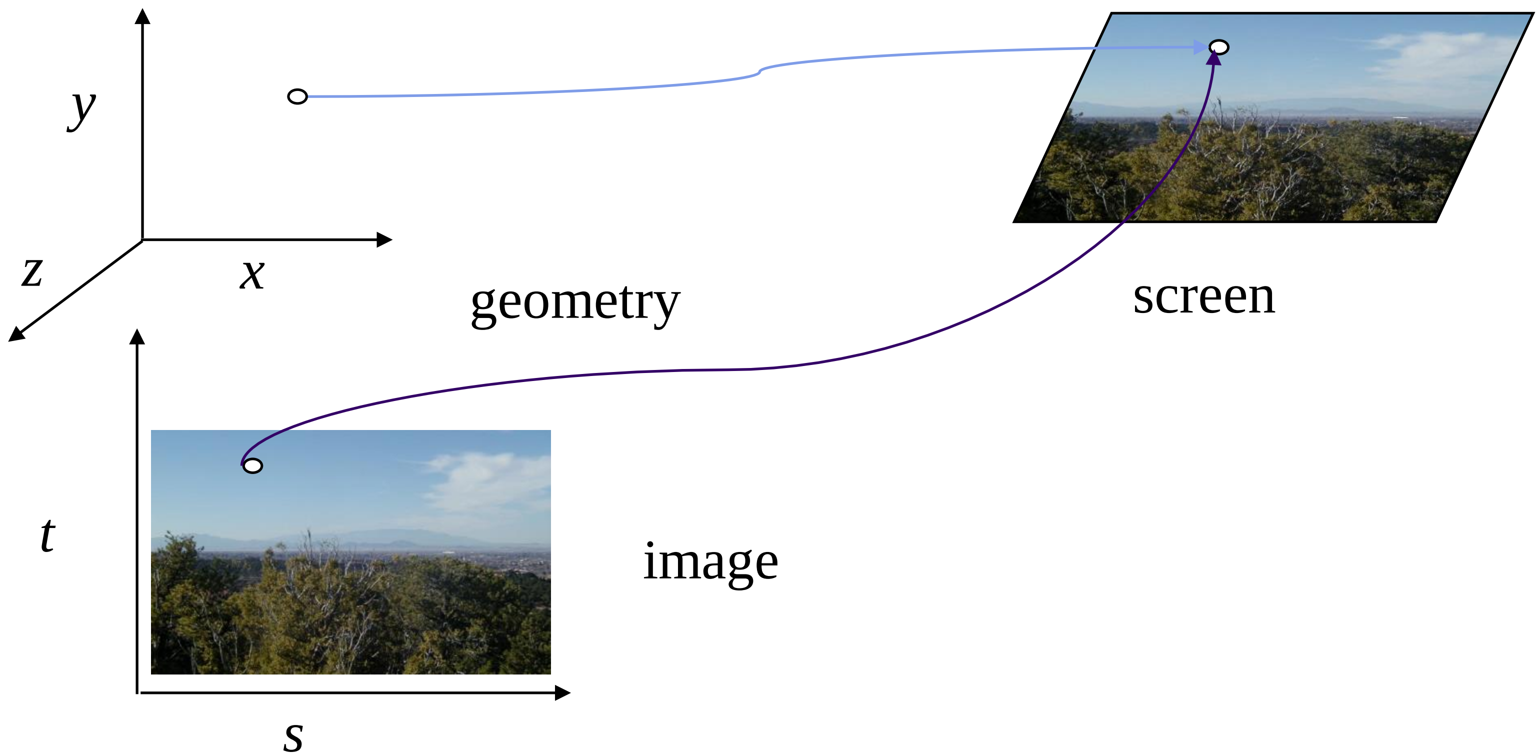
The Simplest Fragment Shader

```
#version 430

in vec4 color;

out vec4 fColor; // fragment’s final color

void main()
{
    fColor = color;
}
```

Fragment Shaders

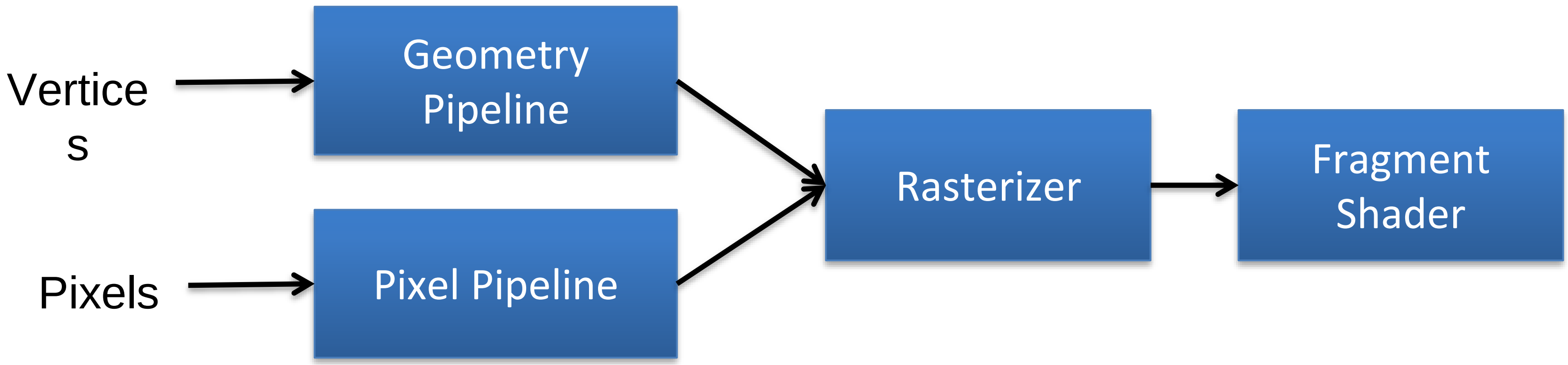
- A shader that's executed for each "potential" pixel
  - fragments still need to pass several tests before making it to the framebuffer
- There are lots of effects we can do in fragment shaders
  - Per-fragment lighting
  - Texture and bump Mapping
  - Environment (Reflection) Maps

Texture Mapping



Texture Mapping and the OpenGL Pipeline

- Images and geometry flow through separate pipelines that join at the rasterizer
  - "complex" textures do not affect geometric complexity



Applying Textures

- Three basic steps to applying a texture
  - specify the texture
    - read or generate image
    - assign to texture
    - enable texturing
  - assign texture coordinates to vertices
  - specify texture parameters
    - wrapping, filtering

Texture Objects

- Have OpenGL store your images
  - one image per texture object
  - may be shared by several graphics contexts

- Generate texture names

```
glGenTextures( n, *texIds );
```

Texture Objects

- Create texture objects with texture data and state

```
glBindTexture( target, id );
```

- Bind textures before using

```
glBindTexture( target, id );
```

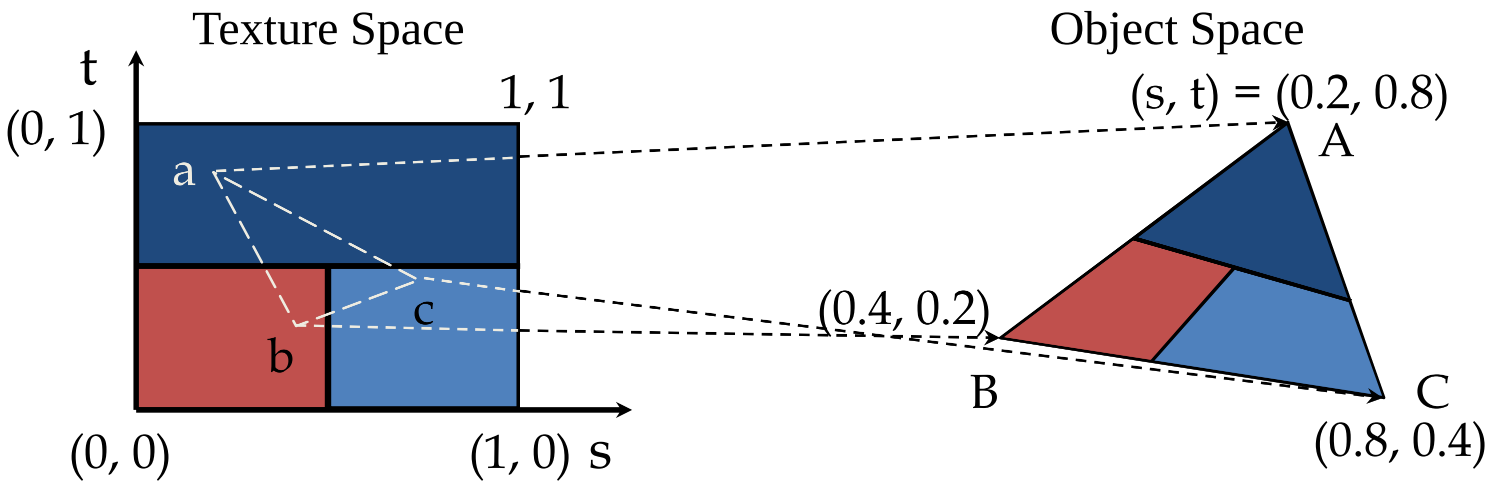
Specifying a Texture Image

- Define a texture image from an array of *texels* in CPU memory

```
glTexImage2D( target, level, components,
              w, h, border,
              format, type, *texels );
```

Mapping a Texture

- Based on parametric texture coordinates
- coordinates needs to be specified at each vertex



Applying the Texture in the Shader

```
in vec4 texCoord;

// Declare the sampler
uniform float     intensity;
uniform sampler2D diffuseMaterialTexture;

// Apply the material color
vec3 diffuse = intensity *
   texture(diffuseMaterialTexture,
texCoord).rgb;
```

Applying Texture to Cube

```
// add texture coordinate attribute  to quad function

quad( int a, int b, int c, int d )
{
    vColors[Index] = colors[a];
    vPositions[Index] = positions[a];
    vTexCoords[Index] = vec2( 0.0, 0.0 );
    Index++;

    vColors[Index] = colors[b];
    vPositions[Index] = positions[b];
    vTexCoords[Index] = vec2( 1.0, 0.0 );
    Index++;
    … // rest of vertices
}
```

Creating a Texture Image

```
// Create a checkerboard pattern
for ( int i = 0; i < 64; i++ ) {
    for ( int j = 0; j < 64; j++ ) {
        GLubyte c;
        c = ((i & 0x8 == 0) ^ (j & 0x8 == 0)) * 255;
        image[i][j][0]  = c;
        image[i][j][1]  = c;
        image[i][j][2]  = c;
        image2[i][j][0] = c;
        image2[i][j][1] = 0;
        image2[i][j][2] = c;
    }
}
```

Texture Object

```
GLuint textures[1];
glGenTextures( 1, textures );

glActiveTexture( GL_TEXTURE0 );
glBindTexture( GL_TEXTURE_2D, textures[0] );

glTexImage2D( GL_TEXTURE_2D, 0, GL_RGB, width, height,
              0, GL_RGB, GL_UNSIGNED_BYTE, image );

glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT );
glTexParameteri( GL_TEXTURE_2D,
                 GL_TEXTURE_MAG_FILTER, GL_NEAREST );
glTexParameteri( GL_TEXTURE_2D,
                 GL_TEXTURE_MIN_FILTER, GL_NEAREST );
```

Vertex Shader

```
in vec4 vPosition;
in vec4 vColor;
in vec2 vTexCoord;

out vec4 color;
out vec2 texCoord;

void main()
{
    color       = vColor;
    texCoord    = vTexCoord;
    gl_Position = vPosition;
}
```

Fragment Shader

```
in vec4 color;
in vec2 texCoord;

out vec4 fColor;

uniform sampler texture;

void main()
{
  fColor = color * texture( texture, texCoord
);
}
```

Shader sample one – ambient lighting

```
// Vertex Shader
in vec4 position

uniform mat4 mvpMat

void main() {
  gl_Position = mvpMat * position;
}


// Fragment Shader
out vec4 fragColor

void main() {
  fragColor = vec4(1.0, 0.0, 0.0, 1);
}
```

Gouraud Shading – Vertex Shader

```
layout(location = 0) in vec3 Position;
layout(location = 1) in vec3 Normal;

out vec3 colorres;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

uniform vec3 viewPos;
uniform vec3 lightPos;
uniform vec3 objectColor;
uniform vec3 lightColor;

void main(){
    //...
}
```

Gouraud Shading – Vertex Shader

```
layout(location = 0) in vec3 Position;
layout(location = 1) in vec3 Normal;

out vec3 colorres;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

uniform vec3 viewPos;
uniform vec3 lightPos;
uniform vec3 objectColor;
uniform vec3 lightColor;

void main(){
    //...
}
```

```
// 设置顶点属性指针
// Position
glEnableVertexAttribArray(0);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void *)0);

// Normal
glEnableVertexAttribArray(1);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex), (void *)offsetof(Vertex, Normal));
```

Gouraud Shading – Vertex Shader

```
layout(location = 0) in vec3 Position;
layout(location = 1) in vec3 Normal;

out vec3 colorres;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

uniform vec3 viewPos;
uniform vec3 lightPos;
uniform vec3 objectColor;
uniform vec3 lightColor;

void main(){
    //...
}
```

```
In Fragment shader:
   in vec3 colorres;
```

Gouraud Shading – Vertex Shader

```
layout(location = 0) in vec3 Position;
layout(location = 1) in vec3 Normal;

out vec3 colorres;
```

```
// ------------------------------------------------------------------------
void setMat4(const std::string &name, const glm::mat4 &mat) const
{
    glUniformMatrix4fv(glGetUniformLocation(ID, name.c_str()), 1, GL_FALSE, &mat[0][0]);
}
```

```
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

uniform vec3 viewPos;
uniform vec3 lightPos;
uniform vec3 objectColor;
uniform vec3 lightColor;

void main(){
    //...
}
```

Gouraud Shading – Vertex Shader

```
layout(location = 0) in vec3 Position;
layout(location = 1) in vec3 Normal;

out vec3 colorres;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

uniform vec3 viewPos;
uniform vec3 lightPos;
uniform vec3 objectColor;
uniform vec3 lightColor;

void main(){
    //...
}
```

```
// ------------------------------------------------------------------------
void setVec3(const std::string &name, const glm::vec3 &value) const
{
    glUniform3fv(glGetUniformLocation(ID, name.c_str()), 1, &value[0]);
}
void setVec3(const std::string &name, float x, float y, float z) const
{
    glUniform3f(glGetUniformLocation(ID, name.c_str()), x, y, z);
}
```

Gouraud Shading – Vertex Shader

```
void main(){
    gl_Position = projection * view * model * vec4(Position, 1.0f);
    outNormal =mat3(transpose(inverse(model))) * Normal;
    Pos = vec3(model * vec4(Position, 1.0))

    //环境光
    float ambientStrength = 0.1;
    vec3 ambient = ambientStrength * lightColor;

    //漫反射
    vec3 norm = normalize(outNormal);
    vec3 lightDir = normalize(lightPos - Pos);
    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = diff * lightColor;

    //镜面高光
    float specularStrength = 0.7;
    vec3 viewDir = normalize(viewPos - FragPos);
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
    vec3 specular = specularStrength * spec * lightColor;

    colorres = (ambient + diffuse + specular) * objectColor;
}
```

Gouraud Shading – Fragment Shader

```
out vec4 FragColor;

in vec3 colorres;

void main(){

    FragColor = vec4(colorres, 1.0);
}
```

Coming from Vertex shader:
`in vec3 colorres`; // Rasterization Interpolation

Phong Shading – Vertex Shader

```
layout(location = 0) in vec3 Position;
layout(location = 1) in vec3 Normal;

out vec3 outNormal;
out vec3 FragPos;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main(){

    gl_Position = projection * view * model * vec4(Position, 1.0f);

    FragPos = vec3(model * vec4(Position, 1.0));

    outNormal =mat3(transpose(inverse(model))) * Normal;
}
```

Phong Shading – Fragment Shader

```
out vec4 FragColor

in vec3 outNormal;
in vec3 FragPos;

uniform vec3 viewPos;
uniform vec3 lightPos;
uniform vec3 objectColor;
uniform vec3 lightColor;

void main(){
    //...
}
```

Phong Shading – Fragment Shader

```
void main(){
    //环境光
    float ambientStrength = 0.1;
    vec3 ambient = ambientStrength * lightColor;

    //漫反射
    vec3 norm = normalize(outNormal);
    vec3 lightDir = normalize(lightPos - FragPos);
    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = diff * lightColor;

    //镜面高光
    float specularStrength = 0.7;
    vec3 viewDir = normalize(viewPos - FragPos);
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), 32);
    vec3 specular = specularStrength * spec * lightColor;

    vec3 result = (ambient + diffuse + specular) * objectColor;
    FragColor = vec4(result, 1.0);
}
```

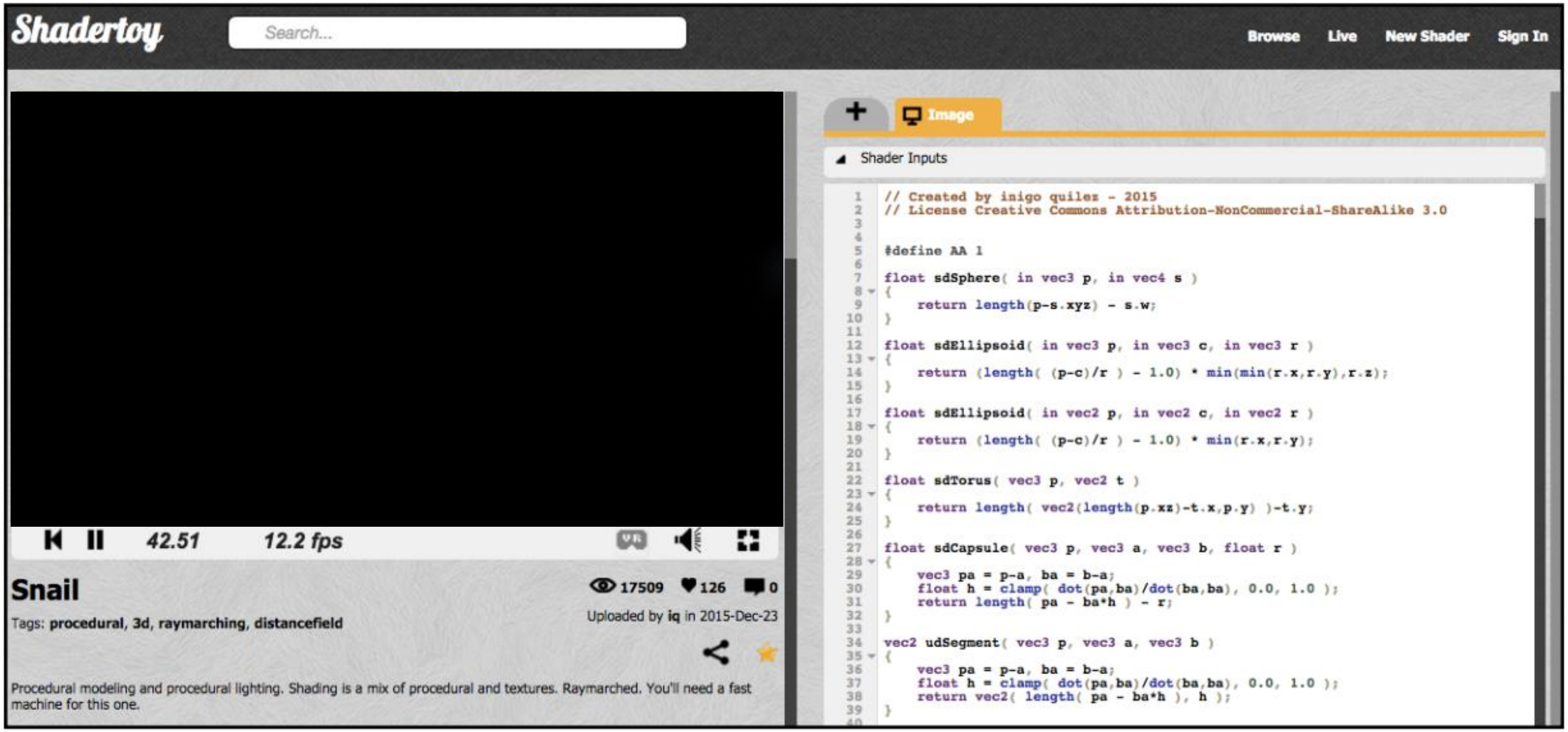
Further Reading

- GLSL tutorial: https://www.opengl.org/sdk/docs/tutorials/TyphoonLabs/

- summary of built-in GLSL functions
  - https://shaderific.com/glsl.html

- GLSL https://www.khronos.org/files/opengles_shading_language.pdf

Snail Shader Program



- Inigo Quilez
  - Procedurally modeled, 800 line shader
  - https://www.shadertoy.com/view/ld3Gz2

Goal: Highly Complex 3D Scenes in Realtime

Unreal Engine Kite Demo (Epic Games 2015)

谢谢

北京航空航天大學
人工智能研究院