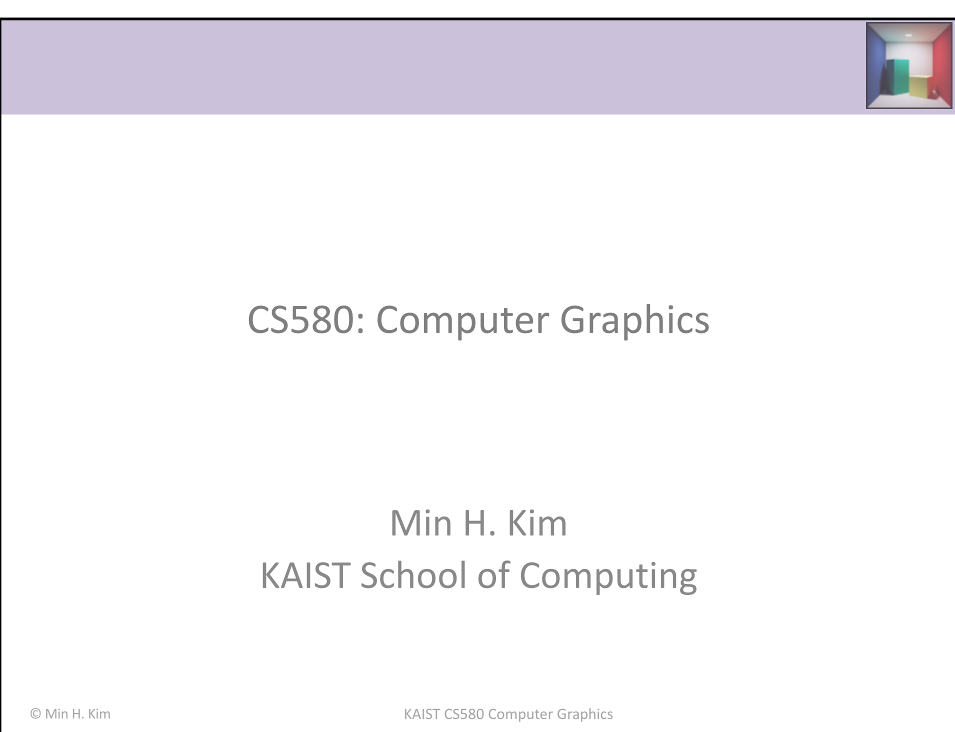
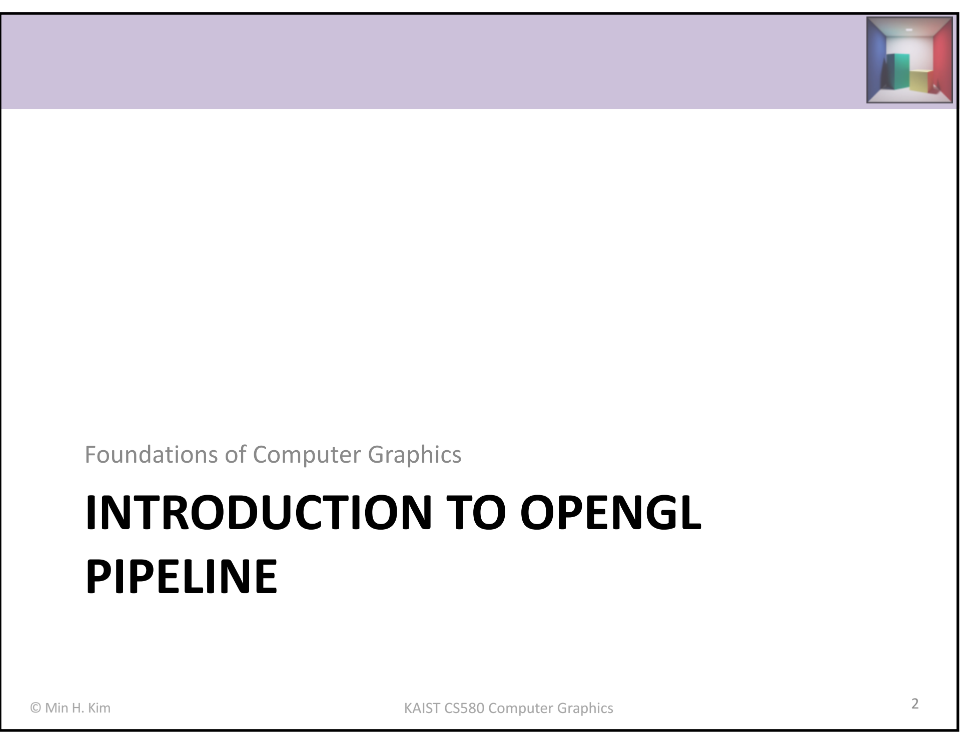
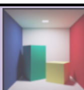

CS580: Computer Graphics
Min H. Kim
KAIST School of Computing
© Min H. Kim
KAIST CS580 Computer Graphics

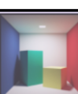

Foundations of Computer Graphics
INTRODUCTION TO OPENGL PIPELINE
© Min H. Kim
KAIST CS580 Computer Graphics
2

What is OpenGL?



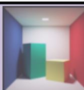
- OpenGL = Open Graphics Library
- An open industry-standard API for hardware accelerated graphics drawing
- Implemented by graphics-card vendors
- Maintained by the Khronos-Group

What is OpenGL?



- Pros:
  + Full specification freely available
  + Everyone can use it
  + Can use it anywhere (Windows, Linux, Mac, BSD, Mobile phones, Web-pages, ...)
  + Long-term maintenance for older applications
  + New functionality usually earlier available through Extensions
- Cons:
  - Inclusion of Extensions to core may take longer
  - Game-Industry

Setup OpenGL Project



- Include OpenGL-header:

  `#include <GL/gl.h> // basic OpenGL`

- Link OpenGL-library “opengl32.lib”
- If needed, also link other libraries (esp. GLEW, see later!).

OpenGL in more detail

- OpenGL-functions prefixed with “gl”:

  `glFunction{1234}{bsifd...}{v}(T arg1, T arg2, ...);`

  Example: `glDrawArrays(GL_TRIANGLES, 0, vertexCount);`

- OpenGL-constants prefixed with “GL_”:

  GL_SOME_CONSTANT

  Example: GL_TRIANGLES

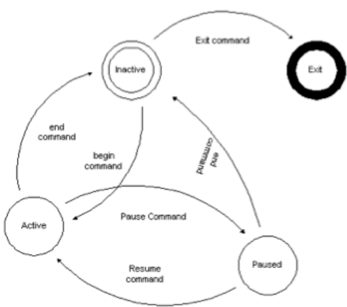
- OpenGL-types prefixed with “GL”:

  GLtype

  Example: GLfloat

OpenGL in more detail

- OpenGL is a state-machine
- Remember state-machines:
  - Once a state is set, it remains active until the state is changed to something else via a transition.
  - A transition in OpenGL equals a function-call.
  - A state in OpenGL is defined by the OpenGL-objects which are current.



© Min H. Kim KAIST CS580 Computer Graphics 7

OpenGL in more detail

- Set OpenGL-states:

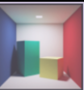
  glEnable(...);
  glDisable(...);
  gl*(...); // several call depending on purpose

- Query OpenGL-states with Get-Methods:

  glGet*(...); // several calls available, depending on what to query

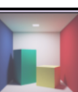
© Min H. Kim KAIST CS580 Computer Graphics 8

OpenGL 2.1



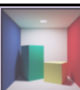
- Released in August 2006
- Fully supported “fixed function” (FF)
- GLSL-Shaders supported as well
- Mix of FF and shaders was possible, which could get confusing or clumsy quickly in bigger applications
- Supported by all graphics-drivers

No-FF in OpenGL 2.1



- Do NOT use the following in OpenGL 2.1:
  - Built-in matrix-functions/stacks:
    - glMatrixMode, glMult/LoadMatrix,
    - glRotate/Translate/Scale, glPush/PopMatrix...
  - Immediate mode:
    - glBegin/End, glVertex, glTexCoords...
  - Material and lighting:
    - glLight, glMaterial, ...
  - Attribute-stack:
    - glPush/PopAttrib, ...
  - some primitive modes:
    - GL_QUAD*, GL_POLYGON

No-FF in OpenGL 2.1



- Do NOT use the following in GLSL 1.1/1.2:
  - ftransform()
  - All built-in gl_*-variables, except:
    - gl_Position in vertex-shader
    - gl_FragColor, gl_FragData[] in fragment-shader
- The list may not be complete!

GLEW

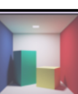
- On Windows only OpenGL 1.1 supported natively.
- To use newer OpenGL versions, each additional function, i.e., all extensions (currently ~1900), must be loaded manually!
- →Lots of work! Therefore:
- Use GLEW = OpenGL Extension Wrangler

GLEW

- Include it in your program and initialize it:

```
#include <GL/glew.h> // include before other GL headers!
// #include <GL/gl.h> included with GLEW already

void initGLEW() {
    GLenum err = glewInit(); // initialize GLEW
    if (err != GLEW_OK) // check for error {
        cout << "GLEW Error: " << glewGetErrorString(err);
        exit(1);
    }
}
```


GLEW



- Check for supported OpenGL version:

```
if (glewIsSupported("GL_VERSION_3_2")) {
  // OpenGL 3.2 supported on this system
}
```

- To check for a specific extension:

```
if (GLEW_ARB_geometry_shader4) {
  // Geometry-Shader supported on this system
}
```

GLEW

- How to use:
  - Include GLEW-header:

```
#include <GL/glew.h> // GLEW
```

  - Link OpenGL-library “opengl32.lib” and “glew32.lib”
  - Copy “glew32.dll” to bin folder
  - You’re ready to go.

OpenGL Program Skeleton

- Typical OpenGL-Application:

Start Application
main()
init()
update()
Main-loop
render()
cleanup()
Exit application

OpenGL Program Skeleton

- main():
  - Program-Entry
  - Create window
  - Call init()
  - Start main window-loop Call cleanup()
  - Exit application
- init():
  - Initialize libraries, load config-files, ...
  - Allocate resources, preprocessing, ...

© Min H. Kim

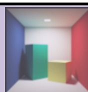
OpenGL Program Skeleton

- Example init()-function:

```
void init() {
        // Create and initialize a window with depth-buffer and
        // double- buffering. See your window-managers documentation.

        // enable the depth-buffer in OpenGL
        glEnable(GL_DEPTH_TEST);

        // enable back-face culling in OpenGL
        glEnable(GL_CULL_FACE);

        // define a clear color
        glClearColor(0.0f, 0.0f, 0.0f, 0.0f);

        // set the OpenGL-viewport
        glViewPort(0, 0, windowWidth, windowHeight);

        // Do other useful things
}
```

© Min H. Kim

OpenGL Program Skeleton

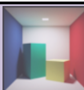


- update():
  - Handle user-input, update game-logic, ...
- render():
  - Do actual rendering of graphics here!
  - Note: Calling render() twice without calling update() in between should result in the same rendered image!
- cleanup():
  - Free all resources

OpenGL Program Skeleton

- The geometry of a 3D-object is stored in an array of vertices called Vertex-Array.
- Each vertex can have so called Attributes, like a Normal Vector and Texture-Coordinates.
- OpenGL also treats vertices as attributes!
- To render geometry in OpenGL, vertex-(attribute)-arrays are passed to OpenGL and then rendered.

OpenGL Program Skeleton



- To do so:
  - Query the attribute-location in the shader:

    ```
    GLint vertexLocation =
    glGetAttribLocation(myShaderProgram, "in_Position");
    ```
  - Enable an array for the vertex-attribute:

    ```
    glEnableVertexAttribArray(vertexLocation);
    ```
  - Then tell OpenGL which data to use:

    ```
    glVertexAttribPointer(vertexLocation, 3, GL_FLOAT,
    GL_FALSE, 0, myVertexArray);
    ```

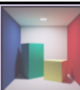

OpenGL Program Skeleton

- Draw ("render") the arrays:

  ```
  glDrawArrays(GL_TRIANGLES, 0, 3);
  // this does the actual draw
  ```
- Finally disable the attribute-array:

  ```
  glDisableVertexAttribArray(vertexLocation);
  ```

OpenGL Program Skeleton



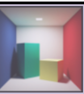
- Example render()-function:

```
 // triangle data
static GLfloat vertices[] = { -0.5, -0.333333, 0, // x1, y1, z1
                              +0.5, -0.333333, 0, // x2, y2, z2
                              +0.0, +0.666666, 0}; // x3, y3, z3
...
void render() {
    // clear the color-buffer and the depth-buffer
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // activate a shader program
    glUseProgram(myShaderProgram);

    // Find the attributes
    GLint vertexLocation = glGetAttribLocation(
        myShaderProgram, "in_Position");
```


OpenGL Program Skeleton

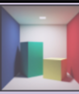
```
    // enable vertex attribute array for this attribute
    glEnableVertexAttribArray(vertexLocation);

    // set attribute pointer
    glVertexAttribPointer(vertexLocation, 3, GL_FLOAT,
        GL_FALSE, 0, vertices);

    // Draw ("render") the triangle
    glDrawArrays(GL_TRIANGLES, 0, 3);

    // Done with rendering. Disable vertex attribute array
    glDisableVertexAttribArray(vertexLocation);

    // disable shader program
    glUseProgram(0);

    // Swap buffers
}
```

OpenGL-Object life-cycle

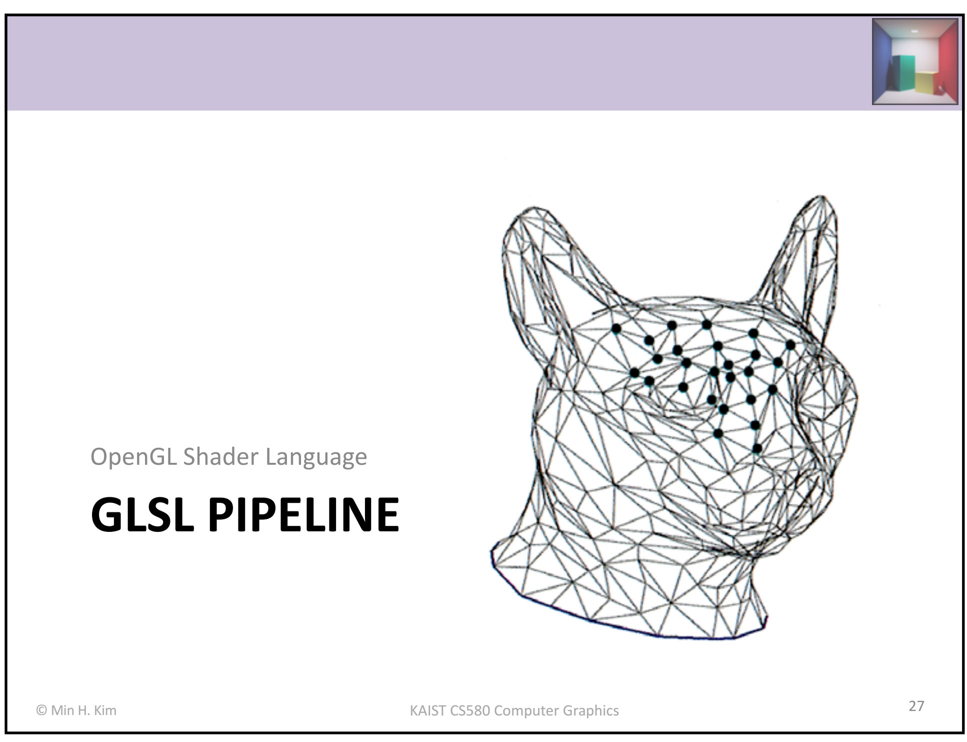


- In OpenGL, all objects, like buffers and textures, are somehow treated the same way.
- On object creation and initialization:
  - First, create a handle to the object (in OpenGL often called a name). Do this ONCE for each object.
  - Then, bind the object to make it current.
  - Pass data to OpenGL. As long as the data does not change, you only have to do this ONCE.
  - Unbind the object if not used.

OpenGL-Object life-cycle

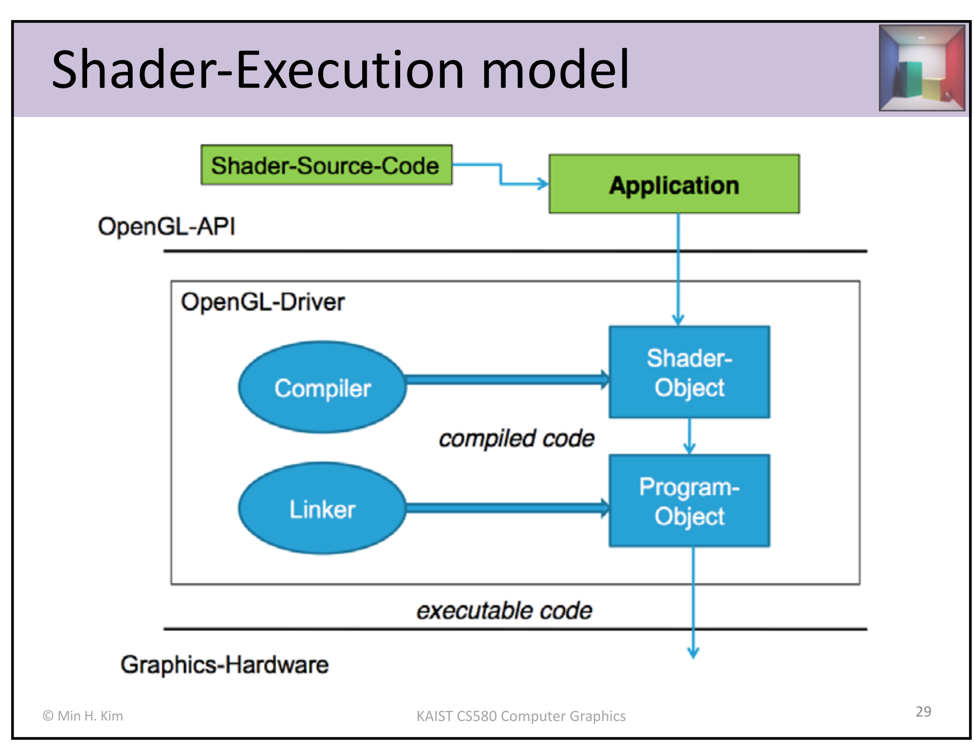


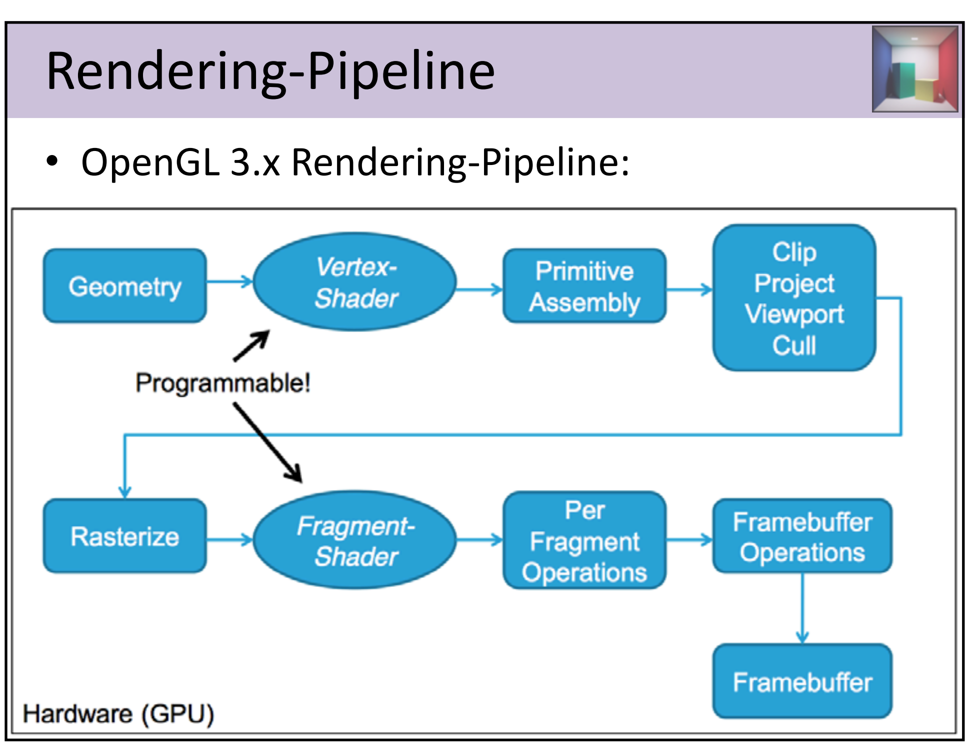
- On rendering, or whenever the object is used:
  - Bind it to make it current.
  - Use it.
  - Unbind it.
- Finally, when object is not needed anymore:
  - Delete object.
  - Note that in some cases you manually have to delete attached resources!
- NOTE: OpenGL-objects are NOT objects in an OOP-sense!

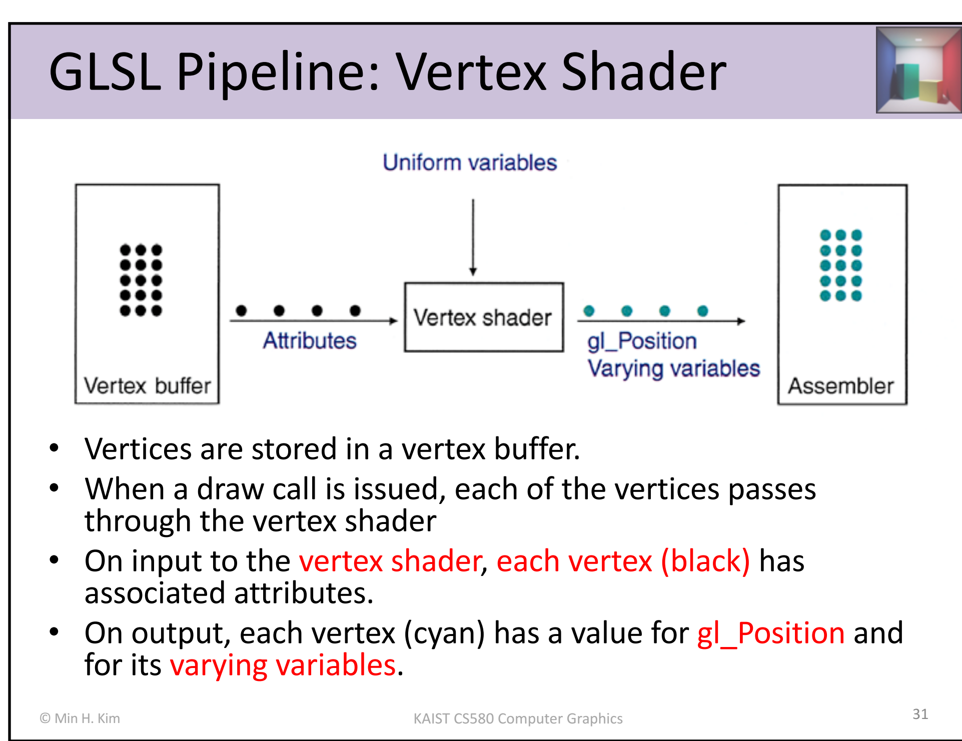


What shaders are

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


Shader-Execution model
Shader-Source-Code
Application
OpenGL-API
OpenGL-Driver
Compiler
Shader-Object
compiled code
Linker
Program-Object
executable code
Graphics-Hardware
© Min H. Kim
KAIST CS580 Computer Graphics
29


Rendering-Pipeline
• OpenGL 3.x Rendering-Pipeline:
Geometry
Vertex-Shader
Primitive Assembly
Clip Project Viewport Cull
Programmable!
Rasterize
Fragment-Shader
Per Fragment Operations
Framebuffer Operations
Framebuffer
Hardware (GPU)



GLSL Pipeline: Rasterization

Assembler

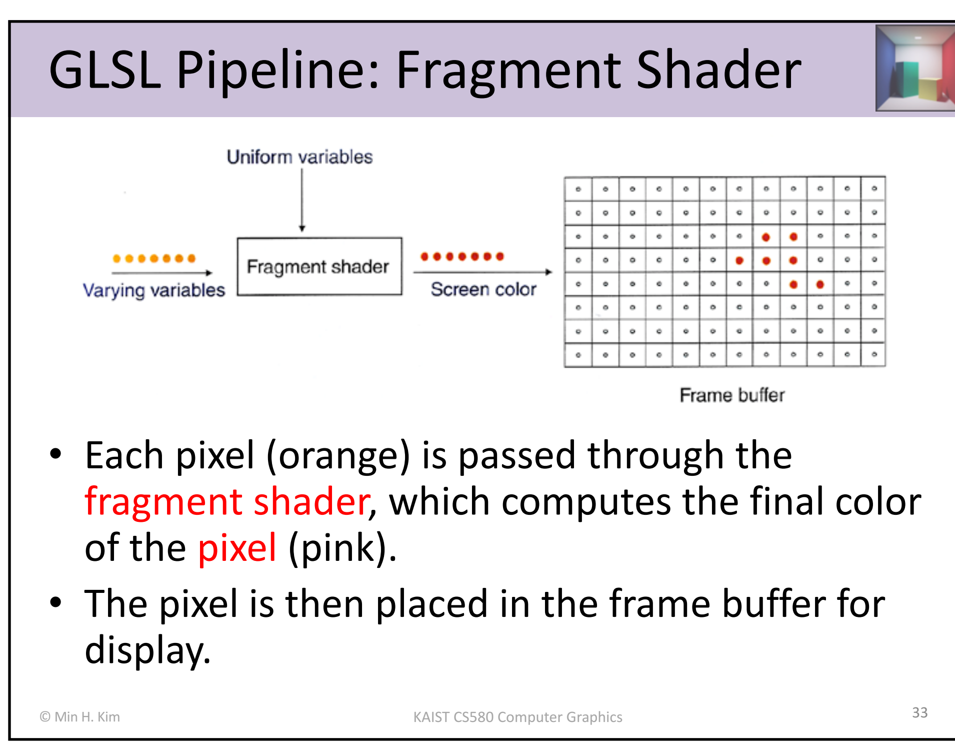
gl_Position
Varying variables

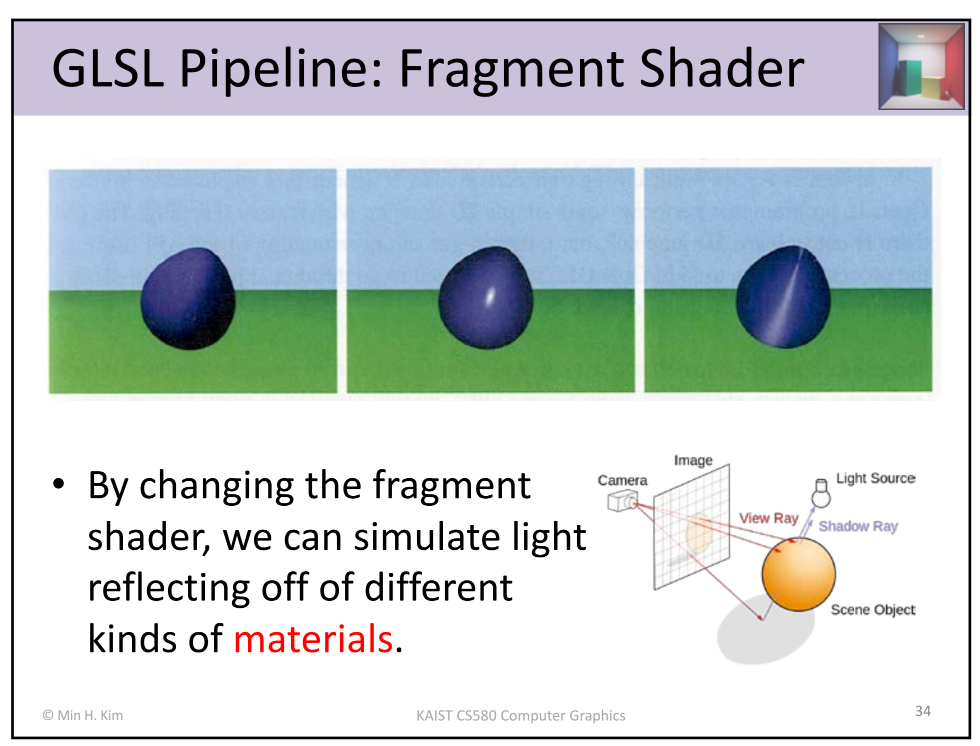
Varying variables

Rasterizer

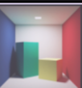
- The data in gl_Position is used to place the three vertices of the triangle on a virtual screen.
- The rasterizer figures out which pixels (orange) are inside the triangle and interpolates the varying variables from the vertices to each of these pixels.

© Min H. Kim


GLSL Pipeline: Fragment Shader
Uniform variables
Fragment shader
Varying variables
Screen color
Frame buffer
Each pixel (orange) is passed through the fragment shader, which computes the final color of the pixel (pink).
The pixel is then placed in the frame buffer for display.
© Min H. Kim
KAIST CS580 Computer Graphics
33


GLSL Pipeline: Fragment Shader
By changing the fragment shader, we can simulate light reflecting off of different kinds of materials.
Camera
Image
Light Source
View Ray
Shadow Ray
Scene Object
© Min H. Kim
KAIST CS580 Computer Graphics
34

GLSL Pipeline



- Remember:
  - The Vertex-Shader is executed ONCE per each vertex!
  - The Fragment-Shader is executed ONCE per rasterized fragment (~ a pixel)!
- A Shader-Program consists of both,
  - One VS
  - One FS

Setting up shaders and programs

- Compile shaders:

```
char* shaderSource; // contains shadersource
int shaderHandle = glCreateShader(GL_SHADER_TYPE);
// shader-types: vertex || geometry || fragment
glShaderSource(shaderHandle, 1, shaderSource, NULL);
glCompileShader(shaderHandle);
```

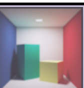
- Create program and attach shaders to it:

```
int programHandle = glCreateProgram();
glAttachShader(programHandle, shaderHandle);
// do this for vertex AND fragment-shader (AND geometry
if needed)!
```

- Finally link program:

```
glLinkProgram(programHandle);
```

Enabling shaders



- Enable a GLSL program:

```
glUseProgram(programHandle);
// shader-program now active
```

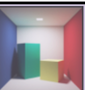
  - The active shader-program will be used until glUseProgram() is called again with another program-handle.
  - Call of glUseProgram(0) sets no program active (undefined state!).

Basic shader layout

- Shader-Programs must have a main()-method
- Vertex-Shader outputs to at least gl_Position
- Fragment-Shader to custom defined output

```
//preprocessor directives like:
#version 150

//variable declarations

void main() {
    //do something and write into output variables
}
```

Shader Parameter



- Shader variable examples:

```
uniform mat4 projMatrix; // uniform input
in vec4 vertex; // attribut-input
out vec3 fragColor; // shader output
```

- Three types:
  - uniform: does not change per primitive; read-only in shaders
  - in: VS: input changes per vertex, read-only; FS: interpolated input; read-only
  - out: shader-output; VS to FS; FS output.

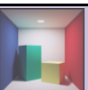
Uniform Parameter

- Set uniform parameters in an application:

```
// first get location
projMtxLoc = glGetUniformLocation(programHandle,
  "projMatrix");

// then set current value
glUniformMatrix4fv(projMtxLoc, 1, GL_FALSE,
currentProjectionMatrix);
```

  - First get the “location” of the uniform-variable
  - Then set the current value
  - Can pass values to vertex- and fragment-shader

Attribute Parameter



- A vertex can have attributes like a normal-vector or texture-coordinates
- OpenGL also treats the vertex itself as an attribute
- We want to access our current vertex within our vertex-shader (as we used to do with gl_Vertex in former GLSL-versions):
- Therefore, we declare in our vertex-shader:

```
in vec4 vertex; // vertex attribut
```


Attribute Parameter

- Now, there are two ways to pass data to this shader attribute-variable, depending on:
  – if you just have an array of vertices (Vertex Array),
  – or an VBO (Vertex Buffer Object).
- To do so: Query shader-variable location
  – Enable vertex-attribute array
  – Set pointer to array
  – Draw and disable array

Attribute Parameter: Vertex-Array

- For a Vertex-Array, pass data like this:

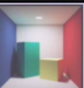
```
// first get the attribute-location
vertexLocation = glGetAttribLocation(programHandle, "vertex");

// enable an array for the attribute
glEnableVertexAttribArray(vertexLocation);

// set attribute pointer
glVertexAttribPointer(vertexLocation, 3, GL_FLOAT, GL_FALSE, 0,
myVertexArray);

// Draw ("render") the triangle
glDrawArrays(GL_TRIANGLES, 0, 3);

// Done with rendering. Disable vertex attribute array
glDisableVertexAttribArray(vertexLocation);
```


Attribute Parameter: VBO

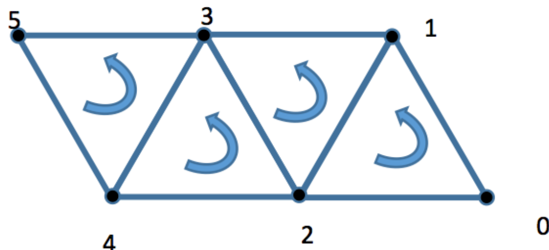
- Setting attribute parameters with VBO:

```
// first get location
vertexLocation = glGetAttribLocation(programHandle, "vertex");

// activate desired VBO
glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);

// set attribute-pointer
glVertexAttribPointer(vertexLocation, 4, GL_FLOAT, GL_FALSE, 0,
0);

// finally enable attribute-array
glEnableVertexAttribArray(vertexLocation); ...
```

Vertices in OpenGL



- OpenGL Order: (0, 1, 2), (2, 1, 3), (2, 3, 4), ...
- Consist orientation of the triangles generated.



Vertex Shader

- Typical use of the v-shader:
  – to determine the final position of the vertices on the screen
- Input: the attribute variables (position, color, text. coor.)
- Output: the reserved output variable, gl_Position
- The x and y coordinates of this variable are interpolated as positions within the drawing window.

Vertex Shader

```
#version 130

uniform float uVertexScale;

in vec2 aPosition;
in vec3 aColor;
in vec2 aTexCoord0, aTexCoord1;

out vec3 vColor;
out vec2 vTexCoord0, vTexCoord1;

void main() {
 gl_Position = vec4(aPosition.x * uVertexScale, aPosition.y, 0,1);
 vColor = aColor;
 vTexCoord0 = aTexCoord0;
 vTexCoord1 = aTexCoord1;
}
```

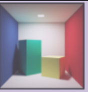

Fragment Output

- Since GLSL 1.3, gl_FragColor is deprecated.
- Therefore, need to define output on our own.
- Declare output variable in FS:

  `out vec4 fragColor; // fragment color output`
- In the application, before linking the shader-program with glLinkProgram(), bind the FS-output:

  `glBindFragDataLocation(programHandle, 0, "fragColor");`
- Finally assign a value to fragColor in the FS.

Example usage



- An application using shaders could basically look like this:

```
//Load shader and initialize parameter-handles

//Do some useful stuff like binding texture, activate
//texture-units, calculate and update matrices, etc.

glUseProgram(programHandle);

//Set shader-parameters
//Draw geometry

glUseProgram(anotherProgramHandle);
...
```


Fragment Shader

- After rasterization (it's not programmable)
- Input: interpolated data of varying variables
- Output: color values in the framebuffer (a part of GPU memory)
- Control the material and geometric properties of the material at that pixel

Fragment Shader

```
#version 130

uniform float uVertexScale;
uniform sampler2D uTexUnit0, uTexUnit1;

in vec2 vTexCoord0, vTexCoord1;
in vec3 vColor;

out vec4 fragColor;

void main(void) {
  vec4 color = vec4(vColor.x, vColor.y, vColor.z, 1);
  vec4 texColor0 = texture(uTexUnit0, vTexCoord0);
  vec4 texColor1 = texture(uTexUnit1, vTexCoord1);

  float lerper = clamp(.5 *uVertexScale, 0., 1.);
  float lerper2 = clamp(.5 * uVertexScale + 1.0, 0.0, 1.0);

  fragColor = ((lerper)*texColor1 + (1.0-lerper)*texColor0) * lerper2 + color * (1.0-lerper2);
}
```

As varying variables, the texture coordinates are interpolated from their vertex values to appropriate values at each pixel.

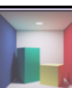
Texture

- auxiliary image data
- read from a file, loaded to openGL, used in fragment shader
- initTextures
- glTexture is a wrapper around a texture handle
- loadTexture
- – reads from file, turns on any “texture unit”, turns on a texture, passes the data.
- binds each texture to a unit (we have 2 here)
- we will send the “texture unit” info as a uniform (see display)

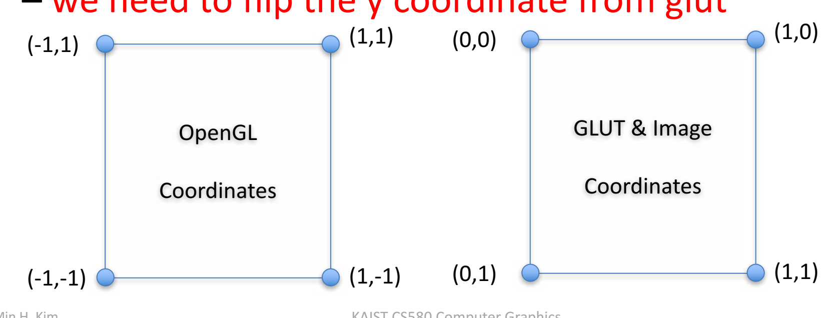
Texture Mapping

- A simple geometric object described by a small number of triangles.
- An auxiliary image called a texture.
- Parts of the texture are glued onto each triangle giving a more complicated appearance.

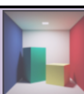
Interaction: Mouse



- callback
- called (due to our registration) whenever the mouse
- is clicked down or up
- we store this information
  – we need to flip the y coordinate from glut

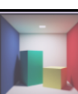


Double buffering



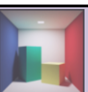
- The monitor displays one image at a time
- So if we render the next image to screen, then rendered primitives pop up
- And even worse, we often clear the screen before generating a new image
- A better solution is ”double buffering”

Double buffering



- Use two buffers: one front and one back
- The front buffer is displayed
- The back buffer is rendered to
- When new image has been created in back buffer, swap front and back

Interaction: Motion



- callback
- called whenever the mouse is moved
- here is where we update objectScale
- then we call glutPostRedisplay so that glut will know to trigger a display callback.
- see display for use of scale
- see vertex shader for use of scale

OpenGL ES 2.0 and 3.0

- a subset of Desktop OpenGL.
- there is no legacy fixed function pipeline in ES.
  - Such as glVertex, glColor, glNormal, glLight, glPushMatrix, glPopMatrix, glMatrixMode, etc...
- OpenGL ES 2.0 and 3.0 are just plain shaders.
- No "3d" is provided for you.
- You're required to write all projection, lighting, texture references, etc yourself.