CS300 OpenGL

State of art

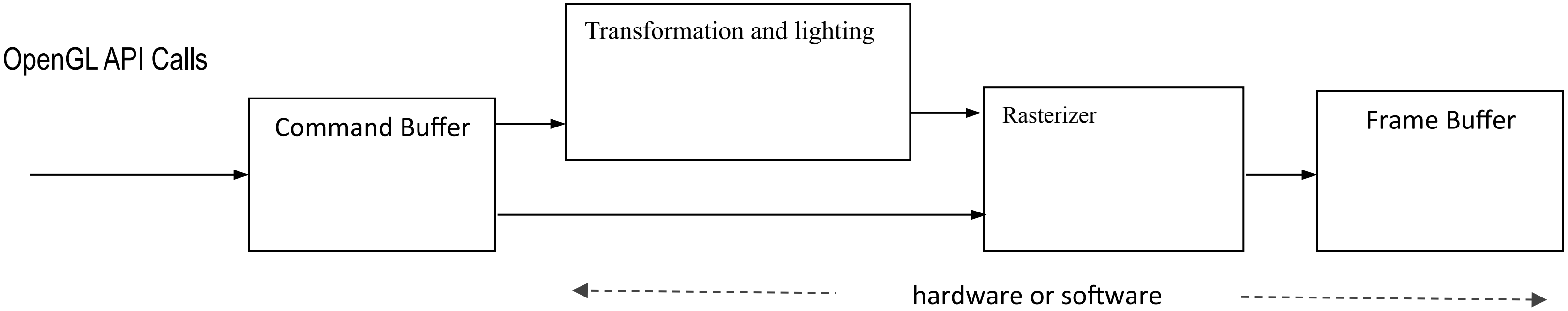
OpenGL Pipeline

- Implementation dependent
- API commands and data are placed in command buffer.
- Command buffer is flushed (software or hardware)
- Commands and data are passed to next stage
- In some implementations, the computer that runs and issues OpenGL commands is not the same computer that does the rendering.

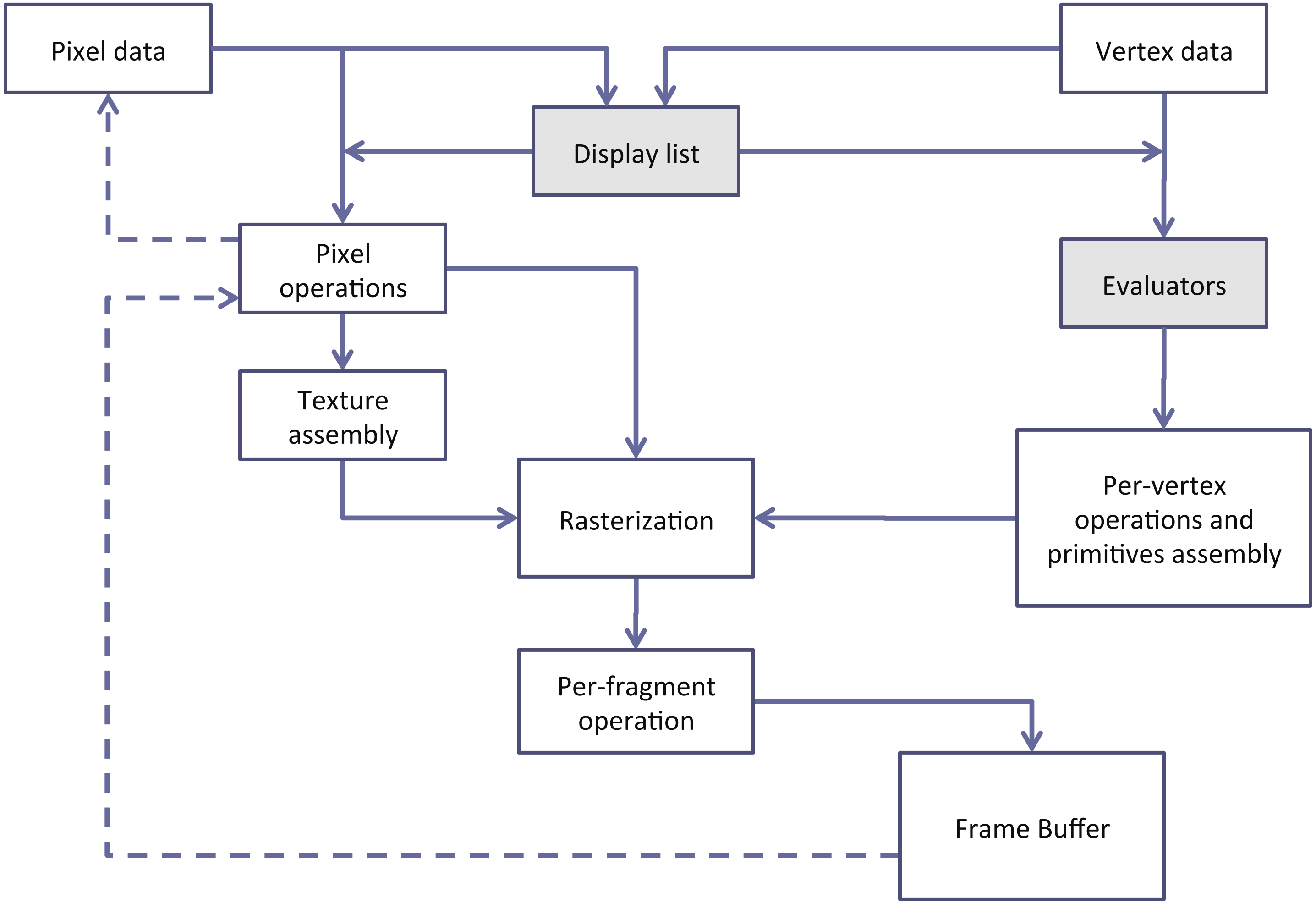
OpenGL Pipeline – version agnostic!

1. Vertices are transformed and lit
2. Processed vertices are passed to Rasterizer (review the graph below) to generate image from geometry, color and texture data
3. Pixels are placed in the frame buffer waiting for display to sync the next refresh cycle

Simplified OpenGL Graphics Pipeline

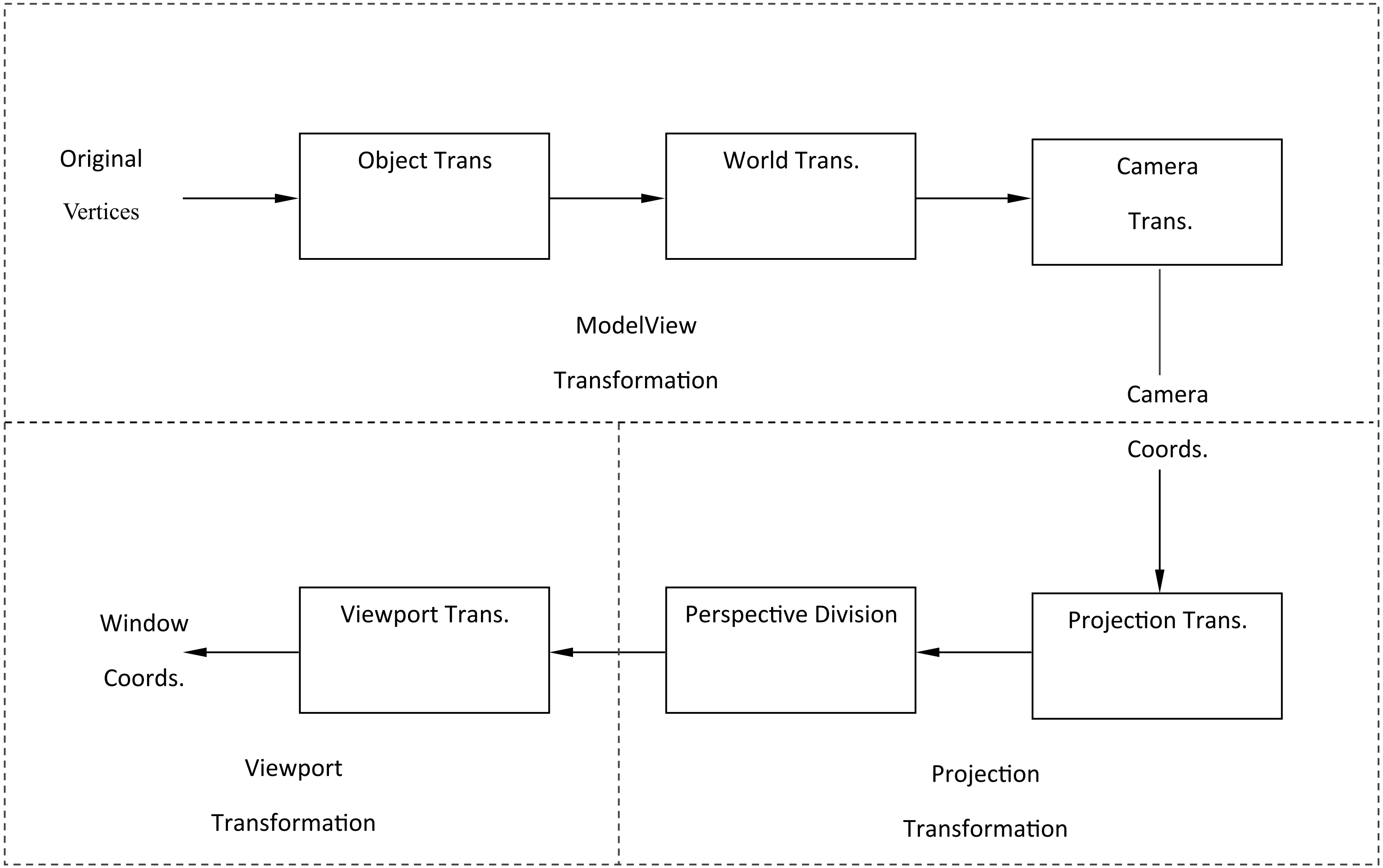


OpenGL Pipeline – version 2.1



OpenGL Transformation Pipeline

- In OpenGL, all vertices will undergo the following transformation:
  - ModelView transformation to move the vertices from model space to view space.
  - Projection transformation to move the vertices from the view space to the normalized projection space.
  - Viewport transformation to move the vertices from the NDC to the viewport (screen) space.
- Notice that the world space is 'skipped'.



Original Vertices

Object Trans

World Trans.

Camera Trans.

ModelView Transformation

Camera Coords.

Projection Trans.

Perspective Division

Viewport Trans.

Window Coords.

Viewport Transformation

Projection Transformation

OpenGL

- Open specification
  - Can be implemented by anybody
  - “Design once, use everywhere”
  - Newer functionality exposed via hardware extensions
- Transition from extension to core functionality may take longer

OpenGL Implementations

- Windows : opengl32.lib
- Linux : libopengl32.so
- Mac OS X: OpenGL Framework

- Also – hardware dependent libraries
  - E.g. NVIDIA display drivers are encapsulated in "nvoglnt32/64.dll" on Windows

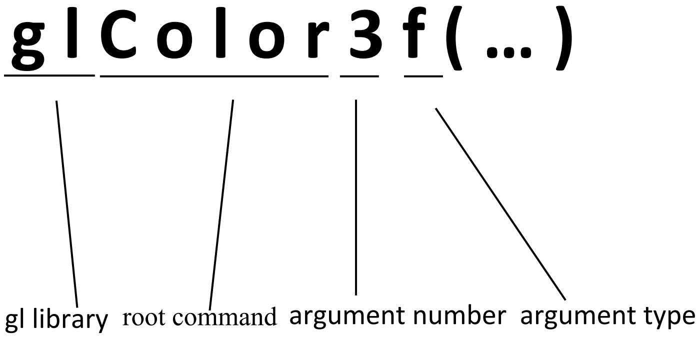
OpenGL Data Type & Function Naming Convention

- Data types and function prototypes are defined in
  - Windows/Linux : “#include<GL/GL.h>”
  - Mac OS X : “#include <OpenGL/OpenGL.h>
- OpenGL function naming convention takes the following format:

```
[library prefix][root command][argument
  count][argument type]
```

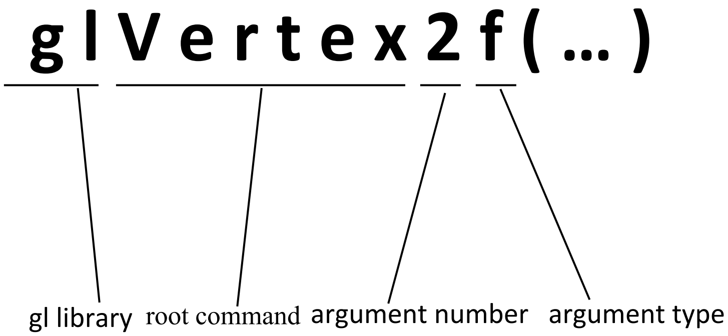
OpenGL Data Type & Function Naming Convention

- Examples:



glColor3f(…)

gl library root command argument number argument type



glVertex2f(…)

gl library root command argument number argument type

OpenGL Naming Conventions

- Constants also prefixed with “GL_”
  - GL_TRIANGLES
- Data types prefixed with “GL”
  - GLfloat, GLbyte
- Standard documentation removes these prefixes for readability
  - “DrawArrays” is actually “glDrawArrays”

OpenGL States

- State Machine architecture
  - Once a state is set, it remains active until explicitly reset/changed
- State change == OpenGL commands
- Set states
  - glEnable(…)
  - glDisable(…)
  - gl*(…)   //depending on the purpose

Opengl States

- Querying current values in the Opengl state
  - glGet*(…)

  - glGetMatrixfv(GL_PROJECTION_MATRIX, mat)

  where

  “GLfloat *mat” is the placeholder for reading back the 4x4 matrix

OpenGL Evolution

- 2.1
  - ▫ Released August 2006
  - ▫ Fully supported fix-function (FF) pipeline
  - ▫ GLSL-shader support added
  - ▫ Ability to mix-and-match FF and Shaders
    - Ugly code for large projects
  - ▫ Supported industry-wide

Opengl Evolution

- 3.0 – GLSL 1.3
    - Released August 2008
    - Staggered transition away from the FF pipeline
        - Offered deprecation for the FF aspects of the API
        - Use of FF allowed, but not recommended
    - Introduced "contexts"
        - Fwd-compatible context (FWD-CC)
            - NO ACCESS to FF allowed
        - Full context
            - Mix-and-match allowed, but not recommended

Opengl Evolution

- 3.1 – GLSL 1.4
  - March 2009
  - Removed deprecated features from 3.0
  - FF can still be accessed using GL_ARB_compatibility extension
  - Supported by recent hardware

Opengl Evolution

- 3.2 – GLSL 1.5
- Introduced "Profiles"
    - Core-profile (newer stuff)
    - Compatibility profile (for older hardware)
- Core profile – access to 3.2 features only
- Compatibility profile – FF can be used

Opengl Evolution

- 3.3 – GLSL 3.3
  - March 2010
  - New extensions
  - Compatibility with older hardware
  - Supported by NVIDIA Fermi architecture
    - Official GPGPU support

OpenGL Evolution

- 4.1 – GLSL 4.10
  - More extensions and shader types added
    - Tessellation shader & evaluators
  - Targeted towards new hardware
    - Backward compatibility through "compatibility profile"

OpenGL Evolution

- 4.5 – GLSL 4.50
  - August 2014
  - Major improvements in OpenGL state access
  - Support for OpenGL ES 3.1
  - DX11 emulation features
    - allow porting between OpenGL and DirectX applications?

- See https://www.opengl.org/sdk/docs/ for documentation

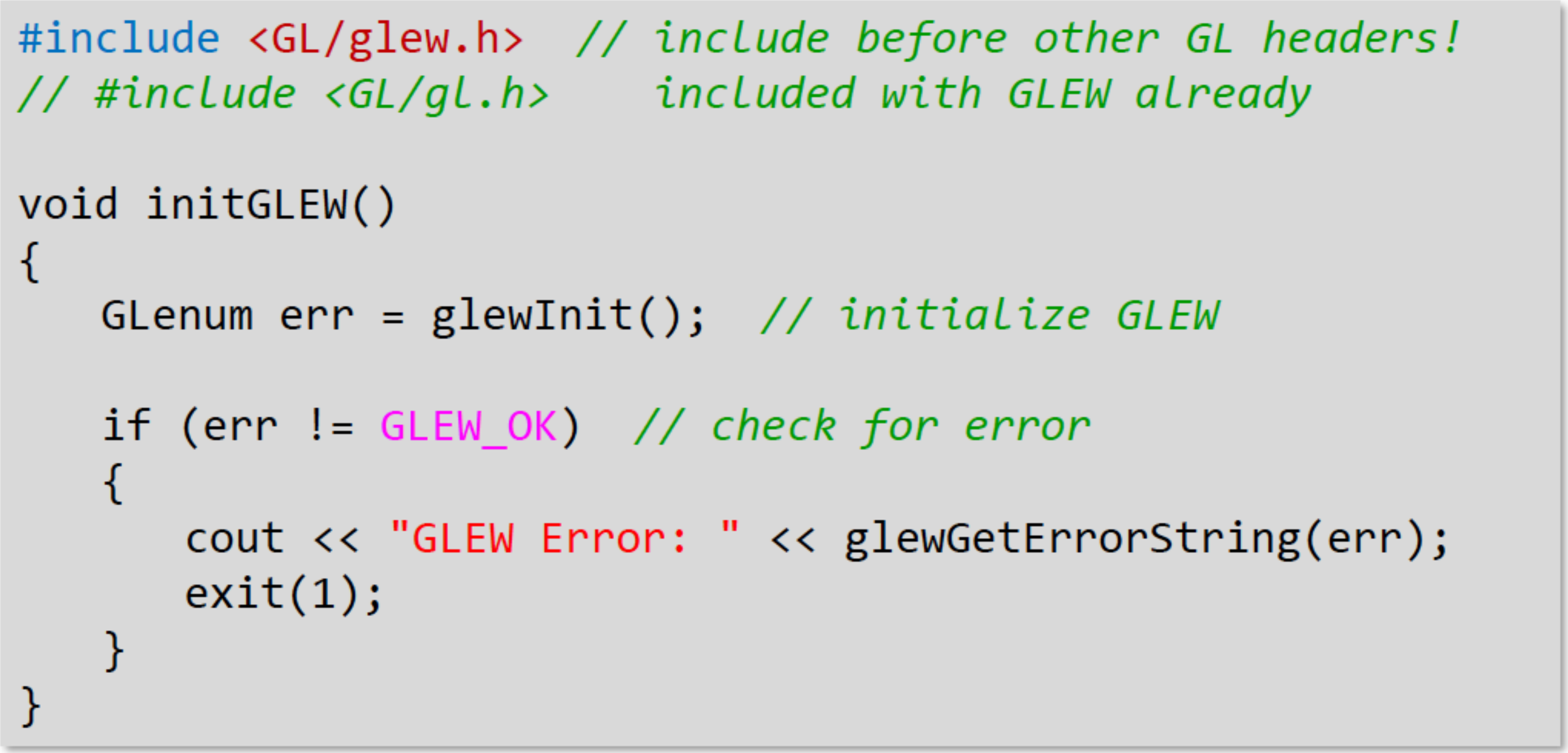
Important distinctions

- From OpenGL 3.x onwards
  - No built-in materials/lighting
  - No immediate mode rendering
  - No matrix stacks and transformations
- The programmer has to provide every information
- Works for us
  - Already did this in CS 250!

Additional libraries – GLEW

- On Windows, only OpenGL 1.1 is supported natively
- To access later functionality, have to query the extensions (about 2000!) individually through the DLLS
  - Nightmare for coders!
- Enter GLEW – GL Extensions Wrangler library
  - Does most of the setup for you (example follows)

Using GLEW



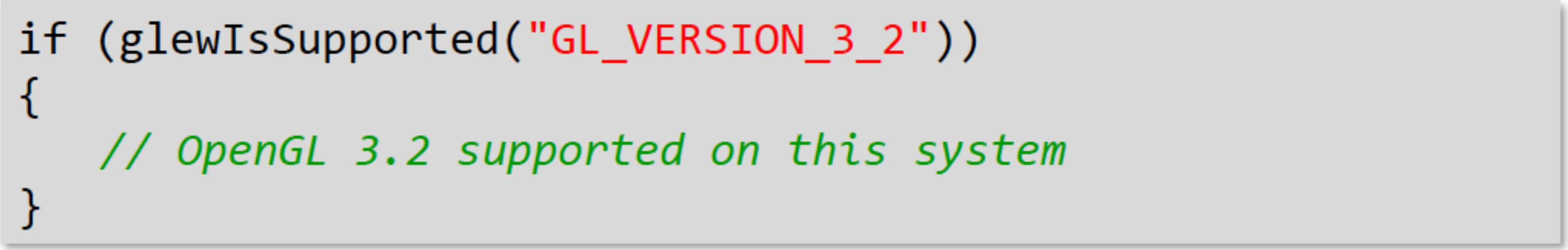
```
#include <GL/glew.h>  // include before other GL headers!
// #include <GL/gl.h>     included with GLEW already

void initGLEW()
{
   GLenum err = glewInit();  // initialize GLEW

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

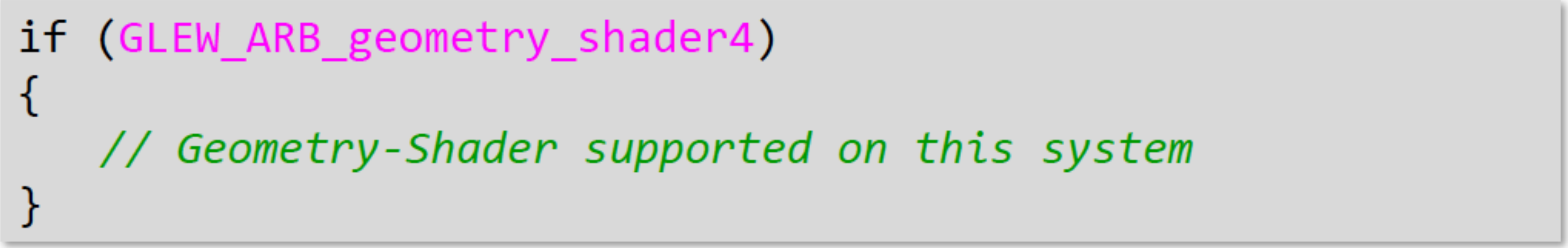
GLEW

- Check for supported version



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

- Check for specific extension



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

Fallback to 2.1

- If your hardware does not support 3.x, then it is possible to create a **2.1 context without FF**
- This means that in your code, DO NOT USE
  - Built-in matrix stacks
  - Immediate Mode rendering
  - Material and lighting
  - Attribute stack
  - Some primitive modes (GL_QUAD*, GL_POLYGON)

GLSL Restrictions

- In GLSL 1.1/1.2, do not use
  - ftransform()
  - All built-in shader variables, except
    - gl_Position (Vertex shader)
    - gl_FragColor, gl_FragData[] (Fragment shader)
- Incomplete list
  - To get complete list, check the official reference pages

References

- [1] OpenGL, http://www.opengl.org
- [2] Khronos Group, http://www.khronos.org
- [3] OpenGL Specification, http://www.opengl.org/registry
- [4] OpenGL 3.2 API Quick Reference Card, http://www.khronos.org/files/opengl-quick-reference-card.pdf
- [5] OpenGL Extension Registry, http://www.opengl.org/registry
- [6] GLEW – OpenGL Extension Wrangler Library, http://glew.sourceforge.net

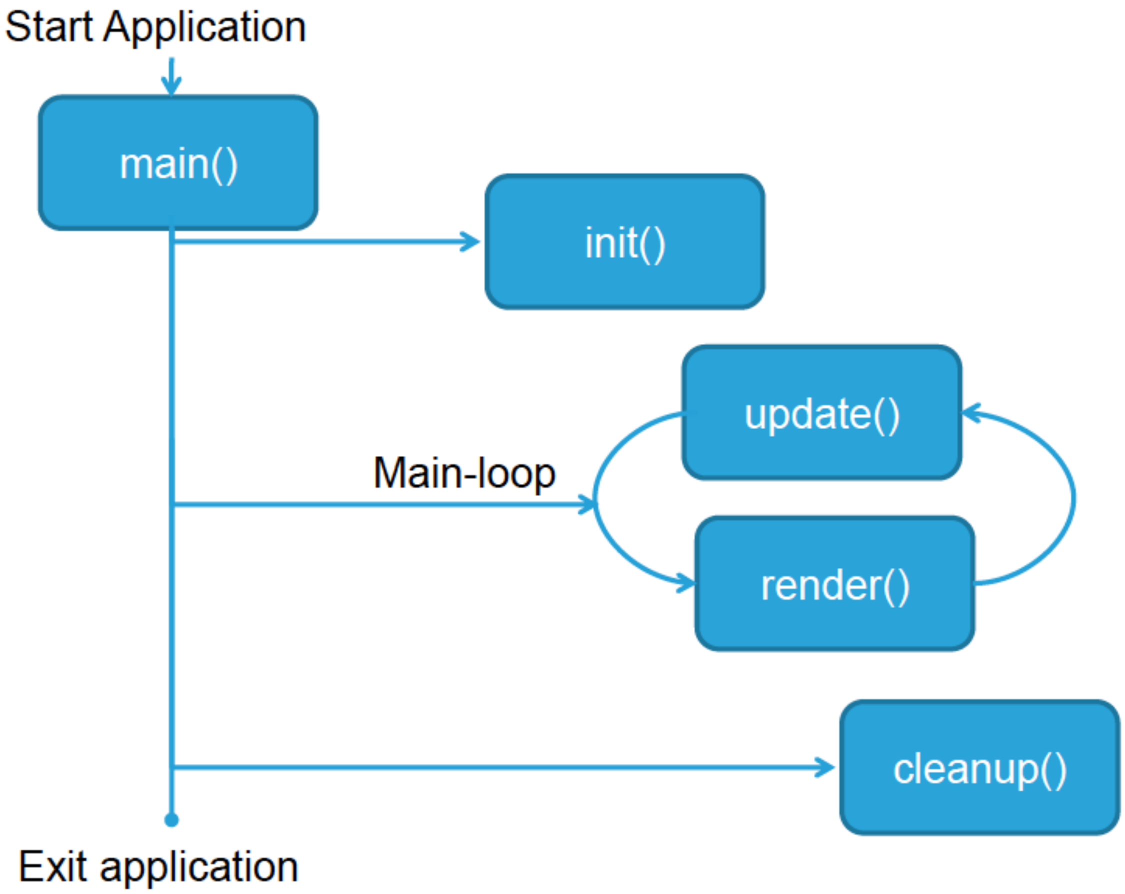
OpenGL Program Example

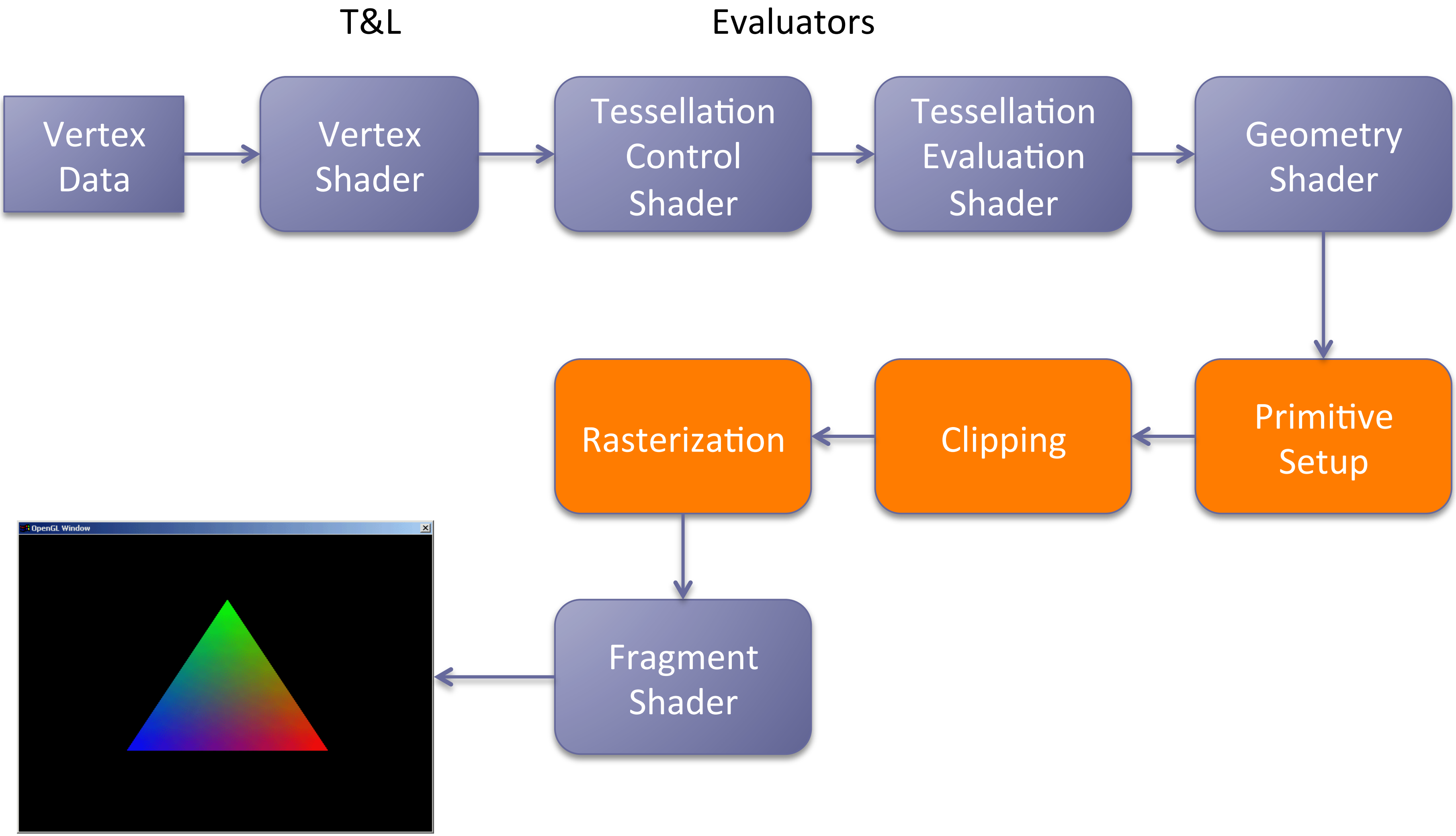
OGL3.3 made easy

OGL Program Implementation

- Typical OGL program runs in a window (/ full screen)
  - Well known window-loop based application
- Platform independence
  - Use third-party libraries like SDL, GLUT, Qt etc.
  - Our example uses freeGLUT
    - Simple set of programming rules
    - Easy setup to get application up and running

Typical application





T&L

Evaluators

Vertex Data

Vertex Shader

Tessellation Control Shader

Tessellation Evaluation Shader

Geometry Shader

Primitive Setup

Clipping

Rasterization

Fragment Shader

OpenGL Object Life-cycle

- All resources in OpenGL are treated in similar fashion
- Object creation
  - Create a **handle** to the object
  - Similar to a pointer in local memory
  - Bind the object to make it current / active
  - Pass relevant data to OpenGL using the handle
  - Unbind object if not immediately used

OpenGL Object Life-cycle

- When using the object (per frame)
  - Bind the object to make it current
  - Use it
  - Unbind it

- When object is no longer needed (at exit)
  - Delete object using handle
  - Delete the data associated with the object

Processing Data in OpenGL

- All data is processed as “buffer objects”
  - Fancy name for a chunk of memory managed by OpenGL server
- Graphical objects are broken into “primitives”
  - Points, lines, and triangles – native primitives
  - Patches – tessellation primitive
  - Adjacency – geometry shaders

Point primitives

- Represented by a single vertex
- Position in 4D homogeneous coordinates
- Represents a square region
  - Width controlled by “glPointSize(GLfloat size)” on CPU
  - “gl_PointSize” variable in vertex, tessellation, and geometry shaders
    - You have to explicitly enable GL_PROGRAM_POINT_SIZE for this to work
    - (confused yet?)

Lines, Strips, Loops

- OpenGL "Line" = Math "Line segment"
- Finite, linear primitive
  - Uses two points for complete specification

- Line loop
  - Sequence of points with the last and first points being joined together
- Line strip
  - Keeps the loop open-ended

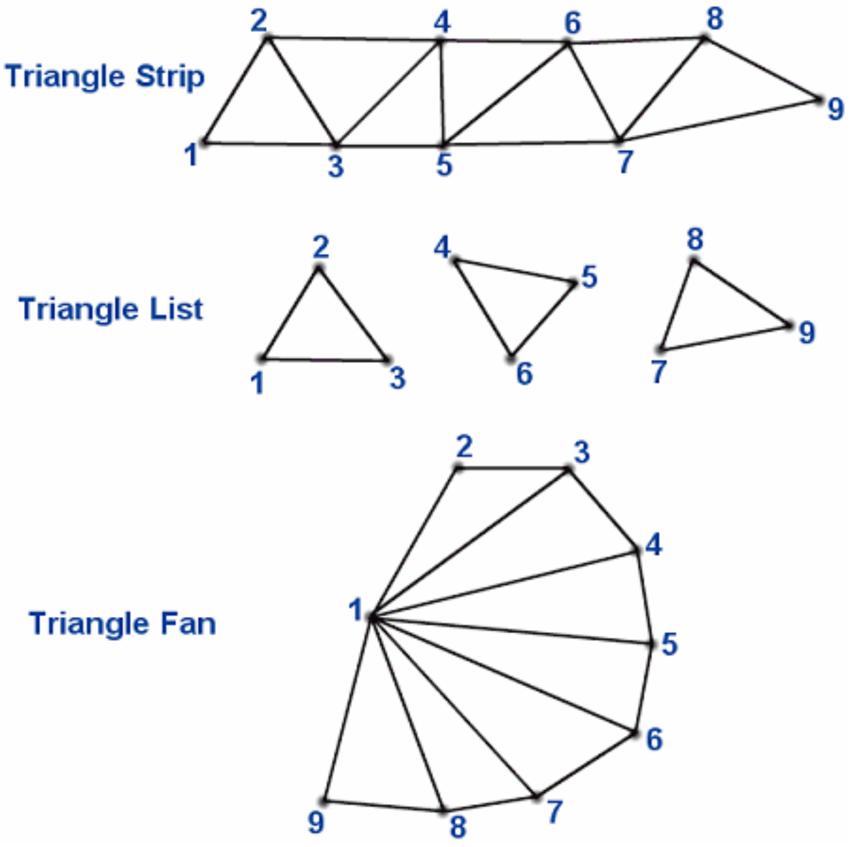
Line Size

- Use “glLineWidth(GLfloat width)”

- No corresponding variable in the shaders
- Repeat rasterization in x- or y- direction for suitable slope values

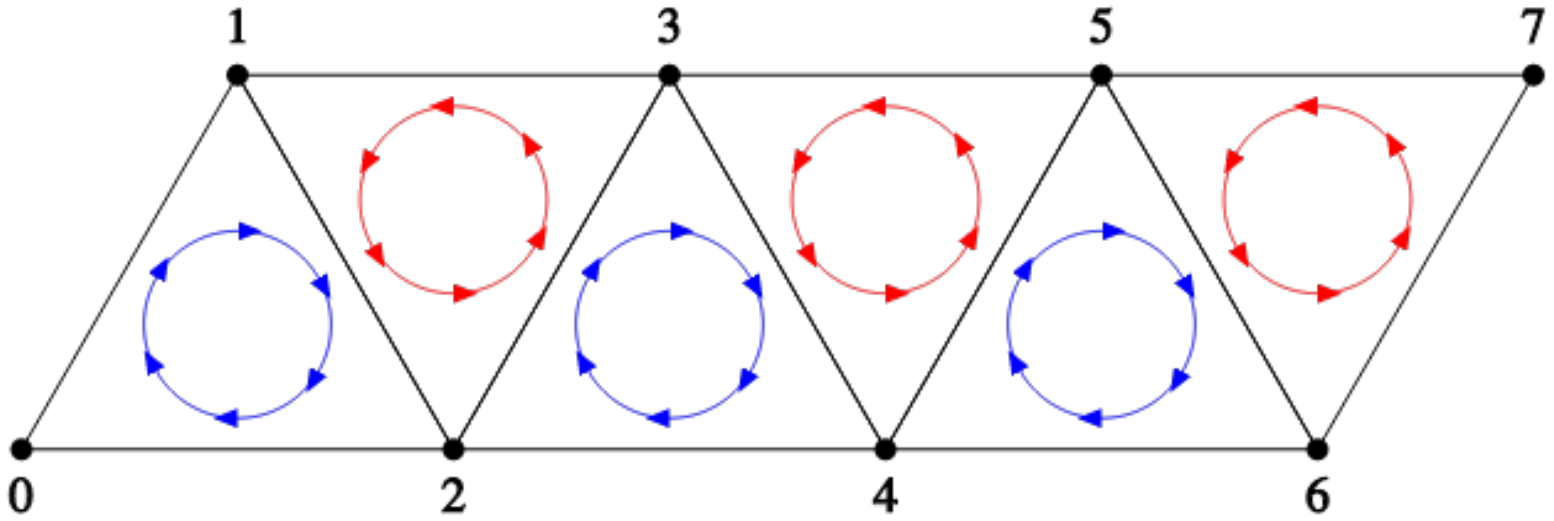
Triangles, Strips, Fans

- Specified using three vertices
- Connected triangles can be specified using
  - Strips
  - Fans



- Triangle Strips
  - A new vertex adds one triangle to the end of previous list
  - Winding is alternated between odd and even triangles
- Triangle Fan
  - A new vertex adds a triangle that includes the first vertex

Triangle Strip Winding Order



OpenGL Buffers

Passing data to the OpenGL Server

OpenGL Data Transfer Model

- All data transfer in OpenGL happens through *buffer objects*
- Buffer objects are OpenGL specific chunks of memory that can be populated with specific type of data
- Buffer objects store information about vertices, indices, adjacency information, and user-defined data

Object Life Cycle

- Generate *named* buffer objects
- Create buffer object by *binding*
- Perform data transfer to/from the buffer
- Delete the buffer object
  - Very important step
  - Failure to do so will leave a memory leak in the OpenGL implementation

- Recommendation: Use smart-class to wrap buffer object functionality

Named Buffers

- Client code cannot directly access server-side memory in OpenGL*
- Indirect references
  - Pointers are useless (,but)
  - Integers are used as indices into the server-side resource pool

* except in one case (more later)

Comparison to C++ resource management

C++

OpenGL

Comparison to C++ resource management

C++

```
void **buffers;
```

OpenGL

```
GLuint  buffers[4];
```

Comparison to C++ resource management

C++	**OpenGL**
void **buffers;	GLuint buffers[4];
buffers = new (void *)[4];	glGenBuffers(4, buffers);

Comparison to C++ resource management

C++	**OpenGL**
void **buffers;	GLuint buffers[4];
buffers = new (void *)[4];	glGenBuffers(4, buffers);
buffers[0] = (Array *) new Array();	glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);

Comparison to C++ resource management

C++	**OpenGL**
void **buffers;	GLuint buffers[4];
buffers = new (void *)[4];	glGenBuffers(4, buffers);
buffers[0] = (Array *) new Array();	glBindBuffer(GL_ARRAY_BUFFER, buffers[0]);
buffers[1] = (Texture *) new Texture();	glBindBuffer(GL_TEXTURE_BUFFER, buffers[1]);

What is “binding”?

- Binding specifies the type of data that will be written to/ read from the specific buffer object

- glBindBuffer(GLenum target, GLuint buffer)
  - target: a valid “buffer type” as specified by OpenGL
    - 14 types in OpenGL 4
  - buffer: previously generated id (name) for an OpenGL buffer object
- If the call is made for first time for a particular id, then the buffer object is created with that name

Fill it up!

- Preceding steps create an empty buffer object
- Data transfer
  - More than one way to do it
  - Explicit storage
  - Replace data partially through a larger chunk
  - Generate data with OpenGL and record it into a buffer object

glBufferData(GLenum target, GLsizeiptr size, const GLvoid *data, GLenum usage)

- target : valid buffer object target
- size : amount of storage in bytes
- data : source memory location
- usage : hint to GL server
  - GL_<Token1>_<Token2>
  - Token1 : STATIC, DYNAMIC, STREAM
  - Token2 : DRAW, READ, COPY
- glBufferData MAY reallocate memory for the buffer already created

glBufferSubData

- glBufferSubData(GLenum target, GLintptr offset, GLsizeiptr size, const GLvoid *data)
- Overwrite part of the target buffer object starting from *offset* and of length *size* bytes, addressed by *data*

Other Functionality

- glClearBufferData / glClearBufferSubData
- glCopyBufferSubData
- glGetBufferSubData

- glMapBuffer / glUnMapBuffer
  - Maps the entire buffer object to the client's address space so that client application can manipulate the buffer object
  - GL_READ_ONLY, GL_WRITE_ONLY, GL_READ_WRITE
  - If access mode is invalidated, behavior is *undefined*

Discarding buffer data

- glInvalidateBufferData(GLuint buffer)
- glInvalidateBufferSubData(GLuint buffer, GLintptr offset, GLsizeiptr length)

- Informs OpenGL server that we do not need this data anymore

Thank You