**Chapter 1: Introduction**

* What the graphics pipeline is and how OpenGL relates to it.
* The origins of OpenGL and how it came to be the way that it is today.
* Some of the fundamental concepts that we’ll be building on throughout the book.

**What is an abstraction layer?**

* Provide an abstraction layer between application and underlying graphics subsystem (high-performance hardware)
* Abstraction level must be balanced
* OpenGL must keep up with advancements in technology

**Shaders and the Graphics Pipeline behind it:**

* Shader cores: small programmable processors that run shaders
* Vertex Fetch -> Vertex Shader -> Tessellation Control Shader -> Tessellation Evaluation Shader -> Geometry Shader -> Rasterization -> Fragment Shader -> Framebuffer Operations

**OpenGL:**

* Silicon Graphics Inc. made high-end graphics workstations
* 1990: released GL as an open library, royalty free
* 1992: first version of Open GL, 1.0 and Architectural Review Board (ARB)
* OpenGL 4.5 released in 2014
* Focus on graphics and not about dependencies
* Operating and window systems are independent
* Always use vertex and fragment shaders
* We will use programmable pipeline and not fixed
* Euclidean Space: 4D homogeneous coordinates (x,y,z,w) – Vertex VBOs must be stored in VAO

**What is a primitive?**

* Fundamental unit of rendering in OpenGL: points, lines and triangles
* Break complex shapes into primitives and send them to OpenGL, using rasterizer (accelerator)
* Triangles: Always convex
* Concave polygons can be broken down into triangles
* Rasterizer (hardware) converts 3D representation of triangle into pixels, drawn at screen

**Major components of the Graphics Pipeline:**

* Vertex: point within coordinate space
* Vertex Processor: carry out coordinate transformation, compute color for each vertex
* Processing: Transformations in world coordinates (4x4 matrix)
* Clipper/Primitive Assembler: automatically by viewport, performed on primitives (visible part)
* Rasterization: Transform vertices to window coordinates
* Fragment Processing: takes fragments by rasterizer and update pixels in frame buffer
* Model-View-Projection: model to screen coordinate system, position to world and camera
* Model-View: object to camera space
* Projection: Camera to Window Space (homogeneous clip space)
* Vector: representation into a large number of independent pixels
* Front End: vertices and primitives -> points, lines, triangles -> rasterizer
* Back end: depth, stencil testing, fragment shading, blending, updating output image

**Color:**

* Segmentation, visible spectrum the human eye can see, electromagnetic radiation
* RGB: wavelengths responded by human eye
* Each color component stored separately in the frame buffer
* Range: 0.0 – 1.0, none to all, float range: 0 – 255 (8 bits per component in buffer)
* Hue (what color), Saturation (how far from gray), Value (how bright), visualize cone
* CMYK: Cyan, Magenta, Yellow, Black (used for movies and printing, better than RGB)
* Check if side can be textured and what color
* NVIDIA and AMD: NVIDA is more powerful

**Example for Triangle:**

* Define 3 points
* 3 x 3 array (x, y ,z): float points[] = {…};
* Copy from CPU, send to GPU in a unit called VBO (Vertex Buffer Object)
* Rasterize (do I want to render these objects?)
* Take objects to scene (Screen) – fragment shader (pixel to screen)
* Vertex Array Object (VAO): collection of VBOs to hold vertex data
* Set up mesh then draw on surface, bind VAO

**Shaders:**

* Programmable Shaders: define how to draw the VAO with a piece of code
* Vertex Shader: responsible for geometry, 1 input variable: vec3, once per vertex
* Fragment Shader: responsible for color, 1 output variable: vec4 (RGBA), once per fragment (0-1)

**Using Shaders:**

* Load string into GL shader and compile
* Compiled shaders must be combined into single executable
* Specify GPU shader program, attach shaders
* Set VAO as input variables
* Draw triangles starting from 0, 3 points at a time
* Have display while loop that refreshes window

**Client – Server:**

* Popularized by X Windows System (Direct X)
* Clients use the client servers by a workstation
* Client: OpenGL applications
* Server: workstations with devices, provide output based on input devices

**Input Modes:**

* Devices contain a trigger which can be used to send a signal to operating system
* Return information when triggered
* Three Input Modes: Request, Sample (Polling), Event
* Request: input provided only when user triggers device (keyboard input)
  + Trigger process -Measure Process – Program
* Sample Mode: input is immediate, no trigger needed
* Problem: user has to identify which device provides input (hard for big applications)
* Event: each trigger generates event whose measure is put in an event queue
  + Event Queue: tracked triggers one by one
  + Positioned between Measure process and program
  + Types: window resize, mouse clicks and motion, keyboard shortcuts
* Callbacks: interface for event driven input, define error callback function for error handling
* Main event loop: processes events in event queue

**Drawing:**

* Double buffering: causes flickering (read and display conflicts)
  + Solution: two different frame buffers, front (display) and back (draw)
* Controlling Speed: GPU can render millions of primitives per second
  + Solution: Timing Mechanisms – lock swapping of buffers to screen refresh rate
* Hidden Surface Removal: remove surfaces which are not visible to viewer
  + Solutions: object space and Image space algorithms
* Object Space: use pairwise testing between polygons (objects) – O(n2) for n polygons
* Painter’s Algorithm: Render polygons back to front so further ones are painted over
* Depth Sort: ordering of polygons by Z-order index – O(nlogn)
  + Advantages: simple, easy transparency, might not need sorting
  + Disadvantages: hard to complex geometry, expensive sorting, penetration
  + Must be implemented by programmer
* Image-Space: Look at each projector (for n x m framebuffers), find closest of k polygons
  + Complexity: O(nmk)
  + Ray Tracing
* Z-Buffer Algorithm: use z-buffer to store depth of closest objects when objects are rendered
* Compare to z-buffer depth
  + If less, update color & z-buffer
  + Pros: simple (no sorting), independent on geometric primitives
  + Cons: Memory intensive, tough transparency & blend, z-fighting
* In GLFW and OpenGL:
  + glfwWindowHint(GL\_DEPTH\_BITS, 24); - ask for a depth buffer when creating window
  + glEnable (GL\_DEPTH\_TEST); - call this in program’s initialization routine
  + glDepthFunc(GL\_LESS); - set depth function
  + glClear(GL\_DEPTH\_BUFFER\_BIT); - call everytime you draw

**PROGRAMMABLE SHADERS:**

* GPU pipeline can be modified by developer (graphics algorithms with fast implementation)
* Shaders: Vertex and Fragment Processors
* **Vertex Processor:**
  + Vertex Transformation
  + Normal Transformation
  + Texture Coordinates generation/transformation
  + Lighting
  + Color and Material
* Vertex Shader: replaces vertex processor and entire functionality
* Fragment Processor:
  + Operates fragments after interpolation and rasterization
  + Operations on interpolated vertex values (Phong)
  + Texture access
  + Color sum
  + Work in parallel
  + No access to neighboring fragments
* GLSL:
  + High level programming language
  + Data types: vector, matrix, texture
  + Control flow, same as C
  + Uniform variables: remain constant along a primitive, can be read but not written
  + Input variables (in): read only variables (vertex: position, normal, uv, color)
  + Output variables from vertex become input to fragment
  + Output variables: interpolated output variables from VS (declared in both VS and FS)