Computer Graphics

Dr. rer. nat. Martin Möddel

April 6, 2021

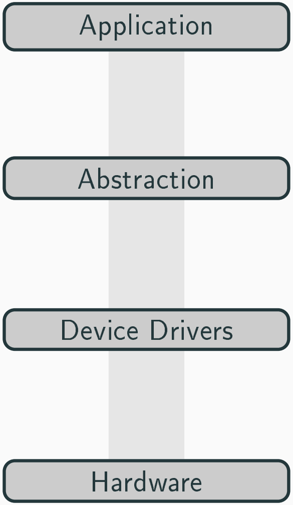
Institut für Biomedizinische Bildgebung

OpenGL

OpenGL - What is OpenGL

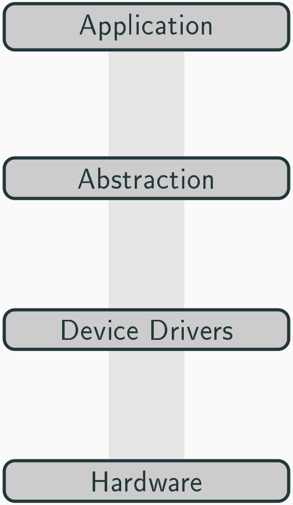
- API for real-time computer graphics
- Large state machine with state-changing and state-using functions
- Supported across many platforms, languages and devices
- Provides to implementers support for extensions (can be vendor specific), which eventually become part of the standard
- Applied in computer-aided design, virtual reality, video games, and more
- Managed by a non-profit technology consortium (Khronos Group).
- Initial release in june 30th 1992
- Latest stable release is version 4.6 (july 31st 2017)

OpenGL - Software Pipeline



Software Pipeline
The software pipeline exists to describe what we'd like to see on the screen and consists of several different layers, each with their own very specific purposes. This pipeline serves as a relay from your program to the dedicated hardware and the only part of the software pipeline that you will actually use in your programs is the Application layer, which exposes the APIs to you.

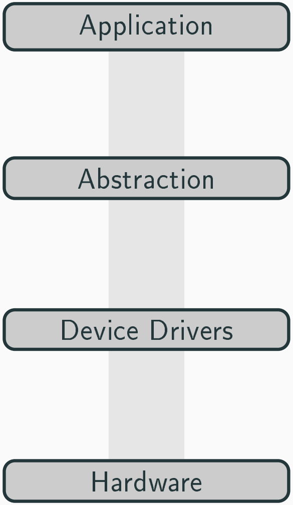
OpenGL - Software Pipeline



Application Layer
This is the program that invokes drawing commands. The application serves as a controller of the overall process and oversees all of the user-level operations such as

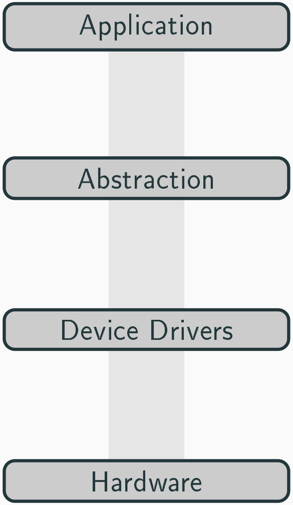
- creating windows
- threads
- memory allocation
- complex user data-types
- making calls to external libraries such as OpenGL

OpenGL - Software Pipeline



Abstraction Layer
This layer contains the OpenGL API implementations, which are different from the API in the Application layer. In C terms: you can think of the Application layer as the header file containing only definitions, while the Abstraction layer is the source file containing the actual functionality.

OpenGL - Software Pipeline



Device Driver
Is a software communication layer to the hardware, which is entirely invisible to the developer, since it cannot be interacted with through your program. The device driver interprets the commands passed to it by the Abstraction layer and relays them to the underlying device in a format that the hardware can understand and easily process.

OpenGL - History

- OpenGL was released by Silicon Graphics (founded in 1981) as specification of how the API should work
- The OpenGL Architecture Review Board (ARB) was founded in 1992, which comprised of several high profile software and hardware vendors who collectively decided the future of the OpenGL standard
- True OpenGL implementation are approved by the ARB through conformance testing
- Microsoft implemented OpenGL in Windows NT 3.5, which was released in 1994 (very slow implementation)
- Time of support ended whith OpenGL 1.1 for Windows 95 and Windows NT 4.0 in favor for their own API Direct3D

OpenGL - History

- In the late 1990s, OpenGL established itself as an industry standard for 3D computer graphics
- Games such as Quake 2, Unreal, and Half-Life took full advantage of OpenGL and were widely popular
- First consumer-grade dedicated 3D graphics hardware appeared (Voodoo Graphics by 3Dfx Interactive and 3D Rage by ATI)
- In 1999 NVIDIA caught up with their GeForce 256 add-on card that they termed GPU (Graphics Processing Unit)
- GPU performance grew exponentially during the early 2000s and more features were moved to the GPU
- CPU became the major bottleneck for real-time 3D graphics
- To solve the issue data would be stored the GPU's memory and stay there until no longer needed

OpenGL - History

- Microsoft released Direct3D 8.0 in 2000, which supported shaders (little programs that run directly on the GPU)
- OpenGL did not officially support shaders until the release of OpenGL 2.0 in 2004 and the simultaneous release of the OpenGL Shading Language (GLSL)
- From 2004 to 2006, Direct3D 9.0 was dominating the games market
- From 2006 the Khronos Group took over management of OpenGL
- It took until 2009 when OpenGL 3.2 finally cached up with Direct3D
- OpenGL 4.0 was released in 2010 as an API for the latest generation of GPUs similar to Direct3D 11.

OpenGL - Initialization

- Before OpenGL can be used in a program it must be initialized
- Create a OpenGL context
- Create an application window to draw in
- Load all of the necessary functions to use OpenGL
- Libraries such as GLFW can be used to create a window, attach an OpenGL context to this window, and manage basic input for that window

OpenGL - Context

- A context stores all of the state associated with an instance of OpenGL
- Contexts are localized within a particular process of execution
- A process can create multiple OpenGL contexts each with its own set of OpenGL Objects, which are independent of those from other contexts but may be shared with other contexts
- Any OpenGL commands only affects the **current** context
- The current context is a thread-local variable
- A single process can have several threads, each of which has its own current context
- A single context cannot be current in multiple threads at the same time
- Each context can represent a separate viewable surface

OpenGL - Example Context Creation

Context Creation with GLFW using Julia

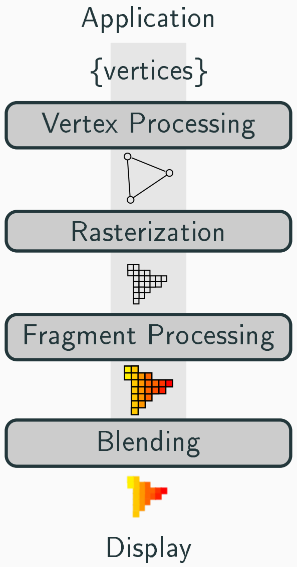
```julia
using GLFW

# Create a window and its OpenGL context
window = GLFW.CreateWindow(640, 480, "GLFW.jl")

# Make the window’s context current
GLFW.MakeContextCurrent(window)

# Loop until the user closes the window
while !GLFW.WindowShouldClose(window)

    # After initialization all the rendering takes place in the main loop
    # as long as the window hasn’t been closed

end

GLFW.DestroyWindow(window)
```

- GLFW is written in C and supports Windows, macOS, the X Window System and the Wayland protocol
- handles windows and user input
- Support for multiple windows, multiple monitors
- Support for keyboard, mouse, gamepad, time and window event input, via polling or callbacks

OpenGL - Double Buffering

- It is difficult to draw a display so that pixels do not change more than once
- If we draw directly onto the screen this results in stutter, tearing, and other artifacts
- Therefore, a double buffer is used
- The front buffer contains the final output image that is shown at the screen
- The back buffer is drawn on by the OpenGL state machine
- As soon as all rendering commands are finished the buffers can be swapped to display the most recent image on screen
- The swap does not happen automatically, but must be issued by the application

OpenGL - The OpenGL Pipeline



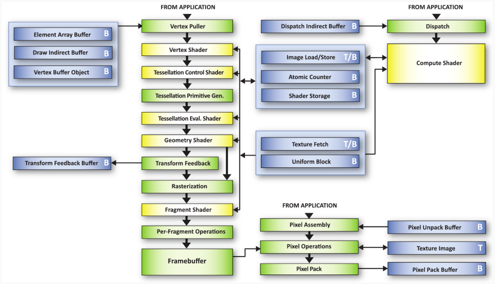


Figure 1: OpenGL pipeline Taken from the OpenGL 4.6 API reference guide

OpenGL - The OpenGL Pipeline

- The OpenGL pipeline transforms vertex data to 2D pixels
- First transforms your 3D coordinates into 2D coordinates
- The second part transforms the 2D coordinates into actual colored pixels
- Because of their parallel nature, graphics cards of today have thousands of small processing cores to quickly process your data in parallel
- The processing cores run small programs on the GPU for each step of the pipeline
- These statges are either fixed functions that can not be modified or programmable so called **shaders**
- Shaders are written in the OpenGL Shading Language (GLSL)

OpenGL - Shaders

- Modern OpenGL requires that we at least set up a vertex and fragment shader
- Each shader stage has a separate set of inputs and outputs, as well as built-in variables
- These shaders need to be written in the shader language GLSL (OpenGL Shading Language) and then compiled at run time
- Once compiled the shaders are linked to a shader program object
- Multiple such shader program objects can be created, hwoever only one can be active
- The active shader program's shaders will be used when issueing a rendering call

OpenGL - Vertex Input

- To start drawing OpenGL needs some input vertex data
- Submitting vertex data for rendering requires creating a stream of vertices, and then telling OpenGL how to interpret that stream (primitives)
- Internally, OpenGL is a 3D graphics library so all coordinates that we specify in OpenGL are in 3D
- OpenGL only processes 3D coordinates when they're in a specific range between -1.0 and 1.0 on all 3 axes (**canonical viewing volume**)
- Input data is not limited to the vertex position, but also covers vertex attributes such as normal vector, color and texture coordinate

OpenGL - Vertex Input

- The order of vertices in the stream defines how OpenGL will process and render the primitives the stream generates
- There are two ways of rendering with arrays of vertices
  1. Directly, generating a stream in the array's order, e.g.
     ```
     vertices = [ [0, 0, 1], [0, 0, 0], [1, 1, 1], [0, 0, 1], [0, 0, 0], [0, 0, 1] ]
     ```
  2. Using a list of indices to define the order, which is useful in most tight meshes, were vertices are used multiple times
     ```
     vertices = [ [1, 1, 1], [0, 0, 0], [0, 0, 1] ]
     indices = [2, 1, 0, 2, 1, 2]
     ```
- Additional attributes such as texture coordinates

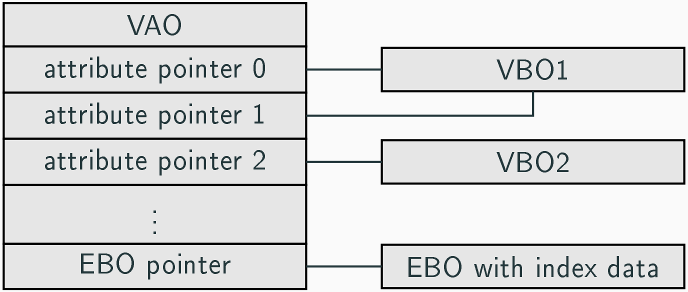
```
textureCoordinates = [ [0, 0], [0.5, 0], [0, 1] ]
```

will modify the stream

```
        vertexStream = [ [[0, 0, 1], [0, 1]], [[0, 0, 0], [0.5, 0]], [[1, 1, 1], [0,
              0]], [[0, 0, 1], [0, 1]], [[0, 0, 0], [0.5, 0]], [[0, 0, 1], [0, 1]] ]
```

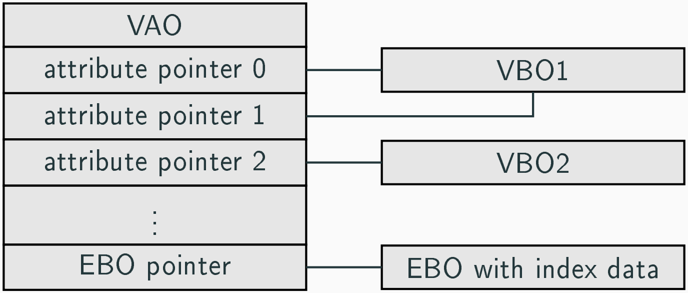
OpenGL - Vertex Array Object

- A Vertex Array Object (VAO) is an OpenGL object that stores all of the state needed to supply vertex data



OpenGL - Vertex Array Object

- VAO stores the format of the vertex data as well as the Vertex Buffer Objects (VBO) providing the vertex data arrays
- VAOs do not copy, freeze or store the contents of the referenced buffers!
- Element Buffer Objects (EBO) store indices that define the order in which vertices are send to the if bound to the VAO



OpenGL - Vertex Shader

- As input to the graphics pipeline we pass vertex data, which is a collection of vertices
- Each vertex is a collection of data per 3D coordinate (e.g. 3D position and some color value)
- The vertex shader that takes as input a single vertex, transforms 3D coordinates, and allows us to do some basic processing on the vertex attributes

OpenGL - Early Primitive Assembly

- Happens if geometry shader or tessellation is active
- The primitive assembly stage convert a vertex stream into a sequence of base primitives
- OpenGL requires you to hint what kind of primitive you want to form with the data
- Some of these hints are `GL_POINTS`, `GL_TRIANGLES`, and `GL_LINE_STRIP`
- The output of the primitive assembly stage is passed to the tesselation stage

OpenGL - Tesselation Stage

- Stage at which primitives are further subdivided into smaller Primitives
- Optional
- Governed by two shader stages and a fixed-function stage
- The tessellation control shader determines how much tessellation to do
- The tessellation primitive generator takes the input patch and subdivides it based on values computed by the TCS
- The tessellation evaluation shader takes the tessellated patch and computes the vertex values for each generated vertex

OpenGL - Geometry Shader

- A geometry shader is able to transform a single input primitive to completely different primitives possibly generating much more vertices than were initially given
- Optional
- Geometry shader invocations take a single Primitive as input and may output zero or more primitives
- The maximum number of vertices that can be output is limited
- The total maximum number of output components that can be output is limited

OpenGL - Vertex Post-Processing

- Transform feedback: Values output from the last vertex processing stage can be recorded into Buffer Objects
- Clipping: Primitives generated by previous stages are collected and then clipped to the view volume
- Perspective divide: Clip-space positions are transformed into normalized device coordinates
- Viewport transform: Normalized device coordinates are transformed to window space
- Primitive Assembly: Vertex stream is converted into a sequence of base primitives, which can be discarded based on their apparent facing (face culling)

OpenGL - Rasterization

- Rasterization is the process whereby each individual primitive is broken down into discrete elements called fragments
- A fragment in OpenGL is all the data required for OpenGL to render a single pixel
- Clipping discards all fragments that are outside your view
- The remaining fragments are send to the fragment shader

OpenGL - Fragment Shader

- Process a Fragment into a depth value, a possible stencil value, and zero or more color values
- The stage where all the advanced OpenGL effects occur
- Take a single fragment as input and produce a single fragment as output
- Each fragment has a window space position, a few other values, and it contains all of the interpolated per-vertex output values like lights, shadows, color of the light and so on

OpenGL - Per-Fragment Processing

- Fragments output from a Fragment Shader are processed, and their resulting data are written to various buffers
- This stage contains a number of tests that can be performed to determine visibility
- Pixel ownership test: Fragments aimed at pixels not owned by the current OpenGL context (if the window one is rendering is partially obscured by another window) are discarded
- Depth test: Fragments are discarded based on a conditional test between the fragment's depth value and the depth value stored in the current depth buffer (allows for occlusion)
- Stencil test: Additional test using a buffer very similar to the depth buffer that may be used to limit the area of rendering (stenciling) in the simples case

OpenGL - Per-Fragment Processing

- Multisample Antialiasing (MSAA): Samples each pixel at the edge of a polygon multiple times with a slight offset to all screen coordinates. The fragment shader is only run once per pixel and the color is then stored inside each subsample, which is coverd by the primitive. Finally, all these colors are then averaged per pixel resulting in a single color per pixel
- Blending: Each of the colors in the fragment can be combined with the corresponding pixel color in the buffer that the fragment will be written to
- More

OpenGL - Example

```
julia susuzanne_lambert_shading.jl
```