Graphics APIs

Mostly OpenGL

Rendering

- In graphics, “rendering” is the process of converting a description of a scene into an image
- This is a hard and computationally expensive problem
- Consider the simplest case with no complicated “shading”
- For each pixel in the image we need to determing which part of the scene is visible
- The complexity is O(screen size) which is O(width^2) (quadratic) and O(f(scene complexity)) where `f` depends on the scene + rendering algorithm, etc
- We'll be looking at real time rendering today where we want to render a scene at 60 FPS

History

- Basic rendering of 3D scenes with simple shading requires 2 main operations shading + occlusion
- Shading means given the piece of a 3D object visible at a given pixel, what color should it be? It depends on
  - Surface properties (solid color or varying texture)
  - Lighting including direct from light sources or indirect bounces off of other objects
  - View angle
- Occlusion: what's part of the scene is visible at a given pixel (usually what's the closest bit of scene geometry)

Graphics Hardware

- Occlusion can be implemented by “Rasterization” and a “Depth Buffer”
  - Rasterization: For each triangle, figure out which pixels it covers
  - Depth Buffer: store distance to the “nearest” thing we've seen at each pixel so far, and update whenever we see something closer
- These are easily parallelizeable operations, and by the 90s companies were selling graphics accelerators to do this
- Shading usually requires lots of operations on 3D vectors of floats, those operations are also easy to build into hardware
- Some names: SGI (made workstations for VFX), VoodooFX (early popular graphics card for gamer's PCs)

APIs

- Imagine you're a game dev in 1997
- Your customers might have a number of different graphics cards
- To use their features, you need to write custom code for each one that calls functions exposed by those drivers (or maybe pokes the hardware directly)
- This will be miserable, tons of work, and probably buggy!
- The OpenGL and DirectX APIs aimed to solve this problem

OpenGL

- The API for app devs is a relatively high level set of functions for scene specification
  - Geometry data (usually triangle meshes)
  - Textures/Colors/Shinyness/etc
  - Lights (placement, direction, intensity, etc)
- The graphics card manufacturers implement these APIs with low level instructions to the hardware (copy geometry data to a particular part of memory, do dot products, write a color to the “framebuffer”, etc)
- Any compliant graphics card can run any app written using OpenGL

Progress Marches On

- OpenGL was a good match for the feature set of hardware when it was created, but not anymore
- Original had a “fixed function pipeline” where all geometry was rasterized (chopped into pixels) then shaded with hardwired shading models
- Eventually, graphics hardware became user programmable, so users could write programs to run at various stages of the rendering process
- For example “fragment shaders” are programs that run for each pixel. You can customize inputs to this stage and the output will be the final color at that pixel
- Modern versions of OpenGL REQUIRE devs to write programmable shaders and the “fixed function” API is completely removed

Progress Marches Further

- As graphics code written by app devs got crazier, they wanted more performance out of their driver
- In general, you can't call multiple OpenGL functions safely from multiple threads, among other issues
- Eventually lots of stakeholders decided that the OpenGl model wasn't a good design anymore
- Out came Metal (Apple) Vulkan (the OpenGL consortium) and new versions of DirectX (MS) which all provided a lower level API that exposed more of the capabilities of modern graphics cards
- The APIs are also simpler for the drivers, so hopefully drivers should be less buggy and app code will be easier to optimize

Where we are (were)

- We're sort of back where we were at the end of the 90s
- If you want your game to run on an apple platform you must use Metal
- If you want it to run on Windows, you should probably use new Direct X
- Vulkan runs lots of places, but not macs!
- Maintaining all those versions of your app is a nightmare!

One More API: WebGL

- JS devs can write OpenGL code to run in a browser using WebGL
- This was always sort of Meh
- Now it's kind of slow/awkward because, for example macs have poor OpenGL support, so it might actually be a translation layer above Metal
- BUT...

Super new thing: WebGPU

- This is “Metal for JS” which is an upgrade for browsers
- BUT, it was designed to be easy to use OUTSIDE of browsers as well
- There's a C API for it and some nice wrappers for rust + C++ and some others so far
- Some folks are saying that this should be the future for any code using GPUs for all platforms

What about 6018?

- We're going to look at “modern” OpenGL on Android
- Some aspects will be similar for the newer APIs, and it's well supported + mature on Android, so a good choice for dipping out toes into accelerated graphics

Some Concepts

- Buffers: These are arrays of data that will eventually be in GPU memory (copying between GPU/CPU is slow on many platforms, so we'd like to avoid it)
- Shaders: These are programs we write that get run at various stages of the rendering process. They will get run MANY times, for each “piece of geometry”
  - Vertex Shaders: run for each vertex. Their job is to take a point in “world” space, and eventually return pixel coordinates
  - Fragment Shaders: run for each visible triangle pixel. Take in data from previous stages and output the color of that pixel
- Most of the work we'll be doing is setting up buffers and shaders

Coordinate Transforms

- A big job of the vertex shader is transforming points between several coordinate systems:
  - Model coordinates: coordinate system we use in, for example our 3d modeling software. Could be anything
  - World coordinates: where those models should go in our scene. The models might be scaled up/down, rotated, repositioned, etc
  - Projection Transform: This converts world coordinates to screen coordinates. The most common is the “perspective projection” which makes far away objects in world space appear smaller in screen space
- These transforms are represented as 4×4 matrices, so “applying” them is just matrix vector multiplication

Shaders

- OpenGL shaders are written in a language called GLSL (GL Shading Language)
- It's basically C with a few nice built in types for vectors/matrices
- There's a few special kinds of variables
  - `attribute` — per-vertex data passed in buffers
  - `uniforms` — data that all vertices share
  - `varying` — data “interpolated” from a previous rendering phase
  - Some special variables which are the “output” of our shaders. We write to them instead of returning something

Setting up shaders

- Shaders must be compiled and linked at runtime
- So we pass the string containing our shader code to `glShaderSource` and then run `glCompileShader` to compile them
- Then we “link” all shaders into a “program” using `glLinkProgram`
- I'm omitting lots of steps here, we'll step through an example, but to write real code, you'll need to look at the documentation

Android

- OpenGL requires a “Context” which is bascially a place to draw onto
- Even in a compose base app, a `GLSurfaceView` seems to be the best approach
- It has a `setRenderer` method and we'll put all our GL code in our renderer class which implements the `Renderer` interfaces
- The most important methods for us are `onSurfaceCreated` and `onDrawFrame`

Recap

- Hopefully the history is interesting and gives you some context for why things are the way they are
- If you build an Android app today, OpenGL might be an OK choice, but probably not for much longer
- The stuff we covered today will be similar in the newer graphics APIs.
- The newer APIs require you to set up the various pipeline phases and handle a bunch of other low level details which OpenGL basically doesn't let you customize