Preface: What is OpenGL?

On the most fundamental level, OpenGL is a software interface that allows a programmer to communicate with graphics hardware. Of course, there is much more to it than that, and you will be glad to know that this book explains the finer details of OpenGL. But before we get our hands dirty and start coding, you'll need to know a little about the history of Computer Graphics and OpenGL.

In the preface we'll explore the following topics: * The inception of computers and computer graphics * What OpenGL is and how it came to be * How computer graphics work * Hardware and software requirements for this book

> **Attention:**
> If you want to get straight into graphics programming without reading this lengthy history, you may skip straight to the "requirements" section below, read it, and start the next chapter. I have to stress, however, that having a thorough understanding of the history of computing, computer graphics, and OpenGL can be important in understanding future developments.

In The Beginning

Whether through writing, painting, or body language, imaging has always been an important player in relaying information and chronicling history. The requirement for visual feedback is so important that it is hard to acknowledge the existence of something if you cannot see it. Bacteria, for instance, were purely speculative before their visual discovery in the seventeenth century by Antonie van Leeuwenhoek who invented the microscope, but became an integral part of modern science.

Computer data is represented in nothing more than electrical pulses, which are also invisible to the naked eye. A method of displaying this data had to be invented, so early computer scientists would get visual feedback from their machines through a series of lamps mounted onto boards or long perforated paper tapes, so called "punch cards."

As you can imagine, this information was far from readable and much interpretation was required to convert the information into a human-readable format. And even though computers were eventually equipped with electric typewriters, the output was far from optimal.

Display: Cathode Ray Tubes

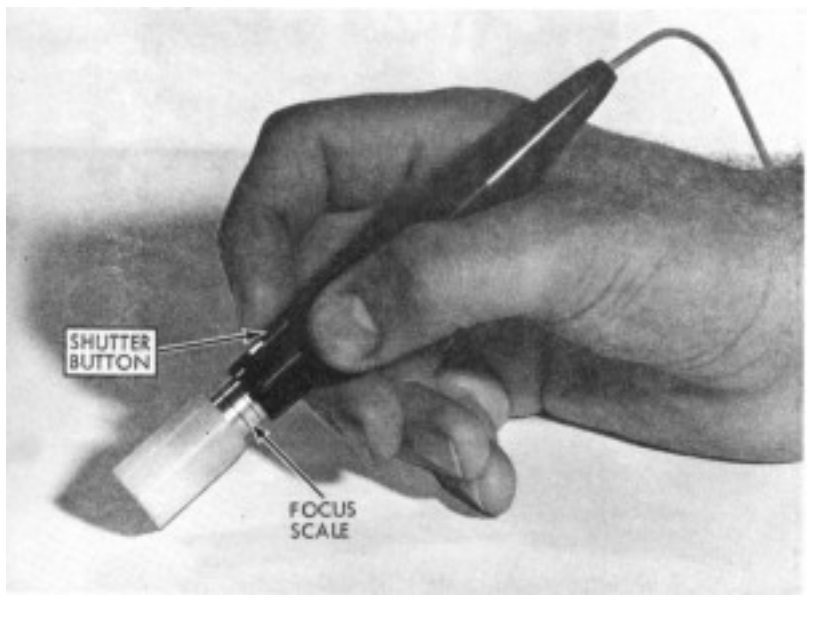
In 1897, Ferdinand Braun invented the CRT (Cathode Ray Tube) in Germany as a type of vacuum tube whose purpose was to display an image onto a screen. You may have seen or used them yourself in the form of glass-tube televisions and computer monitors that were the norm until very recently.

CRTs were already being used for television and oscilloscope output, but nobody had thought of combining this technology with computers. The first time they were used to display computer output was in 1951 at MIT (the Massachusetts Institute of Technology) where the Whirlwind computer was developed as a United States Navy flight simulator. CRTs allowed the operators to instantly see the output of a computer program without having to interpret punch cards, rows of lamps or sort through reams of printouts.

While the Whirlwind project itself wasn't very successful due to high operating cost, it was an important step into the direction of computer graphics by introducing the CRT as a viable computer output device. The CRT became an important player in the development of

computer graphics, and remained the output device of choice for over 50 years until it was replaced by newer flat panel technologies.

The First Interactions



Though CRTs allowed computers to display their output, it was mostly text that was simply meant to read out the computer's current state. This remained true for some time since no one thought of the computer in any other fashion than a pure computational device. It was not until 1961 when Ivan Sutherland developed a computer program called Sketchpad for his thesis at MIT that would change the way we look at computers dramatically.

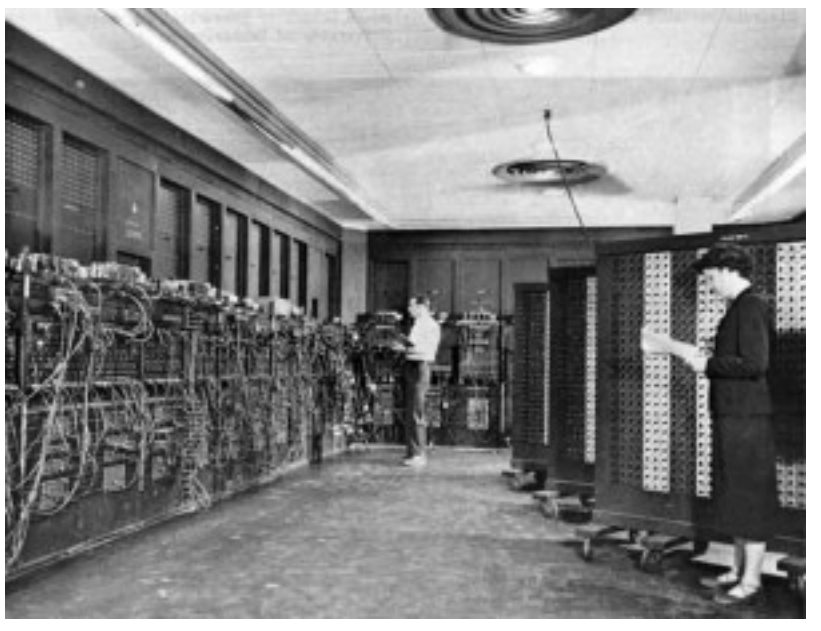
Sutherland's Sketchpad program allowed users to draw geometrical shapes onto a CRT with a light pen in real-time, something that was groundbreaking at the time and remains remarkable even many years after. It not only defined computer graphics, but also introduced the precursor of a GUI (pronounced "gooey," which stands for Graphic User Interface) and laid the foundations of what was to become a concept known as Object Oriented Programming. Sketchpad created a paradigm shift in that computers were no longer simply number crunching devices, but could also be used to display geometric shapes.

> **FYI: Real-Time Computer Graphics**
> Real-time computer graphics are generated on the fly, and usually in response to the user's input from a mouse, keyboard, or any other input device. Real-time graphics are often applied in applications such as video games and design programs.

In 1968, Ivan Sutherland and Bob Sproull engineered another technological feat, namely "The Sword of Damocles," the forerunner of what we now call virtual reality. This system displayed simple three-dimensional wireframe models to the user through a headset, suspended from a ceiling because of its weight. This in itself may have been one of the first (if not the first) time a form of 3D graphics were generated by a computer.

Smaller, Faster, Cheaper

Of course, technology didn't just jump from punch cards to interactive graphics; computers gradually evolved from massive machines to the small devices that you use every day.

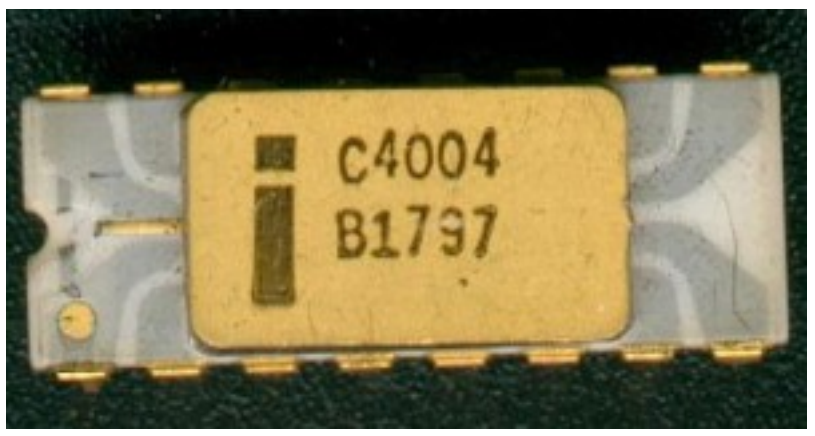


From the 1940s to the mid-1950s, computers used vacuum tubes for processing and would take up entire rooms. A vacuum tube is a device that can modify an electronic signal in some way or another, such as switching, which is a function that is imperative to computing. The components required to assemble a computer were big and ran hot to the touch. None of the machines built during this era were the same, and thus, their programs were not compatible with any other machine. This era is usually referred to as the **first generation** of computers and represents the first step in modern computer science. These are the machines mentioned in the first section that used punch cards and lamps as output devices, so they contributed little to the development of computer graphics.

Transistors started to replace vacuum tubes during the mid 1950s, creating the smaller, faster, cheaper, and more energy efficient computers of the **second generation**. Transistors not only allowed for the creation of smaller computers due to their small size, but ushered in a whole new generation of consumer electronics as well. Radios, for example, could now be powered by batteries and carried around, whereas before they were heavy stationary boxes.

But transistors themselves were not the Holy Grail for computing, and in the mid-1960s, a new technology called the Integrated Circuit brought forth the **third generation** of computers. Integrated Circuits miniaturized a certain function that would normally have been performed by a series of individual transistors onto a single chip. During the third generation, many computers were being equipped with devices such as the keyboard, monitors, and a new type of software called the Operating

System, which allowed the computer to run multiple programs. An important Operating System from this era was called UNIX, which would influence many (if not all) following developments in Operating Systems.



In 1971, a major breakthrough in the field of computing occurred with the invention of the microprocessor by the Intel Corporation, ushering in the **fourth generation** of computers. Whereas CPUs (Central Processing Units) normally were boards with many Integrated Circuits soldered onto it, the Intel's 4004 microprocessor contained all of this functionality on a single chip. Because of a cheaper production process, the computer slowly transitioned from being a specialized device used by large companies and governments to something much more accessible to the masses. We are currently still in the fourth generation of computers, since we have not yet moved away from microprocessors.

Personal Computing

The first Personal Computers started to appear in the mid to late 1970s, but were regarded as enthusiast machines for hobbyists. This changed somewhat with the release of the Apple II by Apple Computer in 1977, and the PET by Commodore International. These machines popularized the concept of computers for the home, but did not have too much to offer in terms of computer graphics.

This somewhat changed in the 1980s when technologies such as GUI were introduced to the personal computing market. The first dedicated graphics add-on cards also started to appear during this era, notably the CGA (Color Graphics Adapter) by IBM was the first color graphics card for the IBM PC platform, which would pave the way for future developments by standardizing a method of drawing computer graphics but didn't offer much in terms of graphics capabilities.

During the 1970s and 1980s, most video games ran on specialized systems, movies made use of computer animation only sparsely, and real-time 3D graphics were for visualization purposes only since there was no consumer hardware that was fast enough. These years were known for their many firsts on consumer hardware, but it wasn't until the late 1980s to the early 1990s when computer games took a strong hold on the PC platform and a real push for better looking and better performing real-time graphics began.

One of the computer games that pushed what was possible on the hardware of the day was Wolfenstein 3D, a first-person shooter released in 1992 by id Software. While Wolfenstein wasn't truly 3D it defined the standard for future 3D computer games. Only a year later, id Software released Doom, their first true 3D game. For the first time, the player could explore their environment through the use of staircases and elevators, without being stuck to a certain elevation. Many games followed that imitated the look and feel of Doom, and were aptly dubbed "Doom-clones." Doom used a software renderer to render its real-time graphics to the screen, as did all of the other games of the early 1990s, but this was about to change.

OpenGL: The First Decade

Silicon Graphics (commonly referred to as SGI) was a company founded in 1981 that specialized in 3D computer graphics and developed software and hardware specifically for this purpose. One software library that SGI developed was IRIS GL (Integrated Raster Imaging System Graphical Library) used for generating 2D and 3D graphics on SGI's high performance workstations. This library was about to evolve into one of the most important computer graphics developments from the 1990s.

In the early 1990s, SGI was the market leader in 3D graphics workstations because of their high performance hardware and easy to use software. IRIS GL was the de facto industry standard 3D graphics library, overshadowing all other developments and attempts to standardize a 3D graphics interface. But despite its popularity, IRIS GL had one major problem: it was a proprietary system fused to SGI's own platforms, and competitors were closing in on SGI's advantage with their own APIs (Application Programming Interface).

In a bold move, SGI cleaned up IRIS GL, removed all functionality that did not relate to computer graphics and released it to the public in 1992 as OpenGL (Open Graphics Library), a cross-platform standardized API for real-time computer graphics.

Software vendors would have to provide their own implementations of the OpenGL standard on their platforms, and hardware vendors programs that allowed OpenGL to talk to the underlying graphics hardware called "device drivers." SGI already provided this to their customers together with a few high-level APIs while other vendors caught up with this new and easy to use API.

Flexibility

Since SGI did not provide any actual source-code, but merely a specification of how the API should work. An abstraction presented itself that allowed hardware and software vendors great freedom on how they chose to implement OpenGL; this level of abstraction is still present today. Because of this, OpenGL is supported across many platforms and devices; in fact, you will be hard-pressed to find a modern platform without at least some level of OpenGL support.

But perhaps the greatest advantage that OpenGL provides to implementers is its support for extensions. If the OpenGL specification does not provide support for specific functionality, the hardware or software vendor may decide to add this functionality themselves through the use of extensions. Many vendors choose to do this and their extensions can be distinguished by their prefixes, e.g. `NV_` for NVIDIA, `AGL_` for Apple, and so on. Extensions can provide powerful functionality, but are usually specific to the vendor's implementation of OpenGL.

You can then call the functionality provided by these extensions by loading them in you program though an extension loading mechanism that retrieves a function pointer. This loading mechanism is however not standardized, so sadly each platform has its own specific extension loading functions. This limitation is most apparent on the Microsoft Windows platform where the OpenGL header files have not been updated since OpenGL version 1.1, even in the latest Windows development kits. There will be more details about why this is so, later in this chapter.

An Open Standard

The name "OpenGL" was not just chosen because it sounded like a fine buzzword, it also contains some actual meaning. Since OpenGL is an evolving specification, someone has to decide what goes in it. So in 1992, the ARB (OpenGL Architecture Review Board) was founded, which comprised of several high profile software and hardware vendors who collectively decided the future of the OpenGL standard through a voting system. Besides determining what new features went into the OpenGL specification, it also decided which extensions would be promoted to become core features of the next OpenGL release.

Although anyone was free to develop an implementation of OpenGL, for it to be recognized as a true OpenGL implementation, the ARB had to approve it through conformance testing. These tests verify any claims made by the implementer of compatibility with a specific OpenGL version through rigorous testing procedures.

> **FYI: Sample Conformance Tests**
> To see some of the conformance testing output, visit Mesa3D.org and click on the Conformance Testing header.

OpenGL quickly became the industry leading real-time graphics API, as it was basically the only one available on multiple platforms.

OpenGL on Windows

OpenGL was already being implemented on UNIX based workstations when Microsoft entered the market with their workstation operating system, Windows NT in 1993. Windows NT was released as a direct competitor to UNIX with networking (the NT acronym meaning Network Technology) and 32-bit hardware support. Windows NT introduced features that are still being used today, such as the Win32 API for creating Windows applications. But having no 3D graphics library native to their system, Microsoft pledged to add support for OpenGL to Windows NT.

Microsoft finally got around to implementing OpenGL in Windows NT 3.5, which was released in 1994, but implemented

OpenGL only to the point that they could claim compatibility by implementing the sample implementation provided by SGI at the time. This sample implementation was meant to serve only as a demonstration of how one could implement OpenGL and as a guideline to vendors. Needless to say, this implementation was dreadfully slow since it was not optimized and graphics accelerators for the PC were virtually nonexistent. In fact, this performance issue was so apparent that there is a Microsoft knowledge base article (KB121282) warning that using NT 3.5's OpenGL screensavers may slow your machine down since it took significant time from the computer's CPU.

DirectX

Seeing a market opportunity in video games, Microsoft sought their own 3D graphics API for Windows to entice game developers to leave DOS behind (Microsoft's first operating system) and develop games purely for Windows. Their first attempt at this was WinG, which simply passed commands to the underlying GDI (Graphics Device Interface) interface and offered no 3D functionality. For this, Microsoft had to acquire a company named RenderMorphics in 1995, who produced a 3D graphics API called Reality Lab. This API was renamed and shipped as Direct3D in an SDK (Software Development Kit) called DirectX that bundled a few other game development-specific APIs as well:

- DirectDraw for drawing fast 2D graphics
- DirectInput for capturing joystick input
- DirectPlay for networking and communication
- DirectSound for sound playback

The first versions of Direct3D were uncomfortable to work with and developers were slow to adopt the API. This caused Microsoft to keep supporting OpenGL while simultaneously putting much effort into making Direct3D a competitive API. The OpenGL 1.1 specification was implemented into Windows 95 and Windows NT 4.0, and it came with a much needed performance boost although it was the last time that Microsoft's implementation of OpenGL would be updated in favor of their own API.

The Beginning of the "API Wars"

While Microsoft had always insisted that OpenGL was best used for "professional graphics," meaning CAD (Computer Aided Design) on the workstation, it was starting to see adoption in video games - an industry that Microsoft sought to dominate on the Windows platform with DirectX. OpenGL's biggest breakthrough came when the influential developer John Carmack of id Software ported his famous video game Quake to use the OpenGL API on Windows and showed developers how easy it was to do so.

In December of 1996, Carmack released a document that outlined his grievances with the Direct3D API. He outlined the differences between the APIs by comparing the code required by both APIs to draw a triangle to the screen; OpenGL required only four lines of code in his samples, while Direct3D required a plethora of commands and assignations.

Carmack's blunt way of explaining things was so damaging to Direct3D's reputation that Direct3D developer Alex St. John posted a follow up in February of 1997 defending his API and strangely enough for Microsoft admitting its flaws. He explained that the Direct3D was designed with direct access to hardware in mind rather than software and that the resulting interface may not have been pretty, but it got the job done. And once again, Microsoft pushed the point that OpenGL was a CAD library and wouldn't be supported on consumer type hardware anytime soon.

This response rattled SGI's cage, and in June of 1997, they came out with their own response to St. John's critique. This lengthy document outlined "some of the most notable deficiencies of the design and current implementation of Direct3D." It noted the differences between the two APIs in technical terms rather than marketing ones and elaborated on Carmack's mention of ease-

of-use with more code samples.

FYI: The Back-and-Forth

1. John Carmack's .plan file on OpenGL.
2. Alex St. John's write-up on Direct3D, OpenGL, and John Carmack.
3. The SGI document mentioned above is now only available through Archive.org.

For even more reading on this and the Microsoft culture at the time and Alex St. John, pick up the excellent book *Renegades of the Empire* or check out his blog at AlexStJohn.com where he occasionally talks about his time at Microsoft as well as the still ongoing OpenGL vs. Direct3D debate.

Direct3D became more usable with version 5.0, which removed some of the uncomfortable features from the API. At this point, both APIs were quite user-friendly and similar in feature set, but this status quo was about to change.

Driver Debacle

With OpenGL and Direct3D being at a stalemate on Windows NT, Microsoft needed an edge. OpenGL drivers were implemented in Windows NT by using a Mini-Client Driver (MCD), which was a low performing compromise between hardware and software, but the easiest solution for creating drivers. The MCD allowed vendors to pick and choose pieces that they wanted to accelerate on their hardware while the rest would run on the provided software implementation (or vice versa). Microsoft obtained the edge they needed by not allowing the licensing of MCDs on Windows 95, thus effectively limiting OpenGL only to the software implementation provided by Microsoft.

This decision was a massive blowback for OpenGL that was trying to find a way into the consumer market, not to mention hardware vendors who had been implementing these drivers for months. Thankfully, SGI provided a solution that would bring hardware drivers to Windows 95 called the Installable Client Driver (ICD). In fact, this type of implementation was several magnitudes faster than the MCD driver model, so it proved to be quite a blessing in disguise. Hardware vendors jumped on this opportunity and quickly began supplying drivers. Not long after, game developers started implementing OpenGL into their games, proving it was once again a viable alternative to Direct3D.

Hardware Evolution

In the late 1990s, OpenGL established itself as an industry standard for 3D computer graphics, not just for CAD programs, although it was the only contender in that market. PC video games such as Quake 2, Unreal, and Half-Life took full advantage of OpenGL to show off their full potential and were widely popular. Around this time, the first consumer-grade dedicated 3D graphics hardware started to appear, changing the video game industry forever.

One of the first 3D accelerators was the Voodoo Graphics by 3Dfx Interactive, a high performance add-on card that set the standard as soon as it hit the market. While there were other add-on cards such as the ATI 3D Rage and the S3 ViRGE, the 3Dfx card blew them all out of the water both performance and feature wise. On top of that, 3Dfx provided its own 3D graphics API called Glide, which had direct access to the underlying graphics hardware. At the time, Glide was the fastest API in town, but due to the fact that it was vendor-specific, it was made obsolete by competing APIs only a few years later. Nevertheless, Glide made its impact on the industry, causing the other APIs to play catch-up for quite a while.

NVIDIA soon caught up in 1999 with their GeForce 256 add-on card that they termed GPU (Graphics Processing Unit), which supported a brand new technology called Transform & Lighting (commonly referred to as T&L). T&L moved the vertex transformation calculations and lighting calculations from the computer's CPU to the GPU. The main advantage of a GPU is that it does floating-point operations very quickly since the hardware is dedicated to this task, while a CPU specializes in integer and more general-purpose operations. 3Dfx never implemented T&L, which eventually contributed to their demise as more software functionality was being moved over to the GPU.

After 3Dfx went bankrupt, NVIDIA acquired much of its intellectual property (including the well-known SLI technology) but did not continue the Voodoo product line, nor did they support any of 3Dfx's old products. By the year 2000, the only

competitors remaining in the GPU markets were NVIDIA with GeForce 2, and ATI with their Radeon 7000 series of GPUs. These two vendors only offered support for OpenGL and Direct3D, clearing the playing field for these APIs to go head to head.

FYI: Software vs. Hardware
When we say something is software functionality, it means that it is executed on the CPU instead of the GPU. When we say something is hardware functionality, it means that the feature is executed on dedicated hardware. For example, software rendering is done solely through the CPU and hardware rendering is done solely on the GPU.

Paradigm Shifts

During the early 2000s, GPU performance grew exponentially as more software features were moved to the GPU. The CPU became obsolete for rendering real-time 3D graphics since it could not keep up with GPU developments. In fact, the current method of rendering 3D graphics saw the CPU as such major bottleneck that new methods had to be invented to circumvent its use.

Buffers

To get things to render on the screen up until this point, programmers issued lists of commands from their program that would be interpreted by the GPU, called the immediate mode. This methodology performed fine for smaller data sets, but with larger data sets, performance was hindered by the performance of the CPU since all function calls originated from the program itself.

The new method came in the form of buffer objects. Buffers had been around for a while in the form of display lists and vertex arrays, but they each had their drawbacks. Display lists still used the immediate mode, and vertex arrays were stored in the system's memory so they had to be transferred to the GPU every single call.

Instead, the new buffer objects would be stored in the GPU's memory after initialization and would stay there until they were no longer needed. In OpenGL, these objects are called Vertex Buffer Objects (VBOs), and in Direct3D, they are called Vertex Buffers. You will learn much about VBOs in a future chapter, which will introduce you to VBOs and why exactly they are so fast.

FYI: Buffers
In computer science, a buffer is a place in memory where temporary data is stored. When we are done using this data, the buffer is deleted and the memory is ready for reuse.

Shaders

In the year 2000, Microsoft released Direct3D 8.0, which supported a new feature called shaders. Shaders are basically nothing more than little programs that run directly on the GPU, thus leveraging even more of the GPU's power and moving more functionality away from the CPU. When Direct3D 8.0 was released, two types of shaders were announced, namely vertex shaders and pixel shaders.

A vertex shader is a GPU program that is executed once per vertex that is assigned to, and a pixel shader is a GPU program that is executed once per pixel. Shaders allowed for greater programmability and performance by eliminating the CPU for many tasks, but they were very hard to program due to their syntax, which resembled the Assembly programming language for the CPU.

Microsoft recognised this shortcoming, and in 2003, a major breakthrough in shaders came in the form of the High-Level Shader Language (HLSL) with the release of Direct3D 9.0. This new language allowed the manipulation of shaders in a high-level language whose syntax was based on C. At this point, shaders became much more viable to use and adoption was widespread. You'll learn much more about vertex and pixel shaders in future chapters, where we'll go in-depth about how all of this is achieved and how you can use shaders in your own programs.

OpenGL Stagnates

The above paragraph makes no mention of OpenGL for a good reason, namely that OpenGL didn't support any shaders at the time. OpenGL did not officially support shaders until the year 2004 with the release of OpenGL 2.0 and the simultaneous release of the OpenGL Shading Language (GLSL). Even though the extensions to use shaders were widely available before 2004, they were not part of the official specification and took several years to implement properly.

OpenGL had fallen behind Direct3D dramatically in terms of core features. Just as Direct3D had been playing catch-up in the late 90's, OpenGL now had to catch-up to Direct3D, which didn't happen. From 2004 to 2006, Direct3D 9.0 was dominating the market and only a few games were released with OpenGL support. Support for Direct3D was increased even more so when the Xbox 360 was released in 2005, with support for Direct3D 9.0. In the meantime, there was no news whatsoever from the ARB, and it seemed like OpenGL was truly dead for a while.

In 2006, OpenGL 2.1 was released as a minor increment to the original 2.0 specification, and brought only a handful of new features. To add insult to injury, Microsoft released Direct3D 10.0 alongside their new Windows Vista operating system, which included a major API overhaul and many new features. Hardware started to move in a new direction, away from the immediate-mode, fixed-function methodology to a more programmable direction, which OpenGL lacked.

Meanwhile, OpenGL developer community was getting restless and demanded an answer from the ARB or SGI, but what they got was something entirely different.

The New OpenGL

It was announced at the 2006 SIGGRAPH that OpenGL would be managed by the Khronos Group in the future instead of SGI, who still owned OpenGL and all the associated copyrights, but it would no longer manage it. The Khronos Group is a consortium of hardware and software vendors with a vested interest in OpenGL that focuses on the creation and maintenance of open standards APIs, most notably before they acquired OpenGL, the COLLADA file format for 3D content. Finally, there was some news from the OpenGL front, and for the first time in two years, there was rumour of a brand new version of the OpenGL API that would bring some major changes.

Longs Peak and Mt. Evans

Two new working versions of OpenGL were announced under the temporary codenames "Longs Peak" and "Mt. Evans" after mountains in Colorado. Longs Peak would be the first specification released in the summer of 2007, and Mt. Evans a few months later in October of that year.

These revisions promised a brand new API that was able to compete with the Direct3D 10 API, and much like Direct3D 10 had done, Mt. Evans would eradicate immediate mode rendering and rely solely on buffers and shaders. This API rewrite was a grand undertaking and required several Technical Sub-Groups, or TSGs, that focused on their own specialized area of the OpenGL specification.

One of these was the Object Model TSG, which dealt with how buffers and other types of objects would be represented in the new API. The proposed Object Model proposed a wonderful new method for creating objects through only a few function calls. Above all else, the methods used would be consistent for all types of objects through the use of templates. This meant that there would be no more discrepancies between vendors who would each provide their object creation functionality in a myriad of ways.

Longs Peak would be an API compatible with the hardware of the time and preserve backwards compatibility with older versions of OpenGL, while Mt. Evans would eliminate backwards compatibility and take a future-forward stance. This together with the Object Model became the staple of the proposed API, drumming up much anticipation from the OpenGL community.

But the summer of 2007 came and went by without word from Khronos, and on October 30th, word came out that the new specification was delayed. The community was a bit disgruntled, but the overall consensus was that the new specification was worth the wait, and another year went by.

OpenGL 3.0

It had been two years since the minor release of OpenGL 2.1, and four years since the last major release when OpenGL 3.0 was

released in July of 2008. Reading the specification, it didn't take long to notice that this was *not* Longs Peak. In fact, it looked like not much had changed at all: the immediate mode was still there, the proposed object model was missing, and there were no plans to include it in any future releases of OpenGL. Some new features were introduced together with something called the deprecation model.

The deprecation model tagged all of the immediate mode functionality as deprecated in favour of methods that are more modern. However, there were no plans to remove any of the deprecated functionality leaving the OpenGL 3.0 specification a fully backwards compatible API with all of the crippling features of the past.

The community was outraged and protested very vocally on the OpenGL community message boards and elsewhere online. Unfounded accusations were made that OpenGL remained backward compatible because Khronos didn't want to lose their CAD customers who still used the immediate mode and refused to move on. Many Windows developers started to leave OpenGL behind in favour of Direct3D, including several of the accused CAD software developers. If the future looked bleak for OpenGL before, it certainly seemed as if OpenGL had lost the API wars once and for all with this release.

After the Shock

But after the initial disappointment had passed, the new specification proved to have some qualities that Direct3D didn't posses. For instance, OpenGL 3.0 contained many of Direct3D 10's features, but was able to access them on Windows XP whereas Direct3D 10 required Windows Vista to function because of a new driver model.

About a year later in March of 2009, OpenGL 3.1 was released, which finally removed all of the immediate mode functionality from the OpenGL specification that 3.0 had marked as deprecated, thus bringing it one step closer to the API promised in Longs Peak and Mt. Evans. With this fast release, OpenGL was finally back on the right path and only a few months later, OpenGL 3.2 was released, bringing the API up to par with Direct3D 10 by including Geometry Shaders. You'll read more about geometry shaders in a future chapter.

Deprecation, Core, and Compatibility

The OpenGL deprecation model can prove to be a bit confusing, so in this section I'll try to explain the associated terms once and for all. OpenGL 3.0 introduced a feature called deprecation that "marked" old and unwanted OpenGL functionality and warned that these features may be removed in future specifications; basically, using deprecated features in your program is not a good idea moving forward.

When a developer wants to use OpenGL in their programs, they need to create a so-called context, which is basically nothing more than an object that allows developers to pass commands to an OpenGL device. In the past, these contexts were all created in the same manner regardless of the implemented OpenGL version.

This meant that when you created a device that used OpenGL 1.5 features, and the driver returned an OpenGL 2.0 device, it would not be a harmful since all context were fully backwards compatible. OpenGL 3.0 introduced a new method of creating contexts asking of the following parameters at context creation:

- A major version number
- A minor version number
- Optional attributes

In the case of an OpenGL 3.0 context, the corresponding values would be 3, 0, and some combination of attributes. This new functionality provides a guarantee that the device returned is the requested OpenGL device or nothing, which means that the version is unsupported. The attributes that can be passed make the selection even more granular with one of the following:

- A Core Profile context flag
- A Compatibility Profile context flag
- A Forward Compatible flag
- A Debug flag

The difference between the two profiles is that the Core Profile does not include any of the features that were removed in previous versions, and the Compatibility Profile does include them. An OpenGL implementation is always guaranteed to

contain the Core Profile of the specification, but not always a Compatibility profile. Moving forward with a Core Profile is the most logical thing to do, and the intention of this type of context creation.

If you set the Forward Compatible flag, the context that is returned will not contain any of the features that were deprecated in the version that you requested, thus making it compatible with future versions that may have removed these features.

If you set the Debug flag, a debug context will be returned that will include additional checking, validation, and other functionality that can be useful during the development cycle.

The above flags can all be combined together, with exception of the Core and Compatibility, since only one of these can be returned. Combining flags is done using the bitwise `OR` operator, the pipe symbol (|) in C.

If these terms are still a bit fuzzy to you, don't worry, we'll go over them again in the appendices where we set up OpenGL contexts from scratch on specific platforms. For now, remember that there are two types of OpenGL profiles, Core, the more modern profile, and Compatibility, the profile that is compatible with older OpenGL functionality.

OpenGL 4.0

A year after the release of OpenGL 3.2, OpenGL 4.0 was released as an API for the latest generation of GPUs similar to Direct3D 11. Simultaneously, OpenGL 3.3 was released implementing as many features from OpenGL 4.0 as possible while remaining compatible with previous generation hardware.

An important new feature in OpenGL 4.0 is called Tessellation, which allows for fine-grained control of surfaces and automated levels of detail in your scenes. We will explore what Tessellation is and how to use it in a future chapter.

This book uses the OpenGL 4.0 specification with the Core Profile enabled, which means that it is not compatible with any previous versions of the OpenGL API. We are also not going to use any deprecated features in order to stay as future-compatible as possible. This may seem a bit daunting at first, but rest assured that it's not as difficult to learn as it may seem, and not learning old features will be beneficial in the long run.

If you already know some immediate mode OpenGL, please be aware that we are not going to cover this type of OpenGL. You will not find commands such as `glBegin`, `glEnd`, `glVertex3f`, and `glColor3f` in this book (besides these references) since they are not present in the OpenGL 4.0 Core Profile. Forget everything that you've learned about immediate mode OpenGL up to this point and that the immediate mode API ever existed, since it is never coming back.

It's a great time to start learning OpenGL since there is a major move of video games being ported over to platforms other than Microsoft Windows, and the only way you'll get real-time computer graphics on platforms other than Windows is through OpenGL. For instance, Half-Life developer Valve has ported many of their popular games over to the Apple Macintosh, using OpenGL as their graphics library.

In addition, modern smart-phones such as the iPhone and Android-based phones all use **OpenGL ES** for interactive 3D graphics, which is an API for embedded systems based on, and very similar to OpenGL. This means that potentially, your code could be portable enough to run on PCs, Macs, consoles, as well as on various mobile devices.

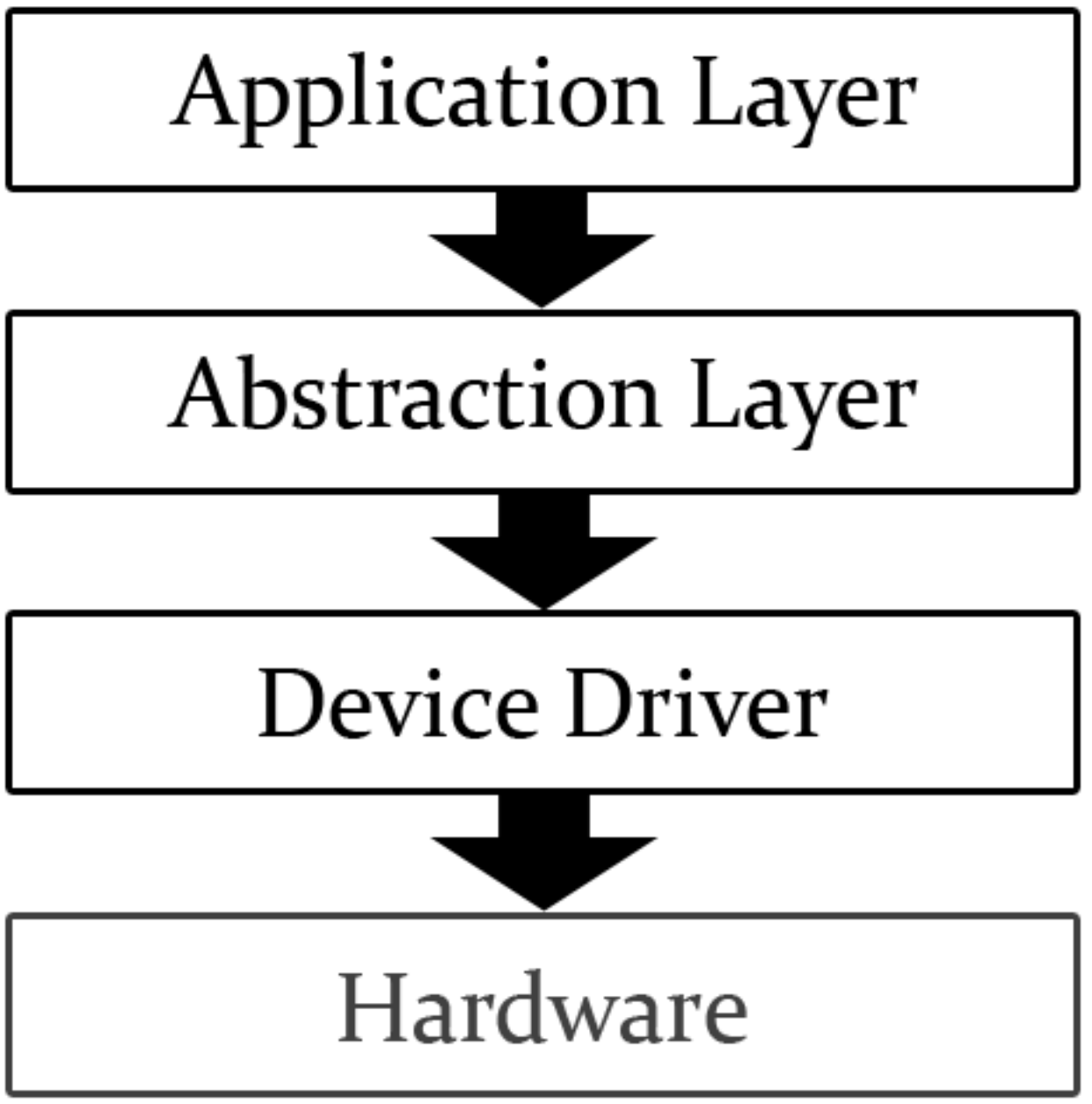
OpenGL ES itself has spawned off yet another API called **WebGL**, a cross-browser and cross-platform compatible 3D graphics API for the web browser that is gaining more traction by the day. This library has the potential to move the web as well as multi-user applications to a whole new level.

All in all, OpenGL is far from dead and thriving as a full-featured modern 3D graphics API.

The Software Pipeline

In rendering real-time computer graphics, the software pipeline exists to describe what we'd like to see on the screen. For example, if we'd like to display a green square on the screen, computer software would describe with which dimensions, color, and at which position of the screen to draw the square.

The software pipeline also provides access to functionality that draws the geometry onto the screen. It is worth noting that the software pipeline does not actually do any drawing or transformations, since on modern systems this functionality is entirely implemented by the hardware.



The software pipeline consists of several different layers, each with their own very specific purposes. Each of these layers may contain a myriad of functionality, but for the sake of brevity and clarity, we'll only discuss the high-level functions of each layer.

The first is the **Application** layer, which is your program, 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, and making calls to external libraries such as OpenGL or Direct3D through their respective interfaces.

The next layer is the **Abstraction** layer, which contains the OpenGL or Direct3D API implementations. It is important to make the distinction between the API in the Application layer and the implementations of OpenGL and Direct3D in the abstraction layer. To put it 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. The Abstraction layer serves as a dispatch to the next layer by implementing hardware-level functionality in a usable and standardized format.

The abstraction layer passes its commands to the **Device Driver**, a software communication layer to the hardware. This layer is entirely invisible to the developer, since it cannot be interacted with through your program. Like many invisible things (remember the bacteria), the device driver is one of the most important parts of the Software Pipeline, since it connects all of the individual pieces together. Because of the amount of specialized functionality that the Device Driver encompasses, it can be quite large in file size.

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.

In essence, the software pipeline serves as a relay from your program to the dedicated hardware. While very important, 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.

The hardware pipeline will be explained in-depth in the another chapter, when we will actually set up an OpenGL context and a window for rendering purposes.

Requirements

Now that we're ready to start, we have to make sure that you're prepared for programming with OpenGL 4.0. Naturally, you need basic knowledge of the C programming language and know how to link to a library with your compiler.

If you don't know the C programming language, there are many books that will teach you how to use the C programming language in a very easy to follow manner. In fact, C is not a very difficult language to learn. I don't expect you to master it in

order to use the book, but you will need to understand the basics such as pointers, data structures, and functions. So, once you've learned some C, or while you're learning it, you are very welcome to return to the book and resume from this point.

If you don't know how to link to a library, you can consult your compiler's documentation. Linking a library is usually nothing more than setting a few command line parameters or options in a GUI.

Last but not least, some basic math skills are required since we will be handling matrices, vectors, and geometry. The samples in the book will not be excessively heavy on math, but eventually, some math will be required.

System Requirements

To be able to run the examples in this book, your GPU must support OpenGL 4.0 in its entirety.

- If you own an **NVIDIA** GPU, it must at least be in the Fermi family of products, starting with the GeForce GTX 400 series.
- If you own an **AMD/ATI** GPU, it must at least be in the Radeon HD 5000 series.

If you're not sure about your GPU, refer to the manufacturer's website for more information and the latest device drivers.

Finally, if your GPU does not support OpenGL 4.0, you will not be able to run the samples provided in this book, at which point you must upgrade your hardware to be able to continue. However, many of the concepts discussed are available in earlier versions of OpenGL, you just will not be able to copy the code samples without rewriting them for your supported version. Also, please make sure that you have the latest device drivers for your graphics hardware since they usually contain many bug fixes and performance improvements.

Software Requirements

I have tried to mention only easy to acquire, free and Open Source software in the following sections so that you can get started with OpenGL quickly and anywhere you have an internet connection. The main thought behind OpenGL is portability, so the code mentioned in the book will be portable across multiple platforms as well.

What this all means for you is if you ever decide to use another operating system, or simply wish to develop OpenGL programs for other operating systems, the samples in this book will still work. Platform specific samples can be found Appendix A, but I cannot recommend them if you are not familiar with OpenGL, C, and your operating system's API.

The most important piece of software that you will need for this book is a C compiler which will transform the C code from its human readable text format into a binary format that can be interpreted and executed by your machine.

If you use **Microsoft Windows**, you can get a free version of Visual C++ from Microsoft's website called Visual C++ Express. This program contains a modern full-featured C and C++ development environment, based on the more complete Visual Studio: http://www.microsoft.com/express/windows/

If you use **Linux**, chances are that you already have a C compiler on your system, called GCC (the GNU Compiler Collection). This is only a compiler, meaning that you will have to compile your programs through a command line interface instead of a development environment. http://gcc.gnu.org/

Another option for compiling C and C++ code on Linux is the Intel C++ compiler, which is available for free for non-commercial usage from the Intel site: http://software.intel.com/en-us/intel-compilers/

If you're on a **Macintosh**, you can use Xcode, which is a free download from the App Store. Like Visual C++, Xcode is a full-fledged development environment, but focused on Objective-C, instead of C and C++. https://itunes.apple.com/us/app/xcode/id497799835

If you're on any other platform, please consult your operating system's vendor for details on where to acquire a C compiler, or use a search engine to find out for yourself.

IDEs (Integrated Development Environments) are a great way to organize your projects and can help you code faster. A good development environment for many operating systems is Eclipse, which supports C and C++, as well as a several other programming languages. You can get the Eclipse development environment for free at eclipse.org

Of course, you don't *have* to use an IDE, there are many text editors out there that are great for editing C code such as Vim,

Emacs, Notepad++, or Sublime Text.

Besides a compiler, make sure that you have support for OpenGL by referring to the compiler's documentation, or checking the include directories for the header file `gl.h`, and the library directories for the OpenGL library, usually named `opengl.lib`, or `opengl.so`, depending on your platform. If you can't find these files, or if you're just not sure, refer to your compiler documentation or vendor website for support.

Other than a C compiler and OpenGL support, there are three additional libraries that are required for the examples in this book.

FreeGLUT

Since OpenGL is merely a graphics library, window and context creation must be handled by an external library, usually provided by the operating system. But since you could be using a different operating system than me, a cross-platform library must be used, and FreeGLUT is an Open Source library that does exactly that.

Modeled after the long abandoned but still popular GLUT library (OpenGL Utility Toolkit), FreeGLUT provides a modern Open Sourced alternative that is easy to use, cross platform compatible, and suitable for creating demonstrative programs such as the ones in this book.

You can obtain a copy of FreeGLUT at freeglut.sf.net, please make sure that you get version 2.6.0 or higher in order to be able to create an OpenGL 4.0 context. FreeGLUT is licensed under the X-Consortium license, meaning that you can use it in any program even if it's proprietary.

GLEW

Loading extensions can be quite a platform-dependent hassle, so the next library you will need is the GLEW library (OpenGL Extension Wrangler), which makes it a breeze to use OpenGL extensions in your programs. If you worry that we're skipping an important part of OpenGL by using this 3rd party library, Appendix A will explore extension loading in detail for several platforms.

You can get a copy of the GLEW library at glew.sf.net for free, as it is also Open Source and licensed under several unrestricted licenses that allow you to use GLEW in any code base, similar to the license that FreeGLUT uses. Please make sure to get version 1.5.4 or higher for OpenGL 4.0 support.

Setting up Guides

Setting up OpenGL, GLEW, and FreeGLUT in Visual C++

Other Libraries

Any other library dependencies will be listed at the beginning of each chapter.

Conclusion

Now that we've covered most of the background on computer graphics and OpenGL, and know the requirement for continuing, we're ready to get our hands dirty and set up an OpenGL rendering context in the next chapter.

Media Sources and Attribution

1 Source, License: Creative Commons Attribution-Share Alike 3.0 Unported

2 Source, License: Creative Commons Attribution-ShareAlike 2.0 Generic

3 Source, License: Creative Commons Attribution 3.0 Unported

4 Source, License: Creative Commons Attribution 2.0 Generic

5 Source, License: Creative Commons Attribution-Share Alike 2.0 Generic