Direct3D vs. OpenGL: Which API to Use When, Where, and Why

I have tried to keep this article as devoid of error and opinion as possible. All of my opinions are marked as such, and all errors are purely accidental. Please excuse any minor errors that I may make. I am of the opinion that everything is correct; all facts have been confirmed by third party sources. Please also understand that this article is written for beginners who are unsure how to start graphics programming, and that while I have tried my best to keep this article neutral, this issue is very debatable.

For many years now, the question "Which should I use: Direct3D or OpenGL?" has elicited heated arguments. Most of the things said do very little to help you decide which API you should use. There are fanatics on both sides; some people will tell you that OpenGL is the only way to go, and others will swear by Direct3D. Others will talk about how they have used both, and that you should do the same and decide which you like better, but for someone starting out, that's not much help either. The point is, it can be a tough decision to make, and I am writing this article to help you do it. My specialty is primarily in Win32 programming, so that will be my focus. No need to fear though, I discuss other platforms as well.

In this article, I concentrate on only two API's: OpenGL (version 1.3) and Direct3D (version 8.1). I don't have enough experience with any other API to discuss it in detail, but I'll at least comment on Glide later. I'll attempt to present you with the hard facts about each API, their structure, and what kind of code you can expect. I do not, however, actually include any code here. You can find that easily enough in other areas on GameDev.net. There really are no prerequisites to understanding this article (except possibly knowing English). Thus far, I've assumed that you know what an API is, and what purpose APIs serve. In case you don't know this, API stands for Application Programming Interface, and it's essentially a set of functions that allow you to perform some specific task. For our purposes, that task is interfacing with the graphics hardware.

The Hard Facts: Direct3D8

For a long time, Direct3D was considered to be pretty bad compared to OpenGL. Recent advancements in the API, however, have made it very powerful and stable. Many people now believe that Direct3D is the standard for graphics on Windows platforms, not OpenGL. Microsoft works very closely with graphics hardware companies to make sure that any new features that they introduce will be supported in Direct3D. Often, Direct3D supports features before cards do.

History

Well after the debut of Windows 95, the overwhelming majority of games were still being made for DOS. Microsoft wanted game developers to move away from DOS and make games for Windows, in turn making Windows a more popular platform. Windows, however, did not provide a good gaming platform. The many layers of abstraction meant that access to video and sound hardware was very limited and slow. They decided to create an API that would allow game developers more direct access to the hardware to allow games on Windows to run at acceptable speeds.

Rather than develop their own 3D API from scratch, they noticed a very promising 3D API being developed by a company called RenderMorphics. It was a small project the company had written and was showing at a trade show when Microsoft discovered it. (As an interesting side note, the API was submitted by the creators as a college project and flunked for having deviated from the assignment somewhat.) They integrated it into their own sort of mini-Graphics Library known as the Game SDK. They expected it to be the perfect solution for game developers. They were wrong.

What later became known as DirectX 1.0 ended up not being very widely accepted. It was buggy, slow, badly structured, and overly complex.

Of course, Microsoft wasn't about to just give up. They kept working at it, asking the community for ways to improve it. The first truly viable version of DirectX was DirectX 3.0. A few years later, they released DirectX 5 (skipping 4 entirely), which was the first truly useful version. Incremental improvements were made with version 6. Then came DirectX 7.0.

DirectX 7 was the first one to actually be embraced by game developers. It worked well, making game programming reasonably easy, and a lot of people liked the interface. Now here's where the big one hits. Up to and including DirectX 7, DirectX included two components for graphics: DirectDraw and Direct3D. DirectDraw was pretty much 2D only, but quite powerful. Direct3D was built on top of DirectDraw.

With DirectX8, the deprecated DirectDraw interface, which hadn't changed much for several versions, finally got thrown out as a separate component. Although this was a controversial action, Microsoft wanted to focus on the 3D aspects of the API. Most things that are done in 2D can be duplicated using 3D techniques, with the advantage of being faster due to the use of hardware acceleration. In addition, the older DirectDraw interfaces would still be available due to the nature of COM. And that's pretty much where we are today.

Structure

The structure of Direct3D, and DirectX in general, is hugely different from OpenGL. As time goes on however, DirectX is slowly becoming more and more like OpenGL. I won't show all the class diagrams here, but basically, DirectX is based on the COM object model. What this means is that you create pointers to classes and use those, rather than just calling functions that don't clearly show associations.

A COM object is analogous to a C++ class. You can even derive from the DirectX classes, although doing so is one of the most profoundly stupid things anyone could ever do. A typical DirectX app will have a few structures such as LPDIRECT3DDEVICE8. Using the - notation, you can call members of these classes. Unfortunately, writing DirectX applications in pure C is not the most fun thing to do. Because of the way COM works, you must explicitly dereference a VTable, as well as pass a pointer to the interface, both of which are done implicitly in C++. What this means is that instead of writing D3DDevice->DoSomething(Params), you have to write D3DDevice-lpVtbl-DoSomething(D3DDevice, Params). Not pretty. Microsoft realized this and made some macros to ease the paid, but the equivalent macros are often just as bad (for example, IDirect3DDevice8_DoSomething(D3DDevice, Params).

DirectX apps typically make heavy use of C structs, and often your app will be build around nice custom classes. The rendering is done in between two calls: Direct3DDevice8-BeginScene() and Direct3DDevice8-EndScene(), a format which was "borrowed" from OpenGL. When you shut down your program, you follow a certain pattern: you call X->Release, where X is a COM object, to free resources allocated for the object.

DirectX also provides something called the Direct3DX function library, introduced with DirectX 7.0. This is a static library that provides gobs of useful functions that help make your life easier. There is one catch, though: many of the things that have been added are C++ specific. D3DXMATRIX is derived from D3DMATRIX, for example, and many of the operators are overloaded as well. Of course, if you're using C++ anyway, this really isn't a problem. The Direct3DX library provides a lot of useful functions which have been very heavily optimized. Amongst other things, it will load and save meshes, do matrix and vector math, provide over a hundred C++ enhanced data structures, and manage things such as 3D fonts.

The last thing to be mentioned is the DirectX Application Framework, now known as the DirectX Common Files. As I mentioned earlier, even drawing a simple triangle takes a lot of code. To help reduce the amount of code required to perform even simple operations, Microsoft wrote a set of classes that do most of it for you. All the initialization is hidden from you. Even somewhat complicated operations such as listing all the capabilities of the graphics card are handled by the Common Files. The source is available to, so that you can edit it as you like. Of course, because they are built around classes, you need to be using C++ to take advantage of them. I recommend that you use the Common Files, but be aware of how they work. A good way to do this is to use the DirectX AppWizard to create a small D3D project and step through it, watching exactly how the different functions/classes interact.
Looking at the Header

If you've never used COM, the contents of the d3d8.h header may seem foreign to you. Most reactions lie along the lines of "Huh?" or "Wha?!". I spent a long time looking through objbase.h to figure it out. (objbase.h is the header for COM objects) Abstract classes are declared, along with all their methods. The syntax for this is horrible, so you probably don't want to even bother looking at it. The COM stuff in objbase.h knows to look for the implementation in a DLL that has been registered with Windows. Don't worry about registration of the DirectX COM components; that happened automatically when you installed your DirectX runtime. In addition to all the COM declarations you'll

find a lot of #defines and #includes of other headers. The header d3d8types.h, for example, defines most of the data structures used for DirectX.
Strengths

Direct3D does some things extremely well. Two of its major strengths, added with DirectX 8, are Programable Pixel and Vertex Shaders. I won't go into detail about these, but suffice to say that they allow you to replace portions of the rendering pipeline with custom code. Shaders are written with a sort of graphics language that looks very much like assembly code. (Note that OpenGL supports vertex and pixel shaders as well, but it does so through extensions, and at present, graphics vendors have not agreed on a standard way of using them). On the other hand, only the newest graphics cards (such as the GeForce 3 or 4 or Radeon 8500) support pixel shaders. (Some older cards support vertex shaders).

Direct3D provides functions to enumerate exactly what graphics hardware is available on a system.

Direct3D's syntax and structure have evolved to a format that many developers believe is more intuitive than previous versions, making it easier to develop for than it used to be.

If you like object oriented programming, and COM in particular, you'll be right at home with Direct3D's interface.

The fact that DirectX uses COM provides a proven method to introduce change without breaking existing code. As a rule, a COM interface cannot change. While on the surface this may seem like a bad thing, it's really not, for two reasons. One, although a given interface cannot change, a new version of an interface can be created, which can change in any way desired. Two, since an interface can never be removed either, new versions of DirectX won't break games written for older versions. The old interface is still there, unchanged, and can still be used.
Weaknesses

DirectX is only updated every year or so, which is a little slow considering how fast the graphics industry moves. They make up for this somewhat by working closely with graphics vendors and adding support for things aren't yet available. Also, when new features are introduced, Direct3D offers a standardized way of accessing them.

D3D requires much more code to initialize the thing. With Direct3D8, the effect has been greatly reduced; it now takes about 200 lines to draw your first triangle, as opposed to 800 lines in Direct3D7. Also, D3D will require one to know much more about how everything works. To begin D3D will take more learning than OGL, but there will be nothing that you won't need to know in OGL eventually either. D3D also has very little portability. How much? None, really. How many times can you say "Windows"? Yep, that's all it does. Windows. Although this includes the X-Box (which will still require platform specific code, and isn't really an option for hobbyist or beginning developers anyway), it can be a real killer for people who want to write multi-platform code.

Direct3D is not an open standard. That means that Microsoft, and only Microsoft, has the final say about what gets included in a release, and if they make a bad decision, it stays that way for about a year. How successful Microsoft has been at "getting it right" is completely a matter of opinion.

Finally, writing a DirectX game, or using any COM component, in pure C can be difficult and lead to frustration. As you saw earlier, the C code for Direct3D is hardly intuitive.

Language Support

One of COM's specifications is that it is language independent. For a long time though, languages other than C/C++ were not supported. With DirectX 7.0, "out of the box" support was included for VB (it doesn't support pointers and such, so a new dll called dx7vb.dll was required.). As far as I know, D3D supports most major programming languages. (People seem to be very unsure whether Java can interface directly with Direct3D OR OpenGL; there is a wrapper called Java3D that allows you to interface with both that most people use anyway.)

The Future

There is not much to look forward to, really. I doubt Microsoft will ever make DirectX available on other platforms. As far as new features, Microsoft will probably continue to keep pace with hardware vendors, adding support for new common hardware features, but doing little to innovate.

Comments/My Advice

Direct3D is very useful for making Win32 games. It is made solely for Win32. Because Direct3D is only for Windows systems, it is usually not used for high-end graphics applications. Many graphics people prefer a Unix or SGI workstation; Direct3D does not (and probably never will) support those platforms. There is really nothing that Direct3D can do that OpenGL can't, but it has the advantage that you don't have to write vendor-specific code to take advantage of the latest hardware features.

Direct3D is a lot more complicated because it does not totally hide the lower level things from you, like OpenGL does. Obviously, simplicity results in a loss of flexibility. But is it flexibility you actually need? There are many more things to consider than you might have guessed.

OpenGL: The Hard Facts

OpenGL is popular among developers in many different industries. It provides a great deal of functionality, and has proven to be a stable API over the 10+ years of its existence. It was written about 10 years ago to provide functionality for the future as far as possible. Unfortunately, that future has come and gone.

History

OpenGL was originally developed in 1992 by Silicon Graphics, as a descendant of an API known as Iris GL for UNIX. It was created as an open standard (not open source, as some people incorrectly believe), and is available on many different platforms. (Now that you know what "Open" refers to, you may also wonder what the "GL" stands for; not surprisingly, it's "Graphics Library".) OpenGL is overseen by a committee known as the Architecural Review Board (ARB). The ARB consists of representatives from major companies involved with the graphics industry, including 3D Labs, SGI, Apple, nVidia, ATI, Intel, id Software, and yes, even Microsoft.

There are two major implementations of OpenGL for Windows: One from SGI version and one from Microsoft. The Microsoft version is based on the SGI implementation. Since the latter is no longer supported, it is recommended that you use the Microsoft version. It corresponds to OpenGL 1.1, but there are no newer headers and libraries available anywhere else (you can, however, access newer OpenGL features using the extension mechanism, which is beyond the scope of this article).

There is another, unofficial OpenGL implementation available known as Mesa3D. I haven't actually tried it, but it is very stable, it's open-source, and it's available on many platforms. The creators of Mesa3D do not own an OpenGL license, so it cannot legally be called an OpenGL implementation, but it uses the same syntax. The only disadvantage to it is that it cannot always take full advantage of hardware.

OpenGL has been used for many years in all different sectors of the tech industry. It was developed from the beginning with a clear vision of the future, and as such, it has remained stable and consistent. On the other hand, the way the ARB operates has caused the core OpenGL specification to evolve relatively slowly. For years, this was fine, since it was designed for high end graphics workstations using professional level graphics cards. But as consumer level cards have caught up with and surpassed that which was cutting edge 10 years ago, developers have been required to rely on the extension mechanism more and more to keep up with new hardware features. Extensions are powerful, but they can make for very messy code. The ARB seems to recognize this problem, and is now updating the core specification more frequently.

Structure

OpenGL is, at its core, a state machine which controls how primitives are processed and rendered. It uses a procedural model (i.e. functions) to modify the state machine and pass data to it. Due to the high efficiency of this mechanism, code is usually short and (usually) easy to understand. This makes it easier to debug. Although OpenGL itself is procedural, it can be used in linear, modular, or OOP fashion equally easily; the choice is the programmer's. OpenGL also conceals a lot of detail about specific hardware devices and such from you. Most basic operations are very easy to do. Rendering is done between the calls glBegin and glEnd.

All OpenGL functions use a naming convention that looks like glFunctionName[argtypes]. An example of an OpenGL style function would be glVertex3f. "gl" tells you that it is an OpenGL call. "Vertex" tells you that you are specifying a vertex. "3" tells you that the vertex is being specified in 3 dimensions, and "f" means that the arguments are floats. (Note that the number isn't present in most functions, but when it is present it refers to how many components or dimensions the function specifies). To help portability, OpenGL defines custom types, all prefixed by "GL", which may be followed by a "u" for unsigned values, and then a name indicating the type, such as short, long, float, etc. An example type is GLuint.

One final thing to note is that OpenGL syntax for C is much more intuitive and simple than the C syntax for D3D.

Looking at the Header

The header is relatively simple. If you know what a function looks like and what __declspec(dllimport) does, you will understand what the header does. It has defines, then lots of __declspec(dllimport) ReturnType glFuncName(Arguments); lines.

Strengths

OpenGL has many strengths. OpenGL is very portable. It will run for nearly every platform in existence, and it will run well. It even runs on Windows NT 4.0, which Direct3D doesn't even do (well, it does, but you have to use DirectX 3; do you really want to use a version of Direct3D that is 5 versions old? Also, note that Windows 2000 and XP, which are both based on the NT kernel, can use the latest versions of Direct3D). The reason OpenGL runs for so many platforms is because of its Open Standard. What this means is that any company with a platform who wishes to support OpenGL may buy a license from SGI and then implement the entire OpenGL feature set for that platform.

OpenGL has a wide range of features, both in its core and through extensions. Its extension feature allows it to stay immediately current with new hardware features, despite the mess it can cause. Because the ARB is made up of a diverse group of companies, the features available in OpenGL represent a wide range of interests, and thus make it useful in many different applications.

OpenGL is used for many different things. Its stability and support on a wide range of platforms has a lot of appeal to people in tech sectors outside of the game industry. OpenGL is used in such places as the military, professional 3D modeling and rendering, CAD, and so on. It's recognized as the industry standard for graphics everywhere except the game industry, where Direct3D provides viable competition.

Weaknesses

I mention extensions here too. Though they are powerful, they do make code messy, very much so at times. They also make it confusing with any compiler that doesn't offer reference tracking (browse file). The worst part is, many newer extensions are completely card or vendor specific. Although useful for testing a graphics card's abilities, vendor-specific extensions are not frequently used by commercial applications.

The function naming conventions can seem like overkill at times, since many IDEs have context sensitive help which can show you the parameters that are required, and in any case, if you know how to use a function, you should already know what parameters it takes. In addition, having 12 different names for a function may seem strange to a C++

programmer accustomed to function overloading (of course, since C doesn't support function overloading, there really isn't any way to get around it).

Language Support

Under Win32 platforms, at least, OpenGL uses a standard DLL with export functions. Using it is just like using the regular Windows API. That means that you can use it anywhere in any language at any time. (Unfortunately, I have not had experience with Java. People are rather vague about whether or not you actually need Java3D. It is probably best to check Sun's website for the Java specification on this.)

The Future

There has been a great deal of activity lately regarding the future of OpenGL. 3D Labs has put together a proposal for OpenGL 2.0, which is currently under discussion by the ARB. The reasoning behind this proposal is that the original OpenGL specification looked into the future and set the standard for many years to come. The graphics world has now passed the original specification, and the ARB is just trying to keep up with it. 3D Labs wants OpenGL to raise the bar and once again set the standard for the future. OpenGL 2.0 as described by the proposal offers a lot of truly powerful things for the future. I suggest you read it yourself at http://www.3dlabs.com/support/developer/ogl2/index.htm. Amongst other things, it heals the sad state of the OpenGL extension system by including support for you to rewrite the OpenGL rendering pipeline in a C-derived language, which is far better than D3D's cryptic ASM-like shader language. (You can see I'm very hyped up about it, but don't let that make OpenGL 1.3 seem any less powerful.)

Comments/My Advice

If you plan to go specifically into the games industry, you would be wise to learn both D3D and OGL. That's just how it is now. For any other 3D industry (and I do mean any other industry, OpenGL will take you far. Ultimately, it's a matter of personal preference. And whether you decide to learn OpenGL or Direct3D first, you can always learn the other later, and it's actually very easy to learn one if you know the other.

A Note on Glide

You may have heard of another graphics API, called Glide. Glide was developed by 3dfx. It can work on other graphics cards, but was made specifically for 3dfx chipsets. As a result, it typically runs rather badly on non-3dfx cards. On every game you play that uses Glide, you will read, "Do not use Glide unless you have a Voodoo graphics card, or another graphics card with a 3dfx chipset." I have not used Glide myself, but I am told that programming in Glide is a great pleasure. Unfortunately, 3dfx is dead, and although there are many 3dfx cards still out there, they are quickly becoming outdated and being replaced. Using an API that only targets an audience that is rapidly shrinking really is not a good idea. That's how I feel on the issue. To put this very bluntly, in my opinion, learning Glide is a waste of time because its target audience is far too small! As a final note, the fact that most 3dfx cards support OpenGL to some extent, there is absolutely no point in learning Glide.

Performance and Quality Factors

I've carefully avoided mentioning performance for most of this article. Why? There is a huge debate on the problem. There are many people who claim that OGL is faster than D3D, and of course, the reverse as well. I've been studying performance results: I've considered my own, my friends', and the results of major companies. I have come to the following conclusion: Performance is no longer an issue! The speed for both APIs has come out exactly even for well written programs. The performance can only be gauged per machine, and that by testing. There is no way to predict which will run faster. Like I mentioned earlier though, game programmers are going to have to write for both. It sucks, but that's the way it has to be, for now at least. And if you aren't going into games, well, there was really no debate to what API to learn anyway (OpenGL!). So as far as performance goes, I cannot really advise you. It would be best if you evaluate the two on your own.

I feel that I should still address what exactly it is that affects the performance. If a game is well written, by people who know how to properly use the API, then performance is in the hands of the guys who make graphics cards and drivers. No one else has anything to do with it anymore. What determines performance is how well the API, driver, operating system, and hardware interact. On one of my computers, for example, OpenGL runs somewhat faster. I do know, however, that this is due to an incompatibility between DirectX and WindowsXP and my GeForce MX100/200 card/driver.

The above applies a great deal to quality as well. The APIs don't really have an edge over each other at all. The most important factor in quality, however, is your artwork. A good artist will make even a bad game look fantastic, but the converse applies as well. If you want high-quality, get a high-quality artist.
Software Renderer? That sounds AWESOME!

It's not. What a software renderer does is instead of using your video card's built-in functionality, you have to write all the functionality yourself. As you can imagine, this adds a LOT of extra overhead to your program. Quality is typically worse, and features are usually constricted, unless 1) you are one hell of an awesome programmer, or 2) the renderer is under development by your company. Since the renderer is running alongside your program, usually in the same process, it is much, much slower than hardware rendering. In addition to that, writing a decent software renderer is insanely hard, especially if you are doing it alone. I usually reason it out like this: If my end-users need to use a software renderer, then they use have a low-end system which doesn't have a video card with 3D acceleration (increasingly rare nowadays). If they don't have a 3D accelerated card, then it follows that the system is likely to be very slow as well. If their system is that slow, they won't be able to run my software renderer at any usable speed to begin with. If they want to play my game, they should get a decent computer. And the converse: If they have a high-end system which is powerful enough to smoothly run my software renderer, then logically they must have a very powerful system, which would certainly have a video card with 3D acceleration, so they won't need my software renderer anyway.

Ok, there are some advantages to using a software render, though they are few. While playing Half-Life and CounterStrike online, I noticed that the D3D and OGL renderers sometimes failed to 'turn on' after I paused the game and went to the menu. Also, the software renderer could turn itself 'off' very quickly. (The system I play Half-Life on is extremely powerful; your system might not actually run the software renderer at a decent speed.)

The Future in General

The graphics programming field has sort of flattened out in terms of development of new things. Everything that is out there is well understood, well documented, and well accepted. Finding out about a feature only requires going on the internet and using your favorite search engine, or downloading/ordering an SDK. But anyway, back to the future. (Back to the future? Get it? Ya know...the movie...ha ha). I think all eyes should be on the graphics hardware companies. They are doing all the new pathfinding. There is a lot for them to be done, and we have to wait for them, since it's our job to make what they design work. People at companies such as nVidia, SGI, 3D Labs, ATI, etc. are doing a lot of work, and it ought to be pretty cool. After the eternal Processor Wars, the graphics industry is probably the most interesting thing to watch.

Putting It All Together

There is a major battle going on in the Windows sector between Direct3D and OpenGL. If you are not writing for Windows platforms, then don't use D3D, because it won't run on those platforms. Instead, stick with OpenGL. If you ARE writing for Windows, well you have one tough decision to make. If you learn one API, it is quite easy to learn the other, since the essential concepts are the same. I personally learned both at the same time (please do not question my judgement on this). If you truly cannot decide, I have found a foolproof method of deciding. Find a coin. It doesn't matter what kind, as long as the top and bottom have different pictures on them. Now choose one side to be "Heads" and the other side to be "Tails". Toss the coin up into the air. When it lands, check which side is face up. If "Heads", learn OpenGL. If "Tails", learn Direct3D. Now what ever API you chose, using whatever method, stick with it! Graphics programming is not easy to learn! You will sometimes find your self in difficult spots and may feel discouraged. Under no circumstance must you think, "What if I had just learned the other API?" It truly does not matter in the end.

But we can't use Direct3D! John Carmack denounced it!

Many OpenGL advocates point to something Carmack wrote several years ago, trashing Direct3D. Only, that was during the days Direct3D 5. In a much more recent post on www.slashdot.org, Carmack said that with the new advances in D3D, it could no longer be ignored. Direct3D has to be used by programmers along with OpenGL.

I can't take it! Direct3D with all its COM crap and such huge code!

If you can't deal with long code like D3D you will do very badly in the game world. The same goes for COM and classes in general. I do however, concede the point that OpenGL will be a lot simpler for most people, at least in the beginning. If you do learn Direct3D though, remember that for a lot of the learning curve, it is 90% initialization code. The trick is to get the other 10% to work right. (Or use the Common Files; they are very nice too.)

Great Games: What do They Use?

I feel that it is only fair to tell you which APIs some games have used. All the Quake games used OpenGL. Carmack wrote those engines; it is called the Quake engine (amazing, isn't it?). Half-Life wrote support for everything. Earth 2150 (not really a great game, but graphically very advanced) wrote support for OpenGL and D3D. MechWarrior 4: Vengeance used only DirectX, but it was just Microsoft's way of showing off Direct3D8.0 anyway. All XBox games use D3D. (Other consoles are not really worth mentioning since they usually use their own custom APIs, though some - like the Playstation 2 - do support OpenGL in various formats.)

I hinted before that most modeling and animation programs use/prefer OpenGL. While this is true, many also provide Direct3D support, such as Discreet's 3D Studio Max and GMax. Chumbalum Soft's Milkshape however, does not. (This is a very advanced and very cheap editor; for those on a low budget it is a very good deal at 20 USD.)

I admit that the above quotes prove little. Even so, it is helpful to have some reference.

Miscellaneous Addenda

In this article, I considered only the DirectX Graphics part of DirectX. It is worth mentioning that DirectX also provides many other interfaces for working with sound, input, etc. OpenGL doesn't include any of this functionality, because it is a pure graphics library. When writing a game, however, you will need these tools. Fortunately, there are many technologies that can be used with OpenGL that provide this functionality. OpenAL provides a cross platform audio library. There's also OpenML, which has not yet been implemented, but when it's done, it'll be to OpenGL what DirectX is to Direct3D. Another increasingly popular library is the Simple DirectMedia Layer (SDL). Don't confuse this with any of the DirectX components. I like using SDL much more than the other common approach: using DirectInput/DirectMusic/DirectSound to do the extras. That to me seems to defeat much of the purpose of using OpenGL.

Wrapping up

That's about it, really. There's not really too much else to say on the subject. Hopefully this guide has given you a better idea of which API to use, or at least what to consider when choosing. Oh, yes, contacts. For comments, suggestions, and questions, email me at comments@zgx.cjb.net. For mistakes in this article, email me at woops@zgx.cjb.net. If you still feel the need to flame me, email me at flames@zgx.cjb.net. NOTE: I cannot receive attachments, so please do not send them!

Editor's Note

The intent of this article is to provide beginners with an unbiased overview of the OpenGL and Direct3D APIs, presenting the facts so that they can make their own decision about which to use. The author has rewritten the article several times in order to remove inaccuracies and statements which could lead to the types of flame wars we're trying to end with this. I myself have made numerous changes to it for the same reasons. Still, if you feel that it still contains statements that are inaccurate or that suggest a bias for one API or the other, please let me know so that we can make the necessary corrections.

Dave Astle
Executive Producer
Contributors

Many people have submitted suggestions to make this article more accurate and useful. Their names are listed below.

Colin Branch
Mike Shaver
CGameProgrammer
Rikard Björklind
David Notario
Sphartex Kelly "CrazedGenius" Dempski
Sami "MENTAL" Hamlaoui
Null and Void
Sean Howe

All the members of the GameDev.net community, the Slashdot community, and everybody who emailed me!

Discuss this article in the forums

Date this article was posted to GameDev.net: 2/24/2002
(Note that this date does not necessarily correspond to the date the article was written)

See Also:
Advice
Featured Articles