Many of the answers here are really, really good. But the OpenGL and Direct3D (D3D) issue should probably be addressed. And that requires... a history lesson.  
  
And before we begin, I know far more about OpenGL than I do about Direct3D. I've never written a line of D3D code in my life, and I've written tutorials on OpenGL. So what I'm about to say isn't a question of bias. It is simply a matter of history.  
  
Birth of Conflict  
  
One day, sometime in the early 90's, Microsoft looked around. They saw the SNES and Sega Genesis being awesome, running lots of action games and such. And they saw DOS. Developers coded DOS games like console games: direct to the metal. Unlike consoles however, where a developer who made an SNES game knew what hardware the user would have, DOS developers had to write for multiple possible configurations. And this is rather harder than it sounds.  
  
And Microsoft had a bigger problem: Windows. See, Windows wanted to own the hardware, unlike DOS which pretty much let developers do whatever. Owning the hardware is necessary in order to have cooperation between applications. Cooperation is exactly what game developers hate because it takes up precious hardware resources they could be using to be awesome.  
  
In order to promote game development on Windows, Microsoft needed a uniform API that was low-level, ran on Windows without being slowed down by it, and most of all cross-hardware. A single API for all graphics, sound, and input hardware.  
  
Thus, DirectX was born.  
  
3D accelerators were born a few months later. And Microsoft ran into a spot of trouble. See, DirectDraw, the graphics component of DirectX, only dealt with 2D graphics: allocating graphics memory and doing bit-blits between different allocated sections of memory.  
  
So Microsoft purchased a bit of middleware and fashioned it into Direct3D Version 3. It was universally reviled. And with good reason; looking at D3D v3 code is like staring into the Ark of the Covenant.  
  
Old John Carmack at Id Software took one look at that trash and said, "Screw that!" and decided to write towards another API: OpenGL.  
  
See, another part of the many-headed-beast that is Microsoft had been busy working with SGI on an OpenGL implementation for Windows. The idea here was to court developers of typical GL applications: workstation apps. CAD tools, modelling, that sort of thing. Games were the farthest thing on their mind. This was primarily a Windows NT thing, but Microsoft decided to add it to Win95 too.  
  
As a way to entice workstation developers to Windows, Microsoft decided to try to bribe them with access to these newfangled 3D graphics cards. Microsoft implemented the Installable Client Driver protocol: a graphics card maker could override Microsoft's software OpenGL implementation with a hardware-based one. Code could automatically just use a hardware OpenGL implementation if one was available.  
  
In the early days, consumer-level videocards did not have support for OpenGL though. That didn't stop Carmack from just porting Quake to OpenGL (GLQuake) on his SGI workstation. As we can read from the GLQuake readme:  
  
Theoretically, glquake will run on any compliant OpenGL that supports the texture objects extensions, but unless it is very powerfull hardware that accelerates everything needed, the game play will not be acceptable. If it has to go through any software emulation paths, the performance will likely by well under one frame per second.  
  
At this time (march ’97), the only standard opengl hardware that can play glquake reasonably is an intergraph realizm, which is a VERY expensive card. 3dlabs has been improving their performance significantly, but with the available drivers it still isn’t good enough to play. Some of the current 3dlabs drivers for glint and permedia boards can also crash NT when exiting from a full screen run, so I don’t recommend running glquake on 3dlabs hardware.  
  
3dfx has provided an opengl32.dll that implements everything glquake needs, but it is not a full opengl implementation. Other opengl applications are very unlikely to work with it, so consider it basically a “glquake driver”.  
This was the birth of the miniGL drivers. These evolved into full OpenGL implementations eventually, as hardware became powerful enough to implement most OpenGL functionality in hardware. nVidia was the first to offer a full OpenGL implementation. Many other vendors struggled, which is one reason why developers preferred Direct3D: they were compatible on a wider range of hardware. Eventually only nVidia and ATI (now AMD) remained, and both had a good OpenGL implementation.  
  
OpenGL Ascendant  
  
Thus the stage is set: Direct3D vs. OpenGL. It's really an amazing story, considering how bad D3D v3 was.  
  
The OpenGL Architectural Review Board (ARB) is the organization responsible for maintaining OpenGL. They issue a number of extensions, maintain the extension repository, and create new versions of the API. The ARB is a committee made of many of the graphics industry players, as well as some OS makers. Apple and Microsoft have at various times been a member of the ARB.  
  
3Dfx comes out with the Voodoo2. This is the first hardware that can do multitexturing, which is something that OpenGL couldn't do before. While 3Dfx was strongly against OpenGL, NVIDIA, makers of the next multitexturing graphics chip (the TNT1), loved it. So the ARB issued an extension: GL\_ARB\_multitexture, which would allow access to multitexturing.  
  
Meanwhile, Direct3D v5 comes out. Now, D3D has become an actual API, rather than something a cat might vomit up. The problem? No multitexturing.  
  
Oops.  
  
Now, that one wouldn't hurt nearly as much as it should have, because people didn't use multitexturing much. Not directly. Multitexturing hurt performance quite a bit, and in many cases it wasn't worth it compared to multi-passing. And of course, game developers love to ensure that their games works on older hardware, which didn't have multitexturing, so many games shipped without it.  
  
D3D was thus given a reprieve.  
  
Time passes and NVIDIA deploys the GeForce 256 (not GeForce GT-250; the very first GeForce), pretty much ending competition in graphics cards for the next two years. The main selling point is the ability to do vertex transform and lighting (T&L) in hardware. Not only that, NVIDIA loved OpenGL so much that their T&L engine effectively was OpenGL. Almost literally; as I understand, some of their registers actually took OpenGL enumerators directly as values.  
  
Direct3D v6 comes out. Multitexture at last but... no hardware T&L. OpenGL had always had a T&L pipeline, even though before the 256 it was implemented in software. So it was very easy for NVIDIA to just convert their software implementation to a hardware solution. It wouldn't be until D3D v7 until D3D finally had hardware T&L support.  
  
Dawn of Shaders, Twilight of OpenGL  
  
Then, GeForce 3 came out. And a lot of things happened at the same time.  
  
Microsoft had decided that they weren't going to be late again. So instead of looking at what NVIDIA was doing and then copying it after the fact, they took the astonishing position of going to them and talking to them. And then they fell in love and had a little console together.  
  
A messy divorce ensued later. But that's for another time.  
  
What this meant for the PC was that GeForce 3 came out simultaneously with D3D v8. And it's not hard to see how GeForce 3 influenced D3D 8's shaders. The pixel shaders of Shader Model 1.0 were extremely specific to NVIDIA's hardware. There was no attempt made whatsoever at abstracting NVIDIA's hardware; SM 1.0 was just whatever the GeForce 3 did.  
  
When ATI started to jump into the performance graphics card race with the Radeon 8500, there was a problem. The 8500's pixel processing pipeline was more powerful than NVIDIA's stuff. So Microsoft issued Shader Model 1.1, which basically was "Whatever the 8500 does."  
  
That may sound like a failure on D3D's part. But failure and success are matters of degrees. And epic failure was happening in OpenGL-land.  
  
NVIDIA loved OpenGL, so when GeForce 3 hit, they released a slew of OpenGL extensions. Proprietary OpenGL extensions: NVIDIA-only. Naturally, when the 8500 showed up, it couldn't use any of them.  
  
See, at least in D3D 8 land, you could run your SM 1.0 shaders on ATI hardware. Sure, you had to write new shaders to take advantage of the 8500's coolness, but at least your code worked.  
  
In order to have shaders of any kind on Radeon 8500 in OpenGL, ATI had to write a number of OpenGL extensions. Proprietary OpenGL extensions: ATI-only. So you needed an NVIDIA codepath and an ATI codepath, just to have shaders at all.  
  
Now, you might ask, "Where was the OpenGL ARB, whose job it was to keep OpenGL current?" Where many committees often end up: off being stupid.  
  
See, I mentioned ARB\_multitexture above because it factors deeply into all of this. The ARB seemed (from an outsider's perspective) to want to avoid the idea of shaders altogether. They figured that if they slapped enough configurability onto the fixed-function pipeline, they could equal the ability of a shader pipeline.  
  
So the ARB released extension after extension. Every extension with the words "texture\_env" in it was yet another attempt to patch this aging design. Check the registry: between ARB and EXT extensions, there were eight of these extensions made. Many were promoted to OpenGL core versions.  
  
Microsoft was a part of the ARB at this time; they left around the time D3D 9 hit. So it is entirely possible that they were working to sabotage OpenGL in some way. I personally doubt this theory for two reasons. One, they would have had to get help from other ARB members to do that, since each member only gets one vote. And most importantly two, the ARB didn't need Microsoft's help to screw things up. We'll see further evidence of that.  
  
Eventually the ARB, likely under threat from both ATI and NVIDIA (both active members) eventually pulled their head out long enough to provide actual assembly-style shaders.  
  
Want something even stupider?  
  
Hardware T&L. Something OpenGL had first. Well, it's interesting. To get the maximum possible performance from hardware T&L, you need to store your vertex data on the GPU. After all, it's the GPU that actually wants to use your vertex data.  
  
In D3D v7, Microsoft introduced the concept of Vertex Buffers. These are allocated swaths of GPU memory for storing vertex data.  
  
Want to know when OpenGL got their equivalent of this? Oh, NVIDIA, being a lover of all things OpenGL (so long as they are proprietary NVIDIA extensions), released the vertex array range extension when the GeForce 256 first hit. But when did the ARB decide to provide similar functionality?  
  
Two years later. This was after they approved vertex and fragment shaders (pixel in D3D language). That's how long it took the ARB to develop a cross-platform solution for storing vertex data in GPU memory. Again, something that hardware T&L needs to achieve maximum performance.  
  
One Language to Ruin Them All  
  
So, the OpenGL development environment was fractured for a time. No cross-hardware shaders, no cross-hardware GPU vertex storage, while D3D users enjoyed both. Could it get worse?  
  
You... you could say that. Enter 3D Labs.  
  
Who are they, you might ask? They are a defunct company whom I consider to be the true killers of OpenGL. Sure, the ARB's general ineptness made OpenGL vulnerable when it should have been owning D3D. But 3D Labs is perhaps the single biggest reason to my mind for OpenGL's current market state. What could they have possibly done to cause that?  
  
They designed the OpenGL Shading Language.  
  
See, 3D Labs was a dying company. Their expensive GPUs were being marginalized by NVIDIA's increasing pressure on the workstation market. And unlike NVIDIA, 3D Labs did not have any presence in the mainstream market; if NVIDIA won, they died.  
  
Which they did.  
  
So, in a bid to remain relevant in a world that didn't want their products, 3D Labs showed up to a Game Developer Conference wielding presentations for something they called "OpenGL 2.0". This would be a complete, from-scratch rewrite of the OpenGL API. And that makes sense; there was a lot of cruft in OpenGL's API at the time (note: that cruft still exists). Just look at how texture loading and binding work; it's semi-arcane.  
  
Part of their proposal was a shading language. Naturally. However, unlike the current cross-platform ARB extensions, their shading language was "high-level" (C is high-level for a shading language. Yes, really).  
  
Now, Microsoft was working on their own high-level shading language. Which they, in all of Microsoft's collective imagination, called... the High Level Shading Language (HLSL). But their was a fundamentally different approach to the languages.  
  
The biggest issue with 3D Labs's shader language was that it was built-in. See, HLSL was a language Microsoft defined. They released a compiler for it, and it generated Shader Model 2.0 (or later shader models) assembly code, which you would feed into D3D. In the D3D v9 days, HLSL was never touched by D3D directly. It was a nice abstraction, but it was purely optional. And a developer always had the opportunity to go behind the compiler and tweak the output for maximum performance.  
  
The 3D Labs language had none of that. You gave the driver the C-like language, and it produced a shader. End of story. Not an assembly shader, not something you feed something else. The actual OpenGL object representing a shader.  
  
What this meant is that OpenGL users were open to the vagaries of developers who were just getting the hang of compiling assembly-like languages. Compiler bugs ran rampant in the newly christened OpenGL Shading Language (GLSL). What's worse, if you managed to get a shader to compile on multiple platforms correctly (no mean feat), you were still subjected to the optimizers of the day. Which were not as optimal as they could be.  
  
While that was the biggest flaw in GLSL, it wasn't the only flaw. By far.  
  
In D3D, and in the older assembly languages in OpenGL, you could mix and match vertex and fragment (pixel) shaders. So long as they communicated with the same interface, you could use any vertex shader with any compatible fragment shader. And there were even levels of incompatibility they could accept; a vertex shader could write an output that the fragment shader didn't read. And so forth.  
  
GLSL didn't have any of that. Vertex and fragment shaders were fused together into what 3D Labs called a "program object". So if you wanted to share vertex and fragment programs, you had to build multiple program objects. And this caused the second biggest problem.  
  
See, 3D Labs thought they were being clever. They based GLSL's compilation model on C/C++. You take a .c or .cpp and compile it into an object file. Then you take one or more object files and link them into a program. So that's how GLSL compiles: you compile your shader (vertex or fragment) into a shader object. Then you put those shader objects in a program object, and link them together to form your actual program.  
  
While this did allow potential cool ideas like having "library" shaders that contained extra code that the main shaders could call, what it meant in practice was that shaders were compiled twice. Once in the compilation stage and once in the linking stage. NVIDIA's compiler in particular was known for basically running the compile twice. It didn't generate some kind of object code intermediary; it just compiled it once and threw away the answer, then compiled it again at link time.  
  
So even if you want to link your vertex shader to two different fragment shaders, you have to do a lot more compiling than in D3D. Especially since the compiling of a C-like language was all done offline, not at the beginning of the program's execution.  
  
There were other issues with GLSL. Perhaps it seems wrong to lay the blame on 3D Labs, since the ARB did eventually approve and incorporate the language (but nothing else of their "OpenGL 2.0" initiative). But it was their idea.  
  
And here's the really sad part: 3D Labs was right (mostly). GLSL is not a vector-based shading language the way HLSL was at the time. This was because 3D Labs's hardware was scalar hardware (similar to modern NVIDIA hardware), but they were ultimately right in the direction many hardware makers went with their hardware.  
  
They were right to go with a compile-online model for a "high-level" language. D3D even switched to that eventually.  
  
The problem was that 3D Labs were right at the wrong time. And in trying to summon the future too early, in trying to be future-proof, they cast aside the present. It sounds similar to how OpenGL always had the possibility for T&L functionality. Except that OpenGL's T&L pipeline was still useful before hardware T&L, while GLSL was a liability before the world caught up to it.  
  
GLSL is a good language now. But for the time? It was horrible. And OpenGL suffered for it.  
  
Falling Towards Apotheosis  
  
While I maintain that 3D Labs struck the fatal blow, it was the ARB itself who would drive the last nail in the coffin.  
  
This is a story you may have heard of. By the time of OpenGL 2.1, OpenGL was running into a problem. It had a lot of legacy cruft. The API wasn't easy to use anymore. There were 5 ways to do things, and no idea which was the fastest. You could "learn" OpenGL with simple tutorials, but you didn't really learn the OpenGL API that gave you real performance and graphical power.  
  
So the ARB decided to attempt another re-invention of OpenGL. This was similar to 3D Labs's "OpenGL 2.0", but better because the ARB was behind it. They called it "Longs Peak."  
  
What is so bad about taking some time to improve the API? This was bad because Microsoft had left themselves vulnerable. See, this was at the time of the Vista switchover.  
  
With Vista, Microsoft decided to institute some much-needed changes in display drivers. They forced drivers to submit to the OS for graphics memory virtualization and various other things.  
  
While one can debate the merits of this or whether it was actually possible, the fact remains this: Microsoft deemed D3D 10 to be Vista (and above) only. Even if you had hardware that was capable of D3D 10, you couldn't run D3D 10 applications without also running Vista.  
  
You might also remember that Vista... um, let's just say that it didn't work out well. So you had an underperforming OS, a new API that only ran on that OS, and a fresh generation of hardware that needed that API and OS to do anything more than be faster than the previous generation.  
  
However, developers could access D3D 10-class features via OpenGL. Well, they could if the ARB hadn't been busy working on Longs Peak.  
  
Basically, the ARB spent a good year and a half to two years worth of work to make the API better. By the time OpenGL 3.0 actually came out, Vista adoption was up, Win7 was around the corner to put Vista behind them, and most game developers didn't care about D3D-10 class features anyway. After all, D3D 10 hardware ran D3D 9 applications just fine. And with the rise of PC-to-console ports (or PC developers jumping ship to console development. Take your pick), developers didn't need D3D 10 class features.  
  
Now, if developers had access to those features earlier via OpenGL on WinXP machines, then OpenGL development might have received a much-needed shot in the arm. But the ARB missed their opportunity. And do you want to know the worst part?  
  
Despite spending two precious years attempting to rebuild the API from scratch... they still failed and just reverted back to the status quo (except for a deprecation mechanism).  
  
So not only did the ARB miss a crucial window of opportunity, they didn't even get done the task that made them miss that chance. Pretty much epic fail all around.  
  
And that's the tale of OpenGL vs. Direct3D. A tale of missed opportunities, gross stupidity, willful blindness, and simple foolishness.