```
[idsoftware.com]
Login name: johnc
In real life: John Carmack
Directory: /raid/nardo/johnc
Shell: /bin/csh
On since Jan 6 14:59:09
4 days 14 hours Idle Time
on ttyp2 from idcarmack
Plan:
```

I am going to use this installment of my .plan file to get up on a soapbox about an important issue to me: 3D API. I get asked for my opinions about this often enough that it is time I just made a public statement. So here it is, my current position as of december '96...

While the rest of Id works on Quake 2, most of my effort is now focused on developing the next generation of game technology. This new generation of technology will be used by Id and other companies all the way through the year 2000, so there are some very important long term decisions to be made.

There are two viable contenders for low level 3D programming on win32: Direct-3D Immediate Mode, the new, designed for games API, and OpenGL, the workstation graphics API originally developed by SGI. They are both supported by microsoft, but D3D has been evangelized as the one true solution for games.

I have been using OpenGL for about six months now, and I have been very impressed by the design of the API, and especially it's ease of use. A month ago, I ported quake to OpenGL. It was an extremely pleasant experience. It didn't take long, the code was clean and simple, and it gave me a great testbed to rapidly try out new research ideas.

I started porting glquake to Direct-3D IM with the intent of learning the api and doing a fair comparison.

Well, I have learned enough about it. I'm not going to finish the port. I have better things to do with my time.

I am hoping that the vendors shipping second generation cards in the coming year can be convinced to support OpenGL. If this doesn't happen early on and there are capable cards that glquake does not run on, then I apologize, but I am taking a little stand in my little corner of the world with the hope of having some small influence on things that are going to effect us for many years to come.

Direct-3D IM is a horribly broken API. It inflicts great pain and suffering on the programmers using it, without returning any significant advantages. I don't think there is ANY market segment that D3D is apropriate for, OpenGL seems to work just fine for everything from quake to softimage. There is no good technical reason for the existance of D3D.

I'm sure D3D will suck less with each forthcoming version, but this is an oportunity to just bypass dragging the entire development community through the messy evolution of an ill-birthed API.

Best case: Microsoft integrates OpenGL with direct-x (probably calling it Direct-GL or something), ports D3D retained mode on top of GL, and tells everyone to forget they every heard of D3D immediate mode. Programmers have one good api, vendors have one driver to write, and the world is a better place.

To elaborate a bit:

"OpenGL" is either OpenGL 1.1 or OpenGL 1.0 with the common extensions. Raw OpenGL 1.0 has several holes in functionality.

"D3D" is Direct-3D Immediate Mode. D3D retained mode is a seperate issue. Retained mode has very valid reasons for existance. It is a good thing to have an api that lets you just load in model files and fly around without sweating the polygon details. Retained mode is going to be used by at least ten times as many programmers as immediate mode. On the other hand, the world class applications that really step to new levels are going to be done in an immediate mode graphics API. D3D-RM doesn't even really have to be tied to D3D-IM. It could be implemented to emit OpenGL code instead.

I don't particularly care about the software only implementations of either D3D or OpenGL. I haven't done serious research here, but I think D3D has a real edge, because it was originally designed for software rendering and much optimization effort has been focused there. COSMO GL is attempting to compete there, but I feel the effort is misguided. Software rasterizers will still exist to support the lowest common denominator, but soon all game development will be targeted at hardware rasterization, so that's where effort should be focused.

The primary importance of a 3D API to game developers is as an interface to the wide variety of 3D hardware that is emerging. If there was one compatable line of hardware that did what we wanted and covered 90+ percent of the target market, I wouldn't even want a 3D API for production use, I would be writing straight to the metal, just like I allways have with pure software schemes. I would still want a 3D API for research and tool development, but it wouldn't matter if it wasn't a mainstream solution.

Because I am expecting the 3D accelerator market to be fairly fragmented for the forseeable future, I need an API to write to, with individual drivers for each brand of hardware. OpenGL has been maturing in the workstation market for many years now, allways with a hardware focus. We have exisiting proof that it scales just great from a $300 permedia card all the way to a $250,000 loaded infinite reality system.

All of the game oriented PC 3D hardware basically came into existance in the last year. Because of the frantic nature of the PC world, we may be getting stuck with a first guess API and driver model which isn't all that good.

The things that matter with an API are: functionality, performance, driver coverage, and ease of use.

Both APIs cover the important functionality. There shouldn't be any real argument about that. GL supports some additional esoteric features that I am unlikely to use (or are unlikely to be supported by hardware -- same effect). D3D actually has a couple nice features that I would like to see moved to GL (specular blend at each vertex, color key transparancy, and no clipping hints), which brings up the extensions issue. GL can be extended by the driver, but because D3D imposes a layer between the driver and the API, microsoft is the only one that can extend D3D.

My conclusion about performance is that there is not going to be any significant performance difference (< 10%) between properly written OpenGL and D3D drivers for several years at least. There are some arguments that gl will scale better to very high end hardware because it doesn't need to build any intermediate structures, but you could use tiny sub cache sized execute buffers in d3d and acheive reasonably similar results (or build complex hardware just to suit D3D -- ack!). There are also arguments from the other side that the vertex pools in d3d will save work on geometry bound applications, but you can do the same thing with vertex arrays in GL.

Currently, there are more drivers avaialble for D3D than OpenGL on the consumer level boards. I hope we can change this. A serious problem is that there are no D3D conformance tests, and the documentation is very poor, so the existing drivers aren't exactly uniform in their functionality. OpenGL has an established set of conformance tests, so there is no argument about exactly how things are supposed to work. OpenGL offers two levels of drivers that can be written: mini client drivers and installable client drivers. A MCD is a simple, robust exporting of hardware rasterization capabilities. An ICD is basically a full replacement for the API that lets hardware accelerate or extend any piece of GL without any overhead.

The overriding reason why GL is so much better than D3D has to do with ease of use. GL is easy to use and fun to experiment with. D3D is not (ahem). You can make sample GL programs with a single page of code. I think D3D has managed to make the worst possible interface choice at every oportunity. COM. Expandable structs passed to functions. Execute buffers. Some of these choices were made so that the API would be able to gracefully expand in the future, but who cares about having an API that can grow if you have forced it to be painful to use now and forever after? Many things that are a single line of GL code require half a page of D3D code to allocate a structure, set a size, fill something in, call a COM routine, then extract the result.

Ease of use is damn important. If you can program something in half the time, you can ship earlier or explore more aproaches. A clean, readable coding interface also makes it easier to find / prevent bugs.

GL's interface is procedural: You perform operations by calling gl functions to pass vertex data and specify primitives.

```
glBegin (GL_TRIANGLES);
glVertex (0,0,0);
glVertex (1,1,0);
glVertex (2,0,0);
glEnd ();
```

D3D's interface is by execute buffers: You build a structure containing vertex data and commands, and pass the entire thing with a single call. On the surface, this apears to be an efficiency improvement for D3D, because it gets rid of a lot of procedure call overhead. In reality, it is a gigantic pain-in-the-ass.

```
(psuedo code, and incomplete)
v = &buffer.vertexes[0];
v->x = 0; v->y = 0; v->z = 0;
v++;
v->x = 1; v->y = 1; v->z = 0;
v++;
v->x = 2; v->y = 0; v->z = 0;
c = &buffer.commands;
c->operation = DRAW_TRIANGLE;
c->vertexes[0] = 0;
c->vertexes[1] = 1;
c->vertexes[2] = 2;
IssueExecuteBuffer (buffer);
```

If I included the complete code to actually lock, build, and issue an execute buffer here, you would think I was choosing some pathologically slanted case to make D3D look bad.

You wouldn't actually make an execute buffer with a single triangle in it, or your performance would be dreadfull. The idea is to build up a large batch of commands so that you pass lots of work to D3D with a single procedure call.

A problem with that is that the optimal definition of "large" and "lots" varies depending on what hardware you are using, but instead of leaving that up to the driver, the application programmer has to know what is best for every hardware situation.

You can cover some of the messy work with macros, but that brings its own set of problems. The only way I can see to make D3D generally usable is to create your own procedural interface that buffers commands up into one or more execute buffers and flushes when needed. But why bother, when there is this other nifty procedural API allready there...

With OpenGL, you can get something working with simple, straightforward code, then if it is warranted, you can convert to display lists or vertex arrays for max performance (although the difference usually isn't that large). This is the right way of doing things -- like converting your crucial functions to assembly language after doing all your development in C.

With D3D, you have to do everything the painful way from the beginning. Like writing a complete program in assembly language, taking many times longer, missing chances for algorithmic improvements, etc. And then finding out it doesn't even go faster.

I am going to be programming with a 3D API every day for many years to come. I want something that helps me, rather than gets in my way.

John Carmack
Id Software

Back <-