John Carmack Archive - .plan (2003)

http://www.team5150.com/~andrew/carmack

March 18, 2007

Contents

Chapter 1

January

1.1 NV30 vs R300, current developments, etc (Jan 29, 2003)

At the moment, the NV30 is slightly faster on most scenes in Doom than the R300, but I can still find some scenes where the R300 pulls a little bit ahead. The issue is complicated because of the different ways the cards can choose to run the game.

The R300 can run Doom in three different modes: ARB (minimum extensions, no specular highlights, no vertex programs), R200 (full featured, almost always single pass interaction rendering), ARB2 (floating point fragment shaders, minor quality improvements, always single pass).

The NV30 can run DOOM in five different modes: ARB, NV10 (full featured, five rendering passes, no vertex programs), NV20 (full featured, two or three rendering passes), NV30 (full featured, single pass), and ARB2.

The R200 path has a slight speed advantage over the ARB2 path on the R300, but only by a small margin, so it defaults to using the ARB2 path for the quality improvements. The NV30 runs the ARB2 path MUCH slower than the NV30 path. Half the speed at the moment. This is unfortunate, because when you do an exact, apples-to-apples comparison using ex-

actly the same API, the R300 looks twice as fast, but when you use the vendor-specific paths, the NV30 wins.

The reason for this is that ATI does everything at high precision all the time, while Nvidia internally supports three different precisions with different performances. To make it even more complicated, the exact precision that ATI uses is in between the floating point precisions offered by Nvidia, so when Nvidia runs fragment programs, they are at a higher precision than ATI's, which is some justification for the slower speed. Nvidia assures me that there is a lot of room for improving the fragment program performance with improved driver compiler technology.

The current NV30 cards do have some other disadvantages: They take up two slots, and when the cooling fan fires up they are VERY LOUD. I'm not usually one to care about fan noise, but the NV30 does annoy me.

I am using an NV30 in my primary work system now, largely so I can test more of the rendering paths on one system, and because I feel Nvidia still has somewhat better driver quality (ATI continues to improve, though). For a typical consumer, I don't think the decision is at all clear cut at the moment.

For developers doing forward looking work, there is a different tradeoff - the NV30 runs fragment programs much slower, but it has a huge maximum instruction count. I have bumped into program limits on the R300 already.

As always, better cards are coming soon.

—

Doom has dropped support for vendor-specific vertex programs (NV_vertex_program and EXT_vertex_shader), in favor of using ARB_vertex_program for all rendering paths. This has been a pleasant thing to do, and both ATI and Nvidia supported the move. The standardization process for ARB_vertex_program was pretty drawn out and arduous, but in the end, it is a just-plain-better API than either of the vendor specific ones that it replaced. I fretted for a while over whether I should leave in support for the older APIs for broader driver compatibility, but the final decision was that we are going to require a modern driver for the game to run in the advanced modes.

Older drivers can still fall back to either the ARB or NV10 paths.

The newly-ratified ARB_vertex_buffer_object extension will probably let me do the same thing for NV_vertex_array_range and ATI_vertex_array_object.

Reasonable arguments can be made for and against the OpenGL or Direct-X style of API evolution. With vendor extensions, you get immediate access to new functionality, but then there is often a period of squabbling about exact feature support from different vendors before an industry standard settles down. With central planning, you can have "phasing problems" between hardware and software releases, and there is a real danger of bad decisions hampering the entire industry, but enforced commonality does make life easier for developers. Trying to keep boneheaded-ideas-that-will-haunt-us-for-years out of Direct-X is the primary reason I have been attending the Windows Graphics Summit for the past three years, even though I still code for OpenGL.

The most significant functionality in the new crop of cards is the truly flexible fragment programming, as exposed with ARB_fragment_program. Moving from the "switches and dials" style of discrete functional graphics programming to generally flexible programming with indirection and high precision is what is going to enable the next major step in graphics engines.

It is going to require fairly deep, non-backwards-compatible modifications to an engine to take real advantage of the new features, but working with ARB_fragment_program is really a lot of fun, so I have added a few little tweaks to the current codebase on the ARB2 path:

High dynamic color ranges are supported internally, rather than with post-blending. This gives a few more bits of color precision in the final image, but it isn't something that you really notice.

Per-pixel environment mapping, rather than per-vertex. This fixes a pet-peeve of mine, which is large panes of environment mapped glass that aren't tessellated enough, giving that awful warping-around-the-triangulation effect as you move past them.

Light and view vectors normalized with math, rather than a cube map. On future hardware this will likely be a performance improvement due

to the decrease in bandwidth, but current hardware has the computation and bandwidth balanced such that it is pretty much a wash. What it does (in conjunction with floating point math) give you is a perfectly smooth specular highlight, instead of the pixelish blob that we get on older generations of cards.

There are some more things I am playing around with, that will probably remain in the engine as novelties, but not supported features:

Per-pixel reflection vector calculations for specular, instead of an interpolated half-angle. The only remaining effect that has any visual dependency on the underlying geometry is the shape of the specular highlight. Ideally, you want the same final image for a surface regardless of if it is two giant triangles, or a mesh of 1024 triangles. This will not be true if any calculation done at a vertex involves anything other than linear math operations. The specular half-angle calculation involves normalizations, so the interpolation across triangles on a surface will be dependent on exactly where the vertexes are located. The most visible end result of this is that on large, flat, shiny surfaces where you expect a clean highlight circle moving across it, you wind up with a highlight that distorts into an L shape around the triangulation line.

The extra instructions to implement this did have a noticeable performance hit, and I was a little surprised to see that the highlights not only stabilized in shape, but also sharpened up quite a bit, changing the scene more than I expected. This probably isn't a good tradeoff today for a gamer, but it is nice for any kind of high-fidelity rendering.

Renormalization of surface normal map samples makes significant quality improvements in magnified textures, turning tight, blurred corners into shiny, smooth pockets, but it introduces a huge amount of aliasing on minimized textures. Blending between the cases is possible with fragment programs, but the performance overhead does start piling up, and it may require stashing some information in the normal map alpha channel that varies with mip level. Doing good filtering of a specularly lit normal map texture is a fairly interesting problem, with lots of subtle issues.

Bump mapped ambient lighting will give much better looking outdoor and well-lit scenes. This only became possible with dependent texture reads, and it requires new designer and tool-chain support to implement

well, so it isn't easy to test globally with the current Doom datasets, but isolated demos are promising.

The future is in floating point framebuffers. One of the most noticeable thing this will get you without fundamental algorithm changes is the ability to use a correct display gamma ramp without destroying the dark color precision. Unfortunately, using a floating point framebuffer on the current generation of cards is pretty difficult, because no blending operations are supported, and the primary thing we need to do is add light contributions together in the framebuffer. The workaround is to copy the part of the framebuffer you are going to reference to a texture, and have your fragment program explicitly add that texture, instead of having the separate blend unit do it. This is intrusive enough that I probably won't hack up the current codebase, instead playing around on a forked version.

Floating point framebuffers and complex fragment shaders will also allow much better volumetric effects, like volumetric illumination of fogged areas with shadows and additive/subtractive eddy currents.

John Carmack

Chapter 2

February

2.1 Feb 07, 2003

The machinima music video that Fountainhead Entertainment (my wife's company) produced with Quake based tools is available for viewing and voting on at: http://www.mtv.com/music/viewers_pick/ ("In the waiting line")

I thought they did an excellent job of catering to the strengths of the medium, and not attempting to make a game engine compete (poorly) as a general purpose renderer. In watching the video, I did beat myself up a bit over the visible popping artifacts on the environment mapping, which are a direct result of the normal vector quantization in the md3 format. While it isn't the same issue (normals are full floating point already in Doom), it was the final factor that pushed me to do the per-pixel environment mapping for the new cards in the current engine.

The neat thing about the machinima aspect of the video is that they also have a little game you can play with the same media assets used to create the video. Not sure when it will be made available publicly.