Renderstate change costs

TomForsyth, 28 January 2008 (created 27 January 2008)

Renderstate changes cost valuable clock cycles on both the CPU and GPU. So it's a good idea to sort your rendering order by least-cost. But you need to be careful you know what "least-cost" means - it's not just the number of SetRenderState calls you send down!

(note - I'm going to suggest some numbers in this section - all numbers are purely hypothetical - they are realistic, but I have deliberately avoided making them the same as any hardware I know about - they're just for illustration)

Typically, graphics hardware is driven by a bunch of internal bits that control the fixed-function units and the flow of the overall pipeline. The mapping between those bits and the renderstates exposed by the graphics API is far from obvious. For example, the various alpha-blend states look simple in the API, but they have large and wide-ranging implications for the hardware pipeline - depending on what you set, various types of early, hierarchical and late Z will be enabled or disabled. In general, it is difficult to predict how some render states will affect the pipeline - I have written graphics drivers, I have an excellent knowledge of graphics hardware, and I'm responsible for some right now, and I *still* have a hard time predicting what a change in a single renderstate will do to the underlying hardware.

To add to the complexity, a lot of modern hardware can have a certain limited number of rendering pipeline states (sometimes called "contexts", in a slightly confusing manner) in flight. For further complexity, this number changes at different points in the pipeline. For example, some hardware may be able to have 2 pixel shaders, 2 vertex shaders, 4 sets of vertex shader constants in flight at once, and 8 different sets of textures (again, numbers purely for illustration). If the driver submits one more than this number at the same time, the pipeline will stall. Note the differing numbers for each category!

In general, there are some pipeline changes which are cheaper than others. These are "value" changes rather than "functional" changes. For example, changing the alpha-blend state is, as mentioned, a functional change. It enables or disables different things - the ordering of operations in the pipeline changes. This can be expensive. But changing e.g. the fog colour is not a functional change, it's a value

change. Fog is still on (or off), but there's a register that changes its value from red to blue - that's not a functional change, so it is usually fairly cheap. Be aware that there are some that look like "value" changes that can be both. For example, Z-bias looks like just a single value that changes from - should be cheap, right? But if you change that value to 0, then the Z-bias circuitry can be switched off entirely, and maybe the pipeline can be reconfigured to go faster. So although this looks like a value change, it can be a functional change as well.

With that in mind, here's a very general guide to state change costs for the hardware, ordered least to most:

- Changing vertex, index or constant buffers (DX10) for another one **of the same format**. Here, you are simply changing the pointer to where the data starts. Because the description of the contents hasn't changed, it's usually not a large pipeline change.

- Changing a texture for another texture identical in every way (format, size, number of mipmaps, etc) except the data held. Again, you're just changing the pointer to the start of the data, not changing anything in the pipeline. Note that the "same format" requirement is important - in some hardware, the shader hardware has to do float32 filtering, whereas fixed-function (i.e. fast) hardware can do float16 and smaller. Obviously changing from one to the other requires new shaders to be uploaded!

- Changing constants, e.g. shader constants, fog color, transform matrices, etc. Everything that is a *value* rather than an enum or bool that actually changes the pipeline. Note that in DX10 hardware, the shader constants are cheap to change. But in DX9 hardware, they can be quite expensive. A "phase change" happened there, with shader constants being moved out of the core and into general memory.

- Everything else - shaders, Z states, blend states, etc. These tend to cause widescale disruption to the rendering pipeline - units get enabled, disabled, fast paths turned on and off. Big changes. I'm not aware of that much cost difference between all these changes.

(note that changing render target is even more expensive than all of the above - it's not really a "state change" - it's a major disruption to the pipeline - first and foremost,

order by render target)

OK, so that's the **hardware** costs. what about the **software** costs?

Well, there's certainly the cost of actually making the renderstate change call. But that's usually pretty small - it's a function call and storing a DWORD in an array. Do not try to optimise for the least number of SetRenderState calls - you'll spend more CPU power doing the sorting than save by minimising the number of calls. Total false economy.

Also note that the actual work of state changes doesn't happen when you do the SetRenderState. It happens the next time you do a draw call, when the driver looks at the whole state vector and decides how the hardware has to change accordingly - this is caled "lazy state changes", and every driver does it these days. Think of it as a mini-compile stage - the driver looks at the API specification of the pipeline and "compiles" it to the hardware description. In some cases this is literally a compile - the shaders have to be changed in some way to accommodate the state changes. This mini-compile is obviously expensive, but the best drivers cache states from previous frames so that it doesn't do the full thinking every time. Of course, it still has to actually send the new pipeline to the card, and stall when the card has too many contexts in flight (see above). So what tends to happen is that on the CPU side, there's states that cause a new pipeline (tend to be higher on the list above), and states that don't (tend to be lower). But there's a much lower difference in cost between top and bottom.

(note that although the **SuperSecretProject** is a software renderer, you still have a certain cost to state changes, and those costs are reflected in the above data. It's not exactly a coincidence that a software renderer and hardware renderers have similar profiles - they have to do the same sort of things. Our consolation is that we don't have any hard limits, so in general the cost of changing renderstates is much lower than hardware).

Of course it's also a good idea to sort front-to-back (for opaque objects), so that has to be reconciled with sorting by state. The way I do it is to sort renderstates hierarchically - so there's four levels according to the above four items. Generally I assign a byte of hash per list item, and I concatenate the bytes to form a uint32 (it's a lucky accident there's four items - I didn't plan it that way). Also note this hash is computed at mesh creation time - I don't go around making hashes at runtime.

So now I have a bunch of hashed renderstates, what order do I put them in? What I generally do is assume the first two types of state change - changing constant buffers and texture pointers - are much cheaper than the others (also, you don't tend to change shader without changing shader constants and textures anyway, so it's a reasonable approximation). So I make some "sub-buckets" of the objects that share

the same state in the last two items. In each sub-bucket I pick the closest object to the camera. Then I sort the sub-buckets according to that distance (closest first), and draw them in that order. Within each bucket, I draw all the objects that have each renderstate in whatever order they happen to be - all I care about is getting all the objects for one state together. This gets me a fair bit of early-Z rejection without spamming the hardware with too many expensive changes.

In theory by sorting within each sub-bucket you could get slightly better driver/hardware efficiency, but I suspect the effort of sorting will be more expensive than the savings you get. I've not measured this, so this is just my experience talking. By all means if you have the time, test these hunches and let me know if my intuition is wide of the mark. It wouldn't be the first time.

Note that obviously I do the "first nearest object" as I'm putting the objects into the sub-bins, not as a later sorting stage, and there's a bunch of other obvious implementation efficiencies. And I use a bucket-sort for Z rather than trying for high precision. So although it sounds complex (it's surprisingly difficult to explain in text!), the implementation is actually fairly simple and the runtime cost is very low.