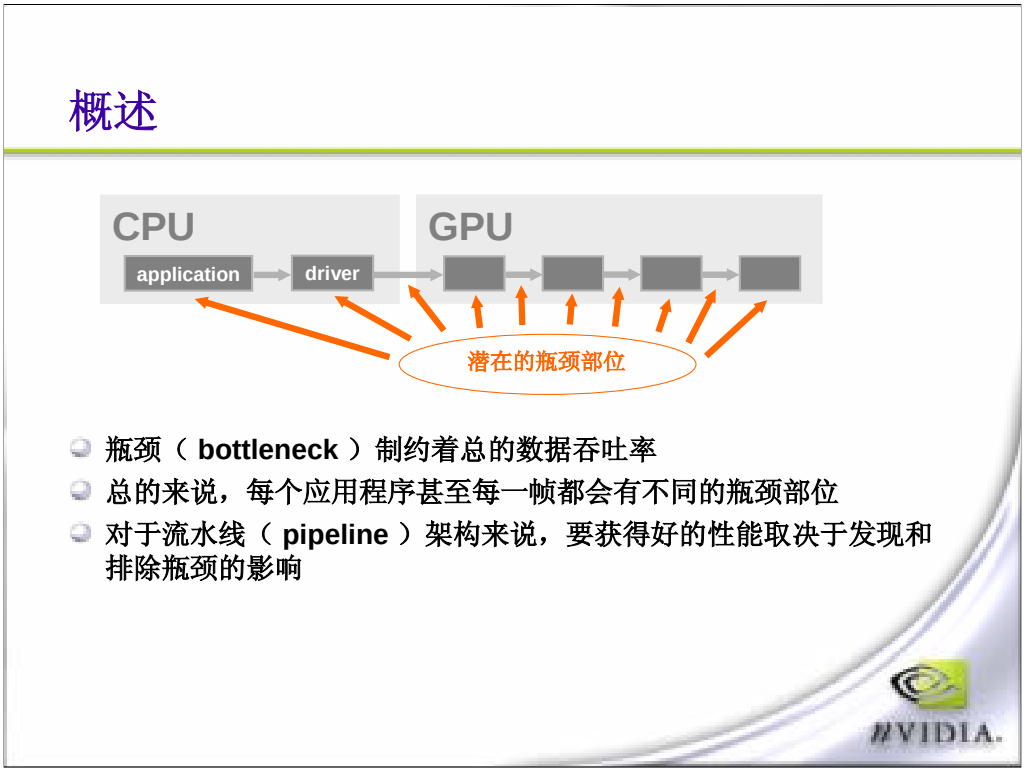
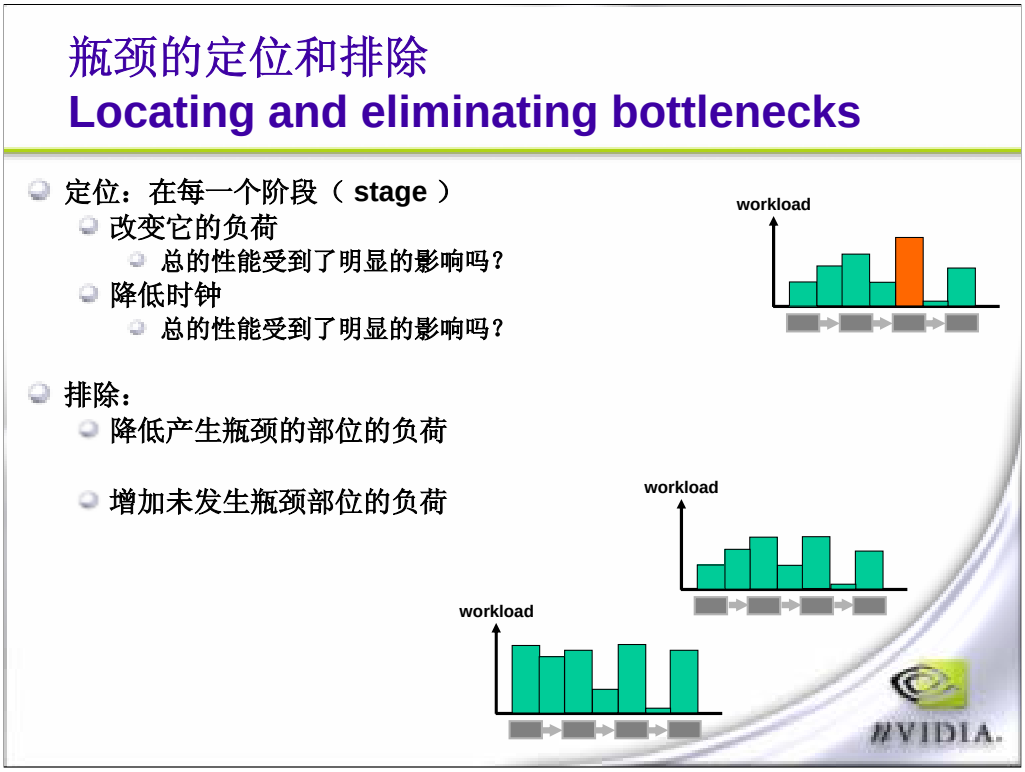
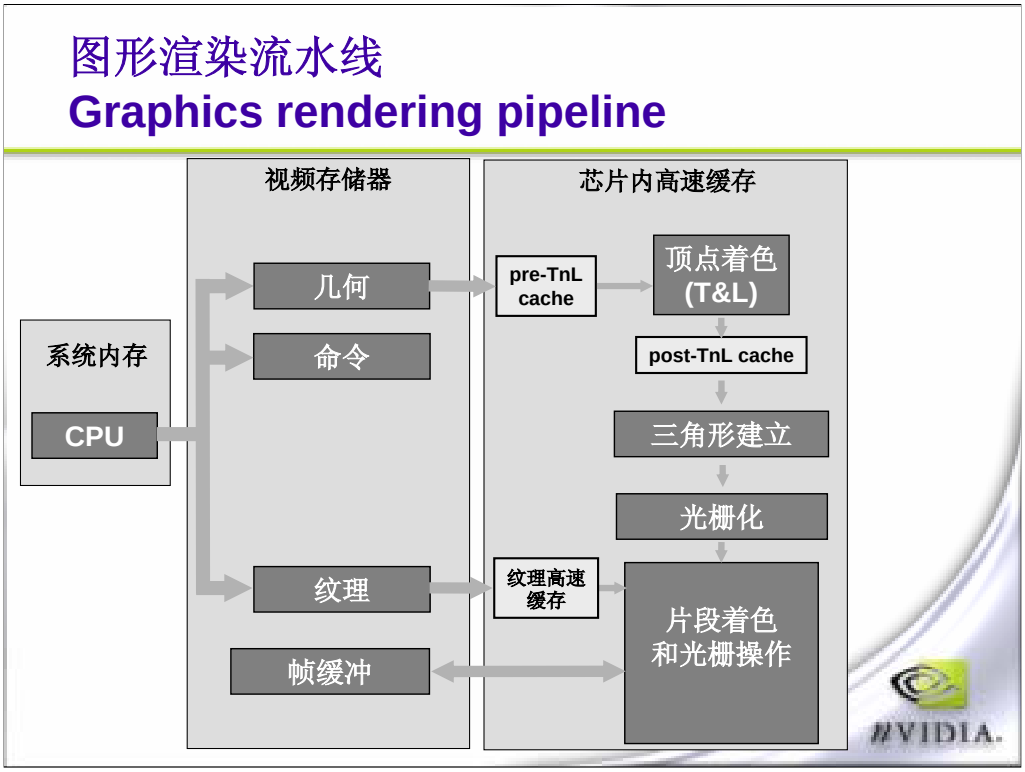
Hi. My name is Koji Ashida. I'm part of the developer technology group at NVIDIA and we are the guys who tend to assist game developers in creating new effects, optimizing their graphics engine and taking care of bugs. Today, I'm going to talk to you about optimizing the graphics pipeline.



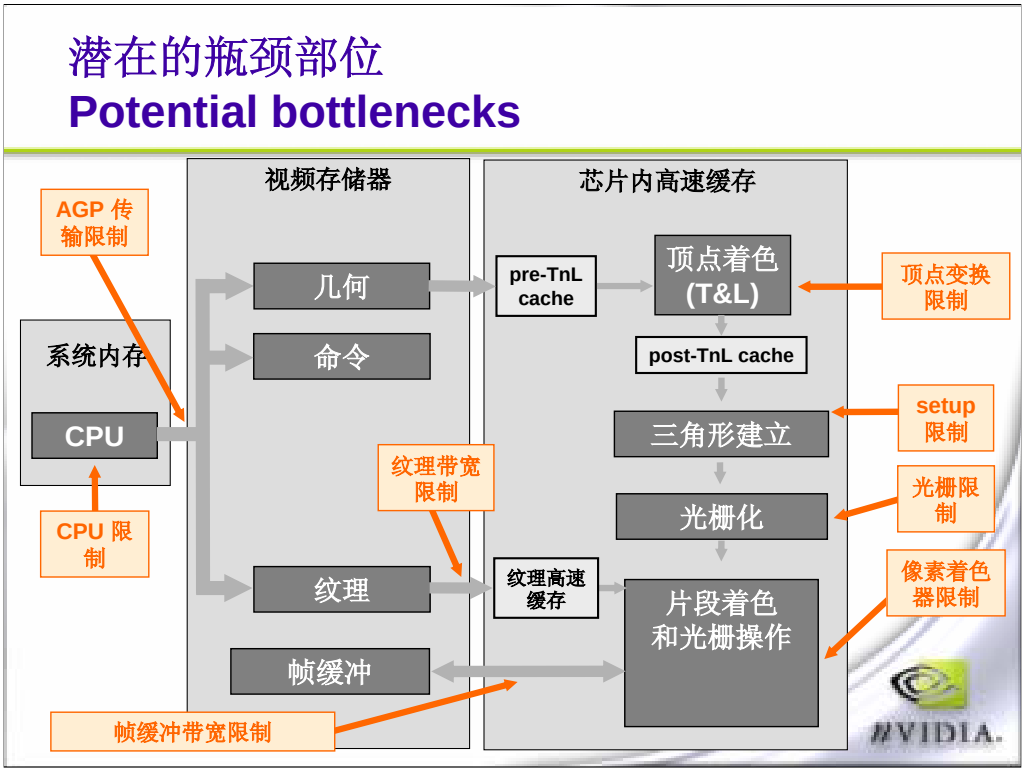
We're going to go through all the stages of the GPU plus talk a little bit about what happens on the CPU side that can bottleneck a graphics application. As we all know, in order to render a scene, the application which is running on the CPU must send data and instructions across to the device. It communicates with the device through the driver. Then, once the device has the data, it processes the data using the graphics chip itself, and finally writes it out to the frame buffer. Because this whole process is a single pipeline from the CPU to the last stage of the GPU, any one of those stages can be a potential bottleneck. The good thing about a pipeline is that it's very efficient and, in particular, it's very efficient at rendering graphics because it can parallelize a lot of operations. The bad thing about a pipeline is that, once you do have a bottleneck, then the whole pipeline is running at that speed so you really want to basically level off all your stages in the pipeline such that they have equal workloads. Hopefully, by the end of this presentation, you'll have a very good idea of how to either reduce the workload on a certain stage or at least increase the workload on the other stage such that you're getting better visual quality.



Now, please remember that, for any given scene, you're going to have different bottlenecks for different objects, different materials, different parts of the scene and so it can be fairly difficult to find where the impact is. Now, you do have two choices once you've identified the bottleneck. One thing you can do is try to reduce the overall workload of that stage and thereby increasing the frame rates or what you can do is increase the workload of all the other stages and, therefore, increasing the visual quality. Now, you want to choose which one you do, depending on your target frame rate so, if the application is already running at 80 frames per second, you may, instead of seeking to run at 100 frames per second, increase the workload on the other stages. The basic theory that we're going to use is that we're going to step through each stage and vary its workload. If it is the bottleneck, then the overall frame rate's going to change. If it isn't, then we're going to see no difference, supposedly.



This is an overall view of the graphics pipeline. To the far left, we see the CPU, which is where the application and driver are going to be running, and the CPU is communicating with the graphics device through the AGP bus. These days, we have AGP8X, so it's running pretty fast. It communicates both with the graphics chip itself and with video memory through the graphics chip memory controller. In video memory, typically what's stored is static geometry or semi-static geometry, also the command stream, textures, preferably compressed, and, of course, your frame buffer and any other intermediate services that you have. Then, on the actual hardware chip, we have some caches to do buffering and to ensure that the pipeline's running as optimally as possible.

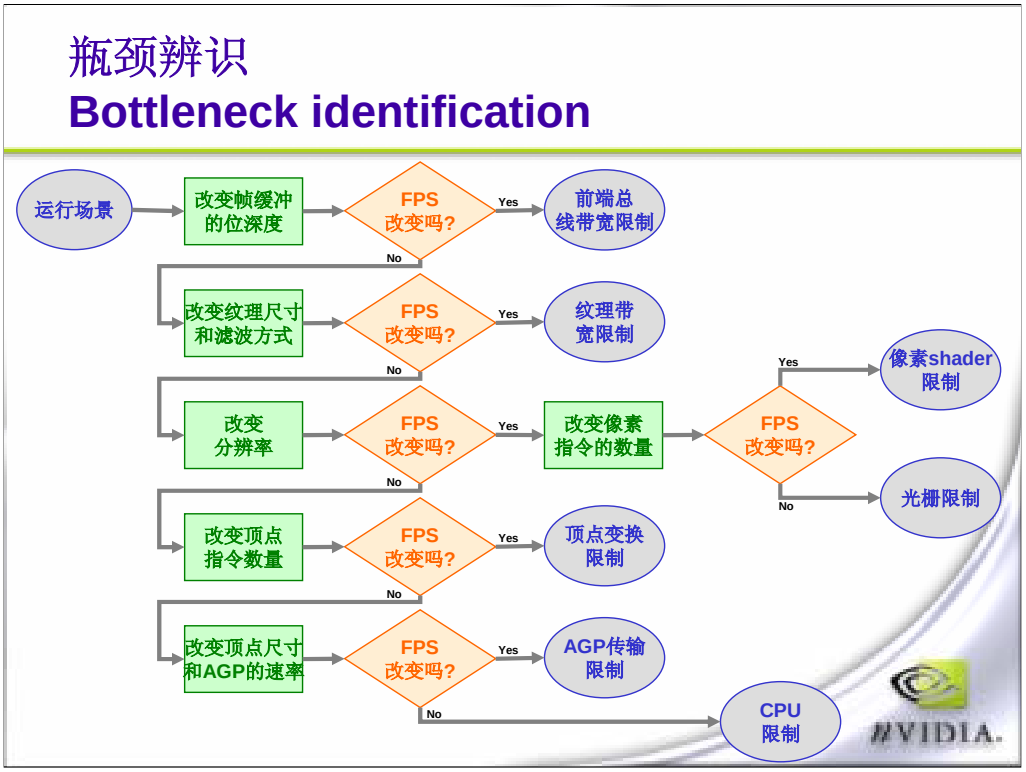


The first real module that we encounter is the vertex Shader, also called the vertex program units, and this is where you're going to do your transform and lighting of vertices. These vertices then get put into some sort of vertex cache that operates in different fashions. Some operate as FIFO's, some, as leased recently used, or LRU, and, obviously, current high-end cards have larger caches than the older generation and mainstream cards. Then the triangle setup stage is reached. This module reads vertices from the cache and this is where basically the polygon is formed. Once the polygon is formed, the rasterization is where it gets broken up into pixels. So now the next stage only accepts pixels and this is the fragment shading or pixel shader stage. The pixel shader stage is typically where a lot of time is spent these days and we're going to spend a good amount of time analyzing this. After the pixel shading stage is the raster operationm meaning the alpha blend and stencil Z buffer. These can contribute to the bottleneck but typically not. And, finally, you can have traffic from the pixel Shader and raster module to the frame by frame back, right, and back because you can do alpha blending, you need to read stenciling to read a Z and also the fragment Shader can read textures so you can have a texture bottleneck as you're accessing memory again.

图形渲染流水线的瓶颈
Graphics rendering pipeline bottlenecks

- 术语“转换限制（ **transform bound** ）”常常意味着瓶颈出现在光栅化之前的任何阶段
- 术语“填充限制（ **fill bound** ）”常常意味着瓶颈出现在“**setup**后的任何阶段”
- 转换和填充限制可以在一帧处理过程中同时存在！

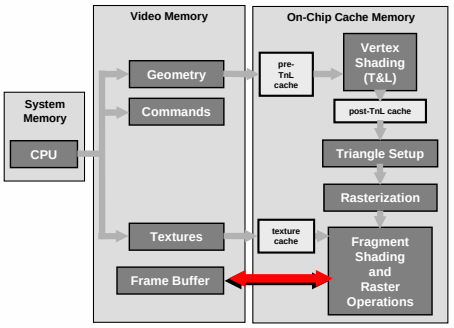
When we're talking about bottlenecks, we tend to oversimplify the pipeline. We talk about being transfer-bound or fill-bound and, when we mean transfer-bound, it's really-- we're saying that it's anything before the polygon gets broken up into pixels; i.e., before they're rasterized. And when we mean fill-bound, we mean a number of things that happen after the polygon is formed, and that could be raster, texture-bound, fragment Shader bound or frame buffer bound.



This is a chart that you can use to identify what part of the graphics pipeline is your bottleneck so I put the slide into the presentation just so you can have it as reference and then, over the next few slides, I'm going to explain each block.

帧缓冲带宽限制
Frame buffer bandwidth limited

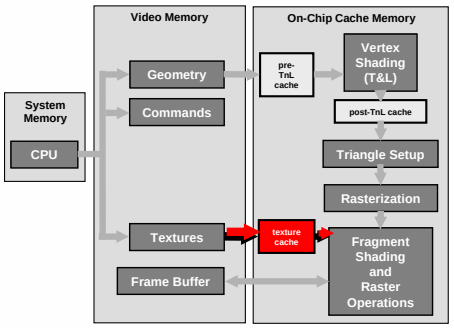
- 改变所有渲染对象的颜色深度 **(16-bit vs. 32-bit)**
  - 如果帧率改变了，应用程序就是帧缓冲带宽限制的



So let's start at the bottom of the pipeline. This is the part where the graphics engine is trying to read from the frame or to frame. Simply to access off-screen surfaces, not necessarily textures. So the easiest way to identify if this is your bottleneck is to vary your bit depth. So if you're running at 32 bits per pixel, run it at 16. If the frame rate varies then you're probably frame buffer limited. This isn't necessarily the case all the time but it does happen.

纹理带宽限制
Texture bandwidth limited

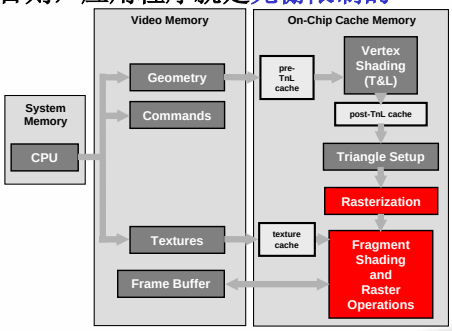
- 否则，改变纹理大小或者是纹理滤波类型
- 强制MIPMAP LOD（细节等级） 偏量到+10
  - **Point filtering vs. bilinear vs. trilinear vs. anisotropic**
  - 如果帧率改变了，应用程序就是纹理带宽限制的



Let's move a little further up and we hit the fragment Shader but, even more importantly, the data that the fragment shader is pulling. This data is texels and texels can be a bottleneck if there are too many of them or if they don't hit the texture cache efficiently or if they're being filtered with an expensive filter that may not be improving the visual quality of the scene. So if you vary the LOD of your MIP maps and basically force them all to the smallest size possible, you'll be able to figure out if the texture resolutions are not adequate and I do hope that you are using MIP maps because it's very important in today's hardware. Next, you can vary the filtering. So if you're using bilinear or tri-linear, you can drop down to point filtering, for example, or you can increase the anisotropic ratio to 8X and, again, just check the frame rate.

片段或者光栅限制
Fragment or raster limited

- 否则，改变所有渲染对象的分辨率
- 如果帧率改变了，改变片段（**fragments**，或者说待处理像素）程序（**fragment programs**）的指令数量
  - 如果帧率改变了，应用程序就是受限于片段着色器（**fragment shader**）
  - 否则，应用程序就是光栅限制的

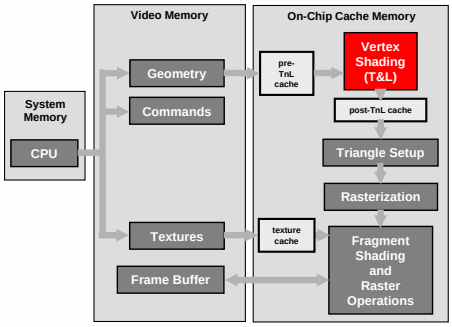


NVIDIA.

Next, we have the fragment program and the rasterization module. The first thing you should do here is vary your render target size, not the bit depth but the actual resolution so it'll go from 640x480 to 1024 to 1280 to 1600, for example. And if your frame rate varies, then you know that your bottleneck is somewhere here, somewhere in the fragment shader or the rasterizer. And to check if you're limited by the fragment shader, you can simply vary the number of fragment shader instructions. If the frame rate doesn't vary, then you are raster limited.

顶点变换限制
Vertex transform limited

- 否则，改变您的顶点程序（ **vertex programs** ）的指令数量
  - 注意：不要增加那些很容易被优化掉的指令
  - 如果帧率改变了，应用程序就是顶点转换限制（**vertex transform limited**）的

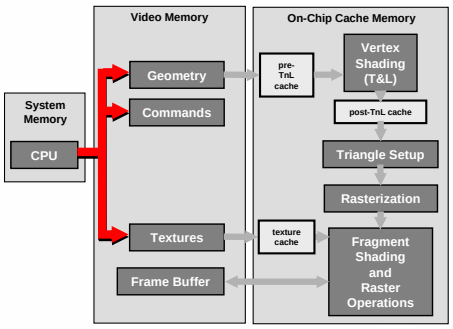


NVIDIA.

Next up the pipeline, we have the vertex engine and the quickest way to vary the load on that is to add or remove the vertex shader instructions. The only caveat here is that the driver compiler and optimizer can easily detect if you are doing operations that are optimizable and therefore easily removed. So try to have dependent instructions and write out some random color to the diffuse register. Again, if the frame rate varies, then this is where your bottleneck is.

AGP传输限制
AGP transfer limited

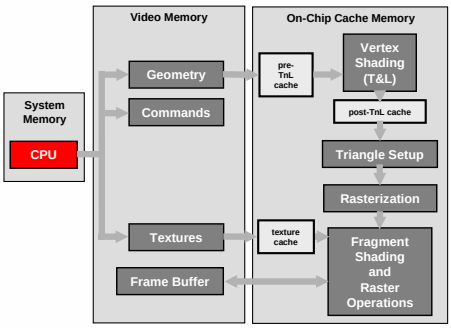
- 否则，改变顶点格式尺寸或者AGP传输率
  - 如果帧率改变了，应用程序就是AGP传输限制的



Even further up the pipeline, we have our AGP bus and this used to be a large bottleneck. If you have a lot of dynamic geometry, this could still be a bottleneck for you and, in that case, you can test it out by varying your vertex format size. Typically, you want to keep vertices to 32 byte multiples, so 32, 64, et cetera.

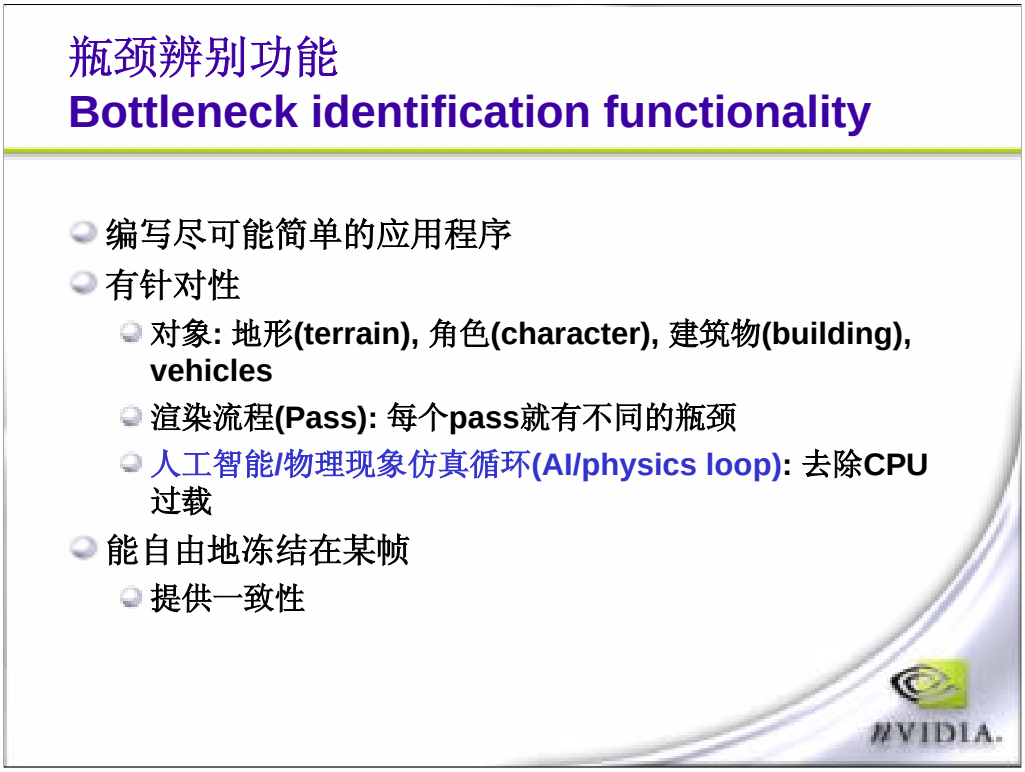
CPU限制
CPU limited

- 否则，应用程序受限于CPU



Finally, if none of that varied your frame rate, then you are CPU limited.

Now, there are a couple of quick tricks you can use to figure out your bottlenecks. First thing to do is to run the hardware on different CPUs, and that way you're varying your CPU and you can immediately tell if it's CPU limited if, and only if, the percentage increases are the same; i.e., the megahertz percentage matches the performance percentage increase. Now, the other thing you can do for AGP bottleneck is to vary the AGP transfer rates in your BIOS so you can drop it down to 1X and see if that impacts your frame rate. The other thing is using Power Strip or some utility, you can under clock your GPU and it will obviously affect the whole pipeline so at least you'll be able to tell if it's somewhere within the graphics chip. You can also slow down the memory clocks and, in that case, you'll be slowing down the texture reads or frame buffer read/write.

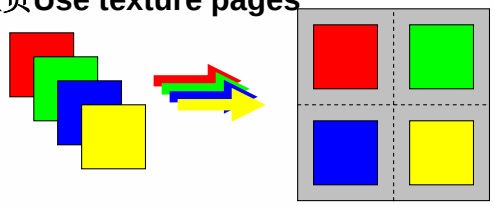


Now, we've had trouble in the past identifying bottlenecks and what helps us a lot is if the application is written in such a way that you can toggle different parts of the scenes. For example, if you can remove the terrain or remove all static objects and just leave character onscreen, then you can actually profile just the characters or, if you remove everything else and just render terrain, you can profile just the terrain and what this is doing is trying to identify the different bottlenecks that are going to occur on your different materials and objects, because each one of those can have a different workload at different stages. The other thing that would be interesting is for each of those objects, if you had multi-pass rendering, toggling the different passes because each pass can also have a different bottleneck. And, finally, something that would be interesting to do is to remove the CPU overhead or the application use of the CPU by, for example, pausing the AI and physics loops. Doing this will also allow you to basically pause the game such that you have no other variables affecting your frame rate and, that way, have a steady scene with which you can toggle the different objects and identify the different bottlenecks for each of them.

Now, there is one bottleneck on the CPU which is rather large and this has turned out to be batching. What we mean by batching is the number of draw primitive calls that you make. So a draw primitive call consists of sending a set of polygons to the driver, hence to the device, with a certain state, right, the state that you set before -- could be texture states, render states, or sampling states, et cetera. So that's what we define as a batch. Now, it turns out that, a P-4, 3 GHz CPU can only send 100,000 batches or 100,000 draw index primitive calls per second. What this means is that, at 50 frames per second, you only have 2,000 draw calls per frame. What this also means is that you can either choose to render 2,000 triangles per frame or 2 million triangles per frame. If you're using all of the 2,000 draw calls per frame, then ensure that you're sending a lot of triangles for every draw call.

总体优化：批处理（ Batching ） II

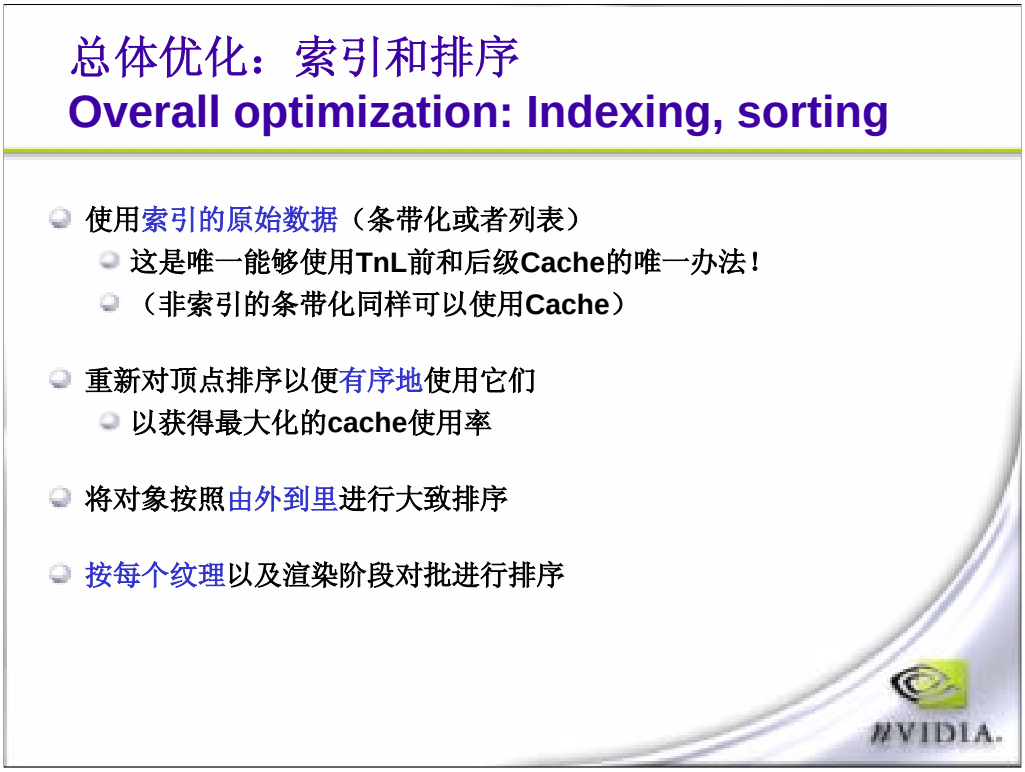
- 消除小批作业 (继续):
  - 使用退化的（ degenerate ）的三角形将条带（ strips ）连接在一起
    - 硬件剔除面积为零的三角形是非常快速的
  - 使用纹理页Use texture pages



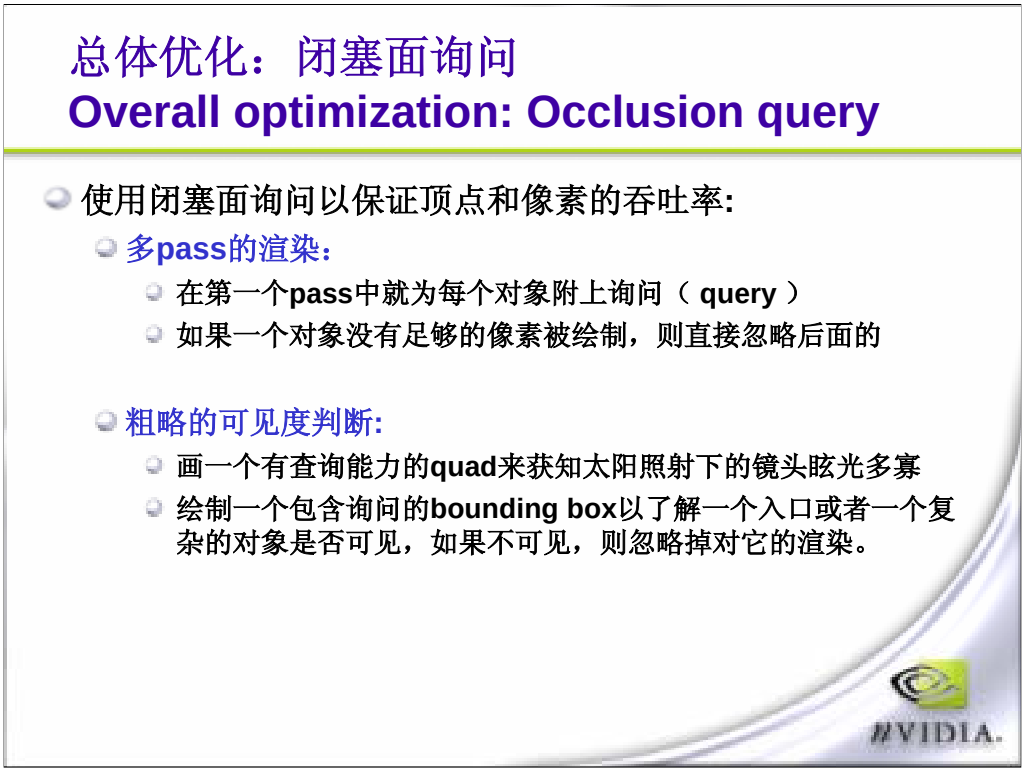
  - 使用顶点着色器（ shader ）将需要的几何打包在一起。
    - VS2.0和VP30具有256个4D向量的常数

NVIDIA.

Here are a couple of things that can help out with batching. You can join up objects or parts of objects by using degenerate triangles. Degenerate triangles are basically zero area triangles where two of the vertices are the same. Now, the culling engine in the chip removes these zero area triangles at basically no cost so don't be worried about sending inefficient triangles to the hardware. We'll take care of it. Next thing you can do is to use texture pages. You often render geometry with different draw calls because they use different materials and the materials use different textures. Well, a good trick that you could use is to store all your textures into a larger texture or effectively a texture page and the only thing you have to worry about then is the border around the texture because of MIP mapping and also filtering so the limitation is that you can't tile these textures so, for any other geometry that doesn't use tiling, this would be a good optimization. Lastly, it's a good idea to use vertex shaders to batch instanced geometry. What this means is that you can render several objects in one call, let's take a robot, for example. Each jointed segment of the robot is rendered with a different matrix. Usually we have to issue a draw call for each segment. What you can do, however, is that for each vertex, you send an index number which tells what matrix the vertex needs to be transformed with. And in the vertex shader, you find the correct matrix from the constant vector using the index, and use that to transform the vertex. This lets us to send all the vertices of a robot in one draw primitive call.



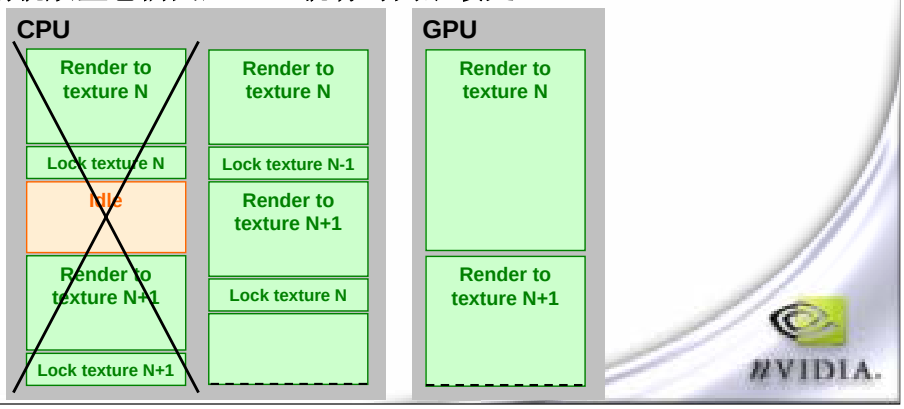
Another general rules for optimizing the engine is to use indexed primitives. Typically, strips are not long enough to efficiently render geometry all the time but indexed lists actually allow us to make best use of the vertex cache because non-indexed triangle lists do not use the vertex cache. The other thing that you want to do is to reorder your vertices such that they hit the same area of memory. Our chip has a DMA cache, and this cache stores vertices that were fetched from memory and if you're loading vertices from very different areas in memory, then you're going to be thrashing the cache and transferring more data than necessary. Next you want to order your scene front to back. What this is doing is taking advantage of our Z-cull technology. Z-cull has been around since GeForce 3 days and is also available in other people's hardware. What it does is accelerated removal of pixels that will never be seen and it's important to take advantage of this to get better performance, especially for scenes with high depth complexity. This can be actually used for a Z first technique that I'll talk about later. And, lastly, you want to sort your batches per texture because texture changes are the hardest hitter of all the state changes possible.



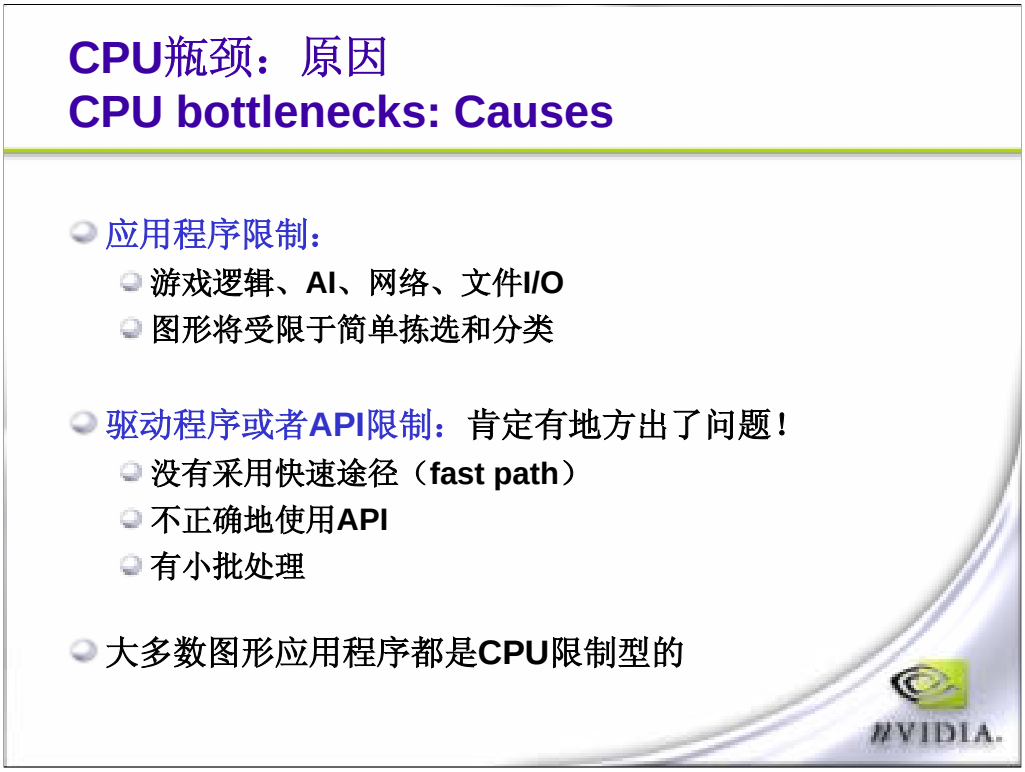
Recently exposed in the DX9 API (but it's been available in OpenGL for a while) are occlusion queries. Occlusion queries use an ID-based approach to query if an object tagged with this ID has been rendered. You let the hardware process it, meaning that there is no CPU intervention, but the GPU doesn't actually write anything, it just does a Z check and then it returns a flag to the driver that says, yes, something's been rendered, and gives you the number of pixels, or no, nothing's been rendered. So what you can do with this is try to render a very low-cost bounding box for your object after laying down large scene occluders, render the rest of the scene, then go back and query the driver to see if that object could have rendered any pixels. If it did, then send the real object down that will actually be visible. A way to optimize this is, in the case of multi-pass rendering, just send down the first pass but tag it. And, at the end of the scene, draw the subsequent passes if, and only if, it passed the occlusion test. The other thing you can use this for is something like a sun glare, where you don't want to be locking the frame buffer to check if the sun pixels were drawn or not. What you can do is render the sun with an occlusion tag and then, afterwards, render the glares, depending if the sun was visible or not and, because we returned the number of pixels drawn, you can basically fade in and fade out the sun.

总体优化：小心资源锁定！
Overall optimization: Beware of resource locking!

- 一个锁定资源的调用**(Lock, glReadPixels)**如果放错了位置的话会成为潜在的一个阻塞部位：
  - **CPU**处于空闲，等待着**GPU** 以继续处理
- 如果可能的话要消除这种情况
- 否则就放置它们因此**GPU**就有时间继续处理:

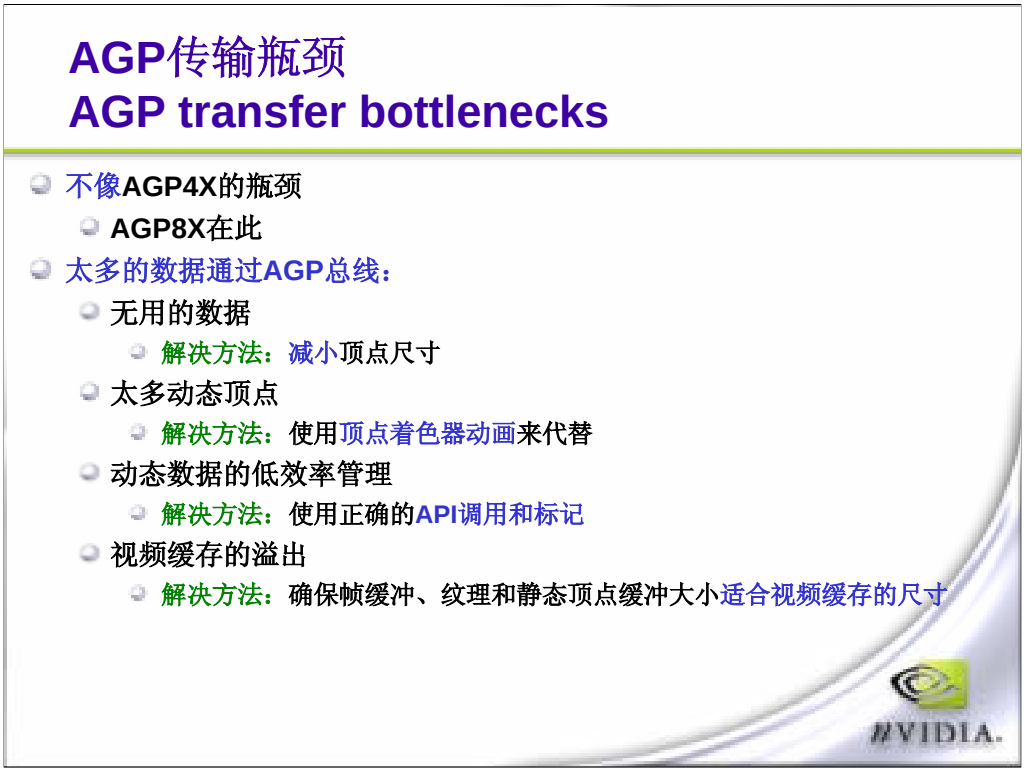


Resource locking basically means that the GPU is waiting to access a resource that, for some reason, you've grabbed. You've told us that you want to read from it, for example. This is bad. This will start synchronizing the GPU to the CPU. You want to avoid this at all cost. However, if you can't live with that, then what you can do is locking a resource that will not be used for another few frames and, that way, you ensure that the graphics unit is not waiting around for you to unlock it.

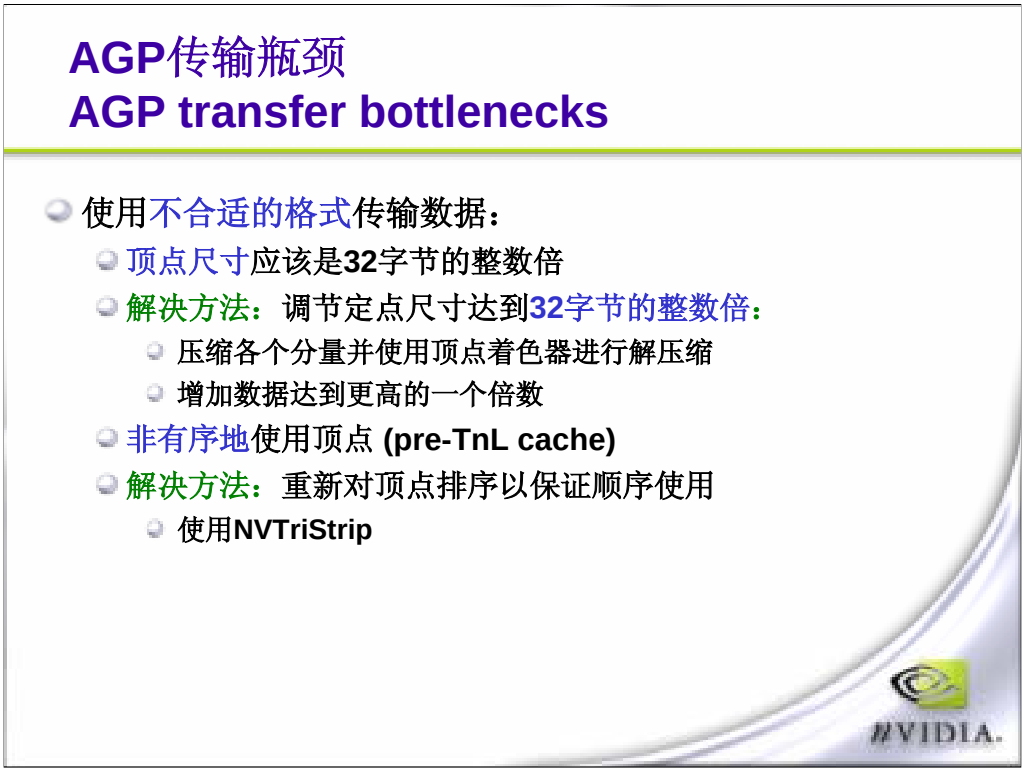


Now, let's move on to potential CPU bottlenecks. Some games are bottlenecked by the application itself, right, doing your usual AI physics, et cetera. In this case, it was probably limited by the graphics processing just to avoid rendering possibly simple geometry. Don't do anything more complicated than simple sorting, culling, and static LOD management. The other area where it could be bottlenecked on the CPU is in the driver or sometimes in the API. In these cases, then there's something very strange going on because we're having to do a lot of work to run the graphics chip. The biggest cause of a driver bottleneck is a great number of batches per frame and we covered the details before. So keep track of those. Also sometimes you may be doing begin and end scene for every draw call. That's not good, either. So just check that you're doing something reasonable with the API. And it should be noted that most graphics applications these days are CPU limited, even with your newer CPUs.

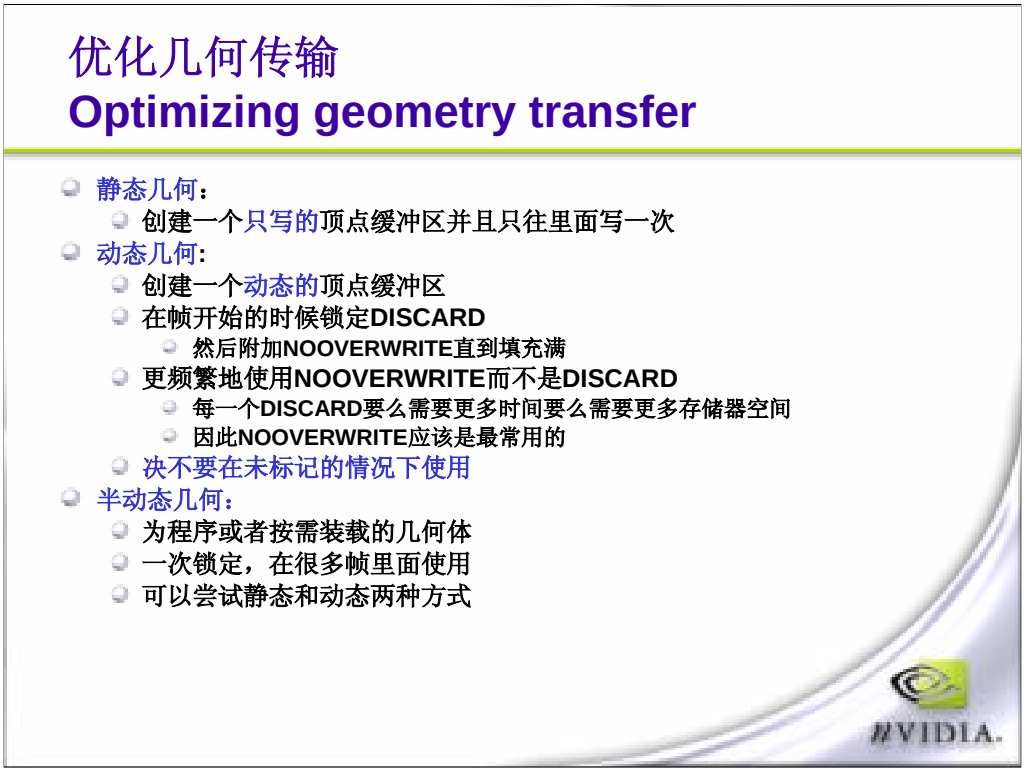
The easiest way to check CPU limitations is to use V-tune, for example. V-tune should let you know how much time you're spending in the application, in the API, and/or in the driver. We do have something called NVPerfHud which should help you out, not only for this but for a couple of other things, and I'll show you an example later. A solution for the batch size problem is to try to increase the number of polygons per batch and, therefore, reduce the number of batches. We talked a little bit of how to do this using texture pages and we're going to go into some detail about how to optimize the other parts of the GPU but I want you to remember that whatever I'm saying about the optimizations you do for the GPU, don't do them if they're going to impact or increase the number of batches dramatically. So when you're optimizing for batches, do it at the expense of the GPU.



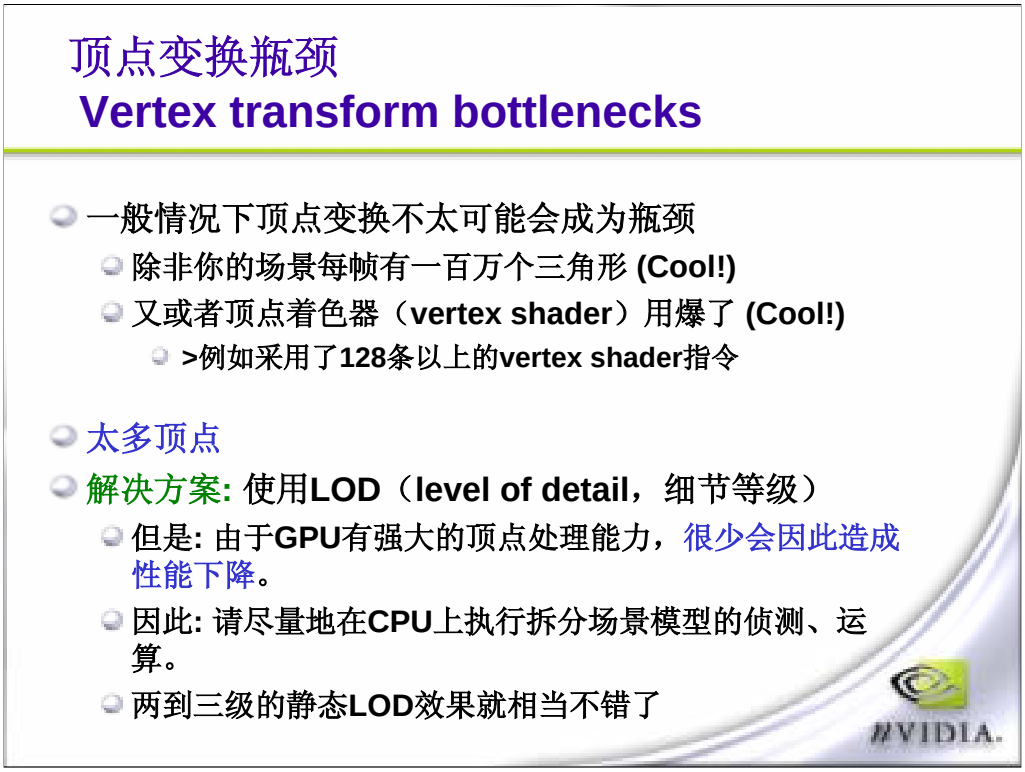
Now, let's step through all the stages of the pipeline after the CPU. The first thing we hit is the AGP bus. Like I said, most of the time, you're not going to be limited by this. The AGP8X bus transfers data at a rate of 2 gigabytes per second. That's plenty of bandwidth to transfer tens of millions of vertices per second. If you do find that you're limited by this, you can reduce the vertex size. We do have vertex formats that are 16 bits instead of 32 bits per component and that should help quite a bit. Also, if you find that you are using too many dynamic vertices, then chances are you're not using vertex Shaders enough. You can do complicated animations on the vertex shader such as skinning and morph targets and you should be doing this instead of manipulating geometry on the CPU. Next, if you do need to have a lot of dynamic data, you should be managing this data with the proper API calls and with the proper flags. We'll go into a little more detail of the correct flags for vertex buffers. And, finally, just make sure that everything is fitting into video memory. So what can happen is that sometimes textures can spill over into AGP memory and then, at some point, your scene hits this texture that's in AGP memory and you're slowing down because you don't have the 20 gigabytes per second to video memory access, you have the 2 gigabytes per second to AGP memory access and the GPU is then sitting idle waiting for this data to be transferred.



When transferring vertices across the AGP bus, you want to make sure that they are in 32-byte multiples. So you're going to have 32, 64, et cetera and ensure that they're being transferred sequentially. We do have a triangle stripper called NVTriStrip that you can download from our site and this will actually reorder the vertices to be in sequential use and make optimal use of the post-T&L vertex cache.



When managing your vertex buffers, you want to create them with a write-only flag if they're going to be static. If you're doing dynamic geometry, then you want to create a dynamic buffer with the dynamic flag, you want to lock with discard and overwrites and the rule of thumb for this is that an overwrite doesn't hit us as bad as a discard. Discard means that the driver has to allocate a new vertex buffer somewhere in order for you to write to it while the GPU is actually using the previous vertex buffer at that point in time. And the takeaway from both of these rules is that you should never not use a flag. Always, always, always tell us how you're going to be managing your geometry. For semi-dynamic geometry, you can try a bit of both. What we've found is that, if you only update your vertex buffer once per frame, for example, you can create it as static and do a discard on it. And that's usually pretty good because it makes it reside in video memory. If you're creating a dynamic vertex buffer, it'll typically reside in AGP memory so just bear in mind where you want to store it and how often you're going to modify it.



Next in the pipeline we have our vertex engine. Now, for the GeForce FX 5900 ultra, we can transform a whole lot of vertices -- we're talking in the order of hundreds of millions of vertices per second. So I really doubt that you're going to be bottlenecked by your Shader instructions. Usually you could be bottlenecked here because your vertices are not in strip order, meaning that you make very poor use of the vertex cache. But let's say that you have too many vertices in general, what you could do is use static LODs. Basically drop off the LODs in the distance but, don't do any smart things in the CPU to do dynamic subdivision, for example. It's possible to do but you don't want to be doing it for all your objects.

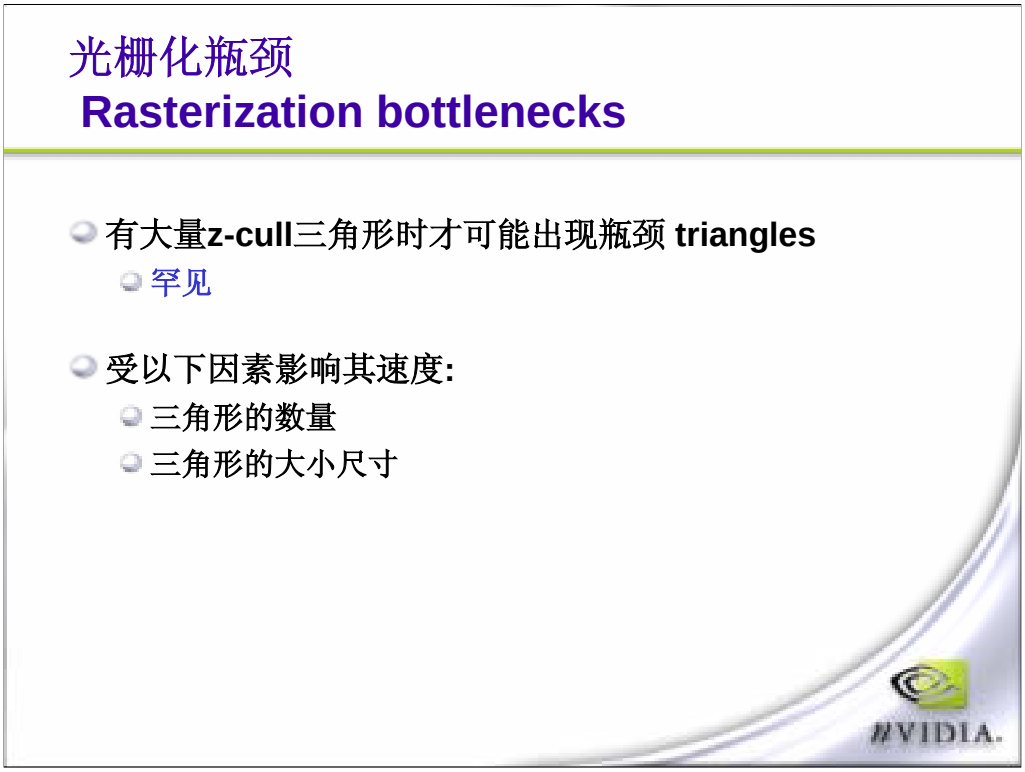
The next thing in the vertex engine is the number of instructions. Now, as I said before, you can transform a whole lot of vertices and, obviously, the more instructions that you execute per vertex, the slower you're going to go. Sometimes you are applying lights that, for example, will not actually light that object. So you may want to be toggling these, switching between different Shaders to ensure that you're optimally processing that vertex. The next thing is that you may be using a lot of the newer instructions such as looping or branching. Looping itself is not particularly bad. Branching starts to get worse. I mean, just like any processing unit, the more you branch, the less efficient you get and it's a lot harder for us to do branch prediction on the GPU. So try to limit it, for now, though in the future, this is going to be a faster approach to-- especially for increasing batching.

Sometimes, when we write Shaders, we write them for convenience. In other words, we just put in a few constants, maybe multiply them in the Shader itself, instead of actually calculating them on the CPU. So these kinds of compilations can generally be removed; however, remember that you shouldn't do this if it's going to impact the number of batches that you're sending across. So, for example, it might be a good idea, if you have lots of batches, to concatenate them and do some per object calculations on the GPU but, if you're not suffering from batching performance and you're suffering from vertex Shader performance, then take out the instructions, basically just constant instructions, and leave them to the CPU. Next, recall that the vertex Shader is a 4D unit. In other words, it's got four units that operate on a vector simultaneously and it's got lots of complex instructions that have been heavily optimized by our architecture team. So you can use a combination of those complex instructions and also remember that you have vector operations that can be swizzled so you can have inputs that map to different source inputs and then output to maybe just one, two, three or even four components of the vector without having to start moving data around yourself. And that brings us to the next point which is we very, very rarely see a need for a move instruction so every time you see one, make sure it's for a very good reason. And, finally, there's a lot of work that's probably not necessary for objects that are really far away. So before we talked about reducing the number of vertices on objects that have lower levels of detail for when they're in the distance. Well, you can do the same thing with shaders so one example is, if you have a character that's skinned with four bones, well, does he really need four bones when he uses only a few pixels on the screen? Chances are not and you can drop that to two or maybe even one bone. And another thing that you could do is, instead of reducing the bottleneck here, you can just increase the bottleneck somewhere else and you can either increase the resolution or enable antialiasing or anisotropic filtering.

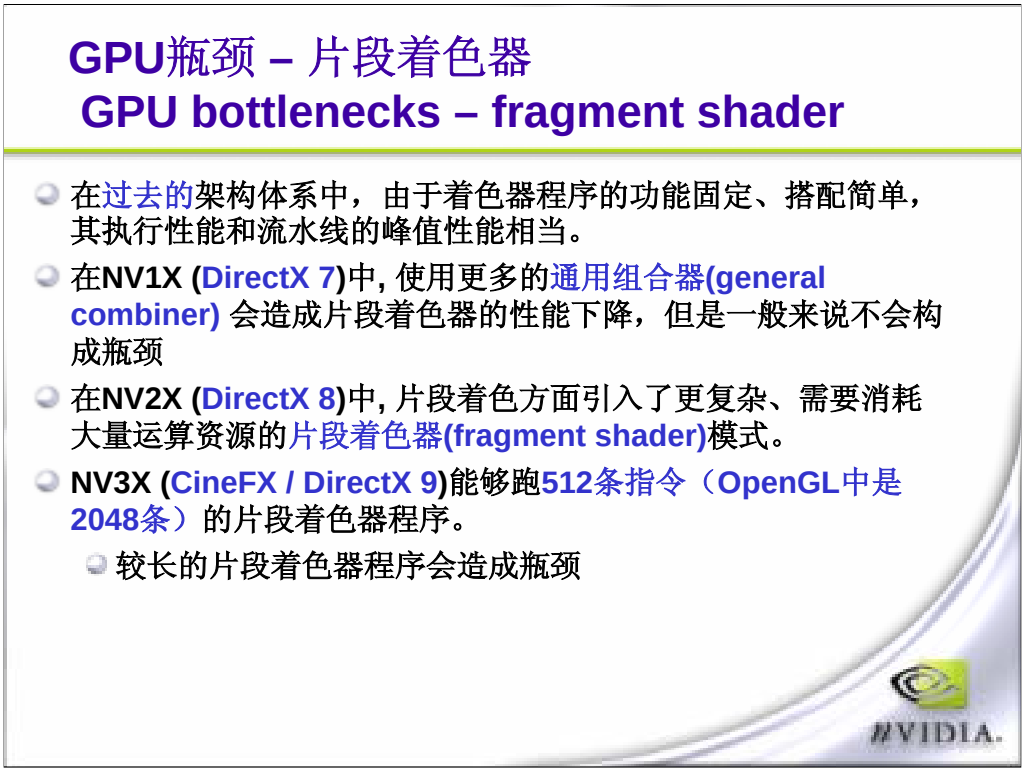
坐标设定（Setup）操作瓶颈
Setup bottleneck

- 几乎从来就不是瓶颈
  - 除了专门针对这类操作的性能测试程序外
- 该操作的性能会受以下情况影响：
  - 三角形的数量多寡
  - 需要进行光栅化处理的每个三角形顶点属性数量
- 想提高Setup操作性能的话，请：
  - 减少合并为真正三角形的比率
  - **But only if that ratio is substantial (> 1 to 5)**

Next module in the pipeline is the setup unit and this is where the vertices get made into triangles or quads. This is almost never a bottleneck so you probably don't have to worry about it. It's influenced by the number of triangles or the attributes that each triangle has. In other words, it's got to set up each of the colors, the texture coordinates or whatever other data you're passing around so the more of that data you have, the longer it'll take to set up that triangle but, typically, this is insubstantial.

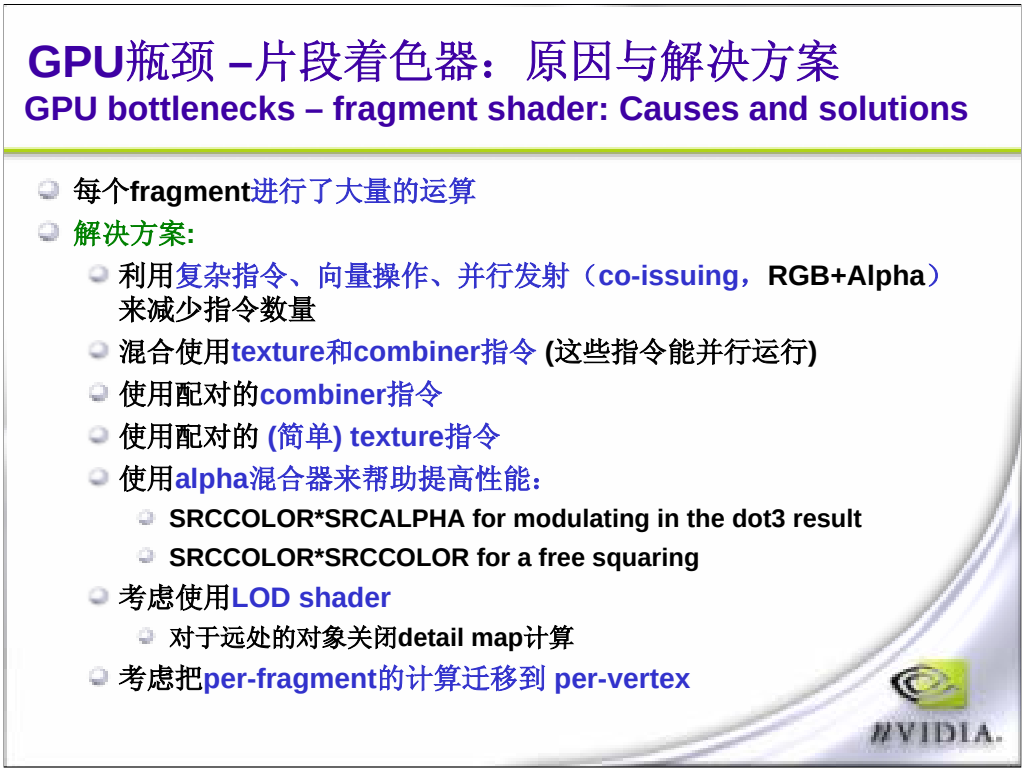


Next is the rasterization module which breaks up the triangles into pixels and, again, this is not a module that typically causes a bottleneck. The only time that this is going to be a bottleneck is if you have a lot of non-visible triangles that are being Z culled. Basically while this triangle's being Z culled, it's introducing lots of bubbles in the pipeline and effectively generating no-ops for all the stages downstream from this rasterization module.

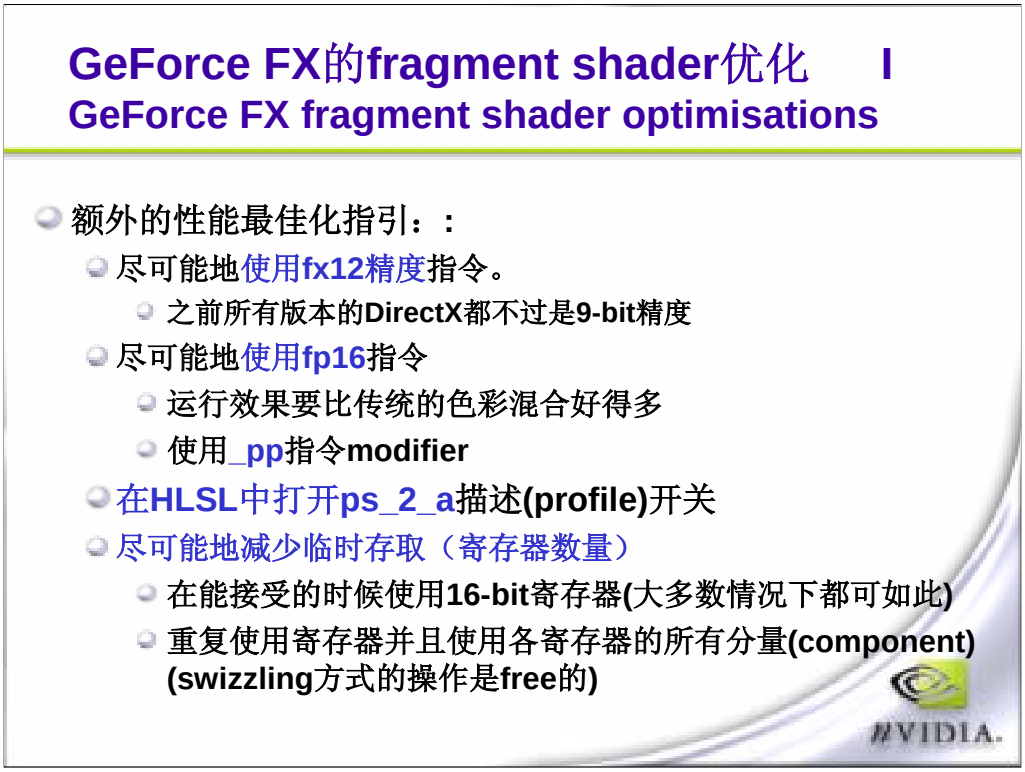


So now we get into the really interesting part, the fragment shader, and this module is much more complicated than it was in the past. What this means is that there is a lot more opportunity to make it go slow. Now, with the CINE-FX architecture, basically all of the GeForce FX family, you can run 512 instructions under DX-9 or 2048 in Open GL. And obviously the longer the shader, the slower it goes and it's very easy to write a long fragment shader, especially when using high level languages. Again, when we're writing shaders, we don't necessarily think about the performance of them until we start running them and you are going to have to write them liberally at first and then go back and optimize them and see where you can start squeezing more performance out of them. I'll go through some points that will cover these topics.

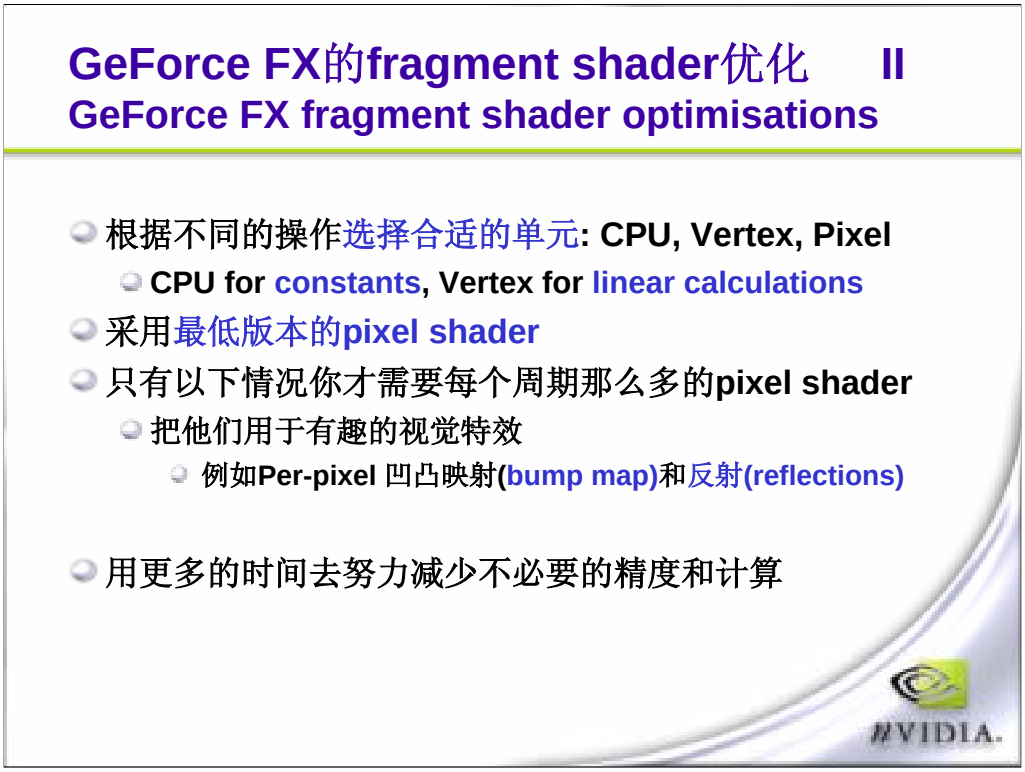
In the first place, you might have too many pixels and this will cause a bottleneck in the fragment shader. What you can do here is to try to use our Z cull engine more efficiently by rendering roughly front to back. Another thing that you can do if you have complex shaders or lots of multi-pass shading is that you want to do a Z only first pass. When you're doing a complex, single pass shader, you want to render Z only, effectively mask out your colors and you can do this using a render state. This will actually let us run incredibly fast because we know that you're not writing any color, so we can really kick the engine into turbo and spit out these pixels and do really fast Z checks. Now, what happens is, that next time when you actually render the real pixels, you will have already written the entire depth surface and you will only be lighting the pixels that are visible. So this is a way of decoupling geometric or your scene complexity from your pixel complexity. Now, what happens if you have multi-pass is you can do the same thing. Instead of writing Z only first, you just write the first pass, including Z and then, for subsequent passes, make sure you turn off Z writes but keep Z checks on.



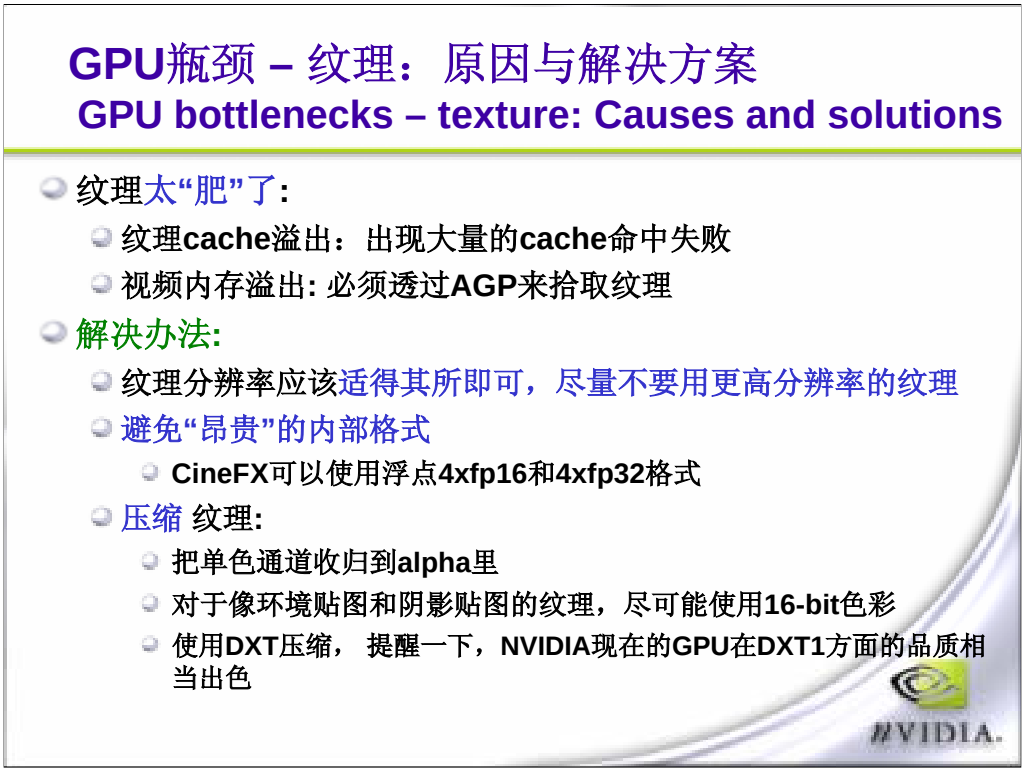
Another cause of this stage being a bottleneck is that you are just running too many instructions per pixel. Just like the vertex shader, this unit operates on 4-D vectors. You can make it actually run on a 3-D vector and a scaler, or what we consider Alpha. This is called co-issuing. You can actually save quite a number of cycles by doing co-issuing and it basically happens automatically as long as you're not using the alpha, the result of the scaler operation in your vector operation. Next is that you can optimize by doing even texture lookups and even map operations. On our hardware, you get two types of lookups and two map operations in a single cycle. If you're not running all these modules, if you're not using all these modules at the same time, then you're basically just going to be wasting their cycles. Next thing is that sometimes you might be able to get away with doing multiplication in the alpha blend unit so there's a final multiply out the back door. Do note that this happens at 32 bits per pixel. In other words, it doesn't happen with floating point formats. And, again, you can do shader level of detail, like we discussed for the vertex engine. Can you really see that the character that's using up five pixels of the screen is truly bump mapped or can you even see that there are eight lights on him? Chances are you can't so just, kill certain instructions, load a different Shader when the object is far away. Another thing that we talked about in the vertex engine was to move constant to work. In other words, by sending them to the CPU. Now, this also applies to the pixel Shader because, basically, if something is going to be linear, it can be done in the vertex engine. If something's going to be constant, it can be put into a constant register. So, consider moving stuff from the actual pixel shader to the other units.



As I said before, the CINE-FX architecture is more complicated than previous ones and we do have a variety of precisions within the fragment shader. The lowest precision is fixed point 12 base and it basically gives you a range of minus two to two. So this is pretty good, considering that, for all previous Direct X versions, the maximum precision was about 9 bits so 12 bit fixed should give you enough to do most of your usual lighting calculations. The next step up from there is floating point 16. It's actually what we call the half precision format and this is great for slightly greater range. It works fine for anything that was previously done with 9 bit precision, obviously, and it's accessible under the DX-9 assembly instructions when using the underscore PP modifier. In HLSL, you can access the half data type using PS2_A and this is also optimized for the CINE-FX architecture. Now, because HLSL or CG makes it very easy to write long, complicated pixel shaders, it's also very easy to dramatically increase the number of temporary registers that you'll be using. So the point is to use half precision registers if possible, i.e., the half data type or the FP16 float registers. And also try to reuse as many registers as possible so don't load up 16 textures into 16 different registers and then use each one individually when you could have actually just loaded two textures at a time into two registers, done a math operation, saved it into one of those whose value was discarded before and then reuse another two registers for the next set of texture loads. The other thing that you can do is that we do have arbitrary swizzling so you can, for example, store two 2-D vectors in a single 4-D vector because this is a 4-D engine and just swizzling, masking two components, is absolutely free.



In the end, what we're talking about is to choose the appropriate unit for the type of data that you're calculating. Constant data can definitely go into the CPU and then be transferred to the vertex and pixel engines as constant registers. Vertex engines used for data that's going to be interpolated across a triangle and the pixel engine is going to be used for data that's going to vary at every pixel. Do try to use the lowest pixel shader version available to you. Basically, what this means is, if you're only doing 4 texture lookups and 8 map operations and you're not doing a dependent texture read, then use PS-1.1 up to 1.3 version. If you're doing some dependent operations and you can fit within 1.4 shader, then use that and, if you're doing lots of dependent operations, then use, only then use pixel shader 2.0. This also guarantees that, for previous pixel shader versions, 1.X, that you're using the low precision register types so that actually saves storage and cycles. The same as you get a certain number of batches per frame, you can also think of getting the same number of pixel Shader cycles per frame. So you've got to basically choose where you're going to spend your cycles. Do effects that are going to distinguish your game visually, stylistically. Don't do effects that have a minor difference on a pixel and they're consuming several cycles to do so.

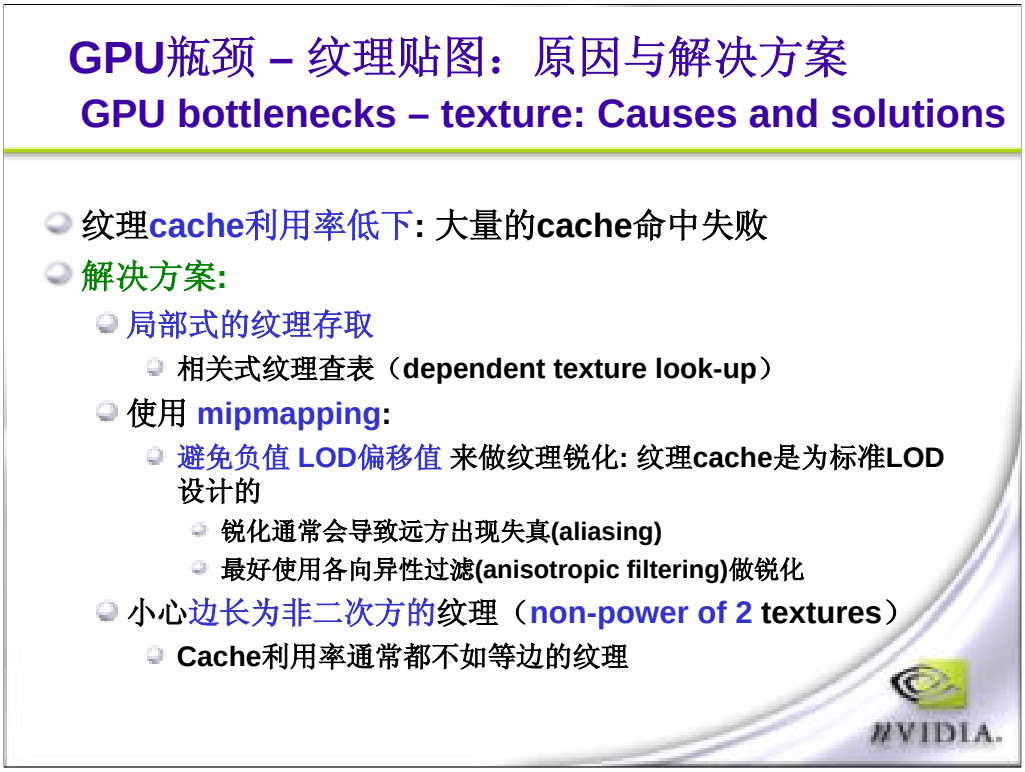


Okay. As part of the input to the fragment engine is the texture unit. The texture unit basically consists of the cache and a decoder for compressed textures. One of the problems that you can run into is that you are thrashing the texture cache a lot or you're loading textures from AGP memory. I mentioned how you can spill over from video memory so one of the things to do for the latter is to use compressed textures. Use 16-bit colors whenever you can. Use DX T-1 compression over the other compression formats but, if that's not good enough quality, then go on to DX T-3 and T-5. Now, textures should also be using MIP maps and when choosing the highest MIP level, basically choose it depending on how much of that texture is going to be on screen at a certain screen resolution. So you don't really want to be drawing a character with 2048 x 2048 textures when, at most they're using a 512 x 128 area of the screen. Now, with the new capabilities that CINE FX offers is that you can choose to store floating point data. Now, floating point data is expensive for two reasons. First, it uses a lot of bandwidth to video memory, whether it be through textures or through render targets. Secondly, it costs a certain amount to decode these floating point surfaces. So use them sparingly. Use them only when needed and you only really need them for high dynamic range and actually the next slide here shows you the format range and precision that they run at.

纹理格式, 范围, 精度
Texture format, range, and precision

Format	Range	Precision	Blend	Render	Texture	Filter
A8R8G8B8	2.4 dB	8 bit	Y	Y	Y	Y
RGBE8	76.8 dB	8 bit	N	Y	Y	N
D3DFMT_R16G16	4.9 dB	16 bit	N	N	Y	Y
RGBA16F	9.3 dB	11 bit	N	Y	Y	N
RGBA32F	76.8 dB	24 bit	N	Y	Y	N

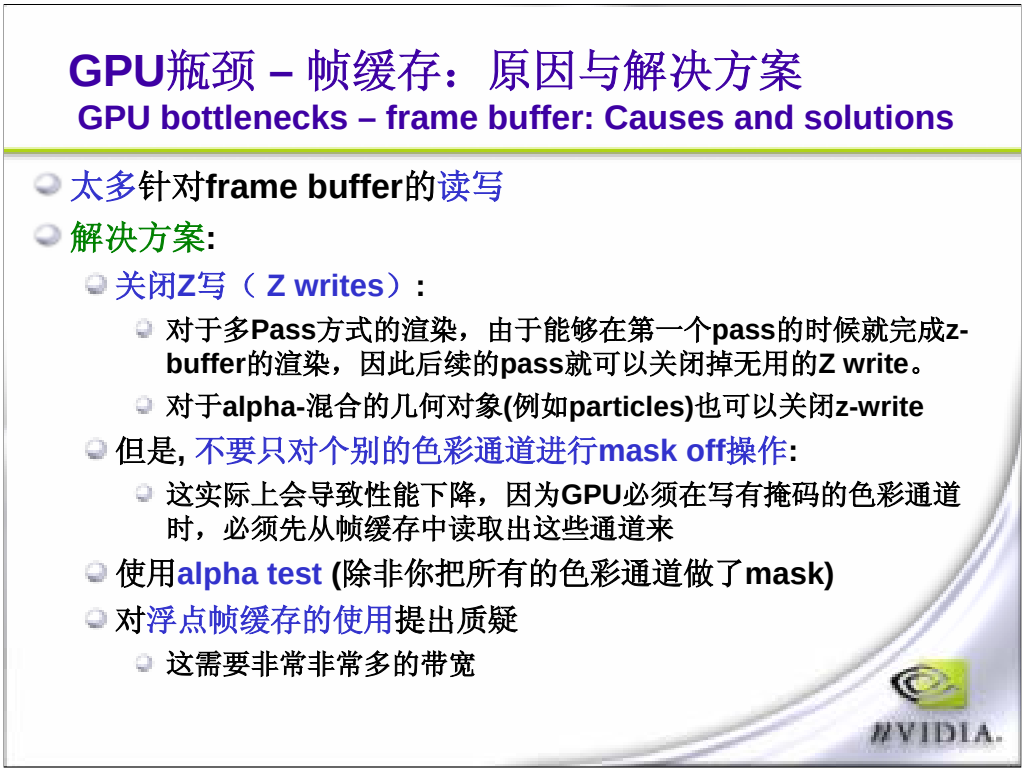
So your typical 32-bit ARGB format gives you 256 levels, it's not much but it supports all the blend, render, texture and filter operations. ARGB-8 is basically the same formats but you're using the alpha as an 8-bit exponential that's shared across the three color components. Now, this gives you a lot more range; however, even though it's a standard format, blending and filtering doesn't really work on that because the exponent can't just be added. The next format is 16-bit high low format. Basically 16 bits integer red, 16-bit integer green making up a 32-bit slot. This gives you slightly more range but the problem is that you can't render to it. Because you can't render to it, you can't blend to it. However, they're great for textures and, in effect, they're actually pretty good for storing really high precision vectors. For example, you can use two of these texture surfaces to make a 4-D 16-bit integer surface that is filtered and running optimally on a hardware. Now, we'll get to the floating point formats. First one is the half-precision. This has greater range than the previous integer formats but you can't blend to it and you can't filter it. And then the 32-bit version of that has the greatest range of all but, again, you can't blend to it and you can't filter to it. In the future, this is likely to change as it becomes much more widespread and more useful.



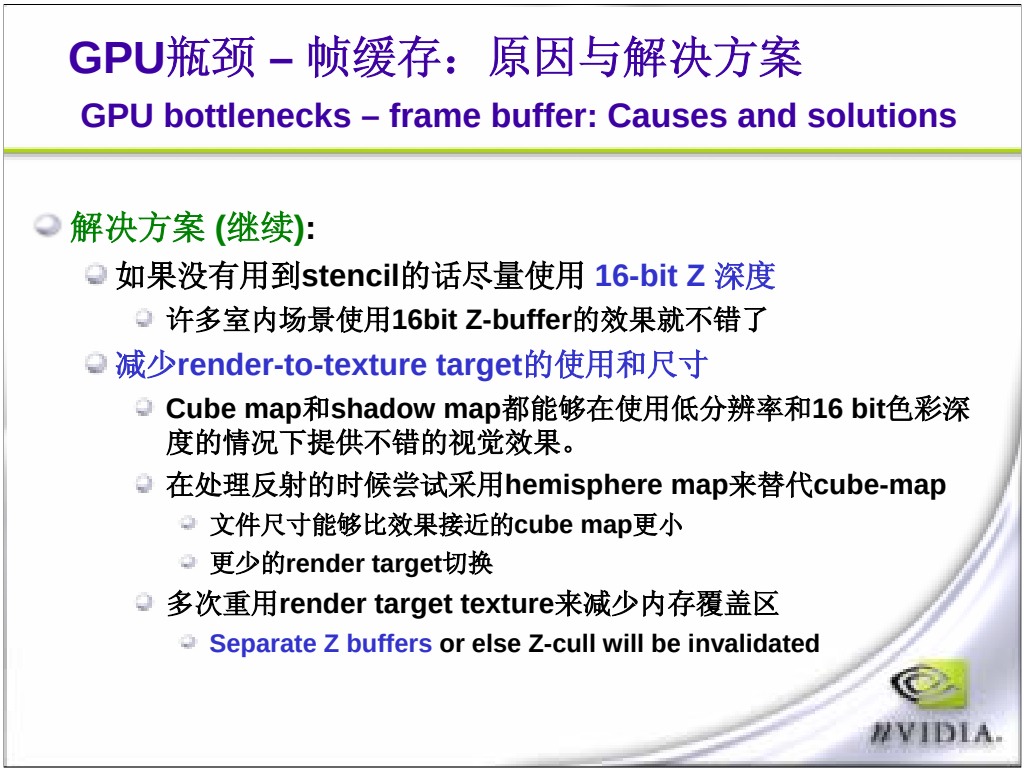
What happens if the texture cache is under-utilized? Well, chances are that you're not using the texture cache well. The easiest way to break the texture cache is to do lots of dependent texture reads and what I mean by dependent texture reads is to generate a texture coordinate in the fragment shader and then go fetch it instead of the interpolated texture coordinate. Let's say you're actually generating texture coordinate from a previous texture read, so, in this case, we've got some sort of random data coming in, doing some math operations and generating brand new texture coordinates. This is going to start fetching texels from areas that are very different in your actual texture map, which is going to cause the cache to be thrashed often. A good way to start avoiding this is to MIP mapping because, basically, calculating the gradient of your dependent texture reads enables us to fetch the texels from a smaller MIP map. But do avoid using negative LOD bias. Negative LOD bias basically sharpens the image but what happens is that it looks sharper for a static image but it actually causes aliasing on the moving image. So, on a screen shot, it might look nicer but, when you actually play the game, you'll see sparklies going around. Instead of doing negative LOD bias, if you do want to sharpen, you should use anisotropic filtering only for those textures that really need it. So what anisotropic filtering does is to sample more in a given direction. We choose the direction by calculating how much texel area is used by a pixel. So if you're using more texels on the X axis than on the Y axis, then we're going to do more filtering on the X axis. And we can have up to an 8:1 ratio of filtering. We can take eight times as many samples in the X direction than in the Y direction. And lastly you should beware of non-power-of-2 textures. They tend to have worse caching behavior than power-of-2 sizes, especially when doing dependent texture reads.

Another texture unit problem could be filtering so tri-linear filtering literally cuts your fill rate in half. Good way to avoid that penalty is to only enable tri-liner filtering on textures that really need it. Light maps, for example, are typically very low resolution and you won't be able to see the MIP line or you might not even hit a lower MIP level until very, very far away, at which point it hardly matters. So there are certain textures that you don't need tri-linear filtering for and, equivalently, there are some textures that you don't need anisotropic filtering for. Pick and choose what type of filtering you do, minimize expensive ones as much as possible.

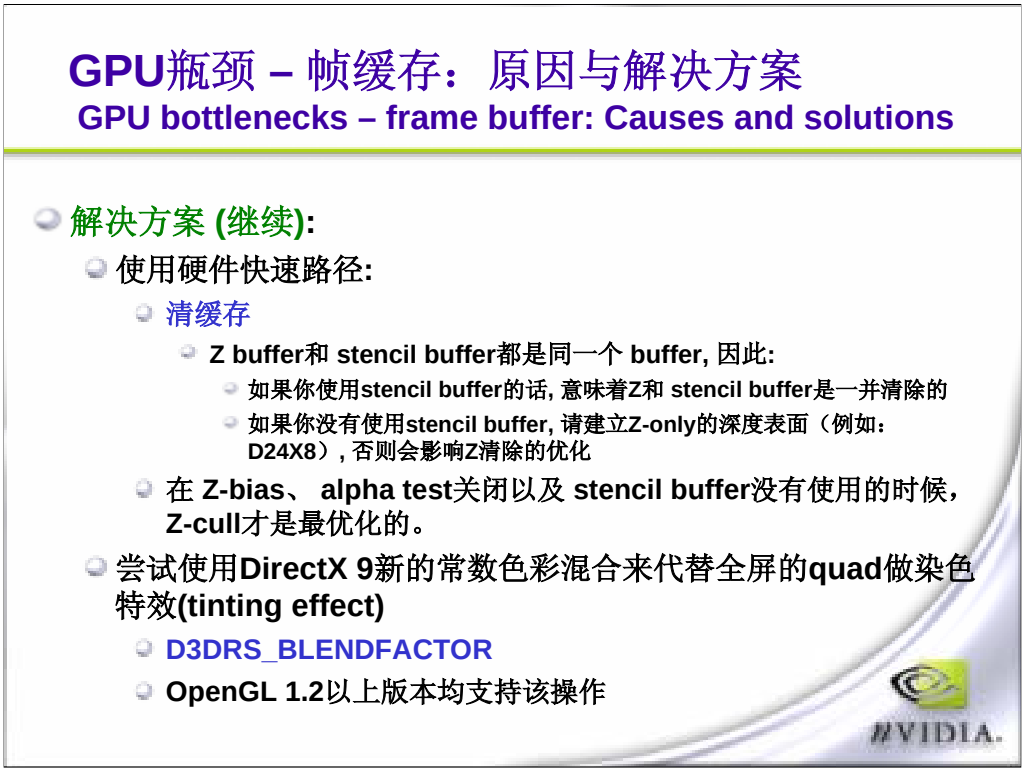
Now, what happens if you're running out of texture space or for some reason you're updating a texture dynamically? Well, what you want to be doing is, again, using the flags that we talked about for vertex buffers. You want to be creating a texture with a dynamic flag whenever you're modifying it and you're going to be locking them with discards but you never, ever, ever want to read a texture. So lock to update it but don't read it. Just discard it. The driver will start up a new area in memory in case the GPU was busy with the previous texture, write to that and then discard the previous memory area once the GPU is done using it.



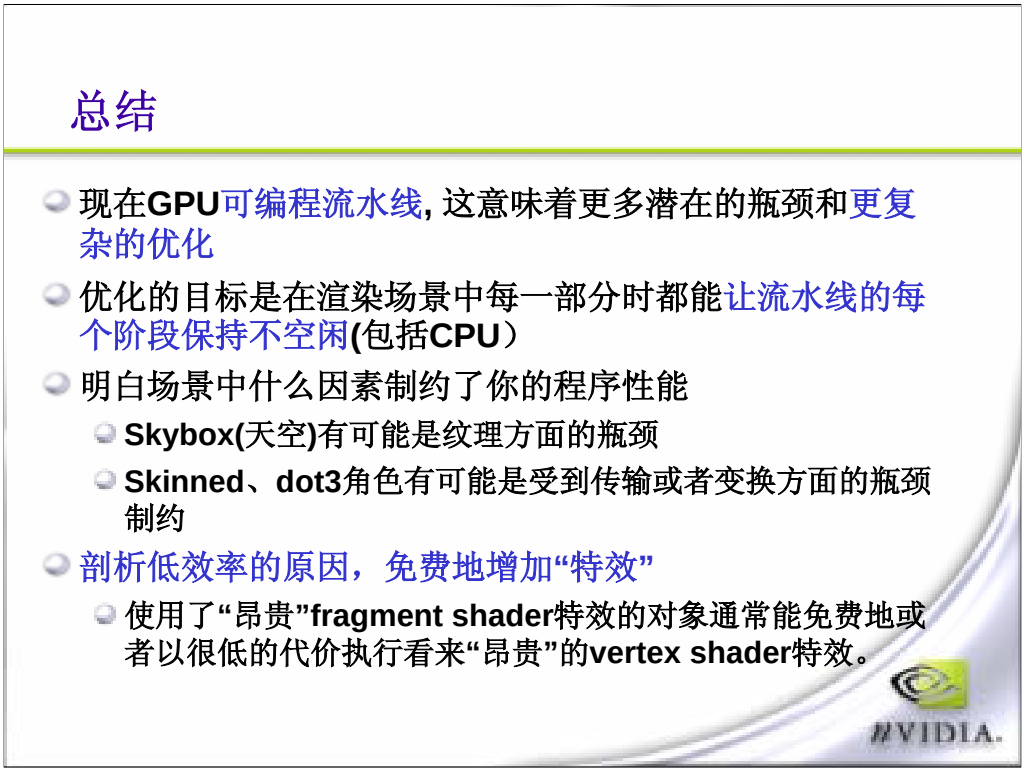
Now we're getting into the final stages of the graphics pipeline and here we're basically talking about reads and writes to the frame buffer. What could be happening is that your read/modify/writing to the Z buffer, which is not always necessary. For example, for multi-pass rendering, after the first pass, you can turn off Z write and instead only do compares. Also, for alpha-blended geometry, you may not need to do a Z write. Now, DX-9 allows you to mask off color channels but if you only mask off some color channels, what happens is that the GPU has to go read the frame buffer, read the masked channels, combine them with the non-masked channels and then write that pixel back again so you either want to turn color writes on or off but don't do partial writes. And, lastly, again on the subject of floating point buffers, you really want to question their use because they use up a tremendous amount of memory bandwidth.



Other tricks that you may be able to get away with is using 16-bit Z buffers. Typically in an interior scene, you don't have a large Z range so this might be good enough for you. Don't trade this with W buffers because our hardware is fully optimized for Z buffers, not for W buffers. Otherwise you're going to lose things like Z cull, for example, and that you definitely don't want to lose. For techniques where you're rendering to a texture, like rendering the environment map for some object, then you want to reduce the resolution of these environment maps or you could even reduce the color depth of these environment maps. Let's say you're mapping this on a sphere, especially on a slightly diffuse or very bumpy sphere, then you won't be able to see the texture detail and the resolution will be useless at that point. So really consider what quality you're looking for, depending on what it's being used for. Now, we talked about spilling over the video memory to AGP memory. Well, sometimes what can happen is that, if you're rendering to off-screen surfaces, you may be consuming so much memory that the driver has to remove textures from video memory. In this case, you want to be re-using those render targets as much as possible; however, keep separate Z buffers because, if you are in the middle of a frame and you render to some off-screen surface without switching the Z surface as well, when you go back to draw the main render target the Z cull buffer will be invalidated and you'll lose all Z cull optimizations for the remainder of the scene.



Lastly, a few general tips to keep in mind. This has been true for all hardware for awhile now. Buffer clears are very fast but do try to keep buffer clears together because the Z surface actually includes the stencil surface, then clearing them together avoids spending time clearing them separately and that's not going to help. We can actually clear Z very, very fast and, if we are able to clear a stencil as well at the same time, then it's practically for free. The Z cull buffer, as well, is particularly fast when you're not using Z bias or alpha testing or the stencil buffer. So try to do as much geometry without using those three things and, lastly, you can actually do color blends for free if, for example, you're rendering the last pass of a multi-pass because you're already doing blending.



We've seen that current generation hardware is quite complicated. We have stepped into the realm of highly programmable architectures which means that it's getting much easier to do bad things and it's also getting harder to identify what you're doing incorrectly. So, hopefully, this presentation has given you a good place to start when optimizing your graphics engine and remember that, because rendering a frame is a pipeline from the CPU to the last stage of the GPU, you want to keep each stage as busy as the slowest stage. Conversely, you want to reduce the load of the slowest stage as much as possible and you have to understand that each part of the scene is going to have a different bottleneck. If you're doing multi-pass rendering then each pass of that multi-pass rendering might have a different bottleneck. And, lastly, just try to get more quality out of the scene if you're not able to reduce the workload.

Questions, comments, feedback?

- **Koji Ashida <kashida@nvidia.com>**
- **http://developer.nvidia.com**

Thank you very much. I hope this was useful and, in particular, that you will be able to make a significant difference to your engine. You can download NVPerfHUD from our developer site.