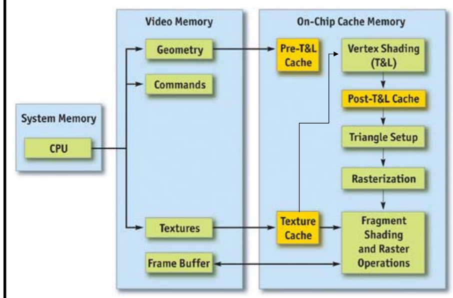
Graphics Pipeline Performance

- http://http.developer.nvidia.com/GPUGems/gpugems_ch28.html

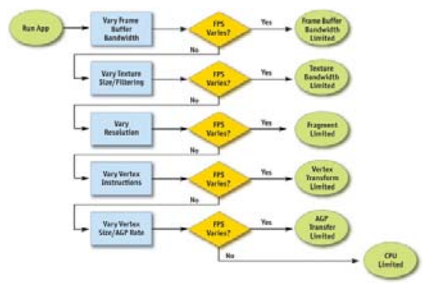
Pipeline



Methodology

1. **Identify the bottleneck.** For each stage in the pipeline, vary either its workload or its computational ability (that is, clock speed). If performance varies, you've found a bottleneck.
2. **Optimize.** Given the bottlenecked stage, reduce its workload until performance stops improving or until you achieve your desired level of performance.
3. **Repeat.** Do steps 1 and 2 again until the desired performance level is reached.

Locating the Bottleneck



Raster Operations

- Reading and writing depth and stencil
- Depth and stencil comparisons
- Reading and writing color
- Alpha blending and testing

- Test by reducing FB bits
  - If reducing your bit depth from 32-bit to 16-bit significantly improves your performance, then you are definitely frame-buffer-bandwidth bound.

Texture Bandwidth

- Texture fetch request goes out to memory
- Texture caches designed to minimize extraneous memory requests
- Texture bandwidth is also a function of GPU memory clock.
- we recommend changing the effective texture size by using a large amount of positive mipmap level-of-detail (LOD) bias. This makes texture fetches access very coarse levels of the mipmap pyramid, which effectively reduces the texture size. If this modification causes performance to improve significantly, you are bound by texture bandwidth.

Fragment Shading

- actual cost of generating a fragment, with associated color and depth values
- fragment shader
- Sometimes combined with Raster Ops and called fill-rate
- Fragment-shading speed is a function of the GPU core clock.

1. change the resolution
2. modify the length of your fragment programs
3. Look at assembly code (optimizer)

Vertex Processing

- taking an input set of vertex attributes (such as model-space positions, vertex normals, texture coordinates, and so on)
- producing a set of attributes suitable for clipping and rasterization (such as homogeneous clip-space position, vertex lighting results, texture coordinates, and more).
- performance in this stage is a function of the work done per vertex, along with the number of vertices being processed.
- Vertex processing speed is a function of the GPU core clock.
- Test: change length of vertex program

Vertex and Index Transfer

- Vertices and indices are fetched by the GPU as the first step in the GPU part of the pipeline.
- They are usually either in system memory—which means they will be transferred to the GPU over a bus such as AGP or PCI Express—or in local frame-buffer memory
- Determining if vertex or index fetching is a bottleneck in your application entails modifying the vertex format size.
- Vertex and index fetching performance is a function of the AGP/PCI Express rate if the data is placed in system memory; it's a function of the memory clock if data is placed in local frame-buffer memory.

Nothing works

- CPU bound?

Reduce Resource Locking

- Anytime you perform a synchronous operation that demands access to a GPU resource, there is the potential to massively stall the GPU pipeline, which costs both CPU and GPU cycles. CPU cycles are wasted because the CPU must sit and spin in a loop, waiting for the (very deep) GPU pipeline to idle and return the requested resource. GPU cycles are then wasted as the pipeline sits idle and has to refill.
- This locking can occur anytime you
  - Lock or read from a surface you were previously rendering to
  - Write to a surface the GPU is reading from, such as a texture or a vertex buffer
- In general, you should avoid accessing a resource the GPU is using during rendering.

Maximize Batch Size

- Minimize the Number of Batches
- **If using triangle strips, use degenerate triangles to stitch together disjoint strips.** This will enable you to send multiple strips, provided that they share material, in a single draw call.
- **Use texture pages.** Batches are frequently broken when different objects use different textures. By arranging many textures into a single 2D texture and setting your texture coordinates appropriately, you can send geometry that uses multiple textures in a single draw call. Note that this technique can have issues with mipmapping and antialiasing. One technique that sidesteps many of these issues is to pack individual 2D textures into each face of a cube map.
- **Use GPU shader branching to increase batch size.** Modern GPUs have flexible vertex- and fragment-processing pipelines that allow for branching inside the shader. For example, if two batches are separate because one requires a four-bone skinning vertex shader and the other requires a two-bone skinning vertex shader, you could instead write a vertex shader that loops over the number of bones required, accumulating blending weights, and then breaks out of the loop when the weights sum to one. This way, the two batches could be combined into one. On architectures that don't support shader branching, similar functionality can be implemented, at the cost of shader cycles, by using a four-bone vertex shader on everything and simply zeroing out the bone weights on vertices that have fewer than four bone influences.
- **Use the vertex shader constant memory as a lookup table of matrices.** Often batches get broken when many small objects share all material properties but differ only in matrix state (for example, a forest of similar trees, or a particle system). In these cases, you can load *n* of the differing matrices into the vertex shader constant memory and store indices into the constant memory in the vertex format for each object. Then you would use this index to look up into the constant memory in the vertex shader and use the correct transformation matrix, thus rendering *n* objects at once.
- **Defer decisions as far down in the pipeline as possible.** It's faster to use the alpha channel of your texture as a gloss factor, rather than break the batch to set a pixel shader constant for glossiness. Similarly, putting shading data in your textures and vertices can allow for larger batch submissions.

Reducing the Cost of Vertex Transfer

- **Use the fewest possible bytes in your vertex format.** Don't use floats for everything if bytes would suffice (for colors, for example).
- **Generate potentially derivable vertex attributes inside the vertex program instead of storing them inside the input vertex format.** For example, there's often no need to store a tangent, binormal, and normal: given any two, the third can be derived using a simple cross product in the vertex program. This technique trades vertex-processing speed for vertex transfer rate.
- **Use 16-bit indices instead of 32-bit indices.** 16-bit indices are cheaper to fetch, are cheaper to move around, and take less memory.
- **Access vertex data in a relatively sequential manner.** Modern GPUs cache memory accesses when fetching vertices. As in any memory hierarchy, spatial locality of reference helps maximize hits in the cache, thus reducing bandwidth requirements.

Optimizing Vertex Processing

- **Optimize for the post-T&L vertex cache.** Modern GPUs have a small first-in, first-out (FIFO) cache that stores the result of the most recently transformed vertices; a hit in this cache saves all transform and lighting work, along with all work done earlier in the pipeline. To take advantage of this cache, you must use indexed primitives, and you must order your vertices to maximize locality of reference over the mesh. There are tools available—including D3DX and NVTriStrip (NVIDIA 2003)—that can help you with this task.
- **Reduce the number of vertices processed.** This is rarely the fundamental issue, but using a simple level-of-detail scheme, such as a set of static LODs, certainly helps reduce vertex-processing load.
- **Use vertex-processing LOD.** Along with using LODs for the number of vertices processed, try LODing the vertex computations themselves. For example, it is likely unnecessary to do full four-bone skinning on distant characters, and you can probably get away with cheaper approximations for the lighting. If your material is multipassed, reducing the number of passes for lower LODs in the distance will also reduce vertex-processing cost.
- **Pull out per-object computations onto the CPU.** Often, a calculation that changes once per object or per frame is done in the vertex shader for convenience. For example, transforming a directional light vector to eye space is sometimes done in the vertex shader, although the result of the computation changes only once per frame.
- **Use the correct coordinate space.** Frequently, choice of coordinate space affects the number of instructions required to compute a value in the vertex program. For example, when doing vertex lighting, if your vertex normals are stored in object space and the light vector is stored in eye space, then you will have to transform one of the two vectors in the vertex shader. If the light vector was instead transformed into object space once per object on the CPU, no per-vertex transformation would be necessary, saving GPU vertex instructions.
- **Use vertex branching to "early-out" of computations.** If you are looping over a number of lights in the vertex shader and doing normal, low-dynamic-range, [0..1] lighting, you can check for saturation to 1—or if you're facing away from the light—and then break out of further computations. A similar optimization can occur with skinning, where you can break when your weights sum to 1 (and therefore all subsequent weights would be 0). Note that this depends on how the GPU implements vertex branching, and it isn't guaranteed to improve performance on all architectures.

Speeding Up Fragment Shading

- **Render depth first.** Rendering a depth-only (no-color) pass before rendering your primary shading passes can dramatically boost performance, especially in scenes with high depth complexity, by reducing the amount of fragment shading and frame-buffer memory access that needs to be performed. To get the full benefits of a depth-only pass, it's not sufficient to just disable color writes to the frame buffer; you should also disable all shading on fragments, even shading that affects depth as well as color (such as alpha test).
- **Help early-z optimizations throw away fragment processing.** Modern GPUs have silicon designed to avoid shading occluded fragments, but these optimizations rely on knowledge of the scene up to the current point; they can be improved dramatically by rendering in a roughly front-to-back order. Also, laying down depth first (see the previous tip) in a separate pass can help substantially speed up subsequent passes (where all the expensive shading is done) by effectively reducing their shaded-depth complexity to 1.
- **Store complex functions in textures.** Textures can be enormously useful as lookup tables, and their results are filtered for free. The canonical example here is a normalization cube map, which allows you to normalize an arbitrary vector at high precision for the cost of a single texture lookup.
- **Move per-fragment work to the vertex shader.** Just as per-object work in the vertex shader should be moved to the CPU instead, per-vertex computations (along with computations that can be correctly linearly interpolated in screen space) should be moved to the vertex shader. Common examples include computing vectors and transforming vectors between coordinate systems.
- **Use the lowest precision necessary.** APIs such as DirectX 9 allow you to specify precision hints in fragment shader code for quantities or calculations that can work with reduced precision. Many GPUs can take advantage of these hints to reduce internal precision and improve performance.
- **Avoid excessive normalization.** A common mistake is to get "normalization-happy": normalizing every single vector every step of the way when performing a calculation. Recognize which transformations preserve length (such as transformations by an orthonormal basis) and which computations do not depend on vector length (such as cube-map lookups).
- **Consider using fragment shader level of detail.** Although it offers less bang for the buck than vertex LOD (simply because objects in the distance naturally LOD themselves with respect to pixel processing, due to perspective), reducing the complexity of the shaders in the distance, and decreasing the number of passes over a surface, can lessen the fragment-processing workload.
- **Disable trilinear filtering where unnecessary.** Trilinear filtering, even when not consuming extra texture bandwidth, costs extra cycles to compute in the fragment shader on most modern GPU architectures. On textures where mip-level transitions are not readily discernible, turn trilinear filtering off to save fill rate.
- **Use the simplest shader type possible.** In both Direct3D and OpenGL, there are a number of different ways to shade fragments. For example, in Direct3D 9, you can specify fragment shading using, in order of increasing complexity and power, texture-stage states, pixel shaders version 1.x (ps.1.1 – ps.1.4), pixel shaders version 2.x., or pixel shaders version 3.0. In general, you should use the simplest shader type that allows you to create the intended effect. The simpler shader types offer a number of implicit assumptions that often allow them to be compiled to faster native pixel-processing code by the GPU driver. A nice side effect is that these shaders would then work on a broader range of hardware.

Reducing Texture Bandwidth

- **Reduce the size of your textures.** Consider your target resolution and texture coordinates. Do your users ever get to see your highest mip level? If not, consider scaling back the size of your textures. This can be especially helpful if overloaded frame-buffer memory has forced texturing to occur from nonlocal memory (such as system memory, over the AGP or PCI Express bus). The NVPerfHUD tool (NVIDIA 2003) can help diagnose this problem, as it shows the amount of memory allocated by the driver in various heaps.
- **Compress all color textures.** All textures that are used just as decals or detail textures should be compressed, using DXT1, DXT3, or DXT5, depending on the specific texture's alpha needs. This step will reduce memory usage, reduce texture bandwidth requirements, and improve texture cache efficiency.
- **Avoid expensive texture formats if not necessary.** Large texture formats, such as 64-bit or 128-bit floating-point formats, obviously cost much more bandwidth to fetch from. Use these only as necessary.
- **Always use mipmapping on any surface that may be minified.** In addition to improving quality by reducing texture aliasing, mipmapping improves texture cache utilization by localizing texture-memory access patterns for minified textures. If you find that mipmapping on certain surfaces makes them look blurry, avoid the temptation to disable mipmapping or add a large negative LOD bias. Prefer anisotropic filtering instead and adjust the level of anisotropy per batch as appropriate.

Optimizing Frame-Buffer Bandwidth

- **Render depth first.** This step reduces not only fragment-shading cost (see the previous section), but also frame-buffer bandwidth cost.
- **Reduce alpha blending.** Note that alpha blending, with a destination-blending factor set to anything other than 0, requires both a read and a write to the frame buffer, thus potentially consuming double the bandwidth. Reserve alpha blending for only those situations that require it, and be wary of high levels of alpha-blended depth complexity.
- **Turn off depth writes when possible.** Writing depth is an additional consumer of bandwidth, and it should be disabled in multipass rendering (where the final depth is already in the depth buffer); when rendering alpha-blended effects, such as particles; and when rendering objects into shadow maps (in fact, for rendering into color-based shadow maps, you can turn off depth reads as well).
- **Avoid extraneous color-buffer clears.** If every pixel is guaranteed to be overwritten in the frame buffer by your application, then avoid clearing color, because it costs precious bandwidth. Note, however, that you should clear the depth and stencil buffers whenever you can, because many early-z optimizations rely on the deterministic contents of a cleared depth buffer.
- **Render roughly front to back.** In addition to the fragment-shading advantages mentioned in the previous section, there are similar benefits for frame-buffer bandwidth. Early-z hardware optimizations can discard extraneous frame-buffer reads and writes. In fact, even older hardware, which lacks these optimizations, will benefit from this step, because more fragments will fail the depth test, resulting in fewer color and depth writes to the frame buffer.
- **Optimize skybox rendering.** Skyboxes are often frame-buffer-bandwidth bound, but you must decide how to optimize them: (1) render them last, reading (but *not* writing) depth, and allow the early-z optimizations along with regular depth buffering to save bandwidth; or (2) render the skybox first, and disable all depth reads and writes. Which option will save you more bandwidth is a function of the target hardware and how much of the skybox is visible in the final frame. If a large portion of the skybox is obscured, the first technique will likely be better; otherwise, the second one may save more bandwidth.
- **Use floating-point frame buffers only when necessary.** These formats obviously consume much more bandwidth than smaller, integer formats. The same applies for multiple render targets.
- **Use a 16-bit depth buffer when possible.** Depth transactions are a huge consumer of bandwidth, so using 16-bit instead of 32-bit can be a giant win, and 16-bit is often enough for small-scale, indoor scenes that don't require stencil. A 16-bit depth buffer is also often enough for render-to-texture effects that require depth, such as dynamic cube maps.
- **Use 16-bit color when possible.** This advice is especially applicable to render-to-texture effects, because many of these, such as dynamic cube maps and projected-color shadow maps, work just fine in 16-bit color.