ATI Technologies Inc.

Performance Optimization Techniques for ATI Graphics Hardware with DirectX® 9.0

Revision: 1.0

Original Date: Dec. 2002

Author: Guennadi Riguer

Acknowledgments

The author would like to thank following individuals for their valuable feedback and great support:

Eric Demers,
Tim Kelley,
Raja Koduri,
Jason Mitchell,
Andy Pomianowski,
Phil Rogers,
Jeff Royle and
Michael Smith.

Contact Information

ATI Technologies Inc.	Developer Relations Hotline .... (905) 882-2636
1 Commerce Valley Drive East	Developer Relations Fax …............. (905) 882-9339
Mailstop: DR-2E	Developer Relations E-mail .... devrel@ati.com
Markham, Ontario	
Canada L3T 7X6	

http://www.ati.com/developer

Contents

1. Introduction

Over the last couple of years the computer graphics have taken a quantum leap in performance, exceeding everybody's expectations and leaving Moore's law far behind. Not that long ago a typical graphics card had just a few megabytes of video memory, and could only render 50-100 thousand Gouraud shaded single textured triangles per second, something almost laughable these days. The consumer graphics hardware of today is capable of storing and crunching immense amounts of data. It is not uncommon to see video cards with 128Mb of memory or even more, that can render hundreds of millions polygons per second. Newest consumer VPU's (Visual Processing Unit) pack multiple graphics engines on a single chip operating at the floating point precision while carrying out complex computations per vertex as well as per pixel. And it is not just the raw performance that has taken everybody by surprise, but also a visual quality of modern real-time graphics that rivals quality of the off-line rendering seen in movies and cartoons, such as "Toy Story" and many others.

These great performance advancements of modern graphics hardware prompt valid questions about importance of optimizations. If the hardware is so much faster is there a need for optimizations? Is it not enough to rely on brute force of the hardware to crunch just about anything? In reality the answers are not that obvious. It is true that the performance has increased, but so have our expectations of photo-realism and the image quality in general. Furthermore, due to its high complexity and various optimization tricks applied by hardware designers, the graphics hardware now is much more sensitive to API and hardware misuses than it used to be. To stay competitive these days it is just as important to squeeze every little bit of power out of the latest graphics cards as it was yesterday.

The purpose of this paper is to introduce and explain complexities of performance tuning and optimizations, while providing real world solutions for state of the art ATI hardware. Primarily this paper is targeted at the latest ATI's offerings – RADEON™ 9500/9700 family, as well as previous generation parts, based on RADEON™ 8500/9000 family. However, some of the optimizations explained here apply to many other graphic cards, including those made by other hardware vendors. In the text you will see appropriate symbols, whenever the discussed optimizations techniques apply only to specific generation of graphics card.

2. Optimization Strategies

Performance optimization of graphics applications is a very tricky business. Modern computers capable of generating 3D graphics are very complex systems consisting of many components with very intricate performance characteristics. There is a CPU, a VPU and various types of memory – system memory, AGP and local video memory. It takes a good understanding of individual component behavior as well as good intuition to make sense of what is happening inside the computer when it renders 3D graphics.

An interesting VPU characteristic is that it has very deep pipeline that can be further broken down into vertex processor, pixel processor and render back-end components. Whenever necessary, this paper will provide some specifics about ATI graphics hardware architecture to help you understand operation of VPU blocks.

The optimal overall system performance can be achieved only then when there are no stalls in the data flow, all components are working in parallel and all of them are operating at the top utilization. Introducing stalls or breaking parallelism of the components leads to lowered utilization of separate parts and the system as a whole.

The overall system performance is only as good as the performance of its worst bottleneck, so the optimization task can be reduced to finding the worst bottleneck and effectively dealing with it. This is an iterative process that should be repeated until the performance is within acceptable limit, or as they often write on shampoo bottles "apply, rinse, and repeat if necessary…"

Types of bottlenecks and their detection

There are many factors that contribute to the graphics system performance, however at the very coarse level performance can be defined to be limited by non-graphics components or graphics components. Non-graphics component bottlenecks can be caused by CPU load or memory bandwidth. You can try identifying this kind of performance bottleneck by varying performance of the graphics system. If performance does not change with varying graphics card performance, but is affected by CPU speed, you are most likely dealing with non-graphics bottleneck. On the other hand if CPU makes little difference, but changing graphics card performance and resolution affect overall efficiency, you have got a graphics system bottleneck.

CPU processing bottleneck: This is by far the most common bottleneck. It is caused by many operations that might have nothing to do with 3D graphics. In games most of the CPU time can be consumed by computing physics, AI, graphics setup and many other things. A good way to detect this performance problem is to check what consumes most of the CPU time with Intel® VTune™ or similar tool. If very little time is spent in the driver, the chances are you have a CPU bottleneck. On the other hand, if a lot of time is spent in the driver, CPU can still be the bottleneck and the most likely sub-optimal API usage is a cause of such poor performance.

Memory bandwidth bottleneck: Very "heavy" memory copies in your application can cause this bottleneck. If the application uses a lot of dynamic vertex data and vertex throughput is quite low, you might need to take measures to solve this problem. Try limiting the amount of dynamic data streamed to the graphics card to check if this is really a bottleneck. Also check if changing vertex size affects the performance.

Graphics component bottleneck, as the name implies is caused by one of the many components of the graphics subsystem. There are many reasons for graphics hardware to become a performance-limiting factor.

Vertex processing bottleneck: This is very rarely a bottleneck considering huge vertex processing power of the latest graphics hardware. It can be detected by simplifying per-vertex calculations through simplification of shaders, reduction of number of lights and so on. Keep in mind that most likely vertex throughput is constraint by memory throughput limitation. Make sure to profile for vertex throughput at low resolutions to eliminate effect of the fillrate.

Fillrate bottleneck: Fillrate commonly refers to number of rendered pixels per second. It is affected by pixel shader execution, texture fetches and rasterization in render back-end. If changing resolution changes performance, check out what might cause fillrate limitation.

Pixel shader bottleneck: This bottleneck might show up in applications that use complex pixel shaders, especially in combination with high overdraw. Try replacing big pixel shaders with smaller and simpler ones and watch for performance changes. Bear in mind that pixel shader execution can be limited by both texture fetch and arithmetic instruction execution.

Texture fetch bottleneck: It can be caused by video memory bandwidth limitation and/or texture filtering performance. Try replacing your textures with smaller ones, like 4x4 texels and switch to simpler filtering modes, or even disable texturing altogether. If you see a drastic improvement in application performance there is a good chance that the application has a texture fetch bottleneck.

Rasterization bottleneck: Rasterization can limit graphics performance mostly because of insufficient memory bandwidth required by back-end for alpha blending, depth buffering, multisampling and etc. Try eliminating or reducing alpha blending and check depth complexity to understand what might cause this bottleneck.

NOTE: Keep in mind that in most real life situations you will rarely see just one type of a bottleneck at a time. In most cases you will have to deal with a combinations of various bottlenecks, which complicates their detection and resolution. For instance video memory bandwidth is shared between different blocks of the graphics chip, and it is too easy to shift bottleneck from one part to another without really resolving it. Do not expect the recommendations from this paper to be a "panacea" and take most of them with a gain of salt. Always perform exhaustive testing to understand what is going on, what affects performance the most; and of course use your best judgment.

3. Optimizing Resource Usage and Management

Textures, surfaces, vertex and index buffers can reside in various types of memory – system memory, AGP memory or local video memory. The performance of the graphics system depends on access time to these resources and in most cases it is the highest when these resources are kept closest to the device that accesses them, and no stalls are incurred. For VPU accesses it is better to store resources in local video memory, while for CPU accesses – in system memory. Improperly created or misused resources can cause severe performance penalties.

3.1. Resource Creation

The growth rate of the amount of video memory is astonishing and it is quite common these days to see graphics cards with 128Mb of memory or even more. At the same time the more complex environments of today's games with complex geometry and greater amounts of highly detailed textures demand more memory. Using FSAA can even further complicate the situation. Running at 1280x1024 with 4x multisampling, a double-buffered swap chain consumes about 45Mb of video memory. Add some render targets on top of that and the next thing you know, there is not enough room to put everything in video memory.

The drivers employ some heuristics and try to allocate resources where they would yield the highest performance. Most of the information about intended resource behavior comes from creation flags specified through Direct3D® API. While driver will try to place resource in the best possible memory pool, it is possible for the allocation to fail due to the lack of free memory. In that case driver or runtime might either try to free up some space by moving other resources around or place resource in sub-optimal memory location. For example textures might spill into AGP memory if local video memory is full. Because of that it is important to maintain certain optimal order in which resources are created and loaded. In general, render targets should be created first, followed by the static resources in the **D3DPOOL_DEFAULT** memory pool, and at last by the managed resources. When resetting device all managed resources should be evicted with **EvictManagedResources** call before recreating other resources.

Another trick that can be used is to make sure the most important resources end up at the best memory location. When creating resources, do so in the order of importance. This importance is rather subjective and might be hard to determine. Most likely the most important textures would be the ones that get accessed a lot and are used to texture large areas on the screen, like a sky or a ground texture.

Managed resources are usually lazily updated in the AGP or local video memory. That is a system memory copy might be uploaded to the graphics card only when resource is used to render the scene. This might cause some unexpected performance problems, such as stuttering due to memory uploads, even if there is enough video memory available. To combat this problem, make sure that all the managed resources that will be used are paged into the video memory by "touching" them. Just render a single triangle textured with all of the textures on the very first frame that never gets presented. The same procedure applies to managed vertex and index buffers.

3.2. Using Vertex and Index Buffers

Creating efficient vertex and index buffers

Dynamic vertex and index buffers always have to be created in **D3DPOOL_DEFAULT** memory pool. If they are allocated in **D3DPOOL_MANAGED** memory pool, two copies of the buffers will be maintained – one in the system memory and another in **D3DPOOL_DEFAULT** pool. Whenever a managed buffer is locked and written to, the application gets a pointer to system memory copy, which will be copied to **D3DPOOL_DEFAULT** pool buffer instance once the buffer is unlocked. This results in a redundant memory copy operation that consumes CPU time and wastes memory bandwidth.

In general it is a bad idea to read back from the buffers allocated in **D3DPOOL_DEFAULT** memory pool, because reads from AGP and especially video memory are slow. To indicate that application will not read back from the buffer use **D3DUSAGE_WRITEONLY** flag. This will ensure the optimal buffer location and locking behavior is used. For the best performance the **D3DUSAGE_WRITEONLY** flag should be specified for **D3DPOOL_MANAGED** resources as well. If you need to read from a buffer, keep your own copy in **D3DPOOL_SYSTEM** pool and do not rely on managed resources.

Optimal vertex and index buffer size

It is important to allocate vertex buffers of reasonable sizes. If buffers are too small and there too many of them, the system and video memory is wasted because of DirectX® and driver overhead. Also extra DirectX® and driver overhead is incurred due to buffer switching. On the other hand, using buffers that are too big can limit amount of video memory available to other resources and can affect managed resource swapping. On resource swapping big buffers can cause bad memory fragmentation and waste a lot of memory.

For static vertex buffers 1-4Mb is a good size to start with, but it might vary depending on the amount of your resources, local video and AGP memory available. If amount of available video memory is low and a lot of resource swapping is expected, a smaller buffer size is a better choice. For dynamic buffers you should not allocate buffers bigger than the data streamed to the card per frame. In most cases 256Kb-1Mb dynamic buffers provide a good starting point for performance tweaking. In DirectX® 9 there is a way to specify byte offset in the vertex buffers when setting up vertex streams. This per-stream offset solves the problem of using multiple dynamic streams without wasting extra memory, which was a case in previous versions of DirectX®. Use this feature to pack multiple vertex formats into the same buffer and reduce the total number of buffers and their switches.

Similar rationale can be applied to index buffers – aim at reducing possible memory fragmentation and waste of memory.

When to use index buffers?

The answer to this question is – "Always". On ATI's hardware starting with RADEON™ 8500, the index buffers are "first class citizens". This means that they can reside in AGP or local video memory and can be natively fetched by VPU without any CPU involvement. Because of this, never force an index buffer into **D3DPOOL_SYSTEM** memory pool. Let the driver decide what the best memory location is.

Also see discussion on using **DrawPrimitive** vs. **DrawIndexedPrimitive**.

Use of static vertex and index buffers

Most of the applications combine the use of both static and dynamic geometry. For our purpose the dynamic data is the one that is constantly uploaded to the video memory. The dynamic character in a game can thus be viewed as static data if it is animated on the VPU. In a typical game a terrain, buildings and a sky most likely will be static, while characters, shadow volumes and such will be dynamic. When it comes to using static and dynamic vertex and index buffer, the rule of thumb is to place static geometry in the static buffers while dynamic geometry – in dynamic buffers. The infrequently updated data, which is updated once per frame or even less frequently, can be treated as static data.

While it is OK to use dynamic geometry whenever necessary, do not abuse it too much. Do not repeatedly stream down to the card geometry that never or rarely changes. Much higher polygon throughput can be achieved with static geometry than with dynamic geometry. If application is CPU or memory bandwidth bound, you might want to start thinking about balancing the workload between CPU and VPU. Turning your dynamic data into static, and employing vertex shaders to move animation from CPU to VPU, can accomplish such workload balancing.

Vertex and index buffer updates

When locking vertex and index buffers for update make sure to indicate your intended behavior with proper lock flags. Use **D3DLOCK_DISCARD** and **D3DLOCK_NOOVERWRITE** flags when locking dynamic data to prevent stalling in the driver until buffer is processed by VPU. The **D3DLOCK_NOOVERWRITE** flag guarantees the application will not corrupt existing data in the buffer and provides very lightweight lock. The **D3DLOCK_DISCARD** flag indicates that the buffer can be disposed of, and might return a pointer to a new buffer instance if current one is not processed by VPU. From the API perspective it looks like the same buffer is locked and returned because of the internal "buffer renaming" in the driver. There might be a limit on how many instances of the buffer can be created internally in the driver, but in most cases it will only be limited by amount of free video memory. Because locking with **D3DLOCK_DISCARD** flag might cause internal buffer creation in the driver it is more expensive than lock with **D3DLOCK_NOOVERWRITE** flag. The **D3DLOCK_DISCARD** flag should be used on the first buffer lock of the frame or when there is not enough room left in the buffer for new data. If there is enough room for placing new data, use **D3DLOCK_NOOVERWRITE** flag. Remember that **D3DLOCK_DISCARD** and **D3DLOCK_NOOVERWRITE** flags are mutually exclusive and should not be used together. These flags also should not be used with static buffers.

For static buffer updates the situation is trickier. First of all, as the name implies, there should be no updates if the data is static. There are situations however, when data is "almost" static and changes very infrequently. Using dynamic buffers and streaming data every frame might not be very efficient. In that case try updating data as much in advance as possible to prevent stalls on the buffers currently used for rendering. Another option might be to maintain some extra buffers and update them in a round-robin fashion.

3.3. Vertex Data

When application is vertex throughput bound, some optimizations might be required to optimally use memory bandwidth. When dealing with dynamic data, the CPU memory writes to AGP or local video memory have to be considered in addition to the video memory fetches by VPU. For static data, only video memory traffic is applicable. Depending on your situation you might need to optimize for both types of memory accesses, but most likely dynamic data updates will be a bigger problem due to lower memory bandwidth and peculiarities of AGP transfers. That is why it is very important to make dynamic data updates AGP-friendly.

Number of vertex streams and performance

The best vertex processing performance can be achieved by utilizing as few vertex streams as possible. It is unfair to say that using multiple vertex streams is bad, but whenever you have an option of reducing the number of streams without sacrificing functionality or performance, you should do that. This is especially true for dynamic data. There are a couple of reasons why using multiple streams might result in lower performance. For dynamically streamed data the biggest and the most important pitfall might be un-optimized AGP writes, since most likely dynamic data will end up in the AGP memory. The discussion on making memory writes "AGP-friendly" can be found below.

Another reason for poorer performance can be attributed to reduced on-chip cache efficiency. Most of the TnL graphic parts have on the chip two vertex caches – pre-TnL cache and post-TnL cache. The former is just a memory cache that serves the purpose of reducing the latency of memory reads. By using multiple vertex streams you can end up thrashing the pre-TnL cache more, but it really depends on the access patterns, which are hard to predict. This most likely will not be a huge problem, but it is just something to be aware of.

The ideal use of the multiple vertex streams is to keep your dynamic data separated from static data. For instance if you have otherwise static geometry with only texture coordinates that change, you can separate dynamic texture coordinates into a different vertex stream and update only them. This will reduce the amount of data that needs to be streamed to the card.

Making data updates AGP-friendly

To improve the performance of AGP writes CPUs use write combining. All memory accesses are combined into a cache line transfers and transferred to memory in single bursts. The internal cache that combines multiple memory requests is called Write Combining (WC) buffer and is 32 bytes for Pentium III. Both Pentium IV and Athlon CPUs have 64 byte WC buffers. Memory accesses that fall outside of the write combining area will close the WC buffer and will result in writes of WC buffer contents to the memory. Flushing partially written WC buffers results in a series of partial write transactions, which reduces effective memory bandwidth of the bus by a factor of eight.

Most likely the dynamic data will be placed in AGP memory, so it is very important to write data in AGP-friendly manner to maximize available AGP bandwidth. The most important strategy is avoiding partial transactions. When only a part of the vertex needs to be updated in the vertex buffer it is better to rewrite the whole vertex to avoid partial writes. When using multiple vertex streams do not update them at the same time, but rather compute and write them separately. Also keep all vertex data cache line aligned.

Vertex and index sizes

Picking the optimal vertex size can minimize the memory bandwidth and latency of vertex fetches. The optimal vertex size depends on the vertex access patterns. For vertices that are accessed in the order close to sequential it is better to keep vertex size as small as possible. Because of the pre-TnL cache, there is a good chance the next vertex or at least a part of it will be pre-fetched. However, if vertices are accessed in more of a random order, the odds are the wrong vertices will be pre-fetched. This might cause worse cache thrashing especially as vertices start unnecessary spanning multiple cache lines. To reduce this cache thrashing it is recommended to make randomly accessed vertices to be cache line aligned. For randomly accessed data, aligning vertex size to 32 bytes works well for most graphic parts.

4. Render State Management

Render state updates can produce a fairly high overhead in DirectX® runtime and the driver. The most expensive are texture and shader changes, followed by vertex and index buffer changes, texture state changes and render state changes. Normally, runtime filters the render state changes and reduces amount of redundant state changes sent to the driver, but it still can be quite expensive. This does not happen for **PURE** devices though. The easiest way of dealing with redundant state changes is to introduce state management in the application. In the simplest case the application can use a simple state cache and defer state updates until primitive rendering, by submitting only state deltas just before **DrawPrimitive** of **DrawIndexedPrimitive** calls. Quite often the application can do much better by employing more intelligent caching schemes, because it knows better than anything else what and how is rendered.

Sorting of rendered objects

To reduce the number of state changes and associated overhead, applications can sort and batch rendered objects by shaders or rather "effects" that include combination of shaders, textures and other states. On the other hand it might be beneficial to sort objects by distance for HYPER Z™ optimizations. Sorting objects by effect/state quite often conflicts with sorting by distance used in front to back rendering. One possible solution to this problem is described in discussion on When Multiple Render Passes Are Better than One. Also, see sections on Shader and Shader Constant management.

5. Rendering Primitives

Nowadays graphics hardware has very complex pipelines consisting of many processing stages. To hide the latency between different stages, a number of caches are employed throughout the pipeline. On the vertex-processing path of the TnL capable chips there are usually two caches – pre-TnL and post-TnL. The former has already been covered in discussion on multiple vertex streams. Now is the time to talk about the latter. The post-TnL cache, as implied by the name is located after vertex processor, and stores result of the vertex transformations. This cache is usually smaller than pre-TnL cache and works with both fixed function and programmable implementation of vertex processor. There are a couple of rules that have to be followed when rendering primitives to take advantage of post-TnL cache.

DrawPrimitive vs. DrawIndexedPrimitive

In cases where there is absolutely no vertex reuse – which are very rare – **DrawPrimitive** will work just as well as **DrawIndexedPrimitive**. In all other cases **DrawIndexedPrimitive** calls are better. To take advantage of the post-TnL cache and skip processing of the vertices that were just recently transformed you have to use indexed primitives. Another good reason to use **DrawIndexedPrimitive** is reduced memory bandwidth. The number of vertices that have to be updated and fetched is less when you render triangle lists with triangles that share vertices; and because indices are much smaller than vertices, even with their extra overhead you win.

Mesh optimizations

To make vertex caches efficient and thus reduce memory bandwidth as well as computational load it is important to optimize meshes for locality of reference. This means that vertices of the mesh have to be ordered for quickest reuse, while random accesses to vertices have to be decreased if not eliminated completely. The performance improvements from this seemingly minimal optimization are huge. It is not uncommon to see the vertex throughput rates increase two times or even more.

The easiest way to optimize mesh in DirectX® 9 is to use **ID3DXMesh::Optimize** function with **D3DXMESHOPT_VERTEXCACHE** flag. Runtime has knowledge of effective post-TnL cache sizes for different chips and can very efficiently optimize mesh for specific hardware.

In DirectX® 9 **D3DQUERYTYPE_VCACHE** type query can be used to find out specifics about hardware vertex cache implementations, if the display drivers support this type of query.

Using triangle strips can provide simple yet pretty good mesh optimization, since strips guarantee that at least two vertices per triangle are reused. To minimize the driver overhead of rendering small triangle lists it is beneficial to concatenate short lists into longer ones with degenerate triangles. If number of degenerate triangles is relatively high, it might be better to use strip ordered triangle lists. All TnL-capable ATI hardware has very efficient hardware implementation of triangle lists; so on any RADEON™ family hardware strip ordered triangle lists are just as good as triangle strips. Considering that triangle lists are much more flexible than triangle strips, in many cases they might be a preferred choice.

Rendering particles

Many applications make extensive use of the particle effects – explosions, fires, atmospheric effects such as fog, rain and etc. In many cases for simple particle rendering it makes sense to use point sprites, the lightweight primitives defined only by one vertex. These primitives minimize the vertex throughput, since they are defined by only a single vertex per sprite, at the expense of flexibility.

Whether rendering heaps of particles with point sprites or any other primitives, several optimization techniques can apply. If particles are not alpha blended and cause huge overdraw, have them sorted for distance and render front to back to take advantage of HYPER Z™ technique as described in section on Depth Buffer optimizations.

When rendering small size particles, especially if they are alpha blended, a good optimization might be to sort particles for locality. Most modern graphic chips have destination caches for color and depth buffers. Sorting for locality might help performance as it reduces destination cache thrashing.

6. Optimizing Shaders

Modern graphics chips offer enormous vertex and pixel processing power; nevertheless there are times when even that power is not enough. When running long and complex shaders it is possible to exhaust all that power and make shader processing a bottleneck. Another "opportunity" to limit performance hides in inefficient shader management. This section will deal with both of these obstacles.

Behind the scenes shader processing

The vertex and pixel shaders in DirectX® 9 are defined as streams of tokens, each token representing an op-code of assembly instruction or macro. This is how they are passed by the application to the shader creation functions of the API. This is also how the driver receives them. None of the macros are expanded by the runtime, and it is up to the driver how to deal with them. If hardware natively supports a macro, it will be executed as is, otherwise it will be expanded into a series of simpler instructions.

A common misconception is that hardware shader implementation exactly matches the shader assembly or op-codes as defined by DirectX®. The direct mapping of the shader code to the hardware might not result in the most efficient shader implementation, and hardware uses many tricks to provide the best performance possible. You should think of the DirectX® shaders as p-code (pseudo code) programs that are passed to the back-end compiler implemented in the driver. The driver compiles the shaders to the hardware native instructions and runs compiled shader through the optimizer. The optimizer knows many intricate details about hardware implementation and is able to allocate registers and schedule instructions in the most efficient way. The following sections will explain some of the hardware implementation details and what can be done to help driver optimizing your shaders.

6.1. Shader Management

Shader switching is one of the most expensive state changes. Batching rendering by vertex shader is always a good idea. When switches between shaders are inevitable, try limiting frequent switches only to recently used smaller shaders as driver and hardware can more effectively cache them. Switching between fixed function and programmable pipeline is in most cases more expensive that switching between equivalent shaders because of the extra driver overhead.

The shader compilation and optimization in the driver is quite a complex and expensive process and it will become even more expensive as shaders grow in size and shader models become more complex. Because of that it is a bad idea to compile too many shaders on the fly. Try pre-creating as many shaders upfront as possible.

6.2. Shader Constant Management

Updating high volumes of shader constants can add considerable amount of overhead to the drivers. Following strategies can help reducing driver overhead associated with constant updates. When there are a lot of scalar constant updates, pack these scalar values into vectors. This should reduce number of scalar constant updates by the factor of four. When picking locations for the frequently updated constants do not scatter them across the whole constant store. This will allow constant updates to happen in continuous ranges, which should reduce runtime and driver overhead. Consider fragmenting the constant store into 4 or 8 constant chunks and updating these chunks atomically. That is if you have to update every other constant in some constant range it is better to update the whole range at once than updating each changed constant individually.

6.3. Optimizing Vertex Shaders

When it comes to optimizing vertex shaders only a few optimizations apply. The reason for that is the driver shader optimizer that does a pretty good job of optimizing shaders.

One subtle vertex shader optimization is to output from the shader only what you need. For instance the shader can export duplicated texture coordinates only to fetch two different textures with the same coordinates. Many developers still do that; however 1.4 and especially 2.0 pixel shaders allow decoupling of texture coordinate sets from texture samplers. Just export unique texture coordinate values from the vertex shaders and use pixel shaders to do proper texture coordinate mapping. Also, when outputting texture coordinates use write masks to indicate how many texture coordinate components have to be interpolated and passed down to pixel shaders.

Fixed function vs. programmable pipelines

RADEON™ 8500/9000 chips have implemented both fixed function and programmable vertex processing in the silicon. Using fixed function with these chips can be slightly more efficient than using vertex shaders because of the optimized hardware implementation of the TnL pipeline. Using fixed function TnL also simplifies shader management and reduces the associated application and driver overhead. Having said that, shaders can be a better solution if used to pack vertex data or take some "shortcuts" in the vertex computations. There is no golden rule as the ultimate solution depends on shader usage and can only be found through extensive experimentation.

RADEON™ 9500/9700 on the other hand has only a programmable pipeline implemented the hardware, and fixed function TnL is emulated with the vertex shaders. This means that for DirectX® 9 class hardware there is no advantage in using fixed function functionality. Using flow control available in 2.0 vertex shaders solves the problem with shader management and allows application to toggle lights, texture transform and other parameters as easily as with a fixed function pipeline.

Use of flow control in VS 2.0

As flexible and as powerful the 1.0-1.1 vertex shaders are, they can also be a great nuisance. Rarely only a single shader is used – some objects require per-vertex lighting with one spot and one directional light, while others need tangent space setup and texture coordinate generation, and so on. By the time you consider all possible permutations that might be required, the number of shaders becomes astronomical. This is where 2.0 vertex shaders become handy. With addition of static flow control shader model has gained a robust mechanism for shader management. Instead of swapping a huge number of very specific shaders it is much better to write just a couple of universal shaders with flow control and replace expensive shader switches with lightweight boolean constant updates. On RADEON™ 9500/9700 flow control instructions are essentially free, however some performance degradation might still occur due to somewhat limited scope of performance optimizations.

Co-issue in vertex shaders

Radeon 9500/9700 has very interesting vertex processor unit design. Each of the vertex processors has two math engines, one vector and one scalar, that can process vector and scalar instruction on the same clock. The idea is somewhat similar to pixel shader co-issue, however there are implementation differences. Vector vertex processing engine operates on full 4D vectors, as opposed to 3D vectors in pixel shaders, and scalar vertex processing engine is more independent from the vector engine.

When the vertex shader optimizer schedules instructions it will try to pair vector and scalar operations for optimal execution. There are a few limitations that might prevent optimizer from co-issuing instructions. To increase the chances of instruction pairing, do not output to the destination registers from scalar instructions and always use write masks to write out only a single channel from scalar instructions such as **POW**, **EXP**, **LOG**, **RCP** and **RSQ**. Also be aware the read port limits apply for vector/scalar instruction pair the same way it is described in DirectX® 9 vertex shader specification for a single instruction.

6.4. Optimizing Pixel Shaders

As pixel shaders progressively become more and more complex, they become more and more an important target for optimizations. In the older 1.0-1.4 pixel shader models there is not that much room for optimization because of low shader complexity. The 2.0 shader model however is a different story. 2.0 pixel shaders are complex enough to implement different optimization strategies, so the following sections will mostly focus on

RADEON™ 9500/9700 pixel shader engine architecture and various pixel shader optimization tricks.

Texture instructions

Texture instructions are the pixel shader instructions that fetch texture such as **TEXLD**, kill pixel processing – **TEXKILL**, and **TEXDEPTH** for 1.4 pixel shaders outputs depth values. When it comes to texture instructions there are few things to be aware of. First, **TEXKILL** instruction does not interrupt pixel shader processing and provides pixel culling only after shader was completely executed. Thus positioning of the **TEXKILL** instruction in the shader does not make any difference and it is wrong to rely on early abortion of pixel shader execution.

In general **TEXKILL** and **TEXDEPTH** (or equivalent depth output in 2.0 pixel shaders with **oDepth**) should be used very carefully because they interfere with HYPER Z™ operation, and if possible should be avoided.

TEXKILL and clip planes

The **TEXKILL** instruction cancels the rendering of pixel based on the texture coordinate values provided. This functionality can be used to implement user clip planes at the rasterizer level. While this is an interesting hack, it does not provide the most efficient way of implementing clip planes. All RADEON™ family chips have support for 6 geometry based clip planes in the TnL engine. Considering that **TEXKILL** instruction has some detrimental impact on performance, as previously described, it is much better to use real clip planes. Use **TEXKILL** only when clipping cannot be properly handled with conventional user clip planes.

Legacy pixel shaders on DirectX® 9 hardware

When designing the RADEON™ 9500/9700 family of chips, one important objective was to create architecture backwards compatible with legacy shader models that would provide the highest performance possible. This resulted in pixel shader engine architecture that natively supports shader instruction co-issue, and most of the source argument and instruction modifiers. Since 2.0 pixel shader model has very limited support for modifiers, they have to be emulated with extra instructions. This means that some of the legacy pixel shaders featuring many modifiers will execute faster than their 2.0 pixel shader equivalents.

Co-issue in pixel shaders

Earlier pixel shader models, namely 1.0-1.4, had a feature called instruction co-issue. It allowed pairing two instructions operating on color and alpha values in one, and executing them on the same cycle. While instruction co-issue provided a great opportunity for optimization and increase of maximum number of instructions, it did complicate shader development and broke instruction and operand orthogonality in the shader model. The co-issue was removed from 2.0 pixel shader model.

RADEON™ 9500/9700 chips have dual-pipe pixel shader units, which operate as two relatively independent engines performing calculations on the different entities. One engine operates on 3D vectors or RGB-colors and the other on scalar or alpha values. This means that in most cases two instructions, one operating on the color and another operating on alpha can be performed at the same time. Such architecture provides a perfect opportunity for optimizing shaders by splitting the computational workload between pipes and thus resulting in up to twofold speedup. Careful examination of a shader for splitting the workload between the pipes should focus on a couple of things – identifying computations that can be executed only in one pipe (vector or scalar) and balancing number of instructions in each pipe. Sometimes scalar or alpha computations can be executed in the color pipe and the other way around, the color computations can be executed in the alpha pipe.

Explicit instruction co-issue in pixel shaders is available only in the older shader models. However, this does not mean that the benefits of instruction pairing can be enjoyed only in the older pixel shader models. On the contrary, the full benefit of instruction co-issue can be achieved in 2.0 pixel shaders with some clever shader programming. In 2.0 pixel shader model, write masks can be used to implicitly indicate opportunity for instruction pairing. The shader optimizer in RADEON™ 9500/9700 drivers will look for write masks to determine which pipe should execute instruction and will try reordering and co-issuing instructions.

There are some nuances the shader developers have to be aware of when optimizing shaders for instruction co-issue. The color and alpha parts of the instruction pair can reference different registers, however attempting to access alpha values in color instruction or to access color values in alpha instructions might break co-issue. This also applies to **.ABGR** or **.WZYX** swizzles available in 2.0 shaders as they force data to cross vector and scalar pipes.

Another important fact is that **RCP**, **RSQ**, **EXP** and **LOG** instructions are always executed in the scalar pipe. For that reason it is better to always use scalar arguments and destinations (**.W** or **.A**) when using these instructions. This will ensure the vector pipe is available for co-issue with these instructions.

Following are fragments of pixel shaders that compute diffuse and specular lighting. This demonstrates how splitting instructions between pipes for co-issue can be used to optimize shaders.

```
ps.2.0
...
dp3 r0.r, r1, r0   // N.H
dp3 r2, r1, r2     // N.L
mul r2, r2, r3     // * color
mul r2, r2, r4     // * texture
mul r0.r, r0.r, r0.r // spec^2
mul r0.r, r0.r, r0.r // spec^4
mul r0.r, r0.r, r0.r // spec^8
mad r0.rgb, r0.r, r5, r2
...
```

Total – 8 instructions

```
ps.2.0
...
dp3 r0.a, r1, r0    // N.H
dp3 r2, r1, r2      // N.L
mul r0.a, r0.a, r0.a  // spec^2
mul r2.rgb, r2, r3    // * color
mul r0.a, r0.a, r0.a // spec^4
mul r2.rgb, r2, r4    // * texture
mul r0.a, r0.a, r0.a // spec^8
mad r0.rgb, r0.a, r5, r2
...
```

Total – 5 instructions

The instructions shown in purple color in the first shader are the instructions that could be co-issued if they were executed in scalar pipe. The second shader illustrates the result of such co-issue with blue and red instruction pairs. It is not required to place instructions that can be paired next to each other, since the shader optimizer can intelligently reorder instructions. In this example the instructions were reordered only to illustrate a concept.

Instruction balancing

On RADEON™ 9500/9700 the highest possible performance of pixel shaders can be achieved by carefully balancing number of texture and arithmetic instructions. Each of the pixel shader engines of RADEON™ 9500/9700 is capable of executing a texture fetch and color/alpha ALU instruction pair on each clock cycle. Because of this high degree of parallelism between texture units and math engines, it is a good idea to keep ratio of texture to ALU instructions close to 1:1. This of course makes sense if application is not texture fetch bound. When using more expensive texture filtering modes the ratio of instructions will be skewed more towards higher number of ALU instructions. For each particular pixel shader the cost of arithmetic vs. texture instructions should be carefully evaluated to find areas that can be implemented more optimally. For instance, if shader is too long because of some complex calculations and there is some memory bandwidth to spare, some function look-up tables can be used to reduce a number of arithmetic instructions. The perfect example is **SINCOS** macro that can be much more efficiently implemented as a texture fetch of one or two channel texture. This instruction balancing should be performed separately at each dependency level in the shader.

Dependent texture reads

Dependent texture reads are quite expensive. On RADEON™ 8500/9000 a two-phase shader is much more expensive than a single-phase shader, however it should be less expensive than running multiple render passes with single-pass shaders. If you are developing application that targets DirectX® 8.1 hardware and uses multiple render passes with 1.0-1.3 pixel shaders, consider implementing a single-pass solution with 1.4 pixel shaders.

RADEON™ 9500/9700 has significantly optimized dependent texture read implementation for performance and efficiency. As well the number of levels of dependency has been increased to four in 2.0 pixel shader model. The best performance on RADEON™ 9500/9700 can be achieved when not exceeding two dependent texture reads. While three or four levels of dependency will provide sufficient performance, it will not be as good as with only one or two levels. Keep in mind that if arithmetic instructions are used to compute texture coordinates before the first texture fetch, they will also be counted as a level of dependency. Also, bear in mind that **TEXKILL** instruction forces a dependency level change in the pixel shaders. To optimize shaders with dependent texture reads try to keep the number of both texture and arithmetic instructions roughly the same at each level of dependency.

6.5. When Multiple Render Passes Are Better than One

One obvious optimization technique is to reduce number of rendering passes. This allows cutting down on the amount of transformed and rendered geometry and decreasing fillrate requirements. However, it turns out that this is not always true. There are a growing number of cases where multi-pass rendering can result in better performance. Consider a situation when overdraw is very high, and complex pixel shaders are used. When using long and complex pixel shaders, the chances are the performance might be hampered by shader execution. If overdraw is high, these complex shaders are run many times even for the pixels that are occluded by other geometry. This is a huge waste of VPU shader processing power. Ordering all geometry by distance and rendering it front to back might not be such a good idea since it might affect sorting by effect, shader or render state. The solution is to use multi-pass rendering since vertex processing is rarely a bottleneck. On the first render pass just initialize the depth buffer with proper depth values for your scene by rendering all geometry without any pixel shaders and outputting only depth and no color information. Since no shaders are used, it is possible to render everything in the front to back order without causing any major render state changing overhead. Then render everything once again with proper shaders. Because depth buffer is already initialized with proper depth values, the early pixel rejection can happen due to HYPER Z™ optimizations, thus creating effective overdraw of one on the shader pass. Of course if overdraw is already low, there is no sense in using this technique. As scene and shader complexity increases this rendering method becomes more and more important.

6.6. Using High Level Shader Languages

It might be a lot of fun to develop vertex and pixel shaders in assembly, while chasing every single opportunity to squeeze out an extra execution cycle here and there with highly optimized handcrafted assembly code. In the real world of big and demanding projects and tight schedules this just might not be the most practical way of developing shaders. It is a well-known fact that higher-level languages provide much better productivity.

With the introduction of Direct® 9, High Level Shader Language (HLSL) was introduced. This C-like language with extra constructs to deal with vectors, matrices and other graphics related features could be a great help in shader development. Besides greater readability of the code and overall increased productivity, using high-level language instead of assembly allows us to focus on code reuse and high-level algorithmic optimizations, which quite often can be more valuable than low-level optimizations.

It does not mean that low-level optimizations are unimportant. The HLSL compiler is aware of many low-level optimization tricks and can produce code that rivals some of the best handcrafted assembly. To help compiler recognize optimization opportunities, use appropriate types of variables. If only a scalar needs to be computed, do not use a vector to store it. Likewise, use **float3** type to store 3-component vectors and so on. When computing shader output values for everything other than texture coordinates use **float4** variable for holding the final result and avoid using type casts.

Another good way to help compiler recognize areas for possible optimizations is to use built-in intrinsic functions. For instance, use **dot()** or **lerp()** functions instead of implementing your own functional equivalents.

In HLSL pixel shaders make sure to use **tex1D()** intrinsic function whenever it makes sense. In DirectX® 9 there are no 1D textures, so they are emulated through 2D textures Using **tex1D()** instead of **tex2D()** can sometimes save an extra instruction when sampling a texture with 1x*N* dimensions, since there is no need to worry about second texture coordinate component.

If for some reason shader performance is still bellow your expectations, have HLSL compile high-level code into assembly and then go over it with fine-tooth comb.

7. Using Textures

When using textures, one of the biggest mistakes is using textures that are too big. Quite often artists will use huge textures to make objects look “better”, but in many cases instead of creating improved look they make performance suffer. For instance, it makes no sense to use something like 1Kx1K texture on an object that never becomes bigger than quarter or half screen size. Using big textures wastes precious video memory, can contribute to memory fragmentation and limit resource swapping ability, as well as decreases texture cache efficiency.

When rendering to textures, attempt to use non-power of two textures to reduce memory footprint. Padding the texture size to the power of two can waste quite a bit of memory – for instance at 1600x1200 resolution the overhead is over 8.6Mb per render target. ATI’s hardware has efficient implementation for non-power of two renderable textures and their performance should be on par with power of two textures. The limitations that have to be considered are the ones exposed with **D3DPTEXTURECAPS_NONPOW2CONDITIONAL** cap – texture addressing modes have to be set to **D3DTADDRESS_CLAMP**, texture wrapping should be disabled and mipmapping cannot be used.

Another potential area for optimizations is texture changes. Do not change textures too often, and in general follow the guidelines set in the discussion on Render State Management. Batching small textures together into a bigger texture can help reducing number of texture switches, however it presents some other challenges. Sub-texture regions have to be padded to prevent sampling from wrong location, especially in case of multisampling. Also, this approach limits the usability of mipmapping. Carefully weigh all the pros and cons of this texture packing method.

Updating textures

If there is a requirement for dynamically updated textures, check for availability of dynamic texture support. In DirectX® 9 **D3DCAPS2_DYNAMICTEXTURES** cap indicates that textures can be created as dynamic textures, that is they can be locked even if they are allocated in video memory. To create a dynamic texture use **D3DUSAGE_DYNAMIC** creation flag and create texture in **D3DPOOL_DEFAULT** memory pool since they cannot be managed. **D3DLOCK_DISCARD** lock flag can be used when updating dynamic textures.

7.1. Using Texture Cache Effectively

Different caches are used in different parts of the graphics pipeline to hide the latency of memory accesses and reduce the memory bandwidth. The texture units are not an exception. This means that the most efficient use of textures is when texture accesses are optimized for locality of reference. Sampling from all over the texture on subsequent fetches for nearby pixels can thrash the texture cache and result in poor performance.

This can happen when using big bilinear filtered texture mapped on an object appearing small in screen space. Using bilinear filtering with a mipmap can help cache efficiency.

Drivers might employ some heuristics to detect texture usage and try configuring texture cache in the most efficient way. To make this driver task easier consider using consistent texture usage throughout the application. For instance always bind more cache demanding textures to the lower texture samplers, use cube maps on the same sampler all the time and etc.

7.2. Texture Filtering

Whenever texture fetching is a bottleneck, carefully picked filtering modes can greatly improve performance. It all comes down to experimenting with performance and carefully balancing texture caching, raw memory bandwidth and cost of texture filtering. The trilinear filtering can improve texture cache efficiency and visual quality of rendered images; however it can be more expensive than bilinear filtering since twice as many samples have to be fetched. Avoid using LOD-bias for sharpening images as it hurts performance. Bear in mind that LOD-bias is different for various hardware vendors with different driver quality settings. It is non unusual to see driver implementations tweaking LOD-bias for performance, that might result in over-sharpened textures and reduced performance on ATI hardware. Consider checking LOD-bias settings on a wide range of hardware to ensure the ultimate image quality and performance.

Anisotropic filtering is even more expensive because of even higher number of samples required. RADEON™ family chips employ anisotropic filtering optimizations through use of adaptive sampling that makes performance reasonable even at 16:1 maximum anisotropy level. Trilinear anisotropic filtering is the most expensive and should be used only when absolutely necessary. Use of anisotropic filtering with environment bump mapping is also not advisable since visual benefits of it are rather questionable.

RADEON™ 9500/9700 can perform point or bilinear filtering of one texture request per clock cycle per pixel shader pipe, if texture format does not exceed 32 bits. For texture formats fatter than 32 bits it will take 2 clocks for processing 64 bit texture formats and 4 clocks for 128 bit formats. Trilinear filtering doubles number of clocks because it requires two bilinear blends. For all floating point formats on RADEON™ 9500/9700 only point filtering is supported.

While some of the texture filtering modes are quite expensive, it does not mean they should not be used at all. Because of the texture instructions vs. ALU instructions balancing in RADEON™ 9500/9700 it might be reasonable to use more expensive filtering modes and fatter texture formats if shaders are rather ALU computation bound. Refer to the discussion on pixel shader instruction balancing for more information.

7.3. Reducing Texture Bandwidth and Storage

With demand for greater visual realism grows the need for texture bandwidth, as higher number of textures is used for rendering sophisticated visual effects. Under such circumstances any memory size and bandwidth improvements positively reflect on overall 3D graphics performance. Whenever possible use DXTn compressed textures to reduce texture footprint and required memory bandwidth. While in most cases compressed textures provide good quality vs. size tradeoff, sometimes they result in visual artifacts. Textures containing a lot of noise, such as images of asphalt, stone and etc. can be compressed without any noticeable artifacts. On the other hand, some textures with diagonal or circular gradients are not very good candidates for compression. If compression produces inadequate image quality consider using other smaller formats. Some textures can be stored in 16 bit formats without creating major artifacts. Consider other smaller formats such as **D3DFMT_A8** and others. Light maps are the perfect candidates for such slim texture formats.

If possible, consider packing textures together. When using multiple textures fetched with the same texture coordinates check for unused channels and reuse them. For example, if only RGB channels of the base map are used, the alpha channel can be used for storing something like a gloss map. This is especially beneficial for expensive texture filtering modes such as anisotropic filtering.

Dealing with normal maps

Per-pixel lighting and many other effects require normal maps. The problem with normal maps is that they do not compress well with standard color compression algorithms. Using DXTn compression on normal maps produces nasty visual artifacts that can only be amplified by further computations such as raising values to some power. One of the solutions for compact storage of normal maps on RADEON™ 9500/9700 is to use two-channel textures and derive the third component of the normal vector in the 2.0 pixel shaders. This reduces the storage and memory bandwidth in half, and is especially beneficial if application is not shader computation bound. But even in cases where shaders are already a bottleneck, this approach might still prove to be useful if it reduces a number of texture swaps and prevents texture spilling into AGP. The good formats to use are – **D3DFMT_V8U8** and **D3DFMT_V16U16**.

8. Depth Buffers

When it comes to depth buffers there are two main causes of bottleneck – video memory bandwidth and application accesses to the depth buffer. The former is dealt with at the hardware level with help of the HYPER Z™ technology. To further help with bandwidth limitation disable depth buffering when it is not really needed. For example, when rendering application user interface or alpha blending HUD's (heads-up display) neither depth reads or depth writes are needed.

If application is using multiple depth buffers, the most render-intensive depth buffers should be allocated first, since they would have a higher performance. Create all of the depth buffers before creating other resources such as vertex buffers and textures. If possible, avoid creating multiple instances of depth buffers for multiple render targets. Reuse the same depth buffer with different render targets by just clearing it before rendering to another render target.

The depth buffer access issue comes from the fact that some applications lock depth buffer and read data from it. It is a very bad idea to lock depth buffer for a couple of reasons. First and the most important, it breaks CPU and VPU parallelism by stalling the pipeline and waiting for all pending rendering to finish. As processors and graphics chips become more and more powerful, the bigger effect stalling has on the relative performance. Another reason why locking depth buffer is bad is special data formats that might be employed by the graphics accelerators. If HYPER Z™ is enabled, the VPU will be forced to uncompress depth buffer and disable optimizations such as Hierarchical Z since depth buffer contents might change without graphics hardware knowledge. Also most of the RADEON™ family chips only support swizzled depth buffer formats, which means CPU will have to un-swizzle depth buffer and swizzle it back whenever buffer is locked. This can totally destroy application performance.

Alternatives to locking depth buffer

Most likely depth buffers are locked for performing some kind of visibility testing. For example sun or any other light source might be examined for visibility when implementing flare effect. DirectX® 9 has a great alternative for implementing this kind of effect – occlusion query (**D3DQUERYTYPE_OCCLUSION** query type) that returns number of pixels that pass z-test. The only slight limitation of this method is delayed result of the query by 1-2 frames, which should not be a big deal in case of rendering light flares.

8.1. What Is HYPER Z™ and How to Make It Work

The video memory bandwidth is obviously one of the biggest bottlenecks of the graphics subsystem. As depth buffer accesses consume a great chunk of that bandwidth, it was realized that something needs to be done to reduce this memory bandwidth and increase overall system efficiency. This is how HYPER Z™ was born. The HYPER Z™ is a combination of several technologies that make it all possible – Fast Z Clears, Z Compression, Hierarchical Z and for RADEON™ 9500/9700 also Top of the Pipe Z Reject.

Fast Z Clears

Fast Z Clear technique allows to clear depth buffer without actually writing out depth values for all pixels in the depth buffer. Not only it is essentially free, but it also initializes buffer into an optimized state for the first depth read. By clearing depth buffer you can have much more optimized depth buffer operation than by using various tricks that avoid depth buffer clears. To make sure Fast Z Clear technique is used, use Clear function instead of clearing depth with your own geometry. Clearing the whole depth buffer surface makes Fast Z Clears as efficient as it can be. If stencil buffer is also present, it has to be cleared together with depth buffer for Fast Z Clears to happen.

Z Compression

RADEON™ chips employ a lossless depth buffer compression that on RADEON™ 9500/9700 in extreme cases can achieve up to 24:1 compression rate. This compression is fully transparent to the API or application and there is not much that can be done to optimize it.

Hierarchical Z

By far the most important part of HYPER Z™ is the hierarchical depth testing. This technique allows culling of large pixel blocks very efficiently based on the hierarchical view of the depth buffer that is stored on the chip. Unlike previous chips, RADEON™ 9500/9700 performs multiple hierarchical depth tests at the various places in the pipeline, making this technique even more effective.

There are a couple of rules that have to be followed to reap the benefits of the Hierarchical Z. First, and the most important, do not change sense of the depth comparison function in the course of a frame rendering. That is if using **D3DCMP_LESS** depth function, do not change it to **D3DCMP_GREATER** for some part of a frame. Second, **D3DCMP_EQUAL** and **D3DCMP_NOTEQUAL** depth comparison functions are not very compatible with Hierarchical Z operation, so avoid them if possible or replace them with other depth comparisons such as **D3DCMP_LESSEQUAL**. In addition, few other things interfere with hierarchical culling; these are – outputting depth values from pixel shaders and using stencil fail and stencil depth fail operations. Last but not least, for the highest Hierarchical Z efficiency place near and far clipping planes to enclose scene geometry as tightly as possible, and of course render everything front to back.

Top of the Pipe Z Reject

RADEON™ 9500/9700 has the ability to cull pixels very early in the pipeline, before any of the pixel shader processing takes place. This is especially beneficial when using big, expensive pixel shaders. To take advantage of Top of the Pipe Z Reject do not kill pixels with **TEXKILL** shader instruction or alpha test, and do not output depth values from the pixel shader.

9. Stencil Buffers

A stencil buffer is created and stored together with a depth buffer, so it should not be treated any differently from a depth buffer. Make sure to clear stencil buffer together with a depth buffer and avoid partial surface clears. Check section on HYPER Z™ to see how stencil operations can affect graphics performance.

9.1. Two-Sided Stencil

Shadow volume technique is the most popular shadow technique used in recent 3D graphics applications. In this technique shadow volumes cast by occluded geometry are computed and rendered to the stencil buffer twice. On the first pass front-facing triangles of the shadow volume increment stencil values, while on the second pass back-facing triangles decrement stencil values. Non-zero stencil values after rendering shadow volume indicate shadowed pixels. The problem with this technique is immense fillrate and increased geometry processing requirements due to the multiple passes of shadow volume rendering. DirectX® includes an optimization for shadow volume rendering that is supported by RADEON™ 9500/9700. This technique is called Two-Sided Stencil. It allows merging of both shadow volume render passes by providing two sets of stencil operations based on the triangle face orientation – one for clockwise triangles and another for counterclockwise triangles. Check **D3DSTENCILCAPS_TWOSIDED** cap to test for Two-Sided Stencil support. **D3DRS_TWOSIDEDSTENCILMODE** render state is used to enable Two-Sided Stencil operation, while **D3DRS_CCW_STENCILFAIL**, **D3DRS_CCW_STENCILZFAIL**, **D3DRS_CCW_STENCILPASS** and **D3DRS_CCW_STENCILFUNC** render states specify stencil operation for counterclockwise triangles. For clockwise triangles regular stencil operation render states are used.

10. Renderable Buffers and Rasterization

Similar to depth buffer, locking back buffer can seriously cripple performance, so whenever possible avoid locking it. Just like with depth buffer, locking back buffer destroys CPU and VPU parallelism and might affect back buffer surface swizzling. Surfaces are normally swizzled for higher memory access efficiency, and forcing back buffer to become un-swizzled might cost some fillrate. If it is necessary to lock off-screen plain surfaces or render targets use **D3DLOCK_DONOTWAIT** flag with **LockRect** method. Instead of spinning in the runtime or driver, the function can immediately return with **D3DERRR_WASSTILLDRAWING** value if buffer cannot be instantly locked due to pending rendering. This allows application to reclaim CPU cycles, which should help with CPU and VPU parallelism.

In general AGP and especially local video memory read accesses are very slow, so reading big portions of color buffers with CPU can be very expensive and should be avoided.

Color clears can be another area for optimization. When clearing back buffer use **Clear** API call and do not attempt to clear the buffer by rendering a screen space quad. For the best clear performance avoid partial clears, as they might interfere with hardware optimizations available for clearing buffers. In most cases it might be even better to avoid clearing back buffer at all and rely on re-rendering the whole frame every time.

When creating render targets make sure to create them before vertex buffers, index buffer and textures, as explained in discussion on resource creation.

Rasterization and alpha blending

A big part of video memory bandwidth is used for outputting color information into a back buffer. Carefully controlling rasterization can help ease the fillrate burden. Color write masks can be used to save memory bandwidth when no color needs to be output. A typical case would be laying out depth buffer on multi-pass rendering or rendering shadow volume.

Using alpha blending can even further strain memory bandwidth. To help this situation RADEON™ 9500/9700 employs a number of optimizations. The hardware can look at the incoming color and detect pixels that can be discarded. It is possible for alpha channel to contain irrelevant information that can interfere with this optimization. Following table summarizes the blend modes that can be optimized and what needs to be forced into alpha channel if it contains otherwise useless information.

D3DRS_BLENDOP	D3DRS_SRCBLEND	D3DRS_DESTBLEND	Force alpha value
ADD	ONE	ONE	0.0
ADD	ZERO	SRCCOLOR	1.0
ADD	DESTCOLOR	ZERO	1.0

Multisampling

RADEON™ 9500/9700 is the first ATI hardware that implements true multisampling with 2, 4 and 6 samples per pixel. Locking back buffer with multisampling enabled can cause some extra performance degradation since back buffer has to be resolved by VPU before it can be locked.

RADEON™ 9500/9700 includes a number of optimizations that make multisampling performance very acceptable. One such optimization is color compression that reduces memory bandwidth required for accessing multiple samples. Because of that do not expect performance to drop linearly with number of samples. In many cases involving pretty heavy texture usage and alpha blending, 2x multisampling is free. Visual quality improvement vs. its cost makes multisampling "a must" feature for most applications.

Keep in mind that a direct benefit from enabling multisampling on primary buffer can only be derived when rendering to that buffer. Rendering to textures and then compositing the intermediate rendering results to the multisampled primary buffer, as done in various screen space post-processing techniques such as depth-of-field or tone mapping, invalidates the benefits of multisampling. In this case multisampling will only waste valuable video memory and performance.

11. Presenting Scene

The default behavior of **Present** method prior to DirectX® 9 was to wait for VPU to become available for performing presentation operation, thus causing a stall in runtime or driver. This behavior changed in DirectX® 9 by including a special presentation flag in swap chain **Present** method. Instead of using device's **Present** method (which is really a shortcut to swap chain's presentation function) retrieve a swap chain and use it for presentation with **D3DPRESENT_DONOTWAIT** flag. If VPU is currently unavailable for performing presentation, **Present** will return **D3DERR_WASSTILLDRAWING** error code. Using this presentation method can give scarce CPU cycles back to the application and improve CPU and graphics hardware parallelism.