CONSOLE API EFFICIENCY & PERFORMANCE ON PCs

An Overview of Direct3D* 12

Abstract

Microsoft Direct3D* 12 is an exciting leap forward in PC gaming technology, giving developers greater control of their games and adding greater CPU efficiency and scalability.

Michael Coppock
michael.j.coppock@intel.com
Table of Contents

Introduction ........................................................................................................................................... 2

1.0 – Blast from the Past ..................................................................................................................... 2

1.1 - Closer to the Metal ...................................................................................................................... 3

2.0 – Pipeline State Object ............................................................................................................... 4

3.0 – Resource Binding .................................................................................................................... 8

3.1 – Resource Hazards ..................................................................................................................... 9

3.2 – Resource Residency Management .......................................................................................... 10

3.3 – State Mirroring ....................................................................................................................... 10

4.0 – Heaps and Tables .................................................................................................................... 11

4.1 – Redundant Resource Binding .................................................................................................. 11

4.2 – Descriptors ............................................................................................................................. 12

4.3 – Heaps ....................................................................................................................................... 12

4.4 – Tables ....................................................................................................................................... 13

4.5 – Bindless and Efficient .............................................................................................................. 14

4.6 – Render Context Review ........................................................................................................... 14

5.0 – Bundles ...................................................................................................................................... 16

5.1 – Redundant Render Commands ............................................................................................... 16

5.2 – What are Bundles? .................................................................................................................. 17

5.3 – Code Efficiency ....................................................................................................................... 18

6.0 – Command Lists ....................................................................................................................... 20

6.1 – Command Creation Parallelism ............................................................................................. 20

6.2 – Lists and the Queue ............................................................................................................... 21

6.3 – Command Queue Flow ......................................................................................................... 22

7.0 – Dynamic Heaps ...................................................................................................................... 23

8.0 – CPU Parallelism ...................................................................................................................... 24

9.0 – Summary ................................................................................................................................ 27

References and Related Links ......................................................................................................... 28

Notices and Disclaimers .................................................................................................................. 28
At GDC 2014, Microsoft announced stunning news for PC Gaming in 2015—the next iteration of Direct3D, version 12. D3D 12 returns to low level programming; it gives more control to game developers and introduces many new exciting features. The D3D 12 development team is focused on reducing CPU overhead and increasing scalability across CPU cores. The goal is console API efficiency and performance, console games can use the CPU/GPU more effectively with great results. In the world of PC gaming Thread 0 often does most, if not all the work. The other threads only handle OS or other system tasks. Few truly multithreaded PC games exist. Microsoft wants to change that with D3D 12, which is a superset of D3D 11 rendering functionality. Meaning modern GPUs can run D3D 12, as it will more effectively use today's multi-core CPUs and GPUs. No need to spend money on a new GPU to get the advantage of D3D 12. The future of PC Gaming on Intel® processor-based systems is very bright indeed!

1.0 Blast from the Past

Low-level programming is common in the console industry because the specifications of each console are fixed. Game developers can take time to fine-tune their games and squeeze all the power and performance possible out of the Xbox One* or PlayStation* 4. PCs, by their nature, are a flexible platform, with countless options. There are many permutations to plan for when developing a new PC game. High-level APIs like OpenGL* and Direct3D* help simplify development. The problem is that the API, and to a slightly lesser extent the drivers, have reached such a level of complexity that they can get in the way when rendering a frame, and that incurs a performance cost. That is where low-level programming comes into play.

Low-level programming on PCs died off when the MS-DOS* era ended, and vendor-specific APIs, like 3Dglide* by 3DFX* made way for APIs like Direct3D. PCs lost performance in the name of much needed convenience and flexibility. The hardware market had grown complex, with too many options. Development time increased because developers wanted to make sure everyone who bought their game could play it. Not only had things changed on the software side, but CPU power efficiency became more important than performance. Now, though, instead of focusing on raw GHz, multiple cores and threads in CPUs and parallel rendering on modern GPUs are where the future performance gains lie. It is time for PC gaming to look to game consoles for direction. It is time for PC gaming to enter the 21st century.
1.1 Closer to the Metal

To bring games “closer to the metal,” we must reduce the size and complexity of the API and driver. There should be fewer layers between the hardware and the game itself. The API and driver spend too much time translating commands and calls. Some, if not most, of that control will be given back to the game developer. D3D 12’s reduced overhead will improve performance, and fewer layers between the game and GPU hardware will mean better looking and performing games. The other side to this, of course, is that some developers may not want to control areas the API used to deal with, such as GPU memory management. Perhaps that is where game engine developers come in, but only time will tell. As the release of D3D 12 is still a ways off, there is plenty of time to figure that part out. So with all these great promises, how will it be accomplished? Primarily through four new features called the Pipeline State Object, Command Lists, Bundles, and Heaps.
2.0 Pipeline State Object

To discuss the Pipeline State Object, or PSO, let's first review the D3D 11 render context, then cover the changes in D3D 12. Figure 1 contains the D3D 11 render context as shown by Max McMullen, the D3D Development Lead, at BUILD 2014 in April.

![Render Context: Direct3D 11](image)

*Figure 1: The D3D 11 Render Context. [Reprinted with permission from Microsoft.]*

The large bold arrows indicate the individual pipeline states. Each state can be retrieved or set based on the needs of the game. The other states at the bottom are fixed function states such as viewport or scissor rect. The other relevant features in this diagram will be explained in later sections of this article. For the PSO discussion, we only need to review the left side of the diagram. D3D 11’s small state objects did reduce the CPU overhead from D3D 9, but there was still additional work in the driver taking these small state objects and combining them into GPU code at render time. Let's call it the hardware mismatch overhead. Take a look at another diagram from BUILD 2014 in Figure 2.
Direct3D 11 – Pipeline State Overhead

Small state objects $\rightarrow$ Hardware mismatch overhead

The left side shows a D3D 9 style pipeline, which is what the application uses to do its work. The hardware on the right side of Figure 2 needs to be programmed. State 1 represents the shader code. State 2 is a combination of the rasterizer and the control flow linking the rasterizer to the shaders. State 3 is the linkage between the blend and pixel shader. The D3D Vertex Shader effects hardware states 1 & 2, the Rasterizer state 2, Pixel shader states 1-3, and so on. Most drivers do not want to submit the calls at the same time as the application. They prefer to record and defer until the work is complete so that they can see what the application actually wants. This means additional CPU overhead as old and outdated data is marked “dirty.” The driver’s control flow checks the states of each object at draw time and programs the hardware to match the state the game has set. With the extra work, resources are drained and things can go wrong. Ideally once the game sets the pipeline state, the driver knows what the game intends and programs the hardware once. Figure 3 shows the D3D 12 pipeline, which does just that in what is called the Pipeline State Object (PSO).
Direct3D 12 – Pipeline State Optimization

Figure 3: The D3D 12 pipeline state optimizations streamline the process.

Figure 3 shows a streamlined process with less overhead. A single PSO with the state information for each shader can set all the HW states in one copy. Remember some states were marked “other” in the D3D 11 render context. The D3D 12 team realized the importance of minimizing the PSO size and allowing the game to change the render target without affecting a compiled PSO. Things like viewport and scissor rect are left separate and programmed orthogonally to the rest of the pipeline (Figure 4).
Instead of setting and reading each individual state, we now have a single point; reducing or removing entirely the hardware mismatch overhead. The application sets the PSO as it needs while the driver takes the API commands and translates them to GPU code without the additional flow control overhead. This “closer to the metal” approach means that draw commands take fewer cycles, increasing performance.
3.0 Resource Binding

Before we discuss resource binding changes we need to quickly review the resource binding model used in D3D 11. Figure 5 shows the render context diagram again with the D3D 12 PSO on the left and the D3D 11 resource binding model on the right.

![Render Context: Pipeline State Object (PSO)](image)

*Figure 5: The Render Context with the D3D 12 PSO on the left and the D3D 11 resource binding model on the right.*

In Figure 5, the explicit bind points are to the right of each shader. An explicit bind model means that each stage in the pipeline has specific resources it can refer to. Those bind points reference resources in GPU memory. These can be textures, render targets, buffers, UAVs, etc. Resource binding has been around a long time; in fact it predates D3D. The goal is to handle multiple properties behind the scenes and help the game efficiently submit rendering commands. However the system needs to execute many binding inspections in three key areas. The next section reviews these areas and how the D3D team optimized them for D3D 12.
3.1 Resource Hazards

Hazards are usually a transition, like moving from a render target to a texture. A game may render a frame that is meant to be an environment map around that scene. The game finishes rendering the environment map and now wants to use it as a texture. During this process, both the runtime and the driver track when something is bound as either a render target or texture. If the runtime or driver ever see something bound as both, they will unbind the oldest setting and respect the newest. This way a game can switch as needed and the software stack manages the switch behind the scenes. The driver must also flush the GPU pipeline in order for the render target to be used as a texture. Otherwise, the pixels are read before they are retired in the GPU, and you will not have a coherent state. In essence a hazard is anything that requires extra work in the GPU to ensure coherent data.

As with other features and enhancements in D3D 12, the solution is to give more control to the game. Why should the API and driver do all the work and tracking when it is one point in a frame? It takes roughly a 60th of a second to switch from one resource to another. The overhead is reduced by giving control back to the game, and the cost only has to be paid once, when the game makes a resource transition (Figure 6).

```c
D3D12_RESOURCE_BARRIER_DESC Desc;
Desc.Type = D3D12_RESOURCE_BARRIER_TYPE_TRANSITION;
Desc.Transition.pResource = pRTTexture;
Desc.Transition.Subresource = D3D12_RESOURCE_BARRIER_ALL_SUBRESOURCES;
Desc.Transition.StateBefore = D3D12_RESOURCE_USAGE_RENDER_TARGET;
Desc.Transition.StateAfter  = D3D12_RESOURCE_USAGE_PIXEL_SHADER_RESOURCE;
pContext->ResourceBarrier( 1, &Desc );
```

*Figure 6: The resource barrier API added to D3D 12*

The resource barrier API in Figure 6 declares a resource and both its source and target usage, followed by the call to tell the runtime and driver about the transition. It becomes something explicit instead of something tracked across the frame render with lots of conditional logic, making it a single time per frame, or whatever frequency the game needs to makes transitions.
3.2 Resource residency management

D3D 11 (and older versions) behave as if the calls are queued. The game believes the API immediately executes the call. This is not true. The GPU has a queue of commands where everything is deferred and executed later. While this allows for greater parallelism and efficiency between the GPU and CPU, it requires a lot of reference counting and tracking. All this counting and tracking require CPU effort.

To fix this, the game gets explicit control over the resource lifetime. D3D 12 no longer hides the queued nature of a GPU. A Fence API has been added to track GPU progress. The game can check at a given point (maybe once per frame) and see what resources are no longer needed, and then that memory can be freed for other uses. Tracking resources for the duration of the frame render using additional logic to release resources and free memory is no longer needed.

3.3 State Mirroring

Once the three areas above were optimized, an additional element that could be made more efficient was discovered (though with smaller performance gains). When a bind point is set, the runtime tracks the point so that the game can call a Get later to find out what is bound to the pipeline. The bind point is mirrored, or copied. A feature designed to make things easier for middleware so that componentized software can discover the current state of the rendering context. Once resource binding was optimized, mirrored copies of states were no longer needed. In addition to the flow control removal from the previous three areas, the Gets for state mirroring were also removed.
4.0 Heaps and Tables

There is one more important change to resource binding to review. At the end of Section 4 the entire D3D 12 render context will be revealed. The new D3D 12 render context is the first big step towards the goal of greater CPU efficiency in the API.

4.1 Redundant Resource Binding

After analyzing several games, the D3D development team observed that typically games use the same sequence of commands from one frame to the next. Not only the commands, but the bindings tend to be the same frame over frame. The CPU generates a series of bindings, say 12, to draw an object in a frame. Often the CPU has to generate the same 12 bindings again for the next frame. Why not cache those bindings and give developers a command that points to a cache so the same bindings can be reused?

In Section 3 we talked about queuing. When a call is made, the game believes the API immediately executes that call. That is not the case however. The commands are put in a queue where everything is deferred and executed at a later time by the GPU. So if you make a change to one of those 12 bindings we talked about earlier, the driver copies all 12 bindings to a new location, edits the copy, then tells the GPU to start using the copied bindings. Usually many of the 12 bindings have static values, with just a few dynamic ones requiring updates. When the game wants to make a partial change to those bindings, it copies all 12, which is an excessive CPU cost for minimal change.
4.2 Descriptors

What is a descriptor? Simply put, it is a piece of data that defines resource parameters. Essentially it is what is behind the D3D 11 view object. There is no operating system lifetime management. It is just opaque data in GPU memory. It contains type and format information, mip count for textures, and a pointer to the pixel data. Descriptors are the center of the new resource binding model.

![Descriptor Diagram](image_url)

*Figure 7: The D3D 12 descriptor, a small piece of data defining a resource parameter.*

4.3 Heaps

When a view is set in D3D 11, it copies the descriptor to the current location in GPU memory that descriptors are being read from. If you set a new view in the same location, D3D 11 will copy the descriptors to a new memory location and tell the GPU in the next draw command to read from that new location. D3D 12 gives explicit control to the game or application when descriptors are created, copied, etc.

![Descriptor Heaps Diagram](image_url)

*Figure 8: The descriptor heap is an array of descriptors.*
Heaps (Figure 8) are just a giant array of descriptors. You can reuse descriptors from previous draws or frames. You can also stream new ones as needed. The layout is owned by the game, and there is little overhead to manipulate the heap. The heap size depends on the GPU architecture. Older and low power GPUs size may be limited to 65k, while higher end GPUs are memory limited. Exceeding the heap is a possibility for lower power GPUs. So D3D 12 allows for multiple heaps and switching from one descriptor heap to the next. However, switching between heaps in some GPUs causes a flush, so it is a feature best used sparingly.

Now, how do we associate shader code with specific descriptors or sets of descriptors? The answer? Tables.

4.4 Tables

Tables are a start index and size in the heap. They are context points, but they are not API objects. You can have one or more tables per shader stage as required. For example, the vertex shader for a draw call can have a table pointing to the descriptors in offset 20 through 32 in the heap. When work begins on the next draw, the offset may change to 32 through 40.

Using current hardware D3D 12 can handle multiple tables per shader stage in the PSO. You could have a table with just the things that are changing frequently call over call, while a second table contains the things that are static from call to call, frame to frame. Doing this avoids copying all the descriptors from one call to the next. However older GPUs are limited to one table per shader stage. Multiple tables are only possible on current and future hardware.

*Figure 9: Descriptor tables are simply a start index and size in the descriptor heap.*
4.5 Bindless and Efficient

Descriptor heaps and tables are the D3D team's take on bindless rendering, except that scales across PC hardware. D3D 12 supports everything from low-end SoCs to high-end discrete cards. This unified approach provides game developers many binding flow possibilities. Additionally the new model includes multiple frequency updates. Allowing for a cached table of static bindings for reuse and a dynamic table with data that is changing with each draw, thus eliminating the need to copy all the bindings with each new draw.

4.6 Render Context Review

Figure 10 shows the render context with the D3D 12 changes discussed so far. It also shows the new PSO and the removal of the Gets, yet it still has the D3D 11 explicit blind points.

Let's remove the last of the D3D 11 render context and include the descriptor tables and heaps. Now we have a table per shader stage, or multiple tables as shown by the pixel shader.
The fine grain state objects are gone, replaced with a Pipeline State Object. Hazard tracking and state mirroring is removed. Explicit bind points have been exchanged for application/game-managed memory objects. CPU efficiency is greater with the reduced overhead and from removing the control flow and logic in both the API and driver.
5.0 Bundles

We finished covering the new Render Context in D3D 12 and saw how D3D 12 gives control back to the game, getting it “closer to the metal.” Yet there is more D3D 12 does to remove or streamline API churn. There is still overhead in the API that is lowering performance, and there are additional ways to efficiently use the CPU. What about command sequences? How many repeated sequences are there and how can they be made more efficient?

5.1 Redundant Render Commands

Examining render commands frame-over-frame, the Microsoft D3D team found that only 5-10% of the command sequences are deleted or added. The rest are reused frame-over-frame. So the CPU is repeating the same command sequences 90-95% of the time!

How can this be more efficient? Why has D3D not tried this until now? At BUILD 2014, Max McMullen said, “It’s very hard to build a way to record commands that is both conformant and reliable. So it behaves the same way, across multiple different GPUs on multiple different drivers, and simultaneously with that, make it performant.” The game needs to count on any recorded command sequences being executed as quickly as individual commands. What changed? D3D changed. With the new PSO, descriptor heaps, and tables, the state required to record and playback commands is greatly simplified.
5.2 What are Bundles?

Bundles are a small list of commands that are recorded once, yet they can be reused across frames or in a single frame—there are no restrictions on reuse. Bundles can be created on any thread and used an unlimited number of times. The bundles are not tied to the PSO state, meaning the PSO can update the descriptor table, then when the bundle is run again with the different bindings, the game gets a different result. Like a formula in an Excel* spreadsheet, the math is always the same, but the result is based on the source data. Certain restrictions exist to ensure the driver can implement bundles efficiently, one being that no command changes the render target. But that still leaves many commands that can be recorded and played back.

Figure 12: Bundles are oft-repeated commands that are recorded and played back as necessary.

The left side of Figure 12 is a rendering context sample, a series of commands generated by the CPU and passed to the GPU for execution. On the right are two bundles containing a command sequence recorded on different threads for reuse. As the GPU runs the commands, it eventually reaches an execute bundle command. It then plays back the recorded bundle. It returns to the command sequence when complete, continues, and finds another bundle execute command. The second bundle is then read and played back before continuing on.
5.3 Code Efficiency

We have gone through the control flow in the GPU. Now we will see how bundles simplify the code.

Example code without bundles

Here we have a setup stage that sets the pipeline state and descriptor tables. Next we have two object draws. Both use the same command sequence, only the constants are different. This is typical D3D 11 and older code.

```c
// Setup
pContext->SetPipelineState(pPSO);
pContext->SetRenderTargetViewTable(0, 1, FALSE, 0);
pContext->SetVertexBufferTable(0, 1);
pContext->IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
```

*Figure 14: Setup stage in typical D3D 11 code*

```c
// Draw 1
pContext->SetConstantBufferViewTable(D3D12_SHADER_STAGE_PIXEL, 0, 1);
pContext->SetShaderResourceViewTable(D3D12_SHADER_STAGE_PIXEL, 0, 1);
pContext->DrawInstanced(6, 1, 0, 0);
pContext->SetShaderResourceViewTable(D3D12_SHADER_STAGE_PIXEL, 1, 1);
pContext->DrawInstanced(6, 1, 6, 0);
```

*Figure 13: Draw in typical D3D 11 code*

```c
// Draw 2
pContext->SetConstantBufferViewTable(D3D12_SHADER_STAGE_PIXEL, 1, 1);
pContext->SetShaderResourceViewTable(D3D12_SHADER_STAGE_PIXEL, 0, 1);
pContext->DrawInstanced(6, 1, 0, 0);
pContext->SetShaderResourceViewTable(D3D12_SHADER_STAGE_PIXEL, 1, 1);
pContext->DrawInstanced(6, 1, 6, 0);
```

*Figure 16: Draw in typical D3D 11 code*
**Example code with bundles**

Now let's see the same command sequence with bundles in D3D 12. The first call below creates a bundle. Again, this can happen on any thread. In the next stage the command sequence is recorded. These are the same commands we saw in the prior example.

```c
// Create bundle
pDevice->CreateCommandList(D3D12_COMMAND_LIST_TYPE_BUNDLE, pBundleAllocator, pPSO, pDescriptorHeap, &pBundle);
```

Figure 17: Bundle creation code sample

```c
// Record commands
pBundle->IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
pBundle->SetShaderResourceViewTable(D3D12_SHADER_STAGE_PIXEL, 0, 1);
pBundle->DrawInstanced(6, 1, 0, 0);
pBundle->SetShaderResourceViewTable(D3D12_SHADER_STAGE_PIXEL, 1, 1);
pBundle->DrawInstanced(6, 1, 6, 0);
pBundle->Close();
```

Figure 18: Bundle recording code sample

The code samples in Figures 17 and 18 accomplish the same thing as the non-bundle code in Figures 14-16. They show how bundles dramatically reduce the number of calls needed to perform the same task. The GPU is still executing the same commands and getting the same result, just more efficiently.
6.0 Command Lists

Through bundles, the PSO, descriptor heaps, and tables you have seen how D3D 12 improves CPU efficiency and gives more control to developers. The PSO and descriptor model allow for bundles, which in turn are used for common and repeated commands. This simpler "closer to the metal" approach reduces overhead and allows more efficient use of CPUs for "Console API efficiency and performance." Previously we discussed that PC games have thread 0 do most, if not all, the work while the other threads handle other OS or system tasks. Efficient use of multiple cores or threads in PC gaming is tough. Often, the work required to make a game multithreaded is expensive both in manpower and resources. The D3D development team wants to change that with D3D 12.

6.1 Command Creation Parallelism

As mentioned several times in this paper, deferred command execution is when it seems as if each command is executed immediately, but the commands are actually queued and run at a later time. This function remains in D3D 12, but is transparent to the game. There is no immediate context because everything is deferred. Threads can generate a command in parallel to complete a list of commands that are fed into an API object called the command queue. The GPU will not execute the commands until they are submitted via the command queue. The queue is the ordering of the commands and the command list is the recording of said commands. How are command lists different from bundles? Command lists are designed and optimized, so multiple threads can simultaneously generate commands. Command lists are used once, then deleted from memory and a new list is recorded in its place. Bundles are designed for multiple uses of commonly reused rendering commands within a frame or in multiple frames.

In D3D 11 the team attempted command parallelism, also known as deferred context. However, it did not achieve the D3D team's performance goals due to the required overhead. Further analysis showed many places with a lot of serial overhead, resulting in poor scaling across the CPU cores. Some of the serial overhead was removed in D3D 12 with the CPU efficiency designs reviewed in Sections 2-5.
6.2 Lists and the Queue

Imagine two threads generating a list of rendering commands. One sequence should run before the other. If there are hazards, one thread uses a resource as a texture, but the other thread uses that same resource as a render target. The driver needs to look at the resource usage at render time and resolve the hazards, ensuring coherent data. Hazard tracking is one area of serialized overhead in D3D 11. With D3D 12 the game, not the driver, is responsible for hazard tracking.

D3D 11 allows for a number of deferred contexts, but they come with a cost. The driver tracks the state per resource. So as you start recording commands for the deferred context, the driver needs to allocate memory to track the state of every resource used. This memory is retained while the deferred context is generated. When done, the driver has to delete all the tracking objects from memory. This results in unnecessary overhead. The game declares the maximum number of command lists that can be generated in parallel at the API level. Then the driver arranges and allocates all the tracking objects up front in a coherent single piece of memory.

It is common to use dynamic buffers (context, vertex, etc.) in D3D 11, but there are multiple instances of memory tracking discarded buffers behind the scenes. For instance, there may be two command lists generated in parallel and MapDiscard is called. Once the list is submitted, the driver must patch into the second command list to correct discarded buffer information. Like the hazard example earlier, this requires a bit of overhead. D3D 12 has given the renaming control to the game; the dynamic buffer is gone. Instead the game has fine grain control. It builds its own allocators and can subdivide the buffer as needed. Then the commands can point to the explicit point in memory.

As discussed in Section 3.1, the runtime and driver track the resource lifetime in D3D 11. This requires a lot of resource counting and tracking and everything must be resolved at submit time. The game has resource lifetime and hazard control in D3D 12, removing the serial overhead for more CPU efficiency. Parallel command generation is more efficient in D3D 12 after optimizing these four areas, allowing for improved CPU parallelism. Additionally the D3D development team is building a new driver model, WDDM 2.0, with plans for further optimizations to reduce the command list submission cost.
6.3 Command Queue Flow

Figure 19: The Command Queue, with two command lists generated in parallel and two bundles of repeated commands.

Figure 19 shows the bundle diagram from Section 5.2, but it is multithreaded. The Command Queue on the left is the sequence of events submitted to the GPU. Two command lists are in the middle, and on the right are two bundles recorded before the scenario began. Starting with the command lists, which are generated in parallel for different parts of the scene, Command List 1 completes recording, is submitted to the command queue, and the GPU starts executing it. In parallel, the command queue control flow starts and Command List 2 records on Thread 2. While the GPU executes Command List 1, Thread 2 completes generating Command List 2 and submits it to the command queue. When the command queue completes executing Command List 1, it moves to Command List 2 in a serial order. The command queue is the serial order the GPU needs to execute commands. Although Command List 2 was generated and submitted to the command queue before the GPU finished executing Command List 1, it was not executed until execution of Command List 1 was complete. D3D 12 offers more efficient parallelism across the entire process.
7.0 Dynamic Heaps

As discussed earlier, the game controls resource renaming in order to make command generation parallel. Additionally resource renaming was simplified in D3D 12. D3D 11 has typed buffers: vertex, constant, and index buffers. Game developers requested the ability to use that reserved memory however they wish. And the D3D team complied. D3D 12 buffers are no longer typed. The buffer is just a chunk of memory that the game allocates as needed in the necessary size for the frame (or multiple frames). A heap allocator can even be used and subdivided as necessary, creating a more efficient process. D3D 12 also has a standard alignment. GPUs will be able to read the data as long as the game uses the standard alignment. The more we standardize, the easier it is to create content that works well across the many variations of CPUs, GPUs, and other hardware. The memory is also persistently mapped, so the CPU always knows the address. Allowing for more CPU parallelism as you can have a thread point the CPU to that memory, and then have the CPU determine what data is required for the frame.

**Figure 20:** Allocation of buffers vs suballocation.

The top part of Figure 20 is the D3D 11 style with typed buffers. Below that is the new D3D 12 model with a heap controlled by the game. A single piece of persistent memory is present instead of each typed buffer being in a different memory location. Also the buffer sizes are adjusted by the game based on the rendering needs of the current or even next several frames.
8.0 CPU Parallelism

It is time to bring everything together and illustrate how D3D 12’s new features create a truly multi-threaded game on the PC. D3D 12 allows for several parallel tasks. Command lists and bundles provide parallel command generation and execution. Bundles record repeated commands and run them in multiple command lists several times, within a frame or across multiple frames. Command lists can be generated across multiple threads then fed to the command queue for GPU execution. Finally, persistently mapped buffers generate dynamic data in parallel. Both D3D 12 and WDDM 2.0 are designed for parallelism. D3D 12 removes the constraints of past D3D versions, allowing the developer to parallelize their game or engine in whatever way makes sense for them.

D3D11 Profiling

The diagram in figure 21 shows a typical game workload on D3D 11. The application logic, D3D runtime, UMD, DXGKernel, KMD, and present usage work across a CPU with four threads. Thread 0 does most the heavy lifting. Threads 1-3 are not really used except for application logic and the D3D 11 runtime generating rendering commands. The User Mode Driver isn’t even generating commands on these threads due to the D3D 11 design.
Now let's look at the same workload but with D3D 12 (Figure 22). Again, the application logic, D3D runtime, UMD, DXGKernel, KMD, and Present usage work across a CPU with four threads. However, the work is evenly split across all threads with D3D 12 optimizations. Thanks to true command generation, the D3D runtime runs in parallel. The kernel overhead is drastically reduced with the kernel optimizations in WDDM 2.0. The UMD works on all the threads, not just Thread 0, showing true command generation parallelism. Lastly, bundles replace the redundant state change logic of D3D 11 and reduce the application logic time.
D3D11 v D3D12 numbers

Figure 23: Side-by-side comparison of D3D 11 vs D3D 12 parallelism.

Figure 23 shows a side-by-side comparison of both versions. With true parallelism we see a relatively even CPU usage between Thread 0 and Threads 1-3. Threads 1-3 do more work so the “GFX only” shows an increase. Moreover with the reduced workload on Thread 0 and the new runtime and driver efficiencies, the overall CPU usage is reduced by ~50%. Looking at the application plus GFX, the split across the threads is more even and CPU usage is reduced by ~32%.
9.0 Summary

D3D 12 offers greater CPU efficiency with a less granular PSO. Instead of the ability to set and read each individual state, developers now have a single point, reducing or entirely removing the hardware mismatch overhead. The application sets the PSO while the driver takes the API commands and translates them to GPU code. The resource binding's new model removes the churn caused by the once necessary control flow logic.

With heaps, tables, and bundles D3D 12 offers both greater CPU efficiency and scalability. Explicit bind points are gone in favor of application/game-managed memory objects. Frequent commands can be recorded and played multiple times a frame or in many frames through bundles. Command lists and the command queue allow for parallel command list creation across multiple CPU threads. Now most, if not all, of the work is evenly split across all threads in the CPU, unlocking the full potential and power of 4th and 5th generation Intel® Core™ processors.

Direct3D 12 is a leap forward in PC gaming technology. Game developers can get “closer to the metal” with a thinner API and a driver with fewer layers. This increases efficiency and performance. Through collaboration, the D3D development team created a new API and driver model that favors developer control, which allows them to create games closer to their visions with terrific graphics and performance to match.
References and Related Links

- Direct3D® 12 API Preview
- Microsoft DirectX® 12 Blog
- Intel® Core™ M Processors
- 4th Generation Intel® Core™ Processors
- Intel® Atom™ Processors
- Intel® Processor Graphics Developer Guides

Relevant Intel Links
Corporate Brand Identity http://intelbrandcenter.tagworldwide.com/frames.cfm
Intel® product names http://www.intel.com/products/processor_number/

Notices and Disclaimers
See: http://legal.intel.com/Marketing/notices+and+disclaimers.htm

About the Author

Michael Coppock specializes in PC game performance and graphics and has been with Intel since 1994. He helps game companies get the most out of Intel GPUs and CPUs. Focusing on both hardware and software, he has worked on many Intel products, all the way back to the 486DX4 Overdrive processor.

Notices

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL PRODUCTS. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

UNLESS OTHERWISE AGREED IN WRITING BY INTEL, THE INTEL PRODUCTS ARE NOT DESIGNED NOR INTENDED FOR ANY APPLICATION IN WHICH THE FAILURE OF THE INTEL PRODUCT COULD CREATE A SITUATION WHERE PERSONAL INJURY OR DEATH MAY OCCUR.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

The products described in this document may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata
are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or go to: http://www.intel.com/design/literature.htm

Intel, the Intel logo, Intel Atom, and Intel Core are trademarks of Intel Corporation in the U.S. and other countries.

*Other names and brands may be claimed as the property of others

Copyright© 2014 Intel Corporation. All rights reserved.

**Optimization Notice**

<table>
<thead>
<tr>
<th>Optimization Notice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2®, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.</td>
</tr>
</tbody>
</table>

**NOTE:** additional notices and disclaimers may be needed, depending on the content of the collateral. Generally, they can be found in the following areas, and must be appended to the Notices section of the paper.

- **General Notices:** Notices placed in all materials distributed or released by Intel
- **Benchmarking and Performance Disclaimers:** Disclaimers for Intel materials that use benchmarks or make performance claims.
- **Technical Collateral Disclaimers:** Disclaimers that should be included in Intel technical materials that describe the form, fit or function of Intel products.
- **Technology Notices:** Notices for Intel materials when the benefits or features of a technology or program are described.

Note for technology disclaimers - if every product being discussed (e.g., ACER ULV) has the particular technology/feature, then you can remove the requirements statement in the disclaimer. If you have multiple technical disclaimers, you can consolidate the “your performance may vary” statements and only put in a single “your mileage may vary”.

Notice revision #20110804