MASTER'S THESIS

Synchronization problems in modern graphics APIs

de Bruijn, D.G.A.

Award date:
2022

Link to publication

Synchronization problems in modern graphics APIs

Master Thesis

D.G.A. de Bruijn

Date: 30/01/2022

Open Universiteit
www.ou.nl

SYNCHRONIZATION PROBLEMS IN MODERN GRAPHICS APIs

MASTER THESIS

by

D.G.A. de Bruijn

in partial fulfillment of the requirements for the degree of

Master of Science

in Software Engineering

at the Open University of the Netherlands, Faculty of Management, Science & Technology
Master's Programme in Software Engineering

Date:	30/01/2022	
Course code:	IM9906	
Course name:	SE Graduation Assignment	
Committee:	Dr. S. T. Q. Jongmans (chairman),	Open University of the Netherlands
	Dr. C. P. T. de Gouw (supervisor),	Open University of the Netherlands

Contents

LIST OF FIGURES

List of Tables

Summary

Around the introduction of Windows 10 in 2015, a new generation of graphics APIs emerged, introducing a new programming paradigm. This new generation is going for a more low-level approach of graphics programming, where tasks such as resource management and synchronization are no longer implicitly handled by the driver and runtime, abstracted away behind the API. Instead the API explicitly opens up tools for the developer to manage these things.

The idea behind this approach is that the application developer can manage resources and synchronization more efficiently than the API can, and the old approach of inferring the logic via the runtime and the API implicitly is actually bottlenecking the system, leading to a lot more CPU overhead, and poor use of multiple cores/threads.

The result is that these new APIs are more difficult to use, as the developer has to manage a lot more state. In this thesis we look at how DirectX 12 works, and how we can integrate existing tools such as Microsoft's DirectX 12 Residency Starter library and the Windows threadpool to create a framework that simplifies these tasks, without trading in too many of the advantages of the new API.

A set of Proxy objects is presented to automate the work required for using the DirectX Residency Start library. A task framework similar to the .NET Task Parallel Library in C# is presented for use with C++, which introduces the novelty of supporting both CPU and GPU tasks, and allows you to freely combine them with synchronization.

The complexity of code written with the new framework will be evaluated by comparing before and after code snippets and calculating code metrics. The DirectX 12 debug layer will be used to detect if resource management actually makes resources resident at the right time.

The CPU performance is benchmarked by measuring the CPU time used for rendering frames, and comparing these measurements between the different versions of the code.

The results show that adding residency management to an existing (non-managed) DirectX 12 application takes only minimal adjustments to the code. The performance hit was measured at about 0.12 ms of additional time per frame (less than 1% of the frame time at 60 fps).

The results also show that using the task framework to distribute a render workload over multiple threads via a threadpool, makes the test application considerably less complicated (36 less lines of code) than the original version which used regular threads and synchronization via events. The performance of the code was unaffected, and could even be improved trivially.

1 Introduction

This thesis covers synchronization problems that exist when developing applications for modern graphics processors, using contemporary graphics APIs. In order to describe the problems that this research will focus on, it is important to first define the context of the hardware and software in which these problems occur, namely the GPUs and the graphics APIs, before getting into the central research question, the problems in detail, the research context, the approach taken, and the results achieved.

GPUs and graphics APIs were mainly designed to solve a single problem: triangle rasterization. Rasterization is the conversion of a shape to a raster of pixels, so that it can be displayed on a pixel-oriented screen, or stored as a bitmap image. The triangle in particular is a very simple and unambiguous shape, which lends itself well for efficient rasterization in hardware. All three vertices of a triangle lie on the same plane by definition. And the triangle consists of three edges, which can be interpolated linearly. This makes triangle rasterization an “embarrassingly parallel” problem, which can be accelerated in hardware with relatively simple circuitry.

GPUs have evolved from simple ‘hardwired’ acceleration circuits to fully programmable processing units. It is however important to understand that they are quite different from CPUs. The programming model for CPUs is very low-level. CPUs are assumed to have a Von Neumann architecture, and native code is generated directly by the compiler, and placed in memory, as a special form of data. Memory is accessed directly at the byte or word-level. While there are different instruction set architectures for CPUs, they all follow the same basic principles of operation. Concepts such as cores and threads are reasonably standardized.

GPUs are accessed at a higher level of abstraction. There is no standardized architecture or instruction set at the hardware level, unlike for example x86 and ARM in the CPU world. Even concepts of cores and threads are not very clearly defined, and are mainly used as a marketing tool by the vendors. From a programming perspective, the GPU is seen as a ‘virtual machine’ that is defined by the API programming model, such as Direct3D, OpenGL or Vulkan. A driver is responsible for translating the high-level API calls and program code to native code for the GPU. This has the advantage that software is not tied directly to a specific hardware model. This allows rapid development of new GPU technology without breaking software compatibility. As long as the new GPU has an appropriate driver, it can run existing software without modification.

GPUs work in a fire-and-forget fashion: an API call simply places a command inside a driver command queue, and returns as quickly as possible, so that the main CPU can continue executing the main program. The driver performs the translations in the background, and sends the commands to the GPU. The GPU will signal completion of a command with an interrupt, which will trigger the driver to retire the current command and send the next command to the GPU. This is generally transparent to the application layer. The behaviour can be compared to out-of-order-execution on a CPU: the commands appear to be executed in-order to the application, even though the underlying implementation may execute instructions in a different order than they appear in the actual code. In the case of graphics, most API calls appear to complete 'instantly', even though the actual GPU may be an entire frame behind the application code or more.

This programming model is both a blessing and a curse. An advantage is that a lot of the low-level complexity can be hidden in the driver layer. The APIs generally do not evolve as quickly as the GPUs do. Aside from the advantage mentioned earlier that the translation layer can be implemented for newer GPUs, this can also be a disadvantage. It means that over time, graphics APIs and their programming model may become a poor fit to the underlying GPU architecture, and the driver has to perform very complex translations from API to native hardware, which leads to poor efficiency.

Over time, GPUs have become so fast, that they are generally bottlenecked by the CPU: the CPU simply cannot feed commands to the GPU as fast as the GPU can process them, leaving untapped performance on the table. This is mainly a result of the fact that the programming model assumed a purely sequential mode of operation: each API call must be processed in the order in which it is sent to the driver command queue. This means that the driver can generally only use one CPU core/thread to process the command queue. It is very difficult to extract parallelism at the command queue level.

Recently, a revolution has occurred in the world of graphics APIs. In order to reduce the bottleneck and make better use of multithreading on the CPU side, graphics APIs have chosen to reduce the level of abstraction. Where historically the driver was responsible for memory management and synchronization of access to GPU resources, this is now left to the application. In other words: the driver is no longer holding hands, and if the application does not perform these tasks correctly, the rendering process will yield undefined results (such as accessing a resource that is not currently resident in memory, or reading from a resource before a previous write operation has completed).

There are three major new APIs, which all use the same new low-level philosophy. There is the multi-platform Vulkan [18], a standard maintained by the Khronos Group, who also maintain the OpenGL standard. Vulkan is not seen as a replacement of OpenGL, but as a complement to the existing OpenGL and OpenGL ES APIs [31]. Microsoft offers DirectX 12 on their Windows-based platforms, which also is seen as a complement of DirectX 11, rather than a replacement. DirectX 11 will continue to be supported and updated [36].

Apple offers Metal on their platforms, which is their first graphics API. Previously, Apple used OpenGL and OpenGL ES on their platforms, which will be supported until further notice.

The philosophy is that the application developer has more knowledge of the exact access patterns and opportunities for parallelism than a generic driver and API would have. So by putting the responsibility for memory management and synchronization in the hands of the application developer, the graphics API gives the developer the tools to achieve better efficiency and more performance from the same hardware [31].

Command queues are also explicitly available on the application side. It is now the application's responsibility to build command lists, and send them to a queue, rather than sending commands implicitly via individual API calls. There are two advantages here. The first is that an application can build multiple command lists in parallel, by using multiple threads. The second is that GPUs can expose multiple hardware queues to the application, so that multiple command lists can be executing at the same time on the GPU asynchronously. DirectX 12 has 3 different types of queues:

- Graphics queues (known as 'direct' queues), for standard graphics operations using the triangle rasterization pipeline
- Compute queues, for operations that do not use rasterization
- Copy queues, for data transfers

The copy queue is the most basic queue type. Compute queues are a superset of copy queues, as they can accept both copy and compute commands. Graphics/direct queues are a superset of compute queues, and will accept all possible commands. Figure 1.1 is a

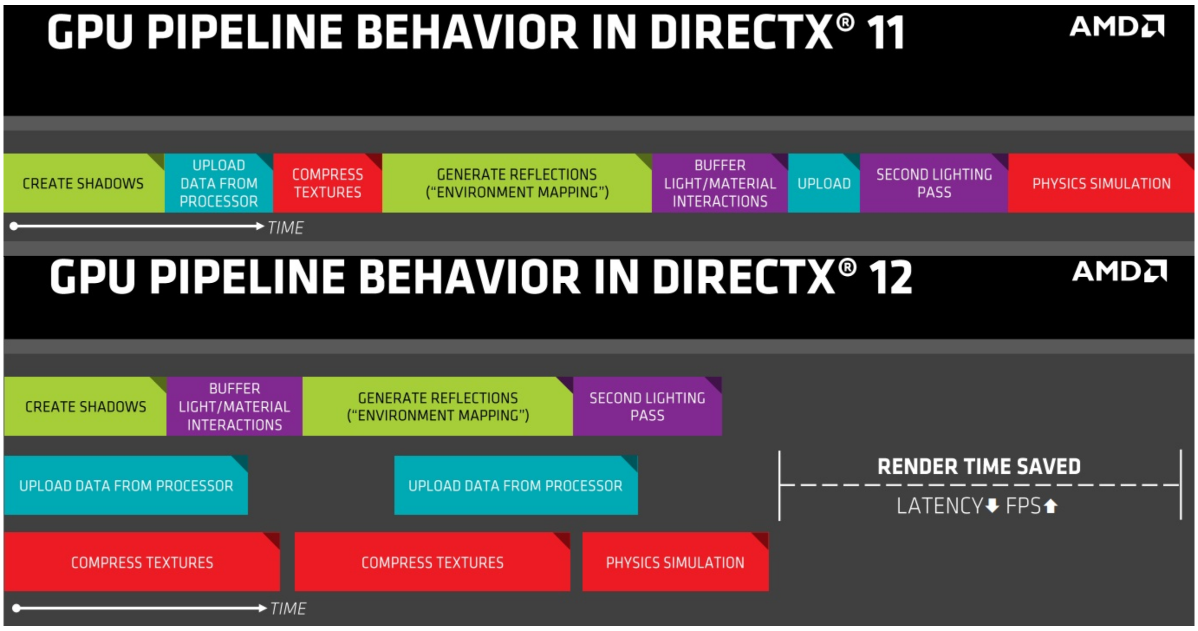

GPU PIPELINE BEHAVIOR IN DIRECTX® 11
AMD
CREATE SHADOWS
UPLOAD DATA FROM PROCESSOR
COMPRESS TEXTURES
GENERATE REFLECTIONS ("ENVIRONMENT MAPPING")
BUFFER LIGHT/MATERIAL INTERACTIONS
UPLOAD
SECOND LIGHTING PASS
PHYSICS SIMULATION
TIME
GPU PIPELINE BEHAVIOR IN DIRECTX® 12
AMD
CREATE SHADOWS
BUFFER LIGHT/MATERIAL INTERACTIONS
GENERATE REFLECTIONS ("ENVIRONMENT MAPPING")
SECOND LIGHTING PASS
UPLOAD DATA FROM PROCESSOR
UPLOAD DATA FROM PROCESSOR
RENDER TIME SAVED
LATENCY⬇ FPS⬆
COMPRESS TEXTURES
COMPRESS TEXTURES
PHYSICS SIMULATION
TIME

Figure 1.1: AMD slide demonstrating the parallel hardware command queues in DirectX 12.

slide from an AMD presentation [2] which gives a good overview of the difference between DirectX 11 and DirectX 12. It gives an abstract view of how all GPU tasks in a game are rendered strictly in a serial fashion in DirectX 11. Then it shows the same tasks distributed over the three types of queues (graphics, compute and copy), where certain tasks can run in parallel, reducing the total render time of a single frame. In this simple example, a potential data race can already be spotted: data is uploaded in parallel with rendering operations. If any rendering operation requires data from an upload task, it will need to be synchronized with the upload task to make sure the upload has finished before the rendering operation starts.

To illustrate this with some basic code examples, in DirectX 11 the API calls will implicitly place commands in an internal buffer:

```
// Send drawing commands.
m_immediateContext->OMSetRenderTargets(1, &renderTargetView, &depthStencilView);

m_immediateContext->IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
m_immediateContext->IASetVertexBuffers(0, 1, &m_vertexBuffers);
m_immediateContext->IASetIndexBuffer(&m_indexBuffer, DXGI_FORMAT_R16_UINT, 0);
m_immediateContext->DrawIndexedInstanced(36, 1, 0, 0, 0);
```

In DirectX 12, the same commands are not direct API calls, but instead, are called upon a command list object, to add them to the list. They are not executed until the list is sent to a queue:

```
// Indicate this resource will be in use as a render target.
CD3DX12_RESOURCE_BARRIER renderTargetResourceBarrier =
CD3DX12_RESOURCE_BARRIER::Transition(m_deviceResources->GetRenderTarget(),
D3D12_RESOURCE_STATE_PRESENT, D3D12_RESOURCE_STATE_RENDER_TARGET);
m_commandList->ResourceBarrier(1, &renderTargetResourceBarrier);

// Record drawing commands.
m_commandList->OMSetRenderTargets(1, &renderTargetView, false, &depthStencilView);

m_commandList->IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
m_commandList->IASetVertexBuffers(0, 1, &m_vertexBufferView);
m_commandList->IASetIndexBuffer(&m_indexBufferView);
m_commandList->DrawIndexedInstanced(36, 1, 0, 0, 0);

// Indicate that the render target will now be used to present when the command list is
 done executing.
CD3DX12_RESOURCE_BARRIER presentResourceBarrier =
CD3DX12_RESOURCE_BARRIER::Transition(m_deviceResources->GetRenderTarget(),
D3D12_RESOURCE_STATE_RENDER_TARGET, D3D12_RESOURCE_STATE_PRESENT);
m_commandList->ResourceBarrier(1, &presentResourceBarrier);

m_commandList->Close();

// Execute the command list.
ID3D12CommandList* ppCommandLists[] = { m_commandList.Get() };
m_deviceResources->GetCommandQueue()->ExecuteCommandLists(_countof(ppCommandLists),
ppCommandLists);
```

This difference in command flow is also shown in Figure 1.2 for DirectX 11 and Figure 1.3 for DirectX 12.

Note also that there are these new synchronization objects called ‘ResourceBarrier’ that need to be added between certain operations, which were implicit in the DirectX 11 programming model. The synchronization was done automatically by the underlying implementation.

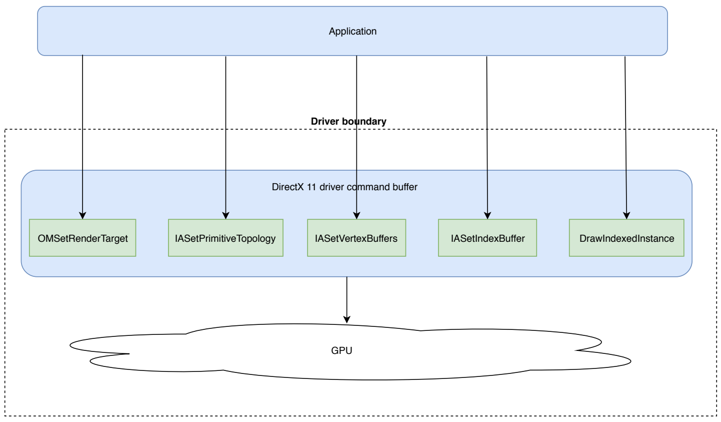


Figure 1.2: Command flow in DirectX 11.

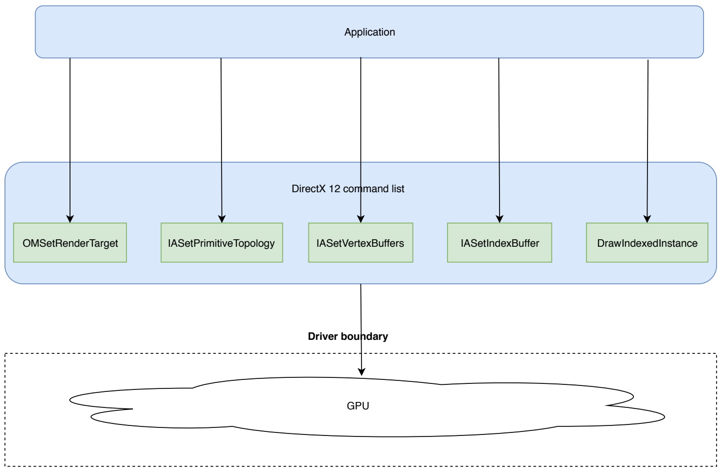


Figure 1.3: Command flow in DirectX 12.

The obvious downside is that it adds a lot of extra complexity to the application layer. The application developer now has to explicitly keep track of his resource usage, and decide where, when and how to add synchronization points in the application code [31]. The goal of this research is to find easy-to-use and efficient ways to implement this synchronization, by attempting to apply ideas from synchronization solutions in other areas of computing, mainly aimed at generic CPU processing.

The current literature on modern graphics APIs such as DirectX 12 and Vulkan tends to focus on very basic functionality. These are often examples that are simple enough to be encoded in a single command list, using hardcoded synchronization points. This is not representative for real-world use cases. The other extreme is using game engine middleware, such as UnrealEngine, CryEngine, FrostBite or Unity. These game engines bring the abstraction level up, which makes writing a graphics application more of a scripting exercise. How the synchronization is actually handled internally is not documented. There appears to be no middle-ground between these extremes so far. What makes this an even more interesting area of research is that even the current state-of-the-art game engines often have trouble extracting better efficiency from the modern APIs. Many games can be run in both DirectX 11 or DirectX 12 mode, for backward compatibility (DirectX 12 is only supported on Windows 10 or newer, and in combination with recent GPUs). More often than not, DirectX 12 offers little or no extra performance [6, 15, 11, 17]. In some cases, DirectX 12 can actually be slower [33], or require more memory, due to inefficiencies in the application layer (sometimes the low-level nature of the API requires dedicated code-paths for specific hardware for the best performance) [31]. After all, DirectX 11 has had many years of optimization from highly specialized driver teams, and the synchronization and memory management implemented in drivers today is quite difficult to beat.

In order to better understand the specific problems that need to be solved in modern graphics APIs, Chapter 2 will explain the basics of graphics APIs and GPUs, and Chapter 3 will describe the problems that exist with synchronizing resource usage. Chapter 4 will pose the research questions and explain the scope, background and methodology. Chapters 5, 6 and 7 will then cover the solutions that address these three problems. Chapter 8 will position this research in the context of prior research work in related areas. Finally, Chapter 9 will discuss the results and propose suggestions for future research.

2 GPU PRIMER

2.1. Terminology and jargon

In order to fully understand the context and jargon of this research, a small introduction into the world of GPUs and graphics APIs is required. This is not an easy task, since the world of graphics is not quite like the world of regular computing in various ways. Most importantly, the world of graphics is very much driven by a commercial market. In the mid-to-late 90s, graphics accelerators were introduced onto the consumer market, to offer gamers higher resolutions, better image quality, and higher framerates. This market turned into a very successful and competitive field. As a result, the amount of research and development invested into consumer hardware would quickly surpass that of professional graphics solutions. And after a few years, professional graphics hardware was nothing more than a high-end spinoff of gamer hardware. Also, a few strong players wiped out virtually all the competition overnight, and for the past 2 decades, only NVIDIA, AMD (formerly ATi) and Intel remain as developers of consumer GPUs (with Intel only offering integrated GPUs on their CPUs, not sold separately, although a discrete DG1 GPU was offered to select OEMs in 2020, and another discrete DG2 GPU is announced for 2022). Also, it is a relatively new field, and documentation is mostly available online only.

On the software side, the APIs are also developed by only two major parties: Microsoft developing DirectX and the Khronos Group developing OpenGL and Vulkan. The Khronos Group is a non-profit organization, but Microsoft is a commercial party, and DirectX is part of their Windows and Xbox product lines.

A lot of new technologies are developed by NVIDIA, AMD, Intel or Microsoft, and used in a marketing offensive. As a result, a lot of common terminology in the graphics world is actually based on marketing material rather than purely technical literature. In various cases, multiple companies come up with similar technology at similar times, and the market decides which technology will ultimately win out and become the de-facto standard. Therefore there may sometimes be different names for the same technology, depending on the context. This especially goes for DirectX and OpenGL/Vulkan, which are technically very similar (after all, both APIs run on the exact same underlying hardware), but each have their own view on the technology, and their own jargon.

One example is the term ‘shader’. This term was originally coined by the offline renderer RenderMan, developed by Pixar, with the introduction of its RenderMan Shading Language (RSL) [35]. In that context, a ‘shader’ is a simple script/program which can be executed in specific parts of the RenderMan pipeline (such as a ‘light source shader’ or a ‘surface shader’). With the introduction of the NVIDIA GeForce3 video card, it was now possible to make the GPU execute simple programs in specific parts of its hardware pipeline as well, and so they also used the term ‘shader’ to refer to these programs. NVIDIA and Microsoft developed DirectX 8/9 and the High Level Shading Language (HLSL) to make use of this functionality. They ‘borrowed’ the idea from RenderMan, although the language was not compatible with RSL, and the shader types did not map onto the RSL types either, since consumer graphics hardware used a very different type of rendering pipeline.

The two types of shaders on the GeForce3 were ‘vertex shaders’ and ‘pixel shaders’. When OpenGL caught up with the new technology, and introduced their own Graphics Library Shader Language (GLSL), they renamed ‘pixel shaders’ to ‘fragment shaders’. This is a more technically accurate term, but the average gamer will probably not know what a fragment is, and therefore it will not be an effective term to use in marketing material. Gamers do know what pixels are (picture elements). A fragment is a single sample taken for a pixel. When using anti-aliasing, you may use more than one sample per pixel (supersampling), therefore, you will have multiple fragments per pixel. The term ‘fragment shader’ is correct, because the shader will be executed for every sample, and the resulting pixel will only be calculated after the samples are resolved for the final anti-aliased image. This example demonstrates how marketing may dominate over more correct technical terminology in the graphics world. What makes it even more confusing is that strictly the software is a ‘shader’ or ‘shader program’, and the hardware is a ‘shader unit’, but in practice, both tend to be referred to as ‘shaders’ for brevity, and whether this refers to the software or the hardware, has to be derived from the context.

Another example is the term ‘texture’. A texture map is an image applied to the surface of a 3D object [7]. As the name implies, the original idea was to simulate the surface detail (the regular meaning of the word ‘texture’) using an image as an approximation. Depending on the type of image used, and the way the image is applied to the surface, many properties of the surface can be mapped. In practice, the most common use of the technique became to simulate the colour detail of the surface, by using a colour map image. As a result, the word ‘texture’ became more or less synonymous with ‘colour map’, and ‘texture mapping’ became synonymous with ‘colour mapping’. It is now common to refer to any kind of image map as a texture, regardless of what properties of the surface it models. By extension, the hardware that samples image maps is described as a ‘texture unit’, and in shader programming, the instructions for sampling image maps have a ‘tex’-mnemonic.

Perhaps an even better example is the term ‘GPU’ itself. GPU stands for Graphics Processing Unit. This is a fairly recent term, and is said to be originally coined by Sony for the graphics processor in their PlayStation, in 1994. It was popularized by NVIDIA however, when they introduced their GeForce256 in 1999, and marketed it as the World’s first GPU [30]. Their definition of the term is as follows:

The technical definition of a GPU is "a single chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second."

This 'definition' is more of a product description than a technical definition. Especially the minimum performance requirement seems nonsensical from a technical point-of-view. ATi launched their own 'Visual Processing Unit' (VPU) term at about the same time, with their competing Radeon chip. That term never caught on however, and all graphics chips are referred to as GPUs since. In fact, people even refer to older chips (which do not meet the requirements of NVIDIA's definition) as GPUs. Some even go as far as referring to an entire graphics card as a GPU. The gist of NVIDIA's definition is firstly that it should be an analog to the term 'CPU', so it is referring a single chip, not an entire card. And secondly that this chip is capable enough to perform processing by itself, as in: executing a program. Consumer chips before the GeForce256 were not entirely self-sufficient, and could only render individual triangles. They relied on the CPU to pre-process the data into 2D screenspace and feed it to the rendering engine. The GeForce256 was the first consumer chip that had its own transforming and lighting engine. This meant that you would store actual 3D data into video ram, such as your geometry, your lights, and your camera and other transform matrices, and the GPU would perform all mathematics by itself. For the next frame, the CPU would only have to update some transform matrices to animate objects, and the GPU would again render the entire scene by itself. This was a big step in graphics acceleration, since the CPU now had to do a lot less. It also meant that the bus connecting the CPU to the graphics card (usually PCI or AGP in those days) was no longer a big bottleneck, because the CPU did not have to push all the geometry data over the bus every frame. The GPU was entirely self-sufficient, and a PC with a CPU and GPU was now a true multiprocessing system.

In the world of regular CPUs and software, there are already many cases where the same terminology might have different meanings depending on the context [34]. In the world of graphics, this holds even more so. This is a difficult problem to overcome. I have already covered the most important concepts above (GPUs, shaders, textures), and have attempted to explain how they are used in the context of this thesis.

In this thesis, I will use all terms in the context of Direct3D 12 [20]. So the context is specifically GPUs that support Direct3D 11 and 12, and specifically the shaders as supported by the Direct3D 11 and 12 programming models. As said, the GPU in this context is a 'virtual machine', very similar to the concept of Java. Today's GPUs have very little resemblance to the original GeForce256 GPU at the hardware level. And GPUs from different brands are not necessarily similar in design and operation either, which is very different from the CPU world, where both Intel and AMD make use of the x86 architecture, and their CPUs are binary-compatible.

At this higher level of abstraction, it is also not quite clear how a GPU executes code exactly, and in which way it exploits parallelism. In practice, most GPUs at the time of writing, compile and execute shader programs as 'scalar threads', on a Single-Instruction-Multiple-Data (SIMD) execution unit [12]. So the same shader program can be run in parallel on many vertices or pixels at the same time. Most GPUs at the time of writing have multiple of these SIMD-like execution units, where each unit has its own program counter, and therefore can run its own code. This means that technically these GPUs can be classed as Multiple-Instruction-Multiple-Data (MIMD) processors [12]. Compared to CPUs, GPUs have a simplified, less generic instruction set. They are not Turing complete, and they are not a Von Neumann-architecture. The code is not accessible by the shader units themselves.

The SIMD-part is the "embarrassingly parallel" part of GPU parallelism. The parallelism is implicit in the design of the GPU and the programming model. The programming model has various constraints to benefit and simplify parallel execution. For example, inside a shader program, you can only access the current vertex or pixel, any other data has to come from read-only resources such as textures or constant buffers. The subject of threading/synchronization problems in this thesis is at a higher abstraction level. This is the level made possible by the MIMD architecture: each MIMD-cluster can execute its own instruction stream.

This parallelism occurs at the command lists and queues mentioned in the previous chapter. Namely, the new graphics APIs allow the programmer to create multiple command queues, which in turn allows the programmer to submit multiple command lists in parallel. These lists may be executed asynchronously, distributed over clusters of MIMD processing units. This depends on how the GPU implements its command queues. Some GPUs have multiple work queues in hardware, and will actually be able to run two or more command lists concurrently, by making use of the aforementioned MIMD architecture. This means that GPU state and resource usage needs to be synchronized between these command lists, to avoid race conditions and other corruption.

2.2. THE GRAPHICS PIPELINE

Figure 2.1 shows a diagram of the pipeline taken from the Direct3D 11 documentation [25]. The high level of abstraction is indicated by the fact that all memory resources are depicted as a single pool, which can be accessed by any of the stages. In practice the API is not quite this abstract, and different stages do need to create their own specific view of a resource, but apparently this was not considered relevant for the documentation.

The common case for rendering 3D objects, constructed from meshes of triangles, as used by all major graphics APIs, is as follows:

1. All objects are stored in so-called 'object space' or 'model space', where generally the center of the object is the origin of this space. A transformation matrix known as the 'world matrix' is applied to each vertex of the object to transform the object into 'world space'.

2. Once in world space, lighting calculations at the per-vertex level are applied.

3. A 'view' (sometimes also 'camera') matrix is applied to transform each vertex from world space into view space.

4. A ‘projection’ (sometimes also called ‘perspective’) matrix is applied to transform each vertex from view space into ‘post-perspective space’. This stage is the move from a 3D world to a 2D projection of the world, ready to be rasterized to a bitmap.

5. Each triangle is sent to the rasterizer, which converts triangles from vector-based geometry to a 2D raster. This raster is effectively the collection of all pixels inside the triangle, in screen-space.

6. Lighting and texturing calculations at the per-pixel level are applied, to generate the final 2D image.

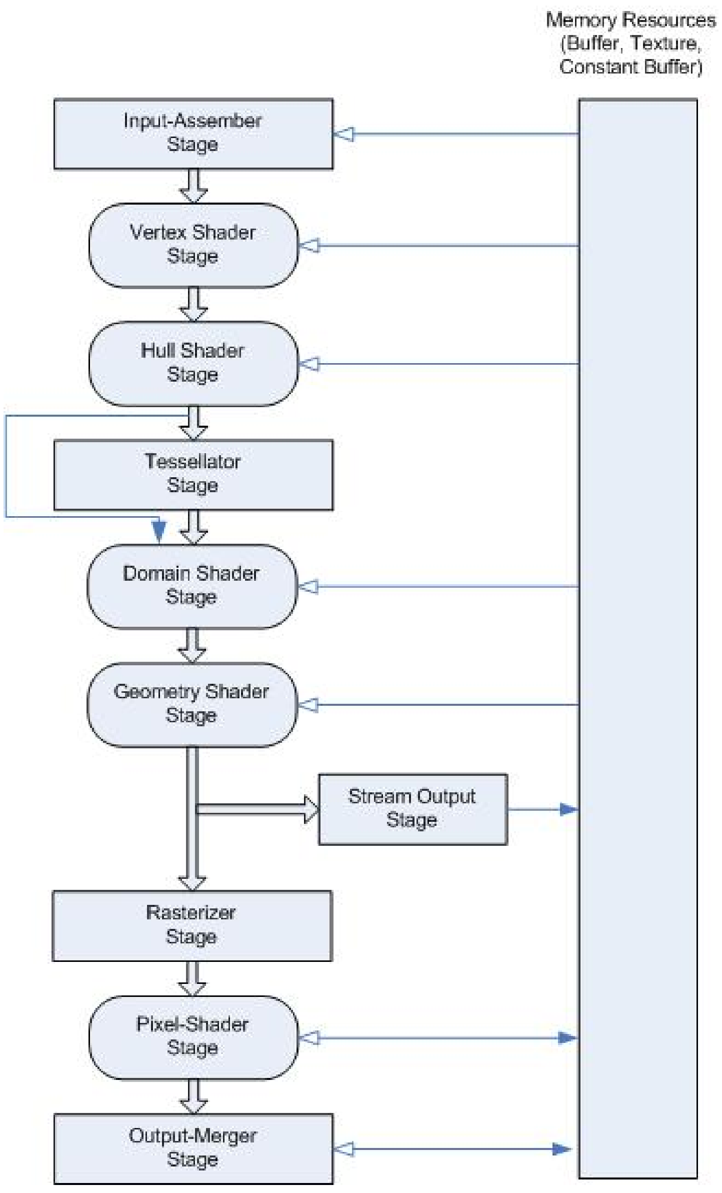


Figure 2.1: Overview of the Direct3D 11 pipeline [25].

The Direct3D pipeline has a rigid order. At the start of the pipeline, the geometry is fed into the input-assembler. This stage will fetch the geometry, which is normally stored in memory as a stream of triangles, where each triangle is represented by 3 consecutive vertices. Each vertex is then fed to the vertex shader stage. The vertex shader is responsible for the object-to-world space transformation, the per-vertex lighting calculations, the world-to-view space transformation, view-to-post-perspective space transformation.

As can be seen, every shader stage in the pipeline can read from memory resources. Only the stream output stage and the pixel shader stage can write to memory resources.

The hull shader, tessellator stage, domain shader and geometry shader are all part of dynamic tessellation, which is beyond the scope of this thesis. They are an optional part of the Direct3D pipeline.

The output of a vertex shader is a vertex with a set of properties, such as colour, texture coordinates and light intensity. These properties are referred to as 'gradients'. The rasterizer will process each triangle, so three vertices at a time. The rasterizer determines which pixels of the resulting 2D image belong to the triangle. It will also perform perspective-linear interpolation of the gradients, so that each pixel is assigned an interpolated value of each property. In other words: each property of the triangle, as stored in the vertices, is interpolated over the flat triangle surface. Each pixel, with its gradients evaluated at the position of that pixel, is then fed to a pixel shader.

The stream output stage is an optional stage which allows an application to write the processed stream of vertex data back to memory for later use. It is beyond the scope of this document.

The pixel shader performs any remaining lighting calculations per-pixel, and will apply any texture maps to each pixel. The output, which is normally a pixel colour and a per-pixel depth value, is then fed to the output-merger state.

The output-merger stage collects the output of all pixel shaders, and stores each pixel in the final 2D image.

The Direct3D 12 graphics pipeline diagram [24], shown in Figure 2.2, shows the same processing stages. However, the resources are far more detailed in the diagram, which illustrates how the programming model has changed in the new API. It is now far more important to be aware of the different resources, views and transitions between them.

In the Direct3D 12 documentation there is also a separate diagram for compute shaders, shown in Figure 2.3. In Direct3D 11, this diagram would be trivial, since compute shaders operate outside the graphics pipeline, and are only a single stage. In Direct3D 12 however, it is still relevant to depict how the compute shader interacts with samplers and resources.

2.3. THE PROGRAMMING MODEL

As mentioned earlier, the GPU is programmed as a 'virtual machine'. Just like Java or .NET, the code is actually compiled to native code at runtime. This is more or less a requirement for the Direct3D programming model: any hardware can be supported, as long as

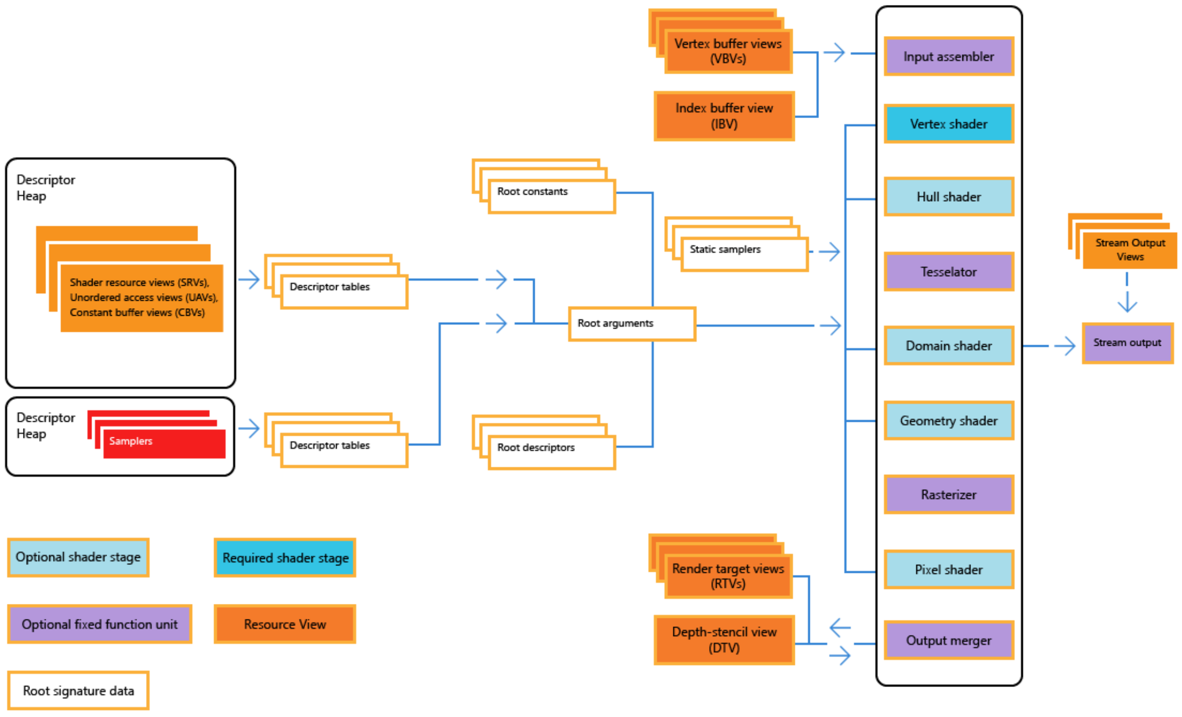


Figure 2.2: Overview of the Direct3D 12 graphics pipeline [24].

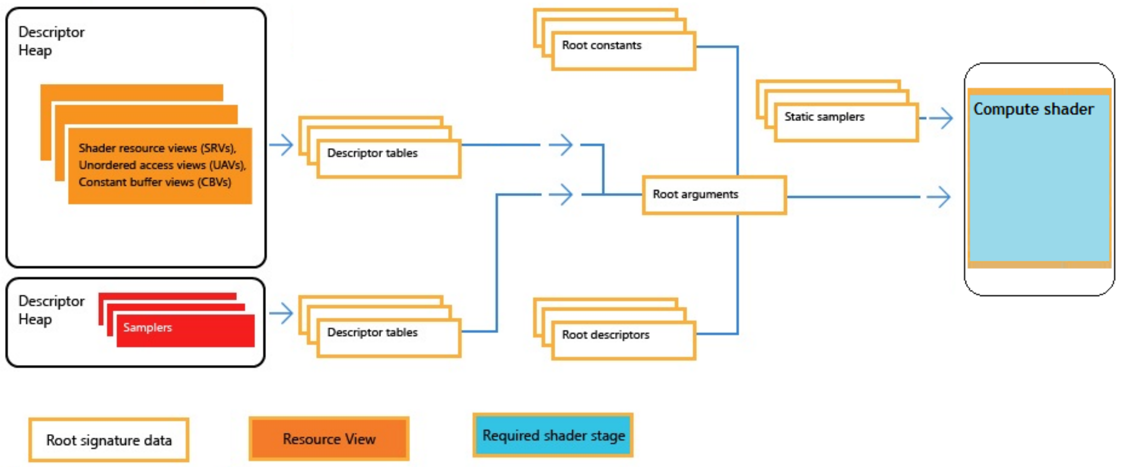


Figure 2.3: Overview of the Direct3D 12 compute pipeline [24].

the hardware vendor supplies a suitable driver. This driver contains a compiler to compile the hardware-neutral shader bytecode to the native GPU, and will also perform any other translations required from API-level to hardware-level. This amounts mainly to interpreting the command lists.

This means that most of the GPU code will be generated just-in-time. Command lists are often generated on-the-fly by the graphics engine, depending on the application state. For example, in games, what is to be drawn depends on the user input and the AI-routines that control the computer players and other logic, and will be evaluated at every frame to be rendered.

The nature of GPU code as executed by Direct3D and similar APIs means that it is difficult to perform static analysis on command lists, or to add additional syntax to the language itself. The commands are implemented as API calls in C++, not as a programming language in its own right, and they are not compiled in advance. They are generated on-the-fly by the Direct3D runtime and the underlying driver.

In the next chapter we will look at how resources are used by GPUs, and how the graphics API exposes these to the programmer.

3 Issues with resource usage in graphics APIs

Memory access on a GPU is abstracted to a higher level than on CPUs. On a CPU, buffers are addressed directly via a memory address. In most programming languages, there is a pointer datatype to store a memory address, which also forms the basis for more complex datastructures such as arrays and objects.

No direct access at the address level on a GPU is exposed by the API. Instead, the API provides a resource interface (ID3D12Resource), which can store data in various forms. Resource types include 1D, 2D and 3D textures (effectively arrays of pixel data), and a more generic buffer object. Accessing these resources is done via so-called 'samplers'. A sampler is an abstraction of the texture unit hardware. A texture unit will take floating point texture coordinates as input, to perform a 1D, 2D or 3D lookup into the resource. Instead of just performing an array lookup with integer indices, these fractional coordinates allow a sampler to perform interpolation between neighbouring elements in the resource.

Modern GPUs use processing units for floating point data in a SIMD-like configuration. This makes parallel float data the 'native' data format for a GPU. The most common form of resource data is sets of pixels with R, G, B and A elements. These are stored as 4D vectors, and samplers will return a 4D floating point vector as the result of a lookup.

In the next three sections, the main problems with resource usage and management in modern graphics APIs are explained. The first problem is making sure that resources are made resident when in use, and evicted when not in use (video memory effectively acting as a Least Recently Used (LRU) cache). The second problem is in efficiently creating and executing multiple command lists concurrently, while synchronizing execution at key positions where a specific serial order of execution is required. The third problem is in having to perform so-called 'transitions' of data from one internal format to another, depending on the type of operation that is being done on the data. This thesis will focus on finding solutions to these three problems.

3.1. MEMORY RESIDENCY

One aspect of memory handling that is different from CPUs is that GPUs do not support page-faulting. On a CPU, virtual memory can be implemented automatically by marking memory pages as inaccessible when they are paged-out. When a CPU tries to access such a memory page, it triggers an exception. The OS has installed an exception handler, and will page-in the required memory, restore access rights, and then continue execution. With GPUs, there is no such mechanism, so the application has to keep track of memory usage itself. This means that the application has to track the usages of the resource objects, and decide which resources to have resident at a given time, and to decide when it is time to page-in new resources. When the GPU tries to access a resource that is not resident, no exception will be triggered, and the memory for the resource will simply be undefined. The rendering operation will continue, and will yield undefined results. Microsoft offers a D3D12 Residency Starter Library to help programmers manage their resources automatically [28].

3.2. CONCURRENCY

A second issue stems from the aforementioned command queues: it is possible to have multiple command lists active simultaneously. This also means that it is possible for multiple command queues to read or write the same resources simultaneously.

In previous graphics APIs, all resource management was handled by the driver (and since there was only one implicit command queue, the concurrency issue between command queues did not exist). The driver would use heuristics to infer the semantics based on the current driver state and the API function called. In the new generation of graphics APIs, all these issues require manual synchronization by the application, to avoid undefined results. Likewise, drivers would be able to perform various optimizations transparently under previous APIs. For example, when a resource was in use in read-only form, and then the application wanted to use it for write-only, the driver could 'alias' the resource, by creating a new resource with the same characteristics, which is referenced under the same handle (analogous to register renaming in modern out-of-order-processing CPUs). Since the access is write-only, the application cannot read the contents, and therefore the actual contents are don't-care. This means that the new write operations can start execution before the read operations on the original resource are complete. Once the read operations have been completed, the driver can unalias the resource, by replacing the old resource with the new one. Again, new graphics APIs require the application to explicitly perform such optimizations.

This means that it is not only more difficult to write applications using the new graphics APIs, but it also means that it is difficult to make them perform well. The smart drivers offer a very good base-level of performance. While the new graphics APIs make it theoretically possible to write more efficient applications, in practice it takes a lot of clever mechanisms in code to beat the standards set by these drivers, in terms of memory usage and throughput.

3.3. Storage formats

Resources do not necessarily store the data in floating point format. Samplers can sample resources such as textures and related types of buffers, by fetching data from the resource, and converting the in-memory format to the correct format for the GPU. For example, a texture can contain RGBA data, stored as 8-bit values per component. A sampler will convert the 8-bit data to a 4-wide floating point vector datatype, where the 0..255 range of the byte is converted to 0..1 range.

Another abstraction at this level is the exact storage of the data in memory. For example, compressed texture formats can be used, which are transparent to the GPU, since the decompression happens at the sampler stage. It is also possible to 'swizzle' (reorder) the data, so that caching can be improved for the average case (linear storage would mean that caching of data is only efficient in one direction, and useless in the worst-case scenario of the perpendicular direction). It is also possible to pre-filter the data for magnification and minification, using so-called mip-maps. All this is generally hidden from the programmer, and a sampler will simply perform a lookup based on the fractional coordinates that are passed to it, and convert the data on-the-fly. Basic bilinear, trilinear or anisotropic filtering is also performed by the sampler automatically.

Certain operations require the data in a resources to be in a specific kind of format. For example, the GPU can render to a texture, but its rasterizer cannot render in a swizzled format. Also, when an application wants to read back from a resource with the CPU, the data must be in a known format, which is normally linear. In other words, when you write to a texture, the pipeline requires it to be in a write-friendly format, and when you read from a texture, it has to be in a read-friendly format. These formats are not necessarily the same (the exact formats and requirements may depend on the hardware used, and are abstracted by the driver), and a conversion from one format to another must take place.

Another example is a multisampled buffer. It cannot be used directly after rendering. The samples in the buffer must be 'resolved' to a regular downsampled buffer first.

In short, resources might have to 'transition' from time to time. In legacy graphics APIs this was handled transparently, but with modern APIs, transitions have to be performed explicitly by the application. A transition must occur when one type of operation on the resource is complete, and another type of operation is about to start. For example, after a render-to-texture operation (write operation), the resource is now used as a colourmap texture (read operation). There are various semantics for using resources, such as read-only access, write-only access, read-and-write access, use as render-target, use as compute shader buffer, and more. Each of these has its advantages and limitations for various types of use. Therefore applications need to transition resources often, or even create copies of the same resource with different usage semantics. Tracking all these resources and their states manually is complex and error-prone. A robust automated system would make it easier and faster to write correct applications.

Now that the basics of GPUs, graphics APIs and resources have been covered, we can formulate our research questions.

4

Research questions, scope, and methodology

The focus of this thesis is on the synchronization problems that result from the way modern graphics APIs work with multiple concurrent queues of command lists (concurrency), the usage of shared resources, and the management of memory residency of resources. The goal is to find ways to make these problems easier to handle when writing applications. We can now formulate our research question.

4.1. Research question

How can resource management and synchronization for GPUs using modern graphics APIs be simplified or automated?

This can be divided into various sub-questions:

RQ1 How can residency management be automated?

RQ2 How can concurrency be simplified?

RQ3 How can transition management be automated?

RQ4 How can the solution be evaluated and benchmarked, to quantify how well the solution has solved the problem?

The problems as described earlier, are not unique to GPUs and these new graphics APIs. They are merely specific instances of more generic problems in software development. The memory residency problem comes up in any situation where a computer system needs to manipulate more data than is physically available in the system. Various caching and memory paging systems have been developed over the years to solve these problems. The problem of synchronization of access to resources is also a generic problem, which is common in many types of concurrent/parallel computer systems. The tasks will be sharing resources via some kind of scheduling system, which will introduce non-determinism, because this scheduling makes it impossible to predict when a certain task is running, and when it will be accessing data. This leads to data race issues.

In the simplest case of "embarrassingly parallel" problems, all tasks are completely independent of each other, which means that there will be no data race issues. So even though the execution order of the tasks is non-deterministic, the results are deterministic.

In more complex parallel problems, there may still be certain subsets of tasks that are independent, and can run in parallel. Synchronization is then required with the next dependent subset of tasks, to avoid data races. A (partial) ordering of the parallel tasks can guarantee that the results will be equivalent to an "obvious" sequential program: *sequential-equivalent semantics* [5].

The problem also bears a strong resemblance to the concept of work-stealing in dynamically multithreaded applications [4]. This concept is perhaps best known from its implementation in the .NET Task Parallel Framework, with its underlying thread pool, and Task objects. New tasks can be scheduled to start when a previous task (or set of tasks) completes, which allows the developer to create a partial ordering of the parallel tasks, to ensure deterministic results.

This means that previous work has been done on related problems, and this previous work can form a foundation for possible solutions to this problem. In Chapter 8, we will look at relevant related work that may be applied to synchronization of concurrent/parallel computer systems.

4.2. SCOPE

In order to limit the scope and context of this thesis, a number of assumptions and choices have been made. These will be discussed below.

First and foremost, the goal of this thesis is to arrive at a solution that will be useful in real-world scenarios. This translates to a number of sub-goals:

- A primary design goal of modern APIs such as DirectX 12 and Vulkan is that overhead is reduced, in order to maximize performance. From this, it follows that we are only interested in solutions that perform well in practice. Otherwise, they would defeat the point of using a modern API in the first place.
- The solution should make using the modern APIs less complex, not more complex.

These goals will be used as guidelines when making decisions on what is within the scope of this thesis, and what is not.

In order to reduce complexity, some assumptions about the context of the program code can be made:

- We only consider a single application using a single GPU. This is a reasonable assumption because the OS already abstracts the GPU and API in a way that each application appears to have exclusive control over the GPU and API, even though multiple applications can share the GPU on a single system in practice.
- Since all GPU-related code resides in an application that is under our control, we can assume that we have full control over this code, and that each unit of code can be given full access to the state of any other unit of code, where required.

These assumptions can allow us to take shortcuts that might not apply in other types of synchronized systems, such as systems without shared memory, having only message-based communication over a network.

4.3. METHODOLOGY

The general approach to the three problems is to take the existing C++ interfaces for developing DirectX code, as provided by the Microsoft Windows SDK, combined with additional libraries from Microsoft, and build on this with more C++ code to develop a framework that abstracts some of the low-level functionality. The goal of this C++ framework will be to provide simpler interfaces and automate some of the resource management and related state-tracking.

I started out by writing my own DirectX 12 engine in C++ (see Figure 4.1), loosely based around earlier code I had written for DirectX 11, which includes basic loading and displaying of animated geometry. I then used this engine as a testbed for developing the C++ framework. Once the framework was fully functional, I extracted it into a separate library, and integrated it into Microsoft's DirectX 12 example code [22]. This allows me to firstly verify that the managed framework indeed works as expected in other environments than my own code. And secondly to compare before and after situations of external code, to draw conclusions about the ease of use and performance. This reduces the chance of the sample code being biased favourably towards the managed framework.

The next three chapters will cover which solutions have been chosen to address these problems, and will go into more specifics on how the solutions are verified and validated.

In the next chapter, we will discuss how to approach the answering of these research questions.

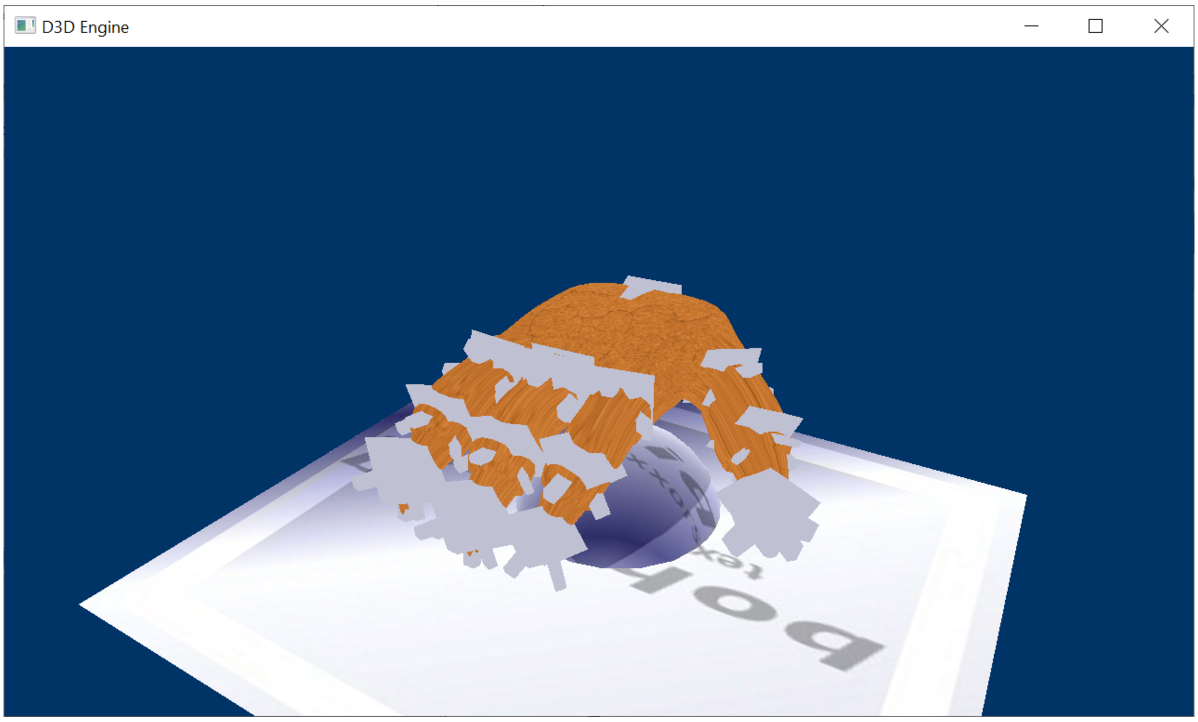


Figure 4.1: DirectX 12 engine.

5 AUTOMATED RESIDENCY MANAGEMENT

In this chapter we will address the problem of managing the residency of resources that may or not be in use by any active command lists on the GPU. As briefly covered in Chapter 3, DirectX 12 allows you to page video memory in and out, similar to virtual memory management on the CPU. However, it does not have an automated system that triggers a page-fault on access, and automatically pages memory in before use. DirectX 12 leaves it up to the developer to make sure that the memory for a resource is made resident before the resource is accessed.

5.1. PROBLEM ANALYSIS

The interface hierarchy for DirectX 12 is shown in Figure 5.1. The base interface for handling memory residency in DirectX 12 is I3D12PAGEABLE. This interface does not introduce any methods of its own. Instead, the ID3D12DEVICE interface has MAKERESIDENT() and EVICT() methods, which both take an array of ID3D12PAGEABLE objects. The ID3D12PAGEABLE interface merely functions as a 'marking' (also known as 'tagging') interface, to mark that an object can be paged in and out by the device.

Even though most D3D12 objects inherit from ID3D12PAGEABLE, residency changes are only supported on the following objects:

- Descriptor heaps (ID3D12DESCRIPTORHEAP)
- Query heaps (ID3D12QUERYHEAP)
- Heaps (ID3D12HEAP)
- Committed resources (ID3D12RESOURCE)

Of these committed resources, the following types support residency/eviction [26]:

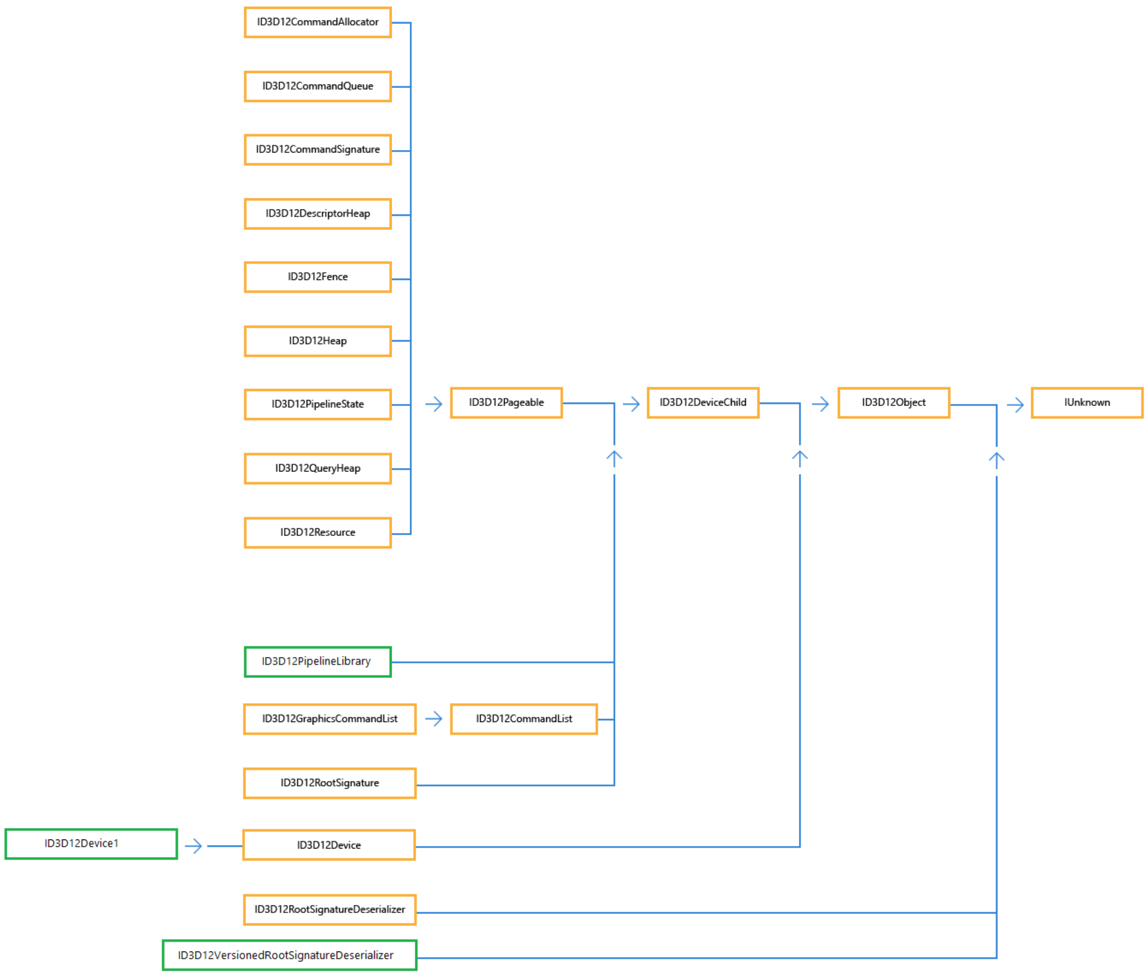


Figure 5.1: Interface inheritance hierachy of DirectX 12 [23].

- Textures (shader resources)
- Unordered access buffers (compute resources)
- Render targets
- Depth/stencil buffer
- Vertex buffers
- Index buffers
- Constant buffers

In order to make DirectX 12 as easy to use as earlier versions, the residency of these resources needs to be tracked automatically, so that residency can be ensured when a resource is being used.

5.2. USING THE MICROSOFT D3D12 RESIDENCY STARTER LIBRARY

Microsoft supplies a basic library to automatically handle the residency problem of resources [28]. Microsoft uses Proxy objects around the native ID3D12PAGEABLE interface for their implementation. The Proxy objects and the resource manager track object state in a way similar to a typestate system, except that it uses a dynamic runtime, rather than compile-time analysis. The Proxy objects derive the proper actions to ensure memory residency based on the state. This means that the library does not just perform verification. It performs synthesis, because it actually brings the resource objects in the desired state when the application requires it. It can do this because the states are very straightforward: a resource is either resident, or it is not. Whenever a non-resident resource is being accessed, there is a simple way to resolve the problem: make the resource resident. Similar to automated garbage collection, this managing of the resident state can be done automatically at runtime.

This library offers some basic functionality to track the residency of objects at runtime, and appears to be a good starting point. However, the library itself is very basic. It requires the user to manually create a MANAGEDOBJECT for every ID3D12PAGEABLE that they create, and wish to be managed. Also, the user is required to manually track when MANAGEDOBJECTS are used. This is done using the RESIDENCYSET object that is also provided by the library.

The library handles residency on a per-command list basis. It does this by providing an alternative implementation for the EXECUTECOMMANDLISTS() method of ID3D12COMMANDQUEUE. This implementation of RESIDENCYMANAGER::EXECUTECOMMANDLISTS() takes not only a set of command lists as its parameters, but also a set of RESIDENCYSETS. For each command list you want to execute, you are to provide a RESIDENCYSET, in which you have inserted all MANAGEDOBJECTS that will be used by the command list, and whose residency needs to be ensured by the library.

As such, the library does the basic job of tracking the residency of objects that are currently queued for execution. It will ensure that objects are resident before the command list starts executing. It will also automatically evict resources that are not in use (on a least-recently-used basis), to make sure that there is enough memory available before making the required objects resident.

The library takes away the manual work of calling MAKERESIDENT() and EVICT(), and tracking memory usage on the GPU. However, in return it now gives the user additional MANAGEDOBJECTS and RESIDENCYSETS to manage.

It is the responsibility of the developer to make sure each RESIDENCYSET is filled correctly. If the set contains additional MANAGEDOBJECTS that are not actually used by the command list, the manager will make them resident anyway, which may result in redundant work, and using more video memory than is required. If objects are omitted, they will not be made resident even when they are used, which may lead to visual glitches.

The developer can choose not to manage certain objects however. If a resource is never added to any RESIDENCYSET, then it will never be made resident or evicted by the manager. As such, it is possible to divide up resources in multiple classes: a managed class and and unmanaged class. Then the unmanaged class can either be kept resident for the lifetime of the application, or only be made resident or evicted at specific points in the lifetime of the application.

The next step then is to automate this process as well. I chose to do this in a similar way to the alternative EXECUTECOMMANDLISTS() method that is provided by the library: by providing Proxy objects with API functions that are equal to, or at least similar to, the native API, but which allow extra processing and state tracking between the call from the application to our Proxy object, and the actual call to the native API. The Proxy objects can then keep track of the used resources in linked lists that they keep internally. When the EXECUTECOMMANDLISTS() method is called, each Proxy object for the command list can then build a RESIDENCYSET on-the-fly and fill it with the list of resources that have been tracked.

5.3. AUTOMATING THE USE OF MANAGEDOBJECTS

A resource object functions as an abstraction of allocated memory for use by the GPU. The resource can be located in either system memory or video memory. The main difficulty with automatically computing the residency sets of resource objects is that the residency works at the level of the ID3D12PAGEABLE interface (which are used by the ID3D12DEVICE::MAKERESIDENT() and ID3D12DEVICE::EVICT() methods), while the actual rendering APIs do not access resources at this level of abstraction.

The rendering API calls do not use the resource objects directly. They use a lower level of abstraction than the resource objects, and work on simple 'descriptor' or 'view' datastructures, which contain a direct pointer into GPU-accessible memory (which can be derived from the resource), and some information to describe the memory layout. So there is aliasing between the resource objects and the actual parameters passed to rendering calls.

With CPUs, an automated paging system can be implemented, by using the built-in Memory Management Unit (MMU). The CPU can generate an exception when non-resident memory areas are accessed, and an exception handler can then make the memory resident, and return control to the code where the exception occured. GPUs do not have an MMU yet, so there is no way to automatically detect access to non-resident memory areas, and solve it transparently. The application will actively have to track access to resources by monitoring their use in the rendering API calls. So the aliasing that is introduced by creating views on the resource objects, needs to be resolved.

For example, this is a D3D12_VERTEX_BUFFER_VIEW, which is used as a parameter in the ID3D12GRAPHICSCOMMANDLIST::IASETVERTEXBUFFERS() method:

```
typedef struct D3D12_VERTEX_BUFFER_VIEW
{
    D3D12_GPU_VIRTUAL_ADDRESS BufferLocation;
    UINT SizeInBytes;
    UINT StrideInBytes;
}   D3D12_VERTEX_BUFFER_VIEW;
```

So when using a vertex buffer in a command list, we must resolve this aliasing by remembering which resource it came from, so that we can get its ID3D12PAGEABLE and its MANAGEDOBJECT, to manage its memory. Other types of resources have similar aliasing through indirect relationships between the resource that allocates the memory and the object that is passed to an API call.

This management is automated in two steps:

1. Composition objects that link the API parameter datatype to the ID3D12PAGEABLE that allocates it
2. A translation layer which uses these objects to manage the ID3D12PAGEABLE memory, and then passes the aliased parameter to the actual API call

The composition step is done with some simple structs that combine a reference to the ID3D12PAGEABLE, the MANAGEDOBJECT for the Residency Library and the parameter datatype. The de-aliasing is then implemented as a set of Proxy objects for the DirectX 12 objects that expose the relevant API calls. The Proxy objects add new API calls that are similar to the original ones, but instead take the new objects with managed data as their parameters, rather than the original unmanaged objects. The API call can then perform the required tracking with the Residency Library (by adding the composition object to its internal linked list), and pass on the underlying parameter datatype to the non-managed API call.

A generic base struct for composition is created by combining the ID3DPAGEABLE and the MANAGEDOBJECT, as well as some other relevant metadata regarding the size of the object:

```
struct ManagedResource
{
    ComPtr<ID3D12Pageable> resource;                      // The D3D12 resource that
    will be managed by the residency library.
    D3DX12Residency::ManagedObject* trackingHandle;       // The residency library
    works in units of ManagedObjects. They can be treated opaquely like a handle.
    UINT64 size = 0;
    UINT index = 0;
};
```

This base struct can then be subclassed for specific composition types which add the parameter datatype. For example, a MANAGEDVERTEXBUFFER will look like this:

```
struct ManagedVertexBuffer : public ManagedResource
{
    D3D12_VERTEX_BUFFER_VIEW view;
};
```

The non-managed ID3D12GRAPHICSCOMMANDLIST::IASETVERTEXBUFFERS() method looks like this:

```
void IASetVertexBuffers(UINT StartSlot, UINT NumViews, const D3D12_VERTEX_BUFFER_VIEW*
pViews);
```

The managed Proxy object can then supply a managed method MANAGEDCOMMANDLIST::IASETVERTEXBUFFERS() like this:

```
void STDMETHODCALLTYPE IASetVertexBuffers(UINT StartSlot, UINT NumViews,
ManagedVertexBuffer* pViews);
```

DirectX 12 only exposes abstract interfaces for its objects, and not the actual classes with the implementation. This means it is not possible to use subclassing to create a Proxy object. The only way is via aggregation. In this case, MANAGEDCOMMANDLIST implements the ID3D12GRAPHICSCOMMANDLIST interface, and adds its new managed Proxy methods to the existing interface. A reference to the actual ID3D12 object is stored inside the Proxy object, and the inherited ID3D12 methods can be redirected to the underlying ID3D12 object as-is, while the new managed objects will perform the required management logic automatically, before calling the underlying unmanaged ID3D12 method.

To complete this management scheme, the following MANAGEDRESOURCE subclasses were created:

- MANAGEDCPUDESCRIPTOR, to manage a D3D12_CPU_DESCRIPTOR_HANDLE
- MANAGEDGPUDESCRIPTOR, to manage a D3D12_GPU_DESCRIPTOR_HANDLE
- MANAGEDGPUADDRESS, to manage a D3D12_GPU_VIRTUAL_ADDRESS
- MANAGEDVERTEXBUFFER, to manage a D3D12_VERTEX_BUFFER_VIEW
- MANAGEDINDEXBUFFER, to manage a D3D12_INDEX_BUFFER_VIEW

And the following Proxy objects were created:

- MANAGEDDEVICE, implementing ID3D12DEVICE
- MANAGEDCOMMANDLIST, implementing ID3D12GRAPHICSCOMMANDLIST
- MANAGEDROOTSIGNATURE, implementing ID3D12ROOTSIGNATURE
- MANAGEDDESCRIPTORHEAP implementing ID3D12DESCRIPTORHEAP
- MANAGEDCOMMANDQUEUE, implementing ID3D12COMMANDQUEUE

The included source code contains the implementation details of these Proxy objects.

In DirectX12, the ID3D12DEVICE interface provides methods to instantiate the ID3D12GRAPHICSCOMMANDLIST, ID3D12ROOTSIGNATURE, ID3D12DESCRIPTORHEAP and ID3D12COMMANDQUEUE objects. The Proxy object MANAGEDDEVICE provides analogous methods to instantiate MANAGEDCOMMANDLIST, MANAGEDROOTSIGNATURE, MANAGEDDESCRIPTORHEAP and MANAGEDCOMMANDQUEUE objects. When using the MANAGEDCOMMANDLIST to buffer render calls, in combination with MANAGEDRESOURCE-derived resources, the usage of these resources will be tracked automatically in a linked list (duplicate resources will be filtered out before adding to the list). Then, when calling MANAGEDCOMMANDQUEUE::EXECUTECOMMANDLISTS(), the internal linked lists of resources will be converted to RESIDENCYSETS on-the-fly by calling MANAGEDCOMMANDLIST::UPDATERESIDENCYSET(). These RESIDENCYSETS will automatically be sent to the RESIDENCYMANAGER::EXECUTECOMMANDLISTS() Proxy method provided by the D3D12 Residency Started Library. This RESIDENCYMANAGER will take care of the actual MAKERESIDENT() and EVICT() calls required, to ensure that all resources will be resident prior to executing the command lists.

5.4. LIMITATIONS

The above approach only considers the use of resources from the API side. This means that we determine which resources are bound to the rendering pipeline at specific API calls, and at which specific slots they are bound in the root signature. So in other words: which resources are made available to the GPU. This does not necessarily mean that the resources are actually used by the GPU. It is possible that the shaders being executed do not actually access all of the bound resources.

So with the proposed solution, the effectiveness depends on how the developer uses resource binding and shaders. If the resource bindings match the requirements of the shader code closely, then this approach will work well in practice. However, if the developer goes for a 'greedy' approach by binding a large number of resources in one go, while executing shaders that each only access a subset of these resources, then this approach may not work that well in practice, as it will also act 'greedy' and will try to keep all resources resident all the time. This will mean that it will use more video memory than required. It will also lead to extra CPU overhead, as each element that is tracked in the list takes time to process, convert to a RESIDENCYSET, and evalaute the state of residency.

So in its current state, users of the residency framework are recommended to design their code to match resource bindings closely with their use in shader code, for best results.

5.5. VALIDATING THE SOLUTION

Three criteria are important to evaluate with this solution:

1. Correctness
2. Ease of use
3. Performance

Correctness Correctness in this case is defined as follows: All resources used by a commandlist are made resident upon execution of this command list.

Because of the nature of DirectX 12, evicting a resource does not necessarily mean that using the resource will lead to incorrect visual results. It merely means that the memory can be re-used for future allocations, and it may be overwritten. I have found that it was very hard to actually force such a situation, even when evicting the original textures, and then continuously loading new textures to a point where all the physical memory should have been filled. I only managed to generate a visual artifact on one machine, where one of the textures turned to black (see Figure 5.2). That is not entirely what I would have expected if

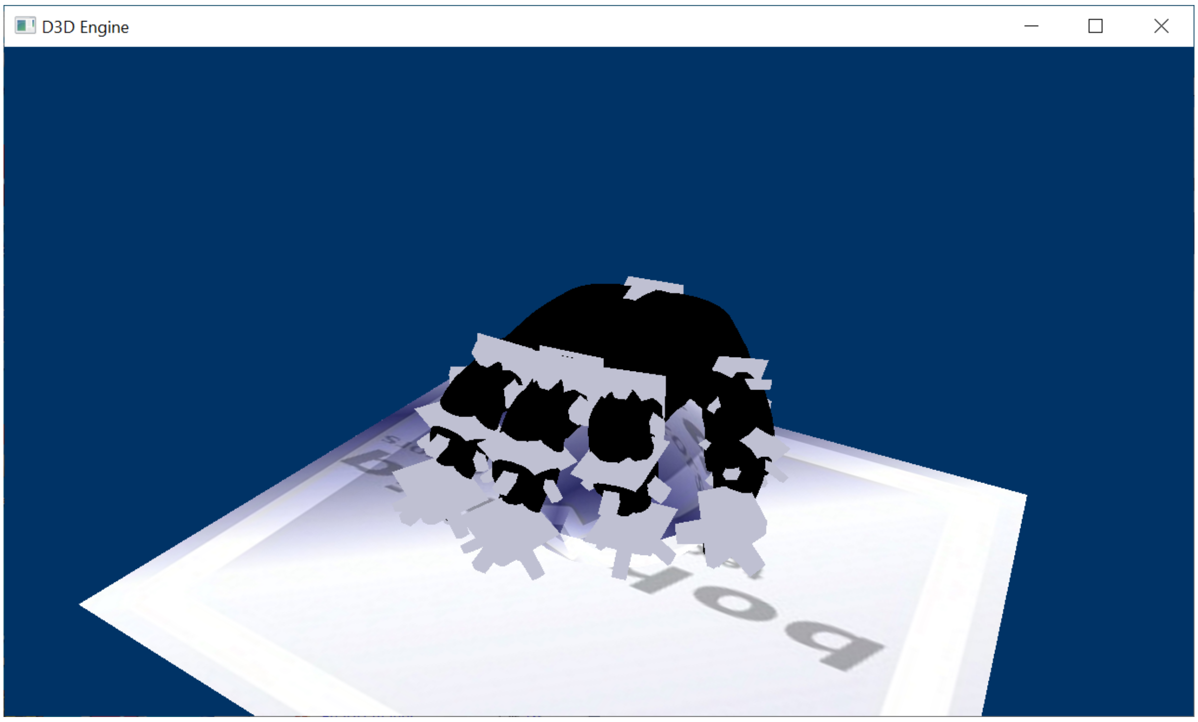
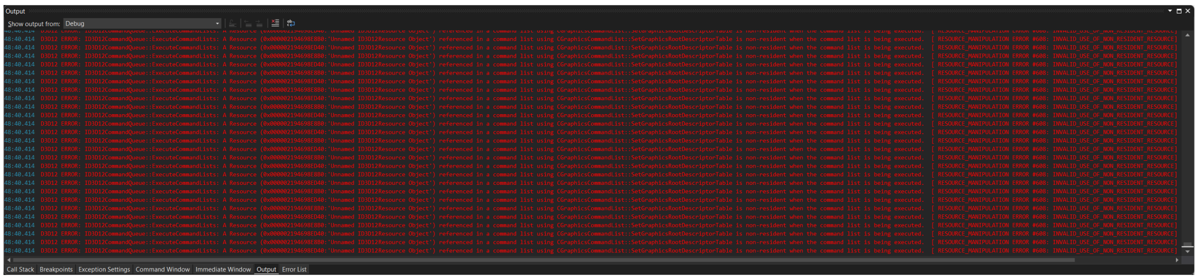


Figure 5.2: DX12 missing evicted texture.

the memory were actually overwritten with the other textures that had been loaded. At any rate, visual verification is not the way to determine whether textures are actually resident or not.

The DirectX 12 debug layer has a checking mechanism for this, and will display an error on the debug output for any resource that is not resident while used (see Figure 5.3.) This func-



```
48:40.414  D3D12 ERROR: ID3D12CommandQueue::ExecuteCommandLists: A Resource (0x000002194698E8B0:'Unnamed ID3D12Resource Object') referenced in a command list using CGraphicsCommandList::SetGraphicsRootDescriptorTable is non-resident when the command list is being executed.  [ RESOURCE_MANIPULATION ERROR #608: INVALID_USE_OF_NON_RESIDENT_RESOURCE]
48:40.414  D3D12 ERROR: ID3D12CommandQueue::ExecuteCommandLists: A Resource (0x000002194698ED40:'Unnamed ID3D12Resource Object') referenced in a command list using CGraphicsCommandList::SetGraphicsRootDescriptorTable is non-resident when the command list is being executed.  [ RESOURCE_MANIPULATION ERROR #608: INVALID_USE_OF_NON_RESIDENT_RESOURCE]
48:40.414  D3D12 ERROR: ID3D12CommandQueue::ExecuteCommandLists: A Resource (0x000002194698E8B0:'Unnamed ID3D12Resource Object') referenced in a command list using CGraphicsCommandList::SetGraphicsRootDescriptorTable is non-resident when the command list is being executed.  [ RESOURCE_MANIPULATION ERROR #608: INVALID_USE_OF_NON_RESIDENT_RESOURCE]
48:40.414  D3D12 ERROR: ID3D12CommandQueue::ExecuteCommandLists: A Resource (0x000002194698ED40:'Unnamed ID3D12Resource Object') referenced in a command list using CGraphicsCommandList::SetGraphicsRootDescriptorTable is non-resident when the command list is being executed.  [ RESOURCE_MANIPULATION ERROR #608: INVALID_USE_OF_NON_RESIDENT_RESOURCE]
48:40.414  D3D12 ERROR: ID3D12CommandQueue::ExecuteCommandLists: A Resource (0x000002194698E8B0:'Unnamed ID3D12Resource Object') referenced in a command list using CGraphicsCommandList::SetGraphicsRootDescriptorTable is non-resident when the command list is being executed.  [ RESOURCE_MANIPULATION ERROR #608: INVALID_USE_OF_NON_RESIDENT_RESOURCE]
48:40.414  D3D12 ERROR: ID3D12CommandQueue::ExecuteCommandLists: A Resource (0x000002194698ED40:'Unnamed ID3D12Resource Object') referenced in a command list using CGraphicsCommandList::SetGraphicsRootDescriptorTable is non-resident when the command list is being executed.  [ RESOURCE_MANIPULATION ERROR #608: INVALID_USE_OF_NON_RESIDENT_RESOURCE]
48:40.414  D3D12 ERROR: ID3D12CommandQueue::ExecuteCommandLists: A Resource (0x000002194698E8B0:'Unnamed ID3D12Resource Object') referenced in a command list using CGraphicsCommandList::SetGraphicsRootDescriptorTable is non-resident when the command list is being executed.  [ RESOURCE_MANIPULATION ERROR #608: INVALID_USE_OF_NON_RESIDENT_RESOURCE]
48:40.414  D3D12 ERROR: ID3D12CommandQueue::ExecuteCommandLists: A Resource (0x000002194698ED40:'Unnamed ID3D12Resource Object') referenced in a command list using CGraphicsCommandList::SetGraphicsRootDescriptorTable is non-resident when the command list is being executed.  [ RESOURCE_MANIPULATION ERROR #608: INVALID_USE_OF_NON_RESIDENT_RESOURCE]
48:40.414  D3D12 ERROR: ID3D12CommandQueue::ExecuteCommandLists: A Resource (0x000002194698E8B0:'Unnamed ID3D12Resource Object') referenced in a command list using CGraphicsCommandList::SetGraphicsRootDescriptorTable is non-resident when the command list is being executed.  [ RESOURCE_MANIPULATION ERROR #608: INVALID_USE_OF_NON_RESIDENT_RESOURCE]
48:40.414  D3D12 ERROR: ID3D12CommandQueue::ExecuteCommandLists: A Resource (0x000002194698ED40:'Unnamed ID3D12Resource Object') referenced in a command list using CGraphicsCommandList::SetGraphicsRootDescriptorTable is non-resident when the command list is being executed.  [ RESOURCE_MANIPULATION ERROR #608: INVALID_USE_OF_NON_RESIDENT_RESOURCE]
48:40.414  D3D12 ERROR: ID3D12CommandQueue::ExecuteCommandLists: A Resource (0x000002194698E8B0:'Unnamed ID3D12Resource Object') referenced in a command list using CGraphicsCommandList::SetGraphicsRootDescriptorTable is non-resident when the command list is being executed.  [ RESOURCE_MANIPULATION ERROR #608: INVALID_USE_OF_NON_RESIDENT_RESOURCE]
48:40.414  D3D12 ERROR: ID3D12CommandQueue::ExecuteCommandLists: A Resource (0x000002194698ED40:'Unnamed ID3D12Resource Object') referenced in a command list using CGraphicsCommandList::SetGraphicsRootDescriptorTable is non-resident when the command list is being executed.  [ RESOURCE_MANIPULATION ERROR #608: INVALID_USE_OF_NON_RESIDENT_RESOURCE]
48:40.414  D3D12 ERROR: ID3D12CommandQueue::ExecuteCommandLists: A Resource (0x000002194698E8B0:'Unnamed ID3D12Resource Object') referenced in a command list using CGraphicsCommandList::SetGraphicsRootDescriptorTable is non-resident when the command list is being executed.  [ RESOURCE_MANIPULATION ERROR #608: INVALID_USE_OF_NON_RESIDENT_RESOURCE]
48:40.414  D3D12 ERROR: ID3D12CommandQueue::ExecuteCommandLists: A Resource (0x000002194698ED40:'Unnamed ID3D12Resource Object') referenced in a command list using CGraphicsCommandList::SetGraphicsRootDescriptorTable is non-resident when the command list is being executed.  [ RESOURCE_MANIPULATION ERROR #608: INVALID_USE_OF_NON_RESIDENT_RESOURCE]
48:40.414  D3D12 ERROR: ID3D12CommandQueue::ExecuteCommandLists: A Resource (0x000002194698E8B0:'Unnamed ID3D12Resource Object') referenced in a command list using CGraphicsCommandList::SetGraphicsRootDescriptorTable is non-resident when the command list is being executed.  [ RESOURCE_MANIPULATION ERROR #608: INVALID_USE_OF_NON_RESIDENT_RESOURCE]
48:40.414  D3D12 ERROR: ID3D12CommandQueue::ExecuteCommandLists: A Resource (0x000002194698ED40:'Unnamed ID3D12Resource Object') referenced in a command list using CGraphicsCommandList::SetGraphicsRootDescriptorTable is non-resident when the command list is being executed.  [ RESOURCE_MANIPULATION ERROR #608: INVALID_USE_OF_NON_RESIDENT_RESOURCE]
48:40.414  D3D12 ERROR: ID3D12CommandQueue::ExecuteCommandLists: A Resource (0x000002194698E8B0:'Unnamed ID3D12Resource Object') referenced in a command list using CGraphicsCommandList::SetGraphicsRootDescriptorTable is non-resident when the command list is being executed.  [ RESOURCE_MANIPULATION ERROR #608: INVALID_USE_OF_NON_RESIDENT_RESOURCE]
48:40.414  D3D12 ERROR: ID3D12CommandQueue::ExecuteCommandLists: A Resource (0x000002194698ED40:'Unnamed ID3D12Resource Object') referenced in a command list using CGraphicsCommandList::SetGraphicsRootDescriptorTable is non-resident when the command list is being executed.  [ RESOURCE_MANIPULATION ERROR #608: INVALID_USE_OF_NON_RESIDENT_RESOURCE]
48:40.414  D3D12 ERROR: ID3D12CommandQueue::ExecuteCommandLists: A Resource (0x000002194698E8B0:'Unnamed ID3D12Resource Object') referenced in a command list using CGraphicsCommandList::SetGraphicsRootDescriptorTable is non-resident when the command list is being executed.  [ RESOURCE_MANIPULATION ERROR #608: INVALID_USE_OF_NON_RESIDENT_RESOURCE]
48:40.414  D3D12 ERROR: ID3D12CommandQueue::ExecuteCommandLists: A Resource (0x000002194698ED40:'Unnamed ID3D12Resource Object') referenced in a command list using CGraphicsCommandList::SetGraphicsRootDescriptorTable is non-resident when the command list is being executed.  [ RESOURCE_MANIPULATION ERROR #608: INVALID_USE_OF_NON_RESIDENT_RESOURCE]
48:40.414  D3D12 ERROR: ID3D12CommandQueue::ExecuteCommandLists: A Resource (0x000002194698E8B0:'Unnamed ID3D12Resource Object') referenced in a command list using CGraphicsCommandList::SetGraphicsRootDescriptorTable is non-resident when the command list is being executed.  [ RESOURCE_MANIPULATION ERROR #608: INVALID_USE_OF_NON_RESIDENT_RESOURCE]
48:40.414  D3D12 ERROR: ID3D12CommandQueue::ExecuteCommandLists: A Resource (0x000002194698ED40:'Unnamed ID3D12Resource Object') referenced in a command list using CGraphicsCommandList::SetGraphicsRootDescriptorTable is non-resident when the command list is being executed.  [ RESOURCE_MANIPULATION ERROR #608: INVALID_USE_OF_NON_RESIDENT_RESOURCE]
48:40.414  D3D12 ERROR: ID3D12CommandQueue::ExecuteCommandLists: A Resource (0x000002194698E8B0:'Unnamed ID3D12Resource Object') referenced in a command list using CGraphicsCommandList::SetGraphicsRootDescriptorTable is non-resident when the command list is being executed.  [ RESOURCE_MANIPULATION ERROR #608: INVALID_USE_OF_NON_RESIDENT_RESOURCE]
48:40.414  D3D12 ERROR: ID3D12CommandQueue::ExecuteCommandLists: A Resource (0x000002194698ED40:'Unnamed ID3D12Resource Object') referenced in a command list using CGraphicsCommandList::SetGraphicsRootDescriptorTable is non-resident when the command list is being executed.  [ RESOURCE_MANIPULATION ERROR #608: INVALID_USE_OF_NON_RESIDENT_RESOURCE]
48:40.414  D3D12 ERROR: ID3D12CommandQueue::ExecuteCommandLists: A Resource (0x000002194698E8B0:'Unnamed ID3D12Resource Object') referenced in a command list using CGraphicsCommandList::SetGraphicsRootDescriptorTable is non-resident when the command list is being executed.  [ RESOURCE_MANIPULATION ERROR #608: INVALID_USE_OF_NON_RESIDENT_RESOURCE]
48:40.414  D3D12 ERROR: ID3D12CommandQueue::ExecuteCommandLists: A Resource (0x000002194698ED40:'Unnamed ID3D12Resource Object') referenced in a command list using CGraphicsCommandList::SetGraphicsRootDescriptorTable is non-resident when the command list is being executed.  [ RESOURCE_MANIPULATION ERROR #608: INVALID_USE_OF_NON_RESIDENT_RESOURCE]
```

Figure 5.3: Debug messages for non-resident resources.

tionality can be used to verify the correctness: Create all managed resources in an evicted state. If the residency manager works correctly, it will make all the resources resident, and there will be no error messages.

Ease of use While 'ease of use' may be a somewhat subjective issue, we can perform some code metrics on the code which will give us objective, empirical data on the complexity of the code. This approach is also used to determine the effect of code refactoring [8, 3], or to select refactoring strategies to improve code quality [32]. I have taken one of the more advanced examples from Microsoft [22], namely the D3D12MULTITHREADING example [21] (see Figure 5.4). This sample does not have any residency management at all. I have modified it to make use of the new managed Proxy objects. This is not refactoring, because the functionality of the code is changed: the residency management is being added. However, it is a similar problem, and code metrics can tell us how the quality of the code is affected by adding the managed Proxy objects. The changes to the code are very minor, and mainly

Figure 5.4: D3D12 Multithreading Sample.

concentrate on a select few method calls. Existing objects are replaced with Proxy objects, which by their nature, are very similar in design. Also, both the original DirectX 12 objects and their managed Proxy replacements are considered to be black boxes. This means that code metrics at the class/object level do not apply. The following metrics do apply however:

- Lines of code (LOC)
- Message chain (MC)
- Long parameter list (LPL)
- Cyclomatic complexity (CC)

I will now present some before and after snippets of code, and compare how the usage of the managed Proxy objects affects these code metrics. LOC will only include relevant lines, blank lines or lines with only comments will be ignored. MC is counted in number of lines that depend on previous lines. LPL is counted in number of parameters per function or method call. CC is counted as the number of possible branches that occur in the control-flow of the code (such as if-statements or switch-statements).

First the RESIDENCYMANAGER has to be added to the code and initialized:

```
ResidencyManager m_residencyManager;
...
// Initialize the D3DX12 Residency Manager
// Average stats for determining latency
const int NumberOfBufferedFrames = 2;
const int NumberOfCommandListSubmissionsPerFrame = 5;

ThrowIfFailed(m_residencyManager.Initialize(m_device.Get(), 0, m_adapter.Get(),
NumberOfBufferedFrames * NumberOfCommandListSubmissionsPerFrame));
```

Essentially these are just two lines of code: the definition of a variable for the RESIDENCYMANAGER, and the call to INITIALIZE(). The call takes 4 parameters. For clarity, the fourth parameter is derived from two constants that can easily be adjusted, so in this case there are 4 lines in total.

LOC: +4
LPL: +4

Then, the declaration of the variables has to be modified to use the managed objects.

Before:

```
ComPtr<ID3D12Device> m_device;
ComPtr<ID3D12CommandQueue> m_commandQueue;
ComPtr<ID3D12RootSignature> m_rootSignature;
ComPtr<ID3D12DescriptorHeap> m_rtvHeap;
ComPtr<ID3D12DescriptorHeap> m_dsvHeap;
ComPtr<ID3D12DescriptorHeap> m_cbvSrvHeap;
ComPtr<ID3D12DescriptorHeap> m_samplerHeap;
...
D3D12_VERTEX_BUFFER_VIEW m_vertexBufferView;
D3D12_INDEX_BUFFER_VIEW m_indexBufferView;
ComPtr<ID3D12Resource> m_textures[_countof(SampleAssets::Textures)];
ComPtr<ID3D12Resource> m_indexBuffer;
ComPtr<ID3D12Resource> m_vertexBuffer;
```

After:

```
ComPtr<ManagedDevice> m_device;
ComPtr<ManagedCommandQueue> m_commandQueue;
ComPtr<ManagedRootSignature> m_rootSignature;
ComPtr<ManagedDescriptorHeap> m_rtvHeap;
ComPtr<ManagedDescriptorHeap> m_dsvHeap;
ComPtr<ManagedDescriptorHeap> m_cbvSrvHeap;
ComPtr<ManagedDescriptorHeap> m_samplerHeap;
...
ManagedVertexBuffer m_vertexBuffer;
ManagedIndexBuffer m_indexBuffer;
ManagedResource m_textures[_countof(SampleAssets::Textures)];
```

This is a simple search-and-replace operation. In fact, because the MANAGEDVERTEXBUFFER and MANAGEDINDEXBUFFER integrate the D3D12_VERTEX_BUFFER_VIEW and D3D12_INDEX_BUFFER_VIEW respectively, there are actually two less lines required. So the lines-of-code metric is down by 2 lines.

LOC: -2

Then the various managed Proxy objects need to be inintialized. First there is the MANAGEDDEVICE. This is initialized by passing a ID3D12DEVICE to its constructor, and adds one extra line of code:

```
m_device = new ManagedDevice(device.Get());
```

LOC: +1
LPL: +1

Next up is the command queue.

Before:

```
ThrowIfFailed(m_device->CreateCommandQueue(&queueDesc, IID_PPV_ARGS(&m_commandQueue)));
```

After:

```
ThrowIfFailed(m_device->CreateManagedCommandQueue(&queueDesc, &m_residencyManager, &
m_commandQueue));
```

Both require a single line. The IID_PPV_ARGS() macro actually generates two arguments, based on a COM object (first argument is the UUID of the object to create, the second is a pointer that receives the newly created object). In the managed version, there is no need for this construction, because it simply uses a pointer to a MANAGEDCOMMANDQUEUE pointer, rather than the COM-based mechanism with a separate UUID. It does however require an extra parameter to pass in a pointer to the residency manager that is to be used. But this is trivial. So depending on how the code is viewed, the long parameter list metric either stays the same, at 3 parameters, or it goes up from 2 to 3 parameters. We will take the worst case here.

LPL: +1

The exact same goes for the descriptor heaps: parameter pair changed to single pointer-to-pointer, and extra parameter to pass in the residency manager.

Before:

```
ThrowIfFailed(m_device->CreateDescriptorHeap(&rtvHeapDesc, IID_PPV_ARGS(&m_rtvHeap)));
```

After:

```
ThrowIfFailed(m_device->CreateManagedDescriptorHeap(&dsvHeapDesc, &m_residencyManager,
&m_dsvHeap));
```

LPL: +1

With the command lists, it is the same again.

Before:

```
ThrowIfFailed(m_device->CreateCommandList(0, D3D12_COMMAND_LIST_TYPE_DIRECT,
m_commandAllocator.Get(), m_pipelineState.Get(), IID_PPV_ARGS(&commandList)));
```

After:

```
ThrowIfFailed(m_device->CreateManagedCommandList(0, D3D12_COMMAND_LIST_TYPE_DIRECT,
m_commandAllocator.Get(), m_pipelineState.Get(), &m_residencyManager, &commandList));
```

LPL: +1

The root signature is again an example where the managed code is actually somewhat simpler than the regular DirectX 12 code. This is because DirectX12 requires you to serialize a root signature definition into an ID3DBLOB first, and then pass the serialized version. The managed version takes the signature description structure directly, and does the serialization implicitly.

Before:

```
ThrowIfFailed(D3DX12SerializeVersionedRootSignature(&rootSignatureDesc, featureData.
HighestVersion, &signature, &error));
ThrowIfFailed(m_device->CreateRootSignature(0, signature->GetBufferPointer(), signature
->GetBufferSize(), IID_PPV_ARGS(&m_rootSignature)));
```

After:

```
ThrowIfFailed(m_device->CreateManagedRootSignature(0, &rootSignatureDesc, featureData.
HighestVersion, &m_rootSignature));
```

LOC: -1
MC: -1
LPL: -1

Now we get to the actual MANAGEDRESOURCE objects. For textures, the following code needs to be added:

```
D3D12_RESOURCE_DESC textureDesc = pRes->GetDesc();

// Manage this object
D3D12_RESOURCE_ALLOCATION_INFO info = m_device->GetResourceAllocationInfo(0, 1, &
textureDesc);
m_textures[i].Initialize(&m_residencyManager, pRes.Get(), info.SizeInBytes, 0);
```

That is the most complex so far. The first part is to retrieve the description of the resource after it has been created (so that the driver has determined the exact layout in memory, and thereby the required pixel size, alignment, row pitch and such). Then the GETRESOURCEALLOCATIONINFO() method can be called, to get the actual size of the resource in bytes. This size is then passed to the INITIALIZE() method of the MANAGEDRESOURCE, so that it can be tracked by the RESIDENCYMANAGER.

LOC: +3
MC: +3
LPL: +7

Then, there is a slight change with the way we use descriptor handles for views. We will see this method of addressing resource views with the root descriptor as well: a combination of a heap reference and a slot index. This allows the Proxy objects to track the underlying resources of the views. This would not be trivial when using the opaque descriptor handles directly. This approach should be familiar to developers who have used earlier versions of DirectX, since these earlier versions also used slots to assign views to.

This actually results in having to use less code overall, because we can do away with the CPU handle variable.

Before:

```
// Get the CBV SRV descriptor size for the current device.
const UINT cbvSrvDescriptorSize = m_device->GetDescriptorHandleIncrementSize(
D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV);

// Get a handle to the start of the descriptor heap.
CD3DX12_CPU_DESCRIPTOR_HANDLE cbvSrvHandle(m_cbvSrvHeap->
GetCPUDescriptorHandleForHeapStart());

{
    // Describe and create 2 null SRVs. Null descriptors are needed in order
    // to achieve the effect of an "unbound" resource.
    D3D12_SHADER_RESOURCE_VIEW_DESC nullSrvDesc = {};
    nullSrvDesc.ViewDimension = D3D12_SRV_DIMENSION_TEXTURE2D;
    nullSrvDesc.Shader4ComponentMapping = D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING;
    nullSrvDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
    nullSrvDesc.Texture2D.MipLevels = 1;
    nullSrvDesc.Texture2D.MostDetailedMip = 0;
    nullSrvDesc.Texture2D.ResourceMinLODClamp = 0.0f;

    m_device->CreateShaderResourceView(nullptr, &nullSrvDesc, cbvSrvHandle);
    cbvSrvHandle.Offset(cbvSrvDescriptorSize);

    m_device->CreateShaderResourceView(nullptr, &nullSrvDesc, cbvSrvHandle);
    cbvSrvHandle.Offset(cbvSrvDescriptorSize);
}
...
{
    const UINT subresourceCount = texDesc.DepthOrArraySize * texDesc.MipLevels;
    UINT64 uploadBufferSize = GetRequiredIntermediateSize(m_textures[i].Get(), 0,
    subresourceCount);
    ThrowIfFailed(m_device->CreateCommittedResource(
    &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_UPLOAD),
    D3D12_HEAP_FLAG_NONE,
    &CD3DX12_RESOURCE_DESC::Buffer(uploadBufferSize),
    D3D12_RESOURCE_STATE_GENERIC_READ,
    nullptr,
    IID_PPV_ARGS(&m_textureUploads[i])));

    // Copy data to the intermediate upload heap and then schedule a copy
    // from the upload heap to the Texture2D.
    D3D12_SUBRESOURCE_DATA textureData = {};
    textureData.pData = pAssetData + tex.Data->Offset;
    textureData.RowPitch = tex.Data->Pitch;
    textureData.SlicePitch = tex.Data->Size;

    UpdateSubresources(commandList.Get(), m_textures[i].Get(), m_textureUploads[i].Get
    (), 0, 0, subresourceCount, &textureData);
    commandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(m_textures[i
    ].Get(), D3D12_RESOURCE_STATE_COPY_DEST, D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE
    ));
}

// Describe and create an SRV.
D3D12_SHADER_RESOURCE_VIEW_DESC srvDesc = {};
srvDesc.ViewDimension = D3D12_SRV_DIMENSION_TEXTURE2D;
srvDesc.Shader4ComponentMapping = D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING;
srvDesc.Format = tex.Format;
srvDesc.Texture2D.MipLevels = tex.MipLevels;
srvDesc.Texture2D.MostDetailedMip = 0;
srvDesc.Texture2D.ResourceMinLODClamp = 0.0f;
m_device->CreateShaderResourceView(m_textures[i].Get(), &srvDesc, cbvSrvHandle);

// Move to the next descriptor slot.
cbvSrvHandle.Offset(cbvSrvDescriptorSize);
```

After:

```
{
    // Describe and create 2 null SRVs. Null descriptors are needed in order
    // to achieve the effect of an "unbound" resource.
    D3D12_SHADER_RESOURCE_VIEW_DESC nullSrvDesc = {};
    nullSrvDesc.ViewDimension = D3D12_SRV_DIMENSION_TEXTURE2D;
    nullSrvDesc.Shader4ComponentMapping = D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING;
    nullSrvDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
    nullSrvDesc.Texture2D.MipLevels = 1;
    nullSrvDesc.Texture2D.MostDetailedMip = 0;
    nullSrvDesc.Texture2D.ResourceMinLODClamp = 0.0f;

    m_device->CreateShaderResourceView(nullptr, &nullSrvDesc, m_cbvSrvHeap.Get(), 0);

    m_device->CreateShaderResourceView(nullptr, &nullSrvDesc, m_cbvSrvHeap.Get(), 1);
}
...
{
    const UINT subresourceCount = texDesc.DepthOrArraySize * texDesc.MipLevels;
    UINT64 uploadBufferSize = GetRequiredIntermediateSize(pRes.Get(), 0,
    subresourceCount);
    ThrowIfFailed(m_device->CreateCommittedResource(
    &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_UPLOAD),
    D3D12_HEAP_FLAG_NONE,
    &CD3DX12_RESOURCE_DESC::Buffer(uploadBufferSize),
    D3D12_RESOURCE_STATE_GENERIC_READ,
    nullptr,
    IID_PPV_ARGS(&m_textureUploads[i])));

    // Copy data to the intermediate upload heap and then schedule a copy
    // from the upload heap to the Texture2D.
    D3D12_SUBRESOURCE_DATA textureData = {};
    textureData.pData = pAssetData + tex.Data->Offset;
    textureData.RowPitch = tex.Data->Pitch;
    textureData.SlicePitch = tex.Data->Size;

    UpdateSubresources(commandList.Get(), pRes.Get(), m_textureUploads[i].Get(), 0, 0,
    subresourceCount, &textureData);
    commandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(pRes.Get(),
    D3D12_RESOURCE_STATE_COPY_DEST, D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE));
}

// Describe and create an SRV.
D3D12_SHADER_RESOURCE_VIEW_DESC srvDesc = {};
srvDesc.ViewDimension = D3D12_SRV_DIMENSION_TEXTURE2D;
srvDesc.Shader4ComponentMapping = D3D12_DEFAULT_SHADER_4_COMPONENT_MAPPING;
srvDesc.Format = tex.Format;
srvDesc.Texture2D.MipLevels = tex.MipLevels;
srvDesc.Texture2D.MostDetailedMip = 0;
srvDesc.Texture2D.ResourceMinLODClamp = 0.0f;
m_device->CreateShaderResourceView(&m_textures[i], &srvDesc, m_cbvSrvHeap.Get(), i + 2)
;
```

So we see that we can now actually remove 5 lines of code in total, which dealt with defining and updating the CPU descriptor value in the loop. Instead, our calls to CREATESHADERRESOURCEVIEW()) now need two parameters to pass the correct descriptor slot, where previously we only had one.

LOC: -5

MC: -3
LPL: +3

Vertexbuffers make use of a subclass of MANAGEDRESOURCE. As was already stated earlier with the variable declarations, the view description is included in the object now. This makes them very easy to use.

Before:

```
// Initialize the vertex buffer view.
m_vertexBufferView.BufferLocation = m_vertexBuffer->GetGPUVirtualAddress();
m_vertexBufferView.SizeInBytes = SampleAssets::VertexDataSize;
m_vertexBufferView.StrideInBytes = SampleAssets::StandardVertexStride;
```

After:

```
// Initialize the vertex buffer view.
m_vertexBuffer.view.BufferLocation = pVertexBuffer->GetGPUVirtualAddress();
m_vertexBuffer.view.SizeInBytes = SampleAssets::VertexDataSize;
m_vertexBuffer.view.StrideInBytes = SampleAssets::StandardVertexStride;

// Manage this object
m_vertexBuffer.Initialize(&m_residencyManager, pVertexBuffer.Get(), m_vertexBuffer.view
.SizeInBytes, 0);
```

So three lines only need to be slightly modified so that the VIEW member of the MANAGEDVERTEXBUFFER is used, rather than a separate variable. And one line needs to be added to call the INITIALIZE() method to start tracking the object with the RESIDENCYMANAGER. Since the size must already be specified in the view, this call is now trivial, as we do not need the device to calculate the allocation size.

LOC: +1
MC: +1
LPL: +4

Indexbuffers are analogous to vertexbuffers.

Before:

```
// Initialize the index buffer view.
m_indexBufferView.BufferLocation = m_indexBuffer->GetGPUVirtualAddress();
m_indexBufferView.SizeInBytes = SampleAssets::IndexDataSize;
m_indexBufferView.Format = SampleAssets::StandardIndexFormat;
```

After:

```
// Initialize the index buffer view.
m_indexBuffer.view.BufferLocation = pIndexBuffer->GetGPUVirtualAddress();
m_indexBuffer.view.SizeInBytes = SampleAssets::IndexDataSize;
m_indexBuffer.view.Format = SampleAssets::StandardIndexFormat;

// Manage this object
m_indexBuffer.Initialize(&m_residencyManager, pIndexBuffer.Get(), m_indexBuffer.view.
SizeInBytes, 0);
```

LOC: +1
MC: +1
LPL: +4

The final change required to get the managed Proxy objects working, is to set up the root descriptor table.

Before:

```
pSceneCommandList->SetGraphicsRootDescriptorTable(0, cbvSrvHandle);
```

After:

```
pSceneCommandList->SetGraphicsRootDescriptorTable(0, m_cbvSrvHeap.Get(), nullSrvCount +
 drawArgs.DiffuseTextureIndex);
```

Here we see the same heap + slot index addressing for the descriptors as mentioned earlier, with the SETRESOURCE() call to register resources on the descriptor heap. So instead of one handle parameter, there are now two parameters required to bind resource views to the descriptor table.

LPL: +1

This brings the total to:
LOC: +2
MC: +1
LPL: +26
CC: 0

So, making use of the Proxy objects effectively added only two lines of code to the total (as a result of various operations actually requiring less code than the original DirectX 12 interface). And only one extra line of message chain (calls that are dependent on previous calls). The cyclomatic complexity is unchanged, because all the code is just straightforward method calls, no actual decision logic, branching, looping or anything else is required. The biggest increment is in the parameter count. The total number of +26 looks big, but this is spread out over all the method calls required. Per method call, it is only one extra parameter. These are usually quite straightforward. One common change is that the RESIDENCYMANAGER instance has to be passed to certain calls. Another common change is that CPU handles have been replaced by pairs of descriptor heap reference + slot index. This change is equivalent to using arrays with indices rather than using direct pointers to memory locations. This may lead to increased code metrics, but that does not necessarily mean the code is more complex.

The most complex part is the loading of the textures. This process is already complex to begin with, since it requires uploading from a temporary resource to the target resource by executing a COMMANDLIST. The example code performs this in-place with a loop in the LOADASSETS()) method, but for a more advanced application, a simple helper function can be created, which can hide the complexity from the rest of the application.

The trade-off for the small increase in total code size is that we now have automated residency management of all our code. And since this is done at the same abstraction level as the native DirectX 12 API, the design of the application is not affected in any way. All the assumptions about concurrency and synchronization still hold. The application can still build multiple command lists in parallel using multiple threads, and it can still execute command lists on the GPU while preparing other command lists on the CPU at the same time.

Performance The D3D12 Multithreading Sample has a simple performance measurement built-in: it records the time spent between the start and end of rendering a frame with a high-performance counter. It accumulates these per-frame measurements in a counter that is displayed once every N frames. By default this is set to N = 200 frames. It displays the average elapsed CPU time per frame in milliseconds in the title bar of the application. By default, the sample runs in a small window of 1280x720 resolution, and makes no use of antialiasing. This means that the amount of pixels the GPU has to process is relatively small, and the performance will mainly be CPU-limited. This allows us to measure the difference in CPU usage between the different versions more accurately. The GPU-workload will remain identical for all versions, as no changes have been made to what is rendered, or how it is rendered.

I have added some code to log these timings to a file, and left the system running with power management disabled for a few minutes, to collect the timing data. I created a fully managed version, and a 'dummy' version, which performs all the required management in the Proxy objects, but instead of calling ResidencyManager::ExecuteCommandLists(), it calls the regular unmanaged ID3D12CommandQueue::ExecuteCommandLists(). By bypassing the ResidencyManager, we can measure only the overhead of the Proxy objects, without any added overhead that the ResidencyManager may add.

The specifications of the test system used are:

- Intel Core i7 860 processor
- NVIDIA GeForce GTX970 GPU
- 8 GB of system memory
- 4 GB of video memory
- 4 TB of storage
- Windows 11 Pro Operating System

All code was compiled for the x64 architecture, using Visual Studio 2022.

The last 10 frame timings from the runs of the original, dummy and managed version are shown in Table 5.1.

	Original	Dummy	Managed
	0.4967	0.5509	0.6091
	0.5091	0.5547	0.6446
	0.5063	0.5473	0.6381
	0.5501	0.5495	0.6141
	0.5587	0.5512	0.6504
	0.5404	0.5533	0.6179
	0.5005	0.5497	0.6690
	0.4889	0.5522	0.6599
	0.4986	0.5518	0.6617
	0.5442	0.5622	0.6324
Average:	0.51935	0.55228	0.63972

Table 5.1: Frame times of original, dummy and managed residency versions of the D3D12 Multithreading sample.

This shows that the Proxy layer requires about 6% extra CPU overhead, and the entire managed layer requires about 23% extra CPU overhead. So most of the time is spent in the RESIDENCY MANAGER, not in the Proxy objects. The added total overhead seems relatively high. However, since DirectX 12 is very efficient, the original frametime was extremely low, at 0.5 ms per frame. That would allow a theoretical framerate of 1000 / 0.5 ms = 2000 fps. In absolute terms the managed layer adds 0.12 ms extra per frame. The theoretical framerate is still about 1000 / 0.64 ms = 1562.5 fps. To put the timings into a realistic perspective, we need to look at a more acceptable framerate (the framerate will be much lower when rendering at a larger resolution and using antialiasing). For example, at 60 fps, a single frame is displayed for 16.67 ms. The 0.12 ms of overhead for the managed layer is 0.7% of that. So at 60 fps, the added overhead is negligible, because anything under 1% can be considered to be within the margin of error in terms of frame time measurements.

6

Concurrency Management

This chapter will look into the concurrency problems relating to the usage of multiple queues with command lists, alongside tasks running on the CPU.

6.1. Problem analysis

On top of the regular concurrency issues in conventional multithreaded applications on the CPU, DirectX 12 adds concurrency on the GPU. This means that all possible combinations of synchronization between CPU and GPU tasks can occur: CPU-to-CPU, CPU-to-GPU and GPU-to-GPU.

DirectX 12 offers only a single synchronization primitive for task concurrency, the ID3D12Fence. This is a special type of fence that can be read and signaled by both the CPU and the GPU. The fence has an internal 64-bit unsigned integer value. The CPU can read the current fence value or signal a specific 64-bit value. On the GPU side, the fence can be used on a per-command queue basis. A queue can either be instructed to wait for a fence to reach a specific 64-bit value, or to signal a specific 64-bit value after the previous operations in the queue have completed.

On the CPU side, reading the current value of a fence can be used for a simple polling loop to wait for the GPU. However, a more sophisticated mechanism is also provided. A fence object can be instructed to set a Windows event object when a specified 64-bit value has been reached. This method allows for more advanced synchronization on the CPU side, using threads or thread pools. Event objects can take advantage of the OS scheduler, and do not need to use spinlocks to wait for the GPU. Instead, a thread that is waiting on an event can be taken out of the active thread list, and will not be scheduled until the event is set. This results in very little overhead, leaving the CPU free to perform other tasks.

6.2. CONCURRENCY MANAGEMENT

Chapter 4 briefly explained that a DirectX 12 application essentially runs on an asymmetric multiprocessing system. There is the CPU, which is generally a multicore CPU, capable of running multiple threads in parallel. And there is the GPU, which can run multiple command queues in parallel. In a DirectX 12 application, synchronization is required not only between CPU tasks/threads, but between all permutations of CPU and GPU tasks:

- CPU waiting on CPU
- CPU waiting on GPU
- GPU waiting on CPU
- GPU waiting on GPU

As mentioned, DirectX 12 offers some basic synchronization on the GPU's ID3D12COMMANDQUEUES via ID3D12FENCE. The ID3D12FENCE also supports signaling an EVENT object for synchronization with the CPU.

The Task Parallel Library in .NET, mentioned in Chapter 4, introduces the concept of 'continuation tasks', which would be very interesting to extend to the GPU. However, there is no official support for .NET in the DirectX 12 SDK. The only programming environment that Microsoft supports with the DirectX 12 SDK is native C++. As such, we must first implement our own version of continuation tasks on the CPU side. So let us first look at how it works in .NET.

Threadpools and Task Parallel Library in .NET The basis of running code in parallel on modern multicore/multiprocessor systems is multithreading. The OS supports the creation and scheduling of threads. On top of the basic mechanism for thread scheduling, an abstraction layer can be implemented to support 'tasks': simple workloads, usually just a function or method, which can be executed on one of the available threads of the threadpool. This makes programming both simpler and more efficient, because there is no need to create and destroy threads, or keep track of how many threads there are active at a time.

The Task Parallel Library in .NET also offers a mechanism known as 'continuation tasks'. A task is an object that offers various methods. One of which is CONTINUEWITH(TASK). This allows you to queue up one or more tasks, to be started automatically when a previous task ends. For example:

```
    Task A = new Task(() => Console.WriteLine("Task␣A"));
    Task B = new Task(() => Console.WriteLine("Task␣B"));

    A.ContinueWith(B);
    A.Run();
```

This will start task B as soon as task A is complete. So the output is always serialized: Task A will always complete before task B.

This is a very interesting way to synchronize multiple independent tasks. The concept could be extended to more than just CPU-tasks, and generalized across asymmetric multiprocessing systems such as the CPU+GPU combination that is used in a DirectX 12 environment. If CPU tasks can wait for GPU tasks, and vice versa, that can make programming DirectX 12 synchronisation simpler.

Implementing tasks in C++ on Windows Windows supplies us with a basic lowlevel threadpool, which offers the basic building blocks to create CPU tasks:

- SubmitThreadpoolWork() to start a task immediately
- SetThreadpoolWait() to start a task when the specified Event is set
- SetEventWhenCallbackReturns() to set an Event when a task is complete

This allows us to create a simple continuation task system based around Events. Each task must contain its own Event that signals completion. When the task is started, the SetEventWhenCallbackReturns() will be called with its completion event. Then the next task can be started with SetThreadpoolWait() where the completion event of the previous task is passed. The completion Event can also be used with WaitForSingleObject() to implement a Task::Wait() method, to wait for completion of a task without starting a new one.

Once this basic continuation task functionality works on the CPU, it is straightforward to start a CPUTask after a GPU fence is reached, with ID3D12Fence::SetEventOnCompletion(). The GPUTask must create a fence and a completion event, tell the command queue to signal that fence after it has completed executing the given command list, and tell the fence to signal the completion event. This event can then be used to start a CPUTask via SetThreadpoolWait(), just like before.

Going the other way is slightly more complicated. A GPUTask consists of adding one or more command lists to a command queue (which is a fire-and-forget operation), and setting up the fence for completion of these command lists. This has to be done with the CPU. The GPU will execute the command lists in its command queues independently from the CPU. This means that a GPUTask has two components:

1. The CPU-code to add the command list to the queue
2. The GPU-code that is encoded in the command list

Both components require synchronization. You need to be able to add command lists to a queue in a specific order. And you need to be able to wait for completion of a command list by the GPU, before starting a new one. This means that the semantics for a GPUTask are slightly more complicated than for a CPUTask.

The GPUTask::ContinueWith() will continue the next task when the GPU has completed the command lists of the current task, namely when the ID3D12CommandQueue has set the fence, and the fence in turn has triggered the event via the ID3D12Fence::SetEventOnCompletion() mechanism. Where CPUTask::Run() will queue a task on the threadpool and return immediately, the equivalent GPUTask::Run() will add the command list(s) to the GPU queue, and then return. This means that a call to GPUTask::Run() will execute on the current thread, and may have some additional overhead, because it will perform all resource management at this point, before returning. However, this choice makes the GPUTask very simple and predictable to use: since GPUTask::Run() is a synchronous call, you always know that the command list(s) will be added when you make the call, and that the queueing of the command lists is complete when it returns.

Of course it is still possible to run this code on a separate thread by wrapping the call to GPUTask::Run() in another CPUTask. The most common way to use a GPUTask however is probably to prepare a number of command lists in separate CPUTasks. When all tasks have completed, the command lists can be executed in a single GPUTask.

When a GPUTask is started from the ContinueWith() mechanism of another task, the GPUTask::Run() method will be called implicitly. In this situation it can not be known when exactly the command lists will be queued. We would have to perform a Wait() to make sure that the GPUTask has completed. But that would defeat the ability to run code concurrently on the CPU and the GPU, because the CPU would then wait until the GPU has completed all work, rather than just until the command lists have been added to the queue, so that new lists can be added after them, in the correct order.

For that purpose, the GPUTask interface adds an additional mechanism. It sets an additional internal event 'queued', after the command lists have been queued, and the fence and done-event have been set up. So at this point we know that new command lists will be added after the lists of this GPUTask, and we can guarantee their execution order.

The GPUTask::WaitQueued() method allows you to wait for this event. At this point you know that new command lists will be added after the command lists of this task, when using the same queue.

The GPUTask::ContinueWithQueued() method will start a new ITask when the queued-event is triggered.

With these two methods, it is possible to efficiently add command lists to a queue in a strict order, without having to wait for every command list to complete execution on the GPU.

Passing code to a CPUTask is done via the following type:

```
typedef std::function<void()> WorkLoad;
```

A WorkLoad is a function with no parameters, and no return value. It supports the use of lambda functions (also known as closures, anonymous functions or implicit functions), and capture of variables. This allows you to pass information to the task, so that it can access application state.

Passing code to a GPUTask is very similar to a call to ExecuteCommandLists(): a GPUTask needs to be created with a reference to a device, a command queue, and an array of 1 or more command lists:

```
GPUTask(ManagedDevice* pDevice, ManagedCommandQueue* pQueue,
    _In_  UINT NumCommandLists,
    _In_reads_(NumCommandLists)  ManagedCommandList* const* ppCommandLists);
```

So the GPU code is simply contained in the command lists passed to the GPU Task.

6.3. EXAMPLE

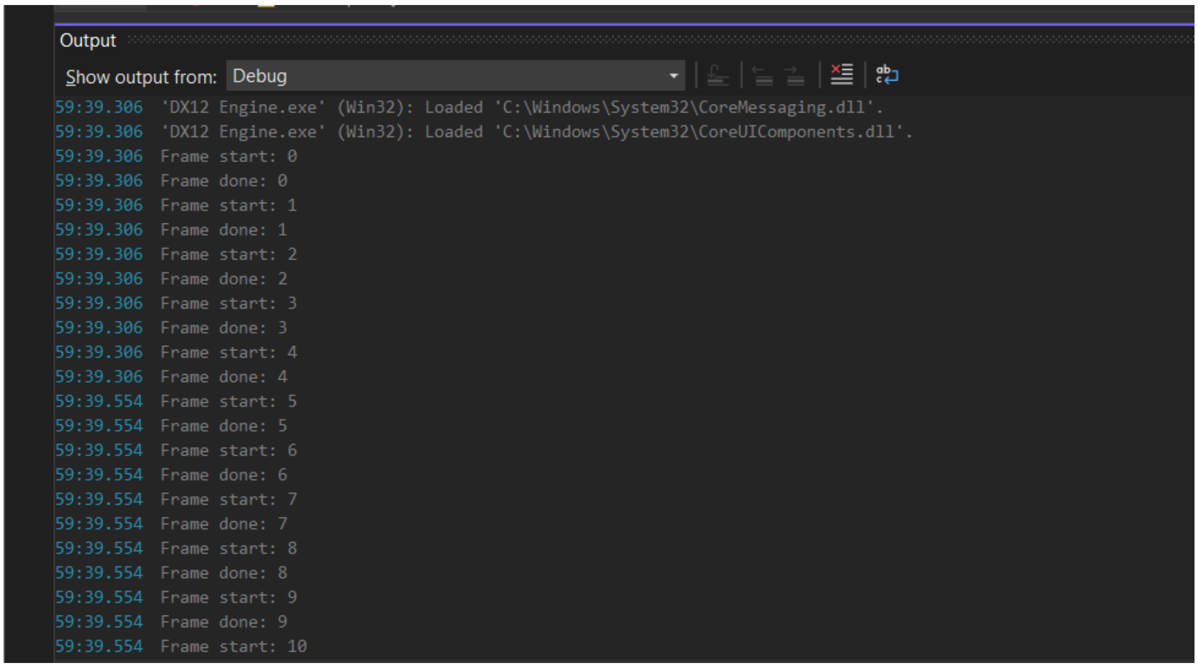
Here is a small proof-of-concept:

```
void CEngine::DoStart()
{
    WCHAR buf[512];

    wsprintf(buf, L"Frame␣start:␣%d\n", m_frame);

    OutputDebugString(buf);

    // Record all the commands we need to render the scene into the command list.
    PopulateCommandList();
}

// Render the scene.
void CEngine::Render()
{
    // Execute the command list.
    CPUTask start([=]
    {
        DoStart();
    });

    ManagedCommandList* ppCommandLists[] = { m_context.commandListSet.commandList.Get()
     };
    GPUTask task(m_context.pDevice.Get(), m_commandQueue.Get(), _countof(ppCommandLists
    ), ppCommandLists);

    CPUTask done([=, i = m_frame]
    {
        WCHAR buf[512];

        wsprintf(buf, L"Frame␣done:␣%d\n", i);

        OutputDebugString(buf);
    });

    start.ContinueWith(&task);
    task.ContinueWith(&done);
    start.Run();

    task.WaitQueued();

    // Present the frame.
    ThrowIfFailed(m_swapChain->Present(1, 0));

    m_frameIndex = m_swapChain->GetCurrentBackBufferIndex();

    m_frame++;

    ITask* tasks[] = { &start, &task, &done };

    ITask::WaitAll(_countof(tasks), tasks, INFINITE);
}
```

What we see here in the RENDER() method is that a chain of three tasks is created:

- A CPU task that runs the DOSTART() method, which fills the command list for the next frame.
- A GPU task which executes the populated command list.
- A CPU task that simply outputs some debug text, so we can verify that it is being called, and it is called before the next frame starts.

For the first two tasks it is important that they run in the correct order. So the GPU task can not start before the command list is complete. Otherwise the DirectX 12 runtime will signal an exception that the command list you are trying to execute has not been closed yet. The third task does not affect the execution and rendering of the application itself, but is merely confirmation that a CPU task can also be started after a GPU task. Indirectly we can also verify that the next frame is not started before the current frame has been completed and all three tasks have finished (see the output in Figure 6.1).



```
59:39.306  'DX12 Engine.exe' (Win32): Loaded 'C:\Windows\System32\CoreMessaging.dll'.
59:39.306  'DX12 Engine.exe' (Win32): Loaded 'C:\Windows\System32\CoreUIComponents.dll'.
59:39.306  Frame start: 0
59:39.306  Frame done: 0
59:39.306  Frame start: 1
59:39.306  Frame done: 1
59:39.306  Frame start: 2
59:39.306  Frame done: 2
59:39.306  Frame start: 3
59:39.306  Frame done: 3
59:39.306  Frame start: 4
59:39.306  Frame done: 4
59:39.554  Frame start: 5
59:39.554  Frame done: 5
59:39.554  Frame start: 6
59:39.554  Frame done: 6
59:39.554  Frame start: 7
59:39.554  Frame done: 7
59:39.554  Frame start: 8
59:39.554  Frame done: 8
59:39.554  Frame start: 9
59:39.554  Frame done: 9
59:39.554  Frame start: 10
```

Figure 6.1: Debug output of CPU Tasks.

In this code, an interesting part is just before the call to PRESENT(). I use the WAITQUEUED() method on the GPUTask. This ensures that the start-task has been completed, the GPU task has been started, and has performed the EXECUTECOMMANDLISTS(), before the call to PRESENT() is made. Intuitively one might think that one should wait for the GPU Task to finish before calling PRESENT(). However, this demonstrates that PRESENT() is a special case where DirectX 12 will perform the required synchronization itself (the (PRESENT() call is actually performed by the DXGI layer, which is also shared with DirectX 10 and 11, which might explain this behaviour).

At the end of the RENDER() method, a WAITALL() call is done to make sure that all tasks have completed before returning. Note that we already waited for the start task earlier. So this verifies that waiting for a task that is already complete will behave as expected. In this particular case, this call could actually be removed. Namely, in the destructor of a task, there is always a WAIT() call if the task is in a running state. Since these tasks are created as local variables, they will go out of scope when the function returns, and as such, their destructor is called, and a WAIT() would be performed if they had not finished already.

Another interesting part is that both the start- and done-tasks make use of an implicitly defined function in C++. This is to mimic how tasks are often used in C#. The done-task is defined entirely inline. The start-task is merely a call to the CENGINE::DOSTART() method. So like in C#, it is possible to access the object's state and methods from within the implicit function.

DirectX 12 samples The D3D12 MULTISAMPLING sample used earlier is one of the more elaborate samples. It performs multiple passes (lighting and shadow passes), and it distributes the rendering over multiple threads. We will attempt to modify it to make use of the new task library. We will build on the managed version that was created in the previous chapter. By default it starts 3 worker threads, which are synchronized with events from the main thread. The workflow is shown in Figure 6.2. The synchronization between

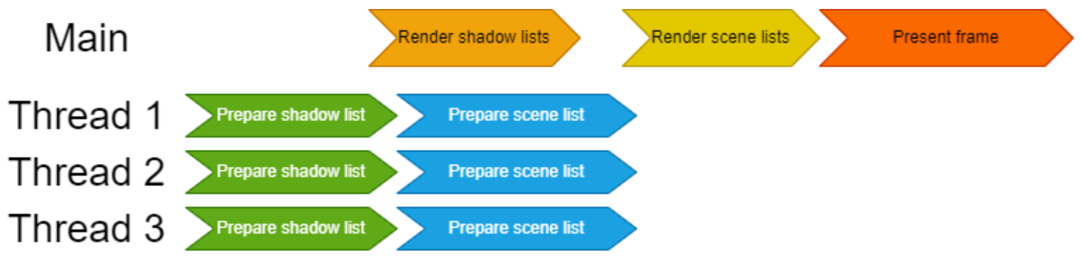


Figure 6.2: D3D12 Multithreading workflow.

the main thread and the worker threads is shown in Figure 6.3. The 3 threads are created once at startup, and will loop until the application ends. Our task-based approach works differently, as it makes use of a threadpool. So the actual threads are abstracted away by the underlying implementation, and we do not directly control when threads are started or stopped, or on which thread our task will actually execute.

So we cannot create a version with the new task library that is technically equivalent to the original version. But we can create a version that is functionally equivalent. Because of the different usage of threads and synchronization, performance may vary.

Logically each worker thread performs two tasks one after another: the preparing of a command list for the shadow pass, and the preparing of a command list for the scene pass. After each task, there is a synchronization point where the main thread has to wait for all threads to finish their command list, before executing them.

This translates easily to our CPUTASK objects: we can create three shadow tasks and three scene tasks. Then we can create a list for each set of tasks and perform a WAITALL() on the tasks to synchronize with executing the lists. This synchronization pattern is shown in Figure 6.4.

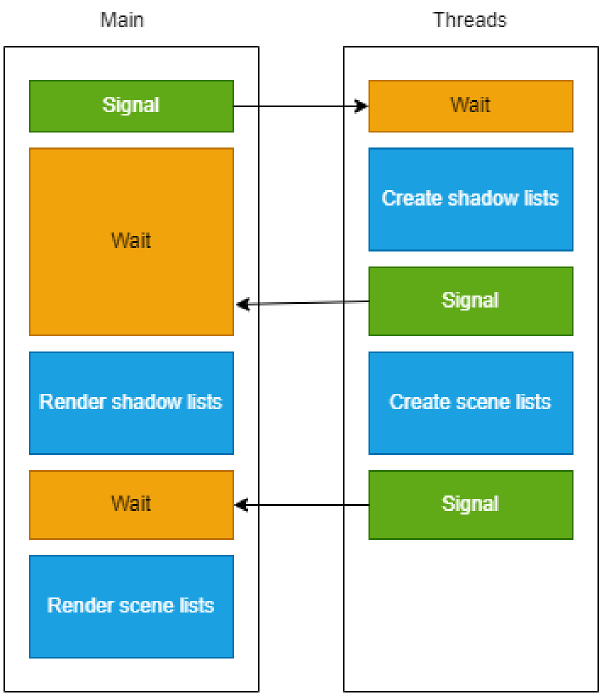


Figure 6.3: D3D12 Multithreading synchronization.

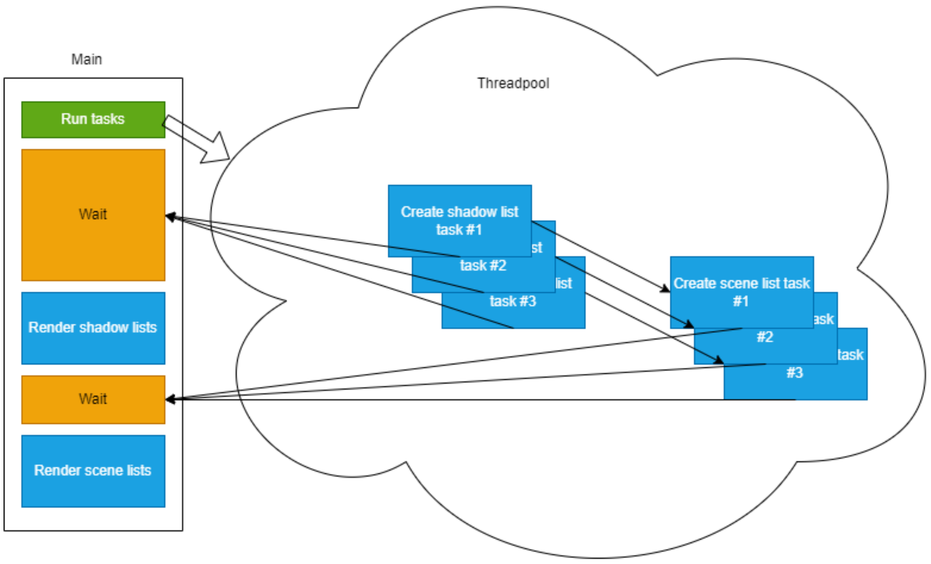


Figure 6.4: D3D12 Multithreading tasks.

Correctness The correctness of our solution depends on whether all tasks are executed, and whether they are executed in the correct order, with proper synchronization between the different parts of the application. In general it is trivial to verify this sample visually, as each task contributes to part of what is seen on the screen. If one or more shadow tasks are not executed correctly, then parts of the shadow will not appear. And if one or more scene tasks are not executed correctly, some objects will be missing from the scene. Aside from that, each task fills a command list. So if an attempt is made to execute a command list before the task has finished, the command list will still be in the OPEN state, and the DirectX 12 debug layer will report an exception and close the application. Therefore, if the application does not close unexpectedly, and it is visually identical to the original sample, it is correct.

Ease of use One change that could be made easily, is during initialization. Textures and various other resources have to be set up once at the start of the application. This requires executing some command lists, which have to complete before the main rendering loop starts. This is a good case for using the GPUTask.

Before:

```
ID3D12CommandList* ppCommandLists[] = { commandList.Get() };
m_commandQueue->ExecuteCommandLists(_countof(ppCommandLists), ppCommandLists);

// Create frame resources.
for (int i = 0; i < FrameCount; i++)
{
    m_frameResources[i] = new FrameResource(m_device.Get(), m_pipelineState.Get(),
    m_pipelineStateShadowMap.Get(), m_dsvHeap.Get(), m_cbvSrvHeap.Get(), &m_viewport, i
    );
    m_frameResources[i]->WriteConstantBuffers(&m_viewport, &m_camera, m_lightCameras,
    m_lights);
}
m_currentFrameResourceIndex = 0;
m_pCurrentFrameResource = m_frameResources[m_currentFrameResourceIndex];

// Create synchronization objects and wait until assets have been uploaded to the GPU.
{
    ThrowIfFailed(m_device->CreateFence(m_fenceValue, D3D12_FENCE_FLAG_NONE,
    IID_PPV_ARGS(&m_fence)));
    m_fenceValue++;

    // Create an event handle to use for frame synchronization.
    m_fenceEvent = CreateEvent(nullptr, FALSE, FALSE, nullptr);
    if (m_fenceEvent == nullptr)
    {
        ThrowIfFailed(HRESULT_FROM_WIN32(GetLastError()));
    }

    // Wait for the command list to execute; we are reusing the same command
    // list in our main loop but for now, we just want to wait for setup to
    // complete before continuing.

    // Signal and increment the fence value.
    const UINT64 fenceToWaitFor = m_fenceValue;
    ThrowIfFailed(m_commandQueue->Signal(m_fence.Get(), fenceToWaitFor));
    m_fenceValue++;

    // Wait until the fence is completed.
    ThrowIfFailed(m_fence->SetEventOnCompletion(fenceToWaitFor, m_fenceEvent));
    WaitForSingleObject(m_fenceEvent, INFINITE);
}
```

After:

```
ManagedCommandList* ppCommandLists[] = { commandList.Get() };
GPUTask task(m_device.Get(), m_commandQueue.Get(), _countof(ppCommandLists),
ppCommandLists);
task.Run();

// Create frame resources.
for (int i = 0; i < FrameCount; i++)
```

```
{
    m_frameResources[i] = new FrameResource(m_device.Get(), &m_residencyManager,
    m_pipelineState.Get(), m_pipelineStateShadowMap.Get(), m_dsvHeap.Get(),
    m_cbvSrvHeap.Get(), &m_viewport, i);
    m_frameResources[i]->WriteConstantBuffers(&m_viewport, &m_camera, m_lightCameras,
    m_lights);
}
m_currentFrameResourceIndex = 0;
m_pCurrentFrameResource = m_frameResources[m_currentFrameResourceIndex];

// Create synchronization objects.
{
    ThrowIfFailed(m_device->CreateFence(m_fenceValue, D3D12_FENCE_FLAG_NONE,
    IID_PPV_ARGS(&m_fence)));
    m_fenceValue++;

    // Create an event handle to use for frame synchronization.
    m_fenceEvent = CreateEvent(nullptr, FALSE, FALSE, nullptr);
    if (m_fenceEvent == nullptr)
    {
        ThrowIfFailed(HRESULT_FROM_WIN32(GetLastError()));
    }
}
```

We can see that the code is indeed considerably shorter now. We no longer need to manually set up a fence to signal on the command-queue, connect an event to it, and then wait for that event to occur. All this is done automatically when the (Wait()) method of the GPUTask is executed. Note that in this case we do not actually call it explicitly. The destructor takes care of it automatically, as explained earlier.

LOC: -4
MC: -3

Note also that the fence and fence-event are still created in the GPUTask-based version of the code. This is not required for this specific bit of code. However, the same fence and event are also used in other parts of the application, so they could not be removed entirely, without rewriting parts of the application that are beyond the scope of this modification. Otherwise we could reduce the Cyclomatic Complexity by 1, and reduce relevant the lines of code by about 5.

We will now look at the variables that are declared for the threads and synchronization.

Before:

```
// Synchronization objects.
HANDLE m_workerBeginRenderFrame[NumContexts];
HANDLE m_workerFinishShadowPass[NumContexts];
HANDLE m_workerFinishedRenderFrame[NumContexts];
HANDLE m_threadHandles[NumContexts];
...
struct ThreadParameter
{
    int threadIndex;
};
ThreadParameter m_threadParameters[NumContexts];

void WorkerThread(int threadIndex);
```

After:

```
// Synchronization objects.
```

```
CPUTask m_shadowTasks[NumContexts];
CPUTask m_sceneTasks[NumContexts];
```

As we can see here, the standard way of using threads and events for synchronization in the WIN32API is quite cumbersome. We need three handles, for the 3 different states in a frame (begin, finish shadow pass and finish frame). Then we need another handle for each thread. We also need a function to pass as our THREADPROC when starting each thread, in this case WORKERTHREAD(). And we need a way to pass data to this THREADPROC, in this case the THREADPARAMETER struct.

Using CPUTASKS however is very straightforward. We only need to define the tasks themselves. They already encapsulate all the events and other data required. Because we are able to use lambda functions, we do not need a specific THREADPROC either, and we do not need a workaround like the THREADPARAMETER struct to pass data to it.

LOC: -6

Now for the LOADCONTEXTS() method which sets up the threads/tasks for use.

Before:

```
void D3D12Multithreading::LoadContexts()
{
    struct threadwrapper
    {
        static unsigned int WINAPI thunk(LPVOID lpParameter)
        {
            ThreadParameter* parameter = reinterpret_cast<ThreadParameter*>(lpParameter
            );
            D3D12Multithreading::Get()->WorkerThread(parameter->threadIndex);
            return 0;
        }
    };

    for (int i = 0; i < NumContexts; i++)
    {
        m_workerBeginRenderFrame[i] = CreateEvent(
            NULL,
            FALSE,
            FALSE,
            NULL);

        m_workerFinishedRenderFrame[i] = CreateEvent(
            NULL,
            FALSE,
            FALSE,
            NULL);

        m_workerFinishShadowPass[i] = CreateEvent(
            NULL,
            FALSE,
            FALSE,
            NULL);

        m_threadParameters[i].threadIndex = i;

        m_threadHandles[i] = reinterpret_cast<HANDLE>(_beginthreadex(
            nullptr,
            0,
            threadwrapper::thunk,
            reinterpret_cast<LPVOID>(&m_threadParameters[i]),
            0,
            nullptr));
    }
}
```

After:

```
void D3D12Multithreading::LoadContexts()
{
    for (int i = 0; i < NumContexts; i++)
    {
        m_shadowTasks[i].SetWorkLoad([=]
        {
            DoShadowPass(i);
        }
        );

        m_sceneTasks[i].SetWorkLoad([=]
        {
            DoScenePass(i);
        }
        );
    }
}
```

This is again a lot less code that is required. The code is also more straightforward now. The old way of having to use a special helper-struct to be able to access class parameters from another thread is very clumsy with the use of the THREADWRAPPER struct here, to THUNK the THREADPROC down to the actual WORKERTHREAD() method of the class. In our case, the lambda function automatically captures the local scope of the class, and we can call the methods directly.

LOC: -23
LPL: -25

The old code also needs to explicitly clean up the handles that were created:

```
// Close thread events and thread handles.
for (int i = 0; i < NumContexts; i++)
{
    CloseHandle(m_workerBeginRenderFrame[i]);
    CloseHandle(m_workerFinishShadowPass[i]);
    CloseHandle(m_workerFinishedRenderFrame[i]);
    CloseHandle(m_threadHandles[i]);
}
```

That is another few lines that can be scrapped, because the CPUTasks have their own destructor, which takes care of that. Since we have declared them statically, we do not need to call the destructor explicitly. It will be called automatically when the containing class goes out of scope.

LOC: -5
LPL: -4
CC: -1

Now to look at the actual code. The WORKERTHREAD() method looks like this:

```
// Worker thread body. workerIndex is an integer from 0 to NumContexts
// describing the worker's thread index.
void D3D12Multithreading::WorkerThread(int threadIndex)
{
    while (threadIndex >= 0 && threadIndex < NumContexts)
    {
        // Wait for main thread to tell us to draw.

        WaitForSingleObject(m_workerBeginRenderFrame[threadIndex], INFINITE);

        DoShadowPass(threadIndex);

        // Submit shadow pass.
        SetEvent(m_workerFinishShadowPass[threadIndex]);

        DoScenePass(threadIndex);

        // Tell main thread that we are done.
        SetEvent(m_workerFinishedRenderFrame[threadIndex]);
    }
}
```

This is where one part of the synchronization happens. The threads are constructed as loops, so they can be re-used for every frame, saving the overhead of having to create new threads everytime. But that means each thread has to wait for an event at every iteration. Then it performs the shadow pass, signals an event, does the scene pass, and signals another event. This whole method is no longer required, as all the synchronization will be moved into the main routine.

LOC: -7
LPL: -6
CC: -1

And finally the actual main rendering routine.

Before:

```
BeginFrame();

for (int i = 0; i < NumContexts; i++)
{
    SetEvent(m_workerBeginRenderFrame[i]); // Tell each worker to start drawing.
}

MidFrame();
EndFrame();

WaitForMultipleObjects(NumContexts, m_workerFinishShadowPass, TRUE, INFINITE);

// You can execute command lists on any thread. Depending on the work
// load, apps can choose between using ExecuteCommandLists on one thread
// vs ExecuteCommandList from multiple threads.
m_commandQueue->ExecuteCommandLists(NumContexts + 2, m_pCurrentFrameResource->
m_batchSubmit); // Submit PRE, MID and shadows.

WaitForMultipleObjects(NumContexts, m_workerFinishedRenderFrame, TRUE, INFINITE);

// Submit remaining command lists.
m_commandQueue->ExecuteCommandLists(_countof(m_pCurrentFrameResource->m_batchSubmit) -
NumContexts - 2, m_pCurrentFrameResource->m_batchSubmit + NumContexts + 2);
```

After:

```
for (int i = 0; i < NumContexts; i++)
{
    m_shadowTasks[i].Reset();
    m_sceneTasks[i].Reset();
    m_shadowTasks[i].ContinueWith(&m_sceneTasks[i]);
}

BeginFrame();

for (int i = 0; i < NumContexts; i++)
    m_shadowTasks[i].Run();

MidFrame();
EndFrame();

ITask* tasks[NumContexts];

for (int i = 0; i < NumContexts; i++)
    tasks[i] = &m_shadowTasks[i];

ITask::WaitAll(_countof(tasks), tasks, INFINITE);

// You can execute command lists on any thread. Depending on the work
// load, apps can choose between using ExecuteCommandLists on one thread
// vs ExecuteCommandList from multiple threads.
m_commandQueue->ExecuteCommandLists(NumContexts + 2, m_pCurrentFrameResource->
m_batchSubmit); // Submit PRE, MID and shadows.

for (int i = 0; i < NumContexts; i++)
    tasks[i] = &m_sceneTasks[i];

ITask::WaitAll(_countof(tasks), tasks, INFINITE);

// Submit remaining command lists.
m_commandQueue->ExecuteCommandLists(_countof(m_pCurrentFrameResource->m_batchSubmit) -
NumContexts - 2, m_pCurrentFrameResource->m_batchSubmit + NumContexts + 2);
```

As we can see, the synchronization has moved into the main method here. At the start of each frame, we first RESET() our tasks. We also need to (re-)set the CONTINUEWITH relation at every frame. We tell each shadow task to continue with its respective scene task.

Then, in the old code the beginRenderFrame event was set to start the next iteration for each thread. Now we call RUN()) to start each task.

In the old code, a WAITFORMULTIPLEOBJECTS() call was performed to wait for all threads to signal the finishShadowPass event. In the new code, we can simply build a list of our shadow tasks, and perform an ITASK::WAITALL() on these tasks.

The same change is done from the finishedRenderFrame events to the scene tasks. So this is where we have to trade in some of our gains in the earlier parts.

LOC: +9
LPL: -2
CC: +3

On the whole, we still require significantly less code, and also less complicated code, to reach the same functionality of having the rendering spread out over multiple threads.

Total:
LOC: -36
MC: -3
LPL: -37
CC: +1

As we can see, the code complexity is greatly reduced. We need far less lines of code to implement the same functionality. We also need to pass far less parameters around, and we have less statements that depend directly on their predecessor's output. The only metric that has gone up by one is the cyclomatic complexity. Which is the result of the fact that we build an array of tasks twice, in this example, using a for-loop. As we've seen in the previous proof-of-concept example, it uses a call to ITASK::WAITALL() without requiring a for-loop, as the tasks could be statically defined in an array in that case. In this specific case, they depend on the NUMCONTEXTS constant, which makes that more difficult.

Performance We can now compare the CPU time metrics between the threaded version and the task-based version. The threaded version is the same as the managed version from Chapter 5. The frame time results are shown in Table 6.1.

	Threaded	Task-based
	0.6091	0.6409
	0.6446	0.6413
	0.6381	0.6386
	0.6141	0.6416
	0.6504	0.6395
	0.6179	0.6410
	0.6690	0.6457
	0.6599	0.6302
	0.6617	0.6204
	0.6324	0.6315
Average:	0.63972	0.63707

Table 6.1: Frame times of threaded vs task-based implementation of D3D12 Multithreading sample.

As we can see, the performance between the two versions is virtually identical, which shows that the implementation of the Windows threadpool is very efficient. There is no significant performance difference between running threads and synchronizing manually with the use of events, or by using the threadpool to start tasks, either immediately or on an event. Likewise, having the threadpool signal when a task is done is also very efficient.

An advantage of using tasks is that they are more flexible than the three fixed threads in the original code. If we observe the implementations of DOSHADOWPASS() and DOSCENEPASS(), we see that they both use their own dedicated command lists and other resources. This means that they can run concurrently. This allows us to change the code as follows:

```
for (int i = 0; i < NumContexts; i++)
{
    m_shadowTasks[i].Reset();
    m_sceneTasks[i].Reset();
}

BeginFrame();

for (int i = 0; i < NumContexts; i++)
```

```
    m_shadowTasks[i].Run();

for (int i = 0; i < NumContexts; i++)
    m_sceneTasks[i].Run();

MidFrame();
EndFrame();
...
```

So instead of using CONTINUEWITH() to start each scene task after a shadow task, we start them all at once. This means that the threadpool will try to run them all in separate threads, if they are available. In the ideal situation, that means that the scene tasks will start immediately, and therefore will complete earlier than they would in the earlier versions of the code.

The results of a timed run of the new code are shown in Table 6.2.

	Run all tasks
	0.6219
	0.6211
	0.6412
	0.6191
	0.6259
	0.6255
	0.6160
	0.6743
	0.6313
	0.6200
Average:	0.62963

Table 6.2: Frame times for running all tasks of D3D12 Multithreading sample concurrently.

And indeed, this version turns out to be marginally faster on the test system, which has 4 cores and HyperThreading, which gives it a capability of running up to 8 threads in parallel.

If we want a simpler version of the code instead, and we don't mind trading in some performance, we can use locally definied tasks instead:

```
CPUTask shadowTasks[NumContexts];
CPUTask sceneTasks[NumContexts];

for (int i = 0; i < NumContexts; i++)
{
    shadowTasks[i].SetWorkLoad([=]
    {
        DoShadowPass(i);
    }
    );

    sceneTasks[i].SetWorkLoad([=]
    {
        DoScenePass(i);
    }
    );
}
```

```
BeginFrame();

for (int i = 0; i < NumContexts; i++)
    shadowTasks[i].Run();

for (int i = 0; i < NumContexts; i++)
    sceneTasks[i].Run();
...
```

This means that we can remove the LOADCONTEXTS() method altogether, as we can just set the lambda function every time. We also do not need a RESET()-call before every frame, because the tasks are new, and have not been run yet. And we do not need to define class member variables to store the tasks. All in all, this makes the code even more compact, and also more localized. Everything that is required for the definition, creation, initialization, starting of tasks and synchronization is now contained within the ONRENDER() method. There is a slight tradeoff, because creating and destroying new tasks for every iteration means that the OS also needs to create and destroy the underlying synchronization and threadpool objects, which is redundant, and may generate additional CPU-overhead.

We can assess the damage by doing a test-run of this version, shown in Table 6.3

	Dynamic tasks
	0.6506
	0.6485
	0.6432
	0.6465
	0.6541
	0.6494
	0.6493
	0.6428
	0.6496
	0.6431
Average:	0.64771

Table 6.3: Frame times for dynamically created and destroyed tasks.

As we can see, the impact on performance is marginal. Which means that in many cases, this can be a good tradeoff between performance and readability/maintainability of code.

7
Transition Management

7.1. Problem analysis

Transitions are managed by the Resource Barrier synchronization primitive. Resource barriers are placed inside command lists to signal transitions in resource use. The resource barrier needs to signal the before and after state of the transition. All possible states are described by the D3D12_RESOURCE_STATES enumeration:

```
typedef enum D3D12_RESOURCE_STATES {
    D3D12_RESOURCE_STATE_COMMON,
    D3D12_RESOURCE_STATE_VERTEX_AND_CONSTANT_BUFFER,
    D3D12_RESOURCE_STATE_INDEX_BUFFER,
    D3D12_RESOURCE_STATE_RENDER_TARGET,
    D3D12_RESOURCE_STATE_UNORDERED_ACCESS,
    D3D12_RESOURCE_STATE_DEPTH_WRITE,
    D3D12_RESOURCE_STATE_DEPTH_READ,
    D3D12_RESOURCE_STATE_NON_PIXEL_SHADER_RESOURCE,
    D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE,
    D3D12_RESOURCE_STATE_STREAM_OUT,
    D3D12_RESOURCE_STATE_INDIRECT_ARGUMENT,
    D3D12_RESOURCE_STATE_COPY_DEST,
    D3D12_RESOURCE_STATE_COPY_SOURCE,
    D3D12_RESOURCE_STATE_RESOLVE_DEST,
    D3D12_RESOURCE_STATE_RESOLVE_SOURCE,
    D3D12_RESOURCE_STATE_GENERIC_READ,
    D3D12_RESOURCE_STATE_RAYTRACING_ACCELERATION_STRUCTURE,
    D3D12_RESOURCE_STATE_PRESENT,
    D3D12_RESOURCE_STATE_SHADING_RATE_SOURCE,
    D3D12_RESOURCE_STATE_PREDICATION,
    D3D12_RESOURCE_STATE_VIDEO_DECODE_READ,
    D3D12_RESOURCE_STATE_VIDEO_DECODE_WRITE,
    D3D12_RESOURCE_STATE_VIDEO_PROCESS_READ,
    D3D12_RESOURCE_STATE_VIDEO_PROCESS_WRITE,
    D3D12_RESOURCE_STATE_VIDEO_ENCODE_READ,
    D3D12_RESOURCE_STATE_VIDEO_ENCODE_WRITE
};
```

For this thesis, only a subset of all possible combinations of these states will be considered. Not all combinations are valid, and some valid combinations are highly unusual, and beyond the scope of this thesis. For example, the last 6 states deal strictly with video encoding, decoding and processing. Another deals with raytracing. Both video and raytracing are topics are outside the scope of this thesis, which focuses on conventional triangle rasterization.

7.2. Transition Management

Direct3D 12 has a default resource state known as D3D12_RESOURCE_STATE_COMMON. Resources in this state can be transitioned implicitly to certain states, and will also implicitly transition back to the COMMON state after the operation is complete [29]. This means that for many of the trivial operations, it is not required to perform transitions manually by adding resource barriers. Microsoft also recommends that this mechanism be used where possible [19].

This also means that for the simple cases, the Direct3D 12 environment already offers help for the developer to keep things simple, and we do not need to do anything.

For the more advanced cases however, we need to derive the usage patterns from the code at runtime, by analyzing the API calls. The transition problem is related to the residency problem, and can build on the code that is already being used to track resource usage.

However, as already mentioned in Chapter 5, the currently implemented solution only tracks the binding of the resources to the pipeline. It does not track the actual usage. For transition management, the usage is even more important. Firstly, because superfluous transitions may incur a much higher performance penalty than superfluous residency of some resources. Secondly, because less-than-exact resource management at the binding level may lead to ambiguous states.

For example: a resource can be used as a read-only texture in a rendering call, and then as a read-and-write (unordered access) buffer in a compute call. For each specific usage, the developer must create a view. So in this case we have both a SHADER RESOURCE VIEW for the read-only texture usage, and an UNORDERED ACCESS VIEW for the compute access. As explained earlier in Chapter 5, it is the view that gets bound to the pipeline, not the resource itself. So a situation can occur where both these views of the same resource are bound to the pipeline at the same time.

In the case of residency management, it is not an issue, because the resource will just be considered a duplicate. It has to be resident, regardless of whether you want to read or write it. So it does not matter if the same resource is encountered multiple times. It will be made resident on the first occurence in an API call, and other occurrences will be ignored, because the internal linked lists will filter out duplicates.

With transition management this is not the case, however. The actual commands stored in the command list need to be analyzed, and a resource barrier will need to be inserted in front of a command that requires a transition.

Implementing a solution for this transition management problem is outside the scope of this research, and will remain for future research.

8
Related Work

Various other areas of research deal with related problems. We will look at some of them here, and how they relate to the discussed problems and chosen solutions.

Proxy objects The Proxy pattern was documented by the Gang of Four [13]. Using a Proxy object as a middle-man between the accessing object and the restricted object, allows the Proxy object to control access to the restricted object. The Proxy object can manage access to resources based on the application state. This pattern was applied to the residency problem by creating Proxy objects that closely matched the original DirectX 12 objects, but performed the additional state-tracking required as a middle-man between the application and the native DirectX 12 objects.

Protocol languages The problem of synchronizing multiple parallel/concurrent tasks can be seen as a form of communication. The required synchronization can be described in a communication protocol. Various languages exist, such as Scribble [39] and Reo [16], which allow designers to formally write down the communication protocol between components, by applying constraints to the input and output. These protocols can then be verified statically or at run-time (or a combination of both), to ensure that a given program complies to the protocol. For example, verification can check for deadlock freedom and for data race conditions.

8.1. Type systems

The area of type systems appears to be an interesting choice to attack the synchronization problems. Let us first look at the concepts of linearity, ownership and typestate in more detail. Then we can discuss how these concepts are related to the problems of synchronization, state tracking and resource access.

Linearity The concept of linearity was first brought forward in Girard's Linear Logic [14]. This was applied to programming in linear type systems. Objects of a linear type can be used exactly once. This means that aliasing of objects is not possible, which avoids data race conditions. Memory management can also be simplified because of this restriction. Linear type systems are supported in various programming languages, including LinearML, Linear Haskell and Rust.

Ownership Ownership is the concept of each object having a property of 'owner' as part of their type, which itself is another object [9]. Constraints can then be placed on access. For example, only the owner can have access, or other objects may have read-only access, where the owner is also allowed to modify. When only the owner can modify an object, data race conditions can be avoided. Rust is a programming language that supports ownership in its type system.

Typestate Typestate is a concept where 'state' is added to a type system [37], where each object type is modeled as a finite state machine. This allows each object to be in different states, where operations on these objects can be allowed only in certain states, and disallowed in others. For example, a file can only be read if it is in the open state. A program written in a language with a typestate system can be analyzed at compile-time, to verify that no semantically nonsensical method calls are made. Typestate can also help with memory management: it can aid in finalization and garbage collection of objects, based on their state.

The states can be used to define a certain (partial) order in which methods can be called on objects. Eg. method Y can only be called after method X, but never before Z. This can aid in synchronizing parallel/concurrent tasks, and avoiding data races.

The automatic finalization and garbage collection can be found in various languages today, including Java and languages based on the .NET framework (such as C# and VB.NET). A complete typestate system can be found in some experimental academic research languages, including Plaid [1] and Obsidian [10]. Early versions of Rust (up to 0.4) also had support for a complete typestate system. This was dropped in later versions, because of some complications that were too difficult to resolve. However, other features of the Rust language provide the necessary building blocks to implement the *branding pattern*, which achieves the same goal as typestate in practice [38].

Linearity, ownership and typestate are all related concepts. Typestate can be seen as the most generalized form of the three. Ownership and linearity can be implemented via a typestate system. And linearity can be seen as a very restricted form of ownership: a variable can only be assigned once, so only the 'owner' of that variable can use it.

All these concepts can be implemented in the type system of a language. And in some cases, a hybrid form can be implemented, where part is implemented in the type system itself, and can be verified at compile-time, and another part is implemented through a runtime, and is handled dynamically.

It seems plausible that the aforementioned problems of resource residency and transformation can be mapped to a model of polymorphic types. That is, one could see these transformations as objects being cast from one type(state) to another. Any cast operation will implicitly be a synchronization point.

The key to such an approach might be that the command lists need to follow very strict rules on their input and output states. When such strict rules can be enforced, command lists can be treated as some sort of ‘primitives’, where their inputs need to be cast to the correct form before executing the command list, and the command list itself will cast these inputs to another specific, known form after executing.

A related strict rule would then be that each command list may contain only one set of transformations, in other words: only one synchronization point.

The resulting abstraction should lead to building blocks which always have inputs and outputs with known types, and known semantics. These blocks can be connected to each other, and the connections can be validated statically by checking the types and casts, effectively constructing a directed, acyclic graph. This should lead to a system that avoids any data race conditions on resource access, both on accessing non-resident resources and on using resources that have not been transformed into the correct form for the required operation. The system should be able always add the correct synchronization operation automatically, while avoiding any deadlocks.

8.2. VERIFICATION VS SYNTHESIS

Verification is the process of analyzing an existing program for correctness. Many of the above methods are aimed at verification. That is, the concepts are implemented in the language in a way that they can be analysed by a tool, usually at compile-time.

Synthesis is the process of generating correct code, based on a given specification. In general, synthesis is more complicated than verification. However, when modeling object types as some form of finite state machine, the system is capable of tracking the state of objects. In specific cases, there may be enough constraints and rules in place that the correct code can be derived from a given state trivially, because there is only one logical next step from that state.

Automated finalization and garbage collection, as mentioned above with typestate systems, is a simple example of synthesis. Once an object has reached its ‘final’ state (there are no more references to it), the only operation that can still be done is to finalize and garbage collect the object. So if the system can determine when an object has reached this state, it can also perform the finalization and garbage collection on it. In Java and the .NET framework, this is performed dynamically by the runtime, but in a true typestate system, this could be derived at compile-time.

If a form of synthesis is possible, it is preferred over just verification, within the context of this research. To take the example of automatic garbage collection: it is better to have the system manage the memory for you, and avoid memory leaks altogether, by generating correct code automatically, than it is to just have compile-time warnings or errors. In the latter case the programmer would still have to actively think about the problem and write code to solve it. There is still a chance that bugs will occur in that code.

The implementation of using Proxy objects to track the usage of resources in linked lists, can be seen as a problem that is closely related to garbage collection. The linked lists track the usage of resources in a command list, and a RESIDENCYSET is synthesized at the point where the command list is sent to a command queue for execution. Given the similarities between these problems, it may be possible that solutions for garbage collection may also be adapted to resource tracking in DirectX 12 and similar APIs. It may be possible to apply more efficient methods of synthesis at runtime, or even to replace some parts of synthesis with compile-time verification.

9 DISCUSSION

9.1. DISCUSSION

During this research, I have developed a framework in C++ that can be used as a companion to DirectX 12. The modification of the D3D12 MULTITHREADING SAMPLE as covered in Chapter 5 can be used as a guide for other developers, and the results appear to indicate that it is reasonably simple to add it to an existing application.

The DirectX 12 Residency Starter Library may solve the issue of evicting and making resources resident, but still leaves the developer to solve the problem of relating the resources to the parameters actually passed to commands in command lists and other API operations. My framework abstracts away most of these details of how to use the Residency Starter Library.

An advantage of the framework is that it operates at the same level of abstraction as the DirectX 12 API itself. This means that an existing application does not require a lot of redesign, and no extra 'management' layer is required. This also means that the framework works within the same design parameters in terms of concurrency. It does not introduce any extra need for synchronization, or any possibilities of race conditions. The design of the application can remain unchanged, and concurrency will continue to work as before. This sets it apart from 'middleware' solutions which introduce an extra level of abstraction between the application and the API.

Another advantage is that the framework adds the option of automatically managing resources, but it does not enforce it. The non-managed objects and APIs are still available, and non-managed and managed resources can be used side-by-side in the code. This means that an application can be designed with multiple types of objects: permanently resident, managed in batches at a higher level, and mananged automatically at the command list level.

The task library developed to manage the concurrency problems is still in a very early stage of development, but appears to get the job done. While it supports both CPU and GPU tasks, it could be used as a CPU-only framework for any kind of C++ application, not related to DirectX 12, graphics or GPUs at all. It offers an easier-to-use, more modern interface to the basic threadpool that has been available in Windows since Vista, and which turns out to be very efficient in use. The task library could also be extended to support different types of tasks, for other types of processors in the system.

Within the scope of this thesis, there was only enough time to do an initial analysis on the problem of transition management. A practical solution could not be implemented and tested at this time.

9.2. STATIC VS DYNAMIC VALIDATION

The current implementation makes use of dynamic validation. A number of arguments can be presented against using dynamic validation. Dynamic validation will require a runtime framework that performs validation checks while the application is running. This sort of validation is exactly what has been removed from modern graphics APIs in the first place, in order to increase performance.

Aside from that, there already are debugging tools available for development, which already allow developers to detect problems resulting from resources being in the wrong state at the wrong time. Therefore a dynamic validation layer would only be of limited added use.

What a developer actually wants is the guarantee that a program always executes valid code only. This is also the situation as developers know it from previous graphics APIs, where the synchronization is implicit.

The course that the D3D12 Residency Starter Library has taken, is a combination of static and dynamic validation (or actually synthesis). This appears to be the best of both worlds. The Proxy objects make use of the built-in strong typing system in the C++ language, to give static validation where possible. My framework, combined with the Residency Starter Library, adds extra functionality at runtime, which would be difficult or impossible to perform statically at compile-time.

9.3. CONCLUSION

We will now look at the individual research questions and evaluate the results we have found in relation to them.

RQ1: How can residency management be automated? We have found that Microsoft already offers a building block for managing state, in the DirectX 12 Residency Starter Library. This however left the problem of relating resources to the actual views (descriptors) being used, to the developer. We have implemented a framework using Proxy objects which track these relations, and abstract the Residency Starter Library away.

RQ2: How can concurrency be simplified? The .NET Task Parallel Library was an inspiration here. Building on the native threadpool available in Windows, a task library was developed in C++, which mimics the style of programming and the functionality of .NET, including the use of lambda functions when starting CPU tasks. Adding to that, a GPU Task was introduced as well, something that .NET does not offer. By sharing the same ITASK interface that the CPU task uses, CPU and GPU tasks interact with eachother almost seamlessly.

RQ3: How can transition management be automated? The problem has been analyzed and broken down into the actual states that have to be managed, and an indication of where and how this can be done. An actual implementation and evalution of the proposed solution is left for future research.

RQ4: How can the solution be evaluated and benchmarked, to quantify how well the solution has solved the problem? The solution can be evaluated by comparing versions of code that do not use our new framework with versions that do. 'Ease of use' can be quantified to a certain extent by applying code metrics and comparing between the versions. Performance can be benchmarked by implementing high-performance timers in the code and measuring CPU time spent per frame, on average.

The results for the residency management indicate that the performance hit is marginal. Especially if we consider that the platform used for testing is far from cutting-edge. The Core i7 860 CPU is a mainstream CPU from 2009, and the GeForce GTX970 is a mainstream GPU from 2014. There are much faster CPUs and GPUs available today, which would take even less than the 0.7% of overhead at 60 fps. The framerate is more or less a fixed target, as the human eye cannot process frames at much higher rates than 60 Hz. Special gaming monitors do exist, which support framerates of up to 144 Hz. But even at those extremes, the overhead would still be marginal, as a single frame still takes 6.94 ms. So the measured 0.12 ms of overhead is still less than 2% of the total frame time at 144 Hz.

Adding residency management to an existing application (with no management at all) only increases the code complexity metrics marginally. Importantly the cyclomatic complexity is not affected. Code metrics do not tell the full story of complexity, however. All code is very straightforward, and generally only involves adding one or two parameters to a function. The concept of addressing resources via the slot index in their descriptor heap (like indexing an array) is arguably simpler than manually creating the D3DX12_CPU_DESCRIPTOR_HANDLE and D3DX12_GPU_DESCRIPTOR_HANDLE and passing them around as a sort of memory pointer. So it is a very simple trade-off to make: getting automated residency management without having to write significantly more code, significantly more complex code, or significantly different code.

The results for the concurrency management show that it is possible to get a coding style in C++ that is very similar to the .NET Task Parallel Library, which will be familiar for people who have programmed with tasks in C#. Passing code to a GPU task looks very similar to a call to EXECUTECOMMANDLISTS(), which is normally used to execute code on the GPU. Moreover, the performance benchmarks show that there is little or no tradeoff between the added simplicity of the code, and the actual performance. The performance of the task-based code is virtually identical to the original version with manually created and synchronized threads.

9.4. FUTURE IMPROVEMENTS

9.4.1. RESIDENCY

While the current solution for resource management is an acceptable compromise (instruct the developer to keep resource binding closely matched to shader usage), it is possible to also keep track of the access patterns of shaders. Microsoft offers a number of shader reflection interfaces [27], which will allow you to inspect the shader code, and deduce which resource slots are actually accessed, and how. Future research could refine the resource management by not only tracking which resources are bound to the pipeline, but also tracking which resource slots are actually accessed by the shaders being executed.

Future work could also look into optimizing the framework further. Currently a simple linked list is used to track resources. Other datastructures could be investigated, which may be more efficient for larger datasets. Since most of the time is spent in the Microsoft Residency Manager, rather than the Proxy objects, future research could look into ways of optimizing the Residency Manager itself.

9.4.2. CONCURRENCY

The .NET Task Parallel Library offers more functionality than what is currently implemented in the C++ task framework (such as making a task cancellable, and using WHENANY(), WHENALL() or YIELD()). Also, the .NET task can return a RESULT variable. The C++ framework could be extended to implement all the functionality, rather than the current subset.

Continuation tasks are currently always implemented asynchronously. The library could be extended to allow the option to run them synchronously if possible.

The Windows threadpool also allows the user to set the priority for a work item. This functionality could also be exposed through the task library.

Finally, as mentioned before, the ITASK interface could also be generalized across other asymmetric multiprocessing systems as well, by adding implementations of the interface for other specific processing units than just the CPU and DirectX 12 GPU implementations developed so far.

9.4.3. TRANSITION MANAGEMENT

We have seen that the transition management problem is a very complex one to tackle. It should be possible to implement the required additional state tracking on top of the current solution that tracks residency. As mentioned earlier, one improvement would be to analyze the actual shader code with reflection, to find only the resources that are actually in use. Shader reflection can also tell us how resources are used: read-only, write-only, or read and write. Another improvement will be to track the resource state of the resources during the population of a command list, and updating the state at relevant calls which might require a transition (most notably the ones that execute the shader code, such as ID3D12GRAPHICSCOMMANDLIST::DISPATCH() and ID3D12GRAPHICSCOMMANDLIST::DRAW(INDEXED)INSTANCED()).

A
Contact information

Student	Name:	D.G.A. de Bruijn, B.Sc.
	Address:	Van der Hoopstraat 55
		2523HE Den Haag
	Telephone:	06-51164282
	E-mail:	dga.debruijn@studie.ou.nl
Chairman	Name:	Dr. S. T. Q. Jongmans
	Address:	Valkenburgerweg 177
		6419AT Heerlen
	E-mail:	sung-shik.jongmans@ou.nl
Supervisor	Name:	Dr. C. P. T. de Gouw
	Address:	Valkenburgerweg 177
		6419AT Heerlen
	E-mail:	stijn.degouw@ou.nl

Bibliography

Academic Literature

[1] Jonathan Aldrich, Robert Bocchino, Ronald Garcia, Mark Hahnenberg, Manuel Mohr, Karl Naden, Darpan Saini, Sven Stork, Joshua Sunshine, Éric Tanter, et al. “Plaid: a permission-based programming language”. In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM. 2011, pp. 183–184.

[3] Gabriele Bavota, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Fabio Palomba. “An experimental investigation on the innate relationship between quality and refactoring”. In: *Journal of Systems and Software* 107 (2015), pp. 1–14. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2015.05.024. URL: https://www.sciencedirect.com/science/article/pii/S0164121215001053.

[4] Robert Blumofe and Charles Leiserson. “Scheduling multithreaded computations by work stealing”. English. In: *Journal of the ACM (JACM)* 46.5 (1999), pp. 720–748.

[5] Robert L Bocchino. “Alias control for deterministic parallelism”. In: *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Springer, 2013, pp. 156–195.

[7] Edwin Catmull. *A Subdivision Algorithm for Computer Display of Curved Surfaces*. PhD. Thesis. Utah University, Dec. 1974.

[8] O. Chaparro, G. Bavota, A. Marcus, and M. Penta. “On the Impact of Refactoring Operations on Code Quality Metrics”. In: *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 2014, pp. 456–460. DOI: 10.1109/ICSME.2014.73. URL: https://doi.ieeecomputersociety.org/10.1109/ICSME.2014.73.

[9] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. “Ownership types: A survey”. In: *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Springer, 2013, pp. 15–58.

[10] Michael Coblenz, Reed Oei, Tyler Etzel, Paulette Koronkevich, Miles Baker, Yannick Bloem, Brad A Myers, Joshua Sunshine, and Jonathan Aldrich. “Obsidian: Typestate and Assets for Safer Blockchain Programming”. In: *arXiv preprint arXiv:1909.03523* (2019).

[12] Michael J Flynn. “Some computer organizations and their effectiveness”. In: *IEEE transactions on computers* 100.9 (1972), pp. 948–960.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. “Design Patterns: Elements of Reusable Object-Oriented Software”. In: Addison-Wesley, 1994. Chap. 4, pp. 233–245. ISBN: 978-0201633610.

[14] Jean-Yves Girard. “Linear logic”. In: *Theoretical computer science* 50.1 (1987), pp. 1–101.

[16] Sung-Shik TQ Jongmans, Sean Halle, and Farhad Arbab. “Reo: a dataflow inspired language for multicore”. In: *2013 Data-Flow Execution Models for Extreme Scale Computing*. IEEE. 2013, pp. 42–50.

[32] Ally S. Nyamawe, Hui Liu, Zhendong Niu, Wentao Wang, and Nan Niu. “Recommending Refactoring Solutions Based on Traceability and Code Metrics”. In: *IEEE Access* 6 (2018), pp. 49460–49475. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2868990.

[33] Mikael Olofsson. *Direct3d 11 vs 12: A performance comparison using basic geometry*. 2016.

[34] David Lorge Parnas. “Software Aging”. In: *Proceedings of the 16th International Conference on Software Engineering*. ICSE ’94. Sorrento, Italy: IEEE Computer Society Press, May 1994, pp. 279–287. ISBN: 0-8186-5855-X. URL: http://dl.acm.org/citation.cfm?id=257734.257788.

[37] Robert E Strom and Shaula Yemini. “Typestate: A programming language concept for enhancing software reliability”. In: *IEEE Transactions on Software Engineering* SE-12.1 (1986), pp. 157–171.

[39] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. “The Scribble protocol language”. In: *International Symposium on Trustworthy Global Computing*. Springer. 2013, pp. 22–41.

ONLINE RESOURCES

[2] AMD. *Asynchronous Shaders - unlocking the full potential of the GPU*. 2015 (accessed January 14, 2022). URL: http://developer.amd.com/wordpress/media/2012/10/Asynchronous-Shaders-White-Paper-FINAL.pdf.

[6] Steve Burke. *Initial Vulkan Benchmark vs. DirectX 11 - AMD & NVidia in Talos Principle*. 2016 (accessed October 3, 2021). URL: https://www.gamersnexus.net/guides/2319-initial-vulkan-vs-dx11-benchmark-amd-v-nvidia.

[11] CrimsonRayne. *How Much Does DX12 REALLY Improve Performance? | DirectX 12 vs DirectX 11 & OpenGL Vs Vulkan*. 2018 (accessed October 3, 2021). URL: http://www.redgamingtech.com/how-much-better-is-performance-with-modern-apis-directx-12-vs-directx-11-opengl-vs-vulkan/.

[15] Joel Hruska. *New DirectX 11 vs. DirectX 12 comparison shows uneven results, limited improvements*. 2017 (accessed October 3, 2021). URL: https://www.extremetech.com/gaming/246377-new-directx-11-vs-directx-12-comparison-shows-uneven-results-limited-improvements.

[17] Brent Justice. *DX12 versus DX11 Gaming Performance Video Card Review*. 2017 (accessed October 3, 2021). URL: https://www.hardocp.com/article/2017/03/22/dx12_versus_dx11_gaming_performance_video_card_review/11.

[18] Khronos. *Khronos Reveals Vulkan API for High-efficiency Graphics and Compute on GPUs*. 2015 (accessed October 3, 2021). URL: https://www.khronos.org/news/press/khronos-reveals-vulkan-api-for-high-efficiency-graphics-and-compute-on-gpus.

[19] Bill Kristiansen. *A Look Inside D3D12 Resource State Barriers*. 2019 (accessed December 20, 2021). URL: https://devblogs.microsoft.com/directx/a-look-inside-d3d12-resource-state-barriers.

[20] Microsoft. *Direct3D 12 Graphics*. 2018 (accessed July 21, 2019). URL: https://docs.microsoft.com/en-us/windows/win32/direct3d12/direct3d-12-graphics.

[21] Microsoft. *Direct3D 12 multithreading sample*. 2016 (accessed November 21, 2021). URL: https://github.com/microsoft/DirectX-Graphics-Samples/tree/master/Samples/Desktop/D3D12Multithreading.

[22] Microsoft. *DirectX-Graphics-Samples*. 2016 (accessed November 21, 2021). URL: https://github.com/microsoft/DirectX-Graphics-Samples.

[23] Microsoft. *Interface Hierarchy*. 2021 (accessed January 15, 2022). URL: https://docs.microsoft.com/en-us/windows/win32/direct3d12/interface-hierarchy.

[24] Microsoft. *Pipelines and Shaders with Direct3D 12*. 2018 (accessed July 21, 2019). URL: https://docs.microsoft.com/en-us/windows/desktop/direct3d12/pipelines-and-shaders-with-directx-12#direct3d-12-graphics-pipeline.

[25] Microsoft. *Programming Guide for Direct3D 11 | Graphics Pipeline*. 2018 (accessed July 21, 2019). URL: https://docs.microsoft.com/en-us/windows/desktop/direct3d11/overviews-direct3d-11-graphics-pipeline.

[26] Microsoft. *Residency*. 2018 (accessed December 18, 2019). URL: https://docs.microsoft.com/en-us/windows/win32/direct3d12/residency.

[27] Microsoft. *Shader Interfaces (Direct3D 12 Graphics)*. 2021 (accessed December 20, 2021). URL: https://docs.microsoft.com/en-us/windows/win32/direct3d12/d3d12-graphics-reference-shader-interfaces.

[28] Microsoft. *The D3D12 Residency Starter Library*. 2018 (accessed August 12, 2019). URL: https://github.com/microsoft/DirectX-Graphics-Samples/tree/master/Libraries/D3DX12Residency.

[29] Microsoft. *Using Resource Barriers to Synchronize Resource States in Direct3D 12*. 2021 (accessed December 20, 2021). URL: https://docs.microsoft.com/en-us/windows/win32/direct3d12/using-resource-barriers-to-synchronize-resource-states-in-direct3d-12#implicit-state-transitions.

[30] NVIDIA. *Graphics Processing Unit (GPU)*. 1999 (accessed July 21, 2019). URL: http://www.nvidia.com/object/gpu.html.

[31] NVIDIA. *Transitioning from OpenGL to Vulkan*. 2016 (accessed October 3, 2021). URL: https://developer.nvidia.com/transitioning-opengl-vulkan.

[35] Pixar. *The Shading Process: An Overview.* 2015 (accessed July 21, 2019). URL: https://renderman.pixar.com/resources/RenderMan_20/shadingProcessOverview.html.

[36] Ryan Smith. *Microsoft Details Direct3D 11.3 & 12 New Rendering Features.* 2014 (accessed October 3, 2021). URL: https://www.anandtech.com/show/8544/microsoft-details-direct3d-113-12-new-features.

[38] Patrick Walton. *Typestate Is Dead, Long Live Typestate!* 2012 (accessed September 16, 2019). URL: https://pcwalton.github.io/2012/12/26/typestate-is-dead.html.