Emil Fransson 911023-2015
Emil Högstedt 991026-6650
2022-05-28
DV2551; 3D Programming 3

DV2551: DirectX 12 Project

Emil Fransson
Emil Högstedt

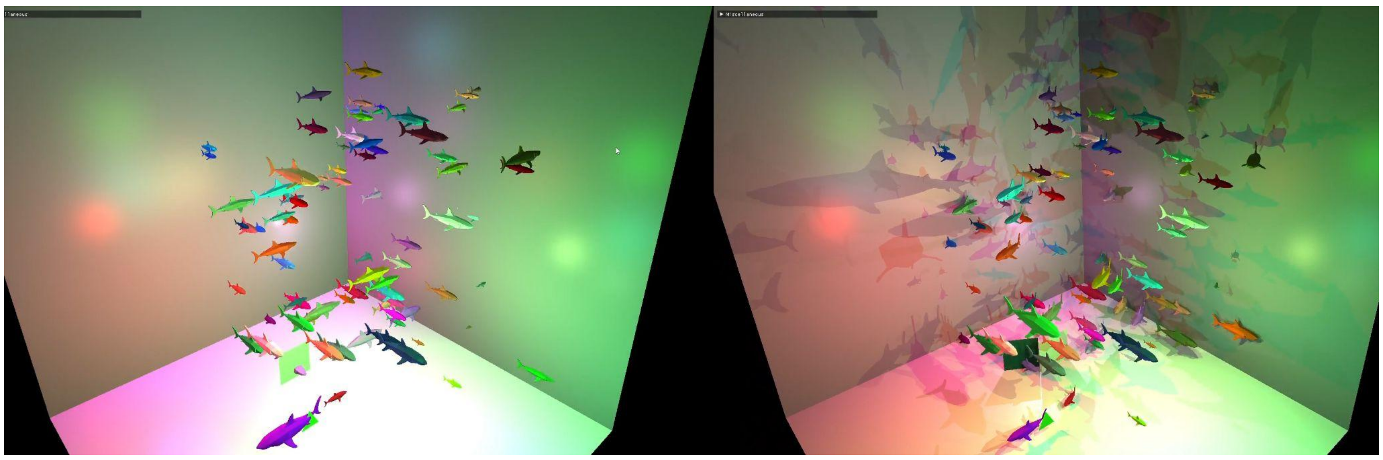

Figure 1 - Comparison of the visual result of our implementation of Inline Ray Tracing (right) and not using it (left).

1 Introduction

In this project Vertex Pulling, Multiple Frames Active Concurrently (Frames in Flight) and Inline Ray Tracing using DXR 1.1 have been implemented solely by the writers of this report. The visual part of the application presents the results of the implementation of Inline Ray Tracing and a comparison can be seen in Figure 1. A profiler was implemented to properly be able to test the Frames in Flight technique because of the lack of visual feedback, although the profiler was also used for measurements with regards to the effect of Inline Ray Tracing on the performance of the application. What these specific techniques entail and what exact measurements were taken will be further discussed in their respective parts of the report.

2 Techniques

2.1 Vertex Pulling

2.1.1 Theoretical background

When feeding vertex data to the Vertex Shader for processing in the GPU pipeline the developer has the option to employ a technique called *Vertex Pulling*. Traditionally a part of the pipeline called the *Input Assembler* has been used along with an *Input Layout*. An Input Layout is, as the name suggests, a structure that tells the Input Assembler how to interpret (assemble) the vertex data being passed to the GPU. Information required is, e.g., the ordering of the vertex attributes, the attribute formats, semantics and the offset into the bytes of memory for a given attribute. Forming such an input layout could for example tell the GPU to interpret the second vertex attribute as consisting of three float-numbers, with the semantic “COLOR” and an offset into the vertex data memory of 16 bytes. When compiling the program debug-functionality may assert that the proposed input layout’s semantics match with the semantics in the Vertex Shader.

Using the input layout may well help the program to interpret the data passed from the CPU to the GPU, however it comes with some added detriments. First, it requires the developer to put time and effort into forming such an input layout for every vertex data structure, or optionally build a system for dynamic input layout construction. Also, when utilizing Direct3D 12 the developer gets explicit control over the root signature, and how the 64 different DWORD:s worth of slots are used. Using an Input Layout along with the Input Assembler means one DWORD must be sacrificed, limiting the total DWORD:s to 63. Avoiding the use of an Input Layout means saving both precious development time and Root Signature slots.

This is where Vertex Pulling comes into play. Vertex Pulling entails binding the vertex data to the Vertex Shader directly and there assemble a vertex manually. The vertex data is bound to the shader directly by setting the corresponding *Vertex Buffer* and, optionally, *Index Buffer* for use on a command list. More precisely, the Root Signature must be primed with the root parameters for the used buffers, of type *Shader Resource View* and with the Vertex Shader stage as minimum visibility flag. Then the Vertex- (and index-)buffer virtual GPU addresses are used for setting a graphics root shader resource view for the exact vertex data to be used in the pipeline. The Vertex Shader has access to the vertex data via structured buffers. Accessing the data means indexing into the vertex structured buffer with a vertex ID that is an optional input for the vertex shader the developer may use. If an index buffer is used the vertex ID is instead used to index into the index structured buffer. This returns the final index that is *then* finally used to index into the vertex structured buffer. The developer may then access the vertex attributes as normal, which basically comes down to using structure member data by using the dot-operator on the formed vertex object.

2.1.2 Implementation

Vertex Pulling is used throughout the entire program in this implementation - no Input Layout possibility is given. The vertex data is all stored together and the creation of the vertex data object simply comes down to the previous theoretical explanation, here shown in code:
Vertex input = vertices[indices[vertexID]], where vertices is the structured Vertex Buffer, indices is the structured Index Buffer and vertexID is the vertex ID supplied for the developer. The rest of the program supports Vertex Pulling by the aforementioned implementation details. Also, of course, since indices is used this means using the *DrawInstanced*-function on the command list, giving as argument the index count of the Index Buffer as the vertex count per instance, basically telling the Vertex Shader how many times to be invoked for the bound Vertex Buffer.

2.1.3 Profiling

Due to the nature of this technique being an option of preference it is reasoned that this technique is not eligible for profiling. It is therefore skipped.

2.1.4 Discussion

Having Vertex Pulling as a technique can in some sense be a strict improvement since it saves a slot on the Root Signature. This is a pro for the technique. However, it is up to the developer to correctly form the Vertex object in the Vertex Shader, and interpret the data. If the application uses an Input Layout the Debug Layer of Direct3D12 can help the developer to discover signature mismatches and provide other useful insights that might be of benefit to the development process. Bugs might be noticed and fixed more easily. In this sense, it is a con that even more responsibility is put on the developer. However, this is of course a loop-sided statement, as having to form a correct input layout means that extra care must be taken to do so correctly as well.

All in all it is believed to be optimal for this project to use Vertex Pulling, no improvements can be seen, just other paths to take, of which none were of interest.

2.2 Multiple Frames Active/Alive Concurrently

2.2.1 Theoretical background

Performing rendering work on a computer usually involves both the CPU and GPU in tandem. The CPU more often than not deals with updating the scene logic, performs preemptive culling and submits render commands for use by the GPU, among many other things. Having two separate entities working together means work synchronization becomes an important topic. Failing to synchronize work between the CPU and GPU leads to undefined behavior and general application malfunction. In general, this is because the application must ensure that memory in use by the GPU is not manipulated by the CPU at the same time. Resources need to be in the right state at the right time, dependencies must be respected, and so on [1]. In Direct3D 11 the developer could for the most part treat CPU-GPU-synchronization as a black box, simply issuing commands and trusting the drivers to arrange and rearrange the work in such a way as to not violate different dependencies. For Direct3D 12, life is not so simple anymore. Memory must be allocated and handled more or less explicitly, commands such as resource transition barriers must be issued manually and synchronization must now be performed directly. Less work is done by the drivers, and more responsibility is placed on the developer. Synchronization is now done using so-called *Fences* with associated fence values, a construct that places certain “points of synchronization” in the command queue, which must be respected.

A naive approach would of course be to let the CPU perform all work for frame X and simply stall after having issued the GPU to perform the work for frame X. When the GPU is done with frame X's work, the CPU can work on frame X + 1, this time having the GPU wait for the CPU. This workflow would continue as such, ensuring CPU-GPU-synchronization. While this might be fine for certain applications, the drawbacks are quite glaring - namely that of performance. Having the CPU just hanging around waiting for the GPU (and vice versa) means precious work time is lost, spent on doing nothing. A solution to this problem is having more "*frames in flight*". This means that, while the GPU is working on frame X, instead of having the CPU wait, it is simultaneously updating scene logic and submitting commands for frame X + 1, or even later frames. Now the CPU is, in theory, used more efficiently. Adding more frames means the application must keep track of the current active frame, ensuring that it "swings back round" once it has reached its maximum value. For instance, having three frames in flight (0 - 2) and having the CPU perform frame 4, means going from two to zero, if that makes sense. Having multiple frames also means that all resources that are manipulated in some way by both the CPU and GPU must have multiple copies, one for each frame in flight. So if the CPU is preparing frame X + 1 while the GPU is working on frame X, there are no synchronization worries since the CPU is working on another copy of a resource entirely than the GPU is currently working on. This also means more memory must be managed correctly by the application. Just adding more frames in flight is generally not a valid option. While the CPU in theory can work on more frames before stalling, it also adds latency to the application since the CPU is ahead of the GPU. Also, depending on how the application is performance-bound (CPU or GPU) it might not add much to the performance.

It is still sufficient to use one Fence for synchronization, one Command List and one Command Queue, however multiple command allocators must now be used (one per frame in flight). This is because, while Command Lists are safe to reset before the commands have been executed fully on the GPU, Command Allocators that actually store the commands are not. Also, one fence value per frame in flight must now be used in order to keep track of what the fence values to signal and wait for are over the frames.

2.2.2 Implementation

For the implementation details the reader is suggested to refer to figure 2 continuously for clearer understanding. Note that figure 2 primarily focuses on how the memory is laid out to accommodate frames in flight ; it does not display all the relationships in regards to the back buffer. Also note that the picture displays how the memory will be laid out for three frames in flight, but the reader should be able to visualize the layout for other frame counts as well. Lastly, the implementation is experimental; this is by no means an optimal solution. It is simply for learning and experimenting.

As it stands there are two constants the user/developer may change to alter the number of back buffers and frames in flight that will be used by the program. This determines the values at compile time; there is unfortunately no possibility to change the values at run time. They are both found in DXHelper.h.

For dynamic frame count functionality, a basic memory manager has been implemented. The developer may query the memory manager for a shader visible heap of certain type(s) and size. The memory manager will set up the heap, however the size will, behind the scenes, be a multiple of the number of frames in flight in use. As such, and for three frames in flight, a query for a shader visible heap of size 250'000 and of type SRV/CBV/UAV would allocate such a heap, but of size 250'000 * 3 = 750'000. It would start off as empty. The developer may then query the memory manager to create a range of a certain size on this heap. Again, this range would then internally be a multiple of the number of frames in flight. Looking at figure 2, the shader visible heap has a range called "Transform", with a size equal to N * 3, with 3 being the frame count. N here is user defined, as aforementioned. Furthermore, the memory manager may be queried for a non-visible heap, that is connected to a certain range on a certain visible heap. Internally the memory manager will set up X distinct such non-visible heaps. This can also be seen in figure 2. Finally an entity can be given a view by the memory manager, such as a constant buffer view. This is a type defined for this project. This view is simply a struct with all the CPU/GPU-virtual addresses that make the connections in figure 2 possible.

Using figure 2 as an example, asking the memory manager for a transform constant buffer view, it would give the entity three addresses for the three different non-visible heaps, all with a correct offset depending on the current number of views in use. It would also give three different addresses, but *in the same shader visible heap*. Tying it all together, a single entity with a constant buffer view would be represented in three different "places" in three different non-visible heaps and in three different "places" in *one* shader visible heap. See more in the discussion section on why this particular strategy was chosen.

Continuing using the transform as basis for explanation, if the program would like to update the world matrix transform for an object, it would first query the application for the current active frame. It would then use this frame as an index into the correct heap addresses, which of course in turn are handles into the right place in the right heaps. This means that while the GPU is busy using Transform 1.1 in the shader visible heap (see figure 2!) the program may freely map and copy resource data between Transform 1 in shader non visible heap 2, to Transform 1.2 in the shader visible heap. There will be no conflicts and no simultaneous read/write errors. The changes happen at different places in memory. In regards to CPU/GPU-synchronization between frames, and the number of needed dx12-resource interfaces needed to account for the frame count, see the previous section "Theoretical background" - this project's implementation follows that structure.

Having analyzed the program using Microsoft PIX it was determined that the application did in fact not utilize Multiple Frames Active (MFA) correctly. The original implementation wrongly had the CPU wait for the GPU every frame which explains the surprising profiling results displayed in the original report and presentation. This bug has since been fixed and the application now correctly utilizes the MFA-feature. This is proven both in Figure 3 which displays how PIX registers how the GPU is performing various commands while the CPU is recording new commands for following frame(s), as well as the updated profiling results following the fix of the bug.

2.2.3 Profiling

The profiling is done the same way as in the original implementation, including the number of models being rendered, and the various MFA tests done. This means the only part of the application loop being profiled is updating the scene models, rendering the scene (including the swapchain present method), and waiting on CPU/GPU-synchronization. The application is given some time to start up corresponding to 2000 frames. Following the start up frames the profiler records the duration of one iteration and saves the time in a vector. This is performed 5000 times, after which an average is calculated and presented. As such the profiling results diagrams display the average time duration over 5000 frames (or iterations!). In addition to these profiling tests one additional test has been added. In the feedback it was suggested that a good way to measure the difference of MFA profiling results was to add up the frame time over several frames and

compare. As such the new test shows the result of the summed-up render loop frame time over 500 frames, rendering 100 objects corresponding to 130000 vertices and 193200 indices.
The results are displayed in Figure 4a - 4f. The results were measured in milliseconds in Release configuration and with optimizations turned on, on a i7 7700k @ 4,5 GHz and a GTX 1080 ti. All external factors were limited. Lastly, the tests were limited to profiling up to five frames in flight, with an equal number of back buffers in use.

2.2.4 Discussion

Starting off with the implementation, as was said in the beginning it is entirely experimental. No guides were followed, no best practices were looked up. The reason only a single shader visible heap was used for a certain type comes down to the fact that only one such shader visible heap can be bound to the pipeline at any point in time. Resetting multiple such heaps was deemed to be a more costly operation than simply setting one, and having that single heap be segmented into different ranges. As such, it is believed to be an optimization. Using multiple non-visible heaps comes down to the fact that they are not bound and rebound every frame or draw-call, and as such multiple can be used. This means they do not have to be split up into multiple segments, and can be traversed in a linear fashion without offsetting much. However, it is difficult to gauge on-fly what is best here for cache coherency: using multiple different heaps or a single one that is being heavily offset into. An improvement to the memory manager could be had but tests would have to be made to know what optimization to strive for. **In regards to the profiling results, for rendering a low number of models (low vertex/index count), in the range 1300 – 13000 vertices, the sweet spot seems to be three MFA. Of course, when very little GPU work is submitted per frame it can most probably be done with the work in such a fast way that the importance of having MFA lessens. This could explain the inconsistent results for rendering one and five models. Putting more load on the GPU and for rendering 130000 – 13000000 vertices the results are more in line with what one might suspect; the frame time is heavily improved from going from one to two MFA and only marginally better (if not slightly worse!) from increasing MFA to three and beyond. Looking at the results of the final test done (summed-up render loop frame time) again it is evident that going from one to two MFA improves the frame time quite a lot, while increasing MFA to three and beyond does not really add any improvements.**

2.3 Inlined Ray Tracing using DXR 1.1

The DirectX Ray Tracing (DXR) 1.1 technique has in the project been used to achieve ray traced shadows, fully working with multiple lights and multiple different models in a dynamic scene.

2.3.1 Preparation Work

First off, when implementing Inlined Ray Tracing a shader version of 6.5 has to be used to get access to the new HLSL types used for the technique, which will be discussed later during this section. This is where the first problem is encountered if the FXC compiler was originally used for the compilation of shaders during runtime as it is not compatible with shader models of version 6.0 or higher. The DXC compiler was therefore added to the project as a dynamic library and used to compile the shaders during runtime. This implementation is simply a single function that takes the name of the *.hlsl*-file, the name of the "main" function and the shader model as parameters, very similar to how the *D3DCompileFromFile*-function is called when using the FXC compiler, and then returns an *IDxcBlob* for that shader which is then used like normal during the creation of the *D3D12_GRAPHICS_PIPELINE_DESC*. The implementation of this function is fairly standard and straightforward, and is very well described by Simon Coenen in his blog post "*Using the DirectXShaderCompiler C++ API"*[2].

Secondly, vertices to perform the ray tracing against were needed, so that some kind of result could be visualized in real time. The Assimp library was therefore added to the project as a dynamic library and is used to load *.obj* files which correspond to different models and contain different meshes which in turn contain vertex positions, normals and indices. This data is saved in a vertex- and index buffer within the *Mesh* class of the application, with the meshes belonging to the same model then being stored in a vector in the *Model* class. Once a model has been loaded it is stored within the *Scene*-class of the application to then be used by objects within the scene. It is made sure that each unique model is only loaded once, so if a model that has already been loaded is requested for a new object the already existing *Model*-instance is used. This structure can be seen visually as a part in Figure 5.

Because of the fact that Inlined Ray Tracing (1.1) was implemented, and not 1.0 DXR, the shader pipeline still looks the same as in a normal rasterization engine, meaning we still rasterize our meshes using vertex- and index buffers, and therefore use a Vertex Shader and a Pixel Shader.

2.3.2 The Ray Tracing Acceleration Structures

To even be able to have any geometry to ray trace against during the shader pipeline the ray tracing acceleration structures first have to be set up. These acceleration structures contain the geometry data of everything that the shader should be able to trace against, in this case, in the Pixel Shader. These acceleration structures can also contain data with regards to arbitrary geometry, meaning more than vertex based geometry can be used, although the implementation of this report only uses vertex based geometry.

The *RayTracingManager* of the application is responsible for holding all the acceleration structures. There exist two different acceleration structures; there are Bottom level acceleration structures and Top level acceleration structures. Both types of acceleration structure use two buffers, which are *ID3D12Resource*'s with memory allocated using the *CreateCommittedResource*-function in the implementation, which are a *Result* buffer where the acceleration structure ends up in GPU memory after being built and a *Scratch* buffer where the build will store temporary data.

The bottom level acceleration structure holds primitives, while the top level acceleration structure holds instances of the different bottom level acceleration structures. The number of bottom acceleration structures therefore depends on two factors in the application; the number of unique models loaded and the number of meshes these models have. The application loops through all loaded models and for each mesh in each model a *Result* buffer and a *Scratch* buffer are created, these buffers and the mesh's vertex- and index buffer are then put in a *D3D12_BUILD_RAYTRACING_ACCELERATION_STRUCTURE_DESC*-struct which in turn is used for building the acceleration structure for that mesh. The buffers and the acceleration descriptor are then stored in the *RayTracingManager* by adding them to vectors which are inserted into unordered maps where the model's name is the key. An important restriction to mention that exists in the implementation is that the bottom level acceleration structures are only built at startup, this results in all unique models having to be loaded before the acceleration structures are built. Rebuilding the bottom level acceleration structures is needed when the primitives belonging to a structure change, for example when a model is animated, but since the meshes in this project are all static it is never needed.

In this report's implementation only one top level acceleration structure exists. In practice this results in all instances of all different bottom level acceleration structures existing within one top level acceleration structure. The top level acceleration structure is built by first assembling an array of *D3D12_RAYTRACING_INSTANCE_DESC*'s, with the size of the array being the total number of meshes in the scene. Each mesh of each object of each unique model is then looped through and their *D3D12_RAYTRACING_INSTANCE_DESC* get its transform set to the object's transform and the GPU address to the bottom level acceleration structure corresponding to the current mesh is connected to the descriptor. In our implementation we simply set the *InstanceID* of the descriptor to a unique number, but if a developer would want to use materials for their objects they could set this member variable to a material ID which then can be accessed when the object is traced against in the Pixel Shader.

After all the descriptors have been properly defined an *Instance* buffer, *ID3D12Resource*, is mapped and the descriptors are copied into the buffer's GPU memory. This buffer together with a *Result*- and *Scratch* buffer are then used when building the top level acceleration structure. An overview of how everything is connected in the implementation can be seen in Figure 5.

The top level acceleration structure that exists in the *RayTracingManager* is then updated and rebuilt every frame in case any object in the scene has its transform changed, this means that the code also very easily could be modified to also allow for objects to dynamically be added or removed during runtime. The top level acceleration structure is then sent to the shader by calling the *CommandList*'s function *SetGraphicsRootShaderResourceView*, meaning the acceleration structure is bound to the root signature as a shader resource view which is only visible to the Pixel Shader as defined when setting up the root signature.

During the implementation of the acceleration structures for this project the open source document "*DirectX Raytracing (DXR) Functional Spec*" which is posted on Microsoft's own github was used as reference to gain understanding and to achieve proper functionality[3].

2.3.3 The Shaders

Only the Pixel Shader has been altered in a significant way to allow for ray traced shadows. The only minor change in the Vertex Shader was to send the world position of the vertex as input to the Pixel Shader, meaning it gets interpolated. This was to allow for this point to be used as the origin of the ray to be used when tracing. The Pixel Shader gets invoked as many times as in a normal rasterization pipeline, once for every texel in question. A *RayQuery* and a *RayDesc* is then created for every light in each invocation using the world position of the texel and the direction to the light and is then traced against the acceleration structure. The resulting color is then decided using normal phong shading, with attenuation, but if the ray hit something and the distance to that hit is lower than the distance to the light then only the ambient lighting is included in the calculation, meaning the texel is in shadow of that light. After this is calculated for all lights the results are added together to create the final color for the texel.

2.3.4 Profiling

After the technique had been implemented a few measurements were taken using the profiler implemented for the project. The results can be seen in Figure 6.

For profiling a way of turning off the ray tracing and the updating of the acceleration structure was implemented and bound to a keyboard key. This was to allow for easier profiling during runtime and to more easily be able to see the difference that the implemented technique makes visually.

The measurements that were made investigated the significance that two variables had on the average frame time of the application; index count and light count. The same measurements were taken for both when the Ray Tracing was turned on and off.

The lines in Figure 6 that has the label “Few indices” used roughly 4’000 indices, the ones with the label “Many indices” used roughly 15’000’000 indices and the data points labeled “Extreme” used roughly 30’000’000 indices and were only tested with one light.

As Figure 6 shows; the performance difference does not seem to increase nor decrease as the number of indices increases, meaning that the amount of performance loss when using ray tracing gets lower with regards to percentage as the index count increases. As can be seen in the figure the performance linearly declines as the number of lights increases when using ray tracing, while the performance barely gets affected if ray tracing is not used.

2.3.5 Discussion

In our eyes the result of our implementation of the technique is satisfactory and highly increases the visual complexity and realism of the scene.

A negative of our implementation is that because of the reduction of performance when the number of lights increases our implementation linearly becomes worse and worse, although a very big positive of using this technique is the inadvertent high definition shadows that do not depend on the resolution of the light’s depth buffer if for example shadow mapping is used. Another positive of using Inlined Ray Tracing instead of for example the 1.0 DXR is the familiarity to a normal rasterization pipeline, it is therefore not as big of a step to learn if you already are familiar with either DirectX 11 or 12. Another negative might be that, unless you are rendering a quad or making sure that no void is visible in your scene, arbitrary geometry included in the acceleration structure will not be seen unless it overlaps with rasterized geometry and a ray is traced from the camera for every texel which it currently is not in our implementation.

An alternative to our approach of only having one top level acceleration structure would be to instead build one top level acceleration structure per unique model, or even per unique mesh, with all instances of that model or mesh existing in one acceleration structure. There are positives and negatives to both approaches, with the clear positive of the multiple top level acceleration structure approach being that within the pixel shader it would be easier to determine what model or mesh is currently being traced against. The negative of this approach, also being the reason that using this approach was decided against in this project, is that when a lot of different objects are moving in the scene a lot of top level acceleration structures would have to be updated and rebuilt. Although it also works both ways, so if you would have a scene where only ever one object would ever change it would be better to use multiple top level acceleration structures, as only one structure containing one instance would ever have to be rebuilt instead of rebuilding one acceleration structure containing a lot of objects that do not need to be updated and rebuilt.

The improvements that we see could be made to the application with regards to this technique is the addition of materials and textures for more complex objects within the scene. This together with dynamic lights that move and change and the ability to remove and add both lights and objects during runtime would increase the visual complexity even more. Optimizing the ray tracing by culling unnecessary lights by for example first checking the dot product of the normal in the world position of the texel and the direction to the light from the point and not performing the trace towards that light at all if the normal points away from the light would increase performance. Since the same vertex- and index buffers that are used for rasterization are also used for the bottom level acceleration structures they include not only the vertex positions but also the normals, this reduces cache friendliness but was chosen as our approach because of the ease of using the same buffers and also for the possibility of maybe being able to access this information when a triangle is hit during the ray tracing in the pixel shader to allow for easy reflections. No way to be able to extract this information from the acceleration structure within the shader was found though.

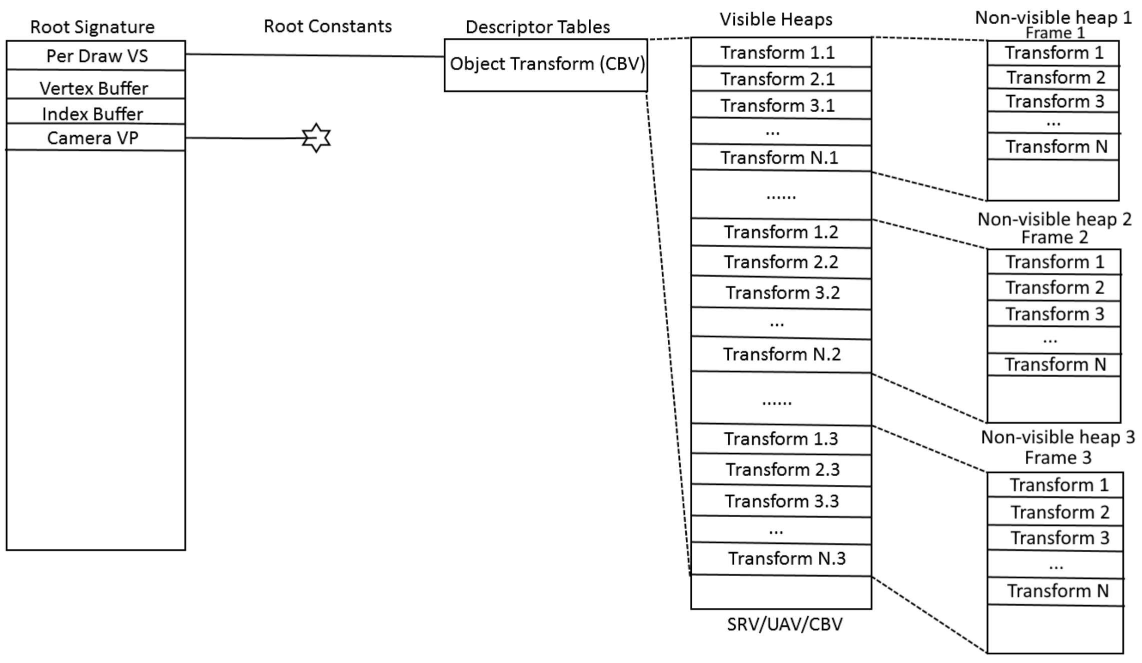

Figure 2 - Heap memory layout example of three frames in flight, as managed by the memory manager. Figure is not exhaustive but instead highlights frames in flight concept.

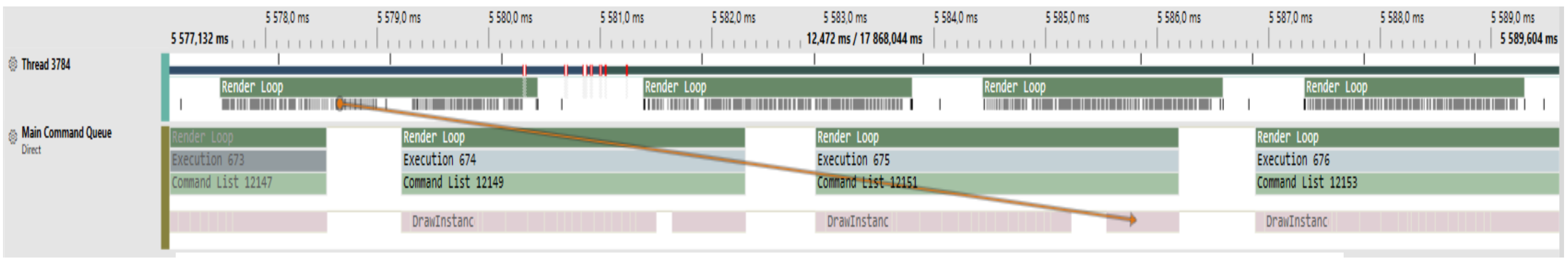

Figure 3 - Microsoft PIX visualization of three MFA. The arrow denotes how a command being executed by the GPU was recorded by the CPU on an earlier frame.

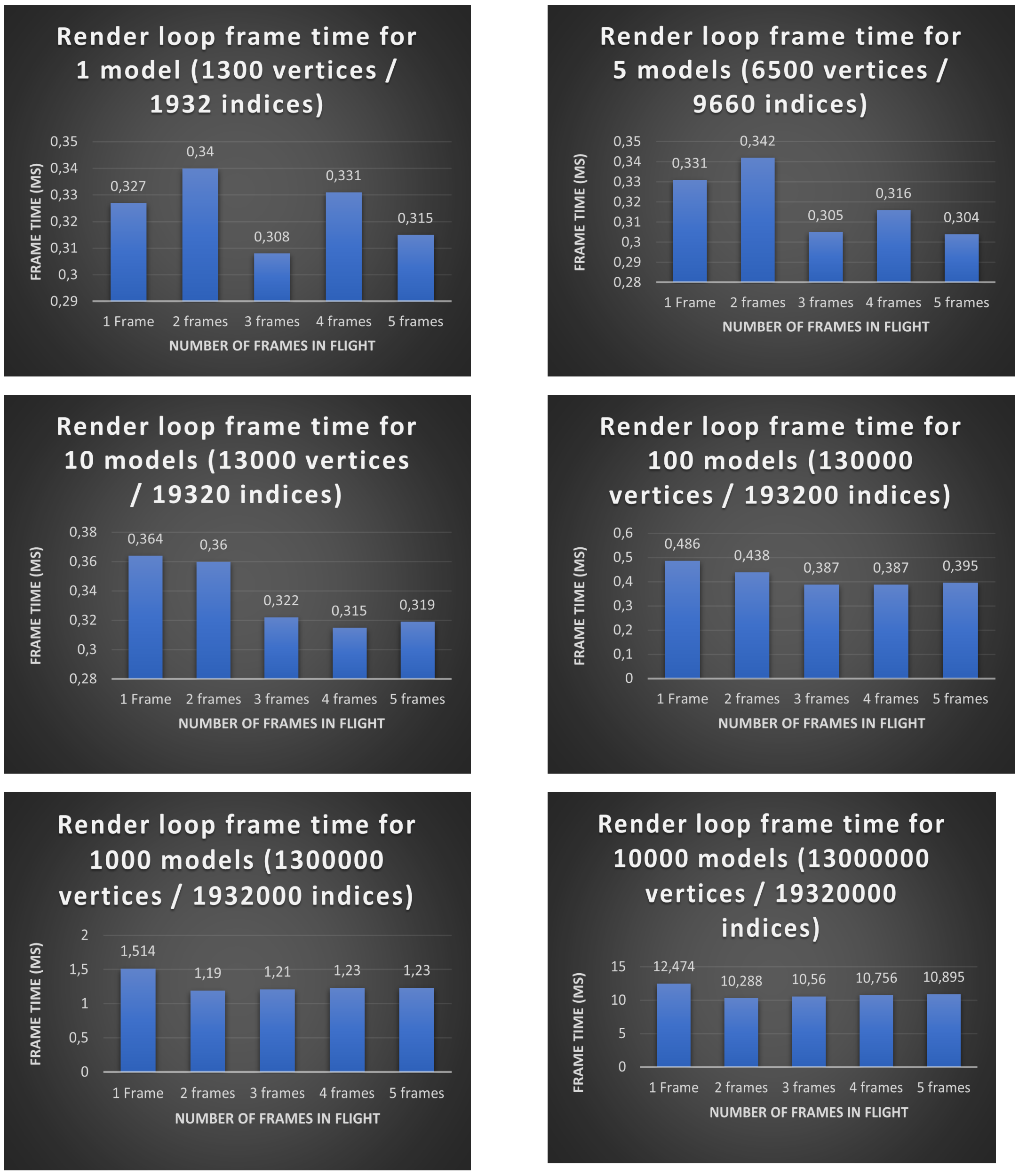

Figure 4af - Render loop profiling results in milliseconds using 1, 5, 10, 100, 1000 and 10000 models consisting of 1300 vertices/1932 indices each. Lower results are better.

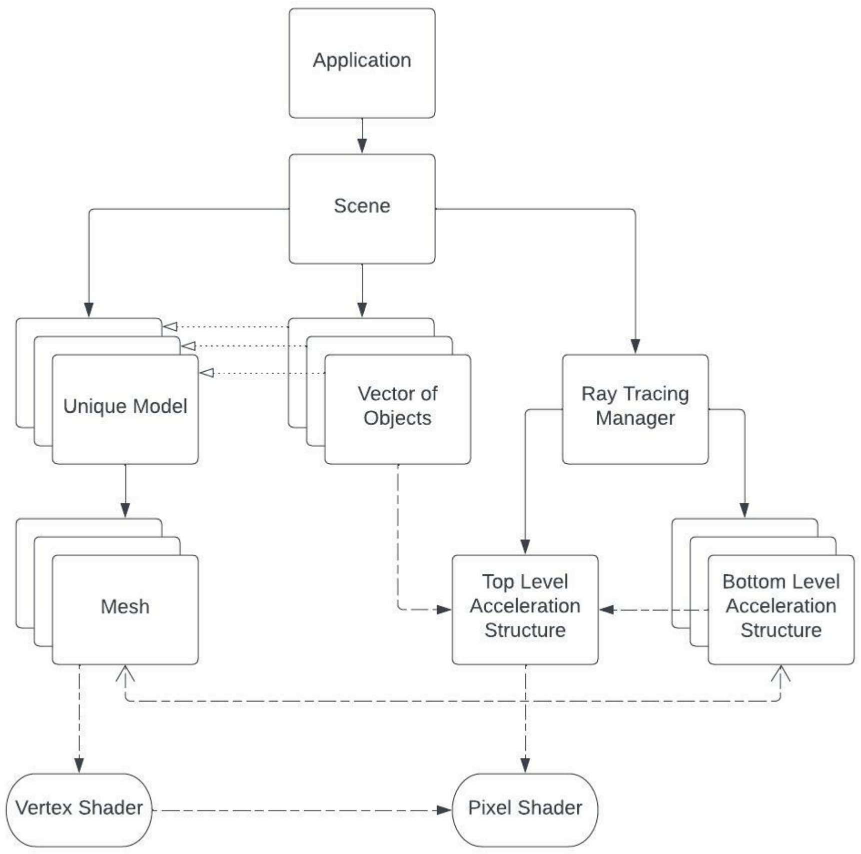

Figure 5 - Overview of the part of the application responsible for the meshes and Ray Tracing acceleration structures. Black arrow equates to ownership. Dotted white arrows indicate that one vector exists per model. Dashed black arrow means the source gets sent to the destination where it is used. Dashed arrow means one bottom level acceleration structure exists per mesh that exists.

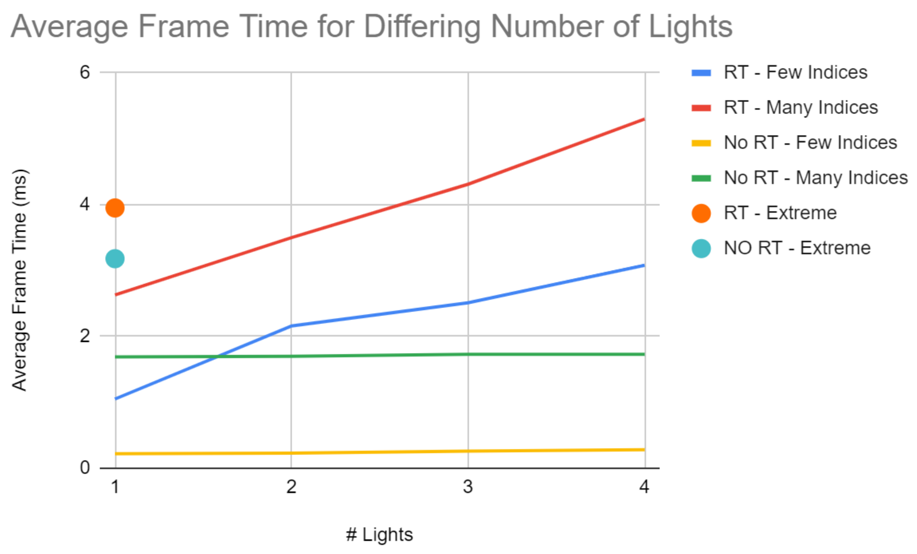

Figure 6 - Results of the measurements made of the Inlined Ray Tracing Technique. Few Indices = ~4'000. Many Indices = ~15'000'000. Extreme = ~30'000'000.

References

[1] Overvoorde, A., 2022. *Frames in flight - Vulkan Tutorial*. [online] Vulkan-tutorial.com. Available at: <https://vulkan-tutorial.com/Drawing_a_triangle/Drawing/Frames_in_flight> [Accessed 28 May 2022].
[2] Coenen, S., 2020. *Using the DirectXShaderCompiler C++ API*. [online] Simon Coenen Game Programmer. Available at: <https://simoncoenen.com/blog/programming/graphics/DxcCompiling> [Accessed 28 May 2022].
[3] Github. 2022. *DirectX Raytracing (DXR) Functional Spec*. [online] Available at: <https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html> [Accessed 28 May 2022].