Cuypers Yentoff

DLL injected 3D model and texture exporter

Graduation work 2017-18

Digital Arts and Entertainment

Howest.be

CONTENTS

Abstract 2
Introduction 3
Research 4
1. Injection 4
1.1. Intro 4
1.2. Exploiting DLL behavior 4
1.3. API hooking 5
2. Graphics pipeline 8
2.1. Intro 8
2.2. Getting texture data 9
2.3. Getting model data 11
2.4. Replacing textures 13
Development: Hooking Framework 14
1. introduction 14
2. DLL Injection 15
3. API Hooking 18
Development: DirectX 11 26
1. Introduction 26
2. Exporting Textures 29
3. Exporting Models 32
4. Replacing Textures 43
Conclusion 45
References 46
Appendices 49

ABSTRACT

Dynamic Link Libraries (DLL) are a very convenient way to update and extend applications or share API's which are prone to updates. They can easily be updated by simply overriding them without having to recompile the whole application. Unfortunately (or maybe fortunately in some cases) the dynamic loading of DLLs and the automation of this process comes with several ways to exploit them. One such exploit is DLL Injection which tricks an application in loading a customized DLL and executing its code. Using this method we can redirect the application's execution flow and use its data however we want.

INTRODUCTION

Code injections are a commonly used method for adding 3rd party tools to games, ranging from FPS meters to video recording software to overlays for chat programs to cheats and hacks. One such method is DLL injection, which in the case of games usually exploits the graphics API libraries such as DirectX and OpenGL. The basic concept of the injection is a 'man-in-the-middle' attack. The DLL pretends it is the official code by passing all the data to the real one and returning its results, but in the meantime it collects whatever data it wants.

In this paper we'll explore the above method and use it to extract the raw texture and mesh data from a DirectX application. The paper will go into as much detail as possible about the injection process, hooking of the functions and the setup to extract the data we want. It will also take a deeper look into the DirectX 11 pipeline and data structures and explore different ideas and methods to extract the previously mentioned data.

The final goal of this paper is to gain a deeper knowledge of DLL injections and windows API hooking, to learn how graphics pipelines can be exploited and/or extended and gain an improved knowledge of the inner workings of the DirectX 11 pipeline and data structures.

This paper aims to achieve these goals by first discussing the topics on a more theoretical level laid out in the research, followed by more in depth explanations about different methods and thought-processes in the development chapters.

RESEARCH

1. INJECTION

1.1. Intro

In software development injection is a common exploit to add custom code to an existing application. This can be used to change the flow of the application or to adjust or collect data.

There are a lot of different ways to accomplish this. Such as using existing functions from the operating system to load code into another application, Breakpoint-trapping, DLL redirection, DLL replacement and DLL inline redirection.

These days there are plenty of tools out there called Injectors which take care of loading the DLL or code into a running application by abusing the operating system's API to hook functions and redirect them. In this paper we will use a combination of the DLL replacement and DLL inline redirection to inject our code into the target application.

This paper will briefly discuss what DLLs are, how they are loaded and how we can exploit that.

1.2. Exploiting DLL behavior

A DLL is a binary Windows file which can contain executable code and resources much like an EXE. Rather than being executed itself it can be dynamically loaded at runtime by an executable file (application). The application will either load the DLL when it is required or load all of them at once during startup.

A common use case of DLLs is to share an API or allow for easy updates of the application. The application simply calls the functions in the DLL and expects them to do what the API says they do. The application does not care about the implementation of the API, only that the functions are there and it can call them. Together with COM objects, which we'll discuss later, this makes it very easy to extend or update existing applications without the need for users to reinstall a completely recompiled version of the application.

On the flip side, this also comes with several vulnerabilities. Since the target application only knows how to call the DLL and not what it actually does internally, all we have to do is somehow trick it to use our custom DLL which uses the same interface as the original but does something entirely different.

Surprisingly enough, tricking the target application isn't too hard. Windows comes with several ways to add our custom DLL or replace the original one with ours. One such method would be using the 'AppInit_DLLs infrastructure'. This is a Windows Registry setting which will automatically load the listed DLLs into any process connected to the User32.dll (which is virtually every process). Adding our own DLL to this list will automatically load it into any process ready to be hooked. Unfortunately this doesn't always work and isn't a very user-friendly method to get others to use your tool.

No worries, I said there are multiple methods and one of them is even easier to abuse. This method is known as 'DLL Replacement'. As the name indicates all you do is replace the original DLL with yours. Typically an application doesn't use hardcoded paths to the DLLs as this might break it when it's installed on another PC or moved to another location. Therefore they will first try and look for the DLL in the root or 'bin' directory of the application. If it cannot

be found there, the application will typically look in the Windows System folders (e.g. system32) where API DLLs are typically installed. So what we can do is find the DLL and replace it with our customized version.

The only problem with DLL replacement is that, unless we have the original source code, we no longer have access to the original functionality. This is a problem because it would break the application. As mentioned in the introduction we're aiming to do a 'man-in-the-middle' attack. Considering we don't have the source code we still somehow have to return the expected data to the application.

A third method is using 'DLL Inline Redirection'. With inline redirection we make use of a technique called Detours and Trampolines to redirect the flow of the application. This method allows us to perform a man-in-the-middle attack by redirecting to our custom code and at the same time allowing us to call the original code. More on this method will be explained later.

If we bring both DLL Replacement and DLL Inline Redirection together we get a successful injection framework. We use abuse the vulnerability that a DLL is looked for in the root directories first to place our DLL with the same name as the original so ours is loaded first. Then we use Inline Redirection to hook the functions we want and redirect the flow and data as we see fit.

One important thing to note for the replacement DLL is that it has to export the same functions as the original one. In other words, it has to use the same interface as the API for the application to be able to call it. More on this in the next part about API hooking.

1.3. API hooking

COM and IUnknown interfaces

The Component Object Model is a standard for creating and connecting components across different environments. A COM component is a binary interface from which a COM object can be created. These binary COM objects can easily be used by many different applications with each their own goal without needing the source code or the actual implementation.

The COM specification addresses a method to create binary interface objects which can be used across languages, networks, machine architectures and more. To accomplish this it specifies the usage of a fixed binary structure and enforces the implementation of the IUnknown interface.

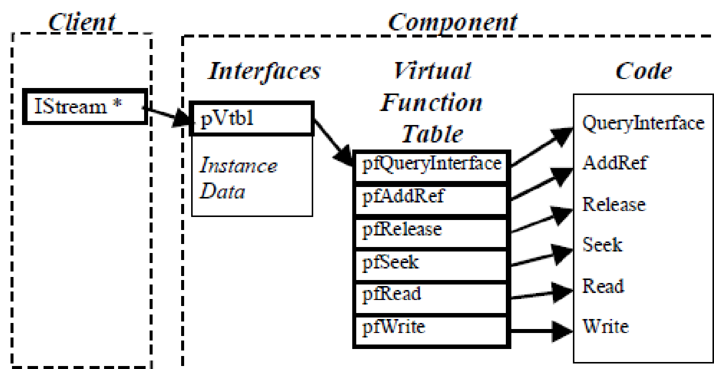

FIGURE 1. BINARY INTERFACE MAPPING. (INTERCEPTING AND INSTRUMENTING COM APPLICATIONS, GALEN C. HUNT, MICHAEL L. SCOTT)

Each COM object can implement multiple interfaces. The instance and data stay the same or each object but depending on which interface is requested you get a pointer to a different virtual table with different functions.

The format defined for a COM object is similar to the common representation of a C++ class with a pure abstract virtual function table. The first 4 bytes of each COM object are a pointer to the virtual table which holds all the function pointers. The bytes that follow are made up of the member variables the object holds.

Every COM interface is required to implement the IUnknown interface. This interface has two uses: object lifetime management through reference counting, and access to various predefined interfaces. To be able to identify which interface to use at runtime the IUnknown interface requires the usage of a GUUID, which is a unique identifier for each interface.

The IUnknown consists of the following three functions:

`IUnknown::QueryInterface`

Retrieves pointers to the supported interfaces on an object. More info on the requirements for this implementation on the MSDN page.

`IUnknown::AddRef`

Increments the reference count for an interface on an object. This method should be called for every new copy of a pointer to an interface on an object.

`IUnknown::Release`

Decrements the reference count for an interface on an object. When the reference count on an object reaches zero, this function must cause the interface pointer to free itself. When the released pointer is the only existing reference to an object the implementation must free the object.

Each COM object can reference multiple interfaces, for each interface there is a separate virtual table.

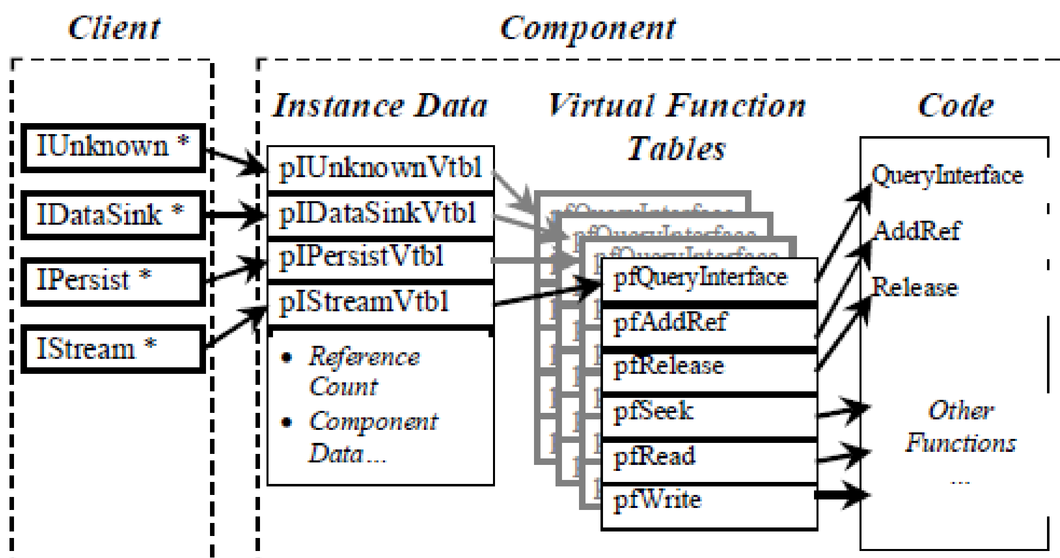

FIGURE 2. SIMPLE OBJECT LAYOUT. (INTERCEPTING AND INSTRUMENTING COM APPLICATIONS, GALEN C. HUNT, MICHAEL L. SCOTT)

Virtual tables

A virtual table is basically a list of function pointers. When dealing with inheritance we can have multiple virtual tables per object, with each pointing to their derived implementation of the functions. Typically when an object is a created, a pointer is added to this list at the start of the object. For COM objects the first 4 bytes is the pointer to the Virtual Table. The virtual Table for COM objects always has the three required functions from the IUnknown interface at the top.

Since the same virtual table is reused for every instance of a class, the compiler passes the pointer to the object the function has to be used on as well. This pointer is typically known as 'this'. In many object oriented languages it is a hidden variable that is implied when using member variables and functions from within a class. In reality it is passed at the front of every function that requires it.

Detours and trampolines

Detours and trampolines are a method of hooking function calls. The basic idea is to replace the first few instructions of a function call with a jump to a custom detour function. The replaced instructions are then used to create a trampoline function which at the end has a jump call back to the original function plus the offset of our detour jump call.

We intercept and redirect the exported DLL functions when they are first called or when the DLL is first loaded by the main application. This results in the application always calling our detour (hooked) function and whenever we want to use the original we simply call the trampoline.

DETOUR

A jump instruction at the start of the original (target) function to our custom (hooked) function.

TRAMPOLINE

Function with the replaced parts of the original function and a jump back to the original function, plus the offset of our detour jump. In x86 (32-bit) this is 5 bytes.

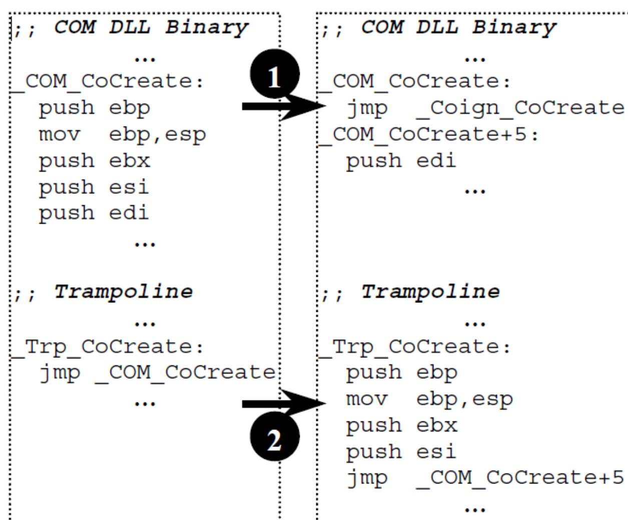

FIGURE 3. INLINE REDIRECTION. (INTERCEPTING AND INSTRUMENTING COM APPLICATIONS, GALEN C. HUNT, MICHAEL L. SCOTT)

2. GRAPHICS PIPELINE

2.1. Intro

A graphics pipeline is a concept in computer graphics that describes what steps have to be taken in order to render a 2D or 3D scene. Unfortunately the steps that are to be taken depend on the software and hardware used as well as the desired result. Although there is no universal pipeline which works for all current and future cases, there are several graphics API's, such as Direct3D and OpenGL. These API's try to unify as many similar steps as possible and take care of controlling the hardware accelerator.

As explained in '1.3. API hooking' we can use a public API to our advantage to hook into the application and interact with the data as we see fit.

To limit the scope of this paper we'll only use the DirectX 11 API, but the basic principles discussed here can mostly be translated into other API's as well.

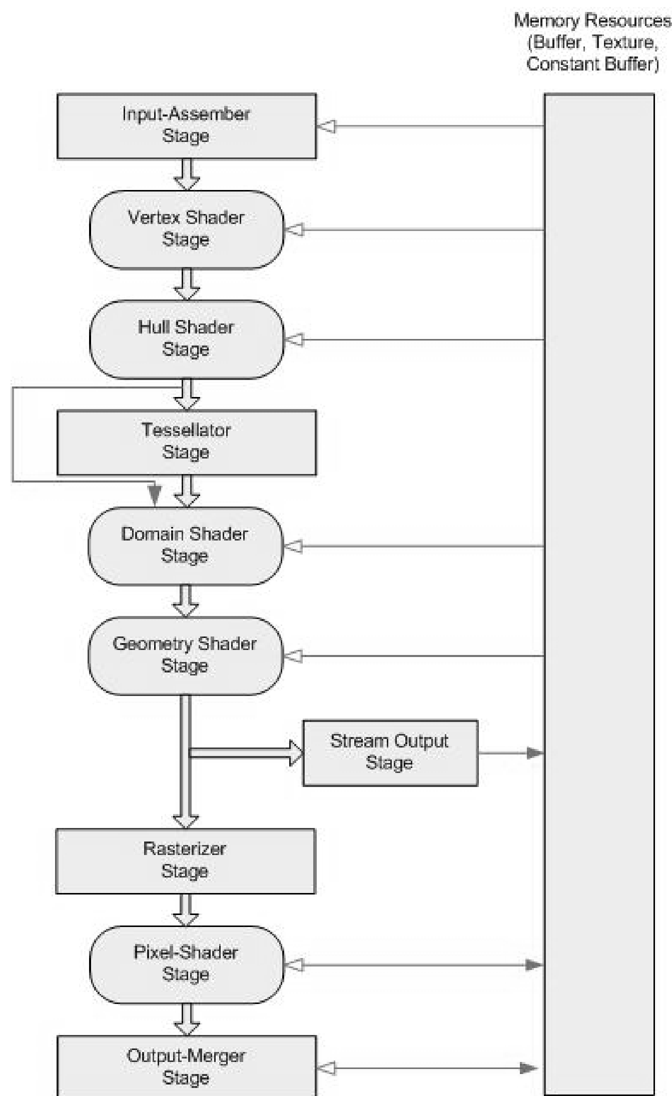

FIGURE 4. DATA FLOW FROM INPUT TO OUTPUT THROUGH EACH OF THE PROGRAMMABLE STAGES (DIRECTX 11 GRAPHICS PIPELINE, MSDN)

The typical flow of a DirectX 11 application can be described in 3 steps:

- Initialization
- Update loop
- Cleanup

During the initialization, the application creates a window, initializes the DirectX 11 API and starts the update loop. During the DirectX initialization, it creates Direct3D COM objects such as 'ID3D11Device', 'ID3D11DeviceContext' and 'IDXGISwapChain'. These objects are used to configure the programmable pipeline stages.

During the update loop, the application updates based on input data and configures the different stages each with their own requirements. At the end of the update loop the 'IDXGISwapChain::Present' function is called to present the rendered image to the user.

The cleanup stage is triggered by an exit event during the update loop and takes care of cleaning up all used resources, freeing memory and correctly closing the application.

2.2. Getting texture data

In order to export textures from an application we first need to intercept the texture data that's being loaded into the application. The easiest way to do this is to hook the function where a texture is created. Another possibility would be to hook the function where the textures are for the pipeline stages before the draw calls.

DirectX 11 has three functions for creating textures:

- `CreateTexture1D`
- `CreateTexture2D`
- `CreateTexture3D`

As their names explain, they are respectively for creating 1D, 2D and 3D textures. For this project we're only interested in 2D textures so 'CreateTexture2D' is the function we're interested in hooking and intercepting data from.

When textures are created the developer has to specify the 'usage' type. This type is used to determine whether the GPU and/or the CPU can read and/or write to the texture resource.

`D3D11_USAGE_DEFAULT`

Default usage is used for a resource that requires read and write access by the GPU.

`D3D11_USAGE_IMMUTABLE`

Immutable usage is meant for resources that can only be read by the GPU.

`D3D11_USAGE_DYNAMIC`

Dynamic usage can be read by the GPU and written to by the CPU.

`D3D11_USAGE_STAGING`

Staging usage is used for a resource that can be copied from the GPU to the CPU. This is the type we need to get CPU read access and export the data.

Another setting used to determine CPU access is the 'CPU Access Flags'. This flag is default set to '0' and will be ignored, resulting in GPU only access.

There are two other settings available:

`D3D11_CPU_ACCESS_WRITE`

CPU can write to this resource. This resource cannot be used as output for the pipeline and must be created with dynamic or staging usage.

`D3D11_CPU_ACCESS_READ`

CPU can read from this resource. This resource cannot be used as input or output for the pipeline and must be created with staging usage.

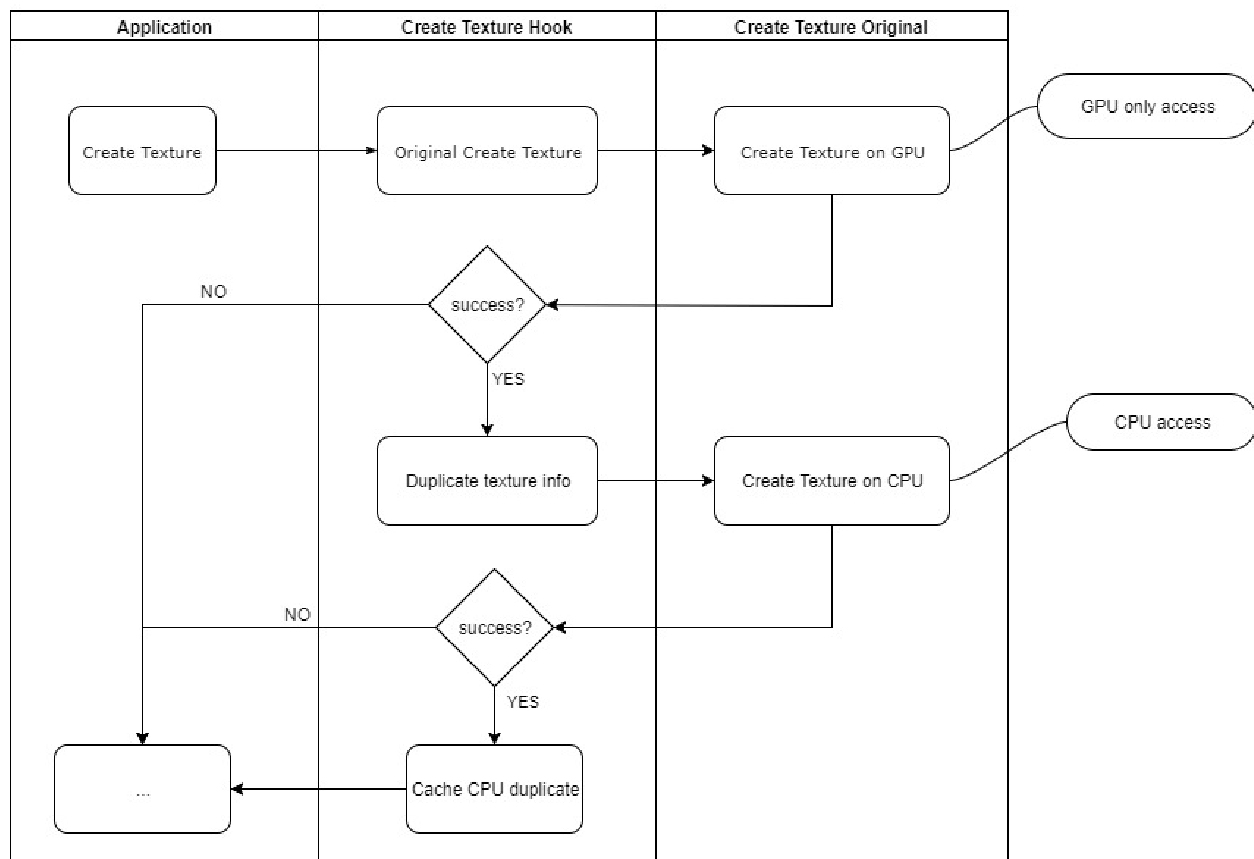

FIGURE 5. ABSTRACT FLOWCHART FOR CACHING TEXTURE DATA

Knowing all this you might think we're done, just hook the 'CreateTexture2D' function, pass the staging usage type instead of the application's and dump the raw data to a file. This isn't entirely incorrect but there are some things to take into consideration.

For example, the main application is expecting the texture to be created with the data it passes and thus changing any of these will cause issues. A second issue is performance. If we start exporting data every time a texture is created we'll lock down the application a lot. And final a third big issue is duplicates. If we just start exporting whatever is being created, there's a good chance we are exporting textures that have been used or created before.

To solve the first issue we simply take an extra step of creating a duplicate resource with the same size and parameters, except the staging and CPU access flags are set to allow reading from the CPU. Afterwards we can ask the DirectX API to copy the resource data into our duplicate resource which can only be read by the CPU.

Solving the second issue is pretty straight forward, instead of exporting the data right away, we cache them in a list and export them whenever the users asks for it with some form of input.

This second solution already solves part of our third issue. To solve the duplicates all that's left is to calculate a hash value for the raw data when it's first created and compare it to already cached values. If two values match we know it's the same and we can just skip it.

2.3. Getting model data

Exporting models from an application uses some similar principles as the texture exporting. First we have to intercept the relevant data such as vertex buffers and index buffers. Unfortunately these two aren't enough. The index and vertex buffers are loaded into memory as a set of bytes with a structure defined by the main application. Thus to be able to export the data to a usable file format we need to reverse engineer the data structures used.

For reconstructing the original structure and data of one model we need the vertex buffer, index buffer, primitive topology, input layout and the buffer description. The vertex buffer and index buffer hold the byte data, whilst the primitive topology, input layout and buffer description explain how it is structured.

A second problem is that the buffers and input layouts can be created in any order and can be used for multiple models. The only time all these are connected to each other is during the draw calls. At this point in the pipeline the buffers, topology and input layout are bound to the 'input-assembler stage' to be used as input for the draw calls and shaders.

Some functions we can use to get all the relevant data are:

`ID3D11Device::CreateBuffer`

Creates a buffer (vertex buffer, index buffer, or shader-constant buffer). Similar to the CreateTexture2D function we can use this to cache our buffers.

`ID3D11Device::CreateInputLayout`

Creates an input-layout object used to describe the input-buffer (vertex buffer) for the input-assembler stage.

`ID3D11DeviceContext::IAGetIndexBuffer`

Gets the pointer to the index buffer currently bound to the input-assembler stage. As well as the format and starting offset. The format determines the size and data type of each index in the buffer. The only formats allowed are, 16 bit unsigned integer 'DXGI_FORMAT_R16_UINT' and 32 bit unsigned integer 'DXGI_FORMAT_R32_UINT'.

`ID3D11DeviceContext::IAGetVertexBuffers`

Gets the pointer to the vertex buffer(s) currently bound to the input-assembler stage. As well as the size of one vertex (stride) and the starting offset.

`ID3D11DeviceContext::IAGetPrimitiveTopology`

Gets the primitive topology type (enum) that is currently bound to the input-assembler stage. This describes the primitive type and data order of the input data. Can be used to reconstruct indices per face.

`ID3D11DeviceContext::IAGetInputLayout`

Gets the pointer to the input layout currently bound to the input-assembler stage.

Now our first step to exporting models is the same as with exporting textures. We cache the buffers created during the 'CreateBuffer' function calls based on the passed parameters we can differentiate between the different types of buffers. We're only interested in caching the index and vertex buffers. The same issues pop up as with the textures, such as duplicates and CPU read/write privileges. But since the index and vertex buffers can be used with different input layouts or buffer combinations we cannot tell whether two vertex buffers with exactly the same data will be used in the same way. Therefore we allow duplicate data, but we'll hash the pointer to the buffers and compare them during the draw calls to determine which one is used. This may result in duplicate models in the end but we also won't miss any.

A second part is to also cache all the input layouts and the array of input layout descriptions during the 'CreateInputLayout' calls. The input layout itself is just a pointer which we'll compare during the draw calls to map the correct description to the vertex buffer. The descriptions contain the semantic name with the format and offsets. The format determines the size in bytes of that part of the vertex structure.

Next during the draw calls we try to map all the cached data to the data currently bound to the input-assembler stage and cache these mappings so we can later use the correct combination of vertex buffer, index buffer, input layout and topology to reverse engineer the data and save it to a useable file format.

There is just one big issue with mapping and exporting meshes during the draw calls, and that is performance. If we're constantly comparing lists and caching new data, the draw calls are almost permanently stalling. To prevent this we can use a 'snapshot' principle where we only do our mappings for a set amount of time after the user asks for it. Another improvement to performance is to use multi-threading to do the actual exporting because writing a lot of data to files can be really slow. When using multi-threading we do have to be careful with the typical problems that arise such as shared lists and mutually exclusive data access etc.

2.4. Replacing textures

To replace textures at runtime we can use the same principles as exporting models. There the basic idea is to cache the relevant data and match it during the draw calls before exporting it.

For replacing textures we also start out by caching the data we need but instead of exporting anything we set our replacement texture during the initialization of the draw call and resources so ours will be used.

To know where or how to do the switch we have to know how textures are bound to shaders and used during the draw calls.

Every texture that has to be used by a shader needs to be created with a 'ShaderResourceView' (SRV). This SRV can then be set to the shader stages.

`ID3D11Device::CreateShaderResourceView`

Creates a SRV for accessing data in a resource. This function can be hooked to cache the SRVs used by the application.

`ID3D11DeviceContext::PSSetShaderResources`

Binds an array of shader resources to the pixel shader stage. This function is called during the draw call initialization to set textures and other shader resources to the pixel shader stage.

As mentioned above we first need to cache the SRVs because they are automatically linked with the textures used and we can do this by hooking the 'CreateShaderResourceView' function. Since we won't be exporting any data we don't need CPU access and therefore don't need to do any conversions as with exporting textures and models. Here we can simply cache the pointers to the SRV and texture so we can later compare them with a 'selected' index to choose which one to replace.

Next, the 'PSSetShaderResources' function takes an array of SRVs to be set for the current draw call and can therefore be used to compare the SRVs every frame with our selected one to be switched out. Here we can simply pass the original data if we don't want any change or pass our own data or the textures we do want to replace.

Again with this solution there are some concerns with performance as we are constantly comparing and switching data, but so far this seemed to worked best and does not have problems with cached data on the GPU.

Another possibility I looked into was switching out the addresses (pointers) to the textures and shader resources used. Even though the addresses where successfully switched out the actual textures used were not updated which is most likely due to some form of caching used behind the scenes for which I did not find a way to change it.

DEVELOPMENT: HOOKING FRAMEWORK

1. INTRODUCTION

In the research we discussed the basics of DLL injection and API hooking, we also discussed the COM interface and its usage as well as the basics of virtual tables. In this part we will try to discuss these topics a bit more in detail. To do this we will be using the ReShade framework as a basis. ReShade is an advanced post-processing framework for virtually any game using DirectX 9, 10, 11 and OpenGL.

ReShade[1] makes use of the type of DLL injection we discussed earlier and serves as a good base for hooking different DLLs. My code at the end of this project is a heavily modified version, having redone the input and logging systems as well as removed the post-processing code and replaced it with the code for our goal of exporting textures and models and replacing textures at runtime.

For the API hooking, ReShade makes use of the MinHook library. MinHook is a minimalistic 32 and 64 bit Windows API hooking library. It takes care of creating the detour and trampoline functions which we discussed earlier in 'Research 1.3. API Hooking'.

MinHook in its turn makes use of the Hacker Disassembler Engine (HDE) for runtime disassembling functions and creating the detours and trampolines.

In this topic we will start of by explaining how we can setup the basic DLL with the correct exports for the DLL injection using DLL replacement. Followed by how the actual hooking is managed in our framework.

[1] Open-source on github.com/crosire/reshade

2. DLL INJECTION

Creating a DLL

Similar to creating a simple win32 console application and specifying the entry point in C++ with the 'main' function. When creating a DLL, we can optionally specify an entry point function. This function is then called when a process or thread wants to attach (load) themselves to the DLL or detach (unload) themselves from the DLL. We can then use this entry point function to initialize or destroy data structures required by the DLL.

The basic entry point layout is as follows:

```
BOOL APIENTRY DllMain(
    HANDLE hModule,// Handle to DLL module
    DWORD ul_reason_for_call,// Reason for calling function
    LPVOID lpReserved) // Reserved
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
        // A process is loading the DLL.
        break;
    case DLL_THREAD_ATTACH:
        // A process is creating a new thread.
        break;
    case DLL_THREAD_DETACH:
        // A thread exits normally.
        break;
    case DLL_PROCESS_DETACH:
        // A process unloads the DLL.
        break;
    default:
        // Something went wrong, this should never be triggered.
        break;
    }

    return TRUE;
}
```

FIGURE 6. DLL ENTRY POINT

Since our DLL will be loaded by a process, we can use the 'process attach' and 'process detach' to initialize and clean up our framework.

Process Attach

```
case DLL_PROCESS_ATTACH:
{
    //Disables the DLL_THREAD_ATTACH and DLL_THREAD_DETACH notifications for the specified DLL.
    //(This can reduce the size of the working set for some applications.)
    DisableThreadLibraryCalls(hModule);
    g_module_handle = hModule;

    Logger::LogInfo(L"Registering modules.");

    //Retrieve the path of our module (where did we place the DLL)
    runtime::s_dxhooks_dll_path = filesystem::get_module_path(hModule);
    //Retrieve path of the executable file of the current process
    runtime::s_target_executable_path = filesystem::get_module_path(nullptr);
    const filesystem::path system_path = filesystem::get_special_folder_path(filesystem::special_folder::system);

    //Register and attempt to hook the DLL's we want to use.
    HookManager::Register_module(system_path / "d3d11.dll");
    HookManager::Register_module(system_path / "dxgi.dll");
    HookManager::Register_module(system_path / "user32.dll");

    Logger::LogInfo(L"Modules registered.");

    break;
}
```

FIGURE 7. DLL PROCESS ATTACH INITIALIZATION

During the initialization we setup our global paths of where our DLL is and where the application we attached to is located which we can later use to look for other DLLs to use for chain-loading or writing to a log file in a sensible folder or exporting data. Chain-loading is a technique to allow other 3rd party tools which use the same DLL injection to still be loaded. Here we also attempt to register and hook the DLLs we want to use.

Process Detach

```
case DLL_PROCESS_DETACH:
{
    DisableThreadLibraryCalls(hModule);

    //Clear our input data (initialized during runtime object initialization)
    Input::Uninstall();

    Logger::LogInfo(L"Uninstalling hooks.");
    //Uninstall hooks and Uninitialize MinHook when no hooks left.
    HookManager::Uninstall();
    Logger::LogInfo(L"Hooks uninstalled.");

    //Clean up the logger
    Logger::Release();
    break;
}
```

FIGURE 8. DLL PROCESS DETACH CLEAN UP

During the cleanup we have to make sure to uninstall all our hooks to allow the program to exit correctly. We also make sure to cleanup left over input data and our logger.

Export and import functions

Exports are functions in a DLL that are made available to the outside. These are the functions that can be used by another application. These are also the functions that we can try and hook using our hooking framework.

Imports are the opposite, these are functions defined in an application that should be imported from a DLL. Usually a header file defines both the export and import function separated with an 'ifdef' statement.

To export a function you can either add the function keyword '__declspec(dllexport)' or declare it in a module-definition (.def) file which lists the exported DLL functions.

To import a function you use the '__declspec(dllimport)' function keyword.

In our framework we will make use of the module-definition (def) file as this seems to work better. The def file is used by the Visual Studio Linker at compile time to add the correct headers and exports to the DLL.

Typically a def file contains a 'LIBRARY' statement to define the name of the DLL where it I used. This statement is optional. For adding exports the 'EXPORTS' statement is used, followed by a list of function names with optionally their ordinal values. It is also possible to already define a function to redirect to as can be seen for the user32.dll functions.

```
EXPORTS
    ; d3d11.dll
    D3D11CreateDevice
    D3D11CreateDeviceAndSwapChain

    ; dxgi.dll
    CreateDXGIFactory
    CreateDXGIFactory1
    CreateDXGIFactory2

    DXGIDumpJournal
    DXGIReportAdapterConfiguration

    ; user32.dll
    RegisterClassA = HookRegisterClassA
    RegisterClassW = HookRegisterClassW
    RegisterClassExA = HookRegisterClassExA
    RegisterClassExW = HookRegisterClassExW
    RegisterRawInputDevices = HookRegisterRawInputDevices
    GetMessageA = HookGetMessageA
    GetMessageW = HookGetMessageW
    PeekMessageA = HookPeekMessageA
    PeekMessageW = HookPeekMessageW
    GetCursorPos = HookGetCursorPosition
    SetCursorPos = HookSetCursorPosition
```

FIGURE 9. EXPORTS MODULE-DEFINITION FILE WITH DX11 AND USER32 FUNCTIONS TO EXPORT

The functions we declare here are functions the target application has to call first for us to hook them and create our hooked objects.

Name mangling

In C++ at compile time every function gets mangled by default. This is basically gives each function a unique name with some code for the return and parameter types as well as class hierarchy. The mangling is useful when dealing with inheritance and to allow the same type and function definition to be reused for a different class or a derived class. When using API's on the other hand this can be unwanted. Therefore common practice for many C++ API's is to declare functions to be interpreted as C-style functions to prevent the compiler from mangling the function name.

Where to place the DLL

We know that applications don't use fixed paths. In fact it is usually handled by Windows. To make sure our DLL is loaded first we want to place it where the application or Windows (LoadLibrary function) will look first.

Windows searches for the DLL in this order:

1. The directory where the executable module for the current process is located.
2. The current directory.
3. The Windows system directory. The GetSystemDirectory function retrieves the path of this directory.
4. The Windows directory. The GetWindowsDirectory function retrieves the path of this directory.
5. The directories listed in the PATH environment variable.

3. API HOOKING

Register Module

The 'Register_Module' function attempts to hook the passed through library. To do this it first tries to hook the Win32 'LoadLibrary' functions so we can override the default behavior of these to setup the detour and trampolines.

```
void Register_module(const filesystem::path &target_path)
{
    Install(reinterpret_cast<Hook::func_ptr>(&LoadLibraryA), reinterpret_cast<Hook::func_ptr>(&HookLoadLibraryA));
    Install(reinterpret_cast<Hook::func_ptr>(&LoadLibraryExA), reinterpret_cast<Hook::func_ptr>(&HookLoadLibraryExA));
    Install(reinterpret_cast<Hook::func_ptr>(&LoadLibraryW), reinterpret_cast<Hook::func_ptr>(&HookLoadLibraryW));
    Install(reinterpret_cast<Hook::func_ptr>(&LoadLibraryExW), reinterpret_cast<Hook::func_ptr>(&HookLoadLibraryExW));

    //...

}
```

FIGURE 10. 'REGISTER_MODULE' HOOK WIN32 'LOADLIBRARY'

Once these functions are successfully hooked we have two cases for hooking the specified library. The first is the library we aim to use for injection. The second is a library we aim to simply hook the functions and change the functionality when it is used by the target application.

Example, we want to use DirectX 11 as our injection entry point, so we name our DLL d3d11.dll. The application starts and enters the entry point of our DLL. We now try to hook the d3d11.dll by registering it. Since we want to hook these functions right away we use our 'Install' function to setup the hook. Besides DX11 we also want to hook the User32.dll which has functions for creating windows and dealing with windows messages (e.g. input events).

Since we don't know if the application will use all the possible DLLs we can hook, we start by only installing our injection target DLL and add the others to a queue which will be checked and hooked the first time one of its functions is used.

```
void Register_module(const filesystem::path &target_path)
{
    Install(reinterpret_cast<Hook::func_ptr>(&LoadLibraryA), reinterpret_cast<Hook::func_ptr>(&HookLoadLibraryA));
    Install(reinterpret_cast<Hook::func_ptr>(&LoadLibraryExA), reinterpret_cast<Hook::func_ptr>(&HookLoadLibraryExA));
    Install(reinterpret_cast<Hook::func_ptr>(&LoadLibraryW), reinterpret_cast<Hook::func_ptr>(&HookLoadLibraryW));
    Install(reinterpret_cast<Hook::func_ptr>(&LoadLibraryExW), reinterpret_cast<Hook::func_ptr>(&HookLoadLibraryExW));

    Logger::LogFormat(Info, L"Registering hooks for %s ...", target_path.wstring().c_str());

    const auto target_filename = target_path.filename_without_extension();
    const auto replacement_filename = filesystem::get_module_path(g_module_handle).filename_without_extension();

    if (target_filename == replacement_filename)
    {
        Logger::LogInfo(L"> Delayed until first call to an exported function.");

        s_export_hook_path = target_path;
    }
    else
    {
        HMODULE handle = nullptr;

        if (is_module_loaded(target_path, handle))
        {
            Logger::LogInfo(L"> Libraries loaded.");


            install(handle, g_module_handle, hook_method::function_hook);
        }
        else
        {
            Logger::LogInfo(L"> Delayed.");

            s_delayed_hook_paths.push_back(target_path);
        }
    }
}
```

FIGURE 11. 'REGISTER_MODULE' FUNCTION

Install hook

Hook

For our hooking framework we want a rather generic setup that we can use for different types of hooks. We also want to keep track of all the hooks we made without hardcoding them. To do this we have a struct 'Hook' which holds the detour (replacement) and trampoline functions and uses MinHook to install the hooks.

```
struct Hook
{
public:
    typedef void * func_ptr;
    enum class status
    {
        unknown = -1,
        success,
        not_executable = 7,
        unsupported_function,
        allocation_failure,
        memory_protection_failure,
    };

    Hook();
    Hook(func_ptr target, func_ptr replacement);

    bool IsValid() const;

    bool IsEnabled() const;
    bool IsInstalled() const;

    bool Enable(bool enable = true) const;
    status Install();
    status Uninstall();

    func_ptr Call() const;
    template <typename T>
    inline T Call() const
    {
        return reinterpret_cast<T>(Call());
    }


    func_ptr fpTarget, fpReplacement, fpTrampoline;
};
```

FIGURE 12. HOOKING FRAMEWORK 'HOOK' STRUCTURE

HookManager

To keep track of the installed hooks and to manage the different types of hooking we use a ‘HookManager’. The HookManager is just a namespace with global functions because we don’t want to have to create an object and pass it through to everywhere. We simply want to hook and unhook functions, use them and keep track of them.

```
namespace HookManager
{
    /// <summary>
    /// Install hook for the specified target function.
    /// </summary>
    /// <param name="target">The address of the target function.</param>
    /// <param name="replacement">The address of the hook function.</param>
    /// <returns>The status of the hook installation.</returns>
    bool Install(Hook::func_ptr target, Hook::func_ptr replacement);
    /// <summary>
    /// Install hook for the specified virtual function table entry.
    /// </summary>
    /// <param name="vtable">The virtual function table pointer of the object to targeted object.</param>
    /// <param name="offset">The index of the target function in the virtual function table.</param>
    /// <param name="replacement">The address of the hook function.</param>
    /// <returns>The status of the hook installation.</returns>
    bool Install(Hook::func_ptr vtable[], unsigned int offset, Hook::func_ptr replacement);
    /// <summary>
    /// Uninstall all previously installed hooks.
    /// Only call this function as long as the loader-lock is active, since it is not thread-safe.
    /// </summary>
    void Uninstall();
    /// <summary>
    /// Register the matching exports in the specified module and install or delay their hooking.
    /// Only call this function as long as the loader-lock is active, since it is not thread-safe.
    /// </summary>
    /// <param name="path">The file path to the target module.</param>
    void Register_module(const filesystem::path &path);

    /// <summary>
    /// Call the original/trampoline function for the specified hook.
    /// </summary>
    /// <param name="replacement">The address of the hook function which was previously used to install a hook.</param>
    /// <returns>The address of original/trampoline function.</returns>
    Hook::func_ptr Call(Hook::func_ptr replacement);
    template <typename T>
    inline T Call(T replacement)
    {
        return reinterpret_cast<T>(Call(reinterpret_cast<Hook::func_ptr>(replacement)));
    }
};
```

FIGURE 13. 'HOOKMANAGER' NAMESPACE

Hooking methods

For installing a hook we have three different situations to take care of.

- Function hook
- Virtual Table hook (COM objects)
- Export hook (exported DLL functions)

Function hook

The function hook is the most basic form where we simply use the detour and trampoline system to redirect the original call to our function and allow us to call the target function when we want.

```
Hook hook(target, replacement);
hook.fpTrampoline = target;

Hook::status status = Hook::status::unknown;

status = hook.Install();
```

FIGURE 14. FUNCTION HOOK

Virtual Table hook

The virtual table hook is used to hook function from the virtual function table from an object. In our case we will use this to hook COM object functions, such as the 'IDXGIFactory::CreateSwapChain' function. These functions are part of an object so we first have to create an object and then hook the function using the offset of the function in its virtual table.

```
bool Install(Hook::func_ptr vtable[], unsigned int offset, Hook::func_ptr replacement)
{
    assert(vtable != nullptr);
    assert(replacement != nullptr);

    DWORD protection = PAGE_READONLY;
    Hook::func_ptr &target = vtable[offset];

    if (VirtualProtect(&target, sizeof(Hook::func_ptr), protection, &protection))
    {
        const std::lock_guard<std::mutex> lock(s_mutex_vtable_addresses);
        const auto insert = s_vtable_addresses.emplace(target, &target);

        VirtualProtect(&target, sizeof(Hook::func_ptr), protection, &protection);

        if (insert.second)
        {
            if (target != replacement && install(target, replacement, hook_method::vtable_hook))
            {
                return true;
            }

            s_vtable_addresses.erase(insert.first);
        }
        else
        {
            return insert.first->first == target;
        }
    }

    return false;
}
```

FIGURE 15. INSTALL VIRTUAL TABLE HOOK

Export hook

With export hooking we try to load a DLL, check all the exported functions and try to match as many of them with our own exported functions and do a function hook for each of the matches found.

First we look for all the exports in the target DLL (target module) and our own DLL (replacement module). If no exports found in the target we don't have to do anything.

```
bool install(const HMODULE target_module, const HMODULE replacement_module, hook_method method)
{
    assert(target_module != nullptr);
    assert(replacement_module != nullptr);

    // Load export tables
    const auto target_exports = get_module_exports(target_module);
    const auto replacement_exports = get_module_exports(replacement_module);

    if (target_exports.empty())
    {
        Logger::LogInfo(L"> Empty export table! Skipped.");

        return false;
    }
```

FIGURE 16. GET MODULE EXPORTS FOR EXPORT HOOKING

If there are exports we try to match them with our own and if a match is found we cache it.

```
size_t install_count = 0;
std::vector<std::pair<Hook::func_ptr, Hook::func_ptr>> matches;
matches.reserve(replacement_exports.size());

Logger::LogInfo(L"> Dumping matches in export table:");
Logger::LogInfo(L"  +--------------------+---------+----------------------------------------------------+");
Logger::LogInfo(L"  | Address            | Ordinal | Name                                               |");
Logger::LogInfo(L"  +--------------------+---------+----------------------------------------------------+");

// Analyze export table
for (const auto &symbol : target_exports)
{
    if (symbol.name == nullptr || symbol.address == nullptr)
    {
        continue;
    }

    // Find appropriate replacement
    const auto it = std::find_if(replacement_exports.cbegin(), replacement_exports.cend(),
        [&symbol](const module_export &moduleexport)
    {
        return std::strcmp(moduleexport.name, symbol.name) == 0;
    });

    // Filter uninteresting functions, add matches
    if (it != replacement_exports.cend() &&
        std::strcmp(symbol.name, "DXGIReportAdapterConfiguration") != 0 &&
        std::strcmp(symbol.name, "DXGIDumpJournal") != 0)
    {
        Logger::LogFormat(Info, L"  | 0x%-16p | %-7u | %-50S |", symbol.address, symbol.ordinal, LPCWSTR(symbol.name));

        matches.push_back({ symbol.address, it->address });
    }
}

Logger::LogInfo(L"  +--------------------+---------+----------------------------------------------------+");
Logger::LogFormat(Info, L"> Found %u match(es). Installing ...", matches.size());
```

FIGURE 17. FIND MATCHING EXPORTS

Finally we try to install our matched exports using the function hook.

```
    // Hook matching exports
    for (const auto &match : matches)
    {
        if (install(match.first, match.second, method))
        {
            install_count++;
        }
    }

    return install_count != 0;
}
```

FIGURE 18. INSTALL MATCHED EXPORTS

Getting module exports

We can get the exported functions of a DLL by looking at the file header. For Windows DLLs use the same format as EXEs. This makes it easier to use simple reinterpret casting to cast the binary file to the header structures provided by the Win32 API.

```
std::vector<module_export> get_module_exports(HMODULE handle)
{
    std::vector<module_export> exports;
    const auto imagebase = reinterpret_cast<const BYTE *>(handle);
    const auto imageheader = reinterpret_cast<const IMAGE_NT_HEADERS *>
        (imagebase + reinterpret_cast<const IMAGE_DOS_HEADER *>(imagebase)->e_lfanew);

    if (imageheader->Signature != IMAGE_NT_SIGNATURE
        || imageheader->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].Size == 0)
    {
        return exports;
    }

    const auto exportdir =
        reinterpret_cast<const IMAGE_EXPORT_DIRECTORY *>
        (imagebase + imageheader->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);
    const auto exportbase = static_cast<WORD>(exportdir->Base);

    if (exportdir->NumberOfFunctions == 0)
    {
        return exports;
    }

    const auto count = static_cast<size_t>(exportdir->NumberOfNames);
    exports.reserve(count);

    for (size_t i = 0; i < count; i++)
    {
        module_export symbol;
        symbol.ordinal = reinterpret_cast<const WORD *>
            (imagebase + exportdir->AddressOfNameOrdinals)[i] + exportbase;
        symbol.name = reinterpret_cast<const char *>
            (imagebase + reinterpret_cast<const DWORD *>(imagebase + exportdir->AddressOfNames)[i]);
        symbol.address = const_cast<void *>(reinterpret_cast<const void *>
            (imagebase + reinterpret_cast<const DWORD *>(imagebase + exportdir->AddressOfFunctions)[symbol.ordinal - exportbase]));

        exports.push_back(std::move(symbol));
    }

    return exports;
}
```

FIGURE 19. 'GET_MODULE_EXPORTS' FUNCTION

Hooked LoadLibrary

During the register module function we hooked the load library functions to be able override the behavior. The main goal for this was for the delayed hooking of modules. When the target application uses 'LoadLibrary' to import a module to use its functions we now intercept that call and attempt to hook the functions we want.

If a library is already installed we simply pass through the handle.

```
HMODULE WINAPI HookLoadLibraryA(LPCSTR lpFileName)
{
    static const auto trampoline = call_unchecked(&HookLoadLibraryA);

    const HMODULE handle = trampoline(lpFileName);

    if (handle == nullptr || handle == g_module_handle)
    {
        return handle;
    }

    {
        const std::lock_guard<std::mutex> lock(s_mutex_delayed_hook_paths);
        const auto remove = std::remove_if(s_delayed_hook_paths.begin(), s_delayed_hook_paths.end(),
            [lpFileName](const filesystem::path &path)
        {
            HMODULE delayed_handle = nullptr;

            if (!is_module_loaded(path, delayed_handle))
            {
                return false;
            }

            Logger::LogInfo(L"Installing delayed hooks for ? (Just loaded via 'LoadLibraryA(\"?\")') ...");

            return install(delayed_handle, g_module_handle, hook_method::function_hook);
        });

        s_delayed_hook_paths.erase(remove, s_delayed_hook_paths.end());
    }

    return handle;
}
```

FIGURE 20. HOOKED 'LOADLIBRARY' FUNCTION

Get Virtual table from instance

Since the pointer to the virtual table of COM objects and typical C++ class objects is the first 4 bytes, we simply return the address of the object instance interpreted as a pointer. Which effectively returns the pointer to the virtual table.

```
template <typename T>
inline Hook::func_ptr *vtable_from_instance(T *instance)
{
    static_assert(std::is_polymorphic<T>::value, "can only get virtual function table from polymorphic types");

    return *reinterpret_cast<Hook::func_ptr **>(instance);
}
```

FIGURE 21. FUNCTION TO RETRIEVE POINTER TO VTABLE FROM AN OBJECT INSTANCE

DEVELOPMENT: DIRECTX 11

1. INTRODUCTION

After having setup our hooking framework we can now start by hooking the DirectX 11 pipeline. Since DirectX 11 works with different objects that offer functionalities and not just functions alone we have to acquire access to these. The easiest way to do this is by hooking the one function that will always be called to create a DirectX 11 application. This is the 'D3D11CreateDevice' function. This is a static function which will create a device for the application to use. By hooking this device we can create our own fake device which implements all our functionalities whilst keeping a reference to the actual device the application wishes to create.

Since we're trying to mimic the original d3d11.dll we have to export the some of the same functions used as an entry point.

Our first function is the 'D3D11CreateDevice(..)'. Since we also want to hook the swap-chain object for the present calls so we can hook into the update loop and have the option to add data at runtime we redirect the create device function to a hooked 'D3D11CreateDeviceAndSwapChain' function.

```
HOOK_EXPORT HRESULT WINAPI D3D11CreateDevice(IDXGIAdapter *pAdapter, D3D_DRIVER_TYPE DriverType, HMODULE Software,
    UINT Flags, const D3D_FEATURE_LEVEL *pFeatureLevels, UINT FeatureLevels, UINT SDKVersion, ID3D11Device **ppDevice,
    D3D_FEATURE_LEVEL *pFeatureLevel, ID3D11DeviceContext **ppImmediateContext)
{
    Logger::LogFormat(Info, L"Redirecting 'D3D11CreateDevice'(%p, %u, %p, %#x, %p, %u, %u, %p, %u, %p)' ...",
        pAdapter, DriverType, Software, Flags, pFeatureLevels, FeatureLevels, SDKVersion, *ppDevice, pFeatureLevel, *ppImmediateContext);
    Logger::LogInfo(L"> Passing on to 'D3D11CreateDeviceAndSwapChain'.");

    return D3D11CreateDeviceAndSwapChain(pAdapter, DriverType, Software, Flags, pFeatureLevels, FeatureLevels, SDKVersion,
        nullptr, nullptr, ppDevice, pFeatureLevel, ppImmediateContext);
}
```

FIGURE 22. HOOKED 'D3D11CREATEDEVICE'

In our create device and swap-chain function we start out by calling the trampoline to create an actual device and swap-chain object.

```
HOOK_EXPORT HRESULT WINAPI D3D11CreateDeviceAndSwapChain(IDXGIAdapter *pAdapter, D3D_DRIVER_TYPE DriverType, HMODULE Software, UINT Flags,
    const D3D_FEATURE_LEVEL *pFeatureLevels, UINT FeatureLevels, UINT SDKVersion, const DXGI_SWAP_CHAIN_DESC *pSwapChainDesc,
    IDXGISwapChain **ppSwapChain, ID3D11Device **ppDevice, D3D_FEATURE_LEVEL *pFeatureLevel, ID3D11DeviceContext **ppImmediateContext)
{
    Logger::LogFormat(Info, L"Redirecting 'D3D11CreateDeviceAndSwapChain'(%p, %u, %p, %#x, %p, %u, %u, %p, %u, %p)' ...",
        pAdapter, DriverType, Software, Flags, pFeatureLevels, FeatureLevels, SDKVersion, *ppDevice, pFeatureLevel, *ppImmediateContext);


#ifdef _DEBUG
    //Flags |= D3D11_CREATE_DEVICE_DEBUG;
    Flags &= ~D3D11_CREATE_DEVICE_PREVENT_ALTERING_LAYER_SETTINGS_FROM_REGISTRY;
#endif

    D3D_FEATURE_LEVEL FeatureLevel = D3D_FEATURE_LEVEL_11_0;

    HRESULT hr = HookManager::Call(&D3D11CreateDeviceAndSwapChain)(pAdapter, DriverType, Software, Flags, pFeatureLevels, FeatureLevels,
        SDKVersion, pSwapChainDesc, ppSwapChain, ppDevice, &FeatureLevel, ppImmediateContext);

    if (FAILED(hr))
    {
        Logger::LogFormat(Warning, L"> 'D3D11CreateDeviceAndSwapChain' failed with error code %#x!", hr);
        return hr;
    }
```

FIGURE 23. HOOKED 'D3D11CREATEDEVICESWAPCHAIN' TRAMPOLINE CALL

If this succeeds we want to create our fake objects (proxy) to handle our custom actions. If we successfully create our proxy objects we overwrite the ones created by the original API with our proxy objects, whilst keeping a reference to the original in our objects.

```
if (ppDevice != nullptr)
{
    IDXGIDevice1 *dxgidevice = nullptr;
    ID3D11Device *const device = *ppDevice;
    ID3D11DeviceContext *devicecontext = nullptr;

    assert(device != nullptr);

    device->QueryInterface(&dxgidevice);
    device->GetImmediateContext(&devicecontext);

    assert(dxgidevice != nullptr);
    assert(devicecontext != nullptr);

    const auto device_proxy = new D3D11Device(device);
    const auto devicecontext_proxy = new D3D11DeviceContext(device_proxy, devicecontext);
    device_proxy->_dxgi_device = new DXGIDevice(dxgidevice, device_proxy);
    device_proxy->_immediate_context = devicecontext_proxy;

    if (pSwapChainDesc != nullptr)
    {
        assert(ppSwapChain != nullptr);

        if (pAdapter != nullptr)
        {
            pAdapter->AddRef();
        }
        else
        {
            hr = device_proxy->_dxgi_device->GetAdapter(&pAdapter);

            assert(SUCCEEDED(hr));
        }

        IDXGIFactory *factory = nullptr;

        hr = pAdapter->GetParent(IID_PPV_ARGS(&factory));

        assert(SUCCEEDED(hr));

        Logger::LogInfo(L"> Calling 'IDXGIFactory::CreateSwapChain':");

        hr = factory->CreateSwapChain(device_proxy, const_cast<DXGI_SWAP_CHAIN_DESC *>(pSwapChainDesc), ppSwapChain);

        factory->Release();
        pAdapter->Release();
    }

    if (SUCCEEDED(hr))
    {
        Logger::LogFormat(Info, L"Returning 'IDXGIDevice1' object %p", device_proxy->_dxgi_device);
        Logger::LogFormat(Info, L"Returning 'ID3D11Device' object %p", device_proxy);

        *ppDevice = device_proxy;

        if (ppImmediateContext != nullptr)
        {
            Logger::LogFormat(Info, L"Returning 'ID3D11DeviceContext' object %p", devicecontext_proxy);

            devicecontext_proxy->AddRef();
            *ppImmediateContext = devicecontext_proxy;
        }
    }
    else
    {
        device_proxy->Release();
    }
}
```

FIGURE 24. HOOKED 'D3D11CREATEDEVICESWAPCHAIN' CREATE PROXY OBJECTS

To setup our runtime framework we hooked the ‘IDXGIFactory_CreateSwapChain’ function. By calling the ‘CreateSwapChain’ function when creating the d3d11 proxy objects the ‘IDXGIFactory_CreateSwapChain’ function is called down the line. Within this function we create a proxy object and return this to the target application. Here we also setup our runtime objects which are called every frame with the present function.

```
HRESULT STDMETHODCALLTYPE IDXGIFactory_CreateSwapChain(IDXGIFactory *pFactory, IUnknown *pDevice, DXGI_SWAP_CHAIN_DESC *pDesc, IDXGISwapChain **ppSwapChain)
{
    Logger::LogFormat(Info, L"Redirecting 'IDXGIFactory::CreateSwapChain(%p, %p, %p, %p)' ...", pFactory, pDevice, pDesc, ppSwapChain);

    IUnknown *device_orig = pDevice;
    D3D11Device *device_d3d11 = nullptr;

    if (pDevice == nullptr || pDesc == nullptr || ppSwapChain == nullptr)
    {
        return DXGI_ERROR_INVALID_CALL;
    }
    else if (SUCCEEDED(pDevice->QueryInterface(&device_d3d11)))
    {
        device_orig = device_d3d11->_orig;

        device_d3d11->Release();
    }

    dump_swapchain_desc(*pDesc);

    const HRESULT hr = HookManager::Call(&IDXGIFactory_CreateSwapChain)(pFactory, device_orig, pDesc, ppSwapChain);

    if (FAILED(hr))
    {
        Logger::LogFormat(Warning, L"> 'IDXGIFactory::CreateSwapChain' failed with error code 0x%u!", hr);

        return hr;
    }
    IDXGISwapChain *const swapchain = *ppSwapChain;

    DXGI_SWAP_CHAIN_DESC desc;
    swapchain->GetDesc(&desc);

    if ((desc.BufferUsage & DXGI_USAGE_RENDER_TARGET_OUTPUT) == 0)
    {
        Logger::LogWarning(L"> Skipping swap chain due to missing 'DXGI_USAGE_RENDER_TARGET_OUTPUT' flag.");
    }
    else if (device_d3d11 != nullptr)
    {
        device_d3d11->AddRef();

        const auto runtime = std::make_shared<d3d11::d3d11_runtime>(device_d3d11->_orig, swapchain);

        if (!runtime->on_init(desc))
        {
            Logger::LogFormat(Error, L"Failed to initialize Direct3D 11 runtime environment on runtime %p.", runtime.get());
        }

        device_d3d11->_runtimes.push_back(runtime);

        *ppSwapChain = new DXGISwapChain(device_d3d11, swapchain, runtime);
    }
    else
    {
        Logger::LogWarning(L"> Skipping swap chain because it was created without a (hooked) Direct3D device.");
    }

    Logger::LogFormat(Info, L"Returning 'IDXGISwapChain' object %p.", *ppSwapChain);

    return S_OK;
}
```

FIGURE 25. HOOKED 'DXGI_CREATESWAPCHAIN'

To update our runtime we simply call it from our DXGISwapChain proxy object during the present call.

```
HRESULT STDMETHODCALLTYPE DXGISwapChain::Present(UINT SyncInterval, UINT Flags)
{
    switch (_direct3d_version)
    {
        case 11:
            assert(_runtime != nullptr);
            std::static_pointer_cast<d3d11::d3d11_runtime>(_runtime)->on_present();
            break;
    }

    return _orig->Present(SyncInterval, Flags);
}
```

FIGURE 26. HOOKED 'DXGISWAPCHAIN::PRESENT'

2. EXPORTING TEXTURES

We briefly explained the steps to take to retrieve the texture data in the research of this paper. Our goal isn't to just cache or access the data. We also want to export the textures. To do this we first try and cache the textures, then whenever the user asks for it we try to export them.

Caching textures

CreateTexture2D

During the research we determined that the 'CreateTexture2D' function will allow us to cache all the textures the application creates. Previously we decided to not just hook separate functions of DirectX11 but to create our own runtime, device and device context objects with the latter having a link to the original ones requested by the application.

Since we don't want to change the textures right from the start, we simply ask the original device object to create the texture for us. If this succeeds we are sure the texture is valid and we can continue to try and cache it.

The first step in caching the texture is to try and create a staging texture. This texture allows CPU access which we can use to hash the data to prevent caching duplicates. We also need CPU access to later write the texture data to a file.

```
HRESULT STDMETHODCALLTYPE D3D11Device::CreateTexture2D(const D3D11_TEXTURE2D_DESC *pDesc,
    const D3D11_SUBRESOURCE_DATA *pInitialData, ID3D11Texture2D **ppTexture2D)
{
    //Pass through to the original function to create the texture the application wants.
    auto hr = _orig->CreateTexture2D(pDesc, pInitialData, ppTexture2D);

    if (SUCCEEDED(hr))
    {
        d3d11::d3d11_Texture2D data;
        //Attempt to create a staging texture to cache and export, also set the hash value for duplicate comparison.
        auto tempHR = GetStagingTextureHash(data, pDesc, ppTexture2D);
        if (FAILED(tempHR))
        {
            Logger::LogHResult(tempHR, L"D3D11Device::CreateTexture2D(..) - GetStagingTextureHash(..) failed.");
            return hr;
        }

        //Create a resource to wrap the d3d11 data
        dxhooks::Resource* res = new dxhooks::Resource();
        res->impl = make_unique<d3d11::d3d11_Texture2D>(data);

        //Try to cache texture
        auto hrR = dxhooks::runtime::add_Textures(res);
        if (hrR == dxhooks::HR_RESOURCE::FAIL)
        {
            SafeRelease(data.texture);
            SafeDelete(res);
        }
        else if (hrR == dxhooks::HR_RESOURCE::ALREADY_EXISTS)
        {
            SafeDelete(res);
        }
    }

    return hr;
}
```

FIGURE 27. HOOKED 'CREATETEXTURE2D' FUNCTION

GetStagingTextureHash[2]

To create a staging texture ready for caching we have to follow these steps:

1. Check if texture is a multi-sampled texture (MSAA)
2. If MSAA texture, try to resolve one of the sub textures and create a staging texture from it
3. If not MSAA texture, check if it's already a staging texture
4. If already staging texture, no need to create one so continue
5. If not staging texture, create a staging texture

Once we have the staging texture all that's left to do is to cache it and hash the raw data for comparison with other textures to prevent duplicates.

To create a staging texture we have to create a 'D3D11_TEXTURE2D_DESC' description with:

- CPU Access Read, allow memory to be read from CPU
- Usage Staging, support data transfer from the GPU to the CPU.
- No bind flags, since we don't want to bind it to anything

```
// Not MSAA or staging texture
// Try to create a staging texture which can be read by the CPU (won't be usable for GPU so clean up later).
else
{
    description.CPUAccessFlags = D3D11_CPU_ACCESS_READ;
    description.Usage = D3D11_USAGE_STAGING;
    description.BindFlags = 0;

    auto hr = _orig->CreateTexture2D(&description, nullptr, &pStaging);
    if (FAILED(hr))
    {
        SafeRelease(pStaging);
        Logger::LogHResult(hr, L"Could not create a staging texture.");
        return hr;
    }

    // Copy actual resource data to staging texture
    // (Copy the entire contents of the source resource to the destination resource using the GPU.)
    _immediate_context->CopyResource(pStaging, *ppTexture2D);
}
```

FIGURE 28. CREATE STAGING TEXTURE

Then we use the original CreateTexture2D to create a texture resource which we can cache to later be able to access it and export the data. Since we don't specify the initial data, we just created an empty resource. To copy over the data from the actual texture into our staging duplicate we have to use the device context's 'CopyResource' function. This function can copy the entire content of a resource into another resource using the GPU. For more info about the limitations you can check the 'ID3D11DeviceContext::CopyResource' MSDN page.

[2] Full code for 'GetStagingTextureHash' in Appendix A

We now successfully created a duplicate of the original texture, but now with CPU access. To be able to read from a resource we have to use the device context's 'Map' and 'Unmap' functions.

```
{
    D3D11_MAPPED_SUBRESOURCE mapped;
    // Lock memory
    auto hr = _immediate_context->Map(pStaging, 0, D3D11_MAP_READ, 0, &mapped);
    //Something went wrong, return error
    if (FAILED(hr)) { ... }

    // If mapped, hash the data for duplicate comparison
    data.hash = Hash(static_cast<char*>(mapped.pData));

    // Unlock memory
    _immediate_context->Unmap(pStaging, 0);
}
```

FIGURE 29. MAP, HASH AND UNMAP STAGING TEXTURE

Map retrieves a pointer to the data contained in the subresource and denies the GPU to access it as long as it's map. In other words, it sets a memory lock. During this memory lock we can access the data with the CPU. In this case we'll use it to get a hash value using a hash function. Before we finish we have to use the unmap function to release the memory lock. Finally cache the resource in a list we can use later for exporting.

Export texture

At this point we have all the textures created by the application cached as staging textures. We can now use the DirectXTK library to do the texture exporting for us. All we have to do now is specify which resource we want to save to a file and to which file we want to write. Since DirectX does not care for file names when creating textures, there is no way for us to know the original name. So we simply make our own with the index of caching added to it.

```
void d3d11_runtime::ExportTexture(Resource* res, int i)
{
    auto sr = res->impl->as<d3d11_Texture2D>();

    stringstream ss;
    ss.str();
    ss << "Exports\\Textures\\tex_" << i << ".dds";
    auto path = runtime::s_target_executable_path.parent_path() / ss.str().c_str();

    auto hr = SaveDDSTextureToFile(_immediate_context.get(), sr->texture.get(), path.wstring().c_str());
    Logger::LogHResult(hr, path.wstring().c_str());
}
```

FIGURE 30. EXPORT TEXTURE

To improve performance we use a thread-pool to queue the export of each texture. This way multiple textures can be seemingly exported at the same time and we don't lock up the main thread whilst writing to files.

```
void runtime::ExportTextures()
{
    Logger::LogInfo(L"runtime::ExportTextures() start");

    CheckAndCreateExportFolder("Textures");

    int i = 0;
    for (auto res : s_TextureResources)
    {
        m_ThreadPool.enqueue([&](int id, Resource * pResource)
        {
            Logger::LogFormat(Info, L"Start ThreadPool Task ExportTexture %u", id);
            ExportTexture(pResource, id);
            Logger::LogFormat(Info, L"End ThreadPool Task ExportTexture %u", id);
        }, ++i, res);
    }

}
```

FIGURE 31. QUEUE TEXTURE EXPORTS IN THREAD-POOL

3. EXPORTING MODELS

The basic idea behind exporting models is the same as exporting textures. Cache the model data, and export it when requested by the user. As explained in the research about getting model data, it is not as straightforward as exporting textures though. Models are built up from different buffers and interpretations of the data within those buffers. Another issue is that we don't know when which buffer is created and which model they'll be used for.

Before we can export models we have to cache the Index buffers, Vertex buffers and the Input Layouts. Then during the draw call we can determine which combination of index buffer, vertex buffer, input layout and primitive topology is used to draw a certain mesh. At this point we can cache it and export the models.

Caching Index and Vertex buffers

As we explained before, in DirectX 11 we can use the device's 'CreateBuffer' function to get all the buffers that are created. From these we only want the vertex and index buffers. We can check which one is created using the bind flag of the description that is passed through. The same way we created a staging texture, we can create a staging buffer here which we can cache and later use to access the raw data for exporting.

Since we don't know which buffer will be linked with which mesh during the draw call we can't prevent duplicates as that would mean possibly missing connections. What we can do is prevent duplicates based on the instance of the object used by comparing the addresses. If one was already created for that address we don't have to cache it again.

We cache the index and vertex buffers to separate lists so we can easily differentiate between them later without having to recheck the descriptions.

```
HRESULT STDMETHODCALLTYPE D3D11Device::CreateBuffer(const D3D11_BUFFER_DESC *pDesc,
    const D3D11_SUBRESOURCE_DATA *pInitialData, ID3D11Buffer **ppBuffer)
{
    auto hr = _orig->CreateBuffer(pDesc, pInitialData, ppBuffer);

    if (SUCCEEDED(hr) && ((pDesc->BindFlags == D3D11_BIND_INDEX_BUFFER) || (pDesc->BindFlags == D3D11_BIND_VERTEX_BUFFER)))
    {
        //create copy desc with cpu read access and staging
        D3D11_BUFFER_DESC newDesc;
        newDesc.ByteWidth = pDesc->ByteWidth;
        newDesc.CPUAccessFlags = D3D11_CPU_ACCESS_READ;
        newDesc.Usage = D3D11_USAGE_STAGING;
        newDesc.MiscFlags = 0;
        newDesc.BindFlags = 0;

        ID3D11Buffer *pCopyBuffer = nullptr;
        auto copyHR = _orig->CreateBuffer(&newDesc, nullptr, &pCopyBuffer);

        _immediate_context->CopyResource(pCopyBuffer, *ppBuffer);

        if (SUCCEEDED(copyHR))
        {
            if (pDesc->BindFlags == D3D11_BIND_INDEX_BUFFER)
            {
                d3d11::d3d11_IndexBufferData idBuffer;
                idBuffer.pReadIndexBuffer = pCopyBuffer;
                idBuffer.pIndexBuffer = *ppBuffer;
                idBuffer.hash = HashPointer(*ppBuffer);

                dxhooks::Resource* res = new dxhooks::Resource();
                res->impl = make_unique<d3d11::d3d11_IndexBufferData>(idBuffer);

                auto hrRes = dxhooks::runtime::add_IndexBuffer(res);
                if (hrRes == dxhooks::HR_RESOURCE::FAIL)
                {
                    SafeRelease(idBuffer.pReadIndexBuffer);
                    SafeRelease(idBuffer.pIndexBuffer);

                    SafeDelete(res);
                }
                else if (hrRes == dxhooks::HR_RESOURCE::ALREADY_EXISTS)
                {
                    SafeDelete(res);
                }
            }
            else if (pDesc->BindFlags == D3D11_BIND_VERTEX_BUFFER)
            {
                d3d11::d3d11_VertexBufferData vBuffer;
                vBuffer.pReadVertexBuffers = pCopyBuffer;
                vBuffer.pVertexBuffers = *ppBuffer;
                vBuffer.hash = HashPointer(*ppBuffer);

                dxhooks::Resource* res = new dxhooks::Resource();
                res->impl = make_unique<d3d11::d3d11_VertexBufferData>(vBuffer);

                auto hrRes = dxhooks::runtime::add_VertexBuffer(res);
                if (hrRes == dxhooks::HR_RESOURCE::FAIL)
                {
                    SafeRelease(vBuffer.pReadVertexBuffers);
                    SafeRelease(vBuffer.pVertexBuffers);

                    SafeDelete(res);
                }
                else if (hrRes == dxhooks::HR_RESOURCE::ALREADY_EXISTS)
                {
                    SafeDelete(res);
                }
            }
        }
    }


    return hr;
}
```

FIGURE 32. HOOKED 'CREATEBUFFER' FUNCTION

Caching Input Layouts

To be able to reverse engineer the data in the vertex buffer we need the Input Layout to determine which data each vertex holds and the size of it. For DirectX 11 there are some fixed semantic names which we can use to at least determine the most common data types such as position, normal, texture coordinates.

Again we can't prevent duplicates based on data because we don't know which ones the application will use at runtime. To improve the comparisons later on during export we wrap the input layout descriptions in a custom structure using an enum for the semantic names.

```
HRESULT STDMETHODCALLTYPE D3D11Device::CreateInputLayout(const D3D11_INPUT_ELEMENT_DESC *pInputElementDescs, UINT NumElements,
    const void *pShaderBytecodeWithInputSignature, SIZE_T BytecodeLength, ID3D11InputLayout **ppInputLayout)
{
    auto hr = _orig->CreateInputLayout(pInputElementDescs, NumElements, pShaderBytecodeWithInputSignature, BytecodeLength, ppInputLayout);

    if (SUCCEEDED(hr))
    {
        //Duplicate Description with semantic Enum for easy comparison later
        ILDescription * pInputElementDescsCopy = new ILDescription[NumElements];
        for (auto i = 0; i < NumElements; ++i)
        {
            //pILSemantics[i].SemanticName = pInputElementDescs[i].SemanticName;
            pInputElementDescsCopy[i].SemanticIndex = pInputElementDescs[i].SemanticIndex;
            pInputElementDescsCopy[i].AlignedByteOffset = pInputElementDescs[i].AlignedByteOffset;
            pInputElementDescsCopy[i].Format = pInputElementDescs[i].Format;

            //Semantic Type
            if (strcmp(pInputElementDescs[i].SemanticName, "POSITION") == 0)pInputElementDescsCopy[i].SemanticType = ILSemantic::POSITION;
            else if (strcmp(pInputElementDescs[i].SemanticName, "NORMAL") == 0)pInputElementDescsCopy[i].SemanticType = ILSemantic::NORMAL;
            else if (strcmp(pInputElementDescs[i].SemanticName, "COLOR") == 0)pInputElementDescsCopy[i].SemanticType = ILSemantic::COLOR;
            else if (strcmp(pInputElementDescs[i].SemanticName, "TEXCOORD") == 0)pInputElementDescsCopy[i].SemanticType = ILSemantic::TEXCOORD;
            else if (strcmp(pInputElementDescs[i].SemanticName, "TANGENT") == 0)pInputElementDescsCopy[i].SemanticType = ILSemantic::TANGENT;
            else if (strcmp(pInputElementDescs[i].SemanticName, "BINORMAL") == 0)pInputElementDescsCopy[i].SemanticType = ILSemantic::BINORMAL;
            else if (strcmp(pInputElementDescs[i].SemanticName, "BLENDINDICES") == 0)pInputElementDescsCopy[i].SemanticType = ILSemantic::BLENDINDICES;
            else if (strcmp(pInputElementDescs[i].SemanticName, "BLENDWEIGHTS") == 0)pInputElementDescsCopy[i].SemanticType = ILSemantic::BLENDWEIGHTS;
            else pInputElementDescsCopy[i].SemanticType = ILSemantic::NONE;

        }

        d3d11::d3d11_InputLayoutData il;
        il.pInputLayout = *ppInputLayout;
        il.NumElements = NumElements;
        il.pInputElementDescs = pInputElementDescsCopy;
        il.hash = HashPointer(*ppInputLayout);

        dxhooks::Resource* res = new dxhooks::Resource();
        res->impl = make_unique<d3d11::d3d11_InputLayoutData>(il);

        auto hrR = dxhooks::runtime::add_InputLayout(res);
        //Clean up if exists/error
        if (hrR == dxhooks::HR_RESOURCE::FAIL)
        {
            SafeDelete(il.pInputElementDescs);
            SafeRelease(il.pInputLayout);

            SafeDelete(res);
        }
        else if (hrR == dxhooks::HR_RESOURCE::ALREADY_EXISTS)
        {
            SafeDelete(res);
        }
    }

    return hr;
}
```

FIGURE 33. HOOKED 'CREATEINPUTLAYOUT' FUNCTION

Combining during the draw calls

Now that we've cached all the relevant data it's time to determine which buffers go with which. As mentioned before, the easiest way to do this is to check it during the draw calls. The only problem is that the draw calls are made every single frame. It would be very bad to loop through a bunch of lists every frame. So instead of checking every draw call, we will only check them for a few seconds after the user asks us to. In other words, we take a snapshot of all the frames within a certain timeframe. This way we don't get all the models at once but since they're cached we can later compare and add to them when the player moves elsewhere ad takes another snapshot.

For the following example we use the 'ID3D11DeviceContext::DrawIndexed' function to try. As explained in the research about getting model data, we can use the 'ID3D11DeviceContext::IAGetPrimitiveTopology' function to retrieve the current topology set on the pipeline. We cache this here to use later during the export to determine which indices to use per face. At the end of the draw call function we still pass the original draw call to allow the application to draw the wanted models.

Matching Input Layout

As mentioned above we start off by checking if we are in a snapshot frame. If so we continue to check if an input layout is set on the pipeline, this should always be true, but in case it isn't we can just skip everything else. Next up we quickly cache the primitive topology to use for this model.

Matching the input layouts is pretty straightforward. We loop through the list of cached input layouts, compare the pointer with the current one set on the pipeline and if we have a match we cache it.

```
void STDMETHODCALLTYPE D3D11DeviceContext::DrawIndexed(UINT IndexCount, UINT StartIndexLocation, INT BaseVertexLocation)
{
    if (dxhooks::runtime::s_bSnapshotFrameActive)
    {
        ID3D11InputLayout* pInputLayout;
        _orig->IAGetInputLayout(&pInputLayout);

        if (pInputLayout)
        {
            d3d11::d3d11_Mesh mesh;
            _orig->IAGetPrimitiveTopology(&mesh.topology);

            // Match Input Layout
            auto ils = dxhooks::runtime::get_InputLayouts();
            for (auto res : ils)
            {
                auto il = res->impl->as<d3d11::d3d11_InputLayoutData>();

                if (il->pInputLayout && il->pInputLayout == pInputLayout)
                {
                    mesh.inputLayout = *il;

                    if (mesh.inputLayout.pInputElementDescs)
                        break; // Exit loop if a matching IL and good desc is found
                }
            }
```

FIGURE 34. MATCHING INPUT LAYOUT IN DRAWINDEXED

Matching Index Buffer

To match the index buffers, we first ask the device context to give us the index buffer that's currently set on the pipeline and then continue with a pointer comparison the same way we did for the input layouts. Conveniently enough the 'IAGetIndexBuffer' function also sets the buffer format and offset used which we later need to deconstruct the contents of the index buffer.

```
// Match Index Buffer
d3d11::d3d11_IndexBufferData idBuffer;
_orig->IAGetIndexBuffer(&idBuffer.pIndexBuffer, &idBuffer.Format, &idBuffer.Offset);

auto idBuffers = dxhooks::runtime::get_IndexBuffers();
for (auto res : idBuffers)
{
    auto ib = res->impl->as<d3d11::d3d11_IndexBufferData>();

    if (ib->pIndexBuffer && ib->pIndexBuffer == idBuffer.pIndexBuffer)
    {
        idBuffer.pReadIndexBuffer = ib->pReadIndexBuffer;
        break;
    }
}

mesh.indexBuffer = idBuffer;
```

FIGURE 35. MATCHING INDEX BUFER IN DRAWINDEXED

Matching Vertex Buffer

By now, matching the vertex buffers should be no mystery. We ask the device context for the current vertex buffer set on the pipeline, compare pointers and cache. For our hash we'll use the vertex buffer for now to prevent duplicates. Same as with the 'get index buffer' function the vertex one gives us some extra useful info too about the size and offset.

```
// Match Vertex Buffer
d3d11::d3d11_VertexBufferData vBuffer;
_orig->IAGetVertexBuffers(_currentVBufferStartSlot, _currentVBufferNumBuffers,
    &vBuffer.pVertexBuffers, &vBuffer.strides, &vBuffer.offsets);

auto vBuffers = dxhooks::runtime::get_VertexBuffers();
for (auto res : vBuffers)
{
    auto vb = res->impl->as<d3d11::d3d11_VertexBufferData>();

    if (vb->pVertexBuffers && vb->pVertexBuffers == vBuffer.pVertexBuffers)
    {
        vBuffer.pReadVertexBuffers = vb->pReadVertexBuffers;
        break;
    }
}

mesh.vertexBuffer = vBuffer;
mesh.hash = HashPointer(vBuffer.pVertexBuffers);
```

FIGURE 36. MATCHING VERTEX BUFFER IN DRAWINDEXED

Export model

Exporting the cached models is similar to the textures. We can choose to let the user determine when he wants to export them, or in our case we simply call the export function right after our snapshot finishes caching models.

```
//Before drawing overlay because input is checked in there
if (s_bSnapshotFrameActive)
{
    //Try matching models for some time accross frames, otherwise it's possible you get nothing.
    auto currTime = std::chrono::high_resolution_clock::now();
    auto timeSinceSnapStart = fmod(
        chrono::duration_cast<chrono::nanoseconds>(currTime - m_SnapshotStartTime)
        .count() * 1e-6f, 16777216.0f);
    if (timeSinceSnapStart > m_SnapshotTimeout)
    {
        ExportMeshes();
        s_bSnapshotFrameActive = false;
    }
}
```

FIGURE 37. EXPORT MODELS AT END OF SNAPSHOT IN RUNTIME

Thread-Pool

To improve performance we can use the same thread-pool we used for the exporting textures to queue our mesh exports. The model export probably has a much bigger impact on performance than the textures because a lot more data has to be reverse engineered and written to a standard text file rather than a binary memory dump.

```
void runtime::ExportMeshes()
{
    Logger::LogInfo(L"runtime::ExportMeshes() start");

    CheckAndCreateExportFolder("Meshes");

    int i = 0;
    for (auto res : s_MeshResources)
    {
        m_ThreadPool.enqueue([&](int id, Resource * pResource)
        {
            Logger::LogFormat(Info, L"Start ThreadPool Task ExportMesh %u", id);
            ExportMesh(pResource, id);
            Logger::LogFormat(Info, L"End ThreadPool Task ExportMesh %u", id);
        }, ++i, res);
    }

}
```

FIGURE 38. QUEUE MODEL EXPORTS IN THREAD-POOL

ExportMesh

To export a model we first have to break down all the cached data into buffers we can easily use to write to a usable file format. We pass through the topology which we need during the writing to determine which indices go with which face. We also pass an id which is just so we can write this in the file as metadata.

Next we try to deconstruct the binary data in the Index buffer. As explained in the research, the index buffer can use two data types. Either a 16 bit or 32 bit unsigned integer. Then we deconstruct the vertex buffer and finally write all the reverse engineered data to a file using a file format of our choice. Here we use the WaveFront .OBJ file format as it is pretty simple and easy to check. It is also supported by a lot of 3D tools.

```
void d3d11_runtime::ExportMesh(Resource* res, int i)
{
    Logger::LogFormat(Info, L"d3d11_runtime::ExportMesh(Resource* res, int i): start %u", i);

    auto mr = res->impl->as<d3d11_Mesh>();

    stringstream ss;
    ss.str();
    ss << "Exports\\Meshes\\mesh_" << i << ".obj";
    auto path = runtime::s_target_executable_path.parent_path() / ss.str().c_str();


    Mesh mesh;
    mesh.id = i;
    mesh.topology = mr->topology;

    //Index Buffer
    if (mr->indexBuffer.pReadIndexBuffer && mr->indexBuffer.pIndexBuffer)
    {
        D3D11_BUFFER_DESC indDesc;
        mr->indexBuffer.pIndexBuffer->GetDesc(&indDesc);
        Logger::LogFormat(Info, L"d3d11_runtime::SetIndices() mesh_%u", i);
        if (mr->indexBuffer.Format == DXGI_FORMAT_R16_UINT)
            SetIndices<USHORT>(mesh, indDesc, mr->indexBuffer.pReadIndexBuffer);
        else
            SetIndices<UINT>(mesh, indDesc, mr->indexBuffer.pReadIndexBuffer);
    }


    //Vertex Buffer
    Logger::LogFormat(Info, L"d3d11_runtime::SetVertices() mesh_%u", i);
    SetVertices(mesh, mr->vertexBuffer, mr->inputLayout);

    WaveFrontOBJWriter::Write(mesh, path.wstring().c_str());
}
```

FIGURE 39. 'EXPORTMESH' FUNCTION

Deconstructing Index Buffer

Going by our previous knowledge of accessing data from a staging buffer (staging textures), reading the index buffer is pretty straightforward. We first map the buffer resource to a mapped resource with read access. Then we use the calculated index count to loop through the raw data using the correct size for each value (16/32 bit unsigned integer) and fill our list. Finally release the memory lock with unmap and that's it.

To determine the index count, we simply take the size of the whole buffer, which we get from the buffer description. Then we divide this by the size of the data type used by the buffer, which determined based on the format we cached during the draw call with the 'IAGetIndexBuffer' function.

```
template<class T>
void d3d11_runtime::SetIndices(Mesh & mesh, const D3D11_BUFFER_DESC & desc, ID3D11Buffer * pReadBuffer) const
{
    auto indexCount = desc.ByteWidth / sizeof(T);

    Logger::LogFormat(Info, L"d3d11_runtime::SetIndices(): bytewidth %u | indexCount %u ", desc.ByteWidth, indexCount);

    D3D11_MAPPED_SUBRESOURCE mappedResource;
    auto hr = _immediate_context->Map(pReadBuffer, 0, D3D11_MAP_READ, 0, &mappedResource);
    if (FAILED(hr))
    {
        Logger::LogHResult(hr, L"Could not map Index buffer.");
        return;
    }
    //else continue
    for (auto i = 0; i < indexCount; ++i)
    {
        auto val = static_cast<T*>(mappedResource.pData)[i];
        mesh.indices.push_back(val);
    }
    _immediate_context->Unmap(pReadBuffer, 0);
}
```

FIGURE 40. 'SETINDICES' FUNCTION

Deconstructing Vertex Buffer[3]

To be able to convert a binary vertex buffer back to a readable format we have to know how each vertex in the buffer is built up. In DirectX 11 this layout is described in the Input Layout. With the use of semantic names, indices, format types and offsets the pipeline determines how to interpret the data. Since we cached all this data and matched it to the current model to export we can do the same. We use the Input Layout to determine which semantics are used and where in the vertex its data starts and how big it is. For the sake of simplifying the code a little bit we didn't use the format of each description in the input layout but used the most common data types used.

Formats used:

POSITION	XMFLOAT3
NORMAL	XMFLOAT3
TEXCOORD	XMFLOAT2
COLOR	XMFLOAT4

[3] Full code for 'SetVertices' in Appendix B

The input layout is an array of input layout descriptions. Each description determines the parameters for a semantic. These can be in any order so we have to loop through them and compare the types. During the Input Layout caching, we already converted the string names to enum values so we can more easily compare them here.

```
auto ieDesc = inputLayoutData.pInputElementDescs;

int positionOffset = -1, normalOffset = -1, texcoordOffset = -1, colorOffset = -1;

for (auto l = 0; l < inputLayoutData.NumElements; ++l)
{
    if (ieDesc[l].SemanticIndex == 0)
        switch (ieDesc[l].SemanticType)
        {
        case POSITION:
        {
            positionOffset = ieDesc[l].AlignedByteOffset;
            break;
        }
        case NORMAL:
        {
            normalOffset = ieDesc[l].AlignedByteOffset;
            break;
        }
        case TEXCOORD:
        {
            texcoordOffset = ieDesc[l].AlignedByteOffset;
            break;
        }
        case COLOR:
        {
            colorOffset = ieDesc[l].AlignedByteOffset;
            break;
        }
        case NONE:
        default: break;
        }
}
```

FIGURE 41. DETERMINE VERTEX OFFSETS FROM INPUT LAYOUT DESCRIPTIONS

Finally we can start converting the binary data and extracting the positions, normals, texcoords and colors from it. First we calculate the vertex count using the buffer's size divided by the stride (size in bytes) of each vertex. This stride was cached during the draw call using the 'IAGetVertexBuffers' function.

Then we simply map the readable vertex buffer to a readable mapped resource and use a binary reader to move through it and extract the data we want for each vertex. And finally remember to release the memory using the unmap function.

```
auto vertexCount = desc.ByteWidth / data.strides;

BinaryReader br;

D3D11_MAPPED_SUBRESOURCE mappedResource;
auto hr = _immediate_context->Map(data.pReadVertexBuffers, 0, D3D11_MAP_READ, 0, &mappedResource);
if (FAILED(hr)) { ... }
//else continue
auto pDataStart = static_cast<char*>(mappedResource.pData);
br.Open(pDataStart, desc.ByteWidth);

for (auto i = 0; i < vertexCount; ++i)
{
    auto vertexOffset = i * data.strides;

    // Position
    if (positionOffset >= 0)
    {
        br.SetBufferPosition(vertexOffset + positionOffset);
        mesh.positions.push_back(br.Read<XMFLOAT3>());
    }
    // Normal
    if (normalOffset >= 0)
    {
        br.SetBufferPosition(vertexOffset + normalOffset);
        mesh.normals.push_back(br.Read<XMFLOAT3>());
    }
    // TexCoord
    if (texcoordOffset >= 0)
    {
        br.SetBufferPosition(vertexOffset + texcoordOffset);
        mesh.texcoords.push_back(br.Read<XMFLOAT2>());
    }
    // Color
    if (colorOffset >= 0)
    {
        br.SetBufferPosition(vertexOffset + colorOffset);
        mesh.colors.push_back(br.Read<XMFLOAT4>());
    }

}
_immediate_context->Unmap(data.pReadVertexBuffers, 0);
```

FIGURE 42. EXTRACT VERTEX DATA WITH BINARY READER

Write to File[4]

For writing to a file we will be using the WaveFront .OBJ file format. This is a pretty simple straightforward format that is supported by a lot of 3D tools. This function isn't anything special, it is simply writing the data we extracted to a text file.

The only thing we do have to do here is to determine which indices are used for which face. This can be determined from the primitive topology and is done in the last part for writing the indices. The full code of this function is in Appendix C.

```
void WaveFrontOBJWriter::Write(const Mesh& mesh, wstring filename)
{
    Logger::LogFormat(Info, L"WaveFrontOBJWriter::Write() %s", filename.c_str());
    if (mesh.positions.size() <= 0) return;

    wofstream os;
    os.open(filename, std::ios::out);
    if (!os.is_open())
        return;

    // Name
    os << L"#Object name\n";
    os << L"o mesh_" << mesh.id << L"\n\n";

    // Vertices
    Logger::LogFormat(Info, L"WaveFrontOBJWriter::Write() %s | Vertices", filename.c_str());
    os << L"#Vertices\n";
    //--positions
    os << L"#--Positions\n";
    for (auto p : mesh.positions)
        os << L"v " << p.x << L" " << p.y << L" " << p.z << L"\n";

    os << L"\n";
    //--normals
    Logger::LogFormat(Info, L"WaveFrontOBJWriter::Write() %s | Normals", filename.c_str());
    os << L"#--Normals\n";
    for (auto n : mesh.normals)
        os << L"vn " << n.x << L" " << n.y << L" " << n.z << L"\n";

    os << L"\n";
    //--texcoords
    Logger::LogFormat(Info, L"WaveFrontOBJWriter::Write() %s | TexCoords", filename.c_str());
    os << L"#--TexCoords\n";
    for (auto t : mesh.texcoords)
        os << L"vt " << t.x << L" " << t.y << L"\n";

    // Indices
    bool bIndicesWritten = true;

    Logger::LogFormat(Info, L"WaveFrontOBJWriter::Write() %s | Indices", filename.c_str());
    os << L"\n";
    os << L"#Indices\n";
    if (mesh.indices.size() == 0) { ... }
    else { ... }

    os.close();
}
```

FIGURE 43. 'WAVEFRONTOBJWRITER::WRITE' FUNCTION

[4] Full 'WaveFrontOBJWriter::Write' function in Appendix C

4. REPLACING TEXTURES

Caching Shader Resource Views

As explained in the research about replacing textures, for each texture used there has to be a Shader Resource View (SRV) created. During the input assembler stage of the pipeline the required shader resources are set and at this point we can swap it out with our own SRV using our custom texture.

To cache the SRV's we simply use the hooked 'ID3D11Device::CreateShaderResourceView' function, check if the SRV to be created is meant for a 2D texture and save the pointers. At this point we don't need the texture pointer but we might be able to use it at a later time to do something else with it.

```
HRESULT STDMETHODCALLTYPE D3D11Device::CreateShaderResourceView(ID3D11Resource *pResource,
    const D3D11_SHADER_RESOURCE_VIEW_DESC *pDesc, ID3D11ShaderResourceView **ppSRView)
{
    auto hr = _orig->CreateShaderResourceView(pResource, pDesc, ppSRView);

    if (SUCCEEDED(hr))
    {
        if (pDesc && (pDesc->ViewDimension == D3D11_SRV_DIMENSION_TEXTURE2D || pDesc->ViewDimension == D3D11_SRV_DIMENSION_TEXTURE2DARRAY))
        {
            d3d11::d3d11_Texture2D data;
            data.texture = pResource;
            data.srv = *ppSRView;
            data.hash = HashPointer(*ppSRView);

            static_cast<ID3D11Texture2D*>(pResource)->GetDesc(&data.desc);

            dxhooks::Resource* res = new dxhooks::Resource();
            res->impl = make_unique<d3d11::d3d11_Texture2D>(data);

            dxhooks::runtime::add_SRVResources(res);
        }
    }

    return hr;
}
```

FIGURE 44. CACHING SHADER RESOURCE VIEWS

Replacement texture

Before we can switch out the textures we need to create a replacement texture. For this proof of concept we will use a simple 1x1 texture filled with a green color. We create this texture during the initialization of our runtime environment.

```
static const uint32_t s_pixel = 0xFF00FF00;

D3D11_SUBRESOURCE_DATA initData = { &s_pixel, sizeof(uint32_t), 0 };

D3D11_TEXTURE2D_DESC texDesc = {};
texDesc.Width = texDesc.Height = texDesc.MipLevels = texDesc.ArraySize = 1;
texDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
texDesc.SampleDesc.Count = 1;
texDesc.Usage = D3D11_USAGE_IMMUTABLE;
texDesc.BindFlags = D3D11_BIND_SHADER_RESOURCE;

//Create texture
ID3D11Texture2D* pTex = nullptr;
auto hr = _device->CreateTexture2D(&texDesc, &initData, &pTex);
Logger::LogHResult(hr, L"d3d11_runtime::on_init(): CreateTexture2D");
```

FIGURE 45. CREATE REPLACEMENT TEXTURE

If we successfully created the texture we also create an SRV using this texture which we will use to do the switch.

```
if (SUCCEEDED(hr))
{
    //SRV description
    D3D11_SHADER_RESOURCE_VIEW_DESC srvDesc;
    ZeroMemory(&srvDesc, sizeof(D3D11_SHADER_RESOURCE_VIEW_DESC));
    srvDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
    srvDesc.ViewDimension = D3D11_SRV_DIMENSION_TEXTURE2D;
    srvDesc.Texture2D.MipLevels = 1;
    srvDesc.Texture2D.MostDetailedMip = 0;

    //Create srv
    ID3D11ShaderResourceView* pSRV = nullptr;
    hr = _device->CreateShaderResourceView(pTex, &srvDesc, &pSRV);
    Logger::LogHResult(hr, L"d3d11_runtime::on_init(): CreateShaderResourceView");
    if (SUCCEEDED(hr))
    {

        d3d11::d3d11_Texture2D data;
        data.texture = pTex;
        data.desc = texDesc;
        data.srv = pSRV;

        s_SRVResource = new dxhooks::Resource();
        s_SRVResource->impl = make_unique<d3d11::d3d11_Texture2D>(data);
    }
}
```

FIGURE 46. CREATE REPLACEMENT SHADER RESOURCE VIEW

Making the switch

As discussed before we will do the actual switch every frame during in the 'PSSetShaderResources' function from the device context. We compare the current selected pointer from our cache list with the list that's passed and overwrite it when we have a match.

```
void STDMETHODCALLTYPE D3D11DeviceContext::PSSetShaderResources(UINT StartSlot, UINT NumViews,
    ID3D11ShaderResourceView *const *ppShaderResourceViews)
{
    //Set requested SRVs
    _orig->PSSetShaderResources(StartSlot, NumViews, ppShaderResourceViews);

    //Check if we have a selected SRV resource and the passed SRV list isn't null
    auto selectedSRVResource = dxhooks::runtime::get_SelectedSRVResource();
    if (selectedSRVResource && *ppShaderResourceViews)
    {
        //Get the selected SRV and our placeholder Replacement resource (check if it exists)
        auto selectedSRV = selectedSRVResource->impl->as<d3d11::d3d11_Texture2D>()->srv;
        auto srvResource = dxhooks::runtime::GetSRVReplacement();
        if (srvResource)
        {
            auto replaceSRV = srvResource->impl->as<d3d11::d3d11_Texture2D>()->srv.get();
            ID3D11ShaderResourceView* const srvs[] = { replaceSRV }; //static array of SRV pointer(s)
            //Look through all the SRVs in the list and see if one matches our selected SRV
            for (UINT i = 0; i < NumViews; ++i)
            {
                auto srv = (*ppShaderResourceViews) + i;

                //If match found -> Overwrite the matched SRV with ours
                if (selectedSRV.get() == srv)
                {
                    _orig->PSSetShaderResources(i, 1, srvs);
                    return;
                }
            }
        }
    }
}
```

FIGURE 47. REPLACING TEXTURE (SRV) IN 'PSSETSHADERRESOURCES'

CONCLUSION

In this paper we researched the methods for code injection, more specifically DLL injection. We looked into a few different existing methods, such as simply replacing the DLL file and why it works. We then followed this up by looking into hooking APIs and to better understand how these are built up. We discussed the use of COM interfaces and how these can be exploited due to their requirements such as having to implement the IUnknown interface and the first 4 bytes always pointing to the virtual table which holds all the functions. We also discussed the basic principle behind the detour and trampoline method for redirecting the flow of code at runtime.

DLL injection and hooking is a powerful technique to exploit applications by redirecting and intercepting data. This exploit can be used for common things such as hacks, but also for analyzing and debugging applications.

After we learned about DLL injection and hooking APIs, we looked into how we could exploit these methods and use them to hook into and intercept data in a DirectX 11 application. We tried our hand at three different cases. First, exporting textures. Next we looked into exporting models, which took more effort debugging and reverse engineering the data, but essentially works the same way as exporting textures. Finally we looked into replacing textures at runtime, which turned out much easier than expected by simply making the switch during the pipeline stage setup.

The main struggles I faced during development of the DirectX 11 functionalities was the lack of deeper knowledge of DX11 functions and data structures. With all the new knowledge I gained from working on this most of it comes down to core principles of DirectX 11, such as memory management and correct descriptions for creating objects.

REFERENCES

[D3D11] Creating a dynamic texture? (2010). Retrieved from Gamedev.net: https://www.gamedev.net/forums/topic/583161-d3d11-creating-a-dynamic-texture/

Christoph. (2008, December 16). *Fast String Hashing Algorithm with low collision rates with 32 bit integer*. Retrieved from Stackoverflow: https://stackoverflow.com/a/372996

Component Object Model. (n.d.). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Component_Object_Model

Crosire. (2017). *ReShade*. Retrieved from Github: https://github.com/crosire/reshade

Crosire. (n.d.). *ReShade*. Retrieved from ReShade: https://reshade.me/

D3DX11SaveTextureToFile function. (n.d.). Retrieved from Microsoft Developer Network (MSDN): https://msdn.microsoft.com/en-us/library/windows/desktop/ff476298(v=vs.85).aspx

Direct3D 11 Reference. (n.d.). Retrieved from Microsoft Developer Network (MSDN): https://msdn.microsoft.com/en-us/library/windows/desktop/ff476079(v=vs.85).aspx

DisableThreadLibraryCalls function. (n.d.). Retrieved from Microsoft Developer Network (MSDN): https://msdn.microsoft.com/en-us/library/windows/desktop/ms682579(v=vs.85).aspx

DLLs in Visual C++. (n.d.). Retrieved from Microsoft Developer Network (MSDN): https://msdn.microsoft.com/en-us/library/1ez7dh12.aspx

Drop. (2013, July 3). *How to read vertices from vertex buffer in Direct3d11*. Retrieved from Stackoverflow: https://stackoverflow.com/a/17447416

DXGI Reference. (n.d.). Retrieved from Microsoft Developer Network (MSDN): https://msdn.microsoft.com/en-us/library/windows/desktop/bb205169(v=vs.85).aspx

Exporting from a DLL Using __declspec(dllexport). (n.d.). Retrieved from Microsoft Developer Network (MSDN): https://msdn.microsoft.com/en-us/library/a90k134d.aspx

Galen C. Hunt, & Michael L. Scott. (1999, May 1). *Intercepting and Instrumenting COM Applications [PDF].* Retrieved from Microsoft: https://www.microsoft.com/en-us/research/publication/intercepting-and-instrumenting-com-applications/

Get Raw Data from D3D11Texture2D C++. (2017). Retrieved from Gamedev.net: https://www.gamedev.net/forums/topic/688535-get-raw-data-from-d3d11texture2d-c/

Glatt, J. (2006, March 29). *COM in plain C*. Retrieved from Codeproject: https://www.codeproject.com/Articles/13601/COM-in-plain-C

Graphics pipeline. (n.d.). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Graphics_pipeline

Graphics Pipeline. (n.d.). Retrieved from Microsoft Developer Network (MSDN): https://msdn.microsoft.com/en-us/library/windows/desktop/ff476882(v=vs.85).aspx

How to use UpdateSubresource and Map/Unmap? (2013). Retrieved from Gamedev Stackexchange: https://gamedev.stackexchange.com/questions/60668/how-to-use-updatesubresource-and-map-unmap

IUnknown. (n.d.). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/IUnknown

IUnknown interface. (n.d.). Retrieved from Microsoft Developer Network (MSDN): https://msdn.microsoft.com/en-us/library/windows/desktop/ms680509(v=vs.85).aspx

Ivanov, I. (2002, December 3). *API hooking revealed*. Retrieved from Codeproject: https://www.codeproject.com/Articles/2082/API-hooking-revealed

Kageyu, T. (2015, March 17). *MinHook - The Minimalistic x86/x64 API Hooking Library*. Retrieved from Code Project: https://www.codeproject.com/Articles/44326/MinHook-The-Minimalistic-x-x-API-Hooking-Libra

Kaisar. (2008, July 6). *C++: Accessing the virtual table directly*. Retrieved from Kaisar's blog (Blogspot): http://kaisar-haque.blogspot.be/2008/07/c-accessing-virtual-table.html

Malik, A. (n.d.). *DLL Injection and Hooking*. Retrieved from Security Xploded: http://securityxploded.com/dll-injection-and-hooking.php

McDonald, J. (2012, March 5-9). *Don't Throw it all Away: Efficient Buffer Management [PDF].* Retrieved from NVIDIA Developer: https://developer.nvidia.com/sites/default/files/akamai/gamedev/files/gdc12/Efficient_Buffer_Management_McDonald.pdf

Microsoft/DirectXTex: DirectXTex texture processing library. (2017). Retrieved from Github: https://github.com/Microsoft/DirectXTex

Microsoft/DirectXTK: The DirectX Tool Kit (aka DirectXTK) is a collection of helper classes for writing DirectX 11.x code in C++. (2017). Retrieved from Github: https://github.com/Microsoft/DirectXTK

Name mangling. (n.d.). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Name_mangling

Patil, S. (2014, March 20). *Code Injection and API Hooking Techniques*. Retrieved from Security Xploded Blog: http://nagareshwar.securityxploded.com/2014/03/20/code-injection-and-api-hooking-techniques/

Podobriy, S. (2014, December 28). *Windows API Hooking Tutorial (Example with DLL Injection)*. Retrieved from Apriorit: https://www.apriorit.com/dev-blog/160-apihooks

Progsch, J., & Zeman, V. (2014, September 26). *ThreadPool*. Retrieved from Github: https://github.com/progschj/ThreadPool

Reading one pixel from texture to CPU in DX11. (2011). Retrieved from Gamedev.net: https://www.gamedev.net/forums/topic/594722-reading-one-pixel-from-texture-to-cpu-in-dx11/

Robertson, C., Blome, M., Hogenson, G., & Cai, S. (2016, November 4). *Module-Definition (.Def) Files*. Retrieved from Microsoft Docs: https://docs.microsoft.com/nl-nl/cpp/build/reference/module-definition-dot-def-files

Schaub, J. (2009, February 7). *c++ - How can i avoid name mangling?* Retrieved from Stackoverflow: https://stackoverflow.com/a/524636

Search Path Used by Windows to Locate a DLL. (n.d.). Retrieved from Microsoft Developer Network (MSDN): https://msdn.microsoft.com/en-us/library/7d83bc18.aspx

Shamray, V. (2011, February 2). *Intercepting Calls to COM Interfaces*. Retrieved from Codeproject: https://www.codeproject.com/Articles/153096/Intercepting-Calls-to-COM-Interfaces]

Swapping addresses of pointers in C++. (2009). Retrieved from Stackoverflow: https://stackoverflow.com/questions/1826203/swapping-addresses-of-pointers-in-c

Swapping pointers. (2013). Retrieved from Stackoverflow: https://stackoverflow.com/questions/14180365/swapping-pointers

Virtual method table. (n.d.). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Virtual_method_table

Walbourn, C. (2017, January 13). *DirectX 11 - How to create a very simple 2D texture*. Retrieved from Stackoverflow: https://stackoverflow.com/a/41628521

What exactly are DLL files, and how do they work? (2008, September 23). Retrieved from Stackoverflow: https://stackoverflow.com/a/124550

What is a DLL? (n.d.). Retrieved from Microsoft Support: https://support.microsoft.com/en-us/help/815065/what-is-a-dll

APPENDICES

APPENDIX A

```
HRESULT D3D11Device::GetStagingTextureHash(d3d11::d3d11_Texture2D& data,
	const D3D11_TEXTURE2D_DESC* pDesc, ID3D11Texture2D** ppTexture2D) const
{
	ID3D11Texture2D* pStaging = nullptr;
	D3D11_TEXTURE2D_DESC description;
	// Get original desc data.
	(*ppTexture2D)->GetDesc(&description);

	// MSAA content must be resolved before being copied to a staging texture
	if (description.SampleDesc.Count > 1)
	{
		description.SampleDesc.Count = 1;
		description.SampleDesc.Quality = 0;

		ID3D11Texture2D* pTemp = nullptr;
		auto hr = _orig->CreateTexture2D(&description, nullptr, &pTemp);
		if (FAILED(hr))
		{
			SafeRelease(pTemp);
			Logger::LogHResult(hr, L"MultiSampling: Could not create temp
staging texture without MSAA.");
			return hr;
		}

		//Check MSAA format support;
		DXGI_FORMAT fmt = EnsureNotTypeless(description.Format);

		UINT support = 0;
		hr = _orig->CheckFormatSupport(fmt, &support);
		if (FAILED(hr))
		{
			SafeRelease(pTemp);
			Logger::LogHResult(hr, L"MultiSampling: Failed to check format
support.");
			return hr;
		}

		//Error: MSAA resolve operations are not supported, return
		if (!(support & D3D11_FORMAT_SUPPORT_MULTISAMPLE_RESOLVE))
		{
			SafeRelease(pTemp);
			Logger::LogHResult(E_FAIL, L"MultiSampling: MSAA resolve operations
are not supported. (For more info, see ID3D11DeviceContex::ResolveSubresource)");
			return E_FAIL;
		}
```

```
			//MSAA texture - ResolveSubresource;
			for (UINT item = 0; item < description.ArraySize; ++item)
			{
				for (UINT level = 0; level < description.MipLevels; ++level)
				{
					UINT index = D3D11CalcSubresource(level, item, 
description.MipLevels);
					_immediate_context->ResolveSubresource(pTemp, index, 
(*ppTexture2D), index, fmt);
				}
			}

			// The resource is to be mapable so that the CPU can read its contents.
			// Resources created with this flag cannot be set as either inputs or 
outputs to the pipeline
			// and must be created with staging usage (see D3D11_USAGE).
			description.CPUAccessFlags = D3D11_CPU_ACCESS_READ;
			// A resource that supports data transfer (copy) from the GPU to the CPU.
			description.Usage = D3D11_USAGE_STAGING;
			// Resource won't be bound to anything
			description.BindFlags = 0;
			description.MiscFlags &= D3D11_RESOURCE_MISC_TEXTURECUBE;

			hr = _orig->CreateTexture2D(&description, nullptr, &pStaging);
			if (FAILED(hr))
			{
				SafeRelease(pStaging);
				SafeRelease(pTemp);
				Logger::LogHResult(hr, L"MultiSampling: Could not create staging 
texture.");
				return hr;
			}

			// Copy resolved resource data to staging texture
			// (Copy the entire contents of the source resource to the destination 
resource using the GPU.)
			_immediate_context->CopyResource(pStaging, pTemp);
		}
		// Not an MSAA texture, but already a staging texture
		else if ((description.Usage == D3D11_USAGE_STAGING) && (description.CPUAccessFlags 
& D3D11_CPU_ACCESS_READ))
		{
			pStaging = (*ppTexture2D);
		}
```

```
	// Not MSAA or staging texture
	// Try to create a staging texture which can be read by the CPU (won't be usable
for GPU so clean up later).
	else
	{
		description.CPUAccessFlags = D3D11_CPU_ACCESS_READ;
		description.Usage = D3D11_USAGE_STAGING;
		description.BindFlags = 0;

		auto hr = _orig->CreateTexture2D(&description, nullptr, &pStaging);
		if (FAILED(hr))
		{
			SafeRelease(pStaging);
			Logger::LogHResult(hr, L"Could not create a staging texture.");
			return hr;
		}

		// Copy actual resource data to staging texture
		// (Copy the entire contents of the source resource to the destination
resource using the GPU.)
		_immediate_context->CopyResource(pStaging, *ppTexture2D);
	}

	//Something went wrong, return error
	if (!pStaging)
	{
		Logger::LogWarning(L"pStaging is nullptr");
		return E_FAIL;
	}
	//Try to map staging texture and get hash value
	else
	{
		D3D11_MAPPED_SUBRESOURCE mapped;
		// Lock memory
		auto hr = _immediate_context->Map(pStaging, 0, D3D11_MAP_READ, 0, &mapped);
		//Something went wrong, return error
		if (FAILED(hr))
		{
			SafeRelease(pStaging);
			Logger::LogHResult(hr, L"Could not map temp staging texture.");
			return E_FAIL;
		}

		// If mapped, hash the data for duplicate comparison
		data.hash = Hash(static_cast<char*>(mapped.pData));

		// Unlock memory
		_immediate_context->Unmap(pStaging, 0);
	}

	// Save in data encapsuling object
	data.texture = pStaging;
	data.desc = description;

	return S_OK;
}
```

APPENDIX B

```
void d3d11_runtime::SetVertices(Mesh& mesh, const d3d11_VertexBufferData & data, const
d3d11_InputLayoutData & inputLayoutData) const
	{
		if (!data.pVertexBuffers || !data.pReadVertexBuffers)
			return;

		D3D11_BUFFER_DESC desc;
		data.pVertexBuffers->GetDesc(&desc);

		auto ieDesc = inputLayoutData.pInputElementDescs;

		int positionOffset = -1, normalOffset = -1, texcoordOffset = -1;
		int colorOffset = -1, binormalOffset = -1, tangentOffset = -1;
		int blendIndicesOffset = -1, blendWeightsOffset = -1;

		for (auto l = 0; l < inputLayoutData.NumElements; ++l)
		{
			if (ieDesc[l].SemanticIndex == 0)
				switch (ieDesc[l].SemanticType)
				{
				case POSITION:
					positionOffset = ieDesc[l].AlignedByteOffset;
					break;
				case NORMAL:
					normalOffset = ieDesc[l].AlignedByteOffset;
					break;
				case TEXCOORD:
					texcoordOffset = ieDesc[l].AlignedByteOffset;
					break;
				case COLOR:
					colorOffset = ieDesc[l].AlignedByteOffset;
					break;
				case BINORMAL:
					binormalOffset = ieDesc[l].AlignedByteOffset;
					break;
				case TANGENT:
					tangentOffset = ieDesc[l].AlignedByteOffset;
					break;
				case BLENDINDICES:
					blendIndicesOffset = ieDesc[l].AlignedByteOffset;
					break;
				case BLENDWEIGHTS:
					blendWeightsOffset = ieDesc[l].AlignedByteOffset;
					break;
				case NONE:
				default: break;
				}
		}

		auto vertexCount = desc.ByteWidth / data.strides;
```

```
		Logger::LogFormat(Info, L"d3d11_runtime::SetVertices(): bytewidth %u |
vertexCount %u ", desc.ByteWidth, vertexCount);

		BinaryReader br;

		D3D11_MAPPED_SUBRESOURCE mappedResource;
		auto hr = _immediate_context->Map(data.pReadVertexBuffers, 0,
D3D11_MAP_READ, 0, &mappedResource);
		if (FAILED(hr))
		{
			Logger::LogHResult(hr, L"Could not map Vertex buffer.");
			return;
		}
		//else continue
		auto pDataStart = static_cast<char*>(mappedResource.pData);
		br.Open(pDataStart, desc.ByteWidth);

		for (auto i = 0; i < vertexCount; ++i)
		{
			auto vertexOffset = i * data.strides;

			// Position
			if (positionOffset >= 0)
			{
				br.SetBufferPosition(vertexOffset + positionOffset);
				mesh.positions.push_back(br.Read<XMFLOAT3>());
			}

			// Normal
			if (normalOffset >= 0)
			{
				br.SetBufferPosition(vertexOffset + normalOffset);
				mesh.normals.push_back(br.Read<XMFLOAT3>());
			}

			// TexCoord
			if (texcoordOffset >= 0)
			{
				br.SetBufferPosition(vertexOffset + texcoordOffset);
				mesh.texcoords.push_back(br.Read<XMFLOAT2>());
			}

			// Color
			if (colorOffset >= 0)
			{
				br.SetBufferPosition(vertexOffset + colorOffset);
				mesh.colors.push_back(br.Read<XMFLOAT4>());
			}

		}
		_immediate_context->Unmap(data.pReadVertexBuffers, 0);
	}
```

APPENDIX C

```
void WaveFrontOBJWriter::Write(const Mesh& mesh, wstring filename)
{
	Logger::LogFormat(Info, L"WaveFrontOBJWriter::Write() %s", filename.c_str());
	if (mesh.positions.size() <= 0) return;

	wofstream os;
	os.open(filename, std::ios::out);
	if (!os.is_open())
		return;

	// Name
	os << L"#Object name\n";
	os << L"o mesh_" << mesh.id << L"\n\n";

	// Vertices
	Logger::LogFormat(Info, L"WaveFrontOBJWriter::Write() %s | Vertices",
filename.c_str());
	os << L"#Vertices\n";
	//--positions
	os << L"#--Positions\n";
	for (auto p : mesh.positions)
		os << L"v " << p.x << L" " << p.y << L" " << p.z << L"\n";

	os << L"\n";
	//--normals
	Logger::LogFormat(Info, L"WaveFrontOBJWriter::Write() %s | Normals",
filename.c_str());
	os << L"#--Normals\n";
	for (auto n : mesh.normals)
		os << L"vn " << n.x << L" " << n.y << L" " << n.z << L"\n";

	os << L"\n";
	//--texcoords
	Logger::LogFormat(Info, L"WaveFrontOBJWriter::Write() %s | TexCoords",
filename.c_str());
	os << L"#--TexCoords\n";
	for (auto t : mesh.texcoords)
		os << L"vt " << t.x << L" " << t.y << L"\n";

	// Indices
	bool bIndicesWritten = true;

	Logger::LogFormat(Info, L"WaveFrontOBJWriter::Write() %s | Indices",
filename.c_str());
	os << L"\n";
	os << L"#Indices\n";
	if (mesh.indices.size() == 0)
	{
		for (auto i = 0; i < mesh.positions.size() - 2; i += 3)
		{
			os << L"f " << i + 1 << L" " << i + 2 << L" " << i + 3 << L"\n";
		}
	}
```

```
        else
        {
                switch (mesh.topology)
                {
                case D3D_PRIMITIVE_TOPOLOGY_UNDEFINED:
                case D3D_PRIMITIVE_TOPOLOGY_POINTLIST:
                {
                        os << L"p";
                        for (auto i : mesh.indices)
                        {
                                os << L" " << i + 1;
                        }
                        os << L"\n";
                }
                break;
                case D3D_PRIMITIVE_TOPOLOGY_LINELIST:
                case D3D_PRIMITIVE_TOPOLOGY_LINESTRIP:
                case D3D_PRIMITIVE_TOPOLOGY_LINELIST_ADJ:
                case D3D_PRIMITIVE_TOPOLOGY_LINESTRIP_ADJ:
                {
                        os << L"l";
                        for (auto i : mesh.indices)
                        {
                                os << L" " << i + 1;
                        }
                        os << L"\n";
                }
                break;
                case D3D_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP:
                case D3D_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP_ADJ:
                {
                        if (mesh.indices.size() >= 3)
                                for (auto i = 0; i < mesh.indices.size() - 2; ++i)
                                {
                                        os << L"f " << mesh.indices[i] + 1 << L" " <<
mesh.indices[i + 1] + 1 << L" " << mesh.indices[i + 2] + 1 << L"\n";
                                }
                }
                break;
                default: bIndicesWritten = false; break;
                }

        if (!bIndicesWritten)
        {
        auto indicesPerFace = 1;
        switch (mesh.topology)
        {
        case D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST:
        case D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST_ADJ:
        case D3D_PRIMITIVE_TOPOLOGY_3_CONTROL_POINT_PATCHLIST:
                indicesPerFace = 3; break;
        case D3D_PRIMITIVE_TOPOLOGY_1_CONTROL_POINT_PATCHLIST: indicesPerFace = 1; break;
        case D3D_PRIMITIVE_TOPOLOGY_2_CONTROL_POINT_PATCHLIST: indicesPerFace = 2; break;
        case D3D_PRIMITIVE_TOPOLOGY_4_CONTROL_POINT_PATCHLIST: indicesPerFace = 4; break;
        case D3D_PRIMITIVE_TOPOLOGY_5_CONTROL_POINT_PATCHLIST: indicesPerFace = 5; break;
        case D3D_PRIMITIVE_TOPOLOGY_6_CONTROL_POINT_PATCHLIST: indicesPerFace = 6; break;
        case D3D_PRIMITIVE_TOPOLOGY_7_CONTROL_POINT_PATCHLIST: indicesPerFace = 7; break;
```

```
        case D3D_PRIMITIVE_TOPOLOGY_8_CONTROL_POINT_PATCHLIST: indicesPerFace = 8; break
        case D3D_PRIMITIVE_TOPOLOGY_9_CONTROL_POINT_PATCHLIST: indicesPerFace = 9; break;
        case D3D_PRIMITIVE_TOPOLOGY_10_CONTROL_POINT_PATCHLIST:indicesPerFace = 10; break;
        case D3D_PRIMITIVE_TOPOLOGY_11_CONTROL_POINT_PATCHLIST:indicesPerFace = 11; break;
        case D3D_PRIMITIVE_TOPOLOGY_12_CONTROL_POINT_PATCHLIST:indicesPerFace = 12; break;
        case D3D_PRIMITIVE_TOPOLOGY_13_CONTROL_POINT_PATCHLIST:indicesPerFace = 13; break;
        case D3D_PRIMITIVE_TOPOLOGY_14_CONTROL_POINT_PATCHLIST:indicesPerFace = 14; break;
        case D3D_PRIMITIVE_TOPOLOGY_15_CONTROL_POINT_PATCHLIST:indicesPerFace = 15; break;
        case D3D_PRIMITIVE_TOPOLOGY_16_CONTROL_POINT_PATCHLIST:indicesPerFace = 16; break;
        case D3D_PRIMITIVE_TOPOLOGY_17_CONTROL_POINT_PATCHLIST:indicesPerFace = 17; break;
        case D3D_PRIMITIVE_TOPOLOGY_18_CONTROL_POINT_PATCHLIST:indicesPerFace = 18; break;
        case D3D_PRIMITIVE_TOPOLOGY_19_CONTROL_POINT_PATCHLIST:indicesPerFace = 19; break;
        case D3D_PRIMITIVE_TOPOLOGY_20_CONTROL_POINT_PATCHLIST:indicesPerFace = 20; break;
        case D3D_PRIMITIVE_TOPOLOGY_21_CONTROL_POINT_PATCHLIST:indicesPerFace = 21; break;
        case D3D_PRIMITIVE_TOPOLOGY_22_CONTROL_POINT_PATCHLIST:indicesPerFace = 22; break;
        case D3D_PRIMITIVE_TOPOLOGY_23_CONTROL_POINT_PATCHLIST:indicesPerFace = 23; break;
        case D3D_PRIMITIVE_TOPOLOGY_24_CONTROL_POINT_PATCHLIST:indicesPerFace = 24; break;
        case D3D_PRIMITIVE_TOPOLOGY_25_CONTROL_POINT_PATCHLIST:indicesPerFace = 25; break;
        case D3D_PRIMITIVE_TOPOLOGY_26_CONTROL_POINT_PATCHLIST:indicesPerFace = 26; break;
        case D3D_PRIMITIVE_TOPOLOGY_27_CONTROL_POINT_PATCHLIST:indicesPerFace = 27; break;
        case D3D_PRIMITIVE_TOPOLOGY_28_CONTROL_POINT_PATCHLIST:indicesPerFace = 28; break;
        case D3D_PRIMITIVE_TOPOLOGY_29_CONTROL_POINT_PATCHLIST:indicesPerFace = 29; break;
        case D3D_PRIMITIVE_TOPOLOGY_30_CONTROL_POINT_PATCHLIST:indicesPerFace = 30; break;
        case D3D_PRIMITIVE_TOPOLOGY_31_CONTROL_POINT_PATCHLIST:indicesPerFace = 31; break;
        case D3D_PRIMITIVE_TOPOLOGY_32_CONTROL_POINT_PATCHLIST:indicesPerFace = 32; break;
        default:  break;
        }

        if (mesh.indices.size() >= indicesPerFace)
                for (auto i = 0; i < mesh.indices.size() - (indicesPerFace - 1); i +=
indicesPerFace)
                {
                        os << L"f ";
                        for (auto j = 0; j < indicesPerFace; ++j)
                                os << mesh.indices[i + j] + 1 << L" ";

                        os << L"\n";
                        }
                }
        }

        os.close();
}
```