FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Ovlivňování vykreslování 3D scén v reálném čase pro DirectX 11
Student:	Jakub Čudka
Vedoucí:	Ing. Filip Štěpánek
Studijní program:	Informatika
Studijní obor:	Bezpečnost a informační technologie
Katedra:	Katedra počítačových systémů
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

DirectX 11 je kolekce aplikačních rozhraní (API) pro zpracovávání zvuku, obrazu, videa a dalších multimediálních informací. Cílem práce je vytvořit univerzální postup pro útok pomocí injektování DLL, který ovlivní vykreslení 3D scény v reálném čase bez přístupu ke zdrojovým kódům aplikace. Útok bude demonstrován na scéně vytvořené studentem a bude možné ho použít na reálnou aplikaci (scénu) využívající DirectX 11 API.

Práci rozdělte na následující kroky:
- Seznamte se s technologií DirectX 11 (vykreslování 3D scén).
- Analyzujte stávající útoky pro DirectX 11 využívající tzv. "injektování DLL".
- Navrhněte a realizujte jednoduchou scénu, která bude sloužit pro demonstraci útoku.
- Navrhněte a realizujte útok na vykreslování scény pomocí "injektování DLL". Škodlivý kód bude obsahovat ovládací prvky útočníka (změna parametrů objektů ve scéně, zobrazení skrytých objektů, přidávání či odebírání objektů, apod.).
- Výsledek porovnejte se stávajícími útoky a vyhodnoťte výsledky.

Seznam odborné literatury

Dodá vedoucí práce.

prof. Ing. Róbert Lórencz, CSc.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 2. února 2018

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra počítačových systémů

Bakalářská práce

Ovlivňování vykreslování 3D scén v reálném čase pro DirectX 11

Jakub Čudka

Vedoucí práce: Ing. Filip Štěpánek

15. května 2018

Poděkování

Děkuji svému vedoucímu práce Filipu Štěpánkovi za všechny rady a čas, který mi při práci poskytl. Dále bych chtěl poděkovat všem, kteří mi z práce pomáhali odstranit chyby jak gramatické, tak faktografické.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 15. května 2018 .....

Odkaz na tuto práci

Čudka, Jakub. *Ovlivňování vykreslování 3D scén v reálném čase pro DirectX 11*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Práce se zaměřuje na útoky na grafické programy, které využívají knihovny DirectX 11 pod operačním systémem Windows 10. Tyto útoky mají jako hlavní cíl počítačové hry, ale dají se využít kdekoliv, kde se tato knihovna používá. Cílem práce je analyzovat fungování zmíněných knihoven a na základě této analýzy vytvořit útok, který bude DirectX 11 zneužívat pro změnu vykreslované scény bez zásahu do zdrojového kódu napadené aplikace. K řešení jsem použil programovací jazyk C++, který je nejvhodnější pro práci s DirectX a techniku zvanou DLL injection. Výsledek práce je dynamická knihovna, která umožňuje manipulovat se scénou v testovacím programu.

Klíčová slova DLL injection, DirectX 11, DirectX 9, Ovliňování vykreslování grafiky, VF Table Hooking

Abstract

This Bachelor thesis focuses on attacks on graphics programs that use DirectX 11. These libraries are used under operating system Windows 10 in this case. The main aim of these attacks are computer games. The first goal of this work is to analyse the functioning of DirectX 11 libraries. The second one is to design an attack on mine own graphical application. It changes the behavior of the application during it's runtime. Result of this work is dynamic C++ library that is able to modify 3D graphical scene.

Keywords DLL injection, DirectX 11, DirectX 9, VF Table Hooking, Tampering with graphical scene

Obsah

Seznam zdrojových kódů **xvii**

Úvod **1**
Cíle práce 2

1 Analýza **3**
1.1 DirectX 3
1.2 Základní objekty Direct3D 4
1.3 Graphics Pipeline Direct3D 7
1.4 DXGI 13
1.5 Způsoby ovlivňování 3D scény pomocí DirectX 15
1.6 Rozdíl mezi DirectX 9 a DirectX 11 21
1.7 Vkládání škodlivého kódu do původní aplikace 23

2 Návrh řešení **29**
2.1 Příprava vlastní grafické aplikace 29
2.2 Příprava útoku na grafickou aplikaci 31
2.3 Struktura útoku 32
2.4 Nástroje použité pro řešení 33

3 Realizace **35**
3.1 Vlastní Aplikace 35
3.2 Šablona pro ovlivnění scény 42
3.3 Vlastní úprava tabulek virtuálních funkcí 43
3.4 Útoky na vlastní aplikaci 44

4 Testování **53**
4.1 Rotující kostky 53
4.2 Stone Giant 55

Závěr **63**

Literatura **65**

A Seznam použitých zkratek **69**

B Obsah přiloženého CD **71**

C Manuál **73**
C.1 Pracovní prostředí Windows 10 73
C.2 Příprava VisualStudia a instalace grafických aplikací 73
C.3 Modul DirectX 74
C.4 Vložení dynamické knihovny s úpravami do aplikace 74
C.5 Ovládání grafických aplikací a vložených úprav 75

Seznam obrázků

1.1 Na obrázku je zobrazen příklad, jak může vypadat topologie PointList po zpracování. 5
1.2 Zde je znázorněn objekt po zpracování v topologii LineList. Jednotlivé vrcholy jsou spojeny do oddělených čar. 5
1.3 Topologie LineStrip se moc neliší od topologie LineList 1.2. Rozdílem je že se spojí postupně všechny vrcholy. 6
1.4 Obrázek vyobrazuje topologii TriangleList. v této topologii se vrcholy zpracovávají do samostatných trojúhelníků. 7
1.5 Topologie TriangleStrip je podobná s topologií TriangleList 1.4. Rovněž se z vrcholů vykreslují trojúhelníky, ale tentokrát do souvislé plochy. 7
1.6 Schéma zobrazuje Graphics Pipeline a jeho fáze, které využívá pro zpracování 3D scény v pořadí v jakém se volají. 9
1.7 Schéma umístění vrstvy DXGI v architektuře programu a jeho komunikaci s Hardwarem 14
1.8 Na obrázku je znázorněno, jakým způsobem se prohazují grafické zásobníky ve SwapChain 15
1.9 Na obrázku je ukázka útoku WallHack na reálné aplikaci. Je na něm znázorněno, že se postavy a jiné objekty zobrazí i přesto, že mají být skryté za překážkami. 18
1.10 Na obrázku jsou znázorněny tři způsoby ovlivnění 3D Scény. Růžové plus je ukázka Custom Crosshair. Modrý rámeček a bílý text je vyobrazení možného ESP Hack. Červená tečka v každém rámečku je znázornění útoku AimBot. Obrázek je pořízen z reálné aplikace. 19
1.11 Na obrázku jsou vyobrazeny možnosti pro útok Custom Crosshair. Jedná se pouze o malý výběr. Autor útoku může navrhnout svůj vlastní, který bude odpovídat jeho potřebám. 20
1.12 Na obrázku je zobrazen útok DirectX Overlay. V této implementaci útoku je menu pro ovládání ostatních útoků pro COD 4. 21

1.13 Na obrázku je vyobrazen postup vložení dynamické knihovny do aplikace třetí strany bez přístupu ke zdrojovým kódům. 26

2.1 Na tomto schématu je znázorněna architektura aplikace a jak spolu jednotlivé komponenty souvisí. Červeně jsou označeny ty, jejichž kód jsem měnil, aby vyhovoval potřebám této práce. 30
2.2 Na tomto schématu je znázorněno rozložení útoků vůči samotné aplikaci. Jednotlivé úpravy jsou rozdělené podle funkcí, které zneužívají. 32

3.1 Na obrázku je zobrazena vlastní scéna. Je navržena pro prezentování útoků ze zadání. Objekty jsou rozděleny do tří základních vrstev. Zelená hvězda v pozadí, žlutá „postava“ uprostřed a modrý nápis a šedý obrazec v popředí jako překážka 36
3.2 Na obrázku je zobrazeno červeně označené vlastní menu s ovládacími prvky. Je patrná jeho velikost a umístění. Obsah menu je vidět v detailu na obrázku 3.3. 46
3.3 Obrázek znázorňuje detail vlastního ovládacího menu z obrázku 3.2. Je vidět jaké útoky jsou implementovány a jaké jsou možnosti por nastavení útoků. 47
3.4 Obrázek znázorňuje, jak byla scéna změněna, když se nepovedlo jednotlivé modely jednoznačně identifikovat pomocí Stride. Proto se změnila celá scéna. Bílá barva byla zvolena v menu. 48
3.5 Na obrázku je vyobrazen vliv odhalení skrytých objektů. Protože v původní aplikaci je postava za obdélníkem schovaná, nevykreslí se. Po Aktivaci útoku se vypne ZBuffer, a proto se postava zobrazí. 49
3.6 Obrázek vyobrazuje překreslení modelu v pozadí do popředí. Ukazuje také, že tato změna neovlivní modely vykreslované později. Postava je vykreslena úplně nahoře, protože je opět zapnutý ZBuffer. Pro lepší kontrast byl použit útok pro změnu pozadí. 50
3.7 Na obrázku je vyobrazena ukázka vykreslování vlastních objektů, které sledují postavu. Je možné měnit tvar a barvu nového obrazce. 51
3.8 Na obrázku je ukázka odstranění objektu ze scény. Odstraněna byla hvězda v pozadí. 52

4.1 Na obrázku je vyobrazen vzhled aplikace rotujících kostek. 54
4.2 Na obrázku je odhalena skrytá kostka ze scény rotujících kostek. . 54
4.3 Na obrázku je zobrazena grafická scéna z aplikace Stone Giant. Skládá se z jednoho obra a několika pavouků umístěných v jeskyni. 55
4.4 Na obrázku je vyobrazena scéna s obrem, který je překrytý skálou. 56
4.5 Na obrázku je zobrazena scéna 4.4, na které je pomocí naimplementovaného útoku zobrazen obr, který byl částečně skryt za skálou. 57
4.6 Na obrázku je vyobrazena scéna se skálou. Za skálou je ovšem schovaný pavouk. 57

4.7 Na obrázku je zobrazena scéna 4.6, na které jsou pomocí navržených úprav zobrazení pavouci, kteří byli do té doby skryti. 58
4.8 Na obrázku je zobrazeno, jak jsou do scény přidány a vykresleny vlastní nové objekty. 58
4.9 Na obrázku je zobrazen obr, který se pomocí změny vykreslovaní ze scény odstraní na obrázku 4.10. 59
4.10 Na obrázku je zobrazena scéna 4.9, ze které byl odebrán obr. Zajímavý detail k povšimnutí je, že zmizel i obrův stín. 59
4.11 Na obrázku je zobrazen pavouk, který se pomocí změny vykreslovaní ze scény odstraní na obrázku 4.12. 60
4.12 Na obrázku je zobrazena scéna 4.11, ze které byl odebrán pavouk. Zajímavý detail k povšimnutí je, že zmizel i pavoukův stín. 60
4.13 Na obrázku je zobrazena scéna po vyčištění, proto na ní nejsou žádné objekty. Barva nového podkladu záleží na volbě v menu. . . 61

C.1 Na obrázku je zobrazena program pro vkládání dynamických knihoven GH Injector s pořadím, v jakém se se nastavuje. Kroky jsou popsané na straně 74. 74

Seznam zdrojových kódů

3.1 Kostra metody, která řídí vykreslování. Jsou zde ukázány volání ostatních komponen potřebných pro vykreslení. 38
3.2 Nejdůležitější prvky pro nastavení ZBufferu. Celé nastavení je dostupné ve zdrojovém kódu v příloze. 39
3.3 Restartování snímku do výchozí podoby. 39
3.4 Funkce z knihoven DirectX, která složí k prezentování vykreslené scény. 40
3.5 Nastavení zásobníku vrcholů pro statický model. 40
3.6 Přiřazení zásobníku vrcholů a indexů do Draphics Pipeline . . . 40
3.7 Změna polohy pohyblivého modelu. Změna spočívá v tom, že se do paměti zásobníku vrcholů vloží změněné vrcholy. 41
3.8 ColorShaderClass vykresluje obrazce pomocí této funkce. . . . 42
3.9 Kód zobrazuje postup pro nahrazení adress metod v tabulce virtuálních funkcí. 43
3.10 Kód zobrazuje funkci pro vložení položky do menu. 46
3.11 Kód zobrazuje, jak se mění vykreslovaný objekt. Nejdříve se identifikuje, a pokud je útok aktivní, tak se provede. 48
3.12 Kód zobrazuje mapování souřadnic středu pohyblivého modelu. 51
3.13 Kód zobrazuje, jak vymaže objekt ze scény. 52

Úvod

Když jsem byl mladší a hrávali jsme s přáteli počítačové hry, vždy se našlo několik podvodníků, kteří obcházeli pravidla. To se nezměnilo ani později, přestože se technologie stále rozvíjejí a zlepšují, a to i po bezpečnostních stránkách. Narazil jsem na grafické knihovny DirectX, když jsem se později začal zajímat o způsoby, jak z aplikací získávat dodatečné informace a podsouvat jí jiné. Knihovny DirectX slouží jako aplikační rozhraní pro grafické aplikace pro operační systém Windows.

Při hledání jsem zjistil, že se dynamické knihovny dají zneužít. Je možné upravit jejich funkce tak, aby při vlastní práci prováděly nějakou další funkcionalitu. Objevil jsem, že je způsob jak ovlivnit grafickou aplikaci, která využívá DirectX. Problém je, že jsou důkladně zmapované pouze knihovny z verze DirectX 9. Ty jsou dnes už zastaralé a nové aplikace je nevyužívají, protože průběhem času byla vyvinuta pokročilejší verze. Tou je DirectX 11, která je v grafických aplikacích dnešní doby hojně využívaná. Stále ovšem není dostupný žádný univerzální postup, jak novější verzi knihoven zneužít k vlastním účelům. Proto se tato práce zabývá verzí DirectX 11, aby tuto mezeru zaplnila.

Zneužívání grafických knihoven se používá pro získání výhody pomocí vizuálních vjemů. Celé to spočívá v tom, že se aplikace přinutí zobrazit i data, které jinak skrývá. Osobně mě nenapadá lepší využití než podvádění v multimediální formě zábavy, a to je i důvod proč se touto problematikou zabývám. Většinou nemám přístup ke zdrojovým kódům aplikací, které chci upravit, a proto musím využít způsobů, jež se dají aplikovat za běhu aplikace.

Pro správné uchopení celého problému jsem potřeboval pochopit, jakým způsobem fungují knihovny DirectX 11, a proč se nedají použít postupy ze starších verzí. Když jsem pochopil principy zobrazování a fungování grafických knihoven, potřeboval jsem zjistit obecné postupy pro ovlivňování grafické scény. Všechny běžně používané úpravy scény jsou popsány pro starší verzi, takže poté co jsem pochopil, jak fungují, musel jsem je převést do technologií novějších verzí. Nejlepší způsob, jak si všechny postupy vyzkoušet, je navrhnout si vlastní jednoduchou aplikaci, která poslouží jako odrazový můstek.

Jakmile zvládnu úpravy aplikovat na vlastní aplikaci, mohu se přesunout na reálné aplikace.

Cíle práce

Cílem této práce je prozkoumat fungování grafických knihoven DirectX 11 a zanalyzovat známé postupy na úpravu grafické scény ze starších verzí. Podle těchto informací se má sestavit univerzální postup jak ovlivňovat zobrazení grafické aplikace za jejího běhu. To znamená najít způsob, jak jednotlivé modifikace scény převést do verze DirectX 11. Proto by výsledkem této práce měla být šablona pro upravování aplikace, která bude obecně použitelná na jakoukoliv grafickou aplikaci využívající příslušné knihovny. Šablona by měla obsahovat všechny úpravy ze zadání, což je například přidávání a mazání objektů nebo změna jejich parametrů.

Dále je cílem navrhnout a naprogramovat vlastní grafickou 3D scénu za použití knihoven DirectX 11. Scéna by měla být dostatečně komplexní, aby na ní bylo možné prezentovat všechny možnosti, jak ji ovlivnit podle zadání. Po přečtení bakalářské práce by čtenář měl být schopný popsané postupy replikovat.

V první analytické části se popisují základní principy vykreslování grafické scény pomocí DirectX 11, které jsou nezbytně nutné pro pochopení celého procesu. Dále se zabývá rozborem již existujících úprav scény za běhu programu, jež se běžně používají na reálných aplikacích. Zkoumá také rozdíly mezi verzemi knihoven DirectX, konkrétně mezi devátou a jedenáctou verzí. Na závěr analýzy se shrnují možnosti vložení úprav do běžící aplikace.

Po pochopení postupů z analýzy následuje návrh aplikace a šablony. Návrh se skládá z přípravy architektury programu, rozvržení jeho komponent a návaznosti mezi nimi. V návrhu šablony jsou rozepsány úpravy ze zadání a jejich návaznost na samotnou implementaci. Na konci návrhu jsou vyjmenovány všechny nástroje, které byli při práci použity.

V realizaci práce je pak popsáno konkrétní řešení jednotlivých komponent aplikace a popis prvků grafické scény. V části, která se věnuje implementaci šablony, jsou rozebrány jednotlivé úpravy navržené scény a rozděleny do kategorii podle povinných částí ze zadání.

V poslední řadě je popsáno testování, kdy byl vypracovaný postup zkoušen na jiných aplikacích než na té vlastní.

Kapitola 1

Analýza

Tato kapitola se zabývá teorií za vykreslováním 3D scén na počítači. Rozebírá běžné knihovny, které se k tomu využívají na OS Windows. Zaměří se na verzi knihovny DirectX. Popisuje především verzi DirectX 11, její grafické moduly a způsob, jakým pracují. Dále popíše běžně používané metody zneužití právě těchto knihoven[1] k ovlivňování 3D scény a původní aplikace. Implementace jednotlivých metod pro DirectX 9 je velmi dobře popsaná a dostupná na internetu, ale již není kompatibilní s novější verzí DirectX 11, proto se v této kapitole popisují rozdíly mezi těmito verzemi. Na závěr zkoumá možnosti, jak dostat cizí kód, který má za účel ovlivnit vykreslování, do původní aplikace za jejího běhu, s využitím takzvaného „DLL injection“.

1.1 DirectX

DirectX je sada aplikačních rozhraní vyvinutá společností Microsoft[2] v roce 1995 a byla navržena primárně pro práci s multimédii a další formy zábavy v IT prostředí [1]. Účelem bylo usnadnit tvorbu multimédií (počítačových her, uživatelských rozhraní a dalších) pro platformy s OS Windows, a motivovat tak vývojáře pracovat pod tímto operačním systémem. Umožňuje programu posílat instrukce přímo grafickému a zvukovému hardwaru a periferiím (klávesnice, myš, joystick a další). Díky přímému přístupu k hardwaru pracují aplikace se svými zdroji efektivněji a zlepšuje se jejich výkon.

Nejaktuálnější verze je DirectX 12, která je součástí Windows 10 [1]. Tato práce se ovšem zaměřuje na DirectX 11. Tato verze byla vydána s OS Windows 7. Za dobu, kdy byla tato verze nejaktuálnější, byla vyvinuta spousta aplikací s její podporou.

DirectX je složený z několika samostatných knihoven rozdělených podle oblastí, o kterou se starají. Tato práce se bude blíže zabývat pouze knihovnami

[1] Konkrétně z verze DirectX 9 a DirectX 11

[2] Proto se DirectX dá použít pouze pro operační systém Windows a později i pro Xbox.

Direct3D a DXGI, které jsou důležité, protože se starají o vykreslování 3D scén a jejich zobrazování. Ostatní jsou k nalezení na stránkách Microsoft MSDN [2].

1.2 Základní objekty Direct3D

Nejdříve se tato práce zaměří na vykreslování základních objektů, protože je důležité chápat, jak se tvoří vykreslovaná scéna, aby bylo možné tento proces zneužít a ovlivnit, tak aby se scéna změnila požadovaným způsobem.

Aplikační rozhraní Direct3D se stará o vykreslování 3D grafiky do na grafického zásobníku (v originále buffer), který bude zobrazován monitorem. S jeho pomocí je aplikace schopna vykreslovat ze zadaných vrcholů pět základních topologií. Tyto topologie jsou:

- 1.2.1 PointList (obr. 1.1, str. 4)
- 1.2.2 LineList (obr. 1.2, str. 5)
- 1.2.3 LineStrip(obr. 1.3, str. 5)
- 1.2.4 TriangleList (obr. 1.4, str. 6)
- 1.2.5 TriangleStrip (obr. 1.5, str. 6)

Jednotlivé objekty těchto topologií se nazývají Primitives[3]. Jakékoliv další komplexnější obrazce z nich musí být složené[4] [3]. Pokud se vykresluje 3D scéna, tak se tyto objekty používají k modelování světa, do kterého je tato scéna zasazená. To znamená, že se z nich tvoří stavby, přírodní útvary, rostliny, živočichy, projevy počasí a mnoho dalších.

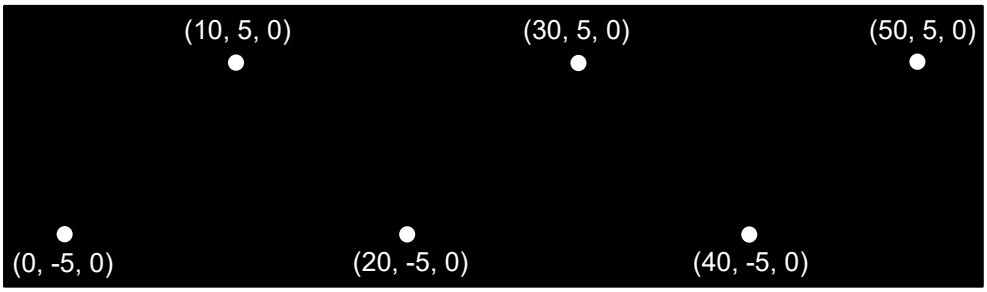
Při vykreslování objektů se musí dbát na matici světa a pohled do ní. Jedná se o kartézskou soustavu souřadnic o třech dimenzích a definování polohy „kamery“ ve scéně a směr, jakým je natočená. Na pohledu záleží kvůli způsobu, jakým se zpracovávají zmíněné topologie. Vrcholy topologií jsou zadávány právě pomocí souřadnic v této soustavě a zobrazí se pouze za určitých podmínek. Za prvé musí být v oblasti, kam směřuje pohled, a dále vrcholy primitivních topologií celého objektu musí být zadány v určitém pořadí. Tyto vrcholy se zadávají do zásobníku vrcholů, který se dále zpracovává. Jednotlivé topologie mají svoje další pravidla pro zpracování.

1.2.1 PointList

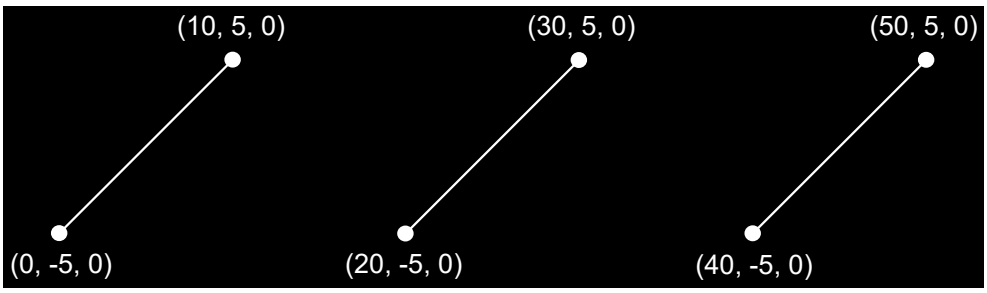
U této topologie nezáleží na pořadí zadaných vrcholů, a jelikož se jedná o samostatné body bez vazby na ty další, není ani jejich počet zatížen podmínkou. Jeden bod má velikost jednoho pixelu. Ukázka jak se zpracovává tato

[3] Primitive je jeden bod, jedna čára nebo jeden trojúhelník.

[4] Od verze DirectX 10 přibyly ještě rozšířené topologie, které kombinují těch pět základních.



Obrázek 1.1: Na obrázku je zobrazen příklad, jak může vypadat topologie PointList po zpracování.



Obrázek 1.2: Zde je znázorněn objekt po zpracování v topologii LineList. Jednotlivé vrcholy jsou spojeny do oddělených čar.

topologie je vyobrazeno na obrázku 1.1. PointList se využívá především na vykreslení scény s hvězdnou oblohou, tečkovanou čáru, k označení pozice na mapě[5] a další [4].

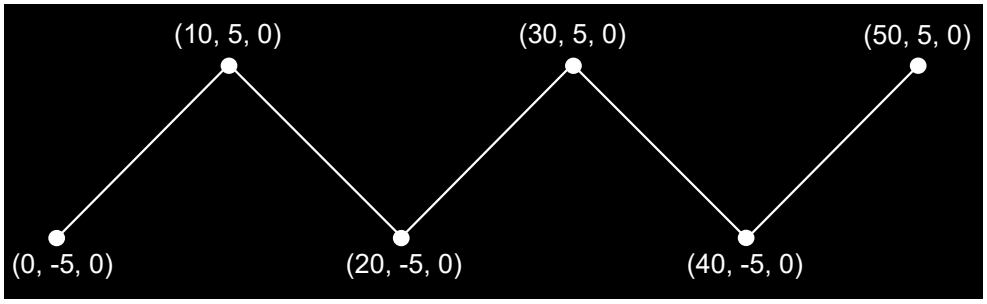
1.2.2 LineList

U této topologie se vrcholy na grafickém zásobníku spojují po dvojicích do samostatných úseček. Takže první vrchol se spojí s druhým, třetí se čtvrtým a takto to pokračuje, dokud nejsou spojeny všechny body viz Obrázek 1.2. Proto u této topologie platí, že počet vrcholů musí být sudý, jinak by zůstal osamělý vrchol a Direct3D by nevěděl, co s ním udělat. LineList se používá zejména na vykreslování 3D mřížky, hustého deště a dalších [4].

1.2.3 LineStrip

LineStrip je topologie velice podobná topologii LineList. Jediný rozdíl je, že spojuje všechny body do jedné spojené čáry viz Obrázek 1.3 . Z toho plyne, že nemusí mít sudý počet vrcholů, přesto musí mít alespoň dva pro vytvoření

[5]Něco jako označení pozice v navigaci. S jediným rozdílem, že tato navigace ukazuje pozici ve světě, ve kterém se zobrazovaná scéna nachází



Obrázek 1.3: Topologie LineStrip se moc neliší od topologie LineList 1.2. Rozdílem je že se spojí postupně všechny vrcholy.

úsečky. Tato topologie je užitečná především pro vytváření obrazců s linkovou strukturou[6] nebo například pro listy trávy [4].

1.2.4 TriangleList

Tato topologie vykresluje ze zadaných bodů samostatné trojúhelníky viz Obrázek 1.4. Princip je skoro totožný jako u vykreslování úseček a čar, ale je třeba splnit několik podmínek, aby se trojúhelníky vykreslily. Každý trojúhelník spojuje trojici bodů, a proto musí být počet všech vrcholů dělitelný třemi, protože zbylé body by nebylo, jak zpracovat. První trojúhelník tvoří první trojice ze zadaných vrcholů, druhý druhá trojice. Takto se pokračuje, dokud se nevyčerpají všechny vrcholy na zásobníku. Aby se trojúhelník zobrazil, musí být jeho trojice bodů zadána ve směru hodinových ručiček vůči definovanému pohledu. Tedy body trojúhelníku z obrázku 1.4 mohly být zadány v jednom ze tří pořadí[7]. Pokud by byli zadány v jiném, tak se nezobrazí[8], přesto jsou body zpracovány, neboť kdyby byl pohled definován z jiného úhlu, už by se tento trojúhelník mohl zobrazit. Dá se tedy chápat, že „zadní“ strana této topologie je neviditelná. Toho se využívá právě u 3D objektů, aby se zbytečně nevykreslovaly části, které jsou určené pro pohled z jiných směrů. TriangleList se využívá hlavně k implementaci efektů, jako jsou silové pole nebo exploze [4].

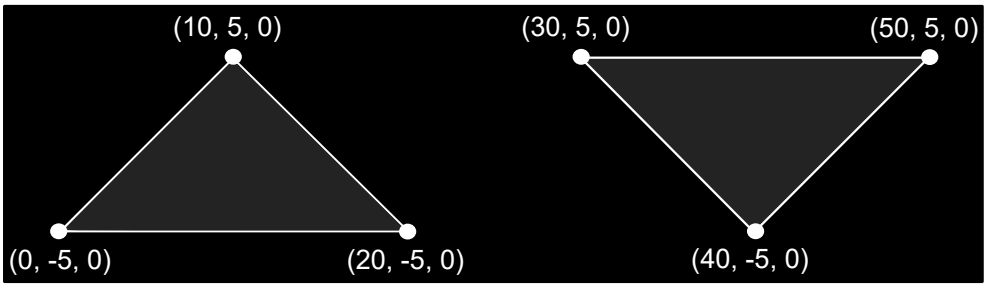
1.2.5 TriangleStrip

TriangleStrip opět vykresluje trojúhelníky, tentokrát ale do souvislé plochy viz Obrázek 1.5. Zpracování probíhá tak, že se z prvních tří vrcholů utvoří trojúhelník a každý další bod se k němu přidá a zformuje tak přilehlý trojúhelník. Použijí se k tomu dva poslední vrcholy. Tedy druhý trojúhelník je vytvořen z druhého, třetího a čtvrtého vrcholu. Tímto způsobem se zpracují

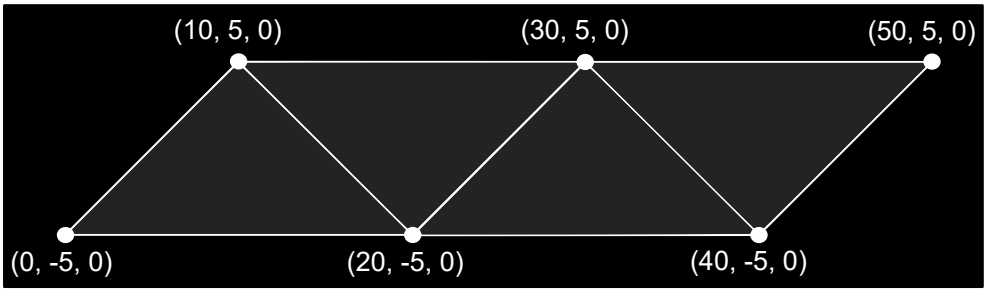
[6] Šrafovaná grafika, vyobrazení vláken látky a dalších.

[7] A to {(20, 5, 0),(0, -5, 0),(10, -5, 0)}, {(10, 5, 0),(20, -5, 0),(0, -5, 0)} nebo {(0, 5, 0),(10, -5, 0),(20, -5, 0)}.

[8] Toto chování se dá přenastavit, ale tím se tato práce nebude zabývat.



Obrázek 1.4: Obrázek vyobrazuje topologii TriangleList. v této topologii se vrcholy zpracovávají do samostatných trojúhelníků.



Obrázek 1.5: Topologie TriangleStrip je podobná s topologií TriangleList 1.4. Rovněž se z vrcholů vykreslují trojúhelníky, ale tentokrát do souvislé plochy.

všechny zadané vrcholy. Díky tomu nemusí být počet vrcholů dělitelný třemi, ale měl by mít alespoň tři vrcholy. Právě tato topologie se používá pro vykreslování 3D modelů [4].

1.3 Graphics Pipeline Direct3D

DirectX 11 potřebuje víc než jen seznam vrcholů, aby vykreslil požadovanou scénu. Mezi další informace, jež je potřeba dodat, patří například barva, nasvícení, textura a mnoho dalších [5]. Způsob vykreslování 3D scény by se dal přirovnat ke skládání skládačky, kde si Direct3D vyrábí jednotlivé dílky[9] za pochodu. Hned, jak takový dílek vyrobí, tak ho umístí na správné místo. Tento proces se nazývá Graphics Pipeline [6], probíhá na grafické kartě, která se přidělí všechny výše zmíněné informace a další potřebné zdroje. Vykreslovací proces je rozdělen do několika stádií:

- 1.3.2 Fáze Input–Asembler (str. 10)
- 1.3.3 Fáze Vertex Shader (str. 11)
- 1.3.4 Fáze Tesselace (str. 11)

[9]Každý dílek je objekt určený k vykreslení

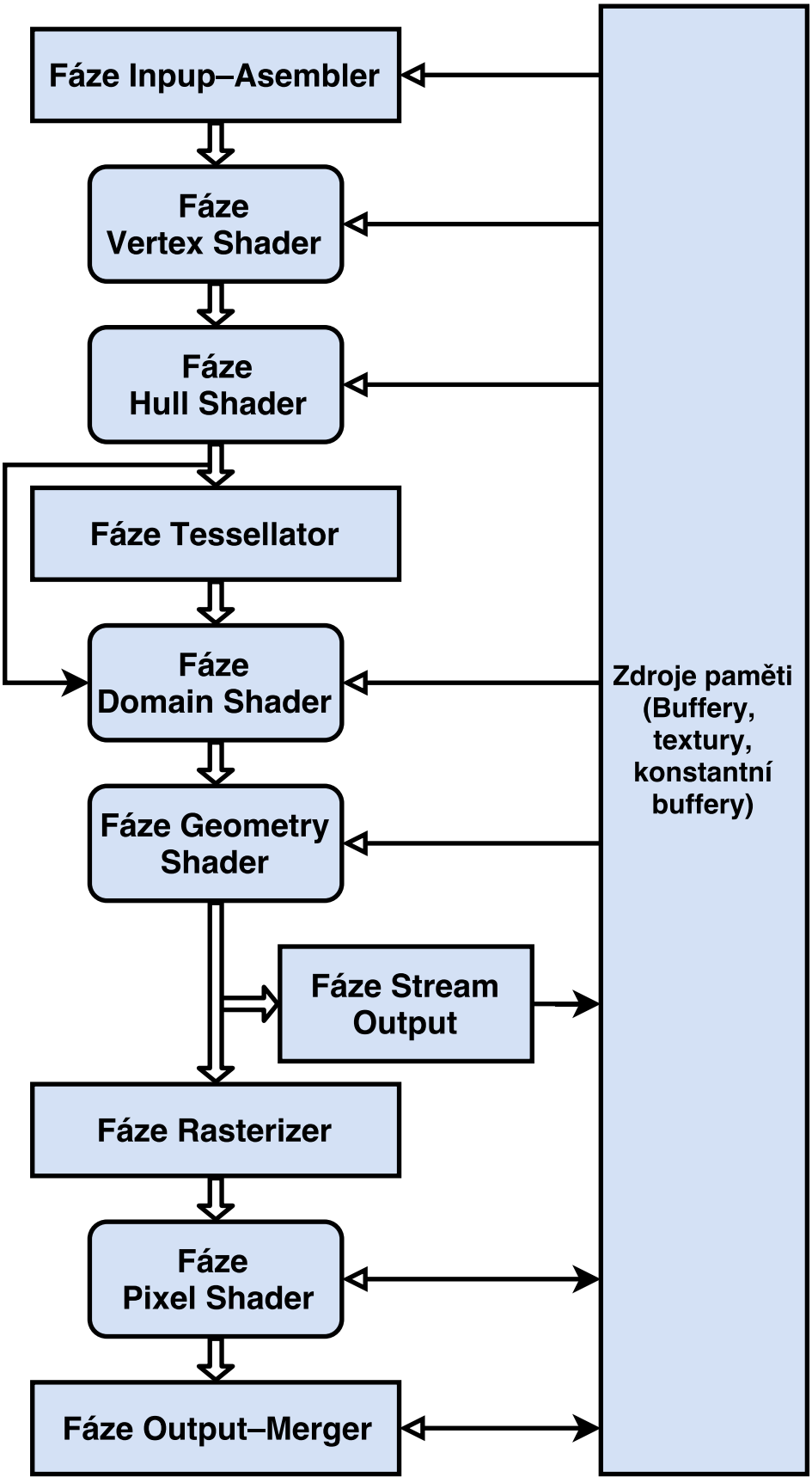
- 1.3.5 Fáze Geometry Shader (str. 11)
- 1.3.6 Fáze Stream–Output (str. 12)
- 1.3.7 Fáze Rasterizer (str. 12)
- 1.3.8 Fáze Pixel Shader (str. 13)
- 1.3.9 Fáze Output-Merger (str. 13)

Jeho průběh je vidět na schématu 1.6 a je popsán níže. Pro snazší pochopení průběhu Graphics Pipeline je nejdříve potřeba zmínit, jakým způsobem se uchovávají data pro 3D scénu. Každý dílek z celkového obrazu je jedna ze základních topologií. Tyto topologie byly popsány v sekci 1.2 Základní objekty Direct3D. Dílek může být libovolně velký a složený z libovolného množství Primitives. Ty ale nemohou patřit do dvou nebo více topologií. Většinou je každý dílek uložen v paměti[10] jako seznam vrcholů. Výjimkou je, když jsou vrcholy definované v samotné shaderu. Zásobník, na kterém je dílek uložen, může být bud konstantní nebo dynamický. Dílky se ukládají na dynamický zásobník, pokud jsou předurčené k pohybu, protože je pak možné měnit pozice vykresleného objektu, aniž by se musel generovat nový zásobník vrcholů s posunutým dílkem.

Takový seznam vrcholů slouží jako vstup pro Graphics Pipeline. Jakým způsobem se body vykreslí záleží na nastavení jednotlivých fází. To se určuje pomocí programovatelných shaderů nebo různých zásobníků, které autor 3D scény přiděluje Graphics Pipeline. Dále se proces vykreslování řídí nastavitelnými příznaky, které se buď nastavují v průběhu zpracování, nebo jsou nastaveny autorem. Výstup z této Pipeline se zpravidla zapisuje na BackBuffer (Tento pojem je blíže vysvětlen v sekci 1.4 DXGI), který je následně použit pro zobrazení. BackBuffer je grafický zásobník určený pro monitor, který podle něj určí, který pixel má rozsvítit a jak. Výsledek se zobrazí až poté, co do něj jsou vloženy zpracované dílky. Nemusí to nutně být všechny definované dílky. Některé mohou být určené jen pro jiný snímek. Z toho plyne, že Graphics Pipeline se volá několikrát, než je dílo kompletní. O tomto procesu se bude dále mluvit jako o vykreslovací smyčce.

Běžnou praxí je grafický zásobník vyčistit než započne vykreslovací smyčka, protože by v něm jinak zůstaly předešlé modely, které by změnili kontext nového snímku. Tyto staré objekty nebo jejich části by mohly překrýt ty nové. Bez vyčištění by bylo velice komplikované tyto staré části vyhledat a vyjmout z konečného grafického zásobníku. Kdyby ve scéně zůstaly staré části, tak by se nová a stará scéna překryly a sloučily.

[10] Každý do svého vlastního zásobníku vrcholů.



Obrázek 1.6: Schéma zobrazuje Graphics Pipeline a jeho fáze, které využívá pro zpracování 3D scény v pořadí v jakém se volají.

1.3.1 Shader

Shader je jednoduchý program, který se stará o matematiku skrytou za vykreslováním grafických objektů. Výpočty se provádí na grafické kartě. Mezi takové matematické operace patří například transformace do tří–dimenzionálního prostoru nebo obarvování jednotlivých pixelů. Existuje několik druhů shaderů, a to Vertex Shader, Pixel Shader, Geometry Shader a další. Každý shader je volán velice často, a proto musí být jednoduchý [7].

1.3.2 Fáze Input–Asembler

Účelem této fáze je připravit vrcholy, které definují jeden dílek, k průchodu Graphics Pipeline. To se skládá z několika kroků:

1. Přiřazení základní topologie celému dílku.

   Není možné přiřadit dvě různé na jeden dílek. Pokud to celková scéna vyžaduje, je nutné takový obrazec rozložit do více dílků a ty pak složit dohromady. Jak již bylo řečeno, dílek je uložen na svém zásobníku jako seznam vrcholů. To co určuje jaký bude mít výsledný tvar, je právě topologie. Dejme tomu, že je na zásobníku připravená hvězda pro TriangleStrip. Při volbě jiné topologie by tento obrazec nemusel hvězdu ani připomínat.

2. Nastavení takzvaného Index Bufferu.

   Index Buffer je zásobník, který pomáhá určit pořadí vrcholů a usnadňuje tak práci grafické kartě. Dá se to představit jako obsah v knize. Ukazuje, kde se mají hledat další vrcholy. To umožňuje měnit pořadí vrcholů a tím i samotný obrazec, aniž by se musel upravovat zásobník, na kterém je dílek uložen.

3. Generování pomocných informací pro budoucí zpracování.

   Díky těmto informacím jsou pozdější fáze mnohem efektivnější, protože usnadňují některé výpočty nebo je dokonce umožňuje přeskočit. Mezi tyto informace patří například:

   - Identifikační čísla pro jednotlivé Primitives.
   - Značka, jestli je daná Primitive natočená směrem ke kameře.
   - Další značky a bližší informace o nich jsou na stránkách MSDN [8].

V této části lze ovlivnit výslednou scénu nejvíce. Pokud změníme zásobník vrcholů nebo jeho Index Buffer, tak jsme schopni úplně změnit pozici objektu (za předpokladu, že není uložen na konstantním zásobníku), nebo dokonce objekt samotný. Je důležité zmínit, že se takto nedá změnit reálná pozice

objektu nebo jeho typ v původní aplikaci[11]. Změní se jen místo a tvar jakým se výsledek zobrazí na monitoru.

1.3.3 Fáze Vertex Shader

V druhé fázi hraje hlavní roli Vertex Shader, což je malý a jednoduchý program určený k manipulaci s jednotlivými vrcholy[12]. Tento shader vždy pracuje pouze s jedním vrcholem dílku. Mezi operace, které provádí, patří například transformace [9]. Tato fáze nemůže být vynechána a Graphics Pipeline musí mít vždy přidělený nějaký Vertex Shader. Pokud chceme, aby se v této fázi s dílkem nic nestalo, musíme nastavit takový shader, který nic nedělá, takže každý vrchol vrátí nedotčený.

Pokud se prohodí Vertex Shader za jiný, je možné dále ovlivnit zpracování jednotlivých vrcholů nebo přidat nové, pokud se správně zvolí shader.

1.3.4 Fáze Tesselace

Fáze Hull–Shader, Tessellator a Domain-Shader ze schématu 1.6 se starají o tesselaci vykreslovaného objektu. Tesselace je proces, kdy se dílek vykreslí s mnohem větším počtem detailů, než měl původně definován. Docílí se toho tak, že se výsledné plochy vyplní větším množstvím Primitives z topologie TriangleList nebo TriangleStrip. Tesselace jsou dvojího typu:

- Softwarová
- Hardwarová

DirectX 11 využívá hardwarového řešení, které je mnohem efektivnější a dokáže produkovat výrazně větší množství detailů. Tato fáze nemusí být aktivní a testovací aplikace této funkcionality nevyužívá. Proto se jimi tato práce nebude více zabývat. Podrobné informace, jak je tesselace řešená na DirectX 11, jsou na stránkách Microsoft MSDN [10].

1.3.5 Fáze Geometry Shader

Geometry Shader pracuje velice podobně jako Vertex Shader s tím rozdílem, že dokáže zpracovat celou Primitive naráz.

- Po jednom bodě zpracovává PointList.
- Po dvou bodech zpracovává LineList a LineStrip.

[11] Původní aplikace si uchová původní pozici a tvar objektu. K tomu se využívají jiné nástroje a postupy.

[12] Nesmí být moc složitý, protože je volán velice často. Kdyby scéna obsahovala například 5 000 trojúhelníků, tak se na každý snímek musí zavolat 15 000krát. Při obnovovací frekvenci šedesáti snímků za sekundu to činí 900 000 volání za sekundu.

- Po třech bodech zpracovává TriangleList a TriangleStrip.

Navíc tento shader zvládá pracovat i s rozšířenými topologiemi. Ani tato fáze není povinná a testovací aplikace tento Shader nevyužívá. Způsob, jak přesně tento shader funguje, je popsán na stránkách Microsoft MSDN [11].

1.3.6 Fáze Stream–Output

Účel této fáze je neustále ukládat výsledky dosavadního postupu. Ty je možné uložit na jeden nebo více zásobníků. Takto uložená data se mohou opětovně načíst do Graphics Pipeline nebo mohou být zkopírované do sdílených paměťových zdrojů, aby byla přístupná i pro klasický procesor (CPU).

1.3.7 Fáze Rasterizer

Z předchozích fází[13] je vygenerovaný vektorový obrázek tvořený tvary z Primitives a v této části se převede na rastrový obrázek složený z pixelů[14]. Tento výsledek je určený k zobrazení monitorem. Než se tak stane, je potřeba tento dílek zasadit do celkové skládačky. Během převodu obrázku je každá z Primitive převedena na pixely. Převod zahrnuje:

- Počítání perspektivy
- Mapování na umístění v definovaném světě
- Mapování do dvou–dimenzionálního prostoru
- Určení, jak se použije Pixel Shader

1.3.7.1 ZBuffer

V této části se rovněž začíná testování takzvaného Zbufferu a provádí se i v každé následující fázi. ZBuffer určuje hloubku dané Primitive a jeho název je odvozen z osy z [12]. Používá se ke zjištění překryvu jednotlivých částí obrazu. Pokud s je část překrytá jiným, tak se pomocí ZBufferu určí, který z nich se má zobrazit. Určuje to podle vzdálenosti od definované kamery. Takže pokud bude daná 3D scéna obsahovat dům s jeho vlastníkem uvnitř, tak nám tento mechanizmus zařídí, že se vlastník nezobrazí, protože je schovaný za zdí svého domu. Program o něm ale ví a musel ho vyrobit, aby mohl posoudit jestli je uvnitř domu nebo předním. ZBuffer je volitelná vlastnost a dá se vypnout. V takovém případě nové obrazce překreslují ty staré. V této fázi se nastaví pouze příznak a samotné testování provádí až následující fáze.

[13] Fáze Input–Asembler, Fáze Vertex Shader, Fáze Tesselace a Fáze Geometry Shader.
[14] Stále se jedná pouze o jeden dílek celkového obrazu.

1.3.8 Fáze Pixel Shader

Tato fáze je úzce spjatá s fází 1.3.7 Fáze Rasterizer, protože ta aktivuje Pixel Shader pro každý pixel. Tato část určuje barvu dílku a pokrývá ho texturou, pokud je potřeba. K tomu, jak vybarvit jednotlivé pixely, se používají všechny dostupné informace, ať už jsou výsledkem předchozích fází nebo dosazené napevno. Mezi informace, které se v tomto procesu používají, patří:

- Konstantní proměnné
- Textury
- Interpolované vektory
- Lokaci Pixelu
- A další [8]

Zde se rovněž testuje ZBuffer v rámci jednoho vykreslovaného dílku. To znamená bez návaznosti na celkovou scénu. Tím pádem kontroluje překryv jednotlivých Primitive a vykreslí tak konečnou podobu před zasazením do 3D scény.

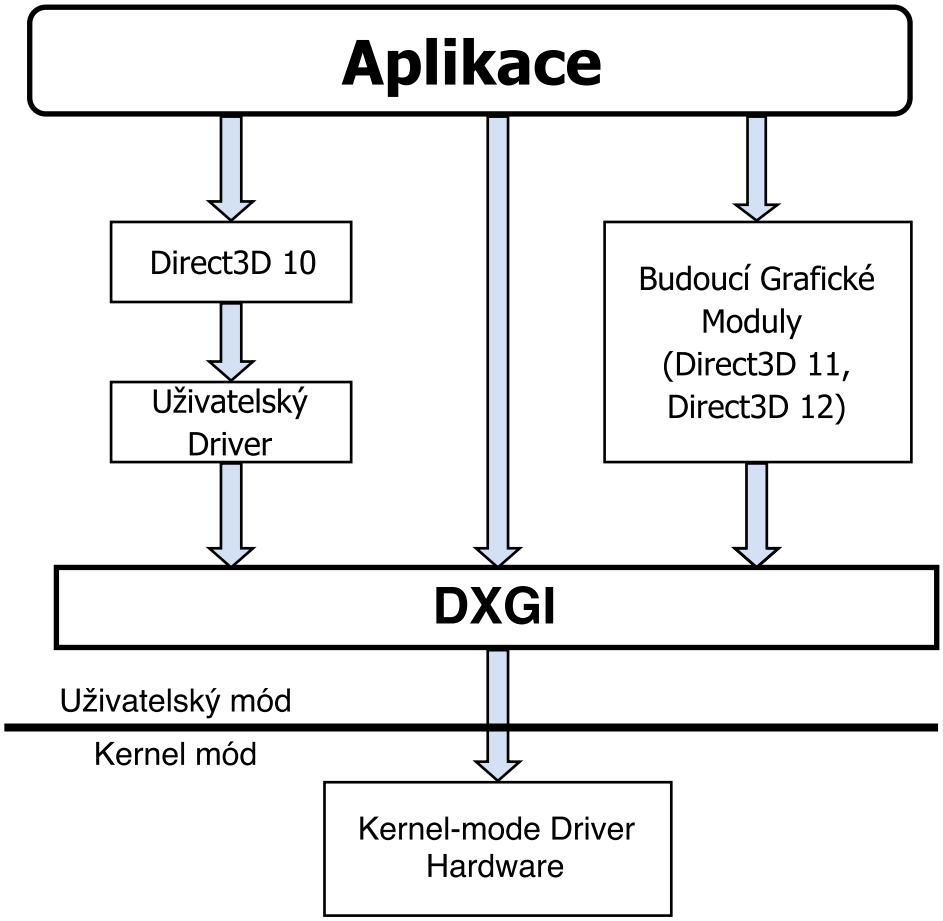
Této a předchozí části se rovněž hojně využívá. V těchto fázích již není možné změnit pozici nebo tvar objektu, ale lze ovlivnit způsob, jakým se vykreslí, a to změnou nastavení ZBufferu a Pixel Shaderu. Pak se objekty skryté za jinými mohou zobrazit i když by neměli nebo změnit jejich barvu či texturu. Toho se v praxi využívá nejvíce viz sekce 1.5 Způsoby ovlivňování 3D scény pomocí DirectX.

1.3.9 Fáze Output-Merger

Když je dílek skládačky hotový, musí se umístit na své místo. A to doslova. V této části se vytvořený výsledek zapíše na grafický zásobník, který slouží pro zobrazení na monitor. Při zápisu se naposledy otestuje ZBuffer. Tentokrát se v potaz vezme celá 3D scéna. Takže se bude nově vložený dílek testovat s těmi, co už ve scéně jsou. Pokud je pro nějaký dílek ZBuffer vypnutý, tak se do testování nezahrnuje a zůstane vždy „nahoře“. Tuto vlastnost lze využít pro ovlivnění scény, kdy se na objektu, který chceme odkrýt, vypne ZBuffer. Tato část určí konečnou podobu pixelu, který se vykreslí.

1.4 DXGI

Microsoft DirectX Graphics Infrastructure (DXGI) je aplikační rozhraní, které se přidalo do DirectX s verzí 10 a je obsažena i ve všech novějších. Jeho úlohou je převzít od Direct3D nízkoúrovňové úkoly, jež mohou být nezávislé na zpracování grafických úkonů DirectX [13]. Jak je vidět na schématu 1.7, leží

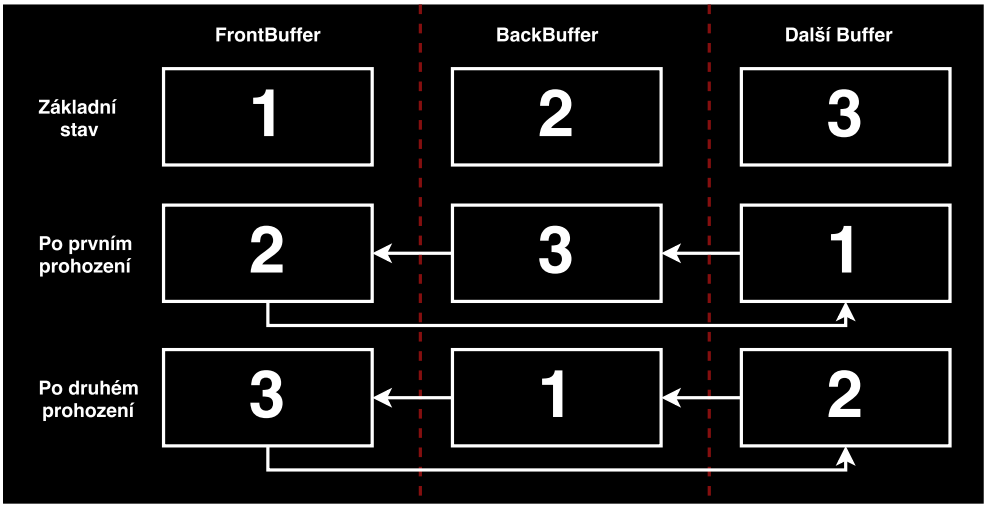


Obrázek 1.7: Schéma umístění vrstvy DXGI v architektuře programu a jeho komunikaci s Hardwarem

tato vrstva mezi hardwarem a grafickými knihovnami a stará se o správu otevřených oken a předávání grafických zásobníků monitoru případně několika monitorům, čímž usnadňuje práci s více výstupními zařízeními. Tento zobrazovaný grafický zásobník je vygenerovaný postupem popsaným v sekci 1.3 Graphics Pipeline Direct3D.

Tato knihovna spravuje takzvaný SwapChain, který byl zaveden pro větší plynulost obrazu. Jedná se o princip dvou a více grafických zásobníků, které se střídají. Tyto zásobníky jsou rozděleny na:

- FrontBuffer
- BackBuffer
- Ostatní grafické zásobníky napojené na SwapChain



Obrázek 1.8: Na obrázku je znázorněno, jakým způsobem se prohazují grafické zásobníky ve SwapChain

FrontBuffer je právě zobrazován na obrazovce a do BackBufferu se většinou provádějí změny z předchozí sekce 1.3 Graphics Pipeline Direct3D. Když jsou všechny změny zapsané, jednoduše se FrontBuffer a BackBuffer prohodí[15] viz. Schéma 1.8 a vymění si i role. Tento mechanizmus je použit, aby monitor nemusel neustále překreslovat změny na obrazovku, což způsobuje potíže s plynulostí obrazu. A proto se všechny změny prezentují najednou. K této knihovně se dá přistoupit buď na přímo, nebo prostřednictvím Direct3D.

Tohoto mechanizmu se využívá k přidávání úplně nových objektů, které nebyly definované v původní aplikaci. Dělá se to tak, že útočník, těsně předtím než se FrontBuffer a BackBuffer prohodí, vyvolá další průchody Graphics Pipeline k vykreslení svých nových objektů.

1.5 Způsoby ovlivňování 3D scény pomocí DirectX

Existuje mnoho způsobů, jak zneužít grafických knihoven DirectX. Dále se o těchto metodách bude mluvit jako „útoky na DirectX“. Tyto útoky nejsou omezeny na žádnou verzi, ale musí být náležitě přizpůsobeny podobě a obsahu verze DirectX, na kterou je útok navrhován, protože ne všechny verze knihoven DirectX používají stejné postupy viz sekce 1.6 Rozdíl mezi DirectX 9 a DirectX 11. Tyto útoky se používají zejména při podvádění v multimediálních aplikacích [14], kde proti sobě lidé (hráči) soutěží v různých hrách. Vzhledem k tomu, že takovou komerční aplikaci zpravidla vyvíjí třetí strana, jsou tyto útoky prováděny bez přístupu k původním zdrojovým kódům.

[15] Posunou se i všechny ostatní pomocné zásobníky.

Tato práce se zaměřuje na útoky, jež ovlivňují vykreslování obsahu, ale z dat poskytnutých knihovnám DirectX se dá zjistit spousta dalších užitečných informací o vykreslovaných objektech:

- Pozice objektu
- Počet objektů
- Typ objektu
- A další

Tyto informace se dají využít pro manipulaci s původní aplikací. Při správné analýze dat je možné vypracovat rozsáhlé útoky jako například umělá inteligence, která nahradí hráče. Jako ukázka takového využití DirectX může posloužit Starcraft 2 Automated Player [15]. V této bakalářské práci se ovšem cílí na jednodušší a přímočařejší využití zneužití DirectX, které reálně změní, co se na obrazovce objeví. Útoky, které se dosud běžně používají, jsou:

- 1.5.1 WallHack (str. 17)
- 1.5.2 ESP Hack (str. 18)
- 1.5.3 AimBot (str. 19)
- 1.5.4 Custom Crosshair (str. 20)
- 1.5.5 Odstranění stínů (str. 21)
- 1.5.6 DirectX Overlay (str. 21)

Zde uvedené útoky jsou dobře zmapované a zdokumentované pro Verzi DirectX 9, proto jsou jejich implementace dobře dostupné na internetu [14]. Cílem této práce je využít principů těchto útoků pro navržení vlastních způsobů, jak ovlivnit 3D scénu ve verzi DirectX 11, ke které už tolik implementací známo není. Princip útoků zůstává pro všechny verze stejný[16], co se ale změnilo je struktura samotné knihovny. Už s verzí DirectX 10 zmizela zpětná kompatibilita s DirectX 9 a DirectX 11 je v zásadě jen rozšíření pro DirectX 10 [16]. V následující sekci 1.6 Rozdíl mezi DirectX 9 a DirectX 11 se tato práce podívá na nejdůležitější změny mezi posledními používanými verzemi DirectX.

Všechny útoky využívají výše zmíněné Graphics Pipeline[17] a modulu DXGI, a to tak, že z ní extrahují data o vykreslovaných objektech, upravují parametry a nastavení z originální aplikace, přidají nové objekty nebo naopak některé informace smažou úplně. Konkrétní a finální podoba útoku záleží na autorovi. Je

[16] Je dokonce aplikovatelný na OpenSource Grafickou knihovny OpenGL.

[17] Zde popsaná Graphics Pipeline je z verze DirectX 11, ale v jiných podobách byla i ve starších verzích [17], [10].

přitom ovšem potřeba dbát i na původní aplikaci, neboť její nastavení nemusí umožňovat některé úpravy a navíc jednotlivé implementace nejsou přenositelné mezi různými aplikacemi, protože jsou psány pro ovlivnění konkrétních objektů. Nepřenositelné jsou proto, že jiné aplikace tyto objekty nemají[18]. Metody, jak vložit cizí kód, jsou blíže popsány v sekci 1.7 Vkládání škodlivého kódu do původní aplikace.

Útoky jsou popsány podle informací z těchto zdrojů [18], [19], [20].

1.5.1 WallHack

WallHack je útok, kdy se odhalí objekty skryté za libovolnou překážkou. Případy použití tohoto útoku se dají znázornit u hry na schovávanou. Hráč, který hledá, takto dostane výhodu, protože uvidí všechny hráče i přesto, že jsou schovaní a nebude se tak muset namáhat s jejich hledáním. Je to jako kdyby každý schovaný hráč napsal na svou skrýš velkým červeným písmem „Jsem Tady!“, tak aby na nápis bylo odevšud vidět.

WallHack se provádí tak, že se při průchodu vykreslovací smyčkou deaktivuje ZBuffer pro objekty, které chce útočník odkrýt, v okamžiku kdy se objekt vykresluje. To zapříčiní, že se ukáže úplně na popředí. Poté, co se objekt vykreslí, je ZBuffer opět aktivován, aby se zbytek scény vykreslil tak, jak bylo původně zamýšleno. Vybraný objekt se při vypnutém ZBufferu překryje všechny modely, které ve scéně jsou. Jelikož není zahrnutý do testování ZBufferu, tak se z tohoto objektu stává pozadí pro jakékoliv další objekty, které se zpracovávají později, a proto se může stát, že bude tento objekt znovu překryt jinými. Často se mění i textura hráče na nějakou ze základních barev[19], aby lépe vynikl v celkové scéně, jak je vidět na obrázku 1.9.

Je potřeba předem identifikovat model odkrývaného objektu, aby bylo možné vypínat ZBuffer jen pro něj. K tomu se používá takzvaný Stride, ByteWidth případně dalších parametrů zásobníku vrcholů. Stride nebo ByteWidth se používají hlavně proto, že určují rozměry modelu a počet vrcholů. Stride určuje velikost jednoho vrcholu a ByteWidth říká, jak je velký celý objekt[20]. Jak je z obrázku 1.9 patrné, pokud se nepodaří jednoznačně identifikovat například model hráče, tak se útok aplikuje i na všechny ostatní modely odpovídající zvolenému identifikátoru. WallHack je aktivní, i když vykreslovaný objekt není nikde schovaný.

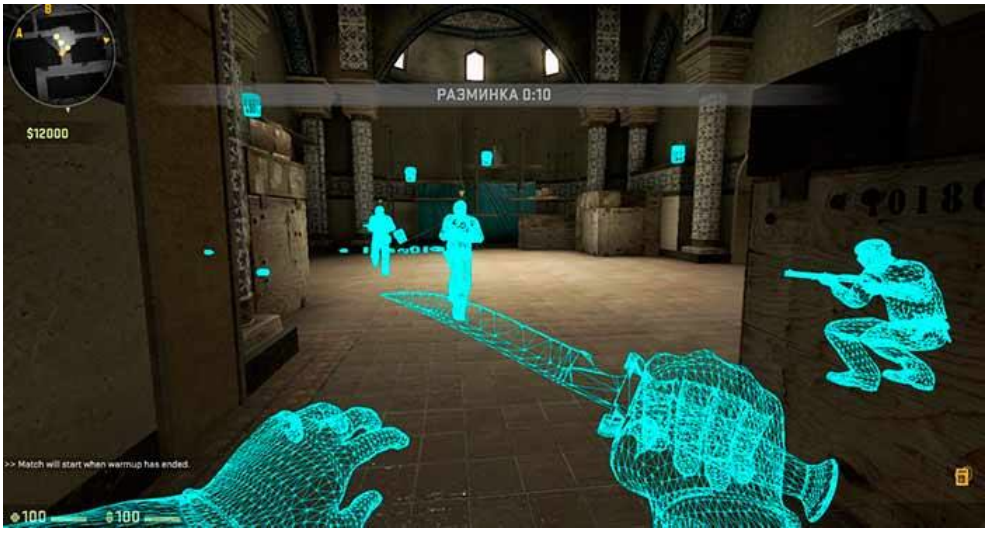
Jelikož WallHack manipuluje s texturami[21] a ZBufferem, tak se aplikuje až v posledních fázích, a to 1.3.7 Fáze Rasterizer a 1.3.8 Fáze Pixel Shader, ještě předtím, než se vloží do finální scény.

[18] Například útok navržený pro scénu města plnou lidí nebude fungovat na scéně lesa plné zvířat.

[19] Obvykle na červenou nebo zelenou, podle toho jestli se jedná o spoluhráče nebo protivníka, ale je možné zvolit libovolnou barvu.

[20] ByteWidth se spočítá jako Stride * počet vrcholů.

[21] Textury jsou zpracovávány speciálními shadery, které mapují texturu na vykreslovaný objekt. Volají se ve Fázi Pixel Shader.



Obrázek 1.9: Na obrázku je ukázka útoku WallHack na reálné aplikaci. Je na něm znázorněno, že se postavy a jiné objekty zobrazí i přesto, že mají být skryté za překážkami.

1.5.2 ESP Hack

Extra Sensory Perception Hack je útok velice podobný útoku WallHack, aplikuje se ve stejných fázích Graphics Pipeline a navíc přidá do scény nové objekty, jež musí rovněž projít celou pipeline. Na rozdíl od WallHack ale nemanipuluje s texturou. Stále využívá vykreslovací smyčku pro identifikaci jednotlivých objektů, ale tentokrát je nebude ovlivňovat. Namísto toho se vykreslí okolo objektu rámeček[22] a vypíší se k němu informace, které jinak aplikace uživateli neposkytuje. Jedná se například o vzdálenost objektu nebo kondice protivníka viz obrázek 1.10. Je možné z dat zjistit mnoho dalších informací, jež nemusí nutně souviset pouze s vykreslovaným objektem. Tyto informace se musí buď spočítat, najít v paměti aplikace, nebo jinak odvodit. V případě vzdálenosti se vezmou souřadnice objektu a hráče, podle nich se spočítá přesná vzdálenost. Kondice se většinou hledá v paměti aplikace. Jakmile je taková informace v paměti nalezena, tak se namapuje do zdrojů, které využívá ESP Hack a následně se zobrazí. Tyto změny se zjišťují a aktualizují pro každý vykreslovaný snímek. Podobnost s útokem WallHack je ta, že se tento rámeček s informacemi vykresluje opět úplně do popředí scény a určí tak pozici objektu i když je skrytý.

[22] Jedná se o nový objekt a je možné vykreslit cokoliv. Nejběžnější je ovšem rámeček.

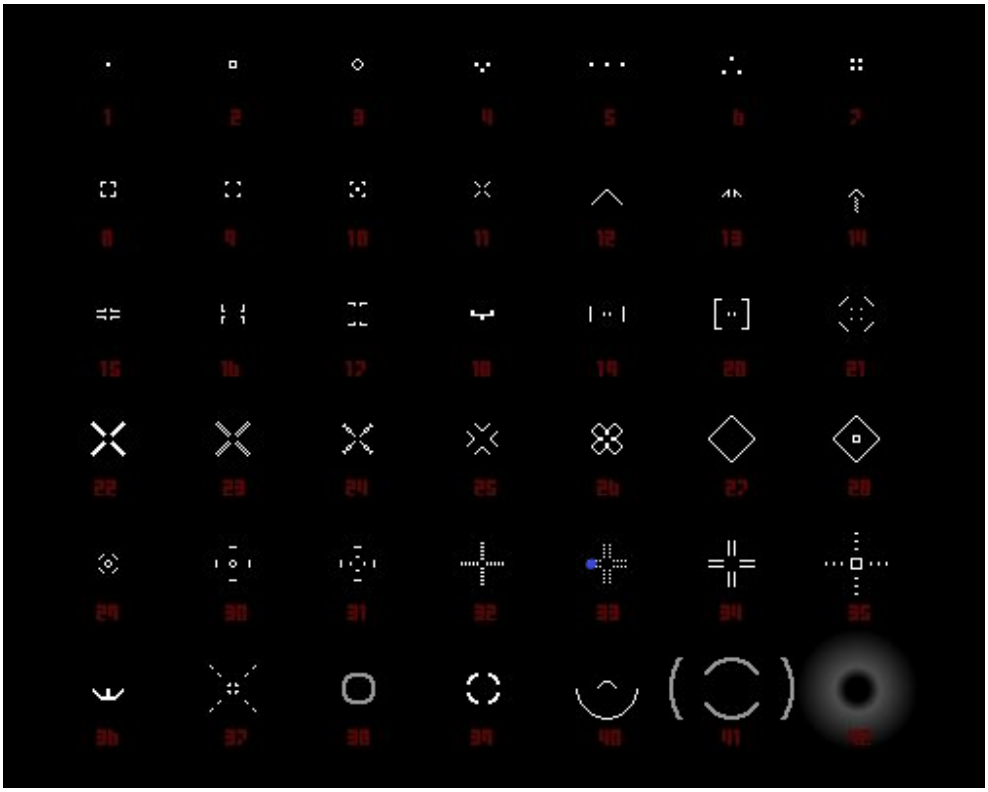
Obrázek 1.10: Na obrázku jsou znázorněny tři způsoby ovlivnění 3D Scény. Růžové plus je ukázka Custom Crosshair. Modrý rámeček a bílý text je vyobrazení možného ESP Hack. Červená tečka v každém rámečku je znázornění útoku AimBot. Obrázek je pořízen z reálné aplikace.

1.5.3 AimBot

Jak již překlad názvu napovídá, jedná se o pomocníka při míření. Tento útok úzce souvisí s předchozím útokem. Na obrázku 1.10 jsou vidět postavy na kterých je aplikovaný ESP Hack, postava se zbraní a ukazatel kam zbraň míří. Na hlavě každé postavy je navíc vidět červený bod. To je místo, kam se bude AimBot snažit mířit. Na to, jak byl tento bod nalezen je jednoduchá odpověď a je vidět i v samotném obrázku 1.10.

Vzhledem k proporcím postavy a způsobu jakým je rámeček vykreslován se odvodí, kde se přibližně nachází hlava postavy, na tu se pak AimBot zaměří. AimBot vlastně nemíří na hlavu protivníka ale do určitého místa v rámečku, takže kdyby vykreslovaný objekt nebyl člověk ale pes, tak ho všechny střely minou[23]. To vysvětluje, jak nalézt bod, kam nasměřovat AimBot. Jediné, co zbývá, je při výstřelu nasměrovat mířidlo na správné místo, a to je přesně to, co AimBot dělá. Při vykreslování se zjistily souřadnice všech červených bodů

[23]Za předpokladu, že se rámeček nezměnil



Obrázek 1.11: Na obrázku jsou vyobrazeny možnosti pro útok Custom Crosshair. Jedná se pouze o malý výběr. Autor útoku může navrhnout svůj vlastní, který bude odpovídat jeho potřebám.

a AimBot následně nastaví kurzor na stejné souřadnice. Takto se lze zaměřit na libovolný bod v 3D scéně. Okamžik, kdy se tento útok aplikuje, záleží na jeho autorovi. Může celý proběhnout spolu s ESP Hack, nebo být rozdělen na dvě části. Nejdříve se zjistí potřebné informace, jako kdyby útok probíhal celý, ale přesun kurzoru se nechá až na později a podmíní se stiskem zvolené klávesy.

1.5.4 Custom Crosshair

Na obrázku 1.10 je vidět růžové plus, značící kam zbraň míří. Díky tomuto útoku se dá tento ukazatel libovolně měnit[24]. Na obrázku 1.11 jsou vidět nějaké varianty, ale konečný vzhled záleží na představivosti autora útoku. Tento útok se může aktivovat buď už při vykreslování ve Fázi Rasterizer, kdy se původní ukazatel vymění úplně, anebo těsně před prohozením FrontBufferu a BackBufferu, kdy se původní ukazatel překryje novým.

[24] Ani zde zobrazené růžové plus není původní

Obrázek 1.12: Na obrázku je zobrazen útok DirectX Overlay. V této implementaci útoku je menu pro ovládání ostatních útoků pro COD 4.

1.5.5 Odstranění stínů

Tento druh útoku se využívá k prosvětlení 3D scény nebo jejích částí. Využití tohoto útoku spočívá v tom, že pomáhá najít a rozeznat objekty schované ve stínech. Kdyby scéna zobrazovala zhasnutou knihovnu, a útočník chtěl najít jednu konkrétní knihu, mohl by do scény umístit světlo, což je nový definovaný objekt pro vykreslení. Tento útok se provádí těsně před prohozením zobrazovaných grafických zásobníků ve SwapChain, a to tak, že se vyrobí objekt světla a vloží se do scény na požadované místo.

1.5.6 DirectX Overlay

DirectX Overlay je jeden z nejdůležitějších útoků, protože se s jeho pomocí tvoří řídící struktura pro všechny ostatní útoky, jak je vidět na obrázku 1.12. Jedná se vlastně o uživatelské rozhraní pro útočníka a umožňuje mu libovolně zapínat a vypínat jednotlivé útoky. Navíc udává přehled o tom, jaké útoky jsou aktivní a jaké ne. Tento útok se aktivuje těsně před výměnou grafických zásobníků ve SwapChain a spočívá v tom, že se vykreslí nové objekty. Musí se namapovat klávesy pro ovládání k tomu, aby bylo toto uživatelské rozhraní interaktivní a dalo se s ním něco ovládat.

1.6 Rozdíl mezi DirectX 9 a DirectX 11

Zaměření na porovnání mezi devátou a jedenáctou verzí má svůj důvod. Tím důvodem je, že výše zmíněné útoky byly v posledních pár letech prováděny

hlavně pro verzi DirectX 9, která je v některých zemích stále aktuální [14].

Důvodem ztráty zpětné kompatibility mezi DirectX 11 s DirectX 9 je dána právě změnou struktury celého procesu. To je důvodem, proč se útoky musí implementovat odlišným způsobem. Stále se provádí ve stejných fázích a využívá stejných principů, ale je nutné použít jiné metody. Nejdůležitější změny pro tuto práci jsou:

- Rozdělení zařízení dvě části (zařízení a jeho kontext).
- Přidání vrstvy DXGI.
- Nahrazení všech fixních funkcí dynamickými a programovatelnými shadery.

Další větší změny jsou přidání podpory pro vícevláknové zpracování a tesselace, ale tyto dvě funkce nejsou pro tuto práci důležité, protože tesselaci nevyužívá a vícevláknové zpracování přímo neovlivní vykreslovací funkcionality [21], [22]. Tím, že se celý proces rozbil do více samostatných částí, se stal efektivnější. Dalším efektem bylo, že je útok na DirectX 11 komplikovanější, protože se musí všechny potřebné funkce lokalizovat v jednotlivých částech.

1.6.1 Zařízení a jeho kontext

Do verze DirectX 10 byly veškeré grafické operace zpravovány pouze objektem zařízení. Navíc se před jeho použitím muselo vytvořit rozhraní pro Direct3D API (Application Programming Interface). V DirectX 11 je zařízení rozděleno na dvě části:

1. Samotné zařízení (ID3D11Device).

   V DirectX 11 se zařízení používá hlavně k poskytnutí zdrojů pro renderování. To znamená, že s jeho pomocí je možné:

   - Nahrávání textur pro grafický procesor (GPU).
   - Vytváření pohledů[25].
   - Vytvoření a nastavení ZBufferu pro jeho testování.
   - Vytváření shaderů.
   - Všechny možnosti zařízení jsou na stránkách MSDN [23].

2. Jeho kontext (ID3D11DeviceContext).

   Kontext se využívá pro nastavování stavů Graphics Pipeline, volání příkazů pro vykreslování 3D scény a jejich jednotlivých objektů a mapování

[25] Takzvané Views, jejich definice je důležitá pro způsob, jakým se bude 3D scéna vykreslovat, protože udává z jakého úhlu se do ní aplikace kouká. Možnost využít pohledy přibyla až v DirectX 11.

zdrojů v grafické paměti vytvořené zařízením. Je to právě kontext, který je stěžejní změnou oproti DirectX 9, protože přebírá všechny funkce zodpovědné za vykreslování [24] a právě tyto metody se budou používat pro ovlivňování celé 3D scény.

Obě tyto části se získají voláním jedné metody (D3D11CreateDevice) a již není nutné vytvářet žádné další pomocné konstrukty k ovládání zařízení [22].

1.6.2 Přidání vrstvy DXGI

Důležitá změna, která přišla s touto vrstvou je, že spravuje SwapChain. ještě v DirectX 9 se o to muselo starat samotné zařízení [25]. Při ovlivňování 3D scény se zneužívá metody Present z DXGI pro poslední úpravy. Tato metoda se volá pouze jednou za snímek[26], na rozdíl od vykreslovacích funkcí. Proto se využívá pro změny, které se nepotřebují provádět opakovaně, jako například vykreslení vlastního menu[27].

1.6.3 Nahrazení fixních funkcí Graphics Pipeline

DirectX 11 pracuje na bázi programovatelných shaderů, které se nahrávají do Graphics Pipeline a nahrazují fixní funkce používané v DirectX 9. To zajišťuje lepší kompatibilitu pro přesouvání na různé stroje, ale opět to znesnadňuje některé útoky, protože veškeré objekty, které nebyly součástí původní aplikace, potřebují svůj vlastní shader, a pokud je aplikace nemá, tak je musí naprogramovat útočník. Dříve stačilo zavolat jednu metodu.

1.7 Vkládání škodlivého kódu do původní aplikace

Teď když jsou známé principy fungování knihoven DirectX a útoků na ně, je možné navrhnout vlastní útoky, které ovlivní danou scénu. Poté co jsou útoky navržené, je potřeba je nějakým způsobem vložit do původní aplikace, aby mohli upravovat výstup programu, podle potřeb útočníka. Nejběžnější a nejpoužívanější způsob je využití technik „VF Table Hooking“ a „DLL injection“ [14], které se využívají i v této práci a jsou detailně popsány níže. Proto se nezabývá jinými možnými řešeními[28].

Operační systém Windows poskytuje knihovny, které tyto metody značně usnadňuje [28]. Těmi knihovnami jsou například „kernel32.dll“ a „Windows.h“, které ulehčují práci s procesy a knihovnami.

Samotné vložení útoku se provádí pomocí upravení tabulky virtuálních funkcí (z originálního „VF Table Hooking“) jednotlivých tříd a jejích ob-

[26]Protože po jejím zavolání se vykreslený snímek zobrazí na obrazovku a další změny jsou prováděny a BackBuffer.

[27]Z kategorie DirectX Overlay.

[28]Například, DLL Proxying, DLL Hijacking, Code Injection a další [26], [27].

jektů[29]. Následně pro aplikování těchto úprav lze využít metody „DLL injection“, která zvolené úpravy provede v aplikaci. Proto je v této práci implementace ovlivňování 3D scény napsaná do dynamické knihovny (Dynamic-link library neboli DLL), která se následně vloží do původní aplikace.

Na závěr této podkapitoly je stručně popsáno, jak se těmto útokům bránit.

1.7.1 Úprava tabulky virtuálních funkcí

Nejdříve práce popisuje postup při úpravě tabulky virtuálních funkcí, neboť její vložení do původní aplikace je až poslední krok. Změny v tabulce virtuálních funkcí spočívají v tom, že se vymění adresy metod zapsané do této tabulky za jiné. Tyto nové funkce by měly odpovídat počtem parametrů a návratovou hodnotou těm původním, protože se musí dbát na původní aplikaci, které je stále připravuje na zásobník jednotlivé parametry pro původní funkci a očekává od ní návratovou hodnotu určitého typu[30]. Pro ovlivňování vykreslování není potřeba znát implementace původních metod, protože se z nových metod volají i ty staré. Takže celé ovlivnění spočívá v tom, že se nahradí funkce starající se o vykreslování, metodou útočníka, ve které změní parametry vykreslovací smyčky popsané v sekci 1.3 Graphics Pipeline Direct3D, vykreslí jiné objekty nebo provedou jakékoliv jiné změny a následně se zavolá původní funkce, aby vše vykreslila.

Tabulku virtuálních funkcí má každá třída a jsou v ní uložené pouze adresy jednotlivých metod dané třídy. Odkaz na tuto tabulku je v paměti instance na prvním místě. Ostatní proměnné jsou uložené až za ní. Všechny instance jedné třídy sdílejí jedinou statickou tabulku virtuálních funkcí, proto když se upraví tato tabulka, tak to ovlivní všechny instance této třídy. Dále je důležité zmínit, že aby se tabulka virtuálních funkcí využila, je potřeba vytvořit instanci a volat její metody dynamicky a ne staticky[31], protože při statickém přístupu je adresa metody známá kompilátoru a tak ji do výsledného binárního souboru doplní přímo.

K jednotlivým funkcím se přistupuje pomocí indexu, proto se musí nejdříve najít indexy vykreslovacích metod. Všechny metody třídy jsou volány s „__thiscall“ konvencí. Její jméno je odvozeno od ukazatele „this“, který se používá k odkazování na zrovna aktivní instanci třídy. Proto je všem metodám přidán jeden pseudo–parametr. Jedná se o ukazatel „this“ uložený do registru ECX v době

[29] V tomto případě se jedná o třídy a objekty knihoven DirectX 11.

[30] Náhradní metody mohou být úplně jiné než ty původní. Protože datový typ se určuje pro kompilátor aby věděl jak hodnotu zpracovat, aplikace přistupuje k surovým datům, které zpracovává podle toho, jak byla přeložena kompilátorem. Proto nová funkce může klidně vracet návratovou hodnotu typu „int“ i když původní vracela „float“, ale i tak s touto vrácenou hodnotou bude aplikace dále pracovat jako s desetinným číslem [29]. Může mít i jiný počet parametrů, ale je potřeba následně uklidit zásobník.

[31] To znamená volat metodu přes šipkovou notaci „instance->metoda()“, což je dynamický přístup. Při statickém přístupu se využívá tečkové notace „instance.metoda()“.

volání funkce[32]. To se zmiňuje proto, že se na to při úpravě tabulek virtuálních funkcí musí brát ohled, aby pak náhradní funkce proběhly správně. V této práci se bude manipulovat s tabulkou virtuálních funkcí tříd:

1. ID3D11DeviceContext

   Tento objekt reprezentuje kontext zařízení a obstarává všechny funkce, které se starají o vykreslování.

2. IDXGISwapChain

   V této třídě jsou metody, které se starají o jednotlivé grafické zásobníky (tedy FrontBuffer a BackBuffer) a jejich prohazování.

Tyto třídy, stejně jako všechny ostatní z knihoven Direct3D, slouží pouze jako obal okolo C–API, a proto jsou všechny metody předělané na funkce s konvencí „_stdcall“, které přijímají instanci jako první parametr [14], což je vidět i v hlavičkových souborech těchto tříd. Proto v této práci není použitá konvence „_thiscall“.

1.7.2 DLL injection

Samotné vložení dynamické knihovny (v originále DLL Injection) sestává z několika kroků:

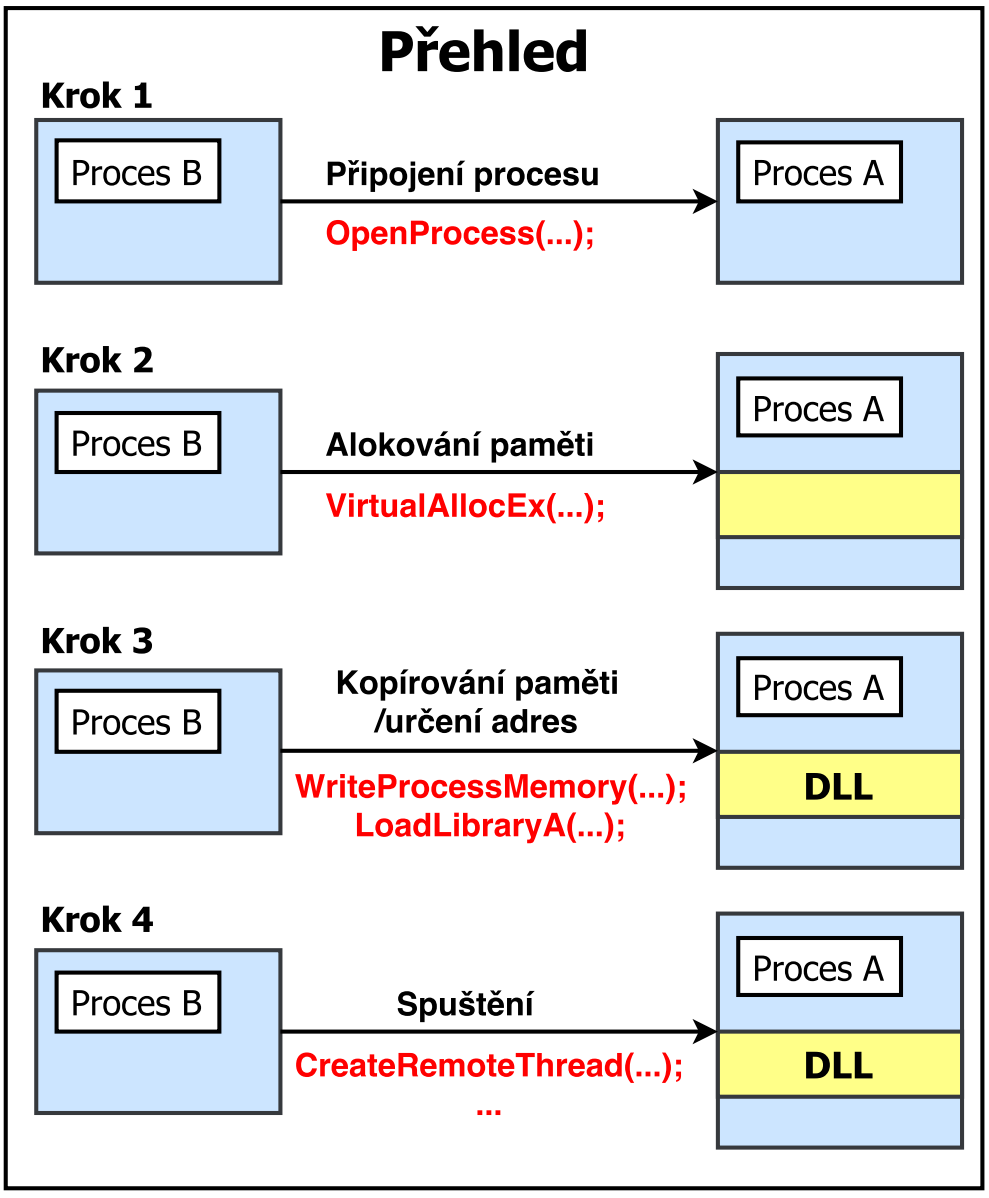
- 1.7.2.1 Připojení procesu útočníka k procesu původní aplikace (str. 25).
- 1.7.2.2 Rezervování místa v paměti pro vkládanou dynamickou knihovnu (str. 27).
- 1.7.2.3 Zkopírování cizí dynamické knihovny do původní aplikace (str. 27).
- 1.7.2.4 Spuštění vložené dynamické knihovny [27] (str. 27).

Nezbytné kroky jsou znázorněné na obrázku 1.13. Operační systém Windows poskytuje sadu funkcí, které lze při jednotlivých krocích použít.

1.7.2.1 Připojení procesu útočníka

Prvním krokem je zajistit si přístup k procesu původní aplikace. Nejlepší způsob je přistoupit k procesu skrze jeho okno. S využitím názvu okna, jež je vyhledatelný například ve správci úloh operačního systému, se vyrobí identifikátor procesu. Ten se jako jeden z parametrů předá funkci „OpenProcess(...)“ a ta následně vytvoří handler k procesu označeným daným identifikátorem. Handler nám dovolí dále manipulovat s procesem. Režimy, v jakých lze handler vytvořit, jsou vypsány v této knize [30] jedná se základní sadu přístupů jako například čtení, psaní nebo spuštění.

[32]To je jediný rozdíl mezi konvencemi „_thiscall“ a „_stdcall“



Obrázek 1.13: Na obrázku je vyobrazen postup vložení dynamické knihovny do aplikace třetí strany bez přístupu ke zdrojovým kódům.

1.7.2.2 Rezervování místa v paměti

Druhý krok je alokovat paměť pro vkládanou dynamickou knihovnu. Využije se k tomu proces handler získaný výše a využije se ve funkci „VirtualAllocEx(...)“. V této funkci se určuje velikost a režim přístupu, ve kterém je pamět vytvořena. Jelikož se z této paměti bude spouštět dynamická knihovna, musí být nastavena v režimu pro spouštění [28].

1.7.2.3 Zkopírování cizí dynamické knihovny

Poté co je paměť připravena, je potřeba nahrát dynamickou knihovnu. Znovu se k tomu využijí funkce operačního systému Windows. Nejlepší je použít funkci „LoadLibrary(...)“, protože sama vynutí spuštění dynamické knihovny [28]. To znamená, že spustí metodu „DllMain()“, která slouží jako vstupní bod (z anglického EntryPoint) pro dynamickou knihovnu.

1.7.2.4 Spuštění vložené dynamické knihovny

Protože se funkce „DllMain()“ spustí vždy po připojení dynamické knihovny, slouží jako odrazový můstek pro cizí kód. V případě této práce se jedná o aplikování úprav tabulek virtuálních funkcí všech potřebných tříd, a to za použití postupu z části 1.7.1 Úprava tabulky virtuálních funkcí. Vložení dynamické knihovny lze využít pouze k jednorázové akci a je potřeba do ní napsat aktivní kód a ne pouze knihovní funkce. Je to logické, protože původní aplikace v době překladu netušila, že dostane další dynamickou knihovnu. A z toho plyne, že nebude sama volat žádnou metodu, která je ve vložené knihovně obsažena. Funkce „DllMain()“ se skládá z dvou hlavních větví. Jedna se spustí při připojení dynamické knihovny a druhá při odpojení. Vstupní bod „DllMain()“ se spouští v načítacím zámku (z originálního *loader lock*). Jedná se o synchronizační zámek, který používají i jiné metody, jež pracují s moduly nahraných do procesu. Proto funkce „DllMain()“ nesmí být moc složitá, neboť by mohla zablokovat další chod aplikace. Pokud je potřeba provádět rozsáhlejší operace a funkce, je vhodné pro ně vytvořit samostatné vlákno. Pak bude mít vstupní bod možnost se rychle ukončit a zámek co nejdříve odemknout [28].

1.7.3 Jak se bránit útokům tohoto typu

Jelikož se útoky využívající knihovny DirectX aplikují na počítači, nad kterém má uživatel plnou kontrolu, tak není snadné těmto útokům zabránit. Je to proto, že má útočník přístup k paměti aplikace nebo ho může pomocí operačního systému snadno získat. Útočník pak má možnost upravit binární soubor aplikace a obejít tak většinu kontrol. Přesto existuje několik možností, jak aplikaci chránit proti nepovolaným změnám. Prostředky pro ochranu aplikace zde budou jen stručně zmíněny, protože obrana proti těmto útokům není součástí zadání. Podrobnější informace jsou k nalezení zde [31].

Ochrana proti manipulaci s aplikací se nejčastěji provádí pomocí skenování paměti. Tento postup se zakládá na jednoduchém principu. Autorům je známá podoba binárního souboru aplikace, a proto mohou program na počítači uživatele oskenovat a porovnat s výchozí podobou. Pokud se liší, je jasné, že s programem někdo manipuloval. Vývojáři k tomu využívají několik nástrojů, které jsou mnohem komplexnější než obyčejný sken paměti:

- The Warden Toolkit
- The PunkBuster Toolkit
- The ESEA Anti-Cheat Toolkit
- The VAC Toolkit
- A další

Jako ochrana proti změnám virtuálních tabulek funkcí je zabudovaná už v DirectX a jeho funkcích. Tato ochrana spočívá v tom, že se při volání některých funkcí restartují tabulky virtuálních funkcí do výchozí podoby. To znamená, že se do ní opět zapíší adresy původních metod [32].

Kapitola 2

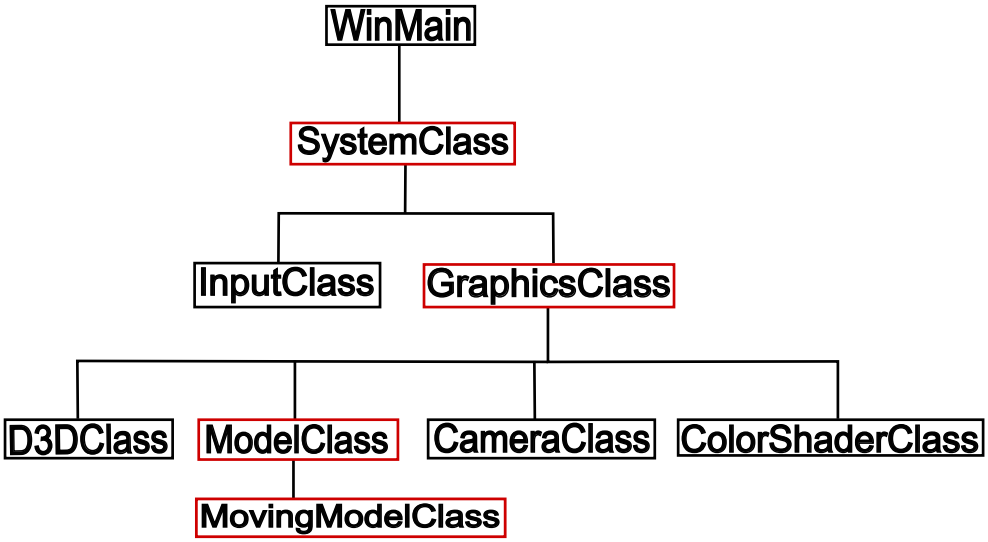
Návrh řešení

V této kapitole se popisuje, jak vypadá návrh řešení. Nejdříve je potřeba naprogramovat vlastní aplikaci, která využívá DirectX 11. Tuto variantu jsem zvolil kvůli autorským právům, neboť na vlastní aplikaci si mohu následně provádět libovolné změny (i ty bez přístupu ke zdrojovému kódu). Návrh grafické aplikace je detailněji popsán v sekci 2.1 Příprava vlastní grafické aplikace. Dalším krokem je navrhnout šablonu pro samotný útok a přizpůsobit jí konkrétní aplikaci. To znamená rozvrhnout dynamickou knihovnu, která se postará o změnu tabulek virtuálních tříd objektů a implementaci nových funkcí, které při tomto útoku nahradí ty stávající. To je popsané v sekci 2.2 Příprava útoku na grafickou aplikaci. Vlastní aplikace i útok jsou navrženy pro 32-bitovou architekturu procesoru. Na konci kapitoly jsou vypsány všechny nástroje a technologie, které jsem pro práci využíval.

2.1 Příprava vlastní grafické aplikace

Na stránkách, odkud jsem čerpal návody pro vytvoření aplikace [33], je popsáno, jak programovat s využitím DirectX 11. Autor těchto návodů u většiny z nich zveřejnil zdrojové kódy nebo zkompilované aplikace. Podle těchto návodů, zdrojových kódů a oficiální dokumentace od firmy Microsoft jsem navrhl a naprogramoval vlastní grafickou aplikaci. Dále jsem tuto předlohu rozšířil o dynamické prvky a umožnil jsem uživateli ovládat část navržené scény.

K vykreslování a prezentování výsledné scény se používají metody z knihoven Direct3D a DXGI popsané v kapitolách 1.2 Základní objekty Direct3D, 1.3 Graphics Pipeline Direct3D a 1.4 DXGI. Konkrétně se jedná o funkci „Present“ z knihovny DXGI a „DrawIndexed“ z Direct3D, které se budou následně zneužívat k vykreslování objektů útočníka. Tato aplikace byla navržena s ohledem na následující ovlivňování 3D scény. To znamená, že jsem se aplikaci pokusil navrhnout tak, aby se na ní daly ukázat zamýšlené úpravy scény, tedy jak definuje zadání:



Obrázek 2.1: Na tomto schématu je znázorněna architektura aplikace a jak spolu jednotlivé komponenty souvisí. Červeně jsou označeny ty, jejichž kód jsem měnil, aby vyhovoval potřebám této práce.

- 2.2.1 Ovládací prvky útočníka (str. 31)
- 2.2.2 Změna parametru objektu ve scéně (str. 31)
- 2.2.3 Zobrazení skrytých objektů (str. 31)
- 2.2.4 Přidávání objektů do scény (str. 31)
- 2.2.5 Odebírání objektů ze scény (str. 32)
- 2.2.6 Změna pozadí scény (str. 32)

Jednotlivé úpravy jsou popsány níže.

Na schématu 2.1 je znázorněna architektura grafické aplikace podle výše zmíněného návodu. Červeně označené komponenty jsou ty, které jsem modifikoval. Do SystemClass jsem přidal ovládací prvky k pohyblivým částem 3D scény. Pohyblivé modely v původní aplikaci nebyly, a proto jsem je přidal. Metody na jejich nastavení jsem naimplementoval do komponenty MovingModelClass. Jelikož jsem změnil i počet objektů ve scéně, změnila se i komponenta ModelClass. V poslední řadě bylo třeba upravit GraphicsClass, aby uměla zpracovávat více modelů včetně toho pohyblivého. Ostatní komponenty jsou ponechány nezměněné.

2.2 Příprava útoku na grafickou aplikaci

Návrh útoku se skládá z přípravy úprav 3D scény uvedených v zadání této práce. Tyto útoky jsou v této podkapitole blíže popsány a jsou implementovány do dynamické knihovny, která bude vkládána do vlastní aplikace. Spolu s útoky bude v této knihovně umístěna i úprava tabulky virtuálních funkcí, která umožní vložení útoků. Vlastní dynamické knihovny vkládám pomocí programu třetí strany od „GuidedHackin“ [34]. Jedná se o takzvaný „DLL Injector“. Nyní zde popíšu návrh útoků ze zadání.

2.2.1 Ovládací prvky útočníka

Jedná se o menu, které umožní určit, jaké útoky se mají aktivovat a jaké ne. Ovládací prvky jsou řešeny jako útok popsaný v podkapitole 1.5.6 DirectX Overlay. Ovládací prvky nezávisí na vykreslované scéně, a proto fungují[33] na libovolné aplikaci využívající knihovny DirectX 11.

2.2.2 Změna parametru objektu ve scéně

Parametr objektu, který jsem se rozhodl ovlivňovat, je barva, protože je to nejčastěji měněný parametr v reálném použití. Vetšinou je tato změna součástí jiného útoku jako například 1.5.1 WallHack.

2.2.3 Zobrazení skrytých objektů

Scénu jsem navrhl tak, aby některé objekty byly v popředí a jiné v pozadí. Pohyblivý objekt je pak umístěn mezi nimi, aby bylo možné znázornit, že se objekt vykreslí, i když je za překážkou. Scéna je tak uzpůsobená k znázornění útoku WallHack, což znamená odkrytí dosud skrytých objektů.

2.2.4 Přidávání objektů do scény

Navrhl jsem několik obrazců, které budu následně umisťovat do scény. Mezi tyto obrazce patří:

- Pentagram
- Trojúhelník
- Čtverec
- Vyplněná kružnice
- Slunce

[33] Tím myslím, že se zobrazí samotné menu. Útoky namapované na jednotlivé ovládací prvky už fungovat nemusí, protože jednotlivé útoky mohou záviset na konkrétní scéně.

- Přilba

Obrazce mohou měnit barvu dle volby v menu. Nad rámec zadání jsem umožnil těmito obrazci pohybovat. Je třeba zmínit, že do této kategorie se dají zařadit i ovládací prvky, protože výsledné menu je rovněž nově přidaný objekt.

2.2.5 Odebírání objektů ze scény

Vlastní grafická aplikace se skládá z několika objektů, aby bylo možné znázornit odstranění jednoho konkrétního modelu. Nakonec jsem umožnil dokonce odstranit celou scénu bez ohledu na počet prvků a objektů. Je nutno podotknout, že objekty se pouze nezobrazí a v datech samotné aplikace zůstávají nezměněny.

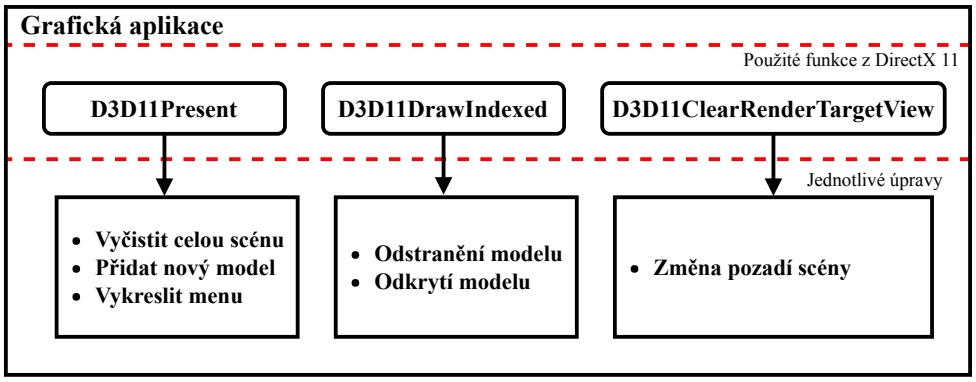
2.2.6 Změna pozadí scény

Rozhodl jsem se rozšířit šablonu a přidat další metody z tabulek virtuálních funkcí. To mi umožnilo změnit pozadí scény. V tomto případě se jedná pouze o předefinování barvy prázdného světa.

2.3 Struktura útoku

Samotný útok se skládá ze dvou částí. První částí je úprava virtuálních tabulek funkcí a druhá je tvorba nových metod, které obstarají úpravy. Obě části útoku se zapisují do dynamické knihovny, která je určená pro vložení do běžící aplikace. Po načtení knihovny do aplikace se provedou všechny změny chování původních metod.

Útoky se zaměřují na pouhé tři metody z knihoven DirectX 11, jak ukazuje schéma 2.2. Tyto tři metody stačí k provedení všech požadovaných změn.



Obrázek 2.2: Na tomto schématu je znázorněno rozložení útoků vůči samotné aplikaci. Jednotlivé úpravy jsou rozdělené podle funkcí, které zneužívají.

Některé úpravy jsou navržené v několika volitelných variantách:

- Vyčistit celou scénu

  Poskytuje pouze jednu variantu, která vše odstraní.

- Přidat nový model

  Umožňuje volit barvy a typ objektu. Barev i typů je omezený výběr.

- Vykreslit menu

  Je pouze jedna fixní varianta menu a obsahuje všechny možnosti.

- Odstranění modelu

  Pominu-li vyčištění celé scény[34], jsou pro tuto úpravu dvě možnosti. To znamená, že se implementuje pro dva samostatné modely.

- Odkrytí modelu

  Tato úprava je rovněž implementována pro dva samostatné modely. Spolu s touto úpravou se mění i barva modelu a ta je opět volitelná.

- Změna pozadí scény

  Pro úpravu pozadí jsou implementované barevné varianty.

2.4 Nástroje použité pro řešení

V této podkapitole jsou vypsány všechny nástroje, rozhraní, knihovny a programovací jazyky, které jsem si vybral pro tuto práci a proč.

2.4.1 Programovací jazyk C++

Tento programovací jazyk jsem si vybral z několika důvodů. Prvním a nejdůležitějším důvodem je, že poskytuje dobrou kompatibilitu s technologiemi DirectX a pohodlný přístup ke knihovnám operačního systému Windows. Všechny ukázky kódu na oficiálních stránkách firmy Microsoft jsou psané právě v C++. Nalezená šablona pro ovlivňování grafických scén je rovněž psána v tomto jazyce. Dále C++ umožňuje objektově orientovaný přístup k řešení, které vyžadují knihovny DirectX.

[34] Protože se řeší jiným přístupen než mazání jednotlivých objektů.

2.4.2 Visual Studio IDE

Visual Studio je vývojové prostředí pro technologie podporované na operačním systému Windows, a to včetně zvoleného C++. Toto Vývojové prostředí usnadňuje prací s knihovnami, které budu v této práci využívat. Pomáhá udržovat strukturu a přehled v rozsáhlejších aplikacích. Jelikož se vlastní aplikace skládá z mnoha komponent, je tento nástroj ideální volbou pro tuto práci. Visual Studio je ke stažení na stránkách firmy Microsoft [35]

2.4.3 Microsoft DirectX SDK (June 2010)

Microsoft DirectX SDK (Software Development Kit) je sada souborů, které obsahují jednotlivé komponenty DirectX. Tuto verzi SDK (June 2010) jsem vybral, protože je to nejaktuálnější verze s podporou pro DirectX 11. Tyto knihovny jsou potřebné pro jakoukoli práci v DirectX v programovacím jazyce C++. Microsoft DirectX SDK (June 2010) je poskytnutý volně ke stažení na stránkách firmy Microsoft [36]. Před startem je potřeba tyto knihovny naimportovat do projektu s řešením.

2.4.4 Šablona pro ovlivňování grafické scény

Pro mou práci jsem využil šablonu pro útoky, které se v praxi nejčastěji používají. Tato šablona mi posloužila pouze jako odrazový můstek, protože jsem se v této práci snažil ukázat principy těchto útoků a možností jak ovlivnit grafickou scénu. V praxi jsou tyto útoky zaměřeny na podvádění v počítačových hrách, a proto šablona neobsahuje všechny potřebné prvky pro tuto práci. To je důvod, proč bylo potřeba šablonu upravit a doprogramovat všechny útoky ze zadání. Původní šablona je dostupná na stránkách [37].

2.4.5 Guided Hacking – DLL Injector

Pro vkládání předem připravené dynamické knihovny do navržené aplikace používám program třetí strany. Jedná se o takzvaný „DLL Injector“, který vkládá dynamickou knihovnu podle postupu popsaném v kapitole 1.7.2 DLL injection. Tento program pochází od [34].

2.4.6 Cheat Engine

Cheat Engine je nástroj pro analýzu paměti aplikace. Využil jsem tuto aplikaci k nalezení souřadnic středu pohyblivého objektu ve scéně. Cheat Engine je volně dostupný na stránkách [38].

Kapitola 3

Realizace

V této Kapitole je popsána samotná realizace a překážky, na které jsem narazil při implementaci. Nejdříve jsem musel navrhnout a naprogramovat vlastní aplikaci, a proto podkapitola 3.1 Vlastní Aplikace (str. 35) popisuje jednotlivé komponenty a funkce, které jsem použil. Podrobnější popis vlastní aplikace je uveden, aby bylo zřejmé, proč se při ovlivňování scény použily konkrétní metody a nastavení. U reálných aplikací je nutné nejdříve provést důkladnou analýzu.

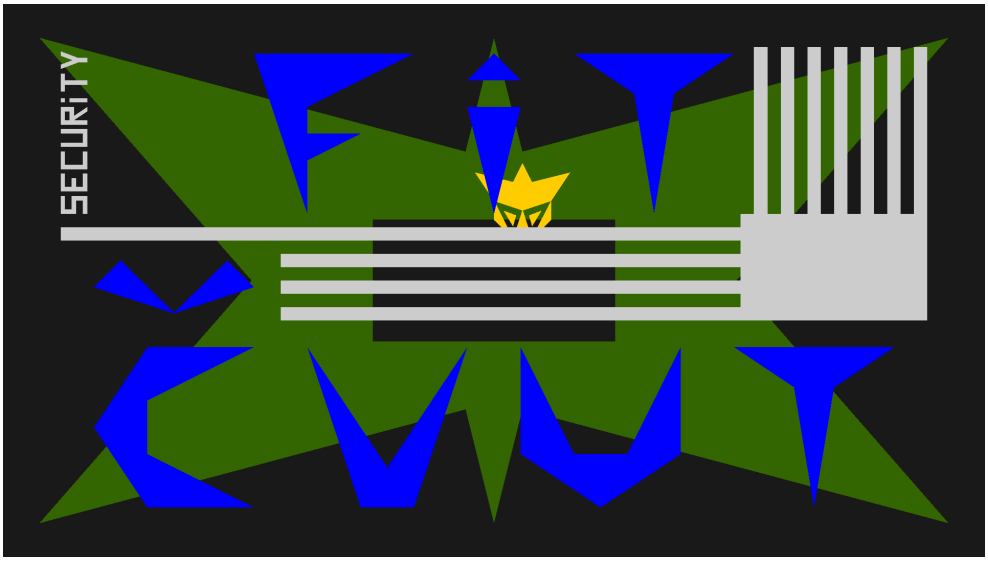
K implementaci útoku jsem využil šablonu. V průběhu řešení jsem několikrát narazil na problém, kdy jsou reálné aplikace komplexnější než moje zvolené ukázkové programy a díky tomu jsem prozkoumal další způsoby, jak pracovat s knihovnami DirectX. Šablonu jsem musel upravit, aby odpovídala zadání práce. Původní šablona obsahovala základ pro reálné aplikace a běžné použití, proto musela být upravena. Realizaci útoků se věnuje druhá část kapitoly a je rozdělena do šesti částí:

- 3.2 Šablona pro ovlivnění scény (str. 42)
- 3.3 Vlastní úprava tabulek virtuálních funkcí (str. 43)
- 3.4 Útoky na vlastní aplikaci (str. 44)

V popisu řešení jednotlivých podkapitol jsou popsány hlavně změny oproti návodu a použitých předloh. Dále se zaměřuji na nejdůležitější části aplikací, u kterých vysvětluji, proč jsem je nezměnil.

3.1 Vlastní Aplikace

Vzhled výsledné aplikace je ukázán na obrázku 3.1. Scéna je složena z několika vrstev. Vyšší vrstvy překrývají ty spodní. Černé pozadí je v nejspodnější vrstvě. V tomto případě se jedná o pomyslnou vrstvu, neboť černá plocha je prázdný svět, do kterého se vkládají navržené objekty. Následující vrstva



Obrázek 3.1: Na obrázku je zobrazena vlastní scéna. Je navržena pro prezentování útoků ze zadání. Objekty jsou rozděleny do tří základních vrstev. Zelená hvězda v pozadí, žlutá „postava“ uprostřed a modrý nápis a šedý obrazec v popředí jako překážka

obsahuje zelenou hvězdu. Je zde umístěna proto, aby bylo zdůrazněno, že se objekty překrývají podle jejich umístění. V další vrstvě je umístěna žlutá „postava“. Jak je na obrázku 3.1 vidět, tato postava překrývá zelenou hvězdu, která je pod ní, ale zároveň je překrytá nápisem z vyšší vrstvy. V této vrstvě je modře napsáno „FIT ČVUT“ a dále je zde znázorněn šedý obrazec s nápisem „SECURITY“[35]. Rozdělení do vrstev je řešeno pomocí osy *z*. Přestože je scéna navržená jako 2D, je umístěna do 3D modelu světa, proto je možné pro překrývání využít osu *z*.

Nyní rozeberu, jak je aplikace řešená na úrovni kódu. Obrázek 2.1 zobrazuje rozložení programu do tříd. Podle tohoto návrhu je popis rozdělen do podkapitol:

- 3.1.1 SystemClass (str. 37)
- 3.1.2 InputClass (str. 37)
- 3.1.3 GraphicsClass (str. 38)
- 3.1.4 D3DClass (str. 38)
- 3.1.5 ModelClass (str. 40)

[35] Jedná se ve skutečnosti o vrstvy dvě, ale na scéně to vidět není, proto to není tak důležité.

- 3.1.6 MovingModelClass (str. 41)
- 3.1.7 CameraClass (str. 41)
- 3.1.8 ColorShaderClass (str. 42)

Na schématu 2.1 jsou znázorněny všechny třídy, ze kterých se aplikace skládá, a jejich návaznost. Červeně jsou označeny ty, jež jsem upravoval. Jednotlivé objekty slouží jako rozhraní pro knihovny DirectX. To znamená, že se pomocí těchto objektů inicializují zobrazovací nastavení grafických knihoven. Přesněji řečeno, jednotlivé komponenty spravují průchod vykreslovací smyčkou a potřebná data. WinMain slouží pouze jako vstupní bod do aplikace, vše ostatní obstarávají jednotlivé komponenty. Jejich úloha a nejdůležitější součásti jsou popsány níže.

3.1.1 SystemClass

SystemClass spojuje dvě hlavní komponenty programu:

1. Vstup od uživatele
2. Vykreslování 3D scény

Využívá InputClass pro získání vstupu od uživatele, který následně předá grafické komponentě GraphicsClass. Ta posune dynamický objekt požadovaným směrem, poté postupně vykreslí jednotlivé obrace do grafického zásobníku a ten na závěr předá monitoru k zobrazení.

3.1.2 InputClass

InputClass se stará o vstup od uživatele. Tato třída pouze indikuje, jaké klávesy jsou stisknuté, a reakce na tyto stisky jsou řešeny v nadřazené třídě SystemClass. Vstup od uživatele je možné přijímat dvojím způsobem:

1. Využít operační systém Windows.

   Funguje na bázi zpráv od operačního systému, které aplikace zpracovává a překládá je na sdělení. Tento přístup je pomalejší než řešení s DirectX.

2. Využít metod DirectX.

   Má stavové pole pro všechny klávesy, do kterého se ukládají informace o tom, jestli je klávesa stisknuta nebo ne.

Vlastní aplikace využívá oba přístupy. Pro ukončení používá zpracování pomocí operačního systému a pohyb dynamických objektů je řešen knihovnami DirectX a třídou InputClass.

3.1.3 GraphicsClass

Komponenta GraphicsClass zajišťuje vykreslování jednotlivých modelů do scény a jejich případné úpravy. Tato třída nekomunikuje přímo s DirectX, ale využívá k tomu dalších komponent ze schématu 2.1. Až tyto komponenty jsou určené pro volání metod DirectX a starají se i o všechny nastavení. Některé parametry (pohyb dynamického objektu) jsou předány příslušné komponentě až za běhu, protože při inicializaci nebyly známé. Zdrojový kód 3.1 ukazuje, v jakém pořadí jsou komponenty volány. Podrobnější popis řešení jednotlivých komponent je popsán v příslušných podkapitolách.

```
1 bool GraphicsClass::Render(){
2   ❶ D3D->BeginScene(0.1f, 0.1f, 0.1f, 1.0f);
3   ❷ Camera->Render();
4
5   ❸ Model->Render(m_D3D->GetDeviceContext());
6   ❹ ColorShader->Render(D3D->GetDeviceContext(), Models[i]->
      GetIndexCount(), worldMatrix, viewMatrix, projectionMatrix);
7
8   ❺ MovingModel->Render(D3D->GetDeviceContext());
9   ColorShader->Render(D3D->GetDeviceContext(), MovingModel->
      GetIndexCount(), worldMatrix, viewMatrix, projectionMatrix);
10
11  ❻ D3D->EndScene();
12 }
```

Zdrojový kód 3.1: Kostra metody, která řídí vykreslování. Jsou zde ukázány volání ostatních komponen potřebných pro vykreslení.

Prvním krokem při vykreslení scény je vyčistit minulou scénu ❶. V parametrech se předávají barvy a průhlednost. Dále se nastavuje kamera ❷. Podle ní se pak vytvoří pohled do scény. Když je takto scéna připravena, umístí se do ní postupně všechny modely. Nejdříve se na správné místo vloží model ❸, a až poté se vybarví pomocí ColorShaderClass ❹. Pro pohyblivý model je postup stejný ❺. V této metodě se objekty pouze vykreslují, proto je změna polohy dynamických objektů řešena zvlášť pomocí SystemClass. Úplně nakonec se zásobník s kompletní scénou pošle k zobrazení na monitor.

3.1.4 D3DClass

Tato komponenta se stará o správné nastavení grafického zásobníku, jež je určený pro zobrazení na monitor. Dále udržuje reference na zařízení a jeho kontext, které se také využívají pro vykreslování a nastavení Graphics Pipeline. Navíc spravuje SwapChain, což znamená, že provádí i výměnu FrontBufferu za BackBuffer. V této souvislosti projdu nejdůležitější součásti této třídy. Mezi tyto části patří:

- Nastavení Zbufferu
- Vyčištění scény

- Prohození grafických zásobníků

Na tyto části se zaměřuji proto, že se později využívají k ovlivnění scény.

ZBuffer se skládá z rozsáhlého množství nastavení, ale já se zaměřil pouze na ty, jež ovlivní vykreslování nejvíce, a které se při útoku budou měnit. Kód 3.2 ukazuje základní a nejdůležitější nastavení ZBufferu.

```
1  ❶ depthStencilDesc.DepthEnable = true;
2  depthStencilDesc.DepthFunc = D3D11_COMPARISON_LESS;
3  ...
4  ❷ device->CreateDepthStencilState(&depthStencilDesc,
5    &depthStencilState);
6  ❸ deviceContext->
7      OMSetDepthStencilState(depthStencilState, 1);
```

Zdrojový kód 3.2: Nejdůležitější prvky pro nastavení ZBufferu. Celé nastavení je dostupné ve zdrojovém kódu v příloze.

Nejdříve je nutné určit vlastnosti, podle kterých bude ZBuffer pracovat ❶. V ukázce je znázorněno nastavení parametrů, které ovlivní, jestli bude ZBuffer aktivní nebo ne a jakým způsobem se porovnávají jednotlivé objekty umístěné do scény. Poté co je nastavení kompletní, se začnou vytvářet jednotlivé části, potřebné k fungování ZBufferu.

Zde si je možné povšimnout, že vytváření jednotlivých součástí obstarává zařízení, a práci s nimi kontext. Pomocí metody ❷ se vytvoří stav ZBufferu, který se následně přiřadí do kontextu zařízení v aplikaci ❸. Z názvu metody je vidět, do jaké fáze vykreslovací smyčky se ZBuffer přiřazuje. Zkratka „OM“ znamená fáze Output–Merger. Stav ZBufferu může být aktivní jen jeden, ale je ho možné měnit, čehož využiji při ovlivňování scény.

Předtím než se začne vykreslovat nový snímek, je potřeba vyčistit ten předchozí. To obstarává metoda 3.3.

```
1 void BeginScene(float red, float green, float blue, float alpha){
2   ❶ float color[4];
3   ❷ deviceContext->
4     ClearRenderTargetView(renderTargetView, color);
5   ❸ deviceContext->ClearDepthStencilView(depthStencilView,
      D3D11_CLEAR_DEPTH, 1.0f, 0);
6 }
```

Zdrojový kód 3.3: Restartování snímku do výchozí podoby.

Funkce dostane v parametrech barvu, jakou má vybarvit výchozí scénu. Tuto barvu dostane po složkách a je nutné ji předat jako celek. Proto se barva složí do čtyř–prvkového pole ❶. Poté se s touto barvou vyčistí snímek ❷, a na závěr se restartuje i ZBuffer ❸.

V poslední řadě se D3DClass využije k prohození grafických zásobníků. Jejich prohozením dojde k zobrazení změn na monitor. Výměna se vyvolá pomocí funkce 3.4 z knihovny DXGI.

```
IDXGISwapChain::Present(UINT SyncInterval, UINT Flags);
```

Zdrojový kód 3.4: Funkce z knihoven DirectX, která složí k prezentování vykreslené scény.

3.1.5 ModelClass

V této komponentě se definují jednotlivé modely. Definice obsahuje i vlastnosti objektů, podle kterých se budou ve fázi útoku identifikovat. Následně je přiřazen do Graphics Pipeline. Obrazec je definovaný pomocí seznamu vrcholů, který je uložen v samostatném souboru. Při inicializaci se tyto modely překopírují do aplikace, kde se s nimi dále pracuje. Nejdůležitější část komponenty ModelClass pro tuto aplikaci je nastavení zásobníku vrcholů. Zde se definují jeho vlastnosti včetně ByteWidth (Stride se definuje později). V kódu 3.5 je vidět nastavení zásobníku pro vlastní aplikaci. Právě tyto informace jsem pak použil k identifikování jednotlivých modelů. Na pořadí příkazů nezáleží.

```
❶ vertexBufferDesc.Usage = D3D11_USAGE_DEFAULT;
❷ vertexBufferDesc.ByteWidth = sizeof(VertexType) * vertexCount;
❸ vertexBufferDesc.BindFlags = D3D11_BIND_VERTEX_BUFFER;
vertexBufferDesc.CPUAccessFlags = 0;
vertexBufferDesc.MiscFlags = 0;
```

Zdrojový kód 3.5: Nastavení zásobníku vrcholů pro statický model.

Tato komponenta definuje statické objekty, proto se musí i zásobník nastavit tak, aby nedovoloval změny po nahrání ❶. Dále se určí, jak velký je model ❷. Pole s vrcholy je přiřazováno do zásobníku vrcholů zvlášť a ByteWidth udává, jak je toto pole velké. VertexType je vlastní proměnná, která definuje jeden vrchol. Je také nutné určit, jaká data jsou na zásobníku uložená, protože se vrcholy indexují a jednotlivé indexy mají svůj vlastní zásobník ❸. Nastavení zásobníku indexů probíhá stejně.

Při vykreslování modelu se musí jeho zásobník přiřadit do vykreslovací smyčky. Přidává se už do fáze Input–Assembler, aby se mohly vrcholy předzpracovat.

```
void RenderBuffers(ID3D11DeviceContext* deviceContext){
  ❶ unsigned int stride = sizeof(VertexType);
  unsigned int offset = 0;

  ❷ deviceContext->IASetVertexBuffers(0, 1, &vertexBuffer,
  &stride, &offset);
  deviceContext-> IASetIndexBuffer(indexBuffer,
    DXGI_FORMAT_R32_UINT, 0);
  ❸ deviceContext->
    IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
}
```

Zdrojový kód 3.6: Přiřazení zásobníku vrcholů a indexů do Draphics Pipeline

Stride udává velikost jednoho vrcholu ❶, a nastavuje se až při přiřazování zásobníku vrcholů do vykreslovací smyčky, jak ukazuje kód 3.6. Stride se liší model od modelu, protože je definovaný i informací o barvě nebo textuře. Pak se přiřadí zásobník vrcholů a zásobník indexů ❷, který určí pořadí, v jakém se budou vrcholy zpracovávat. Vrcholy jsem definoval ve správném pořadí už v souboru, ze kterého se načítaly, proto zásobník indexů obsahuje popořadě čísla od jedné do počtu vrcholů. V poslední řadě se musí určit, podle jaké topologie se budou vrcholy zpracovávat ❸. Zvolil jsem topologii TriangleList kvůli jednoduchosti návrhu modelu, i přestože je efektivnější topologie TriangleStrip.

3.1.6 MovingModelClass

Tato třída byla původně děděna od ModelClass, ale v průběhu práce se návrh změnil, a ve finální podobě je samostatná. Velkou část vlastností má společnou s komponentou ModelClass. Nejdůležitější odlišnost v nastavení je zásobník vrcholů, který je nastaven jako dynamický. To spolu s ovládacími prvky umožňuje měnit polohu objektů této třídy. Pohyb je řízen komponentou GraphicsClass.

Pohyb objetu je řešen pomocí definovaného středu modelu. Střed je bod, kolem kterého se model vykresluje. Pohyb se tak skládá ze dvou částí. Nejdříve se posune střed a následně se něj přemapuje celý model. Přemapování modelu spočívá v tom, že se podle středu přepočítají polohy vrcholů a tyto změny se pak nahrají do zásobníku vrcholů. V kódu 3.7 je ukázáno, jakým způsobem se tyto změny provádějí.

```
1  ❶ deviceContext->Map(vertexBuffer, 0, D3D11_MAP_WRITE_DISCARD,
     0, &resource);
2  ❷ memcpy(resource.pData, vertices,
3    sizeof(VertexType) * vertexCount);
4  ❸ deviceContext->Unmap(vertexBuffer, 0);
```

Zdrojový kód 3.7: Změna polohy pohyblivého modelu. Změna spočívá v tom, že se do paměti zásobníku vrcholů vloží změněné vrcholy.

Nejdříve si musím namapovat zdroje modelu ❶. Poté mohu změnit data tohoto zásobníku ❷. Změna probíhá pomocí kopírování změn. V mém případě kopíruji všechny vrcholy, protože jsem změnil pozici celého modelu. Tuto změnu jsem si připravil předem. Vytvořil jsem si nové pole vrcholů, kam jsem nahrál nové pozice a ty pak kopíruji do původního zásobníku vrcholů. Na závěr se uvolňují zdroje ❸.

3.1.7 CameraClass

Tato třída se stará pouze o umístění kamery do scény. Tato komponenta definuje z jakého úhlu a na kterou část scény se zrovna dívám. To ovlivní, jaká část scény se bude vykreslovat na grafický zásobník.

3.1.8 ColorShaderClass

Tato třída se stará o vykreslování výsledných obrazců, proto se v této třídě nastavují všechny potřebné shadery. Samotné shadery jsou uložené ve vlastních souborech a v ColorShaderClass jsou pouze inicializovány a přiřazeny do vykreslovací smyčky. U této aplikace se jedná o Pixel Shader a Vertex Shader.

```
1 void RenderShader(ID3D11DeviceContext* deviceContext,
2                    int indexCount){
3   ❶ deviceContext->IASetInputLayout(layout);
4   ❷ deviceContext->VSSetShader(vertexShader, NULL, 0);
5   ❸ deviceContext->PSSetShader(pixelShader, NULL, 0);
6   ❹ deviceContext->DrawIndexed(indexCount, 0, 0);
7 }
```

Zdrojový kód 3.8: ColorShaderClass vykresluje obrazce pomocí této funkce.

V momentě, kdy je volaná metoda 3.8, jsou už všechny shadery inicializované a pouze se přiřazují do Graphics Pipeline. Při inicializaci shaderů si DirectX vygenerovalo rozložení vrcholů v paměti, a to se nyní předá fázi Input–Assembler ❶. V dalším kroku je do celého procesu přiřazen Vertex Shader ❷ a Pixel Shader ❸. Když je vše nastavené, spustí se vykreslovací smyčka pro daný objekt ❹. Výsledný obrazec je umístěn do finálního grafického zásobníku. Poté co jsou všechny objekty vykresleny tímto postupem, zobrazí se změny na monitor.

3.2 Šablona pro ovlivnění scény

K realizaci útoku jsem našel šablonu, která byla navržena k identifikování jednotlivých objektů ve scéně a následné doprogramování funkcí se samotným útokem. Pomocí této šablony jsem identifikoval jednotlivé objekty v mé 3D scéně. Při identifikaci jednotlivých objektů ve scéně jsem objevil problémy s kompatibilitou šablony a mojí scény. Předloha útoku byla navržena pro reálné aplikace, které jsou mnohem komplexnější než navržená vlastní scéna. Proto byl další krok odhalit nesrovnalosti v nastavení a přizpůsobit předlohu mým potřebám. Předloha se skládá z těchto komponent:

- Soubor knihoven pro práci s textem[36].
- Úprava tabulky virtuálních funkcí, které jsou běžně používané pro útoky na komerční grafické aplikace.
- Šablona pro vykreslování 2D textur základních tvarů (čtverec, vyplněná kružnice, trojúhelník, apod.).
- Šablona pro menu s ovládacími prvky útočníka.

[36]Protože ve verzi DirecX 11 již nejde zapisovat text přímo.

- Nástroj pro identifikování objektů ve scéně.
- Předloha pro útok ESP Hack s přípravou pro útok AimBot.
- Předloha pro útok WallHack.

Z šablony jsem odstranil nepotřebné metody, zjednodušil některé funkcionality[37] a vložil vlastní útoky dle zadání. Konkrétní útoky jsou vysvětleny v podkapitole 3.4 Útoky na vlastní aplikaci na straně 44. V tabulce jsem nahradil jednu metodu navíc, která mi umožnila větší manipulaci se scénou. Úprava tabulek virtuálních funkcí je popsána v podkapitole 3.3 Vlastní úprava tabulek virtuálních funkcí.

3.3 Vlastní úprava tabulek virtuálních funkcí

Po vložení vlastní dynamické knihovny se vytvoří vlákno, jehož úkolem je změnit tabulky virtuálních funkcí tak, aby odkazovaly na funkce, které implementují jednotlivé útoky. Je nutné změnit více tabulek virtuálních funkcí, protože je v DirectX 11 vykreslování a zobrazování rozděleno do dvou samostatných komponent. Funkce, které chci nahradit, jsou:

1. IDXGISwapChain
   - Present(...)
2. ID3D11DeviceContext
   - ClearRenderTargetView(...)
   - DrawIndexed(...)

Postup úpravy tabulky virtuálních funkcí je nastíněn v ukázce zdrojového kódu 3.9.

```
1    ❶ IDXGISwapChain* pSwapChain;
2    ID3D11DeviceContext *pContext = NULL;
3
4    // Po inicializaci se musí jednotlivé objekty vytvořit. K tomu se využívají rozsáhlé
      nastení a metody DirectX.
5
6    ❷ D3D11CreateDeviceAndSwapChain(..., &pSwapChain,..., &pContext)
7
8    ❸DWORD_PTR* pSwapChainVtable = (DWORD_PTR*)pSwapChain;
9    pSwapChainVtable = (DWORD_PTR*)pSwapChainVtable[0];
10
11   DWORD_PTR* pContextVTable = (DWORD_PTR*)pContext;
12   pContextVTable = (DWORD_PTR*)pContextVTable[0];
13
```

[37] Protože vlastní jednoduchou aplikaci měnili trvale a nevypočitatelně.

```
14    ❹ MH_CreateHook((DWORD_PTR*)pSwapChainVtable[8],
        hookD3D11Present,
15      reinterpret_cast<void**>(&phookD3D11Present))
16    MH_EnableHook((DWORD_PTR*)pSwapChainVtable[8])
17
18    MH_CreateHook((DWORD_PTR*)pContextVTable[12],
        hookD3D11DrawIndexed, reinterpret_cast<void**>(&
        phookD3D11DrawIndexed))
19    MH_EnableHook((DWORD_PTR*)pContextVTable[12])
20
21    MH_CreateHook((DWORD_PTR*)pContextVTable[50],
        hookD3D11ClearRenderTargetView, reinterpret_cast<void**>(&
        phookD3D11ClearRenderTargetView))
22    MH_EnableHook((DWORD_PTR*)pContextVTable[50]
23
24  ❺  pDevice->Release();
25    pContext->Release();
26  }
```

Zdrojový kód 3.9: Kód zobrazuje postup pro nahrazení adress metod v tabulce virtuálních funkcí.

Jelikož všechny objekty odkazují na stejnou tabulku virtuálních funkcí, stačí vyrobit vlastní objekty daných tříd ❶. K inicializaci těchto objektů se využívá funkce DirectX ❷. Z analýzy je známo, že tabulka virtuálních funkcí je na prvním místě v paměti objektu, proto je možné vytvořit odkaz na tuto tabulku ❸. Když mám odkaz na tabulku virtuálních funkcí, mohu nahradit ty správné adresy ❹. Jejich index jsem zjistil z hlavičkových souborů knihoven DirectX. V tabulce jsou zapsány podle pořadí, v jakém byly definované. Při nahrazování adres se ukládají i ty původní adresy funkcí, aby se dala zavolat i ta původní metoda. Když jsou nahrazeny všechny funkce, už nejsou potřeba ty vytvořené objekty, a tak jsou uvolněny ❺.

3.4 Útoky na vlastní aplikaci

Jednotlivé implementované útoky lze rozdělit do kategorií podle zadání a to následovně:

- 3.4.1 Změna barvy (str. 45)
- 2.2.1 Ovládací prvky útočníka (str. 31)
  - 3.4.2 Menu s útoky (str. 46)
- 2.2.2 Změna parametru objektu ve scéně (str. 31)
  - 3.4.3 Přebarvit všechny modely ze scény (str. 47)
- 2.2.3 Zobrazení skrytých objektů (str. 31)

  - 3.4.4 Odkrýt postavu (str. 48)
  - 3.4.5 Odkrýt obrazec v pozadí (str. 49)
- 2.2.4 Přidávání objektů do scény (str. 31)
  - 3.4.6 Ukázka principu ESP Hack (str. 50)
- 2.2.5 Odebírání objektů ze scény (str. 32)
  - 3.4.7 Odstranit postavu nebo obrazec v pozadí (str. 52)
- 2.2.6 Změna pozadí scény (str. 32)
  - 3.4.8 Změna pozadí scény (str. 52)

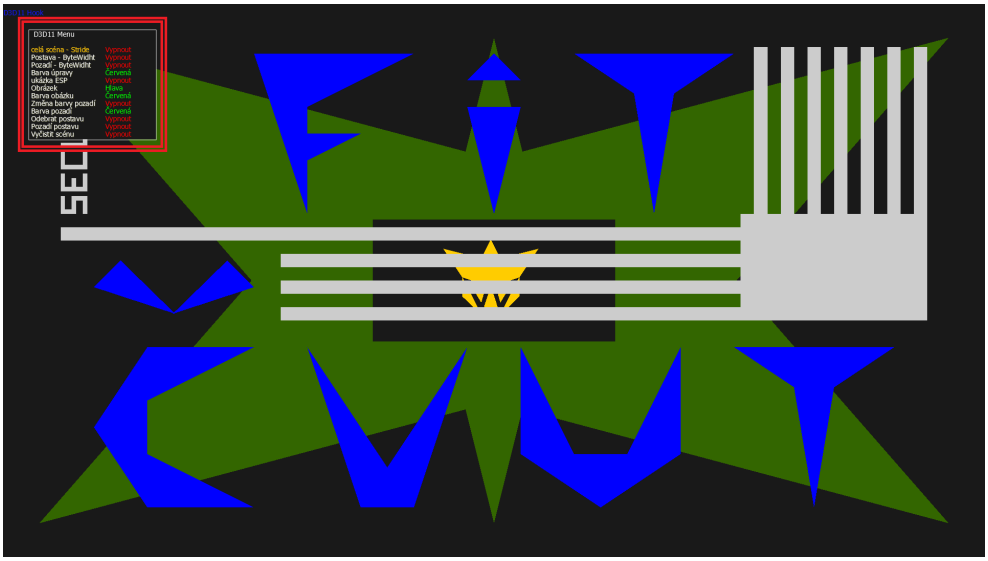
Útoků je více než metod, které se upravovaly v tabulkách virtuálních funkcí, protože se spoustu útoků provádí ve stejné fázi.

Původní šablona definovala svůj vlastní pohled do scény, což bylo nutné pro komplexnější reálné aplikace, ale na jednoduchou vlastní scénu to fungovalo zkázonosným účinkem. Díky tomuto novému pohledu zmizela celá scéna a nebylo ji možné nijak obnovit. Proto byl nový pohled odstraněn a vynechán. Poté vše začalo fungovat jak má.

3.4.1 Změna barvy

Co zde není uvedeno, je změna barvy některých komponent či celých útoků, protože se dá barva nastavit u většiny z nich. Barvy, které jsem povolil, jsou:

- Červená
- Zelená
- Modrá
- Žlutá
- Oranžová
- Růžová
- Fialová
- Kaki
- Hnědá
- Černá
- Bílá



Obrázek 3.2: Na obrázku je zobrazeno červeně označené vlastní menu s ovládacími prvky. Je patrná jeho velikost a umístění. Obsah menu je vidět v detailu na obrázku 3.3.

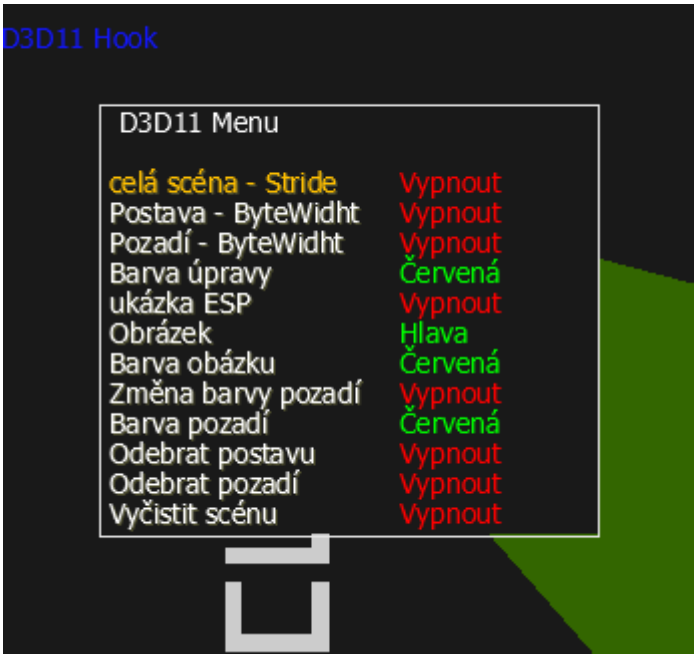
Pro každou barvu se musí vyrobit Pixel Shader, kterým se nahradí původní shader, který se staral o vybarvování pixelů. K vykreslování nových obrazců využívám metody šablony, které jsem rozšířil o pár nových objektů. Funkce původní šablony určují barvu pomocí pole o čtyřech prvcích. Prvky pole jsou základní barevné složky (RGB) a průhlednost. Dále se tyto shadery a barvy přiřazují dle potřeby.

3.4.2 Menu s útoky

Prvním implementovaným útokem je řídící struktura pro ostatní útoky. Ovládací prvky jsou řešeny pomocí menu, které je umístěno v levém horním rohu, jak je vidět na obrázku 3.2. Položky menu jsou řešené pomocí globálního pole. Každá položka menu má svůj typ a název. Podle toho se pak menu vykresluje. Nejdříve se připraví rámeček a ten se pak vyplní názvy položek menu a volbami, které daná položka má. Pomocí aktuálního stavu položek v menu se zapínají a vypínají jednotlivé útoky se zvoleným nastavením. Funkce pro přidání položky je ukázána v kódu 3.10.

```
1 void AddOption(LPCWSTR Name, BYTE menuItemType){
2   ❶ sOptions[Items].Name = (LPCWSTR)Name;
3   ❷ sOptions[Items].Function = 0;
4   ❸ sOptions[Items].Type = menuItemType;
5   Items++;
6 }
```

Zdrojový kód 3.10: Kód zobrazuje funkci pro vložení položky do menu.



Obrázek 3.3: Obrázek znázorňuje detail vlastního ovládacího menu z obrázku 3.2. Je vidět jaké útoky jsou implementovány a jaké jsou možnosti por nastavení útoků.

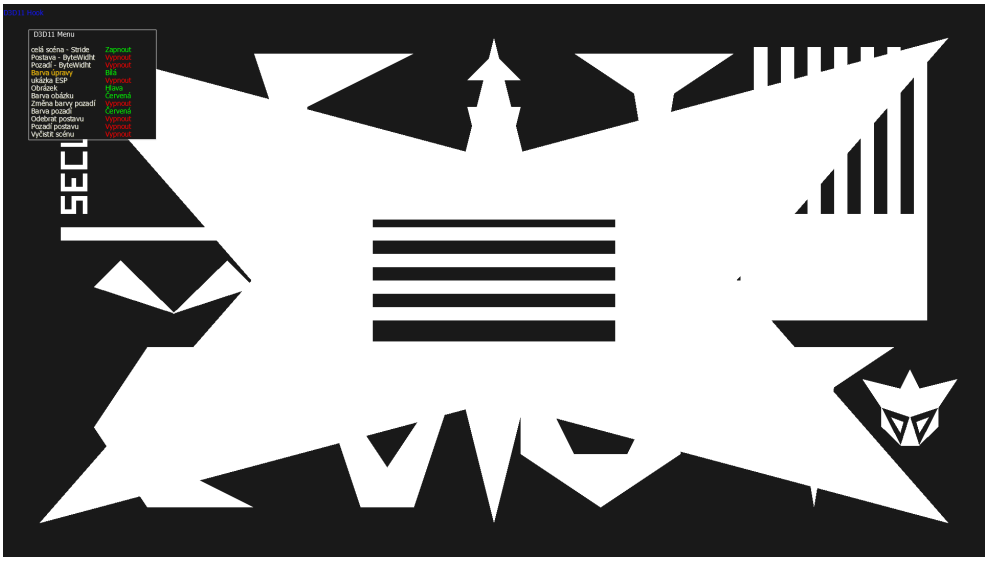
Při vytváření položky menu se musí určit jméno volby do menu ❶. Dále se musí nastavit aktuální volba ❷, ta podle typu určuje, jestli má být útok aktivní nebo v jakém nastavení má být vykreslen. Možnosti, které se dají takto nastavit, se také musí určit ❸. Položky menu se dělí do tří typů:

1. Možnost ano / ne pro aktivaci a deaktivaci útoku
2. Výběr barvy pro daný útok, pokud to podporuje
3. Volba dodatečně vykreslovaného obrazce

Konkrétní volby pro tuto práci jsou vidět na detailu menu na obrázku 3.3 a odpovídají výčtu ze začátku kapitoly. Je rozšířen o možnost změnit barvu.

3.4.3 Přebarvit všechny modely ze scény

V původní šabloně se k identifikaci modelu používal Stride. Při pokusu tímto způsobem identifikovat žlutou postavu ze scény, se označily buď všechny modely, nebo žádný. Proto jsem identifikaci pomocí Stride využil pro přebarvení celé scény, jak je vidět na obrázku 3.4. Scéna byla přebarvena na bílo. Pomocí volby v menu je možné barvu změnit. V kódu 3.11 je znázorněno, jak se provádí změna objektů ve vykreslované scéně. K ukázce je použito odkrytí postavy s



Obrázek 3.4: Obrázek znázorňuje, jak byla scéna změněna, když se nepovedlo jednotlivé modely jednoznačně identifikovat pomocí Stride. Proto se změnila celá scéna. Bílá barva byla zvolena v menu.

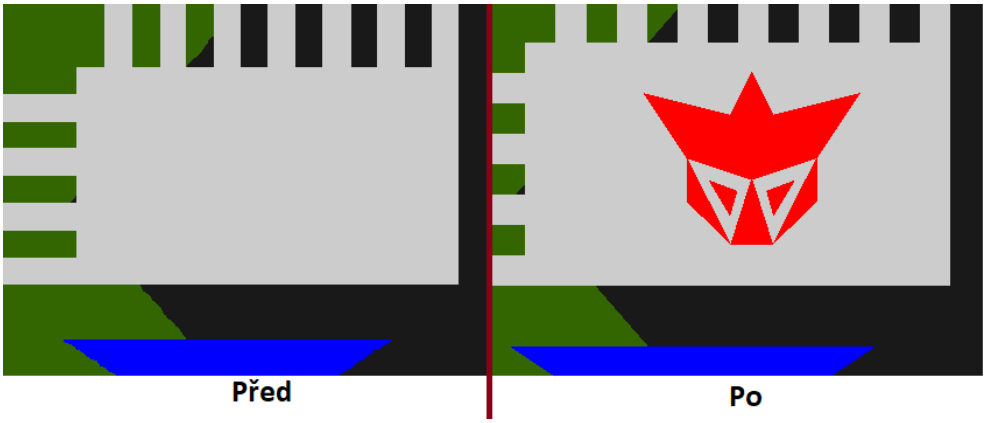
identifikací podle ByteWidth kvůli větší názornosti. Přebarvení scény probíhá stejně, jen některé prvky jako vypínání ZBufferu jsou zbytečné, protože se při přebarvení všech objektů na stejnou barvu neprojeví.

3.4.4 Odkrýt postavu

Poté co jsem zjistil, že identifikace pomocí Stride není dostatečná, jsem použil k určení modelu ByteWidth, což je přesnější způsob určování modelů, protože závisí i na počtu vrcholů. Díky této metodě se mi podařilo identifikovat jednotlivé komponenty scény. Upravoval jsem pouze modely ze spodních vrstev scény, protože se na nich projeví i vypnutí ZBufferu. Ukázka zdrojového kódu 3.11 znázorňuje, jakým způsobem se mění vlastnosti vykreslovaných modelů.

```
1 ❶ if (vedesc.ByteWidth == 924){
2   ❷ if (sOptions[0].Function == 1){
3     ❸ SetDepthStencilState(DISABLED);
4     ❹ setColorOfExisting(pContext, sOptions[3]);
5     ❺ phookD3D11DrawIndexed(pContext, IndexCount,
      StartIndexLocation, BaseVertexLocation);
6     ❻ SetDepthStencilState(ENABLED);
7   }
8 }
```

Zdrojový kód 3.11: Kód zobrazuje, jak se mění vykreslovaný objekt. Nejdříve se identifikuje, a pokud je útok aktivní, tak se provede.



Obrázek 3.5: Na obrázku je vyobrazen vliv odhalení skrytých objektů. Protože v původní aplikaci je postava za obdélníkem schovaná, nevykreslí se. Po Aktivaci útoku se vypne ZBuffer, a proto se postava zobrazí.

Při vykreslování se přidala kontrola na ByteWidth ❶. Pokud vykreslovaný objekt odpovídá nalezenému parametru[38] a pokud je ovlivňování scény zapnuté ❷, tak se vykreslovaný model změní. Provádí se dvě základní změny:

1. Vypnutí ZBufferu ❸.

   Vypíná se proto, aby se vybraný objekt zobrazil úplně nahoře nehledě na překážky.

2. Změna barvy objektu ❹.

   Barva se mění přiřazením jiného shaderu. Já používám jednoduché jednobarevné shadery, ale je možné použít i vlastní texturové shadery.

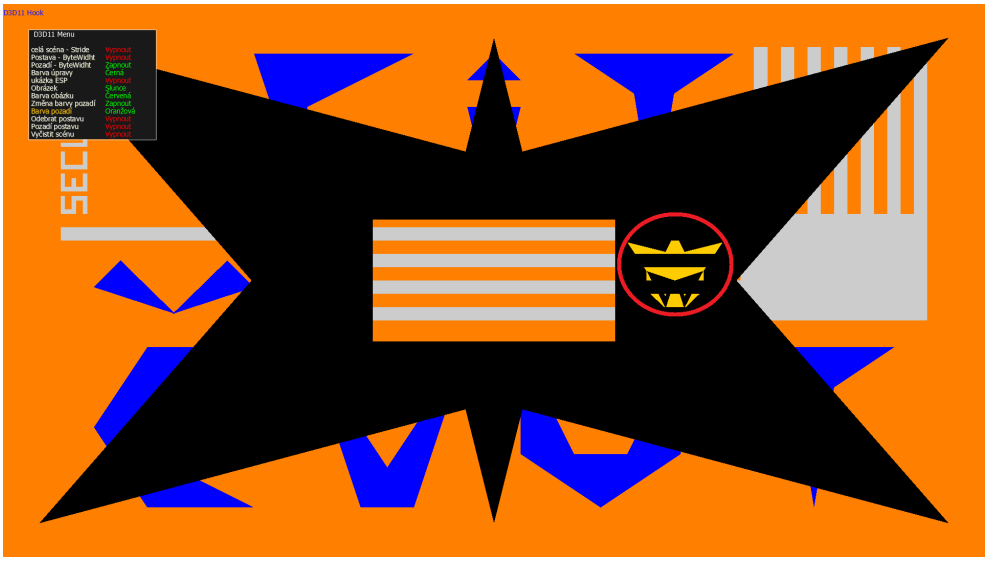
Po změně nastavení se zavolá původní vykreslovací funkce ❺ a vyřeší za mě samotné vykreslení. Na závěr musím opět aktivovat ZBuffer ❻. Pixel Shader se řešit nemusí, protože se nastavuje před každým vykreslením dalších modelů znovu. V této fázi lze měnit pouze textura, barva nebo určit zda se objekt vůbec zobrazí, což je ukázáno v podkapitolách o mazání objektů ze scény.

Výsledná úprava scény je vidět na obrázku 3.5. Pro zobrazení jsem zvolil červenou barvu kvůli kontrastu. Je možné barvu měnit v menu s ovládacími prvky.

3.4.5 Odkrýt obrazec v pozadí

Překreslení pozadí do popředí probíhá stejně jako odhalení postavy. To znamená, že v momentě kdy se vykresluje popředí, překryje všechny objekty, co

[38] Parametr byl nalezen za pomoci původní šablony, a protože se známými hodnotami tento prvek ztratil smysl, byl vymazán.



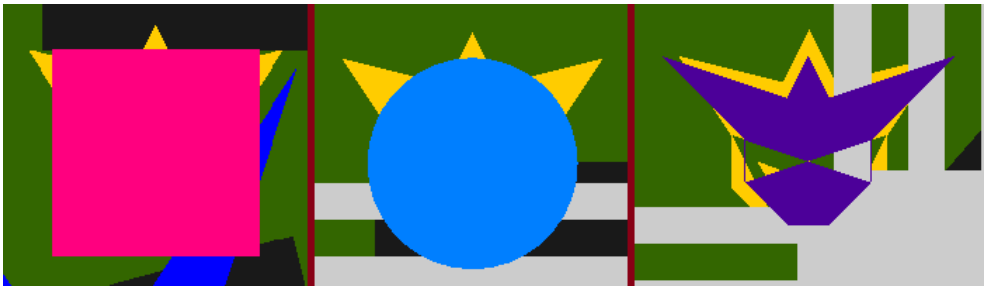
Obrázek 3.6: Obrázek vyobrazuje překreslení modelu v pozadí do popředí. Ukazuje také, že tato změna neovlivní modely vykreslované později. Postava je vykreslena úplně nahoře, protože je opět zapnutý ZBuffer. Pro lepší kontrast byl použit útok pro změnu pozadí.

ve scéně doposud jsou. Ovšem tato úprava se už neprojeví pro modely, které se vykreslují později. To je ukázáno na obrázku 3.6.

Všechny pozdější objekty se vykreslí správně, protože tento způsob ovlivní pouze zobrazení modelu, ale ne jeho vlastnosti a umístění. V aplikaci se nejdříve vykreslí nápisy a až pak hvězda na pozadí. Proto když se vykresluje hvězda a její odkrytí je v menu aktivní, překryje vše, co bylo vykresleno. Novou barvu hvězdy lze měnit v menu s ovládacími prvky. Postava se ale vykresluje až později. Protože je ZBuffer opět aktivní a v aplikaci jsou pozice objektů nezměněny, fungují pravidla pro vykreslování pořád stejně jako v původní aplikaci. Proto je tvar objektů stejný, i když byly překryty pozadím. To je rovněž zobrazeno na obrázku 3.6 a zvýrazněno červeným kruhem. Původní plán byl, aby pozadí překrylo i postavu, ale nepovedlo se mi postup výše obejít, aby se tak stalo.

3.4.6 Ukázka principu ESP Hack

Tento útok přidá do scény vlastní objekt. Možné objekty pro přidání jsou vypsány v návrhu řešení na straně 31. Objekty jsou navrženy a napsány tak, aby sledovaly postavu ve scéně. Ve vlastní aplikaci jsem toho docílil mapováním reálných souřadnic středu postavy do dynamické knihovny. Díky tomu jsem mohl dodatečný objekt vykreslovat přesně na místo, kde se postava nachází. Na obrázku 3.7 jsou ukázány některé z možných obrazců.



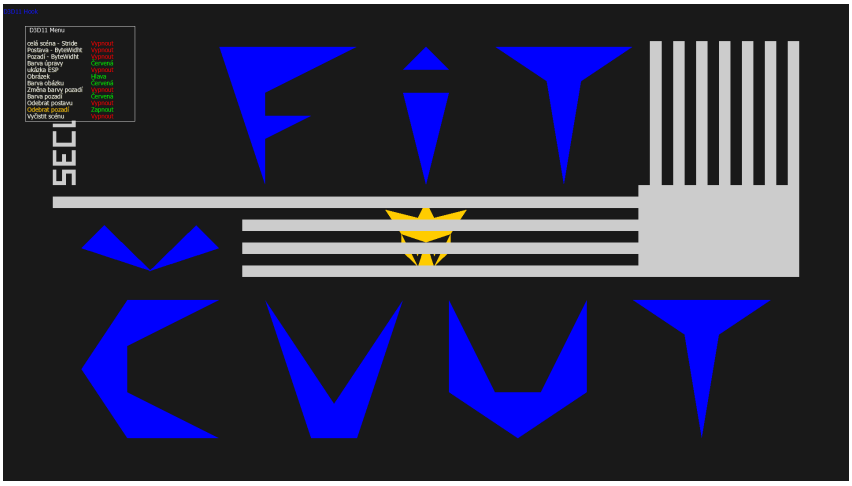
Obrázek 3.7: Na obrázku je vyobrazena ukázka vykreslování vlastních objektů, které sledují postavu. Je možné měnit tvar a barvu nového obrazce.

Nové obrazce překrývají vše v původní scéně, protože jsou přidávány až těsně před zobrazením na monitor, to znamená, že aplikace už nebude nic vykreslovat a tak tyto obrazce nemá co překrýt. Původní šablona obsahovala knihovnu pro vykreslování základních obrazců (například čtverec nebo trojúhelník), kterou jsem rozšířil o své další (slunce, vyplněná kružnice a další). Z těchto základních tvarů jsem následně poskládal přidávané obrazce. Používají se k tomu stejné pravidla, jako používá aplikace k vykreslování scény. V kódu 3.12 je ukázáno, jak jsou mapovány souřadnice pohyblivého modelu.

```
1  ❶DWORD base = (DWORD)GetModuleHandle(NULL);
2  ❷ base = base + 0x3154C;
3  ❸DWORD firstJump = *(DWORD*)base;
4   firstJump += 0x10;
5  DWORD secondJump = *(DWORD*)firstJump;
6   secondJump += 0xC;
7  DWORD thirdJump = *(DWORD*)secondJump;
8
9  ❹ picCoordinateX = (float*)(thirdJump + 0x8);
10  picCoordinateY = (float*)(thirdJump + 0xC);
```

Zdrojový kód 3.12: Kód zobrazuje mapování souřadnic středu pohyblivého modelu.

Nejdříve pomocí funkcí z operačního systému Windows najdu adresu procesu programu ❶. Poté se pomocí počátku odkážu na statickou proměnou ❷, ze které se pak doskáče na reálnou hodnotu souřadnic ❸. Všechny hodnoty skoků a statická počáteční adresa byly nalezeny pomocí Cheat Enginu. Tyto skoky jsou nutné, protože hodnoty souřadnic jsou odkazovány víceúrovňovými ukazateli, což je princip, který má ztížit hledání a následné zneužití dat v paměti programu. Odkaz na počátek proměnné souřadnic je řešený relativně k adrese procesu, protože tak obchází ochranu paměti pomocí takzvaného ASLR (Address Space Layout Randomisation). Na konci řetězce víceúrovňových ukazatelů je ukryta hodnota souřadnic. Odkaz na ni si uložím do svých proměnných ❹, které pak budou vždy odkazovat na správnou hodnotu, i když se poloha objektu v původní aplikaci změní.



Obrázek 3.8: Na obrázku je ukázka odstranění objektu ze scény. Odstraněna byla hvězda v pozadí.

3.4.7 Odstranit postavu nebo obrazec v pozadí

Postup je podobný jako u odkrývání a měnění objektů s tím rozdílem, že se po nalezení objektu přeskočí vykreslování. Odstranění objektu je zobrazeno na obrázku 3.8 a postup ukázán v kódu 3.13. Takto mohu odstranit libovolný nalezený objekt a díky tomu mohu vyčistit i celou scénu.

```
1 ❶ if (vedesc.ByteWidth == 924){
2   if (sOptions[9].Function == 1){
3     ❷ return;
4   }
5 }
```

Zdrojový kód 3.13: Kód zobrazuje, jak vymaže objekt ze scény.

Nejdříve se identifikuje objekt, který se má vymazat ❶, a poté, když je útok aktivní, se funkce ukončí, aniž by se spustilo vykreslování ❷.

3.4.8 Změna pozadí scény

Pozadí scény se na reálné scéně většinou neprojeví, protože bývá zaplněná úplně celá. Ve vlastní aplikaci, kde je jen pár modelů lze tuto úpravu demonstrovat. Pro změnu pozadí scény bylo potřeba upravit metodu, která ale v původní šabloně nebyla[39]. Tato metoda se volá při vykreslování nové scény a nedělá nic jiného, než že vymaže všechny objektu a celou scénu vyplní jednou barvou. Při tomto útoku jsem pouze změnil barvu na nějakou vlastní a zavolal původní metodu s novou barvou. Tato úprava je zobrazena na obrázku 3.6 spolu se změnou parametrů hvězdy.

[39]konkrétně se jedná o metodu „D3D11ClearRenderTargetView“.

Kapitola 4

Testování

V této kapitole je ukázáno, že je postup opravdu univerzální. Povedlo se mi vytvořit útoky pro další dvě aplikace. První z nich byla nalezena na stránkách s návody použitých k vytvoření vlastní grafické scény od RasterTek. Nazval jsem ji Rotující kostky díky scéně, kterou vyobrazuje. Druhou aplikací je pak Stone Giant od společnosti NVIDIA. Je to jedna z aplikací, které slouží k prezentaci možností DirectX 11.

Na každou z těchto aplikací jsem musel navrhnout vlastní šablonu, protože jak bylo zmíněno dříve, má každá aplikace své vlastní definice objektů a dat. Proto bylo nejdříve nutné aplikace zanalyzovat a až poté navrhnout nové podoby útoků. Přesto je zdrojový kód téměř identický s útoky na vlastní aplikaci a nejzásadnější změny v něm jsou identifikátory jednotlivých modelů scény.

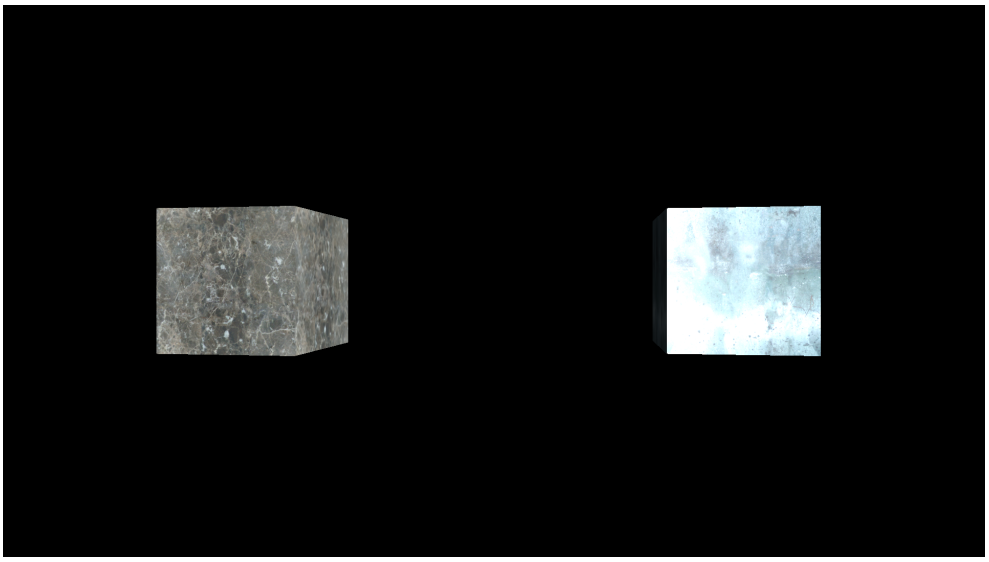
Kvůli podobnosti zdrojových kódů bude v této kapitole ukázán pouze dopad ovlivňování na uvedené aplikace.

4.1 Rotující kostky

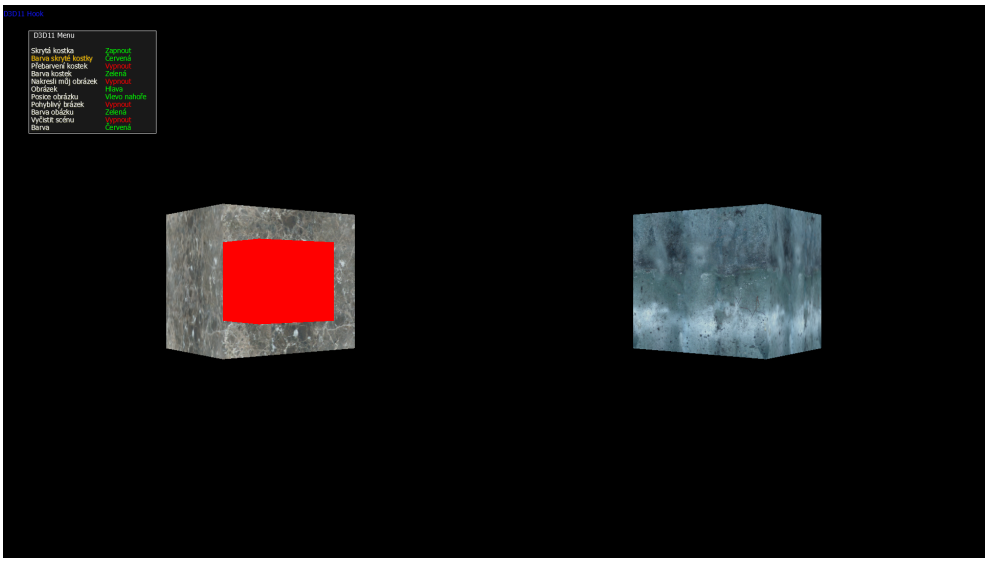
Tato aplikace je velmi podobná jako má vlastní z realizace a útok má stejnou strukturu, která se liší hlavně v identifikaci objektů. Z těchto důvodů se jí v této části nevěnuji podrobně, ale je obsažena v příloze. Základní vzhled je ukázán na obrázku 4.1.

Jediná zvláštnost této scény je, že se ve skutečnosti skládá ze tří kostek a bez mého útoku je možné pouze hádat, kde je poslední kostka skrytá. Za použití útoku pro odhalení skrytých objektů (útok typu WallHack) se tato kostka odkryje, jak je znázorněno na obrázku 4.2.

Ostatní úpravy scény mají téměř stejný průběh jako výchozí aplikace, a proto se jejich názorná ukázka nechává na vlastním zájmu čtenáře, který si je může vyzkoušet sám. Aplikace i implementace útoku jsou přiložené v příloze B Obsah přiloženého CD.



Obrázek 4.1: Na obrázku je vyobrazen vzhled aplikace rotujících kostek.



Obrázek 4.2: Na obrázku je odhalena skrytá kostka ze scény rotujících kostek.

Obrázek 4.3: Na obrázku je zobrazena grafická scéna z aplikace Stone Giant. Skládá se z jednoho obra a několika pavouků umístěných v jeskyni.

4.2 Stone Giant

Stone Giant je profesionální aplikace, která využívá grafické knihovny DirectX 11. Jejím autorem je společnost NVIDIA a vyvinula ji jako demo k prezentování možností a vlastností zmíněných knihoven. I pro tuto aplikaci jsem musel navrhnout vlastní dynamickou knihovnu s útokem. Tuto aplikaci a útoky na ni zde ukážu podrobněji, abych ukázal, že navržené útoky fungují i na komplexních aplikacích s pokročilými grafickými prvky, jak je zobrazeno na obrázku 4.3.

Aplikace má pohyblivou kameru, proto budou jednotlivé úpravy grafické scény zobrazeny z různých úhlů, aby byly co nejlépe znázorněny. Všechny úpravy se odvíjejí od těch na vlastní aplikaci, což je důvod, proč používá stejné barvy pro úpravy a stejný vzhled pro menu možností.

První úprava, kterou ukážu je odhalení obra. Nejdříve je na obrázku 4.4 na straně 56 znázorněn stav před použitím úpravy. Obr je nejdříve částečně skryt za skálou. Po úpravě je obr zobrazen i přes překážku, jak je zobrazeno na obrázku 4.5 na straně 57.

Další ukázka je podobná té předchozí s tím rozdílem, že identifikuje a odhaluje modely pavouků namísto obra. Na obrázku 4.6 na straně 57 je ukázána pouze skála, ale za tou skálou je schovaný pavouk. Po aplikování navrženého útoku pro jeho odhalení se zobrazí, jak je zobrazeno na obrázku 4.7 na straně 58.

V návrhu útoku na aplikaci Stone Giant zůstala i možnost přidat nový objekt. Barva a tvary přidávaných obrazců zůstala původní. I zdrojový kód této části je stejný, protože přidávání není závislé na vlastnostech již existujících

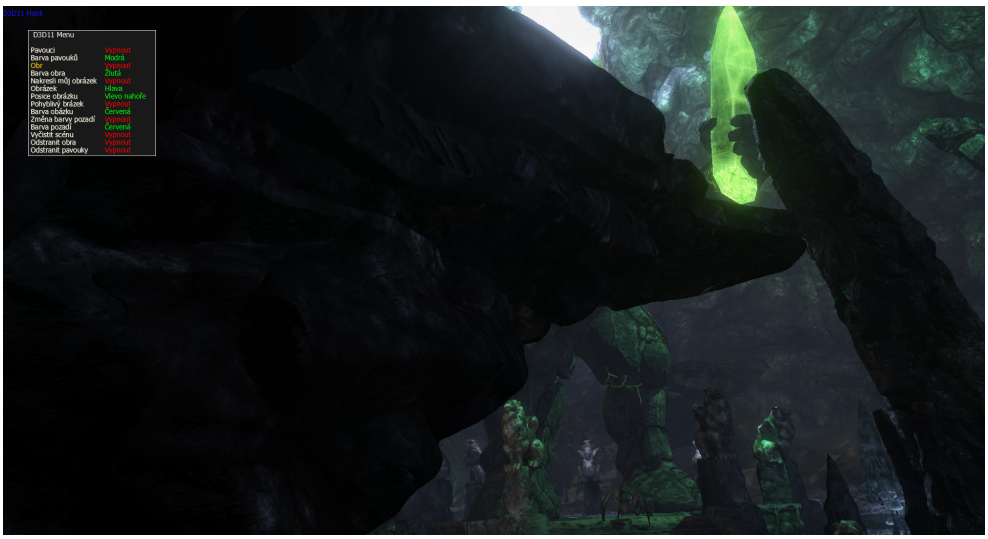
modelů. U obrazce je možné zapnout možnost hýbat obrazcem. Jak vypadá tato úprava ve scéně s obrem, ukazuje obrázek 4.8 na straně 58.

V mojí šabloně je ponechaná i změna barvy pozadí z realizace, ale ta se na scéně s obrem neprojeví, dokud se scéna nevyčistí, protože je celá scéna zaplněná modely. Na obrázku 4.13 na straně 61 je zobrazena scéna po vyčištění, proto na něm nejsou vykresleny žádné obrazce.

Ze scény lze vyjmout identifikované objekty. Nejdříve je ukázáno, jak odstranit obra. Na obrázku 4.9 na straně 59 je ve scéně stále přítomen obr. Po upravě obr zmizí a aplikace se dále chová, jako by ve scéně nikdy neexistoval. Zmizí dokonce i jeho stín. Scéna bez obra je zobrazena na obrázku 4.10 na straně 59.

Vyjmout lze i pavouky. Na obrázku 4.11 na straně 60 je scéna s pavoukem a na obrázku 4.12 na straně 60 je ta samá scéna, ale bez pavouka. Ani po pavoukovi nezůstal ve scéně stín.

Tímto jsem vyčerpal všechny implementované úpravy. Ukázal jsem, že je postup pro ovlivnění scény univerzální a lze jej použít na libovolnou aplikaci, která využívá knihovny DirectX 11. všechny úpravy byly funkční na všech testovaných aplikacích.

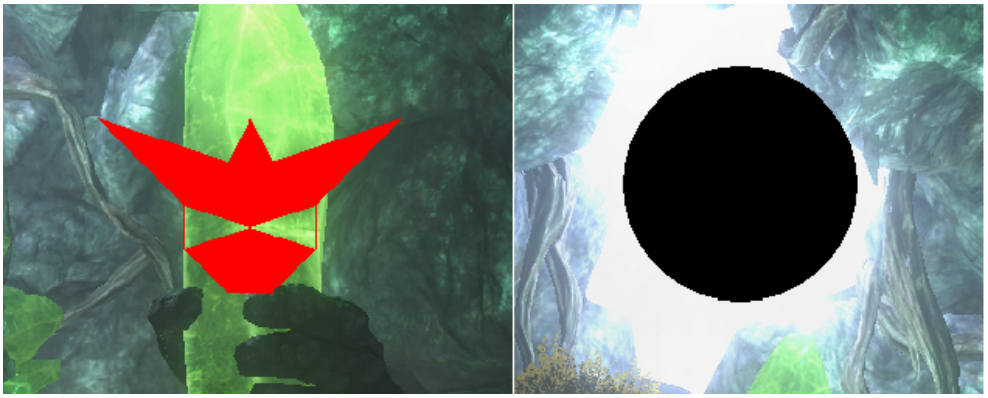


Obrázek 4.4: Na obrázku je vyobrazena scéna s obrem, který je překrytý skálou.

Obrázek 4.5: Na obrázku je zobrazena scéna 4.4, na které je pomocí naimplementovaného útoku zobrazen obr, který byl částečně skryt za skálou.

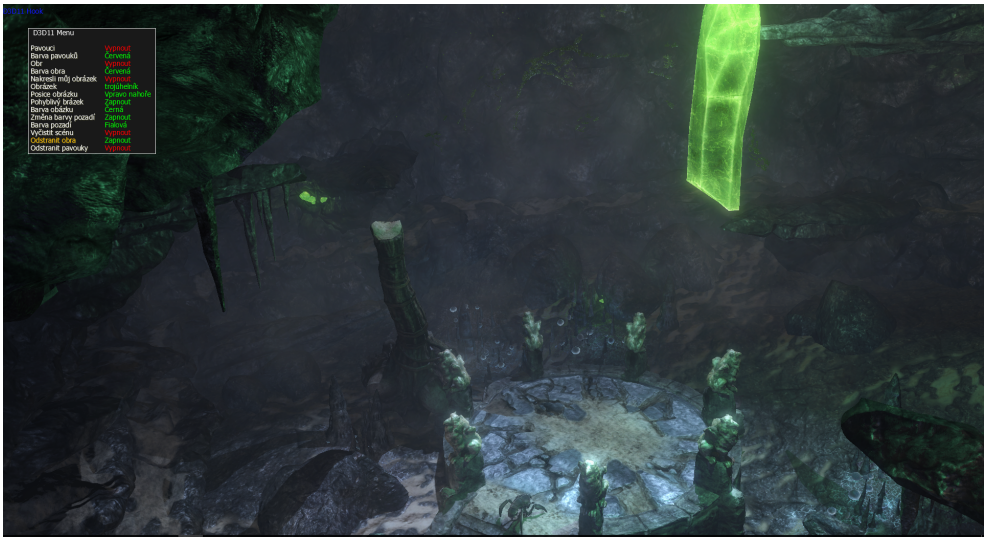
Obrázek 4.6: Na obrázku je vyobrazena scéna se skálou. Za skálou je ovšem schovaný pavouk.

Obrázek 4.7: Na obrázku je zobrazena scéna 4.6, na které jsou pomocí navržených úprav zobrazení pavouci, kteří byli do té doby skryti.



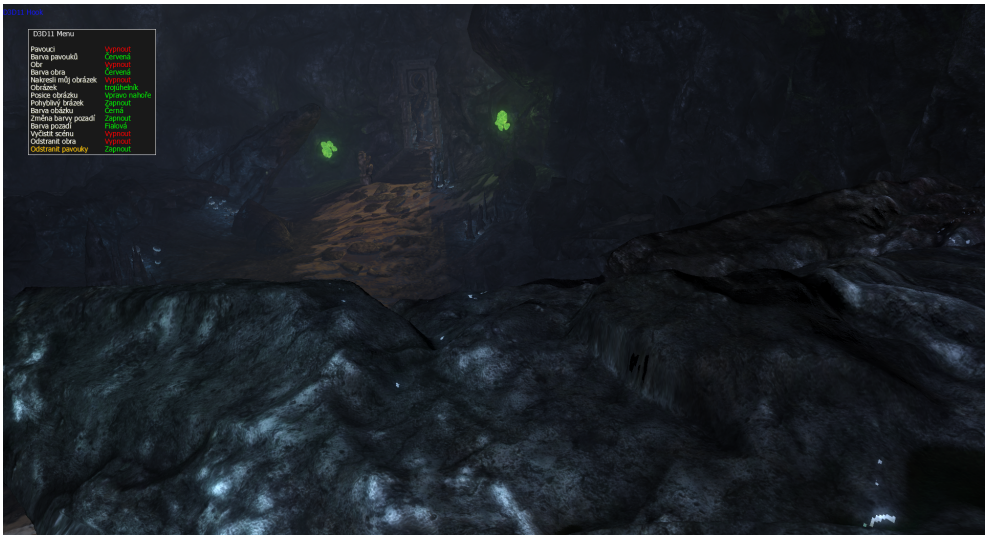
Obrázek 4.8: Na obrázku je zobrazeno, jak jsou do scény přidány a vykresleny vlastní nové objekty.

Obrázek 4.9: Na obrázku je zobrazen obr, který se pomocí změny vykreslovaní ze scény odstraní na obrázku 4.10.



Obrázek 4.10: Na obrázku je zobrazena scéna 4.9, ze které byl odebrán obr. Zajímavý detail k povšimnutí je, že zmizel i obrův stín.

Obrázek 4.11: Na obrázku je zobrazen pavouk, který se pomocí změny vykreslovaní ze scény odstraní na obrázku 4.12.



Obrázek 4.12: Na obrázku je zobrazena scéna 4.11, ze které byl odebrán pavouk. Zajímavý detail k povšimnutí je, že zmizel i pavoukův stín.

Obrázek 4.13: Na obrázku je zobrazena scéna po vyčištění, proto na ní nejsou žádné objekty. Barva nového podkladu záleží na volbě v menu.

Závěr

Na začátku práce jsem stanovil cíle, které se povedlo splnit. Díky pochopení vykreslovacích procesů knihoven DirectX 11 jsem byl schopný navrhnout funkční postup pro ovlivnění grafické aplikace. To znamená, že se mi povedlo provést všechny úpravy definované v zadání jako například odstranění vybraných objektů, změnu jejich vlastností nebo přidání úplně nových. Výsledkem práce je tedy univerzální šablona, která obsahuje tyto základní úpravy. Univerzálnost jsem ověřoval na aplikacích třetí strany, a to konkrétně aplikaci Stone Giant od společnosti NVIDIA, u které jsem neměl přístup ke zdrojovým kódům. Tato aplikace byla navržena pro ukázku fungování grafických knihoven DirectX 11, proto jsem ji využil pro testování univerzálnosti.

Výsledkem práce je také vlastní aplikace. Je složená pouze z 2D modelů, ale ty jsou umístěny do 3D prostoru, proto má výsledná scéna všechny vlastnosti trojrozměrného světa. Aplikace se nakonec skládá ze tří částí. Těmi částmi jsou popředí, pozadí a pohyblivý objekt mezi nimi. Toto rozložení jsem zvolil, aby se daly co nejlépe prezentovat jednotlivé úpravy. Na této aplikaci funguje základní navržená šablona s modifikacemi scény, kterou proto není potřeba dále přizpůsobovat tomuto konkrétnímu programu.

Během práce jsem se odrážel od postupů známých z DirectX 9 a úprav grafické scény, které se nejčastěji používají v reálném prostředí. Odhalil jsem komplexní proces při návrhu grafické aplikace, díky kterému se musí každý program a scéna důkladně zanalyzovat, aby jí bylo možné šablonu náležitě přizpůsobit. Při implementaci jednotlivých úprav jsem například zjistil, že grafická scéna může definovat více pohledů a kontextů, podle kterých se jednotlivé modely nebo scéna vykreslují. To se ukázalo být velmi důležitým faktorem při návrhu úprav, protože špatné nestavení buď nenávratně poškodí scénu, nebo znemožní některé úpravy. Velmi proto záleží na původní aplikaci, protože každá změna ovlivní možnosti úprav scény, s vhodným nastavením je možné některé modifikace znemožnit úplně.

Vlastní testovací aplikace má prostor pro modifikace. Můj zvolený návrh cílil na funkčnost, a proto poskytuje prostor pro estetické rozšíření. Bylo by

například možné modely převést do vyššího třetího rozměru oproti původnímu druhému. Také lze nahradit základní barevné shadery obohatit o různorodé textury. Případně v aplikaci zůstal prostor pro použití pokročilejších funkcionalit grafických knihoven a přidat do scény zdroje světla, které jsou schopné vrhat stíny. Také v mé aplikaci zůstaly nevyužité komponenty pro vyhlazování obrazu za použití tesselace. pomocí všech těchto rozšiřujících by bylo možné navrhnout scénu, která by připomínala reálný svět.

V této práci jsem se zaměřil na nejzákladnější úpravy scény, proto jsem k tomu zneužíval pouze část vykreslovacího procesu. V šabloně se tak mění pouze vlastnosti vykreslování, neboli jinak řečeno vybarvování, jednotlivých objektů, ale už neovlivňuje určení jejich pozice. Proto i zde je prostor pro rozšíření šablony. Je možné napsat šablonu tak komplexní, že vytvoří zcela novou scénu. Je důležité podotknout, že za pomoci zneužití grafických komponent knihoven DirectX 11, je možné změnit pouze výsledný vzhled aplikace. Tudíž vnitřní struktura a data programu zůstanou původní a jeho práce s nimi se nezmění. Je to dáno tím, že se zneužívají pouze funkce vykreslování. To znamená, že se ovlivňuje pouze fungování grafických knihoven, nikoli původní aplikace. Přesto je možné přidat do šablony vlastní strukturu s řídícími prvky a vytvořit tak stínovou aplikaci, která zvládne reagovat i na vstup od uživatele.

Vlastní analýza aplikace byla zdlouhavá a musela být řešena pomocí postupu hrubou silou. Je to dáno tím, že dosud neexistuje žádný nástroj, který by tento proces automatizoval. Při řešení praktické části jsem zjistil, že by tento proces bylo možné do jisté míry zautomatizovat. To je dalším potencionálním rozšířením této práce.

V průběhu sepisování realizace jsem v kódu odhalil spoustu možností, jak celý program i šablonu zpracovat lépe, aby efektivněji pracoval s daty a mohl vypustit některé pomocné konstrukce.

Tímto jsem vyčerpal všechna témata, kterými se práce zabývala. Chtěl bych poděkovat všem čtenářům za to, že se dočetli až sem.

Literatura

[1] Computer Hope: *DirectX [online]*. 15.09.2017, [cit. 07.03.2018]. Dostupné z: https://www.computerhope.com/jargon/d/directx.htm

[2] Microsoft Corporation: *DirectX Graphics and Gaming [online]*. [cit. 07.03.2018]. Dostupné z: https://msdn.microsoft.com/en-us/library/windows/desktop/ee663274(v=vs.85).aspx

[3] Microsoft Corporation: *Graphics Pipeline [online]*. [cit. 07.03.2018]. Dostupné z: https://msdn.microsoft.com/en-us/library/windows/desktop/bb205124(v=vs.85).aspx

[4] DirectXtutorial: Understanding Graphics Concepts. [online], ©2006-2018 DirectXTutorial.com, [cit 07.03.2018]. Dostupné z: http://www.directxtutorial.com/Lesson.aspx?lessonid=11-4-1

[5] Microsoft Corporation: *Getting Started with Direct3D [online]*. [cit. 08.03.2018]. Dostupné z: https://msdn.microsoft.com/en-us/library/windows/desktop/hh769064(v=vs.85).aspx

[6] Microsoft Corporation: *Primitive Topologies [online]*. [cit. 08.03.2018]. Dostupné z: https://msdn.microsoft.com/en-us/library/ff476882.aspx

[7] ResterTek: *Tutorial 4: Buffers, Shaders, and HLSL [online]*. [cit. 10.03.2018]. Dostupné z: http://www.rastertek.com/dx11tut04.html

[8] Microsoft Corporation: *Semantics [online]*. [cit. 14.03.2018]. Dostupné z: https://msdn.microsoft.com/en-us/library/windows/desktop/bb509647(v=vs.85).aspx

[9] Microsoft Corporation: *Vertex Shader Stage[online]*. [cit. 14.03.2018]. Dostupné z: https://msdn.microsoft.com/en-us/library/windows/desktop/mt787172(v=vs.85).aspx

[10] Microsoft Corporation: *Tessellation Stages [online]*. [cit. 08.03.2018]. Dostupné z: https://msdn.microsoft.com/en-us/library/windows/desktop/ff476340(v=vs.85).aspx

[11] Microsoft Corporation: *Geometry Shader Stage [online]*. [cit. 10.03.2018]. Dostupné z: https://msdn.microsoft.com/en-us/library/mt787170.aspx

[12] Microsoft Corporation: *Output-Merger Stage[online]*. [cit. 10.03.2018]. Dostupné z: https://msdn.microsoft.com/en-us/library/bb205120.aspx

[13] Microsoft Corporation: *DXGI Overview [online]*. [cit. 08.03.2018]. Dostupné z: https://msdn.microsoft.com/en-us/library/windows/desktop/bb205075(v=vs.85).aspx

[14] Cano, N.: Applying Jump Hooks and VF Hooks to Direct3D. In *Game Hacking: Developing Autonomous Bots for Online Games*, editace R. M. Laurel Chun, Jennifer Griffith-Delgado, William Pollock, první vydání, 2016, ISBN 1-59327-669-9, s. 175–176, 181.

[15] Fisher, M.: Starcraft 2 Automated Player. [online], Stanford 2014, [cit 18.03.2018]. Dostupné z: http://graphics.stanford.edu/~mdfisher/GameAIs.html

[16] Microsoft Corporation: *Migrating to Direct3D 11[online]*. [cit. 19.03.2018]. Dostupné z: https://msdn.microsoft.com/en-us/library/windows/desktop/ff476190(v=vs.85).aspx

[17] Microsoft Corporation: *Direct3D Architecture (Direct3D 9)[online]*. [cit. 19.03.2018]. Dostupné z: https://msdn.microsoft.com/en-us/library/windows/desktop/bb219679(v=vs.85).aspx

[18] Cano, N.: Using Extrasensory Perception to Ward Off Fog of War. In *Game Hacking: Developing Autonomous Bots for Online Games*, editace R. M. Laurel Chun, Jennifer Griffith-Delgado, William Pollock, první vydání, 2016, ISBN 1-59327-669-9, s. 189–202.

[19] Smith, F.: In-depth: Extravagant cheating via Direct X. [online], 03.04.2012, [cit 18.03.2018]. Dostupné z: https://www.gamasutra.com/view/news/167781/Indepth_Extravagant_cheating_via_Direct_X.php

[20] GuidedHacking: GH Hack Video Tutorials. [online], ©2010 – 2018 XenForo, [cit 18.03.2018]. Dostupné z: https://guidedhacking.com/forums/gh-hack-video-tutorials.426/

[21] Microsoft Corporation: *Direct3D 11 Features[online]*. [cit. 20.03.2018]. Dostupné z: `https://msdn.microsoft.com/en-us/library/windows/desktop/ff476342(v=vs.85).aspx`

[22] Microsoft Corporation: *Migrating to Direct3D 11[online]*. [cit. 20.03.2018]. Dostupné z: `https://docs.microsoft.com/en-us/windows/uwp/gaming/understand-direct3d-11-1-concepts`

[23] Microsoft Corporation: *ID3D11Device interface[online]*. [cit. 21.03.2018]. Dostupné z: `https://msdn.microsoft.com/library/windows/desktop/ff476379`

[24] Microsoft Corporation: *ID3D11DeviceContext interface[online]*. [cit. 21.03.2018]. Dostupné z: `https://msdn.microsoft.com/library/windows/desktop/ff476385`

[25] Microsoft Corporation: *IDirect3DDevice9::Present method[online]*. [cit. 21.03.2018]. Dostupné z: `https://msdn.microsoft.com/en-us/library/windows/desktop/bb174423(v=vs.85).aspx`

[26] Cano, N.: Process Puppeteering. In *Game Hacking: Developing Autonomous Bots for Online Games*, editace R. M. Laurel Chun, Jennifer Griffith-Delgado, William Pollock, první vydání, 2016, ISBN 1-59327-669-9, s. 133–185.

[27] Research, O.: Windows DLL Injection Basics. [online], 08.01.2013, [cit 26.03.2018]. Dostupné z: `http://blog.opensecurityresearch.com/2013/01/windows-dll-injection-basics.html`

[28] Cano, N.: Injecting DLLs for Full Control. In *Game Hacking: Developing Autonomous Bots for Online Games*, editace R. M. Laurel Chun, Jennifer Griffith-Delgado, William Pollock, první vydání, 2016, ISBN 1-59327-669-9, s. 143–144.

[29] Erickson, J.: Programming. In *HACKING: THE ART OF EXPLOITATION*, editace M. D. Christina Samuell, William Pollock, druhé vydání, 2008, ISBN 1-59327-144-1, str. 54.

[30] Cano, N.: Reading from and Writing to Game Memory. In *Game Hacking: Developing Autonomous Bots for Online Games*, editace R. M. Laurel Chun, Jennifer Griffith-Delgado, William Pollock, první vydání, 2016, ISBN 1-59327-669-9, s. 120–122.

[31] Cano, N.: Staying Hidden. In *Game Hacking: Developing Autonomous Bots for Online Games*, editace R. M. Laurel Chun, Jennifer Griffith-Delgado, William Pollock, první vydání, 2016, ISBN 1-59327-669-9, s. 145–165.

[32] Cano, N.: VF Table Hooking. In *Game Hacking: Developing Autonomous Bots for Online Games*, editace R. M. Laurel Chun, Jennifer Griffith-Delgado, William Pollock, první vydání, 2016, ISBN 1-59327-669-9, s. 156–159.

[33] ResterTek: *DirectX 11 Tutorials [online]*. [cit 03.04.2018]. Dostupné z: `http://www.rastertek.com/tutdx11.html`

[34] Broihon: GH DLL Injector V2.4 (Source included). [software], 04.09.2016, [cit 03.04.2018]. Dostupné z: `https://guidedhacking.com/threads/gh-dll-injector-v2-4-source-included.8417/`

[35] Corporation, M.: Visual Studio IDE. [software], © Microsoft 2018, [cit 10.04.2018]. Dostupné z: `https://www.visualstudio.com/`

[36] Corporation, M.: DirectX Software Development Kit. [software], 06. 07. 2010, [cit 10.04.2018]. Dostupné z: `https://www.microsoft.com/en-us/download/details.aspx?id=6812`

[37] DrNseven: D3D11-Worldtoscreen-Finder. [online], 13.01.2018, [cit 03.04.2018]. Dostupné z: `https://github.com/DrNseven/D3D11-Worldtoscreen-Finder`

[38] Heijnen, E.: Cheat Engine. [software], 13.11.2017, [cit 15.04.2018]. Dostupné z: `http://www.cheatengine.org/`

[39] Rybička, J.: *LaTeX pro začátečníky*. Brno: Konvoj, třetí vydání, 2010, ISBN 80-7302-049-1.

[40] Kočička, P.; Blažek, F.: *Praktická typografie*. Brno: Computer Press, 2004.

[41] WWW Consorcium: *Scalable Vector Graphics (SVG) 1.1 Specification [online]*. [cit. 2011-07-07]. Dostupné z: `http://www.w3.org/TR/2003/REC-SVG11-20030114/`

[42] Úřad pro technickou normalizaci, metrologii a státní zkušebnictví: *ČSN ISO 690 Informace a dokumentace – Pravidla pro bibliografické odkazy a citace informačních zdrojů*. 2011.

Příloha **A**

Seznam použitých zkratek

API Application Programming Interface

VF Table tabulka virtuálních funkcí

VF Table Hooking úprava tabulky Virtuálních Funkcí objektů

DLL Dynamic-link library neboli dynamické knihovny

DLL injetion nahrání do aplikace vlastní dynamickou knihovnu

DXGI DirectX Graphics Infrastructure

Primitives základní objekty, které může DirectX vykreslit

PointList základní topologie, zadané vrcholy jsou chápány jako body o velikosti jednoho pixelu

LineList základní topologie, zadané vrcholy jsou chápány jako samostatné čáry

LineStrip základní topologie, zadané vrcholy jsou chápány jako jedna spojená čára

TriangleList základní topologie, zadané vrcholy jsou chápány jako samostatné trojúhelníky

TraingleStrip základní topologie, zadané vrcholy jsou chápány jako trojúhelníky spojené do jedné plochy

Graphics Pipeline proces pro generování grafiky

vykreslovací smyčka proces, kdy se do výsledné scény vykreslí všechny objekty.

Input-Asembler jednotka, která se stará o rozřazení vrcholů do topologii

Shader jednoduchý program, který počítá vykreslování objektů na grafické kartě.

Vertex Shader shader, který zpracovává jednotlivé vrcholy

Pixel Shader shader, který zpracovává jednotlivé pixely k zobrazení.

Tesselace proces, který umožňuje vykreslovat objekty vě větších detailech

Geometry Shader funguje podobně jako Vertex Shader, ale pracuje s více vrcholy

rastrový obrázek obrázek složený z pixelů

Zbuffer mechanika, která se stará o překrytí objektu podle vzdálenosti (osy Z)

Pixel Shader program, který kombinuje informace o vrcholech do každého pixelu.

Depth-Stencil View implementuje ZBuffer

SwapChain fronta grafických zásobníků, které se postupně mění, jeden z nich je zobrazován na monitor

FrontBuffer právě zobrazovaný grafický zásobník

BackBuffer grafický zásobník, do kterého se provádí změny před zobrazením

Index Buffer seznam určující pořadí vrcholů.

Útok na DirectX konkrétní metoda pro ovlivnění vykreslování pomocí DirectX

WallHack útok, který umožňuje zobrazit objekty schované za překážkou

ESP Hack (Extra Sensory Perception) zobrazí jinak skryté informace o hře

DirectX Overlay vykreslení vlastní rozhraní pro ovládání (čehokoliv)

AimBot pomáhá při míření

Custom Crosshair upraví kursor dle vlastní volby

SDK Software Development Kit – sada pro vývoj s určitou technologií

Stride určuje velikost vrcholu

ByteWidth určuje velikost celého modelu

ASLR Address Space Layout Randomisation je nástroj jak chránit paměť počítače

Příloha **B**

Obsah přiloženého CD

Na přiloženém DVD jsou nahrány všechny soubory pro instalaci aplikací a dynamických knihoven z realizace. Struktura souborů je zobrazena na straně 72.

Složka Aplikace obsahuje spustitelné soubory pro všechny použité programy a jejich zdrojové kódy zabalené do projektu z VisualStudia. Aplikace Stone Giant je pouze instalační soubor. Další dva programy jsou FitScena a Rotující kostky.

Složka DirectX-utoky obsahuje implementace jednotlivých útoků pro každou aplikaci. Dále byla přiložena upravená šablona, která byla použita pro inspiraci návrhu jednotlivých útoků. Šablona obsahuje i nástroj pro identifikaci objektů. Každá podsložka patří jednomu útoku a obsahuje projekt do VisualStudia i dynamickou knihovnu již s naimplementovaným útokem.

Složka GHInjector obsahuje nástroj pro vkládání dynamické knihovny do běžícího programu. Je to nástroj třetí strany a je využíván pro vložení útoků do aplikací.

Složka Microsoft DirectX SDK (June 2010) obsahuje knihovny pro vývoj aplikace s podporou DirectX. Jsou k práci přiloženy, protože bez nich není možné aplikaci sestavit.

Na DVD je přiložen instalační soubor DirectX dxwebsetup.exe, aby bylo možné nainstalovat tyto knihovny do systému Windows. Bez nich není možné zkompilovanou aplikaci spustit.

Na DVD je uložen i text této práce a to ve formátu PDF a LaTeX.

- `Aplikace` — implementace Aplikace
  - `FitScena`
    - `Source` — VisualStudio projekt
    - `Spustitelna-aplikace`
      - `data` — data pro modely v aplikaci
      - `shaders` — shadery pro aplikaci
      - `Engine.exe` — aplikace ve formátu exe
  - `RotujiciKrabice`
    - `Source` — VisualStudio projekt
    - `Spustitelna-aplikace`
      - `data` — data pro modely v aplikaci
      - `shaders` — shadery pro aplikaci
      - `Engine.exe` — aplikace ve formátu exe
  - `StoneGiant.exe` — aplikace Stone Giant
- `DirectX-utoky` — implementace útoků
  - `BoxesHack`
    - `Source` — VisualStudio projekt
    - `BoxesHack.dll` — DLL s implementaci utoku
  - `FitSceneHack`
    - `Source` — VisualStudio projekt
    - `FitSceneHack.dll` — DLL s implementaci utoku
  - `StoneGiantHack`
    - `Source` — VisualStudio projekt
    - `StoneGiantHack.dll` — DLL s implementaci utoku
  - `Sablona`
    - `Source` — VisualStudio projekt
    - `Sablona.dll` — DLL s implementaci utoku
- `GHInjector`
  - `GH Injector.exe` — nastroj pro "DLL Injection
- `Microsoft DirectX SDK (June 2010)` — vývojové knihovny
- `dxwebsetup.exe` — instalace DirectX
- `bakalarska-prace.pdf` — text práce ve formátu PDF
- `bakalarska-prace.tex` — text práce ve formátu LaTeX

Příloha C

Manuál

V manuálu je nejdříve popsáno, jak si připravit prostředí, nainstalovat jednotlivé aplikace a sestavit dynamické knihovny. Dále je popsáno, jak potřebné programy ovládat. Přiložené DVD rovněž obsahuje zkompilované verze aplikací a dynamických knihoven z projektů.

C.1 Pracovní prostředí Windows 10

Celá práce je vyvíjena v prostředí operačního systému Windows 10. Tento systém lze použít jako virtuální prostředí ve VirtualBox. Návod jak připravit pracovní prostředí je popsán na stránkách `https://www.wikihow.com/Install-Windows-10-in-VirtualBox`. Přesto bych raději zvolil nainstalovat systém přímo do počítače, protože se bude pracovat s grafickou kartou a některé části řešení potřebují velký výkon. Hlavně se jedná o vývojové rozhraní VisualStudio a aplikaci Stone Giant.

C.2 Příprava VisualStudia a instalace grafických aplikací

Vlastní aplikace, aplikace rotující kostky a všechny dynamické knihovny jsou řešené jako projekt ve VisualStudio 2017. Jak naistalovat VisualStudio je popsáno na stránkách `https://docs.microsoft.com/en-us/visualstudio/install/install-visual-studio`.

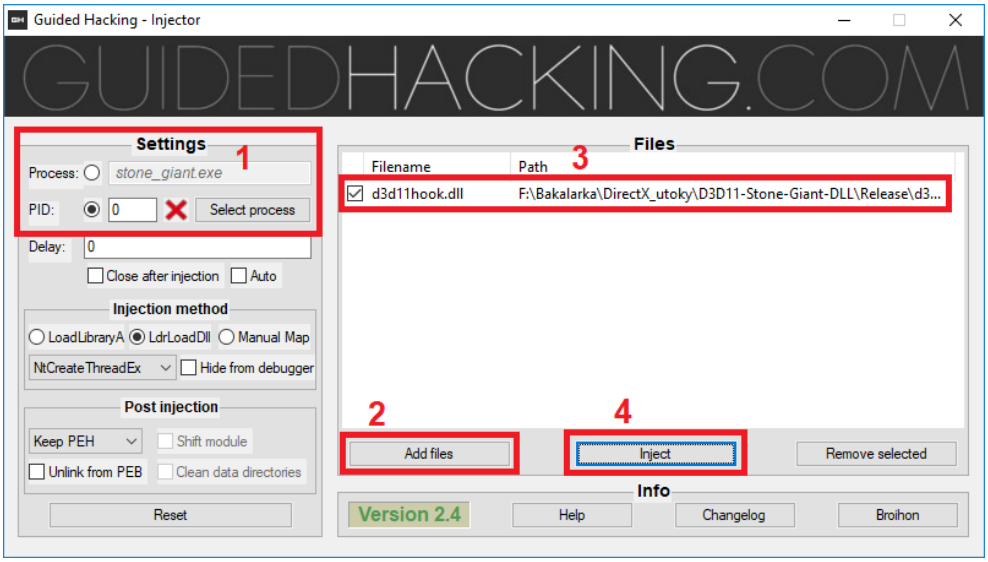
Na stránkách `http://www.rastertek.com/dx11tut01.html`, jak nastavit projekt ve VisualStudiu, tak aby dokázal pracovat s DirectX. Popis je psán pro VisualStudio 2010, ale postup je stejný i pro novější verze. Vkládané SDK, které obsahuje knihovny DirectX je ke stažení na `https://www.microsoft.com/en-us/download/details.aspx?id=6812` nebo jsou všechny knihovny přidány na přiloženém DVD.

Následně stačí projekty jen sestavit. Je důležité projekty sestavovat pro 32-bitový systém, pro který byly navržené. Aplikace Stone Giant se instaluje přímo z instalačního souboru přiloženého na DVD.

C.3 Modul DirectX

Je třeba se ujistit, že je v pracovním prostředí nainstalovaný modul DirectX, jinak se aplikace nespustí. Modul DirectX je ke stažení na stránkách `https://www.microsoft.com/cs-CZ/download/details.aspx?id=35` nebo je opět přiložený na DVD. Je potřeba jej po stažení nainstalovat.

C.4 Vložení dynamické knihovny s úpravami do aplikace



Obrázek C.1: Na obrázku je zobrazena program pro vkládání dynamických knihoven GH Injector s pořadím, v jakém se se nastavuje. Kroky jsou popsané na straně 74.

Nástroj pro vkládání dynamických knihoven se jmenuje GH Injector a je ukázán na obrázku C.1. Znázorňuje postup, jak vložit dynamickou knihovnu. Kroky jsou následující:

1. Nastavení procesu do kterého se budou dynamické knihovny vkládat. Dají se vyhledat i podle názvu programu nejen podle proces ID.

2. Přidání dynamické knihovny do nástroje GH Injector.

3. Ze seznamu přidaných dynamických knihoven zaškrtnout ty, které se mají vložit do aplikace.

4. Tímto tlačítkem se provede vložení dynamických knihoven do aplikace.

C.5 Ovládání grafických aplikací a vložených úprav

Zde bude popsáno, jak se ovládají grafické aplikace a jejich úpravy, když jsou do nich vložené.

C.5.1 FitScena

Pohyb postavy je řešen pomocí kláves „W“, „A“, „S“, „D“ a aplikace se ukončí stiskem klávesy „ESC“.

C.5.2 Rotující kostky

Rotující kostky jsou neinteraktivní scéna, a proto se dá v základu pouze vypnout klávesou „ESC“.

C.5.3 Stone Giant

V aplikaci Stone Giant se pohybuje pouze kamerou. Pohyb po prostoru je řešen klávesami „W“, „A“, „S“, „D“. Pohyb nahoru klávesou „E“ a pohyb dolu klávesou „Q“. Kamerou lze otáčet pomocí myši. Aplikace se ukončuje stiskem klávesy „ESC“.

C.5.4 Ovládání úprav

Po vložení útoku se ovládací prvky rozšíří o „INSERT“ pro ukázání a skrytí menu s ovládacími prvky a o šipky k pohybu v tomto menu.