GYMNASIUM JANA KEPLERA

Parléřova 2/118, 169 00 Praha 6

Grafický Engine

Maturitní práce

Autor: Tadeáš Čejp

Třída: 4.C

Školní rok: 2025/2026

Předmět: Informatika

Vedoucí práce: Igor Vujovič

Praha, 2026

GYMNASIUM JANA KEPLERA

Kabinet informatiky

ZADANÍ MATURITNÍ PRÁCE

Autor:	**Tadeáš Čejp**
Třída:	**4.C**
Školní rok:	**2025/2026**
Vedoucí práce:	**Igor Vujovič**
URL repozitáře	**https://github.com/bagons/graphicengine**

Pokyny pro vypracování:

Podstatou maturitní práce je vytvořit jednoduchý Grafický Engine ve formě C++ knihovny na základě OpenGL a GLFW, který zvládne spravovat entity, načítat soubory typu .obj s podporou různých materiálů, textur atd., dále udržovat systém světel, pro podporu uživatelem napsaných světelných efektů, a zpracovávat uživatelské vstupy z klávesnice i myši.

Prohlášení

Prohlašuji, že jsem svou práci vypracoval samostatně a použil jsem pouze prameny a literaturu uvedené v seznamu bibliografických záznamů. Nemám žádné námitky proti zpřístupňování této práce v souladu se zákonem č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších předpisů.

V Praze dne 1. března 2026 Tadeáš Čejp

Poděkování

Rád bych poděkoval svému vedoucímu práce Mgr. Igoru Vujovičovi, za podporu a jeho ochotné odpovědi na mé dotazy. Dále bych poděkoval svému bratru, který mi v těch nejtěžších chvílích pomáhal vymýšlet vhodné datové struktury a vhodné způsoby správy paměti.

Obsah

Úvod

Tvorba video her mě bavila už odmala. Díky vývoji video her jsem se naučil programovat. S každým dalším projektem, s každou další hrou, jsem chtěl udělat o něco víc, něco o kousek lepšího. Postupoval jsem od *HTML* s *JavaScriptem*, přes velké herní enginy jako *Unity* a *Godot* k *Pythonu* s *pygamem*. Časem jsem vyráběl nejen hry, ale pustil jsem se i do jiných projektů, jako jsou mobilní a webové aplikace, servery, scrapery, sázečky atd..

Kutění bylo pro mě vždycky zdrojem vědění a zábavy. Když jsem se zamýšlel nad tím, co bych mohl udělat na maturitní práci z informatiky, napadlo mě něco, o čem jsem si jako malý myslel, že ani ve snu nedokážu.

Vytvoření game enginu nebo, i zjednodušené verze, **grafického enginu** bylo na internetu vyzdvihované jako něco nemožného, něco co se rozhodně nedoporučuje, něco co bude trvat roky. S plno těmito výroky bych i souhlasil, ale přijde mi, že se z toho dělá větší monstrum, než které to ve skutečnosti je.

Mohl bych se tvářit, že všechno co jsem se předtím naučil, všechny projekty, které jsem udělal, vedly právě k tomuto momentu. K tvorbě grafického enginu. Ale popravdě, řadu let jsem byl jen semi-autonomní dítě, které dělalo, co ho bavilo, a moc nad tím nepřemýšlelo.

Grafický engine není žádná revoluční záležitost, je to takzvaně *vyřešený problém*. Nicméně mou motivací bylo se hlavně něco naučit a proniknout hlouběji do technologií počítačové grafiky, protože můžu. A jsem za to rád, že jsem mohl po dlouhé době dělat projekt bez žádného konkrétního účelu, čistě pro zábavu a znalosti, stejně jako jsem začínal se svými prvními projekty.

1. Teoretická část

1.1 Co je grafický engine?

Grafický engine (*engine, česky motor*), občas nazýván *rendering engine* nebo také *3D engine*, většinou najdeme jako součást videoherního enginu. Je tedy dobré vědět, co je videoherní engine.

1.1.1 *pozn. Co je videoherní engine?*

Videoherní engine je software nebo platforma, která poskytuje základní funkce a nástroje potřebné pro vývoj, provoz a správu videoher [16], je stručná deskriptivní definice, pro dnešní dobu dostačující. První kapitoly knihy *Game Engine Architecture* nahlíží na videoherní enginy i z historického pohledu, tam autor přichází s podobnou, ale obecnější definicí. *Termín „herní engine“ bychom měli vyhradit pro software, který je rozšiřitelný a lze jej použít jako základ pro mnoho různých her bez větších úprav.* [17]

Co to ale v praxi znamená. Při vývoji video her člověk naráží na plno technických problémů. Například: jak vykreslit 3D model na obrazovce, jak zjistím, že uživatel zmáčkl klávesu na klávesnici, co se stane když do sebe narazí 2 míče, jak mohu vědět, že se tak vůbec stalo.

Tyto technické problémy je potřeba vyřešit pro většinu videoher. Ať jde o střílečku z první osoby, z třetí osoby, strategickou hru s pohledem ze shora, puzzle hru nebo RPG, všechny hry se s nimi do nějaké míry setkají. Potřebují umět vykreslovat, zpracovat interakce objektů, uživatelský vstup atd., což jsou právě ty "*základní funkce a nástroje pro vývoj, provoz a správu video her*", které nám engine poskytne.

Většina vývojářů se chce zaměřit na vývoj samotné hry a nechtějí se zaobírat těmito problémy, což dalo za vznik trhu videoherních enginů (s hráčy jako Unreal Engine 5, Unity, Godot, GameMaker), který je k roku 2025 odhadován na 3.43 miliard USD. [12]

S grafickým enginem se setkáme většinou v rámci videoherního enginu. Grafický engine je podstatnou částí herního enginu a zařizuje vykreslovaní: problém jak z čísel popisujících 3D svět dostat obrázek.

Nicméně grafický engine může existovat i samostatně, i když jde spíše o okrajovou záležitost. Jde o software nebo knihovnu, ve které člověk může vytvářet hry nebo vizualizace, ale je ochuzen o druhou podstatnou část plnohodnotného herního enginu a to o **fyzikální engine**, který řeší simulace, nárazy, gravitaci, v podstatě jakékoliv fyzikální interakce.

Grafické enginy jsou zpravidla využívány na hry, které nepotřebují náročné fyzikální interakce, různými vizualizacemi v architektuře nebo matematice, nebo prostě zájemci, kteří si chtějí napsat svůj vlastní fyzikální engine a připojit ho k již existujícímu grafickému enginu.

Plno grafických enginů již existuje, jako *OGRE* [5], *Raylib* [6], *pygame* [7] *Irrlicht* [3]. Stejně tak existují i samostatné fyzikální enginy jako například open-source *Bullet3* [1], ale většina komerčních her je vytvářena v plnohodnotných videoherních enginech.

1.2 Požadavky grafického enginu

Potřebujeme *"základní funkce a nástroje pro vývoj, provoz a správu"grafických projektů*. Nejdůležitější požadavek je samozřejmě **schopnost vykreslovat 3D grafiku**.

Tato 3D grafika by však neměla být omezena na sadu předem definových tvarů, proto bude potřeba **podpora vlastních modelů uživatele**. Dále je třeba nějaký rozumný systém, ve kterém se **definuje rozložení a chování modelů**. V rámci vykreslování by bylo vhodné, mít možnost **modely osvětlit**, pro tvorbu zajímavějších renderů. A nakonec jako rozšíření mít možnost **zpracovávat uživatelské vstupy**, aby se šlo například pohybovat skrz svět.

Z toho nám vyplyne pár další požadavků, kromě samotného vykreslování:

- **načítání modelů ze souborů**
- **správa entit**
- systém světel
- správa uživatelských vstupů

Tučně zvýrazněné jsou pro grafický engine opravdu nezbytné a zbylé dvě jsou velice vhodné, ale teoreticky postradatelné.

Každý požadavek má svou vlastní problematiku, kterou i rozebereme.

1.2.1 *pozn. Jednoduchý grafický engine*

Požadavky jsou rozšiřitelné, lze přidávat více a specifičtějších *nástrojů pro vývoj, provoz a správu grafických projektů*, nicméně vzhledem k náročnosti celého projektu, kterou je i zadání práce ovlivněno, se práce zaměřuje na **jednoduchý grafický engine**.

Jednoduchý, ve smyslu nesofistikovaný, znamená, že při výběru řešení bude jednoduchost, jak konceptu, tak i implementace, faktor, který bude nutný zvážit.

1.3 Vykreslování (Rendering)

Nejprv se zaměřme na samotné vykreslovaní. Klíčová vlastnost grafického enginu na hry je schopnost vytvářet obrázky v reálném čase (tj. 60+ snímků za vteřinu, tedy maximálně 16ms na tvorbu obrázku).

Vykreslovacích metod existuje mnoho: *path-tracing, ray-tracing, ray-marching, rasterizace atd.* Avšak v současné době pro vykreslovaní v reálném čase je stále nejlepší možností rasterizace, protože ostatní metody jsou řádově pomalejší.

Nicméně *rasterizace*, není formálně správný název. Rasterizace je sice klíčový, ale nikoliv jediný krok, v procesu při vytváření obrázku. Metodu, kterou hodlám popsat, lze najít pod termínem *Zobrazovací řetězec*[8], z anglického *Rendering pipeline*.

Avšak zobrazovací řetězec je až příliš obecné pojmenování, jelikož i ostatní metody mají svůj řetězec, který vede k zobrazování. Proto pro účely textu zvolím, za mě, vhodnější termín a to *Rasterizační metoda (vykreslovaní)*.

Rasterizační metoda má pár hlavních částí.

1. Transformace vrcholů
2. Rasterizace
3. Vybarvení fragmentů
4. Dokončení obrázku

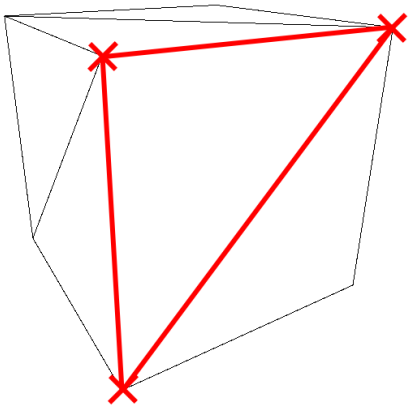
1.3.1 *pozn. Polygonové 3D modely*

Pro pochopení následujících sekcí je důležité znát, jak funguje reprezentace 3D modelů v počítačové grafice. *Víte-li, jak fungují 3D modely, poznámku lze přeskočit.*

Existuje několik druhů reprezentací modelů v počítačové grafice. Nicméně jeden druh reprezentace je utvořen právě pro rasterizační metodu, a jde o polygonovou reprezentaci.

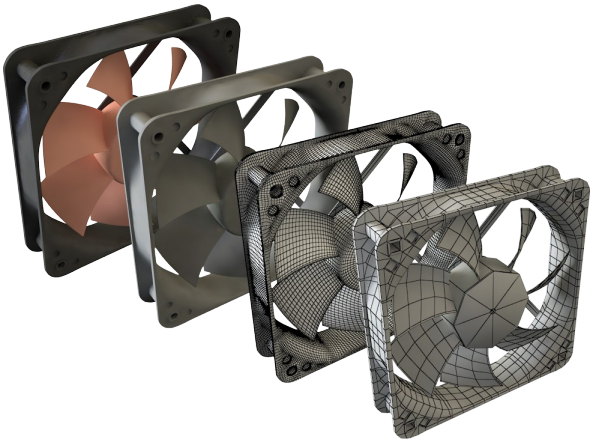
Modely jsou popsány pomocí množiny vrcholů (*bodů*) v prostoru, které jsou seskupeny do trojic tvořící trojúhelníky. Pomocí těchto trojúhelníku jde aproximovat povrchy objektů, a tak tedy modely reprezentovat.

Trojúhelníková aproximace povrchu tvoří polygonovou síť, na kterou jsou aplikovány textury a materiály.



Obrázek 1.1: *Krychle definovaná trojicemi bodů tvořící trojúhelníky, jedna trojice zvýrazněna*

Zrovna u této krychle trojúhelníky perfektně model reprezentují, nicméně někdy model nejde pospat přesně a tvar se jen aproximuje. Například v následujícím modelu větráku jsou zakulacené části jen aproximovány a při dost velkém přiblížení to poznáme, nicméně pro účely modelu to postačí.

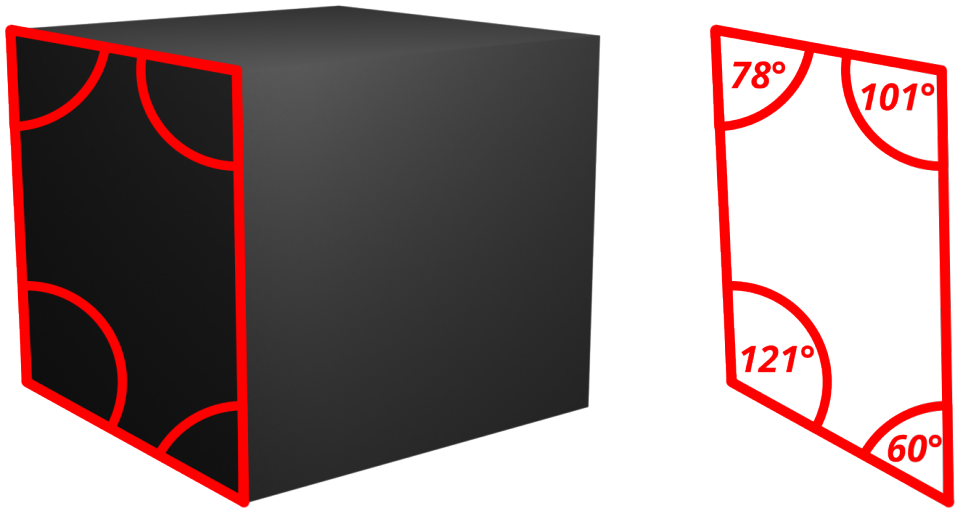


Obrázek 1.2: *Odpředu: polygonová síť, vyhlazená síť, model bez materiálů, model s materiály* [20]

1.3.2 Transformace vrcholů

Když chceme vykreslovat 3D objekty narážíme na fundamentální problém dimenzí. Náš monitor, do kterého kreslíme obrázek, je dvojrozměrný, a objekt, který chceme kreslit (např.: krychle), je trojrozměrný. Budeme muset použít podobné triky jako používají malíři, aby vytvořili iluzi 3D objektu na ploše.

Vezmeme si tuto vykreslenou krychli. Krychle má z definice stěny vytvořené z čtverců. Ovšem když si obkreslíme jednu z těchto stěn, zjistíme, že čtvercová rozhodně není.



Obrázek 1.3: *Krychle přetransformována do plochy [2]*

Právě tímto se zabývá **první část** rasterizační metody, a to transformací jednotlivých vrcholů krychle tak, abychom je mohli přenést na plochu.

Na transformaci vrcholů se nejčastěji používají matice, díky jejich užitečným vlastnostem, viz [27]. V jedné matici můžeme popsat rotaci, posun pozice a změnu velikosti.

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + T_x \\ y + T_y \\ z + T_z \\ 1 \end{pmatrix}$$

$$\begin{bmatrix} S_1 & 0 & 0 & 0 \\ 0 & S_2 & 0 & 0 \\ 0 & 0 & S_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} S_1 \cdot x \\ S_2 \cdot y \\ S_3 \cdot z \\ 1 \end{pmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ \cos\theta \cdot y - \sin\theta \cdot z \\ \sin\theta \cdot y + \cos\theta \cdot z \\ 1 \end{pmatrix}$$

$$\begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta \cdot x + \sin\theta \cdot z \\ y \\ -\sin\theta \cdot x + \cos\theta \cdot z \\ 1 \end{pmatrix}$$

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta \cdot x - \sin\theta \cdot y \\ \sin\theta \cdot x + \cos\theta \cdot y \\ z \\ 1 \end{pmatrix}$$

Obrázek 1.4: *posun, zvětšení a rotace (X, Y, Z) - popsány maticemi [27]*

Při transformaci vrcholů se matice tři používají, *model*, *view* a *projection*. [26] Teoreticky stačí jen jedna, která je výsledkem pronásobení všech třech, nicméně, jak záhy zjistíme, je užitečnější mít je rozdělené.

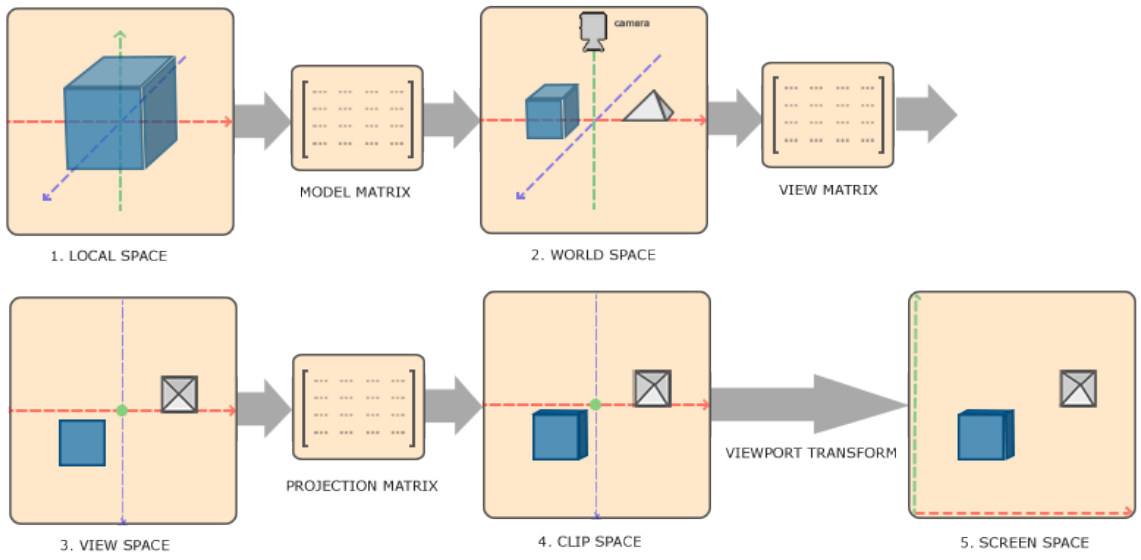
První tzv. *model* matice převede souřadnice vrcholů v modelu (*model space*) na souřadnice vrcholů v 3D prostoru světa (*world space*).

Druhá tzv. *view* matice, převede vrcholy v prostoru světa na relativní souřadnice vůči kameře (*view space*).

Třetí a poslední tzv. *projection* matice udělá projekci bodů na základě vlastností kamery. Například matice určuje zda je kamera perspektivní, nebo ortografická.

Tímto vynásobením dostanme body do tzv. *clip spacu*, kde vrcholy, které půjdou vidět na obrazovce, mají hodnoty X, Y na intervalu $< -1, 1 >$. Body by šly přenést rovnou do plochy, nicméně zatím to neděláme a zachováme i jejich hodnotu Z, která vypovídá o hloubce.

Jelikož jsme v *clip spacu*, v češtině *ořezávacího prostoru*, následuje ořezávání (*clipping*). Ořežou trojúhelníky, které nejsou v zorném poli (*tedy* $|X| > 1 \vee |Y| > 1$). Případně, pokud nějaký trojúhelník přečnívá přes hranu obrázku, body jsou doplněny tak, aby trojúhelník skončil přesně na hraně.



Obrázek 1.5: *transformace vrcholů pomocí model, view a projection matic [26]*

1.3.3 Rasterizace

Následuje rasterizace, proces, při kterém se vektorově definovaná grafika konvertuje na rastrově definované obrazy. [4] Převede nám geometrické trojúhelníky s nekonečně tenkými hranami do rozlišením omezených trojúhelníků.

Výsledkem rasterizace jsou tzv. *fragmenty*.

Fragmenty se často zaměňují s pixely, nicméně není to správně. Fragment je pouze

kandidát na pixel, ještě se bude rozhodovat na základě hloubky, ze kterého fragmentu se stane finální pixel ve výsledném obrázku.

1.3.4 Vybarvení fragmentů

Po vytvoření sady fragmentů je potřeba fragmentny zpracovat, to znamená že je vybarvíme. Většinou nám jde o aplikování textur a o nastínění (*shading*). Toto dobarvení dělají programy, které se nazývají *shadery*, jejichž kód se spustí paralelně na všech fragmentech trojúhelníku. Kód tedy musí být napsán univerzálně, aby fungoval pro každý fragment, bez možnosti komunikace fragmentů mezi sebou.

1.3.5 Dokončení obrázku

Následně jen zbývá porovnat hodnotu hloubky jednotlivých fragmentů, kterou jsme si celou dobu drželi. Vybereme ty fragmenty (*kandidáty na pixel*), které jsou nejblíž kameře a napíšeme je do obrázku.

Celá rasterizační metoda je vymyšlená tak, že většina výpočtů jde paralelizovat a nezávisí na sobě (transformace vrcholů, rasterizace, vybarvení fragmentů), a právě na tuto paralelizaci byly navrženy grafické karty, které mají hodně nezávislých primitivních jader, na rozdíl od CPU.

1.4 Grafické API

Což nás zavádí k grafickému API. Když chceme využít výpočetní síly grafických karet, většinou se používá specifické API, které slouží jako takový most mezi CPU a GPU. Umožní kompilovat stejný kód pro různé grafické karty a obecně nám práci značně zjednoduší.

Grafických API je v dnešní době několik. *OpenGL*, *Vulkan*, *DirectX11*, *DirectX12* a při zvažování výběru je potřeba zhodnotit několik vlastností.

Grafické API *DirectX* jsou proprietární, vyvíjeny společností *Microsoft* a jsou určeny pouze pro *Windows*. Oproti tomu *OpenGL* a *Vulkan* jsou spravovány konsorciem *Khronos Group* a jejich kód je otevřený a podporuje různé operační systémy.

Dále *DirectX11* a *OpenGL* jsou zastaralejší než *DirectX12* a *Vulkan*, ale na druhou stranu jsou mnohem víc "*high-level*"(mají vyšší úroveň abstrakce). Znamená to tedy, že *OpenGL* a *DirectX11* jsou jednodušší pro začátečníky.

Tabulka 1.1: Provnání grafických API

Vlastnost	**OpenGL**	**Vulkan**	**DX11**	**DX12**
Úroveň abstrakce	**Vysoká**	Velmi nízká	**Střední**	Nízká
Výkon	Střední	Vysoký	Střední	Vysoký
Multithreading	Omezený	Vynikající	Omezený	Vynikající
Náročnost	**Snadné**	Obtížné	**Snadné**	Obtížné
Podpora platforem	**Cross-platf.**	**Cross-platf.**	Windows	Windows

Tučně zvýrazněna důležitá kritéria pro cíle práce

OpenGL je i přes svůj věk mnohem snazší pro začátečníky. Trojúhelník nakreslíte ve 100 řádcích kódu. Ve Vulkanu to může zabrat 1000 řádků. To není vtip, můj první pokus o renderování ve Vulkanu zabral 1055 řádků. [21]

Co se týče mého osobního výběru API, byla pro mě důležitá podpora různých platforem. I přesto, že *DirectX* má být teoreticky optimalizovanější pro *Windows*, tak to v praxi nebude takový rozdíl. A nemám důvod diskriminovat například uživatele Linuxu, kterým zároveň jsem.

Rozhodnutí bylo tedy mezi *OpenGL* a *Vulkanem* a vzhledem k cílům práce je **OpenGL** nejlepší volbou.

1.5 Okno

Řekněme, že jsme udělali všechny potřebné výpočty, přesně jak nastínila teorie vykreslovaní. Teď máme někde v paměti uložený výsledek. Jak se nám dostane na obrazovku?

Mohli bychom vytvořit vlastní drivery na monitory, ty následně použít, abychom na monitoru zobrazili obrázek. Fungovalo by to, nicméně v praxi využijeme operační systém. Konkrétně jeho správce oken a pomocí něj budeme zobrazovat render na obrazovce.

Nicméně, jelikož není jen jeden operační system, není ani jeden správce oken. Existuje například Windows správce oken, X11 a Wayland. Každý z nich bude mít různé API a budeme s tím muset počítat. Ideálně jich chceme podporovat co nejvíce, aby grafický engine fungoval na různých operačních systémem.

S tím nám může pomoci nějaká knihovna a dokonce *OpenGL wiki* má jejich seznam. [11]

Tabulka 1.2: Porovnání knihoven na okna s *OpenGL* podporou

Vlastnost	**GLFW**	**SDL**	**freeglut**	**Qt**
Správa okna	**Ano**	**Ano**	**Ano**	**Ano**
Správa vstupů	**Základní**	Pokročilá	**Základní**	Pokročilá
Správa audia	Ne	Ano	Ne	Ano
UI widgety	Ne	Ne	Ne	Ano
Věk	**Moderní**	**Moderní**	Zastaralé	**Moderní**
Velikost	**Malá**	Střední	**Malá**	Obří
Jednoduchost	**Vysoká**	Střední	**Velmi vysoká**	Nízká

Tučně zvýrazněna důležitá kritéria pro cíle práce

GLFW je pro účely práce ideální a dokonce nám později pomůže i se správou uživatelských vstupů, zároveň podporuje i *Vulkan*, kdybychom se rozhodli engine měnit API.

1.6 Systém entity

Dokud vykreslujeme v reálném čase (60 snímků za vteřinu), musíme 60x za vteřinu říct grafickému enginu, co chceme vykreslovat. Mohli bychom si vytvořit nějaké funkce jako `nakresli_kostku(x, y, z)`, následně v `render()` funkci, která se spustí pro každý snímek, budeme spouštět funkci `nakresli_kostku(0, 0, 1)`. Nicméně to nestačí, musíme definovat i stav kamery. Takže bychom ještě museli spouštět funkci `nastav_kameru_na(x, y, z, alfa, beta, gamma)` *(pozice a rotace)*

Tento systém funguje, nicméně u větších projektů kód začne být trochu nepřehledný. Když máme 121x protivníků, které vypadají jako kostky, budeme muset spouštět 121x `nakresli_kostku(x, y, z)` na různých souřadnicích. A orientovat se v 121 prakticky identických řádcích kódu není příjemný zážitek.

Možná už některé programátory napadá, jak napsat kód tak, aby byl přehledný. Ale tato řešení se už blíží k něčemu, co nazýváme systém entit. Způsobů, jak implementovat entity, je několik [18], nicméně mají jednu společnou vlastnost, a to stavové ovládání světa.

Místo toho, aby se při každém snímku přikazovalo enginu *nakresli kostku zde a zde, kameru dej tam*, se na začátku programu řekne, *vytvoř kameru, její pozice bude zde, vytvoř kostku a její pozice bude tam.* V tom okamžiku byly vytvořeny nějaké entity. Ty si engine uloží při vykreslovaní každého snímku s nimi počítá do té

doby, než je někdo smaže. Vlastnosti entit, třeba jako jejich pozice, jde upravovat. Tudíž pro našich 121 protivníků, vytvoříme 121 entit, které se už sami spravují, jdou za hráčem, při doteku ubírají zdraví atd.

V kódu tedy definujeme stav světa, který se nám má zobrazovat, než abychom příkazy nařizovali, co se přesně má stát.

Dokonce to umožňuje enginu, aby prováděl **optimalizace při vykreslování**. Když engine ví, že má vykreslit 10 modrých krychlí, 10 červených krychlí, 10 modrých válců a 10 červených válců, je efektivnější nakreslit tvary stejné barvy pohromadě a může to tak tedy udělat, místo toho, aby spoléhal na to, že uživatel bude posílat příkazy v optimalizovaném pořadí.

Co se týče řešení systému entit, existují dva hlavní druhy [17]. *Objektově Orientovaný Systém* a tzv. *Entity Component System* (viz tabulka). *O. O. Systém* je zpravidla jednodušší na implementaci a považuje se za né moc škálovatelný, jelikož hluboké dědění ve velkých projektech může vést do nepříjemných situací.

Proto většina komerčních enginů používá *ECS*, nicméně z osobních zkušeností, například z *Unity*, mi tento systém entit osobně nikdy nesedl a raději používám více *Objektově Orientované Systémy* jako má *Godot*. *Navíc, když člověk tento systém používá rozumně a vytváří entity hiearchicky hlavně do šířky, tak nikdy na problémy škálovatelnosti nenarazí.

Tabulka 1.3: Provnání řešení systému entit

Vlastnost	**OOS**	**ECS**
Jednoduchost	**Vysoká**	Nízká
Škálovatelnost*	Střední	Vysoká
Preference	*Vysoká*	*Střední*

Tučně zvýrazněna důležitá kritéria pro cíle práce

1.7 Parser modelů ze souborů

Dosud text v příkladech mluvil o prostých modelech (krychle, válec), nicméně správný grafický engine by měl podporovat jakékoliv *polygonové modely*.

Je několik formátů polygonových modelů např.: `.fbx`, `.gltf`, `.blend`, `.obj`.

Teoreticky by jich grafický engine měl podporovat co nejvíce, nicméně vzhledem k cílům práce bude stačit jen jeden. Formát `.obj` bude pro naše potřeby ten nejvhod-

Tabulka 1.4: Porovnání formátů modelů

Vlastnost	**FBX**	**glTF**	**BLEND**	**OBJ**
Otevřený standard	Ne	**Ano**	Částečně	**Ano**
Čitelné člověkem	Ne	JSON varianta	Ne	**Ano**
Podpora animací	Ano	Ano	Ano	Ne
PBR materiály	Omezená	Ano	Ano	Částečně
Hiearchie v modelu	Ano	Ano	Ano	Ne
Rozšířenost	**Vysoká**	**Vysoká**	pouze Blender	**Vysoká**

Tučně zvýrazněna důležitá kritéria pro cíle práce

nější právě kvůli své **jednoduchosti**, **otevřenosti** a **čitelnosti** a, i přes jeho určité nedostatky, je velice rozšířený. [14]

1.7.1 .OBJ formát

OBJ formát byl založen americkou společností Wavefront Technologies a je otevřený, zároveň hrozně jednoduchý, jsou to v podstatě jen souřadnice vrcholů zakódované pomocí ASCII do souboru. [13]

Takto vypadá obsah souboru, který popisuje plošku (ležící čtverec).

```
o Plane
v -1.000000 0.000000 1.000000
v 1.000000 0.000000 1.000000
v -1.000000 0.000000 -1.000000
v 1.000000 0.000000 -1.000000
vn -0.0000 1.0000 -0.0000
vt 0.000000 0.000000
vt 1.000000 0.000000
vt 1.000000 1.000000
vt 0.000000 1.000000
s 0
f 1/1/1 2/2/1 4/3/1 3/4/1
```

Znak `v` definuje vrchol, `vn` definuje normálový vektor, `vt` definuje souřadnici pro textury (*UV*) a znak `f` definuje stěnu (*face*), pomocí kombinace předešlých hodnot. `1/1/1`, znamená 1. vrchol, 1. texturová souřadnice a 1. normálová souřadnice.

Těchto trojic je potřeba minimálně 3, aby vytvořily trojúhelník, ale mohou být i 4, které se během parsování rozdělí z 1 čtverce na 2 trojúhelníky.

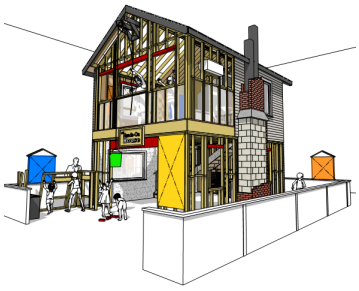
Formát `.OBJ` není moc prostorově efektivní, už jenom proto, že float, který se většinou reprezentuje 4 byty, je reprezentován 9 znaky, což je 9 bytů. Vzhledem k tomu, že většina obsahu modelu jsou jen souřadnice bodů, můžeme očekávat, že model `.obj`, bude alespoň 2x větší něž ostatní formáty modelů. Nicméně alespoň je čitelný člověkem.

1.8 Světelný systém

I přesto, že některé rendery mohou vypadat dobře i bez světel, pro většinu grafických projektů jsou světla nedílnou součastí. Jak už jsme se předtím zmínili, stínění (tedy i osvětlovaní) dělají shadery (česky *stíniče*).

Shadery jsou jen kód a krom některých základních šablon, chceme uživatelům dát možnost si napsat vlastní shader kód, pro větší flexibilitu.

Pokud si například chtějí vytvořit vlastní světelné efekty, třeba místo plynulého přechodu z osvíceného do neosvíceného může louč osvětlovat prostor stupňovitě.



Obrázek 1.6: *Příklad renderu bez stínění, který nevypadá špatně [10]*

Pokud chceme, aby pozice a intenzita světel nebyly natvrdo vepsány do kódu shaderu, musíme data o světelném rozpoložení vykreslovaného světa dodat shaderům. Když už budeme dodávat informace ohledně světel jednomu shaderu, je lepší je dát na jedno místo, kde k nim budou mít přístup všechny shadery, které data budou potřebovat.

Naštěstí přesně něco takového *OpenGL* podporuje, tzv. *Uniform buffer object* je nástroj, pomocí kterého můžeme ukládat data na grafickou kartu na jedno místo, a pak shaderům k datům nastavit přístup. [25] [9]

1.9 Správa uživatelských vstupů

V poslední podkapitole teoretické části se podíváme na vstupy uživatele.

Většina enginů nepoužívá rovnou čistá data uživatelských vstupů. Data jsou většinou dolazena různými způsoby, aby zaručila, že se uživatelské vstupy přeloží do hladkých, pří-

jemných a intuitivních chování ve hře. Dále většina enignů přidává alespoň jednu úroveň abstrakce mezi vstupy a akcemi. Například klávesy prochází skrz tabulku, která přenáší zmáčknutí kláves na určité akce ve hře, což umožňuje uživatelům upravit klávesy tak, jak jim je libo. [19]

Dále autor knihy *Game Engine Architecture* vyjmenovává seznam funkcionalit typického správce uživatelský vstupů:

- **remapování kláves**
- **detekce událostí (například. klávesa zvednuta, klávesa stisknuta)**
- **nastavení tzv. Cursor módu**
- rozpoznání klávesových kombinací
- podpora různých druhů ovladačů
- mrtvé zóny
- analog signal filtering
- podpora několika vstupních zařízení pro několik hráčů
- dočasná deaktivace určitých vstupů

Mnoho těchto požadavků se vztahuje k videoherním ovladačům a k jejim joystickům, které ale engine podporovat zatím nebude. Pro cíle práce jsou nejdůležitější tučné body v seznamu, a ty budou implementovány.

2. Implementace

Druhá část práce se zaměřuje na implementaci grafického enginu. **Zdrojový kód projektu je přístupný**, viz odkaz na repozitář v zadání, proto není třeba vše rozebírat do podrobna. Nicméně i přesto nastíním vývoj a popíšu nějaká řešení pro inspiraci.

2.1 Zprovoznění OpenGL

Implementace začala výběrem programovacího jazyku. *C++* se pro tvorbu enginů používá velice často a umožní i implementaci OOP požadavků. Následovalo samotné zprovoznění *OpenGL*, abych vůbec mohl začít pracovat.

Jelikož jsem nikdy předtím nepoužíval *OpenGL*, *GLFW*, *CMake* a natož *C++*, samotné zprovoznění trvalo celkem dlouho. Hodně pomohl tento článek [22] s touto šablonou [23]. Tak či tak varuji, že tato část může být pro nezkušené, jako já, velice frustrující.

2.2 První trojúhelník

Ekvivalent programu `print("Hello World!")` v počítačové grafice je trojúhelník na obrazovce. V práci samozřejmě nešlo o vytvoření trojúhelníka, nicméně právě jedním trojúhelníkem jsem začal, abych se postupně naučil *OpenGL*, využil zdrojů *LearnOpenGL* od autora *Joey de Vries* [24], který existuje jak v digitální, tak i knižní formě, a rozhodně je doporučuji.

2.3 Zdroje (Resources)

Po úspěšném vykreslení trojúhelníku bylo potřeba kód zobecnict. Stejně jako v systému entit jsem zvolil pro zdroje OOP strukturu.

Napsaly se tedy classy pro:

- Mesh (*polygonová síť*)
- Model (*sada Meshů, viz dále*)
- Material
- Texturu
- *Shader Program*
- `Shader`

Jak se tedy zdroje používají? Řekněme, že uživatel chce načíst zdroj, zkonstruje objekt `Mesh("object/file.obj")`, to alokuje Mesh na grafické kartě a následně ho objekt reprezentuje. Při jeho dekonstrukci se z grafické karty smaže.

Protože jeden Mesh může být využíván několika entitami, je lepší aby zdroj byl ve formě `std::shared_ptr<Mesh>`, což i zaručí, že se Mesh z GPU nedealokuje, zatímco je ještě využíván.

Bohužel má tento systém jedno velké úskalí, které jsem zjistil až později. Jelikož není žádné omezení, kde zkonstruovat objekt, může se stát, že kdyby se Mesh zkonstruoval na jiném vlákně, kde nebude *OpenGL* kontext, tak příkazy zamítne a pro jistotu vypne program.

Znamená to tedy, že **nejdou vytvářet zdroje na jiných vláknech**. Pokud tuto možnost chceme, řešením je udělat centrální alokátor na správném vlákně, jemuž se přidávají požadavky do fronty a on je následně ve správný čas alokuje a vrátí referenci. Analogicky to platí i pro mazání zdrojů.

Pro naše účely to není takový problém, ale pokud by někdo měl plánovat vytváření zdrojů na různých vláknech, rozhodně doporučuji na to myslet při implementaci od začátku a navrhnout systém vhodně.

Tento OOP systém s `std::shared_ptr` je analogický pro zbytek až na *ShaderProgram* a `Shader`.

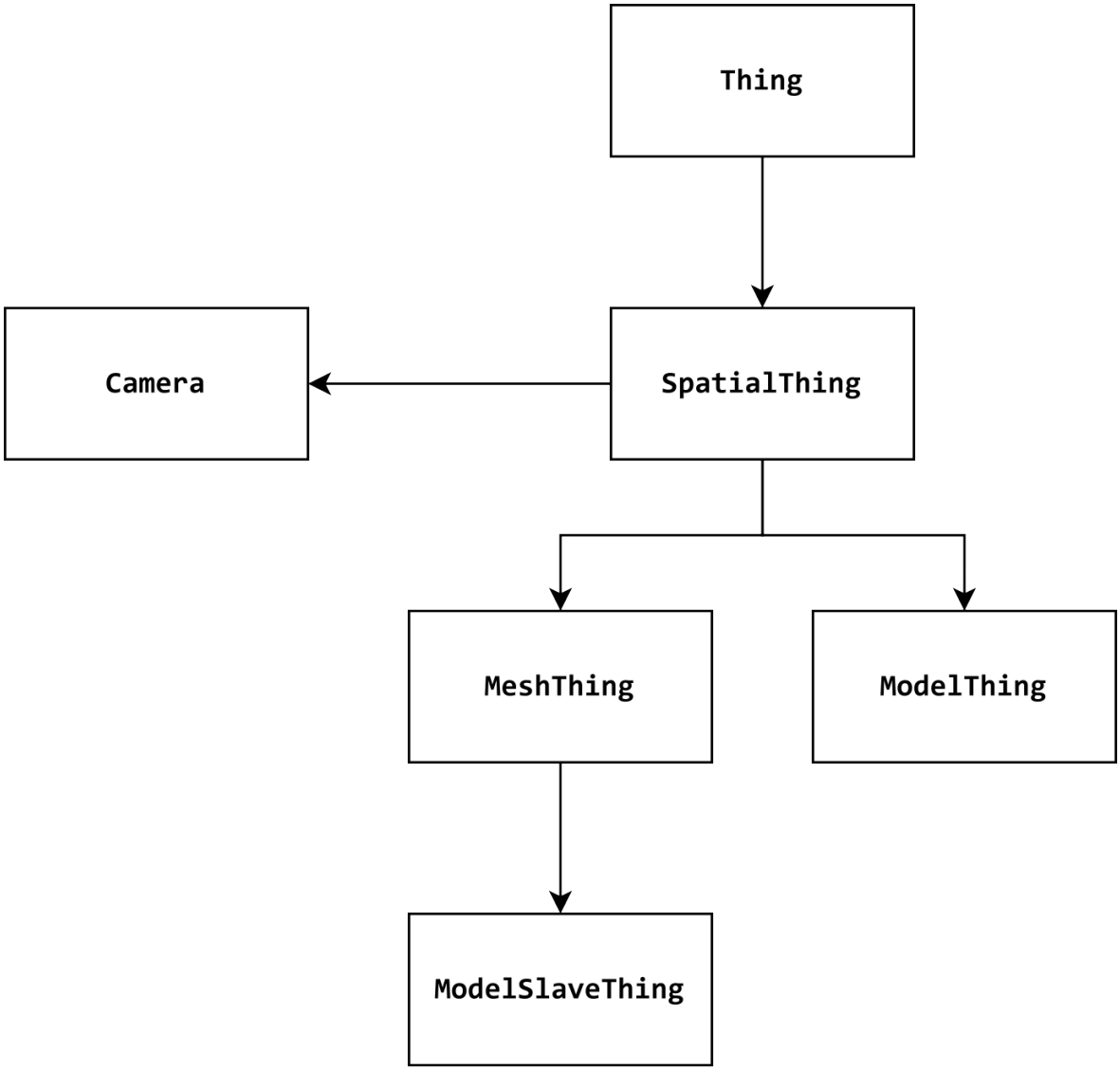
Jelikož `Shader` je objekt na jedno použití, není třeba řešit `std::shared_ptry`, a engine s ním pracuje jako s prostým objektem.

ShaderProgram je vpodsatě jen wrapper na *OpenGL ID*, proto jsem usoudil, že není třeba objekt sdílet, objekt se tedy kopíruje a jeho počet využití se počítá centrálně. Což by byl problém, jelikož počítání není vyřešené pro různá vlákna, nicméně to

stejně současný systém nepodporuje, takže to ničemu nevadí.

2.4 Systém entit

Zdroje ale samy o sobě nic nedělají, někdo je musí využít, což dělají právě entity.



Obrázek 2.1: *Diagram dědičné strukury entit*

- **Thing** classa reprezentuje obecnou entitu, která může mít nějaké chování, má několik vlastnosti, `paused`, `visible` a `render_layer`, k tomu se vážou dvě metody této classy, `update()` a `render()`, které pochopitelně mají i všechny její potomci.
  Je to důležité, protože právě podle `paused` a `visible` spouští engine na všech entitách v *update fázi* `update()` metodu všech entit a v *render fázi* `render()` metodu všech entit.

- **SpatialThing** má navíc vlastnosti pozice, rotace a velikosti.
- **Camera** je spíše interní classa pro engine a reprezentuje kameru, z jejíhož pohledu se bude vykreslovat. Uživatel jich může vytvořit několik a střídat je dle libosti.
- **MeshThing** je entita, která se již dá vykreslovat s pomocí jednoho zdroje `Mesh` a jednoho zdroje `Material`
- **ModelThing** umí reprezentovat i více barevné objekty. Jelikož více barevné modely jsou vpodstatě mýtus, je to jen kombinace několika jednobarevných modelů, byla tato reprezentace implementována tak, že na základě zdroje `Model` se při vytváření této entity vytvoří dalších *N* entit typu **ModelSlaveThing**, které se pak chovají jako děti vytvořené ModelThing entity.

Tvorba entit má ale jeden zásadní problém v C++ (a v dalších jazycích, ve kterých se manuálně spravuje pamět). Když uživatel vytvoří entitu, musí o ni dát enginu vědět, aby s ní mohl engine pracovat. Problém nastane, když uživatel entitu smaže, aniž by o tom enginu dal věděl. Engine pak při projíždění entit v *update* a *render* fázi, šáhne na smazanou pamět a program ukončí operační systém (*segmentation fault*).

Toto nesmíme dopustit. Vlastníkem entity musí být engine, uživatel si pouze musí vyžádat vznik a dostane nějaké *ID*, pomocí kterého bude mít k entitě přístup.

Implementace tohoto systému je za mě celkem povedená, proto ji i rozepíšu.

Entita se vytvoří pomocí metody `add` v Engine classe, ale tato metoda je trochu speciální, jelikož je to tzv. *temaplete* (šablona). Místo toho, abychom měli funkci `add` pro `Thing`, `SpatialThing`, `ModelThing` atd., máme šablonu a kompilátor už vytvoří požadované varianty funkcí.

Parametry metody `add` parametry se v metodě přesunou do konstruktoru entity, kterou chceme vytvořit, uloží si ji jako `std::unique_ptr` a vrátí interní objekt `geRef`, který obsahuje interní *ID* v enginu.

```
template<typename T, typename... Args>
requires std::is_base_of_v<Thing, T>
geRef<T> add(Args&&... args) {
    geRef<T> ref{next_thing_id, this};
    next_thing_id++;

    auto thing = std::make_unique<T>(std::forward<Args>(args)...);

    /.../
```

```
    things[ref.id] = std::move(thing);

    return ref;
};
```

2.5 .OBJ parser

Implementace `OBJ` parseru byla zajimavá v tom, že jsem ho opravdu implementoval, místo využití nějaké knihovny.

Není jedním z nejrychlejších, nicméně jeho časová složitost je **O(n)**, kde **n** je počet řádků, něco, co se může zdát jako samozřejmost, ale není...

OpenGL chce data o polygonové síti jako jeden seznamu floatů (tzv. *vertex data*). Jednotivé vrcholy jsou po N číslech, kde N je minimálně 3 (*souřadnice X, Y, Z bodu*), ale dále tam mohou být normály, texturové souřadnice, barvy nebo cokoliv jiného. Po 3 vrcholech (*tedy po* $3 * N$ *číslech*) je 1 trojúhelník. Takto je popsána celá polygonová síť.

Nicméně, tento systém může vést ke spoustě opakování v datech. Naříklad každý čtverec je reprezentován 2 trojúhelníky, pomocí 6 vrcholů, ale přitom jen 4 jsou unikátní. (*až +50% místa*)

Proto se používají tzv. *indecies*, což je seznam celých čísel (indexy vrcholů ve *vertex data* seznamu), které po 3 definují trojúhelníky.

Avšak formát .obj neobsahuje senzam *indecies* a v tom spočívá netrivialita **O(n)** parseru. Během parsování, protřebujeme mít **O(1)** kontrolu, jestli už vrchol nebyl definován předtím, abychom použili pouze jeho *indecy* a nemuseli ho definovat znovu. K tomu nám poslouží skvělá datová struktura `std::unordered_map`, a dále nějaký rychlý způsob jak vytvořit hash reprezentující vrchol. A protože je právě definován třemi čísly `v/vt/vn`, tak se hash udělal z nich.

```
size_t hash = std::hash<size_t>()(v) ^
            (std::hash<size_t>()(vt) << 8) ^
            (std::hash<size_t>()(vn) << 16)
```

Standard formátu `.obj` je dobře zdokumentován na wikipedii a nikde jinde jsem

moc dokumentace nenašel, zdroje z wikipedie už také nefungují. [13] Ale popravdě formát zas tak složitý není.

Dobré ještě podotknout, že při parsování `.obj` souborů člověk narazí i na parsování `.mtl` souborů, které popisují materiály. (*specifikace* [15])

S tím, že, aby materiály byly aplikovány rovnou v enginu, je třeba implementovat základní shadery (`res/shaders/obj_*.glsl`), které se automaticky aplikují na načtený model.

2.6 Systém světel

Implementace systému světel nebyla nijak komplikovaná, jelikož šlo využít flexibilní struktury systému entit. Vytvořily se classy: `DirectionalLight`, `PointLight` a `SpotLight`, které reprezentují jednolivé druhy světel.

Rozšíří se metoda `add`, pokud vytváříme jeden z typů světel, tak se jeho reference uloží do správce světel `Lights`, který každý snímek updatuje *OpenGL* **Uniform Buffer Object**.

Na straně shaderu se přidá následující kód:

```
/* <GRAPHIC ENGINE TEMPLATE CODE> */
struct PointLight{
    vec4 light_data;
    vec3 position;
};

struct DirectionalLight{
    vec4 light_data;
    vec3 direction;
};

struct SpotLight{
    vec4 light_data;
    float cut_off;
    vec3 position;
    vec3 direction;
};

layout (std140) uniform LIGHTS
{
```

```
    vec3 BASE_AMBINET_LIGHT;
    PointLight point_lights[NR_POINT_LIGHTS];
    DirectionalLight directional_lights[NR_DIRECTIONAL_LIGHTS];
    SpotLight spot_lights[NR_SPOT_LIGHTS];
};
/* </GRAPHIC ENGINE TEMPLATE CODE> */
```

a systém lze využívat.

Při kompilaci shaderů engine zjistí, zda shader tato data bude potřebovat, a pokud ano pomocí `glUniformBlockBinding(...)` je *Shader Programu* napojí.

Důležité podotkout, že shader kód počítá s fixním počtem světel. Například s 32 bodovými světly. I přesto, že existuje jen jedno, dvě, či žádné. V cyklu, který počítá celkové osvětlení, jede *for* cyklus vždy přes 32 světel. Světla, která neexistují mají, vynulovaná data, tedy nic neosvítí.

Může se to zdát neefektivní, ale právě opak je pravdou. Tím, že je v shaderu definován konstatní počet světel v času kompilace, shader kompilátor dokáže cyklus zoptimalizovat (tím, že ho ručně rozepíše) a nemusí tedy využívat pomalé *jump* instrkuce.

Na sčítání a násobení velkých kvant čísel je grafická karta dobrá a úkoly jdou velice dobře zparalelizovat, přičemž přeskakování v cyklech může situaci dost zkomplikovat.

Co se stane, pokud počet světel předčí definovaný počet. V tu chvíli jsme v pasti.

Engine má implementována 2 řešení, resp. 3. Za prvné může uživatel prostě zvýšit limit světel, podle toho, kolik sám bude potřebovat. Nebo si může nastavit v enginu 2 řešení tohoto problému, tzv. `SORT_BY_PROXIMITY` nebo `CANCEL_NEW`. Jedno najde každý snímek N nejbližších světel ke kameře. A `CANCEL_NEW` vám zakáže vytvářet nové entity světla, dokud se nějaké nebudou odstraněny.

Obě řešení mají úskalí. `CANCEL_NEW` vám moc nepomůže a `SORT_BY_PROXIMITY` může být celkem pomalý. Ideální složitost řazení k členů v n dlouhém seznamu je $O(n * \log(k))$, nicméně zde můžeme využít takový trik. Nám nejde seřazení světel podle vzdálenosti, ale jen o nalezení k nejbližších...

Lze tedy využít funkce `std::nth_element(...)`, která zaručuje, že nějaký Ntý člen bude na místě, kde by byl, kdyby seznam byl seřazen, a zároveň zajistí, že všechna čísla před ním jsou menší a všechna čísla za ním jsou větší. A to se složitostí $O(n)$.

Tento systém funguje skvěle, ale samozřejmě časem přestane, na mém počítači při počtu 10000 světel začal znatelně klesat počet snímků za vteřinu.

Možné rozšíření by mohlo být využití nějakého *oktetového stromu*, ve kterém si budeme udržovat vztah jednotlivých světel v prostoru a tedy najít k nejbližších světel, nicméně pro účely práce to takto stačí.

2.7 Vstup uživatele

Vytvořit systém vstupu uživatele už nebylo vůbec složité.

Využilo se *GLFW* a byla sepsána sada jednoduchých funkcí, které API obalují, za vzniku funkcí jako `is_pressed(int action)`, `is_just_pressed(int action)` atd. viz. *technická dokumentace*.

GLFW dodává o myši jen relativní pozici od rohu okna. *Input manager* si proto ukládá jak současnou, tak poslední pozici myši a počítá relativní posun myši, který jde využít například při ovládání hráče z první osoby, také viz *technická dokumentace*.

3. Technická dokumentace

Poslední kapitola se věnuje technické dokumentaci projektu.

3.1 Jak engine spustit?

Engine je naimplementován jako knihovna. Zde je jednoduchý návod jak zprovoznit základní prázdný projekt, ze kterého můžete stavět dál.

Základní prázdný projekt:

I. Vytvořte prázdou složku vašeho projektu.

II. Do ní naclonujte / stáhněte repozitář knihovny. (**https://github.com/bagons/graphicengine**)

III. Do složky vašeho projektu přidejte *main.cpp* ve kterém budete psát apliaci. Příklad základního prázdného projektu, doporučuji zkopírovat:

```
#include "graphicengine.hpp"
#include <iostream>

// Nastavení enginu
Engine ge{"window name", 500, 500};

int main(){
    std::cout << "Project started" << std::endl;

    while (ge.is_running()){
        ge.update();
        ge.send_it_to_window();
    }

    std::cout << "Project ended" << std::endl;

    return 0;
}
```

IV. Do složky vyšeho projektu přidejte *CMakeLists.txt*

```
# CMake Nastavení
cmake_minimum_required(VERSION 3.31)
set(CMAKE_CXX_STANDARD 20)
# Název vašeho projektu
project(demo)

# Připojení knihovny
add_subdirectory(graphicengine)
include_directories(graphicengine/include)

# Soubor(y) vašeho projektu
add_executable(${PROJECT_NAME} main.cpp)
target_link_libraries(${PROJECT_NAME} graphicengine)

# Překopírování složek zdrojů k sputitelnému souboru
graphicengine_setup(${PROJECT_NAME})
```

V. Zkompilujte program pomocí *CMake*u. Stačí v složce spustit příkazy:
   (a) `cmake .`
   (b) `cmake --build .`

3.2 Jak engine používat?

Funkcí v enginu je mnoho, proto místo vypisování veškeré dokumentace do této práce odkazuji na externí dokumentaci, ve které lze vyhledávat, proklikávat a vše je mnohem přehlednější, než by kdy to bylo zde, a zároveň obsahuje i návody.

Externí dokumentace: **https://bagons.github.io/graphicengine/**

Závěr

I když se to možná nemusí zdát, projekt byl celkem náročný. Všechny požadavky zadání byly splněny a výsledekem je jednoduchý, ale použitelný grafický engine.

Během projektu jsem narážel na plno zádrhelů a nezčetně krát jsem si musel vzít papír nebo tabuli, abych si to v hlavě dostatečně uspořádal. Také nepomohlo učení se nového programovacího jazyka, ale jsem rád, že jsem si tím prošel.

Zajimavé je, že po tomto zážitku začínám chápat všechny ty podivné implementace a ovládaní videoherních enginů. Člověk naráží na podobné problémy během vývoje, uvědomí si, že bude muset udělat nějaký kompromis, a pak mu dojde, že stejný kompromis zažil i v jiných enginech, a zjistí, že nástroje, které ho celou dobu trápily v podsatě jinak udělat nejdou.

Když se nad projektem zamyslím zpětně, tak se mi to nezdá tak komplikované, což nemusí znamenat, že to tak je, ale že jsem se něco naučil.

Seznam použité literatury

[] *Bullet Physics SDK: real-time collision detection and multi-physics simulation for VR, games, visual effects, robotics, machine learning etc.* URL: https://github.com/bulletphysics/bullet3.

[] *Cube render*. URL: https://www.vecteezy.com/png/22651650-3d-cube-rendering.

[] *Irrlicht Engine is an open source realtime 3D engine written in C++*. URL: https://irrlicht.sourceforge.io/.

[] *Kapitola 4 Rasterizace objekt*. URL: https://mathonline.fme.vutbr.cz/pg/flash/TeorieGrafika/pocGrafika4.pdf.

[] *Object-Oriented Graphics Rendering Engine*. URL: https://www.ogre3d.org/.

[] *raylib is a simple and easy-to-use library to enjoy videogames programming.* URL: https://www.raylib.com/.

[] *raylib is a simple and easy-to-use library to enjoy videogames programming.* URL: https://www.pygame.org.

[16] *Zobrazovací řetězec a obrazová paměť, operace s fragmenty*. 2016. URL: https://cw.fel.cvut.cz/old/_media/courses/b0b39pgr/08-zobrretezec.pdf.

[17] "Uniform Buffer Object". In: *Wiki Khronos* (2017). URL: https://wikis.khronos.org/opengl/Uniform_Buffer_Object.

[23] *Unshaded render from a forum*. 2023. URL: https://forum.vectorworks.net/index.php?/topic/91511-unshaded-polygon-no-lines-render-mode-needed/.

[24] "Related toolkits and APIs". In: *Wiki Khronos* (2024). URL: https://wikis.khronos.org/opengl/Related_toolkits_and_APIs#OpenGL_initialization.

[25a] "Game Engine Market Size, Share and Industry Analysis". In: *Fortune Business Insights* (2025). URL: https://www.fortunebusinessinsights.com/game-engine-market-111802.

[25b] "Wavefront .obj file". In: *Wikipedia* (2025). URL: https://en.wikipedia.org/wiki/Wavefront_.obj_file.

[26] *10924 availabe .obj models ke dni 31. 1. 2026*. 2026. URL: https://free3d.com/3d-models/obj.

[DT95] Linda Rose Diane Ramey a Lisa Tyerman. "MTL material format (Lightwave, OBJ)". In: (1995). URL: https://paulbourke.net/dataformats/mtl/.

[Fra24] Pouhela Franc. *3D Game Engine Development with C++ and OpenGL*. Ún. 2024, s. 4. ISBN: 9798878071727. URL: https://www.researchgate.net/publication/380075599_3D_Game_Engine_Development_with_C_and_OpenGL.

[Gre09a] Jason Gregory. *Game Engine Architecture*. CRC Press, 2009. Kap. 1.3.

[Gre09b] Jason Gregory. *Game Engine Architecture*. CRC Press, 2009. Kap. 16.2.

[Gre09c] Jason Gregory. *Game Engine Architecture*. CRC Press, 2009. Kap. 9.5.

[Kar] Ondřej Karlík. "Úvod do problematiky polygonového modelování". In: *Fakulta Elektrotechnická ČVUT, Praha* (). URL: http://keymaster.powermac.cz/temp/b.pdf.

[Ksh25] Kshitijaucharmal. "Why Vulkan Is Better (But You Might Want OpenGL Anyway)". In: *Medium - FOSSible* (2025). URL: https://medium.com/fossible%20why-vulkan-is-better-but-you-might-want-opengl-anyway-f797cf9cfaea.

[Piw21a] Robert Piwowarek. "Journey through setting up hello world OpenGL project in CLion and a multitude of issues along the way". In: (2021). URL: https://robpiwowarek.github.io/gamedev/2021/09/03/opengl-helloworld.html.

[Piw21b] Robert Piwowarek. "OpenGL Seed Project". In: (2021). URL: https://github.com/RobPiwowarek/OpenGL_Seed_Project.

[Vri14a] Joey de Vries. "LearnOpenGL.com". In: (2014). URL: https://learnopengl.com/.

[Vri14b] Joey de Vries. "LearnOpenGL.com - Advanced OpenGL - Advanced GLSL". In: (2014). URL: https://learnopengl.com/Advanced-OpenGL/Advanced-GLSL.

[Vri14c] Joey de Vries. "LearnOpenGL.com - Getting started - Coordinate Systems". In: (2014). URL: https://learnopengl.com/Getting-started/Coordinate-Systems.

[Vri14d] Joey de Vries. "LearnOpenGL.com - Getting started - Transformations". In: (2014). URL: https://learnopengl.com/Getting-started/Transformations.

Seznam obrázků

Seznam tabulek