2024

Křesťanské gymnázium

Kozinova 1000

102 00 Praha 10 – Hostivař

3D simulátor kolejových vozidel

MATYÁŠ BARTALOŠ

Prohlášení

Prohlašuji, že jsem maturitní práci zpracoval samostatně a že jsem uvedl všechny použité informační zdroje a literaturu v seznamu použitých zdrojů.

Poděkování

Zde bych chtěl poděkovat paní profesorce Mgr. Ivoně Spurné za vedení projektu, svým rodičům za jazykovou korekturu práce a svým kamarádům za podporu při tvorbě projektu.

Abstrakt

Maturitní práce se zabývá tvorbou 3D enginu pro simulaci kolejových vozidel v programovacím jazyce C++. Projekt je rozdělen na čtyři části: sestavení programovacího prostředí, vytvoření systému pro pohyb objektů po kolejích, načítání 3D dat ze souborů ve formátu XML a tvorba uživatelského rozhraní. Tyto čtyři části tvoří jádro tohoto projektu.

Klíčová slova

Vlakový simulátor, metro, 3D grafika, herní engine, XML soubory, OGRE (Object-Oriented Graphics Rendering Engine), PhysX, CMake, PBR (Physically Based Rendering)

Abstract

The graduation thesis deals with the creation of a 3D game engine for train simulation written in the C++ programming language. The project consists of four parts: preparing an environment for development, creating a system for moving objects along rails, loading of 3D data from files in the XML format, and creating a graphical user interface. These four parts form the core of this project.

Key words

Train simulator, subway, 3D graphics, game engine, XML files, OGRE (Object-Oriented Graphics Rendering Engine), PhysX, CMake, PBR (Physically Based Rendering)

Obsah

Souhrny objektů

Seznam obrázků

Seznam tabulek

1. Úvod

V dnešní době technologického pokroku a digitalizace se stále více oblastí našeho života přesouvá do virtuálního prostoru. Jedním z průkopnických odvětví v tomto směru je simulace. Simulátory nabízejí uživatelům možnost získat realistickou zkušenost bez nutnosti fyzické přítomnosti v daném prostředí, což má široké uplatnění nejen v zábavním průmyslu, ale i v profesionálních a vzdělávacích aplikacích. Tato práce se zaměřuje konkrétně na simulaci kolejových vozidel (tj. vlaků, metra či tramvají).

Cílem této maturitní práce je prozkoumat a vytvořit funkční prototyp 3D vlakového simulátoru, který bude schopen realisticky replikovat základní aspekty řízení a provozu vlaků. Práce je strukturována do několika částí. Teoretická část se věnuje výběru vhodných nástrojů a technologií k tvorbě projektu a popisuje základní architekturu projektu. Praktická část je rozdělena na čtyři části podle jednotlivých výstupů. V závěrečné části jsou analyzovány výsledky testování a diskutována možná vylepšení a rozšíření simulátoru.

Vzhledem k rostoucí popularitě simulátorů a jejich významu v oblasti vzdělávání a odborné přípravy má tento projekt ambici nejen poskytnout zábavu, ale i podnítit zájem o technologie a inženýrské obory obecně. Práce by měla sloužit jako ukázka toho, jak moderní technologie mohou přispět k efektivnímu a interaktivnímu učení.

2. Teoretická část

Před zahájením tvorby projektu bylo potřeba rozhodnout, jak a s jakými nástroji projekt udělat. Jedna možnost byla využít existující herní enginy jako je Unity, Unreal nebo Godot. V těch už je implementována většina základních funkcí – vykreslování grafiky, simulace fyziky, komunikace s operačním systémem, načítání souborů atd. K vytvoření hry potom většinou stačí vytvořit scénu v editoru a napsat logiku hry pomocí skriptů. Vzhledem k tomu, že cíl mého projektu je vytvořit simulátor, který má být schopen načítat herní obsah ze souborů, blíží se spíše hernímu enginu než konkrétní hře. Vyvíjet herní engine v již existujícím herním enginu by moc nedávalo smysl.

Rozhodl jsem se tedy vymyslet a napsat vlastní herní engine v programovacím jazyce C++. Nejdůležitější částí enginu je takzvaný renderer, který slouží pro vykreslování (renderování) obsahu na obrazovku [1]. Další důležitou částí je detekce kolize a fyzikální simulace, díky které pohyby a interakce ve hře působí realisticky. Dále je potřeba systém uživatelského rozhraní a zpracování dat z vstupních zařízení (klávesnice, myš). Větší enginy často obsahují i systémy pro zvuk, skriptování, síťovou komunikaci, umělou inteligenci či vlastní editor. Tyto součásti v mém enginu zatím nejsou implementovány.

2.1. Renderer

Renderovací systém lze vytvořit několika způsoby. Lze přímo využít renderovací rozhraní (API – Application Programming Interface), které komunikuje přímo s ovladačem grafické karty. Toto řešení je sice velmi výkonné, ale jeho vývoj je poměrně náročný. Navíc podpora různých renderovacích API se liší mezi operačními systémy.

OpenGL	multiplatformní (na Androidu a Applu špatná podpora)
OpenGL ES	mobilní platformy (Android, na iOS ukončena podpora 2018 [2])
Direct3D	Windows a Windows Phone
Vulkan	multiplatformní (na macOS a iOS skrz abstrakční vrstvu MoltenVK [3])
Metal	macOS, iOS

Tabulka 1: Přehled renderovacích API a podporovaných operačních systémů

Proto jsem se rozhodl použít na renderování externí knihovnu. Ta sice také interně používá tyto renderovací API, ale uživatel s knihovnou komunikuje přes jednotné rozhraní. Pro výběr vhodné knihovny jsem si definoval následující kritéria:

1. Musí být použitelná v jazyce C++
2. Zdrojový kód musí být volně dostupný
3. Musí podporovat všech 5 nejpoužívanějších operačních systémů (Windows, Linux, macOS, Android a iOS)
4. Měla by podporovat schopnosti moderních grafických karet
5. Měla by být v aktivním vývoji

Nakonec jsem si k vývoji renderovacího systému vybral knihovnu OGRE (Object-Oriented Graphics Rendering Engine) [4], která splňovala všechna kritéria. V průběhu tvorby projektu jsem přešel na novější vývojovou větev OGRE-Next, která je rychlejší a má lepší podporu pro systém dynamického osvětlení [5].

Pro detekci kolize a fyzikální simulaci jsem zvolil knihovnu Nvidia PhysX, která je volně dostupná, je často používána velkými projekty, je velmi stabilní a rychlá a má přehledné a intuitivní rozhraní [6].

2.2. Jádro enginu

Naprostá většina dnešních počítačů má vícejádrové procesory, které umí zpracovávat více věcí paralelně. Procesy se mohou skládat z tzv. vláken, která běží nezávisle na sobě. Jádro enginu je rozdělené na dvě vlákna – jedno na logiku a druhé na renderování. Renderovací vlákno komunikuje s knihovnou OGRE a kromě renderování se stará o vytvoření okna, načtení konfiguračních souborů a získávání dat ze vstupních zařízení. Logické vlákno zpracovává vstupní data, simuluje fyziku (pomocí PhysX) a načítá a zpracovává soubory obsahují definici scény.

2.3. Hierarchie objektů v knihovně OGRE

K zobrazení libovolného objektu na obrazovku pomocí knihovny OGRE je potřeba vytvořit instanci `Ogre::SceneNode` a na něj připojit `Ogre::Item` (předpona `Ogre::` značí v jazyce C++ název jmenného prostoru, slouží tak k rozlišení tříd, které spadají spadají pod můj

projekt a které pod knihovnu [7]). `Ogre::SceneNode` je jakýsi „kontejner", který obsahuje informace o pozici, rotaci a měřítku. Lze k němu připojovat další objekty včetně `Ogre::SceneNode` a `Ogre::Item`. `Ogre::Item` je instancí objektu ve scéně [8]. Může to být vlak, hráč, nástupiště, cokoliv. Objekty `Ogre::Item` jsou založeny na diskrétních kolekcích geometrie, které jsou reprezentovány třídou `Ogre::Mesh`. Více instancí `Ogre::Item` může být založeno na stejné `Ogre::Mesh`, protože často je potřeba ve scéně vytvořit více kopií stejného typu objektu. Tímto způsobem lze vytvořit veškerou grafiku ve hře, ale protože se jedná pouze o vizuální záležitost, objekty mezi sebou nijak neinteragují a mohou skrz sebe volně procházet.

Pro simulaci fyziky se používá metoda zvaná *collision detection*, která detekuje, jestli se nějaké objekty překrývají, a případně simuluje jejich odraz pomocí základních fyzikálních zákonů. K tomuto účelu slouží knihovna PhysX.

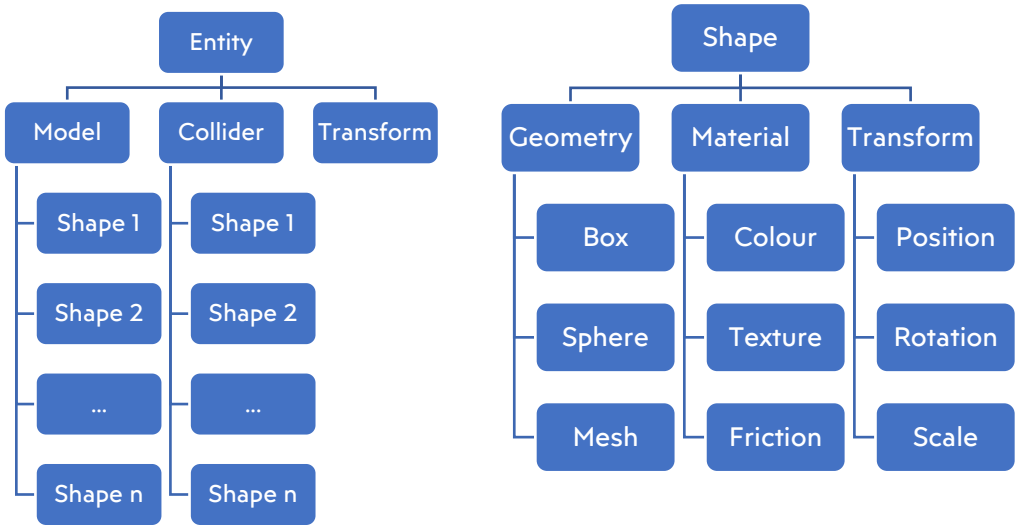
2.4. Hierarchie objektů v knihovně PhysX

PhysX má lehce jinou strukturu objektů [6]. Základním objektem je scéna (`PxScene`), do které jsou přidávány objekty pomocí třídy `PxActor`. Každý `PxActor` může obsahovat jeden nebo více instancí `PxShape`, které definují vlastnosti a tvar objektu použitého v simulaci. `PxShape` je soubor pozice, rotace, materiálu (`PxMaterial`), který definuje fyzikální vlastnosti, např. tření, a geometrie (`PxGeometry`), což je samotný tvar objektu. `PxGeometry` může být kvádr (`PxBoxGeometry`), koule (`PxSphereGeometry`) nebo uživatelem definovaný tvar (`PxTriangleMeshGeometry`; je to v podstatě analogie `Ogre::Mesh`).

2.5. Hierarchie objektů v mém projektu

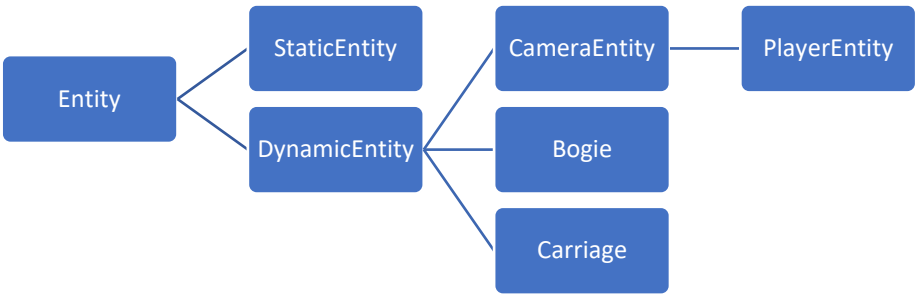
Náročným úkolem bylo navrhnout systém, který by stmeloval oba výše zmíněné koncepty a umožnil snadné přidání další knihovny (např. pro 3D zvuk). Implementaci jsem provedl pomocí třídy `GameEntity`, která obsahuje třídy `Model`, `Collider` a `Transform` (souhrnné označení pro pozici, rotaci a měřítko). `Model` definuje vlastnosti pro grafický vzhled, zatímco `Collider` definuje vlastnosti pro fyzikální simulaci. Každá `GameEntity` může (ale nemusí) obsahovat `Model` a `Collider`. `Model` i `Collider` se skládají z jedné nebo více instancí třídy `Shape`, která se skládá (podobně jako `PxShape`) z transformace (`Transform`),

materiálu (`Material`) a geometrie (`Geometry`). Třída `Material` může obsahovat jak fyzikální vlastnosti (tření, pružnost), tak parametry grafického vzhledu (barva, odrazivost, hrubost povrchu, kovovost, index lomu a textury).



Obr. 1: Hierarchie virtuálních objektů

Každý objekt, který má být součástí mého virtuálního světa, musí dědit ze třídy `GameEntity`. Pro zvýšení účinnosti simulace se `GameEntity` dělí na dva podtypy: `StaticEntity` a `DynamicEntity`. `StaticEntity` je určena pro objekty, které během hry nebudou nikdy měnit svoji pozici (např. stěny, nástupiště, stromy, …), zatímco `DynamicEntity` slouží pro objekty, které se mohou pohybovat (hráč, vlaky, dveře, …). Z `DynamicEntity` dále dědí `CameraEntity`, která implementuje funkce pro pohyb po světě a `PlayerEntity`, která implementuje přímo simulaci hráče (tedy včetně detekce kolize). Obdobně fungují i třídy `Bogie` a `Carriage`, které implementují pohyb vlaku.



Obr. 2: Hierarchie entit

2.5.1. Objekt hráče

Hráč (`Player`) je unikátní objekt, který slouží k volnému pohybu po virtuálním světě a jeho zorné pole je vykresleno na obrazovku. Je přímo ovládán vstupem z myši a klávesnice. Díky detekci kolize je ošetřeno, že hráč nemůže procházet skrz stěny a podobné pevné objekty. Pokud hráč vstoupí do vlaku, stává se jeho strojvedoucím a může ho ovládat.

2.5.2. Objekt kolejí

Samotné jádro vlakového simulátoru je pohyb po kolejích. Kolejová dráha je definována jako množina bodů, kterými má trať procházet. Model kolejí je vygenerován dynamicky při spuštění hry, čímž se předejde vizuálním artefaktům ve spojích kolejí. Díky tomu koleje působí přirozeně a realisticky. Koleje mohou být nakloněny, aby se vyrovnaly boční síly při průjezdu vlaku obloukem ve vyšších rychlostech. Náklon (anglicky *cant*) se udává ve stupních. Výškový rozdíl mezi nižší a vyšší kolejí se nazývá *superelevace* [9].

3. Praktická část

Před zahájením programování samotného projektu bylo potřeba si připravit vývojové prostředí. Vývoj probíhal na operačním systému Linux v programu Visual Studio Code. Jako první krok bylo potřeba stáhnout a sestavit všechny potřebné softwarové knihovny. V rámci této práce jsem projekt sestavoval pro Windows a Linux. Později mám v plánu přidat podporu i pro macOS, Android a iOS, aby bylo podporováno 5 nejčastějších operačních systémů.

3.1. Sestavení potřebných knihoven

Jedna z velkých nevýhod programovacího jazyka C++ je, že neexistuje jednotný systém pro sestavování programů nebo knihoven. Každý operační systém má vlastní nástroje pro vytváření programů, takže proces sestavování se velmi liší. Pro sestavení multiplatformního programu by tedy bylo nutné nastavit kompilační nástroje na každém operačním systému zvlášť. Byl by to zdlouhavý a na chyby náchylný proces, který je navíc do budoucna velmi špatně udržitelný. Pro řešení tohoto problému vznikl nástroj CMake, který nastavení kompilačních nástrojů provede pomocí CMake skriptů [10]. Tyto skripty jsou jednotné na všech operačních systémech.

3.1.1. Postup sestavování

Všechny knihovny, které jsou v projektu použity, také používají k sestavení CMake. Tyto knihovny bylo potřeba každou zvlášť sestavit. Proces se mezi knihovnami lehce liší, protože nejsou definována striktní pravidla, jak by se mělo sestavení provést. V ideálním případě by proces vypadal zhruba následovně:

1. Stáhnout z internetu zdrojový kód knihovny
2. Vytvořit složku build
3. Nakonfigurovat prostředí pro sestavení knihovny pomocí CMake
4. Sestavit samotnou knihovnu
5. Zkopírovat sestavené soubory do předem určené složky (Tento krok se nazývá *instalace*; provádí se, protože při sestavování vznikne poměrně velké množství souborů nutných pouze pro sestavení. Instalace zkopíruje pouze potřebné soubory do nové čisté lokace.)

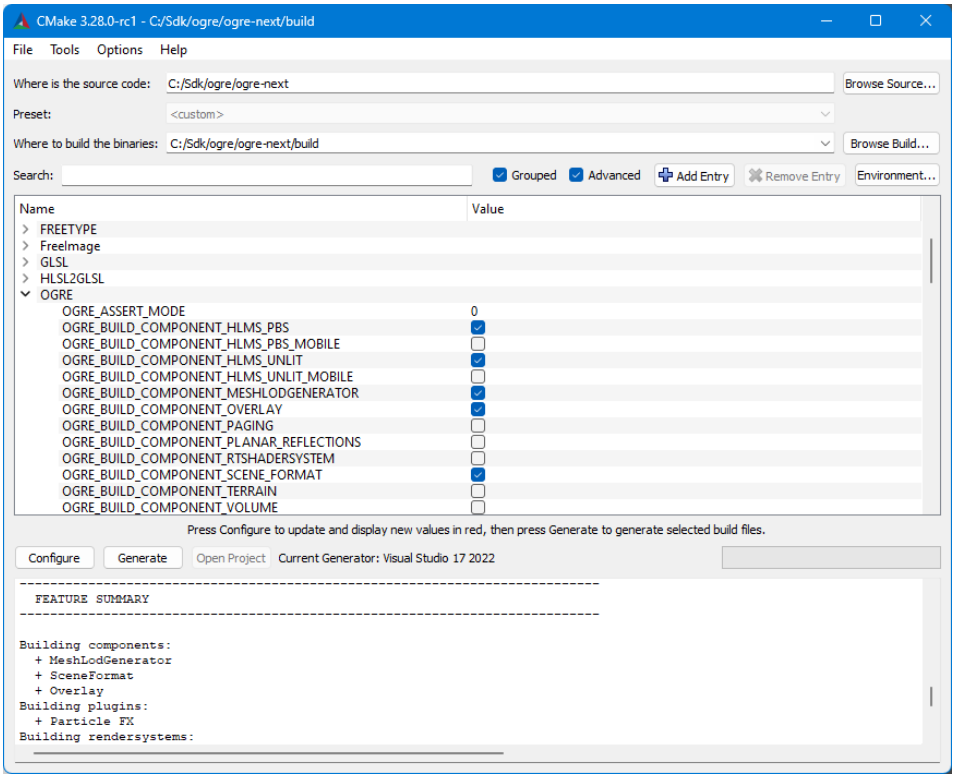
Ukázka skriptu, který stáhne a sestaví knihovnu *abc*:

```
git clone https://github.com/abc.git # stahne zdrojovy kod
mkdir build                          # vytvori slozku build
cd build                             # zmeni aktualni slozku na build
cmake ..                             # nakonfiguruje knihovnu
cmake --build .                      # sestavi knihovnu
cmake --build . --target install     # nainstaluje knihovnu
```

Větší a složitější knihovny, jako je například OGRE, vyžadují ke své funkci další knihovny, kterým se říká *závislosti* (anglicky *dependencies*). Všechny závislé knihovny je potřeba nejprve sestavit před samotnou kompilací potřebné knihovny. OGRE poskytuje balíček knihoven, na kterých závisí; ty se potom sestaví jako jeden celek výše uvedeným postupem. PhysX má svůj vlastní systém, který během sestavování potřebný software automaticky stáhne.

3.1.2. Konfigurace sestavovacího procesu

Konfigurační proces knihoven lze upravit podle vlastních potřeb nastavením konfiguračních možností. Například se tím specifikuje, jaké části knihovny se mají sestavit (pokud nejsou potřeba všechny), nebo kde se nacházejí předem sestavené závislosti. K zobrazení všech konfiguračních možností lze využít nástroj cmake-gui, který možnosti zobrazí v grafickém rozhraní.



Obr. 3: Ukázka konfigurace v nástroji cmake-gui

3.1.3. Sestavování mého projektu

Po sestavení knihoven bylo potřeba napsat CMake skript na sestavení mého projektu. Výchozím skriptem je soubor CMakeLists.txt v kořenovém adresáři projektu a popisuje postup, jak má být projekt sestaven.

Níže je zjednodušená ukázka CMake skriptu pro sestavení.

```
# Nastavi minimalni verzi CMake
cmake_minimum_required (VERSION 3.25...3.28)

# Specifikuje nazev, verzi a programovaci jazyk projektu
project("MetroSimulator3D" VERSION 0.0.1 LANGUAGES CXX)

# Specifikuje, ze vysledny soubor ma byt spustitelny program
add_executable(${PROJECT_NAME} WIN32)

# Prida soubory se zdrojovym kodem, ktere se maji sestavit
file(GLOB SOURCES CONFIGURE_DEPENDS "src/*.cpp" )
target_sources(${PROJECT_NAME} PRIVATE ${SOURCES})

# Zahrne cestu k hlavickovym souborum projektu
target_include_directories(${PROJECT_NAME} PUBLIC "include/")

# Najde potrebne knihovny
find_package(OgreNext REQUIRED COMPONENTS HlmsPbs HlmsUnlit Overlay)
find_package(PhysX REQUIRED COMPONENTS Extensions CharacterKinematic PvdSDK)
find_package(RmlUi REQUIRED)

# Pripoji knihovny k projektu
target_link_libraries(${PROJECT_NAME} OGRE::Ogre PHYSX::PhysX RmlUi::RmlUi)
```

3.2. Pohyb po kolejích

Základní funkcí vlakového simulátoru je pohyb po kolejích. V zásadě to znamená, že určitý objekt (vlak) se pohybuje po předem určené dráze. Dráha je definována množinou bodů v prostoru. Pohybu lze dosáhnout například těmito způsoby:

1. Využít detekci kolize a nechat vlak „sunout“ po dané dráze ve vymezeném prostoru
2. Interpolovat mezi traťovými body a v každém snímku posunout vlak směrem k dalšímu bodu o vzdálenost danou rychlostí a uplynulým časem

3.2.1. Pohyb vlaku pomocí detekce kolize

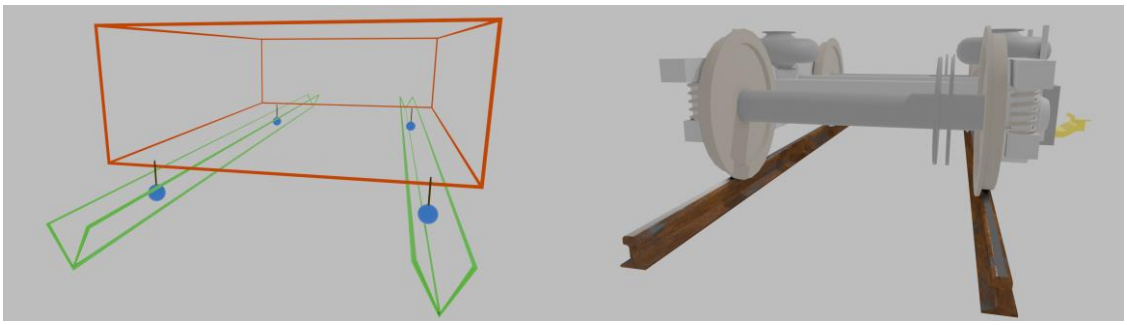
Tento způsob je velmi blízký realitě, nejprve je vygenerována kolej a na ní je následně přesně položen podvozek s koly. Na podvozky je připevněna skříň vlaku a dohromady tvoří jeden celek – vagón. Jednotlivé vagóny mohou být dále spojovány k sobě.

Tento systém pohybu je ale příliš realistický [11] a pro účely tohoto projektu se moc nehodí, protože koleje by musely být generovány velmi přesně, aby nedošlo k vykolejení (například u výhybek by toto byl problém). Navíc vzhledem k tomu, že aplikace má běžet v reálném čase, takové řešení by bylo výpočetně příliš náročné s vysokými nároky na hardware. Tímto způsobem řeší pohyb například hra Rigs of Rods.

Obr. 4: Snímek ze hry Rigs of Rods

Rozhodl jsem se tedy systém zjednodušit: místo kolejnice se vygeneruje jakýsi „žlábek“ ve tvaru „V“, ve kterém se budou pohybovat menší kuličky napevno připevněné k podvozku místo složitých kol. Uživatel rozdíl nepozná, protože graficky by to vypadalo identicky, pouze fyzikální simulace by byla jiná. Kuličky by se nemusely točit, stačilo by jim nastavit velmi malé tření, aby volně „klouzaly“ žlábkem.

Obrovskou výhodou této metody je, že jednotlivé části vlaku se spojí pomocí knihovny PhysX a ta veškerý pohyb vyřeší. Stačí pouze aplikovat zrychlení na lokomotivu nebo podvozky (závisí na typu vlaku) a celý vlak se bude rozjíždět najednou. Navíc při překročení maximální rychlosti v oblouku boční síly způsobí vykolejení vlaku, což dobře odpovídá realitě. Na obrázku níže je znázorněn tento koncept pohybu. Vlevo je ukázka toho, jak to „vidí“ knihovna PhysX, vpravo pak jak by to viděl uživatel na obrazovce.



Obr. 5: Vizualizace fyzikální simulace

Problém ale nastává při vytváření a umísťování vlaku. Podvozky musejí být přesně zarovnané s kolejemi, aby nedošlo hned k vykolejení. Umístění lze poměrně snadno provést na rovném úseku, ale v oblouku, jak se ukázalo, je to poměrně nespolehlivé. Problém by to byl například při umísťování AI vlaků (Artificial Intelligence; jsou to vlaky, které nejsou řízeny hráčem), protože ty se mohou umístit na náhodném místě při načtení.

Tím, že veškerý pohyb je řešen knihovnou PhysX, nelze kódem snadno zjistit, v jaké části tratě se vlak právě nachází. Nutné je to například při detekci, jestli je vlak ve stanici, pro přehazování výhybek nebo zobrazování polohy vlaku na přehledové mapě. Pro zjištění přesné pozice by bylo potřeba projít všechny body všech tratí a spočítat, který je nejbližší. Tento výpočet je však výpočetně velmi náročný a pro použití v reálném čase se nehodí.

3.2.2. Pohyb vlaku pomocí interpolace mezi body

U tohoto způsobu veškerý pohyb řeší můj vlastní algoritmus, který v každém snímku posune všechny podvozky a vagóny o danou dráhu. Dráha je dána vztahem $s = v * t$, kde v je rychlost vlaku a t je čas uplynulý od předchozího snímku. Mezi jednotlivými body kolejové dráhy lze interpolovat pomocí matematické funkce. Do této funkce se dosadí bod p a parametr t, který musí být v rozsahu 0-1 podle toho, kde přesně se objekt mezi dvěma sousedními body vyskytuje. Výstupem funkce je pozice v 3D souřadnicích.

V mém projektu jsou implementovány 3 typy interpolačních funkcí:

1. Lineární
2. Kruhová
3. Beziérova

Lineární funkce interpoluje podle vztahu

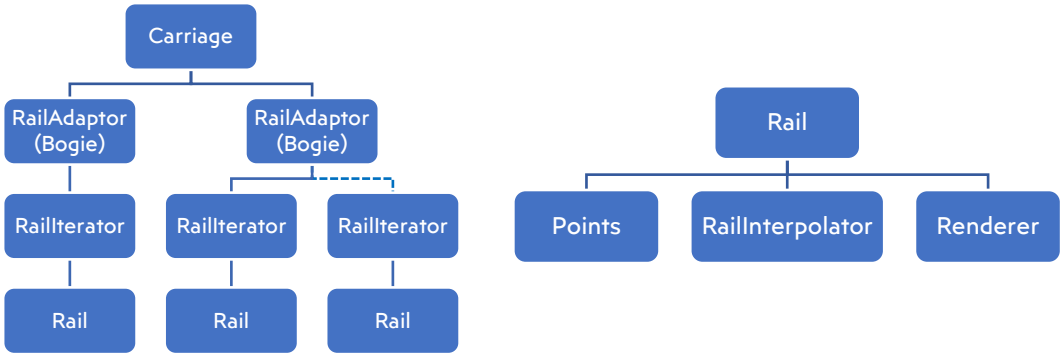
$$a + (b - a) * t$$

kde a je 1. bod, b je 2. bod a t je parametr v rozmezí 0-1.

Kruhová funkce využívá funkci $nlerp$, která bod vypočítá pomocí rotace po povrchu koule [12]. Beziérova funkce je definována několika takzvanými kontrolními body, které jsou vypočítány při načítání kolejí. Interpolace potom probíhá pomocí několika polynomů [13]. Náklon trati je mezi body vždy interpolován lineárně.

3.2.3. Implementace pohybového algoritmu

Implementace algoritmu je napsána pomocí několika C++ tříd. Třída Rail je objekt popisující trasu trati. Obsahuje jak množinu bodů, kudy trať prochází (`Points`), tak interpolační funkci (ta je implementována pomocí třídy `RailInterpolator`). Implementuje také funkce pro grafické vykreslování kolejí v 3D prostoru (`Renderer`), které jsou potom zobrazeny pomocí `StaticEntity`.



Obr. 6: Schéma tříd potřebných k pohybovému algoritmu

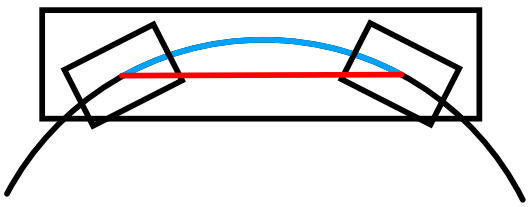
`RailAdaptor` je abstraktní třída (abstraktní třída je speciální typ třídy, která musí být zděděna jinou třídou, aby mohla být vytvořena [14]), kterou musí zdědit každý objekt, který se má pohybovat po kolejích. V mém projektu zatím jezdí po kolejích pouze podvozky, které jsou implementovány třídou `Bogie`. Aby však mohl vlakem cestovat hráč, je k tomu potřeba ještě jeden objekt – `Carriage`, což je samotná skříň vlaku. Ta může být připojena k jednomu nebo více podvozkům a stará se mimo jiné i o posouvání všech objektů, které jsou uvnitř.

Pohyb je řešen pomocí funkce `RailAdaptor::Translate`, která má jeden parametr – vzdálenost. Parametr t je vypočítán podílem vzdálenosti uražené od aktuálního bodu a celkové vzdálenosti mezi aktuálním a dalším bodem. Při přesažení této vzdálenosti se posune aktuální bod a je přepočítána vzdálenost k dalšímu bodu. Informace o aktuálním bodu obsahuje právě třída `RailIterator`. Pokud je aktuální bod poslední a je přesažen, proběhne vyhledání navazující koleje. Navazující kolej je taková kolej, která má svůj první bod přesně v místě posledního bodu aktuální koleje. Pokud navazující kolej není nalezena, vlak vykolejí.

Největší výhodou této metody je absolutní kontrola nad pohybem vlaku, takže ho lze snadno počítačově ovládat, rychle zjistit jeho polohu vůči trati, či ho směrovat pomocí výhybek. Pohyb je navíc díky interpolaci velmi hladký a přesný.

3.2.4. Synchronizace podvozků

Při pohybu obloukem nastává mírný problém: tím, že jsou všechny podvozky vlaku posouvány konstantní rychlostí, mění se jejich vzájemná poloha vůči skříni vlaku. Způsobeno je to tím, že podvozky by od sebe měly být vzdáleny přímou čarou určitou konstantní vzdáleností (červená čára), ale v oblouku jsou kvůli konstantní rychlosti vůči koleji vzdáleny o konstantní vzdálenost po koleji (modrá čára). Důsledkem toho se při vjezdu a výjezdu z oblouku podvozky pohybují vůči skříni vlaku. Vyřešit by se to muselo kompenzací rychlostí podvozků při průjezdu obloukem, což by byl poměrně komplikovaný algoritmus. Tento algoritmus jsem nyní neimplementoval a místo toho se soustředil na jiné funkce (např. náklon trati). Pro koncového uživatele je to celkem nepodstatný detail, který lze spatřit pouze při bližším zkoumání.



Obr. 7: Vizualizace délky oblouku

3.2.5. Zhodnocení obou pohybových metod

Po zvážení výhod a nevýhod obou metod jsem se rozhodl implementovat 1. metodu, mj. i pro výrazně jednodušší implementaci. Vše fungovalo relativně dobře do té doby, než jsem se pokusil rozjet vlak vyšší rychlostí (tj. více jak 140 km/h). V tu chvíli simulace přestala být stabilní a celý vlak se začal mírně třást. Vzhledem k tomu, že tento projekt by měl podporovat i simulaci vysokorychlostních vlaků, které jezdí i rychlostmi většími než 300 km/h, by toto třesení bylo nepřípustné. Důvodem byla nejspíš, ačkoliv se mi to nepodařilo prokázat, nedostatečná triangulace trati nebo omezená schopnost detekce kolize ve vysokých rychlostech v knihovně PhysX. Celý projekt jsem tedy předělal na 2. způsob, který byl na implementaci výrazně složitější, což mi zabralo poměrně dost času. Kvůli tomu mi do dalšího výstupu nezbyl čas například na lepší vlakovou dynamiku.

3.3. Načítání obsahu z datových souborů

Jedním z cílů projektu je, aby se veškerý grafický obsah načítal z externích souborů místo přímo z kódu (tzv. data-driven design). Rozhodl jsem se pro definice objektů ve virtuálním světě použít formát XML. Formát a strukturu těchto souborů jsem zvolil tak, aby byla co nejjednodušší a zároveň modulární. Tyto soubory se shromažďují do takzvaných balíčků, které se vždy načítají jako jeden celek. V budoucnu mám v plánu založit online platformu na uchovávání takovýchto balíčků, které půjdou stahovat přímo ze hry.

3.3.1. Popis formátu XML souborů

Každý XML soubor začíná standartní hlavičkou, která specifikuje verzi XML a kódování a obsahuje právě jeden kořenový tag (v ukázce **`<root>`**) [15].

```
<?xml version="1.1" encoding="UTF-8"?>
<!-- komentář -->
<root atribut="hodnota atributu">
    …obsah dokumentu
</root>
```

Kořenový tag může mít libovolný název, v mém formátu jsem si zvolil **`<file>`**, což označuje generický název pro soubor. Každý soubor musí v hlavním tagu **`<file>`** definovat atribut *type*, který specifikuje typ souboru.

Implementovány jsou zatím 2 typy souborů: *mapy* (**`type="map"`**) a *knihovny* (**`type="lib"`**). Mapy obsahují pouze definice objektů (entit) a tratí, které jsou potom umístěny do virtuálního světa. Knihovny obsahují definice **`Model`**ů, **`Material`**ů a **`Collider`**ů (souhrnně *prostředky*), které jsou používány definicemi entit v mapových souborech.

Načítání hry probíhá v několika fázích:

1. Načtení všech platných mapových XML souborů v balíčku. Během načítání definic jsou uloženy názvy všechny potřebných závislostí.
2. Načtení všech platných knihovnových XML souborů. Jsou načteny definice prostředků, které jsou používány objekty z předchozího kroku.
3. Vytvoření světa a načtení potřebných prostředků – z definic entit jsou vytvořeny instance, je vytvořen hráč, koleje, vlaky, …; jsou načteny modely a textury z balíčku.

3.3.2. Popis struktury definic objektů v XML souborech

Struktura XML souborů přímo odpovídá hierarchii herních objektů popsaných v teoretické části. Entita (každý objekt, který je součástí hry) je definována pomocí tagu `<entity>`. Atribut *type* může být buď *static* nebo *dynamic* podle toho, jestli má být vytvořena instance `StaticEntity` nebo `DynamicEntity`. Jak bylo popsáno v teoretické části, entita se skládá z modelu, collideru a transformace. Model se definuje pomocí tagu `<model>`, obdobně se definuje i `<collider>`. Transformace se definuje pomocí jednotlivých složek, pozice jako `<position>`, rotace jako `<rotation>` a měřítko jako `<scale>`. Modely a collidery se skládají z jednotlivých tvarů (`Shape`), kterých je k dispozici několik typů. Kvádr je definován tagem `<box>`, koule tagem `<sphere>` a uživatelem definovaný tvar jako `<mesh>`. V budoucnu bych chtěl přidat další implementace primitivních tvarů jako je válec (`<cylinder>`), kužel (`<cone>`), kapsule (`<capsule>`), rovina (`<plane>`) nebo billboard (`<billboard>`), což je 2D obdélník, který je vždy otočen směrem k hráči.

Každý z těchto tvarů obsahuje společné parametry, a sice transformaci a materiál. Transformace se definuje úplně stejně jako u entit. Materiál popisuje grafický vzhled objektu. Renderovací systém používá techniku zvanou Physically based rendering (PBR, fyzikálně založené vykreslování). Cílem PBR je dosažení fotorealističnosti vykreslované scény [16]. Základní barva objektu je popsána pomocí `<diffuse>`, odrazová barva pomocí `<specular>`. Jednotlivé složky barvy jsou popsány atributy *r*, *g* a *b*. Mezi další parametry patří emisní barva (`<emissive>`), hrubost povrchu (`<roughness>`), kovovost (`<metalness>`), index lomu (`<ior>`) a průhlednost (`<transparency>`). Jako vstup těchto parametrů může být použita i textura (`<texture>`), kde atribut *src* specifikuje cestu k souboru s bitovou mapou.

Každý typ tvaru zároveň musí obsahovat parametry pro daný typ, například pro kvádr jsou to jeho rozměry `<size>`, pro kouli je to poloměr (`<radius>`) a pro `<mesh>` je to cesta k souboru s 3D tvarem (atribut *src*).

Složitější prostředky jako je `Model`, `Collider`, `Material` a `Mesh` lze definovat zvlášť v knihovnovém (lib) souboru, aby se zachovala přehlednost mapového souboru. Při definici těchto prostředků se uvede jejich jméno (atribut *name*), které musí být pro tento typ prostředku v celém balíčku jedinečné. Použití této definice se potom definuje speciálním

atributem *use*, který v podstatě „najde“ a „importuje“ definici z jiného souboru. Jedna z variant definice entity by mohla vypadat například takto:

```
<entity type="static">
    <position x="10" y="0" z="10" />
    <rotation yaw="90" />
    <model>
        <box>
            <size x="1" y="1" z="1" />
            <position x="1" y="5" z="0" />
            <material>
                <diffuse r="1" g="0" b="0"/>
            </material>
        </box>
        <mesh src="ukazkovy-objekt.mesh">
            <rotation pitch="45"/>
        </mesh>
    </model>
    <collider use="object-collider" />
</entity>
```

Pokud by byla tato definice umístěna do mapového souboru, ve hře se vykreslí entita na souřadnicích [10, 10] (y je výška), která se skládá z červené krychle s délkou hrany 1 metr a 45 stupňů podle osy x otočeného tvaru, který je uložen v souboru "ukazkovy-objekt.mesh". Collider bude načten z jiného souboru. Ten by mohl obsahovat například toto:

```
<?xml version="1.1" encoding="UTF-8"?>
<file type="lib">
    <collider name="object-collider">
        <box>
            <size x="5" y="10" z="5" />
        </box>
    </collider>
</file>
```

Znamenalo by to, že entita z předchozí ukázky by byla ohraničena 10 metrů vysokým kvádrem, který by sloužil jako neprůchodná (ale neviditelná) zeď.

Tag **<player>** slouží k definici počáteční pozice a vzhledu hráče. Definuje se stejně jako entita.

Železniční tratě jsou definovány tagem **<track>**, který může obsahovat jednu nebo více kolejí (**<path>** nebo **<circle>**). Rovná trať se definuje pomocí dvou bodů, složitější křivky potom více body. Každý bod odpovídá jednomu **<point>**, kde atributy *x*, *y* a *z* definují souřadnice a *cant* definuje náklon trati ve stupních. Kladná hodnota naklápí koleje po směru

hodinových ručiček. Oblouk je definován pomocí **`<circle>`**, kde atribut *radius* značí poloměr a *turn* směr oblouku (*left* doleva, *right* doprava). Obsahovat musí právě dva body, které definují počáteční a koncový bod oblouku.

3.4. Uživatelské rozhraní

Uživatelským rozhraním je myšlen systém, který zprostředkovává uživatelský vstup s aplikací. Místo toho, aby se hráč objevil po spuštění hry přímo ve virtuálním světě, je nejdříve zobrazena uvítací obrazovka (*splash screen*). Během jejího zobrazení se na pozadí načítají soubory potřebné k běhu hry.

Po načtení hry je uživateli zobrazeno hlavní menu, ze kterého se může rozhodnout, co bude dál dělat. Může buď přímo vstoupit do hry, nebo se nejdříve podívat, jak se má hra ovládat. V budoucnu bych chtěl přidat obrazovku s nastavením hry a manažer obsahu, přes který půjde stahovat balíčky 3D obsahu.

3.4.1. Integrace knihovny RmlUi

Uživatelské rozhraní jsem implementoval pomocí knihovny RmlUi. Ta k vykreslení používá zjednodušený systém HTML a CSS, které jsou známé z internetového prostředí (používají se v prohlížečích). Pomocí tohoto systému lze relativně snadno vytvořit komplexní uživatelské rozhraní, které splňuje cíle data-driven systému (tj. veškerý obsah je načítán z externích souborů).

Integrace této knihovny do mého projektu ale nebyla vůbec jednoduchá (ač se to ze začátku zdálo jako triviální úkol). RmlUi se integruje pomocí takzvaných rozhraní (*interface*), které musí aplikace používající knihovnu implementovat [17]. Taková rozhraní jsou celkem čtyři:

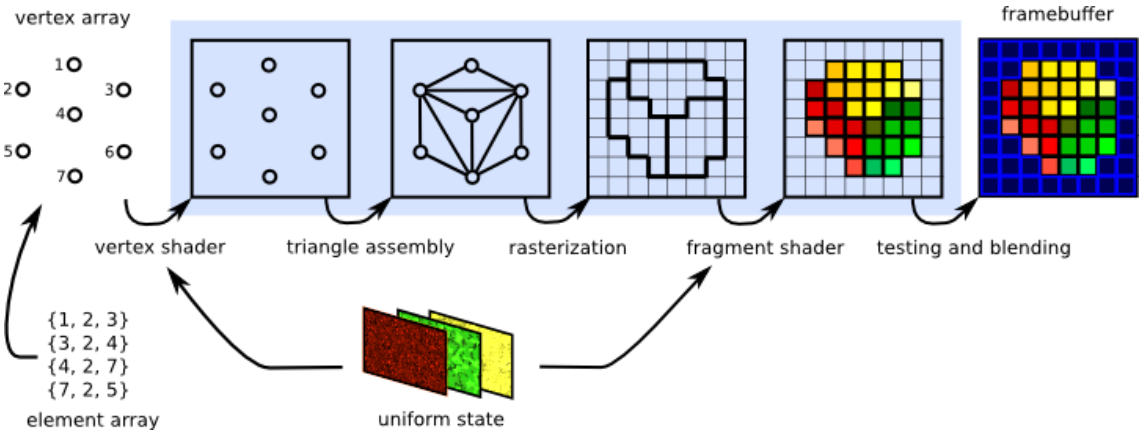
1. Systémové rozhraní
2. Renderovací rozhraní
3. Souborové rozhraní
4. Fontové rozhraní

Implementovat systémové rozhraní bylo jednoduché: stačilo implementovat funkci pro získání času uplynulého od spuštění aplikace a funkce pro změnu vzhledu kurzoru. Souborové rozhraní bylo taktéž jednoduché, ke čtení souborů jsem využil souborové funkce knihovny OGRE. Fontové rozhraní nebylo potřeba implementovat, protože RmlUi poskytuje výchozí implementaci, která pro účely tohoto projektu postačuje. Nejnáročnější tedy bylo implementovat renderovací rozhraní. RmlUi sice obsahuje implementace tohoto rozhraní pro běžné vykreslovací systémy jako je OpenGL nebo Direct3D, jenomže můj

projekt pro vykreslování používá knihovnu OGRE. Po důkladném prohledání Internetu jsem našel dva užitečné zdroje, podle kterých bych mohl implementaci napsat. První z nich je zdrojový kód knihovny libRocket [18], což je již neudržovaný předchůdce RmlUi. libRocket obsahuje i ukázkovou implementaci pro OGRE, jen je určena pro velmi starou verzi OGRE, takže v mém případě kód nefungoval. Druhým velmi užitečným zdrojem informací jsou samotná uživatelská fóra knihovny OGRE, kde se jiný uživatel také pokoušel implementaci napsat [19].

3.4.2. Implementace renderovacího rozhraní

Pro napsání implementace jsem si musel prostudovat literaturu pojednávající o funkci grafických karet; mimo jiné pochopit, jak funguje grafický řetězec (*rendering pipeline*), což je sekvence procesů, které je potřeba provést pro vykreslení něčeho na obrazovku. Výrazně mi k tomu pomohl tutoriál Learn OpenGL [20], který vše od základu vysvětluje. Tutoriál je ale poměrně dlouhý a náročný na pochopení, takže si to vyžádalo mnoho času a úsilí.



Obr. 8: Schéma funkce grafického řetězce [21]

Procesy grafického řetězce probíhají přímo na grafické kartě počítače (GPU) a jsou řízeny takzvanými *shadery*. Shader je speciální druh počítačového programu, který běží na grafické kartě. Zde nebudu koncept popisovat, protože je příliš složitý na popsání v několika větách.

Dalším krokem je výpočet projekční transformační matice, pomocí které se jednotlivé prvky uživatelského rozhraní vykreslí ve 2D prostoru na obrazovku. Pro vykreslování 2D grafiky je vhodná ortografická projekce. Pro zobrazení něčeho na obrazovce je potřeba vykonat jednotlivé renderovací příkazy, které grafiku vykreslí. Ty se zprostředkovávají knihovnou OGRE. Protože tohle není standartní použití této knihovny, není k tomuto procesu

prakticky žádná dokumentace. Většinu jsem se musel naučit z výše zmíněného fóra nebo přímo ze zdrojového kódu knihovny, který je poměrně komplikovaný.

K funkci vykreslování grafického rozhraní bylo potřeba napsat dva druhy shaderů. Jeden vrcholový (*vertex shader*), který umísťuje vrcholy objektů na správné místo na obrazovce pomocí projekční matice, a dva pixelové (též fragmentové) shadery, které určují výslednou barvu každého pixelu. Pixelové shadery jsou v mém projektu dva, první pro vykreslování s texturou a druhý pro renderování bez textury.

Ukázka vertex shaderu:

```
#version 330 core

uniform vec2 _translate;
uniform mat4 _transform;

layout (location = 0) in vec2 position;
layout (location = 1) in vec4 colour;
layout (location = 2) in vec2 uv;

out vec2 fragUvCoord;
out vec4 fragVertexColor;

void main() {
    fragUvCoord = uv;
    fragVertexColor = colour;

    vec2 translatedPos = position + _translate;
    vec4 outPos = _transform * vec4(translatedPos, 0, 1);

    gl_Position = outPos;
}
```

Tento program umístí každý vrchol vykreslované geometrie na správné místo na obrazovce pomocí projekční matice (`_transform`) a pošle barvu vrcholu (`colour`) a souřadnice textury (`uv`) dále do pixelového shaderu.

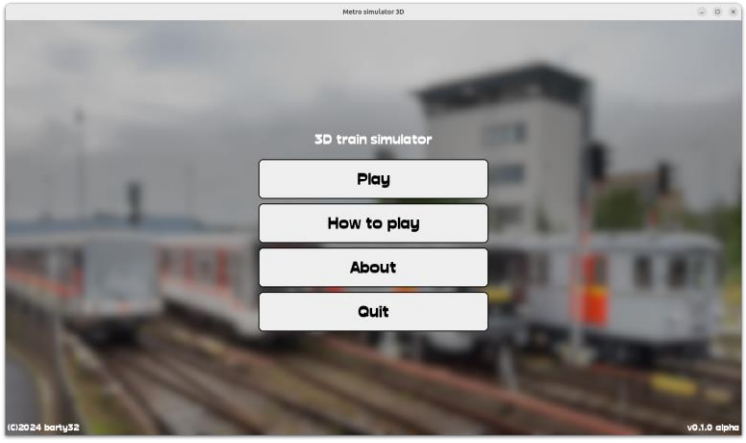
Pro úplnou funkčnost renderovacího rozhraní bylo potřeba ještě implementovat načítání textur z disku do GPU, k tomu opět posloužily souborové funkce systému OGRE.

3.4.3. Části uživatelského rozhraní

Implementovány jsou celkem 3 části uživatelského rozhraní:

1. Hlavní menu
2. Stránka s instrukcemi pro hraní hry
3. Stránka s informacemi o projektu

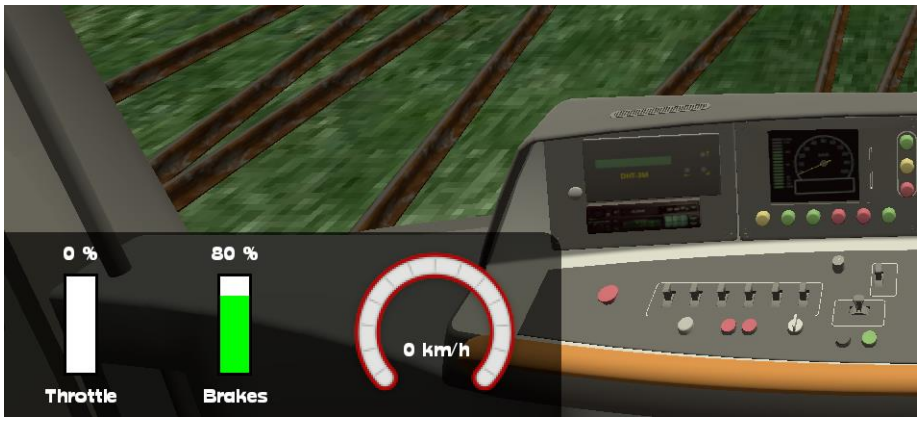
Hlavní menu slouží jako takový rozcestník; uživatel se může rozhodnout, co bude dál dělat. Tlačítkem Play se dostane přímo do hry. Před hraním je obvykle vhodné si přečíst, jak se hra ovládá, instrukce se zobrazí stisknutím tlačítka How to play. Stránka About obsahuje informace o aktuální verzi aplikace, autora projektu a použité softwarové knihovny. V budoucnu mám v plánu přidat obrazovku s nastavením hry a rozhraní pro stahování a načítání balíčků s obsahem.



Obr. 9: Hlavní menu

3.4.4. Herní menu

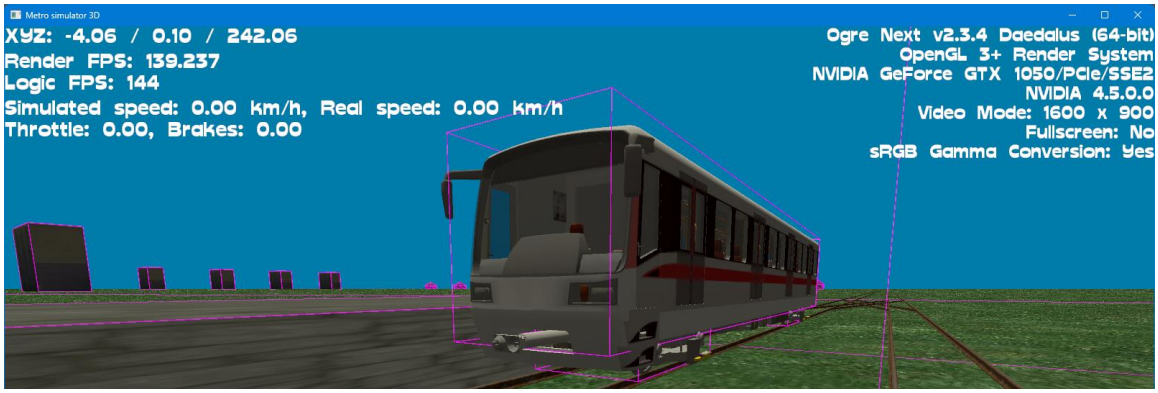
Pokud ve hře hráč nastoupí do vlaku, v levém dolním rohu obrazovky se zobrazují informace o vlaku (rychlost, stav motorů a brzd). Tento styl rozhraní se nazývá heads-up display (HUD) [22]. V mém projektu toto menu není interaktivní, pouze zobrazuje aktuální stav v reálném čase.



Obr. 10: HUD menu při pohledu z kabiny

3.4.5. Ladící menu

Po stisknutí klávesy F3 se ve hře zobrazí ladící menu. Zobrazuje informace o aktuální pozici hráče, počty snímků za sekundu a různé údaje o renderovacím systému, jako je typ grafické karty, verze ovladačů a použité renderovací API. Růžovými čarami jsou vyznačeny obrysy objektů simulovaných knihovnou PhysX. Toto menu je určené hlavně pro vývoj projektu a k řešení případných problémů.

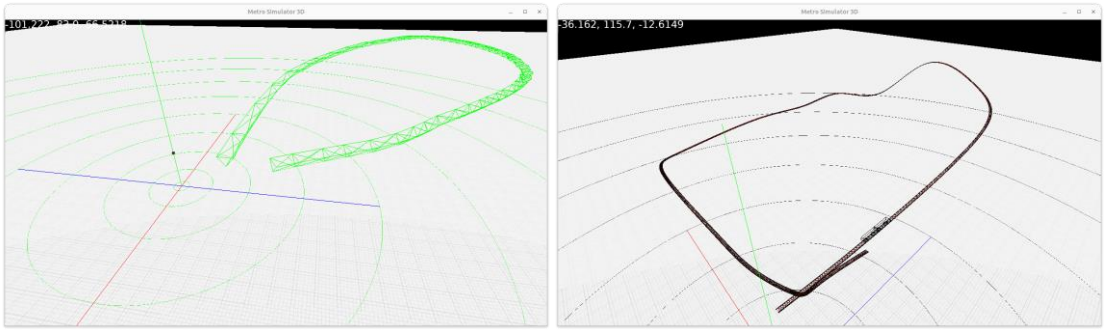


Obr. 11: Ladící menu

4. Výsledky

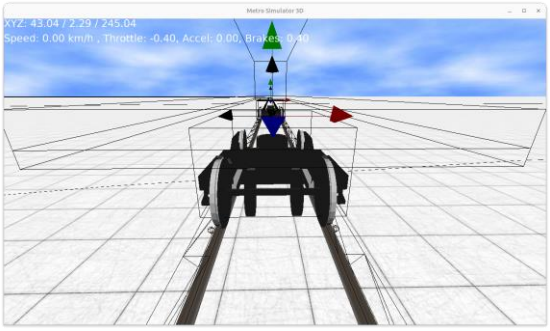
Výsledkem práce je funkční vlakový simulátor, který načítá 3D data z externích souborů. Nakonec se povedlo implementovat všechny čtyři části projektu. Tato kapitola shrnuje celý vývoj několika snímky obrazovky.

Na následujících snímcích jsou vidět první pokusy o generování průjezdné trasy (říjen 2023).

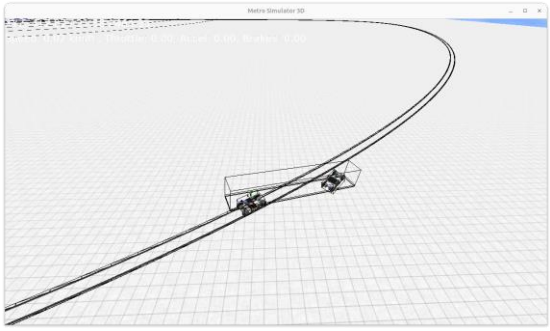


Obr. 12: První pokusy o generování průjezdné trasy

Níže jsou výsledky simulace fyzikální metodou (listopad 2023), vpravo je vidět problém s „pokládáním“ vlaku v oblouku – vlak okamžitě vykolejil.

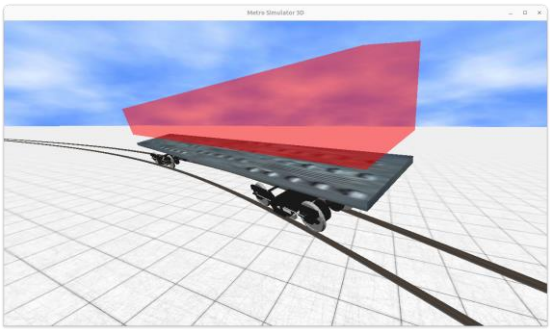


Obr. 13: Podvozek s vizualizací fyzikálně simulovaných částí

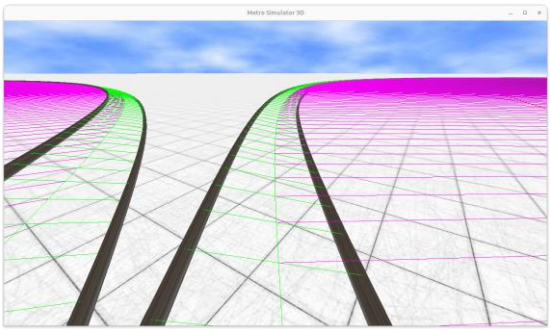


Obr. 14: Vykolejený vlak

V prosinci 2023 probíhal vývoj interpolační metody pohybu vlaku a generování oblouku kolejí s náklonem. Červená část znázorňuje oblast, ve které je detekována přítomnost hráče.

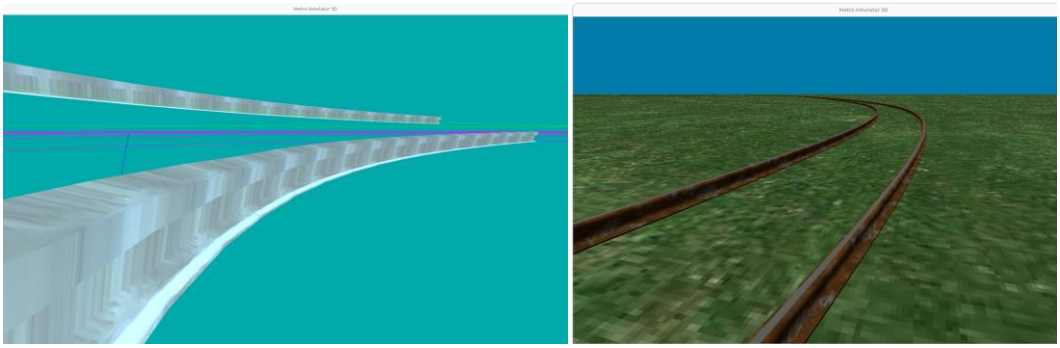


Obr. 15: Simulace vlaku pomocí interpolační metody



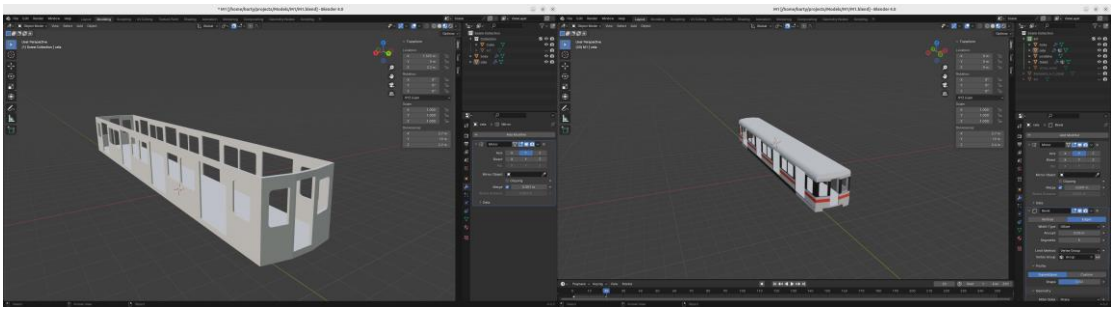
Obr. 16: Vizualizace středu kružnice

V další části jsem se zaměřil mimo jiné na lepší vzhled generovaných kolejí (leden 2024); snímek vpravo byl pořízen o 2 měsíce později, když už byla napsána implementace načítání dat ze souborů.



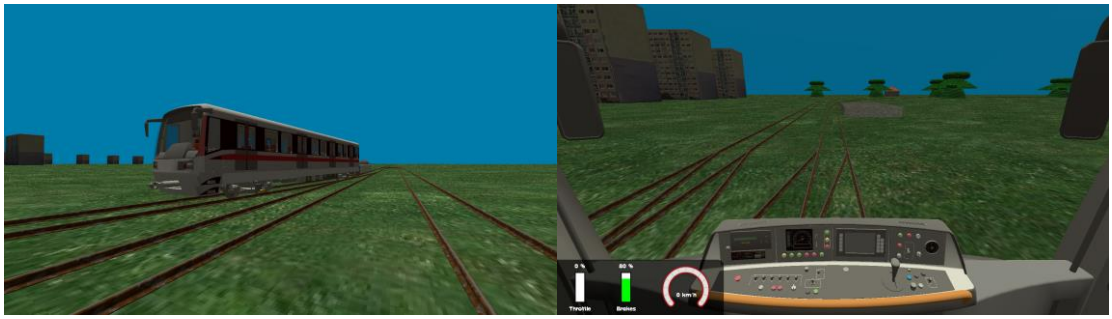
Obr. 17: Pokrok v generování kolejí

Důležitým krokem bylo namodelování samotných 3D modelů (únor 2024). Modely jsem vytvářel v programu Blender.



Obr. 18: Ukázka modelování v Blenderu

Poslední snímky zachycují dokončený projekt, pohled z venku a z kabiny (březen 2024). 3D prostředí by mohlo vypadat lépe, ale hlavní cíl tohoto projektu byla tvorba aplikace, která data umí načíst ze souborů. S dostatkem času a podkladů pro tvorbu 3D modelů by bylo snadné je do hry přidat.



Obr. 19: Náhled dokončené hry

5. Závěr

V rámci této práce se mi podařilo úspěšně implementovat všechny čtyři navržené části projektu, ačkoliv ne vše proběhlo zcela bez problémů. Engine se mi úspěšně podařilo sestavit jak pro Linux, na kterém jsem software vyvíjel, tak pro Windows. Pro ostatní operační systémy jsem se projekt zatím nepokoušel sestavit, protože by to vyžadovalo složitější úpravy v sestavovacím systému. Co se týče pohybu vlaku po kolejích, po experimentech s různými metodami jsem dosáhl uspokojivých výsledků s jednou z nich, zatímco druhá metoda se ukázala pro potřeby tohoto projektu jako nedostačující. Načítání XML souborů se také podařilo zprovoznit i přes větší úpravy v jádru enginu. Uživatelské rozhraní je sice zatím v základní formě, ale jeho nejnáročnější část je již implementována, takže další rozšíření už by mělo být snadné.

Celý projekt poskytuje robustní základ pro rozvoj plnohodnotného vlakového simulátoru. Architektura projektu je navržena tak, aby bylo snadné ho rozšířit o další funkce. Mezi další klíčové funkce patří například správa balíčků s 3D obsahem, generování terénu, přidání zvukových efektů, vylepšení ovládání vlaku a podpora pro další operační systémy včetně mobilních platforem. Tato práce tedy nejen že posouvá moje technické dovednosti a znalosti v oblasti programování 3D aplikací a modelování, ale také přispívá k rozvoji digitálních simulátorů vzdělávacího a zábavního charakteru.

6. Zdroje

[1] Herní engine. *Wikipedia.* [Online] https://cs.wikipedia.org/wiki/Hern%C3%AD_engine.

[2] Axon, Samuel. ARS Technica. *The end of OpenGL support, plus other updates Apple didn't share at the keynote.* [Online] 6. 6. 2018. https://arstechnica.com/gadgets/2018/06/the-end-of-opengl-support-other-updates-apple-didnt-share-at-the-keynote/.

[3] MoltenVK. [Online] https://moltengl.com/moltenvk/.

[4] *OGRE - Open Source 3D Graphics Engine.* [Online] Torus Knot Software Ltd. https://www.ogre3d.org/.

[5] Goldberg, Matias N. Ogre 2.1 FAQ. *OGRE Wiki.* [Online] https://wiki.ogre3d.org/tiki-index.php?page=Ogre+2.1+FAQ.

[6] *PhysX Documentation.* [Online] Nvidia, 12/2023. https://nvidia-omniverse.github.io/PhysX/physx/5.3.1/index.html.

[7] C++ Namespaces. *C++ reference.* [Online] https://en.cppreference.com/w/cpp/language/namespace.

[8] The Core Objects. *OGRE Documentation.* [Online] https://ogrecave.github.io/ogre/api/latest/_the-_core-_objects.html.

[9] Plášek, Otto. Konstrukční uspořádání koleje. [Online] https://www.fce.vutbr.cz/zel/plasek.o/studium/2_Prevyseni_a_vzestupnice.pdf.

[10] CMake. [Online] Kitware. https://cmake.org/.

[11] Simpson, Michael David. *Real-Time Simulation of Rail Vehicle Dynamics.* 9/2016 https://core.ac.uk/download/pdf/153780823.pdf.

[12] Ogre::Quaternion Class Reference. *OGRE Documentation.* [Online] https://ogrecave.github.io/ogre/api/latest/class_ogre_1_1_quaternion.html#a16987149991882e148170fe3f1c7da4b.

[13] Smooth Bézier Spline Through Prescribed Points. *Particle in Cell Consulting.* [Online] 17. 6. 2012. https://www.particleincell.com/2012/bezier-splines/.

[14] Abstract class. *C++ reference.* [Online] https://en.cppreference.com/w/cpp/language/abstract_class.

[15] Extensible Markup Language (XML). [Online] https://www.w3.org/XML/.

[16] Physically based rendering. *Wikipedia.* [Online] https://en.wikipedia.org/wiki/Physically_based_rendering.

[17] Ragazzon, Michael R. P. Custom interfaces. *RmlUi Documentation.* [Online] https://mikke89.github.io/RmlUiDoc/pages/cpp_manual/interfaces.html.

[18] Wimsey, David. libRocket. [Online] https://github.com/libRocket/libRocket.

[19] Ogre3d 2.1 Ported LibRocket. *Ogre Forums.* [Online] 16. 3. 2016. https://forums.ogre3d.org/viewtopic.php?t=85318.

[20] Vries, Joey de. *Learn OpenGL.* [Online] 2014. https://learnopengl.com/.

[21] Graphics Pipeline. [Online] https://graphicscompendium.com/intro/figures/graphics-pipeline.png.

[22] HUD (video games). *Wikipedia.* [Online] https://en.wikipedia.org/wiki/HUD_%28video_games%29.