STŘEDNÍ ŠKOLA TECHNICKÁ A EKONOMICKÁ BRNO,
OLOMOUCKÁ, PŘÍSPĚVKOVÁ ORGANIZACE

MATURITNÍ PRÁCE

2D Physics Engine v C++

Brno 2025 Aleš Gabrhelík

Čestné prohlášení

Prohlašuji, že předložená maturitní práce pod názvem „2D Physics Engine v C++“ je ve smyslu autorského zákona výhradně mým autorským dílem. Maturitní práci jsem vypracoval samostatně s použitím uvedené literatury a dalších uvedených zdrojů na základě konzultací s vedoucím maturitní práce.

Datum podpis

Poděkování

Rád bych poděkoval vedoucí maturitní práce Mgr. Janě Morbacherové za odborné vedení, cenné rady a vstřícnost při konzultacích a celkovém vedení práce.

SŠTE BRNO
OLOMOUCKÁ

ZADÁNÍ MATURITNÍ PRÁCE

Školní rok: 2024/2025

Obor: 18-20-M/01 Informační technologie

Žák: Aleš Gabrhelík

Téma maturitní práce: 2D Physics Engine v C++

Vedoucí maturitní práce: Mgr. Jana Morbacherová

Cíle maturitní práce: detekce kolizí a pohyb těles, gravitace a složitější pohyb těles, demo ukázky fyzikálních experimentů a jevů

Harmonogram:

Zadání práce:	do 30.09.2024 (včetně)
Hodnocená konzultace č.1	do 15. 11. 2024 (včetně)
Hodnocená konzultace č.2	do 19. 12. 2024 (včetně)
Hodnocená konzultace č.3	do 07. 02. 2025 (včetně)
Odevzdání maturitní práce (nejpozději do):	do 12:00, 28. 02. 2025

Obsah, členění a rozsah maturitní práce:

1. Titulní list
2. Čestné prohlášení
3. Poděkování
4. Zadání práce
5. Abstrakt
6. Obsah
7. Úvod – vymezení tématu a cílů práce
8. Vlastní text práce (v rozsahu minimálně 16 stran textu)
9. Závěr
10. Seznam použité literatury
11. Seznam použitých značek a symbolů
12. Seznam použitých softwarů
13. Seznam použitých obrázků
14. Seznam příloh
15. Přílohy

Způsob vyhotovení práce:

1. Práci odevzdejte v jednom vázaném vyhotovení a elektronické podobě garantovi ZMP do termínu odevzdání
2. Práce musí být vyvázána (např. kroužková vazba, hůlková vazba)
3. Elektronická verze musí být nahrána na server školy do termínu odevzdání práce a musí být shodná s vyvázanou prací

Kritéria hodnocení práce:

1. Závěrečná maturitní práce (dále jen ZMP) je v souladu s vyhláškou č. 177/2009 Sb. O bližších podmínkách ukončování vzdělávání ve středních školách maturitní zkouškou ve znění pozdějších předpisů.
2. Závěrečnou maturitní prací žák prokazuje úroveň osvojených praktických dovedností ve svém oboru vzdělávání.
3. Neodevzdá-li žák práci ve stanoveném termínu, bude tato práce hodnocená stupněm „nedostatečná".
4. Body z obhajoby jsou přičteny k bodům od oponenta a vedoucího práce a jsou součástí hodnocení
5. Bude-li obhajoba ZMP hodnocena méně než deseti body, bude práce považována za neobhájenou a klasifikována stupněm „nedostatečná"
6. V případě výjimečného stavu může ředitel školy rozhodnout o zrušení obhajoby ZMP. V takovém případě se body z obhajoby k hodnocení nepřipočítávají a hodnocení práce je provedeno podle tabulky: Ukončení bez obhajoby maturitní práce

Kritérium	Vedoucí práce	Oponent
Vlastní téma práce	0 - 2 body	x
Splnění cíle zadání	0 - 10 bodů	0 - 10 bodů
Kvalita výsledné práce (kvalita dokumentace, přínos, praktické využití, funkčnost řešení atp.) a formální stránka práce	0 - 24 bodů	0 - 24 bodů
Přístup autora k řešení (využití odborné literatury, tvůrčí přístup atp.)	0 - 12 bodů	x
Účast na konzultacích	0 - 30 bodů	x
Obhajoba ZMP	0 - 14 bodů	0 - 14 bod2

Hodnocení s obhajobou

Celkový počet bodu	Výsledné hodnocení
130-140 bodů	Výborný
119-129 bodů	Chvalitebný
103-118 bodů	Dobrý
87-102 bodů	Dostatečný
0-86 bodů	Nedostatečný

Hodnocení bez obhajoby

Celkový počet bodu	Výsledné hodnocení
102-112	Výborný
91-101	Chvalitebný
75-90	Dobrý
59-74	Dostatečný
0-58	Nedostatečný

Ředitel školy

Mgr. Jana Mertlíková — Vedoucí práce

Podpis studenta

Anotace

Cílem práce je implementace aplikace jednoduché fyzikální simulace. Program simuluje ve dvoudimenzionálním prostoru dynamiku těles, detekci a řešení kolizí, podporuje dva tvary (obdélník a kruh). Součástí aplikace je také GUI, které podporuje editaci a ukládání scén, interakce uživatele se simulací a nastavení fyzikálních parametrů simulace.

Obsah

1. Úvod

Příchod počítačů ve 40. letech minulého století umožnil vznik nového oboru počítačových simulací. Zpočátku šlo o velké armádní projekty s ohromnými rozpočty a předními vědci. Za vůbec první počítačovou simulaci jsou považovány výpočty balistických křivek, které americká armáda prováděla na legendárním počítači ENIAC. S rozvojem výpočetní techniky se počítače začaly dostávat i do obyčejných domácností, a tím se zvyšoval zájem o počítačovou zábavu, jako jsou videohry nebo počítačově generované filmy. Tyto aplikace vyžadovaly reálné chování objektů, což vedlo k vývoji specializovaných systémů nazývaných „physics engine“. Physics enginy využívají poznatky z vědeckých počítačových simulací, aby docílily napodobení fyzikálního chování v počítačovém prostředí.

Physics engine má typicky tří klíčové funkce: dynamiku těles, detekce kolizí mezi objekty a věrohodné řešení kolizí. Maturitní práce se zaměřuje na vytvoření physics enginu s těmito základními funkcemi. Téma práce je pro mě zajímavé, protože spojuje mé dva největší zájmy, fyziku a programování. Hlavním cílem práce je implementovat physics engine ve dvoudimenzionálním prostoru, který bude schopen simulovat pohyb a interakce tuhých těles.

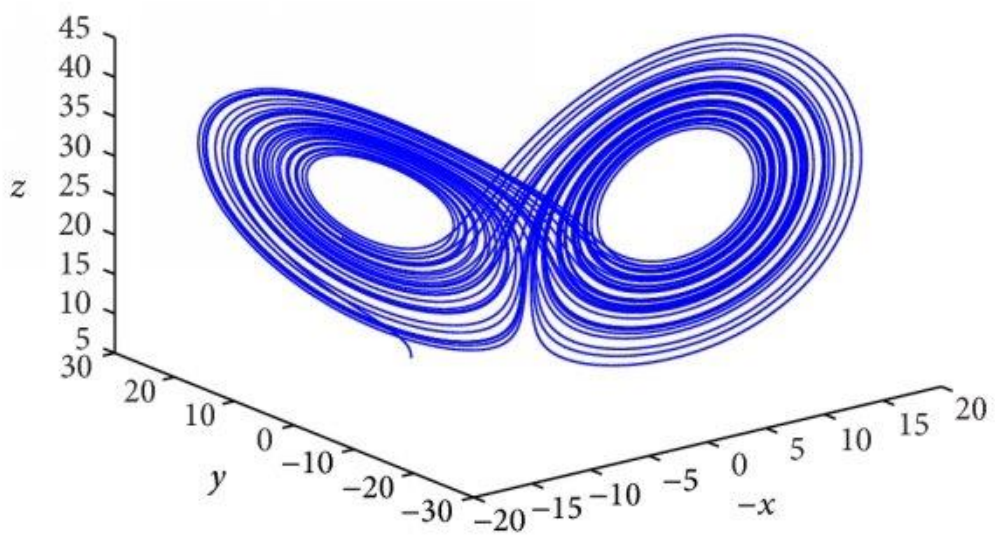
Práce je členěna na dvě části, v teoretické části je popis obecného physics enginu a matematických metod, které používáme k implementaci jeho klíčových funkcí. Více detailněji jsou popsány metody, které jsem následně implementoval v praktické části, ostatní metody jsou zde pro ucelení a porovnání přístupů. Praktická část práce se věnuje popisu struktury a architektury projektu, a implementaci samotného physics enginu, který jsem nazval „Fyzix“. Toto označení slouží pro zjednodušení textu a usnadnění orientace v něm. Dále se praktická část zabývá výhodami a nevýhodami jazyka C++ a knihoven, které projekt používá.

2. Teoretická část

2.1 Physics engine

Systém, který aproximálně simuluje fyziku, se odborně nazývá physics engine, jeho využití se liší podle jeho komplexity. Jednodušší physics enginy jsou převážně využívány ve videohrách, kde potřebujeme napodobovat chováni reálného světa, ale můžeme je využít i například v počítačové grafice pro animaci fyzikálních jevů nebo ve vzdělávání pro demonstraci fyzikálních zákonů.

Systémy pro simulace ve vědecké oblasti jsou označovány pojmem „physics engine“ pouze okrajově, ale fungují na podobných principech jako physics enginy ve videohrách. Více se setkáme s termínem „počítačové modelování“, které je využíváno v širokém spektru oborů. Ve fyzice se hlavně využívá v oblastech, ve kterých nám teorie neposkytuje přesná řešení nebo kdy je analytický výpočet moc složitý. Typicky se jedná o chaotické nebo stochastické systémy, konkrétně např. v poli deterministického chaosu jde o simulace více kyvadlových systémů nebo simulace turbulencí [1]. Dále jsou užitečné v oblastech, kde provedení experimentu je obtížné, nemožné nebo finančně náročné, jedná se např. o astrofyziku (orbitální pohyby, modelování černých děr a gravitačních vln), dynamika kapalin a plynů (meteorologie, aerodynamika aut a letadel), jaderná a částicová fyzika (modelování jaderných reakcí v extrémních podmínkách, simulace interakce částic).



Obrázek 1: Lorentzův atraktor [7]

Kromě využití můžeme physics enginy rozdělit na dvě třídy, simulace v reálném čase (real-time) a simulace s vysokou přesností (high precision). Real-time physics enginy jsou hlavně využívány ve videohrách, jejich výpočty jsou často zjednodušené, jelikož mají omezený čas na výpočet každého kroku v simulaci. High precision physics enginy slouží k simulacím, kde nejdůležitější je přesnost simulace a není omezena časem na výpočet, jedná se o vědecké simulace nebo o simulace pro počítačovou grafiku (např. CGI ve filmech). Simulace s vysokou přesností jsou počítány na výpočetních clusterech, které mají vysoký výpočetní výkon díky paralelizaci a jiným optimalizacím, zatím co real-time simulace jsou dělány pro běžné počítače.

2.1.1 Game physics engine

Ve většině videohrách je hratelnost a plynulost více důležitá než přesnost simulace. Proto jsou herní physics enginy navrhovány tak, aby působily fyzikálně věrohodným chováním, ale zdaleka ne přesným. Mezi nejpopulárnější physics enginy v této kategorii patří PhysX (od Nvidia) a open-source projekty jako Bullet, Box2D, Chipmunk.

Videoherní physics enginy dále můžeme rozdělit podle implementace na dvě kategorie: impulse-based a constraint-based. Impulse-based physics enginy používají pro pohyb objektů integrační metody a při kolizi na objekty aplikují impulsy, které změní jejich fyzikální parametry. Constraint-based enginy mají mezi objekty definované tzv. constraints (omezení chování objektu), pro lepší představu se může jednat např. o omezení vzdálenosti dvou objektů nebo jejich úhlu. K tomu, aby constraints byly splněny, se používají iterativní metody, které hledají fyzikální veličiny (poloha, rychlost atd.) vyhovující pro dané omezení. Constraint-based enginy jsou matematicky i implementačně mnohem komplexnější, dosahují vyšší přesnosti a jejich hlavní výhodou je schopnost simulovat přesné vztahy mezi objekty, jako je např. pružina, kloub nebo kyvadlo. Způsoby implementací se dají i kombinovat, a tak můžeme eliminovat slabé stránky jednotlivých metod. Jelikož Fyzix je impulse-based engine, v dalších kapitolách budu mluvit o metodách, které jsou využívány u impulse-based enginů.

2.2 Dynamika těles

Pro dynamiku těles je potřeba každému tělesu přiřadit základní fyzikální veličiny pro translaci (posuvný pohyb): pozice, rychlost, zrychlení, a fyzikální veličiny pro rotaci: úhel natočení (značený *théta* θ), úhlová rychlost (značená *omega* ω), úhlové zrychlení (značené

epsilon ε), moment setrvačnosti. Víme, že pokud je zrychlení nerovnoměrné, neznáme analytické řešení pohybových rovnic, proto musíme pohybové rovnice aproximovat pomocí numerických metod.

Jedna z hlavních komponent physics enginu je tzv. kroková funkce (step function), která využívá numerické metody pro aproximaci pozice a rotace, k tomu používá parametr změny času *dt* (změna času je uběhlá doba od začátku poslední krokové funkce). Přesnost aproximace závisí na změně času. Problém je, že čím menší je změna času, tím máme méně času na výpočty v krokové funkci. Nejjednodušší numerická metoda pro tento účel se nazývá Eulerova metoda, ale existují i přesnější metody jako je Verletova metoda nebo jiné Runge-Kuttovy metody. [3]

2.2.1 Eulerova metoda

Eulerova metoda je nejjednodušší numerickou metodou pro řešení obyčejných diferenciálních rovnic. Umožňuje aproximovat rychlost a polohu tělesa, pokud známe jeho zrychlení. Metoda přičte k aktuálnímu stavu derivaci (pro rychlost zrychlení, pro polohu rychlost) vynásobenou o konečnou diferenci, v našem případě o změnu času *dt*. V matematické formě metodu můžeme zapsat způsobem (pokud známe zrychlení jako funkci času):

$$\vec{v}(t + dt) \approx \vec{v}(t) + \vec{a}(t + dt) \cdot dt, \quad (1)$$

$$x(t + dt) \approx x(t) + v \cdot (t + dt) \cdot dt, \quad (2)$$

$$y(t + dt) \approx y(t) + v \cdot (t + dt) \cdot dt, \quad (3)$$

kde a je zrychlení, v je rychlost, x a y je pozice, t je čas posledního kroku a dt je změna času. Stejné rovnice můžeme aplikovat i na fyzikální veličiny pro popis rotace tělesa. Do rovnic stačí za zrychlení dosadit úhlové zrychlení a za rychlost dosadit úhlovou rychlost, konečná veličina bude úhel natočení tělesa.

Kromě nepřesnosti má Eulerova metoda, také nevýhodu v tom, že nezachovává energii v simulaci, jedná se o tzv. symplektickou vlastnost (sympletic). Symplektické metody zaručují, že numerické řešení přibližně zachovává určité konstanty systému, jako je energie. Eulerova metoda tedy není symplektická, absence této vlastnosti vede k systematickému nárůstu nebo poklesu energie v průběhu simulace, což může vést k nerealistickým

výsledkům a ztrátě fyzikální věrohodnosti. Z tohoto důvodu se pro dlouhodobé simulace dynamických systémů často upřednostňují symplektické metody, jako je například Verletova metoda nebo speciální Eulerova metoda, která kombinuje Verletovu a Eulerovu metodu.

2.2.2 Verletova metoda

Verletova metoda nebo také nazývaná Verletova integrace je numerická metoda pro řešení pohybových rovnic. Vyvinul ji francouzský fyzik Loup Verlet pro simulování molekulární dynamiky. Její hlavní odlišení od ostatních numerických metod je, že k výpočtu si u každého tělesa musíme uložit jeho předchozí pozici. Výhodou je, že díky tomu nemusíme mít explicitně definovanou rychlost tělesa. K výpočtu nové pozice (x_{n+1}), použijeme rovnici:

$$x_{n+1} = 2x_n - x_{n-1} + a(t) \cdot dt^2, \tag{4}$$

kde x_n je aktuální pozice, x_{n-1} je pozice v předchozím kroku, a je zrychlení a dt je změna času. Metoda je přesnější, pokud je délka kroku (změna času) konstantní. Verletova metoda je díky své symplektické povaze dlouhodobě stabilní. Další vlastnost, která z této povahy plyne, je časová vratnost, to znamená, že pokud simulaci spustíme dopředu a poté ji pustíme pozpátky se stejným časovým krokem, vrátíme se do téměř původního stavu (odchylka je způsobena numerickými chybami) [6].

2.2.3 Runge-Kuttova metoda

Runge-Kuttovu metodu vytvořili kolem roku 1900 němečtí matematici Carl Runge a Wilhelm Kutta. Je důležité zdůraznit, že Runge-Kuttovy metody nepředstavují jednu konkrétní metodu, ale spíše rodinu numerických metod (patří do ní i Eulerova metoda) pro přibližné řešení obyčejných diferenciálních rovnic.

Nejpoužívanější Runge-Kuttova metoda se označuje jako klasická nebo také zkratkou RK4 (je to metoda čtvrtého řádu, Eulerova metoda je prvního řádu). RK4 metoda počítá stav následujícího kroku pomocí váženého průměru čtyř směrnic (derivací) vypočítaných v různých bodech v rámci kroku dt. Každá z těchto směrnic (k_1, k_2, k_3, k_4) představuje odhad změny stavu za čas dt. Tyto směrnice se vypočítají následovně:

$$k_1 = f(t_n, y_n), \tag{5}$$

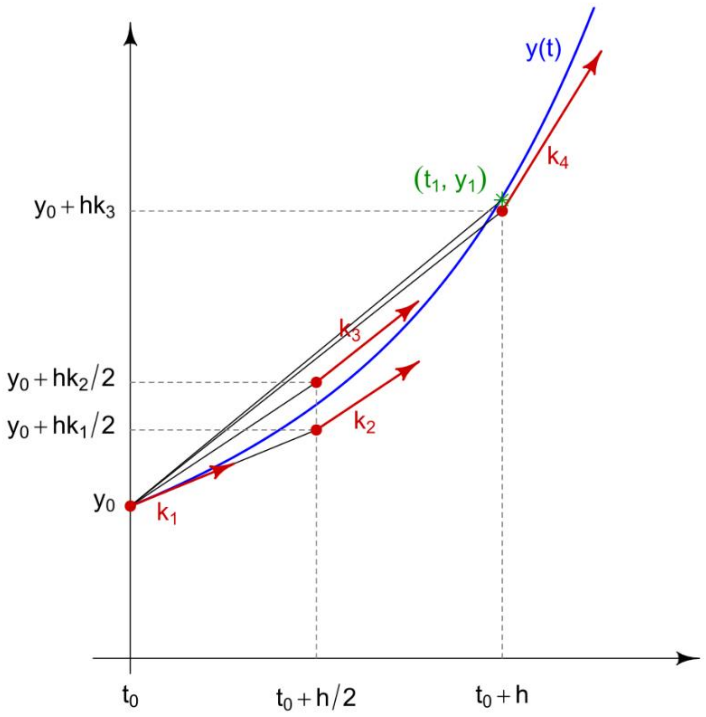
$$k_2 = f\left(t_n + \frac{dt}{2}, y_n + dt \cdot \frac{k_1}{2}\right), \tag{6}$$

$$k_3 = f\left(t_n + \frac{dt}{2}, y_n + dt \cdot \frac{k_2}{2}\right), \tag{7}$$

$$k_4 = f(t_n + dt, y_n + dt \cdot k_3), \tag{8}$$

kde $f(t, y)$ je funkce, která definuje, jak se stav tělesa mění v čase (funkce f, nám říká derivaci vektoru stavu y v daném čase t), y_n je vektor stavu v čase t_n (pozice, rychlost, zrychlení).

První směrnice počítá, jaká je derivace v aktuálním bodě, následující směrnice jsou vždy počítány z bodu předchozí směrnice (viz. Obrázek 2). Druhou a třetí směrnici, počítáme pouze s poloviční změnou času a polovičním počátečním bodem pro lepší vyváženost odhadu.



Obrázek 2: graf zobrazující směrnice pro metodu RK4 [2]

Konečná rovnice pro výpočet nového stavu y_{n+1} v čase $t_n + dt$:

$$y_{n+1} = y_n + \frac{dt}{6} \cdot (k_1 + 2k_2 + 2k_3 + k_4), \tag{9}$$

v rovnici počítáme průměr směrnic a výsledek násobíme změnou času dt pro získání změny vektoru stavu. Druhá a třetí směrnice mají dvojnásobnou váhu, a proto musíme pro výpočet průměru dělit šesti místo čtyřmi.

RK4 nabízí výrazně vyšší přesnost než Eulerova metoda, a to za cenu větší výpočetní náročnosti. Metoda není symplektická, ale její chybovost se projevuje až po dlouhé době, protože její chyba na krok je úměrná h^5, kde h je velikost kroku (v našem případě změna času dt). Např. pokud simulace běží 600krát (60 snímku za sekundu, 10 kroků za snímek) za sekundu, její krok dosahuje velikosti tisícin sekundy, v takovém případě je chyba pro real-time simulace téměř zanedbatelná. Další výhodou RK4 je, že může být použita na širší škálu problémů oproti Verletově metodě, která je určená pouze pro pohybové rovnice.

2.3 Detekce kolizí

Jakmile physics engine simuluje pohyb, můžeme pozorovat, že tělesa mezi sebou neinteragují a pouze se míjejí. Pro implementaci interakcí je nejprve nezbytné detekovat kolize, což je problém, jímž se zabývají algoritmy pro detekci kolizí. Detekce kolizí je obecně složitý problém, a to jak z matematického, tak z výpočetního hlediska. Cílem není pouze detekovat kolizi, ale také vypočítat hloubku a body kolize pro její následné řešení. Je rovněž důležité zdůraznit, že detekce kolizí se zabývá i tím, jakým způsobem se dva objekty mají pohnout, aby už nedocházelo k překrývání (kolizi). Až poté přichází na řadu řešení kolize, které se snaží simulovat reakci objektů na kolizi tak, aby byla co nejvíce uvěřitelná.

Nejjednodušší implementací detekce kolizí je algoritmus AABB (Axis-Aligned Bounding Box). Jeho omezením je, že pokud strany objektů nejsou zarovnány na stejné ose, detekce kolizí je nepřesná. Z tohoto důvodu se AABB obvykle využívá v simulacích, kde se pracuje pouze s nerotujícími obdélníky (kvádry v 3D prostoru). Mezi pokročilejší algoritmy patří SAT (Separating Axis Theorem), GJK (Gilbert-Johnson-Keerthi) a EPA (Expanding Polytope Algorithm), které poskytují přesnější detekci kolizí pro složitější geometrické tvary.

2.3.1 AABB

Implementace AABB je poměrně jednoduchá, jak už z názvu vychází pro každý objekt si vytvoříme ohraničující box, následně pro detekci budeme kontrolovat, jestli se hranice dvou boxů nepřekrývají. Pro kolizi musí boxy splňovat následující podmínky:

$$xmax_1 \geq xmin_2, \tag{10}$$

$$xmin_2 \leq xmax_2, \tag{11}$$

$$ymax_1 \geq ymin_{2,} \tag{12}$$

$$ymin_1 \leq ymax_2, \tag{13}$$

kde $xmax_i$ je maximální hodnota x i-tého boxu v kolizi, $xmin_i$ je minimální hodnota x i-tého boxu v kolizi, $ymax_i$ a $ymin_i$ jsou také maximum a minimum i-tého boxu, ale na ose y. Pokud jsou všechny podmínky splněny, objekty jsou v kolizi.

Omezení AABB byly už řečeny, ale existuje alternativa, která je dokáže obejít a detekovat kolize s přijatelnou přesností. Nazývá se AABBTree, jde o stejný princip jako AABB, akorát místo toho, abychom pro každý objekt vytvořili pouze jeden box, tvar objektu rozkouskujeme na několik menších boxů, díky tomu můžeme použít AABB i u složitějších tvarů, které nejsou pouze obdélníky nebo i u rotovaných tvarů.

2.3.2 SAT

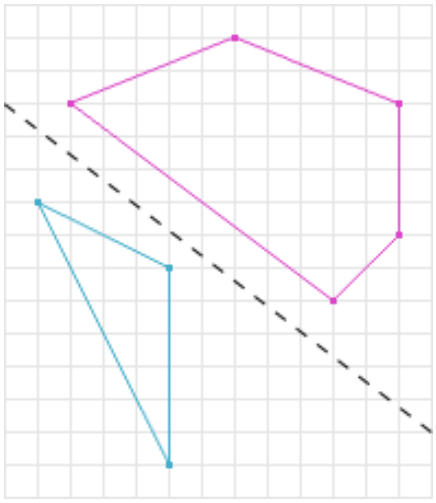
Separating Axis Theorem (SAT) je populární algoritmus pro detekci kolizí, jeho výhodou je, že není závislý na jednom konkrétním tvaru, jako je tomu například u AABB. SAT dokáže detekovat kolizi mezi jakýmikoli dvěma konvexními tvary. Pokud potřebujeme detekovat kolize i u konkávních tvarů, musíme je rozdělit na několik konvexních částí a pracovat s nimi jako s jedním celkem.

Pro lepší pochopení principu SAT si můžeme představit situaci ve volném vesmíru, kde máme před sebou dva kvádry a naším úkolem je zjistit, zda jsou v kolizi, tedy zda se navzájem dotýkají. Nejprve se podíváme z perspektivy, která je souběžná s jednou stranou jakéhokoli z tvarů. Pokud se tvary nedotýkají, můžeme si být jisti, že nejsou v kolizi, a máme odpověď. Pokud se však dotýkají, musíme pokračovat a zkoumat další perspektivy, které jsou souběžné s ostatními stranami tvarů. Pokud se podíváme ze všech těchto perspektiv a zjistíme, že se tvary dotýkají ve všech případech, můžeme s jistotou říct, že jsou v kolizi.

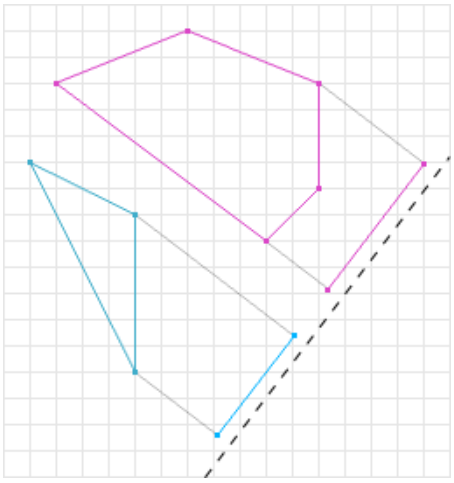
Je důležité, aby tvary byly konvexní, protože u konkávních tvarů bychom si nemohli být jisti, zda se skutečně dotýkají, nebo zda to tak vypadá pouze z určitého úhlu pohledu. Například si představte misku, ve které levituje lžíce. Z vnější perspektivy nevidíme mezeru

mezi nimi, a mohou se nám zdát jako jeden souvislý objekt, i když ve skutečnosti tomu tak není.

Formální definice SAT zní: „Pokud se dva konvexní tvary nepenetrují, existuje oddělující osa, na které se projekce tvarů nebude překrývat [4].“ Osa na, které se projekce tvarů nepřekrývají se říká oddělující osa (separating axis). Abychom pochopili tuhle definici nejdříve si musíme vysvětli, co je to projekce. Projekce konvexního tvaru na osu je proces, při kterém se určuje, jaký úsek na této ose odpovídá tvaru. Tento úsek je určen minimální a maximální hodnotou projekce všech vrcholů tvaru na danou osu. Můžeme si ji představit jako stín tvaru, který dopadá na osu. Je důležité podotknout, že projekci neděláme na oddělující ose (která je souběžná se stranou tvaru), ale na ose, která je kolmá na oddělující osu. V našem přirovnání na začátku kapitoly to však není zásadní, protože strany kvádru jsou na sebe kolmé, a tudíž se projekce může provádět i na oddělující osu. Matematicky se projekce vypočítává pomocí skalárního součinu. Pro každý vrchol tvaru uděláme skalární součin s osou, z těchto hodnot jsou pro nás důležité pouze hodnota maximální a minimální, z těch se skládá výsledek projekce.

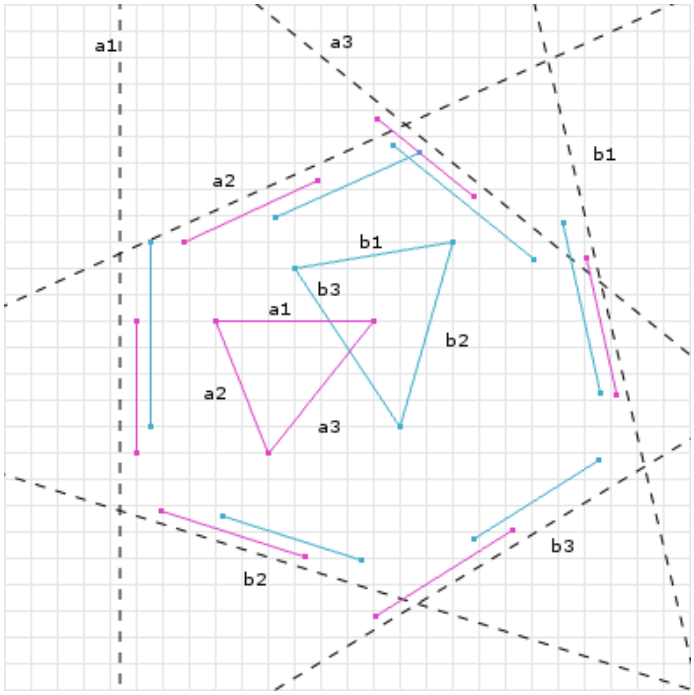


Obrázek 3: zobrazení oddělující osy [4]



Obrázek 4: projekce tvarů na osu [4]

Jakmile získáme projekci obou tvarů, zjistíme, jestli se intervaly projekce prolínají. Pokud se neprolínají, máme ověřeno, že tvary nejsou v kolizi. V opačném případě se potvrdí, že se na této ose objekty překrývají. Pro zjištění, jestli jsou tvary v kolizi musíme porovnat projekce tvarů u všech ostatních os (které jsou kolmě na strany tvarů), jakmile se projekce tvarů prolínají na všech os, tak víme, že se tvary překrývají tzv. jsou v kolizi (viz. Obrázek 4).



Obrázek 5: projekce tvarů na všech osách [4]

Algoritmus nejen detekuje kolize, ale také poskytuje Minimum Translation Vector (MTV) pro řešení kolizí. MTV je vektor, který znázorňuje nejmenší translační pohyb jednoho tvaru, potřebný k tomu, aby tvary nebyli v kolizi a pouze se dotýkali. Další důležitou vlastností MTV je, že jeho velikost odpovídá hloubce kolize. Pro jeho získání stačí během algoritmu uchovávat minimální prolnutí mezi projekcemi objektů a osou, na kterou byla projekce provedena (na obrázku 4 se jedná o osu b3).

Hlavní účel MTV je pro vyřešení překrytí objektů. K tomu nám stačí MTV rozdělit na dva vektory opačného směru a následně posunout objekty o vektory, tak se nám překrytí vyřeší. V implementaci také pro realističtější výsledky měníme velikost posuvného vektoru podle hmotnosti tělesa. Například při kolizi lehkého tělesa s mnohonásobně těžším, většinu pohybu vykoná lehčí těleso.

2.4 Řešení kolizí

Řešení kolizí (anglicky Collision Resolution) se zabývá algoritmy a modely, které simulují změnu pohybu dvou těles po kolizi. Většina physics enginů používá impulse-based model, který je založen na počítání impulsů v bodech kolize a jejich následné aplikování na tělesa. Physics enginy s Verletovým integrátorem mají zde další výhodu, neboť nemusí řešení kolizí implementovat. Důvodem je, že Verletův integrátor nevyžaduje explicitně

definovanou rychlost, pokud vyřešíme překrytí těles, tak se implicitní rychlost sama upraví pomocí minulé pozice.

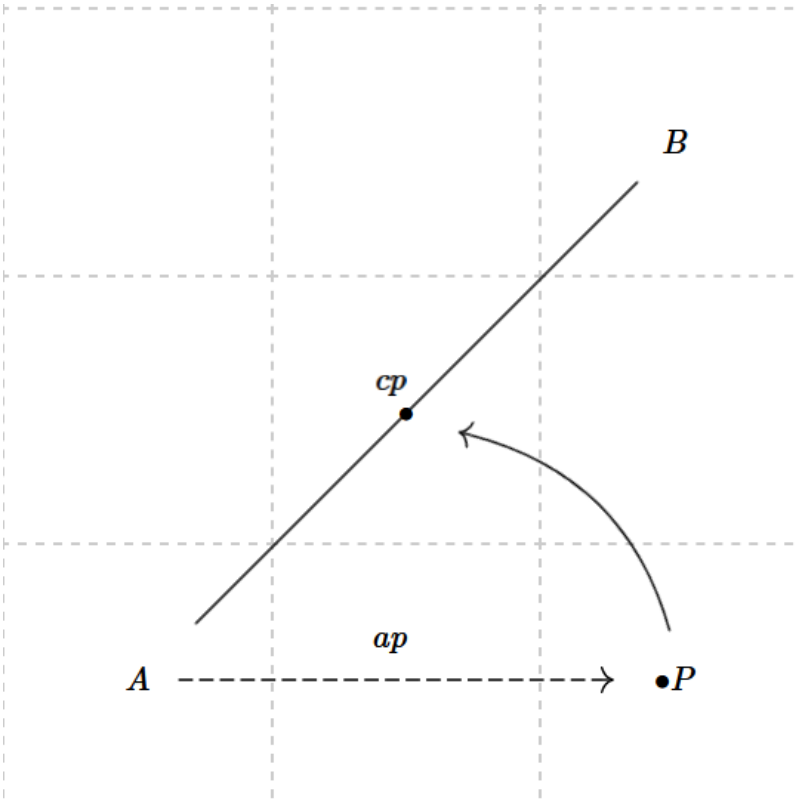
Naopak physics enginy, které používají metody s explicitně definovanou rychlostí, musí změnu rychlosti po kolizi naimplementovat např. pomocí impulse-based modelu. Impuls ve fyzice vyjadřuje časový účinek působení síly, vyvolává změnu hybnosti tělesa. Vzhledem tomu, že většina simulací pracuje s konstantními hmotnostmi, je nutné změnit rychlost tělesa. Pro výpočet impulsu musíme nejdříve zjistit z jakého bodu na tělese impuls působí, proto musíme zjistit body kde kolize nastala.

2.4.1 Body kontaktu

Body kolize, známé také jako body kontaktu, představují místa, kde se tělesa dotýkají po vyřešení jejich překrývání. Pro jejich určení je nezbytné nejprve pochopit, jak vypočítat vzdálenost bodu od úsečky. Pokud chceme zjistit vzdálenost bodu *P* od úsečky definované body *A* a *B*, začneme tím, že vytvoříme vektor *AP*, který spojuje bod *A* s bodem *P*. Následně provedeme projekci vektoru *AP* na vektor úsečky *AB* pomocí skalárního součinu. Tímto způsobem získáme hodnotu, která nám umožní určit, jak daleko je bod *P* od úsečky. Pro určení bodu na úsečce, který je neblíže k bodu *P* použijeme rovnici:

$$cp = A + AB \cdot \left(\frac{proj}{||AB||}\right), \tag{14}$$

kde cp je výsledný bod, $proj$ je skalární součin *AP* s *AB*. Projekci dělíme velikostí vektoru *AB*, abychom dostali normalizovanou velikost projekce. Výslednou vzdálenost vypočítáme pomocí definice vektoru mezi body *P* a *cp*, velikost vektoru se rovná vzdálenosti bodu *P* od přímky *AB*.

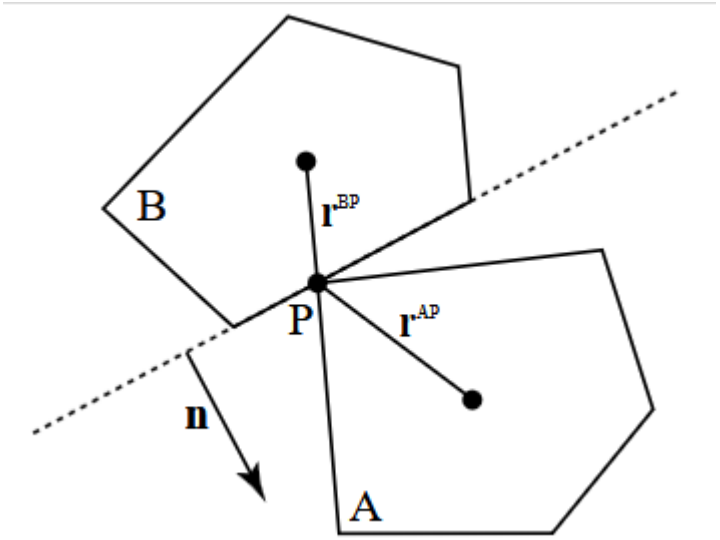


Obrázek 6: výpočet vzdálenosti bodu od přímky

Algoritmus pro výpočet bodů kolize, iteruje přes všechny strany jednoho tělesa a pro každou stranu počítá vzdálenost mezi všechny vrcholy druhého tělesa, vždy ukládá pouze nejmenší vzdálenost a bod *cp* pro danou vzdálenost (cp je zkratka pro contact point). To samé uděláme znovu, ale prohodíme role těles. Také je důležité zmínit že kolize může mít dva kontaktní body, pokud je rozdíl vzdálenosti bodu a aktuální minimální vzdálenosti dostatečně malý, považujeme oba body jako kontaktní. Tento přístup zohledňuje preciznost a numerické chyby, které mohou nastat při práci s desetinnými čísly v počítačových výpočtech.

2.4.2 Impuls kolize

Jak už víme, impuls je změna hybnosti tělesa po kolizi. Pro každý kontaktní bod v kolizi vypočítáme impuls. Podle Newtonova třetího zákona platí, že síly působící mezi dvěma tělesy jsou stejně velké a opačného směru. To znamená, že působí-li na těleso *A* impuls, tak na těleso *B*, se kterým je těleso *A* v kolizi, bude působit stejně velký impuls opačného směru. Samotná impulsová metoda vychází ze zákona zachování hybnosti a využívá součinitele restituce k modelování pružnosti srážky.



Obrázek 7: Řešení kolize těles A a B [5]

Předpokládejme dvě tělesa *A* a *B*, která se srazí v bodě *P*. Nejprve musíme definovat relativní rychlost v bodě kontaktu:

$$v_{AB} = (v_B + \omega_B) - (v_A + \omega_A) \tag{15}$$

Pokračujeme určením normálu n v bodě kontaktu (ten se počítá normalizací MTV), která směruje od tělesa *B* k tělesu *A*. Relativní rychlost ve směru normálu je dána skalárním součinem:

$$v_n = v_{AB} \cdot n \tag{16}$$

Impuls j se vypočítá pomocí rovnice:

$$j = \frac{-(1+e) \cdot v_n}{n \cdot \left(\frac{1}{m_A} + \frac{1}{m_B}\right) + \frac{(r_{AP} \cdot n)^2}{I_A} + \frac{(r_{BP} \cdot n)^2}{I_B}} \tag{17}$$

kde e je součinitel restituce určující míru pružnosti srážky, nabývá hodnoty od 0 do 1 (0 pro dokonale nepružnou srážku a 1 pro dokonale pružnou srážku). m_A a m_b jsou hmotnosti těles, I_A a I_b jsou momenty setrvačnosti těles, r_{AP} a r_{BP} jsou vektory z bodu středu tělesa k bodu kolize *P* (viz. Obrázek 7) [5].

Aplikace impulsu na rychlosti těles je vyjádřeno následujícími vztahy:

$$v_A = v_A + \frac{j}{m_A} \cdot n, \tag{18}$$

$$v_B = v_B - \frac{j}{m_B} \cdot n, \tag{19}$$

a pro úhlovou rychlost:

$$\omega_A = \omega_A + \frac{r_{AP} \times (j \cdot n)}{I_A}, \tag{20}$$

$$\omega_B = \omega_B - \frac{r_{BP} \times (j \cdot n)}{I_B} \tag{21}$$

Při implementaci impulse-based modelu je důležité prvně vypočítat impulsy u všech kontaktních bodů, a až poté aplikovat impulsy na tělese, pokud bychom aplikovali impuls hned po jeho výpočtu, ovlivnili bychom vstupní hodnoty pro další kontaktní bod.

Způsob řešení kolizí pomocí impulsů není ideální a má hodně nedostatků. Hlavním problémem je numerická stabilita. Akumulace numerické chyby může vést k nežádoucímu chování simulace a narušení stability systému. Další nevýhodou je, že malé nepřesnosti ostatních systémů (numerického integrátoru, algoritmu pro výpočet kontaktních bodů) ovlivňují impuls-based model znatelným způsobem, může se tak stát, že systém vyřeší kolizi nevěrohodným způsobem, jelikož impuls bude dosahovat příliš velkých hodnot. Tyto faktory činí impulsový model vhodným zejména pro rychlé a přibližné simulace, avšak méně vhodným pro přesné fyzikální simulace s vysokou věrností interakcí.

3. Praktická část

Praktická část práce se zabývá implementací projektu Fyzix, od popisu jeho struktury až po samotnou implementaci. Na rozdíl od teoretické části, která se zaměřuje na matematický a fyzikální popis metod pro tvorbu physics enginu, je praktická část věnována hlubšímu zkoumání informatických aspektů jeho implementace.

3.1 Architektura a struktura projektu

Souborová struktura projektu je rozdělena do tří hlavních složek pro kód a dvou sekundárních složek. První složka se nazývá „include“, nachází se v ní všechny hlavičkové soubory (header files) mého kódu. Hlavičkové soubory slouží v jazyku C a C++ pro dopřednou deklaraci tříd, funkcí, proměnných a jiných identifikátorů. Pokud používáme identifikátory, které se nenachází ve stejných kompilačních jednotkách, můžeme je dopředně deklarovat v hlavičkovém souboru, kde uvedeme základní informace o nich např. jako typ, argumenty nebo metody. Složka pro zdrojový kód se jmenuje „source“, definuje a implementuje soubory, které byly deklarovány v hlavičkových souborech. Celkově je v obou složkách přibližně 40 souborů (20 hlavičkových a 20 zdrojových), ale jejich podrobnosti si objasníme později. Třetí složka s kódem slouží pro jakýkoliv kód, který je od třetí strany, proto se jmenuje „thirdparty“. Nachází se v něm dvě statické knihovny projektu, „Dear ImGui“ (UI knihovna) a „nloohman JSON“ (knihovna pro pracovaní s JSON soubory).

Ostatní složky v projektu slouží pro ikony uživatelského rozhraní (složka „assets“) a pro JSON soubory, které slouží pro ukládání scén z aplikace (složka „scenes“). Ve souborové složce projektu se také nachází dva soubory, první je konfigurační soubor „CMakeLists.txt“ pro sestavení projektu (více o Cmake v 3.1.4), druhý soubor slouží pro licenci a nazývá se „LICENCE“. Licence projektu je svobodná licence MIT, povoluje jakékoliv použití i v proprietárním softwaru za podmínkou, že software bude dodáván s textem MIT licence.

Pokud chceme projekt sestavit ze zdrojového kódu, standardní postup je vytvoření si složky pro sestavení (např. s názvem „build“), poté stačí v této složce zadat dva příkazy z terminálu, první slouží k přípravě souborů na sestavení, jedná se o příkaz „cmake ..“, musíme použít argument dvou teček, protože se konfigurační soubor nachází nad složkou pro sestavení. Druhý příkaz „make“ je už pro samotnou kompilaci kódu.

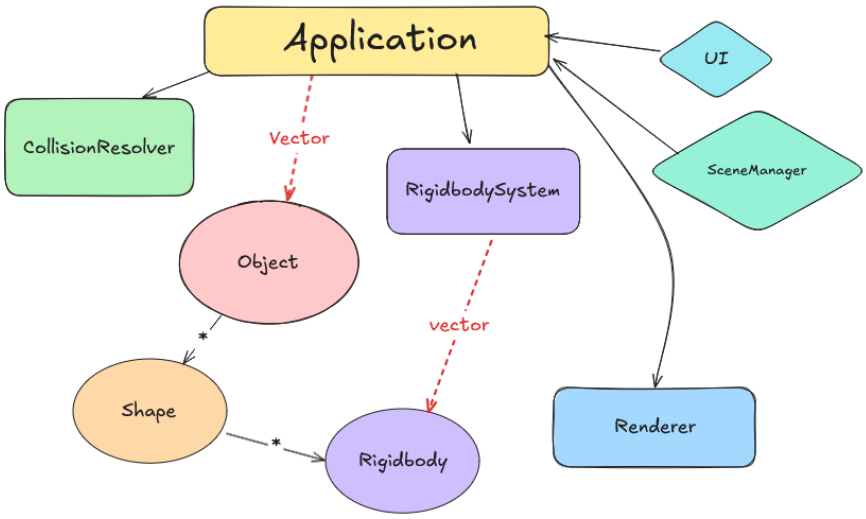
3.1.1 Programovací paradigma

Programovací paradigma je styl, který určuje, jakým způsobem se programy navrhují a implementují. Paradigma ovlivňuje strukturu a organizaci kódu, stejně jako způsob, jakým se řeší problémy. Jelikož je C++ rozsáhlý jazyk, podporuje více paradigmat, např. programy mohou být čistě procedurální, objektově orientované, funkcionální nebo kombinací těchto paradigmat.

Projekt Fyzix používá hlavně paradigma objektové, které se také nazývá pojmem „objektově orientované programování“. Jednou z hlavních výhod objektového programování je jeho schopnost modelovat reálný svět prostřednictvím objektů, které reprezentují fyzikální entity, jako jsou tělesa, tvary a interakce. Tento přístup usnadňuje pochopení a údržbu kódu, protože programátor vidí podobnost mezi částí kódu a reálným světem. Polymorfismus je další klíčovou výhodou objektového programování, umožňuje vytvářet generické a škálovatelné systémy, kde různé objekty reagují jinak na stejné metody, to je užitečné u fyzikálních objektů, které sdílí stejné rozhraní, ale mohou mít odlišné chování. Na druhou stranu objektové programovaní může vést k přílišné abstrakci kódu, která není potřeba. Zamezit tomu můžeme např. tak, že nejdříve kód zprovozníme procedurálně a až poté ho předěláme do objektové formy. Přílišné abstrakce mohou zpomalit celkový výkon, který je u physics enginu kritický.

3.1.2 Rozhraní a třídy

Fyzix je programován hlavně pomocí tříd. Hlavní třída se jmenuje Application, spravuje všechny ostatní systémy enginu a obsahuje hlavní smyčku programu. Dále zde máme tři systémy. Systémem je myšlena třída, která bude mít po celou dobu pouze jednu instanci. První systém je třída RigidbodySystem, která spravuje dynamiku těles. Dalším systém se nazývá CollisionResolver, jeho účelem je detekce a řešení kolizí. Poslední systém Renderer je k vykreslovaní objektů na obrazovku. Třída Object slouží pro objekty v simulaci. Tvary implementujeme pomocí rozhraní Shape. Objekty fyzikálně popisuje třída Rigidbody.



Obrázek 8: Diagram aplikace

3.1.3 C++

Klíčový důvod, proč je Fyzix implementován v programovacím jazyce C++ je jeho efektivita a rychlost, ale ne jenom to, také nám umožňuje užitečné funkce jako přetěžování a polymorfismus. Dalším důvodem je, že jsem se chtěl s jazykem více seznámit. Jelikož mé znalosti o C++ byly před začátkem projektu pouze na začátečnické úrovni, snažil jsem se během implementace experimentovat a vzdělávat se v programovacích metodách, které s C++ souvisí. Například jsem se snažil spravovat alokace a dealokace paměti pomoci metody RAII, na rozdíl od přístupu moderního C++ s chytrými ukazateli (smart pointers). Také jde vidět mé zlepšení v jazyce na nekonzistentnosti kódu, ať už jde o nazývaní proměnných nebo o samotné programování.

Jazyk C++ má však i nevýhody, zejména jeho složitost a neoptimální implementaci některých funkcí. Také sestavovaní, spravovaní knihoven a závislostí je zbytečně komplikované. K tomuto účelu se převážně používá nástroj Cmake.

3.1.4 Cmake a Git

Pro správu verzí byl pro vývoj Fyzix používán všeobecně známý systém Git. Díky tomu je Zdrojový kód aplikace uložen na platformě GitHub i se všemi změnami kódu [9].

Jak už bylo zmíněno pro sestavení a správu knihoven byl použit nástroj Cmake. Cmake ke konfiguraci používá soubor „CMakeLists.txt“, ve kterém specifikujeme, jaké soubory chceme nalinkovat během kompilace. Například pokud se jedná o dynamickou knihovnu Cmake se ji snaží najít v počítači, na kterém chceme projekt zkompilovat. U statických knihoven stačí cesta, kde jsme knihovnu v projektu uložili.

3.2 Implementace physics enginu

3.2.1 Třída Vec2

Třída Vec2 slouží pro vektorové fyzikální veličiny, jelikož je Fyzix implementován ve dvou dimenzích, stačí ve vektoru uchovávat pouze dvě skalární veličiny – jednu pro každou osu. Mezi vektorové veličiny v enginu patří např. pozice, rychlost, zrychlení a síla.

Třída využívá tzv. přetížení operátoru (operator overloading). Přetížení umožnuje pro objekty třídy používat klasické operátory, které jsou ve většině programovacích jazycích implementovány pouze pro datové typy (např. int, double). Pro porovnaní v jazycích bez přetěžování bychom museli implementovat například součet vektorů jako běžnou metodu

např. *„Vec2 add(Vec2 v)“*, díky přetěžovaní můžeme stejný kód, umístit do přetížení operátoru +, což v C++ vypadá následovně: „Vec2 operator+(Vec2 v)“. Přetížení umožňuje intuitivnější zápis operací, takže můžeme mezi dvěma vektory použít operátor +, přičemž jeho chování bude odpovídat kódu definovanému v přetížení. Tato abstrakce výrazně zlepšuje čitelnost kódu a usnadňuje práci s třídou. Při použití přetěžování je důležité dodržovat jistá pravidla, například by nemělo přetížení alokovat paměť na haldě. Třída Vec2 využívá přetížení převážně u matematických operátorů, s výjimkou u operátoru <<, který se hlavně používá pro výstup konzole. Díky přetížení lze instanci třídy snadno vypsat podle definovaného formátu.

Kromě přetížených operátorů obsahuje Vec2 také klasické metody. Statická metoda zero() vrací nulový vektor. Metoda normal() vrací normálový vektor ke vektoru, ze kterého byla volána. Metoda normalize() vrací vektor s normalizovanými hodnotami (v rozsahu 0 až 1). Metoda magnitude() vrací velikost (normu) vektoru. Metoda dot(Vec2 v) vypočítá skalární součin dvou vektorů a metoda cross(Vec2 v) slouží k výpočtu vektorového součinu dvou vektorů.

```
1. #include <ostream>
2. class Vec2 {
3. public:
4.     double x;
5.     double y;
6.     Vec2(double _x = 0, double _y = 0):x(_x), y(_y) {};
7.     Vec2(const Vec2& v): x(v.x), y(v.y) {};
8.     Vec2 operator+(const Vec2& v) const;
9.     Vec2& operator+=(const Vec2& v);
10.      Vec2& operator-=(const Vec2& v);
11.      Vec2 operator-(const Vec2& v) const;
12.      Vec2 operator-() const;
13.      Vec2 operator*(const Vec2& v) const;
14.      Vec2 operator*(const double& a) const;
15.      Vec2 operator/(const double& a) const;
16.      bool operator!=(const Vec2& v) const;
17.      static Vec2 zero();
18.      Vec2 normal() const;
19.      Vec2 normalize() const;
20.      double magnitude() const;
21.      double dot(const Vec2& v) const;
22.      double cross(const Vec2& v) const;
23.
24.      friend std::ostream& operator<<(std::ostream& os, const Vec2& vec);
25. };
26.
27. inline std::ostream& operator<<(std::ostream& os, const Vec2& vec) {
28.     os << "(" << vec.x << ", " << vec.y << ")";
29.     return os;
30. }
```

Kód 1: hlavičkový soubor třídy Vec2

3.2.2 Třída Rigidbody

Třída Rigidbody představuje abstrakci fyzikálního tělesa. Obsahuje datové atributy reprezentující fyzikální veličiny potřebné pro dynamiku těles. Většina těchto veličin využívá třídu Vec2, s výjimkou hmotnosti a veličin popisujících rotaci.

Veškeré fyzikální veličiny jsou vyjádřeny v hlavních jednotkách soustavy SI. Proto je důležité si uvědomit, např. že veličiny popisující rotaci nejsou udávány ve stupních, ale v radiánech. Rigidbody si můžeme představit jako hmotný bod, který určuje chování objektu v simulaci.

Třída obsahuje dvě *getter* metody pro získání souřadnic pozice a metodu renderVelocityVector(), která slouží k vizualizaci rychlosti tělesa – vykreslí vektor rychlosti jako šipku.

```
1. #include "Vec2.hpp"
2. #include "Renderer.hpp"
3. class Rigidbody {
4.    public:
5.     Vec2 pos;  // position; pozice
6.     Vec2 v;    // velocity; rychlost
7.     Vec2 a;    // acceleration; zrychleni
8.     Vec2 f;    // force; sila
9.     double m;  // mass; hmotnost
10.
11.      double theta;    // rotational angle
12.      double omega;    // angular velocity; uhlova rychlost
13.      double epsilon;  // angular acceleration; uhlove zrychleni
14.
15.      int idx;
16.
17.      Rigidbody(double p_x = 0, double p_y = 0, double v_x = 0,
18.                double v_y = 0, double a_x = 0, double a_y = 0,
19.                double f_x = 0, double f_y = 0, double _m = 1,
20.                double _theta = 0, double _omega = 0,
21.                double _epsilon = 0);
22.      Rigidbody(const Rigidbody& rb);
23.      ~Rigidbody();
24.
25.      double getX() const;
26.      double getY() const;
27.      void checkRestingPosition();
28.      void renderVelocityVector(Renderer* renderer);
29.  };
```

Kód 2: hlavičkový soubor Rigidbody

3.2.3 Třída Shape

Třída Shape slouží jako rozhraní (čistě virtuální třída v C++) pro třídy, které implementují různé tvary objektů do simulace. Mezi datové atributy patří ukazatel na instanci Rigidbody

daného objektu, jednotlivé rozměry tvaru jsou definovány pro každou implementaci jinak proto nejsou v rozhraní. Ve Fyzix rozhraní Shape implementují třida rectangleShape pro obdélník a třída circleShape pro kruh.

Rozhraní obsahuje dvě metody pro vykreslovaní render() a renderOutline(), první vykreslí celý tvar a druhá vykreslí pouze okraj. Další metoda project() slouží pro detekci kolizí pomocí metody SAT. Metoda momentOfInertia() vrací moment setrvačnosti podle tvaru a fyzikálních parametrů tělesa. Poslední dvě metody slouží k broad-phase detekci kolizí.

```
1. #include "Rigidbody.hpp"
2. #include "Axis.hpp"
3. #include "Renderer.hpp"
4. class Shape{
5. public:
6.     Rigidbody* rigidbody;
7.     char type; // r = rectangle, c = circle
8.
9.     Shape(Rigidbody* rb, char _type);
10.      Shape(const Shape& sh);
11.      virtual ~Shape();
12.      virtual void render(Renderer* renderer, int color) = 0;
13.      virtual void renderOutline(Renderer* renderer, int color) =
    0;
14.      virtual Vec2 project(const Axis& axis) const = 0;
15.      virtual double momentOfInertia() = 0;
16.      virtual bool containsPoint(const Vec2& point) = 0;
17.      virtual double getRadius() = 0;
18.  };
```

Kód 3: Hlavičkový soubor třídy Shape

3.2.4 Třída Object

Třída Object slouží jako abstrakce celého objektu v simulaci. Mezi její datové atributy patří ukazatel na instanci třídy, která implementuje Shape odpovídající tvaru daného objektu. Dalším důležitým atributem je type, který určuje chování objektu v simulaci.

Ve Fyzix existují dva typy objektů: fixed a dynamic. Fixed objekty jsou statické – nastaví se na začátku simulace a během jejího průběhu zůstávají nehybné. Typickým příkladem je zem. Dynamic objekty naopak podléhají fyzikálním zákonům a pohybují se podle simulované dynamiky. Atribut color přiřazuje objektu barvu. Podporované barvy jsou definovány v souboru Renderer.hpp.

Mezi metody třídy Object patří hlavní metoda init(), která objekt přidá do simulace. Metoda render() volá stejnojmennou metodu v instanci Shape a předává jí barvu objektu. Další důležitou metodou je metoda translate(), která slouží k translaci objektu o MTV vektor.

```
1. #include "Shape.hpp"
2. class Object {
3. public:
4.     enum Type {
5.         FIXED,
6.         DYNAMIC,
7.         INVISIBLE
8.     };
9.     Shape* shape;
10.      Type type;
11.      Color color;
12.      int idx;
13.
14.      Object();
15.      Object(const Object& ob);
16.      ~Object();
17.      void init(Application* app, Shape* _shape,
18.                Type t = DYNAMIC, Color c = BLUE);
19.      void render(Renderer* renderer);
20.      bool isFixed();
21.      void translate(const Vec2& mtv);
22.      double inverseInertia();
23.  };
```

Kód 4: Hlavičkový soubor třídy Object

3.2.5 Třída Application

Třída Application slouží jako třída, která jak už název napovídá, zastřešuje celou aplikaci. Spravuje všechny systémy physics enginu a obsahuje hlavní smyčku aplikace. Mezi datové atributy patří převážně ukazatele na pod systémy physics enginu. Další důležitý atribut je ukazatel m_scene, který odkazuje na aktuálně zvolenou scénu. Všechny objekty v simulaci jsou uloženy na haldě a jejich ukazatele ukládáme do dynamického pole jménem m_objects, které je dalším datovým atributem třídy Application.

Metody addObject() a removeObject() slouží k přidání a odstranění objektu v simulaci. Scény načítáme pomocí metody loadScene(). Pro vykreslení všech objektů použijeme metodu render(). Hlavní smyčka aplikace je umístěna v metodě run(), kterou si rozebereme více.

```
1. #include <SDL.h>
2. #include "CollisionResolver.hpp"
3. #include "Object.hpp"
4. #include "RigidBodySystem.hpp"
5.
6. class Application {
7.    public:
8.     Application(int _fps = 30);
9.     ~Application();
10.      void addObject(Object* object);
11.      void removeObject(Object* object);
12.
13.      void render();
14.      void loadScene(SceneManager::Scene* scene);
15.      void run();
16.
17.      AppFlags appFlags;
18.      Renderer* m_renderer;
19.      SDL_Window* m_window;
20.      RigidbodySystem* m_rbSystem;
21.      CollisionResolver* m_cResolver;
22.      std::vector<Object*> m_objects;
23.      SceneManager::Scene* m_scene;  // current scene
24.
25.    private:
26.      int fps;
27. };
```

Kód 5: Hlavičkový soubor třídy Application

Hlavní smyčka je nejdůležitější část programu, voláme z ní všechny stěžejní metody, které se opakují každou iteraci. Jedna iterace ve smyčce vyprodukuje jeden snímek. Jelikož chceme, aby počet snímku za sekundu byl stálý, musíme ve smyčce měřit čas a každé iteraci přidělit konstantní časovou délku, za kterou by měla proběhnout. Pokud výpočty v simulaci proběhnou rychle a nám zbyde ještě čas, na konci smyčky spustíme delay délky zbývajícího času (k tomu slouží podmínka na konci smyčky). Oproti tomu, pokud nám čas nevychází, snížíme počet kroků ve výpočtech a samozřejmě tím také snížíme přesnost simulace. Pro měření času používáme knihovnu SDL, která obvykle dosahuje přesnosti nanosekund (záleží na hardwaru).

V hlavní smyčce se taky nachází další smyčka while, která slouží pro výpočtové kroky. Výpočty simulace můžeme dělat vícekrát za snímek pro dosáhnutí větší přesnosti. V samotné smyčce prvně voláme krokovou funkci, která pohne tělesy podle zákonů dynamiky. Následně metodou checkCollision() detekujeme a řešíme kolize, nakonec odečteme krokový čas od proměnné accumulator. Jakmile hodnota accumulator je rovna nule, musíme pokračovat dále v hlavní smyčce, kde objekty vykreslíme pomocí funkce render() na obrazovku. Ukázkový kód (Kód 5) je zjednodušený a ukazuje pouze nejdůležitější části hlavní smyčky.

```
1. const double expected_frame_time = 1000.0 / 30.0;
2. const double physics_time_step = expected_frame_time / 10.0;
3. double accumulator = 0.0;
4. Uint64 start = SDL_GetPerformanceCounter();
5. while (isRunning) {
6.     Uint64 current = SDL_GetPerformanceCounter();
7.     double elapsed = static_cast<double>(current - start) /
8.                      SDL_GetPerformanceFrequency();
9.     start = current;
10.      accumulator += elapsed;
11.
12.      // vypočty ve physics enginu
13.      while (accumulator >= physics_time_step) {
14.          rbSystem->step(physics_time_step);
15.          cResolver->checkCollisions(this);
16.          accumulator -= physics_time_step;
17.      }
18.      // Renderovaní
19.      render();
20.      SDL_RenderPresent(renderer->sdl_renderer);
21.
22.      Uint64 end = SDL_GetPerformanceCounter();
23.      double frame_time = static_cast<double>(end - current) /
24.                          SDL_GetPerformanceFrequency() *
25.                          1000.0;
26.      if (frame_time < expected_frame_time) {
27.          SDL_Delay(static_cast<Uint32>(expected_frame_time -
28.              frame_time));
29.      }
30.  }
```

Kód 6: Zjednodušená hlavní smyčka

3.2.6 Implementace dynamiky těles

Dynamiku těles ve třídě RigidbodySystem. Datové atributy třídy jsou dvě dynamická pole. První slouží pro ukazatele na instance třídy Rigidbody od každého objektu. Druhé pole slouží pro tzv. generátory síly. Jedná se o třídy, které implementují rozhraní ForceGenerator. Součástí rozhraní je metoda ApplyForce(), jejímž argumentem je pole ukazatelů na instance třídy Rigidbody. Generátory síly jsou užitečné, pokud chceme na všechna tělesa působit určitou silou. To se může hodit např. při implementaci gravitace.

Pro simulaci dynamiky byla původně ve Fyzix využita metoda Eulerova, ale později byla nahrazena za přesnější metodu RK4. Numerická metoda je implementována v metodě step() ve třídě RigidbodySystem. Metoda iteruje přes všechna tělesa a počítá nové hodnoty pomocí numerické metody. Implementaci Eulerovy metody můžeme vidět v Kódu 6.

Ve smyčce na konci musíme vynulovat datový atribut síly daného tělesa, jelikož jsme sílu přenesli do zrychlení tělesa. Metoda applyForces() volaná na začátku krokové funkce

aplikuje na všechny tělesa sílu ze všech aktivních generátorů sil. To samé je v krokové funkci, která implementuje metodu RK4 (viz. Kód 7).

```
1. void RigidbodySystem::step(double dt) {
2.     applyForces();
3.     const int s = rigidbodies.size();
4.
5.     for (int i = 0; i < s; ++i) {
6.         Rigidbody* rb = rigidbodies[i];
7.
8.         rb->a = rb->f / rb->m;
9.         rb->v += rb->a * dt;
10.          rb->pos += rb->v * dt;
11.
12.          rb->omega += rb->epsilon * dt;
13.          rb->theta += rb->omega * dt;
14.
15.          rb->f = {0, 0};
16.      }
17.  }
```

Kód 7: Implementace Eulerovy metody

```
1. void RigidbodySystem::step(double dt) {
2.     applyForces();
3.     const int s = rigidbodies.size();
4.
5.     for (int i = 0; i < s; ++i) {
6.         Rigidbody* rb = rigidbodies[i];
7.         Vec2 v0 = rb->v;
8.         Vec2 a0 = rb->f / rb->m;
9.
10.          double omega0 = rb->omega;
11.          double theta0 = rb->theta;
12.          double epsilon0 = rb->epsilon;
13.
14.          Vec2 vK1 = a0 * dt;
15.          Vec2 posK1 = v0 * dt;
16.          double omegaK1 = epsilon0 * dt;
17.          double thetaK1 = omega0 * dt;
18.
19.          Vec2 vK2 = ((rb->f / rb->m) + vK1 * 0.5) * dt;
20.          Vec2 posK2 = (v0 + vK1 * 0.5) * dt;
21.          double omegaK2 = (epsilon0 + omegaK1 * 0.5) * dt;
22.          double thetaK2 = (omega0 + omegaK1 * 0.5) * dt;
23.
24.          Vec2 vK3 = ((rb->f / rb->m) + vK2 * 0.5) * dt;
25.          Vec2 posK3 = (v0 + vK2 * 0.5) * dt;
26.          double omegaK3 = (epsilon0 + omegaK2 * 0.5) * dt;
27.          double thetaK3 = (omega0 + omegaK2 * 0.5) * dt;
28.
29.          Vec2 vK4 = ((rb->f / rb->m) + vK3) * dt;
30.          Vec2 posK4 = (v0 + vK3) * dt;
31.          double omegaK4 = (epsilon0 + omegaK3) * dt;
32.          double thetaK4 = (omega0 + omegaK3) * dt;
33.
34.          rb->v += (vK1 + vK2 * 2 + vK3 * 2 + vK4) / 6.0;
35.          rb->pos += (posK1 +  posK2 * 2 + posK3 * 2 + posK4)
   / 6.0;
36.
37.          rb->omega += (omegaK1 + 2.0 * omegaK2 + 2.0 *
   omegaK3 + omegaK4) / 6.0;
38.          rb->theta += (thetaK1 + 2.0 * thetaK2 + 2.0 *
   thetaK3 + thetaK4) / 6.0;
39.
40.          rb->f = {0, 0};
41.      }
42.  }
```

Kód 8: Implementace metody RK4

3.2.7 Implementace detekce kolizí

Pro detekci kolizí používá Fyzix metodu SAT. Jelikož je její implementace rozsáhlá a umístěna ve více souborech, rozebereme jen její nejdůležitější metody. Na celou implementaci můžete nahlédnout do zdrojového kódu [9] nebo také doporučuji anglický článek od dyn4j, který obsahuje vysvětlení i implementaci v Javě [4].

Jelikož máme ve physics enginu dva typy tvarů, musíme pro každou kombinaci naimplementovat vlastní metodu, takže bude metoda pro detekci kolize obdélníku s obdélníkem, kruh s obdélníkem a kruhu s kruhem. Základní SAT metoda funguje pouze s obdélníky (konvexními tvary), ale po lehké úpravě funguje i s kruhy. Ve skutečnosti je metoda s kruhy mnohem jednodušší na implementaci, a dokonce i méně náročná na výkon. Metody pro tento účel se nacházejí ve třídě CollisionResolver.

V implementaci je použita třída Axis (viz Kód 8), která je velmi podobná třídě Vec2, liší se pouze tím, že její hodnoty jsou vždy normalizované. Metoda vrací MTV vektor, pokud kolize mezi obdélníky není, vrátí nulový vektor. Zbytek metody funguje stejně, jak bylo popsáno v teoretické části (viz kapitola 2.3.2). Metoda pro detekci kolizi kruhu s kruhem funguje tak, že porovná, pokud je vzdálenost mezi středy kruhů větší než součet jejich poloměrů. Metoda pro detekci kolize mezi obdélníkem a kruhem funguje podobně jako standartní SAT, jediný rozdíl je, že iterujeme pouze po osách obdélníku, jelikož kruh osy nemá.

```
1. Vec2 CollisionResolver::detectCollision(const rectangleShape* r1,
2.                                         const rectangleShape* r2)
   {
3.     std::vector<Axis> axes1 = r1->getAxes();
4.     std::vector<Axis> axes2 = r2->getAxes();
5.     double minOverlap = std::numeric_limits<double>::max();
6.     Axis mtvAxis;
7.     for (int i = 0; i < axes1.size(); ++i) {
8.         const Axis axis = axes1[i];
9.
10.          Vec2 proj1 = r1->project(axis);
11.          Vec2 proj2 = r2->project(axis);
12.          if (!Math::overlap(proj1, proj2)) {
13.              return Vec2(0, 0);
14.          } else {
15.              double ol = Math::getOverlap(proj1, proj2);
16.              if (ol < minOverlap) {
17.                  minOverlap = ol;
18.                  mtvAxis = axis;
19.              }
20.          }
21.      }
22.
23.      for (int i = 0; i < axes2.size(); ++i) {
24.          const Axis axis = axes2[i];
25.
26.          Vec2 proj1 = r1->project(axis);
27.          Vec2 proj2 = r2->project(axis);
28.          if (!Math::overlap(proj1, proj2)) {
29.              return Vec2(0, 0);
30.          } else {
31.              double ol = Math::getOverlap(proj1, proj2);
32.              if (ol < minOverlap) {
33.                  minOverlap = ol;
34.                  mtvAxis = axis;
35.              }
36.          }
37.      }
38.      return Vec2(mtvAxis.getX() * minOverlap, mtvAxis.getY() *
   minOverlap);
39.  }
```

Kód 9: implementace SAT mezi dvěma obdélníky

3.2.8 Implementace řešení kolizí

Fyzix k řešení kolizí používá impulse-based model. Jeho implementace je v metodě resolveCollision() a jedná se převážně o převedení rovnic do kódové podoby. Argument metody je ukazatel na instanci třídy Collision, která ukládá informace o kolizi jako je MTV, ukazatele na objekty v kolizi a kolizní body. Dále v metodě můžeme vidět, že je rozdělena na dvě smyčky, první slouží pro výpočet impulsu a druhá smyčka impulsy aplikuje na tělesa. Proto si ukládáme impulsy a také jiné proměnné do pole (viz Kód 9).

```
1. void CollisionResolver::resolveCollision(Collision* collision) {
2.     Vec2 mtv = collision->mtv;
3.     Object* objectA = collision->A;
4.     Object* objectB = collision->B;
5.     std::vector<Vec2> cps = collision->contacts;
6.     std::array<Vec2, 2> impulses;
7.     std::array<double, 2> jArray;
8.     Rigidbody* rbA = objectA->shape->rigidbody;
9.     Rigidbody* rbB = objectB->shape->rigidbody;
10.      const double e = _e;
11.      Vec2 normal = mtv / mtv.magnitude();
12.
13.      Vec2 raArray[2];
14.      Vec2 rbArray[2];
15.
16.      for (int i = 0; i < cps.size(); ++i) {
17.          Vec2 ra = cps[i] - rbA->pos;
18.          Vec2 rb = cps[i] - rbB->pos;
19.
20.          raArray[i] = ra;
21.          rbArray[i] = rb;
22.
23.          Vec2 raNormal = ra.normal();
24.          Vec2 rbNormal = rb.normal();
25.
26.          Vec2 AngLinVelocityA = raNormal * rbA->omega;
27.          Vec2 AngLinVelocityB = rbNormal * rbB->omega;
28.
29.          Vec2 relativeVelocity = (rbB->v + AngLinVelocityB) -
30.          (rbA->v + AngLinVelocityA);
31.          double vAlongNormal = relativeVelocity.dot(normal);
32.
33.          double raNdotProd = raNormal.dot(normal);
34.          double rbNdotProd = rbNormal.dot(normal);
35.
36.          double j = -(1 + e) * relativeVelocity.dot(normal);
37.          double inverseMassSum =  (1 / rbA->m) + (1 / rbB->m);
38.
39.          j /= inverseMassSum +
40.               (raNdotProd * raNdotProd) * objectA->inverseInertia() +
41.               (rbNdotProd * rbNdotProd) * objectB->inverseInertia();
42.          j /= (double)cps.size();
43.
44.          Vec2 impulse = normal * j;
45.          impulses[i] = impulse;
46.      }
47.
48.      for (int i = 0; i < cps.size(); ++i) {
49.          Vec2 impulse = impulses[i];
50.          Vec2 ra = raArray[i];
51.          Vec2 rb = rbArray[i];
52.          rbA->v += -impulse / rbA->m;
53.          rbA->omega += -ra.cross(impulse) * objectA->inverseInertia();
54.          rbB->v += impulse / rbB->m;
55.          rbB->omega += (rb.cross(impulse) * objectB->inverseInertia());
56.      }
57.  }
```

Kód 10: Implementace řešení kolizí pomocí impulsů

3.3 UI a ostatní systémy aplikace

3.3.1 Uživatelské rozhraní

Pro grafické uživatelské rozhraní Fyzix využívá knihovnu Dear ImGui. Její hlavní výhoda je, že může být snadno integrovaná s SDL oknem. ImGui je určena hlavně pro vývojáře počítačových her, proto je navržena pro aplikace, které mají vlastní hlavní smyčku. To je pro physics engine ideální, protože do těchto aplikací patří. Do projektu je ImGui nalinkována staticky, to znamená že je naklonována v repozitáři (ve složce thirdparty).

Všechny funkce pro UI jsou umístěny ve stejnojmenném namespace (jmenný prostor). Každá z hlavních UI funkcí vykresluje určité okno nebo část rozhraní. Funkce renderSceneSelectWindow() vykresluje úvodní okno, kde se volí scéna simulace. Funkce renderSettingsWindow() slouží pro vykreslení okna pro nastavení celkové simulace. Pro vykreslení panelu nástrojů je určena funkce renderToolbar(). Poslední hlavní funkce UI je renderObjectWindow(), která vykreslí okno pro nastavení objektu.

```
1. #include "Application.hpp"
2. #include "SceneManager.hpp"
3.
4. namespace UI {
5.
6. void initUI(SDL_Renderer* renderer);
7. SDL_Texture* loadTexture(SDL_Renderer* renderer, std::string
   path);
8.
9. void renderSceneSelectWindow(Application* app,
10.                              std::vector<SceneManager::Scene*>&
   scenes);
11.  void renderSettingsWindow(Application* app, bool& open);
12.  void renderToolbar(Application* app,
13.                     bool& isSimRunning);
14.  void renderObjectWindow(Application* app, const int objectIdx,
15.                          bool &open);
16.  void renderObjectSettingsButton(std::vector<int>&
   objectWinIndices,
17.                                  int selected, Vec2 pos);
18.  void renderMainMenuBar(Application* app,
19.                         std::vector<SceneManager::Scene*>
   scenes,
20.                         bool& isRunning, bool& showSettings);
21.  }  // namespace UI
```

Kód 11: Hlavičkový soubor namespace UI

3.3.2 Ukládání scén

Součástí Fyzix je i ukládání scén. Scény ukládáme do složky s názvem „scenes“ ve formátu JSON, a proto, abychom mohli s formátem pracovat je v projektu použita knihovna

„nloohman json“. Všechny funkce pro správu scén jsou v namespace nazvaném SceneManager. Za zmínku stojí funkce sceneToJSON(), která převede aktuální stav simulace do JSON formátu. Také opačná funkce JSONToScene(), pomocí které uložené soubory načítáme zpět do aplikace. Scény si ukládají pouze tři informace, první je název scény, druhý je informace o tom, jestli má být v simulaci gravitace a poslední je seznam všech objektů v simulaci (viz třetí rádek v Kódu 10).

```
1.
2. namespace SceneManager {
3. struct Scene {
4.     std::string name;
5.     bool gravity = false;
6.     std::vector<Object*> objects;
7. };
8.
9. constexpr char SCENES_PATH[] = "../scenes/userScenes/";
10.
11.  std::vector<Scene*> loadScenes();
12.  void clearScenes(std::vector<Scene*> scenes);
13.
14.  void eraseFile(std::string name);
15.  void renameFile(std::string oldName, std::string newName);
16.
17.  Scene JSONToScene(nlohmann::json json);
18.  Vec2 JSONformatToVec(std::string jsonString);
19.  nlohmann::json sceneToJSON(Application* app);
20.  bool isUniqueName(std::string);
21.  void saveSceneToFile(Application* app);
22.  }  // namespace SceneManager
23.
```

Kód 12: Hlavičkový soubor namespace SceneManager

4. Závěr

Hlavní cíl práce – implementace physics engine, byl splněn v plném rozsahu. Projekt simuluje dynamiku těles, detekuje a řeší kolize, splňuje i dílčí cíle ze zadání jako je gravitace nebo tření. Místo demo ukázek je v aplikaci implementován editor scén, který umožňuje uživateli si simulaci nastavit podle svých parametrů. Projekt také obsahuje uživatelské rozhraní, které je nad rámec stanovených cílů maturitní práce. V teoretické části práce jsem se snažil vysvětlit základní metody pro fyzikální simulace a věřím, že se mi to podařilo.

Práce mi pomohla rozšířit znalosti a zkušenosti v programovacím jazyce C++, který jsem na začátku projektu znal pouze na začátečnické úrovni. Naučil jsem se pracovat s větším projektem a seznámil s problémem sestavování aplikace. Také jsem díky dokumentaci zlepšil svou schopnost formulovat myšlenky ve formě textu.

I přesto, že jsem dosáhl stanovených cílů, stále je zde velké místo pro budoucí zlepšení. Physics engine má hlavně nedostatky ve stabilitě a přesnosti simulace, tyto nedostatky především pochází z mé nedostatečné rešerše tématu před implementováním projektu, ale také je to absence mé zkušenosti s implementací fyzikální simulace. Do budoucna by bylo možné pracovat na optimalizaci enginu a využít pokročilejších metod pro získání větší přesnosti.

Použitá literatura a zdroje

[1] LEPIL, Oldřich. Deterministický Chaos. In: *Fyzika aktuálně: Příručka nejen pro učitele*. Prometheus, 2010, s. 172-174. ISBN 978-80-7196-381-3.

[2] WIKIPEDIA FOUNDATION. *Runge-Kutta methods*. Online. Dostupné z: https://en.wikipedia.org/wiki/Runge%E2%80%93Kutta_methods. [cit. 2025-02-17].

[3] Newcastle University: Numerical Integration Methods. Online. S. 1-7. Dostupné z: https://research.ncl.ac.uk/game/mastersdegree/gametechnologies/previousinformation/physics2numericalintegrationmethods/2017%20Tutorial%202%20-%20Numerical%20Integration%20Methods.pdf. [cit. 2025-02-20].

[4] *Dyn4j: Separating Axis Theorem.* Online. Dostupné z: https://dyn4j.org/2010/01/sat/. [cit. 2025-02-17].

[5] HECKER, Chris. Physics, Part 3: Collision Response. Online. S. 11-16. Dostupné z: https://www.chrishecker.com/images/e/e7/Gdmphys3.pdf. [cit. 2025-02-18].

[6] *Numerické metody*. Online. Dostupné z: https://www.askamathematician.com/2017/09/q-why-are-numerical-methods-necessary-if-we-cant-get-exact-solutions-then-how-do-we-know-when-our-approximate-solutions-are-any-good/. [cit. 2025-02-17].

[7] *Lorenzův atraktor*. Online. Dostupné z: https://www.researchgate.net/figure/The-phase-portraits-of-x-y-z-for-fractional-Lorenz-system-a-b-g-098-left-and_fig2_277927597. [cit. 2025-02-17].

[8] MOCNÝ, Ondřej. *Real-time fyzikální simulace pro mobilní zařízení*. Online, Bakalářská práce, vedoucí Mgr. Ondřej Sýkora. Praha: Univerzita Karlova v Praze Matematicko-fyzikální fakulta, 2009. Dostupné z: https://dspace.cuni.cz/bitstream/handle/20.500.11956/31029/BPTX_2008_1_11320_0_232322_0_65839.pdf?sequence=1&isAllowed=y. [cit. 2025-02-17].

[9]GABRHELÍK, Aleš. *Zdrojový kód Fyzix*. Online. Dostupné z: https://github.com/Ales-h/2DPhysicsEngine. [cit. 2025-02-23].

Seznam zkratek a značek

GUI – Graphical User Interface, Grafické uživatelské rozhraní

SAT – Separating Axis Theorem, Věta o oddělující ose

AABB – Axis-Aligned Bounding Box, metoda pro detekci kolize

SDL – Simple DirectMedia Layer

ImGui – Dear ImGui, knihovna grafického uživatelského rozhraní

MTV – Minimal Translate Vector – vektor, v jehož směru je potřeba posunout objekty, aby se nepřekrývali

JSON – JavaScript Object Notation, souborový formát pro ukládaní dat v objektech

Seznam obrázků a tabulek

Seznam kódů

Přílohy

Příloha č. 1: Zdrojový kód aplikace

Příloha č. 2: Sestavená aplikace pro uživatele