VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

Detekce kolizí v 3D scéně
Collision Detection in 3D Scene

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **TOMÁŠ PŘIBYL**
AUTHOR

VEDOUCÍ PRÁCE **Ing. JAN PEČIVA, Ph.D.**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2021/2022

Zadání bakalářské práce

Student: **Přibyl Tomáš**
Program: Informační technologie
Název: **Detekce kolizí v 3D scéně**
Collision Detection in 3D Scene
Kategorie: Počítačová grafika

Zadání:

1. Nastudujte si témata vztahující se k detekci kolizí v 3D scéně.
2. Navrhněte webovou aplikaci realizující detekci kolizí a jednoduchou fyzikální simulaci ve 3D scéně.
3. Navrženou aplikaci implementujte. Implementace může používat veřejně dostupné knihovny, jako například Babylon.js. Volitelně doplňte různé optimalizace vztahující se k detekci kolizí či fyzikální simulaci. Výsledná aplikace může být realizována formou hry.
4. Proveďte měření použitých kolizních a fyzikálních algoritmů.
5. Vyhodnoťte výsledky. Diskutujte možná budoucí vylepšení. Práci zveřejněte na internetu pod některou z open-source licencí, nebo zvolte jinou formu prezentace projektu (plakátek, apod.).

Literatura:

- dle doporučení vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- Prototyp funkční aplikace.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Pečiva Jan, Ing., Ph.D.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 11. května 2022
Datum schválení: 1. listopadu 2021

Abstrakt

Tato práce popisuje postup tvorby algoritmů pro detekci kolizí v Javascriptovém programovacím jazyku. Součástí práce jsou i testovací aplikace. Tyto aplikace zjistí vlastnosti algoritmů a změří jejich výpočetní čas.

Abstract

This thesis is focused on algorithms for collision detection in Javascript programming language. Part of this thesis is also about testing applications. These applications test properties of algorithms and measure their calculation time.

Klíčová slova

3D grafika, Babylon.js, detekce kolizí, grafika na webu, webový prohlížeč, WebGL

Keywords

3D graphics, Babylon.js, collision detection, graphics on web, web browser, WebGL

Citace

Přibyl, Tomáš. Detekce kolizí v 3D scéně. Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jan Pečiva, Ph.D.

Detekce kolizí v 3D scéně

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jana Pečivy Ph.D.. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal

. . . . . .

Tomáš Přibyl

10. května 2022

Poděkování

Chtěl bych poděkovat svému vedoucímu bakalářské práce Ing. Janu Pečivovi Ph.D. za ochotu, vstřícnost, pomoc a vedení.

Obsah

1. Úvod

Tato práce se zabývá algoritmy, které slouží k detekci kolizí mezi objekty ve trojrozměrném prostoru. Algoritmy na detekci kolizí najdou uplatnění především v počítačových hrách, výpočetní geometrii, fyzikálních simulacích, robotice nebo i optimalizaci jiných složitých a pomalých algoritmů na detekci kolizí, přičemž, pokud budou vytvořené algoritmy dostatečně účinné mohly by se v budoucnosti využít právě v těchto odvětvích. Tyto algoritmy nemusí být stoprocentně přesně a očekává se od nich určitá nepřesnost a nedokonalost. Především jde o to, aby algoritmy byly různorodé a využívaly velké množství poznatků a technik, které budou mít vliv na odlišné chování těchto algoritmů. Algoritmy nejen, že je potřeba navrhnout a implementovat, ale také v praxi vyzkoušet a změřit. Proto bude také potřeba vytvořit testovací aplikace, které umožní, jak je který algoritmus rychlý, jak si algoritmy poradí s různým počtem objektů, které jsou v kolizi, jak si algoritmy poradí s různým počtem kolizí a jak si poradí s různými objekty odlišných tvarů a které mají odlišnou rotaci. Vše bude implementováno v jazyce Javascript neboť testovací aplikace musí být webové a musí se dát otevřít v běžném internetovém prohlížeči. Na konci práce by tak měly vzniknout algoritmy o, kterých je známo jaké mají silné a slabé stránky, například u kterých objektů zachytí kolizi příliš pozdě, u kterých objektů zachytí kolizi příliš brzy, nebo jejich jakékoli nechtěné chování. To vše s grafy na kterých je jasně vidět rychlost těchto algoritmů a podmínky ve kterých této rychlosti bylo dosaženo, například počet kolizí nebo počet objektů. Kromě vytvořených algoritmů se také provede měření fyzikální knihovny Cannon.js. U této knihovny se ovšem neprovdané pouze měření výpočetní rychlosti pro kolize, ale i pro fyzikální výpočty. Změřením Cannonu.js by se také mělo zjistit, jaký zhruba by měly být průběhy grafů a jejich křivky u vytvořených algoritmů.

Nejdříve je nutné se věnovat teorii, ze které se získají vlastnosti použitelné při návrhu a implementace algoritmů. V této části ne nemluví pouze o samotných existujících algoritmech a jejich teorii, ale i o enginech, programech a aplikacích, které využívají nebo se zabývají detekcí kolizí. Tím se získají alespoň částečné informace, jak by algoritmy mohly ve výsledku vypadat a jaké mít přibližné vlastnosti, aby splňovaly požadavky profesionálních enginů a aplikací. Nakonec je ještě nutné seznámit se pokročilou geometrií a jejími výpočty, především s tou trigonometrickou. Algoritmy mohou matematiku využít k rotaci objektů a některé matematické poznatky využít k samotné detekci kolizí.

Následuje návrh a implementace. Zde se popisuje jak algoritmy pracují, jak fungují a jak jsou implementovány. Jsou tu tak popsány jejich teoretické vlastnosti, které se později ověří nebo vyvrátí během testování. U každého algoritmu je tak popsáno zda používá vestavěnou funkci, která je součástí WebGL frameworku, zda používá funkci převzatou od jiných vývojářů nebo zda využívá Axis-Aligned Bounding Box či Oriented Bounding Box. Pokud jsou převzaty nějaké funkce od jiných vývojářů, jsou zde tyto funkce podrobně popsány a je i vysvětleno k čemu slouží, aby bylo jasné,

že tyto funkce jsou pouhým nástrojem k dosažení výsledku a ne, že odvedou úplně celou práci.

Dále tu jsou testovací aplikace. Jsou to právě tyto aplikace, které odhalí a potvrdí vlastnosti implementovaných algoritmů. U každé aplikace se popíše, co aplikace testuje a jakým způsobem to testuje. Pokud aplikace testuje rychlost, bude u ní popsáno jak zajistí plynulý průběh zvyšujících se kolizí, které se mohou změřit. Pokud se testují vlastnosti, bude u ní popsáno na jakých objektech se tyto vlastnosti testují. K této části patří i výsledky měření. Jsou zde tak uvedeny grafy, které názorně zobrazují jak si který algoritmus vedl. Výsledky budou porovnány navzájem mezi sebou a budou i porovnány s existující knihovnou, která má svou vlastní detekci kolizí. Konkrétně to bude knihovna Cannon.js, která se používá k přidání fyziky do WebGL aplikací

Poslední částí je závěr. V této části se shrne čeho se dosáhlo a jak to zlepšit. Nejdříve se zhodnotí jestli se vytvořené algoritmy vůbec k něčemu hodí, například jestli se hodí na skutečnou detekcí kolizí, jestli je z nich užitečná pouze některá část nebo jestli se hodí k optimalizaci lepších a přesnějších, ale za to pomalejších algoritmů. Dále se uvažuje nad tím, jak tyto algoritmy zlepšit. Ať už použitím jiného programovacího jazyka, kombinací algoritmů a smíchání tak jejich částí nebo najít v implementaci chyby, které algoritmy akorát zpomalují. Nejdůležitější otázka, ale je zda bylo vytvořeno, to co se požadovalo.

2. Teorie

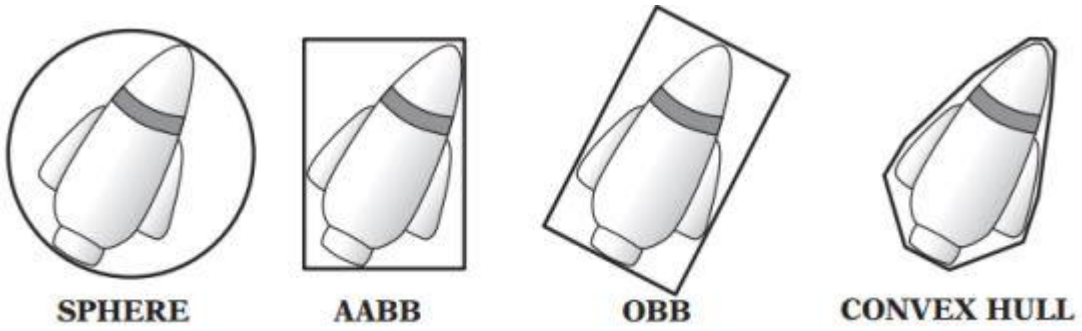
2.1. Detekce kolizí

Detekce kolizí jsou označeny algoritmy pro zjištění kolize dvou objektů, tj. zda existuje průsečík dvou daných těles. Bez detekce kolizí by postavy ve videohrách pouze procházely skrz zdi a jiné překážky. Ve hrách nemusí být kolize stoprocentně přesně jako v realitě. Stačí aby byly podobné a přitom se hráči hrála hra co nejlépe. Například v závodních hrách nemusí chování auta odpovídat vlastnostem reálného modelu, ale auto se ve hře musí řídit dobře. Při vyhodnocování kolizí nezáleží, zda se pohybující objekt, například vesmírná loď a meteorit naprosto přesně dotknou a odrazí se (může stát, že se trochu do sebe zanoří), podstatné je, aby nedocházelo vlivem výpočetní náročnosti k výraznému snížení snímkové frekvence a tím i k nežádoucímu zasekávání aplikace.

2.2. Optimalizační metody

Mnoho algoritmů na detekci kolizí pracuje dobře, pokud jsou využity na pouhou dvojici objektů. Pokud se, ale využijí ve scéně, kde jsou desítky nebo stovky objektů, pak je výpočetní náročnost příliš vysoká. Proto je dobré tyto algoritmy kombinovat s jednoduššími, které rozpoznají zda jsou objekty dostatečně blízko na to, aby vůbec

mohly být v kolizi. Tyto algoritmy objekty “obalí” jakýmsi obalem a poté zkouší, zda se neprotnou tyto obaly.



Obrázek 2.1: ukázka jak optimalizační algoritmy “obalí”

Existují čtyři základní typy těchto obalovacích algoritmů:

1. algoritmus - Sphere - Tento algoritmus je ten nejjednodušší, ale i nejméně přesný. Nejdříve obalí objekty jejich největším rozměrem. Poté zkoumá zda součet těchto rozměrů není větší než vzdálenost objektů.

2. algoritmus - Axis-Aligned Bounding Box - Tento algoritmus obalí objekt obálkou ve tvaru kvádru. K vytvoření tohoto kvádru stačí zjistit minimální a maximální hodnoty na hlavních osách. Ovšem při transformaci původního tělesa je nutno tuto obálku přepočítat na nové minimální a maximální hodnoty, což může mít nemalý dopad na výkonnost.

3. algoritmus - Oriented Bounding Box - I tento algoritmus obalí objekt do schránky ve tvaru kvádru, ale když se objekt rotuje, tak se rotuje i tato schránka místo toho, aby se prostě zvětšila. Algoritmus je tak mnohem účinnější, zvláště pro objekty nerovnoměrných tvarů jako například deska.

4. algoritmus - Convex Hull - Jedná se o nejpřesnější optimalizační algoritmus, ale také je nejvíce náročný na výpočtový výkon. V některých případech by tak mohl být spíše kontraproduktivní. Tento algoritmus obalí objekt body, které poté vhodně spojí a vytvoří tak obal, který velmi přesně napodobuje tvar objektu.

2.3. Three.js

Jedná se o Javascriptovou knihovnu, která slouží k vytváření a zobrazování 3D grafiky ve webovém prohlížeči a využívá přitom aplikačního rozhraní WebGL. Vytvořena byla roku 2010. Její první verze byla napsána v jazyce ActionScript. Ovšem nakonec byl zdrojový kód přepsán do JavaScriptu. Díky tomu se odstranila nutnost kompilace kódu před každým načtením a zajistila se i nezávislost knihovny na platformě. Jelikož je knihovna dostupná i v jednom souboru, tak k použití knihovny ve webové stránce ji stačí pouze zahrnout klasickým Javascriptovým

příkazem “src”. Tato knihovna funguje ve všech prohlížečích podporujících WebGL a HTML5. Je dostupná pod licencí MIT. Knihovna má samozřejmě i manuální stránky, ale tyto stránky nejsou vhodné pro začátečníky v Javascriptu, neboť jsou velmi stručné a u všech funkcí či modulů je popsán pouze základní popis. Není vysvětleno jak tyto funkce či moduly použít, ani nejsou k dispozici příklady.

2.4. Babylon.js

Babylon.js je Javascriptová knihovna sloužící k vytváření a zobrazování 3D grafiky ve webovém prohlížeči. Vytvořena byla roku 2013. Babylon.js nedokáže vytvářet fyzikální jevy a kolize mezi objekty. K tomuto účelu je k ní potřeba přidat fyzikální engine jako je Cannon.js nebo Oimo.js. Tato knihovna funguje ve všech prohlížečích podporujících WebGL a HTML5. Je dostupná pod licencí Apache License 2.0. Ke knihovně jsou dostupné i manuální stránky. Tyto stránky ve srovnání s Three.js stránkami jsou mnohem více propracovanější a detailnější. Nejen, že je zde podrobně vysvětleno, co která funkce či modul dělá, ale také jsou k dispozici jednoduché příklady využití těchto funkcí a modulů a dokonce je k dispozici i “playground”, což je editor kódu, který i ukazuje, co takový kod vytvoří. Uživatel si tak může v praxi vše vyzkoušet přímo v těchto manuálových stránkách. Babylon.js je tak velmi vhodnou knihovnou pro ty, kteří s Javascriptem nemají takové zkušenosti a potřebují detailnější instrukce.

2.5. Per-triangle detekce kolizí

Per-triangle detekce kolizí neobalí zkoumané objekty jedním přilehlým tvarem, jako je kvádr nebo koule, ale velmi těsně přilehlým obalem, který tvoří pospojované trojúhelníky. Při samotné detekce kolizí se poté navzájem testují tyto trojúhelníky jeden po druhém. Protože, ale takto se může otestovat tisíc trojúhelníků proti tisícu trojúhelníků, tak se tento algoritmus využívá spíše, pokud je přednější přesnost než rychlost. Nejde jen o množství trojúhelníků, ale i matematiku, která se využívá u tohoto algoritmu je velmi složitá. Počítá se nejen s vrcholy trojúhelníku, ale i s jeho hranami, normálami a některé implementace algoritmu využívají i rovnice roviny na které trojúhelníky leží a to ve tvaru Ax + By + Cz + D = 0.

2.6. Fyzikální enginy

Fyzikální engine je počítačový software , který poskytuje přibližnou simulaci určitých fyzikálních systémů, jako např. pohyby těles (včetně detekce kolizí) a pohyb kapalin. Jejich hlavní využití je ve videohrách, v takovém případě jsou simulace v reálném čase . Termín se někdy používá obecněji k popisu jakéhokoliv softwarového systému pro simulaci fyzikálních jevů, jako je vysoce výkonná vědecká simulace. Obecně existují dvě třídy fyzikálních enginů: v real-time a high-precision. High-precision fyzikální enginy vyžadují větší výpočetní výkon k výpočtu velmi přesné fyziky a obvykle je používají vědci a počítačově animované filmy. Real-time

fyzikální enginy se používají ve videohrách a dalších formách interaktivního počítání. Proto používají zjednodušené výpočty a sníženou přesnost pro výpočet včas, aby hra mohla reagovat přiměřenou rychlostí pro hraní hry.

Ve většině počítačových her je rychlost procesů a hratelnost důležitější než přesnost simulace. To vede k návrhům fyzikálních enginů, které produkují výsledky v reálném čase, ale replikují fyziku reálného světa pouze pro jednoduché případy a obvykle s určitou aproximací. Častěji je simulace zaměřena na poskytování „percepčně správné“ aproximace spíše než na skutečnou simulaci. Některé herní enginy, jako například Source vyžadují přesnější fyziku, aby například hybnost objektu mohla srazit překážku nebo zvednout potápějící se objekt. Fyzikálně založená animace postav v minulosti používala pouze dynamiku tuhého těla, protože je rychlejší a snadněji se počítá, ale moderní hry a filmy začínají používat fyziku měkkého tělesa. Fyzika měkkých těles se také používá pro částicové efekty, kapaliny a látky. Některá forma omezené simulace dynamiky tekutin je někdy poskytována pro simulaci vody a jiných kapalin, jakož i proudění ohně a explozí.

2.7. Princip fyzikálních enginů

Fyzikální engine je zodpovědný za řešení pohybové rovnice a za detekci kolizí. Fyzikální engine si můžete představit jako nepřetržitou smyčku. Začíná simulací vnější síly, jako je gravitace. Při každém novém časovém kroku detekuje případné kolize a poté vypočítá rychlost a polohu objektu. A nakonec odešle údaje o poloze do GPU. Tato nepřetržitá smyčka vytváří iluzi, že objekt padá vlivem gravitace.

Fyzikální engine je zodpovědný za výpočet výsledných zrychlení, rychlosti a posunutí objektu ze sil a točivých momentů působících na tělo. U fyzikálních enginů jsou důležité především dvě rovnice:

Rovnice druhého Newtonova zákona, která se používá k výpočtu síly jakou má těleso v pohybu.

Rovnice pro výpočet rotační síly nebo také moment síly, která se používá k výpočtu jaký má na těleso efekt rotace.

2.7.1. Cannon.js

Tento engine není přepsaný z C++ do Javascriptu, ale je v Javascriptu vyvíjen už od začátku a může tak využívat i jeho funkce. Ve srovnání s ostatními enginy je kompaktnější, srozumitelnější, výkonnější a také srozumitelnější, ale nemá tolik funkcí.

2.7.2. Oimo.js

Oimo.js je jednoduchý 3d fyzikální engine pro JavaScript. Jedná se o do Javascriptu přepsanou verzi enginu OimoPhysics. Pro kolizní geometrii využívá Koule, Box, Válec, Kužel, Kapsle a Konvexní obal.

2.7.3. Ammo.js

Jedná se o Bullet engine přeložený z jazyka C++ do Javascriptu bez jakékoliv lidské úpravy. Funkčně se tak jedná o naprosto stejný engine jako je Bullet.

2.8. Detekce kolizí cannon.js

Cannon.js testuje detekci kolizí ve třech fázích. První fáze se nazývá Broadphase a určuje které dvojice objektů se budou nadále testovat na kolize. Cannon.js tuto fázi řeší dvěma způsoby mezi kterými je možné přepínat. První způsob se nazývá NaiveBroadphase. Tento způsob objekty vůbec nefiltruje a pouze vytvoří dvojice všech objektů, které poté posílá na další testování. Druhý způsob, jakým řešit Broadphase, se nazývá SAPBroadphase. Tento způsob každou dvojici otestuje, zda jsou objekty dostatečně blízko sebe, aby vůbec mohly být v kolizi a to pomocí pozice objektů a jejich největšího rozměru. Druhá fáze detekce kolizí je určení objektů, kterých se kolize absolutně vůbec netýká. K zjištění tohoto se používá funkce jménem “needBroadphaseCollision”. Tato funkce pomocí filtrů odstraní dvojice objektů, pokud oba objekty jsou statické, ve “spánku” nebo neprojdou filtry, které má funkce k dispozici. Třetí a poslední fáze zkoumá zbylé dvojice objektů, zda se neprotínají a to pomocí funkce “intersectionTest”. Tato funkce zkoumá protínání dvěma možnými způsoby. První možnost je, že kolem objektů vytvoří Axis-Aligned Bounding Box a zkoumá zda se tyto Bounding Boxy nepřekrývají. Pokud tento způsob nezabere, tak funkce vypočítá největší rozměry obou objektů a vzdálenost mezi nimi. Pokud součet velikostí objektů je větší než vzdálenost mezi objekty, tak jsou objekty v kolizi. Čili Sphere

2.9. Srovnání enginů

Ammo.js je pro projekt naprosto nevhodný. Jelikož je strojově přeložen, tak je velice nepřehledný, bez jakéhokoli řádkování nebo komentářů a také jeho kod je ze všech enginů nejdelší. Oimo.js je sice výrazně kratší, ale i tento engine je strojově přepsán bez jakékoliv úpravy přímo v textu programátorem, tudíž opět žádné odřádkování ani komentáře. Cannon.js je oproti tomu vyvinut a napsán programátory. Obsahuje proto odřádkování, komentáře a také velice podrobnou dokumentaci, díky které je velmi snadné se v tomto velmi dlouhém kodu orientovat. Detekci kolizí v Cannon.js je tak možné pochopit a nastudovat. Tyto znalosti je poté možné využít při implementaci vlastních algoritmů pro detekci kolizí. Velice to i usnadní měření rychlosti zpracování detekce kolizí, neboť je snadné poznat kde detekce začíná a kde končí.

2.10. intersectsMesh() funkce

Jedná se o jednoduchou vestavěnou funkci v knihovně Babylon.js, která slouží k detekci kolizí. Určuje detekci mezi pouze dvěma objekty. K detekci využívá Bounding Box, kterým obalí oba objekty a zkoumá zda tyto Bounding Boxy nejsou v kolizi. Pokud jsou, funkce vrátí hodnotu boolean true. Pokud nejsou vrátí hodnotu boolean false. Funkce obsahuje nepovinný parametr, který určuje zda se objekty obalí Axis-Aligned Bounding Box nebo Oriented Bounding Box. Axis-Aligned Bounding Box má nižší výpočetní dobu, ale je mnohem méně přesný pokud je objekt rotován. Při rotaci totiž snímá detekci i v místech, kde objekt není, protože bounding box se při rotaci nerotuje spolu s objektem, ale pouze se zvětší. Oriented Bounding Box má vyšší výpočetní rychlost je ovšem velice přesnější u rotovaných objektů, protože v tomto případě se bounding box rotuje spolu s objektem a tak po rotaci je bounding box těsně u objektu. Pro účely projektu se použije Oriented Bounding Box verze, neboť se hledá funkce, která nabízí dobrou rovnováhu mezi přesností a vysokou výpočetní rychlostí, přičemž Axis-Aligned Bounding Box verze je pro tyto účely příliš nepřesná zato Oriented Bounding Box verze je dostatečně přesná i s poměrně uspokojivou výpočetní rychlostí. Obecně platí i to, že Oriented Bounding Box má především smysl používat pokud rotujeme a to se v této práci, pro účely testování, stane mnohokrát.

2.11. Funkce pro více FaceT

Jeden algoritmus pracuje s FaceT a s jeho polohami. Aby ale přesnost takového algoritmu byla dostačující je potřeba, aby objekt měl dostatek FaceT. Některé objekty s jednoduchými tvary a velkými rozměry, např. stěna, ovšem dostatek FaceT nemají. Proto je potřeba tyto objekty rozdělit na více FaceT, ale musí to být takové množství, které zaručí dostatečnou přesnost, ale také to musí být množství, které algoritmus příliš nezpomalí. Počet minimálního počtu FaceT byl stanoven na základě pokusů, kdy se jednomu objektu dal určitý počet FaceT a sledovalo se jak na tento počet zkoumaný algoritmus reaguje. Když byl počet příliš malý, algoritmus nezachytil kolizi. Když byl počet příliš velký, algoritmus byl příliš pomalý. A tak metodou pokus-omyl, byl stanoven ten nejvhodnější počet FaceT. Celkový počet FaceT musí být minimálně alespoň osmkrát větší než celková rozloha objektu. Funkce samotná byla převzata z manuálních stránek Babylon.js

2.12. Funkce pro detekci kolizí mezi trojúhelníky

Detekce kolizí tím složitější čím jsou geometricky složitější objekty, které jsou v kolizi, zvláště pokud jsou objekty v 3D prostoru a pokud jsou samotné objekty 3D. Pokud jsou ale objekty jednoduché jako např čtverec nebo koule, pak je možné pro ně i ve 3D prostoru napsat relativně jednoduché a přesné funkce na detekci kolizí. Tyto jednoduché funkce se potom dají využít pro detekci kolizí mezi geometricky složitými objekty tím, že se složité objekty rozdělí na vícero jednoduchých objektů, mezi

kterými se dokáže spolehlivě a rychle detekovat kolize i když to bude znamenat, že výsledný zjednodušený objekt nebude na sto procent odpovídat originálu. Ovšem objekt musí být rozdělen alespoň podle 2D objektů, protože detekce kolizí v 3D prostoru mezi obyčejnými úsečkami je sice velmi jednoduchá, ale k detekci mezi složitými objekty je velmi nevhodná. Kolem objektu se sice může vytvořit síť tvořená úsečkami a zkoumat zda tyto úsečky nějaká jiná úsečka neprotnula, ale to by těch úseček muselo příliš mnoho na to aby mezi nimi nějaký dostatečně malý objekt neproklouzl. Proto je vhodné používat na toto rozdělení trojúhelníky. Jedná se totiž o 2D objekt, kterým se v dostatečném množství dá obalit celý objekt a poté stačí na kolizi testovat jednotlivě tyto trojúhelníky navzájem proti trojúhelníkům jiného objektu. Funkce, která by tak dokázala zjistit zda jsou dva trojúhelníky v kolizi, by se tak mohla velice užitečně využít pro tvorbu algoritmů detekce kolizí. Jediné co by algoritmus musel zajistit, by bylo vhodné a přesné obalení objektu trojúhelníky, zapamatovat si souřadnice jejich hran a tyto souřadnice potom předat funkci na detekci kolizí mezi trojúhelníky. V této práci je s tímto účelem převzata funkce, která je součástí aplikace, která názorně ukazuje detekci kolizí mezi trojúhelníky. Ovšem tato aplikace a samotná funkce jsou napsány pomocí Javascriptové knihovny THREE.js zatímco aplikace v této práci jsou napsány pomocí Javascriptové knihovny Babylon.js. Ve funkci se tak musel změnit, způsob jakým se funkci předají informace o zkoumaných trojúhelnících. Zatímco v originální funkci se jako argumenty předávaly dva objekty typu “THREE.triangle”. V Babylonu ovšem takový objekt není. Proto se místo tohoto objektu předávají dvě array pole, kde každé pole obsahuje tři vektory. Každý vektor představuje souřadnice vrcholu trojúhelníku. Informace z těchto polí se poté předají proměnným a zbytek funkce může zůstat zachovalý. Funkce si nyní z těchto vrcholů vypočítá hrany ve formě směrového vektoru a také normály trojúhelníku. Pomocí série geometrických rovnic se tyto normály a hrany navzájem otestují na protnutí. Pokud se najde i jen jedno protnutí, pak jsou trojúhelníky v kolizi.

2.13. Matematika

Jeden algoritmus využívá pro detekci kolizí vektory. Vektor je definován jako prvek vektorového prostoru. Vektor směřující od bodu A do bodu B má nejen velikost, ale i směr. Algoritmus tohoto využívá a vypočítá vektor mezi jedním bodem jednoho objektu a jedním bodem druhého objektu podle vzorce:

$$\mathbf{a} = B - A$$

Rovnice 2.13.1:

a je výsledný vektor s velikostí a směrem. B je bod objektu, kde vektor má konec. A je bod jiného objektu, kde vektor má začátek.

Některé algoritmy využívají při svém řešení i pokročilejší matematiku. Jedním z jednodušších vzorců je Pythagorova věta pro 3D prostor. Tento vzorec se především

používá na zjištění vzdálenosti mezi dvěma body nebo celkový největší rozměr objektu. Vzorec vypadá takto:

$$a = sqrt(x*x + y*y + z*z)$$

Rovnice 2.13.2:

x je největší rozměr v ose X. y je největší rozměr v ose Y. z je největší rozměr v ose Z.

Některé algoritmy vytváří kolem objektů jakousi “hranici” a zkoumají zda tato hranice není překročena. Ovšem tato hranice se mění s rotací a algoritmus tak musí vypočítat nové rozměry a polohu této hranice. K tomu využívá řadu trigonometrických funkcí. Nejdříve se uvnitř objektu vytvoří pomyslný pravoúhlý trojúhelník pomocí pozice a rozměrů objektu. Poté se vypočítají vnitřní úhly tohoto trojúhelníku pomocí vzorce:

$$úhel = arcsin(odvěsna/přepona)$$

Rovnice 2.13.3:

Nyní je k dispozici dostatek informací k výpočtu nových rozměrů objektu a nebo pozice jeho hraničních bodů. Vypočítají se pomocí vzorců:

$$x' = r \times \cos(\alpha + \beta)$$

Rovnice 2.13.4:

$$y' = r \times \sin(\alpha + \beta)$$

Rovnice 2.13.5:

x′ a y′ jsou nové rozměry odvěsen trojúhelníku. r je rozměr přepony. α a β jsou původní úhel objektu a nový úhel objektu.

Jsou to právě tyto funkce, které zajišťují výrazné zvýšení přesnosti u velmi jednoduchých algoritmů, které nejsou náročné na výpočetní výkonnost.

Další algoritmus také počítá ůhly pro zvětšení přesnosti. Ovšem tento algoritmus pracuje s krajními body a s jejich souřadnicemi, což výpočet výrazně ulehčuje. Protože rotace vždy ovlivňuje pouze dvě souřadnice, tak pro každý bod a při každé rotaci se počítají pouze tyto ovlivněné souřadnice pomocí vzorců:

$$x' = x \times \cos(\beta) - y \times \sin(\beta)$$

Rovnice 2.13.6:

$$y' = y \times \cos(\beta) + x \times \sin(\beta)$$

Rovnice 2.13.7:

x′ a y′ jsou nové souřadnice bodů, které jsou ovlivněny rotací. β je úhel rotace.

Všechny výpočty se provádí právě třikrát, protože tolik je os ve, kterých se může objekt rotovat. Postupně se tak počítá každá osa, přičemž na pořadí nezáleží, neboť to nemá na konečný výsledek žádný vliv.

3. Návrh

Odlišné algoritmy detekce využívají dostupné funkce pro určení zda jsou objekty vůbec dostatečně blízko na to, aby mohly být v kolizi, dále na eliminaci objektů, které v kolizi být nemohou a nakonec na určení zda objekty vůbec v kolizi jsou. Pokud ano objekty změní barvu na znamení, že kolize je skutečně detekována.

1. algoritmus, algoritmus s vestavěnou funkcí - použije vestavěnou funkci Babylon.js, která určí zda jsou dva objekty v kolizi. Algoritmus bude akorát potřebovat dva objekty jako argumenty, které má zkoumat na kolizi.

2. algoritmus, algoritmus s největším rozměrem - bude pracovat s pozicí objektu a s jeho největším rozměrem. Algoritmus nejdříve vypočítá hrubý největší rozměr objektu podle Pythagorovy věty ve 3D do, které dosadí výšku, šířku a hloubku objektu. Poté získá vzdálenost mezi objekty pomocí podobného výpočtu do, kterého se dosadí souřadnice polohy objektu. Nakonec součet největších rozměrů porovná se vzdáleností mezi objekty. Pokud jsou rozměry větší než vzdálenost, jsou objekty v kolizi. Tento algoritmus využívá poznatky o Sphere algoritmu.

3. algoritmus, algoritmus s AABB - bude pracovat s pozicí objektu a s jeho rozměry - výška, šířka, hloubka. Algoritmus podle pozic vypočítá vzdálenosti v osách xyz, poté tuto vzdálenost srovná se součtem rozměrů objektů. Pokud je součet rozměrů menší než vzdálenost ve všech 3 osách, pak jsou objekty v kolizi. Je potřeba i upravit rozměry podle rotace objektu, jinak by algoritmus byl velmi nepřesný pro objekty s nevyváženými rozměry jako desky a tyče. Tento algoritmus využívá poznatky o Axis-Aligned Bounding Box algoritmu.

4. algoritmus, algoritmus s FaceT a počítadlem vzdálenosti - pracuje s FaceT. Každý objekt je rozdělen na části zvané FaceT. U těchto částí se dá zjistit jejich pozice. Pokud by tak u objektů byly dva nejbližší FaceT až příliš u sebe byla by to kolize. Pokud by nějaký objekt měl příliš málo FaceT, pak je potřeba počet FaceT rozšířit, aby algoritmus nebyl příliš nepřesný pro jednoduché objektu jako kostky. Pokud by naopak nějaký objekt byl příliš složitý a měl příliš FaceT, algoritmus by byl příliš pomalý, proto se některé FaceT budou vynechávat. Tento algoritmus využívá poznatky o Convex Hull algoritmu.

5. algoritmus, algoritmus s OBB - pomocí rozměrů a pozice objektu určí osm krajních bodů. Pozice bodů se mění i podle rotace kvůli přesnosti. Následně se z těchto bodů vytváří jakýsi 3D prostor a zkoumá se zda nějaký bod druhého objektu se nachází v tomto 3D prostoru tvořeném body prvního objektu. Pokud ne, provede se to samé, ale tentokrát se testuje přítomnost bodů prvního objektu ve 3D prostoru tvořeném body druhého objektu. Pokud je jakýkoliv nález bodu uvnitř objektu pozitivní, pak je to kolize. Tento algoritmus využívá poznatky o Oriented Bounding Box algoritmu.

6. algoritmus, algoritmus s FaceT a vektory - vezme zhruba každý dvacátý FaceT a vytvoří z těchto bodů jakousi schránku kolem objektu. Poté zkoumá zda objekty nejsou v kolizi podle těchto schránek. Dělá to tak, že mezi těmito body vytvoří objektů vytvoří navzájem směrové vektory. Pokud tyto směrové vektory míří aspoň ve dvou osách do všech čtyř směrů: vlevo-nahoru, vlevo-dolů, vpravo-nahoru, vpravo-dolů, znamená to, že objekty si čelí. Pokud je navíc třetí rozměr dostatečně krátký, pak jsou objekty v kolii. Pokud se kolize nedetekuje, tak se vektory vypočítají v opačném směru, tedy pokud se předtím počítaly od prvního objektu d druhému, nyní se budou počítat od druhého objektu k prvnímu. Pokud se ani tentokrát kolize nedetekuje, pak ke kolizi zřejmě nedošlo. Tento algoritmus využívá poznatky o Convex Hull algoritmu.

7. algoritmus, algoritmus s OBB a per triangle - bude velice podobný pátému algoritmu, ale bude mít odlišnou samotnou detekci kolizí. Pomocí rozměrů a pozice objektu určí osm krajních bodů. Pozice bodů se mění i podle rotace kvůli přesnosti. Těchto osm bodů tak tvoří kolem objektu jakési pouzdro ve tvaru kvádru kolem jakkoli geometricky složitého objektu. Nyní se testuje zda se tato pouzdra neprotínají. To se udělá tak, že se z těchto osmi bodů obou testovaných objektů vezmou tři body, vytvoří se z nich trojúhelník a tyto trojúhelníky se otestují na kolizi, pomocí funkce, která je relativně jednoduchá a velmi přesná. Postupně se tak na kolizi otestují všechny trojúhelníky obou objektů tvořené všemi body. Pokud alespoň jednou jsou jakékoli dva trojúhelníky v kolizi, pak jsou v kolizi i objekty. Narozdíl od detekce kolizí v pátém algoritmu se tato detekce provede pouze jednou a nikoliv dvakrát s odlišným pořadím objektů jako argumentů pro funkce. Tento algoritmus využívá poznatky o Oriented Bounding Box algoritmu.

8. algoritmus, algoritmus s FaceT a per triangle - bude velice podobný šestému algoritmu, ale bude mít odlišnou samotnou detekci kolizí. Algoritmus vezme zhruba každý dvacátý FaceT a vytvoří z těchto bodů jakousi schránku kolem objektu. Poté zkoumá zda objekty nejsou v kolizi podle těchto schránek, která se dělá úplně stejně jako u sedmého algoritmu. Opět se z každé schránky kolem objektu vezmou tři body, vytvoří se z nich trojúhelníky a tyto trojúhelníky se otestují na kolizi, dokud se kolize nedetekuje nebo dokud se neotestují všechny trojúhelníky jednoho objektu proti trojúhelníkům druhého objektu. Narozdíl od detekce kolizí v šestém algoritmu se tato detekce provede pouze jednou a nikoliv dvakrát s odlišným pořadím objektů jako argumentů pro funkce. Tento algoritmus využívá poznatky o Convex Hull algoritmu.

4. Implementace

Implementace je primárně v těle videohry a využívá tak funkce, které obsahuje knihovna Babylon.js. Díky tomu algoritmy nemusí pracovat pouze s polohou objektu a AABB, ale i například s FaceT. Pomocí měření pak můžeme tyto algoritmy srovnávat v účinnosti. Všechny algoritmy mají jednu společnou vlastnost a to je způsob jakým bere dvojici objektů, které otestuje na kolizi. Dělají to pomocí dvou for cyklů a seznamu všech mesh objektů v scéně. Každá dvojice meshů se otestuje pouze jednou, aby se neplýtvalo časem.

Přepínání mezi algoritmy - mezi algoritmy je potřeba přepínat přímo v aplikaci a ne tím, že se něco přepíše ve zdrojovém kódu a to tak aby uživatel věděl na jaký algoritmus právě přepnul a musí mu být i potvrzeno, že se přepnutí stalo, protože některé algoritmy mají natolik podobné vlastnosti a schopnosti, že pokud by uživatel zmáčkl tlačítko špatně algoritmus tím nepřepnul, vůbec by si toho nemusel všimnout. Přepínání algoritmu se tak provádí pouze jednou klávesou a to: G pro první algoritmus, H pro druhý algoritmus, J pro třetí algoritmus, K pro čtvrtý algoritmus, L pro pátý algoritmus, P pro šestý algoritmus, O pro sedmý algoritmus, I pro osmý algoritmus. Pokaždé když uživatel algoritmus přepne v konzoli se vypíše stručná hláška o změně algoritmu se stručným a jasným popisem na jaký algoritmus se zrovna přeplo.Tím může uživatel mezi algoritmy snadno přepínat a přitom mít přehled , co se zrovna děje.

4.1. Manuál přepínání

Jelikož přepínání algoritmů může být poněkud chaotické kvůli poměrně vysokému počtu algoritmů a protože klávesy na ovládání přepínání příliš nenapovídají, tak zde je stručný manuál, která klávesa přepíná na který algoritmus.

Klávesa G - algoritmus využívající vestavěnou funkci “intersectsMesh”.

Klávesa H - algoritmus, který vytváří kolem objektů kruhové pole, sphere.

Klávesa J - algoritmus, který vytváří kolem objektů AABB.

Klávesa K - algoritmus využívající FaceT a vzdálenost mezi nimi.

Klávesa L - algoritmus, který vytváří kolem objektů OBB a testuje přítomnost bodů uvnitř druhého objektu.

Klávesa P - algoritmus využívající FaceT a vytváří mezi nimi směrové vektory.

Klávesa O - algoritmus, který vytváří kolem objektů OBB a kolize detekuje pomocí trojúhelníků.

Klávesa I - algoritmus využívající FaceT a kolize detekuje pomocí trojúhelníků.

4.2. Implementace algoritmů

1. algoritmus, algoritmus s vestavěnou funkcí - použije vestavěnou funkci Babylon.js “intersectsMesh”, která určí zda jsou 2 objekty v kolizi. Jedná se o jednoduchou funkci která pouze potřebuje jména dvou objektů, které otestuje. Naprosto vyhovující pro ten nejjednodušší algoritmus.

2. algoritmus, algoritmus s největším rozměrem - nejdříve se získají rozměry objektu pomocí “getBoundingInfo”, které umožní získat maximální rozměr objektu v každé xyz ose. Z těchto tří rozměrů se vypočítá přibližný největší rozměr objektu a to pomocí vzorce pro Pythagorovu větu s tím, že se do něj přidá třetí rozměr. Poté se získá vzdálenost objektů. Použijí se k tomu souřadnice objektů. Stejně jako se získal největší rozměr objektu se nyní získá vzdálenost mezi objekty a to tak, že do toho samého vzorce se dosadí rozdíly souřadnic v daných osách. Nakonec se získané rozměry a vzdálenost porovnají. Pokud je součet rozměrů objektů větší než vzdálenost, jsou objekty v kolizi.

3. algoritmus, algoritmus s AABB - nejdříve se získají rozměry objektu pomocí “getBoundingInfo”, které umožní získat maximální rozměr objektu v každé xyz ose. Tento rozměr se připočte k pozici objektu. Každý rozměr ke své ose, čímž se získaly informace o tom, na jakých souřadnicích má objekt hranice. Poté se vypočte nový rozměr objektu podle toho jaký má objekt rotaci. K tomu slouží několik trigonometrických rovnic a funkcí které nejdříve vypočítají potřebné úhly a poté nové rozměry. Výpočet probíhá třikrát pro každý úhel zvlášť a s novými rozměry. Jakmile se získají nové rozměry objektu, srovnají se s pozicemi a rozměry druhého objektu. Pokud vzdálenost objektů je ve všech třech osách xyz menší než součet rozměrů objektů v daných osách pak jsou objekty v kolizi.

4. algoritmus, algoritmus s FaceT a počítadlem vzdálenosti - nejdříve algoritmus zjistí kolik má objekt FaceT. Pokud je příliš málo, což by vedlo k nepřesnostem, tak počet FaceT v objektu zvýší. Poté vybere dvojici FaceT, každý FaceT z jiného objektu, které si jsou nejblíže. Nakonec algoritmus změří vzdálenost mezi těmito dvěma FaceT pomocí Pythagorovy věty pro 3D prostor a pokud jsou tyto dva FaceT dostatečně a přijatelně blízko jsou, pak to je kolize. Výpočet vzdálenosti ovšem neobsahuje výpočet odmocniny. Místo toho se provedla druhá mocnina maximální povolené vzdálenosti mezi FaceT, aby výpočty byly rychlejší. Původně se vzdálenost těchto dvou FaceT zkoumala podle funkce, která na daných souřadnicích dokáže zachytit FaceT, ale tato funkce se projevila jako nepřesná s nečekaným a nevhodným chováním. Tento algoritmus je velice pomalý a je proto nutné ho kombinovat s rozšířením, které testuje i přibližnou vzdálenost nejbližších FaceT předtím, než je otestuje na kolizi. Toto řešení původně vybralo dva nejbližší FaceT z obou objektů, ale ze čtyřikrát menšího množství FaceT a také minimální vzdálenost, kterou FaceT mezi sebou musí mít, aby se mohla otestovat kolize, je o něco větší. Toto rozšíření nezkoumá jednotlivě xyz osy, ale vypočíta z nich pouze jedno číslo.

Ovšem toto řešení nebylo dostatečně účinné pro větší počet objektů a proto se použilo rozšíření s tím samým principem, který využívá 3. algoritmus, tedy Axis-Aligned Bounding Box, protože se jedná o střední cestou mezi přesností a rychlostí. Tím je algoritmus výrazně rychlejší, pokud k žádným kolizím nedochází. Testy ovšem ukázaly, že algoritmus potřebuje nějak zrychlit i samotnou detekci kolizí, která je příliš pomalá pro objekty s příliš a zbytečně velkým počtem FaceT. Proto algoritmus vezme přibližnou rozlohu objektu, celkový počet FaceT objektu a podle toho vypočítá kolik FaceT si algoritmus může dovolit přeskočit a netestovat a tím zvýšit rychlost a zachovat si přesnost. Velkou výhodou tohoto algoritmu je jeho poměrně dobrá přesnost i pro velice nepravidelné objekty.

5. algoritmus, algoritmus s OBB - začne určením pozice osmi krajních bodů a to pomocí pozice objektu a jeho rozměrů v “getBoundingInfo”. Pro lepší manipulaci se tyto objekty uloží do array pole. Nyní se musí vzít v potaz rotace. Nejdříve se rotuje jeden objekt podle své rotace a podle své osy pomocí trigonometrických rovnic. Protože rotace druhého objektu by ztížila poznat zda se v tomto objektu nachází bod prvního objektu, tak se znovu rotuje první objekt podle znegativované rotace druhého objektu a podle pozice druhé objektu. Nyní se snadno podle polohy bodu a prvního body a poloh osmi bodů druhého objektu dá zjistit, zda tento bod leží uvnitř těchto osmi bodů. Jelikož se osmi body nemanipulovalo, tak jsou tyto body dokonale kolmé nebo rovnoběžné s osami xyz. Pakliže žádný bod prvního objektu neleží v prostoru vymezeném osmi body druhého objektu, tak se celý proces zahrnující dvojitou rotaci a test zda nějaký jednotlivý bod neleží v prostoru vymezeném osmi body jiného objektu, provede znovu, ale tentokrát se objekty prohodí a testuje se přítomnost bodů prvního objektu uvnitř druhého objektu. Nevýhodou algoritmu je, že zkoumá pouze krajní body a nikoli celý povrch objektu, což je velice špatné pro přesnost, protože takto algoritmus nezachytí kolizi objektů, které jsou rotované a tím je umožněno aby byly v kolizi pouze prostředky svých těl zatímco jejich kraje v kolizi nejsou.

6. algoritmus, algoritmus s FaceT a vektory - vezme každý dvacátý FaceT obou zkoumaných objektů a uloží si jejich pozice do array pole. Algoritmus nepočítá kolik FaceT přeskočí podle plochy jako 4. algoritmus, algoritmus s FaceT a počítadlem vzdálenosti, protože testy prokázaly, že na konstantní přeskočení reaguje algoritmus lépe. Poté mezi body jednoho objektu a body druhého objektu začne vytvářet směrové vektory. Tyto vektory jsou poté podrobeny analýze, která se zapíše do daných proměnných. Zkoumá se do jakých směrů vektory směřují. Pokud by první objekt byl od druhého objektu příliš daleko, všechny směrové vektory by byly dlouhé a mířily by víceméně stejným směrem, ale pokud by objekty byly velice blízko, tak se vektory roztáhnou a budou směřovat do všech možných stran a vektor se poté drasticky zmenší. To je pak kolize. Algoritmus si proto zapisuje do jakého směru každý vektor směřuje, ale pouze ve dvou rozměrech. Jakmile algoritmus zjistí, že vektory v rámci jakýchkoliv dvou os směřují do všech čtyř směrů: vlevo-nahoru, vlevo-dolů, vpravo-nahoru, vpravo-dolů a že ve třetí ose má vektor dostatečně malý

rozměr aby mohlo dojít ke kolizi, tak algoritmus vyhodnotí situaci jako kolizi. Pokud je detekce neúspěšná, vypočítá se vše ještě jednou, ale tentokrát budou vektory směřovat z druhého objektu do prvního Právě tato vektorová technika by měla zajistit, že body objektů si musí být blízké pouze v jednom rozměru, díky čemuž algoritmus nepotřebuje tolik FaceT informací, jako ostatní algoritmy pro správné fungování a uspokojivou přesnost.

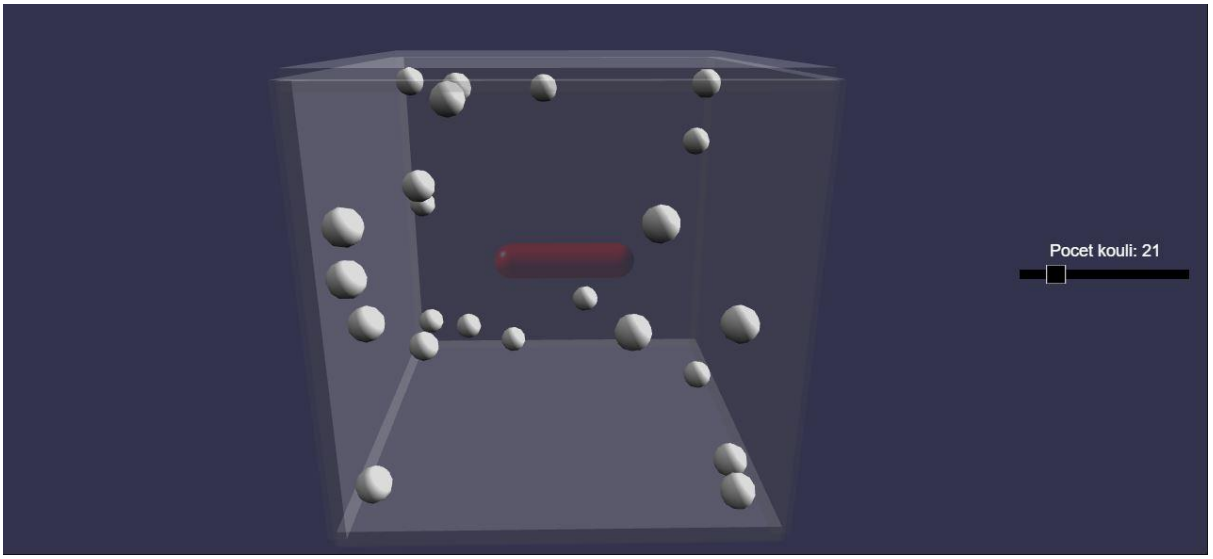
7.algoritmus, algoritmus s OBB a per triangle - začátek je téměř stejný jako v 5. algoritmu, algoritmu s OBB. Začne určením pozice osmi krajních bodů a to pomocí pozice objektu a jeho rozměrů v “getBoundingInfo”. Pro lepší manipulaci se tyto objekty uloží do array pole. Nyní se musí vzít v potaz rotace. Každý z objektů se otočí kolem své vlastní osy podle svých rotací. Pro každý objekt se provedou tři výpočty rotací neboť tolik je os podle kterých se mohou objekty rotovat. Rotaci není potřeba dělat komplikovanějším způsobem jako u 5. algoritmu, algoritmu s OBB, protože pro samotnou detekci to žádný rozdíl není. Pouze potřebuje souřadnice krajních bodů. Nyní když algoritmus zná osm přesných krajních bodů obou objektů, tak z těchto bodů začne vytvářet trojúhelníky a tyto trojúhelníky otestuje na kolizi pomocí funkce. Pokud se i jen jednou zaznamená kolize mezi jakýmikoli trojúhelníky objektů, je to kolize. Narozdíl od detekce kolizí pátého algoritmu, tato detekce se nemusí vykonávat dvakrát pro oba objekty zvlášť, což velice zvyšuje rychlost. Algoritmus také zkoumá celý povrch objektů a ne pouze pozici krajních bodů, takže je i mnohem přesnější.

8. algoritmus, algoritmus s FaceT a per triangle - začátek je úplně stejný jako v 6. algoritmu, algoritmu s FaceT a vektory. Vezme každý dvacátý FaceT obou zkoumaných objektů a uloží si jejich pozice do array pole. Potom, ale pozici těchto FaceT zvětší aby obal kolem objektu byl větší než samotný objekt. Jinak by se stalo, že při kolizích by algoritmus žádné kolize nezaznamenal, protože obal by byl příliš malý na srážku s obalem druhého objektu. Algoritmus nepočítá kolik FaceT přeskočí podle plochy jako 4. algoritmus, protože testy prokázaly, že na konstantní přeskočení reaguje algoritmus lépe. Zbytek je poté stejný jako u 7. algoritmu. I zde se z bodů, které obklupují objekt vytváří trojúhelníky a tyto trojúhelníky se testují na kolizi. Pokud se jen jednou zachytí kolize trojúhelníků objektů, pak jsou v kolizi i objekty. Narozdíl od 7. algoritmu nemusí tento algoritmus počítat s rotací objektů a na rozdíl od 6. algoritmu, algoritmus s FaceT a vektory, nemusí provádět detekci kolizí pro oba objekty zvlášť. Tím by měl algoritmus být přesnější než 7. algoritmus a rychlejší než 6. algoritmus, algoritmus s FaceT a vektory. Po testování, které zahrnovalo velký počet objektů blízko sebe, byl algoritmus rozšířen o optimalizační rozšíření, které způsobí, že funkce na detekci kolizí mezi trojúhelníky se nespustí, dokud nejsou objekty dostatečně blízko sebe. Jedná se o rozšíření s tím samým principem, který využívá 3. algoritmus, tedy Axis-Aligned Bounding Box protože se jedná o střední cestou mezi přesností a rychlostí. Algoritmus se tím zrychlil do takové míry, že ho bylo alespoň možné využít, i když byl chod aplikace výrazně pomalý.

4.3. Testovací aplikace

Testovací aplikace jsou implementovány pomocí knihovny Babylon.js. Tato knihovna byla vybrána pro rozsáhlé množství funkcí a velmi dobře zpracovanou dokumentaci. Díky tomu bylo možné se rychle naučit, jak napsat objekty, které se pohybují s kolizemi a které je možné ovládat stiskem klávesy či myši. Fyzikální vlastnosti objektů, jako je gravitace, kolize a odraz při nárazu, zajišťuje knihovna Cannon.js. Pro účely testování byly vytvořeny tři testovací aplikace.

1. testovací aplikace - Kostka

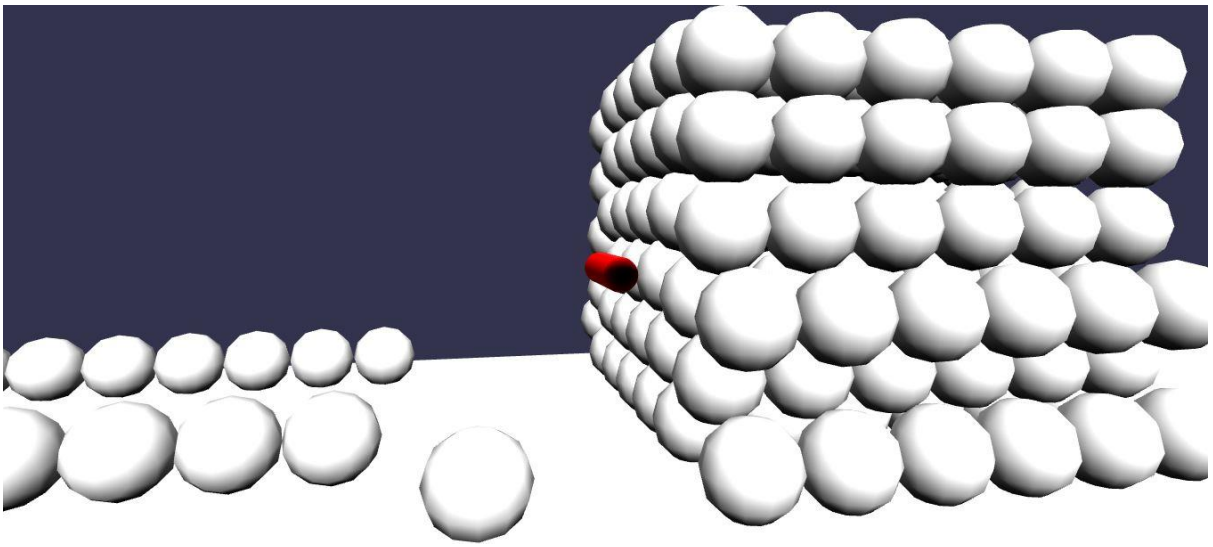


Obrázek 4.1: grafické rozhraní aplikace “kostka”

Tvoří ji uzavřená krychle, tvořena šesti deskami ve které se generují koule. Desky, které tvoří stěny krychle jsou naschvál zesíleny, aby se zabránilo chybám, při kterých koule proletí skrz zeď, pokud na ni narazí příliš rychle. Koule se generují uvnitř krychle a to v náhodném místě a s náhodnou rotací. Pokud koule nejsou v dostatečném pohybu, je možné je rozpohybovat pomocí klávesnice do jednoho ze čtyř směrů. Z tohoto důvodu mají koule náhodnou rotaci, aby se při stisku dané klávesy, každá koule pohybovala jiným směrem. Počet vygenerovaných koulí lze ovládat pomocí slideru, který počet koulí neustále zvětšuje. Stačí sliderem pohnout v jakémkoli směru a podle toho jak velký pohyb byl se vygeneruje daný počet koulí. Zmenšení počtu koulí pro testování není potřeba.

Tato aplikace především testuje jak algoritmy reagují na dané množství objektů v jedné scéně, pokud se tyto objekty neustále pohybují a jsou neustále s něčím v kolizi. Bude algoritmus plynule běžet když se ve scéně nachází padesát objektů, nebo se aplikace zasekne k nepoužití už při deseti objektech? Tato aplikace na tuto otázku spolehlivě odpoví.

2. testovací aplikace - Padání

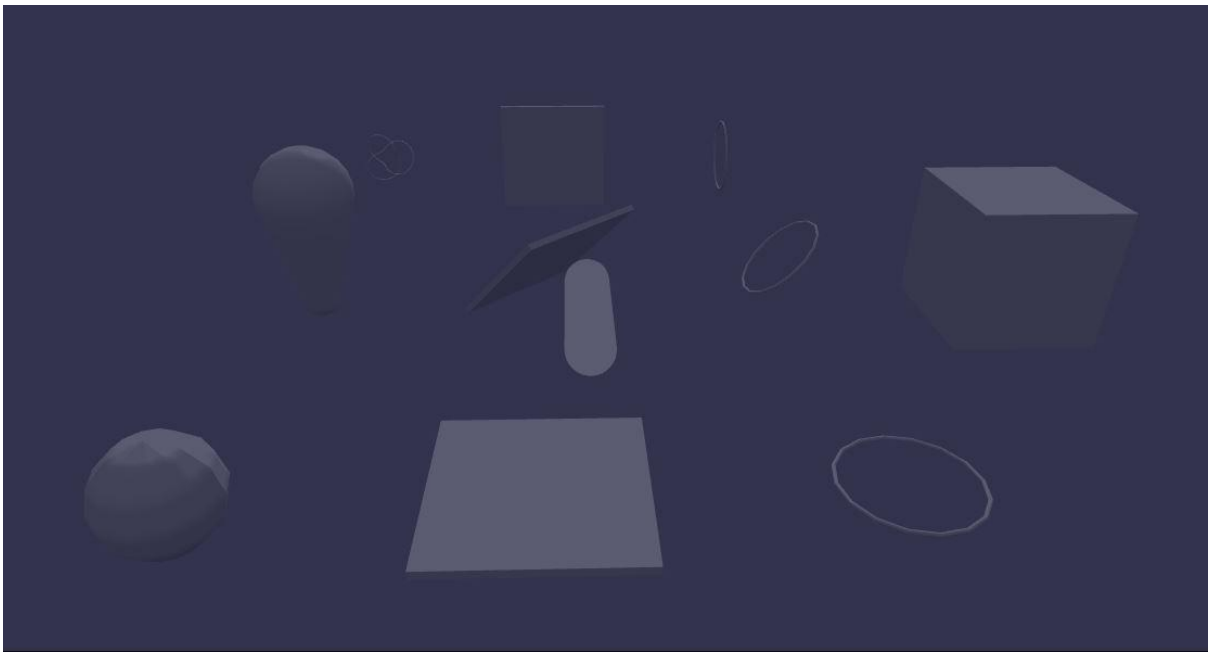


Obrázek 4.2: grafické rozhraní aplikace “padání”

Tvoří ji obrovská rozsáhlá deska, na které se nachází tři sety koulí. V prvním setu se na desce nachází jedna koule a nad ní se nachází také jedna koule. V druhém setu se na desce nachází deset koulí a nad nimi se nachází také deset koulí. Ve třetím setu se na desce nachází třicet šest koulí a nad nimi se nachází dvě stě šestnáct koulí. Koule, které se nachází ve vzduchu, dopadnou pomocí gravitace na koule pod sebou. Díky tomu v jaké jsou odlišné výšce, tak nejdříve dopadne na desku koule z prvního setu, poté co se první set uklidní, dopadnou na desku koule z druhého setu a až se uklidní i druhý set, tak na desku dopadnou i všechny koule ze třetího setu. Zkrátka tak, aby všechny sety dopadly postupně jeden po druhém a nikoli najednou. V této aplikaci jako v jediné je aktivována gravitace, neboť je potřeba aby pohyb objektů byl pravidelný, přirozený a neovládaný uživatelem.

Tato aplikace především testuje rychlost. Proto aplikace při každém spuštění, dělá ty samé pohyby, při kterých se stupňuje počet objektů v kolizi. Algoritmy tyto kolize musí detekovat. Nakonec se vše změří a zapíše do grafu. Důležitá není pouze rychlost, ale i průběh grafu. Jak moc dobře reaguje algoritmus na malý počet kolizí a jak zareaguje na prudký nárůst počtu kolizí? Ovlivní vůbec počet kolizí rychlost algoritmu, nebo je rychlost algoritmu pouze dána počtem objektů ve scéně?

3. testovací aplikace - Hřiště



Obrázek 4.2: grafické rozhraní aplikace “hřiště”

Tvoří ji “hřiště” složené z různých objektů odlišných tvarů, velikostí a rotací. Tyto objekty tvoří tlustá deska ve tvaru kvádru a torus. Tyto dva objekty se v aplikaci vyskytují třikrát, pokaždé s odlišnou rotací. Nejprve bez rotace, poté s rotací 45 stupňů a nakonec s rotací 90 stupňů. Dále tu je koule, kapsule s odlišnými poloměry, “torus knot” a velká kostka. Nakonec tu je objekt ve tvaru kapsule, který uživatel může ovládat a který mění tvar, pokud je v kolizi s jakýmkoliv jiným objektem. Zatímco objekty v předchozí aplikaci mezi sebou kolidovaly a odrážely se od sebe, v této aplikaci kapsle projde skrz objekty, protože se tím velmi usnadní testování. Kapsli je totiž takto možné dát do jakékoliv libovolné pozice a navíc je tak i možné testovat, jak algoritmy reagují na průniky objektů.

Uživatel nyní ovládá kapsli a sleduje, zda je v kolizi s jinými objekty. Sleduje jestli není kolize zaznamenaná příliš brzy nebo příliš pozdě. Z tohoto důvodu se v aplikaci nachází tolik odlišných tvarů. Aby se zjistilo zda si algoritmy poradí s odlišnými rozměry, s odlišnými tvary, s mezerou uvnitř objektu a s rotací. Testuje se tak pouze čirá schopnost algoritmu rozpoznat kolizi. Sleduje se především zda algoritmus nezachytí kolizi příliš brzy nebo příliš pozdě. Pokud ano, tak do jaké míry. Reaguje algoritmus hůře na nějaké tvary? Jak algoritmus reaguje na rotaci? Rychlost se sleduje pouze do té míry, aby byl algoritmus použitelný. Nemá cenu mít přesný algoritmus, který způsobí zmrazení aplikace.

4.4. Ovládání kapsule

Ovládání kapsule ve všech třech testovacích aplikací je důležité, aby uživatel mohl snadno měnit pohled a sledovat, co se zrovna v aplikaci děje. Hlavně ve třetí

testovací aplikaci, kde se testují kolize proti různým objektům s různou rotací je ovládání této kapsule naprosto důležité, aby uživatel mohl kolize pohodlně a plnohodnotně testovat. Kapsule tak musí být schopna pohybu ve všech směrech a rotace, ale zároveň je nutné aby ji bylo možné ovládat pouze dvěma rukama, proto ovládání nesmí být příliš složité a nezahrnovat příliš kláves.

Pohyb nahoru a dolů se tak provádí pomocí kláves W a S. W klávesa pohybuje s kapslí směrem nahoru. S klávesa pohybuje s kapslí směrem dolů. A to tak, že při stisknutí klávesy se buďto ubírá nebo přibírá hodnota pozice kapsule. Samozřejmě pouze v jedné ose.

Pohyb do stran se provádí klávesami A a D. A klávesa pohybuje s kapslí směrem vlevo. D klávesa pohybuje s kapslí směrem doprava. A to tak, že při stisknutí klávesy se kapsule rotuje v jedné ose do strany. Tím kapsle vykonává pohyb, jako když se u auta otáčí volantem.

Pohyb dopředu a dozadu se provádí kolečkem myši. Pohyb myši dopředu rozpohybuje kapsli směrem dopředu. Pohyb myši dozadu rozpohybuje kapsli směrem dozadu. Oba pohyby mají dvě rychlosti. Pokud tak uživatel hýbne kolečkem dvakrát za sebou ve stejném směru, kapsle vykoná stejný pohyb ale rychleji. Pokud bude i nadále hýbat kolečkem ve stejném směru, pohyb kapsle se nezmění. Pokud poté pohne jednou kolečkem v opačném směru, kapsle zpomalí. Pokud pohne kolečkem v opačném směru podruhé, kapsle se zastaví. Pokud pohne kolečkem v opačném směru potřetí, kapsle se dá do pohybu v opačném směru než předtím. Pokud pohne kolečkem v opačném směru počtvrté, kapsle zvýší rychlost pohybu, který již provádí.

Kamera, která je připevněná ke kasli se ovládá pohybem myši jako ve videohrách. Při pohybu myší, vykonává kamera rotační pohyby kolem kapsle a přitom si od kapsle zachovává vzdálenost. Po spuštění aplikace ovládání pohybu kamery myší není aktivováno, to proto aby uživatel mohl s myší manipulovat sliderem na ovládání počtu koulí v první testovací aplikaci. Pokud uživatel chce přepnout do módu ovládání kamery, musí stisknout levé tlačítko myší. Pokud chce uživatel přepnout zase zpět, aby mohl pomocí kurzoru a myši ovládat slider, musí zmáčknout klávesu ESC. Tím je možné mezi oběma módy pohodlně a snadno přeskakovat.

Původně se směry nahoru, dolů, doprava, doleva ovládaly pouze rotací, ale to vedlo k nechtěnému přetočení a kapsle se tak pohybovala křivě a bylo i snadné otočit kapsli vzhůru nohama, což ovládání ještě zhoršilo. Pokud by se směry ovládaly pouze pomocí manipulace s pozicí, pak by se kapsle vůbec nerotovala a některé testy a kolize by tak vůbec nebyly možné. Všech šest směrů se původně ovládalo pouze pomocí kláves, jenže pohyby, které kapsle vykonává jsou 3D zatímco rozložení kláves je 2D. Kvůli tomu bylo ovládání chaotické a uživatel snadno zapomněl, která klávesa ovládá který směr. Klávesy navíc musely být blízko sebe, aby je uživatel mohl ovládat jednou roku, neboť druhou ovládá myš a pohyb kamery.

Tím, ale byla zátěž a nároky na kooperaci na prsty levé ruky příliš vysoké. Tím, že se ovládání rozdělilo rovnoměrně mezi ovládání klávesou a ovládání myší, zaručuje uživateli snadné ovládání, komfort a výrazné usnadnění testovacích aplikací.

5. Měření a testování

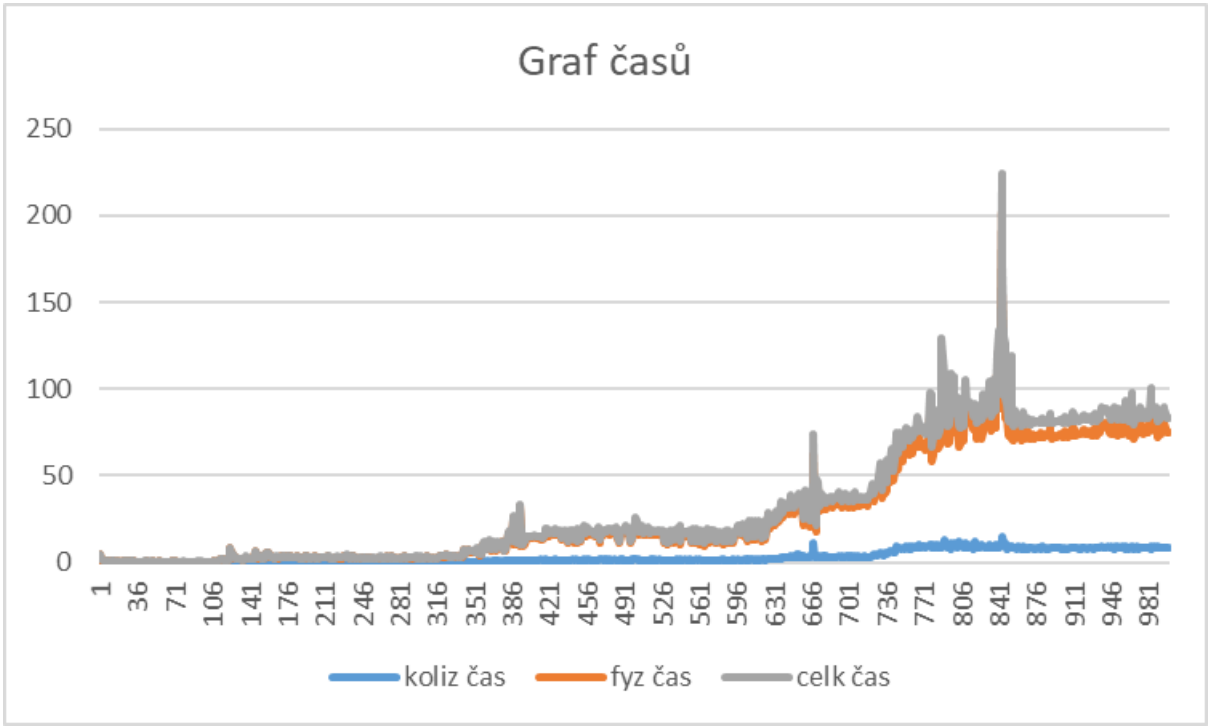
Měřením se zjistí rychlost algoritmu detekce kolizí. Testováním v praxi, čili aplikováním algoritmu do programu a sledováním chování se posoudí schopnosti a vlastnosti algoritmu. Tyto schopnosti, vlastnosti a rychlost se srovnají především s rychlostí a vlastnostmi algoritmu detekce kolizí v Cannon.js a algoritmem využívající vestavěnou Babylon.js funkci “intersectsMesh”, neboť Cannon.js i “intersectsMesh” funkce využívají funkční algoritmy, využívaný fyzikálním enginem a 3D enginem Javascript napsané ve stejném programovacím jazyce. Měření rychlosti detekce kolizí probíhá pomocí funkce na záznam času a získaný údaj je poté vypsán do konzole pomocí funkce console.log(). Na začátku a konci algoritmu je zaznamenám čas a provede se rozdíl koncového a počátečního času. Tím získáme čas, který zabralo zpracování algoritmu detekce kolizí, a zapíšeme tento čas do konzole s console.log() funkcí odkud tyto časy můžeme kopírovat a zpracovat do grafické podoby. Měření doby vykonávání funkcí je ve Javascriptu možné pomocí mnoha funkcí například new Date(), performance.now(), Date.now() nebo performance.mark(). Nakonec byla zvolena funkce performance.now() pro svou přesnost a jednoduché použití. Původně se data vypisovala po jednom a to hned poté, co se údaj získal. To ovšem bylo velice neefektivní. Vznikl tak problém data roztřídit do patřičných kategorií a pokud by se vypsala jedna kategorie na jedno spuštění, mohlo by dojít ke získání špatných dat. Konzole také špatně vypisuje stejné záznamy za sebou: místo aby vypsala několik stejných řádků za sebou, tak vypíše jeden řádek a číslo kolikrát se tento řádek opakuje. Navíc výpis během provádění kódu, který je měřen, způsobil zpoždění a tím i další chybu v měření. Proto se údaje sbírají do array polí a toto pole se vypíše najednou jakmile má v sobě 500 záznamů. Tentokrát se pole vypíše v části kódu, který se neměří. Výpis pole také za sebou píše stejné záznamy, čímž nedochází ke špatnému výpisu. Poté se všechna pole vynulují, znovu naplní daty a znovu vypíší jakmile mají opět 500 záznamů. Každá kategorie má samozřejmě vlastní array pole s poznámkou na konci, aby bylo jasné o které pole se jedná.

Testování probíhalo především praxí. Algoritmus se aplikoval do programu, poté se objekty daly do pohybu, tak aby byly v kolizi s ostatními. Aby bylo vidět, zda algoritmus zaznamenal kolizi, tak objekty mění barvu pokaždé, kdy podle algoritmu v kolizi jsou. Toto testování je nutné udělat pro objekty všeljakých různých vlastností - různé úhly, odlišné rozměry, které jsou velké, malé, rovnoměrné, různé, které mají odlišné tvary a odlišné úhly. Pozorováním pak zjistíme zda algoritmus odhalil kolizi včas, pozdě nebo příliš brzy.

5.1. Výsledky měření v aplikaci Kostka

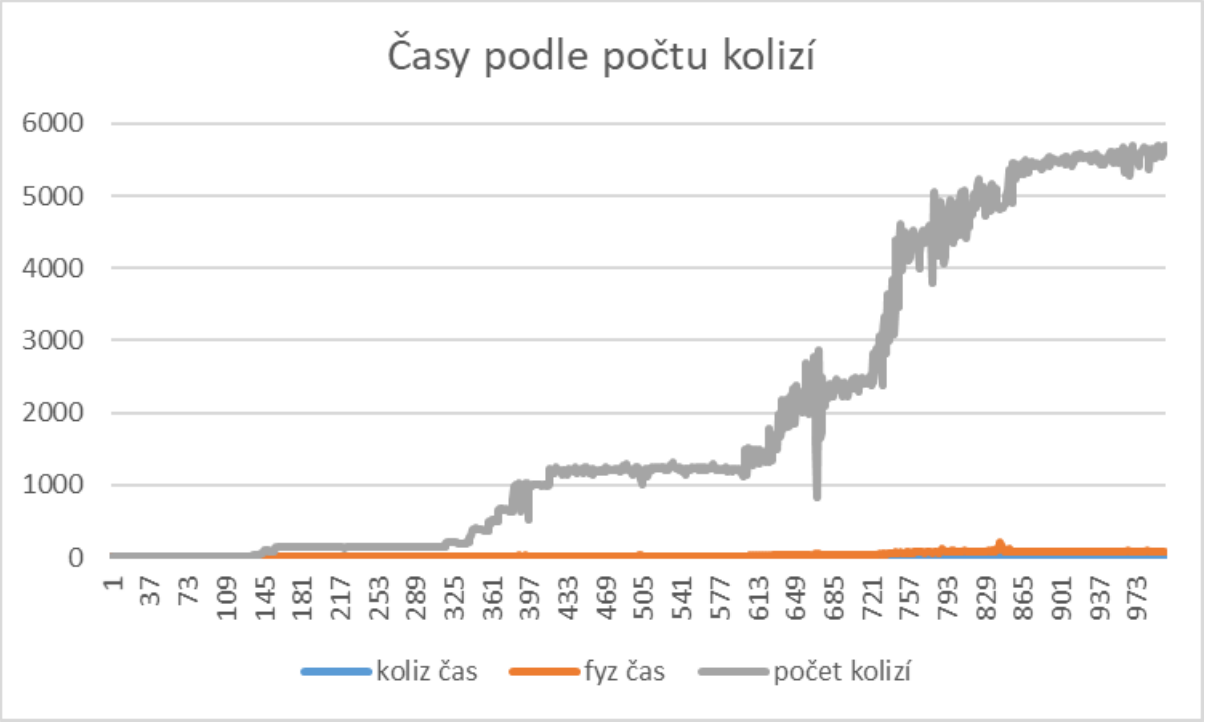
V následující části se nachází grafy vytvořené při testování v 1. testovací aplikaci zvané Kostka. V grafech se nachází čas výpočtu všech kolizí během jednoho výpočtového kroku, počet kolizí, který algoritmus zachytil a počet objektů, který se ve scéně zrovna nachází. Všechny časy jsou uvedeny v milisekundách. Změřen je i fyzikální engine Cannon.js, který má změřen i čas výpočtu času pro fyziku. Na konci sekce se nachází srovnání, jak dlouho kterému algoritmu trvalo průměrně vypočítat jednu kolizi.

Cannon.js



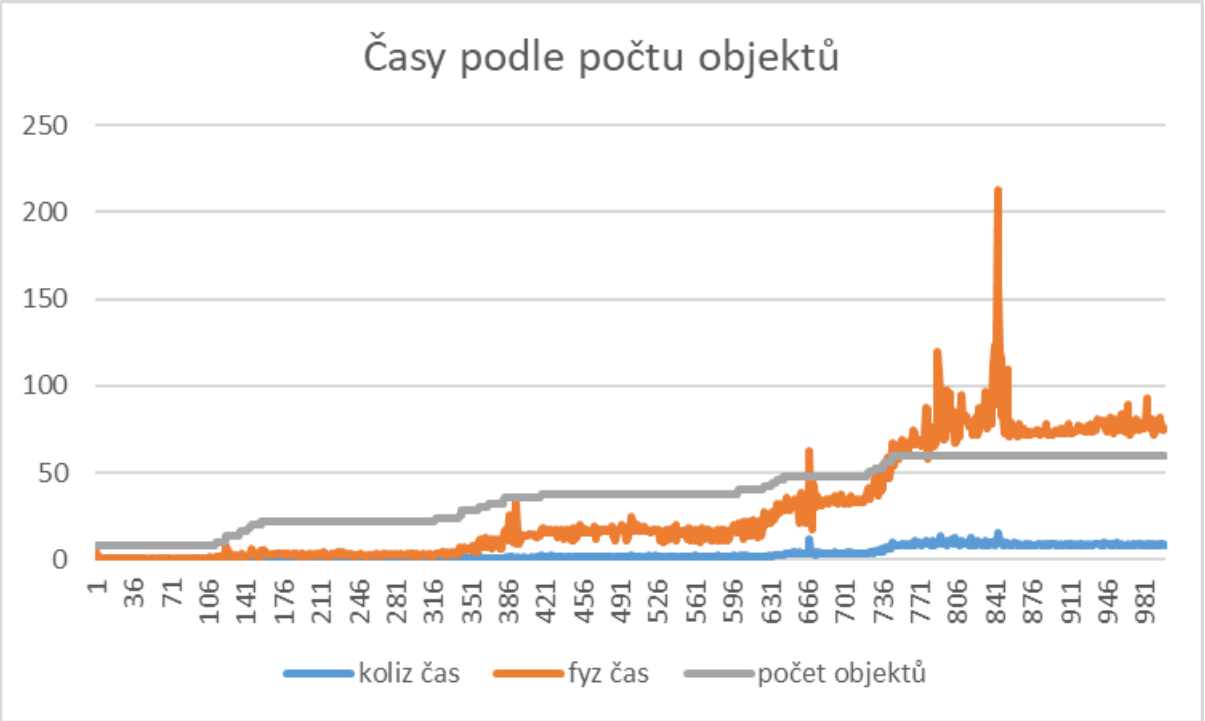
Graf 5.1.1: graf průběhu výpočtu časů

Tento graf znázorňuje jak dlouho trvá Cannonu.js vypočítat kolize, fyziku a oboje dohromady během jednoho výpočtového kroku. Zajímavé je, že kolizní čas se nevychyluje tak moc jako fyzikální čas.



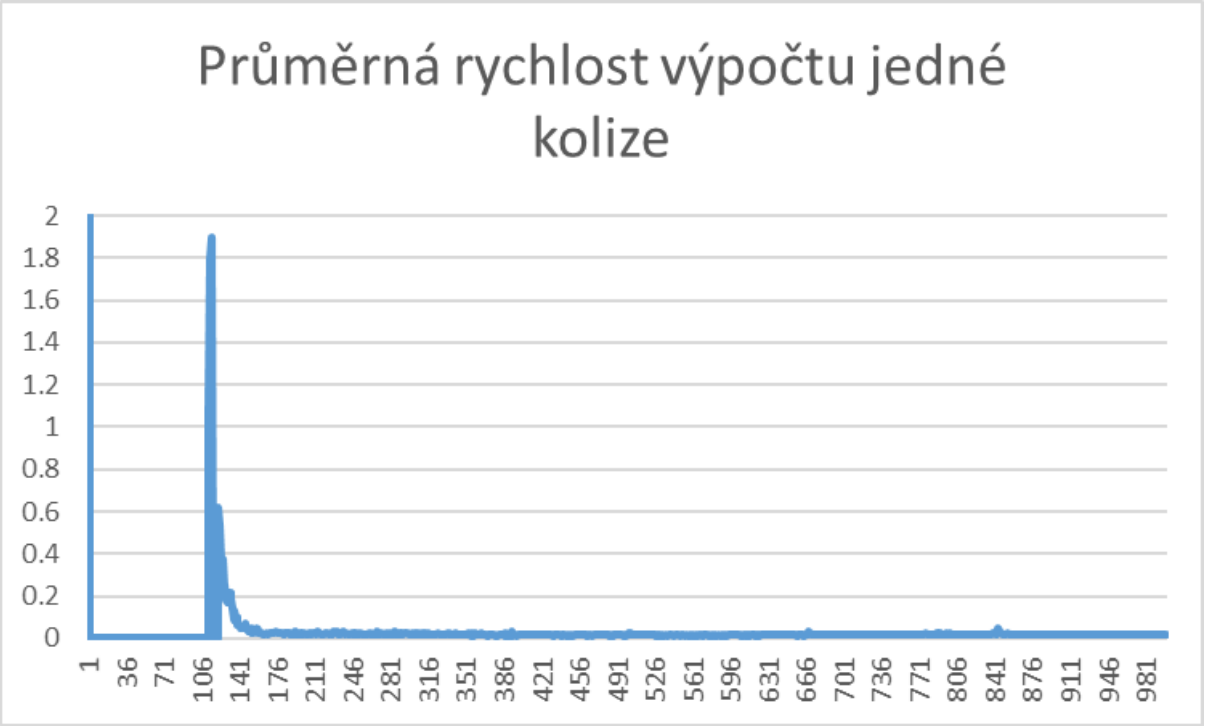
Graf 5.1.2: graf průběhu výpočtu času podle kolizí

Tento graf znázorňuje jak dlouho trvá Cannonu.js vypočítat kolize a fyziku v závislosti na tom, kolik zrovna probíhá kolizí během jednoho výpočtového kroku. Fyzikální čas reaguje na výchylky v počtu kolizí mnohem citlivěji než kolizní čas.



Graf 5.1.3: graf průběhu výpočtu času podle počtu objektů

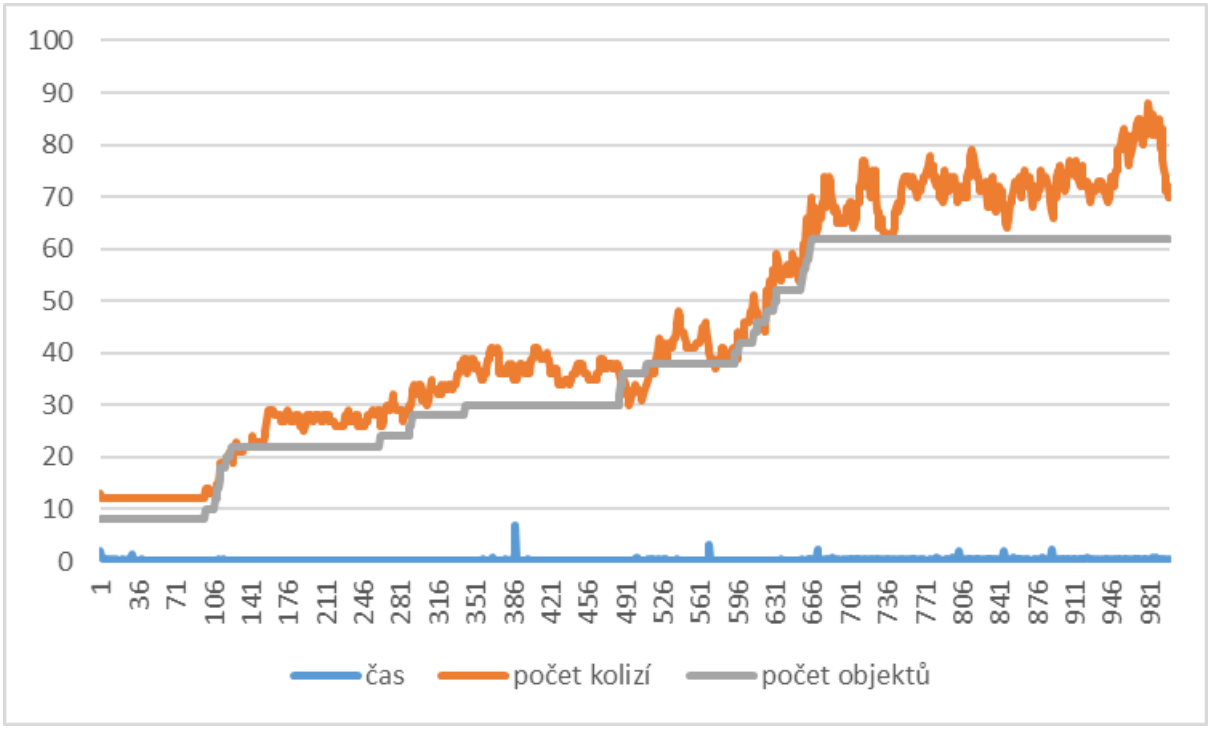
Tento graf znázorňuje jak dlouho trvá Cannonu.js vypočítat kolize a fyziku v závislosti na tom, kolik se zrovna ve scéně nachází objektů a to během jednoho výpočtového kroku. Fyzikální čas reaguje na výchylky v počtu kolizí mnohem citlivěji než kolizní čas.



Graf 5.1.4: graf průměrné rychlosti výpočtu jedné kolize

Až na několik počátečních výkyvů, dává graf stálá data.

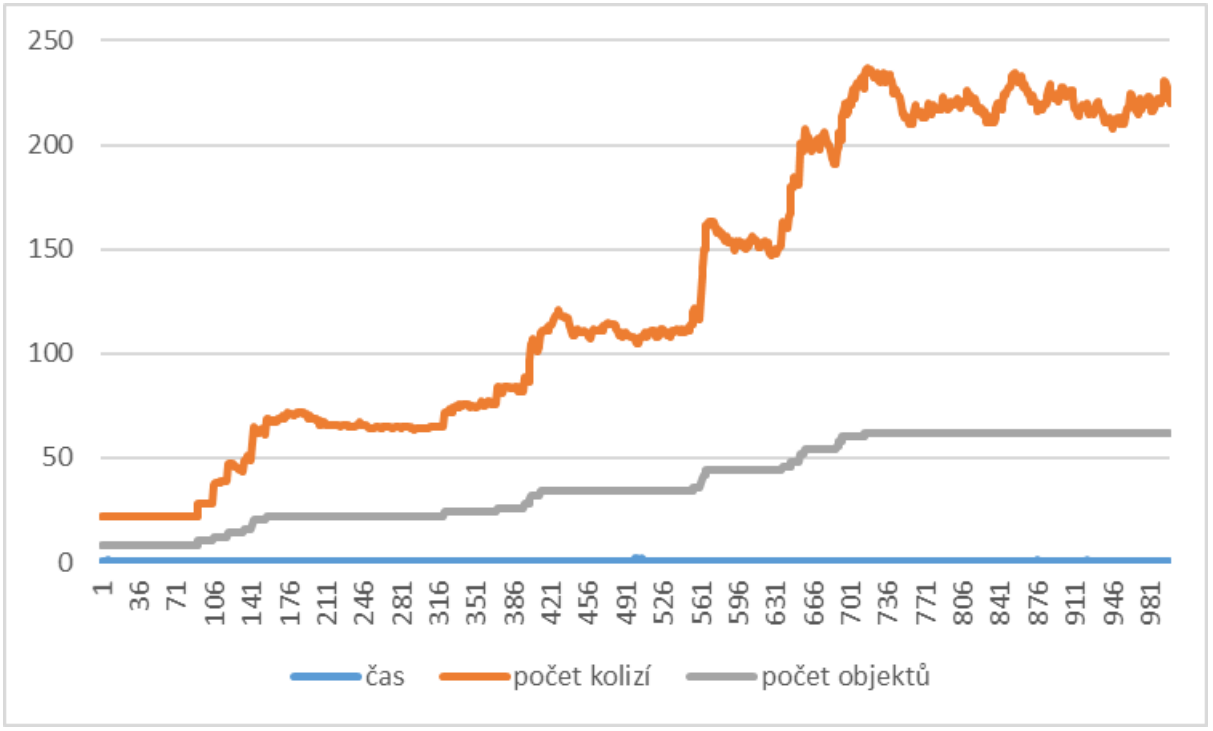
1. algoritmus, algoritmus s vestavěnou funkcí



Graf 5.1.5: graf rychlosti výpočtu kolizí podle počtu kolizí a počtu objektů 1. algoritmu

Algoritmus na změnu počtu kolizí a objektů, téměř vůbec nereaguje a zachovává si stabilní a nízký čas výpočtu po celou dobu.

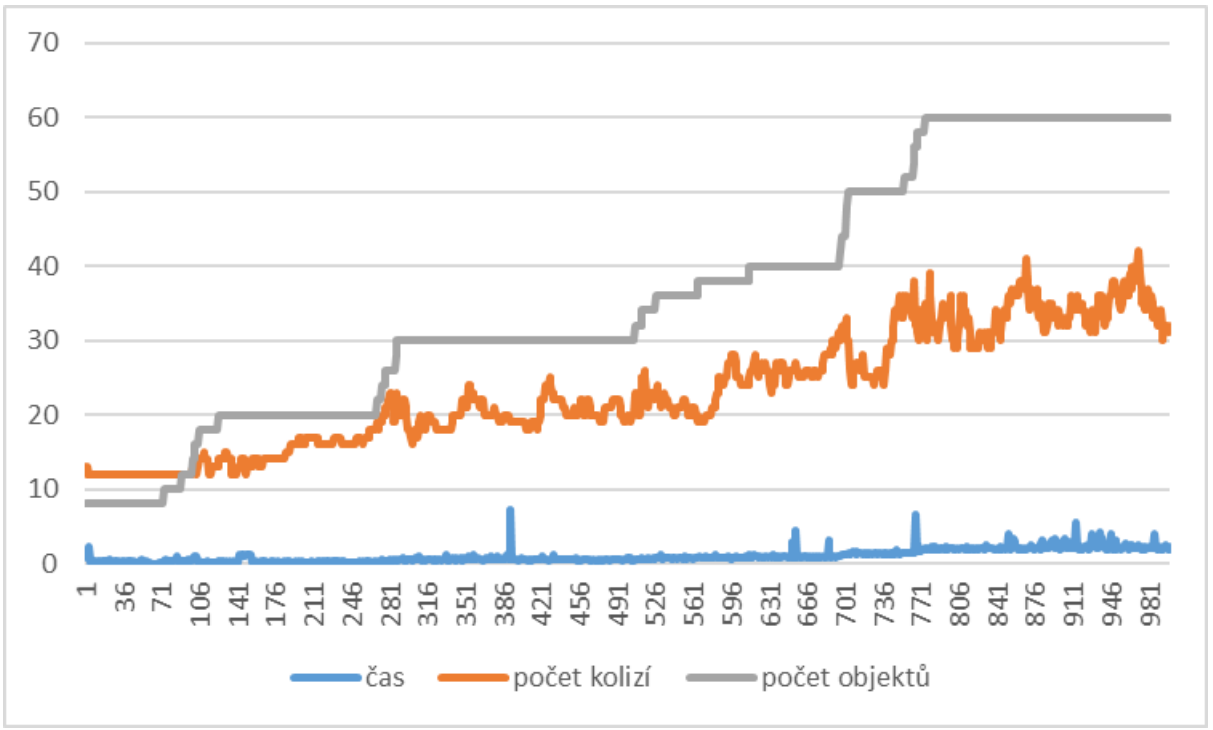
2. algoritmus, algoritmus s největším rozměrem



Graf 5.1.6: graf rychlosti výpočtu kolizí podle počtu kolizí a počtu objektů 2. algoritmu

Algoritmus na změnu počtu kolizí a objektů, téměř vůbec nereaguje a zachovává si stabilní a nízký čas výpočtu po celou dobu. Zaznamenává ovšem poměrně mnoho kolizí.

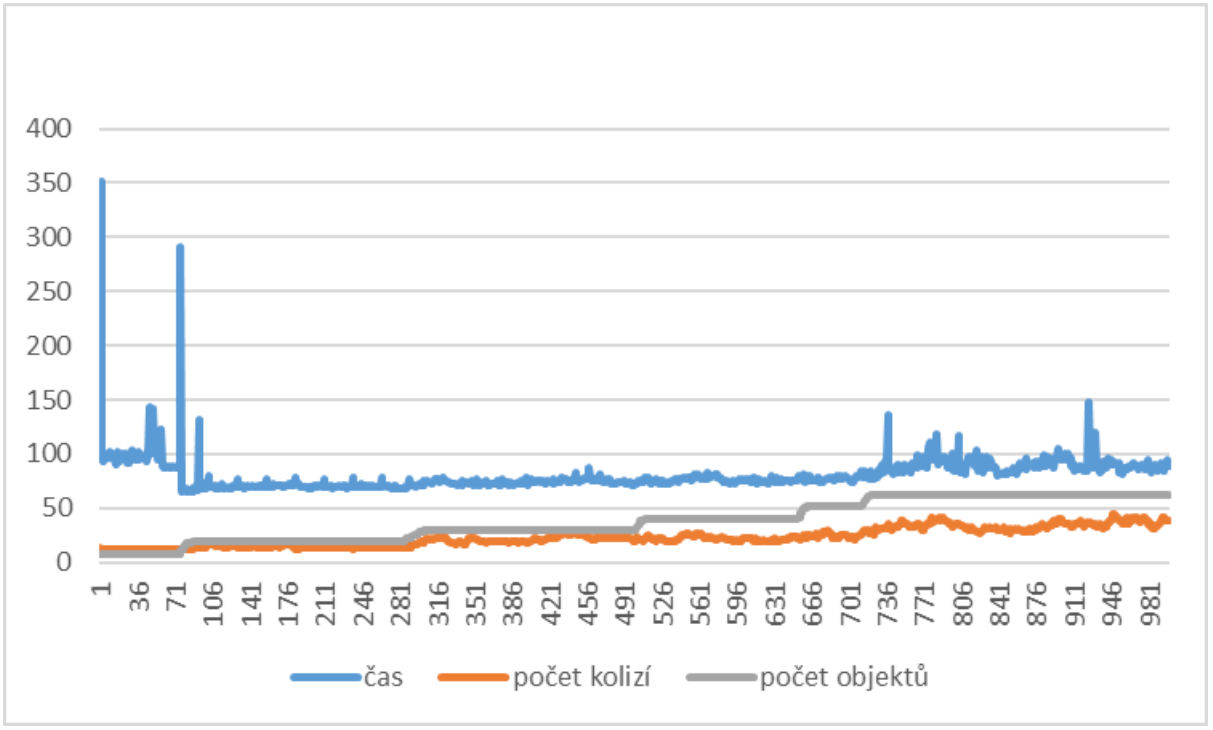
3. algoritmus, algoritmus s AABB



Graf 5.1.7: graf rychlosti výpočtu kolizí podle počtu kolizí a počtu objektů 3. algoritmu

Při vysokém počtu objektů už je vidět zvýšení času výpočtu. Křivka záznamu kolizí není příliš podobná jako křivka zvýšení počtu objektů. Jedná se o první algoritmus, který zaznamenává méně kolizí než je objektů.

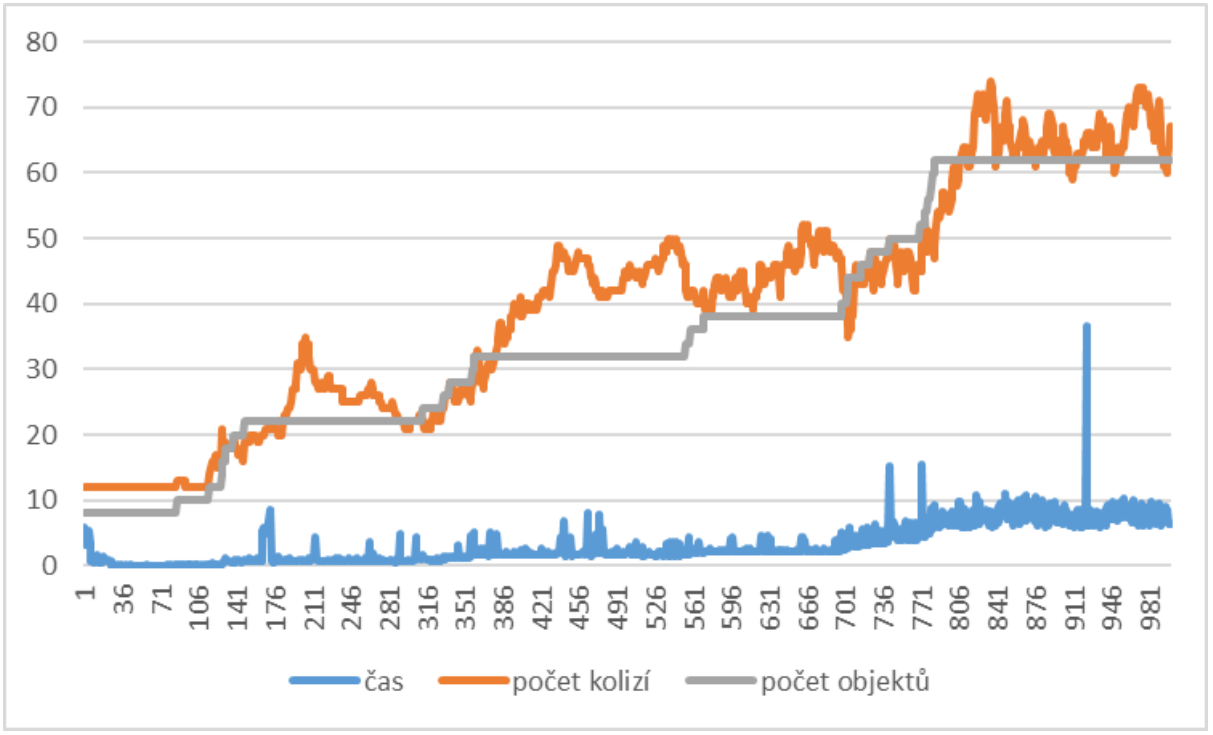
4. algoritmus, algoritmus s FaceT a počítadlem vzdálenosti



Graf 5.1.8: graf rychlosti výpočtu kolizí podle počtu kolizí a počtu objektů 4. algoritmu

Čas výpočtu je velice vysoký a příliš se nemění se změnou počtu objektů. Křivka počtu kolizí je menší než křivka počtu objektů zato dobře napodobuje její tvar.

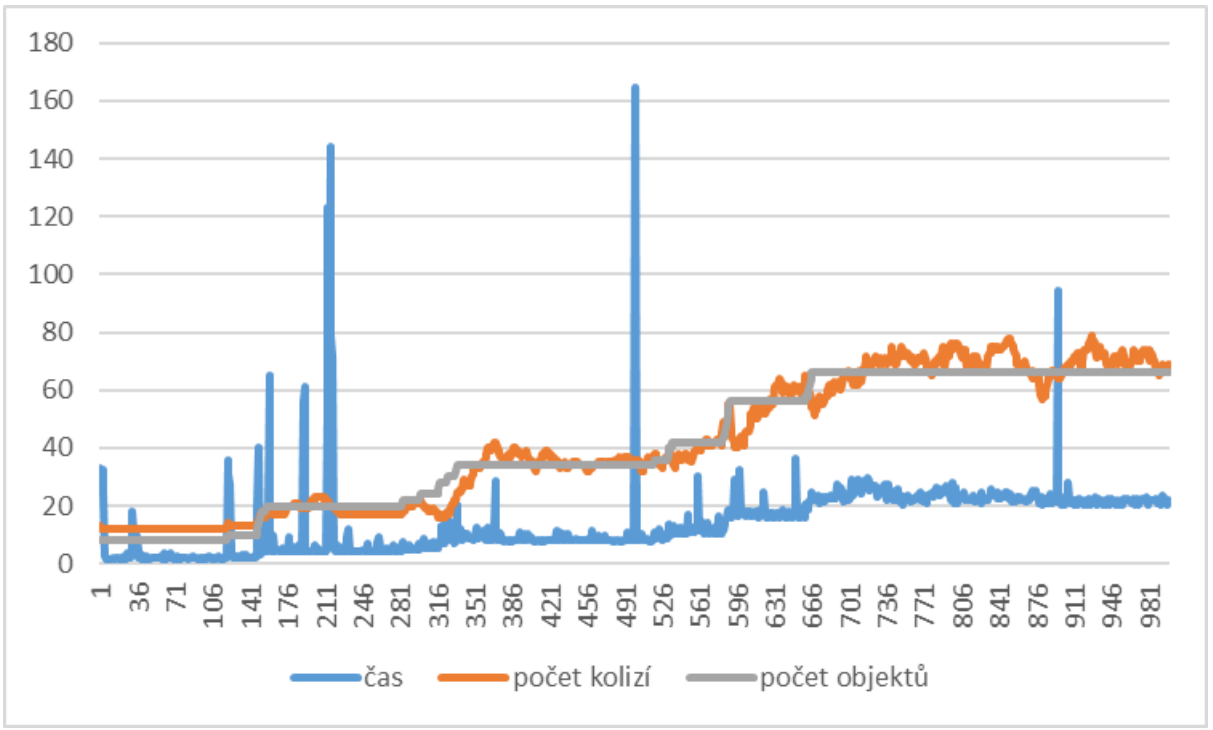
5. algoritmus, algoritmus s OBB



Graf 5.1.9: graf rychlosti výpočtu kolizí podle počtu kolizí a počtu objektů 5. algoritmu

Čas výpočtu je o velmi mírně vyšší, ovšem výrazné zvýšení přichází až s velkým počtem objektů. Křivka počtu objektů a křivka počtu kolizí jsou velmi podobné jako u 1. algoritmu.

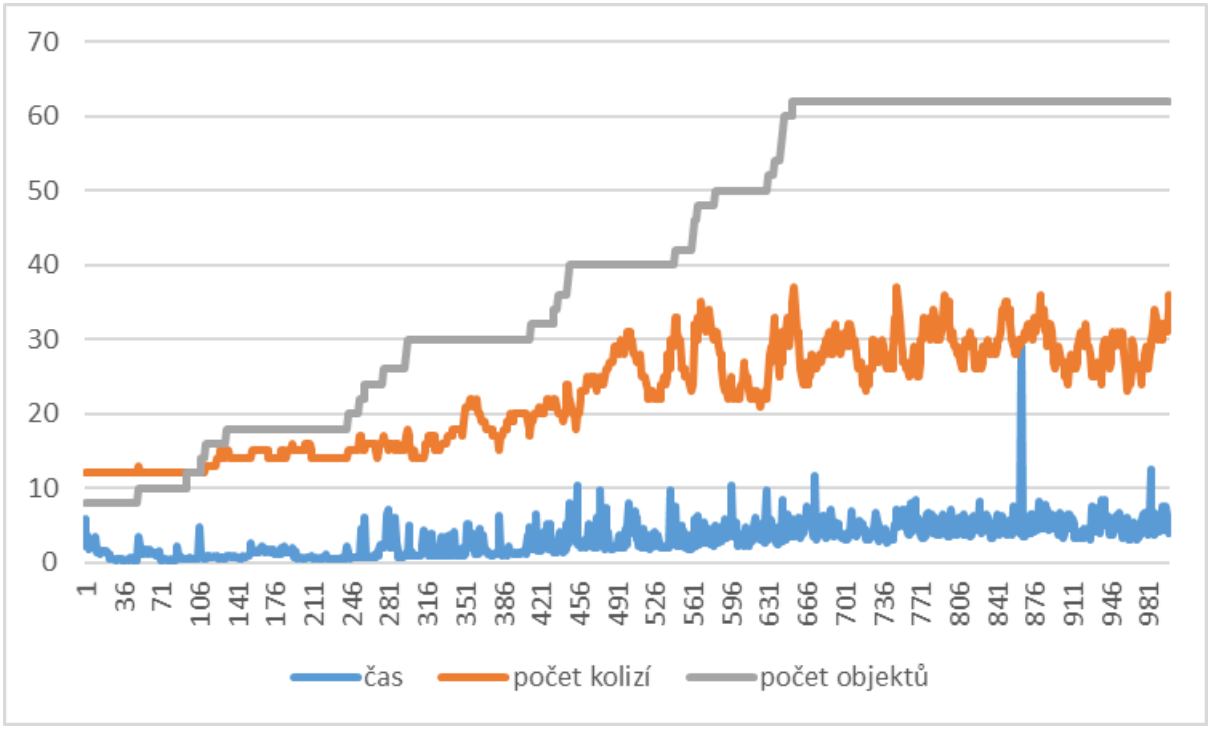
6. algoritmus, algoritmus s FaceT a vektory



Graf 5.1.10: graf rychlosti výpočtu kolizí podle počtu kolizí a počtu objektů 6. algoritmu

Křivka počtu objektů a křivka počtu kolizí jsou velmi podobné jako u 1. algoritmu a 5. algoritmu. Ovšem čas výpočtu je o něco vyšší a obsahuje podivné vysoké výkyvy.

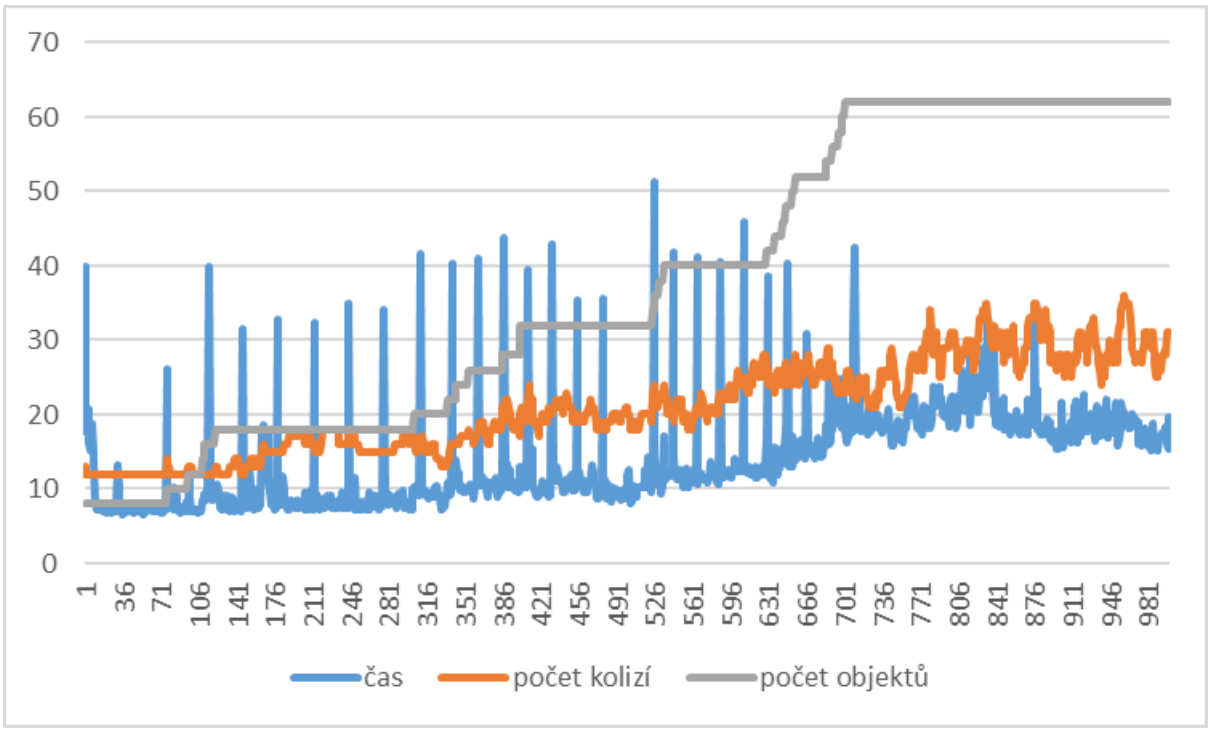
7. algoritmus, algoritmus s OBB a per triangle



Graf 5.1.11: graf rychlosti výpočtu kolizí podle počtu kolizí a počtu objektů 7. algoritmu

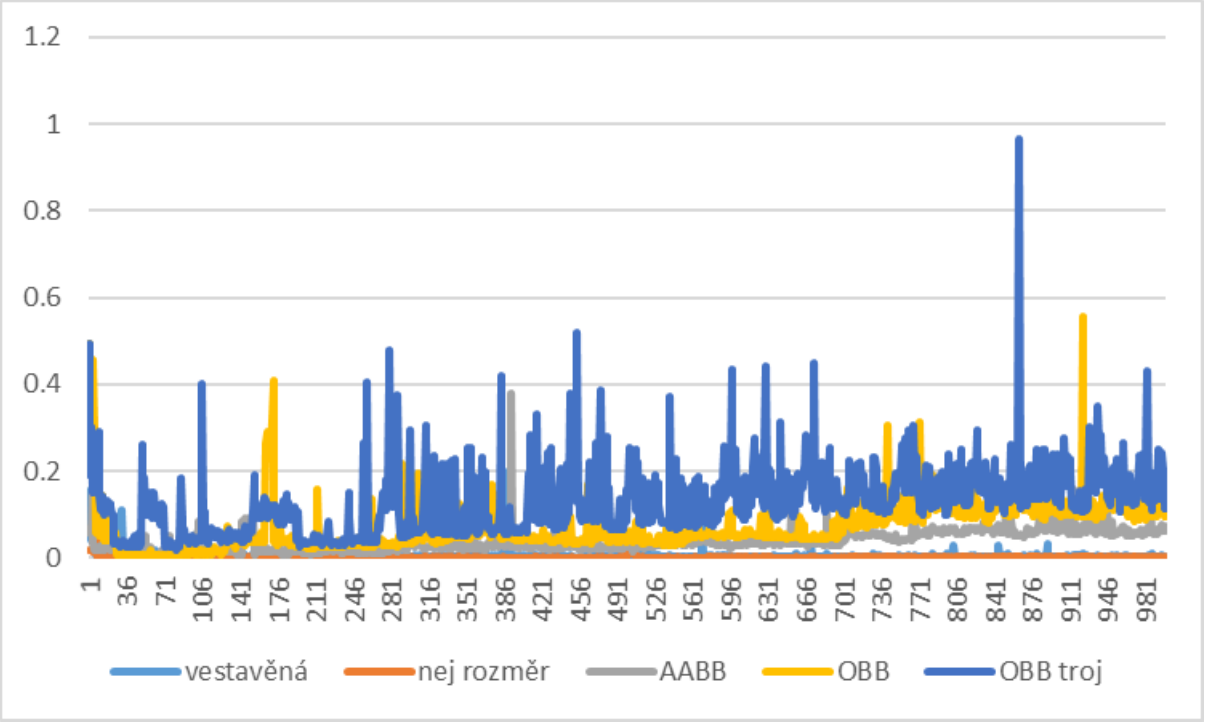
Čas výpočtu je velice kolísavý a téměř vůbec nereaguje na počet objektů. Křivka počtu kolizí je také velmi kolísavá. Křivky počtu objektů a počtu kolizí si nejsou podobné.

8. algoritmus, algoritmus s FaceT a per triangle

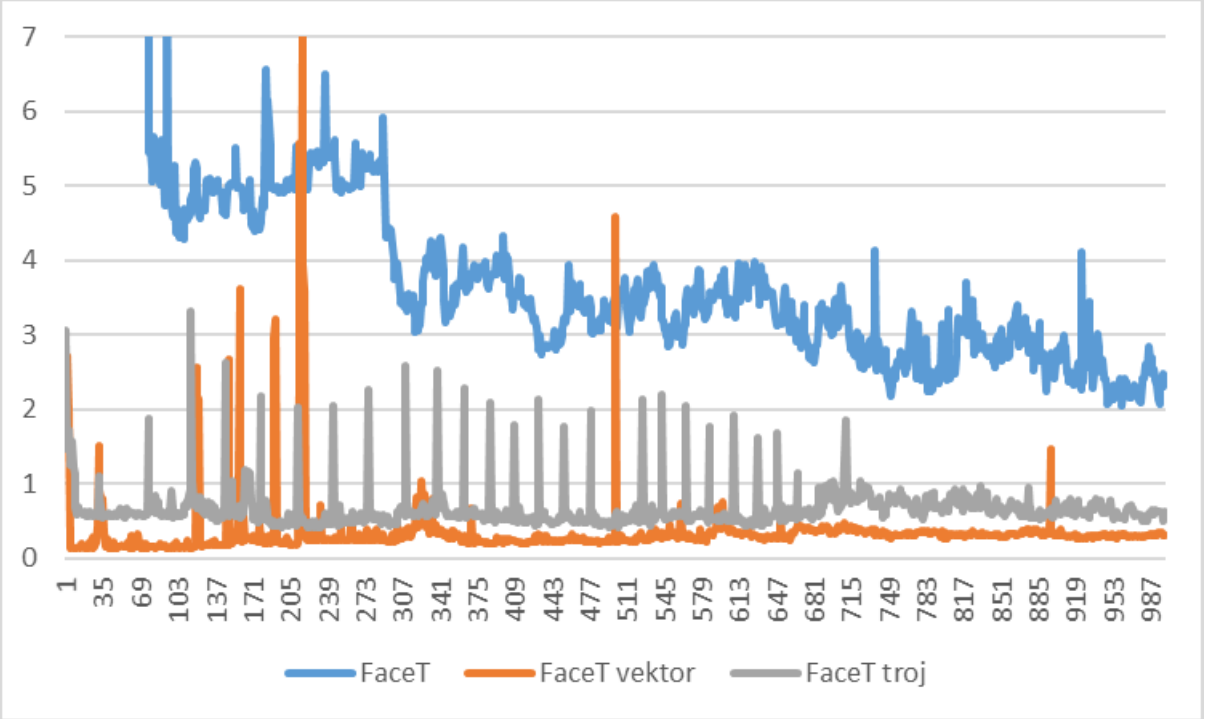


Graf 5.1.12: graf rychlosti výpočtu kolizí podle počtu kolizí a počtu objektů 8. algoritmu

Čas výpočtu obsahuje veliké výkyvy a samotný čas je velice vysoký. Křivka počtu kolizí velmi mírně napodobuje tvar počtu objektů.



Graf 5.1.13: graf rychlosti výpočtu jedné kolize rychlých algoritmů



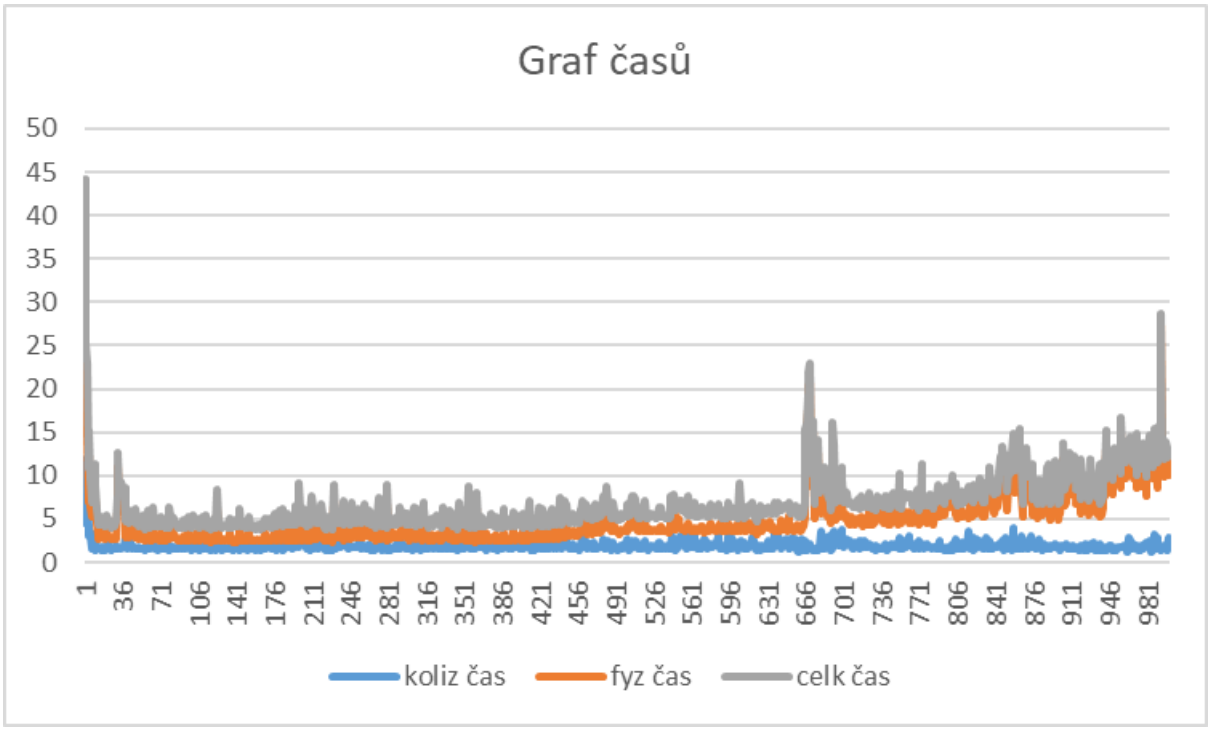
Graf 5.1.14: graf rychlosti výpočtu jedné kolize pomalých algoritmů

Všechny algoritmy využívající FaceT mají nejhorší časy a největší výkyvy. Ostatní algoritmy mají mezi sebou malé rozdíly, ale je vidět, že 1. algoritmus (vestavěná) a 2. algoritmus (nej rozměr) si vedou nejlépe.

5.2. Výsledky měření v aplikaci Padání

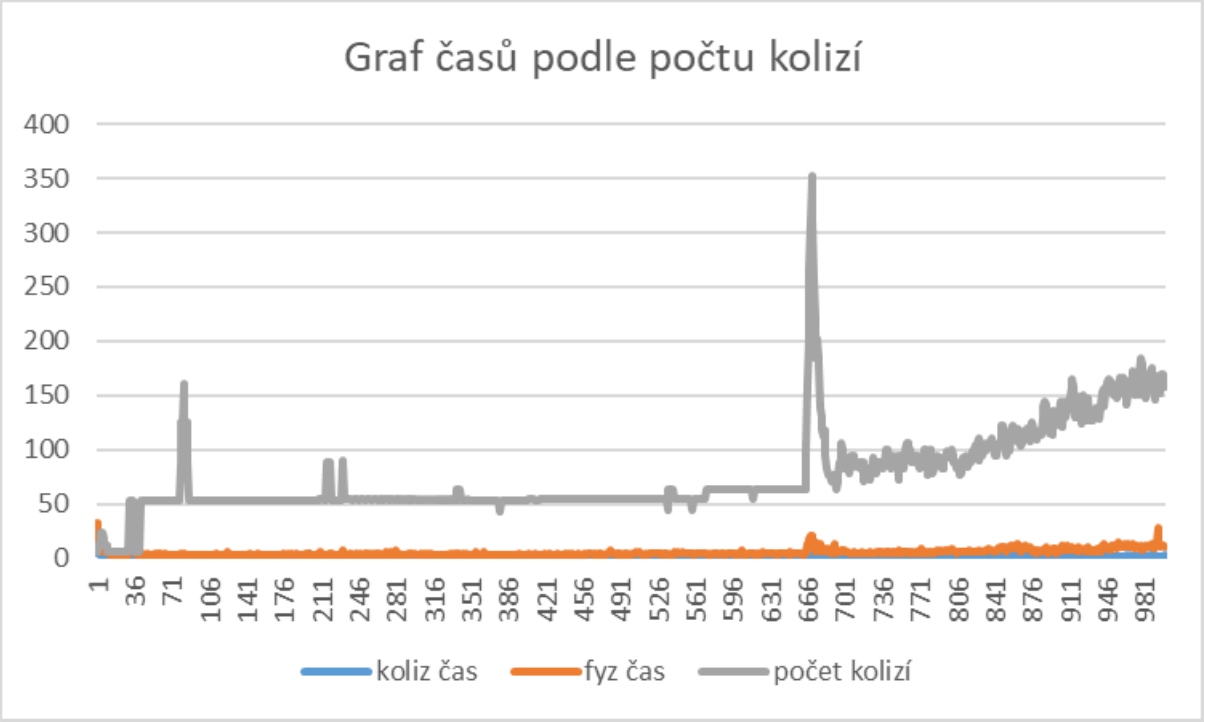
V následující části se nachází grafy vytvořené při testování v 2. testovací aplikaci zvané Padání. V grafech se nachází čas výpočtu všech kolizí během jednoho výpočtového kroku a počet kolizí, který algoritmus zachytil. Všechny časy jsou uvedeny v milisekundách. Změřen je i fyzikální engine Cannon.js, který má změřen i čas výpočtu času pro fyziku. Na konci sekce se nachází srovnání, jak dlouho kterému algoritmu trvalo průměrně vypočítat jednu kolizi.

Cannon.js



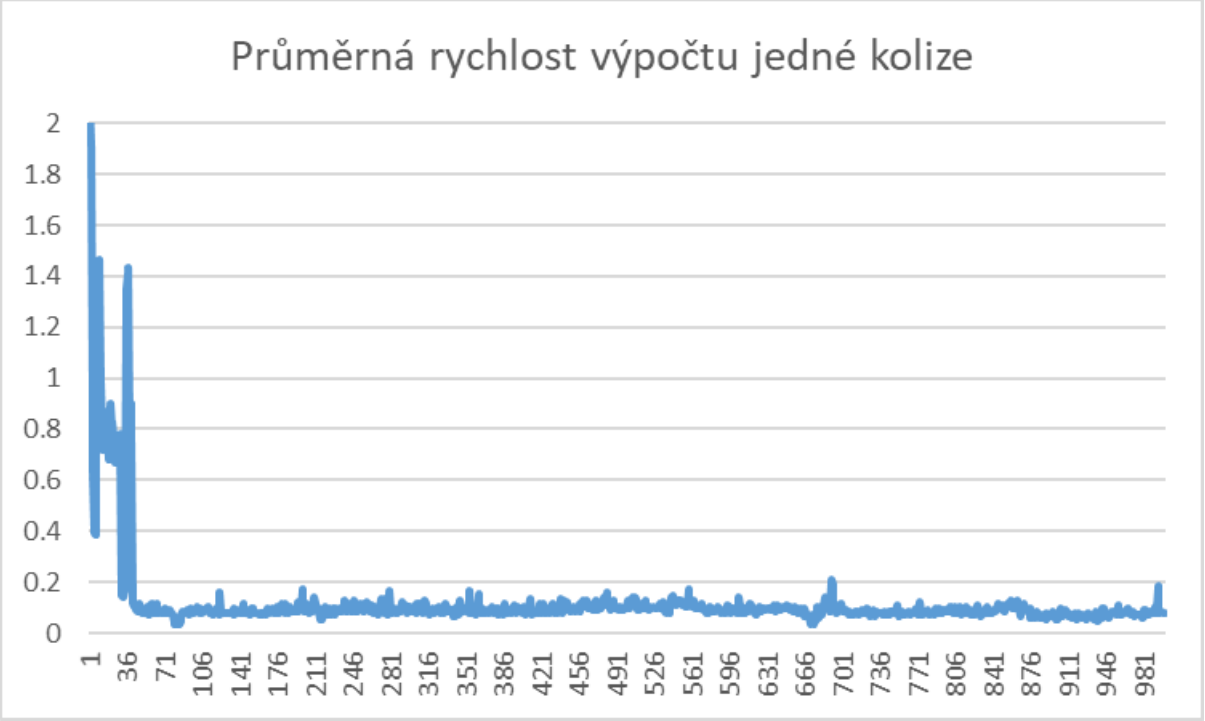
Graf 5.2.1: graf průběhu výpočtu časů

Tento graf znázorňuje jak dlouho trvá Cannonu.js vypočítat kolize, fyziku a oboje dohromady během jednoho výpočtového kroku. Je vidět, že kolizní čas se opět nevychyluje tak moc jako fyzikální čas.



Graf 5.2.2: průběhu výpočtu času podle kolizí

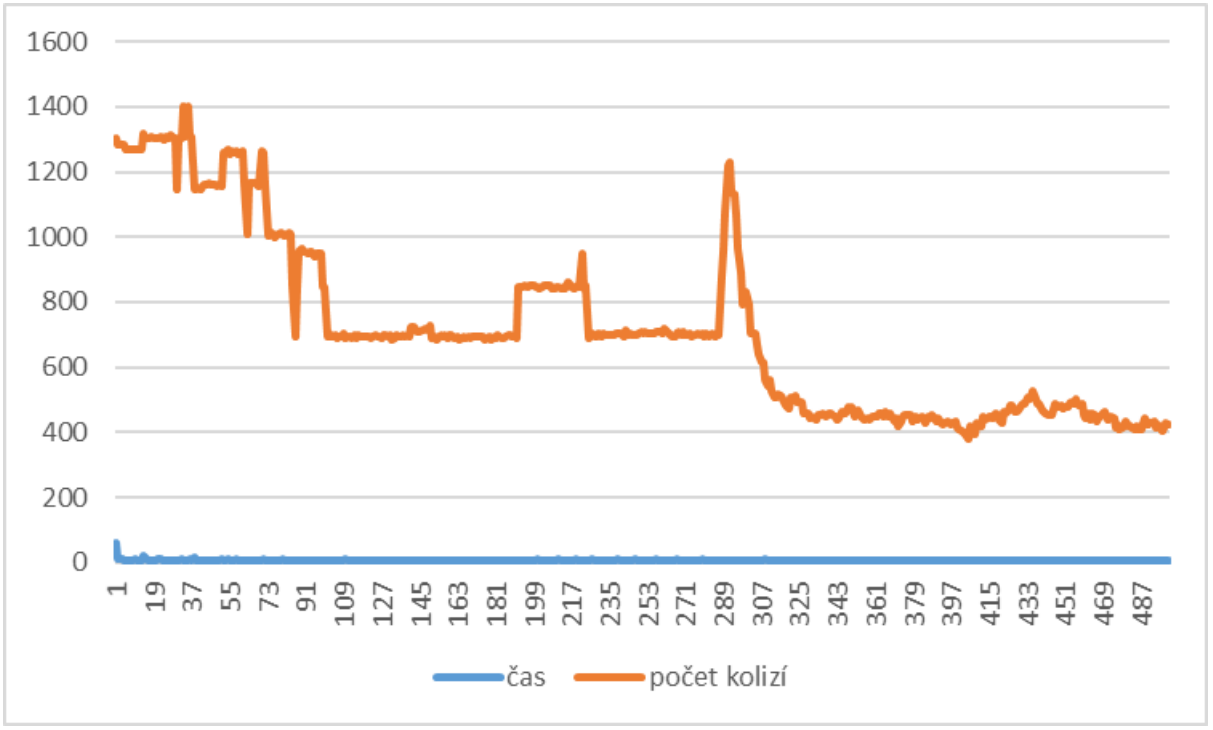
Tento graf znázorňuje jak dlouho trvá Cannonu.js vypočítat kolize a fyziku v závislosti na tom, kolik zrovna probíhá kolizí během jednoho výpočtového kroku. Fyzikální čas opět reaguje na výchylky v počtu kolizí mnohem citlivěji než kolizní čas.



Graf 5.2.3: průměrné rychlosti výpočtu jedné kolize

Výkyvy na začátku již nejsou tak velké jako u 1.aplikace, Kostka. Poté graf dává stálá data, která jsou ovšem vyšší než v 1.aplikaci.

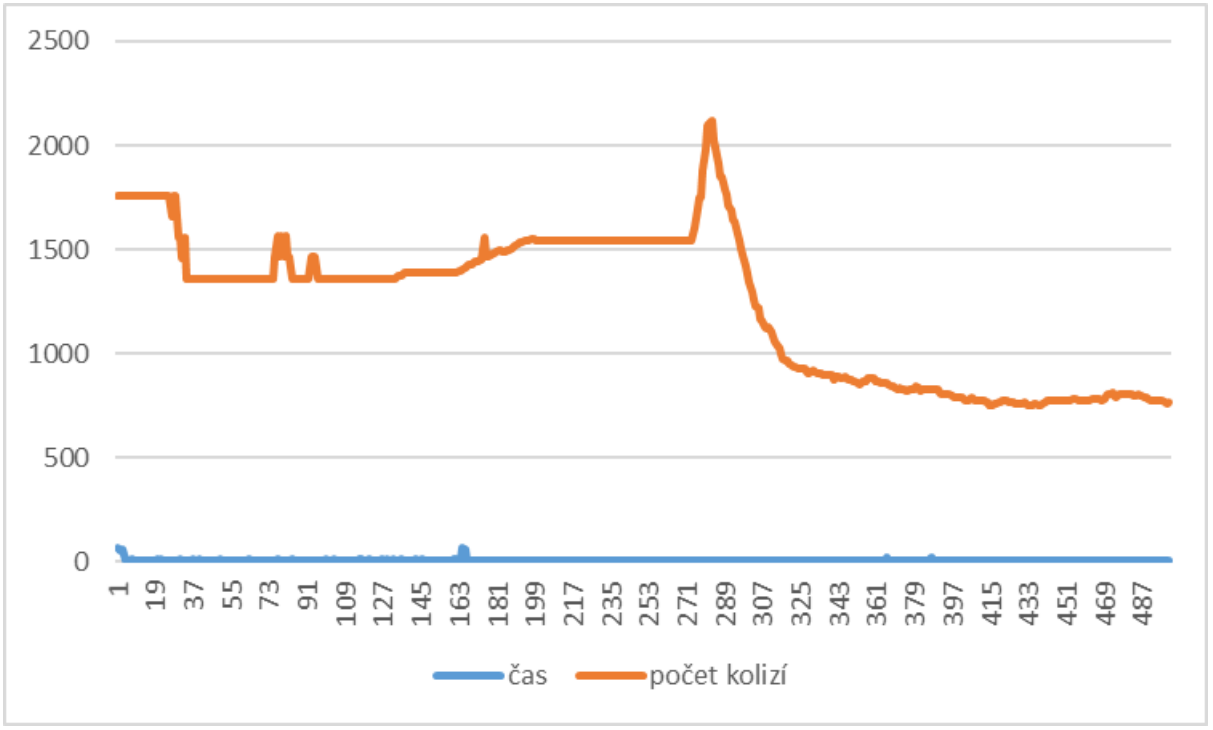
1. algoritmus, algoritmus s vestavěnou funkcí



Graf 5.2.4: rychlosti výpočtu kolizí podle počtu kolizí 1. algoritmu

Čas výpočtu je stále velmi nízký a velmi stálý po celou dobu.

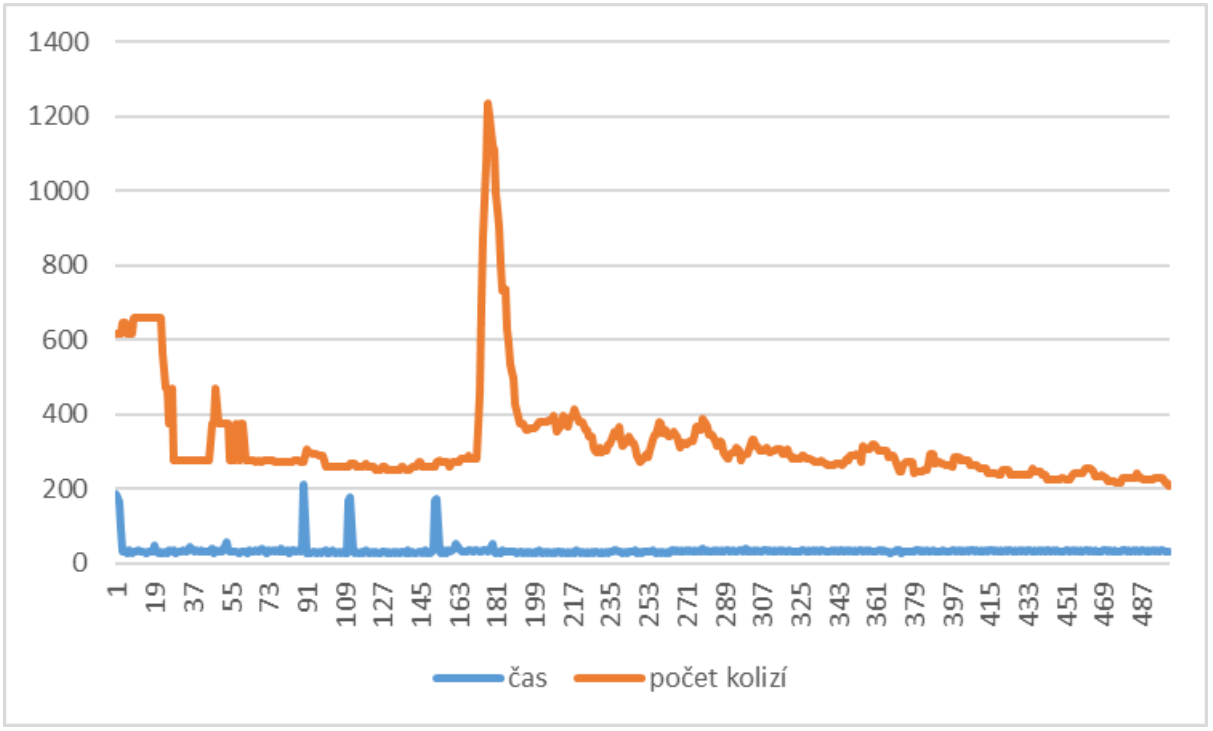
2. algoritmus, algoritmus s největším rozměrem



Graf 5.2.5: rychlosti výpočtu kolizí podle počtu kolizí 2. algoritmu

I zde je čas výpočtu stále velmi nízký a velmi stálý po celou dobu, ale s vyšším počtem kolizí.

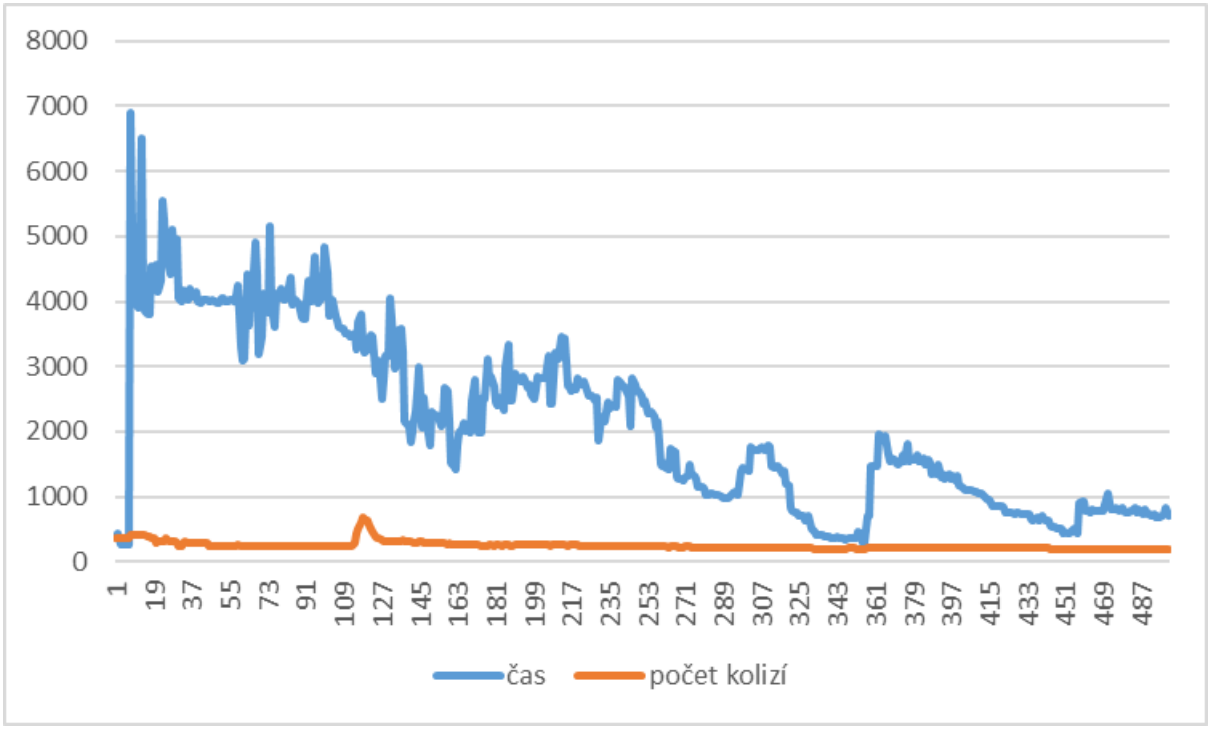
3. algoritmus, algoritmus s AABB



Graf 5.2.6: rychlosti výpočtu kolizí podle počtu kolizí 3. algoritmu

I zde je čas výpočtu stále velmi nízký s několika výkyvy, ale s menším počtem kolizí a s odlišnou křivkou počtu kolizí.

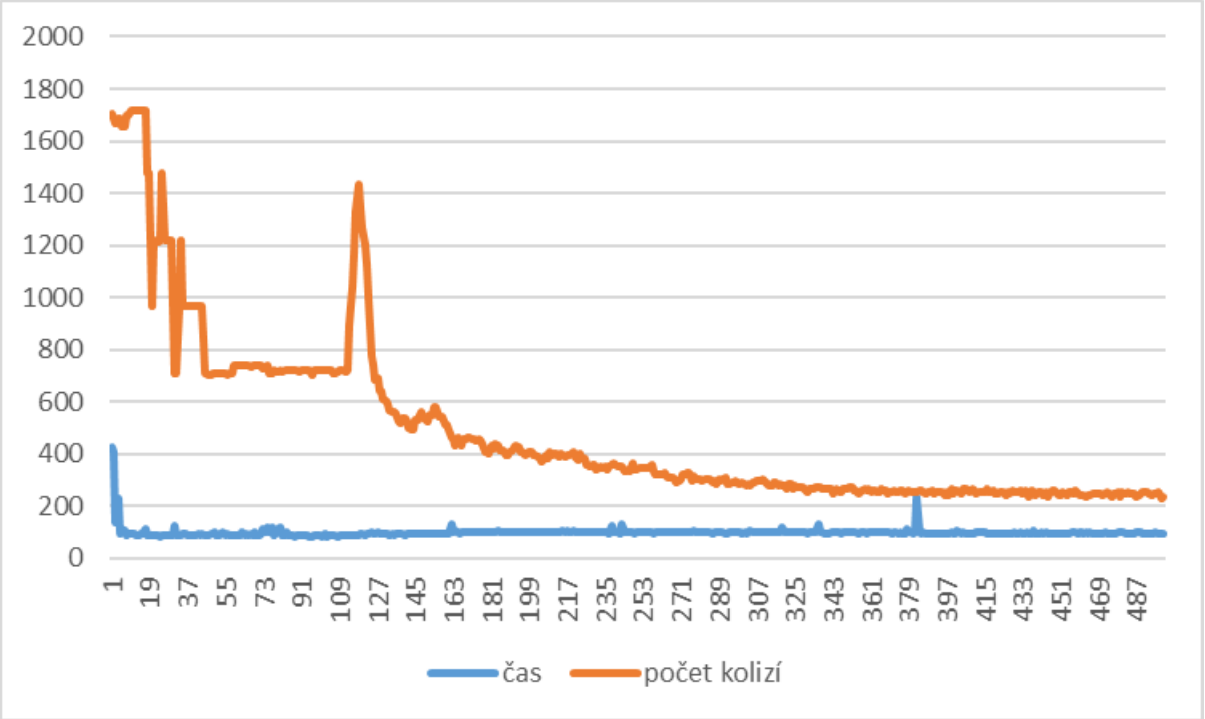
4. algoritmus, algoritmus s FaceT a počítadlem vzdálenosti



Graf 5.2.7: rychlosti výpočtu kolizí podle počtu kolizí 4. algoritmu

Čas je extrémně vysoký a naprosto nestálý. Křivka času vůbec nemění svůj tvar podle křivky počtu kolizí.

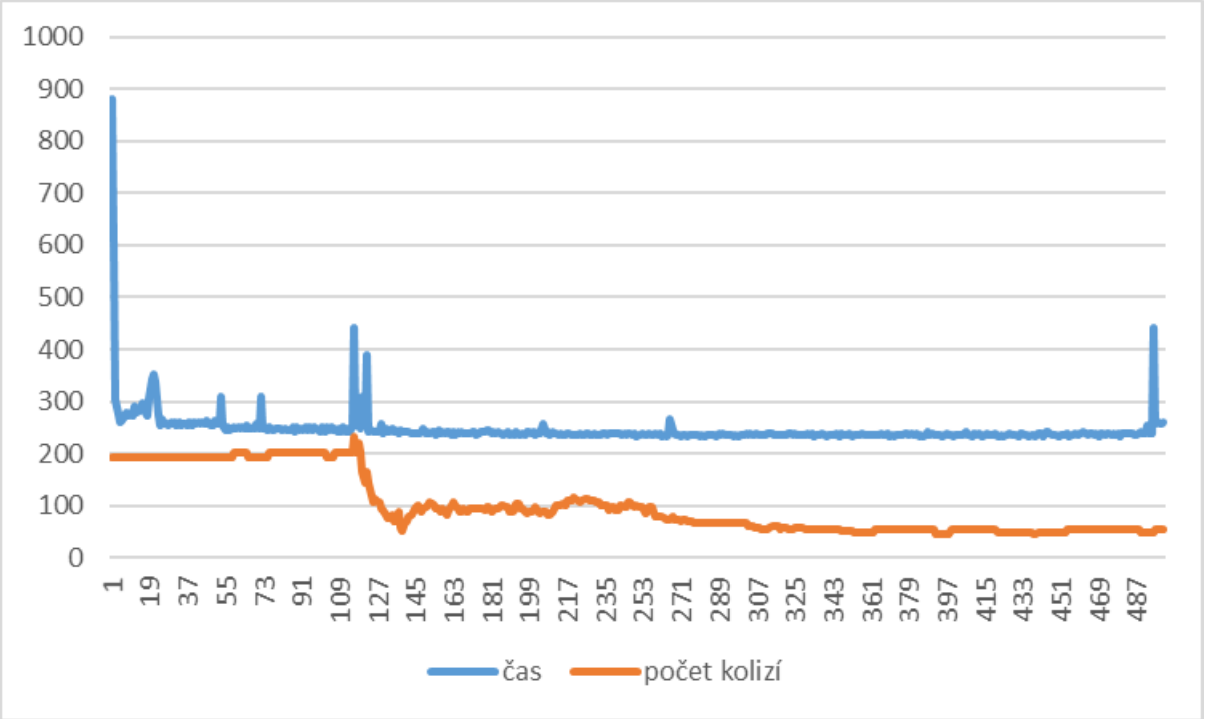
5. algoritmus, algoritmus s OBB



Graf 5.2.8: rychlosti výpočtu kolizí podle počtu kolizí 5. algoritmu

Křivka počtu kolizí padá ve srovnání s ostatními algoritmy rychleji. Čas mírně reaguje na změnu počtu kolizí. Čas výpočtu je stálý a nízký, ovšem vyšší než u některých algoritmů.

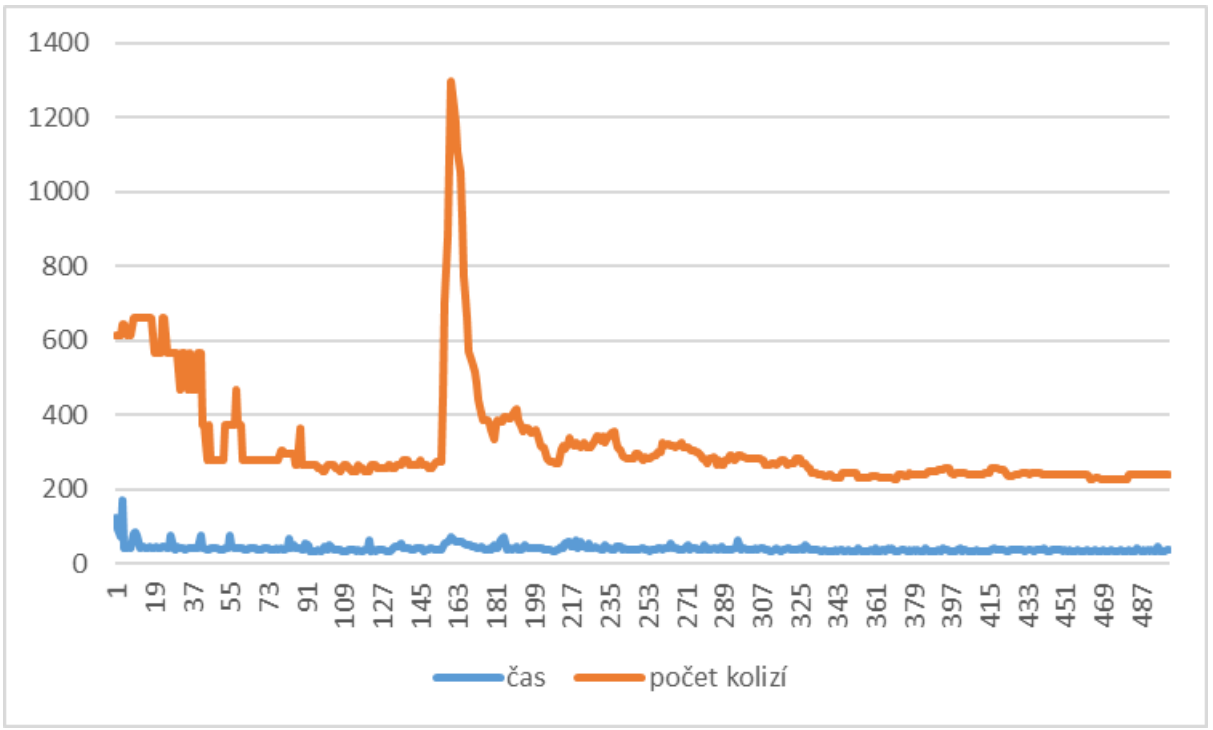
6. algoritmus, algoritmus s FaceT a vektory



Graf 5.2.9: rychlosti výpočtu kolizí podle počtu kolizí 6. algoritmu

Čas výpočtu je stálý, ale také vysoký. Na křivce počtu kolizí není žádné výrazné stoupání.

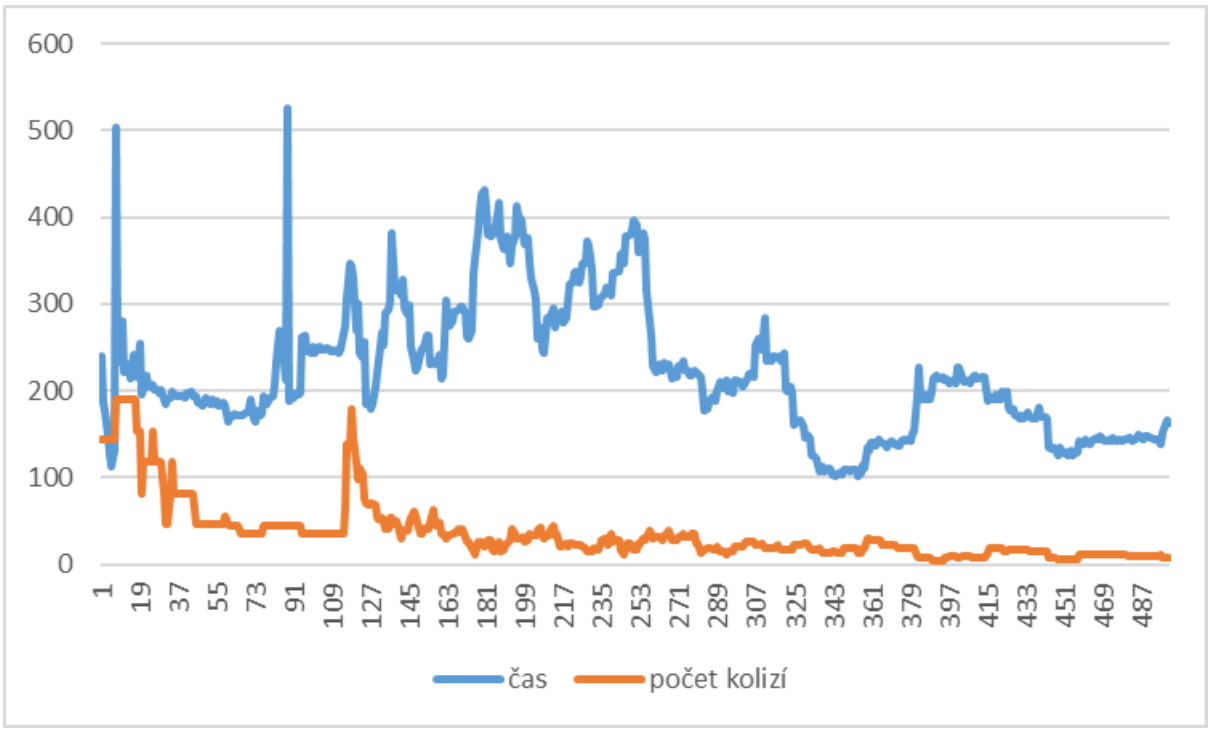
7. algoritmus, algoritmus s OBB a per triangle



Graf 5.2.10: rychlosti výpočtu kolizí podle počtu kolizí 7. algoritmu

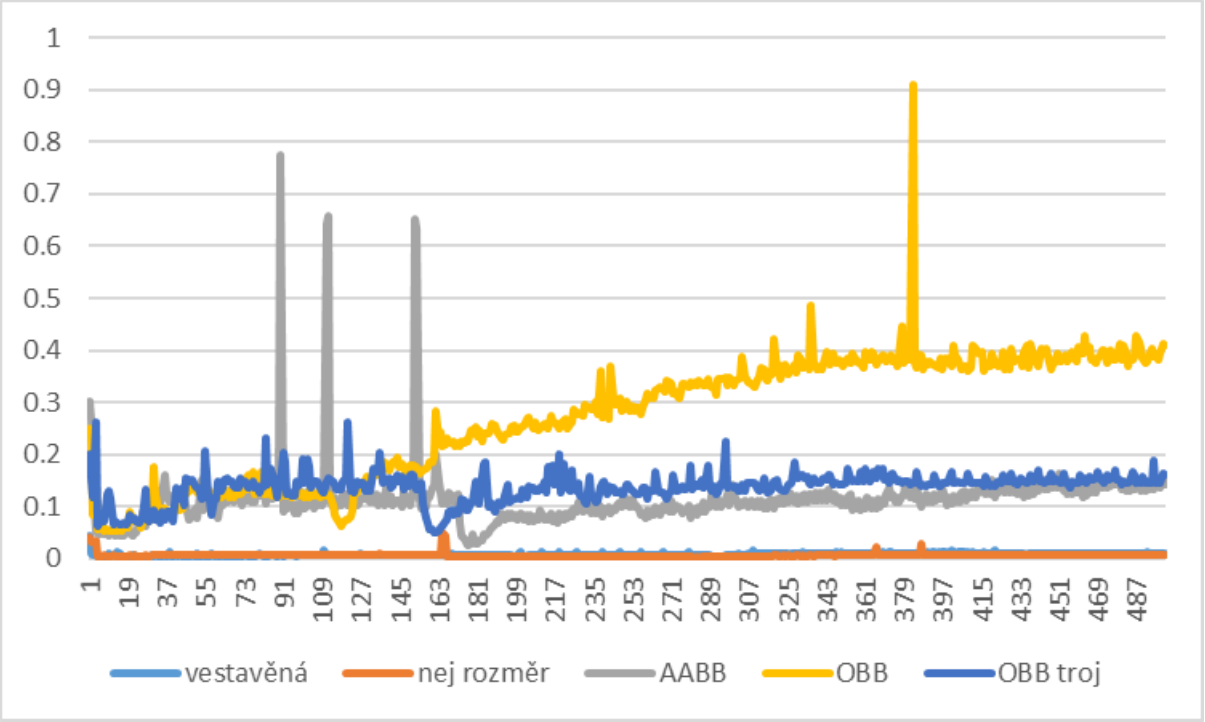
Čas výpočtu je nízký a stálý. Křivka počtu kolizí je spíše standardní.

8. algoritmus, algoritmus s FaceT a per triangle

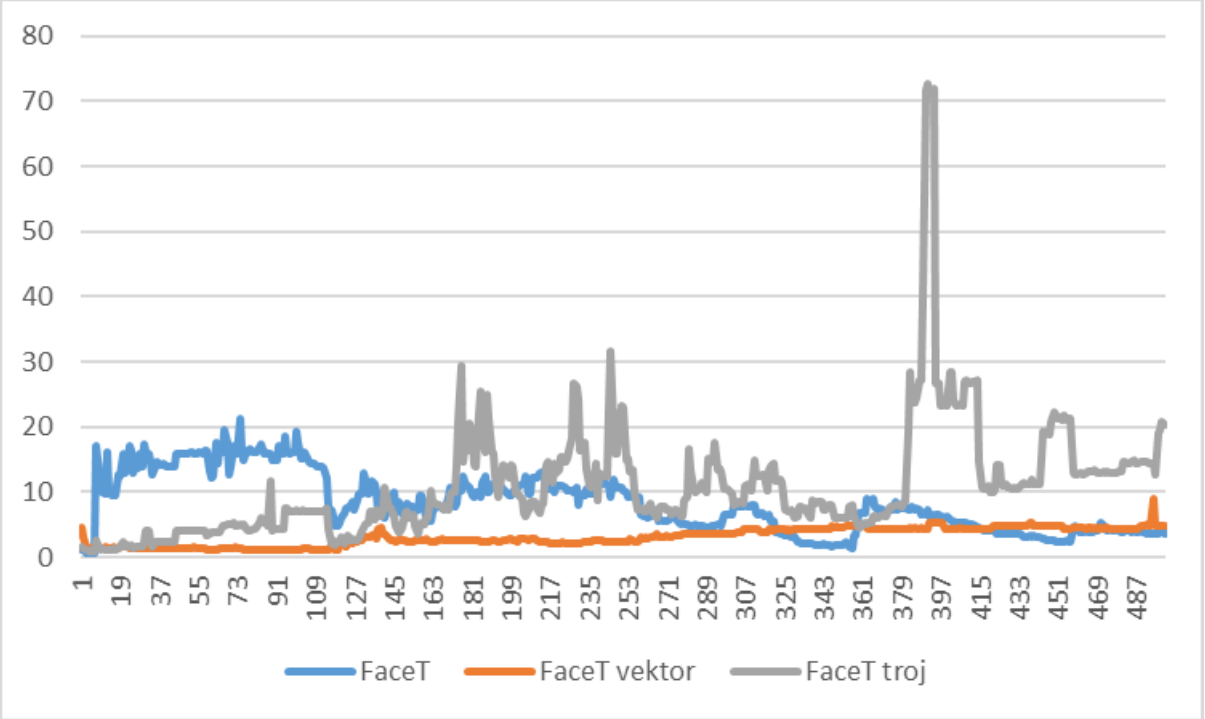


Graf 5.2.11: rychlosti výpočtu kolizí podle počtu kolizí 8. algoritmu

Čas výpočtu je vysoký a nemá žádný řád ani postup. Na jejím stoupání a klesání není vidět žádný vztah s křivkou počtu kolizí. Křivka počtu kolizí je oproti tomu podobná jako u předchozích algoritmů.



Graf 5.2.12: rychlosti výpočtu jedné kolize rychlých algoritmů



Graf 5.2.13: rychlosti výpočtu jedné kolize pomalých algoritmů

FaceT algoritmy opět mají ty nejvyšší a nejméně stálé hodnoty, zatímco ostatní algoritmy mají hodnoty velmi podobné. Zase je vidět že 1. algoritmus (vestavěná) a 2. algoritmus (nej rozměr) jsou o něco rychlejší než ostatní. Ovšem u všech algoritmů jsou hodnoty vyšší než u grafu 4.1.13 a u grafu 4.1.14.

5.3. Výsledky testování v aplikaci Hřiště

1. algoritmus, algoritmus s vestavěnou funkcí

Tento algoritmus je lehce nadprůměrný. Výborně si poradí s rotací. Zvláště dobře reaguje na pravidelné objekty jednoduchých tvarů. Velmi dobře tak reaguje na desky a krychli. U těchto objektů je algoritmus velice přesný i u samých hranic. Nemá, ale dobrou reakci na nepravidelné a složité tvary. U koule jsou nepřesnosti přijatelné, ale knotu je algoritmus naprosto nepřesný a i u kruhu si algoritmus myslí, že dochází ke kolizi i když je kapsle uprostřed kruhu, kde se nic nenachází.

2. algoritmus, algoritmus s největším rozměrem

Tento algoritmus je bezesporu ten nejméně přesný. Nereaguje přesně na absolutně žádný tvar a rotace je nepřesnost ještě zhorší, zvláště u objektů nerovnoměrného tvaru jako je deska.Kolizi detekuje vždy až nepřijatelně brzy.Jediné pozitivum je, že nikdy netvrdí, že ke kolizi nedochází, když k ní dochází. Jednoduchost a primitivnost algoritmu je tak, až příliš vidět.

3. algoritmus, algoritmus s AABB

Tento algoritmu je velice přesný pro pravidelné objekty jako krychle, kvádry a přípustně i koule. Ohledně rotace reaguje dobře na rotaci, která je blízko úhlům 90, 180, 270 a 360 stupňů. Avšak poměrně špatně reaguje na nepravidelné tvary, kdy je kolize detekována příliš brzy. Nejhorší výsledky ovšem byly u dlouhých objektů jako deska pod úhlem 45 stupňů, kdy detekce byla detekována buď až nepřijatelně příliš brzy nebo u okrajů nebyla kolize detekována vůbec. Přesnost tohoto algoritmu tak patří spíše k těm horším.

4. algoritmus, algoritmus s FaceT a počítadlem vzdálenosti

Tento algoritmus je velice přesný pro všechny tvary i rotace. Jako první algoritmus dokáže správně reagovat na prázdný prostředek kruhu a dokonce i na samotný knot. Má, ale i chyby. Když jsou objekty blízko, algoritmus hlásí kolizi o trochu dříve, než by měl. Algoritmus není schopen rozpoznat kolizi, pokud je malý objekt uvnitř velkého, čímž jsou stěny objektů příliš daleko od sebe. Když jsou objekty v kolizi a pohybují se přitom, algoritmus “problikává”. To znamená, že občas detekuje kolizi a občas ne. Tato nepřesnost, by se dala odstranit, jenže poté by algoritmus byl příliš pomalý a jeho použití by vše akorát zaseklo.

5. algoritmus, algoritmus s OBB

Tento algoritmus má velmi podobné vlastnosti jako 1. algoritmus, algoritmus s vestavěnou funkcí. I tento algoritmus reaguje velmi dobře na veškerou rotaci a reaguje špatně na nepravidelné objekty. Největším problémem a největší nepřesnosti se ale objevují, pokud se objekty dotýkají pouze svými “těly” a nikoli

svými hranami. Pokud tak dlouhá kapsle projde skrz úzkou desku, algoritmus to nezaznamená. Při kolizi správně narotovaných objektů by to představovalo problém.

6. algoritmus, algoritmus s FaceT a vektory

Tento algoritmus má velmi podobné vlastnosti jako 3. algoritmus, algoritmus s AABB. Opět reaguje dobře na pravidelné tvary, ale nereaguje dobře na nepravidelné tvary a rotace. Právě při rotacích detekuje kolizi příliš brzy. Ve srovnání s 3. algoritmem, ovšem reaguje odlišně na hrany rotovaných objektů. Zatímco 3. algoritmus při hranách nedetekoval kolizi vůbec, tento algoritmus ji detekuje příliš brzy.

7. algoritmus, algoritmus s OBB a per triangle

Přesnost tohoto algoritmu je velmi kolísavá. U objektů jako je koule a kapsle detekuje kolizi příliš brzy. U desky naopak reaguje naprosto přesně. Kolísavá je i přesnost u detekce rotovaných objektů. Algoritmus má s rotací problémy, které 5. algoritmus, algoritmus s OBB, nemá. U zrotovaných desek, které jsou zrotovány ve dvou osách je algoritmus až nepřijatelně nepřesný a kolizi zachytí buď pozdě nebo vůbec. Ale u rotovaných kruhů, které jsou rotovány pouze v jedné ose reaguje algoritmus dobře a rozpozná zrotované kraje objektu. Ukázalo se tak, že algoritmus umí pracovat s úhly. Stále, ale nedokáže rozpoznat prázdný střed kruhu. Také má problémy rozpoznat zda je objekt uvnitř jiného objektu. V takovém případě ve většinou kolizi zachytí, ale občas nezachytí.

8. algoritmus, algoritmus s FaceT a per triangle

Je velice přesný, téměř stejně přesný jako 4. algoritmus, algoritmus s FaceT a počítadlem vzdálenosti.Algoritmus výborně reaguje na všechny tvary i rotace. Dokáže dokonce i rozpoznat prázdný střed kruhu. U knotu tu jsou jisté nepřesnosti, ovšem i tak algoritmus dokáže rozpoznat očko knotu.Největší nevýhoda je, že algoritmus nezkoumá celý povrch objektu. Jsou tak prázdná místa, která pokud jsou v kolizi, algoritmus tuto kolizi nezaznamená. Tento problém by se dal vyřešit lepším a účinějším vytvářením trojúhelníku z bodového obalu objektu. Algoritmus také nedokáže rozpoznat zda se objekt nachází uvnitř jiného objektu. K zachycení kolize se zkrátka objekty musí dotýkat stěnami.

5.4. Srovnání algoritmů

Testování prokázalo, že algoritmy založené na podobných principech mají podobné vlastnosti. 4. algoritmus, algoritmus s FaceT a počítadlem vzdálenosti a 8. algoritmus, algoritmus s FaceT a per triangle, mají vysokou přesnost a dokáží si poradit i s objekty, které mají velice nepravidelné tvary včetně kruhu, ale jsou velmi pomalé. Ukázaly tak, jak moc jsou nezbytné optimalizační algoritmy, protože bez nich by FaceT algoritmy ani nebylo možné testovat. Oproti tomu algoritmy využívající

principu Oriented Bounding Box mají přijatelný či výborný výpočetní výkon, dobře si poradí s rotací objektů, ale neporadí si s nepravidelnými objekty. Oproti tomu algoritmy využívající Axis-Aligned Bounding Box a Sphere jsou využitelné pouze jako optimalizační algoritmy. Jejich vysoká rychlost a nízký (v případě 2. algoritmus, algoritmus s největším rozměrem, až žalostný) výkon tomuto odpovídají.

Ukázalo se také, že ve srovnání s knihovnou Cannon.js si jsou algoritmy velmi podobné při testování v 1. aplikaci, Kostce. Křivky grafů mají velmi podobný tvar a to jak u počtu kolizí, tak u výpočetního času. Ale u 2. aplikace, Padání, jsou tvary křivek velmi odlišné. Cannon.js zachytí nejvíce kolizí spíše ke konci, ale algoritmy spíše na začátku. U Cannon.js výpočetní čas neustále stoupá, ale u algoritmů je víceméně stálý nebo dokonce klesá. Algoritmy, tak mají podobné vlastnosti jako Cannon.js ve scéně, kde je mnoho objektů daleko od sebe a jen občas jsou v kolizi, ale jsou odlišné ve scéně, kde je mnoho objektů na sebe velmi blízko sebe a tím dochází ke mnoha kolizím najednou.

1. algoritmus, algoritmus s vestavěnou funkcí

Tento algoritmus je zcela určitě nejlepší na optimalizaci a na detekci kolizí u jednoduchých tvarů. Ve srovnání s ostatními Oriented Bounding Box algoritmy má tu největší přesnost a také nejlepší výpočetní výkon. Byl by tak výborný v kombinaci s pomalejšími, ale přesnějšími algoritmy. Ze všech Oriented Bounding Box algoritmů je tento algoritmus bezesporu ten nejlepší. Díky testování je tak velice dobře vidět, že použitá vestavěná funkce byla vytvořena profesionály s cílem zajistit velmi dobrou přenos a excelentní výpočetní rychlost.

2. algoritmus, algoritmus s největším rozměrem

Tento algoritmus se na detekci kolizí vůbec nehodí. Na to je až příliš nepřesný. Dal by se sice použít na optimalizaci, ale to pouze v tom případě, že se ve scéně budou nacházet tvary rovnoměrných rozměrů, jako jsou koule a krychle. Pokud by ve scéně byla “země” s velkou plochou, algoritmus by selhal.

3. algoritmus, algoritmus s AABB

I tento algoritmus je velmi dobře použitelný na optimalizaci i přesto, že má mírně horší výpočetní výkon. Hlavní použití by našel ve scéně, kde objekty nejsou rotované a není tak potřeba použít složitější algoritmus, který rotací počítá. Pokud by se ve scéně objevil rotovaný objekt jako “země” nebo dlouhá tyč, algoritmus by ztratil účinnost.

4. algoritmus, algoritmus s FaceT a počítadlem vzdálenosti

Bezesporu se jedná o ten nejpřesnější algoritmus, ale také o ten nejpomalejší, proto má pouze cenu ho využít tam, kde je potřeba vysoká přesnost, kde jsou k dispozici dostatečně výkonné počítače a to v kombinaci s jakoukoli optimalizací. Velmi

nevhodné by bylo použít algoritmus ve scéně, kde je velké množství objektů příliš blízko sebe.

5. algoritmus, algoritmus s OBB

Tento algoritmus je takovou horší verzí 1. algoritmu, algoritmu s vestavěnou funkcí. Byl by jistě velice dobrý na optimalizaci, ale poměrně mu škodí jeho nepřesnost, když se objekty zrotují tak aby byly kolizi pouze svými těly. To není dobré zvláště, když výhodou tohoto algoritmu by mělo být právě počítání rotací.

6. algoritmus, algoritmus s FaceT a vektory

Přestože tento algoritmus využívá FaceT a má i podobně vysokou rychlost, jeho přesnost připomíná spíše Axis-Aligned Bounding Box algoritmus. Nedá se tak využít ani na optimalizaci, ani na přesnou detekci kolizí. Někdo by možná dokázal tento algoritmus zlepšit, ovšem z těchto algoritmů je tento algoritmus v této podobě ten nejvíce neužitečný.

7. algoritmus, algoritmus s OBB a per triangle

Tento algoritmus je typický příklad špatného využití dobé věci. Přestože per triangle detekce kolizí se hojně využívá a je hodně přesná, u tohoto algoritmu je nepřesná u objektů s rotací v několika osách, což z něj činí zřejmě ten nejméně přesný Oriented Bounding Box algoritmus.

8. algoritmus, algoritmus s FaceT a per triangle

Tento algoritmus je téměř tak přesný jako 4. algoritmus, algoritmus s FaceT a počítadlem vzdálenosti, ale je o něco rychlejší, díky čemuž by bylo možné ho využít i když se ve scéně nachází mnoho objektů blízko sebe. Pokud by se tak zlepšila triangulace povrchu, který tento algoritmus vytváří kolem objektu a algoritmus se zkombinoval s dobrou optimalizací, jednalo by se zřejmě o ten nejvhodnější algoritmus na přesnou detekci kolizí.

6. Závěr

Cílem této práce bylo nastudování si detekce kolizí. Především pak techniky a algoritmy detekce kolizí. Dále bylo nutné nastudovat Javascriptové knihovny na vytvoření testovacích aplikací, které sloužily k měření výkonu vytvořených algoritmů a také k testování jejich vlastností. Vzhledem k tomu, že vybraná Javascriptová knihovna, Babylon.js, má výborné tutoriálové stránky a také vlastní forum, kde se uživatelé ptají pouze na problémy spojené s Baylon.js, bylo toto nastudování relativně jednoduché. Horší bylo nastudování fyzikální knihovny Cannon.js, která zajišťuje aplikaci detekci kolizí a fyzikální vlastnosti. Tuto knihovnu bylo důležité si nastudovat, aby bylo jasné jak vypadá složitá detekce kolizi v 3D prostoru napsaná

profesionálními programátory. Také bylo nutné změřit výkon detekce kolizí této knihovny, aby bylo s čím porovnávat vytvořené algoritmy. Nastudování této knihovny bylo těžší, protože její dokumentace nebyla tak detailní, jak by na tak rozsáhlou knihovnu bylo potřeba. Bylo tak nutné prostudovat postupně kód a pomocí malých zásahů nebo vypsáním proměnných do konzole zjistit, co která část dělá, dokud se nenašly potřebné části kódu, které prováděly detekce kolizí a které tak bylo možné konečně studovat a změřit, neboť se nyní vědělo kam umístit příkazy na zaznamenání času. Náročným problémem bylo i nastudování matematiky, kterou využívají některé algoritmy na rotaci objektů. Nešlo jen o strojové nastudování matematických vzorců, ale i o to jak je zaimplementovat do kodu. Při počátečních implementací totiž zrotované objekty měnily svou pozici a nebo nebyly rotovány podle požadovaných úhlů. Také bylo potřeba pochopit jak zajistit aby se rotace nekonala kolem osy objektu, ale kolem osy jiného objektu, což vyžadovalo nejen pochopení výpočtů, ale i mnoho pokusů.

Implementace nových algoritmů vycházela ze znalostí nastudovaných během studia o detekci kolizí. Některé algoritmy tak jsou pouhé přepsání teorie do programu, například algoritmy využívající Axis-Aligned Bounding Box.nebo Oriented Bounding Box. U jiných algoritmů byly potřeba znalosti o Babylon.js, např algoritmus využívající vestavěnou Babylon.js funkci nebo algoritmy využívající FaceT. Využila se i pokročilá trigonometrická matematika u algoritmů, které pracovaly i s rotací a musely si tak samy objekt zrotovat pomocí původních souřadnic, úhlů rotace a matematických rovnic. Samotné detekce kolizí, tedy části algoritmů, které vysloveně stanoví zda jsou objekty v kolizi nebo ne, využívají různé metody. Některé jsou opět klasické a vyskytují se i v různých článcích o detekci kolizí nebo jsou i v jednoduchých tutoriálech na implementaci detekce kolizí. Například když jsou objekty dostatečně blízko, když jsou jejich nejbližší části dostatečně blízko, když se překříží jejich schránky, kterými jsou obaleny nebo když se nějaký bod objektu nachází uvnitř schránky druhého objektu. Jiné jsou, ale velice svérázné a zřejmě se nevyskytují v mnoha fyzikálních enginech či jiných profesionálních programech, především řešení, které využívá směrové vektory a testuje zda směřují do všech čtyřech směrů. Některá řešení fungují uspokojivě, některá už ne a jsou spíše nepoužitelná. Některá řešení reagují dobře na nepravidelné tvary. Některá reagují dobře na rotaci objektů. Především ty jednodušší algoritmy, které nemají dobrou přesnost, ale mají dobrý výkon a rychlost by se hodily do kombinace s pomalými, ale přesnými algoritmy, čímž by vznikla dobrá rovnováha mezi rychlostí a přesností. Byly tak vytvořeny algoritmy, které by se jistě daly využít i v nějakém fyzikálním či herním enginu.

6.1. Návrhy na zlepšení

Místo Babylon.js pracovat s Three.js. Three.js je stejně jako Babylon.js WebGL framework, díky kterému se mnohem snadněji vytváří 3D aplikace. Three.js, ale nabízí některá rozšíření, která by při vytváření algoritmů mohla být velice užitečná

například objekt trojúhelník. Tato rozšíření by programátoři, kteří mají s Three.js zkušenosti jistě mohli velmi dobře využít a i pokud by algoritmy nezměnily ohledně jejich principu, tak by je mohli zefektivnit a zoptimalizovat. Také je otázka zda má Three.js lepší vestavěné funkce na detekci kolizí, čili Three.js obdobu k Babylon.js funkci “intersectsMesh”.

Místo Javascriptu pracovat s C++. C++ je jazyk ve kterém je napsáno mnoho fyzikálních enginů zabývající se detekcí kolizí, např OpenGL. Využití OpenGl a jeho schopností implementovat maticové a vektorové struktury, by mohly algoritmům velice prospět. Je tak otázka zda se C++ celkově nehodí víc na detekci kolizí než Javascript, zvláště pokud pomineme fakt, že Javascript se využívá na webové aplikace, zatímco C++ desktopové. Přestože srovnávání C++ a Javascriptu není úplně přesné kvůli jejich odlišnému využití, mohlo by to být zajímavé a přinést i zajímavé výsledky.

Algoritmy by se mohly zkombinovat tak aby se nějaký rychlý a nepřesný algoritmus použil na optimalizaci a nějaký pomalý a přesný na samotnou detekci kolizí. Stačilo by pouze vytvořené algoritmy zkombinovat a tyto kombinace prohnat testovacími aplikacemi. Výsledky dají vědět, která kombinace algoritmu je nejúčinnější a dokonale tak kombinuje rychlost s přesností. 8. algoritmus, algoritmus s FaceT a per triangle by také mohl mít lepší způsob, jak vytvořit trojúhelníky z jeho bodového obalu a to tak, aby se zkontroloval celý povrch objektu, ale zároveň se nedělaly zbytečné výpočty navíc. Takový algoritmus by byl ještě přesnější, ale za to rychlejší než 4. algoritmus, algoritmus s FaceT a počítadlem vzdálenosti. 6. algoritmus, algoritmus s FaceT a vektory by jistě bylo možné nějak zlepšit, aby byl přibližně tak přesný jako ostatní FaceT algoritmy, ale byl by stále viditelně rychlejší. Ovšem toto zlepšení by jistě bylo velice náročné a vyžadovalo by jak tvůrčí schopnosti, tak znalosti vektorové matematiky.

Bibliografie

[1] EBERLY, David. Geometric tools. Dynamic Collision Detection using Oriented Bounding Boxes. [online]. [cit. 2016-05-24]. Dostupné z: https://www.geometrictools.com/Documentation/DynamicCollisionDetection.pdf [7] ER

[2] EBERLY, David H. Intersection of Convex Objects: The Method of Separating Axes [online] 2001, 2008. Dostupné z: https://geometrictools.com/Documentation/MethodOfSeparatingAxes.pdf

[3] JIA, Yan-Bin. Quaternions and Rotations [online] Stanford Computer Graphics Laboratory, 2013. Dostupné z: http://graphics.stanford.edu/courses/cs348a-17-winter/Papers/quaternion.pdf

[4] CURTIS, Sean. Fast Collision Detection for Deformable Models using Representative-Triangles Dostupné z: https://www.researchgate.net/publication/220791990_Fast_Collision_Detection_for_Deformable_Models_using_Representative-Triangles

[5] ERICSON, Christer. Real-time collision detection. Boston: Elsevier, c2005. ISBN 1558607323.

[6] ŽÁRA, Ondřej. Programátorské techniky a webové technologie, 2015. ISBN: 9788025150269

[7] ODVÁRKO, OLDŘICH. Matematika pro gymnázia - Goniometrie . ISBN: 978-80-7196-359-2

[8] Babylon.js dokumentace[online]. 2022. Dostupné z: https://doc.babylonjs.com/.