Technická Univerzita v Liberci

Fakulta mechatroniky, informatiky a mezioborových studií

Studijní program: N2612 Elektrotechnika a informatika

Studijní obor: Informační technologie

Pokročilé metody rešení kolizí 3D objektu v aplikaci IREView

Advanced solution methods of 3D objects collisions for IREView

DIPLOMOVÁ PRÁCE

Autor: Bc. Zbyněk Hlava

Vedoucí práce: Ing. Jiří Hnídek PhD.

V Liberci 10.5.2013

Zadání

vložit originální zadání

Prohlášení

Byl(a) jsem seznámen(a) s tím, že na můj magisterský projekt se plně vztahuje zákon č. 121/2000 Sb. O právu autorském, zejména § 60 - školní dílo.

Beru na vědomí, že Technická Univerzita v Liberci(TUL) nezasahuje do mých autorských práv užitím mého magisterského projektu pro vnitřní potřeby TUL. Užiji-li magisterský projekt nebo poskytnu-li licenci k jeho využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Magisterský projekt jsem vypracoval(a) samostatně s použitím uvedené literatury a na základě konzultací s vedoucím projektu.

V Liberci dne _____ _____

podpis

Poděkování

Děkuji především vedoucímu této práce Ing. Jiřímu Hnídkovi PhD. za trpělivost a pomoc při řešení problémů. Dále Ing. Martinu Huškovi za dodávání potřebných podkladů a usilovnou spolupráci na vývoji aplikace a Doc. Ing. Antonínu Potěšilovi Csc. za motivaci a pomoc při psaní této práce. Dále pak panu Sergeji Reichovi za pomoc při provádění úprav ve zdrojových kódech Blenderu. V neposlední řadě bych chtěl poděkovat své rodině za možnost studia a za celkovou podporu.

Abstrakt

Cílem předložené práce je naprogramovat a popsat pokročilé metody řešení kolizí v aplikaci IREView Blender. Práce se věnuje části vývoje aplikace IREView v rámci projektu „Inovace technologie výroby umělých kůží“ firmy LENAM s.r.o., Technické univerzity v Liberci a firmy Magna Exteriors & Interiors (Bohemia), s.r.o. Pro provádění detekce bylo naprogramováno několik různých způsobů. Ty jsou popsány v předložené práci. Vše je naprogramováno v jazyce Python jako skript pro prostředí Blender. Závěrem práce je testování a porovnání skriptů.

Klíčová slova

IREView, Blender, Detekce kolizí, SAT, Bullet Physics, Python

Abstract

Goal of this work is to program and describe advanced methods of collisions solution in IREView Blender. This work is part of a project "Innovation of production technologies for creating imitation leather"of LENAM s.r.o., Technical University in Liberec and Magna Exteriors & Interiors (Bohemia), s.r.o. Several methods of collision detection were implemented. These methods are described in this paper. Collision detection methods wre implemented in Blender using Python programming language. Tests and comparisons were made at the end of this work.

Keywords

IREView, Blender, Collisions detection, SAT, Bullet Physics, Python

Obsah

Seznam obrázků

Seznam symbolů, zkratek a termínů

SAT - Separating Axis Theorem (teorém oddělující osy)

GNU GPL - GNU General Public License

API - Application Programming Interface

IRE - InfraRed Emitter (infračervený zářič)

AABB - Axis Aligned Bound Box (osově zarovnaný hraniční kvádr)

OABB - Object Aligned Bound Box (oběktově zarovnaný hraniční kvádr)

BGE - Blender Game Engine

RB - Rigid Body

FPS - Frames per Second (snímky za sekundu)

SPS - Steps per Second (kroky za sekundu)

Kapitola 1

Úvod

Předložená práce se zabývá částí vývoje aplikace IREView. Ta je vyvíjená týmem řešitelů z firmy LENAM s.r.o., Technické univerzity v Liberci a firmy Magna Exteriors & Interiors (Bohemia), s.r.o. jako jeden z cílů projektu MPO TIP 2009 „Inovace technologie výroby umělých kůží, ev.č. FR - TI1/266“. Vývoji hlavních funkcí aplikace se věnuje Ing. Martin Hušek z firmy LENAM s.r.o.

1.1 Motivace

Na projektu „Inovace technologie výroby umělých kůží, ev.č. FR - TI1/266“ se podílím již několik let a v rámci jeho plnění byla vytvořena moje bakalářská práce i magisterský projekt. Na vývoji aplikace IREView se podílím i mimoškolně. Proto možnost pokračovat i v rámci diplomové práce byla více než nasnadě.

1.2 Cíle práce

Cílem diplomové práce je zvážit možnosti řešení kolizí v aplikaci IREView Blender a vytvořit vhodnou metodu, schopnou detekce kolizí mezi objekty IRE zářičů a dalšími složitými objekty, které se účastní simulace. V ideálním případě pak zdokonalit již existující možnost detekce kolizí mezi samotnými zářiči.

Kapitola 2

Výroba umělých kůží a IREView

Pro lepší seznámení s problematikou výroby umělých kůží, je zde krátký popis jejího postupu a důvody její inovace, jak je uvádí kniha Ohřevy radiací.

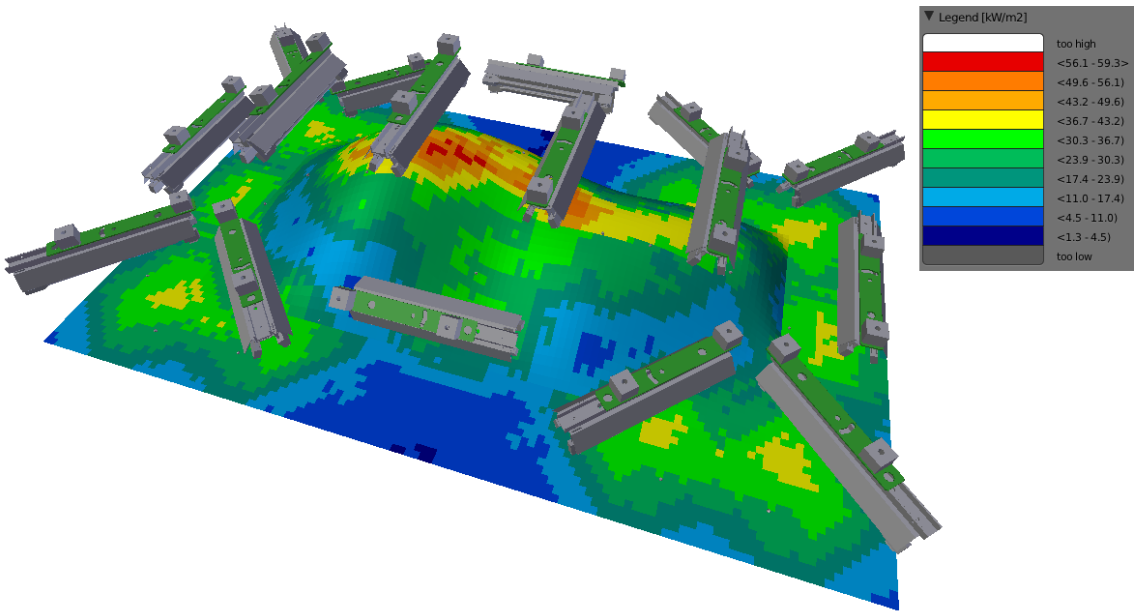
Proces výroby umělé kůže [13] technologií Slush moulding probíhá tak, že na líc horké kovové skořepinové formy se nanese polymerový prášek na bázi PU nebo PVC, který se nataví a slinuje do tenké celistvé vrstvy. Forma dá umělé kůži příslušný tvar a také přesný otisk svého povrchu. Z důvodů produktivity je žádoucí, aby fáze ohřevu i chladnutí formy byly co nejrychlejší. Proces kvalitního slinování materiálu kůže však vyžaduje dodržení poměrně úzkého rozsahu teplot s šířkou přibližně $20°C$.

Už při rozhodování o konceptu evidentně nestacionárního ohřevu skořepivových forem se ukázalo, že technická příprava ohřevu formy a následná realizace vyžaduje užití virtuálních simulací ohřevu. V současné době je k ohřevu forem využíváno řádově desítek zářičů v závislosti na rozměrech a tvarové složitosti formy. Používají se zářiče několika různých typů a radiačních výkonů.

Z výše uvedeného popisu zvoleného typu ohřevu je patrné, že základem technické přípravy ohřevu je rozmístění velkého počtu infarzářičů nad plochou skořepinové formy. První pokusy rozmístit zářiče metodou ad-hoc do držáků na konstrukci tzv. „ohřívacích zad“ však nepřinesly očekávaný výsledek. Proto byla pracovníky firmy LENAM, s.r.o. vyvinuta metoda rozmístění infrazářičů s využitím prostředí softwarových nástrojů CAD (Computer Aided Design) a FEM (Finite Element Method) simulačních systémů.

Bylo také rozhodnuto nahradit původně využívaný licencovaný CAD software (ProE) nástrojem zcela novým, který by byl přívětivější pro uživatele a používané techniky a také měl více speciálních funkcí potřebných k přípravě a vhodné optimalizaci ohřevu forem. Tím je právě aplikace IREView.

Aplikace IREView je vystavěna na nejnovější verzi programu Blender, tj. Blender



Obrázek 2.1: Ukázka simulace ohřevu v IREView

2.6. Pro úpravy a rozšíření prostředí je možné vytvářet skripty v jazyce Python (verze 3.1.) Jejím hlavním cílem je simulovat výše zmíněný proces výroby umělých kůží a z něj potom získat pozice infrazářičů pro rozmístění v reálném provozu. Jedna z funkcí, kterou má software disponovat, je možnost detekce kolizí mezi jednotlivými objekty, které se simulace účastní.

Scéna virtuálního ohřevu se skládá z několika částí. Hlavním objektem je forma, její vhodné ohřátí je cílem celé simulace. Dále infračervené zářiče IRE(InfraRed Emitter), které zařizují dané ozáření. Poslední důležitou součástí je objekt zastupující rám („ohřívací záda“), na kterém budou zářiče upevněny v reálném provozu.

Aplikace disponuje skriptem schopným detekovat kolize mezi jednotlivými objekty zářičů, ten byl vytvořen v rámci mého magisterského projektu. Skript dokáže detekovat kolize pouze mezi zářiči navzájem. V případě detekce kolizí mezi infrazářiči a rámem či formou není dostatečně přesný. Je tedy třeba tuto možnost vytvořit možnost. A to tak, aby detekce probíhala nejpřesněji jak je možné.

Kapitola 3

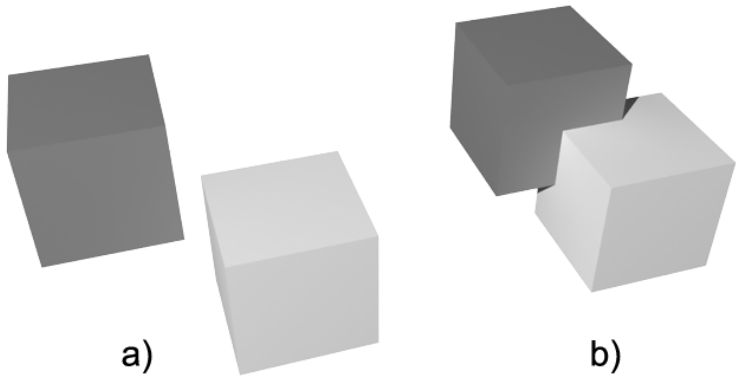
Teorie

Pro lepší pochopení práce budou v následující kapitole vysvětleny některé důležité pojmy a popsány použité techniky a software.

3.1 Kolize

Práce se zabývá řešením kolizí, bylo by tedy dobré nejprve vysvětlit a ukázat, co je vlastně kolize v 3D prostoru. Jedná se o stav, kdy dva či více objektů sdílí stejnou část prostoru. U objektů, s jakými se v IREview pracuje[1] (tuhá tělesa), by v reálném světě, taková situace nemohla nastat. Ve virtuálním prostoru však může uživatel umístit objekty dle libosti, z čehož vyplývá, že k takové situaci může dojít. To je také důvod proč je potřeba tyto situace detekovat a odstranit. Ukázka kolizního a nekolizního stavu dvou objektů na Obrázku 3.1.

[1] Všechny objekty zmiňované v této práci jsou tvořeny hraniční reprezentací, její popis a další druhy je možné najít například na: www.root.cz/clanky/opengl-evaluatory-i/



Obrázek 3.1: Ukázka (ne)kolizního stavu: a) nekolizní stav, b) kolize

3.2 Složitost algoritmu

V průběhu práce bude používán pojem (asymptotická) složitost algoritmu, zde je její definice [14]:

„*Asymptotická složitost algoritmu* ***A*** *je řád růstu funkce f(N), která charakterizuje počet elementárních operací algoritmu* ***A*** *při zpracování dat o rozsahu* ***N***“.

Běžný zápis složitosti algoritmu je „velká O notace“ (z anglického big-O notation), například $O(N)$.

Složitost dělí algoritmy na třídy, v jedné třídě je možné výkonem používaného zařízení vymazat rozdíl náročnosti výpočtu. Například dva algoritmy, jeden se složitostí $O(N^2)$ a druhý $O(200N^2)$. Oba jsou ve stejné třídě, jelikož ke srovnání doby výpočtu stačí pustit druhý algoritmus na dvakrát rychlejším počítači. Pokud ale bude mít druhý algoritmus složitost $O(N!)$ rozdíl se nedá odstranit libovolně výkonným zařízením.

Některé významné (třídy) složitosti (seřazené od nejmenší):

$O(1)$ - konstantní

$O(logN)$ - logaritmická

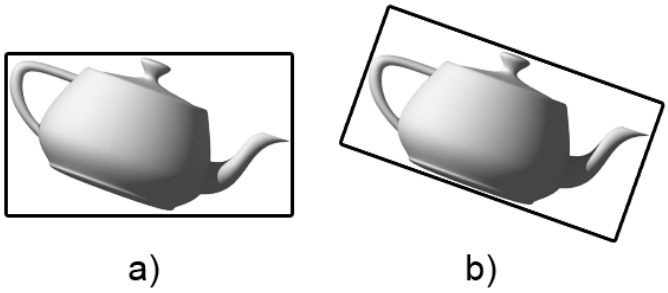
$O(N)$ - lineární

$O(NlogN)$ - lineárnělogaritmická

$O(N^2)$ - kvadratická

$O(X^N)$ - exponenciální

$O(N!)$ - faktoriálová



Obrázek 3.2: Ukázka kolizních obálek: a) AABB, b) OABB

3.3 Kolizní obálky a detekční algoritmy

Kolizní obálky[5][9] jsou aproximací modelů, jejich tvar může být různý. Obálky pak slouží k reprezentaci složitých objektů v procesech fyzické simulace, především pak detekce kolizí. Důvodem užívání obálek je urychlení výpočtů.

Základním a nejjednodušším tvarem užívaným pro detekci kolizí v 3D prostoru je koule. Její tvar udává pouze jedna proměnná, tou je poloměr. Detekce kolize mezi dvěma koulemi[5] je řešena tak, že se porovná vzdálenost(d) středů(k_1, k_2) koulí a součet(sr) poloměrů($k_i.r$). Matematicky vyjádřeno:

$$d = \sqrt{(k_1.x - k_2.x)^2 + (k_1.y - k_2.y)^2 + (k_1.z - k_2.z)^2}$$

$$sr = k_1.r + k_2.r$$

Pokud platí $d > sr$, pak nedošlo ke kolizi v opačném případě koule v kolizi jsou.

Velice užívaným tvarem je pak kvádr. Ten se vyskytuje ve dvou možných variantách. Tou jednodušší je AABB (Axis Aligned Bound Box), neboli osově zarovnaný hraniční kvádr. Aproximace některých objektů se oproti kouli použitím AABB zpřesní.

Detekce pomocí AABB[5] je opět velice jednoduchá. Pro získání výsledku je nutné zkontrolovat vzájemné pozice minimálních a maximálních hodnot kvádrů ve všech třech osách. Neboli, pokud zároveň platí všechny tyto rovnice

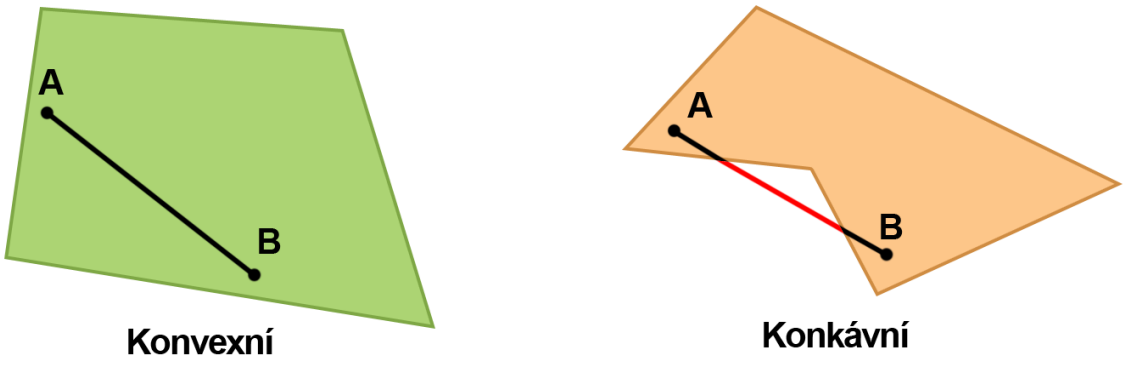
$$B1.MaxX > B2.MinX$$
$$B1.MinX < B2.MaxX$$
$$B1.MaxY > B2.MinY$$
$$B1.MinY < B2.MaxY$$
$$B1.MaxZ > B2.MinZ$$
$$B1.MinZ < B2.MaxZ$$

pak jsou kvádry v kolizi. Detekci lze provést také metodou podobnou detekci koulí. Rozdíl spočívá v tom, že je provedena pro každou osu zvlášť a poloměr je nahrazen polovinou délky příslušné strany.

Druhou variantou je OABB (Object Aligned Bound Box), jak název napovídá, odlišností od předchozí verze je v natočení kvádru společně s objektem. Jedná se o velice používanou variantu, jelikož aproximace objektu a rychlost výpočtu jsou často ve výhodném poměru. Mnoho reálných tvarů je kvádru podobných a jelikož se obálka natáčí spolu s objektem, je aproximační reprezentace objektu stále stejně velká (narozdíl od AABB, kde se její velikost mění, v závislosti na natočení objektu). Detekci lze pak provést například za pomocí teorému oddělující osy, který bude vysvětlen v kapitole 3.4.

Další možná zjednodušení jsou různé jiné konvexní tvary jako *Ellipsoid, Convex Hull, apod.*, které se více či méně přibližují reálnému tvaru objektu. S rostoucí složitostí však klesá rychlost detekce v závislosti na přibývajícím počtu nutných výpočtů. Je také možné seskupovat dané obálky do skupin, čehož se využívá především v případech, kde se jedná o objekty složené z více částí a aproximace jednou obálkou by nebyla dostatečná. Je také možné řešit kolize mezi různými druhy kolizních obálek, například mezi koulí a kvádrem.

Všechny tyto testy mají složitost $O(N^2)$, jelikož je třeba projít všechny páry postupně. Tedy pro N objektů je třeba udělat N^2 detekčních testů. Bude-li k testování například 20 objektů, bylo by třeba vykonat $20^2 = 400$ detekcí. Počet testování lze snížit, není nutné kontrolovat objekt sám se sebou, a jelikož nezáleží na pořadí objektů, stačí provést jen polovinu testů. Třída složitosti však zůstává $O(N^2)$(vysvětleno v kapitole 3.2).



Obrázek 3.3: Konvexní a nekonvexní objekt (2D)

3.4 Teorém oddělující osy

Pro lepší pochopení teorému oddělující osy je nejprve nutné vysvětlit, co je to konvexní objekt (množina).

V matematice se pod pojmem konvexní množina[16] rozumí podmnožina Euklidovského prostoru, která má následující vlastnost:

- úsečka spojující libovolné dva body této množiny je v dané množině rovněž obsažena.

Jde tedy o množinu M takovou, že pro všechny body $A, B \in \mathbf{M}$ platí:

$$AB \subseteq \mathbf{M}.$$

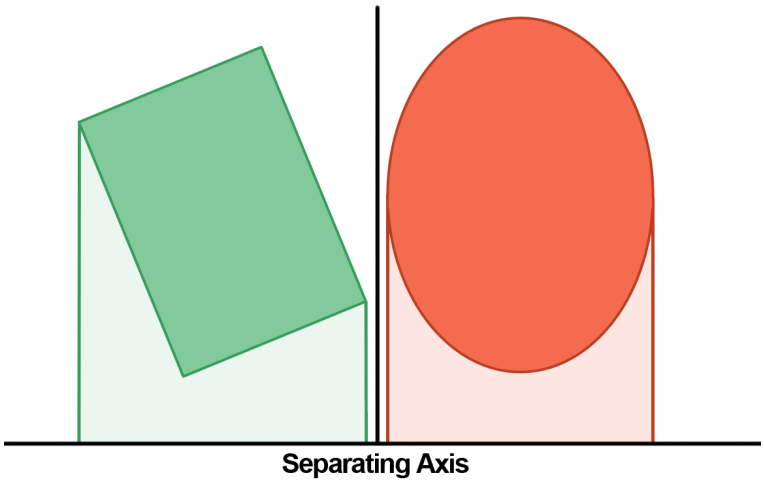
Analyticky to lze vyjádřit tak, že pro všechna $a, b \in \mathbf{M}$ je splněna podmínka :

$$\overline{ab} := \{\lambda a + (1 - \lambda)b \mid 0 \leq \lambda \leq 1\} \subseteq M.$$

Pro lepší představu je grafické znázornění na. Obrázku 3.3.

Další důležitou částí teorému je slovo projekce. Pod pojmem projekce se rozumí několik různých významů. Pro účely této práce je projekce stín 3D tělesa na rovné ploše. Ve dvojrozměrném prostoru se projekce objektu projeví jako úsečka jak je vidět na Obrázku 3.4, kde je znázorněna paralelními úsečkami kolmými k oddělující ose (Separating Axis).

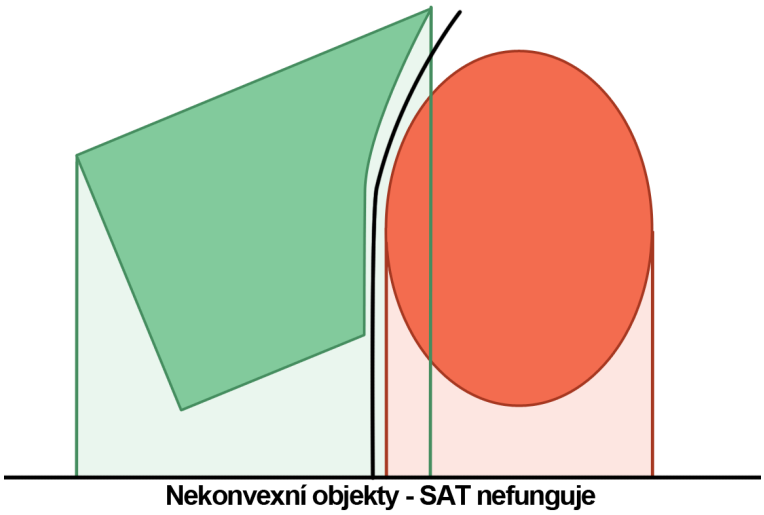
Anglicky *Separating Axis Theorem*, zkráceně SAT tvrdí [2]: **Pokud nejsou dva konvexní objekty v kolizi, pak existuje alespoň jedna osa, na které se jejich projekce nepřekrývají.** Pro 2D prostor je tou osou přímka, v 3D prostoru pak rovina. Teorém funguje pouze pro konvexní objekty, jak je patrné z Obrázku 3.5.



Obrázek 3.4: SAT ve 2D prostoru

Algoritmus využívající SAT je pro detekci kolizí často užíván například v počítačových hrách či simulacích s jednoduchými objekty. Ty kladou nároky na rychlost nikoliv nezbytně na přesnost detekce. Projekčních ploch je v podstatě nekonečno, proto je třeba testování soustředit pouze na určité z nich. Ve 2D prostoru jsou jimi projekční osy, které jsou rovnoběžné s normálovými vektory hran zúčastněných objektů. Pro urychlení je možné vynechat paralelní osy, projekce budou stejné. Pro dva obdélníky je například nutné zkoumat pouze čtyři osy místo všech osmi, jelikož jsou protilehlé strany obdélníku rovnoběžné.

Pokud se na všech těchto osách projekce prolínají, objekty se protínají. Algoritmu však stačí nalézt jednu osu, kde se projekce nepřekrývají a je možné detekci ukončit s výsledkem, že objekty v kolizi nejsou.



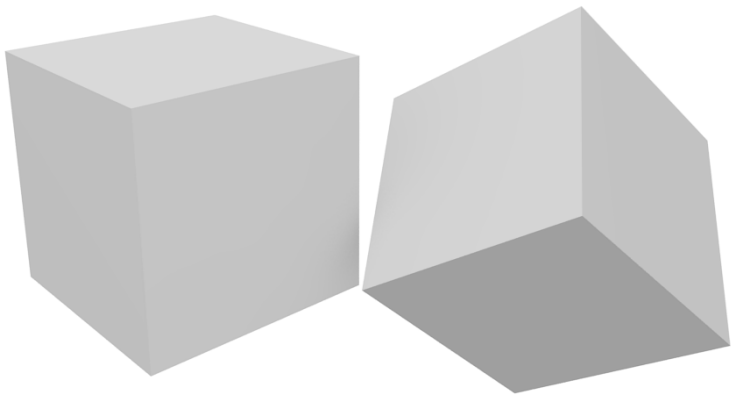
Obrázek 3.5: Nekonvexní objekty a SAT

Samotná implementace algoritmu je možná dvěma způsoby [2][11]. První je shodný s teorií, tedy zjištění možných oddělujících os, následně projekce objektů na zvolené osy a měření vzdálenosti případně hloubky průniku projekcí. Druhou možností je použít oddělující linie procházející stranami objektů (z předchozího textu vyplývá, že jsou kolmé na oddělující osy) a otestovat, zda leží body prvního a druhého zkoumaného objektu na opačných straně této linie. Pokud takový případ nastane, jejich projekce se nepřekrývají, tudíž ani objekty nejsou v kolizi. Tento způsob je vhodný pro složitější konvexní objekty, kde je větší množství neparalelních os a bylo by nutné projekci provést na všech z nich.

Pro 3D tělesa je nutno vzít v potaz ještě další možné oddělující osy (plochy). K jejich nalezení je nutné vzít všechny možné kombinace dvojic hran z prvního a druhého objektu a nalézt jejich normálové vektory. Potencionální oddělující plochy jsou s těmito vektory rovnoběžné. Bez nich hrozí falešný pozitivní výsledek v případě, kdy se objekty takřka dotýkají hranami, viz Obrázek 3.6. Po vyloučení paralelních ploch je pro přesnou detekci dvou kvádrů nutno zkontrolovat 15 možných oddělujících ploch.

3.5 IRE zářiče

Zářiče IRE (InfraRed Emitter) jsou jedním ze základních kamenů IREView, jejich pomocí se ohřívá forma, na které se tvoří umělé kůže. Detekce kolizí se provádí právě kvůli pozicím jednotlivých zářičů. Jednak se kontroluje jejich vzájemná poloha, ale i poloha



Obrázek 3.6: V tomto případě osy ploch nestačí

vůči rámu na kterém jsou upevněny nebo vůči formě kterou ohřívají.

V současné době je IREView vybavena sadou zářičů, která obsahuje čtyři druhy zářičů. Další typy budou postupem času přidány. Důležitou vlastností objektu je naměřená charakteristika. Ta udává intenzitu svícení v závislosti na natočení a vzdálenosti ozařované plochy. Každý druh má svůj vlastní model, podle toho jaké části obsahuje.

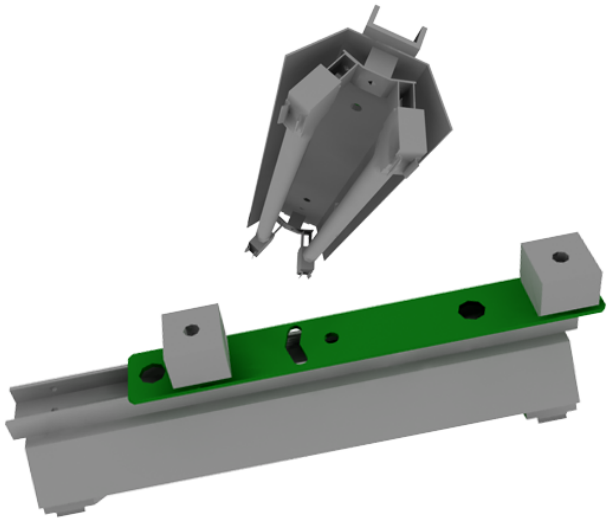
Původní skript užívaný k detekci kolizí je založen na OABB (popis OABB v kapitole 3.3). Ke zvýšení přesnosti detekce jsou zářiče v IREView rozděleny na více objektů, ty reprezentují jednotlivé části reálného IRE.

Je však třeba uživateli zajistit možnost pracovat s daným zářičem jako s jedním objektem. Jedna z částí (na Obrázku 3.7 je označena zeleně) je použita jako součástka hlavní. Její pomocí je ovládán celý zářič. Podřízenost dalších částí zářiče funguje na principu *Parent & Children*, tedy rodič a potomci. Princip podřízenosti je jednoduchý, transformace provedené na rodiči jsou aplikovány i na jeho potomky. Pokud však dojde k transformaci jednoho z potomků, neovlivní se tím ostatní potomci ani rodič.

Proto aby nedocházelo k nechtěným editacím vzhledu zářiče, je potomkům odebrána možnost být uživatelem aplikace vybrán, tím pádem i možnost přímé manipulace.

3.6 Blender

Blender [15] je multiplatformní, open source aplikace, zaměřená na vytváření 3D modelů, animací a dalších prvků počítačové grafiky. Disponuje nástroji pro modelování,



Obrázek 3.7: Ukázka zářiče

rendering, postprodukci a v neposlední řadě tvorbu interaktivních aplikací.

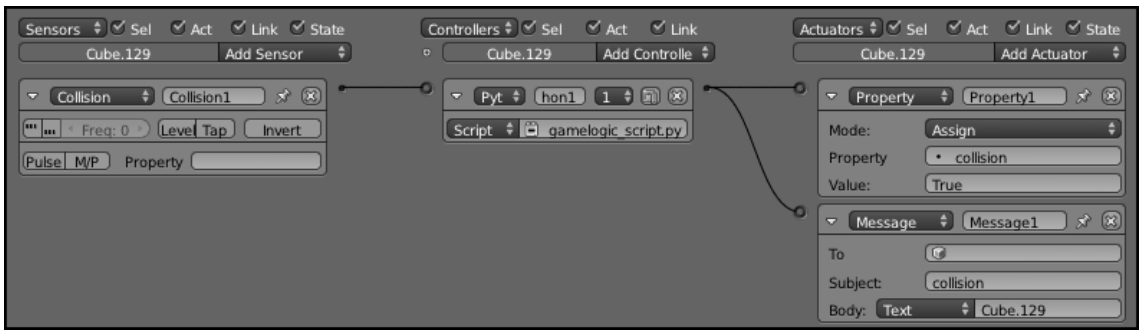
Blender je možné rozšiřovat pomocí Python skritů, možností rozšíření je mnoho, počínaje importem datových formátů, přes změnu uživatelského rozhraní, až po generování objektů.

3.7 Blender Game Engine

Blender Game Engine [4] je samostatná komponenta Blenderu. Je využíván pro tvorbu interaktivních aplikací pracujících v reálném čase, jako jsou například počítačové hry, architektonické, vědecké a průmyslové vizualizace, nebo animace. Je naprogramován v jazyce C++ a má otevřený kód. Disponuje opět možností rozšiřitelnosti pomocí Python skriptů. Součástí Game Engine je od verze 2.40 integrovaný fyzikální engine Bullet Physics.

Blender dovoluje tvořenou aplikaci spouštět přímo ve svém rozhraní nebo je možný export do spustitelného souboru.

Engine používá pro ovládání „virtuálního světa“ systém tzv. „logických cihel“. Jde o kombinaci snímačů (*Sensors*), kontrolerů (*Controllers*) a reakčních článků (*Actuators*). Ty jsou navzájem propojeny v pořadí jaké je uvedeno na Obrázku 3.8. Spojení probíhá v poměru *N:M*, je tedy možné k jednomu senzoru připojit více správců a podobně. Ukázka



Obrázek 3.8: Herní logika

„cihel“ viz. Obrázek 3.8.

3.8 Bullet Physics

Bullet [10] je open source fyzikální engine, distribuovaný pod licencí zlib[2].Jeho autorem je Erwin Coumans. Bullet obsahuje funkce jako detekci kolizí v 3D prostoru nebo dynamiku tuhých a měkkých těles. Využívá se hlavně v elektronickém zábavním průmyslu, především pak v herních enginech a při vizuálních efektech různých filmů.

3.8.1 Tuhé těleso (rigid body)

„*Tuhé těleso [18] je ideální těleso, jehož tvar ani objem se účinkem libovolně velkých sil nemění. Síly, které na těleso působí mají jen pohybové účinky. Z toho vyplývá, že se zanedbávají veškeré deformační účinky sil.*“

V reálném světě tuhé těleso neexistuje, jelikož působením sil se každé těleso více či méně deformuje. Reprezentace tuhým tělesem se používá k teoretickému zkoumání pohybových účinků sil na těleso v případech, kdy dané těleso nelze nahradit pouze hmotným bodem. To jest, když jeho rozměry a tvar nelze zanedbat, případně je nutno vzít v úvahu též jeho rotaci, a přitom jsou deformační účinky sil zanedbatelné.

[2]Její znění je dostupné z WWW: www.gzip.org/zlib/zlib_license.html

3.8.2 Simulace tuhých těles

Engine Bullet mimo jiné dovoluje simulovat mechaniku tuhých těles. Toho využívají některé metody naprogramované v rámci této práce, proto zde bude naznačeno jak.

Celá problematika simulace je rozložena na menší, dobře definované části, přičemž je každá odpovědná za řešení jednoduché úlohy. Všechny části jsou pak svázány dohromady *simulační smyčkou*, která se skládá ze tří hlavních části.

Jedním z důležitých faktorů rychlosti detekce je kolizní obálka, která objekt reprezentuje ve výpočtech fyzikálního enginu. Možné obálky jsou Capsula, Box, Sphere, Cone, Cylinder, Convex Hull a Mesh [9]. Tyto zjednodušující obálky mohou značně urychlit celou detekci.

Prvním krokem smyčky je *Detekce kolizí*, která určuje body dotyku mezi jednotlivými tělesy. Jak bylo zmíněno, při standardním přístupu by bylo složitost detekce $O(N^2)$. Tomu je dobré se vyhnout.

Pro rychlejší provedení je proces detekce rozdělen do několika fází [1]. První fáze se nazývá „široká“. Bullet v široké fázi nahradí objekty obálkami, konkrétně AABB, na nich pak použije jeden ze dvou algoritmů, které mohou značně redukovat počet párů u kterých je možný výskyt kolize. Těmi algoritmy jsou [7] *Sweep and Prune (SAP)*,složitost [12] je $O(NlogN)$ a *AABB Tree*, který má složitost lineární($O(N)$) [6], což je o třídu (respektive dvě) níže, než u detekce každý s každým.

Kolizní páry z první fáze jsou podrobeny testu „střední fáze“, která využívá lokální objektové souřadnice. Detekce zde probíhá na základě hraničního objemu těles (OABB, a jiné konvexní aproximace). Bullet pro detekci kolizí mezi konvexními objekty (obálkami) používá GJK (Gilbert–Johnson–Keerthi) algoritmus [8][10]. Ten je oproti algoritmu SAT univerzálnější a rychlejší [3]. Pro dvojice, které i zde dostaly označení kolizní, přijde na řadu „úzká fáze“[3], kde je pro detekci používána geometrie samotných těles. Tato detekce je časově náročná, proto je prováděna až jako poslední v případech, kde již není možné výpočty urychlit aproximací.

Druhou částí smyčky je pak *řešení kolizí*, kde jsou z kontaktních bodů odvozeny síly, které mění směr a rychlost pohybu objektů, tím zároveň brání jejich prolnutí. Poslední fáze má za úkol zahrnout předchozí zjištění a ostatní síly do časového rámce. Zde jsou objektům vypočítány nové pozice, natočení, rychlost a směr pohybu, atd. Smyčka se opakuje v určitém intervalu, ten je implicitně nastaven na 60 kroků za sekundu (SPS -

[3]pouze v případě, kdy je kolizní typ objektu Mesh, ostatní obálky jsou konvexní, jsou tedy detekovány ve střední fázi

steps per second), což znamená že při 60 FPS (frames per second) se smyčka provede před každým vykresleným snímkem právě jednou. Pokud bude animace vykreslována rychlostí 30 FPS, pak se smyčka provede dvakrát během jednoho snímku, a tak dále. Pokud engine nestíhá výpočty provádět, je FPS automaticky sníženo.

3.9 Python

Python [17] je interpretovaný programovací jazyk, tj. vykonává instrukce zapsané v kódu přímo. Opakem je překladač, který program nejprve přeloží do strojového kódu a teprve pak je možné ho spustit. Python v roce 1990 navrhl Guido van Rossum a je vyvíjen jako open source projekt, který nabízí instalační balíky pro většinu dostupných platforem. Jazyk byl pojmenován podle „Monty Python's Flying Circus", jehož je autor fanouškem. Při vývoji byl kladen důraz na tyto cíle:

- snadný a intuitivní jazyk, avšak dostatečně silný
- otevřený kód
- srozumitelný jako běžná mluva(angličtina)
- vhodný pro běžné úkoly, umožňující vývoj v krátkém čase

K význačným vlastnostem jazyka Python patří jeho jednoduché osvojení. Někteří ho dokonce považují za jeden z nejvhodnějších programovacích jazyků pro začátečníky. Je to dáno tím, že jedním z jeho inspiračních zdrojů byl jazyk ABC, který byl, jako jazyk pro výuku přímo vytvořen. Python ale zároveň bourá zažitou představu, že jazyk vhodný pro výuku není vhodný pro praxi. Podstatnou měrou k tomu přispívá čistota a jednoduchost syntaxe, na kterou se při vývoji jazyka dbalo. Python je také využíván jako skriptovací jazyk v aplikacích jako je Blender, Maya, Photoshop, atd.

Kapitola 4

Metody detekce

Prvním krokem práce bylo zjistit, jaké existují možnosti detekce kolizí, které by bylo možné využít v aplikaci IREView. V průběhu práce byly doplněny další možnosti, z čehož nakonec vyplynul následující seznam možných řešení:

- Úprava původního skriptu
- Využití Boolean operací
- Detekce kolizí pomocí Blender Game Engine
- Detekce založená na simulaci Rigid Body

4.1 Úprava původního skriptu

Původní návrh řešení počítal s tím, že se pro detekci kolizí mezi zářiči a okolními objekty využije upravená verze původního skriptu, vytvořeného v dřívějším vývoji aplikace. Ten byl navržený pro detekci kolizí mezi zářiči navzájem.

Zářič je v prostředí IREView reprezentován geometrickým objektem, který s určitou přesností kopíruje jeho reálný vzhled, což znamená, že se jedná o objekt poměrně složitý a geometricky členitý, viz. Obrázek 3.7.

4.1.1 Původní skript

Výše zmíněný skript byl založen na detekci pomocí SAT (Separating Axis Theorem), který funguje pouze s konvexními objekty (Teorie k SAT se nachází v kapitole 3.4). Jak

bylo zmíněno dříve, jeden z často užívaných konvexních tvarů je kvádr. Podle vzhledu zářiče z obrázku 3.7 je patrné, že při velkém zjednodušení bude kvádr vhodná aproximace, proto byl skript navržen tak, aby pracoval právě s tímto tvarem. V teoretické části bylo zmíněno, že zářiče byly pro potřebu detekce rozděleny na jednotlivé součástky, což umožňuje její zpřesnění. Samotné části je opět možné zjednodušit na kvádry, které pak skript používá pro detekci.

Pro tvorbu obálky je použit tzv. „*Bound Box*“ jednotlivých objektů. „*Bound Box*“ je pomyslná obálka, kterou má každý objekt v prostředí Blenderu. Slouží především v případech, kdy je ve scéně příliš mnoho objektů a uživatel nepotřebuje vykreslovat jejich detailní tvar, pouze chce vidět jejich pozici a rozměr. Obálka je vždy tvaru kvádru a její rozměry kopírují minimální objem objektu, který „obaluje“. Jedná se tedy o OABB (Vysvětleno v kapitole 3.3).

Pokud by byl celý zářič reprezentovaný jedním kvádrem, docházelo by k velkým nepřesnostem při detekci. To by mělo za následek omezení prostoru, kam lze zářiče umístit, neboli by byl omezován primární účel aplikace, kterým je rovnoměrné ozáření formy. Rozdělením na části tedy došlo k částečné eliminaci tohoto problému.

Jelikož se jeden zářič skládá z relativně velkého počtu částí, je dobré provést detekci ve dvou krocích. V prvním kroku je z *Bound Boxů* jednotlivých částí vytvořen nový objekt, jehož obálka poslouží jako testovaný kvádr prvního stupně detekce. Skript si postupně bere dvojice objektů, které takto obalí a provede detekci.

V případě tohoto skriptu došlo k implementaci SAT druhým způsobem (vysvětleno v kapitole 3.4). V původním návrhu skriptu bylo totiž zamýšlené rozšíření kolizních obálek z kvádrů na složitější konvexní tvar *(Convex Hull)*, při kterém by tato implementace fungovala rychleji. Jak bylo v popisu vysvětleno, detekce je založena na zjištění, zda leží body (vertexy) zkoumaných objektů na opačné straně oddělující linie (plochy). To lze zjistit pomocí vektorové projekce.

Funkce ve smyčce vezme možné oddělující plochy odvozené z prvního objektu (objekt A) a zjistí jejich normálový vektor, toho lze dosáhnout pomocí vektorového součinu, který je definovaný například takto [19]:

$$c_1 = a_2 b_3 + a_3 b_2$$
$$c_2 = a_3 b_1 + a_1 b_3$$
$$c_3 = a_1 b_2 + a_2 b_1$$

následně skript vezme ve vnořené smyčce vertex (v_i) druhého z objektů (objekt B) a sestaví vektor **d** tak, že $\mathbf{d} = (v_i, a)$. Projekcí vektoru **d** na vektor **c** lze zjistit, zda je vertex v_i před nebo za zkoumanou plochou. Projekce vektorů je jejich skalární součin, ten je definovaný jako [19]:

$$x = c_1 d_1 + c_2 d_2 + c_3 d_3 = |c||d| \cos \alpha$$

Z toho plyne, že pokud jsou na sebe vektory kolmé, skalární součin je roven 0, pokud je úhel α ostrý, výsledek je kladné číslo a naopak. Tím lze jednoznačně prokázat, zda leží v_i před nebo za danou plochou. Pokud některá plocha vykáže oddělení objektů, je jisté, že kolize nenastala. Pokud žádná z ploch není plochou oddělující, je stejný postup uplatněn na plochy objektu B a případně i na třetí skupinu možných dělících ploch vzniklou z hran (vysvětleno v kapitole 3.4). S nálezem první vhodné plochy je detekce ukončena a nejedná se o kolizi. Pokud jsou prozkoumány všechny zvolené separující plochy a žádná z nich není oddělující, potom se objekty protínají.

Pokud je nahlášena kolize, přejde skript k detekci na úrovni jednotlivých částí zářiče, který probíhá stejným způsobem. Pokud i v tomto případě dojde k pozitivnímu nálezu, pak teprve je pozice zářiče považována za kolizní. V případě, že jsou objekty daleko od sebe, dojde pouze ke kontrole prvního stupně, která vrátí negativní výsledek a není třeba provádět detailnější zkoumání. To může ušetřit množství zbytečných výpočtů. Detekce není zcela přesná, ale pro účely kontroly mezi zářiči byla schválena jako dostatečná.

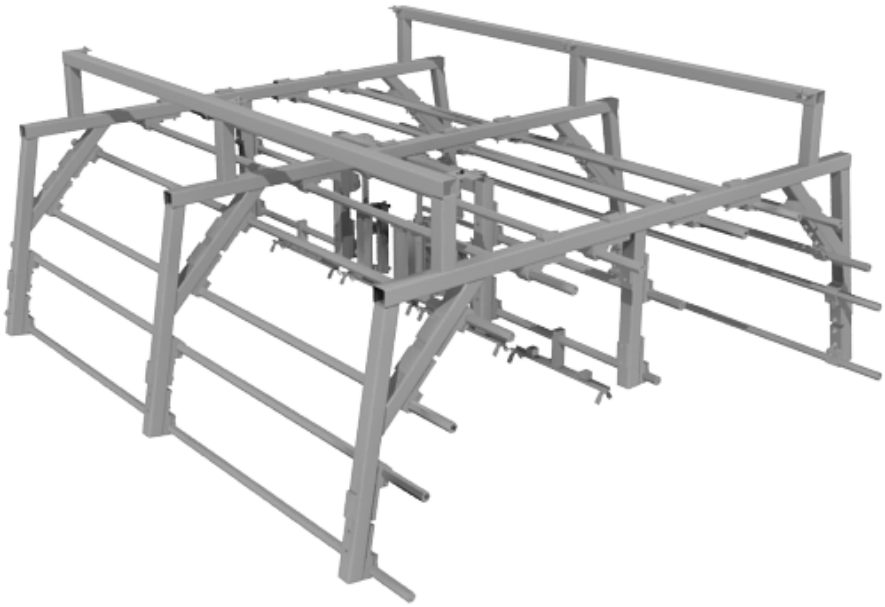
4.1.2 Upravená verze

V požadavcích na detekci kolizí mezi objekty zářičů a okolím bylo zmíněno, že by bylo vhodné pracovat s okolními objekty v celku (nikoliv je dělit na části jak tomu bylo u zářičů). Ale například rámová konstrukce, se skládá z velkého počtu částí, viz. Obrázek4.1. Ty jsou tvarově i rozměrově značně odlišné. Proto by byla detekce s použitím jednoho velkého kvádru příliš nepřesná.

Geometrie objektu v prostředí Blender se skládá ze tří hlavních součástí. Vertex, Edge a Polygon (Face)[1].

Vertex je bod, který je s dalším bodem objektu propojen pomocí hrany (edge). Tři a více bodů pak mohou tvořit plochu (polygon). Blender umožňuje uživateli s těmito komponentami pracovat. Jedna z funkcí, kterou rozhraní nabízí je možnost vybrat všechny vertexy, hrany či plochy, které jsou propojeny navzájem, čehož skript využívá.

[1] v Blender 2.63 došlo ke změně z face na polygon



Obrázek 4.1: Ukázka rámové konstrukce s několika zářiči

Původní skript je rozšířen o funkci, která vybírá jednotlivé části objektu a podle jejich rozměrů pak tvoří jim odpovídající skupinu hraničních kvádrů. Skript funguje obdobně, jako jeho předchozí verze do doby, kdy dojde k nalezení kolize prvního řádu (celý objekt je reprezentován jedním kvádrem). V danou chvíli se zavolá nová funkce, která bude popsána v následujícím textu.

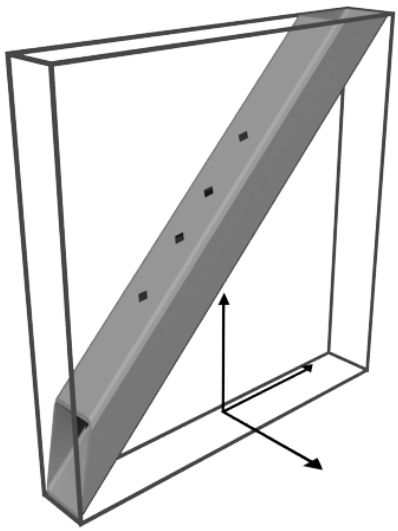
Funkce nejprve inicializuje seznam, který bude sloužit jako úložiště pro jednotlivé části objektu. Tento seznam má následující strukturu: *Parts* = {*part_index:*(v_1,v_2,v_3...v_n), *part_index+1:*(v_{n+1},v_{n+2},v_{n+3},...v_{n+m}),... *part_index+i:*(v_j,v_{j+1},v_{j+2},...v_{j+k})}

Dále je inicializováno pole, (pro další užití v textu bude označené jako pole A), do kterého se uloží indexy jednotlivých vertexů daného objektu. Vše se provede za pomocí smyčky *For*, v jejím průběhu se také zajistí, aby vertexy nebyly označeny jako *Selected*, tedy vybrané, což by narušovalo správný průběh tvorby obálek.

V další smyčce je třeba projít jednotlivé záznamy ve výše vytvořeném poli a rozdělit je následovně. Pomocí indexu v poli A je vybrán vertex nesoucí tento index. Díky vestavěné funkce Blenderu *select_linked*, pak dojde k vybrání všech další vertexů, do kterých se lze pomocí hran dostat z právě zpracovávaného vrcholu. Indexy takto vybraných vertexů jsou jednak uloženy v novém poli (pole B), ale zároveň je jejich záznam odstraněn z pole A. Důvod je zřejmý, část objektu která je složena z daných vrcholů, je již označena a tudíž jsou nadále nezajímavé. V tuto chvíli přichází na řadu výše uvedený seznam. V něm je zaveden nový záznam. Klíčem záznamu je číslo označující část objektu (číslování není nezbytně nutné, sloužilo pro lepší přehlednost při tvorbě funkce) a samotný záznam je vzniklé pole B. Na konci smyčky dojde k inkrementaci proměnné určující označení částí, aby bylo zajištěno jasné rozlišení. Celý cyklus se opakuje, dokud není dosaženo konce pole A.

Nyní se funkce nachází ve stavu, kdy má části objektu rozděleny v seznamu a je třeba vytvořit jejich obálky. Jelikož se nejedná o jednotlivé objekty, není možné využít *Bound boxy* jako tomu bylo v případě původní verze. Je zapotřebí projít jednotlivé části postupně a zjistit maximální respektive minimální souřadnice vrcholů v jednotlivých osách. Výsledkem je šest hodnot, které ve vhodných kombinacích tvoří osm vrcholů kvádru. Funkce se provede pro oba objekty, které jsou v danou chvíli podrobovány detekci. Výsledkem jsou dva seznamy obsahující jednotlivé kolizní kvádry, jež jsou opět postupně podrobeny detekujícímu algoritmu. V případě nalezení kolize je pak za kolizní považován celý objekt.

Nevýhodou řešení je problém zarovnání obálek s objektem, kterého jsou součástí. Vytvořená obálka je zarovnána s osami, nikoli s částmi objektu (obdobný princip jako AABB,



Obrázek 4.2: Problém vzniklé obálky

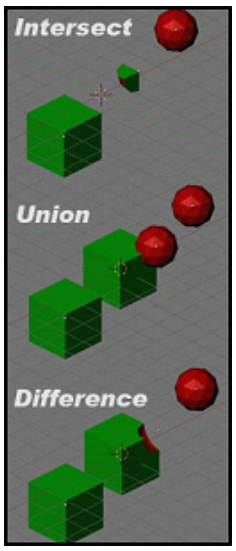
vysvětleno v kapitole 3.3). To velmi snižuje přesnost výpočtu, která je zde důležitá. Snaha o zarovnání obálky s reálným tvarem části byla zamítnuta kvůli přílišné výpočetní náročnosti.

Jedním z pokusů, jak umožnit detekci s okolím pomocí tohoto skriptu byla možnost, importovat objekty rámové konstrukce jednotlivě, jak tomu bylo u zářičů. Objekty jsou do prostředí Blender importovány tak, že jejich *pivot* (bod který určuje jejich pozici a natočení) je zarovnán s osovým systémem 3D prostoru a je umístěn v bodě (0,0,0). *Bound Boxy* objektů jsou orientovány právě podle natočení jejich *pivotu*, tudíž jsou také zarovnány s osami. To má za následek stejný výsledek jako předchozí funkce (viz. Obrázek 4.2). A opět se jedná o nevyhovující řešení.

Funkce tedy přinesla odstranění nutnosti dělit zářiče na jednotlivé součástky ručně, pro detekci s okolím (především rámem) je však daná metoda nevyhovující. To je hlavní důvod zkoumání dalšího možného řešení. Druhý důvod je fakt, že ohřívaná forma, se kterou je třeba také provést kontrolu kolizí, je jeden velký odlitek. Ten by pomocí daného skriptu bylo možné zjednodušit pouze na jeden kvádr, čímž by nedošlo k dostatečně přesné detekci.

4.2 Využití boolean operací

Tento, ne zcela standardní postup pro detekci kolizí je založen na modelovacích nástrojích a možnostech softwaru Blender. Princip metody spočívá v tom, že je na zkoumaných dvojicích objektů provedena jedna z boolean operací (ukázka boolean operací ve 3D na Obrázku 4.3), konkrétně difference (rozdíl) a pokud jsou objekty v kolizi, dojde ke změně jejich geometrických dat. Pokud nesdílí žádný společný prostor a tedy nejsou v kolizi, nedojde k interakci a data to nepoznamená.



Obrázek 4.3: Boolean operace v 3D prostoru

Jeho výhodou oproti předchozím řešením je univerzálnost použití, tuto operaci je možné provést prakticky na jakkoli složitý objekt. Dále také přesnost provedení, která pracuje pouze s geometrií objektu, tudíž je stoprocentní. Avšak velkým problém je rychlost. I v případě, že nedojde ke kolizi, je časová náročnost této operace, v porovnání s matematickými výpočty standardních metod, příliš velká. Pokud ke kolizi dojde, čas potřebný pro výpočet je ještě vyšší. To je způsobeno počtem výpočtů nově vzniklých vertexů, hran a plošek. Dále je také čas ovlivněn hloubkou prolnutí a velikostí ovlivněných dat. Čím více se data změní, tím déle výpočet trvá.

Jak z Tabulky 4.1 vyplývá, jedná se o velice pomalou metodu a čas potřebný k jejímu vykonání je značně nestabilní. Dalším problémem, který je s ní spojen je fakt, že při práci s modely které mají složitou geometrickou strukturu je často nestabilní a způsobuje pád celé aplikace. I když je to metoda přesná, z očividných důvodů došlo k jejímu zamítnutí.

druh	čas
krychle/krychle	0.14s
krychle/rám	4.34s
zářič/rám	2m 14s
zářič/zářič	6m 28s

Tabulka 4.1: Rychlost detekce pomocí boolean funkce

4.3 Detekce kolizí pomocí Blender Game Engine

Jak bylo zmíněno, softwarové prostředí Blender disponuje vlastním herním enginem. Jeho stručný popis se nachází v kapitole 3.7. Ten obsahuje také fyzikální engine Bullet. Jakožto fyzikální engine, musí být schopen detekovat a zpracovávat kolize, čehož bylo využito v následujícím řešení dané problematiky. Jelikož je nutné kontrolovat kolize mezi velice složitými objekty, je výhodné provádět detekce typu. „Mesh To Mesh“. Jedná se o přímou kontrolu kolize mezi geometrickými reprezentacemi objektů. I když se tím snižuje rychlost výpočtu na úkor přesnosti, v tomto případě je přesnost podmínkou, proto není pomalejší detekce překážkou. Oproti předchozímu řešení navíc není dopad na rychlost tak markantní. Řešení je založeno na možnosti nastavit ve scéně objekty tak, aby v herním enginu fungovaly jako detektory kolizí (tj. zaznamenávají průnik s ostatními objekty). Níže popsaná metoda slouží především ke kontrole kolizí mezi zářiči a okolím.

Před samotným vykonáním detekce je nutné vybrat objekty, u kterých má být detekce provedena. To je zapotřebí ve všech skriptech, popisované řešení má však jednu podmínku. Posledním vybraným objektem musí být objekt, se kterým se bude kolize kontrolovat (rám, forma).

Samotný proces detekce se pak skládá ze dvou fází. Prvním krokem je inicializace objektů. Jelikož se jedná o herní engine, je třeba objektům přiřadit herní logiku, aby detekce proběhla vhodným způsobem. Fyzikální engine Bullet, tedy i herní engine Blenderu, podporuje několik typů nahlížení na objekt z hlediska fyziky (Navigation Mesh, Sensor, Occlude, Soft Body, Rigid Body, Dynamics, Static, No Collision).

Pro účely detekce kolizí objektů, které se nepohybují, bylo využito typů *Static, Sensor a No Collision*. Jak již z názvů vyplývá, první typ je statický objekt, druhý je typ senzor, který je taktéž statický, ale je schopen zaznamenávat kontakt s jinými typy objektů (kromě dalších objektů stejného typu). Poslední případ je typ, který se do kolizí nezahrnuje, což je opět patrné z jeho názvu.

4.3.1 Inicializace

Inicializační část je v podstatě to, co vykonává vytvořený skript. Objektům je nutné nastavit kolizní obálku. Herní engine Blenderu umožňuje pro účely řešení kolizí použít několik typů reprezentací. Těmi jsou Capsula, Box, Sphere, Cone, Cylinder, Convex Hull, Triangular Mesh.

Jak bylo zmíněno výše, potřeba přesnosti převažuje potřebu rychlosti, proto byl zvolen typ *Triangular Mesh*. Nejedná se o zjednudušující obálku jako u ostatních možností, jedná se o přesnou kopii objektu, tudíž je kontrola prováděna s nejvyšší možnou přesností.

Dalším krokem v inicializaci řešení je nastavit všechny objekty implicitně do stavu *No Collision*. Tím se zajistí, že nebude docházet k detekci v oblastech, které uživatele v danou chvíli nezajímají. Následně je vybraným zářičům přiřazen typ *Sensor*. Tím se zajistí, že budou schopny detekovat kolize s jinými objekty. Objektům je také vytvořena proměnná *Collision*, která je typu *boolean*, její využití budu objasněno v sekci 4.3.2. Posledním vybraným objektem je rám (případně jiný objekt, jež uživatele zajímá), což je cílový objekt detekce. Tomu je přiřazen typ *Static*, který je možné zachytit pomocí senzorů a zároveň se jedná o typ, jež je nehybný.

Pokud se jedná o první inicializaci, je ve scéně vytvořen nový objekt, tzv. „Collision Handler“. Tento objekt je neviditelný a slouží pouze pro zpracování výsledků detekce. Naskytne se otázka, proč zpracování neprovádí sám Senzor? Důvodem je fakt, že se může tento Senzor měnit a bylo by nutné zároveň s ním měnit i jeho herní logiku.

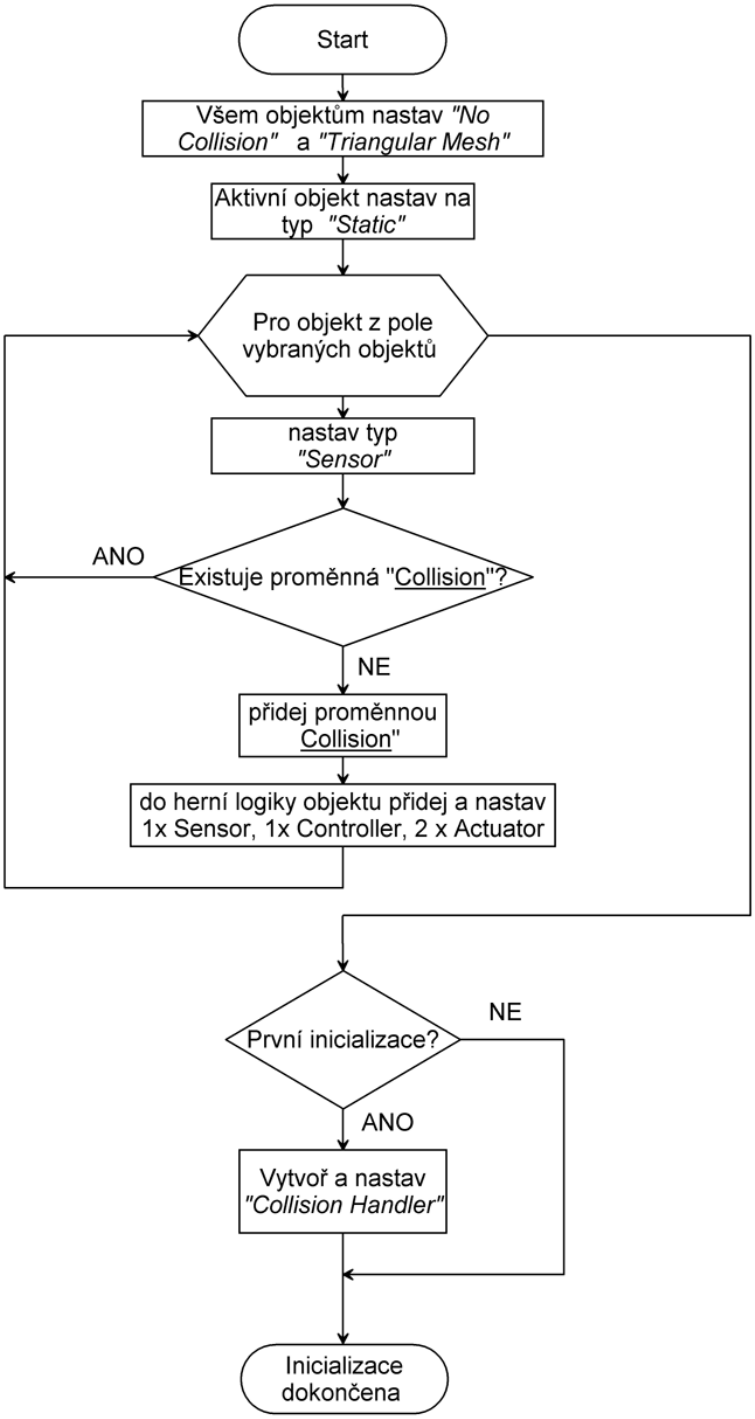
Pro přehlednější a snažší pochopení procesu inicializace je na Obrázku 4.4 vývojový diagram celé této části.

4.3.2 Herní logika a postup

Předem by bylo dobré si upřesnit označení senzor, jelikož v následujících odstavcích bude takto označen jednak objekt, ale také část herní logiky. Pro zjednodušení bude „senzor“ označení v části logiky a „Senzor“ (s velkým S) označení pro objekt s přiděleným fyzickým typem *Sensor* z předchozí kapitoly.

Herní logika je nedílnou součástí herního enginu. Jedná se o způsoby chování objektů ve virtuálním světě. Jednak je zde možné nastavit, na jaké podněty má objekt reagovat, ale hlavně, jak na ně má reagovat, přičemž každý objekt má svojí logiku.

Samotné nastavení se skládá ze tří částí. Zaprvé má objekt přidělený senzor, který reaguje na různé podněty. Další blokem je zpracování podnětu pomocí kontrolerů. V této



Obrázek 4.4: Vývojový diagram inicializace

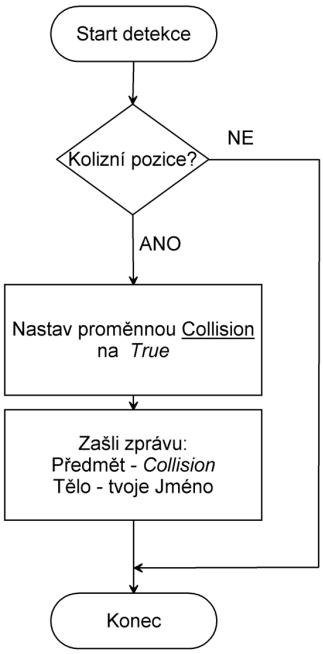
části je možné použít logické operátory pro smíšení více senzorů, případně Python skript, který může podněty upravit, použít pro své účely, a podobně. Posledním článkem logiky jsou bloky, které reagují na získané informace. Opět je zde množství různých nastavení, jak se má objekt zachovat.

V případě typu *Static* není třeba nastavovat žádnou logiku, objekt slouží pouze jako „překážka“, kterou je možné detekovat při kolizi.

Co se týče *Sensorů*, neboli kontrolovaných zářičů, je třeba logiku zavést. První částí je tedy senzor, který reaguje na podnět kolize s dalšími objekty. Pokud se daný předmět nenachází v kolizním stavu, zůstane senzor pasivní a neposílá žádné impulsy. V opačném případě se senzor aktivuje a přes spojení předá informaci o kolizi kontroleru. Ten se signálem nemusí nic dělat, pouze ho pošle dál. V posledním bloku jsou dva reakční články. Jeden nastaví proměnnou *Collision* do stavu *True*. Druhý článek slouží k zaslání zprávy ostatním účastníkům ve scéně. Zpráva oznamuje, že došlo ke kolizi u objektu, které jí vyslal.

Co se týče proměnné *Collision* je zde pouze jako podpůrná funkce, její využití není příliš časté. Herní engine dovoluje při spuštění „hry“ vypisovat proměnné a jejich hodnotu na obrazovku. Což by umožnilo uživateli okamžitou vizuální informaci o stavu jednotlivých testovaných objektů. Bohužel při větším množství kontrolovaných objektů je tento způsob výpisu nepřehledný a proto je použito rozeslání zprávy ostatním účastníkům simulace. Její přítomnost je také kontrolována v procesu přiřazování logických cihel jednotlivým objektům (pokud jí objekt má, není třeba znovu nastavovat logiku).

Jak bylo uvedeno o dva odstavce výše, druhý článek, který reaguje na podněty, zašle všem objektům ve scéně zprávu. Ta se skládá ze dvou částí. Zpráva má předmět (jméno) a také tělo. Předmět zprávy nese označení, na které můžou reagovat další senzory. Zde je jako předmět uvedeno *collision*. V těle zprávy je pak jméno objektu, který jí zaslal. Jediným objektem, který ve scéně zpracovává zprávy je výše zmiňovaný „Collision Handler“, ten zachycuje zprávy právě s předmětem *collision*. (Bylo by možné zpracovávat všechny zprávy bez rozdílu, jelikož nedochází k vysílání žádných dalších, upřesnění je zde jen pro případ, že by v budoucnu byla logika rozšířena o další zprávy.) „Collision Handler“ pak těla zpráv pomocí Python skriptu vypíše jednak do konzole, tak i do souboru. Po zapsání všech detekovaných kolizí se herní engine automaticky ukončí. K automatickému ukončení dojde i v případě, že se žádná kolize nevyskytne.



Obrázek 4.5: Detekce z pohledu Senzoru

Druhou částí řešení je pak samotná detekce, která nedělá nic jiného, než že spouští zabudovaný herní engine.

Řešení má jednu drobnou vadu. Tou je fakt, že je při kontrole nutno spustit „hru“, což se projeví vykreslením herního okna. Vykreslení samo je pro uživatele IREView velice rušivé a proto bylo třeba vymyslet úpravu, která by tento problém odstranila. Tj. aby jej uživatel vnímal pouze jako fyzikální engine, bez viditelných grafických projevů.

4.3.3 Úprava herního enginu

Dosáhnout tohoto cíle se podařilo pomocí úpravy zdrojového kódu Blender tak, aby příkaz vykreslení nebyl vůbec zavolán. To obnášelo vlastní kompilaci Blenderu, jež byla díky podrobnému návodu na internetových stránkách zabývajících se vývojem Blender úspěšně odzkoušena. Bylo tedy třeba prozkoumat zdrojové soubory a zjisti, kde se nachází příslušný příkaz nebo blok příkazů a jak komplikované bude jeho odstranění.

Za asistence jednoho člena vývojové komunity bylo patřičného výsledku dosaženo. To jest, při spuštění herního enginu v prostředí Blender, nedojde k vykreslení dané scény. Běží tedy „pouze“ jako fyzikální engine, což zajišťuje funkční detekci kolizí, bez zbytečného

Obrázek 4.6: Detekce z pohledu "Handleru"

rozptylování uživatele grafickými změnami prostředí.

4.3.4 Speciální vlastnosti řešení pro IREView

Vytvořené řešení je určeno k obecné detekci kolizí objektů v poměru 1:N. V rámci implementace pro použití v IREView bylo nutné udělat pár změn tak, aby detekce probíhala vhodným způsobem v již existujících scénách.

Jak bylo naznačeno v kapitole 3.5, IRE jsou rozděleny na více objektů, jeden z nich slouží k ovládání celého zářiče a všem podřízeným je odebrána schopnost být uživatelem vybrán. Úprava spočívá v malé funkci, která po vybrání rodičovského objektu označí jako vybrané i jeho potomky. Tím je zaručena přítomnost celého zářiče při detekci a ne pouze vybrané části.

Druhou editací metody bylo naopak vynechání určitých částí zářiče pro účely detekce. Jednou z těchto částí je bod (reprezentovaný malou koulí), který slouží pouze jako ukazatel ideální vzdálenosti od ozařované plochy. Je patrné, že tento bod je v reálném světě pouze imaginární a jeho reprezentační objekt tedy slouží pouze pro snažší orientaci uživatele. Proto je u něj nevhodné zkoumat, zdali je v kolizi. Druhou částí zářiče, kterou

je třeba z detekce vypustit, je objekt nesoucí jeho označení. Ten slouží uživateli pouze k ulehčení orientace při větším množství IRE ve scéně.

4.4 Detekce pomocí simulace Rigid Body

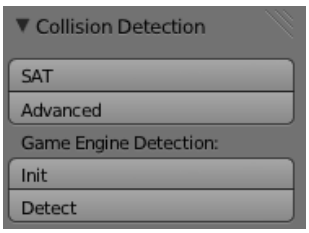
S vydáním verze 2.66 dostal uživatel Blenderu možnost pracovat s tuhými tělesy (rigid body) přímo, bez nutnosti spouštět Game Engine. Konkrétně možnost simulovat jejich chování. Díky této vlastnosti bylo možné vytvořit další metodu detekce kolizí.

Metoda je založena právě na simulaci tuhých těles. Jejich chování je řízeno stejně jako v předchozím řešení pomocí fyzikálního enginu Bullet Physics. Jeho krátký popis a popis simulace Rigid Body je v kapitole 3.8. Hlavní myšlenkou právě popisované metody je změna pozic objektů v kolizním stavu.

Byl opět naprogramován skript, který detekci provádí. Po jeho zavolání se nejprve provede inicializace účastnících se objektů a fyzikálního světa. Je důležité synchronizovat počet volání simulační smyčky, SPS(steps per second) s FPS tak, aby byla detekce v jednom kroku (snímku) provedena pouze jednou. Důvod bude objasněn o odstavec níže. Dále je třeba vypnout gravitační sílu, která je v Blenderu implicitně zapnutá, opět bude důvod vypnutí vysvětlen níže. Samotným objektům je třeba přiřadit *Rigid Body* fyzický model, který zajistí jejich reprezentaci v simulačním procesu. Jako tomu bylo v předchozím případě, i zde je třeba nastavit vhodnou kolizní obálku. Jelikož je přesnost prioritou, je zvolen typ *Mesh*, který bere geometrii objektu bez zjednodušení. Posledním krokem inicializační části je vytvoření seznamu, který nese pozice jednotlivých objektů v danou chvíli. Ty budou využity v dalším průběhu pro porovnávání.

Nyní je nutné provést jeden krok animace. Tím, že je FPS a SPS stejné je zaručen pouze jeden test detekce. V tomto kroku Bullet prověří kolize a vypočítá síly působící na objekty. Ty jsou následně aplikovány a objekty v případě kolize změní pozici či natočení. Standardně je SPS 60 a FPS 24, to znamená, že by proběhla detekce dvakrát během jednoho kroku. To je jednak zbytečné, ale také by mohlo dojít k falešným nálezům (po první změně pozice by mohl objekt kolidovat s jiným).

V tuto chvíli tedy skript znovu projde pozice jednotlivých objektů a porovná je s hodnotami uloženými před simulací. Pokud došlo ke změně, je objekt jistě v kolizi, v opačném případě porovnání vyjde pozitivně a objekt se v problémové pozici nevyskytuje. Právě kvůli porovnání pozic bylo nutné vypnout gravitační sílu, ta by všechna tělesa při kroku



Obrázek 4.7: Tlačítka pro detekci

simulace posunula a tudíž by porovnávání vždy vyšlo negativní.

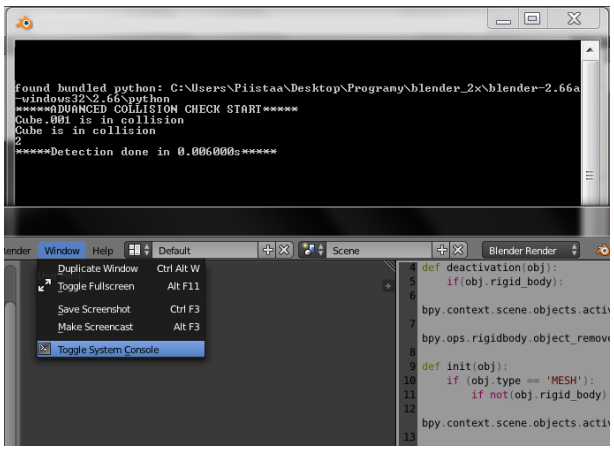
Na konci skriptu je simulace vrácena o krok zpět, čímž se objekty dostanou zpět do pozic, na které je uživatel umístil. Tím je zajištěna stejná výchozí pozice pro další test. Následně je objektům odebrána fyzická reprezentace pro případ, že by v příští detekci nebyla jejich účast třeba. Dalším důvodem odebrání je fakt, že Blender označuje tělesa s přiřazeným *Rigid Body* jiným způsobem než ostatní objekty ve scéně, což na uživatele aplikace působí rušivě.

4.5 Ovládání skriptů

Pro správné používání skriptů je třeba vysvětlit, jaký je postup při detekci z pohledu uživatele. Všechna řešení se uživateli graficky projevuje pouze jako sada tlačítek v postranní liště hlavního okna IREView (Blender).

Každá z detekčních metod má vlastní tlačítko. Detekce s pomocí herního enginu je pak ovládána dvěma tlačítky (viz. Obrázek 4.7). Jedno z nich slouží pro inicializaci, tedy nastavení herní logiky, druhé pak ke spuštění samotného testu.

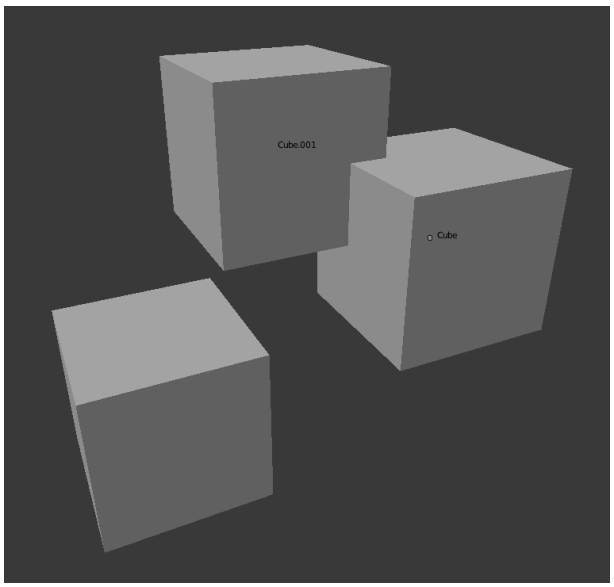
První podmínka je samozřejmě mít scénu, ve které je několik objektů, mezi kterými je zájem detekci provést. Dalším krokem je vhodné vybrání požadovaných objektů. Uživatel musí vybrat minimálně dva objekty, jinak nemá smysl detekci provádět. Všechny skripty mají vstupní podmínku, které tuto situaci ošetřuje.



Obrázek 4.8: Konzole s výpisem, tlačítko pro její zpřístupnění

U detekce prováděné pomocí herního enginu, je výběr složitější. Jak bylo vysvětleno výše, jedná se o detekci *1:N*, tedy jeden objekt vůči N dalším. Nejprve je nutné vybrat právě N objektů, které budou podléhat detekci a následně vybrat objekt, se kterým bude ona detekce provedena. Ve skutečnosti je třeba aby byl objekt označen jako aktivní, čehož lze v Blenderu docílit více způsoby. Nejjednodušší z nichž je právě vybrání daného objektu. Blender automaticky poslední vybraný označí jako aktivní. V případě, že je objekt aktivní, lze k němu přistupovat jinak než k ostatním, čehož skript využívá.

Následně je možné provést samotné testování stisknutím příslušného tlačítka funkce. Pro metod s herním enginem opět platí odlišný postup. Stiskem tlačítka *Init* (inicializace) se provede první část metody, tedy přiřazení herní logiky a vytvoření objektu, který zajišťuje zpracování výsledků. Následně je možné tlačítkem *Detect* (detekce) spustit test, není již nezbytné mít objekty vybrané. Pokud dojde pouze ke změně pozic objektů, lze další detekci provádět bez nutnosti znovu inicializovat scénu.



Obrázek 4.9: Označení kolizních objektů

Po provedení detekce kterýmkoliv způsobem jsou jména kolizních objektů vypsána do konzole Blenderu. Ta je implicitně skryta, je jí možné zobrazit pomocí tlačítka *Toggle System Console*. Druhým, pro uživatele užitečnějším způsobem oznámení výsledků, je zobrazení jména objektu, který je v kolizi, viz. Obrázek 4.9. Tímto způsobem má uživatel přímo přehled o tom, které zářiče je nutno přemístit, bez toho, aby je musel vyhledávat podle jména.

Kapitola 5

Testování a měření

V rámci práce byly také otestovány jednotlivé detekčních možnosti. Testy se týkaly jak rychlosti, tak přesnosti detekce. Aby bylo možné srovnat výsledky, bylo třeba upravit skript detekující kolize pomocí herního enginu, tak aby fungoval na principu každý s každým, jako tomu je u ostatních metod. To bohužel způsobilo náhodné nepřesnosti v detekci. Jelikož je ale původní účel skriptu test *1:N*, nebyl důvod tento problém odstraňovat.

K testování byl použit počítač s následujícími parametry:

- Intel Core i5 3570K @ 3.40GHz
- 8GB RAM
- NVidia GTX 580
- Windows 7 Ultimate 64bit

Před samotným testováním byly stanoveny předpoklady, které vycházejí z teoretické znalosti fungování naprogramovaných metod. Testy posloužily k jejich ověření.

Předpoklady: [1]

- Detekce každý s každým má náročnost $O(N^2)$
- PS není závislí na počtu kolizí
- PS a UPS mají v bezkolizním prostředí totožnou rychlost

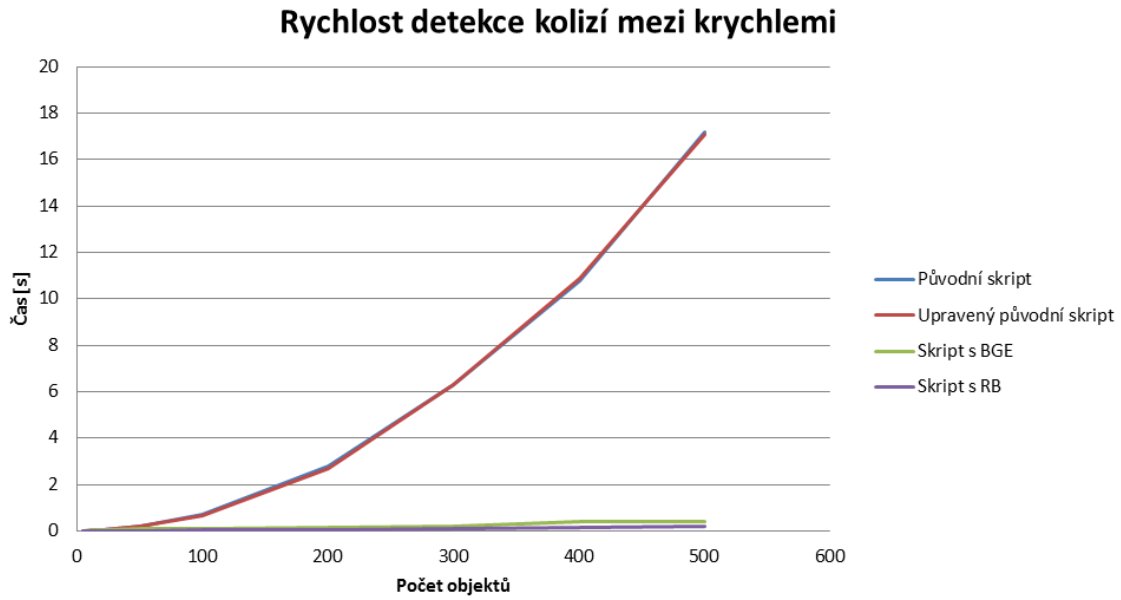
[1] PS - původní skript, UPS - upravený původní skript, BGE - skript využívající herní engine, RB - skript detekující pomocí simulace tuhých těles

- PS a UPS nejsou v bezkolizním prostředí ovlivněny složitostí geometrie
- RB je při detekci nejrychlejší
- RB a BGE jsou rychlostně na stejné úrovni
- RB a BGE jsou nejpřesnější

5.1 Testy rychlosti

Testování rychlosti bylo provedeno ve dvou scénách, první obsahuje krychle, druhá pak modely zářičů (zářiče nebyly rozděleny po částech na samostatné objekty). Jejich počet po každém testu stoupl a byla jim přiřazena pozice tak, aby nedocházelo ke kolizím. Tím bylo možné ověřit rychlost naprogramovaných algoritmů bez ovlivnění výsledků počtem kolizí. Testy byly prováděny opakovaně, aby došlo k eliminaci náhodných chyb.

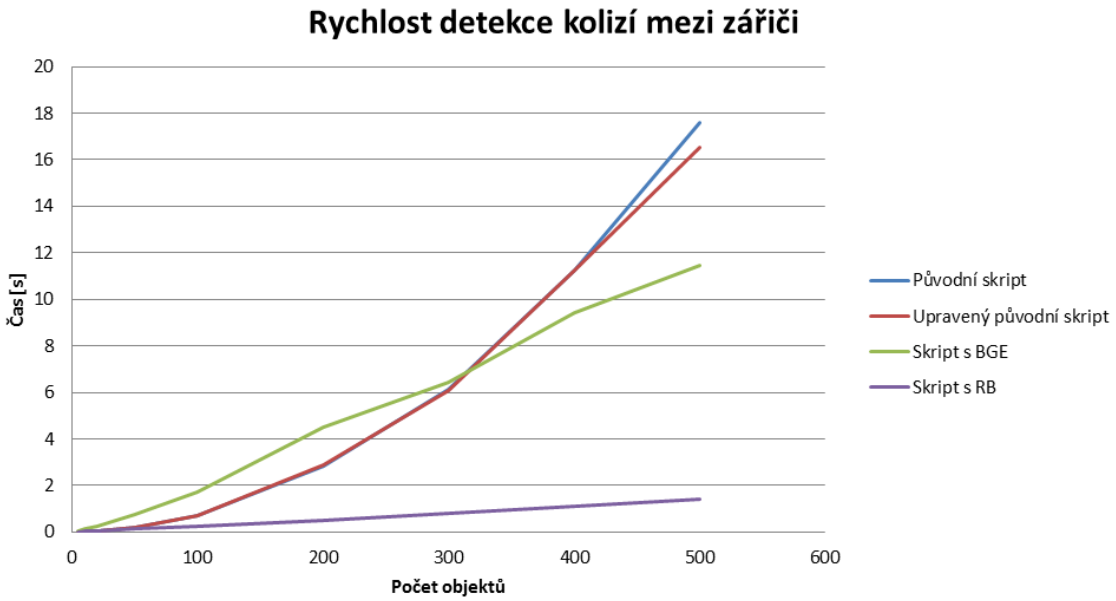
Jak je z grafů 5.1 a 5.2 patrné, poslední skript využívající simulaci tuhých těles byl při větším množství testovaných objektů rychlejší než ostatní varianty. V případě velkého množství objektů se složitou geometrií byl pak rozdíl rychlostí oproti ostatním metodám výrazný. Při menším počtu objektů byly naměřené rychlosti metod prakticky totožné.



Obrázek 5.1: Graf rychlosti detekce s krychlemi

Z grafů je vidět, že tvar křivky závislosti času na počtu objektů, se u SAT algoritmů přiklání k tvaru kvadratické funkce, což potvrzuje složitost funkce $O(N^2)$. Při porovnání obou grafů, je potvrzen také fakt, že složitost geometrie v bezkolizním prostředí nemá na první dvě metody vliv. Na metody využívající Bullet engine (RB i BGE) má složitost objektů v nekolizním prostředí vliv, což nebylo úplně očekávané. Jedním z důvodů je nutnost zjistit velikosti objektů (v nejširší fázi detekce), aby bylo možné vytvořit AABB, to u složitější geometrie znamená delší hledání nejkrajnějších bodů modelu. U BGE metody je pak třeba inicializovat celý „virtuální svět“, což se velice projevilo do výpočetního času a graf 5.2 vyvrací předpoklad, že budou skripty detekující přes Bullet engine stejně rychlé. Inicializace BGE (tj. čas od spuštění enginu do vykreslení scény) zabral při větším počtu objektů přibližně 70% času. V bezkolizním prostředí pak trvala inicializace přibližně 95% celkového času. Tento problém se negativně projevuje také v případech, kdy je ve scéně mnoho objektů, avšak detekce je prováděna jen na některých z nich.

Z grafů je také patrné, že časová náročnost řešení využívající Bullet Physics je takřka lineární, čímž je dokázána $O(N)$ náročnost AABB Tree v nejširší fázi (popsáno v kapitole 3.8.2). Dalším důvodem rozdílů rychlostí je pak programovací jazyk, ve kterém se provádí výpočty. Jak bylo zmíněno v teoretické části, Python je interpretovaný jazyk, a jeho provádění je pomalejší než provedení stejného bloku příkazů v překládaných jazycích jako je C nebo C++, ve kterých je naprogramovaný Blender respektive Bullet.



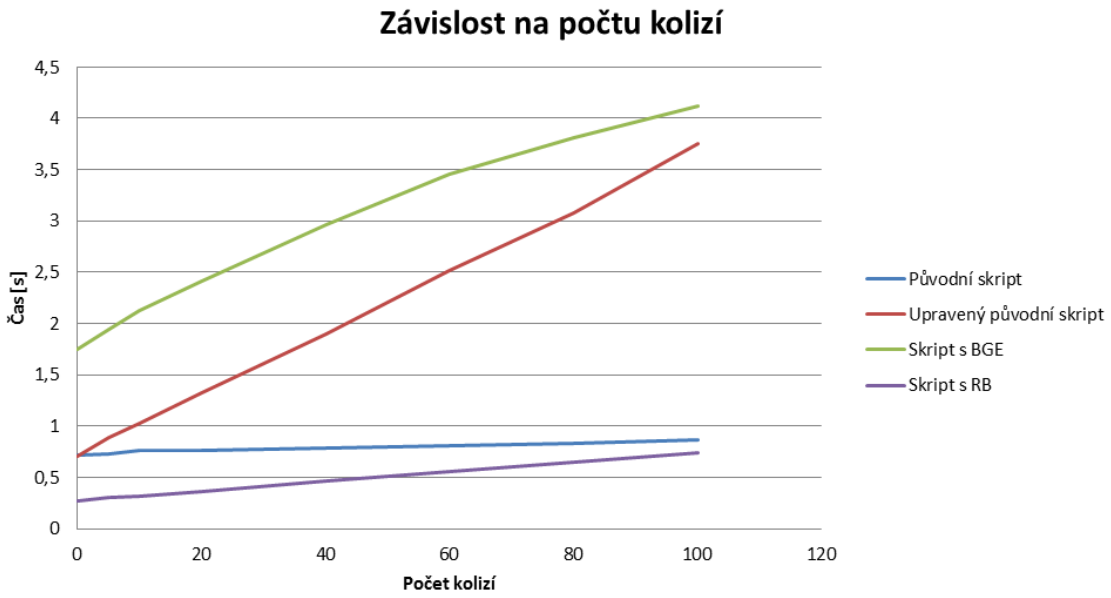
Obrázek 5.2: Graf rychlosti detekce se zářiči

Graf na Obrázku 5.3 vychází z dalšího testování jedné z předchozích scén. Test byl proveden se sto zářičí a postupně byl upravován počet kolizí mezi nimi. Předpoklad, že rychlost původního skriptu je ovlivněna pouze počtem zkoumaných objektů se téměř potvrdil, drobný časový nárůst je způsoben fungováním SAT algoritmu. Tedy, pokud je nalezena projekční plocha, není nutné prohledávat další, což u menšího počtu kolizních objektů dovoluje vynechat část výpočtů. Důvodem předpokladu byla jedna úroveň prohledávání, kde jsou všechny dvojice zkoumány stejným způsobem. Jeho upravená varianta je naopak výrazně závislá na počtu kolizí. Lze pozorovat přímou závislost mezi počtem kolizí a rychlostí výpočtu. Je tomu tak i u metod využívající k detekci Bullet engine. Důvodem jsou urychlující metody, jako je detekce na více úrovních, které dovolují redukovat počet zkoumaných dvojic.

5.2 Testy přesnosti

Pro testování přesnosti byla připravena scéna obsahující dvě stovky zářičů, z nichž se každý skládá z přibližně tisíce polygonů. Pomocí skriptu byly zářiče náhodně rozmístěny do omezeného prostoru[2]. Objektům byl také nastaven náhodný úhel natočení. Následně

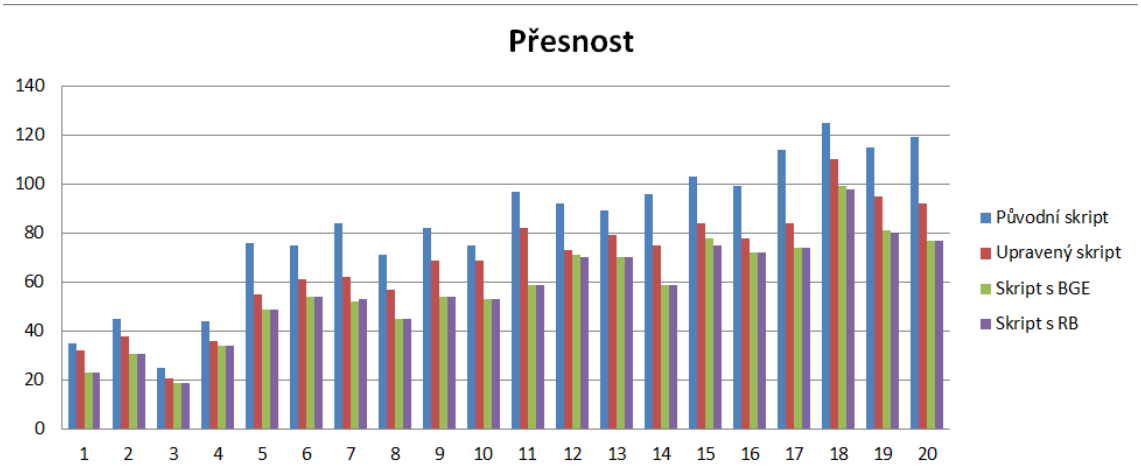
[2]velikost prostoru se měnila aby byla zajištěna širší škála výsledků



Obrázek 5.3: Graf ovlivnění rychlosti počtem kolizí

byla provedena detekce všemi čtyřmi skripty a celý proces se opakoval dvacetkrát.

Z grafu na Obrázku 5.4 je patrné, že poslední způsob detekce vykázal nejmenší výskyt kolizních objektů. (U dvou testovaných scén byly vizuálně ověřeny kolize a výsledky se shodovaly s hodnotami poslední použité metody, proto byly její hodnoty brány jako skutečný výskyt kolizí pro porovnání přesnosti ostatních metod.) U detekce pomocí herního enginu jsou výsledky takřka shodné, ale jak bylo zmíněno na začátku kapitoly, jeho úprava pro tento druh testu zapříčinila drobné nepřesnosti ve výsledcích. Při testování *1:N*, pro které je skript navržen, nedošlo k žádným potížím.



Obrázek 5.4: Počet detekovaných kolizí

Z porovnání vyplývá zřejmé, původní skript byl při testech nejméně přesný, detekoval průměrně o 44% více kolizí, než bylo jejich skutečné množství. Upravená verze původní metody pak detekovala o 17% více kolizí. Způsob využívající BGE byl v tomto testu přesný pouze z 99.6% (Opět nutno zmínit, že pro tento způsob detekce nebyl navržen). Při testech *1:N* se výsledky detekce shodovaly s hodnotami metody založené na simulaci tuhých těles. To je v oblasti přesnosti staví na stejnou úroveň.

Kapitola 6

Závěr

Při zkoumání metod vhodných pro řešení kolizí v aplikaci IREView bylo objeveno případně vytvořeno několik možných řešení. Původní záměr vystavět pokročilou detekci kolizí na stávající metodě byl proveden, tím se zlepšila přesnost detekce v případě, že zářiče nejsou rozděleny na jednotlivé části. Navíc se podařilo eliminovat několik problémů původního řešení a zrychlit její průběh. Pro přesnější detekci s okolím však bylo třeba vytvořit další řešení. Díky vestavěnému enginu Bullet Physics v prostředí Blender došlo k vytvoření dvou dalších možností detekce. Obě řešení jsou přesnější než původní způsob i jeho upravená verze. Ve většině případů pak také rychlejší. Řešení využívající simulaci tuhých těles je nejrychlejší a navíc univerzální, jeho pomocí lze řešit jak detekci mezi zářiči navzájem tak i mezi zářiči a dalšími objekty ve scéně.

Co se týče samotného řešení kolize ve smyslu jejich odstranění, tým vývojářů IREView došel k závěru, že nejlepší možností je manuální úprava pozice zářiče, jak tomu bylo doposud. Jelikož primárním cílem aplikace je vhodné ozáření formy, automatická úprava pozic IRE založená pouze na výsledcích detekce kolizí by znamenala nekontrolovanou změnu ozáření, což je v rozporu s hlavní funkcí IREView.

V plánu vývoje je i možnost automatického rozmisťování zářičů tak, aby došlo k co nejrovnoměrnějšímu osvícení bez nutnosti zářiče umísťovat ručně. V takovém případě je detekce a ošetření kolizí nepostradatelnou funkcí. Avšak jak bylo zmíněno výše, je třeba ošetřit zachování ideálního ozáření formy. To je jedno z vhodných témat pro další vývoj pokročilých metod řešení kolizí v IREView. Bylo by také dobré mít možnost volat přímo detekci kolizí prováděnou enginem Bullet. To by obnášelo značný zásah do zdrojových kódů Blenderu, výsledkem by však byla velice rychlá a přesná metoda detekce.

Zdrojové kódy jednotlivých naprogramovaných skriptů, stejně tak i jejich implementace v Blenderu jsou dostupné na přiloženém CD.

Literatura

[1] BENDER, Jan,et al. *Interactive Simulation of Rigid Body Dynamics in Computer Graphics*. [2012].
dostupné z WWW: `bullet.googlecode.com/files/STAR.pdf`

[2] BITTLE, William. *SAT (Separating Axis Theorem)*. [Leden 2010].
dostupné z WWW: `www.codezealot.org/archives/55`

[3] BITTLE, William. *GJK (Gilbert–Johnson–Keerthi)*. [Duben 2010].
dostupné z WWW: `www.codezealot.org/archives/88`

[4] Blender Documentace. *Introduction to Game Engine*. [Květen 2013].
dostupné z WWW:
`http://wiki.blender.org/index.php/Doc:2.6/Manual/Game_Engine`

[5] BUILDER, informační server o programování. *Jak vyzrát na kolize 2.díl - kolize v prostoru*. [Únor 2002].
dostupné z WWW: `www.builder.cz/rubriky/c/c--/`
`jak-vyzrat-na-kolize-2-dil-kolize-v-prostoru-156031cz`

[6] Dokumentace BULLET Physics. *BtDbvt dynamic aabb tree*. [Srpen 2008].
dostupné z WWW: `bulletphysics.org/mediawiki-1.5.8/index.php/`
`BtDbvt_dynamic_aabb_tree`

[7] Dokumentace BULLET Physics. *Broadphase*. [Únor 2010].
dostupné z WWW:
`www.bulletphysics.org/mediawiki-1.5.8/index.php/Broadphase`

[8] Dokumentace BULLET Physics. *Collision Detection and Physics FAQ*. [Březen 2010].
dostupné z WWW: `bulletphysics.org/mediawiki-1.5.8/index.php/`
`Collision_Detection_and_Physics_FAQ`

[9] Dokumentace BULLET Physics. *Collision Shapes.* [Duben 2013].
dostupné z WWW:
`www.bulletphysics.org/mediawiki-1.5.8/index.php/Collision_Shapes`

[10] COUMANS, Erwin. *Bullet 2.80 Physics SDK Manual.* [2012].
dostupné z WWW:
`bullet.googlecode.com/svn/trunk/Bullet_User_Manual.pdf`

[11] EBERLY, David. *Intersection of Convex Objects: The Method of Separating Axes.* [2008].
dostupné z WWW:
`www.geometrictools.com/Documentation/MethodOfSeparatingAxes.pdf`

[12] Jitter Physics. *Sweep and Prune.* [Říjen 2011].
dostupné z WWW: `jitter-physics.com/wordpress/?p=31`

[13] Kolektiv autorů. Editor MLÝNEK, Jaroslav - POTĚŠIL Antonín.
Ohřev radiací - teorie a průmyslová praxe. Techvická univerzita v Liberci. [červenec 2012].
`ISBN 978-80-7372-884-7`

[14] VRANÝ, Jiří. *Pojem algoritmus, Složitost algoritmů (přednáška).* [2010].
dostupné z WWW: `www.nti.tul.cz/~vrany/alds/prednaska01_2010.pdf`

[15] WIKIPEDIE, otevřená encyklopedie. *Blender.* [květen 2013].
dostupné z WWW: `cs.wikipedia.org/wiki/Blender`

[16] WIKIPEDIE, otevřená encyklopedie. *Konvexní množina.* [květen 2013].
dostupné z WWW: `cs.wikipedia.org/wiki/Konvexní_množina`

[17] WIKIPEDIE, otevřená encyklopedie. *Python.* [květen 2013].
dostupné z WWW: `http://cs.wikipedia.org/wiki/Python`

[18] WIKIPEDIE, otevřená encyklopedie. *Tuhé Těleso.* [květen 2013].
dostupné z WWW: `cs.wikipedia.org/wiki/Tuhé_těleso`

[19] ŽÁČEK, Jiří,et al. *Moderní pocítacová grafika (2. vydání). Computer Press.* [2005].
`ISBN 80-251-0454-0`

Příloha A

Tabulky ke Kapitole 4

Počet krychlí	5	10	20	50	100	200	300	400	500
Původní Skript	0,003s	0,015s	0,044s	0,19s	0,71s	2,79s	6,31s	10,8s	17,2s
Upravený ps	0,003s	0,014s	0,043s	0,18s	0,68s	2,69s	6,28s	10,9s	17,1s
Skript s BGE	0,010s	0,031s	0,053s	0,05s	0,12s	0,14s	0,21s	0,38s	0,42s
Skript s RB	0,004s	0,006s	0,008s	0,01s	0,03s	0,06s	0,09s	0,13s	0,19s

Tabulka A.1: Test rychlost s krychlemi

Počet zářičů	5	10	20	50	100	200	300	400	500
Původní Skript	0,005s	0,017s	0,067s	0,20s	0,72s	2,84s	6,11s	11,25s	17,58s
Upravený ps	0,005s	0,014s	0,045s	0,19s	0,69s	2,87s	6,07s	10,24s	16,53s
Skript s BGE	0,06s	0,15s	0,27s	0,76s	1,73s	4,53s	6,45s	9,42s	11,48s
Skript s RB	0,004s	0,048s	0,066s	0,14s	0,27s	0,52s	0,81s	1.09s	1,41s

Tabulka A.2: Test rychlost se zářiči

Počet zářičů	5	10	20	40	60	80	100
Původní Skript	0,73s	0,76s	0,76s	0,79s	0,81s	0,83s	0,87s
Upravený ps	0,89s	1,03s	1,32s	1,90s	2,51s	3,08s	3,75s
Skript s BGE	1,94s	2,13s	2,41s	2,96s	3,45s	3,81s	4,12s
Skript s RB	0,30s	0,31s	0,36s	0,47s	0,56s	0,65s	0,74s

Tabulka A.3: Závislost rychlosti na počtu kolizí

ID testu	Původní skript	Upravený ps	Skript s BGE	Skript s RB
1	35	32	23	23
2	45	38	31	31
3	25	21	19	19
4	44	36	34	34
5	76	55	49	49
6	75	61	54	54
7	84	62	52	53
8	71	57	45	45
9	82	69	54	54
10	75	69	53	53
11	97	82	59	59
12	92	73	71	70
13	89	79	70	70
14	96	75	59	59
15	103	84	78	75
16	99	78	72	72
17	114	84	74	74
18	125	110	99	98
19	115	95	81	80
20	119	82	77	77

Tabulka A.4: Přesnost jednotlivých skriptů