VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TENCHOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

DETEKCE KOLIZÍ V POČÍTAČOVÉ GRAFICE

COLLISION DETECTION IN COMPUTER GRAPHICS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **FILIP STUPKA**
AUTHOR

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

BRNO 2021

Abstrakt

Tato práce se zabývá řešením detekcí kolizí triviálních matematických i komplexních objektů složené z trojúhelníkových sítí modelů v trojrozměrném prostoru. Simulace kolizí objektů je z hlediska výkonnostního velmi náročné téma a i přesto, že existují postupy a metody, jak k tomuto problému teoreticky přistoupit, ve většině případů tyto postupy jsou příliš pomalé a tedy je třeba optimalizovat a hledat alternativní řešení. U simulace kolizí je také třeba pracovat s diskrétním i spojitým časem, neboť to souvisí s tím, jak přesné kolize objektů chceme a do jisté míry musíme předpovídat pohyb určitých těles. Tato práce je tedy zaměřena na vývoj herních enginů, optimalizací a implementace kolizních algoritmů.

Abstrakt

This thesis is focused on the problem of collision detection between math defined primitive models and also on triangle networks that form complex polygonal models. Simulation of collision detection is very complex topic from performance standpoint and even though multiple methods that solve these problems do exist, in most cases they are too slow to be any useful and therefore it is encouraged to find optimalisations and alternate solutions. In order to be able to work with collision simulation we need to understand discrete and continuous movement techniques and we need to be able to predict behavior of certain objects. This thesis therefore is based on game engine development, optimalisation and implementation of collision detection algorithms.

Klíčová slova

Detekce kolizí, engine, separating axis theorem, trojúhelníkové sítě, matematická primitiva, simulace, C++, SDL2, OpenGL

Keywords

Collision detection, engine, separating axis theorem, triangle network, math primitives, simulation, C++, SDL2, OpenGL

Citace

STUPKA, Filip. *Detekce kolizí v počítačové grafice*. Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jan Pečiva, Ph.D.

Detekce kolizí v počítačové grafice

Prohlášení

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

. . . .
Filip Stupka
7. května 2021

Poděkování

Chtěl bych velmi poděkovat vedoucímu této práce, panu Ing. Janu Pečivovi, který mi radil, pomáhal a vedl tuto práci.

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2020/2021

Zadání bakalářské práce

Student: **Stupka Filip**
Program: Informační technologie
Název: **Detekce kolizí v počítačové grafice**
Collision Detection in Computer Graphics
Kategorie: Počítačová grafika
Zadání:

1. Nastudujte si metody detekce kolizí ve scéně mezi jednoduchými tělesy i mezi trojúhelníkovými sítěmi a různé optimalizace nad těmito algoritmy pro zvýšení výkonu.
2. Navrhněte aplikaci pro demonstraci vybraných algoritmů z bodu 1. Soustřeďte se především na algoritmy kolizí. Výběr algoritmů a použitých optimalizací konzultujte s vedoucím. Demonstrační aplikace může být realizována formou hry.
3. Navrženou aplikaci implementujte. Volitelně doplňte algoritmy fyziky objektů.
4. Vyhodnoťte zkušenosti s aplikací a výkon algoritmů kolizí.
5. Diskutujte možný další vývoj kolizních algoritmů.
6. Práci publikujte na internetu pod některou z open-source licencí.

Literatura:

- dle doporučení vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- Funkční prototyp aplikace demonstrující jednoduché kolize mezi jednoduchými tělesy.

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 2020
Datum odevzdání: 12. května 2021
Datum schválení: 30. října 2020

Obsah

1. Úvod......6
2. Teoretická část......6
2.1 Simulace......6
2.2 Spojitý a diskrétní pohyb......9
2.3 Práce ve trojrozměrném prostoru......11
2.3.1 Projekce......11
2.4 Barycentrické souřadnice......12
2.5 Načítání a reprezentace trojúhelníkových sítí......14
2.6 Rotace trojúhelníkových sítí......14
3. Implementace......15
3.1 Kolize primitivních trojrozměrných objektů......15
3.1.1 Koule a koule diskrétně......15
3.1.2 Koule a koule spojitě......17
3.1.3 Kvádr a kvádr diskrétně......18
3.1.4 Kvádr a kvádr spojitě......19
3.1.3 Válec a válec diskrétně......20
3.1.4 Válec a válec spojitě......21
3.1.5 Fazole a fazole......22
3.2 Kolize obecnějších trojrozměrných objektů......23
3.2.2 Plocha a plocha......23
3.2.3 Trojúhelník a trojúhelník diskrétně......25
3.2.4 Koule a trojúhelník diskrétně......27
3.2.5 Koule a trojúhelník spojitě......28
3.2.6 Elipsoid a trojúhelník......30
3.3 Trojúhelníkové sítě......32
3.3.1 Separating axis theorem (SAT)......32
3.3.2 Přístup hrubou silou......33
3.3.3 Rozdělení na podčásti......34
3.4 Dělení prostoru......35
3.4.1 Myšlenka......35
3.4.2 Vlastní implementace......35
3.4.3 Porovnání......36
3.5 Implementační problémy......37
3.5.1 Čísla s plovoucí řádovou čárkou......37
3.5.2 Spojitá kolize dvou trojúhelníkům......38
3.5.3 Kolize hešů podčástí scény......41
3.7 Použité knihovny a nástroje......42
4. Měření......42
4.1 Rozdělení prostoru na podprostory......42
4.2 Trojúhelníkové sítě......44
5. Závěr......48
6. Literatura......49

1. Úvod

Tato práce se zabývá výzkumem, technikami a optimalizacemi detekce kolizí trojrozměrně prostorových matematicky definovaných objektů a primitiv. Detekce a rezoluce kolizí je matematicky velice náročný problém a je důležité správně jednotlivé přístupy pochopit abychom je mohli správně implementovat a zužitkovat. Tyto algoritmy pak mohou najít využití při implementaci různých fyzikálních simulačních enginů, které se pak dále dají různě využít jako například pro herní vývoj, vědecké simulace či testování odolnosti různých materiálů.

Abychom si mohli o těchto algoritmech začít povídat je tedy třeba si stanovit cíle, ke kterým se snažíme spět. Nejzákladnějším cílem je vytvořit simulaci fyzikálního světa, ovšem pouze její zjednodušenou variantu, neboť simulace skutečného světa je pomocí dnešní technologie nemožné a je tedy se třeba obrátit na fyzikální zákony, které nám umožní vytvořit kompetentní model simulující realitu. Je samozřejmě nutné si uvědomit, že i tyto zákony a rovnice nejsou vždy optimální a musíme se snažit vždy optimalizovat a pokoušet se zpracovat danou scénu co nejrychleji, neboť pak si můžeme dovolit simulovat složitější systémy.

Proto se v první části této práce zaměříme na teoretické techniky a nástroje, které nám umožní pracovat ve 3D prostoru. Jaké problémy mohou v při používání těchto nástrojů nastat a jak se s nimi vypořádat. Dále si vysvětlíme, jak jednotlivé základní poznatky z analytické geometrie reprezentovat v kódu a ukážeme si, jak tyto poznatky použít pro detekci kolizí jednotlivých základních matematických primitiv. V další sekci se podíváme na na kolizi trojúhelníkových sítí, jakými úhly pohledu lze k tomuto problému přistoupit a jak řešit vysokou algoritmickou složitost těchto pohledů.

V závěru si pak řekneme o tom, jak udržovat a rozdělovat scénu na menší podprostory za účelem vyloučení zbytečných kontrol kolize takových objektů, které jsou od sebe navzájem nepřístupně daleko a zbytečně nespotřebovávaly hodinové cykly procesoru. V další sekci porovnáme jednotlivé optimalizační přístupy k řešení určitých problémů a zjistíme, které metody jsou použitelné a které jsou příliš algoritmicky složité, přičemž všechny tyto testy byly prováděny nad demonstrační aplikací. A úplně na konci jsou vysvětleny implementační problémy a překážky, se kterými jsem se v této práci setkal.

2. Teoretická část

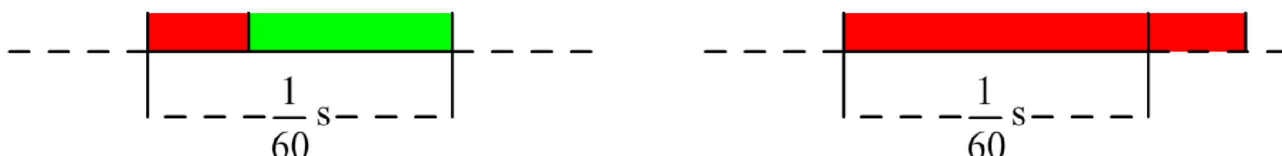
2.1 Simulace

Simulace v dnešní době je velké téma, nejenom ve vesmíru videoher, ale i například zkoumání aerodynamiky auta, či simulace dráhy rakety. A proto je veliký důraz na přesnost a autentičnost, ovšem na druhé straně je pak také schopnost tuto simulaci na moderních počítačích v rozumné době odsimulovat. Jelikož jde ve většině případů o velmi složité algoritmy a soustavy diferenciálních rovnic, je třeba některé aspekty simulace buďto oželet a nebo pokusit se o různé optimalizace, které nám umožní simulovat takzvaně realtime, neboli že simulovaný čas se snaží být roven reálnému času. Hlavní téma simulace objektů je čistě jejich pohyb a tato sekce byla převzata od pana Glenn Fieldera[7]. Pokud simulovaný objekt definujeme jako pozici, vektor rychlosti a typ, pak celistvou simulaci můžeme vnímat následovně: V každém simulovaném kroku vezměme každý objekt a k jeho pozici přičteme jeho rychlostní vektor.

```
function main_loop()
{
    while(true)
    {
        for every object in the scene
        {
            object.position = object.position + object.movement_vector
        }

        redraw_scene()
    }
}
```

Kód 2.1.1: Pseudokód ovládání simulace a rezoluce nového stavu objektů ve scéně.

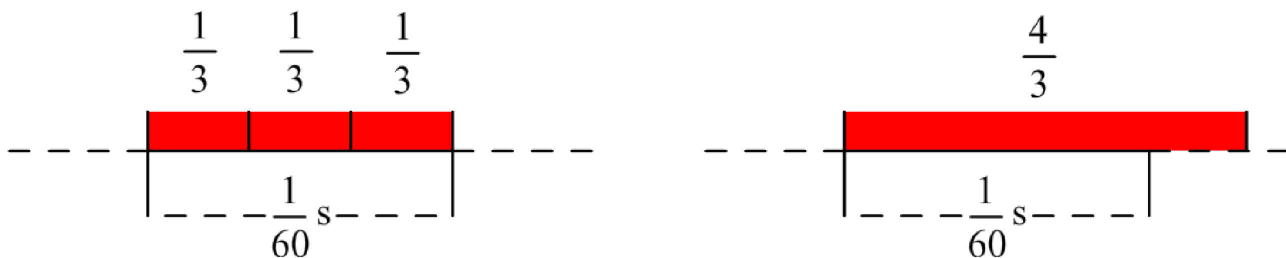
S tímto přístupem je ovšem problém, že nemůžeme předvídat, jakou rychlostí vlastně naše aplikace poběží. Tento faktor může záležet na operačním systému, který může různými způsoby přiřazovat procesorový čas našemu procesu, nebo jaký hardware vlastně používáme. K tomuto problému se dá přistoupit nemálo způsoby. Jedním z nich je limitování času simulačního kroku. Definujme si například, že chceme aby naše simulace dokázala zpracovat a vyrenderovat šedesát snímků za sekundu a tedy platí, že jeden snímek musí vždy konstantně trvat $\frac{1}{60}s$. Abychom dokázali se snímky pracovat, musíme vytvořit časovač, který nám spočte interval zpracování daného snímku t . Pokud dokážeme zpracovat snímky rychleji, než požadovaná frekvence snímků, jednoduše pak spočítáme rozdíl $d=\frac{1}{60}-t$ a po tuto dobu můžeme instruovat operační systém aby náš proces uspal.



Obrázek 2.1.2: Čas zpracování jednoho snímku. Červená barva značí skutečné zpracování snímku simulace a zelená značí dobu, po kterou proces spí. Napravo je pak snímek, který se nevešel do požadované frekvence a zasahuje do dalšího snímku, přičemž další snímek musí čekat.

Potíž nastává, pokud požadovanou frekvenci snímků nedokážeme naplnit, ať už když zpracováváme složitější scénu, nebo nemáme dostatečně silný hardware. V takovémto případě celá simulace se zpomalí a naruší se tak simulovaný pohyb objektů.

Abychom mohli tento problém vyřešit, musíme k samotné simulaci přistoupit poněkud jiným způsobem a to tak, že požadovanou frekvenci snímků nesmíme vnímat jako pevně daný fakt, ale spíše jako orientační hodnotu. Každému simulovanému kroku pak musíme předat informaci o tom, jak dlouho byl předchozí krok zpracováván vzhledem k požadované frekvenci snímků f a díky tomu pak současný simulační krok může manipulovat s rychlostí jednotlivých objektů tak, že při každé změně pozice objektu, translační vektor je vynásoben hodnotou f .



Obrázek 2.1.3: V tomto konceptu se nečeká na frekvenci ale snímky jsou zpracovávány neustále, přičemž se počítá jejich poměr času zpracování vůči referenčnímu času.

```
function main_loop()
{
    old_time = current_time()
    framerate = 1 / 60

    while(true)
    {
        new_time   = current_time()
        delta_time = (new_time - old_time) / framerate
        old_time   = current_time()

        for every object in the scene
        {
            object.position = object.position + object.movement_vector * delta_time
        }

        redraw_scene()
    }
}
```

Kód 2.1.4: Pseudokód pro dynamický management simulace pohybu objektů.

S tímto přístupem, i když náš hardware nestíhá, či scéna je příliš složitá, rychlost simulace zůstává vždy konstantní což je pro naši simulaci nejdůležitější. Ovšem je stále potíž s tou situací, pokud simulační krok je delší, než požadovaná frekvence snímků, ač se to na první pohled nemusí zdát. Pokud poměr času snímku vůči referenčnímu f , je větší než-li jedna, stane se to, že simulace bude jednotlivé simulační kroky přeskakovat, což může mít za následek nepřesné detekce kolize, či špatné změně rychlosti daného objektu. Co tedy s tím? Abychom mohli zachovat škálování simulačních kroků, ale zároveň nepřekročili limit simulačního kroku při výkonnostním vrcholu, či nedostatečném počítačovém vybavení, musíme detekovat, zda-li delta snímku f je větší jak jedna. Poté po jednotkách simulujeme kroky dokud neplatí podmínka $f>1$ a od f odečítáme jedničky při každém cyklu. Musíme si ale dát pozor, co jsme vlastně tímto vlastně vytvořili. Toto řešení na méně silnějších počítačích může způsobit takzvanou spirálu smrti, kde škálování simulačního kroku f bude neustále narůstat, protože nedokáže stihnout odsimulovat všechny kroky. V tomto bodě se tedy musíme smířit s faktem, že budeme moci zvládnout odsimulovat správně scénu pouze při lokálních výkonnostních vrcholech a slabších zařízení se musíme vzdát tak, že natvrdo nastavíme limit, kolik cyklů se může provést pokud platí podmínka $f>1$.

```
function main_loop()
{
    old_time  = current_time()
    framerate = 1 / 60
    limit     = 6

    while(true)
    {
        new_time      = current_time()
        delta_time    = (new_time - old_time) / framerate
        old_time      = current_time()
        limit_counter = 0
        if(delta_time > 1)
        {
            while(delta_time > 1 and limit_counter < limit)
            {
                for every object in the scene
                {
                    object.position = object.position + object.movement_vector * delta_time
                }

                delta_time = delta_time - 1
                limit = limit + 1
            }
        }
        for every object in the scene
        {
            object.position = object.position + object.movement_vector * delta_time
        }

        redraw_scene()
    }
}
```

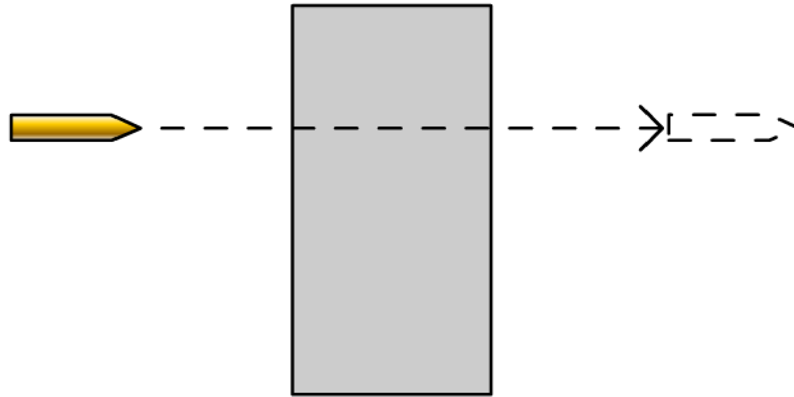
Kód 2.1.4: Simulace za pomocí škálování simulačního kroku a zamezení spirály smrti.

V tomto bodě pak máme plně kompetentní simulační smyčku, se kterou se dá velice příjemně nejenom pracovat, ale pěkně se dokáže přizpůsobit různým monitorům s různou obnovovací frekvencí.

2.2 Spojitý a diskrétní pohyb

Jak jsme si vysvětlili v sekci 2.1 pohyb těles v našem simulační enginu musí být vždy diskrétní a ještě ke všemu tomu tento krok, přesto že dané těleso má konstantní rychlost, simulace může upravit velikost tohoto kroku za účelem korektní prezentace pohybu na obrazovce. Potom tedy závisí na síle daného počítače, na kterém simulace běží, jak skutečně velký tento krok bude.

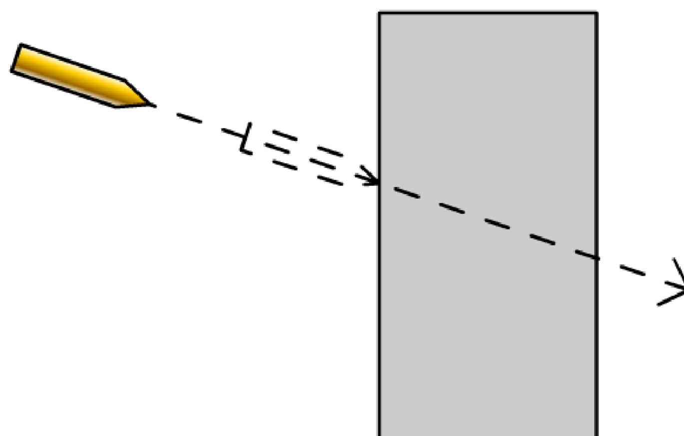
Pokud bychom pracovali čistě s tím faktem, že objekty se mohou pohybovat pouze diskrétně a stavěli bychom algoritmy na základě tohoto faktu, pak bychom velice rychle narazili na nepříjemný problém. Pokud bychom měli slabé zařízení, které by bylo nuceno posunovat jednotlivé objekty po velkých krocích, nebo bychom chtěli simulovat například výstřel z pistole, jehož rychlost bývá obrovská a chtěli bychom zjistit, kdy se tento výstřel dotkne vertikální zdi, pak můžeme narazit na problém, kdy se nám podaří kolizi takzvaně propásnout[9] a pokud bychom byli v situaci, kdy tato zeď má střelu zničit, může se stát, že pak tato střela vyletí mimo relevantní simulovaný prostor a bude nám po zbytek simulace zbytečně zabírat místo a procesorový čas.



Obrázek 2.2.1: Střela má příliš vysokou rychlost a posunutí střely způsobí propásnutí kolize při diskrétní implementaci pohybu objektu.

Jak tento problém řešit? První a naivní myšlenka je celistvý krok střely rozdělit na menší podkroky. S tímto přístupen pak bychom mohli detekovat menší nuance při velkých změnách. Potíží s tímto přístupem je hned několik. Za prvé, špatně se tento přístup škáluje. Pokud bychom například nastavili pevné rozdělení na 3 podkroky, vždy můžeme specifikovat takovou rychlost, která tento problém znovuobnoví. Druhý problém je, že využíváme mnohonásobně více procesorového času na detekci kolizí. V některých simulacích se sice s tímto problémem nesetká díky malé rychlosti jednotlivých objektů, ovšem nemít možnost simulovat vysokorychlostní objekty je veliké a nepříjemné omezení. Proto se tedy k tomuto problému musí přistoupit jinak.

Náš problém je v tom, že při simulaci pohybu objektu nejdříve posuneme objekt na základě jeho rychlosti a až po posunu testujeme, zda-li se náhodou s jiným objektem nedotýká. Abychom mohli zachytit vysokorychlostní kolize, musíme tento přístup do jisté míry otočit. Tedy že nejdříve zjistíme, zda-li něco není v cestě daného objektu a poté tento objekt posuneme. S tímto přístupem ovšem pak nemůžeme použít původní metody na detekci kolize pouze v dané stavu. Do kolizního algoritmu musíme zapojit také rychlost neboli vektor, definující změnu pozice objektu v daném simulačním kroku a pak hledat parametr t, který bude definovat místo, kde rychlostní vektor protnul daný objekt.



Obrázek 2.2.2: Zjištění pozice, kdy střela na své cestě penetrovala zeď pomocí spojité detekce kolizí.

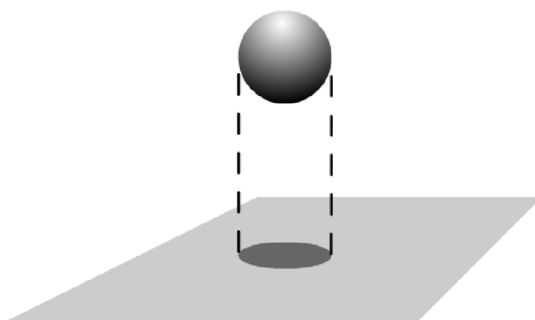
Tato metoda jen mnohonásobně přesnější a spolehlivější, ovšem implementace těchto metod je také o to náročnější a tato metoda také má jednu malou nevýhodu. Tato nevýhoda má bohužel zdroj v implementaci čísla s plovoucí řádovou čárkou. Může se stát, že pokud oba objekty jsou přesně na hranici překrytí a budeme mít zároveň smůlu na špatné zaokrouhlení, může se stát, že se kolize nezjistí a objekty mezi sebou jenom proletí. Tento problém se dá řešit několika způsoby, například že spojitou a diskrétní detekci kolizí zkombinujeme a to tak, že nejdříve provedeme spojitou detekci a posun objektu a poté ještě realizujeme diskrétní detekci kolize a toto nám zajistí skoro perfektní pohyb objektu. Je tedy faktem, že obě varianty, jak spojité tak diskrétní, jsou užitečné v různých případech a tento fakt je motivací implementovat obě verze pro každý objekt.

2.3 Práce ve trojrozměrném prostoru

Analytická geometrie úzce souvisí se zpracováním scény v počítačové grafice. Počítačová grafika používá nemálo konceptů a poznatků z analytické geometrie a tyto poznatky nám mohou pomoci i při řešení detekce a rezoluce kolizí.

2.3.1 Projekce

Jakožto jeden z nejdůležitějších nástrojů, který v této práci používám, je projekce různých útvarů do nižší dimenze. Tato myšlenka je v v této práci použita několikrát a je v zásadě jedna ze základních stavebních kamenů této práce, přičemž byla převzata z knihy pana Jamese M. Van Vertha[8]. Projekci můžeme chápat jako transformaci objektu z jednoho prostoru do prostoru jiného a to s nižší dimenzí, než má originální prostor. V našem případě pak ve většině případů jde o projekci bodu ve 3D prostoru na plochu která je tvořena 2D prostorem a poté se v této práci také hojně vyskytuje projekce bodu ve 3D prostoru na přímku tvořenou 1D prostorem.



Obrázek 2.3.1.1: Jedním z příkladů projekce je například vrhání stínu.

Obě tyto myšlenky používají operaci z analytické geometrie a tou je skalární součin dvou vektorů

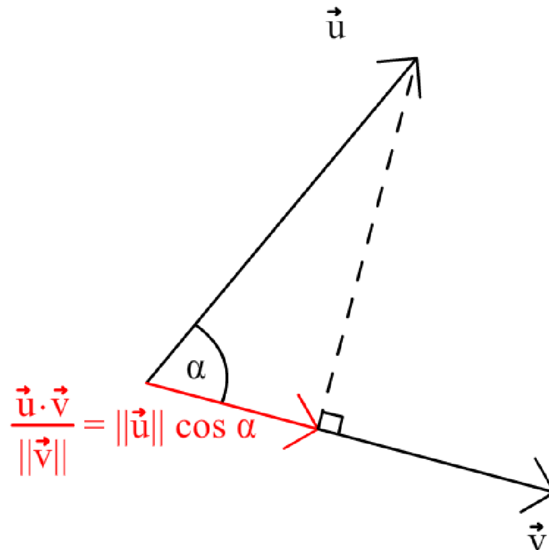
$$\vec{u}=(x_u, y_u, z_u), \vec{v}=(x_v, y_v, z_v)$$
$$\vec{u}\cdot\vec{v}=x_u x_v + y_u y_v + z_u z_v = \|\vec{u}\|\cdot\|\vec{v}\|\cdot\cos\alpha$$

kde $\vec{u}, \vec{v}$ jsou naše vstupní vektory a úhel α je úhel, který tyto dva vstupní vektory mezi sebou svírají. Hlavní myšlenka skalárního součinu, je zjištění, zda-li dva vektory mají podobný směr. Tento fakt vyjadřuje samotná absolutní hodnota výsledku a znaménko pak udává směr, ve kterém si tyto dva vektory jsou podobny. Výsledný skalární součin ovšem nezávisí pouze na směrech obou vektorů, ale také na jednotlivých délkách těchto vektorů, což je důležitý fakt. Pokud si definujeme, že budeme chtít vytvořit projekci vektoru $\vec{u}$ na vektor $\vec{v}$ tak si musíme uvědomit, že jediné s čím bychom měli pracovat, je směr a délka vektoru $\vec{u}$ a u vektoru $\vec{v}$ by nás měl zajímat pouze

směr. Toho můžeme docílit tak, že vektor $\vec{v}$ znormalizujeme $\hat{v}=\frac{\vec{v}}{\|\vec{v}\|}$. Díky této myšlence pak můžeme původní definici skalárního součinu upravit následujícím způsobem

$$\frac{\vec{u}\cdot\vec{v}}{\|\vec{v}\|}=\|u\|\cdot\cos\alpha \quad .$$

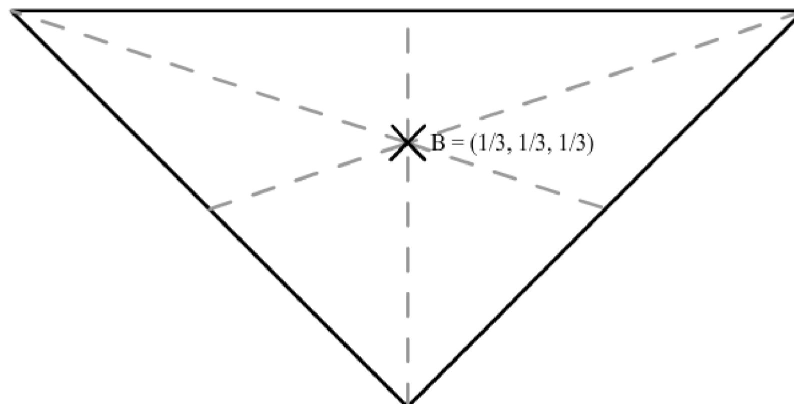
Díky definici cosinové funkce pak získáme velikost projekce vektoru $\vec{u}$ na vektor $\vec{v}$ což je přesně to, co jsme chtěli.



Obrázek 2.3.1.2: Projekce vektoru $\vec{u}$ na vektor $\vec{v}$.

2.4 Barycentrické souřadnice

Dalším z užitečných nástrojů, které tato práce bude využívat, jsou barycentrické souřadnice. Tento koncept je zvláště znám a hojně používán v počítačové grafice a to konkrétně při rasterizaci trojúhelníků. Jde o způsob, jak vyjádřit pozici bodu v trojúhelníku, jakožto vážené hodnoty vzdáleností od jednotlivých vrcholů trojúhelníku. Tato sekce byla převzata od Christera Ericsona[6].



Obrázek 2.4.1: Bod v trojúhelníku reprezentovaný barycentrickými souřadnice.

Abychom pochopili hlavní myšlenku barycentrických souřadnic, definujme si dva body A,B. Pak bod C ležící někde mezi body A,B můžeme definovat pomocí parametrické definice úsečky jako $C=A+t(B-A)=(1-t)A+tB$, kde parametr $t\in\langle 0,1\rangle$ nebo můžeme tento výraz přepsat ještě jednodušeji jako $C=uA+vB$, kde pro parametry u,v platí $u+v=1$ a $u\in\langle 0,1\rangle, v\in\langle 0,1\rangle$. Potom parametry (u,v) jsou pak barycentrickými souřadnicemi bodu C vůči úsečce AB. Tyto souřadnice nám v zásadě parametrický rozdělují prostor a vyjadřují váhu jednotlivých vrcholů a tedy pokud by bod C ležel na stejném místě jako bod A, pak $(u,v)=(1,0)$ a pokud by ležel v bodu B, pak by platilo $(u,v)=(0,1)$. Nejčastější použití barycentrických souřadnic právě najdeme u vyjádření bodu na trojúhelníku, pro zjištění váhy ovlivnění vrcholů na daný bod. Definujme si tedy trojúhelník ABC který je složen ze tří vrcholů A,B,C. Bod P ležící uvnitř tohoto trojúhelníku potom můžeme definovat jako $P=uA+vB+wC$, kde pro parametry u,v,w platí $u+v+w=1$. Tato reprezentace je pouze přepis rovnice

$$P=A+v(B-A)+w(C-A)=(1-v-w)A+vB+wC \quad .$$

Můžeme si všimnout, že díky nezávislosti vektorů AB a AC, kteréžto formují prostor s počátkem v bodu A, můžeme díky tomu znát pouze parametry v,w přičemž poslední parametr u se dá lehce dopočítat.

Jak ovšem tedy spočítat parametry v,w? Pokud si vezmeme definici bodu $P=A+v(B-A)+w(C-A)$ můžeme přehodit jednotlivé parametry následujícím způsobem $v(B-A)+w(C-A)=P-A$. Tento výraz dále můžeme přepsat na $v\vec{v}_1+w\vec{v}_2=\vec{v}_3$. Nyní abychom mohli vyřešit obě neznáme v,w budeme potřebovat systém minimálně dvou lineárních rovnic. To můžeme zařídit tak, že rovnici $v\vec{v}_1+w\vec{v}_2=\vec{v}_3$ na obou stranách upravíme pomocí vektorového skalárního součinu a to pomocí vektorů $\vec{v}_1$ a $\vec{v}_2$ což nám vytvoří soustavu rovnic

$$\begin{aligned}(v\vec{v}_1+w\vec{v}_2)\cdot\vec{v}_1&=\vec{v}_3\cdot\vec{v}_1\\(v\vec{v}_1+w\vec{v}_2)\cdot\vec{v}_2&=\vec{v}_3\cdot\vec{v}_2 \quad .\end{aligned}$$

Tuto soustavu díky linearitě skalárního součinu lze přepsat na

$$\begin{aligned}v(\vec{v}_1\cdot\vec{v}_1)+w(\vec{v}_2\cdot\vec{v}_1)&=\vec{v}_3\cdot\vec{v}_1\\v(\vec{v}_1\cdot\vec{v}_2)+w(\vec{v}_2\cdot\vec{v}_2)&=\vec{v}_3\cdot\vec{v}_2 \quad .\end{aligned}$$

Tuto soustavu rovnic pak lze algoritmicky vyřešit pomocí Kramerova pravidla a následně naimplementovat následovně

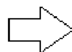
```cpp
glm::vec3 barycentric(const glm::vec3& a, const glm::vec3& b, const glm::vec3& c, const glm::vec3& p)
{
    glm::vec3 d0 = b - a;
    glm::vec3 d1 = c - a;

    glm::vec3 d2 = p - a;
    float d00 = glm::dot(d0, d0);
    float d01 = glm::dot(d0, d1);
    float d11 = glm::dot(d1, d1);
    float d20 = glm::dot(d2, d0);
    float d21 = glm::dot(d2, d1);
    float denom = d00 * d11 - d01 * d01;

    float v = (d11 * d20 - d01 * d21) / denom;
    float w = (d00 * d21 - d01 * d20) / denom;

    return glm::vec3(v, w, 1.0 - v - w);
}
```

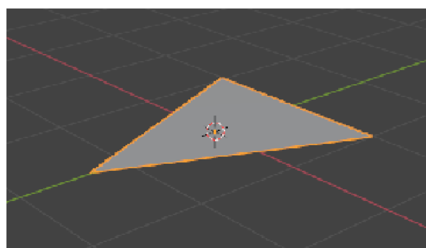
Kód 2.4.2: Implementace výpočtu barycentrických souřadnic v jazyce C++.

2.5 Načítání a reprezentace trojúhelníkových sítí

V této práci dojde na introdukci objektů reprezentované takzvanou trojúhelníkovou sítí. Jelikož moderní vykreslovací algoritmy a grafické karty očekávají na svém vstupu objekty tvořené z trojúhelníků, je velice výhodné v podobném formátu takovýto objekt ukládat na disku a potom jej načítat do paměti co možná v nejpříjemnější formě právě pro grafickou kartu, aby se zamezilo zbytečným konverzím, které by nás stály procesorové cykly. Trojúhelníková síť pak definuje tvar a jak se tento objekt ve scéně vykreslí. Pro reprezentaci trojúhelníkových sítí na disku jsem použil Wavefront Object 3D Model[10], pro který jsem byl nucen napsat modul na načítaní a zpracování. Trojúhelníkové sítě se pak skládají z vrcholů a indexů do těchto vrcholů tvořící plochy trojúhelníků.

```
o Triangle
v -1.045171 0.000000 -0.996179
v 1.012510 0.000000 -0.996179
v -0.016331 0.000000 1.339126
vt 0.000000 0.000000
vt 0.000000 0.000000
vt 0.000000 0.000000
vn 0.0000 -1.0000 0.0000
f 2/1/1 3/2/1 1/3/1
```





Obrázek 2.5.1: Definice trojúhelníku ve formátu Wavefront Object.

Z formátu Wavefront Object lze taky přečíst a definovat materiály ploch, normály ploch a také může definovat místo trojúhelníků, plochy sestavené ze čtyř vrcholů takzvaný quad. Tyto atributy jsou ovšem nad rámec této práce a proto jsem se jimi nezabýval.

2.6 Rotace trojúhelníkových sítí

Schopnost dokázat otáčet objekt je snad pro každý renderovací engine nedílnou součástí přičemž pokud je objekt natočen je také třeba s tímto faktem počítat v kolizních algoritmech pro tyto objekty. K rotaci objektů ve trojrozměrném prostoru se dá přistoupit dvěma způsoby a to eulerovými úhly nebo rotační maticí. Jelikož eulerové úhly obsahují zásadní omezení zvané gimbal lock, který v zásadě neumožňuje rotaci objektu za určitých podmínek. Proto jsem se ve své implementaci zaměřil na použití rotačních matic, které nám dávají plnou kontrolu nad rotací objektu. Rotaci bychom chtěli mít implementovánu tak, že rotační funkci bychom předali referenční osu $\vec{n}=(x_n, y_n, z_n)$, podle které by se objekt otáčel a úhel α o který by se měl objekt kolem referenční osy otočit, přičemž z těchto argumentů by se vypočítala rotační matice a všechny troj-

helníky daného objektu by se transformovali pomocí této rotační matice. Základem výpočtu rotační matice je takzvaný kvaternion[11]. Kvaternion je reprezentován čtyř-dimenzionálním vektorem $\vec{q}=(x_q, y_q, z_q, w_q)$, přičemž jednotlivé parametry se dají vypočítat jako

$$x_q=\cos\left(\frac{\alpha}{2}\right)$$

$$y_q=x_n \cdot \sin\left(\frac{\alpha}{2}\right)$$

$$z_q=y_n \cdot \sin\left(\frac{\alpha}{2}\right)$$

$$w_q=z_n \cdot \sin\left(\frac{\alpha}{2}\right)$$

Tento kvaternion lze poté převést na rotační 4x4 matici M . Pokud poté budeme chtít transformovat 3D bod $B=(x_B, y_B, z_B)$ pomocí rotační matice M musíme nejdříve bod B převést na čtyř-dimenzionální vektor tak, že jako poslední složku nastavíme na jedničku $B'=(x_{B'}, y_{B'}, z_{B'}, 1)$ abychom tento bod byl kompatibilní s rotační maticí M a poté tento bod s rotační maticí vynásobíme $M'=M \cdot B'$ a tím dostaneme transformovanou matici, kterou poté můžeme převést zpět na tří-dimenzionální vektor a tím dostaneme transformovaný 3D bod.

3. Implementace

3.1 Kolize primitivních trojrozměrných objektů

Při každé rezoluci kolizí budeme potřebovat určitým způsobem reprezentovat výsledek. Co je výsledkem detekce a rezoluce kolizí? Odpověď zní: příznak, zda-li kolize nastala a pokud nastala tak by se hodil vektor, který by definoval hloubku zanoření k tomu, abychom tyto dva objekty mohli odseparovat od sebe.

```
struct Data
{
    Data() {}

    bool       occurred     { false };
    glm::vec3 displacement { 0 };

    explicit operator bool() const { return occurred; }
};
```

Kód 4.1: Definice struktury výsledku kolize.

Každá funkce, která má za úkol vyřešit kolizi pak tuto strukturu vrací a dá se použít na další zpracovaní a analýzu kolize.

3.1.1 Koule a koule diskrétně

Detekce kolizí mezi koulemi je jeden z nejlehčích algoritmů v této práci. Nejdříve je třeba si definovat kouli ve trojrozměrném prostoru abychom ji mohli reprezentovat datovou strukturou v mém enginu. Kouli lze definovat jako trojrozměrný poziční bod reprezentující střed koule a poloměr.

```
struct Ball
{
    Ball() {}
    Ball(const glm::vec3& pos, float radius) : pos(pos), radius(radius) {}

    glm::vec3 pos    { 0 };
    float     radius { 0 };
};
```

Obrázek 3.1.1.1: Definice struktury 3D koule.

Z matematického hlediska 2 koule se dotýkají, nebo spíše překrývají, pokud vzdálenost mezi dvěma koulemi je menší než-li součet jejich poloměrů. Zjistit euklidovskou vzdálenost mezi dvěma pozičními body (x1, y1, z1) a (x2, y2, z2) je vskutku triviální a porovnání s poloměry (r1) a (r2) je ještě jednodušší.

$$\sqrt{((x1-x2)^2+(y1-y2)^2+(z1-z2)^2)}\leq r1+r2$$

Je také možné se při výpočtu euklidovské vzdálenosti vyvarovat používání druhé odmocniny tak, že umocníme obě strany na druhou a tím pádem zoptimalizujeme tuto kontrolu.

$$((x1-x2)^2+(y1-y2)^2+(z1-z2)^2)\leq(r1+r2)^2$$

Co ale bude výsledkem kolize? Samozřejmě vektor. A u vektoru musíme nastavit směr a velikost. Směr můžeme zjistit velice jednoduše díky perfektní zaoblené nátuře koule. Směr je zkrátka rozdíl středů koulí. S velikostí to je složitější. Zde musíme vypočítat, jak hluboko v sobě koule vlastně jsou a zde se bohužel již odmocnině nevyhneme protože musíme vypočítat velikost rozdílů středů koulí.

$$depth=r1+r2-|\overrightarrow{(x1-x2,y1-y2,z1-z2)}|$$

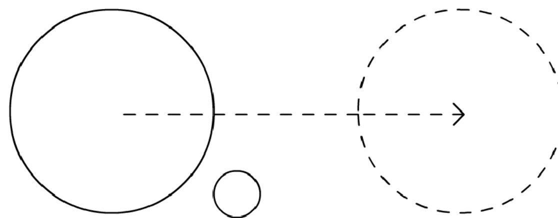
Celistvý kód pro kontrolu pak v C++ je triviální naimplementovat.

```
Data BallVsBall(const glm::vec3& pos1, const float& radius1,
                const glm::vec3& pos2, const float& radius2)
{
    Data res;

    //calculate delta
    glm::vec3 delta = pos1 - pos2;

    //if the distance between balls is smaller than sum of their radii, collision happened
    if (glm::dot(delta, delta) <= (radius1 + radius2) * (radius1 + radius2))
    {
        res.occurred     = true;
        res.displacement = glm::normalize(delta) * (radius1 + radius2 - glm::length(delta));
    }

    return res;
}
```

Kód 3.1.1.2: Detekce kolizí mezi koulemi.

3.1.2 Koule a koule spojitě

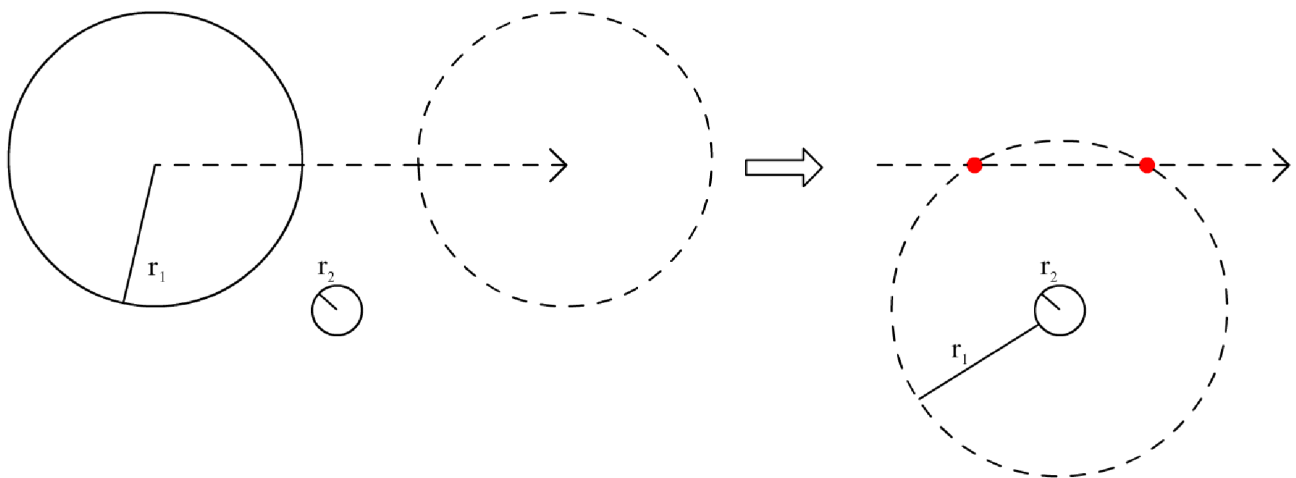
S tímto přístupem je ovšem ten problém, který jsem zmínil v sekci 2.2 a je to ten, že při pouhé detekci překrytí máme problém se závislostí na velikosti simulačního kroku a pokud se obě koule pohybují příliš rychle, může se stát, že kolizi takříkajíc propásneme a nebudeme schopni ji detekovat. Proto se k tomuto problému musí přijít z jiné strany a kromě dvou koulí musíme také počítat s pohybovou rychlostí jedné z daných koulí.



Obrázek 3.1.2.1: Demonstrace problému příliš rychlé koule.

V některých simulacích toto není problém, protože buďto mají výkonný počítač, který dokáže simulační krok mí dostatečně malý, nebo existují simulace, které takto rychlé objekty nemá. Ovšem například při simulaci pohybu náboje se tento přístup rozhodně nehodí. V dalších úvahách a implementacích tedy budeme pracovat s dvojicí pohyblivý objekt s rychlostí a statický objekt bez pohybu.

Jak tedy tento problém vyřešit a zakomponovat rychlost jedné z koulí do naší funkce? Základní myšlenka je taková, že převedeme problém kolize koule a koule na kouli a úsečku, potom bychom mohli díky vlastnostem pohybující se koule testovat, zda-li originální vektor rychlosti protíná statickou kouli, ovšem statická koule by nyní měla poloměr ne *r2* ale *r1 + r2*.



Obrázek 3.1.2.2: Převedení problému koule a koule na koule a úsečka.

Díky tomuto převodu můžeme přesně říci, kde se pohybující se koule dotkla statické. Jak ale zjistit zda-li úsečka protla kouli? Zde musíme zabrousit do matematické definice úsečky a koule. Dle definice pana Garetha[1] jakýkoliv bod na úsečce se dá reprezentovat pomocí formule $p_1 + t(p_2 - p_1)$ kde p_1 je bodový počátek úsečky p_2 je koncový bod úsečky a t je parametr, v našem případě neznámá, nabývající hodnot mezi 0 až 1. Jakýkoliv bod x na kouli musí mít se poloměrem koule vztah

$$|x - q| = r$$

což v zásadě znamená, že aby mohl ležet bod na kouli, musí nutně být od středu koule vzdálen délkou, která je rovná poloměru koule. Pokud za x dosadíme naši reprezentaci úsečky pak dostáváme

$$|p_1+t(p_2-p_1)-q|=r$$

a pokud obě strany umocníme na druhou,

$$|p_1+t(p_2-p_1)-q|^2=r^2$$

můžeme využít vlastnosti skalárního součinu vektorů, kde $|\vec{v}|^2=\vec{v}\cdot\vec{v}$ pro každý vektor $\vec{v}$

$$(p_1+t(p_2-p_1)-q)\cdot(p_1+t(p_2-p_1)-q)=r^2$$

Po vypočítání skalárního součinu a přesunutí všech proměnných na jednu stranu, dostáváme

$$t^2((p_2-p_1)\cdot(p_2-p_1))+2t((p_2-p_1)\cdot(p_1-q))+(p_1\cdot p_1+q\cdot q-2p_1\cdot q-r^2)=0$$

což je kvadratická rovnice s atributy

$$a=(p_2-p_1)\cdot(p_2-p_1)$$

$$b=2((p_2-p_1)\cdot(p_1-q))$$

$$c=p_1\cdot p_1+q\cdot q-2p_1\cdot q-r^2$$

kde pak existují dvě řešení a to

$$t=\frac{-b\pm\sqrt{(b^2-4ac)}}{2a}\ .$$

Dává smysl, že bychom měli dvě řešení? Ano, protože úsečka může protnout danou kouli ve dvou bodech. Co je poté ještě třeba zkontrolovat, je ten fakt, zda-li obě řešení jsou v intervalu 0 až 1. Pokud jsou mimo, znamená to, že bod průniku přesahuje hranice přímky a tedy je pro nás nepoužitelný. Dále pokud jsou obě řešení potenciálně validní, zjistíme, který bod dotyku s koulí je blíže bodu p_1 a to bude náš finální výsledek kolize.

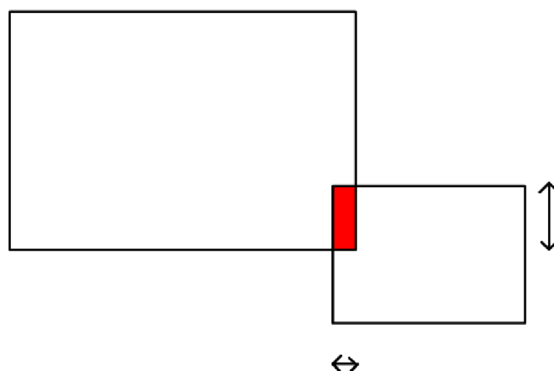
3.1.3 Kvádr a kvádr diskrétně

Kvádr si musíme, podobně jako kouli, definovat tak, abychom jej mohli reprezentovat datovou strukturou. Kvádr lze definovat jako trojrozměrný poziční bod a poté je potřeba znát výšku, šířku a délku.

```
struct Box
{
    Box() {}
    Box(const glm::vec3& pos, const glm::vec3& dims) : pos(pos), dims(dims) {}

    glm::vec3 pos  { 0 };
    glm::vec3 dims { 0 };
};
```

Kód 3.1.3.1: Reprezentace kvádru.

Všimněme si, že tato struktura je zarovnaná podle os a tedy nelze ji otáčet. Tím pádem se detekce kolizí mezi kvádry podstatně zjednoduší a zrychlí. Kdy se tedy dva kvádry překrývají? V každé z os musíme zjistit, zda-li rozdíl v té dané ose je menší než-li součet šířek, výšek nebo hloubek obou kvádrů, podle toho, na jaké ose zrovna pracujeme.



Obrázek 3.1.3.1: Detekce překrytí ve dvourozměrném prostoru.

V obrázku 3.1.3.1 můžeme vidět tento problém ve 2D přičemž do 3D lze toto velice jednoduše rozšířit přidáním jedné další osy. Musíme tedy zkontrolovat celkem 3 podmínky, které když jsou všechny pravdivé, můžeme s klidem říci, že oba kvádry se překrývají.

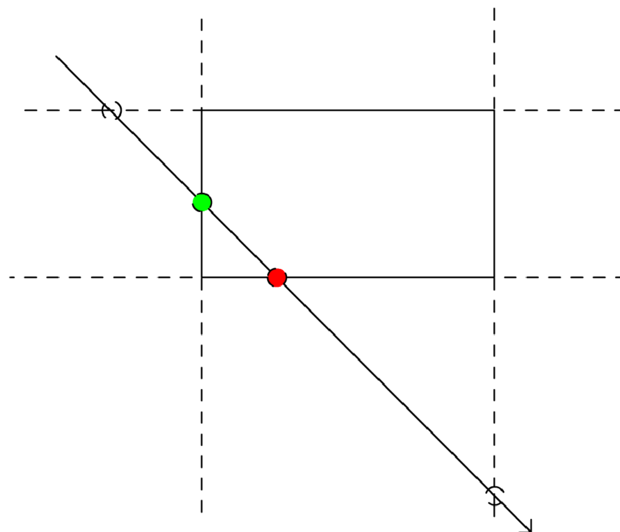
$$|x1 - x2| \leq \frac{w1}{2} + \frac{w2}{2}$$
$$|y1 - y2| \leq \frac{h1}{2} + \frac{h2}{2}$$
$$|z1 - z2| \leq \frac{d1}{2} + \frac{d2}{2}$$

Pokud víme, že se oba kvádry překrývají, musíme si v podstatě vybrat osu, která bude figurovat jako výsledek kolize. Když se podíváme znovu na obrázek 3.1.3.1 pak si můžeme všimnou, že samotné překrytí vytváří jakýsi samostatný kvádr vyznačený v obrázku červeně. Pokud se budeme držet teorie minimální práce potřebná k odseparování objektů, pak si musíme vybrat takovou osu, ve které červený kvádr má nejmenší zastoupení. V literatuře se můžeme dočíst o takzvaném: minimal translation vector. Tedy naším cílem je najít nejkratší stranu červeného kvádru a tuto šířku potom nastavit jako výsledek kolize, což je v kódu velice jednoduchá záležitost.

3.1.4 Kvádr a kvádr spojitě

Když bude zase chtít zapojit do našeho puzzle rychlost, budeme muset se zase na danou problematiku podívat z jiného úhlu a bude mít velice podobný nápad, který jsme využili u koule v sekci 3.1.2. Esenciálně zase převedeme problém kvádr a kvádr na úsečku a kvádr, přičemž kvádr který bude testován vůči úsečce, bude mít šířku, výšku a délku nastavenou na součet obou kvádrů.

Jak ale detekovat kolizi úsečky a kvádru? Každý kvádr se skládá z šesti ohraničených ploch, které definují jeho hranice. Pokud bychom tyto plochy rozšířili na nekonečně velké plochy a rychlostní vektor bychom také prodloužili do nekonečně velké vzdálenosti, vždy alespoň ve dvou plochách by je takováto přímka protla. Pokud bychom zjistili bod průniku a poté zkontrolovali zda-li tento bod průniku leží uvnitř omezené strany kvádru, získali bychom tím informaci o kolizi i bod, ve kterém kolize nastala. Je ovšem třeba si třeba uvědomit, že přímka rychlosti může protnout více jak dvě plochy daného kvádru a tedy je třeba se rozhodnout, který bod s kterou plochou je ten správný. Tento problém se vyřeší velice jednoduše vybráním takového bodu, který je nejblíže středu pohyblivého kvádru (kvádr ze kterého vektor rychlosti vychází a má v něm počátek).



Obrázek 3.1.4.1: Průnik úsečky a kvádru.

3.1.3 Válec a válec diskrétně

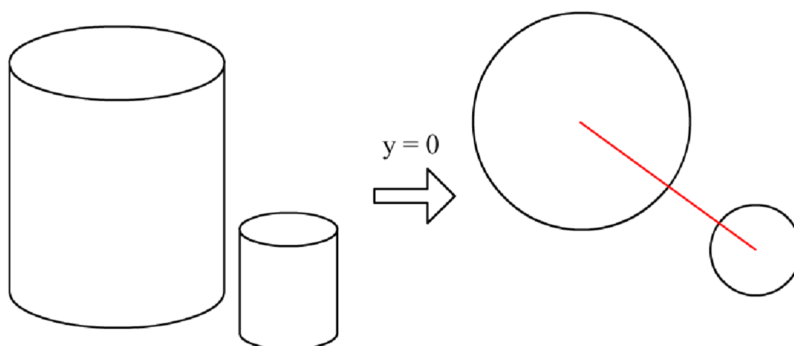
Válec si můžeme definovat velice podobně jako kvádr nicméně místo třech prostorových vlastností tedy výška, šířka a hloubka použijeme pouze dva atributy a to výška a rádius.

```
struct Cylinder
{
    Cylinder() {}
    Cylinder(const glm::vec3& pos, const glm::vec2& radius) : pos(pos), dims(dims) {}

    glm::vec3 pos  { 0 };
    glm::vec2 dims { 0 };
};
```

Kód 3.1.3.1: Struktura válce.

Válec jako takový je už složitější geometrický objekt a v mém návrhu řešení budeme muset spojit síly vícero myšlenek tak, abychom s válci mohli pracovat a to konkrétně spojíme myšlenky, které jsme použili při implementaci kvádru i koule zároveň.

Vertikální překrytí vyřešíme jak u kvádru, jde zjistíme vertikální rozdíl středů válců a pokud tento rozdíl bude menší než výška obou válců, pak máme vertikální překrytí vyřešené. Horizontální překrytí bude trochu jiné. Co uděláme, že vytvoříme projekci obou válců z vrchního pohledu, což nám vytvoří iluzi dvou 2D kružnic. Tuto projekci uděláme tak, že pokud si definujeme y osu jako vertikální, pak tuto osu vynulujeme, nebo ji můžeme úplně vyloučit z našich výpočtů.



Obrázek 3.1.3.2: Projekce z vrchu pomocí vynulování vertikální osy.

Pokud pak využijeme stejnou myšlenku jak u koulí v sekci 3.1, můžeme jednoduše převést tento problém do dvourozměrného prostoru a zjistit zda-li se obě kružnice překrývají. Nyní můžeme obě myšlenky zkombinovat a aby ke kolizi došlo musí obě podmínky platit.

$$|y1-y2| \leq \frac{h1}{2}+\frac{h2}{2}$$

$$(x1-x2)^2+(z1-z2)^2 \leq (r1+r2)^2$$

Nyní jak na rezoluci? To právě záleží, jestli se dva válce dotkly právě na svých čepičkách nebo na svých bocích. Čili musíme spočítat jak hluboko jsou v sobě zanořeny vzhledem k čepičkám a hloubku zanoření ku svým obalům a která hloubka bude menší, tu si vybereme jako výsledek kolize. Obě hloubky spočítáme úplně stejně tak jak v sekcích 3.1.2 a 3.1.4 až na to, že u výšky budeme počítat pouze s jednou osou a u obalu budeme používat 2D kružnice místo 3D koule.

$$depth_{vertical}=\left(\frac{h1}{2}+\frac{h2}{2}\right)-|y1-y2|$$

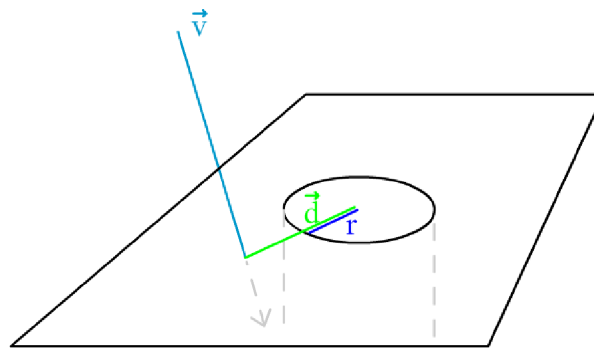
$$depth_{horizontal}=(r1+r2)-\sqrt{\left((x1-x2)^2+(z1-z2)^2\right)}$$

Pak už je lehce porovnáme, která z těchto hodnot je menší a zároveň využijeme tyto hodnoty pro zápis výsledku kolize.

3.1.4 Válec a válec spojitě

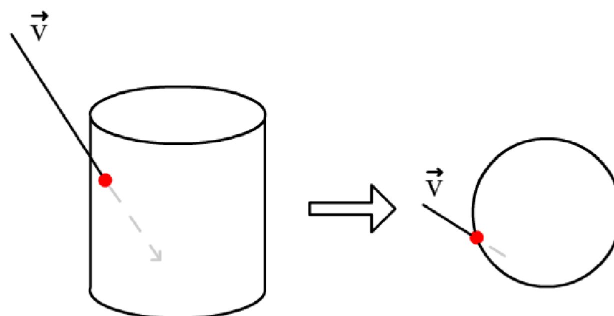
U spojité varianty využijeme stejné myšlenky jako u předchozích spojitých variant, tedy že budeme testovat kolizi úsečky, reprezentující pohyb za jeden simulační krok, proti válci se sečtenou velikostí obou testovaných válců.

Bohužel, zde už nemůžeme použít žádný trik jako u předchozích objektů a budeme muset rozdělit válec na jednotlivé podčásti, abychom byli schopni detekovat kolizi. V prvé řadě se zaměříme na podstavy. Podstavy v tomto případě budou vždy osově zarovnané plochy omezené kružnicí. Pokud rozšíříme podstavy na nekonečné plochy, můžeme využít detekci kolize úsečky a nekonečné plochy, jak jsem použili u spojité kolize kvádrů a poté zjistit vzdálenost mezi bodem průniku a středem naší podstavy. Pokud vzdálenost je menší jak rádius, můžeme prohlásit, že úsečka protla danou podstavu. Toto je ovšem nutné zkontrolovat pro obě podstavy.



Obrázek 3.1.4.1: Průnik úsečky s plochou.

Podstavy tedy máme, co s pláštěm? Základní myšlenka, kterou jsem použil, byla, že samotný plášť protáhneme vertikálně do nekonečna, zjistit bod průniku úsečky a nekonečně vysokého válce a pokud tento bod se vyskytuje mezi podstavami válce, znamená to, že úsečka úspěšně protla plášť. Jak ale protáhnout válec do nekonečna? Velmi jednoduše. Definujme, že osa y v 3D prostoru definuje výšku pláště, pak zkrátka tuto osu budeme ignorovat což způsobí projekci do 2D prostoru a můžeme zjišťovat kolizi úsečky a pláště.



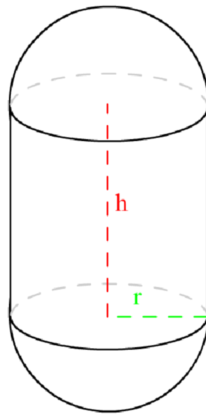
Obrázek 3.1.4.2: Projekce válce ignorováním vertikální osy.

Je ovšem třeba pamatovat na jednu věc, pokud počátek leží uvnitř nekonečně vysoké varianty válce, pak tuto kontrolu je třeba přeskočit, protože neexistuje takový bod, který by protínal plášť a přitom nejprve nenarazil na jednu z podstav.

Může se tedy stát, že z těchto tří kontrol můžeme dostat mezi žádným až třemi různými body průniku. Který tedy vybrat? Samozřejmě ten, který je nejblíže počátku. To bude náš bod průniku a můžeme ho vracet jako výsledek kolize.

3.1.5 Fazole a fazole

Fazole je v podstatě spojení předchozích myšlenek v jednu. Fazole v mém provedení je z matematického hlediska pouze válec, přičemž podstavy tvoří pravidelné polokoule. Reprezentovat fazoli v kódu pak lze stejně jako válec přičemž poloměr polokoulí je dán poloměrem válce.



Obrázek 3.1.5.1: Reprezentace fazole.

```
struct Bean
{
    Bean() {}
    Bean(const glm::vec3& pos, const glm::vec2& dims) : pos(pos), dims(dims) {}

    glm::vec3 pos  { 0 };
    glm::vec2 dims { 0 };
};
```

Kód 3.1.5.2: Struktura fazole.

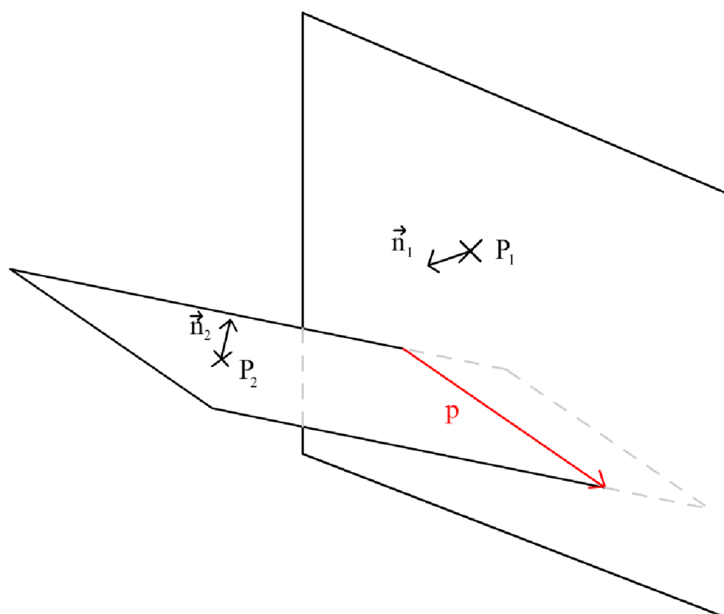
Jelikož tento útvar se skládá z objektů pro které jsme již detekci kolizí naimplementovali přičemž polokouli můžeme reprezentovat jako celistvou kouli a daná koule se nikdy nedotkne pláště válcovité části fazole, pak můžeme pouze zkontrolovat kolizi mezi jednotlivými koulemi a pak mezi plášti a pokud u některé z kontrol detekujeme kolizi, vracíme výsledek a máme hotovou diskrétní kolizi.

U spojité využijeme spojitých verzí detekce kolizí koulí a válců a poté vybereme ze všech kandidátů na kolizi toho, který je nejblíže počátku rychlostního vektoru. Takovýto objekt složený z více objektů se pak dá použít například pro reprezentaci hráče, pohybující se po 3D trojúhelníkové síti.

3.2 Kolize obecnějších trojrozměrných objektů

3.2.2 Plocha a plocha

Detekce kolizí ploch znamená hledání přímky, která sdílí příslušnost obou ploch. Je také dobré mít na paměti ten fakt, že se může stát, že taková to přímka nebude existovat, pokud budou obě plochy navzájem rovnoběžné. Z hlediska matematického, plocha je definovaná jako trojrozměrný bod v prostoru a potom normála, definující směr plochy.



Obrázek 3.2.2.1: Definice ploch.

```
struct Plane
{
    Plane() {}
    Plane(const glm::vec3& pos, const glm::vec3& nor) : pos(pos), nor(nor) {}

    glm::vec3 pos { 0 };
    glm::vec3 nor { 0, 1, 0 };
};
```

Kód 3.2.2.2: Struktura plochy.

Jak tedy zjistit danou přímku p ? Přímku můžeme definovat podobně jako plochu bod v prostoru A a směr přímky $\vec{u}$ přičemž ještě musíme přidat parametr t , který pak bude definovat každý bod na naší nekonečné přímce $p=A+t\cdot\vec{u}$. Zjistit směr přímky je velice jednoduchá záležitost známe-li definici vektorového součinu, který při vstupních dvou vektorech nám vypočítá nový vektor, který je na oba vstupní vektory kolmý. A to je přesně co můžeme použít na vypočítání směru

$$\vec{u}=\vec{n}_1\times\vec{n}_2 \quad .$$

Nyní si ale musíme dát pozor na ten případ, kdy obě plochy jsou k sobě navzájem rovnoběžné a tedy žádná přímka neexistuje. To se dá zjistit tak, že pokud determinant výsledku $Det(\vec{n}_1,\vec{n}_2)=((\vec{n}_1\times\vec{n}_2)\cdot(\vec{n}_1\times\vec{n}_2))=\vec{u}\cdot\vec{u}$ vektorového součinu byl nulový, pak tedy obě plochy jsou navzájem rovnoběžné, $\vec{u}$ nemá žádný směr a nelze s ním dále pracovat. Po této kontrole musíme přistoupit ke hledání bodu A . Bod A je definován rovnicí

$$A=\frac{((\vec{u}\times\vec{n}_2)*-(P_1\cdot\vec{n}_1))+((n_1\times\vec{u})*-(P_2\cdot\vec{n}_2))}{Det(\vec{n}_1,\vec{n}_2)} \quad .$$

Pak oba parametry, tedy bod a směrový vektor uložíme do struktury a můžeme ji vrátit jako výsledek kolize.

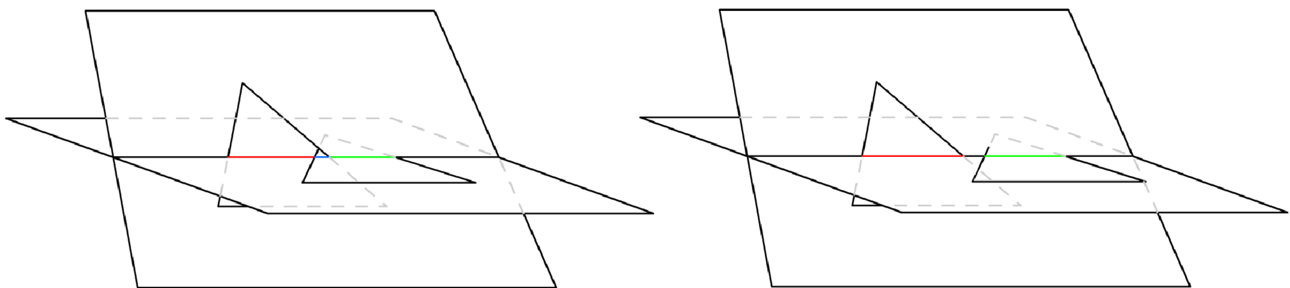
3.2.3 Trojúhelník a trojúhelník diskrétně

Proč jsme vlastně implementovali detekci kolize ploch v sekci 3.2.2? Tento algoritmus bude velice užitečný při detekci kolizí dvou trojúhelníků. Tento algoritmus je ten nejdůležitější algoritmus v této práci a byl převzat od pana Tomase Möllera[2] v jeho práci: A Fast Triangle-Triangle Intersection Test. Jaká je tedy myšlenka? Nejdříve si definujme strukturu trojúhelníku. Trojúhelník můžeme definovat jako trojici bodů či lépe vrcholů ve 3D prostoru T_0, T_1, T_2 .

```
struct Triangle
{
    Triangle() {}
    Triangle(const glm::vec3& p1, const glm::vec3& p2, const glm::vec3& p3) : p1(p1), p2(p2), p3(p3) {}

    glm::vec3 p1 { 0 };
    glm::vec3 p2 { 0 };
    glm::vec3 p3 { 0 };
};
```

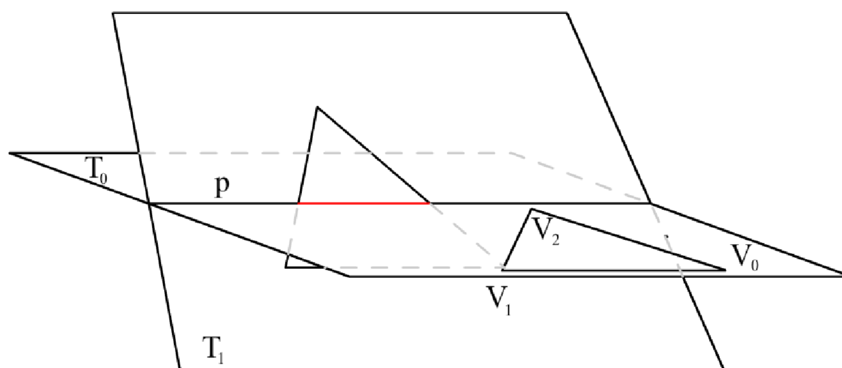
Kód 3.2.3.1: Struktura trojúhelníku.

Základní myšlenka je taková, že z trojúhelníků t_0, t_1 vytvoříme nekonečné plochy P_0, P_1 a nalezneme přímku průniku. Poté vytvoříme projekci této přímky na oba trojúhelníky, což nám z obou trojúhelníků vytvoří dva intervaly, které potom otestujeme zda-li se překrývají.



Obrázek 3.2.3.2: Hlavní myšlenka kolize trojúhelníků.

Čili ze všeho nejdříve je potřeba zkonstruovat plochy z daných trojúhelníků. Jak jsme si ukázali v sekci 3.2.2 pak plocha potřebuje bod ležící na dané ploše a pak normálu, která definuje směr plochy. Bod si můžeme vybrat jakýkoliv z vrcholů trojúhelníku například $A_{t1} = V_0$ a při výpočtu normály znovu využijeme vlastnosti vektorového součinu, přičemž dva vektory, které figurují jako vstup, budou libovolné hrany našeho trojúhelníka, jako například $\vec{n}_{t0} = (V_1 - V_0) \times (V_2 - V_0)$. Takto sestrojíme plochy P_0, P_1 pro oba trojúhelníky t_0, t_1 a pokud nejsou rovnoběžné, najdeme přímku p procházející oběma plochami jak jsme si ukázali v sekci 3.2.2. Nyní, jak jsme si řekli, bychom měli přistoupit k projekci trojúhelníků na danou přímku, ovšem je si potřeba uvědomit, že takovou projekci nelze vždy provést. Kdy taková situace nastane? Pokud přímka p neprotíná alespoň jeden z trojúhelníků, nebo lépe řečeno, pokud všechny vrcholy trojúhelníku t_0 (respektive t_1) existují v tom samém poloprostoru rozdělující prostor plochy P_1 (respektive P_0).

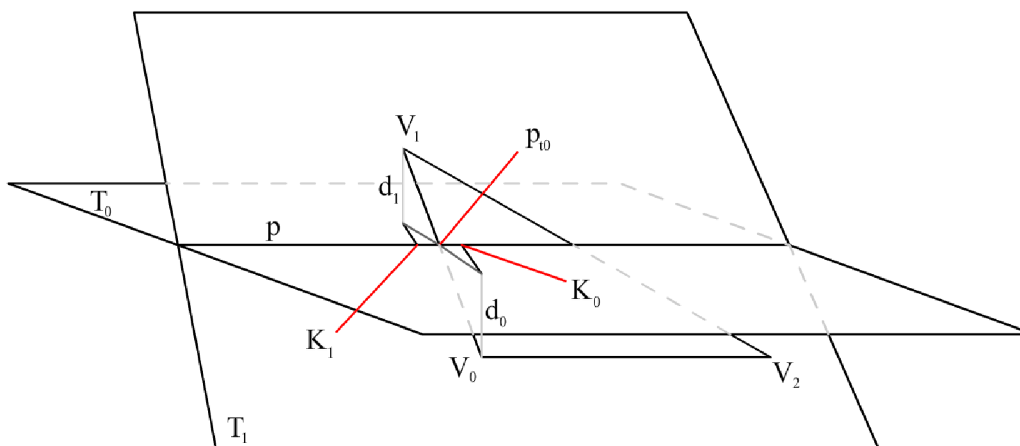


Obrázek 3.2.3.3: Všechny body V_0, V_1, V_2 leží na stejném poloprostoru od plochy T_1 .

Tuto kontrolu musíme provést pro oba trojúhelníky t_0, t_1 a potom teprve můžeme provádět projekci vrcholů trojúhelníků na přímku p . Pro zjištění na které poloprostoru se vyskytuje daný vrchol využijeme další vlastnosti výsledku skalárního součinu a to jeho znaménko. Tato vlastnost nám dokáže určit, na které straně vektory vůči sobě jsou. Čili pro trojúhelník $t_0=(V_0, V_1, V_2)$ a plochu $T_1=A_{t1}+t\vec{n}_{t1}$ aby všechny vrcholy byly na jednom poloprostoru, musí platit

$$sign((\vec{n}_{t1}\cdot V_0)-(\vec{A}_{t1}\cdot\vec{n}_{t1}))=sign((\vec{n}_{t1}\cdot V_1)-(\vec{A}_{t1}\cdot\vec{n}_{t1}))=sign((\vec{n}_{t1}\cdot V_2)-(\vec{A}_{t1}\cdot\vec{n}_{t1})) .$$

Podobným způsobem stejnou kontrolu musíme provést pro trojúhelník t_1 a plochu T_0 . Pokud obě tyto podmínky neplatí potom musíme přistoupit k projekci trojúhelníků na přímku.



Obrázek 3.2.3.4: Výpočet parametru p_{t0} pro vrcholy $V_0 V_1$.

Samotnou projekci vrcholů provedeme pomocí skalárního součinu $p_i=\vec{p}_{dir}\cdot V_i$ ovšem to nám neposkytne finální projekci. Jak jsme si řekli jde nám o průnik trojúhelníků a přímky p a tedy je potřeba tyto projekce pozměnit tak, abychom získali parametry přímky p_{ti} přičemž takovéto parametry můžeme počítat pouze pro 2 vrcholy ležící na opačných poloprostorech plochy určené druhým trojúhelníkem. Jelikož v této fázi víme, že ne všechny vrcholy leží na jednom poloprostoru (pokud by ležely, tak už jsme je odmítly) tak musíme detekovat, které dva vrcholy leží na opačných stranách a to provedeme pomocí detekce opačných signatur, které jsme již vypočítali. Samotný parametr pak vypočítáme díky podobnosti trojúhelníků například pro vrcholy $V_0 V_1$ kde $K_0=\vec{p}_{dir}\cdot V_0$, $K_1=\vec{p}_{dir}\cdot V_1$, $d_0=(\vec{n}_{t1}\cdot V_0)-(\vec{A}_{t1}\cdot\vec{n}_{t1})$ a $d_1=(\vec{n}_{t1}\cdot V_1)-(\vec{A}_{t1}\cdot\vec{n}_{t1})$ kde

$$p_{t0}=K_0+(K_1-K_0)*(\frac{d_0}{d_0-d_1}) \quad .$$

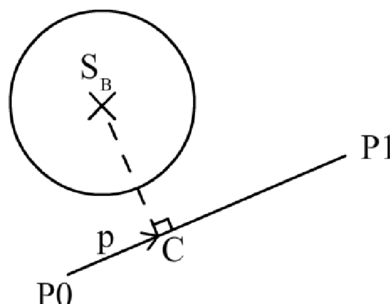
Pokud to takto spočteme pro všechny vrcholy, které leží na opačných poloprostorech, získáme tím dva intervaly, které pak můžeme porovnat, zda-li se nepřekrývají. Výsledek pak vracíme.

3.2.4 Koule a trojúhelník diskrétně

Pokud se budeme snažit například ve videohře reprezentovat hráče pomocí koule a budeme chtít testovat kolizi vůči nějaké 3D namodelované mapě a tedy trojúhelníkové síti, velmi by se nám hodil algoritmus na detekce kolize mezi jednotlivými trojúhelníky a samotnou koulí. Jak tedy k tomuto problému přistoupit? Řekněme, že B je naše koule a T je trojúhelník, vůči němu budeme testovat. Pokud bychom mohli vytvořit projekci středu koule S_B na trojúhelník T, mohli bychom zjistit, zda-li střed koule leží uvnitř trojúhelníku a na základě této informace bychom pak mohli se rozhodnou mezi dvěma variantami. Pokud projekce středu leží uvnitř trojúhelníku můžeme testovat vzdálenost d středu od nekonečné plochy definované trojúhelníkem T a pokud tato vzdálenost je menší než-li rádius koule r_B, pak můžeme tvrdit, že koule se dotýká trojúhelníku, přičemž výsledek kolize je pak daný rovnicí $v=r_B-d$ definující zanoření koule do trojúhelníku.

V opačném případě pokud projekce středu koule neleží uvnitř je třeba testovat kolizi všech tří hran trojúhelníku E_1, E_2, E_3 vůči kouli B. Kolize hrany neboli úsečky a koule se pak dále rozkládá na dvě části a to samotné kolize těla hrany a potom vrcholů hrany. Vrcholy hrany se testují velice jednoduše, neboť jde o kolizi bodu a koule, což lze vyřešit pouhým porovnáním vzdálenosti středu koule a tohoto bodu s rádiem koule a pokud tato vzdálenost je menší než-li rádius, dostáváme kolizi.

Se samotným tělem úsečky se musíme vypořádat jinak. Myšlenka je taková, že vypočítáme vzdálenost úsečky m dána středem koule S_B a bodem ležící na dané hraně E_i, přičemž tato úsečka m je kolmá na hranu E_i a tedy vlastně se jedná o nejmenší možnou úsečku mezi hranou E_i a středem koule S_B.

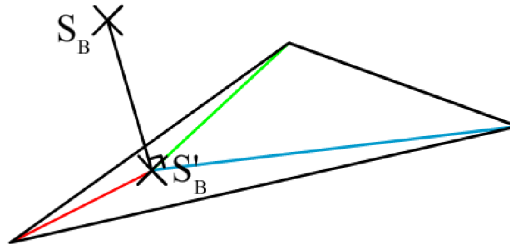


Obrázek 3.2.4.1: hledání bodu C pomocí projekce S_B na úsečku $P0P1$.

Tuto úsečku můžeme získat pomocí kombinace skalárního součinu a vektorového součinu tak, že nejdříve vytvoříme projekci středu koule S_B na danou hranu pomocí rovnice $p=(\widehat{P1_{Ei}-P0_{Ei}})\cdot(\overrightarrow{P0_{Ei}-S_B})$ což nám dá parametr úsečky E_i. Pokud tento bod leží uvnitř úsečky, pokračujeme v hledání úsečky m, v opačném případě přistoupíme ke kontrole vrcholů úsečky. Úsečku m pak lze lehce zkonstruovat pomocí parametru p a úsečky kde $C=P0_{Ei}+p*(\overrightarrow{P1_{Ei}-P0_{Ei}})$ což je ten vrchol na úsečce, který jsme hledali a druhý vrchol je střed koule S_B což nám dává novou úsečku CS_B. Získat délku úsečky $|CS_B|$ je pak už maličkost. Poté postupujeme podobně jako u trojúhelníkové plochy, kdy zjistíme, zda-li platí nerovnice

$|CS_B|<r_B$ a poté zjistíme zanoření koule do těla úsečky pomocí rovnice $v=r_b-|CS_B|$ což je potom výsledek detekce kolize.

Jak ale zjistit projekci bodu na trojúhelník a ještě navíc zjistit, zda-li bod leží uvnitř trojúhelníku? Naštěstí existuje algoritmus, který všechny tyto problémy vyřeší za nás a to je převod souřadnic 3D bodu reprezentovaný v kartézské soustavě souřadnic do barycentrických souřadnic trojúhelníku.

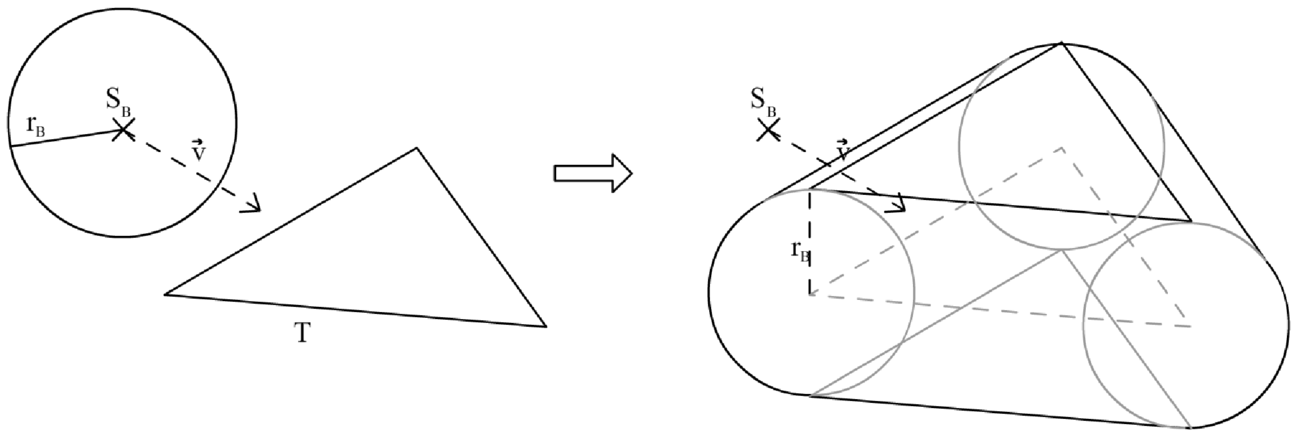


Obrázek 3.2.4.2: Projekce bodu na plochu trojúhelníku a definování barycentrických souřadnic bodu S_B .

Tento algoritmus funguje na tom principu, že nejdříve vytvoří 2D projekci 3D bodu na plochu definovanou trojúhelníkem a poté vypočítá jednotlivé barycentrické souřadnice. Tyto souřadnice poté můžeme lehce otestovat, zda-li všechny její složky jsou v intervalu $\langle 0,1\rangle$ a poté můžeme tvrdit, že projekce bodu leží uvnitř trojúhelníku a můžeme zjišťovat vzdálenost středu od plochy trojúhelníku, či kontrolovat hrany trojúhelníku, pokud střed uvnitř neleží. Implementace byla převzata z knihy Real-Time Collision Detection[6].

3.2.5 Koule a trojúhelník spojitě

Náš algoritmus ze sekce 3.2.4 bude fungovat bezchybně pokud změny v pozici dané koule budou dostatečně malé. Pokud rychlost bude příliš velká, může se stát, že daná koule proletí daným trojúhelníkem a to ve většině případů není žádoucí chování. U předchozích spojitých algoritmů jsme si poradili tak, že jsme daný problém převedli na kolizi mezi úsečkou a zvětšeným objektem. U tohoto problému takovýto přímý přístup nelze použít a budeme muset složit dohromady několik myšlenek dohromady a to detekci kolize s tělem trojúhelníku a poté detekce kolize s hranami trojúhelníku.



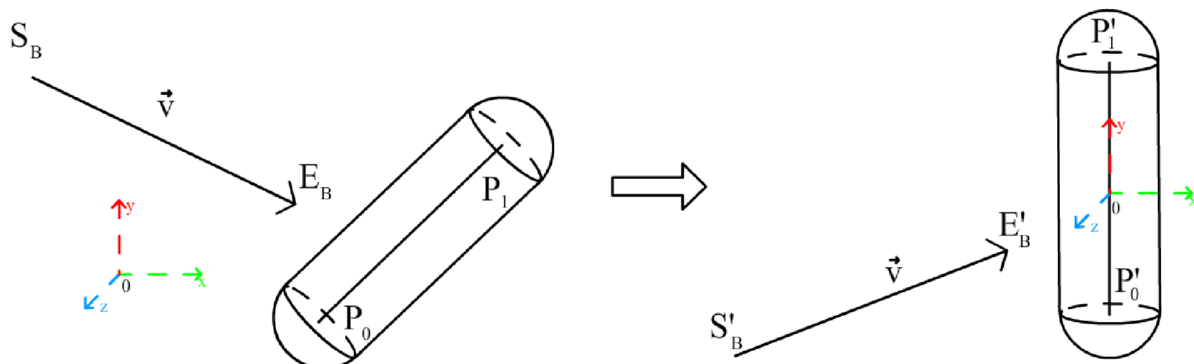
Obrázek 3.2.5.1: Transformace problému. Ve skutečnosti testujeme úsečku vůči dvěma trojúhelníkům a třem fazolím.

Jak vidíme v obrázku 3.2.5.1 kolize se rozděluje na několik podčástí a to kontrolování kolize úsečky a dvou trojúhelníkových ploch a poté se jedná o úsečku a tři fazolovité útvary což nám dohromady dává pět různých situací, kdy mohlo dojít ke kolizi. Jako v předchozích problémech uložíme si všech pět výsledků a poté vrátíme jako výsledek kolize ten, jehož bod kolize je nejblíže středu koule S_B .

Pojďme tedy se tedy nejdříve podívat na myšlenku a implementaci kolize úsečky a trojúhelníku. Nejdříve sestrojíme z trojúhelníku nekonečnou plochu, abychom zjistili zda-li úsečka trojúhelník vůbec protíná a tedy zjišťujeme bod protnutí úsečky a nekonečné plochy. Získávání plochy z trojúhelníku bylo popsáno v sekci 3.3.2. Bod protnutí se pak získá tak, že nejdříve zkontrolujeme, zda-li úsečka a plocha nejsou navzájem rovnoběžné. V takovémto případě pak bod protnutí nemůže existovat. Tuto kontrolu provedeme pomocí skalárního součinu $d=\vec{n}\cdot\widehat{(P_1-P_0)}$ a pokud je výsledek nulový pak to znamená, že úsečka je rovnoběžná s plochou a můžeme kolizi odmítnout. Je ovšem na pováženou jeden fakt a to ten, že pomocí této myšlenky můžeme zavrhnout takovou situaci, kdy úsečka a plocha se perfektně překrývají. Tuto situaci jsem se rozhodl ignorovat ovšem v některých speciálních situacích by mohl nastat problém nepřesnosti kolize. Pokud pak úsečku budeme vnímat jako přímku danou rovnicí $p=P_0+t*\widehat{(P_1-P_0)}$ pak hledáme parametr t , který definuje protnutí přímky a plochy. Tento parametr pak vypočítáme pomocí rovnice $t=\frac{\vec{n}\cdot(N-P_0)}{d}$. Tento parametr pak otestujeme, zda-li leží mezi body P_0,P_1 a dostáváme bod průniku plochy a úsečky. Nyní musíme otestovat, zda-li tento bod skutečně leží v trojúhelníku a to provedeme stejným způsobem, jak jsme si popsali v sekci 3.2.4 a to pomocí vypočítání barycentrických souřadnic tohoto bodu a kontrola, zda-li všechny složky barycentrických souřadnic leží v intervalu $\langle 0,1\rangle$.

Tento algoritmus provedeme pro oba trojúhelníky a můžeme přistoupit ke kontrole hran a tedy fazolí. Detekci kolize úsečky a fazole jsme si popsali v sekci 3.1.5 ovšem tato fazole byla zarovnaná podle os a neumožňovala aplikovat otočení na fazoli a pokud se podíváme na obrázek 3.2.5.1, tak otáčet tuto fazoli potřebujeme. V tomto bodě je možno využít triku, který se používá například při vykreslování 3D scény a konkrétněji při konverzi z world space do view space. Myšlenka je v tomto případě taková, že pokud se snažíme transformovat kameru, abychom se podívali na scénu z jiného úhlu, tak co se ve skutečnosti hýbe je všechno kolem kamery. Kamera ve skutečnosti zůstává na stejném místě, konkrétně v počátku soustavy souřadnic a všechny transformace aplikované na kameru, se ve skutečnosti reverzně aplikují na celou scénu.

Tuto myšlenku využijeme způsobem, že úsečku a fazoli transformujeme tak, že fazole bude vzpřímeně zarovnaná podle os a úsečku tomu přizpůsobíme.



Obrázek 3.2.5.2: Transformace problému abychom mohli využít již implementovaný algoritmus pro fazoli a úsečku.

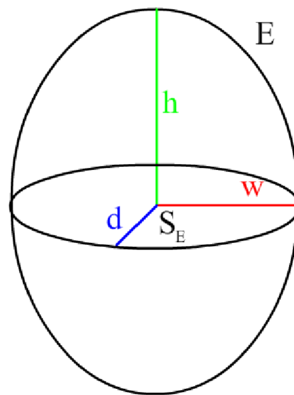
Tuto transformaci bychom mohli provést velice jednoduše, kdybychom měli rotační matici dané fazole, který ovšem nemáme a jeho spočítání by nebylo zrovna nejjednodušší. A proto jsem se rozhodl pro poněkud suboptimální řešení a to za pomocí eulerových úhlů. Ze všeho nejdříve bychom chtěli získat vektor $\vec{r}$ který je kolmý na vektor $\widehat{(P_1-P_0)}$. Pomocí knihovny GLM můžeme získat úhel mezi vektory úsečky a vertikálního jednotkového vektoru $a=glm::angle(\vec{(0,1,0)},\widehat{(P_1-P_0)})$. Pokud tento úhel se rovná nule či π radiánů pak vektor $\vec{r}$ se nedá lehce specifikovat a jsme tedy nuceni mu nastavit základní hodnotu, což jsem zvolil jako vektor $\vec{(1,0,0)}$. Pokud úhel a se nerovná speciálním hodnotám, pak vektor $\vec{r}$ se dá spočítat pomocí rovnice $\vec{r}=\vec{(0,1,0)}\times\widehat{(P_1-P_0)}$. Díky kombinaci úhlu a a vektoru $\vec{r}$ pak můžeme transformovat úsečky $S_B E_B$ a $P_0 P_1$ za pomocí další užitečné GLM funkce a to otočení vektoru pomocí úhlu a referenčního vektoru následujícím způsobem.

$$S'_B=glm::rotate(S_B-P_0,-a,\vec{r})$$
$$E'_B=glm::rotate(E_B-P_0,-a,\vec{r})$$
$$P'_0=(0,\frac{|P_1-P_0|}{2},0)$$
$$P'_2=(0,-\frac{|P_1-P_0|}{2},0)$$

Nyní můžeme využít spojité verze algoritmu 3.1.5, kterému tyto transformované parametry předáme a poté je výsledek transformovat zpět aplikováním inverzní transformace. Toto provedeme pro všechny tři hrany trojúhelníku a pošleme do řadícího algoritmu, který pak vybere kolizní bod, nejbližší středu koule S_B a to je výsledek kolize.

3.2.6 Elipsoid a trojúhelník

Elipsoid je rozšíření matematického modelu koule, přičemž nemá jenom jeden rádius, ale tři prvky definující výšku, šířku a délku.

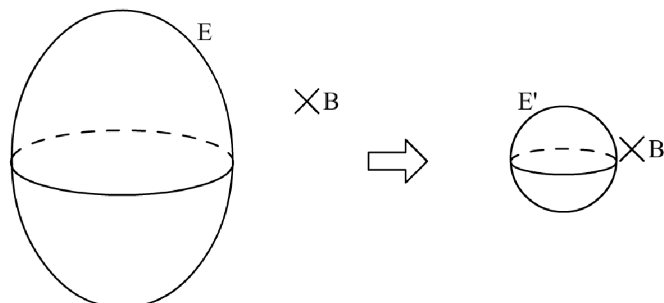


Obrázek 3.2.6.1: Definice elipsoidu.

```
struct Ellipsoid
{
    Ellipsoid() {}
    Ellipsoid(const glm::vec3& pos, const glm::vec3& dims) : pos(pos), dims(dims) {}

    glm::vec3 pos  { 0 };
    glm::vec3 dims { 0 };
};
```

Kód 3.2.6.2: Struktura elipsoidu.

Tento fakt při zpracování elipsoidů nám činí nemalé potíže, neboť u koule, který je s tímto útvarem velmi spjat, jsme řešili pouze rádius a nemuseli se starat o tři velmi nepěkné parametry. Co tedy s tím? Pokud by se nám podařilo elipsoid převést na kouli, pak bychom mohli využít všechny předchozí algoritmy, které s koulí pracují, ovšem jak? Pojďme si zjednodušit tento problém na zjištění, zda-li prostý 3D bod B je uvnitř elipsoidu E . Co můžeme udělat, je začít upravovat prostor vzhledem k elipsoidu tak, abychom ve výsledku z elipsoidu měli kouli jejíž rádius se rovná jedné, přičemž okolní scéna se vzhledem k těmto změnám také zdeformuje. Pokud máme parametry elipsoidu (w,h,d) pak se tedy snažíme vytvořit elipsoid s parametry $(1,1,1)$. To provedeme tak, že jednotlivé parametry elipsoidu vynásobíme $(\frac{1}{w},\frac{1}{h},\frac{1}{d})$. Tímto získáme jednotkovou kouli ovšem nesmíme zapomínat na náš bod B . Tento bod vůči elipsoidu transformujeme takto $B'=(x_{Se}+\frac{x_{Se}-x_B}{w},y_{Se}+\frac{y_{Se}-y_B}{h},z_{Se}+\frac{z_{Se}-z_B}{d})$, kde $S_E=(x_{Se},y_{Se},z_{Se})$ a $B=(x_B,y_B,z_B)$ což nám dává transformovaný bod B' , který pak můžeme testovat vůči jednotkové kouli tak, že musí platit $|B'Se|\leq 1$ jak jsme si již ukázali v předchozích sekcích.



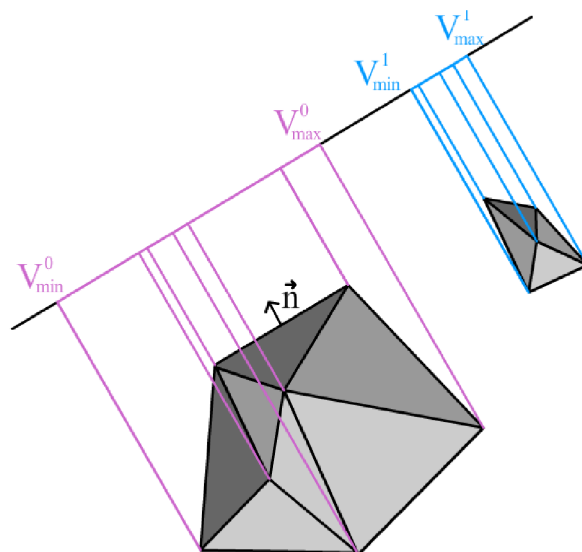
Obrázek 3.2.6.3: translace bodu a škálování elipsoidu na jednotkovou kouli podle jeho atributů.

Tuto myšlenku pak právě můžeme využít při detekci kolize mezi elipsoidem a trojúhelníkem, kde každý vrchol přetransformujeme pomocí formule specifikovanou výše, z elipsoidu vytvoříme jednotkovou kouli a všechny tyto argumenty pošleme algoritmu specifikovaný v sekci 3.2.4 nebo 3.2.5. Ovšem nesmíme zapomenout že výsledek těchto funkcí je deformován právě transformacemi prováděné nad jednotlivými body a elipsoidem samotným a tedy je nutné tento výsledek inverzně transformovat. Jelikož výsledkem kolize je vektor $R'=(x_R', y_R', z_R')$, pak skutečný vektor a výsledek kolize je $R=(x_R'*w, y_R'*h, z_R'*d)$.

3.3 Trojúhelníkové sítě

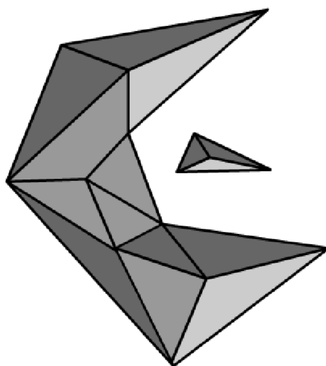
3.3.1 Separating axis theorem (SAT)

V tomto bodě jsme si vysvětlili detekci kolizí základních a předefinovaných geometrických objektů. Nicméně pokud dostaneme obecný útvar, se kterým bychom chtěli kolidovat, jako je například krajina, město či jakákoliv mapa, která se může skládat z milionů trojúhelníků a nelze dost dobře předpovídat útvar, který daný objekt by mohl nabývat, musíme zabrousit do obecnějších algoritmů na detekci kolizí. Jedním z takových algoritmů je takzvaný separating axis theorem[4]. Jeho základní myšlenka je následující: Nechť M_0, M_1 jsou objekty složené z trojúhelníků. Pak aby oba objekty M_0, M_1 se dotýkaly, pak musí pro každý trojúhelník T_i z M_0 (respektive M_1), musí platit následující: Vypočítáme-li normálu $\vec{n}_i$ trojúhelníku T_i a pomocí skalárního součinu p vytvoříme projekci všech trojúhelníků z M_0, M_1 na $\vec{n}_i$ dostaneme změť bodů, zprojektované na přímce u kterých pak podle příslušnosti k M_0 nebo M_1 musíme najít minimální a maximální bod. Tyto minima a maxima nám potom vytvoří dva intervaly a pokud se tyto dva intervaly překrývají ve všech trojúhelníkových normál, můžeme prohlásit, že oba objekty mají potenciál se dotýkat.



Obrázek 3.3.1.1: Projekce objektů na jednu trojúhelníkovou normálu.

Jak jsem řekl, je to pouze potenciál. Proč? Protože je ještě jedna podmínka, kterou musí oba objekty splňovat a to je, že oba objekty musí být konvexní. Pak můžeme tvrdit, že se oba objekty dotýkají.



Obrázek 3.3.1.1: Problém nekonvexních objektů. I přesto, že nenajdeme u jediného trojúhelníku mezeru mezi intervaly, oba objekty se nedotýkají. V tomto případě SAT nefunguje.

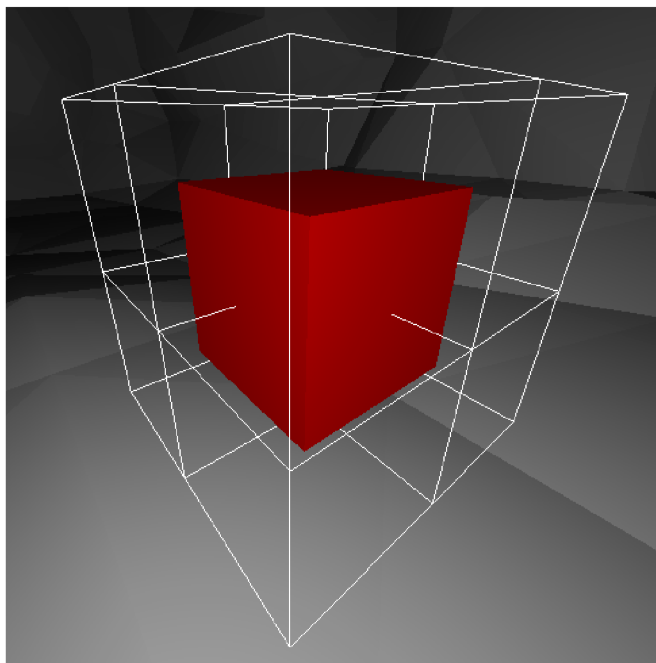
Kromě omezenosti, s jakými objekty tento algoritmus může pracovat, je s ním ještě jeden problém a to je algoritmická složitost. Jelikož musíme projít každý trojúhelník s každým dostáváme kvadratickou časovou složitost a trochu konkrétněji: $O(m^2 * n + n^2 * m)$! Což je velmi nežádoucí a špatně se tento algoritmus škáluje. V mé simulaci, jejíž žádoucí rychlost je šedesát snímků za sekundu, při dvou objektech které každý z nich měl 100 trojúhelníků simulace nedosahovala ani poloviny žádoucí rychlosti. Ovšem jedna z výhod tohoto algoritmu, je velice jednoduché hledání minimálního translačního vektoru, pomocí něhož pak můžeme oba objekty odseparovat. Při každé kontrole překrytí intervalů projekce objektů, můžeme si postupně ukládat a zjišťovat, při kterém trojúhelníku bylo zanoření nejmenší. S touto informací také ukládáme, při kterém trojúhelníku toto překrytí bylo nejmenší a pokud nenajdeme žádné dva intervaly, které by se nepřekrývaly a tedy prohlásíme, že se dva objekty dotýkají, pomocí těchto uložených informací můžeme velice jednoduše určit směr i velikost minimálního translačního vektoru a tento samý vektor můžeme vrátit jako výsledek detekce kolize.

3.3.2 Přístup hrubou silou

Hlavními problémy SAT jsou tedy algoritmická složitost a neschopnost řešit kolize konkávních objektů. Jak tedy k těmto problémům přistoupit, aniž bychom nevznítili náš procesor a lépe dokázali testovat kolize? Nejdříve k tomuto problému můžeme přistoupit naivním způsobem a to tak, že zkrátka otestujeme kolizi každého trojúhelníku s každým. Pokud využijeme algoritmu popsaný v části 3.2.3 můžeme pro každý trojúhelník z objektu O_1 otestovat s každým trojúhelníkem O_2 čímž vlastně vyřešíme problém neschopnosti testování konkávních objektů, protože testujeme každý trojúhelník individuálně. I přesto že jsme s časovou složitostí o zlomek pohnuli je bohužel stále kvadratická $O(m * n)$.

3.3.3 Rozdělení na podčásti

Jak se tedy vypořádat s touto velmi nepříjemnou časovou složitostí? Jedním ze způsobů je rozdělení objektu na podčásti a ze všeho nejdříve otestovat kolizi těchto podčástí a až poté testovat kolizi trojúhelníků v těch podčástech, které se dotýkají. Způsobů, kterým můžeme objekt rozdělovat na podprostory může být nepřeberné množství jako například octree, kde na základě složitosti objektu postupně rekurzivně rozdělujeme oblast na podčásti na základně složitosti objektu, či binary space partitioning, který též rekurzivně rozděluje oblast ale jako podčásti používá nadroviny. Každý algoritmus má své výhody a nevýhody. V mém přístupu jsem zvolil cestu zjednodušeného octree, přičemž jsem odstranil rekurzivní náturu algoritmu, takže počet podčástí je vždy pevně daný.



Obrázek 3.3.3.1: Rozdělení objektu na 8 podčástí.

Toto je zajištěno tak, že při načítaní a konstrukci objektu, celistvý útvar znormalizujeme. Tedy že největší rozměr v dané ose bude jedna a díky tomuto faktu také můžeme vytvoří ohraničující kvádr, který pak bude užitečný později. Poté projdeme všechny vrcholy trojúhelníků a zjistíme, do které podčásti patří a pak celý trojúhelník do této podčásti přiřadíme. Jednotlivé podčásti jsou pak reprezentované kvádry, které pak obsahují členství neboli odkazy na jednotlivé trojúhelníky daného objektu. S touto reprezentací podčástí pak můžeme přistoupit k samotné kolizi dvou obecných objektů. Ze všeho nejdříve zjistíme, zda-li ohraničující kvádry obou objektů se dotýkají. Kontrolu kolize kvádrů můžeme zajistit algoritmem popsaný v části 3.1.3 a pokud se dotýkají, pak můžeme přejít ke kontrole podčástí. V opačném případě kolizi zavrhujeme. Podobným způsobem pak kontrolujeme kolizi podčástí, kde pak vybereme kandidáta, či kandidáty, které se navzájem dotýkají a z nich extrahujeme jednotlivé trojúhelníky, kde pak provedeme kontrolu kolize stylem každý s každým.

Při rotaci daného objektu je pak potřeba přepočítat příslušnost daného trojúhelníku do dané podčásti, protože tyto podčásti jsou pevně dané a to může mít zásadní dopad na výkon programu. Tento problém by se dal vyřešit použitím takzvaných oriented bounding box (OBB) pomocí nichž bychom mohli otáčet i s jednotlivými podčástmi ovšem algoritmus na překrytí dvou OBB je o to složitější.

Abychom nepřímo zamezili zbytečnému kontrolování trojúhelníků či přepočítávání podčástí, při konstrukci trojúhelníkové sítě a tedy po načtení z pevného disku, můžeme vypočítat ohraničující krychli definující maximální hranici daného objektu. Předtím, než přistoupíme k samotné detekci kolize trojúhelníkových sítí, ať už je to SAT, každý trojúhelník s každým či rozdělení objektu na podčásti, můžeme nejdříve zjistit, zda-li se hraniční krychle překrývají pomocí algoritmu popsaný v sekci 3.1.3. Pokud ano přistupujeme k detailní detekci kolize, v opačném případě tuto kontrolu zahazujeme a tedy zbytečně nekontrolujeme kolizi, která nemůže nikdy nastat.

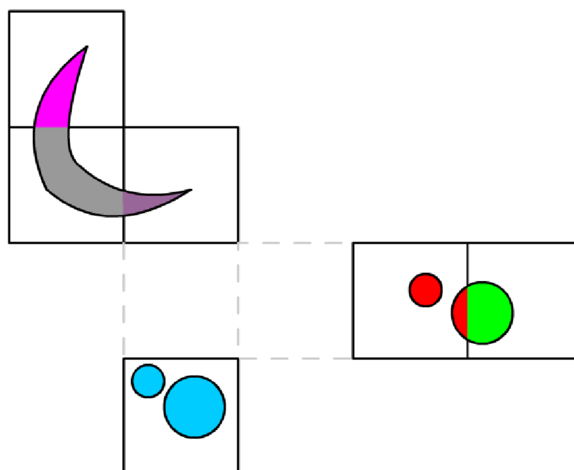
3.4 Dělení prostoru

3.4.1 Myšlenka

Pokud chceme detekovat kolize mezi jednotlivými tělesy, tak v simulačním programu musíme pro každé dva objekty volat funkci, která tuto kolizi vyřeší a dá simulaci vědět, v jakém stavu obě tělesa jsou. Ovšem tento úkon bez hlubšího zamyšlení dosahuje kvadratické algoritmické složitosti, jelikož je to kontrola kolize každý s každým objektem. A pokud v simulačním prostoru pracujeme s deseti tisíci různými objekty a simulace se například snaží běžet rychlostí šedesát snímků za sekundu i s velmi moderním počítačem bude mít v této situaci co dělat aby požadované rychlosti dosáhl. A proto bychom měli klást velký důraz na názor: “Pokud jsou objekty příliš daleko od sebe, neměli bychom je zahrnovat do naší kontroly kolize”. S touto myšlenkou bychom měli začít náš prostor dělit a optimalizovat tak, že bychom testovali kolizi jenom těch objektů, které jsou ve společném podprostoru. V dnešní době existuje mnoho způsobů jak k tomuto problému přistoupit, jako jsou například octtree, axis aligned bounding box tree či binary space partition tree. V této fázi jsem se snažil experimentovat s různými přístupy a po čase jsem přišel s vlastní variantou rozdělení prostoru na podprostory. Je nutné ovšem říci, že přesto se snažíme tento problém z kvadratické časové složitosti převésti na nižší, může se se stát, že jakýkoliv algoritmus může zdegenerovat zpět na kvadratickou časovou složitost, pokud dostatečné množství objektů bude stisknuto do jednoho podprostoru a tím pádem všechna námaha bude k ničemu. Je tedy také nutné správně nastavit velikosti jednotlivých podprostorů aby nebyli příliš velké nebo příliš malé.

3.4.2 Vlastní implementace

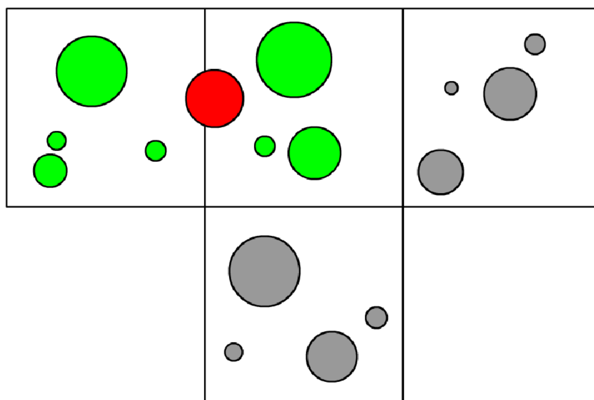
Moje hlavní myšlenka byla, aby samotná struktura řídící a ovládající rozdělování prostoru na podprostory zabírala co nejméně místa a aby byla zároveň rychlá. Místo konvenčních přístupů, kde se ve většině případů používá jako datová struktura strom, použil jsem ve své implementaci hešovací tabulku, kde klíč je pozice podprostoru a hodnota je kontejner reference objektů, patřící do podprostoru daný klíčem. Této struktuře je nejprve nutno předat velikost podprostoru S, na který prostor se má rozdělovat. Podprostor je pak krychle s velikostí právě o této zadané straně S. Tyto podprostory jsou pak přesně zarovnané podle mřížky.



Obrázek 3.4.2.1: Asociace objektů s podprostory.

Pokud tedy chceme přidat objekt do našeho manažeru, nejdříve musíme vypočítat do který podprostorů zasahuje. To uděláme tak, že vypočítáme hraniční kvádr daného objektu a pak můžeme využít algoritmus ze sekce 4.1.3 pro zjištění do kterých podprostorů daná obálka zasahuje. Pokud podprostor neexistuje je třeba ho vytvořit a pak do jeho seznamu objektů přidat referenci na daný objekt.

S tímto je pak celá struktura vytvořena a v této fázi se dá použít tak, že pokud máme dynamický objekt, který chce nějakou formou interagovat s ostatními objekty spravované naší hašovanou strukturou, předáme jí pozici objektu a ona nám vrátí všechny objekty v blízkosti daném číslem S .

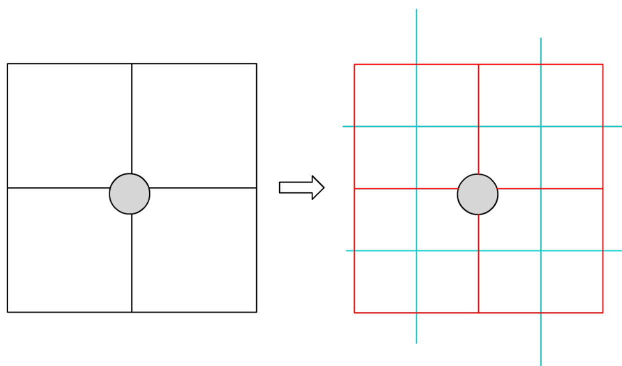


Obrázek 3.4.2.2: Červený objekt je ten, podle něhož získáváme blízké objekty. Zelené jsou pak ty, které se vrátí jako jeho nejbližší sousedi a s těmito pak červený objekt může interagovat.

3.4.3 Porovnání

Pokud se zaměříme a začneme porovnávat například se strukturou octree, zjistíme, že octtree má mnohem dynamičtější velikost podprostorů a tím pádem také dokáže vytvářet vlastní velikosti podprostorů a tedy dokáže pracovat s objekty různých velikostí a tvarů. Ovšem pro přístup k danému podprostoru je třeba logaritmická složitost, kdežto s použitím hešovací tabulky ve většině případů dostaneme složitost konstantní. U mého přístupu je také potřeba předat hešovací struktuře funkci, která z daného objektu extrahuje hraniční kvádr, pomocí něhož pak struktura zjišťuje, zda-li se daný objekt vyskytuje v daném podprostoru.

Jedno z potenciální vylepšení mé hašovací struktury byla myšlenka, že struktura by nebyla reprezentována pouze jednou mřížkou ale dvěma navzájem vychýlenými mřížkami, přičemž objekt by byl přiřazen do obou mřížek.



Obrázek 3.4.3.1: Nalevo kulatý objekt musí být přiřazen do čtyř podprostorů, napravo tomu je taktéž ovšem to platí pouze pro červenou mřížku, v modré mřížce zasahuje pouze do jednoho podprostoru a tedy získáme přesněji sousedy objektu.

S tímto přístupem by se pak stávalo méně častěji, že objekt se vyskytuje na hranici daného podprostoru a musí se vracet více podprostorů, než je třeba. Nicméně toto vylepšení by se určitě projevilo v zabraném prostoru, co se spravování podsekcí týče a to tak, že by se alokovaná paměť minimálně zdvojnásobila.

3.5 Implementační problémy

Chyby a problémy nastávají v jakémkoliv typu softwaru a tato práce není výjimkou. V celistvé implementaci se objevilo nemálo problémů a chyb, které mohou narušit chod programu a bylo nutné se jim přizpůsobovat a ošetřit je, jak je to nejlépe možné.

3.5.1 Čísla s plovoucí řádovou čárkou

Jedním z největších problémů tohoto projektu, je práce s plovoucí řádovou čárkou a to konkrétně porovnávání čísel s plovoucí řádovou čárkou. Tento problém například je relevantní v sekci 3.2.2, kde se snažíme zjistit, zda-li dvě plochy jsou navzájem rovnoběžné. V tomto algoritmu porovnáváme výsledek používající plovoucí řádovou čárku s nulou, přičemž můžeme si všimnout toho faktu, že tento výsledek používáme jak dělitel v dalších krocích tohoto algoritmu. Co se může stát, že výsledek může být velmi blízko nule, ovšem nule se přímo nerovná a pokud toto velmi malé číslo použijeme jako dělitel v určitých formulích, může se na některých platformách stát, že jednotka na počítání nedokáže vypočítat výsledek s takto malým dělitelem protože by byl příliš velký a tedy může v programu vyhodit výjimku o dělení nulou. Tento problém se dá řešit několika způsoby.

Pokud se budeme držet algoritmu ze sekce 3.2.2, můžeme celistvý algoritmus obalit zachytáváním výjimky dělení nulou. V tomto případě bychom pak mohli říci, že obě plochy jsou navzájem rovnoběžné i přesto, že daný výsledek nebyl nulový. Nicméně zpracování výjimek není výkonnostně nejlepší metoda, obzvlášť v algoritmu u kterého nám jde o co možná nejrychlejší provedení.

Jiný způsob, jak k tomuto problému přistoupit je zaokrouhlování či převod na celá čísla. Toto si samozřejmě nemůžeme vždy dovolit pokud nám jde o určitou přesnost, ale jako alternativa se může hodit.

Poslední přístup, který můžeme k tomuto problému přistoupit, je vytvořit vlastní porovnávací funkci, která kromě dvou čísel x, y by brala navíc parametr r, který by definoval rozsah, o kolik se vstupní hodnoty mohou lišit a pak bychom mohli vracet booleovskou hodnotu na základě nerovnice

$$|x-y|<r \quad .$$

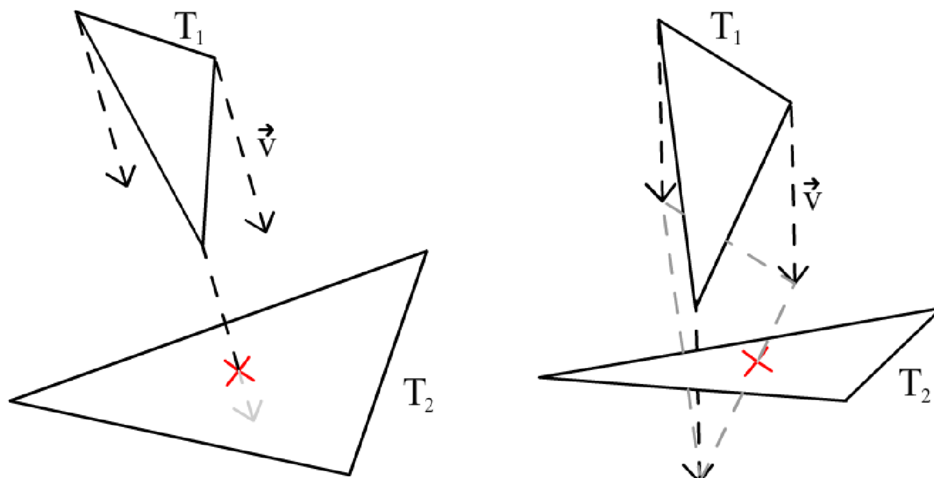
Jak si ale můžeme všimnout, takováto funkce může přidávat nemálo procesorových cyklů, ovšem je vysoce škálovatelná a velmi příjemná na použití.

3.5.2 Spojitá kolize dvou trojúhelníkům

V této práci jsem se snažil pro každý diskrétní kolizní algoritmus vytvořit jeho spojitou variantu. Ovšem můžeme si všimnout, že spojitá verze pro kolizi trojúhelníků bohužel chybí. Literaturu jsem na toto téma bohužel nenašel, čili jsem se snažil k tomuto problému přistoupit samostatně a i přesto, že se mi podařilo dostat tento algoritmus do funkčního stavu, detekce kolizí nebyla přesná a už rozhodně ne spolehlivá. V jiných spojitých metodách jsme mohli využít převedení na jiné útvary, či jsme použili úsečky a objektu jehož velikost byla součtem velikostí obou testovaných objektů. Pokud bychom aplikovali stejné myšlenky zde, vždy nás to zavede do slepé uličky neboť výsledné převedené útvary na testování jsou příliš složité a nelze na ně aplikovat žádná rozumná pravidla.

Prvotní myšlenka byla kdy vlastně ke spojité kolizi trojúhelníků T_1, T_2 přičemž trojúhelník T_1 má rychlostní vektor $\vec{v}$, může vůbec dojít a kdy můžeme dopředu odmítnout kolizi. Podobně jako v algoritmu v sekci 3.2.3 tak nejdříve z trojúhelníků vytvoříme nekonečné plochy P_1, P_2. Definujme si, že pohybující se trojúhelník T_1 se skládá z vrcholů V_1, V_2, V_3. Potom pokud jakýkoliv z vektorů $\vec{v_1}=V_1+\vec{v}, \vec{v_2}=V_2+\vec{v}, \vec{v_3}=V_3+\vec{v}$ protne plochu P_2 existuje potenciální kolize a v opačném případě kolizi zavrhujeme. Nyní ale právě vstupujeme do situace, kde musíme skutečně rozhodnout, zda-li kolize nastala či ne, a v tomto místě jsem narazil na mnoho slepých uliček.

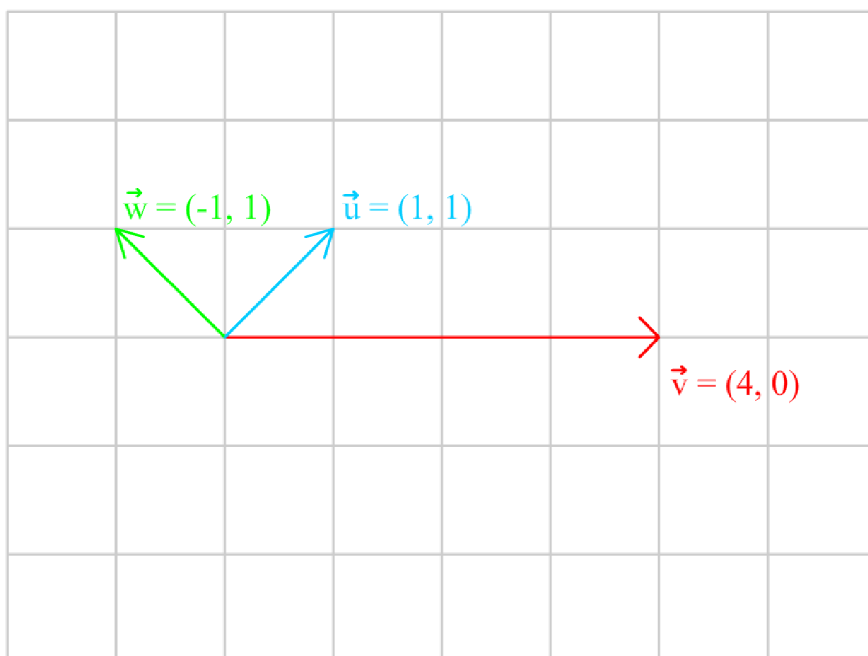
Jeden z pokusů, jak zjistit zda-li trojúhelník T_1 penetroval na své cestě trojúhelník T_2 byl následující: Pokud jeden z vektorů $\vec{v_1}, \vec{v_2}, \vec{v_3}$ proťal plochu P_2 a zároveň bod definující protnutí jednoho z vektorů $\vec{v_1}, \vec{v_2}, \vec{v_3}$ a plochy P_2 leží uvnitř trojúhelníku T_2, pak kolize musela nastat.



Obrázek 3.5.2.1: První technika spojitého protnutí dvou trojúhelníků a problém, u kterého sice nastane kolize, ale nejsme ji schopni detekovat.

Tato myšlenka ovšem zahrnuje pouze speciální případ protnutí a jak vidíme v obrázku 3.6.2.1, existuje situace, kdy kolize nastane, ale nejsme ji schopni touto metodou detekovat v případě, že to co se protlo jsou pouze hrany trojúhelníků T_1, T_2 . Druhý problém s touto metodou je, pokud jednotlivé vrcholy trojúhelníku T_1 leží na opačných poloprostorech plochy P_2 , jinými slovy že P_2 již protíná trojúhelník T_1 , kolize bude velmi těžko detekovatelná. Tento problém se dá ovšem jednoduše vyřešit tak, že vyměníme role trojúhelníků tak, že trojúhelník T_2 bude mít opačný rychlostní vektor $-\vec{v}$ a trojúhelník T_1 bude figurovat jako statický.

Jak tedy nyní řešit spojité kolize hran? Definujme si hrany E_1, E_2 přičemž E_1 má rychlostní vektor $\vec{v}$. První pokus jak tuto kolizi vyřešit bylo pomocí kombinace najití nejmenší vzdálenosti mezi dvěma úsečkami a skalárního součinu a jeho schopnost vyjadřovat vztah směrů dvou vektorů a konkrétně co ve skalárním součinu vyjadřuje znaménko výsledku. Znaménko samotné nám dokáže vyjádřit, na jaké straně vůči sobě dva vektory jsou. Myšlenka byla taková, že pokud startující pozice hrany E_1 existuje na jednom poloprostoru plochy P_2 a konečná pozice hrany $E_1+\vec{v}$ je na druhém poloprostoru plochy P_2 , pak teoreticky by měla existovat kolize mezi hranami E_1, E_2 .



$\vec{u} \cdot \vec{v} = 4 \qquad \vec{w} \cdot \vec{v} = -4$

Obrázek 3.5.2.2: Skalární součin a znaménko výsledku na základě poloroviny

Pokud se nám podaří najít úsečku U_1 definující nejmenší vzdálenost mezi hranami E_1, E_2 a nejmenší úsečku U_2 mezi hranami $E_1 + \vec{v}, E_2$, můžeme pak využít skalárního součinu pro zjištění na kterém poloprostoru se vrcholy úseček U_1 a U_2 nacházejí a pokud jsou na opačných poloprostorech, je možné, že nastala kolize.

Jak tedy zjistit nejmenší úsečku mezi dvěma přímkami? Definujme si přímky jako $p_1(t) = E_{1start} + t(E_{1end} - E_{1start})$ a $p_2(d) = E_{2start} + d(E_{2end} - E_{2start})$, kde $E_{1start}, E_{1end}, E_{2start}, E_{2end}$ jsou vrcholy hran a t, d jsou parametry přímky a $Q(t,d)$ je úsečka jejíž vrcholy leží na přímkách p_1, p_2 a jsou definované parametry t, d. Aby úsečka byla mezi dvěma přímkami nejmenší musí nutně být pravoúhlá na obě přímky. Díky tomuto faktu můžeme vytvořit soustavu rovnic tak, že dva vektory na sebe pravoúhlé jsou tehdy, pokud jejich skalární součin je nula

$$\overrightarrow{(E_{1end} - E_{1start})} \cdot Q(t,d) = 0$$
$$\overrightarrow{(E_{2end} - E_{2start})} \cdot Q(t,d) = 0$$

, přičemž úsečku $Q(t,d)$ lze vyjádřit jako $Q(t,d) = p_1(t) - p_2(d)$. Nejdříve si vytvoříme substituce $\overrightarrow{(E_{1end} - E_{1start})} = \vec{e}_1$ a $\overrightarrow{(E_{2end} - E_{2start})} = \vec{e}_2$. Pak po těchto substitucích dostáváme rovnic

$$\vec{e}_1 \cdot (p_1(t) - p_2(d)) = \vec{e}_1 \cdot (\overrightarrow{(E_{1start} - E_{2start})} + t\vec{e}_1 - d\vec{e}_2) = 0$$
$$\vec{e}_2 \cdot (p_1(t) - p_2(d)) = \vec{e}_2 \cdot (\overrightarrow{(E_{1start} - E_{2start})} + t\vec{e}_1 - d\vec{e}_2) = 0 \quad .$$

Tato soustava rovnic se dá dále přepsat jako

$$(\vec{e}_1 \cdot \vec{e}_1) t - (\vec{e}_1 \cdot \vec{e}_2) d = -(\vec{e}_1 \cdot \overrightarrow{(E_{1start} - E_{2start})})$$
$$(\vec{e}_2 \cdot \vec{e}_1) t - (\vec{e}_2 \cdot \vec{e}_2) d = -(\vec{e}_2 \cdot \overrightarrow{(E_{1start} - E_{2start})}) \quad .$$

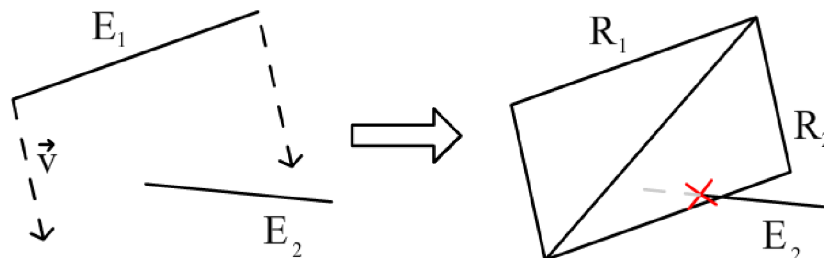
Tato výsledná soustava se dá vyřešit pomocí Cramerova pravidla

$$t=\frac{bf-ce}{d}$$
$$d=\frac{af-bc}{d} \quad ,$$

kde $a=\vec{e_1}\cdot\vec{e_1}, b=\vec{e_1}\cdot\vec{e_2}, c=\vec{e_1}\cdot(\overrightarrow{E_{1start}-E_{2start}}), e=\vec{e_2}\cdot\vec{e_2}, f=\vec{e_2}\cdot(\overrightarrow{E_{1start}-E_{2start}})$ a $d=ae-b^2$. Pokud platí podmínka $d=0$, znamená to, že hrany E_1, E_2 jsou paralelní a museli bychom tento stav ošetřit jinak. Díky těmto spočteným parametrům pak můžeme jednoduše definovat body na přímkách p_1, p_2 , které nám potom vytvoří nejmenší možnou úsečku Q .

Toto řešení ovšem detekuje kolize tak, že hrany E_1, E_2 bere jako nekonečné přímky což znamená, že i přesto že kolize mezi dvěma trojúhelníky neexistuje, tento přístup může vrátit výsledek kolize jako pozitivní. Je pravda, že nejmenší úsečku mezi hranami lze omezit jejími vrcholy, nicméně tato úprava nemá na výsledné znaménka žádný vliv. A tedy bylo nutné tohoto přístupu se vzdát a pokusit se o něco jiného.

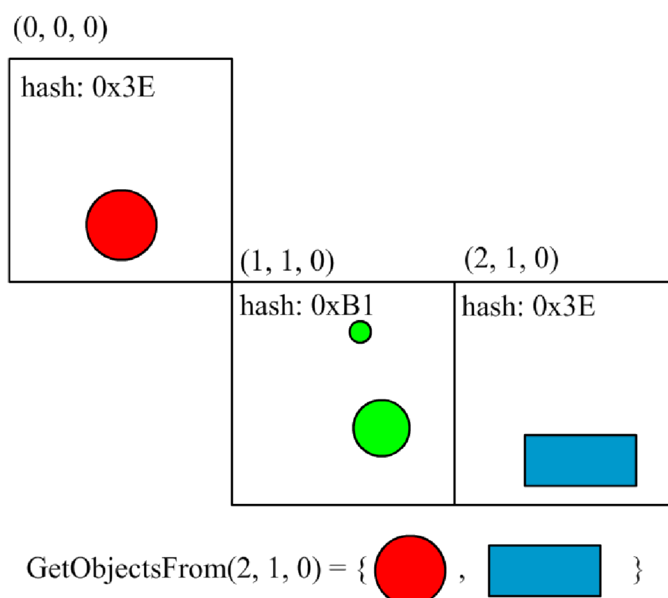
Další přístup, co jsem použil bylo převedení problému kolize dvou úseček na kolizi dvou trojúhelníků a úsečky. Pohybující se hrana E_1 a její konečná hrana $E_1+\vec{v}$ tvoří rovnoběžník R , který můžeme vyjádřit pomocí dvou trojúhelníků R_1, R_2 . V tomto bodě potom můžeme použít algoritmus na hledání bodu kolize trojúhelníku a úsečky ze sekce 3.2.5, kde zjistíme kolizi mezi R_1 a E_2 nebo R_2 a E_2 . Pokud najdeme kolizní bod, můžeme pak říct, že kolize mezi úsečkami nastala. I přesto že tento přístup vypadal velice slibně, z důvodů neznámých se mi nepodařilo dostat algoritmus do funkčního stavu a trojúhelníky ač se protínaly, kolize detekována nebyla.



Obrázek 3.5.2.3: Převedení problému kolize dvou úseček na kolizi dvou trojúhelníků a úsečky

3.5.3 Kolize hešů podčástí scény

V sekci 3.4 jsme si ukázali jednu z metod na dělení prostoru na podprostory, přičemž jako hlavní datová struktura, která je stavebním kamenem této myšlenky je hešovací tabulka, přičemž heš se počítá z 3D pozičního bodu. Ať se nám to líbí, nebo ne, může se stát, že pro dvě různé souřadnice se vypočítá stejný heš. Tento ošklivý fakt má pak dopad na získávání objektů z podčástí prostoru a to ten, že pokud v hešovací tabulce máme obě podčásti s kolidujícím hešem a budeme chtít získat objekty z jedné nebo druhé podčásti, vždy se vrátí všechny objekty z obou podčástí. Samozřejmě tento problém čistě závisí na kvalitě heše, který ze složek 3D pozičního bodu vytváříme. GLM knihovna naštěstí nabízí velmi pěknou vestavěnou hešovací funkci přímo pro daný 3D poziční bod.



Obrázek 3.5.3.1: Kolize dvou hešů, výsledkem je pak získávání objektů z úplně jiného podprostoru.

3.7 Použité knihovny a nástroje

Aby bylo pro mě příjemnější implementovat jednotlivé kolizní funkce a pak je také demonstrovat, použil jsem několik knihoven, které mi mnohonásobně usnadnili implementaci projektu. V prvé řadě to je knihovna GLM (OpenGL Mathematics), která implementuje vektorové a maticové struktury a plně využívá předefinování operátorů pro snadný zápis vektorových a maticových výrazů. Tato knihovna také definuje různé utilitní funkce pro práci s těmito strukturami jako je například rotace vektoru, či skalární součin kde může využívat vektorové SIMD optimalizace pro zrychlení operací. Dále pro samotné renderování 3D scény jsem použil renderovací knihovnu OpenGL a pro interakci s operačním systémem, management okna a zpracovávání událostí jsem použil knihovnu SDL2. Celý projekt je pak napsaný v C++17. Můj renderovací engine používá všechny tyto knihovny a myšlenky pro vizualizaci výsledků a detekcí kolizí jednotlivých objektů a funkcí. Projekt se mi podařilo zkompilovat a spustit na systémech Windows 10 a Linux 4.20 distribuce Manjaro. Jako projektový a kompilační systém jsem použil cmake na Linux systému a Visual Studio na Windows systému.

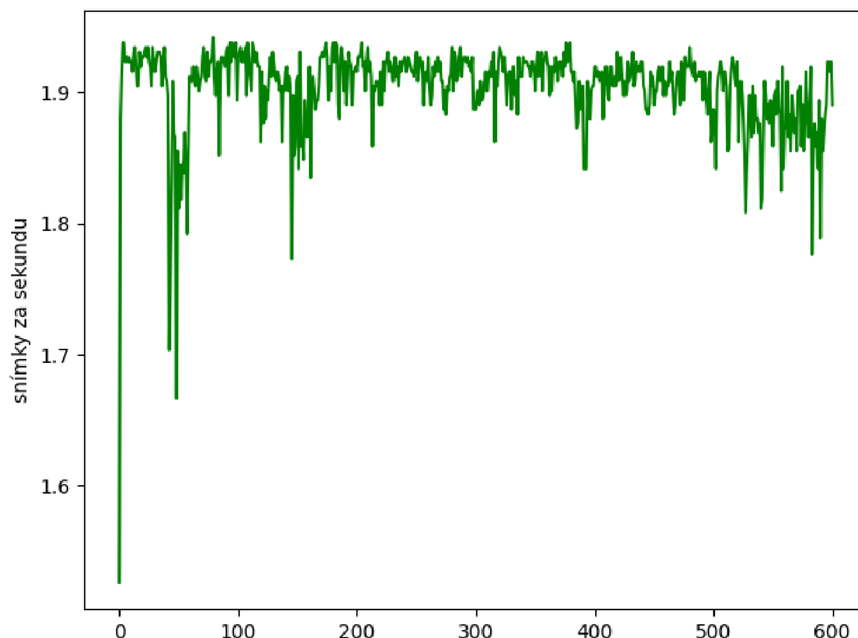
4. Měření

V mém enginu jsou dvě možnosti jak nastavit rychlost požadované simulace. Buďto se pevně nastaví požadovaná rychlost snímků za sekundu a celý engine se pod touto pevnou hodnotou ohýbá a nebo můžeme nastavit mód, kdy jednotlivé simulační kroky se dynamicky mění na základě požadované frekvence snímků jak bylo popsáno v sekci 2.1. Všechna měření byli provedeny v módu dynamické frekvence snímků, což jako výstup statistiky je stabilita a počet snímků za sekundu. Všechna měření byly prováděny v mé demonstrační aplikaci na počítači s procesorem AMD Ryzen 5 4600HS, RAM pamětí 16GB DDR4 a grafickou kartou NVIDIA GeForce RTX 1650. V této sekci je testování optimalizací algoritmů ukázány v předchozích sekcích.

4.1 Rozdělení prostoru na podprostory

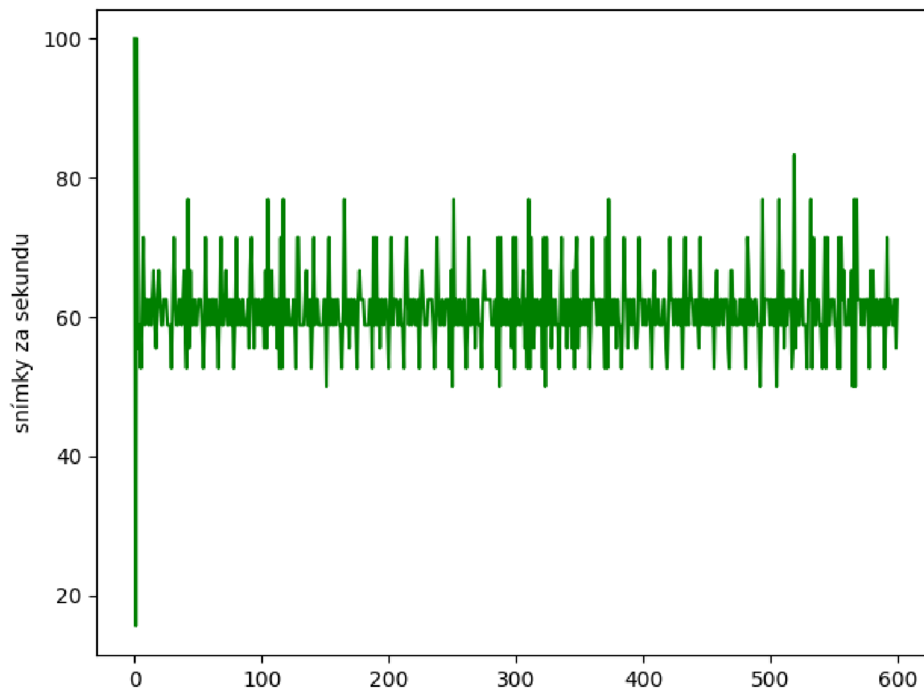
Nejdříve se podívejme na simulaci scény kde budeme mít scénu o pěti tisících koulích, které kolidují mezi sebou navzájem. Pro detekci kolize použijeme algoritmus ze sekce 3.1.1. Tato scéna a

samotná detekce kolize pak má z hlediska algoritmické složitosti kvadratickou časovou náročnost a pokud změříme počet vykreslených snímků za sekundu dostaneme nepřekvapující výsledek.



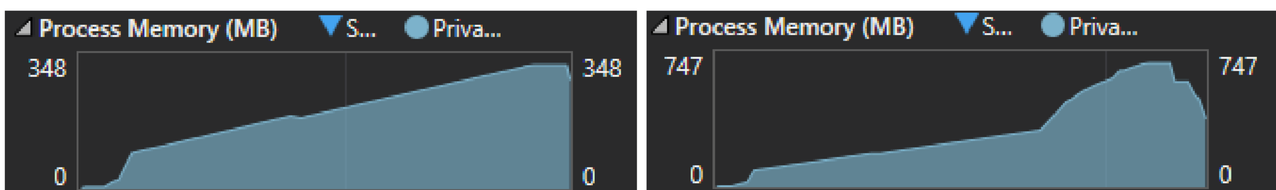
Obrázek 4.1.1: Simulace pěti tisíců koulí a jejich vzájemná detekce kolizí.

Jak můžeme vidět na obrázku 4.1.1 frekvence snímků se pohybuje kolem 1.9 snímku za sekundu což je velmi daleko od optima, pokud si stanovíme že referenční frekvence zpracování snímků je 60 snímků za sekundu. Simulace je velmi pomalá, táhne se, přičemž děláme kontroly detekce kolizí úplně zbytečných objektů, které nemají šanci se dotýkat. Pokud použijeme optimalizace rozdělení prostoru na podprostory vysvětlené v sekci 3.4 a začneme spravovat objekty tak že je do těchto podprostoru začneme přiřazovat a zjišťovat, které objekty jsou v tom samém podprostoru, dostaneme příjemné zlepšení



Obrázek 4.1.2: Simulace deseti tisíce koulí, za použití rozdělování prostoru na podčásti.

Nyní vidíme, že simulace velmi pohodlně zvládá dosáhnout požadované frekvence snímků, a pokud bychom je nelimitovali hranicí, pak by tento limit dokonce mnohonásobně převýšil. Ovšem tento výkon byl způsoben nejenom použitím této techniky, ale rozumné nastavení velikosti podprostoru. Pokud tento rozměr nastavíme příliš veliký tak, že se do jednoho podprostoru vejde všech deset tisíc koulí, přičemž velmi rychle zjistíme, že rozdělování prostoru na podprostory bylo úplně zbytečné neboť celý problém zdegeneruje na počáteční stav. Tedy že algoritmická složitost bude znovu kvadratická. Pokud naopak zvolíme velikost podprostoru jako příliš malý, konstrukce podprostorů a spravování jednotlivých objektů bude poněkud složitější a to obzvlášť vytváření samotných podprostorů a z hlediska zabrané paměti je toto řešení daleko od ideálního i přesto, že rychlost zůstane více méně stejná.

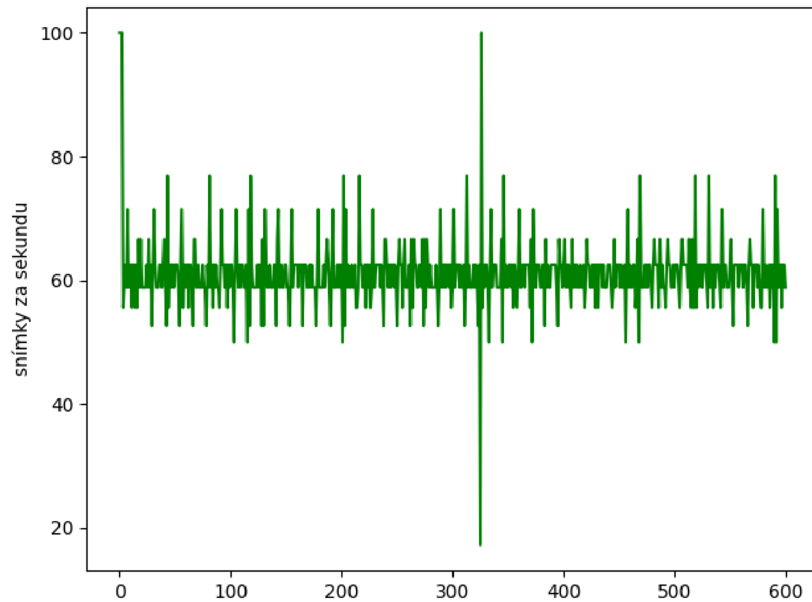


Obrázek 4.1.3: Porovnání procesorové paměti s normální velikostí (vlevo) podprostorů a s příliš malou velikostí (vpravo) podprostorů.

4.2 Trojúhelníkové sítě

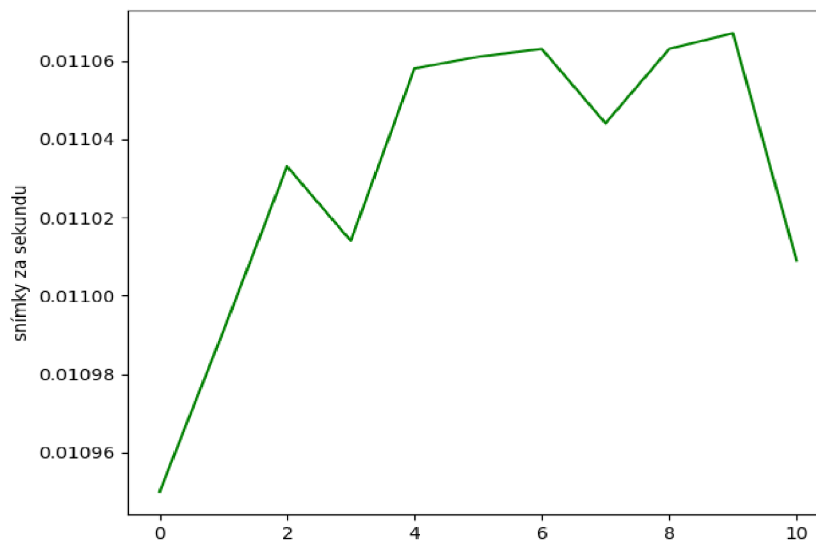
Další optimalizace, kterou jsem se v této práci zabýval, je kolize samotných trojúhelníkových sítí. V tomto měření jsem testoval kolizi několika objektů složených z trojúhelníkových sítí, přičemž každá trojúhelníková síť měla různý počet trojúhelníků a různé tvary. Nejprve se podívejme

jak rychle dokáže detekovat kolizi technika Separating Axis Theorem popsán v sekci 3.3.1. První test byl prováděn s dvěma krychlemi každá složená z dvanácti trojúhelníků.



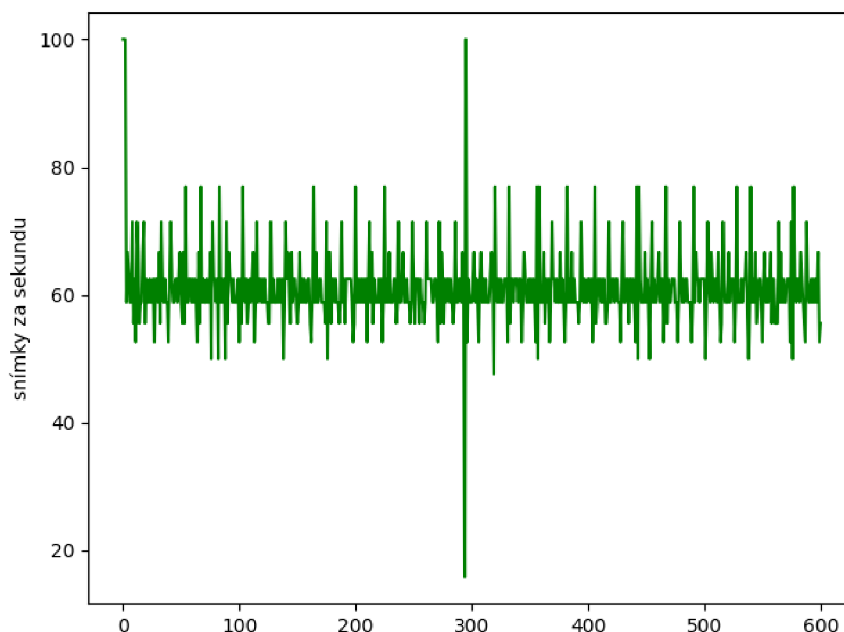
Obrázek 4.2.1: Kolize dvou krychlí pomocí Separating Axis Theorem algoritmu.

Můžeme si všimnout, že simulace si s tímto problémem hravě poradí, čili je třeba využít složitějšího objektu, který skutečně demonstruje velmi špatnou škálovatelnost tohoto algoritmu. V dalším měření jsem tedy testoval kouli tvořenou z 3,968 trojúhelníků s krychlí z předchozího měření a výsledek byl nad očekávání velmi nepěkný.



Obrázek 4.2.2: Kolize krychle a koule pomocí Separating Axis Theorem algoritmu.

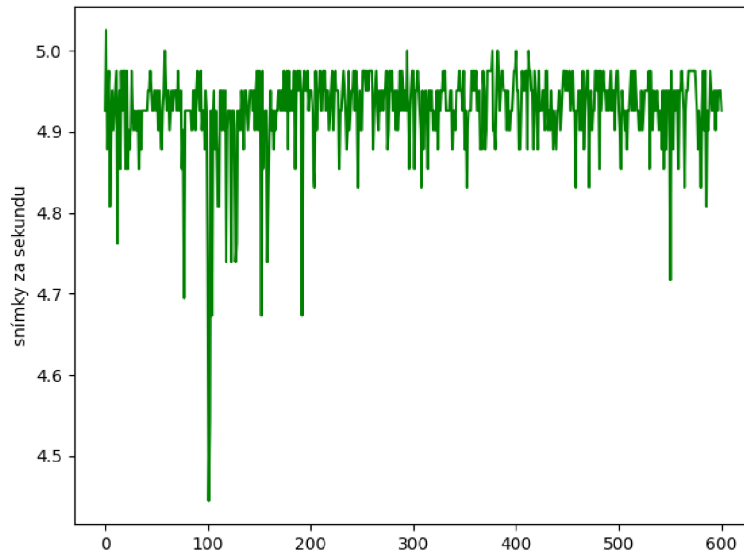
Jak si můžeme všimnout, stačí jenom aby jeden z daných objektů byl složitý a celistvé řešení tohoto problému se tím mnohonásobně prodlouží, nehledě k tomu, že tyto objekty musí být konvexní aby byly v tomto algoritmu použitelné. Proto v dalším měření jsem se zaměřil na detekci kolize jednotlivých trojúhelníků popsané v sekci 3.2.3. Protože algoritmická složitost tohoto algoritmu je stále kvadratická, očekávali bychom, že až tak lepší oproti Separating Axis Theorem algoritmu nebude. Pokud ovšem trojúhelníkovému algoritmu předáme ty samé objekty, tedy kouli a krychli dostaneme zajímavý výsledek.



Obrázek 4.2.3: Kolize krychle a koule za pomocí testování kolize trojúhelníků.

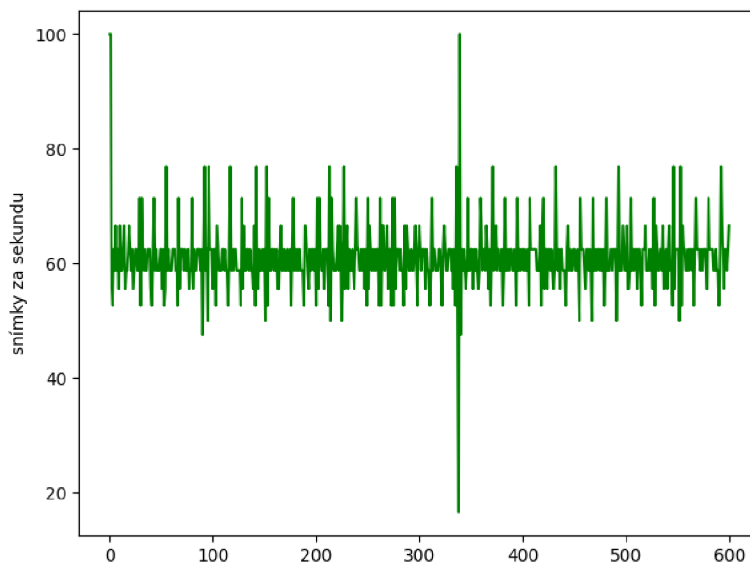
V obrázku 4.2.3 můžeme vidět, že tento algoritmus je časově na tom mnohonásobně lépe, než-li Separating Axis Theorem algoritmus. Jak je to možné? Oproti předchozímu algoritmu, tento algoritmus můžeme aplikovat pouze z pohledu jednoho objektu vůči druhému objektu. V Separating Axis Theorem musíme kontrolu provést z obou stran, tedy že zkontrolujeme, zda-li neexistuje mezera mezi prvním a druhým objektem a hned poté musíme zkontrolovat existenci mezery mezi druhým a prvním objektem, což má velmi negativní dopad na škálovatelnost.

Pokud ovšem začneme kontrolovat kolizi dvou koulí s tímto přístupem pak už dle očekávání dostáváme horší výsledky.



Obrázek 4.2.4: Kolize dvou koulí za pomocí testování kolize trojúhelníků.

Vidíme v obrázku 4.2.4 že kvadratická časová složitost odvádí svoji práci znamenitě a proto by v žádném projektu by neměla být využívána a abychom se snažili vytvářel lepši algoritmy, obzvlášť pokud budeme testovat několik takovýchto koulí zároveň. Jak tedy vypadá naše optimalizace s podčástmi? Pokud otestujeme dvě koule přičemž počet podčástí objektu je 8. Díky tomu, že ze všeho nejdříve testujeme kolizi podčástí tvořené z krychlí a objekty se nedotýkají, pak samozřejmě vůbec nemusíme žádné kolize trojúhelníků testovat a i když se dvě podčásti dotýkají, kontrolujeme pouze podmnožiny trojúhelníkových sítí a tedy výkon nám masivně vzroste.



Obrázek 4.2.5: Kolize dvou koulí za pomocí dělení objektu na podčásti.

Pokud ovšem podčást jednoho objektu zasahuje do vícero podčástí objektu druhého, tedy že jsou objekty v sobě zanořeny, může se stát, že výkonnost prudce klesne. Toto je způsobeno tím, že

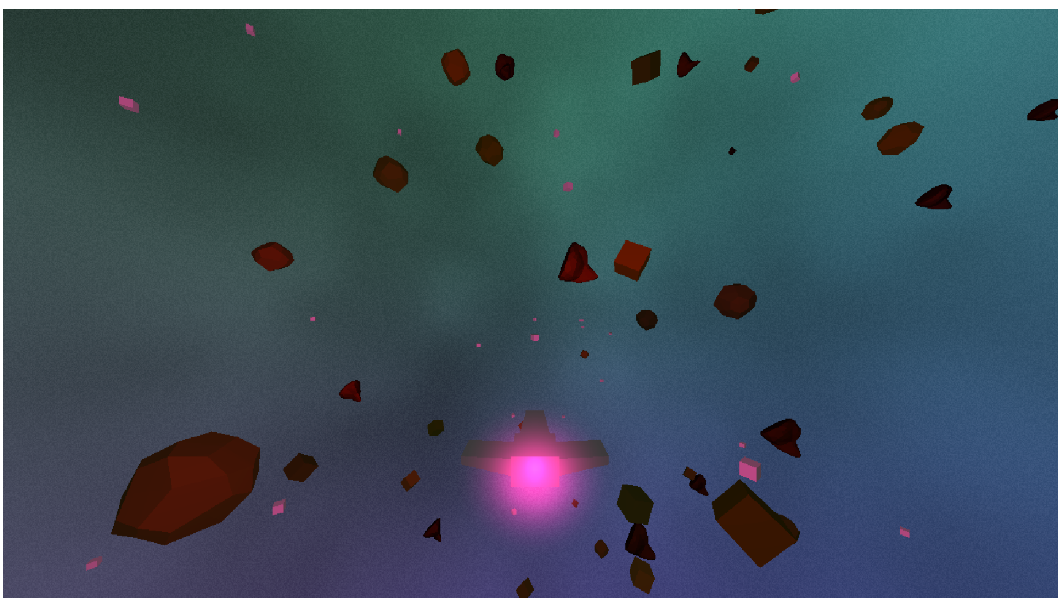
podčást v prvním objektu může zasahovat do vícero podčástí druhého objektu zároveň, což má za následek zbytečného kontrolování vícero podčástí navzájem.

5. Závěr

Cílem této práce bylo seznámit se a shromáždit algoritmy na detekci kolizí a jejich různých variant pro různé matematické objekty a optimalizací. Tyto algoritmy pak byly použity ve 3D renderovacím enginu, který pak měl na starost simulaci a rezoluci kolize 3D objektů. Aby toto bylo možno implementovat, bylo nutné nastudovat teorii ohledně 3D herních enginů, které nejenom zajišťují vykreslení 3D scény, ale také spravuje tuto 3D scénu ve velmi chytrých strukturách tak, aby bylo možné tuto scénu vykreslovat takzvaně "realtime", tedy že stav scény můžeme jako uživatel vidět vždy v aktuálním stavu, aniž bychom museli čekat na její zpracování, neboli že simulační čas je rovný reálnému času. U samotných kolizí pak bylo třeba nastudovat a implementovat diskrétní či spojité verze detekce kolizí různých matematických objektů, přičemž každá technika nabízí své výhody i nevýhody. U jednotlivých kolizních algoritmů jejichž vnitřnosti a myšlenky jsou závislé na matematických reprezentací těchto objektů, bylo nutné nastudovat techniky a rovnice z analytické geometrie, které mi pak napomohli při řešení a rezolucí kolizí.

Celistvý projekt byl napsán v jazyce C++ a díky některým generickým myšlenkám, jako například dělení prostoru podle generických objektů, bylo nutné se naučit pokročilejší C++ techniky jako jsou šablony a compile-time programming. Při renderování mi pak dělal největší problém samotná renderovací knihovna OpenGL a díky své robustnosti mi trvalo, než jsem pojal většinu jejích myšlenek a dostal projekt do stavu, kdy mi engine vyrendroval první trojúhelník. Hlavičková knihovna GLM pak mnohonásobně usnadnila práci s vektory i maticemi a umožnila velmi pohodlnou implementaci jednotlivých kolizních algoritmů.

Tento projekt by se pak měl implementačně a tématicky přibližovat nynějším populárním knihovnám na detekci kolizí jako cgal, box2D, bullet či havoc. Plná implementace či nadstavba této knihovny se samozřejmě v rozumném čase nedala stihnout a přesahuje zadání této práce. Ovšem samotný engine i s kolizními algoritmy jsem použil na implementaci několika menších a konceptuálních videoher a tedy jednotlivé kolizní funkce a engine lze využít pro jednoduché demonstrační simulace. Co se rozšíření týče, implementace plného fyzikálního modelu a prostoru pro tyto kolizní algoritmy je rozhodně jeden z hlavních kandidátů. Další z rozšíření by bylo, použití paralerismu pro zpracování jednotlivých objektů či trojúhelníkových sítí pro zrychlení či přímé použití grafické karty pro řešení těchto kolizních problémů. Díky implementaci spojitých kolizí jsem v podstatě nechtěně byl nucen implementovat kolizi paprsků s různými matematickými útvary a tedy jsem položil v tomto enginu základ pro vykreslovací metodu ray-tracing, což s tímto enginem by bylo velmi možné nad tímto stavět.



Obrázek 5.1: Snímek z demonstrační aplikace.

6. Literatura

[1] REES, Gareth. *Line segment to circle collision algorithm*. [online]
Dostupné z: https://codereview.stackexchange.com/questions/86421/line-segment-to-circle-collision-algorithm

[2] MÖLLER, Tomas. *A Fast Triangle-Triangle Intersection Test*. [online]
Stanford, CA, USA: Stanford University, 1997.
Dostupné z: http://web.stanford.edu/class/cs277/resources/papers/Moller1997b.pdf

[3] GOLDMAN, Ronald. *"Intersection of Three Planes." Graphics Gems I (ed. Andrew S. Glassner).*
Elsevier Science, 1990. ISBN: 978-0122861666

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

[5] EBERLY, David H. *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics.*
CRC Press; 1st edition, 2000. ISBN: 978-1558605930

[6] ERICSON, Christer. *Real-Time Collision Detection.*
Elsevier Science, 2005. ISBN: 1-55860-732-3

[7] FIEDLER, Glenn. *Game Physics, Fix Your Timestep!* [online]
2004.
Dostupné z: https://gafferongames.com/post/fix_your_timestep/

[8] VAN VERTH, James. *Essential Mathematics for Games and Interactive Applications.* BISHOP Lars. Boca Raton, Florida, USA: CRC Press, 2004. ISBN: 978-0-12-374297-1

[9] RÁBOVÁ, Zdeňka, ZENDULKA Jaroslav, Češka Milan, PERINGER Petr, JANOUŠEK Vladimír. *Modelování a simulace.* Vysoké učení technické v Brně, 1992.

[10] BOURKE, Paul. *Object Files (.obj)* [online]
Dostupné z: http://paulbourke.net/dataformats/obj/

[11] BAKER, Martin John. *Maths – AxisAngle to Quaternion* [online]
1998-2021.
Dostupné z: http://www.euclideanspace.com/maths/geometry/rotations/conversions/angleToQuaternion/