# TECHNICKÁ UNIVERZITA V LIBERCI

**Fakulta mechatroniky, informatiky a mezioborových studií**


Studijní program: N2612 – Elektrotechnika a informatika

Studijní obor: 1802T007 – Informační technologie

# Vyhledávání optimální trasy v 3D modelu terénu

# Searching the optimal path in 3D terrain model

**Diplomová práce**


Autor: Bc. Petr Holec

Vedoucí práce: Ing. Jiří Jeníček, Ph.D.


V Liberci 10.05.2012

# Prohlášení

Byl jsem seznámen s tím, že na mou diplomovou práci se plne vztahuje zákon č. 121/2000 Sb. o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu TUL.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladu, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Diplomovou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím diplomové práce a konzultantem.

Datum

Podpis

# Abstrakt

Práce se zaobírá problematikou umělé inteligence a vykreslení třírozměrného prostředí. Z oblasti umělé inteligence se zaměří na interpretaci prostředí pro jednotky umělé inteligence a na algoritmy potřebné pro vyhledávání nejkratší cesty. V oblasti vykreslování popíše možné varianty z pohledu implementace a přenositelnosti. Dále dokument popíše použité struktury a techniky implementací zmíněných algoritmů. V závěru představí některé široce používané optimalizační metody pro rychlejší a efektivnější běh aplikací.



# Klíčová slova

umělá inteligence, navigace, hledání cesty, navigační síť, 3D prostředí



# Abstract

The work deals with problems of artificial intelligence and three-dimensional environment rendering. It will focus on the interpretation of the environment for artificial intelligence units and the algorithms needed to search the shortest path. The plot describes the possible options in terms of implementation and portability. After that the document describes the structure and techniques used by the implementation of these algorithms. At the end of the work some widely used optimization method for faster and more efficient applications will be presented.



# Keywords

artificial inteligence, navigation, path-finding, navigation mesh, 3D environment

# Obsah

# Seznam ilustrací



# Seznam tabulek



# Seznam textových ukázek

# Seznam důležitých termínů a zkratek

| | |
|---|---|
| API | rozhraní pro programování aplikací |
| OpenGL | API pro práci s grafickou kartou |
| shader | specializovaný program pro grafickou kartu |
| pixel | nejmenší část obrazu (picture element) |
| textura | obraz nanášený (nebo jinak aplikovaný) na geometrii |
| Java | moderní programovací jazyk |
| buffer | část paměti vyhrazená pro dočasné uchování dat |
| polygon | mnohoúhelník |
| vertex | vrchol geometrie |
| AI | umělá inteligence (Artificial Intelligence) |
| bot | počítačem řízená jednotka simulující lidské chování |
| avatar | virtuální reprezentace uživatele v aplikaci |
| graf | množina vrcholů a hran spojujících vrcholy |
| PoI | kartografické označení „zajímavého“ bodu (Point of Interest) |
| OBJ | přípona textového souboru ukládající 3D grafická data |
| mesh | polygonální síť objektu |

# Úvod - motivační úloha

Často se člověku stane, že nemůže najít správnou cestu, bloudí nebo zkrátka neví, kde je. V posledních letech se stále více a více využívá satelitní lokalizace GPS (Global Positioning System). Využití této technologie v makroměřítku, jako je silniční navigace, je již k dnešním dnům naprosto běžnou věcí. Díky tomu je i spousta zmapovaných lokalit a pomocí aplikací, ať už na mobilní telefony nebo na desktopové počítače, lze náš cíl naleznout během několika okamžiků.

Méně rozšířená je navigace v interiérech. Důvodem k tomu bude fakt, že v budovách je méně záchytných bodů a povaha terénu je komplexnější oproti exteriérům. Základní myšlenkou této práce je vytvořit aplikaci, která by sloužila jako interaktivní průvodce po budovách a jiných objektech, ať už jsou to školy, muzea, nemocnice nebo jiné budovy, kde je potřeba najít pozici nebo cestu zadané místnosti nebo místa.

Aplikace se dotkne takových problematik jako jsou algoritmy umělé inteligence pro vyhledávání nejkratší cesty, optimalizační úlohy a vykreslovací techniky. Otestuje základní znalosti z teorie grafů, analytické geometrie a lineární algebry. Dalším možným rozšířením aplikace je propojení přes internet a získávání aktuálních dat po síti. Těmito daty můžou být rozvrhy osob pracujících v objektu nebo fotografie (obyčejné nebo panoramatické) společně s informací o místě, ze kterého byla fotografie pořizována.

# 1 Využité prostředky

Vývoj aplikace vyžaduje vhodné zvolení správného prostředí vzhledem k použití aplikace. Pro tuto aplikaci bylo pro práci s grafickou stránkou zvoleno rozhraní OpenGL kvůli jeho přenositelnosti. Ze stejného důvodu byl zvolen jazyk Java a knihovna JOGL, která implementuje rozhraní OpenGL do tohoto jazyka.

## *1.1 OpenGL*

OpenGL je aplikační rozhraní, nebo-li API (Application Programming Interface) pro práci s grafickou kartou (akcelerátorem). Je multiplatformní a jde ho tudíž implementovat na skoro všechny známé počítačové platformy.

První verze OpenGL byla do světa vypuštěna v roce 1992 společností Silicon Graphics, Inc. (dále jen SGI). Společnost SGI byla založena v roce 1982 dvojicí Jimem Clarkem a Abbey Silverstonovou v Kalifornii. V dnešní době má pobočky na všech obydlených kontinentech.

Produktem této společnosti je výpočetní technika určená k náročným, převážně grafickým, operacím. Jsou to tedy servery a clustery (seskupení více počítačů pro výpočet), úložiště dat a vizualizační aplikace (např. VUE). V dnešní době je standard OpenGL spravován skupinou Khronos Group do níž patří např. Intel, AMD, Apple, Fujitsu, NVIDIA a spousta dalších. Od verze 2.0 podporuje OpenGL shadery.

Shader je program pro zpracovávání dat přímo na grafické kartě. Pro OpenGL se píše v jazyku GLSL (OpenGL Shading Language) pro DirectX pak HLSL. Prozatím jsou známé tři typy shaderů rozšířené o shadery řídící tesalaci (pro OpenGL - Tessellation control shader a Tessellation evaluation shader). Prvním je vertex shader. Ten provádí operace na jednotlivých vrcholech, nevýhodou oproti geometry shaderu (zmíněný níže) je v tom, že vstupuje vždy jeden vertex (vrchol) a vystupuje jeden. Pixel shader se stará o manipulaci s pixelem, který dostane od vertex shaderu. V OpenGL se pixel shader označuje za fragment shader. Hlavní funkcí tohoto shaderu je nanášení hodnoty osvětlení, textury a jiných složitějších efektů (např. bump mapping). Posledním typem shaderu je geometry shader. Stojí mezi vertex shaderem a pixel shaderem. Jeho schopností je upravit geometrii přidáním nebo ubráním vertexů. Využitím může být třeba simulace zatravnění, aplikace displacement mapy nebo tesalace. Ačkoliv je síla tohoto nástroje veliká, podpora u API je až od verze 10 u DirectX a nativně od verze 3.0 u OpenGL (dříve skrze rozšíření - extensions).

S příchodem shaderů se na grafických kartách začaly používat paralelní shaderovací jednotky (pipeline), podle příslušných shaderů (vertex, pixel). Tento přístup byl dost neflexibilní a v případě, že aplikace byla náročnější na práci s texturami a osvětlením, tudíž pro pixel shader jednotku, jednotka vertex shaderu zůstala zatím nevyužita. S příchodem nové generace grafických karet přišlo i řešení tohoto problému. Vznikla unifikovaná shader jednotka (unified shader pipeline), která může fungovat jako kterákoliv z uvedených. Vytížení procesoru se tak rovnoměrně rozloží. Jelikož se OpenGL snaží o maximální nezávislost na platformách, používají se častěji než OpenGL jeho rozšiřující balíčky jako jsou GLU, GLUT nebo „frameworky“ jako je například Tao Framework pro C#.

## 1.2 Java

Java je vyšší programovací jazyk a výpočetní platforma. První verze byla vydána společností Sun Microsystems v roce 1995. Od roku 2007 je jazyk Java uvolněn jako *open source*. Jelikož se jedná o objektově orientovaný jazyk, umožňuje programátorům využívat výhod takto zaměřených jazyků – ať už se jedná o polymorfismus, zapouzdření, strukturování kódu nebo, a to především, znovupoužitelnost kódu.

Filosofie Javy spočívá v co největším rozšíření na nejrůznější platformy. Pro tento účel využívá virtuální stroj (JVM – Java Virtual Machine), který je už platformově závislý. Programy jsou pak přenositelné, o běh se stará JVM. Pro spouštění pak používá kompilovaný bytecode (někdy portable code), který je již z části optimalizovaný, ale stále platformově nezávislý. Po spuštění se bytecode zkompiluje do strojového kódu čitelného pro danou platformu a tím je zaručena rychlost běhu aplikace. Takovýmto kompilátorům se říká just-in-time kompilátor (JIT). Java platforma je zodpovědná za správu přiřazené paměti a pro tento účel má specializovaný mechanismus – *Garbage Collector*, který zjišťuje, na které objekty uložené v paměti už neexistuje reference a tudíž toto místo v paměti lze uvolnit, protože neexistuje způsob, jak se k danému objektu dostat.

Java patří mezi nejrozšířenější programovací jazyky na světě. Jeho výhody využívá přes miliardu stolních počítačů a přes tři miliardy mobilních telefonů. Java pohání aplikace stolních počítačů, webové aplikace, mobilní telefony, blue-ray přehrávače, televize, set-top boxy a mnoho dalšího.

# 2 Teorie a algoritmy

## 2.1 Lineární algebra

Lineární algebra je odvětví matematiky zabývající se vektory, vektorovými prostory, soustavami lineárních rovnic a operacemi nad nimi. Výpočty související se zobrazováním hojně využívají poznatků získaných z této disciplíny. Ale to nebude téma našeho zájmu, protože o to je díky knihovnám OpenGL postaráno. Cílem, na který se zaměříme, budou soustavy rovnic. V aplikaci je použito řešení soustavy rozměru 3x3 nebo menší, proto tento rozměr bude brán i jako příklad. Větší rozměry soustav se řeší analogicky. Nejprve představíme základní pojmy lineární algebry, poté i jeden důležitý algoritmus právě pro řešení soustavy rovnic.

*Skalár* nebo skalární veličina je označení pro veličinu, která je vyčíslitelná jediným číselným údajem. Lze říci, že skalár je jednorozměrný vektor. Tedy *vektor* je uspořádaná n-tice skalárů. Ve fyzice se vektory používají pro vyčíslení veličin, které mají kromě velikosti i směr (například rychlost, síla). Označují se horizontální šipkou nad označením ( například $\vec{a}$ ). *Matice* je uspořádaná n-tice skalárů. Na těchto elementech jsou definovány operace jako jsou sčítání (po prvcích), násobení (po prvcích), násobení matic, skalární součin, vektorový součin a další. Důležitým pojmem je *lineární závislost vektorů*. Pokud vztahujeme tento pojem ke dvěma vektorům, znamená to, že jeden vektor lze vyjádřit druhým vektorem v součinu se skalárem. Například $\vec{a}=(2;1;3)$ je lineárně závislý na $\vec{b}=(1;0{,}5;1{,}5)$ . Lineární

závislost je důležitým aspektem pro řešení soustavy lineárních rovnic.

$$\begin{aligned} a_1 \cdot x + b_1 \cdot y + c_1 \cdot z &= r_1 \\ a_2 \cdot x + b_2 \cdot y + c_2 \cdot z &= r_2 \\ a_3 \cdot x + b_3 \cdot y + c_3 \cdot z &= r_3 \end{aligned} \quad \rightarrow \quad \left( \begin{array}{ccc|c} a_1 & b_1 & c_1 & r_1 \\ a_2 & b_2 & c_2 & r_2 \\ a_3 & b_3 & c_3 & r_3 \end{array} \right)$$

*Ilustrace 1: Použití matice pro zápis soustavy lineárních rovnic*

Pro řešení soustavy rovnic zapsaných v matici se používá *Gaussova eliminace*. Princip tohoto algoritmu je takový, že se matice upraví na jednotkovou matici. Jednotková matice je taková, která má na hlavní diagonále jedničky a na ostatních pozicích nuly. Povolené úpravy jsou přičítání jiného řádku, násobení řádku nenulovým číslem, prohození řádků, prohození sloupců nebo složitější operace vzniklé kombinací uvedených. U prohození sloupců je nutno si uvědomit, že se sloupcem prohazujeme i proměnné. Všechny operace je nutné aplikovat i na vektor $\vec{r}$ , ve kterém po skončení algoritmu je řešení. Gaussova eliminace se používá i pro výpočet inverzní matice.

$$\left( \begin{array}{ccc|c} 1 & 0 & 0 & q_1 \\ 0 & 1 & 0 & q_2 \\ 0 & 0 & 1 & q_3 \end{array} \right) \quad \rightarrow \quad \begin{aligned} x + 0 + 0 &= q_1 \\ 0 + y + 0 &= q_2 \\ 0 + 0 + z &= q_3 \end{aligned}$$

*Ilustrace 2: Výsledek eliminace*

Postup eliminace je takový, že se nejprve vynulují prvky pod diagonálou, a to nejprve první sloupec, pak druhý. V další fázi se eliminují prvky nad diagonálou, nejprve posledního sloupce, poté druhého sloupce.

## 2.2 Analytická geometrie

Ačkoliv analytické geometrie se v tomto projektu využívá pouze té základní, je nezbytnou součástí. Je potřebná pro korekci vyhledané cesty a poznatky z tohoto odvětví se hojně používají při detekci kolizí. Hlavními prvky jsou bod, přímka a rovina.

Bod je v třírozměrném (jiný uvažovat nemusíme) definován vektorem o třech složkách. Přímka v prostoru je jednoznačně určena dvěma různými body. Z této vlastnosti i vychází

parametrický tvar přímky. Platí tedy $X = X_0 + k \cdot \vec{s}$ , kde X je libovolný bod přímky, $X_0$ počátek, $\vec{s}$ směrový vektor a $k \in \mathbb{R}$ je parametr. Pokud uvažujeme úsečku jako oboustranně omezenou přímku, pak $\vec{s}$ je vektor určený koncovými body úsečky a parametr nabývá hodnot z intervalu <0; 1>.

Pokud se jedná o rovinu, tak rovina je jednoznačně určena třemi různými body, které neleží na jedné přímce. Platí tedy $X = X_0 + k \cdot \vec{a} + l \cdot \vec{b}$ , kde vektory $\vec{a}$ a $\vec{b}$ jsou lineárně nezávislé, což je dané už určením roviny (tři různé body). Parametry *k* a *l* jsou opět z množiny reálných čísel. $X_0$ je libovolný bod roviny. U roviny a přímky se často ještě uvádí obecný tvar, kde u roviny je obecný tvar přímo závislý na normálovém vektoru. Normálový vektor (normála) je vektor kolmý k rovině tudíž i k oběma vektorům, které ji určují. Vypočítá se jako vektorový součin těchto dvou vektorů.

Normálový vektor se často normalizuje tak, aby měl velikost rovnou jedné (jednotkový vektor). Činí se tak například kvůli výpočtu osvětlení tělesa. OpenGL má pro normalizaci normálových vektorů přímo funkci, avšak je pro vykreslování v reálném čase vhodnější zaručit, že vektory splní požadavek na velikost. Ušetří se výpočetní čas, který by stálo přepočítávání velikosti při každém snímku. Normalizace vektoru se provede vydělením jeho složek velikostí daného vektoru.

### 2.2.1 Vzájemná poloha geometrických těles

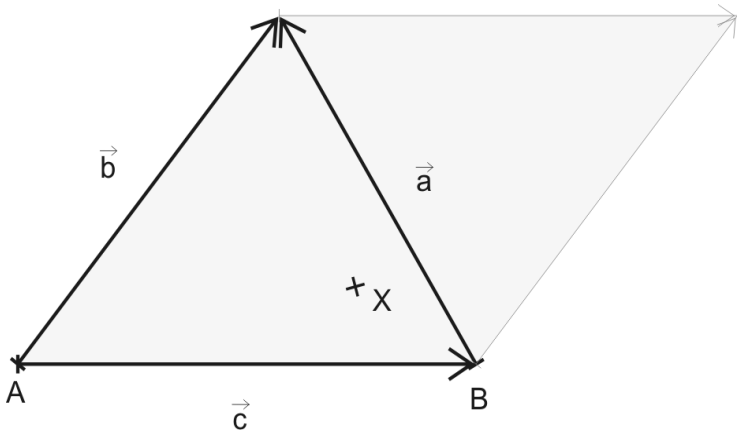
Existuje několik variant vzájemné polohy rovin a přímek v prostoru. Dvě přímky v prostoru mohou být mimoběžné, neleží v jedné rovině a nemají tedy společné body. Směrové vektory jsou lineárně nezávislé. Dále přímky mohou být rovnoběžné, leží v jedné rovině. Mají společné body pouze v případě, že jsou totožné. Při rovnoběžnosti jsou oba směrové vektory lineárně závislé. Poslední případem je různoběžnost, kdy přímky mají jeden společný bod (leží v jedné rovině), směrové vektory jsou (stejně jako u mimoběžnosti) lineárně nezávislé.

U rovin je možností méně. Pokud jsou normálové vektory lineárně závislé, pak jsou buď rovnoběžné a nebo totožné. To znamená, že nemají společný bod, nebo jejich průsečíkem je rovina. Pokud nejsou lineárně závislé, jejich průsečíkem je přímka.

Poloha přímky a roviny nabývá tří stavů. Ke dvěma z nich dochází v případě, kdy směrový vektor přímky je lineárně závislý na vektorech roviny. Vektor přímky se dá vyjádřit jako součet vektorů roviny, které mohou být vynásobené libovolným reálným číslem. V takovémto případě společné body neexistují, nebo v případě, že leží v rovině, je průnikem zadaná přímka. Třetí stav nastává, pokud jsou všechny směrové vektory (roviny a přímky)

lineárně nezávislé. Pak je průsečíkem jeden bod.

Častou úlohou je průnik trojúhelníku a přímky v prostoru. Nejprve se tedy provede zjištění průsečíku roviny trojúhelníku a přímky a poté se zjistí, jestli zjištěný průsečík (pokud existuje) leží uvnitř trojúhelníku. Průsečík dostáváme v případě, kdy dosadíme do parametrického tvaru rovnice roviny z parametrického tvaru rovnice přímky. Dostaneme pro tři složky prostoru (x,y,z) a tři parametry (dva pro rovinu, jeden pro přímku) soustavu tří rovnic o třech neznámých. Po vyřešení známe hodnoty parametrů. U přímky nemá jeho hodnota v konkrétní úloze větší význam. Podle hodnot parametrů pro rovinu lze určit, jestli průsečík leží uvnitř trojúhelníku. Je však třeba, aby rovina byla popsána tak, jako tomu je na obrázku (Ilustrace 3) a nebo ekvivalentním způsobem.




*Ilustrace 3: Průnik trojúhelníku*

Bod leží uvnitř trojúhelník, pokud hodnoty parametrů jsou z intervalu <0; 1> a jejich součet je menší nebo roven jedné. Pokud bychom omezili oba parametry **pouze** na intervaly <0; 1>, dostali bychom definici rovnoběžníku. Předpokládejme tedy, že pro důkaz postačí dokázání limitního případu, kde oba parametry jsou z intervalu <0; 1> a jejich součet je **roven** jedné. Pak bychom měli dostat rovnici pro stranu *a*. Dalším popisem bude proveden důkaz daného tvrzení.

$a : X = B + m \cdot \vec{a} \quad \rightarrow \quad X = B + m \cdot (C - B)$ *Toto je tvar, kterého chceme dosáhnout.*

$X = A + k \cdot \vec{c} + l \cdot \vec{b}$ *Libovolný bod se dá vyjádřit jako součet počátku a vektorů s patřičnými parametry. Pro nás je počátek bod A.*

$k + l = 1 \quad \rightarrow \quad k = 1 - l$ *Součet parametrů je roven jedné, jak bylo řečeno.*

$X = A + k \cdot (B - A) + l \cdot (C - A)$ *Vyjádříme vektory jako rozdíl hraničních bodů.*

$X = A + (1-l)\cdot(B-A) + l\cdot(C-A)$ *Dosadíme za k.*

$X = B + l\cdot C - l\cdot B$ *Po roznásobení a upravení dostáváme už jen tyto členy.*

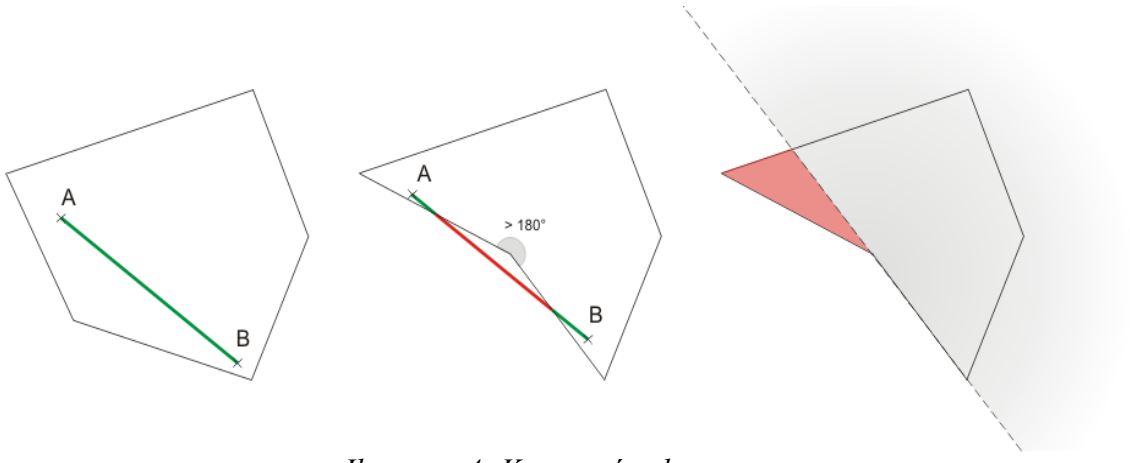
$X = B + l\cdot(C-B) \quad \rightarrow \quad X = B + l\cdot\vec{a}$ *Což lze dále upravit na toto řešení.*

### 2.2.2 Konvexní tělesa

Důležitým pojmem z geometrie pro různé algoritmy výpočtu detekce kolize nebo optimalizační a vykreslovací algoritmy je konvexní polygon (mnohoúhelník) nebo polyhedron (mnohostěn). Definujeme-li konvexní polygon tak, že všechny jeho vnitřní úhly jsou menší než 180°, dostáváme zajímavý fakt, že pro libovolnou úsečku, jejíž hraniční body jsou unvitř polygonu, platí, že všechny její body leží uvnitř polygonu (viz Ilustrace 4 - vlevo a uprostřed).

V praxi to znamená, že pokud se chceme dostat nejkratší cestou z libovolného místa A do libovolného místa B uvnitř konvexního polygonu, pak tou nejkratší cestou je přímá cesta. Další fakt, proč je termín konvexní polygon tak používaný, je způsob, jak jej lze vymezit.

Polygon je určen konečným počtem úseček. Každá z těchto úseček rozděluje rovinu na dvě poloroviny. Uvažujeme-li pro každou úsečku tu polorovinu, ve které leží náš polygon, pak polygon je jednoznačně určen průnikem všech polorovin pro tyto úsečky. Toto tvrzení neplatí pro nekonvexní polygony (viz Ilustrace 4 - vpravo). Nejjednodušší konvexní polygon je trojúhelník, to že je konvexní je zaručeno tím, že součet vnitřních úhlů je 180°. Proto žádný z jeho vnitřních úhlů nemůže překročit tuto hranici.




*Ilustrace 4: Konvexní polygon*

## 2.3 Interpretace navigačního prostředí

I přes to, že se ve finále použije téměř vždy stejný algoritmus pro vyhledání nejkratší cesty, interpretace prostředí, na kterém se tento a další (post-processing) algoritmy aplikují, má veliký význam. Ať už z hlediska výpočetní doby, přesnosti nebo možností další funkcionality je volba prostředí kritickou součástí AI.

### 2.3.1 Waypoint systém

Hojně používaná a relativně rychlá interpretace je pomocí tak zvaných *waypoints* neboli trasových bodů. Umělí protivníci (říkejme boti) v počítačových hrách často používají těchto bodů pro svojí navigaci v prostoru. Výhodou této interpretace je rychlý výpočet při nízké úrovni detailů sítě bodů a nízká paměťová náročnost. Velkou nevýhodou je malá flexibilita na změnu prostředí a, vezmeme-li v úvahu i více typů botů, tak i větší požadavky na místo.

Waypoint systém je založený na předgenerování sítě grafu v závislosti na velikosti bota. Některé průchody v prostředí můžou být neprůchodné, pokud velikost bota přesáhne určitou míru. To znamená, že pokud máme více velikostí botů, je potřeba vygenerovat více grafů. V katastrofickém případě můžeme mít bota, který je schopen změnit velikost za běhu programu. V tomto okamžiku se musí jeho síť celá přegenerovat, což může být relativně časově náročné.

Systém rovněž obtížně reaguje na situaci, kdy se na trase objeví překážka. Ačkoliv tato překážka se může dát obejít krokem stranou, systém pravděpodobně vygeneruje buď z části novou síť a nebo začne hledat novou cestu. Další nevýhodou této interpretace je, že nepočítá s parametrickým posunem. Například budeme-li mít vozidlo, které se neumí otočit na místě (třeba motocykl), waypoint systém s touto informací neumí pracovat. I když převážně estetická nevýhoda je „cik-cak“ efekt, ke kterému dochází velmi často, pokud jsou podklady pro vytvoření sítě trojúhelníky z triangulace prostředí.

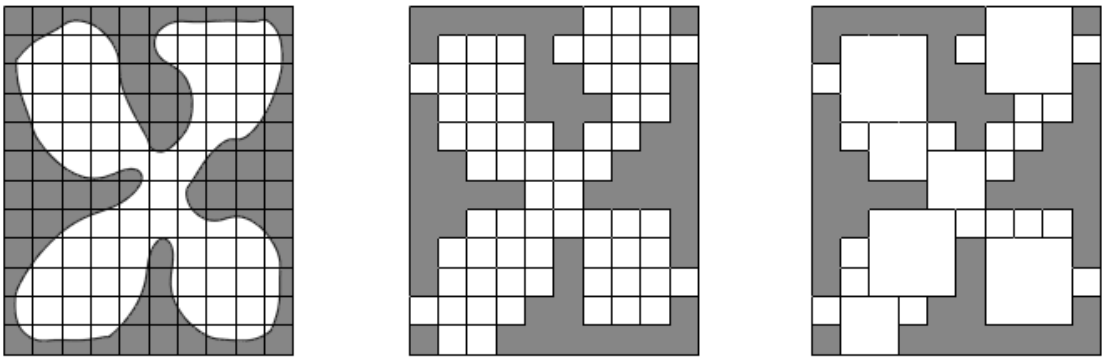
### 2.3.2 Čtvercové reprezentace

Čtvercové reprezentace by se daly spíš označit za diskrétní, protože existuje i kruhové reprezentace. Často se označují za „grid“ systémy (z angl. mřížka). Základní princip této reprezentace je, že se celý dostupný prostor vyplní čtverci (případně krychlemi u 3D prostoru). Velikost této krychle je brána jako čtvrtina z největšího rozměru bota. Začne se na začátku (ve vstupním bodě) prostředí a průchodem do šířky či hloubky se vždy provede test krychle s prostředím a podle výsledku a porovnání s kritérii se zahodí nebo ponechá. V případě, že se jedná o homogenní síť, tak zde proces končí.

Tato triviální reprezentace se v minulosti používala často pro strategické počítačové hry,

kde se uvažovalo, že jedna buňka mohla obsahovat jednoho agenta umělé inteligence. Detekce kolizí byla pak pouze jednoduchým porovnáváním, zda dílčí buňka v cestě je obsazená či nikoliv.

Rozšířit diskrétní reprezentaci lze optimalizačním krokem, při němž se menší souvislé bloky slučují do větších krychlí a zachovává se informace o jejich sousedech. V praxi pak vzniknou větší bloky navzájem spojené menšími, které se občas označují za brány. Což může někomu připomínat *navigační mesh* (2.3.3 Navigation mesh). Tato metoda není příliš přesná už jen proto, že používá krychle. Na hranici s diagonální stěnou může docházet k cik-cak efektu. Na trojobrázku (Ilustrace 5) je vlevo vidět prostředí před diskretizací, uprostřed po diskretizaci a vpravo po optimalizaci na heterogenní síť.



*Ilustrace 5: Diskretizace na síťovou reprezentaci*

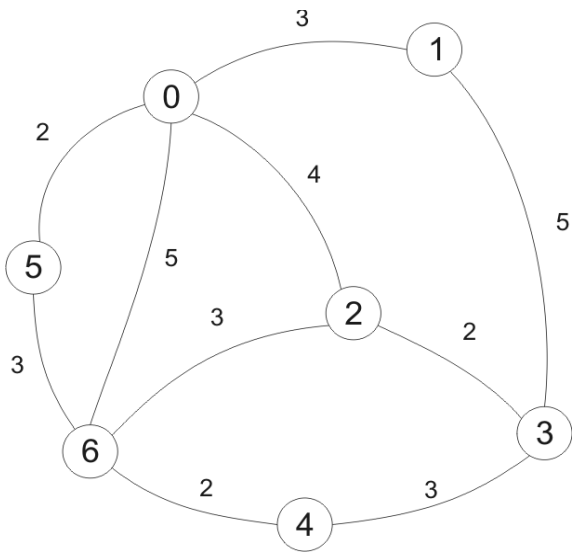
### 2.3.3 Navigation mesh

Zobecněním heterogenní čtvercové (síťové) reprezentace získáváme navigační mesh. Jedná se tedy o využití konvexních polygonů navzájem propojených bránami. Jednotlivé jednotky systému mohou kromě polygonu uchovávat informaci o maximální propustnosti dle velikosti polygonu, o typu terénu a další. Vygenerování grafu lze provést například ze středů těchto polygonů a propojit přes brány nebo jako uzly grafu použít středy bran. Navíc lze použít stejný mesh, který má kolizní model, ačkoli to může někdy přinést nadbytečné detaily a tím zvýšit vyhledávací nároky. Díky tomu, že je zachovaná informace o prostoru, ve kterém se bot pohybuje, může těchto informací využít pro úhybné manévry v případě zatarasení jeho aktuální cesty. Navíc lze odstranit i cik-cak efekt různými metodami. Jednou z nich je v dalších kapitolách zmíněný funnel algoritmus.

## 2.4 Teorie grafů

Nejprve, než rozebereme použité algoritmy, je třeba nadefinovat některé často používané pojmy. Prvním bude *graf*, který byl již dříve zmíněný. Graf se skládá z množiny *vrcholů* a množiny *hran*. Hrana spojuje právě dva vrcholy. Pokud jsou oba vrcholy totožné, označuje se hrana za smyčku. Pro náš konkrétní případ smyčky neuvažujme. Pokud je hranám přiřazené ohodnocení, což je náš případ, jedná se o ohodnocený graf. V případě, že jsou hrany orientovány, to znamená, že závisí na pořadí přiřazení vrcholů hraně, pak se jedná o orientovaný graf. Což není náš případ.

Uvažujeme-li tedy neorientovaný graf, pak sled je posloupnost hran taková, že vždy po sobě jdoucí hrany v této posloupnosti mají společný alespoň jeden uzel. Tah je takový sled, kde všechny hrany v posloupnosti mají zároveň i unikátní výskyt. Tudíž se v posloupnosti hrany nesmí opakovat. A konečně cesta je takový tah, kde se nevyskytuje žádný uzel více než jednou, respektive uzly v cestě mají unikátní výskyt. Na obrázku (Ilustrace 6) je neorientovaný ohodnocený graf. Tento graf je souvislý a obsahuje jednu smyčku u vrcholu 5.




*Ilustrace 6: Ohodnocený neorientovaný graf*

Pro práci s grafy je velmi podstatná možnost projít celý graf nebo jeho část a získat a nebo zapsat nějaké informace. Typickým příkladem je potřeba zjistit, zda je jeden uzel dosažitelný z uzlu jiného a tudíž (v neorientovaném grafu), zda je graf souvislý a nebo se skládá z komponent.

Další velmi rozšířenou úlohou z rastrové grafiky je výplň barvou (floodfill). Jednotlivé pixely jsou uzly. Soused uzlu je takový pixel, který má náležité souřadnice a stejnou barvu (nebo případně odstín).

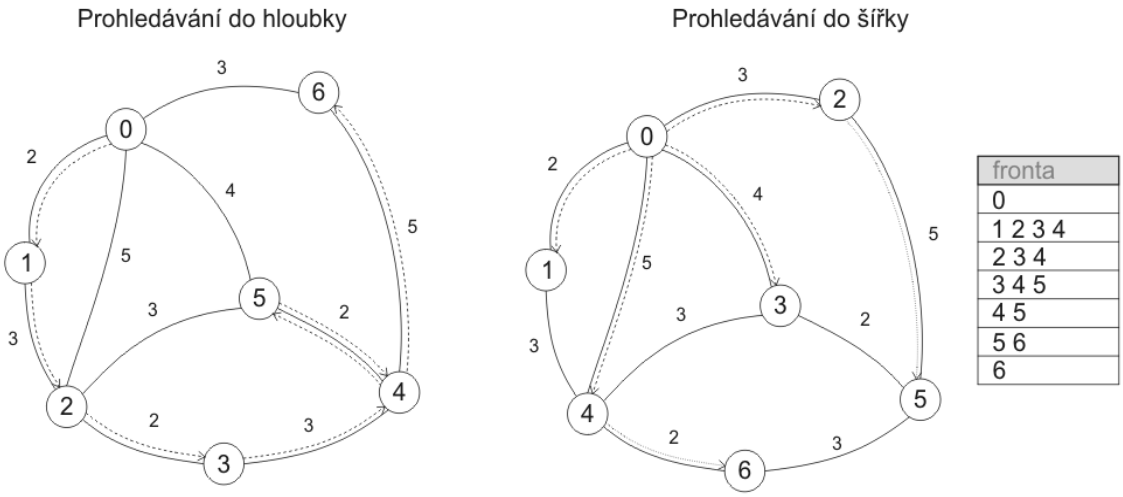
### 2.4.1 Prohledávání do šířky

První ze dvou typických přístupů k prohledávání grafu je prohledávání do šířky (angl. Breadth-first Search). Tento přístup se považuje za paměťově náročnější a jeho princip spočívá v tom, že se vytvoří datová fronta. Do fronty se přidá výchozí uzel. Z fronty se postupně odebírají uzly, označí se jako prohledané a jeho neprohledaní sousedé se přidají na konec fronty. Algoritmus končí ve chvíli, kdy je fronta prázdná a nebo v závislosti na řešené úloze. V rámci algoritmu lze k uzlům poznamenávat předcházející uzel a díky těmto vazbám lze později vytvořit kostru grafu (strom pokrývající celý graf), pokud původní graf je spojitý.

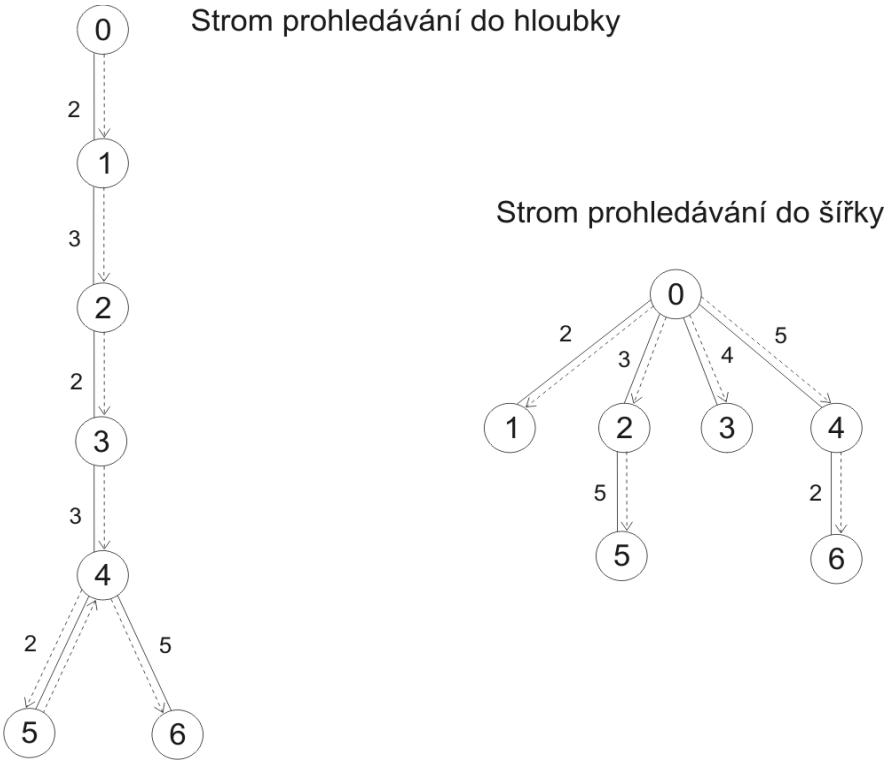
### 2.4.2 Prohledávání do hloubky

Druhý přístup k prohledávání je prohledávání do hloubky (angl. Depth-first Search). Tento algoritmus využívá princip rekurze (funkce volající sama sebe). Rekurentní funkce začne ve výchozím uzlu, označí ho za prohledaný a pro všechny jeho neprohledané sousedy volá sama sebe. Stejně jako u prohledávání do šířky lze (mimo jiné) poznamenávat uzel, ze kterého se konkrétní uzel našel a vytvořit tak zpětně kostru. Nevýhodou algoritmu jsou zvýšené nároky na zásobník funkcí kvůli rekurzi.

Přiblížením algoritmu prohledání do hloubky je výpis hierarchické struktury, například osnovy strukturovaného textu. V něm se nejprve vypíše název první kapitoly, pak její podkapitoly (a dále do hloubky). Až když se vypíše prvek nejhlouběji vnořený, pak se pokračuje s výpisem druhé kapitoly.




*Ilustrace 7: Prohledávání grafu*




*Ilustrace 8: Prohledávací stromy*

Na obrázku (Ilustrace 7) jsou dva prohledané grafy, levý demonstruje prohledávání do hloubky, pravý do šířky. Označení vrcholů znázorňuje, kdy byl vrchol poprvé navštíven. Výchozím uzlem je uzel 0. Pro výběr dalšího souseda se priorizuje ten, který je spojený hranou s nižším ohodnocení. Výsledek prohledávání lze zobrazit pomocí prohledávacího stromu. Jedná se

o graf, ve kterém se zanedbávají hrany, které nebyly při prohledávání použité. Strom je acyklický graf. V ilustraci (Chyba: zdroj odkazu nenalezen) jsou oba prohledávací stromy z předchozího příkladu.

### 2.4.3 Nejkratší cesta a Dijkstrův algoritmus

V předchozí kapitole byly vysvětleny důležité prvky pro vyhledávání nejkratší cesty. Nejkratší cesta mezi dvěma body je taková cesta, že součet ohodnocení hran této cesty je minimální ze všech možných cest mezi těmito dvěma body. Právě hledání nejkratší cesty je častou úlohou AI (z angl. „Artificial Intelligence“) neboli umělé inteligence.

Algoritmus pro vyhledávání nejkratší cesty uvažuje, že graf je souvislý nebo že počáteční a cílový uzel cesty jsou součástí stejné souvislé komponenty. Souvislý neorientovaný graf je takový, že pro každé dva uzly tohoto grafu existuje cesta, která by je spojovala.

Dijkstrův algoritmus byl vymyšlen a představen nizozemským počítačovým vědcem Edsgerem W. Dijkstrou v roce 1959. Vyhledává nejkratší cestu mezi počátečním uzlem a každým dalším uzlem daného grafu (za předpokladu souvislosti). Algoritmus uvažuje nezáporné ohodnocení hran. V našem případě je tímto ohodnocením Euklidovská vzdálenost mezi uzly v prostoru. Čímž jsou podmínky splněné. Pro tento algoritmus zavedeme pojmy „ohodnocení uzlu“ pro délku aktuálně známé nejkratší cesty a „předchůdce uzlu“ pro předcházející uzel v cestě. „Soused uzlu“ bude uzel přímo spojený s tímto uzlem hranou. „Vyřešený uzel“ budeme nazývat uzel, který má již nejkratší cestu určenou. Pro popis algoritmu označíme počáteční uzel za „START“ a koncový za „GOAL“.

Postup algoritmu je následující:

1. **Inicializační krok**
   - všechny uzly ohodnotíme + ∞
   - START ohodnotíme 0
   - všechny sousedy START ohodnotíme vždy součtem ohodnocení START a ohodnocením hrany mezi START a daným sousedem
   - všem sousedům START nastavíme předchůdce na START
2. **Nalezení nejmenšího uzlu M**

- nalezneme z nevyřešených uzlů takový, co má nejmenší ohodnocení
- označíme M za vyřešený
- všem sousedům M nastavíme ohodnocení na ohodnocení M + ohodnocení hrany M a daného souseda pokud je nové ohodnocení menší, než stávající a pokud tomu tak je, nastavíme předchůdce souseda na M

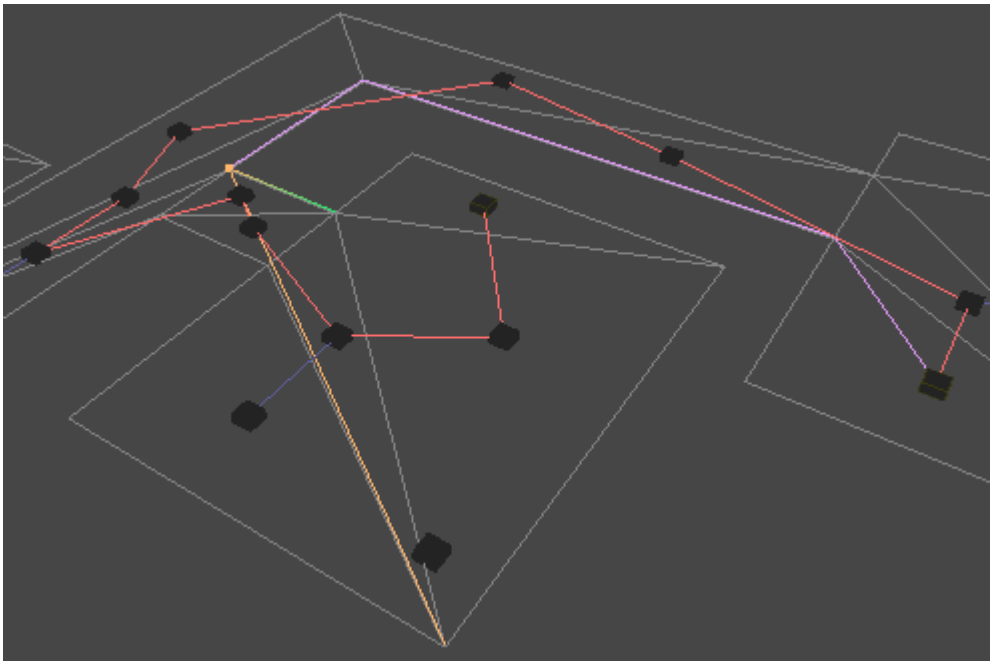
3. **Cyklus**
   - opakujeme krok 2. do chvíle, kdy se za M zvolí GOAL
4. **Zpětné traverzování**
   - zpětným traverzováním od GOAL za pomocí předchůdců dostaneme cestu

Složitost originálního Dijkstrova algoritmu je $O(|V|^2)$. V praxi je nejpoužívanější úprava A* (čti „ej star“). Na rozdíl od Dijkstrova algoritmu A* využívá heuristickou funkci pro priorizování dalšího vrcholu ke zpracování. Pro rozsáhlé grafy dochází k efektivnějšímu nálezu nejkratší cesty. Pro výběr dalšího vrcholu se nejčastěji používá porovnávání euklidovské vzdálenosti konkrétního vrcholu a cíle. Laicky řečeno se pokusí jít nejprve přímou cestou. Nevýhodou A* je fakt, že jeho výsledkem nemusí být nutně nejkratší cesta.

## 2.5 Funnel algoritmus

Název *funnel* neboli trychtýř mírně napoví princip tohoto algoritmu. Funnel algoritmus využívá obousměrné fronty (anglicky označován za *Deque*). Jedná se o koncept datové struktury podobné frontě nebo zásobníku, ze které lze vybírat jak ze začátku, tak z konce (respektive zleva a zprava).

Koncepce algoritmu zahrnuje *channel* (kanál), *path* (cesta), *apex* (hrot) a *funnel* (trychtýř). Kanál je prostor, na kterém probíhá algoritmus. V našem případě se jedná posloupnost trojúhelníků vytyčenou Dijkstrovým algoritmem. Obecněji vzato algoritmus operuje na sekvenci konvexních polygonů. Jednotlivé polygony v rámci kanálu jsou spojeny vnitřní hranou. Budeme je nazývat *gate* (brána). Path je doposud získaná část cesty tímto algoritmem. Apex je bod, ve kterém se dotýkají path a funnel. Apex současně virtuálně rozděluje obousměrnou frontu na levou a pravou část.



*Ilustrace 9: Prvky funnel algoritmu*

Na obrázku (Ilustrace 9) je znázorněna vypočtená cesta pomocí Dijkstrova algoritmu (červená linka) uvnitř navigačního meshe (šedá linka). Fialová linka označuje už připravenou cestu z funnel algoritmu, oranžový bod pak apex, zelená linka levou část funnelu a oranžová pravou část.

Postup algoritmu je následující:

5. **Inicializační krok**
   - do funnel se přidá výchozí bod (START)
   - vezmeme první bránu z kanálu a její vrcholy přidáme do funnelu, levý na levý konec funnelu a pravý na pravý konec
6. **Další brána**
   - vezmeme další bránu a vybereme z ní vrchol (M), který nebyl ještě přidán
7. **Přidání vrcholu**
   - určíme stranu (S), na kterou přidat vrchol podle posledního přidávání, opačnou stranu označíme za (NS)
   - poslední vrchol strany S označíme za $V_1$ a předposlední vrchol za $V_2$

– určíme S-točivost (pravo/levo) pro dvojci vektorů $\overrightarrow{V_1V_2}$ a $\overrightarrow{V_1M}$, pokud $V_1$ je apex, určuje se NS-točivost

– pokud test vyjde pozitivní, vrchol M se přidá na konec S strany funnelu a opakuje se od kroku 2.

– pokud test vyjde negativní, vyjme se z funnelu vrchol $V_1$ a přidá se do cesty

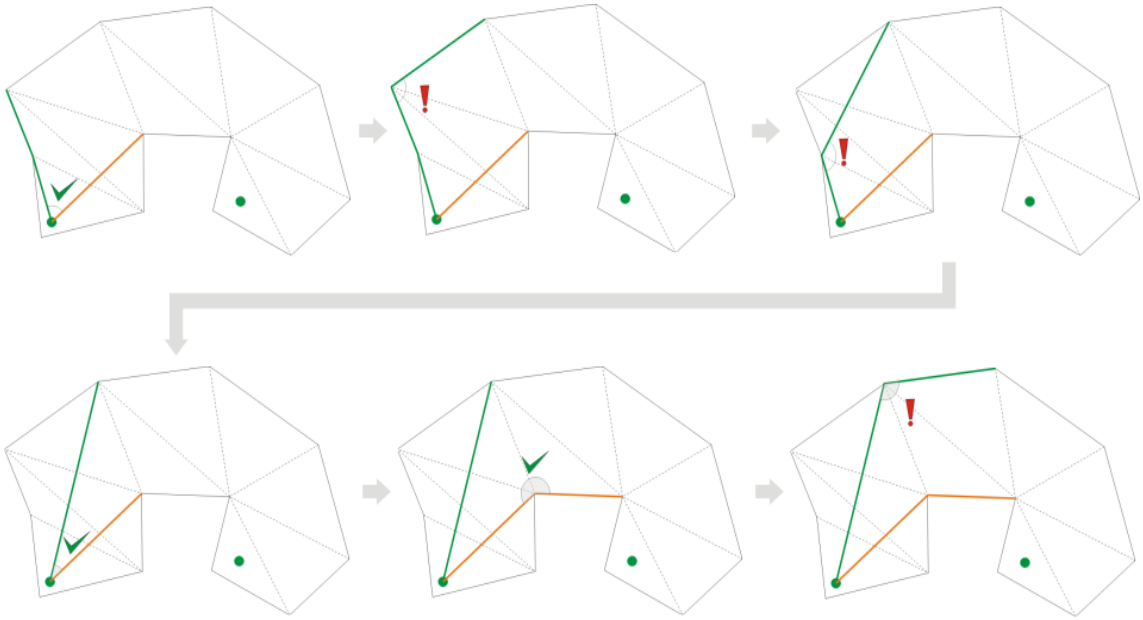
– pokud $V_1$ je apex, nastaví se $V_2$ na apex

– opakuje se krok 3.

**8. Přidání konce**

– za M zvolíme cílový vrchol (GOAL), uplatníme krok 3., stranou (S) bude například levá

– do cesty se přidají všechny vrcholy z levé strany funnelu počínaje prvním vyjmutým a konče apexem (včetně)



*Ilustrace 10: Logika přidávání vrcholů*

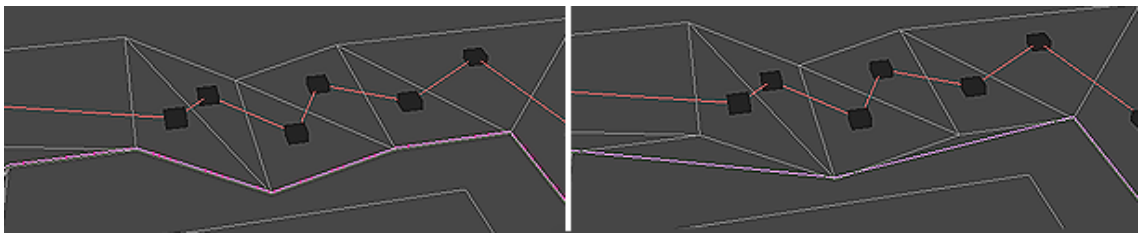
Krok 2 se opakuje s lineární závislostí na počtu bran. V každém kroku se pak vyřadí z původní cesty jeden vrchol. Proto lze tento algoritmus provádět už za běhu například průchodu nebo ho lze jednoduše editovat při změně prostředí (například překážka na trase). Navíc algoritmus lze

rozšířit i do třetího rozměru. V průběhu přidávání vrcholů se kontrolují vnitřní úhly tak, aby nebyly konvexní. Levá a pravá část pak skutečně připomínají trychtýř, po čemž dostal algoritmus i svůj název.

## 2.6 Korekce cesty

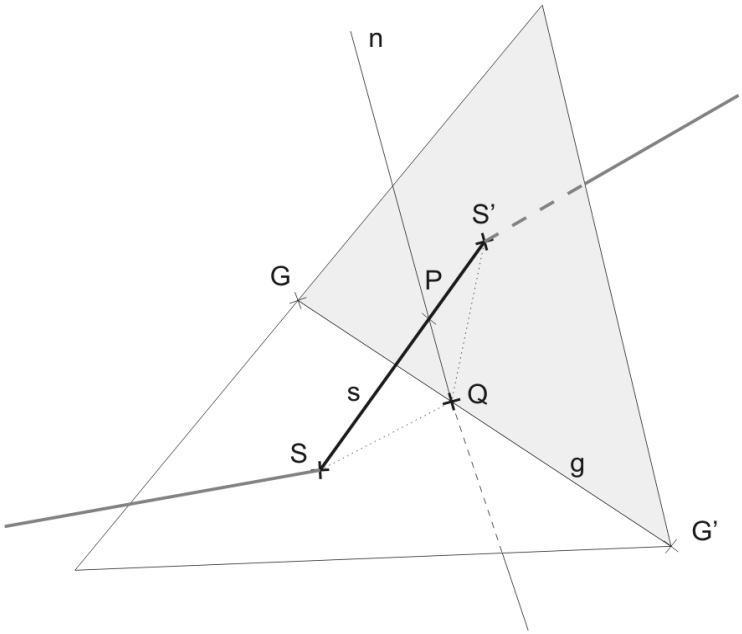
Po provedení Dijkstrova algoritmu a následně funnel algoritmu je cesta již relativně optimální, ale není zaručeno, že je korektní. V případě, že mezi jednotlivými polygony kanálu budou výškové rozdíly, respektive terén kanálu nebude v rovině, je třeba finální cestu upravit tak, aby se jednotlivé její fragmenty nedostávaly pod povrch nebo nevznášely nad ním.

Tato korekce se provádí při přidání dalšího bodu cesty. Při přidávání bodu do cesty se uloží mimo jiné i index brány, ke které bod náleží. Pak se pro všechny brány mezi posledním bodem cesty a aktuálně přidávaným provede korekce. Nekorektní cesta se pomocí roviny definované směrovým vektorem fragmentu cesty, vertikálním vektorem (0; 1; 0) a počátečním bodem fragmentu promítne do konkrétní brány. Pokud se nalezne průsečík této roviny a úsečky (brány), přidá se do nové cesty. Takto se postupuje pro všechny brány v daném úseku. Nakonec se přidá i koncový bod fragmentu.



*Ilustrace 11: Korekce cesty*

Na obrázku (Ilustrace 11) je názorně poukázáno na význam korekce cesty. Levá část představuje opravenou cestu, pravá pak cestu bez zapnuté korekce. Je evidentní, že původní cesta nekopíruje povrch navigačního meshe. Pro výpočet průsečíku zjištění, jestli existuje, se použije Gaussova eliminace. Jak se zjistí matice hodnot a vektor pravé strany je popsáno na následujících řádcích a obrázkem (Ilustrace 12).




*Ilustrace 12: Korekce cesty - geometrie*

Podle obrázku je evidentní, že je potřeba přidat další bod do křivky cesty. Jedná se o bod Q, který bude přidán mezi body S a S'. Přímka *n* je určena body P a Q a její směrový vektor je (0; 1; 0) pro náš případ. Přímka *g* je brána mezi dvěmi znázorněnými trojúhelníky.

| | |
|---|---|
| $P = S + k \cdot \vec{s}$ | *rovnice cesty s* |
| $P = Q + l \cdot \vec{n}$ | *rovnice pomocné přímky n* |
| $Q = G + m \cdot \vec{g}$ | *rovnice brány g* |
| $S + k \cdot \vec{s} = Q + l \cdot \vec{n}$ | *Dosazením rovnice cesty do rovnice pomocné přímky získáváme rovinu, ve které leží obě přímky.* |
| $S + k \cdot \vec{s} = G + m \cdot \vec{g} + l \cdot \vec{n}$ | *Následným dosazením z rovnice brány do rovnice roviny dostaneme už vztah pro náš bod.* |
| $m \cdot \vec{g} + l \cdot \vec{n} - k \cdot \vec{s} = S - G$ | *Po pár úpravách máme rovnici ve správném tvaru.* |
| $m \cdot g_x + l \cdot n_x - k \cdot s_x = S_x - G_x$ | *Při rozepsání na jednotlivé složky podle souřadnic vidíme,* |
| $m \cdot g_y + l \cdot n_y - k \cdot s_y = S_y - G_y$ | *že máme soustavu tří rovnic o třech neznámých, použijeme* |
| $m \cdot g_z + l \cdot n_z - k \cdot s_z = S_z - G_z$ | *Gaussovu eliminaci pro zjištění hodnot koeficientů k, l, m.* |

Jelikož se nejedná o přímky, ale o úsečky, je potřeba ve finále testovat hodnotu parametru *m*, jestli je z intervalu < 0; 1 >.

# 3 Řešení v praxi

Pro koncového uživatele se jedná o vizualizaci vyhledané cesty. Aplikace umožňuje vyhledat cestu pro zajímavé elementy v prostředí. Základní vyhledávání je vyhledávání místnosti. Z pohledu implementace a algoritmu jde o nastavení podmínky přerušení Dijkstrova algoritmu (viz 2.4.3). Pokud je vybrán za vyřešený uzel ten, který má shodný *identifikátor místnosti* jako hledaná místnost dojde k ukončení algoritmu.

Pro vyhledávání osoby probíhá vyhledávání téměř stejně, akorát je potřeba zjistit, v které místnosti se osoba v zadaný čas nachází. Poslední vyhledávaný element je *místo zájmu* (PoI – Point of Interest), který má přímou vazbu na navigační mesh.

Vyhledávání lze omezit tím, zda uživatel může použít výtahy nebo schodiště (bezbariérový přístup). V takovémto případě se může stát, že cesta nebude nalezena vůbec. Aby data v aplikaci stále odrážela reálnou skutečnost, je potřeba tato data aktualizovat. K tomuto účelu byla zvolena architektura klient-server. Klientská aplikace je hlavní součástí práce. Serverová část je zprostředkována pomocí jednoduchého webového skriptu. Komunikace probíhá pomocí protokolu HTTP (Hypertext Transfer Protocol) a má specifickou formu. Pro získávání dat ze serveru se odesílají požadavky v následující formě (viz Tabulka 1).

| Označení | Dotaz | Hodnota dotazu | Typ odpovědi | Hodnota odpovědi |
|---|---|---|---|---|
| Q | findMeDomain | Název domény | - | - |
| Q0 | findMe | Verze programu | konstanta | relevant_server |
| Q1 | findMeQ1 | dataList.xml (řetězec) | XML soubor | dataList.xml |
| Q2 - a | findMeQ1 | Cesta k mesh.obj | OBJ soubor | mesh.obj |
| Q2 - b | findMeQ1 | Cesta k naviMesh.obj | OBJ soubor | naviMesh.obj |
| Q2 - c | findMeQ1 | Cesta k roomMap.map | MAP soubor | roomMap.map |
| Q3 | findMeQ2 | Identifikátor osoby, čas | řetězec | Identifikátor, čas, čas |
| Qh | findMeQh | Cesta souboru | Hash (32 znaků) | MD5 hash souboru |

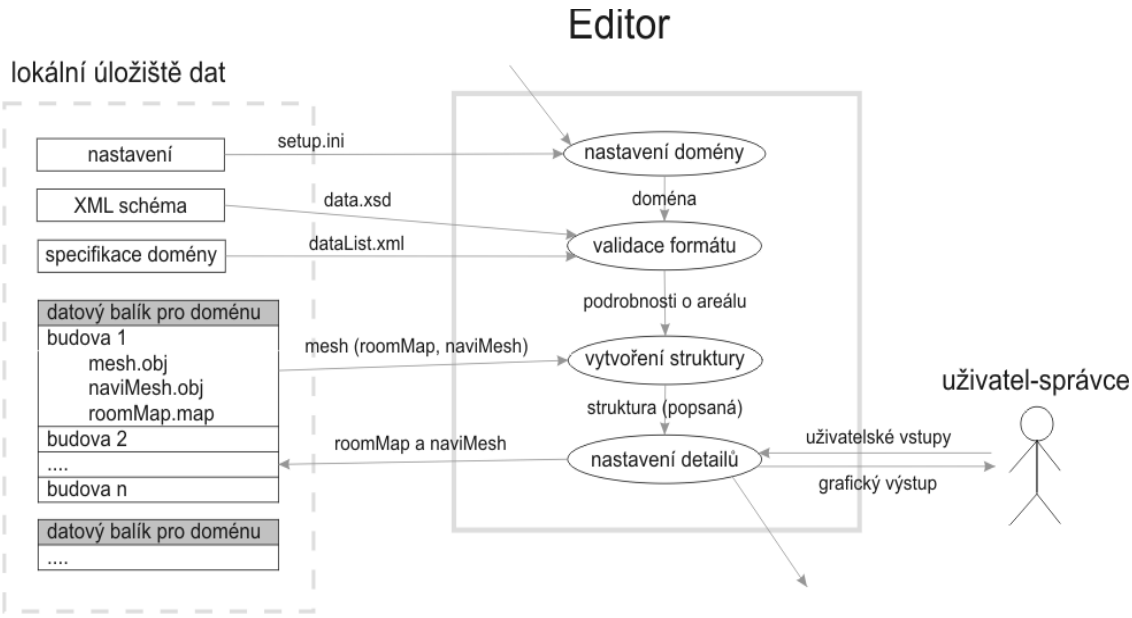
*Tabulka 1: Seznam dotazů*

## 3.1 Vstupní a výstupní soubory

Aplikace pracuje ve dvojím režimu. První režim je administrátorský, ve kterém uživatel

poskytne dva hlavní soubory. XML soubor (*dataList.xml*) se seznamem budov, místností, osob a PoI a OBJ soubor pro každou budovu s její geometrií (*mesh.obj*). Volitelným souborem, který správce může poskytnout, je OBJ soubor (*navi*M*esh.obj*) s navigačním meshem (geometrií).

V rámci práce v administrátorském režimu správce přiřazuje objekty (místnosti, PoI) a nastavuje omezení (výtahy, schodiště) pro jednotlivé části navigačního meshe budovy. Výsledek lze poté uložit do dvou souborů. Jeden slouží pro uložení navigačního meshe v případě, že soubor s ním neexistuje, druhý je pro uložení namapovaných místností (a jiného) na mesh. Tyto jednotlivé kroky jsou zachycené na schématu (Ilustrace 13), kde přechody ukazují tok dat.




*Ilustrace 13: Schéma Editoru*

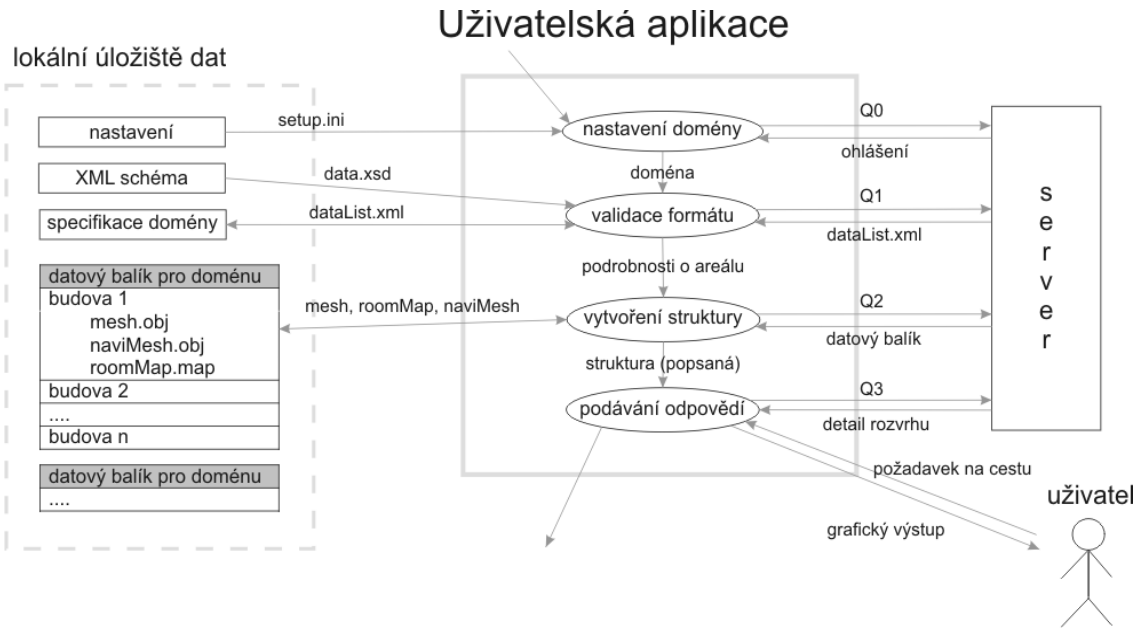
V druhém režimu se struktura nedá v prostředí měnit a slouží pouze pro vyhledávání. Řekněme, že se jedná o část aplikace pro koncového uživatele. Jak už bylo zmíněno dříve, aplikace synchronizuje data s externím zdrojem (konkrétně webovým serverem). K tomu používá dvojích dotazů, informativních a datových.

Informativní jsou Q0 a Qh, datové jsou ostatní. Qh slouží pro zjištění otisku souboru pomocí hashovací funkce MD5. Jedná se o alfa-numerický řetězec (hexadecimální soustava) o 32 znacích. Pokud otisk nesouhlasí s otiskem souboru u uživatele, v dalším kroce aplikace zažádá o nový soubor datovým požadavkem. První myšlenkou, jak zjistit pozici osoby (ve které místnosti se nachází) v závislosti na čase, bylo implementovat plánovací strukturu a data do ní

jednorázově přenést. To však s sebou přináší komplikaci v porozumění problematiky plánování.

Z obecného hlediska osoba může mít velmi specifický harmonogram. Pokud by se jednalo o zaměstnance, který pracuje na směnný provoz, může se tento plán každý týden měnit. V případě zaměstnance školy nebo jiného „sezónního“ zaměstnance může být plán pro každý týden stejný ovšem kromě týdnů mimo sezónu. Dále zde dochází i ke konfliktům, co se týče nejmenší měřené jednotky (hodina proti školní hodině). Z těchto důvodů bylo zvoleno ponechat řešení této části na straně serveru a dotazovat se na výsledek pouze stylem : „Osoba *Jan Novák* v *16:34* je kde?“

Rozdíly proti administrátorskému režimu jsou zejména v dotazování na server a role uživatele (viz Ilustrace 14). Společně se všemi požadavky se odesílá i doména (Q) pro případ, že server slouží pro více domén.




*Ilustrace 14: Schéma Uživatelské aplikace*

### 3.1.1 XML formát

XML neboli Extensible Markup Language je rozšířený strukturovaný datový formát, jehož specifikace spadá pod konsorcium W3C. Primárním účelem tohoto formátu je šíření dat po internetu. Jedná se o textový formát, který lze jednoduše strojově číst a je srozumitelný i pouhým lidským okem.

Základním stavebním prvkem struktury XML dokumentu je element, který je uvozený

*tagem* neboli značkou. Element má textové tělo a může mít přiřazené atributy (viz Text 1). V rámci těla může element obsahovat další elementy a tím se vytváří struktura dokumentu.

```
<room id="a1">
    <name>Přednášková místnost A1</name>
    <description>Sidlo velkych vedatoru a mezinarodni org. mozku</description>
</room>
```

*Text 1: Ukázka XML elementu*

Struktura dokumentu se dá dále definovat pomocí schémat, ať už jsou typu DTD, Relax-NG a nebo XSD. Validací oproti schématu lze zaručit, že dokument bude mít námi požadovanou strukturu. Omezeními mohou být názvy jednotlivých tagů, jejich hierarchické uspořádání, kardinalita nebo parcialita, datové typy atributů a další (v závislosti na typu schématu). Pro tuto aplikaci bylo zvoleno XSD (XML Schema Definitions) a v souboru *data.xsd* je popsán formát vstupního XML souboru se seznamem objektů. Pro poskytnutí správného formátu lze využít převodní transofmace XSLT, pomocí které lze převádět XML dokumenty do jiného formátu.

### 3.1.2 OBJ formát

OBJ formát je textový formát pro popsání třírozměrné geometrie včetně normálových vektorů, texturovacích souřadnic, materiálů a možnosti popsat například bezierovy křivky. Pochází z rukou společnosti Wavefront Technologies, která vyvíjela animační software pro trikové záběry některých hollywoodských filmů. Jelikož se jedná o textový formát, je jej jednoduché implementovat a díky tomu, že aplikace společnosti Autodesk mají možnost data exportovat v tomto formátu, byl zvolen pro tuto aplikaci.

Uložení dat probíhá tak, že nejprve se definují vrcholy a jejich souřadnice v prostoru (*v*). Pak se volitelně definují normálové vektory (*vn*) a mapovací vektory (*vt*). Poté už lze definovat jednotlivé polygony jedním z povolených zápisů. Určení vrcholů se dá zapsat v závislosti na přítomnost nebo absenci normálovém vektoru (případně mapovací souřadnice) následujícími způsoby *v/vt/vn* při všech složkách, *v* pouze při složce vrcholu, *v//vn* při absenci mapovací složky a *v/vt* při absenci normálového vektoru.

```
# Max2Obj Version 4.0 Mar 10th, 2001
#
# object Spindle01 to come ...
#
v  0.000000 1.388026 0.000000
v  0.632014 2.938521 0.000000
v  -0.316007 2.938521 -0.547340
# ........
# 59 vertices

vn  -0.463109 0.376987 0.802129
vn  0.473093 0.323625 -0.819421
vn  0.473093 0.323624 -0.819421
# ........
# 59 vertex normals

g Spindle01
f 1//1 15//15 16//16
f 1//1 16//16 17//17
f 1//1 17//17 15//15
# ........
# 110 faces
g
```

*Text 2: Ukázka OBJ zápisu z mark.obj*

V ukázce (Text 2) je použita varianta při absenci mapovací složky. Polygony jsou definovány třemi složkami tudíž se jedná o trojúhelníky. Ukázka je zkrácená část souboru exportovaného z programu 3D Studio Max 9 od společnosti Autodesk.

### 3.1.3 MAP soubor

Jedná se o jednoduché textové úložiště. Do souboru tohoto typu se ukládají čtyři různá přiřazení. Prvním je přiřazení uzlů grafu k místnosti, čímž se definuje, kde všude se místnost rozprostírá na navigačním meshi. Řádek je uvozen identifikátorem místnosti (získaným z dataList.xml), poté následuje výčet interních identifikátorů pro uzly grafu.

Druhým ukládaným přiřazením je přiřazení PoI k uzlu grafu (části navigačního meshe) a jeho pozice v prostoru. Řádek je uvozen klíčovým slovem „!poi", následuje identifikátor, poté tři souřadnice v plovoucí řádové čárce *x, y, z* a poslední je interní identifikátor uzlu grafu.

Předposledním přiřazením je omezení teleport. Jedná se o výtahy a podobné (třeba i teleporty). Z hlediska grafu je to spojení dvou nebo více uzlů hranami s nulovým ohodnocením. To znamená, že při vyhledávání se uvažuje vzdálenost dvou takto spojených bodů za zanedbatelnou. Uvozením řádku je klíčové „!restrict teleport", následuje název teleportu a potom seznam interních indetifikátorů uzlů grafu.

Posledním zapisovaným přiřazením jsou schodiště a jiné nesjízdné plochy. Řádek je uvozený klíčovým „!restrict stairs" a pak opět následuje seznam interních identifikátorů uzlů grafu. Na přiložené ukázce (Text 3) je uložen popis pro dvě místnosti, jeden výtah o dvou stanicích, dva PoI a schodiště přes čtyři polygony navigačního meshe.

```
c1 NODES_n6 NODES_n7
c2 NODES_n25 NODES_n33 NODES_n24 NODES_n29 NODES_n28 NODES_n27 NODES_n44
!restrict teleport Teleport1 NODES_n19 NODES_n41
!poi poi2 -6.551354250119217 15.843704511128538 -30.76591171261893 NODES_n9
!poi poi5 2.610801416915685 15.85568897350335 -18.008959095280215 NODES_n15
!restrict stairs NODES_n48 NODES_n49 NODES_n50 NODES_n51
```

*Text 3: Ukázka MAP zápisu z cvičného souboru roomMap.obj*

## 3.2 Volný pohyb avatara

Avatarem je myšlena reprezentace uživatele ve virtuálním světě. Avatar se může pohybovat dvojím způsobem. Buď je jeho pohyb neomezený nebo je omezený prostředím – nemůže vrážet do překážek, působí na něj gravitace a jiné síly.

Prvním zkoumaným přístupem k omezenému pohybu avatara je navržení a implementace vlastního modulu pro výpočet fyzikálních jevů (fyzikální engine), který by detekoval kolize avatara s prostředím, řešil odraz po srážce a aplikoval na něj gravitační a jiné síly. Při velké složitosti prostředí by složitost výpočtu pro hledání kolize byla přímo úměrná počtu polygonů v prostředí. Existují jisté optimalizace, které složitost radikálně sníží (viz 4.2 Obecné optimalizační metody).

Namísto tohoto přístupu byl využit fakt, že avatar by měl mít možnost se pohybovat pouze v rámci navigačního meshe a toho, že konkrétní prostředí je víceméně statické. Proto se avatar uvažuje jako agent navigačního systému. K samotnému pohybu pak dochází v několika krocích. Nejprve je potřeba zmínit, že navigační mesh se uvažuje za dvourozměrný a přirozeně schůdný (bez kolmých stěn, stropů aj.) a tudíž při omezení pohybu v něm lze výškovou složku ignorovat.

V prvním kroku pohybu se zjistí, zda nová pozice je uvnitř výchozího polygonu. Pokud ano, promítne se bod rovnoběžně s výškovou osou do roviny polygonu a algoritmus skončí. V opačném případě se zjistí, kterou bránu protíná úsečka vytyčená stávající pozicí a novou pozicí a pro polygon na druhé straně brány se tyto kroky algoritmu opakují. Je potřeba za běhu ukládat informaci, které polygony byly již testovány, aby nedošlo k zacyklení. Ideální případ je

testování na jeden až dva polygony – výchozí polygon a jeden soused. Při zvýšení velikosti kroku avatara náročnost může vzrůstat. Je tu určitá analogie s detekcí kolize pomocí fyzikálního enginu. Neustále však uchováváme informaci o polygonu, kde se avatar nachází, a proto se vždy testují nejbližší polygony jako první.

## 3.3 Výběr pomocí kliknutí myši

Jednou metodou, jak přistupovat k řešení problému, na který objekt bylo kliknuto kurzorem myši, je metoda sledování paprsku (raytracing). Při úplné implementaci fyzikálního enginu se jedná o triviální úlohu. Je potřeba ale zajistit, aby se pro kontrolu průniku paprskem uvažovala pouze relevantní tělesa. Zejména ta, která jsou uvnitř zorného pole. Toho lze dosáhnout některými optimalizačními postupy zmíněnými v kapitole 4.

OpenGL má způsob, jak dosáhnout tohoto cíle jinou cestou. Každému vykreslenému objektu lze přiřadit „jméno“. Nejedná se o jméno v běžném slova smyslu, protože se ukládá jako celočíselný datový typ. Jména se dají i vnořovat a tím lze dosáhnout toho, že nejen že víme, který objekt byl vybrán, ale lze zjistit, která jeho část.

Objekty je potřeba vykreslit v „select“ módu a určit jména. V aplikaci k tomuto režimu vykreslování dochází pouze při zjištění, že bylo stisknuto tlačítko myši, aby nedocházelo k nadbytečným výpočtům. Přejde se tedy do vykreslovacího režimu „select“, scéna se vykreslí i se jmény, ale pouze její malý výřez, který odpovídá pozici myši. Během vykreslování je naplněný „select buffer“, po skončení se přejde do standartního režimu vykreslování, a tím je fáze získávání dat skončena.

Do bufferu se uloží informace o všech objektech, které byly po tu dobu vykresleny do zvoleného výřezu. V dalším kroku je třeba získaná data zpracovat, což znamená pro náš případ získat nejblížší objekt, který byl vybrán.

Select buffer má následující formát:

počet jmen pro první objekt (n)

minimální hloubka objektu ve výřezu (z-minimum)

maximální hloubka objektu ve výřezu (z-maximum)

jméno 1

….

jméno n

Poté následují data pro další objekty. Ty jsou řazeny podle hloubky, tudíž objekt nejblíže kameře a tudíž i ten, po kterém se ptáme, je první v bufferu. Počet jmen je (pokud se jedná o objekt se jmény) vždy roven dvěma. Vyplývá to z topologie vykreslování. První jméno je index objektu GPrototype v seznamu objektu GEnvironment a druhé jméno je index objektu GEntity v seznamu objektu GPrototype. Více o této hierarchii v kapitole 3.5.2 a 3.5.4. Hloubky uložené v bufferu jsou hodnoty ze z-bufferu vynásobené $2^{32} – 1$ a zaokrouhlené na nejbližší integer.

Pro zjištění přesné polohy bodu v prostoru, kde bylo kliknuto, se používá OpenGL funkce (gluUnProject), která zpětně transformuje pozici ve výřezu a hloubku ze z-bufferu do vektoru souřadnic v prostoru. Další variantou je použít průnik paprsku s nyní už známým objektem. Varianta výběru objektů pracuje s vykeslováním objektů do výřezu.

Pro zanedbání některých detailů objektů, které by mohly nežádoucím způsobem ovlivnit výběr, lze vykreslit v „select“ módu zcela jiný objekt než ten, co je vykreslen ve standartním módu. Protože pouze objekty vykresleny ve standartním módu budou zobrazeny. Toho lze využít i pro zjednodušení vykreslené scény a snížení časových nároků.

## 3.4 Zobrazení minimapy

Jako minimapu uvažujeme obraz, který se naskytne z ptačí perspektivy. Minimapa se dá dělit podle toho, zda je ve výřezu zobrazena fixně scéna (co se týče otáčení) a při otáčení avatara se otáčí avatar na minimapě. A nebo naopak, kdy se avatar neotáčí, ale otáčí se scéna.

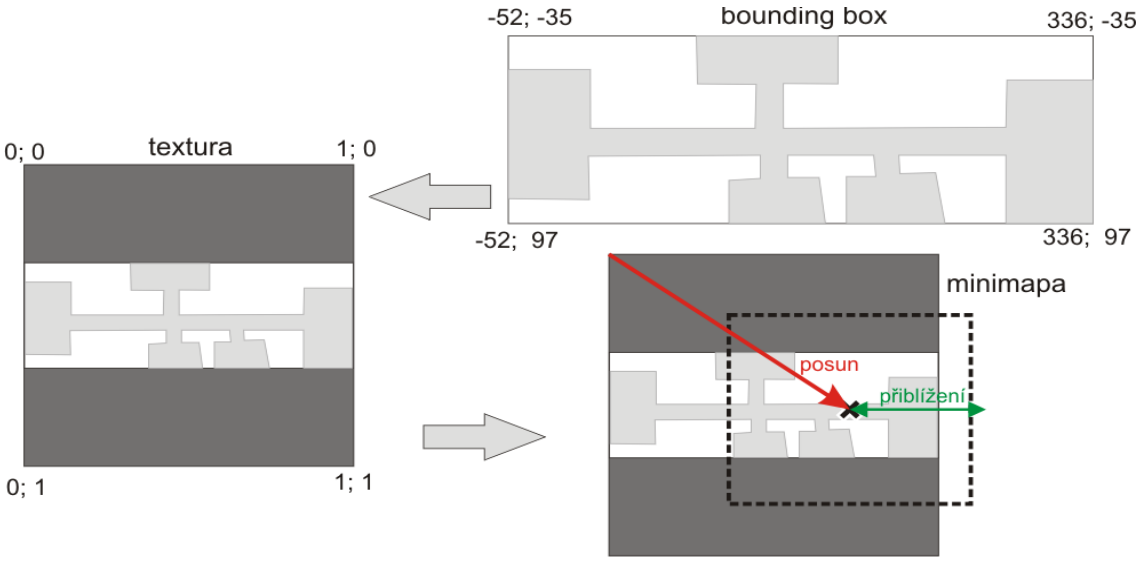
Jedním, nejjednodušším přístupem, jak zobrazit minimapu, je vytvoření nového viewportu, nastavení kamery a jeho napasování na část hlavního okna. Výhodou je vysoká flexibilita, kamera neustále zabírá aktuální stav scény. Velkou nevýhodou a rovněž důvod, proč nebyl zvolen tento způsob, je fakt, že veškerá data jsou zpracovávána znovu a tudíž se nároky na vykreslení scény mohou až zdvojnásobit (v závislosti na optimalizaci a velikosti minimapy).

Druhým způsobem je danou scénu předkreslit a na místo minimapy umístit už jen obdélník s namapovanou uloženou texturou. Nevýhodou je nízká flexibilita, protože textura je vykreslena pouze na začátku, a tudíž při bližším pohledu na minimapu může ovlivnit estetický dojem relativně nízké rozlišení textury.

Hlavní problém s vykreslením minimapy nastává v případě, kdy se některé části scény překrývají. Příkladem z reálného světa může být hora s jeskyní uvnitř. Pokud jsme vně, na hoře, chceme zobrazit povrch hory. Avšak pokud jsme uvnitř, v jeskyni, chceme zobrazit vnitřek jeskyně. Myšlenka řešení této úlohy tkví v nalezení co možná nejmenšího počtu vzájemně se nepřekrývajících (při průmětu do roviny půdorysu) skupin polygonů z navigačního meshe. Za

předpokladu, že jednotlivá patra budov se liší jen velmi málo, se této problematice nevěnovala velká pozornost a za dostatečné řešení se vzalo vykreslení jednoho patra jako minimapa pro všechna ostatní.

Vykreslení minimapy je aplikace textury na připravený obdélník v rohu obrazovky. Aby docházelo k relativnímu pohybu avatara po minimapě (avatar je vystředěný, ale pohybuje se povrch pod ním), je třeba nejdřív pochopit, jak dochází k mapování textury na povrch. V třírozměrném prostředí je konvence vyhrazovat písmena *x, y, z* pro osy souřadného systému. Pro mapovací složky je konvencí používat označení *u, v (w)* pro vertikální a horizontální mapovanou souřadnici vrcholu na texturu. Rozměry textury v OpenGL jsou pro učely mapování normalizované tak, že je definovaná v jednotkovém čtverci. Mimo tyto rozměry lze dodefinovat chování textury (opakování, zrdcadlení, barva okrajových texelů ...).




*Ilustrace 15: Vytvoření minimapy*

Na obrázku (Ilustrace 15) je ve třech krocích ukázáno, jak se vytváří a aplikuje minimapa. V horní části obrázku je horní pohled na celé prostředí, které je obalenou obalovým obdélníkem, který je orientovaný souběžně se souřadnicovým systémem (AABB – axis-aligned bounding box).

Nastaví se patřičně kamera, vykreslí se scéna do bufferu a odtud se zkopíruje a uloží jako textura (levá část obrázku).

V dolní části obrázku je znázorněna textura a přerušovaný obdelník je již minimapa. Je

potřeba vypočítat správné mapování – to se skládá ze dvou složek. První složka je vektor posunutí středu a druhá skalární - přiblížení minimapy. Přiblížení je pevná hodnota získaná z uživatelského vstupu. Je zapotřebí, aby byla větší než nula, aby nedošlo ke zrdcadlovému efektu. Pro zjištění složek vektoru posunu je potřeba reálnou pozici avatara normalizovat za pomoci rozměrů původního obalového obdélníku.

## 3.5 Implementace

V dalších podkapitolách budou představeny některé třídy, některé podrobněji, jiné méně. Nahlédneme do jejich významu pro aplikaci a práce s nimi. Spojnicí mezi Java komponentami, jejich událostmi a obecnou objektovou strukturou nezávislou na komponentách je statická třída *Loader*. Ta se za prvé stará o zprostředkování přístupu k datům pro naplnění například seznamu místností, PoI a jiných nebo vykreslovacím metodám. Za druhé je to pak načítání nastavení aplikace a dat ze souborů. Udržuje veškeré instance třídy GEnvironment.

### 3.5.1 Třída Mesh

Mesh v oblasti počítačové grafiky je označení pro trojrozměrnou, síťovou geometrii tělesa. Třída *Mesh* a její objekty se tedy starají o uložení dat geometrie a operace nad nimi. Metoda *FilterByAngle* odebere z *meshe* všechny plochy, jejichž normálový vektor nespadá do tolerance maximální odchylky od zadaného normálového vektoru. Tato funkcionalita se používá při generování navigační struktury, aby se odstranily neschůdné plochy. *SetUpStaticData* je metoda, která danou geometrii předvykreslí do tak zvaného *DisplayListu*, což je struktura OpenGL.

Jedná se o metodiku uložení příkazů vykreslení přímo do paměti grafické karty a následně lze pak provést jedním příkazem jejich volání. Největším přínosem je dramatické urychlení vykreslování. Odpadne potřeba přesunovat neustále (každý snímek) data z operační paměti do paměti grafické karty. Nevýhodou je případ, kdy tato data jsou často měněna. V takovémto případě se musí znovu nahrát do paměti grafického akcelerátoru. Proto je tato metoda vhodná pro statické předměty ve scéně, jako jsou například budovy, terén krajiny a jiné předměty, u kterých se geometrie nemění.

Podstatnou metodou pro navigační algoritmy jsou *ClockWiseVectors2d*, která vezme pouze „výškovou“ složku vektorového součinu a určí, jestli vstupní vektory jsou pravotočivé nebo levotočivé. Poslední zmíněnou metodou je statická metoda třídy *Gtriangle*. Nazývá se *HasGate* a určuje společné dva vrcholy zadaných trojúhelníků a tudíž i „bránu“ mezi nimi.

```
  // nastavení display listu a vykreslení do paměti grafické karty (inicializační část)
GL.glNewList(this._displayListIndex, GL.GL_COMPILE);
for (GTriangle item : _triangles)
{
   // vykreslení jednotlivých trojúhelníků
}
GL.glEndList();
Mesh.lastDisplayListIndex++;
  /*……
     ……
     …… */
  // volání funkce pro vykreslení z display listu (už v rámci vykreslovaní každého snímku)
if (this._displayListIndex != -1 && Mesh.loaded)
{
    GL.glCallList(this._displayListIndex);
}
```

*Text 4: Kód použití DisplayListu*

### 3.5.2 Třídy GPrototype a GEntity

Tyto dvě třídy slouží pro manipulaci s objekty ve scéně a jejich vykreslování. Důvodem, proč k tomuto účelu jsou použité dvě třídy, je úspora paměti. Třída *GPrototype* je zástupcem prototypu (jak její název napovídá) objektu. Nese data jeho geometrie a metody s tímto spojené. Rovněž se stará o entity vytvořené od tohoto prototypu. „Stará“ v tomto kontextu znamená, že obsahuje datový typ pro udržení jejich referencí a metody pro přístup k nim. V podstatě se GPrototype stará o vytváření různých typů grafických objektů a *GEntity* společně s jejím „rodičem“ *GObject* jsou pak třídy, které vytváří konkrétní individuální objekty mající vlastní pozici, otočení a aktuální stav animační sekvence.

Zde je třeba rozlišovat pojmy „sekvence” a “animace”. Animace v rámci této práce bude soubor po sobě jdoucích animačních klíčů. Sekvence je pak soubor po sobě jdoucích animací. Přechod mezi jednotlivými klíči animace je řešen pomocí časově a prostorově lineární interpolace.

### 3.5.3 Třídy GGraph, Navigation a Funnel

Následující tři třídy pohání celý proces vyhledávání nejkratší cesty a její následné vyhlazení. Postupy a algoritmy již byly zmíněné v předešlých kapitolách (kapitoly 2.4.3, 2.5 a 2.6). Zde si pouze představíme jejich strukturu a důležité datové konstrukty, které využívají. *GGraph* společně s třídou *GNode* a *GEdge* reprezentují graf (z teorie grafů ovšem). GNode reprezentuje uzel a GEdge hranu. Metoda *Path* třídy GGraph vypočítává nejkratší cestu v grafu. Metoda

*RealDistance* vypočítává euklidovskou vzdálenost mezi dvěma uzly v grafu a slouží pro ohodnocení hran.

Třída *Navigation* je odvozena od třídy GGraph a metodou *CalculatePath* zaštiťuje celý navigační modul. Mimo zmíněné metody má třída Navigation další dvě podstatné metody. První, *MakeNaviMesh* filtruje mesh pomocí dříve zmíněné metody FilterByAngle. Metoda *MakeGraph* vygeneruje graf z navigačního meshe pomocí HasGate, jak již bylo předesláno.

V této metodě se mimo jiné kontroluje překrývání dvou sousedních částí navigačního meshe v rovině půdorysu. Hrana, která by pak v grafu odpovídala spojení těchto dvou částí, je zahozena. Při ponechání docházelo při výpočtu funnel algoritmu k nežádoucímu jevu prohození logiky pravé a levé strany. Ve výpočtu se tak začala uvažovat možnost „chodit hlavou dolů“.

A konečně třída *Funnel* společně s třídou *Deque* jsou prostředníky k aplikaci *funnel* algoritmu. Třída Deque je implemetací obousměrné fronty. Jedná se o obousměrný spojový seznam. Každá položka seznamu obsahuje odkaz na předcházející a následující položku. Třída navíc obsahuje metody pro přímý přístup ke kontrétnímu n-tému prvku zleva či zprava a další metody pro vyjmutí (např. *PopLeft*) posledního prvku a vložení posledního prvku (*PushLeft*). Třída Funnel obsahuje dvě metody pro přidávání vrcholů na příslušné konce funnelu a to *AddPointLeft* a *AddPointRight*. Obě metody v rámci své funkcionality volají metodu *FixedAddPath*, která přidá korektní fragmenty cesty (viz kapitola 2.6 Korekce cesty). Čtvrtá metoda *ProcessFunnel* prochází přes části kanálu a pro jednotlivé brány volá buďto metodu AddPointLeft nebo AddPointRight. Na konci provede dodatečné přidání do cesty.

### 3.5.4 Třídy GEnvironment a ostatní

Třída *GEnvironment* je správce veškerých dat daných pro jednu budovu. Implementuje rozhraní *RenderToTexture*, které slouží jako náhrada anonymních funkcí v prostředí Java. Absence anonymních funkcí v jazyku Java se dá řešit právě pomocí rozhraní. Namísto předávání anonymní funkce se předává objekt implementující dané rozhraní. Třída GEnvironment je spojnicí mezi „správcem“ okna (GView) a vším ostatním. Obsahuje „globální“ proměnné jako je aktivní kamera a úložiště pro prototypy. Spravuje hlavní vykreslovací funkci, která aplikuje transformace kamery a vedlejší funkci pro vykreslení orientační mřížky a vykreslení minimapy. V konstruktoru se provádí většina inicializačních kroků, například načítání dat ze souborů, vytváření objektů nebo inicializace textur.

V poslední řadě budou krátce představeny nezmíněné třídy s popisem jejich významu. Třída *GCamera* reprezentuje kameru. Z pohledu OpenGL, vykreslování a transformací není rozhodující, zda se provede transformace na kameru nebo inverzní transformace na vše ostatní.

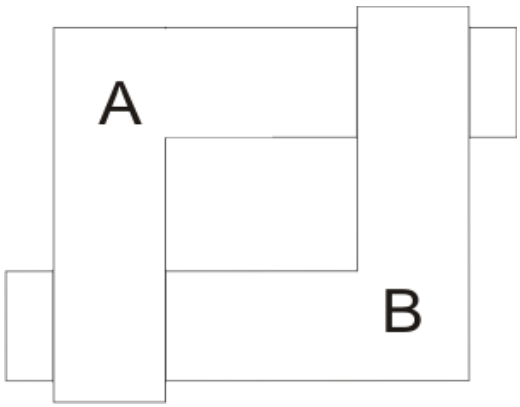
Z praktického hlediska je jedno, jestli se kamera přiblíží ke scéně nebo scéna ke kameře. Výsledný obraz bude stejný v obou případech. To je i důvod, proč při aplikace pozice v metodě vykreslení kamery má pozice hodnoty vynásobené -1.

*GLight* je elementární třída. Společně s GCamera jsou obě odvozené od GObject a tudíž animovatelné. Třída *ObjMeshLoader* se týká načítání ze souborů s příponou „obj“. Jedná se o textový formát od Wavefront Technologies zmíněný již v předešlé kapitole (3.1.2 OBJ formát).

# 4 Optimalizační metody

## *4.1 Malířův algoritmus*

Jak už název napovídá, malířův algoritmus se používá pro vykreslování scény. Princip je jednoduchý, vykreslují se nejprve objekty, které jsou v pozadí a poté (přes ně) objekty v popředí. To částečně zaručí, že objekty v popředí správně překryjí ty, které jsou v pozadí. Ovšem dochází zde ke kritickým situacím, kdy část jednoho objektu překrývá část jiného, a ten zase naopak překrývá část prvně zmíněného (viz Ilustrace 16).

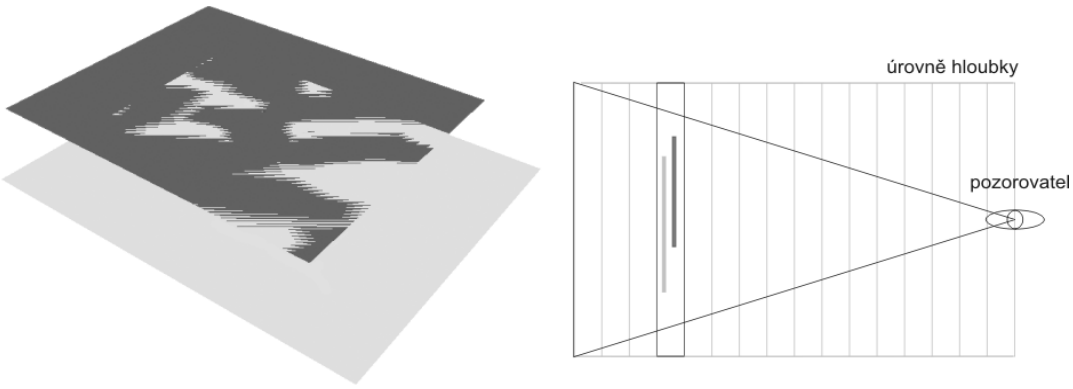



*Ilustrace 16: Problém malířova algoritmu*

Za předpokladu, že bychom těmto situacím dokázali zabránit, lze vypnout hloubkové testování (z-buffer testing), a tím urychlit vykreslování celé scény. Je potřeba však zajistit správné pořadí vykreslovaných objektů (například pomocí BSP).

Malířův algoritmus má ještě další použití jiné než optimalizaci. Jelikož hloubkový

buffer je omezen v rámci rozlišení jednotlivých úrovní hloubek, může docházet k artefaktů při vykreslování dvou objektů na stejné nebo velmi blízké úrovni. Na obrázku (Ilustrace 17) je vidět vzniklý artefakt (vlevo) a příčina jeho vzniku (vpravo). Představme si situaci, kdy chceme vykreslit podlahu (jeden objekt) a na ní koberec (druhý objekt). Oba objekty mají, při pohledu shora, stejnou hloubkovou souřadnici a dochází zde k nepřirozenému překryvu. Zde se uplatní malířův algoritmus. Nejprve se vypne hloubkové testování, vykreslí se objekt podlahy, poté objekt koberce, pak se scéna může dále vykreslovat už se zapnutým testováním hloubky.




*Ilustrace 17: Z-fighting artefakt*

Dalším uplatněním malířova algoritmu je při použití průhlednosti pomocí alfa kanálu. Zde se dotýkáme už samotné aplikace. Jelikož při vykreslování průhledných objektů dochází k zápisu do hloubkového bufferu, tak všechny další objekty, které vykreslíme za průhledné objekty se zapnutým testováním hloubky, budou neviditelné, protože neprojdou hloubkovým testováním.

Nejprve se tedy vykreslí všechny neprůhledné objekty a poté průhledné od nejvzdálenějšího k nejbližšímu. Kdybychom ignorovali pořadí průhledných objektů, mohlo by dojít k vynechání aplikace barvy jednoho objektu, protože by neprošel testováním hloubky a byl by v tomto kroku „zahozen“. Poloprůhlednosti je využito pro zobrazení odvrácených polygonů meshe budovy, které by jinak byly neviditelné.

## 4.2 Obecné optimalizační metody

*BSP* neboli *Binary Space Partitioning* v překladu znamená binární dělení prostoru. Jedná se o všeobecně používanou myšlenku využití binárního stromu. *Binární strom* je speciální typ

grafu.

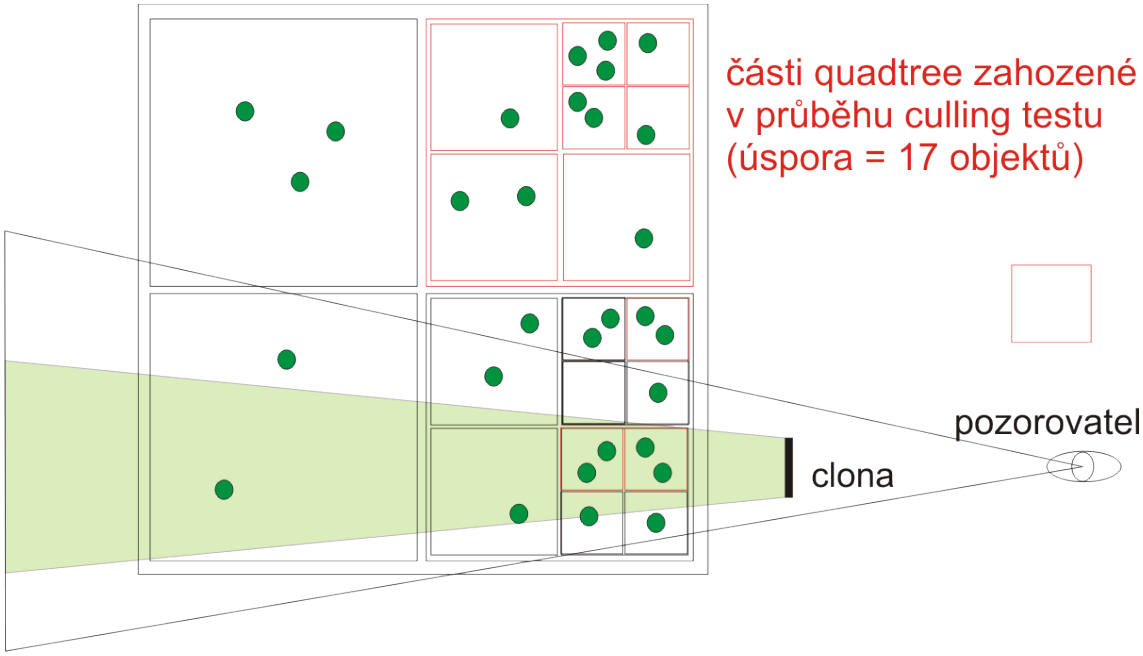
Strom je graf, který neobsahuje cyklus. Vybereme-li jeden z uzlů takového grafu a označíme ho za vrchol stromu, pak jeho přímí sousedé se označují za *potomky* a tento uzel za jejich *rodiče*. To platí obecně v celém stromu. Uzel, který nemá potomky, se občas označuje za *list*. V binárním stromě platí, že každý uzel má nanejvýš dva potomky (proto binární). U aplikace binárního stromu u BSP záleží na pořadí potomků a každý uzel má právě dva potomky.

Princip BSP vychází z rekurzivního dělení prostoru rovinami. Každá rovina rozdělí prostor na dva poloprostory, poloprostor před rovinou a poloprostor za rovinou (použití normálového vektoru). Tím se vytvoří dva potomci pod konkrétním procházeným uzlem. Zde závisí na pořadí, protože pořadí určuje, zda se jedná o uzel před rovinou nebo za ní. Veškeré objekty se při prvním dělení tak rozdělí do dvou skupin. Takto dělení pokračuje do doby, kdy jsou splněny určité podmínky, které závisí na kokrétní aplikaci.

Kromě klasického BSP, které dělí rovinou vždy na poloprostory, existují varianty *quadtree* a *octree.* Jak název napovídá, bude se jednat o struktury vycházející ze stromů mající vždy maximálně čtyři (quadtree), respektive osm (octree), přímých potomků na uzel. Quadtree se využívá zejména ve dvourozměrném prostoru, případně třírozměrném, kde je velmi malé využití ve třetím rozměru („svět je relativně plochý“). Octree se analogicky ke quadtree používá ve třírozměrném prostoru.

Princip dělení u quadtree (u octree se jedná pouze o analogii) je, že celý „svět“ se zapouzdří do jednoho čtverce a ten se dále dělí na další čtyři a rekurzivně se pokračuje, dokud nejsou splněné podmínky přerušení. Podmínkami většinou bývá počet objektů. BSP a quadtree se využívají zejména ve zobrazování složitějších scén, kde se této techniky používá k ořezávání neviděných grafických elementů nebo při výpočtu kolizí, kde není třeba testovat kolizi dvou objektů, které nejsou ani ve stejném kvadrantu. Je potřeba mít na paměti, že udržování konzistentního stromu vyžaduje také režijní čas a v případě, kdy objekt zasahuje do více listů stromu, je třeba s ním počítat (doslova) v každém z nich.

Dříve se BSP používalo i za účelem možnosti vypnutí testování hloubky a zvýšení tak výkonu (viz 4.1 Malířův aloritmus*).* Na obrázku (Ilustrace 18) je demonstrována optimalizace počtu vykreslovaných objektů pomocí ořezávání a rozložení prostoru do *quadtree*. Za dělící požadavek je zvoleno, aby v listu stromu nebyly více jak tři objekty.




*Ilustrace 18: Význam cullingu a quadtree*

## 4.3 Grafické optimalizační metody

V případě, že se optimalizuje vykreslování, existují dva hlavní přístupy. Prvním je snížit počet volání funkcí grafického API a přesun dat z operační paměti do grafické karty a druhým, obecnějším, je snížit elementy, které se budou vykreslovat. Pro snížení volání funkcí se používají prostředky jako jsou display listy, vertex arrays, VBO (Vertex Buffer Object) nebo modernější využití geometry shaderů.

Snížení vykreslovaných elementů lze dosáhnout „cullingem“ neboli ořezáváním. Základní ořezávání je *frustum culling*, ořezávání objektů, které nejsou viděny aktuální kamerou. Tím se zamezí nadbytečnému „scissor testingu“. Druhým a obtížnějším *cullingem* je *occlussion culling*, neboli ořezávání objektů, které jsou zakryté jiným objektem. Jedním ze způsobů, jak toho lze dosáhnout, je zmíněné BSP.

Další metodou je *portal rendering*, při kterém se svět rozdělí na několik menších částí, které jsou mezi sebou propojeny portály. Vykreslují se pak pouze ty části, ve kterých se pozorovatel nachází nebo jejichž portály jsou viděné z pozice pozorovatele. Obdobou je *antiportal*, který představuje překážku, která blokuje ve výhledu.

Poslední zmíněnou metodou v tomto odstavci bude *Level of Detail* (LoD), neboli úroveň detailů. Jedná se o princip, že co není vidět, nebo je málo vidět, nemusí být detailní. Například strom v dálce na horizontu nemusí být zpracováván jako komplexní geometrie, když

ve finále na monitoru bude mít velikost 5x10 pixelů. Principem se podobá LoD *mipmappingu*, akorát důvod je jiný. Grafický objekt může mít více variant sama sebe a podle vzdálenosti od pozorovatele se vždy použije ten, který je pro konkrétní rozmezí vzdáleností určený. Nebo lze využít teselaci nebo jiné algoritmy pro zjemnění povrchu.

## 4.4 Navigační optimalizační metody

Jedním ze způsobů, jak snížit výpočetní čas (konkrétně čas výpočtu nejkratší cesty), je využít efektivnější algoritmus A*, který pro relativně přímé cesty dosahuje menší složitosti než Dijkstrův algoritmus. Dalším způsobem je víceúrovňová hierarchie uzlů (TRA*). Princip této metody tkví ve více úrovních grafu, na kterém se vyhledává. Na nejnižší úrovni je graf stejný jako v případě, že se tato metoda vůbec nepoužije. Na vyšší úrovni se skupiny slučují do dalšího uzlu se zachovanými vazbami. Tak se pokračuje, až zbydou pouze izolované uzly. Při vyhledávání nejkratší cesty postupuje algoritmus od nejvyšší úrovně, čímž velmi rychle i zjistí, zda je cesta dostupná. Na nižších úrovních se pak prochází pouze ty uzly, jejichž ekvivalent ve vyšší vrstvě byl součástí cesty. Při velmi komplexní síti může tento postup ušetřit spoustu výpočetního času.

# 5 Závěr

Cílem práce bylo vytvořit aplikaci, která by řešila vyhledávání nejkratší cesty v reálném modelu. V rámci této práce byla nastíněna teorie potřebná pro vývoj aplikace ve třírozměrném prostoru a pro práci s grafovými algoritmy. Pro formát vstupního souboru byl zvolen textový soubor OBJ, který díky tomu, že je právě textový, je přehlednější a práce s jeho daty transparentnější.

Jako forma reprezentace navigačního prostředí byl zvolen navigační mesh, který je ze zmíněných přístupů nejobecnější a nejflexibilnější. Co se týče algoritmu, byl použit všeobecně známý Dijkstrův algoritmus společně s ne tak známým funnel algoritmem, který vylepšil celkový dojem z výsledné cesty, která tak působí daleko přirozeněji. Poté byla provedena korekce cesty, aby nedocházelo k navigačním artefaktům. Ačkoliv je známá i varianta zmíněného funnel algoritmu, která uvažuje i nenulovou šířku bota, bylo by toto rozšíření součástí až další práce.

Aplikace byla rozšířena o modul potřebný pro komunikaci přes internet a získávání tak externích dat o konkrétních bodech zajímavých pro vyhledávání (PoI). S tím souvisí i možnost nastavit pozice těchto bodů přímo uživatelem v módu *Editor*. Vývoj probíhal v jazyku Java

a grafickém rozhraní OpenGL. U obou platí, že jsou jednoduše přenositelné na většinu rozšířených platform. V úvahu připadá vytvoření příjemnějšího a přitažlivějšího uživatelského rozhraní právě pomocí OpenGL.

V závěru byly představeny některé metody, jak aplikaci, jako je tato, optimalizovat. Jejich implementaci bude při dalším vývoji nutno zvážit, protože ne všechny z nich jsou nezbytně nutné a ne všechny se hodí pro aplikaci tohoto typu. Ať už se jedná o flexibilitu prostředí nebo optimalizaci objemu vykreslované masy. Ideálním použití této aplikace a dalším směrem, kam by se mohl vývoj ubírat, je platforma mobilních telefonů, protože tímto směrem směřují poslední trendy.

# Seznam použité literatury

Demel J. : *Grafy a jejich aplikace*, Academia, 2002, ISBN 80-200-0990-6

Smed, J., Hakonnen H. : *Algorithms and Networking for Computer Games*, Wiley, 2006, ISBN 978-0470018125

*Lighthouse3D - Picking tutorial* [online], [cit. 2011-10-22], URL : <http://www.lighthouse3d.com/opengl/picking/>

Pelikán, J. : *3D počítačová grafika na PC* [online], 2010 [cit. 2011-11-05], URL : <cgg.mff.cuni.cz/~pepca/lectures/pdf/Grafika2003.pdf>

Demyen, D. J. : *Efficient Triangulation-Based Pathfinding* [online], 2007 [cit. 2012-02-13], URL : <http://skatgame.net/mburo/ps/thesis_demyen_2006.pdf>

Khronos Group : *OpenGL FAQ / 15 Transparency, Translucency, and Blending* [online], c1997 [cit. 2012-05-01], URL : <http://www.opengl.org/archives/resources/faq/technical/transparency.htm>

Strachota, P. : *Mapování textur* [online], poslední revize 28.4.2010 [cit. 2011-10-30], URL :<http://saint-paul.fjfi.cvut.cz/base/public-filesystem/admin-upload/POGR/POGR2/11.mapovani_textur.pdf>

Khronos Group : *OpenGL 2.1 Reference Pages* [online], c1997 [cit. 2012-05-01], URL : <http://www.opengl.org/sdk/docs/man/>

Tišnovský, P. : *Seriál Grafická knihovna OpenGL* [online], 2003-2004 [cit. 2011-05-13],

URL : <http://www.root.cz/serialy/graficka-knihovna-opengl/>

Murray, J. D. , Van Ryper, W. : *Encyclopedia of Graphics File Formats*, O'Reilly Media, 1996, ISBN 1-56592-161-5

Wróblewski, P.: *Algoritmy, datové struktury a programovací techniky,* Computer Press, 2004, ISBN 80-251-0343-9

W3C : *Schéma*, c2010 [cit. 2012-04-16],

URL : <http://www.w3.org/standards/xml/schema>

Java.com : *Learn About Java Technology* [online], 2010 [cit. 2012-05-02],

URL : <http://www.java.com/en/about/>