VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

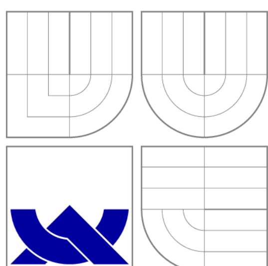
INDEXOVÁNÍ OBJEKTŮ V 3D PROSTORU

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

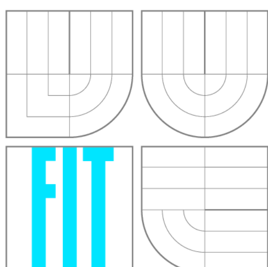
AUTOR PRÁCE
AUTHOR

Bc. MIROSLAV DRBAL

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

INDEXOVÁNÍ OBJEKTŮ V 3D PROSTORU

3D OBJECT SPATIAL INDEXING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. MIROSLAV DRBAL
AUTHOR

VEDOUCÍ PRÁCE — Ing. FILIP ORSÁG, Ph.D.
SUPERVISOR

BRNO 2010

Abstrakt

Diplomová práce vymezuje definici pojmu indexace a v úvodu se zabývá známými indexovacími algoritmy a strukturami pro indexování objektů v 3D prostoru. Je zde diskutován rozdíl mezi indexováním statických - nepohyblivých a pohybujících se objektů. Praktická část diplomové práce je zaměřena na návrh a implementaci indexovacího algoritmu pro open source aplikaci MaNGOS s ohledem na generičnost návrhu a efektivitu výsledné implementace, zejména pak na efektivitu prostorových vyhledávacích dotazů pro vyhledání objektů daných vlastností v zadané oblasti. V závěru práce prezentuji a diskutuji dosažené výsledky.

Abstract

This diploma thesis defines the term indexing and in preamble are discussed known indexing algorithms and difference between indexing static and moving objects. The practical part of this diploma thesis is aimed to designing and implementing of indexing algorithm for open source application MaNGOS with respect to generic design pattern and effectiveness of spatial search queries for selection of the objects given properties in the specified area. At the end I present and discuss reached results.

Klíčová slova

Prostorové indexování, 3D, R-Tree, KD-Tree, PR-Tree, Grid, Quad-Tree, TPR-Tree, Hilbert-Tree, C++, meta-programování, šablony

Keywords

Spatial indexing, 3D, R-Tree, KD-Tree, PR-Tree, Grid, , Quad-Tree, TPR-Tree, Hilbert-Tree, C++, metaprogramming, templates

Citace

Miroslav Drbal: Indexování objektů v 3D prostoru, diplomová práce, Brno, FIT VUT v Brně, 2010

Indexování objektů v 3D prostoru

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Filipa Orsága, Ph. D.

....
Miroslav Drbal
26. května 2010

Poděkování

Na tomto místě bych rád poděkoval svým rodičům, kteří mi poskytli kvalitní zázemí ke studiu, tak i své přítelkyni, která mi byla trpělivou oporou při psaní této práce.

© Miroslav Drbal, 2010.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **4**
1.1 Organizace práce a obsah jednotlivých kapitol 6

2 Používané indexovací algoritmy pro nepohybující se objekty **7**
2.1 Grid index 7
2.2 Quad-Tree index 8
2.3 kD-Tree index 9
2.3.1 Adaptivní kD-Tree 11
2.4 R-Tree index 11
2.4.1 Varianty R-Tree indexu pro dynamicky se měnící data 12
2.4.2 Varianty R-Tree indexu pro statická data 14
2.4.3 STR Packed R-Tree 15

3 Používané indexovací algoritmy pro pohybující se objekty **16**
3.1 Q+R-Tree 16
3.2 TPR-Tree 17

4 Volba indexu pro implementaci **19**
4.1 Struktura map a stávající indexovací systém v aplikaci MaNGOS 19
4.2 Analýza herních dat a jejich distribuce v prostoru 20

5 Implementace indexu **23**
5.1 Popis navržené indexovací struktury 23
5.2 Odbočka k šablonovému meta-programování 24
5.3 Operace nad navrženou indexovací strukturou 28
5.3.1 Algoritmus dělení listu v R*-Tree 28
5.3.2 Algoritmus vynuceného znovuvložení objektů v R*-Tree 29
5.3.3 Algoritmus ošetření přetečení v R*-Tree 30
5.3.4 Algoritmus vložení objektu do **R***-Tree 30
5.3.5 Algoritmus vložení objektu do Quad-Tree 31
5.3.6 Operace vložení nebo vymazání objektu z indexovací struktury 31
5.4 Popis implementovaných datových struktur 33
5.4.1 Struktura Point3D 33
5.4.2 Třída BoundingBox3D 34
5.4.3 Třída BasicVisitor 35
5.4.4 Třída BaseNode 36
5.4.5 Třída RTreeNode 36
5.4.6 Třída RTreeLeafNode 37

5.4.7 Třída RTree 38
5.4.8 Třída QTreeNode 40

6 Testování 41
6.1 Testování funkcionality jednotlivých tříd 41
6.2 Systém automatického testování výkonnosti 42
6.2.1 Struktura BenchArgs 43
6.3 Metodika testování výkonnosti 43
6.4 Syntetické testování výkonnosti 44
6.4.1 Výkon operace vkládání dat 45
6.4.2 Výkon operace vyhledávání v datech 46
6.4.3 Výkon operace aktualizace dat 49
6.5 Testování výkonnosti na reálných datech 51

7 Závěr 54

A Obsah DVD 57

B Tabulky výčtových typů pro strukturu BenchArg 58

C Ukázky delších fragmentů kódu 59

Cíl práce

Cílem diplomové práce je uvést čtenáře do problematiky indexování prostorových dat. Poukázat na rozdíly mezi indexováním statických a pohybujících se objektů. Podrobněji představit nejznámější indexovací struktury a algoritmy pro indexování prostorových dat a to jak statických tak pohybujících se.

Praktická část diplomové práce si klade za cíl analyzovat data v open source aplikaci MaNGOS a na základě této analýzy provést implementaci indexovací metody vhodné pro daný typ prostorových dat. Návrh a implementace je zaměřena na generičnost návrhu a efektivitu vyhledávacích dotazů.

Kapitola 1

Úvod

S rostoucím výkonem počítačů a jejich rozšiřováním do rozličných oborů lidské činnosti se stále větší oblibě těší nejrůznější GIS[1] a multimediální systémy. Jako zástupce GIS systémů, se kterými se většina lidí setká i při běžné práci na počítači, můžeme jmenovat například portály http://mapy.cz nebo aplikaci Google Earth[2]. Další dynamicky se rozvíjející oblastí, ve které se pracuje s velkým množstvím prostorových dat, jsou MMOG[3] hry, zvláště jejich početná podmnožina MMORPG[4] hry.

Na tomto místě by mohla vyvstat otázka, co mají tyto systémy společného s indexováním v 3D prostoru? V první řadě si je potřeba uvědomit, že za podstatnou částí těchto systémů stojí nějaký databázový stroj, ve kterém jsou uložena všechna zobrazovaná data. Nad touto databází můžeme provádět různá vyhledávání, jejichž parametrem jsou například údaje vymezující nějaký prostor a nebo vzdálenost od známého objektu. Jako příklad můžeme použít dotaz: *vyhledej všechny bankomaty vzdálené N metrů od hotelu H.* V případě online her se jedná převážně o nalezení sousedních NPC[5], kterým se následně zasílají události generované na základě interakci mezi jinými NPC nebo hráči.

Kdybychom k vyřešení těchto dotazů použili naivní způsob a procházeli celou kolekci objektů a každý zvlášť testovali, zda-li splňuje zadané vlastnosti, dostali bychom se ve velkých kolekcích objektů k problému, že zpracování dotazu trvá neúměrně dlouhou dobu. Na tomto místě nastupují indexovací algoritmy, které tím, že organizují kolekci objektů do různých efektivnějších struktur, než je pouhý seznam, dokáží časovou náročnost vyhledávacích dotazů podstatně zkrátit. Motivace pro výzkum v oblasti 3D indexovacích algoritmů je tedy více než zřejmá.

Formálnější definice indexování by mohla znít následovně: *indexování je organizování kolekce dat podle specifikovaných klíčových atributů do takových struktur, které nám umožňují provádět některé operace(vyhledávání), závisející na těchto klíčových atributech, rychleji.*

Indexovat se však dají nejenom body v prostoru, ale i celé geometrické útvary jako *úsečka*, *čtverec*, *N-úhelník* a další. V systémech, které nám dovolují zadávat složitější geometrické objekty jako jednotlivé entity, pak můžeme definovat daleko složitější dotazy jako jsou například: *najdi všechny bytové jednotky J v oblasti A pod kterými vede potrubí P.* Výsledkem takového dotazu jsou takové bytové jednotky, které se nacházejí v průniku oblastí A a P. V této práci se ovšem touto tematikou příliš zabývat nebudu a práce bude

[1] Vysvětlení pojmu GIS lze nalézt na http://cs.wikipedia.org/wiki/Geografický_informační_systém

[2] Aplikace je dostupná na URL: http://earth.google.com/intl/cs/

[3] Massive(ly)-Multiplayer Online Game

[4] Massive(ly)-Multiplayer Online Role-Playing Game

[5] Jedná se o postavy ve hře ovládané umělou inteligencí, zkratka znamená non-player character

soustředěna především na indexování objektů v prostoru. Pod pojmem *objekt* zde budu uvažovat takovou entitu, která je popsána konkrétním vektorem svých polohových souřadnic x, y, z a případně disponuje informací o své velikosti definované jako čtyřúhelník v n-rozměrném prostoru (obdélník, kvádr, ...), který do sebe pojímá celý objekt a který budu dále označovat jako *bounding box*.

1.1 Organizace práce a obsah jednotlivých kapitol

Práci jsem rozdělil do několika tematických kapitol. V kapitole 2 se zabývám již existujícími implementacemi indexovacích algoritmů pro statické objekty, které se dají použít pro indexování v 3D prostoru. Následuje kapitola 3, kde se věnuji problematice indexování pohybujících se objektů v 3D prostoru. Kapitola 4 se zabývá volbou algoritmu pro náhradu Grid Indexu[6] v open source aplikaci MaNGOS. Vlastní implementací se zabývá kapitola 5. Metodiku testování a rozbor naměřených hodnot prezentuji v kapitole 6. V poslední kapitole 7 shrnuji celkové dosažené výsledky a nastiňuji případný další směr vývoje.

[6] podrobnější popis této indexovací struktury je v kapitole 2.1

Kapitola 2

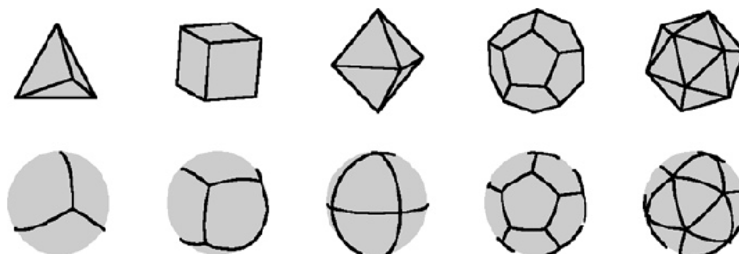
Používané indexovací algoritmy pro nepohybující se objekty

Tato kapitola se zabývá již existujícími indexovacími algoritmy.

2.1 Grid index

Grid index je na implementaci nejjednodušší možný indexovací algoritmus. Spočívá v tom, že se prostor rozdělí na více stejných částí nejčastěji čtvercového nebo obdélníkového průřezu. Indexování pomocí gridu můžeme úspěšně použít v geografických systémech, kde požadujeme vyhledávání podle GPS souřadnic, protože polohu udanou pomocí zeměpisné délky a šířky lze snadno přepočítat na index buňky v gridu. Nevýhodou gridu při indexování povrchu planety Země může být jisté zkreslení, které je způsobeno transformací ze sférických souřadnic do kartézského souřadného systému.

Pro účely mapování povrchů různých těles se využívá jiných tvarů buněk gridu. Touto problematikou se blíže zabývá [7]. Na Obrázku 2.1 je uvedena ukázka různých tvarů buněk gridu použité v závislosti na tvaru povrchu indexovaného tělesa.



Obrázek 2.1: *Různé tvary buněk gridu, přejato z [7]*

Na první pohled by se Grid index mohl jevit jako velice jednoduše implementovatelný a efektivní algoritmus. Efektivní vzhledem k tomu, že ze zadaných souřadnic lze velice snadno vypočítat cílovou buňku gridu. Problémy však mohou nastat v případě, že bychom chtěli vyhledat objekty daných parametrů ve specifikovaném okolí od zadaného bodu. V tomto případě musíme vzít do úvahy, že pokud zvolíme velikost buňky příliš velkou, musíme projít celou kolekci objektů uložených v buňce, ověřit zda se objekt nachází ve vymezené oblasti

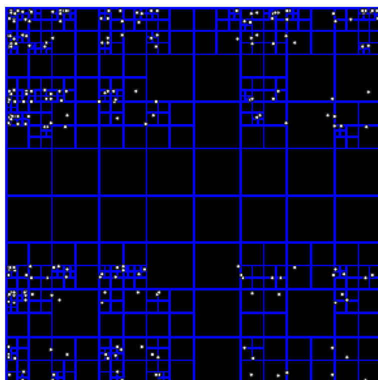
a poté ověřit zda nalezený objekt splňuje dané vlastnosti[1]. Zvolíme-li velikost buňky příliš malou, budeme muset pro malé vyhledávací poloměry navštívit všechny buňky, které leží na ploše kruhu vymezující vyhledávací oblast. Z výše popsaných úskalí algoritmu pak vyplývá, že i přes jeho značnou jednoduchost a rychlé nalezení požadované buňky, mohou být některé dotazy ve spojení se špatným návrhem velikosti buňky výpočetně náročné. Tyto problémy se snaží řešit další uvedené algoritmy založené na stromové struktuře.

2.2 Quad-Tree index

Název Quad-Tree vychází z principu dělení dvoudimenzionálního prostoru podle os x a y, kdy takto vzniknou čtyři kvadranty. Quad-Tree nachází své uplatnění nejčastěji pro indexaci dvoudimenzionálních dat, ale s úspěchem ho leze používat i pro vícerozměrné prostory.

Algoritmus se snaží odstranit nedostatky uvedené u Grid indexu v podkapitole 2.1 rekurzivním dělením plochy (prostoru) podle os tak, aby počet objektů v těchto kvadrantech nepřekročil stanovenou mez. Tímto postupem se algoritmus snaží eliminovat příliš plné buňky a zachovat tak jejich rozumnou velikost.

Vlastní data se nacházejí v listech stromu. Uzly definují plochu (prostor), ve které se nacházejí všechna data ve zbylém podstromu, který přísluší danému uzlu.



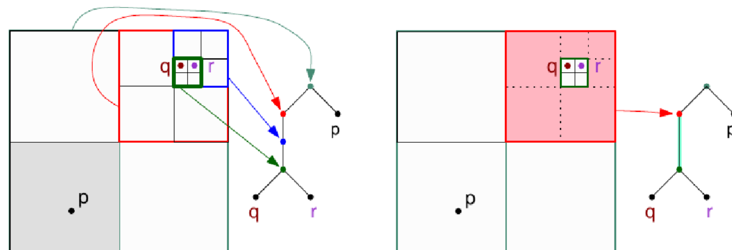
Obrázek 2.2: *Ukázka vybudovaného indexu nad body plochy, přejato z [12]*

Vyhledávání v tomto indexu je realizováno jako průchod stromem od kořene směrem k listům[2]. V každém uzlu se vyhodnotí zda-li plocha, kterou představuje tento uzel, má neprázdný průnik s oblastí, ve které provádíme vyhledávání. Pokud ano, postupujeme rekurzivně k hlubším uzlům. Listy takto nalezeného posledního uzlu představují vyhledaná data.

V závislosti na rozložení objektů na ploše (prostoru) mohou při budování indexu vzniknout uzly, které neobsahují žádná data. Abychom ušetřili místo, nekonstruujeme tyto uzly, ale nahrazujeme je null-ovým ukazatelem v mateřském uzlu. Další optimalizací je vynechání těch uzlů na cestě, které neobsahují odkazy na žádné další uzly. Budování stromu je znázorněno na Obrázku 2.3. Vzniklý strom se nazývá komprimovaný Quad-Tree.

[1]Pořadí vyhodnocování vzdálenosti a vlastností objektu se může měnit v závislosti na tom, která z operací je výpočetně náročnější

[2]Nejčastěji se tento průchod realizuje jako pre-order



Obrázek 2.3: Demonstrace komprimace Quad-Tree indexu, přejato z [9]

Velikost vybudovaného komprimovaného stromu se pohybuje v třídě $O(n)$. Hloubka komprimovaného stromu je ohraničena intervalem $\langle\Omega(\log(n)), O(n)\rangle$. Operace vyhledání, vložení a vymazání se pohybují ve složitostní třídě $O(h)$.

Podrobnější informace o datové struktuře Quad-Tree můžete nalézt v [4, 9].

2.3 kD-Tree index

kD-Tree, kde k označuje stupeň dimenze, ve které budeme provádět indexování, je ve své podstatě binární strom. kD-Tree nachází své uplatnění v mnoha aplikacích, ve kterých je požadováno multidimenzionální vyhledáváni (vyhledávání na základě zadaného poloměru nebo nalezení nejbližšího souseda).

kD-Tree je binární strom, kde každý uzel reprezentuje bod v k-rozměrném prostoru. Uzly stromu jsou generovány tak, že celý indexovaný prostor je rekurzivně dělen hyperplochami (takovými že obsahují alespoň jeden nebo více bodů, které nenáleží do jiné hyperplochy) na dva menší podprostory. Vzhledem k tomu, že není nikterak blíže specifikováno jakým způsobem se volí osa použitá pro dělení prostoru, otevírají se zde možnosti pro výzkum různých sofistikovaných algoritmů a heuristik k určení následujícího nejoptimálnějšího dělení podprostoru.

Nejpoužívanější metoda konstrukce kD-Tree striktně dodržuje následující dvě pravidla:

1. Volba dělících os probíhá cyklicky: Pokud byl kořenový uzel rozdělen podle osy x oba jeho následníci budou rozděleni podle osy y, následníci těchto následníků podle osy z, dále podle x atd.[3]

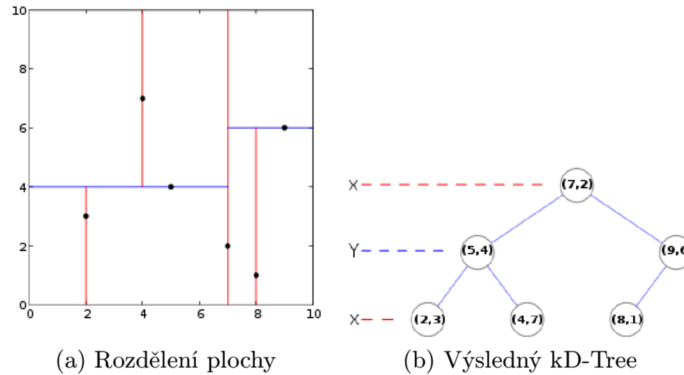
2. Bod, kterým bude hyperplocha procházet, je zvolen jako medián bodů v daném podprostoru (podploše) seřazených podle dílčí souřadnice v závislosti na zvolené ose pro tvorbu hyperplochy[4].

Tato metoda vede ke konstrukci vyváženého kD-Tree, kde každý list má přibližně stejnou vzdálenost od kořenového uzlu. Nicméně vyvážené kD-Tree nejsou vždy pro všechny aplikace nejvhodnější.

[3]Nastíněné řešení byla ukázka pro 3-rozměrný prostor.

[4]Například bude-li hyperplocha kolmá na osu x, medián se bude určovat z *x-ových* souřadnic bodů v daném podprostoru (podploše)

Obrázek 2.4 prezentuje rozdělení dvoudimenzionální plochy a výsledný kD-Tree.



(a) Rozdělení plochy (b) Výsledný kD-Tree

Obrázek 2.4: Demonstrace rozdělení 2D plochy a výsledný kD-Tree [11]

Pojďme se nyní podívat na časové složitosti kD-Tree indexu. Z předchozího textu může být již předem patrné, že při konstrukci kD-Tree indexu bude hrát hlavní roli algoritmus pro výpočet mediánu, od kterého se odvíjí celková časová složitost vybudování indexu z n bodů.

Při použití algoritmu pro nalezení mediánu, který pracuje v časové složitosti $O(n)$ bude kD-Tree vybudován v časové složitosti $O(n \log(n))$ [5]. Pokud bychom použili pro nalezení mediánu algoritmus pracující v časové složitosti $O(n \log(n))$, složitost vybudování kD-Tree bude ležet v $O(n \log^2(n))$ [11]. Vzhledem k tomu, že každá hyperplocha obsahuje právě jeden a nebo více bodů velikost stromu (rozuměj počet uzlů) roste lineárně s počtem indexovaných bodů a leží tedy v $O(n/k)$, kde k je konstanta určující počet bodů obsažených v hyperploše. Dotaz, kterým hledáme všechny body obsažené v prostoru vymezeném kvádrem, jehož stěny jsou rovnoběžné s osami indexovaného prostoru (nebo čtyřúhelníkem, jehož strany jsou rovnoběžné s osami indexované plochy) je vyhodnocen s časovou složitostí $O(\sqrt{n} + k)$, kde k je počet nalezených bodů [5].

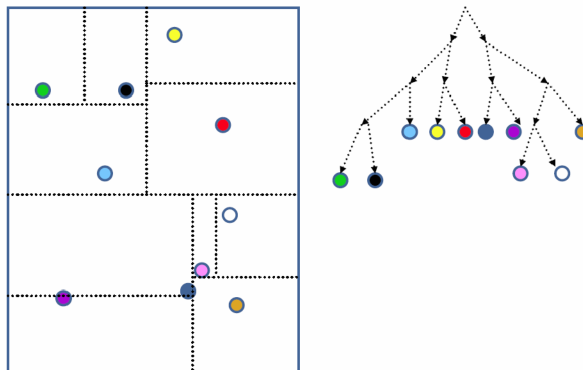
Jedním z problémů u kD-Tree je odstranění bodu z indexu.Odstraňujeme-li prvek, který se nachází v listu stromu, nenastávají žádné komplikace, protože list může být bez následků odstraněn. O něco komplikovanější situace nastává, je-li potřeba odstranit bod, který se nachází v nelistovém uzlu. Tímto bodem je totiž vedena hyperplocha,která rozděluje v tomto bodě strom na další dva podstromy. V zásadě jsou dvě možnosti, jak se s tímto problémem vypořádat [2]:

1. Bod odstraníme včetně hyperplochy a provedeme rekonstrukci obou podstromů. Tento postup však může být velice drahý, co se výpočetního času týká.

2. Bod ponecháme i s hyperplochou ve stromu, ale nastavíme příznak o jeho vymazání. Tímto zabezpečíme, že prvek se nebude aktivně účastnit ve vyhledávání a zůstane zachována konzistence stromu. Takto označené prvky z indexu vypustíme při příštím znovusestavení stromu.

2.3.1 Adaptivní kD-Tree

Jedná se o variantu kD-Tree, která se od původního algoritmu odlišuje tím, že hyperplocha, která dělí prostor rovnoběžně s osami, tak aby v obou podstromech byl pokud možno stejný počet bodů, neprochází žádným bodem. Tímto způsobem se eliminují body obsažené v hyperplochách a veškeré reference na objekty jsou přesunuty až do listových uzlů. Díky tomuto je eliminován problém popsaný výše v kapitole 2.3.



Obrázek 2.5: Ukázka Adaptive kD-Tree

2.4 R-Tree index

R-Tree index, je hierarchická datová struktura podobná B^+-Tree[5]. Tento algoritmus se používá pro dynamické indexování objektů v k-dimenzionálním prostoru, přičemž každý objekt je reprezentován svým k-dimenzionálním čtyřúhelníkovým bounding boxem. Každý uzel v R-Tree indexu odpovídá nejmenšímu bounding boxu, který obklopuje všechny potomky daného uzlu. Na tomto místě bych rád poznamenal, že minimální bounding boxy příslušející různým uzlům se mohou překrývat. Objekty jsou uloženy v listech stromu. Každý list stromu může pojmout pouze předem definovaný omezený počet objektů. Pokud při operaci vložení dojde k „přetečení“ definovaného limitu, listový uzel se rozdělí. Pro dělení listových uzlů existují tři základní algoritmy [3]:

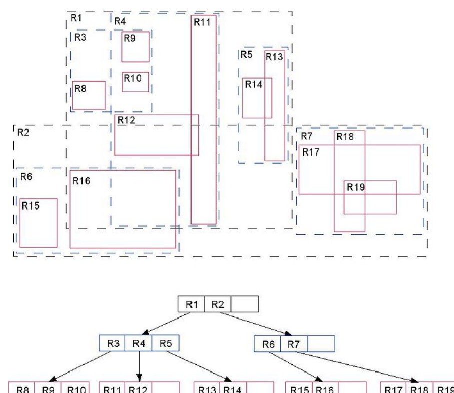
- **Linear Split**: Při rozdělovaném uzlu se vyberou dva nejvzdálenější body, které slouží jako počáteční objekty v nových listových uzlech. Zbytek bodů v původním listu je náhodně procházen a každý bod je přiřazen do takového nového listu, ve kterém po jeho přidání vznikne menší minimální bounding box.

- **Quadratic Split**: Při rozdělování uzlu se vyberou dva body, které slouží jako počáteční objekty v nových uzlech, aby jejich spojením vzniklo co nejvíce prázdného místa. Prázným místem je zde myšlen prostor, který zůstane z minimálního bounding boxu těchto dvou bodů po odstranění vlastních bounding boxů bodů. Dále vybíráme ze zbylých bodů postupně takové body pro vložení do nových uzlů, aby po jejich vložení bylo prázdné místo v uzlu, do kterého vkládáme, co největší (maximalizací volného místa v uzlu provádíme minimalizaci výsledného bounding boxu). bounding box.

[5]Informace o struktuře B^+-Tree lze nalézt na http://en.wikipedia.org/wiki/B%2B_tree

- **Exponential Split**: Tato varianta je z uvedených dvou nejvíce výpočetně náročná. Spočívá ve vygenerování všech možných kombinací rozdělení uzlu a následném zvolení takové varianty, která tvoří nejmenší možný bounding box.

Na obrázku 2.6 je znázorněno rozdělení plochy na minimální bounding boxy a výsledný R-Tree.



Obrázek 2.6: Ukázka R-Tree, převzato z [13]

2.4.1 Varianty R-Tree indexu pro dynamicky se měnící data

V této podkapitole se budu zabývat různými variantami R-Tree indexu optimalizovanými pro práci s dynamicky se měnícími indexovanými objekty. Pod pojmem *měnící se objekt* si lze představit například množinu objektů, které v indexovaném prostoru mění svoji polohu a je tedy nutné index optimalizovat především pro operace vkládání a rušení nových objektů.

R$^+$-Tree

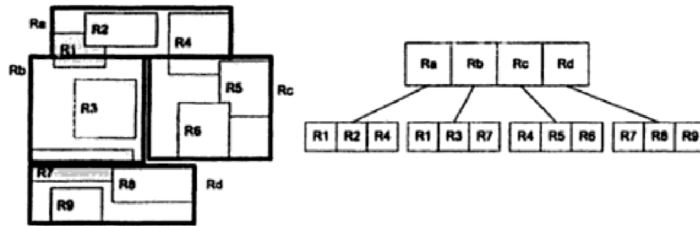
Tato indexovací datová struktura byla navržena tak, aby se zamezilo vícenásobnému navštívení stejných podprostorů při vyhledávání prvku[6]. K zabránění daného jevu se využívá oříznutí jednotlivých bounding boxů na stejné úrovni. R$^+$-Tree tedy má na stejné úrovni naprosto prostorově disjunktní bounding boxy. Tato optimalizace s sebou přináší několik komplikací.

Při vkládání objektů do indexu musí být objekty rozděleny do dvou a nebo více částí, což znamená, že některé specifické objekty mohou být uloženy redundantně do více listových uzlů. Navíc tato negativní vlastnost vede opět k drobnému degradování výkonnosti při vyhledávání objektů. Negativní dopad není však tak markantní, aby plně vyvážil získaný benefit.

Další negativní důsledek ořezávání bounding boxů se projevuje opět při vkládání objektů do indexu. Vlivem vložení objektu může dojít ke zvětšení bounding boxu na cestě k listovému uzlu, což má za následek vyvolání oříznutí. Toto oříznutí může opět vyvolat řetězové ořezávání dalších bounding boxů, což může v některých případech vést až k deadlocku[7]. Na obrázku 2.7 je vidět ukázka R$^+$-Tree, kde je patrná i duplicita některých listových uzlů. [3]

[6] V klasickém R-Tree indexu se mohou jednotlivé bounding boxy překrývat, což znamená, že při vyhledávání se některé části prostoru procházejí vícekrát, důsledek tohoto je snížení výkonu operace vyhledání.

[7] Vysvětlení pojmu *deadlock* lze nalézt na http://cs.wikipedia.org/wiki/Deadlock



Obrázek 2.7: Ukázka R+ Tree, převzato z [3]

R*-Tree

Jedná se o v literatuře nejčastěji uváděnou variantu R-Tree s ohledem na výkon. R*-Tree používá na rozdíl od R$^+$-Tree nebo R-Tree pokročilejší metody pro rozdělení listu. Při rozdělování listu je využita technika tzv. *vynuceného znovu vložení*[8], kdy část objektů, které „přetečou" přes limit buňky[9], je z buňky odebrána a aplikuje se na ně operace vložení, která probíhá znovu od kořene stromu.

Další inovující vlastností R* indexu je to, že bere do úvahy i další kritéria, nejenom minimalizaci výsledného minimálního bounding boxu jak tomu bylo u R$^+$-Tree nebo R-Tree. Mezi tyto další kritéria patří minimalizace překryvu minimálních bounding boxů na stejné úrovni stromu[10] společně se snahou minimalizovat výsledné minimální bounding boxy.[3, 14]

Hilbert R-Tree

Tato indexovací struktura má hybridní charakter. Je založena na R-Tree a B$^+$-Tree. Ve své podstatě je to B$^+$-Tree, který může obsahovat vícerozměrné geometrické objekty (úsečky, plochy, 3D objekty, ...). Výkonnost Hilbertova R-Tree je závislá na kvalitě algoritmu, který rozkládá data do jednotlivých uzlů. Hilbertovy R-Tree využívají *křivky pro vyplnění prostoru*[11], obzvláště pak Hilbertovu křivku, pomocí které zavádí do indexu lineární uspořádání na jednotlivých datových regionech. Toto uspořádání musí být takové, aby seskupovalo podobné datové regiony tak, aby výsledný minimální bounding box byl co nejmenší.

Hilbertův R-Tree řadí datové regiony v závislosti na Hilbertově vzdálenosti[12] středů jednotlivých minimálních bounding boxů. Díky tomuto uspořádání má každý uzel dobře definovanou množinu příbuzných uzlů.

Dynamické Hilbertovy R-Tree jsou vhodné pro aplikace, kde se operace vkládání, vymazání nebo aktualizace vyskytují v reálném čase. Dynamické Hilbertovy R-Tree disponují flexibilním mechanismem na rozdělování listů, který dokáže v případě potřeby rozdělení odložit a přispívá tak k lepšímu využití místa. [10, 6, 3]

[8] Anglicky forced reinsertion

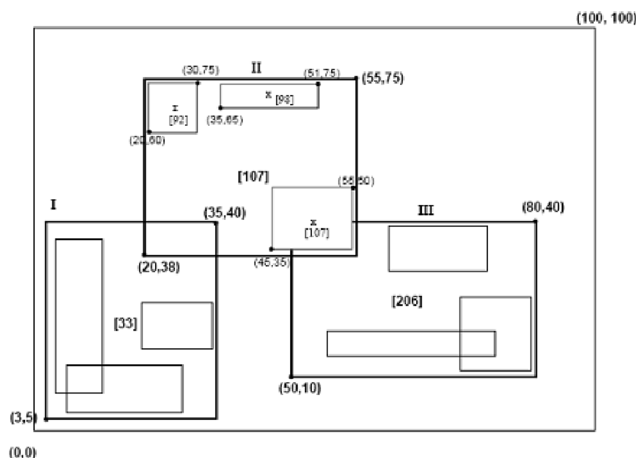
[9] [3] uvádí 30%

[10] R*-Tree se nesnaží překryvy zcela eliminovat ořezem jak tomu bylo u R$^+$-Tree, ale snaží se tyto překryvy pouze minimalizovat.

[11] Anglicky space-filling curves

[12] Hilbertova vzdálenost mezi body A a B je délka Hilbertovy křivky, která tyto body spojuje.

Na obrázku 2.8 je ukázka vybudovaného Hilbertova R-Tree. Čísla v [] psaná tučně udávají Hilbertovu vzdálenost pro střed minimálního bounding boxu, čísla v [] u bodů označených x udávají Hilbertovy vzdálenosti těchto bodů, čísla v () udávají absolutní souřadnice daných bodů.



Obrázek 2.8: *Ukázka Hilbertova R-Tree, převzato z [10]*

Vkládání do indexu probíhá tak, že na každé úrovni Hilbertova R-Tree se testují Hilbertovy vzdálenosti všech uzlů s Hilbertovou vzdáleností vkládaného objektu. Uzel s nejmenší Hilbertovou vzdáleností ze všech testovaných uzlů na dané úrovni stromu, která je však větší než Hilbertova vzdálenost vkládaného objektu je zvolen k dalšímu prozkoumání. Takto se pokračuje, dokud není nalezen list, do kterého se objekt vloží.

Dojde-li díky vložení nového objektu k přetečení listového uzlu, algoritmus se nejprve pokusí data rozdistribuovat do sousedních uzlů. Uzel se tedy dělí pouze jsou-li všechny sousední uzly plné a nelze do nich přesunout žádné objekty. Tato heuristika je podobná heuristice, která se používá u B*-Tree, kde se také provádí redistribuce a 2-to-3 rozdělení uzlů[13] v případě přetečení. Hilbertovy R-Tree vykazují nejlepší výsledky z celé třídy dynamických R-Tree indexovacích algoritmů i když poněkud selhává na objekty, které mají velké vlastní bounding boxy.[3]

2.4.2 Varianty R-Tree indexu pro statická data

V této podkapitole se budu zabývat různými variantami R-Tree indexu optimalizovanými pro práci se statickými daty. Tato kategorie indexovacích algoritmů má své majoritní zastoupení například v kartografii, kde se v podstatě nesetkáváme s operacemi typu vkládání nebo rušení. Naopak je zde kladen velký důraz na operaci vyhledání.

Packed R-Tree

Algoritmus se od standardního R-Tree algoritmu liší v počáteční konstrukci stromu. Objekty jsou seřazeny na základě nějakého prostorového atributu (třeba podle *x-ové*) souřadnice a seskupeny do listových uzlů. [3]

[13] 2-to-3 rozdělení znamená, že ze dvou uzlů jsou vytvořeny 3.

Packed Hilbert R-Tree

Opět se jedná o obdobu již dříve zmíněného algoritmu. Objekty jsou na začátku seřazeny podle svojí Hilbertovy vzdálenosti a seskupeny do listových uzlů. Tento postup se vyznačuje vysokým procentem využitého místa (kolem 100%). [3]

2.4.3 STR Packed R-Tree

Zkratka STR znamená *Sort-Tile-Recursive*. Jedná se o algoritmus pro vytvoření Packed R-Tree z množiny známých objektů pomocí S řezů prostorem (plochou) tak, aby každá část rozděleného prostoru obsahovala dostatečné množství objektů aby vzniklo přibližně $\sqrt{N/C}$ uzlů, kde C je maximální kapacita jednoho uzlu v R-Tree.

Na počátku si určíme počet listových uzlů ze vztahu $L = [\sqrt{N/C}]$. Nechť $S = \sqrt{L}$. Dále všechny objekty seřadíme v závislosti na jejich *x-ové* souřadnici a vytvoříme S řezů. Každý řez pak obsahuje $S \cdot C$ objektů, které v jednotlivých řezech seřadíme podle jejich *y-ové* souřadnice. Tyto objekty jsou zabaleny do uzlů výsledného R-Tree. Tuto metodu opakujeme dokud nejsou zformovány všechny vrstvy stromu.

Tato metoda vykazuje lepší výsledky než Packed R-Tree nebo Packed Hilbert R-Tree v podkapitole 2.4.2. Nicméně pro některé vzorky dat se v některých případech ukazuje efektivnější algoritmus Packed Hilbert R-Tree. [3]

Kapitola 3

Používané indexovací algoritmy pro pohybující se objekty

Tato kapitola se zabývá algoritmy a strukturami použitými pro indexaci pohybujících se objektů v 3D prostoru.

Proč indexovat pohybující se objekty? Rozvoj a postupné rozšiřování telekomunikačních a výpočetních prostředků do všemožných odvětví lidské činnosti se naskýtá možnost pomocí GPS, mobilních telefonů a dalších technologií monitorovat v reálném čase například dopravu (trolejbusy, autobusy, vlaky, tramvaje, taxíky a další). V takovýchto rozsáhlých systémech je potřeba každou sekundu nejen efektivně zpracovat netriviální množství vstupních údajů, ale také dokázat v těchto datech efektivně vyhledávat na základě zadaných parametrů.

Otázkou jaké algoritmy nebo indexovací struktury zvolit a jaké jsou jejich výhody nebo nevýhody, se zabývají následující podkapitoly.

3.1 Q+R-Tree

Q+R-Tree, je symbiózou Quad-Tree a R-Tree indexu. R-Tree poskytuje dostatečně rychlé odpovědi na prostorové dotazy, ale co se týká aktualizací dat v indexu, nevede si zas až tak dobře. Quad-Tree poskytuje velice dobrý výkon při aktualizaci indexu, ale v dotazování je pomalejší než R-Tree. Q+R-Tree se snaží eliminovat zmíněné nedostatky kombinací jejich výhod.

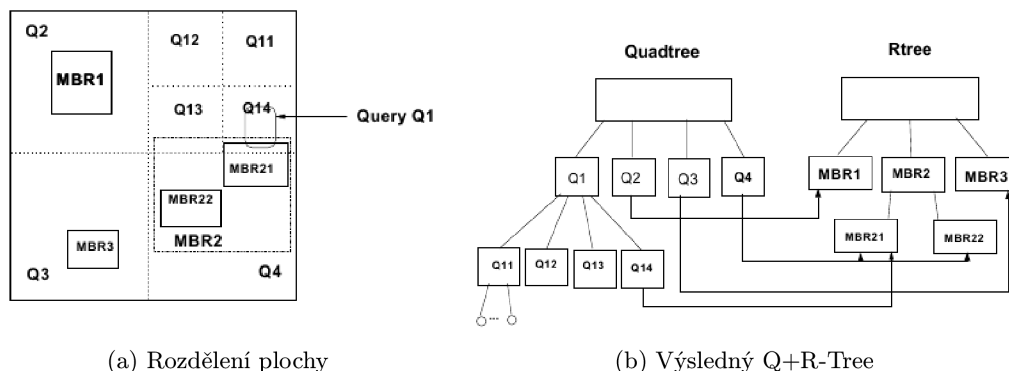
Použití kombinace Quad-Tree + R-Tree je založeno na pozorování, že v reálném prostředí můžeme pohyblivé objekty rozdělit na dvě kategorie:

- **Kvazi-statické**: Objekty, které se nepohybují vysokou rychlostí a převážně se vyskytují ve stále stejném omezeném prostoru. Těchto objektů je obecně majorita.
- **Rychle se pohybující**: Objekty, které se pohybují velkou rychlostí a jejich pohyb není nijak limitován.

Kvazi-statické objekty jsou indexovány pomocí R-Tree s využitím tzv. *líné aktualizace*[1], která spočívá v tom, že index je aktualizován pouze tehdy, pokud objekt svým pohybem opustí minimální bounding box mateřského uzlu. Rychle se pohybující objekty jsou indexovány pomocí Quad-Tree, u kterého je zvoleno hrubé dělení prostoru. Předpokládá se, že rychle se pohybujících objektů je minorita a není tedy třeba prostor dělit nikterak jemně.

[1] Anglicky lazy update

Navíc se i částečně zamezí migraci těchto bodů do jiných uzlů Quad-Tree indexu. Na obrázku 3.1 je vidět konstrukce Q+R-Tree z Quad-Tree a R-Tree.



(a) Rozdělení plochy

(b) Výsledný Q+R-Tree

Obrázek 3.1: *Kompozice R-Tree a Quad-Tree do Q+R-Tree, převzato z [17]*

Propojení Quad-Tree s R-Tree spočívá v tom, že každý listový uzel Quad-Tree obsahuje ukazatel na příslušné minimální bounding boxy v R-Tree, které pokrývá.

Aktualizace indexu probíhá pro kvazi-statické objekty následujícím způsobem. Pokud se objekt přemístil, ale stále se nachází ve stejném minimálním bounding boxu R-Tree, aktualizují se pouze souřadnice uzlu v daném listu. Pohnul-li se objekt mimo minimální bounding box a jeho novým souřadnicím neodpovídá žádný další minimální bounding box v R-Tree, objekt je od tohoto okamžiku považován za rychle se pohybující a je přemístěn do Quad-Tree. Operace aktualizace pro rychle se pohybující objekty probíhá v zásadě stejně.

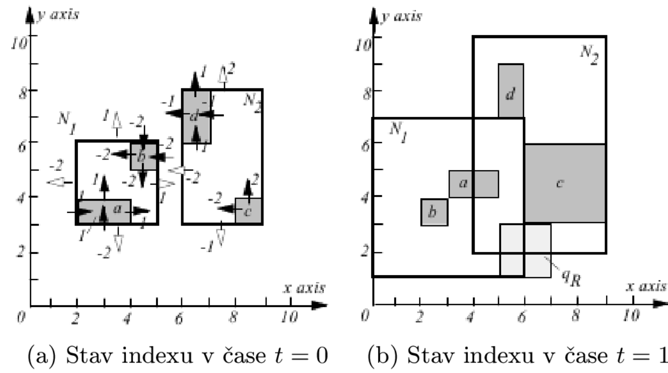
Vyhledávání v Q+R-Tree začíná v Quad-Tree části, kde se vybere odpovídající listový uzel a určí se, které rychle se pohybující objekty splňují kritéria pro vyhledání. Dále se pomocí ukazatelů na minimální bounding boxy v R-Tree určí, zda-li existuje minimální bounding box, který má neprázdný průnik s prohledávanou oblastí. Pokud ano, projdou se jeho listové uzly a pro všechny objekty obsažené v těchto uzlech se ověří, zda některý z nich vyhovuje vyhledávacím kritériím. [17]

3.2 TPR-Tree

Tento indexovací algoritmus je založen na R-Tree. Písmena TP v názvu znamenají *Time Prioritized*. Hlavní myšlenka algoritmu spočívá v tom, že strom v sobě uchovává informace o objektech a minimálních bounding boxech v čase t a dotazy z „budoucnosti“ se snaží predikovat na základě informací o výchozí poloze, směru a rychlosti pohybu bodů. Díky tomu, že algoritmus se snaží predikovat polohu objektů v budoucnosti, výrazně se tak redukuje nutnost aktualizací indexu při přesunu jednoho bodu z místa A do místa B.

Algoritmu je optimalizovaný především pro operace vyhledávání objektů ve specifikované oblasti.

Každý pohybující se objekt je reprezentován v TPR-Tree svým minimálním bounding boxem a rychlostním bounding boxem v čase $t = 0$. Na obrázku 3.2 je znázorněn stav indexu a jeho minimální / rychlostní bounding boxy pro časy $t = 0$ a $t = 1$.



(a) Stav indexu v čase $t = 0$ (b) Stav indexu v čase $t = 1$

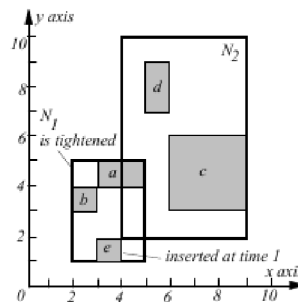
Obrázek 3.2: *TPR*-Tree, převzato z [16]*

Šipky u jednotlivých obdélníků znázorňují vektory rychlostí, čísla znamenají velikost vektoru, přičemž záporné hodnoty znamenají pohyb proti směru osy (obrázek 3.2a). Na obrázku 3.2b je zobrazena situace v čase $t = 1$. Všimněte si, jak se každá hrana minimálního bounding boxu posunula vzhledem ke svojí rychlosti. Minimální bounding box každého nelistového uzlu stále obsahuje svoje objekty, ale nemusí je již těsně obepínat, jak tomu bylo v čase $t = 0$.

Dotazy probíhají stejně jako v klasickém R-Tree. Vyhodnocování začíná v kořenovém uzlu a postupně se prochází stromem. Vyhodnocují se průniky s vypočtenými hranicemi minimálních bounding boxů pro konkrétní čas dotazu t_q. Kupříkladu dotaz q_r pro čas $t = 1$ je znázorněn na obrázku 3.2b. Je zřejmé, že se budou prozkoumávat oba uzly N_1 a N_2 i přesto, že by se v čase $t = 0$ vůbec nezkoumaly.

TPR-Tree je optimalizovaný pro dotazy, které se vejdou do časového okna $[T_C, T_C + H]$, kde T_C je čas poslední aktualizace indexu a H je parametr stromu nazývaný *horizont*, který udává jak moc index „vidí“ do budoucnosti.

Při vložení nového objektu do indexu se vybere list stromu jehož střed minimálního bounding boxu je objektu nejblíže. Objekt se přidá do tohoto listu a provede se přepočítání minimálního bounding boxu, tak aby pevně obepínal množinu všech svých objektů zvětšenou o nově přidaný objekt. Obrázek 3.3 demonstruje přepočet minimálního bounding boxu při vložení nového objektu jako listového uzlu pod rodičovský uzel N_1.



Obrázek 3.3: *Přepočet min. bounding boxu v TPR-Tree, převzato z [16]*

Minimální bounding box uzlu N_2 není přepočítán, protože se ho vložení nového objektu nijak netýkalo. [16]

Kapitola 4

Volba indexu pro implementaci

V této kapitole se zabývám volbou vhodné indexovací struktury pro indexování objektů v open source aplikaci MaNGOS[1]. Na tomto místě bych rád úvodem MaNGOS představil a seznámil tak čtenáře s činností aplikace a její stávající implementací indexování objektů.

MaNGOS je open source aplikace[2] vyvíjená pod licencí GNU/GPLv2 jako výukový projekt, který si klade za cíl prohloubit znalosti jazyka C++, ve kterém je implementována, zdokonalit znalosti v oblasti používání nástrojů pro zprávu zdrojových kódů při vývoji rozsáhlejších projektů v týmu a naučit vzájemné spolupráci při tvorbě rozsáhlejších aplikací.

Vlastní aplikace implementuje serverovou část, takzvaný emulátor[3], pro populární hru World of Warcraft^{TM} . Aplikace je implementována, jak již bylo možná patrné z předchozího odstavce, v jazyce C++, herní data jsou persistentně uložena v databázi. V současné době je implementována podpora pro dva hlavní open source databázové servery, kterými jsou MySQL a PostgreSQL. Vzhledem k tomu, že hra World of Warcraft^{TM} je typu MMORPG[4], která je koncipovaná pro velké množství hráčů hrajících zároveň a pohybujícími se ve stejném světě, je zde nemalý důraz kladen i na bezpečnost a bezpečnou autentizaci hráče serveru. Ta probíhá pomocí protokolu SRP6. Veškerá další komunikace probíhá kryptovaně a data mohou být před odesláním komprimovaná proudovou gzip kompresí, aby se ušetřila kapacita linky.

4.1 Struktura map a stávající indexovací systém v aplikaci MaNGOS

Svět World of Wacraftu^{TM} je rozdělen geograficky na několik kontinentů (Azeroth, Kalimdor, Outland, Northrend). Každý z těchto kontinentů disponuje svojí mapou s unikátním číselným identifikátorem[5]. Velikost každé mapy je limitována rozměrem přibližně $34133,33 \times 34133,33$ s tím, že souřadnice $[0;0]$ se nachází ve středu této čtvercové oblasti. Jakýkoliv herní objekt se může nacházet v této vymezené ploše s transformovanými souřadnicemi v intervalu $(-17066,665; 17066,665)$ s přesností, kterou udává datový typ *float*.

[1] Zkratka MaNGOS vznikla ze slov *Massive Network Game Object Server*

[2] Domovská stránka projektu je dostupná na adrese URL: http://getmangos.com

[3] Emulátor proto, že se aplikace snaží emulovat chování oficiálního serveru, který provozuje firma Blizzard©

[4] Pojem MMORPG je vysvětlen kapitole 1

[5] Ve skutečnosti zde existuje daleko více map než uvedené čtyři, které náleží různým battlegroundům, dungeonům, atd. ty však pro další analýzu nemají přílišný přínos, protože počet objektů náležící těmto mapám je zanedbatelný v porovnání s výše uvedenými kontinenty.

Vzdálenost $|1.0|$ je ve hře základní délkovou jednotkou označovanou jako *1 yard*.

Indexovací systém v MaNGOSu je postaven na obdobě Grid indexu, který je popsán v kapitole 2.1. Každá herní mapa, obsahuje svoji vlastní instanci Grid indexu. Celá herní plocha je rozdělena pravidelnou čtvercovou sítí na 64×64 polí - *grids*, každé o velikosti přibližně $533,33 \times 533,33$. Každý grid je následně rozdělen na dalších 8×8 buněk - *cells*, které obsahují ukazatele na vlastní herní objekty, každá o přibližné velikosti $66,66 \times 66,66$. Zde může být patrná jistá výhoda tohoto systému, která spočívá v konstantní složitosti $O(k)$ nalezení buňky podle zadaných souřadnic.

Implementace Grid indexu v MaNGOSu je provedena velice efektivním způsobem pomocí šablonových tříd, šablonových funkcí a meta-programováním, kterým jazyk C++ disponuje a o kterém se budu blíže bavit v kapitole 5.2 věnované nastínění právě meta-programování.

Každá buňka gridu funguje jako kontejner, který v sobě dokáže uchovávat instance různých typů objektů, aniž by bylo nutné těmto objektům definovat nějakého společného předka, na kterého by se provádělo přetypování při vložení do kontejneru.

Nemálo podstatnou roli hraje Grid index i při periodických aktualizacích jednotlivých herních objektů. Vzhledem k tomu, že aktuální implementace MaNGOSu využívá pro update všech aktivních map a herních objektů v nich obsažených pouze jedno výpočetní vlákno[6], je skoro nemyslitelné, aby se aktualizovaly veškeré herní objekty (počet herních objektů se pohybuje v řádu stovek tisíc jak bude prezentováno dále), což by spotřebovalo příliš mnoho výpočetních prostředků a způsobilo by velice výrazné zpomalení herního serveru daleko za hranice hratelnosti. O řešení tohoto problému se opět stará Grid index, který provede update jen těch objektů, které se nacházejí v takzvaném *aktivním gridu*.

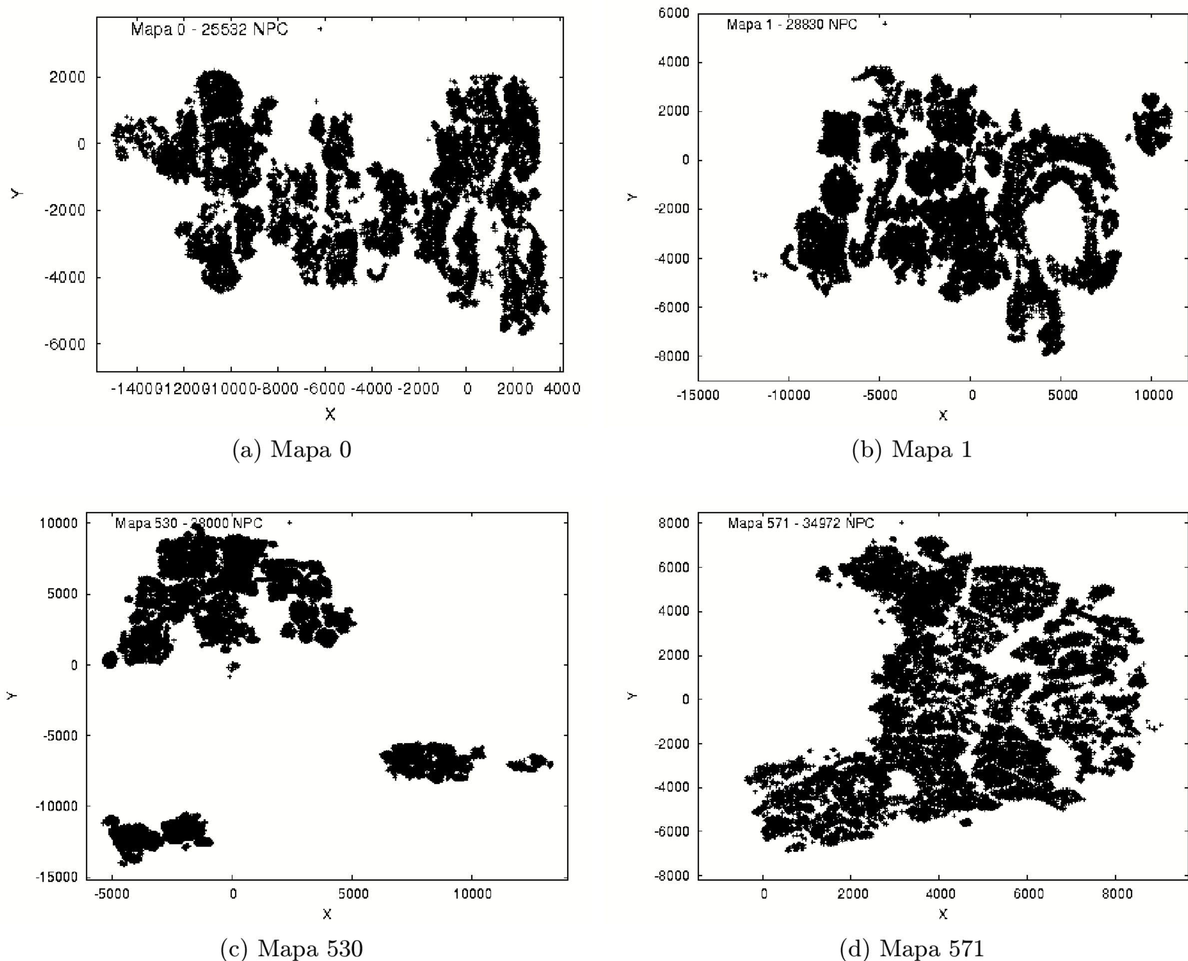
Grid je považován za aktivní, pokud se v něm a nebo v jeho bezprostředně sousedních gridech nachází hráč nebo herní objekt, který má nastavený speciální příznak aktivního objektu.

4.2 Analýza herních dat a jejich distribuce v prostoru

Pro výběr kvalitní indexovací metody bývá nezbytné znát distribuci objektů v indexovaném prostoru. Abychom dostaly lepší představu o tom, jak jsou rozložené objekty v herní databázi MaNGOSu, vytvořil jsem „letecké“ snímky pro mapy 0, 1, 530 a 571[7].

[6]V současné době se vyvíjí jistá iniciativa a reorganizace základních systémových struktur a závislostí tak, aby bylo možné tyto periodické aktualizace map provádět paralelně v několika souběžných vláknech.

[7]Jedná se o mapy kontinentů pojmenované v minulé kapitole, pro jednoduchost uvádím pouze jejich číselné označení



(a) Mapa 0

(b) Mapa 1

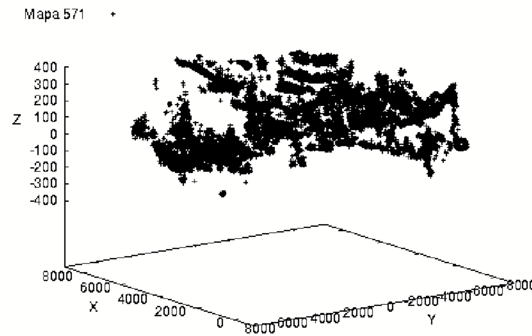
(c) Mapa 530

(d) Mapa 571

Obrázek 4.1: *Ukázka plošného osídlení kontinentů NPC postavami.*

Z obrázku 4.1 je patrné, že žádný kontinent nezabírá kompletně celou plochu mapy, ale ve většině případů pouze její malý výsek. Na obrázcích 4.1a, 4.1b, 4.1c jsou viditelné shluky NPC, což by mohlo být výhodné při použití indexovací struktury založené na stromu s použitím minimálních bounding boxů.

Další otázkou je, jak jsou data rozložena v ose *z*. Pokud by se jednalo o data rozložená rovnoměrně s minimálními výškovými rozdíly, mohla by se *z-ová* souřadnice zanedbat a indexování provádět pouze na úrovni roviny s tím, že výsledný přesný výpočet vzdálenosti podle zadaných kritérií a podle všech tří pozičních složek objektu by se provedl až při procházení koncových listů.



Obrázek 4.2: *Osídlení mapy 571 v 3D prostoru.*

Na tuto otázku nám dává odpověď obrázek 4.2. Podíváme-li se na tento obrázek je patrné, že objekty jsou rovnoměrně distribuované v *z-ové* ose. Může být proto výhodnější stavět indexovací algoritmus rovnou tak, že tvoří minimální bounding boxy pomocí kvádrů v 3D prostoru. Díky tomu vznikne jemnější rozčlenění dat, což může ve výsledku urychlit operace vyhledávání.

Poté co známe charakter rozložení objektů v prostoru zbývá určit, zda-li se jedná převážně o objekty pohybující se nebo statické. U GameObjectů[8] je rozhodování snadné, jedná se jen a pouze o statické objekty. U NPC postav je situace poněkud odlišná. Některé z nich se náhodně pohybují v zadaném poloměru, jiné mají pevně dané cesty po kterých se pohybují a zbytek stojí nehybně na místě do doby, než je vyprovokován k nějaké akci (třeba útok na hráče). V tabulce 4.1 je kumulativně shrnuto do jaké míry se v daném prostředí vyskytují pohybující se objekty.

Tabulka 4.1: Tabulka procentuálního zastoupení pohybujících se a statických objektů v celkovém měřítku.

Typ Objektu	Pohybujících se [%]	Statických [%]	Celkem ve světě
GameObject	0	100	136931
NPC	22	78	139037
Hráč	100	0	2000
Celkem	11,7	88,3	277968

Kritéria pro statický objekt jsou zvolena tak, že objekt se nesmí vůbec hýbat nebo se náhodně pohybuje v poloměru menším než 15 yardů. Počet hráčů byl zvolen jako vysoce nadprůměrná hodnota v rámci českých serverů.

[8]Do této kategorie spadají všechny dveře, truhličky, kytičky, ruda, atd.

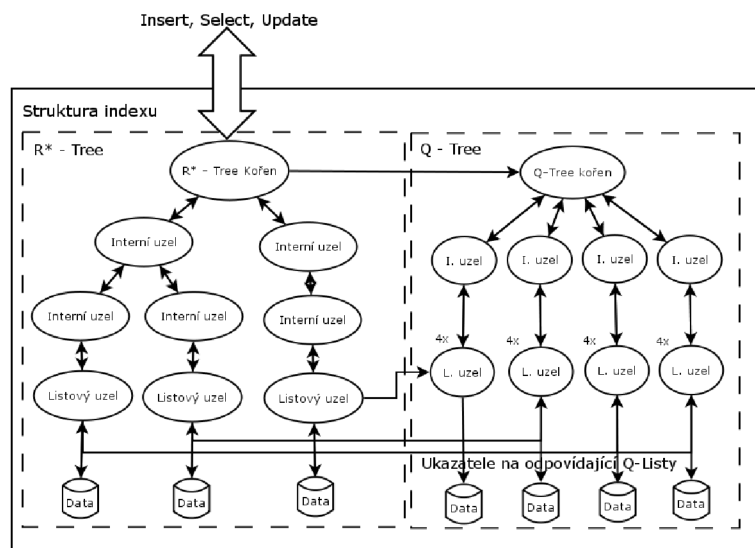
Kapitola 5

Implementace indexu

Jako implementační jazyk jsem si zvolil C++. K této volbě mě vedlo několik důvodů. Hlavním důvodem volby tohoto jazyka byl fakt, že mám s tímto jazykem již mnohaleté zkušenosti a je mi z celé palety jazyků, které ovládám, nejbližší. Dalšími důvody, jsou dobrá přenositelnost kódu mezi platformami (při dodržení některých základních pravidel), výborná rychlost výsledného strojového kódu a široká paleta vývojových nástrojů.

Další důležitou vlastností C++ je podpora pro šablonové meta-programování, které ve veliké míře při implementaci indexu využívám a jemuž je pro uvedení do dané problematiky věnována podkapitola 5.2. V následující kapitole představím strukturovaný návrh indexovací struktury se zběžným popisem funkcionality, kterou následně detailně rozeberu v podkapitole 5.1. V podkapitole 5.3 budu diskutovat možné operace prováděné nad indexovacím hybridním stromem.

5.1 Popis navržené indexovací struktury



Obrázek 5.1: *Skladba navržené indexovací struktury.*

Jak jsem již zmínil v kapitole 4.2 v tabulce 4.1 v herním prostředí hry World of Wacraft^{TM} se vyskytují objekty převážně statického charakteru (88,3%). Z tohoto faktu

vyplývá i vlastní návrh indexovací struktury, kterou jsem navrhl jako hybridní kombinaci R*-Tree pro indexaci statických dat a Q-Tree pro indexaci pohybujících se dat. Návrh reprezentuje obrázek 5.1.

Pozorný čtenář by si mohl na tomto místě povšimnout, že se jedná o strukturu, která je popisovaná v kapitole 3.1, Q+R-Tree. Rozdíl je však v tom, v které části indexu začíná vyhledávání a jakým způsobem jsou vázány listové uzly jednotlivých stromových struktur. V mnou implementované variantě se tedy spíše jedná o R*+Q-Tree. Tento návrh jsem zvolil z několika důvodů. Prvním, důvodem, je že aplikace MaNGOS obsahuje z větší části data statického charakteru a je tedy lepší začít prohledávat právě tato data. Druhým, silnějším, argumentem je fakt, že v aplikaci MaNGOS jsou prostorové dotazy většinou lokálního charakteru. Pokud bychom započali prohledávání v Quad-Tree části, bylo by velice pravděpodobné, že nalezený listový uzel by obsahoval větší množství ukazatelů na R^*-Tree uzly, což by znamenalo projít velké množství objektů, které vůbec nemusí být obsaženy ve specifikovaném prostoru pro hledání. Toto můžeme tvrdit s relativně vysokou mírou jistoty, protože hloubka Quad-Tree je limitovaná a plocha pokrytá jedním listovým uzlem Quad-Tree stromu je podstatně větší než plocha pokrytá jedním listovým uzlem R^*-Tree. Tento fakt je dobré míti na paměti při volbě limitního počtu objektů v listovém uzlu R^*-Tree a limitní hloubky Quad-Tree.

Indexovací struktura je navržena tak, aby operace, které lze provést nad indexovanými daty, nebyly limitovány veřejným rozhraním. Kdybychom pro každou operaci definovali veřejně dostupnou metodu, mohlo by se stát, že bychom skončili s enormním množstvím metod, které by sdílely podobný návrhový vzor, ale každá by prováděla jinou operaci nad indexovanými daty. I přesto by se nám mohlo lehce stát, že zde nebude metoda, která by se hodila našim potřebám. V tom případě bychom si ji museli buď sami implementovat a nebo provést prostorový dotaz, nalezené objekty bychom jako kolekci ukazatelů předali metodě, která by provedla patřičný úkon. Oba přístupy mi přijdou neefektivní. První zbytečně komplikuje veřejné rozhraní velkým počtem úzce specializovaných metod a druhý představuje zvýšení výpočetních nároků ve formě zbytečného kopírování ukazatelů ven ze struktury.

Tento problém jsem vyřešil implementací tzv. návštěvníků, anglicky visitor. Na tomto místě bych rád uvedl, že v dalším textu budu používat pouze termín visitor, protože mi jeho použití přijde vhodnější a lze se s tím to termínem setkat i v jiné odborné literatuře.

Visitor je šablonová struktura, kde pomocí šablonových argumentů specifikujeme datový typ, se kterým má visitor pracovat a který je uložen v listech indexovací struktury, a datový typ určující polohu a vyhledávací poloměr. Důležitou metodou visitoru je metoda *operator()*, ve které je specifikovaná celá činnost visitoru. Tato metoda je zavolána pro každý nalezený objekt vyhovující specifikovaným podmínkám přímo v indexovací struktuře.

5.2 Odbočka k šablonovému meta-programování

V této kapitole bych čtenáři rád objasnil co meta-programování znamená, jaké přináší přínosy pro programátora a uvedl bych zde některé pokročilejší šablonové konstrukce, které jsou poté využity při implementaci indexu. V následujícím textu bude programovací jazyk implicitně znamenat jazyk C++.

Co je to tedy *meta-program*? Jak již je z názvu cítit, bude se jednat o nějaký kus programového kódu, který bude vykonávat nějakou funkci. Jeho funkcí je manipulovat s existujícím programovým kódem a nebo tento kód generovat před jeho samotným překladem. Z této vlastnosti přímo vyplývá jeden vedlejší nepříjemný efekt šablonového meta-programování, kterým je obecně delší doba nutná k překladu aplikace.

Dalo by se říci, že meta-program je i použití maker v programovacím jazyce C++. Takto vytvořené meta-programy jsou však velice omezené, neboť makra v programovacím jazyce C++ umožňují pouze manipulaci s textovými řetězci a substituci. Na rozdíl od maker je šablonové meta-programování obecně turingovsky kompletní, což znamená, že jakýkoliv algoritmizovatelný problém by měl být v nějaké formě řešitelný pomocí šablonového meta-programování. Další vlastností šablonového meta-programování je absence přepisovatelných proměnných, což znamená, že jakmile je nějaká proměnná v meta-programu inicializována na nějakou hodnotu, již tuto hodnotu v průběhu programu nezmění. Díky této vlastnosti jsou meta-programy občas připodobňovány k obdobě funkcionálních programů. Této podobnosti nahrává i fakt, že jediným implementovaným způsobem kontroly běhu meta-programu je rekurze.

V jaké situaci tedy sáhnout po prostředcích šablonového meta-programování místo použití konvenčního programovacího přístupu? Převážně se jedná o situace, kde potřebujeme nějaký kód duplikovat tak, aby pracoval s více datovými typy nebo chceme na základě seznamu datových typů vygenerovat nějaké funkční celky.

Nyní bych zde představil několik příkladů demonstrující šablonové meta-programování. Jednou ze základních a nejvíce používaných konstrukcí je vytvoření šablonové třídy.

Představme si, že chceme implementovat třídu *Array* podobnou třídě *std::vector*. Její definice by mohla vypadat způsobem, který je znázorněn na obrázku 5.2.

```
1 template <typename TYPE>
2 class Array
3 {
4       ...
5       private:
6           TYPE m_data[];
7           size_t m_size;
8 };
```

Obrázek 5.2: Nástin šablonové definice třídy Array.

Důležitou částí definice třídy je její prefix *template <typename TYPE>*, který říká, že se jedná o šablonovou třídu a že jejím šablonovým argumentem je jméno datového typu, který budeme uvnitř třídy reprezentovat jménem *TYPE*. Konkrétní datový typ je šabloně předán při instanciaci třídy *Array*, tak jak je to vidět na obrázku 5.3.

```
1       ...
2       Array<int> array;
3       ...
```

Obrázek 5.3: Instanciace šablonové třídy Array.

Kdybychom pro každý datový typ museli psát speciální třídu nejen, že bychom dospěli k neúměrně dlouhému zdrojovému kódu, ale pokud bychom chtěli změnit implementaci některé z metod, museli bychom tuto změnu ručně přenést do všech zbývajících tříd pro ostatní datové typy. Šablonový návrh se o všechny tyto starosti postará sám.

Parametr šablony však nemusí být jen datový typ, ale i hodnota datového typu, například celočíselná konstanta typu *int*. Této vlastnosti můžeme s výhodou využít například při implementaci třídy pro maticové výpočty, kde rozměr matice můžeme specifikovat v době

překladu a kompilátor bude moci „rozbalit" případné cykly, protože bude znát přesný počet opakování cyklu. Dokonce nám šablony v tomto případě mohou posloužit i pro detekování zda mají matice korektní rozměry pro operaci maticového násobení v době překladu aplikace. Ilustrace takového návrhu je na obrázku 5.4.

```
1  template <typename TYPE, int M, int N>
2  class Matrix
3  {
4      public:
5          template <int P>
6          Matrix<TYPE, M, P> operator*(Matrix<TYPE, N, P> const& mat2)
7          { /* algoritmus nasobeni */ }
8      private:
9          TYPE m_data[M][N];
10 };
11 ...
12 Matrix<float, 10, 10> mat1;
13 Matrix<float, 3, 3>   mat2;
14 Matrix<float 10, 5>   mat3;
```

Obrázek 5.4: Nástin šablonové definice třídy Matrix.

Při pokusu mezi sebou vynásobit *mat1* a *mat2* by překlad skončil s chybou. Překladač by totiž nebyl schopný nalézt odpovídající implementaci pro metodu *operator**. Z definice metody totiž parametr *M* u *mat2* musí mít stejnou hodnotu jako parametr *N* u *mat1*. Matice *mat1* a *mat3* by šlo mezi sebou vynásobit bez problémů. Násobení by se nezdařilo ani v případě, kdyby matice byly různého typu.

Poslední šablonovou konstrukcí, kterou se budu v této podkapitole zabývat a která hraje klíčovou roli v implementaci indexovací struktury je využití takzvaných *typelistů* pro generování kódu. *Typelist* je struktura obsahující výčet datových typů. V jazyce C++ se dá definovat jako šablonová prázdná struktura obsahující pouze zadefinování alternativních jmen pro předané datové typy. Vzhledem k tomu, že se jedná o seznam, nové názvy jsou většinou *Head* a *Tail*. Definice *typelistu* je znázorněna na obrázku 5.5.

```
1 template <typename H, typename T>
2 struct
3 {
4     typedef H Head;
5     typedef T Tail;
6 };
```

Obrázek 5.5: Definice Typelistu.

S *typelistem* je dále spojena speciální struktura označovaná jako *TypeNull*, která slouží v *typelistu* jako „zarážka" za posledním prvkem v seznamu datových typů. Definice této „zarážky" je na obrázku 5.6.

```
1 class TypeNull { };
```

Obrázek 5.6: Definice „zarážky" TypeNull.

Pro snadnější konstrukci *typelistů* se používají makra jejichž činnost demonstruje obrázek 5.7.

```
1 #define TYPELIST_1(T1)          TypeList<T1, TypeNull>
2 #define TYPELIST_2(T1, T2)      TypeList<T1, TYPELIST_1(T2) >
3 #define TYPELIST_3(T1, T2, T3) TypeList<T1, TYPELIST_2(T2, T3) >
4 ...
```

Obrázek 5.7: Definice maker pro definování Typelistů.

Nad takto vytvořeným *typelistem* lze vytvořit celou řadu algoritmů. Můžeme implementovat procházení seznamem, přidávání či rušení prvků v seznamu, počítání prvků seznamu a další. Tyto operace zde nebudu detailněji rozebírat a případné zájemce o hlubší studium šablonového meta-programování bych odkázal na publikaci [1]. Poslední část této kapitoly věnuji nástinu konstrukce univerzálního datového úložiště ve kterém lze uchovávat kolekce datových typů specifikovaných pomocí *typelistu* aniž by museli sdílet nějakého společného předka.

Základem tohoto datového úložiště je struktura obsahující kontejner typovaný pro první datový typ *typelistu* a instanci sebe sama s kontejnerem typovaným pro další datový typ atd. dokud není rozgenerovaný celý *typelist*. Na obrázku 5.8 je vidět ukázka zdrojového kódu, který toto rozgenerování implementuje. Máme-li tomuto kódu porozumět, je třeba

```
1 template<class OBJECT> struct ContainerSet
2 {
3     std::set<OBJECT*> _element;
4 };
5
6 template<> struct ContainerSet<TypeNull>
7 {
8 };
9
10 template<class HEAD, class TAIL> struct ContainerSet< TypeList<HEAD, TAIL> >
11 {
12     ContainerSet<HEAD> _elements;
13     ContainerSet<TAIL> _TailElements;
14 };
```

Obrázek 5.8: Rekurzivní generování vnořených datových kontejnerů.

k němu přistupovat jako by se jednalo o kód napsaný ve funkcionálním jazyku. Rozgenerování započne na řádku 10, kde se z *typelistu* oddělí datový typ z jeho hlavy, na který se aplikuje kód z řádku 1 a vytvoří se tím datové úložiště *std::set* vygenerované pro první datový typ uvedený v *typelistu*. Na ocas seznamu se opět aplikuje pravidlo z řádku 10, atd. Rozgenerovávání končí v případě, ze hlava obsahuje datový typ *TypeNull*, který slouží jako „zarážka“ na konci *typelistu*. Pro tento datový typ se uplatní kód z řádku 6, kde je vygenerována prázdná struktura a rekurzivní generování končí.

S takto vygenerovanou strukturou by se však v dalším kódu pracovalo obtížně, je proto nutné k takto vygenerovanému úložišti přidat ještě obálku, která zjednoduší manipulaci s úložišti pro jednotlivé datové typy. Tato obálka využívá stejný princip jako byl použit pro vygenerování úložné struktury. Místo datových úložišť nyní budeme generovat metody, které budou k jednotlivým úložištím přistupovat. Nástin konstrukce této obálky je uvedený

na obrázku C.1 v přílohách.

5.3 Operace nad navrženou indexovací strukturou

V této kapitole podrobněji rozepíši operace, které jsou implementované a lze je provádět nad indexem včetně popisu použitých algoritmů při jejich implementaci a jejich grafem aktivit pro názornější pochopení popisovaného algoritmu. Při popisu algoritmů budu postupovat od interních operací, které nám poslouží jako základní stavební bloky pro další operace, směrem k operacím dostupných uživateli.

Při popisu algoritmů nad R*-Tree vycházím z publikace [8].

5.3.1 Algoritmus dělení listu v R*-Tree

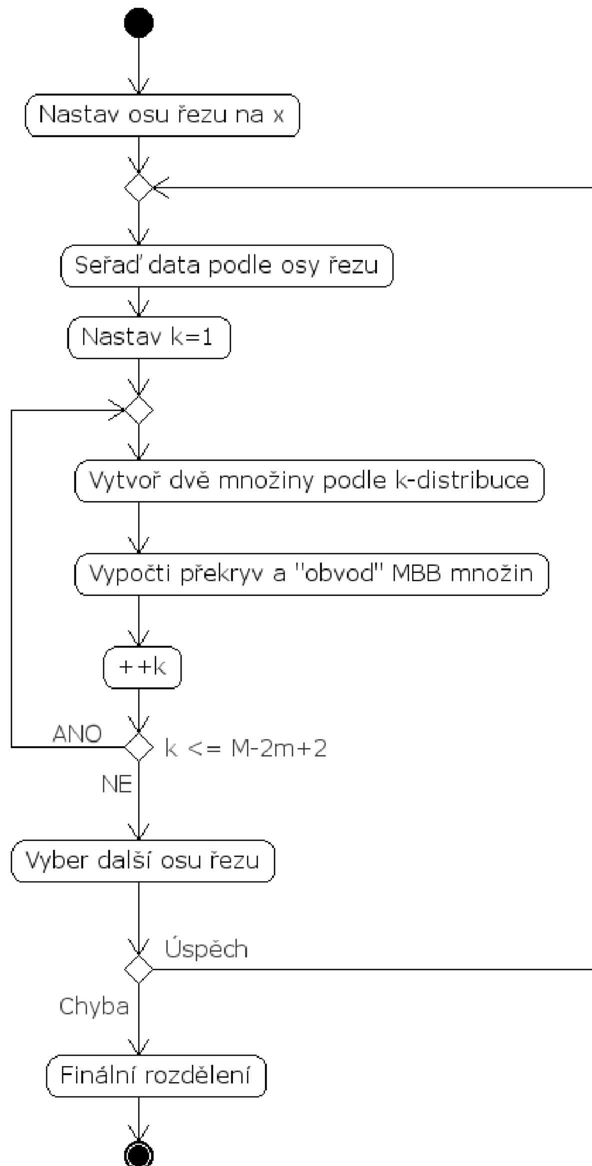
Algoritmus dělení listu je výpočetně nejnáročnější operací v R*-Tree. Probíhá zde řazení všech objektů obsažených v děleném listu podle všech os a analýza možných rozdělení této seřazené posloupnosti objektů na dvě podmnožiny tak, aby se minimalizovala velikost jednotlivých bounding boxů obklopujících tyto dvě podmnožiny a aby se pokud možno eliminoval překryv jednotlivých bounding boxů podmnožin. Algoritmus dělení listu pracuje následujícím způsobem.

Nejprve je potřeba zjistit podle které osy bude nejoptimálnější provést rozdělení listu. Postupujeme tak, že pro každou z os x, y, z provedeme seřazení všech $M + 1$ objektů podle aktuálně zkoumané osy, kde M je maximální kapacita listu. List následně rozdělíme na dvě disjunktní množiny objektů, přičemž je potřeba prozkoumat všech k možných distribucí, kde $k \in \langle 1, M - 2m + 2 \rangle$. Konstanta m je volena jako 20%, 30%, 40% nebo 45% z M. První výsledná množina obsahuje prvních $(m - 1) + k$ objektů, druhá množina obsahuje zbytek, tedy $(M + 1) - ((m - 1) + k)$. Pro každou osu se spočte suma délek hran v obou minimálních bounding boxech pro všechny možné distribuce objektů. Osa, ve které je tento součet minimální, je vybrána za osu, podle níž se provede dělení listu.

Dále je potřeba v množině objektů uložených v děleném listu nalézt bod, ve kterém se dělení provede. Ten nalezneme tak, že v seřazené posloupnosti objektů podle osy vybrané v předešlém kroku spočteme velikost překryvu minimálních bounding boxů obou podmnožin pro všechny distribuce a vybereme tu, jejíž překryv je nejmenší.

Literatura [8] uvádí jako nejoptimálnější volbu koeficientu m hodnotu 40%. Proto tuto hodnotu používám v implementaci jako výchozí.

Algoritmus dělení listu je graficky reprezentován jako diagram aktivit na obrázku 5.9.



Obrázek 5.9: *Diagram aktivit dělení listu v R*-Tree.*

5.3.2 Algoritmus vynuceného znovuvložení objektů v R*-Tree

Při vložení objektu do R*-Tree může dojít k přetečení objektů v listu a je potřeba takto nastalou situaci ošetřit. Před vlastním dělením listu nastupuje algoritmus vynuceného znovuvložení objektů.

Algoritmus pracuje tak, že všechno objekty v listu vzestupně seřadí v závislosti na jejich vzdálenosti od středu minimálního bounding boxu, který danou množinu objektů obklopuje. Z této množiny je následně odstraněno p první objektů a jsou upraveny hranice minimálního bounding boxu, tak aby těsně obklopoval zmenšenou množinu objektů. Zbylé objekty jsou znovu od kořene postupně zařazeny do stromu. Toto může způsobit v krajních případech přetečení v dalších listových uzlech vyvolat tak dělení uzlů ve větším počtu, běžnější situací však je, že objekty jsou zařazeny do zbylých listových uzlů, aniž by přetečení vyvolaly.

Literatura [8] uvádí, že strom vykazuje nejlepší vlastnosti pro koeficient $p = 30\%$ z maximální kapacity listu.

Algoritmus je opět pro názornost reprezentován jako graf aktivit na obrázku 5.10.

Obrázek 5.10: *Diagram aktivit vynuceného znovuvložení objektů v R*-Tree.*

5.3.3 Algoritmus ošetření přetečení v R*-Tree

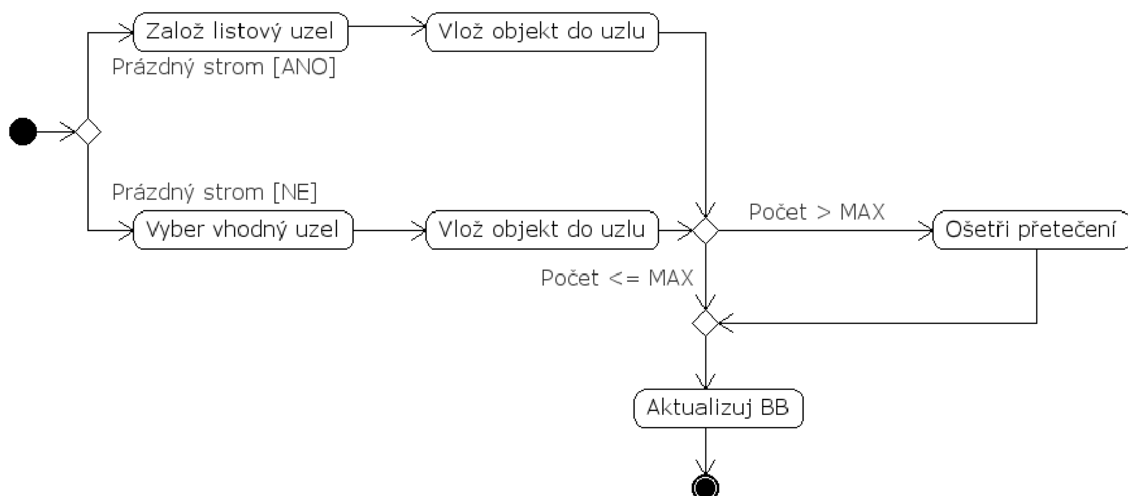
Při vkládání objektu do R*-Tree, které bude detailněji popsáno v následující kapitole 5.3.4, může nastat situace, kdy bude překročena maximální zadaná kapacita listu. V tomto případě je vyvolán algoritmus pro ošetření přetečení.

Algoritmus ošetření přetečení je ve své podstatě velice jednoduchý a pracuje s výše definovanými operacemi vynuceného znovuvložení objektu a rozdělení listu. Pokud se nenacházíme na úrovni kořene a k přetečení došlo poprvé v průběhu přidávání jednoho objektu, je zavolána metoda vynuceného znovuvložení. Ve všech ostatních případech dochází k rozdělení listu. Algoritmus je graficky reprezentován jednoduchým grafem aktivit na obrázku 5.11.

Obrázek 5.11: *Diagram aktivit průběhu ošetření přetečení v R*-Tree.*

5.3.4 Algoritmus vložení objektu do R*-Tree

Pokud vkládáme objekt do prázdného stromu, vytvoříme v kořenu nový listový uzel, do kterého vložíme vkládaný objekt a aktualizujeme jeho minimální bounding box, aby tento objekt zahrnoval. Vkládáme-li do již existující stromové struktury, je nejprve zavolán algoritmus pro nalezení vhodného listu. Ten prochází stromem a hledá minimální bounding box od jehož středu má vkládaný objekt nejmenší vzdálenost. Do takto nalezeného listového uzlu je objekt vložen. Pokud vložením objektu do listového uzlu došlo k překročení jeho kapacity - přetečení, je zavolán algoritmus pro ošetření přetečení. Na závěr jsou aktualizovány hranice všech minimálních bounding boxů na cestě od listu ke kořenu tak, aby zahrnovaly nově přidaný objekt. Popsaný algoritmus reprezentuje diagram aktivit na obrázku 5.12.

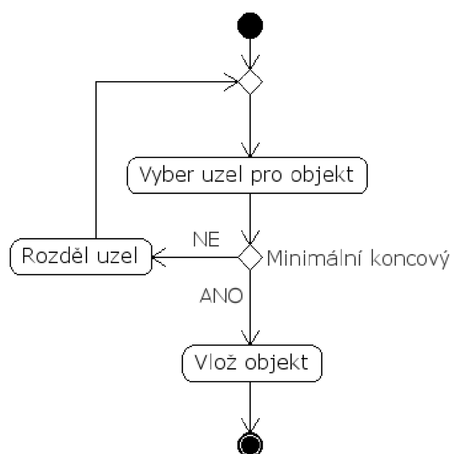


Obrázek 5.12: *Diagram aktivit průběhu vkládání objektu do R^*-Tree.*

5.3.5 Algoritmus vložení objektu do Quad-Tree

Vkládání objektu do Quad-Tree je v porovnání s vkládáním objektu do R*-Tree mnohem jednodušší. Objekt putuje přes jednotlivé uzly stromu, ve kterých je obsažen až k cílovému listovému uzlu, do kterého je vložen. Pokud se na cestě vyskytne uzel, který je listový, ale pokrývá větší prostor než který je specifikovaný pro listový uzel, vyvolá se jeho rozdělení na čtyři části. Toto dělení rekurzivně pokračuje dokud objekt není zařazen do listového uzlu, který pokrývá přesně specifikovanou část prostoru.

Algoritmus je opět demonstrován formou diagramu aktivit na obrázku 5.13.

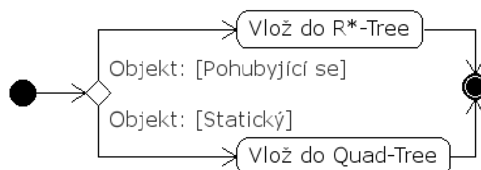


Obrázek 5.13: *Diagram aktivit průběhu vkládání objektu do Quad-Tree.*

5.3.6 Operace vložení nebo vymazání objektu z indexovací struktury

Při vkládání objektu do indexovací struktury je potřeba s objektem předat i informaci o tom, zda-li se jedná o objekt statický nebo pohybující se. Indexovací struktura sama

o sobě tuto otázku rozhodnout nemůže, toto rozhodnutí náleží tedy aplikaci využívající index. Statické objekty jsou vkládány do R*-Tree a pohybující se do Quad-Tree části, tak jak je znázorněno v diagramu aktivit na obrázku 5.14.



Obrázek 5.14: *Diagram aktivit průběhu vkládání objektu do hybridní struktury.*

Operace vymazání objektu z indexu probíhá ve stejném duchu. Dle typu objektu se vybere patřičná struktura, ze které se poté daný objekt odstraní.

5.4 Popis implementovaných datových struktur

V této kapitole se budu zabývat podrobněji vlastní implementací indexovací struktury a jejích dílčích prvků, jejich objektovým návrhem a popisem jednotlivých metod.

5.4.1 Struktura Point3D

Jedná se o základní strukturu, která reprezentuje bod v 3-rozměrném prostoru. Struktura je navržena jako šablonová a umožňuje v době překladu specifikovat interní typ proměnných, ve kterých jsou uloženy souřadnice bodu.

Z důvodu, že je někdy výhodné přistupovat k těmto souřadnicím prostřednictvím přímého odkazu jménem na danou složku a někdy je naopak výhodné přistupovat k jednotlivým souřadnicím jako k prvkům pole s indexy v intervalu $\langle 0, 2 \rangle$ kde indexu 0 odpovídá hodnota x-ové souřadnice, indexu 1 hodnota y-ové souřadnice a indexu 2 hodnota z-ové souřadnice, rozhodl jsem se datové úložiště pro tyto hodnoty implementovat za pomocí anonymního unionu a anonymní struktury. Union pracuje tak, že proměnné v něm definované se v paměti překrývají. Na obrázku 5.15 je pro lepší představu tato konstrukce uvedena.

```
1  template <class PRECISION>
2  struct Point3D
3  {
4  ...
5      union
6      {
7          struct
8          {
9              PRECISION x;
10             PRECISION y;
11             PRECISION z;
12         };
13
14         PRECISION coords[MAX_DIM];
15     };
16 ...
17 };
```

Obrázek 5.15: Třída Point3D s definicí úložiště souřadnice.

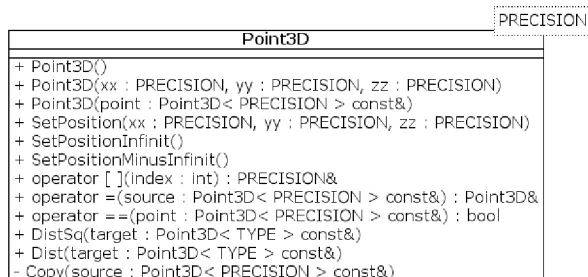
V třídě jsou dále implementovány metody pro výpočet vzdálenosti dvou bodů v prostoru. Tyto metody jsem implementoval ve dvou verzích. Metoda *DistSq* vrací jako výsledek neodmocněnou vzdálenost dvou bodů. Toto je výhodné v případech kdy nás nezajímá konkrétní absolutní hodnota vzdálenosti, ale provádíme například test, zda-li se dva objekty nacházejí v dané vzdálenosti. Můžeme pak s výhodou vypustit výpočetně náročné odmocňování a počítat jen s druhými mocninami vzdáleností. Metoda *Dist* vrací reálnou hodnotu vzdálenosti.

Další vlastností třídy je implementace metody *operator[]*, pomocí které je možné přímo přistupovat k jednotlivým souřadným složkám. Máme-li tedy objekt *point*, který bude instancí třídy *Point3D*, přístup k x-ové souřadnici lze realizovat jako *point[DIM_X];* [1] nebo *point.x;* .

[1] Pro zvýšení abstrakce jsem definoval výčtový typ Dimensions, kde položka DIM_X má hodnotu 0.

Pro pohodlné nastavení všech souřadnic na maximální / minimální hodnotu,kterou lze reprezentovat v datovém typu, který je specifikovaný šablonovým argumentem třídy, lze využít metody *SetPositionInfinit* / *SetPositionMinusInfinit*.

Návrh třídy *Point3D* je zobrazen na obrázku 5.16.



Obrázek 5.16: *Návrh třídy Point3D.*

5.4.2 Třída BoundingBox3D

Třída *BoundingBox3D* je stejně jako třída *Point3D* šablonová a opět je pomocí šablony specifikován datový typ pro reprezentaci interních datových typů. Třída představuje trojrozměrnou obálku, jejíchž hraniční plochy jsou vždy rovnoběžné s osamy souřadného systému. Tuto obálku tedy nelze nikterak „natáčet“. Díky této vlastnosti se zjednodušují některé výpočetní operace, jako je například zjištění, zda-li se dva tyto bounding boxy překrývají, či nikoliv.

Informace o poloze a velikosti bounding boxu v prostoru jsou uloženy ve formě dvou instancí třídy *Point3D*, které udávají souřadnice levého dolního a pravého horního rohu (úhlopříčně). V instanci třídy *BoundingBox3D* je na základě těchto informací také vypočítán její střed (taktéž uložen jako instance třídy *Point3D* a délky jednotlivých hran.

Důležitými metodami této třídy jsou metody pro výpočet průniku s instancí třídy *Point3D* tak s jinou instancí třídy *BoundingBox3D*. Pro výpočet zda-li existuje průnik mezi dvěmi instancemi třídy *BoundingBox3D* jsem využil teorém dělících přímek[15]. Podle tohoto teorému pro dva objekty konvexního tvaru existuje dělící přímka tehdy a jen tehdy, pokud tyto objekty nesdílí žádný průnik. Díky tomu, že *BoundingBox3D* má své stěny vždy rovnoběžné s osami souřadného systému, tak i dělící přímka, pokud existuje, musí být rovnoběžná s osami souřadného systému. Test na průnik lze tedy pro tyto objekty provést jednoduchým porovnáním jejich souřadnic.

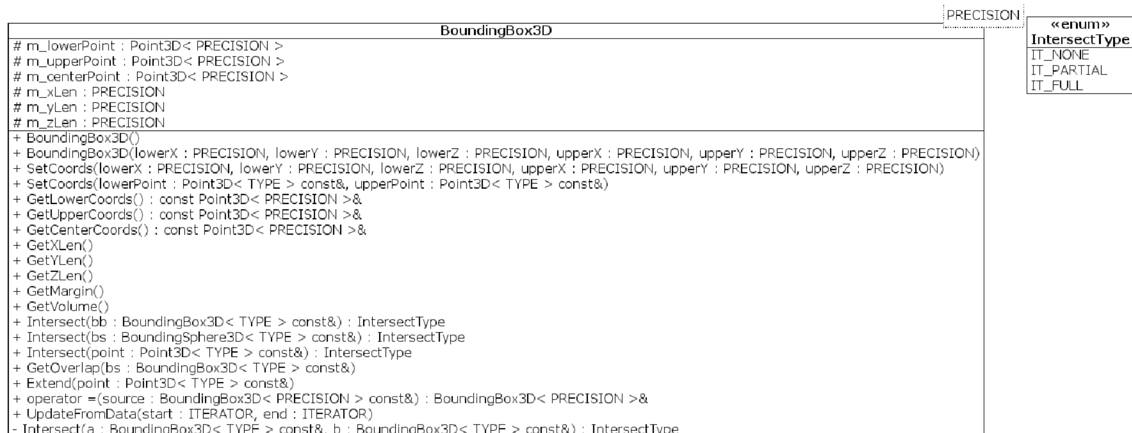
Je-li u dvou bounding boxů identifikován průnik, může nás v některých případech (například dělení listu v R*-Tree) zajímat objem společné oblasti. Tuto hodnotu lze získat pomocí metody *GetOverlap*, jejíž argumentem je další instance třídy *BoundingBox3D*.

Pro výpočet objemu, který bounding box zaujímá v prostoru, slouží metoda *GetVolume*. Tato metoda společně s metodou *GetMargin*, která vrací součet délek bounding boxu v jednotlivých osách, hraje důležitou roli v optimalizačních heuristikách dělení listu v R^*-Tree.

Třída obsahuje i metodu pro rozšíření svých hranic, aby pojala specifikovaný bod v prostoru. Voláním metody *Extend* s parametrem typu *Point3D* způsobí rozšíření hranic tak, že specifikovaný bod bude zahrnut v oblasti pokrývané bounding boxem. V případě, že potřebujeme rozšířit nebo zmenšit bouding box tak, aby těsně obepínal danou množinu dat, využijeme metodu *UpdateFromData*, jejíž dva argumenty jsou počáteční a koncový iterátor do kolekce objektů podle kterých chceme bounding box modifikovat. Nutno podotknou, že

se jedná o kompletní znovunastavení krajních bodů bounding boxu a není tedy zachována ani souřadnice jeho středu.

Návrh třídy *BoundingBox3D* je zobrazen na obrázku 5.17.



Obrázek 5.17: *Návrh třídy BoundingBox3D.*

5.4.3 Třída BasicVisitor

BasicVisitor je abstraktní třídou definující základní rozhraní pro další uživatelské visitory. Každý uživatelský visitor musí být poděděn z této třídy. Třída je navržena jako šablonová, jejíž šablonové argumenty specifikují typ objektu nad kterým se bude „návštěva“ provádět a typ bounding boxu, kterým se specifikuje oblast ve které se mají navštívené objekty nacházet.

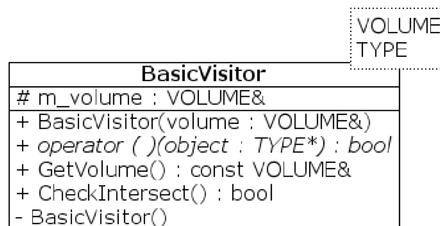
Výhodou visitoru je možnost uchovávat si v sobě svůj vlastní interní stav, který se může měnit v průběhu vykonávání definované operace nad nalezenými objekty, což umožňuje tvorbu dynamičtějších a komplexnějších dotazů, než které bychom byli schopni konstruovat s využitím konvenční, úzce specializované, metody *Select*.

Činnost visitoru je definována metodou *operator()* s logickou návratovou hodnotou udávající zdali při aplikaci metody visitoru na daný objekt nedošlo k jeho přemístění v prostoru a jestli je zapotřebí aktualizovat index s novou polohou objektu.

Třída *BasicVisitor* také obsahuje virtuální metodu *CheckIntersect* jejíž logická návratová hodnota udává zdali se při průchodu visitoru stromem má kontrolovat průnik visitoru s jednotlivými uzly stromu. V případě, že visitor reprezentuje prostorový dotaz, je zřejmé, že je potřeba kontrolovat průniky. V některých případech však může vizitor fungovat jen jako globální funktor pro všechny objekty uložené v indexu[2]. V takovém případě by akční prostor pro visitor byl shodný s prostorem, který je pokrytý indexem a každý test na průnik by byl kladně vyhodnocený. Tato metoda tedy zabraňuje zbytečnému výpočtu průniků v případech, kdy to není nezbytně nutné a šetří tak procesorový čas.

Návrh třídy *BasicVisitor* je zobrazen na obrázku 5.18.

[2]Například je potřeba pro všechny uložené objekty zavolat globálně metodu *Update*.

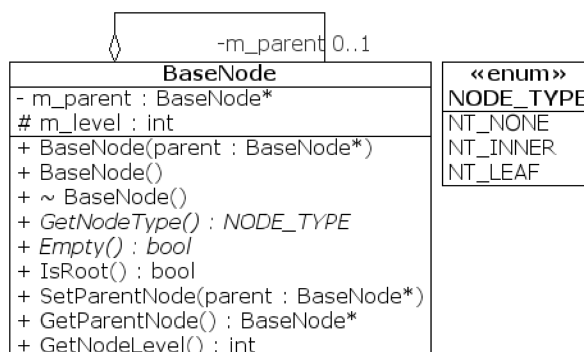


Obrázek 5.18: *Návrh třídy BasicVisitor.*

5.4.4 Třída BaseNode

Abstraktní třída implementující společné vlastnosti pro všechny uzly jak v R*-Tree, tak i v Quad-Tree. Třída obsahuje čistě virtuální metody *Empty* pro zjištění prázdnosti uzlu a *GetNodeType* pro zjištění typu uzlu, který je daný výčtovým typem *NODE_TYPE*. Typ uzlu může být vnitřní uzel stromu nebo listový uzel stromu. V třídě je uchován ukazatel na rodiče uzlu. Je-li tento ukazatel nastaven na hodnotu *NULL*, pak se jedná o kořenový uzel. Tato vlastnost je reflektována metodou *IsRoot*, která pravdivostní hodnotou reflektuje výše popsané pravidlo. Samozřejmostí jsou i metody pro nastavení nebo získání ukazatele na rodičovský uzel *SetParentNode* a *GetParentNode*.

Návrh třídy *BaseNode* je zobrazen na obrázku 5.19.



Obrázek 5.19: *Návrh třídy BaseNode.*

5.4.5 Třída RTreeNode

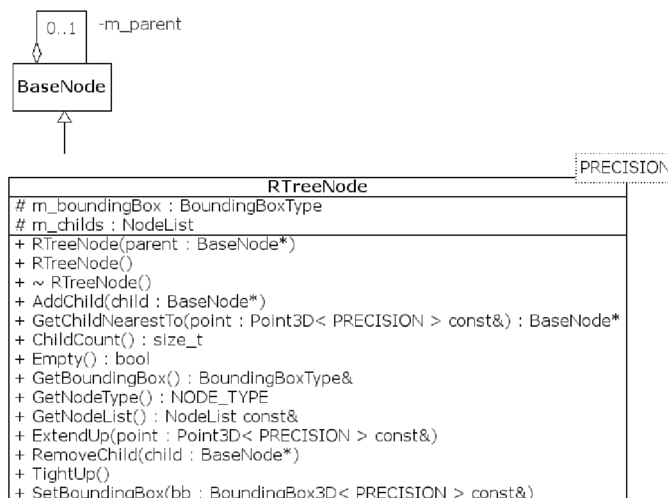
Třída reprezentuje nelistový uzel v R*-Tree. Jedná se o potomka třídy *BaseNode*, rozšířeného o další metody a atributy. Tato třída je na rozdíl od svého rodiče šablonová. Šablonovým parametrem se zde specifikuje datový typ, který je použitý pro interní data bounding boxu, který je spjatý s daným interním uzlem stromu.

Každá instance této třídy obsahuje instanci třídy *BoundingBox3D*, která vymezuje část prostoru, kterou uzel reprezentuje. Dále si každý interní uzel uchovává seznam ukazatelů na svoje následníky. Následníci mohou být další interní uzly a nebo právě jeden listový uzel. Pro přidání následníka pod daný uzel slouží metoda *AddChild*. Počet následníků daného uzlu vrací metoda *GetChildCount*. Tato metoda volá metodu *size* použitého STL[3] kontej-

[3] Zkratka STL znamená Standart Template Library.

neru. Proto se v některých implementacích nedoporučuje volání této metody, v kombinaci s porovnáním na nenulovost, používat pro zjištění, zda-li je kontejner prázdný. Proto jsem implementovat metodu *Empty*, která volá příslušnou metodu *empty* STL kontejneru, která je v některých implementacích rychlejší než uvedená metoda *size*. V situaci, kdy procházíme posloupností uzlů a hledáme vhodný listový uzel pro zařazení nového prvku, je vhodné mít metodu, která ze všech následníků uzlu vybere ten s nejmenší vzdáleností od daného objektu specifikovaného souřadnicí v prostoru pomocí typu *Point3D*. Přesně tuto funkci plní metoda *GetChildNearestTo*. Další poměrně důležitou metodou využitou při vkládání objektů do stromu je metoda *ExtendUp*, která rozšíří minimální bounding boxy všech svých předků na cestě ke kořenu tak, aby pojaly konkrétně specifikovaný bod v prostoru daný instancí třídy *Point3D*.

Návrh třídy *RTreeNode* je zobrazen na obrázku 5.20.



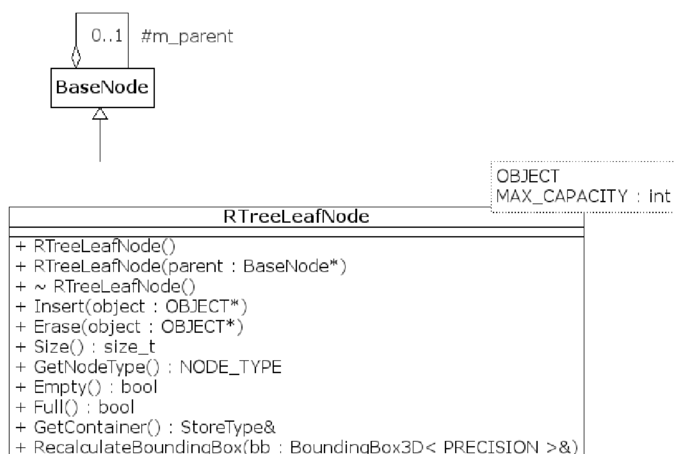
Obrázek 5.20: *Návrh třídy RTreeNode.*

5.4.6 Třída RTreeLeafNode

Šablonová třída poděděná opět z třídy *BaseNode*. Šablonovými argumenty se specifikuje datový typ objektu, který bude uložen v listu a maximální kapacita listu.

Tato třída je inteligentní obálkou okolo STL kontejneru *std::vector*, ve které jsou uloženy ukazatele na data. V třídě jsou implementované standardní metody pro zjištění počtu uložených objektů *Size*, příznaku prázdnosti *Empty* a příznaku naplnění maximální kapacity *Full*. Třída rovněž obsahuje metody *Insert* a *Erase* pro vložení a odebrání specifikovaného objektu z kolekce uložených objektů.

Návrh třídy *RTreeNode* je zobrazen na obrázku 5.21.



Obrázek 5.21: *Návrh třídy RTreeLeafNode.*

5.4.7 Třída RTree

Šablonová třída reprezentující celý R*-Tree. Šablonové atributy specifikují datový typ uloženého objektu, maximální kapacitu listových uzlů a datový typ použitý pro uchovávání souřadnic minimálních bounding boxů nelistových uzlů stromu.

Třída obsahuje metodu *Insert* pro přidávání objektů do stromové struktury a metodu *Erase* pro odebírání objektů ze stromové struktury.

Provádění operací nad stromem zajišťuje metoda *Visit*. Argumentem této metody je instance potomka třídy *BasicVisitor*, která v sobě implementuje logiku operace k provedení. Činnost metody spočívá v zavolání pomocné soukromé metody *VisitHelper*, s argumenty kořenového uzlu a referencí na instanci visitoru. Tato metoda rekurzivně prochází stromem a aplikuje visitor na příslušné nalezené objekty. Po skončení běhu pomocné metody se předá řízení zpět metodě *Visit*, která projde seznam objektů pro opětovné vložení, které mohly vzniknout aplikováním visitoru a modifikováním souřadnic objektů, a vloží tyto objekty zpět do stromu.

Pro úspěšné zkombinování R*-Tree a Quad-Tree do dále popsané hybridní struktury RQ-Tree bylo zapotřebí implementovat systém oznamování o vzniku nového listového uzlu, o změnách minimálního bounding boxu listového uzlu (při operaci dělení nebo vynuceného znovuvložení) a o situaci kdy visitor úspěšně nalezl listový uzel s daty. Tyto informace jsou kritické pro udržování relace mezi listovými uzly R*-Tree a listovými uzly Quad-Tree v dále implementované hybridní struktuře.

Oznamování jsem implementoval jsou sérii tří tzv. *CallBacků*[4]. Ukázka definice jednoho z těchto *CallBacků* je pro lepší představu zobrazena na obrázku 5.22.

Jak je vidět *CallBack* je složen ze dvou částí. První je definice veřejného rozhraní *IOnLeafCreateOrUpdateCallBack*, druhá je definice šablonového *CallBacku OnLeafCreateOrUpdateCallBack*. Šablonové argumenty generují strukturu pro konkrétní typ objektu *OWNER* pro který budeme volat jeho metodu *METHOD*. Při instanciaci této třídy je potřeba jí předat referenci na instanci třídy typu *OWNER*, abychom věděli, pro jakou konkrétní instanci budeme metodu volat.

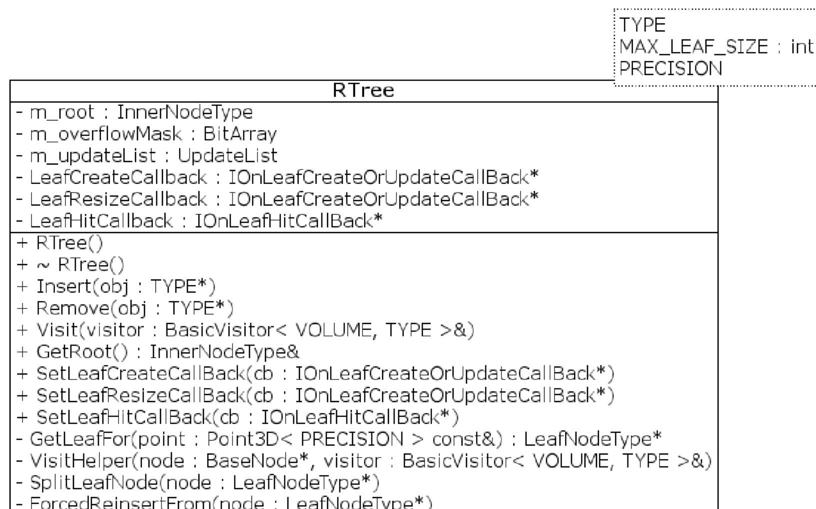
[4] CallBack je v našem případě struktura zapouzdřující ukazatel na členskou metodu jiné třídy, kterou lze pomocí CallBacku zavolat.

```
struct IOnLeafCreateOrUpdateCallBack
{
    virtual void Call(LeafNodeType* leaf, BoundingBox3D<PRECISION> const& bb)
        = 0;
};

template <class OWNER, void (OWNER::*METHOD)(LeafNodeType*,
BoundingBox3D<PRECISION> const&)>
struct OnLeafCreateOrUpdateCallBack : public IOnLeafCreateOrUpdateCallBack
{
    OnLeafCreateOrUpdateCallBack(OWNER& owner) : m_owner(owner) { }
    void Call(LeafNodeType* leaf, BoundingBox3D<PRECISION> const& bb)
    {
        (m_owner.*METHOD)(leaf, bb);
    }
    private:
        OWNER& m_owner;

        OnLeafCreateOrUpdateCallBack();
};
```

Obrázek 5.22: Ukázka definice jednoho z CallBacků třídy R^*-Tree.

V třídě jsou taktéž implementovány metody pro dělení listu *SplitLeafNode* a pro vynucené znovuvložení objektů *ForcedReinsertFrom*. Obě tyto metody mají jako argument listový uzel R*-Tree. Metody jsem záměrně implementoval do této třídy a ne do třídy *RTreeLeafNode*, protože je nutné z těchto metod přistupovat ke *CallBackům* a volat je. Tento problém by šlo vyřešit i tak, že by každý listový uzel stromu obsahoval ukazatel nebo referenci na instanci třídy R*-Tree ovšem za o něco zvýšené paměťové nároky.

Návrh třídy *RTree* je zobrazen na obrázku 5.23.



Obrázek 5.23: *Návrh třídy RTree.*

5.4.8 Třída QTreeNode

Šablonová třída, potomek třídy *BaseNode*, reprezentuje jak vnitřní, tak listový uzel Quad-Tree indexovací struktury. Šablonové argumenty zde udávají informaci o maximální hloubce stromu, o typu objektů, které budou v listech uloženy a o datový typ, který je použitý pro souřadný systém.

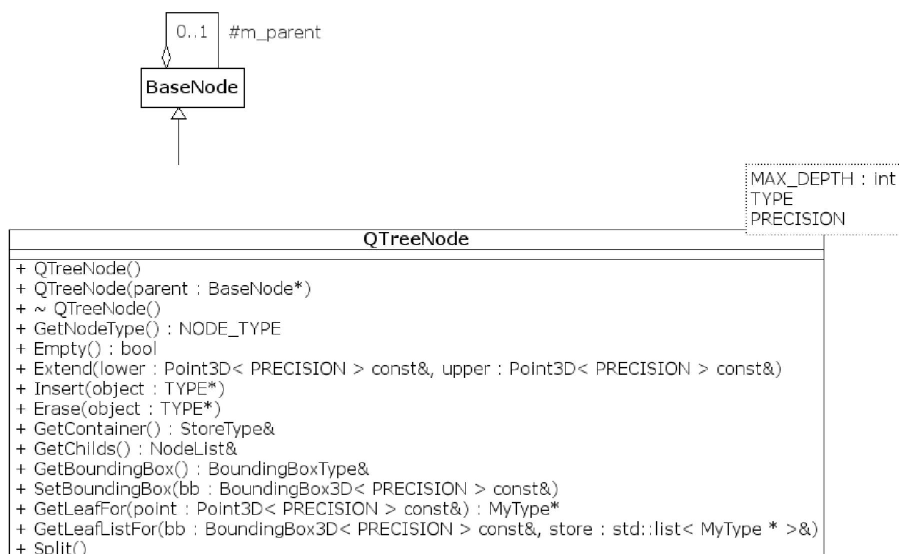
Třída obsahuje metody *Insert* a *Erase* pro vložení nebo odstranění objektu z datového úložiště listu. Volání těchto metod pro nelistový uzel způsobí vyvolání makra *assert* a násilné ukončení běhu programu.

K nalezení listového uzlu slouží metoda *GetLeafNodeFor*, jejímž argumentem je instance třídy *Point3D*. Je-li metoda vyvolána pro nelistový uzel, projde kolekci svých následníků a pro prvního následníka, jehož bounding box pokrývá daný bod, je opět zavolána metoda *GetLeafNodeFor*. Metoda volaná pro listový uzel vrací ukazatel *this*. Pokud ve stromu, v době volání této metody, neexistují patřičné uzly vedoucí k cílovému listovému uzlu, je celá cesta až k listu vytvořena pomocí opakovaného volání metody *Split* a metoda vrací ukazatel na nově založený listový uzel.

Pro potřebu linkovat listové uzly z R*-Tree na listové uzly v Quad-Tree je zde implementovaná metoda *GetLeafListFor*, jejíž argumenty jsou konstantní reference na instanci třídy *BoundingBox3D* a reference na datové úložiště typu *std::vector<QTreeNode*>*. Metoda pracuje velice podobně jako *GetLeafNodeFor* s tím rozdílem, že se hledá průnik s bounding boxem, a ne s bodem, a jsou prozkoumány všechny následnické uzly s neprázdným průnikem, a ne jen první z nich jako u metody *GetLeafNodeFor*. V listových uzlech je přidán do datového úložiště ukazatel *this* pro dosažený listový uzel.

Dělení uzlu probíhá, jak již bylo popsáno výše, pomocí metody *Split*. Vzhledem k tomu, že Quad-Tree dělí prostor na čtyři další podprostory, nabízí se otázka, jak zvolit osy pro dělení. Kdybychom chtěli dělit podle všech tří os, dostaly bychom osm podprostorů, což logika Quad-Tree nedovoluje. V implementaci jsem se rozhodl dělit prostor pouze podle os x a y. Důvodem k této volbě je analýza prostorových dat v cílové aplikaci diskutovaná v kapitole 4.2, kde je vidět že rozptyl v ose z je řádově menší než v osách x a y.

Návrh třídy *QTreeNode* je zobrazen na obrázku 5.24.



Obrázek 5.24: *Návrh třídy QTreeNode.*

Kapitola 6

Testování

Tato kapitola se zabývá testováním vytvořené indexovací struktury. Testování jsem rozdělil na několik dílčích částí. V následující podkapitole 6.1 představím mnou navržený testovací model pro testování dílčích třídních funkcionalit, podkapitola 6.4 se zabývá syntetickým testováním výkonnosti implementované indexovací struktury a na závěr v podkapitole 6.5 shrnu dosažené naměřené výsledky při použití na reálných datech v aplikaci MaNGOS.

6.1 Testování funkcionality jednotlivých tříd

Při implementace netriviální aplikace nebo knihovny, nejen v objektově orientovaných jazycích, je velice důležitou částí implementace průběžné testování jednotlivých implementovaných komponent. Díky možnosti otestovat jednotlivě každou dílčí komponentu výsledného programového celku na skupině navržených testů se můžeme později vyhnout komplikacím při sestavování těchto komponent do větších funkčních celků a částečně tak eliminujeme riziko, že se v průběhu implementace knihovny nebo aplikace budeme muset potýkat se složitým hledáním chyby způsobené nekorektním chováním některé použité komponenty. Čas strávený na připravě jednotlivých testů nám tak může v pokročilejších fázích implementace ušetřit daleko více času, než který jsme strávili psaním testů.

Pro účely testování funkcionality jednotlivých tříd a jejich metod jsem se rozhodl nepoužít žádné dostupné existující nástroje[1], ale implementoval jsem vlastní, dostatečně jednoduchý a pro účely základního testování funkcionality tříd, naprosto dostačující framework.

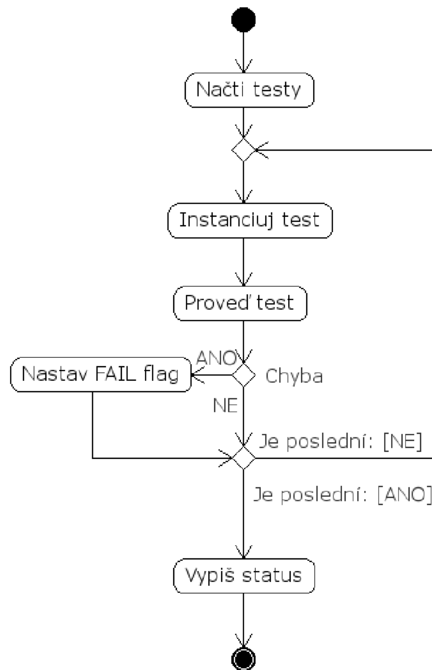
Jeho základem je abstraktní třída *BaseTest*, jejíž čistě virtuální metodu *Run* musí implementovat každý test poděděný z této rodičovské třídy. V této metodě je definována činnost testu, přičemž metoda může zahrnovat i více po sobě následujících testů a testovat tak nejen dílčí třídu nebo metodu, ale i skupiny tříd nebo metod. Metoda vrací hodnotu typu *bool*, která signalizuje zda daný test proběhl úspěšně či nikoliv.

Pro každý test musí být definovány dvě další netřídní metody. První pro vytvoření instance daného testu a druhá pro jeho zaregistrování do úložiště testů. Každý test také musí být jednoznačně identifikovatelný svým jménem, reprezentovaným jako řetězec znaků. Při startu testovací aplikace nejsou tedy vytvořeny naráz všechny instance testů, ale testy se vytváří dynamicky v průběhu testování. Díky tomu je snížena paměťová náročnost celé testovací aplikace.

Jednotlivé testy jsou ukládány v instanci třídy *TestStore* jako seznam dvojic (jméno, ukazatel na metodu konstruující test). Třída disponuje metodami pro dávkové spuštění

[1] Například CppTest dostupný na URL:http://cpptest.sourceforge.net/

všech testů a nebo jednotlivých testů zadaných jejich jménem. Po skončení každého testu je na standardním výstupu oznámeno, jestli test proběhl úspěšně či nikoliv. Navíc každý test může v průběhu běhu na standardní výstup vypisovat i detailnější informace o tom, která jeho část selhala s případnými detailními informacemi o chybě. Pro detailnější představu o tom, jak testování probíhá můžete získat z obrázku 6.1.



Obrázek 6.1: *Diagram aktivit průběhu testování.*

Na tomto místě bych také rád poznamenal, že vytvoření tohoto testovacího frameworku a vůbec celá myšlenka testovaní jednotlivých funkcionalit tříd se ukázala jako velice prozíravá. Díky těmto testům odhalil řadu „hloupých“ chyb, které by se mi v pokročilejším stádiu implementace hledali velice obtížně.

6.2 Systém automatického testování výkonnosti

Pro potřeby automatického testování výkonnosti jednotlivých implementovaných indexovacích metod jsem navrhl jednoduchý testovací nástroj. Jeho základem jsou třída *TimeCounter*, která poskytuje funkcionalitu pro měření času a struktura *BenchArgs*, která definuje jednotlivé testy prováděné nad indexovacími strukturami. Tyto struktury podrobněji rozeberu v následujících dvou podkapitolách. Vykonání testů je provedeno v metodě *TestTree*, která je pomocí šablon rozgenerována pro všechny typy testovaných indexovacích struktur, a která jako parametr dostává konstantní referenci na instanci struktury typu *BenchArgs*.

Výsledky jednotlivých testů jsou uloženy do souborů, které mají jmenný formát *<jméno indexu>_bench_<typ operace>_<typ dat>.dat*

Soubor obsahuje na každém řádku výsledek testu pro konkrétní operaci a počet objektů obsažených v indexu. První hodnota je počet objektů, druhá je doba potřebná ke zpracování operace v milisekundách. Jako oddělovač je použit tabulátor.

6.2.1 Struktura BenchArgs

Jedná se o prostou strukturu obsahující pouze proměnné a žádné metody. Význam jednotlivých proměnných je shrnut v následující tabulce 6.1. Popisy jednotlivých výčtových typů

Tabulka 6.1: Popis struktury BenchArgs.

Typ Proměnné	Název proměnné	Význam
BenchFlags	benchFlag	Příznak udávající pro jaký typ indexu se provede test. Viz tabulka B.1.
unsigned int	objCount	Udává pro jaký počet objektů v indexu se provede test.
unsigned int	executionCount	Udává kolikrát se má daný test zopakovat, výsledek je průměrem jednotlivých opakování.
unsigned int	operationCount	Udává kolik operací daného typu se má provést měření času.
unsigned int	maxX, maxY, maxZ	Definují maximální meze pro indexovaná prostor, spodní hodnota je implicitně 0.
ObjectType	objType	Udává jaký typ objektů se bude vyskytovat v indexu a v jakém poměru. Viz tabulka B.2.
BenchType	benchType	Udává o jaký typ testu se jedná. Viz tabulka B.3

jsou uvedeny v příloze B.

6.3 Metodika testování výkonnosti

Jednotlivé implementované indexovací struktury byly přeloženy s parametry a testovány na sestavě uvedené v tabulce 6.2.

Tabulka 6.2: Popis testovací sestavy a jejího sw. vybavení.

Hardware	
CPU	AMD Athlon(tm) 64 X2 Dual Core Processor 4400+
Frekvence CPU	2300.02-MHz
RAM	6GB (800MHz)
HDD	2xWDC WD6400AAKS-65A7B0 (ZFS Mirror)
Software	
OS	FreeBSD 8.0-STABLE (64b)
GCC	verze 4.2.1
Parametry překladu	
CXXFLAGS	-O3 -funroll-loops -ffast-math -fomit-frame-pointer -march=athlon64 -pipe

Každý test byl pro daný počet prvků v indexu a danou operaci spuštěn 5x a výsledný čas potřebný k vykonání operace byl pak vypočítán jako průměrná hodnota naměřených časů.

6.4 Syntetické testování výkonnosti

Syntetické testování je testování za pomoci uměle generovaného rozložení bodů v prostoru. Taktéž mohou být pro testování použity některé posloupnosti objektů, které se pro testovanou indexovací metodu považují obecně za nevhodné, aby se zjistila výkonnost testované indexovací metody v nejhorším možném případě. Následující syntetické testování jsem provedl na uniformě rozloženém vzorku náhodně generovaných dat. Testy na takovémto vzorku dat poskytují informaci o střeńí výkonnosti indexu.

Syntetické testy probíhaly v prostoru vymezeném souřadnicemi kvádru $< 0, 0, 0 >$ pro levý dolní přední roh a $< 15000, 15000, 1000 >$ pro pravý horní zadní roh. Postupně do tohoto prostoru bylo generováno 1M až 5M objektů uniformně v kroku po 200k objektech náhodně rozprostřených v celém testovaném objemu. Podle typu testu se jednalo o objekty statické, pohybující se a nebo kombinace obou typů v daném poměru.

Indexovací struktury byly nastaveny na parametry uvedené v tabulce 6.3 pro R*-Tree a 6.4 pro Quad-Tree. Datový typ *int* pro reprezentaci souřadnic v prostoru jsem vybral

Tabulka 6.3: Nastavení parametru testovaného R^*-Tree.

Přesnost datového typu souřadnic	int
Maximální kapacita listu	100
Faktor vynuceného znovuvložení	30%
Poměr dělení listu	40%:60%

z důvodu, že operace s tím to datovým typem jsou velice rychlé a pro potřeby syntetického testování nebylo nutné pracovat v plovoucí řadové čárce.

Maximální kapacitu listu jsem zvolil jako 0,01% z minimálního počtu testovacích objektů.

Faktor vynuceného znovuvložení a poměr dělení listů jsem nastavil na doporučené hodnoty uvedené v publikaci [8].

Tabulka 6.4: Nastavení parametru testovaného Quad-Tree.

Přesnost datového typu souřadnic	int
Maximální hloubka stromu	4
Hranice indexovaného prostoru	$< 0, 0, 0 >$ ... $< 15000, 15000, 1000 >$

Maximální hloubka stromu byla volena tak, aby byl zachován předpoklad, že velikost minimálního bounding boxu v listovém uzlu je $<<$ než velikost bounding boxu v listovém uzlu R*-Tree. Hloubka 4 tedy znamená, že celý prostor je rozdělen na 4^4 podprostorů, které mají rozměr $< 937, 937, 1000 >$. Pokud uvážíme že se v prostoru nachází maximální počet objektů (5M), jednoduchou kalkulací zjistíme, že v jednom listovém uzlu se průměrně nachází přibližně 15625 objektů, což s vysokou pravděpodobností splňuje výše uvedenou podmínku.

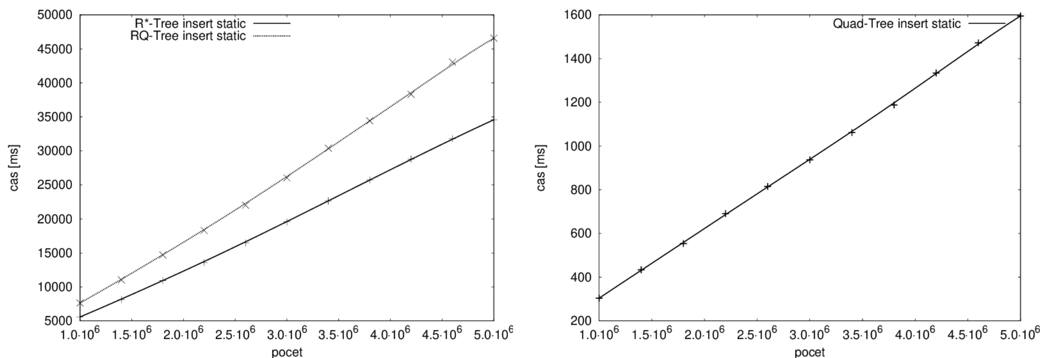
6.4.1 Výkon operace vkládání dat

Rychlost operace vložení daného počtu nepohybujících se objektů do indexu je shrnuta v následující tabulce 6.5

Tabulka 6.5: Tabulka srovnání naměřených časů pro jednotlivé indexovací struktury operace vložení statických dat.

Počet objektů	Čas R*-Tree [ms]	Čas Quad-Tree [ms]	Čas RQ-Tree [ms]
1000000	5584	304	6411
1400000	8148	434	11035
1800000	10924	554	14705
2200000	13588	691	18301
2600000	16465	815	22007
3000000	19559	937	26027
3400000	22643	1062	30390
3800000	25733	1188	34425
4200000	28810	1334	38311
4600000	31795	1472	43066
5000000	34578	1595	46572

Z testů je zřejmá převaha Quad-Tree nad všemi ostatními indexovacími strukturami. Je to dáno jeho architekturou. Vzhledem k nízké maximální hloubce stromu, je požadovaný listový uzel nalezen velice rychle. R^*-Tree a jeho RQ-Tree kombinace musejí provádět výpočetně náročnou heuristiku při operaci dělení listu, z této heuristiky plyne právě onen enormní nárůst potřebného času k zaindexování daného počtu objektů. Grafická reprezentace naměřených hodnot je zanesena do grafů na obrázku 6.2.



(a) Porovnání výkonnosti R^*-Tree a RQ-Tree.

(b) Quad-Tree.

Obrázek 6.2: Grafy časové složitosti pro operaci vkládání jednotlivých indexů.

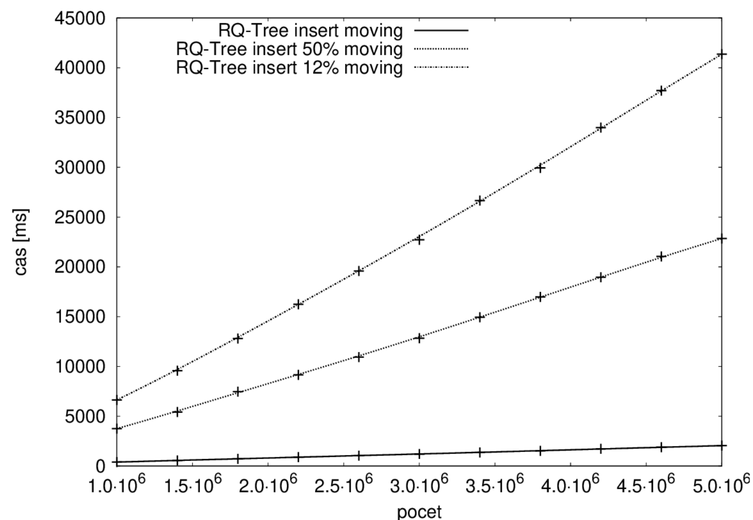
Z grafů je patrný nárůst potřebného času k vložení objektů do indexu u RQ-Tree. Tento nárůst je způsoben tím, že při každé operaci vytvoření, rozdělení listu nebo vynuceného znovuvložení objektů jsou přepočítány reference listových uzlů R^*-Tree na patřičné listové uzly v Quad-Tree.

Výkonnost kombinace RQ-Tree je při operaci vkládání dat velice závislá na jejich typu. Výkonnostní rozdíly pro různé typy objektů jsou shrnuty v následující tabulce 6.6 a grafu

na obrázku 6.3.

Tabulka 6.6: Změna naměřených časů při vkládání různých typů objektů do RQ-Tree.

P. objektů	Čas[ms] 100% pohyblivých	Čas[ms] 50% pohyblivých	Čas[ms] 12% pohyblivých
1000000	393	3758	6637
1400000	563	5419	9579
1800000	718	7471	12807
2200000	892	9158	16238
2600000	1046	10936	19579
3000000	1208	12840	22720
3400000	1366	14937	26654
3800000	1519	16985	29944
4200000	1723	18951	33989
4600000	1886	21046	37707
5000000	2043	22848	41357



Obrázek 6.3: Změna časové složitosti při vkládání různých typů objektů do RQ-Tree.

6.4.2 Výkon operace vyhledávání v datech

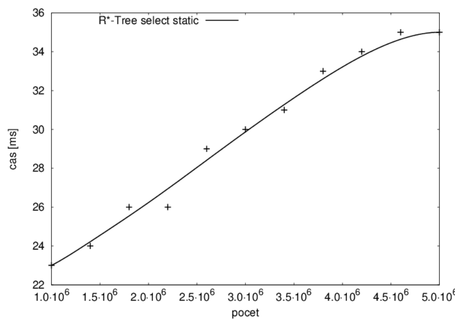
Výkon vyhledávání jsem testoval na sadách objektů stejného počtu jako v předchozí kapitole, která se zabývala výkonem vkládání objektů. Pro každou testovací sadu bylo provedeno 10000 operací vyhledání v náhodně generované části indexovaného prostoru o náhodné velikosti nepřekračující 5% celkového objemu indexovaného prostoru. Každý test byl opět proveden 5x a výsledná hodnota je průměrem všech měření.

Rychlost operace vložení daného počtu nepohybujících se objektů do indexu je shrnuta v následující tabulce 6.7 a v grafech na obrázku 6.4.

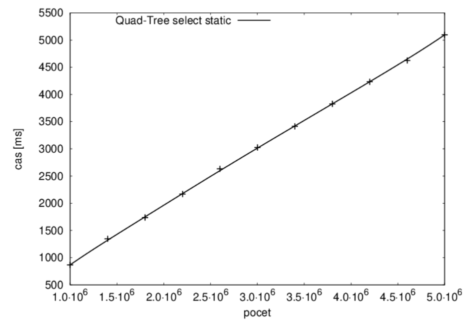
Z tabulky je patrná převaha R*-Tree nad Quad-Tree, který byl v nejhorším případě 145x pomalejší. Toto zpomalení Quad-Tree je dáno tím, že jeho listový uzel pokrývá značnou

Tabulka 6.7: Tabulka srovnání naměřených časů pro jednotlivé indexovací struktury operace vyhledání ve statických datech.

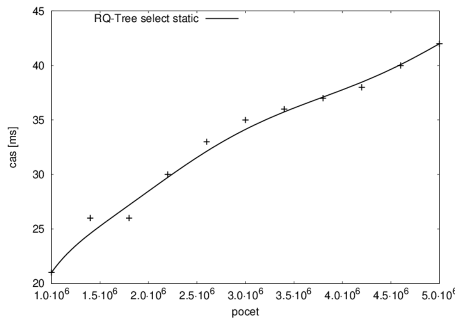
Počet objektů	Čas R*-Tree [ms]	Čas Quad-Tree [ms]	Čas RQ-Tree [ms]
1000000	23	867	21
1400000	24	1346	26
1800000	26	1738	26
2200000	26	2169	30
2600000	29	2632	33
3000000	30	3024	35
3400000	31	3413	36
3800000	33	3826	37
4200000	34	4233	38
4600000	35	4624	40
5000000	35	5096	42



(a) R^*-Tree.



(b) Quad-Tree.



(c) RQ-Tree.

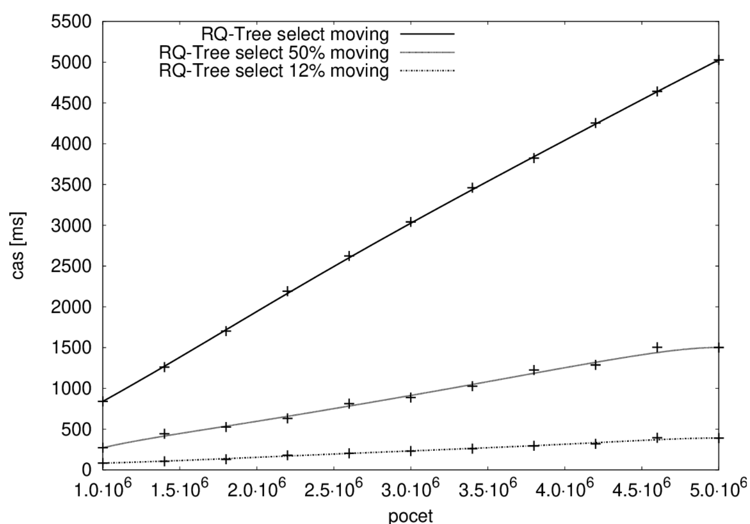
Obrázek 6.4: Grafy časové složitosti pro operaci vyhledání jednotlivých indexů.

část indexovaného prostoru a obsahuje tak velké množství objektů, které je nutno všechny prozkoumat a ověřit zda patří do prohledávané oblasti. Naprostým „neštěstím“ pro Quad-Tree je, pokud prohledávaná oblast protíná všechny čtyři listy.

Následující tabulka 6.8 a graf na obrázku 6.5 shrnuje změnu rychlosti operace hledání pro QR-Tree při indexaci dat s různým poměrem pohybujících se a statických objektů.

Tabulka 6.8: Změna naměřených časů při hledání v objektech různého typu v RQ-Tree.

P. objektů	Čas[ms] 100% pohyblivých	Čas[ms] 50% pohyblivých	Čas[ms] 12% pohyblivých
1000000	840	272	83
1400000	1260	443	105
1800000	1703	526	130
2200000	2192	631	179
2600000	2623	812	204
3000000	3041	887	231
3400000	3460	1026	261
3800000	3823	1226	295
4200000	4255	1287	319
4600000	4642	1505	394
5000000	5028	1501	390



Obrázek 6.5: *Změna časové složitosti při hledání v objektech různého typu v RQ-Tree.*

Z grafů a tabulek lze snadno vyčíst že RQ-Tree pro množiny objektů s obsahem 12% pohybujících se objektů podává velice dobré výsledky, které jsou přibližne 3,9x - 9,2x horší než pro čistě statická data, což je způsobeno nutností projít všechny listové uzly Quad-Tree, které pokrývají oblast všech dosažených listových uzlů v R^*-Tree, ale z grafu je patrné, že nárůst času s přibývajícím počtem indexovaných objektů není nikterak strmý.

Taktéž velice uspokojivých výsledků dosáhl test s obsahem 50% pohyblivých objektů, kde je hodnota naměřeného času nižší než teoreticky očekávaná hodnota (pokud při objemu 12% pohyblivých objektů je dosažen čas 83ms, pak při 50% objemu pohybujících se objektů se dá očekávat přibližně 345ms) v průměru o 17,5% na testovaných zaplněních indexu. Tento náskok oproti teoretickému odhadu je dán tím, že objekty jsou rozloženy na půl mezi oběma indexovacími stromy. Díky tomu jsou listové uzly Quad-Tree zaplněny jen z poloviny což

spolu s velice rychlým vyhledáváním v R^*-Tree části indexu poskytuje zmíněný náskok. Ten však bude díky lineárně se zvyšující složitosti vyhledávání v Quad-Tree části při dosažení jisté limitní hranice dorovnán a za touto hranicí bude index již vykazovat horší poměrné hodnoty.

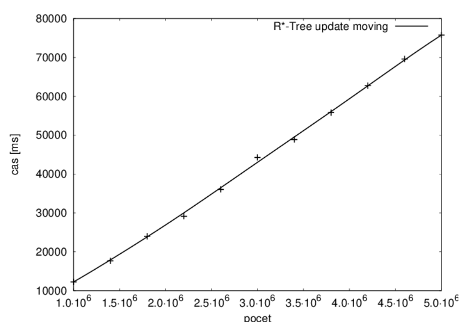
Testování na vzorku pouze pohybujících se objektů odpovídá přibližně stejným časům jako pro samotný Quad-Tree, nedochází zde k žádným benefitům z R^*-Tree části indexu.

6.4.3 Výkon operace aktualizace dat

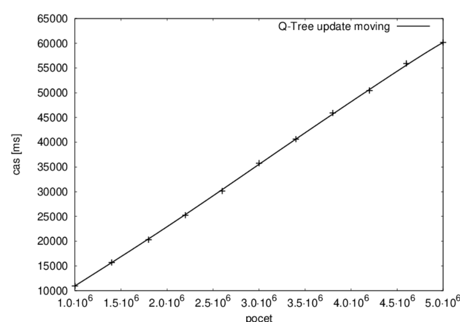
Výkon operace aktualizace jsem testoval na vzorku pouze pohybujících se objektů pro všechny typy indexů provedením 100 aktualizačních dotazů nad každým objektem. Testy byly opět provedeny 5x a výsledky zprůměrovány. Naměřené hodnoty shrnuje tabulka 6.11 a graficky reprezentují grafy na obrázku 6.6.

Tabulka 6.9: Tabulka srovnání naměřených časů pro jednotlivé indexovací struktury operace vyhledání ve statických datech.

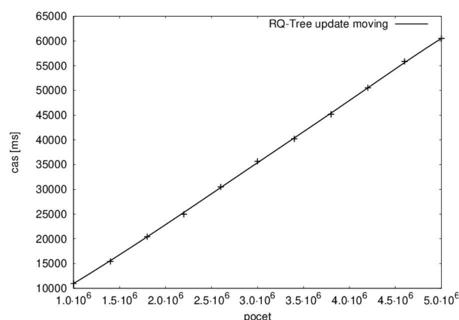
Počet objektů	Čas R*-Tree [ms]	Čas Quad-Tree [ms]	Čas RQ-Tree [ms]
1000000	12235	10944	10968
1400000	17657	15691	15423
1800000	23979	20295	20432
2200000	29143	25254	24961
2600000	36044	30143	30500
3000000	44275	35766	35664
3400000	48879	40632	40221
3800000	55801	45922	45191
4200000	62758	50466	50545
4600000	69607	55928	55858
5000000	75764	60184	60522



(a) R^*-Tree.



(b) Quad-Tree.



(c) RQ-Tree.

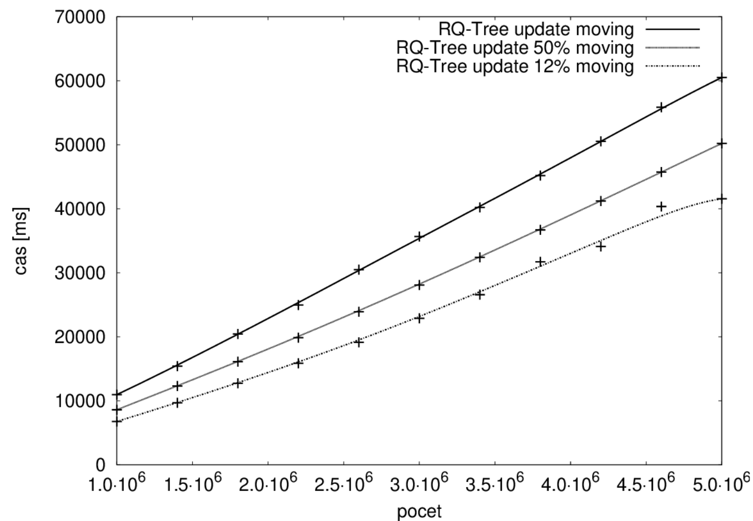
Obrázek 6.6: Grafy časové složitosti pro operaci aktualizace pohyblivých objektů jednotlivých indexů.

V testech výkonnosti aktualizace pohyblivých dat se ukázal jako nejrychlejší Quad-Tree, což odpovídá teoretickým předpokladům. R*-Tree vykazoval v tomto testu přibližně o 26% horší časy. Kombinace RQ-Tree vykazovala srovnatelné výsledky s Quad-Tree zvýšené o drobnou hodnotu, která je spojená s režií na spojení R*-Tree a Quad-Tree.

Následující test je srovnání časů pro RQ-Tree s různým zastoupením pohybujících se objektů. Naměřené hodnoty shrnuje tabulka 6.12 a graficky znázorňuje obrázek 6.7

Tabulka 6.10: Změna naměřených časů při aktualizaci objektů různých typů v RQ-Tree.

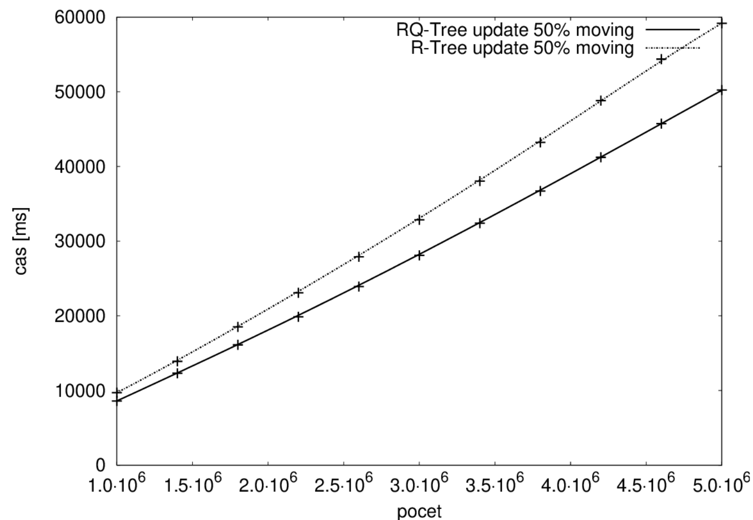
P. objektů	Čas[ms] 100% pohyblivých	Čas[ms] 50% pohyblivých	Čas[ms] 12% pohyblivých
1000000	10968	8600	6769
1400000	15423	12314	9680
1800000	20432	16110	12751
2200000	24961	19866	15855
2600000	30500	23919	19134
3000000	35664	28089	22891
3400000	40221	32410	26576
3800000	45191	36705	31717
4200000	50545	41223	34107
4600000	55858	45734	40356
5000000	60522	50222	41559



Obrázek 6.7: *Změna časové složitosti při aktualizaci objektů různých typů v RQ-Tree.*

Jak vyplývá z tabulky 6.12 a grafů na obrázku 6.7 operace aktualizace je i v RQ-Tree problematická a čas potřebný ke zpracování je úměrný počtu pohybujících se objektů obsažených v indexu.

Srovnání výkonnosti R*-Tree a RQ-Tree je znázorněno grafem na obrázku 6.8.



Obrázek 6.8: *Porovnání výkonnosti RQ-Tree a R*-Tree pro 50% pohybujících se objektů.*

Je zde vidět jistý benefit, který má RQ-Tree oproti R*-Tree, ale který se naplno projeví až u kolekcí s velým počtem indexovaných objektů.

6.5 Testování výkonnosti na reálných datech

Do všech implementovaných indexů byla načtena data z aplikace MaNGOS (všechna NPC na mapě 571). Celkem bylo importováno 63559 objektů jejichž typ odpovídal analýze provedené v kapitole 4.2. Pro testování byla poupravena nastavení jednotlivých indexů, která

shrnují následující tabulky 6.11 a 6.12

Tabulka 6.11: Nastavení parametrů testovaného R^*-Tree pro reálná data.

Přesnost datového typu souřadnic	float
Maximální kapacita listu	100
Faktor vynuceného znovuvložení	30%
Poměr dělení listu	40%:60%

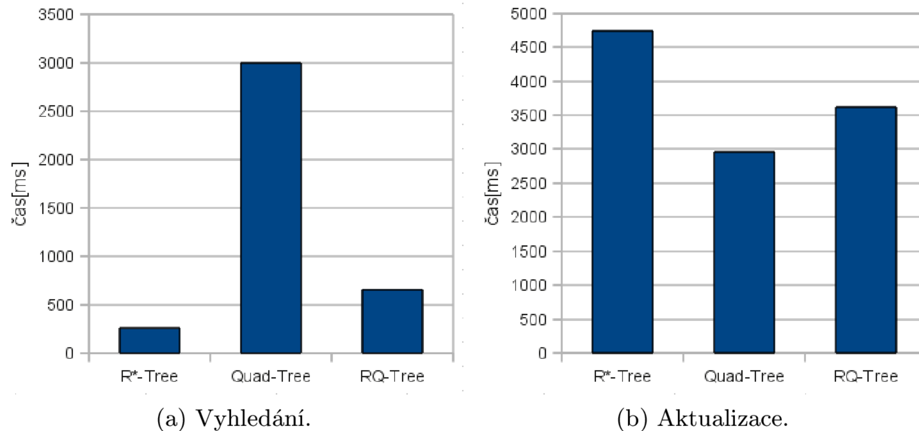
Tabulka 6.12: Nastavení parametru testovaného Quad-Tree pro reálná.

Přesnost datového typu souřadnic	float
Maximální hloubka stromu	4
Hranice indexovaného prostoru	$< -17k, -17k, -17k >$ ... $< 17k, 17k, 17k >$

Nad naplněnými indexovacími strukturami byly provedeny operace vyhledání (měřeno pro 10000 operací) a operace aktualizace (měřeno pro 1000 operací). Všechny testy byly spuštěny 5x a výsledky zprůměrovány. Naměřená data shrnuje následující tabulka 6.13 a graf na obrázku 6.9.

Tabulka 6.13: Naměřené časy pro jednotlivé indexy a operace nad reálnými daty.

Operace	Čas[ms] R^*-Tree	Čas[ms] Quad-Tree	Čas[ms] RQ-Tree
Vyhledání	256 (100%)	2996 (1170%)	657 (256%)
Aktualizace	4746 (100%)	2955 (62%)	3623 (76%)



(a) Vyhledání. (b) Aktualizace.

Obrázek 6.9: Porovnání naměřených časů pro operaci vyhledání v reálných datech.

Z naměřených hodnot je zřejmá dominance R^*-Tree ve vyhledávacích operacích a Quad-Tree v aktualizačních operacích. RQ-Tree kombinace se ukázala jako výkonnostní kompromis mezi uvedenými indexovacími strukturami. Nárůst času, potřebného pro provedení 10000 vyhledávacích operací, u RQ-Tree v porovnání s R*-Tree je přibližně na 256%. Tento nárůst je způsoben režií potřebnou k linkování R*-Tree listů na Quad-Tree listy. Pokles

potřebného času na 76% R*-Tree u operací vyhledávání odpovídá teoretickým předpokladů při průměrném počtu 22% pohybujících se objektů.

Kapitola 7

Závěr

V práci jsem vymezil pojem indexace a prezentoval existující indexovací algoritmy pro objekty v 3D prostoru a to jak pro statické objekty, tak pro data obsahující pohyblivé objekty. Provedl jsem analýzu prostorových dat využívaných v aplikaci MaNGOS.

Na základě této analýzy jsem navrhl nové indexovací řešení RQ-Tree, jehož návrh je detailně popsán v kapitolách 5.1, 5.3 a 5.4. Navržené řešení jsem implementoval společně s indexovacími strukturami R*-Tree a Quad-Tree. Implementované indexy využívají šablonového návrhu, což umožňuje jejich snadnou adaptaci pro indexování různých objektů s různými požadavky na přesnost souřadného systému.

Pro dosažení co největší generičnosti řešení, jsem dotazovací systém nad indexy postavil na bázi univerzálních visitorů, které umožňují při vykonávání dotazu udržovat a aktualizovat svůj vnitřní stav a dovolují tak tvořit složitější dynamické dotazy.

Všechny implementované indexovací struktury jsem podrobil detailnímu testování jak na syntetických testech, využívajících uniformě rozložené náhodně generované objekty v prostoru, tak na testech vycházejících z reálných dat aplikace MaNGOS.

Testy potvrdily teoretické předpoklady. Na poli prostorových vyhledávacích dotazů se projevila naprostá dominance R*-Tree, který však v testech aktualizací pohyblivých objektů nevykazoval tak dobré výsledky jako Quad-Tree, který byl přibližně o 25% rychlejší.

Hybridní struktura RQ-Tree vykazovala výsledky kombinující vlastnosti obou uvedených struktur. Její výkonnost je však velice závislá jak na poměru typů jednotlivých objektů indexovaných strukturou, tak na parametrech určující maximální kapacitu listu pro R*-Tree část a maximální hloubku stromu pro Quad-Tree část.

Výsledky testů odpovídají návrhu struktury, která je primárně určena pro statická data s malým procentem pohybujících se objektů. Poskytuje velice dobrý vyhledávací výkon díky R*-Tree části a snižuje čas potřebný k aktualizaci pohybujících se objektů, které jsou indexovány v Quad-Tree části. Může být tedy úspěšně použita v aplikacích pracujících s tímto charakterem dat.

Literatura

[1] Alexandrescu, A.: *Modern C++ design: generic programming and design patterns applied*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001, ISBN 0-201-70431-5.

[2] Andrew W. Moore: An intoductory tutorial on kd-trees. http://www.autonlab.org/autonweb/14665/version/2/part/5/data/moore-tutorial.pdf?bra 1991 [cit. 2010-01-01].

[3] Apostolos N.; Papadopoulos,Yannis Manolopoulos: *Nearest neighbor search: a database perspective*. Springer Science+Bussines Media, Inc., 2005, iSBN 0-387-22963-9.

[4] Dinesh P. Mehta; Sartaj Sahni: *Handbook of data structures and applications*. Chapman & Hall / CRC, 2005, iSBN 1-58488-435-5.

[5] Hemant M. Kakde: Range Searching using Kd Tree. www.cs.fsu.edu/ lifeifei/cis5930/kdtree.pdf, 2005-08-25 [cit. 2010-01-01].

[6] I. Kamel; C. Faloutsos: Hilbert R-tree: An improved R-tree using fractals. In Proc. of VLDB Conf. Santiago, Chile, 1994 [cit. 2009-12-27], s. 500–509, also available as Tech. Report UMIACS TR 93-12.1 CS-TR-3032.1.

[7] Kevin Sahr; Denis White; A. Jon Kimerling: Geodesic Discrete Global Grid Systems. http://www.sou.edu/cs/sahr/dgg/pubs/gdggs03.pdf, 2003 [cit. 2009-12-27].

[8] Norbert Beckmann; Hans-Peter Kriegel; Ralf Schneider; aj.: The R*-tree: an efficient and robust access method for points and rectangles. In *INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA*, ACM, 1990, s. 322–331.

[9] Sariel Har-Peled: Lecture notes: Approximation Algorithms in Geometry. http://valis.cs.uiuc.edu/šariel/teach/notes/aprx/lec/02_quadtree.pdf, 2009 [cit. 2009-12-27].

[10] Wikipedia: Hilbert R-Tree. http://en.wikipedia.org/wiki/Hilbert_R-tree.

[11] Wikipedia: kd-tree. http://en.wikipedia.org/wiki/Kd-tree.

[12] Wikipedia: Quadtree. http://en.wikipedia.org/wiki/Quadtree.

[13] Wikipedia: R-Tree. http://en.wikipedia.org/wiki/R-tree.

[14] Wikipedia: R*-Tree. http://en.wikipedia.org/wiki/R*_tree.

[15] Wikipedia: Separating Axis Theorem.
http://en.wikipedia.org/wiki/Separating_axis_theorem.

[16] Yufei Tao; Dimitris Papadias; Jimeng Sun: The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries.
www.cs.ust.hk/ dimitris/PAPERS/VLDB03-TPR.pdf, [cit. 2009-01-04].

[17] Yuni Xia; Sunil Prabhakar: Q+Rtree: Efficient Indexing for Moving Object Databases [online]. www.cs.purdue.edu/homes/sunil/pub/yuniDASFAA03.pdf, [cit. 2008-01-03].

Dodatek A

Obsah DVD

Přiložené DVD obsahuje:

- Text práce ve formátu pdf v souboru */diplomova_prace.pdf*.
- Zdrojový kód této práce v TEX formátu v adresáři */doc*.
- Zdrojové kódy příspěvku do soutěže EEICT v TEX formátu v adresáři */eeict*.
- Kompletní zdrojové kódy implementovaných indexovacích metod, benchmarků a testů v adresáři */source*.
- Vygenerovaná Doxygen dokumentace ke zdrojovým kódům v adresáři */src/doc*

Dodatek B

Tabulky výčtových typů pro strukturu BenchArg

Tabulka B.1: Popis výčtového typu BenchFlags.

Název	Hodnota	Význam
BF_NONE	0	Test nebude proveden.
BF_QTREE	1	Test bude proveden pouze pro Quad-Tree.
BF_RTREE	2	Test bude proveden pouze pro R*-Tree.
BF_RQTREE	4	Test bude proveden pouze pro hybridní strom.
BF_ALL	7	Test bude proveden pro všechny indexy.

Tabulka B.2: Popis výčtového typu ObjectType.

Název	Hodnota	Význam
OT_NONE	0	Neplatná hodnota pro test.
OT_STATIC	1	Všechny objekty mají statický charackter.
OT_S50M50	2	Polovina objektů je statická, druhá polovina je pohybující se.
OT_S88M12	3	88% objektů je statických zbytek je pohybující se.
OT_MOVING	4	Všechny objekty se pohybují.

Tabulka B.3: Popis výčtového typu BenchType.

Název	Hodnota	Význam
BT_INSERT	0	Testovanou operací je vkládání objektů do indexu.
BT_SELECT	1	Testovanou operací je prostorový dotaz nad objekty.
BT_UPDATE	2	Testovací operací je aktualizace objektů.

Dodatek C

Ukázky delších fragmentů kódu

```
template <class OBJECTS>
class TypeContainerSet
{
    public:
        template <class SPECIFIC_TYPE> void Insert(SPECIFIC_TYPE* object)
        {
            TypeContainerSet::Insert(m_elements, object);
        }

    private:
        ContainerSet<OBJECTS> m_elements;

        // Insert helpers
        template<class SPECIFIC_TYPE> static bool
        Insert(ContainerSet<SPECIFIC_TYPE>& elements, SPECIFIC_TYPE* obj)
        {
            elements._element.insert(obj);
            return true;
        }
        template<class SPECIFIC_TYPE> static bool
        Insert(ContainerSet<TypeNull>& /*elements*/, SPECIFIC_TYPE* /*obj*/)
        {
            return false;
        }
        template<class SPECIFIC_TYPE, class T> static bool
        Insert(ContainerSet<T>& /*elements*/, SPECIFIC_TYPE* /*obj*/)
        {
            return false;
        }
        template<class SPECIFIC_TYPE, class HEAD, class TAIL> static bool
        Insert(ContainerSet< TypeList<HEAD, TAIL> >& elements, SPECIFIC_TYPE*
            obj)
        {
            bool ret = TypeContainerSet::Insert(elements._elements, obj);
            return ret ? ret : TypeContainerSet::Insert(elements.
                _TailElements, obj);
        }
};
```

Obrázek C.1: Šablonová obálka okolo univerzálního kontejneru.