VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

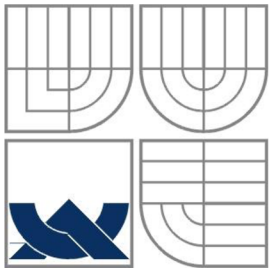
PROSTOROVÉ ROZŠÍŘENÍ OBJEKTOVÉ DATABÁZE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

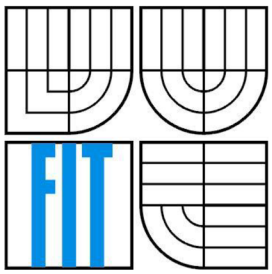
AUTOR PRÁCE
AUTHOR

Bc. ONDŘEJ POLÁCH

BRNO 2012



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

PROSTOROVÉ ROZŠÍŘENÍ OBJEKTOVÉ DATABÁZE

OBJECT DATABASE SPATIAL EXTENSION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. ONDŘEJ POLÁCH

VEDOUCÍ PRÁCE
SUPERVISOR

RNDr. MAREK RYCHLÝ, Ph.D.

BRNO 2012

Abstrakt

Na databázové systémy jsou kladeny stále vyšší požadavky, protože uživatelé potřebují pracovat se stále složitějšími daty. Díky historickému vývoji databázových systémů, se v dnešní době používají hlavně postrelační databázové systémy. Jak již plyne z jejich názvu, jsou postaveny na relačních databázových systémech a rozšiřují je tak, aby byly schopny pracovat se složitějšími daty. Velká většina dnešních prostorových databázových systémů je založena právě na postrelačních databázích. Tato práce se ovšem bude snažit najít spojení mezi objektovými a prostorovými databázemi a nabité poznatky reflektovat do implementace prostorového rozšíření objektové databáze.

Abstract

Still increasing requirements are made on database systems, because users need to work with still more complex data. Because of the historical development of database systems, post-relational database systems are mainly used today. As follows from their name, post-relational database systems are built on relational database systems and expand them so that they are able to work with complex data. The vast majority of today's spatial database systems is based on post-relational databases. However, this work is trying to find a connection between object and spatial databases. The obtained knowledge is reflected in the implementation of the object database spatial extension.

Klíčová slova

Exaktní výpočetní geometrie, objektová databáze, prostorová databáze, souřadnicové systémy, db4o, SQL/MM Spatial, DE-9IM.

Keywords

Exact geometric computation, object database, spatial database, coordinate reference systems, db4o, SQL/MM Spatial, DE-9IM.

Citace

Polách Ondřej: Prostorové rozšíření objektové databáze, diplomová práce, Brno, FIT VUT v Brně, 2012

Prostorové rozšíření objektové databáze

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením RNDr. Marka Rychlého, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Ondřej Polách
23.05.2012

Poděkování

Především bych chtěl poděkovat své rodině za velkou podporu při psaní této práce. Také děkuji panu RNDr. Marku Rychlému, Ph.D. za vedení této práce.

Obsah

Úvod

Tato práce se zabývá prostorovým rozšířením objektové databáze. Nejdříve čtenáře seznámí s pojmy objektová a prostorová databáze. Následně ho stručně provede teorií geometrických výpočtů a poté mu postupně předloží podklady prostorového rozšíření. Na závěr poskytne zhodnocení s návazností na doporučení, která by mohl čtenář, jako potenciální vývojář vlastního prostorového rozšíření, použít.

Relační databáze byly v době svého vzniku evolučním krokem v historii databázových systémů. Jejich model je postaven na jednoduchém matematickém základu, obsahují standardizovaný, všeobecně uznávaný dotazovací jazyk a jsou nejvíce používanými databázovými systémy současnosti. Objektové databáze byly ovšem revolučním přelomem, kdy se sjednotily dva světy, aplikační a databázový. Odstraňovaly některé neduhy relačních databází. V některých případech bylo jejich použití jednodušší a přímočařejší. Bohužel trpí určitými nedostatky, které brzdí jejich zvýšení konkurenceschopnosti k relačním databázím. Nejsou například postaveny na matematickém modelu, neexistují uznávané standardy apod. I přes to si k nim v dnešní době hledá cestu stále více vývojářů, kteří si uvědomují jejich potenciál, na který v určitých situacích relační databáze prostě nestačí.

Prostorové databáze spadají do kategorie postrelačních databází. Postrelační databázový systém je databázový systém, který si díky novým požadavkům, na něj kladoucích, už nevystačí pouze s relačním schématem, protože zpracování dat by bylo neefektivní, ba snad nemožné. Jak je vidět, prostorové databáze jsou více svázány s relačními databázemi, než s objektovými databázemi, což plyne z již zmíněné historie vývoje databázových systémů. Tato práce se ovšem bude snažit najít spojení mezi objektovými a prostorovými databázemi a nabité poznatky reflektovat do implementace prostorového rozšíření objektové databáze.

Nejpočetnějšími klienty prostorových databázových systémů jsou bezpochyby geografické informační systémy (GIS). Z pohledu GIS se dají prostorová data modelovat buď pomocí vrstev v rastru nebo vektorově. Přičemž každý model je vhodný pro různé aplikace. Pouze pro představu, výškový model je velmi vhodné modelovat pomocí rastru a naopak analýzu sítí je vhodné provádět na vektorovém modelu. Další výklad o GIS je bohužel nad rámec této práce. Proto bych rád případného zájemce odkázal na zdroj [1]. Nutno podotknout, že se pro rozšíření předpokládá pouze vektorový model.

První kapitola čtenáře seznámí s pojmem objektová databáze. Popisuje také skupinu Object Data Management Group (ODMG), která se zabývá specifikací a standardizací objektových databází. Tato kapitola také obsahuje popis hlavních rysů a konceptů objektové databáze db4o a shrnuje důvody jejího použití pro implementaci prostorového rozšíření.

Ve druhé kapitole je čtenář seznámen s pojmem prostorová databáze a jejími rysy. Obsahuje také stručný popis rodiny algoritmů R-Tree a úvod do souřadnicových systémů.

Třetí kapitola čtenáře uvádí do problému robustnosti geometrických výpočtů. Seznamuje jej s exaktní výpočetní geometrii, která je v posledních dvou desítkách let hlavním směrem řešením tohoto problému.

Další, v pořadí již čtvrtá kapitola, se začíná zabývat vývojem prostorového rozšíření. Konkrétně je zde popsán návrh architektury. V návaznosti na tuto kapitolu, popisuje pátá kapitola geometrickou knihovnu spolu s podstatnými implementačními aspekty. Následně je v šesté kapitole popsána integrační vrstva, kde je čtenář seznámen s perzistencí a dotazováním prostorových dat.

V sedmé a osmé kapitole je stručně popsáno testování a použití rozšíření. Čtenář je seznámen s postupem, jakým se rozšíření testovalo a je zde také popsáno jeho použití.

V závěru této práce je čtenáři předloženo zhodnocení, náměty na další vývoj a případné doporučení, pokud by měl v úmyslu implementovat vlastní prostorové rozšíření.

I. Teoretický úvod

1 Objektová databáze

Objektová databáze je databázový systém, který poskytuje perzistentní úložiště pro objekty používané v objektově-orientovaných programovacích jazycích. Aplikační i databázová vrstva používají stejný datový model. Díky tomu, objektové databáze zabraňují problému přizpůsobení (impedance mismatch), který je nevýhodou relačních databází [2].

Problém přizpůsobení obecně vzniká tam, kde jsou použity rozdílné modely na aplikační a databázové vrstvě. V případě spojení objektového a relačního světa, musí být objektový model určitým způsobem transformován na relační. Existují aplikační rámce, které tuto transformaci automatizují a snaží se jí tak odstínit od vývojáře. Doslovný překlad označení těchto aplikačních rámců by zněl objektově-relační mapovače, nebo-li Object-Relational Mappers (ORM). Obecně do architektury systému přidávají mezivrstvu, která provádí mapování mezi objektovým a relačním modelem. Díky tomu ORM snižují výkonnost aplikace pro komplexní struktury objektů. Mezi ORM například patří Entity Framework, Hibernate a jiné [3].

1.1 Stručná historie

První generace objektových databází se objevila roku 1986 se systémem G-Base. Druhá generace se objevila o tři roky později, v roce 1989, příchodem systému Ontos. V tomto roce, pan Malcolm Atkinson, vytvořil první manifest objektově-orientovaných databází [4]. Manifest byl výsledkem úsilí definovat objektové databáze. Definuje 13 povinných vlastností, které vycházejí jak z databázových systémů, tak také z objektově-orientovaných systémů. Dále definuje 5 vlastností, které jsou volitelné a nakonec poskytuje několik vlastností, jejichž použití je na zvážení. Jejich detailní popis je mimo rozsah této práce, proto bych rád čtenáře, který by se chtěl více seznámit s tímto manifestem, odkázal na zdroj [4], kde se manifest nachází v elektronické podobě. Třetí generace objektových databází přišla v roce 1990 se systémy jako Orion/Itasca, O_2 a jinými. Jako reakce na nedostatečnou standardizaci ukládání objektů v databázích, byla roku 1991 založena skupina Object Data Management Group, která v roce 1993 vydala první verzi standardu. V roce 1997 vydala druhou verzi tohoto standardu a její úsilí zatím vyvrcholilo v roce 1999, kdy byla vydána zatím poslední, třetí, verze standardu [3]. V době psaní této práce, plánovala skupina ODMG čtvrtou verzi standardu.

1.2 ODMG 3.0

Členové skupiny Object Data Management Group (ODMG) pracují na specifikacích pro podporu perzistence dat objektově-orientovaných programovacích jazyků. Na začátek této podkapitoly nutno podotknout, že standard ODMG není skupinou dodavatelů zcela respektován. Skupinu tvoří všichni hlavní dodavatelé objektových databází. Specifikace se zaměřuje na dva typy produktů. Prvním z nich jsou Object Database Management Systems (ODBMSs), což jsou databázové systémy, které integrují schopnosti databázových i objektově-orientovaných systémů. ODBMS tedy rozšiřují objektově-orientované programovací jazyky o perzistenci dat, řízení konkurence, obnovení dat, dotazování a jiné databázové schopnosti. Druhým produktem jsou Object-to-Database Mappings (ODMs), což jsou systémy, které integrují relační, nebo jiné „neobjektové“ databáze se schopnostmi objektově-orientovaných programovacích jazyků, nebo-li snažící se řešit již zmíněný problém přizpůsobení. Jedná se vlastně o již zmíněné ORM produkty [5].

Před založením ODMG chyběly standardy pro ukládání objektů do databází, což mělo velký dopad na přenositelnost aplikací mezi databázovými systémy. Díky standardům ODMG mohou

vývojáři použít standardizované rozhraní pro ukládání objektů a zvýšit tak přenositelnost svých aplikací mezi databázovými systémy.

Standard ODMG verze 3.0 se skládá z několika hlavních komponent. Jsou jimi objektový model, jazyk pro specifikaci objektů, objektový dotazovací jazyk a vazby na programovací jazyky C++, Smalltalk a Java.

Objektový model vychází z modelu skupiny Object Management Group (OMG), který vznikl jako reakce na požadavek na standardizaci objektového modelu pro objektově-orientované systémy. OMG se zabývá veškerými aspekty objektově-orientovaného programování. Patří mezi ně například objektová analýza a návrh, jazyk Unified Modeling Language (UML), Model Driven Architecture (MDA) a mnoho jiných. Čtenáře, který by se chtěl o OMG dozvědět více, bych rád odkázal na [6].

Specifikace jazyka pro popis objektů se dělí do dvou kategorií. A sice, Object Definition Language (ODL) a Object Interchange Format (OIF). ODL je specifikace jazyka, který se používá pro definici typů objektů, která je v souladu s objektovým modelem. OIF je specifikací jazyka používaného pro ukládání a načítání aktuálního stavu databázového systému.

ODMG definuje specifikaci pro deklarativní objektový dotazovací jazyk, chcete-li Object Query Language (OQL). OQL vychází ze standardu Structured Query Language (SQL) a rozšiřuje ho o podporu dotazů na objektový model.

Z vazby na jazyky C++, Smalltalk a Java plyne, že je možné pracovat se stejnými databázemi pomocí těchto jazyků. Tedy, dodržením tohoto standardu lze dosáhnout určité míry přenositelnosti aplikace.

Detailní popis, ať už samotného objektového modelu nebo ODL, či OQL je opět nad rámec této práce, proto bych rád čtenáře odkázal na zdroj samotného standardu ODMG 3.0 [5].

1.3 Versant db4o

Objektová databáze db4o je open-source produktem, který je sponzorován, vyvíjen a distribuován společností Versant Corporation. Nabízí širokou komunitu vývojářů, kvalitní dokumentaci a duální licenci, díky níž lze db4o využít jak volně, tak i v komerční sféře. Aplikační rozhraní db4o je dostupné pro platformy Java a Microsoft .NET. Databázi db4o lze použít buď lokálně nebo v architektuře klient/server. Následující podkapitoly čtenáře seznamují se základními koncepty db4o. Poslední podkapitola uvádí důvody výběru db4o pro tuto práci.

1.3.1 Transakce

Aplikační rozhraní db4o obsahuje kontejner objektů, kterým se přistupuje k samotné databázi. Každý kontejner objektů vlastní svoji transakci, která zajišťuje atomičnost, konzistenci, izolovanost a trvalost (ACID). Transakce samozřejmě podporují operace potvrzení a navrácení. Databáze db4o používá úroveň izolace read-committed. To znamená, že jedna transakce nemůže pracovat s nepotvrzenými daty druhé transakce [7].

1.3.2 Identita objektů

Databáze db4o spravuje objekty pomocí jejich identit. Za tímto účelem používá vnitřní identifikátory objektů typu integer nebo *Universally Unique Identifier* (UUID). Je-li to nutné, lze použít také vlastní identifikátory na úrovni vlastností tříd, podobně jako primární klíč u relačního schématu. To je velmi vhodné v rozpojeném prostředí. Kontejner objektů udržuje mapování mezi objekty v paměti a v databázi tak, aby bylo zajištěno, že každý uložený objekt má pouze jednu paměťovou reprezentaci. Toto platí pouze v kontextu jednoho kontejneru a nelze to předpokládat pro objekty načtené různými kontejnery. S jedním kontejnerem se pracuje obecně v režimu vestavěné

databáze nebo na desktopových aplikacích. S více kontejnery je nutné pracovat v rozpojeném prostředí (disconnected environment), jakým je například webová aplikace [7].

1.3.3 Aktivace

Objekty jsou instanciovány pomocí mechanismu aktivace. Představme si, že máme v databázi uloženu rozsáhlou objektovou strukturu, která je provázána jako strom. Každý objekt ve struktuře má tedy několik následníků a jednoho předchůdce. Struktura obsahuje také kořenový objekt, který chceme nyní získat prostřednictvím dotazu. Tento objekt se tedy vytvoří v paměti. Ovšem, s ním se musí vytvořit všechny objekty, které se nacházejí níže ve stromu. To může způsobit přeplnění paměti a není to tedy určitě efektivní způsob. Jako přijatelné řešení se jeví načtení pouze podmnožiny objektů ze struktury podle určitého kritéria. Daným kritériem je aktivační hloubka. Aktivační hloubku si můžeme představit jako počet referencí od daného objektu dále po struktuře. Reference za tuto aktivační hloubku jsou nastaveny buď na výchozí, nebo na *NULL* hodnotu. To znamená, že referencované objekty za aktivační hloubkou nejsou načteny v paměti. Pro pozdější aktivaci používá db4o slabé reference (*Weak Reference*). Slabé reference tvoří mapovací tabulku, která mapuje adresy na perzistentní identifikátory neaktivních objektů. Zmíněný postup s aktivační hloubkou je použit v db4o. Hodnotu aktivační hloubky lze samozřejmě nastavit. Aktivace probíhá buď implicitně podle této nastavené hodnoty, nebo ji lze provést i explicitně. Pro složitě provázané objektové struktury, kde by byla strategie aktivace příliš složitá, lze s určitou mírou implementace použít transparentní aktivaci [7]. Zmiňovaný koncept aktivace zobrazuje Schéma 1.

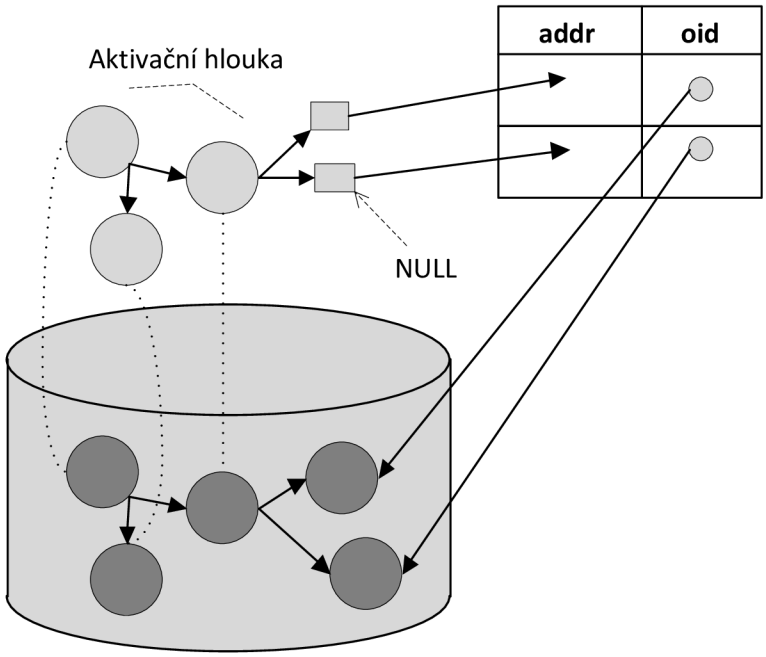


Schéma 1: Koncept aktivace.

1.3.4 Manipulace s objekty

Při vytváření, aktualizování a odstraňování objektů musí db4o řešit několik problémů. Při vytváření a aktualizaci objektů musí db4o nejdříve rozpoznat, zda má být daný objekt vytvořen nebo pouze aktualizován. Tento problém řeší pomocí identit objektů, pomocí nichž se pokouší najít daný objekt v paměti. Je-li objekt v paměti nalezen, znamená to, že by měl být aktualizován. Pokud takový objekt neexistuje, měl by být vytvořen. Následně musí rozhodnout o rozsahu aktualizací.

Použije k tomu hloubku aktualizací, která má výchozí hodnotu rovnu jedné. To znamená, že je aktualizován pouze daný objekt a nemusí se procházet struktura objektů přes reference za účelem hledání objektů, které musí být také aktualizovány. Je-li tedy potřeba provést aktualizaci referencovaných objektů, musí se toto provést buď zvýšením hodnoty aktualizační hloubky nebo samostatným aktualizováním. Při odstraňování objektu je odstraněn daný objekt, ale referencované objekty nikoli. Lze ovšem nastavit kaskádové odstraňování, kdy mohou být odstraněny i referencované objekty. Nejlepším způsobem, jak řešit tyto problémy, je použití transparentní perzistence spolu s transparentní aktivací [7].

1.3.5 Dotazování

Databáze db4o podporuje několik typů dotazovacích jazyků. Jsou jimi Simple Object Database Access (SODA), nativní dotazy a dotazy pomocí příkladu (Query by example, nebo-li QBE). Následující odstavce popisují zmíněné jazyky. U každého je zobrazena ukázka dotazu v jazyce Java, který vrací všechny osoby, které mají *30 let* a jmenují se *Jan* (Zdrojový kód 1, 2, 3).

SODA tvoří dotazovací jazyk nižší úrovně. Všechny dále zmíněné dotazovací jazyky jsou překládány do tohoto jazyka, je-li to možné. Vývojáři ovšem mohou tento jazyk použít také přímo. Pro identifikaci třídních členů, používá SODA textové řetězce. Díky tomu není typově příliš bezpečný. Vývojáři se ale snaží omezit textový základ jazyka na minimum. Psaní dotazu pomocí SODA je náročnější, než je tomu u vyšších jazyků. Je však velmi vhodné pro použití v aplikacích, které potřebují dynamické generování dotazů. SODA provádí dotazy ve dvou fázích. V první fázi vytváří strom identit kandidátů pomocí indexu. Tato fáze je prováděna v dotazovacím procesoru nad indexy. Ve druhé fázi použije všechny identity z první fáze a na objekty dané těmito identitami aplikuje podmínky dotazu. Je-li podmínka regulární výraz, jsou porovnány hodnoty přímo v databázi. Je-li ovšem nutné provést další výpočty, SODA vytvoří instance kandidátních objektů a provede nad nimi dané výpočty. Tato fáze se provádí v SODA dotazovacím procesoru. Poté ještě provádí seřazení výsledné množiny a následně vrací množinu identit všech výsledných objektů [7].

```
Query query = container.query();
query.constrain(Person.class);
query.descend(„age“).constrain(30)
       .and(query.descend(„name“).constrain(„Jan“));
ObjectSet<Object> result = query.execute();
```

Zdrojový kód 1: Ukázka dotazu v jazyce SODA.

Nativní dotazy tvoří základní dotazovací jazyk db4o. Díky tomuto jazyku mohou být dotazy vyjádřeny pomocí prostředků objektově-orientovaného programovacího jazyka a plně tak využít jeho potenciálu. Dotazy jsou plně přístupné typové i syntaktické kontrole, refaktoringu apod. Nejsou již založeny na textovém popisu, jako je tomu například u SQL, OQL, SODA apod. Právě díky těmto vlastnostem se zdají být novým směrem v dotazovacích jazycích. Při použití nativních dotazů lze použít techniku anonymních tříd. Je poskytnuta abstraktní třída nazvaná *Predicate*, která obsahuje abstraktní metodu *match* přebírající každou instanci dané třídy a vracející logickou hodnotu. Tato metoda je implementována specificky pro každý dotaz. Více informací o nativních dotazech se nachází ve zdrojích [7] a [8].

```
List<Person> persons = container.query<Person>(
    new Predicate<Person>() {
            public boolean match(Person person) {
                  return person.getAge() == 30
                        && person.getName().equals(„Jan“);
            }
    });
```

Zdrojový kód 2: Ukázka dotazu v nativním jazyce.

Dotazy pomocí příkladu poskytují jednoduchý dotazovací jazyk. V principu se prochází všechny členy příkladového objektu. Pokud člen nemá výchozí hodnotu, bude použit jako podmínka v dotazu. Jakmile je celý objekt analyzován a jsou vytvořeny podmínky, provede se dotaz jako kterýkoli jiný. Toto rozhraní má ale několik omezení, se kterými je nutné počítat. Pro představu, nelze například použít operátory *and*, *or* apod. Je však velmi vhodné pro začínající uživatele [7].

```
Person example = new Person();
example.setAge(30);
example.setName(„Jan“);
ObjectSet<Person> result = container.queryByExample(example);
```

Zdrojový kód 3: Ukázka dotazu pomocí příkladu.

Výkonnost dotazů zajišťuje indexovací struktura B+-Tree. Indexovány mohou být různé typy, kromě polí a kolekcí [7].

1.3.6 Rozšiřitelnost

Databáze db4o nabízí několik bodů rozšiřitelnosti. Pomocí zpětných volání lze ovlivňovat proces manipulace s objekty při jejich perzistenci. Lze tak například kontrolovat integrační omezení při ukládání objektů do databáze. Díky nízko-úrovňovým rozhraním *TypeHandlers* nebo *Translators* lze implementovat vlastní strategii pro perzistenci dat. Také je možné nakonfigurovat databázi dle vlastních potřeb [7]. Nelze ovšem integrovat vlastní indexační strategii pro specifická data, což je pro tuto práci zásadní nedostatek, který byl bohužel zjištěn až v závěrečných fázích vývoje. V současné době není ani jasné, zda bude někdy tato možnost existovat [9].

1.3.7 Důvody výběru db4o

Při výběru vhodné objektové databáze pro tuto práci bylo stanoveno několik kritérií. Bezpodmínečně musí být zveřejněny zdrojové kódy a musí být dostupná dostatečně podrobná dokumentace. Právě díky splnění zmíněných kritérií byla vybrána objektová databáze db4o. Ve hře byla ještě například objektová databáze ObjectDB, která ale díky jejím licencím nevyhověla.

V čase, kdy se tato práce blížila k závěru se vyskytly různé, více, či méně důležité kritéria. Některé z nich měly být známy již v době, kdy se rozhodovalo o použití databáze, protože by to určitě ovlivnilo její výběr. Jedním z takových kritérií je již zmíněná schopnost integrace vlastní indexační strategie.

2 Prostorová databáze

Prostorová databáze je databázový systém s podporou ukládání a zpracování prostorových dat. Prostorovými daty rozumíme data, které se váží k určitému prostoru, bez ohledu na to, jak velký tento prostor je. Jedná se především o množinu relativně jednoduchých geometrických entit, které mohou modelovat například domy, města, řeky, cesty, pozemky apod. Prostorem můžeme rozumět souřadný systém, který má několik dimenzí. Prostor zapouzdřuje vztahy mezi entitami. V této práci budeme implicitně pracovat s 2-dimenzionálním prostorem [10].

2.1 Reprezentace souřadnic

Pro jednoduchost uvažujme Euklidovský prostor, který je spojitý a každý bod p je definován souřadnicí ve dvou dimenzích: $p = (x, y) \in \mathbb{R}^2$, kde $\mathbb{R}$ označuje množinu reálných čísel. Nad Euklidovským prostorem existuje velké množství užitečných a použitelných geometrických algoritmů, které jsou postaveny nad teoretickou nekonečně přesnou aritmetikou a spojitým prostorem [10], [11].

V počítačových systémech je nemožné zachytit úplnou množinu reálných čísel. Zachycujeme pouze diskrétní prostor. V určitých případech nejsme tedy schopni například reprezentovat průsečík dvou úseček, protože právě tento bod nespadá do daného oboru povolených hodnot. Praktické použití teoreticky korektního algoritmu tedy nemusí být nezbytně vhodné. Ignorování tohoto faktu může v mnoha případech snižovat robustnost, topologickou korektnost, spolehlivost a efektivnost daného geometrického algoritmu.

2.2 Deskriptory

Jako obecný databázový koncept, je deskriptor, nebo-li *realm*, konečná, dynamická a uživatelsky definovaná základní struktura datových typů. V prostorových databázích je deskriptor definován jako planární graf nad diskrétní mřížkou. Jedná se tedy o množinu bodů a nekřížících se úseček, které mají tyto vlastnosti:

- každý samostatný bod je bodem diskrétní mřížky,
- každý koncový bod úsečky a složitějšího útvaru je bodem diskrétní mřížky,
- žádný vnitřní bod úsečky a složitějšího útvaru není zaznamenán v diskrétní mřížce,
- žádné dva složitější útvary nemají žádný společný bod, mimo koncové.

Deskriptory na implementační úrovni řeší problém reprezentace souřadnic. Díky tomu odstiňují prostorové algebry a geometrické algoritmy, které jsou nad nimi postaveny, od tohoto problému [12].

2.3 Algebra

Prostorová algebra definuje prostorové datové typy a operace nad těmito typy. Prostorové datové typy slouží k reprezentaci prostorových dat. Definice prostorových datových typů a operací by měla splňovat několik kritérií. V první řadě, musí formálně definovat prostorové datové typy a operace nad nimi, přičemž musí být zohledněna aritmetika s konečnou přesností. Dále, v systému musí být zahrnuta podpora pro konzistentní popis prostorově souvisejících objektů. V neposlední řadě, definice dat a operací by měla být nezávislá na konkrétním systému řízení báze dat (SŘBD), ale přitom s daným SŘBD úzce spolupracovat [10].

V zásadě existují 3 základní kategorie prostorových operací, které je možné na prostorové datové typy aplikovat:

- *Predikáty*: vstupem jsou hodnoty, z nichž alespoň jedna je nějakého prostorového typu a výsledkem je pravdivostní hodnota. Patří zde například *intersects*, *within*, *overlap* ad.
- *Geometrické relace*: vstupem je hodnota, či množina hodnot nějakého prostorového typu a výsledkem je jedna čí více hodnot prostorového typu, nebo relace, která takové typy obsahuje alespoň v jednom atributu. Zde patří například *union*, *intersect* ad.
- *Výpočetně náročné operace*: Například *buffer*, *convex hull*, *distance*, *area* ad.

Kromě zmíněných operací je nutné, aby byly podporovány i operace prostorové selekce, spojení, aplikace prostorové funkce a další množinové operace. Prostorová selekce spočívá v selekci založené na prostorovém predikátu. Prostorové spojení je spojení založené na predikátu testujícím prostorové atributy. Aplikace prostorové funkce je použití této funkce jako podmínky při selekci [10].

Existuje několik standardů definujících prostorové datové typy a operace. Jedním z nich je standard skupiny Open Geospatial Consortium (OGC) nazvaný Simple Features Specification for SQL (SFS). Podrobnosti o tomto standardu můžete získat na webovém portálu OGC [13]. Od tohoto standardu byl odvozen standard skupiny International Organization for Standardization (ISO) nazvaný SQL Multimedia and Application Packages (SQL/MM): Spatial. V této práci je použit standard SQL/MM Spatial, který bude podrobně popsán v kapitole zabývající se aplikačním rozhraním rozšíření. SFS je použit jako základ knihovny Java Topology Suite (JTS). SQL/MM Spatial je podporován například Oracle Spatial 11g Release 2, ale zdá se, že neexistuje knihovna podobná JTS, která by jej implementovala a dala by se tak použít při implementaci prostorového rozšíření.

2.4 Vztahy

Jedním z důležitých aspektů dané prostorové algebry je korektní určení vztahů mezi objekty. Zpočátku se objevily myšlenky, zda existuje nějaká hranice pro definici vztahu dvou objektů v prostorech různé dimenze. Průkopníkem v této oblasti byl pan Egenhofer [14], který nejdříve studoval plošné objekty bez děr a vnořených objektů. Tento koncept byl dále rozvíjen. Nakonec bylo zjištěno 256 kombinací, z nichž 52 bylo platných. Výčet byl sice dostatečný, ale i tak velmi vysoký. Naštěstí se podařilo najít alternativu s 5 operacemi a 3 operátory na extrakci hranice. Mezi operace patří *dotek (touches)*, *uvnitř (within)*, *přes (crosses)*, *přesah (overlaps)* a *disjunkce (disjoint)*. Tyto operace spolu s operátory umožňují realizovat všech 52 kombinací. Následovalo další rozšíření pro oblasti s dírami, skládané oblasti a další [10]. Reference na některé články ohledně vývoje určování vztahů prostorových objektů lze nalézt ve zdroji [15].

V této práci použiji model pro určení vztahů, který se nazývá Dimensionally Extended Nine-Intersection Model (DE-9IM). Jedná se též o Clementiniho matici. Tento model je použit i ve standardech SQL/MM Spatial a SFS a vychází z dříve zmíněných.

2.5 Dotazovací jazyk

Prostorový dotazovací jazyk rozšiřuje dotazovací jazyk o prostorovou algebru. Musí tedy podporovat prostorové datové typy a musí být schopen vyjádřit prostorové operace. Pro připomenutí, mezi prostorové operace řadíme především prostorovou selekci a spojení, aplikace prostorové funkce a další množinové operace, jako voronoi, fúze a překrytí. Měl by také splnit několik požadavků, které ve zdroji [16] popsal pan Egenhofer. Z těchto požadavků vyplývá vhodnost pro použití grafického vstupu a výstupu pro prostorová data. Uveďme nyní obecné příklady prostorových dotazů, bez zápisu v dotazovacím jazyce [10]:

- *Prostorová selekce:* Vyhledej všechny řeky, které protínají dotazovací okno.
- *Prostorové spojení:* Najdi všechny města v blízkosti řek.
- *Aplikace prostorové funkce:* Ke každé řece protékající Moravou vypiš jméno, kudy protéká a délku příslušné části.

Pro efektivní dotazování prostorových dat je nutné použít vhodnou přístupovou metodu s využitím indexovací strategie.

Pro uvedení do problematiky indexování prostorových dat uvažujme 1-dimenzionální prostor P. Dále mějme hodnotu $x \epsilon P$. Jelikož lze nad daty prostoru P definovat jednoznačné uspořádání, potom lze jednoznačně určit hodnotu předchůdce a následníka hodnoty x jednoduše jako, $x - 1$ a $x + 1$. Díky této vlastnosti lze pro indexování 1-dimenzionálního prostoru použít známé algoritmy jako B-Tree aj. U prostorových dat ovšem mluvíme o 2 a více dimenzích, pro které zmíněná vlastnost neplatí a je nutné tento problém řešit. Jedním z možných řešení je transformace více-dimenzionálního prostoru do 1-dimenzionálního. Toto řešení je však nevhodné, ba někdy dokonce nerealizovatelné. Následující dvě řešení vychází z indexování bodových dat. První je založené na stromových strukturách. Patří zde algoritmy jako K-D-Tree, BSP-Tree a Quad-Tree. Druhé využívá hashování. Zde patří algoritmy jako Grid File a jeho varianty. Indexování bodových dat patří sice mezi základní požadavky indexačního algoritmu, nevystačíme si s ním však u indexování více-dimenzionálních dat. Metody indexování takových dat se rozdělují obecně do 4 skupin. Transformace, překrývání, ořezávání a hashování. Metoda transformace je postavena na mapování objektu do více-dimenzionálního prostoru, kde se stává bezrozměrným. Díky zásadním nevýhodám není transformace vhodná k použití. U překrývání vzniká více vyhledávacích cest díky tomu, že se indexační struktury překrývají. Výhodou ale je, že je celý objekt obsažen pouze v jedné indexační struktuře. Metoda ořezávání zakazuje překrývání indexačních struktur. Provádí to rozdělením objektu podle hranic takových struktur na více pod-objektů. Při vyhledání objektu je tedy nutná jeho rekonstrukce z pod-objektů. Metoda založená na hashování vychází z hashování pro bodová data. Mezi zástupce patří například Multi-Layer Grid File [10]. Ještě bych rád podotknul, že zmíněné algoritmy jsou pouze základním výčtem a nejedná se tedy o úplnou množinu algoritmů.

Co se týče použití indexovací strategie v rozšíření, je nutné připomenout, že objektová databáze db4o neobsahuje rozšiřovací bod pro vlastní index. I přes to, následující podkapitola seznamuje čtenáře se základními algoritmy pro indexování prostorových dat.

2.5.1 Rodina algoritmů R-Tree

Tato podkapitola čtenáře seznamuje s rodinou algoritmů R-Tree. Zaměří se hlavně na samotný R-Tree. Poté okrajově zmíní principy R+-Tree a R*-Tree.

Algoritmus R-Tree byl definován panem Guttmanem v roce 1984. Je postaven na metodě překrývání popsané výše. Každý listový uzel obsahuje záznamy ve formátu *(MBR, object-id)*, kde *object-id* je ukazatel na datový objekt a *MBR* (Minimum Bounding Rectangle) je minimální ohraničující obdélník tohoto objektu. Datový objekt může být typu bod, křivka nebo polygon. Každý vnitřní uzel obsahuje záznamy ve formátu *(MBR, child-pointer)*, kde *child-pointer* je ukazatel na svého potomka a *MBR* je minimální ohraničující obdélník potomka. Každý uzel může obsahovat pouze určitý počet záznamů. Tento počet se nazývá branching factor *M* (faktor větvení *M*). Ten musí být vhodně zvolen, aby odpovídal velikosti stránky na disku, a to z rozsahu mezi hodnotou m a M, kde $m \leq \lceil M/2 \rceil$ je minimální počet záznamů pro uzel. Při vkládání nového objektu, je tento přidán do listu stromu. Uzly, které přetečou M, jsou rozděleny a toto rozdělení se šíří stromem směrem ke kořeni. Při odstranění objektu může dojít k situaci, že uzel podteče m. Nastane-li tato situace, je daný uzel odstraněn a záznamy, které se v něm nacházejí jsou opět vloženy. Opět se toto šíří stromem směrem ke kořeni. Při změně objektu je změněn i MBR. Potom všechny jeho záznamy musí být

odstraněny, aktualizovány a opět vloženy. Vyhledávání v R-Tree probíhá tak, že se postupuje od kořene stromu k listům a na každém vnitřním uzlu se rozhoduje o další cestě stromem. Díky překrývání může být potřeba prohledat více cest. Aby bylo vyhledávání maximálně efektivní, je nutné držet počet překrývajících se uzlů na minimu. Toho je dosaženo již zmíněnou rozdělovací strategií. Jak je vidět, operace vkládání, odstranění a aktualizování R-Tree jsou náročnější a pomalejší. Ovšem vyhledávání je díky tomu rychlé. Čtenáře, který by měl zájem dozvědět se více detailů o R-Tree bych rád odkázal na zdroje [17], [18] a [19].

Metoda ořezávání, která byla zmíněná výše, je použita v algoritmu R+-Tree. Další algoritmus z rodiny R-Tree je algoritmus nazvaný R*-Tree. Ten používá jinou metodu vkládání, kdy při přetečení faktoru *M* neprovádí ihned rozdělení uzlu, ale odstraní několik záznamů z daného uzlu a tyto znovu vloží do stromu. Více se o těchto algoritmech můžete dozvědět ve zdroji [18].

2.6 Souřadnicové systémy

Souřadnicový systém je obecně prostředek pro vyjádření polohy bodu v nějakém prostoru [1]. Použitý prostor závisí hlavně na samotné aplikaci. Jak již bylo zmíněno v úvodu této práce, velkými klienty prostorových databázových systémů jsou GIS. Z pohledu GIS se mluví o geografickém prostoru. V nejjednodušším případě lze prostorem uvažovat také již zmíněný Euklidovský prostor. Existuje velké množství souřadnicových systémů. Uvažujeme-li geografický prostor, lze rozlišit dva typy souřadnicových systémů. Globální a lokální.

Globální souřadnicové systémy se snaží postihnout celý geografický povrch Země. Nejedná se o pravoúhlý systém a nelze tedy použít jednoduchý Euklidovský vzorec pro výpočet vzdálenosti. Jejich výhodou je univerzálnost v popisu. Ovšem nevýhodou může být malá přesnost pro lokální území. Patří zde WGS-84 (World Geodetic System) a UTM (Universal Transverse Mercator). WGS-84 je založen na zeměpisné délce a šířce. Na WGS-84 je postaven například globální polohovací systém GPS (Global Positioning System).

Lokální souřadnicové systémy vznikly velmi specifickou transformací územně platného náhradního elipsoidu na plochu a díky tomu lze uplatnit jednoduchý vzorec pro výpočet vzdálenosti. Pro území České republiky a Slovenska se používá S-JTSK (Souřadný systém Jednotné trigonometrické sítě katastrální).

Některé aplikace vyžadují převod mezi určitými souřadnicovými systémy. Příkladem může být převod z globálního WGS-84 do lokálního S-JTSK. Tyto transformace jsou velmi specifické pro každý souřadnicový systém. Podrobnější výklad o souřadnicových systémech je nad rámec této práce. Proto bych případného zájemce rád odkázal na zdroj [1].

3 Robustnost

Algoritmy výpočetní geometrie jsou běžně navrženy pro reálná čísla, která jsou při implementaci nahrazena čísly s omezenou přesností. Toto nahrazení způsobuje numerické chyby ve výpočtech, což může vést k nerobustnímu chování. Důvod je prostý. Současná, a teoreticky ani budoucí výpočetní technika nedokáže reprezentovat celou množinu reálných čísel.

V literatuře existují různé přístupy řešící nerobustnost v geometrických výpočtech. Tyto mohou být klasifikovány do dvou hlavních skupin. A sice, aritmetické nebo geometrické.

Aritmetické řešení se dívá na numerické chyby jako na chyby vyplývající z aritmetiky s omezenou přesností. Typickým příkladem takové aritmetiky jsou čísla v plovoucí řádové čárce, které jsou používané v běžných procesorech a práce s nimi je vysoce optimalizovaná. Bohužel jejich použití v případech, kde je nutná robustnost výpočtů, je nevhodná. A to díky zaokrouhlovacím chybám. Proto se v těchto případech používají čísla s neomezenou přesností. Bohužel, procesory nejsou pro tento typ čísel optimalizovány. Z toho vyplývá hlavní nedostatek tohoto řešení, a sice výkonnost [20].

Geometrické řešení garantuje, že jistá geometrická vlastnost je algoritmem kontrolována a dodržována. Například hlídání, že planární graf je opravdu planární apod. Tyto řešení lze dělit do dalších podskupin lišících se principem. Patří zde například řešení postavené na principu překreslování, které se jmenuje *finite resolution geometry* [21], a mnoho dalších [20].

3.1 Exaktní výpočetní geometrie

V posledních dvou desítkách let se zdá být hlavním směrem řešení nerobustnosti geometrických výpočtů exaktní výpočetní geometrie (EGC). Jedná se o aritmetické řešení. EGC rozděluje každý geometrický výpočet do dvou částí, kombinační a numerické. Kombinační část představuje diskrétní vztahy mezi geometrickými objekty a numerická část obsahuje numerickou reprezentaci těchto objektů. Tedy, geometrické algoritmy charakterizují kombinační struktury numerickým výpočtem diskrétních vztahů mezi geometrickými objekty. Nerobustnost vzniká, když numerická chyba ve výpočtu způsobí nesprávný vztah mezi objekty. EGC zajišťuje, že všechny geometrické predikáty jsou vyhodnoceny správně a tím pádem je správně vypočítána i kombinační struktura [20].

EGC je založen na třech klíčových principech, díky kterým je toto řešení úspěšné. Jsou jimi, určování hranice nuly, aproximovaný výpočet výrazu a numerické filtry [20].

Praktickým výsledkem dosavadního výzkumu jsou robustní geometrické knihovny CGAL, CORE Library, LEDA ad. Převážná většina těchto knihoven je implementována v jazycích C/C++ a je dostupná ke stažení.

Téma zabývající se EGC je díky dlouholetému výzkumu velmi obsáhlé. Proto je tato podkapitola jen stručným úvodem do této problematiky a případného zájemce bych proto rád odkázal na následující zdroje. Zdroj [20] obsahuje základní popis EGC. Na něj navazuje zdroj [22], který obsahuje mnohem více detailů. Kvalitním zdrojem informací jsou také články pana Chee Yapa, které jsou dostupné ve zdroji [23]. Vzniká také kniha zabývající se EGC. Ve zdroji [24] se nacházejí návrhy kapitol, které se v této knize objeví. Zdroj je pravidelně aktualizován a zájemce si může kapitoly prohlédnout ještě před samotným vydáním publikace.

II. Vlastní řešení

4 Architektura prostorového rozšíření

Tato kapitola se začíná zabývat samotným prostorovým rozšířením objektové databáze db4o. Konkrétně, je zde stručně popsána architektura rozšíření. Navazující kapitoly poté obsahují detailní popis hlavních částí architektury spolu s návazností na implementaci.

Rozšíření je složeno ze dvou hlavních částí. Geometrické knihovny a integrační vrstvy. Geometrická knihovna zapouzdřuje práci s prostorovými daty a integrační vrstva zajišťuje perzistenci těchto dat v objektové databázi db4o. Nyní budou stručně popsány jednotlivé části architektury od spodních vrstev. Architektura je zobrazena na Diagramu 1.

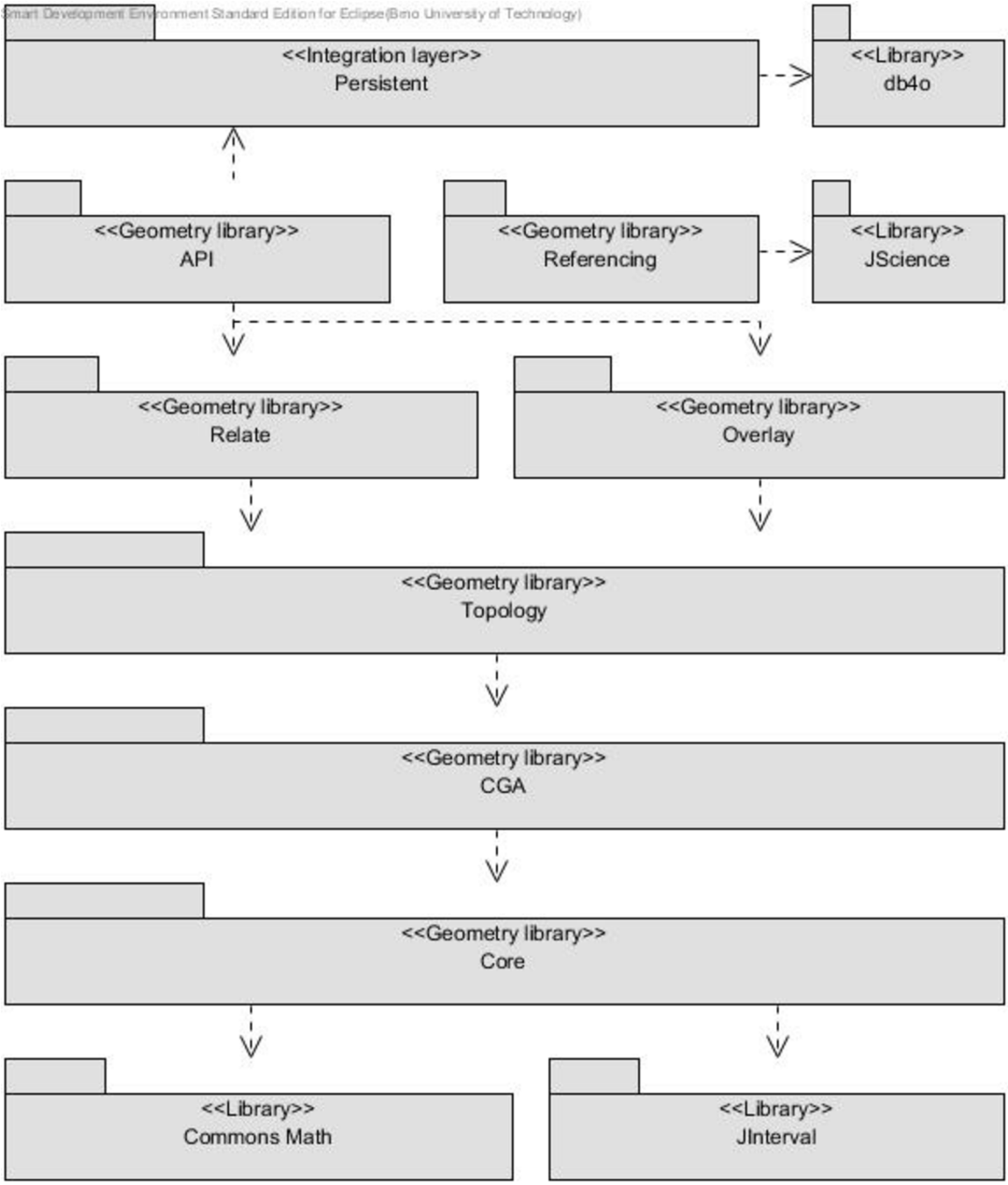


Diagram 1: Architektura prostorového rozšíření db4o.

Vrstvy nesoucí stereotyp *Library* jsou externí knihovny, na kterých je řešení závislé. Ostatní vrstvy jsou součástí vlastního řešení.

Jádro (*Core*) geometrické knihovny je postaveno na robustním výpočtu znaménka determinantu. Tato vrstva využívá externí knihovny *Apache Commons Math* [25] pro výpočet LU dekompozice (Lower and Upper triangular matrix) matice a *JInterval* [26] pro intervalovou aritmetiku. Vrstva *CGA* (Computation Geometry Algorithms) obsahuje základní geometrické algoritmy, které jsou založeny na robustním jádře. Mezi tyto algoritmy patří například robustní test orientace bodu k úsečce. Na vrstvě *CGA* je závislá vrstva *Topology*, která obsahuje práci s topologickým grafem. Patří sem planarizace geometrie a vytvoření struktury nazvané Doubly Connected Edge List (DCEL), která je potřebná pro množinové a predikátové operace. Vrstva *Relate* obsahuje predikátové operace a vrstva *Overlay* obsahuje množinové operace. Jak již bylo zmíněno, obě vrstvy využívají vrstvu *Topology*. Dostáváme se k nejvrchnějším vrstvám geometrické knihovny. Vrstva *API* obsahuje aplikační rozhraní knihovny. Jedná se hlavně o geometrické typy a továrnu na geometrie. Vrstva *Referencing* zapouzdřuje práci se souřadnicovými systémy. Tato vrstva používá externí knihovnu *JScience* [27], která obsahuje implementaci standardu ISO 19111 [28]. Tento standard definuje souřadnicové systémy založené na souřadnicích. Na vrstvě *Referencing* je závislá většina vrstev knihovny, protože obsahuje implementaci souřadnic a jejich továrny. Následuje poslední vrstva architektury, která již představuje integrační vrstvu a nese název *Persistent*. Jak již název napovídá, zapouzdřuje perzistenci geometrických typů v objektové databázi db4o. Perzistence geometrických typů je v principu dosaženo dědičností těchto typů z abstraktní třídy, která ji zajišťuje a menšími implementačními zásahy v těchto typech.

Z hlediska návrhu by bylo velmi výhodné, aby byla integrační vrstva zcela nezávislá. To by dovolovalo použití různé objektové databáze pro perzistenci prostorových dat. Z hlediska implementace se mi bohužel nepovedlo tuto nezávislost dodržet a integrační vrstva je těsně svázána s databází db4o a s geometrickou knihovnou. Použití jiné databáze by tedy vyžadovalo zásah do implementace horních vrstev knihovny a vytvoření nové integrační vrstvy pro danou databázi.

I když jsem se snažil, aby byl návrh co možná nejjednodušeji rozšiřitelný, obávám se, že některé rozšíření mohou být i přes to dosti náročná. Jako vzor pro rozšiřující návrh by mohla dalším vývojářům posloužit například knihovna CGAL.

5 Vývoj geometrické knihovny

V této kapitole bude čtenář detailně seznámen s návrhem a implementací geometrické knihovny. Jednotlivé podkapitoly popisují, mimo jiné, vrstvy architektury geometrické knihovny od spodních vrstev směrem k vzhůru.

5.1 Inspirace

V předchozím textu byla zmiňovaná geometrická knihovna JTS, která je vyvíjena pro platformu Java. Pro připomenutí, je založena na standardu SFS. V implementaci vlastní geometrické knihovny jsem se touto knihovnou v některých aspektech inspiroval. Ve velké většině případů jsem se ale snažil o vlastní řešení, které jen vychází z této inspirace. Celkově si myslím, že moje geometrická knihovna je oproti JTS z velké části zjednodušená a možná v některých částech přehlednější. Navíc je postavena na jiném standardu. Nemůže se s ní ovšem rovnat, co se týče funkčnosti a výkonnosti.

5.2 Dimenze

Geometrické algoritmy jsou implementovány pro 2 dimenze. Aby bylo možné algoritmy rozšířit na 3 dimenze, použil jsem návrhový vzor strategie všude, kde to bylo možné. Rozšíření na 3 dimenze by ovšem bylo nejnáročnější nejspíše ve vrstvě topologie.

5.3 Robustní jádro

Jak již bylo stručně řečeno v kapitole o architektuře, jádro geometrické knihovny je založeno na robustním výpočtu znaménka determinantu. Tento algoritmus je základem většiny geometrických algoritmů. Abych dosáhl robustnosti s co nejmenším vlivem na výkon, využívám techniku pozdního vyhodnocení s použitím dynamického filtru z EGC, který je založen na intervalové aritmetice [29].
Determinant je vyhodnocen metodou LU dekompozice ve tvaru PA=LU. Následující Diagram 2 zobrazuje jádro spolu se závislostmi. Algoritmus LU dekompozice je implementován v knihovně *Commons Math*. Abych mohl použít intervalovou aritmetiku, musel jsem zapouzdřit typ *RealInterval* z knihovny *JInterval* tak, abych ho mohl použít v algoritmu LU dekompozice. Toho jsem jednoduše dosáhl implementací rozhraní *FieldElement*. Exaktní výpočet je založen na numerickém typu s neomezenou přesností nazvaném *BigFraction*. Implementuje racionální aritmetiku. Vyhodnocení znaménka je zobrazeno v poznámce metody *computeSignOfDeterminant* třídy *RobustKernel* a má tedy následující kroky:

1. Determinant se vyhodnotí pomocí intervalové aritmetiky.
2. Nelze-li v prvním kroku rozhodnout o znaménku, vyhodnotí se determinant exaktně za použití aritmetiky nad čísly s neomezenou přesností.

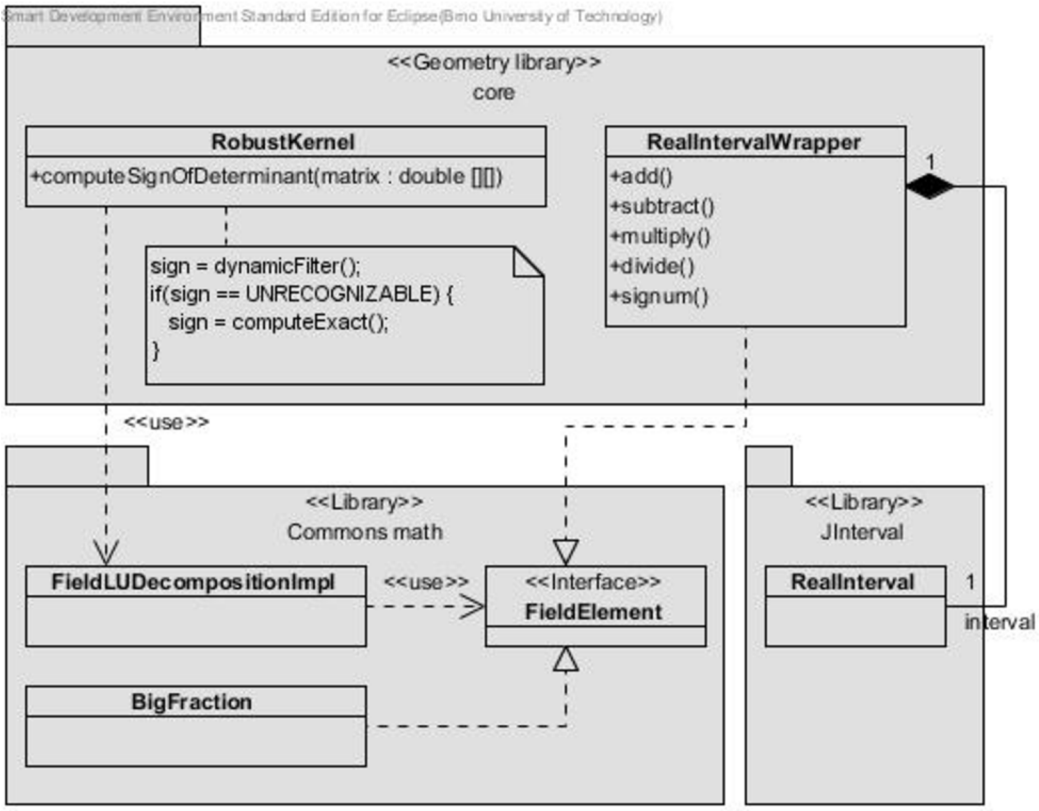


Diagram 2: Robustní jádro geometrické knihovny.

Zvažoval jsem použití zásuvného jádra. To znamená, že bych měl pouze adaptér jádra, na který bych napojil již implementované robustní jádro pro geometrické výpočty. Bohužel taková jádra existují pouze pro jazyky C/C++ a nikoli pro Javu. Jedná se o již několikrát zmiňovanou CGAL, CORE Library apod. Pro jejich použití by bylo nutné implementovat nativní rozhraní (Java Native Interface). V době návrhu jádra se mi toto zdálo jako velký výkonnostní nedostatek a proto jsem se rozhodl k implementaci vlastního jádra.

Jádro nepoužívá exaktní vstup, nýbrž pouze čísla s omezenou přesností. Konkrétně datový typ *double*. Díky tomu lze robustnosti dosáhnout pouze, není-li vstup zaokrouhlen. To se děje vždy, když pochází z aplikačního rozhraní. Například souřadnice nějakého bodu. Problém s robustností nastává, když je na vstup přivedena hodnota, která byla předtím zaokrouhlena. Například po výpočtu průsečíku dvou přímek.

Díky tomuto problému by bylo vhodné implementovat jádro způsobem, který by dovolil exaktní vstup. Vhodným řešením by možná bylo takové, které dovolí použití numerického typu bez znalosti, zda je, či není exaktní. Například použitím rozhraní.

Jelikož je jádro na nejnižší úrovni architektury, projevil se tento problém samozřejmě i ve vyšších vrstvách. Například při implementaci planarizace geometrie ve vrstvě topologie, kdy jsem musel použít jednoduchý algoritmus s časovou složitostí $O(n^2)$ namísto rychlého algoritmu nazvaného obecně *Planesweep*, který potřeboval právě exaktní vstup kvůli průsečíkům přímek. Více o důsledcích tohoto problému v kapitolách popisujících vrstvy, kterých se výrazně dotknul.

5.4 Základní algoritmy

Tato kapitola čtenáře seznamuje s principy, návrhem a implementací základních geometrických algoritmů, které stojí nad robustním jádrem. Jedná se o vrstvu *CGA*. Mezi tyto algoritmy patří test orientace bodu k úsečce, test příslušnosti bodu k úsečce nebo k oblasti, orientace

okruhu, výpočet úhlu mezi dvěmi úsečkami, test a výpočet průniku dvou úseček. Při implementaci některých algoritmů jsem se inspiroval buď v knihovně JTS, nebo ve zdroji [30]. Není-li řečeno jinak, jsou algoritmy implementovány přesně tak, jak jsou popsány. Popis většiny algoritmů je doprovázen schématem pro lepší pochopení. Proměnné, které se ve schématech nacházejí, jsou v textu psány kurzívou pro odlišení.

5.4.1 Test orientace bodu k úsečce

Vstupem tohoto algoritmu jsou 3 souřadnice *p*0, *p*1 a *q*. Úsečka *s* je dána souřadnicemi *p*0-*p*1. Algoritmus rozhoduje o pozici souřadnice *q* vzhledem k úsečce *s*. Využívá k tomu výpočet znaménka determinantu. Možné situace algoritmu ukazuje Schéma 2.

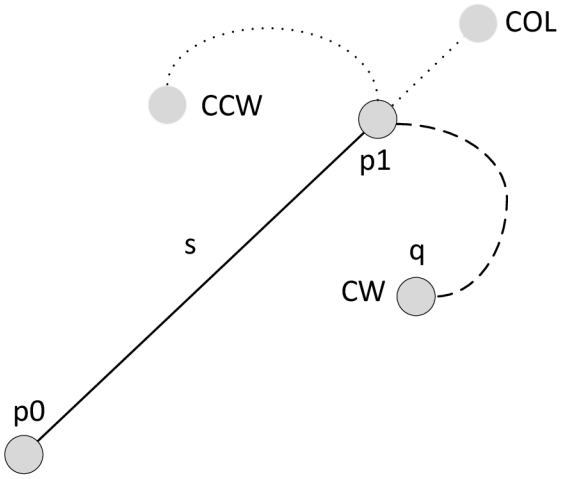


Schéma 2: Situace algoritmu orientace bodu k úsečce.

Množina možných výsledků obsahuje 3 hodnoty. Jsou jimi *counter-clockwise* (*CCW*), *collinear* (*COL*) a *clockwise* (*CW*). Determinant se počítá z matice uvedené ve Vzorci 1 běžným způsobem:

$$\begin{bmatrix} p1.x - p0.x & p1.y - p0.y \\ q.x - p0.x & q.y - p0.y \end{bmatrix}$$

Vzorec 1: Matice orientace bodu *q* k úsečce *p0-p1*.

Hodnota výsledku se odvíjí od znaménka determinantu (D) následovně:

- D < 0 *CW*, vpravo
- D > 0 *CCW*, vlevo
- D = 0 *COL*, v ose

Tento algoritmus je základem dále zmiňovaných základních algoritmů.

5.4.2 Test příslušnosti bodu k úsečce

Bod *q* přísluší k úsečce *s*, pokud se nachází v jejím vnitřku. Přičemž, vnitřek úsečky je tvořen všemi body, které na ní leží, kromě jejich dvou koncových bodů, *p*0 a *p*1. Vstupem algoritmu jsou tedy 3 souřadnice *p*0, *p*1 a *q*, přičemž úsečka je tvořena body *p*0-*p*1. Pro výpočet je použit předchozí algoritmus orientace. V první fázi se rozhoduje, je-li výsledek orientace hodnota *COL*. Poté se provádí množina porovnání, které určí, zda-li bod *q* leží mezi body *p*0 a *p*1. Situaci algoritmu ukazuje Schéma 3.

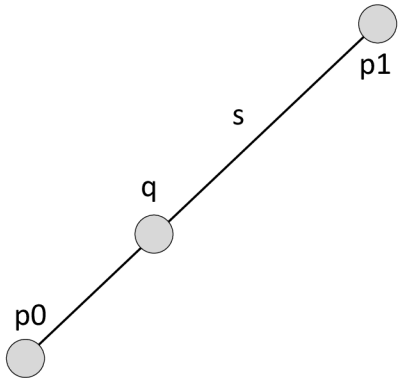


Schéma 3: Situace algoritmu příslušnosti bodu k úsečce.

5.4.3 Test příslušnosti bodu k oblasti

Vstupem tohoto algoritmu je množina souřadnic, které musí tvořit okruh a souřadnice *q*, která se testuje. Okruh je množina úseček, které na sebe navazují, nekříží se a počáteční souřadnice první úsečky je rovna koncové souřadnici poslední úsečky, tzn. jsou uzavřené. Situaci algoritmu zachycuje Schéma 4.

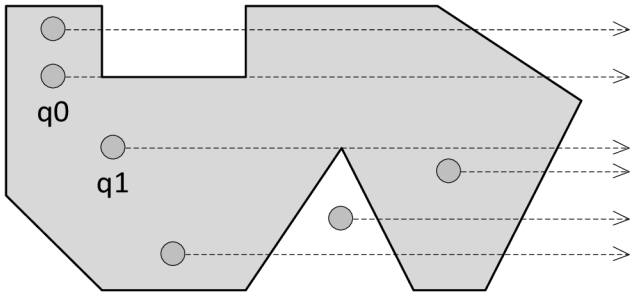


Schéma 4: Situace algoritmu testu příslušnosti bodu k oblasti.

Souřadnice *q* přísluší oblasti dané okruhem právě tehdy, když leží uvnitř této oblasti, tzn. neleží na hranici, která je dána množinou úseček okruhu. V první fázi algoritmu se testuje, není-li souřadnice *q* rovna některé koncové souřadnici úsečky z okruhu. Je-li tomu tak, neleží bod *q* v oblasti. Jinak se počítá počet průniků přímky, která má počátek v souřadnici *q* a vede horizontálně v kladném směru osy *x*. V této fázi se opět používá test orientace bodu k úsečce. Souřadnice *q* náleží oblasti, je-li počet průniků lichý. Mohou nastat dvě situace, které toto pravidlo porušují, ale i přes to je souřadnice uvnitř oblasti. Jsou jimi souřadnice *q*0 a *q*1 ve schématu. Implementovaný algoritmus se ovšem vypořádá i s těmito situacemi.

5.4.4 Orientace okruhu

Okruh je tvořen na sebe navazujícími úsečkami, přičemž platí, že počáteční bod první úsečky je roven koncovému bodu poslední úsečky. Okruhy jsou základním stavebním kamenem ploch. Algoritmus pro výpočet orientace okruhu je opět založen na testu orientace bodu k úsečce. Vstupem je množina souřadnic tvořících okruh. Množina výsledků obsahuje 2 hodnoty. Jsou jimi *counter-clockwise* (*CCW*) a *clockwise* (*CW*). Algoritmus hledá souřadnici *high*, která má nejvyšší hodnotu *y*, spolu s jejím následníkem *next* a předchůdcem *prev*. Celý okruh má potom orientaci stejnou jako křivka, která je definována souřadnicemi *prev*, *high*, *next*, v tomto pořadí. Postup výpočtu je popsán následujícím pseudo algoritmem:

IF (souřadnice *prev*, *high* a *next* jsou vzájemně kolineární, tedy v ose) THEN
  IF *(prev.x < next.x)* THEN *CW*
  ELSE IF *(prev.x > next.x)* THEN *CCW*
ELSE výsledek je roven výsledku testu orientace těchto tří souřadnic.

Implementace obsahuje predikát orientace okruhu, který vrací pravdivostní hodnotu *pravda*, je-li okruh orientován proti směru hodinových ručiček, tedy *CCW*.

5.4.5 Výpočet úhlu

Vstupem tohoto algoritmu je množina 3 souřadnic *p*0, *p*1 a *p*2. Přičemž souřadnice *p*0-*p*1 tvoří úsečku *s* a souřadnice *p*1-*p*2 tvoří úsečku *t*. Algoritmus počítá úhel α po směru hodinových ručiček mezi úsečkami *s* a *t* kolem společné souřadnice *p1*, jak ukazuje Schéma 5.

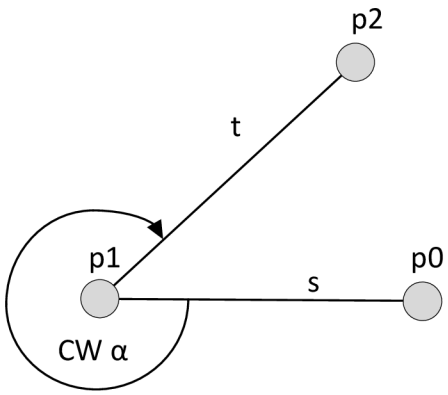


Schéma 5: Situace algoritmu pro výpočet úhlu mezi dvěmi úsečkami v CW.

Výsledkem algoritmu je hodnota úhlu ve stupních, nikoli radiánech.

5.4.6 Průnik dvou úseček

Tento algoritmus lze rozdělit na část predikátovou a část konstrukční. Predikát vrací pravdivostní hodnotu *pravda*, pokud se dvě úsečky protínají. Konstrukční část vrací i hodnotu průsečíku tohoto průniku. V implementaci je použita strategie, která zapouzdřuje tyto dvě části do jediné třídy *LineIntersector2D*. Lze tak pomocí jediného objektu rozhodnout, zda se dvě úsečky protínají, zda se protínají v koncovém bodě a lze také získat aproximovanou souřadnici průsečíku.

Vstupem jsou souřadnice *p*0-*p*1 definující úsečku *s* a *q*0-*q*1 definující úsečku t, jak lze vidět na Schématu 6.

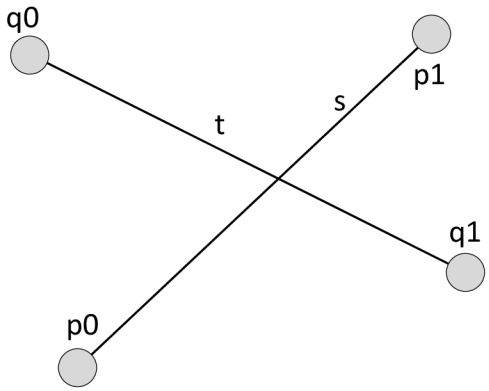


Schéma 6: Situace algoritmu testu a výpočtu průniku dvou úseček.

Predikátová část je postavena na testu orientace bodu k úsečce a je tedy robustní. Úsečky *s* a *t* se protínají pravě tehdy, když *p*0 leží na opačné straně úsečky *t* než *p1* a zároveň *q*0 leží na opačné straně úsečky *s* než *q*1. Konstrukční část je postavena na homogenních souřadnicích [31]. Úsečka definovaná rovnicí $ax + by + c = 0$ má homogenní souřadnici rovnu (a, b, c). Bod (x, y) má homogenní souřadnici rovnu $(x, y, 1)$. Výpočet je sice proveden robustně, protože není použita operace dělení, ale při získávání průsečíku je tento převeden zpět na datový typ *double*.

Zde bych se rád vrátil k problému zmiňovanému u robustního jádra. Vhodné by bylo, aby se při výpočtech nemusela použitá hodnota průsečíku aproximovat, ale mohla se použít přesná hodnota. To by bylo možné, kdyby jádro mělo exaktní vstup. Poté by se daly použít rychlejší algoritmy na vyšších vrstvách, které tento algoritmus používají.

5.5 Topologie

Tato kapitola se zabývá jednou z nejdůležitějších vrstev architektury. Čtenáře seznámí s návrhem řešení topologie. Nejdříve bude definován graf geometrie spolu se stručným popisem algoritmu pro jeho vytvoření. Následně budou detailně popsány jednotlivé fáze tohoto algoritmu. Pro výklad bude použit příklad na dvojici polygonů, které budou označeny *A* a *B*, spolu s pomocnými schématy. Postup výkladu kopíruje kroky implementace.

5.5.1 Topologický graf geometrie

Graf geometrie je definován jako graf. Pouze pro připomenutí, mějme graf G. Dále mějme množinu uzlů U a množinu hran H, přičemž uzly jsou propojeny hranami. Potom můžeme psát $G = (U, H)$. Graf geometrie je vytvořen z geometrie, kde bodům odpovídají uzly a úsečkám odpovídají hrany. Graf geometrie můžeme nazvat topologickým, je-li planární, má vytvořenu strukturu DCEL a má označkované uzly a hrany. Připomínám, že planární graf byl definován v kapitole 2.2. Dále v textu budeme pojmem graf chápat topologický graf geometrie, nebude-li řečeno jinak.

Implementace obsahuje třídu *GeometryGraphBuilder*, která má zodpovědnost za vytvoření grafu. Vstupem může být jedna, či dvě geometrie. Jednotlivé fáze algoritmu na sebe navazují a každá další fáze je závislá na předešlé. Některé fáze algoritmu jsou delegovány na jiné třídy, které se na konstrukci podílejí. První fáze je delegována třídě *Geometry2GraphTransformer*, která transformuje geometrii(e) na uzly a hrany. Vytvořené uzly a hrany jsou označkovány vzhledem ke geometrii, ze které pocházejí a tvoří počáteční graf. Je-li geometrie typu plocha, potom je vnější hranice orientována po směru a hranice díry proti směru hodinových ručiček. Ve druhé fázi se provádí tzv. *noding*, kdy se graf z předešlé fáze planarizuje, tzn. převede se na planární. Následující, již třetí, fáze vytváří strukturu DCEL. Jedná se o poměrně složitý proces, který je částečně delegován na třídy nesoucí zodpovědnost za DCEL. V poslední, čtvrté, fázi je dokončeno značkování vzhledem ke druhé geometrii. Po této fázi je graf vytvořen a je možné ho použít v příslušných prostorových operacích.

5.5.2 Transformace geometrie

Jak již bylo řečeno, vstupem této fáze je buď jedna, nebo dvě geometrie. Výstupem je množina uzlů a hran, které tvoří počáteční graf. Situaci zobrazuje Schéma 7.

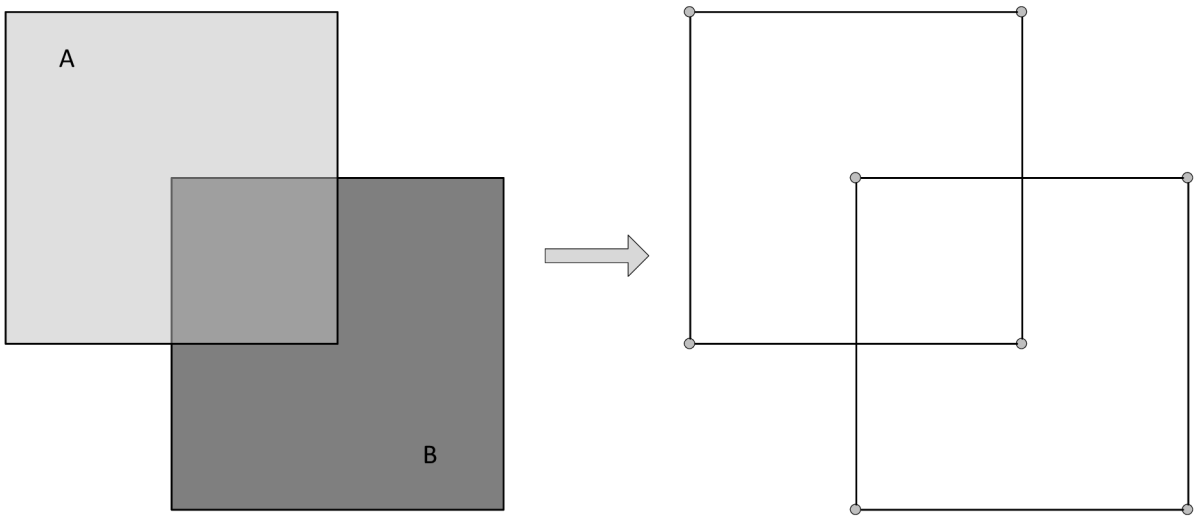


Schéma 7: Transformace geometrií na graf.

Jsou dány dva jednoduché polygony *A* a *B*, které se částečně překrývají. Každá geometrie obsahuje 4 uzly a 4 hrany. Počáteční graf tedy obsahuje 8 uzlů a 8 hran. Nutno podotknout, že tento graf není planární, protože obsahuje hrany, které se vzájemně kříží.

5.5.3 Planarizace grafu

Vstupem této fáze je počáteční graf a výstupem je planární graf. Situace je zobrazena na Schématu 8.

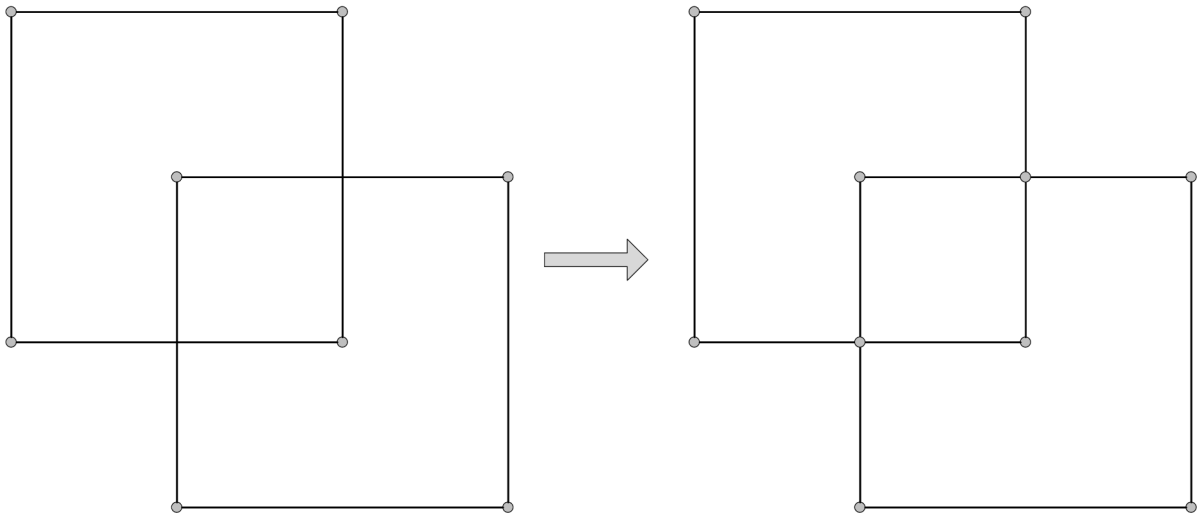


Schéma 8: Planarizace počátečního grafu.

Výsledný planární graf obsahuje 10 uzlů, kde 2 uzly jsou nové. Jedná se o průsečíky úseček dvou geometrií. Počet hran se zvýšil na hodnotu 12.

Implementace pro planarizaci grafu používá termín *noding*. Jádrem této fáze je robustní počítání průsečíků množiny hran. Jako první se tímto algoritmem zabývali pánové Greene a Yao. Jejich práce, která již byla jednou zmiňována, se nachází ve zdroji [21]. Na jejich práci navazovaly postupně další. Jejich přehled lze nalézt ve zdroji [32].

Algoritmus musí řešit již zmiňovaný základní problém výpočetní geometrie, a sice nemožnost reprezentovat celou množinu reálných čísel. Důvodem je, že některé vypočítané průsečíky nemohou být reprezentovány číselným typem s omezenou přesností. Díky tomu vznikají topologické změny, které jsou v počítání s omezenou přesností nevyhnutelné. Ovšem, lze rozlišit dva typy těchto změn.

Degeneraci a inverzi. Degenerace nastává, když se hrana grafu změní na bod a neovlivní to jiné vztahy. Degenerace je akceptovatelnou změnou. Oproti tomu, inverze nastává, když bod, který leží uvnitř určité oblasti se po změně ocitne mimo tuto oblast. Inverze je neakceptovatelná a pro dosažení robustnosti je nutné ji řešit [32].

V implementaci je algoritmus zastřešen strategií nazvanou *NoderStrategy*, která pro dosažení svého cíle deleguje činnosti dalším strategiím, kterými jsou *EdgeSetIntersectorStrategy* a *RounderStrategy*. Díky tomuto přístupu lze implementovat různé strategie algoritmu noding. Hlavně co se týče časové a prostorové náročnosti. V následujícím textu budou jednotlivé strategie detailněji popsány.

Strategie *EdgeSetIntersectorStrategy* počítá průsečíky množiny hran grafu. Používá k tomu strategii *LineIntersector2D* pro výpočet průniku dvou úseček, který byl popsán v základních algoritmech. Rozšíření implementuje tuto strategii velmi jednoduchým způsobem, kdy jsou počítány průsečíky každé hrany s každou. Časová složitost tohoto algoritmu je tedy $O(n^2)$. Existuje výkonnější algoritmus, který je založen na obecném algoritmu *PlaneSweep*. Případného zájemce, který by se chtěl dozvědět více o tomto algoritmu, bych rád odkázal na práci pánů Bentleyho a Ottmanna, která se nachází ve zdroji [33]. Práce obsahuje popis algoritmu pro výpočet průsečíků množiny hran založený na *PlaneSweep*. Pokoušel jsem se tento algoritmus použít, ale vyskytl se problém, který byl již popsán v kapitole robustního jádra.

Strategie *RounderStrategy* provádí zaokrouhlování uzlů grafu do mřížky s pevně danou přesností a následné překreslování hran přes tyto uzly, čímž je zajištěna robustnost této fáze. Souřadnice ve zmiňované mřížce jsou celočíselné. Tato strategie je založena na práci pana Hobbyho [34], která používá princip nesoucí název *snap rounding*. Tento princip bude vysvětlen na Schématu 9.

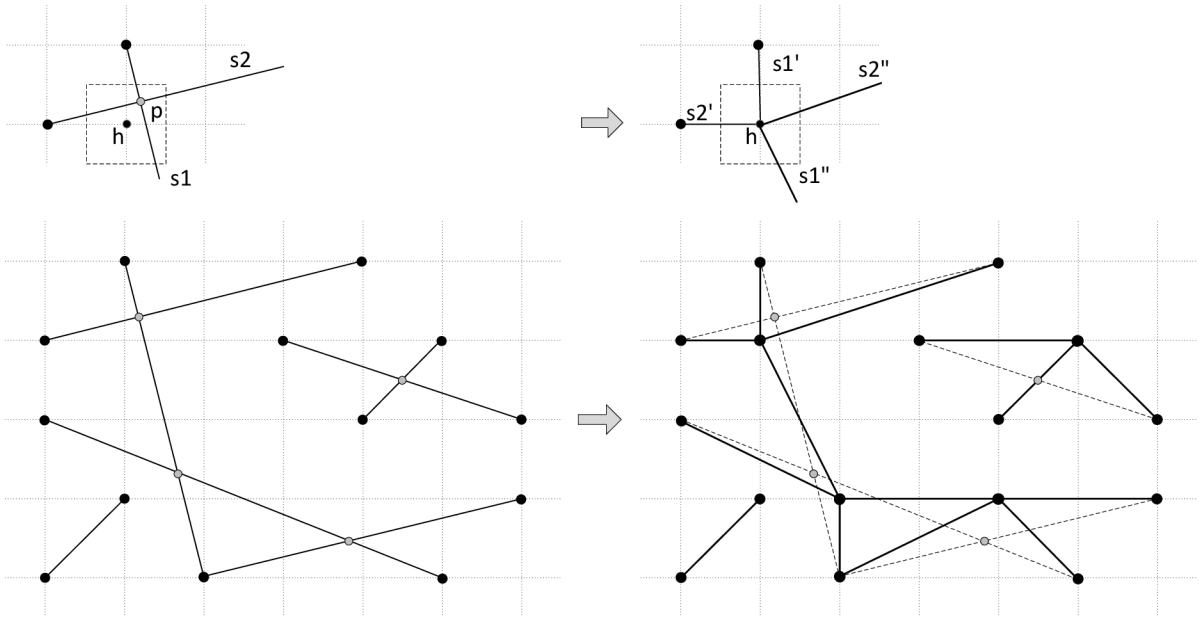


Schéma 9: Snap rounding založený na Hot Points [32].

Toto schéma je horizontálně rozděleno na dvě části, přičemž horní část zobrazuje detail spodní části, na které je zobrazena celková situace principu *snap rounding*. Pro každý vypočítaný uzel průsečíku a zbylé uzly v grafu se zaokrouhlením na celá čísla vytvoří tzv. *hot point* (dále h-bod). Každý h-bod představuje souřadnici ve zmiňované mřížce a obsahuje čtvercové okolí, nebo-li zaokrouhlovací interval. Tento interval má velikost $\pm\frac{1}{2}$. Tato situace je zobrazena v levé horní části schématu, kde průsečík hran *s1* a *s2* je označen písmenem *p* a jeho h-bod písmenem *h*. Nyní je potřeba překreslit

každou hranu přes všechny h-body, které se vyskytují v její blízkosti, přičemž h-body se nacházejí v přirozeném pořadí od počátku do konce původní hrany. Hrana se nachází v blízkosti h-bodu právě tehdy, když protíná jeho zaokrouhlovací interval. Překreslení hran *s1* a *s2* je zobrazeno v pravé horní části schématu. Procházela-li by nějaká další hrana přes zaokrouhlovací interval h-bodu *h*, pak by i tato byla přes tento h-bod překreslena. Toto je implementováno také s využitím strategie *LineIntersector2D*.

5.5.4 Struktura Doubly Connected Edge List

Vstupem této, předposlední fáze tvorby grafu, je planární graf. Výstupem je struktura Doubly Connected Edge List, která je s grafem úzce svázaná. Informace o DCEL jsou čerpány ze zdroje [35], který poskytuje kvalitní přehled o této problematice. Další informace o DCEL lze najít ve zdroji [30]. Situaci algoritmu zobrazuje Schéma 10.

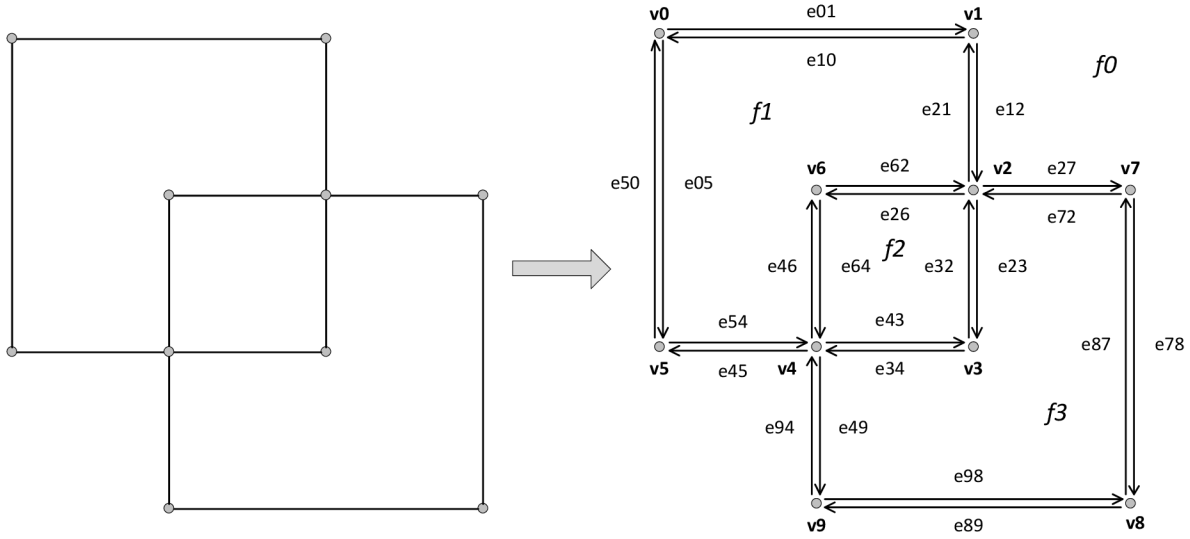


Schéma 10: Vytvoření struktury DCEL.

Struktura DCEL obsahuje záznam pro každý vrchol (*vertex*), půl-hranu (*half-edge*) a oblast (*face*). Ve schématu jsou vrcholy značeny písmenem *v*, půl-hrany písmenem *e* a oblasti písmenem *f*. Půl-hranu si lze představit jako orientovanou úsečku. Informace zapouzdřené v záznamech zobrazuje Tabulka 1.

Záznam	Informace
Vrchol	• souřadnice • přilehlá půl-hrana (*incident edge*)
Půl-hrana	• počáteční vrchol (*origin*) • půl-hrana, která má opačný směr (*twin*) • přilehlá oblast (*incident face*) • následník (*next*) • předchůdce (*prev*)
Oblast	• půl-hrana, která je součástí vnější hranice (*outer component*) • seznam půl-hran, které jsou součástí vnitřních hranic (*inner components*)

Tabulka 1: Záznamy struktury DCEL.

Následující Schéma 11 zobrazuje veškeré informace zapouzdřené v půl-hraně s názvem *e* přímo v grafu. Jak je vidět, půl-hrana tvoří s opačnou hranou původní hranu grafu. Díky provázanosti půl-hran lze strukturu poměrně lehce procházet.

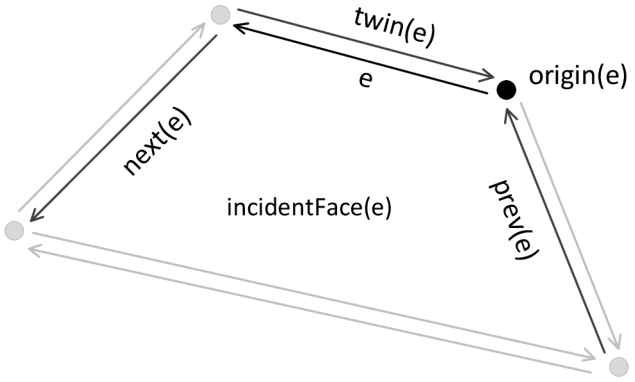


Schéma 11: Ukázka DCEL záznamu pro half-edge v grafu.

Konstrukce struktury DCEL je poměrně složitý proces, který obsahuje několik kroků. V následujícím textu bude tento proces detailně popsán krok po kroce.

V prvním kroce se vytváří záznamy pro všechny vrcholy a půl-hrany, a správně se nastavují informace, jako předchůdce a následník půl-hrany apod. V tomto kroce se zatím nepracuje s oblastmi. Ty jsou vytvořeny až ve druhém kroce. Používá se k tomu graf cyklů. Nejdříve se v grafu naleznou cykly, které jsou vloženy do grafu cyklů jako uzly. Následně se do grafu vkládají hrany dle určitého pravidla, aby se vytvořily spojené komponenty grafu. Ty budou tvořit výsledné oblasti. Tento druhý krok je detailně popsán v následujícím textu s pomocí Schémat 12, 13 a 14.

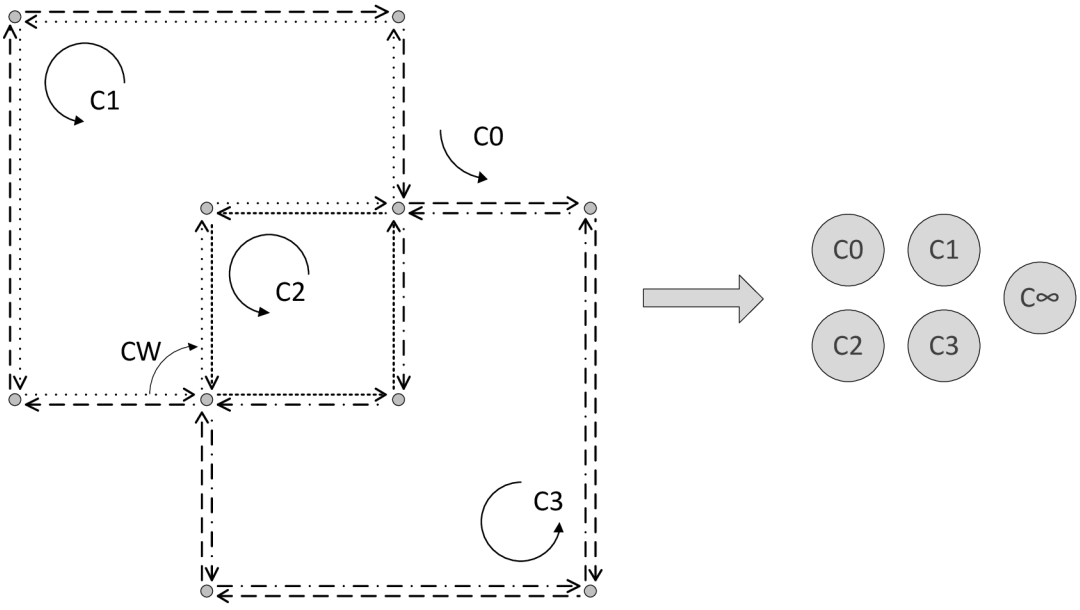


Schéma 12: Cykly při konstrukci DCEL oblastí a graf cyklů.

V levé části Schématu 12 se nachází částečná struktura DCEL obsahující vrcholy a půl-hrany spolu s vyznačenými cykly. Každý cyklus je tvořen půl-hranami, které na sebe navazují a počáteční vrchol první je roven koncovému vrcholu poslední půl-hrany. Pokud se při procházení půl-hran narazí na vrchol, ze kterého vede více půl-hran, pokračuje se první půl-hranou po směru hodinových ručiček, což je ve schématu znázorněno úhlem s popisem *CW* v cyklu *C1*. Při hledání cyklů lze tedy

s výhodou použít již existující částečnou strukturu DCEL, kdy jsou využiti hlavně následníci půl-hran a výpočet úhlů, což je základní algoritmus. V pravé části Schématu 12 je zobrazen graf cyklů, který obsahuje $4 + 1$ uzlů, ale zatím žádné hrany. Uzel C_∞ má speciální význam. Představuje imaginární vnější hranici neomezené oblasti. Neomezená oblast v tomto případě je oblast, která začíná vně cyklu *C0* a končí v nekonečnu, tedy v C_∞.

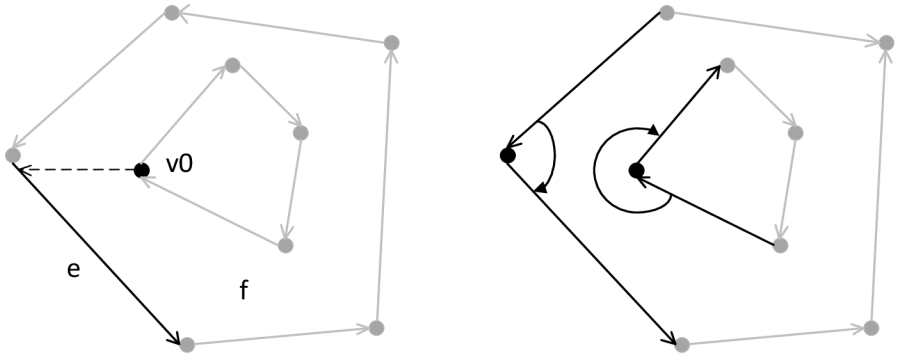


Schéma 13: Hledání spojených komponent v grafu cyklů.

Poté, co jsou nalezeny všechny cykly a jsou vloženy do grafu cyklů jako uzly, je potřeba vložit také hrany. K tomuto účelu se používá následující pravidlo. Mezi dva uzly grafu cyklů se vloží hrana právě tehdy, když jeden cyklus je hranicí díry a druhý cyklus má půl-hranu bezprostředně vlevo od nejlevějšího vrcholu cyklu díry. Tato situace je znázorněna v levé části Schématu 13. Nejlevější vrchol cyklu díry nese název *v*0 a bezprostředně nejlevější půl-hranou je půl-hrana s označením *e*. Mezi tyto dva cykly se tedy vloží hrana. Tím pádem budou součástí jedné spojené komponenty grafu cyklů. Spojená komponenta může být tvořena 1 a více uzly. Není-li nalezena žádná půl-hrana bezprostředně vlevo od nejlevějšího vrcholu, je vložena hrana mezi uzel daného cyklu díry a uzel cyklu C_∞. Zbývá zodpovědět otázku, jak zjistit, který cyklus je hranicí díry a který nikoli. Odpověď je jednoduchá. Je-li úhel ve směru hodinových ručiček mezi dvěmi půl-hranami v nejlevějším vrcholu cyklu větší nebo roven úhlu 180^O, potom se jedná o díru. Tato situace je znázorněna v pravé části Schématu 13. Aplikujeme-li toto pravidlo na dosavadní graf cyklů, vznikne graf cyklů, který již obsahuje hrany mezi uzly. Takovýto graf cyklů je zobrazen na následujícím Schématu 14.

Schéma 14: Výsledný graf cyklů se 4 spojenými komponentami.

Nakonec se pro každou spojenou komponentu grafu cyklů vytvoří záznam oblasti a správně se nastaví jeho informace. Všechny cykly z této spojené komponenty jsou přilehlé k dané oblasti. Pro ukázkový graf cyklů budou vytvořeny 4 záznamy pro oblasti *f*0-*f*3. Aby bylo možné provést poslední fázi konstrukce grafu geometrie, je nutné aby oblasti obsahovaly také značení vůči geometriím. Toto značení je provedeno při vytváření oblastí implementačně specifickým způsobem, kdy jsou použity značky hran vzhledem ke geometrii, ze které pocházejí. Podrobnosti o značkách se nacházejí v následující kapitole. Následující Tabulka 2 zobrazuje značky právě vytvořených oblastí.

Oblasti	Vzhledem ke geometrii	Hodnota značky
f0	A B	Exterior Exterior
f1	A B	Interior Exterior
f2	A B	Interior Interior
f3	A B	Exterior Interior

Tabulka 2: Tabulka označení oblastí.

5.5.5 Značkování

Vstupem poslední fáze je graf, který je svázán s vytvořenou strukturou DCEL. Výstupem bude hotový graf, jehož uzly a hrany budou plně označkovány.

Každý uzel a hrana grafu obsahuje dvě značky. Jedna určuje pozici vzhledem ke geometrii, ze které pochází a druhá určuje pozici vzhledem ke druhé geometrii, je-li obsažena. Značka uzlu obsahuje jediný atribut, a sice *on*. Značka hrany obsahuje atributy tři. Jsou jimi *left*, *on*, *right*. Množina hodnot těchto atributů obsahuje *Interior*, *Boundary* a *Exterior*. Atributy tedy reprezentují informaci, zda se vlevo, resp. vpravo, resp. na dané úsečce nebo bodu nachází vnitřek, vnějšek nebo hranice dané geometrie (vlastní nebo druhé). Tento princip pochází ze zdroje [30]. Značky pro vlastní geometrii jsou vytvářeny v první fázi konstrukce grafu. Značky ke druhé geometrii jsou vytvářeny v této fázi s použitím oblastí ze struktury DCEL.

Následující Schéma 15, a k němu příslušné Tabulky 3 a 4, zobrazují značky pro dané uzly a hrany. Tabulka 3 obsahuje značky uzlů a Tabulka 4 obsahuje značky hran. Pro jejich vysvětlení je použit uzel *n*0 a hrana *e*26. Uzel *n*0 leží na hranici geometrie *A* a mimo geometrii *B*. Proto obsahuje značku s hodnotou *on* = *Boundary* vzhledem ke geometrii *A* a značku s hodnotou *on* = *Exterior* vzhledem ke geometrii *B*. Hrana *e*26 vedoucí z uzlu *n2* do uzlu *n6* tvoří hranici geometrie *B* a leží zároveň v geometrii *A*. Proto obsahuje značku pro geometrii *A* s hodnotami *left* = *Interior, on* = *Interior*, *right* = *Interior* a značku pro geometrii *B* s hodnotami *left* = *Interior*, *on* = *Boundary* a *right* = *Exterior*.

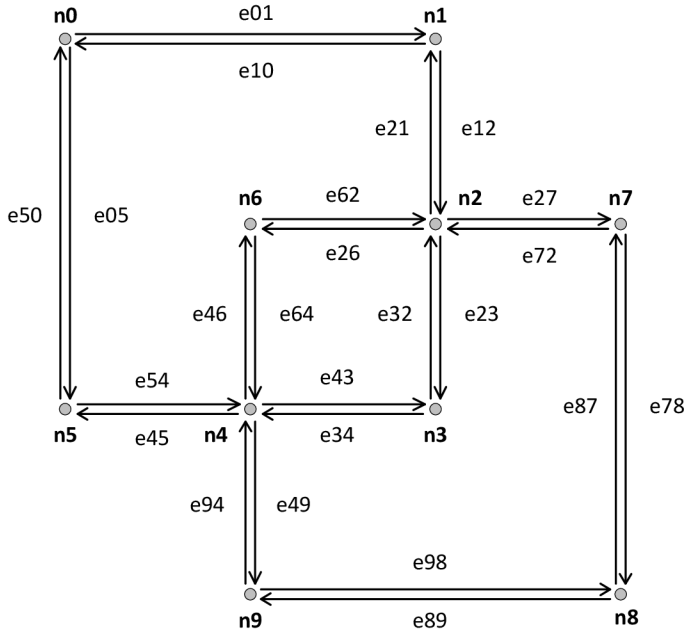


Schéma 15: Značkování uzlů a hran.

Uzly	Vzhledem ke geometrii	Hodnota atributu *on*
n0, n1, n5	A B	Boundary Exterior
n7, n8, n9	A B	Exterior Boundary
n6	A B	Interior Boundary
n3	A B	Boundary Interior
n2, n4	A B	Boundary Boundary

Tabulka 3: Tabulka označení uzlů.

Hrany	Geometrie	Atribut *left*	Atribut *on*	Atribut *right*
e45, e50, e01, e12	A B	Exterior Exterior	Boundary Exterior	Interior Exterior
e27, e78, e89, e94	A B	Exterior Exterior	Exterior Boundary	Exterior Interior
e21, e10, e05, e54	A B	Interior Exterior	Boundary Exterior	Exterior Exterior
e49, e98, e87, e72	A B	Exterior Interior	Exterior Boundary	Exterior Exterior
e23, e34	A B	Exterior Interior	Boundary Interior	Interior Interior
e46, e62	A B	Interior Exterior	Interior Boundary	Interior Interior
e43, e32	A B	Interior Interior	Boundary Interior	Exterior Interior
e26, e64	A B	Interior Interior	Interior Boundary	Interior Exterior

Tabulka 4: Tabulka označení hran.

5.6 Prostorové operace

Tato kapitola se zabývá vrstvami *Overlay* a *Relate*. Také obsahuje popis dalších prostorových operací, a sice konstrukčních a metrických.

5.6.1 Množinové operace

Množinové operace představují prostorové operace typu geometrické relace. V implementaci rozšíření nesou název *Overlay*, což je také název dané vrstvy. Spadají zde operace průniku (*intersection*), rozdílu (*difference*), symetrického rozdílu (*symmetricDifference*) a spojení (*union*) dvou geometrií. Tyto operace je nutné kvůli robustnosti provádět na topologickém grafu geometrie, který byl popsán výše. Výstupní geometrie musí být konstruována v závislosti na typu operace za pomocí značek oblastí grafu. Za tímto účelem by měla být použita struktura DCEL, která tyto informace uchovává. V aktuální verzi rozšíření je operace *overlay* implementována z 90 %, kde zbylých 10 % představuje právě konstrukce výstupní geometrie. Za tímto účelem je v rozšíření vytvořena třída *Graph2GeometryTransformer*, která obsahuje prázdné metody, které by bylo nutné doimplementovat. Díky zaokrouhlování ve *snap roundingu*, který byl popsán v předchozí kapitole,

není operace *overlay* příliš přesná. S touto nepřesností se ale u této operace počítá. Schéma 16 zobrazuje ukázku množinových operací na dvojici polygonů *A* a *B*.

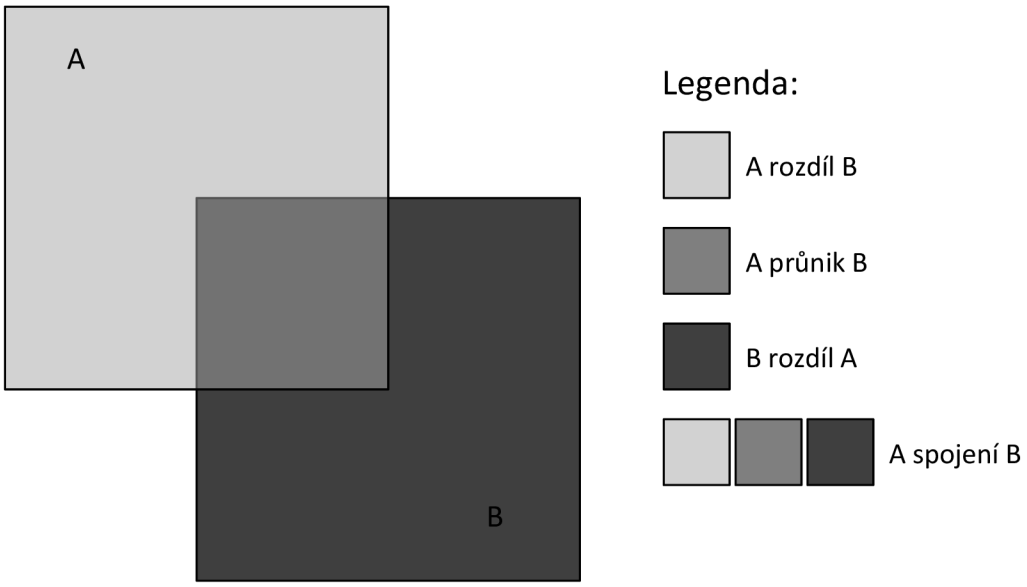


Schéma 16: Ukázka některých množinových operací.

5.6.2 Predikátové operace

Predikátové operace, jak již název napovídá, spadají do prostorových operací typu predikát. Vrstva, která se jimi zabývá, nese název *Relate* a aktuální verze rozšíření obsahuje úplnou podporu těchto operací. Kvůli robustnosti jsou podobně jako operace *overlay* založeny na topologickém grafu geometrie Jak již bylo řečeno dříve, implementace používá model DE-9IM, který bude popsán v následujícím v textu.

DE-9IM je založen na definici hranice, vnitřku a vnějšku geometrických typů. Obecné vysvětlení hranice, vnitřku a vnějšku zní následovně. Hranice je množina geometrických objektů další nižší dimenze. Vnitřek obsahuje body, které zůstanou uvnitř, jakmile se odstraní hranice. Vnějšek obsahuje body, které nejsou na hranici ani uvnitř. Lze si povšimnout souvislosti mezi značkami uzlů a hran v topologickém grafu geometrie a pojmy z modelu DE-9IM. Značky obsahují hodnoty hranic, vnitřku a vnějšku a jsou tak použity k výpočtu tabulky reprezentující daný prostorový vztah. Model definuje následující vztahy: *equals*, *disjoint*, *intersects*, *touches*, *crosses*, *within*, *contains* a *overlaps*. Počítá se dimenze průniku u devíti kombinací z hranic, vnitřku a vnějšku, jak ukazuje Tabulka 5.

DE-9IM matice	Uvnitř: I	Hranice: B	Vně: E
Uvnitř: I	$\dim(I(a) \cap I(b))$	$\dim(I(a) \cap B(b))$	$\dim(I(a) \cap E(b))$
Hranice: B	$\dim(B(a) \cap I(b))$	$\dim(B(a) \cap B(b))$	$\dim(B(a) \cap E(b))$
Vně: E	$\dim(E(a) \cap I(b))$	$\dim(E(a) \cap B(b))$	$\dim(E(a) \cap E(b))$

Tabulka 5: Tabulka zobrazující matici modelu DE-9IM.

Nyní je potřeba vysvětlit důležité pojmy, které jsou nutné ke správnému pochopení. Hodnoty tabulky jsou dimenzemi vztahu dvou geometrií. Je nezbytné, aby tato dimenze nebyla zaměňována za dimenzi souřadného systému (2D, 3D, nD). Obor hodnot a jejich vysvětlení obsahuje Tabulka 6.

Hodnota	Vysvětlení
0	Průnikem na dané pozici je bod.
1	Průnikem na dané pozici je úsečka.
2	Průnikem na dané pozici je plocha.
T	Průnik na dané pozici je neprázdný a nabývá hodnoty 0, 1 nebo 2.
F	Průnik na dané pozici je prázdný. Jedná se o hodnotu -1.

Tabulka 6: Hodnoty tabulky DE-9IM.

Následující Tabulka 7 popisuje, co tvoří vnitřek, vnějšek a hranici základních geometií, což je také důležité ke správnému pochopení.

Geometrie	Uvnitř (I)	Hranice (B)	Vně (E)
Bod	Bod	Prázdná množina	Body mimo I a B
Křivka	Body na křivce, kromě B	Dva koncové body	Body mimo I a B
Polygon	Body uvnitř vnějšího okruhu a vně vnitřních okruhů (děr)	Množina okruhů	Body mimo I a B

Tabulka 7: Vnitřek, Hranice a Vnějšek geometrií.

Predikát prostorového vztahu přebírá vzorovou matici. Každá její buňka může obsahovat následující hodnoty: $\{T, F, 0,1,2,*\}$, kde:

- Hodnoty $T, F, 0,1,2$ byly popsány v Tabulce 6.
- Hodnota hvězdička (*) značí, že na hodnotě nezáleží.

Pokud prostorový vztah odpovídá jedné z možných matic dle vzorové matice, potom je predikát pravdivý, jinak nepravdivý. Vzorovou matici lze také reprezentovat řetězcem o délce 9 znaků, který se vytváří shora dolů, zleva doprava. Potom, zmíněné vztahy mají vzorové matice zobrazené v Tabulce 8, přičemž ještě záleží na typu geometrií, které porovnáváme:

Vztah	Vzorové matice
equals	T*F**FFF*
disjoint	FF*FF****
intersects	T******** nebo *T******* nebo ***T***** nebo ****T****
touches	FT******* nebo F***T**** nebo F**T*****
crosses	T*T****** nebo T*****T** nebo 0********
within	T*F**F***
contains	T*****FF*
overlaps	T*T***T** nebo 1*T***T**

Tabulka 8: Predikáty a odpovídající vzorové matice modelu DE-9IM.

Čtenář se může o tomto modelu predikátových operací dozvědět více ve zdrojích [15], [30], [36], [37] a [38].

5.6.3 Konstrukční operace

Konstrukční operace jsou výpočetně náročné prostorové operace. Patří mezi ně výpočet konvexní obálky a buffer. Aktuální verze rozšíření tyto operace neimplementuje.

Konvexní obálka je tvořena množinou bodů tvořících obálku dané geometrie. Buffer představuje všechny body, které mají od dané geometrie určitou vzdálenost danou prahovou hodnotou. Jedná se o prostředek, který je hojně využíván v GIS. Více informací o těchto operacích a jejich případných implementacích se může čtenář dozvědět ve zdroji [30].

5.6.4 Metrické operace

Metrické operace patří také do výpočetně náročných prostorových operací. Jedná se o operace pro výpočet vzdálenosti, plochy, centroidu apod. Aktuální verze rozšíření tyto operace neimplementuje.

Vzdálenost by mohla být implementována jako nejmenší vzdálenost mezi body daných dvou geometrií. Jednoduchý algoritmus, který porovná každý bod z každým jiným by měl kvadratickou složitost. Výpočet centroidu nebo plochy lze provést s využitím triangulace, kterou čtenář může nalézt ve zdroji [30].

5.7 Aplikační rozhraní

Tato kapitola čtenáře seznamuje s vrstvou aplikačního rozhraní rozšíření. Obsahuje popis standardu SQL/MM Spatial a na něm založeném objektovém modelu geometrických typů. Bude také zmíněna problematika kontroly pravidel geometrie při modifikaci. Jelikož je standard SQL/MM Spatial určen pro SQL, budou popsány implementační konvence pro programovací jazyk Java.

5.7.1 Standard SQL/MM Spatial

Cílem standardu ISO/IEC 13249 SQL/MM je standardizovat rozšíření pro multimédia a aplikačně specifické balíčky v SQL. Samotný standard je rozdělen do několika částí. V této práci nás hlavně zajímá část třetí, která definuje prostorové datové typy a operace s využitím postrelačního databázového systému a SQL.

Prostorové datové typy jsou definovány objektovým modelem s využitím dědičnosti. Všechny typy reprezentují geometrie ve 2 nebo 3-dimenzionálním prostoru (rozšíření pracuje zatím s 2 dimenzemi). Pro datové typy *ST_Geometry*, *ST_Surface* a *ST_Curve* nelze vytvořit instance. *ST_Geometry* je abstraktní datový typ na vrcholu hierarchie prostorových datových typů standardu.

Hodnoty typu *ST_Point* jsou 0-dimenzionální geometrie reprezentující jeden bod. Každý bod je definován souřadnicí (x,y) v souřadném systému. Hodnoty typu *ST_MultiPoint* jsou kolekcemi hodnot typu *ST_Point*, tedy kolekcemi bodů. Přičemž se body mohou překrývat.

Křivky jsou 1-dimenzionální geometrie. Standard rozlišuje mezi *ST_LineString*, *ST_CircularString* a *ST_CompoundCurve*. *ST_LineString* je definována sekvencí bodů, které určují referenční body křivky. Výsledná křivka je získána lineární interpolací mezi jejími referenčními body. *ST_CircularString* je podobný *ST_LineString*, ale místo lineární, používá kruhovou interpolaci mezi referenčními body. Každý kruhový segment je složen právě ze tří referenčních bodů. První určuje začátek segmentu, druhý leží někde mezi prvním a třetím na segmentu a třetí určuje konec segmentu. Pokud hodnota obsahuje více segmentů, potom koncový bod jednoho segmentu je zároveň počátečním bodem dalšího segmentu. Hodnota typu *ST_CompoundCurve* modeluje spojení *ST_LineString* a *ST_CircularString*. Hodnota typu *ST_MultiCurve* reprezentuje kolekci hodnot typu *ST_Curve*. Hodnota typu *ST_MultiLineString* modeluje kolekci hodnot typu *ST_LineString*. Nutno podotknout, že aktuální verze rozšíření neimplementuje datový typ *ST_CircularString* a nelze s ním tedy počítat ani u ploch..

Plochy jsou 2-dimenzionální geometrie, které jsou opět definovány sekvencí bodů. Hranicí každé plochy je jedna křivka, nebo množina křivek, obsahuje-li plocha díry. Hodnota typu *ST_CurvePolygon* je obecná plocha. Hodnota typu *ST_Polygon* dodržuje podmínku, že hranice jsou lineární okruhy. To znamená, že mohou být tvořeny pouze hodnotami typu *ST_LineString*. Hodnota typu *ST_MultiSurface* modeluje kolekci hodnot typu *ST_Surface*, které musí tvořit oddělené plochy. Hodnota typu *ST_MultiPolygon* modeluje kolekci hodnot typu *ST_Polygon*.

Zmíněná hierarchie přímo nereprezentuje prázdnou geometrii, která může být například výsledkem nějaké operace. Standard dovoluje, aby hodnoty každého instanciovatelného typu mohly nabývat prázdné geometrie. Prázdná geometrie je reprezentována prázdnou množinou bodů.

Datové typy obsahují množinu metod, které se dědí napříč hierarchií. Metody by se daly rozdělit do 4 kategorií:

1. metody, které konvertují mezi geometrií a externím datovým formátem,
2. získávají vlastnosti geometrie,
3. porovnávají dvě geometrie na základě jejich prostorového vztahu a
4. generující nové geometrie.

Standard podporuje několik externích datových formátů pro prostorové data. Well-known text representation (WKT), well-known binary representation (WKB) a geography markup language (GML). Pro práci s těmito formáty slouží metody kategorie 1. Metody kategorie 2 jsou jako jiné metody v objektových modelech, které získávají specifické vlastnosti objektů. Tyto mohou například získat dimenzi geometrie apod. Do metod kategorie 3 spadají predikáty z prostorové algebry. Metody kategorie 4 reprezentují geometrické relace nebo výpočetně náročné operace prostorové algebry.

Standard specifikuje také podporu pro určování směru a úhlu. Za tímto účelem definuje typy *ST_Direction* a *ST_Angle*. Aktuální verze rozšíření tyto dva typy neimplementuje.

Standard přímo nespecifikuje koncept souřadnicových referenčních systémů. Pouze obsahuje koncept jejich zakomponování. Pro použití je doporučován již zmiňovaný standard ISO 19111, který popisuje jejich koncept.

Zájemce o hlubší studium tohoto standardu bych rád odkázal na zdroje [38] a [39].

5.7.2 Objektový model

Objektový model vychází ze zmiňovaného standardu SQL/MM Spatial. Model je popsán diagramem tříd, který lze vidět na Diagramu 3. Zobrazuje většinu geometrických typů, kromě podtypů kolekce geometrií. Diagram ukazuje dědičnosti, agregace a použití generických typů. Pouze pro připomenutí, rozhraní *Coordinates* pochází z knihovny *JScience*.

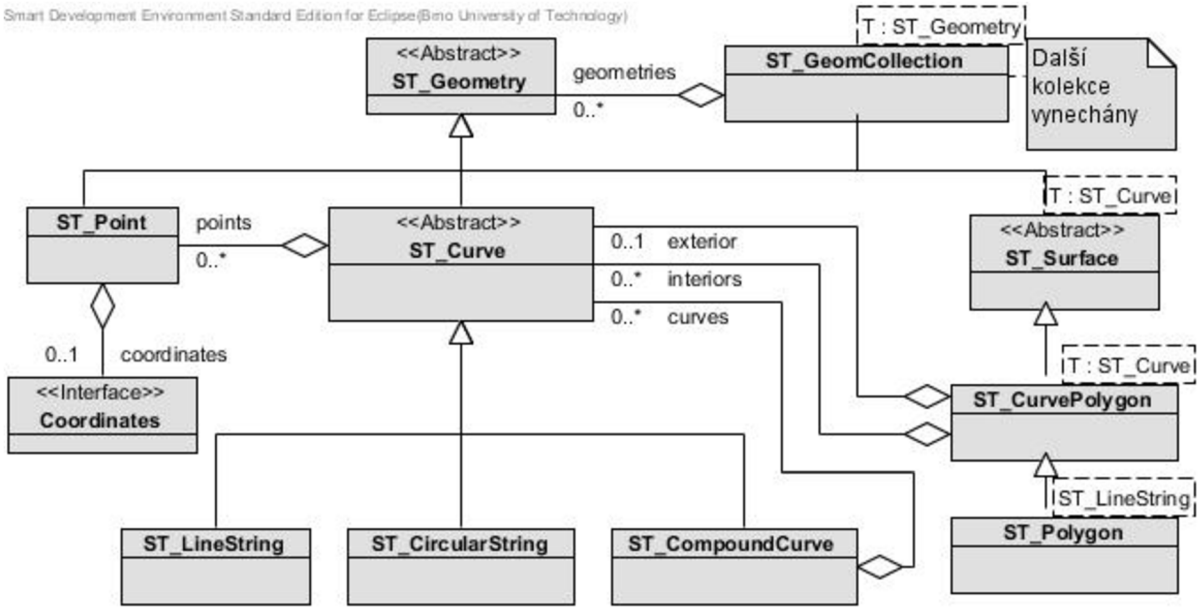


Diagram 3: Objektový model geometrických typů.

Pro vytváření instancí geometrií je použit návrhový vzor továrna. Instance geometrických typů by tedy měly být vytvářeny pomocí továrny, která je v rozšíření implementována. Vývojář by neměl

vytvářet instance geometrií přímo přes konstruktor. Továrna se vytváří s parametrem továrny na souřadnice, která bude popsána dále. Znamená to tedy, že všechny geometrie vytvořené danou instancí továrny používají stejnou továrnu na souřadnice. To také znamená, že geometrie jsou ve stejném souřadnicovém systému. Továrna na geometrie obsahuje metody pro vytváření všech typů geometrií ať už prázdných nebo na základě vstupních hodnot.

Příklad objektového grafu instance geometrie je zobrazen na Diagramu 4. Každý objektový graf obsahuje jedinečné objekty, tzn. žádné objekty nejsou sdíleny. Prochází-li například dvě křivky jediným totožným bodem, je tento bod reprezentován různými objekty pro každou křivku. Toho je dosaženo kopírováním objektů, při jejich přiřazování do objektů geometrií. Tento způsob je výhodný také z pohledu perzistence.

Každý geometrický typ musí splňovat pravidla stanovená standardem. Pouze pro představu, díra daného polygonu musí ležet uvnitř tohoto polygonu, křivka nesmí obsahovat totožné, po sobě jdoucí, body apod. Tyto pravidla musí být kontrolována při každé modifikaci instance geometrie. To sebou nese určitou režii. Obecný princip této kontroly bude vysvětlen také na následujícím diagramu objektů nesoucího název Diagram 4, který zobrazuje instanci geometrie typu polygon.

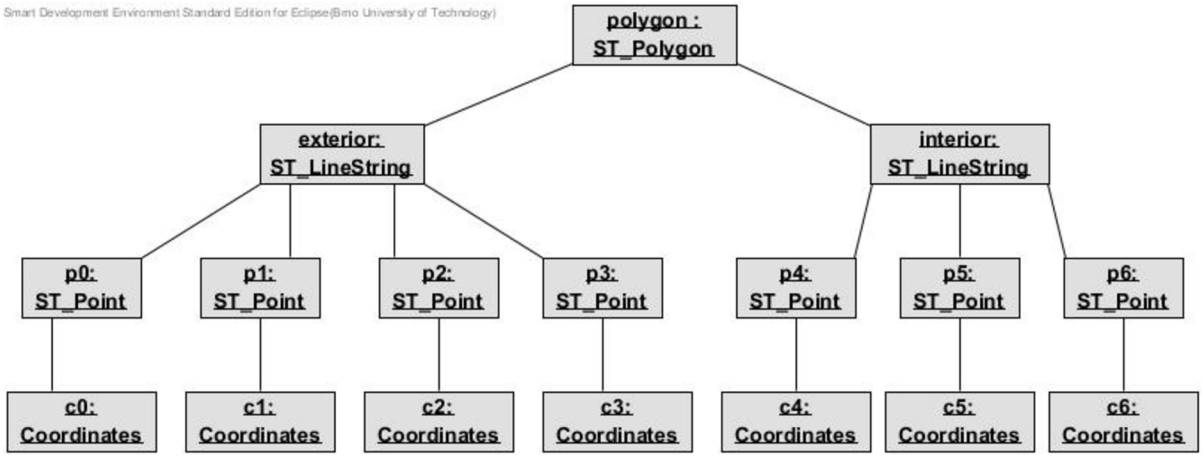


Diagram 4: Objektový graf instance polygonu.

Třída každého typu geometrie obsahuje kontrolu pravidel pro něj stanovených. Tato kontrola je prováděna vždy při změně geometrie. Přičemž změna může nastat buď přímo v objektu dané geometrie, nebo také v objektu, který se nachází níže v objektovém grafu. Například změna souřadnice bodu, který je součástí křivky, která je součástí polygonu. V tomto případě, je nutné informovat objekty výše v geometrickém grafu, aby také provedly kontrolu. To znamená, že kontrolu musí provést také křivka a následně i polygon. Dojde-li na jakékoli úrovni k porušení stanovených pravidel, je vyvolána výjimka a změna není provedena. Ještě bych rád podotknul, že jediná možnost, jak modifikovat bod, je přiřazení nové instance souřadnice, protože rozhraní souřadnice z knihovny *JScience* nedovoluje modifikaci souřadnice, tzn. souřadnice je neměnná. Také je nutné zmínit, že metoda *isValid*, kterou standard definuje, slouží k jinému účelu. Protože validita geometrie je dána jinými pravidly.

Rozšíření implementuje převážnou většinu standardních metod, které definuje standard. Některé metody jsou implementovány jen v určitých typech a některé zatím nejsou implementovány. Rád bych zde zmínil, které metody implementovány nejsou a nebo jen částečně. Typ *ST_CompoundCurve* neobsahuje implementaci metody *isSimple*. Křivky, obecně, neimplementují metodu *getLength* a plochy neimplementují metody *getArea*, *getPerimeter*, *getCentroid* a *getPointOnSurface*. Všechny ostatní metody jsou implementovány.

5.7.3 Konvence pro implementaci standardu

Jelikož je standard určen primárně pro SQL, stanovil jsem určité implementační konvence pro jazyk Java. Následuje jejich stručný výčet:

- názvy metod začínají malým písmenem a není použita předpona *ST_*
- názvy privátních členů začínají malým písmenem a není použita předpona *ST_Private*
- namísto datového typu *smallint* je použit datový typ *int*
- u predikátových metod a členů je použit datový typ *boolean* namísto *int*
- pro některé metody patřící do 2. kategorie z předešlé kapitoly je použita předpona *get*

5.8 Souřadnicové systémy

V této kapitole bude čtenář seznámen s vrstvou zapouzdřující souřadnicové systémy. Standard SQL/MM Spatial přímo nespecifikuje koncept souřadnicových systémů. Pouze obsahuje koncept jejich zakomponování, přičemž pro použití doporučuje standard ISO 19111, jak již bylo řečeno. Implementace rozšíření za tímto účelem využívá knihovnu *JScience*. Z této knihovny je použito souřadnicové rozhraní. Díky tomu lze použít různé typy souřadnic a souřadnicových systémů, které toto rozhraní implementují. Samotná *JScience* obsahuje základní výčet, který je dán zmiňovaným standardem. Pro podporu základního kartézského prostoru obsahuje rozšíření generické souřadnice pro 2D (a případně 3D) spolu s generickými souřadnicovými systémy. Jedná se o třídy *Generic2D_Coordinates* a *Generic2D_CRS* (podobně pro 3D). Pro tyto třídy je implementována továrna na souřadnice, která je součástí továrny na geometrie, jak již bylo řečeno. Aby mohly být použity další typy souřadnic, je nutné implementovat továrnu, která je podporuje. Za tímto účelem je opět použit návrhový vzor strategie.

6 Vývoj integrační vrstvy

Kapitola čtenáře detailně seznámí s perzistencí geometrických typů z knihovny, která byla popsána v předchozí kapitole. Obsahuje popis návrhu a implementace vrstvy perzistence a dotazování prostorových dat z databáze db4o.

V počátečních fázích vývoje jsem hledal inspiraci pro perzistenci v podobných řešeních. Nalezl jsem jediné životaschopné, které nese název db4o2D [40]. Toto řešení bylo postaveno na již staré verzi 6, databáze db4o a údajně používalo nízko-úrovňové rozhraní *TypeHandlers* pro implementaci perzistentní strategie pro geometrické typy. Tento přístup již není v nové verzi db4o nutný, protože již obsahuje strategie pro ukládání typů, ze kterých jsou geometrie složeny. Jako geometrickou knihovnu používalo JTS. Tedy toto řešení bylo postaveno na standardu SFS. Zdá se, že se jedná pouze o experiment, který se v praxi příliš, nikoli však vůbec, nepoužívá. Bohužel jsem neměl možnost nahlédnout do zdrojových kódů a nebylo tedy možné se nijak více inspirovat. Perzistenci jsem tedy navrhl a implementoval podle svého uvážení.

6.1 Perzistence

Pro dosažení perzistence je implementována abstraktní třída nazvaná *Persistent*, ze které dědí každý typ, který má být perzistentní. Vrstva perzistence je těsně svázána s databází db4o. Používá zpětná volání, transparentní aktivaci a perzistenci a konfiguraci databáze.

V návrhu perzistence jsem se snažil, aby mohl vývojář použít techniky, na které je zvyklí. Aby nemusel při ukládání geometrie ukládat každý objekt z objektového grafu zvlášť, ale aby mohl použít jednořádkovou perzistenci. Jednořádková perzistence znamená, že je jediným příkazem, na jednom řádku zdrojového kódu, uložen celý objektový graf geometrie. Z pohledu návrhu by bylo mnohem výhodnější, kdyby byla vrstva nezávislá na databázi, to se mi ovšem v současné verzi nepodařilo. Návrh by šel ovšem upravit například přidáním vrstvy adaptéru databáze.

Abstraktní třída *Persistent* zapouzdřuje každému perzistentnímu objektu jedinečný identifikátor, což je běžné spíše v relačních databázích. Má to ovšem svá opodstatnění. Identifikátor je z výhodou použit při udržování konzistence ve zpětných voláních a je nezbytnou součástí objektů, které se vyskytují v rozpojeném prostředí. Datovým typem identifikátoru je *UUID*, který je databází db4o podporován.

Rozpojené prostředí je prostředí webových aplikací. V tomto prostředí vzniká problém, který je potřeba řešit. Typické chování je takové, že je objekt získán z databáze a poslán klientské straně, kde uživatel tento objekt aktualizuje. Objekt je následně poslán zpět na server. Kontejner objektů ovšem tento objekt nezná a pokud by se jen jednoduše uložil, způsobilo by to, že bude vytvořen nový objekt namísto aktualizace stávajícího originálu. Řešení spočívá na straně serveru, kde se musí po přijetí objektu z klientské strany, získat originál z databáze a poté se musí spojit změny z modifikovaného objektu a tohoto originálu. Poté je možné aktualizovat originál v databázi. Díky tomu je potřeba nějakým způsobem identifikovat objekty. Je k tomu použit již zmiňovaný identifikátor objektů. Spojení změn lze provádět ručně nebo lze použít existující nástroje, například nástroj *Dozer*. Více se o tomto problému, může čtenář dozvědět ve zdroji [7]. Aktuální verze rozšíření obsahuje návrh podpory pro toto prostředí. Za tímto účelem je poskytnuto rozhraní strategie nazvané *GeometryObjectMerger*. Obsahuje metodu, která musí zajistit spojení změn objektu, který přišel z klientské strany a originálního objektu z databáze. Rozšíření také obsahuje třídu *SimpleGeometryObjectMerger*, která toto rozhraní implementuje, ale není zcela hotova.

6.1.1 Zpětná volání

Zpětná volání jsou použita pro udržení konzistence databáze při aktualizaci a odstraňování objektů. Za tímto účelem obsahuje abstraktní třída *Persistent* výchozí implementaci metod *handleUpdating* a *handleDeleting*, které jsou napojeny na příslušná zpětná volání v konfiguraci. Tyto metody mohou být přetíženy ve zděděných objektech, je-li nutné implementovat jiné chování. První zmiňovaná metoda je použita pro uchovávání objektů, které je nutné odstranit při aktualizaci geometrie. Toto chování je nutné například při změně objektu souřadnice bodu, který je součástí křivky. Druhá zmiňovaná metoda je použita pro kaskádové odstraňování objektů, kdy je při odstranění geometrie odstraněn její celý objektový graf.

6.1.2 Transparentní aktivace a perzistence

Aktivační strategie by byla pomocí nastavování aktivační hloubky příliš složitá. Podobně složité by bylo nastavit i hloubku aktualizace. Díky tomu je použita transparentní aktivace a s tím související transparentní perzistence. To sebou nese určitou implementační režii. Abstraktní třída *Persistent* implementuje rozhraní *Activatable* a aktivace je provedena v každé metodě typu *getter* a *setter*, je-li to žádoucí. Transparentní aktivace a perzistence je zapnuta v konfiguraci databáze. S tímto faktem by měli počítat vývojáři, kteří by toto prostorové rozšíření používali. Protože, je-li jednou transparence zapnuta, je zapnuta pro celou aplikaci. Výchozí chování je takové, že implementuje-li objekt zmiňované rozhraní, je při aktivaci a perzistenci použita transparence. Pokud ne, pak je aktivován celý objektový graf. Více informací o transparentní aktivaci a perzistenci a jejich chování může čtenář nalézt ve zdroji [7].

6.1.3 Konfigurace db4o

Z předchozího textu vyplývá, že konfigurace db4o je použita k tomu, aby uvedla předešlé aspekty do činnosti. Pro úplnost, zapíná podporu identifikátorů *UUID*, napojuje příslušné metody na zpětná volání a zapíná transparentní aktivaci a perzistenci. Pro jednoduché použití je toto chování zapouzdřeno ve třídě *SpatialSupport*, kterou lze přidat jedním příkazem do konfigurace při otevírání kontejneru db4o.

6.2 Dotazování prostorových dat

Pro dotazování prostorových dat v db4o není potřeba navrhovat nový prostorový dotazovací jazyk, nýbrž je možné využít existující dotazovací jazyky db4o, protože se zdá, že tyto objektově orientované dotazovací jazyky implicitně podporují prostorové dotazovací jazyky.

Následující jednoduchá ukázka dotazu (viz Zdrojový kód 4) zobrazuje predikát vracející pravdivostní hodnotu *pravda* pro všechny zákazníky, kteří se jmenují *John* a jejichž umístění je vzdáleno od *určitého místa 1 km*.

```
public boolean match(Customer customer) {
   return customer.getLocation().getDistance(place, SI.METRE) < 1000 &&
         customer.getName().compareTo(„John“) == 0;
}
```

Zdrojový kód 4: Ukázka prostorového dotazu 1.

Zdrojový kód 5 zobrazuje predikát, který je použit pro vrácení všech *velkých měst*, kterými protéká *určitá řeka*.

```
public boolean match(Town town) {
   return town.getSurface().isIntersects(river) &&
         town.getPopulation() > 10000;
}
```

Zdrojový kód 5: Ukázka prostorového dotazu 2.

Z obou zmiňovaných dotazů lze vidět, že lze kombinovat dotazy na prostorová i klasická data v jediném dotazu. Rád bych zde zopakoval zásadní problém s výkonností dotazů díky nemožnosti integrovat vlastní indexační strategii do databáze db4o. Díky tomuto problému je praktické použití tohoto prostorového rozšíření pro db4o velmi složité, ba snad nemožné.

7 Testování prostorového rozšíření

V této kapitole bude čtenář seznámen s postupem, jak bylo rozšíření testováno a bude mu poskytnut rozsah jednotlivých testů pro vrstvy architektury. Všechny testy úspěšně prošly.

Při vývoji rozšíření byl použit testovací nástroj *JUnit* verze 4. Cyklus vývoje neodpovídal přesně vývoji řízeném testy, ale použil jen jeho určité části. Byly psány jednotkové testy za účelem ověření validity. Z toho plyne samozřejmě výhoda při refaktorizaci. Kdykoli došlo k úpravě kódu, musela projít daná sada testů. Testy byly ale většinou psány až po napsání kódu, který testovaly. Nyní bude popsáno co, a v jaké míře bylo testováno.

Jednotkové testy pro jádro a základní algoritmy pokrývají co nejúplnější množinu možností. To znamená, že by měla být plně ověřena funkčnost jádra a základních algoritmů.

Pro vrstvu topologie byly napsány pouze základní testy, které ověřují korektnost pro úzkou skupinu možností. Ovšem i přes to, by měla být tato vrstva korektní. Jednotlivě byly testovány všechny fáze vytváření topologického grafu geometrie. Od transformace, přes noding až po značkování.

Pro testování predikátových operací byl použit příklad, na kterém byla vysvětlována vrstva topologie.Tedy dva vzájemně se protínající čtverce.

Na aplikační vrstvě byla testována práce s geometrickými typy. Jako modifikace, kontrola pravidel apod. Nejsou ovšem pokryty všechny geometrické typy, ale pouze body a křivky.

Byla testována také perzistence. Za tímto účelem byl použit následující princip. Před samotným spuštěním testu se vytvořil soubor databáze, který byl po ukončení testu vymazán. V samotném průběhu se testovalo vytvoření, modifikace a odstranění instancí v databázi a kontrolovaly se správné počty objektů, které tvoří objektový graf geometrie. Tímto přístupem byla ověřena konzistence databáze. To znamená, že například po odstranění polygonu byly odstraněny také objekty křivek, bodů a souřadnic, které jej tvořily. Nebo při odstranění křivky definující díru polygonu, byl odstraněn její objekt spolu s ostatními objekty, které ji tvořily apod.

Nebyly provedeny výkonnostní testy, které by srovnávaly mé řešení geometrické knihovny například s knihovnou JTS. Tyto testy by téměř určitě prokázaly, že mé řešení je mnohem méně výkonné. To se dá předpokládat hlavně díky použitým algoritmům.

8 Případ použití

Díky malé výkonnosti implementace nepředpokládám široké praktické využití tohoto prostorového rozšíření db4o. Avšak, i přes to, tato kapitola poskytuje čtenáři postup jak použít toto rozšíření.

V první řadě, je nutné zapnout podporu tohoto rozšíření v konfiguraci databáze db4o. To se provede následujícím způsobem (viz Zdrojový kód 6).

```
configuration.common().add(new SpatialSupport());
```

Zdrojový kód 6: Zapnutí podpory pro prostorové rozšíření.

Jakmile je podpora zapnuta, je nutné se rozhodnout, jaký chceme použít souřadnicový systém. Za tímto účelem se musí vytvořit instance továrny na souřadnice, která tento souřadnicový systém podporuje (současná verze obsahuje jedinou továrnu na kartézské 2D souřadnice). Následně je možné vytvořit továrnu na geometrii, která bude použita k tvorbě instancí geometrií. Zdrojový kód 7 zobrazuje tento krok.

```
CoordinateFactory coordFactory = new CoordinatesFactoryGeneric2D();
GeometryFactory factory = new GeometryFactory(coordFactory);
```

Zdrojový kód 7: Vytvoření továrny na geometrie.

Nyní je možné vytvořit instanci geometrie. Zdrojový kód 8 ukazuje vytvoření instance křivky, která se skládá ze dvojice bodů. Poslední řádek představuje jednořádkovou perzistenci, kdy se uloží celý objektový graf křivky.

```
ST_Point p1 = factory.createPoint(new double[] { 1.1, 1.1 });
ST_Point p2 = factory.createPoint(new double[] { 2.3, 2.3 });
ST_LineString line = factory.createLineString(Arrays.asList(
    new ST_Point[] { p1, p2 }));
db4o.store(line);
```

Zdrojový kód 8: Vytvoření geometrie.

Zdrojový kód 9 zobrazuje příklad aktualizace námi vytvořené křivky, kterou jsme získali nějakým dotazem na databázi. Nejdříve se získá počáteční bod křivky, u kterého se změní souřadnice. Poté se ještě přidá nový bod na konec této křivky. Posledním řádkem se tato změna stane perzistentní, přičemž je zajištěna konzistence databáze, tzn. je odstraněn objekt původní souřadnice prvního bodu a je vytvořen nový bod a souřadnice, které byly přidány na konec.

```
ST_Point p1 = line.getStartPoint();
p1.setCoordinate(new double[] { 0.0, 0.0 });
line.addPoint(someNewPoint);
db4o.store(line);
```

Zdrojový kód 9: Modifikace geometrie.

Třída každé geometrie obsahuje metody pro získání jejího obsahu. Například bod obsahuje metodu pro získání souřadnice, křivka pro získání bodů a polygon pro získání vnějších a vnitřních hranic. Zdrojový kód 10 ukazuje možnost použití těchto metod.

```
for (ST_Point point : line.getPoints()) {
   someBean.printPoint(point);
}
```

Zdrojový kód 10: Použití geometrie.

Jakmile chceme instanci geometrie odstranit z databáze, provede se to voláním jediné metody na kontejneru databáze, jak je ukázáno na následujím Zdrojovém kódu 11.

```
db4o.delete(line);
```

Zdrojový kód 11: Odstranění geometrie.

Závěr

Tato závěrečná kapitola poskytuje čtenáři zhodnocení práce. Pokud by měl čtenář potenciální zájem o implementaci vlastního prostorového rozšíření objektové databáze, může využít některá z uvedených doporučení, která mu pomohou s rozhodováním, jako například co udělat a čemu se naopak vyhnout. Úplný závěr této práce patří návrhům na další vývoj tohoto prostorového rozšíření db4o.

Projekt nese název *db4o-sp* a je hostován na adrese *http://sourceforge.net/projects/db4o-sp/*, kde je volně dostupný repositář zdrojových kódů a blog s různými informacemi ohledně vývoje.

Zhodnocení

Při práci na tomto prostorovém rozšíření jsem se seznámil s disciplínou nazvanou exaktní výpočetní geometrie. Získal jsem mnoho informací o geometrických výpočtech. Také jsem se podrobně seznámil s objektovými a prostorovými databázemi. Nabité znalosti jsem se snažil použít při implementaci prostorového rozšíření objektové databáze db4o. Osobně si myslím, že by tato práce mohla sloužit jako inspirace pro další vývojáře a nepředpokládám tedy využití v praxi.

Doporučení

Pokud by byl čtenář potenciální vývojář hledající inspiraci pro implementaci vlastního prostorového rozšíření objektové databáze, rád bych mu poskytl následující postřehy z mého vývoje, které mu mohou pomoci.

Při výběru databáze je o ní potřeba zjistit co nejvíce užitečných informací. Zaměřit se hlavně na body rozšíření, jako integrace vlastní indexační strategie apod.

Při implementaci jádra by měly být použity principy exaktní výpočetní geometrie. Pokud možno by měla být použita robustní knihovna CGAL, která má v době psaní této práce ve vývoji wrapper pro platformu Java. Případně lze použít jiné, jako například LEDA nebo CORE Library. Pokud budou opomenuty problémy s robustností, musí být k tomu stanoven jasný důvod. Například, aplikace nepotřebuje velkou míru robustnosti.

Základní geometrické algoritmy musí být řádně otestovány, aby byla zaručena jejich korektnost, protože je na nich postavena spousta dalších algoritmů.

V implementaci by měl být použit typ algoritmu *PlaneSweep*, případně jiné rychlé algoritmy, kdekoli je to možné.

Integrační vrstva by měla být nezávislá na geometrické knihovně a použité databázi jak jen je to možné. Pokud se toto podaří, má vývojář vysokou šanci zaměnit použité databáze.

Existuje-li možnost, že řešení bude používáno v rozpojeném prostředí, měla by pro toto být poskytnuta podpora ve formě metod pro spojování změn objektů. K tomuto účelu lze použít existující nástroje, jako *Dozer* apod.

Jako poslední doporučení bych chtěl uvést, že je potřeba, aby byla dodržena vysoká použitelnost. To znamená, že je nutné zjistit, jak se chovají potenciální uživatelé a poskytnout jim co nejpřijatelnější používání. U databáze db4o se jedná například o jednořádkovou perzistenci.

Další vývoj

Aktuální verze prostorového rozšíření db4o obsahuje pouze základní funkčnost. Proto přicházejí v úvahu různá rozšíření této práce. Nejedná se ovšem o úplný výčet, ale jen o ty nejdůležitější rozšíření.

Dle pana Egenhofera by měla každá prostorová databáze podporovat grafický vstup a výstup. Standard SQL/MM Spatial definuje tuto možnost. Aktuální verze ji ovšem neimplementuje. Pokud by například byl podporován vektorový formát Scalable Vector Graphics (SVG), lze použít jakýkoli vektorový grafický editor (například *Inkscape*) jako grafické uživatelské rozhraní (GUI) pro manipulaci s geometriemi.

Současná verze zatím přímo podporuje souřadnice ve 2D kartézském prostoru. Bylo by vhodné rozšířit podporu o další typy souřadnic a souřadnicových systémů z knihovny *JScience* implementací dalších továren na souřadnice.

Díky zmiňovaným problémům, mají důležité algoritmy pouze kvadratickou složitost. Pro zopakování zde patří například *noding*. Dalším rozšířením proto může být optimalizace těchto algoritmů. Nejspíše by se muselo reimplementovat jádro. Například postupem, který byl již navržen. Teprve poté by bylo možné použít algoritmy založené na *PlaneSweep*, což by výrazně urychlilo výpočet.

Možná neřešitelným problémem je nalezení cesty k optimalizaci dotazů díky nemožnosti integrovat vlastní indexační strategii do db4o. Tato možnost neexistuje a nejspíše také existovat nebude. Ovšem určitá cesta této optimalizace by měla přece jen existovat.

Jak již bylo zmiňováno, aktuální verze podporuje pouze 2D. Proto se nabízí rozšíření do 3D. Také již ovšem bylo zmiňováno, že toto rozšíření bude možná složité ve vrstvě topologie.

Jako poslední rozšíření, které bych zde uvedl, je osamostatnění integrační vrstvy, aby bylo možné jednoduše měnit databáze pro ukládání geometrických typů. Současná verze má totiž tuto vrstvu těsně svázánu s databází db4o.

Literatura

[1] HRUBÝ, Martin. *Geografické Informační Systémy (GIS)*. Brno, 2006. Studijní opora. FIT VUT v Brně.

[2] GROSSNIKLAUS, Michael a Moira NORRIE. Object-Oriented Databases. *ODBMS.ORG* [online]. 2010 [cit. 2012-05-07]. Dostupné z: http://www.odbms.org/download/eth-oodb-2009.zip

[3] SIGNER, Beat. Introduction to databases: Object and Object-Relational Databases. *ODBMS.ORG* [online]. 2010 [cit. 2012-05-07]. Dostupné z: http://www.odbms.org/download/lecture_12_objectDatabases.pdf

[4] ATKINSON, Malcolm, François BANCILHON, David DEWITT, Klaus DITTRICH, David MAIER a Stanley ZDONIK. The Object-Oriented Database System Manifesto. *School of computer science* [online]. 1989 [cit. 2012-05-07]. Dostupné z: http://www.cs.cmu.edu/afs/cs.cmu.edu/user/clamen/OODBMS/Manifesto/

[5] CATTELL, Rick. *Object data standard:ODMG 3.0*. San Francisco: Morgan Kaufmann, 2000, 280 s. ISBN 15-586-0647-5.

[6] OMG: About OMG. *OMG* [online]. 2011 [cit. 2012-05-07]. Dostupné z: http://www.omg.org/gettingstarted/gettingstartedindex.htm

[7] VERSANT CORPORATION. *Versant Developer Community*: *db4o 8.0 Java Reference doc* [online]. 2011 [cit. 2012-05-07]. Dostupné z: http://community.versant.com/documentation/reference/db4o-8.0/java/reference/

[8] ROSENBERGER, Carl a William R. COOK. Native Queries for Persistent Objects - A Design White Paper. [online]. 2006 [cit. 2012-05-07]. Dostupné z: http://www.cs.utexas.edu/~wcook/papers/NativeQueries/NativeQueries8-23-05.pdf

[9] Pluggable index structure to db4o. In: *Versant developer community*: *Forums* [online]. 2012 [cit. 2012-05-07]. Dostupné z: http://community.versant.com/Forums/tabid/98/aft/11961/Default.aspx

[10] KOLÁŘ, Dušan. *Pokročilé databázové systémy (Prostorové databáze)*. Brno, 2006. Studijní opora. FIT VUT v Brně.

[11] SCHNEIDER, Markus. Computational Geometry on the Grid: Traversal and Planesweep Algorithms for Spatial Applications. *Praktische Informatik Iv*: *D- Hagen* [online]. 1998 [cit. 2012-05-07]. Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.46.3006

[12] GÜTING, Ralf Hartmut a Markus SCHNEIDER. Realms: A Foundation for Spatial Data Types in Database Systems. In: ABEL, David a Beng Chin OOI. *Advances in spatial databases*: *third international symposium, SSD '93, Singapore, June 23-25, 1993 :*

proceedings [online]. New York: Springer-Verlag, c1993. ISBN 3-540-56869-7. Dostupné z: http://dl.acm.org/citation.cfm?id=718759

[13] OPEN GEOSPATIAL CONSORTIUM INC. *OGC® Standards and Specifications*: *OGC* [online]. 2011 [cit. 2012-05-07]. Dostupné z: http://www.opengeospatial.org/standards

[14] Max J. Egenhofer. THE UNIVERSITY OF MAINE. *Spatial Information Science and Engineering* [online]. [cit. 2012-05-07]. Dostupné z: http://www.spatial.maine.edu/~max/

[15] STROBL, Christian. Dimensionally Extended Nine-Intersection Model (DE-9IM). *Encyclopedia of GIS* [online]. 2008, s. 240-245 [cit. 2012-05-07]. Dostupné z: http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/s/Strobl:Christian.html

[16] EGENHOFER, Max. Spatial SQL: A Query and Presentation Language. *IEEE Transactions on Knowledge and Data Engineering* [online]. 1994 [cit. 2012-05-07]. Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.85.1824

[17] GUTTMAN, Antonin. R-trees: a dynamic index structure for spatial searching. *ACM SIGMOD Record* [online]. 1984-06-01, roč. 14, č. 2 [cit. 2012-05-07]. ISSN 01635808. DOI: 10.1145/971697.602266. Dostupné z: http://portal.acm.org/citation.cfm?doid=971697.602266

[18] OOSTEROM, Peter van. Spatial access methods. In: LONGLEY, Paul. *Geographical information systems*: *principles, techniques, management, and applications*. 2nd ed., abridged. Hoboken, N.J.: John Wiley, c2005. ISBN 978-0-471-73545-8.

[19] GÜTING, Ralf Hartmut. An introduction to spatial database systems. *The VLDB journal*: *very large databases : a publication of the VLDB Endowment* [online]. 1994 [cit. 2012-05-07]. ISSN 1066-8888. Dostupné z: http://dl.acm.org/citation.cfm?id=615206

[20] YAP, Chee a Vikram SHARMA. Robust Geometric Computation. [online]. 2004 [cit. 2012-05-07]. Dostupné z: http://www.cs.nyu.edu/sharma/pap/RobustGeometricComputation-EOA.pdf

[21] GREENE, Daniel H. a F. Frances YAO. Finite-resolution computational geometry. In: *27th Annual symposium on foundations of computer science*: *October 27-29, 1986* [online]. Washington, D.C: IEEE Computer Society Press, 1986. ISBN 0-8186-0740-8. Dostupné z: http://dl.acm.org/citation.cfm?id=1382942

[22] LI, Chen, Sylvain PION a Chee YAP. Recent Progress in Exact Geometric Computation. *J. of Logic and Algebraic Programming* [online]. 2004, č. 1, s. 85-111 [cit. 2012-05-07]. Dostupné z: http://cs.nyu.edu/yap/papers/SYNOP.htm

[23] YAP, Chee. *Downloadable Papers* [online]. 2012 [cit. 2012-05-07]. Dostupné z: http://cs.nyu.edu/yap/papers/

[24] MEHLHORN, Kurt a Chee YAP. *Robust Geometric Computation* [online]. 2004, 2012 [cit. 2012-05-07]. Dostupné z: http://cs.nyu.edu/yap/book/egc/

[25] APACHE. *Commons Math* [online]. 2012 [cit. 2012-05-08]. Dostupné z: http://commons.apache.org/math/

[26] NADEZHIN, Dmitry. *Java library for interval computations*: *JInterval* [online]. 2012 [cit. 2012-05-08]. Dostupné z: http://java.net/projects/jinterval

[27] *JScience* [online]. 2012 [cit. 2012-05-08]. Dostupné z: http://jscience.org/

[28] ISO 19111:2007. *Geographic information-Spatial referencing by coordinates*. Dostupné z: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=41126

[29] BRONNIMANN, Hervé, Christoph BURNIKEL a Sylvain PION. Interval Arithmetic Yields Efficient Dynamic Filters for Computational Geometry. *Discrete Applied Mathematics* [online]. 2001, č. 109, s. 25-47 [cit. 2012-05-08]. Dostupné z: http://hal.inria.fr/docs/00/34/42/81/PDF/interval_journal-1.pdf

[30] JENA, Sanjay Dominik a Jackson ROEHRIG. A Java Implementation of the OpenGIS™ Feature Geometry Abstract Specification: ISO 19107 Spatial Schema. *Technical report* [online]. 2007, 0.1 [cit. 2012-05-09]. Dostupné z: http://www.tt.fh-koeln.de/e/itt/staff/jackson_roehrig/download/TechnicalReportISO19107_V01.pdf

[31] PFEIFLE, Julian. Numbers and Points: Overview. In: *Computational Geometry 2011/12* [online]. 2011 [cit. 9.5.2012]. Dostupné z: http://www-ma2.upc.es/~geoc/NumbersAndPoints.pdf

[32] MEHLHORN, Kurt a Chee YAP. Geometric Approaches: Lecture 5. *Robust Geometric Computation* [online]. 2004, 2012 [cit. 2012-05-11]. Dostupné z: http://cs.nyu.edu/yap/bks/egc/09/5geomApproach.pdf

[33] BENTLEY, Jon L. a Thomas A. OTTMANN. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*. 1979, č. 9, 643–647.

[34] HOBBY, John D. Practical Segment Intersection with Finite Precision Output. *Computational Geometry: Theory and Applications* [online]. 1999 [cit. 2012-05-11]. Dostupné z: http://ect.bell-labs.com/who/hobby/93_2-27.pdf

[35] FARSHI, Mohammad. Computing the Overlay of Two Subdivisions. In: [online]. [cit. 10.5.2012]. Dostupné z: http://cs.yazduni.ac.ir/farshi/Teaching/CG3902/Slides/DCEL.pdf

[36] HERRING, John R. OpenGIS Implementation Standard for Geographic information - Simple feature access - Part 1: Common architecture. *OGC*. 2011, OGC 06-103r4.

[37] Point Set Theory and the DE-9IM Matrix. OSGEO. *GeoTools* [online]. 2011 [cit. 2012-05-07]. Dostupné z: http://docs.geotools.org/latest/userguide/library/jts/dim9.html

[38] ISO/IEC 13249-3:2006. *SQL multimedia and application packages - Part 3: Spatial*. 2006.

[39] STOLZE, Knut. *SQL/MM Spatial: The Standard to Manage Spatial Data in Relational*

Database Systems [online]. Leipzig, 2003 [cit. 2012-05-07]. Dostupné z: http://doesen0.informatik.uni-leipzig.de/proceedings/paper/68.pdf

[40] MEIER, Sven, Mathias VATERLAUS a Stefan KELLER. Db4o2D. [online]. 2007 [cit. 2012-05-12]. Dostupné z: http://community.versant.com/Projects/html/projectspaces/db4o2d.html

Seznam použitých zkratek

ACID	Atomicity, consistency, isolation, durability
API	Application interface
CGA	Computation geometry algorithms
CCW	Counter-clockwise
COL	Collinear
CW	Clockwise
DCEL	Doubly Connected Edge List
DE-9IM	Dimensionally Extended Nine Intersection Model
EGC	Exact Geometric Computation
GIS	Geographic Information System
GPS	Global Positioning System
GUI	Graphic User Interface
ISO	Organization for Standardization
JTS	Java Topology Suite
LU	Lower and Upper triangular matrix
MBR	Minimum Bounding Rectangle
MDA	Model Driven Architecture
ODBMSs	Object Database Management Systems
ODL	Object Definition Language
ODMG	Object Data Management Group
ODMs	Object-to-Database Mappings
OGC	Open Geospatial Consortium
OIF	Object Interchange Format
OMG	Object Management Group
OQL	Object Query Language
ORM	Object-Relational Mapper
QBE	Query by Example
S-JTSK	Souřadný systém jednotné trigonometrické sítě katastrální
SFS	Simple Features Specification for SQL
SODA	Simple Object Database Access
SQL	Structured Query Language
SQL/MM	SQL Multimedia and Application Packages
SŘBD	Systém řízení báze dat
SVG	Scalable Vector Graphics
UML	Unified Modeling Language
UTM	Universal Transverse Mercator
UUID	Universally Unique Identifier
WGS-84	World Geodetic System 1984

Seznam obrázků, vzorců, tabulek a zdrojových kódů

Seznam příloh

Příloha 1. CD obsahující aktuální zdrojové kódy a vygenerovanou dokumentaci.