**Slovenská technická univerzita v Bratislave**
**FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ**

Tím č. 16 (GrafIT)

# *INTERAKTÍVNA VIZUALIZÁCIA GRAFOVÝCH ŠTRUKTÚR V 3D PRIESTORE*

Tímový Projekt

Vedúci tímového projektu: Ing. Peter Kapec

Členovia tímu: Bc. Tomáš Hurban, Bc. Milan Laslop, Bc. Jaroslav Prokop, Bc. Matúš Juhas

Akademický rok: 2010/2011

# Obsah

# Úvod

Aby sme lepšie pochopili nejaký systém, je vhodné vizualizovať ho, napríklad vo forme grafu. To nám uľahčuje jeho pochopenie, ale aj nájdenie nových súvislostí, napríklad medzi jeho jednotlivými časťami, modulmi, alebo procesmi. Z tohto hľadiska je vizualizácia vhodná ako pri analýze, tak aj pri overení, či systém funguje správne.

## *Ciele projektu*

Cieľom projektu je vytvorenie komplexného a modulárneho systému na vizualizáciu informácií reprezentovaných grafovými štruktúrami v 3D priestore, pričom sa pokračuje vo vývoji existujúceho systému pridávaním nových vlastností a funkcionality. Vytvorené riešenie zahŕňa interaktívne prehliadanie grafových štruktúr v 3D priestore, čo nie je triviálna úloha. Súčasťou riešenia je aj návrh vhodnej reprezentácie grafových štruktúr a nástrojov pre prácu s nimi. Vytváraný systém je univerzálne použiteľný, avšak je primárne určený na vizualizáciu softvérových artefaktov získaných zo zdrojových kódov, dokumentácie a pod.

## *Štruktúra dokumentu*

Keďže tento projekt nadväzuje na minuloročné tímové projekty, uvádzame v prvej kapitole analýzu existujúcej aplikácie.

V druhej kapitole uvádzame časť analýzy technológii a metód pre rozšírenie existujúceho systému.

V tretej kapitole je uvedená špecifikácia požiadaviek na tímový projekt.

V štvrtej kapitole je podrobne rozobraný vytvorený návrh riešenia jednotlivých častí aplikácie.

V piatej kapitole uvádzame popis vykonaných nefunkcionálnych zmien v aplikácii a overenie prototypu.

# 1 Analýza existujúcej aplikácie

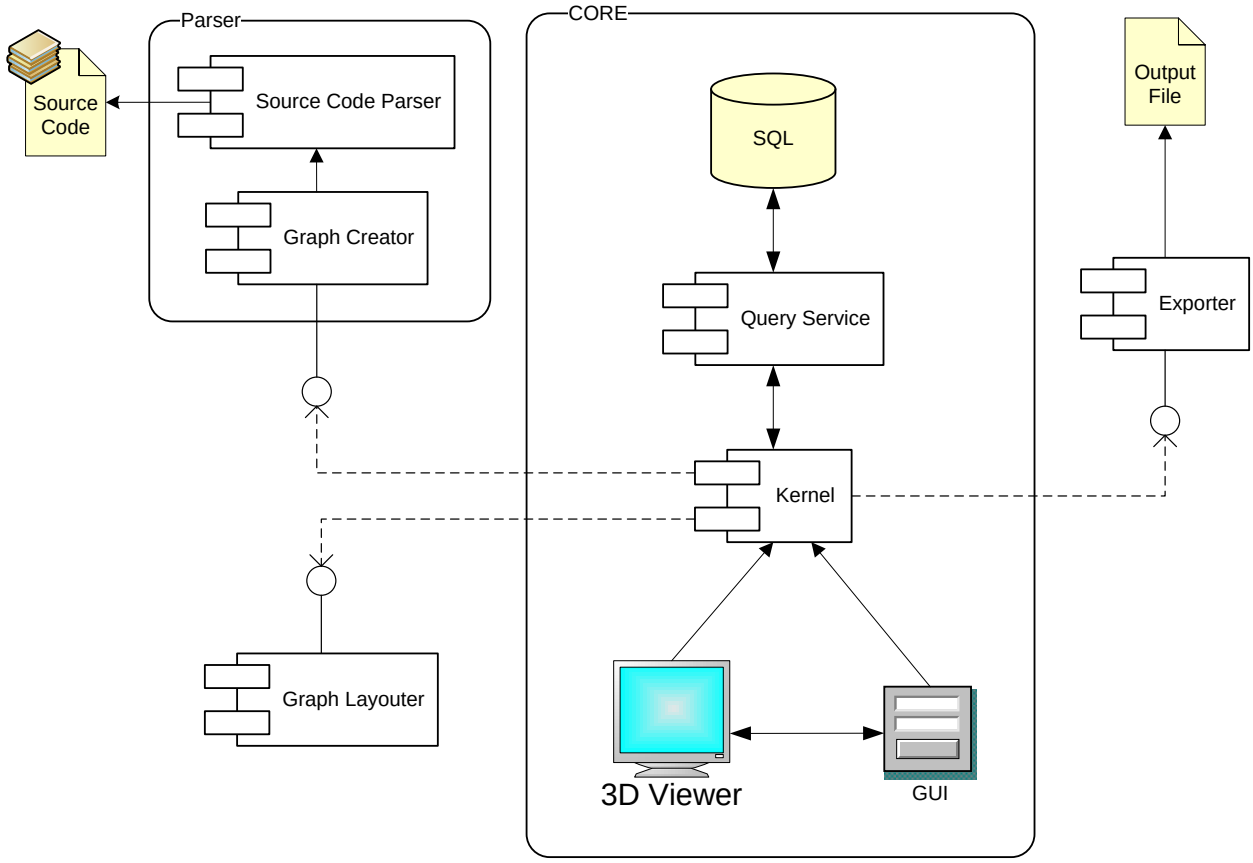
V tejto kapitole uvádzame analýzu existujúceho systému zameraného na 3D vizualizáciu grafových štruktúr. V kapitole uvádzame vývoj architektúry systému od jeho vzniku, vytvorenú databázovú architektúru a grafické rozhranie systému.

## *1.1 Opis architektúry*

V tejto kapitole uvádzame opisy architektúr navrhnutých predošlými tímami riešiacimi rovnaký projekt.

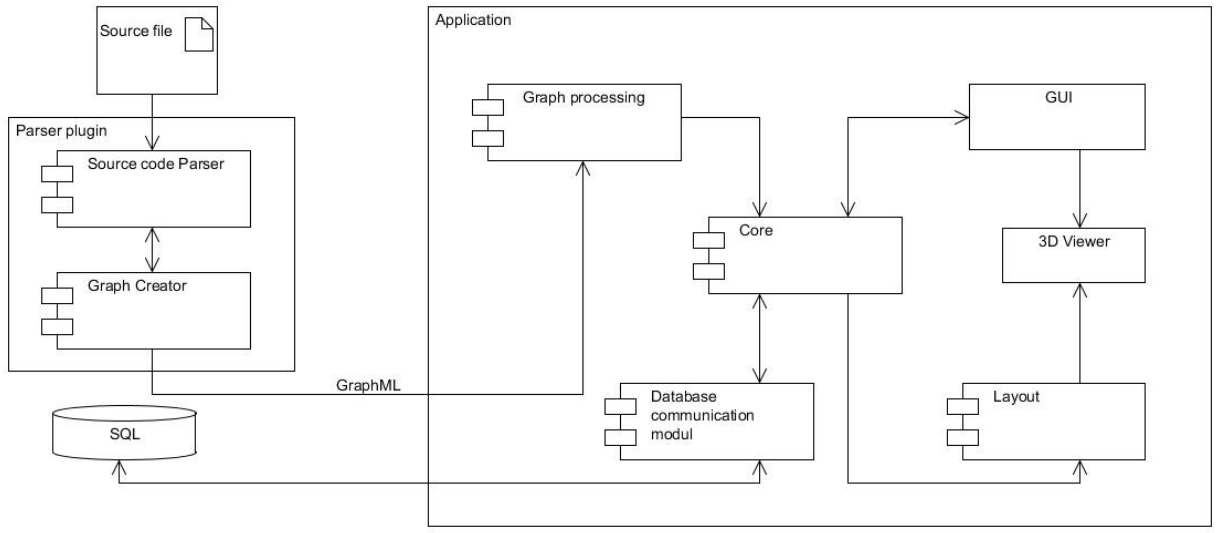
### 1.1.1 Pôvodne navrhovaná architektúra systému (2008/09)

Na nasledujúcom obrázku 1-1 je znázornený pôvodný návrh architektúry systému.




**Obrázok 1-1: Pôvodne navrhovaná architektúra systému (2008/09). [1]**

### 1.1.2 Upravená architektúra systému (2009/10)

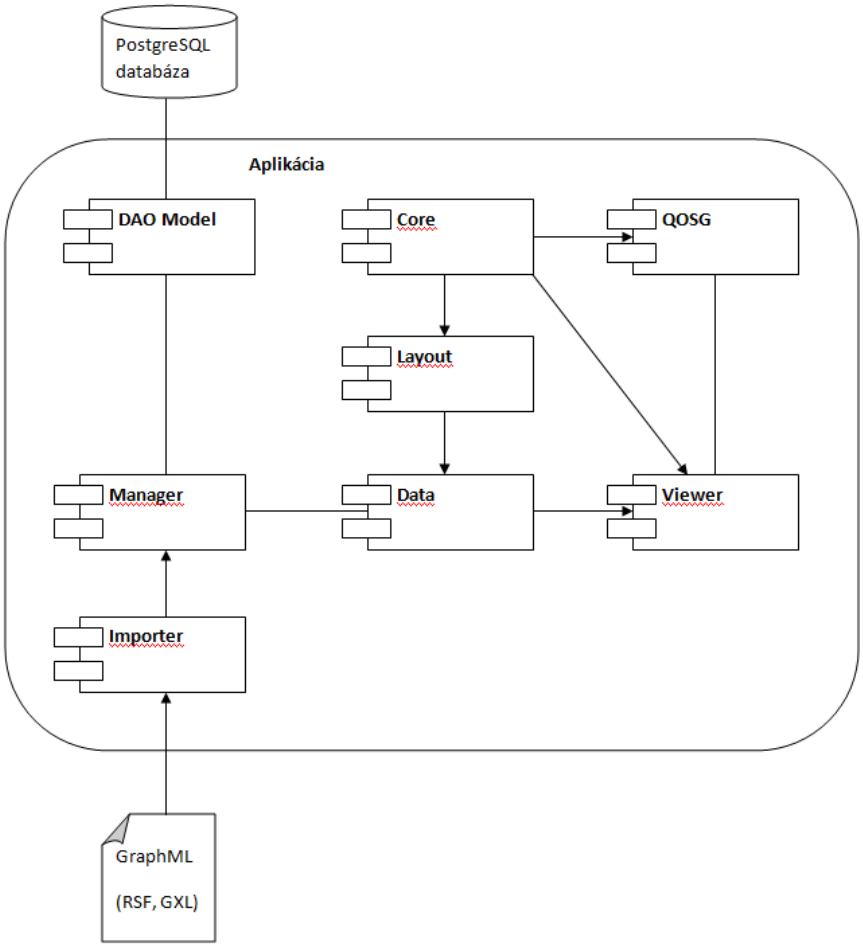



**Obrázok 1-2: Upravená architektúra systému (2009/10). [2]**

Základné rozdiely v novej architektúre (obr. 1-2) oproti pôvodne navrhovanej (obr. 1-1):

- V návrhu novej architektúry chýba modul ***Exporter***, ktorý mal slúžiť na ukladanie grafov do súborov. V súčasnom systéme, je tento modul súčasťou modulu ***Manager***, kde je pripravená funkcia pre export grafov, no nie je implementovaná.
- Layoutovací modul (v pôvodnom návrhu ***Graph Layouter*** modul, v novom ***Layout*** modul) bol pôvodne navrhnutý ako externý modul, v novom návrhu je súčasťou jadra systému.
- Naopak databáza bola pôvodne navrhovaná ako súčasť jadra aplikácie, no v novom návrhu vystupuje ako externý modul. Možno práve preto neboli doposiaľ implementované hlavné funkcie pre ukladanie a vyberanie dát z/do databázy.
- V jadre systému prebehlo tiež niekoľko úprav (ako je vidieť na obr. 1-1 a obr. 1-3). Modul ***Kernel*** v pôvodnom modely sa rozdelil na dva moduly ***Graph processing*** a ***Core*** a zmenili sa závislosti medzi jednotlivými modulmi.

### 1.1.3 Súčasná architektúra systému (2010/11)




**Obrázok 1-4: Súčasná architektúra systému.**

Rozdiely v navrhovanej (upravenej - obr. 1-4) a v súčasnej (existujúce - obr. 1-5j) architektúre systému:

- Podľa [2] mal modul ***Parser plugin*** slúžiť na načítanie vstupných súborov a vygenerovanie GraphML, GXL alebo RSF formátu, ktorý by bol ďalej spracovávaný modulom ***Graph processing***. V existujúcom systéme ***Parser plugin*** modul nie je a ***Graph processing*** modul bol súčasťou modulu ***Manager***, z ktorého bol neskôr vyčlenený do samostatného modulu ***Importer***.
- Modul ***Core*** bol rozdelený na dva moduly – ***Core*** a ***Manager*** (jednotlivé moduly sú stručne opísané nižšie).
- Niektoré moduly boli premenované (pozri časť rozdelenie modulov nižšie).
- K pôvodnému návrhu architektúry pribudlo niekoľko nových modulov (stručný opis jednotlivých modulov je nižšie):
    - *Data*
    - *Noise*

- *Util*
- *Viewer*

V súčasnosti je architektúra systému rozdelená do nasledujúcich modulov (podrobný opis väčšiny modulov je v [2]:

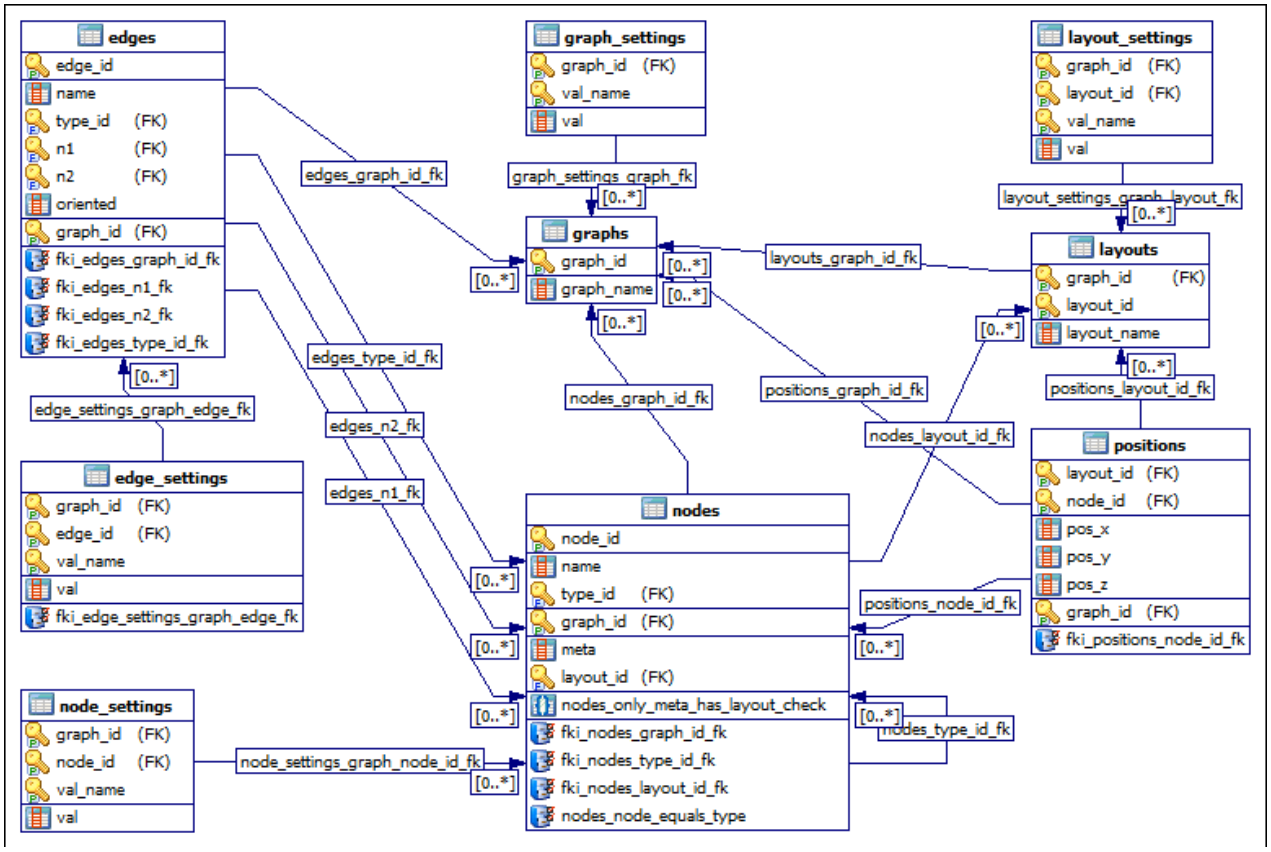
- **Core** – jadro systému. Hlavná obslužná trieda, ktorá poskytuje len minimálnu funkcionalitu.
- **Data** – modul dátovej štruktúry grafu. Jednotlivé triedy reprezentujú jednotlivé prvky grafu (graph, node, edge, type, layout, ...).
- **Importer** (pôvodný názov ***Graph processing***) – v pôvodnom systéme nebol. Bol vyčlenený z modulu *Manager*. Slúži na parsovanie vstupných súborov. V existujúcom systéme pracoval len so vstupnými súbormi typu GraphML, v súčasnosti sa pracuje na pridaní podpory GXL a RSF formátov.
- **Layout** – obsahuje funkcie na rozmiestňovanie uzlov v 3D priestore. Z existujúceho systému bola odstránená knižnica iGraph, pretože nevyhovovala požiadavkám [3] – nedosahovala požadovaný výkon, neumožňovala požadovanú variabilitu nastavení a nebolo možné dosiahnuť dostatočnú kontrolu nad prácou algoritmu.
- **Manager** – modul spravovania grafov. V súčasnosti pracuje len s jedným aktívnym grafom. Pôvodne v ňom bol začlenený modul *Importer*, pretože sa uvažovalo len s podporou formátu GraphML, pričom iné formáty by museli byť transformované najskôr do tohto formátu, aby ich bolo možné načítať.
- **Model** (pôvodný názov ***Database communication modul***) – modul mapovania objektov na databázu (DAO model – data access object). Rozhranie medzi systémom a databázou PostgreSQL. Obsahuje funkcie na pripojenie k databáze a ukladanie a vyberanie prvkov z/do databázy. V súčasnosti obsahuje funkcie, pomocou ktorých je možné ukladať a načítavať len jednoduché grafy.
- **Noise** – slúži na vytvorenie pozadia pri vykresľovaní grafov, aby bol dojem z 3D priestoru realistickejší.
- **OsgQtBrowser** (pôvodný názov ***3D Viewer***) – modul obsahuje funkcie pre mapovanie jednotlivých kláves v systéme a vizualizáciu načítaných grafov.
- **QOSG** (pôvodný názov ***GUI***) – modul, ktorý slúži na vykresľovanie rozhrania aplikácie (vykresľuje hlavné okno aplikácie, okno s nastaveniami, ...).
- **Util** – obsahuje funkcie pre vyčistenie pamäte a konfiguráciu nastavení aplikácie.
- **Viewer** – modul obsahuje funkcie pre pohyb v 3D priestore

## *1.2 Databázový modul*

Dátový model databázy aj aplikácie je podrobne rozpísaný v dokumentácii predchádzajúceho tímu [1].

V súčasnosti je práca s databázou vyriešená čiastočne. Štruktúra databázy už bola vytvorená a taktiež niektoré funkcie pre prácu s databázou a pre pripojenie k databáze už boli implementované.

### 1.2.1 Súčasná štruktúra databázy




**Obrázok 1-6: Fyzický model údajov v databáze. [2]**

Existujúca databáza sa skladá z 9 tabuliek (plus 11 funkcií a 3 sekvencie):

- **edge_settings** (*[PK] int* ***graph_id***, *[PK] int* ***edge_id***, *[PK] chars* ***val_name***, *chars* ***val***)
- **edges** (*[PK] int* ***edge_id***, *chars* ***name***, *int* ***type_id***, *int* ***n1***, *int* ***n2***, *boolean* ***oriented***, *[PK] int* ***graph_id***)
- **graph_settings** (*[PK] int* ***graph_id***, *[PK] chars* ***val_name***, *chars* ***val***)
- **graphs** (*[PK] int* ***graph_id***, *chars* ***graph_name***)
- **layout_settings** (*[PK] int* ***graph_id***, *[PK] int* ***layout_id***, *[PK] chars* ***val_name***, *chars* ***val***)
- **layouts** (*[PK] int* ***graph_id***, *[PK] int* ***layout_id***, *chars* ***layout_name***)
- **node_settings** (*[PK] int* ***graph_id***, *[PK] int* ***node_id***, *[PK] chars* ***val_name***, *chars* ***val***)

- **nodes** (*[PK] int* ***node_id****, chars* ***name****, int* ***type_id****, [PK] int* ***graph_id****, boolean* ***meta****, int* ***layout_id***)
- **positions** (*int* ***layout_id****, int* ***node_id****, double* ***pos_x****, double* ***pos_y****, double* ***pos_z****, [PK] int* ***graph_id***)

### 1.2.2 Implementované funkcie

Funkcie, ktoré sú už implementované pre prácu s databázou:

- pripojenie (a odpojenie) sa k databáze PostgreSQL

Graf:

- načítanie a uloženie údajov z/do tabuľky graph – ID grafu, názov grafu
- zmena mena grafu v databáza

Layout:

- načítanie a uloženie layoutov z/do databázy

Node:

- odstránenie uzla z databázy
- kontrola, či uzol už v databáze existuje

Edge:

- odstránenie hrany z databázy
- kontrola, či hrana už v databáze existuje

Type:

- pridanie typu do databázy
- odstránenie typu z databázy
- kontrola, či typ už v databáze existuje

### 1.2.3 Chýbajúce funkcie

Čo je treba ešte dokončiť, aby bolo možné pracovať s databázou aspoň na základnej úrovni (t. j. ukladanie a načítavanie aspoň jednoduchých grafov):

- Implementácia funkcie pre vkladanie uzlov do databázy
- Implementácia funkcie pre vkladanie hrán do databázy
- Implementácia funkcií pre výber jednotlivých častí grafu (uzly, hrany, názov grafu) z databázy a ich následné spojenie do celistvého grafu
- úprava štruktúry databázy (ako napríklad odstránenie nepotrebných závislostí niektorých prvkov, odstránenie automatického sekvenčného číslovania uzlov, ...)

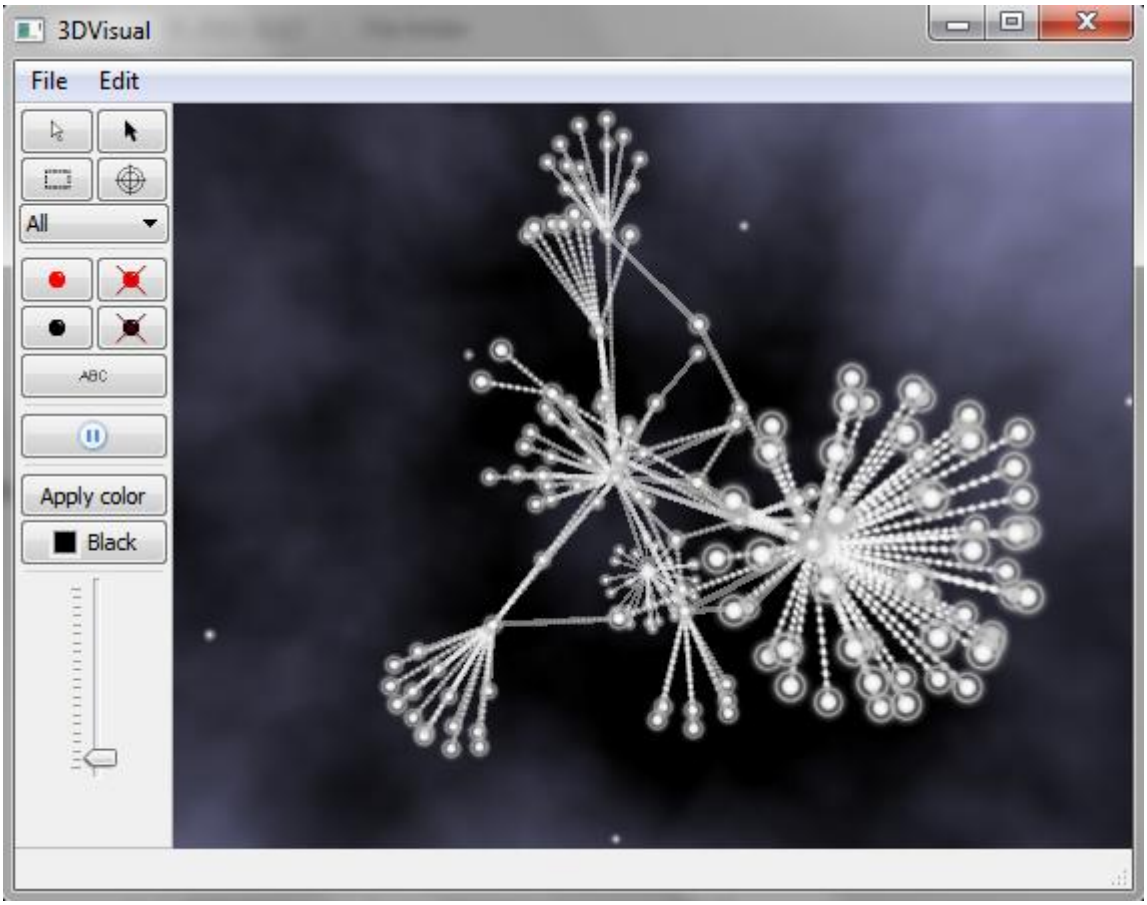
Ďalšie funkcie, ktoré chýbajú:

- ukladanie a načítavanie pozícií jednotlivých uzlov z/do databázy
- ukladanie a načítavanie nastavení jednotlivých prvkov (edge_settings, node_settings, graph_settings, layout_settings) z/do databázy

V pôvodnom systéme sa pracovalo iba s tabuľkou graph a layout, kde sa po načítaní grafu zo súboru uložil názov grafu a názov layoutu.

## *1.3 Používateľské rozhranie*

V nasledujúcej časti je opísané prostredie existujúceho systému a jeho funkcií. V prvej časti je opísané rozhranie aplikácie z pohľadu používateľa, ďalej spôsob rozmiestňovania uzlov v priestore a nakoniec sú spomenuté niektoré chyby, ktoré boli odhalené počas testovania aplikácie. Našim cieľom je v tomto projekte ďalej pokračovať a vylepšovať ho, preto je potrebné poznať, ktoré funkcie sú už v systéme implementované, a ktoré časti je treba vylepšiť.



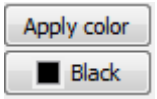

**Obrázok 1-7: Hlavné okno aplikácie.**

Na pohyb v 3D priestore aplikácie môžeme používať nasledovné spôsoby:

1. pravé tlačidlo myši slúži na otáčanie kamery

2. koliesko myši slúži na približovanie a vzďaľovanie pohľadu kamery
3. kurzorové klávesy (hore, dole, vpravo, vľavo) slúžia na pohyb dopredu, dozadu a do strán

**Tabuľka 1-1: Popis funkčných tlačidiel aplikácie.**

| | |
|---|---|
|  | pohyb vybraných uzlov v priestore |
|  | výber jedného uzla |
|   | výber viacerých uzlov a hrán, podľa zvoleného typu - všetko/uzly/hrany |
|  | vycentrovanie pohľadu kamery do stredu 3D priestoru |
|  | pridanie (odstránenie) meta uzlu vybraným uzlom v grafe |
|  | fixovanie (uvoľnenie) vybraných uzlov v grafe, t. j. nebudú sa pohybovať v závislosti od pôsobenia síl ostatných uzlov |
|   | zapnutie/vypnutie zobrazenia popisu uzlov |
|  | pozastavenie/spustenie prekresľovania grafu |
|   | nastavenie zvolenej farby vybraným uzlom |
|  | zmena veľkosti sily pôsobiacej medzi uzlami |

### 1.3.1 Aplikácia umožňuje

- Otvárať a vykresľovať grafy zo súborov GraphML
- Pracovať súčasne len s jedným grafom
- Pohybovať kamerou v 3D ohraničenom priestore okolo zobrazeného grafu
- Vo vykreslenom grafe umožňuje výber jedného alebo viacerých uzlov a hrán, ktoré je možné ďalej presúvať
- V aplikácii sa používajú nasledujúce typy uzlov:
    - *klasické uzly* – pohybujú sa v závislosti od ostatných uzlov pôsobením príťažlivých a odpudivých síl

- *fixné uzly* – klasické uzly, ktoré je možné premiestňovať, no samovoľne nemenia svoju polohu; ak s nejakými vybranými klasickými uzlami pohneme, tieto sa automaticky zmenia na fixné
- *meta uzly* – jednému alebo viacerým uzlom môžeme priradiť meta uzly, ktoré majú fixovanú pozíciu a pôsobia silami na zvolené uzly; tieto uzly sa nenačítavajú z GraphML súborov, sú to len akési pomocné uzly, ktoré pridáva používateľ

- Jednotlivé uzly sa dajú graficky rozlišovať – je možné ich zafarbiť rôznymi farbami
- Zapnutie/vypnutie zobrazovania popisov jednotlivých uzlov
- Možnosť zastavenia/spustenia vykresľovania grafu a nastavenie veľkosti síl pôsobiacich medzi uzlami

### 1.3.2 Zistené chyby v aplikácii (na 32-bit OS Windows 7)

Testovanie aplikácie prebiehalo v prostredí 32-bitového operačného systému Windows 7 s použitím nasledovných knižníc:

- OpenSceneGraph verzia 2.9.9 (development release)
- Qt verzia 4.7.0 pre Visual Studio 2008

Počas testovania aplikácie boli zistené nasledujúce chyby:

- podľa [1] je algoritmus vykonávaný v samostatnom vlákne a vykonávanie prebieha v nekonečnej slučke, pričom rozhranie umožňuje prerušenie vykonávania v ktoromkoľvek momente. Z analýzy testovania existujúcej aplikácie však vyplýva, že nie vždy to platí, najmä pri zobrazovaní rozsiahlejších grafov okno aplikácie prestane reagovať, pričom graf sa ďalej bez problémov vykresľuje,
- nie vždy sa zobrazí okno na načítavanie uložených grafov (File->Load), čo má za následok, že aplikácia prestane reagovať,
- pri vycentrovaní pohľadu kamery, sa kamera nasmeruje do stredu 3D priestoru, no nie vždy sa graf vykreslí presne do stredu.

## *1.4 Rozmiestňovanie uzlov*

Aplikácia využíva Fruchterman-Reingold algoritmus, upravený študentom FIIT STU Jakubom Ukropom, na rozmiestnenie uzlov v trojrozmernom priestore. Princípom tohto algoritmu je pôsobenie príťažlivých a odpudivých síl medzi jednotlivými uzlami.

Princípom algoritmu Fruchterman-Reingold je, že sa uzly správajú ako vesmírne častice. Pôsobia medzi nimi príťažlivé a odpudivé sily. Príťažlivá sila pôsobí iba medzi vrcholmi, ktoré sú navzájom spojené, ale odpudivá sila pôsobí na všetky uzly. Navyše algoritmus obsahuje systém regulovania maximálneho presunu uzlu v jednej iterácii. Analógiou na tento

systém je princíp maximálnej teploty a postupného chladnutia. Každá iterácia obsahuje výpočet odpudivých síl pre všetky uzly, výpočet príťažlivých síl pre každú hranu a výpočet vektora posunutia. Následne sa zníži parameter maximálneho presunu uzla (chladnutie). Algoritmus je ukončený počtom iterácií alebo ukončovacou podmienkou zohľadňujúcou parameter teploty. Výhodou je rýchle sa priblíženie vzdialenosti uzlov k pokojovej vzdialenosti. To je spôsobené príťažlivou silou, ktorá sa úmerne zvyšuje približovaním sa k pokojovej vzdialenosti. Nevýhodou je pomerne veľká časová zložitosť, ktorá ale môže byť optimalizovaná rôznymi spôsobmi. [6]

Použitie algoritmu Jakuba Ukropa prinieslo nasledujúce rozšírenia [6]:

- nastavenie veľkosti grafu
- prístup k medzivýsledkom v priebehu iterácie
- nastavenie flexibility pohybu uzlov
- ovládanie grafu – pozastavenie, oživenie zmrazeného grafu
- ovládanie veľkosti pôsobiacich síl
- funkcia meta uzlov a metahrán
- obmedzenie vplyvu vzdialenejších uzlov
- vylepšená synchronizácia so zobrazovaním

## *1.5 Chýbajúca funkcionalita*

Opis funkcionality, ktorá by sa mala doplniť, vychádza najmä z:

1. dokumentácie predchádzajúceho tímu [1] (časť *8.1 Nedokončené časti projektu*)
2. zadania projektu

Analýza potrebných formátov a nástrojov na zabezpečenie tejto funkcionality je uvedená v ďalších častiach.

### 1.5.1 Uchovávanie grafov v databáze

Je potrebné dokončiť ukladanie grafov do databázy. Podľa časti *8.1.2 Modul mapovania objektov na databázu* je hotový dátový model a práca s grafmi a layoutami. Chýba napríklad uloženie hrán a grafických informácií (napríklad farieb, veľkostí, pozícií) do databázy.

Databáza poskytuje úložisko grafov, teda aplikácia môže pracovať s grafmi vyberanými z tohto úložiska. Preto treba doplniť triedu `Manager::GraphManager`, aby sa dalo pracovať s touto množinou grafov (teda napríklad vybrať si aktívny graf). S tým súvisí aj doplnenie používateľského rozhrania, aby sa dali prezerať grafy uložené v databáze a vybrať z nich aktívny graf.

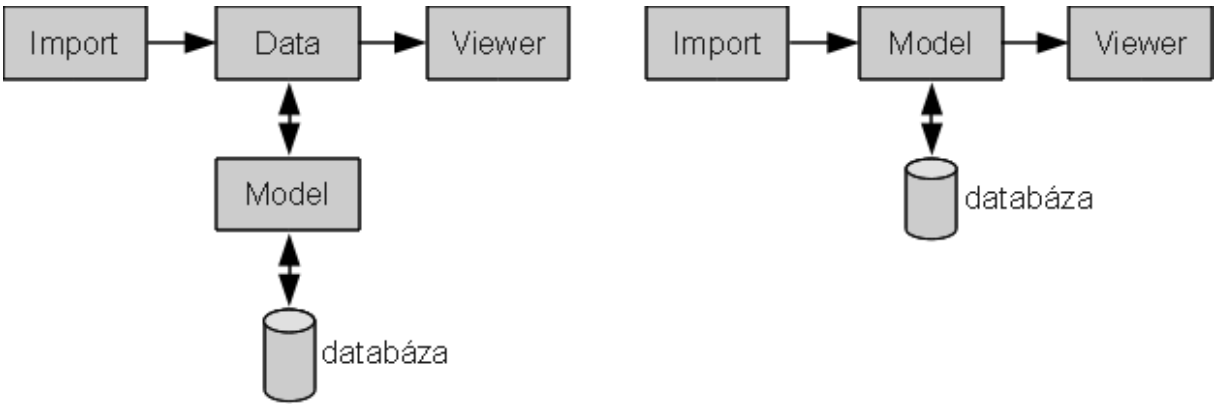
Databáza je navrhovaná iba na „odkladanie" grafov. Aktuálny graf je celý načítaný v pamäti. Štruktúry v pamäti poskytujú základ, s ktorým sa pracuje, a počas ich zmien iniciujú zodpovedajúce zmeny v databáze. Uchovávanie v databáze je teda počas práce s grafom druhoradé a podpora databázy môže byť aj úplne vypnutá.

Je nutné analyzovať potrebu uchovávania celých grafov v pamäti, keďže práca priamo s databázou by mohla priniesť možnosť práce s rozsiahlymi grafmi, ktorých reprezentácia sa nezmestí do pamäte. Vtedy by sa zmenila funkcionalita viacerých modulov:

- import by prebiehal priamo do databázy
- knižnica OpenSceneGraph by načítavala dáta podľa potreby priamo z databázy

V tomto prípade by v pamäti bola len podmnožina grafu, ktorá sa práve spracováva (napríklad iba jeden uzol, ktorého vlastnosti sa práve načítavajú zo súboru).

Rozdiel medzi terajším a navrhovaným prístupom je znázornený na obrázku 1-8. Teraz sa primárne pracuje s pamäťou (vľavo), v novom prístupe by sa pracovalo priamo s databázou (vpravo).




**Obrázok 1-8: Moduly aplikácie a prenos dát medzi nimi. Vľavo aktuálny stav, vpravo možný stav pri práci priamo s dátami v databáze.**

### 1.5.2 Podpora vnorených grafov, multihrán a hypergrafov

Do aplikácie je možné zahrnúť podporu zovšeobecnení grafov:

- vnorené grafy (grafy v uzloch a hranách, môže byť aj viac grafov v jednom uzle alebo hrane)
- multihrany (viacnásobné hrany)
- hypergrafy (hrana môže spájať aj viac ako dva uzly)

Zmeny by zasiahli tieto moduly aplikácie:

- *Import* (rozlišovanie nových prvkov v už používanom *GraphML* formáte, podpora v nových pridaných formátoch)
- *Data* (rozšírenie dát v štruktúrach a metód na prácu s nimi – napríklad hrana môže spájať viac uzlov alebo uzol môže obsahovať v sebe graf)

- *Layout* (prispôsobenie algoritmu, ktorý počíta rozloženie grafu – treba brať do úvahy hrany spájajúce viac uzlov alebo vnorené grafy)
- *Model* (prispôsobenie dátového modelu a metód na prácu s ním)
- *Viewer* (vykreslenie nových prvkov v grafoch)

### 1.5.3 Zlepšenie práce s metauzlami

Metauzly sú nové pridané uzly do grafu, na ktoré layoutovací algoritmus nevplýva, ale slúžia na zmenu umiestnenia uzlov, s ktorými sú spojené hranami.

Práca s metauzlami by sa dala rozšíriť:

- lepšia možnosť voľby pozície metauzla (teraz sa vytvorí v ťažisku vybraných uzlov – s tými je aj následne spojený hranami)
- možnosť voľby vzdialenosti metauzla od kamery

### 1.5.4 Iné spôsoby práce s kamerou

V súčasnosti je kamera ovládaná klávesnicou a myšou. Mohli by byť pridané ďalšie spôsoby ovládania:

- otáčanie kamery na základe zosnímanej pozície tváre používateľa
- iné spôsoby ovládania klávesnicou a/alebo myšou

### 1.5.5 Import ďalších formátov

Import grafov je možný iba zo súborov vo formáte *GraphML*. Keďže navrhujeme oddelenie importu do nového modulu, môžeme do tohto nového modulu pridať aj import zo súborov v iných formátoch:

- *GXL*
- *GraphXML*
- RSF - formát súborov, ktoré sú výstupom nástrojov analyzujúcich zdrojové kódy (modul by mohol vedieť importovať tieto formáty bez toho, aby museli byť najprv prevedené do niektorého z používaných grafových formátov)

Je potrebné navrhnúť mapovanie dát v súboroch na jednotný výstup importovacieho modulu, aby sa tento výstup dal jednotným spôsobom používať v ďalších moduloch (ukladať do dátových štruktúr alebo databázy) bez ohľadu na to, z akého typu súboru boli dáta importované.

# 2 Analýza technológií a metód pre rozšírenie existujúceho systému

## *2.1 Dátové formáty*

### 2.1.1 GXL

#### Jednoduchý graf

Vo formáte GXL (Graph eXchange Language ) sa dá jednoduchý orientovaný graf vyjadriť vymenovaním uzlov a hrán. Uzly majú identifikátory, na ktoré sa odkazujú hrany.

```
...
<graph id="g1" edgeids="true" edgemode="defaultundirected"
hypergraph="false">
      <node id="n1" />
      <node id="n2" />
      <node id="n3" />
      <edge id="e1" from="n1" to="n2" />
      <edge id="e2" from="n2" to="n3" isdirected="true" />
</graph>
...
```

#### Orientované hrany

To, či je hrana orientovaná, sa vyjadruje atribútom `isdirected` (hodnota `true` alebo `false`). Pre celý graf sa dá nastaviť `edgemode`:

- `defaultdirected` – hrany sú orientované, ak nemajú uvedené inak
- `defaultundirected` – hrany sú neorientované, ak nemajú uvedené inak
- `directed` – všetky hrany musia byť orientované
- `undirected` – všetky hrany musia byť neorientované
- Atribúty

Uzly a hrany môžu mať atribúty rôznych typov (jednoduchých aj zložených) – pomocou nich môžeme uložiť napríklad informácie upresňujúce zobrazenie (napríklad farbu).

```
...
<edge id="e1" from="n1" to="n2" isdirected="true">
      <attr name="edge_color">
            <string>red</string>
      </attr>
</edge>
...
```

### Hypergrafy

Hrany spájajúce viac uzlov sa dajú vyjadriť pomocou relácie:

```
...
<graph id="g1" edgeids="true" edgemode="directed" hypergraph="true">
      <node id="n1" />
      <node id="n2" />
      <node id="n3" />
      <rel id="r1">
            <relend role="n1 role" target="n1" direction="out" />
            <relend role="n2 role" target="n2" direction="in" />
            <relend role="n3 role" target="n3" direction="in" />
      </rel>
</graph>
...
```

Atribút `direction` môžeme nastaviť na `in`, `out` alebo `none`.

### Vnorené grafy

Vnorené grafy vyjadríme pomocou definície grafu umiestnenej v uzle (analogicky môže byť vnorený graf aj v hrane):

```
...<node id="n">
      <graph id="g1" edgeids="true" edgemode="directed" hypergraph="false">
            <node id="n1" />
            <node id="n2" />
            <edge id="e1" from="n1" to="n2" />
      </graph>
</node>
...
```

### Schéma

Všetky uzly, hrany a relácie sú typové – typ určujeme odkazom pomocou `<type>` na typ určený v schéme.

Schéma je GXL graf, kde:

- uzly sú typy (uzlov, hrán a relácií)
- hrany sú vzťahy medzi typmi, napríklad určenie možnosti vychádzania hrany nejakého typu z uzla nejakého typu

## 2.1.2 GraphML

### Jednoduchý graf

Vo formáte GraphML je graf vyjadrený vymenovaním uzlov a hrán. Hrany sa odkazujú na uzly pomocou ich identifikátorov.

```
...
<graph id="g1" edgedefault="undirected>
      <node id="n1" />
      <node id="n2" />
      <node id="n3" />

      <edge source="n1" target="n2" />
      <edge source="n2" target="n3" directed="true" />
</graph>
...
```

### Orientované hrany

To, či je hrana orientovaná, sa vyjadruje atribútom `directed` (hodnota `true` alebo `false`). Pre celý graf sa dá nastaviť `edgedefault`:

- `directed` – hrany sú orientované, ak nemajú uvedené inak
- `undirected` – hrany sú neorientované, ak nemajú uvedené inak

### Atribúty

Atribúty uzlov a hrán sa pridávajú pomocou `data`.

```
...
<edge source="n1" target="n2">
      <data key="color">red</data>
</edge>
...
```

Typy atribútov, na ktoré sa odkazujeme pomocou atribútu `key`, sú definované na začiatku súboru pred definíciou grafu:

```
...
<key id="color" for="edge" attr.name="color" attr.type="string" />
<graph>
...
```

### Hypergrafy

Hrany spájajúce viac uzlov sa pridávajú pomocou `hyperedge`.

```
...
<graph id="g1" edgedefault="undirected">
      <node id="n1" />
      <node id="n2" />
      <node id="n3" />
      <hyperedge>

            <endpoint node="n1" type="out" />
            <endpoint node="n2" type="in" />
            <endpoint node="n3" type="in" />
      </hyperedge>
</graph>

...
```

Atribút `type` vyjadruje orientáciu časti hrany – `in`, `out` alebo `undir`.

**Vnorené grafy**

Vnorený graf sa dá vyjadriť vložením definície grafu do uzla alebo hrany. Uzol alebo hrana môže obsahovať viac vnorených grafov.

```
...
<node id="n">
      <graph id="g1" edgedefault="undirected">
            <node id="n1" />
            <node id="n2" />
            <edge id="e1" source="n1" target="n2" />
      </graph>
</node>
...
```

Hrany definované v grafe môžu spájať aj uzly z rôznych podgrafov.

## *2.2 Parsovanie veľkých súborov*

V nasledujúcich kapitolách uvádzam základné typy parserov a pojednávam o výhodnosti ich použitia pri spracovaní veľkých súborov.

### 2.2.1 SAX

Hlavnou charakteristikou SAX (*simple api for xml*) je to, že počas načítania xml vytvára udalosti, ktoré môžeme v našom programe vyvolať. Tento prístup umožňuje vyvolať len tie časti xml, ktoré nás zaujímajú. Nevýhodou je, že musíme zabezpečiť ukladanie všetkých informácií, ktoré by nás mohli potencionálne zaujímať. SAX je väčšinou používaný v aplikáciách vyžadujúcich vysoký výkon, alebo tam kde veľkosť xml môže prekročiť veľkosť pamäte. Súčasná verzia SAX je 2.0. Pre jazyk C++ je k dispozícii napríklad voľne dostupný parser Xerces, alebo libxml++, ktoré obsahujú rôzne parsovacie knižnice.

### 2.2.2 DOM

Parsery typu DOM (*Document Object Model*) sa odlišujú od SAX tým, že vytvárajú v pamäti prezentáciu celého vstupného dokumentu xml a tým pamäť obsadzujú. Tento prístup môže byť pálčivý najmä vtedy, keď potrebujeme parsovať veľké dokumenty, alebo dokumenty obsahujúce veľký počet vzťahov uzlov. V tom prípade sa veľkosť spotrebovanej pamäte znásobí. Z toho dôvodu sa odporúča použiť tento spôsob iba v prípade, kedy je to nevyhnutné, alebo keď spracovávané súbory nie sú veľké. Pre jazyk C++ je k dispozícii opäť voľne dostupný Xerces, alebo MSXML od spoločnosti Microsoft.

### 2.2.3 Pull Parser

Na rozdiel od parserov SAX, ktoré vyvolávajú udalosti vo volajúcej aplikácii, parsery Pull čakajú na volanie aplikácie. Pri tom si vyžiadajú nasledujúcu možnú udalosť. Aplikácia vytvára volania až pokým neprejde na koniec súboru. Parsery Pull sa používajú v hlavne aplikáciách, ktoré spracovávajú priveľké dáta na to, aby mohli byť uložené v pamäti, alebo ak v danom momente ešte nie sú dostupné všetky časti spracovávaných dát. Na rozdiel od parserov SAX môžu parsert Pull preskakovať udalosti ktorými sa nie je potrebné zaoberať.

Práve v tom je ich výhoda pri veľmi veľkých dátach, ktoré presahujú možnosti dostupnej pamäte.

### 2.2.4 Zvolenie vhodného parsera na veľký xml súbor

Z hľadiska veľkých súborov sa ako najvhodnejší javí parser typu SAX. Parser DOM je príliš pamäťovo náročný. Parser Pull sa hodí skôr na veľké dátové streamy. Pre to do návrhu a implementácie vyberáme praser typu SAX.

### 2.2.5 QT – QXmlStreamReader

**Výhody**

- Spracovávanie *XML* súboru pomocou triedy `QXmlStreamReader`, ktorú poskytuje knižnica *Qt*, má pre nás niekoľko výhod:
- na rozdiel od `QDomDocument` nenačítava celý dokument do pamäte, preto poskytuje vhodný spôsob načítavania veľkých súborov
- používa „pull" princíp (volaním metód si pýtame informácie o ďalších častiach súboru) – na rozdiel od „push" princípu netreba definovať metódy, ktoré parser automaticky volá (máme lepšiu kontrolu nad procesom spracovávania)
- je súčasťou *Qt* knižnice, ktorá už je v projekte používaná

**Zdroj dát**

Dáta môžu byť načítavané z:

- QString
- QByteArray
- QIODevice – môže reprezentovať ľubovoľné zariadenie s klasickým rozhraním na čítanie a zápis dát (môžeme použiť napríklad na načítavanie zo súboru)

**Vzor načítavania**

Na načítanie súboru sa používa nasledujúci spôsob:

- inicializujeme objekt poskytnutím zdroja dát
- v cykle prechádzame súbor (kým nenastala chyba alebo nie sme na konci)
- vypýtame si informáciu o nasledujúcej časti *XML* súboru (môže to byť napríklad začiatok prvku, koniec prvku, komentár alebo textové dáta) a zistíme, o aký typ časti ide
- podľa nájdeného typu použijeme podrobnosti, napríklad pri nájdení začiatku prvku môžeme získať jeho názov alebo zoznam atribútov

```
QXmlStreamReader xml (&dev);
```

```
while ((!xml.hasError ()) && (!xml.atEnd())) {
      QXmlStreamReader::TokenType token = xml.readNext ();

      if (token == QXmlStreamReader::StartElement) {
            if (xml.name () == "node") {
                  QXmlStreamAttributes attrs = xml.attributes();
                  QStringRef id = attrs.value ("id");
                  ...
            }

            if (xml.name () == ...
            ...
      }

      if (token == ...
      ...
}

if (xml.hasError ()) {
      ...
}
```

**Spracovávanie chýb**

Pri práci s chybami sa používajú nasledujúce metódy:

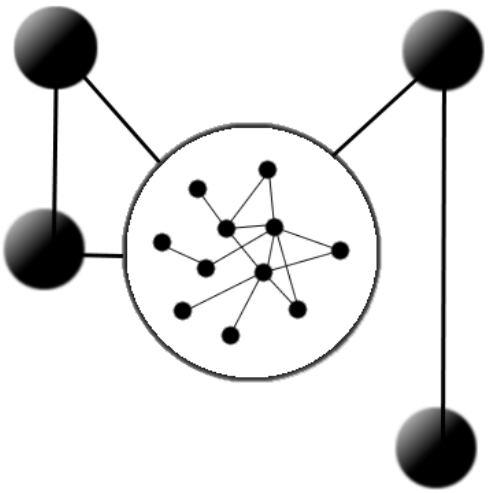
- `hasError()` – vráti `true`, ak nastala chyba
- `error()` – vráti typ chyby
- `errorString()` - vráti text chyby
- `lineNumber()`, `columnNumber()` - získanie aktuálnej pozície

Ak nastane chyba, ktorá súvisí s konkrétnymi spracovávanými dátami a nie so štruktúrou *XML* súboru, môžeme po zistení takejto chyby vyvolať vlastnú chybu metódou `raiseError`. Vtedy sa vytvorí chyba typu `CustomError` a kód používajúci parser sa správa tak, ako keby nastala chyba v parseri.

## *2.3 Možnosti zobrazenia*

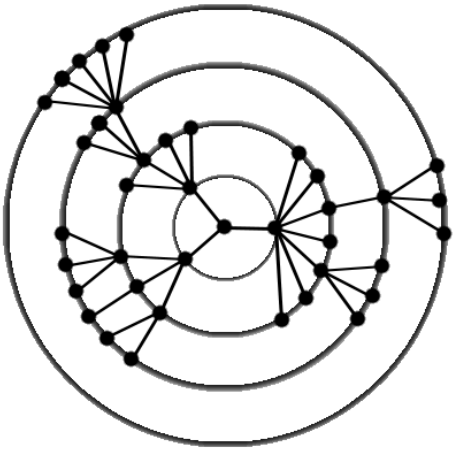
### 2.3.1 Rozmiestnenie uzlov do gule (spherical layout)

Rozmiestnenie uzlov do tvaru gule môžeme využiť najmä, ak chceme v systéme používať vnorené grafy (obr. 2-1).

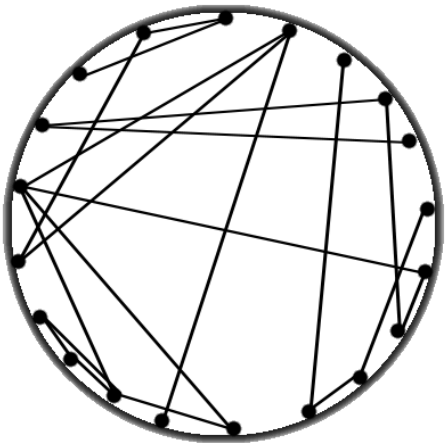


**Obrázok 2-1: Príklad vnoreného grafu.**

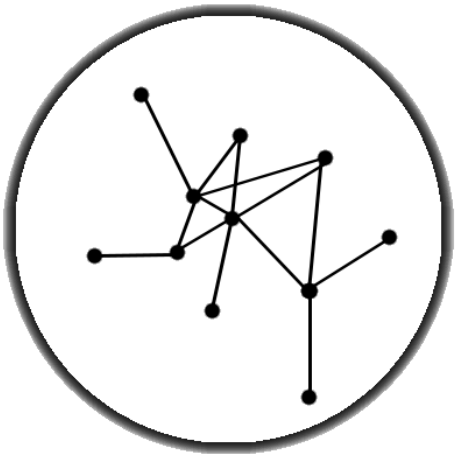
Ideálne rozmiestnenie takéhoto vnoreného grafu je práve do tvaru gule. Niektoré z možností ako takéto rozmiestnenie uzlov dosiahnuť sú zobrazené na obr.2-2, 2-3 a 2-4.



**Obrázok 2-2: Rozmiestnenie uzlov do gule (viac vrstiev).**



**Obrázok 2-3: Rozmiestnenie uzlov do gule (jedna vrstva).**



**Obrázok 2-4: Rozmiestnenie uzlov do gule (bez vrstiev).**

Možnosti ako dosiahnuť rozmiestnenie uzlov do tvaru gule:

1. Určíme si jeden centrálny uzol (najlepšie uzol v strede najdlhšej cesty v grafe) a k nemu budeme postupne dopĺňať jednotlivé vrstvy uzlov, ktoré sú s ním spojené (obr.2-2).

2. Všetky uzly rozmiestnime pravidelne na jednej guľovej vrstve a pospájame ich hranami. Uzly by v tomto prípade mali fixnú pozíciu (pevne danú dĺžku od stredu) (obr. 2-3).

3. Použijeme algoritmus na rozmiestnenie uzlov, ktorý sa už v systéme používa (rozšírený Fruchterman-Reingold algoritmus), no obmedzíme priestor, kde sa môžu uzly rozmiestňovať, na priestor gule so stanoveným polomerom. (obr. 2-4).

### 2.3.2 Možnosti rozklikávania uzlov

Aby sme mohli v grafe používať rozklikávanie uzlov, je potrebné tieto uzly, ktoré sa dajú rozklikávať, odlíšiť od ostatných, či už farbou alebo tvarom uzlov. Po kliknutí na takýto uzol by sa zobrazili ďalšie uzly, ktoré sú s ním spojené. Tu by sme mohli uvažovať o pridaní parametru - hĺbky rozklikávania uzlov. Po rozbalení uzla by sa zobrazila nielen jedna vrstva uzlov, ktorá je spojená s aktuálnym, ale viac vrstiev, podľa veľkosti parametra.

Na výber máme dve možnosti načítavania grafu:

1. Načítanie celého grafu a vykreslenie len jeho vybranej časti, pričom ďalšie časti by sa zobrazovali v závislosti od používateľa.

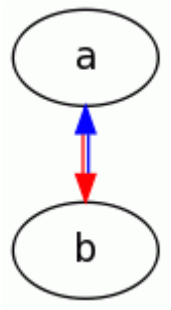
2. Načítanie vybranej časti grafu a jej zobrazenie, pričom ďalšie časti by bolo potrebné opätovne načítať zo súboru alebo databázy.

Riešenie 1 by vyžadovalo viac pamäte a o každom uzle by bolo potrebné uchovávať informácie, či je uzol zobrazený alebo nie. Riešenie 2 nevyžaduje toľko pamäte v prípade, že nechceme zobraziť celý graf, t.j. nerozklikáme všetky uzly. Na druhej strane by bolo pomalšie

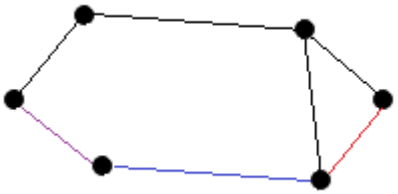
pri rozklikávaní uzlov, keďže nové uzly by neboli uložené v pamäti a bolo by potrebné ich načítať.

### 2.3.3 Zobrazenie multihrán (paralelných hrán)

Multihrany, respektíve paralelné hrany, sú také, ktoré spájajú rovnaké dva uzly. Ak ich chceme zobraziť, hlavným problémom je, aby bolo jasne viditeľné, že dva vrcholy spája viac ako jedna hrana. Na obrázku 2-5 je znázornený prípad, kde dva uzly *a* a *b* spájajú dve orientované hrany. Sú farebne odlíšené.




**Obrázok 2-5: Orientované paralelné hrany.**



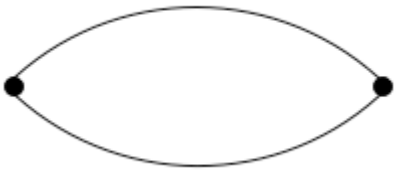
**Obrázok 2-6: Farebne odlíšené hrany.**

Môže však nastať prípad, že uzly spájajú viac ako dve hrany. Otázkou je aké grafické zobrazenie zabezpečí prehľadný graf. Je niekoľko alternatív.

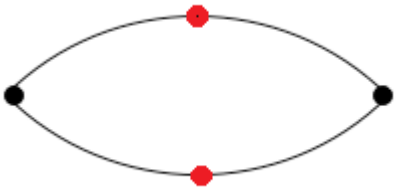
Prvá možnosť je farebne odlíšiť hrany medzi dvoma uzlami. Každá farba by znamenala iný počet hrán medzi uzlami, respektíve by zafarbená hrana znamenala, že vrcholy spája viac ako jedna hrana. Následne by bolo možné selektovať tieto uzly a zobrazil by sa výpis všetkých hrán. Toto riešenie je zobrazené na obrázku 2-6.

Ďalšou možnosťou je grafické vykreslenie hrany. Napríklad hrana zakončená krúžkom reprezentuje paralelné multihrany.

Taktiež grafickým riešením je zobrazenie multihrán tak ako je to znázornené na obrázku 2-7. Toto riešenie by bolo vhodné pre malé grafy a nie veľký počet hrán medzi dvoma rovnakými uzlami. Vyžadovalo by si však výpočet zakrivenia pre všetky hrany v závislosti od nejakého ich ohodnotenia.



**Obrázok 2-7: Vykreslenie dvoch multihrán.**



**Obrázok 2-8: Prídavné uzly multihrán.**

Iným prístupom je rozdelenie paralelných hrán. Je to myslené tak, že sa každá paralelná hrana rozdelí na dve hrany medzi ktorými je špeciálny uzol. Takéto uzly môžu byť graficky odlíšené od všeobecných uzlov. Toto riešenie je zobrazené na obrázku 2-8. Nové pridané uzly sú znázornené červenou farbou.

Z hľadiska existujúcej aplikácie sa ako najvhodnejšie riešenie javí posledná možnosť, teda pridať do grafu špeciálne vrcholy symbolizujúce multihrany. Je to pre to, že netreba vytvárať nový typ hrany a ani upravovať layoutovací algoritmus. Jednoducho sa pridajú ďalšie vrcholy, ktoré môžu byť rôzne označené a každá multihrana sa rozdelí na dve.

## *2.4 Knižnica Bullet*

Bullet Physics je profesionálna open source knižnica na detekciu kolízií a simulovanie dynamiky tuhých a mäkkých telies.

Hlavné vlastnosti:

- Open source C++ kód pod Zlib licenciou a dostupný na komerčné použitie na všetkých platformách vrátane Playstation 3, Xbox 360, Vii, Microsoft, Linux, Mac OSX a iPhone
- Diskrétna a spojitá detekcia kolízií. Tvary kolízií zahŕňajú konkávne a konvexné siete a všetky základné tvary
- Rýchla a stabilná dynamika pevných telies, dynamika vozidiel a zavesenia
- Dynamika mäkkých telies pre tkaniny, lano a deformovateľné telesá s dvojsmernou interakciou s pevnými telesami, vrátane podpory tlaku

Začlenenie Bullet physics do projektu

Každý frame volá *stepSimulation* na dynamiku sveta a synchronizuje transformáciu sveta pre grafické objekty. Požiadavky:

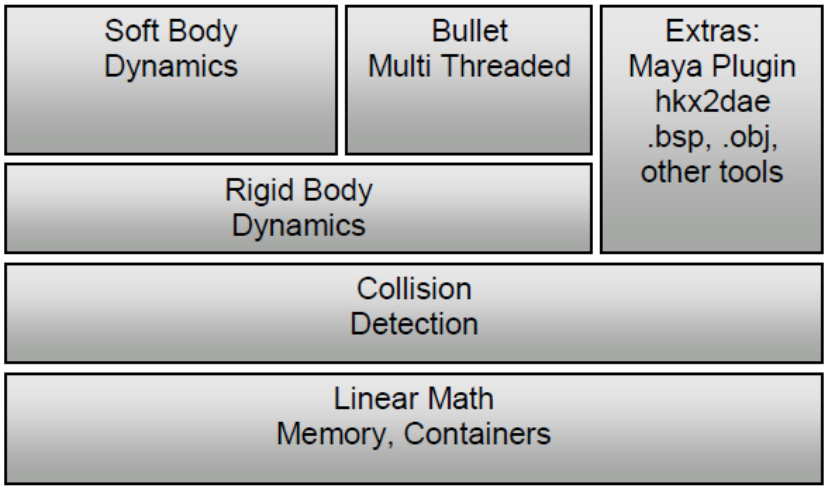
- #include "btBulletDynamicsCommon.h" v zdrojovom súbore
- potreba vložiť cestu k priečinku: Bullet/src
- potreba knižníc: BulletDynamics, BulletCollision, LinearMath

Aktuálny názov knižníc môže byť odlišný. Napríklad kompiler Visual Studia pridáva príponu .lib, na UNIX systémoch zvyčajne sa pridáva prefix lib a prípona .a.

### 2.4.1 Software Design

Bullet bol navrhnutý ako prispôsobiteľný a modulárny. Vývojár môže:

- používať len komponenty detekcie kolízie
- používať komponenty dynamiky pevných telies bez komponentov dynamiky mäkkých telies
- používať len malé časti knižnice a rozšíriť knižnicu v mnohých smeroch
- vybrať, či bude používať verziu knižnice pre jednoduchú presnosť alebo dvojitú presnosť
- použiť vlastný alokátor pamäti, pripojiť vlastný ladič vykresľovania
- Hlavné komponenty sú organizované nasledovne [3]:

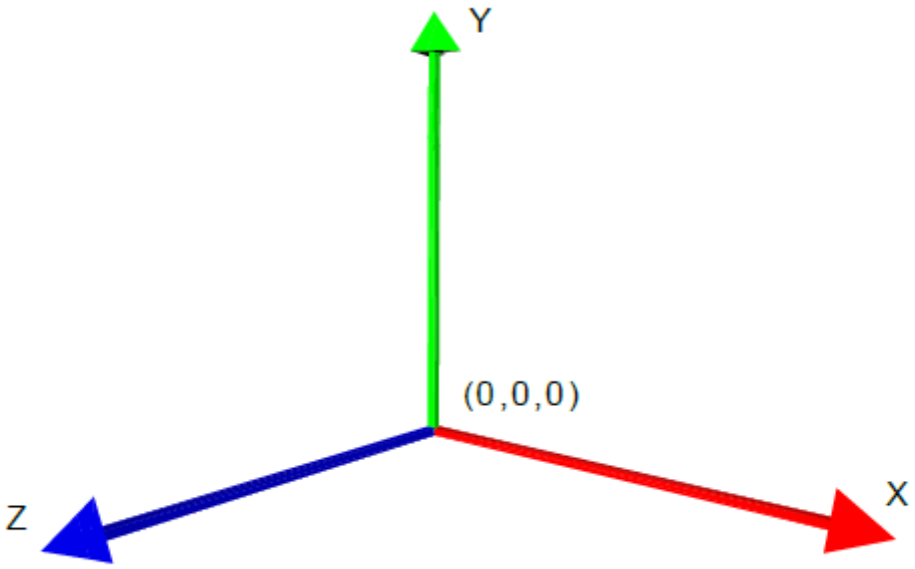



**Obrázok 2-9: Komponenty Bullet.**

### 2.4.2 Základné dátové typy a Matematická knižnica

Základné dátové typy, manažment pamäte a containery sú uložené v *Bullet/src/LinearMath*.

- *btVector3* – 3D pozície a vektory môžu byť reprezentované použitím btVector3. btVectror3 má 3 skalárne x, y, z komponenty. Má však aj štvrtý, nepoužívaný w komponent z dôvodu zarovnania a SIMD kompatibility.
- *btQuaternion a btMatrix3x3* – 3D orientácie a rotácie môžu byť reprezentované použitím btQuaternion alebo btMatrix3x3
- *btTransform* – je kombináciou pozície a orientácie. Môže byť použitý na transformovanie bodov a vektorov z jednorozmerného priestoru do iného.
- Bullet využíva pravotočivý súradnicový systém:




**Obrázok 2-10: Pravotočivý súradnicový systém.**

### 2.4.3 OSGBullet

OsgBullet je množstvo softvérových nástrojov pre aplikácie, ktoré používajú súčasne OpenSceneGraf a Bullet.

OsgBullet knižnice zahrňujú nasledovnú funkcionalitu:

- podporuje prekladané/sériové fyzikálne kroky a render, samostatné vlákno na fyzikálne simulácie. Vlastnosti vláknovej fyzikálnej simulácie ako účinný trojitý bufferovací mechanizmus pre zdieľané dáta.
- Pomocné funkcie pre prevod medzi pevnými telesami v Bullet a OSG grafom scény(napríklad vytvorenie grafu scény na vykreslenie pevného telesa, alebo vytvorenie pevného telesa z grafu scény)
- pomocné funkcie pre prevod medzi Bullet a OSG maticami a vektorovými dátovými typmi

- COLLADA – základné funkcie pre čítanie a zapisovanie pevných telies a informácii a kolíziách
- trieda MotionState, ktorá umožňuje Bullet-u špecifikovať OSG transformačnú maticu

## *2.5 Možnosti snímania tváre pre lepšiu 3D vizualizáciu*

V projekte môžeme využiť snímanie tváre na automatické otáčanie grafov. Zosnímame obraz pomocou web kamery, rozoznáme na ňom tvár človeka. Keď človek pozerajúci sa na obrazovku pohne hlavou doprava, ako keby sa na graf chcel pozrieť zľava, graf sa mu automaticky natočí, aby videl jeho ľavú stranu, resp. keď pohne hlavou doľava, graf sa natočí, aby bolo vidieť jeho pravú stranu. Na rozoznávanie tváre na obrazovke môžeme použiť knižnicu OpenCV.

### 2.5.1 OpenCV

OpenCV je voľná open source knižnica počítačového videnia pre C/C++. OpenCV je knižnica, ktorá sa používa na detekciu tváre, sledovanie tváre a rozlišovanie tváre. OpenCV je multiplatformová, podporuje aj Windows, aj Linux a MacOSX. Okrem jednej výnimky (CVCam) sú jej rozhrania platformovo nezávislé.

**Všeobecné počítačové videnie a algoritmus spracovanie obrazu**

Použitím týchto rozhraní môžeme experimentovať s množstvom štandardných algoritmov počítačového videnia bez nutnosti ich programovať. Obsahujú detekciu hrán, priamok, rohov, pyramídové obrázky pre mnohoúrovňové spracovanie, porovnávanie vzorcov, rôzne transformácie(Fourier, diskrétna kosínusová a transformácie vzdialenosti).

**Vysokoúrovňové moduly počítačového videnia**

OpenCV obsahuje množstvo vysokoúrovňových schopností, ako napríklad detekciu tváre, rozoznávanie a sledovanie, to zahŕňa optický tok (použitím pohybu kamery na určenie 3D štruktúr).

**Umelá inteligencia a metódy strojového učenia**

Aplikácie počítačového videnia často potrebujú metódy strojového učenia, alebo iné metódy umelej inteligencie. Niektoré sú dostupné v OpenCV balíku strojového učenia.

**Vzorkovanie obrázkov a transformácie zobrazenia**

Často je výhodné spracovávať skupinu pixelov ako jeden prvok. OpenCV obsahuje rozhrania pre extrakciu subregiónov z obrázku, náhodné vzorkovanie, menenie veľkosti, deformáciu, rotáciu a aplikovanie perspektívnych efektov.

**Metódy pre vytváranie a analyzovanie binárnych obrázkov**

Binárne obrázky sú často používané v kontrolných systémoch, ktoré zisťujú chyby tvaru, alebo počet častí.

**Metódy pre počítanie 3D informácie**

Tieto funkcie sú užitočné pre mapovanie a lokalizáciu, s viacerými pohľadmi kamery.

*Matematické algoritmy pre spracovanie obrázkov, počítačové videnie a interpretáciu obrázkov*

OpenCV obsahuje zvyčajne používané matematické algoritmy z lineárnej algebry, štatistiky a počítačovej geometrie.

**Grafika**

Toto rozhranie umožňuje písanie textu a vykresľovanie obrázkov. Je vhodné aj na označovanie útvarov na obrázku.

**Dátové štruktúry a algoritmy**

S týmito rozhraniami môžeme efektívne ukladať, vyhľadávať a manipulovať s veľkými zoznamami, skupinami, grafmi a stromami.

**Životnosť dát**

Táto metóda poskytuje pohodlné rozhranie pre ukladanie rôznych typov dát na disk a ich neskoršie získavanie.

OpenCV funkcionalita je rozložená do niekoľkých modulov.

*CXCore* obsahuje základné definície dátových typov. Napríklad dátové štruktúry pre obrázky, body, štvorce sú definované v cxtypes.h. CXCore tiež obsahuje lineárnu algebru a štatistické metódy a ošetrenie chýb.

*CV* obsahuje spracovanie obrázkov a metódy na kalibrovanie kamery. Funkcie výpočtovej geometrie sú tiež tu definované.

*CVAUX* obsahuje okrem iného aj najjednoduchšie rozhrania pre rozoznávanie tváre.

*ML* obsahuje rozhrania strojového učenia.

*HighGUI* obsahuje vstupno-výstupné rozhrania a multiplatformové schopnosti okien.

*CVCAM* obsahuje rozhrania pre prístup videa cez DirectX na 32bitových Windows platformách.

Pomocou tejto knižnice môžeme snímať obraz z web kamery, rozoznať na videu tvár používateľa a podľa pohybov jeho tváre natáčať graf.

## 2.6 Zhodnotenie analýzy

Z uvedenej analýzy vyplýva nasledovné. Aplikácia v súčasnej podobe (prebraný výsledok od posledného tímu) poskytuje funkcionalitu pre vykreslenie základných grafov pomocou uzlov a hrán. Umožňuje načítanie iba formátu typu GraphML a následné vykreslenie 3D grafu, jeho prehliadanie v priestore, zobrazenie názvov uzlov a hrán a vytvorenie pomocných (meta) uzlov. Aplikácia neposkytuje podporu hypergrafov, multigrafov ani vnorených grafov. Tieto typy grafov sa však dajú mapovať na existujúcu dátovú štruktúru založenú na reprezentácii základných elementov grafu, ktorými sú uzly a hrany. Je však potrebné doplniť funkcionalitu na ich načítanie, na ich mapovanie do existujúcej dátovej a databázovej štruktúry a upraviť spôsob ich vykreslenia. Taktiež chýba funkcionalita týkajúca sa prepojenia s databázou, ukladania a načítania grafov a ich častí z databázy. Ďalej sme pre implementáciu parsovania vstupných súborov zvolili parser typu SAX.

Z hľadiska vizualizácie aplikácia podporuje vykreslenie grafu a doplnenie pomocných uzlov. Je vhodné doplniť funkcionalitu zameranú na nastavenie vizuálnych vlastností uzlov a hrán na základe ich vlastností a atribútov v rámci grafu (možnosť vyfarbenia multi-hrán a podobne).

# 3 Špecifikácia požiadaviek

Požiadavky vyplývajú z chýbajúcej nedokončenej funkcionality systému a z návrhov na ďalšiu možnú funkcionalitu.

## *3.1 Funkcionálne požiadavky*

- podpora ďalších formátov súborov s uloženými informáciami o grafoch (GXL, GraphML, GraphXML)
- dokončenie ukladania grafov do databázy; doplnenie používateľského rozhrania o možnosť zobrazenia uložených grafov a výberu grafu na načítanie z databázy
- pridať podporu vnorených grafov, multihrán a hypergrafov
- vylepšiť prácu s metauzlami – lepšia možnosť voľby pozície metauzla a voľba vzdialenosti od kamery
- pridať iné spôsoby práce s kamerou
- možnosť dopytovania – vyhľadávanie a filtrovanie v dátovom úložisku
- možnosť nastaviť mapovanie vlastností prvkov grafu na ich vizuálne vlastnosti

## *3.2 Nefunkcionálne požiadavky*

### 3.2.1 Závislosti na knižniciach

- prispôsobenie projektu najnovšiemu stavu knižníc, najmä *OSG* a *QT*
- zrušenie závislosti na už nepoužívaných knižniciach; vymazať nepoužívané súbory knižníc z projektu
- zníženie závislostí na knižniciach *OSG* a *QT* v tých moduloch, ktoré nesúvisia s používaním týchto knižníc (teda nesúvisia s vykresľovaním grafu ani s používateľským rozhraním)

### 3.2.2 Architektúra systému

- oddelenie importu zo súboru do samostatného modulu
- oddelenie záležitostí súvisiacich so zobrazením od štruktúry grafu (v module *Data*)

### 3.2.3 Podpora platforiem

- snažiť sa o podporu platforiem Windows a Linux, napr. vhodným výberom multiplatformových knižníc a písaním kódu skompilovateľného na týchto platformách)

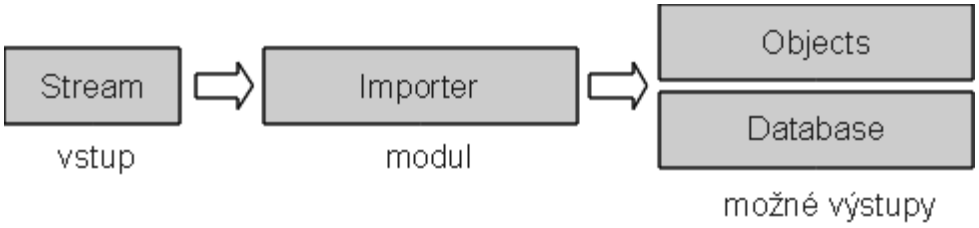
# 4 Návrh systému

## *4.1 Import dát - Architektúra modulu Importer*

### 4.1.1 Vstup a výstup modulu

Import je implementovaný v samostatnom module `Importer`. Úlohou modulu je transformovať dáta získané zo streamu na nejakú reprezentáciu vhodnú na používanie v aplikácii (Obr.4-1). Možné sú napríklad tieto reprezentácie:

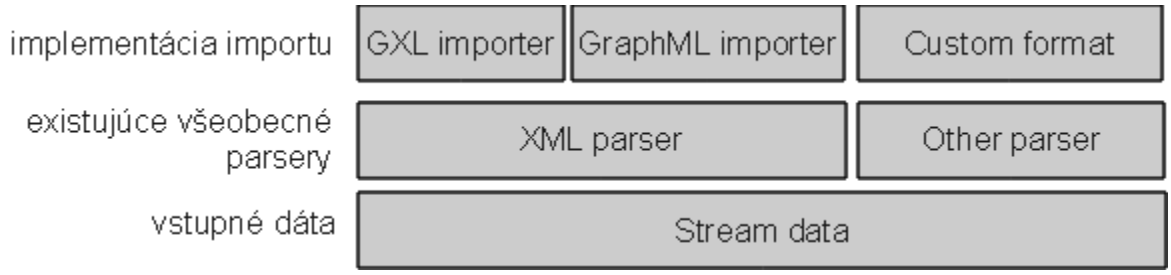
- objekty (súčasné riešenie)
- databáza (ďalšie možné riešenie)




**Obrázok 4-1: Vstup a výstup modulu na import dát.**

### 4.1.2 Vrstvy spracovávania dát

Modul spracováva streamy, ktorých dáta môžu mať rôzny formát, preto môže existovať viac implementácii kódu na spracovávanie vstupu. Spracovávanými dátami môžu byť napríklad súbory vo formáte *XML* (*GraphML* a *GXL* sú napríklad formáty založené na *XML*). Implementácie spracovávajúce formáty založené na *XML* môžu byť postavené nad existujúcim parserom (Obr. 4-2).

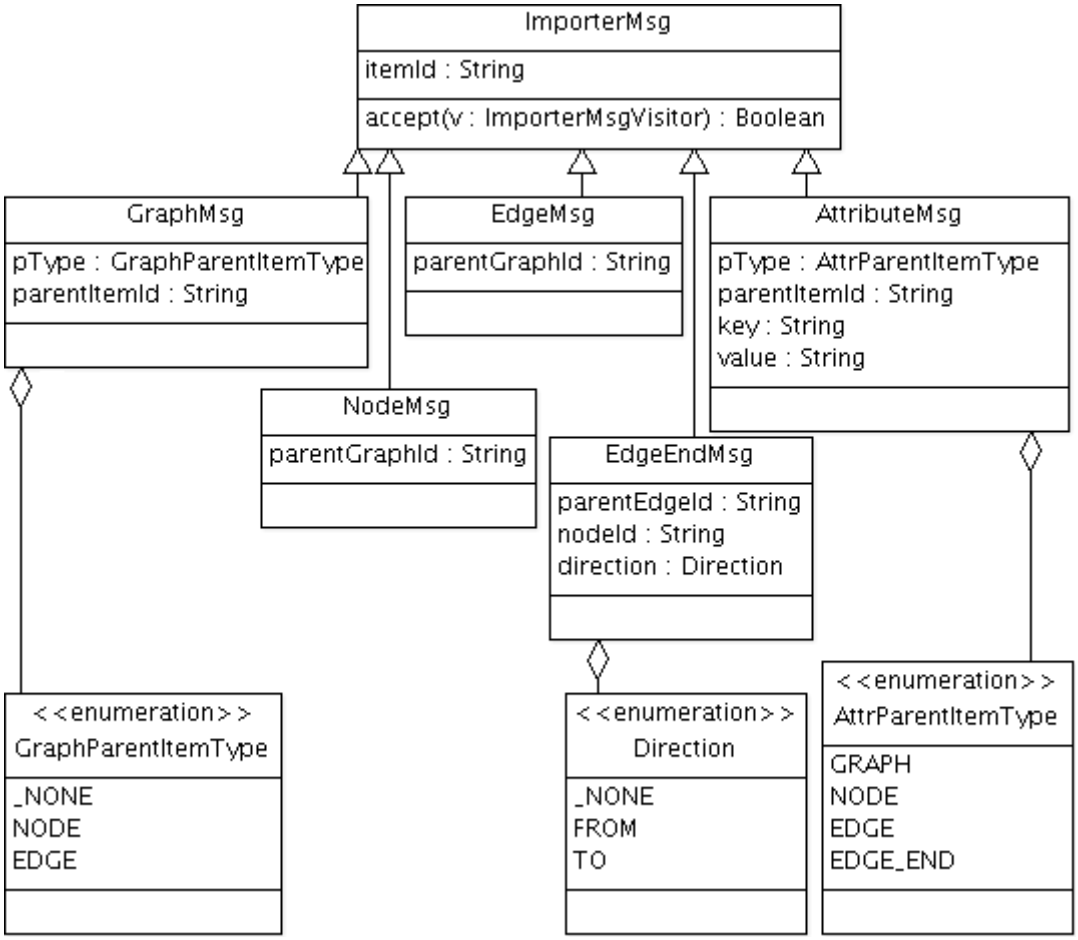



**Obrázok 4-2: Úrovne spracovávania dát.**

### 4.1.3 Správy o načítaných dátach

Modul číta dáta a priebežne informuje o nájdených položkách pomocou tzv. správ (Obr. 4-3). Keď napríklad prečíta informáciu o uzle grafu, vytvorí objekt reprezentujúci správu o uzle grafu s informáciami o tomto uzle. Modulu sa pred samotným čítaním dát poskytne implementácia, ktorá má byť zavolaná pre každý typ správy. Modul teda nemusí vedieť, ako sa s dátami grafu ďalej pracuje a tiež nemusí dáta nikam ukladať – sú priebežne posielané

ďalej. Niektoré implementácie objektov, ktoré budú spracovávať správy, si môžu dočasne ukladať informácie, ak by si potrebovali pamätať nejaký kontext.




**Obrázok 4-3: Správy o nájdených položkách grafu.**

Každá položka grafu alebo atribút položky grafu má jednoznačný identifikátor (`itemId`), na ktorú sa môžu iné prečítané položky odkazovať. Môžu byť použité nasledujúce väzby:

`GraphMsg`, ak ide o vnorený graf, sa môže pomocou `parentItemId` odkázať na uzol alebo hranu, ktorej patrí (vtedy je nastavené `parentType` na `NODE` alebo `EDGE`)

- `NodeMsg` a `EdgeMsg` sa pomocou `parentGraphId` odkazujú na graf, ktorému patria

`EdgeEndMsg` sa pomocou `parentEdgeId` odkazuje na hranu, ktorej patrí, a pomocou `nodeId` na uzol, na ktorý sa daný „koniec" hrany pripája

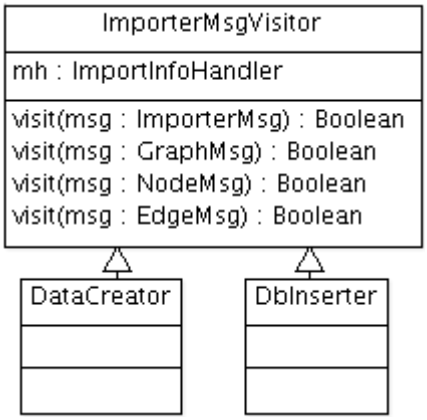
`AttributeMsg` sa pomocou `parentItemId` odkazuje na položku grafu, ktorej vlastnosť patrí, pričom `pType` je nastavené na typ tejto položky

Atribúty (`AttributeMsg`) špecifikujú bližšie informácie o položkách, napríklad o grafickej reprezentácii.

Aby sme mohli načítavať aj hypergrafy (hrana môže spájať viac uzlov), informácia o hrane je rozdelená do správy o samotnej hrane (`EdgeMsg`) a správy o jej „koncoch" (`EdgeEndMsg`), z ktorých každá má informáciu o uzle, ku ktorému sa daný koniec pripája (`nodeId`) a tiež smer (či smeruje od toho uzla, k tomu uzlu alebo či to nie je orientovaná časť hrany).

### 4.1.4 Spracovávanie správ

Spracovávanie správ o prečítaných položkách grafu je riešené pomocou návrhového vzoru *Visitor*. Modulu pred samotným importom poskytneme objekty triedy, ktorá má implementovanú metódu pre každý typ možnej prijatej správy. Takýchto tried môžeme mať viac, napríklad na import do objektového modelu alebo do databázového modelu (Obr. 4-4).




**Obrázok 4-4: Triedy spracovávajúce správy.**

### 4.1.5 Kontext a spolupráca s GUI

Objekty, s ktorými kód importujúci dáta môže pracovať, sú združené do objektu triedy `ImporterContext` (Obr. 4-5). Ide o objekty nasledujúcich typov:

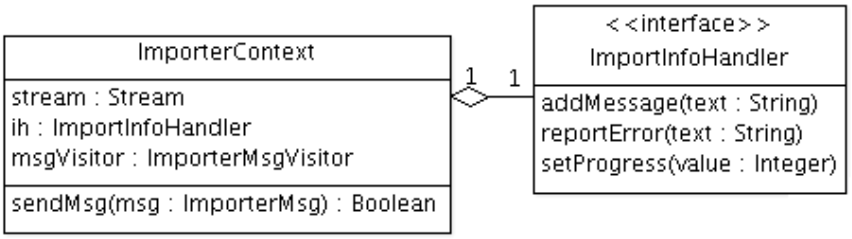
- `Stream` – stream, z ktorého sa importuje
- `ImportInfoHandler` – implementácia reakcií na:

pribudnutie správy o importe pre používateľa, napríklad názvu práce importovanej časti definície grafu (jedna z implementácií môže napríklad vypisovať takéto správy do okna v používateľskom rozhraní)

  - chybu počas importu (konkrétna implementácia môže napríklad zobraziť okno s textom chyby)

postup priebehu importu na nejaké percento dokončenia operácie (implementácia môže napríklad obnovovať informáciu o dokončenej časti procesu v percentách zobrazenú v používateľskom rozhraní)

- `ImporterMsgVisitor` – trieda s implementáciami metód na spracovávanie správ



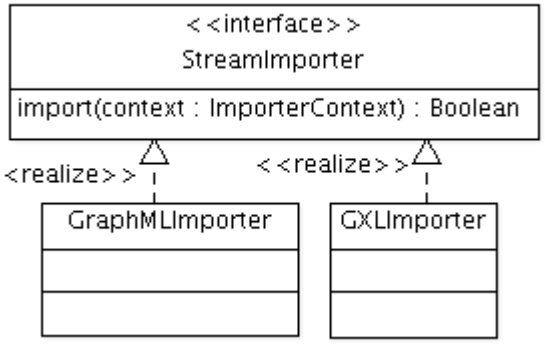

**Obrázok 4-5: Kontext importovania dát a trieda na spoluprácu s GUI.**

Objekt triedy `ImportInfoHandler` je poskytnutý aj objektu triedy `ImporterMsgVisitor`, lebo aj tam môže nastať napríklad chyba, ktorú treba oznámiť.

Metóda `sendMsg` je pripravená na volanie z kódu, ktorý vykonáva samotné spracovávanie vstupných dát, aby mohol poslať správu zloženú z práve prečítaných dát.

### 4.1.6 StreamImporter

Samotný import, teda spracovávanie dát zo vstupného streamu, vykonáva `StreamImporter` v metóde `import` (*Obr.* 4-6). Môže existovať viac implementácií pre rôzne formáty, tieto môžu využívať napríklad už existujúce parsery a nemusia tak pracovať priamo nad dátami zo streamu. Metóda dostane ako parameter objekt typu `ImporterContext`, kde má všetko, s čím môže pracovať.




**Obrázok 4-6: Trieda StreamImporter a odvodené triedy.**

### 4.1.7 Spracovávanie chýb

Každá funkcia, v ktorej prebieha časť importu, vracia hodnotu typu `bool`, ktorá označuje, či sa vykonávanie skončilo úspešne. Ide o funkcie:

- `StreamImporter::import (hlavný kód spracovávania dát)`
- `ImporterMsgVisitor::visit (spracovanie jednej správy)`

### 4.1.8 Vybratie vhodného importera

Keďže existuje niekoľko implementácií importu, je nutné vyberať vhodnú triedu, ktorá sa môže použiť na načítanie nejakých konkrétnych dát. Trieda `ImporterFactory` preto rieši vytvorenie inštancie vhodnej triedy, ktorá môže byť zvolená napríklad na základe prípony súboru.

## *4.2 Zjednodušenie modulu Importer použité v prototype*

Pri implementácii prototypu je použité nasledujúce zjednodušenie modulu:

- *Importer* objekty neposielajú správy o nájdených položkách grafu, ale priamo upravujú dátovú štruktúru grafu vkladaním nových prvkov do nej

- graf sa ukladá iba do štruktúr v pamäti; na ukladanie do databázy je využitá pripravená funkcionalita (automatické zmeny v databáze po zmene štruktúr v pamäti)

V prototype sú implementované triedy na import formátov uvedených v tabuľke 4-1.

**Tabuľka 4-1: Importované formáty.**

| Trieda | Spracovávaný formát | Použitá knižnica |
|---|---|---|
| `GraphMLImporter` | *GraphML* (*XML*) | *QT* (`QdomDocument`) |
| `GXLImporter` | *GXL* (*XML*) | *QT* (`QXmlStreamReader`) |

Trieda na import formátu *GraphML* už v projekte existoval a bol iba presunutý do tohto modulu. Boli z neho oddelené časti, ktoré môžu byť spoločné pre viac implementácií importu:

- `ReadNodesStore` – odkladanie referencií na načítané uzly, aby mohli byť použité, keď sa na ne budú odkazovať neskôr načítané hrany
- `GraphOperations` – operácie vykonávané počas importu nad dátovými štruktúrami grafu, napríklad pridanie základných typov uzlov a hrán do grafu

## *4.3 Reprezentácia multihrán v dátovom modeli v prototype*

Funkcionalita pre zobrazenie a prezentáciu multihrán vyžaduje úpravu jeho triedy *Graph* modulu *Data*.

Metóda *addEdge* triedy *Graph* plní funkciu pridávania hrán do grafu. Metóda overí, či sa jedná o multihranu tak, že skontroluje všetky hrany vychádzajúce zo zdrojového uzla vstupnej hrany *srcNode* a overí, či niektorá z hrán nemá rovnaký cieľový uzol. Ak zistí že sa jedná o multihranu, vytvorí pomocný uzol typu *MULTI_NODE_TYPE*, ktorý spojí so zdrojovím a cieľovým uzlom vstupnej hrany pomocnými hranami typu *MULTI_EDGE_TYPE*:

- `addEdge`

  ***Vstup:***

  - názov, typ, <*node*> zdrojový uzol, <*node*> cieľový uzol, orientácia

  ***Výstup:***

  - <*edge*> pridaná hrana

Trieda *Graph* je doplnená o nasledovné metódy:

- `addMultiEdge`

  ***Vstup:***

  - názov, typ, <*node*> zdrojový uzol, <*node*> cieľový uzol, orientácia

***Výstup:***

- dve hrany typu *multi* spojené v pomocnom <*node*> uzle typu *multi* a prvá je napojená na cieľový uzol, druhá na zdrojový uzol

- `getMultiEdgeNeighbour`

  ***Vstup:***

  - hrana typu *MULTI_EDGE_TYPE*

***Výstup:***

- susedný uzol, ktorý nie je typu *MULTI_NODE_TYPE*

- `isParralel`

  ***Vstup:***

  - hrana, ktorej vlastnosť chceme overiť

***Výstup:***

- boolovská hodnota určujúca, či je daná hrana *multihranou*

Výsledkom operácie metódy *addEdge* triedy *Graph* modulu *Data* je vloženie pomocného <*node*> uzla a pomocných <*edge*> hrán do dátovej štruktúry. Tieto objekty sa vykreslia rovnako ako hrany a uzly iných typov.

## *4.4 Reprezentácia hyperhrán v dátovom modeli*

Funkcionalita pre zobrazenie a prezentáciu hyperhrán vyžaduje úpravu jeho triedy *Graph* modulu *Data*. Hyperhrana je tvorená pomocným uzlom a na neho naviazanými pomocnými hranami spájajúcimi koncové body hyperhrany. Do triedy *Data* je pridaná metóda *addHyperEdge* a metóda *getHyperNode*:

- `addHyperEdge`

  Metóda vytvorí časť hyperhrany. Ak je to prvá časť, vloží pomocný uzol typu *HYPER_NODE_TYPE* a tento uzol a cieľový uzol prepojí pomocná hrana typu *HYPER_EDGE_TYPE*. Ak už je vytvorená nejaká časť hyperhrany, na vstupe je referencia na jej vytvorený pomocný uzol. K nemu sa pripojí nová časť hyperhrany – pomocná hrana typu *HYPER_EDGE_TYPE*.

***Vstup:***

- názov, typ, <*node*> cieľový uzol, <*edge*> hyper-hrana (jedna hrana hyperhrany, ak je to nová hyperhrana potom *Null*), orientácia

  ***Výstup:***

  - <*node*> pridaný pomocný uzol hyperhrany

- `getHyperNode`

  Metóda vráti pomocný uzol typu *HYPER_NODE_TYPE* vstupnej <*edge*> hyperhrany.

***Vstup:***

- <*edge*> časť hyperhrany

***Výstup:***

- <*node*> uzol tvoriaci hyperhranu

## *4.5 Vyhľadávanie a filtrovanie v databáze*

V databáze sa dajú vyhľadávať celé grafy, no niekedy nepotrebujeme zobraziť celý graf, ale iba jeho určitú časť, alebo filtrovať iba body s určitými vlastnosťami. Na tento účel by bolo dobré aby sa z databázy dali vyberať aj časti grafu, nie len celý graf.

Vyhľadávanie a filtrovanie v databáze môžeme použiť nasledovne:

- môžeme zvoliť určitý bod a nechať vybrať len body, ktoré sú s ním priamo spojené. Na výber len takýchto bodov by sme mohli použiť príkaz „SELECT n2 FROM edges WHERE graph_id=:graph_id AND n1=:n1“, kde n1 je určitý uzol, ktorý sme vybrali cez rozhranie.
- filtrovať len metauzlov. Na filtrovanie len metauzlov použijeme príkaz „SELECT node_id FROM nodes WHERE graph_id=:graph_id AND meta=false“
- filtrovať len regulárne uzly (všetky okrem meta)
- vyhľadať len uzly s určitými vlastnosťami
- Na vyhľadávanie a filtrovanie uzlov v databáze používame SQL príkazy, pomocou ktorých vyberáme len určité časti grafu, ktoré následne môžeme zobraziť.

## *4.6 Doplnenie podpory práce s grafmi v GUI*

Pre prácu s grafmi je zatiaľ možné vkladať meta uzly. Z tohto dôvodu budú implementované ďalšie funkcie ako vkladanie uzlov, hrán, vymazávanie.

### 4.6.1 Funkcie v GUI aplikácii

Pre funkcie vkladanie uzlov pridáme jedno tlačidlo *Add Node* a pre dané tlačidlo vytvoríme funkciu *add_NodeClick*. Najprv získame existujúci graf, ktorý vložíme do *currentGraph*. Ak neexistuje pri pridávaní uzla ho vytvoríme. Pri volaní *getSelectionCenter* získame pozíciu, kde je kamera centrovaná (otočená). Uzol pridáme volaním funkcie *addNode*. Po vložení uzla musíme overiť či je *layoutovač* zapnutý alebo nie.

Ďalšou funkciou je vkladanie hrany a vytvoríme tlačidlo *Add Edge*. Pomocou tejto funkcie medzi dva uzly vložíme neorientovanú hranu. Po stlačení tlačidla sa vola funkcia *add_EdgeClick*. Volaním funkcie *getActiveGraph* získame *currentGraph*. Z *viewerWidget* voláme funkciu *getSelectedNodes,* ktorá nám do *QLinkedLis*tu *selectedNodes*. Overíme či tento zoznam obsahuje 2 uzly. Ak áno vložíme hranu medzi uzly volaním funkcie *addEdge*.

Poslednou vytvorenou funkcionalitou je mazanie hrán a uzlov. Vytvoríme tlačidlo *remove*, ktoré volá funkciu *removeClick.* Pre *currentGraph* voláme funkciu *removeEdge* pre každú položku zoznamu *selectedEdges.* Tú získame volaním funkcie *getSelectedEdges*. To isté spravíme aj pre uzly.

## *4.7 Doplnenie podpory importovania grafov s využitím RSF parsera*

Implementované parsery podporujú formáty (*GraphML* a *GXL*). Pre zvýšenie použiteľnosti naprogramujeme metódy pre import *RSF* formátu.

### 4.7.1 RSF Importer

Pre import *RSF* formátu implementujeme nový parser, pretože má odlišnú štruktúru ukladania údajov ako formáty *GraphML* a *GXL*. V *RSF* súboroch sú uložené v riadku 3 slová. Prvé predstavuje hranu (spojenie), druhé zdrojový a tretie cieľový vrchol.

Na importovanie budeme využívať `QTextStream`. Vo formáte rsf môžu byť slová v riadku oddelené rôznym počtom medzier a tabulátorov. Na potlačenie viacerých medzier a tabulátorov využívame triedu `QRegExp` a regulárny výraz [ \t] . `QTextStream` budeme načítavať po riadkoch do `QStringList` a následne uložíme jednotlivé slová do `QString`u pre hranu a zdrojový a cieľový vrchol. Nový vrchol vložíme do grafu a do triedy `ReadNodesStore`. Tá slúži na overenie existencie vrcholu. Ako vrchol pridáme aj hranu špeciálneho typu s využitím funkcie *addHyperEdge*. Na spojenie medzi vrcholmi a hranou vložíme hranu `defaulttype`. Pri pridávaní novej hrany a vrcholov využijeme na existenciu uzlov `ReadNodesStore`. Ak takýto vrchol existuje, zavoláme metódu *readNodes.get(NodeName)*. Na overenie spojenia využijeme metódu *getHyperEdge*. Tá nám vráti vrchol reprezentujúci hranu, ak existuje. Ak takáto hrana neexistuje, vytvoríme ju pomocou metódy *addHyperEdge* a spojíme hranami `defaulttype`.

**Tabuľka 4-2: Metódy volané v RSF Importer**

| **Metóda** | **Parametre** | **Použitie** |
|---|---|---|
| `getHyperEdge` | `srcNodeName` – názov zdrojového uzla<br>`edgeName` – názov uzla (HyperNode)<br>`*mapa` – zoznam všetkých hrán vložených v grafe | vráti vrchol reprezentujúci hranu (hyperhranu) a umožní nám vkladať nové spojenia s danou hyperhranou |
| `addHyperEdge` | `edgeName` – názov uzla (HyperNode) | vytvorí vrchol reprezentujúci hyperhranu |

## *4.8 Reprezentácia vnorených grafov*

Vnorený graf predstavuje graf, ktorý sa nachádza v rámci uzla grafu, ktorý je vnorenému grafu nadradený. Takýto typ grafu nie je v dátovej časti aplikácie podporovaný, preto je potrebné doplnenie podpory tohto typu do dátového modulu. Návrh vychádza z existujúcej štruktúry, teda dátovej reprezentácie uzol, hrana, graf. Do týchto existujúcich prvkov, respektíve tried, sú doplnené atribúty a metódy potrebné pre vytváranie kompozitnej dátovej štruktúry na úrovni uzlov grafu, ktoré majú buď rodičovské uzly (nadradené), alebo obsahujú vnorené uzly.

### 4.8.1 Dátová reprezentácia

Princípom vnárania sa a tým pádom vytvárania stromovej štruktúry je to, že si každý prvok nižšej úrovne pamätá svojho rodiča, teda prvok z nadradenej úrovne. Pre to pridávame do dátovej triedy reprezentujúcej uzol *Node* atribút *nested_parent.* Metódy pre prístup k tomuto atribútu, ako aj metódy používané pri vytváraní a spravovaní vnorených grafov, nachádzajúce sa v triede *Graph*, sú uvedené v tabuľke 4-2. Pri vytváraní uzla aj hrany zároveň nastavíme jej škálu veľkosti, závislú od stupňa vnorenia, ktorá sa uloží do atribútu *scale*. Je to pre to, aby sme dodatočne nemuseli prechádzať celý graf znova a zisťovať úroveň vnorenia jednotlivých prvkov, keďže pri vytváraní týchto prvkov je ich vnorenie známe. Škálovanie dosiaľ nebolo podporované, jeho návrh je opísaný v kapitole 4.8.2. Z dátového hľadiska sú uvedené metódy a atribúty dostatočné pre reprezentáciu akéhokoľvek grafu vnoreného do uzla.

**Tabuľka 4-2: Metódy modulu data používané pri používaní dátovej reprezentácie vnorených grafov.**

| Metóda | Parametre | Použitie |
|---|---|---|
| `getNestedParent` | `žiadne` | Metóda triedy *Node* navráti rodičovský uzol typu *Node* |
| `setNestedParent` | `node – uzol, v rámci ktorého je daný uzol typu Node vnorený` | Metóda triedy *Node* nastaví rodičovský uzol typu *Node* |
| `createNode` | `scale – škála veľkosti uzla vzhľadom na jeho vnorenie`<br>`bbState – stav uzla` | Metóda triedy *Graph* vytvorí nový uzol, ktorý bude uložený do zoznamu v rámci jej inštancie |
| `createNestedGraph` | `srcNode – uzol, do ktorého vnárame graf` | Metóda triedy *Graph* uvedie jej inštanciu do stavu vytvárania vnoreného grafu. V tomto stave objekt typu *Graph* zaraďuje všetky prvky typu *Node* a *Edge* ako vnorené do uzla *srcNode* typu *Node* |
| `closeNestedGraph` | `žiadne` | Metóda triedy *Graph* uvedie jej inštanciu do stavu, v ktorom sa prvky začnú pridávať na vyššiu úroveň |

| | | |
|---|---|---|
| | | grafu |
| isInSameGraph | nodeA - uzol<br>nodeB - uzol | Metóda triedy *Graph* slúži na overenie, či dva uzly typu *Node* sa nachádzajú v rovnakom podgrafe. Táto metóda je využívaná pri vytváraní obmedzovačov rozloženia prvkov podgrafu, kedy je potrebné overiť, či sa dva uzly nachádzajú v jednom podgrafe. Ak áno, potom podliehajú tieto uzly rovnakému obmedzovaču. |

### 4.8.2 Grafická reprezentácia

Grafické odlíšenie prvkov vnoreného grafu od ostatných prvkov grafu bude ralizované zmenou farby prvkov na červenú. To sa netýka prvkov typu *HyperEdge* a *MultiEdge*, ktorým bude ponechaná ich vlastná farba. Farebné odlíšenie je použité kvôli lepšej prehľadnosti.

Zmena farby je realizovaná zavedením nových typov do zoznamu typov typu *Type*. Sú zavedené typy pre uzol a hranu, označené ako *NESTED_NODE_TYPE a NESTED_EDGE_TYPE*. Týmto typom je priradený farebný atribút nastavujúci ich červené zobrazenie.

Ďalším grafickým prvkom je zavedenie zobrazenia priehľadnej gule ohraničujúcej vnorený graf okolo jeho nadradeného uzla. Táto guľa bude mať polomer totožný s polomerom použitým pri vytváraní obmedzovača pre rozmiestnenie uzlov do gule okolo nadradeného grafu. Polomer vypočítava metóda *getGraphRadius* prináležiaca triede *Graph*. Trieda *Graph* pri vytváraní grafu odpamätáva úroveň aktuálne vkladaných prvkov. Táto úroveň je ďalej použitá pri výpočte škály veľkosti polomeru gule metódou *getGraphRadius*.

V tabuľke 4-3 je zoznam pridaných a upravených metód slúžiacich pre doplnenie a zmenu grafickej reprezentácie vnorených grafov.

**Tabuľka 4-3**

| **Metóda** | **Parametre** | **Použitie** |
|---|---|---|
| getNestedNodeType | žiadne | Metóda triedy *Graph* ktorá navracia typ *Type* nesúci grafické vlastnosti pre vnorenú hranu grafu |
| getNestedEdgeType | žiadne | Metóda triedy *Graph* ktorá navracia typ *Type* nesúci grafické vlastnosti pre vnorený uzol grafu |
| initNodes | žiadne | Metóda triedy *NodeGroup* |

| | | |
|---|---|---|
| | | zabezpečujúca inicializáciu grafického zobrazenia uzlov grafu |
| `getNodeGroup` | `node`<br>`parentEdge`<br>`graphScale` | Metóda triedy *NodeGroup* ktorá navracia zoznam uzlov grafu. Táto metóda bola zmenené tak, aby sa tdo zoznamu grafických prvkov spolu s uzlami ukladali pre nadradené uzly aj vykreslené ich ohraničujúce gule |
| `updateNodeCoordinates` | `interpolationSpeed` | Metóda triedy *NodeGroup* ktorá aktualizuje koordináty polohy uzlov grafu. Táto metóda bola upravená tak, aby aktualizovala aj pozície vykreslených gulí okolo vnoreného grafu |
| `getGraphRadius` | `žiadne` | Metóda triedy *Graph* navracia priemer gule zobrazenej okolo nadradeného uzla ako aj priemer pre obmedzovač vnoreného grafu |

## *4.9 Doplnenie podpory vnorených grafov a hyperhrán do parserov*

Existujúce parsery (*GraphML* a *GXL*) dokážu spracovávať zatiaľ iba jednoduché grafy, preto je potrebné pridanie podpory vnorených grafov a hyperhrán.

### 4.9.1 Vnorené grafy

Pridanie podpory vnorených grafov riešime oddelením časti kódu, ktorá spracováva graf, do samostatnej metódy (`processGraph`), aby sa dala rekurzívne zavolať pre každý nájdený podgraf (aj pre viac úrovní vnorenia).

Formáty *GraphML* a *GXL* obsahujú definície vnorených grafov priamo v definíciách uzlov alebo hrán, do ktorých sú vnorené. To umožňuje sekvenčné spracovávanie, takže si netreba odkladať informácie načítané zo súboru na neskoršie použitie. Možnosť takéhoto čítania súboru je výhodná pri *GXL* parseri využívajúcom `QXmlStreamReader` (ten poskytuje dáta postupne v takom poradí, v akom sa vyskytujú v súbore). *GraphML* parser využívajúci `QDomDocument` by sa dal použiť, aj keby vnorené grafy boli v súbore zapísané na inom mieste, keďže pri spracovávaní pomocou `QDomDocument` je celá štruktúra súboru načítaná v pamäti a môžeme pristupovať ku ktorejkoľvek jeho časti.

V tabuľke 4-5 sú uvedené metódy modulu *Data*, ktoré sa používajú pri pridávaní vnorených grafov.

**Tabuľka 4-4: Metódy modulu Data volané pri pridávaní vnorených grafov.**

| Metóda | Parametre | Použitie |
|---|---|---|
| `createNestedGraph` | `node` – uzol, v ktorom je graf vnorený | začiatok pridávania obsahu vnoreného grafu v danom uzle (uzly pridané po tomto volaní patria danému vnorenému grafu) |
| `closeNestedGraph` | (žiadne) | koniec pridávania obsahu vnoreného grafu (uzly pridané po tomto volaní patria znovu pôvodnému nadradenému grafu) |

### 4.9.2 Hyperhrany

Tabuľka 4-6 zobrazuje tagy v súboroch *GraphML* a *GXL*, ktoré súvisia s hyperhranami.

**Tabuľka 4-5: Tagy formátov GraphML a GXL slúžiace na zápis hyperhrán.**

| Formát | Hyperhrana | Koniec hyperhrany |
|---|---|---|
| *GraphML* | hyperedge | endpoint |
| *GXL* | rel | relend |

Tagy na určenie koncov hyperhrán sú zapísané vo vnútri tagu hyperhrany.

Tag hyperhrany je v *XML* štruktúre na rovnakej úrovni ako tagy uzlov a obyčajných hrán. Preto aj v zdrojovom kóde je spracovávanie na rovnakej úrovni ako spracovávanie uzlov a hrán. V *GraphML* parseri sú kvôli prehľadnosti oddelené časti kódu na spracovávanie týchto troch prvkov grafu do samostatných metód: `processGraph_Nodes`, `processGraph_Edges`, `processGraph_Hyperedges`.

V tabuľke 4-7 sú uvedené metódy modulu *Data* používané na pridávanie hyperhrán.

**Tabuľka 4-6: Metódy modulu Data volané pri pridávaní hyperhrán.**

| Metóda | Parametre | Návratová hodnota | Použitie |
|---|---|---|---|
| `addHyperEdge` | `name` – názov hyperhrany | `node` – uzol reprezentujúci hyperhranu | pridanie hyperhrany – získanie uzla, ktorý reprezentuje hyperhranu, aby sa pomocou neho dalo na hyperhranu odkazovať pri pridávaní koncov hyperhrany |
| `addEdge` | `name` – názov hyperhrany<br>`srcNode` – zdrojový | `edge` – pridaná hrana reprezentujúca časť hyperhrany | pridanie konca hyperhrany – pomocou pridania obyčajnej hrany, ktorá spája uzol na |

| | | | |
|---|---|---|---|
| | uzol<br>`dstNode` – cieľový uzol<br>`type` – typ hrany<br>`isOriented` – či je hrana orientovaná | | konci hyperhrany s uzlom reprezentujúcim hyperhranu (získaným z volania `addHyperEdge`) |

## *4.10 Obmedzovanie layoutu*

Pridávame možnosť určiť obmedzenie, kde v priestore sa môže nachádzať daný uzol. Zmeny zasahujú do modulu *Layout*:

- nové triedy na definovanie a výpočet obmedzení
- zásahy do behu layoutovacieho algoritmu (trieda `FRAlgorithm`)
  - zmena pozície navrhnutej obmedzovačom na pozíciu spĺňajúcu obmedzenie
  - ignorovanie uzlov s nastaveným parametrom `ignored` (napríklad kvôli pomocným uzlom, ktoré definujú tvar obmedzenia a nemôžu vplývať na iné uzly a tiež nemôžu byť presúvané layoutovacím algoritmom)

### 4.10.1 Princíp práce obmedzovača

Obmedzovač je začlenený do práce layoutovača (metóda layoutovača `applyForces`):

- layoutovač navrhne pozíciu uzla
- ak táto pozícia nespĺňa podmienku obmedzovača, obmedzovač navrhne novú pozíciu (ak pozícia navrhnutá layoutovačom už spĺňa podmienku, obmedzovač len vráti túto pôvodnú pozíciu)
- nová pozícia sa použije rovnakým spôsobom, ako keby ju navrhol layoutovač

### 4.10.2 Tvary obmedzenia

Sú navrhnuté základné tvary obmedzení uvedené v tabuľke 4-8.

**Tabuľka 4-7: Tvary obmedzení.**

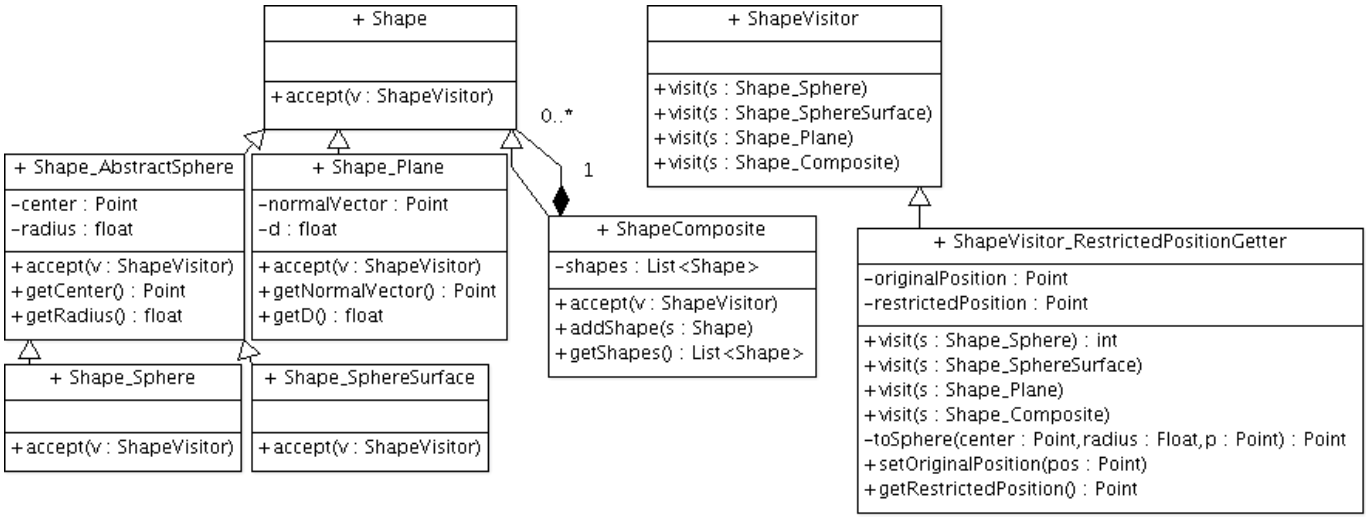
| **Trieda** | **Význam** | **Parametre obmedzenia** | **Hľadanie bodu spĺňajúceho obmedzenie** |
|---|---|---|---|
| `Shape_SphereSurface` | povrch gule | stred<br>polomer | nový bod je spoločný bod povrchu gule a priamky prechádzajúcej pôvodným bodom a stredom gule, bližšie k pôvodnému bodu |

| | | | |
|---|---|---|---|
| | | | (nový bod je teda najbližší bod na povrchu gule) |
| `Shape_Sphere` | guľa | stred<br>polomer | ak sa pôvodný bod nachádza v guli, zachová sa, inak sa vypočíta nový bod rovnako ako pri obmedzení na povrch gule |
| `Shape_Plane` | rovina | normálový vektor `n=[a,b,c]`<br>parameter `d`<br>(rovnica roviny je potom `ax + by + cz = d`) | nový bod je kolmý priemet pôvodného bodu do roviny (nový bod je teda najbližší bod v danej rovine) |
| `Shape_Composite` | zložený tvar | množina tvarov jednoduchých typov (rovina, povrch gule...) | nový bod je bod obmedzený do niektorého z daných jednoduchých tvarov – vyberá sa najbližší možný bod k pôvodnému bodu |

Keďže `Shape_SphereSurface` a `Shape_Sphere` majú rovnaké parametre (rovnaké členské premenné triedy), dedia od triedy `Shape_AbstractSphere`, ktorá uchováva dané parametre a obsahuje metódy na ich získanie.

Je výhodné použiť návrhový vzor *Visitor* – tvary môžu obsahovať iba svoje parametre spolu s metódami na ich získavanie, a práca s tvarmi môže byť oddelená v samostatných triedach. Zatiaľ je použitá jedna implementácia roly *Visitor* – výpočet nového bodu na základe tvaru a pôvodného bodu (trieda `ShapeVisitor_RestrictedPositionGetter`). Všetky výpočty nového bodu sú teda združené v jednej triede, čo je výhodné napríklad pri obmedzovaní podľa `Shape_SphereSurface` a `Shape_Sphere`, keďže používajú jednu spoločnú časť výpočtu (obmedzenie na povrch gule – metóda `toSphere`). Keďže metódy tried s rolou *Visitor* nemôžu mať parametre ani návratovú hodnotu (aby spĺňali rozhranie), vstup a výstup sa v tomto prípade uchováva v členských premenných `originalPosition` a `restrictedPosition`, ku ktorým sa pristupuje pomocou metód `setOriginalPosition` a `getRestrictedPosition`.

**Obrázok 4-7: Hierarchia tvarov.**




**Obrázok 26: Hierarchia tvarov**

### 4.10.3 Dynamické získavanie tvarov

Parametre tvaru, do ktorého chceme obmedziť pozíciu niektorého uzla, sa môžu v čase meniť, napríklad ak je obmedzenie pridané pomocou používateľského rozhrania a používateľ chce následne manipulovať s obmedzením (napríklad presúvať guľu, do ktorej je uzol obmedzený alebo meniť jej polomer).

Preto definícia obmedzenia nie je tvorená priamo inštanciou niektorej z tried definujúcich tvar, ale inštanciou potomka triedy `ShapeGetter`, ktorého metóda `getShape` vráti inštanciu niektorej z tried definujúcich tvar. Táto inštancia už presne definuje tvar a jeho parametre, ktoré sa majú v danom čase použiť.

Používané možnosti dynamického získavania tvarov sú uvedené v tabuľke 4-9.

**Tabuľka 4-8: Dynamické získavanie tvarov obmedzení.**

| **Trieda** | **Parametre** | **Vrátený tvar** |
|---|---|---|
| `ShapeGetter_Const` | `shape` – inštancia tvaru | stále rovnaký tvar (tvar uložený ako parameter) – náhrada za statické definovanie tvaru |
| `ShapeGetter_Plane_ByThreeNodes` | `node1`, `node2`, `node3` – 3 uzly grafu | vrátenie roviny určenej aktuálnou pozíciou troch uzlov |
| `ShapeGetter_Sphere_AroundNode` | `node` – uzol grafu<br>`radius` – polomer gule | guľa, ktorej stred je určený aktuálnou pozíciou daného uzla; polomer je konštantný |

| | | |
|---|---|---|
| `ShapeGetter_Sphere_ByTwoNodes` | `centerNode`, `surfaceNode` – uzly grafu | guľa, ktorej stred je určený aktuálnou pozíciou uzla `centerNode` a polomer vzdialenosťou uzlov `centerNode` a `surfaceNode` (`surfaceNode` je teda uzol na povrchu danej gule) |
| `ShapeGetter_SphereSurface_ByTwoNodes` | `centerNode`, `surfaceNode` – uzly grafu | povrch gule určenej polohou dvoch uzlov rovnako ako v predchádzajúcom prípade |

### 4.10.4 Správa obmedzení

Priradenia tvarov definujúcich obmedzenie (presnejšie inštancií potomkov triedy `ShapeGetter`, ktoré dokážu dané tvary poskytovať) k uzlom obsahuje trieda `RestrictionsManager`.

Jej dve základné zodpovednosti sú uvedené v tabuľke 4-10.

**Tabuľka 4-9: Zodpovednosti triedy RestrictionsManager.**

| **Zodpovednosť** | **Metóda** | **Parametre** | **Použitie** |
|---|---|---|---|
| určovanie obmedzení | `setRestrictions` | `nodes` – množina uzlov, ktoré majú byť obmedzené daným tvarom<br>`shapeGetter` – určenie obmedzenia | – pridávanie obmedzení cez používateľské rozhranie<br>– pridávanie obmedzení potrebných na prehľadné vykreslenie vnorených grafov |
| získanie bodu | `applyRestriction` | `node` – uzol, pre ktorý | – layoutovač – |

| | | | |
|---|---|---|---|
| spĺňajúceho obmedzenie | | získavame obmedzenie<br>`originalPosition` – pôvodná pozícia, ku ktorej chceme nájsť novú (spĺňajúcu obmedzenie) | zmena navrhnutej pozície uzla na pozíciu, ktorá spĺňa obmedzenie |

Pri určovaní obmedzení, ak niektoré uzly v množine `nodes` už majú priradené obmedzenie, nahradí sa pôvodné obmedzenie novým. Ak `shapeGetter` je `NULL`, obmedzenie všetkých uzlov z množiny sa vymaže.

Pri získavaní bodu spĺňajúceho obmedzenie sa najprv nájde `ShapeGetter` k danému uzlu. Od `ShapeGetter` sa získa tvar, do ktorého má byť uzol v danom čase obmedzený. Pomocou typu tvaru a jeho parametrov sa vypočíta obmedzenie a vráti sa nový bod (alebo pôvodný, ak uzol nemá priradené žiadne obmedzenie alebo ak jeho pozícia už vyhovuje obmedzeniu).

Každý graf (inštancia triedy `Data::Graph`) obsahuje svoju vlastnú inštanciu triedy `RestrictionsManager`.

### 4.10.5 Pridávanie obmedzení pomocou používateľského rozhrania

Do používateľského rozhrania sú pridané tlačidlá na pridanie obmedzení: na povrch gule, do gule a do roviny. Interakcia prebieha nasledovne:

2. Používateľ vyberie uzly, ktorých pozícia má byť obmedzená.
3. Zvolí pridanie obmedzenia (tlačidlom podľa typu obmedzenia).
4. Pridajú sa pomocné uzly (do stredu výberu), ktoré definujú tvar obmedzenia (pre guľu 2 uzly – stred a povrch; pre rovinu 3 uzly).
5. Presúvaním pomocných uzlov používateľ definuje polohu a iné parametre tvaru obmedzenia.

## *4.11 Doplnenie funkcií pre spájanie a rozklikávanie uzlov*

### 4.11.1 Spájanie uzlov

K vyznačeným uzlom, ktoré chceme spojiť, pridáme meta uzol a spojíme ho hranami s jednotlivými vybranými uzlami. Aby sme tento meta uzol, ktorý použijeme ako spoločný uzol pre všetky vybrané uzly, odlíšili od ostatných meta uzlov, zafarbíme ho na modro (obyčajné meta uzly sa automaticky farbia na červeno). Podľa počtu vybraných uzlov nastavíme meta uzlu veľkosť, kde

*veľkosť meta uzla = štandardná veľkosť uzla + (počet vybraných uzlov / 2)*

pričom štandardná veľkosť uzla je nastavená na hodnotu 8, t. j. pre 1 vybraný uzol bude veľkosť pridaného meta uzla približne rovnaká ako je veľkosť ostatných uzlov a so zväčšujúcim sa počtom spájaných uzlov, bude rásť aj veľkosť pridaného meta uzla. Taktiež musíme overovať, či pri spájaní máme nejaké uzly vybrané, t. j. aby sme neumožňovali

pridanie meta uzla, ak nemáme žiadne uzly vybrané, pretože obyčajné meta uzly môžeme pridávať aj do prázdneho grafu.

Vytvorený meta uzol potom hranami spojíme s uzlami, s ktorými sú spojené uzly, ktoré spájame. Hrany, ktorými sú spojené vybrané uzly navzájom, ignorujeme.

Keď už máme vytvorený meta uzol a všetky potrebné hrany, musíme všetky pôvodné (vybrané) uzly a hrany skryť, aby to vyzeralo ako keby sme ich spojili do jedného vytvoreného meta uzla. Uzly môžeme skryť pomocou funkcie *setNodeMask(0)*, ktorú obsahuje trieda *osg::Node*. Pre hrany však takáto funkcia neexistuje, tie preto skryjeme tak, že nastavíme šírku vybraných hrán na veľkosť 0. Medzi tieto hrany patria:

- hrany, ktorými sú spojené vybrané uzly s meta uzlom
- hrany, ktorými sú spojené vybrané uzly navzájom
- hrany, ktorými sú spojené vybrané uzly s ostatnými uzlami

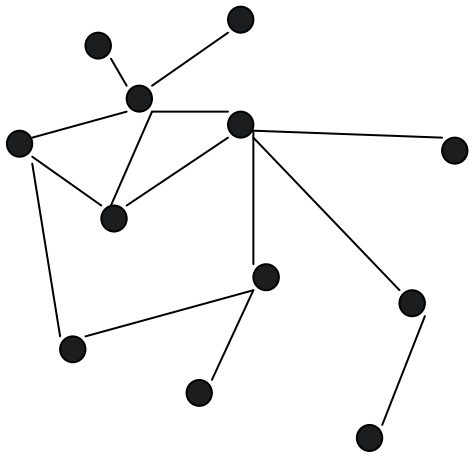
Takto zostanú viditeľné len:

- hrany, ktorými je spojený vytvorený meta uzol s ostatnými (nevybranými) uzlami, s ktorými boli spojené vybrané (spájané) uzly.
- vytvorený meta uzol

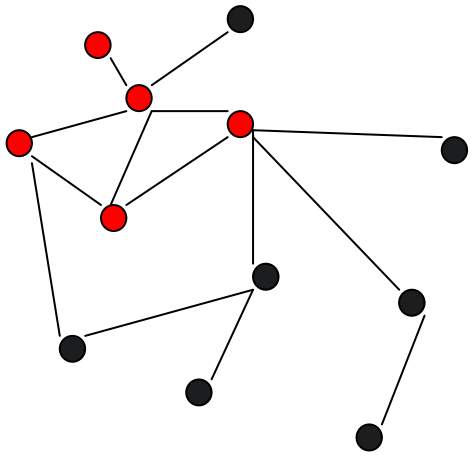
Takto skryté uzly a hrany nie je možné označiť používateľom a ten ich teda nemôže presúvať alebo vymazať.

Označeným (spájaným) uzlom nastavíme pozíciu, ktorú má vytvorený meta uzol, t. j. pozíciu v strede medzi vybranými uzlami. Táto pozícia sa určuje pomocou funkcie *getSelectionCenter(true)*. Týmto zabezpečíme, že spájané uzly sa rozmiestnia okolo vytvoreného meta uzla (vytvoria akoby podgraf) a budú sa pohybovať spolu s meta uzlom, pretože sú s ním navzájom poprepájané).

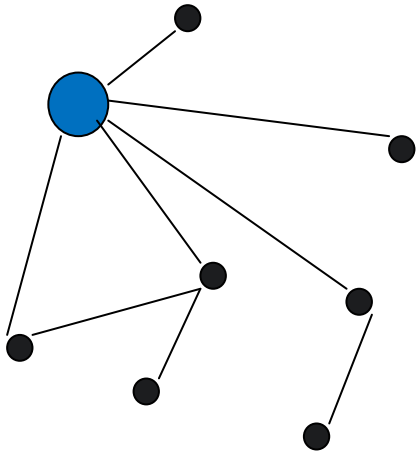
Nakoniec vybrané uzly a hrany odznačíme, pretože keby ostali označené, používateľ by s nimi ešte mohol pohybovať.



**Obrázok 4-8: Pôvodný graf.**

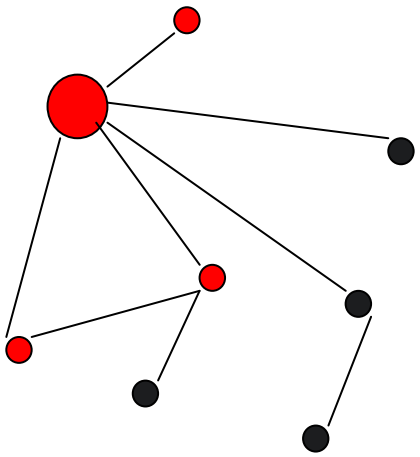


**Obrázok 4-9: Vybrané uzly v grafe, ktoré chceme spojiť do jedného (červené uzly).**

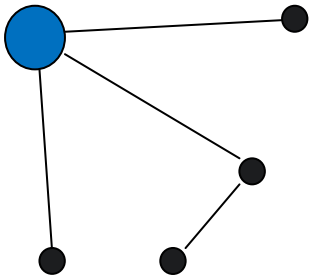


**Obr. 4-10: Vytvorený (spoločný) meta uzol.**

Vytvorený (meta) uzol má tie isté vlastnosti ako klasický uzol a môžeme ho teda s ďalšími vybranými uzlami opäť spojiť do nového meta uzla (obr. 4-11 a 4-12).



**Obr. 4-11: Nové vybrané uzly.**



**Obr. 4-12: Spojenie vybraných uzlov do nového meta uzla.**

### 4.11.2 Rozklikávanie uzlov

Pre vybrané uzly, ktoré chceme rozdeliť, najskôr kontrolujeme, či tieto uzly sú skutočne spojené uzly. Postup pri rozklikávaní uzlov je opačný ako pri ich spájaní.

Najskôr zviditeľníme skryté uzly a hrany. Prechádzame jednotlivé hrany, ktorými je spoločný uzol spojený so skrytými uzlami a tieto uzly odkryjeme opäť pomocou funkcie *setNodeMask(~0).* Potom kontrolujeme hrany odkrytých uzlov a ak sú skryté a ak nie sú spojené s inými skrytými uzlami (napr. ak máme uzly skryté vo viacerých vrstvách vnorených v sebe), tak ich zviditeľníme tak, že im nastavíme štandardnú veľkosť, ktorú získame pomocou funkcie *getEdgeScale().*

Potom im znovu nastavíme pozíciu spoločného meta uzla a pomocou už existujúcej funkcie *removeNode()* nakoniec odstránime spoločný meta uzol. Táto funkcia automaticky odstráni aj hrany, ktorými je meta uzol spojený s inými uzlami (so skrytými aj viditeľnymi). Pozíciu zmazaného meta uzla odkrytým uzlom nastavujeme preto, aby sme dosiahli efekt, že uzly sa začnú rozmiestňovať v priestore z jedného (spoločného) bodu a bude to vyzerať akoby skutočne boli spojené v jednom bode.

### 4.11.3 Doplnenie GUI aplikácie

Nakoniec je potrebné doplniť používateľské rozhranie aplikácie, aby mohol používateľ navrhnuté funkcie používať.


**Obr. 4-13: Návrh doplnenia používateľského rozhrania aplikácie.**

Doplnené budú 2 tlačidlá (obr. 4-13):

- tlačidlo *mergeNodes* (  ), ktoré bude volať funkciu na spájanie uzlov
- tlačidlo *separateNodes* (  ), ktoré bude volať funkciu na rozdeľovanie uzlov

## 4.12 Doplnenie funkcionality aplikácie pre prácu s databázou

Požadovaná funkcionalita:

1. Ukladanie viacerých layoutov pre jeden graf, t. j. každý graf môže mať v databáze uložených viacero rozmiestnení uzlov
2. Opravenie načítavania typov uzlov a hrán z databázy, t. j. rozlišovanie klasických uzlov a meta uzlov (a hrán)
3. Ukladanie a načítavanie atribútov uzlov a hrán, napr. ich farbu
4. Pridanie možnosti odstrániť vybrané grafy uložené v databáze
5. Pridanie možnosti odstrániť vybrané layouty jednotlivých grafov uložené v databáze
6. Doplniť používateľské rozhranie aplikácie pre implementované funkcie

### 4.12.1 Ukladanie viacerých layoutov pre jeden graf

Pri ukladaní viacerých layoutov je potrebné ukladanie pozícií jednotlivých uzlov.

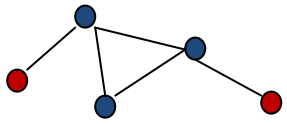
Máme 2 typy uzlov:

- klasické uzly
- meta uzly (používateľom pridané uzly)

Klasické uzly sa po načítaní grafu zo súboru nedajú do grafu pridávať ani z neho odstrániť, preto stačí ak ich do databázy uložíme len raz a pre každý layout budeme ukladať len ich pozície.

Pri meta uzloch je situácia iná, pretože tie môžeme kedykoľvek do grafu pridávať aj ich z neho zmazať. Nestačí preto len ukladať ich pozície do databázy, ale musíme si aj pamätať ktoré meta uzly patria ku ktorému layoutu uloženému v databáze. Tieto meta uzly preto budeme pre každý layout ukladať do databázy zvlášť vždy s inými ID, keďže v databáze nemôžu byť uložené dva uzly s rovnakým ID uzla aj ID grafu.

Napríklad:



**Obr. 4-14: ukážka jednoduchého grafu s meta uzlami.**

Graf obsahuje 3 klasické uzly (modré) s ID 1, 2 a 3 a 2 meta uzly (červené) s ID 4 a 5.

Layouty sa do databázy uložia tak, ako je to zobrazené v tabuľke 4-11.

**Tabuľka 4-10: Ukážka priradenia ID uzlom.**

| **1. layout – uloží sa pri načítaní grafu zo súboru** | **2. layout** | **3. layout** |
|---|---|---|
| graph ID = 1 | graph ID sa nemení | graph ID sa nemení |
| layout ID = 1 | layout ID = 2 | layout ID = 3 |
| klasický uzol ID = 1<br>klasický uzol ID = 2<br>klasický uzol ID = 3 | ID klasických uzlov sa nemenia, uložia sa len ich nové pozície | ID klasických uzlov sa nemenia, uložia sa len ich nové pozície |
| meta uzol ID = 4<br>meta uzol ID = 5 | meta uzol ID = 6<br>meta uzol ID = 7 | meta uzol ID = 8<br>meta uzol ID = 9 |

Podobne ako s uzlami, to isté platí aj pre hrany. Klasické hrany sa uložia len raz a meta hrany sa ukladajú zvlášť pre každý layout, podobne ako uzly.

Aby sme vedeli určiť aké ID máme priradiť jednotlivým meta uzlom (resp. hranám), musíme pri každom ukladaní nového layoutu zistiť z databázy maximálne ID uzla uloženého

pre daný graf. Potom si vytvoríme tabuľku, kde každému meta uzlu, ktorý chceme uložiť do databázy priradíme nové ID také, aby bolo pre daný graf v databáze jedinečné. Potom pri ukladaní jednotlivých prvkov do databázy (či už meta uzlov, meta hrán, pozícií meta uzlov, ich farby, ...) vždy pozrieme do tabuľky či sa tam daný prvok nenachádza, a ak áno, do databázy uložíme jeho nové ID.

V súčasnosti sa ID uzlov aj hrán určujú automaticky, t. j. pomocou premennej `ele_id_counter`, ktorej hodnota sa automaticky zvyšuje po každom pridanom elemente do grafu. Aby sme zabezpečili, že grafy budú z databázy do dátových štruktúr načítané tak, ako sú uložené v databáze, je potrebné doplniť funkcie pre pridávanie uzlov a hrán o parameter, ktorý predstavuje ID prvku.

### 4.12.2 Opravenie načítavania typov uzlov a hrán z databázy

V súčasnosti sa všetky prvky (uzly a hrany) z databázy načítavajú ako klasické prvky a teda ich nie je možné z grafu odstrániť.

Je potrebné pomocou funkcií *getNodeMetaType()* a *getEdgeMetaType()* zistiť typ meta uzlov a hrán používaných v grafe a tento typ nastaviť prvkom načítaným z databázy.

### 4.12.3 Ukladanie a načítavanie atribútov uzlov a hrán

V súčasnosti je v aplikácií spomedzi atribútov možné používať len rôznu farbu uzlov a hrán.

Táto časť teda predstavuje implementáciu funkcií pre ukladanie farieb uzlov a hrán:

- *addNodesColorToDB()*
- *addEdgesColorToDB()*

Do databázy nie je potrebné ukladať farby všetkých uzlov a hrán, stačí ak uložíme farby prvkov, ktoré majú farbu inú než je automaticky nastavená, t. j. biela. Preto pre každý prvok pri ukladaní najskôr skontrolujeme, či má inú farbu ako bielu a ak áno, potom túto farbu uložíme do databázy.

Načítaným prvkom potom môžeme farbu nastaviť pomocou funkcie *setColor(osg::Vec4 color)*.

### 4.12.4 Pridanie možnosti odstrániť vybrané grafy uložené v databáze

Vždy keď načítame nejaký graf zo súboru, ten sa nám automaticky pridá aj do databázy. Aby sa nám tieto grafy zbytočne nehromadili, bolo by dobré, aby sme vybrané grafy mohli z databázy odstrániť.

Pri zmazaní grafu sa taktiež vymažú aj všetky prvky spojené s daným grafom: layouty, uzly, hrany, ich atribúty, ... .

Funkcie pre odstránenie grafov a ich prvkov sú už implementované, ide najmä o doplnenie používateľského rozhrania. To je popísané v časti 4.12.6 (Doplnenie používateľského rozhrania).

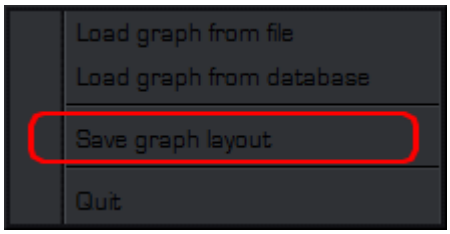
### 4.12.5 Pridanie možnosti odstrániť vybrané layouty uložené v databáze

Pre každý graf môžeme maž v databáze uložených viacero layoutov. Je potrebné implementovať funkciu *removeLayout(qlonglong graphID, qlonglong layoutID, QSqlDatabase* conn)*, ktorá na základe zadaných parametrov (ID grafu a ID layoutu) vymaže tento layout aj všetky jeho časti (meta uzly a hrany spojené s layoutom, ich atribúty, pozície) z databázy.

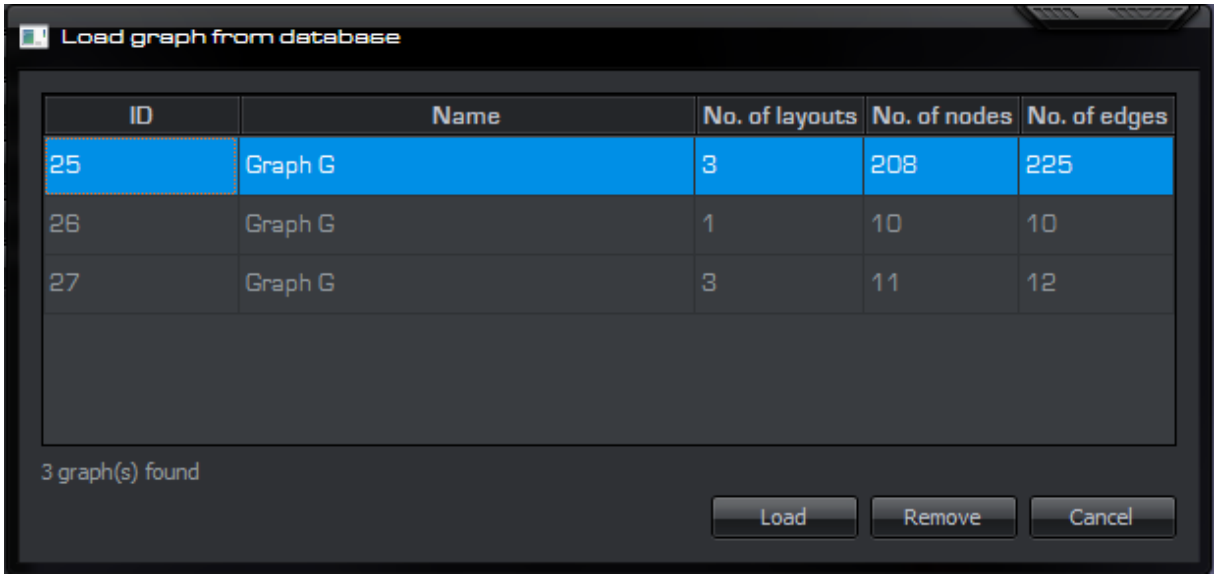
### 4.12.6 Doplnenie používateľského rozhrania aplikácie

Je potrebné doplniť:

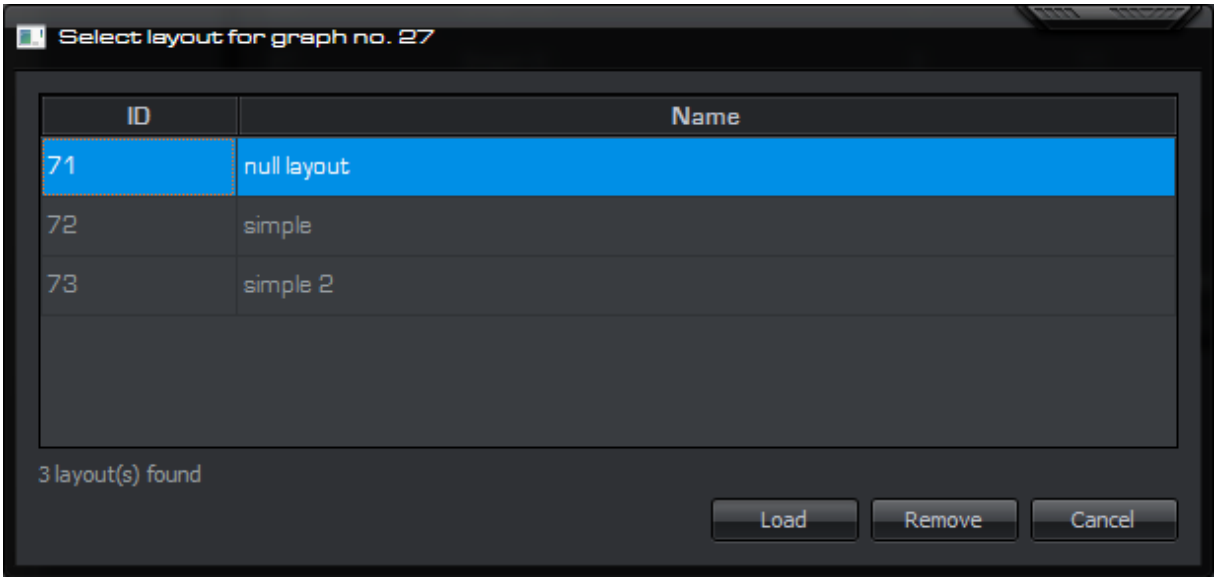
- používateľské menu – pridanie možnosti uložiť aktuálny layout grafu do databázy (obr. 4-15)
- implementácia dialógu pre výber grafu z databázy (obr. 4-16)
- implementácia dialógu pre výber layoutu grafu z databázy (obr. 4-17)




**Obr. 4-15: Doplnenie používateľského rozhrania.**




**Obr. 4-16: Okno pre výber grafu z databázy.**




**Obr. 4-17: Okno pre výber layoutu grafu z databázy.**

# 5 Prototyp

## *5.1 Nefunkcionálne zmeny v projekte*

Počas analýzy zdrojových kódov vytvorených predchádzajúcim tímom a počas prispôsobovania tohto kódu najnovším verziám knižníc sme identifikovali niekoľko potrebných zmien.

#### Prispôsobenie najnovšej verzii knižnice *OpenSceneGraph*

Trieda `osgGA::MatrixManipulator` z knižnice *OpenSceneGraph* bola v novej verzii knižnice premenovaná na `CameraManipulator`.

Bola použitá v triede `Viewer::CameraManipulator` (táto trieda od nej dedila). Preto je nutné nahradiť názov `osgGA::MatrixManipulator` názvom `osgGA::CameraManipulator`.

Zmena zasiahne:

- súbory triedy `Viewer::CameraManipulator`

#### Trieda `Layout::Layout` a závislosť na knižnici *igraph*

Knižnica *igraph* sa používala na rozmiestnenie uzlov grafu (používanie knižnice bolo implementované v triede `Layout::Layout`). Táto funkcionalita bola nahradená triedou `FRAlgorithm`. Trieda `Layout::Layout` môže byť vymazaná a tým sa tiež odstráni závislosť od triedy *igraph*.

Zmena zasiahne:

- súbory triedy `Layout::Layout` (vymazanie)
- *CmakeLists.txt* na najvyššej úrovni (kvôli vymazaniu zdrojových súborov a odstráneniu závislosti na knižnici *igraph*)

#### Adresár *dependencies* a nepoužívané knižnice

V adresári *dependencies/source* sa nachádzajú niektoré z knižníc, ktoré aplikácia používa. Keďže sa zdrojové kódy niektorých knižníc, ktoré sú tu umiestnené, nepoužívajú (sú používané inštalácie knižníc vyhľadávané v systéme), môžeme ich odstrániť.

Zmena zasiahne:

1. adresár *dependencies/source* (zmazanie nepotrebných knižníc)
2. *CmakeLists.txt* v *dependencies/source* (nebude odkazovať na zmazané knižnice)

#### Oddelenie importu zo súborov od triedy `Manager::GraphManager`

Trieda `Manager::GraphManager` (slúžiaca na správu grafov, s ktorými pracujeme) obsahuje aj funkcionalitu importu zo vstupného súboru (formát *GraphML*).

Import zo súboru oddelíme do samostatného modulu, čo nám umožní lepším spôsobom pridať kód na podporu ďalších formátov, ktoré budeme chcieť importovať. Takýto nový modul teda bude zabezpečovať načítavanie dát zo súborov rôznych typov do reprezentácie, s ktorou bude pracovať aplikácia.

Budeme musieť navrhnúť vhodnú štruktúru zdrojových kódov modulu, aby mali napríklad triedy slúžiace na import formátov rovnaké rozhranie.

Modul by mal vedieť spracovávať rozsiahle dáta. Nemal by teda pracovať tak, že načíta celý súbor do pamäte a až následne prechádzaním takto vytvorenej štruktúry ju bude prevádzať do tvaru, aký potrebujeme v aplikácii. Import teda treba navrhnúť tak, aby priebežne, počas čítania súboru, poskytoval informácie o čítaných dátach, ktoré budeme môcť hneď ukladať na miesta v štruktúrach, kam patria.

Ukladanie importovaných dát do cieľových štruktúr by malo byť dostatočne abstraktné, lebo je možnosť napríklad importovať priamo do databázy a nie do štruktúr v pamäti.

Zmena zasiahla:

1. súbory triedy `Manager::GraphManager`
2. adresárovú štruktúra projektu (pribudne adresár nového modulu)
3. *CmakeLists.txt* na najvyššej úrovni (kvôli pridaniu nových zdrojových súborov tvoriacich nový modul)

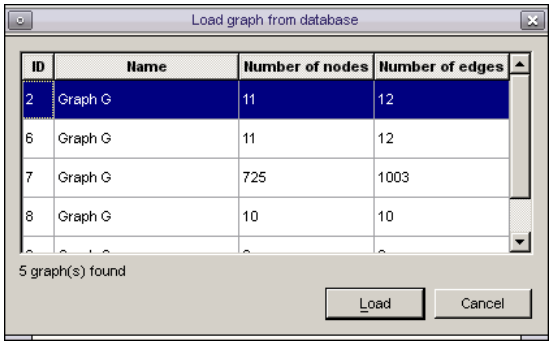
## 5.2 Overenie prototypu

### 5.2.1 Podpora importu formátu GXL

Implementovali sme import základných grafov (bez hyperhrán a vnorených grafov) z formátu GXL. Okrem súborov s príponou formátu GraphML sa teraz v používateľskom rozhraní dajú vybrať aj súbory GXL. Výber správnej implementácie na import súboru sa zatiaľ vykonáva iba na základe prípony vstupného súboru.

### 5.2.2 Dokončenie ukladania grafov do databázy

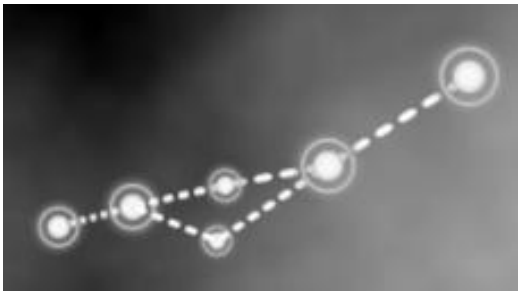
Doplnili sme ukladanie hrán, uzlov a rozložení grafu do databázy. Bola doplnená štruktúra databázy a pribudli možnosti používateľského rozhrania pre prácu s databázou – ukladanie a načítavanie grafov (Obr. 5-1).




**Obrázok 5-1: Používateľské rozhranie načítavania grafu z databázy.**

### 5.2.3 Pridanie podpory multihrán

Bol doplnený modul *Data* o pridávanie pomocných uzlov, ktoré ovplyvňujú rozloženie multihrán (Obr. 5-2).



**Obrázok 5-2: Zobrazenie grafu s jednou multihranou.**

### 5.2.4 Oddelenie importu do samostatného modulu

Bol vytvorený nový modul *Importer*. Použili sme zjednodušený návrh architektúry, takže modul ešte neobsahuje všetky pôvodne navrhované vlastnosti (podpora importu veľkých dát a podpora rôznych spôsobov ukladania importovaných dát), ktoré boli obsiahnuté v pôvodne navrhovanej architektúre.

## *5.3 Zhodnotenie prototypu*

Splnili sme definované funkcionálne požiadavky, pričom sme sa snažili dodržiavať nefunkcionálne požiadavky. Prototyp vyhovuje:

- požiadavke používať knižnice *QT* a *OSG* na zabezpečenie funkcionality, ktorú neobsahujú základné šablóny z knižnice *STL* (súčasne sme sa vyhli pridávaniu závislostí projektu od iných knižníc)
- požiadavke podporovať platformy Windows a Linux (prototyp je skompilovateľný, spustiteľný a použiteľný na obidvoch týchto platformách)

Navyše boli pridané možnosti nastavenia aplikácie, ktoré uľahčili prácu s aplikáciou počas vývoja:

- nastavenie veľkosti hlavného okna aplikácie po jej spustení
- nastavenie určujúce, že k databáze sa dá pripojiť aj bez zabezpečeného spojenia (*SSL*)

## Použitá literatúra


[1] HALUŠKA, M., KUTENICSOVÁ, D., LYSINA, A., RADOŠINSKÝ, M., REPÁŇ, V., RUTTKAY-NEDECKÝ, I.: 3D vizualizácia softvérových artefaktov. FIIT STU, 2009. Dokumentácia k tímovému projektu.

[2] GABURA, Š., PAPRČKA, M., PAULOVIČ, A., PAVLÍK, M., PAŽITNAJ, A., PÉČI, M., PERDÍK , P.: Vizualizácia softvérových artefaktov v 3d priestore . FIIT STU, 2010. Dokumentácia k tímovému projektu. Dostupné na: http://labss2.fiit.stuba.sk/TeamProject/2009/team20is-si/wp-content/uploads/2009/10/tp_dokumentacia1.pdf

[3] Erwin Coumans: Bullet 2.76 Physics SDK manual

[4] HOLT, R., SCHÜRR, A., SIM, S. E., WINTER A.:Graph eXchange Language. Dostupné na: http://www.gupro.de/GXL/

[5] BRANDES, U., EIGLSPERGER, M., LERNER, J.: GraphML Primer. Dostupné na: http://graphml.graphdrawing.org/primer/graphml-primer.html

[6] UKROP, J: Zavedenie obmedzení do vizualizácie grafov založenej na pružinovom modeli. FIIT STU Bratislava. 2009. Bakalársky projekt.