Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Irányítástechnika és Informatika Tanszék

Ábrahám Zoltán Marcell

ORVOSI ADATOKAT MEGJELENÍTŐ ÉS FELDOLGOZÓ PROGRAM FEJLESZTÉSE

BSc - Szakdolgozat

KONZULENS

Dr. Magdics Milán

BUDAPEST, 2021

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott **Ábrahám Zoltán Marcell**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2021. 12. 09.

...…….
Ábrahám Zoltán Marcell

Összefoglaló

A PET rekonstrukció kutatása során rengeteg adat keletkezik, amiket mennyiségük, jellegük vagy diverzitásuk miatt bonyolult lehet ellenőrizni egy erre specializált szoftver nélkül. A szakdolgozatomban ennek a szoftvernek a tervezéséről és implementációjáról fogok beszélni. A célom előállítani egy grafikus felülettel rendelkező szoftvert, ami képes elemezni, feldolgozni és vizuálisan ábrázolni a statikus és dinamikus PET rekonstrukció során keletkező adatokat. A fejlesztés célja, hogy ez a szoftver legyen intuitíven használható, és könnyen bővíthető, mivel a szoftvert egy kutatás támogatására fogják felhasználni, így fontos, hogy a programot is könnyen lehessen adaptálni a kutatás változó körülményeihez és követelményeihez.

A dolgozatomban áttekintem a rekonstrukció során keletkező adatokat, és ezeknek a kiértékelési és összehasonlítási módszereiket. Felvázolom a tervezés során figyelembe vett körülményeket, illetve követelményeket. Bemutatom az implementációhoz választott GUI framework-öt, és ennek kiválasztásának a szempontjait. Végül pedig funkciókra bontva bemutatom ezeknek a célját, ahol lényeges, a mögöttes elméletét és implementációját a programban. A szagdolgozat végén pedig a lefektetetett követelmények alapján értékelem az elkészült szoftvert.

Abstract

PET reconstruction research generates a large volume of specialized data, which can be difficult to work with this without specialized software due to its quantity, type, and diversification. In my thesis, I will talk about the planning and implementation of this specialized software. My goal is to develop a software, which can analyze, evaluate, and visualize data generated during static and dynamic reconstruction. This software should be intuitive to use, and easy to maintain and expand, since it supports a research project, and it should be easily adaptable for the changing circumstances and requirements.

In my thesis, I will review the data generated during the reconstruction, and its evaluation and comparison methods. I will talk about the circumstances and the requirements of the software, and the planning phase. I will present the GUI framework, which I chose to implement the project with, and the reasoning behind this choice. And finally, I will talk about the different functions, the underlying theory, and the implementation of the given function. At the end of my thesis, I will review the finished software.

1 Bevezetés

1.1 Motiváció

Leendő mérnökként feladatomnak érzem, hogy ahol és amikor lehet, tudásommal és időmmel másoknak segíthessek. Emiatt kifejezetten szimpatikus az orvosi és egészségügyi téma. Egy PET-hez kapcsolódó kutatás támogatásával hozzájárulhatok, hogy a jövőben az egészségügynek jobb eszközök állhassanak rendelkezésre, amivel hatékonyabb lehet a gyógyítás. Örülök, hogy ilyen témát választhattam, aminek hosszútávon is haszna van, és a hozzátartozó kutatás segíthet embereken, és nem egy sokadik tömeggyártott alkalmazást kell elkészítenem.

1.2 Igény a szoftverre

A szoftver az IIT tanszék egy PET szkennerekhez kapcsolódó kutatásának támogatására készült. A kutatás célja a szkennerek hatékonyságának fejlesztése, nagyobb felbontású képek és pontosabb, gyorsabb kiértékelés. A kutatás során rengeteg három- és négydimenziós volumetrikus adat keletkezik, amiket több különböző szempont alapján kell kiértékelni, hogy meghatározható legyen a kutatás alatt éppen vizsgált algoritmus sikeressége, vagy éppen bukása. Ez az adatok mennyisége és jellege miatt egy komplex és időigényes feladat. A legtöbb esetben ezek az adatok megjelenésben csak a voxelek értékei felsorolva egymás után, amit automatizálás nélkül szinte lehetetlen értelmezni. A felsoroltak miatt született meg az igény egy olyan szoftverre, ami képes egy teljes PET rekonstrukció adatait beolvasni, emberként is értelmezhető formában és formátumban grafikusan megjeleníteni, illetve különböző kiértékeléseket végezni rajtuk.

1.3 PET áttekintés

A Pozitronemissziós Tomográfia (PET) egy modern képalkotó módszer, amely képes a testről vagy annak egy területéről egy háromdimenziós képet adni. Például egy röntgennel ellentétben nem csak tisztán a kép megalkotására, hanem metabolikus folyamatok változásának mérésére is kiválóan alkalmazható. Működési elve a következő: Úgynevezett tracer-t, vagyis radioaktív anyagot juttatnak a vérbe, ami a vérárammal a magas sejtaktivitású területekre jut. A tracer anyagot a szkenner detektorai érzékelni képesek, és különböző valószínűségszámításokon alapuló, erősen számításintenzív

iteratív algoritmusok [1] alapján képes kiszámítani a radioaktív anyag koncentrációját. A vizsgált térfogat, amiben a szkennelni kívánt alany található, voxelekre (térbeli pixelekre) van osztva. A végleges háromdimenziós képet a szkenner az egyes voxelek intenzitásának meghatározásával tudja előállítani, vagyis meghatározza, hogy az egyes voxelekben mekkora a radioaktív anyag koncentrációja a kibocsátott récsecskék mennyiségének kiszámolásával.

A PET rekonstrukció célja meghatározni a tracer által kibocsátott részecskék térbeli eloszlását. Kétféle rekonstrukciót különböztethetünk meg: a statikusat és a dinamikusat [2]. A kettő között az a különbség, hogy a statikusnál voxelenként csak egy értékünk lesz, és ennek az időbeli változására nincs szükség, csak a végeredményére. A dinamikusnál viszont ez alatt az idő alatt több mintát is veszünk egy voxelhez, és voxelenként egy időgörbét próbálunk rekonstruálni az aktuális értékeikből. Ez akár úgy is elképzelhető, mint a (hosszú expozíciós idejű) fénykép és videó közötti különbség.

Az időgörbe a dinamikus rekonstrukcióhoz tartozó fontos fogalom [3]. Ez egy, a dinamikus rekonstrukció különböző időpillanatainak voxeleinek értékére illesztett görbe. Ennek meghatározásához szükség van a voxelek értékeire a különböző időpillanatokban, amik diszkrét értékek és a voxel aktivitásának az egyes időablakokra vett időintegrálját adják meg. Ezekre a diszkrét értékekre különböző, néhány paraméter által leírt biológiai indíttatású modeleket (ún. *kinetikus modeleket*) próbálunk illeszteni, amiből már meghatározható ez a tetszőleges időpillanatban kiértékelhető, folytonos görbe. A dinamikus rekonstrukciós folyamatnak része a kinetikus model illesztése, a program kimenete így tartalmazza az illesztett model paramétereit minden egyes voxelre, egy ún. *parametrikus volumetrikus adat* formájában. Emellett gyakori feladat, hogy a rekonstrukció eredményére utólag más kinetikus modelt vagy más algoritmus szerint illesszünk, azaz nagy igény van egy utólagos görbeillesztő eszközre is.

A dinamikus PET vizsgálatok gyakori célja, hogy különféle anyagoknak (pl. gyógyszerek) a vérből a test többi szövetébe történő terjedését, diffúzióját nyomon kövessük. A leggyakrabban használt kinetikus modelekben ebből kifolyólag megjelenik a tracer kezdeti, vérben mért aktivitása, a *véraktivitás-függvény*, melyet a rekonstrukció szintén figyelembe vesz.

2 Követelmények elemzése

Egy szoftver megtervezésének egyik első fontos lépése a követelmények pontos ismerete és elemzése, mivel ezek ismerete nélkül nincsenek meg azok a stabil alapok, amikre a jövőben is építhetjük a szoftvert. A fejlesztési ciklusban természetesen gyakran változnak ezek a követelmények, de ezt a problémát ki lehet védeni különböző szoftverfejlesztési technikák alkalmazásával, hogy a változtatás minél kevesebb értékes fejlesztési időt vegyen igénybe. Ennek ellenére mindenképp segít a részletes követelmény lista, ennek kidolgozása több szempontból is előnyös: Az egyik, hogy a megrendelő is részletesebben át tudja gondolni az igényeit, így később kevesebb utólagos módosítási szándéka lesz. A másik, hogy a fejlesztő is pontosabban átlássa a feladatot már a projekt kezdetekor, így eszerint tudja a szoftver felépítését megtervezni. Az én esetemre igaz az is, hogy nem csak használati szempontból kell jónak lennie a szoftvernek, hanem kód szinten is. Mivel ez egy hosszabb életciklusú projekt, aminek vannak még nem ismert funkciói és részletei, illetve később más fejlesztők is valószínű, hogy csatlakozni fognak a fejlesztéshez, ezért a kódminőségre és a fejleszthetőségre az átlagosnál nagyobb figyelmet kell fordítanom.

2.1 Figyelembe vett tervezési szempontok

A cél előállítani egy grafikus felülettel rendelkező szoftvert, ami képes vizuálisan ábrázolni, elemezni és feldolgozni a statikus és dinamikus PET rekonstrukció során keletkező adatokat. Ehhez a programhoz meg kell terveznem egy intuitíven és jól használható felhasználói felületet, ami könnyen bővíthető újabb elemekkel. A felület legyen szétbontva átlátható méretű és bonyolultságú ablakokra, és ezek között legyen logikus az átjárás, adott esetben az interakció.

Az alkalmazás tudjon megnyitni statikus- és/vagy dinamikus rekonstrukciókat, a hozzá tartozó volumetrikus adatokat, illetve beolvasni az egyéb hozzájuk tartozó adatokat és paramétereket. Ezeket az adatokat az adott rekonstrukció vizsgálata során a memóriában kell tartani. Egyes volumetrikus adatok mérete és mennyisége túllépheti a memória méretét. Ahhoz, hogy a memóriába mindenképp elférjenek az adataink, egy gyorsítótár-rendszert kell kialakítani a nagyméretű volumetrikus adatok tárolására, és adott esetben csak akkor betölteni ezeket, amikor szükség van rájuk.

A fejlesztéshez egy meglévő, de még fejlesztés alatt álló SDK-t (Software Development Kit) kell használnom. Ebben a rekonstrukcióhoz kapcsolódó adatszerkezetek kódon belüli ábrázolásai, különböző ki- és bemeneti műveletei, illetve segédfüggvényei, fontosabb, gyakran használt algoritmusok vannak megvalósítva. Mivel az SDK még fejlesztés alatt áll, így a lehető legtöbb esetben le kell választani a program többi részéről. Ehhez alkalmazni kell a, szoftverfejlesztésben sok helyen alkalmazott, wrapper technikákat. Ennek segítségével változás esetén csak a wrapper class-t kell átírni, ezzel csökkentve a függőséget az SDK-tól.

A nézetek feladatai logikusan legyenek szétválasztva mind felhasználói, mind kódolási szempontból. A különböző funkciócsoportok legyenek jól elkülönítve, ne legyenek túl bonyolultak és túlzsúfoltak a felületek. Ennek ellenére, mivel az eszköz felhasználói erősen műszaki beállítottságúak és az eszköz is egy fejlesztés támogatását szolgálja, egy átlagos alkalmazáshoz képest bonyolultabb felület nem okoz problémát. Viszont a használhatóságot és a fejlesztést is megkönnyíti, hogy nem kell a bonyolultabb beállításokat vagy kapcsolókat az átlagos felhasználók elől elrejteni, mert minden, az eszközt használó felhasználó érti a legbonyolultabb beállításokat is.

A tervezés és implementáció során figyelembe kell venni, hogy a követelmények a program teljes életciklusa alatt változhatnak, és változni is fognak. A szoftvert úgy kell elkészíteni, hogy könnyen bővíthető és módosítható legyen. Ehhez törekedni kell arra, hogy a különböző nézetek egymástól függetlenek legyenek, és a kód minél modulárisabb legyen, minél kevesebb függőség legyen a különböző nem-kapcsolódó részek között. A moduláris kialakításhoz az MVVM (Model-View-Viewmodel) architektúrát vettem alapul. Ezt az implementációnál részletezem. A teljes szoftver úgy legyen kialakítva, hogy megkövetelje minden hozzáadott elemtől az eddig kialakított struktúra megtartását, de ez legyen úgy megtervezve, hogy segítse és ne akadályozza az új elemek hozzáadását.

2.2 Rekonstrukció felépítése

A rekonstrukció egy mérésnek a kimenete, amin a programnak dolgoznia kell. Egy rekonstrukció adatait egy fájlkönyvtár tartalmazza, aminek a neve a rekonstrukció neve is. A rekonstrukcióban használt összes adat itt található meg. Vannak olyan fájlok, amik minden rekonstrukcióban megtalálhatóak, rekonstrukciótípus függőek, illetve teljesen opcionálisok is, amikre csak akkor van szükség, ha ahhoz kapcsolódó adatot

szeretnénk kiértékelni. Ezenkívül megkülönböztethetjük a volumetrikus adatokat tároló fájlokat, illetve a konfigurációs fájlokat.

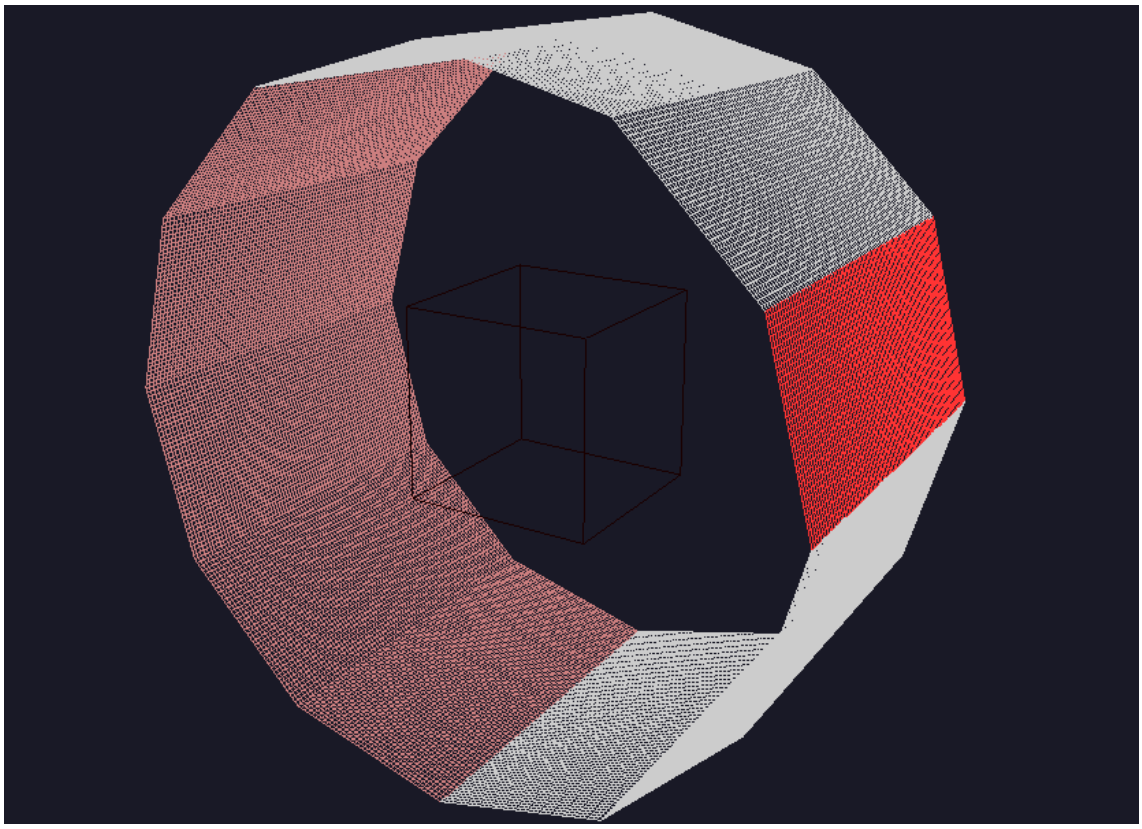
Minden rekonstrukcióban megtalálhatóak bizonyos konfigurációs fájlok. Ezek a rekonstrukció során felhasznált folyamatok beállításait és kimeneteit tárolják. Egy kiemelt fájl a *recon.std* nevű fájl. Ez reprezentálja magát a rekonstrukciót, illetve a rekonstrukció megnyitásakor a program is az ilyen kiterjesztésű fájlokat listázza ki. Ez a fájl tartalmazza, hogy az adott rekonstrukció statikus vagy dinamikus, illetve a legfontosabb konfigurációs fájlok nevei is elérhetőek benne. A rekonstrukció tartalmaz még sok egyéb konfigurációs fájlt, amelyekből ki lehet olvasni a rekonstrukcióhoz használt fájlok elérési útvonalai, a mérési idő hosszát, vagy éppen a volumetrikus adatok fizikai leíróját.

A volumetrikus adatok két különböző fájltípusban találhatóak meg. Ezek vagy bináris *vox*, vagy szöveges *mvol* kiterjesztésűek. Minden volumetrikus adat tartalmazza a nevében a saját felbontását. Mindegyikhez tartozik egy leíró konfigurációs fájl is, ami szintén tartalmazza a felbontást, illetve a fizikai méreteket, és a mérés középpontjához mért relatív határait is az adatnak. Az adathoz tartozó leíró alapvetően a rekonstrukcióhoz tartozó volumetrikus adatleíró, azonban vannak olyan volumetrikus adatok, amikhez saját leíró tartozik. Ezenkívül egy volumetrikus adat háromféle kategóriába sorolható. Az első a szkennelés során keletkezett és kiírt adat. Ezek statikusnál rekonstrukcióknál iterációkhoz, dinamikusnál iterációhoz, és készítési időhöz vannak rendelve. A második az evaluált típusú adat, első típusnak egy már valamilyen módon kiértékelt változata. A harmadik kategóriába pedig az egyéb volumetrikus adatok tartoznak, amik az első két kategóriába nem sorolhatóak, és valami kiegészítő funkcióhoz vannak használva, például a régió térképhez tartozó adat ilyen.

A rekonstrukció által tartalmazott volumetrikus adatok voxelekből állnak, és ezeket az adatokat különböző metrikák szerint ki lehet értékelni. Ehhez a kiértékeléshez szükséges ismerni a volumetrikus adat fizikai méreteit, így a rekonstrukcióban ezeket szükséges tárolni. Ezenkívül van, amikor a volumetrikus adatnak csak egy bizonyos részét szeretnénk kiértékelni. Ehhez adnak segítséget a régiótérképek. A régiótérképek olyan volumetrikus adatok, amik egy adott voxelhez egy bizonyos indexű régiót rendelnek, így ezek a térfogatok külön vizsgálhatóak. Ezzel a mérési területnek különböző meghatározott területeire lehet koncentrálni, illetve ennek segítségével lehet különféle szűrési- és regularizációs technikákban a priori anatómiai információt, pl.

szövethatárokat figyelembe venni. Ez hasznos, ha például csak egy bizonyos szerven belüli aktívitás-t szükséges evaluálni. A régiótérkép egy opcionális adat, ami nem feltétlenül található meg minden rekonstrukcióban.

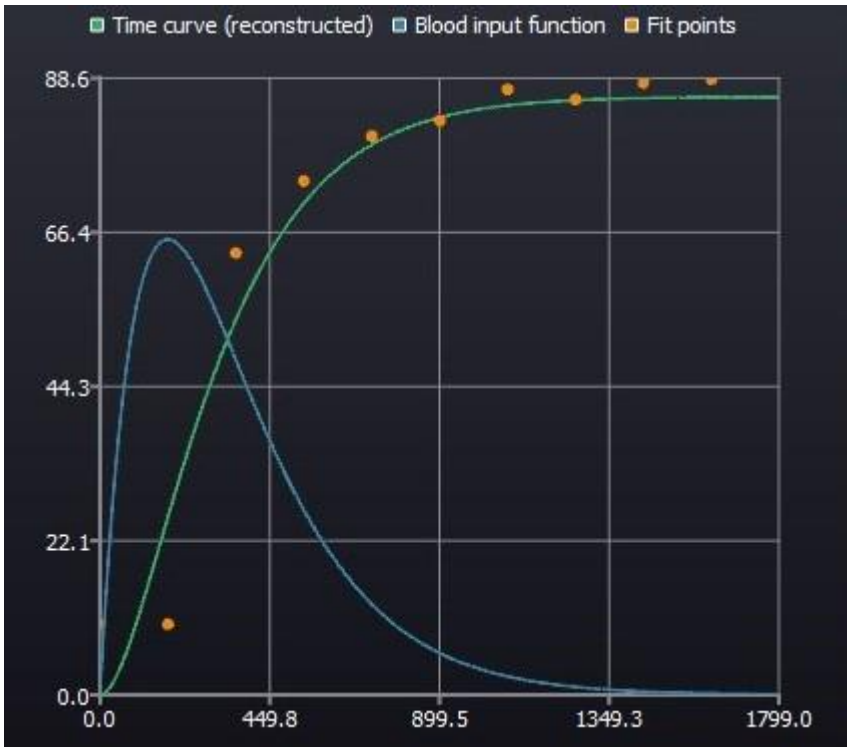
A PET szkenner fizikai adatai különböző számokkal leírhatók, amik később segíthetnek más adatokat kiértékelni, vagy a szkenner geometriáját kirajzolni. A *modulszám* írja le, hogy a PET szkenner fizikailag hány érzékelőkomponenseket, úgynevezett *detektorkristályokat* tartalmazó panelből áll. Ezek a vizsgált terület körül helyezkednek el egy *n* csúcsszámú szabályos sokszög formában, ahol az *n* a modulszámmal egyenlő. A sokszög sugarát a belső sugárérték írja le. A komponensek mennyiségét és elhelyezkedését az axiális és tangenciális felbontásokból lehet meghatározni. Ezek írják le, hogy a tengely és a tengelyre merőlegesen hány érzékelőkomponens található. Végül a koincidencia értéke azt határozza meg, hogy egy modul hány darab vele szemben lévő modulra „lát rá”. Ha például a koincidenciaszám 5, az azt jelenti, hogy 1 modul 5 másik szemben lévő modullal képes együttműködni a végső értékek meghatározásában. Erre az 1. ábrán látható példa.



1. ábra: Detektorgeometria és koincidencia szám ábrázolása, ahol a modulszám 12 és a koincidenciaszám pedig 5

A dinamikus rekonstrukciókban megtalálható néhány egyéb adat is. A *dynamic.cfg* tartalmazza az ehhez kapcsolódó paramétereket. Néhány fontosabb paraméter a mérési időegységek (*frames*), ezek felbontása (*ticks*), a véraktivitás-függvény paraméterei, és maga a véraktivitás-függvény is. Ebben a dinamikus rekonstrukcióban található minden iterációhoz és frame-hez egy volumetrikus adat, ami az adott időpillanatban leírja a mérési területen található aktívitásokat.

Az görbe illesztés a dinamikus rekonstrukciók kiértékeléséhez egy elengedhetetlen eszköz. Képes meghatározni egy vagy több voxel aktívitására, hogy ezek az idő múlásával hogyan módosultak, illetve ezekre a pontokra az idő és véraktivitás-függvény ismeretében egy folytonos görbét illeszteni, és ezzel a vizsgált időtartam alatt folytonossá tenni ezeket az adatokat. A véraktivitás-függvény értékeit, a hozzá kapcsolódó beállításokat, az egyes voxelekben illesztett kinetikus model paramétereket, és az illesztés bemeneteként szolgáló pontsorozatot a rekonstrukció kimentett adatai mind tartalmazzák. A kiértékelő programmal szemben elvárás, hogy képes legyen megjeleníteni a rekonstrukció által illesztett időgörbét, valamint akár utólag, változtatható kinetikus model és illesztési algoritmusok szerint is tudjon görbét illeszteni. (A görbe illesztésre egy példa a 2. ábrán látható.)



2. ábra: Egy, a programmal illesztett görbe. A kék diagram a véraktivitás-függvény értékeit ábrázolja, a zöld bedig a narancssárga pontokra illesztett görbét.

2.3 SDK

A SDK (Software Development Kit) tartalmazza a rekonstrukció során keletkezett adatok kódbeli reprezentációját, fontosabb adatszerkezeteket és algoritmusokat, amikre szükség lehet a rekonstrukció készítése, beolvasása vagy kiértékelése közben. Ebben az alfejezetben ismertetem az SDK általam használt adatszerkezeteit és algoritmusait. Sok kimentett adat esetében igaz, hogy van megvalósítása az SDK-ban, akár adatosztály, akár leíró konfigurációs fájl beolvasásának megvalósítása. A legtöbb SDK-ban található osztályt nem használom csak 1-2 helyen a kódban, inkább wrapper osztályokat használok, mivel így csökkenthető a függőség az SDK-tól.

A *Volume* osztály tartalmazza a volumetrikus adatokat. Az osztály kihasználja a C++-ban elérhető template rendszert, így lehetőség van bármilyen típusú volumetrikus adatot beolvasni. Ennek ellenére nincs olyan adat, ami nem float-ként van kimentve, illetve használva. A voxelek értékei sorfolytonosan vannak tárolva. Ehhez az osztályhoz szorosan kapcsolódik két másik osztály is: a *VolumeDescriptor* illetve a *VolumeLoader*. Az első volumetrikus adatot képes leírni, és ezt a leírót képes egy fájlútvonal alapján fájlból beolvasni. A *VolumeLoader* pedig szintén fájlútvonal alapján képes beolvasni vox vagy mvol kiterjesztésű volumetrikus fájlokat, aminek a kódba beolvasott eredménye egy *Volume*-ra mutató pointer lesz. A beolvasáshoz szükség van egy leíróra, mivel ez a leíró bekerül a *Volume* osztályba. A *Volume* ennek a leírónak a segítségével képes meghatározni, hogy a sorfolytonosan tárolt értékek a térben hol helyezkednek el, mi a háromdimenziós indexük. A *Volume* megvalósít rengeteg segédfüggvényt, ami megkönnyíti a programbeli használatát. A legfontosabb a különböző index- és háromdimenziós index alapú értéklekérések, illetve egy voxel pozíciójának globális koordináta-rendszerbe konvertálása. Ezenkívül rengeteg egyéb segédfüggvénye is van. Például egy függvény, ami megmondja a volumetrikus adat memóriában elfoglalt méretét. Ezt a volumetrikus adatok gyorsítótárának megvalósításához használtam. Ezenkívül még említésre méltó a *ParametricVolume*, ami olyan volumetrikus adatokat képes tárolni, ami voxelenként tartalmazza a véraktivitás-függvény adott voxelre megadott paramétereit.

A volumetrikus leíró osztályon kívül még rengeteg másik leíró megtalálható az SDK-ban, mindegyik ugyanazon az elven működik. A konfigurációs fájl útvonalát várja paraméterként, amit képes megnyitni, és betölteni fájlból. Ezután a leíró osztályt az azokat használó osztályok egyszerű paraméterként megkapják. Ilyen leíró osztályok például, a

rekonstrukció input és output fájljait vagy a detektorgeometriát leíró struktúra. Egy speciálisabb leíró a dinamikus rekonstrukció paramétereit tartalmazó osztály. Ez csak dinamikus rekonstrukció esetén használható és egy *dynamic.cfg* fájl a bemenete.

A véraktivítást leíró függvényt a *CpFunction* osztály reprezentálja, és a különböző kutatások során használt aktivitás függvények ebből származnak le. Az osztályban különböző lekérdező- és segédfüggvények vannak megvalósítva, illetve a függvény működésének szimulációját segítő függvények, például különböző optimalizációhoz, vagy közelítéshez használt függvények.

Az SDK-ban még az utólagos görbeillesztéshez fontos osztály a Kinetikus model. Ez parametrikus volumetrikus adatok voxeleire létezik, és az idő-görbe leíró, a véraktivitás-függvény és a tick szám alapján ki tudja számolni az adott voxel értékét az adott időpillanatban.

3 GUI framework választása

A tervezett szoftvernek egy hangsúlyos része a GUI (graphical user interface). Mivel az egyik elsődleges feladata a szoftvernek különböző adatok grafikus megjelenítése, ezért a tervezésnek egy fontos szempontja volt a GUI framework választása. A következő fejezetben a lefektetett követelmények alapján részletezni fogom a framework választás menetét. A követelményeken kívül két elsődlegesen figyelembe veendő szempont volt. Az első, hogy valamilyen formában tudnia kell használni az SDK-t. Az alkalmazásnak Windows platformon kell csak futnia, de ne legyen kizárva később egy Linuxos port se. Emiatt elfogadható megoldás, ha a SDK egy külső könyvtár formájában, dll-ként (vagy .so-ként) van használva, azonban a preferált megoldás, ha nem kell wrapper osztályokat írni az SDK-ban található osztályok köré. Emiatt a megfogalmazott követelmény az, hogy vagy könnyen használható legyen a dll-el, vagy direktbe bele lehessen fordítani az SDK-kódját. Másik fontos szempont, hogy az alkalmazásnak alapvető része az OpenGL-es megjelenítés. Ahhoz, hogy egy framework elfogadható legyen, meg kell felelnie annak, hogy egy vagy több OpenGL ablak direktben a GUI-ba építhető legyen, és az ablak és GUI között minimális mennyiségű kóddal mozgathatóak legyenek a megjelenítendő adatok.

Másodlagos, de fontos szempont volt a használhatóság és jó dokumentáció, a népszerűség, és hogy nekem is legyen valamennyi tapasztalatom az adott nyelvvel vagy framework-kel. A használhatóság és a jó dokumentáció szükségessége egyértelműen adódik. A népszerűség az elérhető külső, nem dokumentációs források mennyisége miatt lényeges. Ezek nélkül sokszor bármennyire is jó a nyelv, vagy a framework, nehéz dolgozni csak dokumentációból. Ezenkívül a különböző hibák és megoldásaik is sokkal nagyobb valószínűséggel megtalálhatóak, ha a használt technológia népszerű. A tapasztalat pedig a szoftver mérete miatt lényeges. A nagy kódbázis miatt fontos, hogy a használt framework-nek legalább az alapjai érthetőek legyenek a tervezés kezdetekor is, mivel ekkor akár csak egy kisebb rosszul használt komponens is kihathat az egész program szerkezetére vagy rosszabb esetben akár a működésére is. Bármennyire is jó a dokumentáció, egy ekkora program tervezésénél a tapasztalatot nem tudja pótolni.

Az általam részletesen megvizsgált framework-ök és technológiák a Java és Swing, Kotlin és TornadoFx [4], Python és Pyqt illetve a C++ és Qt framework [5].

Felmerülő, de elvetett framework volt még a C# és Windows Form Application. Ez a lehetséges jövőbeli Linux port miatt került elvetésre. Illetve két „következő generációs” UI toolkit-et vizsgáltam még meg, ez a Jetpack compose [6] és Flutter [7]. A Jetpack compose egy Kotlinos UI toolkit [8], aminek van asztali gépekhez készült változata is, de ezt úgy ítéltem meg, még nem ért el abba a stádiumba, hogy megbízható legyen vele a fejlesztés. Ugyanez a helyzet a Flutter framework asztali változatával is. A legtöbb követelményt lehetséges lenne megoldani vele, de mivel még béta stádiumban van, nagyobb lenne vele a program fejlesztési költsége, mint amennyit nyerni lehetne az előnyeiből (amik például: stateful hot reload, felhasználói interface fejlesztési sebesség).

A fent említett és megvizsgált framework-ök közül a Java-Swing és C++-Qt párosokra fogok részletesen kitérni, és leírni a döntési szempontokat, mivel a másik két framework-öt egyszerűbb szempontokkal sikerült eliminálni a lehetséges döntések közül. Ezeket kifejtem egy-egy bekezdésben, ezután a másik kettőt hosszabban részletezem.

Kotlin és TornadoFx: A Kotlin-t lehetséges használni JVM-mel (Java Virtual Machine), illetve létezik másik implementációja is, a Kotlin/native, ami native kódra fordul. Mivel a TornadoFx egy wrapper a JavaFx nevű Java-s GUI könyvtár körül, ezért ez a megközelítés nem működik native kóddal. Emiatt már sok előnyt elvesztünk, de mivel egy modern nyelv, fejlesztési idő szempontjából még mindig jobb lenne, mint a Java, ami szintén JVM-en fut. Két másik hátrány a kotlin/native elvesztése miatt az OpenGL használata, és az SDK dll-ként használt változatának használata. Természetesen mindkettő megoldható, de arra jutottam, hogy ez a bonyolultsága miatt túl sok hibalehetőséget tartalmaz, és továbbfejleszthetőség szempontjából sem ideális. Ezt megerősítette egy teszteléshez összerakott programom is: fejlesztés közbeni használhatósága sem a legideálisabb. Ezt a megközelítést emiatt elvetettem.

Python és Pyqt: A TornadoFx-es hátrányokhoz képest itt egyik sem jelenik meg. A Python egy script nyelv, ami miatt a komplexebb számításoknál lassabb lenne, mint a natív kód, de a legtöbb folyamatban nem lenne a felhasználó által érzékelhető különbség. A Pyqt a C++-ban írt Qt framework körül egy wrapper, így ez nem okozna lassulást. Illetve a Qt-nak az OpenGL widget-je és funkcionalitása is felhasználható, így ez sem okozna problémát. Rengeteg nagy Python-ban elérhető könyvtár sokszor csak egy wrapper a C++-os könyvtár kód körül, így a dll betöltése, és használata is gond nélkül menne. Ezenkívül a Python mellet szól még a fejlesztési sebesség is. A sok pozitívum mellett, ami mégis a Python ellen szól, az a program könnyű változtathatóságához

kapcsolódik. Mivel a típusok még Python 3-ban is csak az IDE-nek szóló hintek, ezért a refaktorálás sokszor komoly feladat rengeteg hiba lehetőséggel. Sok olyan hiba, ami egy erősen típusos nyelvben fordítási időben előjönne, Pythonban sokszor csak futás közben mutatkozik meg. Emiatt ezt a technológiát sem találtam alkalmasnak a program megírására.

3.1 Swing és Qt közötti döntési faktorok

3.1.1 Java és C++

Alapvetően a C++ és Java is jól skálázódik nagy projektekre. Mindkettő objektumorientált, és tartalmazza a projekthez szükséges eszközöket. Mindkettőt használják „enterprise” projektekben, amik sok százszor komplexebbek, mint ez az alkalmazás. Emiatt a skálázódás nem döntő faktor. A java egy valamivel újabb nyelv, és rendelkezik garbage collection-el. Ez a C++-szal szemben minimális előnyt jelent átláthatóság szempontjából, de valamilyen szinten ez megoldható a smart pointerek használatával, amik a garbage collection-höz hasonlóan referencia számlálással működnek, és képesek felszabadítani a saját maguk által foglalt memóriát, ha az már nincsen használva. Nekem C++-szal minimálisan több tapasztalatom van, de egy szubjektívebb tény, ami a Java mellett szólna, hogy ez egy valamivel népszerűbb nyelv, főleg az egyetemi környezetben, ahonnan később új fejlesztőket lehetne bevonni, ezért valamilyen szinten ezt is figyelembe kell venni a projekt bővíthetőségét illetően.

3.1.2 Együttműködés az SDK-val

Mivel az SDK C++-ban van írva, így itt elsősorban a dll Java-val használhatóságára fogok koncentrálni. A Java a native kódhoz kétféle hozzáférési módot biztosít, a Java native access-t (JNA), és Java native interface-t (JNI). A kettő között a fő különbség az, hogy melyik kódbázisba kerül az interface a másikhoz. A JNA-nál a java kódba kerülnek a kompatibilitáshoz szükséges eszközök, még a JNI-nél a C++-kódból lehet elérni a Java objektumokat. Mindkettőnek vannak előnyei, de mivel a JNI-nél szükséges kód változásakor módosítani a C++, és Java kódon is, ezért a JNA-ra koncentráltam jobban. Több különböző tesztet írtam egy teszt „SDK”-val és teszt JAVA programmal. Mivel a program és az SDK között nagyméretű tömbök és struktúrák átadását meg kell tudni valósítani, ezt is részletesen teszteltem. A tesztjeim végére arra jutottam, hogy teljesen megoldható, de nem praktikus. Ahhoz, hogy az SDK-val lehessen

dolgozni, egy törékeny kompatibilitási réteget fent kellene tartani. Ezenkívül a memória terület felszabadításának felelőssége sem egyértelmű. Az SDK és a Java közötti adat átadása native pointerek átadásával történik. Emiatt nem annak kellene felszabadítani a használt memóriát, aki lefoglalta. Ez a program méretének növekedésével sokat növel a komplexitáson, ráadásul a garbage collection előnyét is elveszti a java. Ehhez képest C++-ban még dll sem, kell, hanem direktben bele lehet fordítani az SDK kódját a programba, így a fenti problémák egyike sem jelenik meg.

3.1.3 Build rendszer

A build rendszer a fordításnál és függőségek kezelésénél fontos. A Java-hoz egy modern és népszerű build-rendszer a Gradle. Ennek használatával nagyjából bármilyen Java-s package elérhető az internetről, és ezeknek a kezelését automatizálja a Gradle. Ez C++ és Qt-nál másfajta elképzelés alapján működik. A Qt-nak van egy saját build-rendszere a qmake, ami a cmake-hez hasonlóan egy fájlban felsorolja a különböző fájlokat, amiket a fordítónak ismernie kell, illetve tartalmazza a Qt specifikus dolgokat is, például modulok, verzió stb. Ezután egy kompatibilis fordító az egészet lefordítja. Ezenkívül cmake-el is együtt tud működni a Qt. Erre később egy követelmény változás miatt szükség lesz, ezért ezt lejjebb részletezem. Itt a kettő közül én a Gradle-t preferáltam, mivel modernebb és kényelmesebb a használata, de ez főleg a nyelv jellegéből adódik, mintsem a build-rendszerből. Ezenkívül a Qt-ban nagyjából minden nekem szükséges elem elérhető modul formájában, így egy modul hozzáadása ugyanannyi időt vesz igénybe, mint Gradle-ben egy package hozzáadása.

3.1.4 Dokumentáció és támogatottság

A Swing és Qt is egy széleskörben használt eszköz. Ezt megemlítve a Qt egy teljes C++-on alapuló framework rengeteg saját megvalósított segéd osztállyal, amik nem feltétlenül csak a megjelenítéshez kapcsolódnak. A Swing-el ellentétben a Qt-t ma is aktívan fejlesztik, és bővül új feature-ökkel, amik naprakészen tartják. Emiatt egy lényegesen összeszedettebb és átláthatóbb dokumentációval rendelkezik, ami meggyorsítja a fejlesztést.

3.1.5 Konklúzió

Összefoglalva, a Swing egyetlen tényleges előnye a Java nyelv Garbage collection-je lenne, de a Qt-val és a modern C++-ban használt smart pointer-ekkel

minimálisan több odafigyeléssel ezt az előnyt sem tartottam mérvadónak. Ezen indokok miatt esett a végleges döntés a Qt framework-re.

3.2 Qt framework

A QT egy több millió fejlesztő által használt és szeretett cross-platform GUI framework [9]. Jelenleg a „The Qt Company” [10] fejleszti, de nyílt forráskódú, és elérhető hozzá nyílt forráskódú licensz, illetve cégeknek kereskedelmi licensz is. C++ -on alapul, emiatt a legtöbb hardware platformon natívan működik, de akár beágyazott rendszereken futtatáshoz is biztosít eszközöket. C++-ban írt programokon kívül lehet használni a framework-öt Pythonban a saját deklaratív nyelvükkel, a Qt QML-lel, vagy akár Javascript-tel is [9]. Saját IDE is tartozik hozzá, a Qt creator, amibe a különböző Qt specifikus funciók mélyen integrálva vannak. A C++ -os változatát lehetséges teljesen ebből a fejlesztői környezetből használni, de a Qt biztosít eszközöket, hogy az extra funkciók, például a meta-object rendszer lefordítható legyen a legtöbb népszerű C++ fordító, és Cmake segítségével is

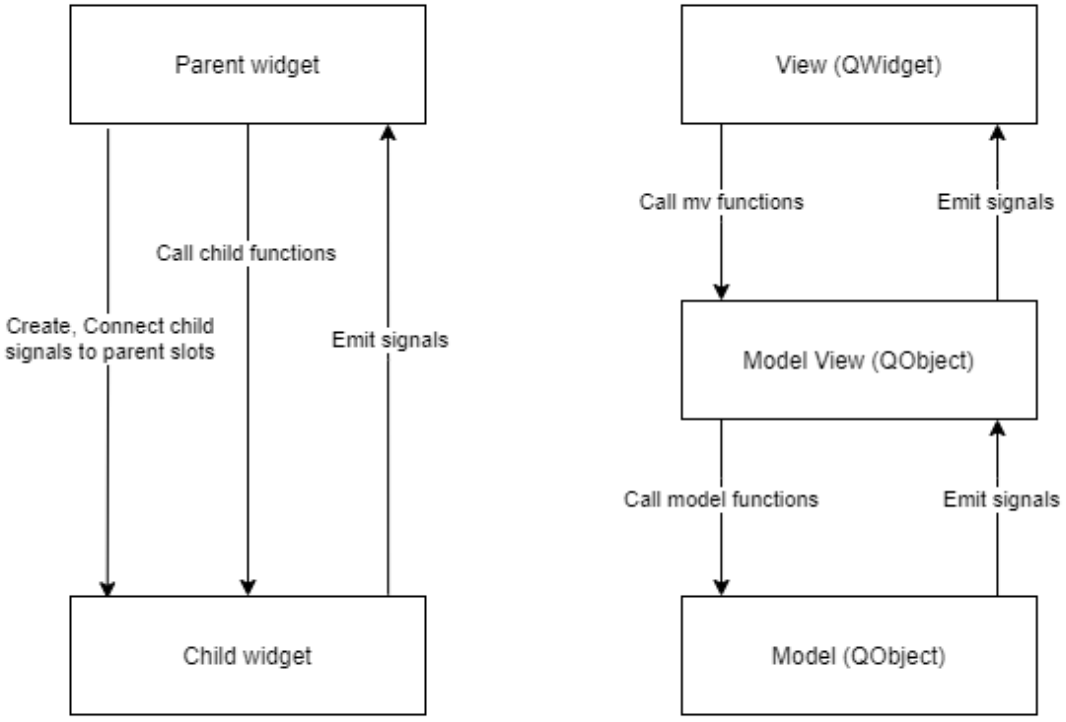
Az első publikus kiadása 1995-ben jelent meg [11]. Azóta több száz widget került be a framework-be, és a hatodik major release-t adták ki. Kezdetben Haavard Nord és Eirik Chambe-Eng által lett kifejlesztve, de ma már több mint 300 ember dolgozik a cégnél.

3.2.1 Signal és slot rendszer

A Qt egyik legnagyobb találmánya és erőssége a signal-slot rendszer [12], ami talán legjobban megkülönbözteti a többi GUI framework-től. Ennek a rendszernek a lényege, hogy a komponensek egymással tudjanak kommunikálni úgy, hogy nem is feltétlenül kell ismerniük egymást, vagy kapcsolódniuk egymáshoz, ezzel biztosítja az összes komponens szeparáltságát. Ezt sok nagy framework callback-ekkel oldja meg. Viszont a signal-okon alapuló megközelítésnek megvan az az előnye, hogy típus biztos, így még fordítási időben kiderülnek az összekötési problémák, az átláthatóságról nem is beszélve. Működése a Qt-s meta-object rendszernek köszönhető, ami megjelöli a fordítónak az így deklarált osztályokat, ami ezekhez hozzágenerálja a szükséges extra kódot, így ezt nem nekünk kell kézzel megírni. Amit ebből mi látunk használat közben, hogy bármelyik osztály, ami a *QObject* osztályból származik le, és el van látva a Q_OBJECT nevű makróval, használhatja ezt a signal-slot rendszert.

Egy objektum kibocsát egy signal-t, ha egy esemény történik, például egy gombra kattintanak, vagy programkódból is bármikor kibocsátható. Az objektumnak az életciklusa során fel lehet iratkozni a signal-jára a connect függvény használatával, és hozzá kötni azt egy slot-hoz. Egy slot egy tagfüggvénye egy adott objektumnak, ami meg van jelölve a fordító számára, hogy ehhez lehet signalt kötni. Ezt leszámítva egy teljesen szokásos tagfüggvény. (Ma már csak átláthatóság kedvéért jelölik meg, hogy egy tagfüggvény slot, enélkül is engedi használni a connect-et a Qt.) Az egyetlen megkötés, hogy a signal és slot paraméterei meg kell, hogy egyezzenek. Ezenkívül a signal és a slot semmit nem tud egymásról, azt se, hogy hány darabhoz van kötve, vagy hogy akár használva van-e. Az sem probléma, ha egy signal úgy emit-álódik, hogy nincs hozzá kötve slot, ilyenkor nem történik semmi.

Mivel a Qt-ban majdnem minden beépített komponens signal-okon keresztül értesít az eseményekről, és a saját komponenseknél is arra bátorít, hogy ezt a rendszert használjuk, ezért érdemes néhány alapszabályt lefektetni azzal kapcsolatban, hogy ezek az összeköttetések mi szerint történjenek a kód egyszerűségének megőrzésének érdekében. Én ezt két különböző esetre szedtem szét: a GUI komponensek hierarchiájára, és a (view)model és a GUI közötti kommunikációjának esetére. Az első esetben azt a Qt által is javasolt megoldást követtem, hogy csak az a nézet komponens csatlakoztathat signal-okat slot-okhoz, ami létrehozta a másik komponenst. Ez praktikus, mivel a létrehozott nézet teljesen független komponens lesz a létrehozótól, és nem függ tőle semmilyen módon. Ezzel a módszerrel a GUI modularitása is megőrizhető. A másik eset a model és viewmodel, illetve a viewmodel és view közötti kapcsolatok. Ilyen esetekben pedig csak szigorúan a viewmodel iratkozhat fel a model signal-jaira, és a view a viewmodel-ére. A visszafelé kommunikáció szigorúan függvényhívásokkal történik. Így megőrizhető a komponensek közötti hierarchia és a „mélyebb” komponenseknek nem kell ismernie a „magasabban” lévőket, ez a model leválasztását, és a tesztelhetőséget is segíti.



3. ábra: Kommunikáció szülő és gyerek widget között (bal) és az MVVM komponensek között (jobb).

3.2.2 Qt könyvtárak

A Qt a különböző elérhető funkciókat és komponenseket könyvtárakba szervezi. Ezeknek egy része települ a Qt-val, és alapértelmezetten használható, ilyen például a *core*, vagy *gui* könyvtár. Illetve lehetőség van Qt-s könyvtárakat telepíteni vagy saját könyvárat hozzáadni. A következő néhány fejezetben az adatvizualizációhoz, illetve a GUI reszponzivitásához és szálkezeléshez kapcsolódó Qt könyvtárakat részletezem.

3.2.2.1 Adatvizualizáció

Az egyszerű szám és szöveg adatokat kivéve két féle adattípust lehet megkülönböztetni a programban. Az egyik különböző diagram típusokon megjeleníthető adat, a másik a teljesen egyedi, kézzel megírandó két- vagy háromdimenzióban reprezentálható adatok megjelenítése.

A diagramokhoz a Qt egy külön telepíthető könyvtárát, a QChart-ot [13] használtam. A könyvtár egy átfogó megoldást biztosít a különböző diagramok megjelenítésére, a Qt Graphics Framework-öt használja, ami miatt könnyen használható az összes Qt-ban megtalálható megjelenítési kontextusban. Erősen és könnyen testre

szabható a könyvtár által biztosított diagram, és a legtöbb általános diagram beállítás megoldható legtöbbször probléma nélkül. A könyvtár biztosít widgetet a diagramok megjelenítésére, illetve a különböző diagramtípusokhoz külön adattípusokat is, amiket a diagram widget képes megjeleníteni. Az általam használt diagramtípusok konténereinek használata mind megegyezik. Tartalmaznak minden pontra egy x és y tengelyhez tartozó értéket. A widget felveszi a különböző pontokat, majd az adattípustól függően megjeleníti azokat, például görbe megjelenítés esetén összeköti ezeket a pontokat.

Az érdekesebb probléma a két- és háromdimenziós adatok megjelenítése volt. Mindezt Qt-s widget-ként, már létező Qt-s GUI-ban. Az egyediségük, és komplex megjeleníthetőségük miatt ezeket OpenGL segítségével célszerű megjeleníteni, de ezt az OpenGL -el renderelt ablakot meg is kell jeleníteni a GUI-n. Szerencsére a Qt erre biztosít eszközöket, van egy külön widget beépítve a *QOpenGLWidget* [14], ami egyben egy OpenGL canvas is. Ez előnyös, mivel ez egy teljesértékű OpenGL ablak, így a különböző event-ek, és egyéb specifikus funkciók is elérhetőek, de a Qt specifikus widget tulajdonságok is megmaradnak. Emellé saját OpenGL kompatibilitási rétege van, amivel a különböző fordítási konfigurációkhoz elkerülhető az újrafordítás, és különböző platformokon különböző konfigurációk használata. Ez előnyös, ha megvalósul a program lehetséges Linux-os változata is a jövőben. A különböző külső könyvtárak kettő- és háromdimenziós vektorjai implementációjának elfedésére pedig a *gui* könyvtárba épített *QVector2D* [15] és *QVector3D* [16] osztályokat használtam. Ez a két vektorosztály rengeteg kényelmi vektor funkciót implementál a különböző matematikai, és megjelenítési műveletekhez, illetve a programon belül nem kell a más forrásokból érkező implementációk kompatibilitásával foglalkozni, ezzel is növelve a program jövőbeli módosíthatóságát.

3.2.2.2 Threading a Qt-ban

Egy GUI könyvtárnak, és általánosságban is egy programnak fontos része, hogy képes legyen kezelni a hosszú futás idejű műveleteket anélkül, hogy blokkolná a GUI-t. Ezt ahogy a legtöbb hasonló framework, a Qt is többszálúsítással oldja meg. A koncepció lényege, hogy a fő szál megmarad a GUI-nak, például hogy kijelezzen egy töltő sávot, és egy másik szálon pedig fut a sokáig tartó művelet. A szálkezeléshez a Qt több eszközt is biztosít. Én ebből kettőt használtam, a *QThread*-et [17], és a Qt concurrent-et [18]. Elsősorban nem a párhuzamosításra, hanem a nem fő szálon futtatásra térek ki.

A *QThread* a core könytárban található. Ez egy alacsonyabb szintű, de nagyobb kontrollt biztosító megoldás. Ezt akkor érdemes használni, ha futás közben kommunikálni kell a futó szállal. Ilyen kommunikáció alatt például a folyamat százalékban mért állapotának kijelzését értem. Én ezt vagy ilyen esetben, vagy akkor használtam, amikor a szálon egy komplexebb folyamat futott, mivel így egy sokkal elszeparáltabb az adott kódrészlet.

A Qt concurrent könyvtár egy magasabb szintű API-t biztosít a szálkezelésre. Automatikusan elosztja a feladatot a szálak között, és sokkal átláthatóbbá teszi a kódot. Ezenkívül egy modern megközelítést, a future-t használja a művelet állapotának kommunikációjára. A future egy olyan objektum, ami a jövőben tartalmazni fogja a művelet eredményét, de a létrehozásának pillanatában még nem. Ahhoz, hogy a future-ös megoldás jobban illeszkedjen a Qt-s szemléletben, egy segédosztály is létre lett hozva, a *QFutureWatcher*, ami segít a future-nek az állapotváltozásait lekövetni, például kibocsátani egy signal-t, ha a műveletnek vége lett. Ezt akkor találtam hasznosnak, amikor rövid, de hosszan futó kódnál meg kellett szüntetni, hogy blokkolja a fő szálat. Ez egyszerűen elérhető volt a következő módon: a blokkoló kódot beletettem egy Qt concurrent által biztosított utasításba, ami visszaadott egy future-t. Ezt a future -t hozzáadtam egy future watcher-hez, és ennek a finished signal-ját pedig hozzákötöttem egy olyan függvényhez, ami lereagálta az eredmény elkészültét.

Természetesen ezeken kívül több, másik szálkezelési megoldás létezik még akár C++-ban, akár Qt-ban is, de az egyszerűségük és Qt-val szoros integráltságul miatt a fentebb említetteket használtam. Mindkettő előnye, hogy a beépítettség miatt csökken a függőségek száma, és kevesebb kóddal megoldható a másik szálon futtatás.

4 Implementáció

Ebben a fejezetben funkciókra vagy nézetekre bontva fejtem ki a megvalósítás lépéseit. Ismertetem az adott funkciónak a feladatát, a hozzá kapcsolódó nézetet, mögöttes elméletet, ahol ez lényeges, illetve magát a megvalósítást. Az implementáció leírását próbálom programozási nyelvtől függetlenül megírni, így sokszor fogok specifikus funkciókat, vagy konténereket általánosítani. Amellett, hogy a Qt a C++ legtöbb alapkönyvtárában megtalálható konténerhez saját megvalósítást is mellékel, ami általában az alap megvalósításra épít, extra funkciókat is ad hozzá, ilyen például a *QVector*-ban [19] az elemtartalmazás vizsgálat. Ezenkívül a modern C++ előnyeit kihasználva az összes nem Qt-által kezelt pointernél smart pointert használtam, ami lényegesen leegyszerűsítette a memóriakezelését. A továbbiakban a *QVector*, illeve az STL *vector* osztályokra csak tömbként, a smart pointerre pedig csak pointerként fogok hivatkozni.

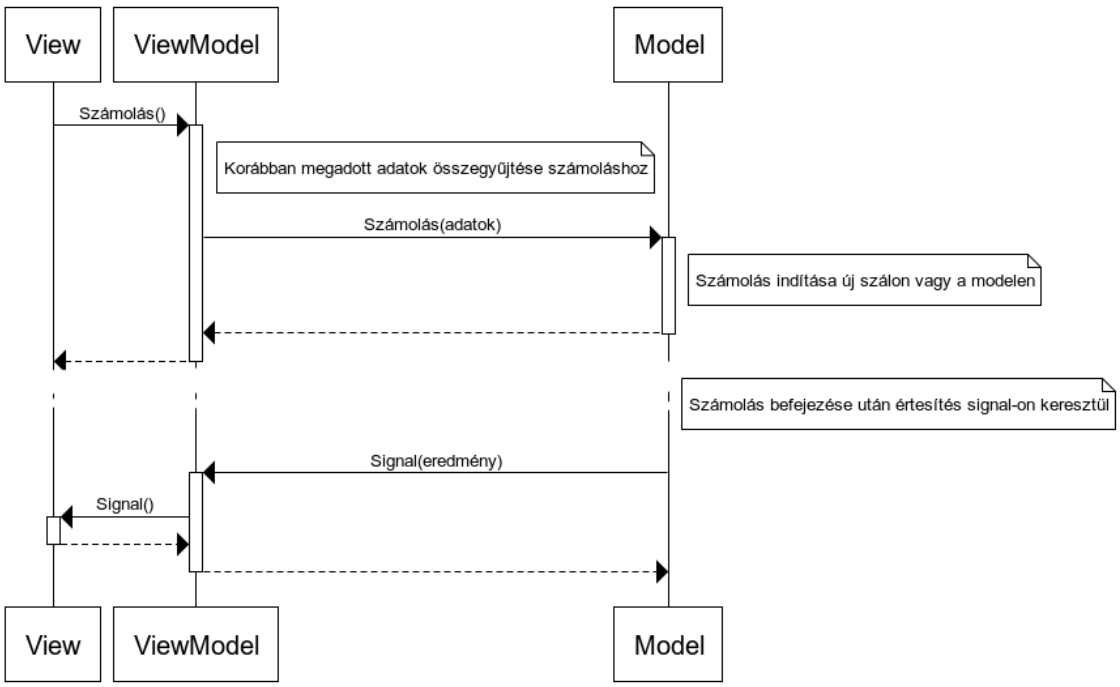
4.1 Szoftver architektúra

Az architektúra kitalálása egy fontos lépése volt a megvalósításnak, mivel elég nagy részben ezen áll vagy bukik a későbbi bővíthetőség. A program GUI orientáltsága miatt az MVC architektúrából indultam ki. Ezzel viszont a szoftver összetettsége miatt a controller osztály mérete lett volna a probléma. Igaz, hogy szét vannak bontva a komponensek, de a model és nézet közötti kommunikációhoz szükséges mennyiségű függvénnyel kezelhetetlen méretűvé nőtt volna az osztály. Emiatt egy elvben hasonló, de ezt kiküszöbölő architektúrát, az MVVM architektúrát választottam.

A „Model-View-ViewModel” architektúrának hasonló alapelve van, mint az MVC-nek. Ez elsősorban a mobil platformokon elterjedt tervezési minta, de az elvei ugyanúgy alkalmazhatóak bármilyen GUI-val rendelkező szoftverfejlesztési projektben. Az MVC-hez hasonlóan az a feladata, hogy a model-t és a nézetet szétválassza, majd ezek egy harmadik osztályon keresztül kommunikáljanak, vagyis ne ismerjék direktben egymást. Tehát a viewmodel-nek hasonló feladata van, mint a controller-nek. A különbség, hogy minden nézethez tartozik egy viewmodel, nem pedig alkalmazás szinten létezik egy singleton controller osztály. Ez több szempontból is előnyös. Egyrészt megmarad, a model-nézet szétválasztás, és mivel a viewmodel-nek legtöbb esetben nem szükséges ismerni a model osztály eredetét, ezért az interface-t leszámítva ettől is

függetlenedik. Másrészt a viewmodel csak azokat az osztályokat és programrészeket ismeri, amik szükségesek az adott feladat megjelenítéséhez, így függetlenebbek egymástól a különböző nézetek, és a modularitásuk miatt könnyebben módosíthatóak.

Qt-ban az MVVM-mel tovább növelhető a rétegek függetlenítése signal-ok és slot-ok használatával. Ahogy a 3.2.1-es fejezetben már szó esett róla, ezek használatával elkerülhető, hogy mélyebb rétegnek ismernie kelljen egy magasabb réteget. A programomban a mélyebb rétegnek szigorúan signal-okon keresztül kell kommunikálni. Ez előnyös, mivel így a magasabb réteg csak azokra a signal-okra iratkozik fel, amire neki szüksége van, így rengeteg boilerplate-kód megspórolható. A magasabb réteg, mivel már eleve ismeri a mélyebbet, függvényhívásokkal kommunikál a mélyebb felé. Ezek a legtöbb esetben állapotváltoztató függvények, aminek az eredménye signal-okon keresztül érkezik vissza. Például egy hosszú műveletet elindítunk egy gombra kattintva. Ilyenkor a nézet meghív egy számolás függvényt a modelview-n, ami összeszedi a korábban már beállított adatokat, és elindítja a háttérben a model osztályon vagy szálon a számolást. Ez signal-lal jelez amikor az adott számolás kész. Ilyenkor a viewmodel beállítja az eredmény alapján saját magán a különböző adatokat, és signal-lal jelez, a nézetnek, hogy megjeleníthető az eredmény. Ezt a folyamatot a 4. ábrán szemléltetem.



4. ábra: A rétegek közötti kommunikáció egyszerű példán keresztüli szemléltetése.

Az eredmény kétféleképpen tud visszakerülni a hívó rétegbe. Ez nálam elsősorban az eredményhez szükséges adatok típusától és mennyiségétől függött. Amikor egy vagy

maximum kettő paraméterben átadható az eredmény, a szebb megoldás, hogy az eredmény elkészültét jelző signal paramétereként adja át az eredmény(eke)t. Viszont van, hogy ezt nem lehetséges paraméterekként visszaadni, és minden signal eredményéhez egy osztályt vagy struktúrát készíteni pedig túlbonyolítaná a kódot. Ilyenkor egy üres signal értesíti a magasabb réteget, hogy megvan az eredmény, aminek a változóit lekérdező függvények segítségével szerzi meg a hívó réteg. Ezek a lekérdező függvények még inicializáláskor előnyösek, például amikor létrejön egy új nézet, és a model jelenlegi állapotát ki kell jelezni. Ezt persze meg lehetne oldani a nézetbe beégetett alapértékekkel is, viszont ez nagyon könnyen inkonzisztenciát tudna okozni, amennyiben egy alapértelmezett érték megváltozik.

Az MVVM architektúra az SDK-tól való függetlenítésben is rengeteget segít. Mivel a viewmodel már alapvetően is elrejti a model-t a nézet elől, így az SDK leválasztása sem volt nehéz. Mivel a GUI-n általában csak primitíveket kell és lehet megjeleníteni, ritkán van csak szükség arra, hogy a nézet bármilyen model-ben található adatszerkezetet ismerjen. Amennyiben mégis primitívtől eltérő adatszerkezet kell, megpróbálom azt egy Qt-ban található beépített osztállyal helyettesíteni. Erre jó példa a *QVector3D*, amivel a model különféle vektorok osztályait fedtem el. A legvégső esetben, amikor nem helyettesíthető az adatszerkezet más módon, és SDK-ban található adatot kell elvezetni a GUI-ig, írtam egy wrapper osztályt, mivel így csak a wrapper osztályban kell lekezelni az SDK változását.

A gyakorlatban nem mindig ennyire egyszerű a helyzet. A legtöbbször nem csak egy model osztály tartozik egy modelview-hoz, viszont az architektúra ezt segít elrejteni a nézet elől. Ennek segítségével megoldható, hogy a modelből a nézet csak a megjelenítéshez kapcsolódó részeket lássa. A modelview emiatt sok esetben egy gyűjtőosztálya volt a különböző mélyebb model-osztályoknak, ami összefogta ezeket, felhasználó által megadott adatokat gyűjtött nekik, lekezelte a közöttük lévő összeköttetéseket és hívta a különböző állapot változtató függvényeiket.

Ezek miatt lett az alapértelmezett architektúrája a programomnak az MVVM, amitől csak ritka esetben tértem el, amikor ez az architektúra nem lett volna lehetséges, vagy rontott volna a kódminőségen. Ennek az eredménye, hogy a szoftver nézetenként, és model-nézet-modelnézet szerint is szét van szeparálva, így a függőségek minimalizálásával mindegyik komponens bármelyik másik módisítására zárt.

4.2 Rekonstrukció betöltésének menete

A legtöbb fejezet az implementációban egy specifikus nézethez kapcsolódik. Mivel egy rekonstrukció és volumetrikus adatok betöltésére a teljes programon belül szükség van, ezért ezek külön fejezetben kaptak helyet. Egy rekonstrukció több, akár ismert, akár nem ismert mennyiségű fájlból áll. Ezért ennek betöltése egy komplex feladat, és ez több lépésre bontható.

Az első lépés kiválasztani a betöltendő rekonstrukciót a merevlemezről. Ehhez felhasználói bemenetre van szükség. A betöltéshez szükség van elsősorban a betöltendő rekonstrukció elérési útvonalára. Ezt valójában egy string-ként is be lehetne kérni a felhasználótól, de a Qt a beépített fájlválasztó dialógusával ezt sokkal könnyebb, és felhasználóbarátabb megoldani. Ez a dialógus az operációs rendszeren megszokott interface-el biztosítja a navigációt a merevlemezen található könyvtárak között. Az aszinkronitásával sem szükséges foglalkozni, mivel ez a dialógus a használt modális módban megakasztja a fő szál futását, de maga a fájlválasztó használható marad. A dialógus paraméternek várja a kiinduló útvonalat, amit alapértelmezetten szeretnénk mutatni a felhasználónak, illetve a megnyitandó fájl kiterjesztését. Ez valójában nem a konkrét kiterjesztést ellenőrzi, hanem azokat a fájlokat listázza ki, amik megfelelnek az adott mintának. Minden rekonstrukcióban megtalálható fájl, a rekonstrukció gyökerében található *recon.std*, amiből meg lehet nyitni a teljes rekonstrukciót. Emiatt a keresett minta a „*.std” lett. Az ehhez a fájlhoz tartozó útvonal segítségével már megnyitható a rekonstrukció.

A recon fájl tartalmazza a rekonstrukcióhoz tartozó legkritikusabb fájlútvonalakat, illetve azt, hogy az adott rekonstrukció statikus, vagy dinamikus-e. Mivel a két esetben különböző féleképpen kell betölteni a rekonstukciót, a következő lépésekhez szükséges ismerni ezt az információt. Ezen információk ismeretében elkezdődik a rekonstrukció tényleges betöltése. Ez a *FileManager* osztály feladata. Ez egy statikus osztály, amin keresztül a fájlműveletek történnek a programban. Az egyetlen kivétel ezek alól a volumetrikus fájlok betöltése, de erre kitérek a következő fejezetben. A fájl manager első lépésként beolvassa az *std* fájlt a megadott útvonalról. Ehhez a Qt-nak a beépített *QSettings* osztályát hívom segítségül. Ez képes szöveges fájlokat megadott formátum alapján beolvasni, és kiolvasni a megadott formátum alapján a kulcsérték párokat. Ezután ezeket az értékeket a kulcsuk alapján a map-hez hasonló módon lehet elérni. Miután megtörtént a szükséges értékek és útvonalak beolvasása, megtörténik

rekonstrukció model osztályának, a *StudyModel*-nek a létrehozása. Ez paraméterül az előbb beolvasott adatokat várja, és az alapján, hogy statikus vagy dinamikus, eltárolja a különböző értékeket és útvonalakat későbbi használatra. A model osztály létrehozása után következik a volumetrikus adatok betöltése. A program megkeresi az összes volumetrikus típusú adatot a rekonstrukció könyvtárában, létrehoz hozzá egy *VoxData* osztályt a hozzátartozó útvonallal a cache alapú betöltéshez (következő fejezetben kifejtem), és kategória alapján eltárolja a model osztályban. A volumetrikus adatoknak háromféle kategóriája lehet. Az első a különböző iterációkhoz tartozó, a második az evaluált, a harmadik pedig az egyéb rekonstrukcióhoz tartozó volumetrikus adatok. Vannak, amik egyik kategóriába sem sorolhatóak, például a rekonstrukcióhoz tartozó különböző régiótérképek. Amennyiben a fentebb írt lépések mindegyike teljesült, a betöltött rekonstrukció készen áll a használatra.

4.3 Volumetrikus adat betöltése és volumetrikus cache

A volumetrikus adat betöltése és a cache szorosan összekapcsolódik, ezért ezt egy fejezet alá vettem. Mivel a volumetrikus adatok mérete a néhány megabyte-tól több gigabyte-ig terjedhet, fel kellett rá készülnöm, hogy egy több 10, vagy akár száz darab volumetrikus adatot tartalmazó rekonstrukció nem biztos, hogy el fog férni a számítógép memóriájában. Ez bizonyos mértékig az operációs rendszer is meg tudná automatikusan valósítani swap fájlok segítségével, de nagy mennyiségű adattal ez egy lassú, és nehezen kontrollálható megoldás lenne. Viszont erre a legtöbb esetben nem is lenne szükség, mivel csak 1-2 volumetrikus adathoz nyúl hozzá a program rendszereresen, a többihez meg elég lenne csak on-demand hozzáférni. Emiatt lett kialakítva a cache, ami a legutóbb megnyitott vagy használt volume-okat tárolja.

A programban a volume-ok a *VoxData* osztályon keresztül vannak használva. Ez azért lett létrehozva, hogy egyrészt elrejtse az SDK-ban található volumetrikus osztályt különböző lekérdező függvények segítségével, másrészt, hogy kényelmes legyen használni cache-t. Egy volumetrikus adat betöltése a következő lépésekből áll. Amikor a programban betöltünk fájlból egy volumetrikus fájlt, valójában csak egy *VoxData* osztály példány jön létre. Ez az osztály a konstruktorába két dolgot vár, a hozzá tartozó fájl elérési útvonalát, illetve a volumetrikus adatleíró osztályt. Ez a két adat szükséges ahhoz, hogy bármikor be lehessen tölteni a merevlemezről az adott fájlt, ha szüksége van rá. Ezenkívül plusz előny, hogy ha a *VoxData* ismeri a leírót, sokkal több információt le lehet kérni az

adatról a betöltése nélkül. Azzal, hogy ez a két információ el van tárolva, elértük, hogy nem szükséges egyből betölteni a fájlt, és az elérési útvonalból egyértelműen azonosítani tudunk egy betöltött adatot. Minderre azért van szükség, mivel a cache a programban úgy működik, hogy a VoxData osztály nem is tárolja a volume-ot.

Az összes volume-ot a programban, a *VolumeCache* tárolja. A *VolumeCache* egy Singleton osztály (C++-os megvalósításban statikus), ami pointereket tárol a betöltött volume-okra. A cache-t az LRU (Least Recently Used – Legrégebben nem használt) elv alapján valósítottam meg, mivel ez egy számunkra elég optimális, amellett, hogy az egyik legkönnyebben megvalósítható is. Ehhez két tárolóra volt szükségem, egy map-re és egyre, ami sorokat képes kezelni. Ezek a Qt-ban megtalálható *QMap,* és *QQueue* lettek. A *QMap* a fájlnév stringhez rendeli hozzá a volume-ra mutató pointert. A queue pedig a fájlneveket foglalja sorredbe. A cache foglaltságát az általa használt memóriaterületből lehet kiszámolni. A maximális cache által használt memóriát egy konstans érték határozza meg.

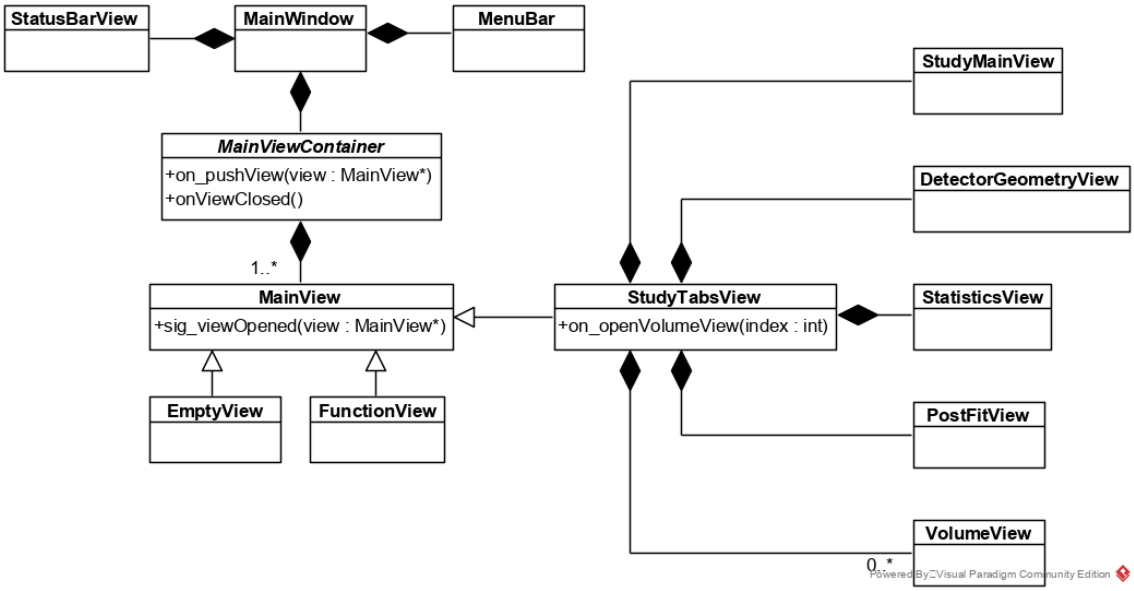
A volumetrikus adathoz a *VoxData* osztály *getVolume* függvényén keresztül lehet hozzáférni. Ebből a függvényből kívülről annyi látszik, hogy visszaad egy pointert egy volume példányra. Ezt a megvalósításban viszont a cache-ből kéri le. Amennyiben a cache tartalmazza az adott útvonalú volume-ot, visszaadja az arra mutató pointert, és a sorrendet számontartó queue-ba áthelyezi, hogy a legutolsó elem legyen. Ha a cache nem tartalmazza az adott elemet, valamivel bonyolultabb a helyzet. Először is meg kell határozni az adat pontos méretét. Ezután, ha elfér még a memóriában, hozzáadjuk a maphez, és a queue végére. Amennyiben nem, addig kell törölni a queue alapján a volume-okat tartalmazó map-ből, ameddig megint elérjük a maximális megengedett cache méretet (az új elemmel együtt). Ezután az új elemet megint csak hozzáadjuk a maphez, és a queue végére.

A fájlból betöltést már az SDK megvalósítja, mivel a *Volume* egy már ebben található osztály. A betöltéshez az SDK biztosítja a betöltő osztályt, aminek szüksége van arra, hogy a betöltött volume milyen primitíveket tartalmaz (ez általában float, de int vagy double is lehetne), a betöltött fájl útvonalára, és leírójára. Ez meg van valósítva az összes SDK-ban található volumetrikus fájlformátumra, így az adott függvény kiválasztásával bármelyik típus betölthető, és egységesen használható.

4.4 Nézetek felépítése

Ebben és a következő néhány fejezetben a nézeteken keresztül mutatom be a program implementációját, mivel így lehet a legátfogóbb képet kapni a megvalósított funkciókról. Ebben a fejezetben általánosságban mutatom be a GUI hierarchiáját, és kitérek azokra az osztályokra és nézetekre, amik nem elég komplexek ahhoz, hogy külön fejezetbe kerüljenek, de azért szükséges őket megemlíteni a program működése szempontjából.

A nézetek közötti hierarchia reprezentálására készítettem egy egyszerűsített osztálydiagramot (5. ábra), amin a fontosabb osztályok és függvények szerepelnek. Ez ábrázolja az általam kifejtett nézet osztályok hierarchiáját. A jobb oldalt látható *StudyTabsView* által tartalmazott osztályokat, és hozzájuk tartozó funkciókat külön fejezetekben részletesen kifejtem, a többihez pedig egy összefoglalót írok az implementációjukról.



5. ábra: A nézetek közötti hierarchia ábrázolása

Mint minden Qt program, ez is egy *QApplication* példány, és a megjelenítő ablak létrehozásával és a Qt eventloop-jának elindításával kezdődik a main-ben. A programban egy kényelmi funkció az alapértelmezett sötét mód. Ez főleg ergonómiai szempontokból előnyös: kevésbé fárasztja a szemet, de ezenkívül még pozitívum, hogy egy egységes kinézetet is kölcsönöz az alkalmazáson belüli nézet elemeknek, ami javítja a felhasználói élményt. Ehhez egy Github-on elérhető témát, a QDarkStyle-et [20] használtam fel, amin

módosítottam a programhoz mért igények szerint. Ezzel a Windows xp-hez illeszkedő Qt-s alapértelmezett stílust le tudtam cserélni egy sokkal modernebb kinézetűre.

A programon belül a gyökér megjelenítő elem a *MainWindow* osztály. Ez tartalmaz minden más widget-et a program futása során (a dialógusokat kivéve). Ennek az osztálynak a feladata lekezelni a minden GUI-val rendelkező programban magától értetődő dolgokat, például a különböző ablak funkciókat (méretezés, program bezárása, stb.). Ezenkívül fontos feladata, hogy kezelje és tovább irányítsa a különböző extra kezelői felületekről érkező események signal-jait, például menüsávban egy gombra kattintás, vagy a statusbar-on a különböző üzenetek vagy töltési állapot kijelzése. Ez az osztály három másik widget-et tartalmaz. A menü- és státuszsáv, a két egyértelműbb, ami a legtöbb asztali alkalmazásban megtalálható. A menüsávban a programban globálisan elérhető funkciók, például egy rekonstrukció betöltése érhetőek el. A státuszsávban pedig a program ad vissza információt az aktuális állapotról, például rekonstrukció betöltésének a százalékát. A harmadik egy saját widget, a *MainViewContainer*, ami a modularitás miatt került a programba. Ez foglalja el a teljes tartalom részét az ablaknak.

A *MainViewContainer* több szempontból növeli a modularitást, és ezzel a program későbbi továbbfejlesztését. Ennek a párja a *MainView* absztrakt widget osztály. A *MainViewContainer* osztály *MainView* leszármazott widget-eket tartalmaz, és ezeket stack alapon jeleníti meg. A konténer widget-nek a feladata megnyitni, megjeleníteni és bezárni a megfelelő leszármazott osztályokat, illetve ezek az osztályok a konténer widget-en keresztül kommunikálnak a program többi része felé. Maga a konténer nem tartalmaz sok megjelenítést az éppen megnyitott ablakot bezáró gomb kivételével, a tényleges tartalom a leszármazott osztályokban található. A stack alapú megjelenítés hasznos, ha a konténer tartalmát meg kell változtatni anélkül, hogy az éppen megnyitott widget elveszne. Ehhez egy Qt-s beépített widget-et használok fel, amibe el lehet helyezni más widget-et, és ezek között index-szel váltani, hogy melyik legyen az aktuálisan megjelenített widget. Az alapértelmezett működést felülírva a megjelenítési stack legelső eleme mindig az *EmptyView*, amit nem lehet kitörölni, vagy bezárni.

Ez egy olyan widget, ami csak néhány gombot tartalmaz a programban elérhető különböző nézetek megnyitására. Jelenleg a programban két ilyen nézet van, egy függvényparaméter tesztelő, és a rekonstrukció megjelenítő. A függvényparaméter tesztelő nem tartalmaz említésre méltó új elemet, de a rekonstrukció megjelenítőt később több fejezetben is részletesen kifejtem. Ezekre a gombokra kattintva, a signal eljut a

konténerben az adott nézetet megnyitó slot-hoz, és annak megfelelően megnyitja azt. Ez bővíthetőségi szempontból előnyös, mivel egyrészt új *MainView* leszármazott osztály hozzáadásakor a meglévő kód módosítása nélkül csak hozzá kell adni egy új gombot az *EmptyView* layout-jához, a megfelelő signal-okat bekötni, és legfeljebb a létrehozó függvényt kell megírni a konténerosztályban. Másrészt emiatt a leszármazott hierarchia miatt gyorsan lehet új változtatásokat implementálni az összes *MainView* osztályba, és ezt csak egy helyen kell megírni, ami gyorsabb fejlesztést, illetve átláthatóbb kódot is eredményez hosszú távon is. Ezenkívül új *MainView* hozzáadása esetén sem kell megírni újra a specifikus kódokat, elég egy leszármaztatást használni.

A konténerhez megjelenített elem hozzáadása és kitörlése is egyszerű ezzel a felépítéssel. Új elem hozzáadásakor egyszerűen csak létre kell hozni az új widget-et, hozzáadni a stack-hez, és beállítani, hogy mostantól ezt jelenítse meg. A teljes folyamat a következőképpen néz ki: Egy slot-on keresztül a konténer érzékeli, hogy új elemet akarunk hozzáadni, minden *MainView*-nak saját slot-ja van. Ezután a konténer létrehozza a megfelelő nézetet. Innentől viszont kihasználható, hogy minden létrehozott nézetnek közös őse van. Ebben a lépésben kerül a stack tetejére a létrehozott nézet. Ehhez hozzá kell adni a stack-et, és be kell állítani a helyes indexet. Ezenkívül, mivel csak a megjelenített nézet signal-jait szükséges fogadni, a stack többi nézetének signal-jainak fogadását kikapcsoljuk, és csak ennek az egynek a signal-jaira iratkozunk fel. Elem törlésekor kitöröljük a stack-ből az adott nézetet, és ilyenkor automatikusan az alatta lévő widget fog újra látszódni. Ebben annyira kell figyelni, hogy mivel eltávolítottuk a Qt-s widget hierarchiából a widget-et, innentől a mi felelősségünk felszabadítani az általa használt memóriát.

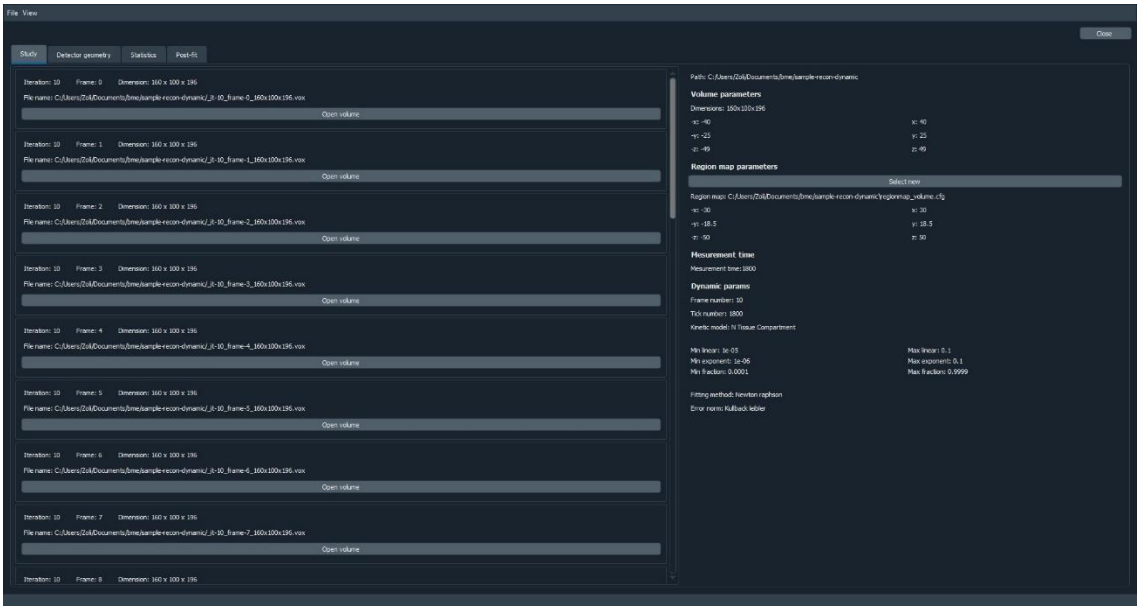
A harmadik *MainView* osztály, amiről eddig még nem esett szó a *StudyTabsView*. A három közül ez a legfontosabb és legkomplexebb is. A rekonstrukció megnyitásakor ez a widget jön létre és kerül a stack tetejére (ez látható megvalósítva a 6. ábrán). Ez tartalmazza és hozza létre az összes rekonstrukcióhoz kapcsolódó widget-et, ahogy látható a fenti diagramon (5. ábra) is, illetve ez az osztály kapja meg a model osztály magját képező *StudyModel* osztályra a pointert, amibe a FileManager már betöltötte az adatokat, ezzel az átadással is még jobban leválasztva ezt a rekonstrukciós részt a program többi részéről, amennyiben a jövőben ez változna. Ennek a nézetnek a tervezésekor jött el a pillanat, hogy ki kellett találni, hogy a különböző nagyjából független alnézeteit a rekonstrukciónak hogyan lehetne úgy elhelyezni a nézetben, hogy nagyjából téma szerint

el legyenek választva, de azért érződjön, hogy az összes egy rekonstrukcióhoz tartozik, és mindezek mellett könnyen kezelhető is maradjon. Az első demo megvalósítás erre egy a *MainViewContainer*-hez hasonlító megoldás volt, hogy különböző gombokra kattintva a hozzá tartozó rekonstrukció nézet lesz látható, és ennek bezárásával lehet visszanavigálni a gombokat tartalmazó nézetre. Ez teljesítette, hogy téma szerint el legyenek szeparálva a nézetek. Viszont megvalósítás szempontjából is körülményes volt, mert a korábbi hasonló kódot nem lehetett újra felhasználni ennek a megvalósításra, másrészt felhasználói élmény szempontjából is elég rossz volt, nehezen volt áltátható, és sokat kellett kattintani a funkciók megtalálásához. Ezt a demo megvalósítást a végleges programban egy asztali alkalmazásokban sokszor használt, tab (fül) alapú megvalósításra cseréltem, ami például böngészőkben, vagy szöveges fájlszerkesztőkben van gyakran van használva. Ehhez szintén biztosít beépített megoldást a Qt, amiben megtalálható nagyjából minden ilyen GUI-kban elterjedt, és felhasználók által ismert funkció. Ennek az előző, stack alapú megvalósításhoz képest rengeteg előnye van. A felhasználónak is átláthatóbb és könnyebben használható, mert egy lényegesen megszokottabb környezetben találja magát. Nem szükséges egy külön nézet csak arra, hogy navigáljunk, hanem bármelyik rekonstrukciós nézetről bármelyikre át lehet ugrani egy kattintással. Ezzel nem csak rengeteg kód spórolható meg, mivel ezek a funkciók a Qt-ban implementálva vannak, de az előbb felsorolt követelményeknek is jobban eleget tesznek, ráadásul a kiépítés jellege miatt sokkal dinamikusabb is, így nem csak jobban fejleszthető, de bármikor meg lehet nyitni és bezárni azokat az oldalakat, amikre nincs szükség. Emiatt végül a különböző volumetrikus fájlokhoz tartozó megjelenítőt is ebbe a tab rendszerbe integráltam, a korábbi leválasztott megvalósítás helyett, így ez is jobban a rekonstrukcióhoz tartozónak érződik. Ezen látszódott, hogy a jövőben más nézeteket is könnyű lesz hozzáadni a felépítés miatt, amit ez a nézet megkövetel.

4.5 Áttekintő nézet

Az áttekintő nézet a megnyitott rekonstrukció „kezdő oldala”. Egy új rekonstrukció megnyitásakor ez a nézet fogad minket. A nézet (6.ábra) megjeleníti a rekonstrukció fontosabb adatait, beállításait, és tartalmazza a rekonstrukcióhoz tartozó volumetrikus fájlok listáját, és ezek fontosabb adatait. Egy volumetrikus adat listaelemre kattintva megnyílik az adott adathoz tartozó szelet megjelenítő, így elsősorban ez a nézet, a navigáció, és áttekinthetőség szempontjából fontos. Emiatt sem szoftverfejlesztési, sem

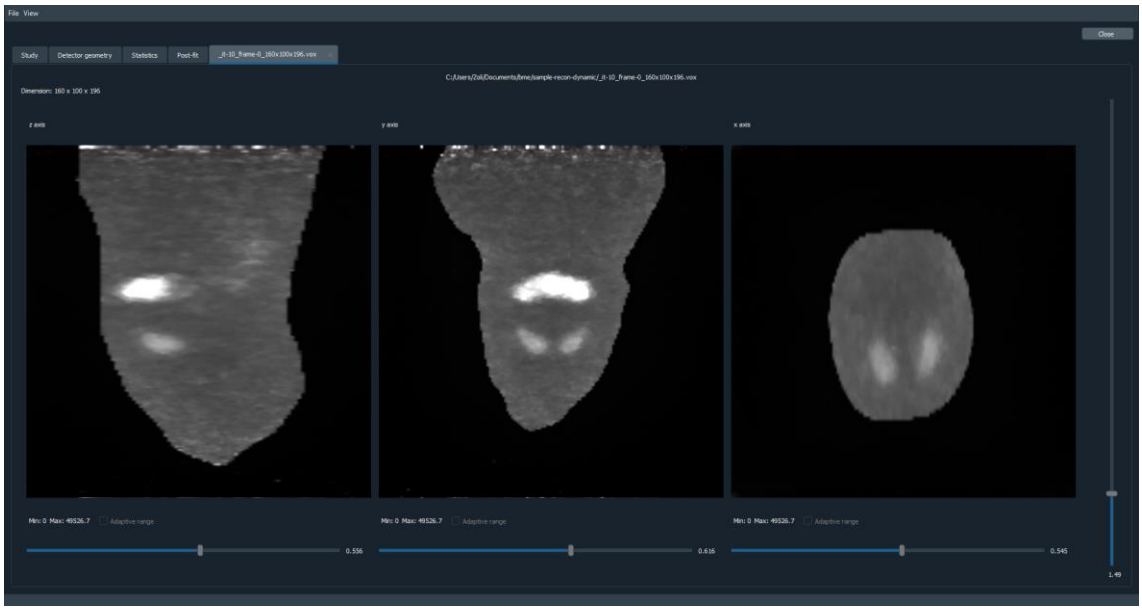
PET rekonstrukciós szempontból nem tartalmaz kifejezetten érdekes témát, így ezt a nézetet nem is fejtem ki részletesebben, csak a teljesség kedvéért lett megemlítve.



6. ábra: A rekonstrukció betöltése után látható kezdőoldal, az áttekintő nézet

4.6 Szelet megjelenítés

PET szkenneres kiértékelések ellenőrzésekor egy hasznos funkció, ha magának a szkennelésnek is meg tudjuk nézni az eredményét. Mivel egy háromdimenziós adatot kell kétdimenziós képen megjeleníteni, így ez a probléma nem teljesen magától értetődő. Ennek egy lehetséges megoldása, hogy a háromdimenziós adatból szeleteket készítünk, vagyis egy bizonyos mélységben lévő metszetét jelenítjük meg a volumetrikus adatnak, amit így már meg lehet jeleníteni kétdimenzióban. Ahhoz, hogy átfogó képet kapjunk a szeletekről, mindhárom tengely mélységében meg kell valósítani ezt a metszet megjelenítést. Emiatt a nézetben (7. ábra) három szeletmegjelenítő widget található. Azt, hogy az adott tengelyen éppen milyen mélységű szelet van megjelenítve, egy 0 és 1 érték közötti csúszkával állítható. Ebből az arányból a volumetrikus adat teljes mélységéhez képest kiszámítható az éppen megtekinteni kívánt szelet pontos mélysége. Hasznos funkció a szelet méretének megnövelése is, ha csak bizonyos részletek érdekesek a szeleten. Erre a funkcióra egy külön csúszka biztosít lehetőséget, amivel a szeleteket tartalmazó widget-ek fizikai méretét lehet változtatni.



7. ábra: A szelet megjelenítő nézetben láthatóak a megnyitott volumetrikus adat különböző irányú szeletei.

A szelet megjelenítőt a Qt-ba beépített OpenGL-widget segítségével írtam meg. A Qt tartalmaz egy beépített OpenGL wrapper-t is, ami megkönnyíti a használatát ezzel a widget-el. Ezt leszámítva a widget, és a wrapper is hasonlóan működik, mint a sima OpenGL, vagy a megszokott különböző nyelvekben megtalálható implementációk, annyi különbséggel, hogy több esetben egy objektumorientáltabb megközelítést követ, mint az egyszerű OpenGL. Az OpenGL-widget-nek másik előnye, hogy azon kívül, hogy egy OpenGL canvas, egyben egy Qt widget is, ami képes lereagálni a különböző méretezési, és bemeneti eseményeket. Ez például a később kifejtett görbeillesztés voxelének a kijelöléséhez, vagy a detektorgeometria irányításhoz lesz felhasználva.

A megjelenítés geometria része egy egyszerű négyzet sarkainak koordinátáinak a létrehozásából, majd annak a VBO-ba (Vertex Buffer Object) való feltöltéséből áll, amit végül az OpenGL a „triangle fan” kirajzoló algoritmus segítségével a képernyőre rajzol. Mivel ezt lehetséges úgy megtenni, hogy a kirajzolt lap négy sarka pont a képernyő négy sarkába kerüljön, és nem szükséges ezt a térben mozgatni vagy a kamerát forgatni, vagy mozgatni, így ezzel nincs további szükséges teendő. A textúra koordináták generálásához az inicializációs függvényben szükséges megadni a szükséges flag-eket.

A volumetrikus adatok egy sorfolytonosan tárolt float tömbben találhatóak. Szerencsére az OpenGL tartalmaz hasonló adatok tárolására beépített adatstruktúrát, a háromdimenziós textúrát. Ez több szempontból is kifejezetten előnyös, mivel egyrészt van egyszerű módja az adatok GPU-ra (Graphics Processing Unit) való feltöltésének,

másrészt a háromdimenziós textúrából a GPU lényegesen nagyobb sebességgel képes kiolvasni egy bizonyos réteget, mintha ezt a tömböt CPU-n (Central Processing Unit) kéne előállítani, majd GPU-ra kétdimenziós textúraként feltölteni. Ezenkívül az egyes voxelek értékein elvégzett számítási műveletek is a CPU-n végzett idő töredékébe kerülnek. Azonban a háromdimenziós textúrát úgy ítélte meg a Qt csapata, hogy ez nem egy annyira gyakran használt feature, ezért ennek az elérésére szükség volt az inicializáló függvényben létrehozni egy olyan osztályt, amin keresztül ez a funkció elérhető. Azonban létrehozás után teljesen hibátlanul működött maga a feature, és az összes hozzá tartozó funkció. A volumetrikus adatok GPU-ra való feltöltéséhez szükséges a megfelelő OpenGL függvény meghívása, a megfelelő paraméterek megadásával. Ezek a paraméterek többek között a textúradimenzió, a feltöltendő adat típusa, mérete és magára az adatokat tartalmazó tömbre a pointer.

A textúra, és geometria feltöltése után következik a kirajzolás fázis. Ehhez két shadert hoztam létre. A Qt-s wrapper a GLSL-t (OpenGL Shading Language) csak a 410-es verziótól támogatja, így én is ezeket a verziókat használtam. A vertex shader egy nagyon egyszerű pár soros GLSL kód, ami a vertex pozíciókat, és a textúra koordinátákat továbbadja a fragment shader-nek. Ebből a folyamatból a legérdekesebb a fragment shader működése.

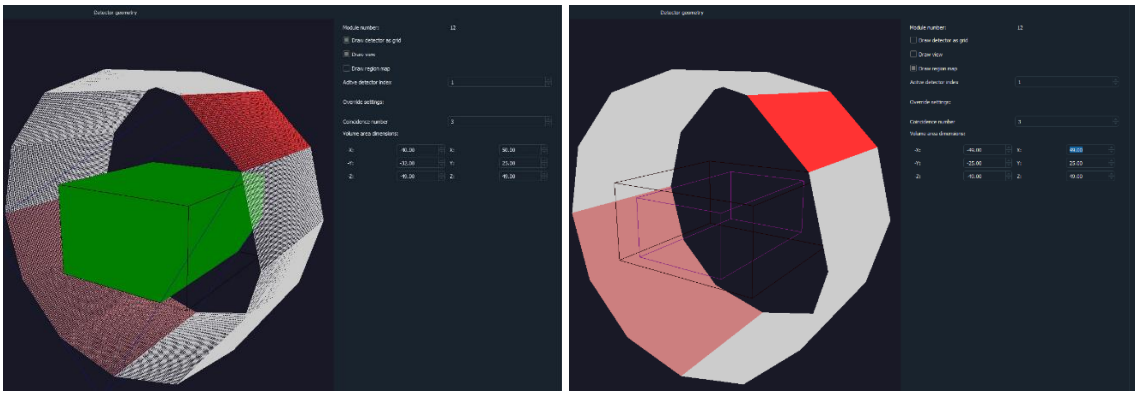
A fragment shader bemenetként megkapja a vertex shader-től a kirajzolandó lap pontjait és textúra koordinátáit. Ezenkívül még kirajzolás előtt beállított uniform értékek a háromdimenziós textúra (sampler3D), a volumetrikus tömbben található minimum és maximum érték, a jelenleg kiválasztott szelet arány, illetve, egy változó, amivel jelezni tudjuk, hogy melyik tengely mentén történik a szelet megjelenítése. A kapott textúra koordinátákból, illetve a jelenlegi szeletből, előállítható egy háromdimenziós vektor, aminek segítségével a sampler-ből már könnyedén kiolvasható az adott helyen található érték. A következő probléma ennek az értéknek a 0 és 1 közé alakítása, mivel az OpenGL csak ebben a tartományban képes színeket megfelelően kirajzolni. Ehhez van szükség a korábban már említett, a rekonstrukció betöltésekor kiszámolt minimum és maximum értékre. Ennek a kettőnek az ismeretében egy egyszerű képlettel az összes értéket a 0 és 1 közötti tartományba mozgathatjuk, és a két szám között pedig lineárisan interpolálhatjuk a többi értékre. Ehhez a következő képletet használtam, ahol v a kapott érték, és v_{min} és v_{max} a minimum és maximum, a x_i pedig az interpolált eredmény: $x_i = \frac{(v - v_{min})}{(v_{max} - v_{min})}$. Összeségében ehhez a nézethez sikerült minden számolást vagy úgy

megoldani, hogy csak a rekonstrukció betöltésekor egyszer kelljen kiszámolni, vagy pedig a GPU-n történjen ezeknek a számolása, így egy kivételesen jól optimalizált, és az egyszerűsége miatt könnyen karbantartható kódot kaptam eredményül.

4.7 Detektorgeometria kirajzolása

A rekonstrukció eredményének vizsgálatakor gyakran van szüksége a PET szkenner fizikai adatainak ismeretére. Ez minden esetben tárolva van a rekonstrukcióban, viszont nem a legfelhasználóbarátabban vannak tárolva ezek a paraméterek, sokszor csak nehezen értelmezhető számok vannak kimentve, amiből csak sok számolással lehet emberileg is érthető információt szerezni. Ennél egy sokkal szebb megoldás, hogy ezeket a számolásokat elvégeztetjük a számítógéppel, és az eredményt kirajzoljuk a képernyőre. Emiatt készült ez a nézet a rekonstrukcióhoz.

Ez egy olyan funkció, ahol a nézet és a model osztályok is kifejezetten érdekesek, mivel egy elég komplex funkciót fednek le, ezért az adatok kiszámolását és megjelenítését is részletesen kifejtem. A probléma több részre és megjelenített elemre bontható, ahol mindegyiket különböző módon kell kiértékelni. Az első része a problémának magának a PET szkenner detektor geometriának a kirajzolása. Itt lehet változtatni, hogy a modult csak egyszerű lapként, vagy a komponenseit is szemléltető rácsként kell kirajzolni. Ezenkívül ehhez hozzá tartozik az aktív vizsgált terület, illetve a régiótérkép által elfoglalt tér jelzése. A másik része a problémának a koincidenciához kapcsolódik. Itt egyrészt ki kell rajzolni, hogy egy kiválasztott modul melyik többi modulra lát rá, és ezt valamilyen módon jelezni, hogy ezek összetartoznak. Másrészt ebben a részben a legkomplexebb feladat, hogy ki legyen rajzolva, adott koincidencia számmal egy modul a szemben lévő modulokkal mekkora részt lát, illetve hagy ki az aktív térfogatból. Ez az aktív térfogat egyenlő a volumetrikus adatok fizikai pozíciójával, és ebből a leíróból is lesz kiszámolva a pontos helyzete. Ezenkívül egy kevésbé komplex, de hasznos funkció, hogy ezeket mind lehet ki- és bekapcsolni (8. ábrán látható ennek a hatása), így megjeleníteni, vagy elrejteni a kirajzolón.



8. ábra: A detektor geometria megjelenítő nézet. Bal oldalt a rács nézet és a látott térfogat, jobb oldalt a régió map kirajzolása van bekapcsolva.

A nézetet (8. ábra) két részre osztottam, az egyszerű ki- és bemeneti widget-ekre, amiken be lehet állítani és ellenőrizni a különböző paramétereket, illetve magára a kirajzoló widget-re, ami a detektor geometriát jeleníti meg háromdimenzióban. Korábban többször írtam, hogy a Qt biztosít olyan widget-et, ami képes megjeleníteni OpenGL canvas-t, amit hasonlóan lehet használni, mint az OpenGL különálló változatát, illetve a kamera mozgatásához szükséges egér eseményeket is képes elkapni. Ehhez a kirajzoló widget-hez is ezt használtam. Model téren is több osztályba osztottam szét a feladatokat, hogy ezek ne függjenek egymástól. Ez abból a szempontból is fontos volt, hogy a szkenner által látott térfogat kiszámolása hosszú időbe telik, ami hosszú ideg blokkolná a UI szálat, így többszálasítást is be kellett vezetni ennek számításakor.

A detektorgeometria kirajzolásának első lépése a modulok helyzetének kiszámolása. Ehhez szükség van az ezt leíró konfigurációs fájl betöltésére és feldolgozására. A nézeten lehet állítani, hogy látszódjanak-e a komponensek rács mintaként kirajzolva. Ez implementációs szinten úgy néz ki, hogy mindenképpen ki lesznek számolva a komponensek koordinátái, és csak az OpenGL-ben állítható kirajzolási mód fog különbözni. Rács kirajzolásához line típusú kirajzolást, tömör lap kirajzolásához pedig quads típusú kirajzolást alkalmazok. Ezzel a megvalósítással persze a GPU-nak sokszor lényegesen több geometriát ki kell rajzolnia, mint feltétlenül szükséges. Ez rácskirajzolási módban az átlapolódó vonalakban, tömör kirajzolási módban pedig a külön négyzetekből kirajzolt modul lap jelenti ezt az extra kirajzolt geometriát az egyben kirajzolt nagy lap helyett. Viszont ez még mindig elhanyagolható mennyiség egy modern grafikus kártya képességeihez képest, ezért sokkal előnyösebb az egyszer, és átláthatóan megírt geometria generáló kód. Mivel a koordináták meghatározása sok számítást igényel, ezért ez egy külön osztályt kapott. Ez ebben az

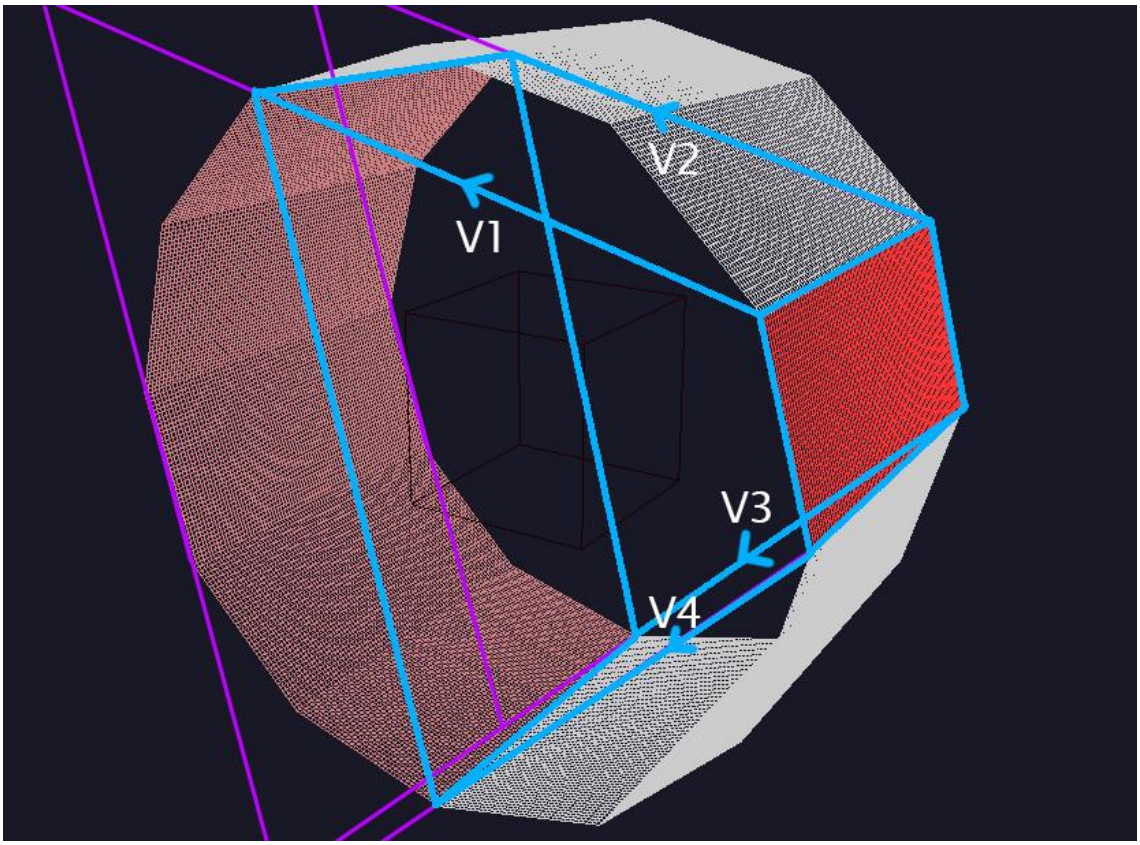
esetben főleg az átláthatóság szempontjából előnyös, mivel az nem valószínű, hogy változni fog a jövőben. Ennek a másik előnye, hogy könnyű egy a többi komponens elől elrejtett cache rendszert kialakítani, mivel életciklusa során ez a geometria soha nem fog változni, emiatt előnyös, hogy csak egyszer kell kiszámolni. A geometria kiszámolásához ez az osztály megkapja a korábban beolvasott paramétereket, majd ebből számolja ki a szükséges köztes változókat, például a modulok kezdő pozícióit, szögeit, komponens méreteit stb., majd ebből kiszámolja a komponensek végleges pozícióit, amit végül egy modulonként szétválasztott struktúrában tárol el a későbbi kirajzoláshoz. A megfelelő widget ezt kirajzoláskor lekéri, is a megadott kirajzolási formában felkerül a GPU-ra. Egy később részletezett, koincidenciához kapcsolódó funkció miatt a kirajzolás is modulonként történik, hogy specifikus moduloknak lehessen módosítani a kirajzolási beállításain, például a színén.

A detektorgeometrián kívül el kell végezni az aktív térfogat, és a régió térképhez tartozó térrészlet kirajzolását. Mindkettő egy volumetrikus adat, amihez tartozik volumetrikus leíró, ami megadja a fizikai térben elfoglalt helyét. Mivel ez egy téglatest lapjainak a középpontól való távolságát adja meg, első lépésként ezt át kell számolni, hogy vertex adat legyen belőle. Miután ez megtörtént, az OpenGL-ben megszokott módon már ki lehet rajzolni, annyi módosítással, hogy ezek a téglatestek szintén vonalakból legyenek kirajzolva amiatt, hogy ne foglaljanak el túl sokat a képernyőből. Ehhez a lapokra kellett bontani a téglatestet, és mindegyik laphoz óramutató járásával megegyezően kirajzolni az éleket a csúcsok között, arra is figyelve, hogy az utolsó és az első között is legyen vonal.

A koincidencia szám megadja, hogy egy modul hány szemben lévő modulra lát rá. A szemben lévő modul indexe (o_i) a következő képlettel kapható meg, ahol a_i a jelenlegi modul indexe, *n* pedig a modulszám: $o_i = \left(a_i + \frac{n}{2}\right) mod\ n$. A modulóra azért van szükség, hogy a képlet a szemben lévő elem indexére ne adhasson vissza a modulszámnál nagyobb indexet, és figyelembe legyen véve, hogy a modulindex ciklikus. A szemben lévő elem indexe és a koincidencia szám ismeretében már meg tudjuk határozni az indexét a szemben lévő modulok első, és utolsó elemére a következő képletekkel, ahol c a koincidencia szám: $o_{min} = \left(o_i - \frac{c}{2}\right) mod\ n$, illetve $o_{max} = \left(o_i + \frac{c}{2}\right) mod\ n$. A modulókra itt is szükség van, mivel ennek számolásakor ugyanúgy belefuthatunk a problémába, hogy a képlettel kapott elem indexe nagyobb, mint a

modulszám. Ennek a kódban való megvalósítása után már meg lehet határozni, a kezdő, és záró indexét a modulok listájának, amire rálát az éppen vizsgált modulunk. Ennek az iterációjakor egyedül arra kell figyelni, hogy amennyiben a körbe ért az index, akkor ne indexeljük túl a modultömböt. Ezen modulok indexeinek meghatározásával a korábban említett modul színezési funkciót fel tudjuk használni. Amikor az OpenGL paint függvényén belül iterálunk végig modulonként a komponenseken, a kiválasztott modulok előtt meg kell változtatni a rajzolási színt. Az aktív modul esetében ez a szín élénk piros, a látott modulok esetében pedig egy halványabb piros (8. vagy 13. ábra).

A komplex feladat a kijelölt modulok által az aktív területből látott rész kiszámítása volt. Ennek az első lépése a probléma pontos megértése és egyszerűsítése. Ehhez először elő kellett állítani azt a térbeli testet, ami lényegében az a térfogat, amit a szemben lévő modulok látnak egymásból. Ez előállítható az aktív modul négy csúcsának, és a szemben lévő első modul kettő első, és az utolsó modul utolsó két csúcsának a térbeli nyolcszögéből (oktaéder), majd szemben lévő modulokhoz közelebb lévő lap modulokon kívülre való transzformálásával. Ezt a transzformációt úgy kell elvégezni, hogy az aktív modul lapját és a szemben lévő modul lapját összekötő éleknek az irányvektorjai ne változzanak (vagyis arányosan méretezni kell a mozgatott lapot). Ezt látható a 9. ábrán is.



9. ábra: A szemben lévő modulokhoz tartozó lap csúcsait a hozzájuk tartozó *v* vektorok irányvektorja mentén kell mozgatni.

Ennek a látott térbeli nyolcszögnek a meghatározásával már leegyszerűsíthető a probléma arra, hogy egy a nyolcszögnek, és az aktív területet jelképező téglatestnek a térben hol található a metszete, és ez hogyan néz ki egész pontosan. Ez a metszet-probléma különböző 3d szoftverekben több módon is implementálásra került, ezért elsőre azt hittem, hogy lesz az interneten elérhető kód, amit fel lehet használni a probléma megoldására. Ez sajnos nem így volt, mivel a legtöbb esetben ez a probléma vagy csak kétdimenzióra van implementálva, és a módszertana nehezen átültethető háromdimenzióba, vagy akkora függőségekkel, és bonyolultsággal járó projektek voltak, amik túlmutatnak ezen a programon, és még azok sem oldották volna meg teljesen a problémát. Emiatt döntöttem úgy, hogy egy saját implementációt használok.

Mivel több biztosan nem változó elemem van, ezek miatt több egyszerűsítéssel is lehet élni. Az első, hogy mindig tudom, hogy a két térbeli sokszög közül melyik a belső, amiből a másik levág részeket. A másik, hogy biztosan tudom, hogy ennek a belső téglatestnek a koordináta-rendszerhez képest csak merőleges lapjai vannak. A harmadik, hogy az előállítási jellegéből adódóan a külső testnek csak olyan lapjai vannak, aminek ismert két lapja ugyanabban a síkban van, mint a belső testnek ugyanilyen irányban álló

két lapja, és a másik négy lapjának pedig ugyanezen síkon rögzítve vannak a normálvektorjai. Ismert az is, hogy a két ugyanolyan normálvektorú síkpárra igaz, hogy a külső testhez tartozó síkok biztosan kívül helyezkednek el, vagyis ez a test az adott tengelyen szélesebb. Ebből több egyszerűsítő következtetés vonható le. Egyrészt csak olyan sík metszheti el a belső testet, aminek a normálvektorja nem párhuzamos a koordináta rendszer egyik tengelyével sem. Másrészt az összes lehetséges metszet által okozott új élnek az irányvektorja egyenlő lesz a két síkpárnak a normálvektorjával, vagyis az egyik tengellyel biztosan párhuzamos.

Ezeknek a számításoknak a leegyszerűsítésére, és az átláthatóság miatt létrehoztam egy csak ilyen geometriával foglalkozó modult, amiben a különböző térgeometriai egységeket megvalósítottam, ilyen egységek a vonal, a sík, a téglatest, és a térbeli sokszög. Ezekre mind megvalósítottam a használatukat leegyszerűsítő függvényeket, illetve a metszéseket számoló algoritmusokat. Ez a modul jól jött máshol is, mivel voltak megvalósított osztályok, amiben a kiszámolt adatokat lehetett tárolni a kirajzoláshoz. Ennek a megvalósításában segített a Qt beépített *QVector3D* osztálya, amiben a gyakran használt vektor műveletek meg voltak valósítva, így ezeket nem kellett implementálni.

Ezek ismeretében már meg tudtam írni egy algoritmust, ami kiszámolja a két test metszetét, amit az OpenGL már képes lesz kirajzolni. Az algoritmus végig iterál a belső téglatest lapjain. Eközben bizonyos kritériumuknak megfelelő pontokat keres a belső lapjain, amikből bizonyos kritériumok alapján egy sokszöget készít, amik végül a metszet test lapjait fogják adni. Egy pont két esetben kerülhet a sokszög csúcsai közé. Az egyik eset, ha az adott csúcsot tartalmazza a külső test. A másik eset akkor következik be, ha végig iterálunk a külső test lapjain, és az éppen vizsgált belső lapot metszi valamelyik. Ebben az esetben a két lap metszetének eredménye egy szakasz. Mivel ilyen esetben a belső test lapját metszi a külső lapja, a kapott szakasz meg fog egyezni a metszet testnek a térben egy ugyanott található élével. Mivel ebben az esetben a szakasz végpontjai biztosan csúcsai is a metszet testnek, ezért ilyen esetben ezek a pontok is hozzáadásra kerülnek az éppen vizsgált lapból készülő sokszög pontjaihoz. Ezzel megkaptuk a metszettest sokszög lapjainak egy részét, és biztosan ismerjük az összes pontját. A következő lépés, hogy a definiáljuk a hiányzó lapokat, amiket ezekből a csúcsokból elő lehet állítani. Csúcsból akkor lehet egy lapot csinálni, ha az összes egy síkon található, és legalább négy csúcsból áll. A hiányzó csúcsok kitöltéséhez végig iterálok a meglévő

lapok élein, és ellenőrzöm, hogy az él mindkét felén található-e lap. Amennyiben nem, összegyűjtöm az összes olyan lehetséges pontkombinációt, ami nem tartalmazza azoknak a lapnak a pontjait, amiben az él van (az élet alkotó két pontot kivéve), és lap alkotható belőlük, majd ezeket hozzáadom a metszettest lapjaihoz. Ezzel az algoritmussal sok belső lap is keletkezik, de mivel a metszet test konvex, így ez nem probléma. Erre egy lehetséges megoldás lenne, hogy leellenőrizzük, hogy a lap középpontja a metszet testen belül vagy kívül helyezkedik el. Miután megvannak a test lapjait alkotó sokszögek a kirajzoláshoz már csak óramutató járással megegyezően sorrendbe kell rendezni őket.

Ebben az algoritmusban több optimalizációt is alkalmaztam, amitől bizonyos esetekben majdnem a teljes metszet számítást ki lehet hagyni. Ilyen optimalizáció például, hogy a teljesen tartalmazott belső testet csak konvertálja át a kimeneti formátumra mindenféle metszet számítás nélkül, ha annak minden csúcsát tartalmazza a külső test. Ugyanilyen, de csak részleges gyorsítás alkalmazva van lapokra is, olyan esetben, ha egy lapnak mind a négy csúcsát tartalmazza a külső test, akkor a lapra nincs szükség metszéseket számolni.

Az algoritmus egyetlen hátránya, hogy elég lassú a térbeli metszet számítások jellege miatt, így külön szálon kell futtatni a számolást. Ehhez a korábban többször említett *QConcurrent* könyvtárat használtam. Mivel az algoritmus eredményét előállító függvény egy egyszerű tömb eredménnyel tér vissza, így a korábban kifejtett future watcher-t használó technikát alkalmazva egyszerűen megoldható volt a háttérszálon való számolás, a GUI további folyamatos működésével.

A felhasználó módosíthat bizonyos beállításokon, és ezeknek a módosításoknak az eredményét a nézetnek ki kell értékelnie. A detektorbeállításoknak a legnagyobb része nem módosítható. Ilyen fix beállításokból kettőt lehet felül írni, hogy láthassuk ezen módosításoknak élőben az eredményét, a koincidencia számot, illetve a volumetrikus adat leírót, ami az aktív térfogatot írja le. Ezenkívül megjelenítéshez kapcsolódó és állítható beállítások a jelenleg aktív és vizsgált modul indexe, és a különböző fent kifejtett funkciók megjelenítésének a kapcsolója.

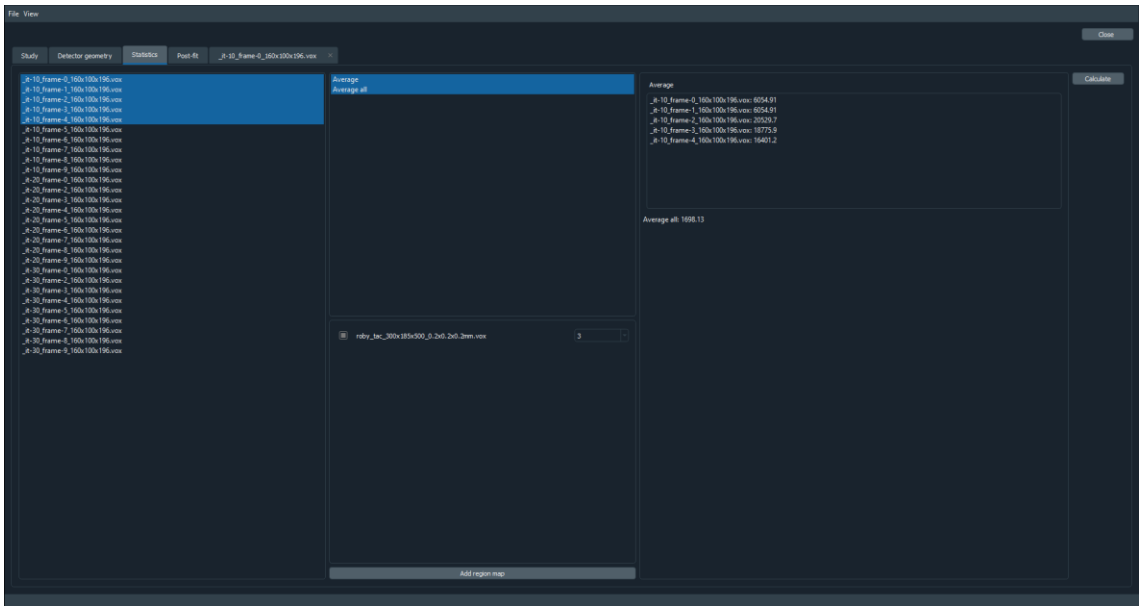
A nézet teljesen működőképes, viszont ehhez több jövőbeli fejlesztést is el tudok képzelni. A metszetszámítás például erős optimalizációra szorul. Ez egy jövőbeli tervezői döntés, hogy a meglévő algoritmus legyen továbbfejlesztve, legyen újraírva, vagy egy nagyobb külső könyvtár felhasználásával legyen átalakítva a meglévő kód. Egy másik fejlesztés a felhasználói élményt tudná növelni. Ez egy gyorsan implementálható

fejlesztés lenne. A metszetben létrehozott forma jelenleg ugyanabban a tömbben van tárolva, mint amibe a másik szál beírja az adatokat. A kirajzoló widget ebben a tömbben tárolt adatokat csak akkor kéri le, amikor kirajzolja a formát, ezért ennél sokkal nagyobb prioritású fejlesztések történtek. Viszont a gyakorlatban néhányszor tesztelés közben elő fordult, hogy egyszerre nyúlt hozzá a két szál a tömbhöz, így értelmetlen adatok lettek a lépernyőre rajzolva. Ez persze nem volt prioritásos, mert nem okozott stabilitási problémát, és az áthelyezés végeztével megjelent a helyes adat. Ezért ennek két megoldási lehetősége van: vagy szálbiztossá tesszük ezt a tömböt, vagy amíg nem fejeződött be a számolás, addig nincs kirajzolva a metszetben található forma.

4.8 Statisztika

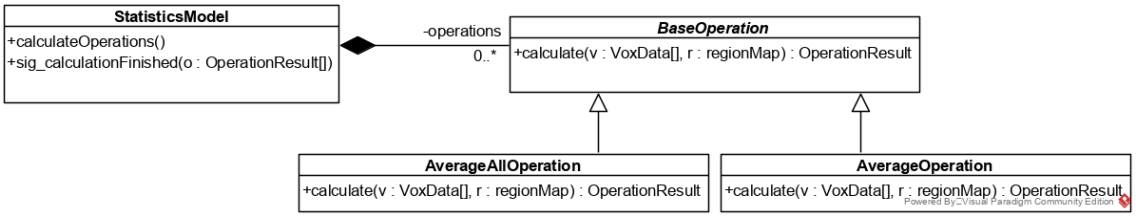
A statisztika funkció a programban a különböző, rekonstrukcióban található volumetrikus adatok kiértékelésére készült. Egy kisebb volumetrikus adat is több tízezer számot tartalmazhat, de a milliós nagyságrend sem elképzelhetetlen. Ezért az ezeken végzett műveletek még egy számítógépnek is sok időt vesznek igénybe nemhogy kézzel elvégezni ugyanezeket a számításokat. Emiatt ez egy fontos funkciója a programnak. Például ilyen volumetrikus adaton elvégezhető művelet lehet a voxelek átlagértékének kiszámolása.

A nézetben (10.ábra) kiválaszthatóak a volumetrikus adatok, amin el kívánjuk végezni a műveletet, az elvégezni kívánt műveletek, illetve egy régiótérkép és a hozzá kiválasztott régió sorszáma, amivel a számolt volumetrikus adatok voxelei leszűkíthetőek azokra, amiket ez az adott régió meghatároz. Ezután egy hosszabb vagy rövidebb számolás után megkapjuk az eredményeket műveletekre bontva. Egy műveletnek lehet egy egysoros eredménye, mint például az összes kiválasztott adat voxeleinek átlaga, ami egy szám, vagy lehet egy több elemből álló eredménye, ami például volumetrikus adatonként számolja az átlagot, így minden adathoz tartozik egy szám. A 10. ábrán látható egy statisztika műveletnek a használata és eredménye.



10. ábra: A kiválasztott volumetrikus adatokon, és régió térképen elvégzett számítás eredménye a két kiválasztott művelettel.

Ahhoz, hogy a fent leírt nézethez tartozó model moduláris legyen, könnyű legyen rajta bővíteni, ne egy osztályban legyen az összes művelet megvalósítva, és a nagy adatmennyiség, illetve hosszú futási idő miatt ne a GUI szálon fusson, fontos volt egy jó alapstruktúrát kialakítani, amiből a jövőbeli bővítések kiindulhatnak. Ehhez a legflexibilisebb megoldás, amit találtam, hogy a műveleteket egy heterogén kollekcióban tárolom. Ennek a 11. ábrán látható a felépítése. Ez sok szempontból előnyös. Minden művelet kódja ebben az esetben egy külön erre szánt osztályba lesz megvalósítva, így átlátható és könnyen módosítható a meglévő kód. Mivel új osztályt kell létrehozni új művelet létrehozásához, így meglévő kódon nem kell módosítani ennek hozzáadásakor. Ráadásul mivel minden művelet teljesen független egymástól, és egymástól független a visszaadott eredményük is, így akár az összes művelet külön szálon is futtatható. A heterogén kollekció lehetővé teszi, hogy minden számolásnak ugyanaz legyen a be- és kimenete, így le tudnak származni egy közös absztrakt osztályból.



11.ábra: A statisztika model-ben használt műveletek osztályhierarchiája, és fontosabb függvényei

A műveletek párhuzamos futtatására a Qt concurrent osztály map függvényét vettem igénybe, mivel ez automatikusan képes párhuzamosan kiértékelni a kapott műveleteket. Ennek eredménye egy *QFuture* lesz, amire egy ehhez kitalált segédosztály segítségével fel lehet iratkozni, hogy küldjön ki egy signal-t, amikor megvan az összesnek az eredménye.

4.9 Görberajzolás és utólagos görbeillesztés

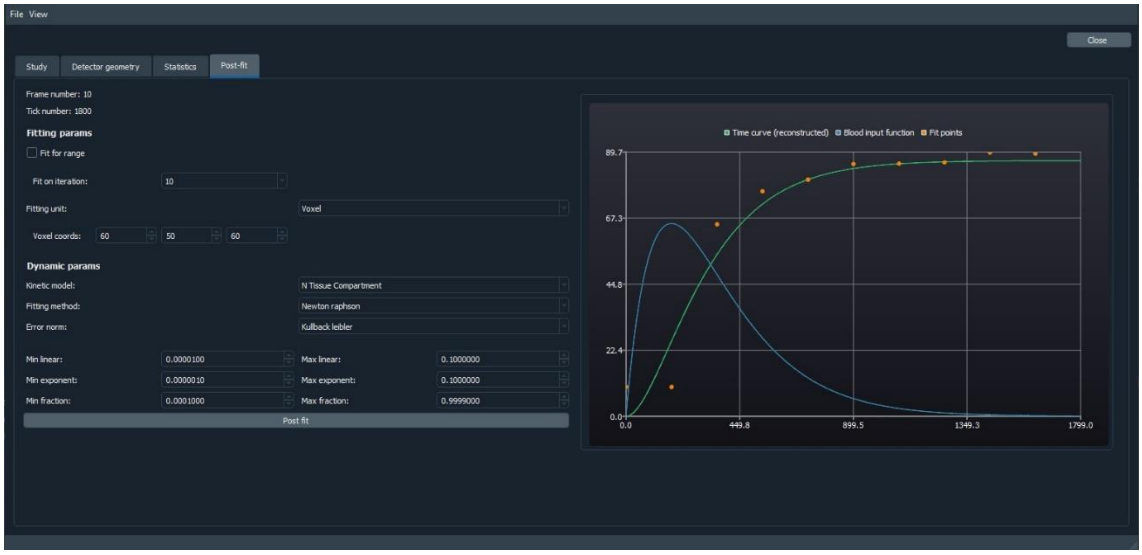
A dinamikus rekonstrukció során a különböző időpillanatokban készített volumetrikus adatokban található voxelek aktívitás értékei változnak. A képalkotás minőségének javításához fontos ismerni ennek a folyamatnak a pontos változásait. Emiatt fontos, hogy legyen a programban olyan funkció, ami képes egy vagy több voxelre meghatározni, hogy az idő múlásával a voxel aktivitása hogyan módosult, illetve hogy a program ezekből az adatokból egy folyamatos, az adatokra illesztett görbét fel tudjon állítani. Ezt egy, vagy maximum néhány voxelre meg lehet jeleníteni egy diagramon, ahol ezek diagramon ábrázolva vannak. Ennél nagyobb mennyiségű voxel esetén pedig egy fájlkimenet az optimális, amibe vagy a kimeneti diagram adatai kerülnek egyszerű szöveges formátumban, vagy az SDK-ban megvalósított parametrikus volumetrikus adat, ami a hasonló adatok tárolására van kitalálva.

Mivel a görbeillesztés egy szerves eleme a rekonstrukciónak, így rengeteg algoritmus és adat osztály meg volt már előre valósítva az SDK-ban, így elsősorban a meglévő kód erre a célra való használatát, illetve az ehhez tartozó számítások gyorsítását cache segítségével kellett megvalósítanom. Mivel a görbe illesztés több helyen szükséges a kódban, és a jövőben még több helyen lehet rá szükség, így ezt is a lehető legkevesebb függőséggel, minél újra felhasználhatóbban kellett megírnom. Ezenkívül az ehhez tartozó model-t úgy kellett megvalósítanom, hogy ez teljesen le legyen választva a program többi részéről, mivel ez a része az SDK-nak gyakran van módosítva. A program fejlesztése alatt is az ehhez a részhez tartozó kódot többször módosítani kellett az SDK változása miatt, így már a fejlesztés alatt sok időt megspórolt ez a leválasztás. A másik indok, ami miatt érdemes leválasztani, az az ehhez kapcsolódó SDK kód fordításának bonyolultsága. Az ehhez tartozó kódban rengeteg speciális kód szerepel, amihez nehéz bekonfigurálni a fordítót. Emiatt a fejlesztés megkönnyítésére, amikor nem ehhez a területhez fejlesztettem, ez a kód nem is volt az alkalmazásba bele fordítva. Mivel a Qt saját fejlesztői környezetében több esetben kényelmesebb a GUI fejlesztés, de nehéz lett volna

a qmake-et ennek a kódrészletnek a lefordítására bekonfigurálni, így ennek a leválasztása sokat gyorsított a fejlesztésen.

Jelenleg a görbeillesztés a programon belül két kontextusban van használva. Az egyik a szeletmegjelenítőben egy voxelre kattintva az adott voxel adataira történő görberajzolás (14. ábra), a másik pedig az utólagos görbeillesztés (12. ábra). A sima görberajzolást elsősorban a szeletmegjelenítővel együtt lehet kényelmesen használni, de valójában bárhonnan lehet használni, ahol ismerjük a voxel pozícióját. Egy szeletre kattintva az egér pozíciójából és az adott szelet mélységéből ki lehet számolni a kattintott voxel pozícióját. Az adott voxel pozícióját ismerve a görbeillesztő osztály már meg tudja határozni, hogy a különböző időpillanatokban készült volumetrikus adatok értékei közül melyikeket kell lekérni, és ezen értékek és időpillanatok ismeretével már meg tudja határozni az adott pontokhoz tartozó görbét.

Az utólagos görbeillesztés hasonló elven működik, mint a görbe kirajzolása. A különbség a kettő között, hogy a kirajzolásnál a rekonstrukció beállítási paramétereit veszi alapul, míg az utólagos illesztéskor ezek a paraméterek kézzel változtathatóak. Ez hasznos lehet például a különböző illesztési algoritmusok, vagy különböző bemeneti paraméterek illesztési eredményeinek az összehasonlítására. Ezeket a beállításokat egy megfelelő SDK-beli osztály definiálja. Ez többek között az illesztés, illetve az illesztett függvény különböző beállításait tartalmazza. Utólagos illesztéskor létrehozok a rekonstrukció beállításaiból készült másolatot, és olyan esetben az kerül átadásra a görbe illesztő model-nek.



12. ábra: Az utólagos görbeillesztéshez tartozó nézet, és a változtatható paraméterek

A cache-elés megvalósítása előtt egy voxel értékeinek illesztése körülbelül 90 másodperc körül volt. A meglévő algoritmusokon nem lehetett optimalizálni, viszont több köztes érték eredményét el lehetett menteni, mivel ezek a számolások között nem változtak. Az egyik ilyen érték a minden illesztés előtt lefuttatott véraktivitás-függvényhez kapcsolódó szimuláció volt. Mivel pontosan lehet tudni, hogy ez csak akkor módosul, amikor a mögöttes leíró struktúra bizonyos beállításai módosulnak, ezért azt csak akkor kell újra számolni, ha ezek is módosultak. Egy másik optimalizáció, ami sokat gyorsított az illesztésen, hogy a görbe illesztő model is használja a meglévő volumetrikus adat cache-t, így ezeket is csak akkor kell beolvasni újra, ha változtak. Ezekkel az optimalizációkkal az első illesztésnek ugyanolyan lefutási ideje maradt, de többszöri újra futtatásra már 10 másodperc körül volt, így ezzel majdnem kilencszeresére sikerült a többszöri görbeillesztést gyorsítani. Ezeknek az optimalizációknak a nagy része a legtöbb esetben viszont nem alkalmazható, amikor utólagos illesztéskor változnak ezek az érékek, amiktől a köztes eredmények is függenek. Ez egy elképzelhető jövőbeli fejlesztés lehet, hogy ezt hogyan lehetne valamilyen módon mégis gyorsabbá tenni. Ehhez egy optimista algoritmust tudnék például elképzelni, ami egy érték megváltozásakor előre elkezd számolni a megváltozott érékekkel. Ennek a hátránya nyilván az ehhez tartozó kód komplexitása lenne. Egy másik lehetséges optimalizáció a párhuzamosítás lenne, de itt is a leglassabban futó ág egy limitáló faktor lenne.

Az illesztés sebessége miatt itt is át kellett rakni a görbe illesztőt egy háttér szálra. Itt a számolás hossza, és a be és kimenő adatok mennyisége miatt a korábbiakkal ellentétben nem Qt concurrent-el, hanem *QThread*-el valósítottam meg. Ez előnyös abból a szempontból, hogy a szálon futtatott kód teljesen független a hívó osztálytól, még a Qt concurrent-ben a hívó osztálynak ismernie kell a hívott függvényeket. Ez a korábban kifejtett teljes szeparáció miatt kifejezetten előnyös. A szál manuálisan lett létrehozva, így lényegesen könnyebb vele futás közben kommunikálni, mint egy future-rel. Ez előnyös lehet a jövőben, amennyiben szükséges a korábban írt potenciális fejlesztés.

4.10 Diagram nézet

A diagram nézet egy több helyen újra felhasznált komponens. Feladata megjeleníteni a különböző típusú pont halmazokat, illetve görbéket. A szoftver jelenlegi állapotában elsősorban az utólagos illesztés eredményének kijelzésére van használva, de úgy lett megírva, hogy bármilyen a Qt Chart-al komptatibilis diagram típusra kibővíthető

legyen a használata. A nézet a QtCharts Qt-s könyvtárban található diagram típus köré épül. Amiatt készült, mivel önmagában ez a diagram típus kényelmetlenül használható. A könyvtár biztosít külön adatszerkezetet a diagramhoz, de ezzel együtt is sok boilerplate kóddal lehet csak használni, amit minden egyes alkalommal újra és újra meg kéne írni, ezzel átláthatatlanabb kódot eredményezve. Erre megoldás ez a diagram nézet.

Két alapvető probléma van a könyvtár által biztosított diagram nézettel. Az egyik, hogy nincs globális beállítása, vagy bármilyen előre konfigurálható beállítás halmaza, így minden létrehozáskor minden egyes példányra külön be kell állítani a szükséges dolgokat. A másik, hogy nincs integrálva a Qt signal-slot rendszerébe, nincs külön olyan slot-ja, ami le tudná kezelni, hogy kap egy megadott diagram halmazt, és ezt automatikusan jelenítse meg. Emiatt készítettem el ezt a widget-et, hogy a programon belül ez a programon belülről bárhonnan, a megjelenítendő adattól függetlenül használható legyen.

Első lépésként létrehoztam egy model osztályt a *SeriesContainer*-t, ami a különböző típusú diagramokhoz képes tárolni az adatokat. Ezek a diagram típusú adattárolók a könyvtárba vannak beépítve. Alapvetően a legtöbb úgy működik, hogy minden x koordinátához tárolnak egy y koordinátát, meg néhány diagram specifikus adatot, például a diagram nevét, színét vagy egyéb ilyen adatokat. Egy adattároló típusból többet is hozzá lehet adni a diagramhoz, ezért a model osztályba ezeket tömbbe (megvalósításban C++-os vector) kell tárolni. Ezenkívül minden adattároló típust diagram típusonként különböző módon kell hozzáadni a diagramhoz, így ezeket külön tömbben kell tárolni. Ennek a végeredménye, hogy a *SeriesContainer* diagram típusonként külön tömbökben tárolja a különböző adattípusokat. A nézetnek létre hoztam több slot-ot, amivel tud reagálni a nézet eseményekre. A legfontosabb az *updateSeriesContainer*, aminek át lehet adni az adat konténert, és ez alapján befrissíti a diagram nézetet.

Egy ilyen diagram adattároló típus hozzáadása sok lépésből áll, részben emiatt is lett létrehozva ez a nézet, hogy ezt elfedje. Egy diagram nézetben több adattípus is megjeleníthető egyszerre. Emiatt a *SeriesContainer* tömbjein, és elemein végig iterálva egyenként lesznek hozzáadva a chart-hoz. A chart egy olyan könyvtárbeli adattípus, amelyet a diagramnézet már képes egy az egyben megjeleníteni. A chart-hoz hozzáadás több lépésből áll. Elsőként meg kell keresni az x és y mentén a maximum értékeket. Ezt a jövőben különböző gradiens alapú maximumkereséssel lehetne fejleszteni, mivel sok vagy nagyméretű adatoknál sok időt vehet igénybe. Ezután diagram típus alapján

szétválasztva történik a hozzáadás. Közös rész, hogy a jelenleg használt összes diagram típushoz van OpenGL alapú gyorsítás, amit be kell kapcsolni. Ezután az összes adattárolót egyenként hozzá kell adni a chart-hoz, és újra számoltatni a különböző diagram típusok relatív méretét egymáshoz képest, hogy helyes legyen a kirajzolás.

Ezenkívül diagram létrehozáskor néhány alapbeállítást is el kell végezni. Ezeket programozási szempontból kevésbé tartom fontosnak, de szükségesnek tartom megemlíteni. A diagram alapvetően a sebességre van optimalizálva, és külön be kell állítani, hogy nagyobb adatmennyiségnél is pontos legyen a megjelenítés, és leolvashatóak legyenek az adatok. Alapvetően elég „blokkos” a megjelenítés, mivel ki van kapcsolva az élsimítás. Mivel alapvetően a diagram CPU-n lenne render-elve, ennek kivétele sokat gyorsít a megjelenítésen. Viszont az általam használt minden diagram típus támogatja az OpenGL alapú gyorsítást, így ez a GPU-n lesz kirajzolva, ahol az élsimítás nem jár észlelhető sebesség vesztéssel. A másik a diagram megjelenítési témája. A felhasználói élmény szempontjából sokat segít a megértésben, ha a felületnek egységes kinézete van. A diagram alap állapotban egy világos témát használ, viszont a programnak sötét témája van, amiatt erősen elüt a felhasználói felület többi részétől, és sokkal kiemeltebbnek tűnik, nehezíti a megértést. Szerencsére vannak beépített diagram témák a könyvtárban. Annak ellenére, hogy a kékes témához így sem illeszkedik teljes mértékben. A diagram nézet használatára látható példa a 14.ábrán.

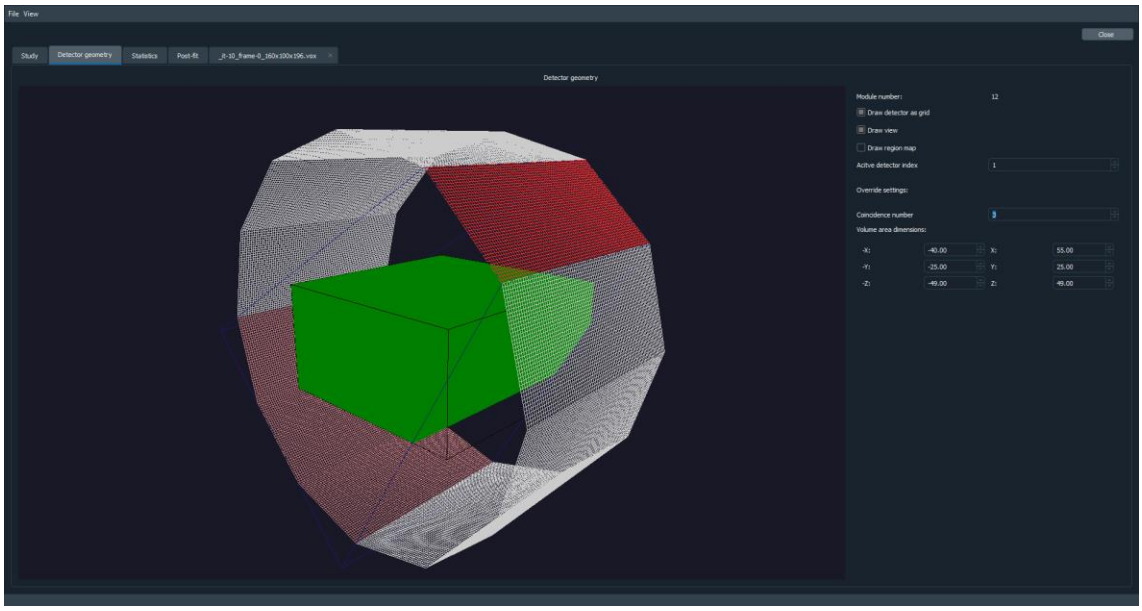
5 Értékelés, tesztelés

A többi fejezetben a tervezési folyamatot és az implementációt mutattam be részletesen. Ebben a fejezetben a célom a projektet bemutatni a felhasználó szemszögéből, illetve a projektet értékelni a megfogalmazott követelmények teljesítését illetően.

5.1 Felhasználói felület értékelése és tesztelése

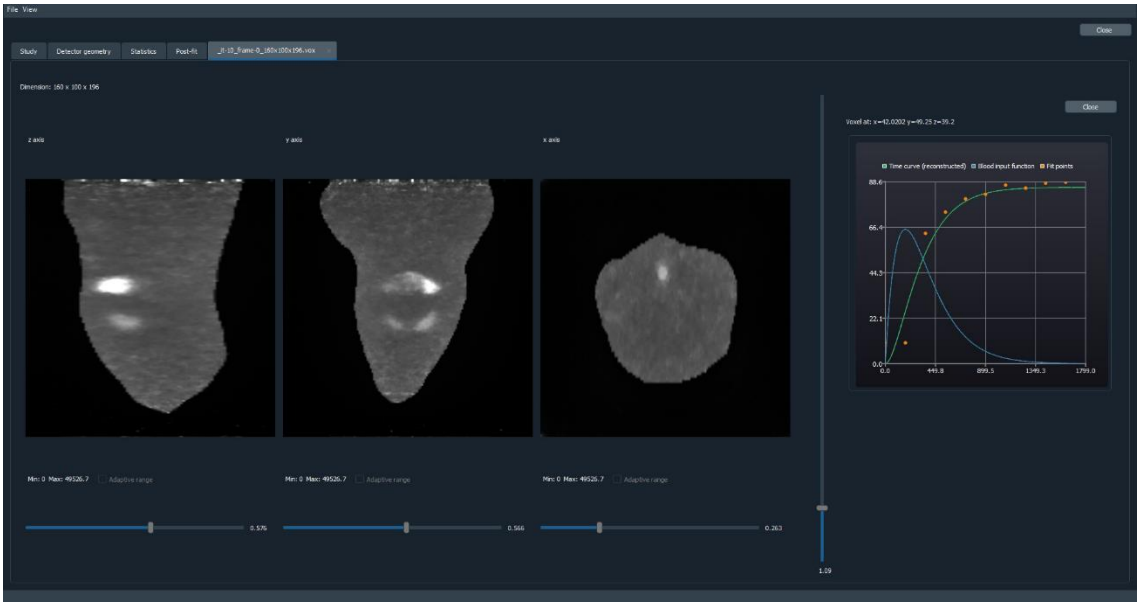
A szoftver végfelhasználói programozók és fejlesztők. Emiatt az általgosnál valamivel bonyolultabb felületet is kényelmesen tudnak használni, ugyanakkor fejlesztőként egy átlag felhasználónál sokkal kritikusabbak vagyunk egy-egy programban megtalálható problémára. Ettől függetlenül viszont ugyanazok a kritériumok is érvényesek, mint egy átlagos felhasználói felületre. Legyen könnyen használható, logikus és ne kelljen sokat várni a programra. Ezeket összességében szerintem sikerült megvalósítanom. A tab elrendezés miatt a rekonstrukción belül egyszerű és gyors a navigáció. Egyértelmű, hogy mi hol van, mivel minden kifér a képernyőre, és nem ikonokból kell kitalálni, hogy melyik gomb mit csinál. Ezen felül a külön nézetek is teljesen átláthatóak. A különböző tab-ok egy-egy funkciót csoportosítanak, és csak azokat tartalmazzák, így összességében a felhasználói felületet könnyű használni.

Két egyedibb nézetre térnék ki, amiknek a megvalósítása sokat adott hozzá a program használhatóságához. Ezek a detektorgeometria megjelenítő, és a szelet megjelenítő, illetve az ehhez tartozó görbeillesztő.



13. ábra: A detektorgeometria nézet használat közben

Ezek hozzáadásával sokat nőtt a felhasználói élmény és a program használhatósága is, mivel egy nagyon egyedi problémára biztosít mind a kettő megoldást. Használat közben mindkettőt logikus használni. A detektor geometriában az egérrel lehet navigálni. A bal egér gombot lenyomva forgatni, a jobb egér gombot lenyomva közelíteni és távolítani lehet a nézetet a középpontól. Ennek a használata megszokható, de az irányítás kényelmesebbé tétele egy potenciális fejlesztés lehet. Ezenkívül a metszet kiszámítása bizonyos esetekben sok időbe telik. Ha ezt a funkciót gyakran kell használni, hosszútávon nem lesz a legkellemesebb az állandó várakozás. Ezt leszámítva kifejezetten kényelmesen használható ez a nézet. Szeletmegjelenítővel szintén hasonlóak a tapasztalatok.



14.ábra: A szeletmegjelenítő nézet, jobb oldalt a kiválasztott voxelre történt illesztés eredménye

Ez a nézet is kiválóan alkalmas a megtervezett funkciók használhatóságának szempontjából. A görbeillesztés sebessége sem zavaró, mivel a megvalósított cache-elés miatt ez csak az első alkalommal lassú. A három szeletnézeten könnyű a navigáció, illetve a nézetek méretezését állító csúszka is kényelmesen használható.

A többi nézetre nem érdemes részletesen kitérni, de azokkal is hasonlóak a tapasztalatok. Egy-két helyen a sebesség optimalizációra szorul, de összességében egy jól, és ami még fontosabb, gyorsan használható felhasználói felület, mivel sok felesleges kattintást meg lehet spórolni, és a funkciókhoz van építve a teljes felület.

5.2 A program megfogalmazott követelményekkel szembeni értékelése

A követelmények közül messze a legfontosabb a tesztelhetőség és a bővíthetőség volt. Emiatt egy, a fejlesztés közben felmerült funkció implementálásának értékelése egy jó lehetőség ennek az ellenőrzésére. Az új funkció az utólagos illesztés, és a hozzátartozó nézet volt. Ennek a tervezési és megvalósítási részleteit kifejtettem a korábbi fejezetekben, viszont arról nem volt szó, hogy ez nem volt a projekt elejétől tervben. Ez előnyös, mert egy kifejezetten komplex új funkció, és tartalmaznia kell korábban már megírt nézetkomponensek, a diagramnézet használatát. Az új nézet hozzáadása a tab widet-hez csak néhány sor volt, ami a nézetet létrehozza, és hozzáadja a tab widgethez. Ezt bármiféle már megírt kód módosítása nélkül végre tudtam hajtani, így az új nézet hozzáadása sikeres volt. Az utólagos illesztéshez szükség volt több, a StudyModel-ben elérhető, változóra, amikre nem volt még lekérdezőfüggvény. Mivel itt is csak új kódot kellett hozzáadni, meglévőn nem kellett módosítani, ezért ezt is sikeresnek tekintem. A diagramnézetet is fel tudtam használni bármiféle módosítás nélkül. A diagram adatokat tartalmazó konténerhez szükséges volt hozzáadni egy új diagramtípust, és ennek a beállításait meg kellett adni a diagram nézetnek az adatfrissítéshez kapcsolódó függvényében. Technikailag itt sem volt szükség meglévő kód módosítására, de sok diagramtípus támogatásakor ez a megoldás már nem lenne praktikus, ilyen esetben ezt a jövőben átírnám. Összességében ez egy elég sok kódot tartalmazó, és komplex funkció, de hozzá lehetett adni a kódbázishoz bármilyen meglévő kódbázis nélkül, így ilyen szempontból ez elérte a sikerét. Természetesen ez egy eléggé különálló funkció is, így nem mutatja meg az összes lehetséges problémát új kód hozzáadásakor, de ennek a funkciónak az implementálásával nem volt ilyen jellegű probléma. A szoftver eddigi

életciklusa során sem ennek a funkciónak a hozzáadásával, sem egyéb másik új funkcióval nem merült fel a bővíthetőséggel probléma, ezért ebből a szempontból a fejlesztést eredményesnek tartom. Sajnos viszont csak hosszú távon derülhet ki, ha ezzel esetleg probléma van.

A többi felállított követelményt szintén teljesíti a program. A felhasználói felület intuitíven használható, könnyen és logikusan bővíthető újabb elemekkel. A rekonstrukciók kezelése a tervezett módon működik, a leírt követelményeknek megfelel. Az egész programban a volumetrikus adatok elérhetőek a gyorsítótáron keresztül, így a lehető leggyorsabban hozzáférhetőek. Az SDK a program többi részéről jól le van választva, egyes helyeken annyira, hogy nem is szükséges az azt tartalmazó osztályt belefordítani a végleges kódba, így ilyen esetben csak az adott funkció esik ki. Ennek ellenére a kód jól fejleszthető maradt, moduláris, és fel van készítve a módosításokra. A jelenleg felépített kódstruktúra megőrzésében segít a sok öröklés és absztrakt osztályból leszármazás. Emiatt összességében a kód is kifejezetten jó minőségű lett. Ugyan ez nem volt a követelményeknek a része, de a fejleszthetőséget a jövőben mindenféleképpen növelni lehetne unit tesztek hozzáadásával. Ezek meggyorsítanák a fejlesztési folyamatot, mivel biztosak lehetnénk abban, hogy egy módosítással vagy új kód hozzáadásával nem rontanánk el már meglévőt.

6 Összefoglalás és további fejlesztési lehetősségek

Megterveztem és implementáltam a PET rekonstrukciós kutatást támogató, grafikus felülettel rendelkező szoftvert. Bemutattam a PET rekonstrukció típusait, illetve az ezek során keletkező vagy kiértékelendő adatokat. Részleteztem a követelményeket kitérve néhány egyedi szempontra, de itt az elsődleges szempont egyértelműen a bővíthetőség volt, ami maradéktalanul teljesítve is lett. Bemutattam a Qt-t, a kiválasztott GUI keretrendszert, kitérve az előnyeire és hátrányaira is egyéb hasonló framework-ökkel szemben. Végül részletesen bemutattam a különböző funkciókat és az ezekhez tartozó implementációt.

Az implementációs fejezetben több lehetséges jövőbeli fejlesztést leírtam, mint például a detektorgeometriában található látott térfogatot kiszámító algoritmus javítása. Ezenkívül az implementációs fejezetben főleg a különböző optimalizációs lehetőségekre tértem ki. Mivel sok funkció úgy van elkészítve, hogy a bővítése is fontos szempont volt, így ezeknél nem csak elképzelhető, de valószínű is a további fejlesztés. A statisztikát rengeteg új művelettel lehet bővíteni, vagy akár a rekonstrukció tab nézetéhez is bármikor hozzá lehet adni egy eddig előre nem látott funkciót. Viszont konkrét fejlesztésre is vannak ötleteim. A rekonstrukcióba jelenleg betöltődik az összes elérhető volumetrikus fájl, viszont az áttekintő nézetben csak az iterációkhoz tartozó, és nem evaluált adatok láthatóak. Lényegesen növelné a rekonstrukcióhoz tartozó volumetrikus adatok áltáthatóságát, ha ezeket valamilyen csoportosítható, például tree (fa) nézetbe rendeznénk. Egy másik potenciális fejlesztés, ami szerintem hasznos lenne, ha a rekonstrukciók elérhetőek és kezelhetőek lennének egy központi adatbázisból, és ezeket a programból egy kattintással meg lehetne nyitni.

Összeségében a követelményeket a program egytől egyig teljesíti. A projekt méretét leszámítva a felépítése kifejezetten jó, és emiatt jól átlátható is. A kód minőségét a fejlesztés elején felállított alapelvek segítettek megőrizni, így mind kisebb modulonként, mind összeségében jó minőségű a kód és a program is. Az előző fejezetben kifejtettem, hogy mitől volt könnyű új funkciót hozzáadni, és hogy ez miért nagy pozitívum a bővíthetőség szempontjából. A projekt teljesítette a célját és a felállított követelményeket, így mind tervezés, mind megvalósítás szempontjából sikeresnek tekintem.

Irodalomjegyzék

[1] Shepp, L. A., & Vardi, Y. (1982). *Maximum likelihood reconstruction for emission tomography. IEEE transactions on medical imaging, 1(2),* 113-122.

[2] Szirmay-Kalos, L., Kacsó, Á., Magdics, M., & Tóth, B. (2018). *Dynamic pet reconstruction on the gpu. Periodica Polytechnica Electrical Engineering and Computer Science, 62(4),* 134-143.

[3] Szirmay-Kalos, L., Kacsó, Á., Magdics, M., & Tóth, B. (2021). *Robust compartmental model fitting in direct emission tomography reconstruction. The Visual Computer,* 1-14.

[4] Edvin Syse: *TornadoFx* https://tornadofx.io/ (2021.11.13)

[5] The Qt company: *Qt* https://www.qt.io/ (2021.11.13.)

[6] JetBrains s.r.o: *Compose Multiplatform* https://www.jetbrains.com/lp/compose-mpp/ (2021.11.13)

[7] Google: Flutter on desktop https://flutter.dev/desktop (2021.11.13.)

[8] JetBrains: *Jetpack compose Github page* https://github.com/jetbrains/compose-jb (2021.11.13)

[9] The Qt company: *The Building blocks of Qt* https://www.qt.io/product/framework (2021.11.17)

[10] The Qt company: *About Us* https://www.qt.io/company (2021.11.17)

[11] Blanchette, Jasmin; Summerfield, Mark (June 2006). *"A Brief History of Qt". C++ GUI Programming with Qt 4 (1st ed.)* https://web.archive.org/web/20190923193951/https://my.safaribooksonline.com/0131872494/pref04 (2021.11.17)

[12] The Qt company: *Signals & Slots* https://doc.qt.io/qt-5/signalsandslots.html (2021.11.17)

[13] The Qt company: *Qt Charts* https://doc.qt.io/qt-5/qtcharts-index.html (2021.12.02.)

[14] The Qt company: *QOpenGLWidget Class* https://doc.qt.io/qt-5/qopenglwidget.html (2021.11.19)

[15] The Qt company: *QVector2D Class* https://doc.qt.io/qt-5/qvector2d.html (2021.11.22)

[16] The Qt company: *QVector3D Class* https://doc.qt.io/qt-5/qvector3d.html (2021.11.22)

[17] The Qt company: *QThread Class* *https://doc.qt.io/qt-5/qthread.html* (2021.11.19)

[18] The Qt company: *Qt Concurrent* *https://doc.qt.io/qt-5/qtconcurrent-index.html* (2021.11.19)

[19] The Qt company: *QVector Class* *https://doc.qt.io/qt-5/qvector.html* (2021.11.25)

[20] ColinDuquesnoy: *QDarkStyleSheet* *https://github.com/ColinDuquesnoy/QDarkStyleSheet* (2021.12.03)