Uniwersytet Kardynała Stefana Wyszyńskiego w Warszawie
Wydział Matematyczno-Przyrodniczy
Szkoła Nauk Ścisłych

Mariusz Niedziółka
Nr albumu: **99287**
Kierunek: **informatyka**

Wzajemna komunikacja bibliotek Qt Quick i VTK na przykładzie aplikacji do renderingu wolumetrycznego.

Praca licencjacka
Promotor:
dr Paweł Łubniewski

Warszawa 2018

Załącznik do Zarządzenia Nr 78/2014 Rektora UKSW
z dnia 14 listopada 2014 r.
Załącznik nr 3 do Zarządzenia Nr 39/2007 Rektora UKSW
z dnia 9 listopada 2007r.

.....
Imię i nazwisko studenta/studentki

.....
Nr albumu

.....
Wydział

.....
Instytut

.....
Kierunek

Dziekan Wydziału .....

.....

OŚWIADCZENIE

Świadomy(a) odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w żadnej uczelni. Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Oświadczam, że poinformowano mnie o zasadach dotyczących kontroli samodzielności prac dyplomowych i zaliczeniowych. W związku z powyższym oświadczam, że wyrażam zgodę na przetwarzanie* moich prac pisemnych (w tym prac zaliczeniowych i pracy dyplomowej) powstałych w toku studiów i związanych z realizacją programu kształcenia w Uczelni, a także na przechowywanie pracy dyplomowej w celach realizowanej procedury antyplagiatowej w ogólnopolskim repozytorium pisemnych prac dyplomowych.

.....
podpis studenta

*Przez przetwarzanie pracy rozumie się porównywanie przez system antyplagiatowy jej treści z innymi dokumentami (w celu ustalenia istnienia nieuprawnionych zapożyczeń) oraz generowanie raportu podobieństwa.

OŚWIADCZENIE

Oświadczam, że niniejsza praca napisana przez
Pana/Panią....., nr albumu .....
została przygotowana pod moim kierunkiem i stwierdzam, że spełnia ona warunki do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

.....
podpis promotora

Słowa kluczowe

VTK, ITK, Qt Quick, rendering wolumetryczny, renderowanie, komunikacja, adaptacja, obsługa zdarzeń, interakcja, wolumin, woksel, GPU ray casting

Dziedzina Socrates-Erasmus

11.3 Informatyka

English title

Mutual communication of Qt Quick and VTK libraries by the example of volume rendering application.

Spis treści

Streszczenie

Praca porusza problem komunikacji bibliotek Qt Quick i VTK, w szczególności problem wyświetlania wygenerowanych przez VTK scen w oknie aplikacji stworzonej przy pomocy biblioteki Qt Quick i języka QML oraz wysyłania do VTK zdarzeń związanych z interakcją użytkownika z aplikacją za pomocą myszy i klawiatury, odebranych przez bibliotekę Qt Quick. Na potrzeby pracy zostanie zaimplementowana aplikacja służąca do odczytywania woluminów zapisanych w formacie NRRD w celu ich przetworzenia przez VTK oraz wygenerowania scen z ich udziałem, a także wyświetlenia tych scen w oknie renderującym Qt Quick. Na jej przykładzie wyjaśniony zostanie poruszony problem oraz zaproponowane zostanie jego rozwiązanie, a także oceniona zostanie jakość i wydajność tego rozwiązania.

Rozdział 1

Wprowadzenie

1.1. Cel pracy i charakterystyka problemu

Głównym celem mojej pracy jest połączenie bibliotek Qt Quick i VTK we wspólnie działającą i komunikującą się nawzajem całość. Proces wzajemnej komunikacji opiszę na podstawie zaimplementowanej na potrzeby projektu aplikacji służącej do odczytywania plików NRRD i renderowania zapisanych w tych plikach obiektów przy pomocy renderingu wolumetrycznego. Dodatkowo aplikacja umożliwi przeglądanie warstw obrazów zapisanych w plikach NRRD, wybór funkcji transferu kolorów wokseli oraz zmianę parametrów viewportów, takich jak kolor tła oraz jasność i kontrast renderowanego obrazu.

Największym problemem z jakim trzeba się zmierzyć jest wyświetlenie obrazu wyrenderowanego przez VTK w interfejsie zdefiniowanym w oparciu o Qt Quick. Niestety nie da się bezpośrednio skomunikować VTK i Qt Quick, ze względu na to, że biblioteki te działają w oddzielnych wątkach aplikacji. Jest to szczególnym problemem jeśli chodzi o renderowanie obrazu. Zarówno Qt Quick jak i VTK mają oddzielne potoki OpenGL, służące do przetwarzania i renderowania sceny. Próba bezpośredniej komunikacji tych dwóch bibliotek doprowadzi do licznych błędów biblioteki OpenGL wynikających z faktu, że dwa oddzielne potoki biblioteki OpenGL próbują renderować sceny w tym samym czasie, a co za tym idzie próbują wzajemnie uzyskać dostęp do obiektów, które nie są jeszcze gotowe, by przekazać dane bibliotece, która ich potrzebuje.

Dla przykładu rozpatrzmy następujący scenariusz, zakładający, że mamy jedno wspólne okno renderujące dla VTK i Qt Quick:

1. Qt Quick renderuje obraz dostarczony przez bibliotekę VTK, wywołując funkcje z biblioteki OpenGL.
2. Użytkownik reaguje na wyświetlony obraz, chcąc np. obrócić kamerę za pomocą myszy.
3. Qt Quick wywołuje zdarzenie związane z kliknięciem i przesunięciem myszy i następnie wywołuje analogiczne zdarzenie w bibliotece VTK informujące ją, że użytkownik kliknął i poruszył myszką.
4. VTK chce obsłużyć to zdarzenie poprzez obrót kamery, a co za tym idzie wywołanie funkcji z biblioteki OpenGL.
5. VTK na koniec chce wyrenderować zmieniony obraz (wywołując funkcje biblioteki OpenGL), ale tego samego chce w danym momencie również Qt Quick.

Od razu można zauważyć, że wywołania funkcji z biblioteki OpenGL będą się przeplatać. Problem polega na tym, że obie biblioteki będą w tym samym momencie chciały wyrenderować obraz do tego samego okna renderującego. Spowoduje to liczne błędy w funkcjonowaniu biblioteki OpenGL, a w najgorszym wypadku krytyczne zatrzymanie programu, wynikające z faktu, że przeplatające się funkcje renderujące obraz będą chciały uzyskać dostęp do tych samych, często niegotowych do pracy obiektów biblioteki VTK.

Skoro nie da się bezpośrednio skomunikować biblioteki Qt Quick z VTK, należy uczynić to pośrednio, tworząc klasę adaptującą i komunikującą te dwie biblioteki, gwarantującą, że przed wyrenderowaniem obrazu w Qt Quick, obraz w VTK zostanie w pełni wyrenderowany i vice versa. Zagwarantuje to nam swobodny dostęp do obiektów niezbędnych do wygenerowania sceny i oddzielność wywoływania funkcji z biblioteki OpenGL.

W projekcie chcę uniknąć sytuacji, gdy jedna biblioteka przejmuje rolę drugiej. Dlatego też aplikacja będzie używać biblioteki Qt Quick wyłącznie w celach tworzenia interfejsu, obsługi zdarzeń oraz wyświetlania wyrenderowanego przez VTK obrazu. Za wczytywanie obrazów NRRD będzie odpowiedzialna biblioteka ITK, zaś za przetwarzanie wczytanych przez ITK obrazów oraz właściwe renderowanie scen odpowiedzialna będzie biblioteka VTK.

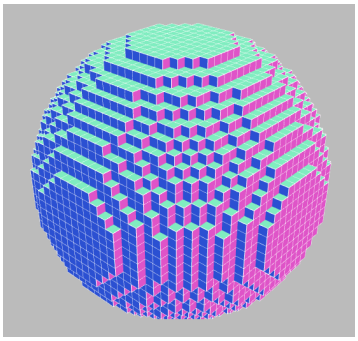
Przejdziemy teraz do omówienia terminologii użytej w pracy, aby móc w pełni zrozumieć poruszane w niej zagadnienia.

1.2. Terminologia

1.2.1. Woksel

Woksel to odpowiednik piksela w przstrzeni trójwymiarowej. Jest to najmniejszy, niepodzielny element obrazu, któremu możemy przypisać parametry niezależne od parametrów sąsiadów, takie jak kolor czy poziom przezroczystości.

Woksele na trójwymiarowej scenie reprezentowane są jako sześciany lub prostopadłościany, z których tworzony jest przestrzenny obraz.

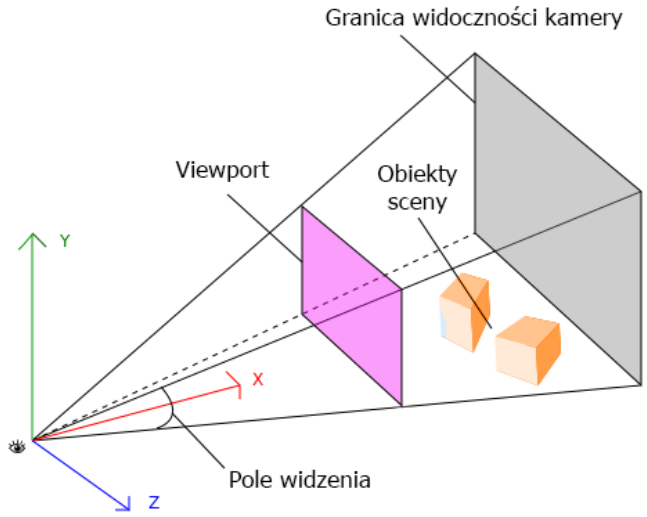


Rysunek 1.1: Przykładowy obiekt 3D zbudowany z wokseli.[1]

[1] https://i.stack.imgur.com/WANeM.png

1.2.2. Viewport

Viewport można rozumieć jako oko kamery. Jest to prostokątny obszar sceny, widziany przez kamerę. Kamera w grafice trójwymiarowej widzi podobnie do oka człowieka - posiada ograniczone pole widzenia w pionie i w poziomie, określające wymiary viewportu.



Rysunek 1.2: Model prezentujący działanie kamery w grafice trójwymiarowej.

1.2.3. GPU ray casting

GPU ray casting to technika renderowania obiektów 3D, polegająca na wysyłaniu przez kamerę promieni w kierunku viewportu, w celu wykrycia kolizji z obiektem na scenie, bądź jej braku. Promienie te są wysyłane pod różnymi kątami zmieniającymi się zarówno w poziomie jak i w pionie. Wysyłanie promieni można porównać do tworzenia obrazu w kineskopie - działo elektronowe w podobny sposób wysyła wiązki elektronów na ekran kineskopu. Każdy promień który wykryje kolizję z obiektem, niesie informację o położeniu, kolorze oraz przezroczystości fragmentu obiektu. Z kolei każdy promień, który nie przejdzie przez żaden obiekt, niesie informację o braku obiektu w danym pikselu obrazu. Skrót GPU w nazwie tej techniki informuje o fakcie, że obliczenia wykonywane są nie na procesorze komputera, a na procesorze karty graficznej.

1.2.4. Rendering wolumetryczny

Rendering wolumetryczny jest techniką tworzenia obrazu 2D na podstawie woluminów, czyli obiektów 3D utworzonych z warstwowo nałożonych na siebie obrazów 2D. Aby uzyskać wolumin, najpierw pozyskuje się odpowiednią liczbę obrazów 2D na różnych poziomach głębokości obiektu, następnie nakłada się te obrazy na siebie, tworząc obraz warstwowy, jednocześnie każdej warstwie nadając odpowiednią wysokość tak, by powstał obiekt przestrzenny.

1.2.5. Format NRRD

„Nearly raw raster data" (w skrócie NRRD) jest formatem zapisu obrazów używanym w nauce oraz medycynie. Pozwala na zapisanie uzyskanego obrazu w postaci n-wymiarowej tablicy zawierającej wartości intensywności pikseli. Dla potrzeb pracy będę korzystał z plików NRRD zawierających trójwymiarowe dane woluminów.

Przestawię teraz listę wykorzystanych w projekcie bibliotek oraz listę ich klas. Część z nich zostanie wykorzystana w rozdziale 2, przy implementacji komunikacji pomiędzy biblioteką Qt Quick i VTK, a pozostała część zostanie wykorzystana w rozdziale 3, gdzie będziemy implementować właściwą aplikację, jako że nie wszystkie opisane klasy są niezbędne do utworzenia poprawnej komunikacji pomiędzy omawianymi bibliotekami.

1.3. Opis wykorzystanych bibliotek oraz lista użytych klas

1.3.1. Visualization Toolkit (VTK)

Visualization Toolkit (w skrócie VTK) jest wieloplatformową biblioteką open-source służącą do generowania grafiki 3D, przetwarzania obrazów oraz wizualizacji danych przeznaczoną dla programów pisanych w języku C++, lecz posiadającą także swoją uboższą w możliwości implementację dla języków Tcl/Tk, Java oraz Python. Jednym z jej najczęstszych zastosowań jest wizualizacja danych medycznych, w szczególności rendering wolumetryczny tworzony z zarejestrowanych wcześniej obrazów techniką tomografii komputerowej bądź rezonansu magnetycznego.

Przedstawię teraz listę klas biblioteki VTK, których użyję w opisywanym projekcie:

Klasa vtkSmartPointer<T>

Jest to szablon klasy służącej to tworzenia tzw. „sprytnych" wskaźników. Ich spryt polega na tym, że utworzone za ich pomocą obiekty nie muszą być ręcznie usuwane przez programistę - biblioteka VTK będzie zliczała odwołania do tworzonych w ten sposób obiektów i kiedy ich liczba będzie równa 0, obiekty te zostaną automatycznie usunięte z pamięci. Parametr **T** w szablonie klasy oznacza typ obiektu, który chcemy stworzyć przy pomocy wskaźnika **vtkSmartPointer**.

Klasa vtkRenderWindow

Klasa ta reprezentuje okno w którym będziemy renderować obraz. Okno to można rozumieć na dwa sposoby - jako widoczny obiekt na ekranie w postaci prostokąta, w którym wyświetlany jest obraz, bądź - to co nas najbardziej interesuje - abstrakcyjny, niewidoczny obiekt w pamięci przechowujący dane o aktualnie wyrenderowanej scenie. Dane o wyrenderowanej scenie przechowywane są w postaci tablicy pikseli typu **unsigned char**, czyli przyjmują wartości od 0 do 255 i są zapisane w postaci RGBA, gdzie R - kanał koloru czerwonego, G - kanał koloru zielonego, B - kanał koloru niebieskiego, A - kanał przezroczystości.

Klasa vtkRenderer

Klasa ta odpowiedzialna jest za renderowanie obrazu. Poprzez renderowanie rozumie się tu rysowanie dwuwymiarowego obrazu przez procesor graficzny (GPU), powstałego poprzez prze-

twarzanie przez kartę graficzną danych matematycznych sceny 3D takich jak np. położenie wokseli w przestrzeni, odbicia i załamania światła czy dane bufora głębi.

Klasa vtkRenderWindowInteractor

Niniejsza klasa jest interfejsem komunikacyjnym pomiędzy użytkownikiem, a oknem biblioteki VTK, w którym wyświetlamy wyrenderowany obraz. Pozwala ona użytkownikowi na interakcję z programem poprzez użycie myszy lub klawiatury. W zależności od zastosowanego stylu interakcji, użytkownik ma możliwość np. sterować kamerą, wyświetlanymi obiektami na scenie, kontrastem i jasnością wyświetlanego obrazu, bądź przechodzenia między plastrami wyświetlanego obrazu.

Klasa vtkInteractorStyleTrackballCamera

Klasa ta reprezentuje styl interakcji używany w tym projekcie przy wyświetlaniu danych wygenerowanych techniką renderingu wolumetrycznego. Pozwala ona sterować obrotem oraz położeniem kamery.

Klasa vtkInteractorStyleImage

Z kolei ten styl interakcji wykorzystywany jest w oknach rednerujących obrazy warstwowe. Dzięki niej możemy przy pomocy myszy przechodzić między warstwami obrazu, bądź sterować jasnością oraz kontrastem wyświetlanego obrazu.

Klasa vtkOpenGLGPUVolumeRayCastMapper

Jest to klasa odpowiedzialna za renderowanie obrazu techniką GPU ray casting. Pobierając kolejne plastry ze wczytanego uprzednio pliku w formacie NRRD generuje trójwymiarowy, wolumetryczny obraz obiektu zarejestrowanego warstwowo na poszczególnych plastrach obrazu NRRD.

Klasa vtkColorTransferFunction

Klasa ta reprezentuje funkcję transferu kolorów, czyli funkcję która zamienia wartości intensywności wokseli na nowe wartości. Ze względu na fakt, że dane medyczne mają różny, często dużo większy od stosowanego w informatyce przedział intensywności kolorów, głównym celem stosowania tej funkcji jest przeskalowanie przedziału intensywności obrazu tak, by mieścił się w przedziale $[0; 255]$. Drugim celem stosowania niniejszej funkcji jest uwidocznienie konkretnych struktur w obrazie, takich jak np. kości, mięśnie czy naczynia krwionośne. Takie uwidocznienie możliwe jest dlatego, że każdy typ tkanki zarejestrowany jest z innym poziomem intensywności.

Klasa vtkPiecewiseFunction

Kolejna opisywana klasa ma funkcjonalność analogiczną do klasy **vtkColorTransferFunction**, z tym faktem, że konwertuje intensywności wokseli na poziomy ich przezroczystości. Służy w głównej mierze do ukrywania wokseli, których nie chcemy widzieć (np. wokseli tła, bądź tkanki, która nas w danym momencie nie interesuje), bądź do uczynienia wokseli półprzezroczystymi w celu polepszenia widoczności wgłąb renderowanego obiektu.

Klasa vtkVolumeProperty

Klasa zawierająca zestaw parametrów woluminu, takich jak opisywane powyżej funkcje transferu kolorów i funkcje przezroczystości, właściwości wyświetlanego materiału - poziom wpływu światła otoczenia (ambient), poziom rozpraszania światła (diffuse), czy poziom światła odbitego od obiektu (specular) oraz parametr określający czy obiekt ma być cieniowany - czy światło ma go oświetlać jednolicie, niezależnie od tego skąd i pod jakim kątem padają jego promienie.

Klasa vtkVolume

Klasa ta reprezentuje wygenerowany przez klasę **vtkOpenGLGPUVolumeRayCastMapper** oraz poddany przekształceniom przez klasę **vtkVolumeProperty** wolumin. Obiekty tej klasy będą już bezpośrednio dodawane do sceny renderera VTK w celu ich wyświetlenia w oknie renderującym.

Klasa vtkCommand

Klasa ta jest odpowiedzialna za obserwację obiektów VTK w celu wykrywania wywołania przez nie zdarzeń i reagowania na nie. Za jej pomocą możemy dla przykładu utworzyć wyspecjalizowaną klasę wykrywającą kliknięcie myszą na oknie renderującym i odpowiednio na nie reagującą (np. dodającą nowy obiekt na scenie w miejscu kliknięcia).

Klasa vtkImageReslice

Klasa odpowiedzialna za tworzenie warstw woluminu. Ze względu na to, że obrazowanie mogło zostać wykonane z dowolnych kierunków - z przodu lub z tyłu, z lewej lub z prawej strony, z góry lub z dołu, klasa ta umożliwia wybór osi wzdłuż której będziemy tworzyć warstwy obrazu. Możemy wybrać dowolną oś, jednak do naszych celów będziemy operować na osiach x, y i z układu współrzędnych, ustawiając uprzednio wczytany wolumin w centrum układu, by upewnić się, że każda z osi warstwowania będzie przechodzić przez środek woluminu.

Klasa vtkLookupTable

Jest to klasa reprezentująca tablicę transferu kolorów i przezroczystości. Odpowiednim wartościom intensywności pikseli w obrazie przypisuje odpowiednie kolory w formacie RGBA. Ta funkcja będzie używana do transferu kolorów w warstwach oglądanego obrazu.

Klasa vtkMatrix4x4

Jest to klasa reprezentująca macierz 4x4, przechowująca elementy typu double. Przy jej wykorzystaniu stworzymy macierz wyznaczającą oś wzdłuż której przeprowadzimy warstwowanie.

Klasa vtkImageMapToColors

Klasa ta używana jest do transferu kolorów i przezroczystości w obrazie 2D za pomocą klasy **vtkLookupTable**. Pobiera ona z obiektu klasy **vtkLookupTable** odpowiednią wartość koloru w formacie RGBA dla aktualnie przetwarzanej intensywności piksela i zapisuje nową wartość w obrazie w miejscu starej.

Klasa vtkImageActor

Jest to klasa reprezentująca gotowy do umieszczenia na scenie obraz 2D. Poprzez aktora w nazwie klasy rozumie się tu dowolny obiekt, który umieszczony został na scenie 3D lub 2D (analogia ta wywodzi się z terminologii teatralnej).

1.3.2. Insight Segmentation and Registration Toolkit (ITK)

Insight Segmentation and Registration Toolkit (w skrócie ITK) jest wieloplatformową biblioteką open-source służącą do rejestracji i segmentacji obrazów oraz szeroko rozumianej ich analizy i przetwarzania. Podobnie jak VTK, jej głównym zastosowaniem jest obrazowanie medyczne. Jest przeznaczona do pisania aplikacji w języku C++, lecz posiada także uboższą implementację dla języków Tcl, Java oraz Python. Powstała pod kierownictwem i za pośrednictwem funduszy Narodowej Biblioteki Medycznej Stanów Zjednoczonych[2].

Dla celów niniejszego projektu biblioteka ITK służyć będzie jedynie do odczytywania obrazów NRRD oraz przekazywania wczytanych danych do biblioteki VTK.

Oto klasy biblioteki ITK wykorzystywane w projekcie:

Klasa Image<TPixel, VImageDimension>

Jest to szablon klasy reprezentujący obraz. Poprzez obraz rozumie się tu n-wymiarową tablicę wartości pikseli. Liczbę wymiarów oraz typ wartości pikseli określają parametry szablonu:

- TPixel - typ piksela w obrazie. Najczęściej jest to typ **unsigned short** dla obrazów szarych i trzyelementowy wektor liczb typu **unsigned short** dla obrazów w formacie RGB.
- VImageDimension - parametr ten oznacza wymiar przestrzeni obrazu. Tak dla obrazów dwuwymiarowych będzie miał on wartość 2, a dla obrazów przestrzennych (w tym NRRD) - 3.

Klasa ImageFileReader<TOutputImage, ConvertPixelTraits>

Jest to szablon klasy służący do wczytywania obrazów z plików. W przypadku tego projektu posłuży do wczytania obrazów NRRD.

Parametry szablonu oznaczają kolejno:

- TOutputImage - typ wczytywanego obrazu - zazwyczaj odpowiednio wyspecjalizowany szablon klasy **Image**
- ConvertPixelTraits - funkcja konwertująca typ wczytanego obrazu na inny, wyspecjalizowany typ. W tym projekcie parametr ten przyjmie wartość domyślną, jako że nie będę używał żadnej funkcji konwersji.

Klasa ImageToVTKImageFilter<TInputImage>

Jest to szablon klasy pośredniczący między biblioteką ITK i VTK. Umożliwia skopiowanie danych obrazu wczytanych przez obiekt klasy **ImageFileReader** biblioteki ITK do obiektów

[2]https://itk.org/ITK/project/about.html

biblioteki VTK odpowiedzialnych za tworzenie i wyświetlanie obrazu w oknie renderującym.

Parametr **TInputImage** oznacza typ wczytanego obrazu i zazwyczaj jest to wyspecjalizowany typ szablonu klasy **Image**.

1.3.3. Qt Quick

Qt Quick jest biblioteką opartą o bibliotekę Qt. Zarówno Qt jak i Qt Quick to biblioteki służące do szeroko rozumianego tworzenia graficznego interfejsu użytkownika. W odróżnieniu od Qt, gdzie interfejs buduje się w języku XML, a logikę implementuje się w języku C++, w Qt Quick zarówno projektowanie interfejsu, jak i implementowanie jego logiki wykonuje się w specjalnie stworzonym na potrzeby Qt Quick języku QML opartym o język JavaScript.

Celem Qt Quick jest uproszczenie budowania dynamicznych i responsywnych graficznych interfejsów, poprzez zastosowanie języka QML, który ma zdecydowanie bardziej czytelną i prostszą składnię od języka XML. Dodatkową zaletą języka QML jest możliwość umieszczania logiki interfejsu bezpośrednio w obiektach definiujących interfejs, co zmniejsza konieczność stosowania zewnętrznych plików JavaScript i C++ do obsługi zdarzeń w programie.

W możliwościach biblioteka Qt Quick nie odbiega znacząco od innych bibliotek mających podobne zastosowanie. Możemy tworzyć w niej różnego rodzaju layouty, kontrolki, czy okna służące do wyświetlania i rysowania grafiki 2D i 3D.

W tym projekcie najbardziej interesuje nas klasa **QQuickPaintedItem**, reprezentująca okno służące do rysowania w nim dowolnych obrazów. Będzie to nasze właściwe okno renderujące, wyświetlające obraz pochodzący z renderera VTK.

1.3.4. OpenGL

Open Graphics Library (w skrócie OpenGL) to wieloplatformowa biblioteka służąca do tworzenia oraz renderowania grafiki 2D i 3D. Stanowi ona interfejs między programistą, a kartą graficzną, umożliwiając tworzenie uniwersalnych aplikacji graficznych, niezależnych od sprzętu zainstalowanego na danym komputerze.

OpenGL komunikuje się bezpośrednio z procesorem graficznym i wykonuje przy jego pomocy obliczenia służące do utworzenia sceny 2D lub 3D. W przypadku tego projektu odegra ona kluczową rolę przy renderowaniu trójwymiarowych woluminów, jako że te są renderowane przy użyciu procesora graficznego w oparciu o technikę GPU ray casting.

Generowanie i wyświetlanie obrazu odbywa się w tzw. potoku. W potoku dane sceny 3D są przetwarzane na obraz 2D, a następnie w razie potrzeby obraz 2D jest dodatkowo przetwarzany. Do przetwarzania scen 3D i gotowych obrazów 2D służą **shadery** - programy wykonywane na procesorze karty graficznej. Głównym zastosowaniem shaderów jest wykonywanie operacji na fragmentach i wierzchołkach siatki 3D (**fragment shader** i **vertex shader**) oraz na pikselach obrazu 2D (**pixel shader**). Za ich pomocą możemy manipulować wyświetlanymi obiektami jeszcze przed ich wyrednerowaniem oraz wyrenderowanymi obrazami 2D, przeprowadzając na nich np. filtrację.

Nie będę stosował tej biblioteki bezpośrednio w projekcie, jednak ze względu na to, że biblioteka VTK używa biblioteki OpenGL do tworzenia i wyświetlania obrazu, uznałem za stosowne zaznaczyć jej ważną rolę w realizacji mojego zadania.

Mając teraz jasny obraz celu jaki chcemy osiągnąć, problemów związanych z komunikacją bibliotek Qt Quick i VTK oraz listę narzędzi z jakich będziemy korzystać przy rozwiązywaniu tych problemów, możemy przejść do meritum - najpierw do utworzenia systemu komunikującego bibliotekę Qt Quick z VTK, a następnie do zaimplementowania aplikacji w oparciu o ten system.

W rozdziale 2 zostanie zaproponowane rozwiązanie problemu komunikacji omawianych bibliotek. Zaimplementowane zostaną trzy klasy, dzięki którym odbywać będzie się ta komunikacja - klasa **QQVTKViewport**, jako klasa obsługująca komunikację po stronie Qt Quick i implementująca jej funkcjonalność, klasa **QQVTKScene**, jako klasa obsługująca komunikację po stronie VTK, umożliwiająca jej klasom pochodnym na tworzenie dowolnego środowiska VTK, a także klasa **QQVTKAdapter**, komunikująca obie powyższe klasy ze sobą, stanowiąca most łączący interfejs aplikacji Qt Quick z logiką i tworzeniem scen za pomocą biblioteki VTK. Przy omawianiu klasy **QQVTKViewport** omówimy też proces rejestracji nowego typu w języku QML, utworzonego przez nas od strony języka C++, aby można było umieszczać obiekty zdefiniowanego przez nas typu w plikach QML, definiujących interfejs aplikacji.

Rozdział 3 będzie niejako kontynuacją rozdziału 2. Mając gotowy system komunikacji między omawianymi bibliotekami, przejdziemy do zaimplementowania na jego podstawie konkretnej aplikacji opartej na Qt Quick i VTK. Omówiony zostanie graficzny interfejs użytkownika, umożliwiający interakcję użytkownika z aplikacją, a w szczególności z biblioteką VTK. Zostaną omówione i zaimplementowane klasy stanowiące funkcjonalność aplikacji:

- Klasy **QQVTKSlicing** i **QQVTKVolumeRendering** - specjalizujące klasę **QQVTKScene** odpowiednio w wyświetlaniu i przeglądaniu warstw woluminów oraz wolumetrycznym renderowaniu tych woluminów z wykorzystaniem techniki GPU ray casting, funkcji transferu kolorów i przezroczystości;
- Klasa **QQVTKNRRDLoader** - wczytująca za pomocą obiektów klasy ITK plik NRRD i przekazująca wczytany obraz do biblioteki VTK;
- Klasa **VTKSlicingCommand** - klasa obsługująca zdarzenie zaszłe nad viewportem wyświetlającym warstwy woluminów, umożliwiająca przechodzenie między ich warstwami przy pomocy myszki;

Rozdział 2

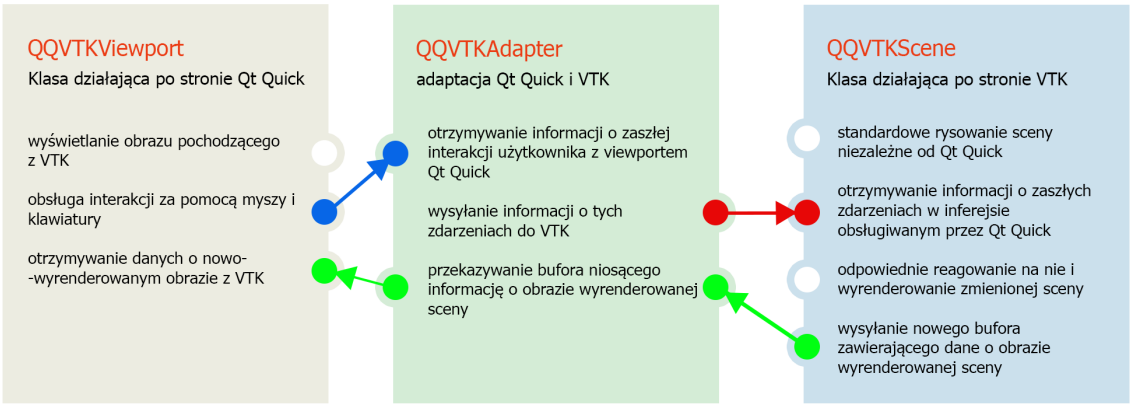
Komunikacja Qt Quick z VTK

Tak jak zostało to napisane w poprzednim rozdziale, aby poprawnie komunikować bibliotekę VTK z Qt Quick (i vice versa), należy utworzyć klasę adaptującą te dwie biblioteki. Powstaje teraz pytanie - co dokładnie chcemy adaptować? Każda z wykorzystywanych bibliotek musi zachować pełnię swoich możliwości, więc użycie klasy adaptacyjnej nie może wnieść żadnych ograniczeń na wykorzystywanie ich funkcjonalności. Przede wszystkim chcę użyć VTK do tego do czego został stworzony - do wizualizacji i przetwarzania obrazów. VTK będzie renderował przetwarzaną scenę w standardowy sposób - w swoim oknie renderującym, z tym faktem, że okno to będzie ukryte. Wyrenderowany obraz będzie następnie kopiowany do bufora przechowującego dane o pikselach wyrenderowanego obrazu znajdującego się w obiekcie Qt Quick odpowiedzialnym za wyświetlanie obrazu na ekranie. Na koniec obiekt ten wyświetli na ekranie obraz utworzony na podstawie przechowywanych w buforze danych o pikselach wyrenderowanego przez VTK obrazu.

W celu realizacji komunikacji między dwoma bibliotekami stworzę następujące klasy:

- **QQVTKViewport** - klasę dziedziczącą po klasie **QQuickPaintedItem**, działającą po stronie Qt Quick służącą do rysowania w oknie aplikacji obrazu powstałego poprzez odczytanie bufora pochodzącego z VTK, opisaną w rozdziale 2.1;
- **QQVTKScene** - klasę abstrakcyjną, działającą po stronie VTK odpowiedzialną za tworzenie niezbędnych obiektów od strony VTK, renderowanie sceny i udostępnianie dla biblioteki Qt Quick swojego okna renderującego w celu pobrania bufora z informacją o pikselach wyrenderowanego obrazu. Klasa ta nie będzie jednoznacznie definiować sposobu tworzenia scen oraz stylu interaktrora, w związku z tym, aby w pełni zaimplementować funkcjonalność aplikacji, w rozdziale 3 zostaną opisane klasy dziedzicące po niej - **QQVTKSlicing** i **QQVTKVolumeRendering** odpowiedzialne odpowiednio za wyświetlanie obrazów 2D z możliwością przechodzenia między warstwami woluminu oraz za rendering wolumetryczny tych woluminów, opisaną w rozdziale 2.2;
- **QQVTKAdapter** - klasę scalającą obie powyższe klasy w całość, umożliwiającą wzajemne komunikowanie się biblioteki Qt Quick i VTK - przekazywanie danych o wyrenderowanym obrazie oraz zdarzeniach zaszłych w wyniku interakcji użytkownika z aplikacją, opisaną w rozdziale 2.3;

Nazwy prawie wszystkich klas tworzonych na potrzeby aplikacji zaczynają się przedrostkiem QQVTK, gdzie QQ to skrót od Qt Quick, a VTK, to skrót od Visualization Toolkit.



Rysunek 2.1: Diagram prezentujący komunikację między klasami QQVTKViewport, QQVTKScene i QQVTKAdapter.

Przejdźmy teraz do zaimplementowania opisanych powyżej trzech klas, zaczynając od klasy **QQVTKViewport** zapewniającej komunikację od strony biblioteki Qt Quick.

2.1. Klasa QQVTKViewport - komunikacja po stronie biblioteki Qt Quick

Pierwsza z opisywanych klas będzie posiadała dwa kluczowe zadania. Pierwszym z nich będzie wyświetlanie wyrenderowanej przez VTK sceny - zdolność tę odziedziczy po klasie bazowej umożliwiającej rysowanie dowolnych obrazów w oknie aplikacji - klasie **QQuickPaintedItem**. Kolejnym kluczowym zadaniem z jakim będzie musiała zmierzyć się opisywana klasa będzie obsługa zdarzeń związanych z interakcją użytkownika z aplikacją za pomocą myszy oraz przekazywanie informacji o tych zdarzeniach do VTK poprzez klasę adaptującą.

Spójrzmy teraz na definicję klasy:

```
class QQVTKViewport : public QQuickPaintedItem {
  Q_OBJECT //Niezbędne definicje obiektu Qt

  //Parametry viewportu, którymi można sterować z poziomu GUI
  int brightness;
  double contrast;

  QSize pixelDataSize; //rozmiar okna renderującego
  unsigned char* pixelData; //bufor z danymi pikseli
  QQVTKAdapter* adapter; //adapter komunikujący klasę z VTK

//Funkcje dostępne z poziomu języka QML
public slots:
  //Zdarzenia wywoływane za pomocą myszy
  void onLeftDown(int, int);
  //Dalsza definicja zdarzeń...

  //Medoty komunikujące Qt Quick z VTK
  void updateScene(int);
  void requestDataUpdate();
  int GetCurrentColorFunction();
```

```
public:
  QQVTKViewport(QQuickItem *parent = 0);

  //Medoda odpowiedzialna za rysowanie w oknie renderującym Qt Quick
  void paint(QPainter*);
  //Metoda uaktualniająca dane w buforze pikseli
  void updateData(unsigned char*, int, int);
  void SetAdapter(QQVTKAdapter*);
};
```

Przypatrzmy się teraz kluczowym elementom tej klasy:

- `Q_OBJECT` - jest to makro niezbędne do utworzenia obiektu dziedziczącego po obiektach z biblioteki Qt. Za jego pomocą Qt tworzy niezbędne metody potrzebne do prawidłowego działania tych obiektów oraz do umożliwienia im komunikowania się z interfejsem napisanym w języku QML.

- `QSize pixelDataSize` - to pole zawierające informację o wielkości okna renderującego. Typ **QSize** zawiera metody **width(void)** i **height(void)** zwracające odpowiednio szerokość i wysokość tego okna.

- `unsigned char* pixelData` - bufor zawierający informację o pikselach wyrenderowanego przez VTK obrazu w formacie RGBA. Piksele zapisane są jako tablica jednowymiarowa ich wartości: najpierw zapisana jest składowa R pierwszego piksela obrazu, potem składowa G, składowa B, a na końcu składowa przezroczystości - A, następnie w identyczny sposób zapisywane są dane kolejnych pikseli. Ze względu na to, że tablica ta jest jednowymiarowa, koniecznym stało się utworzenie dodatkowej zmiennej pomocniczej, przechowującej informacje o wielkości okna renderującego - **pixelDataSize**.

- `QQVTKAdapter* adapter` - obiekt adaptujący viewport Qt Quick ze sceną renderowaną przy pomocy VTK. To za jego pomocą obie te biblioteki komunikują się ze sobą.

- `void onLeftDown(int, int)` - to metoda wywoływana przez Qt Quick w momencie kliknięcia myszą w oknie renderującym obraz. Parametry funkcji typu **int** oznaczają kolejno pozycję x i y myszy w oknie renderującym. Do obsługi zdarzeń związanych z myszą w programie zaimplementowałem cały zestaw metod, wywoływanych podczas kliknięcia i puszczenia dowolnego z przycisków myszy, a także metodę wywoływaną podczas ruchu myszą nad oknem renderującym, lecz metody te są zaimplementowane i działają w sposób analogiczny do metody **onLeftDown**, więc przedstawię tylko ją.

  Implementacja metody odpowiedzialnej za obsługę kliknięcia lewego przycisku myszy wygląda następująco:

```
void QQVTKViewport::onLeftDown(int x, int y) {
  if (!adapter)
    return;

  adapter->UpdateData(x, y, vtkCommand::LeftButtonPressEvent);
}
```

  Metoda **onLeftDown** poprzez obiekt adaptacyjny klasy **QQVTKAdapter** informuje obiekt klasy **QQVTKScene** odpowiedzialny za działanie VTK o tym, że zaszło zdarzenie konkretnego typu (w tym przypadku - kliknięcie lewym przyciskiem myszy).

Spójrzmy jeszcze na obsługę zdarzenia od strony QML, który w momencie kliknięcia lewym przyciskiem myszy wywołuje metodę **onLeftDown**, przekazując jej pozycję myszy w parametrach metody:

```
import QtQuick 2.10
import QQVTK 1.0

//Klasa reprezentująca okno w którym renderujemy obraz z VTK
QQVTKViewport {
  MouseArea {
    id: openGLMouseArea
    anchors.fill: parent

    //Przyciśnięto któryś z klawiszy myszy
    onPressed: {
      //Czy przyciśnięto lewy?
      if (mouse.button & Qt.LeftButton) {
        //Jeśli tak - wywołaj metodę onLeftDown
        parent.onLeftDown(mouse.x, mouse.y);
      }
    }
  }
}
```

- `void requestDataUpdate()` - to metoda działająca analogicznie do metody **onLeftDown**. Informuje ona VTK, że zaszło pewne zdarzenie, jednak nie konkretyzuje jakie to było zdarzenie. Zmusza to VTK do wyrenderowania nowej sceny, bez zmiany jej parametrów. Metoda ta wywoływana jest przy zmianie rozmiaru okna aplikacji, gdyż wtedy zmienia się rozmiar okien renderujących (viewportów), a co za tym idzie następuje konieczność wyrenderowania scen umieszczanych w viewportach o zmienionym rozmiarze. Jej implementacja jest identyczna do metod obsługujących zdarzenia związane z myszką, z tą różnicą, że trzecim parametrem metody **UpdateData** klasy **QQVTKAdapter** jest **vtkCommand::AnyEvent** - identyfikator nieskonkretyzowanego zdarzenia.

- `void paint(QPainter*)` - to medoda odpowiedzialna za rysowanie obrazu zapisanego w buforze **pixelData** w oknie renderującym Qt Quick.

Jej implementacja wygląda następująco:

```
void QQVTKViewport::paint(QPainter *painter) {
  if (!pixelData)
    return;

  brightness = QQmlProperty::read(this, "brightness").toInt();
  contrast = QQmlProperty::read(this, "contrast").toDouble();

  QImage img(pixelDataSize.width(), pixelDataSize.height(), QImage::
      Format_ARGB32);

  for (int i = pixelDataSize.height() - 1, a = 0; i >= 0; --i)
    for (int j = 0; j < pixelDataSize.width(); ++j, a += 4)
    {
      img.setPixel(j, i, qRgba(
        std::max(0, std::min<int>(pixelData[a] * contrast + brightness,
            255)),
        std::max(0, std::min<int>(pixelData[a + 1] * contrast +
            brightness, 255)),
```

```
            std::max(0, std::min<int>(pixelData[a + 2] * contrast +
                brightness, 255)),
            pixelData[a + 3]
        ));
    }

    QPoint p(0, 0);

    painter->drawImage(p, img);
}
```

Prześledźmy jej działanie:

1. Sprawdzamy, czy VTK wyrenderował już jakąś klatkę obrazu - jeśli nie, pole **pixelData** będzie miało wartość **nullptr** i metoda zakończy swoje działanie.
2. Pobieramy z interfejsu QML wartości reprezentujące jasność i kontrast renderowanego obrazu. Jeśli zostały zmienione przez użytkownika, za ich pomocą zmodyfikujemy odpowiednio wartości rysowanych pikseli, zmieniając kontrast i jasność obrazu.
3. Tworzymy zmienną typu **QImage** o nazwie **img**, reprezentującą obraz 2D, który zostanie narysowany w oknie renderującym Qt Quick. To w tym obrazie zostanie narysowany obraz pochodzący z VTK. Konstruktor klasy **QImage** pobiera w tym wypadku 3 argumenty: szerokość obrazu, jego wysokość oraz format pikseli - w naszym przypadku 32 bitowy format zapisu ARGB.
4. Przystępujemy teraz do właściwego narysowania obrazu z VTK w zmiennej **img**. W podwójnej pętli **for**, pobierając kolejne wartości pikseli rysujemy je w obrazie **img**, odpowiednio modyfikując wartości pikseli w zależności od wartości zapisanych w polach **brightness** oraz **contrast**. Kolorujemy kolejne piksele za pomocą metody **setPixel** klasy **QImage**, pozwalającej nam na wypełnienie każdego piksela w obrazie stosowną wartością w formacie RGBA.
5. Na koniec tworzymy pomocniczy punkt typu **QPoint** informujący, że narysowany obraz **img** chcemy umieścić w lewym górnym rogu okna renderującego, tak, by wypełnić cały jego obszar. I w końcu wywołaniem metody **drawImage** klasy **QPainter** rysujemy obraz w oknie renderującym Qt Quick.

Należy zaznaczyć tu ważną rzecz - wiersze pikseli obrazu pobranego z okna renderującego VTK rysowane są od dołu, ponieważ biblioteka VTK w taki właśnie sposób zapisuje informację o pikselach obrazu - zaczynając od lewego dolnego rogu, a kończąc na prawym górnym rogu obrazu, odwrotnie niż jest to robione zazwyczaj.

- `void updateData(unsigned char*, int, int);` - metoda ta odpowiedzialna jest za uaktualnianie danych w buforze zawierającym informację o pikselach obrazu. Wywoływana jest przez obiekt klasy **QQVTKScene** za pomocą adaptera, kiedy ten wyrenderuje nową scenę. Pierwszy parametr metody oznacza wskaźnik na tablicę wartości pikseli pobraną z VTK, drugi i trzeci parametr z kolei to odpowiednio szerokość i wysokość okna renderującego VTK:

```
void QQVTKViewport::updateData(unsigned char* newPixelData, int width,
    int height) {
    pixelData = newPixelData;
    pixelDataSize.setWidth(width);
    pixelDataSize.setHeight(height);
}
```

Pola i metody opisane powyżej to wszystko, czego potrzeba klasie **QQVTKViewport**, aby poprawnie komunikować się z biblioteką VTK. Jednak na potrzeby implementacji programu dodałem do tej klasy kilka innych, dodatkowych elementów, aby usprawnić tę komunikację jeszcze bardziej i wzbogacić funkcjonalność programu:

- `int brightness`, `int contrast` - są to pola, które decydują o jasności i kontraście renderowanego obrazu. Użytkownik może je dowolnie modyfikować przy pomocy interfejsu Qt Quick, a co za tym idzie modyfikować dla potrzeb poprawy widoczności wyświetlany obraz. Pole **brightness** domyślnie przyjmuje wartość 0, jako że jest to zmienna, którą dodaje się do każdej wartości składowych RGB piksela, z kolei **contrast** domyślnie przyjmuje wartość 1, jako że mnoży się ją z wartością składowych RGB każdego piksela.

- `void updateScene(int)` - to metoda służąca do informowania sceny VTK, że użytkownik wybrał inną funkcję transferu kolorów dla woluminów. Parametr **int** tej metody oznacza numer identyfikujący funkcję transferu. Do VTK wysyłana jest poprzez adapter informacja o typie funkcji transferu kolorów, a ten z kolei zmienia sposób wyświetlania wokseli i renderuje nową scenę, po czym wysyła nowe dane do bufora pikseli znajdującego się w obiekcie klasy **QQVTKViewport**.

- `int GetCurrentColorFunction()` - to metoda zwracająca identyfikator aktualnie wybranej przez użytkownika funkcji transferu kolorów. Użytkownik wybiera ją poprzez dostępną w oknie aplikacji rozwijalną listę, która zawiera nazwy funkcji i utożsamione z nimi identyfikatory, w postaci unikalnych liczb typu **int**.

Mamy zatem gotową klasę, która jest w stanie narysować nam obraz otrzymany od VTK w oknie aplikacji Qt Quick. Jednak aby móc korzystać z niego do utworzenia z niego viewportu w plikach QML definiujących interfejs, musimy najpierw zarejestrować nowy typ QML z poziomu funkcji **main** języka C++. Do rejestracji nowych typów w języku QML służy szablon funkcji globalnej biblioteki Qt - **qmlRegisterType<T>**. Parametr T oznacza nazwę klasy rejestrowanego typu, w naszym przypadku jest to **QQVTKViewport**.

Rejestracja naszego typu wygląda następująco:

```
qmlRegisterType<QQVTKViewport>("QQVTK", 1, 0, "QQVTKViewport");
```

Pierwszym parametrem funkcji jest nazwa modułu, jaki trzeba dołączyć w QML, aby uzyskać dostęp do zarejestrowanej klasy. Drugi i trzeci parametr mówią o wersji tego modułu - w naszym przypadku 1.0, zaś ostatni parametr, to nazwa rejestrowanego typu używana od strony QML, w naszym przypadku taka sama jak nazwa klasy od strony języka C++.

Zarejestrowany w powyższy sposób typ możemy już bezpośrednio używać w QML, budując z jego pomocą interfejs. Poniższy kod prezentuje przykład umieszczenia obiektu klasy **QQVTKViewport**, który otrzymuje od VTK wyrenderowany obraz i wyświetla go w oknie aplikacji:

```
import QtQuick 2.10
import QQVTK 1.0

//Główny panel interfejsu
Rectangle {
  width: 640
  height: 480
  color: "#000"
```

```
  //Nasz viewport wypełniający cały panel interfejsu
  QQVTKViewport {
    id: vtkViewport
    objectName: "vtkViewport" //nazwa dzięki której odwołamy się do
        viewportu z poziomu C++
    anchors.fill: parent
  }
}
```

Przejdźmy teraz do utworzenia klasy abstrakcyjnej implementującej działanie VTK - **QQVTKScene**.

2.2. Klasa QQVTKScene - komunikacja po stronie biblioteki VTK

Opisywana w tym podrozdziale klasa będzie odpowiedzialna za stworzenie w pełni działającego środowiska VTK. Klasa ta będzie tworzyła okno renderujące VTK, renderer służący do tworzenia sceny oraz jej renderowania, a także interaktor służący do reagowania na interakcję użytkownika z aplikacją przy pomocy myszy. Posłuży także do zdefiniowania stylu interaktora oraz sposobu tworzenia sceny 3D w klasach pochodnych - **QQVTKSlicing** i **QQVTKVolumeRendering**.

Spójrzmy teraz jak wygląda definicja klasy **QQVTKScene**:

```
class QQVTKScene {
protected:
  //Obiekt klasy odpowiedzialny za wczytywanie plików NRRD
  QQVTKNRRDLoader* nrrdLoader;
  vtkSmartPointer<vtkRenderer> renderer;
  vtkSmartPointer<vtkRenderWindow> renderWindow;
  vtkSmartPointer<vtkRenderWindowInteractor> interactor;

public:
  QQVTKScene();

  //Metoda tworząca scenę
  virtual void CreateScene() = 0;

  void SetNNRDLoader(QQVTKNRRDLoader*);
  vtkRenderWindow* GetRenderWindow();

  ~QQVTKScene();
};
```

Oto najważniejsze elementy tej klasy, niezbędne do zaimplementowania funkcjonalności biblioteki VTK:

- `vtkSmartPointer<vtkRenderer> renderer` - obiekt biblioteki VTK służący do tworzenia i renderowania scen 3D
- `vtkSmartPointer<vtkRenderWindow> renderWindow` - okno VTK, w którego buforze renderer będzie przechowywał dane o wyrenderowanej scenie. W normalnych warunkach - kiedy okno renderujące VTK jest widoczne - wyświetla się w nim wyrenderowana scena. Jednak w naszym przypadku, jako że okno to jest niewidoczne, będzie jedynie służyć do

przechowywania i udostępniania bufora zawierającego dane o pikselach wygenerowanego obrazu. Okno to ukrywa się w konstruktorze klasy **QQVTKScene** przy pomocy metody **SetOffScreenRendering(int)** z klasy **vtkRenderWindow**:

```
renderWindow->SetOffScreenRendering(1);
```

- `vtkSmartPointer<vtkRenderWindowInteractor> interactor` - interaktor VTK, czyli obiekt odpowiedzialny za reagowanie na interakcję użytkownika z aplikacją i odpowiednie zmienianie parametrów sceny w zależności o wywołanego przez użytkownika zdarzenia i zastosowanego stylu interaktora.
- `virtual void CreateScene()` - to metoda czysto wirtualna, która zostanie wyspecjalizowana w klasach pochodnych opisywanej klasy. Będzie ona definiowała sposób tworzenia przez VTK sceny, w zależności od wymagań określanych przez klasy pochodne.
- `vtkRenderWindow* GetRenderWindow()` - jest to metoda służąca do udostępniania innym obiektom okna renderującego VTK.

W klasie znajduje się również wskaźnik na typ **QQVTKNRRDLoader** oraz metoda przypisująca do tego wskaźnika nową wartość. Typ **QQVTKNRRDLoader** zostanie dokładnie opisany w rozdziale 3, ze względu na to, że nie jest niezbędny do poprawnej komunikacji biblioteki VTK z Qt Quick. Wspomnę tu tylko, że dzięki niemu i za pomocą biblioteki ITK odczytamy pliki NRRD i przekażemy wczytany obraz do biblioteki VTK, aby ta mogła go przetworzyć i dodać do generowanej sceny.

Implementacja klas **QQVTKSlicing** i **QQVTKVolumeRendering** specjalizujących działanie klasy **QQVTKScene** wykracza poza tematykę rozdziału drugiego, dlatego też zostanie opisana w kolejnym rozdziale, ze względu na to, że proces komunikacji bibliotek VTK i Qt Quick jest niezależny od sposobu w jaki VTK tworzy sceny oraz w jaki sposób reaguje na interakcję użytkownika z programem.

Przejdźmy teraz do opisania ostatniej z trzech klas używanych do łączenia bibliotek VTK i Qt Quick we wspólnie działającą całość. W ostatnim podrozdziale opiszę klasę adaptującą klasy **QQVTKViewport** i **QQVTKScene**.

2.3. Klasa QQVTKAdapter - adapter komunikujący VTK z Qt Quick

Zadanie tej klasy jest zarazem bardzo proste i niezwykle kluczowe - musi umożliwiać bibliotece Qt Quick przekazywanie do VTK informacji o zdarzeniach wywołanych poprzez interakcję użytkownika z aplikacją, a bibliotece VTK musi udostępnić sposób na przekazanie informacji o wyrenderowanym obrazie do biblioteki Qt Quick.

Spójrzmy więc jak wygląda definicja klasy **QQVTKAdapter**:

```
class QQVTKAdapter
{
  QQVTKViewport* viewport;
  QQVTKScene* scene;

public:
  QQVTKAdapter();
```

```
  void UpdateData(int, int, vtkCommand::EventIds);
  void UpdateScene(QQVTKVolumeRendering::ColorFunctions);
  void Adapt(QQVTKViewport*, QQVTKScene*);

  QQVTKScene* GetScene();

  ~QQVTKAdapter();
};
```

Przedstawię teraz elementy opisywanej klasy:

- `QQVTKViewport* viewport` - jest to wskaźnik na obiekt klasy działającej po stronie Qt Quick, viewport, w którym chcemy wyświetlać obraz otrzymywany z VTK.
- `QQVTKScene* scene` - to z kolei wskaźnik na obiekt klasy działającej po stronie VTK, który będzie przekazywał do **viewportu** wyrenderowany obraz sceny.
- `void UpdateData(int, int, vtkCommand::EventIds)` - to metoda odpowiedzialna za wywoływanie w interaktorze VTK zdarzeń związanych z interakcją użytkownika z programem za pomocą myszy, przekazanych przez bibliotekę Qt Quick. Pierwsze dwa parametry typu **int** to odpowiednio pozycja x i y myszy w oknie renderującym Qt Quick, pobrana w momencie zajścia zdarzenia, zaś drugi parametr informuje VTK o typie zdarzenia.

  Tak wygląda implementacja metody **UpdateData**:

```
void QQVTKAdapter::UpdateData(int x, int y, vtkCommand::EventIds command
    )
{
  int w = 640, h = 480;

  if (viewport->width() > 0)
    w = viewport->width();

  if (viewport->height() > 0)
    h = viewport->height();

  scene->GetRenderWindow()->GetInteractor()->SetEventInformation(x, y);

  switch (command)
  {
  case vtkCommand::LeftButtonPressEvent:
    scene->GetRenderWindow()->GetInteractor()->LeftButtonPressEvent();
    break;
  //Obsługa innych typów zdarzeń...
  }

  scene->GetRenderWindow()->SetSize(w, h);
  scene->GetRenderWindow()->Render();

  viewport->updateData(scene->GetRenderWindow()->GetRGBACharPixelData(0,
      0, w-1, h-1, 0), w, h);
  viewport->update();
}
```

Prześledźmy teraz działanie tej metody:

1. Na początku pobierana jest wielkość viewportu (okna renderującego) Qt Quick i zapisywana jest w zmiennych **w** i **h**.

2. Poprzez metodę **SetEventInformation** klasy **vtkRenderWindowInteractor** informujemy interaktora VTK o pozycji myszy w momencie zajścia zdarzenia.
3. Teraz adapter informuje interaktor o typie zdarzenia, by ten odpowiednio zmienił parametry sceny. W powyższej implementacji pokazałem wyłącznie wywoływanie zdarzenia dla naciśnięcia lewego przycisku myszy, gdyż obsługa innych zdarzeń związanych z interakcją użytkownika za pomocą myszy jest analogiczna.
4. Następnie za pomocą obiektu klasy **QQVTKScene** zmienia się rozmiar okna rednerującego VTK tak, aby dopasować go do rozmiaru viewportu Qt Quick oraz renderuje się obraz sceny zmienionej w wyniku nastąpienia zdarzenia.
5. W przedostatniej linijce pobierane są z okna renderującego VTK dane nowowyrenderowanego obrazu, a ostatnia linijka wywołuje metodę **update** klasy **QQuickPaintedItem** na rzecz viewportu, poprzez którą ten wyświetla wyrenderowany obraz otrzymany przed chwilą z VTK.

- `void Adapt(QQVTKViewport*, QQVTKScene*)` - zadaniem tej metody jest łączenie przekazanych jako jej parametry obiektów. W obiekcie klasy **QQVTKAdapter** zapamiętywane są obiekty, które chcą się ze sobą komunikować oraz w razie potrzeby informuje się te obiekty o tym, który obiekt jest ich adapterem (w tym przypadku informujemy tylko o tym obiekt **viewport**):

```
void QQVTKAdapter::Adapt(QQVTKViewport *viewport, QQVTKScene* scene) {
  this->viewport = viewport;
  this->scene = scene;

  viewport->SetAdapter(this);
}
```

- `QQVTKScene* GetScene()` - jest to metoda zwracająca wskaźnik na obiekt działający po stronie VTK, używana przez obiekt działający po stronie Qt Quick

- `void UpdateScene(QQVTKVolumeRendering::ColorFunctions)` - ostatnia z metod tej klasy służy do informowania VTK o zmianie funkcji transferu kolorów w aktywnym viewporcie. Szerzej ta funkcjonalność zostanie opisana w rozdziale 3.

Mając gotową implementację omawianych trzech klas, musimy wiedzieć jak je dokładnie wykorzystać. Poniższy kod prezentuje sposób utworzenia viewportu w Qt Quick, a także sceny w VTK, służącej do renderingu wolumetrycznego:

```
//Tworzymy okno interfejsu Qt Quick
QQuickView view;

//Wczytujemy do niego plik QML z dysku
view.setSource(QUrl::fromLocalFile(PROJECT_SOURCE_DIR "/main.qml"));

//Pobieramy główny obiekt interfejsu
QObject* root = view.rootObject();

//Tworzymy obiekt wczytujący pliki NRRD
QQVTKNRRDLoader nrrdLoader;

//Pobieramy zdefiniowany w pliku QML viewport:
QQVTKViewport* viewport3D_1 = root->findChild<QQVTKViewport*>("
    viewport3D_1");
```

```
//Przekazujemy do interfejsu w QML obiekt wczytujący pliki NRRD:
view.rootContext()->setContextProperty("nrrdLoader", &nrrdLoader);

//Tworzymy scenę VTK i przekazujemy jej obiekt wczytujący pliki NRRD:
QQVTKScene* volumeRendering = new QQVTKVolumeRendering();
volumeRendering->SetNNRDLoader(&nrrdLoader);

//Tworzymy adapter:
QQVTKAdapter adapter;

//Informujemy obiekt wczytujący pliki NRRD o tym, które obiekty ma
    poinformować w momencie wczytania pliku
nrrdLoader.ToNotifyOnLoaded(volumeRendering);
nrrdLoader.ToNotifyOnLoaded(viewport3D_1);

//Jeśli w interfejsie QML jest zdefiniowany viewport:
if (viewport3D_1)
  adapter.Adapt(viewport3D_1, volumeRendering); //Adaptujemy
```

Powyższy kod działa następująco:

1. Na początku tworzymy interfejs QML poprzez wczytanie pliku QML znajdującego się na dysku, zawierającego zdefiniowany interfejs z viewportem.

2. Teraz pobieramy z interfejsu zdefiniowany viewport, który ma identyfikator „viewport3D_1”.

3. W dalszej kolejności tworzymy obiekt klasy **QQVTKNRRDLoader**, służący do wczytywania plików NRRD i informujemy interfejs w QML o tym, który obiekt jest odpowiedzialny za wczytywanie plików.

4. Następnie tworzymy obiekt adaptera.

5. Informujemy obiekt **nrrdLoader** o tym, które obiekty musi poinformować o zakończeniu wczytywania pliku NRRD. Dzięki temu, gdy ten wczyta plik, VTK od razu przetworzy wolumin znajdujący się w pliku i doda go do swojej sceny, a Qt Quick wyświetli tę wyrenderowaną scenę.

6. Na koniec, jeśli wszystko poszło dobrze i w interfejsie faktycznie jest zdefiniowany viewport o podanym id, adaptujemy scenę VTK i viewport Qt Quick. Od tego momentu rozpoczyna się komunikacja.

Opis klasy **QQVTKAdapter** kończy rozdział poruszający tematykę komunikacji bibliotek VTK i Qt Quick. Za pomocą zaimplementowanych klas jesteśmy w stanie stworzyć środowisko, które bez przeszkód będzie mogło wzajemnie się komunikować. Zdecydowaną zaletą opisanego w niniejszym rozdziale rozwiązania jest fakt, że biblioteki Qt Quick i VTK pozostają niezależne od siebie i żadna z nich nie ogranicza funkcjonalności drugiej. Zarówno interfejs w Qt Quick jak i funkcjonalność biblioteki VTK mogą być budowane dowolnie - jedyne ograniczenie stanowią tu tylko możliwości poszczególnych bibliotek.

Warto zastanowić się też na wydajnością zaproponowanego przez mnie rozwiązania. Ta z pewnością jest niższa niż przy użyciu samej biblioteki VTK lub Qt Quick, z tego względu, że obraz renderowany jest w dwóch miejscach - w oknie renderującym VTK i w viewporcie Qt Quick. Jednak warto zauważyć, że obraz renderowany w Qt Quick jest nieskomplikowany -

jedyną operacją, którą wykonujemy jest tworzenie obrazu 2D, kopiowanie doń pikseli pochodzących z wyrenderowanego przez VTK obrazu i w końcu wyświetlanie go w oknie Qt Quick. Z drugiej strony zaletą VTK jest fakt, że sceny w nim są renderowane tylko, jeśli nastąpiły w nich jakiekolwiek zmiany - np. użytkownik zażyczył sobie zmienić funkcję transferu kolorów lub poruszył kamerą. W przeciwnym wypadku renderer VTK pozostaje uśpiony, oczekując na ewentualne zdarzenia otrzymane od Qt Quick poprzez klasę adaptującą.

Mamy zatem gotowy system komunikacji biblioteki Qt Quick z VTK. Nic nie stoi już na przeszkodzie, aby implementować w oparciu o ten system dowolne aplikacje wykorzystujące omawiane biblioteki. W następnym rozdziale omówiona zostanie i zaimplementowana aplikacja służąca do wczytywania i wyświetlania woluminów zapisanych w formacie NRRD, wykorzystująca interfejs zbudowany w oparciu o Qt Quick oraz funkcjonalność VTK i ITK, umożliwiającą wczytywanie, przetwarzanie i wyświetlanie wczytanych danych wolumetrycznych.

Rozdział 3

Implementacja aplikacji do renderingu wolumetrycznego

Wykorzystajmy zatem nasz system komunikacji, tworząc na jego podstawie aplikację umożliwiającą wczytywanie danych wolumetrycznych i renderowanie ich przy pomocy techniki GPU ray casting. W pierwszym podrozdziale omówiona zostanie dokładna funkcjonalność aplikacji. W rozdziale 3.2 przyjrzymy się zaprojektowanemu przez mnie interfejsowi użytkownika, a w następnych podrozdziałach przejdziemy do implementacji poszczególnych klas projektu.

3.1. Funkcje aplikacji

Przedstawię teraz dokładną funkcjonalność aplikacji w odniesieniu do wykorzystanych bibliotek - Qt Quick, ITK i VTK.

Funkcjonalność stworzona przy pomocy biblioteki ITK:

- Wczytywanie woluminów zapisanych w formacie NRRD;
- Przekazywanie wczytanych obrazów do VTK w celu dodania ich do sceny, przetworzenia i wyświetlenia w oknie renderującym;

Funkcjonalność stworzona przy pomocy biblioteki VTK, zaimplementowana w klasie **QQVTKScene** i skonkretyzowana w klasach **QQVTKSlicing** i **QQVTKVolumeRendering**, opisanych odpowiednio w podrozdziałach 3.4 i 3.5:

- Tworzenie i renderowanie scen przy pomocy techniki GPU ray casting;
- Przekazywanie do Qt Quick poprzez klasę adaptującą wyrenderowanego obrazu;
- Tworzenie interaktora oraz stylu interakcji;
- Odpowiednie reagowanie na interakcję użytkownika z aplikacją w zależności od zastosowanego stylu interakcji;
- Przetwarzanie wyświetlanych woluminów - w szczególności wykonywanie funkcji transferu kolorów wokseli oraz przechodzenie pomiędzy warstwami woluminu;

Funkcjonalność stworzona przy pomocy biblioteki Qt Quick, zaimplementowana w klasie **QQVTKViewport**, opisanej w rozdziale 2.1:

- Stworzenie graficznego interfejsu użytkownika pozwalającego na wyświetlanie scen generowanych przez VTK;
- Umożliwienie użytkownikowi na wczytywanie plików NRRD oraz ustawianie parametrów viewportów, takich jak kolor tła, jasność i kontrast wyświetlanego obrazu, czy wybór funkcji transferu kolorów wokseli;
- Umożliwienie użytkownikowi na interakcję z wyświetlaną sceną za pomocą myszki i klawiatury poprzez odbieranie zdarzeń i przekazywanie ich poprzez klasę adaptującą do VTK;

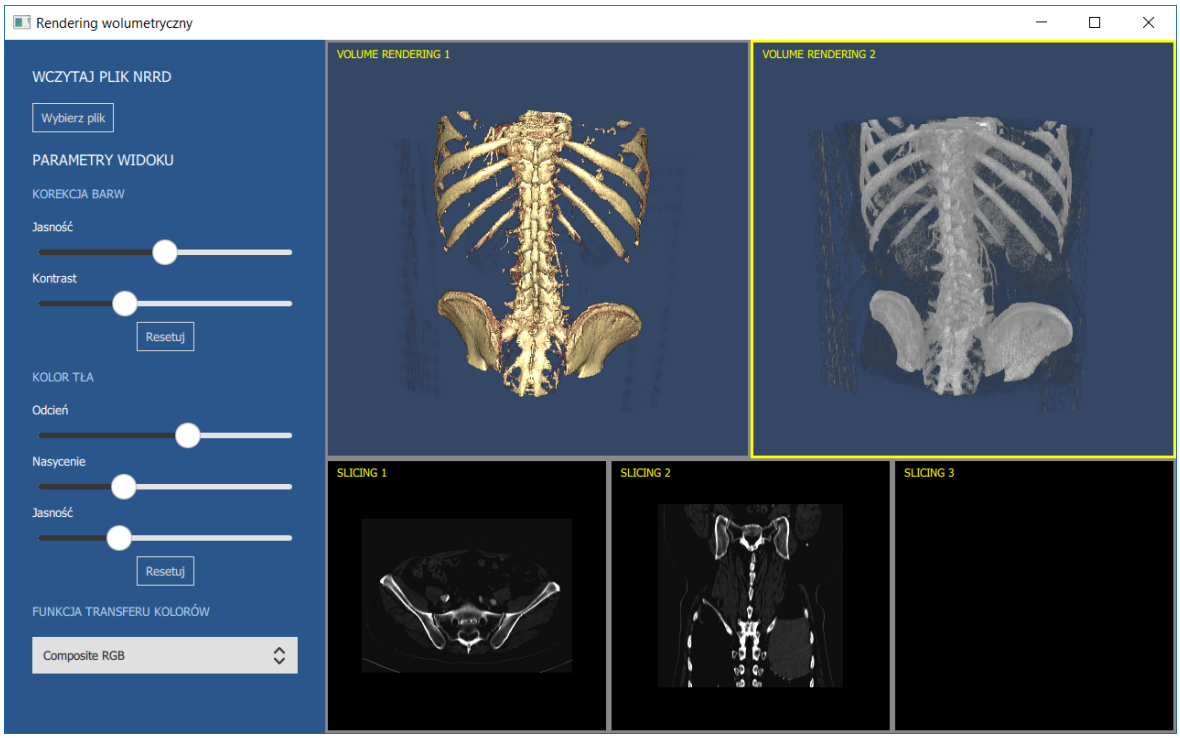
Przestawię teraz listę klas programu, które będą implementować opisaną powyżej funkcjonalność. Implementacja pierwszych trzech klas została opisana w poprzednim rozdziale, dlatego w tym rozdziale zajmiemy się implementowaniem pozostałych klas aplikacji:

- **QQVTKViewport** - klasa umożliwiająca obsługę komunikacji po stronie Qt (patrz: rozdział 2.1);
- **QQVTKScene** - abstrakcyjna klasa umożliwiająca obsługę komunikacji po stronie VTK (patrz: rozdział 2.2);
- **QQVTKAdapter** - adapter komunikujący VTK z Qt Quick (patrz: rozdział 2.3);
- **QQVTKNRRDLoader** - klasa umożliwiająca wczytywanie plików w formacie NRRD oraz przekazująca informację do obiektów klas **QQVTKScene** i **QQVTKViewport** o zakończeniu wczytywania pliku, opisana w rozdziale 3.3;
- **QQVTKSlicing** - klasa pochodna klasy **QQVTKScene**, dodająca funkcjonalność umożliwiającą wyświetlanie warstw woluminu oraz przechodzenie pomiędzy nimi, opisana w rozdziale 3.4;
- **QQVTKVolumeRendering** - klasa pochodna klasy **QQVTKScene**, dodająca do niej możliwość wyświetlania woluminów, używania funkcji transferu kolorów oraz funkcji transferu przezroczystości, a także renderowania tych woluminów za pomocą techniki GPU ray casting, opisana w rozdziale 3.5;
- **VTKSlicingCommand** - klasa dziedzicząca po klasie **vtkCommand**, specjalizująca ją w przechodzeniu pomiędzy warstwami woluminów. Będzie ona wykorzystywana przy wywoływaniu zdarzeń związanych z interakcją użytkownika za pomocą myszy, zaszłych nad viewportem specjalizującym się w wyświetlaniu warstw woluminu, opisana w rozdziale 3.6;

Zanim przejdziemy do implementacji powyższych klas, spójrzmy na zaprojektowany przeze mnie interfejs aplikacji. Da nam to jasny obraz celu, w jakim będziemy implementować powyższe klasy, a także wyjaśni zależności między interakcją użytkownika poprzez bibliotekę Qt Quick i biblioteką VTK.

3.2. Graficzny interfejs użytkownika

Spójrzmy na początek, jak prezentuje się aplikacja:



Rysunek 3.1: Okno implementowanej aplikacji.

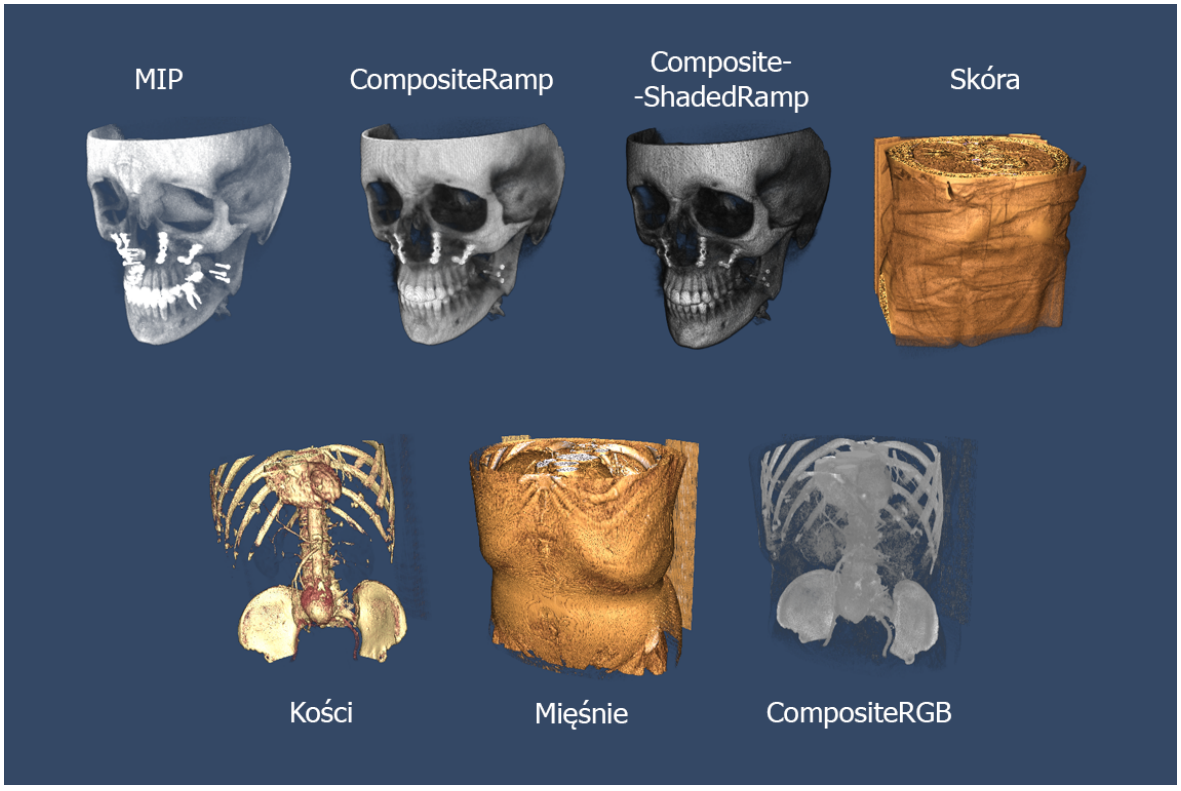
Interfejs aplikacji został podzielony na dwa główne obszary: lewy panel służący do sterowania parametrami viewportu i wczytywania plików NRRD oraz prawy panel, służący do wyświetlania scen generowanych przez VTK.

Oto opis elementów lewego panelu:

- Przycisk służący do wczytywania plików NRRD. Po jego wciśnięciu otwierane jest okno dialogowe, służące do wyboru pliku, który chcemy otworzyć. Gdy plik zostanie wybrany i zostanie poprawnie wczytany, jego zawartość wyświetli się we wszystkich viewportach znajdujących się na prawym panelu;
- Suwaki służące do sterowania jasnością i kontrastem aktywnego viewportu oraz przycisk do ich resetowania.
- Suwaki do sterowania kolorem tła aktywnego viewportu oraz przycisk służący do resetowania ich wartości. Kolor określany jest na podstawie wartości odcienia, nasycenia oraz jasności zaczerpniętych bezpośrednio z przestrzeni barw HSL;
- Rozwijalna lista służąca do wyboru funkcji transferu kolorów. Jest ona dostępna tylko dla górnych viewportów, wyświetlających woluminy generowane metodą GPU ray casting. Dostępne funkcje transferu kolorów prezentują się następująco:
  - **Maximun Intensity Projection (MIP)** - to funkcja która wyświetla wolumin na podstawie intensywności wokseli. Każdy promień przechodzący przez wolumin

znajduje woksel o największej intensywności i taki też woksel jest potem wyświetlany, niezależnie od tego na jakiej głębokości obiektu się znajduje.

- **CompositeRamp** - funkcja ta transferuje kolory wokseli za pomocą liniowej funkcji, której wartości reprezentują kolory od czarnego do białego. Wyświetlanie woluminu przy pomocy tej funkcji najczęściej odpowiada jego reprezentacji na warstwach zarejestrowanego obrazu - obszary ciemniejsze w warstwach są wyświetlane jako ciemniejsze, zaś te, które są bardziej widoczne, jako jaśniejsze;
- **CompositeShadeRamp** - funkcja transferu podobna do powyższej z tą różnicą, że włączone jest cieniowanie - światło nie oświetla już obiektu jednolicie, lecz jasność padającego światła na poszczególne woksele zależna jest od kąta jego padania;
- **Skóra** - jest to funkcja służąca do uwidoczniania tkanek budujących skórę i ukrywania pozostałych tkanek;
- **Kości** - jest to funkcja służąca do uwidoczniania tkanek budujących kości i ukrywania pozostałych tkanek;
- **Mięśnie** - jest to funkcja służąca do uwidoczniania tkanek budujących mięśnie i ukrywania pozostałych tkanek;
- C**ompositeRGB** - jest to funkcja używana do uwidocznienia struktur zarejestrowanych w formacie RGB - uwidoczniane są woksele o intensywnym kolorze;



Rysunek 3.2: Zestawienie funkcji transferu kolorów.

Spójrzmy teraz na elementy prawego panelu aplikacji, w którym wyświetlane są sceny pochodzące z VTK:

- Dwa górne viewporty służą do wyświetlania woluminów przetworzonych przez VTK na podstawie wybranej funkcji kolorów. Umożliwiają one sterowanie kamerą sceny za pomocą myszki;

- Trzy viewporty znajdujące się na dole prawego panelu służą do wyświetlania warstw woluminu i przechodzenia pomiędzy nimi. Każdy z trzech viewportów służy do wyświetlania warstw woluminu zarejestrowanych z różnych stron - z boku obiektu, z góry lub z dołu i z przodu lub z tyłu. Kolejność stron obiektu wyświetlanych przez te viewporty może się różnić, w zależności od sposobu obrazowania obiektu i jego zapisu w pliku NRRD, jednak najczęściej lewy viewport wyświetla warstwy obiektu widziane z góry, środkowy z przodu lub z tyłu, a prawy z boku obiektu;

- Wyświetlaniem viewportów można sterować za pomocą myszy i klawiatury. Za pomocą myszy wybieramy aktywny viewport, klikając na niego lewym przyciskiem myszy. Aktywny viewport zaznaczony jest żółtym obramowaniem. Po uprzednim wybraniu viewportu możemy ukrywać go za pomocą przycisku **H** na klawiaturze, wyświetlać go na pełnym ekranie za pomocą przycisku **W** oraz pokazywać z powrotem wszystkie viewporty za pomocą przycisku **G**;

Zobrazowawszy sobie wygląd i funkcjonalność aplikacji, przejdziemy teraz do kolejnych podrozdziałów, gdzie zaimplementujemy klasy, które dadzą nam tę funkcjonalność.

3.3. Klasa QQVTKNRRDLoader - klasa odpowiedzialna za wczytywanie plików NRRD

W pierwszej kolejności zaimplementujemy klasę umożliwiającą wczytywanie plików NRRD. Dzięki opisanym w rozdziale 1.3 klasom biblioteki ITK odczytamy plik NRRD z dysku, a następnie prześlemy dane o wczytanym obrazie do biblioteki VTK, aby ta mogła tworzyć sceny zawierające wczytany wolumin. Za pomocą tej klasy poinformujemy również obiekty klasy **QQVTKScene** o tym, że wczytano nowy wolumin i należy dodać go do sceny oraz obiekty klasy **QQVTKViewport** o tym, że VTK utowrzył i wyrenderował nową scenę i należy ją wyświetlić. Zacznijmy od wypisania wykorzystywanych szablonów klas z biblioteki ITK oraz zdefiniowania typów za pomocą których wyspecjalizujemy te szablony, tworząc za ich pomocą obiekty służące do odczytania plików NRRD:

- **itk::Image < itk::Vector<int, 1>, 3 >** - typ obiektu reprezentującego obraz NRRD. Pierwszym parametrem szablonu klasy **Image** jest typ pikseli w obrazie - **itk::Vector<int, 1>**. W naszym wypadku oznacza to, że piksele zawierają jedną wartość typu **int**, mianowicie poziom swojej intensywności. Z kolei drugim parametrem szablonu tej klasy jest wymiar wczytywanego obrazu. W naszym przypadku wynosi on 3, jako że zapisane w pliku NRRD woluminy są trójwymiarowe;

- **itk::ImageFileReader < itk::Image <itk::Vector<int, 1>, 3> >** - typ obiektu odpowiedzialnego za wczytywanie plików NRRD. Szablon ten pobiera jeden parametr - typ wczytywanego obrazu. W naszym wypadku jest on taki sam jak opisany powyżej typ obrazu NRRD, mianowicie **itk::Image<itk::Vector<int, 1>, 3>**;

- **itk::ImageToVTKImageFilter < itk::Image<itk::Vector<int, 1>, 3> >** - typ obiektu odpowiedzialnego za przekazywanie przez ITK danych o wczytanym obrazie do

VTK. Szablon ten, podobnie jak szablon klasy **ImageFileReader** pobiera jeden parametr - typ obrazu. W naszym wypadku jest to rzecz jasna **itk::Image<itk::Vector<int, 1>, 3>**;

Wyjaśnienie znaczenia typów jakie przyjmują obiekty ITK było niezbędne, aby móc w pełni zrozumieć implementację omawianej klasy. Mając teraz jaśniejszy obraz tego, w jaki sposób tworzone są obiekty z biblioteki ITK, spójrzmy na definicję klasy **QQVTKNRRDLoader**:

```
class QQVTKNRRDLoader : public QObject
{
  Q_OBJECT //Niezbędne definicje Qt

private:
  //Obiekt odczytujący plik NRRD
  itk::ImageFileReader<itk::Image<itk::Vector<int, 1>, 3>>::Pointer reader
      ;

  //Obiekt wysyłający dane o obrazie do VTK
  itk::ImageToVTKImageFilter<itk::Image<itk::Vector<int, 1>, 3>>::Pointer
      vtkConnector;
  std::string filename; //Nazwa pliku NRRD
  bool isLoaded; //Czy plik NRRD został załadowany?
  //Lista scen VTK, które otrzymają informację o tym, że wczytano nowy
      plik NRRD
  QList<QQVTKScene*> scenesToNotify;
  //Lista viewportów Qt Quick, które otrzymają informację o tym, że
      wczytano nowy plik NRRD
  QList<QQVTKViewport*> viewportsToNotify;


public:
  QQVTKNRRDLoader();
  void ToNotifyOnLoaded(QQVTKScene*);
  void ToNotifyOnLoaded(QQVTKViewport*);
  void NotifyOnLoaded(); //Powiadamianie o wczytaniu pliku NRRD
  Q_INVOKABLE void Load(QString);
  bool IsLoaded();
  itk::ImageToVTKImageFilter<itk::Image<itk::Vector<int, 1>, 3>>::Pointer
      GetVTKConnector();

  ~QQVTKNRRDLoader();
};
```

Wyjaśnię teraz dokładne znaczenie elementów klasy **QQVTKNRRDLoader**:

- `Q_OBJECT` - jest to niezbędne makro służące do tworzenia obiektów dziedziczących po klasach z biblioteki Qt (patrz: rozdział 2.1). Makro to jest niezbędne, ze względu na to, że metoda **Load** będzie wywoływana z poziomu interfejsu zdefiniowanego w plikach QML;

- `itk::ImageFileReader<itk::Image<itk::Vector<int, 1>, 3>>::Pointer reader` - obiekt klasy **Image** służący do odczytywania plików NRRD;

- `itk::ImageToVTKImageFilter<itk::Image<itk::Vector<int, 1>, 3>>::Pointer` - obiekt klasy **ImageToVTKImageFilter** umożliwiający przesyłanie danych wczytanego przez ITK obrazu do VTK;

- `std::string filename` - nazwa wczytanego pliku NRRD;
- `bool isLoaded` - pole klasy przechowujące informację o tym, czy jakikolwiek plik NRRD został wczytany;
- `QList<QQVTKScene*> scenesToNotify` - lista obiektów klasy **QQVTKScene**, które powinny zostać poinformowane o tym, że wczytano plik NRRD. Umożliwi im to natychmiastową reakcję na wczytanie pliku, a więc przetworzenie woluminu i utworzenie nowej sceny zawierającej ten wolumin;
- `QList<QQVTKViewport*> viewportsToNotify` - jest to lista obiektów klasy **QQVTKViewport**, które powinny zostać poinformowane o tym, że wczytano plik NRRD. Dzięki temu od razu po wczytaniu pliku i dodaniu go przez VTK do sceny zostanie on narysowany przez Qt Quick w oknie aplikacji;
- `void ToNotifyOnLoaded(QQVTKScene*)`, `void ToNotifyOnLoaded(QQVTKViewport*)` - metody dodające do opisanych powyżej list obiekty działające odpowiednio po stronie VTK i Qt Quick;
- `void NotifyOnLoaded()` - metoda wywoływana po wczytaniu pliku NRRD informująca obiekty klasy **QQVTKScene** i **QQVTKViewport** o zakończeniu wczytywania pliku. Wywołuje ona na rzecz każdego obiektu typu **QQVTKScene** metodę tworzącą scenę zawierającą nowy wolumin oraz na rzecz każdego obiektu typu **QQVTKViewport** metodę, dzięki której Qt Quick pozyskuje dane o wyrenderowanej przez VTK scenie zawierającej nowo wczytany wolumin i rysuje na ekranie otrzymany od VTK obraz.

  Spójrzmy na implementację tej metody:

```
void QQVTKNRRDLoader::NotifyOnLoaded()
{
  for (QQVTKScene* scene : scenesToNotify)
    scene->CreateScene();

  for (QQVTKViewport* viewport : viewportsToNotify)
    viewport->requestDataUpdate();
}
```

- `Q_INVOKABLE void Load(QString)` - metoda wywoływana z poziomu języka QML służąca do wczytywania pliku NRRD. Scieżka do pliku przekazywana jest jako parametr tej metody. Makro `Q_INVOKABLE` oznacza właśnie fakt, że metoda ta wywoływana jest z poziomu języka QML.

  Oto implementacja tej metody:

```
Q_INVOKABLE void QQVTKNRRDLoader::Load(QString filename)
{
  this->filename = filename.toStdString();
  reader->SetFileName(filename.toStdString().c_str()); //Przekazujemy
      readerowi ścieżkę do pliku

  try {
    reader->Update(); //Wczytujemy plik
    //Przekazujemy dane do VTK
    vtkConnector->SetInput(reader->GetOutput());
    vtkConnector->Update();

    isLoaded = true; //Zaznaczamy, że udało się wczytać plik
```

```
        NotifyOnLoaded(); //Informaujemy o tym zdarzeniu niezbędne obiekty
            po stronie Qt Quick i VTK
    }
    catch (itk::ExceptionObject) {
      //Nastąpił problem przy wczytywaniu pliku NRRD
    }
}
```

Lista zadań wykonywanych przez powyższą metodę jest następująca:

1. Przekazanie do obiektu wczytującego ścieżki do pliku przekazaną z poziomu języka QML.
2. Wczytanie pliku i przekazanie go do VTK. Jeśli wczytywanie pliku nie powiedzie się, zostanie rzucony wyjątek typu **ExceptionObject** z biblioteki ITK i metoda zakończy swoje działanie.
3. Jeśli zaś plik zostanie poprawnie wczytany, metoda ta poinformuje obiekty działające po stronie VTK i Qt Quick o poprawnym wczytaniu pliku NRRD poprzez wywołanie metody **NotifyOnLoaded**. Dzięki niej na ekranie pojawi się nowo wygenerowana scena pochodząca z VTK.

- `bool IsLoaded()` - metoda zwracająca informację o tym, czy plik NRRD został wczytany;
- `itk::ImageToVTKImageFilter<itk::Image<itk::Vector<int, 1>, 3>>::Pointer GetVTKConnector()` - metoda zwracająca wskaźnik na obiekt klasy **ImageToVTKImageFilter** z biblioteki ITK przekazujący do biblioteki VTK wczytany plik;

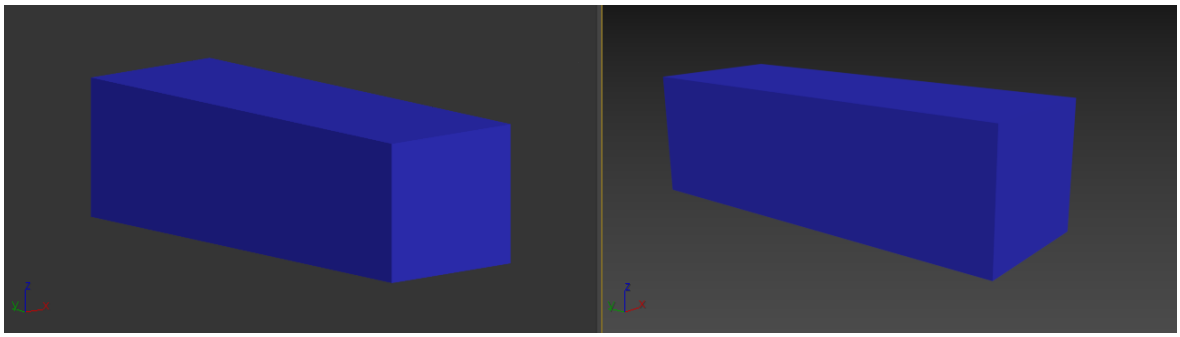
Implementując klasę **QQVTKNRRDLoader** stworzyliśmy sposób na wczytywanie przez naszą aplikację plików NRRD. Teraz nie tylko jesteśmy w stanie wczytywać pliki w tym formacie, ale przede wszystkim jesteśmy w stanie przekazać je do biblioteki VTK, aby ta zajęła się ich przetwarzaniem oraz generowaniem scen z ich użyciem. W następnych dwóch podrozdziałach zajmiemy się implementowaniem klas odpowiedzialnych właśnie za przetwarzanie wczytanych plików NRRD oraz dodawanie zawartych w nich woluminów na scenę.

3.4. Klasa QQVTKSlicing - klasa odpowiedzialna za wyświetlanie warstw woluminu

Jak zostało to opisane w rozdziale 1 - wolumin składa się z warstwowo nałożonych na siebie obrazów 2D uzyskanych np. za pomocą metody rezonansu magnetycznego lub tomografii komputerowej. Obrazowanie wykonuje się na różnych głębokościach obiektu, tak aby uzyskać warstwy woluminu, które następnie nakłada się na siebie, uprzednio nadając im odpowiednią wysokość, aby utworzyć obiekt przestrzenny.

Klasa **QQVTKSlicing** umożliwi wyświetlanie warstw woluminu, a także przechodzenie między nimi za pomocą myszy. Obrazowanie obiektu może być wykonane z dowolnej ze stron, jednak nas będą interesować obrazowania wykonane wzdłuż osi x, y i z trójwymiarowego układu współrzędnych. Oznacza to, że zakładać będziemy, że obrazowanie zostało wykonane z boku obiektu, z dołu lub z góry i z przodu lub z tyłu. Wygenerowany wolumin zostanie umieszczony w środku układu współrzędnych, tak aby osie wzdłuż których będziemy wykonywać warstwowanie przechodziły przez jego środek. Zapewni nam to odpowiednie ustawienie kamery na scenie, a co za tym idzie - możliwość przeglądania wszystkich warstw

pod jednakowym kątem (prostopadle do osi warstwowania). Z racji tego, że na scenie będziemy oglądać obrazy 2D (plastry woluminu), kamera nie będzie generowała perspektywy, aby uniknąć zniekształceń obrazu spowodowanych jej występowaniem.



Rysunek 3.3: Generowanie scen 3D przy uwzględnieniu perspektywy - po prawej oraz przy jej braku, w tzw. trybie szkieletowym - po lewej.

Spójrzmy na definicję klasy **QQVTKSlicing**

```
class QQVTKSlicing : public QQVTKScene
{
private:
  vtkSmartPointer<vtkInteractorStyleImage> interactorStyle;
  vtkSmartPointer<VTKSlicingCommand> slicingCommand;

  vtkSmartPointer<vtkLookupTable> lookupTable;
  vtkSmartPointer<vtkImageMapToColors> imageMapToColors;
  vtkSmartPointer<vtkImageReslice> imageReslice;
  vtkSmartPointer<vtkMatrix4x4> resliceAxes;
  vtkSmartPointer<vtkImageActor> imageActor;

  //Macierze widoku
  static double xMatrix[16];
  static double yMatrix[16];
  static double zMatrix[16];

  double* sideMatrix;

public:
  enum RenderingSides {X, Y, Z};

  QQVTKSlicing();
  void CreateScene() override;
  void SetRenderingSide(RenderingSides side);
  ~QQVTKSlicing();
};
```

Przestawię teraz wyjaśnienia elementów omawianej klasy:

- `vtkSmartPointer<vtkInteractorStyleImage> interactorStyle` - jest to styl interaktora omówiony w rozdziale 1.3, umożliwiający zmianę kontrasu i jasności wyświetlanego obrazu, jego powiększanie i obracanie, a także wywoływanie komend przechodzenia między warstwami;
- `vtkSmartPointer<VTKSlicingCommand> slicingCommand` - jest to obiekt klasy **VTKSlicingCommand**, omówionej w rozdziale 3.6, służącej do przechodzenia pomiędzy warstwami

woluminu. Opisany powyżej interaktor odbierając zdarzenia związane z użyciem kółka myszy, będzie wysyłał do obiektu tej klasy odpowiednią komendę, a ten obsługując ją, będzie wyświetlał poprzednią lub następną warstwę woluminu, w zależności od kierunku obrotu kółka myszy;

- `vtkSmartPointer<vtkLookupTable> lookupTable` - to obiekt przechowujący tablicę transferu kolorów, zamieniający wartości intensywności pikseli wczytanego obrazu na odpowiednie zapisane w tablicy wartości RGB;

- `vtkSmartPointer<vtkImageMapToColors> imageMapToColors` - jest to obiekt wykorzystujący tablicę transferu kolorów **lookupTable** do wykonywania funkcji transferu kolorów wczytanego obrazu;

- `vtkSmartPointer<vtkImageReslice> imageReslice` - jest to obiekt służący do tworzenia warstw ze wczytanego woluminu;

- `vtkSmartPointer<vtkMatrix4x4> resliceAxes` - jest to macierz 4x4, w której zapisanie zostana macierz definiująca oś przebiegu warstwowania;

- `vtkSmartPointer<vtkImageActor> imageActor` - to obiekt reprezentujący aktora 2D, czyli obraz na którym wyświetlana będzie aktualnie przeglądana warstwa, umieszczony na scenie VTK;

- `static double xMatrix[16], yMatrix[16], zMatrix[16]` - to macierze dzięki którym VTK wie, w jaki sposób wygenerować oś warstwowania. Macierze te są następujące:

$$xMatrix = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad yMatrix = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad zMatrix = \begin{bmatrix} 0 & 0 & -1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- `double* sideMatrix` - to wskaźnik na aktualnie wybraną macierz definiującą oś warstwowania;

- `void CreateScene()` - jest to implementacja czysto wirtualnej metody pochodzącej z klasy **QQVTKScene**, implementująca sposób tworzenia przez VTK sceny i dodawania na nią obiektów. Jej implementację wraz z dokładnym opisem ze względu na jej długość, zamieszczę poniżej tej listy;

- `void SetRenderingSide(RenderingSides side)` - jest to metoda służąca do wyboru osi wzdłuż której będziemy warstwować wolumin. Wartości parametru, jaki może przyjąć definiuje typ enumeracyjny **RenderingSides**;

Wróćmy teraz do metody **CreateScene**. Jest to implementacja czysto wirtualnej metody dziedziczonej z klasy **QQVTKScene**. Jej zadaniem jest umieszczenie woluminu w środku układu współrzędnych, wykonywanie funkcji transferu kolorów, wyświetlanie warstw woluminu oraz połączenie obu obiektów **interactorStyle** oraz **slicingCommand**, aby interaktor odpowiedzialny za obsługę interakcji z viewportem wyświetlającym warstwy woluminu wywoływał odpowiednie komendy zdefiniowane przez klasę **VTKSlicingCommand**.

Spójrzmy na pełną implementację tej metody:

```
void QQVTKSlicing::CreateScene()
{
  //Sprawdzamy czy plik NRRD został załadowany
  if (!nrrdLoader || !nrrdLoader->IsLoaded())
    return;

  int extent[6] = { 0 }; //rozmiary wczytanego obrazu
  double spacing[3]; //Odległości pomiędzy warstwami
  double origin[3]; //Punkt ciężkości woluminu

  //Pobieramy dane o wczytanym obrazie
  nrrdLoader->GetVTKConnector()->GetOutput()->GetSpacing(spacing);
  nrrdLoader->GetVTKConnector()->GetOutput()->GetOrigin(origin);

  //Obliczamy środek woluminu
  double center[3];
  center[0] = origin[0] + spacing[0]*0.5 * (extent[0] + extent[1]);
  center[1] = origin[1] + spacing[1]*0.5 * (extent[2] + extent[3]);
  center[2] = origin[2] + spacing[2]*0.5 * (extent[4] + extent[5]);

  //Kopiujemy wybraną macierz definiującą oś warstwowania
  resliceAxes->DeepCopy(sideMatrix);

  //Ustawiamy punkt przez który będziemy warstwować
  resliceAxes->SetElement(0, 3, center[0]);
  resliceAxes->SetElement(1, 3, center[1]);
  resliceAxes->SetElement(2, 3, center[2]);

  //Warstwujemy obraz
  imageReslice->SetInputData(nrrdLoader->GetVTKConnector()->GetOutput());
  imageReslice->SetOutputDimensionality(2);
  imageReslice->SetResliceAxes(resliceAxes);
  imageReslice->SetInterpolationModeToLinear();

  //Tworzymy funkcję transferu kolorów
  lookupTable->SetRange(0, 2000);
  lookupTable->SetValueRange(0.0, 1.0);
  lookupTable->SetSaturationRange(0.0, 0.0);
  lookupTable->SetRampToLinear();

  //Wykonujemy tę funkcję
  imageMapToColors->SetLookupTable(lookupTable);
  imageMapToColors->SetInputConnection(imageReslice->GetOutputPort());

  //Dodajemy aktora do sceny
  imageActor->GetMapper()->SetInputConnection(imageMapToColors->
      GetOutputPort());
  renderer->AddActor(imageActor);

  //Ustawiamy odpowiednie komendy wywoływane poprzez ruch kółka myszy
  slicingCommand->SetImageReslice(imageReslice);
  slicingCommand->SetInteractor(interactor);

  interactorStyle->AddObserver(vtkCommand::MouseWheelForwardEvent,
      slicingCommand);
  interactorStyle->AddObserver(vtkCommand::MouseWheelBackwardEvent,
      slicingCommand);

  //Ustawiamy kamerę oraz renderujemy obraz
  renderer->ResetCamera();
```

```
    renderer->Render();
}
```

Oto dokładny opis zadań realizowanych przez powyższą metodę:

1. Na początku sprawdzamy, czy został wczytany już jakiś obraz. Jeśli nie, metoda zakończy swoje działanie.
2. Teraz pobierane są dane wczytanego woluminu, takie jak jego rozmiary w trzech wymiarach oraz punkt ciężkości, w celu obliczenia środka woluminu i umieszczenia go w centrum układu współrzędnych.
3. Po umieszczeniu woluminu w centrum układu wykonujemy warstwowanie - tworzymy plastry ze wczytanego w woluminu. Warstwy tworzone się wzdłuż wybranej osi warstwowania.
4. Teraz definiujemy funkcję transferu kolorów i wykonujemy ją, zmieniając odpowiednie wartości pikseli obrazu na nowe wartości.
5. Dodajemy aktora 2D w postaci obrazu wyświetlającego warstwy na scenę VTK i ustawiamy zdarzenie wywoływane za pomocą kółka myszy, służące do przechodzenia pomiędzy warstwami woluminu wzdłuż wybranej osi.
6. Na koniec resetujemy kamerę tak, aby środek jej viewportu znajdował się w środku woluminu i renderujemy nowo powstałą scenę.

Od tego momentu nasza aplikacja jest już zdolna do wyświetlania warstw woluminu wzdłuż osi x, y oraz z oraz umożliwia przechodzenie pomiędzy tymi warstwami za pomocą myszy. W następnym podrozdziale zajmiemy się drugim typem viewportu, służącym do renderowania trójwymiarowych woluminów metodą GPU ray casting oraz wykorzystującym funkcję transferu kolorów i przezroczystości.

3.5. Klasa QQVTKVolumeRendering - klasa odpowiedzialna za rendering wolumetryczny

Następna z opisywanych klas odpowiedzialna będzie za wyświetlanie trójwymiarowych woluminów na scenie VTK, transformowanie kolorów ich wokseli przy pomocy funkcji transferu kolorów i przezroczystości oraz umożliwienie użytkownikowi obracania kamerą poprze interakcję z viewportem za pomocą myszy.

Klasa ta ma bardzo podobny charakter do klasy **QQVTKSlicing** opisanej w poprzednim podrozdziale, ze względu na to, że obie dziedziczą funkcjonalność po klasie **QQVTKScene**.

Spójrzmy więc najpierw na definicję klasy **QQVTKVolumeRendering**, aby potem przejść do dokładnego opisu jej elementów i zadania jakie pełnić będzie klasa:

```
class QQVTKVolumeRendering : public QQVTKScene
{
public:
  enum ColorFunctions {
    MIP, CompositeRamp, CompositeShadeRamp, Skin, Bone, Muscle, Composite
  };
```

```
private:
  vtkSmartPointer<vtkInteractorStyleTrackballCamera> interactorStyle;
  vtkSmartPointer<vtkOpenGLGPUVolumeRayCastMapper> mapper;
  vtkSmartPointer<vtkColorTransferFunction> volumeColors;
  vtkSmartPointer<vtkPiecewiseFunction> volumeOpacities;
  vtkSmartPointer<vtkVolumeProperty> volumeProperty;
  vtkSmartPointer<vtkVolume> volume;
  ColorFunctions colorFunction;

public:
  QQVTKVolumeRendering();
  void CreateScene();
  void CreateColorFunction();
  ColorFunctions GetColorFunction();
  void SetColorFunction(ColorFunctions);
  ~QQVTKVolumeRendering();
};
```

Omówmy teraz dokładnie elementy klasy **QQVTKVolumeRendering**

- `enum ColorFunctions` - jest to typ enumeracyjny definiujący identyfikatory funkcji transferu kolorów;
- `vtkSmartPointer<vtkInteractorStyleTrackballCamera> interactorStyle` - jest to styl interaktora umożliwiający użytkownikowi obrót i przesuwanie kamery za pomocą myszki;
- `vtkSmartPointer<vtkOpenGLGPUVolumeRayCastMapper> mapper` - jest to obiekt klasy służącej do generowania trójwymiarowych woluminów metodą GPU ray casting;
- `vtkSmartPointer<vtkColorTransferFunction> volumeColors` - jest to obiekt odpowiedzialny za zamianę wartości intensywności wokseli na kolory;
- `vtkSmartPointer<vtkPiecewiseFunction> volumeOpacities` - jest to obiekt odpowiedzialny za zamianę wartości intensywności wokseli na poziomy ich przezroczystości;
- `vtkSmartPointer<vtkVolumeProperty> volumeProperty` - jest to obiekt zawierający parametry woluminu, m. in. takie jak funkcja transferu kolorów i funkcja transferu przezroczystości (patrz: rozdział 1.3);
- `vtkSmartPointer<vtkVolume> volume` - jest to gotowy do umieszczenia na scenie wolumin;
- `ColorFunctions colorFunction` - identyfikator aktualnie wybranej funkcji transferu kolorów;
- `void CreateScene()` - jest to implementacja czysto wirtualnej metody dziedziczonej z klasy **QQVTKScene**. Za jej pomocą generowane będą sceny wyświetlające woluminy. Dokładny opis oraz implementację tej metody umieszczę pod niniejszą listą elementów klasy;
- `void CreateColorFunction` - jest to metoda służąca do tworzenia funkcji transferu kolorów i przezroczystości, używana przez metodę **CreateScene** do przetworzenia woluminu przed dodaniem go na scenę. Spójrzmy jak tworzona jest za pomocą tej metody funkcja transferu kolorów CompositeRamp (patrz: rozdział 3.2):

```
void QQVTKVolumeRendering::CreateColorFunction()
{
  //Resetujemy obiekty użyte do tworzenia funkcji koloru
```

```
    volumeProperty = vtkSmartPointer<vtkVolumeProperty>::New();
    volumeColors->RemoveAllPoints();
    volumeOpacities->RemoveAllPoints();

    switch (colorFunction)
    {
    //Tworzymy funkcję transferu o nazwie CompositeRamp
    case QQVTKVolumeRendering::CompositeRamp:
      //Definiujemy punkty transferu kolorów
      volumeColors->AddRGBSegment(0, 0.0, 0.0, 0.0, 4096, 1.0, 1.0, 1.0);
      //Definiujemy punkty transferu przezroczystości
      volumeOpacities->AddSegment(0, 0.0, 4096, 1.0);
      //Ustawiamy tryb przezroczystości na Composite
      mapper->SetBlendModeToComposite();
      //Wyłączamy cieniowanie obiektu
      volumeProperty->ShadeOff();
      break;

      //tworzenie pozostałych funkcji transferu kolorów
    }

    //Dodawanie do parametrów woluminu utworzonych funkcji transferu kolor
        ów i przezroczystości
    volumeProperty->SetColor(volumeColors);
    volumeProperty->SetScalarOpacity(volumeOpacities);

    //Tworzenie woluminu
    volume->SetMapper(mapper);
    volume->SetProperty(volumeProperty);
}
```

Spójrzmy teraz na dokładne działanie tej metody na przykładzie tworzenia funkcji CompositeRamp:

1. Resetujemy punkty definiujące funkcję transferu kolorów i przezroczystości oraz tworzymy nowy obiekt, w którym zapiszemy parametry nowego woluminu.
2. Teraz tworzymy funkcję CompositeRamp, definiując najpierw funkcję transferu kolorów i przezroczystości, następnie uzależniając poziom przezroczystości woksela od jego jasności (im jaśniejszy woksel, tym przyjmie mniejszą przezroczystość) i w końcu wyłączając cieniowanie obiektu (obiekt będzie oświetlany jednakowo, niezależnie od kąta pod jakim pada światło).
3. Dodajemy teraz nowo stworzone funkcje transferu jako parametry woluminu, do obiektu odpowiedzialnego za definiowanie tych parametrów.
4. Na koniec tworzymy wolumin i przypisujemy mu utworzone wcześniej właściwości - w tym wypadku transformujemy jego kolory na nowe kolory i na odpowiednie poziomy przezroczystości.

- `ColorFunctions GetColorFunction()` - jest to metoda służąca zewnętrznym obiektom do pobierania identyfikatora aktualnie wybranej przez użytkownika funkcji transferu kolorów;
- `void SetColorFunction(ColorFunctions)` - ostatnia z opisywanych metod służy do zmiany funkcji transferu kolorów. Wywoływana jest w momencie zmiany funkcji transferu przez użytkownika za pomocą klasy adaptującej. Jej parametrem jest identyfikator nowej funkcji transferu kolorów;

Powróćmy teraz do metody **CreateScene**. Jej zadaniem jest tworzenie sceny zawierającej wczytany wolumin z pliku NRRD i przetworzenie go przy użyciu funkcji transferu kolorów i przezroczystości. Na początku dane woluminu pobierane są z klasy **QQVTKNRRDLoader**, odpowiedzialnej za wczytywanie plików NRRD. Następnie wywoływana jest metoda **CreateColorFunction** tworząca nową funkcję transferu kolorów i transformująca kolory i przezroczystości wokseli woluminu za pomocą nowo utworzonej funkcji. Na koniec przetworzony wolumin dodawany jest do sceny, a kamera jest resetowana, w celu skupienia jej viewportu na woluminie i scena jest renderowana.

Implementacja metody **CreateScene** wygląda następująco:

```
void QQVTKVolumeRendering::CreateScene()
{
  //Jeśli nie załadowano żadnego woluminu, zakończ działanie
  if (!nrrdLoader || !nrrdLoader->IsLoaded())
    return;

  //Pobierz dane woluminu wczytanego za pomocą ITK i klasy QQVTKNRRDLoader
  mapper->SetInputData(nrrdLoader->GetVTKConnector()->GetOutput());

  //Wywoływanie metody tworzącej odpowiednią funkcję transferu kolorów
  CreateColorFunction();

  //Dodawanie przetworzonego woluminu do sceny
  renderer->AddViewProp(volume);

  //Resetowanie kamery i renderowanie obrazu
  renderer->ResetCamera();
  renderer->Render();
}
```

3.6. Klasa VTKSlicingCommand - klasa odpowiedzialna za przechodzenie pomiędzy warstwami woluminu

Implementacja aplikacji jest już prawie skończona. W ostatnim z podrozdziałów dotyczących tworzenia i opisywania klas zajmiemy się klasą odpowiedzialną za przetwarzanie zdarzenia związanego z przechodzeniem pomiędzy warstwami woluminu. Klasa ta dziedziczy zdolność obserwacji obiektów VTK z klasy **vtkCommand** (patrz: rozdział 1.3). Dzięki obserwowaniu interaktora viewportu, obiekty klasy**VTKSlicingCommand** przechwycą zdarzenie wywołane obrotem kółka myszy i obsłużą je na wyspecjalizowany sposób.

Spójrzmy na definicję klasy **VTKSlicingCommand**:

```
class VTKSlicingCommand : public vtkCommand {
private:
  vtkImageReslice* imageReslice; //obiekt tworzący warstwy woluminu
  vtkRenderWindowInteractor* interactor; //interaktor okna VTK

public:
  VTKSlicingCommand();
  //Metoda służąca do tworzenia nowych obiektów klasy VTKSlicingCommand
  static VTKSlicingCommand* New();

  //Metoda wywoływana podczas zajścia zdarzenia
  void Execute(vtkObject*, unsigned long, void *) override;
```

```
  void SetImageReslice(vtkImageReslice*);
  void SetInteractor(vtkRenderWindowInteractor*);
  vtkImageReslice *GetImageReslice();
  vtkRenderWindowInteractor *GetInteractor();

  ~VTKSlicingCommand();
};
```

Spójrzmy teraz na listę najważniejszych elementów tej klasy

- `vtkImageReslice* imageReslice` - jest to wskaźnik na obiekt odpowiedzialny za tworzenie warstw woluminu;
- `vtkRenderWindowInteractor* interactor` - jest to wskaźnik na interaktor okna VTK, w którym renderowana jest scena wyświetlająca warstwy woluminu;
- `void Execute(vtkObject*, unsigned long, void *)` - metoda wywoływana w momencie zajścia zdarzenia w interaktorze. Za jej pomocą zdarzenia wywołane przez obrót kółka myszy zostanie przechwycone i nie obsłuży go interaktor, ale właśnie obiekt klasy **VTKSlicingCommand**, który przejął to zdarzenie. Za jej pomocą użytkownik będzie miał możliwość przechodzić pomiędzy warstwami woluminu.

  Spójrzmy zatem na jej implementację:

```
void VTKSlicingCommand::Execute(vtkObject*, unsigned long event, void *)
    {
  vtkRenderWindowInteractor *interactor = this->GetInteractor();

  //Sprawdzamy czy zaszłe zdarzenie to obrót kółka myszy
  if (event == vtkCommand::MouseWheelForwardEvent || event == vtkCommand
      ::MouseWheelBackwardEvent)
  {
    //Pobieramy kierunek obrotu kółka
    int deltaY = event == vtkCommand::MouseWheelForwardEvent ? 1 : -1;

    //Pobieramy dane o woluminie
    imageReslice->Update();
    double sliceSpacing = imageReslice->GetOutput()->GetSpacing()[2];
    vtkMatrix4x4 *matrix = imageReslice->GetResliceAxes();

    //Obliczamy nowe punkty definiujące położenie warstwy w woluminie
    double point[4], center[4];

    point[0] = 0.0; point[1] = 0.0;
    point[2] = sliceSpacing * deltaY; point[3] = 1.0;

    matrix->MultiplyPoint(point, center);
    matrix->SetElement(0, 3, center[0]);
    matrix->SetElement(1, 3, center[1]);
    matrix->SetElement(2, 3, center[2]);

    //Renderujemy scenę zawierającą nową warstwę woluminu
    interactor->Render();
  }
};
```

Opiszę teraz dokładne działanie metody **Execute**:

1. Na początku sprawdzany jest typ zaszłego zdarzenia, jeśli był to obrót kółka myszy, przechodzimy do jego obsługi, w przeciwnym wypadku wychodzimy z metody.
2. Następnie pobierane są dane obrazu NRRD, w celu obliczenia nowego punktu informującego o położeniu warstwy obrazu wzdłuż osi warstwowania. Nowy punkt przesuwany jest wzdłuż tej osi w zależności od kierunku obrotu kółka myszy. Jeśli kółko zostało obrócone górę (zmienna **deltaY** przyjęła wartość 1), to punkt ten przesuwany jest w kierunku przeciwnym do kamery, aby wyświetlić warstwę znajdującą głębiej. Z drugiej strony jeśli kierunek obrotu był odwrotny i zmienna **deltaY** przyjęła wartość -1, punkt przesuwany jest w kierunku kamery, powodując wyświetlenie warstwy, która jest bliżej niej, niż warstwa dotychczas wyświetlana.
3. Po ustaleniu która warstwa ma się pojawić na scenie renderuje się scenę. Na tym metoda kończy swoje działanie.

- Pozostałe metody służą do pobierania i ustalania obiektów wykorzystywanych przez tę klasę w metodzie **Execute**;

Definiując klasę **VTKSlicingCommand** i implementując jej działanie, zakończyliśmy implementację całej aplikacji. W ostatnim rozdziale pracy podsumujemy jej przebieg, przypomnimy sobie problemy związane z komunikacją biblioteki Qt Quick i VTK oraz ocenimy czy wszystkie zamierzone cele zostały zrealizowane.

Rozdział 4

Podsumowanie

Celem mojej pracy było przede wszystkim skomunikowanie biblioteki Qt Quick z VTK. By osiągnąć ten cel pogłębiłem swoją wiedzę dotyczącą przede wszystkim biblioteki VTK, ale również bibliotek Qt Quick i OpenGL oraz przetestowałem kilka potencjalnych rozwiązań, z których tylko jedno - opisane w niniejszej pracy - okazało się skuteczne i nie zaburzające w żaden sposób pracy aplikacji, czy wykorzystanych bibliotek. Jak zostało to zdefiniowane w rozdziale 1.1, największym problemem z jakim musiałem się zmierzyć było poprawne renderowanie obrazu, niezaburzające procesu jego tworzenia przez bibliotekę OpenGL. Żeby uniknąć przeplatania się wywołań funkcji z biblioteki OpenGL przez VTK i Qt Quick, stworzyłem klasę adaptującą **QQVTKAdapter**, gwarantującą, że zarówno VTK, jak i Qt Quick będą miały oddzielne okna renderujące oraz przede wszystkim, że renderowanie w tych bibliotekach nigdy nie będzie przebiegać w tym samym czasie. Do obsługi komunikacji od strony Qt Quick napisałem klasę **QQVTKViewport**, której zadaniem jest wyświetlanie otrzymywanego z VTK obrazu oraz przekazywanie zdarzeń wywoływanych za pomocą myszki do interaktora okna renderującego VTK. Do komunikacji od strony biblioteki VTK została z kolei napisana klasa **QQVTKScene**, która tworzy podstawowe środowisko VTK, zdolne komunikować się z Qt Quick i gotowe na wyspecjalizowanie poprzez odziedziczenie swojej funkcjonalności oraz dowolne rozszerzenie jej poprzez implementację czysto wirtualnej metoty **CreateScene** i dodanie dowolnego stylu interaktora i innych ewentualnych obiektów VTK, potrzebnych do zaimplementowania wyspecjalizowanej funkcjonalności.

Kiedy już środowisko komunikacji Qt Quick z VTK było gotowe, w rozdziale 3 zaimplementowałem w oparciu o nie konkretną aplikację, służącą do wyświetlania woluminów wczytanych z plików NRRD. Jako że klasa **QQVTKScene** jest klasą abstrakcyjną, wyspecjalizowałem jej działanie na potrzeby pisanej aplikacji, tworząc klasy **QQVTKSlicing** oraz **QQVTKVolumeRendering**, dzięki którym aplikacja zyskała funkcjonalność pozwalającą na rendering woluminów metodą GPU ray casting, transformowanie kolorów i przezroczystości wokseli przy pomocy funkcji transferu, a także na wyświetlanie warstw woluminu oraz przechodzenie pomiędzy nimi za pomocą myszy.

Utworzenie klas **QQVTKScene**, **QQVTKViewport** oraz **QQVTKAdapter** otworzyło nowe możliwości tworzenia aplikacji z wykorzystaniem bibliotek Qt Quick i VTK. Przy wykorzystaniu opisanego przeze mnie sposobu komunikacji, jedynym ograniczeniem wykorzystania omawianych bibliotek jest właściwie ich funkcjonalność. Nie ma przeszkód, by za pomocą Qt Quick budować dowolny interfejs, który będzie się komunikował z VTK, a z drugiej strony nie ma też przeszkód, by tworzyć dowolne środowisko VTK w oparciu o dowolne

obiekty z biblioteki VTK i ITK, poprzez wyspecjalizowanie klasy **QQVTKScene**.

Na koniec poruszę jeszcze sprawę wydajności zastosowanego przeze mnie rozwiązania. Zdaję sobie sprawę, że przez to, że obrazy renderowane są w dwóch miejscach (raz w VTK, jako renderowanie właściwej sceny, drugi raz w Qt Quick, kopiując dane wyrenderowanej sceny pochodzące z VTK), wydajność ta jest niższa, niż przy generowaniu scen przy użyciu samej biblioteki VTK, jednak spadek wydajności nie jest na tyle duży, żeby odrzucać opisywane przeze mnie rozwiązanie. Jestem pewien że istnieje wiele dróg na usprawnienie tej wydajności, choćby poprzez bezpośrednie wywołania funkcji z biblioteki OpenGL w celu rysowania obrazu pochodzącego z VTK w viewporcie Qt Quick, albo zastosowania programów zwanych **pixel shaderami**, wykonywanych na procesorze karty graficznej, wykonujących właśnie to kopiowanie. Wówczas proces kopiowania obrazu miałby miejsce nie w pamięci komputera i w jego procesorze, lecz w procesorze i pamięci karty graficznej, czyli jednostce wyspecjalizowanej do właśnie takich obliczeń.

Bibliografia

[1] Dokumentacja biblioteki VTK, dostęp 16.06.2018
https://www.vtk.org/documentation/

[2] Dokumentacja biblioteki ITK, dostęp 16.06.2018
https://itk.org/ITK/help/documentation.html

[3] Dokumentacja biblioteki Qt Quick, dostęp 16.06.2018
http://doc.qt.io/qt-5/qtquick-index.html

[4] R. Mantiuk, *Potok graficzny 3D*, dostęp 16.06.2018
http://rmantiuk.zut.edu.pl/data/wyklad_gk_potok_graficzny.pdf