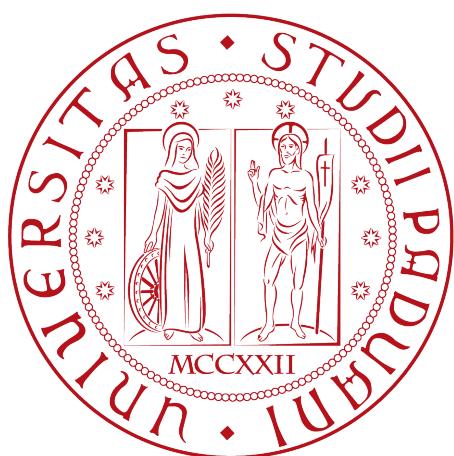


Università degli Studi di Padova
DIPARTIMENTO DI MATEMATICA "TULLIO LEVI-CIVITA"
CORSO DI LAUREA IN INFORMATICA



**Visualizzazione Volumetrica 3D di immagini
radiologiche**

Tesi di laurea triennale

Relatore

Prof. Davide Bresolin

Laureando

Michele Roverato

ANNO ACCADEMICO 2019-2020

Michele Roverato: *Visualizzazione Volumetrica 3D di immagini radiologiche*, Tesi di laurea triennale, © Dicembre 2020.

Ringraziamenti

Innanzitutto, vorrei esprimere la mia gratitudine al Prof. Davide Bresolin, relatore della mia tesi, per l'aiuto e il sostegno fornитоми durante la stesura del lavoro.

Ringrazio tutta G-Squared e in particolar modo il mio tutor aziendale Gianluca Ghelli, per l'opportunità e l'aiuto nello svolgere questo progetto, anche a distanza.

Ringrazio tutti i miei amici, universitari e non, per le esperienze vissute durante questi anni, in particolar modo Andrea, Diego e Francesco. Ringrazio inoltre le persone con cui ho avuto modo di condividere il percorso universitario, tra cui i membri del team di Ingegneria del Software 7DOS.

Desidero infine ringraziare con affetto tutta la mia famiglia per essermi stata vicina in ogni momento durante gli anni di studio.

Padova, Dicembre 2020

Michele Roverato

Indice

| | | |
|----------|--|----------|
| 1 | Introduzione | 1 |
| 1.1 | Sommario | 1 |
| 1.2 | Organizzazione del testo | 1 |
| 2 | Descrizione dello stage | 3 |
| 2.1 | Scelta dello stage | 3 |
| 2.2 | L'azienda | 3 |
| 2.3 | Introduzione al progetto e interessi aziendali | 4 |
| 2.4 | Obiettivi | 4 |
| 2.4.1 | Obiettivi obbligatori | 4 |
| 2.4.2 | Obiettivi desiderabili | 4 |
| 2.4.3 | Obiettivi facoltativi | 4 |
| 2.5 | Contesto Applicativo | 5 |
| 2.6 | Tecnologie Utilizzate | 5 |
| 2.6.1 | C++ | 5 |
| 2.6.2 | Qt5 | 5 |
| 2.7 | Strumenti Utilizzati | 6 |
| 2.7.1 | Visual Studio | 6 |
| 2.7.2 | Qt Creator | 6 |
| 2.7.3 | CMake | 6 |
| 2.7.4 | Git | 6 |
| 2.8 | Aspettative personali | 6 |
| 3 | Volume Rendering e Qt | 7 |
| 3.1 | Radiologia e rendering | 7 |
| 3.1.1 | Visualizzare informazioni | 7 |
| 3.1.2 | Volumi radiologici | 7 |
| 3.1.3 | Standard DICOM | 8 |
| 3.1.4 | Basi di rendering | 8 |
| 3.1.5 | Camera | 9 |
| 3.1.6 | Rendering di oggetti non solidi | 10 |
| 3.2 | Volume rendering | 10 |
| 3.2.1 | Basi di volume rendering | 10 |
| 3.2.2 | Voxel | 11 |
| 3.2.3 | Volume rendering Image-Order | 11 |
| 3.2.4 | Volume rendering Object-Order | 12 |
| 3.2.5 | Funzione di trasferimento | 12 |
| 3.2.6 | Regioni di interesse | 13 |

| | | |
|----------|---|-----------|
| 3.3 | VTK | 14 |
| 3.3.1 | Descrizione e scelta della libreria | 14 |
| 3.3.2 | Build della libreria | 14 |
| 3.3.3 | Oggetti di rendering | 15 |
| 3.3.4 | Particolari oggetti di rendering | 16 |
| 3.3.5 | Pipeline di rendering | 16 |
| 3.3.6 | Widget e interazione utente | 17 |
| 3.3.7 | Integrazione con Qt | 18 |
| 3.4 | Strumenti CTK | 18 |
| 3.5 | Basi di ITK | 19 |
| 3.6 | Software correlati | 19 |
| 3.6.1 | 3D Slicer | 19 |
| 3.6.2 | MITK e SimVascular | 19 |
| 4 | Resoconto Stage | 21 |
| 4.1 | Pianificazione | 21 |
| 4.1.1 | Pianificazione iniziale | 21 |
| 4.1.2 | Sistema di issue e Gantt | 21 |
| 4.1.3 | Discussioni e incontri con il tutor | 21 |
| 4.1.4 | Problemi e ritardi | 22 |
| 4.2 | Implementazione | 22 |
| 4.2.1 | Impostazione ambiente di sviluppo | 22 |
| 4.2.2 | Studio di 3D Slicer | 22 |
| 4.2.3 | Studio e installazione di VTK | 22 |
| 4.2.4 | Primo CMakeLists | 23 |
| 4.2.5 | Ripasso Qt | 24 |
| 4.2.6 | Integrazione Qt-VTK | 25 |
| 4.2.7 | Volume Mapper | 25 |
| 4.2.8 | Primo prototipo | 25 |
| 4.2.9 | Strumenti di base - Rotazione | 25 |
| 4.2.10 | Strumenti di base - Preset | 26 |
| 4.2.11 | Strumenti di base - Mark | 26 |
| 4.2.12 | Strumenti di base - MIP | 27 |
| 4.2.13 | Strumenti di base - Smoothing | 27 |
| 4.2.14 | Strumenti aggiuntivi - Funzione Threshold | 28 |
| 4.2.15 | Strumenti aggiuntivi - Taglio tramite box | 29 |
| 4.2.16 | Strumenti aggiuntivi - Taglio tramite plane | 29 |
| 4.2.17 | Compilazione librerie aziendali | 30 |
| 4.2.18 | Problemi compilazione librerie aziendali | 31 |
| 4.2.19 | Import volume con librerie aziendali | 31 |
| 4.2.20 | Modifiche interfaccia | 33 |
| 4.2.21 | Esempio porting | 34 |
| 4.3 | Documentazione | 34 |
| 4.3.1 | Codice | 34 |
| 4.3.2 | Wiki | 35 |
| 4.4 | Test | 35 |
| 4.4.1 | Possibili test su una UI | 35 |
| 4.4.2 | Test di 3D Slicer e VTK | 36 |
| 4.4.3 | Test implementati | 36 |

| | |
|---|-----------|
| <i>INDICE</i> | vii |
| 5 Conclusioni | 37 |
| 5.1 Consuntivo finale | 37 |
| 5.2 Raggiungimento degli obiettivi | 38 |
| 5.2.1 Obiettivi obbligatori | 38 |
| 5.2.2 Obiettivi desiderabili | 38 |
| 5.2.3 Obiettivi facoltativi | 38 |
| 5.3 Conoscenze e abilità acquisite | 38 |
| 5.3.1 Azienda | 39 |
| 5.3.2 Immagini radiologiche e DICOM | 39 |
| 5.3.3 Volume Rendering | 39 |
| 5.3.4 Linguaggi e tecnologie | 39 |
| 5.3.5 Strumenti | 39 |
| 5.4 Valutazione personale | 40 |
| Bibliografia e sitografia | 41 |

Elenco delle figure

| | | |
|------|--|----|
| 1.1 | <i>Sequenza dei componenti: in evidenza l'oggetto principale del lavoro</i> | 1 |
| 2.1 | <i>Logo azienda G-Squared</i> | 4 |
| 2.2 | <i>Logo EyeRad G-Squared</i> | 5 |
| 3.1 | <i>La luce e la vista</i> | 8 |
| 3.2 | <i>Algoritmo di Ray Tracing</i> | 9 |
| 3.3 | <i>Proprietà della camera</i> | 9 |
| 3.4 | <i>Volume rendering Image-Order</i> | 11 |
| 3.5 | <i>Esempio di "average value" e di MIP</i> | 12 |
| 3.6 | <i>Volume rendering Object-Order</i> | 13 |
| 3.7 | <i>Semplice classificazione binaria (sinistra) e una transizione graduale tra aria, muscoli e ossa (destra).</i> | 13 |
| 3.8 | <i>Esempio di volume rendering con regione di interesse, tagliando il volume.</i> | 14 |
| 3.9 | <i>Logo di VTK</i> | 14 |
| 3.10 | <i>Gerarchia dell'applicazione</i> | 15 |
| 3.11 | <i>Oggetti di rendering di VTK</i> | 16 |
| 3.12 | <i>Oggetti VolumeMapper di VTK</i> | 17 |
| 3.13 | <i>Struttura oggetti della Pipeline</i> | 17 |
| 3.14 | <i>Esempi di widget 3D di VTK</i> | 18 |
| 3.15 | <i>Esempi di widget 3D di CTK</i> | 18 |
| 3.16 | <i>Simulazione del flusso di sangue in una "Coronary Artery Bypass Simulation (CABG)"</i> | 20 |
| 4.1 | <i>Alcuni parametri di configurazione di VTK su CMake-gui</i> | 23 |
| 4.2 | <i>Primo volume rendering</i> | 26 |
| 4.3 | <i>Esempio di Smoothing su GPU, OFF a sinistra e ON a destra</i> | 27 |
| 4.4 | <i>Prima interfaccia con strumenti</i> | 28 |
| 4.5 | <i>Esempio di funzione di Threshold in 3D Slicer</i> | 28 |
| 4.6 | <i>Esempio di taglio del volume tramite vtkBoxWidget</i> | 29 |
| 4.7 | <i>Esempio di taglio del volume tramite vtkPlaneWidget</i> | 30 |
| 4.8 | <i>Esempio di import volume incorretto (ordine errato)</i> | 32 |
| 4.9 | <i>Esempio di import volume incorretto (scala errata)</i> | 32 |
| 4.10 | <i>Esempio di import volume incorretto (problema con vtkImageShiftScale)</i> | 33 |
| 4.11 | <i>Cattura del design finale della UI</i> | 34 |
| 4.12 | <i>Integrazione RenderWidget in altra applicazione</i> | 35 |

Elenco delle tavelle

| | |
|---------------------------------|----|
| 5.1 Consuntivo finale | 37 |
|---------------------------------|----|

Capitolo 1

Introduzione

1.1 Sommario

Il presente documento descrive il lavoro che ho svolto durante il periodo di stage, della durata di 304 ore, presso l'azienda G-Squared Srl di Ponte San Nicolò (PD).

Il progetto di stage prevedeva l'analisi, la progettazione e lo sviluppo di un *widget* Qt (elemento principale per la creazione di interfacce utente in Qt) che permetta la visualizzazione 3D volumetrica di una ricostruzione fatta tramite immagini diagnostiche medicali radiologiche, catturate tramite TC (Tomografia computerizzata) o RM (Risonanza magnetica).

Questo elaborato ha lo scopo di illustrare il contesto aziendale dove è stato svolto lo stage, le attività svolte durante esso, ed infine una valutazione sul lavoro effettuato.

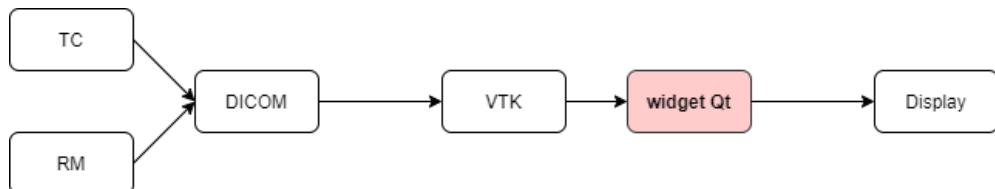


Figura 1.1: Sequenza dei componenti: in evidenza l'oggetto principale del lavoro

Nella figura 1.1 possiamo vedere la sequenza dei componenti: dall'esame radiologico svolto tramite TC o RM che genera i file DICOM, al loro caricamento nella libreria VTK. Il risultato viene quindi elaborato e mostrato nel *widget* Qt, aspetto principale del lavoro di stage, per essere mostrato a schermo.

1.2 Organizzazione del testo

Il testo è così suddiviso:

- * [Il primo capitolo](#) introduce il contenuto della tesi e descrive l'organizzazione del testo;
- * [Il secondo capitolo](#) descrive l'azienda, gli obiettivi dello stage, le tecnologie e gli strumenti utilizzati;

- * Il terzo capitolo descrive la teoria e gli algoritmi alla base del *volume rendering*, con menzione alle librerie utilizzate;
- * Il quarto capitolo descrive l'esperienza effettuata nella progettazione e nello sviluppo del progetto di stage;
- * Il quinto capitolo presenta una valutazione dello stage in relazione agli obiettivi dell'azienda e all'esperienza da me acquisita nel corso del suo svolgimento.

Riguardo la stesura del testo, relativamente al documento sono state adottate le seguenti convenzioni tipografiche:

- * gli acronimi, le abbreviazioni e i termini ambigui o di uso non comune menzionati vengono definiti nel loro primo utilizzo;
- * i termini in lingua straniera o facenti parti del gergo tecnico sono evidenziati con il carattere *corsivo*.

Capitolo 2

Descrizione dello stage

2.1 Scelta dello stage

Il *rendering real-time* e le sue varie applicazioni sono argomenti che ho iniziato ad approfondire già negli anni scorsi: dai videogiochi, al *training*, fino a simulazioni di vario genere. Ho partecipato all'evento Stage-IT 2019, ma durante la ricerca di uno stage a fine 2019 tra i vari colloqui ho deciso di ampliare il mio orizzonte, cercando le opportunità presentate agli eventi Stage-IT degli anni passati.

Dopo varie ricerche, ho trovato G-Squared nel documento di Stage-IT 2017. Contattando e confrontandomi con l'azienda, ho deciso di unire la mia passione per C++ e per il *rendering* a un ambito utile, come quello medica, in modo da imparare di più su come funziona e tutte le tecnologie che vengono impiegate.

2.2 L'azienda

G-Squared Srl è una società di ingegneria informatica specializzata nello sviluppo di *software* applicativo medicale. Nata nel 2008 con sede legale a Vicenza e come spin-off della società Studio Synthesis Sistemi Avanzati srl, ora ha sede operativa a Ponte San Nicolò (PD).

La filosofia di G-Squared si incentra molto su spirito di squadra, rispetto delle regole e valore alla persona e al merito.

Punto di forza dell'azienda è l'esperienza acquisita in oltre vent'anni di attività svolta in ambito medica. Questo permette a G-Squared di offrire ai propri clienti una serie di servizi, tra cui:

- * Interfacciamento con macchine diagnostiche digitali;
- * Consulenze per reti di macchine diagnostiche;
- * Elaborazione di immagini diagnostiche specializzate.

Lavorando con *software* medici, come visualizzatori DICOM (Digital Imaging and COmmunications in Medicine), G-Squared è un'azienda certificata ISO 13485:2016. Possiede inoltre molte certificazioni per i propri *software*, che nel caso delle immagini DICOM si attengono alle relative dichiarazioni di conformità. Più in generale, molti dei *software* prodotti da G-Squared sono stati classificati come dispositivo di classe I secondo i criteri della Direttiva 1993/42/CE.



Figura 2.1: Logo azienda G-Squared

Fonte: gsquared.it

2.3 Introduzione al progetto e interessi aziendali

G-Squared attraverso questo progetto mira ad analizzare la possibilità di integrare un visualizzatore volumetrico in uno dei suoi *software*. Se fatto correttamente, questo può essere particolarmente vantaggioso, permettendo al medico di analizzare il volume tridimensionale insieme alle immagini radiologiche, fornendo una visione più completa.

2.4 Obiettivi

L'obiettivo di questo stage è stato lo sviluppo di un *widget* Qt (definito nella sezione §1.1) che permetta la visualizzazione 3D volumetrica di una ricostruzione fatta tramite immagini diagnostiche medicali radiologiche. Il mio compito quindi era analizzare le funzionalità del *framework* VTK e la sua integrazione con il *framework* Qt, per sviluppare un *widget* che permettesse di caricare un volume da un esame radiologico e di visualizzarlo. Era richiesta la possibilità di applicarci dei filtri tramite dei *preset* prestabiliti e la possibilità di tagliare il volume. Importante era la compatibilità con le librerie aziendali, in modo da rendere il *widget* facilmente integrabile con gli strumenti e i *database* di gestione delle immagini già presenti e utilizzati in azienda.

Durante la stesura del piano di lavoro con il *tutor* aziendale, che è avvenuta prima dell'inizio dello stage, sono stati individuati obiettivi suddivisi in obbligatori, desiderabili e facoltativi.

2.4.1 Obiettivi obbligatori

- * O01: visualizzazione interattiva volumetrica;
- * O02: possibilità di scelta della funzione di trasferimento dei *voxel* (colore, trasparenza).

2.4.2 Obiettivi desiderabili

- * D01: piani di taglio del volume;
- * D02: modifiche alla funzione taglio;
- * D03: ottimizzazione *rendering* GPU¹.

2.4.3 Obiettivi facoltativi

- * F01: algoritmi di segmentazione con ITK;

¹Graphics Processing Unit, l'unità di elaborazione grafica

- * F02: analisi *unit-testing* su GUI-Qt²;
- * F03: *porting* librerie aziendali su CMake.

2.5 Contesto Applicativo

G-Squared Srl offre vari *software* e servizi per elaborare, archiviare e distribuire immagini radiologiche. Uno dei più importanti per il mio stage è EyeRad.



Figura 2.2: Logo EyeRad G-Squared

Fonte: gsquared.it

EyeRad è un sistema di acquisizione, analisi e gestione di immagini digitali provenienti da sistemi che aderiscono allo standard DICOM in ambiente Microsoft Windows. EyeRad è in grado di visualizzare, elaborare, archiviare, inviare e stampare immagini digitali. È nato per soddisfare le esigenze di efficienza e precisione richieste ad una *workstation* per la refertazione radiologica, avvalendosi dell'uso di monitor LCD ad alta risoluzione, 2-3-5 MPixel e sfruttando appieno la possibilità di tali monitor di visualizzare le immagini digitali con 2048 toni di grigio.

Nel caso del mio stage non ho dovuto modificare EyeRad, ma è stato molto interessante vedere come funziona, com'è strutturato e com'è mantenuto quando sono necessarie delle modifiche o delle correzioni.

2.6 Tecnologie Utilizzate

2.6.1 C++

C++ è un linguaggio di programmazione *general-purpose*, sviluppato come evoluzione del linguaggio C inserendo la programmazione orientata agli oggetti. Col tempo ha avuto notevoli evoluzioni, come l'introduzione dell'astrazione rispetto al tipo. È il linguaggio principale usato da G-Squared per i *software* desktop, di conseguenza la sua scelta è stata praticamente obbligatoria. Si è deciso inoltre di utilizzare lo standard C++17, in modo da poter usufruire, se necessario, di tutte le novità presenti nel linguaggio come le lambda-espressioni.

2.6.2 Qt5

Qt è un *framework open-source* multipiattaforma per lo sviluppo di programmi con interfaccia grafica tramite l'uso di *widget*. È il *framework* principale utilizzato da G-Squared con C++ per sviluppare interfacce grafiche, ed è stato quindi il *framework* che ho utilizzato per sviluppare il *widget*. È stato utilizzato con VTK, *software open-source* multipiattaforma per la computer grafica 3D, di cui parleremo nel capitolo 3.

²Graphic User Interface, l'interfaccia grafica per l'utente di un *software*

2.7 Strumenti Utilizzati

2.7.1 Visual Studio

Microsoft Visual Studio, chiamato più comunemente Visual Studio, è un ambiente di sviluppo integrato (Integrated development environment o IDE) sviluppato da Microsoft. Visual Studio è multi-linguaggio, ma è stato utilizzato con C++ con il compilatore di Microsoft chiamato MSVC.

2.7.2 Qt Creator

Qt Creator è un IDE *open-source*, che semplifica lo sviluppo di applicazioni con una GUI. Fa parte dell'SDK di Qt ed è ben integrato con tutti i relativi strumenti. Permette la scelta del compilatore, che nel mio caso è stato MSVC, in modo da utilizzare lo stesso compilatore tra Visual Studio e Qt Creator.

È il principale IDE utilizzato da G-Squared, quindi era importante verificare che tutto funzionasse anche su Qt Creator.

Entrambi gli IDE, sia Visual Studio che Qt Creator, permettono di aprire direttamente un progetto Cmake da un file *CMakeLists.txt*.

2.7.3 CMake

CMake è un software *open-source* multipiattaforma per l'automazione dello sviluppo, il cui nome è un'abbreviazione di "cross platform make". Questo software nasce per fornire un sistema di *build* indipendente dal compilatore, puntando ad essere semplice e modulare da usare. È stato scelto anche per mostrare all'azienda come poter compilare e gestire un progetto C++ con Qt, sostituendo il sistema di generazione di *default* utilizzato da Qt chiamato qmake.

2.7.4 Git

Git è un software di controllo di versione distribuito, utilizzato per fare il versionamento e l'upload del codice nel repository aziendale. Non sono state utilizzate regole particolari come i *feature branch*.

2.8 Aspettative personali

Come descritto nella sezione [Scelta dello stage \(§2.1\)](#), ero alla ricerca di uno stage che mi permetesse di esplorare meglio gli ambiti e gli utilizzi della grafica *real-time*. Inoltre, mi sarebbe piaciuto utilizzare e migliorare le mie conoscenze di C++. Proprio per questi motivi, tra i vari colloqui effettuati, la mia scelta è ricaduta su G-Squared. I miei obiettivi da raggiungere in questo progetto di stage quindi erano:

- * imparare come vengono archiviate e gestite le immagini mediche e gli standard utilizzati, come DICOM;
- * imparare le basi del *volume rendering* utilizzato in ambito medico;
- * migliorare le mie conoscenze e abilità nell'utilizzare C++ e gli strumenti correlati.

Capitolo 3

Volume Rendering e Qt

3.1 Radiologia e rendering

3.1.1 Visualizzare informazioni

Visualizzare fa parte della nostra vita quotidiana: dalle mappe satellitari alla computer grafica dell'industria dell'intrattenimento, possiamo trovare esempi di visualizzazione praticamente ovunque. Ma che cosa significa visualizzare? Informalmente, visualizzare è la trasformazione di dati o informazioni in immagini. Visualizzare coinvolge la vista e le capacità cognitive di chi guarda. Visualizzare è un modo semplice ed efficace per comunicare informazioni complesse e/o voluminose.

L'aumento della capacità di calcolo dei computer moderni ci fornisce tecniche di visualizzazione che alcuni anni fa sarebbero state inimmaginabili, come nella medicina moderna, ambito su cui ci concentreremo.

3.1.2 Volumi radiologici

Le tecniche di diagnostica per immagini, soprattutto in radiologia, sono diventate un importante strumento nella medicina moderna. Ci concentreremo su tecniche come la tomografia computerizzata e la risonanza magnetica.

Queste tecniche utilizzano dei processi di acquisizione e campionamento per raccogliere informazioni sull'anatomia interna di un paziente. Tali informazioni sono raccolte in forma di sezioni trasversali del corpo di un paziente, in maniera simile a come accade per radiografie a raggi X convenzionali. La TC utilizza un fascio di raggi X per acquisire i dati, mentre la RM utilizza un forte campo magnetico unito a impulsi a radiofrequenza. Varie tecniche matematiche vengono utilizzate per ricostruire le sezioni ricavate dallo scanner, dopodiché solitamente queste vengono raccolte in un insieme di dati.

Un'immagine radiologica, o una sezione del volume nel nostro caso, è acquisita come una serie di valori che rappresenta l'attenuazione dei raggi X (nella TC) o il rilassamento dello spin di un atomo (nella RM). Ogni immagine contiene tutti questi dati in un *array*, o in una matrice, tuttavia la quantità di dati è così grande che è impossibile comprenderli nella loro forma "grezza". Per questo, assegnandogli una scala di grigi e visualizzandoli su uno schermo, si riesce finalmente a visualizzare la struttura, permettendoci di visualizzare come una sezione del corpo umano ciò che il computer vede come un insieme di valori.

3.1.3 Standard DICOM

DICOM è uno standard che definisce i criteri per la comunicazione, la visualizzazione, l'archiviazione e la stampa di informazioni di tipo biomedico quali ad esempio immagini radiologiche. La sua diffusione si rivela estremamente vantaggiosa perché consente di avere una solida base di interscambio di informazioni tra apparecchiature di diverso tipo e di diversi produttori.

DICOM raggruppa le informazioni in un *dataset*: ad esempio, un file di un'immagine radiografica del torace può contenere l'ID paziente all'interno del *file*, in modo che l'immagine non possa mai essere separata per errore da queste informazioni. Un oggetto DICOM è costituito da una serie di attributi, inclusi elementi come nome, ID, ecc. e anche un attributo speciale contenente i dati dei pixel dell'immagine. Un singolo oggetto DICOM può avere un solo attributo contenente pixel, per molti casi ciò corrisponde a una singola immagine. Tuttavia, l'attributo può contenere più *frame*, consentendo la memorizzazione di dati *multi-frame*, come un esame TC o RM. In questi casi, i dati tridimensionali o quadridimensionali possono essere incapsulati in un singolo oggetto DICOM. I dati dei pixel possono essere compressi, ma ciò viene fatto raramente.

3.1.4 Basi di rendering

La computer grafica è il processo che genera immagini utilizzando il computer, questo processo viene chiamato *rendering*. Ci sono molti tipi di *rendering*, dal disegno 2D a tecniche 3D sofisticate. In questa sezione vedremo le basi del *rendering* 3D.

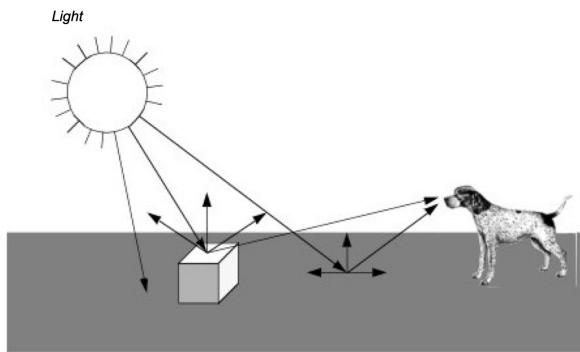
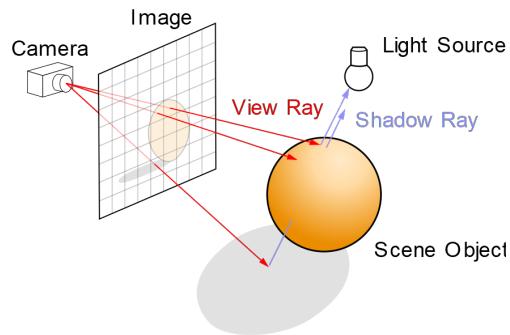


Figura 3.1: La luce e la vista
Fonte: [VTKBook/Chapter3](#)

Nel mondo reale quando guardiamo un oggetto, per esempio una scatola, i raggi di luce emessi da una sorgente luminosa (per esempio il sole) sono emessi in tutte le direzioni. Alcuni di questi colpiscono la scatola, che assorbe una parte di luce e ne riflette il resto. Una parte di questa luce riflessa potrebbe dirigersi verso di noi ed entrare nei nostri occhi; se questo succede noi riusciamo a vedere l'oggetto. Allo stesso modo, se della luce colpisce il terreno, una parte si rifletterà nei nostri occhi.

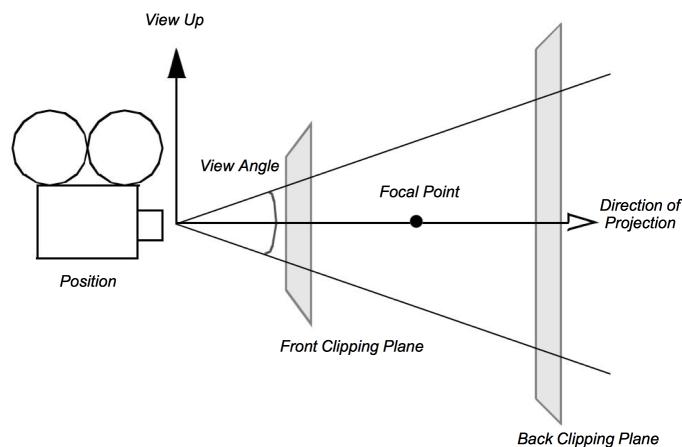
Una tecnica comune ed efficace per la computer grafica 3D è il *ray tracing*, in origine chiamato *ray casting*. Il *ray tracing* simula l'interazione della luce con gli oggetti, seguendo il percorso di ogni raggio. Di solito si segue il raggio all'indietro, dalla posizione dell'osservatore (la camera nello schema della figura 3.2) per determinare che cosa il raggio colpisce. La direzione del raggio, quindi, è la direzione che si sta

**Figura 3.2:** Algoritmo di Ray TracingFonte: [wikipedia.org](https://en.wikipedia.org)

osservando. Quando un raggio colpisce un oggetto, possiamo determinare se quel punto è illuminato da una sorgente di luce: questo viene fatto tracciando un raggio dal punto di intersezione alla luce; se il raggio colpisce qualcosa prima di raggiungere la sorgente luminosa, allora quella luce non contribuirà a illuminare il punto. Se ci sono N sorgenti luminose, questo andrà fatto per ognuna di esse.

Per chi non avesse mai sentito parlare del *ray tracing*, sarà sorprendente scoprire che non è quasi mai usato nella grafica *real-time*: questo perché il *ray tracing* è un processo molto lento e dispendioso in termini di risorse, oltre al fatto che fino a pochi anni fa era possibile implementarlo solo via *software*. Per questo sono state sviluppate altre tecniche che generano immagini sfruttando meglio l'accelerazione *hardware*.

3.1.5 Camera

**Figura 3.3:** Proprietà della cameraFonte: [VTKBook/Chapter3](https://www.vtkbook.com/Chapter3.html)

Un oggetto essenziale per fare il *render* di una scena, è la cosiddetta *camera*, che potremmo chiamare "visuale" in italiano.

Ci sono vari fattori da considerare per determinare come una scena 3D viene proiettata su un piano 2D per generare un'immagine: la posizione, l'orientamento e il punto

focale della *camera*, il metodo di proiezione utilizzato e la posizione dei *Clipping Planes*. La posizione e il punto focale della *camera* definiscono dove si trova e dove punta. Il vettore definito dalla posizione della *camera* al punto focale viene comunemente chiamato *direction of projection* (direzione di proiezione). Il metodo di proiezione controlla come gli elementi della scena sono mappati sul piano dell'immagine.

La proiezione ortografica è un processo di mappatura parallelo: nella proiezione ortografica (o proiezione parallela) tutti i raggi di luce che entrano nella camera sono paralleli al vettore di proiezione. La proiezione prospettica si verifica quando tutti i raggi di luce attraversano un punto comune (cioè il punto di vista o il centro di proiezione). Per applicare la proiezione prospettica dobbiamo specificare un angolo prospettico o angolo di vista della *camera*.

I *clipping planes front* e *back* intersecano il vettore di proiezione e generalmente gli sono perpendicolari; essi sono utilizzati per eliminare i dati troppo vicini o troppo lontani dalla camera. Il *front clipping plane* è al valore di intervallo minimo, mentre il *back clipping plane* è al valore di intervallo massimo.

3.1.6 Rendering di oggetti non solidi

La spiegazione fino a questo punto ha assunto che stessimo facendo il render di un oggetto solido. Tuttavia, oggetti come le nuvole, l'acqua e la nebbia sono "traslucidi" o comunque diffondono la luce che li attraversa. Non si può fare il *render* di questi oggetti utilizzando esclusivamente le interazioni sulle superfici, dobbiamo invece considerare le proprietà mutevoli all'interno dell'oggetto per mostrarlo correttamente. Ci riferiamo quindi a due modelli di *rendering*:

- * *surface rendering*: esegue il *render* della superficie di un oggetto;
- * *volume rendering*: esegue il *render* della superficie e dell'interno di un oggetto.

Le tecniche di *volume rendering* ci consentono di vedere la "disomogeneità" all'interno degli oggetti. Nella TC per esempio, possiamo riprodurre realisticamente immagini a raggi X considerando i valori di intensità sia sulla superficie che all'interno. Vedremo più dettagli sul *volume rendering* nella prossima sezione, ma tornando all'esempio del *ray tracing*, si può immaginare come i raggi non interagiscano solo con la superficie di un oggetto, ma anche con ciò che è al suo interno.

3.2 Volume rendering

3.2.1 Basi di volume rendering

Finora ci siamo concentrati sulla visualizzazione di dati tramite l'utilizzo di primitive come punti, linee e poligoni. Per molte applicazioni questo è chiaramente il metodo migliore per rappresentarli, tuttavia alcune applicazioni ci richiedono di visualizzare dati che sono "volumetrici", più comunemente chiamati immagini 3D o *volume datasets*. Per esempio, nell'*imaging* biomedico potremmo aver bisogno di visualizzare dati ottenuti da una RM, una TC, un microscopio o un'ecografia. Anche l'analisi meteorologica e altre simulazioni producono grandi quantità di dati volumetrici in tre o più dimensioni che richiedono tecniche di visualizzazione efficaci.

Vedremo ora più in dettaglio i principali metodi di *volume rendering* utilizzati in ambito biomedico, chiaramente ce ne potrebbero essere altri, ma noi ci concentreremo sui metodi utilizzati dalla libreria VTK, utilizzata durante lo stage.

Considerando che il *rendering* volumetrico è tipicamente usato per generare immagini che rappresentano un intero set 3D proiettato in un'immagine 2D, bisogna prestare attenzione ad alcuni punti: una classificazione deve essere eseguita per assegnare colore e opacità alle regioni all'interno del volume, e devono essere definite delle tecniche di illuminazione volumetrica per migliorare il risultato, tuttavia in questo progetto saranno solo menzionate.

3.2.2 Voxel

Il termine *voxel* denota un singolo elemento del volume, simile ad un pixel per un'immagine. Ogni *voxel* corrisponde a una posizione nello spazio dati ed ha uno o più valori di dati associati. È da notare che un *voxel* rappresenta solo un singolo punto sulla griglia tridimensionale, non un volume; lo spazio tra ogni *voxel* non è rappresentato in un set di dati e i valori nelle posizioni intermedie si ottengono interpolando i dati sugli elementi del volume adiacenti. Questo processo è noto come ricostruzione e svolge un ruolo importante nel *rendering* del volume e nelle applicazioni di elaborazione.

3.2.3 Volume rendering Image-Order

Gli algoritmi di *volume rendering Image-Order* iterano sui pixel dell'immagine da produrre, anziché sugli elementi della scena. Il *ray tracing* è un esempio di algoritmo di tipo *Image-Order*, dove l'idea di base è determinare il valore di ogni pixel dell'immagine, inviando un raggio dalla posizione del pixel nella scena, secondo i valori della camera. A quel punto si valutano i dati incontrati lungo il raggio, per calcolare il valore del pixel. Come vedremo, il *ray tracing* è una tecnica che può essere usata per fare il *render* di qualsiasi *dataset* 3D producendo una grande varietà di immagini, ed è relativamente semplice estenderlo in modo da utilizzarlo per un set di dati volumetrici per lavorare su "griglie" ben strutturate. Sfortunatamente, il *ray tracing* è un processo abbastanza lento, pertanto si utilizzano una serie di metodi di accelerazione per migliorare le prestazioni, sacrificando della memoria aggiuntiva o parte di flessibilità dell'algoritmo.

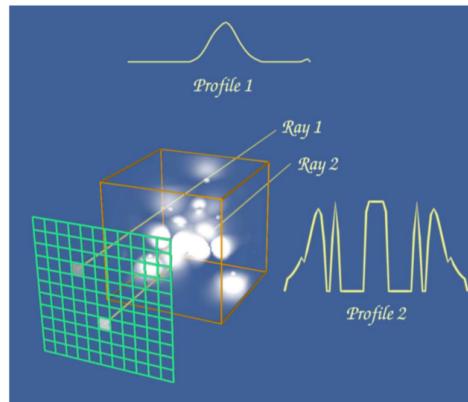


Figura 3.4: Volume rendering Image-Order
Fonte: VTKBook/Chapter7

Un esempio del processo di *ray tracing* è illustrato nell'immagine 3.4: questo esempio utilizza una proiezione ortografica della *camera*, di conseguenza tutti i raggi sono paralleli l'uno all'altro e perpendicolari alla vista (anche chiamata *view plane*). I

dati processati lungo ciascun raggio sono processati con una funzione specifica, che in questo caso determina il valore massimo incontrato e lo converte ad una scala di grigi, dove il valore minimo è mappato come nero trasparente, e il massimo valore è mappato come bianco opaco. Le prime due funzioni, chiamate *maximum value* e *average value*, sono operazioni di base sui valori scalari stessi. Uno dei metodi più semplici per visualizzare un *dataset* volumetrico è la *maximum intensity projection* (MIP) che considera i *voxel* con la massima intensità incrociati dai raggi del *ray tracing*. La MIP è utilizzata per esempio per la rilevazione di noduli polmonari nei programmi di screening del cancro del polmone effettuati tramite scansione TC.

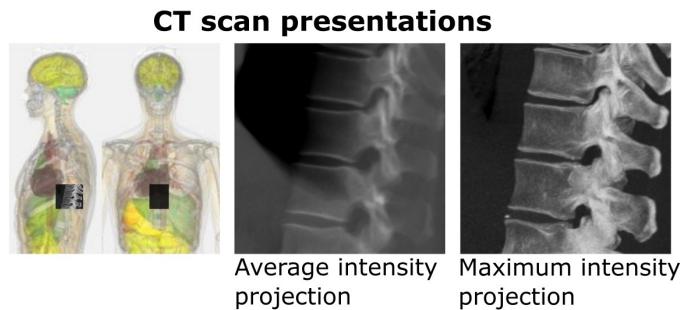


Figura 3.5: Esempio di "average value" e di MIP

Fonte: [wikipedia.org](https://en.wikipedia.org)

3.2.4 Volume rendering Object-Order

Il *volume rendering Object-Order* processa i valori nel volume basandosi sull'organizzazione dei *voxel* nel *dataset* e sulle impostazioni della visuale corrente. Quando un metodo *alpha compositing* è utilizzato, i *voxel* vanno processati in ordine *front-to-back* o *back-to-front* per ottenere risultati corretti (*alpha compositing* è il processo di combinazione di un'immagine con uno sfondo per creare l'aspetto di una trasparenza parziale o totale). Se si sfrutta l'*hardware* dedicato, è preferibile utilizzare l'ordine *back-to-front*, in quanto si evita di calcolare dei valori extra riguardo la trasparenza. Al contrario, se si utilizza il *render software*, l'ordine *front-to-back* è più comune in quanto si ottengono risultati visivamente più significativi e perché è possibile evitare di processare ulteriori dati quando un pixel raggiunge la "piena opacità". Alcune tecniche, come la MIP, possono essere processate in qualsiasi ordine per ottenere risultati corretti.

La figura 3.6 rappresenta un approccio *back-to-front* per calcolare i *voxel*: si parte dal *voxel* più lontano rispetto alla visuale, e si prosegue visitandoli in ordine di distanza finché non sono stati tutti visitati.

3.2.5 Funzione di trasferimento

La funzione di trasferimento è responsabile della mappatura delle informazioni dei *voxel* in valori differenti come materiale, colore o trasparenza. Uno dei punti di forza del *volume rendering* è che può gestire funzioni di trasferimento di complessità molto maggiore di una funzione "a passo binario". Questo è spesso necessario considerando che i *dataset* contengono più materiali e un metodo di classificazione non può sempre riuscire ad assegnare un singolo materiale ad un campione.

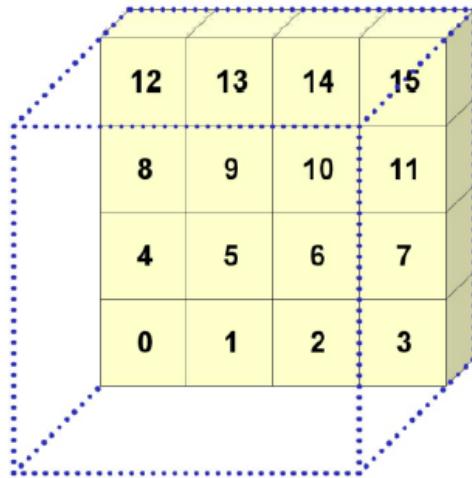


Figura 3.6: Volume rendering Object-Order
Fonte: VTKBook/Chapter7

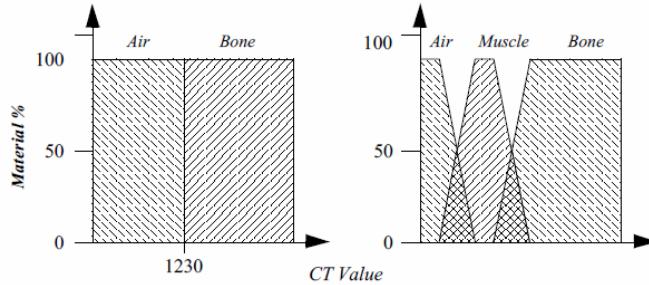


Figura 3.7: Semplice classificazione binaria (sinistra) e una transizione graduale tra aria, muscoli e ossa (destra).

Fonte: VTKBook/Chapter7

Prendendo come esempio una TC, ora possiamo specificare una funzione di trasferimento che definisca una transizione graduale da aria, a muscoli, ad ossa, come mostrato nell'immagine 3.7.

3.2.6 Regioni di interesse

Un problema nel visualizzare dati volumetrici, è che se volessimo studiare alcuni dati al centro del volume, dovremmo guardare a tutto ciò che c'è attorno nel *dataset*. Per esempio, se visualizzassimo il *dataset* di un pomodoro, non riusciremmo a vedere i semi perché, con tecniche come la MIP, vedremmo tutta la polpa circostante.

Possiamo risolvere questo problema di visualizzazione interna definendo una regione di interesse del nostro volume, e facendo il render quindi solo di una porzione del *dataset* come mostrato nella figura 3.8. Ci sono molte tecniche per definire una regione di interesse: si potrebbero utilizzare i *clipping planes near/far* della *camera* per escludere parti del volume; altrimenti si possono definire sei piani (un cubo quindi) per definire un sotto-volume da visualizzare, escludendo il resto all'esterno del cubo; si possono anche utilizzare più piani distinti per escludere sezioni in varie posizioni e orientamenti.

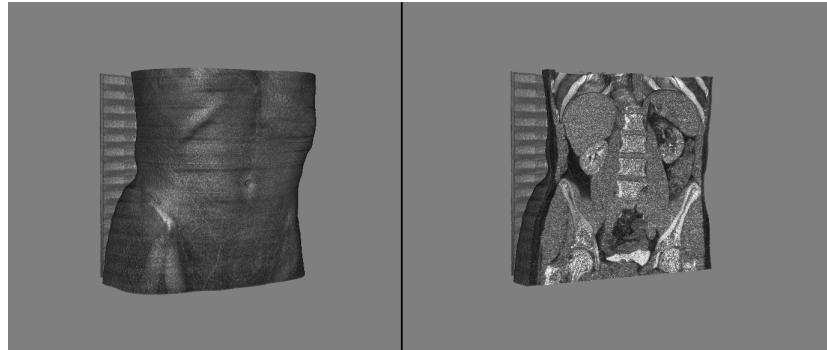


Figura 3.8: Esempio di volume rendering con regione di interesse, tagliando il volume.

Fonte: Stage

3.3 VTK



Figura 3.9: Logo di VTK

Fonte: vtk.org

3.3.1 Descrizione e scelta della libreria

Il Visualization Toolkit (VTK) è un *software open-source* multipiattaforma per la computer grafica 3D, l'elaborazione delle immagini e la visualizzazione scientifica. VTK consiste in una libreria C++ e vari layer di interfaccia, per esempio per Java o Python. VTK è sviluppato e supportato dal team Kitware, lo stesso team che ha sviluppato CMake, lo strumento di *build* nominato in precedenza. VTK supporta vari algoritmi di visualizzazione, e oltre al *framework* di visualizzazione contiene gli strumenti per interagire in 3D con un oggetto, supporta l'elaborazione parallela e si integra con vari *database* e *toolkit GUI* come Qt.

La libreria è stata proposta dal *tutor* dello stage, che l'aveva studiata e provata in passato, e voleva approfondirne le capacità e le funzionalità.

3.3.2 Build della libreria

VTK non offre dei binari precompilati per utilizzare la libreria; per utilizzarla, quindi, bisognerà compilarla manualmente nella propria macchina. Questo è utile anche perché permette di configurare i moduli necessari da compilare e quelli da escludere. Il modo più semplice e raccomandato per fare la *build* della libreria è CMake, nato proprio per supportare lo sviluppo di VTK e ITK. Con CMake possiamo quindi, per esempio, escludere dalla compilazione la documentazione, gli esempi, il supporto a CUDA e l'integrazione con Java e Python, tutti moduli non necessari per il nostro progetto.

Dobbiamo tuttavia specificare che vogliamo usare Qt, in modo da compilarne il modulo che lo supporta. Parleremo meglio di Qt nella sezione [Integrazione con Qt](#) ([§3.3.7](#)). Una volta configurato e generato il progetto con CMake, sarà sufficiente compilarlo per ottenere i file librerie da utilizzare nel proprio programma, configurando appositamente il *linker C++*.

3.3.3 Oggetti di rendering

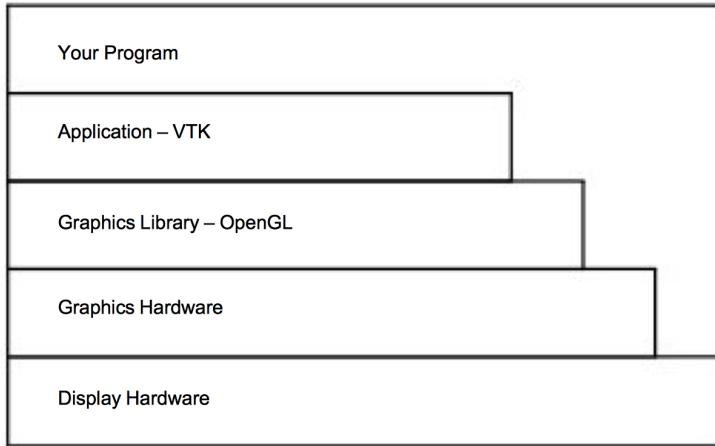


Figura 3.10: Gerarchia dell'applicazione

Fonte: [VTKBook/Chapter3](#)

Ora che abbiamo discusso le basi di *rendering* e di VTK, la buona notizia è che raramente dovremo preoccuparci di implementarle o modificarle. La maggior parte della programmazione grafica viene eseguita utilizzando primitive: la figura 3.10 ci mostra la gerarchia di visualizzazione. Al primo posto c'è il nostro programma, e nei tre livelli sottostanti ci sono i livelli da tenere in considerazione.

VTK contiene molti oggetti che utilizziamo per fare il render di una scena, molti sono dietro le quinte, ma vediamo i principali mostrati anche nella figura 3.11:

- * *vtkRenderWindow*: gestisce una finestra sul dispositivo di visualizzazione, uno o più *renderer* disegnano in un'istanza di *vtkRenderWindow*;
- * *vtkRenderer*: controlla il processo di *rendering* degli oggetti, processando gli attori nella scena, le luci e la vista, in un'immagine;
- * *vtkLight*: una sorgente di luce che illumina la scena;
- * *vtkCamera*: definisce la posizione della *camera*, il punto focale, e altre proprietà di visualizzazione della scena;
- * *vtkActor*: rappresenta un oggetto nella scena di cui fare il *render*, comprese le sue proprietà e la sua posizione;
- * *vtkProperty*: definisce le proprietà di un *vtkActor*, tra cui il colore, la trasparenza, e le proprietà luminose;
- * *vtkMapper*: definisce la rappresentazione geometrica di un attore, più attori quindi possono riferirsi allo stesso *vtkMapper*.

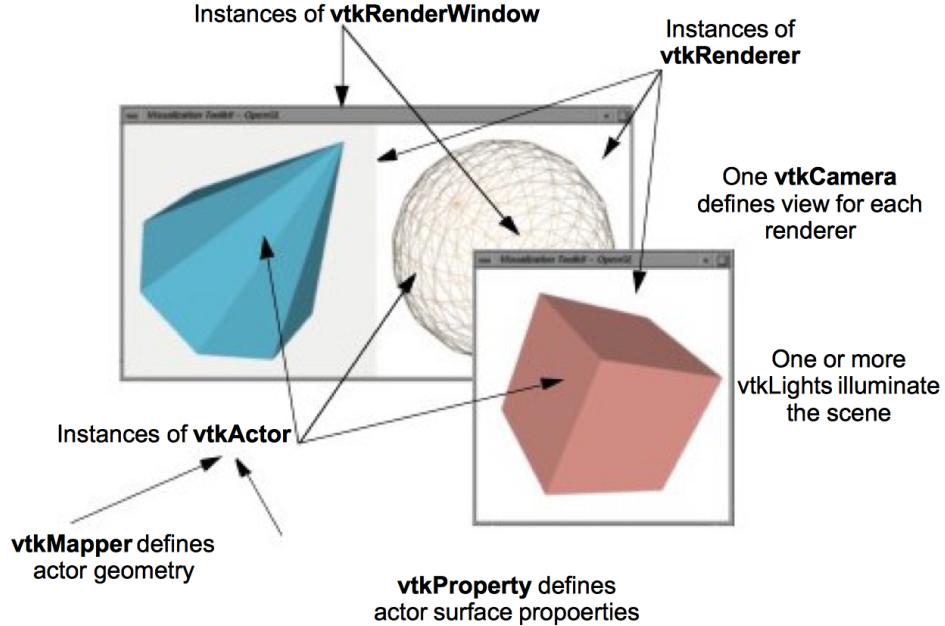


Figura 3.11: Oggetti di rendering di VTK
Fonte: [VTKBook/Chapter3](#)

3.3.4 Particolari oggetti di rendering

Dopo aver visto gli oggetti principali di VTK, diamo un’occhiata più in dettaglio ad alcuni oggetti. VTK fa ampio uso di *Smart Pointer*, facilitando il passaggio e la cancellazione dei puntatori inutilizzati: è disponibile il tipo *vtkSmartPointer*<T> per definire uno smart pointer.

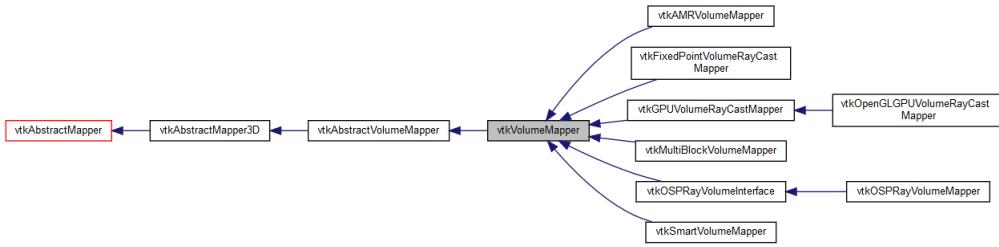
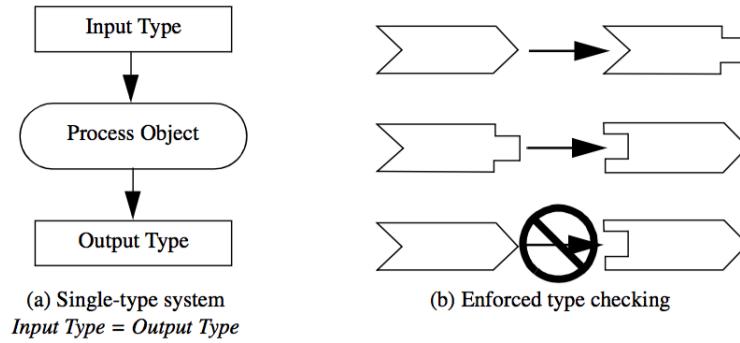
Abbiamo menzionato l’oggetto *vtkMapper*, ma volendo lavorare sul *volume rendering*, ci interessano gli oggetti implementati dalla classe astratta *vtkVolumeMapper*, di cui possiamo vedere una gerarchia nella figura 3.12. Ci sono chiaramente alcune differenze tra gli oggetti mostrati, ma verranno discusse nel prossimo capitolo.

Tutte le proprietà principali di un volume sono definite nell’oggetto *vtkVolumeProperty*, che contiene per esempio le funzioni di trasferimento, definite principalmente utilizzando l’oggetto *vtkPiecewiseFunction*. Queste proprietà possono essere assegnate ad un *vtkVolume*, oggetto principale di gestione di un volume presente nella scena.

3.3.5 Pipeline di rendering

In VTK, gli elementi della pipeline (sorgenti, filtri e *mapper*) possono essere connessi in vari modi per creare una specifica visualizzazione. Tuttavia, ci sono due fattori da considerare: il tipo e la molteplicità.

Con tipo si intende il tipo di dato che un oggetto accetta come input o genera

**Figura 3.12:** Oggetti VolumeMapper di VTKFonte: vtk.org**Figura 3.13:** Struttura oggetti della PipelineFonte: [VTKBook/Chapter4](#)

come output: un oggetto sfera potrebbe generare come *output* una rappresentazione poligonale o una rappresentazione implicita (ad esempio, parametri di un'equazione conica). Un oggetto *mapper* potrebbe prendere come input una rappresentazione poligonale o un una rappresentazione geometrica di punti. L'*input* in un oggetto deve essere specificato e utilizzato correttamente per funzionare.

Il problema della molteplicità si riferisce al numero di dati di input permessi e al numero di oggetti di output creati da un oggetto, per esempio da un filtro. Questo non ci riguarda particolarmente, visto che lavoreremo sempre su un volume singolo, quindi con un *input* e un *output* per ogni filtro.

3.3.6 Widget e interazione utente

Oltre a visualizzare vari elementi, l'interazione è una *feature* essenziale per permettere di analizzare meglio ciò che si sta mostrando. VTK contiene classi come *vtkRenderWindowInteractor* e *vtkInteractorStyle* che catturano eventi della finestra e li traducono in eventi utilizzabili da VTK. Sono utili principalmente per manipolare la visuale o gli attori (per esempio ruotandoli) per ottenere una vista desiderata. Tuttavia, questa funzionalità è abbastanza limitata riguardo l'interazione con i dati non permettendoci, per esempio, di definire una regione di interesse.

Per questo sono stati introdotti i *widget* 3D, che sono in grado di fornire la varietà di tecniche di interazione necessarie in un sistema di visualizzazione di questo tipo. In VTK gli eventi sono catturati da *vtkRenderWindow*, gli osservatori registrati ricevono gli eventi ed eseguono delle azioni, processando l'evento o passandolo ad un altro

osservatore.

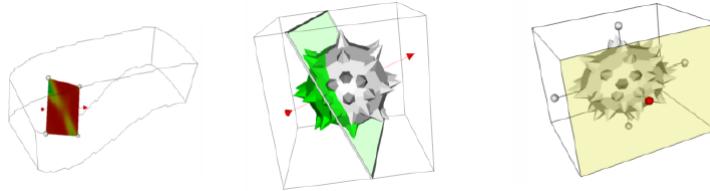


Figura 3.14: Esempi di widget 3D di VTK
Fonte: [VTKBook/Chapter7](#)

Nella figura 3.14 vediamo alcuni esempi di *widget* 3D, in ordine un *vtkPlaneWidget*, *vtkImplicitPlaneWidget* e un *vtkBoxWidget*. È importante che i *widget* siano disegnati in modo da essere intuitivi: per esempio i sei lati che delimitano il *vtkBoxWidget* sono rappresentati da sei piccole sfere, che possono essere selezionate e trascinate per ridimensionarlo. È possibile estendere *vtkCommand* per catturare la *callback* di un *widget* 3D ed utilizzarne le proprietà, come tagliare il volume in base alla nuova dimensione del *vtkBoxWidget*, ma vedremo i dettagli su come questo è stato implementato nel capitolo 4.

3.3.7 Integrazione con Qt

Nel mio caso, dovevo utilizzare VTK con Qt, in modo da tenere Qt come libreria principale di gestione della UI (user interface, interfaccia grafica). Il primo passo, come spiegato, è di compilare VTK con il supporto a Qt: questo richiede un'installazione già presente di Qt in modo da indicare a VTK quale versione e quali file di Qt utilizzare. Io ho utilizzato Qt 5.15, l'ultima versione disponibile durante lo stage.

Una volta preparato VTK, sarà quindi possibile creare un *widget* Qt che deriva da *QVTKOpenGLNativeWidget*, ottenendo un *widget* apposito su cui è possibile mostrare il *render* effettuato da VTK.

3.4 Strumenti CTK



Figura 3.15: Esempi di widget 3D di CTK
Fonte: [commontk.org](#)

Il Common Toolkit (CTK) è una serie di strumenti di supporto per *imaging* medico e scopi correlati, come le applicazioni DICOM. Offre per esempio degli strumenti per interagire con i *database* medicali DICOM, una serie di *plugin* e strumenti a riga di comando e una serie di *widget* 3D per facilitare l'interazione con VTK, come si può vedere nell'immagine 3.15, che mostra un pannello per definire le proprietà dei

materiali, un *double-range-slider* e un pannello per modificare le funzioni, in questo caso la funzione di *threshold*.

Anche se la serie di strumenti offerti da CTK è stata studiata e analizzata, non è stata attivamente utilizzata come libreria durante lo stage.

3.5 Basi di ITK

L'Insight Segmentation and Registration Toolkit (ITK) è un *framework open-source* multipiattaforma, utilizzato per lo sviluppo di programmi di segmentazione o "registrazione" (*registration*) immagini. La segmentazione è il processo di identificazione e classificazione dei dati trovati in una rappresentazione campionata digitalmente, nel nostro caso principalmente da un'immagine acquisita da strumentazione medica come scanner TC o RM, permettendo per esempio di escluderne una grande porzione non rilevante, di conseguenza anche velocizzandone il *rendering*. La registrazione è il compito di allineare o sviluppare corrispondenze tra i dati: ad esempio, in ambiente medico, una scansione TC può essere allineata con una scansione RM per combinare le informazioni contenute in entrambe.

ITK era già nel piano di lavoro definita come una libreria da considerare solo optionalmente per eventuali lavori futuri, e non è stata utilizzata durante lo stage.

3.6 Software correlati

3.6.1 3D Slicer

Un *software* importante che è stato preso in considerazione durante lo stage è 3D Slicer, un *software open-source* per l'analisi delle immagini e la visualizzazione scientifica, che fa ampio utilizzo di VTK, CTK e ITK. È stato preso come principale *software* di riferimento, sia per capire come utilizza VTK e le sue funzionalità, sia come fonte di ispirazione riguardo i *tool* necessari per interagire con un volume. Per esempio: la lista in formato XML dei *preset* delle funzioni di trasferimento, è presa dal *repository* di 3D Slicer, e ne è stato fatto un *parser* apposito, considerando che sono funzioni standard usate in tutti i *software* di visualizzazione medica.

3.6.2 MITK e SimVascular

3D Slicer è solo un esempio dei tanti *software* per l'analisi di immagini scientifiche/mediche. Un altro esempio è il Medical Imaging Interaction Toolkit (MITK), un sistema *software open-source* per lo sviluppo di *software* per l'elaborazione di immagini mediche, che combina ITK e VTK in un'unica applicazione. Una caratteristica particolare di MITK è che include il "Nvidia AI-Assisted Annotation", utilizzando la IA e il *deep learning* per aiutare ad automatizzare l'elaborazione e la comprensione delle immagini medicali.

Un altro esempio è SimVascular, che dalla segmentazione delle immagini mediche offre una *pipeline* di sviluppo per analizzare e simulare il flusso di sangue di un paziente, come si può vedere da un esempio nell'immagine [3.16](#).

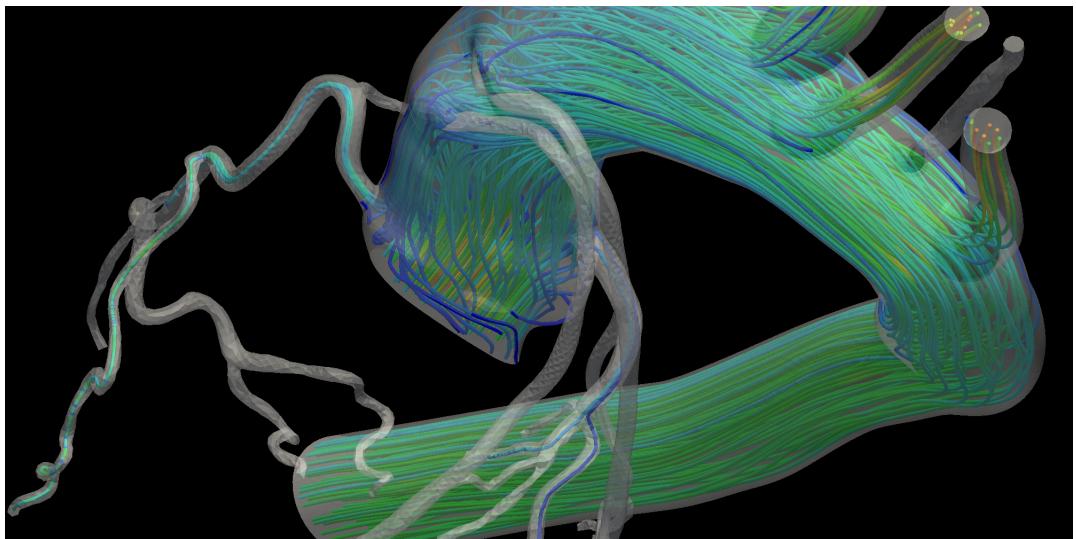


Figura 3.16: Simulazione del flusso di sangue in una "Coronary Artery Bypass Simulation (CABG)"

Fonte: simvascular.github.io

Capitolo 4

Resoconto Stage

4.1 Pianificazione

4.1.1 Pianificazione iniziale

Una prima pianificazione delle attività è stata fatta da me insieme al *tutor* aziendale prima dell'inizio dello stage, principalmente definendo e redigendo il piano di lavoro, documento necessario per iniziare lo stage e per avere un'idea delle tempistiche necessarie al raggiungimento degli obiettivi prefissati. Durante lo svolgimento dello stage tuttavia, la pianificazione originaria ha subito delle piccole modifiche, in quanto alcune attività hanno richiesto più tempo di altre.

4.1.2 Sistema di issue e Gantt

Per essere sempre entrambi aggiornati sullo stato dei lavori, il mio *tutor* mi ha suggerito di utilizzare il sistema di *issue* ampiamente utilizzato in azienda, basato su Redmine. Abbiamo quindi riportato gli obiettivi principali definiti nel piano di lavoro, in modo da tenere traccia di quali erano stati completati e di una stima del tempo richiesto: è infatti possibile segnare una stima delle ore utilizzate per una certa attività. Redmine è inoltre integrato con il git aziendale, permettendo quindi di fare riferimento a specifici commit e di scrivere una *wiki* relativa al progetto.

Impostando correttamente le date e le priorità tra le attività da svolgere nel Redmine aziendale, era anche possibile generare automaticamente un diagramma Gantt per visualizzare in maniera più semplice informazioni riguardo ogni attività, quali: lo stato, la scadenza e i tempi impiegati, per meglio comprendere i possibili ritardi. Essendo questo un progetto relativamente piccolo, tuttavia, la gestione delle ore su Redmine è stata utilizzata poco, ma mi è stata molto utile per imparare come viene gestito un progetto più ampio in azienda.

4.1.3 Discussioni e incontri con il tutor

Lavorando principalmente da casa, discutere spesso con il *tutor* e aggiornarlo sullo stato del progetto e su eventuali dubbi/problemi è stata una parte fondamentale del mio stage. Per questo facevamo una videochiamata al giorno (salvo impegni), per discutere lo stato del lavoro e i successivi passi da fare. Inoltre, ci incontravamo in ufficio almeno una o due volte a settimana, il che mi permetteva di mostrargli di

persona l'applicazione e fare delle piccole dimostrazioni su cosa era stato fatto e in che modo, e anche di discutere su come migliorare alcuni dettagli o come risolvere alcuni problemi.

4.1.4 Problemi e ritardi

Come accennato nella sezione [Pianificazione iniziale](#) (§4.1.1) e come vedremo più nel dettaglio tra poco, la pianificazione iniziale ha subito delle piccole modifiche, in quanto alcune attività hanno richiesto più tempo. Questo non è stato un grosso problema perché, viceversa, alcune attività hanno richiesto meno tempo: i problemi principali sono stati nell'utilizzo di CMake e nell'import di un volume DICOM attraverso le librerie aziendali, mentre lo sviluppo della GUI e della *pipeline* di *rendering* ha richiesto meno tempo di quanto pianificato.

4.2 Implementazione

Vediamo ora nel dettaglio ed in ordine cronologico i passi effettuati durante lo stage per implementare ed utilizzare i concetti discussi nel capitolo 3, utilizzati per sviluppare l'applicazione. Non entrerò nel dettaglio di com'è stata implementata ogni cosa, in quanto il codice appartiene all'azienda.

4.2.1 Impostazione ambiente di sviluppo

La prima settimana era dedicata all'introduzione del progetto: dovevo quindi impostare l'ambiente di sviluppo, discutere con il *tutor* le modalità di lavoro, gli strumenti, configurare il git aziendale e studiare le librerie che avrei utilizzato, tutti passi preliminari e necessari ad un corretto svolgimento dello stage. Dopo aver impostato tutti gli strumenti, come Visual Studio con il compilatore corretto, aver installato l'ultima versione di Qt (5.15) e di QtCreator, il primo passo è stato scaricare e studiare le librerie che sarebbero state utilizzate. Ho quindi iniziato con uno studio preliminare di 3D Slicer, per imparare le basi di come importare un volume DICOM, come viene utilizzato il *volume rendering* e concetti simili, prima di scaricare e analizzare anche VTK.

4.2.2 Studio di 3D Slicer

Per analizzare al meglio il funzionamento di 3D Slicer, ad inizio stage si era pensato di compilarlo dai sorgenti in modo da poterne effettuare il *debug*. Tuttavia, questo si è rivelato un processo molto arduo, in quanto la documentazione su come compilarlo non era molto precisa, e fare una *build* completa (con quasi tutti i moduli) ha richiesto anche più di 6 ore su un processore Intel i7 6700k composto da 4 core/8 thread a 4.2Ghz. Dopo qualche tentativo fallito quindi, l'idea è stata abbandonata in quanto non strettamente necessaria. 3D Slicer è stato quindi installato normalmente dal *setup* disponibile nel sito, ed il suo sorgente è stato sfogliato principalmente attraverso GitHub.

4.2.3 Studio e installazione di VTK

Anche VTK, come ogni libreria, non è perfetta: un problema che purtroppo ho notato subito è che c'è della documentazione molto aggiornata e altra molto data-

ta. Per esempio, cercando su Google: "How to build vtk", uno dei primi risultati è: vtk.org/Wiki/VTK/Building/Windows, pagina con ultimo aggiornamento ad aprile 2014 nel momento in cui questo documento è stato scritto. È comunque una documentazione valida per le basi, ma è chiaro che non può essere utilizzata attivamente. Discretamente meglio è la pagina vtk.org/Wiki/VTK/Configure_and_Build, aggiornata al 2017. Avendo già utilizzato CMake in passato, non ho avuto grossa difficoltà a compilare VTK, in quanto è davvero ben gestita e facile da configurare. CMake può essere utilizzato da linea di comando o da interfaccia grafica tramite CMake-gui, applicazione installata insieme a CMake. Per una libreria di queste dimensioni con molte proprietà da configurare, è molto più comodo utilizzare la GUI, che permette di visualizzare lo stato di tutte le variabili prima di generare il progetto da compilare, come si può vedere nell'immagine 4.1.

| Name | Value |
|------------------------------|--|
| BUILD_SHARED_LIBS | <input checked="" type="checkbox"/> |
| CMAKE_CONFIGURATION_TYPES | Debug;Release;MinSizeRel;RelWithDebInfo |
| CMAKE_CXX_MP_FLAG | <input checked="" type="checkbox"/> |
| CMAKE_CXX_MP_NUM_PROCESSORS | 8 |
| CMAKE_INSTALL_PREFIX | C:/Program Files (x86)/VTK |
| HDF5_BATCH_H5DETECT | <input type="checkbox"/> |
| HDF5_ENABLE_HDFS | <input type="checkbox"/> |
| HDF5_ENABLE_ROS3_VFD | <input type="checkbox"/> |
| HDF5_USE_PREGEN | <input type="checkbox"/> |
| INSTALL_CMAKE_PACKAGE_MODULE | <input type="checkbox"/> |
| INSTALL_DOCS | <input type="checkbox"/> |
| INSTALL_PKG_CONFIG_MODULE | <input type="checkbox"/> |
| NSIS_EXECUTABLE | NSIS_EXECUTABLE-NOTFOUND |
| Python2_EXECUTABLE | Python2_EXECUTABLE-NOTFOUND |
| QT_QMAKE_EXECUTABLE | D:/Programmazione/Qt/5.15.0/msvc2019_64/bin |
| Qt5Core_DIR | D:/Programmazione/Qt/5.15.0/msvc2019_64/lib/cmake/Qt5Core |
| Qt5Gui_DIR | D:/Programmazione/Qt/5.15.0/msvc2019_64/lib/cmake/Qt5Gui |
| Qt5Sql_DIR | D:/Programmazione/Qt/5.15.0/msvc2019_64/lib/cmake/Qt5Sql |
| Qt5Widgets_DIR | D:/Programmazione/Qt/5.15.0/msvc2019_64/lib/cmake/Qt5Widgets |
| Qt5_DIR | D:/Programmazione/Qt/5.15.0/msvc2019_64/lib/cmake/Qt5 |
| VTK_BUILD_DOCUMENTATION | <input type="checkbox"/> |
| VTK_BUILD_EXAMPLES | <input type="checkbox"/> |
| VTK_BUILD_SCALED_SOA_ARRAYS | <input type="checkbox"/> |
| VTK_BUILD_TESTING | OFF |

Figura 4.1: Alcuni parametri di configurazione di VTK su CMake-gui

Fonte: Stage

A parte qualche lacuna nella compilazione, VTK ha un'ottima documentazione sulle classi e offre una ampia gamma di esempi, ed è stato quindi molto interessante ed utile analizzarne i principali per comprenderne il funzionamento e le basi. Oltre agli esempi ufficiali comunque, se ne trovano moltissimi anche su GitHub, grazie al supporto della comunità.

4.2.4 Primo CMakeLists

Come menzionato nella sezione precedente, avevo esperienza nel fare la compilazione di alcune librerie utilizzando CMake, ma non lo avevo mai utilizzato in un progetto personale. Nel piano di lavoro, CMake era segnato come obiettivo facoltativo con: "F03: porting librerie aziendali su CMake". È stato invece deciso con il *tutor* di iniziare il progetto direttamente utilizzando CMake: innanzitutto per imparare le basi e come utilizzarlo, ma anche per non dover cambiare sistema di *build* in seguito.

Qt utilizza di default un sistema di *build* basato su Makefile, generati tramite il comando qmake su un progetto ".pro". Questo funziona molto bene con Qt, ma è specifico per tale utilizzo. CMake invece è un sistema di *build* molto più generico e ampiamente utilizzato. L'azienda voleva esplorarne le capacità e le funzionalità, e anche io ero

molto interessato ad imparare come utilizzarlo. CMake funziona tramite la scrittura di uno o più file *CMakeLists.txt*, che definiscono tutti i parametri di configurazione del progetto, le librerie esterne e i file da compilare. Per nostra fortuna Qt ha il supporto a CMake, quindi non è complesso da integrare. Diamo un'occhiata ad un semplice *CMakeLists.txt* di base per Qt.

```

1 cmake_minimum_required(VERSION 3.15.4)
2
3 project(helloworld)
4
5 set(CMAKE_AUTOMOC ON)
6 set(CMAKE_AUTORCC ON)
7
8 find_package(Qt5 COMPONENTS Widgets REQUIRED)
9
10 add_executable(helloworld
11    mainwindow.ui
12    mainwindow.cpp
13    main.cpp
14    resources.qrc
15 )
16
17 target_link_libraries(helloworld Qt5::Widgets)
```

Analizziamolo per comprenderne i punti principali:

- * *cmake_minimum_required*: definisce la versione minima di CMake richiesta dal progetto;
- * *project*: definisce il nome del progetto e di conseguenza tutte le variabili relative (come per esempio *PROJECT_SOURCE_DIR*);
- * *set*: imposta una variabile ad un certo valore, nel nostro caso le variabili *CMAKE_AUTOMOC* e *CMAKE_AUTORCC* sono variabili utilizzate da Qt, impostate ad ON per specificare di usare il MOC e la gestione dei file di risorse;
- * *find_package*: definisce una libreria da trovare ed utilizzare. In caso la posizione di tale libreria non sia nota (per esempio in una variabile di sistema) bisogna passare a CMake il percorso, con una variabile che nel caso di Qt sarà *Qt5_DIR*;
- * *add_executable*: definisce l'eseguibile da creare con la relativa lista di file da compilare. In questo esempio anche un file ".ui" e uno ".qrc", file specifici di Qt. Notare che non ci sono i file *header* ".h" che di norma vengono letti automaticamente, salvo casi particolari;
- * *target_link_libraries*: definisce le librerie di cui fare il *linking* nell'eseguibile.

4.2.5 Ripasso Qt

Comprese le basi di CMake, dovevo ripassare le funzionalità e la struttura di Qt; avendolo già utilizzato per il progetto di Programmazione ad Oggetti, lo conoscevo abbastanza bene. Dovevo tuttavia discutere con l'azienda che tipo di interfaccia fare, come strutturare le classi dei *widget* e come gestire i segnali di click dei bottoni. Ho dovuto quindi progettare e realizzare l'interfaccia di base che sarebbe poi stata utilizzata. Il mio *tutor* mi ha anche suggerito di utilizzare alcune *feature* particolari, come la funzione *tr()* che permette di inserire una stringa che sarà successivamente possibile tradurre in più lingue, ottenendo così un'interfaccia multilingue.

4.2.6 Integrazione Qt-VTK

Come accennato nella sezione [Integrazione con Qt](#) (§3.3.7), VTK è stato compilato con il supporto a Qt: questo offre classi come *QVTKOpenGLNativeWidget*, da cui è possibile derivare un *widget* Qt. Un punto molto importante discusso con il *tutor* aziendale, è stato fare in modo sin da subito che il *widget* Qt derivato da *QVTKOpenGLNativeWidget*, che d'ora in poi chiameremo *RenderWindow*, fosse indipendente, composto quindi dal minor numero possibile di dipendenze, questo per permettere che fosse facilmente trasferibile ad altre applicazioni. Il *RenderWindow* funzionante si può vedere più avanti nella figura 4.4.

Per testare la funzionalità del primo prototipo di *RenderWindow*, ho cercato degli esempi su GitHub che mostrassero come caricare un semplice modello 3D su VTK, in modo da assicurarmi che funzionasse prima di passare al caricamento di un volume vero e proprio.

4.2.7 Volume Mapper

Come accennato nella sezione [Oggetti di rendering](#) (§3.3.3) e in particolare in [Particolari oggetti di rendering](#) (§3.3.4), un oggetto va visualizzato tramite un *vtkMapper*. Nel caso del *volume rendering*, questo può essere fatto utilizzando *vtkGPUVolumeRay-CastMapper*, un *mapper* che effettua il *ray casting* sfruttando l'accelerazione *hardware* della GPU. Questo è il *mapper* principale che è stato utilizzato, tuttavia in alcuni casi potrebbe non essere disponibile una GPU, è stato quindi deciso di far controllare all'applicazione se è disponibile una GPU sopportata. In caso contrario utilizzerà invece un *vtkSmartVolumeMapper*, un *mapper* "intelligente" che supporta varie modalità: di default utilizzerà una GPU se disponibile, altrimenti passerà automaticamente a *vtkFixedPointRayCastMapper*, un *ray caster software* (CPU) che però è notevolmente più lento.

4.2.8 Primo prototipo

Una volta creato e testato il *RenderWindow*, il primo prototipo di visualizzatore volumetrico è stato fatto caricando le immagini con il *loader* di VTK, chiamato *vtkDICOMImageReader*. È sufficiente infatti indicargli la cartella da leggere perché provi automaticamente a caricare il volume contenuto in tale cartella: il risultato si può vedere nell'immagine 4.2. È importante notare che molti esami TC e RM potrebbero registrare tutte le immagini (e quindi spesso più volumi) in un'unica cartella. In questo caso non è possibile utilizzare *vtkDICOMImageReader*, che con le impostazioni di default cerca invece di caricare un singolo volume da tutti i file presenti nella cartella selezionata. Nel caso i volumi fossero tutti raggruppati in un'unica cartella, se necessario, si possono separare con vari programmi. In ogni caso, per mia fortuna, il primo esame DICOM fornитomi dall'azienda per effettuare le varie prove era già un singolo volume contenuto in una singola cartella, di conseguenza molto semplice da gestire.

4.2.9 Strumenti di base - Rotazione

Ottenuto il primo *volume rendering*, una delle prime cose essenziali era la possibilità di ruotare la visuale/fare lo zoom e interazioni di base simili. Per nostra fortuna VTK fornisce l'oggetto *vtkRenderWindowInteractor*, che consente di definire un'interazione con la nostra finestra di *rendering*. Nel nostro caso volevamo una *camera* che ci consentisse di

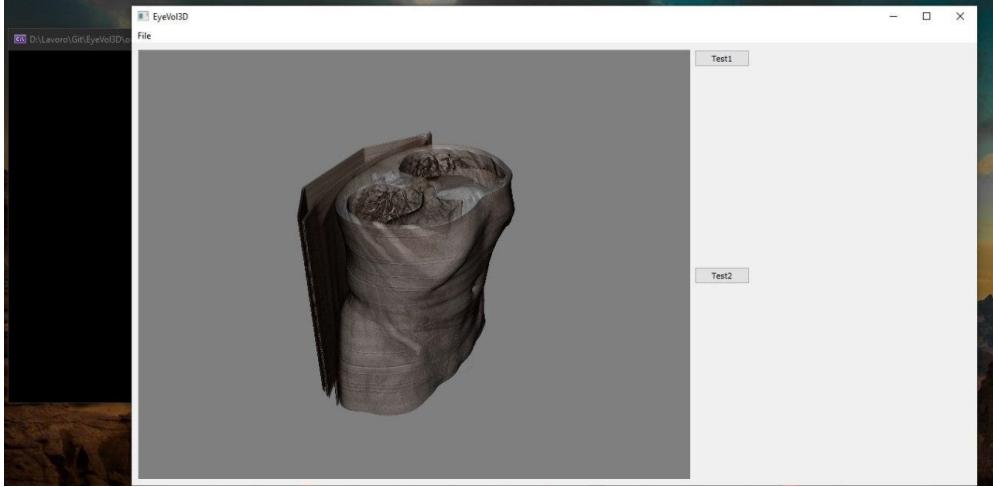


Figura 4.2: Primo volume rendering
Fonte: Stage

ruotare attorno al volume utilizzando il mouse: in VTK è chiamata *vtkInteractorStyleTrackballCamera*. È bastato quindi creare un *vtkRenderWindowInteractor* ed indicargli la finestra di *rendering* da utilizzare; bisognava inoltre assegnargli un’interazione di tipo *vtkInteractorStyleTrackballCamera* per ottenere il risultato desiderato.

4.2.10 Strumenti di base - Preset

A questo punto, si è trattato di aggiungerci i primi strumenti necessari a modificare e meglio visualizzare il volume: il primo e più importante passo è stato aggiungere la lista di *preset*, una lista predefinita e standard di funzioni di trasferimento. Come accennato in precedenza, questa è stata presa dal *repository* di 3D Slicer controllando e rispettandone la licenza. Tuttavia, questa lista era in XML, e un XML non è direttamente utilizzabile, visto che le funzioni di trasferimento vanno inserite in VTK tramite appositi oggetti. Ho deciso quindi insieme al *tutor* di fare una piccola libreria designata solo a fare il *parsing* di tale XML, chiamandola *PresetParser*. Questa libreria carica e fa il *parsing* dell’XML di *preset* e riempie un *array* di oggetti generici chiamati *VolumePropertyEntry* con tutti i dati e le proprietà, sarà poi responsabilità dell’applicazione utilizzare questi valori, nel nostro caso caricandoli su VTK. Nel mio programma il *parsing* viene fatto all’avvio, caricando la lista dei *preset* e visualizzando i nomi disponibili; i valori effettivi della funzione vengono caricati in VTK solo quando questo è selezionato, visto che comunque sono pochi e non è un’operazione complessa. Selezionato un *preset*, quindi, Qt esegue il segnale di "cambio selezione" ed esegue la relativa funzione, che nel nostro caso legge la *entry* del *preset* e crea gli oggetti di VTK *vtkVolumeProperty/vtkPiecewiseFunction/vtkColorTransferFunction*, impostandoli poi per il volume corrente.

4.2.11 Strumenti di base - Mark

Mark è il nome che la mia azienda ha dato ad un piccolo omino stilizzato, già utilizzato in alcune loro applicazioni. Mark è molto utile in un contesto tridimensionale, in cui è necessario comprendere l’orientamento di ciò che si sta osservando: imma-

ginate di guardare la TC di una gamba senza un indicatore di com'è orientata la gamba: ci si potrebbe distrarre e confondere sul sopra/sotto. Come ogni oggetto che viene visualizzato, Mark di base è un *vtkActor*, di cui è fatto il *render* tramite un *mapper* di tipo *vtkDataSetMapper*. Essendo comune avere un *widget* che mostra l'orientamento, VTK offre l'oggetto apposito *vtkOrientationMarkerWidget*, a cui noi assegneremo il nostro *vtkActor* contenente la *mesh* di Mark. Una volta caricato, si può posizionare in un angolo della finestra e abilitare in maniera simile a come fatto per la *vtkInteractorStyleTrackballCamera*: il risultato sarà visibile nella figura 4.4.

4.2.12 Strumenti di base - MIP

Un altro punto molto importante è consentire la possibilità di visualizzare la MIP: questo è stato facile da implementare in quanto VTK offre già dei *flag* per cambiare modalità di visualizzazione, come possiamo vedere nell'esempio sottostante:

```
//Max Intensity Proj
vtkVolumeMapper::SetBlendMode(vtkVolumeMapper::MAXIMUM_INTENSITY_BLEND);
//Min Intensity Proj
vtkVolumeMapper::SetBlendMode(vtkVolumeMapper::MINIMUM_INTENSITY_BLEND);
//Default composite
vtkVolumeMapper::SetBlendMode(vtkVolumeMapper::COMPOSITE_BLEND);
```

Come si può notare è presente anche la *Minimum Intensity Projection*, che raramente è utilizzata, ma è stata inserita comunque sotto consiglio del *tutor*. Com'è intuibile dal nome, funziona come la *Maximum Intensity Projection* ma utilizzando i valori minimi incontrati lungo il raggio. Nel caso fosse necessario, è immediato aggiungere anche la *AVERAGE_INTENSITY_BLEND*.

4.2.13 Strumenti di base - Smoothing

Lo *Smoothing*, chiamato anche *Jittering*, è una funzione che aggiunge rumore (*noise*) alla *texture*, cercando di rimuovere o ridurre l'effetto di "venatura del legno" (wood-grain effect). Si può notare la differenza nell'immagine 4.3, soprattutto nell'ingrandimento. Purtroppo questa funzione è disponibile solo nel *vtkGPUVolumeRayCastMapper*, non è infatti disponibile utilizzando il *mapper software*.

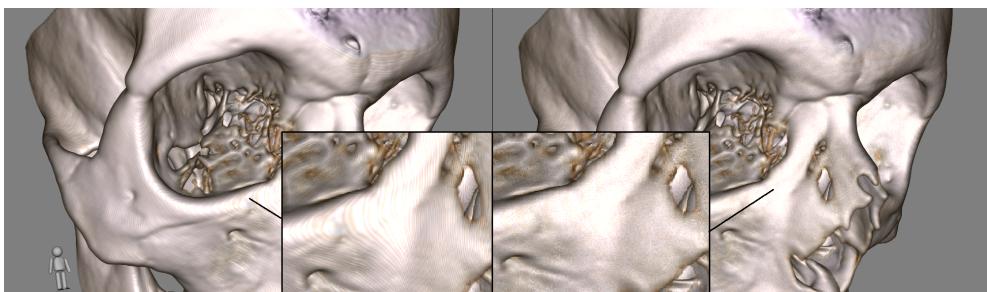


Figura 4.3: Esempio di Smoothing su GPU, OFF a sinistra e ON a destra
Fonte: Stage

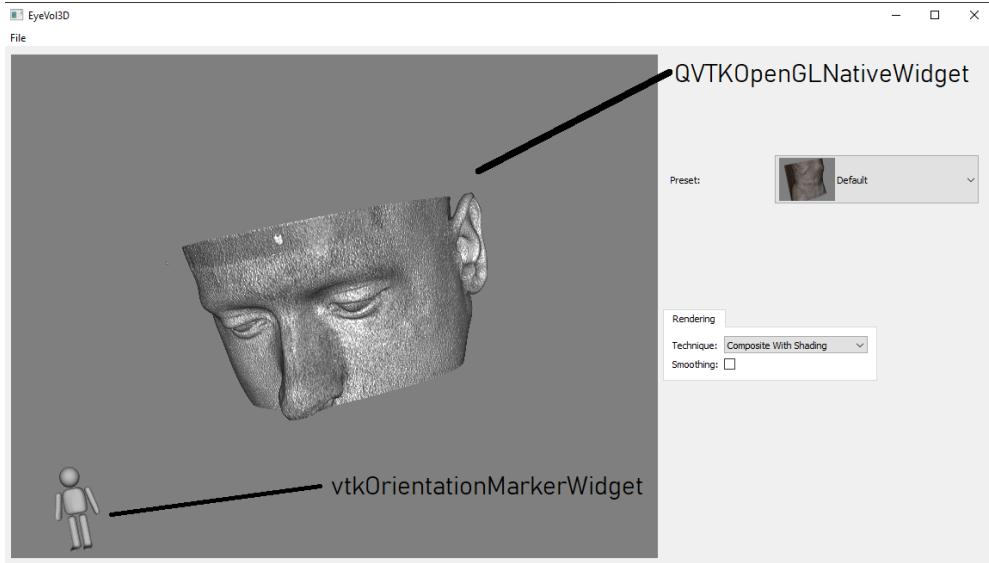


Figura 4.4: Prima interfaccia con strumenti
Fonte: Stage

4.2.14 Strumenti aggiuntivi - Funzione Threshold

La funzione di *Threshold* è una funzione binaria per definire una "soglia", in cui a ogni tipo di tessuto da classificare vengono assegnati due numeri: una soglia bassa e una alta. L'altezza del valore massimo dipende dall'opacità, che come si può vedere nella figura 4.5 si può regolare; il punto alto della funzione quindi avrà lo stesso valore dell'opacità: 0.82 in questo esempio. Affinché un *voxel* sia considerato rappresentativo di quel tessuto, la sua attenuazione deve rientrare nell'intervallo definito dalle soglie bassa e alta.

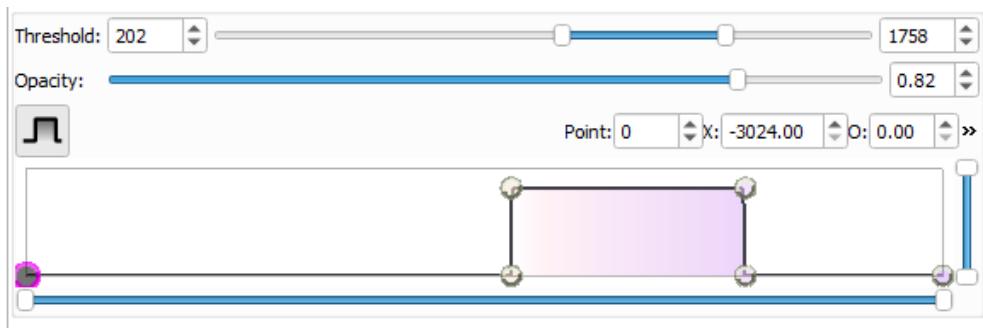


Figura 4.5: Esempio di funzione di Threshold in 3D Slicer
Fonte: Stage

Come detto, la funzione di *Threshold* ha una soglia bassa e una alta, e lo *slider* che si può vedere nell'immagine ha quindi due puntatori: uno per il minimo e uno per il massimo. Implementare questo non è stato immediato, perché Qt non offre nessun doppio-slider. Ho dovuto quindi cercarne uno che soddisfacesse le mie necessità su

GitHub e includerlo nel mio progetto per avere un risultato simile, necessario alla regolazione delle soglie (fonte: [ThisIsClark/Qt-RangeSlider](#)).

4.2.15 Strumenti aggiuntivi - Taglio tramite box

Effettuare il taglio del volume non è stata un'operazione semplice all'inizio: non trovavo esempi coerenti con il mio contesto e molti utilizzavano altri tipi di *mapper* o cercavano, per esempio, di ridurre la griglia di *voxel* da mostrare. Dopo un'attenta ricerca nel codice di 3D Slicer e nella documentazione di VTK, mi sono reso conto dell'esistenza del *vtkBoxWidget*, un *widget* perfetto per effettuare il tipo di taglio che cercavo. Si crea in maniera molto semplice, e basta assegnargli un *vtkProp3D* (classe padre di qualsiasi oggetto come *vtkActor* e *vtkVolume*) per posizionarlo correttamente su quell'oggetto. Una volta posizionato però, non c'era ancora nessun tipo di conseguenza all'interazione. Ho dovuto quindi creare una *callback* da collegare al *vtkBoxWidget* per ricevere ed utilizzare gli eventi di interazione:

```
class vtkCropBoxChangedCallback : public vtkCommand {
public:
    void Execute(vtkObject* caller,
                 unsigned long eventId,
                 void* callData) override;
};
```

Questa *callback*, collegata all'*observer* *vtkCommand::InteractionEvent* del *vtkBoxWidget* riceveva qualsiasi evento di interazione, che ho potuto quindi utilizzare per tagliare il volume. Una volta gestito correttamente il *vtkBoxWidget*, il taglio effettivo non è stato complesso, in quanto ho trovato che il *mapper* contiene già un metodo che riceve i 6 estremi da considerare (min/max per i tre assi x/y/z). Il risultato è visibile nella figura 4.6.

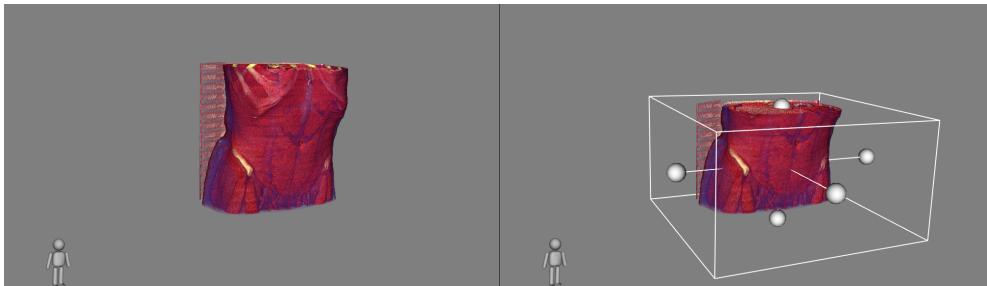


Figura 4.6: Esempio di taglio del volume tramite *vtkBoxWidget*

Fonte: Stage

Notare che, come detto, il metodo di taglio utilizzato con il *vtkBoxWidget* considera solo il min/max di ogni asse. Questo significa che anche abilitando la rotazione del *vtkBoxWidget*, questa non consentirà tagli obliqui in quanto considera solo il punto massimo.

4.2.16 Strumenti aggiuntivi - Taglio tramite plane

Il taglio tramite *plane* è molto simile al taglio tramite *box*: si crea il *widget* *vtkPlaneWidget*, lo si assegna ad un *vtkProp3D* e si crea una *callback*. C'è una differenza

importante però: la funzione di taglio di questo tipo è fatta per supportare N piani statici. Questo significa che nel nostro caso con un piano singolo, per spostarlo e aggiornarne la posizione, il piano andrà rimosso dalla lista dei *Clipping Planes* e nuovamente aggiunto ad ogni modifica. Un vantaggio importante però è che un piano di questo tipo consente tagli obliqui in qualsiasi direzione e orientamento. Il risultato è visibile nella figura 4.7.

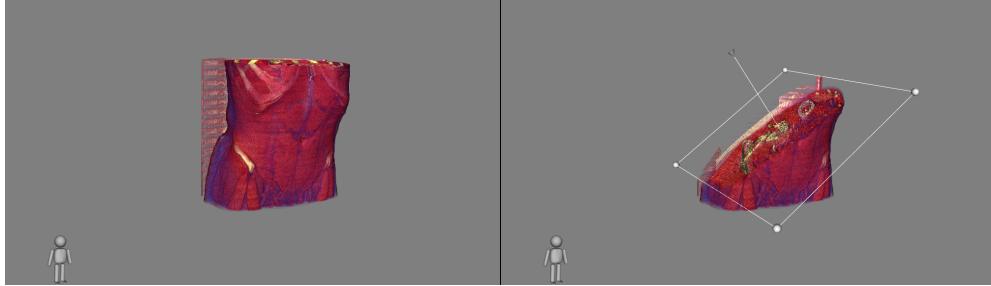


Figura 4.7: Esempio di taglio del volume tramite *vtkPlaneWidget*
Fonte: Stage

Sono anche stati aggiunti metodi e relativi bottoni di utilità nella UI, per reimpostare la posizione/l'orientamento originale del *vtkBoxWidget* o del *vtkPlaneWidget* nel caso fosse necessario da parte dell'utente. Si possono anche nascondere, consentendo di tagliare un volume e successivamente nascondere il *widget* di taglio, così da permettere una visione più pulita del volume.

4.2.17 Compilazione librerie aziendali

Creata tutta la base di visualizzazione e interazione del volume, era importante aggiungere il supporto alle librerie aziendali per l'import di un volume DICOM. Questo richiedeva innanzitutto di portare un considerevole numero di librerie da qmake a CMake, e successivamente di utilizzarle per importare i dati di un volume, che chiaramente andavano convertiti e caricati in VTK.

Questo tipo di dipendenze con CMake è abbastanza facile da gestire: supponiamo di avere una sotto-libreria di una libreria principale, questa si compilerà in maniera simile a quanto visto nel codice nella sezione Primo CMakeLists (§4.2.4), solo che invece di *add_executable* utilizzerà *add_library* come segue:

```

1 # Build library
2 add_library(sublibrary
3   STATIC
4   ${Sources}
5 )

```

Viene compilata così una libreria statica il cui nome è *sublibrary*, da una lista di sorgenti \${*Sources*}. La libreria principale che utilizza quella sotto-libreria ora dovrà utilizzare *ADD_SUBDIRECTORY* per aggiungere una sottocartella al sistema di *build* e compilarne la libreria presente se necessario. A quel punto dopo aver compilato sé stessa dovrà chiamare *target_link_libraries* per dire che la *sublibrary* è strettamente necessaria e ha quindi bisogno di effettuarne il *linking*. Nel complesso quindi, saltando comandi come *project*, si può riassumere così:

```

1 # Add sub library
2 ADD_SUBDIRECTORY(${PROJECT_SOURCE_DIR}/sublibrary/)
3
4 # Build library
5 add_library(library
6   STATIC
7   ${Sources}
8 )
9
10 # Link libraries
11 target_link_libraries(library sublibrary)

```

Abbiamo così creato una libreria che compila ed include le sue dipendenze. A questo punto il CMake principale che crea l'eseguibile dovrà solo includerla con `ADD_SUBDIRECTORY` ed effettuarne il *linking* con `target_link_libraries`.

Notare che `ADD_SUBDIRECTORY` aggiunge solo una subdirectory al sistema di *build*, cercando il file `CMakeLists.txt` presente nella cartella indicata. Per aggiungere una cartella di include si utilizza invece l'istruzione `TARGET_INCLUDE_DIRECTORIES`, come nell'esempio seguente:

```
TARGET_INCLUDE_DIRECTORIES(library PRIVATE ${PROJECT_SOURCE_DIR}/sublibrary/)
```

4.2.18 Problemi compilazione librerie aziendali

All'inizio non ho avuto grossi problemi nel portare le librerie su CMake, molte erano relativamente piccole e avevo già scoperto come collegare più CMake tra di loro con la libreria del *parser* dei *preset*. Inoltre, avendo a disposizione il file di progetto ".pro" per ognuna di esse, era relativamente semplice convertire le istruzioni di compilazione a CMake.

Tuttavia, alcune librerie mi hanno dato grossi problemi, in quanto non riuscivo a compilarle e non riuscivo a comprenderne il motivo. Ne ho discusso con colleghi di lavoro e di università, ma nessuno riusciva a risolvere il problema. Dopo vari giorni di tentativi, discussioni e ricerche ho scoperto che il problema era causato dall'impostazione di CMake, che specifica il linguaggio del progetto di tale libreria a C/C++, essendo i file di libreria ".c". Togliendo completamente quell'istruzione, la compilazione è ripresa correttamente. Il problema è stato difficile da individuare anche perché tale istruzione andava rimossa da più di una libreria, ma l'errore indicato dal sistema rendeva difficile comprendere questo dettaglio.

4.2.19 Import volume con librerie aziendali

Una volta riuscito a compilare correttamente tutte le librerie aziendali necessarie al mio progetto, facendo delle prove mi sono reso conto che importare l'immagine dal tipo di oggetto aziendale a VTK era tutt'altro che semplice. Per comodità, sicurezza e per confronto ho lasciato il metodo di import precedente creato tramite `vtkDICOMImageReader`, e ho fatto l'*overload* della funzione nella classe del `RenderWindow` con il nuovo tipo.

Non è stato semplice comprendere come copiare i pixel delle immagini dentro il tipo di VTK, per cui dopo vari tentativi sono andato nel [forum di VTK \(discourse.vtk.org\)](#) per chiedere suggerimenti e consigli su come fare.

Il primo punto da considerare è se il volume caricato tramite librerie aziendali è composto da immagini a 8, 16 o 32 bit, e da quanti bit di profondità (byte per pixel); queste informazioni vanno considerate per creare correttamente l'oggetto VTK e per calcolare la memoria necessaria. Bisogna anche considerare lo *spacing* tra le immagini. Letti correttamente questi dati, e compreso come copiare i pixel di ogni immagine dentro un oggetto di tipo *vtkImageData*, il risultato è subito parso errato, come si può notare nell'immagine 4.8.



Figura 4.8: Esempio di import volume incorretto (ordine errato)

Fonte: Stage

Questo perché, come ho scoperto con il *tutor*, nell'oggetto aziendale che carica le immagini dalla cartella le immagini non sono ordinate, e sono quindi fornite in ordine errato. Per ovviare a questo problema è stato utilizzato un semplice algoritmo di ordinamento per ordinare le immagini correttamente rispetto all'asse Z, documentando che questo andrà migliorato/cambiato in quanto alcune immagini potrebbero andare ordinate rispetto ad un altro asse. Riordinate le immagini il risultato è stato il seguente, visibile nell'immagine 4.9. Seppur migliore, è chiaramente errato perché non assomiglia in alcun modo al risultato corretto visibile a sinistra nell'immagine 4.8.

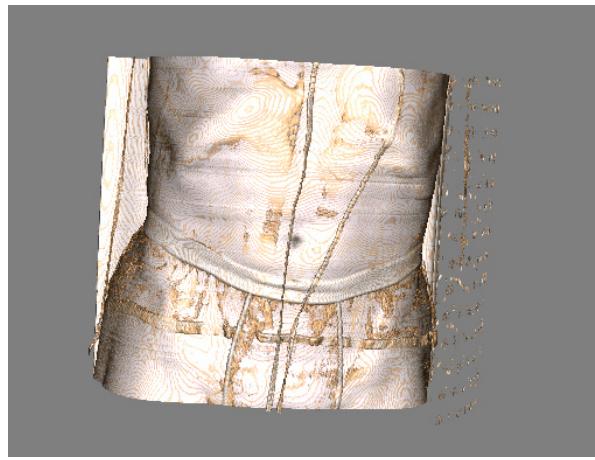


Figura 4.9: Esempio di import volume incorretto (scala errata)

Fonte: Stage

Cosa manca? A quanto pare, sempre discutendo nel forum di VTK, ho scoperto che bisogna considerare gli attributi *RescaleIntercept* e *RescaleSlope* dell'oggetto DICOM, per scalare correttamente i valori dell'immagine, argomento che anche il *tutor* mi aveva menzionato. Ho dovuto quindi leggere quei valori, già presenti nell'oggetto aziendale, e considerare questa equazione:

$$\text{Outputunits} = m * \text{SV} + b$$

in cui SV sono i valori memorizzati (stored values) e in cui m è *RescaleSlope*, quindi anche riscrivibile come:

$$\text{Outputunits} = \text{RescaleSlope} * x + \text{RescaleIntercept}$$

In VTK è stato abbastanza semplice realizzare questo, perché è bastato creare un filtro di tipo *vtkImageShiftScale* fornendogli *RescaleIntercept* e *RescaleSlope*, e applicarlo all'immagine ottenendo così un risultato corretto.

Tuttavia, visto che i casi da considerare sono innumerevoli, i problemi non erano finiti: caricando un'immagine a 12 bit, per esempio, si presentavano errori del tipo che si può notare nell'immagine 4.10, in cui il volume è correttamente disegnato, ma è circondato da valori incorretti, rendendo impossibile visualizzarlo.

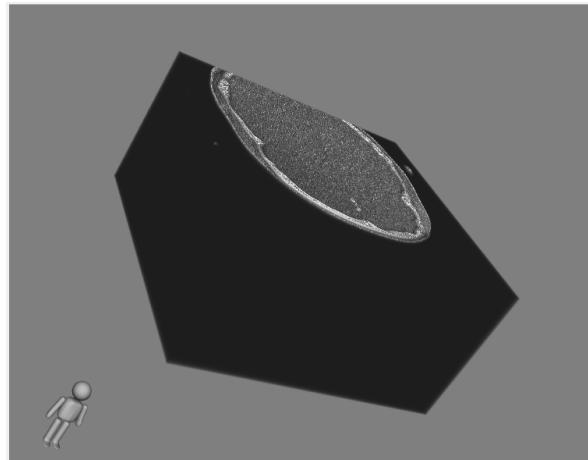


Figura 4.10: Esempio di import volume incorretto (problema con *vtkImageShiftScale*)
Fonte: Stage

Questo è un problema legato a *vtkImageShiftScale*, che di default manda in output valori *unsigned*, mentre a noi servono valori *signed*; è stato quindi sufficiente chiamare *vtkImageShiftScale::SetOutputScalarTypeToFloat()* per indicare di utilizzare valori *signed* per risolvere il problema.

4.2.20 Modifiche interfaccia

Risolti molti problemi nell'import con le librerie aziendali, il *tutor* mi ha fatto notare che era meglio migliorare l'interfaccia per renderla più pulita e portatile. Mi ha suggerito allora di mettere un bottone nel *RenderWindow* per aprire la schermata di impostazioni, in modo da renderla portatile e meno invasiva. Ho quindi creato un bottone e l'ho posizionato nell'angolo in alto a destra del *RenderWindow*: cliccandolo si apre il pannello di impostazioni come visibile nell'immagine 4.11.

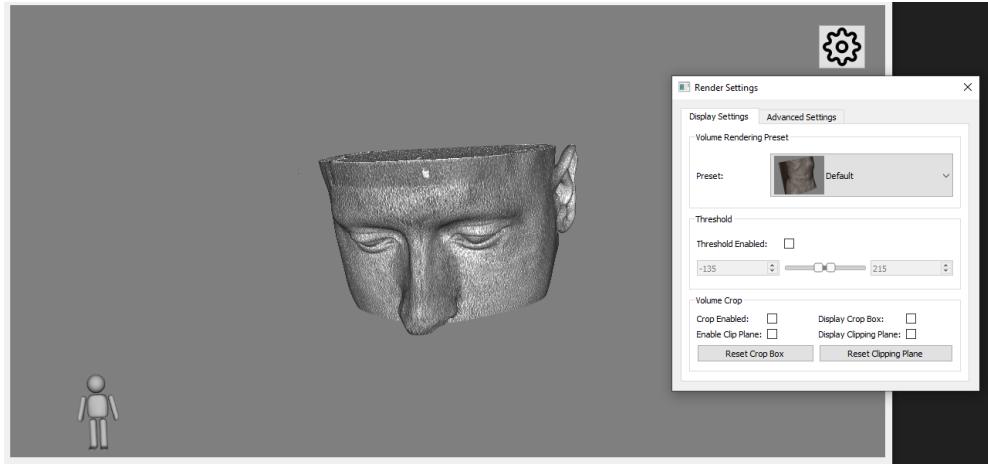


Figura 4.11: Cattura del design finale della UI
Fonte: Stage

Oltre a tutti gli strumenti già discussi in precedenza, nella scheda *Advanced Settings* è presente un tasto per mostrare e, volendo, cambiare il *mapper* da CPU/GPU e viceversa in caso sia necessario. Questo permette anche di visualizzare quale *mapper* è attualmente in uso, per esempio accorgendosi che non è disponibile quello GPU.

4.2.21 Esempio porting

È stato anche realizzato un esempio di *porting* del *RenderWidget* in un'altra applicazione, sia come test per vedere se era possibile, sia per analizzarne il risultato. Nella figura 4.12 si può notare come il *RenderWidget* sia stato inserito dentro un'altra applicazione. Questa è un'applicazione di demo creata dall'azienda per testare delle funzionalità, e mi è stata fornita per provare ad aggiungere il mio *widget* nel quarto quadrante, precedentemente inutilizzato.

Quando viene caricato un volume, quindi, sono visibili le immagini sui 3 assi che si possono scorrere, ed il relativo volume. Un possibile sviluppo futuro sarebbe di mostrare nel volume la sezione che si sta osservando sulle immagini, disegnando una linea/un piano sul volume in corrispondenza dell'immagine che si sta osservando, fornendo una visione più precisa; tuttavia questo non era un requisito e non c'era il tempo di realizzarlo.

4.3 Documentazione

4.3.1 Codice

Il codice è stato propriamente documentato durante la sua scrittura, con commenti nelle funzioni, ma soprattutto nell'*header ".h"*, file interfaccia di C++ che contiene solo la definizione di variabili/funzioni e che di solito è il file di riferimento da analizzare quando si utilizza una classe.

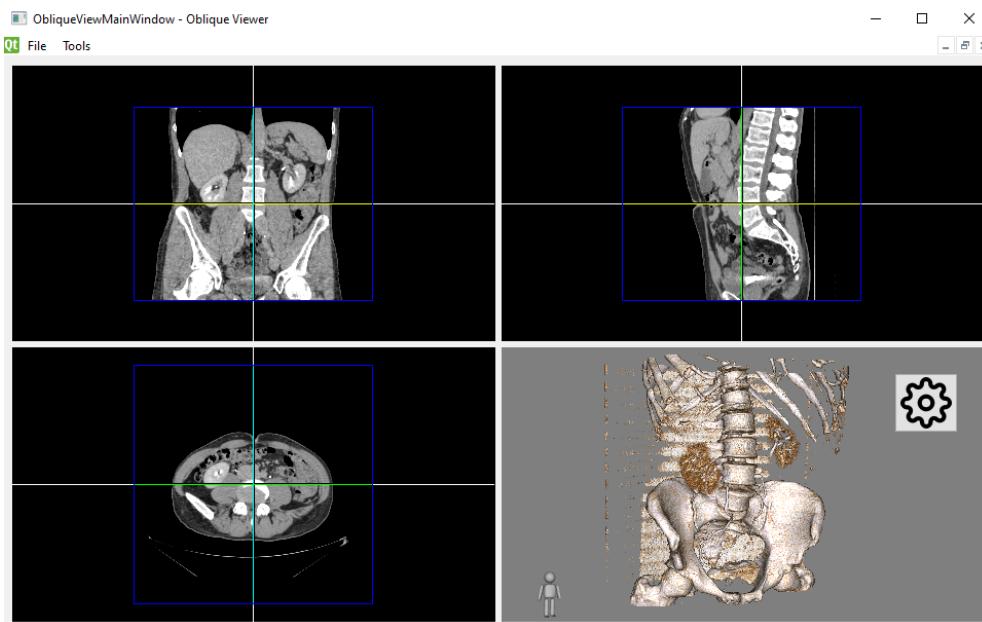


Figura 4.12: Integrazione RenderWidget in altra applicazione
Fonte: Stage

4.3.2 Wiki

Un punto essenziale della documentazione è stato documentare il progetto nella sua interezza nella pagina *Wiki* disponibile nel Redmine aziendale; lì ho scritto in maniera esaustiva:

- * le istruzioni dettagliate su come compilare il progetto, con la lista di dipendenze e di *software* necessari;
- * una guida a com'è gestito il progetto con CMake e a come funzionano i vari file *CMakeLists*;
- * una spiegazione di ogni file di codice prodotto, concentrandomi principalmente su *RenderWidget*, spiegando come e perché sono state realizzate certe funzioni e come utilizzarle;
- * un esempio di sviluppo futuro, con un esempio documentato di come portare il *RenderWidget* in un'altra applicazione.

4.4 Test

4.4.1 Possibili test su una UI

Testare un'applicazione con una interfaccia utente non è banale, e dipende molto da come questa è strutturata. Nel caso del corso di Programmazione ad Oggetti, per esempio, l'applicazione era realizzata tramite *pattern Model-view-controller*: questo rende molto semplice testare le tre parti, essendo esse completamente separate. Nel mio caso non c'era un vero e proprio *model*, se non quello gestito da VTK, e l'applicazione

effettiva era quasi completamente interfaccia grafica. Testare una UI è possibile, si può simulare il click di un bottone, l'inserimento di valori e molto altro, tuttavia quando il risultato è un *render* come nel nostro caso, verificarne la correttezza in modo automatico diventa un'impresa molto ardua.

4.4.2 Test di 3D Slicer e VTK

Un sistema teoricamente fattibile per testare il risultato del render di un volume, discusso con il *tutor*, è confrontare l'immagine ottenuta tramite *render* con un'immagine dello stesso volume che si è certi sia corretta per calcolare la differenza: se c'è troppa differenza qualcosa nel *render* realizzato è incorretto e va corretto. Esplorando i test di VTK, non sembra fare questo nei test del volume, infatti nel codice si trovano commenti del tipo:

```
// For now we are just checking to make sure that the mapper does not crash.
// Maybe in the future we will do an image comparison.
```

Si trovano dei semplici confronti tra immagini, in alcuni test di IO o di filtri semplici, i test del *ray tracing* e dei volumi si concentrano sul testare il corretto caricamento e funzionamento degli oggetti.

3D Slicer allo stesso modo fa i test principalmente della logica o del funzionamento dei *widget* simulando click ed eventi, come anche VTK fa in alcuni punti. Occasionalmente salva lo *screenshot* del risultato del test, o confronta alcuni colori, ma nemmeno in 3D Slicer ho trovato un vero e proprio test per controllare la correttezza di un volume.

4.4.3 Test implementati

Vista l'impossibilità di effettuare un test vero e proprio sul risultato del *render* del volume, nel tempo che mi rimaneva mi sono concentrato nel fare test più utili. Ho quindi studiato ed utilizzato il framework di Qt, chiamato QTest, per effettuare i test di unità:

- * un test di unità per la classe che effettua il *parsing* dell'XML dei *preset* delle funzioni di trasferimento, in modo da assicurare che i dati letti siano coerenti con quelli nel file per non rischiare errori nella lettura;
- * un test di unità per il *widget double slider* utilizzato per la funzione di *Threshold*, in modo anche da imparare come viene testato un *widget*, simulando l'immissione di valori e la relativa lettura, controllando anche come reagisce cercando di inserire valori incorretti.

Una volta realizzati i file cpp dei test, sono stati aggiunti al file CMake del progetto, rendendoli così riconoscibili da QtCreator come test di unità. Questo si effettua aggiungendo *add_test* dopo *add_executable*, in maniera simile a quanto segue:

```
1 # Slider Test
2 add_executable(customslidertest test/customslidertest.cpp)
3 # Add it as test
4 add_test(NAME CustomSliderTest COMMAND customslidertest)
5 # Link libraries
6 target_link_libraries(customslidertest PRIVATE Qt5::Widgets Qt5::Test CustomSlider)
```

Si può notare come nel *target_link_libraries* del test, oltre a Qt c'è *CustomSlider*, l'oggetto dello *Slider* effettivo da testare, di cui ovviamente il test ha bisogno.

Capitolo 5

Conclusioni

5.1 Consuntivo finale

Come previsto nel piano di lavoro, sono state svolte 304 ore in totale. Tuttavia, come accennato nella sezione [Problemi e ritardi](#) (§4.1.4), alcune attività hanno richiesto più tempo di quanto preventivato. Inoltre, la stesura di tutta la documentazione è stata più impegnativa del previsto, vista la mole di aspetti da documentare, mentre il collaudo ha richiesto una quantità di tempo leggermente minore. Il consuntivo finale è visibile nella tabella 5.1.

| Preventivo | Consuntivo | Differenza | Attività |
|------------|------------|------------|--|
| 75 | 75 | 0 | Installazione e studio delle librerie necessarie |
| 26 | 20 | -6 | Studio e progettazione GUI in Qt |
| 9 | 14 | +5 | Sviluppo GUI in Qt |
| 45 | 35 | -10 | Analisi e progettazione della <i>Pipeline</i> 3D |
| 24 | 30 | +6 | Sviluppo <i>Pipeline</i> 3D |
| 85 | 90 | +5 | Sviluppo del prodotto |
| 25 | 20 | -5 | Collaudo prodotto |
| 15 | 20 | +5 | Stesura documentazione finale |

Tabella 5.1: Consuntivo finale

5.2 Raggiungimento degli obiettivi

Nella sezione [Obiettivi](#) (§2.4) è possibile visualizzare gli obiettivi discussi e definiti prima di iniziare l'attività di stage. Ora che lo stage è concluso possiamo verificarne il grado di completezza.

5.2.1 Obiettivi obbligatori

- * [*O01*](#) (visualizzazione interattiva volumetrica): soddisfatto. Il visualizzatore volumetrico realizzato funziona come richiesto e permette di caricare, visualizzare ed interagire con un volume;
- * [*O02*](#) (possibilità di scelta della funzione di trasferimento dei *voxel*): soddisfatto. C'è una lista di funzioni di trasferimento predefinite tra cui scegliere. È inoltre possibile modificare le funzioni di trasferimento predefinite o aggiungerne tramite l'apposito file XML, esterno al programma.

5.2.2 Obiettivi desiderabili

- * [*D01*](#) (piani di taglio del volume): soddisfatto. È possibile tagliare il volume tramite un *box widget* (quindi composto da 6 piani indipendenti) o con un *plane widget* (quindi un piano singolo, anche rotabile a piacimento);
- * [*D02*](#) (modifiche alla funzione taglio): soddisfatto. I *widget* per tagliare il volume permettono l'interazione dell'utente per effettuare il taglio desiderato, posizionando e ridimensionando i *widget* a piacimento. È inoltre possibile nasconderli o reimpostarli alla loro posizione originale;
- * [*D03*](#) (ottimizzazione *rendering GPU*): soddisfatto. Il *mapper GPU* con alcune impostazioni particolari si è dimostrato il metodo più performante per effettuare il *render* del volume.

5.2.3 Obiettivi facoltativi

- * [*F01*](#) (algoritmi di segmentazione con ITK): non soddisfatto. Non si è dedicato del tempo per studiare approfonditamente ITK, in quanto utilizzare gli algoritmi di segmentazione era solo una possibilità non strettamente richiesta dall'azienda;
- * [*F02*](#) (analisi *unit-testing* su GUI-Qt): soddisfatto. Sono stati sviluppati alcuni semplici *unit-test* utilizzando il *framework* di *test* fornito da Qt. Come accennato nella sezione [Test](#) (§4.4) non è stato possibile fare un vero e proprio test sul *render* del volume, ma il *tutor* si è trovato d'accordo con questo risultato;
- * [*F03*](#) (*porting* librerie aziendali su CMake): soddisfatto. Le librerie aziendali scelte dal *tutor* sono state portate su CMake, e molte attivamente utilizzate, come la libreria di caricamento di un volume DICOM.

5.3 Conoscenze e abilità acquisite

Questa esperienza di stage mi ha permesso di apprendere e consolidare diverse nozioni teoriche e pratiche durante i due mesi di stage, che verranno elencate di seguito.

5.3.1 Azienda

Ho avuto modo di imparare come lavora un'azienda che opera in ambito medico: le certificazioni che sono necessarie al *software* per essere distribuito, gli strumenti radiologici utilizzati per effettuare dei test e la gestione delle immagini: dall'archiviazione e la visualizzazione, alla stampa del disco per il paziente. Ho avuto modo di imparare anche come la mia azienda e in particolare il mio *tutor* fanno manutenzione in remoto o di persona presso alcune loro installazioni quando si verifica un problema, e come si rechino direttamente sul luogo per effettuare alcuni test per nuove applicazioni.

5.3.2 Immagini radiologiche e DICOM

Come accennato nella sezione precedente, ho avuto modo di imparare molto di come vengono gestite le immagini radiologiche. Imparare cos'è e come funziona lo standard DICOM: da come riceve le immagini dallo scanner a come vengono archiviate. Leggere nella documentazione tutti gli attributi di cui è composto ed utilizzarlo attivamente è stato molto interessante.

5.3.3 Volume Rendering

Prima di questo stage conoscevo solo delle basi e delle nozioni basilari riguardo il *volume rendering*, studiato per interesse personale. Imparare come viene utilizzato e come si sta espandendo in ambito medico è stato illuminante, ed utilizzarlo personalmente, seppur ad un livello più alto tramite un'astrazione (VTK), è stato estremamente interessante. Studiare ed utilizzare tutto ciò che è necessario per una corretta visualizzazione, dalla correttezza dell'input alle modifiche al volume tramite funzioni di trasferimento, è stata una delle attività più significative. Gli ultimi giorni di stage ho avuto modo anche di analizzare per curiosità personale lo *shader* che VTK utilizza sulla GPU, ed è stato molto entusiasmante poter leggere nel dettaglio come funziona.

5.3.4 Linguaggi e tecnologie

Questo stage mi ha permesso principalmente di migliorare le mie conoscenze di C++, linguaggio già utilizzato sia in Università che a livello personale. Imparare ad utilizzare CMake, già utilizzato in precedenza ma mai scritto personalmente, ha senza dubbio ampliato le mie conoscenze e sono sicuro che mi tornerà molto utile in futuro, in quanto è un sistema di *build* molto diffuso e utilizzato nell'ambito del C++. Migliorare le mie conoscenze di Qt ed utilizzarle per sviluppare un'applicazione completa, seppur concentrata su un *widget* solo, è stato valido, in quanto è un *framework* molto diffuso per sviluppare interfacce grafiche. Infine, imparare ad utilizzare una libreria come VTK, seppur specifica per l'ambito scientifico, è stato molto interessante.

5.3.5 Strumenti

Migliorare le mie conoscenze e le mie abilità nell'utilizzare strumenti come Visual Studio e Qt Creator è stato senza dubbio utile: imparare meglio come utilizzare il *debugger*, come caricare, compilare e gestire un progetto CMake direttamente dall'IDE sono tutte conoscenze estremamente utili. Lo strumento di controllo versione Git era già stato da me utilizzato per dei progetti universitari, ma imparare come viene utilizzato in un contesto aziendale e con che *software* di utilità è stato senza dubbio arricchente.

5.4 Valutazione personale

Personalmente mi ritengo molto soddisfatto dell'esperienza fatta durante lo stage, anche se, lavorando principalmente da casa per colpa dell'attuale emergenza dovuta al Covid, non l'ho vissuto integralmente come hanno fatto altri miei colleghi, che hanno frequentato attivamente l'azienda, con il rapporto e la collaborazione con i colleghi di lavoro ogni giorno. Sono molto fiero dei risultati ottenuti, soprattutto perché quando ho discusso lo stage con il *tutor* la prima volta mi era sembrato un progetto piuttosto complesso da realizzare in due mesi. Con particolare impegno tuttavia sono riuscito a portarlo a termine, e il *tutor* si è dichiarato pienamente soddisfatto del risultato. Gli obiettivi sono stati tutti raggiunti, tranne uno facoltativo; per la prima volta ho sperimentato la soddisfazione di aver concretamente condotto un progetto aziendale dall'inizio alla fine. L'unico obiettivo non completato è quello riguardo ITK, *framework* analizzato a grandi linee e discusso con il *tutor*, ma non realizzato per mancanza di tempo.

Sono anche molto soddisfatto di ciò che ho imparato riguardo l'ambito medico: come funzionano gli esami radiologici tridimensionali, come vengono archiviate e gestite le immagini, come vengono stampati i dischi da dare al paziente, e come sia fondamentale gestire correttamente le immagini che il radiologo andrà ad analizzare. Vedere e provare con mano un monitor radiologico è stata un'esperienza estremamente istruttiva.

Applicare i concetti imparati all'Università (nel mio caso in particolare quelli di Programmazione ad Oggetti) ad un contesto reale in azienda è stato molto interessante; da un lato mi sono reso conto di com'è strutturata un'azienda, di come gestisce un progetto, di quali siano le specifiche figure professionali, i ritmi di lavoro, le collaborazioni fra colleghi, le scelte implementative; dall'altro ho verificato tutti gli aspetti necessari da tenere in considerazione per sviluppare una vera applicazione. Ho quindi preso maggiore consapevolezza delle mie conoscenze e di come ampliarle in caso di difficoltà. In sostanza questa esperienza ha rafforzato il mio interesse verso il settore della programmazione grafica, attualmente molto in espansione.

Bibliografia e sitografia

Riferimenti bibliografici

Will Schroeder Ken Martin, Bill Lorensen. *The Visualization Toolkit - An Object-Oriented Approach To 3D Graphics (4th Edition)*. Kitware, 2006. URL: <https://vtk.org/vtk-textbook/>.

Siti web consultati

CTK Documentation. URL: <https://commontk.org/index.php/>.

DICOM Standard Browser. URL: <https://dicom.innolitics.com/ciods>.

Gsquared.it (biografia e prodotti). URL: <https://www.gsquared.it/it>.

ITK Documentation. URL: <https://itk.org/Doxygen/html/>.

Qt Documentation. URL: <https://doc.qt.io/qt-5/>.

SimVascular. URL: <http://simvascular.github.io/>.

The Medical Imaging Interaction Toolkit (MITK). URL: <https://www.mitk.org/>.

VTK Documentation. URL: <https://vtk.org/doc/nightly/html/>.

VTKExamples (lorensen.github.io). URL: <https://lorensen.github.io/VTKExamples/>.

Wikipedia - Maximum intensity projection. URL: https://en.wikipedia.org/wiki/Maximum_intensity_projection.

Wikipedia - Ray tracing. URL: [https://en.wikipedia.org/wiki/Ray_tracing_\(graphics\)](https://en.wikipedia.org/wiki/Ray_tracing_(graphics)).

Wikipedia - Volume rendering. URL: https://en.wikipedia.org/wiki/Volume_rendering.