Ray Tracer

Progetto di Programmazione ad Oggetti

Luca Biasotto

a.a. 2020/2021

Progetto realizzato da:

Luca Biasotto (mat. 1162290) e Nicla Faccioli (mat.1165370)

Indice

1 Introduzione

1.1 Descrizione

L'applicazione ha lo scopo di effettuare il rendering di forme tridimensionali utilizzando il ray tracing, ovvero una tecnica molto usata (ad esempio nei videogiochi) per la rappresentazione di oggetti e ambientazioni. Tale tecnica, come è possibile intuire dal nome, si basa sul calcolo del percorso dei raggi di luce all'interno della scena da rappresentare per coglierne le interazioni con gli oggetti, e quindi ottenere effetti di luci ed ombre molto verosimili.

1.2 Suddivisione del lavoro progettuale

Per la realizzazione dell'applicazione è stato utilizzato Git come strumento per il versionamento che ha reso possibile una più semplice collaborazione.
Dopo la fase di analisi del problema e una progettazione iniziale fatta in collaborazione, la suddivisione approssimativa del lavoro è stata la seguente:

- Luca Biasotto: progettazione e codifica del modello
- Nicla Faccioli: progettazione e codifica della GUI

Indipendentemente dalla suddivisione del lavoro, si è sempre potuto contare sull'aiuto del collaboratore per la risoluzione di problemi o difficoltà riscontrate durante lo sviluppo. Infine abbiamo deciso di incontrarci per svolgere insieme le fasi di debugging e testing.

1.3 Ore di lavoro richieste

- analisi preliminare del problema: 4 ore
- progettazione modello: 6 ore
- progettazione GUI: 5 ore
- apprendimento libreria Qt: 8 ore
- codifica modello: 32 ore
- codifica GUI: 29 ore
- debugging: 12 ore
- testing: 5 ore

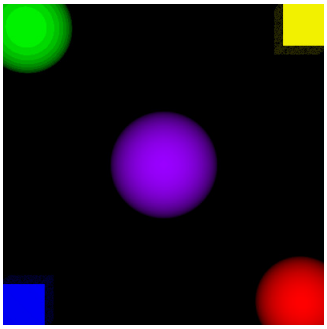
1.4 Strumenti utilizzati

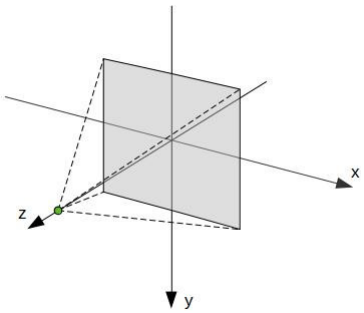
L'applicazione è stata sviluppata su sistema operativo Debian 10.2 con verisone Qt 5.15.2 e g++ versione 10.2.1
Il testing è stato effettuato utilizzando la macchina virtuale fornita dal docente sulla pagina Moodle del corso con le seguenti specifiche:

- Sistema Operativo: Ubuntu 18.04.3 LTS
- Versione Qt: Qt 5.9.5
- Compilatore: gcc 7.3.0

2 Funzionamento

Il programma renderizza un'immagine composta da sfere e cubi in uno spazio tridimensionale, tramite il metodo di ray tracing. Viene definito un osservatore (che ha coordinate x = 0, y = 0, z = 200) e una "finestra" attraverso la quale si vede lo spazio (come da raffigurazione sottostante). Si assume la presenza di una fonte di luce, le cui coordinate coincidono con quelle dell'osservatore. Il programma simula il percorso dei raggi di luce che, partendo dal punto in cui sono presenti osservatore e fonte di luce, attraversano ogni punto della finestra. Quando questi raggi impattano contro un oggetto prensente nello spazio, questo viene illuminato e dunque renderizzato.
La posizione dell'osservatore rispetto all'immagine è rappresentata nello schema seguente:





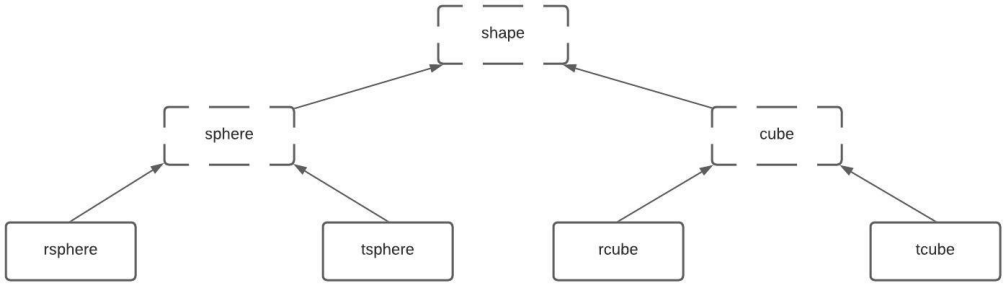
x: positiva a destra
y: positiva in basso
z: positiva uscente

Sono stati implementati due diversi algoritmi di illuminazione. Il primo è quello definito **realistico**, e consiste nell'illuminare ogni punto di un oggetto in proporzione all'angolo formato dal vettore che rappresenta la direzione del raggio di luce e il vettore normale alla superficie dell'oggetto nel punto in cui questo viene colpito dal raggio. Il secondo, chiamato **toon**, segue lo stesso prncipio ma divide in intervalli uguali i valori che l'angolo sopra definito può assumere, assegna a tutti i punti in un intervallo la stessa luminosità.

3 Struttura del codice

3.1 Gerarchia principale

Di seguito è illustrata la gerarchia principale del progetto:



Tale gerarchia è servita per rappresentare le forme di cui è stato implementato il rendering e può essere ampliata con l'aggiunta di ulteriori forme tridimensionali.

3.1.1 Shape

Shape è la classe base astratta della gerarchia. Essa ha come unico campo dati **std::vector<double> _color**, ovvero un vettore di tre valori per rappresentare il colore di una forma in formato RGB. La classe è fornita di metodi *get()* per ciascuno dei tre valori contenuti nel vettore del colore. Possiede inoltre i seguenti metodi virtuali puri, che saranno implementati dalle classi concrete che li ereditano:

- **virtual std::vector<double> normal(const point&) const** : questo metodo restituisce un vettore normale alla superficie dell'oggeto che lo invoca, nel punto che viene passato come parametro. Dato che il procedimento con cui tale vettore viene calcolato dipende dalla forma dell'oggetto, questo metodo deve essere implementato in modo differente da ciascuna delle classi che lo erediteranno.
- **virtual std::tuple<bool, point> intersect(const point&, const std::vector<double>&) const** : questo metodo ha l'obiettivo di calcolare se il raggio di luce colpisce o meno un oggetto. In caso positivo restituirà il punto in cui cioò avviene. Anche in questo caso che l'intersezione avvenga o meno dipende dalla forma dell'oggetto stesso, e dunque l'implementaione viene lasciata alle classi derivate.
- **virtual double lightning(const point&, const std::vector<double>&) const** : questo metodo restituisce un double che rappresenta l'intenistà con cui una superficie viene illuminata. Il calcolo è effettuato considerando l'angolo tra la normale alla superficie stessa e il raggio di luce. Dalle diverse implementazioni di questo metodo dipendono i due diversi tipi di illuminazione messi a disposizione.
- **shape* clone() const** : questo metodo è utilizzato dal puntatore smart per clonare oggetti di tipo shape.

3.1.2 Sphere e sue derivate

La classe astratta **sphere** eredita pubblicamente dalla classe **shape**. Essa ha come campi dati propri il centro e il raggio della sfera e implementa i metodi **normal** e **intersect** ereditati da shape.
Da essa ereditano a loro volta le classi concrete **rsphere** (realistic sphere) e **tsphere** (toon sphere).
La differenza essenziale tra queste due classi sta nel diverso tipo di illuminazione dell'oggetto che rappresentano, caratteristica che si rispecchia nelle diverse implementazioni del metodo ereditato **lightning**. Infatti, nel caso di **rsphere** la luminosità è calcolata per ogni singolo punto in modo da ottenere un risultato più naturale e continuo, mentre per **tsphere** il calcolo è effettuato per gruppi contigui di punti.
L'unico campo dati aggiuntivo rispetto alla classe genitore è **_intervals**, privato in **tsphere** che indica il

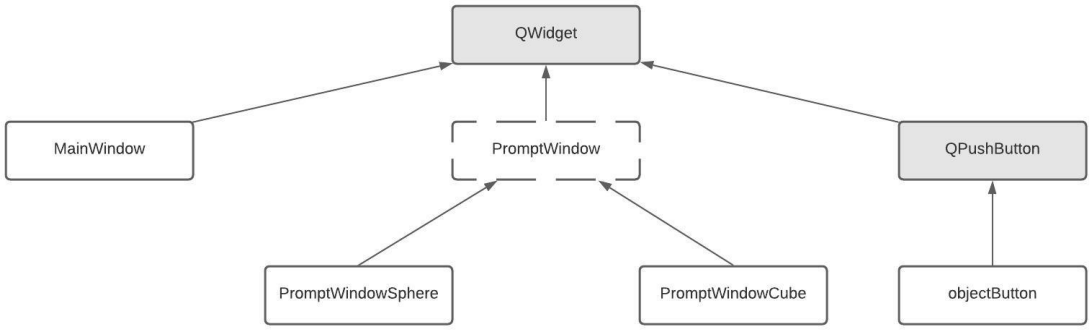
numero di zone in cui dividere la forma per il calcolo della sua luminosità.
Viene inoltre implementato il metodo clone che crea una copia dell'oggetto.

3.1.3 Cube e sue derivate

In modo analogo a **sphere**, la classe **cube** eredita pubblicamente dalla classe **shape**. Essa ha come campi dati propri il centro e il lato del cubo e implementa i metodi **normal** e **intersect** ereditati da shape.
Da essa ereditano a loro volta le classi concrete **rcube** (realistic cube) e **tcube** (toon cube). La differenza essenziale tra queste due classi sta ancora una volta nelle diverse implementazioni del metodo **lightning** descritto sopra e nella presenza dell'unico campo dati aggiuntivo **_intervals**, privato in **tcube**.
Viene inoltre implementato il metodo clone che crea una copia dell'oggetto.

3.2 GUI

Tutte le classi implementate in Qt ereditano pubblicamente da QWidget (direttamente o indirettamente).



Nel grafico, le classi **QWidget** e **QPushButton** sono di colore differente in quanto originali di Qt. **MainWindow** e **PromptWindow** e derivate implementano le varie finestre dell'applicazione. **ObjectButton** invece eredita da QPushButton e implementa un particolare tipo di pulsante utilizzato per i tasti per la modifica e l'eliminazione delle forme.

3.3 Altre classi e funzioni implementate

Oltre alla gerarchia principale è stato necessario implementare alcune classi e funzioni utili al calcolo di dati e alla rappresentazione delle forme.

3.3.1 class point

Un oggetto della classe point rappresenta un punto nello spazio. È quindi caratterizzato da 3 coordinate spaziali x, y e z. La classe mette inoltre a disposizione un metodo per il calcolo della distanza tra due punti.

3.3.2 std::tuple<bool, double> minroot(double a, double b, double c)

Nel file math_utilities è presente questa funzione il cui obiettivo è quello, data un'equazione di secondo grado, di trovarne le soluzioni e restituire quella minore tra di esse.

3.3.3 std::tuple<bool, double> minroot(double a, double b, double c)

Nel file math_utilities è presente questa funzione che risolve l'equazione |a+bx|=c con b>0, c>0 e salva in un vettore le due possibili soluzioni. Tali soluzioni possono essere coincidenti ma devono esistere in quanto b strettamente maggiore di 0.

3.3.4 list e smartP

Si è deciso di implementare una lista templetizzata per contenere puntatori alle forme inserite nell'immagine. Per farlo, è stata inizializzata una variabile globale **list<shape*> global_world** a cui viene aggiunto un nodo in coda per ogni nuovo oggetto inserito. Inoltre è stato implementato un template di classe di puntatori smart da utilizzare all'interno della lista che permettono la gestione automatica della memoria nel momento della cancellazione o dell'aggiunta di nuovi oggetti shape.

3.4 Altre considerazioni

La variabile che rappresenta la lista di oggetti da renderizzare è centrale nel funzionamento del codice e non si prevede che eventuali aggiornamenti ne debbano modificare l'esistenza o l'utilizzo, ma solo eventalmente aggiungere nuovi tipi di oggetti geometrici, il che puoò essere fatto grazie al polimorfismo. Si è dunque deciso di rendere tale vairiabile globale per pura semplicità, renderla una variabile locale e passarla come paramentro alle funzioni che ne fanno uso non comporta alcun cambiamento logico alla struttura del codice. Alla stesso modo si è scelto di dividere il codice in due cartelle e creare due eseguibili separati per il codice del ray tracer vero e proprio e quello dell'interfaccia grafica, dato che fanno uso di gerarchie diverse. Volendo creare un unico eseguibile sarebbe sufficiente sostiruire la *execl()* nella GUI descritta in seguito con una chiamata alla funzione main del codice di ray tracing.

4 Polimorfismo

Di seguito si riportano alcuni esempi di come è stato sfruttato il polimorfismo all'interno del codice del progetto:

- **In tracer.cpp, line 48**: con s->lightning(p, ray). Poiché il metodo lightning è marcato *virtual* nella classe shape, il binding non viene fatto a tempo di compilazione, bensì a tempo di esecuzione. In questo modo viene selezionata la versione corretta del metodo in base al tipo dinamico di s.
- **In tracer.cpp, line 68**: allo stesso modo, (*cit)->intersect(eye, ray) chiamerà la corretta versione del metodo intersect a seconda del tipo dell'oggetto puntato dal const iterator nel momento dell'invocazione.
- **list<shape*> global_world**: la lista è implementata come template e istanziata alla classe astratta *shape**, ma in realtà essa conterrà puntatori a oggetti con tipo dinamico derivato da *shape*, sfruttando quindi il polimorfismo.
- **shape* clone() const**: il metodo è dichiarato virtuale puro all'interno delle classi shape, sphere e cube. Nel momento della sua invocazione viene creata una copia dell'oggetto utilizzando la versione corrispondente al tipo dinamico dell'oggetto stesso.

Il polimorfismo è inoltre utilizzato anche all'interno del codice dell'interfaccia grafica: ad esempio nella classe **PromptWindow** sono presenti metodi e slot virtuali puri che vengono implementati in un secondo

momento dalle classi da essa derivate. Inoltre tutte le classi implementate derivano (direttamente o indirettamente) da QWidget.

5 Funzionalità di Input/Output

L'applicazione utilizza un sistema di Input/Output.
Essa infatti fa uso di un file per la memorizzazione delle forme precedentemente inserite e per l'eventuale salvataggio di nuovi dati. All'apertura dell'applicazione tale file viene letto e viene creato un riquadro nell'apposita zona della Home per ciascuna riga presente all'interno del file. Ogni volta che una nuova forma viene inserita o rimossa, il file viene aggiornato (e con esso anche la lista visibile nell'applicazione).
La risoluzione di base dell'immagine è 100×100 pixel. La prima riga del file contiene un numero intero che rappresenta il fattore per cui la risoluzione iniziale viene moltiplicata, sia in larghezza che in altezza. Le righe successive contengono ciascuna le informazioni relative ad un oggetto nel seguente formato:

- due lettere che indicano se si tratta di una sfera o un cubo ed il tipo di illuminazione
- tre numeri che indicano le coordinate del centro
- un numero che indica la lunghezza del raggio (sfera) o del lato (cubo)
- tre numeri compresi tra 0 e 1 che indicano le componenti RGB del colore
- nel caso di illuminazione di tipo toon, un numero che indica il numero di gradazioni utilizzate

Per rappresentare l'immagine, l'output del programma utilizza il formato *ppm*. La prima riga è composta da un header che contiene le seguenti informazioni:

- *P3*, una stringa che caratterizza il formato
- un numero intero che rappresenta la larghezza in pixel dell'immagine
- un numero intero rappresenta l'altezza in pixel dell'immagine
- un numero intero che rappresenta il valore massimo utilizzato come coordinata RGB

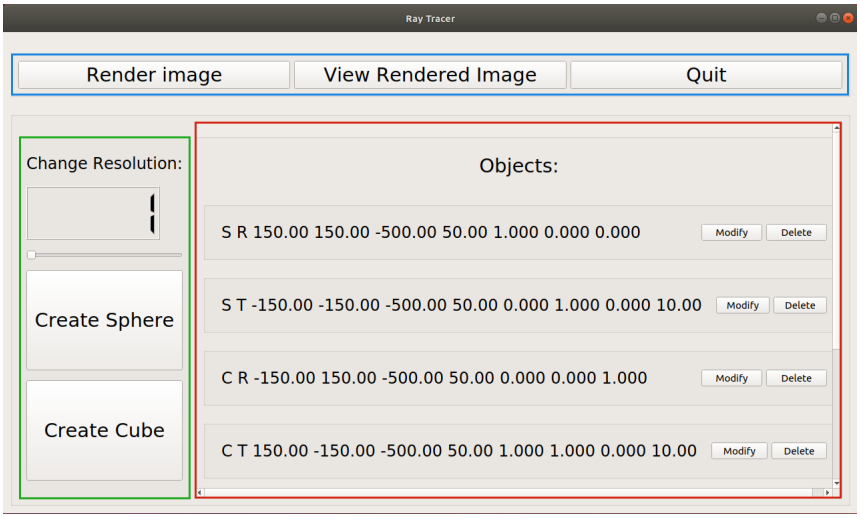
Ogni riga del file contiene tre numeri che rappresentano le coordinate RGB dell'oggetto, compreso in un range tra 0 e il valore indicato dall'ultima cifra dell'header

6 Interfacca grafica

6.1 Home

Quando viene premuto il pulsante per effettuare il rendering dell'immagine, viene utilizzato il comando *execl* che esegue il ray tracer creando un processo figlio.
Per la visualizzazione dell'immagine invece, premendo sul rispettivo pulsante, viene utilizzato il comando *std::system* che esegue il programma *eog*.
I comandi utilizzati per i due casi sono differenti perché, mentre sui computer utilizzati per lo sviluppo del codice essi sono interscambiabili, sulla macchina virtuale non sembrano lavorare allo stesso modo e fornendo quindi risultati differenti.

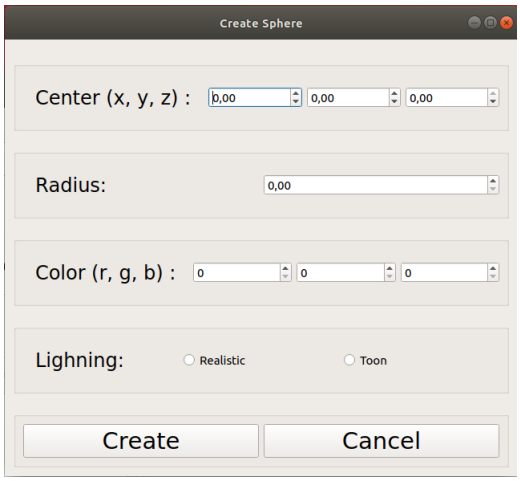
L'accesso ai file di input e di output viene gestito utilizzando il loro percorso relativo anziché il percorso assoluto. Tale scelta è stata presa in modo da rendere il programma indipendente dalla macchina su cui viene eseguito. Tuttavia si richiede che il programma venga eseguito dalla directory nella quale è contenuto l'eseguibile dell'interfaccia grafica.

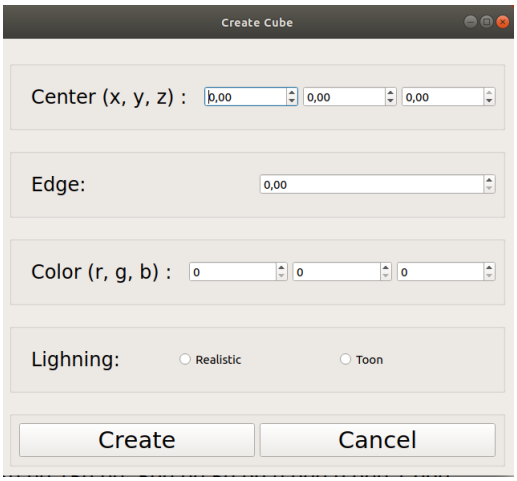


La finestra principale dell'applicazione è composta da tre parti principali:

- **Menù orizzontale**: contiene tre pulsanti che permettono rispettivamente di avviare il rendering dell'immagine composta, visualizzare l'immagine ottenuta e chiudere l'applicazione.
- **Menù verticale laterale**: lo slider permette di cambiare la risoluzione dell'immagine, mentre i due pulsanti permettono di accedere ai form per l'inserimento di nuove forme nell'immagine.
- **Lista di oggetti** : permette di visualizzare gli oggetti precedentemente inseriti. Ogni oggetto nella lista può essere rimosso o modificato attraverso l'utilizzo degli appositi pulsanti.

6.2 Form per inserimento di nuove forme





Premendo su uno dei due pulsanti presenti nel menù verticale a sinistra è possibile inserire nuove forme. In base alla forma scelta si aprirà un form differente per permettere di inserire i parametri corretti (centro e raggio nel caso della sfera, centro e lato nel caso del cubo). In tale form è inoltre possibile scegliere il tipo di luminosità (realistica o toon). Una volta confermato l'inserimento, i dati vengono aggiunti in coda alla lista delle figure già presenti, all'interno del file utilizzato per l'input e l'output. Il documento aggiornato viene quindi riletto dall'applicazione che crea la lista aggiornata di figure inserite, visualizzata nell'apposita zona nella Home.
Nel caso l'oggetto debba essere modificato, si apre una finestra analoga a questa.

7 Compilazione

Il codice del programma è ripartito in due cartelle, *ray_tracer* e *GUI*. La prima contiene il codice riguardante il programma di ray tracing vero e proprio, la seconda l'interfaccia grafica programmata usando Qt. Ogni cartella contiene al proprio interno un Makefile. Quello all'interno di *ray_tracer* è stato scritto direttamente dagli sviluppatori, mentre la cartella GUI contiene inoltre il file *GUI.pro*, che consente di utilizzare il comando qmake per generare automaticamente un Makefile. GUI.pro indica i file header e sorgente utilizzati e indica che verranno usati gli elementi *gui* e *widgets* dalla libreria Qt. Un Makefile è presente anche nella cartello Progetto_p2 che le contiene entrambe, ed esegue in ordine i Makefile nelle sottocartelle. Partendo da zero, il codice può quindi essere compilato digitando i comandi:

- /Progetto_p2/GUI/qmake
- /Progetto_p2/make