Tesina Finale di

Programmazione di Interfacce Grafiche e Dispositivi Mobili

Corso di Laurea in Ingegneria Informatica ed Elettronica – A.A. 2020-2021

Dipartimento di Ingegneria

docente

Prof. Luca Grilli

JRenderEngine

applicazione desktop JFC/Swing

studenti

321310 **Alex Ardelean** alexnicolae.ardelean@studenti.unipg.it

Data ultimo aggiornamento: 17 luglio 2021

0. Indice

1. Descrizione del Problema

L'obiettivo di questo lavoro è lo sviluppo di un'applicazione desktop, denominata *JRenderEngine*, in grado di produrre immagini di semplici scene definite tramite primitive geometriche e renderizzate con un algoritmo di raytracing.

L'applicazione sarà implementata utilizzando la tecnologia JFC/Swing in modo da favorire un'ampia portabilità su diversi sistemi operativi (piattaforme), riducendo al minimo eventuali modifiche al codice sorgente. Tuttavia, il codice prodotto sarà testato e ottimizzato per la piattaforma Linux Ubuntu 18.04.

1.1 Rendering

Il rendering identifica il processo di resa, ovvero di generazione di un'immagine a partire da una descrizione matematica di una scena tridimensionale, interpretata da algoritmi che definiscono il colore di ogni punto dell'immagine digitale.

Un metodo di produzione/visualizzaione dei modelli geometrici può essere diviso in due parti:

- una descrizione di ciò che si intende visualizzare (scena), composta di rappresentazioni matematiche di oggetti tridimensionali, detti "modelli".
- un meccanismo di produzione di un'immagine 2D dalla scena, detto "motore di rendering" che si fa carico di tutti i calcoli necessari per la sua creazione, attraverso l'uso di algoritmi che simulano il comportamento della luce e le proprietà ottiche e fisiche degli oggetti e dei materiali.

Esistono diversi algoritimi che simulano il comportamento della luce, il migliore in termini di realismo dell'immagine prodotta risulta però essere il RayTracing. Tale algoritmo simula il percorso che i raggi di luce compiono se immessi in una scena, ovvero una lista di "modelli" che servono a simulare oggetti reali, è quindi necessario definire la loro geometria affinchè si possa conoscere come interagiscono

con i raggi di luce. In natura la luce che entra in contatto con un oggetto può assumere diversi comportamenti:

- Il percorso del raggio termina nel punto di intersezione (*assorbimento*).
- Il raggio continua il suo percorso rettilineo con una direzione diversa ma non attraversa la superficie (*riflessione*).
- Il raggio attraversa la superficie con una direzione diversa da quella incidente (*rifrazione*).

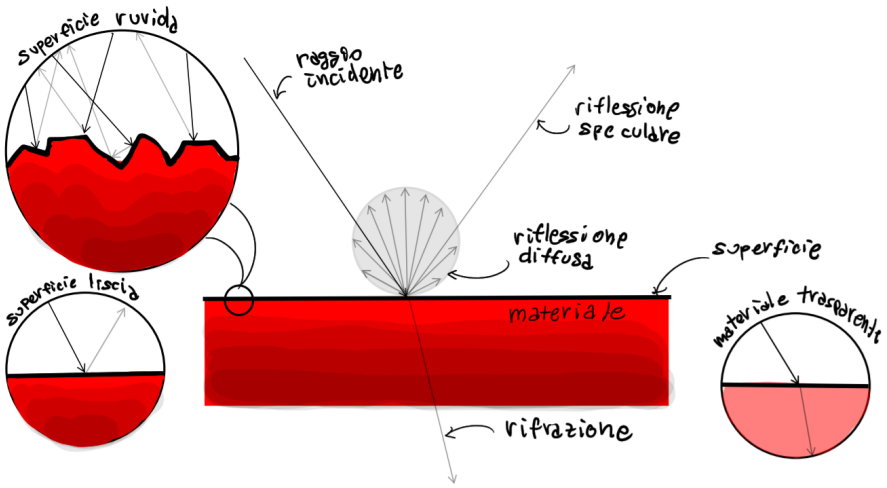

Figura 1.1: Comportamento luce

La rifrazione è descritta dalla legge di Snell: $\frac{sin(\theta_1)}{\sin(\theta_2)} = \frac{n_2}{n_1}$, dove θ_1 e θ_2 sono gli angoli di incidenza e trasmissione mentre n_2 e n_1 sono gli indici di rifrazione dei due mezzi. La riflessione può avvenire in due modi: riflessione speculare, dove la luce viene riflessa con lo stesso angolo di incidenza e riflessione diffusa, dove la luce viene riflessa, con buona approssimazione, uniformemente nell'ambiente.

Da notare che riflessione, assorbimento e rifrazione non si escludono a vicenda, è quindi possibile avere una qualsiasi loro combinazine. Per poter descrivere il comportamento che il raggio avrà una volta colpito un oggetto si deve associare all'oggetto un materiale, ovvero un insieme di parametri che definiscono in modo univoco l'andamento che il raggio assumerà.

I parametri fondamentali per descrivere un materiale sono:

- Un colore che descrive quali componenti della luce sono assorbiti e in che quantità.

- La trasparenza che indica la probabilità che la componente della luce non assorbita venga rifratta, per il principio di conservazione dell'energia la quantità di luce assorbita, rifratta e riflessa deve essere pari alla quantità di luce incidente, e pertanto la trasparenza e l'assorbimento definiscono in modo implicito anche la riflettività della superficie.
- La ruvidezza che ci indica se la luce viene diffusa o riflessa specularmente, materiali ruvidi tendono a riflettere in modo diffuso.
- L'emissione, ovvero la quantità di luce emessa, è diversa da 0 solo per fonti luminose.

Una volta noto come la luce interagisce con tutti gli oggetti nella scena basta seguire il percorso dei raggi di luce e considerare quelli che colpiscono il visualizzatore(occhio, camera, camera virtuale...).

Renderizzare una scena con diversi oggetti e fonti di luce seguendo il percorso che la luce fa a partire dalla fonte fino al visualizzatore risulta però impossibile da realizzare dal punto di vista computazionale. Tuttavia esiste una scorciatoia: non è necessario calcolare il percorso di tutti i raggi emessi dalle fonti per creare un'immagine della scena, ma basta seguire i raggi che effettivamente arrivano al visualizzatore; per fare ciò invece di proiettare i raggi a partire dalla fonte, si proiettano dal visualizzatore e si segue il loro percorso che terminarà in un fonte di luce, se quel raggio è stato effettivamente emesso, altrimenti il raggio andrà all'infinito e il colore del pixel che ha emesso quel raggio risulterà nero o un colore di luce ambientale.

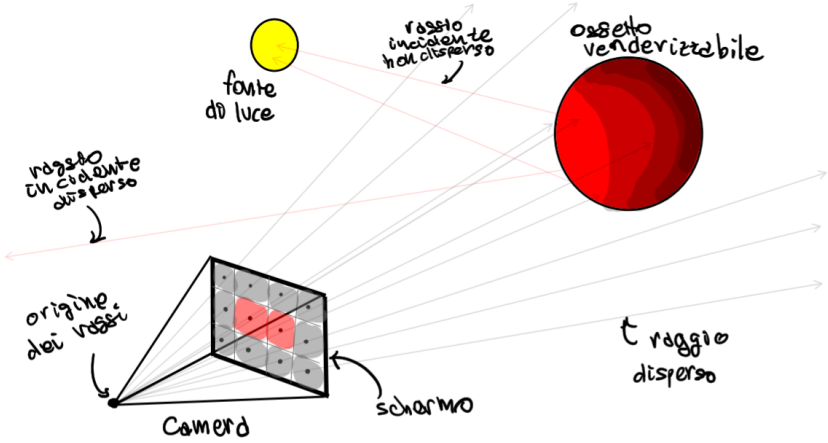

Figura 1.2: Camera virtuale di rendering

Da notare che se si vuole ottenere un effetto realistico, visto che alcuni parametri dei materiali sono definiti in modo probabilistico, bisogna tracciare più raggi per ogni pixel e mediare i risultati, questa tecnica prende il nome di *Path Tracing*.

1.2 L'applicazione JRenderEngine

L'applicazione JRenderEngine si pone l'obiettivo di fornire un'interfaccia che permetta all'utente di creare e visualizzare una scene, in particolare il programma permetterà di:

- Creare la scene, mettendo a disposizione dell'utente varie primitive geometriche per istanziare gli oggetti che conterrà.
- La possibiltà di vedere un anteprima della scena, con parametri che semplificano il costo computazionale, in modo da poter posizionare in modo semplice la camera di rendering.
- Produrre immagini della scena.

2. Specifica dei Requisiti

2.1 Requisiti obbligatori

L'applicazione JRenderEngine che si intende realizzare dovrà soddisfare i seguenti requisiti.

1. L'interfaccia deve prevedere: un menu, un pannello contenente l'elenco degli oggetti presenti nella scena, una pannello preview, un tabbed pane contenente tre pannelli: Oggetto, Scena e Render e infine una barra non interattiva posizionata in basso.

2. La barra non interrattiva mostrerà il numero degli oggetti presenti nella scena e una stima del costo computazionale del rendering in una scala in decimi.

3. Il pannello preview mostrerà un'anteprima della scena e si aggiornerà automaticamente quando l'utente modifica gli oggetti in essa presente oppure posiziona la camera.

4. I parametri di rendering dell'anteprima dovranno consentire all'utente di visualizzare le modifiche in tempo reale e in modo fluido.

5. Possibilità di creare una scena tramite interfaccia grafica: l'utente avrà la possibilità di aggiungere oggetti alla scena tramite un apposito pulsante presente nel tab Scena che aprirà una finestra dalla quale si avrà la possibilità di selezionare la tipologia dell'oggetto da creare da un elenco e di personalizzare le sue caratteristiche base.

6. La tipologia di oggetti disponibile saranno: sfera, piano.

7. Possibilità di selezionare gli oggetti da un elenco oppure direttamente cliccando con il tasto destro del mouse sul oggetto nel pannello preview.

8. Gli oggetti selezionati potranno essere modificati o eliminati tramite il tab Oggetto.

9. Possibilità di personalizzare e avviare il rendering tramite il tab Render.

10. Possibilità di salvare/caricare la scena tramite un apposito pulsante presente nel menu.

11. L'inquadratura della camera di rendering sarà visibile come una porzione rettangolare dell'immagine visualizzata nel pannello preview. La porzione di immagine non appartenete all'inquadratura della camera di rendering sarà leggermente oscurata in modo che l'utente possa sempre distinguere chiaramente l'inquadratura effettiva che avrà nel rendering.

12. Possibilità di posizionare la camera tramite interfaccia grafica: modificando i parametri presenti nel tab Render.

13. Possibilità di posizionare la camera tramite mouse e tastiera.

14. Gli oggetti saranno capaci di riflettere, rifrangere e assorbire la luce.

15. La camera avrà un sistema di anti-aliasing.

2.2 Requisiti facoltativi

1. Tipologie di oggetti: cubi, cilindri, triangoli, poligoni, dischi, toroidi, coni.

2. Possiblità di ruotare gli oggetti.

3. Possiblità di scalare gli oggetti.

4. Possibilità di importare mesh in formato obj.

5. L'anteprima dell'immagine avrà parametri di rendering dinamici: una volta renderizzata e mostrata a schermo l'anteprima con parametri base, se l'utente non fa altre modifiche che neccessitano un nuovo rendering, verranno avviati e mostrati in successione una serie di rendering con qualità maggiore, in modo da avere un'anteprima più vicina possibile al rendering finale.

3. Progetto

3.1 Architettura del sistema software

Si farà uso della variante del pattern architetturale MVC (Model View Controller) ove ogni interazione tra Model e View risulta completamente "mediata" dal Controller; tale variante è anche denominata MVA (Model View Adapter).

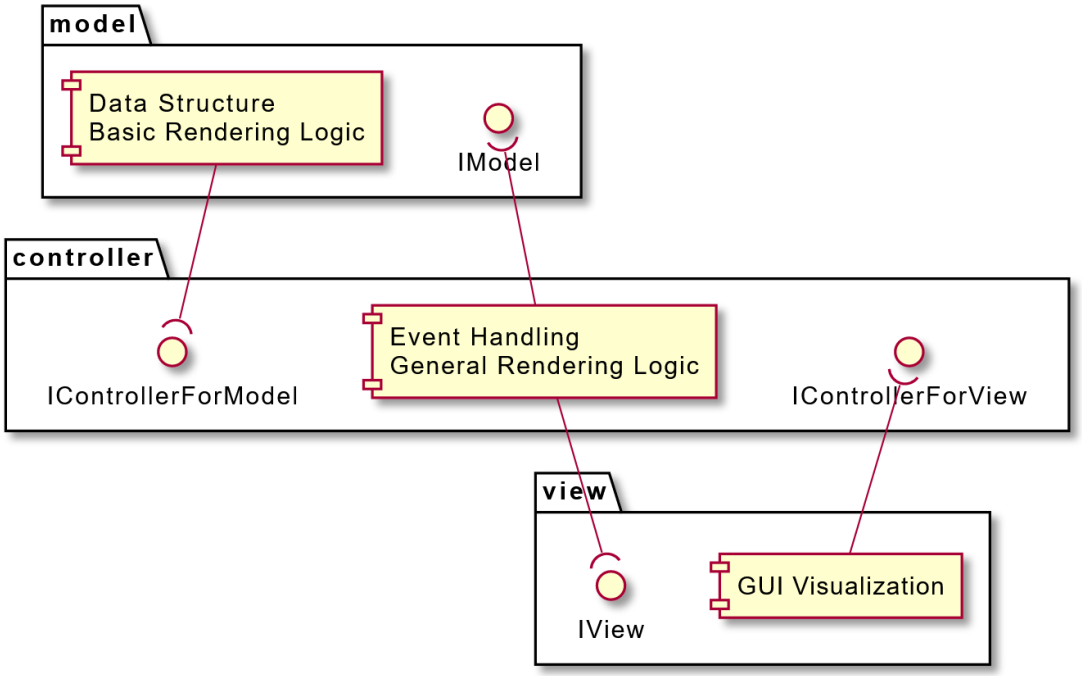

Figura 3.1: Architettura MVC - variante MVA

Si farà inoltre ampio uso del pattern Singleton per ottenere in modo semplice l'accesso ad oggetti concreti che implementano tali interfacce. Nello specifico,

tale pattern garantirà l'esistenza di un unico oggetto di tipo IModel, IControllerForModel, IControllerForView e IView. L'oggetto di tipo IModel sarà ottenibile invocando il metodo statico getInstance() della classe Model; in modo analogo si potranno ottenere le istanze uniche delle altre interfacce appena citate.

Oltre ai moduli già citati il sistema software prevede un ulteriore modulo, chiamato utils che contiene classi accessibili a tutti gli altri moduli.

3.2 Descrizione dei moduli

Di seguito vengono descritti nel dettaglio i moduli precedentemente citati, facendo riferimento anche alle classi che contengono.

3.2.1 Model

Il *model* si occupa di gestire i dati relativi alla scena, ovvero gli oggetti che contiene e la camera, inoltre contiene la logica base di rendering.

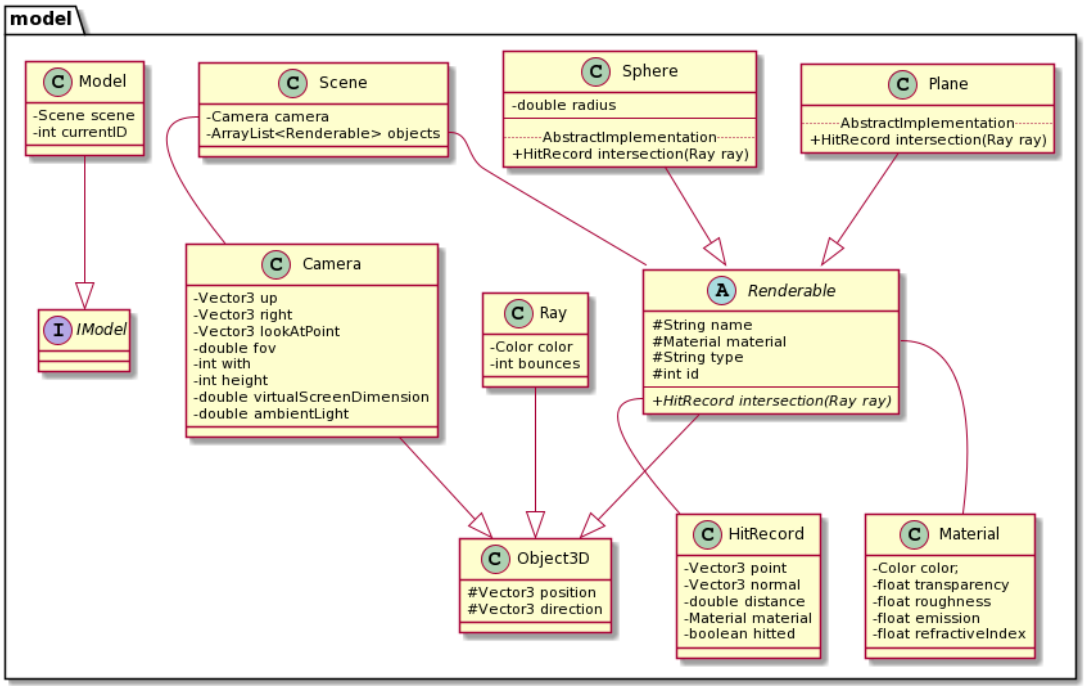

Figura 3.2: Il modulo Model

Ogni oggetto che risiede nello spazio estende(anche non direttamente) la classe *Object3D* che ha gli attributi *position* e *direction* che permettono di identificare l'oggetto nello spazio. Le classi che estendono *Object3D* sono:

- *Ray*: identifica un raggio di luce, contiene gli attributi *color* e *bounces* che rappresentano rispettivamente il colore del raggio e il numero di rimbalzi fatti dal raggio.

- *Camera*: fornisce un'astrazione di una camera reale, è in grado di emettere raggi(*Ray*) e proiettarli attraverso lo schermo virtuale. Il metodo fondamentale è *getPixelColor(int x, int y, int samples, ArrayList<Renderable> objects)* che è in grado di restituire il colore(in formato int) del pixel *(x,y)* del suo schermo virtuale preso con un numero *samples* di campioni.

- *Renderable*: è una classe astratta: ha il metodo astratto *HitRecord intersection(Ray ray)* che restituisce un *HitRecord* dell'intersezione tra il *ray* e l'ogetto *Renderable* su cui viene invocato.

Gli oggetti che effettivamente possono essere renderizzati sono classi che estendono la classe astratta *Renderable* e quindi implementano il metodo *HitRecord intersection(Ray ray)*. JRenderEngine offre la possibilità di renderizzare sfere e piani grazie alle classi:

- *Sphere*: oltre agli attributi ereditati da *Renderable* contiene l'attributo *radius*, l'attributo direction ereditato da *Object3D* non viene usato da questa classe. Una sfera infatti necessita della posizione, del raggio e del materiale per poter essere renderizzata.

- *Plane*: non necessita di altri attributi oltre a quelli ereditati da *Renderable* e *Object3D*, il piano è quindi rappresentato da due vettori (*Vector3*) che rappresentano la posizione e la direzione della normale al piano e dal materiale.

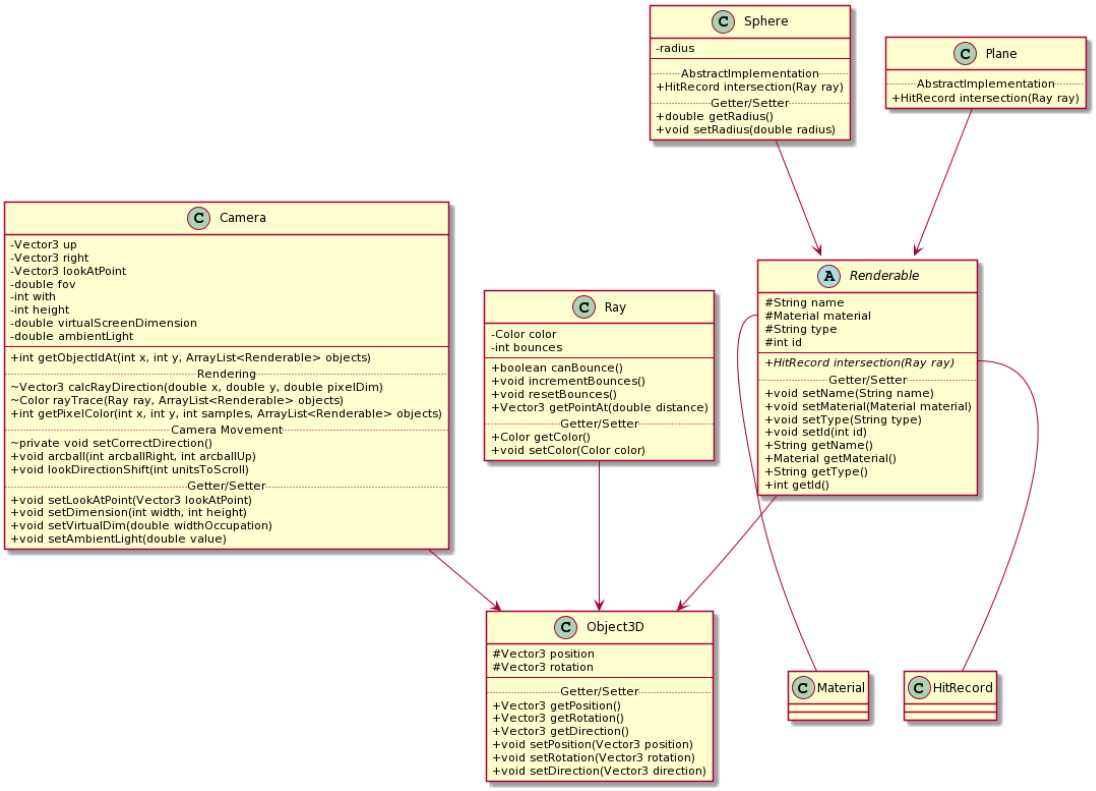

Figura 3.3: Gerarchia oggetti 3d

La classe *Material* rappresenta il materiale di un *Renderable*, ovvero un elenco di parametri che descrivono come la luce ci interagisce. I parametri contenuti sono:

- Il colore
- La trasparenza
- La rividità
- L'emissione di luce
- L'indice di rifrazione

La classe contiene il metodo *Color lightMaterialInteraction(Ray ray, HitRecord hit)* che devia il raggio nella direzione corretta e restituisce il colore che il raggio assumerà dopo l'interazione.

La classe *HitRecord* serve ad incapsulare le informazione relative a una intersezione *Renderable-Ray*, in particolare le informazioni contenute sono:

- Il punto di intserezione (*Vector3*)

- Il vettore normale alla superficie (*Vector3*)
- La distanza percorsa dal raggio
- Il materiale dell'oggetto colpito
- Un booleano che ci dice se c'è stata o meno un intersezione

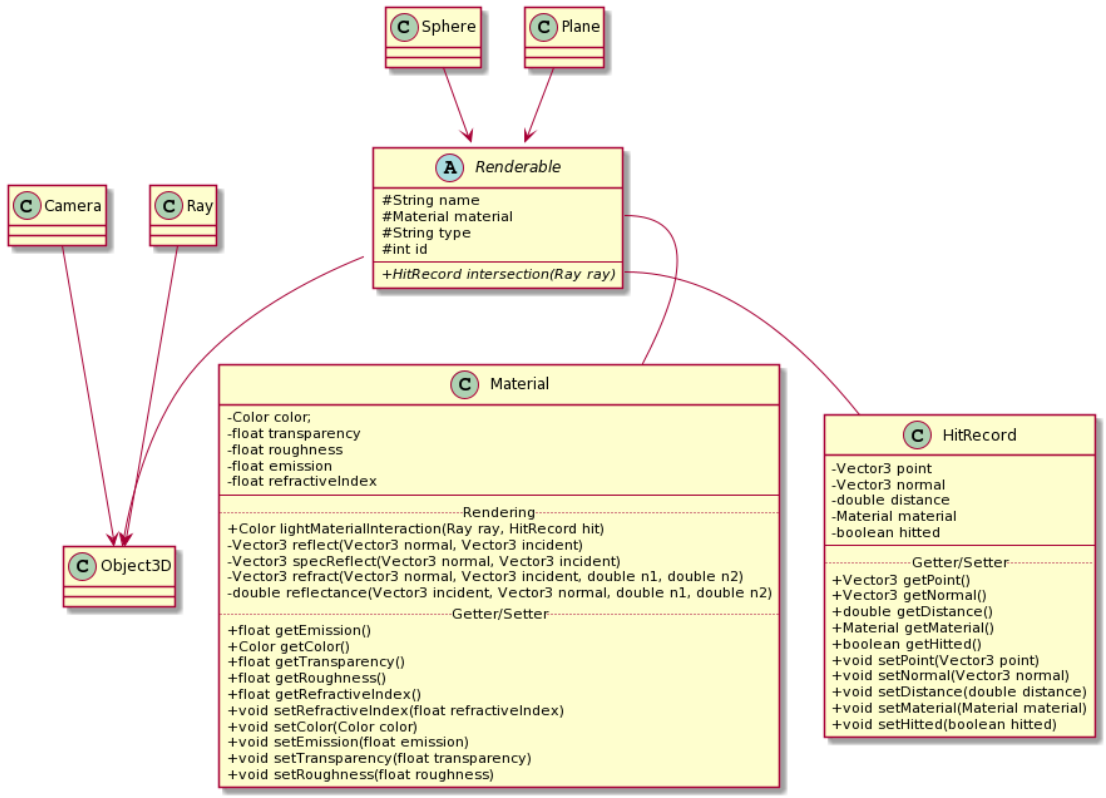

Figura 3.4: Material e HitRecord

La classe *Scene* rappresenta una collezione di *Renderable* e una *Camera*, fornisce un metodo per l'aggiunta di oggetti e di una camera. Inoltre ha due metodi statici per la conversione di *Renderable* in *ObjectProperties* e viceversa. Il metodo *loadScene(SceneFile scn)* permette di caricare una scena da file.

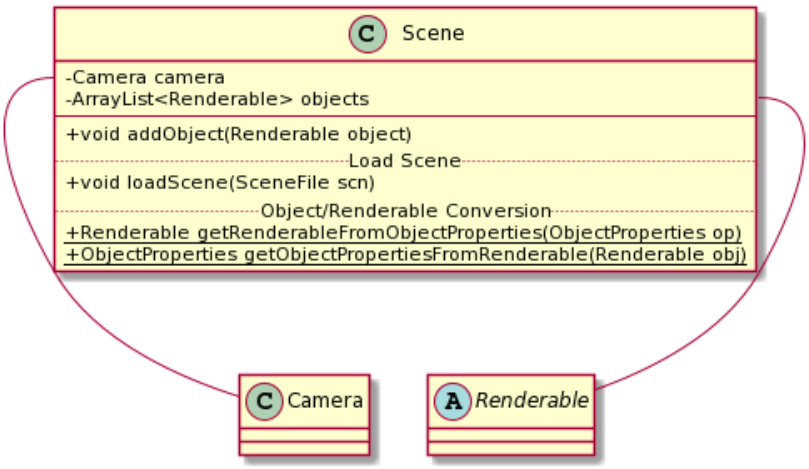

Figura 3.5: Scena contenente la camera e un array di Renderable

La classe *model* è implementata utilizzando il pattern Singleton ed estende l'interfaccia *IModel*. *IModel* fornisce l'interfaccia necessaria al Controller per caricare e salvare una scena, gestire la camera e manipolare gli oggetti presenti nella scena.

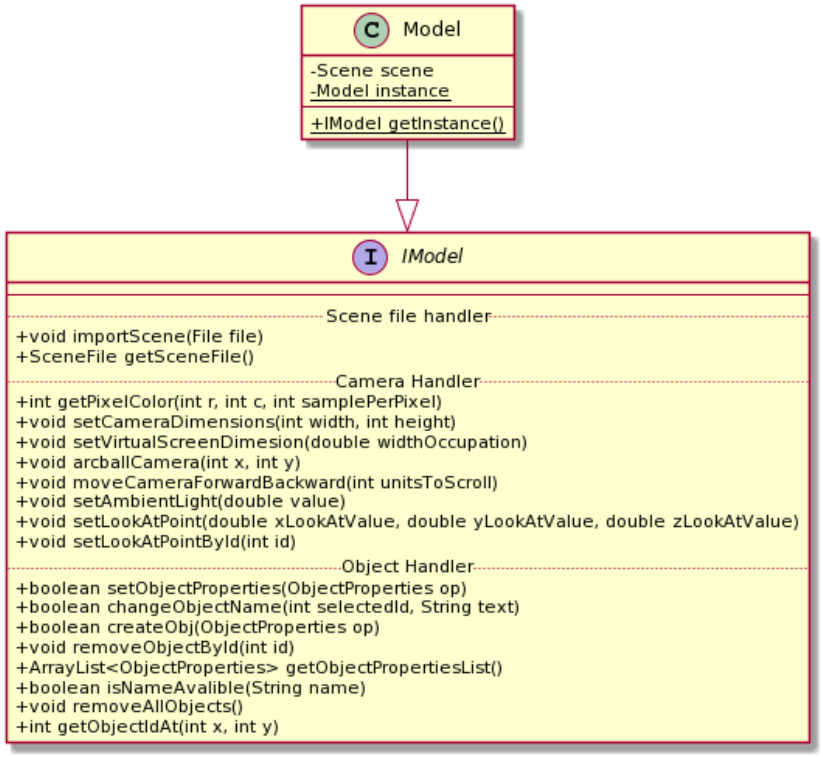

Figura 3.6: Interfaccia del model

Di seguito una descrizione dei metodi più importanti dell'interfaccia *IModel*:

- *void importScene(File file)*: carica una scena da file.
- *boolean setObjectProperties(ObjectProperties op)*: imposta le proprietà di un oggetto già esistente.
- *boolean createObj(ObjectProperties op)*: crea un nuovo oggetto.
- *void removeObjectById(int id)*: rimuove un oggetto tramite l'id.

Come si può vedere dal diagramma UML l'interfaccia presenta anche una serie di metodi per la gestione della camera.

3.2.2 View

Il modulo *view* si occupa di mostrare a schermo una preview della scena e l'interfaccia che consente all'utente di manipolarla.

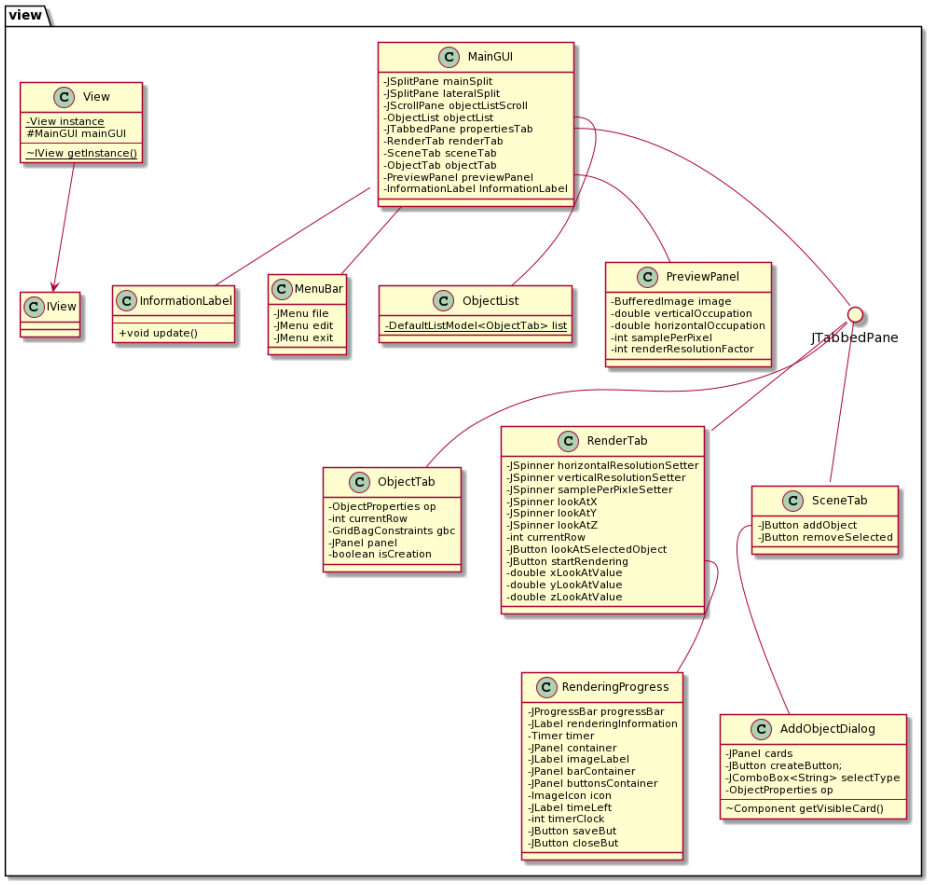

Figura 3.7: Diagramma del modulo view

La classe *mainGUI* rappresenta la finestra principale dell'applicazione, contiene i metodi per inizializzare l'interfaccia.

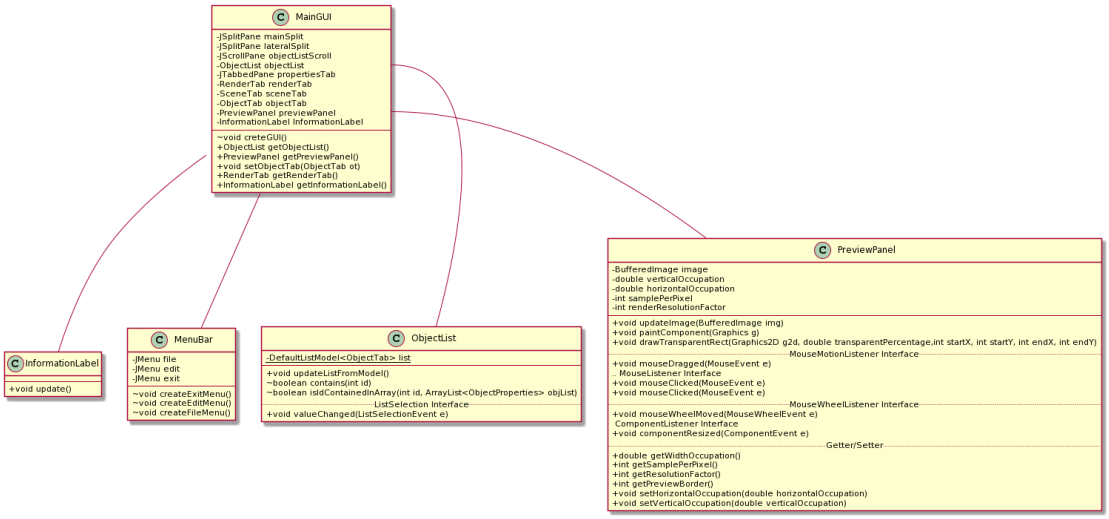

Figura 3.8: MainGUI

I componenti principali dell'interfaccia sono:

- *InformationLabel* estende *JLabel* e fornisce all'utente informazione sulla scene, come il numero di oggetti presenti e la complessità in una scala di decimi.
- *MenuBar* estende *JMenuBar* e mette a disposione tre sottomenu: per la gestione dei file, la modifica della scena e la chiusura dell'applicazione.
- *ObjectList* estende *JList<ObjectTab>* e mostra a schermo una lista degli oggetti presenti nella scena, offre anche la possibilità di selezionare di oggetti da essa.
- *PreviewPanel* estende *JPanel* e serve a mostrare una preview in tempo reale della scena renderizzata a bassa risoluzione. Inoltre tale classe implementa i metodi in grado di recepire gli input dell'utente per il movimento della camera e la selezione degli oggetti: in particolare è possibile muovere la camera lungo i punti di una superficie sferica tramite scorrimento del mouse, diminuire o aumentare il raggio della sfera su cui la camera si muove tramite la rotella del mouse, selezionare gli oggetti cliccando con il mouse destro su di essi.

L'interfaccia principale presenta anche un *JTabbedPane* contenente tre *Tab*:

- *ObjectTab*: permette di modificare le properietà dell'oggetto selezionato.
- *RenderTab*: Permette di configurare le impostazioni di rendering e avviare il rendering.
- *SceneTab*: Permette di aggiungere o rimuovere oggetti nella scena e di modificare l'intesità della luce ambientale.

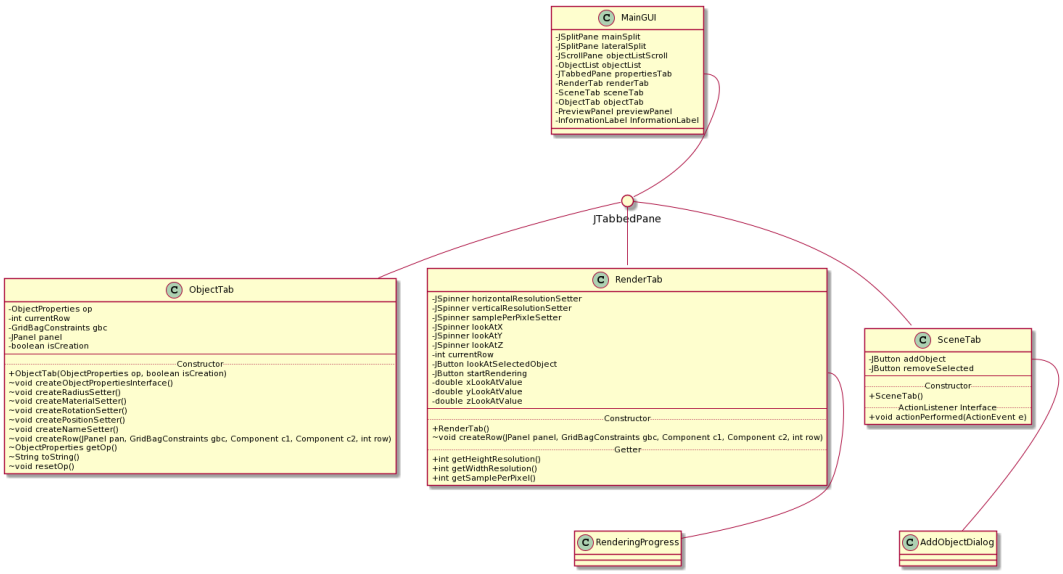

Figura 3.9: JTabbedPane

L'aggiunta di oggetti avviene tramite la classe *AddObjectDialog* che estende *JDialog*, mentre il rendering ad alta qualità viene mostrato tramite la classe *RenderingProgress* che estende *JDialog*.

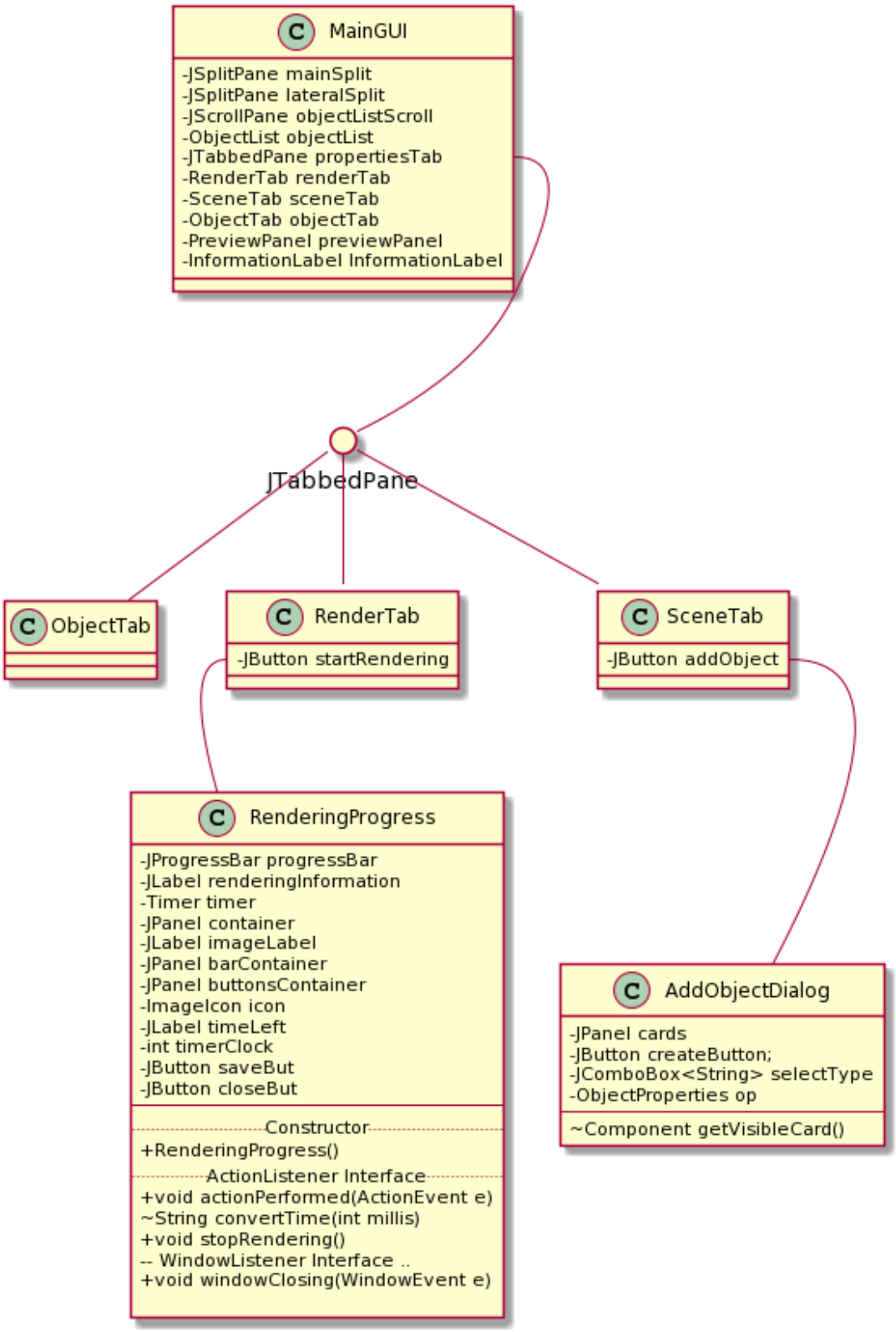

Figura 3.10: Le classi RenderingProgress e AddObjectDialog

La classe *View* è implementata utilizzando il pattern Singleton ed estende l'interfaccia *IView*. *IView* fornisce l'interfaccia necessaria al Controller per la gestione dell'interfaccia grafica.

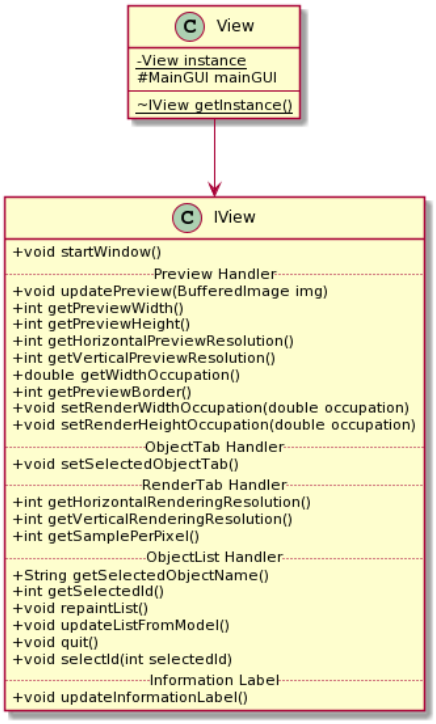

Figura 3.11: L'interfaccia IView

Di seguito un elento dei metodi principali di *IView*:

- *void startWindow()*: permette di avviare la finestra principale.
- *void updatePreview(BufferedImage img)*: aggiorna la preview.
- *void setSelectedObjectTab()*: aggiorna il tab *ObjectTab* con le impostazione dell'oggetto selezionato.
- *void updateListFromModel()*: aggiorna la lista degli oggetti in modo che risulti compatibile con la lista presente nel model.
- *void updateInformationLabel()*: aggiorna le informazione nell' *Information-Label*.

3.2.3 Controller

Il modulo *controller* si occupa di mappare gli input dell'utente agli aggiornamenti del *model*, di aggiornare la view e di una gestione generale del rendering. Il modulo presenta 3 classi: *Main* per l'avvio dell'applicazione, *ControllerForModel* e *ControllerForView* entrambe realizzate con il pattern Singleton e implementano le interfacce *IControllerForView* e *IControllerForModel*. *ControllerForModel* contiene un unico metodo che si occupa di fornire un id unico ad ogni sua chiamata, mentre *ControllerForView* si occupa di gestire gli input dell'utente e gestire il rendering a livello alto.

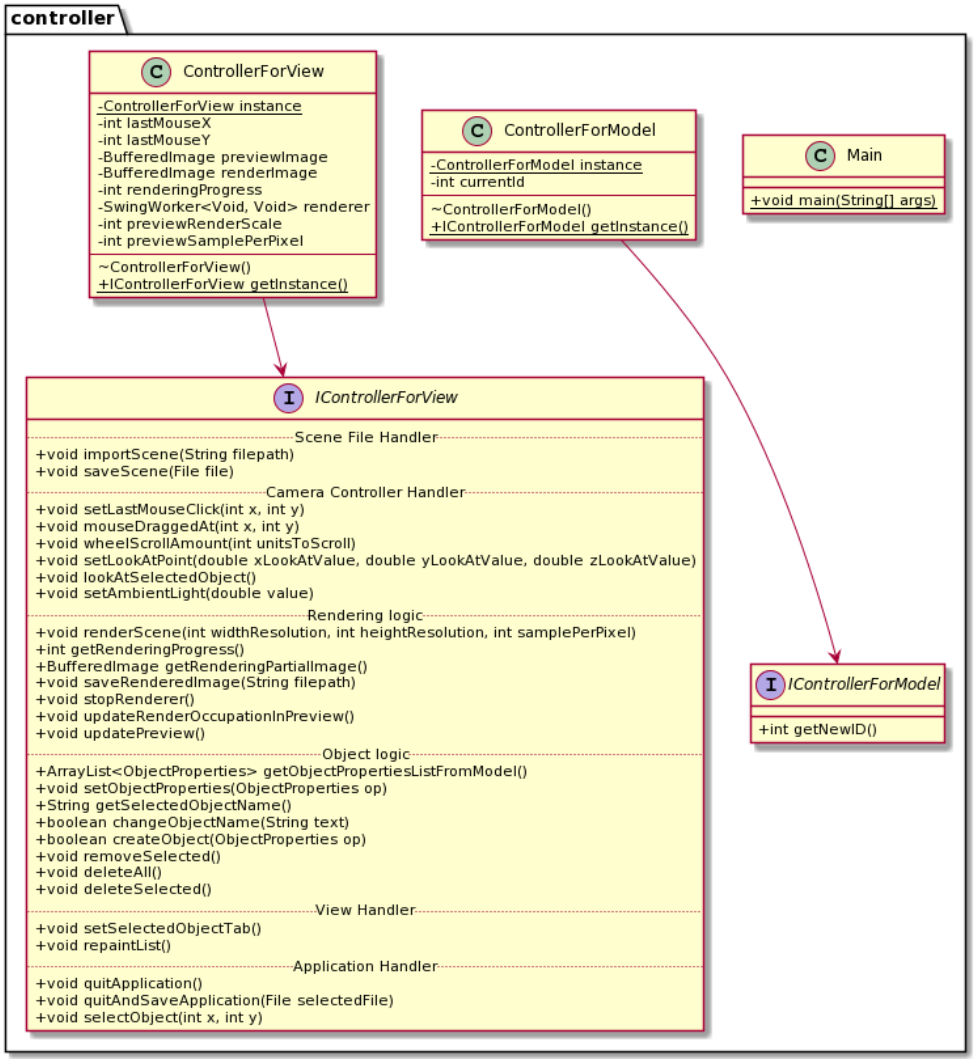

Figura 3.12: Diagramma del modulo Controller

Di seguito un elenco dei metodi più sigificativi dell'interfaccia *IControllerForView*:

- *void importScene(String filepath)*: si occupa di chiamare il metedo *void loadScene(String filepath)* di *model* e di aggiornare la view.
- *void saveScene(File file)*: salva la scena corrente su file.
- Un insieme di metodi per la gestione della camera: *void mouseDraggedAt(int x, int y)*, *void wheelScrollAmount(int unitsToScroll)*, *void setLookAtPoint(double xLookAtValue, double yLookAtValue, double zLookAtValue)*, *void lookAtSelectedObject()*, *void setAmbientLight(double value)*, *void setLastMouseClick(int x, int y)*.
- *void renderScene(int widthResolution, int heightResolution, int samplePerPixel)*: avvia uno *SwingWorker* in grado di renderizzare la scena.
- *int getRenderingProgress()* e *BufferedImage getRenderingPartialImage()*: usati per controllare lo stato del rendering.
- *void saveRenderedImage(String filepath)*: salva l'immagine renderizzata su file.
- *void stopRenderer()*: ferma il rendering.
- *void updateRenderOccupationInPreview()*: imposta la corretta occupazione del immagine di rendering nella preview.
- *void updatePreview()*: aggiorna l'immagine di preview.
- *ArrayList<ObjectProperties> getObjectPropertiesListFromModel()*: restituisce un array di ObjectProperties degli oggetti presenti nella scena.
- *void setObjectProperties(ObjectProperties op)*: modifica le proprietà di un oggetto nella *Scene*.
- *String getSelectedObjectName()*: restituisce il nome dell'oggetto selezionato.
- *boolean changeObjectName(String text)*: modifica il nome dell'oggetto selezionato.
- *boolean createObject(ObjectProperties op)*: crea un nuovo oggetto con le proprietà *op*.
- *void removeSelected()*: rimuove l'oggetto selezionato.

- *void deleteAll()*: rimuove tutti gli oggetti presenti nella scena.
- *void setSelectedObjectTab()*: imposta il corretto *ObjectTab* da visualizzare.
- *void repaintList()*: aggiorna l'interfaccia della lista di oggetti.
- *void quitApplication()*: si occupa di chiudere l'applicazione.
- *void quitAndSaveApplication(File selectedFile)*: salva la scena su file e chiude l'applicazione.
- *void selectObject(int x, int y)*: si occupa di gestire l'input relativo alla selezione degli oggetti tramite il pannello preview.

3.2.4 Utils

Il modulo *utils* serve a semplificare le classi degli altri moduli, infatti fornisce delle classi per la gestione dei colori, vettori e matrici, l'interscambio di dati riguardanti gli oggetti renderizzabili tra il *view* e il *model* e una classe utile a semplificare la gestione delle *Scene* su file.

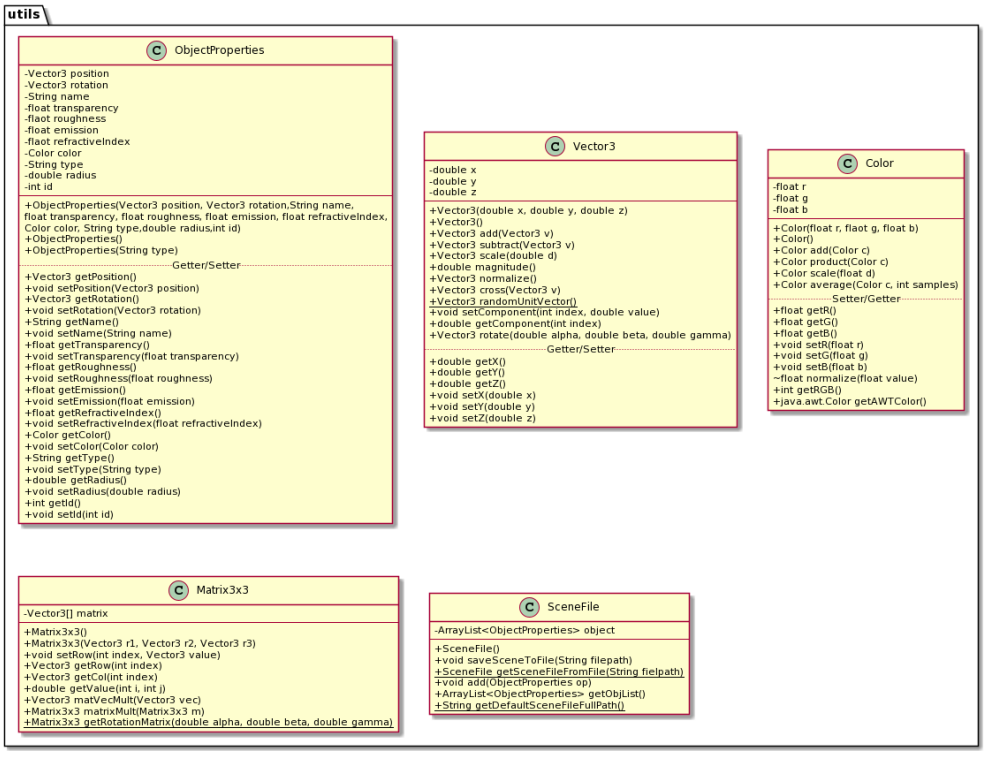

Figura 3.13: Diagramma del modulo utils

Di seguito una breve descrizione delle classi contenute:

- *Vector3*: rappresenta un vettore tridimensionale, fornisce le operazione vettoriali neccessarie alla gestione di oggetti tridimensionali.
- *Matrix3x3*: rappresenta una matrice 3x3 e fornisce un metodo per eseguire il prodotto vettore-matrice e un metodo statico che restituisce una matrice di rotazione. Da notare che la classe è poco utilizzata ma potrebbe risultare di grande utilità nel caso di estensioni all'applicazione.
- *Color*: permette di rappresentare un colore tramite le sue componenti *r,g,b*, mette a disposizioni delle operazioni tra colori.
- *ObjectProperties*: fornisce un formato di interscambio dei dati riguardanti gli ogetti renderizzati. Da notare che la classe complica l'estensibilità dell'applicazione ma semplifica notevolmente il *model* e il *view*.
- *SceneFile*: è in grado di scrivere/leggere su file un insieme di ObjectProperties, inoltre fornisce un metodo statico in grado di restituire il path assoluto del file relativo alla scena che viene caricata di default all'avvio dell'applicazione grazie all'uso di *Reflection*.

Il modulo *utils* non è strettamente necessario al funzionamento dell'applicazione, la maggior parte delle sue funzionalità potevano essere implemetate nei moduli *model* e *controller*, si è comunque deciso di implementarlo al fine di non complicare troppo gli altri moduli.

3.3 Problemi riscontrati

Uno dei problemi riscontrati è stato la gestione dei colori: inizialmente si voleva estendere la classe *java.awt.Color* che però prensentava il limite di componenti(r,g,b) vincolate nel range 0-1, il model invece richiedeva di superare questo limite per poter avere fonti emissive di luce con intesità > 1. Si è quindi deciso di implementare una classe *Color* da zero che però fornisse il metodo *int getRGB()* necessario alla creazione di *BufferedImage*.

Un'altro probelma che si è presentato è stato la modifica degli oggetti tramite interfaccia grafica: avendo implementato l'architettura MVA(Model View Adapter) le classi del *view* non hanno direttamente accesso ai metodi dell'interfaccia del *model* e visto che ogni oggetto presenta molti parametri modificabili, ogni loro modifica risultava in una chiamata al controller che a sua volta faceva una chiamata al model in modo ridondante. Si è quindi deciso di implementare la classe

ObjectProperties presente del modulo *utils* al fine di semplificare tale comportamento. La classe *ObjectProperties* è inoltre risultata di grande utilità anche nella gestione delle scene su file.

4. Appendice

4.1 Descrizione del rendering

Nonostante la precendente descrizione dei moduli si ritiene comunque necessario approfondire le varie fasi del rendering.

Si è deciso di suddividere il rendering dell'immagine nei tre moduli(*model*, *view*, *controller*) nel seguente modo: *model* si occupa di fornire il colore di un singolo pixel dell'immagine(rendering a basso livello), *controller* si occupa di "assemblare" l'immagine, ovvero di creare una matrice di pixel(rendering ad alto livello), *view* si occupa di mostrare il risultato a schermo.

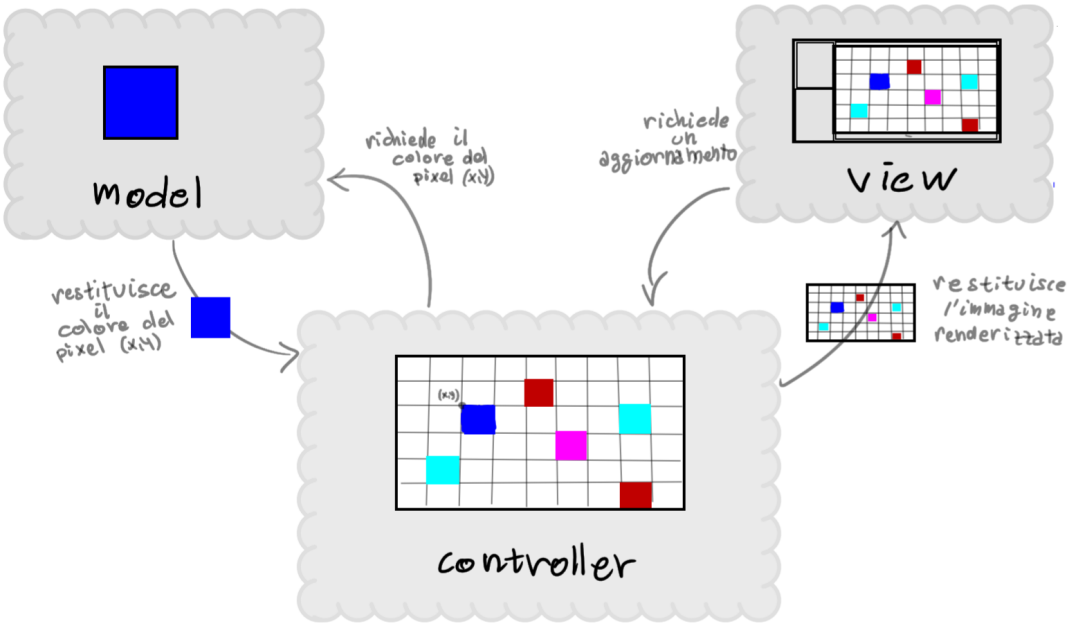

Figura 4.1: Rendering

In particolare il calcolo del colore avviene grazie alle classi *Camera* e *Material*: la classe *Camera* presenta il metodo *getPixelColor* che si occupa di generare i raggi e restituire il colore finale. Il metodo *getPixelColor* fa inoltre uso di altri due metodi privati: *calcRayDirection* utile a calcolare la direzione del raggio e del metodo *rayTrace* capace di trovare l'intersezione piu vicina tra il raggio emesso e gli oggetti della scena, una volta avvenuta l'intersezione il metodo viene richiamato in modo ricorsivo non prima però di aver calcolato il colore del raggio e la sua direzione grazie al metodo *lightMaterialInteraction* della classe *Material*.

4.2 Possibilità di estensione

Il sistema è stato progettato cercando di renderlo il più estensibile possibile. É infatti molto semplice aggiungere nuovi oggeti renderizzabili al *model*: basta creare classi che estendono la classe astratta *Renderable*, infatti l'unico requisito che deve avere un oggetto per essere renderizzato è quello di implementare in maniera concreta il metodo *intersection* che restituisce un *HitRecord* dell'intersezione. L'estensibilità viene però complicata nel caso del *view*: in caso di aggiunta di nuovi tipi di oggetti renderizzabili è infatti necessario implementare l'interfaccia per ogni nuovo attributo modificabile dall'utente, si devono inoltre aggiornare le classe *ObjectProperites* e *SceneFile*.

4.3 Confronto dei risultati con il software Blender

Blender è un software libero e multipiattaforma di modellazione, rigging, animazione, montaggio video, composizione, rendering e texturing di immagini tridimensionali e bidimensionali. JRenderEngine tenta di riprodurre la parte dedicata al rendering di immagini. Al fine di verificare la validità delle immagini prodotto si è cercato di fare un confronto tra i due software facendo un rendering della stessa scena. Il formato di salvataggio delle scene su file è ovviamente diverso, si è pertanto cercato di riprodurre la scena manualmente all'interno di Blender. Il confronto non è quindi molto accurato, in particolare non si è riuscito a riprodurre le condizioni di luce ambientale(fattore che influenza notevolmente i risultati). Tuttavia i risultati prodotti sono abbastanza soddisfacenti, in entrambi i programmi si è definito: un piano con materiale diffusivo verde, una sfera diffusiva rossa, una speculare e una trasparente, inoltre si è definita anche una sfera emissiva posta in lontananza. Anche i tempi di rendering risultano del tutto confrontabili, entrambi i software hanno impiegato circa 1 min a renderizzare l'immagine.

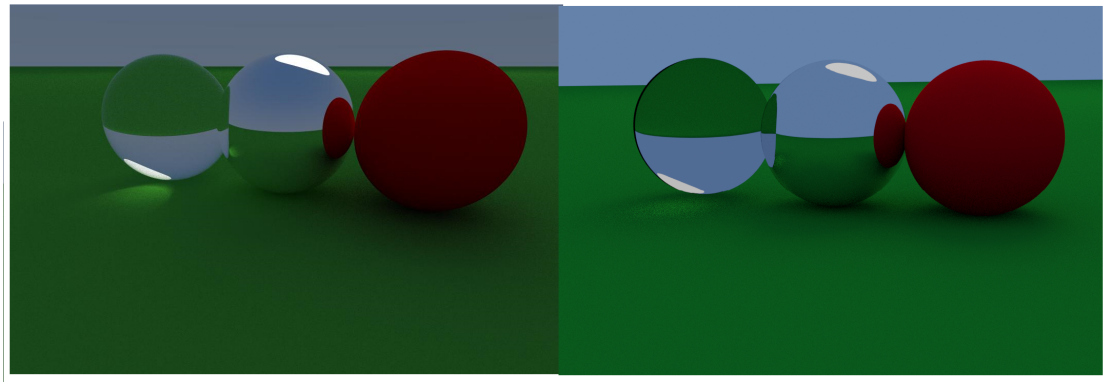

Immagine generata con JRenderEngine ad una risoluzione di 1080x720 con 100 sample per pixel

Immagine generata con Blender ad una risoluzione di 1080x720 con 100 sample per pixel

Figura 4.2: Confronto con Blender

5. Bibliografia

- RayTracing https://en.wikipedia.org/wiki/Ray$_{t}racing_{(}graphics$)
- Snell's Law https://en.wikipedia.org/wiki/Snell's$_{l}aw$
- Reflection/Refraction https://graphics.stanford.edu/courses/cs148-10-summer/docs/2006–degreve–reflection$_{r}efraction.pdf$
- Diffuse Reflection https://en.wikipedia.org/wiki/Diffuse$_{r}eflection$