ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

DIPARTIMENTO DI INFORMATICA - SCIENZA E INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA E SCIENZE INFORMATICHE

Visualizzazione degli effetti gravitazionali sui fotoni attraverso tecniche di Ray-Tracing

Elaborato in:
COMPUTER GRAPHICS

Relatore
Prof.ssa Damiana Lazzaro

Presentata da
Matteo Catena

Sessione unica
Anno Accademico 2024-2025

Sommario

Tra tutti i fenomeni fisici che avvengono nell'universo quando lo si analizza su larga scala, ce n'è uno in particolare che visivamente è molto affascinante: lo studio delle lenti gravitazionali. Con lente gravitazionale si fa riferimento a un corpo celeste con massa talmente elevata da avere effetto sui raggi di luce, deviandoli dalla loro traiettoria rettilinea. La particolarità di queste lenti è che la deviazione aumenta più il raggio di luce passa vicino al corpo e diminuisce fino a diventare trascurabile più la distanza aumenta, per cui non hanno un vero e proprio punto focale come le lenti tradizionali.

Nonostante esistano varie simulazioni di questo fenomeno, la maggior parte di esse sono limitate in un modo o nell'altro, rendendole inutili a livello pratico e soltanto come strumenti di svago. Questo progetto ha l'obiettivo di creare una simulazione fisica che risolva le limitazioni trovate in altri progetti, con l'obiettivo di essere il più utile possibile all'analisi astronomica dell'universo.

Il progetto si chiama GraviLenSim ed è disponibile online all'indirizzo

https://github.com/MatteoGodzilla/GraviLenSim

Indice

Introduzione

no dei fenomeni fisici più affascinanti quando si studia l'universo in larga scala sono le distorsioni causate dalle lenti gravitazionali. Con lente gravitazionale si indica una categoria di fenomeni fisici dove corpi celesti hanno una massa talmente elevata da curvare la traiettoria percorsa dalla luce, formando una o più immagini distorte a seconda della tipologia di fenomeno. Il metodo con cui queste lenti deviano la luce viene spiegato attraverso la teoria della Relatività Generale di Einstein, mettendo in relazione la massa del corpo celeste e la distanza tra corpo e fotone con l'angolo di deviazione subito dal raggio di luce.

Nonostante questi fenomeni fisici siano noti da più di un secolo e confermati più recentemente, la maggioranza dei simulatori fisici di questo tipo contengono serie limitazioni dal punto di vista della usabilità, che li rendono inadatti ad un utilizzo di ricerca spaziale universale. GraviLenSim è un programma che cerca di colmare queste lacune di usabilità, con l'obiettivo di offrire uno strumento che possa essere effettivamente di aiuto durante l'analisi delle immagini ottenute tramite satelliti.

In questa tesi si vuole descrivere il funzionamento di GraviLenSim, progetto sviluppato in C++ che comprende un Ray-Tracer implementato in OpenGL 4.3 attraverso l'utilizzo di Compute Shader, iniziando con i concetti base fondamentali del fenomeno fisico e del Ray-Tracing, per poi descrivere la struttura del progetto in sè e infine andare nel dettaglio dell'implementazione grafica.

In particolare, nel capitolo 1 verranno descritte le lenti gravitazionali sia dal punto di vista storico che dal punto di vista fisico della Relatività Generale, con l'obiettivo di definire il modello fisico utilizzato all'interno della simulazione.

Nel capitolo 2 si proseguirà con la descrizione del Ray-Tracing, le basi ottiche necessarie per capire il funzionamento principale per poi estendere il discorso al contesto universale con corpi massicci.

Una volta terminati i discorsi sui concetti base si procederà con la descrizione dettagliata del progetto, suddivisa in tre capitoli:

- Nel capitolo 3 si analizzeranno i requisiti del progetto, verrà descritta l'architettura implementata e si descriverà la componente logica del progetto eseguita principalmente sulla CPU.
- Il capitolo 4 sarà interamente dedicato alla descrizione approfondita del Ray-Tracer e della sua implementazione all'interno del progetto.
- Infine nel capitolo 5 si descriverà l'editor grafico utilizzato per definire l'universo fisico da simulare, includendo le opzioni più importanti per l'utente utilizzatore.

La tesi si concluderà con i capitoli 6 e 7, in cui verranno mostrati rispettivamente esempi di immagini prodotte dal progetto e commenti finali sulle limitazioni attuali.

Capitolo 1

Lenti gravitazionali

In Fisica con "Lenti gravitazionali" si fa riferimento a una categoria di fenomeni fisici molto complessi e diversi tra loro. Per questo motivo è necessario avere un'idea ben chiara di questi fenomeni e come si comportano, così da poter successivamente trattare il discorso della simulazione fisica in modo chiaro e comprensibile.

1.1 Storia

Nel 1704 Isaac Newton pubblica *Opticks*, una serie di tre libri in cui raccoglie le sue osservazioni sul comportamento della luce. Nei primi due volumi lo scienziato descrive in modo dettagliato i suoi studi sperimentali, mentre nel terzo presenta una serie di domande ("queries"), attraverso le quali formula teorie e ipotesi non ancora dimostrate su diversi ambiti della scienza, non limitandosi all'ottica. Nella prima query del libro si chiede:

"Do not Bodies act upon Light at a distance, and by their action bend its Rays; and is not this action (cœteris paribus) strongest at the least distance?" [0].

Parafrasando in Italiano, Sir.Isaac Newton chiede se la gravità, che è riuscito a descrivere come attrazione tra due corpi, influenza anche i raggi luminosi, intuendo che questa azione viene fatta a distanza e che più questa distanza è breve più i raggi luminosi vengono deviati.

A questa domanda si troverà una prima risposta un secolo più tardi, nel 1804, con l'opera *Ueber die Ablenkung eines Lichtstrals von seiner geradlinigen Bewegung* di Johann Georg von Soldner. In questo lavoro

l'autore riprende la teoria di Newton sulla gravitazione per arrivare alla conclusione teorica che i raggi luminosi vengono effettivamente deviati dai corpi celesti e calcola l'angolo con cui il sole dovrebbe deviare le luci provenienti da stelle lontane, ottenendo un valore di $\tilde{\alpha} = 0.84$ arcsec [0] [0].

Questo risultato verrà confermato dal punto di vista teorico da Einstein con il suo articolo «Über den Einfluß der Schwerkraft auf die Ausbreitung des Lichtes» nella rivista *Annalen der Physik*, in cui l'angolo di deviazione viene modellato dall'equazione [0]:

$$\tilde{\alpha} = \frac{2kM}{c^2\Delta} \tag{1.1}$$

dove k è la costante gravitazionale universale di Newton, M è la massa del corpo, c la velocità della luce nel vuoto e Δ la distanza tra il raggio e il centro del corpo. Quando Einstein pubblicò questo articolo nel 1911 era ancora immerso nella definizione completa della Relatività Generale e nei quattro anni successivi di sviluppo si accorse che l'equazione 1.1 non era corretta.

Alla pubblicazione di «Die Grundlage der allgemeinen Relativitätstheorie» nel 1915, l'equazione 1.1 venne corretta cambiando la costante iniziale da $2kM$ a $4kM$, raddoppiando di fatto l'angolo di deviazione teorizzato:

$$\tilde{\alpha} = \frac{4kM}{c^2\Delta} \tag{1.2}$$

Quale delle due equazioni era corretta? La risposta ci fu con la misurazione effettuata nell'eclissi del 1919 da Dyson, Eddington e Davidson. In quell'occasione si misurò la deviazione della luce di una stella al bordo del sole di un angolo $\tilde{\alpha} = 1.75 \pm 0.30$ arcsec, dimostrando come l'equazione 1.2 fosse effettivamente corretta. Questa rilevazione fu fondamentale perchè diede molta credibilità alla teoria di Einstein, consolidandola dopo la risoluzione delle anomalie osservate nell'orbita di Mercurio, in un contesto in cui ancora la teoria fisica di Newton era considerata affidabile.

Nonostante nel 1919 si dimostrò la validità della teoria di Einstein, si dovette attendere fino al 1979 per avere la prima rilevazione di una lente gravitazionale. In quell'anno Walsh, Carswell e Weymann misurarono lo spettro elettro-magnetico emesso da due quasar talmente simili e vicini tra loro da escludere la possibilità di essere due corpi diversi, interpre-

tandoli invece come due immagini dello stesso corpo, ottenute a causa di una lente gravitazionale [0]. Per confermare questa ipotesi modellarono lo spazio-tempo coinvolto secondo la teoria della Relatività Generale, ottenendo valori che concordavano con quelli misurati.

1.2 Tipologie di fenomeni fisici

Gli effetti fisici causati dalle lenti gravitazionali si classificano in tre categorie [0]:

1. Strong Lensing: quando un oggetto più vicino A separa la luce proveniente da un oggetto più lontano B in modo tale da formare più immagini distinte dell'oggetto B osservate da un punto di vista esterno. Questo fenomeno si verifica soltanto se i due oggetti sono abbastanza allineati tra loro rispetto all'osservatore, condizione che fu osservata per la prima volta nel 1979 [0]. Rientrano sotto questa categoria tutti i tipi di archi e anelli di Einstein.

2. Weak Lensing: quando un oggetto più vicino A distorce la luce proveniente da un oggetto più lontano B, senza formare più immagini distinte. Questo può accadere quando i due oggetti non sono allineati rispetto all'osservatore ma la massa dell'oggetto A è talmente elevata da distorcere comunque l'immagine di B. Questo fenomeno è osservabile soltanto analizzando più sorgenti in contemporanea e attraverso calcoli di tipo statistico che permettono di ricostruire quella che deve essere la massa in superficie di A responsabile di tali distorsioni.

3. Microlensing: quando un oggetto più vicino A devia la luce proveniente da un oggetto più lontano B in più immagini ma solo una di queste raggiunge l'osservatore. Di conseguenza l'immagine osservata varia nel tempo a causa del movimento relativo tra osservatore, lente A e sorgente B.

1.3 Studio della materia oscura attraverso lenti gravitazionali

L'importanza scientifica delle lenti gravitazionali è aumentata notevolmente negli ultimi anni grazie alla scoperta della materia oscura. Attualmente la definizione più completa che abbiamo della materia oscura riguarda particelle WIMP (Weakly Interacting Massive Particles), ovvero particelle che hanno una massa ma che non interagiscono elettricamente con altra materia e non emettono nessun tipo di onda elettro-magnetica (e quindi non emettono luce) [0].

Nonostante la loro natura non è ancora ben definita, negli anni sempre più esperimenti fisici e calcoli stimati di masse delle galassie hanno portato alla conclusione che 80-85% della massa presente nell'universo sia oscura [0], per cui oggi la presenza di materia oscura non è più messa in discussione dalla comunità scientifica.

Tra le varie tipologie di esperimenti fisici fatti per rilevare la presenza di materia oscura, quello dello studio delle lenti gravitazionali rimane uno dei più importanti. Questo perchè nonostante la materia oscura non interagisca *direttamente* con le onde elettro-magnetiche, a causa della sua massa curva comunque lo spazio-tempo, per cui le onde elettro-magnetiche risultano comunque *indirettamente* deviate.

Per fare un esempio di come le lenti gravitazionali possono essere usate nello studio della materia oscura, consideriamo lo Strong Lensing. Partendo dai dati osservativi della sorgente, come il numero di immagini prodotte, le loro dimensioni, posizione, rotazione e amplificazione, è possibile impostare un problema inverso da cui ricavare le dimensioni in massa della lente gravitazionale, da cui successivamente si ricava la quantità di massa oscura che deve essere presente in un determinato punto per deviare i fotoni in modo coerente con le osservazioni.

1.4 Accenni al modello della Relatività Generale

Nella Relatività per descrivere un punto nello spazio non basta descrivere le coordinate spaziali ma è necessario indicare anche una coordinata

temporale. Convenzionalmente questo viene fatto utilizzando un vettore quadrimensionale $\mathbf{X}$ del seguente tipo:

$$\mathbf{X} \equiv \begin{bmatrix} w \\ x \\ y \\ z \end{bmatrix} \equiv \begin{bmatrix} ct \\ x \\ y \\ z \end{bmatrix} \tag{1.3}$$

dove c indica la velocità della luce nel vuoto.

$\mathbf{X}$ si chiama "Evento" e le sue coordinate non sono assolute ma sono calcolate a partire da un sistema di riferimento inerziale (S.R.I.) scelto a priori. Per ottenere le coordinate di un evento definito in un S.R.I. A in un altro S.R.I. B si utilizzano le trasformazioni di Lorentz.

Definiamo ora la norma del vettore quadrimensionale $\mathbf{X}$:

$$|\mathbf{X}|^2 \equiv \mathbf{X}^T \eta \mathbf{X} \tag{1.4}$$

dove η è una matrice del seguente tipo:

$$\eta = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \tag{1.5}$$

Questa norma ha la proprietà principale di essere invariante rispetto a trasformazioni di Lorentz. Questo piccolo dettaglio non è da trascurare perchè ci permette di misurare la distanza spazio-temporale tra due eventi, indipendentemente dal S.R.I. scelto. L'unico problema dell'equazione 1.4 è che è utilizzabile direttamente soltanto nel caso in cui i due eventi sono separati da spazio-tempo completamente piatto. Nel caso in cui lo spazio-tempo non sia completamente piatto bisogna sostituire la matrice η con g, ovvero con il "tensore metrico", ottenendo la seguente equazione:

$$s^2 \equiv |\mathbf{X}|^2 \equiv \mathbf{X}^T g \mathbf{X} \tag{1.6}$$

Arrivati a questo punto, possiamo accennare ai passi necessari per ottenere la traiettoria del fotone. Nel contesto delle lenti gravitazionali è più utile pensare ai fotoni come particelle piuttosto che come onde.

Siano $\mathbf{A}$ e $\mathbf{B}$ due eventi che rappresentano rispettivamente il punto d'inizio e il punto di fine della traiettoria percorsa dalla luce. Sia $f(p)$ la

funzione continua che associa al parametro p l'evento intermedio in cui si troverebbe il fotone lungo la sua traiettoria. Sul percorso completo da $\mathbf{A}$ a $\mathbf{B}$ vogliamo calcolare il tempo proprio impiegato dal fotone, considerando che la traiettoria seguita dalla luce è quella che minimizza tale intervallo temporale.

Per fare ciò suddividiamo il percorso definito da $f(p)$ come insieme di tratti infinitesimali $\mathrm{d}\mathbf{X}$ che congiungono eventi intermedi consecutivi, allo stesso modo con cui si può approssimare una curva nel piano euclideo come insieme di segmenti rettilinei di punti consecutivi. Su ogni tratto $\mathrm{d}\mathbf{X}$ assumiamo che la curvatura sia costante, per cui avrà una distanza infinitesimale misurata attraverso eq. 1.6 di:

$$\mathrm{d}s^2 = \mathrm{d}\mathbf{X}^T g(\mathbf{X}) \mathrm{d}\mathbf{X} \tag{1.7}$$

dove con $g(\mathbf{X})$ si indica il tensore metrico come una funzione che calcola la curvatura locale di un evento, in questo caso quello iniziale del tratto $\mathrm{d}\mathbf{X}$ (si potrebbe usare allo stesso modo l'evento finale del tratto per stabilire il tensore metrico, dato che per ipotesi abbiamo posto che $g(\mathbf{X})$ è costante nel tratto). La funzione $g(\mathbf{X})$ è il primo problema numerico che si pone nel calcolo della traiettoria, perchè di base non è facile da calcolare e deve essere ricalcolato per ogni tratto $\mathrm{d}\mathbf{X}$ che compone la traiettoria definita da $f(p)$.

Il tempo proprio totale $\Delta\tau$, cioè il tempo effettivamente impiegato dal fotone lungo la sua traiettoria, si ottiene dividendo la lunghezza totale percorsa per la velocità della luce c. Per trovare la lunghezza totale percorsa è necessario integrare le lunghezze dei tratti infinitesimali $\mathrm{d}\mathbf{X}$ definiti precedentemente. La lunghezza del tratto $\mathrm{d}\mathbf{X}$ nel S.R.I. del fotone è definita come:

$$L(\mathbf{X}, \mathrm{d}\mathbf{X}) = \sqrt{\mathrm{d}s^2} = \sqrt{\mathrm{d}\mathbf{X}^T g(\mathbf{X}) \mathrm{d}\mathbf{X}} \tag{1.8}$$

A questo punto possiamo definire il tempo proprio totale $\Delta\tau$, riassumendo le ultime considerazioni attraverso una singola equazione:

$$\Delta\tau = \frac{S}{c} = \frac{1}{c} \int_{p_A}^{p_B} L(f(p), f'(p)) \mathrm{d}p. \tag{1.9}$$

Come già detto in precedenza, il fotone percorre la traiettoria per cui questo integrale, ovvero il tempo proprio $\Delta\tau$, è minimo. Questo è il

secondo grande problema numerico che rende il modello esatto impraticabile.

Per ovviare a questi due grandi problemi numerici, cioè ricalcolare il tensore metrico in ogni punto e contemporaneamente trovare la funzione $f(p)$ che minimizza l'integrale 1.9, nelle simulazioni fisiche si utilizza un'approssimazione del fenomeno per la simulazione fisica dei fotoni. In particolare, nel progetto sviluppato in questa tesi è stata implementata quella che in letteratura viene chiamata "Approssimazione Flat-Sky".

1.5 Approssimazione Flat-Sky

Nell'approssimazione Flat-Sky si considera il problema della traiettoria di un fotone come un problema ottico-geometrico in cui si fanno le seguenti supposizioni fondamentali:

1. La distanza tra osservatore e corpo deflettore è talmente elevata per cui i fotoni che compongono l'immagine hanno tutti la stessa direzione (proiezione parallela).

2. Si assume che il corpo deflettore abbia simmetria radiale attorno alla retta che congiunge l'osservatore con il centro del corpo deflettore, per cui anche la deviazione della luce sarà simmetrica rispetto a questa retta.

3. Il raggio del corpo deflettore è molto minore rispetto alla distanza percorsa dal fotone per arrivare all'osservatore, quindi il tempo che il raggio di luce impiega per curvare è molto piccolo. Se la distanza percorsa tende a infinito, allora il tempo di curvatura tende a 0 e quindi il raggio viene deviato al limite in un solo punto.

4. Il punto in cui il raggio viene deflesso giace su un piano passante per il centro del corpo, per cui si modella la deviazione come causata da una lente sottile.

Con lente sottile si fa riferimento a una lente il cui spessore è molto minore del raggio di curvatura della lente lungo l'asse ottico, per cui si assume che la luce venga rifratta una sola volta e non due (una volta entrando nella lente e un'altra uscendo dalla lente). In questo caso il piano

passante per il centro del corpo è il caso limite della lente sottile associata, considerando raggio di curvatura infinito nell'asse ottico e spessore nullo.

Fatte tutte queste premesse, per spiegare dal punto di vista geometrico come il raggio viene deviato consideriamo la versione semplificata in due dimensioni, per poi estendere l'approssimazione al caso tridimensionale.

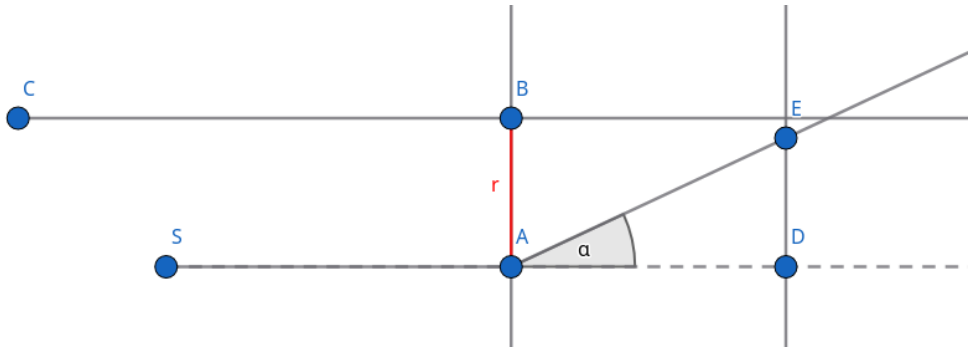


Figura 1.1: Diagramma bidimensionale dell'approssimazione Flat-Sky.

Facendo riferimento alla figura 1.1, sia C il punto che rappresenta l'osservatore e B il centro di massa del corpo deflettore. Costruiamo la retta BC e la sua perpendicolare in B per rappresentare rispettivamente l'asse ottico e la retta della lente sottile.

Definita la lente in questo modo, consideriamo un raggio di luce parallelo alla retta BC che ha origine in S e che colpisce la retta della lente in A. Il raggio di luce in quest'approssimazione viene deviato nel punto A di un angolo α, ottenendo il raggio AE. L'angolo di deviazione viene posto in modo tale che il raggio AE punti verso l'asse ottico e come già descritto nella sezione 1.1, α segue la seguente equazione:

$$\alpha = \frac{4GM}{c^2 r} \tag{1.10}$$

dove G è la costante di gravitazione universale ($G \approx 6.67 \cdot 10^{-11}\ m^3 \cdot kg^{-1} \cdot s^{-2}$), c è la velocità della luce nel vuoto ($c \approx 3.00 \cdot 10^8\ m \cdot s^{-1}$), M è la massa del corpo deflettore e r è la distanza tra il punto A di intersezione e il corpo deflettore B.

Da questa equazione emergono due proprietà fondamentali:

1. Il fattore di proporzionalità è estremamente piccolo, infatti $4G/(c^2) \approx 2.96 \cdot 10^{-27}$. Questo spiega perchè il fenomeno della deflessio-

ne della luce è osservabile soltanto in scala cosmica, coinvolgendo masse molto elevato come quelle delle galassie, e non nell'esperienza quotidiana.

2. La lente sottile che abbiamo definito sul corpo non ha un punto focale, perchè α è proporzionale a $1/r$ e quindi più ci si avvicina al corpo deflettore, maggiore sarà la deflessione della luce.

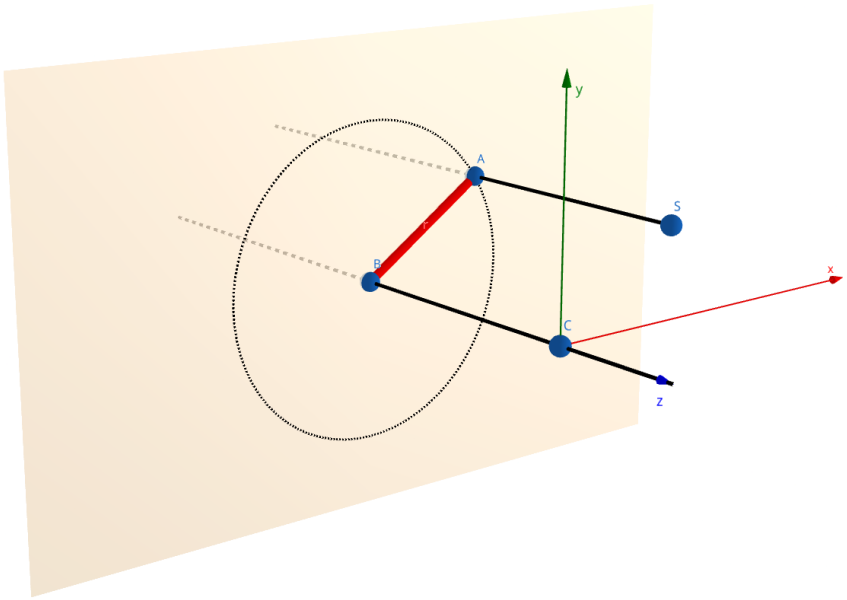
Ora estendiamo l'approssimazione Flat-Sky al caso tridimensionale. Per fare ciò utilizzeremo l'ipotesi fatta al punto 2, ovvero che il corpo deflettore possieda simmetria radiale rispetto all'asse ottico della lente.

Facendo riferimento alla figura 1.2a, sia C il punto che rappresenta l'osservatore e B il centro di massa del corpo deflettore. Costruiamo la retta BC e il piano p perpendicolare a BC passante per B, questi rappresentano rispettivamente l'asse ottico e il piano della lente sottile. Per semplificare i passaggi successivi, consideriamo un sistema di riferimento tale che:

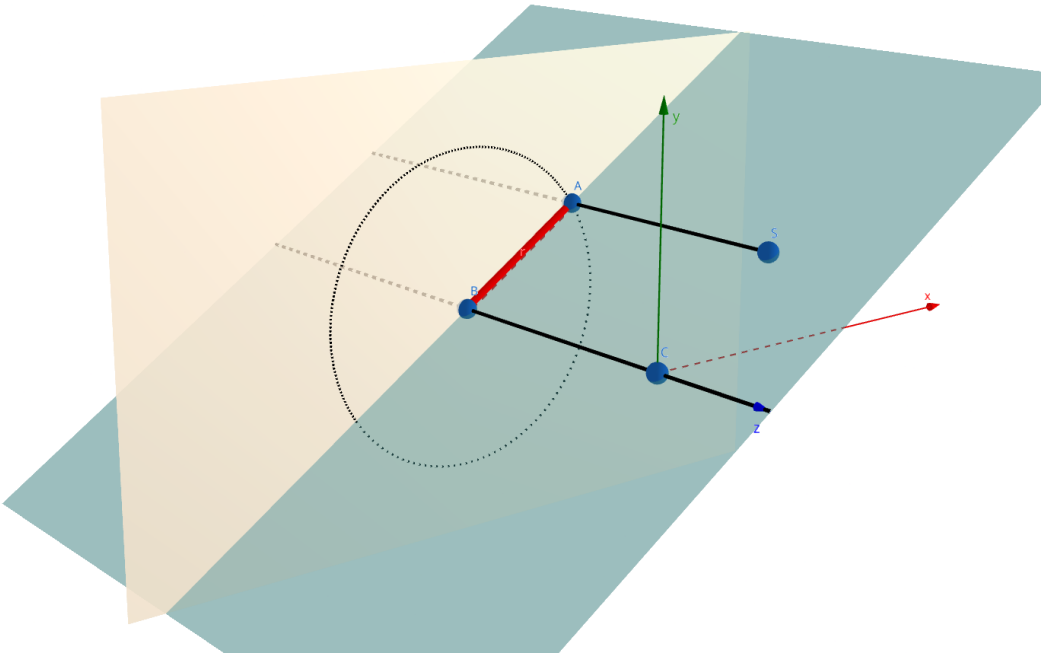
- C si trova nell'origine degli assi.
- La retta BC coincide con l'asse Z.
- Il punto B si trova nella metà negativa dell'asse Z, per cui $B = (0, 0, z_p)$ con $z_p < 0$.

Su questo sistema di riferimento locale il piano della lente in B si può descrivere attraverso l'equazione $z = z_p$. Ora consideriamo un raggio luminoso che ha origine in S con vettore direzione $\vec{v} = (0, 0, -1)$, questo intersecherà il piano p nel punto A, in modo analogo al caso bidimensionale.

Dato che abbiamo supposto per il punto 2 che il corpo deflettore ha simmetria radiale rispetto all'asse ottico (nel sistema di riferimento posto, equivalente all'asse Z), ciò significa che l'angolo di deviazione α dipende soltanto dalla distanza del punto A dal centro B, ovvero r (nella figura 1.2a r corrisponde al segmento rosso). Al contrario, la direzione in cui il raggio viene deviato corrisponde al vettore che punta da A in B, per cui costruiamo il piano q passante per A, B e C (figura 1.2b). Su questo piano la deviazione si applica esattamente come nell'esempio bidimensionale, per cui il raggio luminoso seguirà una retta che giace nel piano q tale che forma un angolo α rispetto alla retta SA secondo l'equazione 1.10.



(a) Costruzione iniziale del problema



(b) Aggiunta del piano q passante per A, B e C

Figura 1.2: Diagramma tridimensionale dell'approssimazione Flat-Sky

Capitolo 2

Teoria del Ray-Tracing e sua estensione alla simulazione gravitazionale

Con Ray-Tracing si indica la modellazione fisica della luce, che illumina una scena come un'insieme di elementi geometrici in reciproca interazione. Ogni fotone viene modellato attraverso un raggio, ovvero una semiretta orientata che ha origine in un punto, che quando interagisce con un'oggetto all'interno della scena può generare ulteriori raggi in varie direzioni, in base alle caratteristiche del materiale dell'oggetto stesso.

Nella maggior parte dei casi questa modellazione fisica ha come obiettivo la produzione di un'immagine, molto spesso fotorealistica, di un ambiente piccolo in cui sono presenti vari oggetti di materiali diversi. Questo contesto è molto diverso da quello analizzato con il progetto di questa tesi, sia per le grandezze assolute degli oggetti coinvolti sia per la natura stessa dei fotoni da simulare.

Per descrivere meglio queste differenze bisogna avere un punto di riferimento su cui basarsi, per cui è necessario descrivere i concetti fondamentali dell'ottica che stanno alla base del Ray-Tracer convenzionali, in particolare la descrizione della camera oscura e del suo funzionamento, per poi successivamente ampliare il discorso anche al contesto universale con lenti gravitazionali.

2.1 Concetti fondamentali

Uno degli obiettivi del Ray-Tracing è quello di simulare fisicamente il comportamento della luce il più fedelmente possibile alla realtà, per cui prima di spiegare il funzionamento base dei Ray-Tracer è necessario descrivere come viene prodotta un'immagine attraverso una fotocamera reale.

Il modello di fotocamera più semplice che fin dall'antichità è stato utilizzato è quello della camera oscura, detta anche fotocamera stenopeica o pinhole camera.

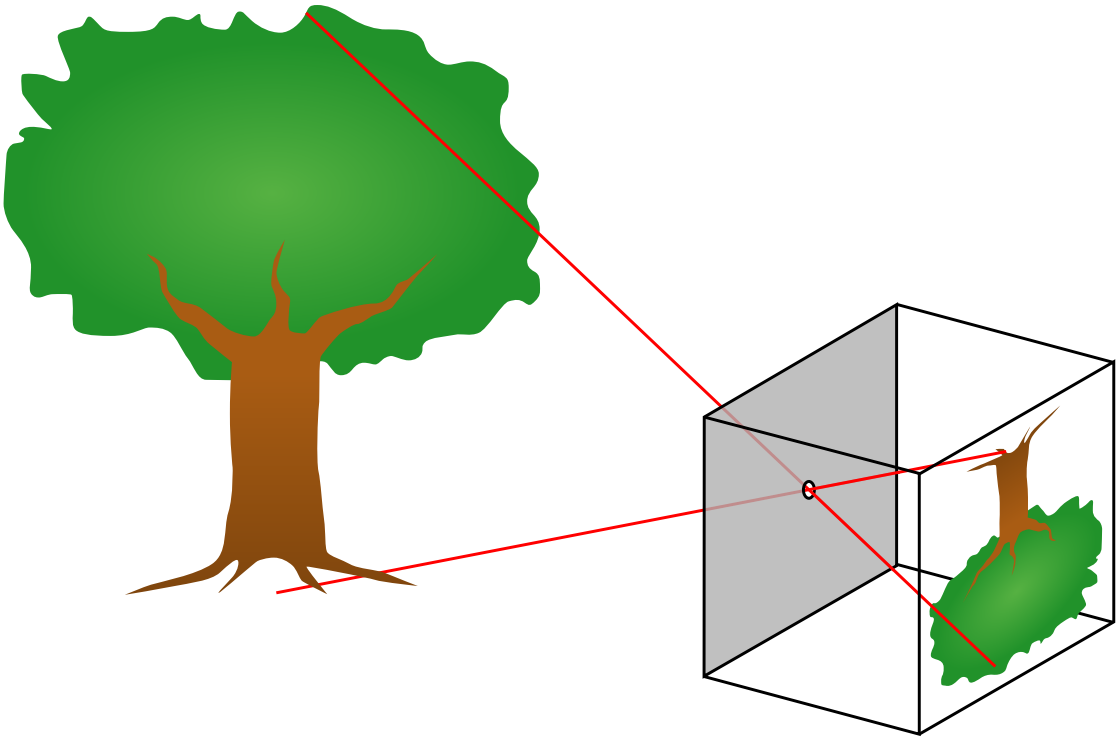
La fotocamera stenopeica è una scatola rivestita di materiale opaco alla luce. Su uno dei lati viene posta una pellicola fotosensibile, mentre sul lato opposto viene praticato un piccolo foro [0]. Una volta prodotto, il foro viene immediatamente ricoperto, in modo da poter orientare la camera verso il soggetto o ambiente da riprendere. Per formare l'immagine il foro viene riaperto per un periodo prolungato, consentendo così l'esposizione della pellicola fotosensibile.

La presenza del foro è fondamentale perchè in questo modo un qualsiasi punto della pellicola viene illuminato soltanto dal raggio di luce che percorre la linea retta che congiunge il punto della pellicola con il foro presente nel lato opposto, come descritto dalla figura 2.1a. Se il foro è troppo grande si rischia di ottenere un'immagine poco nitida perchè esistono più rette che riescono ad oltrepassare il foro e raggiungere la pellicola, per cui deve essere il più piccolo possibile. L'immagine che si ottiene nella pellicola è capovolta sia nell'asse verticale che nell'asse orizzontale.

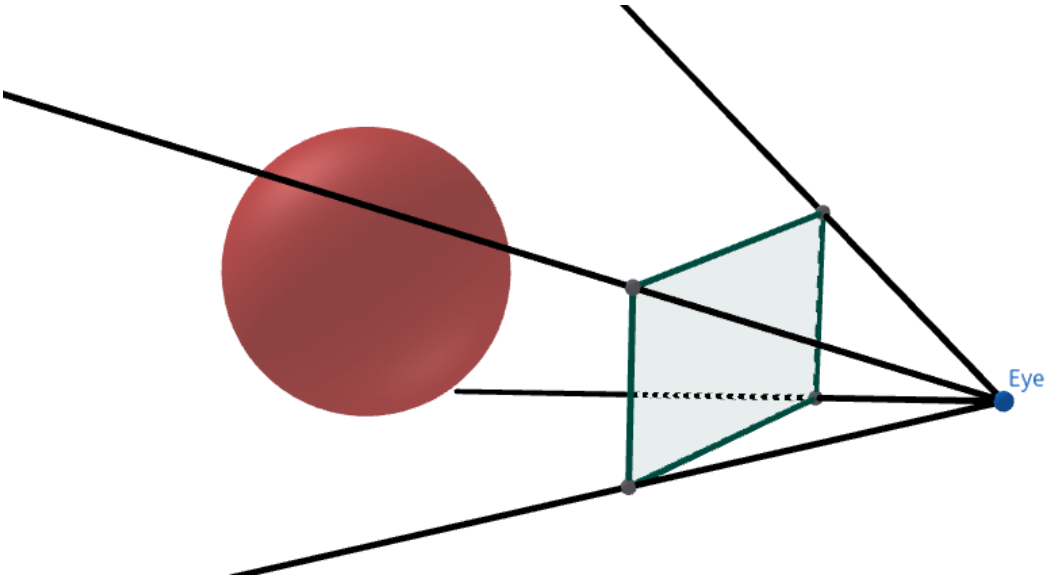
Nella fisica del Ray-Tracing questo modello viene utilizzato in una versione leggermente modificata come mostrato in figura 2.1b. In tale configurazione la pellicola viene posta di fronte al foro e per formare l'immagine si mantiene la proprietà fisica per cui si considerano soltanto i raggi che passano per il foro, che in contesti di Ray-Tracing è un punto singolo che viene chiamato *occhio* o *eye*. L'immagine che si va a formare, dato che la pellicola è posta di fronte, non capovolta.

Nella maggior parte dei Ray-Tracer la pellicola fotosensibile viene modellata attraverso un piano, detto piano immagine, che è perpendicolare alla direzione in cui la camera virtuale sta guardando la scena e si

[1] Attribuzione: en:User:DrBob (original); en:User:Pbroks13 (redraw), Public domain, via Wikimedia Commons



(a) Diagramma dell'immagine prodotta da una fotocamera stenopeica [1].



(b) Diagramma della fotocamera usata nel Ray-Tracing

Figura 2.1: Diagrammi di riferimento nel Ray-Tracing

considerano soltanto i raggi che provengono dal lato opposto in cui si trova l'occhio. La regione considerata per formare l'immagine diventa così una piramide infinita a cui viene tagliata la parte superiore, corrispondente alla porzione compresa tra piano immagine e occhio. Questa piramide molto spesso prende il nome di *frustum*.

Una volta individuata la regione di spazio di interesse, per formare l'immagine è necessario determinare quali fotoni attraversano l'occhio, poichè soltanto questi contribuiscono alla formazione dell'immagine. In base alla direzione in cui viene simulato il movimento del fotone, i Ray-Tracer si dividono in due macro-categorie:

- Ray-Tracer in avanti: Il fotone viene simulato partendo da una sorgente di luce, per poi interagire una o più volte con gli oggetti della scena ed infine raggiungere l'occhio passando attraverso il piano immagine.

- Ray-Tracer all'indietro: Partendo da un punto del piano immagine, si determina quale traiettoria deve percorrere un fotone per raggiungere tale punto, ricostruendo a ritroso le eventuali interazioni con gli oggetti della scena fino a risalire alla sorgente di luce.

La maggior parte dei Ray-Tracer, incluso quello sviluppato in questo progetto, ricade nella seconda categoria. Questo fatto sembra paradossale, perchè bisogna risalire dall'effetto alla causa, però permette di risparmiare una grande quantità di calcoli. Nei Ray-Tracer in avanti può capitare che un fotone venga simulato completamente, si calcolano le interazioni con gli oggetti della scena ma alla fine non passa attraverso l'occhio della camera, per cui il contributo di tale fotone all'immagine è completamente nullo e tutta la simulazione fatta è stata inutile. A questo si aggiunge il problema che per formare un'immagine in questo modo è necessario simulare una grande quantità di fotoni, dato che a priori non si può sapere se un fotone contribuirà all'immagine oppure no e in alcuni casi piccole variazioni di angolo iniziale determinano se un fotone raggiungerà il piano immagine oppure no.

Al contrario, se la simulazione dei fotoni viene eseguita al contrario, partendo da un punto nell'immagine, si considerano per costruzione soltanto i fotoni che alla fine contribuiranno all'immagine. In questo modo si diminuisce drasticamente il numero di calcoli e di interazioni necessarie per formare l'immagine perchè tutto ciò che si trova al di fuori del

frustum non viene considerato se non è strettamente necessario. Per questo motivo d'ora in poi in questo testo faremo sempre riferimento a questo secondo tipo di Ray-Tracer.

2.2 Ray-Tracer all'indietro

Quando si implementa un Ray-Tracer all'indietro la domanda fondamentale che ci si pone è: "so che un fotone è arrivato in un determinato punto seguendo una certa retta, da dove è partito questo fotone per arrivare in questo modo?" Formalizzando di più la domanda, chiamiamo con P e $\vec{d}$ rispettivamente il punto e la direzione presa in considerazione. Dato che dal punto di vista geometrico la traiettoria del fotone viene modellata attraverso una retta, se costruiamo il raggio con origine P in direzione $\vec{d}$ allora qualsiasi punto di tale semiretta può essere interpretato come un possibile punto di origine da cui il fotone è partito per arrivare in P.

Tra tutti questi punti assumiamo che il fotone non sia partito da un punto "in aria" ma che sia partito da un oggetto presente nella scena, per cui i punti di nostro interesse sono quelli in cui la semiretta interseca elementi della scena. Tra tutti questi punti d'intersezione quello che ci interessa è quello la cui distanza dal punto P risulta minima, perchè:

- Se l'oggetto è opaco allora tutti i fotoni che provengono da più lontano vengono bloccati dall'oggetto stesso.
- Se l'oggetto è trasparente o traslucido il fotone potrebbe essere passato attraverso l'oggetto. Dato che stiamo considerando la simulazione opposta a ciò che succede nella realtà, quello che ci interessa è il punto in cui il fotone esce dall'oggetto, che corrisponde all'intersezione più vicina.
- Se l'oggetto è una luce della scena allora possiamo calcolare direttamente quanta luce viene trasmessa dal fotone.

Una volta individuato il punto d'intersezione più vicino, si considera l'oggetto della scena associato e, in base alle caratteristiche del materiale, si generano uno o più raggi per calcolare la quantità di luce che arriva nel punto d'intersezione. Questo procedimento è di natura ricorsiva: per determinare la quantità di luce incidente in un punto occorre

tener conto di tutti i fotoni che possono contribuirvi. Poichè ciascun fotone può provenire da un altro punto, potenzialmente illuminato in modo indiretto, è necessario analizzare quest'ultimo allo stesso modo e così via procedendo a ritroso fino a quando non si arriva a una sorgente di luce.

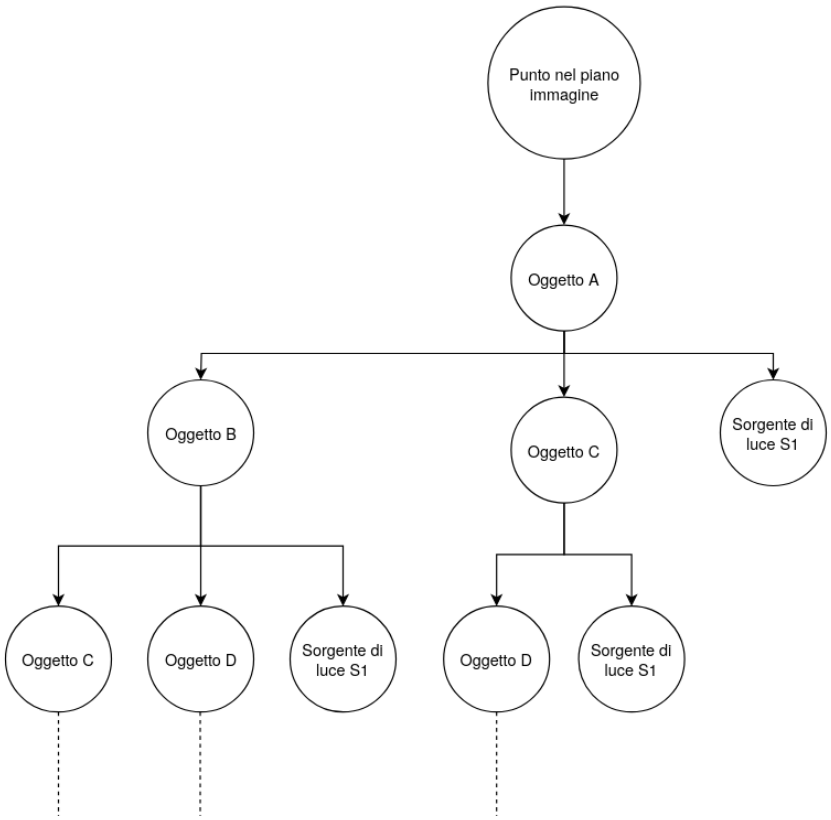


Figura 2.2: Esempio di diagramma ad albero delle dipendenze luminose tra i raggi

Attraverso un diagramma ad albero è possibile descrivere la gerarchia dei raggi considerati dal Ray-Tracer, come per esempio in figura 2.2. In questo caso i nodi del grafo rappresentano gli oggetti della scena e gli archi rappresentano i fotoni che vengono generati quando si analizza la luminosità in un punto. Nel diagramma ogni arco dal nodo A al nodo B è da interpretare come "il punto nell'oggetto A è illuminato da un fotone proveniente dall'oggetto B", per cui nel diagramma il verso dell'arco è opposto a quello percorso effettivamente dal fotone. Questa inversione della direzione è causata dalla natura intrinseca del Ray-Tracer

all'indietro, che simula i fotoni nella direzione opposta. Inoltre nel diagramma i nodi corrispondenti a fonti di luce sono nodi speciali, perchè per definizione sono nodi foglia dell'albero, dato che non generano altri fotoni.

Dal punto di vista computazionale la simulazione ricorsiva dei fotoni viene terminata quando la profondità dell'albero raggiunge una soglia limite oppure quando il contributo luminoso di un fotone è talmente piccolo da essere trascurabile nell'immagine finale.

2.3 Estensione geometrica alla simulazione di lenti gravitazionali

Nella sezione 2.1 la piramide di vista è stata costruita come modifica della geometria presente nella fotocamera stenopeica. Le immagini che vengono prodotte da questo tipo di fotocamera sono immagini che mantengono la prospettiva degli oggetti. Tuttavia, per gli obiettivi di questo progetto tale rappresentazione non è corretta, poichè come descritto al punto 1 della sezione 1.5, l'immagine deve essere generata secondo una proiezione parallela.

Ciò che definisce una proiezione parallela è che i fotoni che arrivano al piano dell'immagine sono tutti paralleli tra loro, invece di avere angoli diversi che puntano verso l'occhio.

Per cambiare il tipo di proiezione si possono applicare due metodi diversi: il primo è quello di aumentare la distanza tra occhio e piano dell'immagine; all'aumentare di tale distanza la differenza angolare tra i raggi diminuisce progressivamente, tendendo a un fascio di raggi paralleli quando la distanza tende a infinito. Il secondo modo è quello di modificare la definizione del raggio iniziale nel Ray-Tracer: invece di considerare la retta passante per l'occhio e per un punto del piano immagine, si utilizza direttamente la retta passante per quel punto e parallela a una direzione di riferimento. In questo modo si garantisce per costruzione che le rette siano tutte parallele tra loro e si evita di dover considerare l'occhio, che nelle proiezioni parallele perde di significato.

Dal punto di vista geometrico il volume che definisce la porzione di scena visibile attraverso il piano d'immagine passa da una piramide tronca a un parallelepipedo con profondità infinita, come visualizzato in figura 2.3.

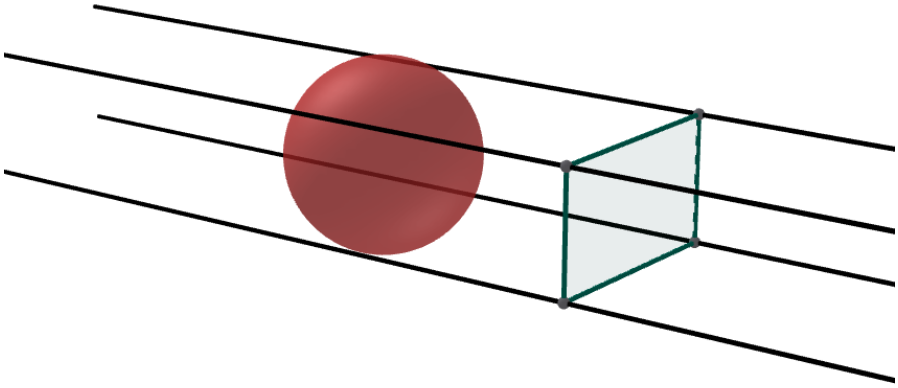


Figura 2.3: Camera con proiezione ortogonale nel Ray-Tracing

Il secondo fattore da considerare nella realizzazione di un Ray-Tracer per la simulazione di lenti gravitazionali è un'assunzione che abbiamo posto per aiutare nella spiegazione teorica del Ray-Tracing che adesso deve essere rivista. Abbiamo assunto che i fotoni all'interno di una scena viaggino secondo una retta, per cui non può essere partito "in aria". In un contesto gravitazionale universale questa assunzione non è più vera, o meglio deve essere ricontestualizzata: dal punto di vista del fotone stesso la traiettoria è rettilinea, ma a causa della curvatura dello spazio-tempo quello che percepiamo è che il fotone segue una traiettoria curvilinea. Di conseguenza, non è più possibile considerare esclusivamente i punti d'intersezione tra il raggio e gli oggetti nella scena, poichè questa semplificazione è corretta esclusivamente nel caso di uno spazio perfettamente piatto, in cui il fotone segue la retta definita dal raggio.

A questo punto ci arriva in soccorso l'approssimazione Flat-Sky che abbiamo preso in considerazione, perchè racchiude tutta la curvatura del fotone in un solo punto appartenente a un piano infinito. In questo modo possiamo ritornare alle assunzioni di prima sulla rettilinearità dei fotoni e aggiungere le intersezioni tra raggio e questi piani infiniti per trovare il punto più vicino.

Nel caso in cui il punto più vicino appartiene a un corpo come un pianeta o una galassia l'illuminazione viene ricalcolata secondo le tecniche standard di Ray-Tracing, mentre se il punto più vicino appartiene a questi piani infiniti allora si ricava il raggio deflesso secondo la matematica descritta nella sezione 1.5.

Capitolo 3

Analisi del progetto

In questo capitolo si inizia a descrivere in dettaglio GraviLenSim, partendo dall'analisi dei simulatori fisici esistenti per individuare le limitazioni presenti, così da poter descrivere gli obiettivi che sono stati posti per lo sviluppo del progetto.

Nella seconda parte del capitolo verrà descritta l'architettura effettiva del progetto, indicando le componenti principali e lo stile di programmazione utilizzato per la codifica.

3.1 Limitazioni osservate ed obiettivi

Nei progetti esistenti che cercano di simulare gli effetti delle lenti gravitazionali sono state trovate alcune limitazioni:

- In alcuni la simulazione viene fatta soltanto in due dimensioni, utilizzando un diagramma "dall'alto" che mostra l'intera traiettoria dei fotoni.
- In altri si ha un'universo statico in background e attraverso l'utilizzo del mouse si sposta un corpo massiccio che deflette la luce.
- Solo in pochi viene data la possibilità di modificare l'universo simulato, spesso utilizzando slider con range completamente arbitrari.

Per questi motivi sono stati scelti i seguenti obiettivi per realizzare questo progetto:

1. La simulazione deve essere tridimensionale.

2. Deve essere data all'utente la possibilità di modificare l'universo nel modo più diretto e più esteso possibile.

3. La simulazione deve essere la più accurata possibile dal punto di vista fisico.

4. La simulazione deve usare internamente delle unità fisiche del SI, così da poter riusare dati misurati esternamente ed avere una simulazione coerente.

5. La performance deve essere sufficientemente veloce da non bloccare l'utente per lunghi periodi di tempo.

6. L'output finale del progetto deve essere un'immagine dal punto di vista di un satellite che analizza una determinata porzione dell'universo.

3.2 Descrizione del progetto

Il progetto realizzato si chiama GraviLenSim ed è un editor GUI in cui l'utente può definire l'intero universo di simulazione, indicando tutti i corpi massicci presenti con i loro dati, per poi eseguire una simulazione fisica dei fotoni per renderizzare l'immagine che una possibile satellite vedrebbe di tale universo.

Tra le feature secodarie di questo progetto c'è la possibilità di renderizzare l'immagine e memorizzarla direttamente sul filesystem, senza l'apertura dell'editor. Infatti l'editor è separato dal sistema che si occupa della simulazione fisica e del rendering, per cui eseguendo dal terminale con opportuni parametri si può renderizzare in questa modalità "one-shot".

Un'altra feature secondaria consiste nella possibilità di sostituire il modello di simulazione fisica utilizzata a runtime, senza la necessità di ricompilare il progetto, poichè tale simulazione è implementata in GLSL.

3.3 Architettura software

In questa sezione si vuole descrivere l'architettura software della porzione di progetto in esecuzione sulla CPU, così da poter dedicare l'intero capitolo 4 per entrare in dettaglio sulla programmazione GPU.

La scelta di usare OpenGL ha causato a cascata la scelta di usare C++ come linguaggio di programmazione per implementare il progetto, l'editor e la logica di ponte con la GPU. Questo perchè le funzioni OpenGL sono pensate per essere utilizzate in primis con C/C++.

Dal punto di vista stilistico il codice è stato scritto cercando di utilizzare il meno possibile feature considerabili "avanzate" nel linguaggio C++ e di scrivere codice che assomigli il più possibile a C. Allo stesso momento non si è collassato il codice completamente al linguaggio C perchè nella Standard Template Library di C++ sono presenti molte librerie e classi, in particolare i container standard come vettori e mappe, che sono veramente molto utili, quasi fondamentali.

Il compromesso stilistico è stato delineato attraverso i seguenti punti:

- Le strutture sono state utilizzate per descrivere oggetti che contengono puramente dati.
- Le classi sono state utilizzate prevalentemente come metodo di suddivisione logica del codice. Contengono al loro interno sia dati che funzioni, ma non sono pensate per essere istanziate più di una volta.
- Per rendere esplicito il "Single Source of Truth", sono utilizzati puntatori (al posto dei riferimenti) per indicare le situazioni quando una funzione/classe fa riferimento a un dato che logicamente non possiede.
- L'utilizzo della Standard Template Library è consentito.
- Operazioni definite attraverso uso di operator overload sono ridotte al minimo.

Concettualmente il progetto è suddiviso in due modalità: Editor e Renderer. Nella funzione `main()` è definita la logica per interpretare i parametri passati attraverso riga di comando, per decidere quale delle due modalità deve essere eseguita. Per ogni modalità viene definita

una funzione `main***()` per rappresentare l'entry point specifico per tale modalità. Per Editor e Renderer tali funzioni sono rispettivamente `mainUI()` e `mainRenderer()`.

Il resto del codice del progetto è suddiviso principalmente nelle seguenti componenti:

- `Raytracer`: è il ponte principale con la GPU, si occupa di preparare il contesto d'esecuzione della simulazione fisica, farla partire e ottenerne l'immagine risultante.
- `Universe`: struttura che definisce l'universo di simulazione.
- `Serialization`: coppia di funzioni `serializeAll()` / `deserializeAll()` utilizzate per il salvataggio e caricamento delle informazioni dell'universo da/a file.
- `Viewport` (solo Editor): si occupa di convertire l'immagine tra formati in modo tale da poter essere visualizzata nella finestra dell'editor, applicando se necessario algoritmi di scaling.
- `UI` (solo Editor): presente solo nella modalità Editor, racchiude tutta la struttura dell'interfaccia utente e il codice che modifica l'istanza del tipo Universe corrente.
- `Flag` (solo Editor): presente solo nella modalità Editor, vengono usati come segnali da UI al resto del codice per indicare azioni che dovrebbero essere eseguite, ma che si trovano al di fuori del controllo diretto di UI.

Capitolo 4

Implementazione del Ray-Tracer

Per la realizzazione di questo progetto si è scelto di utilizzare OpenGL 4.3, sfruttando i Compute Shader scritti in GLSL per implementare la simulazione fisica e il Ray-Tracer. L'utilizzo dei Compute Shader è stato adottato poichè la simulazione risulta intrinsecamente parallelizzabile: nel contesto delle lenti gravitazionali, infatti, non esistono interazioni tra i fotoni, rendendo possibile l'elaborazione indipendende di ciascuno di essi.

Esistono varie tipologie di parallelizzazione, come per esempio i sistemi multi-threaded, ma considerando che il risultato finale è un'immagine renderizzata, decidere di delegare il lavoro alla GPU è stato un pensiero immediato. Tra le tecnologie GPU presenti OpenGL è stato scelto per la sua portabilità (OpenGL 4.3 è stato rilasciato nel 2012[0] per cui tutte le GPU recenti la supportano) e per la sua semplicità di utilizzo (confrontato con Vulkan, che richiede molto più codice di gestione).

4.1 Compute Shader

Con Shader si indica un programma eseguito su una GPU, molto spesso con l'obiettivo di definire il colore (*shade*) di un pixel all'interno di una griglia rettangolare.

Le due tipologie più famose di Shader sono le Vertex Shader e le Fragment Shader, che vengono utilizzate per visualizzare una scena at-

traverso una pipeline standard molto efficiente che non utilizza tecniche di Ray-Tracing. Al contrario le Compute Shader sono state progettate per velocizzare operazioni parallelizzabili che richiederebbero troppo tempo se eseguite su CPU. Per questo motivo non sono comprese nella pipeline di disegno di OpenGL, perchè potrebbero essere utilizzate per contesti non grafici, come per esempio il calcolo di valori hash per validare un blocco di transazioni Bitcoin.

Questa grande flessibilità sulla tipologia di operazioni che si possono eseguire ha un grande problema: mentre gli altri tipi di Shader hanno input e output ben definiti, perchè fanno parte di una pipeline di calcolo, nel caso delle Compute Shader tali elementi devono essere esplicitamente definiti dal programmatore, insieme alle modalità di esecuzione [0].

Per definire l'insieme di input/output all'interno della Compute Shader si possono usare operazioni di lettura/scrittura su immagine (Image load/store operations) per accedere ai dati contenuti in una texture oppure utilizzare Shader Storage Buffer Objects, ovvero buffer in memoria che permettono di passare dati strutturati da CPU a GPU e viceversa.

Quando si utilizzano Compute Shader per eseguire un'operazione in parallelo sulla GPU è necessario tenere a mente che la Shader deve essere scritta dal punto di vista della singola invocazione. Facendo un'esempio contrario, quando si vuole parallelizzare un'operazione sulla CPU nella maggior parte dei casi si scrive un programma unico che nella sua esecuzione si suddivide in più thread di calcolo e alla fine vengono riuniti tutti alla fine dell'esecuzione. Nel caso della parallelizzazione su GPU le operazioni di creazione e distruzione dei thread di esecuzione sono gestiti al di fuori della Compute Shader, che in questa analogia rappresenta esclusivamente i calcoli all'interno del thread.

Per definire quante invocazioni della Compute Shader devono essere fatte, OpenGL utilizza due spazi tridimensionali concettualmente annidati. Al livello più esterno abbiamo lo *spazio globale di calcolo*, dove il programmatore indica quanti gruppi di calcolo devono essere eseguiti specificando le dimensioni intere di un parallelepipedo. All'interno di questo parallelepipedo i gruppi di lavoro sono associati a tutti i punti che hanno coordinate intere con gli estremi massimi esclusi, per cui se il parallelepipedo ha come dimensione $(1, 2, 3)$, allora i gruppi di lavoro avranno coordinate $(0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 1, 0), (0, 1, 1), (0, 1, 2)$.

Per ogni gruppo di lavoro viene definito un altro parallelepipedo annidato, chiamato *spazio locale di calcolo*, le cui dimensioni sono speci-

ficate dalla Compute Shader e i punti che hanno coordinate intere al suo interno rappresentano le effettive esecuzioni della Compute Shader.

In questo modo la Compute Shader viene identificata attraverso due vettori tridimensionali, uno per rappresentare la coordinata del gruppo di lavoro e l'altra per indicare la posizione locale all'interno del gruppo. Queste sono passate alla Compute Shader come variabili globali, rispettivamente `gl_WorkGroupID` e `gl_LocalInvocationID`. In figura 4.1 è presente una visualizzazione grafica di un esempio di dispatch.

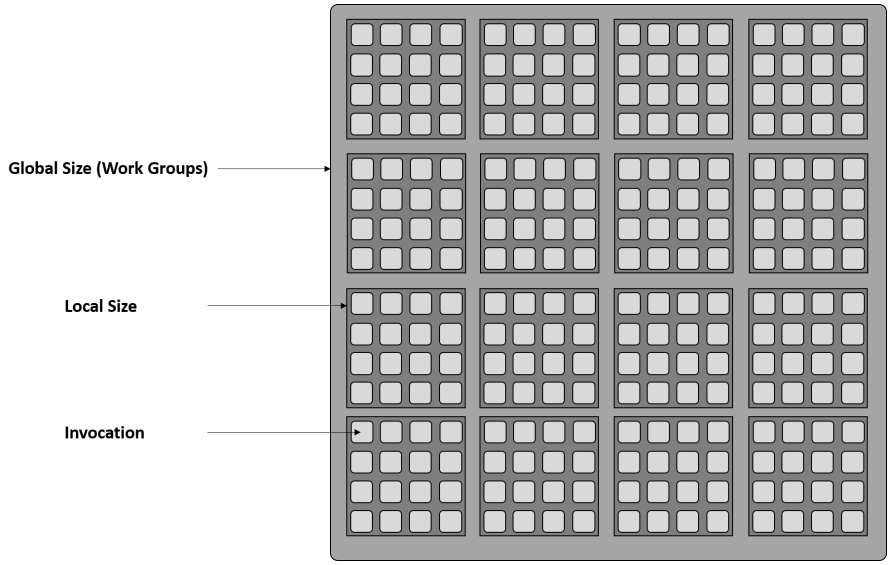


Figura 4.1: Esempio di diagramma di esecuzione per una Compute Shader, in questo caso un dispatch di uno spazio globale $(4, 4, 1)$ con dimensione locale $(4, 4, 1)$ per un totale di $4 \cdot 4 \cdot 1 \cdot 4 \cdot 4 \cdot 1 = 256$ invocazioni [1].

4.2 Gestione delle compute shader nel progetto

In OpenGL le Compute Shader sono definite dallo standard come programmi, esattamente allo stesso modo dei programmi che includono Vertex e Fragment Shader per il rendering classico. Per questo motivo la

[1] Attribuzione: Jonas Sorgenfrei, `https://learnopengl.com/Guest-Articles/2022/Compute-Shaders/Introduction`

loro creazione utilizza le stesse funzioni, cambiando semplicemente un parametro. Nel progetto questa operazione è astratta da delle funzioni di utility.

```
//raytracer.cpp:137
GLuint program = linkProgram({ compileShader(filename, GL_COMPUTE_SHADER) });

//...is equivalent to the following code...

//Read filename to string
std::ifstream shaderFile = std::ifstream(filename);
if(!shaderFile.is_open()){
    return 0; //failed to read the file
}
std::string shaderSource = "";
std::string temp;
while(std::getline(shaderFile, temp)){
    shaderSource += temp + "\n";
}
//Create Compute Shader itself
GLuint computeShader = glCreateShader(GL_COMPUTE_SHADER);
const char* shaderSourceRaw = shaderSource.c_str();
glShaderSource(computeShader, 1, &shaderSourceRaw, NULL);
glCompileShader(computeShader);
//Link to a program
GLuint program = glCreateProgram();
glAttachShader(program, computeShader);
glLinkProgram(program);
```

La Compute Shader che viene utilizzata per la simulazione fisica e per il Ray-Tracer ha i seguenti input e output principali:

```
//shaders/compute.shader:3
struct Planet {
    vec3 position;
    vec3 north;
    vec3 zeroDegree;
    float radius;
    float mass;
    vec3 ambient;
    vec3 diffuse;
    vec3 emission;
    float luminosity;
};

//shaders/compute.shader:39
//Input
layout(local_size_x = 1, local_size_y = 1) in;
layout(std430, binding = 2) readonly buffer transmissionBuffer {
    Planet data[];
};
//Output
layout(rgba32f, binding = 0) writeonly restrict uniform image2D texOutput;
layout(rgba32f, binding = 1) writeonly restrict uniform image2D debugOutput;
```

andando nel dettaglio:

- `layout(local_size_x=1,local_size_y=1) in;` definisce la dimensione dello spazio locale di esecuzione, in questo caso impostando $(1, 1, 1)$ per semplicità.

- `buffer transmissionBuffer` è un Shader Storage Buffer Object che contiene i dati degli oggetti all'interno dell'universo. All'interno di questo buffer sono contenuti soltanto elementi di tipo `Planet`, come descritto dal blocco `{ Planet data[]; }`. `layout(std430)` viene utilizzato per indicare come il singolo elemento `Planet` deve essere disposto in memoria, dettaglio che verrà spiegato più avanti.

- `texOutput` e `debugOutput` sono due oggetti `uniform image2D` che vengono utilizzati rispettivamente per formare l'immagine finale e per contenere dati aggiuntivi per ogni pixel (utilizzato prevalentemente in fase di sviluppo appunto per debuggare).

4.2.1 Shader Storage Buffer Object

La creazione di uno Shader Storage Buffer Object è molto semplice, richiedendo solo due chiamate a funzioni OpenGL:

```
1 //raytracer.h:22
2 GLuint transmissionBuffer;
3
4 glUseProgram(program);
5 //raytracer.cpp:34
6 glGenBuffers(1, &transmissionBuffer);
7 glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 2, transmissionBuffer);
```

Una nota molto importante è il `2` come secondo parametro di `glBindBufferBase`: tale numero magico deve coincidere con il valore di binding definito all'interno della Compute Shader. Dato che nella compute shader il buffer è definito con `layout(..., binding = 2)` (vedere listino precedente), allora `2` indica la base numerica del buffer.

Per quanto riguarda il caricamento dei dati all'interno del buffer, la situazione è leggermente più complicata a causa delle differenze tra C++ e GLSL sulle struct: Come già detto in precedenza `transmissionBuffer` è stato definito indicando `layout(std430)` come layout di memoria. Nella pratica questo layout segue le stesse regole di allineamento e di padding del linguaggio C, *tranne* per quanto riguarda oggetti di tipo `vec3`, perchè questi occupano in realtà lo stesso spazio di un oggetto `vec4` [0].

Per risolvere questa differenza è stata creata una struct dedicata, chiamata `PlanetGLSL`, il cui layout in memoria corrisponde 1:1 con quello che la Compute Shader si aspetta di trovare nel `transmissionBuffer`:

```
//planet.h:22
struct PlanetGLSL {
    alignas(16) glm::vec3 position; //x,y,z, in meters
    alignas(16) glm::vec3 northVector; //normalized
    alignas(16) glm::vec3 zeroDegree; //normalized
    float radius; //in meters
    float mass; //in kilograms
    alignas(16) glm::vec3 ambient; //r,g,b
    alignas(16) glm::vec3 diffuse; //r,g,b
    alignas(16) glm::vec3 emission; //r,g,b
    float luminosity; //watts
};
```

A questo punto il caricamento effettivo del buffer sulla GPU non richiede tante linee di codice:

```
//raytracer.cpp:64
std::vector<PlanetGLSL> converted = planetsToGLSL(&(universe->planets));
glUseProgram(program);
glBindBuffer(GL_SHADER_STORAGE_BUFFER, transmissionBuffer);
glBufferData(GL_SHADER_STORAGE_BUFFER, converted.size() * sizeof(PlanetGLSL), converted
    .data(), GL_STATIC_READ);
```

4.2.2 Image2D

In OpenGL `Image2D` rappresenta un oggetto che permette la lettura e la scrittura arbitraria su una Texture in memoria, per cui la creazione di un oggetto di questo tipo segue la stessa esatta procedura di una qualsiasi Texture. Negli esempi di codice successivi verrà mostrato soltanto il codice per la gestione di `texOutput`, dato che `debugOutput` richiede la stessa gestione ma con piccole differenze.

```
//raytracer.h:20
GLuint textureOutput;

//raytracer.cpp:18
glGenTextures(1, &textureOutput);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, textureOutput);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

Ciò che distingue `Image2D` da una qualsiasi texture è la necessità di informare OpenGL (e quindi la GPU) della possibilità di scrivere nel buffer di memoria che contiene la texture:

```
//raytracer.cpp:45
glBindTexture(GL_TEXTURE_2D, textureOutput);
//textureWidth and textureHeight refer to how big the final image should be, in pixels
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F, textureWidth, textureHeight, 0, GL_RGBA,
    GL_FLOAT, nullptr);
glBindImageTexture(0, textureOutput, 0, GL_FALSE, 0, GL_WRITE_ONLY, GL_RGBA32F);
```

`glBindImageTexture` è la funzione che permette di fare il collegamento tra un oggetto `Image2D` e la texture in memoria. Ogni GPU ha un determinato numero di slot dedicati per la lettura e scrittura da texture, che sono distinti dalle unità texture che invece contengono il buffer effettivo.

Il primo parametro indica il numero di slot immagine che deve essere utilizzato, che nel caso di `texOutput` deve corrispondere al valore definito nella Compute Shader attraverso `layout(..., binding = 0)`. Il secondo parametro contiene invece l'id della texture da associare (in questo caso `textureOutput`)

L'ultimo passo rimanente per preparare le immagini di output è di allocare la memoria necessaria. Nella chiamata a funzione di `glTexImage2D` viene passato come ultimo parametro `nullptr`. Questo è un caso speciale, definito dallo standard [0], in cui viene allocata memoria nella gpu per accomodare una texture di dimensioni `textureWidth` x `textureHeight`, senza nessun tentativo di caricare dati dalla CPU. L'unica nota da tenere a mente è che la memoria non viene inizializzata se si chiama `glTexImage2D` in questo modo.

4.2.3 Dispatch

Arrivati a questo punto abbiamo preparato tutto ciò che è necessario per far funzionare la Compute Shader, tra input e output, per cui non ci resta altro che farla eseguire:

```
1 //raytracer.cpp:129
2 glUseProgram(program);
3 glDispatchCompute(dispatchSize.x, dispatchSize.y, 1);
```

I parametri passati a `glDispatchCompute` sono le dimensioni dello spazio di esecuzione globale, come descritto nella sezione 4.1. Le dimensioni passate per x e y sono rispettivamente la larghezza e l'altezza dell'immagine da visualizzare. Queste dimensioni non vengono passate direttamente, ma vengono limitate superiormente dalle capacità hardware della GPU, in modo da garantire l'esecuzione della Compute Shader. Nelle GPU in cui il progetto è stato testato questi limiti sono talmente alti da non causare problemi ($2^{31} - 1$ in x e y), ma sono comunque dei limiti da considerare.

Per ottenere le dimensioni massime dello spazio globale di esecuzione si usa la funzione `glGetIntegeri_v` nel seguente modo:

```
int maxWidth;
int maxHeight;
int maxDepth;
glGetIntegeri_v(GL_MAX_COMPUTE_WORK_GROUP_COUNT, 0, &maxWidth);
glGetIntegeri_v(GL_MAX_COMPUTE_WORK_GROUP_COUNT, 1, &maxHeight);
glGetIntegeri_v(GL_MAX_COMPUTE_WORK_GROUP_COUNT, 2, &maxDepth);
```

4.3 Algoritmo di Ray-Tracing

Prima di descrivere l'algoritmo di Ray Tracing utilizzato in questo progetto, è necessario definire le strutture utilizzate per la simulazione fisica.

4.3.1 Definizioni delle strutture base

Definiamo di seguito due strutture, `Ray` e `Plane`, assieme alle loro funzioni geometriche di utilità.

```
//shaders/compute.shader:15
struct Ray {
    vec3 position; //coordinates in meters
    vec3 direction; //unit vector
};

struct Plane {
    vec3 origin; //coordinates in meters
    vec3 normal; //unit vector
};

//shaders/compute.shader:74
//Returns the parameter value t of the intersection point between ray and plane
float rayPlaneIntersection(Ray ray, Plane plane){
    vec3 k = ray.position - plane.origin;
    return -dot(k, plane.normal) / dot(ray.direction, plane.normal);
}

//Returns the point on the ray corresponding to the parameter value t
vec3 rayPoint(Ray ray, float t){
    return ray.position + t * ray.direction;
}
```

Nella spiegazione geometrica abbiamo trattato la deviazione considerando soltanto il centro di massa del corpo deflettore, ignorando l'estensione spaziale che questo ha nell'universo, perchè questa non influenza la deviazione impartita sui fotoni. Al contrario, per poter generare un'immagine corretta è necessario considerare l'estensione che il corpo deflettore ha nello spazio, perchè quest'ultimo si assume essere opaco e quindi non permette che un fotone passi attraverso.

La struttura `Planet`, già mostrata in precedenza nella sezione 4.2, rapprsenta un corpo deflettore perfettamente sferico avente densità uni-

forme su tutto il volume, per cui il centro geometrico e il centro di massa coincidono nello stesso punto (`Planet.position`).

Arrivati a questo punto ci manca un solo ultimo passo preliminare, ovvero definire la funzione che si occuperà di calcolare il raggio deviato:

```
//Returns a new ray, starting from closestPoint, bent according to the planet p.
//Usage example
/*
    Ray r = Ray(...);
    Planet p = Planet(...);
    Plane planetLens = Plane(p.position, ...);
    float intersectionT = rayPlaneIntersection(r, planetLens);
    vec3 intersectionPoint = rayPoint(p, intersectionT);
    Ray bent = bendRay(r, p, intersectionPoint);
*/
Ray bendRay(Ray original, Planet p, vec3 closestPoint){
    float impactRadius = length(closestPoint - p.position);
    const float C = 299792458.0;
    const float G = 6.67430e-11;
    float alpha = 4 * G * p.mass / (C * C * impactRadius);
    float deviation = length(original.direction) * tan(alpha);
    vec3 towardsCenter = normalize(p.position - closestPoint);
    vec3 newDir = normalize(original.direction + towardsCenter * deviation);
    return Ray(closestPoint, newDir);
}
```

La funzione `bendRay` restituisce il raggio deviato dalla lente calcolando prima l'angolo di deviazione `alpha`, poi il vettore direzione finale a partire da `towardsCenter`.

4.3.2 Ray-Tracer implementato in GLSL

Alla base del Ray-Tracer è presente una funzione, `castRay`, il cui obiettivo principale è simulare il singolo fotone e restituire le informazioni sulla prima collisione avvenuta con un corpo deflettore.

```
//OUTPUT
//Defines the possible values hitType can be. Treat this like an enum
//shaders/compute.shader:26
const uint HIT_BACKGROUND = 0;
const uint HIT_PLANET = 1;
const uint HIT_LENS = 2;

struct RayHit {
    Ray ray; //Last ray that is calculated by the algorithm, which intersected
        something
    float t; //Parameter t of where the intersection occurred
    float distanceTraveled; //in meters
    uint hitType; //See HIT_* values above
    int deflections;
    int planetIndex; //if hitType == HIT_PLANET, contains the index of the hit planet,
        else it contains -1
};

RayHit castRay(Ray initial, float furthestT);
```

Per la simulazione fisica la funzione precedente utilizza, in aggiunta ai parametri passati, anche `Planet data[]` contenuto nello Shader Storage Buffer Object descritto nella sezione 4.2.

`castRay` restituisce un oggetto di tipo `RayHit`, che contiene al suo interno i dati necessari per determinare il colore finale del pixel associato al raggio. Questo perchè la funzione simula al suo interno tutte le deviazioni causate dai piani delle lenti gravitazionali fino a quando il raggio non interseca un pianeta oppure il piano utilizzato per disegnare lo sfondo.

Arrivati a questo punto, iniziamo a definire la funzione `castRay`.

```
RayHit castRay(Ray initial, float furthestT){
    RayHit result = RayHit(initial, furthestT, 0, HIT_BACKGROUND, 0, -1);

    int deflections = 0;
    const int MAX = 10;
    for(int j = 0; j < MAX; j++){
        float minT = furthestT;
        uint intersectionType = HIT_BACKGROUND;
        int closestPlanetIndex = -1;

        /*
            Calculate closest intersection between planets.
            This process will also change the variables above
        */

        vec3 p = rayPoint(result.ray, minT);
        result.distanceTraveled += length(p - result.ray.position);
        result.t = minT;
        if(intersectionType == HIT_PLANET){
            //We hit a planet, so it doesn't make sense to go on
            result.hitType = HIT_PLANET;
            result.planetIndex = closestPlanetIndex;
            break;
        } else if (intersectionType == HIT_BACKGROUND){
            //We didn't any lens or planet, so we terminate early
            result.hitType = HIT_BACKGROUND;
            break;
        } else {
            //We hit a lens, so we should bend the ray and keep going
            result.ray = bendRay(result.ray, data[closestPlanetIndex], p);
            result.hitType = HIT_LENS;
            result.deflections++;
        }
    }
    return result;
}
```

Per la logica di intersezione si segue il seguente processo:

1. Per ogni pianeta p_i nella lista si costruisce il piano della lente l_i.
2. Si interseca il raggio con ogni lente l_i, ottenendo i punti di intersezione A_i.
3. Si trova il punto A che corrisponde al punto A_i più vicino all'origine del raggio e che si trova davanti a esso.

4. Si considera R, ovvero la distanza tra il punto A e il pianeta p_i, con il raggio r_i di quest ultimo:

    - Se $R > r_i$ allora abbiamo effettivamente un'intersezione con la lente.
    - Se $R \leq r_i$ allora il raggio sta intersecando il pianeta stesso. In questo caso il vero punto d'intersezione lo si ottiene considerando l'intersezione con la sfera associata al pianeta.

Nel progetto questa parte di logica è codificata dal seguente snippet:

```
//Already defined above
float minT = furthestT;
uint intersectionType = HIT_BACKGROUND;
int closestPlanetIndex = -1;

for(int i = 0; i < data.length(); i++){
    vec3 planeNormal = -lookDir;
    Plane planetLens = Plane(data[i].position, planeNormal);
    float lensIntersectionT = rayPlaneIntersection(result.ray, planetLens);
    //Ignore if the intersection is behind the ray
    if(lensIntersectionT > 0 && lensIntersectionT <= minT){
        minT = lensIntersectionT;
        intersectionType = HIT_LENS;
        closestPlanetIndex = i;
        //check if the ray also hit the planet
        vec3 intersectionPoint = rayPoint(result.ray, minT);
        float distanceFromCenter = length(intersectionPoint - data[i].position);
        if(distanceFromCenter < data[i].radius){
            float distanceFromPlane = sqrt(data[i].radius * data[i].radius -
                distanceFromCenter * distanceFromCenter);
            vec3 frontPoint = intersectionPoint + planeNormal * distanceFromPlane;
            vec3 backPoint = intersectionPoint - planeNormal * distanceFromPlane;
            float frontT = length(frontPoint - result.ray.position) / length(result.ray
                .direction);
            float backT = length(backPoint - result.ray.position) / length(result.ray.
                direction);
            minT = min(frontT, backT);
            intersectionType = HIT_PLANET;
        }
    }
}
```

Alla pagina seguente riportiamo la funziona completa. L'indentazione è stata ridotta a due spazi per favorire l'impaginazione

```
//shaders/compute.shader:136
RayHit castRay(Ray initial, float furthestT){
  RayHit result = RayHit(initial, furthestT, 0, HIT_BACKGROUND, 0, -1);

  int deflections = 0;
  const int MAX = 10;
  for(int j = 0; j < MAX; j++){
    float minT = furthestT;
    uint intersectionType = HIT_BACKGROUND;
    int closestPlanetIndex = -1;

    for(int i = 0; i < data.length(); i++){
      vec3 planeNormal = -lookDir;
      Plane planetLens = Plane(data[i].position, planeNormal);
      float lensIntersectionT = rayPlaneIntersection(result.ray, planetLens);
      if(lensIntersectionT > 0 && lensIntersectionT <= minT){
        minT = lensIntersectionT;
        intersectionType = HIT_LENS;
        closestPlanetIndex = i;
        //check if the ray also hit the planet
        vec3 intersectionPoint = rayPoint(result.ray, minT);
        float distanceFromCenter = length(intersectionPoint - data[i].position);
        if(distanceFromCenter < data[i].radius){
          //adjust minT
          float distanceFromPlane = sqrt(data[i].radius * data[i].radius -
              distanceFromCenter * distanceFromCenter);
          vec3 frontPoint = intersectionPoint + planeNormal * distanceFromPlane;
          vec3 backPoint = intersectionPoint - planeNormal * distanceFromPlane;
          float frontT = length(frontPoint - result.ray.position) / length(result.ray.
              direction);
          float backT = length(backPoint - result.ray.position) / length(result.ray.
              direction);
          minT = min(frontT, backT);
          intersectionType = HIT_PLANET;
        }
      }
    }

    vec3 p = rayPoint(result.ray, minT);
    result.distanceTraveled += length(p - result.ray.position);
    result.t = minT;
    if(intersectionType == HIT_PLANET){
      result.hitType = HIT_PLANET;
      result.planetIndex = closestPlanetIndex;
      break;
    } else if (intersectionType == HIT_BACKGROUND){
      result.hitType = HIT_BACKGROUND;
      break;
    } else {
      result.ray = bendRay(result.ray, data[closestPlanetIndex], p);
      result.hitType = HIT_LENS;
      result.deflections++;
    }
  }
  return result;
}
```

Capitolo 5

Editor grafico

L'editor grafico di GraviLenSim è implementato attraverso la libreria *Dear ImGui*, scelta per la sua semplicità di utilizzo e per il suo stile grafico coerente tra diversi sistemi operativi. All'interno della finestra sono presenti un menu principale e tre pannelli estendibili, il cui layout di default è visualizzato in figura 5.1. Per la gestione di quest'ultimi si utilizza il sistema di docking integrato di ImGui, che permette all'utente di modificarne la posizione e la dimensione in modo intuitivo per massimizzare lo spazio disponibile nella finestra.

I pannelli disponibile nell'editor sono i seguenti:

- Pannello `Quick Actions`: permette di accedere con un singolo click ad alcune voci contenute nel menu principale, rendendo più veloce l'utilizzo del programma durante la creazione dell'universo.

- Pannello `Universe`: in questo pannello vengono visualizzati gli oggetti che compongono l'universo simulato, dando la possibilità per ognuno di modificare i parametri considerati dalla simulazione fisica.

- Pannello `Viewport`: visualizza l'immagine prodotta dal Ray-Tracer. Dal menu principale è possibile specificare come l'immagine in output deve essere visualizzata all'interno di questo pannello, scegliendo tra `Pixel Perfect` (un pixel dell'immagine corrisponde 1:1 con un pixel nella finestra), `Fit` (l'immagine viene ingrandita per occupare più spazio possibile, mantenendo il rapporto d'aspetto) e `Stretch` (l'immagine occupa tutta la dimensione del pannello, ignorando il rapporto d'aspetto).

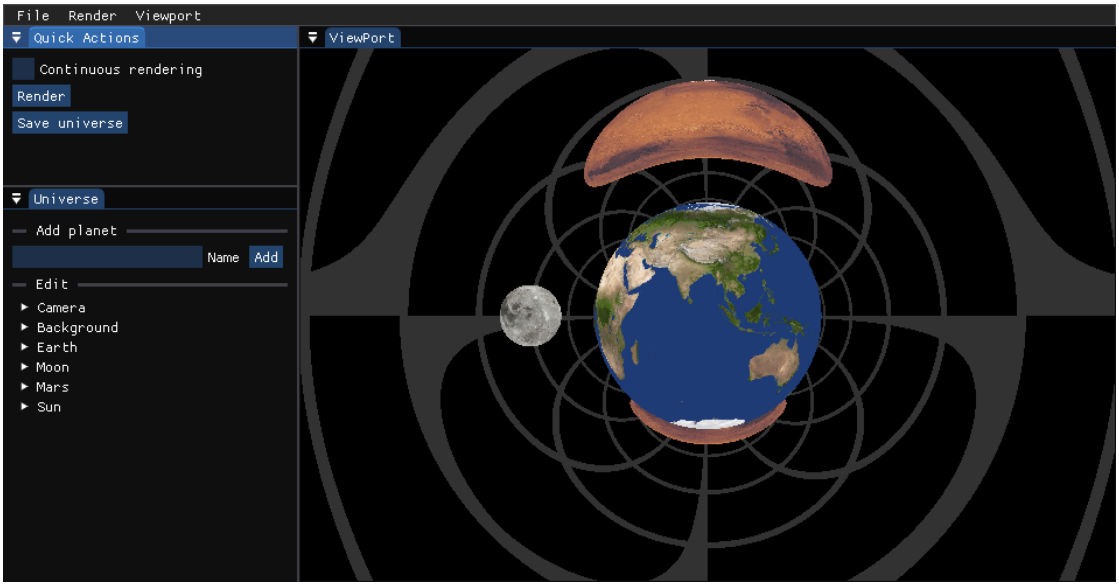
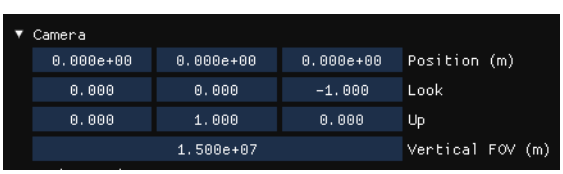
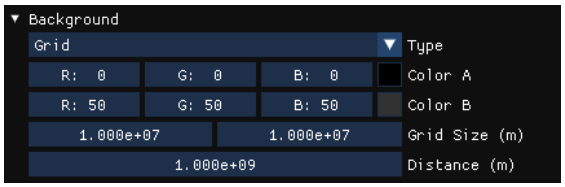


Figura 5.1: Editor di GraviLenSim



(a) Opzioni camera



(b) Opzioni sfondo

Figura 5.2: Opzioni globali all'interno del pannello Universe

Il pannello `Universe` è allo stesso momento il più complesso a causa della quantità di informazioni contenute al suo interno e quello più utilizzato dall'utente, per cui è doveroso un approfondimento sulle opzioni contenute al suo interno. L'universo viene mostrato all'interno del pannello come un elenco di voci collassabili, in cui in cima si trovano le opzioni globali (camera e sfondo) e a seguire è presente la lista di corpi celesti. È possibile aggiungere un nuovo corpo all'universo, specificandone il nome attraverso la casella di testo in cima al pannello e successivamente premendo su `Add`.

Per la camera sono presenti le seguenti opzioni (vedere figura 5.2a):

- `Position`: vettore tridimensionale che contiene le coordinate della posizione, in metri.

- `Look`: versore tridimensionale che definisce la direzione in cui la camera sta osservando l'universo.

- `Up`: versore tridimensionale che permette di ruotare la camera attorno alla direzione definita da `Look`.

Per quanto riguarda lo sfondo sono invece presenti le seguenti opzioni (vedere figura 5.2b):

- `Type`: permette di cambiare la visualizzazione dello sfondo tra `Grid` e `Solid`.

- `Color A`: specifica il colore primario dello sfondo.

- `Color B`: specifica il colore secondario dello sfondo (disponibile solo se la tipologia è impostata su `Grid`).

- `Grid Size`: indica lo spazio in metri che separa le linee parallele della griglia, specificato in modo separato sia per le rette orizzontale che per quelle verticali (disponibile solo se la tipologia è impostata su `Grid`).

- `Distance`: definisce la distanza tra la camera e lo sfondo, perchè questo viene modellato all'interno della simulazione come un piano perpendicolare alla direzione di vista della camera.

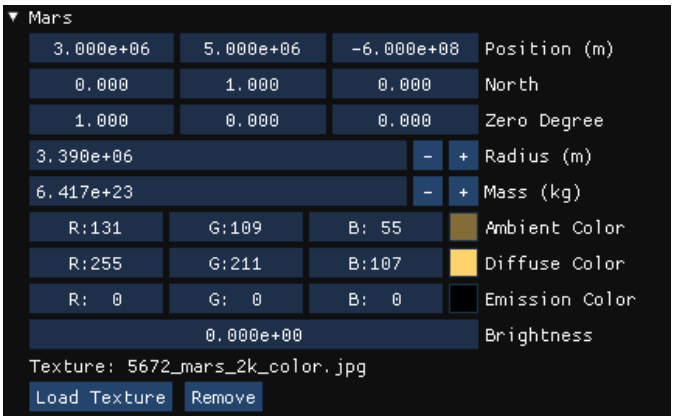


Figura 5.3: Opzioni del singolo pianeta all'interno del pannello Universe

Per ogni corpo celeste che viene aggiunto alla simulazione il pannello `Universe` mostra i seguenti parametri, come mostrato in figura 5.3:

- `Position`: vettore tridimensionale che definisce il centro del corpo celeste, in metri.
- `North`: versore tridimensionale che definisce l'asse di rotazione del corpo, puntando dal centro verso il nord geometrico (rilevante soltanto se viene applicata una texture).
- `Zero Degree`: versore tridimensionale che definisce il meridiano principale (rilevante soltanto se viene applicata una texture).
- `Radius`: indica il raggio del corpo in metri.
- `Mass`: definisce la massa del corpo in kilogrammi.
- `Ambient Color`: specifica il colore di base che viene visualizzato per il corpo.
- `Diffuse Color`: specifica il colore aggiuntivo che viene mostrato nelle regioni illuminate del corpo.
- `Emission Color`: colore che viene emesso dal corpo se il parametro `Brightness` è positivo
- `Brightness`: se questo valore è positivo, indica che il corpo emette luce propria specificando la sua luminosità in watt.
- `Load Texture`: permette la selezione di una texture da applicare sul corpo.
- `Remove`: rimuove il corpo specifico dalla simulazione.

Capitolo 6

Esempi di immagine in output

Questo capitolo è dedicato alla presentazione degli output generati. Gli esempi di immagini forniti hanno lo scopo di dimostrare visivamente le capacità del Ray-Tracer implementato e di convalidare la correttezza della simulazione fisica delle lenti gravitazionali

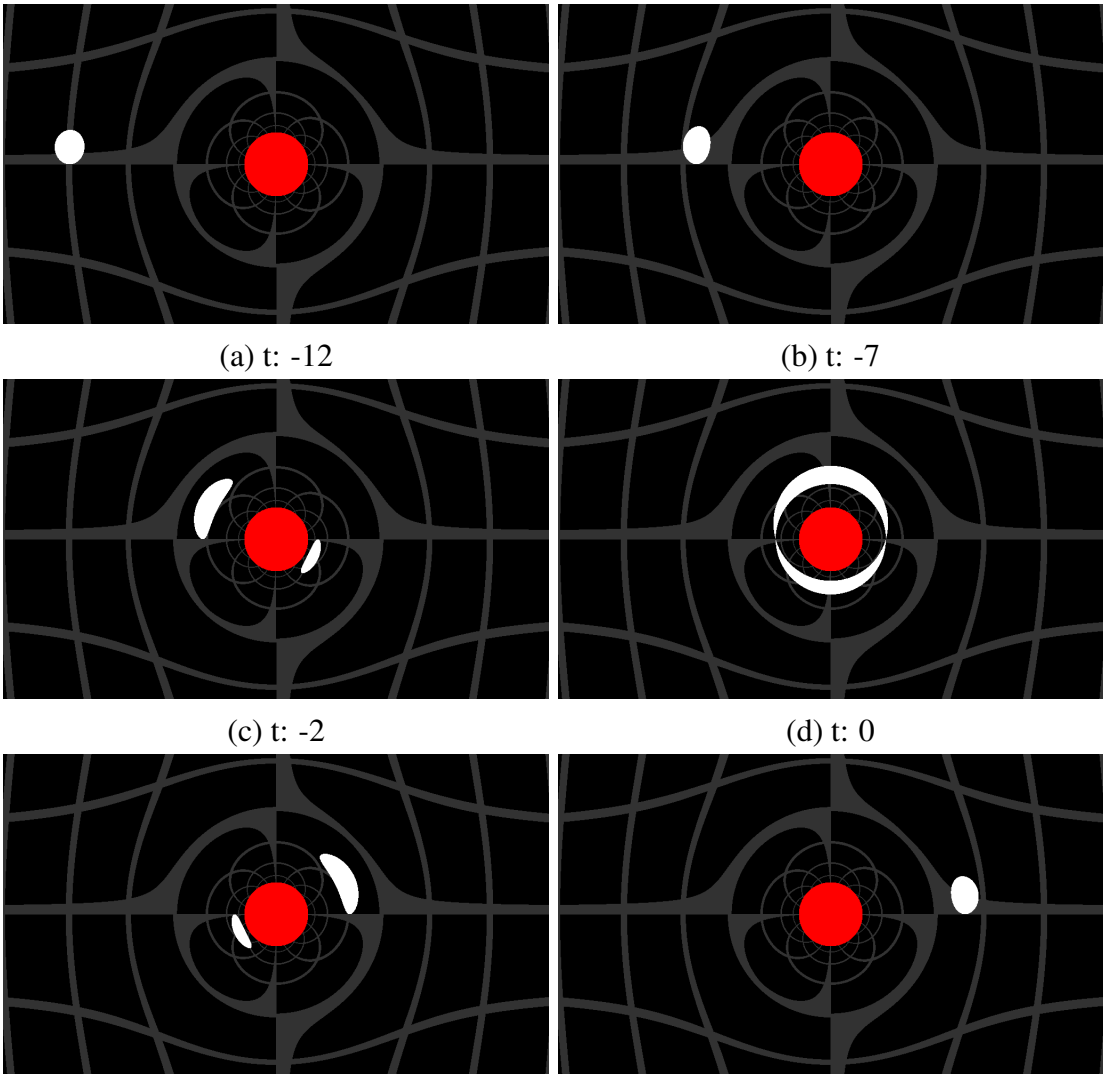
La figura 6.1 illustra la simulazione di una stella (visualizzata in bianco) che transita dietro a una galassia (visualizzata in rosso), che funge da lente gravitazionale. Si può notare come più la stella è allineata con la galassia più l'immagine viene deformata. Nelle figure figure 6.1c e 6.1e è possibile vedere la formazione di una seconda immagine della stella, opposta rispetto al centro della galassia, mentre nella figura 6.1d è presente la formazione quasi completa dell'anello di Einstein.

In figura 6.2 si riprende lo stesso universo della figura 6.1 a cui viene aggiunta una seconda galassia, visualizzata in verde. In particolare nella figura 6.2a l'universo è composto soltanto dalla stella bianca e dalla galassia rossa, per cui la luce della stella viene deviata una singola volta dalla prima galassia, in rosso. Nella figura 6.2b, invece, l'universo è composto dalla stella bianca, dalla prima galassia rossa e dalla seconda galassia verde, in ordine da più distante a più vicino rispetto alla camera.

Questa disposizione degli oggetti nell'universo ha come conseguenza la formazione di un'immagine completamente diversa, in cui la luce della stella viene deviata due volte, la prima dalla galassia rappresentata in rosso e la seconda dalla galassia in verde. Inoltre si può notare come l'immagine della prima galassia in rosso sia anch'essa distorta a causa

della seconda galassia in verde.

In figura 6.3 viene mostrato un sottoinsieme dei pianeti del sistema solare, composto da Terra, Luna, Marte e Sole (non visibile nell'immagine), a cui sono state applicate delle texture illuminate dinamicamente dal Sole. In questo esempio le grandezze sono state esagerate, in particolare la massa della Terra, per rendere immediata la comprensione. A causa della distanza relativa tra Sole, posizionato dietro rispetto alla camera, e Marte l'immagine deformata che si forma di quest'ultimo è molto più debole rispetto alla Terra o alla Luna.

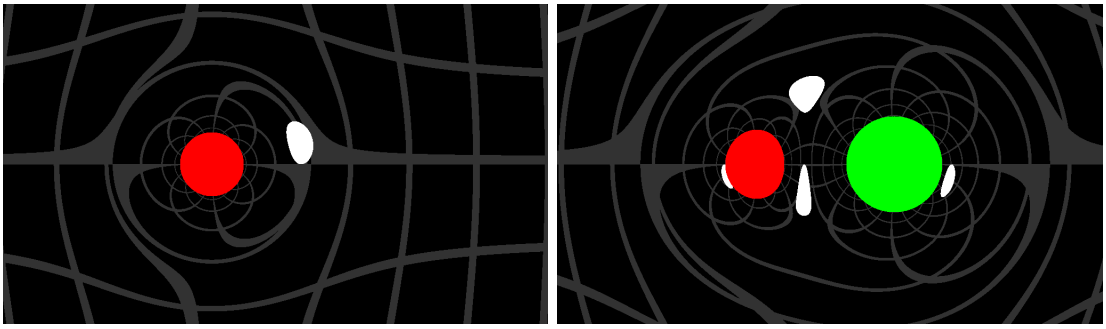


(a) t: -12 (b) t: -7

(c) t: -2 (d) t: 0

(e) t: 2 (f) t: 7

Figura 6.1: Sequenza di immagini che mostrano un esempio di Strong Lensing.



(a) Universo con 2 oggetti (b) Universo con 3 oggetti

Figura 6.2: Esempio di deviazioni multiple di un fotone

Figura 6.3: Esempio di utilizzo di texture applicate ai corpi

Capitolo 7

Conclusione

Per concludere questa tesi rimane soltanto da parlare delle limitazioni di questo progetto allo stesso modo con cui sono state analizzate le limitazioni dei progetti esistenti.

La limitazione principale di questo progetto, come già descritto nella sezione 1.4, ricade nella simulazione fisica in sè. Nonostante l'approssimazione Flat-Sky sia accurata abbastanza da poter generare immagini soddisfacenti, rimane comunque un'approssimazione del modello definito dalla Relatività Generale. Questo è stato il fattore principale che ha spinto all'implementare una delle feature secondarie nel progetto, ovvero la possibilità di sostituire la simulazione fisica senza ricompilazione, così che in futuro la simulazione fisica possa essere resa più accurata.

Un'altra limitazione del progetto, che in parte ricade sempre nella simulazione fisica, riguarda la tipologia di corpi massicci che compongono l'universo simulato. Si potrebbe estendere la simulazione per includere stelle, galassie e soprattutto buchi neri nell'equazione, solo che questo richiederebbe modifiche sostanziali sia all'editor che alle strutture definite nella compute shader in GLSL, visto che come linguaggio quest'ultimo non ha meccanismi di eredità tipici della programmazione ad oggetti.

Questo è GraviLenSim, un progetto che visualizza gli effetti gravitazionali della curvatura dei fotoni attraverso tecniche di Ray-Tracing, disponibile su

https://github.com/MatteoGodzilla/GraviLenSim

(Per questo progetto, sia per quanto riguarda l'implementazione in codice, sia per la stesura di questa tesi, sono state utilizzate soltanto risorse puramente umane).

Bibliografia

[0] Sir. Isaac Newton. *Opticks. or, a Treatise of the Reflections, Refractions, Inflections and Colours of Light*. London, 1704.

[0] Johann Georg von Soldner. *Ueber die Ablenkung eines Lichtstrals von seiner geradlinigen Bewegung*. Berlin: Johann Elert Bode, mar. 1804.

[0] Joachim Wambsganss. «Gravitational Lensing in Astronomy». In: *Living Reviews in Relativity* 1.1 (dic. 1998), p. 12. ISSN: 1433-8351. DOI: 10.12942/lrr-1998-12. URL: https://doi.org/10.12942/lrr-1998-12.

[0] Albert Einstein. «Über den Einfluß der Schwerkraft auf die Ausbreitung des Lichtes». In: *Annalen der Physik* 35 (1911), pp. 898–908.

[0] Albert Einstein. «Die Grundlage der allgemeinen Relativitätstheorie». In: *Annalen der Physik* 49 (1915), pp. 769–822.

[0] F. W. Dyson, A. S. Eddington e C. Davidson. «A Determination of the Deflection of Light by the Sun's Gravitational Field, from Observations Made at the Total Eclipse of May 29, 1919». In: *The Royal Society* 220 (gen. 1920), pp. 571–581.

[0] D. Walsh, R. F. Carswell e R. J. Weymann. «0957+561 A,B: twin quasistellar objects or gravitational lens?» In: *Nature* 279 (mag. 1979), pp. 381–384.

[0] Arthur B. Congdon e Charles R. Keeton. *Principles of Gravitational Lensing*. Springer Praxis Books, 2018.

[0] Katherine Garrett e Gintaras Duda. «Dark Matter: A Primer». In: *Advances in Astronomy* 2011.968283 (2010). DOI: 10.1155/2011/968283. URL: https://onlinelibrary.wiley.com/doi/pdf/10.1155/2011/968283.

[0] S. Vegetti et al. «Strong Gravitational Lensing as a Probe of Dark Matter». In: *Space Science Reviews* 220.5 (lug. 2024), p. 58. ISSN: 1572-9672. DOI: 10.1007/s11214-024-01087-w. URL: https://doi.org/10.1007/s11214-024-01087-w.

[0] Andrew S. Glassner. *An Introduction to Ray Tracing*. Xerox Park: Morgan Kaufmann Publishers, Inc., 1989.

[0] Khronos. *Khronos Releases OpenGL 4.3 Specification with Major Enhancements*. 2012/08. URL: https://www.khronos.org/news/press/khronos-releases-opengl-4.3-specification-with-major-enhancements.

[0] Khronos. *Compute Shader - OpenGL Wiki*. URL: https://wikis.khronos.org/opengl/Compute_Shader (visitato il giorno 05/11/2025).

[0] Khronos. *Interface Block (GLSL) - OpenGL Wiki*. URL: https://wikis.khronos.org/opengl/Interface_Block_(GLSL) (visitato il giorno 05/11/2025).

[0] Khronos. *glTexImage2D - OpenGL 4 Reference Pages*. URL: https://registry.khronos.org/OpenGL-Refpages/gl4/html/glTexImage2D.xhtml#notes (visitato il giorno 05/11/2025).

Librerie utilizzate

[0] *GLFW*. URL: https://github.com/glfw/glfw.

[0] *GLAD*. URL: https://github.com/Dav1dde/glad.

[0] *Dear ImGui*. URL: https://github.com/ocornut/imgui.

[0] *stb_image.h*. URL: https://github.com/nothings/stb/blob/master/stb_image.h.

[0] *stb_image_write.h*. URL: https://github.com/nothings/stb/blob/master/stb_image_write.h.

[0] *GLM*. URL: https://github.com/icaven/glm.

[0] *cxxopts*. URL: https://github.com/jarro2783/cxxopts.

[0] *tinyfiledialogs*. URL: https://sourceforge.net/projects/tinyfiledialogs/.

[0] *json*. URL: https://github.com/nlohmann/json.

Ringraziamenti

Ringrazio tutte le persone che ho conosciuto negli ultimi anni, a partire dal gruppo universitario, passando per quello MTG per arrivare infine ai gruppi più lontani, perchè hanno stravolto in positivo la mia vita, se ripenso al me stesso di cinque anni fa.

Ringrazio la mia famiglia per avermi supportato in questi anni da fuorisede e mi scuso per tutte le chiamate di famiglia perse.

Ma soprattutto ringrazio la moka con finestra a fiori che ha preparato l'innumerevole quantità di caffè necessaria per la stesura di questa tesi.

Ora, vogliate scusarmi, ho un paio di progetti che mi stanno chiamando da anni per essere terminati. :wq acknowledgements