

UNIVERSITÀ DI MILANO

RELAZIONE DI PROGETTO

PROGRAMMAZIONE GRAFICA PER IL TEMPO REALE

**Simulatore di condizioni
meteorologiche**

Studenti:

Daniele Piergigli
Davide Quadrelli

Matricola:

901331
901330

Settembre 10, 2018

Indice

| | | |
|----------|--------------------------------------|-----------|
| 1 | Introduzione | 2 |
| 2 | Librerie utilizzate | 2 |
| 3 | Mondo | 3 |
| 3.1 | Mappa | 4 |
| 3.1.1 | Illuminazione | 4 |
| 3.2 | Camera | 5 |
| 3.3 | Skybox | 6 |
| 4 | Sistema particellare | 8 |
| 4.1 | Pioggia | 9 |
| 4.2 | Neve | 10 |
| 5 | Nebbia | 13 |
| 6 | Fisica applicata al modello | 16 |
| 7 | Valutazione delle prestazioni | 18 |
| 8 | Riferimenti bibliografici | 20 |

1 Introduzione

La relazione presenta un simulatore di condizioni meteorologiche basato sulle librerie grafiche OpenGL [8], dove gli effetti atmosferici sono implementati attraverso tecniche di grafica in tempo reale. La simulazione riproduce gli effetti della pioggia, della neve e della nebbia su un terreno morfologicamente complesso in un ambiente 3D. Il prototipo è quindi in grado di generare, durante l'avanzamento della simulazione, uno o più di questi fenomeni contemporaneamente e di gestirne l'intero ciclo di vita. Oltre alla generazione, il prototipo adatta il paesaggio al fenomeno atmosferico, rendendo il tutto più realistico e omogeneo. Per muoversi all'interno della mappa, si è applicata al modello una camera in prima persona. Questa simula la prospettiva di un personaggio che si muove all'interno dell'ambiente, dove la gestione delle coordinate cartesiane è basata sulla regola della mano destra.

La relazione segue quindi nelle prossime pagine il processo di implementazione del simulatore atmosferico. Da prima, verranno motivate le librerie utilizzate per ottimizzare le performance del prototipo e la sua realizzazione. Successivamente, verranno introdotte le tecniche utilizzate per l'implementazione dell'ambiente, il comportamento della camera al suo interno e la tecnica di *skybox* [7] per aumentare il realismo del cielo. Si passerà quindi a illustrare il funzionamento del sistema particolare, che svolge un ruolo primario all'interno del progetto con una spiegazione dettagliata delle strutture dati create per generare gli effetti di pioggia e neve.

Dopodiché, verrà descritto il modello implementato della nebbia con una soluzione [9] che permette di applicare l'effetto basandosi sull'angolo di visione dell'agente e sul movimento dell'asse Y della camera. Successivamente verrà illustrato il motore fisico per la gestione delle collisioni e della gravità. Questo si baserà su di una libreria esterna, utilizzata principalmente per la generazione dei *collider* e dei *rigidbody* [2]. Infine, verranno mostrati alcuni risultati basati sulla verifica delle performance del modello, attraverso l'analisi del frame rate della scena in particolari condizioni di stress.

2 Librerie utilizzate

Il prototipo è sviluppato utilizzando le librerie grafiche OpenGL nella loro versione 3.3. L'utilizzo di questa particolare versione è motivato dal fatto che quasi tutte le schede video moderne le supportano. Sono state utilizzate, inoltre, per migliorare le performance generali del simulatore e per la sua implementazione, una serie di librerie esterne.

Prima tra tutte GLM [5], una libreria di supporto ad OpenGL che permette di utilizzare matrici e vettori come tipi di dato e di eseguire delle operazioni matematiche

richiamando semplici funzioni. La libreria oltre a funzionare con OpenGL, garantisce anche l’interoperabilità con altre librerie e SDK di terze parti. È quindi un buon candidato per il *rendering*, l’elaborazione delle immagini, le simulazioni fisiche e qualsiasi altro contesto di sviluppo che richiede una libreria matematica.

Altra libreria esterna molto diffusa nel campo della grafica *realtime* è GLFW [6], una libreria Open Source multi-piattaforma per desktop utilizzabile con OpenGL e i suoi derivati che si occupa di fare da *wrapper* per il caricamento delle librerie grafiche. Inoltre, GLFW si presta non solo alla creazione di finestre desktop, ma anche alla gestione degli input da tastiera, rendendola indispensabile in scenari come questo dove la camera deve muoversi nell’ambiente 3D.

Per importare i modelli utilizzati all’interno della scena, il progetto si è inoltre avvalso di Open Asset Import Library [1]. La libreria permette di importare vari formati di modelli 3D in modo uniforme. Questa, mira a fornire una pipeline completa di conversione degli asset da utilizzare nei motori di gioco e/o sistemi di *rendering*, rendendola un valido supporto per il simulatore.

Per il simulatore fisico, invece, si è utilizzato Bullet [3], un motore fisico molto apprezzato dagli sviluppatori il cui utilizzo permette di simulare la fisica di un mondo e di rilevare collisioni in tempo reale all’interno dell’ambiente. Il progetto, quindi, lo utilizza per generare la gravità, i *collider* e i *rigidbody* da applicare alle particelle e al terreno dell’ambiente, in modo da avere una simulazione delle collisioni. La libreria include anche moltissime *utility* di terze parti che si interfacciano con Bullet migliorandone la versatilità. Tra queste, il progetto ha utilizzato TinyObjLoader [4] per generare un *collider* da applicare al terreno del prototipo rispecchiando, nella maniera più fedele possibile, i vertici delle sue *mesh*.

3 Mondo

Per poter costruire il simulatore di condizioni meteorologiche, è stata necessaria la costruzione di un “mondo”, ovvero di una qualche struttura grafica che subisse dei cambiamenti estetici dovuti all’applicazione di differenti condizioni meteo. Si è optato per la costruzione di un cielo tramite *skybox* e il caricamento del modello di un vulcano (con relativa *texture*) dove poter applicare gli effetti di neve, nebbia e pioggia. Per poter visualizzare correttamente il mondo, inoltre, si è optato per una telecamera che può liberamente muoversi nell’ambiente.

3.1 Mappa

La mappa utilizzata nel prototipo consiste nel modello di un vulcano caricato all'interno del progetto tramite *Open Asset Import Library*, che consente di memorizzare tutti i vertici di ogni singola *mesh* che compone il modello. Oltre a queste informazioni, è necessario caricare anche la sua *texture*, indispensabile a rendere graficamente il modello il più verosimile possibile. Per effettuare correttamente il *render* della mappa è inoltre necessario definire la posizione del vulcano, ovvero dove esso si trova nello spazio tridimensionale simulato. In questo modo, dopo aver anche definito la rotazione e la scala da voler applicare ad esso, si può calcolare la *view-model matrix* e la matrice delle normali, fondamentale per l'applicazione delle *texture*. Tutte queste informazioni sono quindi passate allo *shader*, che si occuperà di renderizzare graficamente ogni singolo vertice della mappa.

3.1.1 Illuminazione

La scena, per sembrare realistica, necessita di un'adeguata illuminazione. Per assolvere tale scopo, era necessario un algoritmo che non richiedesse grossi calcoli computazionali, ma che comunque fosse in grado di dare risultati il più realistici possibile. Per questa ragione, si è deciso di utilizzare un *hemisphere lighting*.

Questo tipo di illuminazione riesce, avendo come parametro la direzione della luce, a generare un'approssimazione dell'illuminazione ambientale. Nel dettaglio, dati il colore del cielo e del terreno, la colorazione di ogni frammento dipende dalla sua normale in coordinate globali. Perciò se tale direzione segue la traiettoria della luce, allora assumerà il colore del cielo, altrimenti se è opposta, assumerà il colore del terreno. Tutti gli altri casi sono determinati tramite interpolazione dei due colori. L'*hemisphere lighting*, quindi, si basa sulla seguente formula

$$H_l = \text{lerp}(C_g, C_s, w) \quad (1)$$

dove $C_g, C_s \in \mathbb{N}^3$ rappresentano i 3 valori RGB, rispettivamente, del terreno e del cielo, mentre $w \in \mathbb{R}$ è definita come

$$w = 0.5 * (N \cdot D) + 0.5 \quad (2)$$

dove $N, D \in \mathbb{R}^3$ rappresentano la normale del vertice e la normale la direzione della luce. Computazionalmente, il calcolo della seguente formula non appesantisce molto il processo di *render*, nonostante vada eseguita per ogni frammento del modello ad ogni frame. Gli ultimi parametri necessari al corretto funzionamento dell'illuminazione sono i colori: se per il terreno la scelta del colore corrisponde al valore della

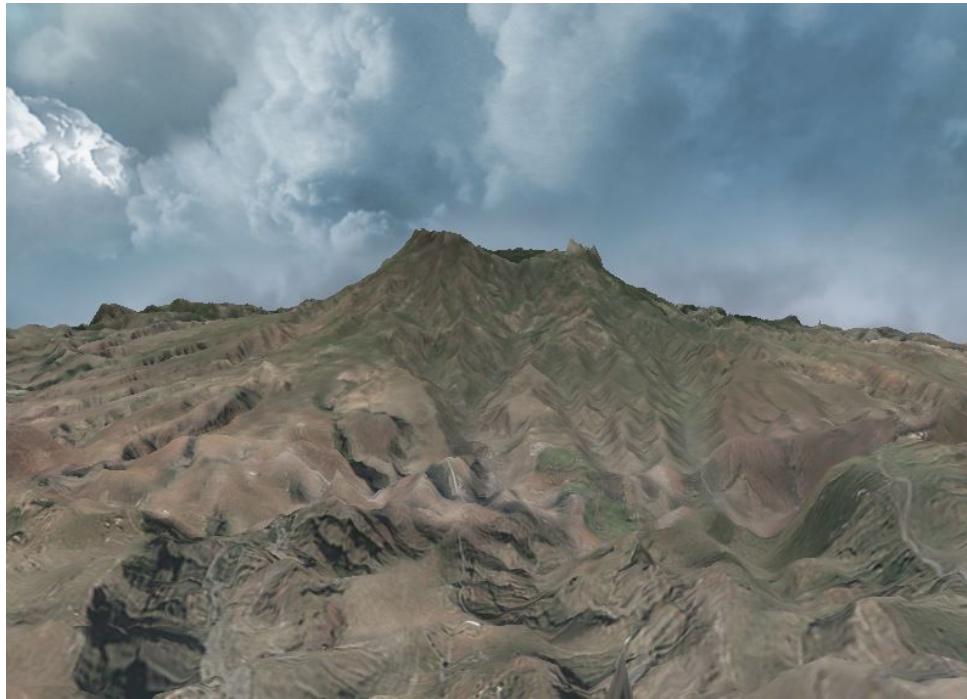


Figura 1: Risultato grafico dell’*hemisphere lighting* applicato al modello della mappa.

texture per quel frammento, per il cielo la soluzione è più ostica. Il colore del cielo va a definire la tonalità verso la quale tenderà la mappa, ed è necessario che corrisponda in maniera abbastanza realistica a quello che si ha all’interno della simulazione. Si è scelto, perciò, di pre-calcolare la media del colore della *texture* (ovvero la media per ogni canale RGB di ogni singolo pixel) utilizzata per il cielo e di utilizzarlo come valore per l’*hemisphere lighting*. Il risultato finale, dato dal modello del vulcano e un cielo nuvoloso, è visibile nella Figura 1.

3.2 Camera

Per poter visualizzare i risultati ottenuti dal simulatore, è necessario un meccanismo che consenta di muoversi all’interno del mondo e di poterlo osservare da diverse posizioni e angolazioni, ovvero una camera.

L’implementazione richiede di definire la sua posizione nello spazio e la matrice di proiezione (definita dall’angolo di visione, l’*aspect ratio* e la profondità di campo). Tale camera consente quindi di poter simulare la visualizzazione degli elementi all’interno del nostro mondo come se fosse una visione in prima persona di un essere



Figura 2: Immagine che mostra come lo *skybox* utilizzato all'interno del progetto consenta la simulazione di un cielo con un ottimo livello di realismo.

umano. Per rendere mobile la camera, e quindi consentire la possibilità di cambiare il punto dal quale si osserva la scena, è necessario cambiare la posizione in base al movimento voluto (ad esempio, per andare avanti, è necessario eseguire uno spostamento lungo l'asse Z).

Si è ritenuta necessaria, inoltre, l'implementazione della rotazione lungo gli assi X e Y, in modo da ruotare la camera di un certo angolo lungo i due assi, cambiando la rotazione rispetto agli assi originali. Sfruttando questa tecnica è possibile muoversi in ogni direzione.

3.3 Skybox

Oltre ad avere degli elementi all'interno della scena e dare la possibilità di poterla osservare da qualsiasi angolazione, si è voluto applicare uno sfondo all'ambiente. Per questo motivo, si è cercato un modo per poter generare un cielo, che fosse il più realistico possibile, ma allo stesso tempo computazionalmente leggero. La soluzione scelta consiste nell'utilizzo di una *cubemap* per ottenere uno *skybox*.

Una *cubemap* consiste in un cubo a cui viene applicata una *texture* unica divisa

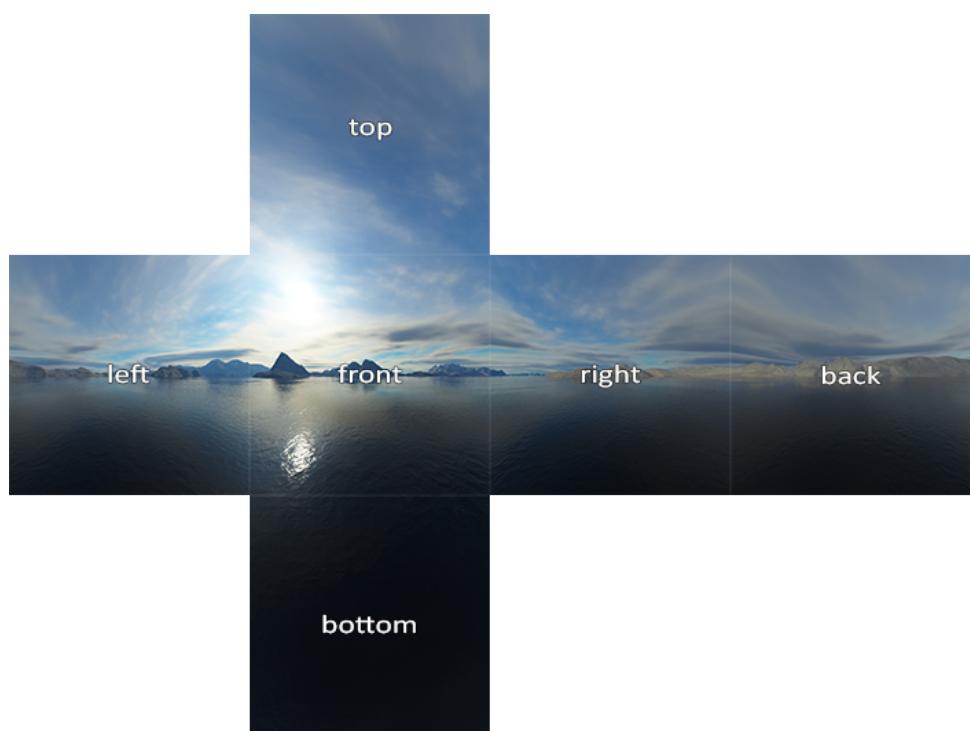


Figura 3: Esempio di *texture* per uno *skybox*: da un'immagine originale vengono ritagliate le 6 facce del cubo, per poi andare ad applicarle alla *cubemap*.

tra le facce che lo costituiscono. Se questo cubo è molto grande, posizionando la telecamera all'interno di esso, il risultato finale è che si è circondati dalla *texture*, ottenendo uno sfondo. Una possibile implementazione, permette l'utilizzo di più *texture* ottenute da un'immagine unica, in modo che esse siano adiacenti fra loro, come nella Figura 3. Applicandole correttamente alla giusta faccia, si ottiene un cielo realistico con uno sforzo computazionale minimo. Il risultato finale ottenuto è visibile nella Figura 2.

4 Sistema particellare

Il sistema particellare è la parte del progetto che si occupa di generare e gestire le particelle degli eventi atmosferici simulati, ovvero pioggia e neve. La gestione delle gocce della pioggia è la stessa dei fiocchi di neve: le uniche differenze fra le due consistono nella grafica, ovvero nei modelli, nei colori delle particelle, nel codice degli *shader* (spiegati nei Capitoli 4.1 e 4.2) e nei parametri fisici utilizzati per rendere la caduta il più realistica possibile.

Questa sezione del progetto ha come scopo, quindi, quello di generare le particelle dell'effetto atmosferico che si sta simulando, interfacciarsi al motore fisico per simulare correttamente la loro caduta, gestire le collisioni e rappresentare graficamente le particelle.

Per la generazione delle particelle, è necessario avere una posizione di partenza da dove farle cadere: per fare ciò è stato implementato un sistema di generazione di posizioni random all'interno di un'area posizionata ad un'altezza fissata. Così, definendo le dimensioni del rettangolo all'interno del quale generare le particelle, e scegliendo da quale altezza le si vuole far cadere, si ottengono delle posizioni casuali utilizzabili come punti di generazione. Ottenute queste posizioni, sarebbe possibile generare tutte le particelle per farle cadere, ma il risultato grafico sarebbe lontanissimo da quello desiderato, in quanto queste nascerebbero nello stesso frame, cadendo tutte assieme e generando una nube densa. Per evitare questo problema, si utilizza un buffer di particelle di dimensione fissata: ad ogni frame, se ne generano solo una parte, fino ad arrivare a riempire totalmente il buffer. Quando una particella avrà una collisione con il terreno, essa verrà rimossa dal buffer e, quindi, potrà essere rilanciata. Con un posizionamento casuale (ma ad un'altezza precisa) e uno *spawn* temporizzato, si ottiene la caduta differita delle particelle, ovvero l'effetto desiderato per poter simulare i due effetti atmosferici.

Per il *rendering* grafico delle particelle, si utilizza lo stesso algoritmo della mappa, descritto in 3.1. A differenza del terreno, le particelle si muovono e ciò richiede che il motore fisico informi il sistema particellare della loro attuale posizione, rendendo possibile la corretta visualizzazione, frame dopo frame, della caduta della particella,



Figura 4: Illustrazione dell’effetto finale del sistema particellare per la simulazione della caduta della pioggia.

come è possibile vedere in Figura 4.

Il sistema particellare è progettato per utilizzare un solo *shader* e, quando esso è attivo, sostituisce anche quello della mappa. Per questo motivo, esso è in grado di renderizzare sia le particelle (costituite solo da un modello e da un colore da applicare), sia il vulcano (che richiede l’applicazione di una sola *texture*). Per il prototipo si è utilizzato uno *shader* per ogni effetto atmosferico.

4.1 Pioggia

La pioggia utilizza la stessa illuminazione dello *shader* della mappa, ovvero l’*hemisphere lighting*, ma si comporta diversamente quando renderizza le particelle e il terreno. Nel caso di una particella, lo *shader* applica l’*hemisphere lighting* come illustrata in 3.1.1, con la sola differenza che applica al canale alpha del colore una trasparenza pari a quella del colore della particella, con entrambi i valori ricevuti in input dal sistema stesso. Quando, invece, si sta renderizzando la mappa, il comportamento dello *shader* varia in base ad un parametro numerico $\gamma \in \mathbb{R}$ che si occupa di dare

un effetto bagnato al terreno. La funzione applicata dallo *shader* risulta essere

$$f_c = \begin{cases} H_l & \text{if } \gamma \leq 0 \\ lerp(H_l, W_c, \gamma) & \text{if } \gamma > 0 \end{cases} \quad (3)$$

dove f_c è il colore finale del frammento, H_l è il colore generato dall'*hemisphere lighting* e W_c è il colore che si vuole che il terreno assuma quando è bagnato. Il sistema particolare si occupa di inizializzare γ quando vengono generate le particelle e, ad ogni frame, di incrementare tale valore fino ad un livello di “saturazione”, oltre al quale non può aumentare ulteriormente. In questo modo, la velocità con la quale si ottiene l’effetto bagnato dipende da quanto viene incrementato γ ad ogni frame. Quanto tempo passi prima che si incominci ad avere un effetto bagnato dipende dal valore iniziale di γ , oltre che al già menzionato incremento. Il massimo livello che si ottiene di effetto bagnato è dettato dal livello di “saturazione”. In questo modo, definendo all’inizio $\gamma \leq 0$, incrementandolo ad ogni frame di un valore $\epsilon > 0$ molto piccolo e definendo il livello di saturazione $\theta \leq 1$, si può andare a parametrizzare il sistema e cercare di ottenere l’effetto che più si preferisce.

In questo caso, si è optato per utilizzare i seguenti valori:

$$\begin{aligned} W_c &= (0, 0, 0) \\ \gamma &= -0.3 \\ \epsilon &= \frac{\Delta_t}{30} \\ \theta &= 0.6 \end{aligned}$$

dove Δ_t corrisponde alla differenza di tempo, in secondi, tra il frame precedente e quello attuale. Con questa configurazione, si ha che il colore della mappa comincerà ad insurirsi lentamente dopo qualche secondo dall’attivazione del sistema particolare, in modo da dare il tempo alle prime particelle di esser già venute a contatto con la mappa. La saturazione finale è molto bassa per ottenere un terreno più scuro, ma che sia ancora ben riconoscibile, in modo tale da sembrare bagnato senza la presenza, però, di riflessione luminosa. Un’illustrazione dell’effetto finale è visibile nella Figura 5.

4.2 Neve

La renderizzazione della neve utilizza un sistema simile a quello della pioggia, descritto dalla formula 3, che varia per via della lenta generazione della neve sulla mappa invece che un lento applicarsi dell’effetto.

Per ottenere questo risultato non si deve lentamente aumentare la saturazione della

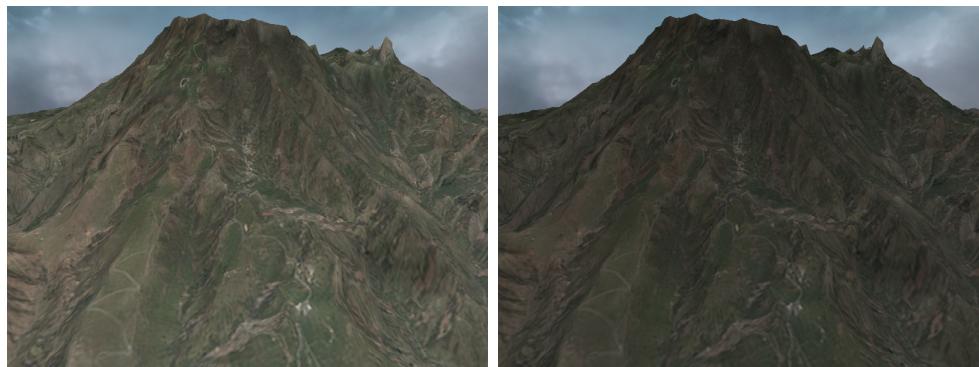


Figura 5: Comparazione fra la mappa in condizioni normali (a sinistra) e la mappa con applicato l'effetto bagnato a saturazione raggiunta (a destra).

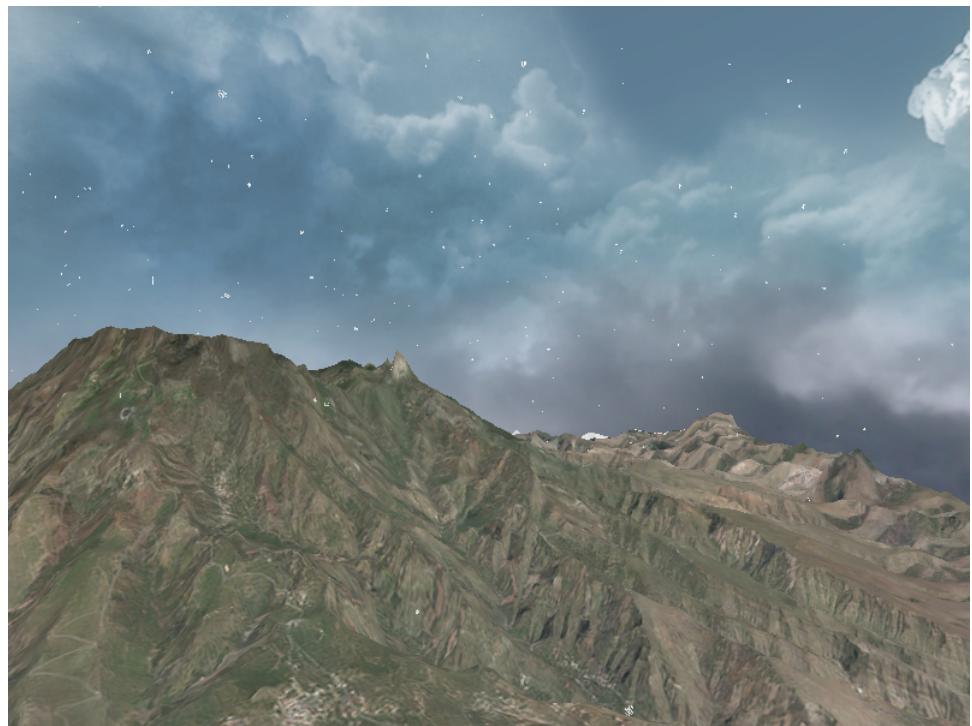


Figura 6: Illustrazione dell'effetto finale del sistema particellare per la simulazione della caduta della neve.

neve sulla mappa, perché altrimenti sarebbe sempre presente sul terreno diventando solo più densa e bianca, ma è necessario trovare un modo per generarla gradualmente, con una densità e un colore fissi. Per realizzare tale effetto, si è utilizzato un confronto tra il prodotto scalare della normale del vertice in coordinate del mondo e la direzione del movimento della neve con un valore di soglia. Se il prodotto la supera, quel vertice è innevato, altrimenti no. Andando ad ogni frame a diminuire questo valore, si ottiene che la neve comincerà a generarsi in maniera sparpagliata sulla mappa, partendo da piccole zone e ingrandendosi sempre più. Tale fenomeno si verifica perché, per sempre più vertici, il prodotto scalare supererà il valore di soglia $\lambda \in \mathbb{R}$ fino a rendere tutta la mappa innevata.

Nel dettaglio, la formula utilizzata per calcolare il colore del singolo vertice risulta essere

$$f_c = \begin{cases} \text{lerp}(S_c, T_c, \alpha) & \text{if } N_V \cdot D_s \geq \lambda \\ H_l & \text{if } N_V \cdot D_s < \lambda \end{cases} \quad (4)$$

dove f_c è il colore finale del frammento, S_c è il colore della neve, H_l è il colore dell'*hemisphere lighting*, $\alpha \in \mathbb{R}$ è la costante di interpolazione tra i due colori, N_v è la normale del vertice in coordinate del mondo e D_s è la direzione della neve.

In questo *shader*, a differenza di quello della pioggia, il valore di soglia non cambia il livello di interpolazione fra i colori, ma il valore della condizione della funzione. Quindi, applicando un $\lambda \geq 0$ e decrementandolo ad ogni frame di $\pi > 0$ molto piccolo sino a un valore di saturazione $\eta \geq -1$, si ha un effetto di lento incremento della neve sulla superficie che raggiunge il suo massimo con $\eta = -1$.

Si è deciso, quindi, di utilizzare i seguenti valori:

$$\begin{aligned} S_c &= (255, 255, 255) \\ \lambda &= -0.1 \\ \pi &= \frac{\Delta_t}{40} \\ \eta &= -0.98 \end{aligned} \quad (5)$$

dove Δ_t corrisponde alla differenza di tempo in secondi tra il frame precedente e quello attuale. Con questa configurazione, si deve attendere qualche secondo prima che il terreno inizi a innevarsi con una velocità moderata fino a una saturazione quasi totale. In questo modo, si lascia qualche parte di mappa ancora normale dando un discreto senso di realismo al sistema. Un'illustrazione dell'effetto finale è visibile nella Figura 7.



Figura 7: Comparazione fra la mappa in condizioni normali (a sinistra) e la mappa innevata a saturazione raggiunta (a destra).

5 Nebbia

Un altro fenomeno atmosferico tipico è la nebbia. La nebbia è il fenomeno meteorologico per il quale una nube si forma a contatto con il suolo. È costituita da goccioline di acqua liquida o cristalli di ghiaccio sospesi in aria. A causa della diffusione della luce solare da parte dell'acqua in sospensione, la nebbia si manifesta come un alone biancastro che limita la visibilità degli oggetti. Con la computer grafica, la nebbia può essere creata in diversi modi, uno dei principali, utilizza la formula (6) per generare la nebbia con un andamento esponenziale della sua densità (vedi Figura 8).

$$f = e^{-d \cdot b} \quad (6)$$

L'equazione, con d la distanza dalla camera e b il fattore di attenuazione, è un modello semplificato del fenomeno che avviene nel mondo reale dove, l'intensità della luce, viene ridotta sulla distanza a causa dell'assorbimento e della dispersione delle particelle d'aria. L'intensità del raggio luminoso, quando incontra le particelle, viene assorbita da un piccolo fattore di attenuazione. L'intensità della luce, quindi, diminuisce in modo esponenziale con la distanza dall'oggetto.

Come per la generazione della nebbia, anche la distanza d può essere ricavata dal modello in molti modi. Una possibilità, è quella di utilizzare la profondità Z della camera. Questa tecnica soffre però di un grosso problema dovuto ad anomalie visive che si possono generare quando si ruota la telecamera (vedi Figura 9). Dovendo ruotare la visuale ed evitare questo effetto indesiderato con oggetti che escono ed entrano all'interno della zona con la nebbia, si è preferita utilizzare una tecnica

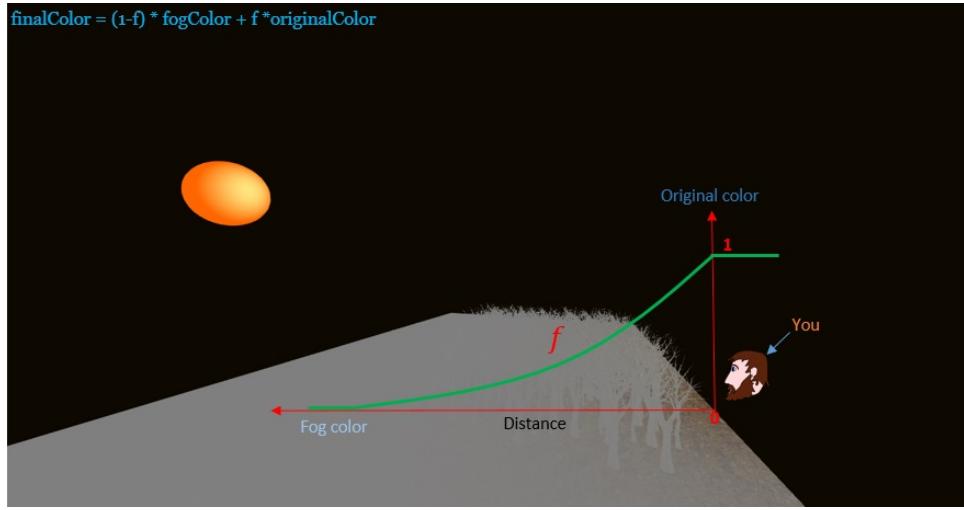


Figura 8: Formula matematica della nebbia applicata ad un modello.

basata sul raggio di visione. In questo modo, si calcola la distanza effettiva tra i vertici e la camera attraverso la formula (7) (vedi Figura 10).

$$d = abs((cameraPos \cdot terrainPos \cdot vertexPos) . z) \quad (7)$$

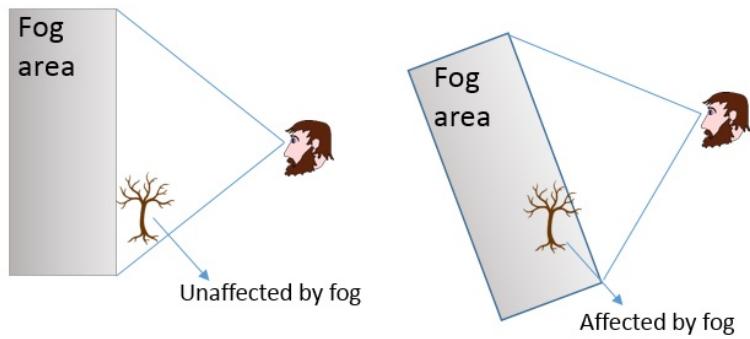


Figura 9: Tecnica basata solo sulla distanza dalla telecamera. Staticamente la nebbia non influenza l'albero ma, ruotando la visuale, l'effetto viene applicato.

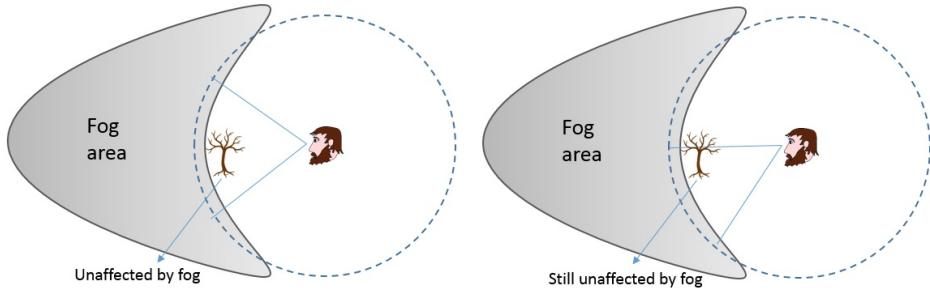


Figura 10: Tecnica basata sul raggio di visione. Anche ruotando la visuale, la nebbia non influenza l'oggetto.

Per dare alla nebbia un effetto più realistico, si è ampliato l'effetto finale aggiungendo i tre processi principali che regolano l'interazione tra la luce e la nebbia: l'assorbimento, l'emissione e lo *scattering*. Possiamo quindi considerare un volume di particelle che emettono, assorbono e disperdoni i raggi luminosi. L'emissione contribuisce all'energia delle particelle luminose, quando la luce viene assorbita, l'energia viene convertita in un'altra forma di energia. Inoltre, la dispersione reindirizza la direzione originale della luce in altre direzioni quando questa colpisce una particella, perdendo intensità. Tale fenomeno è chiamato *out-scattering*. Se si combinano gli effetti di assorbimento con gli effetti di *out-scattering*, otteniamo l'effetto di attenuazione. La luce che rimbalza, inoltre, colpirà un'altra particella aumentandone l'intensità. Questo è chiamato invece *in-scattering*. Applicare queste tecniche alla nebbia del modello richiede molta matematica. Per semplificare i calcoli si è applicata la formula (8) che utilizza la formula (6) per calcolare l'attenuazione f_2 e l'*in-scattering* f_1 .

$$finalColor = (1 - f_1) \cdot fogColor + f_2 \cdot lightColor \quad (8)$$

Da notare che per f_1 si è utilizzata la distanza dalla camera, mentre, per f_2 , la distanza si basa sulla direzione della luce rispetto al terreno. L'unico problema che limita il realismo del prototipo rimane il fattore di attenuazione, che a differenza del mondo reale, dove questo fattore può variare da particella a particella, deve essere considerato costante. Per questo motivo la densità della nebbia sarà uniforme ovunque.

Per via di come il *vertex shader* ed il *fragment shader* interpolano la nebbia, si è dovuto decidere dove calcolare la distanza e gli altri parametri. Se vengono caricati nel *vertex shader*, il colore viene interpolato sulla primitiva (triangoli). Il metodo, è più economico da calcolare e se si hanno dei modelli con molti poligoni, non si

notano molte differenze rispetto alla seconda soluzione. Tuttavia, se si dispone di un terreno piatto, si ottengono risultati visivi negativi per quanto riguarda la profondità della nebbia e, quindi, è meglio effettuare i calcoli attraverso il *fragment shader*. Avendo un terreno morfologicamente complesso da trattare, si è preferito calcolare tutto attraverso il *vertex shader*, in quanto non vi erano differenze visive tra le due soluzioni. Quindi, tutti i calcoli sulle distanze e sulle posizioni delle luci sono stati eseguiti all'interno del *vertex shader* e, successivamente, passati al *fragment*, dove sono stati eseguiti i calcoli per applicare l'effetto finale in Figura 11.



Figura 11: Effetto finale della nebbia sul terreno di simulazione.

6 Fisica applicata al modello

La fisica, può essere applicata ad un modello in molti modi a seconda delle necessità. In questo caso, il prototipo necessitava di una corretta gestione della gravità per il movimento delle particelle e le loro collisioni con il terreno. Per ottenere questo risultato, è stato utilizzato un motore di simulazione fisica contenuto nella libreria Bullet descritta nel Capitolo 2. La libreria, permette di creare un ambiente controllato dove inserire la simulazione del modello, applicandogli una versione personalizzata della fisica reale. Per prima cosa, quindi, è stato generato l'ambiente fisico tramite Bullet a cui è stata applicata una gravità personalizzata. Questa, da sola, non è in

grado di sortire alcun effetto sugli elementi che compongono il modello. Per essere applicata ad un elemento, questo deve avere almeno un *collider* applicato con un *rigidbody*. Quest'ultimo, permette di controllare la posizione di un oggetto nell'ambiente attraverso la sua simulazione fisica.

La libreria permette di applicare ad ogni elemento un certo tipo di *collider*. Nel caso specifico di questo progetto si è preferito applicare, per comodità, alle particelle uno *sphere collider* e al terreno una *convexhull*. Quest'ultima, utilizza quanti più vertici possibili del modello per approssimare la reale morfologia del terreno. Se infatti, per le particelle, una sfera è in grado di avvolgere il modello dell'elemento senza che si notino artefatti visivi durante le collisioni, il terreno ha bisogno di una rappresentazione quanto più fedele all'originale per non riscontrare problemi (vedi Figura 12). Per questo motivo, si è preferito utilizzare un'ulteriore libreria esterna chiamata TinyObjLoader che, dato un .obj in input, genera un *convexhull* in output utilizzando quanti più vertici possibili del modello di partenza per avere una visualizzazione grafica ottimale del *collider* finale. L'utilizzo di questa tecnica viene applicato durante la fase di caricamento della simulazione e mai più ripetuto; per questo motivo, anche se il calcolo è computazionalmente esoso, rimane accettabile ai fini del risultato finale. Il *collider* del terreno, infatti, è statico e per questo motivo, una volta generato, è utilizzato solo per il controllo delle collisioni. Le particelle, invece, sono dinamiche, per cui il *collider* ha una massa che gli permette di utilizzare la gravità e dei parametri che ne controllano il comportamento quando questo finisce a contatto con altri *collider*. In questo caso, sono stati applicati anche dei valori di frizione e smorzamento angolare, per bloccare l'elemento e non permettergli di rimbalzare contro il terreno una volta a contatto.

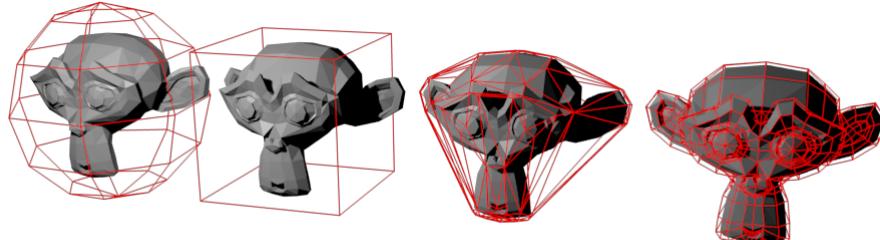


Figura 12: Da sinistra verso destra il modello è stato approssimato tramite una sfera, un cubo e una *convexhull*. L'ultima, è la rappresentazione dei vertici dell'immagine originale.

Ogni elemento ha il suo *rigidbody* e questo viene inserito all'interno di una lista in modo tale da poter essere utilizzato per simularne il movimento mediante Bullet con la gravità definita. La libreria, inoltre, permette di controllare in diversi modi

le collisioni tra *collider*. Il sistema utilizzato si serve di una *callback* per controllare costantemente le collisioni che avvengono all'interno dell'ambiente. Questo costrutto permette, in questo caso, di eseguire sullo stesso processo del *render* grafico dell'ambiente anche il controllo delle collisioni che avvengono nel simulatore. Questo, viene eseguito per ogni elemento presente nell'ambiente, al quale è stato applicato un *collider* e ha un riferimento all'interno di una apposita lista controllata dalla *callback*. Il problema di questa applicazione, però, si verifica quando si vuole identificare, tra tutte le particelle presenti nella lista della *callback*, quale tra queste ha colpito il terreno. Infatti, gli elementi nella lista non sono in ordine e non esiste, quindi, un identificativo univoco da poter ricavare per distinguere nella lista ogni singolo elemento. Perciò, non si è in grado di notificare alla particella di essere arrivata al capolinea e di dover essere fatta sparire e reimessa tra le particelle da generare. Per risolvere il problema, è stata costruita una struttura apposita che viene inserita all'interno della lista della *callback* e utilizzata per i successivi controlli di collisione. La struttura è definita dai seguenti parametri:

- Il tipo di elemento, che può essere particella o terreno;
- Il *rigidbody* dell'elemento, utilizzato dalla *callback* per controllare lo spostamento dell'elemento e quindi le collisioni;
- Il particellare stesso, che rimane non impostato per il terreno.

In questo modo, il *collider* utilizza il *rigidbody* nella struttura per controllare la collisione. Se questa è tra il terreno e una particella, allora la particella viene impostata in maniera tale da essere generata nuovamente, renderizzandola e riposizionandola al frame successivo in un'altra posizione random.

7 Valutazione delle prestazioni

Per valutare le prestazioni del progetto, si è andati a verificare il frame-rate medio della simulazione di ogni effetto atmosferico. Essendo la nebbia sovrapponibile ad ogni altro fenomeno, le valutazioni sono state effettuate con cielo sereno, pioggia e neve applicando e rimuovendo la nebbia. In questo modo, si può valutare l'incidenza degli effetti atmosferici attivati sul frame-rate medio. Ovviamente, si può definire il cielo sereno con nebbia come il caso standard. Per avere una comparazione reale delle prestazioni, inoltre, i test sono stati eseguiti lanciando la simulazione e variando le condizioni climatiche sia sulla scheda integrata, sia su quella dedicata del sistema utilizzato per l'esperimento. In Tabella 1 e in Tabella 2, sono visibili i risultati dei test per le due schede video.

Dalla comparazione, appare evidente come la scheda video dedicata, quando il siste-

ma particellare e il motore fisico non sono attivi, abbia prestazioni decisamente più elevate di quella integrata, mentre in tutti gli altri casi le prestazioni sono simili e, in alcuni casi, peggiori rispetto alla scheda video dedicata. In realtà, la GPU integrata viene stressata, durante i calcoli, tre volte di più rispetto a quella dedicata, ma la leggera differenza di frame è dovuta al passaggio dei dati dalla RAM alla memoria dedicata della scheda video, perché la scheda integrata ha accesso diretto alla memoria centrale e non necessita di questo ulteriore passaggio.

Il netto calo prestazionale tra il tempo sereno, che raggiunge un altissimo numero di frame, e la pioggia e la neve, che ne hanno molti di meno, è dovuto alla CPU che, quando è occupata anche ad eseguire i calcoli del motore fisico e del sistema particellare, fa da collo di bottiglia per le prestazioni grafiche.

| fps medi (integrata/i5 7300HQ) | | | | | | |
|--------------------------------|-----|--------|----|--------|----|---------|
| N° Particelle | S | S + Nb | P | P + Nb | Nv | Nv + Nb |
| 0 | 840 | 805 | X | X | X | X |
| 500 | X | X | 27 | 26 | 26 | 26 |
| 1000 | X | X | 13 | 13 | 13 | 13 |
| 1500 | X | X | 9 | 9 | 9 | 9 |
| 2000 | X | X | 7 | 7 | 7 | 7 |
| 2500 | X | X | 6 | 6 | 6 | 6 |

Tabella 1: Frame-rate del simulatore al variare delle condizioni atmosferiche con scheda integrata, dove S indica tempo sereno, Nb la nebbia, P la pioggia e Nv la neve.

L’andamento generale rimane comunque scarso, soprattutto quando le particelle iniziano ad essere sopra le migliaia. Valutando i dati, possiamo però assumere che il modello stesso di simulazione degli effetti particellari sia computazionalmente oneroso quando applicato al modello. L’assunzione deriva dal fatto che all’avvio del prototipo, quando non si hanno sistemi particellari attivi, le prestazioni sono molto elevate mentre, già con poche particelle, le prestazioni calano drasticamente. In ultima analisi, la nebbia applicata a qualsiasi fenomeno atmosferico incide in maniera trascurabile sulle performance generali del simulatore.

| fps medi (GTX 960/i5 4690) | | | | | | |
|----------------------------|------|--------|----|--------|----|---------|
| N° Particelle | S | S + Nb | P | P + Nb | Nv | Nv + Nb |
| 0 | 5063 | 5189 | X | X | X | X |
| 500 | X | X | 23 | 21 | 19 | 19 |
| 1000 | X | X | 12 | 12 | 12 | 12 |
| 1500 | X | X | 8 | 8 | 9 | 8 |
| 2000 | X | X | 7 | 6 | 7 | 6 |
| 2500 | X | X | 6 | 6 | 6 | 6 |

Tabella 2: Frame-rate del simulatore al variare delle condizioni atmosferiche con scheda dedicata, dove S indica tempo sereno, Nb la nebbia, P la pioggia e Nv la neve.

8 Riferimenti bibliografici

- [1] assimp. assimp - Open Asset Import Library. http://assimp.sourceforge.net/lib_html/, 2012.
- [2] David Baraff. Physically based modeling: Rigid body simulation. *SIGGRAPH Course Notes, ACM SIGGRAPH*, 2(1):2–1, 2001.
- [3] Bullet Physics development team. Bullet Physics SDK. <http://bulletphysics.org/wordpress/>, 2018.
- [4] Syoyo Fujita. tinyobjloader. <https://github.com/syoyo/tinyobjloader>, 2018.
- [5] G-trunc. OpenGL Mathematics. <https://glm.g-truc.net/0.9.9/index.html>, 2018.
- [6] GLFW. GLFW. <https://www.glfw.org>, 2018.
- [7] José M Noguera, Rafael J Segura, Carlos J Ogáyar, and Robert Joan-Arinyo. A hybrid rendering technique to navigate in large terrains using mobile devices. *Computer Graphics International, Singapore*, 2010, 2010.
- [8] Dave Shreiner and The Khronos OpenGL ARB Working Group. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1*. Addison-Wesley Professional, 7th edition, 2009.

- [9] Richard S Wright Jr, Nicholas Haemel, Graham M Sellers, and Benjamin Lipchak. *OpenGL SuperBible: comprehensive tutorial and reference*. Pearson Education, 2010.