ALMA MATER STUDIORUM
UNIVERSITÀ DEGLI STUDI DI BOLOGNA

Seconda Facoltà di Ingegneria
Corso di Laurea in Ingegneria Elettronica, Informatica e delle Telecomunicazioni

VISUALIZZAZIONE REAL-TIME DELLA DIFFUSIONE SUBACQUEA DELLA LUCE

Elaborata nel corso di: Fondamenti di Computer Graphics

Tesi di Laurea di:
MICHELE MATTEINI

Relatore:
Prof. SERENA MORIGI

ANNO ACCADEMICO 2011–2012
SESSIONE II

PAROLE CHIAVE

Rendering

Superficie dell'acqua

Modellazione grafica

Animazione digitale

XNA

Indice

Capitolo 1

Introduzione

L'aumento della potenza grafica disponibile sui computer desktop ha reso possibile la diffusione del 3d in una grande varietà di applicazioni. Un settore nel quale lo sviluppo è stato evidente è quello video-ludico nel quale vengono studiati modelli in grado di fornire rappresentazioni sempre più accurate e fisicamente corrette degli ambienti, che vengono quindi utilizzati per creare algoritmi che eseguono in real-time. Questa tesi vuole discutere un modello per simulare la visualizzazione ed animazione dell'acqua ossia studiare il comportamento della luce sulla sua superficie e la sua diffusione subacquea.

Verrà quindi presentato il progetto di un modulo aggiuntivo per un motore grafico esistente, che permetterà a questo di visualizzare in modo realistico l'oceano. Si affronterà quindi sia lo sviluppo di un modello teorico per il rendering dell'acqua, sia le problematiche di implementazione del modello attraverso un algoritmo real-time e dell'adattamento di questo alla struttura del motore grafico. I risultati dell'implementazione potranno essere visualizzati attraverso uno strumento per l'editing di paesaggi, già presente all'interno del framework del motore grafico.

Capitolo 2

Strumenti e tecnologie

L'implementazione del progetto verrà effettuata sul motore grafico M90 precedentemente sviluppato in .NET. Questo è scritto in linguaggio C# e si basa sul framework XNA per l'IDE Visual Studio 2008. I sorgenti del motore grafico sono disponibili e sarà eventualmente possibile adattarli e modificarli per consentire l'inserimento di un nuovo modulo.

2.1 Xna Game Studio

Microsoft XNA Game Studio è un ambiente di programmazione che estende Visual Studio permettendo la creazione di videogiochi. Contiene al suo interno l'XNA framework, un set di librerie basate su DirectX che facilitano la creazione di videogiochi in ambiente .NET[4]. Le applicazioni create con questa piattaforma possono essere poi esportate su varie piattaforme Microsoft quali Windows(Xp, Vista, 7), Xbox 360, Windows Phone 7. Il linguaggio di sviluppo per XNA è C#, mentre possono essere aggiunti ai progetti degli shader personalizzati e scritti in HLSL. L'ambiente fornisce anche semplici shaders preconfigurati che non necessitano di script aggiuntivi e integra la gestione del suono e dell'input[12]. Sono state rilasciate diverse versioni del framework in corrispondenza dell'aggiornamento delle librerie DirectX, per garantire il supporto delle ultime funzionalità. La versione utilizzata nel corso di questa tesi è la 3.1 che supporta le librerie directX fino alla versione 10.1.

2.2 DirectX

DirectX è un set di librerie che offrono funzionalità di gestione dell'hardware grafico, specifiche per lo sviluppo di giochi 3d. Sono suddivise in vari componenti che si occupano della gestione di: rendering, input, suono, ecc. Queste librerie creano render di oggetti 3d prendendo in input strutture geometriche definite da

vertici e rappresentandole sullo schermo dopo opportune trasformazioni matriciali e rasterizzazione.

2.3 HLSL e shader

La programmabilità dell'hardware grafico in directX è possibile grazie a degli script compilati da queste librerie, sviluppabili in linguaggio HLSL (high level shading language) detti *shader*. In questa tesi si farà riferimento alla versione 3.0 in cui gli shader hanno una struttura fissa:

- **Vertex Shader** Blocco dello script che viene richiamato per ogni vertice della struttura da renderizzare. Qui vengono di solito eseguite le trasformazioni dei vertici e calcolati i parametri di output che verranno utilizzati come input per il pixel shader.
- **Pixel Shader** Eseguito per ogni pixel dell'immagine rasterizzata, ha come input un interpolazione dei parametri in output dai vertici vicini, e come output solitamente un colore RGB rappresentante il colore del pixel.
- **Technique** È la singola tecnica utilizzabile per lo shading di un oggetto, ed è costituita da più *passi*, ognuno dei quali contiene una coppia pixel/vertex shader.

In un file HLSL possono essere presenti più pixel e vertex shader, raggruppati in vari modi attraverso varie *technique*.

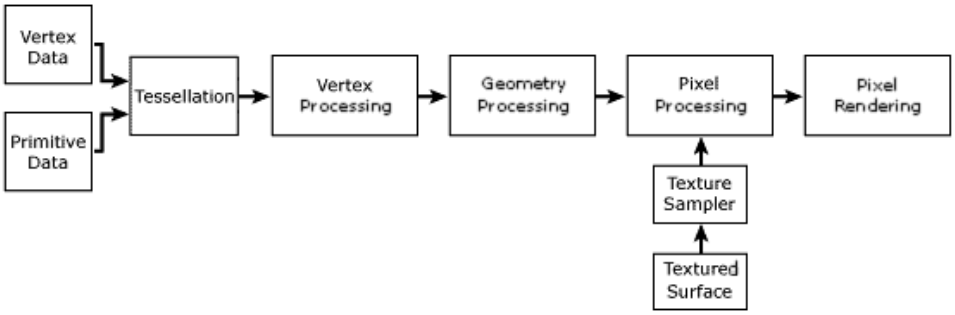


Figura 2.1: *Pipeline di rendering DirectX 9 così come presentata su msdn.microsoft.com: Vertex Processing identifica l'eseguzione del vertex shader, pixel processing quella del pixel shader.*

2.4 M90 Graphic Engine

È il motore grafico sul quale verrà implementato il modulo. Sono disponibili i sorgenti, ma per effettuare un analisi occorre conoscere il suo funzionamento

interno e struttura. M90 verrà quindi documentato in un capito a parte (vedi Sez. 3).

Capitolo 3

Motore grafico M90

M90 è il motore grafico che verrà utilizzato per l'implementazione del modello per l'acqua che verrà trattato in questa tesi. È interamente sviluppato in XNA ed è costituito da un framework che permette il rendering real-time di paesaggi fotorealistici per videogiochi come quello mostrato in Figura 3.1, integra inoltre un editor basato sul suo stesso framework che permette di creare un paesaggio personalizzato e salvarlo su un file.

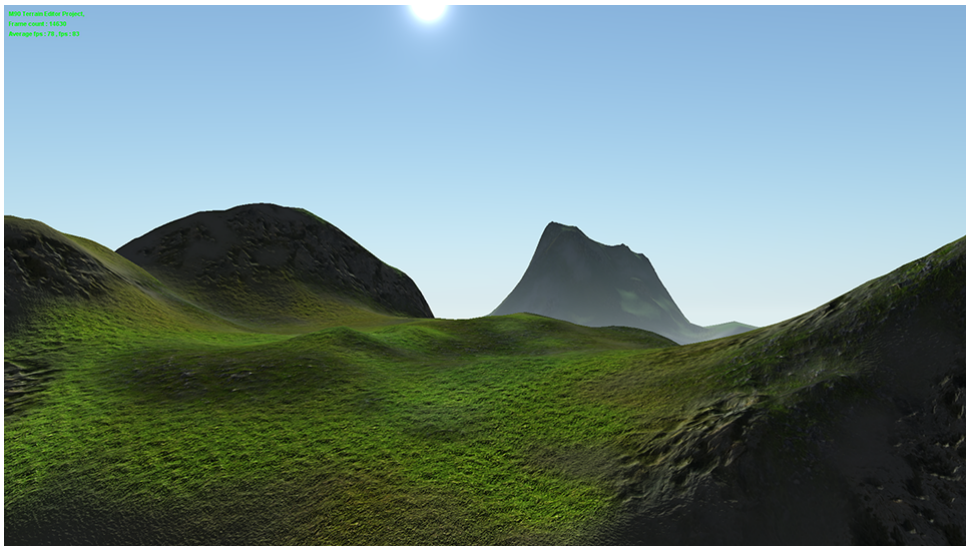


Figura 3.1: *Screenshot di un rendering in M90*

3.1 Struttura modulare

Il motore grafico è costituito da più moduli, ognuno dei quali occupa un namespace diverso, questi moduli sono implementati in Visual Studio come progetti separati.

- **M90** Contiene un insieme di classi, strutture ed utilità non presenti in .NET o XNA, comprese strutture che definiscono vertici per il rendering personalizzate.
- **M90.Engine** Contiene la classe GraphicEngine (vedi Sez. 3.2) e le relative classi derivate, più le strutture e gli enumeratori presenti nell'interfaccia di queste.
- **M90.Movements** Contiene le classi che si occupano di trasformare gli input da mouse e tastiera in posizioni e matrici di traformazione utilizzabili dal motore grafico per spostare il punto di vista all'interno della scena.
- **M90.RealTime** In questo modulo sono implementate tutte le classi che riguardano la gestione del tempo, sia per centralizzare alcuni calcoli e facilitare l'accesso al timer si sistema, sia per gestire eventi temporali nel motore grafico. Infatti in quest'ultimo caso il tempo può essere accelerato, dilatato o fermato ed occorre mantenere una nozione unificata di tempo condivisa da tutti i moduli.
- **M90.Resources** Questo modulo è semplicemente un contenitore per tutte le risorse utilizzate dall'editor. Tutti gli shader utilizzati nel gioco vengono compilati in questo progetto, per facilitare la risoluzione dei riferimenti e favorire la modularità.
- **M90.Resources.Management** Le risorse caricate dal modulo M90.Resources necessitano una gestione particolare poiché occorre allocarle / de-allocarle seguendo determinate politiche. Queste politiche vengono implementate dalle classi presenti in M90.Resources.Management che forniscono inoltre meccanismi per la propagazione di parametri in alcuni tipi di risorse (vedi Sez. 3.3).
- **M90.Scene** Prima della fase di rendering (vedi Sez. 3.2.3) vengono distinti tra i componenti del motore grafico, gli oggetti della scena, ossia quelli che devono essere visualizzati nell'immagine finale. Questi hanno degli attributi in comune che devono essere inizializzati. In questo modulo vengono definiti gli attributi e le politiche comuni a questi oggetti.
- **M90.Scene.Elements.Atmosphere** Il modulo che gestisce l'atmosfera (vedi Sez. 3.5).
- **M90.Scene.Elements.Shadows** Il modula che gestisce le ombre proiettate dagli altri oggetti disegnati. Attualmente non disponibile.

- **M90.Scene.Elements.Terrain** Il modulo che gestisce il terreno (vedi Sez. 3.4).
- **M90.Scene.Elements.Vegetation** Gestione della vegetazione (alberi, erba ecc). Attualmente non disponibile.
- **M90.Scene.Illumination** Si occupa della gestione dei punti di illuminazione e del loro tipo.
- **Xna-Map-Editor** Contiene un programma basato sul motore grafico, che permette attraverso una GUI, la creazione ed il salvataggio di progetti rappresentanti paesaggi (vedi Sez. 3.6).

3.2 La classe GraphicEngine

Questa è una classe del modulo M90.Engine (vedi Sez. 3.1) ed è la classe che un utente deve istanziare per utilizzare le funzionalità messe a disposizione dal framework. Integra ed utilizza tutti i moduli sviluppati e fornisce un interfaccia semplice per caricare e visualizzare i progetti dei paesaggi, precedentemente creati nell'editor (vedi Sez. 3.6).

3.2.1 Costruttore

La classe espone un solo costruttore con la seguente signature:

```
1 public GraphicEngine(GraphicQuality startQuality, IntPtr targetHandle);
```

Il primo parametro è costituito da un istanza di GraphicQuality che inizializza un parametro degli oggetti di scena rappresentante la qualità visiva con la quale vengono renderizzati, ed è gestita internamente dai singoli oggetti.

Il secondo parametro rappresenta un *handle* (puntatore) ad un controllo grafico sul quale effettuare il rendering, ad esempio una finestra di Windows.

3.2.2 Metodi di interfaccia

Verranno di seguito riportati i metodi di interfaccia più importanti che espone questa classe con il relativo commento:

```
/// Inizializza lo stato della classe con un nuovo paesaggio base.
/// <param name="dimension">Dimensione in metri del Paesaggio da
  inizializzare.</param>
public void NewMap(int dimension);

4
/// Salva un paesagggio sullo stream fornito come parametro.
/// <param name="writer"></param>
public void SaveMap(BinaryWriter writer);
```

```
9   /// Carica un paesagggio da uno stream e configura la classe per il suo
      rendering.
    /// <param name="reader"></param>
    public void LoadMap(BinaryReader reader);

    /// Disegna un fotogramma della scena sul controllo di output.
14  public void RenderFrame();
```

I primi tre metodi si occupano di modificare e salvare lo stato del classe, che è rappresentato appunto da un paesaggio. Il metodo **RenderFrame** effettua il rendering del paesaggio, e può essere utilizzato per creare una singola immagine, o chiamato in un loop per produrre filmato continuo.

3.2.3 Shading Pipeline

La procedura di rendering si compone di diversi passi che portano alla composizione dell'immagine finale. In questo motore grafico è costituita da 9 passi che sono implementati nel metodo *RenderFrame()* di questa classe:

1. **Update** In questa fase il motore grafico chiama i metodi di update dei vari componenti di cui è composto, permettendo a questi di aggiornare il loro stato interno al frame corrente (ad esempio lo spostamento del punto di vista causa l'aggiornamento del LOD tree del terreno, il passaggio del tempo cambia la posizione del sole). Durate questa fase possono inoltre essere effettuati dei rendering prelimirari, per creare texture dinamiche da utilizzare durante il rendering della scena vera e propria.

2. **G-Buffer rendering** I G-Buffers (Geometry buffers) sono delle texture sulle quali viene rasterizzata la geometria della scena finale, ma non rappresentano un immagine completa perchè vengono utilizzati shaders che codificano nel colore del pixel, delle informazioni parziali, piuttosto che il colore finale. Questo passo fa parte di una tecnica chiamata *Deferred Shading* in cui l'applicazione dello shader finale viene effettuata dopo la rasterizzazione, in modo da ridurre la complessità computazionale di scene in cui sono presenti molto calcoli per pixel. Qui vengono quindi renderizzati tre G-buffer:

   - *Color:* contente il colore HDR 32bit diffuso dagli oggetti, trasformato secondo una codifica LogLUV[3].
   - *Normal:* contenente la normale in tangent-space della scena in quel punto compressa in 8 bit[6].
   - *Depth:* in cui ogni pixel rappresenta la profondità della scena in quel punto.

   Questi vengono creati in un unico passo attraverso la tecnica MRT *(Multiple render targets)*[4] utilizzando texture della stessa dimensione dell'immagine finale.

3. **Screen-space shading** Una volta creati i G-buffers, questi vengono combinati con un opportuno shader la cui complessità è data dal numero di pixel contenuti nelle texture, e non è più dipendente dalla profondità della scena. Infatti in questo passo viene applicato un puro effetto di elaborazione di immagine, senza prendere in considerazione il modello geometrico della scena. Durante l'applicazione dello shader la posizione spaziale di un pixel viene ricostruita dalla sua profondità (ottenuta dal campionamento del relativo G-buffer) e dalla proiezione utilizzata per rasterizzare la scena[9].

4. **Diffusione atmosferica** A questo punto viene calcolato il colore dell'atmosfera nei pixel dove non è presente alcun oggetto (vedi Sez. 3.5).

5. **Auto-Shaded objects** Il motore grafico supporta anche il rendering di oggetti esterni il cui modello di illuminazione è del tutto separato dal resto della scena. Le chiamate per il rendering di questi oggetti viene effettuata ora, ed è richiesto uno uno shader separato da quello per il deferred shading fornito esternamente.

6. **Alpha-transparent objects** Disegno del oggetti semi-trasparenti : deve essere necessariamente effettuato alla fine della pipeline, desegnarli prima porterebbe ad artefatti grafici dovuti ai test di profondità durante la rasterizzazione.

7. **Copia della scena** La scena ora si trova ancora su una texture, in questa fase viene copiata nel backbuffer.

8. **Oggetti rifrattivi** In questa fase vengono renderizzati tutti gli oggetti rifrattivi, ossia quelli che possono essere attraversati dalla luce, ma che ne provocano la distorsione. Alcuni esempi sono un bicchiere, un cristallo, la superfice dell'acqua ecc. Per simulare questo effetto si fa ricorso all'immagine della scena creata fin'ora, disponibile in una texture.

9. **Post processing** L'ultima fase è costituita da effetti post-process, ossia effetti di elaborazione dell'immagine applicati alla scena finale (Es : modifica del contrasto, anti aliasing, motion blur, ecc).

3.2.4 Proprietà

La classe espone alcune proprietà che permettono di ottenere informazioni sullo stato del rendering o di impostarne alcuni moduli, sono qui elencate le più importanti:

```
1 //Restituisce il passo di rendering currentemente in eseguzione
  //(Vedi Sez. Shading pipeline)
  public PipelinePass CurrentPass{ get; }

  //Imposta la risuluzione dell'oggetto grafico su cui viene
6 //effettuato il rendering.
  public Point TargetResolution{ set; }
```

```
//Restituisce un valore booleano che indica se il motore grafico usa
//l'intero schermo come superficie su cui disegnare.
public bool IsFullScreen { get; }

//Imposta un oggetto rappresentante l'adattatore grafico utilizzato.
public GraphicsDevice Device { get; set; }

//Imposta il gestore di risorse attraverso il quale il motore grafico
//caricherà quelle da lui utilizzate
public AdvContentManager Content { get; }

//Abilita / Disabilita l'input da tastiera e mouse
public bool InputEnabled { get; set; }
```

3.2.5 Eventi

La classe espone anche alcuni eventi, che servono principalmente per sincronizzare azioni del motore grafico con un utilizzatore esterno:

```
public event EventHandler Render;
public event EventHandler LoadResources;
public event EventHandler ReleaseResources;
public event EventHandler Update;
public event EventHandler PostProcess;
```

Ad esempio l'evento *LoadResources* viene chiamato quando il motore grafico carica le risorse esterne prima di iniziare il rendering, una classe esterna composta da *GraphicEngine* può sfruttare questo evento per caricare le propie. L'evento *Render* viene invece invocato alla fine del rendiering di ogni fotogramma, se l'applicazione che utilizza il motore grafico è composta da una GUI di windows, i suoi eventi di refresh possono essere sincronizzati a questo evento, rendendo l'interfaccia fluida senza affrontare problematiche di multithreading sul disegno di un'interfaccia.

3.3 Shaders e global parameters

Gli shaders utilizzati dal motore grafico sono raggruppati in un unico modulo M90.Resources poichè in questo modo possono essere facilmente usati dei riferimenti di tipo *#Include* per renderne il loro sviluppo modulare. Infatti molti shaders necessitano di parametri simili che sarebbe vantaggioso uniformare : ad esempio le matrici di rasterizzazione o le condizioni atmosferiche. Le dichiarazioni di queste variabili globali vengono centralizzate in appositi shaders (es. AtmCommon.fx contiene tutti i parametri e funzioni riguardanti la diffusione atmosferica) che poi vengono inclusi dagli altri, ma questo non è sufficiente da solo perchè Xna non mette a disposizione nessuna gestione di parametri condivisi tra più shaders. Questo meccanismo è stato quindi implementato nel modulo M90.Resources.Management dalla classe che si occupa di gestire risorse globali:

AdvContentManager. Questa tiene traccia di tutte le risorse che sono state caricate, tra cui anche gli shaders, e offre metodi per la gestione trasparente di questi parametri globali. Ecco l'implementazione dell'overload per l'impostazione di un parametro globale di tipo float:

```
public void SetGlobalParameter(string name, float value)
{
        //Scorre i nomi di tutti gli shader caricati
        foreach (string fxName in fxAssetNames)
        {
                //Ottiene il riferimento ad una istanza dello shader
                Effect e = (Effect)resources[fxName].ModifyResource();
                //Se lo shader contiene il parametro richiesto
                //questo viene impostato a "value"
                EffectParameter ep = e.Parameters[name];
                if (ep != null) ep.SetValue(value);
        }
}
```

3.4 Modulo di rendering del terreno

Nella rappresentazione di paesaggi realistici, un'approccio diffuso è creare come base per il terreno un unica mesh, che viene poi modellata per creare montagne, colline, letti di fiumi e laghi ecc. Anche per questo motore grafico è stato seguito questo approccio, che verrà illustrato separando la presentazione del metodo per generare e gestire la mesh di cui è composto, dall'implementazione dello shader.

3.4.1 Modello geometrico e LOD

La mesh del terreno visualizzata alla fine della procedura di rendering non è altro che una griglia di vertici che vengono uniti per formare una superficie uniforme per il terreno, attraversio un index buffer con un semplice pattern per la creazione delle facce. La posizione verticale di ogniuno di questi vertici può poi essere modificata e determina la conformazione del terreno. M90 è stato progettato per consentire il rendering real-time di terreni costituiti da un totale di più di 67 milioni di vertici, che normalmente sarebbe impossibile memorizzare e rasterizzare in una frazione di secondo. Per questo motivo sono state implementate classi che si occupano di gestire la memoria utilizzata dall'allocazione dei vertici e il dettaglio con cui diverse aree della mesh vengono renderizzate. Verranno ora presentate le più importanti contenute nel modulo per il terreno, che permetteranno di capirne il funzionamento:

- **Terrain** È la classe principale che fornisce un unterfaccia per il disegno del terreno. Questo viene gestito internamente, dividendolo in più zone, implementate attraverso una matrice di *TerrainZone*.
- **TerrainZone** Rappresenta un singolo blocco di terreno. Gestisce tutte le risorse associate alla zona, memorizzando le informazioni per ogni vertice

costituite da: posizione, normale, peso dei layers per multitexturing, tinta (vedi Sez. 3.4.2). Memorizzare tutte queste informazioni per 67 milioni di vertici richiederebbe troppa memoria e questa classe si occupa proprio di ridurne l'occupazione: le informazioni sui vertici vengono conservate in un formato compresso che viene convertito in quello esteso su richiesta. Infatti mentre le zone vicine al punto di vista richiedono il formato esteso al massimo livello di dettaglio per essere visualizzate correttamente, per quelle più lontane, che sono disegnate a bassa risoluzione, è sufficiente allocare una quantità ridotta di vertici. Considerando il fatto che la maggior parte delle zone vengono disegnate a minimo dettaglio e che l'allocazione di memoria per queste zone è di circa 3 ordini di grandezza inferiore rispetto a quelle ad alto livello di detteglio, il risparmio di memoria è notevole e rende questo algoritmo molto scalabile rispetto alla dimensione del terreno. Questa classe contiene un istanza di *ResourceManager* utilizzata nell'allocazione del formato esteso delle risorse, ed una di *LODTree* per gestire il livello di dettaglio all'interno della zona.

- **ResourceManager** Durante gli spostamenti del punto di vista sulla superficie del terreno, le zone che si avvicinano causano l'allocazione di risorse e memoria per aumentare il loro dettaglio, mentre quelle che si allontanano eseguono operazioni opposte rilasciando risorse. La memoria totale consumata rimane la stessa, ma le continue allocazioni e liberazioni di memoria rallentano il motore grafico. Questa classe si occupa proprio di evitare questo problema, fornendo un'insieme di risorse pre-allocate condivise da tutte le zone di terreno e le distribuisce in maniera trasparente incapsulando le operazioni di allocazione e rilascio.

- **LODTree** È la classe che implementa l'argoritmo di LOD per una singola zona ed è implementata attraverso un quadtree nel quale ogni nodo rappresenta una divisione della zona di terreno in sotto-aree. Il livello di profondità dell'albero determina il numero di sotto-aree in cui viene divisa una zona e viene regolato run-time. Ogni nodo/foglia dell'albero è rappresentato con un'istanza di *LodNode*.

- **LodNode** È una struttura contenente le informazioni su una singola area di terreno nel quale una *TerrainZone* viene suddivisa ed è dichiarata come segue:

```
public struct LODNode
{
        //codice rappresentante la configurazione delle facce.
        //(Vedi IndexBufferDictionary)
        public ulong LODCode;
        //rappresenta il livello di dettaglio da utilizzare
        //per disegnare quest'area
        public int Tessellation;
        //il più piccolo palallelepipedo in grado di
        //contenere quest'area
        public BoundingBox NodeBox;
```

```
12          //minimo, massimo e media delle derivate parziali
            //su questa area di terreno
            public float MaxDerive;
            public float MinDerive;
            public float MedDerive;
17
            public Vector3 Center
            {
                    get
                    {
22                          return (NodeBox.Min + NodeBox.Max) / 2f;
                    }
            }

}
```

- **IndexBufferDictionary** Per il rendering della mesh per la superficie del terreno, non sono sufficienti i vertici, ma occorre esplicitare l'ordine in cui l'adattatore grafico andrà poi a considerarli per formare le facce. Per fare questo è sufficiente creare due array: il primo in cui salvare tutti i vertici necessari(Vertex Buffer), ed il secondo in cui sarà presente una sequenza di indici che faranno riferimento alle posizioni dei vertici nel primo array (Index Buffer). Durante il rendering sarà quindi sufficiente considerare i vertici nella sequenza espressa dall'Index Buffer. Nornalmente l'index buffer è una struttura statica, che non viene mai modificata, ma nel caso del terreno non è possibile mantenerla tale. I vertici del terreno sono salvati per ogni *TerrainZone* in una struttura unica, ma questi vengono renderizzati a blocchi più piccoli di dimensione variabile (fino a 64 sottoblocchi per zona), con tessellazione variabile, impiegando diversi index buffer e mantenendo costante il vertex buffer per la zona. Precalcolare tutti gli index buffer possibili per una zona non è una strada percorribile perchè il loro numero è elevatissimo : c'è infatti da considerare il fatto che i vari blocchi di terreno visualizzati a risoluzione diversa non combaciano tra loro ed occorre introdurre variazioni per gli index buffer in modo che non si formino buchi nel terreno (*gaps*). La soluzione introdotta è implementata dalla classe IndexBufferDictionary: questa genera index buffer su misura a runtime e li salva nel suo stato in una sorta di cache attraverso una struttura dati di tipo hash table. Questo approccio è efficace perchè è stato constatato che nonostante le possibili configurazio degli index buffer siano diverse decine di migliaia, la maggior parte di queste non si presentano mai e quelle effettivamente utilizzate sono meno di cento.

La classe *Terrain* gestisce il rendering e l'aggiornamento del terreno procedendo come segue:

1. Prima del rendering viene eseguito un algoritmo che calcola il livello di dettaglio da utilizzare per ogni zona: questo scorre tutte le *TerrainZone* ed aggiorna i loro *LODTree*.

2. Nel metodo che si occupa del rendering viene effettuata una iterazione che scorre tutte le zone e configura per ogniuna i parametri necesari al suo rendering tra cui il vertex buffer, che è lo stesso per ogni sotto-area.

3. Per ogni zona viene effettuata un'iterazione che scorre il quadtree e disegna ogni sotto-area. Per ottenere l'index buffer corretto viene effettuato l'accesso a *IndexBufferDictionary* con un codice presente in ogni *LODNode*, se il buffer non è presente, viene calcolato a runtime.

3.4.2 Shading e multitexturing

Lo shader per il terreno si basa su una tecnica chiamata *multitexturing*, in cui il colore del terreno è dato da più texture che vengono interpolate tra loro con una media pesata[2]. Variando i pesi delle texture sul terreno si varia la texture che viene applicata. Ora verranno illustrate le varie fasi che segue il processo di shading del terreno come se fossero parte dello stesso algoritmo, anche se visto che il motore grafico segue una pipeline di tipo deferred (vedi Sez. 3.2.3), lo shader non viene eseguito in un unico blocco ma alcune fasi vengono ritardate fino a dopo la rasterizzazione.

1. Per prima cosa vengono ricavate le informazioni sulla normale e sui pesi delle texture che vengono ricostruiti a partire da quelli presenti nei vertici in formato compresso.

2. Attraverso l'interpolazione del multitexturing viene ricavato il valore di bump map che viene utilizzato per la tecnica di *Parallax mapping* che fornisce come output un offset da aggiungere alle coordinate texture per migliorare l'effetto di profondità sul terreno.

3. Le nuove coordinate fornite dal parallax mapping vengono utilizzate per campionare le texture per il terreno attraverso il multitexturing. A questo colore viene applicata un ulteriore tinta contenuta nei vertici.

4. Il normal per il terreno viene ricavato dalle normal map relative ad ogni color texture per il terreno e viene quindi effettuata una trasformazione per portare questo normal in uno spazio tangente alla superficie del terreno.

5. Viene calcolato un fattore di illuminazione dovuto a tre luci: sole, luna e luce ambientale. Per le prime due si moltiplica il loro colore per il prodotto scalare tra da direzione e la normale del terreno ricavata al passo precedente. Per la luce ambientale si moltiplica il colore con la sua intensità, che dipende solo dall'orario che deve essere simulato. Questi tre fattori vengono sommati e poi moltiplicati al colore del terreno.

6. Viene infine calcolato un fattore volumetrico per rappresentare nebbia e diffusione della luce solare nell'atmosfera. Questo viene interpolato al colore del terreno con un'intensità pari alla percentuale di visibilità in quel punto:

più ci si allontana dal punto di vista, più la visibilità, cala ed il colore del terreno tende a quello della della nebbia.

Nell'approccio deferred le ultime due fasi sono quelle che vengono ritardate: queste sono infatti comuni a tutti gli oggetti renderizati.

3.5 Modulo di rendering dell'atmosfera

L'atmosfera è basata sui modelli fisici di diffusione della luce di Mie e Rayleigh che vengono utilizzati per approssimare il colore del cielo. L'algoritmo si divide in due passi: in una prima fase due texture vengono compilate come LUT per i due modelli di diffusione, nella seconda fase le lut vengono campionate per ottere il colore con cui disegnare il cielo[7], il cui modello geometrico non è altro che una sfera centrata sul punto di vista e con raggio pari alla massima distanza visibile.

- **Passo 1 - approssimazione della diffusione:** Gli assi delle LUT che devono essere compilate con il modello di diffusione rappresentano l'angolazione del cielo in cui stiamo guardando, una delle due angolazioni (quella perpendicolare al percorso del sole durante il giorno) è rappresentata solo per metà perchè lungo quella traiettoria la diffusione è sempre simmetrica. L'argoritmo è di tipo iterativo ed approssima la diffusione in una singola direzione nel seguente modo :

  1. Un segmento virtuale viene tracciato dal punto di vista fino al limite esterno dell'atmosfera, nella direzione che si vuole simulare.
  2. Il segmento viene partizionato in n punti, ogniuno dei quali è rappresentato da un vettore Vn, e viene effettuata una iterazione tra questi.
  3. Per ogni Vn viene calcolata la distanza ottica percorsa da un raggio solare dal sole a Vn e poi da Vn al punto di vista utilizzando un modello atmosferico che tiene conto della variazione della densità atmosferica in funzione dell'altitudine.
  4. Attraverso i modelli di Mie e Rayleigh viene approssimato lo spettro della luce che giunge al punto di vista e gli viene assegnata un'intensità pari alla densità atmosferica in Vn.
  5. I contributi delle diffusioni di ogni Vn vengolo sommati per ottenere lo spettro di diffusione dei due modelli in una determinata direzione.
  6. Spettro ed intensità vengono convertiti in un colore RGB che viene salvato nelle LUT.

  I modelli di diffusione citati dipendono dalla lunghezza d'onda della luce, e per ottenere uno spettro luminoso è necessario calcolarli per ogni lunghezza d'onda. Nell'implementazione originale venivano utilizzate per praticità le sole lunghezze d'onda RGB, in questo modo non è necessaria una conversione per ottenere il colore digitale. In questa implementazione sono stati utilizzati

invece 4 campioni dello spettro completo, che producono un risultato più realistico.

- **Passo 2 - shading dell'atmosfera:** Lo shader che si occupa di calcolare il colore da applicare sulla mesh dell'atmosfera non effettua un sempliche campionamento delle LUT calcolate al passo precedente, ma anche questo effettua varie operazioni:

  - Per prima cosa vengono calcolate le coordinate angolari in cui campionare le LUT a partire dalla posizione dei vertici.
  - La LUT per la diffusione di Rayleigh viene campionata ed il valore viene moltiplicato per una funzione di attenuazione che varia in funzione dell'angolo tra la direzione del sole e la direzione del punto di atmosfera da simulare, rispetto al punto di vista. Questa funzione rappresenta il fatto che l'intensità luminosa diffusa da una particella non è la stessa in tutte le direzioni. Questa potrebbe essere direttamente inserita nella LUT ma il suo costo computazionale è basso ed utilizzarne campioni precalcolati degrada notevolmente la qualità dell'output finale.
  - Alla diffusione principale vengono aggiunti dettagli come stelle e luna. La luna è semplicemente una texture che viene proiettata sulla sfera in una direzione che varia con il tempo. Per le stelle si utilizza invece una cube-map a cui viene moltiplicata una texture di rumore campionata con cordinate variabili nel tempo che crea un effetto di animazione sulla loro luminosità.
  - Viene aggiunta una diffusione statica dovuta alla luce lunare, approssimata con un gradiente di colore.
  - Il colore ottenuto viene interpolato con un ulteriore fattore dovuto alla simulazione della nebbia.
  - A questo punto il colore diffuso nell'atmosfera è completo, ma non viene simulato il sole vero e proprio. Questo viene generato attraverso la diffusione di Mie: questa è la generalizzazione di quella di Rayleigh per particelle di dimensione variabile, e qui viene utilizzata un'approssimazione per particelle abbastanza grandi da rispondere alla stesso modo per ogni frequenza. Questo semplifica notevolmente il modello e si ottiene una funzione di fase con un picco nella direzione della luce stessa che genera appunto una simulazione del fascio di luce solare.
  - Il colore ottenuto con questo algoritmo copre un intervallo di luminosità elevato, per poterlo visualizzare correttamente occorre applicare una scala logaritmica che avvicina tra loro colori con luminosità molto diverse.

L'argoritmo diviso in due fasi permette una certa flessibilità dal punto di vista computazionale: nonostante la seconda fase usi l'output della prima, è comunque parametrica rispetto alla direzione solare, ed il sole può quindi essere spostato senza

ricalcolare le LUT. Durante il giorno la variazione del colore nel cielo nel tempo che passa tra 1 fotogramma e l'altro è trascurabile e questo permette di ridurre il numero di volte che viene eseguito il primo passo, che è quello computazionalmente più pesante.

3.6 Map Editor

Il motore grafico da solo è solo un framework che mette a disposizione strumenti per facilitare lo sviluppo di applicazioni grafiche. In M90 è inclusa anche un applicazione chiamata *Map Editor* che permette la modifica di paesaggi costruiti con M90. Il framework supporta anche un file di configurazione in cui può essere salvata un istanza di paesaggio; dal map editor è possibile aprire, visualizzare, modificare questi file o crerne di nuovi.

3.6.1 Pattern di rendering su Windows.Forms

L'applicazione è costituita da una windows Form, all'interno della quale sono presenti controlli per la modifica del paesaggio, che viene visualizzato in un riquadro al cerntro della finestra. Per ottenere un rendering fluido occorrerebbe chiamare il metodo *RenderFrame()* del motore grafico in un loop, ma questo bloccherebbe l'interfaccia grafica, impedendo l'interazione. Una prima soluzione implementata consisteva nell'inserire una temporizzazione all'interno del loop per permettere alla finestra di processare i propri messaggi, ma questo approccio aveva il difetto di limitare il numero di frame renderizzabili in un secondo. É quindi stato implementato il seguente metodo[1]:

```
/// <summary>
/// Rendering loop.
/// </summary>
private void Application_Idle(object sender, EventArgs e)
{
        if (engine.IsDisposed) return;//prevent looping when engine is not
           ready.

        while (Win32.IsApplicationInIdle())
        {
                engine.RenderFrame();
        }
}
```

L'evento *Application_Idle* viene chiamato quando la coda di messaggi della finestra è vuota e questa non sta processando nessun evento. Qui viene inserito il loop di rendering attraverso un while che continua a ciclare fino a che non arrivano altri messaggi di windows. Se tutti i messaggi vengono processati il metodo viene richiamato e prosegue il rendering. Questo approccio lascia inalterata la reattività della GUI gestendo i messaggi della finestra come prioritari rispetto al rendering dei fotogrammi e sfrutta efficacemente tutti gli intervalli di idle per chiamare *RenderFrame()*.

3.6.2 Utilizzo degli eventi

I fotogrammi completi visualizzati dall'editor non sono solo costituiti dagli elementi disegnati dal motore grafico, ma vengono sfruttati gli eventi di rendering di quest'ultimo per aggiungere elementi alla scena (vedi Sez. 3.2.5).

- L'evento *GraphicEngine.Render* è gestito dall'editor per aggiungere alla pileline il rendering di un cursore: questo viene proiettato nella scena in 3D e accoppiato con il movimento del mouse.
- L'evento *GraphicEngine.PostProcess* viene sfruttato per inserire sul fotogramma finale del testo informativo (es. numero di fotogrammi al secondo).

3.6.3 Funzionalità di editing

L'etitor mette a disposizione diverse funzionalità per personalizzare i vari aspetti del paesaggio:

- **Textures del terreno:** E' possibile personalizzare le texture per il terreno, scegliendo da un insieme di texture predefinite, incluse nell'editor. É consentito il caricamento di un massimo di 4 differenti texture, che possono essere quindi personalizzate per creare infiniti materiali modificandone la tonalità, assegnando filtri di applicazione per range di altezza o pendenza. I materiali vengono applicati sul terreno attraverso un pennello, anche questo personalizzabile per dimensione e durezza.
- **Conformazione del terreno:** Anche la conformazione del terreno viene modificata attraverso un pennello, che può essere utilizzato per creare vari effetti: colline, laghi, montagne, simulare erosione, spianare o livellare il terreno.
- **Oceano:** Molti parametri dell'oceano possono essere personalizzati, tra cui il livello dell'acqua, la dimensione e conformazione delle onde, parametri di riflesso e rifrazione ecc.
- **Atmosfera:** Dalla scheda sull'atmosfera è possibile modificare l'orario (e quindi la posizione del sole), la quantità di nebbia e la sua distribuzione.
- **Qualità grafica:** Ogni oggetto disegnabile espone una proprietà dalla quale è possibile impostare la sua qualità grafica. Da un'apposita scheda dell'editor si possono getire le qualità grafiche di tutti gli oggetti disegnati.

Capitolo 4

Requisiti

È richiesto lo sviluppo un modulo aggiuntivo per il motore grafico M90 che permetta a questo di visualizzare un oceano foto-realistico all'interno dei paesaggi.

Questo motore grafico è stato sviluppato per applicazioni videoludiche quindi il modello per il rendering dell'oceano dovrà presentare tre caratteristiche fondamentali:

Requisito 1 (Real-Time)**.** *L'algoritmo sarà definito real-time se riuscirà a produrre almeno 30 fotogrammi per secondo all'interno del motore grafico.*

Requisito 2 (Fedeltà visiva)**.** *La fedeltà visiva del rendering dovrà essere massimizzata, rispettando il primo vincolo del Real-time.*

Requisito 3 (Correttezza fisica)**.** *Il modello per il rendering dell'oceano dovrà essere corretto dal punto di fista fisico. Vista l'applicazione, questa non sarà una caratteristica fondamentale, ma va intesa come un mezzo per raggiungere la Fedeltà visiva.*

Lo sviluppo del modulo dovrà comprendere inoltre la trattazione dei seguenti problemi:

Requisito 4 (Modello geometrico)**.** *Dovrà essere affrontata la problematica della scelta della giusta mesh, ossia della struttura geometrica utilizzata per l'oceano. Poiché il punto di vista all'interno del motore grafico potrà essere mosso a piacere, la mesh dovrebbe avere idealmente un'estensione infinita in modo che l'orizzonte rimanga fisso e che non sia possibile raggiungerne una fine.*

Requisito 5 (Animazione)**.** *La superficie dell'acqua dovrà essere animata per produrre onde ed increspature realistiche.*

Durante lo sviluppo dovranno essere tenute in considerazione le immagini di riferimento in Figura 4.1.

(a) (b)

(c) (d)

(e) (f)

Figura 4.1: Immagini di riferimento per l'oceano. In 4.1(a), 4.1(b) e 4.1(c) è visibile la variazione di colore con il diminuire della profondità vicino alle rive. 4.1(d) ed 4.1(e) sono viste subacquee, Mentre in 4.1(f) è visibile la conformazione delle onde oceaniche.

Capitolo 5

Analisi del problema

Nella realizzazione del modulo per l'oceano, si possono subito evidenziare le due problematiche principali:

- Realizzazione della **struttura** del un modulo ed eventuali modifiche al quella del motore grafico.

- Progettazione di un modello per l'oceano. Questo definirà il **comportamento** dell'algoritmo di rendering.

5.1 Struttura del modulo per l'oceano

Per rappresentare le relazioni tra i vari strumenti utilizzati e trovare una collocazione per il modulo è necessario partire dal diagramma dei layer dell'intero sistema mostrato in Figura 5.1.

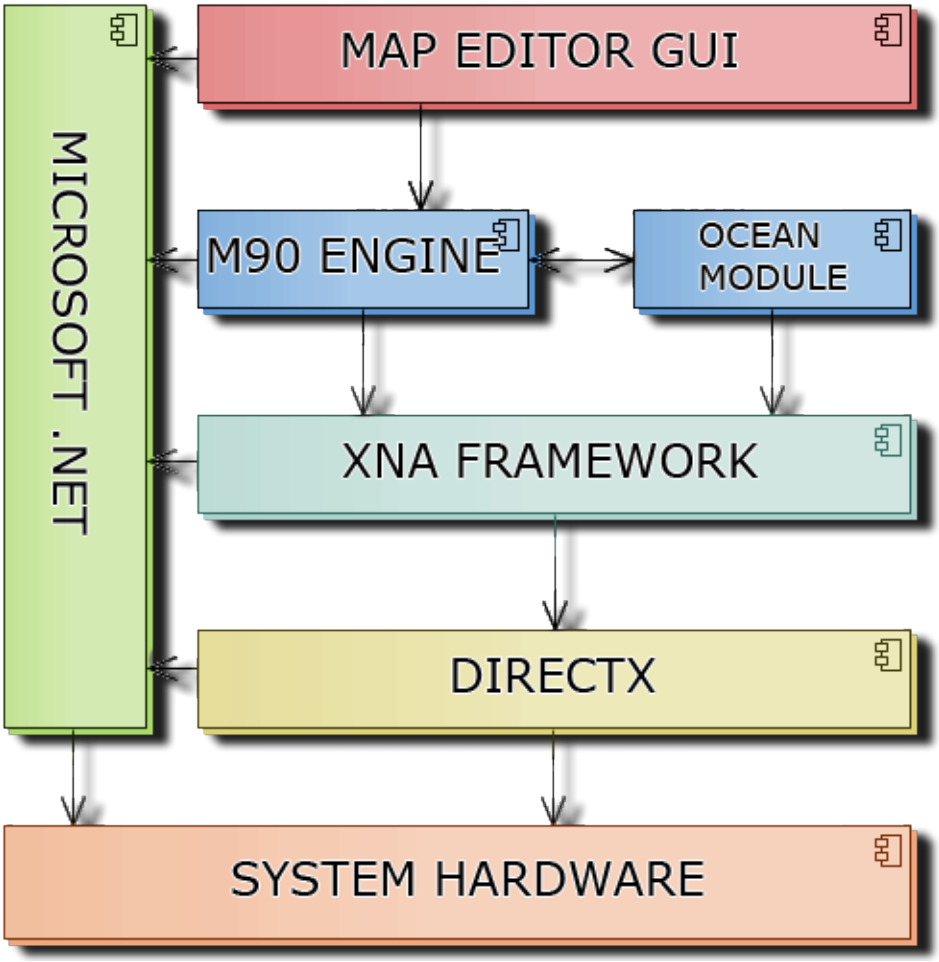


Figura 5.1: *architettura a layer del sistema, che mostra il posizionamento del framework M90 e delle varie tecnogie utilizzate.*

Le frecce rappresentano le dipendenze dei vari blocchi, il modulo da realizzare può essere collocato allo stesso livello del motore grafico: questo sarà realizzato sul framework XNA e dovrà in qualche modo connettersi con il motore grafico. Come si vede dal grafico non sarà presente una dipendenza diretta dell'editor dal modulo, perchè questo continuerà ad utilizzare l'interfaccia di M90 che rimane inalterata. È necessario quindi analizzare la struttura esistente del motore grafico, per capire come realizzare quella del modulo.

I moduli di M90 da analizzare sono:

- **M90.Engine** che dovrà essere modificato per permettere ad M90 di accedere

al modulo sull'oceano e di aggiungerlo alla scena.

- **M90.Resources** che contiene tutti gli shader del framework. Se verranno prodotti nuovi shaders per l'oceano, dovranno essere aggiunti a questo modulo.

La struttura del modulo *M90.Engine* ricavata dai sorgenti è mostrata in Figura 5.2.

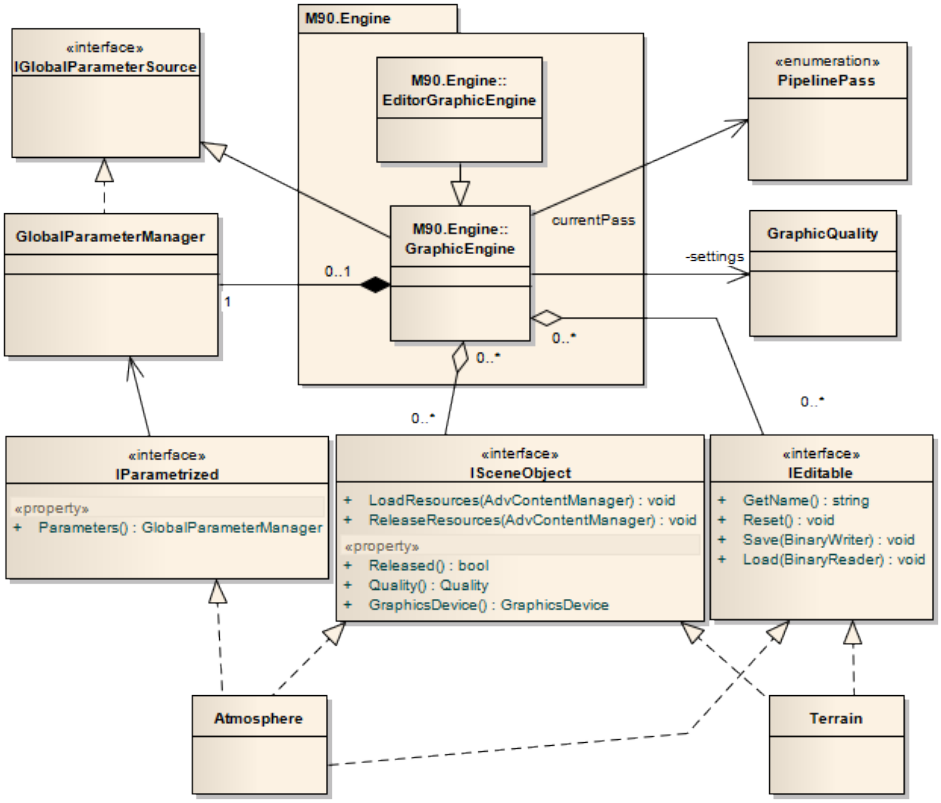


Figura 5.2: *Diagramma di struttura del modulo M90.Engine. È possibile notare come GraphicEngine utilizza i moduli disegnabili Atmosphere, Terrain.*

Per capire come posizionare il modulo sull'oceano occorre focalizzare l'attenzione su moduli che hanno un comportamento simile: *Atmosphere* e *Terrain*. Questi infatti come l'oceano sono disegnabili e fanno parte della scena. Implementano diverse interfacce per interagire con la classe *GraphicEngine* che rappresenta il core del framework(vedi Sez. 3.2) ed il loro compito può essere estratto dai commenti al codice e dal tipo di interazione che permettono:

- **IParametrized** La realizzazione di questa interfaccia permetta alla classe di utilizzare i parametri globali del framework e di accedere quindi a valori messi a disposizione da altri moduli(vedi Sez. 3.3).
- **ISceneObject** Identifica un oggetto disegnabile nella scena e *GraphicEngine*(vedi Sez. 3.2) ne gestirà il caricamento e rispettivo rilascio di risorse, la qualità grafica ed il device utilizzato per il rendering.
- **IEditable** Se un oggetto implementa questa interfaccia, dichiara che il suo stato può essere modificato dall'esterno, e permette a *GraphicEngine* di salvarne i cambiamenti su file, per poi ripristinarli in un secondo momento. Questo è il meccanismo utilizzato per il salvataggio e caricamento dei paesaggi su file.

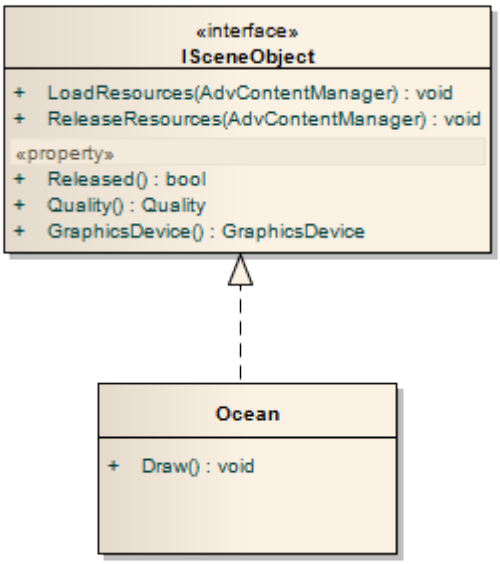


Figura 5.3: *Primo posizionamento del modulo rispetto ad M90.*

In un primo momento il modulo verrà rappresentato come un unica entità *Ocean* che realizza l'interfaccia *ISceneObject* come in Figura 5.3. Questa infatti sarà sicuramente presente, visto che l'oceano sarà disegnato nella scena. Per quanto riguarda le interfacce *IEditable* e *IParametrized*, la loro utilità sarà nota solo dopo aver delineato completamente il comportamento del modulo, per ora non introduciamo quindi riferimenti a queste.

Nelle interfacce presentate finora non sono presenti meccanismi per permettere il disegno di questo modulo, inseriamo quindi anche una nuova operazione *Draw()*. Il fatto che questa operazione non sia stata integrata in *ISceneObject*, significa che

per fare in modo che il motore grafico disegni l'oceano, i comportamenti di alcune operazioni di *GraphicEngine* dovranno essere modificati, sarà quindi necessario tenere conto di questo aspetto durante l'analisi del comportamento(vedi Sez. 5.2) e dell'interazione(vedi Sez. 5.3) di questa sezione.

5.1.1 Posizionamento degli shader in M90

Occorre definire ora dove posizionare i nuovi shader che verranno creati per l'oceano. Questi potrebbero essere semplicemente aggiunti al modulo stesso, ma esistono in M90 meccanismi per sfruttare parametri globali condivisi tra tutti gli shader del motore grafico: per utilizzarli, i nuovi shader devono essere aggiunti al modulo *M90.Resources* dove potranno includere anche altri shader che rappresentano raccoglitori di parametri globali. Il riferimento agli shader creati sarà quindi ottenibile durante l'eseguzione del metodo di interfaccia *ISceneObject.LoadResources*, dove viene passato come parametro un oggetto che si occupa proprio di caricare risorse esterne localizzate in *M90.Resources*.

5.2 Algoritmo per il rendering dell'acqua

Per capire come produrre un algoritmo foto-realistico per la visualizzazione dell'acqua è necessario conoscere la fisica dietro il comportamento della luce nell'acqua. L'immagine che noi abbiamo dell'acqua è ovviamente data dalla luce che viene diffusa, rifratta o riflessa da questa verso di noi, quindi sarà sufficiente tracciare un modello di interazione della luce esterna con l'acqua per poterne simulare l'aspetto.

Per costruire un modello, esaminiamo il caso generale di una superficie opaca in uno spazio 3d rappresentante il fondale di equazione:

$$y - 1 = 0$$

coperta da un volume d'acqua che si estende infinitamente lungo gli assi $\hat{x}, \hat{z}$ e limitata superiormente (considerando $\hat{y}$ come asse verticale) dal piano:

$$y - y_w = 0$$

dove y_w rappresenta la profondità dell'acqua. Fissiamo anche un versore per rappresentare una sorgente di luce direzionale $\hat{S_d}$ (Es. sole), punto di vista C_{pos} ed una direzione costante di vista $\hat{C_{dir}}$. L'intensità della luce è rappresentata attraverso il modello RGB con valori $\mathbb{R}^3$, esprimeremo quindi l'intensità della luce solare con il simbolo I_s. Tutte le variabili introdotte sono rappresentate in Figura 5.4.

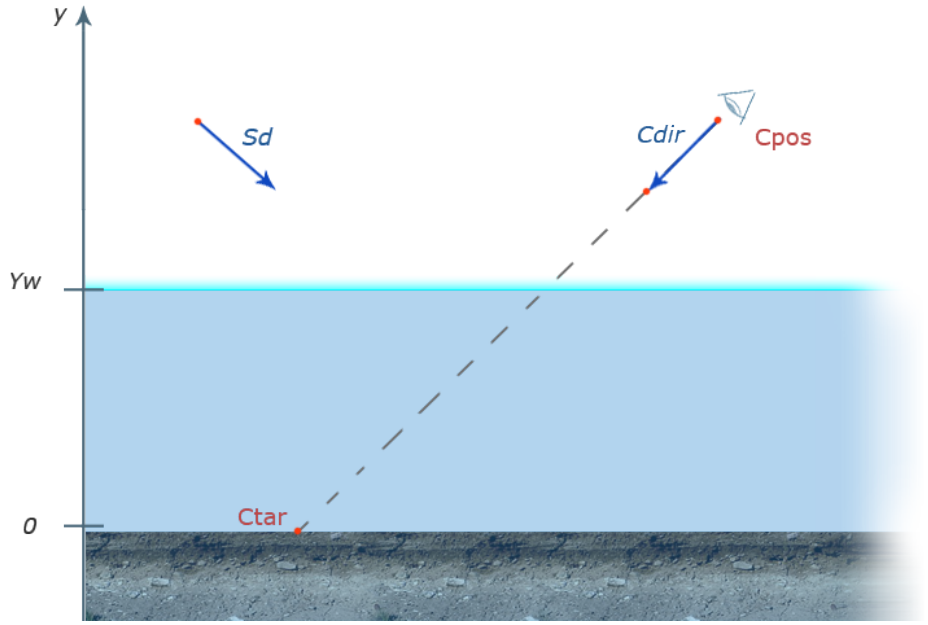


Figura 5.4: *Posizionmento delle variabili introdotte.*

Se l'acqua non fosse presente, la luce in C_{pos} proveniente dalla direzione $\hat{C_{dir}}$ sarebbe semplicemente quella diffusa nel punto C_{tar} calcolabile come:

$$I_s \cdot C_f$$

dove C_f rappresenta il colore RGB del fondale. In realtà sono da tenere in considerazione alcuni fenomeni di interazione della luce con l'acqua che dovranno essere simulati: *riflesso, rifrazione, assorbimento elettromagnetico, diffusione volumetrica.*

5.2.1 Riflesso e rifrazione

Questi sono fenomeni che avvengono quando la luce attraversa una superficie che separa due mezzi con indice di rifrazione diversi. Quando questo avviene, parte della luce non riesce ad attraversare la superficie e viene *riflessa*, parte la attraversa, ma la sua direzione viene distorta (*Rifrazione*).

Esistono due leggi che regolano questi fenomeni: Le *leggi di Fresnel* permettono di calcolare quale percentuale della luce viene riflessa, in funzione dell'angolo tra la direzione della luce e la normale della superficie, mentre la *Legge di Snell* descrive come viene distorta la direzione della luce che viene rifratta.

5.2.2 Leggi di Fresnel

Dati gli indici di rifrazione dei due mezzi n_1, n_2 e la normale della superficie tra i due $\hat{n}$ consideriamo gli angoli θ_i e θ_t che la luce forma con la normale rappresentati in Figura 5.5.

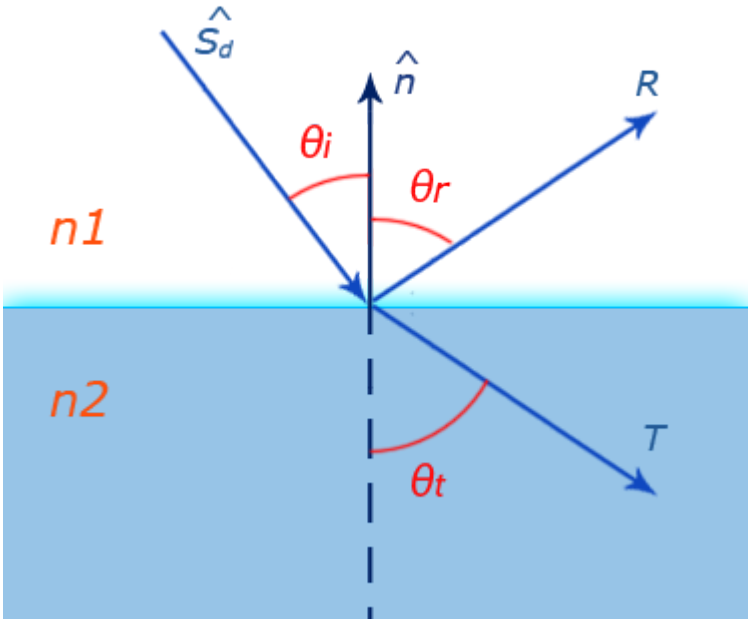


Figura 5.5: *Riflesso e rifrazione della luce. Il vettore R rappresenta la direzione del riflesso mentre T quella della rifrazione.*

La percentuale della luce riflessa dalla superficie è data da[11]:

$$R_s = \left[\frac{\sin(\theta_t - \theta_i)}{\sin(\theta_t + \theta_i)}\right]$$

per la luce con polarizzazione S e da:

$$R_p = \left[\frac{\tan(\theta_t - \theta_i)}{\tan(\theta_t + \theta_i)}\right]$$

per la luce con polarizzazione P. Nel nostro caso considereremo luce non polarizzata, per la quale il coefficient di riflessione vale:

$$R = \frac{R_s + R_p}{2}$$

5.2.3 Legge di Snell

La legge di Snell esprime la relazione tra θ_i e θ_t[13]

$$n_1 \sin(\theta_i) = n_2 \sin(\theta_t)$$

5.2.4 Assorbimento elettromagnetico

Mentre la luce attraversa l'oceano, parte dello spettro luminoso viene assorbito dalla molecola d'acqua a causa delle sue *vibrazioni molecolari*[10]. L'assorbimento è più forte nelle basse frequenze, mentre le lunghezze d'onda più corte vengono lasciate inalterate (vedi Figura 5.6). L'acqua si comporta quindi come un filtro, e la luce che l'attraversa assume un colore blu, tanto più profondo quanto è maggiore la distanza percorsa da questa.

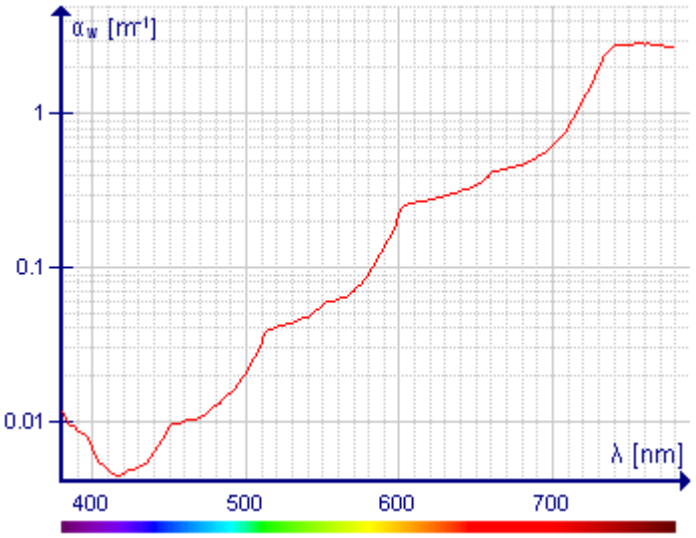


Figura 5.6: *Coefficiente di attenuazione della luce in relazione alla lunghezza d'onda. Come si può notare il blu ha un basso coefficiente, per questo motivo l'acqua assume una colorazione blu.*

5.2.5 Diffusione volumetrica

L'acqua presente nell'oceano non è pura e sono presenti particelle disciolte in essa che influiscono sulla sua colorazione: quando sono colpite dalla luce la diffondono intorno a loro aumentando l'intensità del suo colore, che altrimenti dipenderebbe solo da quello diffuso dal fondale.

5.2.6 Interazioni di superficie e volumetriche

Sono stati evidenziati finora quattro tipi di interazione della luce con l'acqua, ma questi possono essere classificati in due distinte categorie: *riflessi e rifrazioni* si presentano solo in corrispondenza della superficie, questa sarà quindi modellabile in uno spazio 3d ed *Ocean.Draw()* introdotto nella struttura del modulo si occuperà di disegnare quindi la superficie dell'oceano. Per quanto riguarda l'*assorbimento elettromagnetico* e la *diffusione volumetrica*, sono effetti che interferiranno con il colore di qualsiasi oggetto si venga a trovare dietro il volume d'acqua che costituisce l'oceano, rispetto al punto di vista. Questo significa che *i comportamenti volumetrici sviluppati dovranno essere aggiunti al comportamento di ogni oggetto della scena*, compresa la superficie dell'oceano stesso.

Le interazioni di superficie sono quindi incapsulate nello shader *OceanSurfaceShader* utilizzato durante l'operazione di rendering della superficie dell'oceano. Questo utilizzerà le formule di fresnel e di snell per combinare rifrazione e riflesso. *OtherSceneObject* rappresenta tutti gli altri ISceneObject del motore grafico, i cui shaders dovranno includere *OceanVolumeShader* che simulerà le interazioni volumetriche con l'oceano. L'entità *Shader* rappresenta una generalizzazione per tutti gli shader ed implementa l'operazione *Shader.UseShader()* che incapsula il concetto di utilizzabilità di questi per il rendering di un oggetto. Per una rappresentazione grafica della nuova struttura delineata fare riferimento alla Figura 5.7.

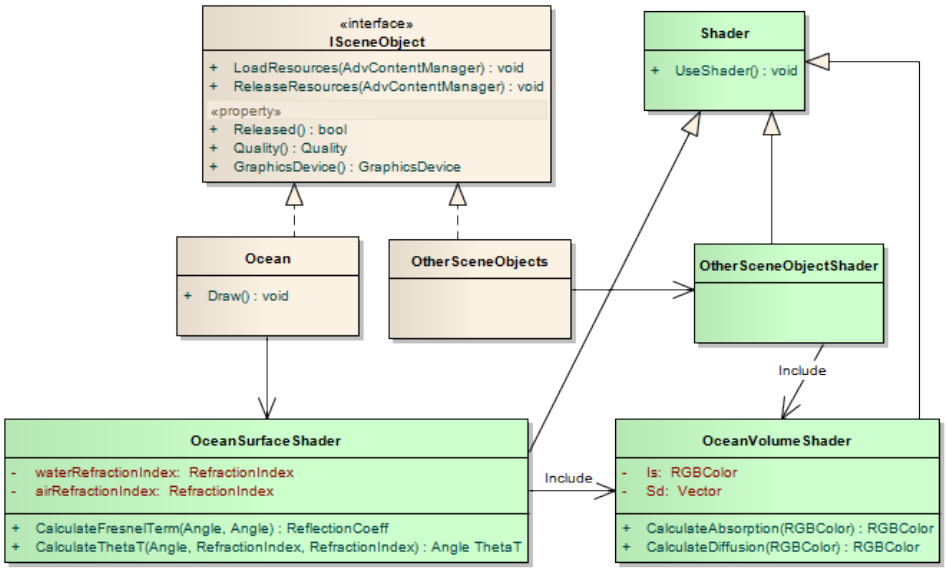


Figura 5.7: *Struttura del modulo delineata fin'ora con rappresentato l'utilizzo dei calcoli volumetrici per l'oceano da parte di altri generici ISceneObject. Gli shader sono rappresentati in verde.*

5.3 Interazione

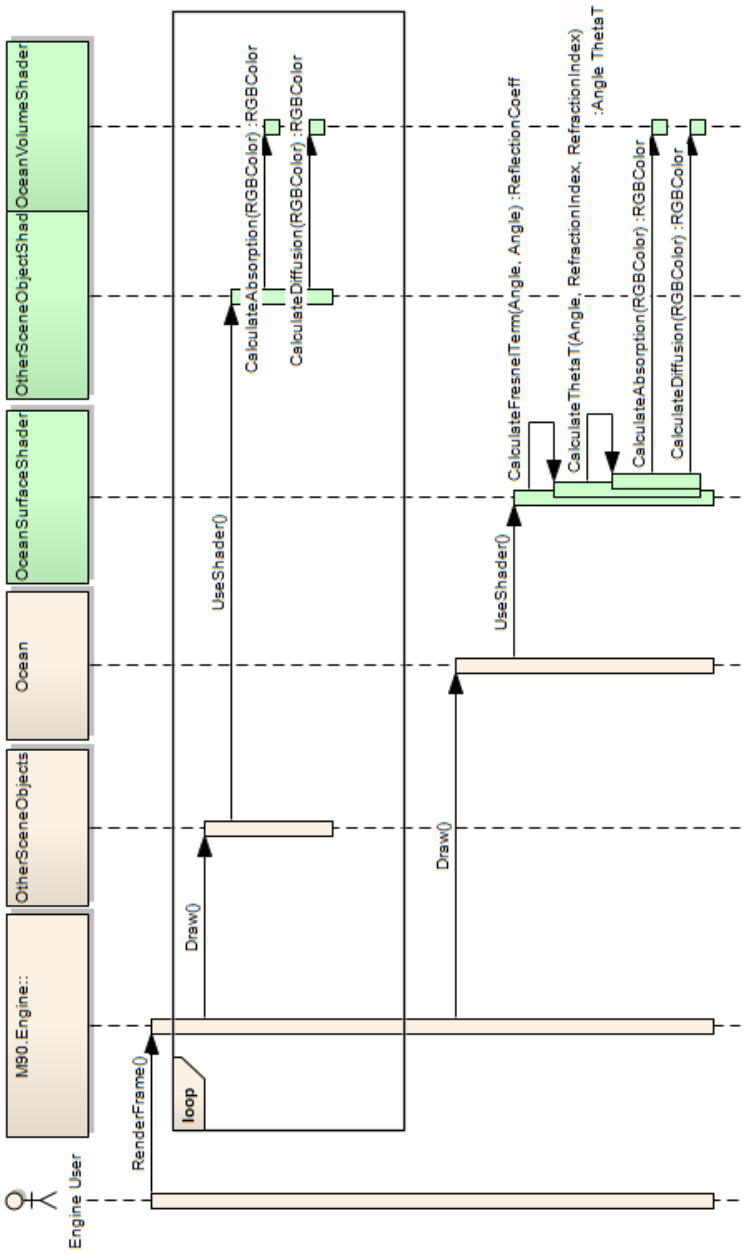


Figura 5.8: *Sequenza di interazioni delle entità viste, durante del rendering di un frame.*

Capitolo 6

Progettazione

Ora che è stato delineato un quadro generale delle interazioni della luce con l'acqua, possiamo approfondire il comportamento delle singole entità viste in Figura 5.7.

6.1 Superficie dell'acqua

Ocean insieme ad *OceanSurfaceShader* si occupano del disegno della superficie. La progettazione di questa verrà scomposta in tre parti, dipendenti tra loro nell'ordine: *Struttura geometrica*, *Animazione*, *Shading*. La prima parte si occuperà di sviluppare un modello geometrico per la superficie dell'acqua, nella seconda parte verrà trattata l'animazione di questo modello, mentre nell'ultima parte ne verrà affrontato lo shading, con l'utilizzo delle interazioni di superfice viste nell'analisi.

6.1.1 Struttura geometrica

Prima di procedere alla progettazione della struttura geometrica per la superficie dell'acqua è necessario ricordare i requisiti principali riguardanti la superficie ed effettuare alcune considerazioni derivanti da questi:

1. La mesh dovrà contenere meno geometria possibile, in modo che possa essere visualizzata in real-time Requisito 1.
2. Poichè il punto di vista potrà essere mosso a piacere, la superficie dovrebbe avere idealmente un'estensione infinita in modo che l'orizzonte rimanga fisso e che non sia possibile raggiungerne una fine Requisito 4.
3. La superficie dovrà poi essere animata, è necessario quindi che presenti una densità di vertici sufficientemente elevata per produrre un'animazione realistica Requisito 2 e Requisito 5.

L'approccio iniziale utilizzato consiste nel costruire una griglia di vertici della stessa dimensione del terreno, in modo che questo possa essere coperto interamente[2].

Questi sono quindi connessi 4 a 4 per formare due facce come mostrato in Figura 6.1.

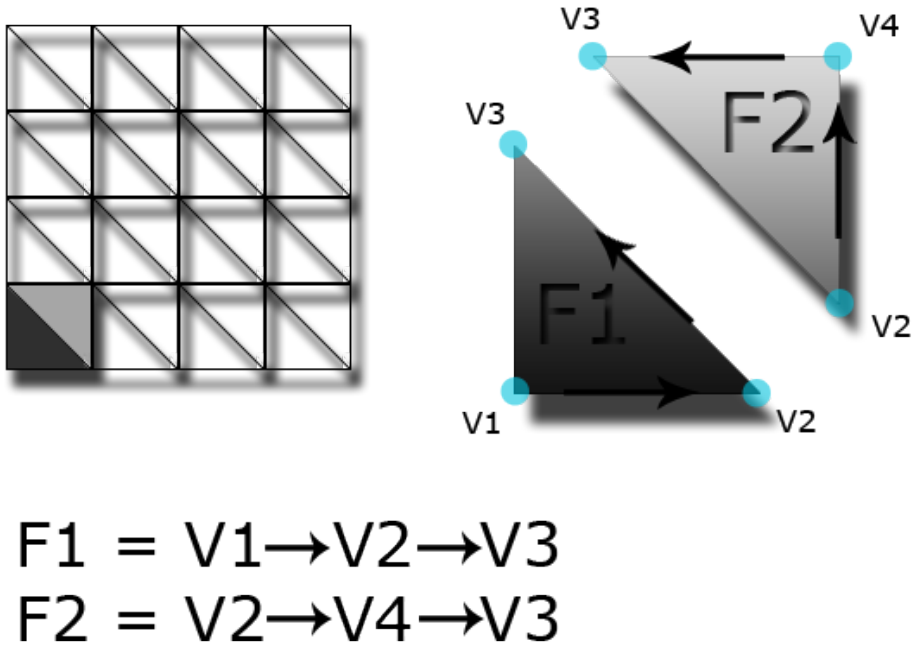


Figura 6.1: *Struttura geometrica che utilizza una griglia di vertici per disegnare una superficie.*

Con questo meccanismo la densità dei vertici è uniforme ma perché possa essere estesa su tutto il terreno consentendo un'animazione corretta, occorrono un numero vertici circa pari a quelli del terreno a massima risoluzione. Considerando che il motore grafico supporta zone di terreno quadrate costituite da 8193 x 8193 vertici, questo numero è troppo elevato e non permette di rispettare il Requisito 1 del real-time. Nemmeno il Requisito 4 è soddisfatto poiché la mesh per la superficie dell'acqua è di dimensioni molto limitate, ed è semplice porsi in una posizione tale che sia visibile il suo bordo. Questo approccio rende quindi impossibile l'illusione di un orizzonte illimitato.

Una tecnica da prendere in considerazione per ridurre il numero di vertici è di applicare alla griglia un algoritmo LOD (Level of detail) simile a quello utilizzato per il terreno. Con questa tecnica è possibile soddisfare il primo requisito, riducendo drasticamente i vertici utilizzati, ma presenta diversi probliemi per quanto riguarda il requisito 3: i blocchi di superficie vicini al proprio punto di vista, risulterebbero quelli a massima risoluzione, e verrebbero animati correttamente, ma

considerando blocchi sempre più lontani, questi finirebbero per avere una risoluzione decisamente ridotta, che non riuscirebbe a supportare il maniera adeguata l'animazione e verrebbero prodotti artefatti. Inoltre neanche utilizzando LOD si potrebbe disegnare una superficie con un'estensione sufficiente a soddisfare il secondo requisito.

La tecnica[5] che si è deciso di adottare nell'implementazione finale pone un limite al Requisito 5 garantendo l'animazione solo di una porzione della superficie totale, quella più vicina al punto di vista. Viene utilizzata di base una griglia di vertici simile a quella del primo approccio mostrato, ma questa invece di essere fissa e coprire l'intero terreno, è mobile e sempre centrata nel punto di vista: in questo modo per quanto possa muoversi l'utente all'interno dell'ambiente, rimarrà sempre nel punto più distante dal bordo, evitando così che riesca a vedere la *"fine* dell'acqua. Per quanto riguarda il Requisito 1, questo viene soddisfatto utilizzando una griglia di piccole dimensioni, che possa essere visualizzata in real-time. Il Requisito 4 viene soddisfatto traslando i vertici che costituiscono il bordo della mesh ad un'elevata distanza dal punto di vista (vedi Figura 6.2), questa distanza è stata scelta uguale a quella massima rappresentabile dal motore grafico e questo, combinato al fatto che la mesh segue la propria posizione, crea l'illusione di un orizzonte illimitato.

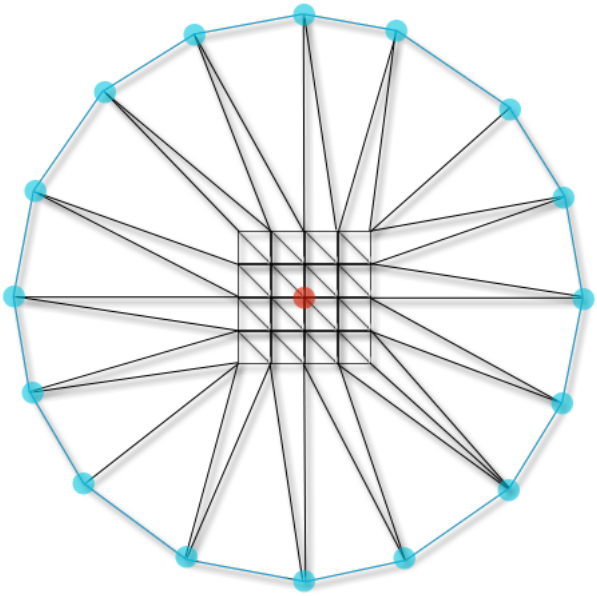


Figura 6.2: *Il punto rosso rappresenta il punto di vista. I vertici contrassegnati in blu sono stati traslati in modo da aumentare la distanza dal punto di vista. L'unica parte della mesh con sufficiente tessellazione per supportare l'animazione è quella centrale.*

Per quando riguarda l'animazione, questa è limitata alla parte centrale della mesh e viene attenuata mano a mano che ci si sposta verso il suo bordo; è essenziale che venga annullata prima di giungere ai vertici traslati altrimenti verrebbero prodotti forti artefatti dovuti alla bassa tessellazione. Questo punto verrà chiarito meglio nella parte dedicata all'animazione (vedi Sez. 6.1.2). Rimane solo un perfezionamento da effettuare e riguarda il posizionamento della mesh: essendo questa essenzialmente (a meno di estensione) un quadrato, ponendosi nel suo centro, almeno la metà dei vertici sono nascosti dietro le nostre spalle e vanno sprecati. Per correggere questo comportamento si effettuano due modifiche :

- Si ruota la mesh rispetto al punto di vista in modo che un angolo sia orientato nella direzione osservata, in questo modo la maggior parte dei vertici si trovano al centro dell'immagine, dove hanno visivamente più impatto.

- Si effettua una traslazione della mesh in modo da portarne l'angolo opposto a quello osservato, nel punto di vista : in questo modo sono sempre visibili la maggior parte dei vertici. Nell'implementazione effettiva, questa traslazione viene leggermente attenuata, questo evita la mancanza di vertici nelle bande laterali a causa di un angolo di visione superiore a 90 gradi.

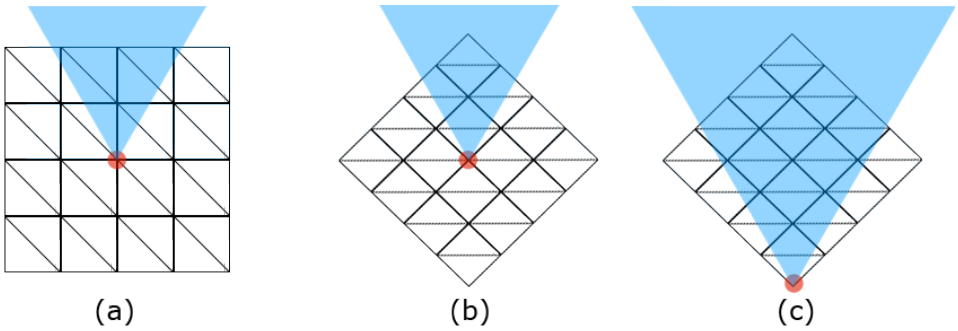


Figura 6.3: *Riposizionamento della mesh rispetto al punto di vista. Il punto rosso rappresenta il punto di vista, il triangolo blu il cono di visione dal punto di vista. In (a) è rappresentata la condizione iniziale, (b) e (c) le due modifiche effettuate: si può notare la crescita del numero di vertici visibili (coperti dal triangolo blu).*

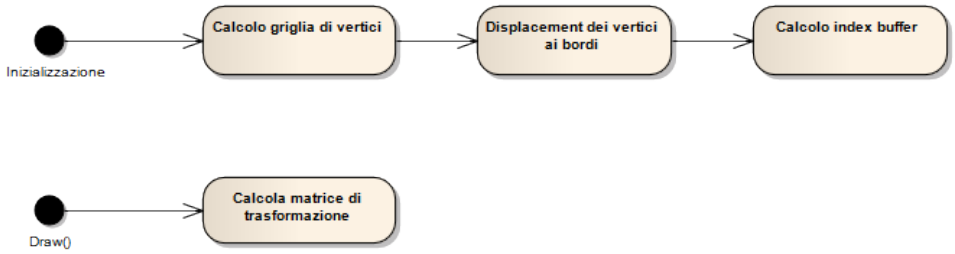


Figura 6.4: *Sequenza delle operazioni da eseguire per il modello geometrico durante inizializzazione e disegno.*

6.1.2 Animazione

L'animazione della superficie dell'oceano si baserà sull'articolo di Jerry Tessendorf[8] che fornisce un metodo statistico per la simulazione delle onde oceaniche. Come mostrato nella sezione precedente, la superficie dell'oceano è costituita semplicemente da una griglia di vertici, quindi animarla vuol dire generare degli offset per i vertici che ne modifichino la posizione nel tempo. L'insieme degli offset verrà quindi salvato in una matrice di vettori, che verranno aggiunti ai rispettivi vertici secondo la semplice equazione:

$$V_{pos} = V_{pos} + (V_{off} \cdot W_{scale}) \tag{6.1}$$

dove V_{pos} è la posizione del vertice, V_{off} il vettore rappresentante l'offset e W_{scale} un fattore di scala per modificare l'entità delle onde.

Ora rimane il problema di come generare una matrice di vettori dinamica per produrre onde realistiche. La tecnica scelta utilizza la IFFT 2D per creare una *heightmap* costituita dalla somma di varie sinusoidi, a partire da uno spettro generato attraverso un equazione creata appositamente per le onde oceaniche, e si sviluppa nei seguenti passi:

1. Generazione di uno spettro 2D complesso della stessa dimensione della matrice dinamica che si vuole ottenere, la dimensione della matrice determina la qualità dell'effetto perchè porta ad una matrice finale con maggiore risoluzione e quindi più dettagliata. La funzione utilizzata per calcolarlo è quella di *Phillips* esposta nell'articolo:

$$P_h(\vec{k}) = A\frac{e^{\frac{-1}{(kL)^2}}}{k^4}|\hat{k}\cdot\hat{w}|^2 \tag{6.2}$$

   Dove $\vec{k}$ è un vettore che rappresenta la direzione di un onda con la sua direzione e la frequenza con il suo modulo. Nel fattore finale $\hat{w}$ rappresenta la direzione del vento ed il prodotto scalare serve ad annullare le onde che si muovono perpendicolarmente alla direzione del vento. Per ottenere lo spettro finale, alla funzione di *Phillips* viene moltiplicato uno spettro casuale con distribuzione gaussiana.

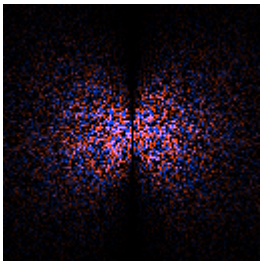


Figura 6.5: *Spettro finale ottenuto con Phillips ed una matrice 128x128.Il blu rappresenta la componente reale mentre l'arancione quella complessa. Come si può notare dall'immagine le onde si attenuano mano a mano che aumenta la loro frequenza(verso i bordi) e se sono perpendicolari al vento. In questa simulazione il vento ha direzione orizzontale.*

2. Per ogni fotogramma:

   (a) **Calcolo delle fasi** dello spettro utilizzando un valore di tempo progressivo t. Questo viene sommato alle sinusoidi complesse rappresentate da $\vec{k}$ attraverso la formula:

$$\vec{k_1} = \vec{k} * e^{i\cdot t} \tag{6.3}$$

Questa è la parte dove avviene l'animazione vera e propria, che si ottiene quindi aggiungendo alla fasi delle varie sinusoidi un offset che viene incrementato con il tempo.

(b) **Esecuzione della IFFT 2D** questa trasforma lo spettro ottenuto alla fase precedente, nella sua rappresentazione reale che è appunto la somma di tutte le sinusoidi e costituisce una *heightmap* che rappresenta la conformazione delle onde.

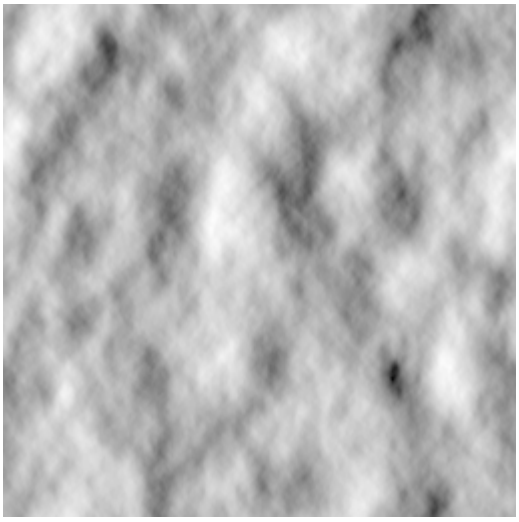


Figura 6.6: *HeightMap generata attraverso IFFT 2D su una matrice 512x512.*

(c) **Rendering** del modello geometrico con offset modificati come mostrato nell'equazione 6.1 dove V_{off} rappresenterà un vettore con tutte le componenti nulle tranne quella verticale, impostata al valore dato dalla *heightmap*.

Essendo la *heightmap* generata attraverso una IFFT, questa è costituita da una somma di sinusoidi. Una sinusoide è una buona approssimazione di un onda in uno specchio d'acqua calmo, ma le onde che si formano negli oceani a causa del vento presentano creste più nette che sono difficilmente generabili attraverso un offset verticale poichè questo causerebbe diversi artefatti dovuti all'eccessiva risoluzione del modello richiesta: infatti lo spettro di questo tipo di onde sarebbe caratterizzato da una maggior presenza di sinusoidi ad alta frequenza. Viene invece aggiunto all'offset originale un ulteriore spostamento sul piano parallelo a quello dell'acqua, che può essere facilmente generato a partire dall'output della IFFT sulla base della sua derivata, che crea dilatazioni nelle zone di minimo e compressioni in quelle di massimo, simulando appunto le creste delle onde oceaniche.

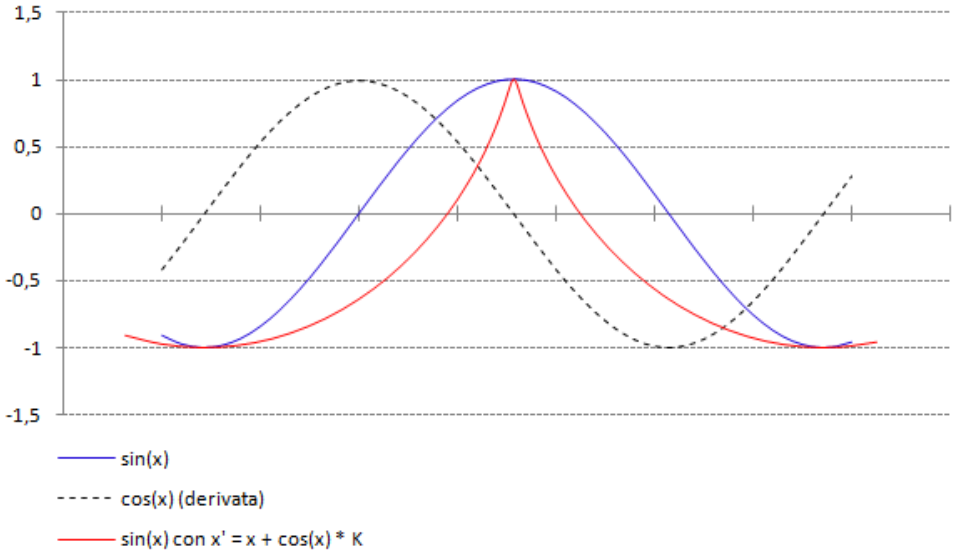


Figura 6.7: *Il grafico mostra come è possibile produrre onde oceaniche accurate a partire da una sinusoide. In blu è rappresentata la funzione sin(x). In nero tratteggiato è rappresentata cos(x) che è la derivata di sin(x). La forma d'onda in rosso è generata a partire dalle stesse ordinate di quella in blu, ma sommando alle ascisse la derivata (cos(x)) moltiplicata per un fattore di scala.*

Attraverso questa tecnica la risoluzione del modello geometrico viene modificata dinamicamente per concentrarla dove la derivata della forma d'onda è maggiore, riducendo il numero di vertici richiesti.

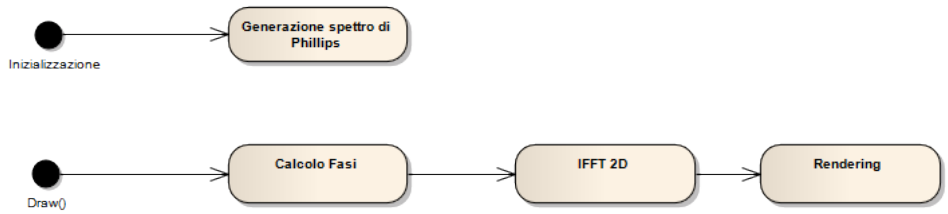


Figura 6.8: *Sequenza delle operazioni da eseguire per l'animazione del modello geometrico durante inizializzazione e disegno.*

6.1.3 Shading

Prima di procedere alla progettazione del modello di illuminazione per la superficie dell'acqua, è necessario definire come vengono ottenute alcune informazioni basilari che verranno poi utilizzate nel modello finale:

- **NomalMap** La normal map per la superficie dell'acqua è ottenuta dallo stesso procedimento utilizzato per l'animazione del modello (vedi Sez. 6.1.2), infatti dalla IFFT di output all'algoritmo, è facilmente ottenibile la normale approssimandola dalla heighmap come:

$$\hat{n_{i,j}} = (nx, ny, nz) = (h_{i,j} - h_{i+1,j}, s_{dist}, h_{i,j} - h_{i,j+1}) \tag{6.4}$$

  dove $h_{i,j}$ è il valore della heighmap nelle coordinate i, j mentre s_{dist} è la distanza tra i campioni della heighmap (passo spaziale).

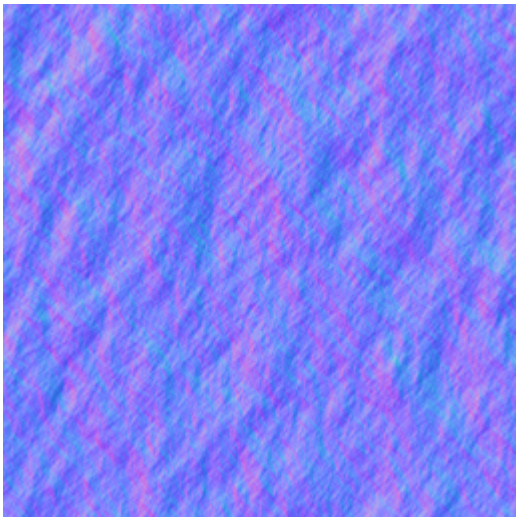


Figura 6.9: *Esempio di normalmap ricavata dalla heighmap calcolata dall'algoritmo: ogni pixel RGB rappresenta un vettore normale in cui in ogni canale è salvata una componente.*

- **Rifrazione** La superficie dell'acqua è in parte rifrattiva, cioè permette di vederci attraverso, Quindi occorrerà ottenere la porzione della scena presente dietro di essa. Un prima soluzione è quella di disegnare la superficie semitrasparente, in modo che non siano necessari dati sul colore rifratto in fase di shading, ma in questo modo verrebbe trascurato d'effetto di distorsione generato dalle increspature e il risultato sarebbe poco realistico. Si è quindi deciso di utilizzare la mappa del colore rifratto messa a disposizione dal motore grafico agli oggetti rifrattivi, che contiene semplicemente l'immagine della scena disegnata fin'ora(vedi Sez. 3.2.3).

- **Riflesso** I riflessi sono un altra caratteristica da tenere in considerazione nello shading della superficie, ma ottenerne una rappresentazione accurata in real-time è difficile, poiché occorrerebbe implementare un algoritmo di

ray-tracing sugli attuali hardware grafici che sono invece progettati per la rasterizzazione. Viene quindi impiegata un'approssimazione generata renderizzando nuovamente la scena a risoluzione ridotta, con un punto di vista ed una direzione specchiati rispetto alla superficie dell'acqua[2]. L'immagine generata verrà da ora riferita come reflex map.

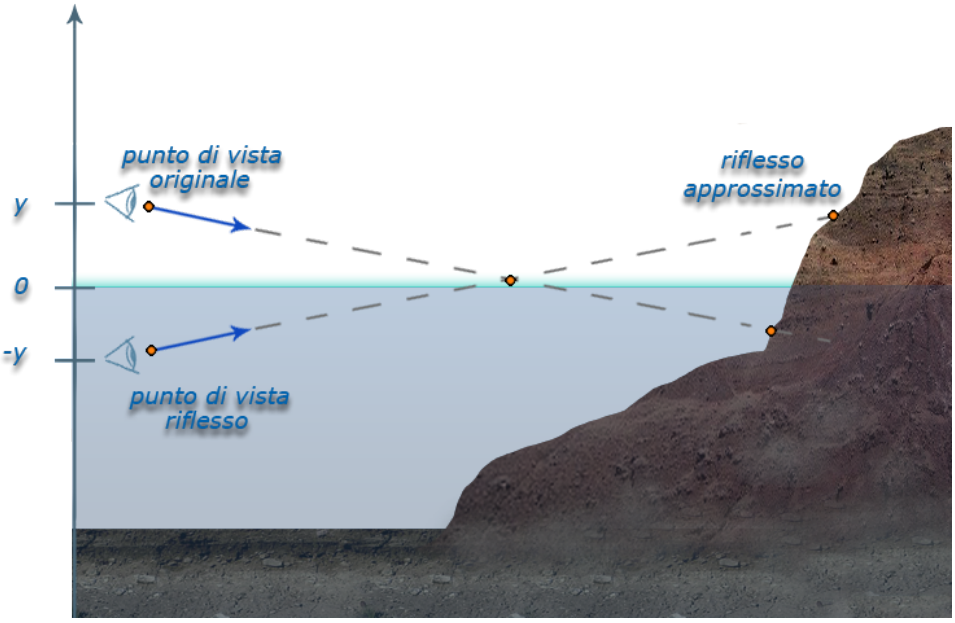


Figura 6.10: *Posizionamento del punto di vista specchiato. Seguendo la linea tratteggiata si può vedere come lo stesso campione nell'immagine originale, corrisponde al riflesso in quel punto nella reflex map.*

Lo shading della superficie dell'acqua avviene in un unico passo che si può scomporre nelle seguenti fasi:

1. Depth-test manuale, effettuato a partire da un depth buffer salvato come texture. Non è possibile effettuare un depth test automatico poichè a causa della deferred pipeline, il precedente depth buffer va perso (vedi Sez. 3.2.3). Il depth buffer è messo a disposizione dal motore grafico.

2. Campionamento del normal ottenuto attraverso IFFT.

3. Calcolo di un normal di dettaglio, creato campionando ed unendo con una media pesata quattro normal ottenuti dalla stessa texture statica con quattro set di coordinate generate dallo shader. Questi normal vengono animati applicando un offset dipendente dal tempo alle coordinate di campionamento e una distorsione dipendente dalla normal map IFFT.

4. Blending dei due normal in modo che nell'area animata siano prevalenti quelli generati dalla IFFT mentre nell'area a bassa tessellazione siano presenti solo gli altri. I primi infatti non sono adatti ad essere ripetuti fino all'orizzonte perché privi di *mipmap* e se vengono proiettati su una zona troppo piccola, il pattern di ripetizione sarebbe chiaro e verrebbero prodotti inoltre artefatti dovuti al sotto-campionamento della normal map. La sequenza delle operazioni per il blending dei due è schematizzata in Figura 6.11.

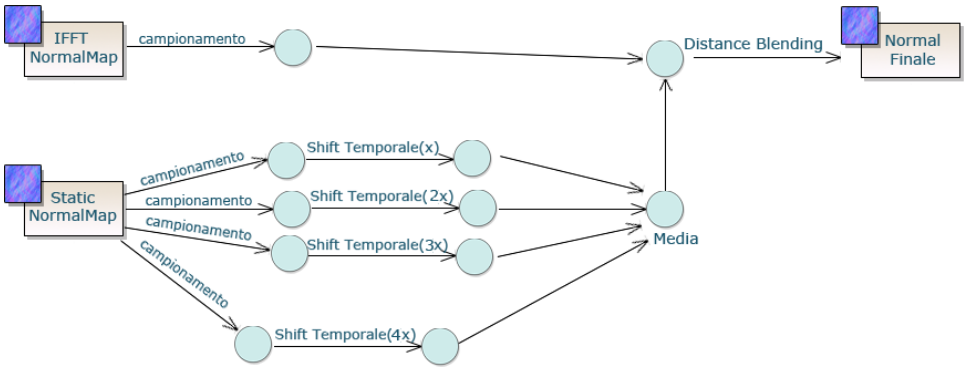


Figura 6.11: *Sequenza di blending dei normal. A sinistra le due normal map disponibili, le frecce indicano i campionamenti.*

5. Le mappe di rifrazione e riflesso vengono quindi campionate, introducendo una distorsione nelle coordinate in funzione della normale della superficie e delle leggi di Snell.

6. Calcolo del coefficiente di riflessione utilizzando la normale calcolata e le equazioni di Fresnel.

7. Il colore della superficie viene quindi calcolato interpolando il colore riflesso con quello rifratto attraverso il coefficiente di riflessione.

8. A questo colore vengono aggiunti i riflessi speculari: per una certa normale della superficie, la luce del sole diretta viene riflessa verso il punto di vista, creando delle zone particolarmente luminose. L'intensità di questo effetto viene simulata attraverso il prodotto scalare tra il vettore della direzione della luce solare riflessa secondo la normale della superficie, e la direzione del vettore che va dal punto della superficie al punto di vista.

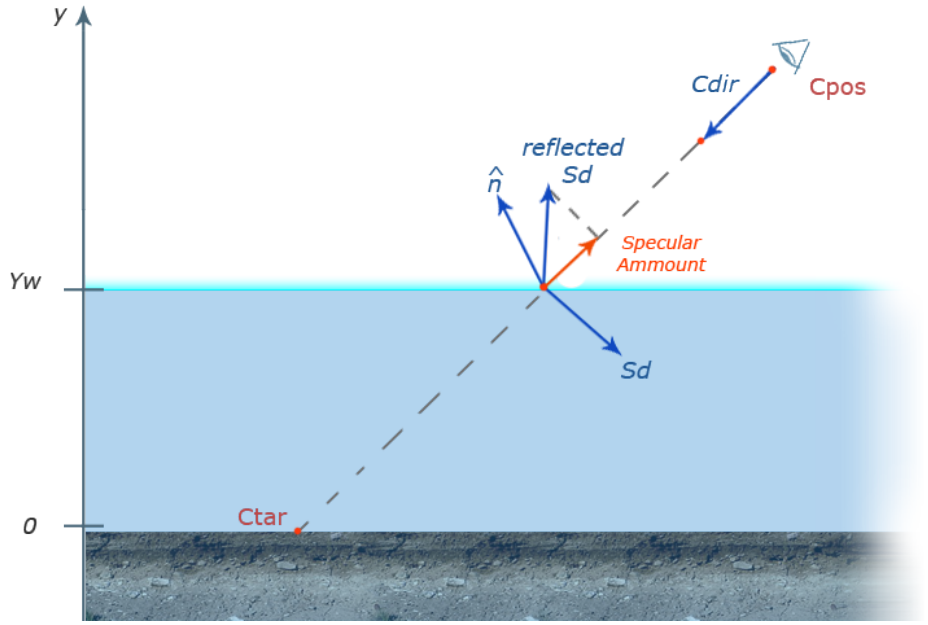


Figura 6.12: *Rappresentazione vettoriale del calcolo dell'intensità della luce diretta riflessa dal sole. Il vettore S_d rappresenta la direzione della luce solare.*

Una volta calcolata l'intensità di questo effetto attraverso il prodotto scalare che chiameremo I_{spec}, questo viene sommato al colore della superficie I_c attraverso l'equazione:

$$I_c = I_c I_{spec}^{P_{spec}} + S_{color}(1 - I_{spec}^{P_{spec}}) \tag{6.5}$$

dove S_{color} è il colore RGB rappresentante la luce solare e P_{spec} è una potenza che rende l'effetto più selettivo e viene di solito impostata a valori elevati (da 32 a 1024).

9. Gli effetti di diffusione atmosferica e nebbia vengono quindi applicati alla superficie per ottenere il colore finale, e come già anticipato nell'analisi, anche alla superficie andranno applicati gli effetti volumetrici dell'oceano.

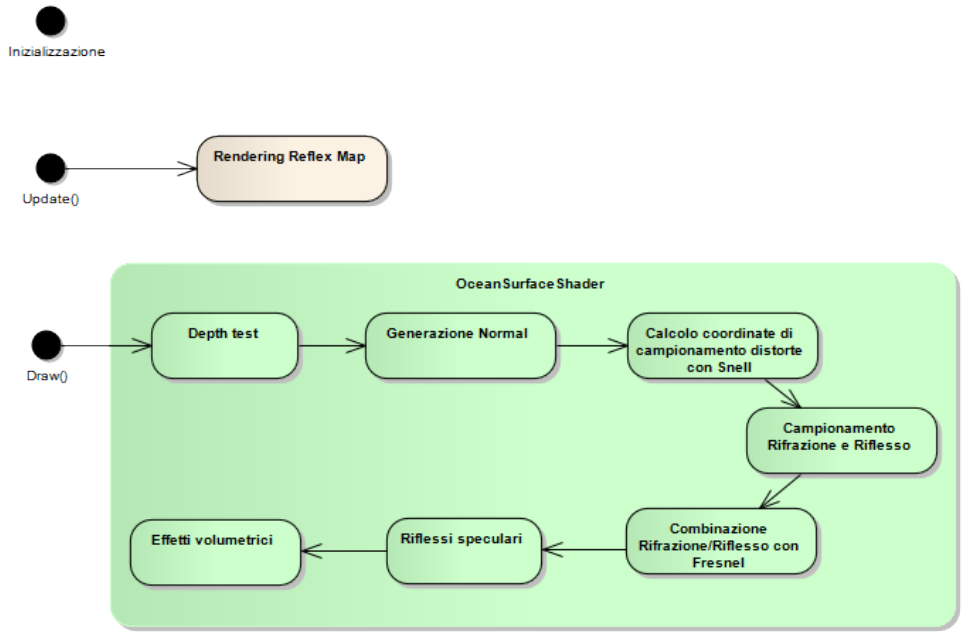


Figura 6.13: *Sequenza delle operazioni da eseguire per lo shading del modello geometrico durante inizializzazione, update e disegno.*

Come si nota dal diagramma delle attività, è stato necessario aggiungere un'operazione all'entità ocean di Update(), questa dovrà essere chiamata durante la fase di update della shading pipeline. La reflex map infatti deve essere aggiornata ad ogni fotogramma ma non può essere fatto durante la chiamata a Draw() perché lo stato del motore grafico verrebbe compromesso (vedi Sez. 3.2.3). Le fasi di shading nel metodo Draw() sono evidenziate in verde perchè verranno effettivamente eseguite dallo shader *OceanSurfaceShader*. Non sono necessarie invece operazioni in fase di inizializzazione per lo shading.

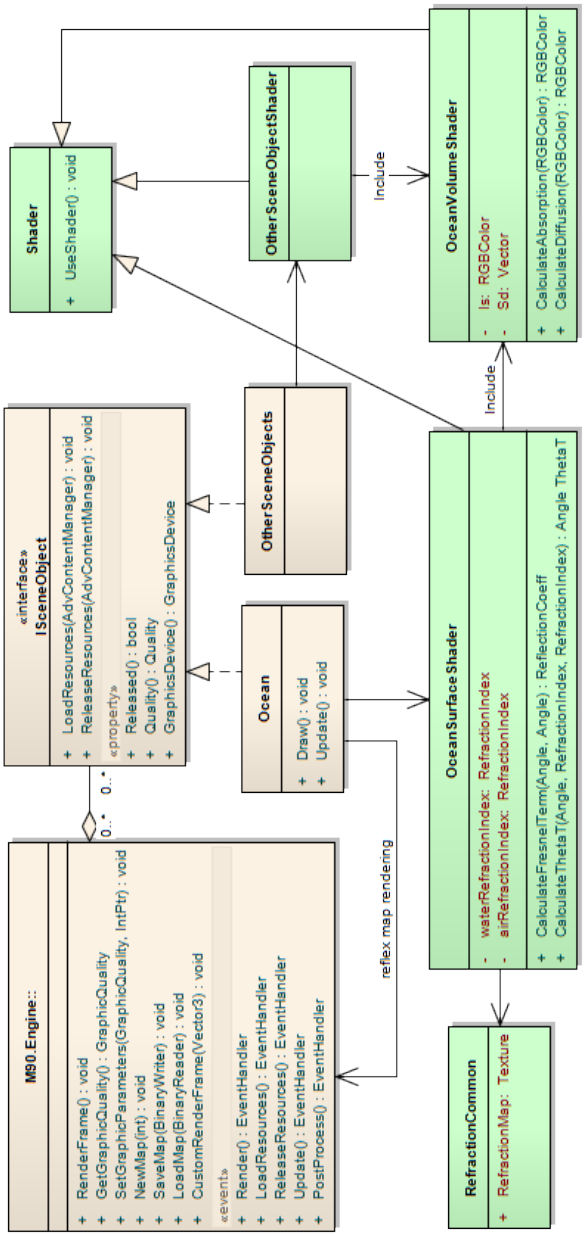


Figura 6.14: *Struttura del modulo aggiornata: Ocean.Update() è stato aggiunto per consentire il rendering della ReflexMap. Un riferimento a GraphiEngine è necessario per richiedere il rendering di un rame completo. Il metodo per effettuare il rendering è GraphicEngine.CustomRenderFrame(). Un riferimento a RefractionCommon è stato aggiunto: questo shader del motore grafico mette a disposizione la refraction map.*

6.2 Rendering volumetrico dell'acqua

Durante l'analisi sono stati identificati due effetti che dovranno essere simulati per produrre un effetto volumetrico per l'oceano: assorbimento elettromagnetico e diffusione volumetrica. Durante la loro trattazione verrà utilizzato il modello di oceano proposto nell'analisi e rappresentato in Figura 5.4.

6.2.1 Water volume ray-tracing

Per prima cosa, data la posizione del punto di vista C_{pos} ed il punto della scena su cui simulare gli effetti volumetrici C_{tar} è necessario:

1. Determinare se il volume dell'acqua viene intersecato, ed è quindi necessario simularlo.

2. In caso positivo, ricavare l'intersezione tra il segmento $[C_{pos}, C_{tar}]$ ed il volume.

La condizione di intersezione è calcolata come:

$$y_w - \min(C_{pos} \cdot \hat{y}, C_{tar} \cdot \hat{y}) > 0 \tag{6.6}$$

se risulta positiva, le componenti del segmento di intersezione $[a, b]$ verranno calcolate come:

$$\vec{C_{dir}} = \frac{C_{tar} - C_{pos}}{|\hat{y} \cdot (C_{tar} - C_{pos})|} \tag{6.7}$$

$$a = C_{pos} + \max(y_w - C_{pos} \cdot \hat{y}, 0) \cdot \vec{C_{dir}} \tag{6.8}$$

$$b = C_{tar} - \max(y_w - C_{tar} \cdot \hat{y}, 0) \cdot \vec{C_{dir}} \tag{6.9}$$

Questa tecnica risulta efficace per ogni configurazione di $[C_{pos}, C_{tar}]$ come mostrato in Figura 6.15. L'effetto volumetrico verrà quindi calcolato solo su questo segmento.

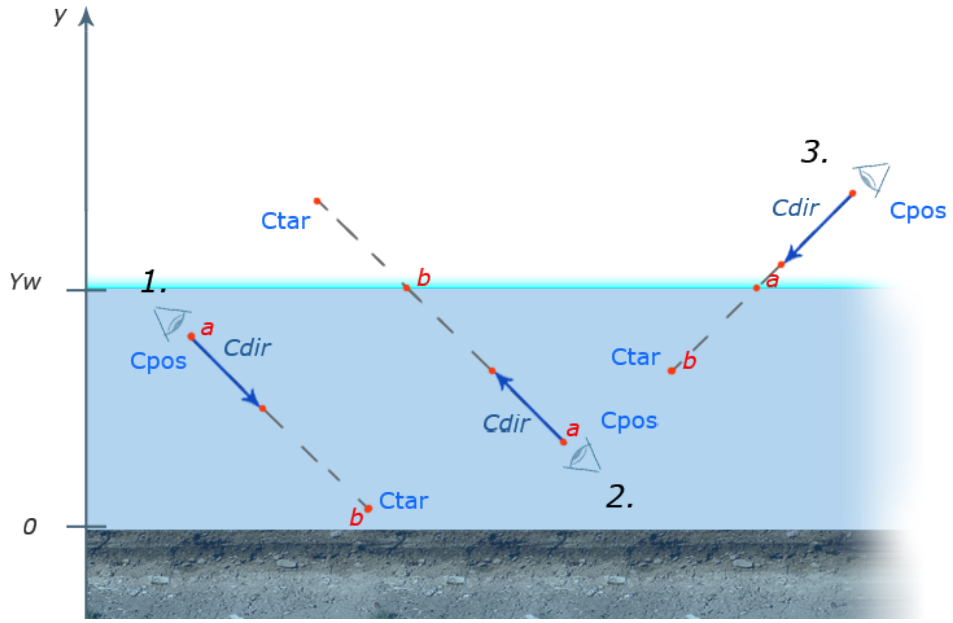


Figura 6.15: *L'immagine mostra il risultato del calcolo dei punti a e b in tre configurazioni diverse.*

6.2.2 Simulazione dell'assorbimento elettromagnetico

Come già spiegato nell'analisi, le molecole d'acqua assorbono parte della radiazione luminosa, possiamo quindi creare un'approssimazione della spettro luminoso della luce solare, dopo che ha percorso una certa distanza nell'acqua. Sappiamo che l'attenuazione introdotta nello spettro luminoso dipende dalla lunghezza d'onda e che le basse frequenze vengono attenuate maggiormente. Per semplificare la gestione di questo spettro, ne gestiamo solo 3 campioni, in corrispondenza delle lunghezze d'onda RGB. Utilizzeremo anche una costante $a_w \in \mathbb{R}^3$ che indicherà il coefficiente di attenuazione dello spettro rispetto alle tre lunghezze d'onda. Dato un colore RGB che rappresenta l'intensità della luce solare I_s, il colore della luce filtrata per una distanza d attraverso all'acqua sarà:

$$I_w(d) = I_s \cdot e^{-a_w d} \tag{6.10}$$

Figura 6.16: *Spettro della luce approssimato dall'equazione introdotta, al crescere della distanza d percorsa dalla luce.*

6.2.3 Simulazione della diffusione volumetrica

Possiamo approssimare la luce diffusa lungo un segmento $[a, b]$ verso a come:

$$I_d(a,b) = \int_a^b g(\alpha) D_w(l) I_w(T_d(l))\, dl \tag{6.11}$$

dove $g(\alpha)$ è la funzione angolare che determina l'intensità della diffusione in funzione dell'angolo α tra la direzione della luce solare e la direzione da simulare secondo il modello di Mie[7], $D_w(l)$ è la funzione di densità dell'acqua, $T_d(l)$ è la distanza percorsa dalla luce fino ad l e poi da l ad a e $dl = \sqrt{dx^2 + dy^2 + dz^2}$ è il differenziale dato dalla lunghezza del segmento percorso.

Questa formula può essere semplificata sulla base di alcune ipotesi:

- La densità dell'acqua è considerata uniforme, $D_w(l)$ si riduce quindi ad una costante moltiplicativa, che nel modello può essere ignorata.

- α rimane costante su tutto il segmento, questa funzione può quindi essere portata fuori dall'integrale.

Si ottiene quindi:

$$I_d(a,b) = g(\alpha) \int_a^b I_w(T_d(l))\, dl \tag{6.12}$$

La distanza $T_d(l)$ cercata può essere calcolata come:

$$T_d(l) = ||a - l|| + \frac{(y_w - l \cdot \hat{y})}{S_d \cdot \hat{y}} \tag{6.13}$$

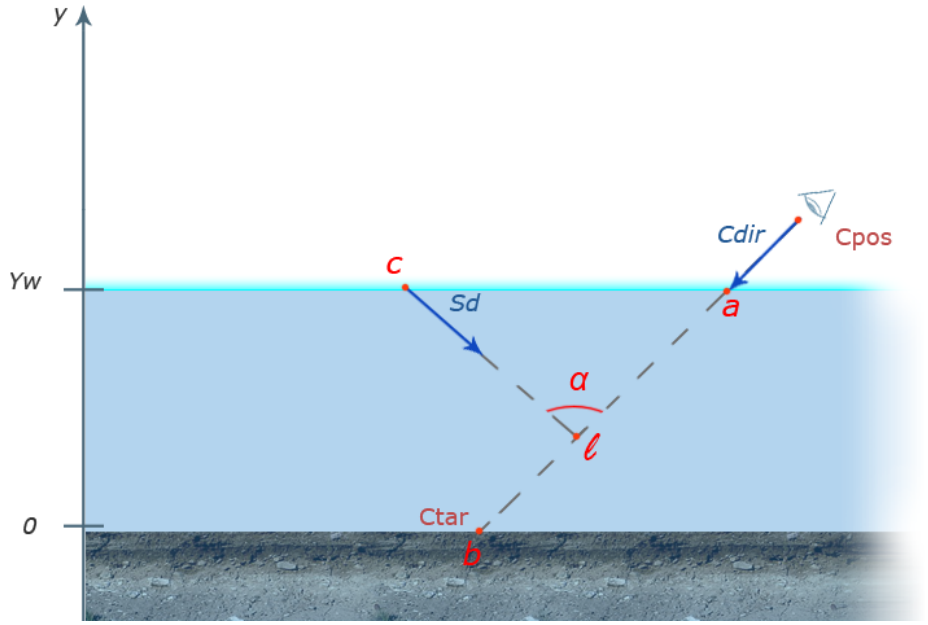


Figura 6.17: *Nell'immagine è rappresentata la distanza T_d cercata, che dallo schema può essere ottenuta come la lunghezza di $\vec{al}$ più quella di $\vec{cl}$.*

Sostituendo I_w ed T_d nella 6.12:

$$I_d(a,b) = g(\alpha) I_s \int_a^b e^{-a_w \left[||a-l|| + \frac{(y_w - l \cdot \hat{y})}{S_{dir} \cdot \hat{y}} \right]} \, dl \tag{6.14}$$

Il valore ottenuto I_d rappresenta l'intensità della luce diffusa dall'acqua lungo il segmento, ma nn è l'unica componente luminosa che giunge al punto di vista: non viene simulato il colore diffuso dagli oggetti dietro il volume, ad esempio dal fondale. La sua porpagazione nell'acqua può essere simulata con una singola diffusione lungo il segmento $[a, b]$ come:

$$I_{ss}(a,b) = I_s e^{-a_w \left[||a-b|| + \frac{(y_w - b \cdot \hat{y})}{S_{dir} \cdot \hat{y}} \right]} \tag{6.15}$$

I_d non è altro che la somma di tutti gli I_{ss} lungo il segmento moltiplicati per la funzione di fase, e può essere quindi riscritto come:

$$I_d(a,b) = g(\alpha) \int_a^b I_{ss}(a,l) \, dl \tag{6.16}$$

In conclusione, per simulare il colore della scena in una determinata posizione, tenendo conto della diffusione della luce nell'acqua, si calcola il segmento $[a, b]$ di intersezione con il volume, poi si approssima il colore come:

$$I_{scene}(a,b,C) = I_d(a,b) + C \cdot I_{ss}(a,b) \tag{6.17}$$

dove C è il colore della scena in quella posizione, senza tecniche volumetriche applicate.

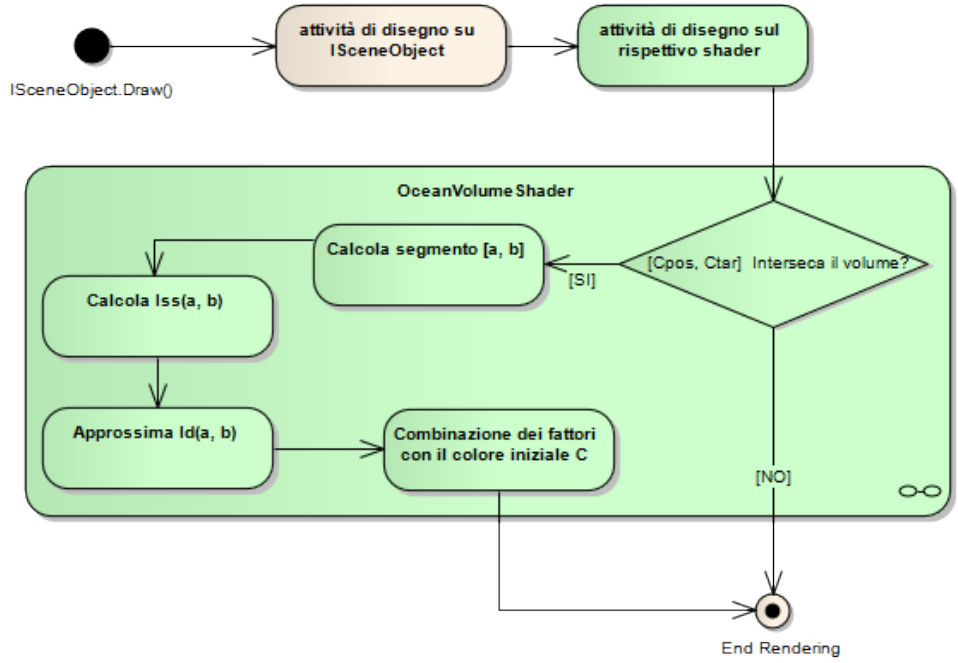


Figura 6.18: *comportamento di OceanVolumeShader, modificato sulla base di quanto detto in questa sezione. Come rappresentato dal diagramma, questo è preceduto dal disegno di un qualsiasi IsceneObject e viene richiamato solo dal suo shader.*

Capitolo 7

Implementazione

7.1 Struttura

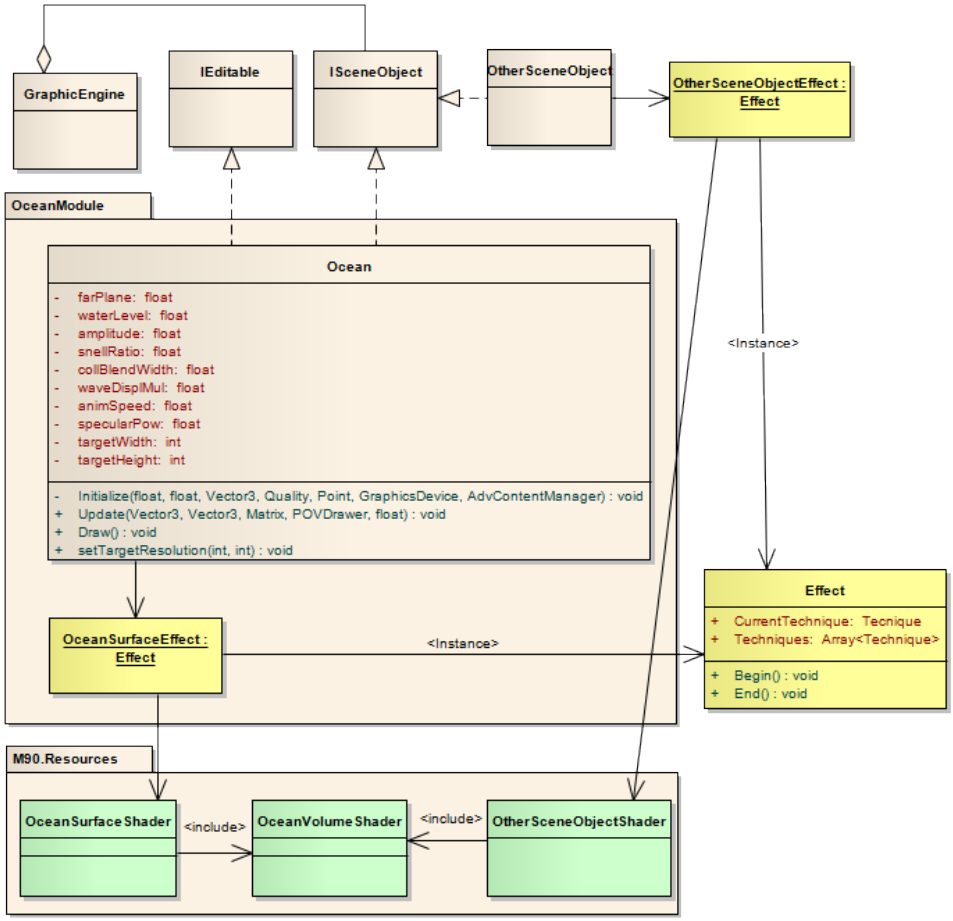


Figura 7.1: *Struttura globale del modulo. In giallo gli effetti, in verde gli shaders.*

La gestione degli shader in XNA richiede alcuni cambiamenti nella struttura, parresentati in Figura 7.1: infatti gli script scritti in HLSL detti appunto shader, non sono utilizzabili direttamente in .Net ma devono essere caricati dall'apposita classe *Effect*. Questa oltre che incapsulare gli script fornisce funzioni per il loro utilizzo: gli shader hanno una struttura fissa (vedi Sez. 2.3) di cui l'unica parte con cui ci interessa interagire da .Net sono le *Techniques*. Queste racchiudono uno script completo di vertex e pixel shader e la classe *Effect* permette di selezionarne una attraverso il modificatore *Effect.Techniques*.

Per eseguire un rendering con un determinato effetto si deve quindi:

1. Caricare lo script da utilizzare attraverso la classe *Effect*.
2. Selezionale la *Technique* con la quale si vuole effettuare il rendering.
3. Chiamare il metodo *Effect.Begin()*
4. Effetturare la chiamata di rendering attraverso il driver grafico, incapsulato dalla classe *Device*.
5. Chiamare il metodo *Effect.End()*

7.1.1 Attributi

DI seguito sono riportate le modifiche effettuate agli attributi della classe principale *Ocean*.

- **farPlane** è la massima distanza rappresentabile dal motore grafico, viene utilizzata per lo shift dei vertici del modello geometrico: come spiegato nell'analisi alcuni vertici vengono traslati ad un elevata distanza dal punto di vista, la distanza scelta è proprio quella massima.
- **waterLevel** livello dell'acqua rispetto alla minima altezza rappresentabile.
- **amplitude** utilizzata come costante moltiplicativa per la heighMap generata dall'algoritmo di animazione. Un valore maggiore genera onde più alte.
- **snellRatio** rapporto delle costanti $\frac{n1}{n2}$ viste nella progettazione.
- **collBlendWidth** durante il rendering della superficie dell'acqua, nei punti in cui questa interseca il terreno la rifrazione non basta a fornire una transizione realistica della riva. Per ovviare a questo problema si è deciso di utilizzare la trasparenza in prossimità delle rive che vengono associate alle zone in cui la profondità dell'oceano è minore di *collBlendWidth*. Per ottenere la profondità dell'acqua in un punto si utilizza la differenza di profondità della scena nella *DepthMap*(texture contenente la profondità senza oggetti rifrattivi) e la quella della superficie dell'acqua. Il parametro *collBlendWidth* serve quindi come fattore di scala per questo effetto, la cui applicazione è mostrata in Figura 7.2.

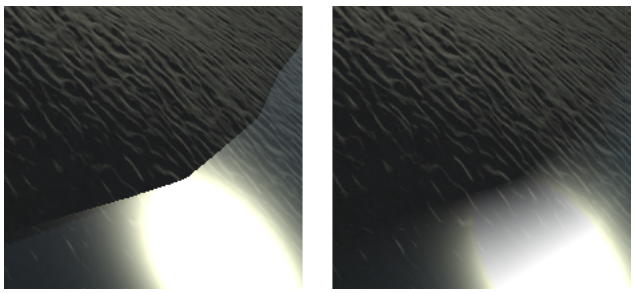


Figura 7.2: *A sinistra la riva senza alcuna tecnica applicata, a destra il blending effettuato sulla base della profondità.*

- **waveDispMul**Questa costante moltiplica l'offset orizzontale dei vertici durante l'animazione creando onde più tondeggianti o più nette.

- **animSpeed** Determina la velocità dell'animazione.

- **specularPow** Determina la selettività dei riflessi speculari, discussa durante la progettazione di questi.

- **targetWidth, targetHeigh** Sono le dimensioni della finestra di rendering del motore grafico. vengono utilizzare per scegliere una dimensione proporzionale per la ReflexMap: questa viene mantenuta sempre alla metà della dimensione della finestra di rendering.

Visto il numero di attributi che consente la personalizzazione dell'oceano, alla classe è stata fatta implementare anche l'interfaccia *IEditable*, in questo modo le modifiche a questi attributi verranno salvate automaticamente su file.

7.1.2 Operazioni

Anche per quanto riguarda le operazioni, sono state effettuate alcune modifiche. Tra le più rilevanti è da notare il parametro di tipo *POVDrawer* passato al metodo di update. Questo è un riferimento ad una funzione del motore grafico che può essere utilizzata per creare un rendering personalizzato della scena, e che viene utilizzato dall'oceano per aggiornare la reflexMap, che non avrà quindi più bisogno di un riferimento diretto al motore grafico. È stata inoltre introdotta l'operazione **setTargetResolution** per aggiornare all'oceano la dimensione della finestra di rendering, in modo che questo possa aggiornare quella della sua *reflexMap* di conseguenza.

7.1.3 Shaders

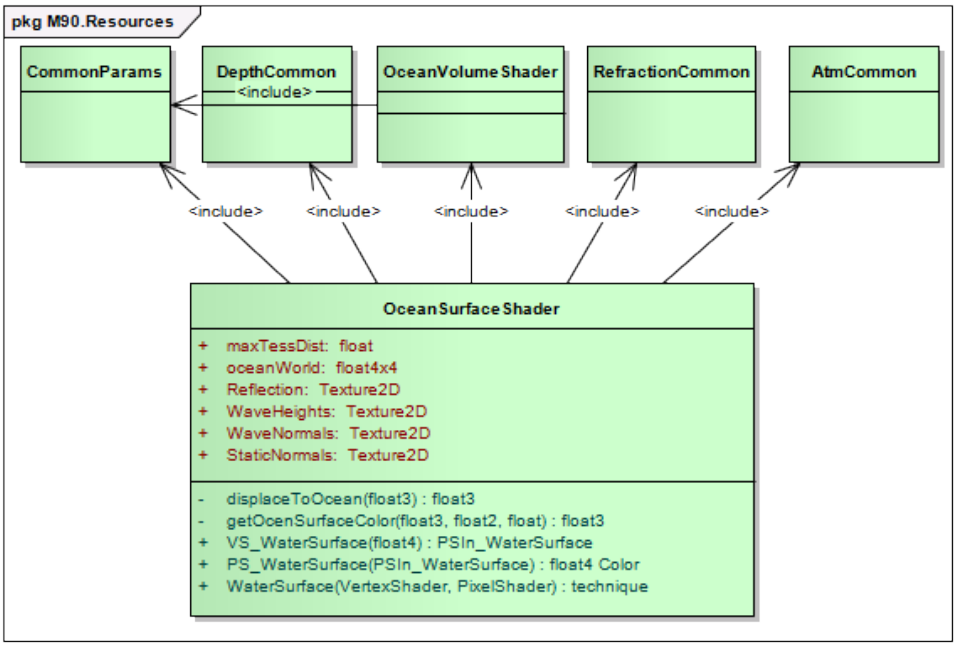


Figura 7.3: *Struttura dello shader per la superfice.*

Per lo shader della superficie sono stati aggiunti tutti i riferimenti agli altri shader utilizzati nel motore grafico, contenenti variabili globali utilizzate. Sono poi state aggiunte tutte le texture citate nella progettazione, più *maxTessDist* che indica l'estensione della zona animabile e *oceanWorld* che è la matrice di trasformazione per effettuare il riposizionamento della mesh Figura 6.3. Tra le operazioni si può notare l'aggiunta di *displaceToOcean*. Questa operazione proietta un qualsiasi punto sulla superficie dell'oceano. Questa è giustificata dal fatto che si è rivelato necessario modificare la tecnica di displacement dei vertici che doveva produrre l'illusione di un orizzonte illimitato Figura 6.2: la massima distanza rappresentabile dal motore grafico è piuttosto limitata e non garantisce un orizzonte statico, ad elevate altezze si può osservare la *fine* dell'oceano. È stato quindi aggiunto un ulteriore bordo esterno, riposizionato allo stesso modo del primo ma la cui posizione verticale viene eguagliata all'altezza corrente del punto di vista.

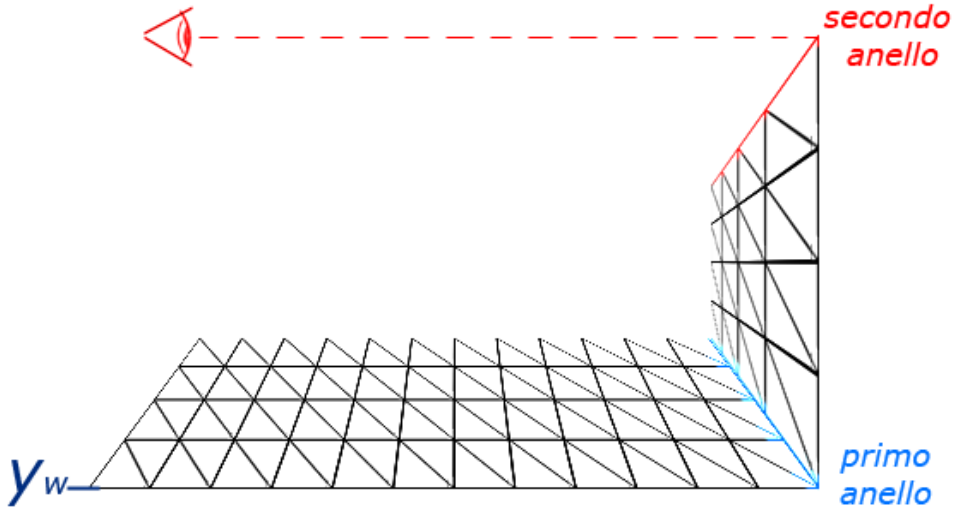


Figura 7.4: *Miglioramento del displacement della superficie dell'acqua con l'introduzione di un secondo bordo*

In questo modo l'illusione viene mantenuta per qualsiasi altezza ma lo shading è dipendente dalla posizione verticale dei vertici e verrebbe distorto. Una posizione reale del vertice viene quindi approssimata con **displaceToOcean**.

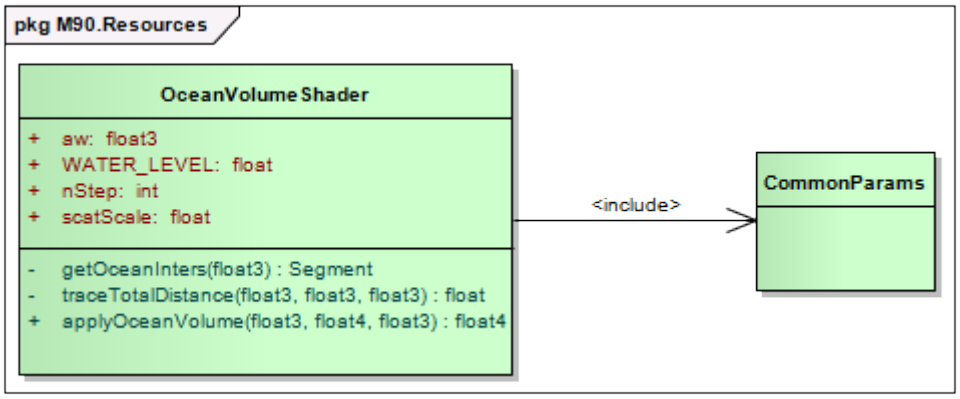


Figura 7.5: *Struttura dello shader per la simulazione volumetrica dell'oceano.*

L'attributo *nStep* è un intero che determina il numero di iterazioni con cui approssimare l'integrale dell'equazione 6.14. Infatti questo non è risolvibile analiticamente e viene approssimato tramete il metodo di Cavalieri-Simpson da *nStep* iterazioni.

7.2 Interazione

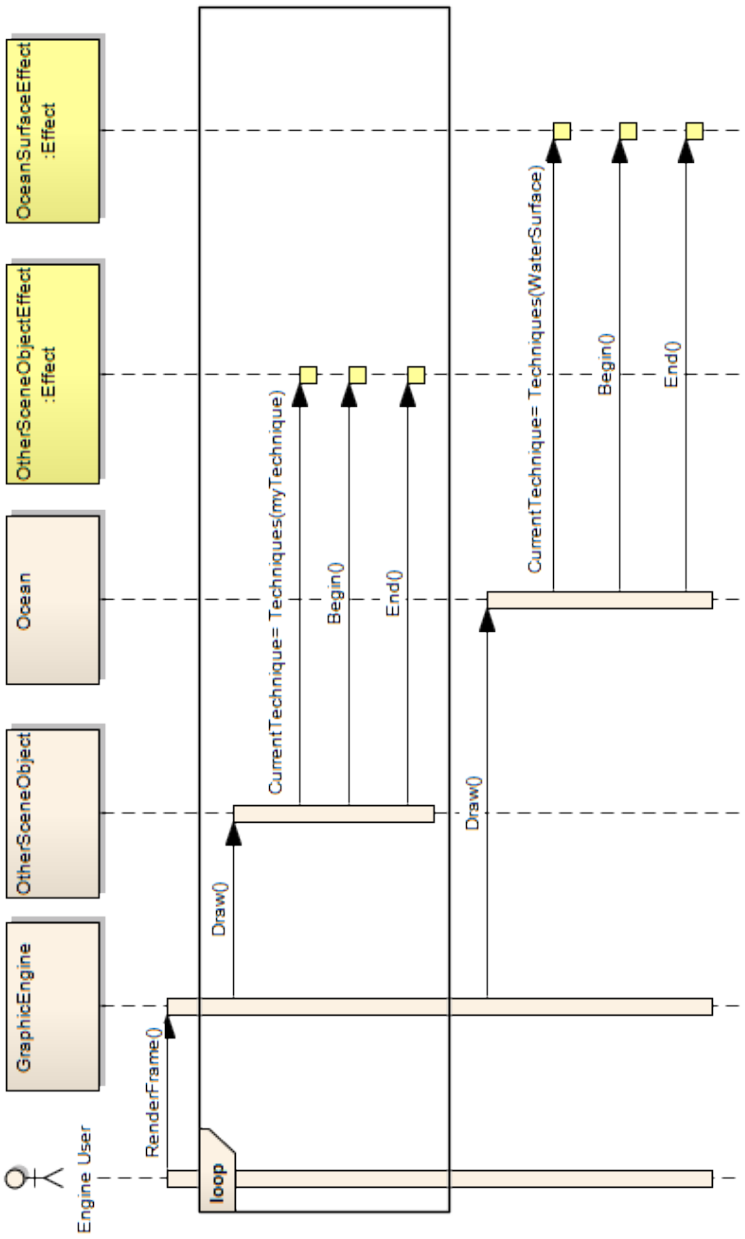


Figura 7.6: *Interazioni tra le classi interessate durante il rendering di un fotogramma.*

Nell'immagine Figura 7.6 si fa riferimento alla interazioni tra le classi delineate durante il rendering. Le chiamate di disegno vere e proprie vengono effettuate attraverso le api messe a disposizione da XNA, tra le chiamate a *Effect.Begin() ed Effect.End()*.

7.3 Comportamento

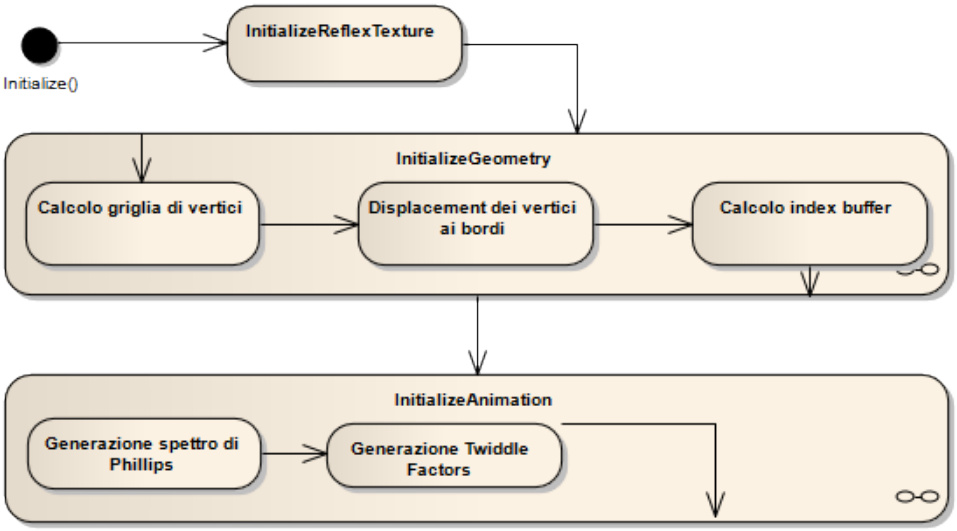


Figura 7.7: *Attività eseguite da Initialize()*

È stata aggiunta una nuova attività all'inizializzazione: *Generazione Twiddle Factors*. Questi sono dei fattori moltiplicativi ricorrenti durante il calcolo della IFFT per l'animazione e vengono quindi precalcolati per aumentare la velocità dell'algoritmo.

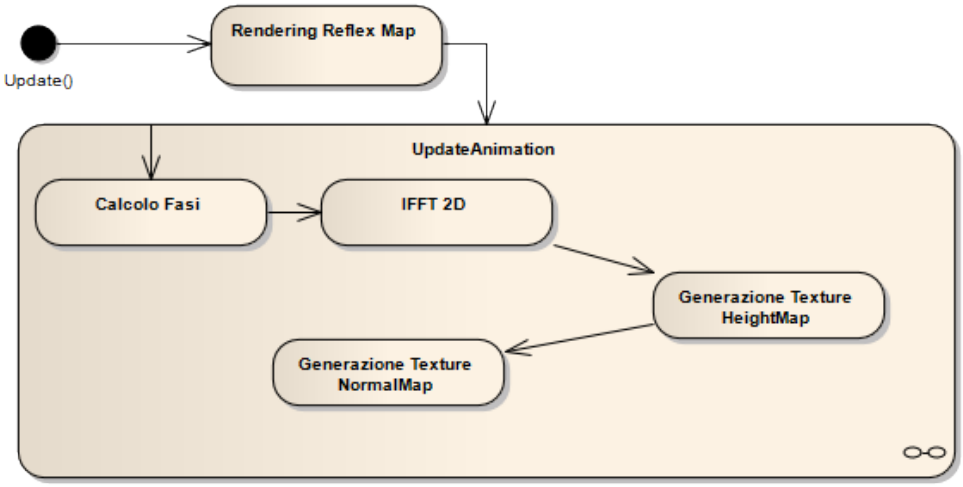


Figura 7.8: *Attività eseguite da Update()*

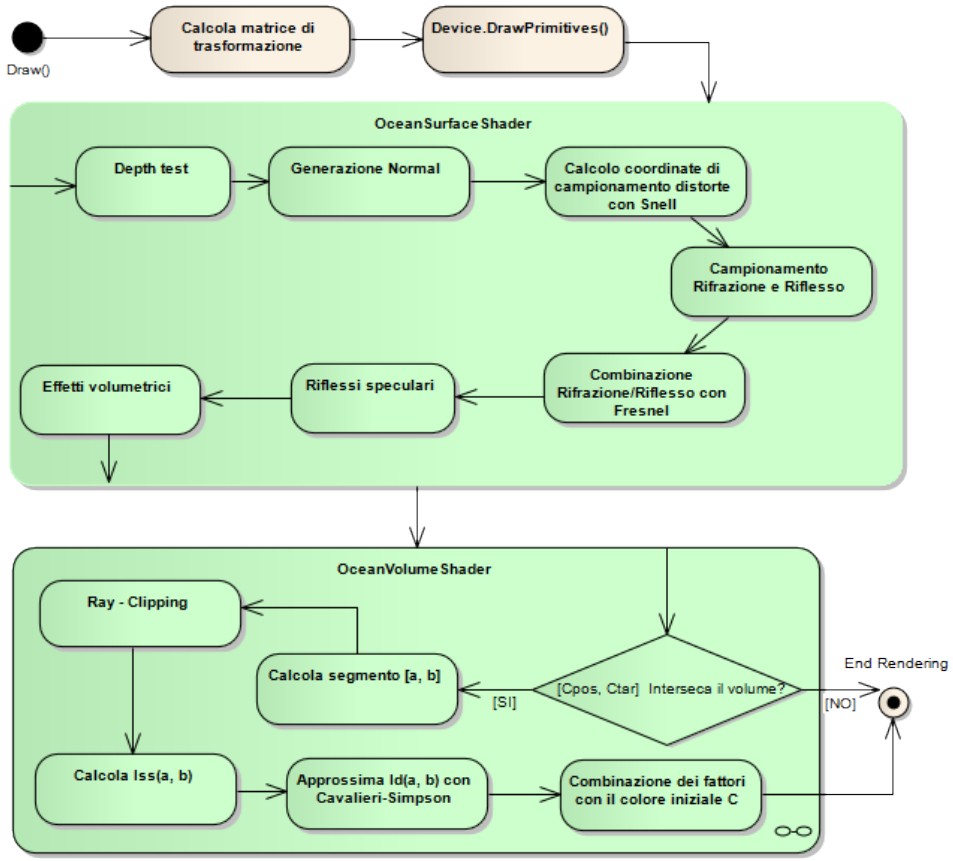


Figura 7.9: *Attività eseguite da Draw()*

Al volume shading è stato aggiunto l'attività di Ray-Clipping che effettua un *"taglio"* della parte finale di alcuni segmenti di intersezione che non contribuirebbero in maniera rilevante allo scattering perché troppo lontani dal punto di vista, ma diminuirebbero la precisione dell'integrazione numerica poiché questa calcola un numero di campioni fissato.

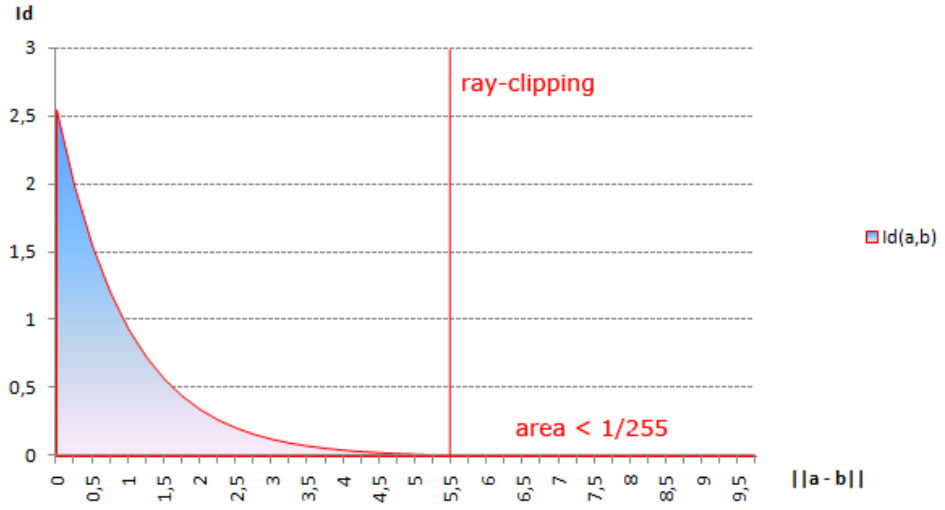


Figura 7.10: *La funzione rappresentata è Id(a, b) in funzione della lunghezza del segmento a, b. Dopo un certo valore critico, continuare ad integrare non può portare al cambiamento del colore in output, poiché l'area sottesa alla curva è minore del valore di un singolo scalino di colore ad 8 bit. Il segmento viene quindi tagliato a tale distanza e l'integrazione viene limitata al solo intervallo*

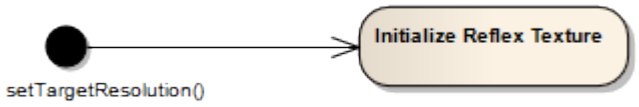


Figura 7.11: *setTargetResolution() viene chiamata ad ogni variazione della dimensione della finestra di rendering. In corrispondenza di questi eventi, la dimensione della reflex map viene aggiornata e mantenuta al 50% di quella della finestra.*

7.4 Modifiche aggiuntive al Motore grafico

A causa dei limiti nell'estensibilità del motore grafico per oggetti rifrattivi, occorre effettuare delle modifiche alla classe *GraphicEngine* per fare in modo che il modulo sia integrato correttamente all'interno della scena.

- **Costruttore** Aggiornato con l'inizializzazione di *Ocean* e l'aggiunta di questo alla liste degli *SceneObject* e degli *IEditable*.

```
public GraphicEngine(GraphicQuality startQuality, IntPtr
  targetHandle)
{
        [...]
        ocean = new Ocean(10000f, 40f, wind.GetGlobalVector(),
          settings.ocean, settings.resolution, device, content);
        sceneObjects.Add(ocean);
        editables.Add(ocean.GetName(), ocean);
}
```

- **Rendering pipeline** Aggiunta della chiamata di update e draw alla rendering pipeline:

```
/// <summary>
/// Disegna un fotogramma della scena sul controllo di output.
/// </summary>
public void RenderFrame()
{
        [...]
        ocean.Update();
        [...]
        ocean.Draw();
        [...]
}
```

- **SetGraphicParameters** Qui vengono impostati i parametri riguardanti la qualità grafica, questi dovranno essere quindi modificati anche per l'oceano:

```
public void SetGraphicParameters(GraphicQuality gq, IntPtr
  targetWindowHandle)
{
        [...]
    ocean.SetTargetResolution(gq.resolution.X, gq.resolution.Y);
    ocean.Quality = gq.ocean;
    [...]
}
```

- **Global Parameters** Gli attributi sullo shader per il volume, sono utilizzati dagli shader di tutti gli altri SceneObjects, la condivisione di questi viene amministrata nella funzione *updateGlobalParams* che deve essere quindi aggiornata:

```
private void updateGlobalParams()
{
        [...]
    content.SetGlobalParameter("aw", ocean.Aw);
    content.SetGlobalParameter("nStep", ocean.NStep);
    content.SetGlobalParameter("sscale", ocean.ScatteringScale);
    content.SetGlobalParameter("WATER_LEVEL", ocean.WaterLevel);
    [...]
}
```

Capitolo 8

Risultati e conclusioni

Il Requisito 1 e il Requisito 2 necessitano di test per valutare se sono stati effettivamente raggiunti.

8.1 Real-time

Per testare la caratteristica di real-time sarà sufficiente caricare nel modulo dell'editor un paesaggio che riproduca una della immagini di riferimento e valutare quanti fotogrammi per secondo è possibile produrre. Questi test dovranno tenere conto di due fattori che influenzeranno il risultato:

- **Configurazione Hardware** Le prestazioni dell'applicazione dipendono dalla piattaforma hardware utilizzata. I test sono stati effettuati su un computer desktop con la seguente configurazione:

Componente	Descrizione
Scheda madre	Asus P5k deluxe wifi-AP
CPU	Intel® Core™2 Duo E6400
Scheda video	AMD HD5770
Archiviazione	OCZ Vertex 3 ssd 120GB
Monitor	Samsung SyncMaster S24B350

Figura 8.1: *Configurazione hardware di test.*

AMD HD5770	
Caratteristica	Valore
Potenza di calcolo	1,36 TeraFLOPs
Memoria	1GB GDDR5
GPU clock	850Mhz
Stream Processors	800
Unità texture	40
unità Z/Stencil ROP	64
unità Color ROP	16

Figura 8.2: *Specifiche tecniche della scheda grafica.*

- **Risoluzione di rendering** La risoluzione del rendering influisce visibilmente sulle performance, poiché nonostante vengano processati gli stessi modelli geometrici, sono necessarie più chiamate ai pixel shaders, quindi per completezza i test verranno effettuati a varie risoluzioni.

I test sono effettuati sul paesaggio in Figura 8.3.

Figura 8.3: *Paesaggio utilizzato per i test.*

Nella tabella in Figura 8.4 sono presenti i risultati dei test. Con onde sinusoidali il modulo risulta piuttosto veloce, producendo quasi 70 fps anche ad una risoluzione full-HD. Se viene abilitato anche il displacement orizzontale per la generazione di onde oceaniche più realistiche, il rendering rallenta, ma anche nel caso peggiore il Requisito 1 è soddisfatto, mantenendo il frame-rate sopra i 30 fps. Il calo di prestazioni con l'utilizzo del displacement orizzontale non è da attribuirsi solamente all'algoritmo in sé, ma anche al formato della texture utilizzata per trasferire i valori di displacement allo shader. Questa utilizza 3 canali a 32bit, uno per ogni componente del vettore di displacement, che richiedendo quindi 12 byte per pixel. Questo si traduce in un elevato quantitativo di memoria grafica da aggiornare ad ogni frame che rallenta l'esecuzione.

Risoluzione	FPS con IFFT	FPS con IFFT + displacement orrizzontale
1024x640	80	53
1280x800	75	52
1440x900	74	52
1680x1050	72	50
1920x1080	69	47

Figura 8.4: *Fotogrammi per secondo prodotti nel rendering di un paesaggio con il modulo per l'oceano creato alle varie risoluzioni. Nella prima colonna sono presenti i risultati nel caso di onde sinusoidali, nella secondo viene effettuato anche il displacement orizzontale per generare le creste.*

L'utilizzo della CPU è rimasto a circa il 60% durante tutti i test, questo dimostra che l'algoritmo sviluppato non è rallentato dal processore, ma piuttosto dalla scheda grafica: questa infatti opera sempre al 100% producendo più fotogrammi possibili.

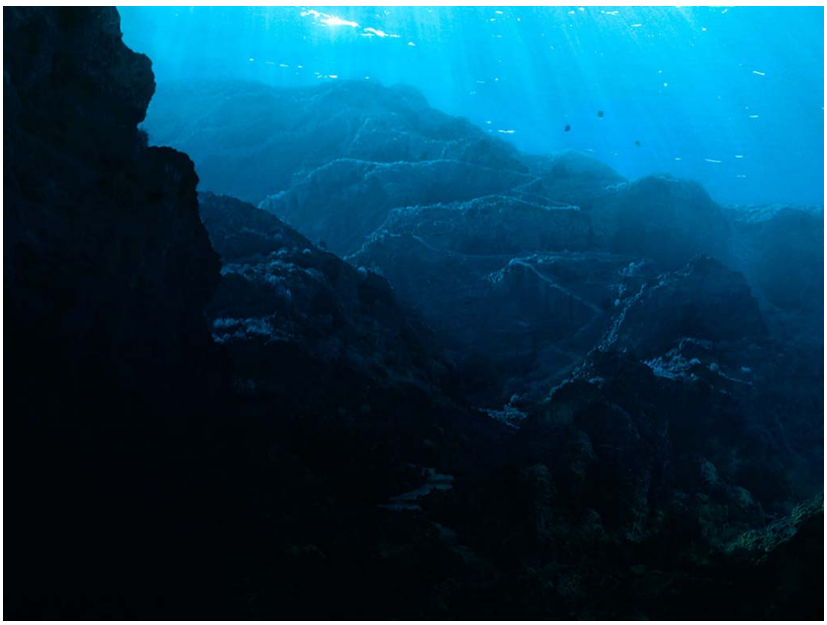
8.2 Fedeltà visiva

Utilizzando l'editor verranno riprodotte alcune delle immagini di riferimento, questo consentirà di valutare il raggiungimento del Requisito 2.
Dalle immagini 8.5 e 8.7 si può notare come la variazione del colore in funzione della profondità sia simulata correttamente. La superficie dell'acqua simulata risulta invece priva di onde a bassa frequenza, chiaramente visibili dall'alto nelle immagini di riferimento.
Dai risultati in figura 8.6 e 8.8 si può valutare l'accuratezza della diffusione subacquea: questa è nel complesso simulata correttamente, nella 8.8 è inoltre visibile l'effetto della *total internal reflection*. In queste immagini però non sono riprodotti i raggi solari, che sono invece visibili sott'acqua nelle immagini di riferimento sotto forma di bagliori sulla superficie in direzione del sole.

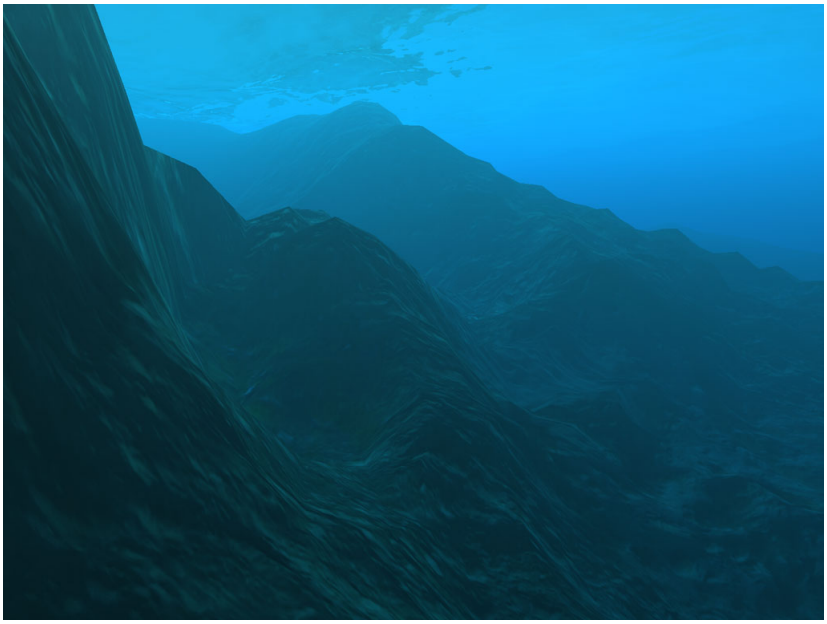
(a)

(b)

Figura 8.5: Riproduzione dell'immagine di riferimento 4.1(a): in (a) è riportata l'immagine originale, in (b) la sua riproduzione.



(a)



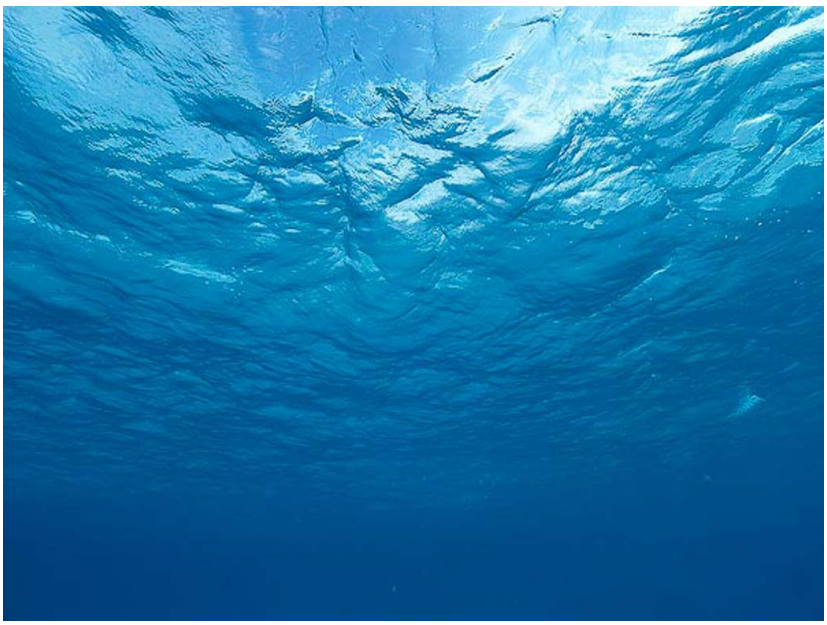
(b)

Figura 8.6: Riproduzione dell'immagine di riferimento 4.1(e): in (a) è riportata l'immagine originale, in (b) la sua riproduzione

(a)

(b)

Figura 8.7: Riproduzione dell'immagine di riferimento 4.1(c): in (a) è riportata l'immagine originale, in (b) la sua riproduzione



(a)

(b)

Figura 8.8: Riproduzione dell'immagine di riferimento 4.1(d): in (a) è riportata l'immagine originale, in (b) la sua riproduzione

8.3 Conclusioni

La simulazione dell'acqua è un problema affrontato oggi in moltissimi motori grafici che puntano alla qualità grafica e comprende un grande numero di fattori, ogni comportamento dell'acqua dipende dalla luce diffusa da altri oggetti piuttosto che dalla luce diretta, quindi è necessario considerare il fatto che migliorare la qualità del resto dell'ambiente migliora anche quella dell'acqua.

In questo documento è stato mostrato un modello completo per il rendering real-time dell'oceano e la sua implementazione su un motore grafico, affrontando tutti i problemi del caso. Il requisito di real-time si è mostrato raggiungibile anche con algoritmi statici basati su FFT, che può essere calcolata in tempi estremamente brevi anche su CPU. Attraverso l'analisi della fisica che regola il comportamento della luce nell'acqua, sono state sviluppate e testate formule di diffusione, rifrazione e riflesso per simulare al meglio questi comportamenti e fornire la massima fedeltà visiva. Un modello geometrico che rispondesse al Requisito 4 è stato sviluppato ed implementato con un algoritmo che ne permette la generazione a run-time. Tutti gli obbiettivi sono stati quindi raggiunti mantenendo il rendering in tempo reale.

L'algoritmo di animazione è stato implementato quasi interamente sulla CPU che dai test si è rivelata abbastanza veloce da mantenere gli FPS sopra la soglia del real-time.
In uno sviluppo futuro prevedo di trasferire gran parte del calcolo della FFT sulla GPU, aumentando le performance e permettendo l'utilizzo di matrici di dimensione maggiore che aumenterebbero di molto la fedeltà visiva.

Elenco delle figure

Bibliografia

[1] Eric Falsken. Simple managed directx render loop. *http://www.codeproject.com*, 2005.

[2] Riemer Grootjans. Riemer's 2d & 3d xna tutorials. *www.riemers.net*, 2003-2011.

[3] Gregory Ward Larson. The logluv encoding for full gamut, high dynamic range images. *Silicon Graphics, Inc. Mountain View, California*, 1999.

[4] Microsoft. Msdn library. *msdn.microsoft.com*, 2012.

[5] Martin Mittring. Finding next gen cryengine2. *Crytek GmbH, SigGraph*, 2007.

[6] Martin Mittring. A bit more deferred. *Triangle Game COnference*, 2009.

[7] Sean O'Neil. Accurate atmospheric scattering. *GPU Gems 2*, 2004.

[8] Jerry Tessendorf. Simulating ocean water. *Computer Graphics Laboratory at UCSD*, 1999-2001.

[9] Carsten Wenzel. Real-time atmospheric effects in games revisited. *GDC*, 2007.

[10] Wikipedia. Electromagnetic absorption by water. *it.wikipedia.org*, 2012.

[11] Wikipedia. Fresnel equations. *it.wikipedia.org*, 2012.

[12] Wikipedia. Microsoft xna. *it.wikipedia.org*, 2012.

[13] Wikipedia. Snell law. *it.wikipedia.org*, 2012.