ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA
Corso di Laurea in Ingegneria e Scienze Informatiche

COME SI PROGETTA UN GIOCO IN REALTÀ AUMENTATA COOPERATIVA USANDO WEBXR

Elaborato in
SISTEMI EMBEDDED ED INTERNET OF THINGS

Relatore
Prof. ALESSANDRO RICCI

Corelatore
Dott. Ing. SAMUELE BURATTINI

Presentata da
FILIPPO GURIOLI

Anno Accademico 2022/2023

Indice

Introduzione

Capitolo 1

Stato dell'arte

Di seguito si riporteranno informazioni circa lo stato dell'arte riguardante tecnologie e metodologie utilizzate per lo sviluppo di videogiochi e come vengano adattate alla realtà aumentata. Si affronteranno le problematiche della gestione cooperativa dell'esperienza di gioco e quali tecnologie ne permettano la realizzazione. Infine si darà una panoramica delle strutture messe a disposizione per creare mondi virtuali attualmente presenti in letteratura.

1.1 Game Programming Pattern

In questa sezione si farà principalmente riferimento al libro *"Game Programming Patterns"* di Robert Nystrom[2], in cui vengono descritti i pattern più comuni nel game design. Nel testo vengono citate 5 macrocategorie:

- Design Pattern rivisitati,
- sequencing pattern,
- pattern comportamentali,
- pattern di disaccoppiamento,
- pattern di ottimizzazione.

1.1.1 Design Pattern rivisitati

I design pattern sono i pattern più comuni e noti, sono stati descritti per la prima volta nel libro *"Design Patterns: Elements of Reusable Object-Oriented Software"* [1]. Nel libro di Nystrom vengono rivisitati alcuni di questi pattern in chiave videoludica ed in particolare si analizzano i seguenti: Command,

Flyweight, Observer, Prototype, Singleton e State. Data la vastità di argomenti che copre quel libro non verranno analizzati tutti, bensì solo quelli che si ritengono più rilevanti per questo progetto di tesi.

Command Pattern Il command pattern è definito nel libro come *"la reificazione di una chiamata di funzione"*, in pratica si tratta di modellare una classe che permetta di creare un livello di astrazione tra quando un comando viene invocato e quando viene eseguito. Si crea un'interfaccia 'Command' che specifica solo un metodo 'execute', dopodichè si creano delle classi che implementano tale interfaccia e che rappresentino un comando ben specifico (e.g. salta, corri, attacca etc.). Nella sua essenza il command pattern è quanto appena detto ma il modo in cui viene utilizzato è forse la cosa più interessante: nell'ipotetica classe 'InputHandler' si potrà dichiarare una istanza della classe 'Command' per ogni tipo di interazione che l'utente può eseguire (e.g. premere un tasto, drag and drop, gestures di vario genere etc.) ed assegnare a questo oggetto una qualsiasi delle classi precedentemente definite, svincolando così l'interazione dall'esecuzione di un determinato evento. I vantaggi del command pattern, spiega Nystrom, non si fermano solo alla possibilità di creare il key binding ma, serializzando i comandi, si possono mandare in rete per realizzare un gioco multiplayer. Si lascia il codice 1.1 come esemplificazione di quanto appena spiegato.

```
interface Command {
    execute(): void;
}

class Jump implements Command {

    /*stuff*/

    execute() {
        character.jump();
    }
}

class InputHandler {
    buttonX: Command;

    constructor() {
        //some logics that decide which key bind to which command
        this.buttonX = new Jump();
    }
```

```
    handleInput(): void {
        if (isPressed(Key.X)) {
            this.buttonX.execute();
        }
    }
}
```

Listato 1.1: Codice d'esempio per l'implementazione del command pattern.

Observer pattern L'observer pattern è uno dei più famosi e più utilizzati pattern presenti in letteratura. La stessa Microsoft ne ha fornito una implementazione all'interno del framework .NET. L'observer pattern si basa sui concetti di *evento* e *osservatore*: quando un oggetto modifica il suo stato e produce un qualche risultato visibile genera un *evento*. Questo può essere osservato dall'*osservatore* che, a sua volta, può reagire all'evento stesso, eseguendo del codice. Questa idea ribalta il concetto normalmente utilizzato in cui per permettere ad un oggetto B di reagire ad un cambiamento di stato di un oggetto A, quest'ultimo deve avere un riferimento all'oggetto B. Questo pattern crea un layer di astrazione tra l'oggetto che genera l'evento e l'oggetto che vuole reagirci, svincolando il generatore dal conoscere l'osservatore. Nel contesto di un videogioco questo pattern è spesso usato in modo verticale su tutta la codebase, a partire dalla gestione generale di eventi di gioco fino ad arrivare alla gestione di contesti particolari come la gestione degli input, degli achievements o delle collisioni. Data la grande versatilità del pattern è facile che possa sfuggire di mano, inserendolo anche dove basterebbe usare una banalissima chiamata di metodo. Per questo Nystrom fornisce un semplice criterio da seguire per evitare di cadere in errore: il pattern, spiega, andrebbe applicato ogni qualvolta l'oggetto che genera l'evento non ha senso che conosca l'osservatore, l'esempio lampante che viene fornito anche nel libro è il voler implementare un achievement alla prima volta che si viaggia ad una certa velocità verticale (si cade). Nel modo più semplice basterebbe mettere nel sistema che gestisce la fisica un controllo che verifica se le condizioni si avverano, nel qual caso chiamare un metodo del sistema di achievement 'unlockAchievement' per completare l'operazione. In questo scenario il motore fisico dovrebbe avere un riferimento alla classe di Achievement; al contrario se il motore fisico generasse un evento non preoccupandosi di chi reagisce a questo ecco che il vincolo referenziale non esisterebbe più. Si lascia il codice 1.2 come esempio di implementazione del pattern observer.

```
interface Observer { //the event listener
    onNotify(event:Event): void;
```

```
}

class Subject { //the event caller

    ObserverList: Array<Observer> = new Array<Observer>();

    public addObserver(observer: Observer): void {
        this.ObserverList.push(observer);
    }

    public removeObserver(observer: Observer): void {
        this.ObserverList.splice(this.ObserverList.indexOf(observer),
            1);
    }

    protected notify(event: Event): void {
        this.ObserverList.forEach(observer => {
            observer.onNotify(event);
        });
    }
}
```

Listato 1.2: Codice d'esempio per l'implementazione dell'observer pattern.

1.1.2 Sequencing pattern

I sequencing pattern sono pattern che permettono di gestire il tempo di gioco, ovvero la sequenza di eventi che caratterizzano il passare del tempo all'interno di un gioco. Si analizzeranno quindi i seguenti pattern: Game Loop e Update Method.

Game Loop È risaputo che i videogiochi rompano il concetto di macchina di Turing, ovvero non esiste un concetto di input, calcolo e output, dopo il quale il programma termina. Al contrario l'eseprienza di gioco risulta in un flusso continuo di operazioni, questo flusso ininterrotto è dato proprio dal game loop. Il game loop è un pattern che permette di gestire il tempo di gioco, ovvero il passare del tempo all'interno di un gioco. Il game loop è un ciclo infinito che si occupa di gestire gli input, aggiornare la logica di gioco e renderizzare il frame corrente.

```
while(true) {
    processInput();
    update();
```

```
    render();
}
```

Listato 1.3: Game loop semplice.

Le sue caratteristiche permettono di scindere il tempo reale dal tempo di gioco sfruttando il concetto di frame rate. Il frame rate è il numero di frame che vengono renderizzati in un secondo. Se non ci fosse alcun controllo sul framerate a cui viaggia il gioco si riscontrerebbero ambiguità per cui se un pc è più potente di un altro il gioco viaggerà più veloce sul primo che sul secondo, con conseguenze ancora peggiori se i due giocatori stessero giocando alla stessa partita in multiplayer. Per questo motivo il game loop è anche responsabile di rendere il gioco indipendente dal framerate, ovvero di aggiornare la logica di gioco in base al tempo trascorso dall'ultimo aggiornamento e non in base al numero di frame renderizzati. Per realizzare ciò (nel modo più semplice) si sceglie un framerate a cui far andare il gioco (e.g. 60 fps) e si calcola il tempo che intercorre tra un frame e l'altro (e.g. 16.6 ms), facendo attendere il gioco se avanzasse tempo (e.g. 60 - 16.6 = 43.4ms).

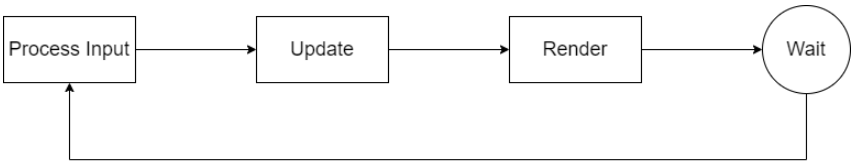

Figura 1.1: Logica di gestione del tempo nel game loop.

```
while(true) {
    let start = Date.now();
    processInput();
    update();
    render();

    sleep(start + FRAMERATE - Date.now());
}
```

Listato 1.4: Game loop con gestione del tempo.

Update Method

1.1.3 Pattern di disaccoppiamento

Component pattern

Event queue placeholder...

Si noti che sono stati analizzati solo alcuni dei pattern presenti nel libro di Nystrom, per una trattazione più approfondita si rimanda al libro stesso. Le scelte delle tematiche sono state fatte per brevità e per rilevanza rispetto al progetto di tesi.

1.2 Framework per la gestione cooperativa

1.2.1 Reti

1.2.2 Client-Server e peer-to-peer

1.2.3 Croquet

1.3 Framework per sviluppo 3D

1.3.1 Blender

1.3.2 Unity

1.3.3 BabylonJS

Capitolo 2

Progettazione e tecnologie

Analisi dei requisiti + tecnologie utilizzate.

2.1 Analisi dei requisiti

L'obiettivo del progetto è creare un'ambiente di realtà aumentata condivisa in cui l'utente possa giocare contro un altro al gioco di carte Yu-Gi-Oh. L'esperienza che il giocatore proverà dovrà essere quanto più simile alla versione proposta nella serie animata omonima.

Al momento dell'avvio l'utente dovrà affrontare un duello contro un'altra persona a Yu-Gi-Oh. Per la decisione del regolamento da seguire si è optato per una versione semplificata del gioco. Il giocatore potrà giocare carte mostro che rappresentano delle truppe schierate dalla parte del possessore. Queste truppe potranno quindi attaccare l'avversario per ridurne i punti vita. Saranno presenti anche carte magia e trappola che, tra i vari effetti, potranno modificare i punti vita, l'ambiete di gioco in cui gli utenti giocano o anche l'attacco e la difesa dei mostri propri e avversari. L'obiettivo del gioco consiste quindi nell'azzerare i punti vita dell'avversario, che comporterà la conclusione della simulazione.

2.1.1 Requisiti funzionali

- Il giocatore sarà in grado di vedere gli ologrammi propri e dell'avversario in tempo reale;

- il giocatore potrà interagire con un mazzo di carte virtuale pescando la prima carta;

- il giocatore potrà posizionare le carte che ha in mano sul campo e di conseguenza far apparire la corrispondente carta nello spazio di gioco condiviso;
- l'utente potrà ordinare l'attacco di un mostro, come attivare effetti di carte o passare il turno, tramite la selezione da un menù apposito;
- Ad ogni danno (o cura) subito (o inflitto) verrà visualizzato un ologramma condiviso che mostra i punti vita rimanenti del giocatore.

2.1.2 Requisiti non funzionali

- L'applicazione dovrà usare la tecnologia webXR per rendere fruibile, tramite un qualsiasi browser compatibile, l'esperienza di gioco.

2.2 Tecnologie

Elenco delle tecnologie utilizzate nell'elaborato.

2.2.1 Hololens

2.2.2 WebXR

2.2.3 MRTK

2.2.4 NodeJS

2.2.5 BabylonJS

2.2.6 Croquet

Capitolo 3

Sviluppo

Design architetturale, più filosofico che implementativo + implementazione dettagliata.

3.1 Design Architetturale

Illustra l'architettura generale del software, includendo diagrammi e spiegazioni delle componenti principali e delle interazioni tra di esse.

3.1.1 Back-end

3.1.2 Front-end

3.2 Progettazione dettagliata

Se necessario, fornisce dettagli aggiuntivi sulle specifiche tecniche e progettuali, come diagrammi di sequenza, diagrammi delle classi, modelli di dati e così via. (Probabilmente da rimuovere)

3.2.1 Model

3.2.2 View

3.3 Implementazione

Offre una panoramica dell'implementazione del software, inclusi gli strumenti utilizzati, la tecnologia impiegata e le scelte di sviluppo.

Conclusioni

Qui il testo delle conclusioni alla tesi. Non deve essere un riepilogo di quanto fatto nella tesi ma piuttosto le conclusioni raggiunte relative al lavoro svolto.

Ringraziamenti

Bibliografia

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Reading, MA, 1994.

[2] Robert Nystrom. *Game Programming Patterns*. Genever Benning, San Francisco, CA, 2014.