Indice

Elenco delle figure

Introduzione

Una *Reactive Floor* è un ambiente multi-modale e interattivo costituito da un tappeto posizionato in una stanza arricchita con semplici componenti audio e video. E-Learn è un esempio di *Reactive Floor* progettato per essere un'alternativa più coinvolgente ai metodi di insegnamento tradizionale, senza dover ricorrere a strumentazione costosa e di difficile reperimento. Nella piattaforma si possono svolgere dei giochi, o meglio dei *Serious Game*, con la partecipazione attiva degli alunni sfruttando contenuti multimediali e il movimento.
Ad ora i giochi sono stati creati da programmatori con conoscenze e capacità che un insegnante difficilmente possiede (ad esempio il gioco *Good or Bad* è stato programmato nel linguaggio *Processing*). Sorge quindi il problema di dover sviluppare degli strumenti che permettano agli educatori di creare i giochi secondo le loro esigenze e scopi, senza l'ausilio di tecnici specializzati. Per fare ciò è necessario progettare un authoring tool, ovvero un software per creazione assistita di applicazioni.
Lo scopo di questa tesi è mostrare come sia possibile sviluppare una serie di tool, concettuali e software, per facilitare la creazione dei *Serious Game* per la *reactive floor*, partendo dalla definizione di un modello per i giochi, fino a giungere all'implementazione di un prototipo di *game engine*.

Capitolo 1

Processi e strumenti di *authoring*

I serious game (SG) sono applicazioni che, oltre alla componente ludica tipica dei classici videogame, hanno anche un fine didattico, e possono essere sfruttate sia da scuole per svolgere lezioni interattive o da aziende o organizzazioni di vario tipo, come corpi di polizia, per il training dei propri dipendenti. La creazione di un qualsiasi videogioco richiede di norma un team di esperti e tecnici, con competenze che variano dal puro disegno artistico, per creare un oggetto o un personaggio, alla programmazione tramite librerie e tool appositi (come Unity3d). I software che facilitano lo sviluppo dei giochi, e più in generale la creazione altri programmi, diminuendo soprattutto la componente di programmazione, si definiscono *authoring tool*. Mentre si definisce processo di authoring un l'insieme di tecniche, passi e algoritmi da seguire per arrivare alla realizzazione assistita di un software; in letteratura viene spesso usato anche il termine *framework*.
Come vedremo, esistono varie tipologie di authoring tool a seconda del tipo di risultato che si vuole ottenere e della categoria di utenti che devono usare questi strumenti. Di nostro interesse sono quelli rivolti a user senza competenze tecniche, dato che vogliamo descrivere un processo di authoring per la creazione di SG da usare in ambito scolastico. In questo lavoro ci focalizzeremo in particolare su SG progettati all'interno di un pavimento interattivo denominato *E-Learn*, descritto nel secondo capitolo. Lo scopo finale di questo progetto è di fornire agli insegnanti degli strumenti per poter progettare e realizzare i propri giochi, senza l'aiuto di programmatori, tuttavia in questa tesi ne analizzeremo e realizzeremo solo alcune parti. In questo capitolo partiremo prima da una sintesi di alcuni lavori correlati a questa tesi, e presenteremo in seguito gli strumenti principali utilizzati.

1.1 Serious Game e authoring tool nella letteratura

Da una ricerca letteraria, si riscontra che nella maggior parte dei lavori finora svolti in questo ambito, il primo passo è la definizione di un modello. Ad esempio in [42] viene

presentato un approccio per riutilizzare i *Learning Object* attraverso l'uso del *Model Driven Approach* (MDA) nella creazione di un gioco. Un *Learning Object* rappresenta un' unità di apprendimento, e può essere ad esempio una lezione o un particolare concetto da apprendere. Il MDA si basa sull'uso di due modelli: il PIM (*platform indipendent model*) e il PSM (*platform specific platform*). Il primo è una rappresentazione astratta, mentre il secondo è processabile da un computer. La trasformazione del PIM in PSM si fa convertendolo in un file XML da dare in pasto a un game engine. Questo approccio non sembra tuttavia aver avuto molto seguito in letteratura, soprattutto per le difficoltà implementative, secondo [6], tuttavia vedremo come l'idea di fondo ricorra spesso in diversi progetti.
Infatti, in [16] si pone come obiettivo quello di colmare la mancanza di modelli per la narrativa e presenta uno schema, *StoryTelling Designing Model* (SDM) che cerca di conciliare diversi elementi importanti per questo genere di applicazioni: in particolare, obiettivi didattici e struttura della narrazione. Questo modello è poi usato come base per produrre uno schema usando un editor ad hoc che fa uso di blocchi *drag and drop*. L'output dell'editor, un file XML, viene dato in input ad un *player*. Tutto ciò viene poi integrato in una piattaforma di e-learning già esistente, IWT(*Intelligent Web Teacher*). L'articolo ha il pregio di presentare un modello teorico e pratico ricco di elementi che tiene conto degli elementi chiave della narrativa e della pedagogia.
Nell'ambito delle piattaforme web troviamo Gamelt[45], pensato per realizzare scenari educativi o modificare quelli già esistenti. Esso si concentra più sulla componente pedagogica dei giochi invece della loro creazione, e cerca di tener conto delle richieste di educatori e studenti. Tuttavia in questo lavoro non viene presentata la parte di authoring da un punto di vista tecnico ed implementativo ma viene mostra solo la piattaforma nella sua struttura generale.

Nel già citato [6] viene messo in evidenza la differenza principale dei serious game da quelli tradizionali, dove l'obiettivo principale è il divertimento del giocatore che a volte non ha bisogno di concentrarsi troppo per giocare; nei serious game questa componente ludica può venire a mancare. Si sottolineano inoltre i vantaggi della *designed complexity* rispetto alla *emergent complexity*: quest'ultima è la complessità generata dall'imprevedibilità dei giocatori durante il *gameplay* e può essere un ostacolo agli scopi educativi dei serious game. Se invece la complessità è progettata a priori, essa può essere controllata. Viene poi proposto un *framework* per la progettazione dei giochi, suddiviso in tre livelli: il livello concettuale che considera il gioco come un sistema di elementi correlati e comunicanti tra di loro; il livello tecnico dove si definiscono i tool di sviluppo; e infine il livello pratico dove si forniscono principi per diminuire la complessità di progettazione.

L'articolo manca tuttavia di dettagli sulla effettiva efficacia di questo approccio.
In [33] invece l'accento è spostato sul costo dei serious game e viene proposta una procedura da seguire per la progettazione efficiente, basato su un'attenta assegnazione dei compiti tra project manager, esperti in pedagogia, artisti e sviluppatori, e l'utilizzo di widget da far utilizzare a chi crea i modelli per creare specifiche precise per il programmatori. Questo approccio è un'alternativa all'utilizzo di un game engine per creare il gioco vero e proprio, ma in questo modo gli insegnanti sono costretti a rivolgersi a un team di sviluppatori, e i costi di progettazione pertanto rimangono alti. Sulla stessa onda, in [44] si propone uno schema per la progettazione di giochi che sfruttano le tecnologie legate alla *Mixed Reality* (definizione progetto, fase creativa, formalizzazione e definizione delle specifiche), e propone un authoring tool (*MIRLEGADEE*) per aiutare gli sviluppatori. Il target di questo lavoro sono ancora i programmatori. Mimesis, mostrato in [56], è un'architettura che permette di generare automaticamente una serie di azioni per raggiungere degli obiettivi. Esso è stato progettato per gli sviluppatori, in modo che questi non debbano prevedere ogni singola situazione che può verificarsi in un gioco. Mimesis presenta un'architettura modulare dove abbiamo diversi componenti che gestiscono varie task, come la generazione delle azioni, la struttura temporale da seguire per l'esecuzione, la comunicazione, etc. Come si può vedere in questo lavoro si è ancora molto distanti dall'idea che dovrebbero essere gli insegnanti a creare i giochi.

Un altro progetto interessante è *U-Create*, presentato in [47], che consiste in una serie di editor per implementare varie parti di una storia: scenario, azioni, storia e stage. Questi tool sono in teoria in grado di integrarsi con tecnologie legate alla *Augmented Reality* e alla *Mixed Reality*. L'articolo inoltre sostiene che il modello UML non sia adatto per rappresentare questo genere di giochi, tuttavia non riporta molto per sostenere questa tesi. Evidenziamo però l'idea di usare più editor e la suddivisione di una storia in varie parti che come vedremo ha avuto molto seguito.
Più legato all'authoring dei giochi è l'approccio mostrato in [2]: si mette a disposizione una piattaforma con dei template e dei pacchetti SCORM che l'insegnante dovrà selezionare per ottenere il risultato desiderato: questo viene fatto dalla piattaforma web *SGAME* presentato in [2]. In questo modo lo sforzo dell'insegnante è minimo ma naturalmente c'è la grossa limitazione determinata dall'uso di *template*: se l'insegnante volesse creare qualcosa che non è presente nella piattaforma dovrebbe rivolgersi agli sviluppatori. Per di più non è chiaro se è possibile controllare l'interazione delle risorse multimediali (immagini, audio, video) con il gioco.
L'authoring tool che forse viene più citato in letteratura è StoryTec, che vediamo proposto in [19]. In questo lavoro si evidenziano tre aspetti principali da conciliare nella

progettazione di un serious game: story-telling, gaming e learning. Viene poi descritto il concetto di NGLOB (Narrative Game-based Learning Object), un modello che rappresenta le componenti di un gioco digitale secondo la suddivisione citata prima. Gli NGLOB sono integrati nella piattaforma di authoring *StoryTec*, che genera dei file ICML, in formato XML, da dare in input allo *Story Engine*. Gobel propone quindi una distinzione delle parti di un gioco, di cui riporta un modello teorico e suggerisce un'implementazione pratica per convertire il modello descrittivo in dati che possano essere processati in un elaboratore. Altri dettagli su *StoryTec* si possono trovare enunciati in [41] e [39]: in particolare troviamo che la struttura di questa piattaforma di authoring è costituita da diversi editor che servono a progettare le varie componenti dei giochi (scena, azione, eventi, etc.). Ad esempio nella definizione delle azioni si usa la programmazione visuale per costruire un albero, collegato a delle regole definite in un altro editor, *Policy editor*. Questo lavoro mira inoltre ad essere indipendente dalle piattaforme in cui il gioco dovrà girare, anche se questo obiettivo non è stato ancora raggiunto. Viene anche sottolineato come *StoryTec* cerchi di colmare il gap tra la pedagogia e i giochi, potendo definire anche gli obiettivi didattici. StoryTec quindi può inoltre utilizzato per più generi di giochi, dall'avventura ai puzzle e agli enigmi. Inoltre, è possibile creare giochi in terza persona integrandolo con il game engine *Wintermute*.

In [55] viene presentato *SeGAE*, un ambiente per l'authoring di serious game: la definizione dei personaggi, eventi, azioni, obiettivi, messaggi e altri oggetti del gioco viene fatto tramite un file XSD (XML Schema Definition). SeGAE è implementato nell'ambiente FLEX Builder, un IDE progettato su Eclipse, seguendo un'architettura MVC (*Model View Controller*), dove le view sono implementare in XML e i controller e i proxy sono realizzati con ActionScript, il linguaggio di scripting di Adobe Flash Player. I dati del gioco sono scambiati tramite file XML generati dall'ambiente di authoring. Per modellare i vari oggetti sono disponibili degli editor appositi (es: editor dei personaggi, dello scenario temporale, etc.). È inoltre possibile adattare la lingua in base all'utente finale: tutti i messaggi sono identificati da un unico identificare dipendente dalla lingua; l'editor manipola questi identificatori e non i messaggi veri e propri. SeGAE permette quindi all'insegnante di definire obiettivi, sequenze di azioni, scenari e personaggi del gioco, oltre a poterne controllare l'andamento in tempo reale. L'articolo purtroppo è scarno di informazioni dettagliate e come per altri lavori non si focalizza sul fatto che il processo di creazione dei giochi deve essere svolto dagli insegnanti.
Possiamo trovare in letteratura anche lavori legati allo sviluppo di applicazioni per smartphone, come in [14]: viene proposta una piattaforma per creare storie che sfruttano sia il mondo digitale che quello fisico; una *app* Android è stata progettata come hub per una

storia creata in una piattaforma web per l'authoring, e degli elementi fisici contrassegnati da tag NFC che svolgono la funzione di trigger per gli eventi della narrazione. L'articolo mette in luce come si possano sfruttare i sensori di uno smartphone per creare storie che interagiscano con la realtà. L'applicazione per la creazione delle scene è stata progettata per eliminare ogni forma di programmazione, pertanto può essere utilizzata anche dagli insegnanti senza l'aiuto di programmatori.
Un altro filone che si può individuare in letteratura, basato sulla definizione di "linguaggi naturali", si può individuare in [7], dove si presenta un approccio per l'authoring di giochi *point-to-click*, in cui il gioco, il modello che descrive gli obiettivi e i percorsi di apprendimento possono essere definiti usando un editor XML. Questo lavoro è un buon passo in avanti per la creazione di metodi per l'authoring adatti anche agli insegnanti senza un background da programmatore: l'autore può descrivere in linguaggio naturale un gioco e poi può usare una grammatica in XML fornita dagli sviluppatori per tradurre il modello, e darlo in pasto a un game engine. Inoltre, l'articolo definisce i concetti di stati del gioco e stati dell'UoL (*Unit of Learning*) per permettere la comunicazione tra il gioco e il flusso di apprendimento, tutto definito in XML. Da questo lavoro è nato *E-Adventure*, una piattaforma per l'authoring con interfaccia grafica e un motore opensource implementato in Java.
In [25], partendo dalla mancanza di una tassonomia comune dei modelli pedagogici, viene sviluppato uno schema astratto che possa rappresentare il maggior numero di modelli possibili. L'*IMS-Learning Design* è in grado di rappresentare in un file XML diversi tipi di giochi e diversi modelli pedagogici. Nello schema sono presenti attività, ruoli, condizioni, ambienti (gli strumenti per completare le attività). IMS-LD è stato pensato come un rappresentazione astratta fatta dagli sviluppatori, ma essa deve essere nascosta agli utenti finali. La fase di authoring può essere realizzata con un editor XML ma risulterebbe troppo arduo per chi non ha abilità da programmatore.

Un altro ambito dove si possono trovare progetti legati ai SG è quello dell'eredità culturale, dove tramite la tecnologia si può migliorare la fruizione dei contenuti di musei e siti dal valore artistico e culturale. Ad esempio in [4] viene mostrato Tie, un SG progettato esplorare musei e palazzi in un ambiente 3D. L'articolo parte dalla descrizione dell'ambiente virtuale e degli approcci usati per realizzarlo, e continua con l'analisi del modello usato per realizzare il gioco. In particolare viene usato un modello *task-based*, basato sull'uso di mini-giochi all'interno dell'ambiente, posizionati in punti di interesse.
BME (Beginning-Middle-End) è un framework concettuale pensato per aiutare gli insegnanti nelle creazione di storie: in [29] viene presentato un tool basato sull'implementazione di BME tramite i *form* di Google Doc. Il framework è la fusione di due modelli

basato sulla suddivisione della narrazione in tre parti (beginning, middle, end) e la struttura ricorsiva della narrazione, ovvero ogni componente della narrazione può essere a sua volta scisso in inizio, mezzo e fine. Tuttavia, la descrizione della fase implementativa è poco tecnica e di alto livello, e non entra abbastanza nei dettagli.
Un lavoro che si avvicina agli scopi di questa tesi lo troviamo in [50]: l'approccio proposto da questa ricerca è quella di creare un processo per la creazione di serious game che parta da una modellazione tramite diagrammi UML, che rappresenti la struttura del gioco (scene, atti, azioni, oggetti, e giocatori) e la sua logica, e arrivi a realizzare un editor visuale composto da elementi *drag e drop*. Quest'ultima parte non viene ancora presentata. Tuttavia la parte di modellazione astratta sembra essere un buon punto di partenza, e secondo il paper è in grado di rappresentare anche diversi sviluppi a seconda delle scelte del giocatore. Vengono anche presentati i requisiti dei SG da qui è stato ricavato il modello, ma non tutte sono necessarie: un *reactive floor* non ha bisogno ad esempio di un'interfaccia grafica per i giocatori.
In [34] si presenta *MoPPLiq*, un modello che rappresenta i SG come un insieme di attività, condizioni, obiettivi, input e output. Esso viene implementato come un file XML e può anche essere visualizzato come un albero o un diagramma ER. Ciò permette di realizzare un authoring tool in grado di rappresentare questo schema, dove un insegnante può intervenire per modificare gli obiettivi pedagogici. Tuttavia il modello non prende in considerazione la possibilità che un insegnante voglia realizzare un gioco da zero, ma solo la modifica di uno esistente. In compenso, MoPPLiq sembra adattarsi bene a diversi giochi già esistenti.
Un altro aspetto importante nella progettazione dei SG è la valutazione della loro efficacia. Ad esempio in [57] viene definito un protocollo di validazione per confrontare le performance con la didattica classica. Altre tecniche le possiamo trovare in [54] dove sono descritte diversi metodi, dai più semplici e utilizzati, come i test di valutazione delle conoscenze, sia di tecniche più complesse basate sulle neuroscienze, come l'elettroencefalogramma.
Osserviamo come XML sia spesso usato per descrivere i giochi. Ne sono esempio lavori già citati come StoryTec e troviamo altre prove a sostegno di ciò in [43], dove si descrive un SG in XML e si progetta una game engine composto da un generatore di alberi (rappresentazione del gioco), un *repository* dei contenuti e il generatore del gioco. Questo è dovuto all'indipendenza di XML dalla piattaforma di utilizzo, dalla grammatica semplice che permette di definire i propri tag (si può salvare in pratica qualsiasi tipo di dato) e dalla disponibilità di librerie per il parsing di questo veicolo di dati, ampiamente usato in ambito Web e Database.
Un'altra componente fondamentale per un authoring tool è l'interfaccia grafica (UI). Ab-

biamo visto che ve ne sono diversi: multi-editor come per StoryTec, una UI per ogni fase dello sviluppo, o più semplicemente un editor XML o UML, o nel caso di tool per professionisti (Unity3d), sia elementi grafici che ambienti di programmazione. Avendo a che fare con un ambiente interattivo e multimodale, dove non si gioca più con mouse e tastiera, la progettazione di un interfaccia grafica adeguata presenta delle nuove sfide [12]. Innanzitutto, in un IME abbiamo diversi canali di input, per via dell'uso di sensori. Anche solo la rappresentazione tramite un simbolo di una gesture o di uno spostamento non ha uno standard definito; e più in generale mancano convenzioni comuni anche nella progettazione e nell'emulazione per questi tipi di ambienti. Inoltre, l'emulazione dell'ambiente cambia con un IME, poiché la piattaforma target delle applicazioni è diversa da quella su cui vengono progettate. Sorge anche la difficoltà di rappresentare un giocatore fisico in un PC.

1.2 Game Engine: concetti base

Descriviamo ora cosa sia un game engine, componente fondamentale per qualsiasi authoring tool. Come suggerisce il nome, un *game engine* è il motore che permette ai giochi di funzionare, ovvero si tratta di un insieme di componenti software necessari ad un computer per poter creare ed eseguire il codice di un videogioco. Tra queste componenti vi sono anche tool per lo sviluppo di questi videogame, come accade per alcuni game engine commerciali come *Unity3d* e *UnReal Engine*, pensati per sviluppatori di professione.
L'idea di fondo in un game engine è di un ciclo continuo, ovvero viene richiamato durante tutto il gioco per aggiornare prontamente le variabili delle componenti e rispondere agli eventi per passare da uno stato ad un altro. Spesso il confine tra gioco e game engine non esiste: può capitare infatti che un gioco contenga delle logiche e del codice con delle caratteristiche peculiari che non possono essere riutilizzati per creare un'altra applicazione; in questi casi non si parla più di game engine. Ciò che fa la differenza è l'architettura orientata ai dati e la riusabilità del codice: ovvero il primo requisito implica la possibilità di definire le componenti di un videogame solo tramite dei dati, ed questo implica di per se il riuso di "parti" di un gioco per poterne creare un altro, con al più qualche modifica. Pertanto possiamo formulare questa definizione:

Definizione 1. Un *game engine* è un software estensibile che può essere usato per creare giochi differenti, con al limite qualche modifica marginale al codice[21].

Il game engine ideale dovrebbe permettere di creare qualsiasi tipo di gioco, tuttavia questo traguardo non è ancora è stato raggiunto ed è probabilmente impossibile da realizzare. I game engine, come tutti i software, devono realizzare dei compromessi tra il target

hardware e la tipologia di applicazioni da implementare. Ad oggi, molti di questi software sono ottimizzati per un particolare genere di gioco, e per determinate piattaforme (PC, console, smartphone, etc.). Ad esempio, *Quake Engine III* permette di modellare qualsiasi FPS (*First-Person-Shooter*), mentre Unreal 4 si spinge oltre, permettendo di realizzare qualsiasi FPS e buona parte dei giochi in terza persona. Ciò si deve al fatto che la differenza da genere a genere è notevole. Un FPS deve immergere il giocatore in un mondo realistico, e pertanto il *rendering* di questi ambienti 3D deve essere molto efficiente, i NPC (*Non Playable Character*) devono avere una AI ben realizzata; la fisica dei corpi, l'audio e l'animazione dei giocatori devono essere implementati al meglio. Se invece consideriamo giochi di strategia come *Age of Empire*, la realtà virtuale è costituita da un layout a griglia, dove quindi la risoluzione è limitata, come le interazioni dell'utente: pertanto il *renderer* non dovrà essere efficiente come quello per gli FPS e il *Physic Engine* (la componente che simula e gestisce la fisica degli oggetti) è praticamente assente.
Osserviamo inoltre che agli albori del campo questi trade-off venivano realizzati per le limitazioni della potenza di calcolo dei dispositivi hardware. Tuttavia, oggigiorno sono presenti schede dedicate all'elaborazione della grafica in buona parte dei computer commerciali, e la potenza di calcolo delle CPU è aumentata di molto. Ciò si traduce nella possibilità di usare ad esempio *renderer 3D* anche laddove una volta si evitava per potersi concentrare sulle funzionalità principali dei giochi, come nei giochi di strategia moderni (League of Legends, Clash of Clans) dove l'utente può interagire in tempo reale e fruire di un'esperienza grafica di buona qualità.

1.2.1 Componenti di un Game Engine

Strutturalmente, un game engine è un sistema software complesso e costituito da diverse parti che cooperano tra loro. In figura possiamo vedere una suddivisione per livelli, dalle componenti hardware fino all'interfaccia che vede l'utente.

Descriviamo di seguito brevemente alcune di queste componenti.

Renderer Engine La gestione della computer grafica di un gioco è compito di questa componente.
Possiamo distinguere più livelli. Il primo è il *low level rendering engine* che disegna tutte le forme geometriche, fornisce un'astrazione dell'area di rendering (*viewport*) con associata una matrice camera-world e parametri di proiezione, gestisce gli *shader*, ovvero la componente che controlla i livelli di luce, ombra e colore in un oggetto, e implementa il codice per la gestione dei dispositivi grafici (tramite l'uso di API come OpenGL e DirectX).

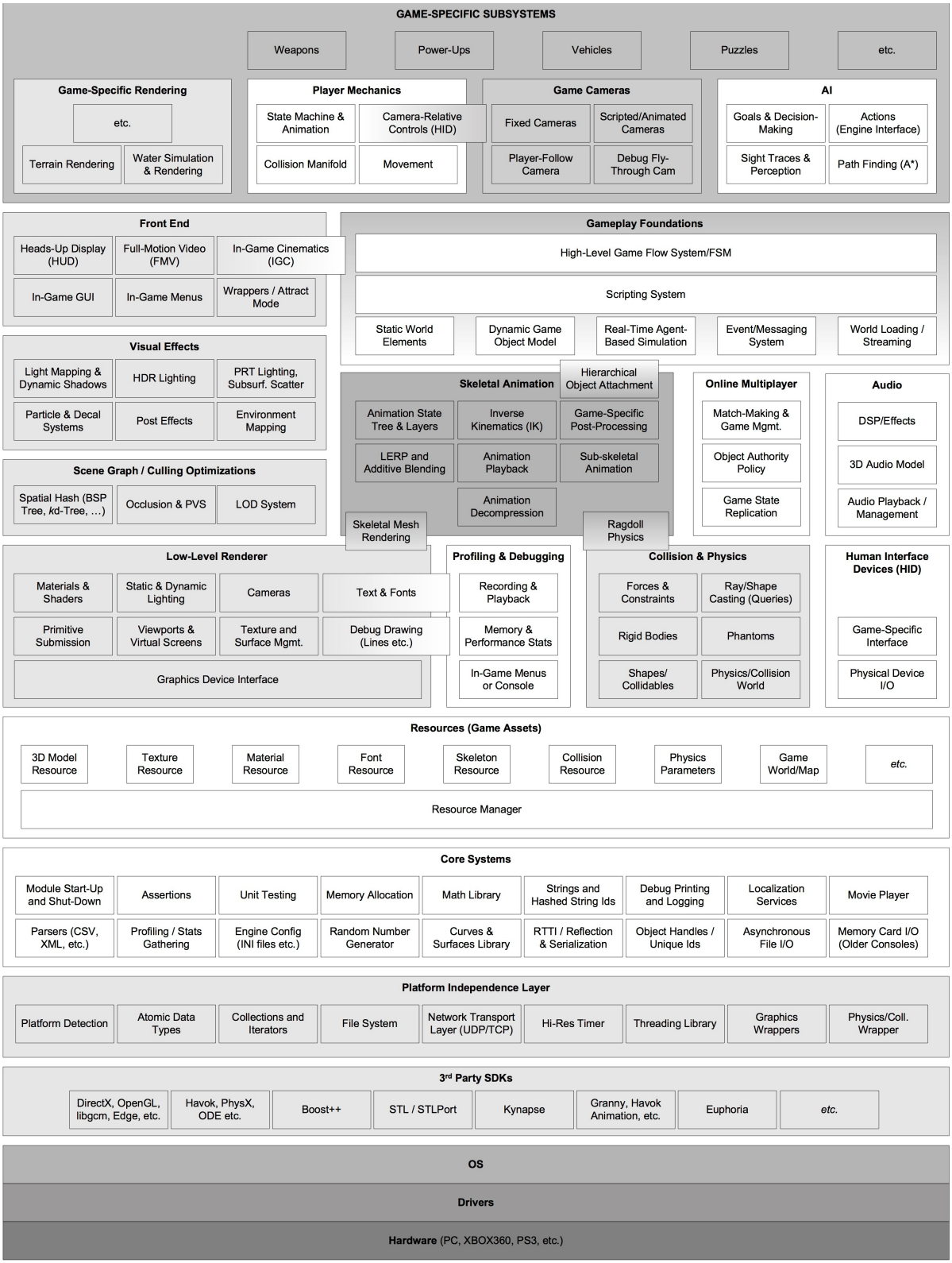

Figura 1.1: Architettura di un runtime engine, fonte [21]

Per ottimizzare il numero di oggetti da inviare al renderer è necessario usare un metodo per determinare quali oggetti sono visibili e che quindi devono essere elaborati: per questo obiettivo bisogna implementare una struttura dati che realizzi una suddivisione spaziale (ad esempio tramite una BSP tree o un kd-tree).
I restanti due livelli servono per la gestione degli effetti visivi (fumo, fuoco, ombre, mappatura delle luci) e il front-end (GUI per i menù del gioco o tool di sviluppo).

Physic Engine Questa componente definisce le regole fisiche del mondo virtuale, ed è in genere accoppiato ad un sistema di *Collision Detection* per evitare che gli oggetti si intersechino tra loro. Viene di solito realizzato utilizzando SDK di terze parti, sia proprietari come *Havok* e *PhysX* (Nvidia), sia open source come ODE (Open Dynamic Engine).

Audio Layer La colonna sonora è molto importante per la riuscita di un buon gioco e pertanto un sistema di elaborazione dell'audio è fondamentale. Questa componente è una di quelle che varia di più da gioco a gioco, poiché richiede un alto grado di personalizzazione a seconda dell'applicazione che si va a realizzare. Alcuni *sound engine* in uso in ambito commerciale sono XACT della Microsoft, Scream (Naughty Dog e Sony Computer Entertainment America) e SoundR!OT (Electronic Arts).

Gameplay Layer Se un gioco fosse un racconto, il gameplay in un certo senso può essere inteso come l'intreccio, ovvero l'insieme degli eventi con cui si può sviluppare una storia, tuttavia esso ha un significato ancora più ampio. Definisce l'inseme delle azioni che i giocatori possono compiere, i loro limiti, gli obiettivi. La metafora narrativa non è stata citata per caso: spesso il gameplay è realizzato come un racconto tramite un linguaggio di scripting di alto livello (possiamo citare *Blueprints* di Unreal).
Questo strato del game engine modella gli oggetti del mondo virtuale come i giocatori, gli NPC, i veicoli, gli edifici, etc. Inoltre di solito realizza un sistema di gestione degli eventi, per far comunicare le varie componenti, e un sistema per la gestione dell'intelligenza artificiale (IA) per i giocatori non reali. Attualmente uno dei middleware per la IA più usati è *Kynapse*, utilizzato in titoli di grande successo come *Mafia 3*, *Fable* e anche nell'Unreal Engine.

Human Interface Device Questa componente ha un fine semplice: la gestione dei dati riceviti in input dall'utente tramite mouse, tastiera, joystick, etc. e i dati che possono essere inviati in output dal gioco (vibrazione del controller o feedback audio).

1.3 Model Driven Approach

L'approccio scelto per svolgere questo lavoro è di tipo Model-Driven: vogliamo prima definire dei modelli astratti per definire dei giochi, e poi basandoci su questi realizzare gli strumenti di authoring.
Le motivazioni principali dietro a questa scelta sono le seguenti:

- vogliamo innanzitutto separare il dominio tecnico della programmazione da quello della progettazione del gioco;
- lo scopo di un modello è astrarre i concetti da realizzare, ciò in genere favorisce poi una rapida implementazione se viene formalizzato con sufficiente accortezza;
- l'uso dell'MDA è una pratica che si è consolidata in letteratura (come visto dalla review dello stato dell'arte), anche se scarseggiano le realizzazioni pratiche.

Come notazione, è stato scelto l'UML per la sua versatilità e lo stretto legame con l'ingegneria del software, di cui vediamo di seguito una breve descrizione.

1.3.1 La notazione UML

UML (*Unified Modeling Language*) è un linguaggio visuale, definito da uno standard, per la progettazione e rappresentazione di sistemi software. Tuttavia, è possibile adoperarlo anche al di fuori di questo contesto, ad esempio per la rappresentazione di processi nei sistemi di produzione; in altre parole è possibile usarlo per modellare qualsiasi tipo di sistema pratico esistente. Un altro punto di forza di questo linguaggio è l'esistenza di *tool* per la conversione dei modelli UML in codice, semplificando di fatto la progettazione di software complessi.
La notazione comprende nove tipi di diagrammi, raggruppabili in tre categorie: *structural modeling*, *behaviour modeling* e *architectural modeling*. Fanno parte della modellazione strutturale, ovvero delle componenti statiche del sistema, sei tipi di diagrammi:

- *Class Diagram*, usato per una visualizzazione statica dell'intero sistema (strettamente collegato con i linguaggi orientati agli oggetti), a livello di componenti e di funzionalità;
- *Object Diagram*, deriva dal diagramma di classe, di cui è una istanza in un dato momento;
- *Deployment Diagram*, hanno come scopo principale la rappresentazione dell'hardware del sistema;

- *Package Diagram*, usato per raggruppare i moduli di un programma per *namespase*
- *Composite Structure Diagram*, usato per mostrare la struttura interna di un *classifier*, ovvero di una categoria di elementi di UML con caratteristiche comuni;
- *Component Diagram*, modellano le componenti fisiche del sistema (ad esempio documenti, file, sorgenti, etc.), ed è strettamente correlato al *Deployment Diagram*.

Invece per descrivere le interazione nel sistema, si usa il *behaviour modeling*, e in particolare si usano:

- *Activity diagram*, rappresenta l'evoluzione del sistema operazione dopo operazione, dove ogni attività modella un'azione e ha come attributi condizioni, limiti, associazioni;
- *Interaction diagram*, costituito a sua volta da *Sequence Diagram* e *Collaboration Diagram*, ha lo scopo di mostrare il flusso di messaggi del sistema e le interazioni tra i vari oggetti e la loro struttura;
- *Use Case Diagram*, rappresentano le funzionalità del sistema, i suoi "attori" e le relazioni tra questi.

Nella modellazione dei SG non si fa in genere uso di tutti gli elementi che caratterizzano il linguaggio UML: principalmente si usano le componenti dell'*activity diagram* per rappresentare l'evoluzione del gioco, e le per rappresentare il layout.

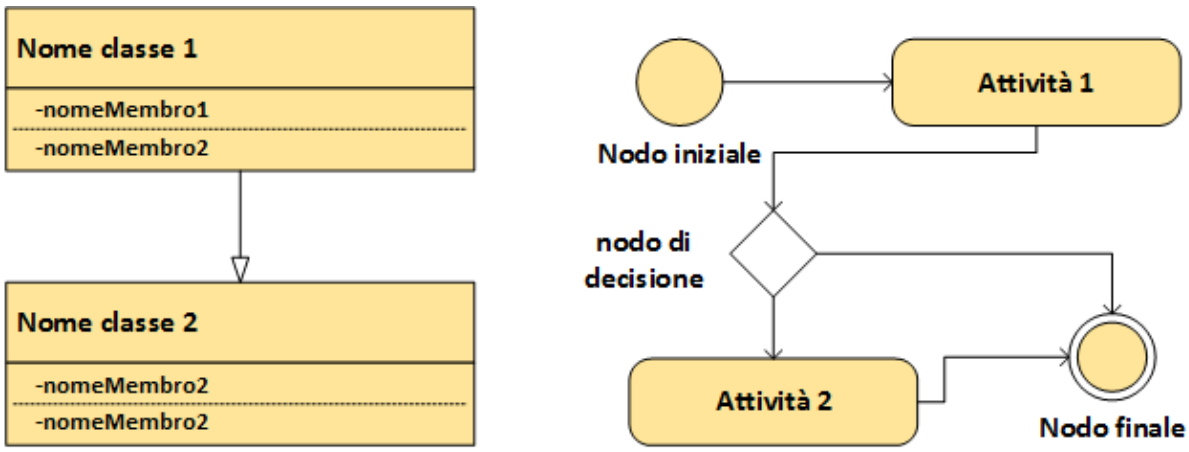

Figura 1.2: Alcuni elementi del *Class Diagram* (a sinista) e un esempio di *Activity Diagram*

Possiamo vedere in 1.2 come si presentato una *Class Diagram* e una *Activity Diagram*: vediamo come quest'ultima segue assomigli ai diagrammi a blocchi usati per rappresentare

gli algoritmi, e fa già supporre che possa essere usato per modellare la logica di un sistema.

Capitolo 2

La piattaforma E-Learn e applicazioni

2.1 Descrizione di E-Learn

E-Learn è un esempio di *Reactive Floor*, ovvero un IME (*Interactive Multimodal Environment*) dove i soggetti possono muoversi liberamente, e questi movimenti sono rilevati e catturati tramite semplici sistemi di visione. Con questo strumento si possono creare delle lezioni alternative alla didattica tradizionale, più coinvolgenti in quanto arricchiti da contenuti audio e video che vengono attivati dalla presenza degli alunni nell'ambiente. Ad esempio alzare un braccio può essere il segnale per far partire una determinata traccia audio, oppure spostarsi da un punto all'altro dell'interfaccia può far partire un video. Tra le possibili applicazioni possiamo citare la possibilità di insegnare una lingua straniera tramite l'ascolto, o supportare bambini con disabilità motorie tramite giochi che richiedono delle *gesture* prestabilite (ad esempio la *Fiaba Magica*).
Dal lato hardware, l'interfaccia è costituita da dispositivi di facile reperibilità e dal costo contenuto, ovvero speaker, proiettori, PC, webcam e amplificatori; come si può notare alcuni di questi strumenti sono già presenti in genere nelle scuole. Oltre a ciò è necessario un ambiente come una classe o una palestra, mentre al giocatore non è richiesto di indossare alcun tipo di sensore. Nella sua ultima implementazione sono stati adoperati i seguenti componenti:

- un sostegno in alluminio (4 m x 3 m) per il proiettore, gli speaker e i device di tracciamento;
- due Kinect v2 con adattatori USB;
- due speaker Genelec 8030A;
- un proiettore Optoma X305ST;

- tre computer dekstop, due per acquisire i dati dai Kinect(uno per ciascuno), e l'altro per l'elaborazione dei dati, l'esecuzione del gioco, e il controllo del sistema di feedback audio-video;
- uno switch Netgear GS208v2 per collegare i PC.

Dal lato software, è un sistema modulare costituito da elementi di input (ad esempio i dispositivi di tracciamento), componenti per l'elaborazione in tempo reale dei dati e da componenti di output. Possiamo vedere in figura 2.1 una schematizzazione di questa architettura.

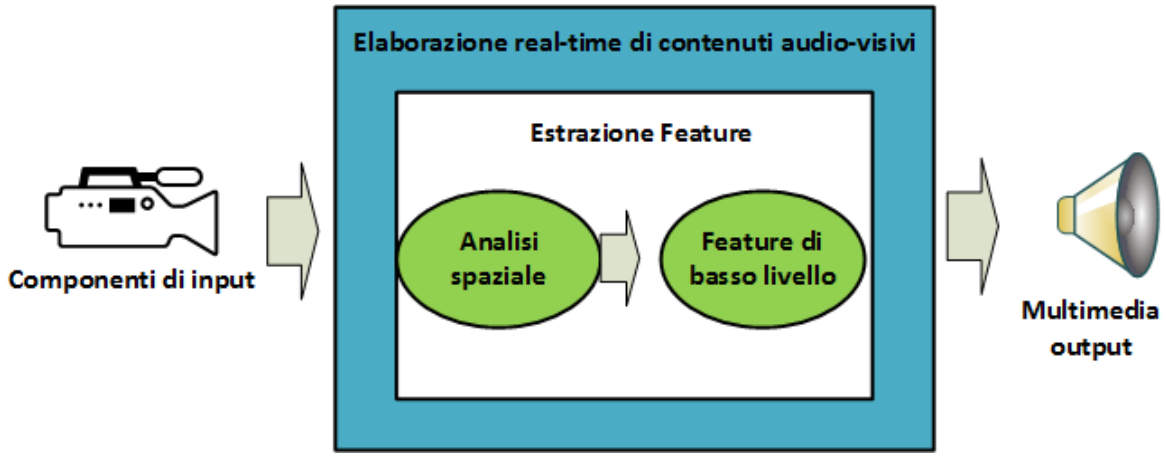

Figura 2.1: Architettura modulare dell'IME

Per ulteriori dettagli si rimanda a [57] e [31].
L'approccio scelto in questa tesi è di tipo bottom-up: partiremo da alcuni giochi già implementati per la piattaforma, per arrivare a realizzare degli strumenti per poterli creare in maniera semplice. Passeremo ora in rassegna i SG, descrivendone il funzionamento ad alto livello.

2.2 Resonant Memory

La prima applicazione che prendiamo in esame è *Resonant Memory*, un SG basato sull'ascolto e sul movimento. Lo spazio di gioco viene diviso in più aree, facendo distinzione tra quella centrale e quelle periferiche (figura 2.2): nella prima l'alunno ascolta una storia e in base alle informazioni contenute in essa si muove verso le zone più esterne, attivando rumori o musiche quando le raggiunge.

In questo modo, il giocatore può esplorare lo spazio e i suoi contenuti al fine di ricostruire la colonna sonora della narrazione.
In particolare, il gameplay del gioco si può riassumere così:

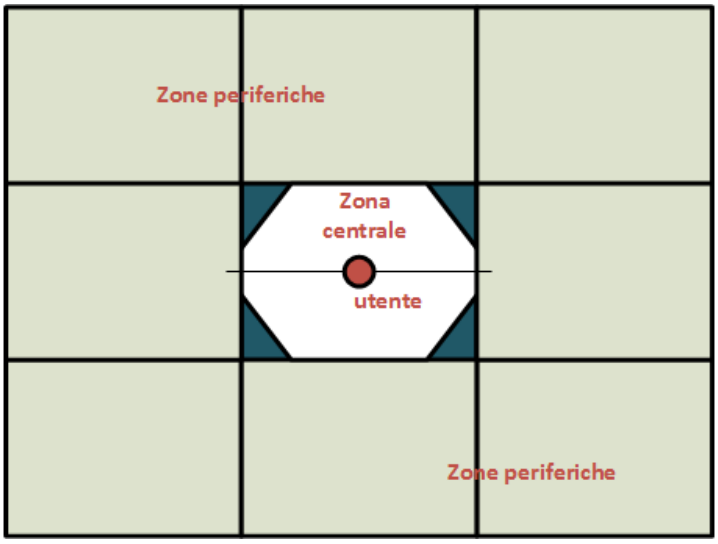

Figura 2.2: Suddivisione dell'interfaccia per *Resonant Memory*

- l'utente esplora le zone periferiche, azionando i suoni associati ad esse ed iniziando così a costruire una sorta di mappa mentale;

- spostandosi nella zona centrale l'utente aziona la storia;

- ascoltando il racconto, il giocatore si sposta nelle zone esterne e ricostruisce così la colonna sonora

2.3 Fiaba Magica

Come già accennato, la *Fiaba Magica* è un'applicazione pensata per i bambini che soffrono di handicap fisici. Grazie alla "Fiaba", possono ascoltare un racconto in maniera interattiva: l'idea di base è di associare una traccia audio e una immagine ad un movimento prestabilito. Come in *Resonant Memory*, lo spazio è suddiviso in aree e spostandosi o tramite una *gesture* si attivano gli spezzoni successivi del racconto. La dinamica di questo SG si articola quindi in due fasi: entrando in una determinata zona dell'interfaccia si attiva una parte di una storia; dopo la fine del file audio, viene proiettata una immagine, con la quale l'utente può interagire, animandola muovendo un braccio di lato.
Tra gli obiettivi di questa applicazione possiamo elencare la trasmissione di concetti spaziali legati all'orientamento e l'aumento della consapevolezza delle parti del corpo in relazione a oggetti, altre persone e la propria posizione nell'ambiente.

2.4 Harmonic Walk

Una progressione armonica è una sequenza di corde musicali poste in un certo ordine. *Harmonic Walk*[30] è stata pensata per far comporre una tale successione a soggetti senza esperienza o capacità in ambito musicale. Per fare ciò si sfrutta l'intuizione che la "progressione" suggerisce di per se l'idea di una sequenza di passi da seguire; trasferendo ciò su uno spazio fisico, tramite l'ascolto di diverse regioni armoniche e la loro collocazione, si può far costruire al soggetto una mappa cognitiva delle loro relazioni, con la possibilità di arrivare a comporre una progressione se si segue un determinato percorso.
Anche qui, lo spazio è suddiviso in varie regioni, corrispondenti a diverse corde che si attivano quando si attraversa una zona sensorizzata. All'inizio l'utente esplora lo spazio e inizia memorizzare i *landmark*, cioè le corde, la loro posizione e il percorso fatto, iniziando così a costruire la mappa cognitiva. Dopo questa fase di esplorazione, è richiesto all'utente di costruire una o più sequenze armoniche. Consideriamo ora un caso particolare dell'*Harmonic Walk*, lo stesso considerato in [30].

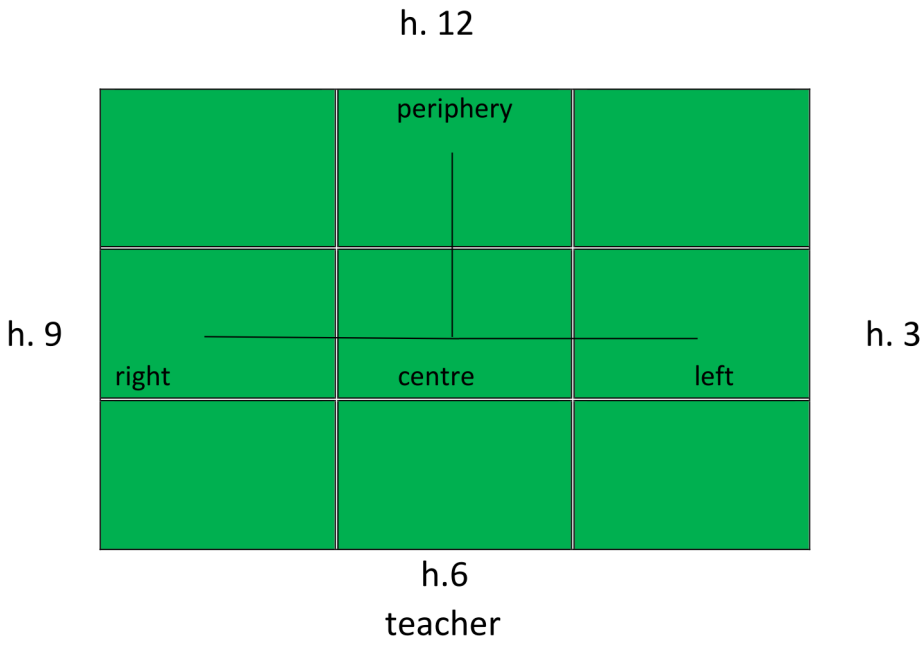

Figura 2.3: Suddivisione della superficie di gioco dell'*Harmonic Walk* suddiviso in nove zone secondo lo schema a "T"

Vediamo in figura 2.3 come la superficie dell'interfaccia di gioco sia divisa in nove, dove si è applicato uno schema spaziale a "T". Ciò implica che vi sono solo quattro zone attive, e le altre sono "vuote"; in questo modo si possono scegliere due progressioni armoniche:

- c# minore (periphery), C maggiore (centre), F maggiore e G maggiore (rispettivamente per le scelte *right* o *left*);
- C maggiore (periphery), C minore(centre), Ab maggiore e Eb maggiore (rispettivamente per le scelte *right* o *left*).

La procedura del gioco si articola dunque in questo modo:

- l'insegnante avverte che ci sono quattro corde nascoste in quattro zone e invita gli alunni ad esplorare l'interfaccia;
- quando ne vengono trovate almeno due, viene loro chiesto di individuare un percorso tra la aree e di suonarlo ripetutamente;
- se è stato trovato il primo cammino, viene chiesto di individuarne un secondo;
- infine, viene chiesto di scegliere uno dei percorsi trovati e di suonarlo ripetutamente.

2.5 Good or Bad?

Good or Bad?[31] è un esempio di puzzle musicale, l'applicazione più complessa che prenderemo in esame: tramite l'ascolto di alcuni pezzi di un brano si cerca di ricostruirlo attraverso il movimento e l'ascolto. Tuttavia gli spezzoni non sono la mera suddivisione nel tempo del brano originale: la ricostruzione si ottiene infatti tramite la sovrapposizione dei vari pezzi del puzzle che costituiscono una divisione per livelli orizzontali della traccia. Per esempio si possono prendere le stesse parti suonate da strumenti musicali diversi. Questo gioco richiede quindi un certo livello di abilità cognitive nell'elaborazione della musica e capacità di discriminazione di vari elementi.
In particolare, si hanno due gruppi di brani, quattro ciascuno, e il tappeto è diviso in due da una linea. Nel lato sinistro si trovano i brani ordinati casualmente all'inizio della partita; in quello destro solo due pulsanti. Il primo giocatore sceglie uno spezzone e il secondo deve decidere se tenerlo o scartarlo (rispettivamente salvando o consumando una vita). Tuttavia le varie parti non sono la mera suddivisione nel tempo del brano originale: la ricostruzione si ottiene infatti tramite la sovrapposizione dei vari pezzi del puzzle che costituiscono una divisione per livelli orizzontali della traccia. Per esempio si possono prendere le stesse parti suonate da strumenti musicali diversi. Questo gioco richiede quindi un certo livello di abilità cognitive nell'elaborazione della musica e capacità di discriminazione di vari elementi. Per terminare il gioco il secondo giocatore deve usare meno di tre vite per finire la composizione.

Capitolo 3

Analisi e modellazione

In questo capitolo descriveremo con maggiore dettaglio alcuni dei modelli trovati in letteratura e li valuteremo provando ad applicarli sui giochi di nostro interesse.

3.1 MoPPLiq

Presentiamo in questa sezione MoPPLiq, un formalismo per la descrizione dei serious game, già citato nel primo capitolo in [34].
Il nome è ottenuto come acronimo francese di *Model for Gaming Aspects and Educational Aspects of SG Scenarios*, un modello influenzato dai concetti del IMS-LD (*IMS Learning Design*)[25], framework molto usato per la schematizzazione di *learning object*, e risulta esserne una derivazione semplificata. Per rappresentare l'evoluzione del gameplay senza entrare nei dettagli di un particolare caso in esame, MoPPLiq divide il scenario in varie componenti, dette *activity*. Queste sono semplicemente dei *blackbox*, e ognuna è caratterizzata da un *goal*, e degli elementi di input e di output. Tramite gli obiettivi è possibile definire gli scopi didattici e ludici da raggiungere.
Diversi elementi di output servono a modellare non-linearità nell'evoluzione del flusso del gioco: ad esempio, se un giocatore è costretto ad una scelta tra due opzioni, una corretta ed una sbagliata, un eventuale errore può essere causato dalla mancanza delle giuste conoscenze o lacune nella preparazione dell'utente; in casi come questi si può predisporre una activity di supporto per aiutare il giocatore. Allo stesso modo avere più input aiuta ad adattare le activity alle scelte e alle conoscenze del *serious-player*. Inoltre, è possibile collegare gli obiettivi con gli output, in modo da far iniziare una nuova attività sole se sono raggiunte determinate condizioni. Queste ultime possono essere solo di due tipi, ovvero "if present" or "if not present", in modo da semplificare la modellazione.
Graficamente, il modello usa la notazione UML (figura 3.1): abbiamo due cerchi per indi-

care l'inizio e la fine dell'applicazione; dei rettangoli rappresentano le *activity*; da queste escono gli input e gli output, rispettivamente a sinistra e a destra.

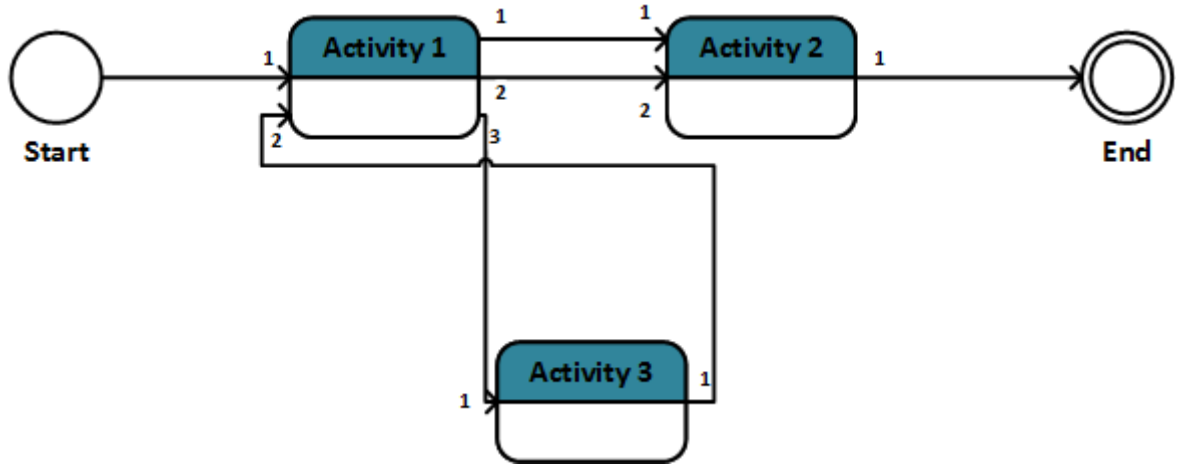

Figura 3.1: Esempio degli elementi grafici di *MoPPLiq*

Su questa base è stata anche realizzata un'applicazione web, APPLiq, che presenta un'interfaccia grafica per creare i giochi e un meccanismo di controllo delle inconsistenze della logica del gioco. Si rimanda sempre a [34] per ulteriori dettagli.
Evidenziamo la caratteristica principale di questo modello: la dinamiche interne di ogni singola attività sono nascoste, pertanto si può supporre che questa fase dovrà essere svolta da un'altra componente dell'authoring tool.

3.1.1 Rappresentazione di "Resonant Memory"

Come visto, MoPPLiq usa delle *blackbox* per modellare gli scenari dei SG, supponendo quindi la realizzazione di un software che metta a disposizione dei blocchi che dovranno essere tradotti in dati processabili da un computer per creare in automatico il gioco. Ad esempio, per rappresentare *Resonant Memory* possiamo definire tre attività, la prima è definita dalla seguenti caratteristiche:

- è l'*activity* iniziale e il giocatore esplora l'ambiente, e l'obiettivo è memorizzare i suoni e le zone associate ad esse;
- l'output è rappresentato dal movimento verso la zona centrale che fa iniziare la storia.

La seconda *activity* è invece definita da questi elementi:

- l'obiettivo è di far partire la riproduzione della storia;

- l'input è rappresentato dall'ingresso del giocatore nella zona centrale (l'output dell'attività precedente);
- si esce da questo scenario quando l'utente esce dalla zona centrale per comporre la colonna sonora.

Infine l'ultima:

- l'obiettivo è di comporre la colonna sonora seguendo il racconto;
- l'input è rappresentato dall'uscita del giocatore nella zona centrale;
- l'output è la somma dei brani composti.

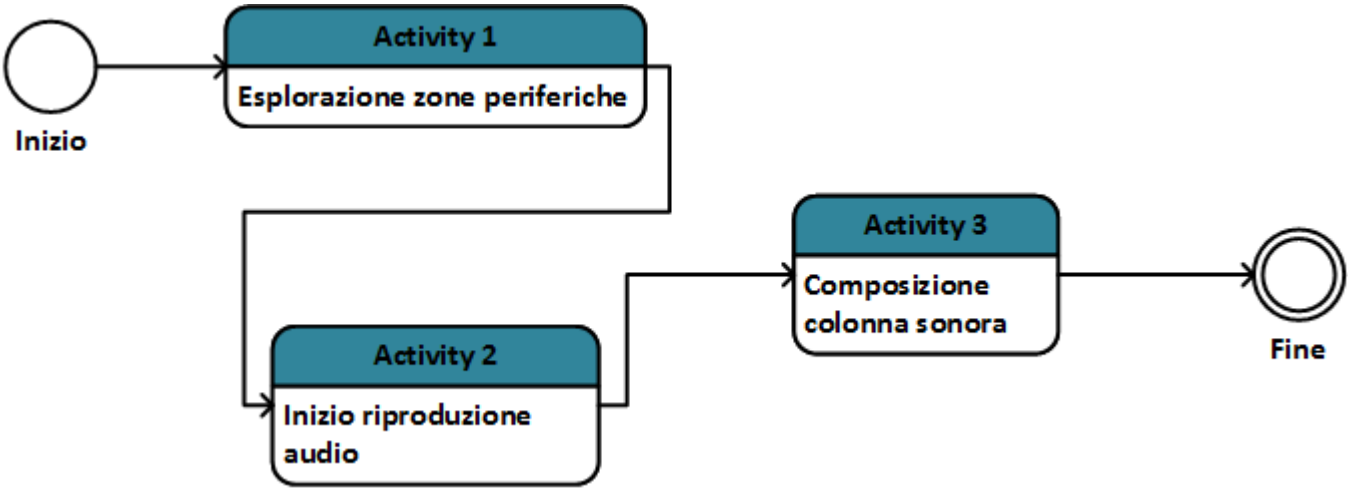

Figura 3.2: Resonant Memory con MoPPLiq

In figura 3.2 vediamo il risultato ottenuto.

3.1.2 Rappresentazione di "Fiaba Magica"

Abbiamo già visto nel capitolo precedente la *Fiaba Magica*, un'applicazione che usa il movimento per narrare una storia e aiutare in questo modo ad aumentare la percezione delle parti del corpo nei bambini con disabilità motorie.
Al fine di applicare *MoPPLiq*, come nel caso precedente possiamo scomporre l'applicazione in due attività: una per l'ascolto della storia, e una per rappresentare il gioco delle animazioni. Queste due attività si alternano più volte, fino al completamento della storia (figura 3.3). L'interazione si svolge tramite movimenti in zone specifiche e *gesture*, pertanto questi possono essere considerati gli input e gli output dei vari scenari e dovranno essere definiti a seconda del giocatore.

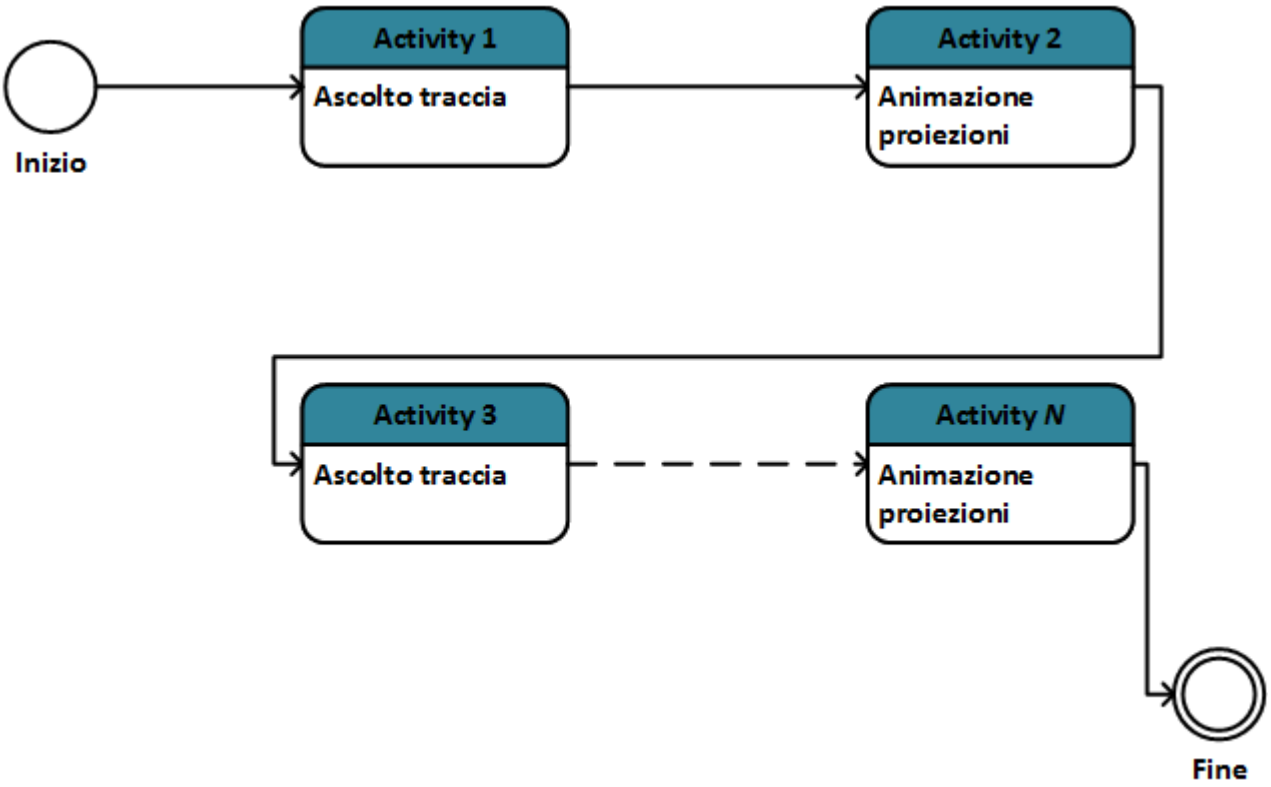

Figura 3.3: Fiaba Magica con MoPPLiq

3.1.3 Rappresentazione di "Harmonic Walk"

Riprendiamo l'esempio di Harmonic Walk visto nel capitolo precedente. Osserviamo che rispetto ai casi precedenti è più arduo suddividere il gioco in attività: la prima idea potrebbe essere quella di definire quattro componenti, una per ognuna delle fasi sopra citati; ma notiamo subito che non abbiamo un chiaro esito di questa *activity* che suggerisca il completamento dei suoi obbiettivi (ad esempio in *Resonant Memory* avevamo una specifica traccia da trovare).
Tuttavia, considerate le task da svolgere (ascolto, movimento e), è sufficiente una *activity* dove l'utente esplora l'interfaccia e fa suonare le zone sensorizzate, e poi le ripercorre per definire i cammini trovati (figura 3.4).

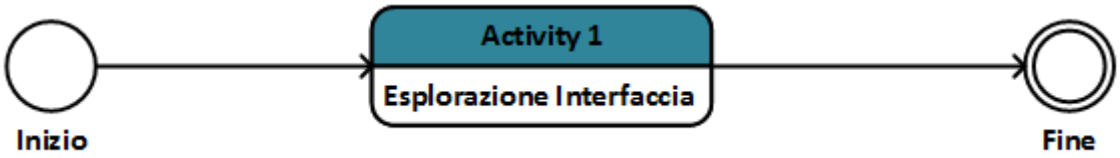

Figura 3.4: Schema per la modellazione di *Harmonic Walk*

3.1.4 Rappresentazione di "Good or Bad"

In *Good or Bad* due giocatori tentano di comporre un brano, cercando quattro traccie che dovranno essere sovrapposte per ottenere il risultato finale. Ad ogni passo del gioco il primo utente sceglie tra due brani, mentre il secondo decide se è compatibile con quanto ascoltato fino ad allora. Se si individuano quattro livelli corretti, il sistema riproduce la composizione. É inoltre presente un sistema di gestione delle vite: all'inizio se ne hanno tre, ma ad ogni scelta sbagliata se ne perda una.
Possiamo vedere in figura 3.5 una possibile rappresentazione. Sono stati definiti in totale quattro tipi di attività: due rappresentano i due giocatori e le scelte che devono compiere; una per la gestione delle composizioni e delle vite, che ha quindi il compito di decidere della bontà delle azioni degli utenti; e infine una per l'eventuale ascolto del brano finale. Il modello è in figura 3.5 è incompleto, poiché a seconda delle scelte dei giocatori e delle vite consumate il gioco può avere lunghezza diversa in termini di numero di *activity*. E questa risulta una criticità nella rappresentazione del gioco.

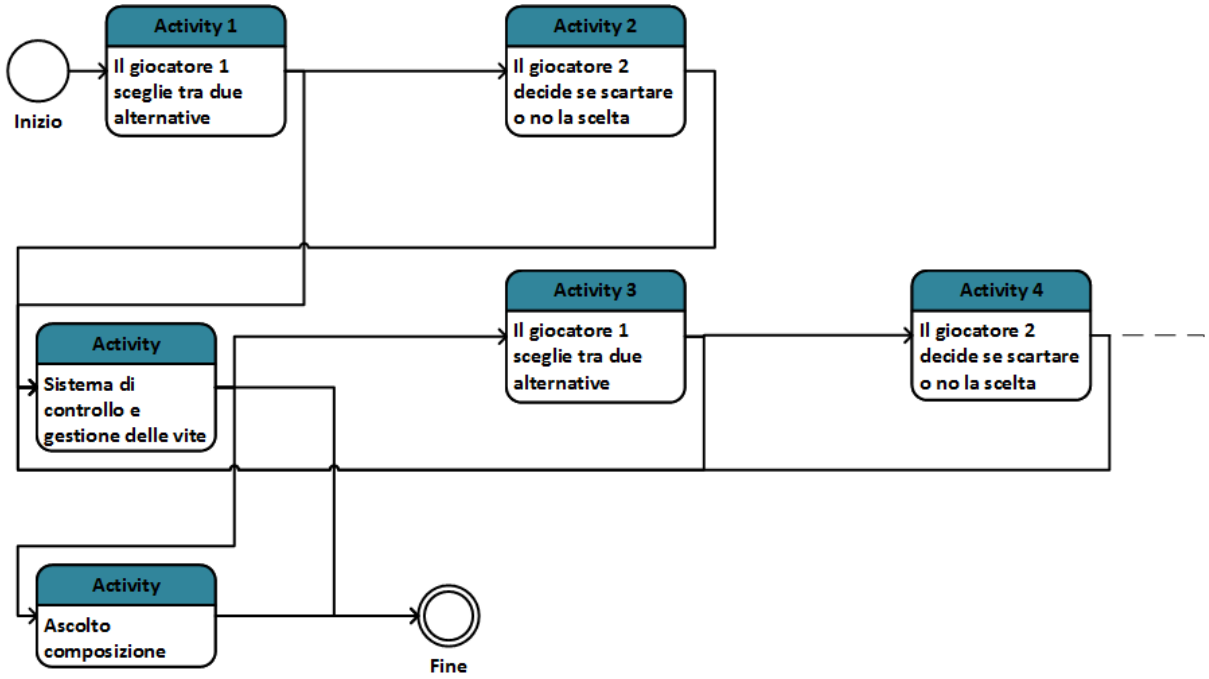

Figura 3.5: Possibile schema per *Good or Bad*

Per riuscire a modellare il gioco nella sua interezza bisogna impiegare le *activity* in maniera diversa. Nella figura 3.6 vediamo che l'attività per il controllo delle vite decide se far continuare il gioco al primo giocatore, entrando quindi in un loop fino a che non si vince o si perde.

Tuttavia il modello rimane comunque problematico: avere un sistema di *authoring* a blocchi implica che quest'ultimi siano delle componenti abbastanza generali da poter

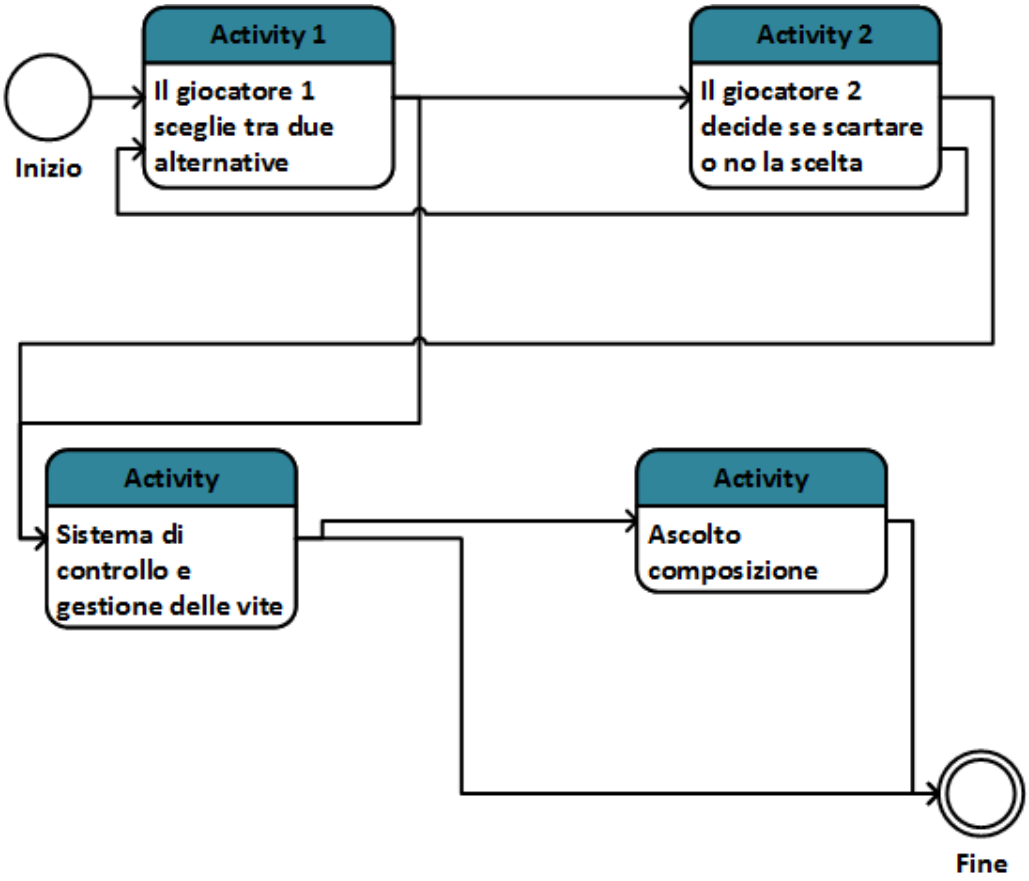

Figura 3.6: Schema alternativo per *Good or Bad*

essere condivisi da diversi giochi, come in effetti avviene nelle applicazioni viste finora. In questa ottica una attività per la gestione delle vite non sembra essere plausibile, essendo inoltre una task che deve svolgere il PC, senza l'interazione dell'utente. Pertanto la rappresentazione più corretta con MoPPLiq si può ottenere togliendo questa activity (fig. 3.7), dove per modellare il sistema di gestione delle vite possiamo aggiungere una condizione in uscita che controlli che il numero di vite sia sufficiente per continuare il gioco.

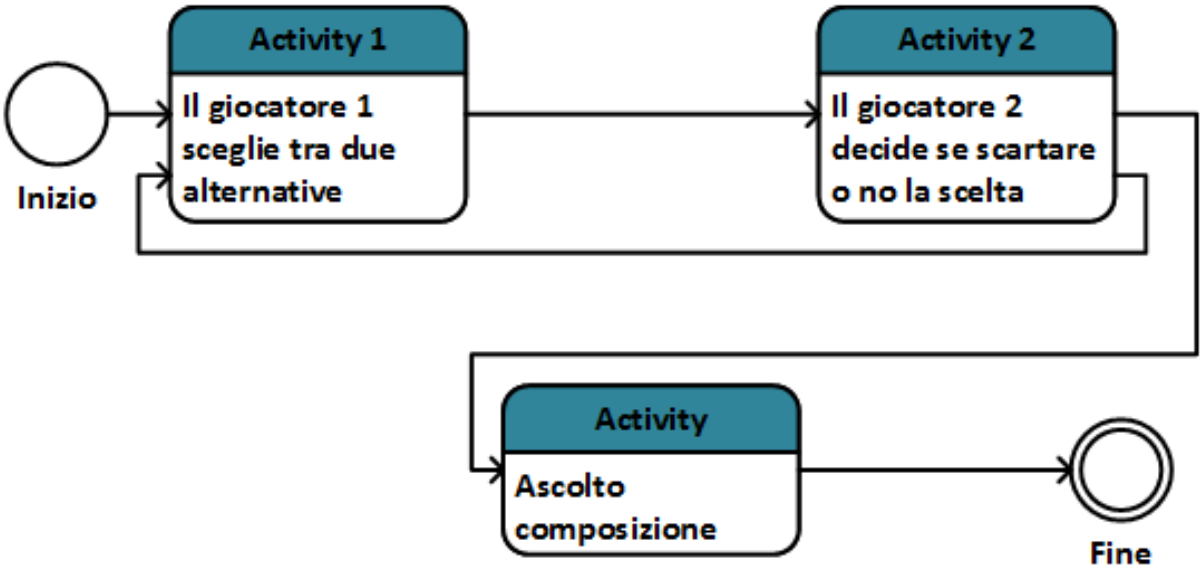

Figura 3.7: Schema corretto per *Good or Bad*

3.1.5 Descrizione di una activity in XML

Per poter creare un'applicazione da un modello UML è necessario poter descriverne le componenti in un formato elaborabile da un computer. A tal proposito, MoPPLiq prevede di usare dei file XML per codificare i modelli, dove possiamo identificare i seguenti elementi: le *activity* che a loro volta contengono gli elementi *name, description* e *input states*, dove sono indicate le condizioni di input e output; i collegamenti delle activity sono indicati all'interno dei tag *links* e in *goals* troviamo indicati gli obiettivi e le connessioni associate ad esse. Possiamo vedere ora un esempio parziale per *Resonant Memory*.

```
<activity id_activity="1">
    <name>Esplorazione interfacce</name>
    <description>Permette di muoversi nella piattaforma e scoprire i
        contenuti multimediali ad esso associati</description>
        <input_states>
```

```
    <input_state id_input="1"   />
    <output_state id_output="2"/>
        </input_states>
</activity>

<activity id_activity="2">
    <name>Inizio riproduzione audio</name>
    <description>Inizia la riproduzione della storia</description>
    <input_state id_input="3"/>
    <output_state id_output="4"/>
</activity>
[...]
<links>
    <output_input_link id_output="2" id_input="3" />
[...]
</links>

<goals>
    <goal id_goal="7" type="ludo">
        <name>
                Memorizzare coordinate spaziali e contenuti associati
                </name>
        <goal_links>
            <goal_link object="output_state" id_object="2"/>
[...]
        </goal_links>
    </goal>

    <goal id_goal="24" type="ludo">
        <name>Movimento nella zona centrale</name>
        <goal_links>
            <goal_link object="output_state" id_object="3"/>
        </goal_links>

    </goal>
    <goal id_goal="38" type="ludo">
        <name>Far iniziare la riproduzione</name>
```

```
        <goal_links>
            <goal_link object="output_state" id_object="4"/>
        </goal_links>
    </goal>
[...]
</goals>
```

3.2 GLiSMo

GLiSMo(*Serious Game Logic and Structure Modeling Language*) è un modello nato da un'analisi dei requisiti dei serious game e dei fattori che li influenzano[50]. In particolare sono stati individuati sette categorie: *learning*, *restriction*, *communication*, *assistance*, *goals*, *adaptation* and *representation*.
Il primo insieme definisce i requisiti didattici di un SG, come ad esempio la necessità di un obiettivo pedagogico, di meccanismi per acquisire nuove conoscenze o abilità, e anche di task per migliorare la ritenzione di ciò che si è appreso. Con *restriction* invece si vogliono indicare i limiti dei SG, sia quelli dovuti al tipo di piattaforma sui cui si gioca (ad esempio spazio esplorabile, movimenti leciti, etc.) sia quelli propri del gioco (ogni attività ludica si basa su un regolamento da rispettare). Inoltre, deve essere possibile per i giocatori interagire tra di loro e ciò è messo in evidenza nella categoria *communication*. Per riuscire a portare avanti il gioco è necessario avere un sistema di feedback (*assistance*) con cui i *serious player* possono interagire e ricevere informazioni e con cui l'insegnante possa avere dei dati sulle performance. Un'altra caratteristica importante dei SG è la definizione di compiti chiari e risolvibili (*goals*) dai giocatori, le cui abilità e conoscenze tuttavia possono variare, da cui deriva il requisito dell'adattività (*adaptation*). Infine, la categoria *representation* mette in evidenza come gli la conoscenza o le competenze da acquisire devono essere rappresentati in maniera accattivante, per esempio tramite l'utilizzo di contenuti multimediali o sensori multimodali.

3.2.1 Elementi strutturali

Descriveremo ora le componenti che definiscono la struttura del gioco. Il come questi elementi interagiscono tra di loro verrà descritto nel paragrafo successivo. Innanzitutto abbiamo il *Serious Game Root*: esso è la radice di tutti gli altri elementi, e ha come proprietà che ogni oggetto collegato direttamente o indirettamente ad esso appartiene allo stesso SG. Il modello definisce gli elementi *Act* e *Scene*: ogni gioco è composto da macro parti detti atti, caratterizzati da degli obiettivi; a loro volta questi possono avere

più scene, che descrivono un luogo specifico del mondo rappresentato dal SG, e sono definiti dagli attributi *name, position* e *scale*.
Per descrivere gli eventuali oggetti che popolano l'ambiente virtuale del gioco con cui il giocatore può interagire, GLiSMo usa l'elemento *Object*, che possiede gli stessi attributi di una *Scene*. Alcuni *item* del gioco possono essere raccolti dai giocatori, sia quelli fittizi che quelli reali, sono descritti dall'elemento *Character*: la distinzione tra i due tipi di personaggi viene fatta con un attributo booleano, *isNPC* (*Non Player Character*).
Per tener traccia delle performance del giocatore e per gestire i feedback sono definiti le componenti *Reward Manager* e *Feedback Manager*: il primo gestisce score e premi, mentre il secondo mostra una valutazione dei traguardi raggiunti.
Infine, abbiamo gli elementi *Audio Manager* e *Video Manager* per gestire i contenuti audio visivi. Tra gli attributi abbiamo il sorgente del file (*videoSrc* e *audioSrc*), e *isLoop* per indicare se la riproduzione deve avvenire in loop. Per gestire l'interfaccia grafica del gioco è definita la componente *GUI Manager*, che ha il compito di mostrare immagini, tasti, punteggi, menù, etc.

3.2.2 Elementi Logici

Finora abbiamo mostrato le componenti per modellare gli elementi che caratterizzano un gioco. Mancano le azioni dei giocatori, gli eventi e i progressi che portano a raggiungere gli obiettivi prefissati dal SG.
Sono definiti uno stato iniziale e uno finale, indicati da cerchi neri (notazione UML). Si definisce inoltre l'elemento *Action*, con degli attributi che descrivono le azioni che possono svolte dal giocatore (ad esempio *select object, put object*). Per rappresentare gli obiettivi da raggiungere vi è la componente *task*, collegato con *assessment*, che valuta invece i risultati ottenuti. Le varie componenti sono collegate tramite frecce (*continuous arrow*), che definiscono quindi il flusso da seguire per completare l'obiettivo. Nella fase di *assessment* si decide se è stato raggiunto e si termina il gioco.

3.2.3 Modifiche preliminari a GLiSMo

Per modellare le nostre applicazioni dobbiamo quindi costruire due schemi ma prima è necessario fare delle modifiche al formalismo originale poiché esso è stato pensato per giochi *point-to-click* mentre noi dobbiamo descrivere giochi a interazione multi-modale.
Facciamo quindi le seguenti assunzioni:

- l'elemento *object* può rappresentare anche file multimediali in quanto nel nostro ambiente si interagisce principalmente con questo tipo di oggetti non tangibili e pertanto definiamo un attributo *media* dove si può indicare il percorso di un file;

- definiamo l'interazione *movement* per rappresentare uno spostamento da una zona sensorizzata a un'altra;
- definiamo l'interazione *explore* per indicare la possibilità di esplorare l'intero spazio di gioco;
- definiamo l'interazione *gesture* per indicare uno specifico gesto usato per relazionarsi con l'ambiente;
- tra i possibili task introduciamo la ricostruzione di un brano.

3.2.4 Rappresentazione di Resonant Memory

Innanzitutto si definisce la struttura del gioco, ovvero una interfaccia divisa in nove zone, a cui sono associate delle tracce audio: definiamo quindi un *act* e nove *scene*, a cui corrispondono altrettanti elementi *Object* per indicare i brani; infine si definisce un *character* che rappresenta il giocatore. In figura 3.8 vediamo il risultato ottenuto.
Ora, possiamo costruire lo schema logico (fig. 3.9) che permette di definire le azioni e le task. Siccome l'interazione in questa applicazione avviene tramite il movimento, nel modello troviamo solo un tipo di *action*, che permette di esplorare lo spazio di gioco. La task da compiere è ricostruire la colonna sonora e pertanto abbiamo un elemento di questo tipo, dove osserviamo che non è stato definito il tipo di in quanto non ne è previsto uno nel formalismo di GLiSMo per descrivere applicazioni di questo tipo. Infine definiamo una componente di *assessment* per verificare che l'obiettivo sia stato completato (ciò si può supporre che richieda l'intervento esterno dell'insegnante).

3.2.5 Rappresentazione di Fiaba Magica

Per questa applicazione si è scelto di modellare il caso in cui la storia sia divisa in due parti. La struttura del gioco (fig. 3.10) risulta quindi essere costituita da due *act*, uno per parte, a cui corrispondono una *scene* ciascuno dove si trovano una traccia audio e una immagine, quindi due *object*. Per l'interazione del giocatore si suppone di avere due tipi di azione: il movimento verso una zona e le *gesture* per giocare con le proiezioni (fig. 3.11).

3.2.6 Rappresentazione di Harmonic Walk

Per riuscire a definire *Harmonic Walk*, riprendiamo l'esempio già adoperato per MoPPLiq. Abbiamo quindi un solo *act* e cinque *scene* corrispondenti alle zone con le corde (fig. 3.12).

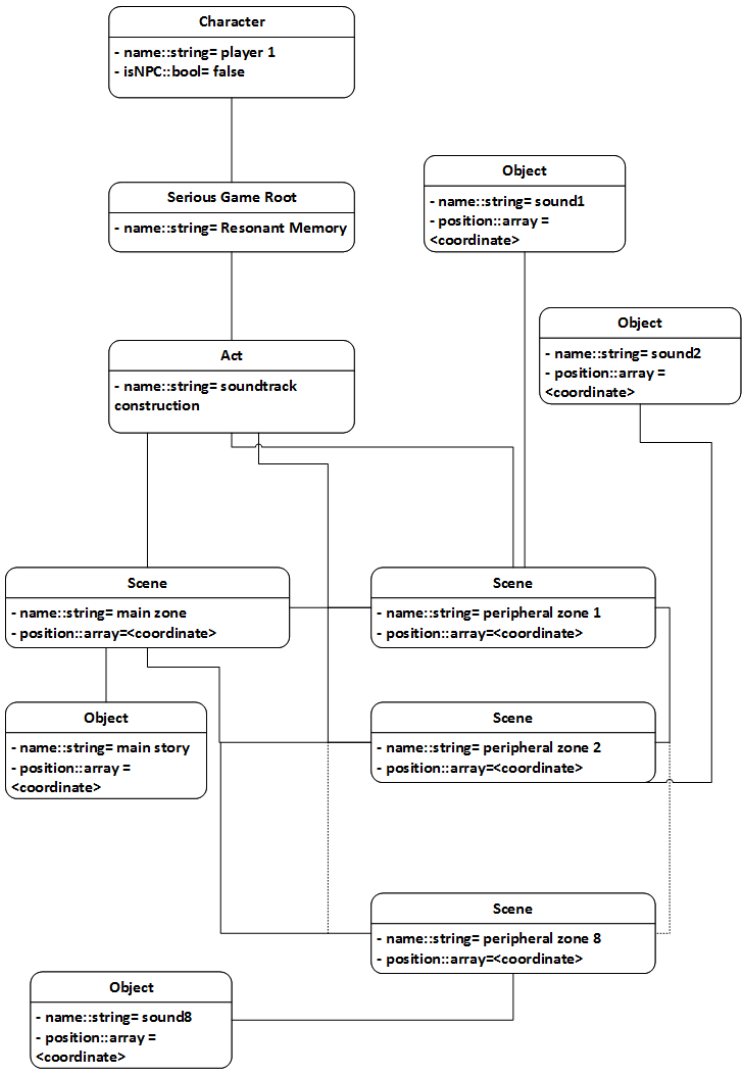

Figura 3.8: Struttura di Resonant Memory con GLiSMo

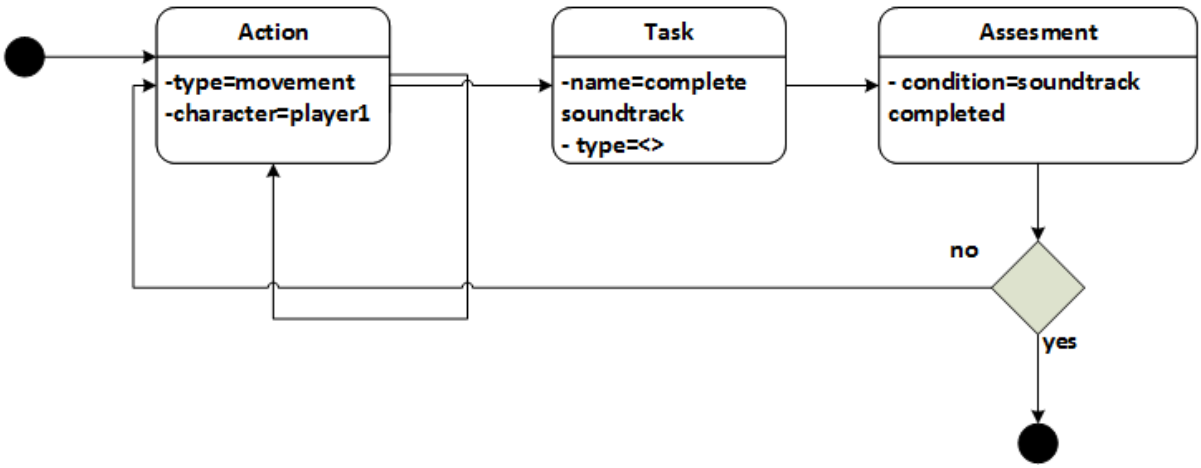

Figura 3.9: Modello logico di Resonant Memory con GLiSMo

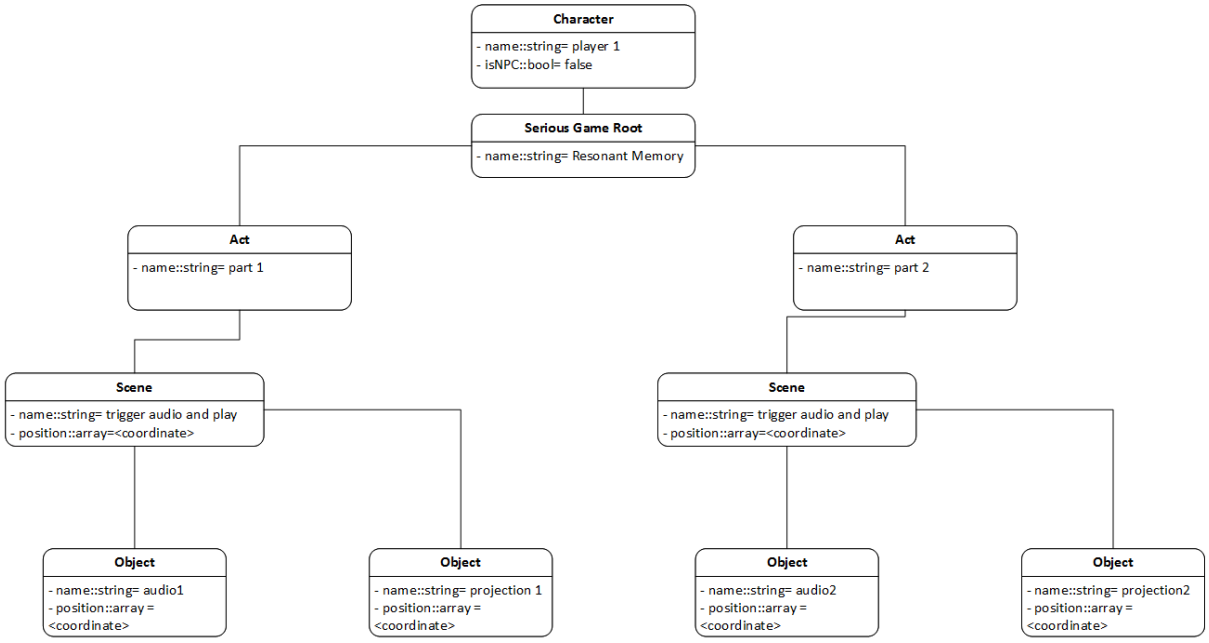

Figura 3.10: Struttura di Fiaba Magica con GLiSMo

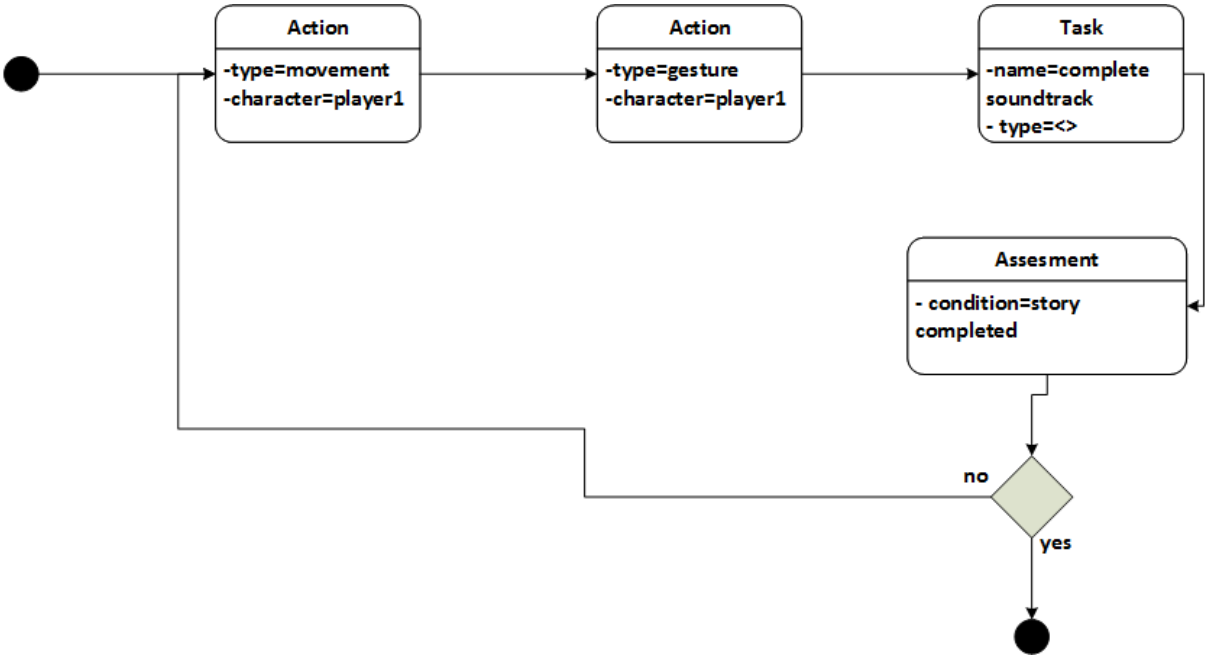

Figura 3.11: Modello logico di Fiaba Magica con GLiSMo

Come per *Resonant Memory* per interagire si usa solo il movimento e il modello logico risulta analogo (3.13).

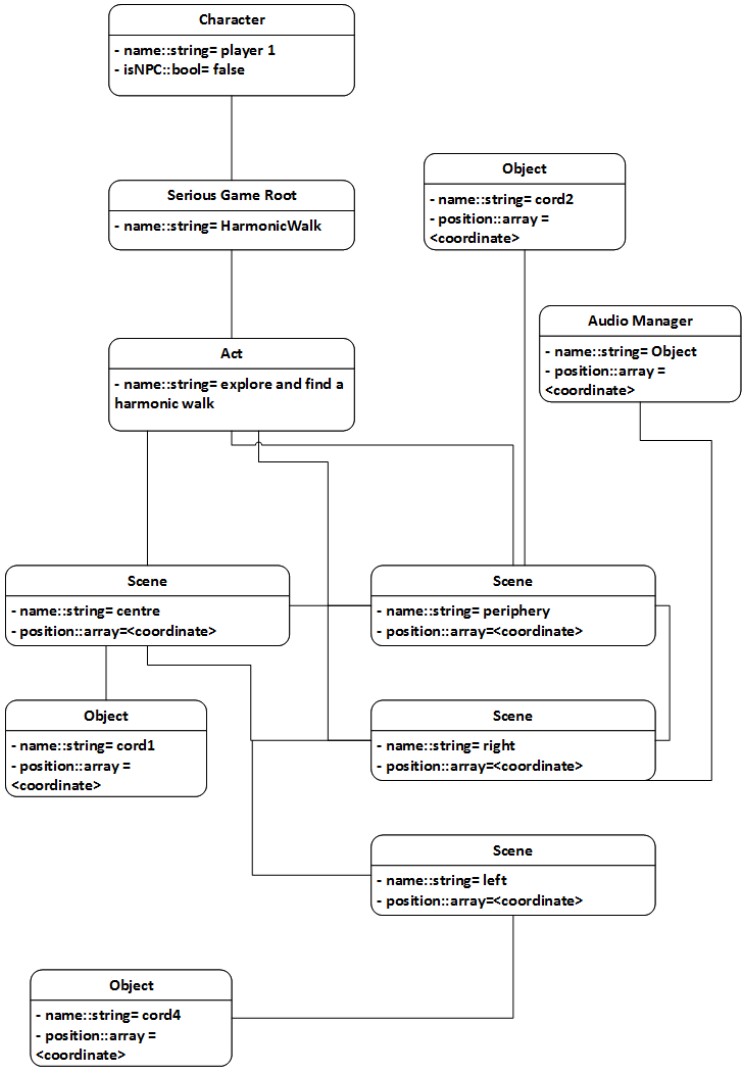

Figura 3.12: Struttura di Harmonic Walk con GLiSMo

3.2.7 Rappresentazione di Good or Bad?

In figura 3.16 possiamo vedere il layout di *Good or Bad?*. Sono definiti tre *act*: il primo rappresenta la porzione dell'interfaccia dedicata al primo giocatore, che sceglie i brani posti in otto *scene*; nel secondo il secondo giocatore sceglie se scartare o tenere la traccia scelta, e pertanto abbiamo due *scene*; nel terzo abbiamo infine la riproduzione del brano se viene composto correttamente.
Inoltre, notiamo l'elemento *Reward Manager* per tener conto del punteggio e quindi può essere usato per gestire le vite a disposizione dei giocatori. Abbiamo anche due elementi

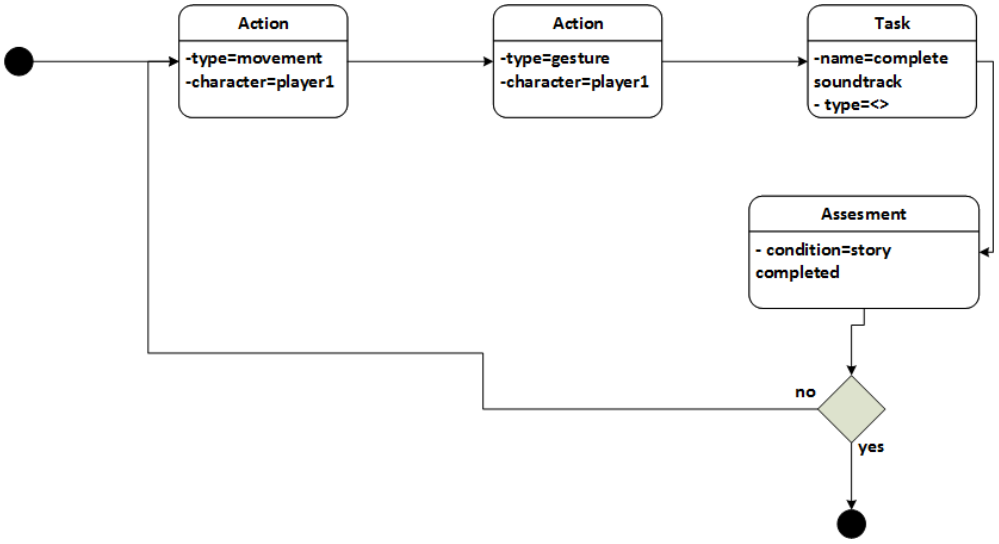

Figura 3.13: Modello logico di Harmonic Walk con GLiSMo

character per rappresentare i due *serious player*.
Il modello logico invece prevede due componenti *action* per rappresentare le azioni dei due giocatori. La task è comporre il brano polifonico senza consumare tutte le vite e pertanto sono necessari due elementi di *assessment*, in cui prima si controlla che gli utenti abbiano abbastanza vite per continuare il gioco (altrimenti il gioco termina) e poi se sono state individuate tutte le tracce corrette. Possiamo vedere in figura 3.15 il risultato.

3.3 Confronto tra i due modelli

Sebbene il concetto di fondo tra MoPPLiq e GLiSMo sia lo stesso, ovvero la creazione di giochi a partire da un modello UML costituito da componenti predefiniti, possiamo osservare diverse differenze tra i due modelli, soprattutto in relazione alla loro capacità di descrivere le applicazione di nostro interesse.
MoPPLiq risulta essere un formalismo più di alto livello che si limita a descrivere gli scenari dei giochi tramite *activity* e le azioni che si possono compiere sono gli obiettivi e gli input di queste attività. Nonostante ciò si possa tradurre in modelli più semplici (se si hanno le componenti adeguate) risulta essere anche un grosso svantaggio. La limitazione più evidente è l'impossibilità di rappresentare i contenuti multimediali e la mancanza di strumenti per la modellazione dell'ambiente di gioco. Inoltre, abbiamo visto che *Good Or Bad?* non viene descritto interamente poiché il sistema di gestione delle vite non è integrabile nel formalismo di MoPPLiq. Infatti, comparandolo a GLiSMo, esso risulta essere più un modello logico, e non considera gli elementi statici e astratti che devono poter essere definiti in qualche maniera da chi usa sta creando un'applicazione di questo

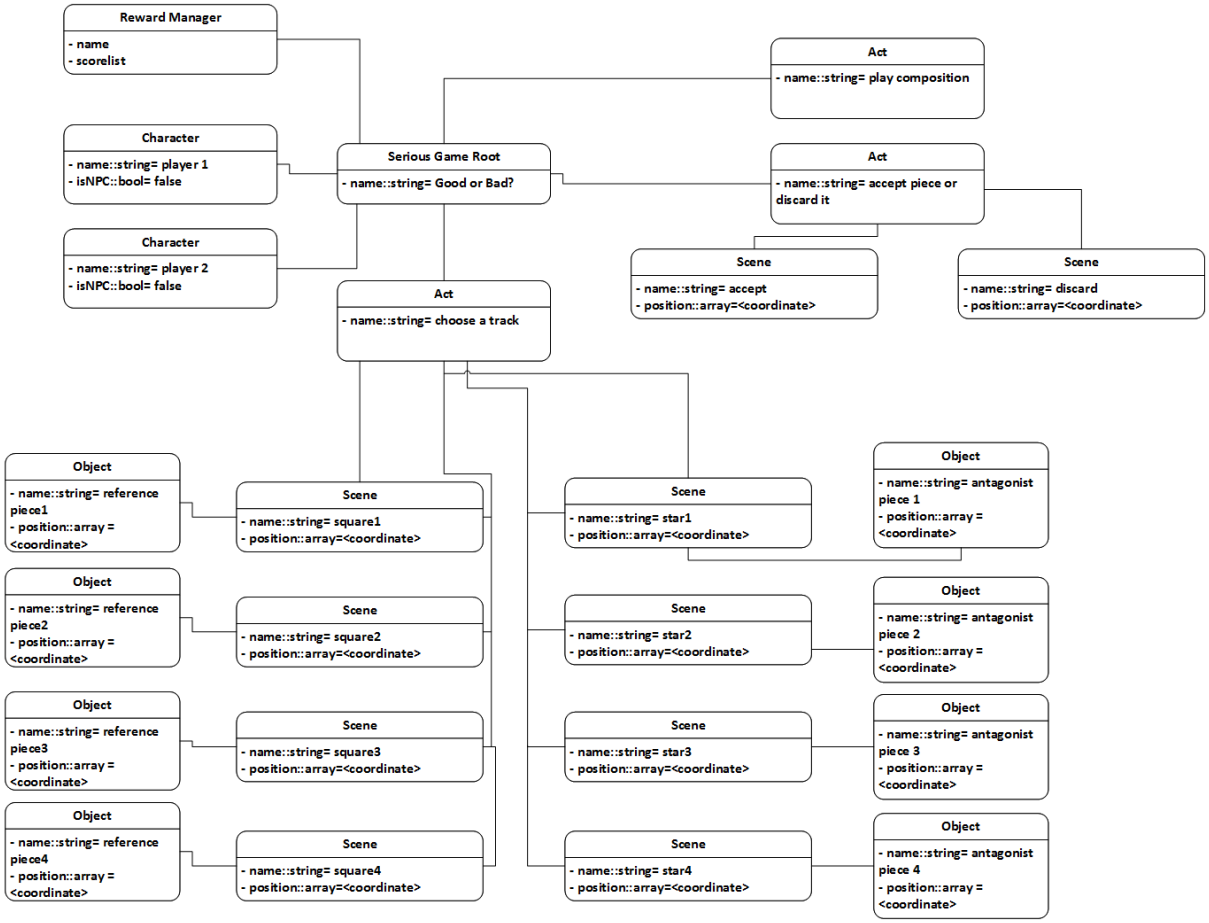

Figura 3.14: Struttura di Good or Bad? con GLiSMo

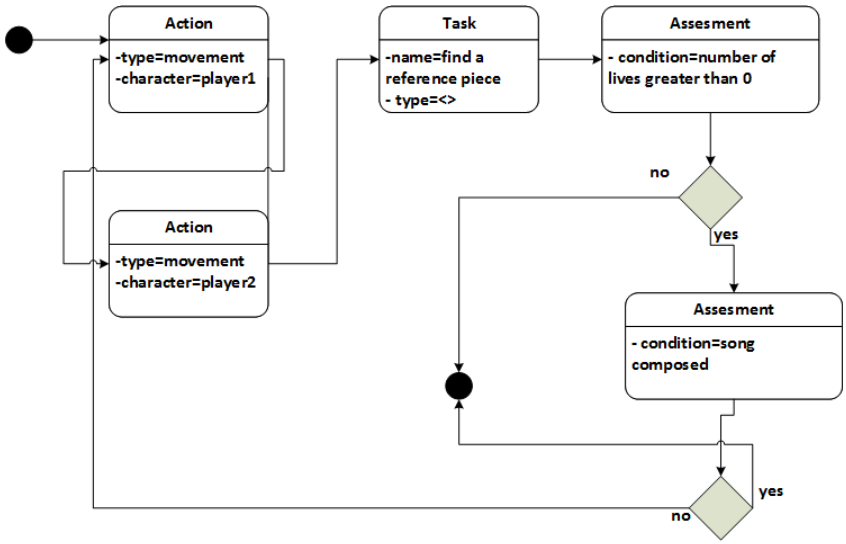

Figura 3.15: Modello logico di Good or Bad? con GLiSMo

tipo.
Invece GLiSMo, già nella sua formulazione originale, dividendo la progettazione in due fasi, permette di definire sia il layout del gioco che le azioni che si possono compiere, grazie ad un buon insieme di elementi che riescono a descrivere con maggiore dettaglio le caratteristiche dell'applicazione. La rappresentazione risulta essere pertanto più completa anche se più complessa, potendo definire ad esempio i file multimediali di ogni zona dell'interfaccia, o avendo una componente per in controllo dei punteggi. Tuttavia non è ancora definito come collegare il modello strutturale di GLiSMo a quello logico è descrivibile come una sorta di macchina a stati. Un altro fattore a favore di GliSMo è la possibilità di manipolare singolarmente gli elementi strutturali del gioco che si traduce in una modellazione più accurata dell'ambiente multimediale, come già citato prima. Al contrario MoPPLiq non specifica come progettare gli oggetti astratti e non che compongono i SG.
Per questi motivi, abbiamo scelto di usare GliSMo come punto di partenza per definire un modello formale per i giochi che vogliamo sviluppare con E-Learn.

3.4 Modello finale per E-Learn

Durante la realizzazione del game engine presentato nel prossimo capitolo, sono emerse ulteriori possibilità di modifica di GliSMo. Descriveremo ora il modello definitivo che è stato ricavato per descrivere le applicazioni per E-Learn. Puntualizziamo già ora che la formalizzazione che segue è stata ottenuta modificando il modello in base alle esigenze di implementazione, e che l'applicazione target è Resonant Memory.

3.4.1 Modello Strutturale e Logico

Oltre alle variazioni precedenti, il modello strutturale è stato modificato come di seguito: l'elemento *Serious Game Root* è stato eliminato poiché rivelatosi inutile ai fini pratici; abbiamo ora un solo elemento di tipo *act* e tante *scene*, una per zona sensorizzata di gioco. Sono state definiti infine gli attributi per le *scene* e gli *object*: per le prime sono sufficienti *name* e*id*; quest'ultimo serve principalmente a collegarlo all'*object* tramite la *scene_ id*; anche gli oggetti hanno un attributo *name* e uno *media* per indicare il percorso del file. Lo scopo di *name* è per facilitare il riconoscimento della scena o dell'oggetto a chi definisce i parametri del gioco. Anche *act* ha tre attributi, *name*, *scenes* (numero di aree) e *scenesPerRow* (numero di aree per riga). Definiamo inoltre un attributo *id* anche per identificare i giocatori in maniera univoca, in modo che possano essere associati alle

azioni che devono svolgere.
Nel modello logico è stato eliminato l'elemento *Assessment*, ma solo dal punto di vista grafico: infatti, le funzioni di questo elemento possono essere svolte dal *Task Manager*, ove possibile, o direttamente dall'ingegnante. Ricordiamo infatti che grazie ad un ambiente come E-Learn anche l'insegnante è direttamente coinvolto nel gioco, e può guidare passo a passo gli alunni.
A scopo di esempio riportiamo la formalizzazione di *Good or Bad?* secondo il modello sopra descritto, invece che quella di *Resonant* che rimane pressoché invariata.

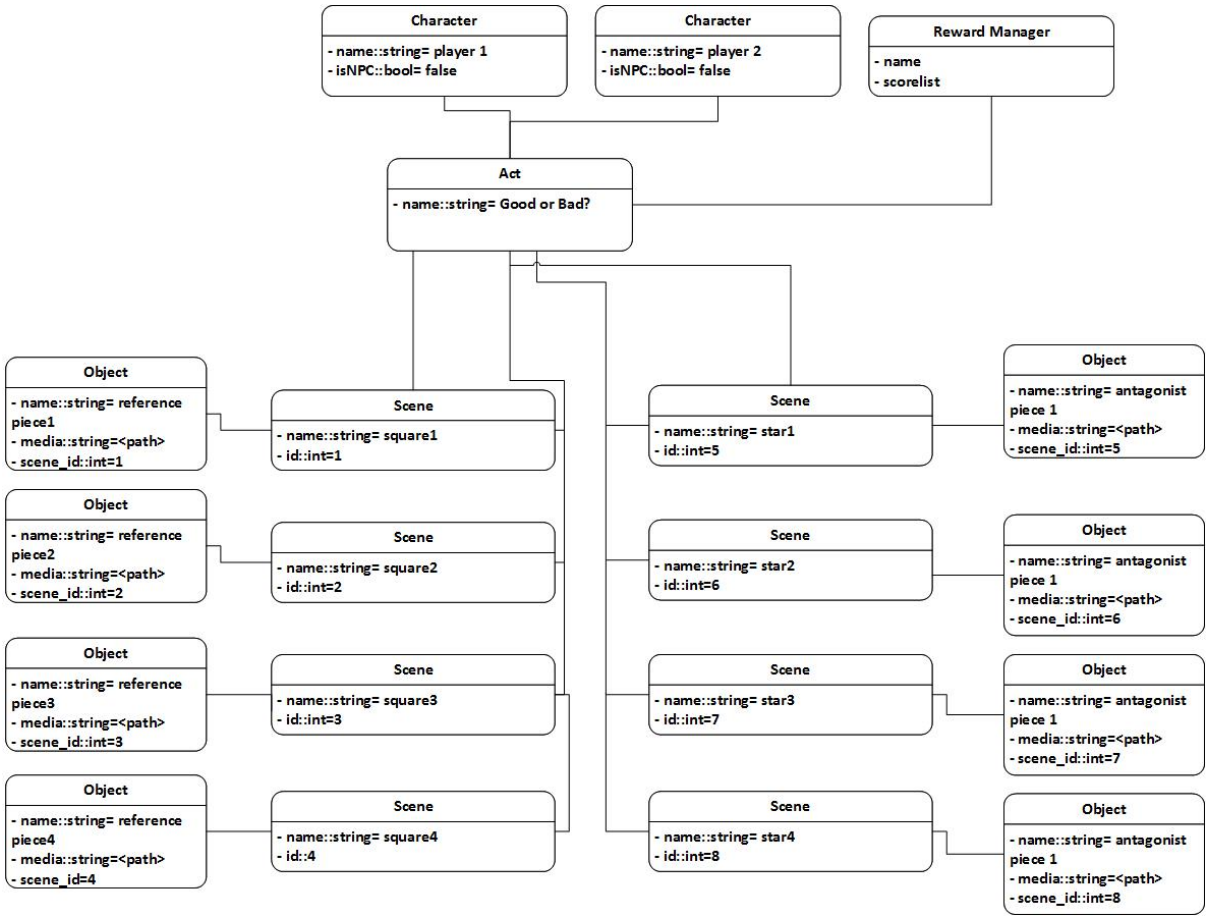

Figura 3.16: Struttura di *Good or Bad?* con il modello definitivo

Il modello logico è invariato rispetto a prima, eccetto l'uso della componente *Task Manager* invece di *Assessment*.

3.4.2 Rappresentazione in XML

Definiti quindi i modelli non ci resta che rappresentarli in XML per poterli dare in input al game engine. Dobbiamo ora definire una sintassi adeguata. Vediamo però che essa è

molto semplice. Di seguito vediamo la definizione delle impostazioni di gioco

```
<?xml version="1.0" encoding="UTF-8"?>

<Act name="Resonant" scenes="9" scenesPerRow="3">
 <Scene name="main zone" id="1"></Scene>
 <Object name="main story"
                        media="ilpescatore.mp3" scene_id="1"></Object>

 <Scene name="pheripheral zone 1" id="1"></Scene>
 <Object name="sound1"
                        media="piero.mp3" scene_id="2"></Object>
        ...
 <Player name="player1" id_player=1"></Player>
        ...
</Act>
```

Per il modello strutturale, l'elemento *root* è *act*, e ha come figli tutti gli altri elementi. Osserviamo come tutti i dati necessari sono inseriti come attributi.
Anche il modello logico è definito in maniera minimalista:

```
<?xml version="1.0" encoding="UTF-8"?>
<gamestates name="Resonant Memory Logic">
 <action name="explore" scene_id="null" action_id="1"
        endCondition="PRESS_KEY" key='q' player_id="null"></action>
</gamestates>
```

Elementi	Attributi
Act	name, scenes, scenesPerRow
Scene	name, id
Object	name, media, scene_id
Player	name, player_id
gamestates	name
action	name, scene_id, action_id, endCondition, key, player_id

Tabella 3.1: Principali elementi e loro attributi della rappresentazione in XML

Tra gli attributi delle *action* troviamo le condizioni di fine e le task da svolgere. In questo caso alcuni valori sono posti a *null* poiché non servono per *Resonant*; tra le

condizioni è stata inserita la possibilità di terminare il gioco con un comando da tastiera. Naturalmente, questa sintassi andrà estesa per poter definire le altre applicazioni oltre *Resonant Memory*.

Capitolo 4

Realizzazione di un game engine con Processing

A questo punto della trattazione non ci resta che mostrare l'efficacia dell'approccio scelto realizzando un piccolo game engine con Processing, seguendo i dettami del modello definito nel capitolo precedente.

4.1 Analisi dei requisiti per un game engine per E-Learn

La progettazione di un game engine per un ambiente come E-Learn pone sfide diverse da quelle di un classico videogioco, in quanto i giochi che vogliamo modellare e creare differiscono molto da quelli dekstop. Questa restrizione pone limiti a quello che dovrà fare un game engine per E-Learn. Il nostro obiettivo è poter creare applicazioni come quelle descritte nel secondo capitolo, ovvero poter creare degli spazi di gioco nel nostro ambiente, associare elementi multimediali a determinate aree, poter leggere il movimento dei giocatori e decidere cosa fare di conseguenza, etc. Pertanto presentiamo un'analisi delle caratteristiche del nostro motore per i giochi.
Innanzitutto, possiamo fare una prima osservazione banale, ovvero non dobbiamo realizzare nessuno delle componenti classiche che distinguono gli engine da zero. Ciò semplicemente perché sono disponibili molte librerie in letteratura da sfruttare, e inoltre il nostro game engine non ha bisogno di tutte le parti che caratterizzano una suite classica per la creazione di contenuti video-ludici. Ne è un esempio lampante quello del motore fisico: non dobbiamo ne simulare giocatori o oggetti virtuali ne rilevare la collisione tra questi. Per la tipologia di gioco a cui miriamo è sufficiente conoscere poter elaborare i dati in input dei giocatori che vengono inviati dai sensori, e poter manipolare file multimediali

come immagini o musica.
Inoltre, non avrebbe senso implementare da zero un sofisticato renderer engine o un audio engine: infatti vedremo che Processing mette a disposizione due rendeder integrati (P2D e P3D) e diverse librerie per l'audio che risultano essere sufficienti ai nostri scopi.
Le componenti di che richiedono più impegno sono la realizzazione del gameplay e il collegamento tra formalismo in XML e la manipolazione delle parti del gioco, dalla definizione del layout alle azioni che i giocatori possono compiere. Il primo non è stato ben descritto in [51], e pertanto è stato realizzato da zero, cercando di mantenere un buon grado di coerenza con il modello strutturale. Un discorso analogo vale per la lettura dei dati nei file XML, usati per definire le impostazioni dell'applicazione da creare. Inoltre, dobbiamo realizzare la gestione delle *task*, senza la quale non potremmo avanzare nel *gameplay*, o dare significato un significato più ricco alle nostre applicazioni (senza le azioni sarebbero fine a se stessi).
Per semplicità, è stato scelto Resonant Memory come target del game engine. Inoltre, dato che l'intento è di sviluppare un prototipo, si usano le coordinate del mouse per simulare la presenza degli utenti e i loro movimenti; di conseguenza la comunicazione tra sensori e game engine non è stata realizzata al momento. Tuttavia, dato che Good or Bad è già stato realizzato con Processing, possiamo assumere che questa componente sia assolutamente realizzabile.

4.2 Perché Processing?

Nato nel 2001, Processing è un ambiente di sviluppo e un linguaggio open source di alto livello, con una sintassi pressoché identica a Java ma semplificato per poter lavorare con le arti visuali, e per poter essere usato a fini didattici per insegnare programmazione. Supportato da una comunità attiva, esso è diventato uno strumento per la prototipazione rapida di software, similmente a Python, ed è una buona scelta per chi vuole provare a realizzare programmi interattivi, come i videogiochi vedremo. Ciò è possibile grazie alla disponibilità di una libreria abbastanza ricca, e alla sua integrazione con altri linguaggi, Java e Python in primis. Supporta inoltre la programmazione ad oggetti.
L'ambiente di sviluppo è minimale, e non richiede di compilare il codice, l'IDE rivela infatti durante la scrittura ogni eventuale errore, come in un linguaggio di *scripting*. Per eseguire il programma basta premere un tasto e immediatamente sarà possibile vedere il risultato in console se il programma lo prevede, e in una finestra con il contenuto creato.
La logica di funzionamento di Processing prevede l'uso di due funzioni base: setup() che viene chiamato all'inizio dell'esecuzione, e draw() viene richiamato in loop per ogni frame. La prima viene usata per inizializzare le variabili e definire il layout iniziale dell'applica-

zione, e può essere usato per realizzare programmi non interattivi. Con draw() invece è possibile azionare eventi ad esempio da tastiera o leggendo la posizione del mouse. Sono presenti infatti diverse funzioni come mousePressed(), mouseMoved(), keyPressed() che vengono richiamate assieme a draw() ad ogni frame. Questo *loop* continuo si sposa bene con l'idea che un game engine ha un ciclo che funziona in continuazione per garantire l'avanzamento da uno stato discreto ad un altro, tramite l'aggiornamento delle variabili in reazione ad eventi azionati dai giocatori.
Integrati con Processing abbiamo due renderer: P2D, renderer di default per oggetti bi-dimensionali, e P3D, che sfrutta OpenGL per disegnare velocemente in 3D. Sono previste diverse primitive per disegnare figure geometriche, come rect() per i rettangoli o sphere() per una sfera. Possiamo vedere in figura un esempio di composizione geometrica ottenuta con poche righe di codice.

```
void setup()
{
  size(640,360,P3D);
  background(0);
  lights();

  pushMatrix();
  translate(130, height/2, 0);
  rotateY(1.25);
  rotateX(-0.4);
  noStroke();
  box(100);
  popMatrix();

  pushMatrix();
  translate(500, height*0.35, -200);
  noFill();
  stroke(255);
  sphere(280);
  popMatrix();
  saveFrame("exampleP3D.png");
}
```

Si potrà già intuire perché è stato scelto Processing per sviluppare un game engine; elenchiamo di seguito le motivazioni maggiori:

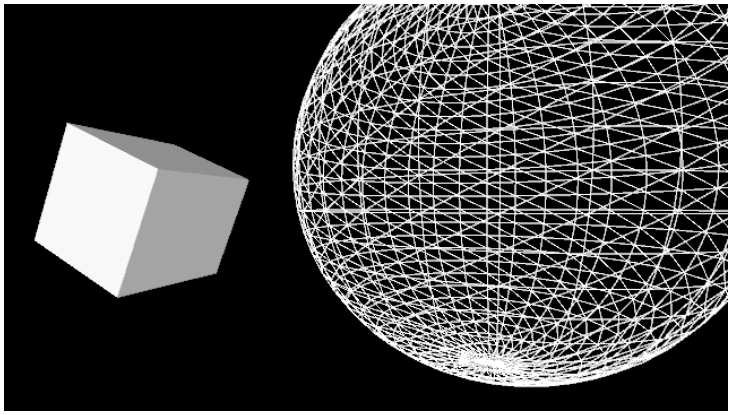

Figura 4.1: Composizione geometrica con Processing

- è un linguaggio creato per le arti visuali, e facilita la gestione dell'interattività e quindi se vogliamo creare giochi multimediali che fanno uso di suoni e immagini, Processing è certamente una scelta molto valida;
- essendo un linguaggio java-like risulta semplice da imparare;
- la sintassi semplificata, la presenza di funzioni per disegnare e leggere dati in input da device del PC lo rendono ideale come linguaggio di prototipazione;
- è presente un *renderer* integrato ed è possibile comunque importare strumenti più complessi;
- parimenti per il comparto audio esistono librerie che gestiscono questo aspetto facilmente e a diversi livelli di complessità;
- è possibile esportare un applicazione in pochi passi direttamente dai menù dell'ambiente di sviluppo, e si ha subito un'applicazione pronta all'uso, supportato da qualsiasi PC con Java installato;
- la funzione draw() si presta bene per contenere la parte centrale dell'engine, visto che viene chiamata ad ogni frame.

4.3 Descrizione delle componenti

Segue ora la descrizione passo a passo dell'implementazione del game engine.

4.3.1 Realizzazione dell'interfaccia di gioco

Resonant Memory prevede di dividere lo spazio di gioco in 9 zone e di associarvi delle tracce sonore. Pertanto tra le prime istruzioni che deve eseguire il nostro game engine è

la lettura dei dati necessari alla realizzazione di questo *setting* di gioco.
Nel file XML sono indicati in numero di zone e in numero di zone per riga. Vediamo prima come sono stati utilizzati questi due valori.

```
b=floor((1/float(rect_per_row))*width);
rect_per_col = ceil(rect/float(rect_per_row));
h=floor((1/float(rect_per_col))*height);
```

Sfruttando questi due dati e le dimensioni dell'intera area di gioco, possiamo trovare le dimensioni delle nostre zone. La funzione rect() necessita delle coordinate dell'angolo superiore sinistro, oltre che della base e dell'altezza; pertanto calcoliamo questi punti, e li salviamo in due vettori per poter sfruttare questa informazione per sapere dove sono i giocatori.

```
int x_k=-b;
int y_k=0;
//draw the rects
for (int i=0; i<rect; i++)
{
    x_k=x_k+b;
    if (b<=width-x_k)
    {
      fill(random(255), random(255), random(255), random(255));
      rect(x_k, y_k, b, h);
      xpositions[i]=x_k;
      ypositions[i]=y_k;
    }
    else if (y_k==0)
    {
       x_k=0;
       y_k=h;
       fill(random(255), random(255), random(255), random(255));
       rect(x_k, y_k, b, h);
       xpositions[i]=x_k;
       ypositions[i]=y_k;
    }
    else
```

```
    {
        x_k=0;
        y_k=y_k+h;
        fill(random(255), random(255), random(255), random(255));
        rect(x_k, y_k, b, h);
        xpositions[i]=x_k;
        ypositions[i]=y_k;
    }
}
```

Non resta che associare i file multimediali alle zone: questo viene fatto sfruttando l'attributo *scene id* che accumuna gli elementi di tipo *Object* e le *Scene*. Salvando in una struttura dati, nel nostro caso una *HashMap*, adeguata le coppie <id,nome file>, possiamo accoppiare in maniera univoca i file e le aree.

```
//save tracks
tracks=new HashMap<Integer,String>(rect_per_row);
for (int i=0; i<object.length; i++)
{
 tracks.put(object[i].getInt("scene_id"),
            object[i].getString("media"));
 println(tracks.get(i+1), i+1);
}
```

Tutto ciò viene svolto all'interno della funzione setup(); inoltre qui vengono inizializzate molte delle componenti e delle variabili che servono al corretto funzionamento del game engine. Possiamo vedere in figura 4.2 il layout ottenuto prima specificando due valori diversi di zone.

4.3.2 Gestione file multimediali

Quando un giocatore entra in una nuova zona, deve iniziare la riproduzione della traccia corrispondente. Dunque è necessario realizzare un lettore multimediale che gestisca questa fase. La libreria scelta per implementare questa componente è Minim: essa è stata realizzata sfruttando JavaSound Api, tramite Tritonus e MP3SPI (librerie opensource), e offre delle facilitazioni per integrare l'audio negli sketch (termine usato per indicare uno script in Processing). Tra le caratteristiche citiamo:

- la riproduzione mono e stereo di file nei formati più usati (WAV, AIFF, AU, SND, and MP3) con AudioPlayer;

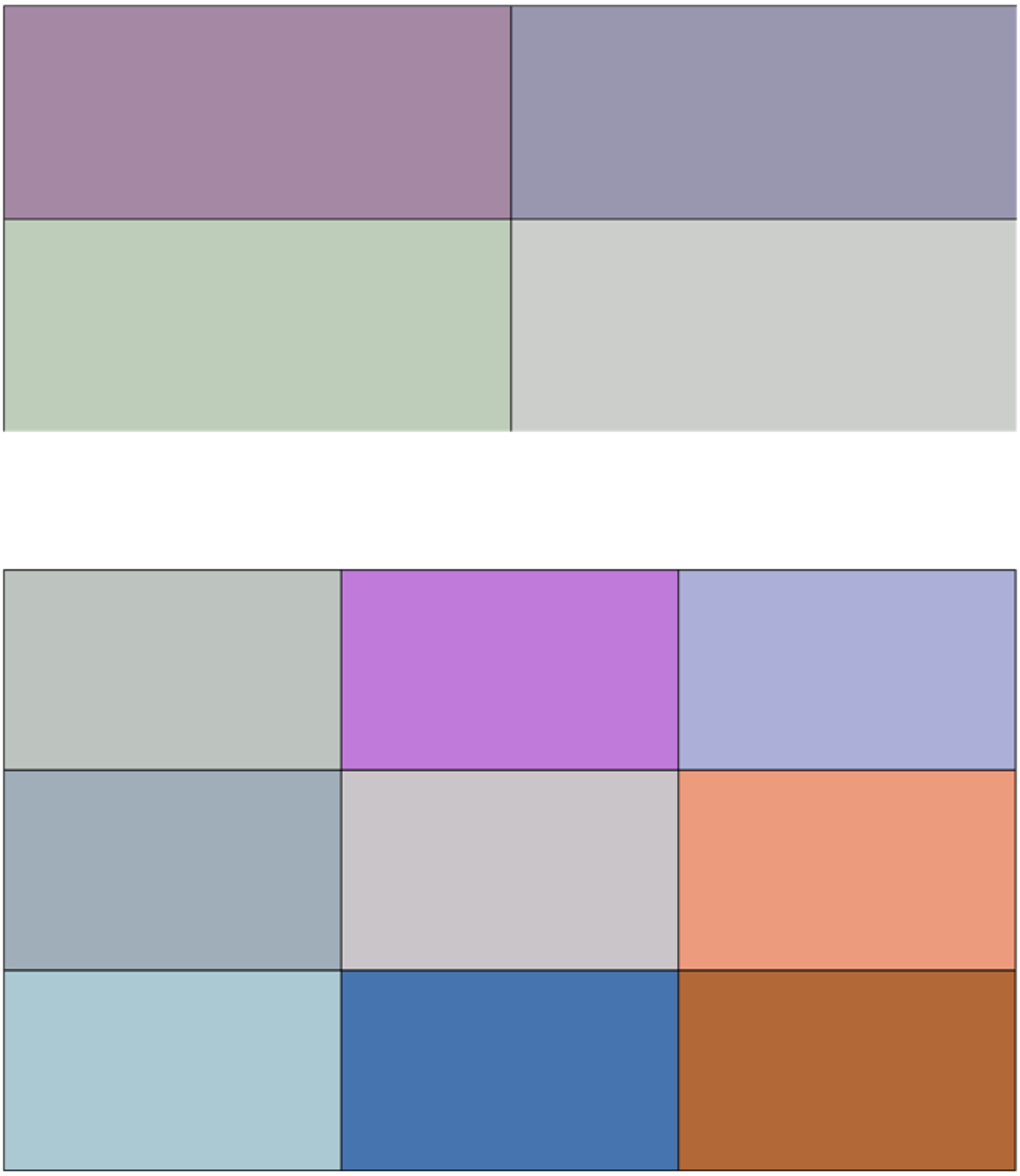

Figura 4.2: L'interfaccia per *Resonant Memory*, sopra specificando una suddivisione in 4 zone, sotto in 9

- con AudioMetaData si possono estrapolare informazioni sui tag del file (nome, lunghezza, etc.);
- l'oggetto AudioRecorder per la registrazione mono e stereo;
- per chi vuole funzioni più avanzate è possibile fare una FFT del segnale per valutarne lo spettro;
- un framework per la sintesi real-time di segnali, UGen.

Per il nostro prototipo sono stati usati in particolare AudioPlayer e AudioMetadata: il primo carica la traccia dalla HashMap definita in setup() tramite la chiave, e lo riproduce; tramite i metadata del file controlliamo se è già in riproduzione e in quel caso non viene compiuta alcuna azione. L'uso della *HashMap* è giustificato, oltre che per la comodità di poter individuare con una coppia chiave-valore i dati, dai tempi di accesso costanti nell'uso pratico. Di seguito riportiamo il codice completo.

```
//media player
void playSoundtrack(int id)
{
  String track=tracks.get(id);
  if (player==null) //no track playing
  {
      minim=new Minim(this);
      player = minim.loadFile(track);
      player.play();
      metadata=player.getMetaData();
      println("Now␣playing␣"+metadata.fileName());
  }
  if(player.isPlaying())
  {
    metadata=player.getMetaData();
    if (metadata.fileName().equals(track)==true)
    {
      return; //track already playing
    }
    else
    {
      minim.stop();
```

```
        minim=new Minim(this);
        player = minim.loadFile(track);
        player.play();
        metadata=player.getMetaData();
        println("Now playing "+metadata.fileName());
      }

    }
    else
    {
        minim.stop();
        minim=new Minim(this);
        player = minim.loadFile(track);
        player.play();
    }
}
```

Il funzionamento del codice segue la seguente logica:

- vediamo come, letto il nome del file corrispondente all'*id*, prima si controlla che l'oggetto *AudioPlayer* sia ancora vuoto, e in caso affermativo si inizializzano due oggetti Minim e Audioplayer, definiti come variabili globali, e si riproduce il brano con la funzione *play()*;
- se invece, è già in uso il lettore, si controlla che non si ricominci a riprodurre lo stesso file in continuazione (ciò porterebbe anche a un *crash* del programma) finché il giocatore si trova ancora nella stessa zona. Questo viene fatto controllando il nome del file tramite con un oggetto di tipo MetaData e la funzione *getMetaData()*;
- infine, se il file è diverso, vuol dire che il giocatore è entrato in una nuova zona, e si inizia la riproduzione della traccia richiesta, fermando quella precedente.

4.3.3 Il *gameplay*

In GLiSMo il modello logico non specifica ancora il funzionamento del *gameplay*. Questo fatto non è stato penalizzante di per sé in quanto avremmo dovuto cambiarne l'implementazione in ogni caso data la diversità di genere dei nostri giochi.
L'idea di base è stata ispirata dalla struttura di un racconto: l'inizio dove il gioco deve ancora iniziare; la parte centrale dove avvengono tutte gli eventi principali; ed infine la conclusione del gioco. Ciò è stato realizzato con il costrutto *switch*, dove i *case* seguono

un intero progressivo che va da 0 a 2. Il caso centrale è quello fondamentale dove avvengono tutte le azioni. Il codice è stato strutturato in modo che, in base a ciò che viene specificato nel file XML sul modello logico, venga scelta l'azione che il giocatore deve eseguire. In particolare, come per le tracce musicali, vengono salvate le azioni in una *HashMap* inizializzando un'istanza della classe *Players*, dove è stato scelto di indicare e realizzare tutte le azioni che un giocatore può compiere. Di seguito si riporta il codice di questa classe.

```
class Players{
  //int n_players;
  Container c;
  String[] actions;

  Players(XML[] file)
  {
    //n_players=n;
    //fill the hashmap
    player_actions=new HashMap<Integer,String>(file.length);
    for (int i=0; i<file.length; i++)
    {
      player_actions.put(file[i].getInt("action_id"),
                                          file[i].getString("name"));

    }
  }

  //action that lets the player explor all the zonen
  Container explore()
  {
    c=checkzone();
    if (c.ishere==true )
    {
      playSoundtrack(c.id);
    }

     return c;
  }
```

```
Container checkzone()//this method return the player position
{
  Container zone=new Container(0, false);

  if (mouseX==0 & mouseY==0)//this avoids null ponter exception
  {
    return zone;
  }
  //Container zone=new Container(0, false);
  boolean is_inx= false;
  boolean is_iny= false;
  int i=0;
  while((is_inx & is_iny) ==false & (i<xpositions.length))
    {
      is_inx=range(xpositions[i], xpositions[i]+b, mouseX);
      is_iny=range(ypositions[i], ypositions[i]+h, mouseY);

      if ((is_inx & is_iny)==true)
      {
        zone.id=i+1;
        zone.ishere=true;
        break;
      }
      i++;
    }

  return zone;
}

//checks if a value t is in (x1,x2)
boolean range(int x1, int x2, int t)
{
  if (x1<t & t<x2)
  {
    return true;
  }
  else return false;
```

```
  }
}

class Container
{
  boolean ishere;
  int id;
  Container(int n, boolean value)
  {
    id=n;
    ishere=value;
  }
}
```

Il costruttore di questa classe, prende in input un array XML[] e lo usa per popolare l'*HashMap player_actions*. Tra le variabili abbiamo un oggetto *Container*: esso ha semplicemente due campi, un numero intero e un booleano. Questa struttura dati viene usata dal metodo checkzone() per dare informazioni sulla posizione del giocatore, restituendo l'*id* della zona e un booleano che assume il valore *true* se effettivamente si trova in nell'area associata all'identificativo. Per capire giocatore (nel nostro caso in realtà del mouse), cadono entro i limiti delle varie zone. Questo permette ad esempio di scegliere la traccia corretta da suonare, come viene fatto con il metodo *explore()*, ed è attualmente l'unica funzione che si può svolgere con il nostro engine.
Strettamente correlata al funzionamento del *gameplay* è l'oggetto *Task_manager*. A questa componente è stato affidato il compito di gestire l'avanzamento degli stati e di eventuali obiettivi. Nel caso di *Resonant Memory* non abbiamo dei veri e propri fini pedagogici che si possano esprimere in forma di codice, in quanto l'insegnante può dirigere in tempo reale le azioni degli alunni; dunque da questo punto di vista, questa funzione del *Task_manager* deve ancora essere ben testata e migliorata, ancor più delle altre componenti. Si riporta di seguito lo stato attuale della classe.

```
//task manager code: this module has the function
//of managing the objectives of the game,
//and since we need to perform tasks to reach
//pedagogical goals it is necessary for advancing
//with the gameplay and for the purpose of
//this prototype of authoring tool
class Condition
  {
```

```
    final static int pRESS_KEY=0;
    final static int lISTEN_ALL=1;
    Condition(){}

  }

class taskManager{
  String[] tasks;
  //Condition cond;

  taskManager(XML[] a)
  {
    tasks=new String[a.length];
    for (int i=0; i<a.length; i++)
    {
      tasks[i]=a[i].getString("endCondition");
    }
  }

  void checkCondition()
  {
    for (int i=0; i<tasks.length; i++)
    {
      if (tasks[i].equals("PRESS_KEY")==true)
      {
        if (key==ending_key)
           gamestate=2;
      }
      if (tasks[i].equals("LISTEN_ALL")==true)
      {
        //need to be made
      }
    }
  }

  void setState(int state)
  {
```

```
    if (state==0)
    {
      gamestate=1;
    }
    if (state==1)
    {
      checkCondition();
    }
  }
}
```

Osserviamo la presenza di due funzioni: setState() gestisce i casi del costrutto *switch* presentato in precedenza; e a sua volta, quando si trova stato 1, controlla la presenza di condizioni da rispettare con il metodo checkCondition().
Non ci resta che analizzare il codice di tutto il gameplay.

```
//testing script for the gameplay
void playGame(){

  Container c;

  switch (gamestate)
  {
    case 0:
      println("Let's play");
      tsk_manager.setState(gamestate);
      break;
    case 1:


      for (int i=0; i<player_actions.size(); i++)
      {
        if(player_actions.get(i+1).equals("explore")==true)
        {
          c=users.explore();
          if (c.ishere==false)
          {
            break;
          }
```

```
            tsk_manager.setState(1);
          }
        }

        break;
      case 2:
        println("Game Ends");
        exit();
        //break;
    }
}
```

Notiamo che al momento la gestione dello sviluppo del gioco è molto semplice: nel *case* "0" abbiamo solo un messaggio di benvenuto, e tramite il Task_manager entriamo nel *case* "1"; qui si controlla l'azione da svolgere (in questo caso si deve esplorare l'ambiente in cerca di tracce audio); infine si aspetta l'input da tastiera per terminare il gioco. Questa funzione viene chiamata dentro draw(), e pertanto viene eseguita per ogni frame.
Per concludere notiamo l'uso di oggetti e funzioni per rappresentare le componenti del game engine; ciò è stato fatto per separare logicamente le funzioni del software, e per poter quindi garantire l'estensibilità futura per altri SG. La soluzione proposta non è esente dai limiti di ogni game engine e authoring tool: in particolare, quando si crea un software di questo tipo si limita per forza la creatività di chi vuole creare il gioco tramite un linguaggio e un formalismo che non sempre possono rappresentare tutte le caratteristiche di un genere di giochi; però abbiamo visto che questa è una caratteristica dei game engine anche più sviluppati, e nel nostro caso viene ad essere ancora più evidente dal fatto di non usare la flessibilità di un linguaggio di programmazione.

4.4 Sviluppi futuri

Il prototipo descritto in questo capitolo vuole essere un primo passo per la creazione di un authoring tool completo. Per poter sviluppare ulteriormente questo progetto, si dovranno aggiungere le funzionalità per supportare le altre applicazioni viste nel secondo capitolo. Contestualmente, si potranno raccogliere dati sulle applicazioni che vorrebbero sviluppare gli insegnanti, ed adattare il game engine di conseguenza, cercando comunque di mantenere l'estensibilità e la riusabilità del codice. Inoltre, bisogna realizzare la comunicazione coi sensori, ma come già detto, il fatto che sia stato possibile già farlo per *Good or Bad?* ci fa pensare che sia possibile integrare questa componente senza troppe difficoltà.

Dopo che questa componente risulterà essere pronta, si potrà sviluppare un tool per la progettazione grafica dei giochi, che potrà usare i file XML come recipiente per salvare i dati sui modelli, e passare quindi le informazioni al game engine per creare il gioco. Il programma potrà avere varie implementazioni: potrà essere una semplice interfaccia per con delle opzioni da specificare, o un'interfaccia interattiva per creare gli schemi UML o ancora uno strumento che mostri la zona di gioco e su cui si possano in real-time definire gli spazi tramite il mouse. Processing si presta bene a queste tipo di applicazioni in quanto è disegnare è lo scopo principale per cui è stato creato; scrivere uno *sketch* che disegni interattivamente dei rettangoli ad esempio è questione di secondi; dovranno essere aggiunti delle funzioni per disegnare senza sovrapporre le zone, e per poter specificare gli oggetti multimediali, ed eventuali immagini da proiettare come pulsanti (come per *Good or Bad?*).

Capitolo 5

Conclusione

In questo lavoro di tesi mi sono posto l'obiettivo di studiare e realizzare degli strumenti per facilitare la creazione di Serious Game, in particolare per la piattaforma interattiva e multi-modale *E-Learn*, dove si possono svolgere lezioni coinvolgenti per gli alunni tramite l'uso di suoni e movimenti.
Partendo da un'analisi letteraria ho scelto di usare il *Model Driven Approach*, ovvero di modellare alcune applicazioni già esistenti per E-Learn con due formalismi che fanno uso della notazione UML, *Moppliq* e *Glismo*. Dopo aver analizzato e applicato questi modelli, ho concluso che Moppliq non era adatto ai nostri scopi in quanto non riusciva a cogliere tutti gli aspetti dei nostri SG; Glismo invece si è rivelato essere più potente come formalismo, distinguendo la fase di definizione dell'ambiente di gioco dalla logica del *gameplay*.
Partendo quindi da questo modello, mi sono ricavato uno strumento adatto alle nostre esigenze, modificando alcuni elementi di Glismo in entrambe le fasi di progettazione. In particolare, ho semplificato la strutturazione del layout e del *gameplay*, facilitando la fase successiva, ovvero la validazione di questo modello attraverso la realizzazione di un prototipo di game engine.
Tramite il linguaggio e IDE Processing, ho estrapolato le informazioni dei modelli codificati come XML, e manipolandole con delle componenti implementate in maniera opportuna, seguendo la logica del formalismo definito precedentemente, sono riuscito a ricreare il gioco *Resonant Memory*. Il codice del game engine è stato realizzato in modo da poter essere estensibile, così da poterne aumentare la compatibilità con le altre applicazioni di E-Learn.

Bibliografia

[1] Sylvester Arnab et al. «Mapping learning and game mechanics for serious games analysis». In: *British Journal of Educational Technology* 46.2 (2015), pp. 391–411. ISSN: 14678535. DOI: 10.1111/bjet.12113.

[2] Enrique Barra et al. «Integration of SCORM packages into web games». In: *Proceedings - Frontiers in Education Conference, FIE* (2013), pp. 685–690. ISSN: 15394565. DOI: 10.1109/FIE.2013.6684913.

[3] F. Bellotti, R. Berta e A. De Gloria. «Designing effective serious games: Opportunities and challenges for research». In: *International Journal of Emerging Technologies in Learning* 5.SPECIAL ISSUE 2 (2010), pp. 22–35. ISSN: 18688799. DOI: 10.3991/ijet.v5s3.1500.

[4] Francesco Bellotti et al. «A serious game model for cultural heritage». In: *Journal on Computing and Cultural Heritage* 5.4 (2012), pp. 1–27. ISSN: 15564673. DOI: 10.1145/2399180.2399185. URL: http://dl.acm.org/citation.cfm?doid=2399180.2399185.

[5] F Bellotti et al. «Assessment in and of Serious Games:An Overview.» In: *In Advances in Human-Computer Interaction Volume 2013* 2013 (2013).

[6] Camilla Bond. «Journal of Computer Assisted Learning». In: *Anzmac* 3.1 (2010), pp. 1–9. ISSN: 02664909.

[7] Daniel Burgos et al. «Authoring game-based adaptive units of learning with IMS Learning Design and <e-Adventure>». In: *International Journal of Learning Technology* 3.3 (2007), p. 252. ISSN: 1477-8386. DOI: 10.1504/IJLT.2007.015444.

[8] Ru Chen e Advisor Chris Pollett. «XML for Video Games Introduction :» in: (2005), pp. 1–13.

[9] Stavros Christodoulakis, Michalis Foukarakis e Lemonia Ragia. «Emerging Technologies and Information Systems for the Knowledge Society». In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 5288.November 2016 (2008), pp. 549–556. ISSN:

0302-9743. DOI: 10.1007/978-3-540-87781-3. URL: http://www.scopus.com/inward/record.url?eid=2-s2.0-56649102025%7B%5C&%7DpartnerID=tZOtx3y1.

[10] Kendra M L Cooper e C. Shaun Longstreet. «Towards model-driven game engineering for serious educational games: Tailored use cases for game requirements». In: *Proceedings of CGAMES'2012 USA - 17th International Conference on Computer Games: AI, Animation, Mobile, Interactive Multimedia, Educational and Serious Games* (2012), pp. 208–212. DOI: 10.1109/CGames.2012.6314577.

[11] Sebastian Deterding et al. «From game design elements to gamefulness». In: *Proceedings of the 15th International Academic MindTrek Conference on Envisioning Future Media Environments - MindTrek '11* (2011), pp. 9–11. ISSN: 1450308163. DOI: 10.1145/2181037.2181040. arXiv: 11/09 [ACM 978-1-4503-0816-8]. URL: http://doi.acm.org/10.1145/2181037.2181040%7B%5C%%7D5Cnhttp://dl.acm.org/citation.cfm?doid=2181037.2181040.

[12] Ralf Dörner, Marcelo Kallmann e Yazhou Huang. «Content creation and authoring challenges for virtual environments: From user interfaces to autonomous virtual characters». In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 8844 (2015), pp. 187–212. ISSN: 16113349. DOI: 10.1007/978-3-319-17043-5_11.

[13] Kim Dung Dang e Ronan Champagnat. «An Authoring Tool to Derive Valid Interactive Scenarios». In: *Papers from the 2013 AIIDE Workshop, Intelligent Narrative Technologies (6th INT, 2013)* 6 (2013), pp. 9–15.

[14] Micah Eckhardt. «A Platform for Creating Stories Across Digital and Physical Boundaries». In: (2014).

[15] Baltasar Fern. «Creating cost-effective adaptive educational hypermedia based on markup technologies and». In: July (2016).

[16] Matteo Gaeta et al. «A methodology and an authoring tool for creating Complex Learning Objects to support interactive storytelling». In: *Computers in Human Behavior* 31.1 (2014), pp. 620–637. ISSN: 07475632. DOI: 10.1016/j.chb.2013.07.011. URL: http://dx.doi.org/10.1016/j.chb.2013.07.011.

[17] Matteo Gaeta et al. «A methodology and an authoring tool for creating Complex Learning Objects to support interactive storytelling». In: *Computers in Human Behavior* 31.1 (2014), pp. 620–637. ISSN: 07475632. DOI: 10.1016/j.chb.2013.07.011.

[18] Alma G??mez-Rodr??guez et al. «Modeling serious games using AOSE methodologies». In: *International Conference on Intelligent Systems Design and Applications, ISDA* (2011), pp. 53–58. ISSN: 21647143. DOI: 10.1109/ISDA.2011.6121630.

[19] Stefan Göbel et al. «Narrative Game-based Learning Objects for Story-based Digital Educational Games». In: *Proceedings of the 1st International Open Workshop on Intelligent Personalization and Adaptation in Digital Educational Games* 14.October (2009), pp. 43–53. URL: http://www.eightydays.eu/uploads/tx%7B%5C_%7Dtakoscientificpublications/GdCMS09.pdf.

[20] Stefan Göbel et al. «Serious games for health». In: *Proceedings of the international conference on Multimedia - MM '10* October (2010), p. 1663. DOI: 10.1145/1873951.1874316. URL: http://dl.acm.org/citation.cfm?doid=1873951.1874316.

[21] Jason Gregory. *Game Engine Architecture*. A cura di Ltd. A K Peters e Massachusetts Wellesley. ISBN: 9781439865262.

[22] Sebastian Kelle. *Game Design Patterns for Learning*. 2012, p. 196. ISBN: 978-3-8440-13726. URL: http://dspace.ou.nl/handle/1820/4512.

[23] Sebastian Kelle, Roland Klemke e Marcus Specht. «Design patterns for learning games». In: *Int. J. Technology Enhanced Learning* 3.6 (2011), pp. 555–569. ISSN: 1753-5255. DOI: 10.1504/IJTEL.2011.045452.

[24] Sebastian Kelle et al. «Standardization of Game Based Learning Design». In: *Computational Science and Its Applications - ICCSA 2011 - Lecture Notes in Computer Science* 6785 (2011), pp. 518–532. ISSN: 03029743. DOI: 10.1007/978-3-642-21898-9_43.

[25] R Koper e B Olivier. «Representing th eLearning Design of Units of Learning». In: *Educational Technology & Society* 7 (2004), pp. 97–111. ISSN: 14364522.

[26] Theodore Lim et al. «Narrative Serious Game Mechanics (NSGM) - Insights into the narrative-pedagogical mechanism». In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 8395 LNCS (2014), pp. 23–34. ISSN: 16113349. DOI: 10.1007/978-3-319-05972-3_4.

[27] Conor Linehan et al. «Practical, appropriate, empirically-validated guidelines for designing educational games». In: *CHI '11 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* July 2015 (2011), pp. 1979–1988. DOI: 10.1145/1978942.1979229. URL: http://eprints.lincoln.ac.uk/4475/.

[28] Rafael P De Lope, Nuria Medina-medina e C Periodista Rafael G. «Using UML to Model Educational Games». In: (2016), pp. 1–4.

[29] V Maike e M C C Baranauskas. «An authoring process for educational Role Playing Games: From the paper to the web». In: *Proceedings of the 21st International Conference on Computers in Education, ICCE 2013* November (2013), pp. 600–610. URL: http://www.scopus.com/inward/record.url?eid=2-s2.0-84896464190%7B%5C&%7DpartnerID=40%7B%5C&%7Dmd5=.

[30] Marcella Mandanici, Antonio Rod e Sergio Canazza. «The " Harmonic Walk ": an Interactive Educational Environment to Discover Musical Chords .» In: September (2014), pp. 1778–1785.

[31] Marcella Mandanici et al. «Looking inside a Musical Score : Stimulating Listening Skills through Playful Engagement .» In: (), pp. 1–26.

[32] Eugenio J. Marchiori et al. «A narrative metaphor to facilitate educational game authoring». In: *Computers and Education* 58.1 (2012), pp. 590–599. ISSN: 03601315. DOI: 10.1016/j.compedu.2011.09.017.

[33] Iza Marfisi-schottman, Sébastien George e Franck Tarpin-bernard. «Tools and Methods for Efficiently Designing Serious Games». In: *4th Europeen Conference on Games Based Learning ECGBL2010* October (2010), pp. 226–234. URL: http://free.iza.free.fr/articles/ECGBL-iza.pdf.

[34] Bertrand Marne e Jean-Marc Labat. «Model and Authoring Tool to Help Teachers Adapt Serious Games to their Educational Contexts». In: *International Journal of Learning Technology* 9.2 (2014), pp. 161–180. ISSN: 17418119. DOI: 10.1504/IJLT.2014.064491. URL: http://hal.upmc.fr/hal-01087301%7B%5C%%7D5Cnhttps://hal.archives-ouvertes.fr/hal-01087301.

[35] Piotr Marszał. «ADDING UNITY3D AN AUTHORING LAYER FOR NON-PROGRAMMERS». In: June (2016).

[36] Amir Matallaoui, Philipp Herzig e Rudiger Zarnekow. «Model-driven serious game development integration of the gamification modeling language GaML with unity». In: *Proceedings of the Annual Hawaii International Conference on System Sciences* 2015-March (2015), pp. 643–651. ISSN: 15301605. DOI: 10.1109/HICSS.2015.84.

[37] Ben Medler e Brian Magerko. «Scribe: A Tool for Authoring Event Driven Interactive Drama». In: *Proceedings of the Third International Conference on Technologies for Interactive Digital Storytelling and Entertainment (TIDSE 2006)* (2006), pp. 139–150. DOI: 10.1007/11944577_14. URL: http://lcc.gatech.edu/

%7B~%7Dbmedler3/papers/MedlerMagerkoTIDSE2006.pdf%7B%5C%%7D5Cnhttp://link.springer.com/10.1007/11944577%7B%5C_%7D14.

[38] Florian Mehm, Stefan Göbel e Ralf Steinmetz. «Authoring of serious adventure games in StoryTec». In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7516 LNCS.September (2012), pp. 144–154. ISSN: 03029743. DOI: 10.1007/978-3-642-33466-5_16.

[39] Florian Mehm, Stefan Goebel e Ralf Steinmetz. «Introducing Component-Based Templates Into a Game Authoring Tool». In: *Proceedings of the 5Th European Conference on Games Based Learning* (2011), pp. 395–403.

[40] Florian Mehm et al. «An Authoring Tool for Adaptive Digital Educational Games». In: *Proceedings of the Seventh European Conference on Technology Enhanced Learning 21st Century Learning for 21st Century Skills* September (2012), pp. 236–249. DOI: 10.1007/978-3-642-33263-0_19.

[41] Florian Mehm et al. «Authoring Environment for Story-based Digital Educational Games». In: *Proceedings of the 1st International Open Workshop on Intelligent Personalization and Adaptation in Digital Educational Games* October (2009), pp. 113–124. URL: ftp://ftp.kom.tu-darmstadt.de/papers/MGRS09.pdf.

[42] M Minović, M Milovanović e S Dušan. «Modelling knowledge and game based learning: Model driven approach». In: *Journal of Universal Computer Science* 17.9 (2011), pp. 1241–1260. ISSN: 0958695X (ISSN). URL: http://www.scopus.com/inward/record.url?eid=2-s2.0-80051761034%7B%5C&%7DpartnerID=40%7B%5C&%7Dmd5=.

[43] P. Moreno-Ger, I. Martínez-Ortiz e B. Fernández-Manjón. «The <E-Game> Project: Facilitating the Development of Educational Adventure Games». In: *International Association for Development of the Information Society (IADIS) Cognition and Exploratory Learning in Digital Age (CELDA)* (2005), pp. 1–6.

[44] Charlotte Orliac, Christine Michel e Sébastien George. «An authoring tool to assist the design of mixed reality learning games». In: *21st Century Learning for 21st Century Skills* (2012), pp. 441–446. DOI: 10.1007/978-3-642-33263-0_40. URL: http://link.springer.com/chapter/10.1007/978-3-642-33263-0%7B%5C_%7D40.

[45] Hercules Panoutsopoulos et al. «“ Create It ” - “ Share It ” - “ Game It ”: the Case of a Web- Based Digital Platform for Creating , Sharing and Delivering Gamified Educational Scenarios». In: July (2015), pp. 4917–4927.

[46] *Processing*. 2017. URL: http://processing.org.

[47] Sebastian Sauer et al. «U-Create: Creative Authoring Tools for Edutainment Applications». In: *Proceedings of the Third International Conference on Technologies for Interactive Digital Storytelling and Entertainment (TIDSE 2006)* 4326 (2006), pp. 163–168. DOI: 10.1007/11944577_16. URL: http://dx.doi.org/10.1007/11944577%7B%5C_%7D16.

[48] Angel Serrano-Laguna et al. «Building a Scalable Game Engine to Teach Computer Science Languages». In: *Revista Iberoamericana de Tecnologias del Aprendizaje* 10.4 (2015), pp. 253–261. ISSN: 19328540. DOI: 10.1109/RITA.2015.2486386.

[49] Angel Serrano-Laguna et al. «Building a Scalable Game Engine to Teach Computer Science Languages». In: *Revista Iberoamericana de Tecnologias del Aprendizaje* 10.4 (2015), pp. 253–261. ISSN: 19328540. DOI: 10.1109/RITA.2015.2486386.

[50] N Thillainathan e J M Leimeister. «Serious Game Development for Educators - A Serious Game Logic and Structure Modeling Language». In: *EDULEARN14 Proceedings* 2014 (2014), pp. 1196–1206. ISSN: 2340-1117.

[51] Niroshan Thillainathan. «for Serious Games . In : Informatik 2013 Doctoral Consortium , Koblenz , Germany . A Model Driven Development Framework for Serious». In: (2013).

[52] Javier Torrente et al. «Instructor-oriented authoring tools for educational videogames». In: *Proceedings - The 8th IEEE International Conference on Advanced Learning Technologies, ICALT 2008* (2008), pp. 516–518. DOI: 10.1109/ICALT.2008.177.

[53] Javier Torrente et al. «Instructor-oriented authoring tools for educational videogames». In: *Proceedings - The 8th IEEE International Conference on Advanced Learning Technologies, ICALT 2008* (2008), pp. 516–518. DOI: 10.1109/ICALT.2008.177.

[54] W. H. Wu et al. «Investigating the learning-theory foundations of game-based learning: A meta-analysis». In: *Journal of Computer Assisted Learning* 28.3 (2012), pp. 265–279. ISSN: 02664909. DOI: 10.1111/j.1365-2729.2011.00437.x.

[55] Amel Yessad, Jean Marc Labat e François Kermorvant. «SeGAE: A serious game authoring environment». In: *Proceedings - 10th IEEE International Conference on Advanced Learning Technologies, ICALT 2010* (2010), pp. 538–540. DOI: 10.1109/ICALT.2010.153.

[56] Rm Young e Mo Riedl. «An architecture for integrating plan-based behavior generation with interactive game environments». In: *Journal of Game Development* 1.1 (2004), pp. 1–29. DOI: 10.1.1.95.1546. URL: http://liquidnarrative.csc.ncsu.edu/pubs/jogd.pdf.

[57] Serena Zanolla et al. «Entertaining listening by means of the stanza logo-motoria: An interactive multimodal environment». In: *Entertainment Computing* 4.3 (2013), pp. 213–220. ISSN: 18759521. DOI: 10.1016/j.entcom.2013.02.001.