2011

Qubet

Progetto di Informatica Grafica

Descrizione del Progetto di Informatica Grafica del Gruppo #34

Enrico Bacis – Daniele Ciriello
#34

Sommario

Introduzione

Il gioco nasce dall'idea di implementare una versione tridimensionale del videogame Space is Key. Lo scopo del videogioco di Chris Jeff è far saltare un oggetto di forma quadrata che si muove a velocità costante attraverso i livelli senza far collidere tale oggetto con gli ostacoli.

Scopo del gioco

Mentre il cubo avanza a velocità costante, il giocatore dovrà evitare gli ostacoli facendo saltare il cubo oppure spostandosi lateralmente, attraverso livelli tridimensionali. Tali livelli potrebbero quindi essere ad una o più "corsie", a seconda della difficoltà.

Il gioco dispone inoltre di due diverse modalità: "Storia", in cui il giocatore dovrà superare diversi livelli in successione e "Arcade", in cui è possibile giocare un livello alla volta tra quelli già presenti o altri creati tramite l'"Editor dei livelli".

Componenti del gruppo

Sebbene al momento dell'iscrizione il **gruppo #34** era formato da tre persone: Enrico Bacis, Daniele Ciriello e Michele Foresti, il progetto è stato iniziato e completato solo dalle prime due, dopo la decisione del terzo componente del gruppo di non partecipare al progetto.

Si è deciso comunque di non privare il videogioco delle features principali, cosa che avrebbe reso lo sviluppo molto meno interessante e meno apprezzabile il risultato.

Abbiamo quindi deciso di implementare comunque l'editor dei livelli tramite il mouse picking, oltre al caricamento ed al salvataggio dei dati in formato XML, le textures e l'audio, composto da colonna e effetti sonori.

Strumenti e Sviluppo

Una parte importante del progetto è stata la scelta degli ambienti di sviluppo, collaborazione e documentazione migliori per le nostre aspettative. Abbiamo così sfruttato questo progetto per poter imparare delle tecniche di sviluppo, piuttosto che puntare solo agli obiettivi del corso.

Alcune di queste tecniche di sviluppo sono state studiate nel corso di Ingegneria del Software, altre invece sono state scelte dopo un'accurata fase iniziale di documentazione sugli strumenti disponibili e sullo stato dell'arte dello sviluppo software.

Abbiamo cercato degli strumenti compatibili con tutti i sistemi operativi, dato che usiamo sia sistemi Windows che Unix-based.

Collaborazione

Per lavorare insieme allo stesso progetto abbiamo usato il sistema di controllo di revisione Mercurial. In particolare il client TortoiseHg, oltre al client integrato nell'ambiente Qt.

Il controllo di revisione

Dovendo lavorare in due sugli stessi files, abbiamo subito deciso di usare il controllo di revisione, che offre numerosi vantaggi, quali:

- Tenere automaticamente traccia della storia e dell'evoluzione del progetto: per ogni modifica lo strumento registra informazioni su chi ha eseguito la modifica, la descrizione della modifica, quando è stata effettuata e cosa è stato modificato.
- Risoluzione dei conflitti: lavorando in gruppo, il software di controllo facilità la collaborazione. Ad esempio, quando due persone effettuano cambiamenti incompatibili, il software aiuterà a identificare e risolvere tali conflitti (in particolare abbiamo deciso di usare KDiff3 come tool di Merging).
- Rettifica degli errori: se si effettua una modifica che più avanti si rivela essere un errore, si può sempre tornare ad una versione precedente.
- Branching: Il software aiuta a lavorare simultaneamente su molteplici rami di un progetto e a gestire gli spostamenti tra un ramo e l'altro.

L'uso di Mercurial

E' stato scelto Mercurial come strumento di controllo di revisione in quanto è uno strumento user-friendly, pur permettendo un'alta scalabilità, e fornendo numerose funzionalità utili. Mercurial è infatti un tool molto più completo di svn, tuttavia ora che abbiamo iniziato ad utilizzare il controllo di revisione, sarà più facile capire e comprendere Git, vero tool di riferimento negli ultimi tempi.
Si può prendere visione delle differenze tra i vari sistemi di controllo di revisione visitando il sito: http://it.whygitisbetterthanx.com/ .

BitBucket

Per hostare il repository Mercurial del progetto è stato usato il servizio BitBucket, che permette di creare gratuitamente repository pubblici e repository privati (con numero di utenti limitato).

Documentazione

Un'altra cosa a cui abbiamo puntato è quella di utilizzare un sistema di documentazione del codice simile a Javadoc per Java, in modo da rendere il codice comprensibile anche a possibili futuri interessati. (La documentazione è stata quindi redatta in Inglese).

Doxygen

Il sistema che ci è sembrato più completo è stata la soluzione Doxygen (integrabile in Qt con un plugin), che è utilizzata da moltissimi sviluppatori professionali.

Doxygen è un sistema di documentazione per C++, C, Java, Objective-C, Python, IDL, Fortran, VHDL, PHP, C#, ed alcune estensioni di D.

Esso può aiutare in diversi modi:

- Può generare una documentazione (in HTML) destinata ad essere visualizzata su di un browser internet, oltre ad un manuale di riferimento off-line (in LaTeX), da un insieme di files sorgente documentati. La documentazione viene estratta direttamente dai sorgenti, ciò riduce lo sforzo di mantenere la consistenza fra documentazione e codice sorgente.
- Si può visualizzare le relazioni tra i vari elementi includendo grafici di dipendenza, ereditarietà, e collaboration diagrams, che possono essere generati automaticamente.

Doxygen è stato sviluppato sotto Linux e Mac OS X, ma è stato progettato per avere un'alta portabilità, infatti è utilizzabile anche su tutte le versioni di Linux e Windows.

Google Documents

Per scrivere i documenti (come questa relazione) in maniera collaborativa abbiamo deciso di sfruttare le potenzialità online di Google Documents che, oltre ad essere fruibile da qualsiasi piattaforma che abbia un browser, permette la collaborazione sui documenti in maniera Real Time; in questo modo è molto più facile scrivere documenti a quattro mani.

LucidChart

Anche per quanto riguarda la generazione di grafici abbiamo scelto una soluzione online per permettere la collaborazione in tempo reale.
La scelta è ovviamente ricaduta su LucidChart, le cui funzionalità permettono di creare grafici complessi con il minimo sforzo.

L’ambiente Qt

Qt (da pronunciare *chiùt*, come l’inglese *cute*), è una libreria multipiattaforma, oltre che un insieme di tool di sviluppo, per creare programmi con interfaccia grafica tramite l'uso di widget. Ampiamente utilizzato nell'ambiente desktop KDE, viene sviluppato dall'azienda Qt Software.

Qt usa il linguaggio C++ standard con un estensivo uso del preprocessore C Make per arricchire il linguaggio; integra funzioni per l'accesso ai database SQL, parsing di documenti XML ed API multipiattaforma per la comunicazione con il sistema sottostante, astraendo il programmatore dal sistema e riducendo quindi gli errori.

Abbiamo infatti scelto di utilizzare Qt soprattutto per rendere più agevole lo sviluppo di codice multipiattaforma che permettesse anche di utilizzare API di alto livello su file e cartelle, per il modulo di parsing dei file XML integrato, il framework audio e il suo sistema signal/slot.

In secondo luogo l’abbiamo scelto perché fornisce la classe `QGLWidget` che è l’equivalente Qt del GL Utility Toolkit, completamente riscritta per Qt, performante e orientata agli oggetti (a differenza di GLUT che è sviluppata in C e che quindi rendeva più difficile raggiungere l’astrazione strutturale orientata agli oggetti che avevamo in mente).

QGLWidget

La classe `QGLWidget` rappresenta un widget per il rendering di grafiche OpenGL. E’ molto semplice da utilizzare e fornisce tre funzioni virtuali da reimplementare nella sotto-classe per svolgere le tipiche mansioni di OpenGL:

- **`paintGL()`**
- **`resizeGL()`**
- **`initializeGL()`**

Uno dei punti forti di Qt è il meccanismo signal/slot, che è un meccanismo di tipo Publish/ Subscribe che permette, dopo aver connesso il segnale di un oggetto allo slot di un’altro oggetto, di generare degli eventi (in Qt si direbbe “emettere dei segnali”) che richiamano delle funzioni (slot) in altri oggetti in maniera asincrona.

Abbiamo usato molto questo paradigma della scrittura del nostro codice, infatti in questo modo si possono risolvere questioni pesanti con molta leggerezza.

Si veda la descrizione della classe Qubet e della classe AudioManager per vedere come questo paradigma è stato ampiamente sfruttato nella realizzazione del progetto.

Il Progetto

Lo sviluppo del codice

Abbiamo deciso di sviluppare il codice sfruttando in maniera estesa il paradigma Object Oriented. Vediamo qui in generale quali sono le classi che sono state sviluppate per il progetto, in seguito verranno esaminate nel dettaglio le loro caratteristiche e i modi con cui sono stati risolti i problemi che si sono presentati durante lo sviluppo.

La classe principale è Qubet, che estende QGLWidget, ed è la classe che comunica con OpenGL. Ci sono poi le classi Menu, LevelEditor e Game che rappresentano le tre diverse viste del gioco.

Altre classi sono quelle di supporto come ActionList, per gestire le azioni da compiere, le varie Utilities, il controllo delle collisioni con il PositionController e altre piccole classi per gestire praticamente ogni aspetto del progetto da un punto di vista ad alto livello.

Ci sono poi diverse classi che rappresentano le varie entità disegnabili e capaci anche di compiere azioni in autonomia come Skybox, Level, Cube (il nostro personaggio giocante) e una struttura che abbiamo chiamato CubeString per avere il nostro meccanismo di scrittura del testo.

Per disegnare la scena, abbiamo avuto l'idea di delegare gli oggetti stessi al loro disegno sulla scena, per fare questo quindi abbiamo dotato ogni entità disegnabile di un metodo `draw()` che permette loro di disegnarsi nella posizione corrente.
In questo modo abbiamo realizzato una struttura che a noi piace chiamare "albero di disegno", infatti ogni nodo (i nodi sono le istanze delle classi disegnabili) si disegna e, successivamente, avvisa tutti i nodi collegati ai suoi rami di fare altrettanto, ottenendo così una reazione a catena che permette di arrivare fino alle foglie dell'albero e quindi di disegnare l'intera scena.
L'unica eccezione è la classe Qubet, che dopo essersi disegnata non chiama tutti i suoi figli a disegnarsi ma solamente il ramo che rappresenta la vista corrente (Menu, LevelEditor o Game).

Il refresh della schermata viene effettuato utilizzando la classe QTimer, che esegue a intervalli di tempo fissati il metodo `draw()` della classe Qubet. In questo modo possiamo decidere la velocità dei movimenti basandoci sul framerate, dato che è costante (se avessimo usato un loop invece la velocità di refresh, e quindi anche del gioco sarebbe dipesa dall'hardware del calcolatore).

Come ottimizzazione, il metodo `draw` riceve un parametro booleano `simplifyForPicking`, infatti quando la scena viene disegnata ai fini del picking, si può semplificare il risultato, ad esempio evitando le textures e riducendo la qualità delle grafiche, per velocizzare la funzione.

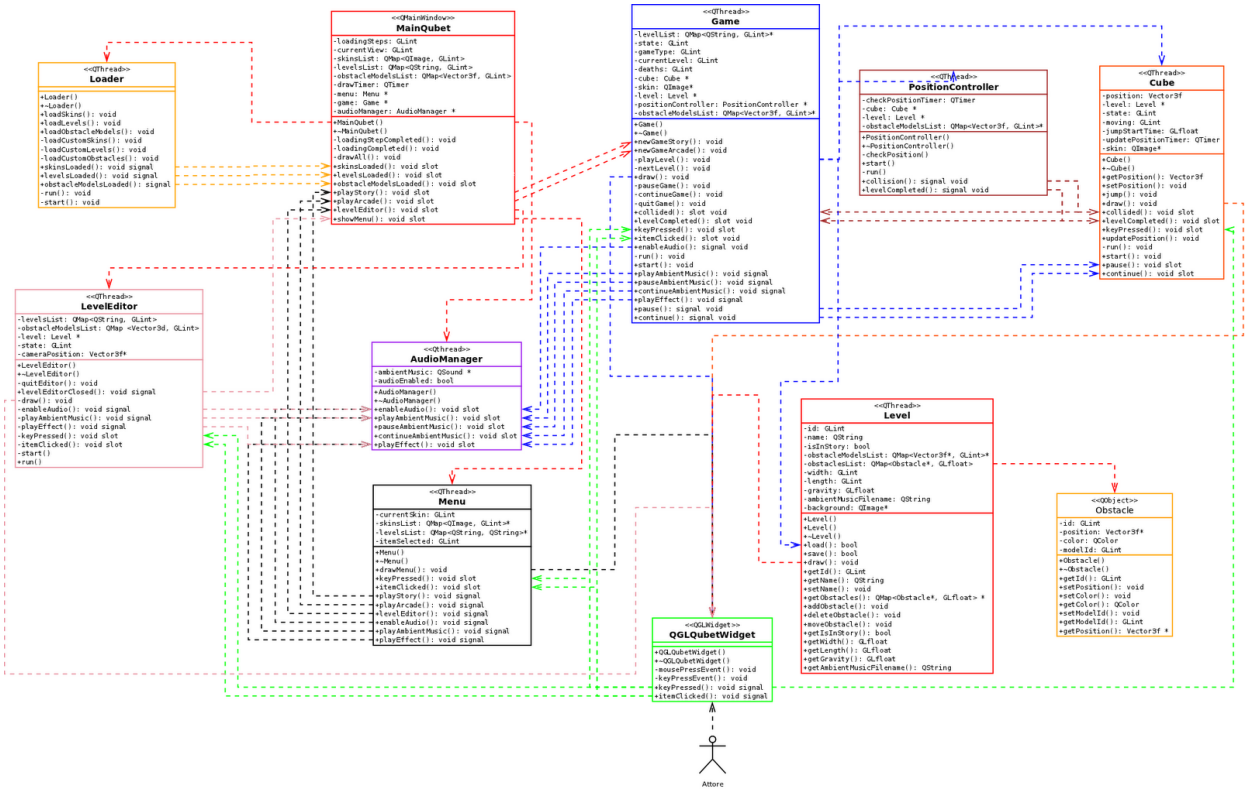
Quando è stato necessario salvare delle strutture di supporto per poterle richiamare all'avvio del programma (come i livelli e le skins) abbiamo fatto uso del formato XML, in modo che fosse facilmente trattabile e i livelli e le skins custom potessero essere condivise tra diversi

giocatori. Per fare questo abbiamo utilizzato il modulo QtXML integrato in Qt per effettuare il parsing e il salvataggio dei dati in strutture XML. In questo documento verranno anche illustrate tali strutture.

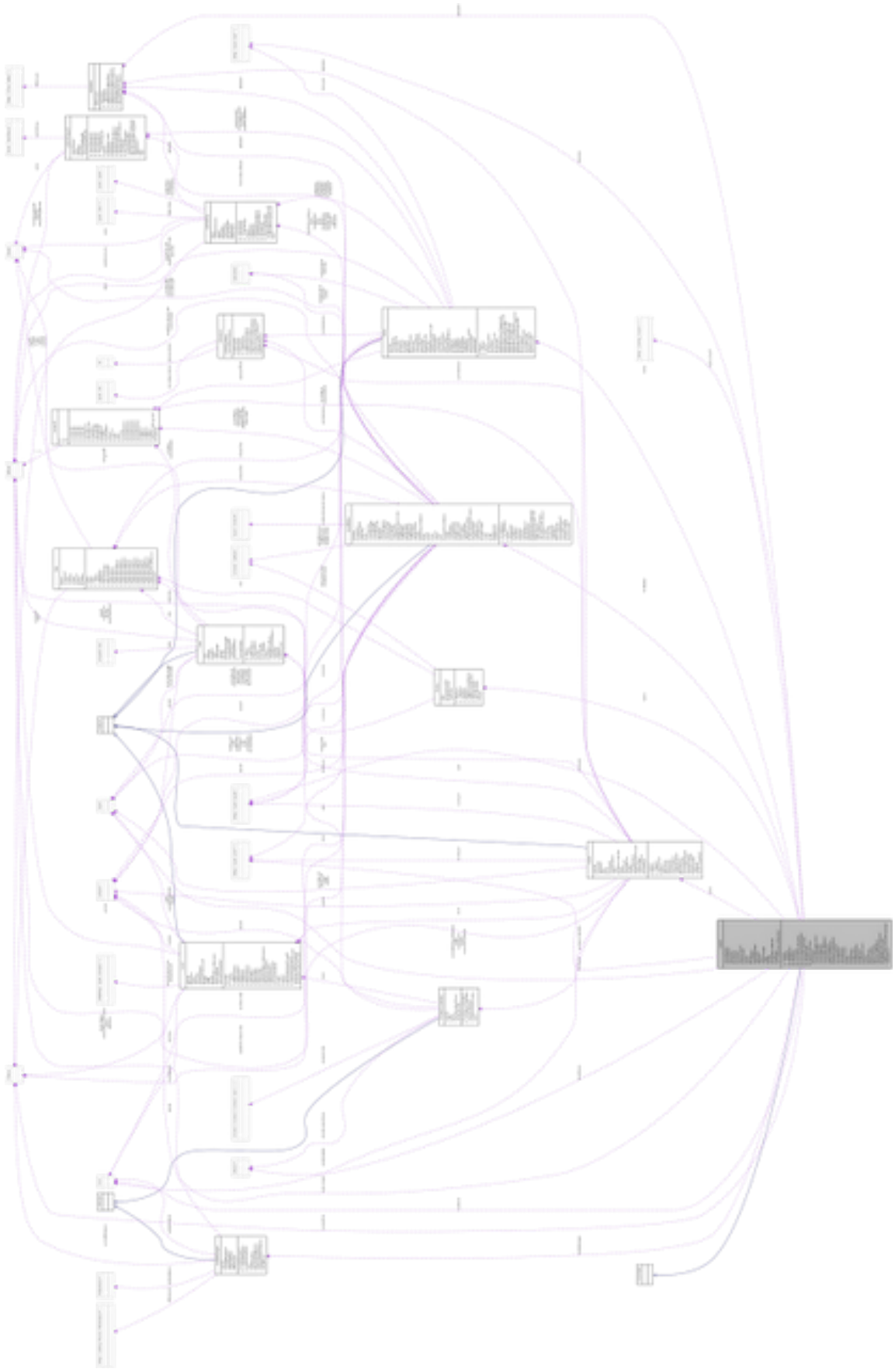
Diagramma delle Classi

Prima di cominciare a scrivere il codice del progetto abbiamo svolto delle attività preliminari quali, una settimana di brainstorming per decidere la struttura del progetto, delle classi e gli strumenti da utilizzare. Una ulteriore settimana per il disegno del diagramma delle classi del progetto.
Riportiamo qui la nostra idea iniziale di diagramma delle classi (abbiamo omesso i parametri dei metodi per motivi di spazio, all'ultima pagina si trovano i link per visionare anche la versione completa in alta definizione), e il diagramma delle classi nella realizzazione finale.

Idea iniziale (le frecce tratteggiate corrispondono a utilizzi e connessioni signal/slot):



Realizzazione Finale (con Collaboration Diagram)
Si noti che nonostante il diagramma finale sia più complesso per via di necessarie modifiche in corso d'opera, la struttura portante di base è la stessa.
E' stato proprio questo studio preliminare approfondito del progetto che ci ha permesso di raggiungere dei risultati (a livello di stesura del codice e sua organizzazione razionale) che a noi sembrano decisamente soddisfacenti.



All’ultima pagina si possono trovare i link alle immagini in alta definizione.

Caratteristiche Principali

Dopo aver visto in generale il progetto, gli strumenti e le tecniche di realizzazione, vediamo in dettaglio alcune delle caratteristiche principali di Qubet.

La classe Qubet

Come già detto, la classe principale è la classe Qubet, che estende la classe Qt QGLWidget, ed è la classe che gestisce la grafica OpenGL.

Come tutti i widget di Qt, anche la classe QGLWidget implementa degli slot che vengono invocati quando l'utente compie delle azioni di input (attraverso mouse e tastiera) sul widget. Abbiamo quindi ideato una struttura che sfrutta il meccanismo signal/slot di Qt per rendere la classe Qubet un dispatcher di segnali.
Questo significa che non è la classe principale a decidere cosa devono fare gli oggetti rappresentati sulla scena, ma gli oggetti stessi, se necessario anche comunicando tra loro.

Essendo gli oggetti a decidere cosa devono fare, essi sono delle entità completamente autonome, infatti come già detto essi hanno la capacità di disegnarsi, ora abbiamo visto che hanno anche la possibilità di decidere in autonomia come interpretare le azioni dell'utente. In questo modo si può modificare il comportamento di un oggetto semplicemente agendo sull'oggetto stesso senza modificare la struttura a monte.

Gli slot di input della classe Qubet quindi, tranne in due casi che vedremo poco più avanti, non fanno altro che rilanciare i segnali che hanno ricevuto, in modo che gli oggetti che si sono connessi a tali segnali possano riceverli. Per comodità abbiamo scritto la funzione `connectInputEvents(QObject *receiver)` per connettere un oggetto a tutti i segnali di input rilanciati da Qubet.

Un altro esempio di utilizzo del paradigma signal/slot è rappresentato dalla classe AudioManager che utilizza proprio questo sistema per ricevere le richieste audio dalle classi connesse ad essa.

Oltre a fare da dispatcher dei segnali, la classe Qubet si occupa anche di leggere le informazioni sulle skins e sui livelli dai files XML, di creare un alfabeto di lettere, di caricare le skybox e le icone, di creare un AudioManager (caricandone anche gli effetti sonori) e di eseguire tutte le operazioni atte ad inizializzare il gioco.

Mouse Clicking

Il mouse clicking è uno dei due slot (insieme al mouse moving che vedremo subito dopo) in cui lo slot di Qubet non si limita a rilanciare il segnale. Infatti questo slot utilizza anche una funzione di picking per determinare l'oggetto in foreground alle coordinate in cui il tasto è stato premuto.
Dopo aver determinato tale oggetto, la funzione costruisce una QList dei nomi di tale oggetto e lancia un segnale contenente tale lista. In questo modo gli oggetti in ascolto di tale segnale non devono conoscere come si effettua il picking, ma possono usare direttamente la lista dei nomi dell'oggetto in foreground per i loro scopi.

Questo permette un incapsulamento della funzione di picking all'interno di Qubet.

Si noti che più oggetti possono essere in ascolto sullo stesso segnale, questo quindi è un meccanismo ideale per gestire gli eventi.

Mouse Moving

Attraverso il segnale `setMouseMovementTracking(int mode)` (che ha come destinazione Qubet), le classi disegnanti possono specificare quando deve essere lanciato il segnale `mouseMoved`.
Le scelte sono tra:

- mai.
- sempre quando il mouse viene mosso.
- solo quando il mouse viene mosso e un tasto del mouse è premuto.

Questo è stato fatto perché, salvo alcuni casi, non è necessario sapere quando il mouse viene mosso senza alcun tasto premuto. Così facendo si lascia la decisione alla classe sulla granularità di questo segnale, ottimizzando le risorse del calcolatore.

Questo segnale inoltre utilizza la stessa funzione di picking discussa nel paragrafo precedente per integrare nel segnale anche una lista di nomi dell'oggetto in foreground che si trova sotto il mouse.

La gestione dell'audio

Introduciamo innanzitutto il framework Phonon, in seguito parliamo della classe AudioManager, la quale istanza si occupa di gestire l'audio di Qubet.

Phonon

Phonon è il framework multimediale di Qt, che fornisce delle API semplici in grado di gestire qualsiasi backend multimediale per cui esista un plugin. Al momento i backend utilizzabili sono Gstreamer, VLC media player (il backend ufficiale) e Xine (non più supportato).

Phonon fornisce una classe MediaObject, la quale istanza permette di lanciare i segnali base per una traccia audio (come play, pause, stop).

Dopo aver definito quale sia la traccia è quindi sufficiente lanciare il segnale `play()` per eseguire la traccia ed il framework provvederà a compiere le azioni ed i controlli necessari per aprire un canale sul dispositivo di output e lanciare la traccia audio.

AudioManager

Abbiamo creato una classe apposita per gestire l'audio in Qubet: la classe `AudioManager`, che sfrutta oggetti di tipo MediaObject per eseguire la colonna sonora e gli effetti.

Questa classe è l'altro esempio oltre alla classe Qubet di utilizzo del paradigma signal/slot, infatti l'AudioManager è un QThread a sé stante, che riceve istruzioni dagli oggetti che sono stati connessi a lui attraverso i seguenti segnali (e i rispettivi slot):

- **`enableAudio(bool enabled)`**
  In caso di ricezione di questo segnale se la variabile `enabled` è `true` e `audioEnabled` è `false` (si vuole attivare l'audio), `audioEnabled` viene settata a `true` e viene lanciato il segnale `play` di `ambientMusic`. Viceversa, viene lanciato il segnale `pause()` di `ambientMusic` e vengono fermati tutti gli effetti sonori.

- **`playAmbientMusic(QString filename)`**
  Viene istanziato un nuovo MediaObject `ambientMusic` , con riferimento al file audio la cui path è la stringa `filename`, dopodiché viene lanciato il segnale `play()` di tale oggetto.

- **`playEffect(QString effectName)`**
  Il comportamento di questo slot è simile al precedente; per gli effectName si possono usare le define definite nel file effects_defines.h caratterizzati dal prefisso "EFFECT_" .

- **`stopAmbientMusic()`**
  Questo slot lancia il segnale `clear()` di `ambientMusic`, in modo che l'istanza di `MediaObject` che rappresenta la musica di sottofondo, fermi lo stream audio.

- **`enqueueMediaObject()`**
  Questo è uno slot privato che permettere la ripetizione in loop della `ambientMusic`.

Per connettere tutti questi slot in un comando solo, abbiamo definito la funzione `connectAudio(QObject *sender)` in Qubet.
Questo esempio e quello precedente della classe Qubet, ci permettono di apprezzare la versatilità e la facilità di utilizzo del paradigma signal/slot di Qt.

Utilities

Abbiamo definito alcune funzioni di utilità generali, utilizzate da diverse classi, che abbiamo unito nella libreria utilities.h (ed il corrispondente .cpp).

Le riassumiamo qui in breve:

- **`drawRectangle(GLfloat x, GLfloat y, GLuint texture = 0)`**
  Disegna un rettangolo con lunghezza `x` e larghezza `y`, dispone di un parametro facoltativo `texture` che se diverso da zero indica la texture da applicare al rettangolo.

- **`drawPrism(GLfloat x, GLfloat y, GLfloat z, Skin *skin = NULL, GLboolean invertBackTexture = false)`**
  Disegna un prisma con `x`, `y`, `z` indicate tramite `GL_QUADS`, il parametro facoltativo `skin` indica la skin da applicare al prisma e `invertBackTexture` causa se true l' inversione della texture applicata alla faccia z-. Quest'ultimo parametro è `true` quando si disegnano le `CubeString`, così le texture posteriori vengono invertite e le lettere sono leggibili anche dopo la rotazione del cubo.

- **`drawObstacle(GLuint id)`**
  Disegna l'ostacolo di tipo `id`. Gli ostacoli più semplici sono disegnati da chiamate di `drawPrism`, mentre gli altri sono disegnati tramite `GL_TRIANGLE_STRIP` e `GL_QUADS`.

- **`setTextureClampToEdge()`**
  Questa funzione evita artifact (glitch) sugli spigoli di giunzione di prismi , la chiamata di questa funzione imposta il parametro `GL_CLAMP_TO_EDGE` sia per `GL_TEXTURE_WRAP_S` che per la `GL_TEXTURE_WRAP_S`, in questo modo la texture si estende anche agli spigoli.

- **`getModelViewPos(Vector3f *vect, bool computeZDepth = false)`**
  A partire dalle coordinate del mouse relative alla finestra, restituisce le coordinate del punto indicato da `vect`. Se `computeZDepth` è `true` si calcola la `z` del primo oggetto in foreground puntato dal mouse, altrimenti si usa la `z` di `vect`.

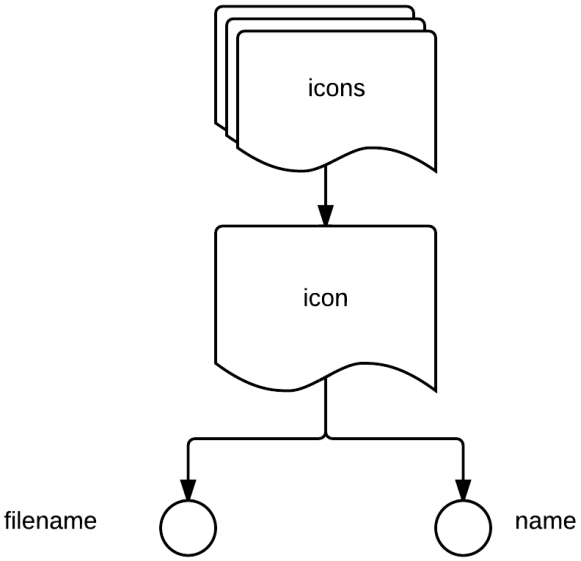
- **`getProjectionPos(Vector3f *vect)`**
  Restituisce le coordinate relative alla finestra a partire da un punto `vect` fornito in coordinate relative alla ModelView.

- **`getPointFromParametricLine(Vector3f *p0, Vector3f *p1, GLfloat t)`**
  Costruisce una retta parametrica passante per i punti `p0` e `p1` e restituisce il punto in cui il parametro di tale retta è uguale a `t`.

- **`getObstacleBoundingBox(GLuint id)`**
  Restituisce la dimensione della bounding box relativa all' ostacolo `id`.

Icons

Le icone sono delle generiche immagini che si vogliono caricare per usi generici. Ogni icona può anche avere un nome, che servirà poi per identificarla.



struttura del file icons.xml

La struttura XML del database delle icone prevede una serie di elementi **icon**, che hanno come attributi obbligatori:

- **filename**: percorso relativo del file.
- **name**: nome con cui tali icone sono state definite all' interno del codice (se come nome viene usato un id numerico, potremo anche usare questo nome poi come nome OpenGL per identificare l'oggetto durante il picking).

Cube

La classe Cube rappresenta il personaggio giocante, questo ha quindi una Skin, si disegna autonomamente quando il gioco è nella Game View ed è inoltre in ascolto sui segnali di input dell'utente. Quando quindi l'utente interagisce con il gioco, è direttamente all'istanza di questa classe che verranno inviati i comandi; essa quindi si occuperà di effettuare i dovuti controlli e quindi, se possibile, di eseguire l'azione richiesta dall'utente, come gli spostamenti o il salto.

Skins

La classe Skin contiene, oltre ad altre informazioni, un insieme di 6 immagini che andranno a creare una veste di un cubo. Nel progetto vengono utilizzate abbondantemente, ad esempio per le Skybox, le CubeStrings e per le skin dei personaggi giocanti Cube, che andremo ora a analizzare.

Personaggi

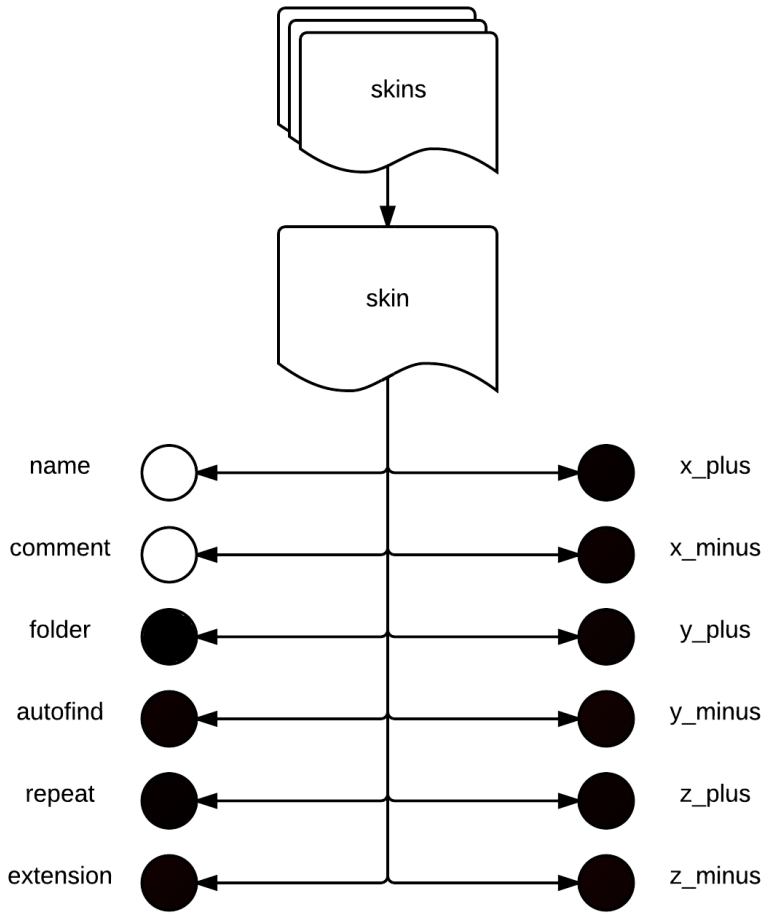
In Qubet è possibile scegliere la veste del proprio personaggio giocante dal Menu. Abbiamo deciso di creare una struttura XML che mantenesse le informazioni sulle varie Skin disponibili, in modo che siano facilmente estendibili dai giocatori che abbiano voglia di aggiungerne delle altre in maniera semplice.

Il database XML delle skins prevede una serie di elementi **skin**, i quali hanno 3 parametri obbligatori:

- **name**: nome della skin.
- **comment**: descrizione della skin.

Oltre a questi vi sono dei parametri opzionali:

- **folder**: nome della cartella che contiene le facce della skin.
- **autofind**: cerca nella cartella file nominati x+, x-, y+, y-, z+, z- con l' estensione indicata dall'attributo **extension**.
- **repeat**: se uguale a `true` si vuole indicare che la skin abbia una sola immagine ripetuta per tutte le facce del cubo.



struttura del file skins.xml

- **extension**: utilizzabile solo quando **autofind** è `true`, indica l'estensione dei files da caricare. Se non viene specificata, l'estensione di default è .png .
- **x**, **y**, **z**, **_minus** e **_plus**: nome del file dell'immagine della rispettiva faccia.

Quindi si hanno tre modi per definire una skin di personaggio:

1. Usando l'autofind e i nomi dei file predefiniti.
2. Usando il parametro repeat per ripetere la stessa immagine su tutte le facce.
3. Definendo una a una le facce.

I "modder" vedono sempre di buon occhio i giochi che permettono di customizzare i personaggi con facilità (generalmente si divertono a dare la propria faccia al personaggio giocante), e in questo caso sono accontentati, basta infatti pochissimo per aggiungere una nuova skin.

CubeStrings

Uno dei primi problemi che abbiamo dovuto affrontare durante lo sviluppo è stato quello di mostrare al giocatore delle scritte a video. A questo problema c'erano 3 diverse soluzioni:

1. Utilizzare il metodo renderText della classe QGLWidget.
2. Utilizzare librerie esterne per la gestione del testo tridimensionale come FreeType.
3. Affidarci alla nostra fantasia.

Il primo metodo non ci soddisfaceva particolarmente trattandosi di un testo bidimensionale, difficilmente gestibile, lo abbiamo usato solo quando strettamente necessario per le comunicazioni di servizio.
Il secondo metodo, nonostante lo avessimo provato, non ci soddisfò per il fatto di dover utilizzare delle librerie esterne, contrarie alla logica del nostro programma che voleva fondarsi sul solo utilizzo di Qt in modo da poter essere eseguito su qualsiasi macchina con installate tali librerie; lo abbiamo quindi scartato, nonostante sia una soluzione molto spesso utilizzata.

Abbiamo così optato per la terza soluzione, abbiamo preso spunto da quei giochi per bambini che consistono nell'accostare dei cubi con impresso delle lettere sulle facce per creare parole. Questa idea, inoltre, si sposava bene con il concetto fondamentale del gioco: il cubo.

Per fare questo abbiamo creato diverse classi, costruendo una struttura altamente riutilizzabile:

Letter

La classe Letter rappresenta una lettera (o un numero, o un simbolo). Ogni Letter possiede una variabile QChar che identifica il carattere e una lista di textures che raffigurano tale carattere.

Alphabet

La classe Alphabet è una lista di oggetti di tipo Letter privati, in questo modo, il programmatore non interagisce mai con la classe Letter direttamente, ma solo con Alphabet; è questa classe infatti che si occupa di mantenere la lista di Letter, in modo da poter effettuare maggiori controlli su di esse.
La classe Alphabet fornisce dei metodi di inserimento di una texture relativa a una lettera in Alphabet (si occuperà poi questa classe di verificare se c'è la necessità di creare una nuova Letter oppure di riutilizzare quella già creata).
Vi sono poi tre tipi di interrogazione della classe Alphabet:

1. Estrazione di una texture casuale di una lettera.
2. Estrazione di n textures casuali di una lettera (abbiamo anche sviluppato un leggero algoritmo a tempo deterministico per garantire, se possibile, che le textures estratte non siano duplicate).
3. Estrazione di un oggetto di tipo Skin che rappresenti un cubo di textures di una lettera (richiama il secondo metodo, garantendo, se possibile, la non duplicità delle textures).

Nella directory `/resources/letters` vi sono delle cartelle, ognuna nominata con la lettera di cui contiene le immagini. La classe Qubet è incaricata di creare l'alfabeto, per farlo

controlla, utilizzando le classi QDir e QFile di Qt, tale cartella, creando una Letter per ogni cartella e caricando in essa le textures contenute.
In questo modo è estremamente facile modificare l'alfabeto che verrà caricato nel gioco, basta infatti modificare le sottocartelle di `/resources/letters`, e i files immagini in esse contenuti.

CubeString

Le CubeString quindi sono delle stringhe formate dall'accostamento di diversi di questi cubi di lettere. Esse sono largamente usate nel progetto per creare delle scritte d'impatto.
I costruttori di questa classe permettono di gestire la stringa da scrivere, la grandezza dei cubi e il nome OpenGL da dare all'insieme di cubi di lettere quando verranno disegnati.
Come ottimizzazione, i singoli cubi delle CubeString utilizzano delle **displayList** per velocizzare il processo di disegno. (Il codice è stato strutturato in modo tale che sia possibile scegliere se utilizzare le displayList o meno semplicemente modificando una define, per garantire maggiore compatibilità con hardware grafico più datato).

CubeStringList

E' stata anche sviluppata una classe che consente l'inserimento di più CubeStrings all'interna di una lista: la CubeStringList. Questa classe è utile per creare dei testi o dei menu, nel secondo caso è anche possibile definire un nome OpenGL diverso per ogni stringa della lista in modo da poter compiere azioni diverse quando vengono cliccate diverse voci del menu.

Animazioni

Sia la classe CubeString che la classe CubeStringList, mettono a disposizione del programmatore dei metodi per poter animare le stringhe in maniera facilitata ed intuitiva. Attraverso l'utilizzo di questi metodi è facile creare delle semplici animazioni per le CubeStrings.

Skybox

Per risolvere la questione dello scenario abbiamo deciso di utilizzare la tecnica Skybox, nella quale si utilizza un cubo con applicate delle texture sulle facce per simulare un ambiente, in questo modo è stato possibile riutilizzare la classe Skin all'interno della classe Skybox.
Le immagini che costituiscono le varie skybox si trovano in delle sottocartelle in `/resources/skyboxes` . Quando viene lanciato il gioco, la classe Qubet si occupa di scandire tale cartella e creare una Skybox per ogni sottocartella, salvando poi le varie skybox in una lista che viene poi utilizzata per il disegno della skybox.
La Skybox che si vede durante il gioco è disegnata direttamente dalla classe Qubet, abbiamo quindi creato un segnale che possono usare le altre classi per cambiare laskybox corrente.
Abbiamo deciso di lasciare a Qubet il compito di disegnare la skybox per due motivi: la skybox è unica e in questo modo il meccanismo è unico, in secondo luogo ci serviva che essa fosse disegnata come prima cosa; infatti per disegnare la skybox disabilitiamo momentaneamente il **DEPTH TEST** di OpenGL, questo ha come effetto che OpenGL non cercherà di intersecare i successivi oggetti con la Skybox per garantire il senso di profondità. In questo modo si avrà l'impressione che l'ambiente è veramente posto all'infinito. Diversamente gli oggetti potrebbero essere tagliati dal cubo della skybox (e non solamente dal near e dal far della perspective).

Livelli

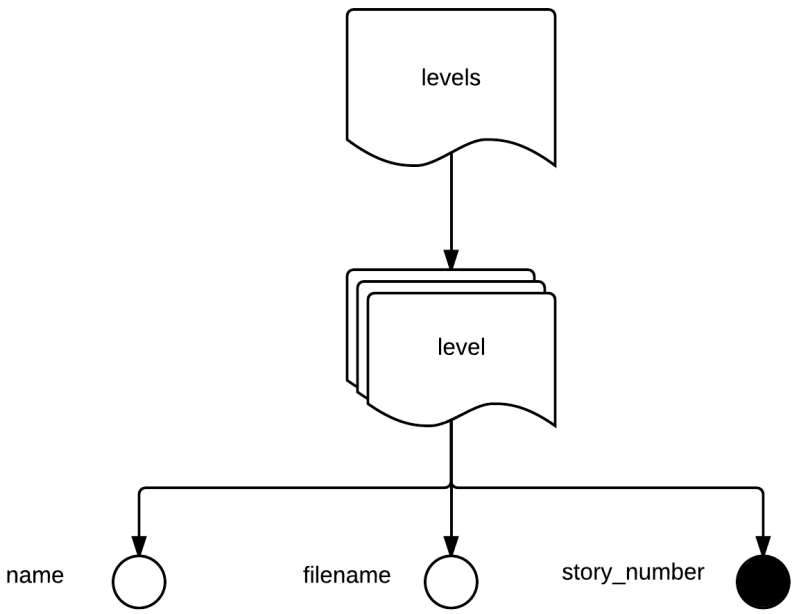
Abbiamo deciso di mantenere separate le informazioni generali del livello (utilizzate già nel menu), dalle informazioni intrinseche del livello, come ad esempio la lista dei suoi ostacoli e le sue proprietà. Questa struttura a doppio strato è infatti regolarmente usata in tutti i giochi, che per velocizzare i tempi di caricamento iniziali, caricano solo le informazioni essenziali sui livelli, posticipando il caricamento degli altri dati a quando il giocatore sceglie effettivamente a quale livello è interessato.

Il database dei livelli prevede una serie di elementi **level** i quali hanno come attributi:

- **name**: nome del livello
- **filename**: percorso relativo del file che contiene l'XML del livello in questione.

Attributo opzionale:

- **story_number**: se presente indica il numero del livello nella modalità storia; altrimenti il livello viene flaggato come arcade.



struttura del files levels.xml

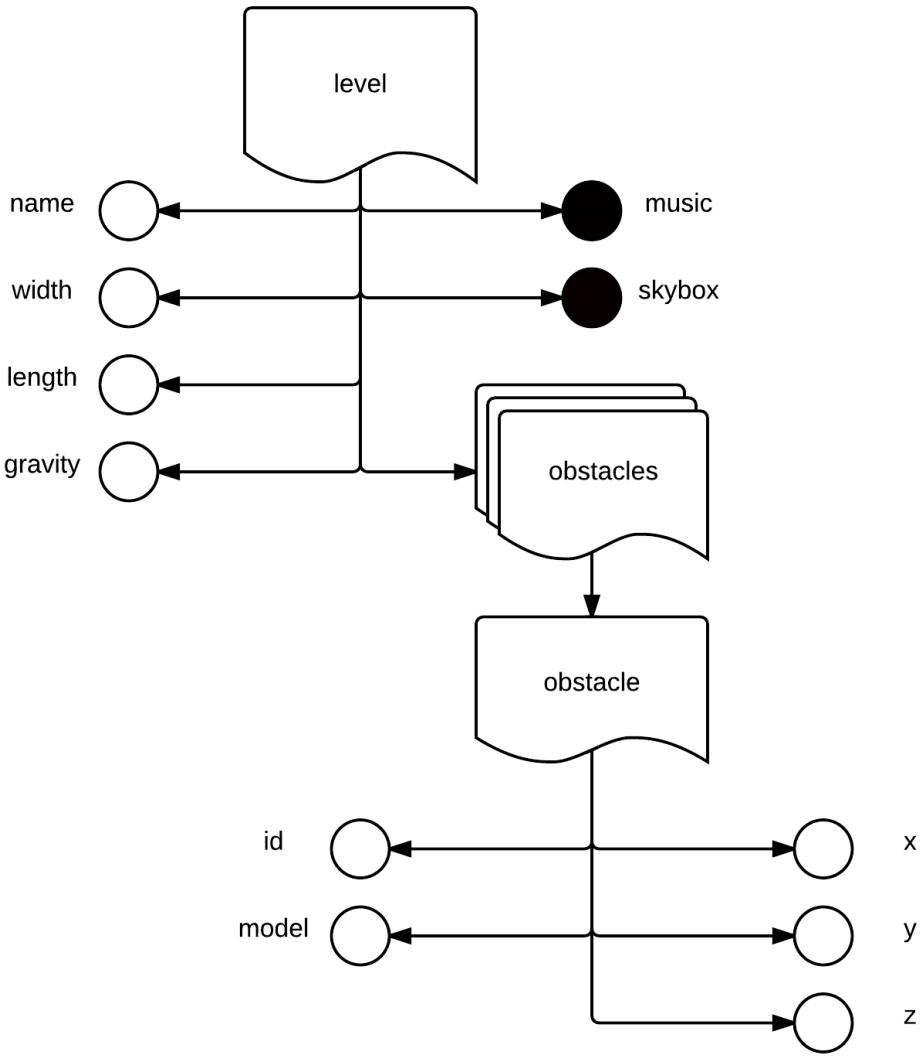
La classe incaricata di caricare la lista dei livelli è `Qubet`, che però si preoccupa solo di creare i livelli presenti in lista inserendone solo il nome ed il parametro relativo alla storia. Chi invece si occupa di caricare ogni singolo file XML contenente ogni dato relativo al livello è la classe `Level` tramite il metodo `load()`. Segue la descrizione della struttura dei file XML di ogni livello.

Ogni file xml **level** ha i seguenti attributi:

- **name**: nome del livello.
- **length**, **width**, **gravity**: lunghezza, larghezza e gravità del livello.
- **music**: nome del file audio da riprodurre durante il gioco.
- **skybox**: nome della skybox da applicare.

Contiene inoltre una serie di elementi **obstacle**, i quali hanno i seguenti attributi:

- **id**: numero identificativo dell'ostacolo.
- **model**: tipo dell'ostacolo fra i quattro modelli disponibili.
- **x**, **y**, **z**: coordinate di cella dell' ostacolo rispetto al livello.



struttura dei files dei singoli livelli

Menu

Il menu è composto da 4 viste: introduction, main, skins e levels view.
Le varie viste sono disposte in un unico ambiente di disegno e la transizione tra esse avviene attraverso delle animazioni che modificano il punto di vista della camera, come rotazioni e spostamento.
Ogni scelta è accompagnata da effetti audio che si sovrappongono alla colonna sonora.
All'ingresso di ogni vista viene abilitata la possibilità di proseguire nelle scelte premendo dei tasti da tastiera.

Introduction View

Questa vista è dedicata esclusivamente ad un'animazione composta da CubeStrings, in cui viene presentato il gruppo ed il nome del progetto.

Main View

La vista principale è composta da tre CubeStrings che permettono la scelta della modalità di gioco: "Story", "Arcade" o "Editor", la selezione delle prime due porta alla vista di selezione delle skin per il cubo mentre l'ultima porta alla vista per la selezione del livello da modificare.

Skins View

In questa vista è presente il cubo, i comandi che permettono la scelta della skin e la descrizione di quest'ultime.
Il cambio della skins avviene inoltre durante un animazione.
Dopo aver effettuato la scelta si viene proiettati nella vista dedicata al gameplay.

Levels View

Questa vista permette di scegliere il livello da modificare, oppure crearne uno nuovo.

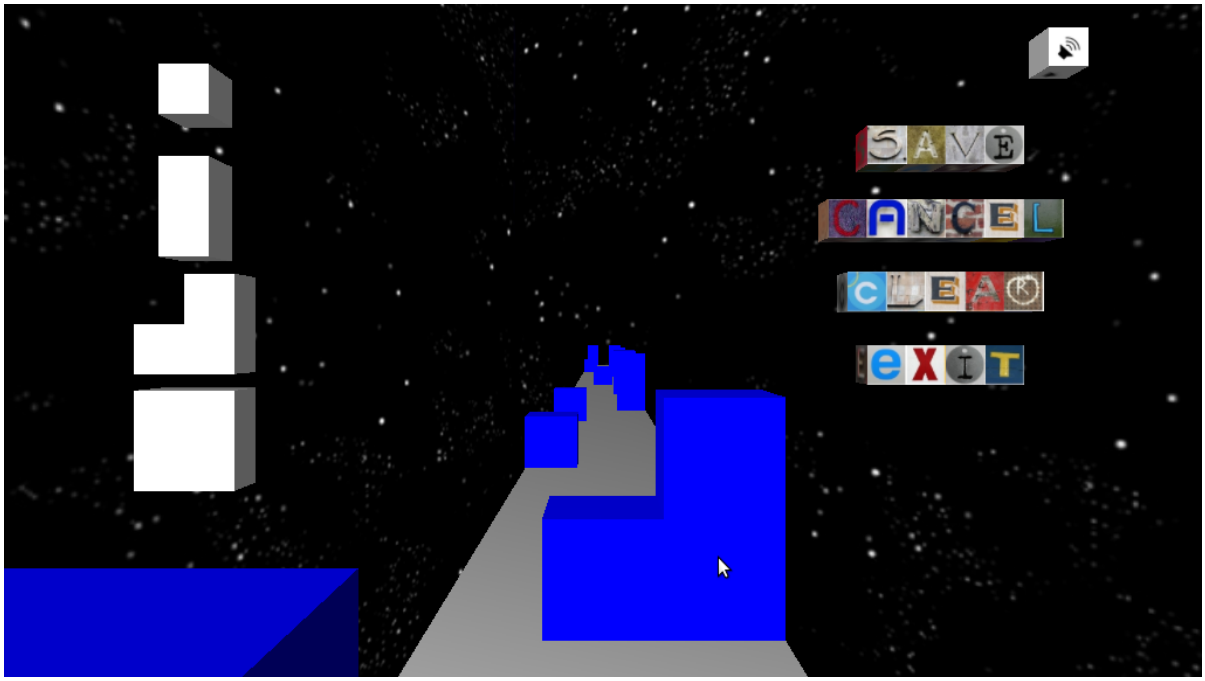
Level Editor

Nel level Editor si possono modificare i livelli: si può modificare il nome, la lunghezza, la larghezza e la gravità del livello, si possono sistemare gli ostacoli nel livello a proprio piacimento attraverso il picking da una toolbar laterale.

Come già descritto parlando dei livello, questi sono divisi a celle, ognuna della dimensione del personaggio giocante. Abbiamo quindi calcolato la cella relativa alla posizione del mouse per posizionare correttamente gli ostacoli trascinati sul livello.

Per fare questo abbiamo ovviamente dovuto risolvere dei problemi parametrici, considerando anche la posizione della camera e l'offset del livello.

Il caricamento ed il salvataggio sono effettuati secondo la struttura XML dei livelli presentata in precedenza.



Game

La fase di gioco prevede una breve introduzione prima del via, momento nel quale il livello trasla velocemente e l'utente deve comandare il cubo in modo da non farlo collidere con gli ostacoli. Nel momento in cui il cubo tocca un ostacolo si esegue un'animazione e si reinizia il livello corrente. Se si arriva alla fine del livello e si è in modalità arcade si viene riportati al menu se invece si è in modalità storia si entra nel livello successivo, nel caso il livello appena finito fosse l' ultimo si viene ancora riportati al menu iniziale.

ActionList

Durante la fase di programmazione si è presentato il problema di dover gestire diverse animazioni contemporaneamente e talvolta con politiche di esclusione. Ad esempio le animazioni portano alle diverse viste del menu si escludono a vicenda, mentre la rotazione del cubo che attiva o disattiva il volume deve avvenire in maniera concorrente alle altre azioni.
Si è pensato dunque di creare una classe ActionList che contiene un valore intero `primaryAction` che identifica l'azione primaria, ed una lista di valori interi, `secondaryActions,` che identificano le azioni secondarie.
Le azioni primarie si settano tramite `setPrimaryAction(int _primaryAction)`, mentre quelle secondarie tramite `appendSecondaryAction(int _secondaryAction)`,
In ogni chiamata di `draw` si estraggono le informazioni dall' istanza di AcionList mediante `getPrimaryAction()`, `getSecondaryActions()`, o `getAllActions()` e si esegue uno `switch-case` in cui viene eseguita una sola azione primaria, e tutte le azioni secondarie all'interno della lista, una volta che l'azione primaria termina è sufficiente settare `primaryAction` a `0`, quando termina una azione secondaria si invoca il metodo `removeSecondaryAction(int _secondaryAction)` per rimuoverla.

Collision Detection

Abbiamo deciso di sviluppare un'apposita classe che estendesse QThread, con un proprio flusso di controllo attivo durante il Game, che controllasse le posizioni relative del giocatore (rappresentato dalla classe Cube) e degli ostacoli del livello.

Abbiamo chiamato questa classe, senza eccedere in fantasia: PositionController.

Controllare ad ogni frame le posizioni di tutti gli ostacoli e delle possibili intersezioni tra questi e il cubo sarebbe stato un compito computazionalmente oneroso.

Quando dunque andiamo a instanziare un PositionController (fornendogli nel costruttore un puntatore a un oggetto di classe Level), questo crea una matrice tridimensionale di valori booleani. Le celle della matrice tridimensionale rappresentano le celle in cui è suddiviso lo spazio del livello di Qubet, il valore booleano informa se la cella è vuota oppure occupata da un ostacolo (o da una sua parte).

In questo modo per verificare se il Cube ha colliso con un ostacolo, basterà verificare le celle occupate dai suoi otto vertici (facendo attenzione al comportamento se il vertice è sul bordo di una cella) e controllare se tra queste, almeno una è occupata da un ostacolo. In quel caso si ha una collisione.
In questo modo il numero di operazioni per la collision detection si riduce di molto, velocizzando il programma.

Conclusioni

Cose che avremmo voluto includere

Sono molte le cose che avremmo voluto includere nel progetto ma che tuttavia, soprattutto per la mancanza di un componente del gruppo, abbiamo dovuto tralasciare.

Anche se abbiamo cercato comunque di non tralasciare gli aspetti funzionali, teniamo qui una lista di idee che ci sono venute durante lo sviluppo del codice, in modo che se in un futuro avremo il tempo di implementarle, avremo già una base di partenza:

- **Visualizzazione**:
  Trasparenza degli oggetti che si trovano tra il punto di vista e il centro della scena.

- **Utilizzo degli Shaders**:
  Ad esempio per il moto particellare dovuto all'esplosione del cubo quando collide con un oggetto.

- **Funzioni che migliorano la giocabilità**:
  Ad esempio Doppi salti, invulnerabilità, rimpicciolimento, bonus, high-scores, possibilità di utilizzare il mouse per interagire con i livelli e con gli ostacoli, accelerazioni e decelerazioni, modalità Surviror (una sola vita).

- **Funzioni che migliorano l'Editor dei livelli**:
  Come la possibilità di inserire ostacoli ad una altezza diversa da quella di default, permettere la rotazione, lo spostamento e l'eliminazione singolare degli stessi.

Qt 4.8 e Qt3D

Abbiamo sviluppato il progetto utilizzando le librerie Qt 4.7. Tuttavia avremmo potuto usare la development version di Qt 4.8 che include le librerie Qt3D che implementa numerose classi e metodi utili per gestire ambienti tridimensionali.

Tutte le entità principali di OpenGL, compresi gli Shader e i gestori delle textures hanno il loro corrispettivo nelle librerie Qt3D, e queste classi, oltre a garantire un'ottima stabilità, forniscono anche numerosi metodi utili per scrivere codice in maniera agile, mantenendo tuttavia un grado molto alto di affidabilità e performance.

Per una questione didattica abbiamo voluto scrivere noi il codice per gli oggetti OpenGL, tuttavia l'uso di Qt3D semplifica di molto lo sviluppo di applicazioni OpenGL in Qt.

Diffusione del codice

Lo sviluppo dell'applicazione è stato portato avanti solamente dai componenti del gruppo, tuttavia, anche grazie all'uso di BitBucket per lo sharing e la collaborazione, abbiamo deciso che, non appena l'uso accademico del progetto ai fini del corso di Informatica Grafica sarà concluso, il repository di codice sorgente di Qubet sarà aperto a tutti, in maniera Open Source sotto licenza GPL3, in modo che se qualcuno volesse continuarne lo sviluppo sarebbe libero di farlo.

I canali di diffusione possibili che si pensano di utilizzare (oltre al codice su BitBucket) sono:

- Launchpad (che utilizza il VCS Bazaar) per la diffusione in ambienti Ubuntu
- OpenSUSE BuildService per la diffusione in ambienti Linux generici
- Sourceforge e Google Code per la visibilità del progetto.

Conclusione

Siamo molto soddisfatti del nostro progetto, e anche se ci ha richiesto molto più tempo di quanto avessimo preventivato, ci ha permesso di prendere conoscenza e dimestichezza con alcuni degli strumenti e dei paradigmi più usati nell'Ingegneria del Software degli ultimi anni.

Alla data odierna il progetto consta di più di 5000 righe di codice, 2000 righe di commenti e 180 revisioni nella repository del progetto.

Collegamenti

Alla pagina **minus.com**/mzHV9eWWu si trova la galleria con tutte le immagini relative a Qubet in alta definizione e altro materiale sul progetto (come questa relazione).

Il servizio online min.us permette la condivisione di files in maniera agile, ed è stato scelto per l'hosting della documentazione del progetto.

Collegamenti agli strumenti citati nella relazione:

- **qt**.nokia.com, sito ufficiale dell'ambiente di sviluppo integrato Qt e delle librerie.
- **mercurial**.**selenic**.com, sito ufficiale di Mercurial SCM.
- **tortoisehg**.bitbucket.org, sito ufficiale di TortoiseHG.
- **bitbucket**.org, social hosting di repositories Mercurial.
- www.**doxygen**.org, sito ufficiale di Doxygen.
- **dev.kofee**.org/projects/qtcreator-doxygen, plugin Doxygen per Qt Creator.
- **docs**.**google**.com, strumenti office online Google Documents.
- www.**lucidchart**.com, strumento collaborativo online per la creazione di grafici.