UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Ingegneria Informatica

Elaborato finale in **Programmazione I**

# *Programmazione C/C++ per lo sviluppo di applicazioni grafiche*

Anno Accademico 2018/2019

Candidato:

**Roberto Basile Giannini**

**matr. N46/2549**

*A mio padre,*
*alla mia famiglia e*
*a Delia*

# Indice

# Introduzione

Con il presente elaborato si vuole trattare l'applicabilità del linguaggio di programmazione C++ nel contesto dello sviluppo di applicazioni grafiche, ponendo un accento particolare a quelle che possono essere definite come applicazioni grafiche particolarmente complesse: i *videogiochi*. Lo scopo è sottolineare come gli strumenti offerti dal linguaggio possono essere utilizzati in tal senso, rendendo realizzazioni in passato proibitive relativamente semplici e prestazionalmente notevoli. La relativa semplicità è legata alla forte astrazione caratteristica del linguaggio in oggetto, in perfetto contrasto con i linguaggi di programmazione a basso livello adottati durante la realizzazione delle prime applicazioni grafiche, soprattutto in termini di videogiochi. Si preda il caso del primo videogioco largamente distribuito, quale *Spacewar!*, che nel 1961 venne principalmente scritto da Steve Russel in linguaggio assembly per il PDP-1, impiegando (solo per la versione base) circa 3 mesi[1]. Il paradigma ad oggetti, l'ereditarietà, il polimorfismo ed altre peculiarità del C++, accompagnate da ricchissimi supporti esterni, rappresentano degli strumenti potentissimi al servizio dei programmatori (di videogiochi e non). Nei primi capitoli dell'elaborato ci si soffermerà sul paradigma ad oggetti, sugli strumenti fondamentali del C++, ma anche sulla gestione della memoria garantita dal linguaggio in questione, osservando come l'opzionalità di un *garbage collector* può condizionare le prestazioni di un videogioco. La seconda parte dell'elaborato evolverà invece nel verso dei supporti

esterni per lo sviluppo, introducendo il paradigma *event-driven* e ponendo maggiore attenzione ad un framework largamente utilizzato nella realtà delle applicazioni grafiche: *Qt*. Nell'ultimo capitolo verrà implementato un piccolo videogioco, quale *Carta, sasso, forbici, lizard, Spock*.

# Capitolo 1: Object Oriented Programming

In questo capitolo tratteremo la OOP applicata al contesto dello sviluppo basilare di videogiochi. In particolare, ci soffermeremo sull'utilizzo di strumenti fondamentali come l'ereditarità e il polimorfismo, avendo come linguaggio di riferimento il C++.

## 1.1 Dalle entità alle classi

Il C++ è un linguaggio di programmazione ad alto livello sviluppato da Bjarne Stroustrup nel 1983. Presentato come evoluzione del linguaggio C, il C++ permette la programmazione orientata agli oggetti, ovvero un paradigma di programmazione che prevede l'utilizzo di oggetti (moduli software) che operano tra loro tramite scambio di informazioni. In dettaglio, un oggetto altro non è che un'istanza di una classe. Definiamo classe come un tipo di dato astratto, ovvero un componente software che rappresenta un'entità reale o concettuale, costituto dalla struttura dati che vogliamo modellare e dalle operazioni consentite su di essa. Queste operazioni vengono definite come *metodi* e possono essere pubbliche (ovvero accessibili al cliente della classe) oppure private (inaccesibili dall'esterno): un qualsiasi utente della classe, per operare con l'entità che rappresenta, dovrà sempre passare per l'interfaccia della stessa. In un contesto come quello dei videogiochi, dove il numero di entità può essere molto elevato, avere uno strumento che permette di descrivere delle astrazioni è estremamente importante. Si pensi ad un semplice elemento di gioco, come per esempio il *player*: esso avrà degli attributi (salute, stamina, esperienza, ecc) e dei metodi (movimento, attacco, ecc). Per renderci

conto dell'applicabilità del C++ sotto questo punto di vista, consideriamo un semplice esempio: *Tetris*. In Tetris l'unica astrazione da considerare (escludendo quelle funzionali al gioco in sé, come l'*heads-up display*, e quelle strutturali dell'architettura adottata) è il tetramino. Come attributi possiamo prendere in esame la *forma*, il *colore* e la *rotazione*. In base al valore di questi attributi, definiamo lo *stato dell'oggetto*, ovvero del nostro tetramino. Come metodi, invece, possiamo considerare la *caduta*, il *movimento laterale* e l'*operazione di rotazione*. Il primo metodo, ovvero quello relativo alla caduta, sarà un metodo privato, ovvero inaccessibile all'utente: questo perché l'operazione di caduta del tetramino non avviene per mano del giocatore, ma nella logica di gioco esso cadrà a prescindere. Per quanto concerne le operazioni di movimento laterale e di rotazione, tali metodi possono essere definiti come pubblici, dal momento che saranno utilizzati direttamente dal giocatore.


**Figura 1.** Classe Tetramino in UML

Una potenziale specifica della classe Tetramino in C++ è la seguente:

```
class Tetramino {
        public:
        ...
        void ruota(...);
        void mov_laterale(...);
        ...

        private:
        void caduta();
        Color colore;
        Shape forma;
        int rotazione;
        ...
```

```
}
```

Il ciclo di vita di ogni oggetto prevede una fase di *costruzione* ed una di *distruzione*. Le funzioni membro (ovvero i metodi) che implementano queste fasi prendono il nome, rispettivamente, di *costruttori* e *distruttori*. Una classe può essere costituita da uno o più costruttori, compatibilmente a come si vogliono inizializzare gli oggetti. Nel nostro esempio ha senso prevedere un costruttore che abbia in ingresso un attributo *Color* ed uno *Shape*, dal momento che si suppone che ogni tetramino generato abbia, di partenza, una rotazione pari a 0° (valore di default pari a 0). Un'ulteriore possibilità sarebbe quella di utilizzare anche un costruttore con tre argomenti che consentirebbe di generare tetramini con rotazione diversa da 0 (magari definita randomicamente, come multipli di 90°). Se volessimo costruire un oggetto a partire da un oggetto preesistente, è fondamentale prevedere un costruttore di copia nel caso in cui l'oggetto in questione fosse caratterizzato da una estensione oltre che da una parte base. Quanto al distruttore, anche in questo caso la sua definizione esplicita diventa necessaria quando l'oggetto prevede una estensione oltre alla parte base. Discuteremo in dettaglio della distruzione degli oggetti in C++ nel capitolo relativo alla gestione della memoria. Ulteriori funzioni membro che caratterizzano i metodi di una classe sono le *funzioni di accesso*. Tramite queste funzioni vengono effettuate le operazioni che comportano l'alterazione delle variabili membro di un oggetto (funzioni *set*) oppure la loro lettura (funzioni *get*).

## 1.2 Ereditarierà

Nello sviluppo di applicazioni grafiche, siano esse videogiochi o non, è ricorrente ritrovarsi in scenari costituiti da un buon numero di entità tra loro accomunate. Un esempio molto semplice è quello nel quale abbiamo un *player*, ovvero il personaggio controllato dal videogiocatore, ed una serie di personaggi controllati dal calcolatore, anche molto diversi tra loro. Possiamo tuttavia osservare che tra il giocatore e queste altre diverse categorie di personaggi esiste una struttura in comune, in termini di attributi e di operazioni. Il C++ prevede i meccanismi per l'implementazione dell'*ereditarietà*.

L’ereditarietà è uno dei concetti fondamentali della OOP. Essa si fonda sull’idea di definire delle *classi derivate* a partire da una *classe base*, ereditandone la struttura. Un oggetto di una classe derivata sarà costituito da una *parte base*, composta dagli attributi e dai metodi della classe base, ed una *parte derivata*, composta da attributi e metodi che caratterizzano la derivazione. Per esempio, nel nostro caso ha senso definire una classe base *entity* che abbia come attributi *salute* e *esperienza*, e come metodo *cammina(…)*. Da essa deriveremo due classi derivate: *player* e *enemy*. Queste *sottoclassi* ci permettono di definire una specializzazione della classe entity che rappresenta la struttura in comune alle due sottoclassi.

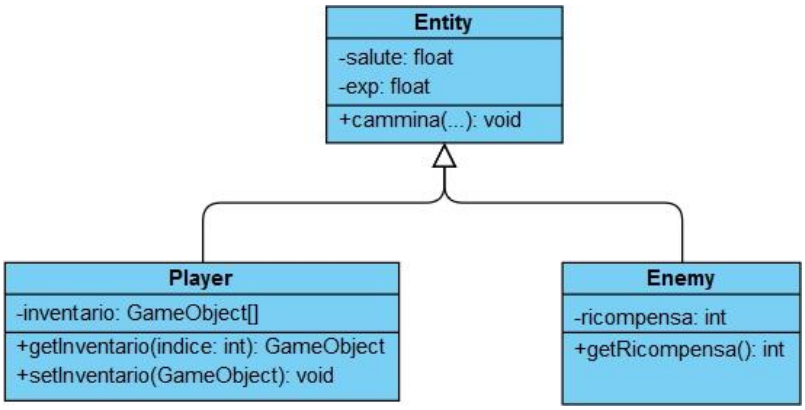


**Figura 2.** Specializzazione della classe Entity

In questo caso abbiamo previsto per il player un inventario, ovvero un array di *GameObject* e dei metodi per operare con esso, mentre per enemy abbiamo come attributo una *ricompensa*, da elargire compatibilmente con la logica di gioco. Entrambe le sottoclassi avranno gli attributi e i metodi della classe base. Questa soluzione va a favore del *riuso del codice*. Un oggetto istanza di una classe derivata sarà costituito dunque da una parte base ed una parte derivata. Queste due parti dovranno essere opportunamente inizializzate, al fine di costruire l’oggetto nella sua totalità. Un costruttore di una classe derivata dovrà prevedere nella sua lista di inizializzazione il costruttore della classe base, al fine di poter costruire ed inizializzare opportunamente la parte base dell’oggetto.

```
//costruttore classe Enemy
```

```
Enemy(float salute, int exp, float ric) :
Entity(salute, exp) {
this->ricompensa = ric;
}
```

## 1.3 Polimorfismo

Sia il nostro player che il nostro enemy dovranno essere caratterizzati da un qualche metodo che permetta loro di interagire, compatibilmente con quelle che sono le dinamiche di gioco. Supponiamo, per esempio, che il metodo che vogliamo realizzare rappresenti un'operazione di attacco, ovvero *attacca()*. Anche in questo caso possiamo parlare di un elemento in comune tra le due classi, ma è bene osservare che, a meno di non considerare il caso banale in cui entrambi abbiano la stessa operazione di attacco, per le due entità sarà previsto un attacco diverso (in termini di danno, animazioni e quant'altro). Ancora, non è detto che avremo a che fare con una singola tipologia di enemy: sfruttando ancora l'ereditarietà, potremo definire una vera e propria *gerarchia* di nemici, accomunati dalla loro struttura base (essere delle entity) e dal loro ruolo (essere degli enemy), ma molto diversi in termini di specializzazione (attacco diverso, in questo caso). Definiamo dunque le *funzioni virtuali*, così da poter introdurre il concetto di *polimorfismo*. Una funzione virtuale consente al programmatore di definire funzioni nella classe base che potranno essere ridefinite nelle classi derivate. La parola chiave utilizzata sarà *virtual*, per esempio nel nostro caso avremo *virtual void attacca()* nella classe entity, funzione che verrà opportunamente re-implementata nelle classi derivate (rispettando l'interfaccia imposta nella classe base). Tuttavia, se non è necessario, la classe derivata potrà anche non ridefinire la funzione virtuale, ma in tal caso è opportuno fare delle osservazioni. Qualora la funzione virtuale avesse una sua implementazione di default (stabilita appunto nella classe base), allora l'effettivo comportamento della stessa nella classe derivata sarà esattamente quello di default, ma se tale funzione non dovesse essere definita nella classe base, allora in questo caso ci troveremmo al cospetto di una *funzione virtuale pura* e la classe base sarebbe una *classe astratta*: la classe derivata dovrà definire la funzione virtuale pura (che rimarrà virtuale, ma non più pura) se non vorrà a sua volta essere una classe astratta. Caratteristica delle classi astratte è che non è possibile istanziare loro

oggetti, pena errore di compilazione. Nel nostro esempio, la classe entity si presta bene ad essere una classe astratta: creare un oggetto entity non avrebbe molto senso. Possiamo rivedere la nostra struttura gerarchica nel seguente modo:

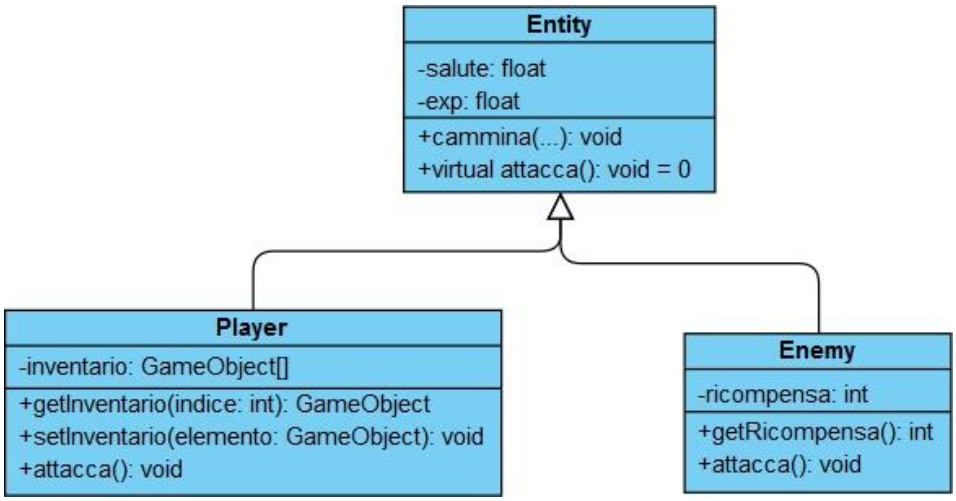


**Figura 3.** Aggiunta della funzione virtuale

dove nella superclasse entity è presente la funzione virtule pura *attacca()*. Consideriamo il caso in cui la classe enemy non implementi tale funzione virtuale e che dunque sia a sua volta una classe astratta. Si vuole avere una situazione di questo tipo dal momento che è prevista una ulteriore specializzazione della suddetta classe. Supponiamo infatti che enemy venga specializzata con due sottoclassi: *minion* e *troll*. Non vogliamo che queste sottoclassi siano astratte e dunque la funzone virtuale pura verrà implementata.

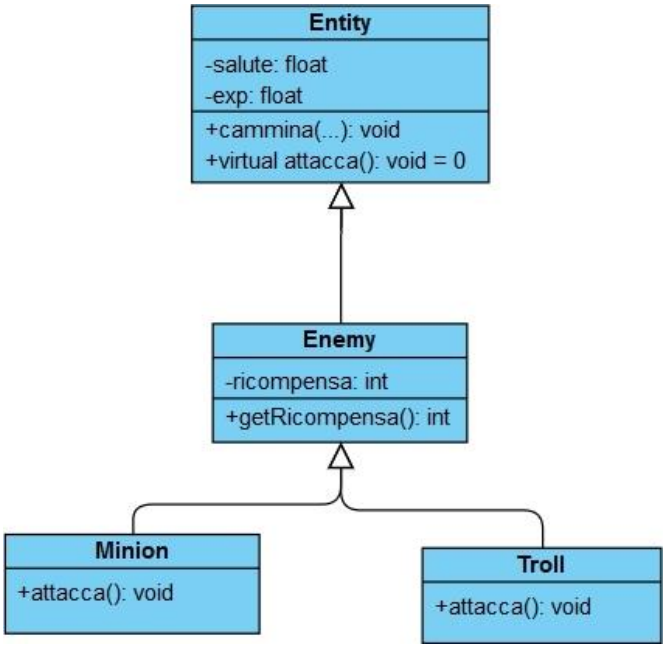


**Figura 4.** Specializzazione di enemy

Una funzione di una classe derivata con lo stesso nome e gli stessi tipi argomento di una funzione virtuale di una classe base prevale sulla funzione virtuale della classe base[2]: la funzione prevalente viene scelta come la più appropriata per l'oggetto per il quale è chiamata. Consideriamo il seguente scenario di gioco, nel quale sono per esempio presenti 4 enemy (tra troll e minion) che, finché è verificata una data condizione, uno di essi scelto a caso effettua un attacco.

```
...
Enemy* en[4];
en[0]= new Minion(…);
en[1]= new Troll(…);
en[2]= new Troll(…);
en[3]= new Minion(…);

//blocco istruzioni per la definizione della
//condizione cond

int i=0;
while(cond) {
                //assegnazione randomica di i
                //tra 0 e 3
                en[i]->attacca();
}
...
```

A seconda che *en[i]* punti ad un troll oppure ad un minion, l'attacco che verrà sferrato sarà quello relativo all'oggetto puntato. Si osservi che ciò funzionerebbe anche se il blocco di istruzioni del *while* venisse scritto prima dell'effettiva implementazione delle classi derivate (o addirittura prima che esse siano state anche solo concepite). Ciò significa che se volessi cambiare la ripartizione o la natura degli elementi puntati dai puntatori nel vettore (cosa possibile, dal momento che spesso vengono generati randomicamente) o se volessi aggiungere ulteriori specializzazioni di enemy, non dovrei effettuare nessuna modifica del blocco *while*. Questo comportamento prende il nome di *polimorfismo* e consente agli oggetti di classi diverse (appartenenti ad una stessa gerarchia, come in questo caso) di assumere un comportamento diverso in tempo di esecuzione al cospetto della medesima chiamata di funzione (in questo caso il metodo *attacca()* ), a seconda del tipo di oggetto (troll, minion o ulteriori specializzazioni non previste in questo esempio). Per ottenere questo comportamento polimorfo in C++, è necessario che i metodi chiamati siano *virtual* (come anticipato) e che gli oggetti coinvolti siano manipolati tramite puntatori o riferimenti. Il polimorfismo fornisce un notevole grado di stabilità ad un programma in evoluzione, oltre a garantire che gli ideali di modularità e di occultamento dei dati vengano rispettati. In progetti particolarmente grandi, uno strumento del genere risulta essere indispensabile. Dal punto di vista prestazionale, il meccanismo delle funzioni virtuali (ovvero il *binding dinamico*) può essere reso efficiente quasi quanto quello di una normale chiamata a funzione[2]: l'overhead in termini di memoria occupata consiste in un puntatore in ogni oggetto di una classe con funzioni virtuali (che punterà ad una *tabella delle funzioni virtuali*, anche detta *vtbl*) più una *vtbl* per ogni classe di quel genere.

# Capitolo 2: Gestione delle risorse

In questo capitolo discuteremo principalmente delle prestazioni in un videogioco, soffermandoci sull'importanza del frame rate dal punto di vista dell'esperienza di gioco, per poi trattare la gestione della memoria con il C++ e come essa può condizionare le prestazioni. Verrà inoltre fatto un breve accenno alle alternative al C++.

## 2.1 Prestazioni in un videogioco

Così come per la progettazione di un qualsiasi software, realizzare un videogioco richiede la necessità di rispettare determinate specifiche, dipendenti sia dalla macchina target, ma anche dal tipo di videogioco che si vuole sviluppare. Lo scopo è quello di definire un punto di incontro tra l'esperienza di gioco e la qualità (in termini di grafica, suono e simulazione fisica) del videogioco stesso, compatibilmente con l'hardware che si ha a disposizione. Diventa dunque necessario garantire il raggiungimento e la stabilità di determinati parametri, sfruttando nel miglior modo possibile la macchina target. In tal senso, un parametro che svolge un ruolo cruciale nel contesto dei videogiochi è la *frequenza dei fotogrammi* (alla quale spesso ci si riferisce con *frame rate*) misurata in *fps*, ovvero *frame per second*, oppure in *Hz*. Essa rappresenta la frequenza alla quale *frame* consecutivi appaiono su schermo in un secondo. Per esempio, un frame rate di 60fps garantisce che 60 frame verranno consecutivamente mostrati su schermo in un secondo. Ovviamente, anche le proprietà dello schermo giocheranno un ruolo cruciale in tal senso: un display con un *refresh rate* (ovvero la frequenza con la quale l'immagine su schermo

viene ridisegnata) di 60Hz sarà indispensabile per mostrare 60 frame in un secondo. La criticità del frame rate è legata all'obiettivo di massimizzare l'esperienza di gioco dell'utente, garantendo dinamiche fluide e piacevoli e soprattutto giocabili. Un altro aspetto di particolare interesse riguarda i videogiochi in VR (*virtual reality*) per i quali la mancanza di un frame rate sufficientemente alto e stabile può comportare l'insorgenza non solo di scarsa fluidità e di una esperienza di gioco sgradevole, ma anche della *cybersickness* la cui lista degli effetti comprende nausea, vertigine, disorientamento ed altro[3]. Sono tre i fattori che condizionano fortemente il frame rate[4]:

- Il sistema hardware di riferimento;
- Le impostazioni grafiche e della fisica di gioco;
- L'ottimizzazione del codice e le prestazioni ad esso associate.

## 2.2 Gestione della memoria in C++

Le risorse di un qualsiasi programma utente risiedono in memoria. La porzione di memoria assegnata ad ognuno di essi può essere variabile e nella fattispecie si suddivide in quattro aree:

1. *Area del codice*, contenente le istruzioni espresse in linguaggio macchina e le costanti;
2. *Area statica*, contenente le variabili statiche e quelle globali;
3. *Area heap*, contenente le varibili allocate dinamicamente;

4. *Area stack,* contenente i record di attivazione.

Le dimensioni dell'area codice e di quella statica sono fisse e sono stabilite in fase di compilazione, così come è fissa l'area complessiva tra heap e stack. Tuttavia, sia l'area heap che quella stack possono variare, a seconda della quantità dei record di attivazione contenuti nell'area stack (compatibilmente con le chiamate a funzione) e delle variabili allocate dinamicamente nell'area heap. L'allocazione dinamica degli oggetti è una tecnica indispensabile quando si vuole garantire che la durata della vita di un determinato oggetto sia dinamica. Con questa espressione si vuole intendere che un oggetto, dinamicamente allocato, potrà cessare di esistere anche prima della fine del blocco nel quale è stato istanziato. In tal senso, il C++ mette a disposizione l'operatore *new*, che permette di allocare oggetti dinamicamente, e l'operatore *delete* che permette di deallocarli. L'operazione di deallocazione risulta necessaria quando si vuole evitare la *fuga di oggetti*, ovvero oggetti non deallocati al momento giusto comportando il rischio di esaurimento della memoria destinata al programma[2]. Da ciò si evince che la gestione della memoria in C++ avviene, di base, manualmente. Infatti, al fine di realizzare un qualsiasi applicativo in C++ non è necessariamente richiesta la presenza di un garbage collector[2] (ovvero un sistema di ''riciclaggio automatico'' di regioni di memoria prive di riferimenti, operante in maniera non deterministica) che comunque rimane opzionale. L'eventuale assenza del garbage collector implica delle responsabilità aggiuntive per uno sviluppatore (di videogiochi e non). Tuttavia, per quanto questa circostanza comporti globalmente un livello di complessità implementativa maggiore, bisogna osservare che avere la possibilità di gestire manualmente la memoria può risultare un'ottima occasione per garantire un certo livello prestazionale anche nelle situazioni di gioco più esose, e ciò si traduce in un frame rate sufficientemente alto e stabile. Non va infatti dimenticato che la garbage collection comporta una ricerca in memoria di tutti quegli oggetti non aventi più un riferimento o comunque marcati per la distruzione, e l'innesco delle necessarie azioni per il rilascio di memoria: questo insieme di operazioni può essere particolarmente dispendioso in termini di elaborazione. Il non determinismo del garbage collector può

essere un problema in termini prestazionali dal momento che il suo intervento può avvenire anche in situazioni impegnative per l'hardware (si pensi, per esempio, ad una dinamica di gioco particolarmente frenetica). Effettuare la distruzione manuale degli oggetti, ovvero invocando in maniera deterministica i loro distruttori, permette all'implementatore di definire delle ''strategie di rilascio'' aventi lo scopo di liberare memoria da quegli oggetti non più utilizzati riducendo l'impatto prestazionale il più possibile. Tuttavia, i game engine più recenti che basano la scrittura della propria logica di gioco in C++ presentano dei garbage collector proprietari a supporto dello sviluppatore[5], permettendo comunque a quest'ultimo di intervenire manualmente qualora fosse necessario (ovvero al cospetto di situazioni prestazionalmente critiche).

## 2.3 Alternative al C++

Il C++ non è l'unico linguaggio di programmazione orientato agli oggetti adottato nello sviluppo di videogiochi. Sono diversi i game engine (uno su tutti Unity3D, largamente utilizzato negli ultimi anni) che, per quanto abbiano un core sviluppato in C++, adottano come linguaggi per l'implementazione della logica di gioco alternative come C#, Java o Python. Uno svantaggio del C++ è certamente legato alla complessità implementativa, ma anche alla sua scarsa portabilità (va ricordato che il C++ è un linguaggio compilato), contrariamente ad esempio a Java che oltre a risultare più semplice (basti pensare che la gestione della memoria avviene necessariamente per mezzo di un garbage collector, riducendo le responsabilità dello sviluppatore in tal senso) garantisce un livello di portabilità decisamente più alto, mediante la JVM[6]. Tuttavia, va anche osservato che la scelta del linguaggio di programmazione dipende fortemente dal tipo di progetto che si vuole realizzare. Come visto, in C++ l'assenza della necessità di un garbage collector permette ad uno sviluppatore di garantire determinismo nella gestione della memoria e ciò può condizionare notevolmente le prestazioni.

# Capitolo 3: Supporto allo sviluppo

In questo capitolo verranno introdotti i principali elementi di supporto esterni adottati per lo sviluppo sia di videogiochi, ma anche di applicazioni grafiche più in generale.

## 3.1 Introduzione

Nei capitoli precedenti ci siamo soffermati sull'applicabilità degli strumenti offerti dal C++ per la costruzione degli elementi strutturali di un videogioco e per la gestione delle risorse (in particolare, della memoria). Come già premesso in partenza, un videogioco altro non è che un'applicazione grafica particolarmente complessa e come tale va opportunamente realizzata. Chiaramente, l'elemento nucleo di un'applicazione grafica è certamente il comparto grafico in sé (inteso come l'insieme di quegli elementi grafici che vestiranno la nostra applicazione) e come esso è connesso con la logica di business. In particolare, una piacevole interfaccia grafica di un'applicazione conferisce alla suddetta un connotato maggiormente *user-friendly*, coerentemente con il principio di usabilità dell'ingegneria del software. Le librerie grafiche mettono a disposizione dello sviluppatore una serie di strumenti che permettono la realizzazione delle *GUI (Graphical User Interface)* della propria applicazione, conferendo all'implementatore il ruolo di modellatore delle suddette. Chiaramente, la scelta di quale supporto esterno si vorrà adottare dovrà essere in armonia con il tipo di applicazione che si sta realizzando. Per esempio, nel caso dello sviluppo di videogiochi 3D, le librerie grafiche maggiormente

utilizzate sono *Direct3D* (di proprietà di Microsoft), *OpenGL* e *Vulkan*. Le ultime due, in particolare, vengono largamente utilizzate anche nel contesto della modellazione tridimensionale in ambito professionale. Altri supporti fortemente utilizzati sono *Allegro* (per lo sviluppo di videogiochi in 2D) e il *framework Qt*.

## 3.2 Event-driven

Un'applicazione costituita da un'interfaccia grafica prevede che esternamente ci saranno delle interazioni con quest'ultima, innescando un opportuno comportamento dell'applicativo. Più in dettaglio, quello che sostanzialmente avviene è una *gestione degli input* da parte dell'applicazione. Da un punto di vista concettuale, possiamo schematizzare la situazione nel seguente modo:

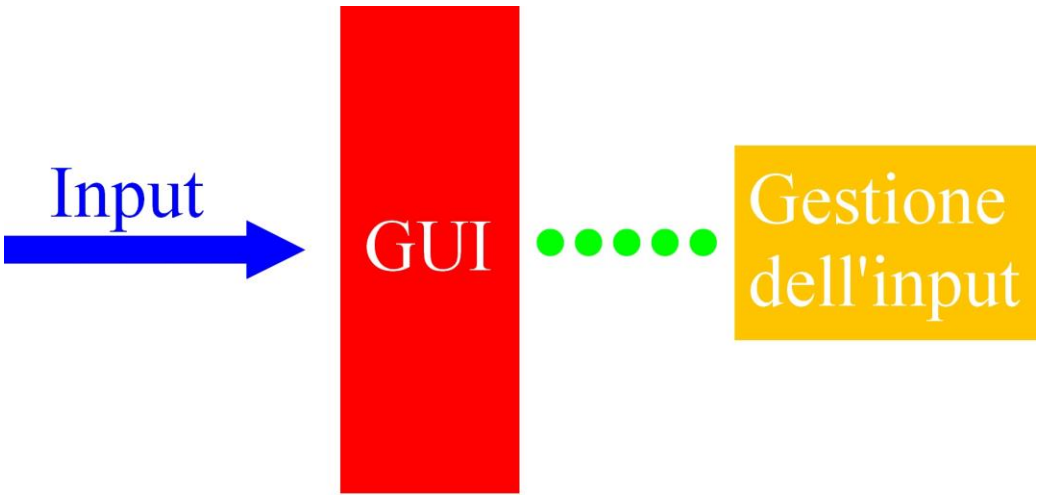


**Figura 5.** Gestione degli input

Questo approccio viene implementato con la *programmazione a eventi*, ovvero un paradigma di programmazione che permette la scrittura di programmi il cui comportamento viene determinato dagli eventi esterni (come, per l'appunto, degli input). Le applicazioni grafiche realizzate seguendo questo paradigma vengono anche dette *programmi event-driven* e sono costituite da elementi che generano questi eventi ed altri

che li gestiscono (*event handlers*). Con la struttura appena definita possiamo osservare le analogie con il *pattern Observer*, che introduce i concetti di *osservato* e di *osservatori*. Un determinato elemento chiamato *soggetto* è costituito da uno *stato* che può variare nel tempo. Al cospetto di tali variazioni, il soggetto notifica l'evento (*notify()*) a degli elementi chiamati *osservatori*, opportunamente registrati al soggetto mediante il metodo *attach(...)* che è dunque da essi *osservato*. Possiamo quindi concludere che il soggetto rappresenterà il generatore degli eventi, mentre gli osservatori i gestori.

## 3.3 Widgets

Ogni GUI è costituita da una serie di componenti grafici, detti *widgets* (*window gadgets*). La loro introduzione nasce con l'intento di definire degli oggetti che permettano non solo di costruire effettivamente una interfaccia grafica che abbia un suo stile, ma soprattutto per mettere a disposizione degli sviluppatori degli elementi che potranno essere riusati in altri progetti. Per quanto siano effettivamente componenti elementari, la loro opportuna composizione può distinguere un'interfaccia grafica amichevole da una incomprensibile: non è un caso che, nei contesti nei quali è possibile, l'interfaccia grafica di un'applicazione venga in parte concordata con l'utente finale. Infatti, per quanto un programma possa essere il più efficace ed efficiente possibile, una scarsa usabilità può impedire all'utente finale di sfruttarne a pieno le caratteristiche. I widgets sono messi a disposizione dalle librerie grafiche, ed è possibile distinguerli in diverse categorie: *widgets di comando* (bottoni, menù, dock), *widgets di dialogo* (finestra di dialogo), *widgets di input* (list box, combo box, test box), *widgets di navigazione* (barra degli indirizzi, barra di scorrimento), *widgets di output* (barra di stato, progress bar).

# Capitolo 4: Il framework Qt

In questo capitolo verrà trattato il framework Qt, brevemente introdotto nel capitolo precedente.

## 4.1 Introduzione

Qt è un framework largamente utilizzato per lo sviluppo di applicazioni con interfaccia grafica mediante l'uso dei widgets. Si tratta di un supporto *multipiattaforma* e presenta la possibilità di sviluppare applicativi anche su mobile e sistemi embedded. Qt si basa su C++, offrendo delle estensioni sintattiche del linguaggio che verranno trattate più avanti. Il suo modulo principale, detto *QtCore*, presenta svariate strutture dati, algoritmi di ordinamento e un buon numero di strumenti che sfruttano le librerie standard del C++. Sono inoltre presenti moduli per la gestione del file system, riproduzione e cattura di contenuti multimediali, unit testing e per le connessioni di rete. Presenta inoltre una ricca documentazione che ha certamente contribuito al successo del framework. Un aspetto non banale è la licenza d'uso del framework distribuito dalla *Qt Company*. Esiste una licenza commerciale che garantisce il supporto tecnico per gli sviluppatori ed una LGPL che presenta alcune restrizioni, ma che si presta bene anche ad usi commerciali. Una delle limitazioni di quest'ultima è legata all'impossibilità di sviluppare applicazioni per sistemi embedded (per esse sarà necessaria la licenza commerciale), mentre per le restanti piattaforme è obbligatorio che le librerie Qt vengano distribuite separatamente dal resto

dell'applicazione che dunque dovrà caricarle dinamicamente. Tale limitazione non risulta problematica il più delle volte. Oltre agli strumenti delle librerie, Qt offre anche dei veri e propri ambienti di sviluppo. *Qt Creator* è un IDE multipiattaforma che sfrutta gli strumenti Qt e che presenta un insieme completo di funzionalità che vanno dalla gestione dei progetti al debugging[7]. *Qt Designer* è invece un'applicazione sempre offerta dal framework, destinata alla modellazione delle interfacce grafiche tramite Qt Widgets[8].

## 4.2 Classe QObject

L'elemento fulcro dell'architettura di Qt è la classe *QObject* che ingloba le funzionalità basilari per la gestione degli eventi. Infatti, come era facile intuire, la programmazione con Qt segue il paradigma event-driven. Coerentemente con gli strumenti offerti (e seguendo quanto precedentemente detto sull'utilità dell'ereditarietà) tutte le classi che sfrutteranno gli strumenti Qt saranno derivate dalla classe QObject. Una classe MyObject derivata dalla classe QObject si presenta nel seguente modo :

```
#include <QObject>
class MyObject : public QObject
{
        Q_OBJECT
public:
        MyObject(QObject *parent = nullptr);
        ~MyObject();
protected:
private:
};
```

La macro *Q_OBJECT* deve essere sempre presente in fase di derivazione dalla classe QObject ed in particolare nella sezione *private* (non specificare la sezione, come in questo caso, significa renderla automaticamente private)[9]. Essa viene espansa in fase di precompilazione e contiene quel codice sorgente necessario alla definizione delle strutture dati fondamentali per l'utilizzo dei servizi forniti dal *meta-object system* di Qt.

## 4.3 Signals & slots

Il meta-object system di Qt fornisce, tra le altre cose, i meccanismi per la gestione dei *segnali* e degli *slot*[10]. Tramite questo meccanismo è infatti possibile realizzare il paradigma event-driven, già illustrato nel capitolo precedente. Viene infatti strutturato il pattern Observer, dove l'oggetto osservato *emette* un segnale per notificare il suo eventuale cambiamento di stato, notificando dunque la presenza di un evento da gestire. Tale segnale verrà associato ad uno *slot* di uno o più oggetti osservatore. In Qt un segnale è un particolare costrutto sintattico, molto simile all'invocazione di un metodo. Nella definizione di una classe MyObject, oltre alle sezioni *public*, *private* e *protected*, troviamo anche una sezione propria di Qt, ovvero *signals*[11]. In tale sezione vengono indicati quei metodi che rappresentano i segnali che la classe può emettere. Una classe MyEmitter può essere definita nel seguente modo:

```
#ifndef MYEMITTER_H
#define MYEMITTER_H
#include <QObject>
// La classe MyEmitter emette un segnale
// quando il suo stato cambia.
class MyEmitter : public QObject
{
        Q_OBJECT
public:
        MyEmitter(QObject *parent = nullptr)
                : QObject(parent)
        {
                something = false;
        }
        ~MyEmitter() {}
        void changeSomething(bool val)
        {
                if (something != val)
                {
                        something = val;
                        emit somethingChanged(something);
                }
        }
/* elenco di segnali */
signals:
        void somethingChanged(bool);
protected:
        bool something;
};
#endif // MYEMITTER_H
```

Quando viene invocato il metodo *changeSomething(bool val)* e per l'appunto viene cambiato il valore dello stato (rappresentato dal booleano *something*), il metodo provvede ad *emettere* il segnale mediante la parola chiave *emit*, anch'essa elemento proprio di Qt. Si noti che il segnale *somethingChanged(bool)* non presenta un'implementazione: esso infatti non va propriamente trattato come un normale metodo di una classe, ma appunto come un segnale da emettere, rappresentativo di un evento da gestire. Ovviamente, così come abbiamo definito un oggetto osservato (il *subject* del pattern Observer), che in particolare sarà un oggetto emettitore, è necessario definire uno o più oggetti osservatore (gli *observers*). Seguendo il procedimento appena visto, anche in questo caso possiamo parlare di specifici metodi caratteristici che indichiamo come *slot*. In particolare, oltre alla sezione *signals* Qt fornisce un'altra sezione propria, ovvero *slots*[11]. Rispetto ai signals, gli slots sono dei veri e propri metodi muniti di una implementazione ed è inoltre necessario specificare il livello di accesso. Una classe MyHandler può essere definita nel seguente modo:

```
#ifndef MYHANDLER_H
#define MYHANDLER_H
#include <QObject>
#include <QDebug>
// La classe MyHandler ha uno slot
// con la stessa firma del segnale emesso dalla
// classe MyEmitter
class MyHandler: public QObject
{
        Q_OBJECT
public:
        MyHandler(QObject *parent = nullptr)
        : QObject(parent) {}

        ~MyHandler() {}
/* elenco di slot pubblici */
public slots:
        void onSomethingChanged(bool val)
        {
                qDebug() << "value changed to: " << val;
        }
};
#endif // MYHANDLER_H
```

Uno slot è invocabile come un normale metodo di una classe, ma il loro uso effettivo prevede che vengano invocati al verificarsi di un determinato evento, in coerenza con il

pattern Observer. Come è ovvio che sia, non è sufficiente istanziare degli oggetti emettitori e degli oggetti gestori se non viene effettivamente garantita una connessione tra loro. Tale connessione avviene mediante il metodo statico *connect(...)*, come di seguito illustrato:

```
#include <QCoreApplication>
#include "myemitter.h"
#include "myhandler.h"
int main(int argc, char *argv[])
{
        QCoreApplication a(argc, argv);
        MyEmitter  emitter;
        MyHandler receiver;
        // Associazione tra segnale e slot
        connect(&emitter,  &MyEmitter::somethingChanged,
                                    &receiver,
              &MyHandler::onSomethingChanged);
        emitter.changeSomething(true);
        return a.exec();
}
```

dove gli argomenti del metodo rappresentano un riferimento all'emettitore (l'oggetto osservato), un riferimento al segnale, un riferimento al ricevitore o gestore (l'oggetto osservatore) e un riferimento allo slot[11]. E' fondamentale che segnali e slot abbiano firme compatibili, relativamente alla lista degli argomenti. Tuttavia, la lista degli argomenti di uno slot può eventualmente essere più breve rispetto a quella del segnale: gli argomenti in eccesso verranno scartati. Infatti, gli argomenti del segnale verranno trasmessi allo slot che li gestirà opportunamente. Un dettaglio implementativo non da poco, che va a favore della leggibilità del codice, riguarda la scelta dei nomi dei segnali e degli slot. Per convenzione, si preferisce infatti che i segnali vengano espressi con un verbo al participio passato (in questo caso *somethingChanged(...)*) e che gli slot siano accompagnati dalla preposizione *on* seguita dal nome del segnale (in questo caso *onSomethingChanged(...)*).

# Capitolo 5: Carta, sasso, forbici, lizard, Spock

In questo capitolo verrà realizzato con Qt un piccolo videogioco, ovvero *Carta, sasso, forbici, lizard, Spock*, una versione estesa del classico *Carta, sasso, forbici* introdotta nella celebre serie TV *The Big Bang Theory*.

## 5.1 Introduzione

Il gioco realizza il modello di *multiplayer locale*, ovvero due giocatori che localmente effettuano la propria scelta tra le quattro disponibili. Ogni volta che sia il player 1 che il player 2 effettuano una scelta, l'applicazione calcola l'esito stabilendo se si tratta di un pareggio oppure decretando il vincitore, secondo le note regole del gioco. Qualora dovesse esserci un vincitore, l'applicazione incrementa un contatore che rappresenta lo score del relativo giocatore, mentre ovviamente il contatore dell'altro giocatore rimarrà invariato. In caso di pareggio, nessuno dei due contatori verrà incrementato. I giocatori hanno la possibilità di giocare ulteriori turni tramite il bottone *Prossimo turno*, oppure di azzerare gli score e ricominciare la partita, semplicemente cliccando sul bottone *rivincita*.

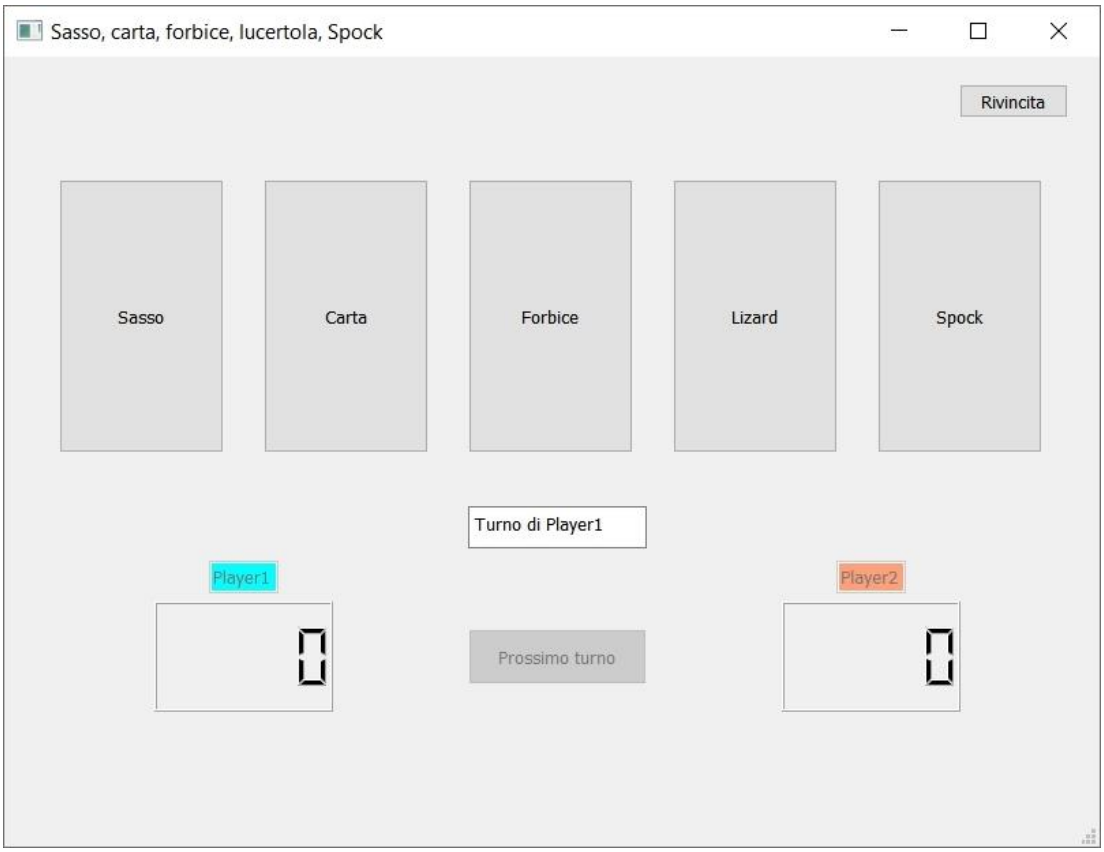


**Figura 7.** Schermata di inizio partita

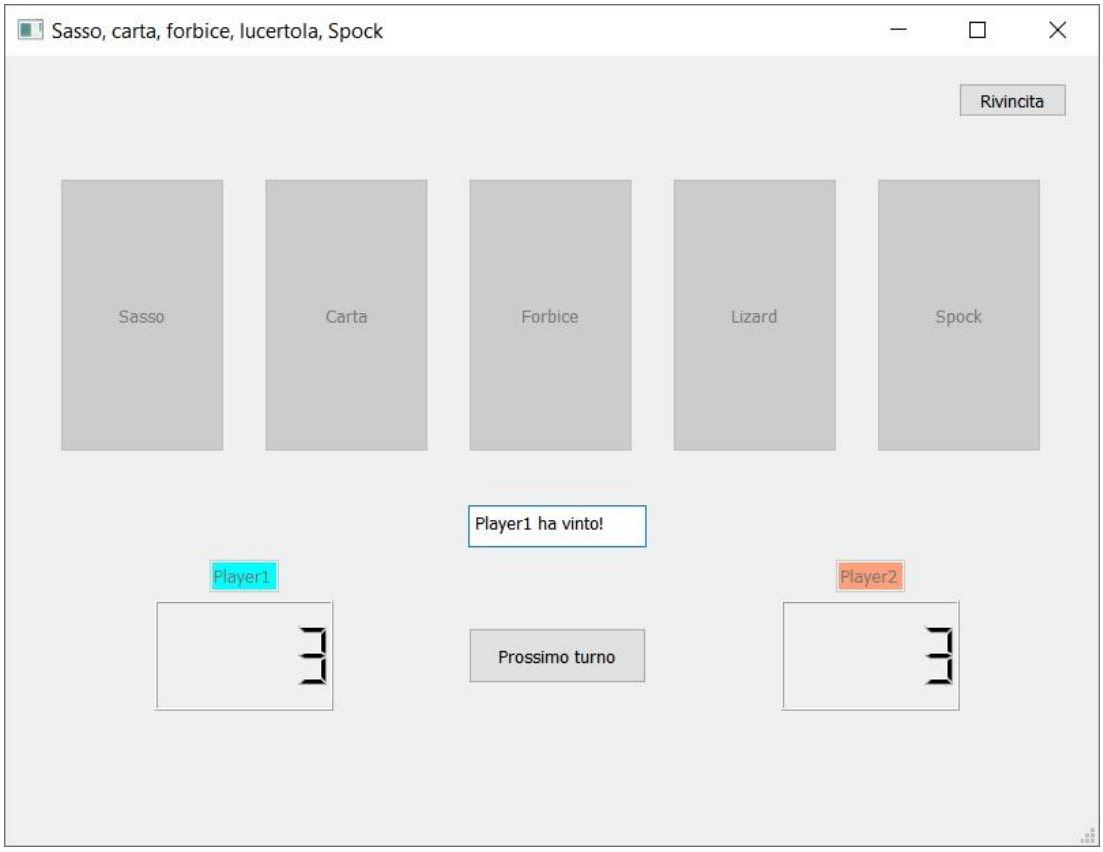


**Figura 8.** Partita in corso

## 5.2 Struttura dell'applicazione

La tipologia di progetto adottata è stata *Qt Widgets Application*, ovvero un'applicazione grafica avente interfaccia costruita mediante widgets. La classe *QGuiApplication*, relativa alle applicazioni munite di interfaccia grafica, viene specializzata dalla sottoclasse *QApplication*, relativa ad applicazioni con interfaccia grafica widget-based[12]. L'istanza di tale classe garantisce la corretta inizializzazione delle impostazioni relative alle widgets e per questo motivo deve essere inizializzata prima di istanziare qualsiasi widget. Il *main* dell'applicazione si presenta nel seguente modo:

```
/*** main.cpp ***/

#include "mainwindow.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
    return a.exec();
}
```

Successivamente l'istanza di QApplication, viene istanziato un oggetto di tipo *MainWindow* che rappresenta la finestra principale che verrà mostrata a schermo con l'invocazione del metodo *show()*[13]. Invocando il metodo *exec()* dell'oggetto *a*, si accede all'*event loop* principale dell'applicazione che termina con la terminazione del programma[12]. Nella specifica della classe MainWindow sono presenti tutti gli attributi e i metodi del caso, inclusi gli slot necessari per l'opportuna gestione dei segnali emessi dai widget dell'interfaccia grafica.

```
/*** mainwindow.h ***/

#ifndef MAINWINDOW_H
#define MAINWINDOW_H
```

```
#include <QMainWindow>

QT_BEGIN_NAMESPACE
namespace Ui { class MainWindow; }
QT_END_NAMESPACE

class MainWindow : public QMainWindow //ereditarietà
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

private slots:
    void onSassoClicked();
    void onCartaClicked();
    void onForbiceClicked();
    void onLizardClicked();
    void onSpockClicked();
    void onProssimoClicked();
    void onRivincitaClicked();

private:
    Ui::MainWindow *ui; //estensione della classe
    int mossaPlayer1;
    int mossaPlayer2;
    int scorePlayer1;
    int scorePlayer2;

    //funzioni di utilità locale
    void calcolaVincitore();
    void vincePlayer1();
    void vincePlayer2();
    void fermaGioco();
    void riprendiGioco();
    void reset();
};
#endif // MAINWINDOW_H
```

Per la costruzione dell'interfaccia grafica a partire dai widget si è utilizzato Qt Designer che oltre a realizzare la struttura della GUI, permette anche di identificare graficamente le connessioni signal-slot.

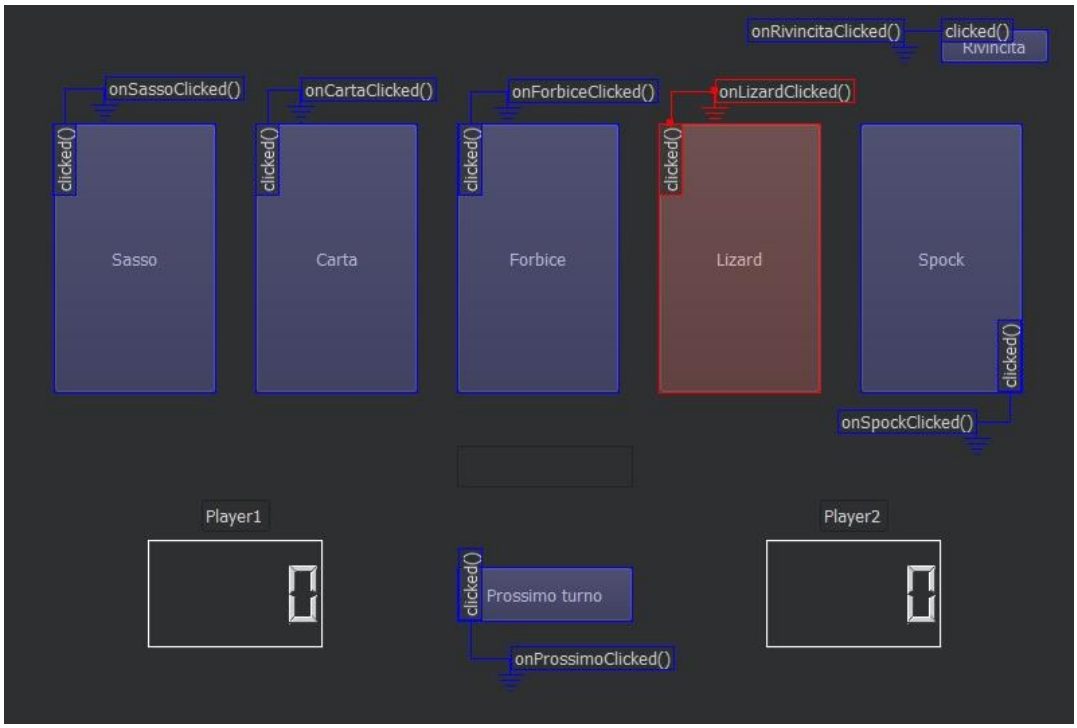


**Figura 9.** Screenshot da Qt Designer. I bottoni coinvolti emettono un segnale *clicked()* quando premuti: tale segnale verrà gestito dalla MainWindow.

Si riporta, per completezza, il .cpp della classe MainWindow.

```
/***mainwindow.cpp***/


#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
    , ui(new Ui::MainWindow)
{
    mossaPlayer1 = 0;
    mossaPlayer2 = 0;
    scorePlayer1 = 0;
    scorePlayer2 = 0;

    ui->setupUi(this);
    ui->Esito->setReadOnly(true);
    ui->Esito->setPlainText(tr("Turno di Player1"));
    ui->TextPlayer1->setDisabled(true);
    ui->TextPlayer2->setDisabled(true);

    ui->Player1->display(0);
    ui->Player2->display(0);

    ui->ProssimoTurno->setDisabled(true);
}
```

```
MainWindow::~MainWindow()
{
    delete ui; //coerentemente con l'estensione ui
}

void MainWindow::vincePlayer1()
{
    scorePlayer1 = scorePlayer1+1;
    ui->Esito->setPlainText(tr("Player1 ha vinto!"));
    ui->Player1->display(scorePlayer1);
}

void MainWindow::vincePlayer2()
{
    scorePlayer2 = scorePlayer2+1;
    ui->Esito->setPlainText(tr("Player2 ha vinto!"));
    ui->Player2->display(scorePlayer2);
}

void MainWindow::calcolaVincitore()
{

    switch (mossaPlayer1)
    {
        case 1:
        if (mossaPlayer2==1)
        {
            ui->Esito->setPlainText(tr("Pareggio!"));
            fermaGioco();
        }
        else if (mossaPlayer2==2 || mossaPlayer2==5)
        {
            vincePlayer1();
            fermaGioco();
        }
        else
        {
            vincePlayer2();
            fermaGioco();
        }
        break;

        case 2:
        if (mossaPlayer2==2)
        {
            ui->Esito->setPlainText(tr("Pareggio!"));
            fermaGioco();
        }
        else if (mossaPlayer2==3 || mossaPlayer2==4)
        {
            vincePlayer1();
            fermaGioco();
        }
        else
        {
            vincePlayer2();
            fermaGioco();
        }
        break;

        case 3:
```

```
        if (mossaPlayer2==3)
        {
            ui->Esito->setPlainText(tr("Pareggio!"));
            fermaGioco();
        }
        else if (mossaPlayer2==1 || mossaPlayer2==4)
        {
            vincePlayer1();
            fermaGioco();
        }
        else
        {
            vincePlayer2();
            fermaGioco();
        }
        break;

        case 4:
        if (mossaPlayer2==4)
        {
            ui->Esito->setPlainText(tr("Pareggio!"));
            fermaGioco();
        }
        else if (mossaPlayer2==1 || mossaPlayer2==5)
        {
            vincePlayer1();
            fermaGioco();
        }
        else
        {
            vincePlayer2();
            fermaGioco();
        }
        break;

        case 5:
        if (mossaPlayer2==5)
        {
            ui->Esito->setPlainText(tr("Pareggio!"));
            fermaGioco();
        }
        else if (mossaPlayer2==2 || mossaPlayer2==3)
        {
            vincePlayer1();
            fermaGioco();
        }
        else
        {
            vincePlayer2();
            fermaGioco();
        }
        break;

    }

}

void MainWindow::fermaGioco()
{
    ui->Carta->setDisabled(true);
    ui->Sasso->setDisabled(true);
    ui->Forbice->setDisabled(true);
```

```
    ui->Lizard->setDisabled(true);
    ui->Spock->setDisabled(true);

    ui->ProssimoTurno->setDisabled(false);
}

void MainWindow::riprendiGioco()
{
     ui->Carta->setDisabled(false);
     ui->Sasso->setDisabled(false);
     ui->Forbice->setDisabled(false);
     ui->Lizard->setDisabled(false);
     ui->Spock->setDisabled(false);

     ui->ProssimoTurno->setDisabled(true);
}

void MainWindow::reset()
{
    mossaPlayer1 = 0;
    mossaPlayer2 = 0;
    scorePlayer1 = 0;
    scorePlayer2 = 0;

    ui->Esito->setPlainText(tr("Turno di Player1"));
    ui->Player1->display(0);
    ui->Player2->display(0);

    riprendiGioco();
}

void MainWindow:: onCartaClicked()
{
    if (mossaPlayer1 == 0)
    {
        mossaPlayer1 = 1;
        ui->Esito->setPlainText(tr("Turno di Player2"));
    }

    else
    {
        mossaPlayer2 = 1;
        calcolaVincitore();
    }
}

void MainWindow:: onSassoClicked()
{
    if (mossaPlayer1 == 0)
    {
        mossaPlayer1 = 2;
        ui->Esito->setPlainText(tr("Turno di Player2"));
    }

    else
    {
        mossaPlayer2 = 2;
        calcolaVincitore();
    }
}

void MainWindow:: onForbiceClicked()
```

```
{
    if (mossaPlayer1 == 0)
    {
        mossaPlayer1 = 3;
        ui->Esito->setPlainText(tr("Turno di Player2"));
    }

    else
    {
        mossaPlayer2 = 3;
        calcolaVincitore();
    }
}

void MainWindow:: onLizardClicked()
{
    if (mossaPlayer1 == 0)
    {
        mossaPlayer1 = 4;
        ui->Esito->setPlainText(tr("Turno di Player2"));
    }

    else
    {
        mossaPlayer2 = 4;
        calcolaVincitore();
    }
}

void MainWindow:: onSpockClicked()
{
    if (mossaPlayer1 == 0)
    {
        mossaPlayer1 = 5;
        ui->Esito->setPlainText(tr("Turno di Player2"));
    }

    else
    {
        mossaPlayer2 = 5;
        calcolaVincitore();
    }
}

void MainWindow:: onProssimoClicked()
{
    mossaPlayer1 = 0;
    mossaPlayer2 = 0;
    ui->Esito->setPlainText(tr("Turno di Player1"));

    riprendiGioco();
}

void MainWindow:: onRivincitaClicked()
{
    reset();
}
```

# Conclusioni

Oltre ad evidenziare le peculiarità sostanziali del C++ e la sua enorme applicabilità, con il presente elaborato si è voluto mettere in evidenza un risultato notevole e di interesse per qualsiasi sviluppatore: la semplicità realizzativa. La combinazione di C++ e dei supporti esterni, sommati all'enorme documentazione e alle esperienze di altri sviluppatori reperibili facilmente in rete, permette di strutturare progetti più o meno complessi con relativa semplicità, riuscendo inoltre a garantire buone prestazioni. La scelta di Qt è stata fatta non solo per la sua grande applicabilità, ma anche per gli strumenti messi a disposizione dal framework, dall'IDE di base all'applicazione per costruire graficamente le GUI. Bisogna tuttavia ten conto che esistono molti altri supporti esterni e anche molto diversi tra loro, compatibilmente con i diversi contesti di applicazione. Nel nostro caso abbiamo realizzato un piccolo videogioco limitandoci all'uso dei widgets di Qt, ma se lo scopo fosse stato ad esempio quello di sviluppare un videogioco in 3D, allora in quel caso avremmo dovuto necessariamente introdurre altri supporti. Vale la pena osservare che comunque Qt presenta la possibilità di sfruttare gli strumenti OpenGL, permettendo lo sviluppo anche in tal senso.

# Bibliografia


[1] *Storia dei videogiochi*, da wikipedia.org/wiki/Storia_dei_videogiochi

[2] Bjarne Stroustrup, *C++. Linguaggio, libreria standard, principi di programmazione*, Pearson, quarta edizione (5 febbraio 2015)

[3] LaViolaJr., J. (2000). *A Discussion of Cybersickness in Virtual Environments*

[4] *Understanding and optimizing videogame frame rates*, da lifewire.com/optimizing-video-game-frame-rates-811784

[5] *Unreal Object Handling*, da docs.unrealengine.com

[6] Pellegrino Principe, *Java 11: Guida allo sviluppo in ambienti Windows, macOS e GNU/Linux*, Apogeo, (22 novembre 2018)

[7] *Qt Creator Manual*, da doc.qt.io/qtcreator/index.html

[8] *Qt Designer Manual*, da doc.qt.io/qt-5/qtdesigner-manual.html

[9] *QObject Class*, da doc.qt.io/qt-5/qobject.html

[10] *The Meta-Object System*, da doc.qt.io/qt-5/metaobjects.html

[11] *Signals & Slots*, da doc.qt.io/qt-5/signalsandslots.html

[12] *QApplication Class*, da doc.qt.io/qt-5/qapplication.html

[13] *QWidget Class*, da doc.qt.io/qt-5/qwidget.html