Simple Image Manipulation Project

PROGETTO DI PROGRAMMAZIONE AD OGGETTI
Samuele Vanini –1193535

A.A. 2019/2020

INTRODUZIONE

SIMP nasce come un software di grafica incentrato sulla creazione di immagini in grafica vettoriale con formato svg, questo data la necessità incontrata di produrre piccole icone per siti web nonostante l'assenza di esperienza con software di grafica più avanzati. Nominato in un primo momento SvgDraw, il progetto decide ci abbracciare completamente la libreria Qt non solo per la UI ma anche per il resto del codice, andando ad integrare il Graphic Framework di Qt all'interno della gerarchia sviluppata dal gruppo. Purtroppo, dopo la progettazione dell'applicativo ed una prima fase di sviluppo, culminante con un primo prototipo funzionante, nel quale erano presentanti feature di disegno e selezione, il progetto fu accantonato data la mole di lavoro eccessiva rispetto al tempo impiegabile nello sviluppo.

Dopo qualche considerazione sulle sorti del progetto e su un possibile cambio di direzione radicale, il gruppo decide di perseverare continuando sulla strada delle applicazioni di grafica. Viene deciso di tenere lo stesso concept ma di ridimensionare in quantità le feature del programma, andando ad ispirarsi ad altri applicativi come MSPaint. Viene deciso anche di ricominciare dalla progettazione, distaccandosi il più possibile dalla libreria Qt ma facendo tesoro delle nozioni apprese dal suo framework grafico (concetto di scena, ecc.).
Nasce così SIMP, un piccolo software per la realizzazione di immagini, presente di tutte le più comuni funzionalità che un programma di disegno dovrebbe avere: disegno, selezione, eliminazione e cambiamento dello stile fanno da base all'applicativo. Viene poi data la possibilità di modificare la dimensione dell'immagine che si vuole creare tramite un comodo dialog e di salvare il lavoro svolto sotto forma di immagine.

Il progetto è stato svolto insieme al mio compagno di corso ed amico Stefano Rizzo, matr. 1193464

MODALITÀ DI SVILUPPO

L'intero progetto si è svolto principalmente in tre fasi:

La prima fase è stata incentrata sulla progettazione, sono stati visionati i tutorati svolti da Benedetto Cosentino, si è discusso di quali feature volessimo fossero presenti all'interno del programma e di quali pattern fare uso per raggiungere tali obiettivi. Una volta completata ciò ci siamo concentrati sul design e su come volessimo di presentasse l'applicazione. Dopo aver ottenuto un'idea chiara di quale sarebbe dovuto essere il risultato ci siamo divisi le mansioni, decidendo poi come proseguire lo sviluppo. Tutto questo è stato fatto tramite videochiamate su zoom e documenti condivisi in cui poter fare brainstorming.

Il lavoro è stato spartito nel seguente modo: Stefano si è occupato principalmente dello sviluppo dell'interfaccia grafica e del testing approfondito dell'applicazione, io invece mi sono occupato di progettare e realizzare la parte riguardante il modello e controller.

La seconda fase, quella di sviluppo vero e proprio, è avvenuta in maniera differita. Grazie alla buona fase di progettazione e all'utilizzo di git come software di versioning (come repositori ci siamo appoggiati a GitHub: https://github.com/SamueleVanini/SIMP) ogni componente ha potuto lavorare in completa autonomia necessitando di confrontarsi solo in caso di merge nel master branch alla fine del ciclo sviluppo di una nuova feature. Tutto il codice è stato scritto tramite l'ide QtCreator, utilizzando come sistema operativo la macchina virtuale Ubuntu fornitaci appositamente per il progetto. Come pattern di sviluppo sono stati usati l'MVC per la struttura generale del progetto ed il Singleton per le risorse condivise di cui veniva richiesta l'unicità.

La terza ed ultima fase ha riguardato il testing e debug. Durante questo lasso di tempo l'applicativo è stato testato e ci si è adoperati alla risoluzione di bug e memory leak tramite strumenti di analisi, quali: Valgrind e GDB.

CONTEGGIO DELLE ORE

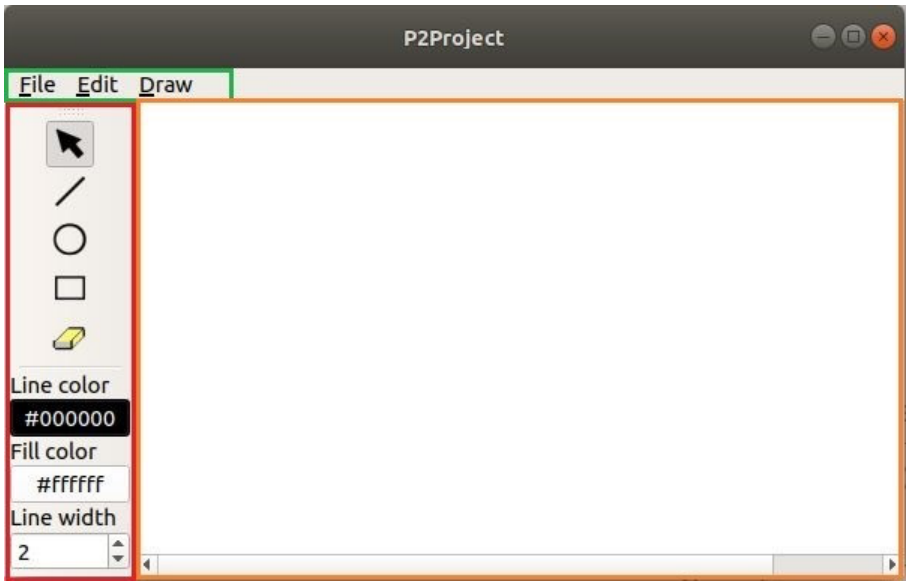
Non è stato possibile tenere un resoconto accurato delle ore impiegate a causa di due fattori rilevanti: il lavoro da remoto e per il cambio di progetto in corso d'opera. Avendo impiegato il tempo iniziale per una buona progettazione e suddivisione dei compiti, non era necessario lavorare simultaneamente, ciò ci ha permesso di sviluppare l'applicativo negli orari a noi più favorevoli rendendo il tracciamento delle ore molto complicato. Aggiungendo a questo il cambio di progetto in corso d'opera, il conteggio preciso diventerebbe impossibile.

Confrontandomi con il mio compagno abbiamo stimato un complessivo di circa 48 ore di lavoro per stupende (senza considerare le ore spese nello sviluppo della versione precedente del progetto.).

In ogni caso, posso stimare che:

- Circa 6 ore sono state impiegate per la visione dei tutorati (comuni)
- Circa 5 ore, divise in più giorni sono state dedicate alla stesura delle funzionalità e all'analisi del problema (comuni)
- Progettazione e sviluppo GUI, circa 30 ore (Stefano)
- Circa 3 ore,per la stesura di una gerarchia del progetto (personali)
- L'apprendimento dei pattern di sviluppo e della libreria Qt hanno impiegato circa 6 ore (personali)
- La progettazione e lo sviluppo del model e dei controller, invece circa 30 ore. (personali)
- Circa 4 ore impiegate in test e conseguente debug. (metà comuni e metà personali)

MANUALE UTENTE



La UI dell'applicativo è composta nel seguente modo:

- Un menu superiore (in verde)
- Una Toolbar sul lato sinistro (in rosso)
- Un canvas (in arangione)

Il menu superiore è formato, nell'ordine da tre sottomenu: File, Edit e Draw.
L'opzione File permette a sua volta tre opzioni: aprire un nuovo progetto eliminando quello attuale, salvare il lavoro svolto sotto forma di immagine o uscire dall'applicativo.
L'opzione Edit invece permette di eliminare tutte le figure presenti a schermo o di modicare la grandezza del canvas, che di default si presenta con dimensione 600x300px.

Per ultima troviamo l'opzione Draw che ci permette di selezionare uno degli strumenti della toolbar dal menu superiore.

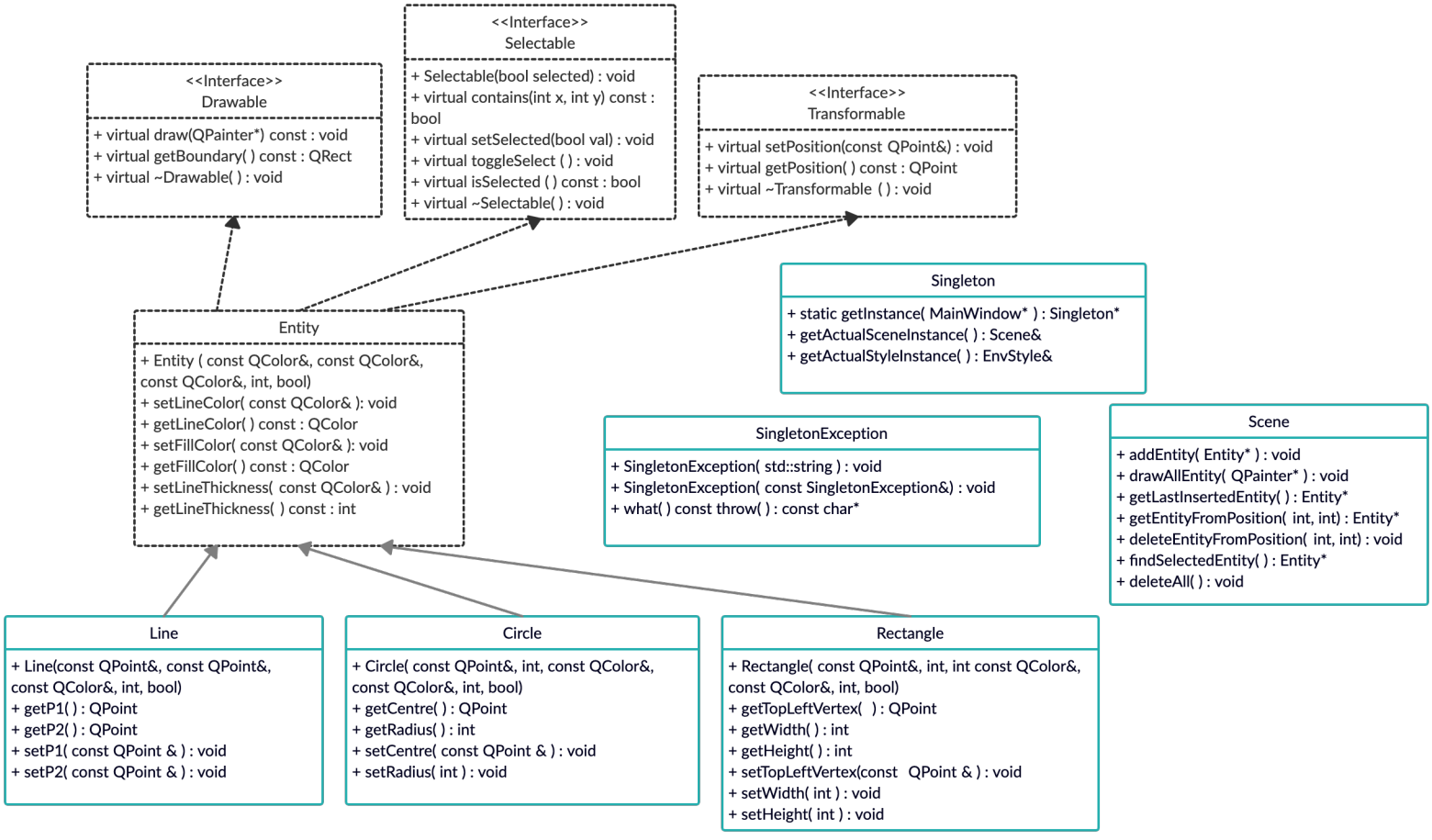
Come secondo componente troviamo la toolbar ovvero il raccoglitore di tool e stili predefinito con cui un utente può interagire con il programma. Punto di forza di questa è la caratteristica di essere mobile, infatti, nonostante sia ancorata al lato sinistro di default tramite i tre pallini situati alla sommità di essa è possibile sganciarla interamente dalla finestra principale o agganciarla ad uno dei tre lati rimanenti, il tutto tramite drag and drop.

Il terzo ed ultimo componente è il canvas. Tela su cui disegnare è la parte centrale di tutta la UI, inserito all'interno di una scrollArea, nel caso in cui lo ridimensionassimo eccedendo la grandezza della finestra questo presenterà delle barre laterali per la navigazione al suo interno.

Da sottolineare è la funzione di ridimensionamento della finestra che non andrà a ridimensionare il nostro canvas (per quello è presente una funzione apposita già spiegata) e la possibilità di cambiare lo stile di una figura dopo averla selezionata con lo strumento di selezione.

MODELLO

Per realizzare il modello ho creato una gerarchia polimorfa che andasse a classificare una qualsiasi entità visualizzabile a schermo. Più nel dettaglio troviamo tre interfacce (Drawable, Selectable, Transformable) che vanno a definire come deve essere una entità che voglia essere, nell'ordine: disegnabile a schermo, selezionabile o ricevere una qualsiasi trasformazione. Subito sotto le tre interfacce troviamo la classe Entity che serve da classe base per tutti gli oggetti che poi andremo a disegnare a schermo in quanto implementa tutte e tre le interfacce



aggiungendo delle proprietà quali: il colore della linea di contorno, lo spessore della linea di contorno ed il colore con il quale riempire l'oggetto. Troviamo poi le 3 implementazioni delle figure (Line, Circle, Rectangle) che si occupano di introdurre la logica propria delle singole figure.

IL CONTENITORE SCENE

Ci si rende conto velocemente durante la progettazione dell'applicativo che tutte le entità che si decidono di creare avranno bisogno di un contenitore dove essere salvate in modo da poter essere recuperate ad ogni repaint del canvas. Dopo un'attenta analisi si è deciso di creare una classe (Scene) che facesse da contenitore e che utilizzare un std::vector come contenitore vero e proprio, ciò è dato dalle ottime performance nell'inserimento e nella scansione lineare oltre che alla presenza di numerose funzionalità già presenti gratuitamente nella Standard Library (iteratori, funzione di compattazione della struttura dati dopo una eliminazione, ecc.). Ciò mi ha permesso di concentrarmi maggiormente su altre parti (Singleton, Controller, ecc.) presentando comunque una soluzione ottimale al fine del programma. Di seguito si trova l'interfaccia pubblica del contenitore:

- `void addEntity(Entity*)` : permette di aggiungere una entità alla scena
- `void drawAllEntity(QPainter*)` : disegna tutte le entità presenti nella scena nel QPainter che viene passato tramite una chiamata polimorfa al motodo Drawable::draw(QPainter*) posseduto da ogni entità
- `Entity* getLastInsertedEntity()` : ritorna l'ultima entità inserita nella scena, metodo accessorio che è rissultato molto utile in molti controller
- `Entity* getEntityFromPosition( int x, int y)` : ritorna il puntatore all'ultima entità inserita data una posizione in coordinate assolute, nel caso la ricerca non dia un esito positivo ritorna un nullptr
- `void deleteEntityFromPosition( int x, int y)` : data una posizione elimina l'ultima entità inserita in quella posizione
- `void deleteAll()` : elimina tutte le entità presenti

Da notare che non è presente ne un costruttore ne un distruttore, questo è perché essi fanno parte della parte privata della classe. Questa scelta è stata fatta per aderire al pattern di sviluppo del Singleton che verrà spiegato in seguito

SINGLETON

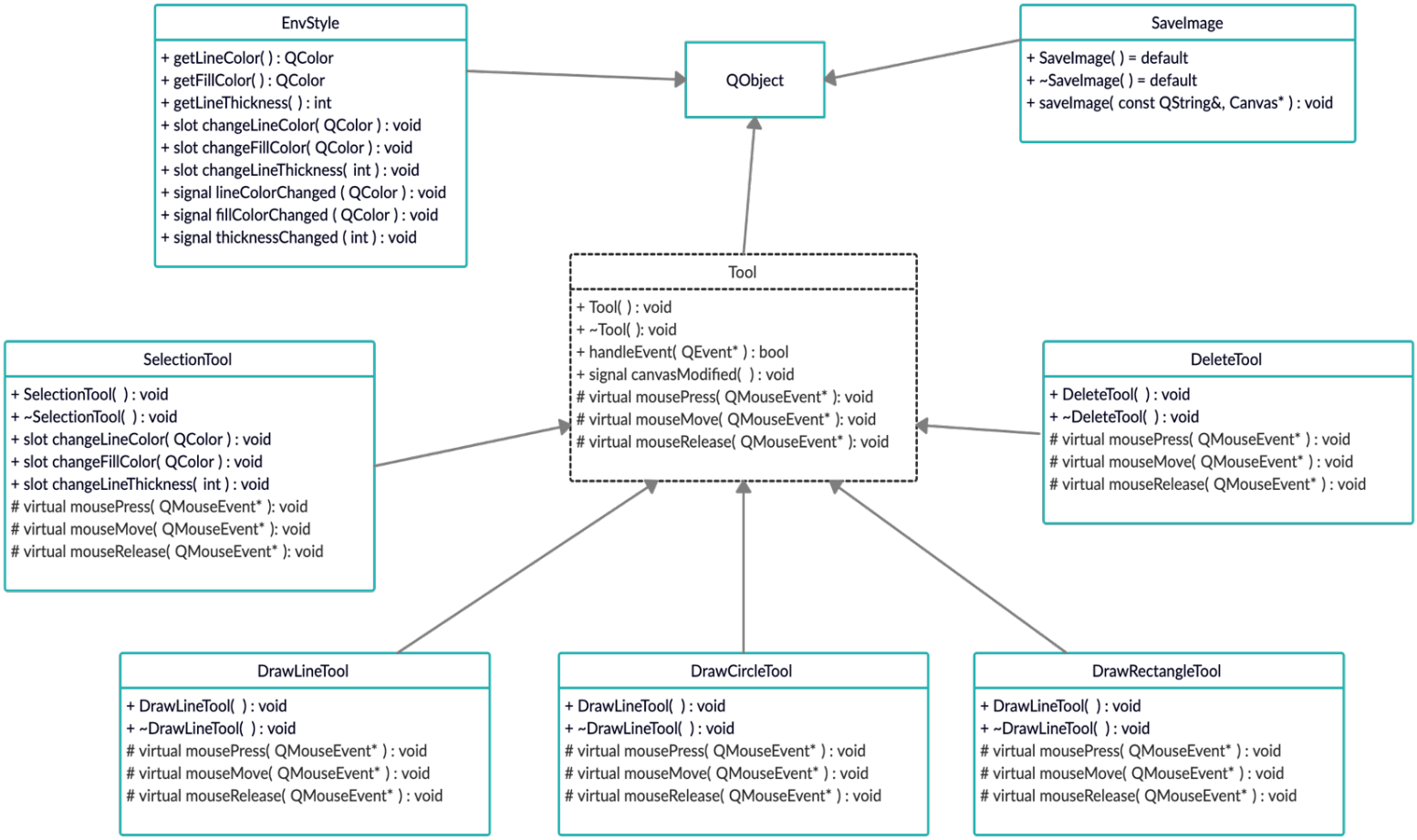
Pattern di sviluppo adottato per sopperire ad alcune problematiche riscontrate durante lo sviluppo. Nello specifico molte classi condividevano delle risorse (Scene, EnvStyle) possedendo un puntatore a queste risorse in cui avveniva della condivisione della memoria controllata. Questo però andava a creare delle problematiche durante la distruzione di questi oggetti, infatti un oggetto non era a conoscenza di quante altre istanze del suo stesso tipo o di altri oggetti stessero attualmente usando quella risorsa, rendendo molto complicata la liberazione della memoria. Delle varie soluzioni analizzate mi sono reso conto che per rendere le varie componenti software indipendenti da una logica di distruzione arbitraria e mantenendo delle buone performance (l'utilizzo massivo di shared_ptr avrebbe rischiato appesantire troppo alcune operazioni al crescere del numero di entità) era necessario la progettazione di un manager per le risorse. Ispirandomi quindi ad alcuni esempi e spiegazioni trovate sul web (https://en.wikipedia.org/wiki/Singleton_pattern), ho creato una classe che gestisse la creazione e distruzione delle risorse condivise con la seguente interfaccia pubblica:

- `static Singleton* getInstance(MainWindow* ui)` : unico metodo statico della classe, gestisce la creazione delle risorse se non ancora inizializzate e di ritornare un'istanza al Singleton stesso con cui accedere alle risorse. Il parametro MainWindow serve per la prima volta in cui si cerca di accedere alle risorse in quanto necessario per inizializzare EnvStyle
- `Scene& getActualSceneInstance()` : ritorna l'istanza corrente della scena creata
- `EnvStyle& getActualStyleInstance()` : ritorna l'istanza corrente dello stile creato
- `Singleton(const Singleton&) = delete` : sintassi introdotta dal c++11, dichiara che non può essere presente un costruttore di copia, andando ad eliminare anche quello di default (non deve essere possibile eliminare il Singleton in quanto manager unico)
- `Singleton& operator=(const Singleton&) = delete` : analogo a quanto visto per il punto precedente ma questa volta per l'assegnazione

Come già visto in precedenza il costruttore è nella parte privata della classe in quanto non deve essere possibile creare più Singleton, la logica di costruzione di questo si trova nel metodo statico getInstance(MainWindow* ui)

CONTROLLER

Per la realizzazione del controller, analogamente a quanto fatto per il model, ho creato una gerarchia di classi polimorfe ereditante da QObject in aggiunta sono presenti due classi singole inserite per rappresentare azioni svolgibili dall'applicativo.



Analogamente a quanto visto per la classe Scene anche la classe EnvStyle non possiede costruttore e distruttore nella sua interfaccia pubblica, questi metodi infatti sono presenti nella sua classe privata.

VIEW

Per la realizzazione della view sono state adoperate tre classi:

- MainWindow
- ColorButton
- Canvas

MAINWINDOW

Componente principale tra le tre classi, estende l'omonima classe di Qt, QMainWindow. Fornisce tutte le funzionalità necessarie alla finestra principale, oltre a questo possiede al suo interno i riferimenti le due classi precedentemente elencate e a tutti i tool disponibili. Per fare in modo che la view possa comunicare con lo strato del controller e con le altre classi della UI vengono forniti all'interno di MainWindow numerosi signal e slot.

COLORBUTTON

Classe ereditante dal QPushButton ricopre il ruolo di selezionatore per il colore della linea di contorno e per quello di riempimento. Per fare in modo che fosse possibile collegare ColorButton ad un'azione, nello specifico, al click del mouse è stato necessaria l'aggiunta di un campo e due metodi. Più nello specifico:

- `QAction* actionOwner` : raw pointer che tiene traccia dell'azione associata al pulsante
- `void setAction(QAction* action)` : esegue il set dell'azione associata al bottone, aggiorna le proprietà del bottone ed esegue le connect e disconnect necessarie in modo da emanare il signal clicked alla emissione del signal triggered dell'Action associata
- `void updateButtonStatusFromAction()` : si occupa di aggiornare il testo e lo stato del bottone

CANVAS

Widget principale della UI, è il principale punto di interazione fra l'utente e l'applicativo. La classe eredita da QWidget ed implementa due metodi da quest'ultima per la gestione degli eventi e del disegno del widget stesso, nello specifico troviamo:

- `bool event(QEvent* event)` : metodo che viene richiamato ogni volta in cui avviene un evento all'interno del canvas. Il suo compito è quello di richiamare l'event handler dei tool o il metodo paintEvent nel caso in cui l'evento arrivato sia del tipo QEvent::Paint. Nel caso un cui si passi la gestione dell'evento ad un tool ne aspetta il risultato per ritornarlo e chiama un update del widget in quanto qualcosa nel modello è cambiato.
- `void paintEvent(QPaintEvent* pe)` : metodo chiamato da event(QEvent*) direttamento o dall'update, è incaricato di chiedere un'istanza della scena corrente e disegnare tutto il suo contenuto sul canvas.

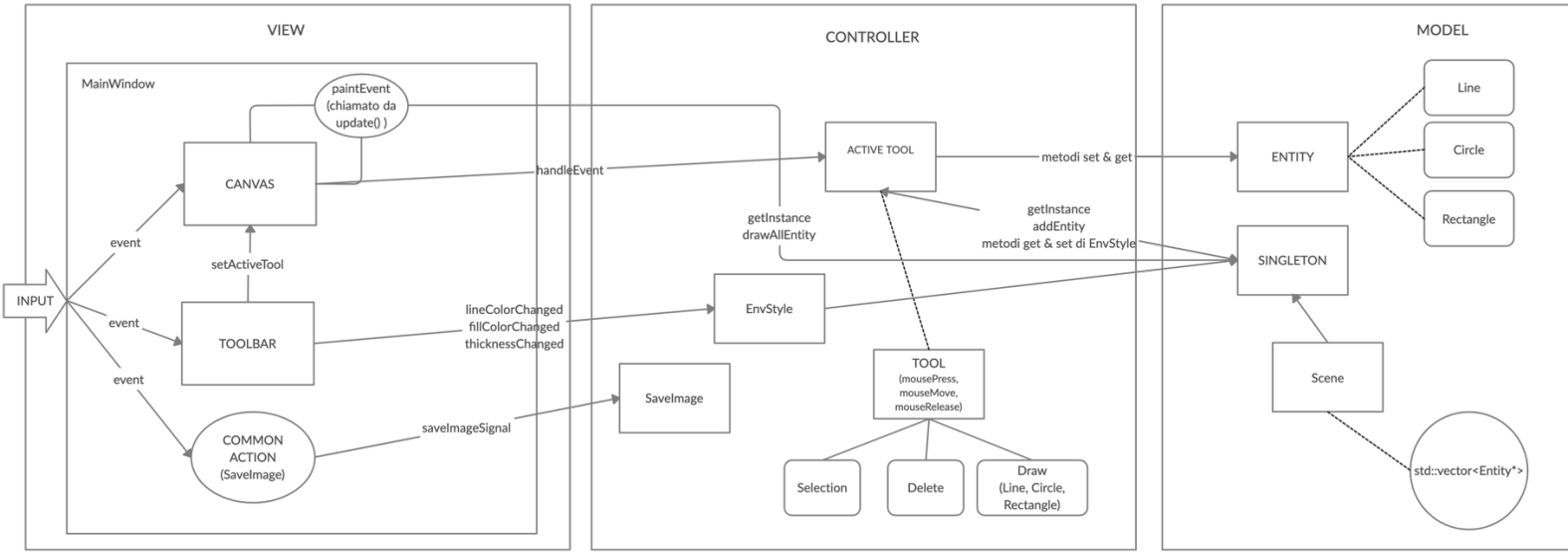
ISTRUZIONI PER LA COMPILAZIONE E L'ESECUZIONE

Con il progetto viene fornito anche il makefile necessario per la compilazione. Pertanto, per compilare ed eseguire il programma, è necessario posizionarsi all'interno della cartella del progetto e lanciare i comandi

1. qmake
2. make
3. ./P2project.sh

FUNZIONAMENTO GLOBALE

Per descrivere al meglio il ciclo di funzionamento è riportato di seguito uno schema riportante i passaggi fondamentali di cui verranno spiegate nel dettaglio le chiamate polimorfe.



CHIAMATE POLIMORFE

All'interno del progetto sono presenti numerose chiamate polimorfe concentrate nei controller e nel model, partendo dal Controller troviamo:

- `bool handleEvent(QEvent *)` : funge da piccolo event handler, una volta ricevuto un evento ne identifica il tipo ed effettua una chiamata polimorfa al metodo corretto di gestione dell'evento nel tool attualmente attivo.
- `void mousePress(QEventMouse *)` : metodo polimorfo per la gestione della pressione di un tasto del mouse
- `void mouseMove(QEventMouse *)` : metodo polimorfo per la gestione del movimento del mouse
- `void mouseRelease(QEventMouse *)` : metodo polimorfo per la gestione del rilascio di un tasto del mouse

Passando poi alla parte di Model, troviamo le seguenti chiamate polimorfe:

- `void draw(QPainter *)` : si occupa di disegnare su un painter una Entity, un suo utilizzo è nel metodo drawAllEntity() della classe Scene, in cui vengono disegnate tutte le Entity presenti all'interno di un std::vector
- `QRect getBoundary()` : ritorna un rettangolo contenente l'Entity, viene utilizzato per disegnare il rettangolo che circonda una figura quando viene selezionata e per dei calcoli nel metodo contains (int, int) della classe Line
- `void setSelected(bool state)` : setta lo stato della Entity
- `void toogleSelected()` : imposta lo stato della Entity su selezionato
- `bool isSelected()` : get dello stato di selezionato
- `void setPosition(const QPoint&)` : setta la posizione della Entity
- `QPoint getPosition()` : get della posizione di una Entity

ESTENSIBILITÀ DEL SOFTWARE

L'estendibilità del codice è garantita dalla struttura con il quale è stato sviluppato il progetto, in altre parole in base alle necessità richieste si dovranno modificare uno o più strati dell'MVC.

La modifica del model, quindi ad esempio l'aggiunta di nuove figure (es. Rombo, Tringolo, ecc.) richiederà che la nuova classe/gerarchia implementi la classe Entity senza preoccuparsi di come la figura dovrà poi essere salvata o inserita nel canvas. Ciò lascia libero lo sviluppatore di concentrarsi su come la nuova figura dovrà disegnarsi.

La modifica del controller, nel caso in cui si desideri aggiungere un nuovo strumento per l'interazione con una vecchia figura o con una nuova appena aggiunta, richiederà solo l'implementazione della classe Tool.

In fine una modifica alla UI non comprendente il canvas lascia il resto della gerarchia completamente immutato a patto che si utilizzino i signal e slot predisposti. Ad esempio, se la funzionalità di salvataggio dell'immagine volesse essere spostata in dialog a parte basterebbe scrivere il nuovo codice per la UI e collegarsi al controller già in uso, di fatto senza intaccare la parte di controller e model.
Se invece si volesse aggiungere una nuova icona alla toolbar già in uso per un nuovo tool, sarà necessario semplicemente salvare un riferimento al nuovo tool sotto forma di shared_ptr alla MainWindow e modificare in modo opportuno i metodi createTools e createLeftToolbar sempre nella classe MainWindow.