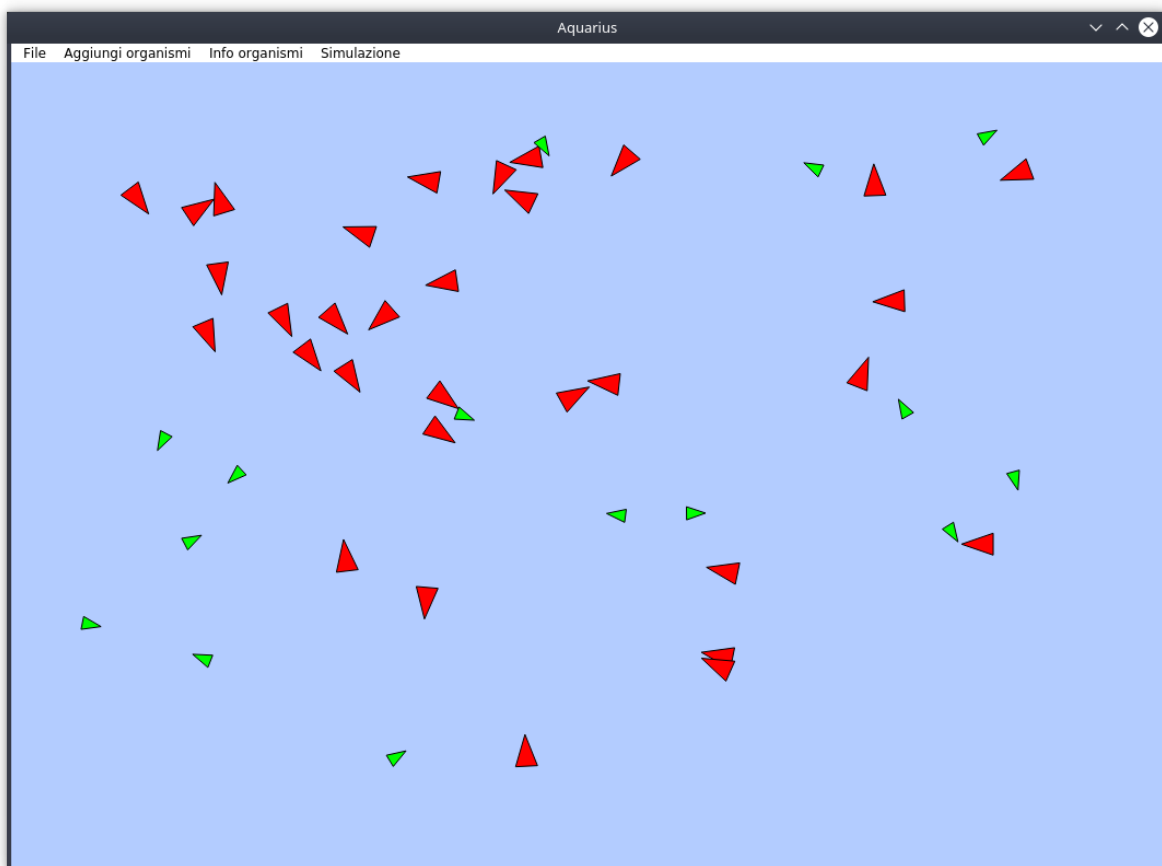


Aquarius

Ispirato da un algoritmo ideato nel 1986 da **Craig Reynolds** ideato per simulare il comportamento degli stormi, ma rivelatosi dalle infinite applicazioni.



Un Acquario simulato popolato da semplici organismi che si muovono in base ad alcune regole e un pizzico di casualità. Alcuni si muovono in branco, altri scappano dai predatori, ma tutti si devono nutrire in un modo o nell'altro. In questo sistema caotico ogni piccolo cambiamento crea una simulazione completamente unica.

Tempo Impiegato

~  Dardouri ~  Furlan

Studio per Veicolo	1h	6h
Studio per Vect2D	30m	1h
Codifica Veicolo	0	5h
Codifica Vect2D	0	2h
Progettazione Model + Container	5h	5h
Codifica Model + Container	15h	20h
Progettazione View, Controller, Stile	2h	1h
Codifica View, Controller, Stile	20h	5h
I/O	30m	3h
Debugging + Calibrazione	10h	10h
Totale	~54	~58

Studio + Codifica Veicolo

Mosso da un pregresso interesse personale, ho dedicato abbastanza tempo nello studio e codifica, distribuito tra visione di documenti e video, per capire e adattare l'algoritmo volto al movimento dei veicoli. Questo stesso algoritmo viene usato in diverse applicazioni tra quali veicoli autonomi, argomento interessante e utile per studi futuri. Questa scelta e le conseguenze che ne derivano è la causa delle ore che sfiorano dalle 50.

Studio + Codifica Vect2D

Mi sono occupato dello studio e implementazione di tutte le possibili operazioni algebriche.

Progettazione + Codifica Model

Mentre la progettazione è stata svolta insieme, la codifica s'è svolta in parallelo alternandoci tra le classi, mi son concentrato maggiormente nella parte riguardo il movimento dei veicoli, il container Vector e il DeepPtr.

Progettazione + Codifica View, Controller, Stile

Mi sono occupato della ricerca e codifica del metodo di paint dei veicoli.

I/O

Me ne sono occupato principalmente i/o.

Debugging + Calibrazione

Data la complessità dovuta dai molti parametri e operazioni in gioco, ho effettuato molteplici test, specialmente nella logica di movimento dei veicoli.

MVC

Il pattern architetturale adottato per lo sviluppo é MVC. Sono presenti due viste, le quali però svolgono funzioni totalmente differenti e per questo i relativi controller sono stati pensati separati e specializzati. Viene comunque mantenuta una certa relazione tra i controller in quanto le relative viste rimangono comunque correlate e una figlia dell'altra.

La simulazione

Tutte le informazioni per la simulazione sono contenute all'interno della classe `Aquarius`. `Aquarius` contiene la lista di tutti gli organismi nella simulazione, la dimensione del riquadro e il nome dell'acquario.

La simulazione procede tramite la chiamata ogni 20 millisecondi del metodo `advance`:

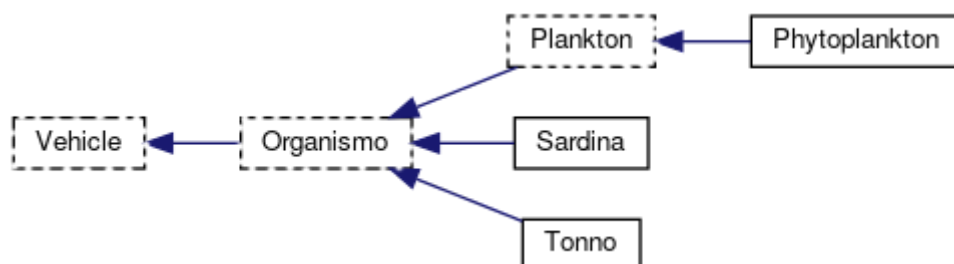
```
// Semplificazione
void Aquarius::advance() {
    // FASE 0
    for (auto& i : organismi) {
        i->advance(this, 0);
    }

    // FASE 1
    for (auto& i : organismi) {
        i->advance(this, 1);
    }
}
```

Si occupa di fare *avanzare* di uno step di animazione, chiamando quindi a sua volta il metodo `advance` di ogni veicolo.

Il metodo è suddiviso in due fasi: la fase 0 dove vengono calcolati i comportamenti di ciascun veicolo e la fase 1 dove essi vengono applicati. Questo è per permettere a **tutti** gli organismi di eseguire i loro calcoli sullo stato attuale.

Gerarchia



Veicolo: classe astratta

È la classe alla base della gerarchia, implementa clonazione e distruzione virtuale.

Veicolo è stata pensata come una classe a parte, nonostante possa benissimo essere integrata in organismo. Questa scelta deriva dalla preferenza di mantenerla generica per qualsiasi futura differente implementazione.

Un Veicolo è caratterizzato da una **posizione** nello spazio, una **velocità** ossia la sua variazione di posizione nello step di animazione successivo e una **accelerazione**, che rappresenta la variazione di velocità nello step di animazione successivo. Questi tre sono rappresentati tramite un oggetto Vect2D.

Vect2D

Rappresenta una coppia di coordinate (x,y) nel piano a due dimensioni. Viene alternativamente riferito anche come Vettore. Possiede una vasta gamma di operazioni con altri vettori e scalari. I metodi sono tutti concatenabili grazie al ritorno per riferimento (nelle versioni non const). Gli stessi sono resi disponibili anche in versione costante, statica e, dove possibile, tramite operatore. Sempre concatenabili, ma per valore.

Il Veicolo ha inoltre i parametri **maxSpeed** e **maxForce**, i quali permettono rispettivamente di limitare la velocità e l'accelerazione a un valore massimo.

Questa classe mette a disposizione una vasta gamma di funzioni, che ritornano il vettore accelerazione, per muovere il veicolo:

- **Seek:** Dato un punto, vacci a tutta velocità;
- **Arrive:** Dato un punto, arrivaci rallentando man mano ti ci avvicini, fino a fermarti;
- **Flee:** Dato un punto, se si è entro una certa distanza, scappa il più lontano possibile;
- **Pursuit:** Dato un veicolo, seguilo;
- **Escape:** Dato un veicolo, scappa;
- **Stop:** Ferma il veicolo;
- **Wander:** Genere un movimento caotico, per "girovagare";
- **StayWithinBorders:** Date le dimensioni dell'acquario, evita le pareti rimanendo all'interno.

Queste funzioni verranno utilizzate per il calcolo del **behaviour** dei veicoli.

Virtuali

```
virtual void behaviour(Acquario*) = 0;
```

Virtuale puro.

Va definito in ciascuna derivata concreta con i comportamenti specifici di ciascun tipo. Ogni derivata dovrà assicurarsi di porre alla fine del corpo della funzione la chiamata al behaviour della classe genitore.

```
virtual bool isInRange(const Vect2D& v) const = 0;
```

Virtuale puro.

Ogni Veicolo ha un range visivo, determinato in diversi modi: circolare, onniscente, a spicchio, rettangolare. In base al range che si preferisce, il metodo ritorna vero se il punto nello spazio passato per parametro è contenuto all'interno del suo range visivo, falso altrimenti.

```
virtual void advance(Aquarius* a, int phase) final;
```

Per assicurarsi in maniera assoluta che il metodo non venga ridefinito, in quanto è completo e non necessiterà mai una ridefinizione, è stato segnato come final.

Questo come spiegato in precedenza serve per, nella prima fase invocare behaviour e quindi effettuare tutti i calcoli, e nella seconda fase applicarli.

Organismo: classe astratta

Organismo è un veicolo con caratteristiche di un essere vivente quali:

- un **ValoreNutrizionale**: assegnato in base alla posizione nella catena alimentare di ciascun tipo derivato, valori alti per predatori, valori bassi per le prede;
- una **Stamina**;
- un ciclo di dormi / veglia (**DayCycle**).

Sono state utilizzate due classi esterne per integrare la logica di questi ultimi due aspetti.

DayCycle

Composto dagli step di animazione che passa da sveglio e a dormire e un contatore che tiene conto del passare del tempo. Fornisce funzioni per incrementare il contatore, anche tramite operatori. Permette di essere interrogato su in che fase del ciclo si è attualmente e sulla percentuale di avanzamento all'interno di essa.

Stamina

Composta da un valore massimo e un valore variabile. Fornisce funzionalità per incrementare e decrementare il valore, mantenendolo compreso tra 0 e il valore massimo. Permette inoltre di essere interrogato ottenendo il valore percentuale del contatore su il valore massimo.

Ciascun organismo definisce in **behaviour** degli "istinti naturali", ossia regole che prevalgono sulle decisioni prese dalle derivazioni. Per esempio, nel caso avesse fame si concentra solamente sul seguire e mangiare la preda.

Virtuali

```
virtual bool canSleep() const;
```

Determina se l'organismo può dormire. Di default è quando il contatore del daycycle segna notte.

```
virtual bool canWakeup() const;
```

Determina se l'organismo può svegliarsi. Di default è quando il contatore del daycycle segna giorno.

```
virtual bool isHungry() const = 0;
```

Virtuale Puro.

Determina se l'organismo è affamato.

```
virtual int getValoreNutrizionale() const = 0;
```

Virtuale Puro.

Determina il valore nutrizionale dell'organismo.

```
virtual std::string getSpecie() const = 0;
```

Virtuale Puro.

Determina la specie dell'organismo. Per sottospecie potrebbe essere concatenata con la chiamata al genitore.

Tonno: classe concreta

Il tonno vede fino a 100 unità di distanza con una ampiezza visiva di ± 150 gradi. Si muove in branco con gli altri organismi nel suo campo visivo, quindi rispetto a loro allinea la sua direzione, cerca di rimanere nel gruppo, ma evita comunque di scontrarsi.

Il Tonno è affamato quando la sua stamina scende sotto il 40%.

Il resto è comportamento di default dell'organismo.

Sardina: classe concreta

La sardina vede fino a 80 unità di distanza in ogni direzione. Il comportamento normale è vagare per l'acquario e, nel momento in cui entra nel suo range visivo un organismo più grande di lui, scappa.

La sardina è affamata quando la sua stamina scende sotto il 20%.

Il resto è comportamento di default dell'organismo.

Plankton: classe astratta

I plankton sono organismi semplici che vedono fino ad un massimo di 50 unità di distanza. Normalmente cercano di evitare i predatori nel proprio campo visivo come la sardina.

Il Plankton essendo astratto non ha un metodo per definire se è affamato o meno.

Il resto è comportamento di default dell'organismo.



Phytoplankton: classe concreta

I Phytoplankton sono Plankton che non si nutrono di organismi, ma fanno la fotosintesi mentre sono svegli.

I Phytoplankton non sono propriamente mai affamati di altri organismi.

Il resto è comportamento di default del Plankton.

Tabella parametri degli Organismi

Classe	maxSpeed	maxForce	valNutr.	Max. Stamina	DayCycle	
						
Tonno	5	0.15	4	40	10000	200
Sardina	4	0.13	3	30	500	100
Plankton	1	0.10		20	300	100
PhytoPlankton	1	0.10	2	20	300	100

Contenitore

È stato scelto il vector per le sue abilità di accedere in tempo costante ad ogni posizione. È stato implementato seguendo la documentazione del vector della libreria standard. È stata definita la classe iterator e const_iterator.

DeepPtr

Puntatore smart implementato seguendo la documentazione dello “unique_ptr” della standard library. Esso acquisisce il puntatore alla costruzione e gestisce la copia, assegnazione e distruzione in maniera profonda.

I/O

Il formato del file usato per Input/Output è JSON.

Sono state usate delle classi offerte da Qt per codificare e decodificare il JSON.

Nell'oggetto root sono presenti l'**altezza**, la **larghezza** e il **nome** dell'aquarius, e l'array degli organismi.

Ciascun organismo salva il suo **nome**, la **posizione** (con coordinate **x**, **y**) e per poi riuscire a ritradurlo a pieno, il **tipo** della gerarchia.

```
{
  "height": 720,
  "name": "Unnamed Aquarius",
  "organismi": [
    {
      "name": "Rocky IV",
      "position": {
        "x": 119.2814787249112,
        "y": 314.49982770347395
      },
      "type": "PHYTOPLANKTON"
    }
  ],
  "width": 1024
}
```

Compilazione

Per una migliore organizzazione del progetto sono stati separati in due directory differenti gli header file (.hpp) e i file sorgente (.cpp). Questo però causa un INCLUDEPATH differente da quello generato automaticamente da qmake -project.

Pertanto è stato fornito il file .pro, il quale permette la compilazione tramite

```
$ qmake; make
```

```
OS: Debian 10 (buster)
```

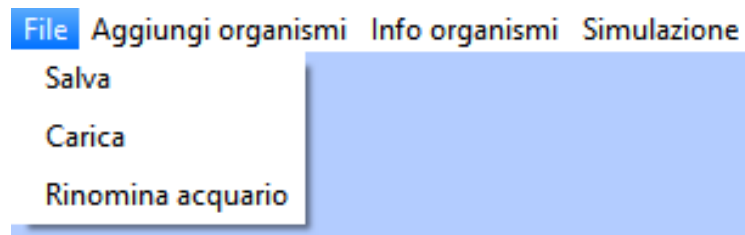
Ambiente di sviluppo

```
gcc version 8.3.0 (Debian 8.3.0-6)
```

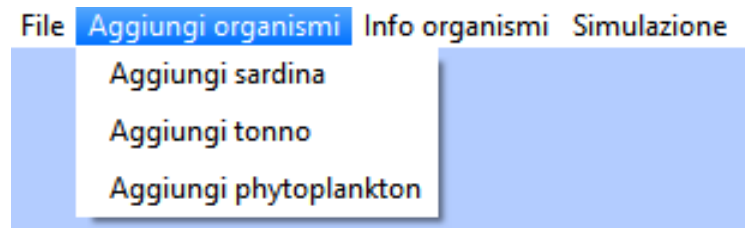
```
Qt version 5.11.3
```

Manuale GUI

File



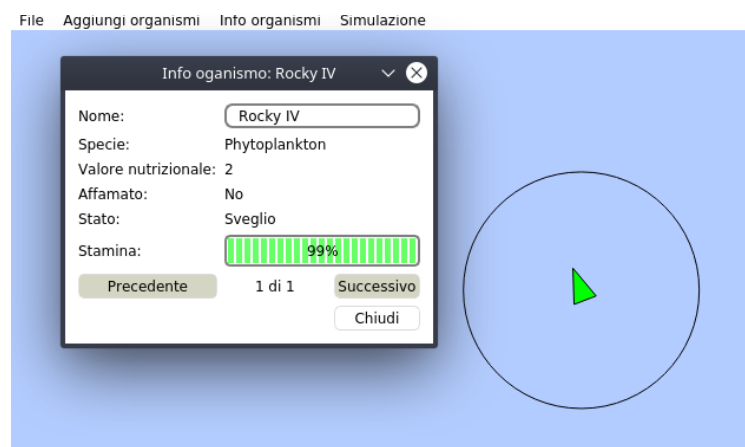
Aggiungi Organismo



Selezionare che tipo di organismo si vuole posizionare nell'acquario.

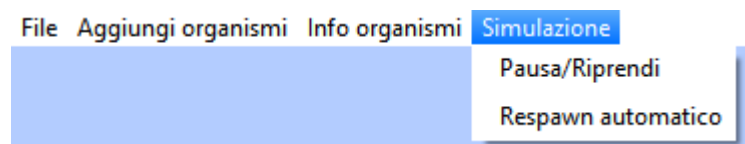
Una volta scelto, cliccare nella posizione dove si vuole crearlo.

Info Organismi



Permette, in tempo reale, di consultare le informazioni di ogni organismo e visualizzarlo all'interno dell'acquario. Per scorrere tra organismi basta usare il tasto precedente e successivo.

Simulazione



Pausa/Riprendi: Ferma/fa riprendere l'esecuzione della simulazione.

Il **respawn automatico**, se attivo, mantiene la popolazione costante, ergo quando un organismo viene mangiato oppure muore di fame, viene rigenerato, mentre invece se fosse spento scomparirebbe.