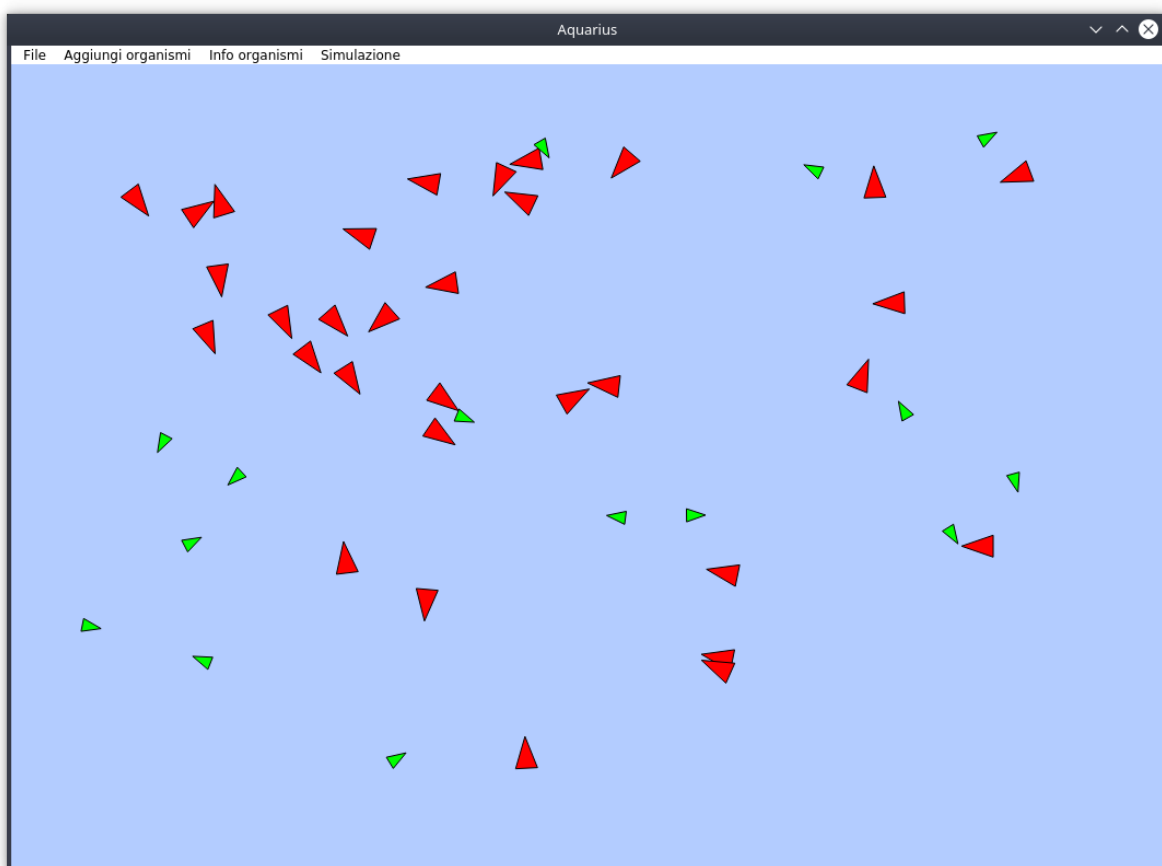


Aquarius

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




Una simulazione di un acquario popolato da organismi che si muovono in base ad alcune regole o secondo casualità.

Indice

Tempo impiegato	2
Suddivisione del lavoro	2
Gerarchia di tipi e uso del polimorfismo	3
Vehicle	3
Vect2D	4
Organismo	4
DayCycle	5
Stamina	5
Tonno	5
Sardina	6
Plankton	6
Phytoplankton	7
Contenitore	7
I/O	7
GUI	8
Compilazione ed esecuzione	9

Tempo impiegato

Argomento	~ 	~ 
	Dardouri	Furlan
Studio dello steering behaviour del 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

Suddivisione del lavoro

Studio dello steering behaviour del Veicolo + Vect2D

Grazie all'interesse pregresso del mio compagno di progetto verso l'ambito trattato, non è stato necessario che io impiegassi tanto tempo in ricerche, in quanto ho potuto usufruire delle risorse da lui consigliatomi per documentarmi sull'argomento.

Progettazione Model + Container

È stata svolta insieme.

Codifica Model

È stata svolta da entrambi in parallelo alternandoci tra le classi, in particolare mi sono dedicata maggiormente alla codifica delle parti relative agli organismi.

Progettazione + Codifica View, Controller, Stile

Me ne sono occupata principalmente io: ha compreso l'apprendimento di Qt, la progettazione e la realizzazione delle schermate dell'applicazione, la connessione tra controller e view, il miglioramento finale della resa estetica con qss.

I/O

Implementazione delle schermate per salvataggio/caricamento dei file, collegamento tra la view, controller e la classe io.

Debugging + Calibrazione

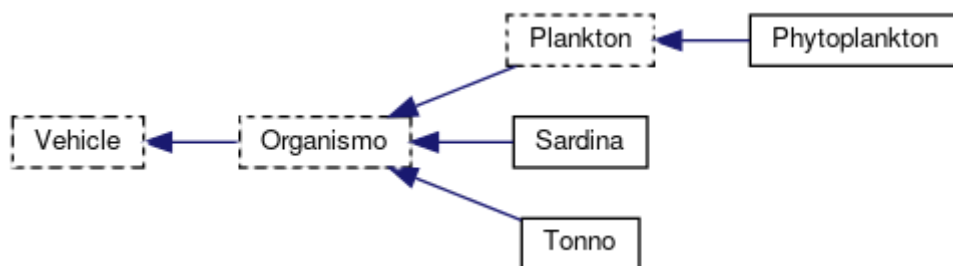
Molti test e aggiustamenti finali per ottenere il risultato desiderato, che mi hanno fatto sfiorare il tetto di 50 ore per il progetto.

Descrizione del progetto

Aquarius è un'applicazione desktop che permette all'utente di creare il proprio acquario personalizzato inserendo tutti gli organismi che desidera, fino ad un massimo di 100 totali. Sono state messe a disposizione tre tipologie di organismo: tonno, sardina e phytoplankton. Una volta posizionati nello spazio, questi si muovono, interagiscono tra loro a seconda del proprio comportamento e si nutrono.

Il pattern architetturale adottato per lo sviluppo è MVC.

Gerarchia di tipi e uso del polimorfismo



Vehicle

La classe base della gerarchia, è una classe **astratta**.

Rappresenta un oggetto generico in grado di muoversi. Possiede una posizione nello spazio (**Vect2D _position**), un'accelerazione (**Vect2D _acc**), una velocità (**Vect2D _velocity**) e un vettore utilizzato per vagare nello spazio senza una meta precisa (**Vect2D _wander**).

Ha inoltre fissate una velocità massima raggiungibile (**const double maxSpeed**) e un'accelerazione massima (**const double maxForce**).

ALCUNI METODI POLIMORFI:

```
virtual void behaviour(Aquarius*) = 0
```

Verrà utilizzato per definire il comportamento.

Puro perché un veicolo generico non possiede un comportamento standard.

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

Pensato per far restituire true se l'oggetto con posizione v si trova nel campo visivo di chi avrà il metodo implementato, false altrimenti.

Puro perché un veicolo generico non possiede un campo visivo.

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

Il comportamento che deve eseguire ogni veicolo ad ogni step di animazione.

È divisa in due fasi: una di calcolo e una di applicazione dei calcoli, questo per fare in modo che tutti i veicoli effettuino le misurazioni basandosi sulla stessa situazione.

Final per bloccare la possibilità di fare override del metodo.

Per permettere, in un'ottica futura, all'interno dell'applicazione l'implementazione di oggetti in movimento che non siano organismi, è stato deciso di creare Vehicle come classe distinta, nonostante con la gerarchia attuale potesse essere fusa in Organismo.

Vect2D

“un vettore è un ente geometrico che rappresenta una grandezza dotata di direzione e verso”

Classe **concreta**.

Ha due campi privati che fungono da coordinate nel piano bidimensionale (**double _x, _y**).

La classe mette a disposizione vari metodi tra cui alcuni per svolgere operazioni matematiche tra due vettori, operazioni tra un vettore e uno scalare, metodi per ruotare il vettore, calcolarne la magnitudine, normalizzarlo ecc. Non possiede metodi virtuali.

Organismo

Classe **astratta**.

Un organismo è un essere vivente che si muove nello spazio.

Deriva da **Vehicle** per acquisire le capacità motorie. Inoltre possiede, come attributi privati, un nome (**std::string _name**), una variabile che tenga traccia se attualmente è sveglio (**bool _awake**) e una che dica se è morto (**bool _gone**) e, come attributi protetti, un daycycle (**DayCycle _daycycle**) e una stamina (**Stamina _stamina**).

ALCUNI METODI POLIMORFI:

virtual void behaviour(Aquarius*) override

Ridefinisce il metodo virtuale di Vehicle, contiene gli istinti naturali di un normale essere vivente: decrementa la stamina e incrementa il daycycle, si occupa di addormentare o svegliare l'organismo, controlla se è affamato e, se lo è, cerca intorno ad esso una possibile preda verso cui andare. Se invece gli istinti naturali sono soddisfatti, l'organismo vaga a caso nello spazio, grazie all'invocazione di alcuni metodi di Vehicle.

virtual bool isHungry() const = 0

Puro perché in ogni organismo la fame può dipendere da soglie diverse della stamina.

virtual int getValoreNutrizionale() const = 0

Puro perché ogni organismo può avere un valore nutrizionale diverso.

virtual std::string getSpecie() const = 0

Puro perché un organismo generico non appartiene a nessuna specie.

virtual bool canSleep() const

Restituisce true se secondo il daycycle dell'organismo è possibile dormire.

È stato reso virtuale per dare la possibilità alle classi derivanti di aggiungere ulteriori condizioni che prevalgano sull'istinto naturale di andare a dormire.

virtual bool canWakeup() const

Restituisce true se secondo il daycycle dell'organismo è possibile svegliarsi.

È virtuale per le stesse motivazioni di sopra e per coerenza.

DayCycle

Classe **concreta**.

Rappresenta un ritmo circadiano: tiene conto della durata relativa di una giornata, ossia la quantità di tempo durante la quale l'organismo dovrà essere sveglio (**unsigned int awakeTime**), e conto della durata relativa di una notte, ossia la quantità di tempo durante la quale l'organismo dovrà dormire (**unsigned int asleepTime**).

Tiene inoltre traccia dello scorrere del tempo per alternare le due fasi (**unsigned int progress**).

Non contiene metodi virtuali, i suoi metodi servono principalmente per accedere ai valori e per incrementarli.

Stamina

Classe **concreta**.

Rappresenta il vigore di un organismo. Tiene traccia del vigore massimo raggiungibile (**double _maxval**) e del valore attuale (**double _val**).

Non contiene metodi virtuali, i suoi metodi servono principalmente per confrontare due vigori, per modificare il vigore e per ottenere il valore attuale in percentuale sul massimo.

In generale la sua funzione è stata pensata in questo modo: quando la stamina di un figlio della classe Organismo arriva sotto ad una certa soglia stabilita in base al suo tipo, diventa affamato e ha bisogno di nutrirsi. Può mangiare solo ciò che ha un valore nutrizionale inferiore al proprio. Se la stamina arriva a 0, muore.

Tonno

Classe **concreta**.

ALCUNI METODI POLIMORFI:

virtual void behaviour(Aquarius*) override

Ridefinisce il metodo virtuale di Organismo. Un tonno tende ad allinearsi agli altri tonni vicini. Infine invoca il metodo di Organismo per soddisfare gli istinti naturali.

virtual bool isInRange(const Vect2D&) const override

Implementa il metodo virtuale puro di Vehicle. Un tonno ha un campo visivo angolare, che copre fino ad una distanza di 100 unità.

virtual bool isHungry() const override

Implementa il metodo virtuale puro di Organismo. Un tonno ha fame se la stamina ha una percentuale inferiore al 40% del totale.

virtual int getValoreNutrizionale() const override

Implementa il metodo virtuale puro di Organismo. Un tonno ha valore nutrizionale 4.

virtual std::string getSpecie() const override

Implementa il metodo virtuale puro di Organismo. Restituisce "Tonno".

Sardina

Classe **concreta**.

ALCUNI METODI POLIMORFI:

virtual void behaviour(Aquarius*) override

Ridefinisce il metodo virtuale di Organismo. Una sardina scappa dai predatori che rileva nel proprio campo visivo, poi invoca il metodo di Organismo per soddisfare gli istinti naturali.

virtual bool isInRange(const Vect2D&) const override

Definisce il metodo virtuale puro di Vehicle. Una sardina ha un campo visivo circolare, che copre fino ad una distanza di 80 unità.

virtual bool isHungry() const override

Ridefinisce il metodo virtuale puro di Organismo.

Una sardina ha fame se la stamina ha una percentuale inferiore al 20% del totale.

virtual int getValoreNutrizionale() const override

Definisce il metodo virtuale puro di Organismo. Una sardina ha valore nutrizionale 3.

virtual std::string getSpecie() const override

Definisce il metodo virtuale puro di Organismo. Restituisce "Sardina".

Plankton

"piccolissimi organismi che vivono sospesi nelle acque lasciandosi trasportare dalle correnti"

Classe **astratta**.

ALCUNI METODI POLIMORFI:

virtual void behaviour(Aquarius*) override

Ridefinisce il metodo virtuale di Organismo.

I plankton cercano di evitare i predatori nel proprio campo visivo.

Infine, invoca il metodo in Organismo per soddisfare gli istinti naturali.

virtual bool isInRange(const Vect2D&) const override

Definisce il metodo virtuale puro di Vehicle. Tutti i plankton hanno lo stesso campo visivo circolare, che copre fino ad una distanza di 50 unità.

virtual bool isHungry() const override = 0

Proviene da Organismo, puro perché non tutti i plankton hanno la stessa soglia di stamina che li rende affamati.

virtual int getValoreNutrizionale() const override = 0

Proviene da Organismo, puro perché non tutti i plankton hanno lo stesso valore nutrizionale.

virtual std::string getSpecie() const override

Definisce il metodo virtuale puro di Organismo. Restituisce "plankton".

Phytoplankton

"l'insieme degli organismi autotrofi fotosintetizzanti presenti nel plancton"

Classe **concreta**.

ALCUNI METODI POLIMORFI:

virtual void behaviour(Aquarius*) override

Ridefinisce il metodo virtuale di Plankton. Un phytoplankton quando è sveglio incrementa la propria stamina facendo la fotosintesi, quando dorme invece decrementa.

Infine, invoca il metodo in Plankton per soddisfare i propri istinti da plankton.

virtual bool isHungry() const override

Ridefinisce il metodo virtuale puro di Plankton.

Un phytoplankton non si nutre di altri organismi, quindi non ha mai propriamente fame.

virtual int getValoreNutrizionale() const override

Ridefinisce il metodo virtuale di Organismo. Un phytoplankton ha valore nutrizionale 2.

virtual std::string getSpecie() const override

Ridefinisce il metodo virtuale di Plankton.

Restituisce "Phyto" concatenato all'invocazione del metodo in Plankton.

Contenitore

Per memorizzare tutti gli organismi che vengono man mano inseriti dall'utente è stato deciso di utilizzare un vettore, così da poter sfruttare l'accesso random agli elementi.

È stata quindi creata una classe templatizzata **Vector** dove sono stati definiti i principali metodi disponibili in un vector della libreria STL. È presente sia l'iteratore in versione costante che quello non. Inoltre è stato definito un puntatore templatizzato **DeepPtr** per la gestione della memoria in modalità profonda.

Queste classi sono state usate per **Vector<DeepPtr<Organismo>> organismi** nella classe **Aquarius**, che rappresenta il modello della nostra applicazione.

Aquarius possiede tra i suoi metodi **void advance()**, ossia il principale per la realizzazione della simulazione all'interno dell'acquario.

I/O

La funzionalità di input/output è stata introdotta nell'applicazione per consentire all'utente di salvare il proprio acquario e permettergli quindi di poterlo ricaricare successivamente.

Il formato che è stato scelto per memorizzare i dati nel file è **JSON**, inoltre sono state usate di supporto alcune classi offerte da Qt tra cui **QFile**, **QJsonDocument**, **QJsonObject** e **QJsonArray**.

Nel file .json vengono salvate le seguenti informazioni:

- **height** e **width**: le dimensioni della finestra principale;
- **name**: il nome dell'acquario;
- **organismi**: un array contenente tutti gli organismi presenti nell'acquario.

Per ogni organismo vengono memorizzati:

- **name**: il suo nome;
- **position**: la sua posizione nella finestra sotto forma di coordinate x e y;
- **type**: il suo tipo.

GUI

L'interfaccia grafica è stata implementata usando due classi:

AcquarioView

Estende **QWidget**, è la schermata principale dell'applicazione.

Uno dei metodi principali è **paintEvent(QPaintEvent*)**, che viene invocato periodicamente da un timer presente nella classe Controller e che serve per aggiornare visivamente le posizioni degli organismi (i calcoli sono infatti svolti in precedenza nel modello).

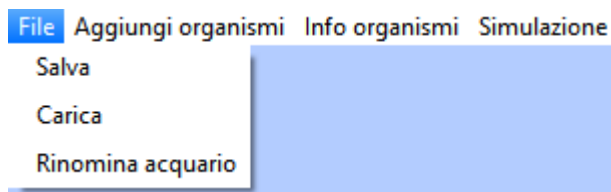
InfoView

Estende **QDialog**, è la finestra che mostra le informazioni sugli organismi presenti.

NOTA: L'applicazione è stata esteticamente modificata con uno stylesheet **.qss**, che poi è stato unito ai file **.cpp** dato che il progetto non ne prevedeva la consegna.

L'applicazione all'apertura si presenta come una schermata vuota azzurra con una barra dei menu in alto. La menu bar è suddivisa in 4 sezioni:

1. File

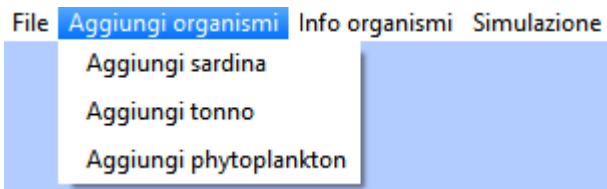


Salva: Per salvare la simulazione in corso;

Carica: Per caricare una simulazione precedentemente salvata;

Rinomina acquario: Per rinominare l'acquario, che di default si chiama "Unnamed Aquarius".

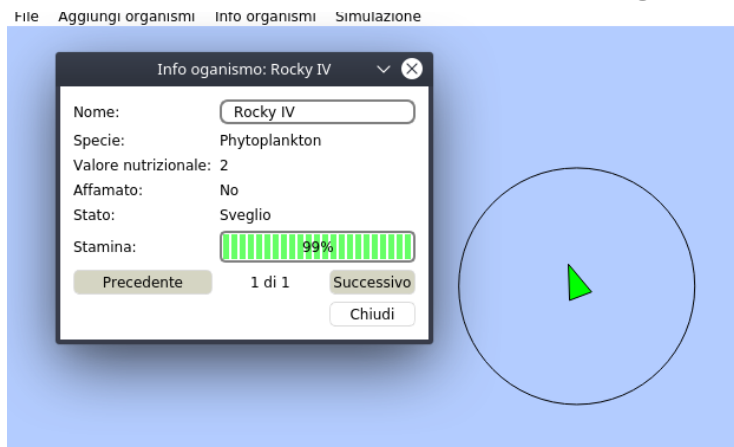
2. Aggiungi organismi



Sono presenti tre strumenti a selezione esclusiva, dopo averne selezionato uno basterà cliccare in un punto qualsiasi della finestra per creare l'organismo desiderato.

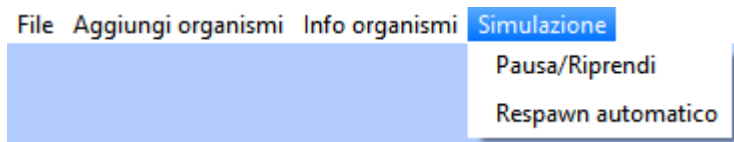
L'aspetto degli organismi (in dimensione e colore) dipende da quali organismi sono presenti in quel momento, infatti se viene aggiunto un organismo con valore nutrizionale maggiore, quelli con valore più basso diventano più piccoli e cambiano colore.

3. Info organismi



Si apre una finestra che mostra varie informazioni riguardanti gli organismi presenti. L'organismo cui appartengono quei dati viene messo in evidenza da un cerchio, come nell'immagine. È anche possibile cambiare il **nome** dell'organismo, digitando nella rispettiva casella.

4. Simulazione



Pausa/Riprendi: arresta/fa ripartire il movimento di tutti gli organismi;

Respawn automatico: quando è selezionato, da quel momento in

poi per ogni organismo che muore (di fame oppure se è stato mangiato) ne viene creato uno nuovo dello stesso tipo, che appare al centro dello schermo. Il numero di organismi resta quindi sempre invariato.

Compilazione ed esecuzione

Per organizzare meglio il progetto sono stati separati in due directory differenti gli header file (.hpp) e i file sorgente (.cpp). Questo però implica un INCLUDEPATH differente da quello generato automaticamente dal comando **qmake -project**.

Pertanto, nel progetto è stato fornito anche il file .pro, il quale permette la compilazione tramite i comandi **qmake → make**.

Specifiche usate:

- Sistema operativo:** Windows 10 Home
- Compilatore:** gcc 6.3.0 e GNU g++ 7.3 (macchina virtuale)
- Versione Qt:** 5.9.5