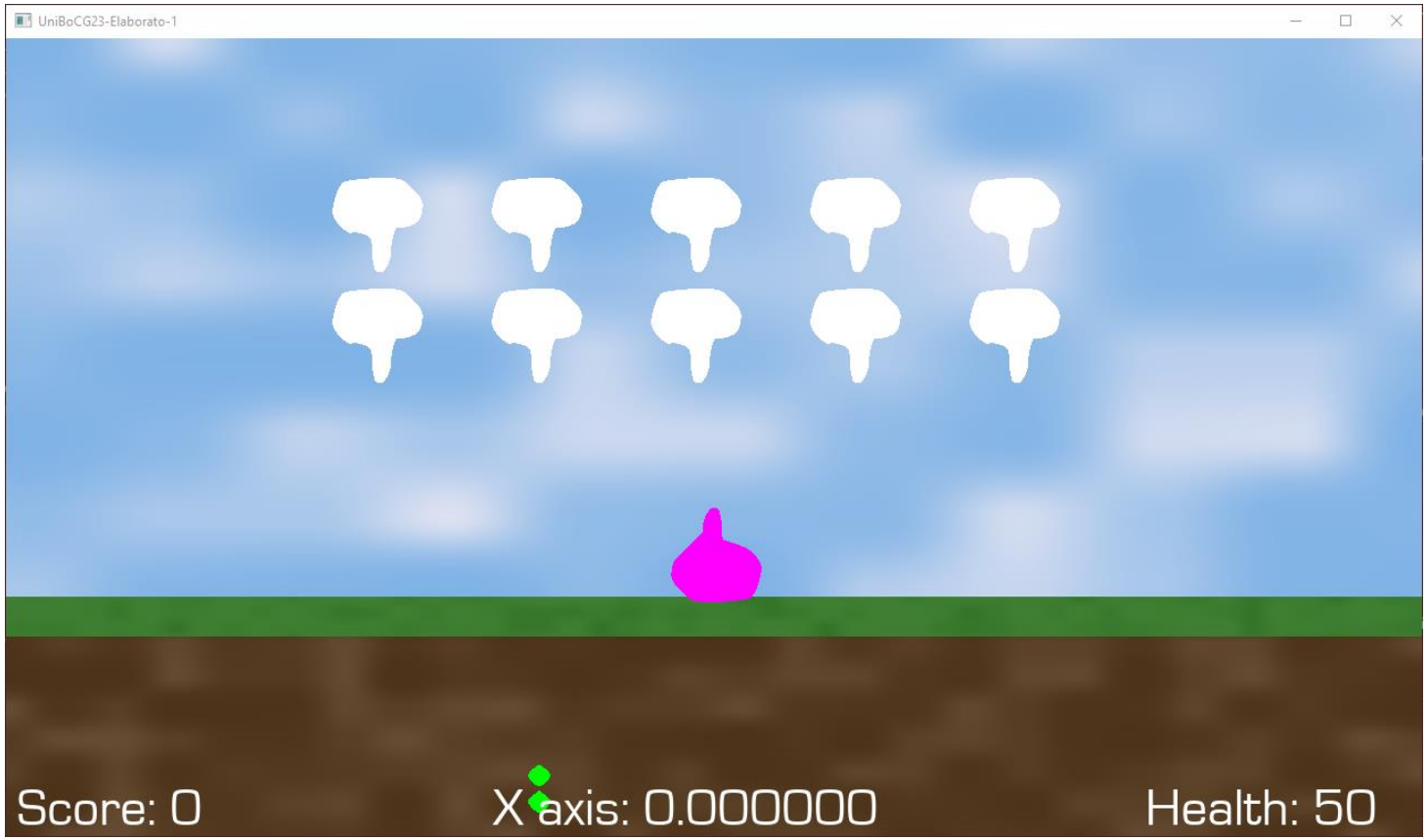
UniBo Computer Graphics

Progetto 1

Struttura

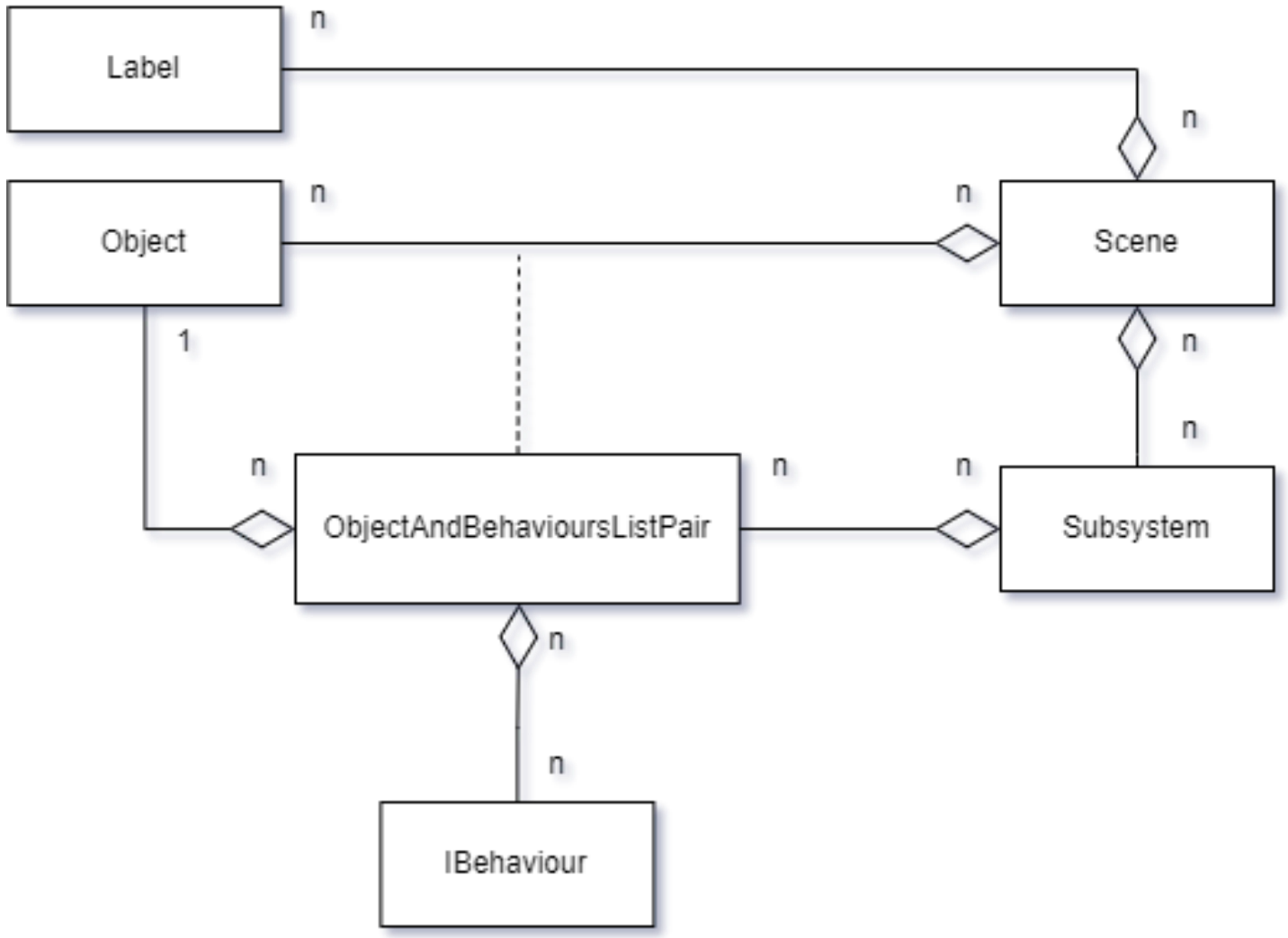
L’elaborato è stato costruito seguendo il paradigma di programmazione a oggetti, anche se, per la natura delle librerie utilizzate originariamente concepite per C, in alcune parti non viene completamente rispettato.



Scena

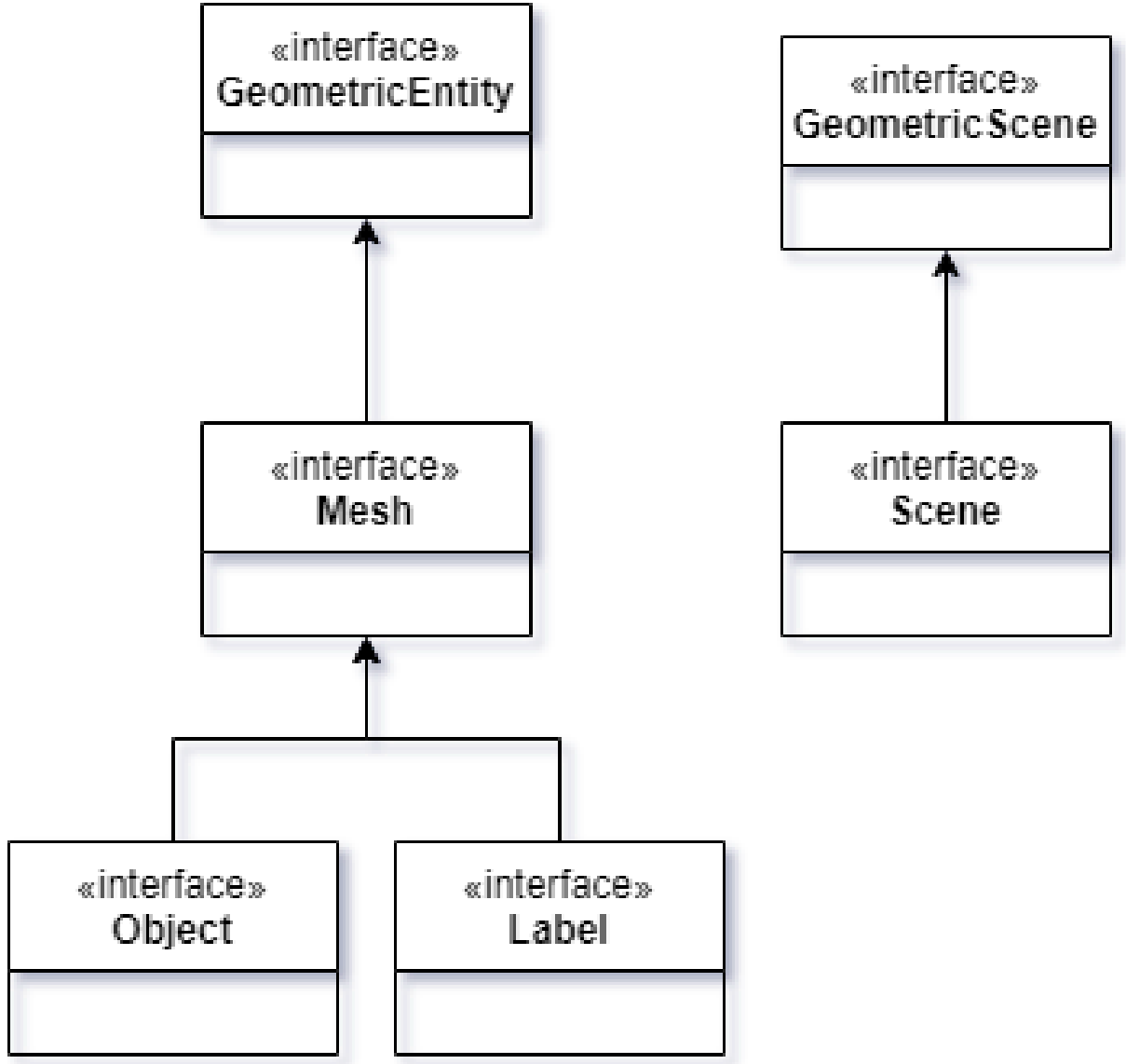
Una scena può essere composta da testo, oggetti e comportamenti che controllano questi.

In questo modo diventa molto facile creare una scena che cambia nel tempo tramite i vari comportamenti assegnati agli oggetti.



Ereditarietà

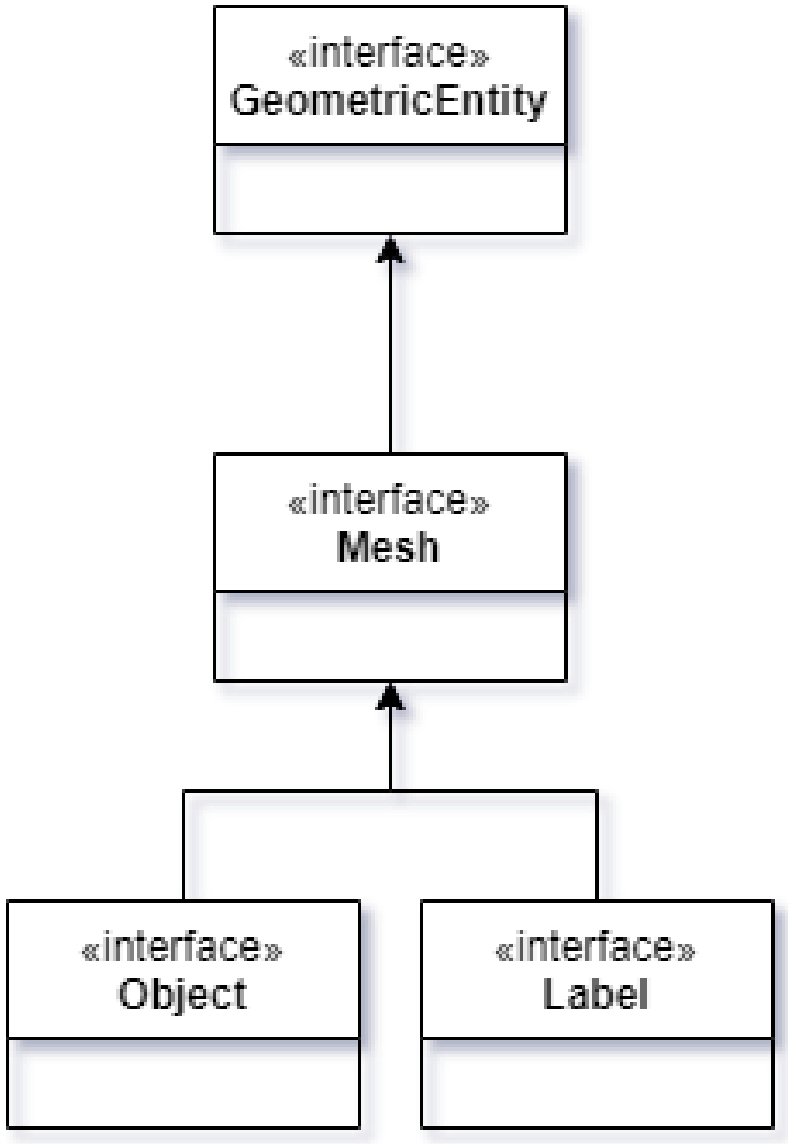
A causa di molte componenti che condividono le stesse proprietà, è stato importante definire un sistema di ereditarietà per semplificare lo sviluppo



Ereditarietà - Elementi scena

Dalla parte degli elementi delle scene, ci sono 3 livelli di ereditarietà:

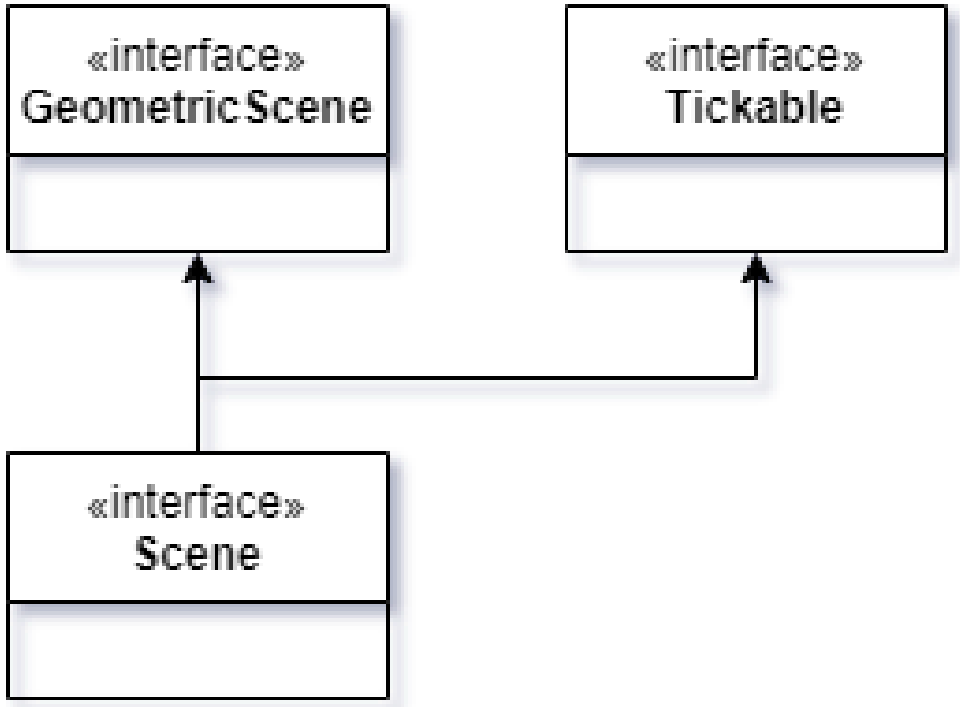
- **GeometricEntity**: rappresenta semplicemente un punto nello spazio con una posizione, rotazione e scalatura
- **Mesh**: possiede geometria, colori, indici e una matrice di modellazione
- **Object**: possiede identificatore e un nome
  Label: come object, ma concettualizzato per il testo



Ereditarietà - Scena

L’idea di scena è costruita su diverse classi

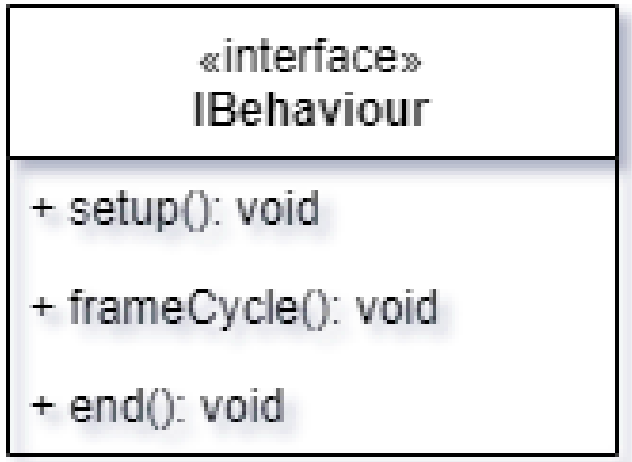
- **GeometricScene**: rappresenta una semplice scena che contiene **GeometricEntity**
- **Tickable**: una entità che esegue delle azioni per ogni tick
- **Scene**: molto più complessa, gestisce oggetti, comportamenti e sottosistemi, usando il sistema di tick per temporizzare le operazioni



IBehaviour

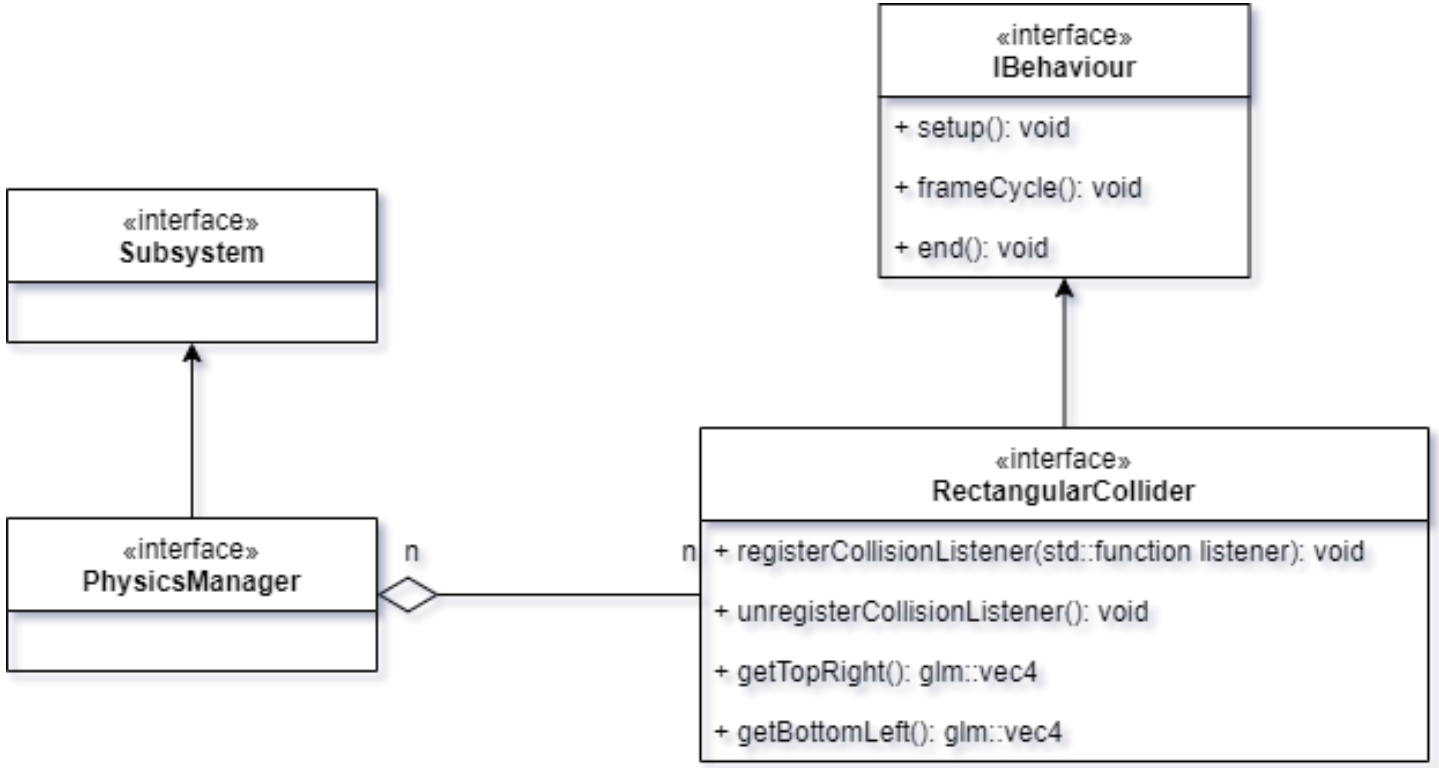
L’interfaccia IBehaviour rende disponibile 3 funzioni:

- setup: viene chiamata all’abbinamento del behaviour all’oggetto
- frameCycle: viene chiamato per ogni singolo frame
- end: viene chiamato quando l’**Object** viene distrutto o quando il behaviour viene staccato dall’**Object**.



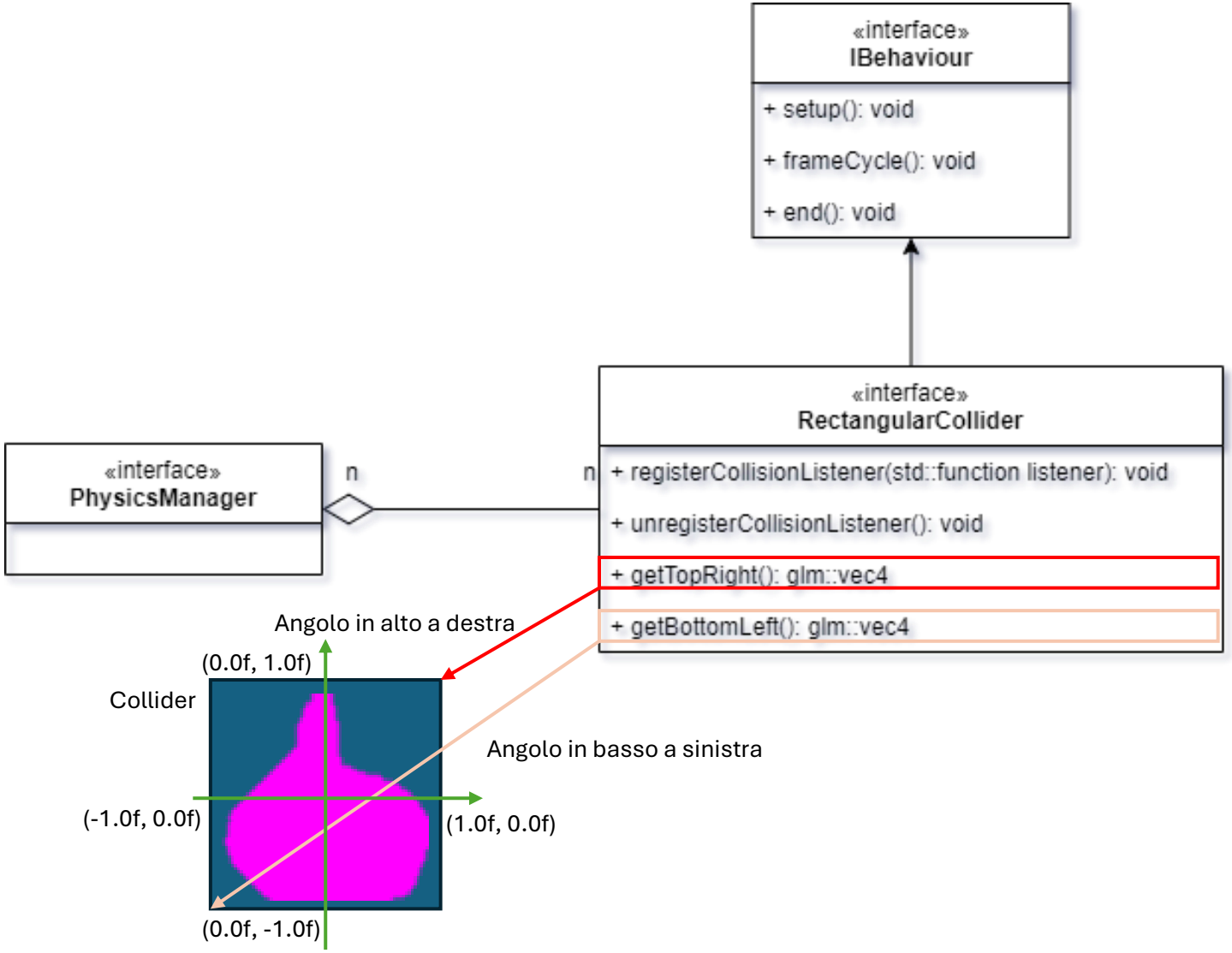
Subsystem

Una scena contiene dei subsystem che si occupano di gestire interazioni con alcuni tipi di behaviour predefiniti, come ad esempio un behaviour di tipo hitbox verrà processato dal physicsmanager per ogni frame, permettendo di gestire l'interazione tra due corpi



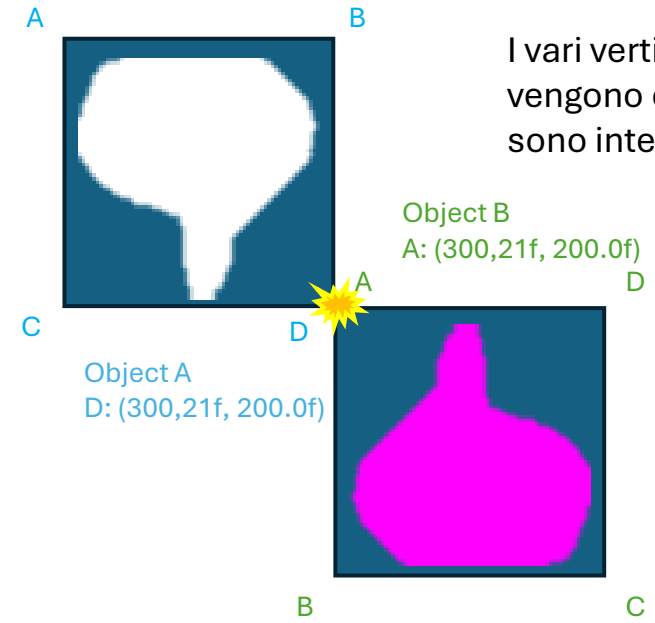
Collisioni

Per comodità, le coordinate della hitbox sono impostate tramite punti normalizzati tra -1 e 1, in modo da rendere semplice la costruzione di una hitbox guardando la geometria della mesh 2D.



Collisioni

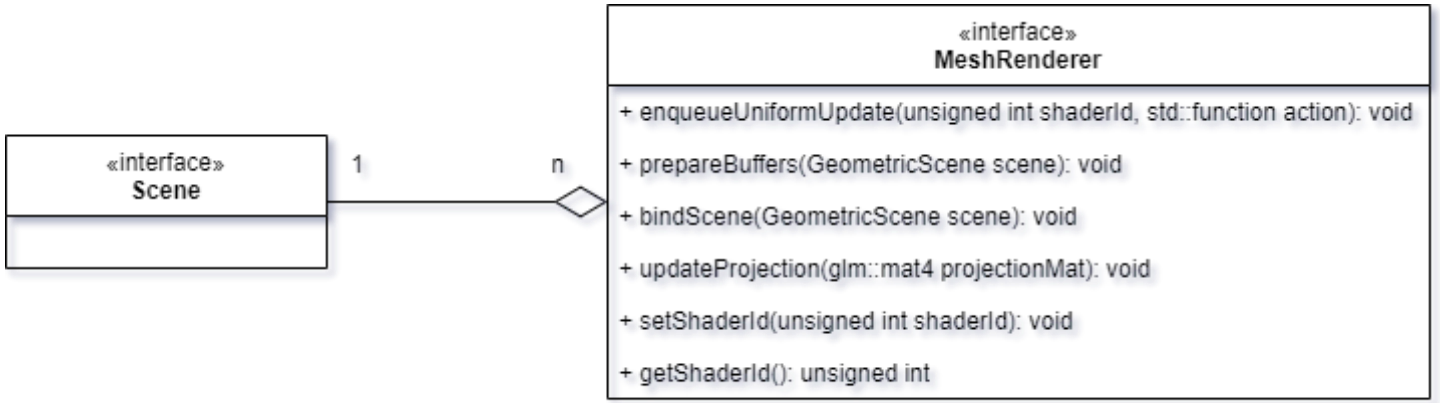
Le coordinate locali della mesh vengono successivamente trasformate in quelle del mondo dal PhysicsManager per calcolare le hitbox.

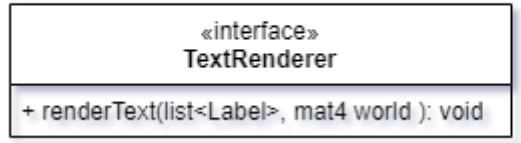


I vari vertici delle hitbox rettangolari vengono confrontati per controllare se ci sono intersezioni per ogni frame

**alcuni valori sono fittizi per questione di semplicità di illustrazione*

Rendering



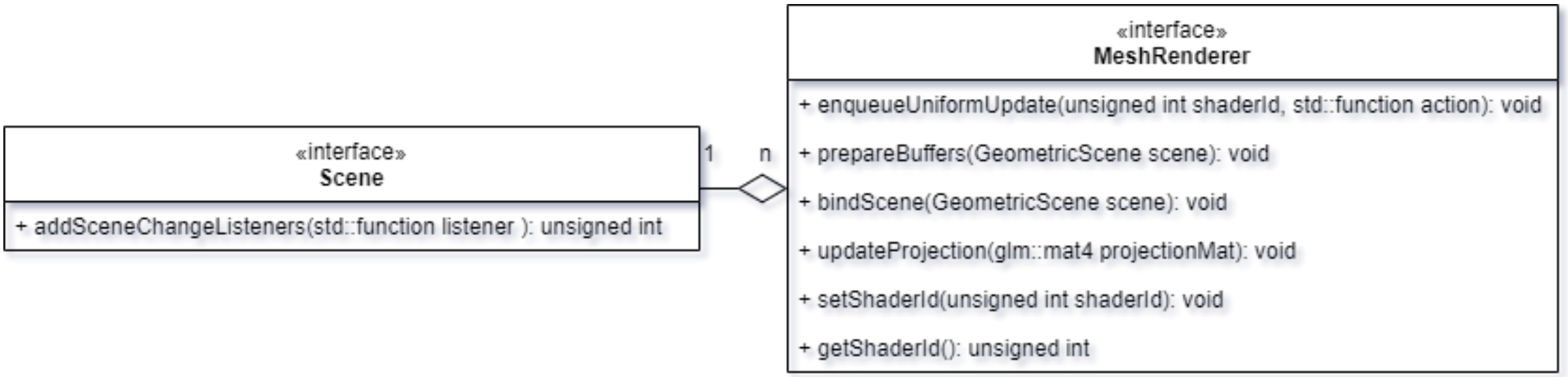


Nel progetto sono presenti due renderer diversi

- **MeshRenderer**: si occupa di eseguire il rendering della geometria che riguarda il gioco
- **TextRenderer**: si occupa di mostrare il testo per l'interfaccia utente

MeshRenderer

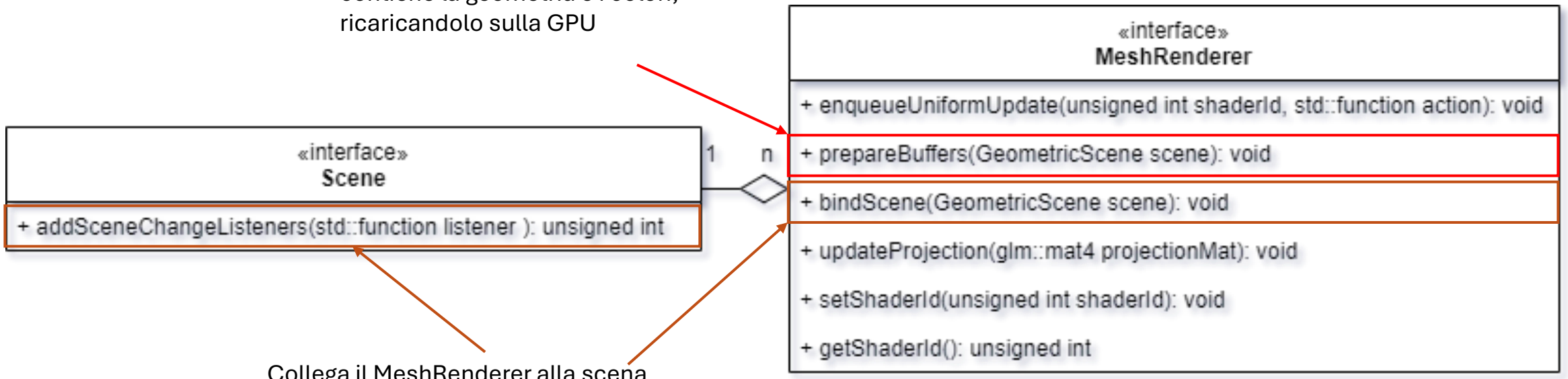
Permette il rendering degli oggetti di una scena, collegandosi ad essa e aggiorna i buffer della GPU ogni qualvolta che avviene un cambiamento della geometria della scena



MeshRenderer

Il collegamento alla scena viene eseguito tramite la funzione prepareBuffers, dopo di che ad ogni aggiornamento alla scena verrà automaticamente rilevato dal MeshRenderer per aggiornare le draw call.

Causa la ricostruzione del buffer che contiene la geometria e i colori, ricaricandolo sulla GPU

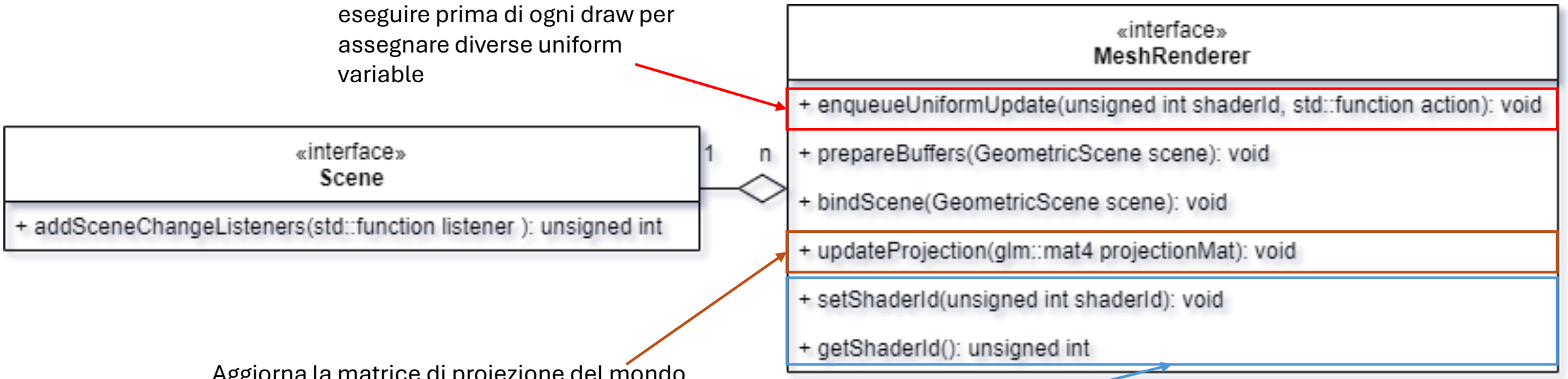


Collega il MeshRenderer alla scena tramite un observer usando la funzione addSceneChangeListeners

MeshRenderer

Per la sua struttura il MeshRenderer utilizzerà sempre lo stesso shader program per eseguire il rendering su tutti gli oggetti presenti nella scena passata.

Imposta delle funzioni da eseguire prima di ogni draw per assegnare diverse uniform variable



Aggiorna la matrice di proiezione del mondo (come enqueueUniformUpdate, ma orientato per la matrice di proiezione)

Il meshrenderer utilizza uno shader program, che è possibile ottenere o scambiare in qualsiasi momento passandogli un identificatore diverso.

MeshRenderer

Funzionamento

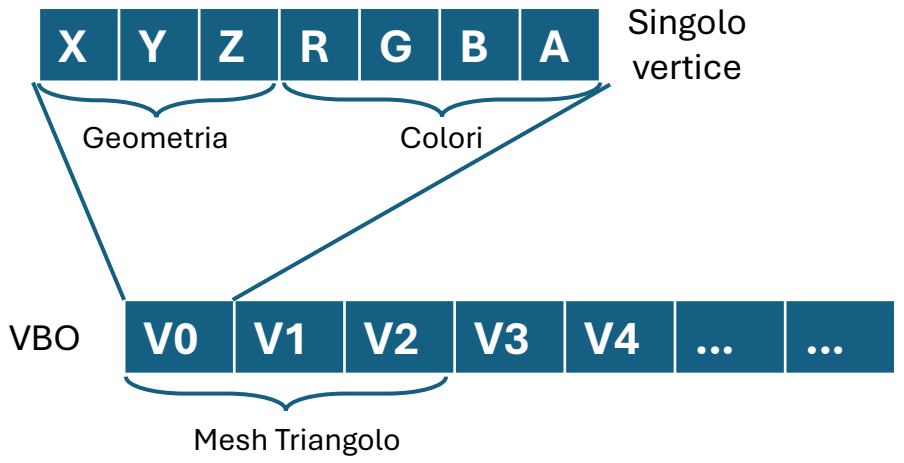
Binding della scena:

1. Colllegamento observer alla scena
2. Chiamata funzione prepareBuffers

Funzione prepareBuffers:

1. Recupero delle mesh dalla scena
2. Inizializzazione VAO
3. **Nel caso di una mesh di triangoli** viene dichiarato un VBO grande quanto:
   $N_{vertici} * sizeof(float) * 3 + N_{colori} * sizeof(float) * 4$
4. Gli indici di inizio e di fine per ogni poligono vengono salvati in un buffer che verrà utilizzato durante le chiamate di disegno

Esempio
(per il rendering di un triangolo)



Gli indici nel VBO verranno salvati e associati alla mesh per essere utilizzati durante le chiamate di disegno.
Nel nostro esempio:

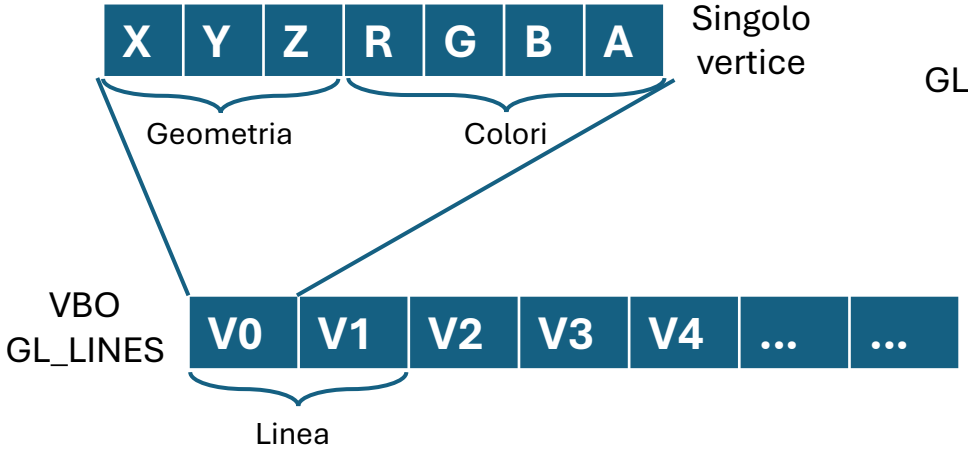
$StartIndex = 0$
$Offset = 2$

MeshRenderer

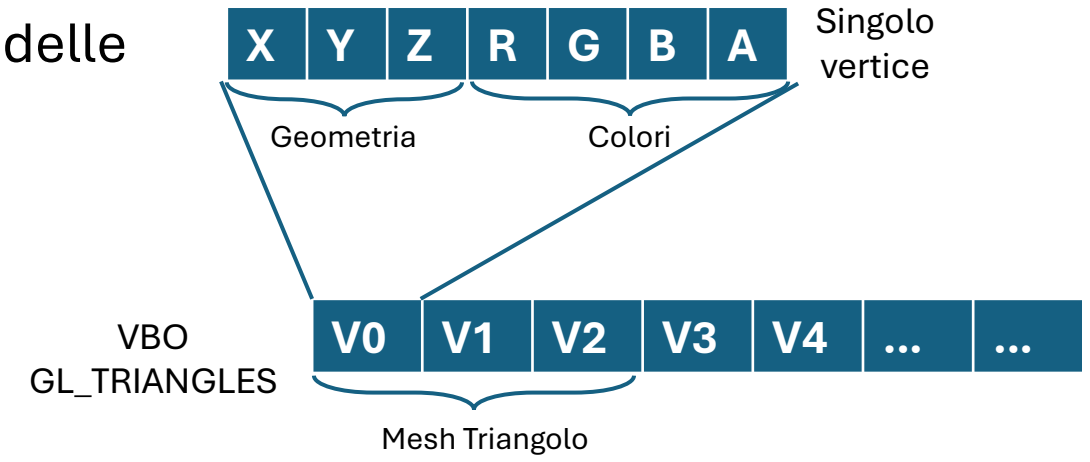
Funzionamento

Viene creato un VBO diverso per ogni singolo draw type, a causa della differenza tra le costruzioni delle diverse geometrie

Esempio
(per il rendering di una linea)



Esempio
(per il rendering di un triangolo)



Gli indici nel VBO verranno salvati e associati alla mesh per essere utilizzati durante le chiamate di disegno.
Nel nostro esempio:

$$StartIndex = 0$$
$$Offset = 2$$

MeshRenderer

Osservazioni

L’utilizzo di un unico shader program e anche di un unico VBO per draw type dello stesso tipo, rende molto efficiente il processo di rendering, poiché la scheda grafica non deve perdere tempo a eseguire switching di contesto.

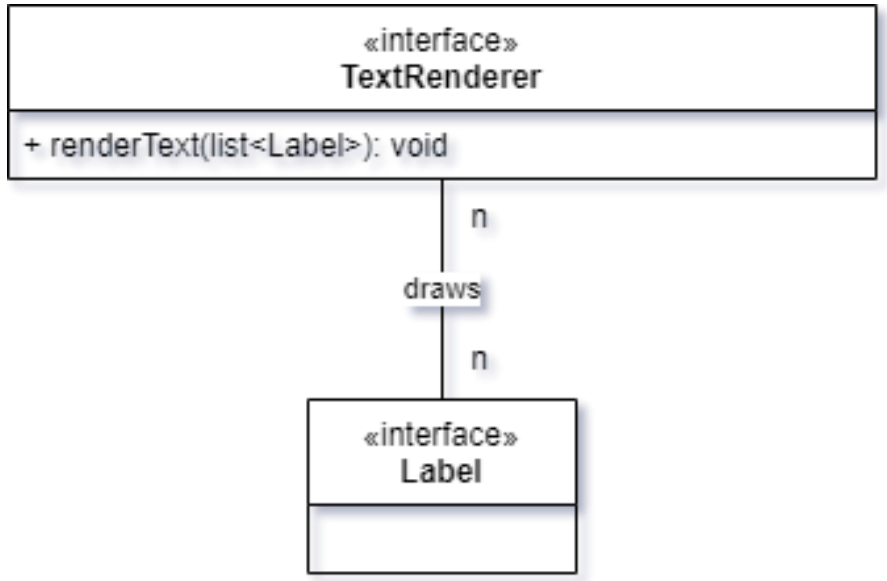
Tuttavia non tutte le ottimizzazioni sono state implementate, come ad esempio:

1. L’utilizzo di un EBO, per memorizzare gli indici, riducendo drasticamente l’utilizzo di memoria grafica, evitando di ripetere vertici condivisi
2. L’hashing delle mesh per evitare di caricare geometrie ridondanti nel vettore, abbattendo ulteriormente i costi di memoria grafica a discapito di un maggiore tempo di creazione / importazione di un asset a runtime

TextRenderer

Come suggerisce il nome, il text renderer ha lo scopo di caricare un font sulla scheda grafica e di eseguire il rendering dei caratteri nell'interfaccia utente.

Dato un percorso e un nome del font e un identificatore dello shader del testo, il TextRenderer si occuperà di caricare i glifi, allocandoli nella memoria grafica.
Al livello di funzioni, il TextRenderer è più semplice, poiché non ha bisogno di una scena per funzionare, ma solo di una lista di label da renderizzare sullo schermo.

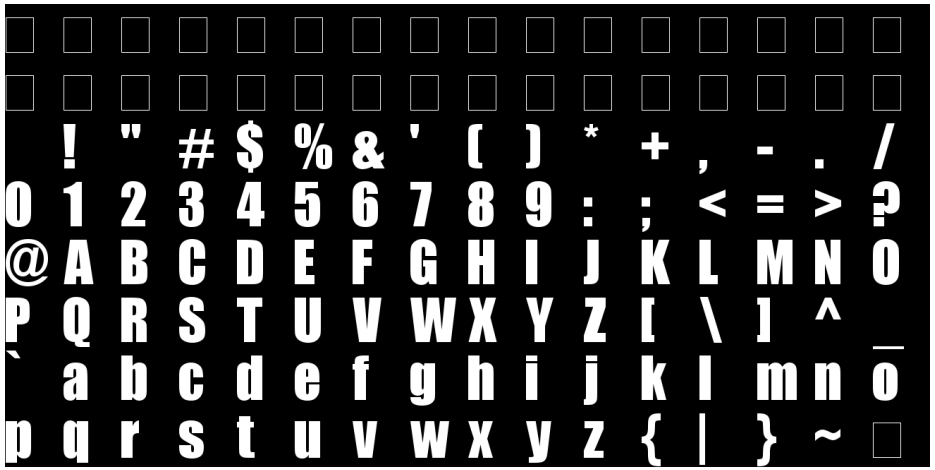


TextRenderer

Funzionamento

Per la gestione del font, il TextRenderer utilizza la libreria FreeType, estrapolando le bitmap dei caratteri, e successivamente caricandole come texture nella GPU.

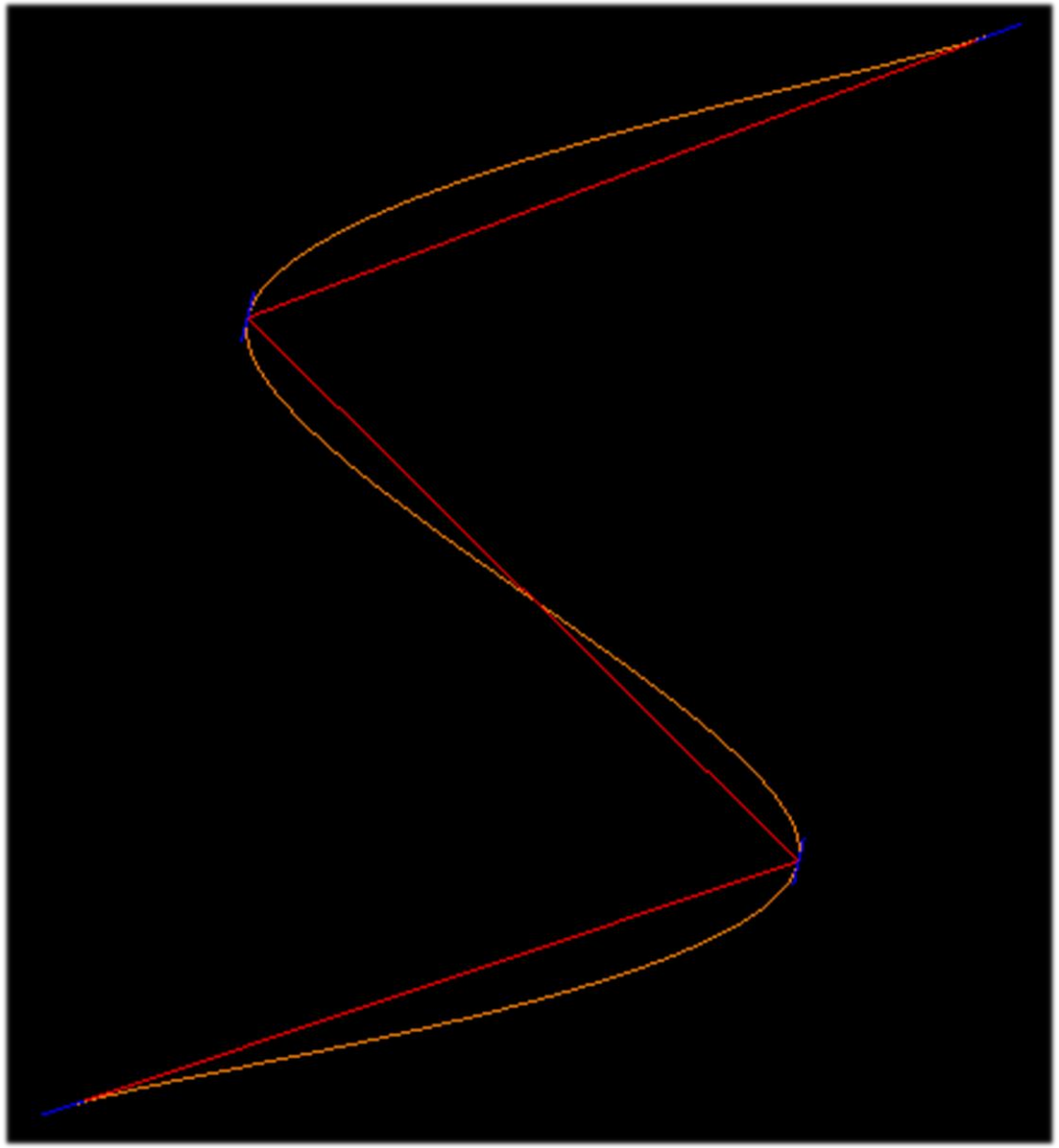
Durante il rendering, la texture corrispondente al carattere viene mappata su un quadrato (o rettangolo) che rappresenta l'area di disegno del carattere sullo schermo.



Costruzione nemici, giocatore e proiettili

Fatta eccezione per lo sfondo, la geometria del gioco si affida molto sull'utilizzo delle curve interpolanti di Hermite, in particolare, di **Spline TCB**.

Data una collezione di vertici di controllo, questi vengono interpolati per creare delle forme dolci, riducendo nettamente lo sforzo che ci vorrebbe modellando a mano la forma.

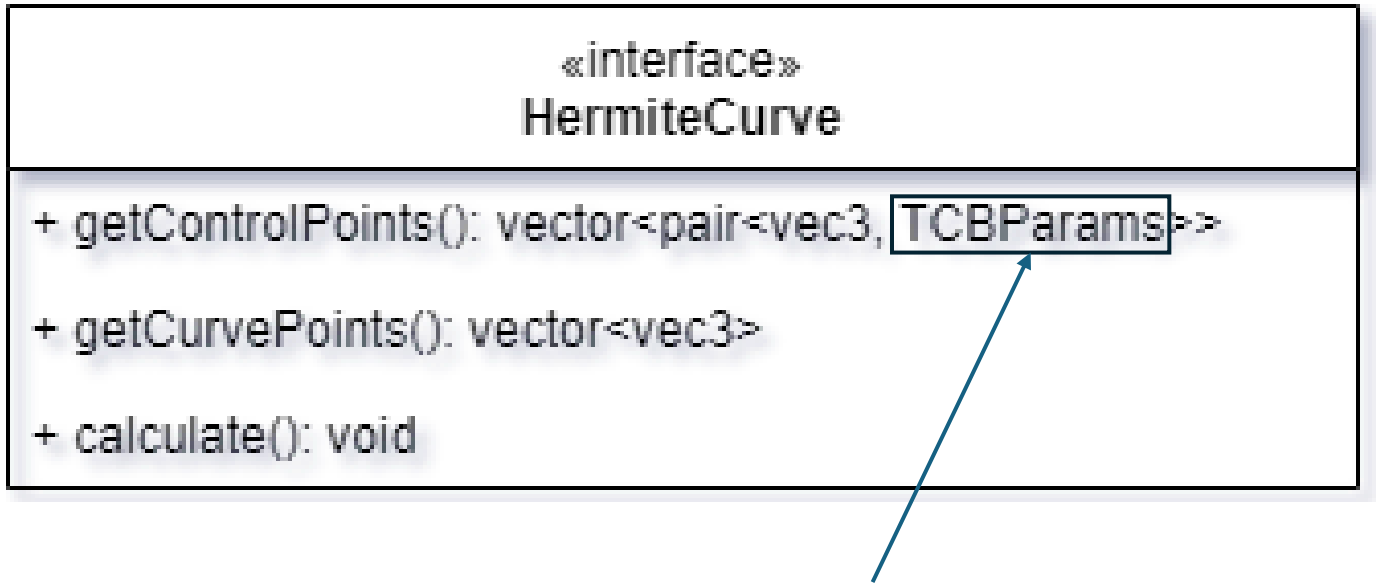


Curve di Hermite

Le curve vengono generate utilizzando la tecnica delle **Spline TCB**, una tecnica che permette il controllo raffinato sulle forme della curva tra i punti di controllo del segmento.

Ogni punto di controllo possiede:

- **Tensione**: rigidità della curva
- **Continuità**: fluidità con cui la curva entra ed esce dai punti di controllo
- **Bias**: controlla la pendenza della curva prima e dopo ogni punto di controllo

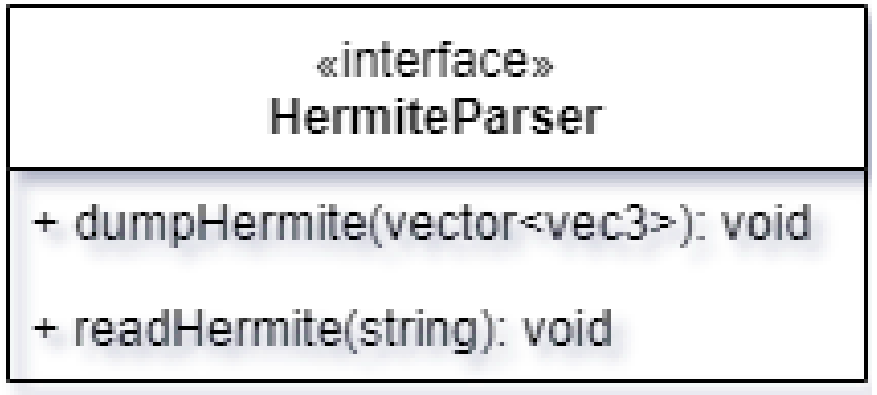


Un semplice enum contenente i 3 parametri di controllo

Curve di Hermite

Per comodità le curve vengono salvate come un file di testo contenente la lista dei vertici di controllo in coordinate XYZ.

Essendo la forma geometrica sotto forma di testo diventa molto facile alterare a mano la geometria dei vari elementi di gioco.

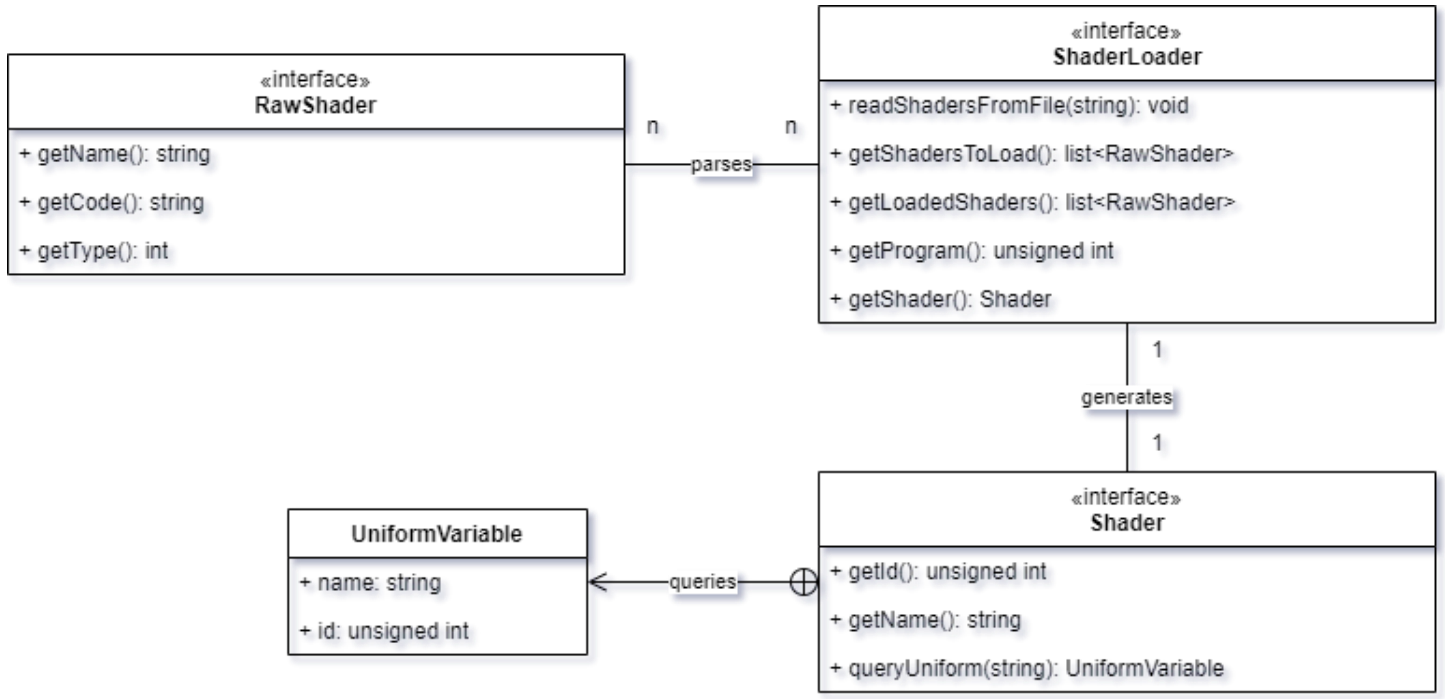


Per esempio questi sono i vertici di controllo, della navicella del giocatore.

```
-0.103125  0.391667  0
-0.0875  0.508333  0
-0.025  0.605556  0
0.0390625  0.597222  0
0.0671875  0.483333  0
0.071875  0.438889  0
0.0890625  0.302778  0
0.20625  0.252778  0
0.296875  0.247222  0
0.423437  0.1  0
0.395313  -0.0916666  0
0.339062  -0.191667  0
0.251562  -0.219444  0
0.065625  -0.236111  0
-0.179688  -0.227778  0
-0.310938  -0.127778  0
-0.365625  -0.0583333  0
-0.385938  0.0194445  0
-0.345312  0.152778  0
-0.215625  0.211111  0
-0.139063  0.227778  0
-0.101562  0.391667  0
```

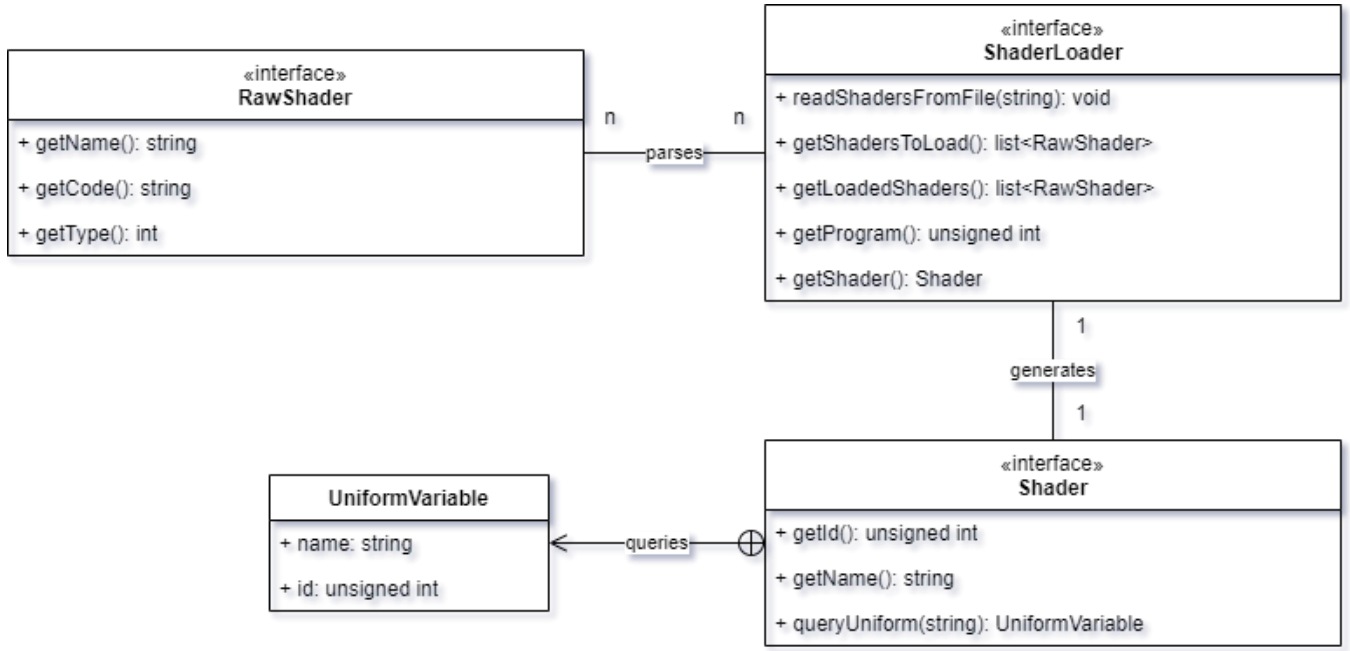
Gestione Shader

Gli shaders vengono gestiti tramite diverse classi che hanno scopi ben definiti in modo da semplificare la creazione e la compilazione di uno shader program



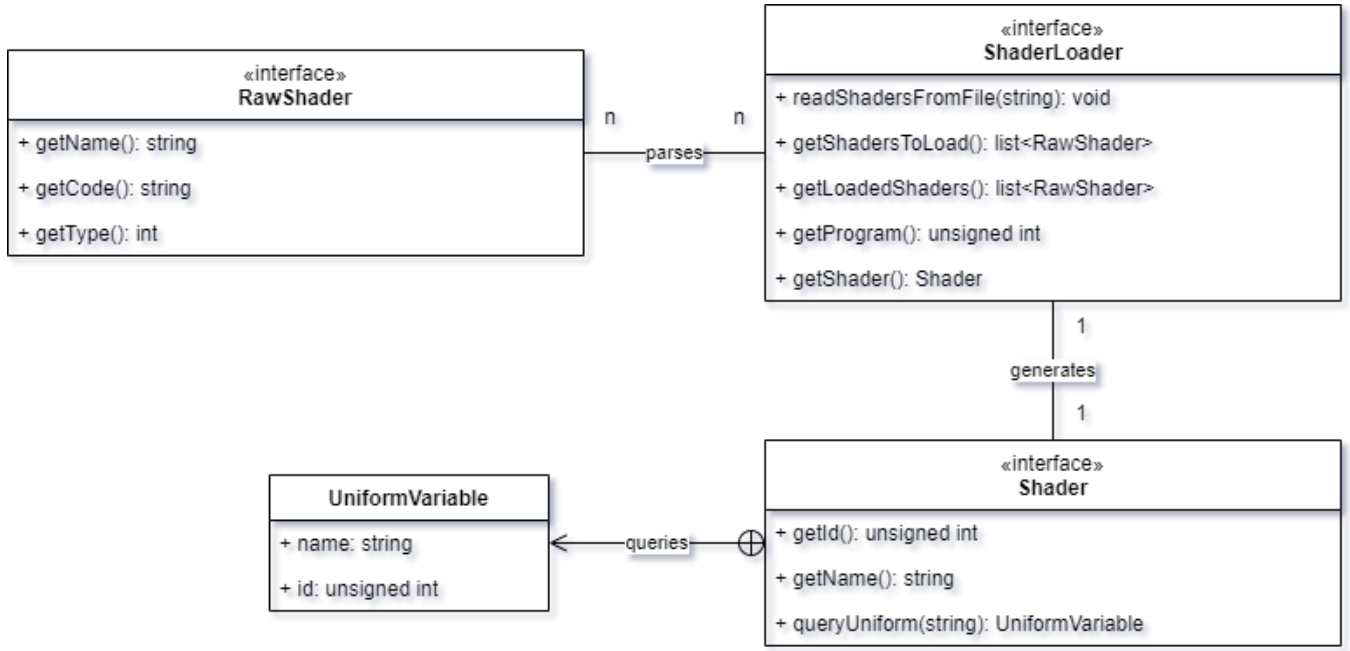
Gestione Shader
Componenti

- **Raw Shader**: contiene un nome che lo identifica, il codice sotto forma di stringa e il tipo di shader (fragment, vertex, etc...)
- **Shader**: rappresenta uno shader compilato, possiede il suo identificatore, il suo nome, e una funzione che permette facilmente di recuperare l'identificatore di una variabile uniform
- **UniformVariable**: possiede un nome e un id che verrà usato per selezionare la variabile uniform da modificare



Gestione Shader
Componenti

- **ShaderLoader**: raccoglie i diversi RawShader per poi compilarli alla chiamata della funzione compileShaders.
  Una volta completata la compilazione, è possibile recuperare lo shader compilato tramite la funzione getShader.



Shader

Il progetto funziona utilizzando solamente 2 shader

- Vertex e Fragment shader per la geometria
- Vertex e Fragment shader per il testo

Shader testo

- **Vertex Shader**: prende le coordinate del vertice, applicandogli la matrice di trasformazione e passando al fragment shader le coordinate di texture
- **Fragment Shader**: campiona la texture corrispondente al glifo su ogni frammento per poi applicarla, aggiungendoci il colore definito nel label

Shader geometria

- **Vertex Shader**: molto semplice, applica la matrice di proiezione al vertice, mandando il colore al fragment shader.
- **Fragment Shader**: diviso logicamente in diverse parti
  - **Generazione dello sfondo**:
    Si utilizza la posizione del frammento basandosi sulla risoluzione, per dividere lo sfondo in sezioni:
    - **Cielo 70% sfondo**: si utilizza il tempo di runtime e la risoluzione dello schermo passate come uniform, applicate a una funzione di generazione del rumore per generare delle nuvole che cambiano nel tempo e che si muovono
    - **Erba 5% e Terreno 25%**: si utilizza la stessa funzione di generazione di rumore, questa volta senza il tempo, per creare un terreno con imperfezioni

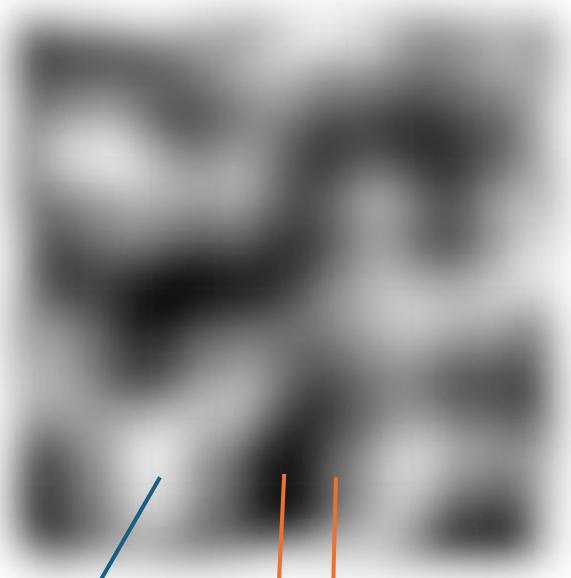
Shader Geometria

Dettagli

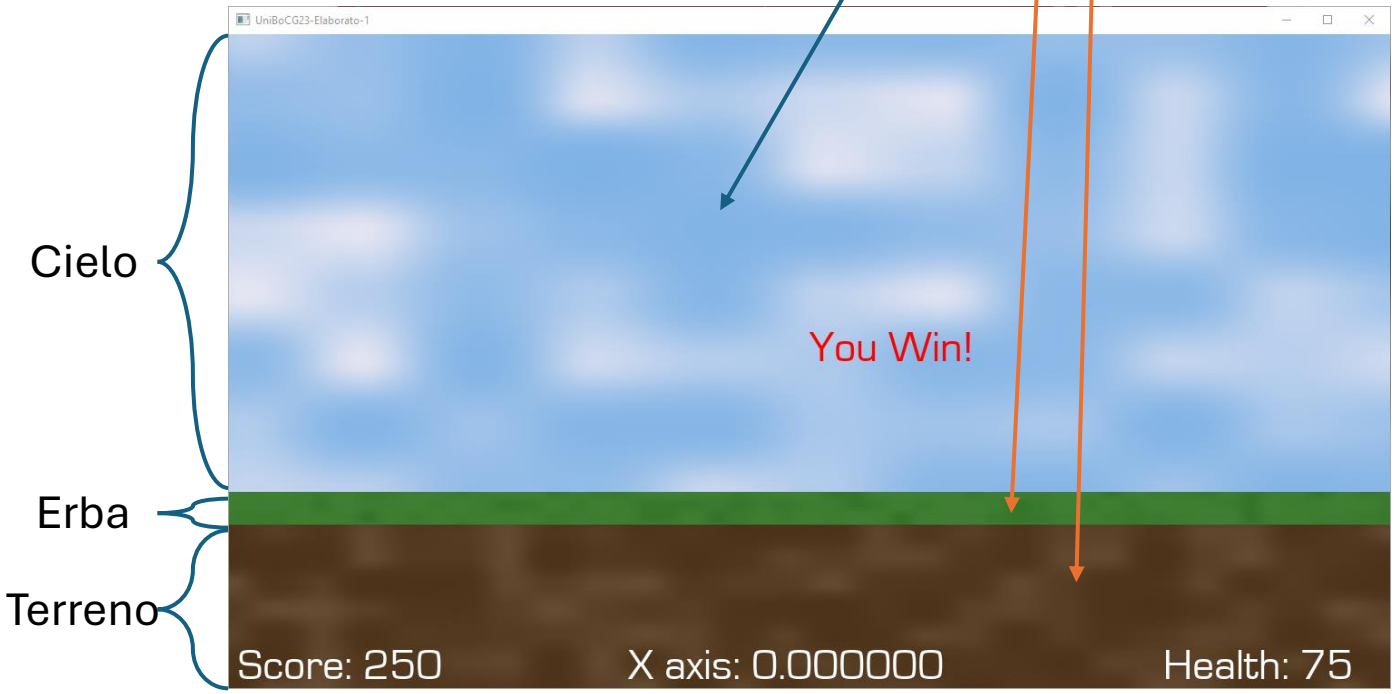
Per la generazione di questo effetto procedurale, nel codice dello shader sono presenti due funzioni fondamentali per la generazione del rumore:

- **rand**: genera un valore pseudocasuale tra 0 e 1 basandosi sulle coordinate 2D
- **noise**: crea un valore di rumore fluido e smussato, utilizzando rand per la generazione di valori casuali ai vertici di una griglia attorno alla coordinata, interpolando poi questi valori con una funzione smoothstep, producendo forme più dolci

Funzione di rumore



Per animare il cielo viene aggiunto un offset (tempo) alla coordinata x
simulando uno spostamento delle nuvole a sinistra



Fonte noise: https://thebookofshaders.com/11/