

UNIVERSITA' DEGLI STUDI DI NAPOLI "PARTHENONE"  
FACOLTÀ DI SCIENZE E TECNOLOGIE  
CORSO DI LAUREA IN INFORMATICA APPLICATA  
MACHINE LEARNING E BIG DATA



# Fruit Ninja GL

## Clone OpenGL del famoso gioco mobile

RELAZIONE PROGETTO COMPUTER GRAPHICS: ANIMATION AND SIMULATION

DOCENTE

*Prof. Maurizio DE NINO*

STUDENTE

*Giuseppe Pio Cianci*

0120000178

Anno Accademico 2018-2019



# Indice

Indice	i
<b>1 Introduzione</b>	<b>1</b>
1.1 Introduzione . . . . .	1
1.1.1 Gameplay . . . . .	1
1.2 Fruit Ninja GL . . . . .	2
1.2.1 Librerie e Software Utilizzati . . . . .	2
1.2.2 Documentazione . . . . .	3
1.2.3 Avvio dell'applicazione . . . . .	4
<b>2 Fruit Ninja GL</b>	<b>5</b>
2.1 Overview . . . . .	5
2.2 Descrizione Architetturale . . . . .	6
2.2.1 Modulo core/ . . . . .	6
2.2.2 Modulo ecs/ . . . . .	8
2.2.3 Modulo engine/ . . . . .	8
2.2.4 Modulo utl/ . . . . .	9
2.3 Entity Component System . . . . .	9
2.3.1 Database . . . . .	10
2.3.2 Entità . . . . .	10
2.3.3 Componenti . . . . .	11
2.3.4 Sistemi . . . . .	14
2.3.5 Esempio di utilizzo . . . . .	17
2.4 OpenGL . . . . .	18
2.4.1 Texture . . . . .	18
2.4.2 Mesh . . . . .	20
2.4.3 Model . . . . .	21
2.4.4 Shader . . . . .	21
<b>Bibliografia</b>	<b>27</b>



# Introduzione

Relazione relativa al progetto d'esame “FruitNinjaGL (fn-GL)” del corso di Computer Graphics: Animation and Simulation (GraficInt, aa 2019)

1.1 Introduzione . . . . .	1
1.2 Fruit Ninja GL . . . . .	2

## 1.1 Introduzione

L'applicativo sviluppato come progetto d'esame è un clone in OpenGL del famosissimo **Fruit Ninja** videogioco sviluppato dalla *Halfbrick Studios* e pubblicato nel 2010 per sistemi iOS ed Android diventando rapidamente una delle applicazioni più scaricate; Ad oggi il numero totale di download supera il miliardo.

Il gioco è stato riproposto in molteplici versioni e piattaforme tra le principali abbiamo **Fruit Ninja Kinect** per Xbox e Windows, **Fruit Ninja THD** ottimizzato per dispositivi con Nvidia Tegra 2, **Fruit Ninja VR** per Oculus, HTC Vive e PlayStation 4 ed infine un arcade game chiamato **Fruit Ninja FX**.

### 1.1.1 Gameplay

In **Fruit Ninja** il giocatore deve affettare della frutta che viene lanciata sullo schermo, trascinando un dito sul touch screen del dispositivo. Lo scopo del gioco è quello di tagliare più frutta possibile. Vengono conferiti punti extra quando si affettano tre o più frutta con uno stesso swipe chiamate *combo*, anche l'esecuzione ripetuta di combo chiamata *blitz* conferisce punti aggiuntivi.



Figura 1.1: Logo di Fruit Ninja

**Fruit Ninja** mette a disposizione diverse modalità di gioco che si basano tutte sul gameplay appena descritto:

Figura 1.2: Esempio di alcune versioni e modalità di gioco di **Fruit Ninja**. Da sinistra verso destra abbiamo (1) Screen della prima versione originale, (2) Versione HD, (3) Ultima versione con un ritorno al passato, (4) Versione VR.

1: la frutta mancata non causa effetti negativi.

- **Classica:** Il giocatore ha a disposizione tempo illimitato e 3 vite, insieme ai frutti possono essere lanciate anche delle bombe esplosive da evitare. Ogni frutto mancato causa la perdita di una vita, se si colpisce una bomba oppure si perdonano tutte le vite il gioco terminerà. Ogni cento punti si guadagna una vita. Non ci sono limiti alla durata della partita o del punteggio se non la bravura del giocatore.
- **Arcade:** Dura 60 secondi, l'obiettivo è battere il record. Le differenze dalla modalità classica sono: il tempo è limitato, non vi sono vite<sup>1</sup> e per ogni bomba colpita il tempo rimasto viene diminuito di 10 secondi. Sono inoltre presenti banane speciali con effetti speciali che conferiscono brevi bonus quando tagliate.
- **Zen:** Nella modalità Zen, la durata della partita è di 1:30 minuti l'obiettivo è battere il record, non saranno presenti bombe e vite e si dovrà quindi unicamente cercare di affettare più frutta possibile puntando sulle combo senza l'ausilio di bonus.

Per lo sviluppo del progetto si è scelta come base la **modalità Zen**.

## 1.2 Fruit Ninja GL



Figura 1.3: Logo della software di fantasia che ha sviluppato il fnGL.

2: Nel gioco originale, mancano totalmente Luci e Collisioni tra i frutti i quali semplicemente si sovrappongono.



Figura 1.4: Alcuni screen di gioco di Fruit Ninja GL.

Il giocatore avrà a disposizione 1:30 minuti per affettare quanti più punti possibili effettuando *combo* e *blitz* per massimizzare il punteggio finale. A completare il tutto sono presenti le musiche e gli effetti originali del gioco cercando di replicare nel dettaglio anche la grafica.

### 1.2.1 Librerie e Software Utilizzati

Fruit Ninja GL è stato sviluppato utilizzando C++20 in ambiente Visual Studio ed è basato sull'ultima versione disponibile di OpenGL la 4.6; Oltre a ciò sono state impiegate diverse librerie o moduli per semplificare lo sviluppo del gioco:

- **GLFW** [3]: Libreria open-source, multi-piattaforma per OpenGL. Fornisce una semplice API per la creazione di finestre, contexts e la gestione di diverse tipologie input ed eventi.
  - **GLEW** [8]: Libreria multi-piattaforma open-source per il loading delle funzioni OpenGL. Fornisce un meccanismo moderno ed efficiente per determinare, a run-time, quale estensione di OpenGL è supportata dalla piattaforma target.
  - **GLM** [4]: Libreria matematica basata sulle specifiche<sup>3</sup> del OpenGL Shading Language (GLSL).
  - **ASSIMP** [7]: Libreria open-source che fornisce una API semplice ed unificata per la gestione di vari formati di modelli 3D, in particolare consentendone il caricamento, lettura oltre che l'esportazione.
  - **stb\_image** [6]: Libreria open-source che consente il loading/decoding da file o memoria di immagini nei formati più comuni.
  - **irrKlang** [5]: Libreria ad alto livello che consente il caricamento e l'esecuzione di numerosissimi formati audio.
  - **{fmt}** [2]: Libreria open-source per il format delle stringhe.
  - **ImGui** [1]: Libreria per il disegno di GUI interattive<sup>4</sup>.

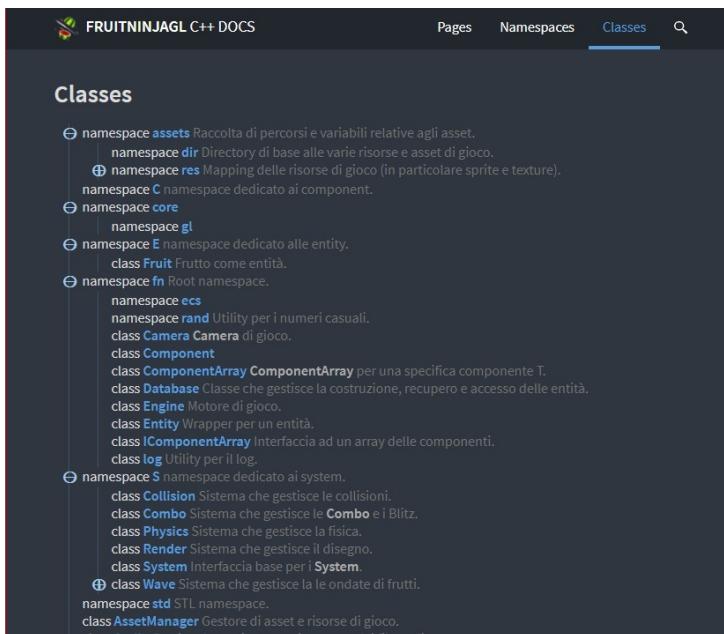
Inoltre è stato utilizzato Visual Studio 2019 per lo sviluppo, Adobe Photoshop per la creazione di sprite e l'editing di texture ed in fine Doxygen + m.css\* per la produzione della documentazione.

## 1.2.2 Documentazione

Tutto il codice è ben documentato e correlato di una documentazione prodotta dai commenti che descrive i vari moduli, classi e funzioni implementati.

3: Le funzionalità non sono però limitate al GLSL ma si estendono anche alle trasformazioni matriciali, quaternioni, data packing, random numbers, noise, etc...

4: Utilizzata nelle fasi iniziali di sviluppo più per un rapido debug e prototipazione. Rimossa nelle fasi finali del progetto.



**Figura 1.5:** Esempio della documentazione prodotta con Doxygen + m.css. Nella figura è mostrata la lista delle classi.

\* A modern, mobile friendly drop-in replacement for the stock Doxygen HTML output. <https://mcss.mosra.cz/mcss/documentation/doxygen/>

5: L'ultima versione dell'eseguibile è già stata spostata ed è attualmente presente.

### 1.2.3 Avvio dell'applicazione

Per poter avviare l'applicazione è necessario effettuare una sorta di processo di installazione<sup>5</sup>.

Una volta compilata l'applicazione l'eseguibile generato va spostato all'interno della cartella `./FruitNinjaGL/FruitNinjaGL/` ovvero quella dove sono presenti i file di progetto VisualStudio; Tale operazione è fondamentale per accedere ai file `.dll` per il linking delle librerie e alla cartella `./res` contenente tutti gli asset e risorse di gioco i cui percorsi sono relativi alla workspace del progetto.

In questo capitolo si descriverà nel dettaglio il progetto d'esame, effettuando prima una breve overview sulle principali caratteristiche e scelte implementative ed architettoniche per poi successivamente passare ai vari dettagli implementativi.

## 2.1 Overview

Fruit Ninja GL è basato in OpenGL 4.6 Core Profile utilizzando la libreria GLFW 3.3.2 per la creazione del contesto, finestra e gestione di input ed eventi. L'applicazione è stata sviluppata in C++20 in ambiente Visual Studio<sup>7</sup> 2019.

### Simile all'originale

Durante lo sviluppo si è cercati di rimanere il più possibile fedeli al gioco nella sua versione originale rilasciata nel 2010. Musiche, suoni, font e sfondi sono liberamente disponibili in rete mentre gli sprite ed elementi della GUI sono stati riprodotti utilizzando photoshop\*.

I modelli 3D dei frutti sia nella loro versione “intera” che “affettata” sono presi da uno dei tanti Fruit pack disponibili in rete.

### Entity Component System

Lo sviluppo delle logiche di gioco è stato basato sull'Entity Component System, un pattern architettonale molto comune nel game development<sup>8</sup>

### Astrazione delle primitive

Tutte le principali primitive grafiche (di OpenGL e non) come *texture*, *mesh*, *model*, *shader* o *audio* sono state astratte in specifiche classi per rendere più semplice, immediato e conveniente il loro utilizzo.

### VAOs, VBOs, Vertex e Shaders

Il rendering dei vari oggetti è effettuato ricorrendo ai vari buffer (Vertex Array Object, Vertex Buffer Array, Index Buffer Array) e Shaders (Vertex e Fragment) di OpenGL. Sono stati implementati diversi shader a seconda dell'elemento grafico o di come lo si vuole disegnare nel framebuffer.

Dopo una iniziale difficoltà l'utilizzo dei buffer unito con gli shader rende lo sviluppo più semplice e flessibile.

2.1 Overview . . . . .	5
2.2 Descrizione Architetturale . . . . .	6
2.3 Entity Component System . . . . .	9
2.4 OpenGL . . . . .	18

7: Il codice è compilabile in ambiente linux utilizzando un altro compilatore.



Figura 2.1: Esempio di sprite realizzato imitando lo stile di Fruit Ninja.

8: Utilizzabile in molti motori grafici come Unity, Unreal Engine o il Cryengine.

Si è scelto questo approccio invece che la semplice *direct API mode* (`glBegin()`, `glEnd()`, ...) perché quest'ultima è deprecata (e dunque nemmeno disponibile nel Core Profile), molto più lenta oltre che poco “compatibile” con il loading di asset e modelli 3D complessi.

\* Tutti i file .psd creati ed utilizzati sono inclusi assieme agli asset.

### Engine a Stati

L'engine di gioco è basato sul Game State 'Stack' un altro pattern architettonico molto diffuso per la gestione dei vari possibili stati di gioco (Loading, playng, pause, game over, ...).

## 2.2 Descrizione Architetturale

Architettonicamente Fruit Ninja GL è strutturato in cinque *macro-moduli* ognuno dei quali si occupa di gestire una parte dell'applicazione.

- ▶ core/ classi finalizzate alla gestione dei dettagli a basso livello di OpenGL e non.
- ▶ ecs/ implementazione dell'ECS con componenti, sistemi e relative classi di gestione delle entità.
- ▶ engine/ implementazione del motore di gioco a stati.
- ▶ logic/ dettagli relativi al gameplay e agli elementi di gioco.
- ▶ utl/ classi e funzioni di utility o debug.

### 2.2.1 Modulo core/

Il modulo core contiene tutta una serie di classi e funzioni con il principale scopo di astrarre il più possibile le funzionalità a basso livello messe a disposizione da OpenGL e da altre librerie. L'idea è quella di creare una serie di API semplici su cui si baserà l'intera applicazione.

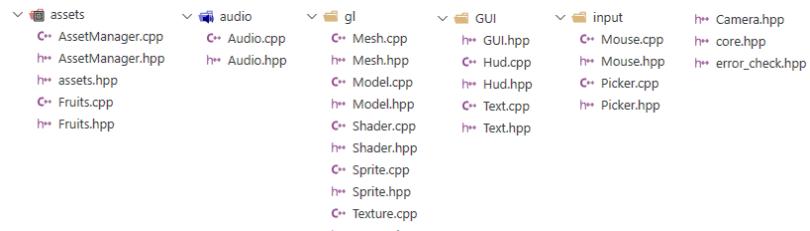


Figura 2.2: Sotto-moduli e relativi file .hpp e .cpp che fanno parte di core.

Come è possibile vedere in figura 2.2 core è strutturato in sotto-moduli ognuno con funzionalità specifiche; nella root principale invece è presente:

- ▶ Camera.hpp una semplice classe che gestisce la camera posizionandola nello spazio e costruendo di conseguenza la *projection matrix* (prospettica) e la *view matrix*.
- ▶ error\_check.hpp contiene macro e callback per il debug ed il checking di errori di OpenGL.

#### Sotto-modulo asset

Dedicato alla gestione di asset e risorse grafiche/multimediali di gioco. Oltre alla semplice definizione di cartelle e percorsi (asset.hpp) o alla strutturazione di asset composti (fruit.hpp) nel modulo si implementa anche un *gestore degli asset* (AssetManager.hpp) che utilizzando il pattern flyweight consente di caricare e riutilizzare in maniera veloce e dinamica le varie risorse di gioco fornendo lo stesso puntatore ogni volta quella risorsa è richiesta.

Figura 2.3: Diagramma UML della classe AssetManager

Tutti gli asset inoltre hanno costruttori di copia e assegnazione disabilitati per evitare duplicazione di risorse.

```
template<typename ...Args>
inline constexpr ModelSP AssetManager::loadModel(const fs::path&
                                                ↪ filepath, Args && ...args)
{
    // Verifico che il modello non sia in cache
    auto f = filepath.string();
    if (AssetManager::s_modelCache.find(f) != AssetManager::
        ↪ s_modelCache.end())
        return AssetManager::s_modelCache[f];

    auto model = std::make_shared<Model>(filepath, std::forward<
                                            ↪ Args>(args)...);
    AssetManager::s_modelCache[f] = model;
    return model;
}
```

**Listing 2.1:** Esempio di metodo utilizzata dal manager per il caricamento dei Modelli.

### Sotto-modulo audio

Dedicato alla gestione delle funzionalità audio sia 2D che 3D dell'applicazione ricorrendo alla libreria irrKlang [5]. La classe Audio è considerata come un asset di gioco, si occupa del caricamento nonché dell'esecuzione dei file audio e fornisce metodi statici per la gestione generale<sup>9</sup>.

<sup>9</sup>: Ad esempio per interrompere tutti i suoni in esecuzione o semplicemente abbassare il volume.

### Sotto-modulo gl

Dedicato all'astrazione delle principali primitive di OpenGL in modo da consentire un uso facile e componibile.

- ▶ **Texture:** gestisce il loading, creazione e binding delle texture. Il caricamento del file in sè avviene tramite la libreria stb\_image [6] mentre la creazione e binding openGL è effettuato al caricamento supporta anche eventuali parametri. La classe è anche un asset.

```
Texture::Parameteri parm = {
    { GL_TEXTURE_WRAP_S, GL_MIRRORED_REPEAT },
    { GL_TEXTURE_WRAP_T, GL_MIRRORED_REPEAT },
    { GL_TEXTURE_MIN_FILTER, GL_NEAREST },
    { GL_TEXTURE_MAG_FILTER, GL_NEAREST },
};

this->m_texture = AssetManager::loadTexture(filepath, parm);
```

- ▶ **Mesh:** gestisce una singola Mesh + texture. I dati sono rappresentati utilizzando le strutture e buffer di OpenGL. La classe definisce anche la struttura e layout dei vertici<sup>10</sup>.
- ▶ **Model:** gestisce un Modello, ovvero una composizione di più mesh. Il caricamento dei modelli dal disco avviene utilizzando la libreria ASSIMP [7].

<sup>10</sup>: Tale struttura è quella che poi è utilizzata in ogni elemento 3D.

11: Uno Shader è composto da un vertex shader e da un fragment shader.

- Shader: gestisce gli shader<sup>11</sup> di OpenGL, la classe ne consente il caricamento del sorgente .glsl, la compilazione, il linking ed in fine l'attivazione o disattivazione.

#### Sotto-modulo input

Dedicato alla gestione degli input utente.

- Mouse, singleton tramite il quale ogni altro componente può sapere quale pulsante è stato premuto e le ultime posizioni assunte dal mouse durante il trascinamento nella finestra creata da GLFW.
- Picker, classe speciale che consente di effettuare il picking delle entità disegnate sul framebuffer.

#### 2.2.2 Modulo ecs/

Il modulo ecs implementa il pattern architettonico Entity Component System.

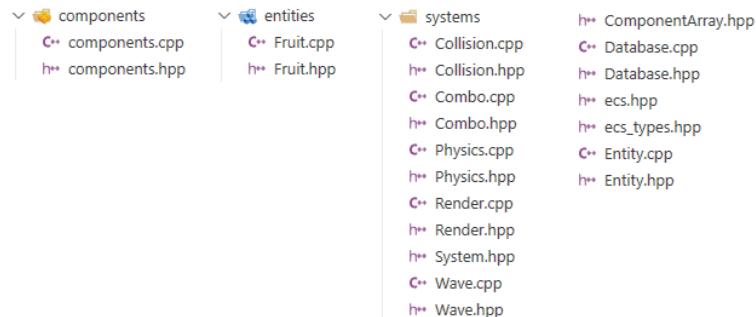


Figura 2.4: Sotto-moduli e relativi file .hpp e .cpp che fanno parte di ecs.

Il funzionamento, le classi e eventuali dettagli implementativi verranno descritti nella sezione [2.3](#).

#### 2.2.3 Modulo engine/

Il modulo engine implementa il pattern architettonico Game state “stack” per la gestione degli stati di gioco ed il game loop.

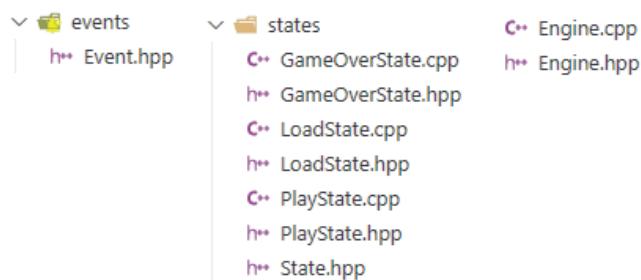


Figura 2.5: Sotto-moduli e relativi file .hpp e .cpp che fanno parte di engine.

La classe principale per il funzionamento di tutta l'applicazione è Engine ovvero il motore di gioco. Engine è un singleton che ha il compito di inizializzare l'intera applicazione (moduli, sotto-moduli, contesti e librerie) per poi avviare il game loop sullo stato correntemente in cima allo stack degli stati.

```

void Engine::loop() {
    auto now = core::seconds(glfwGetTime());
    this->delta_t = now - last_t;
    last_t = now; // Determino il tempo trascorso
    Mouse::update(this->delta_t); // Aggiorno Mouse

    if (_states.empty()) return;
    auto& state = *_states.top(); // Prendo lo stato corrente

    glfwPollEvents(); // Handle inputs
    state.handleInputs();
    state.update(this->delta_t); // Update
    state.render(); // Render
    state.handleEvents(); // Handle Events
}

```

La classe non conosce nulla riguardo il gioco in se ma si limita semplicemente ad aggiornare lo stato correntemente attivo che conterrà tutte le logiche di funzionamento.

#### 2.2.4 Modulo ut1/

Il modulo `utl` contiene delle semplici funzioni di utility per il print, logging (utilizzando la libreria `{fmt}`) e la generazione di numeri e/o vettori casuali (utilizzando la libreria `glm`).

### 2.3 Entity Component System

Entity-component-system (ECS) è un pattern architetturale estremamente diffuso nel game development, incorporato nativamente in molti dei game engine più comuni vantando elevate prestazioni, grande flessibilità e semplicità di progettazione.

Come dice il nome, tre sono i concetti fondamentali:

- ▶ **Entity:** Sono elementi “general porpouse” del gioco, possono essere qualsiasi cosa<sup>12</sup> ogni entity è identificata da un id univoco.
- ▶ **Components:** Sono i raw data rappresentanti un singolo aspetto/caratteristica di un’entità<sup>13</sup>. Ad una singola entità è possibile associare più componenti. Anch’essi sono identificati univocamente con un id.
- ▶ **System:** Contengono la logica di un singolo aspetto atomico del gameplay<sup>14</sup>, e processano tutte le entità che possiedono le componenti richieste dal sistema. La ricerca delle entità compatibili è detta *query*.

L’ECS segue il *composition over inheritance principle* ovvero si preferisce definire le caratteristiche degli oggetti (entity) attraverso la composizione (components) e non mediante l’ereditarietà. Ciò conferisce

**Listing 2.2:** Funzione che esegue un singolo loop di gioco. Una volta determinato il tempo trascorso dall’ultima iterazione in ordine: si effettua il polling degli eventi (glfw), si gestisce l’input, si aggiorna lo stato interno, si effettua il rendering ed in fine si gestiscono gli eventi di gioco.

L’idea di utilizzare questo pattern per strutturare fnGL è stato più uno sfizio personale che una vera necessità.

12: Es. nemici, sprite, particelle, effetti, animazioni, ...

13: Es. posizione, velocità, mesh, AABB, AI, tags, stato, ...

14: Es. Gravità, Collisioni, Spawn, Movimento, Rendering, ...

Si evitano così anche alcuni problemi tipici dell’OOP come le gerarchie profonde, l’eredità multipla, la flessibilità, e la generalizzabilità.

un'enorme flessibilità nello sviluppo dal momento che il comportamento delle entità (systems) può facilmente essere cambiato a runtime semplicemente rimuovendo o modificando i suoi componenti.

Un'ulteriore punto chiave dell'ECS è il **data-oriented design**, ovvero un approccio di strutturazione del codice orientato ai dati (piuttosto che agli oggetti) al fine di ottimizzare quanto più possibile l'utilizzo della cache della CPU focalizzandosi sul layout dei dati, scomposizione degli oggetti, località dei dati etc...

### 2.3.1 Database

15: Anche chiamato world, universe o entity manager.

Per completare il funzionamento del pattern è in realtà necessario un ulteriore elemento il **Database**<sup>15</sup>. Come è facile intuire dal nome, il database ha il compito di creare le entità fornendo loro identificativi univoci, tenere traccia dei componenti assegnati ad esse ed in fine eseguire query e ricerche per i sistemi. Il tutto cercando di mantenere un approccio data-oriented.

L'implementazione dell'ECS di Fruit Ninja GL è fortemente ispirato (per API e filosofia) ai framework più famosi in particolare ad *entt* utilizzato per *minecraft*, *ecsX* e *flecs*. Tutte queste implementazioni fanno forte uso di template, variatics, constexpr e altre funzionalità avanzate del C++.

Nelle seguenti sezione si procederà a descrivere i vari elementi dell'ECS implementati nel framework di fnGL. Infine nella sezione [2.3.5](#) si mostreranno alcuni esempi di utilizzo.

### 2.3.2 Entità

16: Più database sono possibili ma in quel caso è necessario non mischiare le entità create fra le varie istanze.

In fnGL una entity è un `fn::Eid` ovvero un identificativo univoco che altri non è che un `unsigned int`. Tutte le entità devono essere generate da un database<sup>16</sup> e non possono essere create in altri modi.

```
Database database();
fn::Eid e1 = database.create<fn::Eid>();           // (1)
E::Entity e2 = database.create<E::Entity>();         // (2)
E::Entity e3 = database.create();                     // (3)
```

La creazione avviene semplicemente utilizzando il metodo `create` tramite la quale come un template è possibile specificare come deve essere restituita l'entità:

1. Crea l'entità e restituisce semplicemente il suo identificativo.
2. Crea l'entità ma restituisce un oggetto `E::Entity` che banalmente contiene internamente il `fn::Eid`.
3. Uguale al metodo (2), `E::Entity` è la tipologia di default.

### E::Entity

Dal momento che l'Eid è un semplice intero, senza il suo database di appartenenza risulta ovviamente poco utile. Per questo motivo, per comodità, è stata creata la classe E::Entity ovvero un semplice wrapper che contiene sia l'identificativo che un puntatore al database.

Tramite un E::Entity sono richiamabili tutti i metodi del database che contiene senza ovviamente la necessità di dover specificare anche l'Eid.

### E::Fruit

La classe E::Fruit deriva da E::Entity e ne condivide principi e funzionamento di base. L'unica differenza è che mette a disposizione una serie di metodi ed attributi statici di utility per la manipolazione di questo tipo di entità sempre per mezzo del database.

Si tratta quindi di una semplice classe che racchiude alcune logiche del gioco.

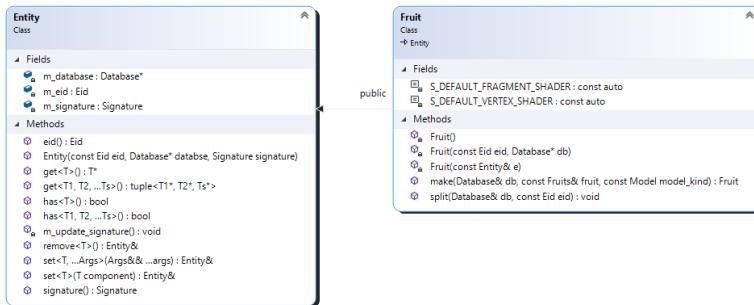
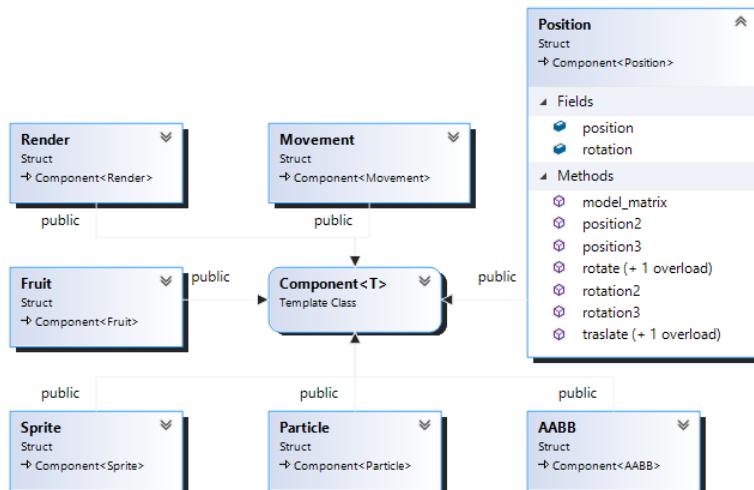


Figura 2.6: Diagramma UML delle classi Entity e Fruit

### 2.3.3 Componenti

In fnGL un componente è una qualsiasi struct/class che estende `struct Component<T>` ed è inizializzabile tramite initialization list. Le componenti, per garantire la località dei dati dovrebbero essere dei POCO\* il cui contenuto è il dato stesso e non un puntatore o un reference.



Tutte le componenti si trovano sotto il namespace C.

Figura 2.7: Diagramma UML di tutte le componenti implementate in fnGL. Come è possibile vedere dal dettaglio di C::Position oltre ai dati in realtà possono essere presenti anche dei semplici metodi per la loro manipolazione.

\* Cioè dei Plain Old C++ object ovvero delle semplici classi o struct che memorizzando l'value di tipi primitivi o aggregati di primitivi.

17: Il valore  $N$  deve essere almeno pari al numero massimo di components; in fnGL si utilizza 32.

**Listing 2.3:** Tutti i Cid delle varie componenti sono generati durante la compilazione in contemporanea con la risoluzione de template da parte del compilatore.

### fn::Cid e fn::Signature

Ogni component è identificata da un fn::Cid anche in questo caso un intero ma memorizzato per convenienza come un std::bitset<N> nel quale ogni bit è associato una componente.<sup>17</sup> In questo modo, i Cid possono essere usati in or o in and per semplificare di molto le operazioni del database. La riduzione in or di uno o più Cid delle componenti di un'entità è chiamata fn::Signature (o firma).

```
namespace C {
    struct Position : public Component<Position> { ... };
    // Position.Cid --> 0000 0000 0000 0001
    struct Movement : public Component<Movement> { ... };
    // Movement.Cid --> 0000 0000 0000 0010
    ...
    struct AABB : public Component<AABB> { ... };
    // AABB.Cid --> 0000 0000 0100 0000
}
```

Tutti i fn::Cid sono calcolati automaticamente a compilation-time insieme alla definizione del componente; la signature (oltre che a runtime) può essere calcolata a compilation-time ricorrendo alla template variable Sign<...Ts>.

**Listing 2.4:** Anche le Sign sono generate durante la compilazione. In ogni caso è comunque possibile a runtime effettuare esplicitamente operazioni manipolazione dei singoli Cid.

```
fn::Sign<C::Position> // --> 0000 0000 0000 0001
auto a = fn::Sign<C::Position, C::Movement>
// a --> 0000 0000 0000 0011
auto b = fn::Sign<C::AABB, C::Movement>
// b --> 0000 0000 0100 0010
auto c = fn::Sign<C::AABB, C::Movement, C::Position, C::Fruit>
// c --> 0000 0000 0100 1011
```

### C::Position

La componente C::Position contiene tutte le informazioni necessarie all'individuazione di un'entità nello spazio. Oltre a ciò sono anche implementati dei metodi di utility come translate(const glm::vec3&) e rotate(const glm::vec3&).

```
struct Position : public fn::Component<Position> {
    glm::vec3 position;           // Vettore posizione dell'entità
    glm::vec3 rotation;          // Rotazione (angoli di eulero)
    ...
};
```

Le coordinate sono espresse secondo il sistema mondo mentre la rotazione è espressa utilizzando gli angoli di Eulero<sup>18</sup> riferiti ai tre assi canonici.

### C::Movement

La componente C::Movement contiene tutte le informazioni necessarie a calcolare il movimento<sup>19</sup> di un'entità nello spazio. Oltre a ciò sono anche implementati dei metodi di utility per la manipolazione basilare

18: L'utilizzo di quaternioni non è stato necessario.

19: Il vettore accelerazione è stato volutamente trascurato.

dei dati come `accelerate(const glm::vec3&)` che semplicemente incrementa la velocità.

```
struct Movement : public fn::Component<Movement> {
    glm::vec3 velocity; // Vettore velocità dell'entità
    glm::vec3 spin; // velocità angolare (angoli di eulero)
    ...
};
```

La velocità è espressa secondo il sistema mondo e la velocità angolare è utilizzata gli angoli di Eulero riferiti ai tre assi canonici.

#### C::Render

La componente C::Render contiene tutte le informazioni necessarie disegnare un'entità tridimensionale.

```
struct Render : public fn::Component<Render> {
    ShaderSP shader;
    ModelSP model;
};
```

Dove `shader` e `model` sono due `std::shared_ptr` il primo allo shader che si vuole utilizzare per disegnare e l'altro al modello da disegnare.

#### C::Fruit

La componente C::Fruit contiene tutte le informazioni relative ad un frutto.

```
struct Fruit : public fn::Component<Fruit> {
    Fruits fruit;
    Fruits::Model model_kind;
};
```

Dove `fruit` è il frutto in questione (uno fra i possibili a disposizione) e `model_kind` è un enum che ne specifica la tipologia che può essere:

- ▶ `whole` per indicare il frutto intero non affettato;
- ▶ `half_front` o `half_back` per indicare una delle due metà del frutto affettato.

#### C::Sprite

La componente C::Sprite contiene tutte le informazioni necessarie disegnare un'entità bidimensionale.

```
struct Sprite : public fn::Component<Sprite> {
    SpriteSP sprite;
};
```

Dove `sprite` è uno `std::shared_ptr` allo sprite<sup>20</sup> da disegnare.

20: Uno sprite non è altro che una mesh bidimensionale quadrata a cui è stata applicata una texture.

**C::Particle**

La componente C::Particle contiene tutte le informazioni necessarie disegnare un'entità temporanea, che appare e scompare con un'animazione di fade.

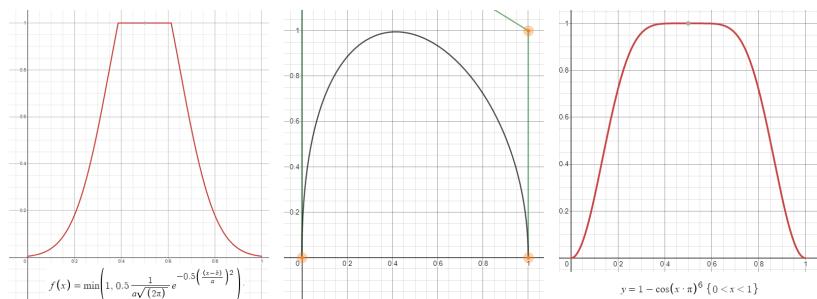
```
struct Particle : public fn::Component<Particle> {
    core::seconds lifetime;
    core::seconds elapsed;
    std::function<float(float)> interpolator;
};
```

21:  $f$  deve essere una funzione del tipo  $f : [0, 1] \mapsto [0, 1]$

Dove `lifetime` e `elapsed` sono rispettivamente il tempo di vita della particella ed il tempo trascorso dalla sua creazione, quando `elapsed > lifetime` l'entità dovrebbe essere distrutta. L'attributo `interpolator` è invece un puntatore ad una funzione  $f$  interpolante<sup>21</sup> da utilizzare per calcolare l'animazione di fade al trascorrere del tempo.

Nella figura 2.8 sono mostrati i grafici di tre delle funzioni  $f$  implementate:

- ▶ A base gaussiana - `asset::anim::gaussian(const float t);`
- ▶ Cubica di Bézier - `asset::anim::bezier(const float t);`
- ▶ A base coseno - `asset::anim::cosine(const float t);`



**Figura 2.8:** Grafici delle tre funzioni interpolabili disponibili negli asset ed utilizzate all'interno del gioco. La scelta di una piuttosto che dell'altra dipende dalla velocità di animazione ricercata nelle due fasi di fade-in e fade-out.

**C::AABB**

La componente C::AABB contiene l'axis-aligned bounding box (AABB) dell'entità necessario al calcolo delle collisioni.

```
struct AABB : public fn::Component<AABB> {
    glm::mat2x3 box;
};
```

Dove `box` è una matrice le cui colonne sono le coordinate in sistema model dei due estremi (top-left e bottom-right) necessari a definire il box. Si è scelto di utilizzare una matrice per facilitare operazioni di trasformazione delle coordinate.

**2.3.4 Sistemi**

Tutti system si trovano nel namespace `S`.

In fnGL un system è una qualsiasi struct/class che estende la classe base `System`. I sistemi implementano tutte le logiche di gestione, aggiornamento ed eventualmente rendering delle entità, ognuno di essi è applicato a tutte le entità che rispettano i requisiti di componenti del sistema.

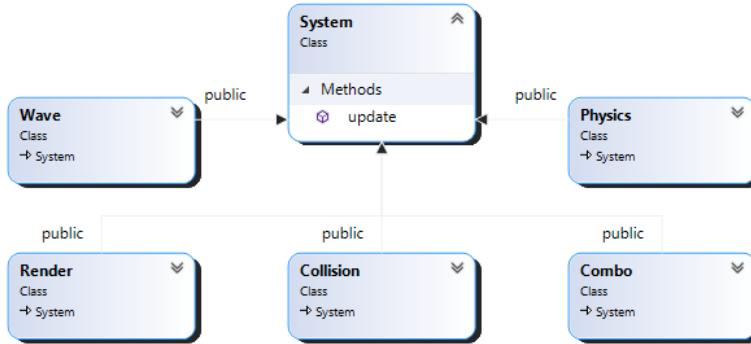


Figura 2.9: Diagramma UML di tutti i sistemi implementati in fnGL.

**Esempio 2.3.1** Il sistema `S::Physics` gestisce la fisica del gioco relativa al movimento delle entità applicando anche l'accelerazione gravitazionale  $g$ . Per funzionare è necessario:

- ▶ la componente `C::Movement` per calcolare i nuovi vettori velocità applicando  $g$ ;
- ▶ la componente `C::Position` per calcolare la nuova posizione.

Dunque, ad ogni update `S::Physics` effettua una query per determinare tutte quelle entità con entrambe le componenti<sup>22</sup> e applica a ciascuna di esse la sua logica interna di aggiornamento.

22: Ovvero almeno con la signature `fn::Sign<C::Position, C::Movement>`.

### `S::Physics`

Il sistema `S::Physics` consente di gestire lo stato fisico delle entità aggiornandone posizione, rotazione, velocità e rendendoli soggetti alla forza gravitazionale. La classe ha anche il compito di eliminare quelle entità che a causa della gravità escono dallo schermo.

I componenti coinvolti dal sistema sono: `C::Position` e `C::Movement`.

### `S::Collision`

Il sistema `S::Collision` si occupa di gestire ciò che avviene quando due oggetti collidono. Implementando sia la rilevazione che la risposta alla collisione. I componenti coinvolti dal sistema sono: `C::Position`, `C::Movement`,

`C::AABB` e `C::Fruit`. Nel gioco originale le collisioni non erano gestite in alcun modo.

Le collisioni sono calcolate soltanto per le entità dotate delle precedenti quattro componenti ristrette ulteriormente ai soli frutti interi<sup>23</sup>.

23: Per scelta i frutti già tagliati sono intangibili.

### Rilevamento delle collisioni

Il rilevamento delle collisioni avviene utilizzando un approccio ibrido fra OOB e la semplice sfera.

1. Al momento del caricamento del modello, viene calcolata anche la AABB relativa alle sue coordinate model.
2. Durante il check delle collisioni al AABB viene trasformata in OOB applicando le stesse trasformazioni di modellazione dell'entità al box.

3. A partire da questo nuovo box viene calcolato il raggio della sfera centrato nel centro tangente al suo lato più lungo. Il raggio individua una sfera che sarà utilizzata per la rilevazione.

Si è scelto questo approccio perché semplice e si adatta bene con la quasi totalità dei frutti i quali sono piuttosto tondeggianti.

#### *Risposta alle collisioni*

Quando due frutti collidono, la risposta alla collisione cerca di imitare gli urti elastici della fisica classica. In particolare, si tiene conto delle velocità iniziali e della massa dei frutti per calcolare le velocità relative a seguito dell'impatto. La formula utilizzata è:

$$\begin{aligned}\mathbf{v}'_1 &= \mathbf{v}_1 - \frac{2m_2}{m_1 + m_2} \frac{\langle \mathbf{v}_1 - \mathbf{v}_2, \mathbf{x}_1 - \mathbf{x}_2 \rangle}{\|\mathbf{x}_1 - \mathbf{x}_2\|^2} (\mathbf{x}_1 - \mathbf{x}_2), \\ \mathbf{v}'_2 &= \mathbf{v}_2 - \frac{2m_1}{m_1 + m_2} \frac{\langle \mathbf{v}_2 - \mathbf{v}_1, \mathbf{x}_2 - \mathbf{x}_1 \rangle}{\|\mathbf{x}_2 - \mathbf{x}_1\|^2} (\mathbf{x}_2 - \mathbf{x}_1)\end{aligned}\quad (2.1)$$

Dove  $\mathbf{x}_1$  e  $\mathbf{x}_2$  sono i centri dei due oggetti al momento dell'impatto,  $\mathbf{v}_1$  e  $\mathbf{v}_2$  le velocità dei frutti ed in fine  $m_1$  e  $m_2$  le loro masse.

#### **S::Wave**

Il sistema S::Wave gestisce il lancio di frutti, raggruppandoli in ondate di vario tipo e stabilendo i tempi di lancio. I componenti coinvolti dal sistema sono: C::Position, C::Movement e C::Fruit.

Tutti i frutti compaiono sempre dal lato basso dello schermo.

Sono stati implementati quattro “pattern di lancio”:

1. WaveType::RANDOM: nel quale le componenti dell'entità sono tutte generate casualmente.
2. WaveType::LEFT\_TO\_RIGHT o WaveType::RIGHT\_TO\_LEFT: nel quale vengono lanciati in sequenza da sinistra verso destra (o viceversa) più frutti dello stesso tipo.
3. WaveType::SPOT: nel quale avvengono lanci veloci di frutti dello stesso tipo da posizioni casuali.

#### **S::Combo**

Il sistema S::Combo gestisce le combo e i blitz quando si affettano più frutti in rapida successione con un solo swipe. I componenti coinvolti dal sistema sono C::Sprite e C::Particle.

#### **S::Render**

Il sistema S::Render disegna le entità tridimensionali e bidimensionali. I componenti coinvolti dal sistema sono: C::Position e C::Render per gli oggetti tridimensionali e C::Sprite per quelli bidimensionali.

Il rendering avviene in due fasi, prima si disegnano i modelli 3D e poi quelli 2D.

### 2.3.5 Esempio di utilizzo

Creiamo ad esempio tre entità e1, e2 ed e3 a partire da una istanza di un `fn::Database` db nel quale i vari componenti sono già registrati.

```
fn::Entity e1 = db.create();
fn::Entity e2 = db.create();
fn::Entity e3 = db.create();
```

Le tre entità adesso sono vuote e non hanno associato alcuna componente.

#### Set delle componenti

Una componente può essere aggiunta ad una `E::Entity` in due modi tramite il database ed il suo `Eid` oppure tramite i suoi metodi.

```
// set attraverso E::Entity
e1.set<C::Position>({}); //Costruttore di default
e1.set<C::Movement>({}); //Init list
    .velocity=glm::vec3(1.5f, 3.6f, 9.1f),
    .spin=glm::vec3(3.14f, -1.21f, -0.1f),
});
e1.set<C::Fruit>({ ... });

e2.set<C::Position>({}); //Costruttore di default

// set attraverso fn::Database
db.set<C::Position>(e3.Eid, {
    .velocity=glm::vec3(1.5f, 3.6f, 9.1f),
    .spin=glm::vec3(3.14f, -1.21f, -0.1f),
});
```

**Listing 2.5:** Il set può avvenire passando una initialization list, i parametri da inoltrare al costruttore o un oggetto componente.

#### Get delle componenti

Con le stesse modalità del set è anche possibile accedere alle componenti usando la funzione `get`. Il valore ritornato è sempre un puntatore alla componente. In aggiunta è possibile anche accedere a più componenti insieme, il valore ritornato in questo caso è una `std::tuple` di puntatori.

```
// get di una singola componente
auto* p = e1.get<C::Position>();

// get di più componenti classico
auto t = e1.get<C::Position, C::Movement, C::Fruit>();
std::get<1>(t)->accelerate(...); //--> C::Movement

// get di più componenti con un-pack (c++17)
auto [p, m, f] = e1.get<C::Position, C::Movement, C::Fruit>();
m->accelerate(...); //--> C::Movement
```

**Listing 2.6:** Utilizzando le funzionalità di un-packing introdotte in c++17 l'utilizzo delle tuple risulta molto meno verboso e conveniente. Di fatto l'accesso "classico" non è mai utilizzato.

## Controllo delle componenti

Con le stesse modalità del get si può controllare se un'entità ha una o più componenti utilizzando il metodo has.

```
bool a = e1.has<C::Movement, C::Fruit>(); //true
bool a = e2.has<C::Sprite>(); //false
```

## Query

Le query del database possono essere effettuate solo su un oggetto database e in due modi:

- ▶ Usando il metodo Database::having<> che semplicemente ritorna un vettore<sup>24</sup> di eid che rispettano la query.
- ▶ Usando il metodo Database::for\_each<> in combinazione con una funzione lambda che viene applicata ad ogni entità che rispetta la query.

**24:** questo metodo in realtà non è mai usato nella pratica.

**Listing 2.7:** I vari metodi sono in realtà tutti equivalenti dal punto di vista funzionale, da quello delle performance invece l'iterazione su un sottoinsieme di componenti è sicuramente la scelta migliore.

Un'ulteriore vantaggio di questo approccio è la possibilità di creare lambda (ad esempio all'interno di eventi di gioco) da applicare alle entità.

```
// (a) lambda sugli eid
db.for_each<C::Movement, C::Fruit>([](fn::Eid e){
    // fai qualcosa con e
});

// (b) lambda sugli E::Entity
db.for_each<C::Fruit>([](E::Entity& e){
    // fai qualcosa con e
});

// (c) lambda sui componenti
db.for_each<C::Position, C::Movement>([](C::Position& p,
                                            C::Movement& m){
    // fai qualcosa con p ed m
});

// (c) lambda sui componenti + E::Entity
db.for_each<C::Sprite>([](E::Entity& e, C::Sprite& s){
    // fai qualcosa con e ed s
});
```

## 2.4 OpenGL

Questa sezione è dedicata alle classi, funzioni e metodi che sono stati utilizzati per astrarre le primitive di OpenGL combinandole in un API che ne consenta un uso più semplice, componibile e di alto livello.

### 2.4.1 Texture

La classe Texture gestisce il caricamento, binding e generazione delle texture 2D<sup>25</sup> in fnGL. Ogni istanza della classe rappresenta una texture caricata ed è dunque considerata come un asset e gestita dall'AssetManager.

**25:** Le texture 1D o 3D non sono state supportate.

```
Texture(const fs::path& texpath,
        const Texture::Type type = Texture::Type::diffuse,
        const Texture::Parameteri& parameteri = {});
```

Il costruttore è molto semplice e richiede del caso minimo soltanto il percorso `texpath` al file immagine contenente la texture; opzionalmente è possibile specificare anche la tipologia di texture (diffuse o specular) e i parametri della texture che saranno passati a `glTexParameter`.

L'argomento opzionale `Texture::Parameteri` `parameteri` del costruttore è un alias per `std::unordered_map<GLint, GLint>` utilizzabile per specificare le coppie (key, value) dei parametri. In caso di map vuota verranno utilizzati:

```
Texture::Parameteri m_parameteri = {
    { GL_TEXTURE_WRAP_S, GL_MIRRORED_REPEAT },
    { GL_TEXTURE_WRAP_T, GL_MIRRORED_REPEAT },
    { GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR },
    { GL_TEXTURE_MAG_FILTER, GL_LINEAR },
};
```

### Caricamento e creazione delle texture

Tutto il lavoro di caricamento delle texture avviene in fase di costruzione dell'oggetto da parte del metodo `protected Texture::load()` il cui codice è mostrato nel listing sottostante.

```
void Texture::load(){
    stbi_set_flip_vertically_on_load(true);

    auto f = this->m_path.string();
    unsigned char* image = stbi_load(f.c_str(), &m_width,
                                      &m_height, 0, STBI_rgb_alpha);
    if (image == nullptr)
        fn::log::error("[ERRORE] Errore durante il load della \
                      texture {} <image=nullptr>\n", f);

    GL_CHECK(glGenTextures(1, &m_id));
    // Eseguo il binding della texture
    GL_CHECK(glBindTexture(GL_TEXTURE_2D, m_id));
    // Setto i parametri
    for (auto& [key, value] : m_parameteri) {
        GL_CHECK(glTexParameter(GL_TEXTURE_2D, key, value));
    }
    // Carico i dati della texture
    GL_CHECK(glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, m_width,
                         m_height, 0, GL_RGBA,
                         GL_UNSIGNED_BYTE, image));
    // Genero eventuali mipmap
    GL_CHECK(glGenerateMipmap(GL_TEXTURE_2D));
    // Pulizia e free
    GL_CHECK(glBindTexture(GL_TEXTURE_2D, 0));
    stbi_image_free(image);
```

**Listing 2.8:** Funzione utilizzata per il caricamento, generazione e creazione delle texture in OpenGL.

```
}
```

La prima cosa che ovviamente è necessario fare è caricare i dati della texture dal file specificato che avviene utilizzando la image-loading library `stb_image.h` [6]. Successivamente, è necessario chiedere ad OpenGL di generare una nuova texture, attivarla eseguendo il binding ed infine settare i parametri.

26: Fino a questo momento i dati si trovano nella memoria ram, con il seguente caricamento verranno trasferiti nella vram.

27: Ad esempio per le luci, ombre, bumps, colori.

**Listing 2.9:** Nel vertice si memorizzano la posizione in coordinate model, la normale al vertice per l'illuminazione e le coordinate texture per appunto applicare la texture.

28: In questo modo si evita di memorizzare più volte uno stesso vertice quando si disegnano facce adiacenti ed inoltre è possibile renderizzare sottoinsiemi o definire i lati delle facce a seconda dell'ordine degli indici.

### 2.4.2 Mesh

La classe Mesh astrae il concetto di mesh, ovvero rappresenta un reticollo poligonale che definisce un oggetto nello spazio, la rappresentazione avviene mediante vertici, indici e texture. Nei vertici sono contenuti i dati dell'oggetto mentre negli indici contengono l'adjacency information cioè come i vertici sono connessi per formare gli spigoli e le facce della mesh.

#### I Vertex

Per poter creare una mesh è prima necessario definire i vertici. In OpenGL un vertice è molto più di un punto nello spazio, esso può infatti contenere informazioni aggiuntive quali, nomali, coordinate texture, colori, tangent, bitangent etc... necessarie rendering avanzato<sup>27</sup> degli oggetti.

L'insieme di tali informazioni insieme al loro ordine e offset all'interno del vertici è detto **Vertex Layout** che ovviamente può essere dinamico e deve essere specificato ad OpenGL. In fnGL il layout è molto semplice:

```
struct Vertex{
    glm::vec3 position; // Vettore posizione del vertice in
                        // coordinate model.
    glm::vec3 normal; // Vettore normale al vertice
    glm::vec2 texCoords; // Coordinate del vertice sulla texture
};
```

#### Adjacency information

I vertici da soli non sono sufficienti alla rappresentazione di una mesh e devono essere affiancati dall'adjacency information. Si tratta di una serie di indici che specificano terne (o quaterne) di vertici che formano le varie facce<sup>28</sup> della mesh.

#### Caricamento e creazione delle mesh

Il caricamento delle mesh avviene con le stesse modalità descritte per le texture nella sezione precedente.

1. Si chiede ad OpenGL di generare un VAO (Vertex Array Object) che rappresenterà la nostra mesh sulla GPU.

2. Una volta fatto il bind del VAO si generano i due buffer che conterranno i vertici e gli indici e si procede a caricare tali informazioni:
  - **VBO** (Vertex Buffer Object) è il buffer di memoria che conterrà i dati dei vertici.
  - **EBO** (Element Buffer Object) è il buffer di memoria che conterrà gli indici.
3. Una volta caricati i dati è necessario però specificare il layout dei vertici<sup>29</sup> La specifica avviene richiamando per ogni attributo del vertice la funzione `glVertexAttribPointer` specificando dimensione ed offset di partenza.

Il codice esatto non verrà riportato per motivi di spazio.

Un'altro elemento da tenere in considerazione in questa fase è il parametro `usage` dato insieme al caricamento dei dati. Si tratta di un hint che specifica il pattern di utilizzo atteso di quei buffer per effettuare ottimizzazioni particolari quando ad esempio i dati sono statici o in stream.

In fnGL tutte le mesh sono statiche, una volta caricate non vengono più modificate dunque si è utilizzato il parametralo `GL_STATIC_DRAW`.

<sup>29</sup>: Ovvero dire ad OpenGL come deve interpretare i dati nel VBO che per ora è un semplice puntatore ad un blocco di byte.

Per l'EBO a meno di specifiche diverse OpenGL assume che il buffer abbia valori ininteri su 4Byte.

### 2.4.3 Model

La classe `Model` astrae il concetto di Modello 3D definito come una composizione di più mesh, solitamente infatti un modello complesso è composto da più mesh e texture. La classe inoltre implementa anche metodi per il caricamento dei modelli utilizzando la libreria ASSIMP.

La libreria ASSIMP legge il file contenente il modello 3D e lo memorizza internamente in una struttura ad albero; La radice è sempre un nodo del tipo `aiScene` e contiene tutte le informazioni base e metadati sul modello 3D, la scena può avere dei sott nodi del tipo `aiNode` che a loro volta possono avere dei sott nodi del tipo `aiMesh` contenenti le mesh.

`Model` dunque oltre a comporre più mesh fa da adapter fra la classe `Mesh` di fnGL e quella di ASSIMP.

In realtà, la struttura ad albero è solo logica, tutti i dati si trovano nell'`aiScene` sotto forma di array, gli altri nodi contengono solo gli indici di accesso a tali dati (ciò consente una maggiore compressione ed evita ripetizioni.)

### 2.4.4 Shader

La classe `Shader` consente di gestire in maniera semplificata alcuni aspetti degli Shaders di OpenGL. In particolare:

1. Lettura, compilazione e link dei file sorgente `.glsl` della coppia vertex shader e fragment shader.
2. Attivazione (e disattivazione) degli shader
3. Set delle uniform con un'interfaccia polimorfica basata su template.

Gli shader consentono di programmare la pipeline grafica manipolando i dati di vertici e texture. In fnGL sono stati implementati 4 tipi di shader per gestire altrettante esigenze di rappresentazione.

### Picking shader

È quello più semplice, il suo scopo è disegnare un oggetto con un unico colore flat; senza luci, ombre o texture. Tale colore identificando univocamente l'oggetto disegnato e potrà essere utilizzato per implementare il picking.

Il Vertex Shader disegna semplicemente i vertici dell'oggetto secondo la model-view-projection.

**Listing 2.10:** Codice sorgente del Vertex Shader. Applica la trasformazione a tutti i vertici del modello.

```
#version 330 core
layout ( location = 0 ) in vec3 position;
layout ( location = 1 ) in vec3 normal;
layout ( location = 2 ) in vec2 texCoords;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main( )
{
    gl_Position = projection * view * model * \
                  vec4( position, 1.0f );
}
```

Il Fragment Shader disegna con un unico colore ( $r, g, b, a$ ) l'oggetto.

**Listing 2.11:** Codice sorgente del Fragment Shader.

```
#version 330

uniform int r;
uniform int g;
uniform int b;
uniform int a;

out vec4 outputF;

void main()
{
    outputF = vec4(r/255.0, g/255.0, b/255.0, a/255.0);
}
```

### Blade shader

È lo shader utilizzato per rappresentare la “blade” ovvero la lama mostrata quando si scorre con il mouse per tagliare i frutti.

Il Vertex Shader è simile a quello della sezione precedente, l'unica differenza è il “forward” delle coordinate texture allo shader successivo in modo che possa utilizzarle per disegnare la lama.

**Listing 2.12:** Codice sorgente del Vertex Shader. Applica la trasformazione a tutti i vertici del modello e ritorna le coordinate texture.

```
#version 330 core
layout ( location = 0 ) in vec3 position;
layout ( location = 1 ) in vec3 normal;
layout ( location = 2 ) in vec2 texCoords;
```

```

out vec2 TexCoords;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main( )
{
    gl_Position = projection * view * model * \
                  vec4( position, 1.0f );
    TexCoords = texCoords;
}

```

Il fragment shader utilizza le coordinate texture per determinare il colore dei pixel della mesh. Le coordinate sono interpolate con un sampler2D che dipende dai parametri specificati alla creazione della texture.

```

#version 330 core

in vec2 TexCoords;
out vec4 color;

uniform sampler2D texture_diffuse;

void main( )
{
    color = vec4( texture( texture_diffuse, TexCoords ) );
}

```

Le lame implementate hanno texture bicolore e per ottenere bordi netti è necessario utilizzare GL\_NEAREST.



**Listing 2.13:** Codice sorgente del Fragment Shader.

**Figura 2.10:** Esempio delle tre texture disponibili per le lame. Tutte e quattro derivano da lame realmente esistenti all'interno del gioco.

### Sprite shader

Questo shader può essere visto come una semplice estensione di quello usato per disegnare la lama che consente in più di specificare il canale alpha del colore della texture, in questo modo è possibile disegnare oggetti trasparenti.

### Fruit Shader

Lo shader dedicato al disegno dei frutti e anche quello più complesso dal momento che oltre alle caratteristiche viste in precedenza implementa anche il lighting secondo il modello di riflessione di Phong.

Nel vertex shader oltre alla posizione dei vertici vengono definite anche:

- ▶ La posizione dell'unica sorgente di luce; che deve poi essere convertita in coordinate view. Si tratta di un parametro fisso.
- ▶ Si ricalcolano le normali dei vertici rispetto la model-view.

Queste informazioni sono poi restituite in output allo shader successivo in modo che possa calcolare secondo il modello di Phong le varie componenti della luce.

**Listing 2.14:** Codice sorgente del Vertex Shader. Applica la trasformazione a tutti i vertici del modello e ritorna le coordinate texture, posizione della luce e le normali.

```
#version 330 core
layout (location = 0) in vec3 aPos;
layout (location = 1) in vec3 aNormal;
layout (location = 2) in vec2 aTexCoords;

out vec3 FragPos;
out vec3 Normal;
out vec3 LightPos;

out vec2 TexCoords;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    // Posizione della luce che illumina la scena
    vec3 lightPos = vec3(0.0f, 30.0f, 30.0f);

    gl_Position = projection * view * model * vec4(aPos, 1.0);
    // Calcolo la posizione del fragment
    FragPos = vec3(view * model * vec4(aPos, 1.0));
    // Calcolo le normali
    Normal = mat3(transpose(inverse(view * model))) * aNormal;
    // Coord world -> coord view per la luce
    LightPos = vec3(view * vec4(lightPos, 1.0));

    TexCoords = aTexCoords;
}
```

Nel fragment shader infine avviene il calcolo della luce e quello del colore della texture, i due valori saranno poi combinato per ottenere il colore finale. La luce scelta di base è bianca.

- ▶ **Luce ambientale:** È la componente “ambientale” della luce derivante dall’ambiente circostante, gli oggetti infatti non sono mai completamente scuri. È derivata dalla luce di base utilizzando un intensità pari a 0.4.
- ▶ **Luce diffusa:** È la componente diffusiva della luce, che varia a seconda di come i raggi impattano sulla superficie degli oggetti e dall’angolo che si viene a creare fra superficie e sorgente. È derivata dalla luce di base attraverso sua posizione e le normali alle superfici.

- **Luce speculare:** È la componente speculare della luce, simula i punti luminosi che si vedono quando la luce riflette su oggetti lucenti. Tale componente varia a seconda dell'angolo fra viewer, superficie e sorgente. Anch'essa è derivata dalla luce di base utilizzando un intensità<sup>30</sup> pari a 0.3 ed un fattore di decay di 32.

L'unione di queste tre componenti, unite al colore del pixel derivato dalla texture consente di determinare il colore finale dell'oggetto.

```
#version 330 core
out vec4 FragColor;

in vec2 TexCoords;
in vec3 FragPos;
in vec3 Normal;
in vec3 LightPos; // posizione luce in wcoord

uniform sampler2D texture_diffuse;

void main()
{
    // Colore della luce, bianco
    vec3 lightColor = vec3(1.0f, 1.0f, 1.0f);

    //::::::::::: CALCOLO DELLA LUCE AMBIENTALE :::::::
    //::::::::::: float ambientStrength = 0.4;
    vec3 ambient = ambientStrength * lightColor;

    //::::::::::: CALCOLO DELLA LUCE DIFFUSA :::::::
    //::::::::::: vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(LightPos - FragPos);
    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = diff * lightColor;

    //::::::::::: CALCOLO DELLA LUCE SPECULARE :::::::
    //::::::::::: float specularStrength = 0.3;
    float decay = 32;

    // versore alla visione del viewer, nel view-space
    // il viewer sta sempre in (0,0,0)
    vec3 viewDir = normalize(-FragPos);
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), decay);
    vec3 specular = specularStrength * spec * lightColor;
```

30: L'utilizzo di un valore maggiore rende i frutti 'plastici' e riduce dunque il realismo.

**Listing 2.15:** Codice sorgente del Fragment shader che implementa l'illuminazione di Phong.

```
// Combino i risultati con il colore della texture
vec3 objectColor = (texture( texture_diffuse, TexCoords)).xyz;
vec3 result = (ambient + diffuse + specular) * objectColor;
FragColor = vec4(result, 1.0);
}
```

# Bibliografia

Qui di seguito i riferimenti in ordine di citazione.

- [1] *Dear ImGui is a bloat-free graphical user interface library for C++. It outputs optimized vertex buffers that you can render anytime in your 3D-pipeline enabled application. It is fast, portable, renderer agnostic and self-contained (no external dependencies).* <https://github.com/ocornut/imgui>.
- [2] *fmt is an open-source formatting library for C++. It can be used as a safe and fast alternative to (s)printf and iostreams.* <https://github.com/fmtlib/fmt>.
- [3] *GLFW, an Open Source, multi-platform library for OpenGL, OpenGL ES and Vulkan development on the desktop. It provides a simple API for creating windows, contexts and surfaces, receiving input and events.* <https://www.glfw.org/>.
- [4] *GLSL + Optional features = OpenGL Mathematics (GLM). A C++ mathematics library for graphics programming.* <https://glm.g-truc.net/>.
- [5] *irrKlang is a high level 2D and 3D cross platform (Windows, macOS, Linux) sound engine and audio library which plays WAV, MP3, OGG, FLAC, MOD, XM, IT, S3M and more.* <https://www.ambiera.com/irrklang/>.
- [6] *single-file public domain (or MIT licensed) libraries for C/C++.* <https://github.com/nothings/stb>.
- [7] *The Open Asset Import Library (short name: Assimp) is a portable Open-Source library to import various well-known 3D model formats in a uniform manner.* <https://www.assimp.org/>.
- [8] *The OpenGL Extension Wrangler Library (GLEW) is a cross-platform open-source C/C++ extension loading library. GLEW provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform. OpenGL core and extension functionality is exposed in a single header file.* <http://glew.sourceforge.net/>.