UNIVERSITA' DEGLI STUDI DI MILANO

Corso di Laurea Magistrale in Informatica

Anno Accademico 2017-2018

PROGRAMMAZIONE GRAFICA
PER IL TEMPO REALE

Implementazione gioco La Goriziana

Prof. Davide Gadia

Studente: Trentadue Giuseppe Fabio 865057

Indice

1. Introduzione

Il lavoro qui presentato nasce durante il corso di Programmazione Grafica per il Tempo Reale e verte sulla progettazione e sviluppo del gioco la Goriziana. Questa è una specialità di gioco del Biliardo all'italiana con 2 squadre, composte da 1 o 2 persone, il cui scopo principale è abbattere i birilli al centro del tavolo per raggiungere un punteggio massimo, deciso all'inizio della partita.

Il progetto è stato realizzato sfruttando le conoscenze fornite a lezione riguardo le principali tecniche di programmazione grafica ed è stato implementato in C++ con il supporto di librerie, quali ad esempio Bullet Physics e GLM, seguendo la specifica OpenGL.

2. La Goriziana

La Goriziana si disputa su un tavolo molto simile a quello da Biliardo, ma privo di buche. La specialità viene giocata con 3 biglie, aventi stessa dimensione ma colori differenti: una bianca ed una gialla, usate dai giocatori o squadre avversarie, ed una rossa con la funzione di pallino. Al centro del tavolo 5 o 9 birilli sono posizionati a croce, perpendicolarmente alle sponde, ad una distanza pari al diametro della biglia.

Lo scopo del gioco è colpire con la stecca la propria biglia e con quest'ultima colpire la biglia avversaria affinché essa o il pallino abbattano dei birilli. In questo caso si ottengono punti a proprio vantaggio. Nel caso in cui uno o più birilli siano abbattuti dalla propria biglia, il totale dei punti viene attribuito all'avversario.

La partita termina al raggiungimento del punteggio fissato dai giocatori all'inizio della partita, che in genere varia dal valore 200 al valore 300.

I punti assegnati per i birilli abbattuti vengono calcolati in base alla loro posizione nella croce, più interni implica più punti, ed alla combinazione di sponde, biglia avversaria e pallino colpite.

Per soffermarsi su aspetti più importanti ai fini del progetto, si è deciso di non implementare tutte le regole del gioco ma solo quelle che riguardano i differenti punteggi dei birilli, in base alla loro posizione: 2 punti per i birilli più esterni ed 8 per il birillo più interno.

3. Implementazione

Il processo implementativo ha coinvolto tre componenti, il cui sviluppo è stato effettuato quasi in parallelo:

-implementazione della grafica della scena

-implementazione ed impostazione della componente fisica

-impostazione di altre librerie e componenti.

Tutto lo sviluppo è incentrato nella connessione di queste componenti in un main, in cui avviene il settaggio delle variabili fondamentali per OpenGL e la dichiarazione di metodi e variabili che consentono la renderizzazione e l'interazione con l'utente.

3.1 Grafica

Lo sviluppo della parte grafica del progetto ha avuto inizio con la realizzazione di classi di supporto, al fine di rendere il codice più leggibile e soprattutto modulare. Queste classi sono:

-classe Shader: si occupa di leggere i file esterni contenenti il Vertex Shader, il Fragment Shader e, quando presente, il Geometry Shader. Una volta che sono stati caricati in memoria i loro contenuti, vengono compilati singolarmente e collegati in un Program Shader, il cui ID viene salvato in un attributo in modo che si possa facilmente risalire ad una precisa coppia di Shader.

-classe Camera: si occupa di gestire la visuale del giocatore sulla scena. È stata implementata, per ottimizzare l'esperienza di gioco, una camera che ruota attorno alla biglia per consentire di prendere la mira in modo semplice, come si può vedere nel codice sottostante. L'interazione tra l'utente ed il gioco è consentita solo mediante il mouse.

```
mat4 RotateAroundPoint(GLfloat angle, vec3 axis){
	GLfloat velocity = this->MouseSensitivity * (-angle);

	vec3 direction = this->selectedBallPos - vec3(0.0f);

	mat4 matrix = translate(mat4(1.0f), direction);
	matrix = rotate(matrix, velocity, axis);
	matrix = translate(matrix, -direction);

	this->Position = matrix * vec4(this->Position, 1.0f);
	this->Front = matrix * vec4(this->Front, 0.0f);

	return lookAt(this->Position, this->Position + this->Front, this->Up);
}
```

-classi Mesh e Model: si occupano del caricamento nella scena di modelli e relative mesh, mediante l'allocazione di buffer e la lettura degli Shader associati al modello.

Occorre precisare che tutta la gestione della componente matematica (matrici, vettori ed operazioni tra di loro) è stata effettuata sfruttando i metodi della libreria GLM, come si può osservare nel codice sopra riportato: `mat4`, `vec3`, `rotate`, `translate`, etc., rappresentano tipi di variabili e metodi implementati nella libreria. Mentre, per il caricamento del modello nella classe Model, viene utilizzata la libreria Assimp che permette la lettura corretta e completa del file .obj e la conseguente renderizzazione.

A seguito della realizzazione di queste classi basilari, è venuta la generazione e renderizzazione di una texture Cubemap, ossia una texture composta da 6 texture 2D che formano un cubo. L'effetto dato da questa Cubemap è quello di trovarsi in un ambiente pressoché infinito, ingannando l'occhio umano. Come per la renderizzazione dei modelli, anche la renderizzazione di questa texture viene fatta mediante degli opportuni Shader che leggono e modificano le componenti di ogni singolo fragment.

Infine, come modello d'illuminazione per la scena, è stato implementato il modello di Cook-Torrance, un modello basato sulla fisica e sull'idea che solo le microfacce avente normale in una certa direzione contribuiscono alla riflessione speculare. Infatti, per la componente diffusiva, viene utilizzato il modello di Lambert mentre per la componente speculare viene applicata la seguente formula

$$\frac{F(\Theta)D(\psi)G}{\pi(v \cdot n)(l \cdot n)}$$

dove:

-F rappresenta il coefficiente di riflettanza di Fresnel, con approssimazione di Schlik, che descrive l'interazione della luce con una superficie, ossia la quantità di luce riflessa sulla base dell'angolo di incidenza della luce. Il valore da cui dipende questo coefficiente è F0.

-D rappresenta la distribuzione di Beckmann delle microfacce. Il valore da cui dipende questo fattore è m, indice di rugosità della superficie, che varia da 0, superficie liscia, e 1, superficie molto rugosa.

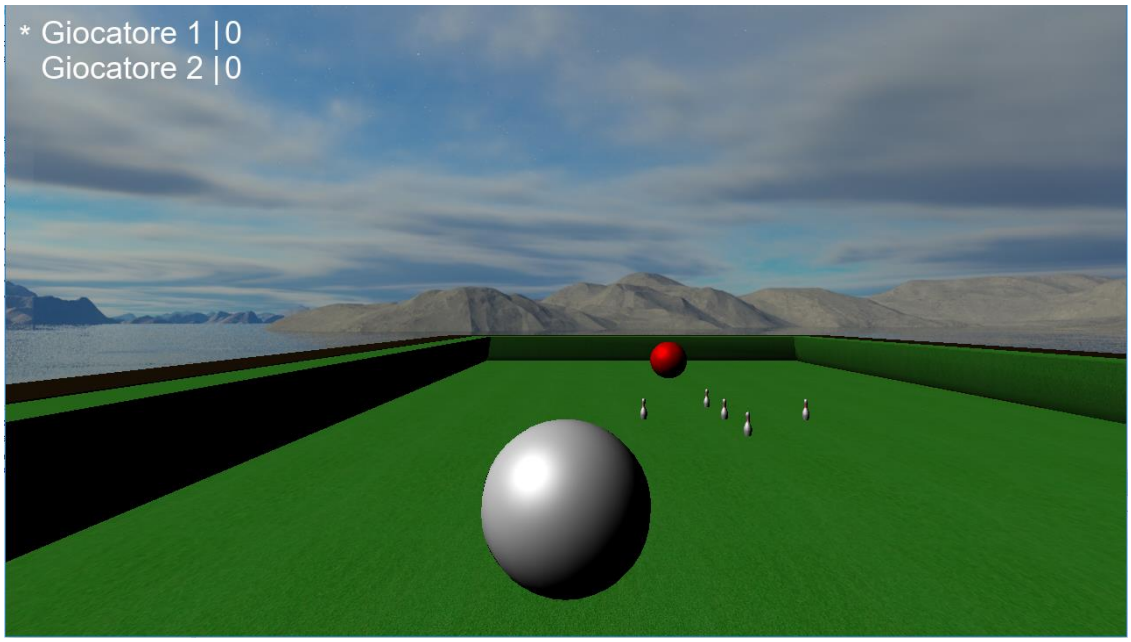
-G rappresenta il termine geometrico, che descrive fenomeni di self-shadowing.

-*l*, *v* ed *n* rappresentano, rispettivamente, il vettore di incidenza della luce con la superficie, con verso opposto, il vettore di vista, con verso opposto, ed il vettore normale.

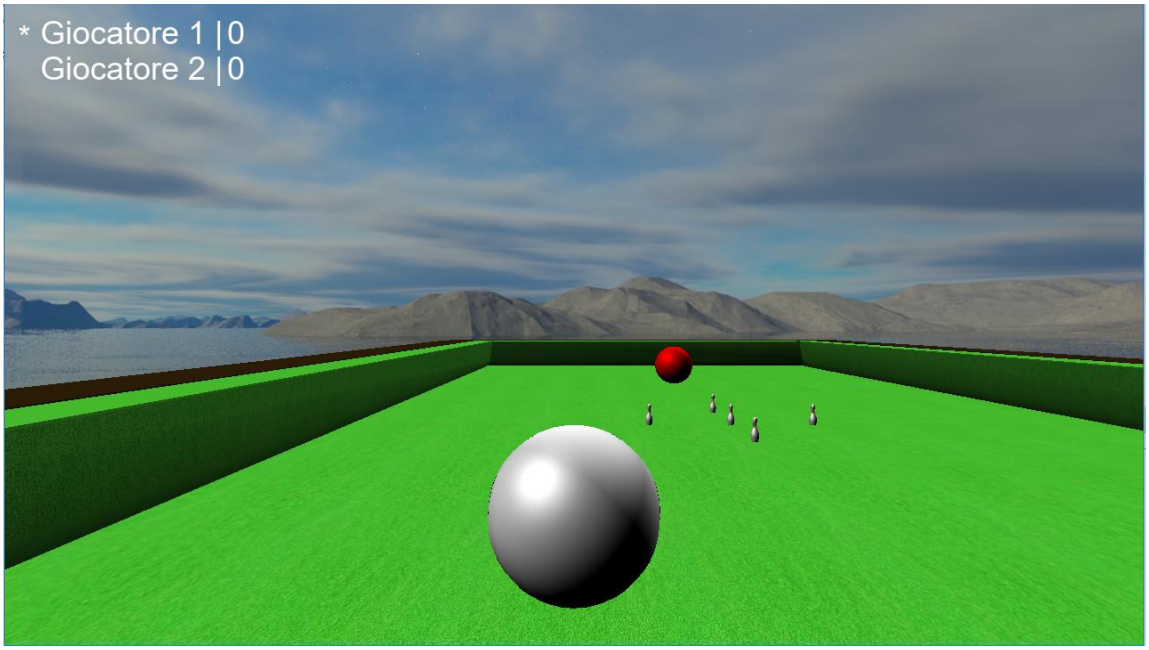
-θ e Ψ rappresentano, rispettivamente, l'angolo tra *l* ed *n* e l'angolo tra *n* ed *h*, dove *h* è il vettore intermedio tra *l* e *v*.

Inoltre è presente anche il coefficiente Kd, impiegato per alterare tutta la componente speculare.

Seguendo questo modello e la Cubemap, sono state inserite due luci direzionali: una simula la luce solare, proveniente dal sole presente nella skybox, e l'altra impiegata per diminuire le ombre introdotte dalla prima. Nell'immagine presente di seguito, viene mostrata la scena con una luce direzionale.



Qui invece la stessa scena, ma con una luce direzionale ed una fill light, per diminuire le ombre.



Dopo una serie di test, sono stati decisi i seguenti valori:

	Modelli senza Texture		**Modelli con Texture**	
	Directional Light	**Fill Light**	**Tavolo**	**Birilli**
m	0.3	0.3	0.6	0.4
F0	2.0	0.1	4.0	2.0
Kd	0.8	0.8	1.0	0.7

3.1.1 Shader

La classe Shader, come asserito precedentemente, ha il compito di leggere e caricare in memoria i file contenenti i diversi tipi di shader: Vertex, Fragment e, quando presente, Geometry.

Ai fini del progetto, sono stati sviluppati 5 diverse coppie di (Vertex, Fragment) per consentire la renderizzazione di 5 componenti grafiche:

1. Shaders per gli oggetti della scena senza texture: il codice utilizzato è basato su quello introdotto durante le lezioni di laboratorio del corso, con lievi modifiche e l'aggiunta della seconda directional light. Riproduce il modello di illuminazione di Cook-Torrance.

2. Shaders per gli oggetti della scena con texture: il codice utilizzato è basato su quello introdotto durante le lezioni di laboratorio del corso, con lievi modifiche e l'aggiunta della seconda directional light. Riproduce il modello di illuminazione di Cook-Torrance.

3. Shaders per il debugger della Bullet: la coppia di Vertex e Fragment shader utilizzati sono stati creati per disegnare i contorni delle btCollisionShape assegnate ai btRigidBody. Vengono passate, come uniform, le matrici Projection, View e Model, e mediante i buffer, la posizione ed il colore dei contorni.

4. Shaders per la Cubemap: questa coppia viene impiegata per la renderizzazione dello skybox. In input, vengono passate le matrici Projection, View e le 6 texture da applicare alle facce della Cubemap.

5. Shaders per il text-rendering: questa coppia di shaders è stata realizzata per la renderizzazione dei singoli glifi-caratteri. Come uniform, si forniscono in input la matrice Orthografic, l'immagine bitmap che rappresenta il glifo ed il colore desiderato. Mediante i buffer, il Vertex shader riceve la posizione e le coordinate di texture.

Si tralascia la descrizione dettagliata degli Shader in quanto il lavoro svolto è incentrato su altri aspetti.

3.2 Fisica

L'implementazione di una simulazione fisica all'interno del gioco è stata realizzata utilizzando il motore fisico Bullet Physics, che gestisce le collisioni tra gli oggetti e le dinamiche di interazione di softBody e rigidBody. Per utilizzare al meglio le componenti di questa libreria, è stata creata una classe Physics che racchiude tutto il codice relativo al settaggio iniziale del dynamicsWorld, il codice utile alla creazione dei rigidBody, da assegnare agli oggetti della scena, e quello relativo alla cancellazione di tutte le componenti fisiche, una volta terminato il suo utilizzo.

Gli attributi principali della classe, nonché attributi necessari alla creazione della simulazione, sono i seguenti:

-btAlignedObjectArray<btCollisionShape*>: array di btCollisionShape in cui sono memorizzate tutte le collision shape dei singoli oggetti presenti nella scena.

-btDefaultCollisionConfiguration*: attributo contenente la configurazione per il collision manager.

-btCollisionDispatcher*: attributo contenente gli algoritmi necessari per la gestione delle collisioni tra coppie di oggetti.

-btBroadphaseInterface*: attributo che fornisce un'interfaccia per individuare le collisioni tra gli oggetti

-btSequentialImpulseConstraintSolver*: attributo utile per gestire i constraint presenti nella scena.

Definiti questi attributi, si passa alla creazione dell'oggetto btDynamicsWorld, nel costruttore della classe Physics, il quale fornisce un'interfaccia che gestisce gli oggetti fisici nella scena ed i loro vincoli.

Un altro metodo definito nella classe Physics è createRigidBody che permette di definire un btRigidBody specificando parametri come la posizione e la rotazione iniziale, la dimensione, la massa ed i coefficienti di attrito e restituzione. Di seguito è riportato il suo prototipo.

```
btRigidBody* createRigidBody(int type, glm::vec3 pos, glm::vec3 size, glm::vec3 rot, float m,
float friction , float restitution);
```

L'ultimo metodo dichiarato è Clear, il quale libera la memoria occupata dagli oggetti sopra dichiarati e termina la simulazione fisica. Di seguito è riportato il suo prototipo.

```
void Clear();
```

Terminata la creazione della classe, il focus è passato al main in cui avviene l'introduzione della fisica. Inizialmente viene creato un oggetto istanza di Physics, sui cui viene chiamato il metodo createRigidBody per ogni elemento della scena: dalle biglie, ai birilli ed infine al tavolo da gioco. Per ognuno di questi, viene realizzato un apposito btRigidBody con una forma precisa: una sfera per ogni biglia, un cilindro per ogni birillo e svariati piani che, posizionati accuratamente, hanno permesso la realizzazione di un corpo rigido statico per il tavolo.

Al fine di comprendere e visualizzare meglio le relazioni tra gli elementi fisici appena creati, è stata sviluppata la classe BulletDebugDrawer che, estendendo la classe astratta btIDebugDraw ed implementando i suoi metodi virtuali, permette la visualizzazione dei contorni di tutti i btRigidBody presenti nella scena. L'attenzione è stata posta su due campi in particolare:

-la creazione del metodo SetMatrices, il quale setta le matrici Projection, View e Model per gli Shader associati al debugger.

-l'implementazione del metodo virtuale drawLine, che renderizza, in base agli Shader associati, i contorni degli oggetti fisici della scena.

Nell'introduzione della simulazione fisica nella scena, sono state riscontrate delle criticità in due aspetti: nella scelta della forma da dare al rigidBody dei birilli e nella determinazione dei coefficienti di attrito e restituzione per tutti gli elementi.

Per quanto riguarda la forma, la scelta è ricaduta su un cilindro o una capsula. Inizialmente la capsula sembra rievocare al meglio la forma del birillo ma, data la conformazione della base, la tendenza era di curvare e non rimanere perfettamente perpendicolare al piano. Pertanto si è optato per il cilindro, il quale però ha il problema opposto dato che la base troppo larga tende a limitare le cadute, a cui si può però ovviare riducendo al minimo la dimensione del rigidBody. Tuttavia, nel legare il modello grafico del birillo al btRigidBody cilindrico, si è presentata una criticità inaspettata: il centro di massa del modello grafico, data la forma peculiare di un birillo, è spostato verso il basso e non perfettamente centrato sull'asse verticale. Di conseguenza, il centro di massa del modello grafico e di quello fisico non coincidono e quindi si verifica la situazione mostrata in figura qui sotto, dove le righe nere rappresentano i contorni della btCylinderShape visualizzati dal debugger descritto precedentemente.

Per risolvere tale problema, è stata inserita la btCylinderShape, collision shape che compone il btRigidBody, all'interno di una btCompoundShape al fine di poter inserire una traslazione locale e spostare la cylinderShape. Questo ha permesso solo di diminuire l'offset tra i due modelli. Dopo una ricerca approfondita sulla scarna documentazione della libreria e sui forum riguardo l'argomento, l'unica soluzione

trovata è stata quella di inserire, oltre alla traslazione nella collision shape, anche una minima traslazione del modello grafico. Il codice riportato qui sotto mostra come è stata gestita questa criticità.

```
if (type == cylinder){
    btVector3 dim = btVector3(size.x, size.y, size.z);

    btTransform localTransform;
    localTransform.setIdentity();
    localTransform.setOrigin(btVector3(0.0, 0.1, 0.0));

    btCylinderShape* innerShape = new btCylinderShape(dim);

    btCompoundShape* shape = new btCompoundShape();
    shape->addChildShape(localTransform, innerShape);
    cShape = shape;

}
```

Per la determinazione dei coefficienti di attrito e restituzione, la criticità riscontrata riguarda la collision shape delle biglie e quella del piano. Dato che il modello fisico della sfera è una sfera perfetta, la superficie di contatto tra di essa ed il modello fisico del piano è costituita da un solo punto e pertanto l'attrito tra i due è inesistente. Per ovviare al movimento infinito della sfera sul piano, sono stati settati altri coefficienti, nel btRigidBody, per limitarne il movimento e realizzare una simulazione più veritiera. I coefficienti considerati sono i seguenti:

-angularDamping: forza che si oppone al movimento rotatorio della sfera.

-rollingFriction: coefficiente di attrito che ha effetto solo sulla rotazione della sfera.

-linearDamping: forza che si oppone al movimento lineare della sfera.

In maniera più lieve, sono stati riscontrati gli stessi problemi tra la collision shape cilindrica dei birilli e quella del piano, quando il birillo ha il suo asse verticale parallelo al piano (quando è stato abbattuto). Nella tabella sottostante si possono osservare i valori determinati per i coefficienti sopra elencati che hanno permesso di ottenere dei movimenti molto vicini a quelli reali.

	Sfera/Piano	**Cilindro/Piano**
angularDamping	0.4	0.05
rollingFriction	0.4	0.04
linearDamping	0.3	null

Qui di seguito viene mostrato il codice in cui questi valori vengono settati.

```
if (type == sphere){
    rbInfo.m_angularDamping = 0.4;
    rbInfo.m_rollingFriction = 0.4;
    rbInfo.m_linearDamping = 0.3;
} else if (type == cylinder){
        rbInfo.m_angularDamping = 0.05;
        rbInfo.m_rollingFriction = 0.04;
    }
```

dove rbInfo è un attributo di btRigidBody, di tipo btRigidBodyConstructionInfo, in cui vengono memorizzati tutti i valori relativi alla fisica, come la massa, i coefficienti dei vari tipi di attrito, il coefficiente di restituzione, il motionState, etc..

3.3 Libreria FreeType

Come ultima componente, è stata introdotta la libreria FreeType che consente di renderizzare del testo nella scena. Nel progetto in esame si è scelta questa libreria perché permette di renderizzare del testo in modo abbastanza semplice, se non occorrono più dei 128 caratteri presenti nella tabella ASCII.

Entrando nel dettaglio, la libreria in questione tratta i TrueType font, che sono collezione di caratteri-glifi definiti da combinazione di funzioni matematiche chiamate spline. Di conseguenza i glifi possono essere generati proceduralmente come se fossero delle immagini vettoriali. Quello che FreeType fa è caricare questa tipologia di font e, per ogni glifo-carattere, generare un'immagine bitmap e calcolare diverse metriche che permettano una corretta visualizzazione.

Innanzitutto, nel main, si inizializza la libreria e si carica il font, come mostrato nel codice sottostante.

```
FT_Library library;
if (FT_Init_FreeType(&library))
    cout << "Errore nell'inizializzazione della libreria FreeType!" << endl;

FT_Face face;
if (FT_New_Face(library, "font/arial.ttf", 0, &face))
    cout << "Errore nel caricamento del font!" << endl;
```

In seguito si crea una struttura Character che rappresenti il singolo glifo caricato e si caricano in memoria, in una struttura map, tutti i 128 caratteri della tabella ASCII in modo da rendere facile la conversione da carattere testuale, che si vuole visualizzare, a carattere renderizzato nella scena. Questo viene fatto dal seguente metodo:

```
void create_dictionary(FT_Face face);
```

A questo punto, possiamo rilasciare la memoria occupata dalla libreria.

Dato che il testo andrà renderizzato, sarà necessario un Vertex ed un Fragment Shader appositi che disegneranno il testo come se fosse una texture.

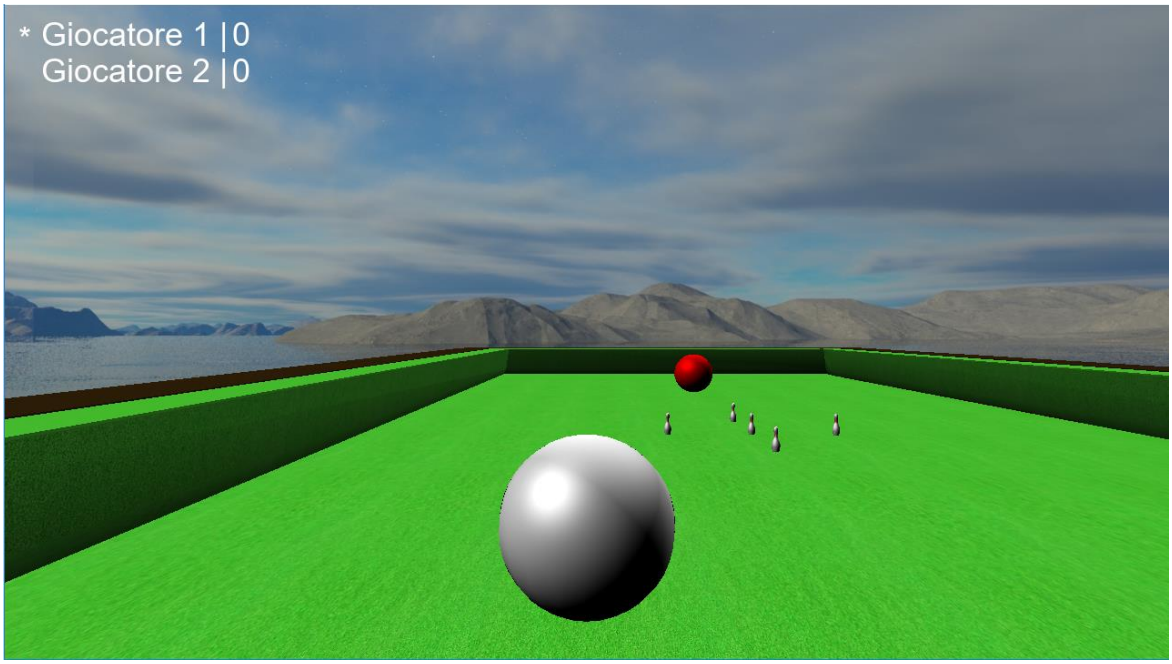
Infine si crea un metodo, mostrato subito sotto, che permetta di convertire il testo inserito dall'utente con una serie di immagini da renderizzare nella posizione indicata.

```
void render_text(Shader &shader, string text, GLfloat x, GLfloat y, GLfloat scale,
glm::vec3 color);
```

Di seguito viene mostrato il codice relativo alla struttura Character.

```
struct Character {
    //Campo utilizzato per memorizzare l'ID della texture del glyph
    GLuint textureID;
    //Campo in cui è salvata la dimensione del glyph
    glm::ivec2 size;
    //Campo in cui è salvato l'offset dalla baseline all'angolo in alto a sinistra
    glm::ivec2 bearing;
    //Campo in cui è salvato l'offset dal punto advance al carattere successivo
    GLuint advance;
};
```

Questa libreria è stata inserita al fine di visualizzare su schermo informazioni riguardo la turnazione dei due giocatori, indicato dal simbolo ' * ' accanto al nome, ed il loro punteggio, come si può vedere nell'immagine sottostante.



3.4 Main

Come detto precedentemente, le componenti descritte finora sono state sviluppate quasi parallelamente e sono state inserite man mano nel main, in modo da essere testate e connesse con le altre.

Dapprima vengono settate tutte le variabili della specifica OpenGL, che indicano la versione che si vuole utilizzare, il livello di astrazione desiderato, etc.. In seguito, vengono dichiarate tutte le funzioni di callback che risponderanno alle interazioni dell'utente con l'applicazione e le opzioni di rendering di OpenGL.

A questo punto, definito il contesto su cui deve agire la componente grafica, sono presenti una serie di dichiarazioni di variabili di tipo Shader e di tipo Model che rappresentano tutti gli shaders ed i modelli che verranno utilizzati. Definite le variabili relative alla grafica, si passa a quelle relative alla fisica. Infatti sono presenti una serie di definizioni di btRigidBody che, come detto precedentemente, si andranno a collegare ai modelli grafici presenti nella scena.

Viene inizializzata adesso la libreria FreeType, si carica il font e vengono caricati, nella struttura dati di tipo map, tutti i 128 caratteri.

Si arriva pertanto al render loop, un ciclo infinito, all'interno del quale:

-vengono calcolate le matrici View e Model in modo da aggiornare in tempo reale gli elementi della scena.

-vengono gestite tutte le interazioni dell'utente con il gioco, mediante il metodo `glfwPollEvents()`, che richiama le funzioni di callback definite precedentemente al verificarsi di particolari eventi, quali lo spostamento del cursore e la pressione nel pulsante sinistro del mouse.

-viene settato il debugger per la simulazione fisica e viene chiamato il metodo stepSimulation, sull'attributo dynamicsWorld della classe Physics, che ad ogni frame aggiorna i parametri fisici degli oggetti.

```
poolSimulation.dynamicsWorld->setDebugDrawer(&debugger);

debugger.SetMatrices(&shaderDebugger, projection, view, model);
poolSimulation.dynamicsWorld->debugDrawWorld();

poolSimulation.dynamicsWorld->stepSimulation(
(deltaTime < maxSecPerFrame ? deltaTime : maxSecPerFrame), 10);
```

-vengono chiamati i due metodi per la renderizzazione degli elementi, uno per quelli senza texture e l'altro per quelli con texture.

```
draw_model_notexture(shaderNoTexture, modelBall, bodyBallWhite, bodyBallRed, bodyBallYellow);

draw_model_texture(shaderTexture, modelTable, modelPin, vectorPin);
```

-viene chiamato il metodo per la renderizzazione della cubemap

```
draw_skybox(shaderSkybox, modelSkybox, textureSkybox);
```

-viene renderizzato il testo necessario ad indicare il turno del giocatore ed il relativo punteggio

```
render_text(shaderText, "*", 15.0f, 670.0f - playerIndexOffset * player, 1.0f, glm::vec3(1.0f));
render_text(shaderText, "Giocatore 1 | ", 40.0f, 675.0f, 1.0f, glm::vec3(1.0f));
render_text(shaderText, to_string(counterPoint[0]), 250.0f, 675.0f, 1.0f, glm::vec3(1.0f));
render_text(shaderText, "Giocatore 2 | ", 40.0f, 635.0f, 1.0f, glm::vec3(1.0f));
render_text(shaderText, to_string(counterPoint[1]), 250.0f, 635.0f, 1.0f, glm::vec3(1.0f));
```

-alla fine, viene gestita la turnazione dei giocatori, spostando la camera sulla biglia relativa e riposizionando i birilli eventualmente abbattuti dal tiro precedente.

Al termine del render loop, quando i giocatori premono il tasto di fine partita, vengono rilasciate tutte le locazioni di memoria allocate per le variabili e gli oggetti utilizzati finora.

In conclusione, si riporta il codice relativo al metodo creato per simulare il lancio di una biglia, mediante la pressione del pulsante sinistro del mouse. Il metodo controlla che la biglia, lanciata dal precedente giocatore, si sia fermata e quindi ricava la posizione del mouse, trasformando le coordinate schermo in coordinate mondo, per determinare la direzione dell'impulso da applicare. A questo punto, attiva il corpo rigido della biglia, passato in input, e chiama il metodo `applyImpulse(impulse, relPos)` che applica un impulso di valore `btVector3 impulse` nella posizione relativa `btVector3 relPos`.

```
void throw_ball(btRigidBody* ball) {
    if (!checkShoot) {
        glm::mat4 screenToWorld = glm::inverse(projection * view);

        GLfloat shootInitialSpeed = 20.0f;

        GLfloat x = (mouseX / SCR_WIDTH) * 2 - 1, y = -(mouseY / SCR_HEIGHT) * 2 + 1;

        btVector3 impulse, relPos;

        glm::vec4 mousePos = glm::vec4(x, y, 1.0f, 1.0f);
        glm::vec4 direction = glm::normalize(screenToWorld * mousePos) * shootInitialSpeed;

        impulse = btVector3(direction.x, direction.y, direction.z);
        relPos = btVector3(1.0, 1.0, 1.0);

        ball->activate(true);
        ball->applyImpulse(impulse, relPos);

        checkShoot = true;
    }
}
```

4. Conclusioni

Durante lo sviluppo, le principali difficoltà e criticità sono state riscontrate nell'utilizzo e nella comprensione della libreria Bullet Physics a causa della documentazione: è presente una documentazione testuale per nulla esaustiva e gli sviluppatori cercano di ovviare a questo fornendo un pacchetto di esempi che copre circa il 20% di tutte le classi, metodi ed attributi implementati. Tuttavia, si può asserire che è molto completa e più adatta ad un lavoro molto più lungo focalizzato principalmente sulla fisica, come può essere lo sviluppo di un videogame.

Come sviluppi futuri e possibili miglioramenti, c'è innanzitutto l'introduzione dell'intero sistema di regole e punteggi della Goriziana. Si può inoltre pensare di fornire la possibilità di cambiare graficamente il gioco, permettendo la modifica degli Shader delle biglie e del tavolo e consentendo di modificare l'ambiente circostante.