Report LAB02

Nicola Modugno

prof.ssa Serena Morigi

Abstract—Questo report descrive lo sviluppo di una demo 2D interattiva basata sui progetto `LAB_2` fornito, con l'aggiunta di animazioni, fisica, interfaccia grafica e meccaniche di gioco. Sono stati integrati un ciclo giorno/notte, la possibilità di trascinare la pallina con il mouse, un sistema di sbarre dinamiche, un'interfaccia utente realizzata con ImGui e diverse migliorie grafiche.

1. Introduzione

Il progetto si basa sulla traccia proposta nel laboratorio `LAB_2`, che richiedeva di compilare ed eseguire i progetti di base forniti e sviluppare una demo 2D interattiva originale. I progetti iniziali coprivano una gamma di esempi: dalla semplice animazione (Jumpball) al videogioco con HUD e shader di sfondo (Alieno). A partire da questi, lo studente ha realizzato un sistema personalizzato, arricchendo la scena con dinamiche fisiche, interazione utente, ciclo giorno/notte, ostacoli, punteggio e timer.

2. Trascinamento della pallina e rotazione delle pale eoliche

Una delle prime funzionalità introdotte è stata la possibilità di trascinare la pallina con il mouse. Quando l'utente clicca sulla pallina, il sistema verifica la distanza tra il punto di clic e il centro della pallina. Se la distanza è inferiore al raggio, viene attivata la modalità di trascinamento. Questo comportamento è implementato nella funzione seguente:

```
void mouse_button_callback(GLFWwindow* window, int
    ↪ button, int action, int mods) {
    if (button == GLFW_MOUSE_BUTTON_LEFT && action ==
    ↪ GLFW_PRESS && !game_over) {
        double x, y;
        glfwGetCursorPos(window, &x, &y);
        float fx = (float)x;
        float fy = (float)(height - y);
        float dx = fx - posx;
        float dy = fy - (posy + 40);
        float distanza = sqrt(dx * dx + dy * dy);
        if (distanza <= 40.0f && dy >= -40.0f) {
            mouse_pressed = true;
        }
    }
    if (button == GLFW_MOUSE_BUTTON_LEFT && action ==
    ↪ GLFW_RELEASE && !game_over) {
        mouse_pressed = false;
    }
}
```

Code 1. Callback per trascinamento della pallina

Le pale eoliche, che inizialmente erano statiche, sono state aggiornate affinché ruotassero nel tempo. Per ottenere questo comportamento, sono state decommentate le righe di codice nel file `LAB2D.cpp` che incrementano l'angolo di rotazione e aggiornano l'immagine sullo schermo.

```
angolo_pala += velocita_rotazione;
```

Code 2. Aggiornamento angolo delle pale

```
mat4 Rotazione = rotate(mat4(1.0), radians(angolo_pala),
    ↪  vec3(0.0, 0.0, 1.0));
Model = translate(Model, posizionePala);
Model = Model * Rotazione;
```

Code 3. Applicazione della rotazione sui triangoli

3. Ciclo giorno/notte animato

Per rendere l'ambiente più dinamico, è stato aggiunto un ciclo giorno/notte. Il sole si muove lungo una traiettoria ellittica e il colore del cielo cambia progressivamente grazie alla funzione di interpolazione lineare:

```
vec4 lerpColor(vec4 a, vec4 b, float t) {
    return vec4{
        (1 - t) * a.r + t * b.r,
        (1 - t) * a.g + t * b.g,
        (1 - t) * a.b + t * b.b,
        (1 - t) * a.a + t * b.a
    };
}
```

Code 4. Interpolazione tra colori

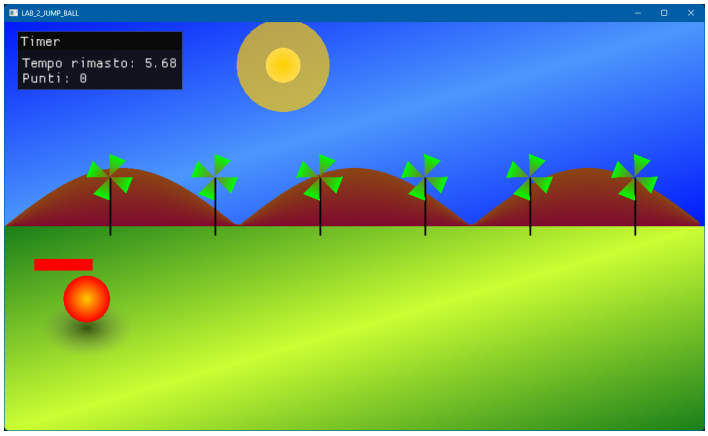

Figure 1. Demo 2D con sole, sbarre, ombra e interfaccia

La posizione del sole è calcolata tramite:

```
float xSole = cx + rx * cos(t + PI);
float ySole = cy + ry * sin(t);
```

Code 5. Posizione ellittica del sole

4. Fisica del rimbalzo e gestione dell'ombra

Inizialmente, la pallina rimbalzava anche contro il "cielo", creando un comportamento poco realistico. Il controllo è stato corretto per limitare il rimbalzo solo al suolo:

```
if (posy + 40 >= altezza_suolo && vy > 0) {
    vy = -vy * 0.8f;
    posy = altezza_suolo - 40;
}
```

Code 6. Rimbalzo corretto sul suolo

Anche l'ombra della pallina è stata migliorata: ora segue la pallina solo in orizzontale quando essa supera l'orizzonte, come mostrato in figura 2.

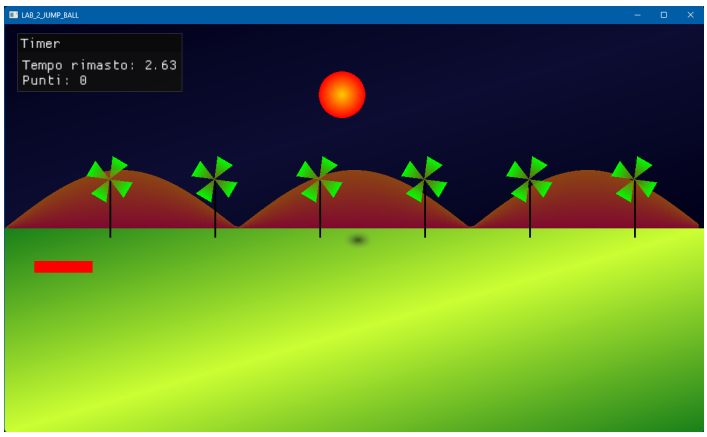

Figure 2. Spostamento dell'ombra

5. Sistema particellare

Il sistema particellare del progetto LAB2D è stato realizzato per simulare effetti visivi dinamici, come scie colorate e piccoli fuochi d'artificio. Le particelle sono memorizzate in un array dinamico di tipo `Pointxy`, inizializzato nel file `init_geometrie.cpp`:

```
int n_Particellari = 1000;
Pointxy* Particelle = new Pointxy[n_Particellari];
```

Code 7. Inizializzazione sistema particellare

Ogni particella è un punto dotato di posizione (x, y) e colore (r, g, b, a). Questi dati venegono utilizzanti nel ciclo `drawScene()` in LAB2D.cpp:

```
Model = mat4(1.0);
glUniformMatrix4fv(MatModel, 1, GL_FALSE, value_ptr(
    ↪ Model));
glGenVertexArrays(1, &VAO_PARTICELLE);
glBindVertexArray(VAO_PARTICELLE);
glGenBuffers(1, &VBO_PARTICELLE);
glBindBuffer(GL_ARRAY_BUFFER, VBO_PARTICELLE);
glBufferData(GL_ARRAY_BUFFER, n_Particellari * sizeof(
    ↪ Pointxy), &Particelle[0], GL_STATIC_DRAW);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 7 *
    ↪ sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
glVertexAttribPointer(1, 4, GL_FLOAT, GL_FALSE, 7 *
    ↪ sizeof(float), (void*)(3 * sizeof(float)));
glEnableVertexAttribArray(1);
glPointSize(3.0);
glDrawArrays(GL_POINTS, 0, p_attivi);
glBindVertexArray(0);
```

Code 8. Rendering sistema particellare

Durante il gioco, le particelle vengono gestite tramite tre funzioni principali:

- `aggiungiParticella(vec3 posizione, vec3 direzione)`: aggiunge una nuova particella nella direzione desiderata, codificando la direzione nel colore (r,g).
- `aggiungiFlussoColorato(vec3 posizione)`: genera un flusso circolare di particelle attorno a un punto dato, iterando su un angolo che va da 0 a 2π.
- `muoviParticella(int index)`: aggiorna posizione e trasparenza della particella, decodificando la direzione a partire dai canali r e g.

Le particelle vengono animate ogni frame dalla funzione `aggiornaParticelle()`, che aggiorna tutte le particelle attive e rimuove quelle con a <= 0.

```
void rimuoviPoiScorri(int index) {
    int i;
    p_attivi--;
    if (index != p_attivi) {
        // Remove the first point, slide the rest down
        for (i = index; i < p_attivi; i++) {
            Particelle[i] = Particelle[i + 1];
        }
    }
}

void aggiungiParticella(vec3 posizione, vec3 direzione)
    ↪ {

    //Eliminiamo particelle oltre il massimo consentito
    while (p_attivi >= n_Particellari)
        rimuoviPoiScorri(0);

    int i = p_attivi;
    Particelle[i].x = (float)posizione.x;
    Particelle[i].y = (float)posizione.y;
    Particelle[i].r = ((float)direzione.x + 1) / 2;
    Particelle[i].g = ((float)direzione.y + 1) / 2;
    Particelle[i].b = 0.3f;
    Particelle[i].a = 1.0f;
    p_attivi++;
}

void aggiungiFlussoColorato(vec3 position) {
    vec3 posizione = position;
    vec3 direzione_start = vec3{ 0,0,0 };
    vec3 direzione = direzione_start;
    float angolo;
    float step = PI / 32;
    for (angolo = 0; angolo < (2 * PI) - step; angolo +=
    ↪  step) {
        direzione.x = glm::cos(angolo);
        direzione.y = glm::sin(angolo);
        aggiungiParticella(posizione, direzione);
    }
}

void muoviParticella(int index) {

    //Calcolo "direzione"
    float r = Particelle[index].r;
    float g = Particelle[index].g;
    float dx = (r * 2) - 1;
    float dy = (g * 2) - 1;

    //Aggiorno posizione e alfa-value
    Particelle[index].x += dx;
    Particelle[index].y += dy;
    Particelle[index].a -= 0.005f;

    //printf("n:%d\tx:%f\ty:%f\tr:%f\tg:%f\n", index,
    ↪ Particelle[index].x, Particelle[index].y,
    ↪ Particelle[index].r, Particelle[index].g);
}

void aggiornaParticelle() {
    int i;
    for (i = 0; i < p_attivi; i++) {
        muoviParticella(i);
        if (Particelle[i].a <= 0)
            rimuoviPoiScorri(i);
    }
}
```

Code 9. Funzioni per il rendering del sistema particellare

Infine, le particelle vengono rese visibili attraverso un controllo nella funzione `update()` che verifica la presenza di particelle attive e lo stato della partita:

```
    if (game_over) {
        if (contatore_esplosione >=
    ↪ intervallo_esplosione) {
            aggiungiFlussoColorato(pos_pala_eolica1);
```

```
            aggiungiFlussoColorato(pos_pala_eolica2);
            aggiungiFlussoColorato(pos_pala_eolica3);
            aggiungiFlussoColorato(pos_pala_eolica4);
            aggiungiFlussoColorato(pos_pala_eolica5);
            aggiungiFlussoColorato(pos_pala_eolica6);
            contatore_esplosione = 0; // reset
        }
        else {
            contatore_esplosione++;
        }
    }

    if (!game_over) {

        game_timer -= difficolta / 60;


        if (game_timer <= 0.0f) {
            game_timer = 0.0f;
            game_over = true;
        }
    }

    if (p_attivi > 0) {
        aggiornaParticelle();
    }
```

Code 10. Controlli per l'attivazione del sistema particellare

6. Sbarre dinamiche e sistema di punteggio

Sono state aggiunte delle sbarre che appaiono casualmente e premiano l'utente quando vengono colpite. La loro generazione è gestita dalla funzione:

```
void generaNuovaSbarra() {
    if (prima_sbarra) {
        punti = 0;
        prima_sbarra = false;
    } else {
        punti += 10;
        difficolta += 0.6f;
    }
    game_timer += 0.6f / 60;
    float margine = 50.0f;
    sbarra_pos.x = margine + randf() * (width - 2 *
     ↪ margine - sbarra_larghezza);
    sbarra_pos.y = height * 0.25f + randf() * (height *
     ↪ 0.25f);
    sbarra_visibile = true;
}
```

Code 11. Generazione di nuove sbarre

Il rendering effettivo delle sbarre è gestito nella funzione `drawScene()` in `LAB2D.cpp` in questo modo:

```
   if (sbarra_visibile) {
        Model = mat4(1.0);
        Model = translate(Model, vec3(sbarra_pos.x,
     ↪ sbarra_pos.y, 0.0f));
        Model = scale(Model, vec3(sbarra_larghezza,
     ↪ sbarra_altezza, 1.0f));
        glUniformMatrix4fv(MatModel, 1, GL_FALSE,
     ↪ value_ptr(Model));
        glBindVertexArray(VAO_SBARRA);
        glDrawArrays(GL_TRIANGLES, 0, vertices_Sbarra);
        glBindVertexArray(0);
    }
```

Code 12. Rendering della nuova sbarra

7. Interfaccia grafica con ImGui

Per visualizzare il punteggio e il tempo rimanente è stata utilizzata la libreria ImGui e realizzata un'apposita funzione `my_interface()` nel file `Gui.cpp` che viene richiamata nel ciclo principale del gioco. L'interfaccia è semplice, e non-collapsabile:

```
ImGui::Begin("Timer", NULL, ImGuiWindowFlags_NoCollapse
     ↪ | ImGuiWindowFlags_AlwaysAutoResize);
ImGui::Text("Tempo rimasto: %.2f\nPunti: %d", game_timer,
     ↪  punti);
ImGui::End();
```

Code 13. Finestra ImGui per HUD

Inoltre, grazie all'integrazione con ImGui, viene mostrata una finestra popup con il messaggio "Game Over" che include anche il totale dei punti accumulati. Questo popup è centrato sullo schermo e non è interattivo: scompare solo al riavvio del gioco o alla chiusura della finestra.

```
    if (game_over) {
        ImGui::OpenPopup("Game Over");

        // Centro il popup
        ImVec2 center = ImGui::GetMainViewport()->
     ↪ GetCenter();
        ImGui::SetNextWindowPos(center,
     ↪ ImGuiCond_Appearing, ImVec2(0.5f, 0.5f));

        if (ImGui::BeginPopupModal("Game Over", NULL,
     ↪ ImGuiWindowFlags_NoCollapse |
     ↪ ImGuiWindowFlags_AlwaysAutoResize)) {
            ImGui::Text("Punti: %d", punti);
            ImGui::EndPopup();
        }

        glfwSetInputMode(window, GLFW_CURSOR,
     ↪ GLFW_CURSOR_NORMAL);
    }
```

Code 14. Finestra pop-up

Figure 3. Stato di Game Over