Slovak Technical University

Faculty of Informatics and Information Technologies

Informatics, Computer Science

Lucas Daniel Espitia Corredor

Adrian Quirante Gonzales

Principles of Computer Graphics and Image Processing

Ing. Peter Kapec, PhD.

Disco Club

Tuesday 16:00

11/12/2025

Contents

1. Introduction

The purpose of this project is to complete the assignment given by Eng. Peter Kapec, PhD, in the course Principles of Computer Graphics and image processing (PPGSO). For the project we use the C++ programming language together with the OpenGL libraries, following good coding practices.

The work will include modelling objects in 2D and 3D, creating animations and geometric transformations, and simulating particles for visual effects. We will also use lighting and shading techniques to make the scene look more realistic.

Finally, we will program a dynamic camera system so the environment can be explored from different angles during the simulation. In short, the simulation is a video at least two minutes long that explores a variety of objects, and by "object" we mean any animated or static element found in a nightclub.

We will also recreate a particle system, for example falling confetti or small effects tied to some objects. The camera angle rotates and changes over time, so the scene shows different viewpoints at certain time intervals.

This part is explained in more detail in the next section. The main goal is to learn about 2D and 3D images, image processing, lighting and shadows, and the set of techniques needed to build computer simulations.

2. Original Proposal

Scene 1 (0:00–0:05)

The camera enters the room through the door. From outside you only see a wall with a sign showing the entrance.

Scene 2 (0:05–0:15)

The camera stays looking at the room (you can sense the people and at the end you can make out the DJ, where the lights come from). The DJ is placed facing the camera, so the camera only needs to move forward to reach the DJ. From the very beginning you hear the DJ's music; it gets louder as the camera goes into the room and approaches the DJ. The party theme is reggaeton, so the music is in that genre.

Scene 3 (0:15–0:25)

The camera moves forward toward the DJ (over the crowd) until it is right in front of him.

Scene 4 (0:25–0:35)

The camera stays still, watching the DJ and his setup (where the DJ stands): his computer, his speakers, his mixing desk, etc. You can also see colored lights coming from behind him toward the dance floor.

Scene 5 (0:35–0:40)

The camera rotates around the DJ, going from the front (where it was) to behind him.

Scene 6 (0:40–0:50)

After the move, you see the DJ from the back and a wide view of the room. The room includes the DJ area, a dance floor, people dancing, a bar on one side, decorations on the other, a disco ball, and lights in the upper corners.

Scene 7 (0:50–0:55)

The camera rises to see the DJ from above.

Scene 8 (0:55–1:05)

Now the camera shows the DJ's table in detail: his computer, his mixing desk, and any other items on the table.

Scene 9 (1:05–1:15)

The camera slowly turns toward the disco ball at the top. While the ball spins on its own axis, there is a quick zoom in and out.

Scene 10 (1:15–1:25)

From there, the camera moves to focus on the bar where drinks are sold.

Scene 11 (1:25–1:35)

The camera shows the bar in detail: the drinks, the bartender, the people, etc.

Scene 12 (1:35–1:45)

The camera returns to the position behind the DJ.

Scene 13 (1:45–1:55)

The camera moves to the side opposite the bar.

Scene 14 (1:55–2:05)

The camera shows the wall decorations.

Scene 15 (2:05–2:15)

The camera goes back to where it was before.

Scene 16 (2:15–2:35)

You see the whole room again. Now confetti comes out from one part of the room (to be decided) and falls down onto the people. At this moment people get excited (they start to jump or move).

Scene 17 (2:35–2:45) (Final)

The camera fades into the confetti

3. Change of proposal

As the project developed, the proposed scenes varied considerably due to the complexity required for certain scenes. Although the idea of recreating a disco club is fundamentally the same as before, the use of some objects has changed slightly or drastically depending on the case. Motivated by creating a project that meets the expectations of the course, drastic decisions were made during its development.

Consequently, the result is not the same as initially defined, and this may generate some controversy. However, the essence of the project remains intact, and the objectives stated in the Introduction section are still the same.

For this reason, it is appropriate to dedicate a section to explaining the new sequence of scenes and, in future sections, the objects that were ultimately defined.

3.1 Defined scenes and final sketch

a. Scene 1 – Entrance

Keyframe 1 (0:00–0:07):
The camera observes the street and the sky (skybox), completely outside the club.

Keyframe 2 (0:07–0:09):
The camera slowly rotates to show the entire skybox until it aligns with the club entrance.

Keyframe 3 (0:09–0:11):
The camera remains in position, staring at the wall next to the entrance. Looking an unstable light.

Keyframe 4 (0:11–0:13):
The camera rotates to the right to focus on the illuminated neon figure of the girl ("Lady Neon")., understanding that lady is the unstable light.

Keyframe 5 (0:13–0:15):
The camera remains focused on the neon light.

Keyframe 6 (0:15–0:18):
The camera pans to the security camera located near the entrance.

Keyframe 7 (0:18–0:20):
The camera zooms in on the security camera, visually moving closer to it.

Keyframe 8 (0:20–0:22):
The camera zooms back to its original distance.

Keyframe 9 (0:22–0:26):
The camera smoothly rotates toward the club's main entrance.

Keyframe 10 (0:26–0:28):
The camera slowly walks toward the entrance.

Keyframe 11 (0:28–0:34):
The camera turns to look directly at the security camera, creating a cinematic moment.

Keyframe 12 (0:34–0:36):
The camera remains focused on the security camera without moving (dramatic pause).

Keyframe 13 (0:36–0:38):
The camera returns to face the main entrance.

Keyframe 14 (0:38–0:42):
The door begins to open, and the camera observes the movement.

Keyframe 15 (0:42–0:44):
The camera slowly moves inside the club.

Keyframe 16 (0:44–0:46):
Once inside, the camera rotates to face the door again from the inside.

Keyframe 17 (0:46–0:48):
The door closes behind the camera; the camera maintains its rotation.
Moonlight is deactivated to begin lighting the club.

b. Scene 2 – Interior

Keyframe 1 (49–52):
The camera starts inside the club, observing the general interior area after the door has closed.

Keyframe 2 (52–54):
The camera moves up and forward to observe the tables from a high vantage point.

Keyframe 3 (54–56):
The camera zooms in on the tables, bringing them closer to highlight them.

Keyframe 4 (56–58):
The camera rotates slightly toward the bar, visually zooming in on the bar area.

Keyframe 5 (58–60):
The camera continues zooming toward the top of the bar.

Keyframe 6 (60–62):
The camera rotates to focus directly on the DJ at the back of the club.

Keyframe 7 (62–64):
The camera zooms out, returning to the panoramic view of the club.

Keyframe 8 (64–66):
The camera returns to the tables, reverting to the initial orientation of this section.

Keyframe 9 (66–70):
The camera descends to table level and begins to move between them along the first row.

Keyframe 10 (70–74):
The camera smoothly moves down the center of the row of tables, focusing on the middle table.

Keyframe 11 (74–76):
The camera moves to the end of the row of tables.

Keyframe 12 (76–78):
The camera shifts to the next row of tables, looking slightly downward.

Keyframe 13 (78–80):
The camera moves toward the table with the "lava" texture, highlighting it.

Keyframe 14 (80–82):
The camera zooms in on the lava table and observes the animated movement of the texture.

Keyframe 15 (82–92):
The camera orbits the lava table in three successive positions (82, 84, 86 seconds), showing the surface from different angles.

Keyframe 16 (92–94):
The camera lingers on the lava table in detail for a brief pause.

Keyframe 17 (94–95):
The camera moves toward the bar, zooming in on the first lamp above it.

Keyframe 18 (95–98):
The camera observes the bar lamp, focusing on its illumination.

Keyframe 19 (98–102):
The camera moves toward the next lamp, panning along the bar.

Keyframe 20 (102–106):
The camera observes the second lamp and then provides a panoramic view of the bar from a high vantage point.

Keyframe 21 (106–110):
The camera pans to face the dance floor.

Keyframe 22 (110–114):
The camera moves forward toward the dance floor, heading toward the center of the emissive tiles.

Keyframe 23 (114–120):
The camera closely examines the animated pattern of the light tiles before the next transition begins.

c. Scene 3 – Dance Floor

Keyframe 1 (120–126):
The camera remains above the dance floor, observing the tiles, which have changed animation. They now move in a wave.

Keyframe 2 (126–130):
The camera rises and moves toward the center of the club to observe the disco ball from above.

Keyframe 3 (130–134):
The camera maintains its high position, focused entirely on the disco ball as it rotates and reflects light.

Keyframe 4 (134–136):
The camera rotates to look directly at the floor, specifically at the dance floor tiles.

Keyframe 5 (136–140):
The camera remains focused, observing the luminous floor animation from a bird's-eye view.

Keyframe 6 (140–142):
The camera lowers and moves toward the DJ, aligning itself with the DJ booth.

Keyframe 7 (142–144):
The camera lingers on the DJ from a medium distance.

Keyframe 8 (144–146):
The camera sharply pans left to focus on one of the effects cannons (confetti machine).

Keyframe 9 (146–149):
The camera remains focused on the cannon, highlighting its lighting and animation.

Keyframe 10 (149–151):
The camera moves to the DJ's position, adopting their point of view looking out at the crowd.

Keyframe 11 (151–155):
The camera maintains this subjective view of the DJ looking at the dance floor, reinforcing the sense of immersion.

Keyframe 12 (155–157):
The camera moves down and closer to the DJ, positioning itself directly in front of their console.

Keyframe 13 (157–160):
The camera remains close to the DJ setup, observing their posture and movements.

Keyframe 14 (160–162):
The camera begins a quick scan of the DJ's setup, rotating slightly to show the equipment.

Keyframe 15 (162–166):
The camera continues the sweeping motion, showing the setup.

Keyframe 16 (166–168):
The camera pulls back to the position behind the DJ, closing the scene with a wide view of the stage.

d. Scene 4 – Chaotic Ending

Keyframe 0 (2:48–2:58):
The camera remains in the position behind the DJ, observing the dance floor as the initial confetti falls.
(Event at 170.0s → The left confetti cannon fires)
(Event at 175.0s → The right cannon also fires)

Keyframe 1 (2:58–3:00):
The camera maintains the same position and continues observing the confetti shower in front of the DJ booth.

Keyframe 2 (3:00–3:02):
The camera moves to the right of the DJ to observe another cannon/effects machine in operation.

Keyframe 3 (3:02–3:05):
The camera remains fixed on that cannon for a few seconds.

Keyframe 4 (3:05–3:08):
The camera moves up to a higher position and observes the upper part of the club (ceiling lights, structure, etc.).

Keyframe 5 (3:08–3:12):
The camera remains in the overhead view during the lighting effect and falling confetti.

Keyframe 6 (3:12–3:20):
The camera moves down and focuses on the dance floor, looking towards the animated tiles and observing the light pattern as the confetti falls around.

4. Most interesting features

The project is built around a small but coherent set of data structures that organize the scene, the objects and their animation.

First, this section discusses data structures and the coherent organization of base classes.

Second, it covers the most important algorithms used for the solution, primarily those necessary for creating objects or special animations.

Third, a tree-like scene graph will be included. Following this, the class diagram will be included.

Finally, a series of screenshots corresponding to the previously defined scenes will be shown (see section 3.1).

4.1 Data Structures

This section presents the core data structures used in the implementation of the 3D demo.

It makes extensive and meaningful use of custom object-oriented data structures that organize the entire scene and its behaviour.

a) ClubbingWindow (Window Manager Class)

The ClubbingWindow class acts as the main window and application controller.
It manages:

- The global Scene instance.
- Initialization of different parts of the club (entrance, interior, dance floor, ending sequence).
- Cinematic sequences, including camera keyframes and scripted events.
- Input handling, storing keyboard and mouse states for movement and interaction.
- Important object references (doors, security camera, DJ set, lights).
- Rendering loop and per–frame updates through onIdle().
- Window resizing, mouse look, and music playback control.

In short, ClubbingWindow is the high-level orchestration class that ties the engine together, initializes content, and drives the entire application.

b) Scene (Central Scene Data Structure)

The Scene class is the core data structure that organizes all objects, lights, shaders and render buffers used in the project.
It acts as a container and manager for every subsystem of the engine.
Key structural responsibilities:

- Holds the current scene ID (Entrance, Interior, DanceFloor, Ending).
- Stores the global Camera object (std::unique_ptr<Camera>).
- Maintains a list of root objects in a scene graph
- Organizes renderable objects into specialized vectors for different shader pipelines:
  - phongObjects
  - normalMapObjects
  - instancedObjects
  - particleObjects
  - skyBoxObjects

  This improves rendering efficiency and shader batching.
- Stores all lights using pointers (std::vector<std::unique_ptr<Light>> lights;
- Maintains all GPU resources needed for shadow mapping, HDR, tone mapping, and bloom post-processing (FBOs, depth maps, ping-pong buffers).
- Stores references to animated lights, such as the neon flickering light.

Overall, Scene is a high-level orchestration data structure that groups together objects, lights, shaders and buffers, providing a central update and render loop for the entire environment.

c) Object (Base Scene-Graph Node)

The Object class is the fundamental data structure of the engine.
It represents a node in the scene graph, storing all spatial data and hierarchical relationships between elements of the 3D world.
Core structural elements:

- Local transformation data:
  - Position.
  - Rotation
  - Scale
  - ModelMatrix
- Parent–child hierarchy:
  - std::list<std::unique_ptr<Object>> childObjects;
  - Object* parentObject;
- Material attributes:
  - CastsShadow
  - IsTransparent
  - SpecularStrength
  - Shininess
  - And More…
- Utility functions:
  - Model matrix generation
  - Rendering / updating of children
  - Depth rendering for shadow maps

In short, Object defines the hierarchical structural backbone of the entire project, enabling grouped transformations, complex composed models, and organized rendering.

d) Light (Lighting Data Structure)

The Light class represents the fundamental data structure for all lighting in the scene.
It supports three types of lights—Directional, Point, and Spot—using a unified structure with shared and type-specific fields.
Key structural elements:

- Light type
- Common lighting components:
  - ambient
  - Diffuse
  - specular
    plus corresponding base* values for animations (e.g., flickering neon).
- Directional-light data:
  - direction
- Point/Spot-light data:
  - position
  - attenuation factors (constant, linear, quadratic)
- Spotlight-specific fields:

    - spotDirection
    - cutOff
    - outerCutOff
- Enabled flag to activate/deactivate lights dynamically.
- Shader upload

In the engine, Light serves as a compact data container for all lighting properties, making it easy to manage arrays of lights and upload them efficiently to shaders.

e) Camera (View & Animation Data Structure)

The Camera class is the data structure responsible for storing all information related to the viewer's position, orientation, and projection settings.
It also integrates the cinematic keyframe system, enabling scripted camera paths.

Key structural components:

- Spatial data:
    - Position
    - Rotation
    - Tilt

  Used to construct the viewMatrix.
- Projection parameters:
    - Fov
    - Aspect
    - NearPlane
    - FarPlane

  Which define the camera's projectionMatrix.
- Keyframe animation system:
    - KeyFrames keyframes;
    - bool useKeyframes;

  This allows the camera to follow smooth cinematic paths by interpolating between timed keyframes.
- Runtime state:
    - age (timeline)
    - debugEnabled
- Utility methods:
    - Updating view/projection matrices
    - Camera movement controls (moveX/Y/Z, rotate)
    - FOV/aspect resizing

The Camera class thus serves both as a mathematical data container for rendering and as a state machine for timeline-based cinematic animation.

f) KeyFrames (Cinematic Animation Data Structure)

The KeyFrames class is a compact data structure designed to handle timeline-based animations for the camera and other objects.

It stores both spatial keyframes and timed callback events, allowing smooth cinematic sequences.
Core structural elements:

- List of keyframes:
  - std::vector<Keyframe> frames;

  Each entry stores time, position, and rotation/FOV.
  During playback, the system interpolates between frames using a smooth curve.
- Timed events:
  - std::vector<TimedEvent> events;

  These allow triggering scripted actions (e.g., firing confetti cannons) at specific times during the animation.
- Animation state:
  - AnimTime
  - Playing

  Used to advance the timeline and control playback.
- Interpolation helpers:
  - smooth() for easing
  - smoothLerp() for smooth vector interpolation

Altogether, KeyFrames functions as a lightweight animation controller, enabling cinematic camera motion and synchronized world events using a clean, data-driven structure.

4.2 Important Algorithms

This section describes several algorithms implemented in the project that go beyond basic rendering and demonstrate advanced graphical and simulation techniques. Likewise in the creation of objects in defined or undetermined positions.

These algorithms are essential for achieving realistic lighting, smooth animation, particle behaviour, and high-quality post-processing effects.

a) Generate Table Set

The project includes a simple but effective procedural placement algorithm used to generate multiple table sets in the interior of the club.
Instead of placing objects manually, the algorithm iterates through a 2D grid and instantiates TableSet objects based on row and column indices.

The algorithm:

- Defines a grid (rows × cols) and spacing parameters.
- Iterates over the grid using nested loops.
- Computes the world position for each table using:
  position = startPos + (c * spacingX, 0, r * spacingZ)
- Creates each table dynamically (std::make_unique<TableSet>).
- Inserts the table into:
  - scene.rootObjects (scene graph)

- scene.phongObjects (render list for the Phong shader)

This algorithm demonstrates procedural object generation, ensuring consistent spatial layout and reducing manual placement effort. It also shows how the engine integrates newly generated objects into both the scene graph and rendering pipeline.

```
void ClubbingWindow::generateTableSets(Scene& scene) {
    const int rows = 2;
    const int cols = 4;
    const float spacingX = 50.0f;
    const float spacingZ = 50.0f;

    const glm::vec3 startPos = {a: -110.0f, b: 0.0f, c: -87.0f};


    for (int r = 0; r < rows; r++) {
        for (int c = 0; c < cols; c++) {

            auto table = std::make_unique<TableSet>( parent: nullptr, TypeTableSet::Table, [&] scene);

            float x = startPos.x + c * spacingX;
            float y = 0.0f;
            float z = startPos.z + r * spacingZ;

            table->position = {a: x, b: y, c: z};
            TableSet* tablePtr = table.get();
            scene.phongObjects.push_back(tablePtr);
            scene.rootObjects.push_back(std::move(table));
        }
    }
}
```

Fig 1. Algorithm to render various table objects in the world.

b) Algorithm in Scene update

Spotlight Shadow Mapping Matrix Algorithm

During each frame update, the engine computes the light-space transformation matrix used for spotlight shadow mapping.
The algorithm performs three steps:

1. View matrix
   Constructed using the spotlight position and direction:
   lookAt(light.position, light.position + light.direction)

2. Projection matrix
   The spotlight cutoff angle is converted into a field of view, and a perspective projection is built for the shadow map.

3. Light-space matrix
   The final transformation is computed as:
   lightSpaceMatrix = projection * view;

This matrix transforms world-space coordinates into the spotlight's clip space and is essential for rendering depth maps used in shadow evaluation.

```cpp
void Scene::update(float time, float dt) {
    camera->update(dt);

    // ---------------------------------------
    // SPOTLIGHT SHADOW MATRIX UPDATE
    // ---------------------------------------
    if (shadowLight) {
        glm::vec3 eye = shadowLight->position;
        glm::vec3 center = shadowLight->position + shadowLight->spotDirection;
        glm::vec3 up( a: 0,  b: 0,  c: 1);

        // 1. View Matrix
        glm::mat4 lightView = glm::lookAt(eye, center, up);
        // 2. Projection Matrix
        float fov = glm::degrees( radians: acos(shadowLight->cutOff));
        glm::mat4 lightProjection = glm::perspective(
            fovy: glm::radians(fov),
            aspect: 1.0f,
            zNear: 1.0f,
            zFar: 100.0f
        );
        // 3. Combined matrix

        lightSpaceMatrix = lightProjection * lightView;
```

Fig 2. Algorithm of Shadow Mapping Matrix Algorithm implementation

Scene Graph Update Algorithm

The same function implements a hierarchical update algorithm for all objects in the scene:

- It iterates over rootObjects.
- Calls each object's updateChildren() method, which recursively updates children in the scene graph.
- If an object reports that it should be removed (e.g., destroyed or expired), it is erased from the list using iterator-safe removal.

This provides an efficient and clean way to update large hierarchical environments.

```
    // ----------------------------------------
    // UPDATE OBJECTS AS USUAL
    // ----------------------------------------
    auto i = std::begin([&]rootObjects);

    while (i != std::end([&]rootObjects)) {
        auto object = i->get();
        if (!object->updateChildren([&]*this, time, dt, glm::mat4{s:1.0f},
entRotation: {a:0, b:0, c:0})) {
            i = rootObjects.erase(i);
        } else {
            ++i;
        }
    }
}
```

Fig 3. Scene Graph Update Algorithm implementation

c) Multi-Pass Render Algorithm

The Scene::render() function implements a complete multi-pass real-time rendering pipeline, combining several advanced algorithms commonly used in modern graphics engines.

The pipeline consists of:

1. Shadow Map Pass

The scene is rendered from the spotlight's point of view into a depth texture.
This produces the shadow map used later for shadow evaluation in the Phong pass.

2. HDR Scene Rendering

The scene is rendered into a floating-point framebuffer (HDR).
This allows bright light values to exceed the [0,1] range, enabling realistic bloom.

3. Skybox Rendering

The skybox is drawn with depth testing set to LEQUAL, ensuring correct background rendering.

4. Phong / Blinn-Phong Lighting Pass

All opaque objects are rendered first.
Transparent objects are then sorted back-to-front using:

5. Instanced Rendering

Dance-floor tiles are drawn in a single draw call using GPU instancing.

6. Particle Rendering (confetti)

Particles are rendered with alpha blending and face-camera orientation (billboarding).

7. Bloom Bright Extraction

A bright-pass shader isolates the high-intensity regions of the scene into a secondary framebuffer.

8. Gaussian Blur Ping-Pong Algorithm

A separable horizontal/vertical blur is applied in multiple passes:

- Each pass alternates between two FBOs,
- reducing bandwidth and achieving smooth bloom.
- This is the standard ping-pong blur algorithm.

9. Final Composite Pass

The engine blends:

- the original HDR scene
- the blurred bloom texture
- the tone-mapping mask
- and outputs the final LDR image to the screen.

This multi-stage pipeline is one of the most complex and interesting systems in the project, combining shadow mapping, HDR rendering, bloom, blending, sorting, and instancing into a cohesive real-time graphics algorithm.

d) Shadow Map Render Algorithm

The renderShadowMap() function implements the shadow-map generation pass used for real-time lighting.

The algorithm works as follows:

- **Bind the shadow framebuffer**
  Rendering is redirected to the depth-only FBO at high resolution (2048×2048), ensuring detailed shadows.
- **Clear depth and configure viewport**
  The depth buffer is reset and the viewport is adjusted to match the shadow-map size.
- **Front-face culling**

“glCullFace(GL_FRONT);”

Fig 4. OpenGL function for Front-face Culling

This reduces self-shadowing artifacts (“shadow acne”) by rendering only the back faces of objects.

- **Use shadow shader**
  The shader receives the precomputed lightSpaceMatrix, which transforms world coordinates into the light's view.
- **Render all shadow-casting objects**
  The algorithm iterates through the scene and skips objects marked as non-shadow-casting.
- **Restore standard rendering state**
  Back-face culling and the default framebuffer are re-enabled.

This algorithm produces the depth map later used in the main lighting pass to determine whether each fragment is in shadow.

```
void Scene::renderShadowMap() {

    glBindFramebuffer( target: GL_FRAMEBUFFER,  framebuffer: depthFBO);
    glViewport( x: 0,  y: 0, SHADOW_WIDTH, SHADOW_HEIGHT);
    glClear( mask: GL_DEPTH_BUFFER_BIT);


    glEnable( cap: GL_DEPTH_TEST);

    glCullFace( mode: GL_FRONT);

    shadowShader->use();
    shadowShader->setUniform( name:  "lightSpaceMatrix", lightSpaceMatrix);

    for (auto* obj :Object*  : phongObjects) {
        if (!obj->castsShadow) continue;
        obj->renderChildren( [&] *this,  [&] *shadowShader);
    }


    glCullFace( mode: GL_BACK);
    glBindFramebuffer( target: GL_FRAMEBUFFER,  framebuffer: 0);
    glViewport( x: 0,  y: 0, screenWidth, screenHeight);
}
```

Fig 5. Render Shadow Algorithm Implementation

e) Screen-Space Fullscreen Quad Rendering Algorithm

The renderQuad() function implements the core algorithm used for all screen-space post-processing passes, such as bloom extraction, Gaussian blur, and final composite rendering.

The algorithm works as follows:

1. Lazy initialization
   The first time the function runs, it creates a VAO and VBO containing six vertices forming two triangles that cover the entire screen.
   Each vertex stores:

- 2D clip-space position ([-1,1] × [-1,1])
- Texture coordinates ([0,1] × [0,1])

2. Upload vertex data
   The quad vertices are uploaded once with GL_STATIC_DRAW, since the fullscreen quad never changes.
3. Configure vertex attributes
   - Attribute 0 → position (2 floats)
   - Attribute 1 → texCoordinates (2 floats)
4. Draw call
   Rendering the quad requires only:

```
glDrawArrays(GL_TRIANGLES, 0, 6);
```

Fig 6. Call function to render simple quad

The active shader determines what happens (tone mapping, bloom combine, blur, etc.).

This algorithm is essential for modern graphics pipelines, enabling all post-processing effects to operate efficiently on 2D textures produced by previous render passes.

```
void Scene::renderQuad() {
    if (quadVAO == 0) {
        float quadVertices[] = {
            -1.0f, -1.0f,   0.0f, 0.0f,
             1.0f, -1.0f,   1.0f, 0.0f,

            -1.0f,  1.0f,   0.0f, 1.0f,
             1.0f, -1.0f,   1.0f, 0.0f,
             1.0f,  1.0f,   1.0f, 1.0f
        };

        glGenVertexArrays(n:1, &quadVAO);
        glGenBuffers(n:1, &quadVBO);
        glBindVertexArray(quadVAO);

        glBindBuffer(target: GL_ARRAY_BUFFER, buffer: quadVBO);
        glBufferData(target: GL_ARRAY_BUFFER, size: sizeof(quadVertices), quadVertices, usage: GL_STATIC_DRAW);

        // position
        glEnableVertexAttribArray(index:0);
        glVertexAttribPointer(index:0, size:2, type: GL_FLOAT, normalized: GL_FALSE,
                stride: 4 * sizeof(float), (void*)0);

        // texCoord
        glEnableVertexAttribArray(index:1);
        glVertexAttribPointer(index:1, size:2, type: GL_FLOAT, normalized: GL_FALSE,
                stride: 4 * sizeof(float), (void*)(2 * sizeof(float)));
    }

    glBindVertexArray(quadVAO);
    glDrawArrays(mode: GL_TRIANGLES, first:0, count:6);
```

Fig 6. Screen Space Quad Rendering Algorithm

f) Dynamic Light Upload Algorithm

The function uploadLightsToShader() implements a dynamic system for sending all active lights in the scene to the shader.

It handles different light types (Directional, Point, Spot) and assigns them proper array indices.

The algorithm proceeds in two stages:

1. Detection and Counting Pass
   The engine iterates through all lights and:
   - Determines whether each type is present (hasDirLight, hasPointLight, hasSpotLight)
   - Counts the number of point lights and spotlights
   - Skips disabled lights for performance
   - Sends the information to the shader

```
void Scene::uploadLightsToShader(ppgso::Shader& shader) const {
    int spotIndex  = 0;
    int pointIndex = 0;

    for (auto &L :const unique_ptr<Light>& : lights) {
        if (!L->enabled) continue;
        switch (L->type) {
            case LightType::Directional:
                hasDir = true;
                break;

            case LightType::Point:
                hasPoint = true;
                pointIndex++;
                break;

            case LightType::Spot:
                hasSpot = true;
                spotIndex++;
                break;
        }
    }
    //Only if there is at least one light in each type
    shader.setUniform( name: "hasDirLight",   hasDir);
    shader.setUniform( name: "hasPointLight", hasPoint);
    shader.setUniform( name: "hasSpotLight",  hasSpot);

    //Add number of lights
    shader.setUniform( name: "numSpotLights", spotIndex);
    shader.setUniform( name: "numPointLights", pointIndex);
```

Fig 7. Detection and Counting Pass implementation in Algorithm

2. Upload Pass
   A second iteration assigns each light an index inside its corresponding uniform array
   Rules:
   - Directional light → always index 0
   - Point lights → sequential indices [0..numPointLights-1]
   - Spotlights → sequential indices [0..numSpotLights-1]

   This avoids hardcoded limits and allows any number of lights without changing shader code

```
    //Reset to add lights
    spotIndex  = 0;
    pointIndex = 0;

    for (auto &L :const unique_ptr<Light> & : lights) {
        if (!L->enabled) continue;
        switch (L->type) {
            case LightType::Directional:
                L->upload( [&] shader,  index: 0);
                break;

            case LightType::Point:
                L->upload( [&] shader, pointIndex);
                pointIndex++;
                break;

            case LightType::Spot:
                L->upload( [&] shader, spotIndex);
                spotIndex++;
                break;
        }
    }

}
```

Fig 8. Uploading Pass to light object to the Shader

g) Instanced Particle System Initialization Algorithm (Confetti System)

The ConfettiSystem constructor implements a GPU-accelerated particle setup using instanced rendering, allowing hundreds or thousands of confetti particles to be rendered efficiently.

The algorithm includes:

1. GPU Resource Allocation

Three dynamic VBOs are created to store per-particle attributes:

- Position buffer (posVBO)
- Colour buffer (colorVBO)
- Rotation angle buffer (angleVBO)

Each buffer reserves memory for MAX_PARTICLES using:

```
glBufferData(..., GL_DYNAMIC_DRAW);
```

Fig 9. Function to reserve memory for instancing in GPU

2. Instanced Attribute Binding

Each VBO is attached to the particle mesh via:

```
meshConfetti->addInstanceBuffer(...);
```

Fig 10. Function Calling to binding the particle mesh

This binds the buffers as per-instance attributes, meaning:

- One quad mesh is stored on the GPU
- Each particle only adds *one instance* to the draw call
- Rendering thousands of particles becomes a single call:

```
void ppgso::Mesh_Assimp::addInstanceBuffer(GLuint vbo, int attribLocation, int components)
{
    for (auto &buffer : buffers) {
        glBindVertexArray(buffer.vao);

        glBindBuffer( target: GL_ARRAY_BUFFER,  buffer: vbo);
        glEnableVertexAttribArray(attribLocation);
        glVertexAttribPointer( index: attribLocation,  size: components,  type: GL_FLOAT,
            normalized: GL_FALSE,  stride: 0,  pointer: nullptr);

        glVertexAttribDivisor( index: attribLocation,  divisor: 1);
    }

    glBindVertexArray( array: 0);
}

void ppgso::Mesh_Assimp::renderInstanced(GLsizei instanceCount)
{
    for (auto &buffer : buffers) {
        glBindVertexArray(buffer.vao);
        glDrawElementsInstanced( mode: GL_TRIANGLES,  count: buffer.size,
            type: GL_UNSIGNED_INT,  indices: nullptr,  primcount: instanceCount);
    }
}
```

Fig 11. Modification of the ppgso::Mesh_Assimp class to perform instantiation correctly.

3. CPU-side Data Storage

Three vectors mirror the GPU buffers:

- instancePositions
- instanceColors
- particles (physics data)

This allows fast updates to particle attributes before pushing them to the GPU.

This algorithm enables the confetti system to render large quantities of particles extremely efficiently, with minimal CPU overhead and maximum GPU parallelism.

h) Hierarchical Scene Graph Update Algorithm

The Object::updateChildren() function implements the engine's recursive update algorithm, which is the core of the scene graph system.

The algorithm works as follows:

1. Local Update and Model Matrix Generation

Each object first updates itself.

```
bool keep = update( [&] scene, time, dt, parentModelMatrix, parentRotation);
```

Fig 12. Calling function to update object to generate the local ModelMatrix

The base implementation generates the local modelMatrix from the parent's matrix, combining translation, rotation and scale.

2. Propagation of Transformations

The object's global rotation is computed from its parent:

```
globalRotation = parentRotation + rotation;
```

Fig 13. Global rotation calculation

3. Recursive Child Updates

Each child object is updated using the current object's modelMatrix as the new parent matrix:

```
if (!child->updateChildren( [&] scene, time, dt, modelMatrix, globalRotation))
```

Fig 14. Recursive calling to update Children

This recursion builds the full scene graph evaluation tree.

4. Safe Removal of Dead Objects

If a child returns false, it is removed from the graph:

```
it = childObjects.erase( it);
```

Fig 15. Save delete to dead children objects

This enables timed objects (particles, temporary effects) to cleanly destroy themselves during runtime.

The project includes a few more, mainly for decision-making in objects, animations, physics algorithms for particles falling in parabolic motion, as well as others for object positioning, but these, although relevant, will not be discussed in detail.

4.3 Scene Graph

The project uses a hierarchical scene graph to organize all elements inside the club environment.

Each object inherits from the base Object class and may contain children, enabling grouped transformations and structured updates.

The root of the graph contains the major subsystems of the scene.

The overall diagram is quite large, so it will be added piece by piece in a logical manner.

On one hand, we have the root objects, which are created in the main scene, and then children of each one will be added.

Each child object is spatially dependent on its parent, following the flow of the tree from top to bottom.

***, Every time you see the asterisk in the diagram, it means that it is a light, and although it is not an object, sometimes has been nested within other objects.**

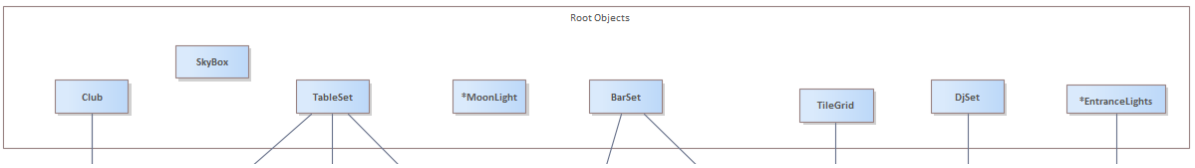

Fig 16. Root Objects

These are the club's core objects, although there are objects that could be encapsulated within the club, such as the barSet or the djSet. It was decided to use this approach and separate them from the parent, since each will have its own responding children. And logically, they can exist even if the "club" itself doesn't exist.

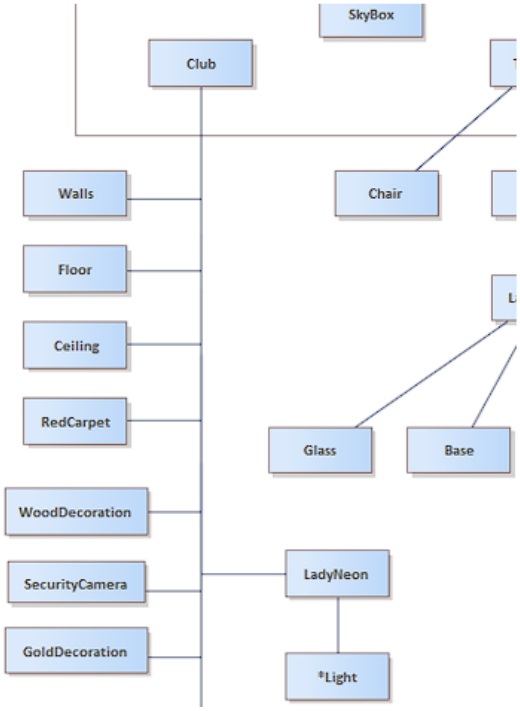

Fig 17. Club's Child Objects, First half.

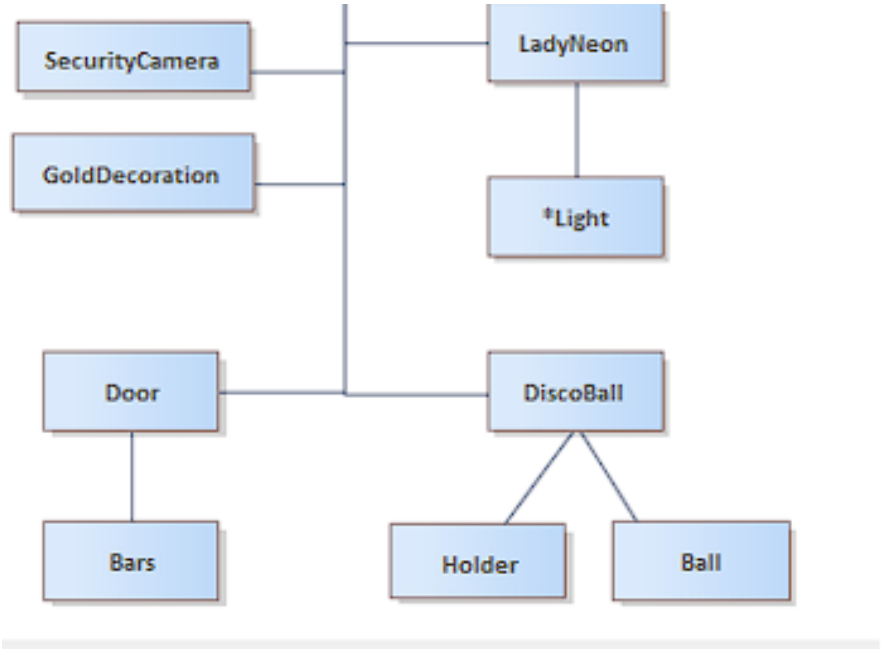

Fig 18. Club's Child Objects, Second Part.

Look at Fig. 18, where within the club, there are children who are simply drawn and that's it. But there are also children who have more children. In the case of Door and his Bars, or the Disco ball, his children are the ball itself and a holder that "holds" it to the ceiling.

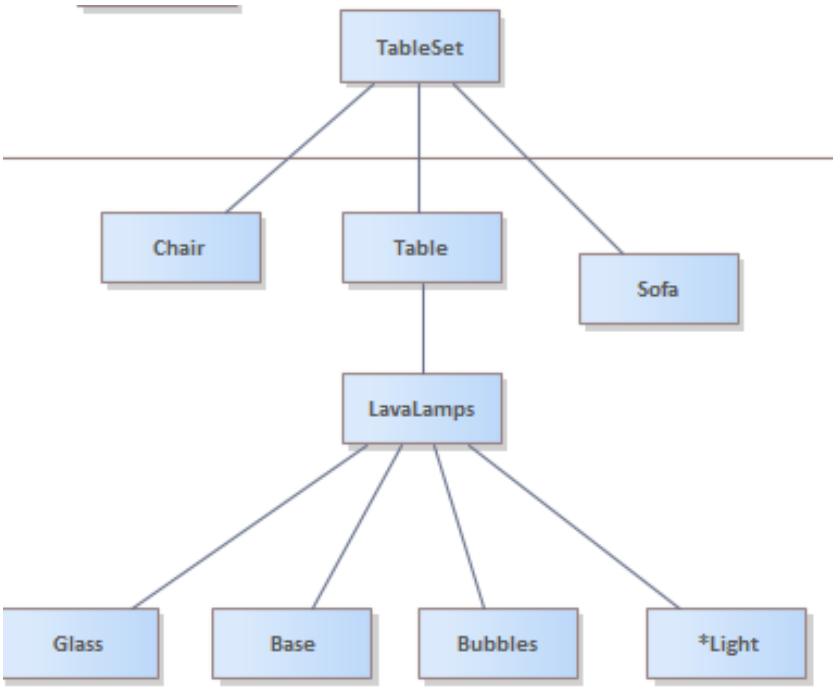

Fig 19. TableSet scene Diagram

The table is the main element. It consists of a table in the center, a sofa, and two chairs. The Lava Lamp object is created on top of the table, which includes its glass, base, light, and a bubble that simulates the liquid in a lava lamp.

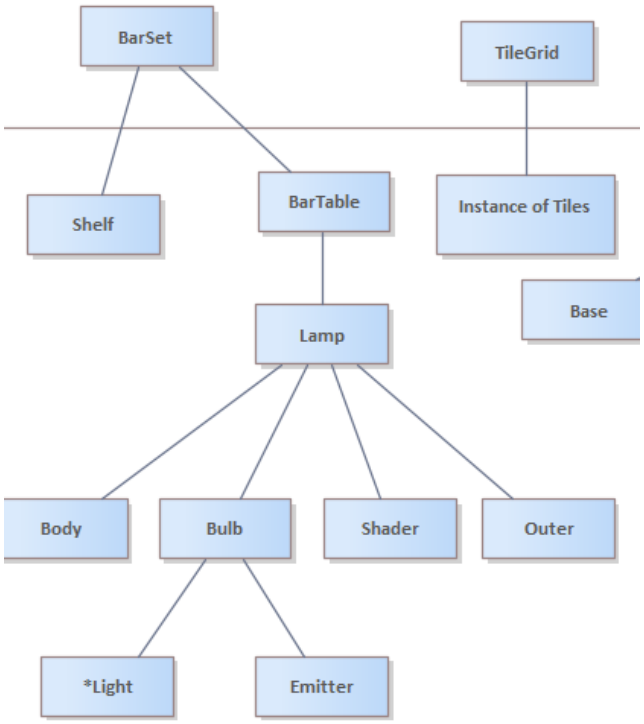

Fig 20. BarSet and TileGrid hierarchy

On one side we have the BarSet, which consists of two main things: the shelves and the bar table. The bar table will have a lamp on top. This lamp itself is made up of several objects: its base, its outer and shader (decoration), and a light bulb. Inside the bulb is a light source and an emitter that acts as a tool to simulate where the light is coming from.

On the other hand we have the TileGrid, which will create an instance of an object depending on the specified size, specifically in this project it has been 16 x 25. Generating 400 instances.

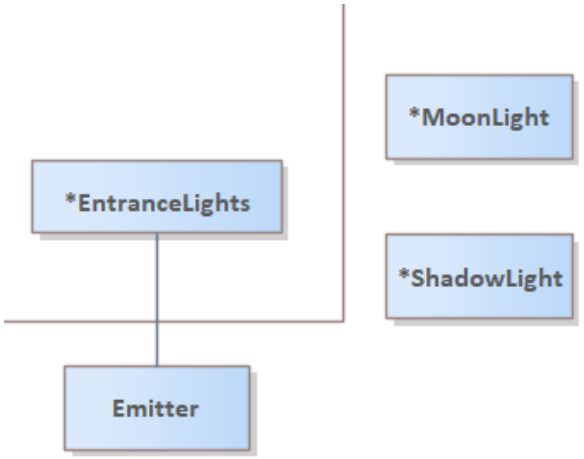

Fig 21. BarSet and TileGrid hierarchy

Although lights are not theoretically considered "objects," in this particular case, the incoming spotlight creates a minimal object hierarchy by establishing an emitter to simulate that the light originates from something.

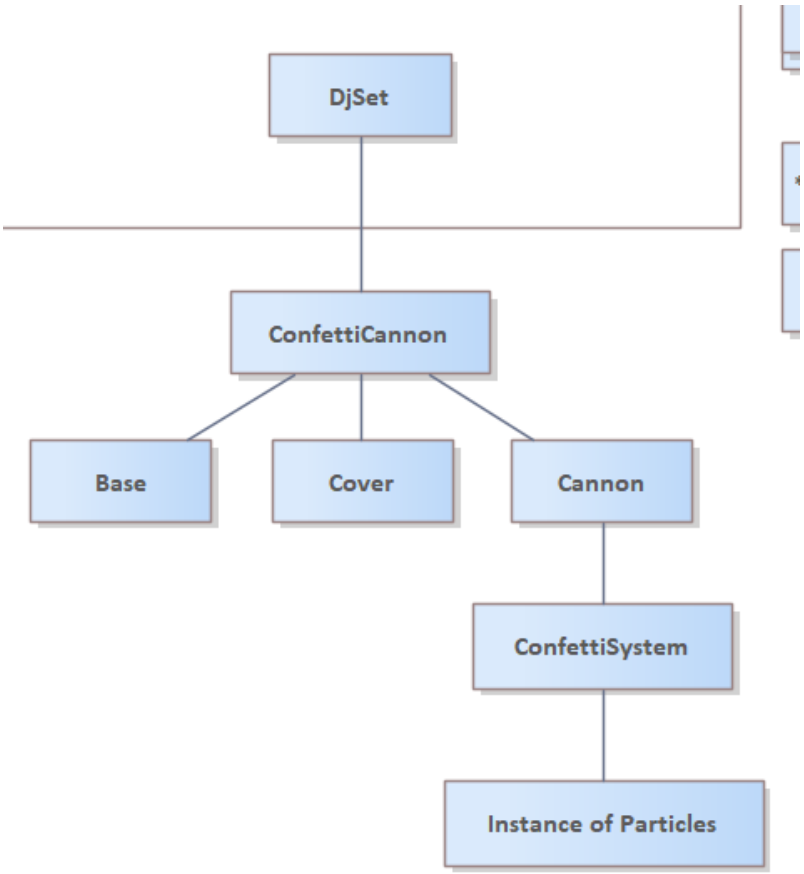

Fig 22. Dj set scene hierarchy

The DJ Set is a complex object in itself, although it could also be considered as part of a larger, overarching system like the club itself. It was decided to separate it, as it contains its own distinct logic. Spatially speaking, it is located within the club. However, the same object could be placed elsewhere if it existed in a larger space. Its components are two cannons, which are located next to the main console (DJ Set).

The cannon itself is composed of several parts: a base that rests on the floor, a cover that protects the top of the object, and the cannon itself. The Confetti system is also created within this system. This system is a particle system that fires instances over time once activated. These particles are fired using a semi-Eulaerian parabolic trajectory.

4.4 Class Diagram

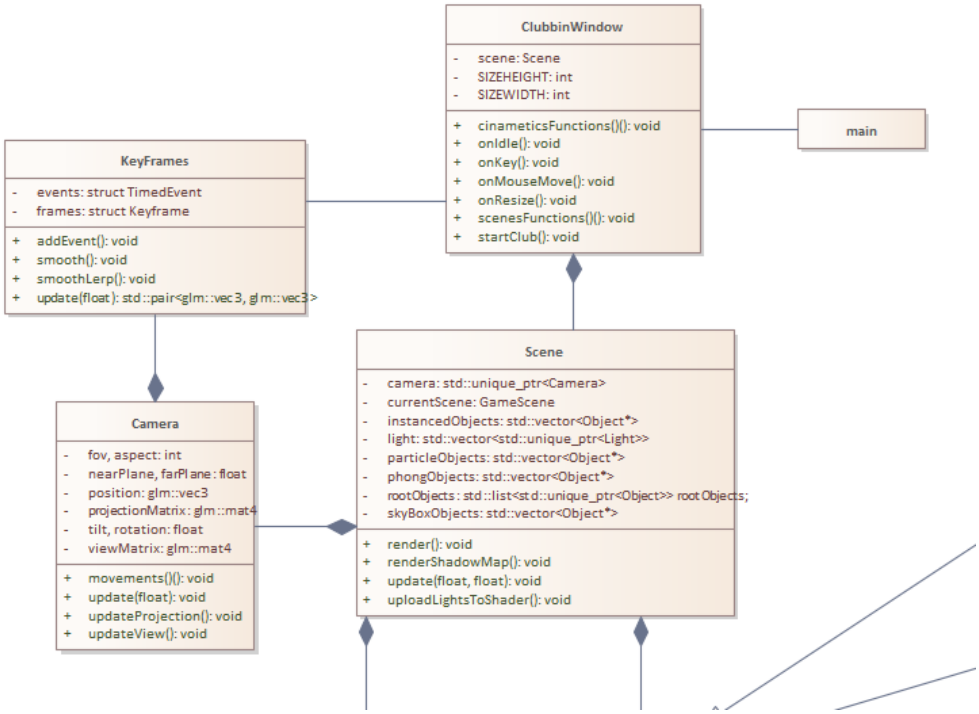

Fig 23. Class Diagram First Part

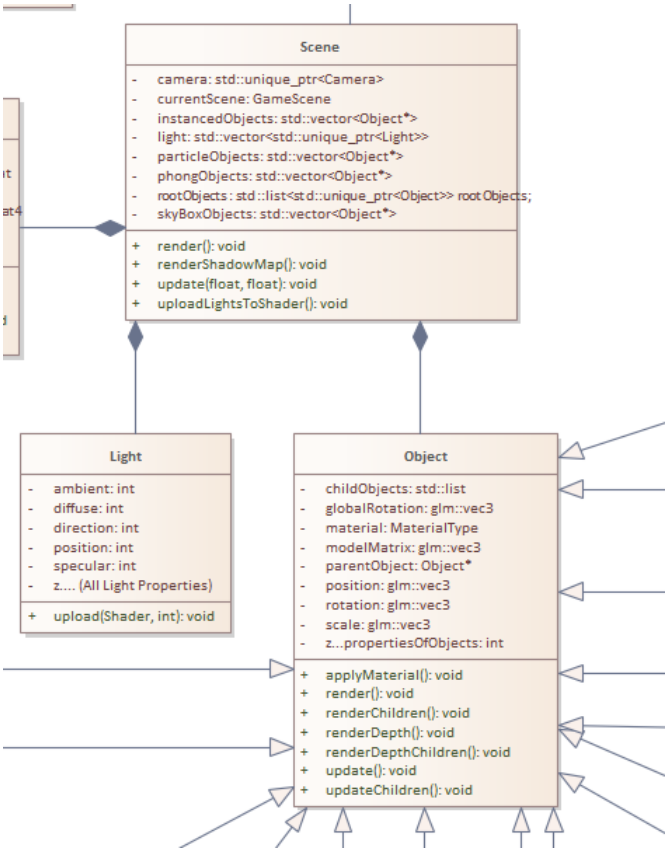

Fig 23. Class Diagram Second Part

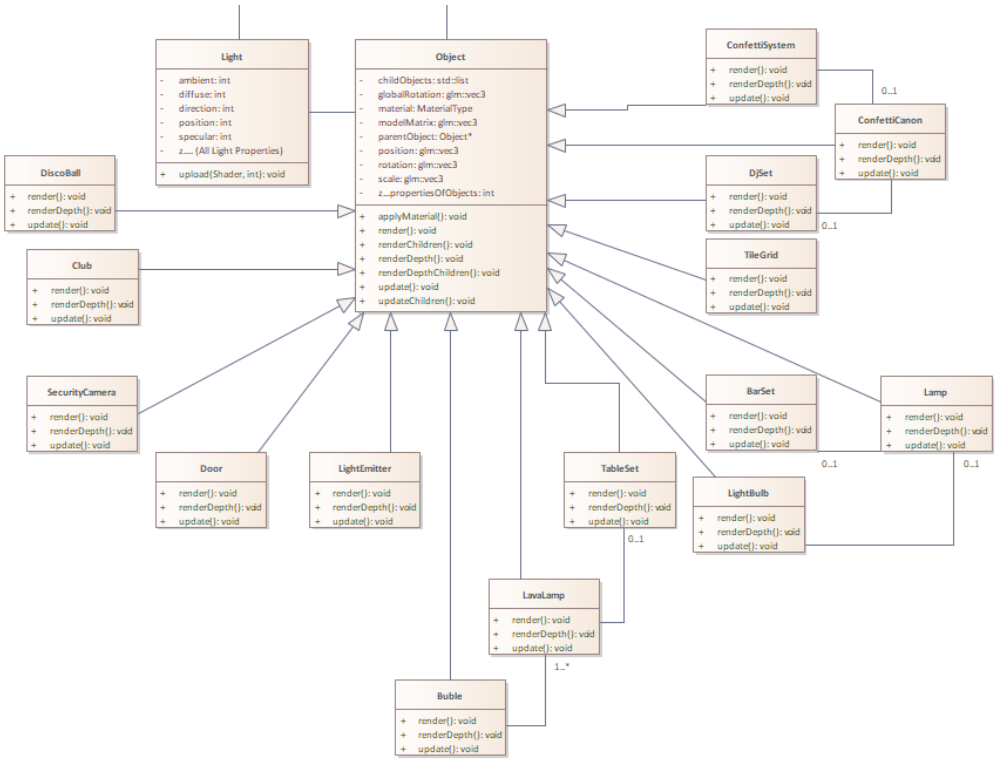

Fig 24. Class Diagram Last Part

5. Sequence of Images Corresponding the Storyboard

5.1 Scene 1- Entrance

Fig 25. Observing Sky Box Street.

Fig 26. Observing Neon Lady

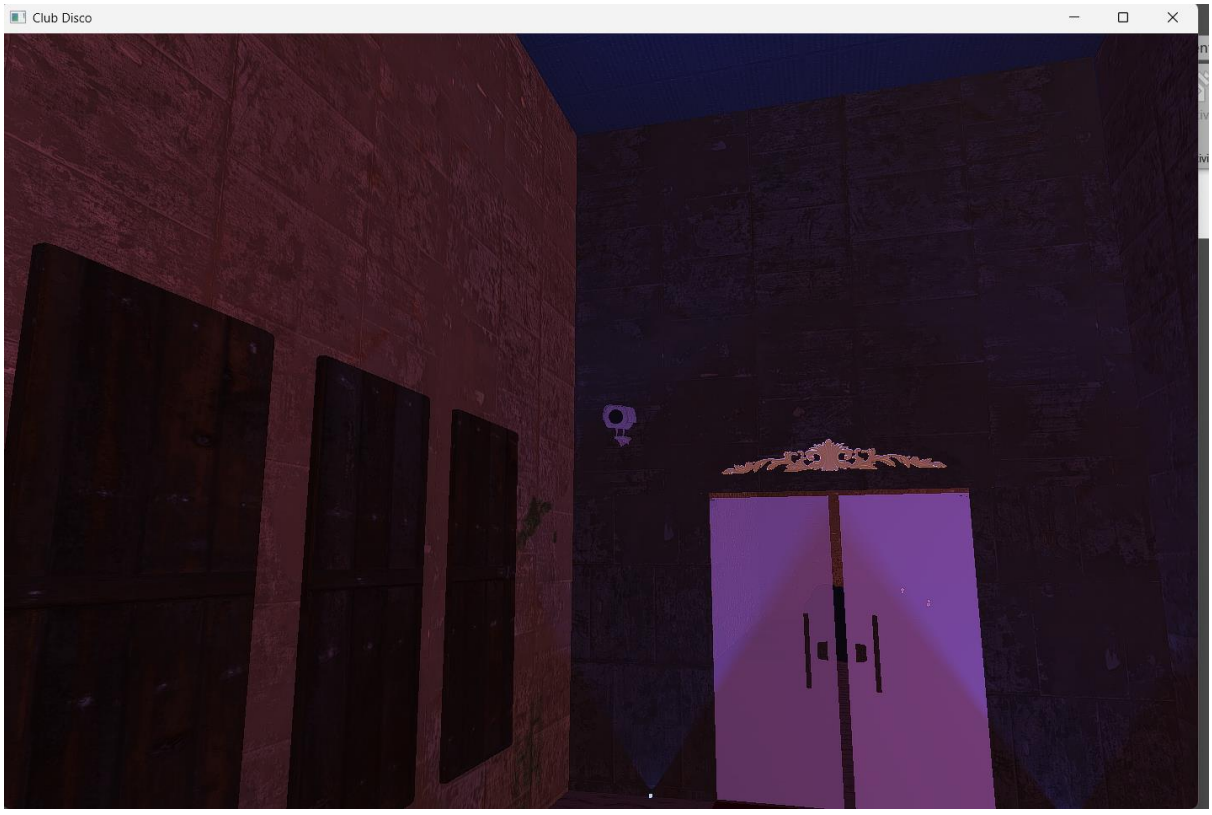

Fig 27. Observing Camera from pov far

Fig 28. Cinematic camera pointing us in front of door.

Fig 29. Doors Opening

5.2 Scene 2- Interior

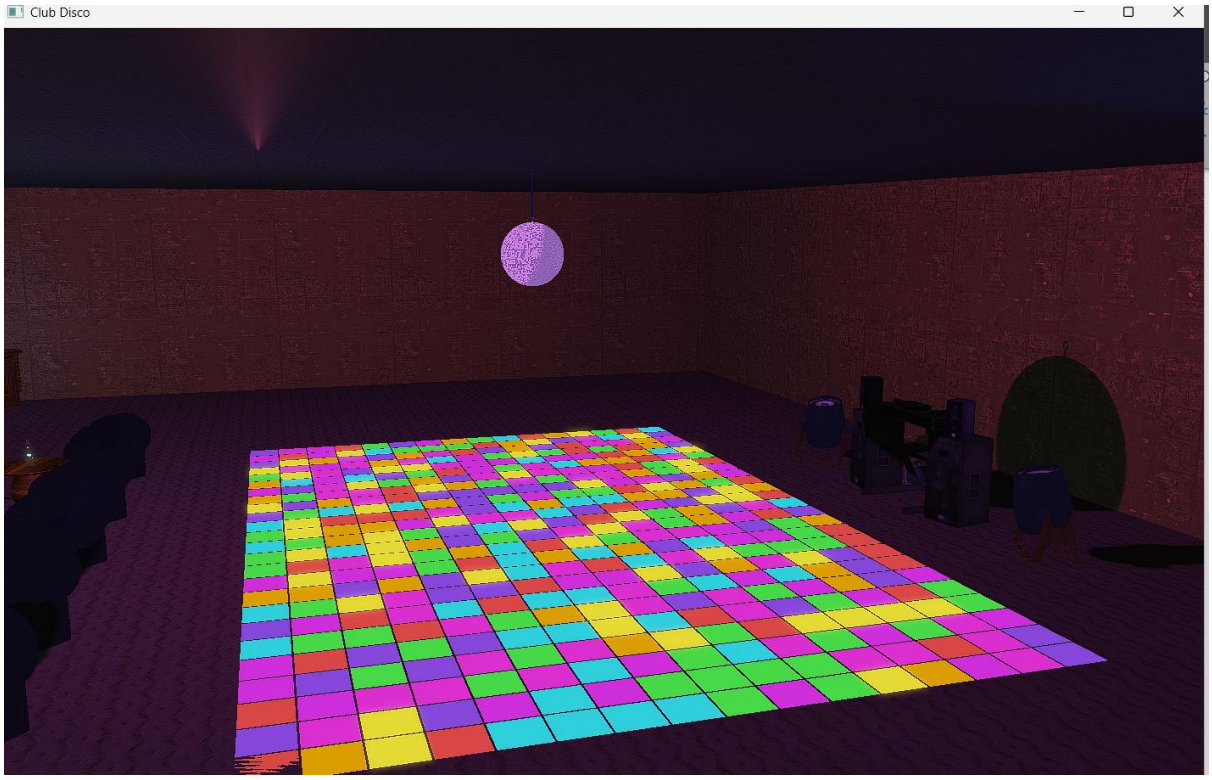

Fig 30. Observing Sky Box Street.

Fig 31. Observing TableSet from Top

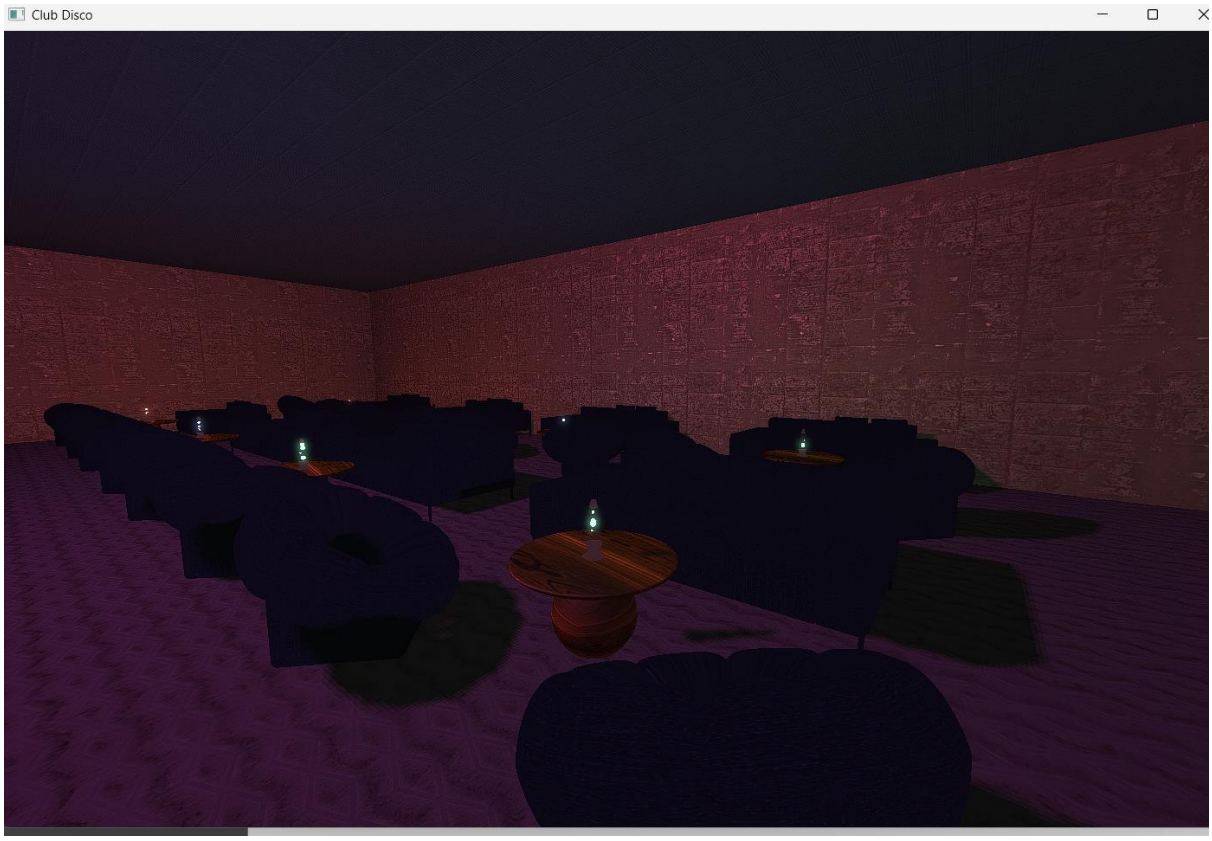

Fig 32. Observing Table Set after moving camera

Fig 33. Observing Lava lamp

Fig 34. Observing Lamp in the bar

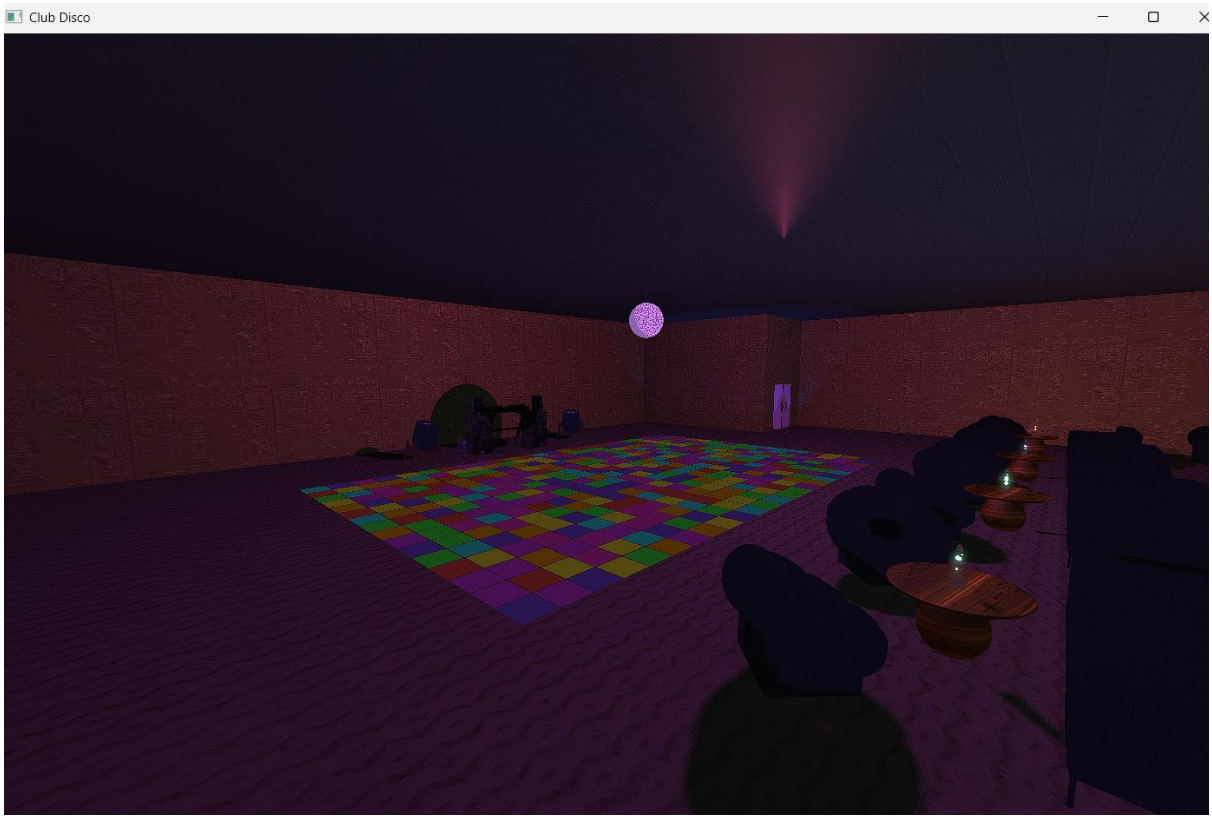

Fig 35. Observing Dance floor from the Bar

5.3 Scene 3 – Dance Floor

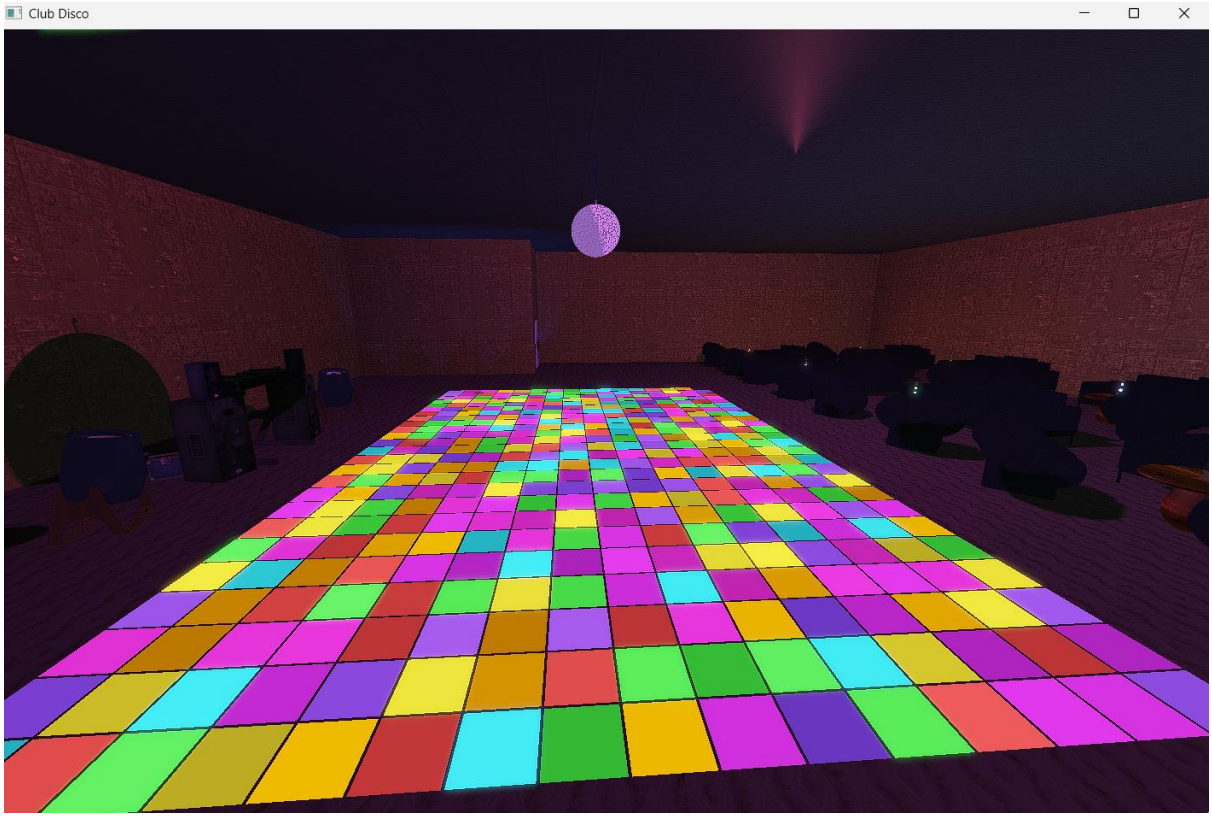

Fig 36. Observing Dance floor from the side

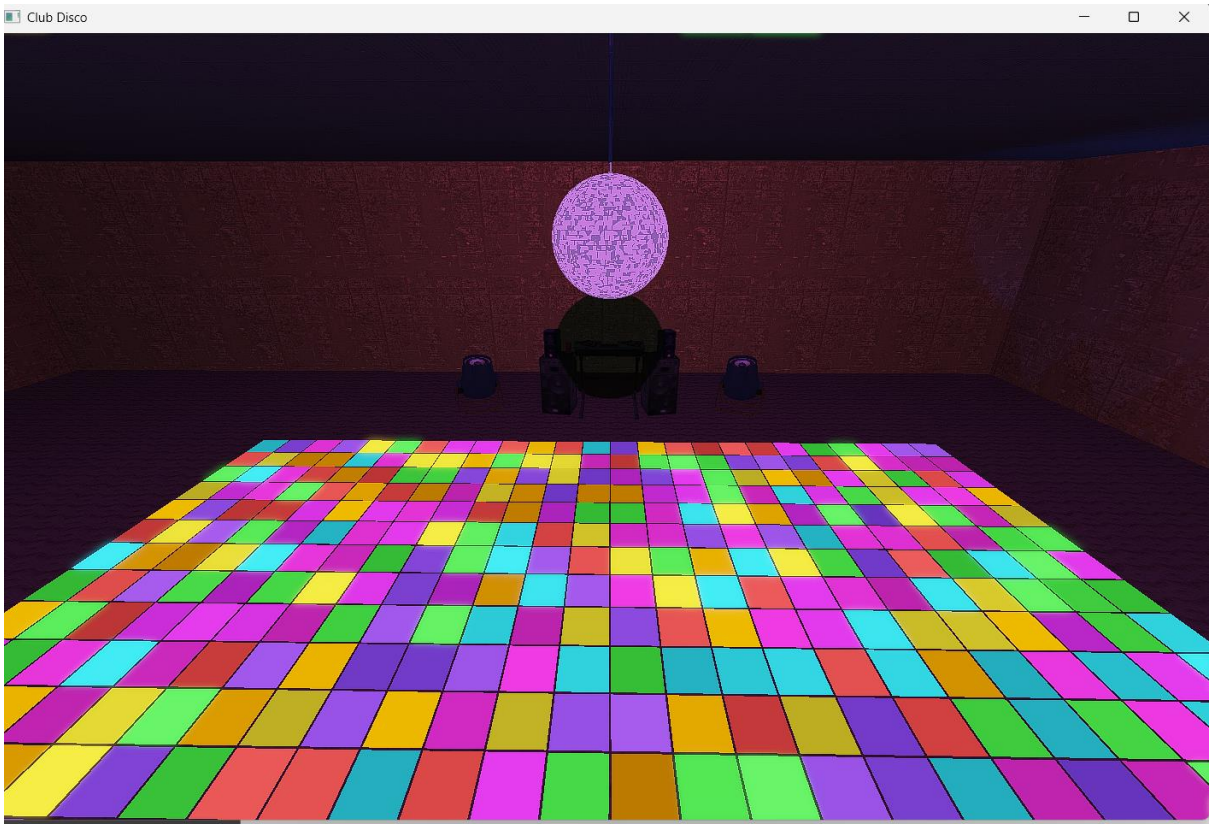

Fig 37. Cinematic Disco Ball and tiles from top

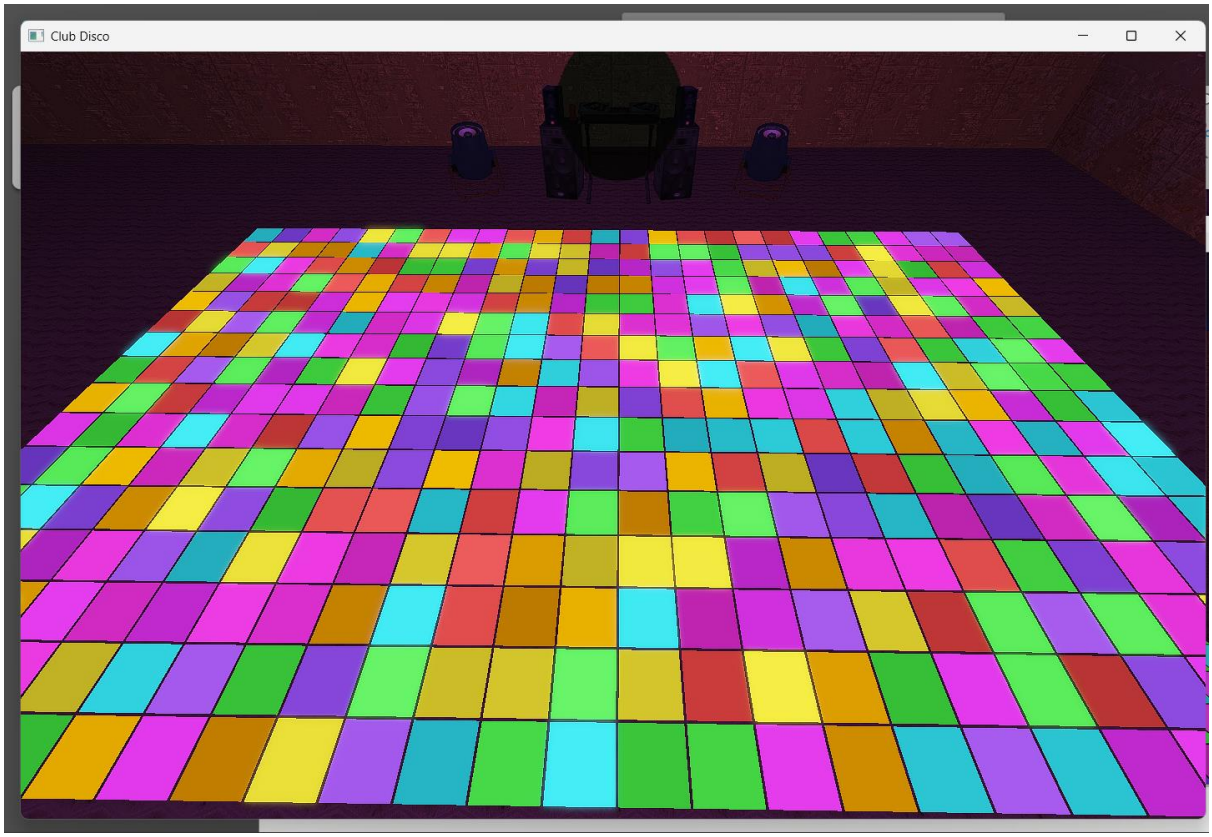

Fig 38. Dance Floor Tiles from top

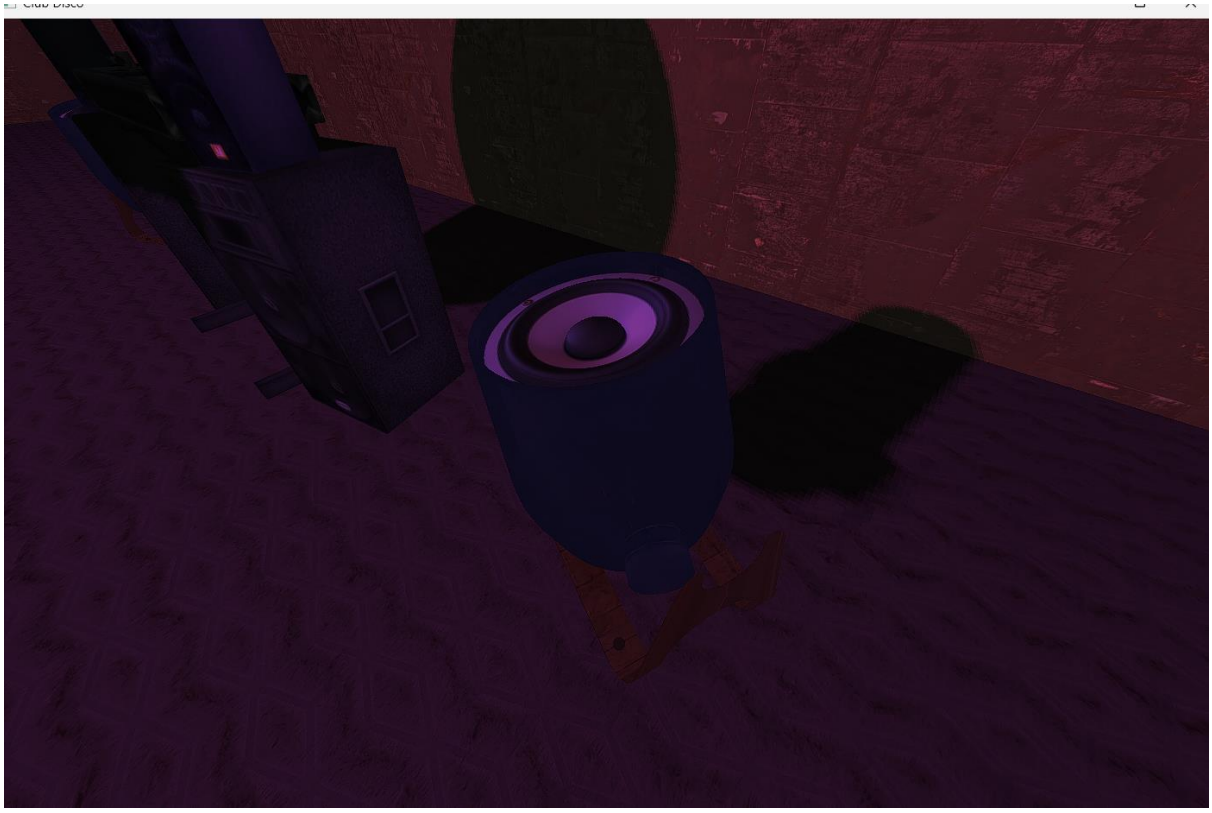

Fig 39. Observing Canon Left

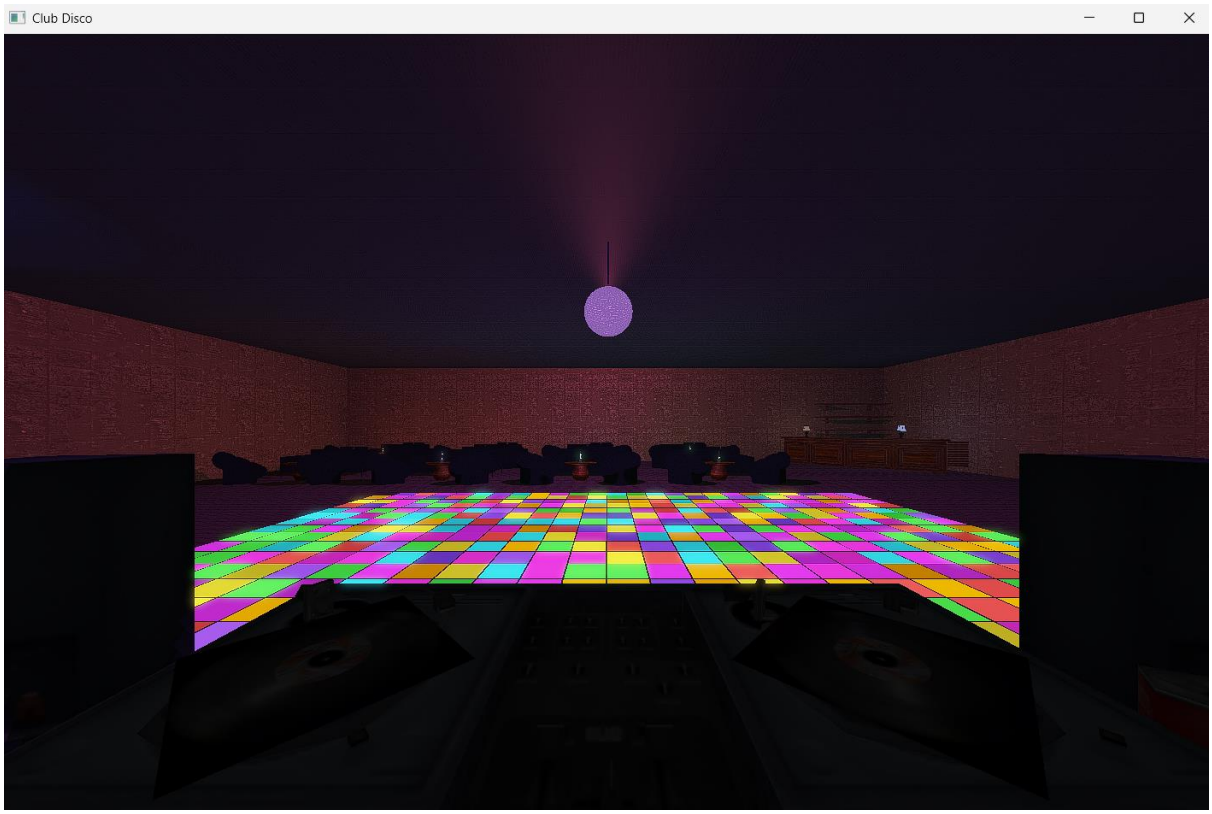

Fig 40. Observing from "Dj" point of view

Fig 41. Panoramic of the club from the dj set

5.4 Scene 4 – Chaotic Ending

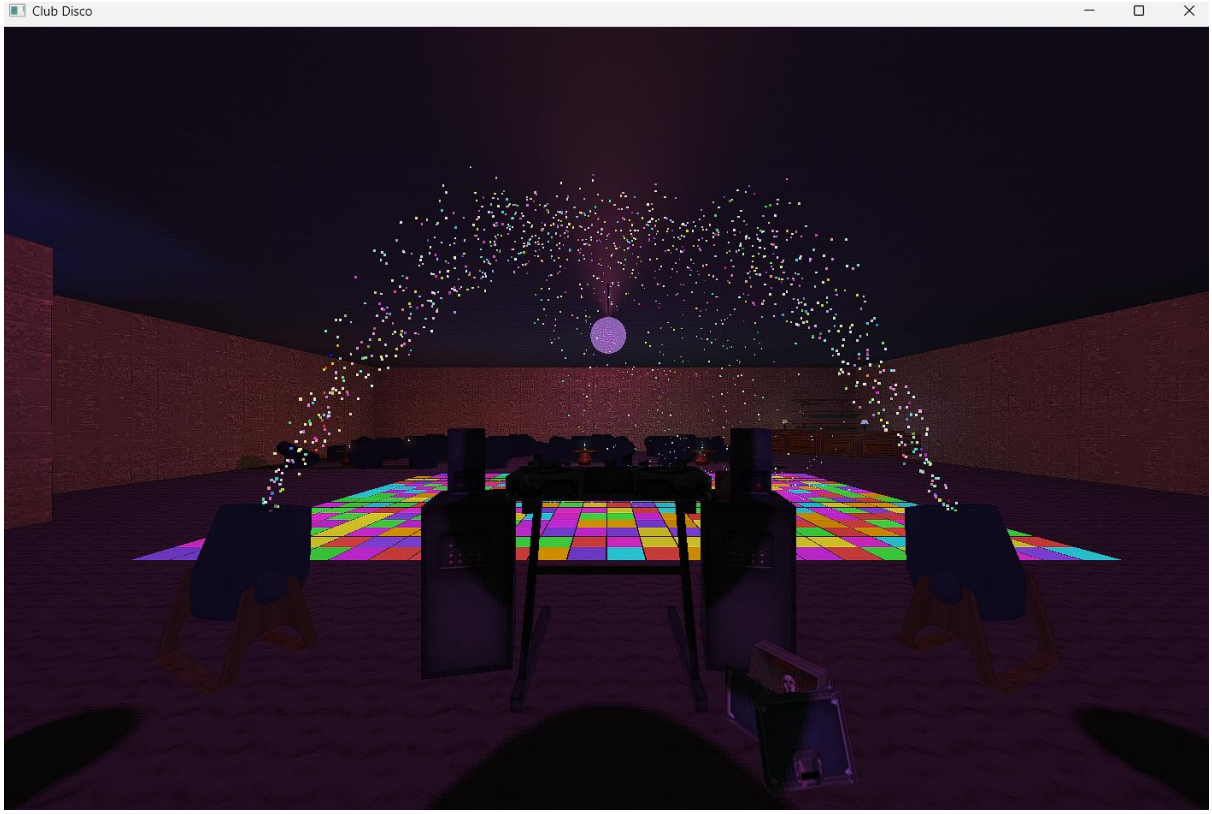

Fig 42. Observing Confetti Cannon shooting confetti

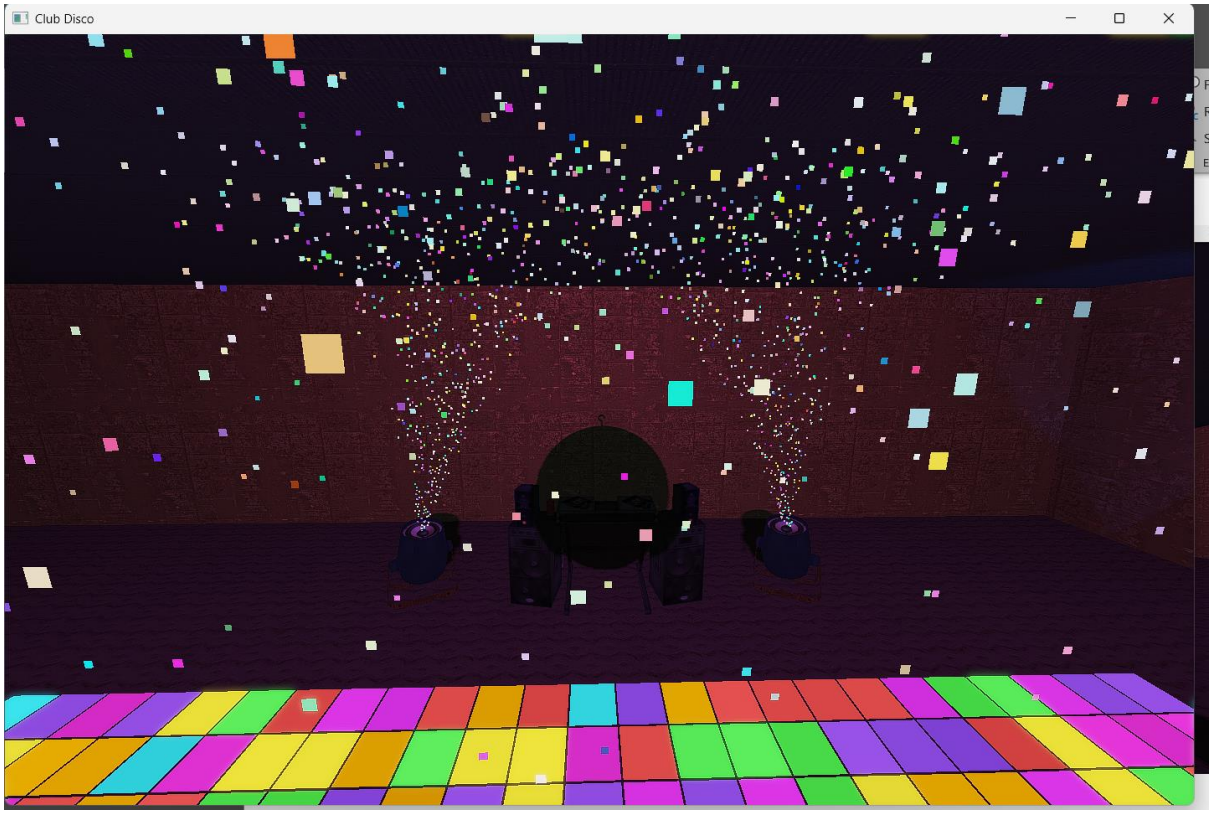

Fig 43. Observing DjSet and confetti falling from top view

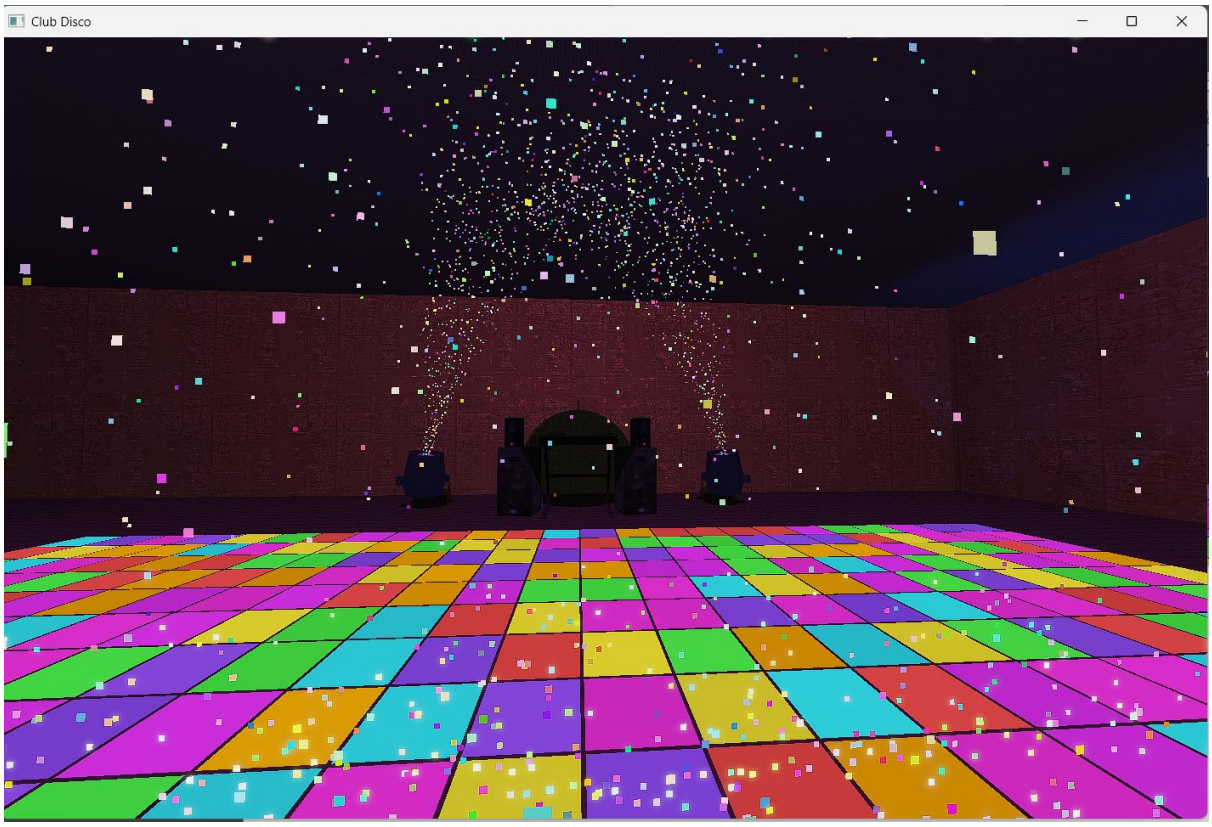

Fig 44. Observing DjSet and confetti falling from ground view