Computer Graphics (UCS505)

3D Space Shooter Game

Branch

B.E. 3rd Year – CSE

Submitted By –

Vivіktha Vujagiri 102217035

Submitted To –

Mr. Armaan Garg

Computer Science and Engineering Department

Thapar Institute of Engineering and Technology

Patiala – 147001

INTRODUCTION

Project Overview:

The **3D Space Shooter Game** is a real-time, arcade-style shooting game built using **C++**, **OpenGL**, **FreeGLUT**, and **GLEW**. Inspired by the classic **arcade space shooter games** from the golden age of video gaming, this project brings that nostalgic gameplay into a modern 3D environment. Players control a spaceship navigating through space, avoiding and destroying enemy ships while maintaining health and earning points.

The game combines fundamental computer graphics principles with gameplay logic to simulate an engaging space battle. The design follows a modular, object-oriented approach, with each gameplay component—player, enemies, bullets, and game logic—encapsulated in separate classes for clarity and scalability.

Key Highlights:

- **Smooth Movement & Controls**: Responsive player controls and fluid navigation through 3D space.
- **Enemy Movement**: Enemies approach the player, and shoot in controlled bursts.
- **Health System**: Player has an energy score that changes colour based on health of the player.
- **Real-time Gameplay**: Fast-paced action with collision detection, shooting logic, and scoring.
- **Dynamic Environment**: Moving star background to simulate motion through space and enhance immersion.
- **Aiming System**: Player is enabled to target the enemies in order to shoot bullets.

Scope of the Project:

The scope of the **3D Space Shooter Game** project includes the design, development, and demonstration of a simple but functional 3D arcade-style space shooting game. The game is developed using **C++ with OpenGL**, along with **FreeGLUT** and **GLEW** libraries to handle rendering and input.
This project aims to simulate key elements of a classic space shooter while introducing fundamental concepts of 3D game development and computer graphics. The scope includes:

In-Scope Features:

- **3D Player Ship Movement**: The player can move their spaceship in multiple directions within the game window
- **Shooting Mechanism**: The player can shoot bullets at enemy cubes.

- **Enemy Behaviour**: Enemies spawn, move toward the player and shoot bullets in bursts at the player.
- **Collision Detection**: Detects collisions between bullets and the player or the enemy cubes, triggering damage.
- **Health System**:
    - The Player has a health score that tracks the energy of the Player Cone.
    - Health score changes colour based on remaining health (i.e., green to red).
- **Score System**: Tracks the number of enemies destroyed (100 points per enemy).
- **Game State Management**: Handles game running and defeat states.
- **3D Rendering**: Objects are rendered in a 3D environment with visual cues like stars for background motion.
- **Modular Codebase**: Clean separation of game logic and object behaviour using OOP principles.

Out-of-Scope (Currently):

- Advanced environment (e.g., asteroids, debris, etc)
- Infinite Game Loop and no actual 'Victory' stage.
- Complex enemy types or power-ups.
- Advanced 3D models (currently uses simple shapes for ships and bullets).
- Audio (music and sound effects).
- Saving/loading game progress or levels.

USER DEFINED FUNCTIONS

S No.	Source file	Function Name	Function Description
1	Bullet.cpp	void draw()	Renders the bullet as a small 3D sphere using OpenGL. The color is yellow for player bullets and orange for enemy bullets.
2		void update()	Updates the bullet's position by moving it in its current direction, scaled by its speed. Called each frame.
3		void setDirection(float xDir, float yDir, float zDir)	Sets the bullet's movement direction and normalizes it to maintain consistent movement speed regardless of vector length.
4	Enemy.cpp	void draw()	Renders the enemy as a red cube using OpenGL. Material properties (ambient, diffuse, specular, shininess) are set to give it a polished appearance.
5		void update()	Moves the enemy forward along the Z-axis (toward the player), adds slight random horizontal movement (X-axis), and increments a timer used to control when the enemy shoots.

SNo.	Source file	Function Name	Function Description
6	Game.cpp	**void Game::InitStars(int count)**	Initializes the starfield background. It clears any existing stars and then populates the stars vector with count randomly positioned stars based on a fixed field of view, aspect ratio, and depth range.
7		**void Game::init()**	Sets up the projection and view matrices and enables OpenGL features such as depth testing and lighting. It also seeds the random number generator, then calls InitStars(300) to initialize the star background.
8		**void Game::drawBackground()**	Renders the starfield background by drawing each star as a point. It temporarily disables lighting and enables blending to render semi-transparent white points, then restores the lighting state.
9		**void Game::renderText(float x, float y, const char* text)**	A helper function that renders bitmap text on the screen at the specified 2D screen coordinates (x, y). It switches to an orthographic projection to draw the text and then restores the previous projection settings.
10		**void Game::screenToWorld(int screenX, int screenY, float &worldX, float &worldY)**	Converts 2D screen coordinates to 3D world coordinates for improved aiming precision. It obtains the current viewport, modelview, and projection matrices and uses two depth values (near and far planes) to construct a ray. The intersection with a target z-plane (set to –10.0) is computed to yield the world coordinates.
11		**void Game::display()**	Responsible for drawing the complete game scene. It clears the buffers, sets the camera with gluLookAt, renders the background stars, draws the player (including setting the aim based on mouse position), then draws all bullets, enemies, and UI elements (energy, score, and game over messages), and finally swaps the buffers.
12		**bool Game::checkCollision(const Bullet& bullet, const Enemy& enemy)**	Checks for collision between a bullet and an enemy using a simple sphere-sphere (distance-based) collision detection. It calculates the squared distance between the bullet and enemy centers and compares it against the squared sum of their collision radii.
15		**bool Game::checkPlayerCollision(const Bullet& bullet)**	Checks if an enemy bullet has hit the player by measuring the distance between the bullet and the player's position. A smaller collision radius is used to make it difficult.
14		**void Game::handleCollisions()**	Iterates over all bullets to check for collisions. Depending on whether a bullet is an enemy bullet or a player bullet, it either decreases the player's energy or inflicts

			damage on an enemy. Colliding bullets (and destroyed enemies) are marked for removal, and then moved in reverse order from their respective vectors to prevent invalidation.
15.		**void Game::update(int value)**	Updates the game state if the game is not over. It updates the starfield (by moving stars and reinitializing them as needed), spawns new enemies based on a counter, updates all bullets and enemies (including enemy shooting behavior), handles collisions, and removes off-screen objects. It ends by posting a redisplay and setting a timer callback to update the game state in approximately 16 ms.
16		**void Game::handleKeys(unsigned char key, int x, int y)** .	Processes keyboard input. If the game is over, only the 'R' key is checked to reset the game. Otherwise, it adjusts the player's movement based on the key pressed (e.g., 'w', 'a', 's', 'd', 'q', 'e') and triggers a bullet shot on the spacebar press.
17		**void Game::handleMouseMove(int x, int y)**	Updates the internal mouse position (mouseX and mouseY) when the mouse moves. This information is used to calculate the aim direction for the player.
18		**void Game::handleMouseClick(int button, int state, int x, int y)**	Handles mouse click events. If the left mouse button is pressed (and the game is not over), it triggers the player to shoot a bullet.
19		**void Game::shootBullet()**	Creates and configures a bullet fired by the player. It calculates the direction from the player's position to the mouse (converted to world coordinates), sets the bullet's speed and direction, and then adds the bullet to the bullets vector.
20		**void Game::resetGame()**	Resets the entire game state by clearing all enemy and bullet objects, reinitializing the player, resetting the energy, score, and game over state, and reinitializing the starfield background via InitStars(300).
21	**Player.cpp**	draw() .	Renders the player's ship with a rotated green cone and base sphere.
22		moveLeft(float speed)	Moves the player left, clamped within the left boundary.
23		moveRight(float speed)	Moves the player right, clamped within the right boundary.
24		moveUp(float speed)	Moves the player up, limited by the top boundary.
25		moveDown(float speed)	Moves the player down, limited by the bottom boundary.
26		moveForward(float speed)	Moves the player forward (along -Z), within a forward boundary.

27		moveBackward(float speed)	Moves the player backward (along +Z), within a backward boundary.
28		setAimDirection(float dirX, dirY, dirZ) .	Sets and normalizes the player's aiming direction.
29		getAimDirection(float& dirX, dirY, dirZ) .	Retrieves the current normalized aiming direction vector.
30	**Main.cpp**	display()	Calls the Game::display() method to render the game scene.
31		update(int value)	Calls the Game::update() method for game logic and sets a recurring timer.
32		keyboard(unsigned char key, int x, int y)	Handles keyboard input by calling Game::handleKeys().
33		mouseMove(int x, int y)	Tracks mouse movement by calling Game::handleMouseMove().
34		mouseClick(int button, int state, int x, int y)	Handles mouse click events via Game::handleMouseClick().
35		reshape(int width, int height)	Adjusts the OpenGL viewport and perspective when the window is resized.
36		main(int argc, char** argv)	Initializes GLUT, sets up callbacks, and starts the main game loop.

CODE SNIPPETS

- Bullet.h File

Project5 (Global S

```
#ifndef BULLET_H
#define BULLET_H
class Bullet {
private:
    float x;
    float y;
    float z;
    float dirX;
    float dirY;
    float dirZ;
    float speed;
    bool enemyBullet;
public:
    // Constructors
    Bullet();
    Bullet(float xPos, float yPos, float zPos, float bulletSpeed, bool isEnemy);
    Bullet(float xPos, float yPos, float zPos, bool isEnemy); // 4-argument constructor
    // Getters
    float getX() const { return x; }
    float getY() const { return y; }
    float getZ() const { return z; }
    float getDirX() const { return dirX; }
    float getDirY() const { return dirY; }
    float getDirZ() const { return dirZ; }
    float getSpeed() const { return speed; }
    bool isEnemyBullet() const { return enemyBullet; }
    // Setters
    void setX(float xPos) { x = xPos; }
    void setY(float yPos) { y = yPos; }
    void setZ(float zPos) { z = zPos; }
    void setSpeed(float newSpeed) { speed = newSpeed; }
    void setEnemyBullet(bool isEnemy) { enemyBullet = isEnemy; }
    // Methods
    void draw();
    void update();
    void setDirection(float xDir, float yDir, float zDir);
};
#endif
```

- Enemy.h File

```
#ifndef ENEMY_H
#define ENEMY_H
class Enemy {
private:
    float x;
    float y;
    float z;
    int energy;
    int shootTimer;
public:
    // Constructor
    Enemy(float startX, float startY, float startZ, int startEnergy = 100)
        : x(startX), y(startY), z(startZ), energy(startEnergy), shootTimer(0) {
    }
    // Drawing method
    void draw();
    // Update method for movement
    void update();
    // Shooting methods
    bool canShoot() const { return shootTimer >= 100; }
    void resetShootTimer() { shootTimer = 0; }
    // Energy methods
    void decreaseEnergy(int amount) { energy -= amount; }
    bool isDestroyed() const { return energy <= 0; }
    // Getter methods
    float getX() const { return x; }
    float getY() const { return y; }
    float getZ() const { return z; }
};
#endif
```

- Player.h File

```
#ifndef PLAYER_H
#define PLAYER_H
class Player {
private:
    float x;
    float y;
    float z;
    // Aiming direction
    float aimDirX;
    float aimDirY;
    float aimDirZ;
public:
    // Constructor
    Player(float startX = 0.0f, float startY = 0.0f, float startZ = 0.0f)
        : x(startX), y(startY), z(startZ), aimDirX(0.0f), aimDirY(0.0f), aimDirZ(-1.0f) {
    }
    // Drawing method
    void draw();
    // Movement methods
    void moveLeft(float speed = 0.1f);
    void moveRight(float speed = 0.1f);
    void moveUp(float speed = 0.1f);
    void moveDown(float speed = 0.1f);
    void moveForward(float speed = 0.1f);
    void moveBackward(float speed = 0.1f);
    // Aiming methods
    void setAimDirection(float dirX, float dirY, float dirZ);
    void getAimDirection(float& dirX, float& dirY, float& dirZ) const;
    // Getter methods
    float getX() const { return x; }
    float getY() const { return y; }
    float getZ() const { return z; }
};
#endif
```

- Game.h File

```h
#ifndef GAME_H
#define GAME_H
#include "Player.h"
#include "Bullet.h"
#include "Enemy.h"
#include <vector>
#include <cstddef> // For size_t
struct Star {
    float x;
    float y;
    float z;
};
class Game {
private:
    Player player;
    std::vector<Bullet> bullets;
    std::vector<Enemy> enemies;
    std::vector<Star> stars;
    int enemySpawnCounter;
    int playerEnergy;
    int score;
    bool gameOver;

    int mouseX;
    int mouseY;
    // Helper methods
    void drawBackground();
    void renderText(float x, float y, const char* text);
    bool checkCollision(const Bullet& bullet, const Enemy& enemy);
    bool checkPlayerCollision(const Bullet& bullet);
    void handleCollisions();
    void InitStars(int count);
    void screenToWorld(int screenX, int screenY, float& worldX, float& worldY);
    void shootBullet();
public:
    // Constructor
    Game();
    void init();
    // Main game functions
    void display();
    void update(int value);
    void handleKeys(unsigned char key, int x, int y);
    void handleMouseMove(int x, int y);
    void handleMouseClick(int button, int state, int x, int y);
    void resetGame();
};
#endif
```

- Bullet.cpp File

```cpp
#include <GL/freeglut.h>
#include "Bullet.h"
#include <cmath>
// Default constructor
Bullet::Bullet() : x(0), y(0), z(0), dirX(0), dirY(0), dirZ(0), speed(0), enemyBullet(false) {
}
// Constructor with speed parameter
Bullet::Bullet(float xPos, float yPos, float zPos, float bulletSpeed, bool isEnemy)
    : x(xPos), y(yPos), z(zPos), dirX(0), dirY(0), dirZ(0), speed(bulletSpeed), enemyBullet(isEnemy) {
}
// 4-parameter constructor (for the version being called with 4 args)
Bullet::Bullet(float xPos, float yPos, float zPos, bool isEnemy)
    : x(xPos), y(yPos), z(zPos), dirX(0), dirY(0), dirZ(0), speed(5.0f), enemyBullet(isEnemy) {
    // Using default speed of 5.0 when not specified
}
void Bullet::draw() {
    glPushMatrix();
    glTranslatef(x, y, z);
    if (enemyBullet) {
        glColor3f(1.0f, 0.5f, 0.0f);  // Orange
    }
    else {
        glColor3f(1.0f, 1.0f, 0.0f);  // Yellow
    }
    glutSolidSphere(0.1, 10, 10);  // sphere
    glPopMatrix();
}
void Bullet::update() {
    // Move bullet based on its direction and speed
    x += dirX * speed;
    y += dirY * speed;
    z += dirZ * speed;
}
void Bullet::setDirection(float xDir, float yDir, float zDir) {
    dirX = xDir;
    dirY = yDir;
    dirZ = zDir;
    float length = std::sqrt(dirX * dirX + dirY * dirY + dirZ * dirZ);
    if (length > 0) {
        dirX /= length;
        dirY /= length;
        dirZ /= length;
    }
}
```

- Enemy.cpp FIle

```cpp
#include "Enemy.h"
#include <GL/freeglut.h>
#include <cstdlib> // For rand()
void Enemy::draw() {
    glPushMatrix();
    glTranslatef(x, y, z);
    // Add some material properties to make the cube look better
    GLfloat mat_ambient[] = { 0.8f, 0.0f, 0.0f, 1.0f };
    GLfloat mat_diffuse[] = { 1.0f, 0.2f, 0.2f, 1.0f };
    GLfloat mat_specular[] = { 1.0f, 0.5f, 0.5f, 1.0f };
    GLfloat mat_shininess[] = { 50.0f };
    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);

    glutSolidCube(2.0f);
    glPopMatrix();
}
void Enemy::update() {
    // Move enemy toward the player (positive Z direction)..... meaning we have right hand coordinate system
    z += 0.1f;
    // Slight random sideways movement
    x += (((rand() % 100) - 50) / 1000.0f);
    // Increment shoot timer
    shootTimer++;
}
```

- Game.cpp File

```cpp
Game::Game()
    : player(0.0f, -0.8f),
    enemySpawnCounter(0),
    playerEnergy(100),
    score(0),
    gameOver(false),
    mouseX(0),
    mouseY(0) {
}

void Game::InitStars(int count) {
    stars.clear();
    float fov = 60.0f;
    float aspect = 800.0f / 600.0f;
    float maxDepth = 100.0f;
    float minDepth = 1.0f;

    for (int i = 0; i < count; ++i) {
        Star s;
        s.z = -((rand() / (float)RAND_MAX) * (maxDepth - minDepth) + minDepth);
        float halfHeight = tan(fov * 0.5f * 3.14159f / 180.0f) * -s.z;
        float halfWidth = halfHeight * aspect;
        s.x = ((rand() / (float)RAND_MAX) * 2 - 1) * halfWidth;
        s.y = ((rand() / (float)RAND_MAX) * 2 - 1) * halfHeight;
        stars.push_back(s);
    }
}

void Game::init() {
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, 800.0 / 600.0, 0.1, 200.0);
    glMatrixMode(GL_MODELVIEW);

    glEnable(GL_DEPTH_TEST);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);

    GLfloat light_pos[] = { 0.0f, 0.0f, 5.0f, 1.0f };
    glLightfv(GL_LIGHT0, GL_POSITION, light_pos);

    srand(static_cast<unsigned int>(time(0)));
    InitStars(300);
}
```

```cpp
void Game::drawBackground() {
    if (stars.empty()) return;

    glDisable(GL_LIGHTING);
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

    glPointSize(1.5f);
    glBegin(GL_POINTS);
    glColor4f(1.0f, 1.0f, 1.0f, 0.5f);
    for (const auto& star : stars) {
        glVertex3f(star.x, star.y, star.z);
    }
    glEnd();

    glDisable(GL_BLEND);
    glEnable(GL_LIGHTING);
}

// Helper function to render text on screen
void Game::renderText(float x, float y, const char* text) {
    glDisable(GL_LIGHTING);
    glDisable(GL_DEPTH_TEST);

    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    gluOrtho2D(0, 800, 0, 600);

    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glLoadIdentity();

    glRasterPos2f(x, y);
    for (const char* c = text; *c != '\0'; c++) {
        glutBitmapCharacter(GLUT_BITMAP_HELVETICA_18, *c);
    }

    glPopMatrix();
    glMatrixMode(GL_PROJECTION);
    glPopMatrix();
    glMatrixMode(GL_MODELVIEW);

    glEnable(GL_DEPTH_TEST);
    glEnable(GL_LIGHTING);
}
```

```cpp
bool Game::checkCollision(const Bullet& bullet, const Enemy& enemy) {
    // Simple distance-based collision detection (sphere-sphere)
    float bulletX = bullet.getX();
    float bulletY = bullet.getY();
    float bulletZ = bullet.getZ();

    float enemyX = enemy.getX();
    float enemyY = enemy.getY();
    float enemyZ = enemy.getZ();

    float dx = bulletX - enemyX;
    float dy = bulletY - enemyY;
    float dz = bulletZ - enemyZ;
    float distanceSquared = dx * dx + dy * dy + dz * dz;

    // Increased collision radius to make hitting enemies easier
    float collisionRadiusSum = 0.2f + 0.6f; // Bullet radius + half enemy size (increased for easier hits)

    return distanceSquared < (collisionRadiusSum * collisionRadiusSum);
}

bool Game::checkPlayerCollision(const Bullet& bullet) {
    // Check if an enemy bullet hit the player
    float bulletX = bullet.getX();
    float bulletY = bullet.getY();
    float bulletZ = bullet.getZ();

    float playerX = player.getX();
    float playerY = player.getY();
    float playerZ = player.getZ();

    float dx = bulletX - playerX;
    float dy = bulletY - playerY;
    float dz = bulletZ - playerZ;
    float distanceSquared = dx * dx + dy * dy + dz * dz;

    // Make player a bit harder to hit by making collision area smaller
    float collisionRadiusSum = 0.1f + 0.08f; // Slightly smaller player hitbox

    return distanceSquared < (collisionRadiusSum * collisionRadiusSum);
}
```

```cpp
void Game::shootBullet() {
    // Get player's current position
    float px = player.getX();
    float py = player.getY();
    float pz = player.getZ();

    // Calculate world coordinates from mouse position directly
    float worldX, worldY;
    screenToWorld(mouseX, mouseY, worldX, worldY);

    // Create a bullet using the player's position
    Bullet bullet(px, py, pz, false); // false = player bullet

    // Calculate direction vector from player to cursor
    float dirX = worldX - px;
    float dirY = worldY - py;
    float dirZ = -10.0f - pz; // Match the target Z used in screenToWor

    // Set the bullet speed higher for better responsiveness
    bullet.setSpeed(12.0f); // Increased from 8.0f for faster bullets

    // Set direction (this will normalize it internally)
    bullet.setDirection(dirX, dirY, dirZ);

    // Add bullet to the list
    bullets.push_back(bullet);
}

void Game::resetGame() {
    // Clear all game objects
    bullets.clear();
    enemies.clear();

    // Reset player
    player = Player(0.0f, -0.8f);

    // Reset game variables
    playerEnergy = 100;
    score = 0;
    gameOver = false;
    enemySpawnCounter = 0;

    // Reinitialize stars
    InitStars(300);
}
```

```cpp
void Game::handleCollisions() {
    std::vector<int> bulletsToRemove;
    std::vector<int> enemiesToRemove;

    // Check player bullets against enemies
    for (int i = 0; i < static_cast<int>(bullets.size()); i++) {
        if (bullets[i].isEnemyBullet()) {
            // Check if enemy bullet hit player
            if (checkPlayerCollision(bullets[i])) {
                playerEnergy -= 10; // Decrease player energy
                bulletsToRemove.push_back(i);

                if (playerEnergy <= 0) {
                    playerEnergy = 0;
                    gameOver = true;
                }
            }
        }
        else {
            // Check player bullets against enemies
            for (int j = 0; j < static_cast<int>(enemies.size()); j++) {
                if (checkCollision(bullets[i], enemies[j])) {
                    bulletsToRemove.push_back(i);
                    // Increased damage to enemies to make them easier to destroy
                    enemies[j].decreaseEnergy(50); // Increased from 25 to 50

                    if (enemies[j].isDestroyed()) {
                        enemiesToRemove.push_back(j);
                        score += 100; // Increase score when enemy is destroyed
                    }
                    break; // A bullet can only hit one enemy
                }
            }
        }
    }

    // Remove bullets (from highest index to lowest to avoid invalidation)
    std::sort(bulletsToRemove.rbegin(), bulletsToRemove.rend());
    for (auto idx : bulletsToRemove) {
        if (idx < static_cast<int>(bullets.size())) {
            bullets.erase(bullets.begin() + idx);
        }
    }

    // Remove enemies (from highest index to lowest to avoid invalidation)
    std::sort(enemiesToRemove.rbegin(), enemiesToRemove.rend());
    for (auto idx : enemiesToRemove) {
```

```cpp
vo
    if (!gameOver) {
        // Update star field
        for (auto& star : stars) {
            star.z += 0.05f;
            if (star.z > 0.0f) {
                float fov = 60.0f;
                float aspect = 800.0f / 600.0f;
                float newZ = -((rand() / (float)RAND_MAX) * (100.0f - 1.0f) + 1.0f);
                float halfHeight = tan(fov * 0.5f * 3.14159f / 180.0f) * -newZ;
                float halfWidth = halfHeight * aspect;

                star.z = newZ;
                star.x = ((rand() / (float)RAND_MAX) * 2 - 1) * halfWidth;
                star.y = ((rand() / (float)RAND_MAX) * 2 - 1) * halfHeight;
            }
        }

        if (stars.size() < 300 && rand() % 10 == 0) {
            float fov = 60.0f;
            float aspect = 800.0f / 600.0f;
            float z = -((rand() / (float)RAND_MAX) * (100.0f - 1.0f) + 1.0f);
            float halfHeight = tan(fov * 0.5f * 3.14159f / 180.0f) * -z;
            float halfWidth = halfHeight * aspect;

            Star s;
            s.z = z;
            s.x = ((rand() / (float)RAND_MAX) * 2 - 1) * halfWidth;
            s.y = ((rand() / (float)RAND_MAX) * 2 - 1) * halfHeight;

            stars.push_back(s);
        }

        enemySpawnCounter++;

        if (enemySpawnCounter >= 1000) {
            float spawnX = ((rand() % 600) - 300) / 100.0f;
            float spawnY = ((rand() % 400) - 200) / 100.0f;
            enemies.push_back(Enemy(spawnX, spawnY, -100.0f));
            enemySpawnCounter = 0;
        }

        for (auto& bullet : bullets) {
            bullet.update();
        }

        for (auto& enemy : enemies) {
            enemy.update();
```

```cpp
        handleCollisions();

        // Remove bullets that are too far
        bullets.erase(std::remove_if(bullets.begin(), bullets.end(),
            [](const Bullet& b) {
                return (b.isEnemyBullet() && b.getZ() > 10.0f) ||
                    (!b.isEnemyBullet() && b.getZ() < -200.0f);
            }), bullets.end());

        // Remove enemies that are too close to camera
        enemies.erase(std::remove_if(enemies.begin(), enemies.end(),
            [](const Enemy& e) { return e.getZ() > 5.0f; }), enemies.end());
    }

    glutPostRedisplay();
    glutTimerFunc(16, [](int val) {
        static Game* g = reinterpret_cast<Game*>(glutGetWindowData());
        if (g) g->update(val);
        }, 0);
}

void Game::handleKeys(unsigned char key, int x, int y) {
    if (gameOver) {
        if (key == 'r' || key == 'R') {
            // Reset game
            resetGame();
        }
        return;
    }

    // Increased movement speed for better player control
    float moveSpeed = 0.15f; // Increased from 0.1f implicit in the movement

    switch (key) {
    case 'w': player.moveUp(moveSpeed); break;
    case 's': player.moveDown(moveSpeed); break;
    case 'a': player.moveLeft(moveSpeed); break;
    case 'd': player.moveRight(moveSpeed); break;
    case 'q': player.moveForward(moveSpeed); break;
    case 'e': player.moveBackward(moveSpeed); break;
    case 32:  // Spacebar - player shoots
        shootBullet();
        break;
    }
    glutPostRedisplay();
}
```

- Player.h File

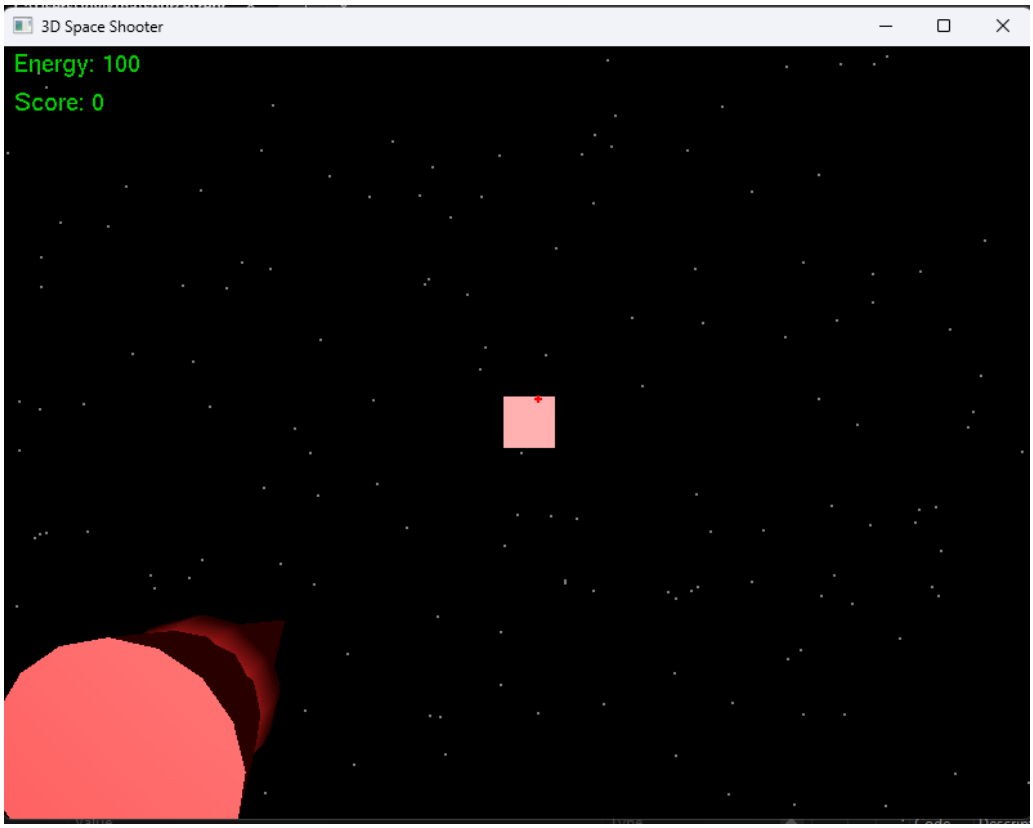
```
#include "Player.h"
#include <GL/freeglut.h>
#include <algorithm> // For std::clamp
#include <cmath> // For vector operations
#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif
void Player::draw() {
    glPushMatrix();
    glTranslatef(x, y, z);
    // Calculate rotation angles based on aim direction
    float rotX, rotY;
    // Calculate the rotation angle around Y-axis (yaw)
    rotY = atan2(aimDirX, aimDirZ) * 180.0f / M_PI;
    // Calculate the rotation angle around X-axis (pitch)
    // We need to calculate the length of the vector projected
    float aimDirLength = sqrt(aimDirX * aimDirX + aimDirZ * ai
    rotX = atan2(aimDirY, aimDirLength) * 180.0f / M_PI;
    // Apply rotations
    glRotatef(rotY, 0.0f, 1.0f, 0.0f); // Yaw rotation (around
    glRotatef(-rotX, 1.0f, 0.0f, 0.0f); // Pitch rotation (aro
    // Draw the player ship base (cone pointing in the -Z dire
    glColor3f(0.0f, 1.0f, 0.0f); // Green color
    glutSolidCone(0.2f, 0.6f, 16, 16); // radius, height, slic
    // Draw a small sphere at the base of the cone to make it
    glTranslatef(0.0f, 0.0f, 0.2f);
    glColor3f(0.0f, 0.8f, 0.0f);
    glutSolidSphere(0.15f, 12, 12);
    glPopMatrix();
}

void Player::moveLeft(float speed) {
    x = std::max(x - speed, -3.5f);
}
void Player::moveRight(float speed) {
    x = std::min(x + speed, 3.5f);
}
void Player::moveUp(float speed) {
    y = std::min(y + speed, 2.5f);
}
```

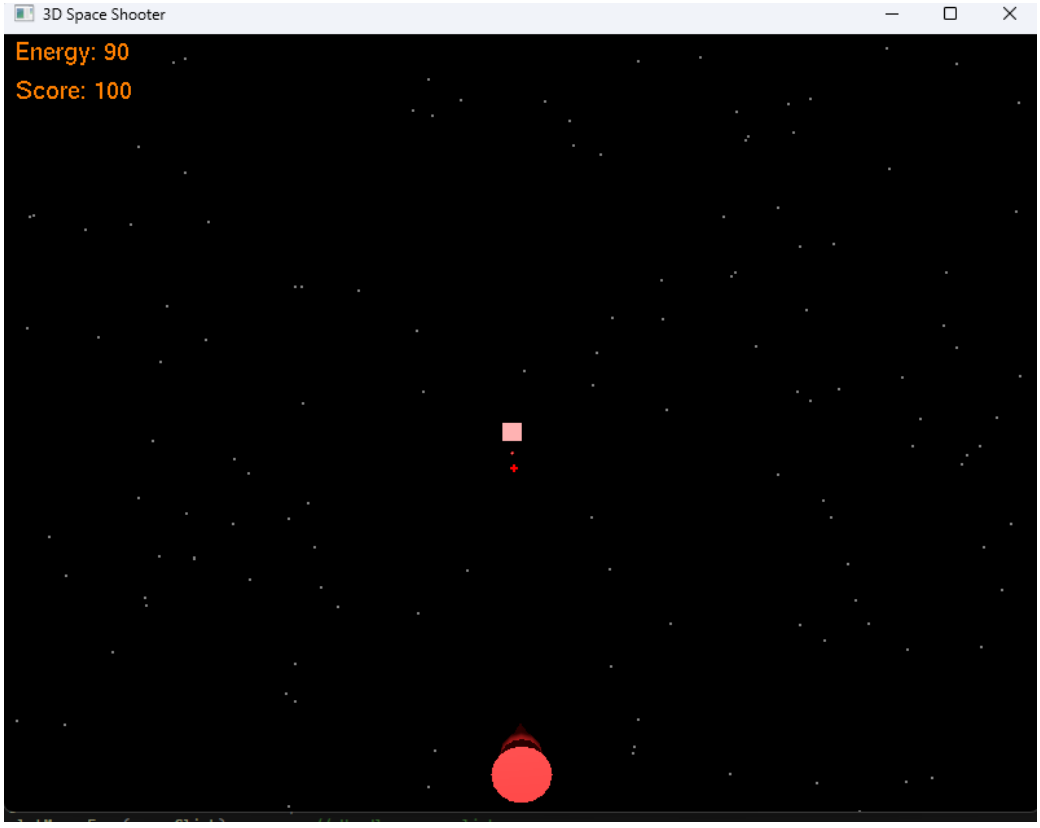
```
void Player::moveDown(float speed) {
    y = std::max(y - speed, -2.5f);
}
void Player::moveForward(float speed) {
    z = std::max(z - speed, -3.0f);
}
void Player::moveBackward(float speed) {
    z = std::min(z + speed, 4.0f);
}
// Set the player's aim direction
void Player::setAimDirection(float dirX, float dirY, float dirZ) {
    aimDirX = dirX;
    aimDirY = dirY;
    aimDirZ = dirZ;
    // Normalize the aim direction
    float length = sqrt(aimDirX * aimDirX + aimDirY * aimDirY + aimDirZ * aimDirZ);
    if (length > 0) {
        aimDirX /= length;
        aimDirY /= length;
        aimDirZ /= length;
    }
}
// Get the player's current aim direction
void Player::getAimDirection(float& dirX, float& dirY, float& dirZ) const {
    dirX = aimDirX;
    dirY = aimDirY;
    dirZ = aimDirZ;
}
```

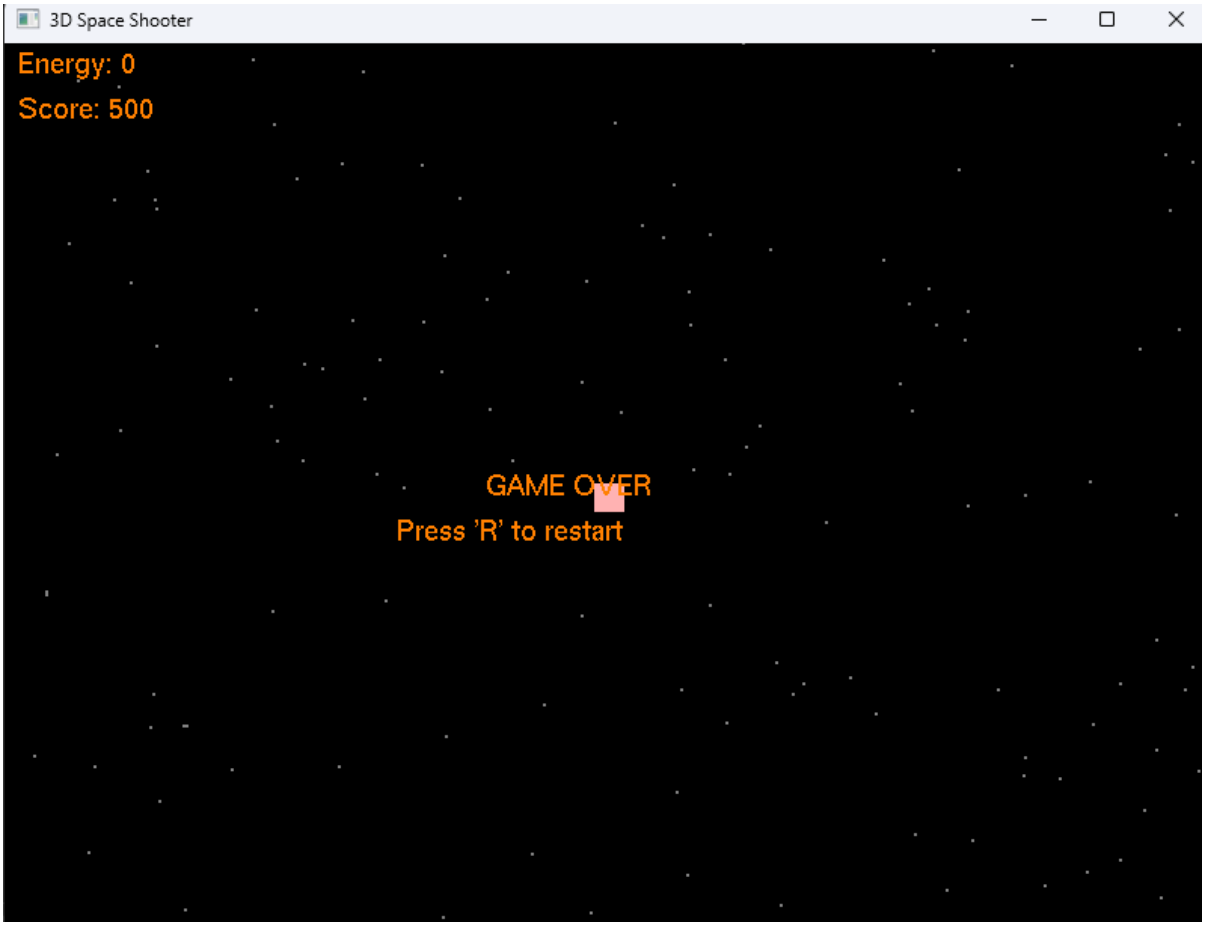
- Main.cpp File

```
#include <GL/freeglut.h>
#include "Game.h"
Game game;
void display() {
    game.display();
}
void update(int value) {
    game.update(value);
}
void keyboard(unsigned char key, int x, int y) {
    game.handleKeys(key, x, y);
}
void mouseMove(int x, int y) {
    game.handleMouseMove(x, y);
}
void mouseClick(int button, int state, int x, int y) {
    game.handleMouseClick(button, state, x, y);
}
void reshape(int width, int height) {
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, (GLfloat)width / (GLfloat)height, 0.1, 200.0);
    glMatrixMode(GL_MODELVIEW);
}
int main(int argc, char** argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(800, 600);
    int window = glutCreateWindow("3D Space Shooter");
    glutSetWindowData(&game);
    game.init();
    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);
    glutReshapeFunc(reshape);
    glutTimerFunc(16, update, 0);
    // Register mouse callbacks
    glutPassiveMotionFunc(mouseMove);  // Track mouse movement without clicks
    glutMouseFunc(mouseClick);         // Handle mouse clicks
    // Set cursor to crosshair for better aiming
    glutSetCursor(GLUT_CURSOR_CROSSHAIR);
    glutMainLoop();
    return 0;
}
```

SCREENSHOTS







References

[1] FreeGLUT software (https://freeglut.sourceforge.net/)
[2] GLEW Extension (https://sourceforge.net/projects/glew.mirror/)
[3] OpenGL-SpaceShooter (https://github.com/Nightey3s/OpenGL-SpaceShooter)
[4] Space-Shooter-Game (https://github.com/sheraadams/Space-Shooter-Game)
[5] SpaceShooter-OpenGL (https://github.com/wiktoriamarczyk/SpaceShooter-OpenGL)
[6] Game Loop Concepts (Basic) (https://dev.to/zigzagoon1/programming-patterns-for-games-game-loop-4goc)