Computer Graphics (UCS505)

Fighter Jet (F-22 Raptor)

B.E. 3rd Year – CSE

Submitted By -
Gurnoor Singh (102216022)
Pranav Duggal (102216023)

Submitted To -
Archana Kumari
(Assistant Professor)

THAPAR INSTITUTE
OF ENGINEERING & TECHNOLOGY
(Deemed to be University)

Computer Science and Engineering Department

Thapar Institute of Engineering and Technology

Patiala – 147001

INTRODUCTION

Project Overview

This project is a 3D fighter jet simulation developed using OpenGL and C++. It visually demonstrates a jet's takeoff, mid-air control, target engagement, and landing sequence in a dynamic virtual environment. The simulation incorporates various real-world visual elements such as an airstrip, textured terrain, enemy targets, and missile firing mechanics. Key features include realistic jet movement controlled via keyboard input, smooth camera transitions, animated takeoff and landing sequences, a missile firing system with collision detection, smoke trails, and mountainous terrain for enhanced realism. The objective of this project is to blend computer graphics techniques with interactive gameplay mechanics to create an immersive flight simulation experience, while also deepening understanding of transformation matrices, 3D object rendering, and animation control in OpenGL.

Scope of the Project

The scope of this project is to simulate a realistic 3D fighter jet environment using OpenGL and C++, focusing on core aspects of flight mechanics, interactive controls, and graphical realism. The simulation includes a complete takeoff-to-landing sequence, missile targeting and firing system, collision detection, camera tracking, terrain elements such as mountains and grass, and atmospheric effects like smoke trails. This project provides a platform to explore key computer graphics concepts such as 3D transformations, texture mapping, lighting, animation, and real-time rendering. While the current implementation is limited to a single-player simulation with predefined animations and interactions. The project is particularly useful for educational purposes, game development experimentation, and demonstrating the capabilities of OpenGL in rendering complex 3D scenes with interactive elements.

USER-DEFINED FUNCTIONS

S.No	Function Name	Description
1	`drawGroundAndStrip()`	Draws the green ground, black airstrip, and white stripes (main and secondary if needed).
2	`drawSmokeTrail()`	Renders a trail of smoke particles using small gray tori.
3	`draw3DMountains()`	Generates and draws multiple triangular 3D mountains with random heights and earthy colors.
4	`updateCamera()`	Updates the camera position to follow the jet during flight.
5	`drawModel()`	Renders a 3D textured model using vertex, normal, and texture coordinate data.
6	`drawJet()`	Displays the fighter jet model with appropriate transformations and scaling.
7	`drawEnemy()`	Spawns and displays the enemy jet model if it's alive and in flight state.
8	`display()`	Main rendering function that draws all scene components and handles missile logic.
9	`update()`	Updates simulation state, jet movement, missile and smoke logic, and triggers re-draw.
10	`keyboardDown()`	Handles key press events for controlling the jet, takeoff, landing, and firing missiles.
11	`keyboardUp()`	Handles key release events to stop continuous movement of the jet.
12	`reshape()`	Adjusts the OpenGL viewport and projection matrix on window resize.

CODE SNIPPETS

Header Files

```
#include <GL/glew.h>
#include <GL/glut.h>
#include <vector>
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <cmath>

#include <AL/al.h>
#include <AL/alc.h>

#define TINYOBJLOADER_IMPLEMENTATION
#include "tiny_obj_loader.h"

#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
```

Global Variables

```
const int WIDTH = 1280, HEIGHT = 720;
const float STRIP_LENGTH = 40.0f;
const float STRIP_WIDTH = 5.0f;

struct SmokeParticle {
   float x, y, z;
   float lifetime;
};

std::vector<SmokeParticle> smokeParticles;


enum State { IDLE, TAKEOFF_START, TAKEOFF_ASCEND, FLIGHT, TARGET_HIT,
LANDING_PREP, LANDING, DONE };
```

```
State state = IDLE;

float jetX = -STRIP_LENGTH / 2.0f + 1.0f;
float jetY = 0.0f;
float jetZ = 0.0f;
float jetAngle = 0.0f;
float speed = 0.05f;
float jetPitch = 0.0f; // NEW: Pitch angle for X-axis rotation


float camX = jetX, camY = 2.0f, camZ = 15.0f;
bool followJet = false;

bool missileFired = false;
float missileX, missileY;
const float missileSpeed = 0.5f;

float enemyX = 15.0f, enemyY = 10.0f;
bool enemyAlive = false;
bool wasdEnabled = false;

bool keyW, keyA, keyS, keyD, keyQ, keyE;
std::vector<float> vertices, normals, texcoords;
std::vector<unsigned int> indices;
```

Audio Setup

```
void initOpenAL() {
   device = alcOpenDevice(nullptr);
   if (!device) {
       std::cerr << "Failed to open audio device." << std::endl;
       return;
   }

   context = alcCreateContext(device, nullptr);
   alcMakeContextCurrent(context);

   alGenBuffers(1, &buffer);
   alGenSources(1, &source);
```

```
    std::vector<char> data;
    ALenum format;
    ALsizei freq;

    if (!loadWavFile("explosion.wav", data, format, freq)) {
        std::cerr << "Failed to load explosion.wav" << std::endl;
        return;
    }

    alBufferData(buffer, format, data.data(), data.size(), freq);
    alSourcei(source, AL_BUFFER, buffer);
}

void playExplosionSound() {
    alSourcePlay(source);
}
void playDelayedExplosionSound(int value) {
    playExplosionSound();
}

void cleanupOpenAL() {
    alDeleteSources(1, &source);
    alDeleteBuffers(1, &buffer);
    alcMakeContextCurrent(nullptr);
    alcDestroyContext(context);
    alcCloseDevice(device);
}
```

Load the Model

```
bool loadModel(const std::string &path) {
    tinyobj::attrib_t attrib;
    std::vector<tinyobj::shape_t> shapes;
    std::vector<tinyobj::material_t> materials;
    std::string warn, err;
```

```
    if (!tinyobj::LoadObj(&attrib, &shapes, &materials, &warn, &err,
path.c_str())) {
        std::cerr << err << std::endl; return false;
    }
    for (auto &shape: shapes) {
        for (auto &idx: shape.mesh.indices) {
            vertices.push_back(attrib.vertices[3*idx.vertex_index]);
            vertices.push_back(attrib.vertices[3*idx.vertex_index+1]);
            vertices.push_back(attrib.vertices[3*idx.vertex_index+2]);
            if (!attrib.normals.empty()) {
                normals.push_back(attrib.normals[3*idx.normal_index]);
                normals.push_back(attrib.normals[3*idx.normal_index+1]);
                normals.push_back(attrib.normals[3*idx.normal_index+2]);
            }
            if (!attrib.texcoords.empty()) {

texcoords.push_back(attrib.texcoords[2*idx.texcoord_index]);

texcoords.push_back(attrib.texcoords[2*idx.texcoord_index+1]);
            }
            indices.push_back(indices.size());
        }
    }
    return true;
}

GLuint loadTexture(const char *file) {
    int w,h,n;
    unsigned char *data = stbi_load(file,&w,&h,&n,0);
    if(!data){ std::cerr<<"Failed: "<<file<<std::endl; return 0; }
    GLuint tex; glGenTextures(1,&tex); glBindTexture(GL_TEXTURE_2D,tex);
    GLenum fmt=(n==4?GL_RGBA:GL_RGB);
    glTexImage2D(GL_TEXTURE_2D,0,fmt,w,h,0,fmt,GL_UNSIGNED_BYTE,data);
    glGenerateMipmap(GL_TEXTURE_2D);

glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR_MIPMAP_LINEA
R);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
    stbi_image_free(data);
    return tex;
}
```

Airstip & Mountains

```
void drawGroundAndStrip(float yOffset = 0) {
    // Green ground
    glColor3f(0.1f, 0.8f, 0.1f);  // Bright green
    glBegin(GL_QUADS);
    glVertex3f(-100, yOffset, -100);
    glVertex3f(100, yOffset, -100);
    glVertex3f(100, yOffset, 100);
    glVertex3f(-100, yOffset, 100);
    glEnd();

    // Black airstrip
    glColor3f(0, 0, 0);
    glBegin(GL_QUADS);
    glVertex3f(-STRIP_LENGTH / 2, yOffset + 0.01f, -STRIP_WIDTH / 2);
    glVertex3f(STRIP_LENGTH / 2, yOffset + 0.01f, -STRIP_WIDTH / 2);
    glVertex3f(STRIP_LENGTH / 2, yOffset + 0.01f, STRIP_WIDTH / 2);
    glVertex3f(-STRIP_LENGTH / 2, yOffset + 0.01f, STRIP_WIDTH / 2);
    glEnd();

    // White stripes
    glColor3f(1, 1, 1);
    float stripeW = STRIP_LENGTH / 20;
    for (int i = 0; i < 20; i += 2) {
        float x0 = -STRIP_LENGTH / 2 + i * stripeW;
        glBegin(GL_QUADS);
        glVertex3f(x0, yOffset + 0.02f, -0.2f);
        glVertex3f(x0 + stripeW, yOffset + 0.02f, -0.2f);
        glVertex3f(x0 + stripeW, yOffset + 0.02f, 0.2f);
        glVertex3f(x0, yOffset + 0.02f, 0.2f);
        glEnd();
    }
}

void drawSmokeTrail() {
    glColor3f(0.7f, 0.7f, 0.7f);  // gray smoke
    for (auto& s : smokeParticles) {
        glPushMatrix();
        glTranslatef(s.x, s.y, s.z);
```

```
        glutSolidTorus(0.05, 0.15, 8, 12);  // small torus
        glPopMatrix();
    }
}

void draw3DMountains() {
    srand(42);  // Fixed seed for consistent randomness

    float baseSize = 10.0f;
    int rows = 2;
    int cols = 15;

    float startX = -90.0f;
    float startZ = -120.0f;  // Further back from airstrip

    for (int row = 0; row < rows; row++) {
        for (int i = 0; i < cols; i++) {
            float baseX = startX + i * (baseSize + 2.0f);
            float baseZ = startZ + row * (baseSize + 5.0f);

            float height = 10.0f + rand() % 15; // 10 to 25

            // Base corners
            float x1 = baseX;
            float z1 = baseZ;
            float x2 = baseX + baseSize;
            float z2 = baseZ;
            float x3 = baseX + baseSize;
            float z3 = baseZ + baseSize;
            float x4 = baseX;
            float z4 = baseZ + baseSize;

            // Peak point
            float peakX = baseX + baseSize / 2.0f;
            float peakZ = baseZ + baseSize / 2.0f;
            float peakY = height;

            // Base color: dark green to brown
            float r = 0.2f + (float)(rand() % 100) / 500.0f;
            float g = 0.3f + (float)(rand() % 100) / 400.0f;
            float b = 0.1f;
```

```
            // Draw four triangular faces
            glBegin(GL_TRIANGLES);

            // Front face
            glColor3f(r, g, b);
            glVertex3f(x1, 0.0f, z1);
            glVertex3f(x2, 0.0f, z2);
            glColor3f(1.0f, 1.0f, 1.0f);
            glVertex3f(peakX, peakY, peakZ);

            // Right face
            glColor3f(r, g, b);
            glVertex3f(x2, 0.0f, z2);
            glVertex3f(x3, 0.0f, z3);
            glColor3f(1.0f, 1.0f, 1.0f);
            glVertex3f(peakX, peakY, peakZ);

            // Back face
            glColor3f(r, g, b);
            glVertex3f(x3, 0.0f, z3);
            glVertex3f(x4, 0.0f, z4);
            glColor3f(1.0f, 1.0f, 1.0f);
            glVertex3f(peakX, peakY, peakZ);

            // Left face
            glColor3f(r, g, b);
            glVertex3f(x4, 0.0f, z4);
            glVertex3f(x1, 0.0f, z1);
            glColor3f(1.0f, 1.0f, 1.0f);
            glVertex3f(peakX, peakY, peakZ);

            glEnd();
        }
    }
}
```

Camera Setup & Drawing Jet

```
void updateCamera() {
   if (followJet) {
       camX = jetX;
       camY = jetY + 2.0f;
       camZ = jetZ + 15.0f;
   }
}
void drawModel() {
   glEnable(GL_TEXTURE_2D);
   glBindTexture(GL_TEXTURE_2D, textureColor);
   glBegin(GL_TRIANGLES);
   for(size_t i=0;i<indices.size();++i){
     if(!texcoords.empty()) glTexCoord2f(texcoords[2*i],texcoords[2*i+1]);
     if(!normals.empty())
glNormal3f(normals[3*i],normals[3*i+1],normals[3*i+2]);
     glVertex3f(vertices[3*i],vertices[3*i+1],vertices[3*i+2]);
   }
   glEnd();
   glDisable(GL_TEXTURE_2D);
}


void drawJet() {
   glPushMatrix();
   glTranslatef(jetX, jetY+1, jetZ);
   glRotatef(-90, 1, 0, 0);
   glRotatef(-jetAngle,0,1,0);
   glRotatef(jetPitch, 1.0f, 0.0f, 0.0f);
   glScalef(0.5,0.5,0.5);
   glColor3f(0.7f, 0.7f, 1.0f);
   drawModel();
   glPopMatrix();
}

void drawEnemy() {
   if (!enemyAlive && state == FLIGHT) {
       enemyAlive = true;
       enemyX = jetX + 15;
       enemyY = jetY ;
```

```
    }
    if (enemyAlive) {
        glPushMatrix();
        glTranslatef(enemyX, enemyY, 0);
        glColor3f(1, 0, 0);
        glRotatef(-90, 1, 0, 0);
        glScalef(0.5,0.5,0.5);
        drawModel();
        glPopMatrix();
    }
}
```

Keyboard Interactions

```
void update(int) {
    switch (state) {
        case TAKEOFF_START:
            jetX += speed;
            speed += 0.0008f;
            if (jetX >= STRIP_LENGTH / 2 - 5) {
                state = TAKEOFF_ASCEND;
            }
            break;

        case TAKEOFF_ASCEND:
            jetAngle += 1.5f;
            jetX += 0.1f;
            jetY += 0.25f;
            if (jetAngle >= 50) {
                jetAngle = 0;
                followJet = true;
                wasdEnabled = true;
                jetX = -10;
                state = FLIGHT;
            }
            break;

        case LANDING_PREP:
            if (jetPitch > 0.1f) {
            jetPitch -= 0.5f; // decrease pitch
```

```
            if (jetPitch < 0.0f) jetPitch = 0.0f; // clamp
             }
             else if (jetPitch < -0.1f) {
            jetPitch += 0.5f; // increase pitch
            if (jetPitch > 0.0f) jetPitch = 0.0f; // clamp
            }
            if (jetPitch == 0)
            {
                jetY -= 0.05f;
                jetX += 0.10f;
                if(jetAngle<=25) jetAngle += 0.5f;

                if (jetY <= 1.0f) {
                    if(jetAngle>=0.0){
                        jetAngle -= 2.5f;
                    } else
                    {
                        jetAngle = 0;
                        jetY = 0.1f;
                        state = DONE;
                    }

                }
            }
            break;

            case FLIGHT:
            jetX += 0.01f;
            break;

        default: break;
    }

    if (keyD || missileFired) {
        smokeParticles.push_back({ jetX, jetY + 1.0f, jetZ, 1.0f });
    }

    // Update existing particles
    for (auto it = smokeParticles.begin(); it != smokeParticles.end(); ) {
        it->lifetime -= 0.02f;
        it->y += 0.005f;  // slowly rise
```

```
        if (it->lifetime <= 0)
            it = smokeParticles.erase(it);
        else
            ++it;
    }

    if (wasdEnabled) {
        if (keyW) jetY += 0.1f;
        if (keyS) jetY -= 0.1f;
        if (keyA) jetX -= 0.1f;
        if (keyD) jetX += 0.1f;
        if (keyQ) jetPitch +=0.5f;
        if (keyE) jetPitch -=0.5f;
    }

    updateCamera();
    glutPostRedisplay();
    glutTimerFunc(16, update, 0);
}

void keyboardDown(unsigned char key, int, int) {
    switch (key) {
        case 't': if (state == IDLE) { state = TAKEOFF_START; speed =
0.05f; } break;
        case 'l': if (state == TARGET_HIT) { state = LANDING_PREP; } break;
        case 'w': keyW = true; break;
        case 'a': keyA = true; break;
        case 's': keyS = true; break;
        case 'd': keyD = true; break;
        case 'q': keyQ = true; break;
        case 'e': keyE = true; break;
        case ' ': if (state == FLIGHT && !missileFired) {
            missileFired = true;
            missileX = jetX;
            missileY = jetY + 1;
            glutTimerFunc(500, playDelayedExplosionSound, 0);

        } break;
    }
}
```

```
void keyboardUp(unsigned char key, int, int) {
    if (key == 'w') keyW = false;
    if (key == 'a') keyA = false;
    if (key == 's') keyS = false;
    if (key == 'd') keyD = false;
    if (key == 'q') keyQ = false;
    if (key == 'e') keyE = false;
}
```

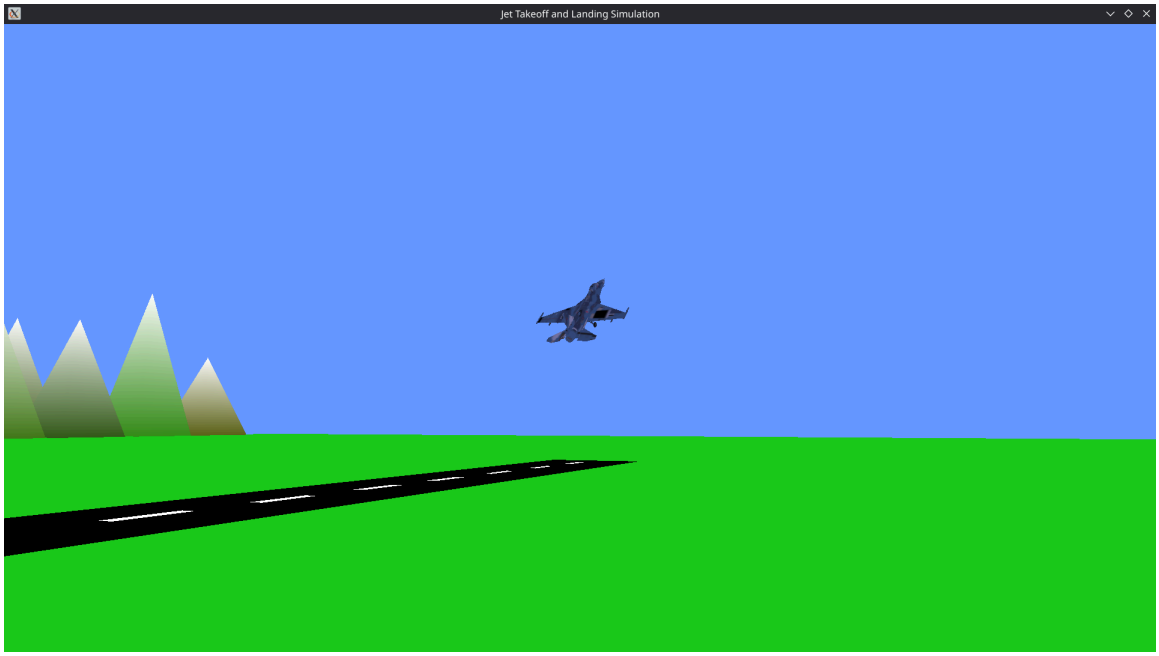
Main Function

```
int main(int argc, char** argv) {
    srand(time(0));
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
    glutInitWindowSize(WIDTH, HEIGHT);
    glutCreateWindow("Jet Takeoff and Landing Simulation");
    initOpenAL();
    glewInit();
    glEnable(GL_DEPTH_TEST);
    glClearColor(0.4f, 0.6f, 1.0f, 1.0f);
    loadModel("model/F-2.obj");
    textureColor = loadTexture("model/textures/F-2_Color.png");
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboardDown);
    glutKeyboardUpFunc(keyboardUp);
    glutTimerFunc(16, update, 0);
    glutMainLoop();
    return 0;
}
```

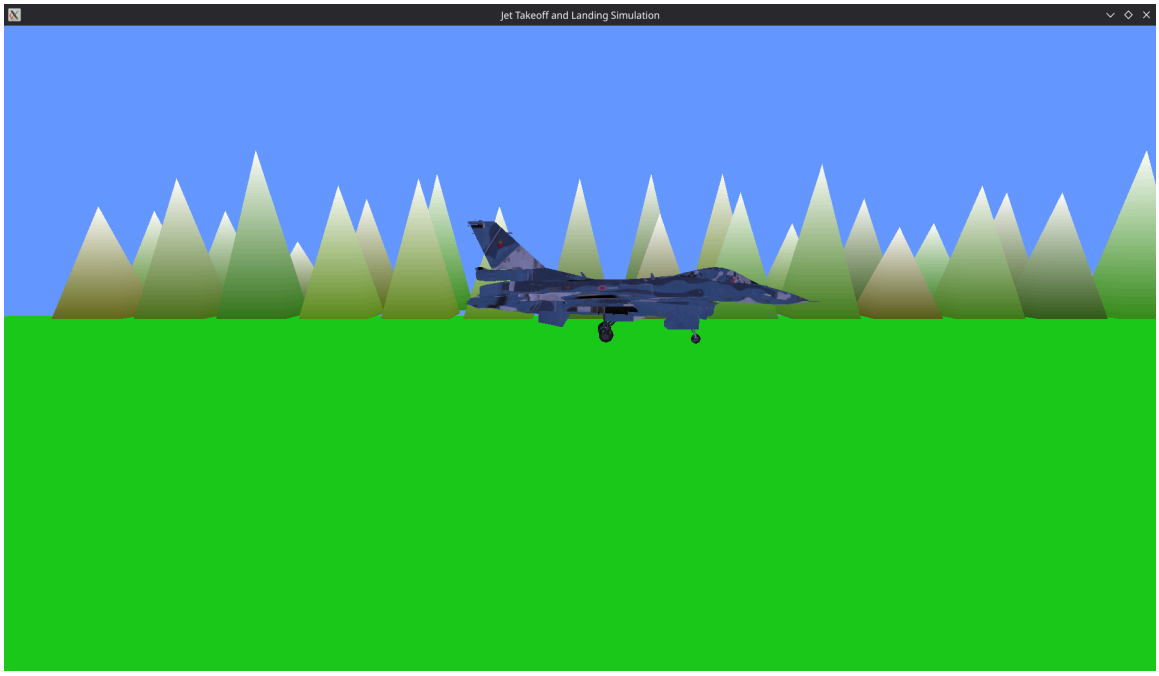
SCREENSHOTS

Initial Position

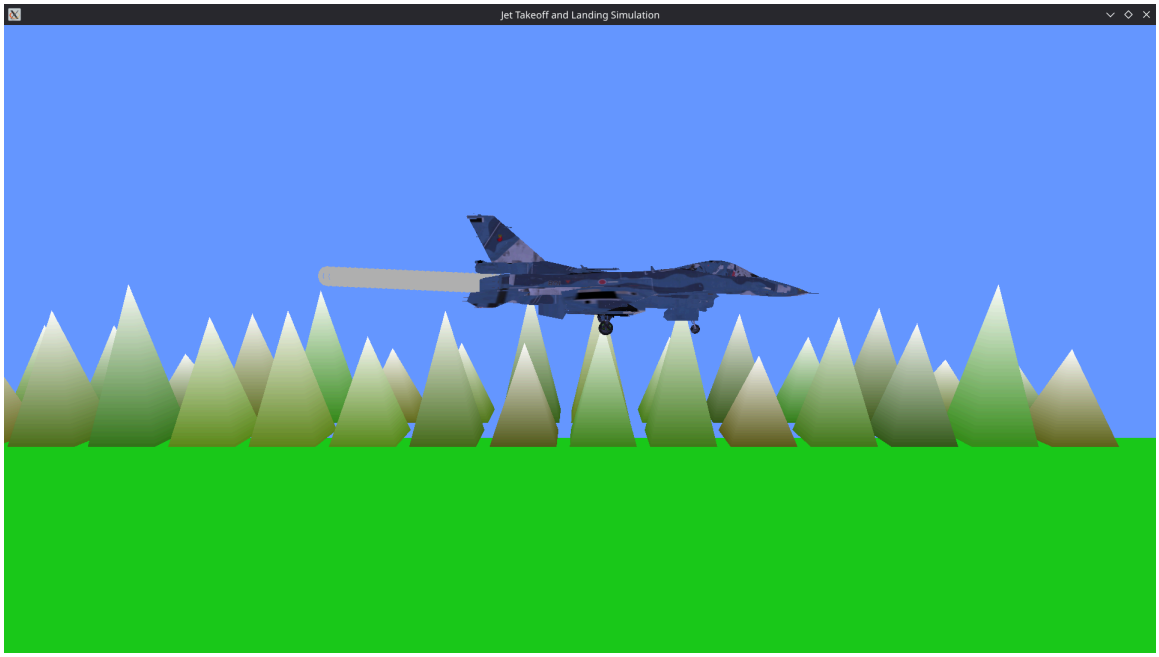
Take Off


Jet Takeoff and Landing Simulation

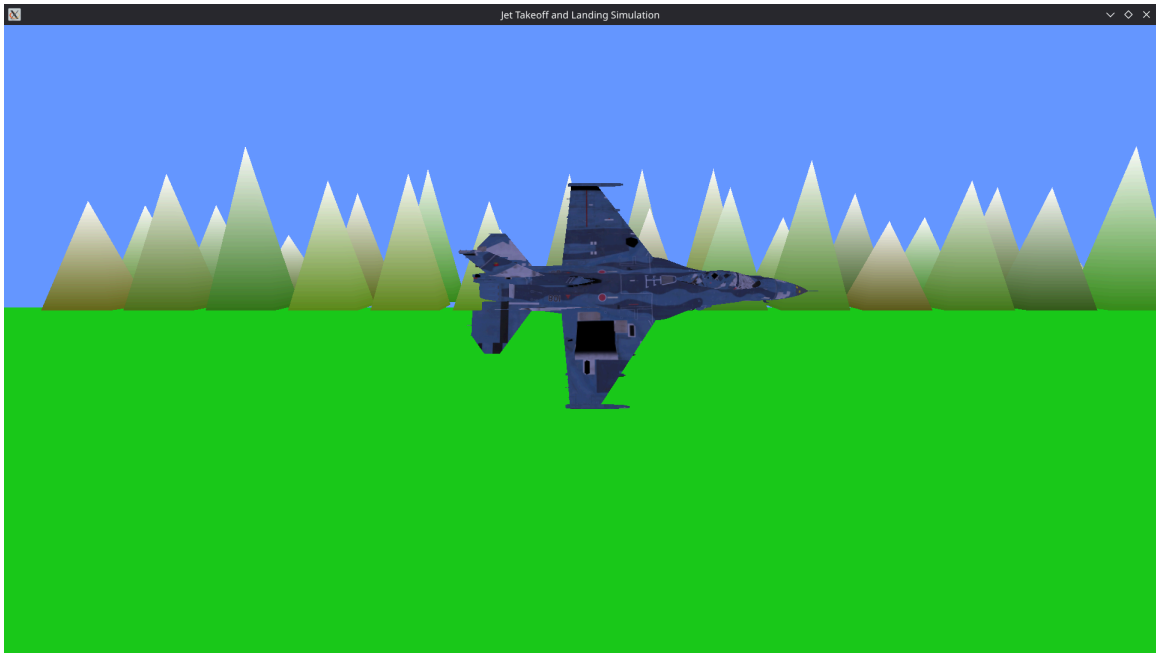
Flight

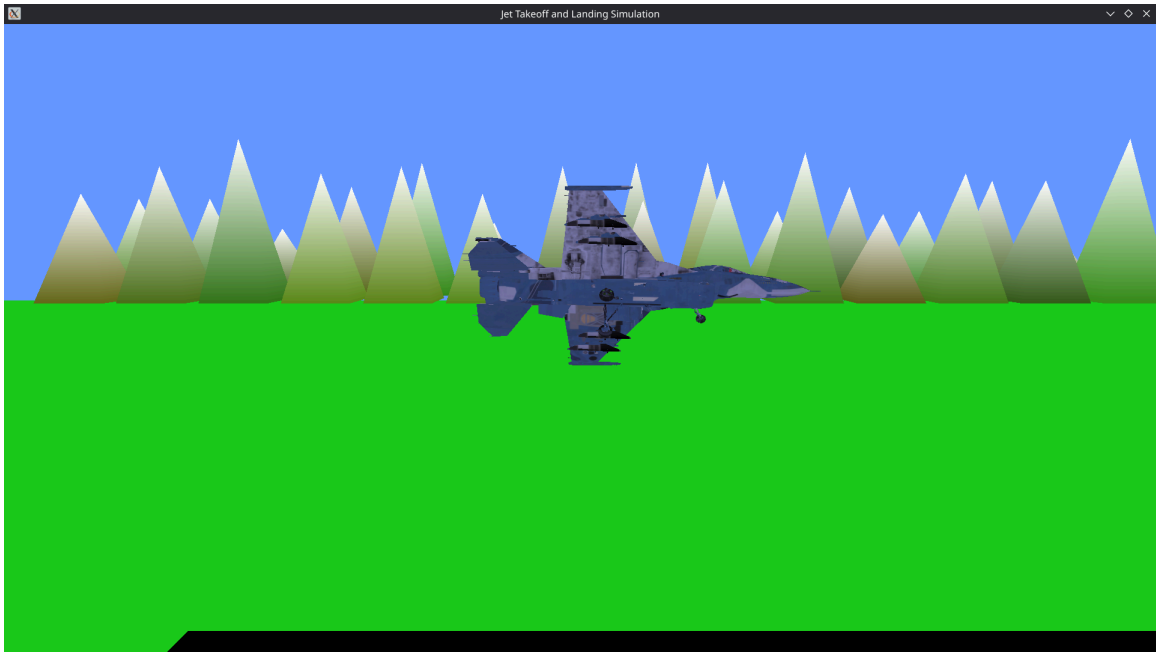

Jet Takeoff and Landing Simulation

Acceleration & Smoke Trail



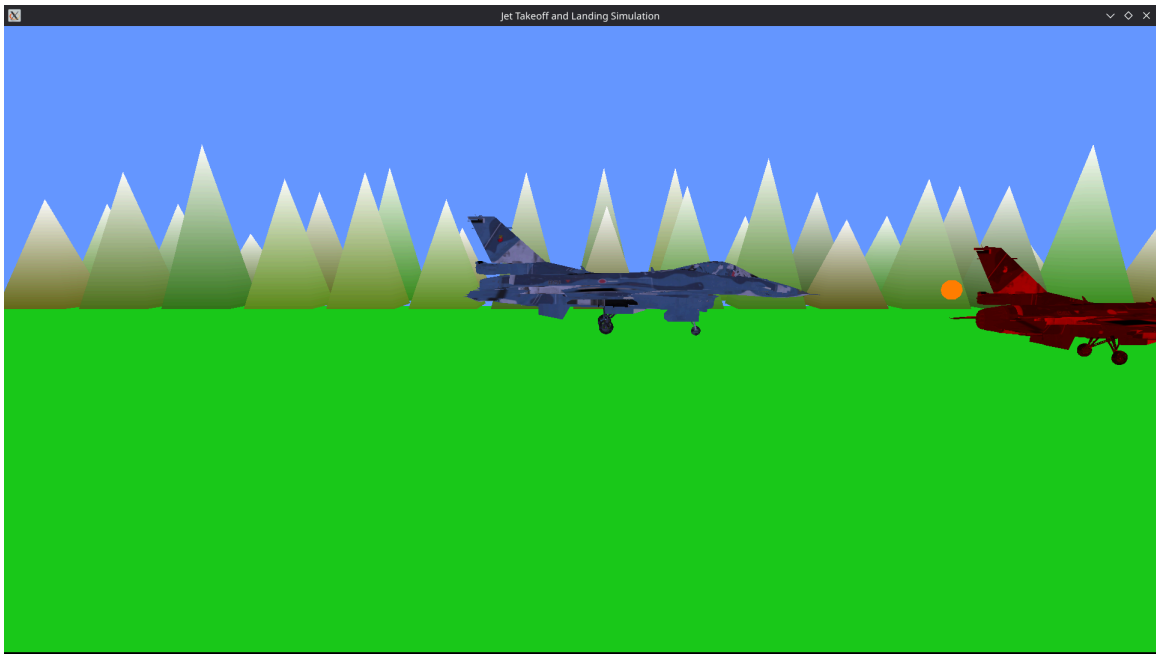
Pitch Control



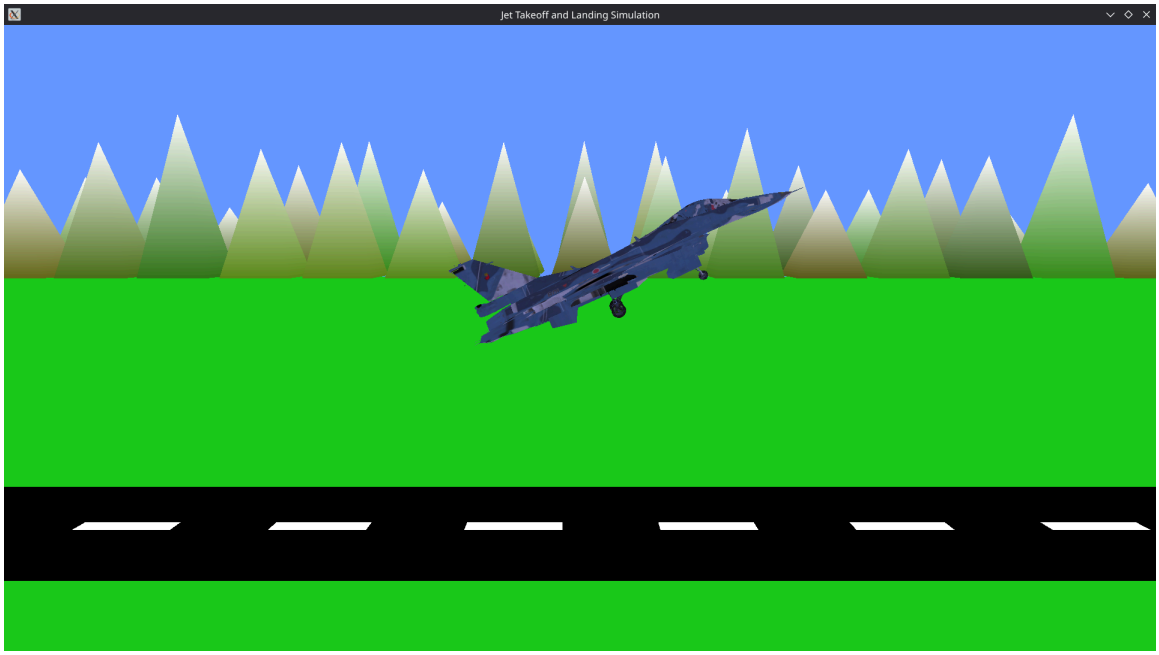


Enemy Detected

Missile Fired



Landing Preparation



Final State

References

1. OpenGL
Description: Cross-platform graphics API for rendering 2D and 3D vector graphics.
Link: https://www.khronos.org/opengl/

2. GLEW (OpenGL Extension Wrangler Library)
Description: Manages OpenGL extensions, allowing access to modern OpenGL features across different systems.
Link: https://glew.sourceforge.net/

3. FreeGLUT
Description: Open-source alternative to the original GLUT library, used for window creation, input handling, and context management.
Link: https://github.com/FreeGLUTProject/freeglut

4. OpenAL (Open Audio Library)
Description: Cross-platform API for rendering multichannel 3D positional audio, used in games and simulations.
Link: https://www.openal.org/

5. TinyOBJLoader
Description: Lightweight, single-header C++ library for loading `.obj` 3D models, useful for real-time applications.
Link: https://github.com/tinyobjloader/tinyobjloader

6. stb_image
Description: Header-only image loading library for `.png`, `.jpg`, `.bmp`, etc., used primarily for texture mapping in OpenGL.
Link: https://github.com/nothings/stb

7. Sketchfab – Fighter Jet 3D Models
Description: Online repository of downloadable 3D fighter jet models in `.obj` and other formats used in this simulation.
Link: https://sketchfab.com/tags/fighter-jet