Assignment #3-1 Report

팀명:	스타벅스	
팀원:	최우석	성창환
학과:	단일계열	컴퓨터공학과
학번:	49003157	20180480
Hemos ID:	f2soundd	chseung

Contents

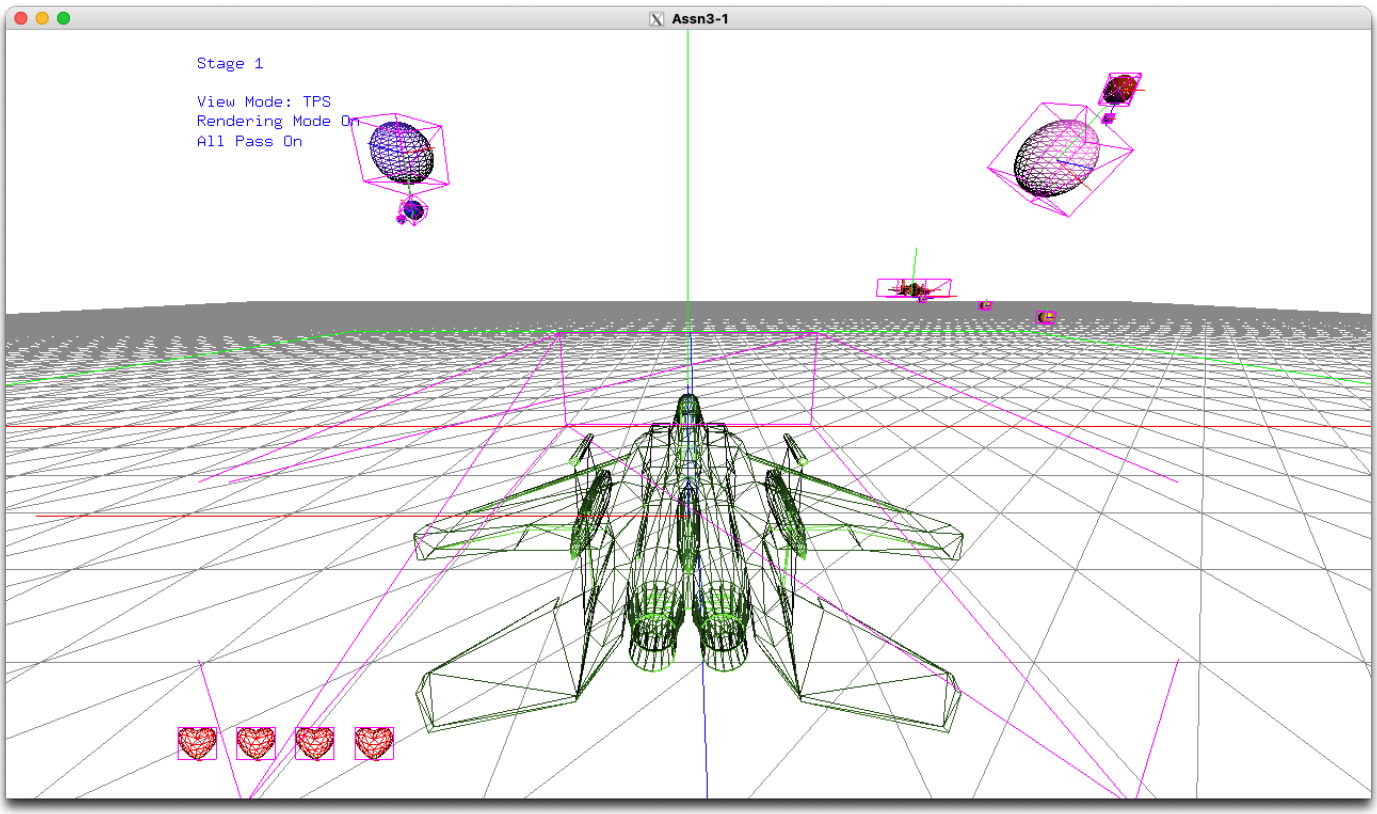
Feature Overview

System Design

- The game runs at 60 fps.
- The world is right-handed coordinate system.
- The game supports viewing in TPS, FPS, and 2D-like.
- The game supports rendering with wire frame.

Requirements

- Depending on the rendering mode, all 3D objects in the world can be drawn with wire frame. The bounding box and positive of model coordinate axises are also drawn with wire frame. Changing the rendering mode is ‘r’.
- The mesh of aircraft, bullet, planetary, item, and life in HUD are defined as an external model(.obj). These model files are in “assets/models/”, and the assimp library was used to load the model.
- The ground represented by the grid is represented by GL_TRIANGLE_STRIP, and the boundary of the game is represented by a green line.
- Depending on the viewing mode, the arguments of gluLookAt() is changed and the camera settings are changed. Changing the viewing mode is ‘v’.



Development Environment

Windows (x64)

- Visual Studio 16.9.1
- freeglut 3.0.0
- glew 2.1.0
- glm 0.9.9.7
- assimp 5.0.1

macOS

- Visual Studio Code 1.54.3
- freeglut 3.2.1
- glew 2.2.0
- glm 0.9.9.8
- assimp 5.0.1

Data Structures

Mesh Class

The "Mesh" class stores hierarchical models formed from external .obj model files. In addition, the bounding box formed by these vertices is automatically calculated and stored.

Object Class

The "Object" class has a "Mesh" object as a member variable. It supports various functions to draw this "Mesh" to the world. In addition, it provides various information of "Mesh" defined in the world. Below member variables and method of “Object” are for transforming “Mesh” to world coordinate.

```
private: // Factors of model-view matrix
    glm::vec3 translatef;
    float scalef;
    float inheritedScalef;
    std::vector<glm::vec3> rotateAxisStack;
    std::vector<float> angleStack;
    glm::mat4 modelViewMat;

public:
    virtual Object::void draw () {
        glPushMatrix();
        glTranslatef(translatef.x, translatef.y, translatef.z);
        glScalef(scalef, scalef, scalef);
        for (int i = rotateAxisStack.size() - 1 ; i >= 0 ; i --)
            glRotatef(angleStack[i], rotateAxisStack[i].x, rotateAxisStack[i].y,
                      rotateAxisStack[i].z);
        if (drawFlag)
            mesh.draw();
        for (Object* child : children)
            child->draw();
        glPopMatrix();
    }
```

In particular, there are methods that returns the model-view matrix, the vector of the direction the model looks at in the world (since the model frame is a left-handed coordinate, it means a positive z-axis vector in the model coordinates defined in the world), the upward vector of the model, and so on. These methods allow us to calculate the camera based on the player's position, and perform collision detection. All of this is obtained from the model-view matrix, and the model-view matrix is computed in every update method.

```
    glm::mat4 Object::getModelViewMat () const {
        return modelViewMat;
    }
    glm::vec3 Object::getUpVec () const {
        return modelViewMat[1]; // second column
    }
    glm::vec3 Object::getFrontVec () const {
        return modelViewMat[2]; // third column
    }
```

Implementation

Implementation - Drawing Wire Frame

Wireframe rendering mode changes Mesh::wireframe to true. The Mesh object storing the information of the .obj model set the parameter of glPolygonMode() to GL_LINE if Mesh::wireframe is true, or to GL_FILL, and draw the mesh recursively.

```
void Mesh::draw () const {
    if (scene == nullptr)
        return;
    glEnable(GL_LIGHTING);
    if (wireframe) {
        glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
        drawBoundingBox();
    }
    else
        glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
    glColor4f(color.r, color.g, color.b, color.a);
    drawMeshes(scene->mRootNode);
}

void Mesh::drawMeshes (const aiNode* node) const {
    for (int i = 0 ; i < node->mNumMeshes; i ++) {
        const aiMesh* mesh = scene->mMeshes[node->mMeshes[i]];
        for (int j = 0 ; j < mesh->mNumFaces ; j ++) {
            const aiFace* face = &mesh->mFaces[j];
            switch(face->mNumIndices) {
                case 1:
                    glBegin(GL_POINTS);
                    break;
                case 2:
                    glBegin(GL_LINES);
                    break;
                case 3:
                    glBegin(GL_TRIANGLES);
                    break;
                default:
                    glBegin(GL_POLYGON);
                    break;
            }
            for(int k = 0 ; k < face->mNumIndices ; k ++) {
                int idx = face->mIndices[k];
                if(mesh->mNormals != NULL)
                    glNormal3fv(&mesh->mNormals[idx].x);
                glVertex3fv(&mesh->mVertices[idx].x);
            }
            glEnd();
        }
    }
    for (int i = 0 ; i < node->mNumChildren ; i ++)
        drawMeshes(node->mChildren[i]);
}
```

Implementation - Loading and Constructing Models

The aircraft of the player and the enemy each adopted a different airplane model, the bullet and Planetary adopted the sphere model, and the item adopted the bullet box model. All models are imported via Mesh::loadModel. The Open Asset Import Library(assimp) library was used to load and store the model.

```
bool Mesh::loadModel (std::string path) {
    scene = aiImportFile(path.c_str(), aiProcessPreset_TargetRealtime_MaxQuality);
    if (scene) {
        calculateBoundingBox();
        width = abs(bbMin.x - bbMax.x);
        height = abs(bbMin.y - bbMax.y);
        depth = abs(bbMin.z - bbMax.z);
        longestSide = max(width, height);
        longestSide = max(longestSide, depth);

        bbVertices.clear();
        bbVertices.push_back(glm::vec3(bbMin.x, bbMin.y, bbMax.z));
        bbVertices.push_back(glm::vec3(bbMax.x, bbMin.y, bbMax.z));
        bbVertices.push_back(glm::vec3(bbMax.x, bbMax.y, bbMax.z));
        bbVertices.push_back(glm::vec3(bbMin.x, bbMax.y, bbMax.z));
        bbVertices.push_back(glm::vec3(bbMin.x, bbMin.y, bbMin.z));
        bbVertices.push_back(glm::vec3(bbMax.x, bbMin.y, bbMin.z));
        bbVertices.push_back(glm::vec3(bbMax.x, bbMax.y, bbMin.z));
        bbVertices.push_back(glm::vec3(bbMin.x, bbMax.y, bbMin.z));

        return true;
    }
    return false;
}
```

Implementation - Grid Ground

Ground is also expressed as an object, and takes charge of the root in the perspective projected scene graph. “World” object draw the ground.

```
void World::draw() {
    glDisable(GL_LIGHTING);
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    float z = -AXIS_LIMIT_ABS;
    bool flag = true;
    int vertexCount = 0;
    glColor3f(0.5f, 0.5f, 0.5f);
    while (z <= AXIS_LIMIT_ABS) {
        float x = -AXIS_LIMIT_ABS;
        glBegin(GL_TRIANGLE_STRIP);
        while (x <= AXIS_LIMIT_ABS) {
            glVertex3f(x, 0.0f, z);
            vertexCount ++;
            if (flag) // 밑으로
                z += TILE_LEN;
            else { // 위로
                x += TILE_LEN;
                z -= TILE_LEN;
            }
            if (x > AXIS_LIMIT_ABS)
                z += TILE_LEN;
            flag = !flag;
        }
        glEnd();
    }

    // x axis (red)
    glColor3f(1.0f, 0.0f, 0.0f);
    glBegin(GL_LINE_LOOP);
    glVertex3f(-WORLD_LIMIT_ABS, 0.0f, 0.0f);
    glVertex3f(WORLD_LIMIT_ABS, 0.0f, 0.0f);
    glEnd();

    // z axis (blue)
    glColor3f(0.0f, 0.0f, 1.0f);
    glBegin(GL_LINE_LOOP);
    glVertex3f(0.0f, 0.0f, -WORLD_LIMIT_ABS);
    glVertex3f(0.0f, 0.0f, WORLD_LIMIT_ABS);
    glEnd();

    // bound of game world (green)
    const int WORLD_TILE_N = TILE_N / 3;
    const int offset = WORLD_TILE_N / 2;
    glColor3f(0.0f, 1.0f, 0.0f);
    glBegin(GL_LINE_LOOP);
    glVertex3f(-TILE_LEN * float(offset), 0.0f, -TILE_LEN * float(offset));
    glVertex3f(-TILE_LEN * float(offset), 0.0f, TILE_LEN * float(offset));
    glVertex3f(TILE_LEN * float(offset), 0.0f, TILE_LEN * float(offset));
    glVertex3f(TILE_LEN * float(offset), 0.0f, -TILE_LEN * float(offset));
    glEnd();

    Object::draw();
}
```

Implementation - Camera Configuration

The “GamePlay” class is in charge of controlling the camera. The “GamePlay" class has arguments for gluLookAt() as its member variables, and when the user presses the'v' key or the player moves, these parameters are appropriately changed and then gluLookAt() is called when rendering.

```
void GamePlay::setViewTPS () {
    glm::vec3 playerPos = player->getWorldPos();
    glm::vec3 playerFrontVec = player->getFrontVec();
    glm::vec3 playerUpVec = player->getUpVec();
    camPos = glm::vec3(playerPos + (-playerFrontVec * 7.0f + playerUpVec * 3.5f));
    at = playerPos + playerFrontVec * glm::vec3(AXIS_LIMIT_ABS);
    camUp = glm::vec3(playerUpVec);
    player->setDraw(true);
}

void GamePlay::setViewFPS () {
    glm::vec3 playerPos = player->getWorldPos();
    camPos = playerPos;
    at = playerPos + player->getFrontVec() * glm::vec3(AXIS_LIMIT_ABS);
    camUp = player->getUpVec();
    player->setDraw(false);
}


void GamePlay::setView2D () {
    camPos = glm::vec3(0.0f, WORLD_LIMIT_ABS * 2.0f, 0.0f);
    at = glm::vec3(0.0f, 0.0f, 0.0f);
    camUp = glm::vec3(0.0f, 0.0f, -1.0f);
    player->setDraw(true);
}
```

...and below methods are called in glut’s display callback function.

```
void GamePlay::renderPerspectiveScene () {
    gluLookAt(camPos.x, camPos.y, camPos.z,
              at.x, at.y, at.z,
              camUp.x, camUp.y, camUp.z);
    perspectiveSceneRoot->draw();
}
```

Additional Implementation

1. Implemented a HUD that outputs the remaining life of the player and various modes of the game as text. HUD is mapped to NDC by glOrtho2D(). From display() which a callback function defined in main.cpp, performs NDC mapping of the game world with gluPerspective() and NDC mapping of the HUD with glOrtho2D().

```
void display () {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    GLfloat base = WINDOW_HEIGHT < WINDOW_WIDTH ? WINDOW_HEIGHT : WINDOW_WIDTH;
    GLfloat widthset = WINDOW_WIDTH / base;
    GLfloat heightset = WINDOW_HEIGHT / base;

    glDepthMask(GL_TRUE);
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(75, WINDOW_WIDTH / WINDOW_HEIGHT, 0.1f, 1000.0f);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gameplay.renderPerspectiveScene();

    glClear(GL_DEPTH_BUFFER_BIT);
    glDepthMask(GL_FALSE);
    glDisable(GL_DEPTH_TEST);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-1.0f * widthset, 1.0f * widthset, -1.0f * heightset, 1.0f * heightset,
0.0f, UI_CAM_Z - UI_Z);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gameplay.renderOrthoScene();

    glViewport(0, 0, WINDOW_WIDTH, WINDOW_HEIGHT);
    glutSwapBuffers();
}


void GamePlay::renderOrthoScene () {
    gluLookAt(UI_CAM_X, UI_CAM_Y, UI_CAM_Z,
              UI_CAM_X, UI_CAM_Y, UI_Z,
              0.0f, 1.0f, 0.0f);
    hud->draw();
}
```

2. The third viewing mode was implemented to show the same view as the assignment 2 by placing the camera at the top of the game world and looking down. By pressing ‘v’ key, the viewing mode changes to TPS->FPS->2D.

How to Build and Run

Windows (x64)

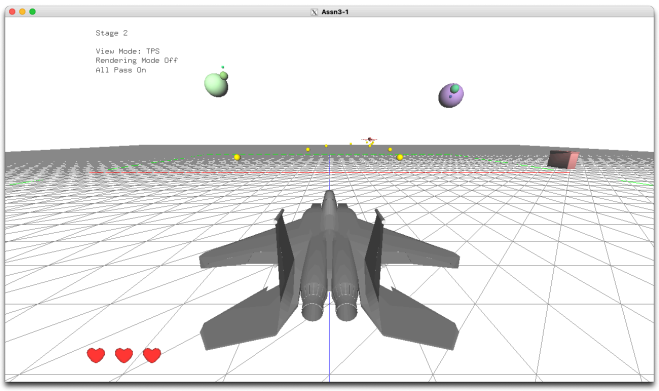
Requirements

- Visual Studio 2019 for C++ Desktop

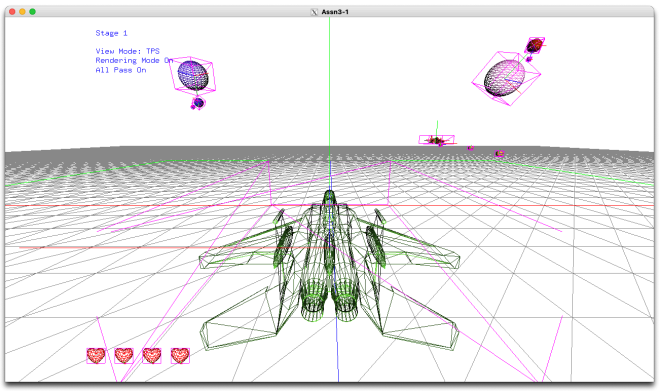
Build and Run

1. Double click file `shmup.vcxproj` to open the project in Visual Studio 2019.
2. Click `Build --> Build Solution` , or press `F7`
3. Click `Debug --> Run without Debug` , or press `Ctrl + F5`

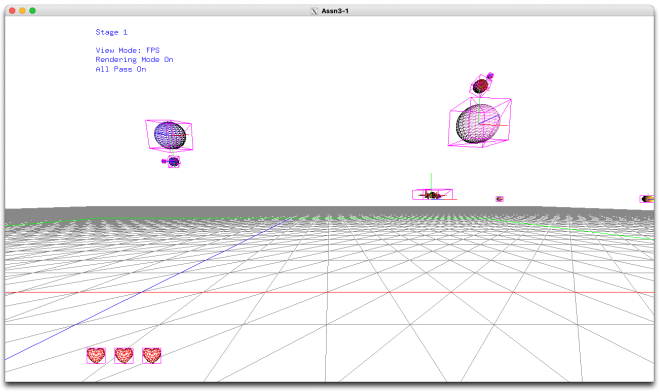
Result Screen



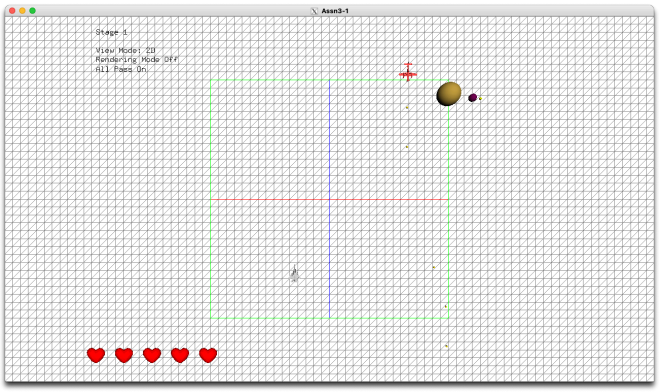
TPS, Wireframe Off



TPS, Wireframe On



FPS, Wireframe On



2D, Wireframe On

Conclusion

Conclusion

1. Through 3D transformation, I clearly understand the fixed pipeline of OpenGL, which was not clearly understood even in Assignment 2.

2. By using the model-view matrix to obtain the object's direction vector, we can better understand the transformation pipeline of OpenGL.

3. The part of dealing with scaling within the scene graph was tricky. For example, if the model-view matrix of the parent object reduces the size by half, the application point is transmitted to the child object as it is, so scaling of the parent object must be considered in order to define the size of the child object through scaling. To solve this problem, in the Object::setLongestSideTo() method that defines the size of the object, scaling can be performed independently by using the scale factor applied to the parent object. I was curious if there is a design pattern to solve this problem.

Improvements

1. When nearly 100 bullets were drawn on the screen, the program slowed down considerably. It would have been better if we had considered drawing bullets in 2D.

2. In the current implementation, adding player orientation with the mouse seems to be a more fun game.

Resources

The Open Asset Import Library (assimp)

https://github.com/assimp/assimp

Player Aircraft Model (“assets/models/player.obj”)

https://www.cgtrader.com/free-3d-models/aircraft/jet/f-15-c-eagle

Enemy Aircraft Model (“assets/models/ebm314.obj”)

https://www.cgtrader.com/free-3d-models/aircraft/military/emb-314-super-tucano

Ammo Crate Model (“assets/models/ammo_crate.obj”)

https://www.cgtrader.com/free-3d-models/military/other/ammo-crate-pbr-lowpoly

Heart Model (“assets/models/love.obj”)

https://www.cgtrader.com/free-3d-models/character/anatomy/love-low-poly

Sphere Model (“assets/models/sphere.obj”)

https://www.cgtrader.com/free-3d-models/space/other/sphere-de6c0ee5-444a-4b26-afd3-37b4880601a0