Assignment #3-2 Report

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

Contents

Feature Overview

System Design

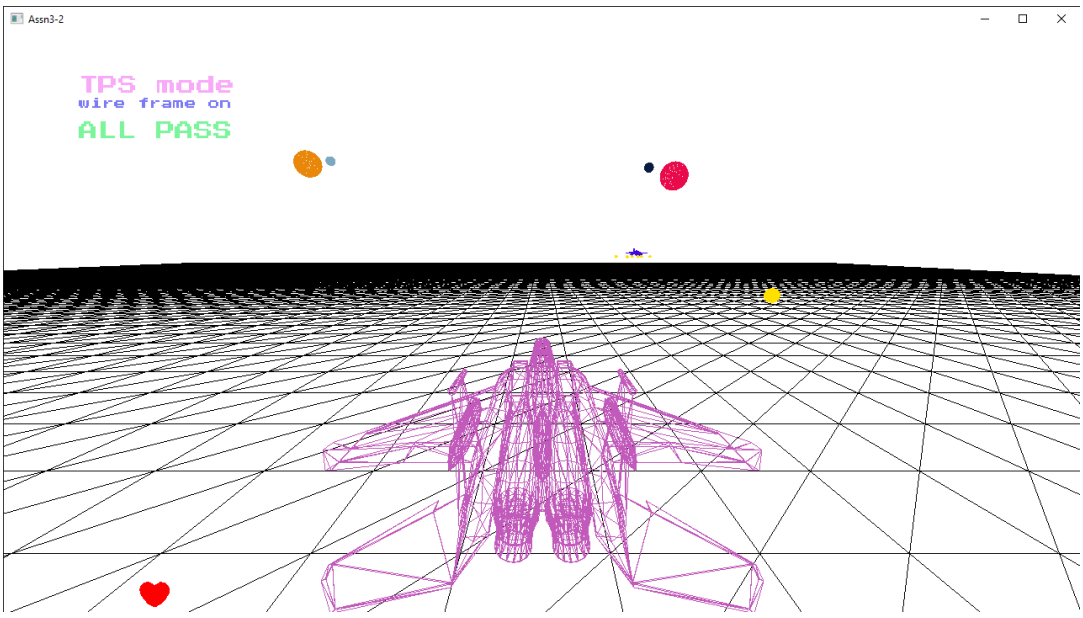
- The game runs at 60 fps.
- The world is right-handed coordinate system.
- The game supports viewing in TPS, FPS, and 2D-like.
- The game supports rendering with wire frame.

Requirements

- Depending on the rendering mode, all 3D objects in the world can be drawn with wire frame. Changing the rendering mode is ‘r’.
- The mesh of aircraft, bullet, planetary, item, hearts, and texts 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 mesh of ground represented by the grid is composed of directly defined mesh.
- The point v_{model} of the model frame is reached to the canonical view coordinate v_{cvc} by the following equation

$$v_{cvc} = M_{projection} \cdot M_{view} \cdot M_{modelView} \cdot v_{model}$$

$MVP = M_{projection} \cdot M_{view} \cdot M_{modelView}$ is calculated on the CPU, and $v_{cvc} = MVP \cdot v_{model}$ is calculated oil the vertex shader.



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

Data Structures

Shader Class

“Shader” class represents a shader program in which a vertex shader and a fragment shader are linked. When the constructor of this class is called, it compiles the vertex and fragment shaders, and attaches and links these two shaders so that it stores the ID of the shader program.

Mesh Class

Based on the vertices array and indices array, VBO, EBO, and VAO are created and bound in “Mesh” class. After that, the information stored in the VAO is drawn by calling the draw() method. Since the primitives type supported by “Mesh" is GL_TRIANGLES, vertices and indices must be configured accordingly GL_TRIANGLES.

ModelViewMat Class

“ModelViewMat” class is designed for a matrix for the model to world space transformation of the “Object" class which will be described below. Matrix calculation was implemented using GLM library, and the transformation order was implemented to be Scale -> Rotate -> Translate.

Object Class

"Object" class represents a model, and this model is geometrically defined as an array of “Mesh”. This class also holds a "Shader" object, so it uses this shader when drawing meshes array. This “Object” class is also designed to be able to act as a node in the scene graph.

Implementation

Implementation - Shader

The vertex shader and fragment shader used in this program are as follows.

shader/vertex.vert

```
#version 400
uniform mat4 mvp;
layout (location = 0) in vec3 aPos;
void main()
{
    gl_Position = mvp * vec4(aPos, 1.0f);
}
```

shader/fragment.frag

```
#version 400
uniform vec4 color;
out vec4 fragColor;
void main()
{
    fragColor = color;
}
```

The "Shader" class is implemented as follows. Each VBO can be drawn through glDrawElements() after calling Shader::bind().

```
class Shader {
public:
    Shader(const std::string& vertexShaderPath, const std::string& fragmentShaderPath) {
        unsigned int vertexShader = readAndCompileShader(vertexShaderPath, GL_VERTEX_SHADER);
        unsigned int fragmentShader = readAndCompileShader(fragmentShaderPath, GL_FRAGMENT_SHADER);
        ID = linkShaders(vertexShader, fragmentShader);
        glDeleteShader(vertexShader);
        glDeleteShader(fragmentShader);
    }
    void bind() {
        glUseProgram(ID);
    }
    void unbind() {
        glUseProgram(0);
    }

private:
    // ... utility methods.

public:
    unsigned int ID;
};
```

Implementation - Drawing Pipeline in Game Loop

Each object is displayed on the screen through the following pipeline.

1. *Update the player's model-view matrix by user input.*
2. *Change the view matrix for the camera by referring to the player's model-view matrix.*
3. *Construct a VP matrix by multiplying the projection matrix and the view matrix.*
4. *The MVP matrix is formed by multiplying the VP with the parent's model-view matrix and the model's model-view matrix.*
5. *Send MVP matrix to shader.*
6. *Calculate $v_{cvc} = MVP \cdot v_{model}$ in the shader.*

Now let's explain by looking at the code.

Below Aircraft::move() method is called when the user moves the player's aircraft by manipulating the arrow keys. The model-view matrix is changed by calling setTranslate() at the end.

```
void Aircraft::move(const glm::vec3 directionInModelFrame) {
    glm::vec4 unit = getModelViewMat() * glm::vec4(directionInModelFrame, 0);
    glm::vec3 newTranslate = getWorldPos() + glm::vec3(unit / glm::length(glm::vec3(unit)) * getSpeed());
    // ... code to prevent leaving the world.
    setTranslate(newTranslate);
}
```

After that, GamePlay calculates the view matrix (perspectiveLookAt) in below methods. The method to be called differs according to the view mode that changes according to the user's v key input.

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

void GamePlay::setViewFPS() {
    glm::vec3 playerPos = player->getWorldPos();
    const glm::vec3 camPos = playerPos;
    const glm::vec3 at = playerPos + player->getFrontVec() * glm::vec3(AXIS_LIMIT_ABS);
    const glm::vec3 camUp = player->getUpVec();
    perspectiveLookAt = glm::lookAt(camPos, at, camUp);
}
```

After that, the GamePlay sends the VP matrix (perspectiveProjection*perspectiveLookAt) as an argument when calling the display method of the root object of the scene graph. Set the drawing options for the wire frame at this time through a call to glPolygonMode().

```
void GamePlay::renderPerspectiveScene() {
    glDepthMask(GL_TRUE);
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);
    if (renderingMode)
        glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    else
        glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
    perspectiveSceneRoot->display(perspectiveProjection * perspectiveLookAt, glm::mat4(1.0f));
}
```

Now, the MVP matrix is constructed in the display() method of each “Object" object, which is a node of the scene graph, and sent to the shader. MVP is defined as $VP \cdot CTM$, where CTM is defined as the model-view matrix of the parent node in the scene graph and the model-view matrix of the current node. Below code is the Object::display() method. In shaders, MVP is declared as a uniform.

```
virtual void display(const glm::mat4& viewProjectionMat, const glm::mat4& parentModelViewMat) {
    const glm::mat4 ctm = parentModelViewMat * this->modelViewMat.get();
    if (shader && drawFlag) {
        shader->bind();
        unsigned int uni = glGetUniformLocation(shader->ID, "mvp");
        glUniformMatrix4fv(uni, 1, GL_FALSE, glm::value_ptr(viewProjectionMat * ctm));
        uni = glGetUniformLocation(shader->ID, "color");
        glUniform4fv(uni, 1, glm::value_ptr(color));
        for (Mesh mesh : meshes)
            mesh.draw();
        shader->unbind();
    }
    for (Object* child : children)
        child->display(viewProjectionMat, ctm);
}
```

Now the shader is executed to draw the VAO of the mesh by calling glDrawElements inside the Mesh::draw() method.

Implementation - Loading Models and Mapping to Vertex Array

.obj files were read using the Open Asset Import Library(assimp). In the "Object" class, the loadModel() method that performs this and the assimpToMesh() method that changes the assimp data structure to the "Mesh" class are defined. When reading the file, with the option of aiProcess_Triangulate, all primitives are changed to triangles and saved. The vertices and indices defined in “aiNode” of assimp are stored in std::vector<glm::vec3> and std::vector<unsigned int>, and new Mesh is created based on these two arrays. After that, the mesh is stored in the Object::meshes array. In other words, "aiNode" of the .obj model is mapped to “Mesh". Below two codes are definitions of Object::loadModel() and Object::assimpToMesh().

```
void loadModel(const std::string& path) {
    Assimp::Importer import;
    const aiScene* scene = import.ReadFile(path, aiProcess_Triangulate | aiProcess_FlipUVs);
    if (!scene || scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE || !scene->mRootNode) {
        cout << "ERROR::ASSIMP::" << import.GetErrorString() << endl;
        return;
    }
    assimpToMesh(scene->mRootNode, scene);
    calcBoundingBox(scene);
}


void assimpToMesh(aiNode* node, const aiScene* scene) {
    const unsigned int* meshIdx = node->mMeshes;
    for (int i = 0 ; i < node->mNumMeshes ; i ++) {
        const aiMesh* mesh = scene->mMeshes[meshIdx[i]];
        std::vector<glm::vec3> vertices;
        std::vector<unsigned int> indices;
        for (int j = 0 ; j < mesh->mNumVertices ; j ++) { // Vertices
            const glm::vec3 pos =
                            glm::vec3(mesh->mVertices[j].x, mesh->mVertices[j].y, mesh->mVertices[j].z);
            vertices.push_back(pos);
        }
        for (int j = 0 ; j < mesh->mNumFaces ; j ++) {
            if (mesh->mFaces->mNumIndices == 3) {
                indices.push_back(mesh->mFaces[j].mIndices[0]);
                indices.push_back(mesh->mFaces[j].mIndices[1]);
                indices.push_back(mesh->mFaces[j].mIndices[2]);
            }
            else {
                std::cout << "WARNING::ASSIMP::NON_TRIANGLE_FACE: " <<
                                                                    mesh->mFaces->mNumIndices << std::endl;
            }
        }
        meshes.push_back(Mesh(vertices, indices));
    }
    for (int i = 0 ; i < node->mNumChildren ; i ++)
        assimpToMesh(node->mChildren[i], scene);
}
```

The Object class has an std::vector<Mesh> for the Mesh object as a member variable. Each Mesh composes the VAO through the vertex array and index array passed as a parameter of the constructor and stores it in the GPU. Afterwards, to draw this mesh, glDrawElements() is called with GL_TRIANGLES (because the primitives of the model were changed to triangles through assmp) in Mesh::draw() method. The following code is the prototype of the Mesh class.

```
class Mesh {
public:
    Mesh(const std::vector<glm::vec3> vertices, const std::vector<unsigned int> indices) {
        this->vertices = vertices;
        this->indices = indices;

        glGenVertexArrays(1, &VAO);
        glBindVertexArray(VAO);

        glGenBuffers(1, &VBO);
        glBindBuffer(GL_ARRAY_BUFFER, VBO);
        glBufferData(GL_ARRAY_BUFFER, vertices.size() * sizeof(glm::vec3), &vertices[0], GL_STATIC_DRAW);
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(glm::vec3), (void*)0);
        glEnableVertexAttribArray(0);

        glGenBuffers(1, &EBO);
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER,
                     indices.size() * sizeof(unsigned int),
                     &indices[0], GL_STATIC_DRAW);
    }
    void draw() {
        glBindVertexArray(VAO);
        glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
        glBindVertexArray(0);
    }

private:
    std::vector<glm::vec3> vertices;
    std::vector<unsigned int> indices;

private:
    unsigned int VAO, VBO, EBO;
};
```

Implementation - Grid Ground

The “Object” class not only loads an external .obj file to form a mesh(Object::loadModel()), but also supports a method of forming a mesh through arbitrary vertex arrays and index arrays(Object::pushMesh()). The “World” class, which represents ground as a triangular grid, inherits the “Object” class, and directly forms the vertex array and index array representing the ground, and loads the vertex array and index array to the GPU by calling Object::pushMesh().

```
World::World(const glm::vec4 color) {
    setColor(color);
    std::vector<glm::vec3> vertices;
    std::vector<unsigned int> indices;
    std::vector<unsigned int> lastLineIdx;
    float z = -AXIS_LIMIT_ABS;
    bool flag = true;
    unsigned int vertexCount = 0;
    while (z <= AXIS_LIMIT_ABS) {
        float x = -AXIS_LIMIT_ABS;
        while (x <= AXIS_LIMIT_ABS) {
            glm::vec3 v;
            v.x = x;
            v.y = 0.0f;
            v.z = z;
            vertices.push_back(v);
            vertexCount++;
            if (flag)
                z += TILE_LEN;
            else {
                x += TILE_LEN;
                z -= TILE_LEN;
            }
            if (x > AXIS_LIMIT_ABS)
                z += TILE_LEN;
            flag = !flag;
        }
        lastLineIdx.push_back(vertexCount);
    }
    unsigned int begin = 2;
    for (int i = 0; i < lastLineIdx.size(); i++) {
        unsigned int end = lastLineIdx[i];
        for (unsigned int third = begin; third < end; third++) {
            unsigned int first = third == begin ? begin - 2 : indices[indices.size() - 2];
            unsigned int second = third == begin ? begin - 1 : indices[indices.size() - 1];
            indices.push_back(first);
            indices.push_back(second);
            indices.push_back(third);
        }
        begin = end + 2;
    }
    pushMesh(vertices, indices);
    setDraw(true);
}
```

Additional Implementation

1. Implemented a HUD that outputs the remaining life of the player and various modes of the game as text. These are defined in the "Hud" class. “GamePlay” class has a perspective scene graph and an orthogonal scene graph, and provides the following two methods to draw each of them. These two methods are called from glut's callback function, display() (defined in main).

```
void GamePlay::renderPerspectiveScene() {
    glDepthMask(GL_TRUE);
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);
    if (renderingMode)
        glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    else
        glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
    perspectiveSceneRoot->display(perspectiveProjection * perspectiveLookAt, glm::mat4(1.0f));
}

void GamePlay::renderOrthoScene() {
    glClear(GL_DEPTH_BUFFER_BIT);
    glDepthMask(GL_FALSE);
    glDisable(GL_DEPTH_TEST);
    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
    hud->display(orthoProjection * orthoLookAt);
}
```

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)

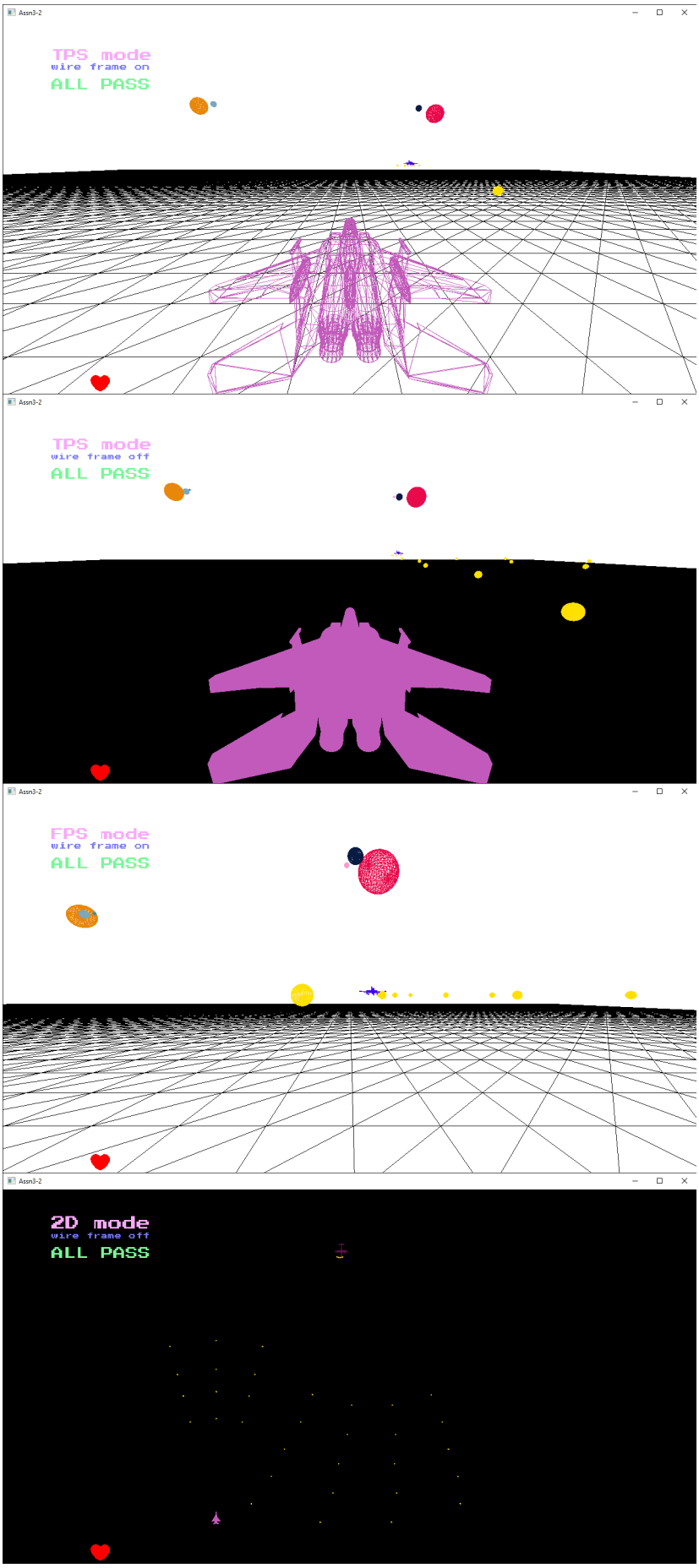
Requirements

- 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



Conclusion

Review

1. We wondered what kind of calculations would be efficient to do in the shader. In other words, we wondered if it would be better to perform the multiplication of all transform matrices in the shader, or if it would be better to send only one transform matrices to the shader after calculating it in the application. Since projection matrix and view matrix were applied in common to all vertices, we chose the latter method in which the application computes both the VP calculation that multiplies these two and the MVP calculation that multiplies each model-view matrix. The following articles have helped us a lot:
   https://stackoverflow.com/questions/16620013/should-i-calculate-matrices-on-the-gpu-or-on-the-cpu

2. In the past assignments, we wanted to efficiently represent a number of objects as singletons such as bullets, since these are differ only in the model-view matrix. This desire was solved a lot with the use of shaders. Originally, "StraightMovingObjectManager" class directly loads several "Object" objects into memory, but now it loads one "Object" into memory and change only "Object::modelViewMat" when drawing several objects.

Resources

3D Models

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

Text Models (“assets/models/text3d_*”)
http://profilki.pl/en/generators/3d-texts

Codes

The Open Asset Import Library (assimp)
https://github.com/assimp/assimp

OpenGL pipeline
https://heinleinsgame.tistory.com/7?category=757483

Shader class
https://heinleinsgame.tistory.com/8?category=757483

Object class
https://heinleinsgame.tistory.com/23?category=757483

Others

https://stackoverflow.com/questions/16620013/should-i-calculate-matrices-on-the-gpu-or-on-the-cpu