Assignment #2 Report

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

Contents

Feature Overview

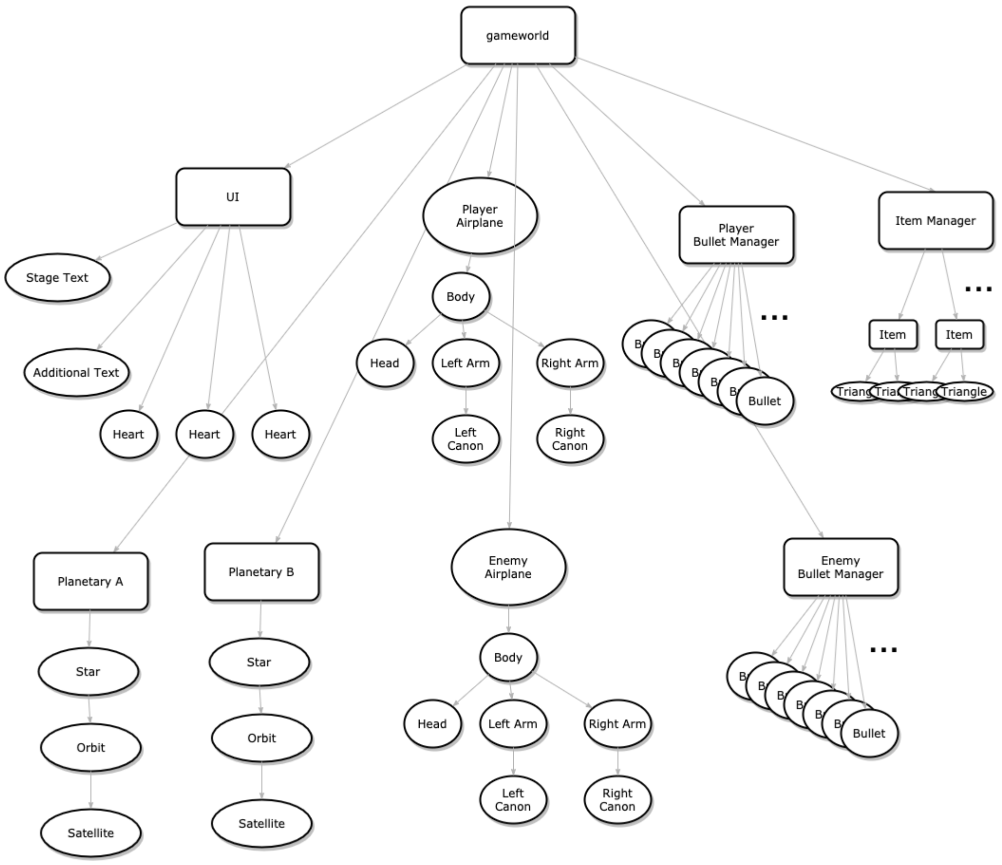
System Design

- The game runs at 60 fps.
- "GamePlay" class updates all objects, handles user i/o, and control the game play.
- The main function has only Glut functions and "GamePlay" object. Glut functions call "update" and "render" method of "GamePlay" object.

Requirements

- All visible objects have or have not vertices but can be visualized when they are nodes of scene graph.
- All visible objects are defined themselves as a hierarchical structure. For example, body, head, arms, and canons are inherited "Object" class and all of them are configured in a graph form, and the child objects follow the transformation of the parent object as it is.
- "GamePlay" has all objects in its scene graph.

Scene Graph



Development Environment

Windows (x64)

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

macOS

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

Data Structures

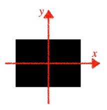
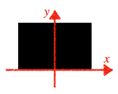
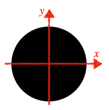
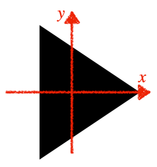
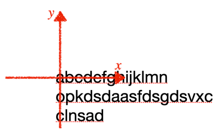
Basic

Basic data structures.

Point2D	Represent a point (x, y).
Rgba	Represent a color. The range of each value is 0.0 to 1.0
TransformationMatrix	Represent a matrix for transformation.

Figures

Data structures that represent figures through vertices. All classes below are inherited "Figure" abstract class.

Rect	Represent a rectangle. Width and height can be defined. The center of the axis is a center of the rectangle.	
BaseRect	Represent a rectangle. Width and height can be defined. The center of the axis is a middle of base.	
Circle	Represent a circle. Radius can be defined. The center of the axis is a center of the circle.	
Triangle	Represent a triangle. Each angles are same(0.5π). Distance from the vertex to the center can be defined. The center of the axis is a center of the triangle. The initial shape is one vertex in the positive of the x-axis.	
Text	Represent a text. String can be defined. The center of the axis is the leftmost side of the top row of string.	

Implementation

Implementation - Scene Graph

Scene Node

include/base/Object.hpp
src/base/Object.cpp

The node of scene graph is “Object” class. Every node can have its own vertices if needed. The code below is for the graph API supported by the “Object” class.

```
public: // Graph API
    Figure* setFigure (const int figureType);
    Figure* operator * ();
    Object* pushChild (Object* child);
    Object* pushChild (Object* child, const int priority);
    Object* pushChild (const int figureType);
    Object* pushChild (const int figureType, const int priority);
    void popBackChild ();
    void popFrontChild ();
    std::list<Object*>& getChildren ();
private:
    Figure* figure;
    Object* parent;
    std::list<Object*> children;
```

The “figure” member variable is its vertices, and pointer operator returns the “figure”. The “setFigure()” method defines a figure. The types of figure is represented in above [Data Structures] section, and this “figure” member variable can have these types based on C++ polymorphism.

The "pushChild(Object* child)” method adds an object already allocated in memory as a child, and other overrided methods that takes “figureType” as an argument allocates a new child to memory. Destructor frees its all children.

Priority means the relative depth of an object. When “priority=FRONT”, the child has a lower depth than the parent, and when “priority=BACK”, the child has a deeper depth than the parent. Default is the same.

The code of methods of “Object" class below is for modifying color and transformation of “figure”.

```
public: // Transformations
    TransformMatrix getModelViewMatrix () const;
    void setMatrix (const TransformMatrix& mat);
    void clearMatrix ();
    void setTranslate (const Point2D p);
    void setTranslate (const GLfloat x, const GLfloat y);
    void setTranslate (const GLfloat x, const GLfloat y, const GLfloat z);
    void translate (const Point2D p);
    void translate (const GLfloat x, const GLfloat y);
    void translate (const GLfloat x, const GLfloat y, const GLfloat z);
    void setRotate (const GLfloat degree);
    void rotate (const GLfloat degree);
    void setScale (const GLfloat x, const GLfloat y);
    void scale (const GLfloat x, const GLfloat y);
```

```
public: // Colors
    void setColor (const GLfloat _R, const GLfloat _G, const GLfloat _B);
    void setColor (const GLfloat _R, const GLfloat _G, const GLfloat _B,
                   const GLfloat _A);
    void setColor (const Rgba _color);
    Rgba getColor () const;
    void setRandomColor ();
    void setColorAlpha (const GLfloat _A);

private:
    Rgba color;
    TransformMatrix modelViewMat;
```

Three transformations are supported; translate, rotate, and scale. The method with "set" word in front of each transformation changes "modelViewMat" to the given argument, and the method without set is to accumulate and apply the value to "modelViewMat". For example, "setTranslate(1.0, 1.0)" makes "modelViewMat.tx = 1.0" and "modelViewMat.ty = 1.0" but "translate(1.0, 1.0)" makes "modelViewMat.tx += 1.0" and "modelViewMat.ty += 1.0".

All objects in the scene graph update their state by calling their own "update" method and draw their figure by calling the "display" method, in pre-order traversal. Since these methods can be customized in objects that inherit this class, they are declared virtual.

```
public: // Called in Pre-order Traversal
    virtual void update ();
    virtual void display () const;
```

The inside of the methods work as follows:

```
void Object::update () {
    for (Object* child : children)
        child->update();
}

void Object::display () const {
    if (parent == nullptr) {
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
    }
    else
        glPushMatrix();
    glTranslatef(modelViewMat.tx, modelViewMat.ty, modelViewMat.tz);
    glRotatef(modelViewMat.degree, 0.0f, 0.0f, 1.0f);
    glScalef(modelViewMat.sx, modelViewMat.sy, 1.0f);
    if (figure != nullptr) {
        glColor4f(color.R, color.G, color.B, color.A);
        figure->draw();
    }
    for (Object* child : children)
        child->display();
    glPopMatrix();
}
```

Depending on the pre-order traversal, the node first does its job and then calls the child's job. With this implementation, if we define the "MyObject" class that derived the "Object" class, define what we want in each of "MyObject::update()", and at the end, we just need to call "Object::update()".

The child node receives the parent node's “modelViewMat” first, which is defined in the “display()” method. Firstly, if a node is root of the scene graph, then it initializes the OpenGL GL_MODELVIEW buffer. Otherwise, it calls “glPushMatrix()” to save the present model-view matrix. Secondly, it apply its model-view matrix through calling “glTranslatef()”, “glRotatef()”, and “glScalef()”. Thirdly, it draw its own vertices by calling “figure->draw()”. Lastly, after executing the child's display methods in sequence, call "glPopMatrix()" to return the model-view matrix before applying its transformation.

Construct Scene Graph

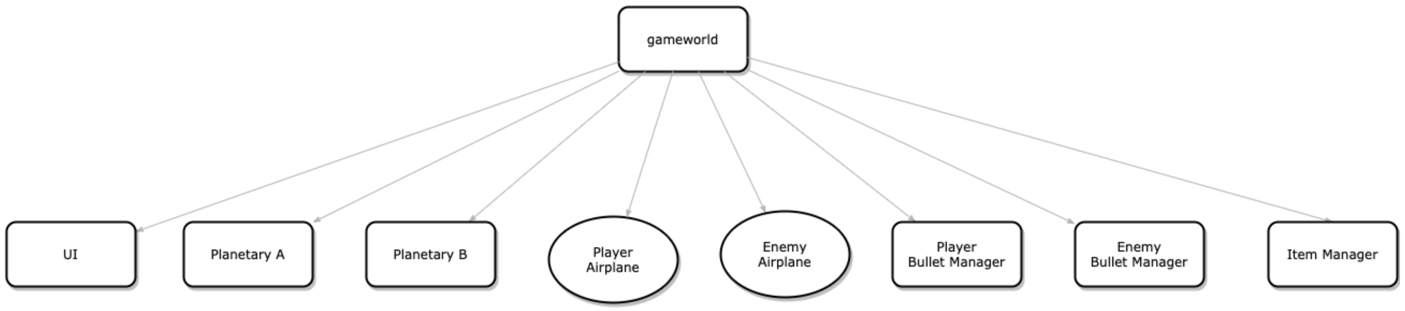
src/gameplay/GamePlay.cpp

Now see how the “GamePlay” class construct the scene graph for all objects.

```
GamePlay::GamePlay () {
    gameworld = new Object;
    ui = new Ui(INIT_PLAYER_LIVES);
    player = new Airplane;
    enemy = new Airplane;
    playerBulletManager = new ThirdObjectManager(BULLET);
    enemyBulletManager = new ThirdObjectManager(BULLET);
    itemManager = new ThirdObjectManager(ITEM);
    planetaryA = new Planetary;
    planetaryB = new Planetary;

    gameworld->pushChild(ui, FRONT);
    gameworld->pushChild(planetaryA, BACK);
    gameworld->pushChild(planetaryB, BACK);
    gameworld->pushChild(player);
    gameworld->pushChild(enemy);
    gameworld->pushChild(playerBulletManager);
    gameworld->pushChild(enemyBulletManager);
    gameworld->pushChild(itemManager);

    //...

}
```

The root node of the graph is “gameworld”. In every frame, “GamePlay” call “gameworld->update()” and “gameworld->display()” for traversing the scene graph in order to update and draw all objects.

Below image represent these graph. A circular node is an object with a figure, and a rectangular node is an object without a figure.



Implementation - ThirdObject (Bullet, Item)

ThirdObjectManager

include/entity/ThirdObjectManager.hpp

src/entity/ThirdObjectManager.cpp

“Bullet” and “Item” classes have some same features. First, they are created and removed when certain conditions are met. Second, when they once created, they move linearly in a certain direction. So I defined “ThirdObjectManager” class which deals with these specific objects. “ThirdObjectManager” also object which inherits the “Object" class, but does not have its own figure. Rather, it can have a bunch of “Bullet” or “Item” objects as its children. Below two methods of “ThirdObjectManager” can show what it does in our game world.

```
public:
    void activateObject (const TransformMatrix& mat, const GLfloat param,
                         const Rgba color, const GLfloat speed);
    bool deactivateObjectWhichIsIn (const Point2D leftTop, const Point2D rightBottom);
```

When bullet or item have to be activated, “activateObject()” method is called. Then activates one of the objects it has as a child, and initializes the model-view matrix of this object with "mat" received as an argument. Then, when “ThirdObjectManager::update()” is called, all activated children move based on their own model-view matrix.

“deactivateObjectWhichIsIn()” method does the opposite. It deactivates the child objects that exist in the rectangle formed by leftTop and rightBottom in the world coordinate. “GamePlay” call this method for detecting ‘airplane is hit or not’ or ‘player get item or not’.

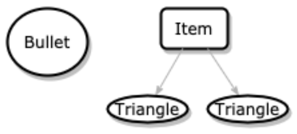
"ThirdObjectManager" defines default deactivation conditions for "Bullet" and "Item" in the "update()" method, respectively; "Bullet" is when the position goes out of the screen, and "Item" is when a certain time has passed. If the object doesn't need to be removed, translate it in the positive y-axis direction of the model-view which the object has.

```
void ThirdObjectManager::update () {
    std::stack<Object*> deactivating;
    for (Object* object : activeObjects) {
        switch (objectType) {
        case BULLET: {
            Bullet* bullet = (Bullet*)object;
            if (bullet->isOutOfBound()) {
                deactivating.push(bullet);
                continue;
            }
            bullet->move(90.0f);
            break;
        }
        case ITEM: {
            Item* item = (Item*)object;
            if (item->isDurationTimeout()) {
                deactivating.push(item);
                continue;
            }
            item->move(90.0f);
            break;
        }
    }
    }
    while (!deactivating.empty()) {
        Object* object = deactivating.top();
        deactivating.pop();
        activeObjects.remove(object);
        pool.push(object);
    }
}
```

ThirdObject

include/entity/Bullet.hpp
src/entity/Bullet.cpp
include/entity/Item.hpp
src/entity/Item.cpp

Both "Bullet" and "Item" inherit the "Object" class, and "figure" of "Bullet" is defined as FigureType::CIRCLE while "figure" of "Item" is empty but has two triangles as children. A circular node is an object with a figure, and a rectangular node is an object without a figure.



Implementation - Airplane

include/entity/Airplane.hpp
src/entity/Airplane.cpp

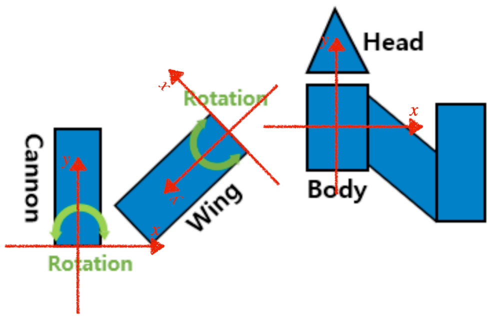
Hierarchical Structure

Airplane construct its parts(body, head, leftArm, rightArm, leftCanon, rightCanon) as an hierarchical structure. All parts are instances of “Object” class, and “Airplane” is derived “Object” class. The code below is member variables of “Airplane” class.

```
private:
    Rect* hitbox; // same pointer as Object::figure;
    Object* body; // Rect
    Object* head; // Triangle
    Object* leftArm; // BaseRect
    Object* rightArm; // BaseRect
    Object* leftCanon; // BaseRect
    Object* rightCanon; // BaseRect
```

The hierarchical model of Airplane is made in the constructor by using each parts’ graph API, and all figures (vertices) are defined in “init()” method.

Default Animations



The origin of model axis of “BaseRect” is middle of base. This can help directly animations of each arm and canon; just rotate. Above picture represents this. Below code is “Airplane::update()” which represents the animation of its arms and canons.

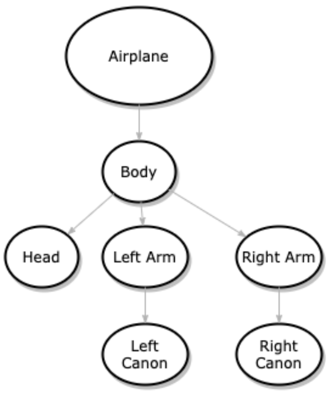
```
void Airplane::update () {
    if (!isAlive())
        return;
    GLfloat degree = idleMotionToken ? 0.2f : -0.2f;
    leftArm->rotate(degree);
    rightArm->rotate(-degree);
    leftCanon->rotate(-degree * 1.2f);
    rightCanon->rotate(degree * 1.2f);
    if (++updateCount > 100) {
        idleMotionToken = !idleMotionToken;
        updateCount = 0;
    }
    Object::update();
}
```

Shotgun

The amount of bullets that can be fired at one time can be increased. The number of bullets that can be fired at one time is defined in member variable "Airplane::shotgunBulletNumber”. When the airplane fires, each bullet's model-view matrix is initialized with a slight change in angle from the Airplane's model-view matrix. The following code shows this.

```
void Airplane::fire (ThirdObjectManager* bulletManager) {
    if (!isAlive())
        return;
    const TransformMatrix& mat = getModelViewMatrix();
    GLfloat addingDegree = 15.0f;
    GLfloat bulletDegree = 0.0f;
    if (shotgunBulletNumber % 2 == 0)
        bulletDegree -= (addingDegree / 2.0f);
    for (int i = 0 ; i < shotgunBulletNumber ; i ++) {
        TransformMatrix bulletMat = mat;
        bulletMat.rotate(bulletDegree += (addingDegree * ( (i % 2 == 1) ? i : -i) ));
        bulletManager->activateObject(bulletMat, BULLET_RADIUS, head->getColor(),
                                      bulletSpeed);
    }
}
```

Below image represent Airplane’s hierarchical structure graph. A circular node is an object with a figure, and a rectangular node is an object without a figure.



Implementation - Planetary System

include/entity/Planetary.hpp
src/entity/Planetary.cpp

Hierarchical Structure

Planetary construct its parts(star, orbit, satellite) as an hierarchical structure. All parts are instances of “Object” class, and “Planetary” is derived “Object” class. The code below is member variables of “Planetary” class.

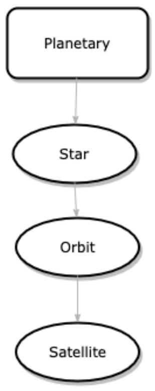
```
private:
    Object* star; // Circle
    Object* orbit; // Circle
    Object* satellite; // Circle
```

The hierarchical model of Planetary is made in the constructor by using each parts’ graph API, and all figures (vertices) are defined in “init()” method.

Default Animations

"satellite" is a child of "orbit" and "orbit" is a child of “star". Initially, translate is applied only to each child object, and by rotating each parent object every frame, we can complete a planetary system in which the child orbits the parent.

```
void Planetary::init (const TransformMatrix& mat, const GLfloat starRadius) {
    GLfloat orbitRadius = starRadius / 2.0f;
    GLfloat satelliteRadius = orbitRadius / 3.0f;

    setMatrix(mat);
    rotate(randomRealNumber(0.0f, 360.0f));
    ((Circle*)**star)->setRadius(starRadius);
    ((Circle*)**orbit)->setRadius(orbitRadius);
    ((Circle*)**satellite)->setRadius(satelliteRadius);

    orbit->setTranslate(0.0f, 3.0f * starRadius);
    satellite->setTranslate(2.0f * orbitRadius, 0.0f);

    star->setRandomColor();
    star->setColorAlpha(0.5f);
    orbit->setRandomColor();
    orbit->setColorAlpha(0.5f);
    satellite->setRandomColor();
    satellite->setColorAlpha(0.5f);
}

void Planetary::update () {
    star->rotate(starAngle);
    orbit->rotate(orbitAngle);
    Object::update();
}
```



Additional Implementation

1. User interface for player’s lives and test of stage and mode.
    include/entity/Ui.hpp
    src/entity/Ui.cpp
   “Ui” class also included in scene graph.

2. When an enemy is destroyed, the dropped item bounces in a random direction when it hits a wall, and if the player does not acquire it, it disappears after a certain period of time.

How to Build and Run

Windows (x64)

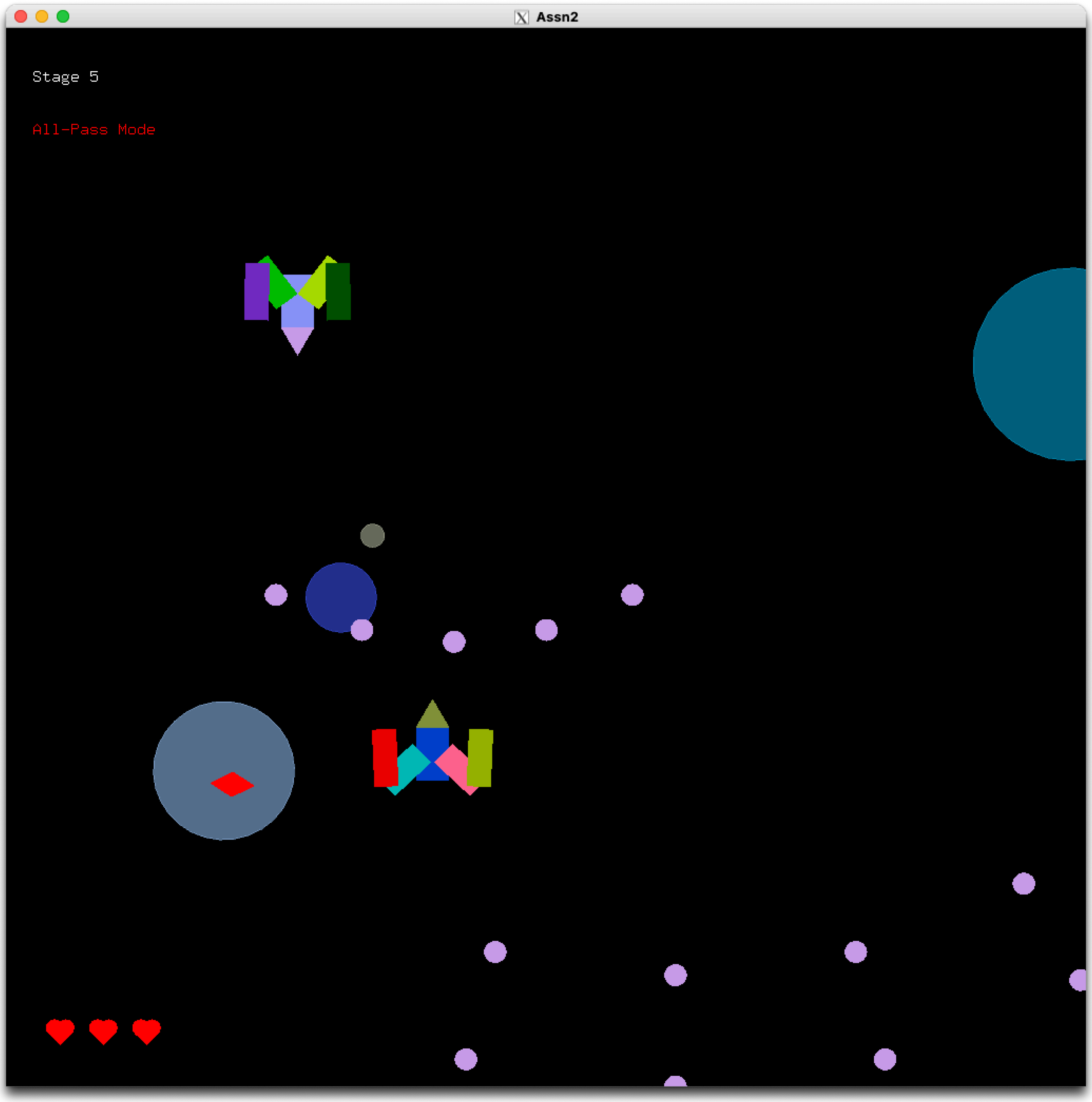
Requirements

- 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



- The above airplane is enemy, and the below one is player.
- Bullets are circle, and its color is same as head of airplane.
- The black area is the playable zone.
- The heart shapes in left-bottom of the screen are remaining lives of the player.
- The text in left-top of the screen means this stage. If ‘all pass mode’ or ‘all fail mode’ is activated, it also notified in red text.

Debates

Hierarchical Structures of Model and Scene Graph

At first, I tried to implement the graph that composes the model hierarchically into the "FigureNode" class and the graph composing the scene graph into the "ObjectNode" class. However, while implementing, I realized that the two graphs were essentially the same concept, and eventually let the "Object" class organize all of this.

Depth of Objects

The planetary system background cannot come in front of the Airplane, and conversely, since the Airplane should not obscure the UI, a way to control the relative depth between objects was needed. For this, the z value of the translation vector was used. In the "Object::pushChild(Object* child, const int priority)" method that takes a "Object" object as an argument, the depth of the child is changed by 1.0, and the depth of the child is changed by 0.01 in the "Object::pushChild(const int figureType, const int priority)” method that takes a "figureType" as an argument. This is because the former method is used to create a scene graph, and the latter method is used to create a hierarchical structure of the model.

```
Object* Object::pushChild (Object* child, const int priority) {
    pushChild(child);
    const GLfloat cmpz = modelViewMat.tz;
    if (priority == FRONT)
        child->translate(0.0f, 0.0f, cmpz >= Window::MIN_DEPTH ? cmpz : cmpz + 1.0f);
    else if (priority == BACK)
        child->translate(0.0f, 0.0f, cmpz <= Window::MAX_DEPTH ? cmpz : cmpz - 1.0f);
    return child;
}

Object* Object::pushChild (const int figureType, const int priority) {
    Object* child = pushChild(figureType);
    const GLfloat cmpz = modelViewMat.tz;
    if (priority == FRONT)
        child->translate(0.0f, 0.0f, cmpz >= Window::MIN_DEPTH ? cmpz : cmpz + 0.01f);
    else if (priority == BACK)
        child->translate(0.0f, 0.0f, cmpz <= Window::MAX_DEPTH ? cmpz : cmpz - 0.01f);
    return child;
}
```

Movement of Objects

Unlike in the previous project where coordinate values were directly translated using a direction macro, in this assignment, we used the method of obtaining the position of the next frame with a trigonometric ratio using the angle of the direction the object will move in the next animation. Thanks to this, we were able to get smooth movement in any situation.

Conclusion, Improvements

Conclusion

Doing this assignment, we learned and discussed the following:

1. How to construct a scene graph, and how to draw vertices by recursively visiting the nodes of the scene graph using "glPushMatrix()" and “glPopMatrix()".

Improvements

Memory Optimization by Creating Objects as Singleton Classes.

In this program, all objects are allocated in memory. Therefore, it can be optimized by making objects that are created in the same form as “Bullet" as singletons so that draw its vertices by receiving only the model-view matrix and color as parameters.

Resources

The idea of scene graph

https://research.ncl.ac.uk/game/mastersdegree/graphicsforgames/scenegraphs/Tutorial%206%20-%20Scene%20Graphs.pdf

Applying alpha of color

https://learnopengl.com/Advanced-OpenGL/Blending

Applying depth of vertices

https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glDepthFunc.xhtml

Drawing circle and triangle

https://stackoverflow.com/questions/22444450/drawing-circle-with-opengl