# Assignment #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 game is designed to be object-oriented.
- “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” method of “GamePlay” object.

## Objects

- There are two types of objects in the game; Bullet and Airplane.
- All objects inherit a rectangle. Therefore, every objects are rectangle.
- All objects renew its status in each frame, by called “update” method.
- All objects draw itself by its “display()” method, and “GamePlay::render()” is the starting point.

## User Input

- There are two types of keyboard input: discrete type and async type.
  - Discrete key is entered only once even the keyboard are held down continuously.
    - “c (all pass)”, “f (all fail)” and “space (fire a bullet)” keys are discrete keys. So you can’t repeating fire, but just quickly press the spacebar several times.
  - Async key is entered continuously if the keyboard remains pressed.
    - Arrow keys for movement of the player airplane are async keys.

# Development Environment

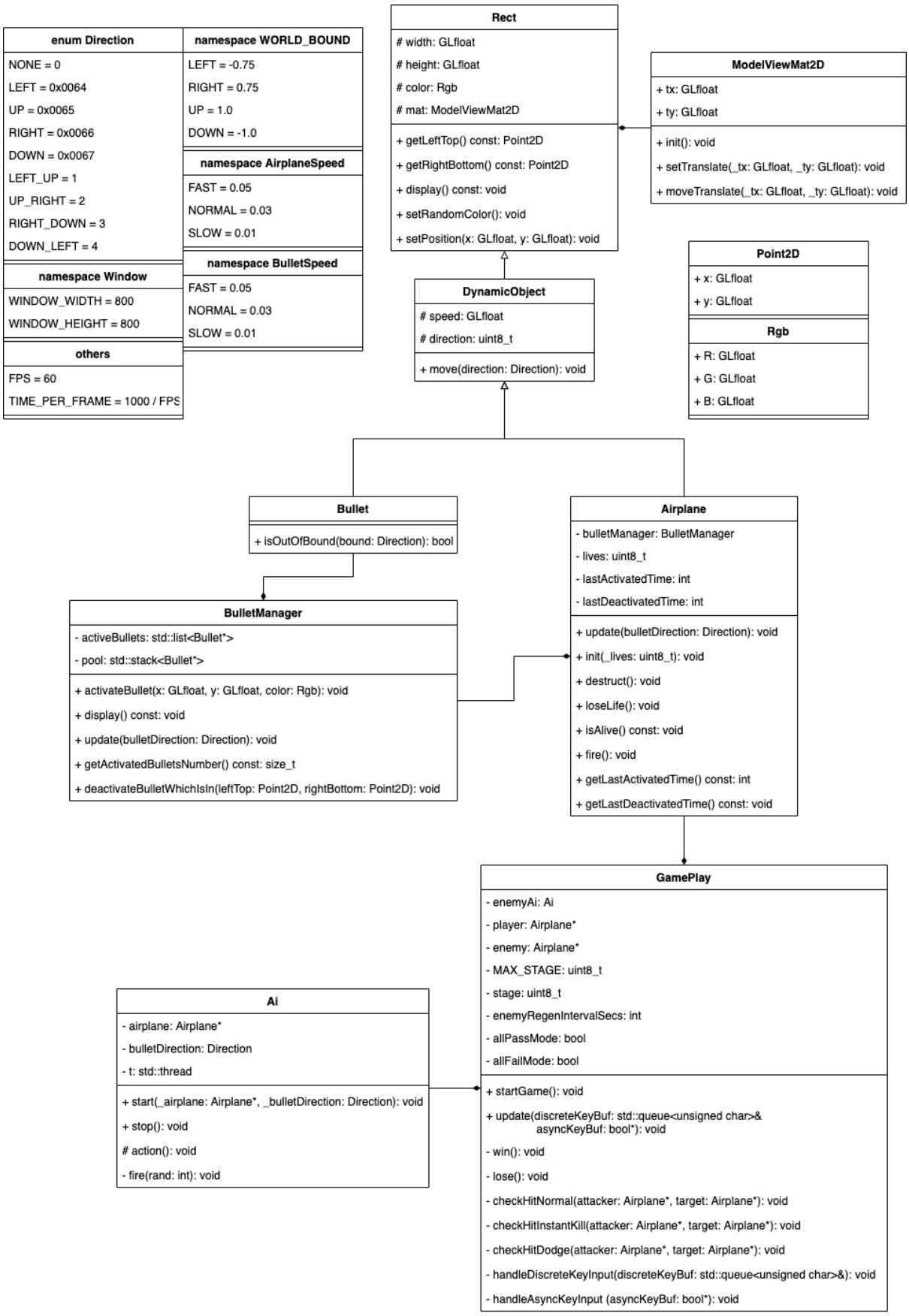
## Windows (x64)

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

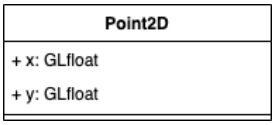
## macOS

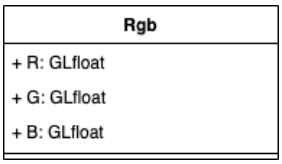
- Visual Studio Code 1.54.3
- freeglut 3.2.1
- glew 2.2.0
- glm 0.9.9.8

# UML


enum Direction
NONE = 0
LEFT = 0x0064
UP = 0x0065
RIGHT = 0x0066
DOWN = 0x0067
LEFT_UP = 1
UP_RIGHT = 2
RIGHT_DOWN = 3
DOWN_LEFT = 4
namespace WORLD_BOUND
LEFT = -0.75
RIGHT = 0.75
UP = 1.0
DOWN = -1.0
namespace AirplaneSpeed
FAST = 0.05
NORMAL = 0.03
SLOW = 0.01
namespace BulletSpeed
FAST = 0.05
NORMAL = 0.03
SLOW = 0.01
namespace Window
WINDOW_WIDTH = 800
WINDOW_HEIGHT = 800
others
FPS = 60
TIME_PER_FRAME = 1000 / FPS
Rect
# width: GLfloat
# height: GLfloat
# color: Rgb
# mat: ModelViewMat2D
+ getLeftTop() const: Point2D
+ getRightBottom() const: Point2D
+ display() const: void
+ setRandomColor(): void
+ setPosition(x: GLfloat, y: GLfloat): void
ModelViewMat2D
+ tx: GLfloat
+ ty: GLfloat
+ init(): void
+ setTranslate(_tx: GLfloat, _ty: GLfloat): void
+ moveTranslate(_tx: GLfloat, _ty: GLfloat): void
DynamicObject
# speed: GLfloat
# direction: uint8_t
+ move(direction: Direction): void
Point2D
+ x: GLfloat
+ y: GLfloat
Rgb
+ R: GLfloat
+ G: GLfloat
+ B: GLfloat
Bullet
+ isOutOfBound(bound: Direction): bool
Airplane
- bulletManager: BulletManager
- lives: uint8_t
- lastActivatedTime: int
- lastDeactivatedTime: int
+ update(bulletDirection: Direction): void
+ init(_lives: uint8_t): void
+ destruct(): void
+ loseLife(): void
+ isAlive() const: void
+ fire(): void
+ getLastActivatedTime() const: int
+ getLastDeactivatedTime() const: void
BulletManager
- activeBullets: std::list<Bullet*>
- pool: std::stack<Bullet*>
+ activateBullet(x: GLfloat, y: GLfloat, color: Rgb): void
+ display() const: void
+ update(bulletDirection: Direction): void
+ getActivatedBulletsNumber() const: size_t
+ deactivateBulletWhichIsIn(leftTop: Point2D, rightBottom: Point2D): void
GamePlay
- enemyAi: Ai
- player: Airplane*
- enemy: Airplane*
- MAX_STAGE: uint8_t
- stage: uint8_t
- enemyRegenIntervalSecs: int
- allPassMode: bool
- allFailMode: bool
+ startGame(): void
+ update(discreteKeyBuf: std::queue<unsigned char>&
asyncKeyBuf: bool*): void
- win(): void
- lose(): void
- checkHitNormal(attacker: Airplane*, target: Airplane*): void
- checkHitInstantKill(attacker: Airplane*, target: Airplane*): void
- checkHitDodge(attacker: Airplane*, target: Airplane*): void
- handleDiscreteKeyInput(discreteKeyBuf: std::queue<unsigned char>&): void
- handleAsyncKeyInput (asyncKeyBuf: bool*): void
Ai
- airplane: Airplane*
- bulletDirection: Direction
- t: std::thread
+ start(_airplane: Airplane*, _bulletDirection: Direction): void
+ stop(): void
# action(): void
- fire(rand: int): void

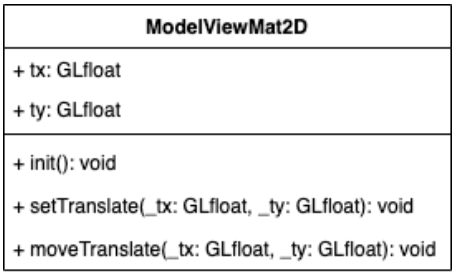
# Data Structures

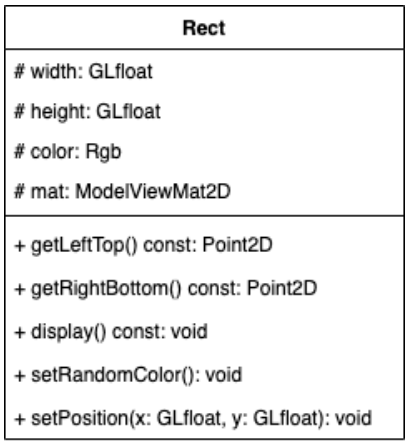



**Point2D** Represent a point $(x, y)$.

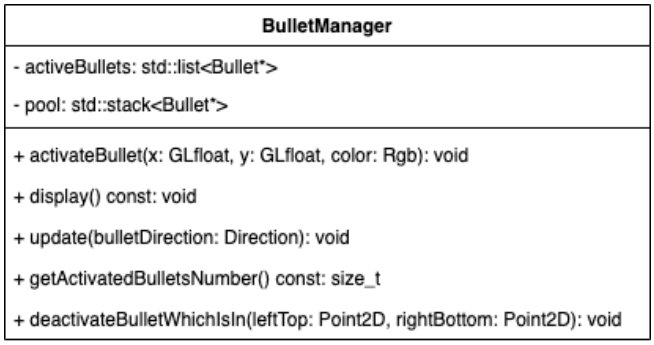
**Rgb** Represent a color. The range of each value is 0.0 to 1.0




**ModelViewMat2D** Represent a model-view matrix for transformation from model frame to world space.

**Rect** Represent a rectangle. display() methods is called for drawing itself.




**BulletManager** Data structure for managing bullets of one airplane. The object pooling technique was used to prevent bullets from frequently allocating and freeing memory. After allocating 100 bullets, it manages the bullets by activating and deactivating them.

# Implementation

## Implementation - Based on Fixed Frame Rate

This game is designed to run in 60 fps. In main function, glutIdleFunc() calls updateFrame() in idle time. updateFrame() function checks to see if it is time for a new buffer to be swapped, and if true, executes it by calling glutPostRedisplay().

```
void updateFrame () {
    const int NOW_TIME = glutGet(GLUT_ELAPSED_TIME);
    if (NOW_TIME - lastRenderTime < TIME_PER_FRAME)
        return;
    gameplay.update(discreteKeyBuf, asyncKeyBuf);
    glutPostRedisplay();
    lastRenderTime = NOW_TIME;
}
```

and gameplay.update() method brings about renewing and drawing all objects in this game.

## Implementation - Handle Events

In updateFrame(), the game examines all possible events and updates all objects accordingly by calling update() method. Some of the events the game should detect are:

- User keyboard inputs.
- Bullet movements.
- Bullet and airplane collision.

### User keyboard inputs

In this game, all keyboard inputs that can occur are “4-directed arrows", “spaces”, “c", and “f”. All keystrokes are first detected in glutKeyboardFunc, glutSpecialFunc, or glutSpecialUpFunc in main function. The keystrokes detection from glut, however, is discrete so it responds as if you hold down a single keyboard in a text editor. This is not a problem when the “c”, “f” and “spaces” keys are pressed in this game. However, in order to implement the movement of the airplane, it should not react this way (i.e., instead of changing the state of the object with the feedback immediately upon receiving the key), but the object must recognize the state in which a specific arrow key is pressed so that the object changes its position in each frame. Below code is handle these things.

```
bool asyncKeyBuf[256];
std::queue<unsigned char> discreteKeyBuf;

/** @brief GLUT callback. Detect “c”, “f”, and “spaces” keys down. */
void keyboardDown (unsigned char key, int x, int y) {
    discreteKeyBuf.push(key);
}

/** @brief GLUT callback. Detect arrow keys down. */
void specialKeyboardDown (int key, int x, int y) {
```

```
    asyncKeyBuf[key] = true;
}

/** @brief GLUT callback. Detect arrow keys up. */
void specialKeyboardUp (int key, int x, int y) {
    asyncKeyBuf[key] = false;
}

// ..in main function
    glutKeyboardFunc(keyboardDown);
    glutSpecialFunc(specialKeyboardDown);
    glutSpecialUpFunc(specialKeyboardUp);
// ..
```

The direction keys, which are the only special keys in this game, are detected by specialKeyboardDown() and specialKeyboardUp(), and whether they are pressed in real time is updated in the asyncKeyBuf table. All other keys are put in the discreteKeyBuf queue in the order they were pressed. Then, asyncKeyBuf and discreteKeyBuf are passed into *GamePlay*::update() method in updateFrame() function, and below two methods of *GamePlay* handles each keyboard inputs.

```
void GamePlay::handleDiscreteKeyInput (std::queue<unsigned char>& discreteKeyBuf) {
    while (!discreteKeyBuf.empty()) {
        unsigned char key = discreteKeyBuf.front();
        discreteKeyBuf.pop();
        switch (key) {
            case ' ':
                if (!allFailMode) player->fire(); break;
            case 'c':
                if (!allPassMode) {
                    allPassMode = true; allFailMode = false;
                }
                else
                    allPassMode = false;
                break;
            case 'f':
                if (!allFailMode) {
                    allFailMode = true; allPassMode = false;
                }
                else
                    allFailMode = false;
                break;
        }
    }
}

void GamePlay::handleAsyncKeyInput (const bool* asyncKeyBuf) {
    const bool* buf = asyncKeyBuf;
    if (buf[GLUT_KEY_LEFT] && !buf[GLUT_KEY_UP] && !buf[GLUT_KEY_RIGHT] && !buf[GLUT_KEY_DOWN])
        player->move(LEFT);
    else if (!buf[GLUT_KEY_LEFT] && buf[GLUT_KEY_UP] && !buf[GLUT_KEY_RIGHT] && !buf[GLUT_KEY_DOWN])
        player->move(UP);
    else if (!buf[GLUT_KEY_LEFT] && !buf[GLUT_KEY_UP] && buf[GLUT_KEY_RIGHT] && !buf[GLUT_KEY_DOWN])
        player->move(RIGHT);
    else if (!buf[GLUT_KEY_LEFT] && !buf[GLUT_KEY_UP] && !buf[GLUT_KEY_RIGHT] && buf[GLUT_KEY_DOWN])
        player->move(DOWN);
    else if (buf[GLUT_KEY_LEFT] && buf[GLUT_KEY_UP] && !buf[GLUT_KEY_RIGHT] && !buf[GLUT_KEY_DOWN])
        player->move(LEFT_UP);
    else if (!buf[GLUT_KEY_LEFT] && buf[GLUT_KEY_UP] && buf[GLUT_KEY_RIGHT] && !buf[GLUT_KEY_DOWN])
        player->move(UP_RIGHT);
    else if (!buf[GLUT_KEY_LEFT] && !buf[GLUT_KEY_UP] && buf[GLUT_KEY_RIGHT] && buf[GLUT_KEY_DOWN])
        player->move(RIGHT_DOWN);
    else if (buf[GLUT_KEY_LEFT] && !buf[GLUT_KEY_UP] && !buf[GLUT_KEY_RIGHT] && buf[GLUT_KEY_DOWN])
        player->move(DOWN_LEFT);
}
```

### Bullet movements

Unlike airplanes that rely on user input, bullets must move continuously at a constant speed and direction. The speed is defined in *DynamicObject* class, which is the parent class of *Bullet* class. And in the update() method of *BulletManager* class that manages *Bullet* objects, the locations of all *Bullets* are updated. This *BulletManager*::update() method also check if the bullet has reached the border of the screen.

```
void Airplane::BulletManager::update (const int bulletDirection) {
    std::stack<Bullet*> deactivating;
    for (Bullet* bullet : activeBullets) {
        if (bullet->isOutOfBound(bulletDirection)) {
            deactivating.push(bullet);
            continue;
        }
        bullet->move(bulletDirection);
    }
    while (!deactivating.empty()) {
        Bullet* bullet = deactivating.top();
        deactivating.pop();
        activeBullets.remove(bullet);
        pool.push(bullet);
    }
}
```

*BulletManager* is nested class in *Airplane* class, so its update() method is called in *Airplane*::update().

### Bullet and airplane collision

Below methods handle the collision between airplane and bullet.

```
void GamePlay::checkHitNormal (Airplane* attacker, Airplane* target) {
    if (!target->isAlive())
        return;
    if (attacker->bulletManager.deactivateBulletWhichIsIn(target->getLeftTop(),
                                                          target->getRightBottom())) {
        target->loseLife();
        if (!target->isAlive()) {
            if (target == player)
                lose();
            if (target == enemy) {
                enemyAi.stop();
                if (stage == MAX_STAGE)
                    win();
            }
        }
        if (target == player)
            player->setRandomColor();
    }
}
```

This is a method of inspecting whether the attacker's bullet is in the target's rectangle and then processing. Therefore, the hit-box of the airplane is its rectangle.

There are other two methods of checking collision.

```
void GamePlay::checkHitInstantKill (Airplane* attacker, Airplane* target);
void GamePlay::checkHitDodge (Airplane* attacker, Airplane* target);
```

The first one is kill target immediately if its hit. The second one is do nothing if the target is hit. These two methods are needed in all pass mode and all fail mode.

## Implementation - Renew Objects

Every objets in this game have update() method to renew itself. In every frame, *GamePlay*::update() is called and it calls update() methods of other objects to apply the changes.

```
// in main.cpp, Callback function for glutIdleFunc()
void updateFrame () {
// ..
    gameplay.update(discreteKeyBuf, asyncKeyBuf); // GamePlay Class
// ..
}

void GamePlay::update (std::queue<unsigned char>& discreteKeyBuf, const bool* asyncKeyBuf) {
// ..
    player->update(UP); // Airplane Class
    enemy->update(DOWN); // Airplane Class
// ..
}

void Airplane::update (const int bulletDirection) {
    bulletManager.update(bulletDirection); // BulletManager Class
}

void Airplane::BulletManager::update (const int bulletDirection) {
// ..
    for (Bullet* bullet : activeBullets)
        bullet->move(bulletDirection); // Bullet Class
// ..
}
```

## Implementation - Draw Objects

Like renewing all objects, every objects in this game have display() method to draw itself in glut window. *GamePlay*::render() is called first and it calls other display() methods of objects.

```
// in main.cpp, Callback function for glutDisplayFunc()
void display () {
    glClear(GL_COLOR_BUFFER_BIT);
    gameplay.render(); // GamePlay Class
    glutSwapBuffers();
}

void GamePlay::render () {
    player->display(); // Airplane Class
    enemy->display(); // Airplane Class
    displayWall();
    displayStage();
    displayPlayerLives();
}

void Airplane::display () const {
    bulletManager.display(); // BulletManager Class
    if (isAlive())
        DynamicObject::display(); // Call method of its parent class
```

```
}

void Airplane::BulletManager::display () const {
    for (Bullet* bullet : activeBullets)
        bullet->display(); // Bullet Class
}
```

Finally, Rect class, which is base class of Airplane and Bullet, call its display() method to draw itself in world coordinate.

```
void Rect::display () const {
// Airplane::display() and Bullet::display()
// call this method for drawing itself.
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(mat.tx, mat.ty, 0.0f);
    glColor3f(color.R, color.G, color.B);
    GLfloat w = width / 2;
    GLfloat h = height / 2;
    glBegin(GL_POLYGON);
        glVertex2f(-w, -h);
        glVertex2f(-w, h);
        glVertex2f(w, h);
        glVertex2f(w, -h);
    glEnd();
    glBegin(GL_LINE_LOOP);
        glVertex2f(-w, -h);
        glVertex2f(-w, h);
        glVertex2f(w, h);
        glVertex2f(w, -h);
    glEnd();

}
```

# Additional Implementation

### Visualization of Player's Lives

```
void GamePlay::displayPlayerLives () {
    for (int i = 0 ; i < player->getLives() ; i ++) {
        Rect rect(0.05, 0.05);
        rect.setColor(1.0f, 0.0f, 0.0f);
        rect.setPosition(-0.9f + (i * 0.08), -0.9f);
        rect.display();
    }
}
```

### Text Output for Stage and Mode

```
void GamePlay::displayStage () {
    std::string str = "Stage " + std::to_string(stage);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(WORLD_BOUND::RIGHT + 0.05f, 0.9f, 0.0f);
    glColor3f(0.0f, 0.0f, 0.0f);
    glRasterPos2d(0.0f, 0.0f);
    for (int i = 0 ; i < str.size() ; i ++)
        glutBitmapCharacter(GLUT_BITMAP_9_BY_15, str[i]);

    if (allPassMode || allFailMode) {
        if (allPassMode)
            str = "all pass";
        else if (allFailMode)
            str = "all fail";
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        glTranslatef(WORLD_BOUND::RIGHT + 0.05f, 0.8f, 0.0f);
        glColor3f(1.0f, 0.0f, 0.0f);
        glRasterPos2d(0.0f, 0.0f);
        for (int i = 0 ; i < str.size() ; i ++)
            glutBitmapCharacter(GLUT_BITMAP_9_BY_15, str[i]);
    }
}
```

...and above two additional implementations are executed in GamePlay::render().

```
void GamePlay::render () {
    player->display();
    enemy->display();
    displayWall();
    displayStage();
    displayPlayerLives();
}
```

The result Screen in below section includes these.

# How to Build and Run

## Windows (x64)

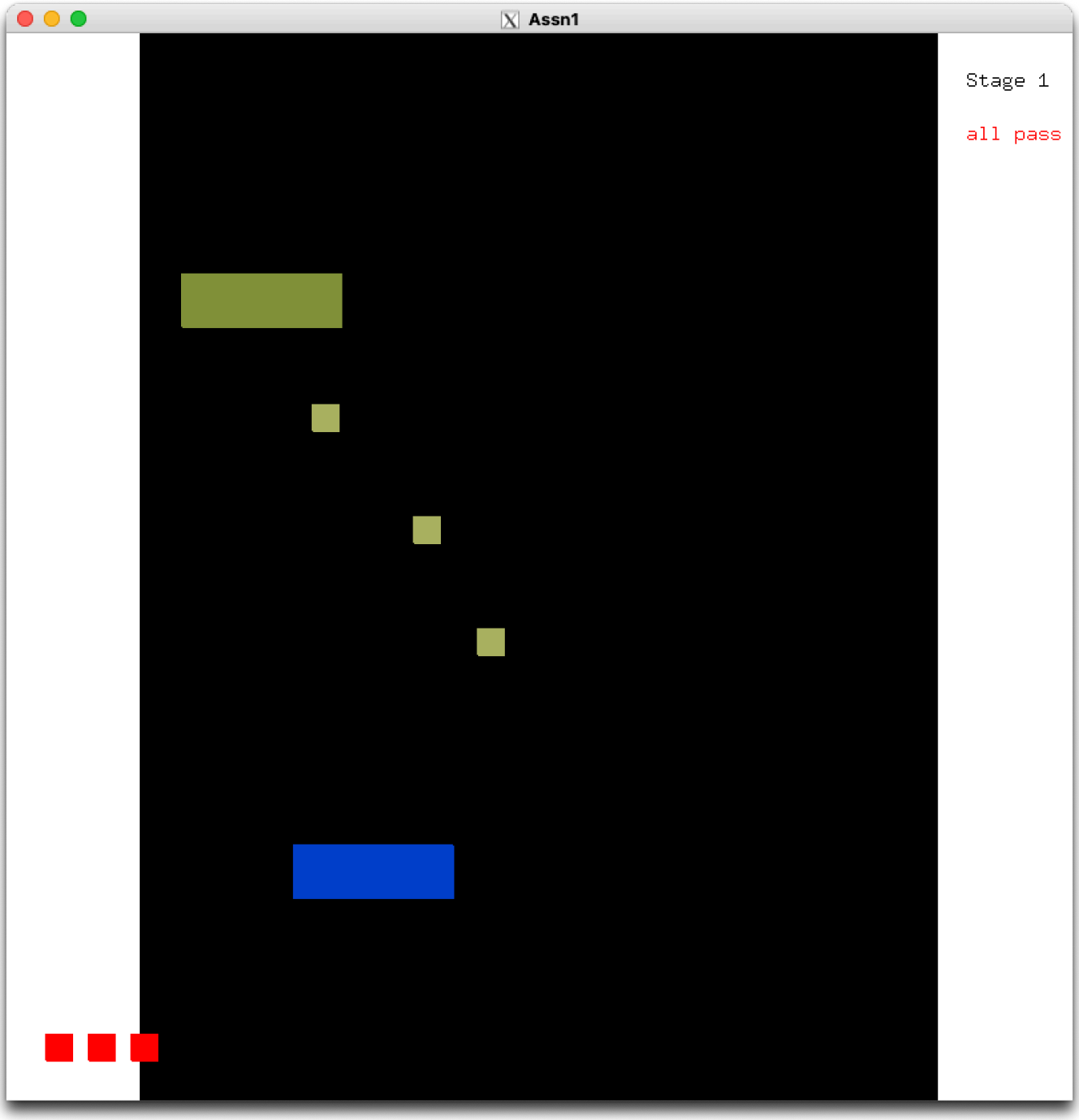
***Requirements***

- 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 green rectangle is airplane of the enemy, and the blue rectangle is airplane of the player.
- Bullets are square, and the color is a little lighter tone than the color of the fired airplane.
- The black area is the playable zone, and the white zones of both sides are walls.
- The red squares in left-bottom of the screen are remaining lives of the player.
- The text in right-top of the screen means this stage. If ‘all pass mode’ or ‘all fail mode’ is activated, it also notified in red text.

# Debates

### Data Structures

We wanted to let our code has easily expandability for next assignments. The OpenGL concept, however, is like a state machine so that it was hard to define what attributes our classes must have for conversation with OpenGL system. Finally, we defined “ModelViewMat2D” class and let our “Rect” class to have it as its attribute so that the “Rect” class is transformed based on its model-view matrix attribute when the “Rect::display()” method is called. It’s still confused because we didn’t know if other games were designed this way.

### Display Order of Objects

We found that the drawn vertex in later covers the already drawn vertex in OpenGL. We applied this in the “GamePlay” class that holds all the objects and draws them through its “render()” method; simply the drawing order of objects. It seems that the scene graph is managing this efficiently, but it has not been applied yet.

### Scale in Model Frame and World Space

The size of the “Rect” object is defined in its constructor, and this size is mapped 1:1 to World Space, so there is no need to apply scale transformation in the model view frame.

### Handling Events

There were several events to update objects. We planned that these all events were dealt in “GamePlay” class. However, we thought there might be needed more structured way when the events were more complicated; multiple enemies, firing bullets in diverse directions, etc. “EventHandler” class which is only deals with these in-game events maybe good solution to this.

# Conclusion, Improvements

## Conclusion

Doing this assignment, we learned and discussed the following:

1. Structure of the OpenGL API.
2. Things to consider when mapping our objects to OpenGL API and displaying them on the screen.
3. How to update objects based on frame rate.

## Improvements

### Modularization for OpenGL Functionality

We'd found GLSL solution in web, which is for drawing vertices. If we apply it in our project later, our classes need to be changed. In our classes of objects, only elements defined in the model frame need to be included, and the contents necessary to convert them to world space need to be separated from them and managed separately.

### Hit Detection

There were only two airplanes in this assignment, so detecting hit was simple. Our implementation, however, was inefficient because it always took $O(b \cdot n)$ times where $b$ is the number of bullets activated and $n$ is the number of airplanes. If the world space can be divided into arbitrary grids and objectified, and the position of each airplane can be defined in rows or cols in grid of the world space, it will be more efficient because each bullet only needs to inspect the airplane that exists in the bullet's orbit.

# Resources

For the code to manage the frame rate, we refer to the following:

https://community.khronos.org/t/displaying-sequence-of-frames-in-loop-on-projector/70174

The idea of dividing keystrokes into discrete and async and handling them accordingly is referenced following:

https://stackoverflow.com/questions/2807053/how-to-make-smooth-movements-in-opengl