UNIVERSITY OF LETHBRIDGE

INTRODUCTION TO SOFTWARE ENGINEERING

CPSC 2720

PROJECT REPORT

TEAM DELTA

DELTA FORCE

Professor in charge: Robert Benkoczi

Members:

<u>Yuhai Shi</u>
Wang Kangning
Jefferson Sylva-Iriogbe

DELTA FORCE PLOT

This game is an endurance aircraft combat game which was developed using c++ 11 and Allegro. This game was developed to be played using a computer keyboard. The game starts out from a start menu and the user has to push the ENTER button to start playing.

In this game there are three kinds of entities. The first entity is the player entity itself, which starts out on the left side of the screen as an aircraft and there are two enemy entities(one substantially larger than the other) which come from the right side of the screen during game-play. While the game is in play, the user tries to avoid the oncoming enemy entities. The player character has the ability to shoot missiles which destroy the smaller enemy on contact. The missiles have no effect on the larger enemy entity, so the player has to avoid it in order to stay in the game. If the player entity comes in contact with any of the enemy entities, the game ends and the user is kicked out to the game over screen.

When the user enters into the game, a counter representing the amount time spent alive in the game is implemented to start counting from 0 counting till the player entity dies. The amount of time spent in the game is displayed on the game over screen.

IMPLEMENTATION

As mentioned earlier, this game was developed using c++ 11 and Allegro. The source code for this game is implemented with **fifteen** different source files which each represent an important functionality of the game(Player character, Enemy Character, Game-play organization..) and a Makefile which is used for building all **fifteen** files to form one executable file. Four additional files are contained in the tests folder used to implement the test driven development method.

The player object has basic characteristics like its x and y axis which is used throughout the game to track the position of the player. The movement of the player is initialized with the Keyboard to move up along the y axis when the UP button is pushed, forward when the RIGHT button is pushed, down when the DOWN button is pushed and backwards when the BACK button is pushed. The SPACE button is initialized to release missiles from the aircraft entity whenever it is pushed. Every entity in this game is shown on the screen using the al_draw_bitmap function.

The game starts off with an image of the start page which tells the user to push ENTER to start the game. This game makes use of enum type which has three states, TITLE, PLAY and LOSE. When the user opens the game, the state of the game is set to TITLE, when the user pushes ENTER to start playing, the state of the game is set to PLAY and when the user dies, the state of the game is set to LOST.

The keyboard files define the Keyboard object to allow six controls throughout this game which are initialized to be ENTER, SPACE, UP, DOWN, LEFT and RIGHT.

A **behavioral variation** technique was used to implement the enemy entities as seen in the EnemyOne.h and Enemy.h implementation. The enemy entities both inherit from one enemy class but are implemented differently to have different bitmap images and different functionality like the speed at which they move and the number of enemy types that are created at every point. The smaller enemies are implemented to continuously populate randomly from the left side of the screen while the larger enemy is implemented to have only one on the screen and moves faster. The smaller enemy types can be destroyed with the bullets but the larger enemy cannot, so the user has to avoid it. A new enemy could easily be **added** by creating a new enemy characteristic and inheriting the characteristics of the EnemyOne class and then the modifier can define the new enemy to have a different set of characteristics.

The bullets are implemented with the use of lists which allows for the user to shoot continuously.
Code extract from bullet.cc :

```
for(list<bulletType*>::iterator it=Blist.begin();it!=Blist.end();it++)
    al_draw_bitmap(Bbitmap,(*it)->getbX(),(*it)->getbY(),NULL);
```

When the bullets come in contact with the smaller enemy type a collision is implemented with a bitmap image and an explosion sound is simultaneously played to emulate a real explosion. When the bullet collides with the smaller enemy, the bullet is immediately removed from the screen to a point outside of the screen size to imitate a vanishing effect. Two types of collision is implemented in this game in the Allegro.cc file – bullet-enemy collision and player-enemy collision. The player-enemy collision results in the user getting kicked out to the game over screen.

The other parts of the game (background displays, the moon images, start image, game over image, time counter features amongst others) are implemented in the Allegro files. The background images are implemented using overlapping bitmap images to move at different speeds hence creating the interesting background display while the time counter was implemented various time functions available with the allegro library.

The game also makes use of various sounds throughout the game. The sound resulting from shooting a missile, collision between a bullet and an enemy and the game sound that is played throughout the game play is implemented using the al_play_sample function which is a part of the allegro library.

TEST DRIVEN DEVELOPMENT

Implemented in the tests folder are some tests for the two types of collision that were implemented which are bullet-enemy collision and player-enemy collision. In these test cases, the functions that were used to create the collisions in the game were imported into the TDD method to check if the boolean value holds the correct value for a collision between two objects that actually collide (true) and two objects that don't (false) as shown below;

This code is extracted from the test file in the tests folder:

```
void collisiontest()
{
  CPPUNIT_ASSERT(collision(35, 35, 35, 35, 30, 35, 30, 35) == true);
}
void collisiontest1()
{
  CPPUNIT_ASSERT(collision(35, 35, 200, 200, 10, 10, 10, 10) == false); //out of range
}
void bulletcollsiontest()
{
  CPPUNIT_ASSERT(collision1(35, 35, 35, 35, 20, 10, 30, 35) == true);
}

void bulletcollsiontest1() {
  CPPUNIT_ASSERT(collision1(35, 35, 200, 200, 20, 10, 30, 35) == false); //out of range
}
```

METRIC SUMMARY

This table below shows measures over the project as a whole.

Metric	Tag	Overall	Per Module
Number of modules	NOM	24	
Lines of Code	LOC	664	27.667
McCabe's Cyclomatic Number	MVG	108	4.500
Lines of Comment	COM	424	17.667
LOC/COM	L_C	1.566	
MVG/COM	M_C	0.255	
Information Flow measure (inclusive)	IF4	15	0.625
Information Flow measure (visible)	IF4v	14	0.583
Information Flow measure (concrete)	IF4c	6	0.250
Lines of Code rejected by parser	REJ	32	

The metric results were generated using the recommended tool CCCC (C and C++ Code Counter, ***http://sourceforge.net/projects/cccc/)***

Legend for the Metric Summary table:

NOM = Number of modules
Number of non-trivial modules identified by the analyzer. Non-trivial modules include all classes, and any other module for which member functions are identified.

- LOC = Lines of Code
  Number of non-blank, non-comment lines of source code counted by the analyzer.
- COM = Lines of Comments
  Number of lines of comment identified by the analyzer
- MVG = McCabe's Cyclomatic Complexity
  A measure of the decision complexity of the functions which make up the program. The strict definition of this measure is that it is the number of linearly independent routes through a directed acyclic graph which maps the flow of control of a subprogram. The analyser counts this by recording the number of distinct decision outcomes contained within each function, which yields a good approximation to the formally defined version of the measure.
- L_C = Lines of code per line of comment
  Indicates density of comments with respect to textual size of program
- M_C = Cyclomatic Complexity per line of comment
  Indicates density of comments with respect to logical complexity of program
- IF4 = Information Flow measure
  Measure of information flow between modules suggested by Henry and Kafura. The analyser makes an approximate count of this by counting inter-module couplings identified in the module interfaces.

A more detailed version of the CCCC Software Metric Report is contained in the git repository of this project.

EVIDENCE OF THE EFFECTIVENESS OF MEMORY MANAGEMENT SCHEME

We implemented the use of lists to create the bullet entities and the enemy entities. To avoid memory leaks we implemented destructors using **delete** to free up any memory that was being used up in the process.

Using the valgrind command on the linux machines to run this software, we were able to check the effectiveness of our memory management scheme. The summary of the results is displayed below.

```
==5819== HEAP SUMMARY:
==5819==     in use at exit: 1,455,022 bytes in 1,416 blocks
==5819==   total heap usage: 195,564 allocs, 194,148 frees, 942,923,424 bytes allocated
==5819==
==5819== LEAK SUMMARY:
==5819==    definitely lost: 1,146 bytes in 10 blocks
==5819==    indirectly lost: 68,704 bytes in 17 blocks
==5819==      possibly lost: 1,228,800 bytes in 1 blocks
==5819==    still reachable: 156,372 bytes in 1,388 blocks
==5819==         suppressed: 0 bytes in 0 blocks
```

LINK TO THE REPOSITORY

https://jeffersonsylva@bitbucket.org/delta1993/project2720.git