Project (Computer Programming) - 2018

Sibt ul Hussain, Hassan Mustafa, Rafia Rahim

April 25, 2018

Design Deadline (Class Diagram): April 28th, 2018 before 17h30

Deadline Implementation: May 7th, 2018 before 23h30

Instructions

- Make sure that you read and understand each and every instruction. If you have any questions or comments, you are encouraged to discuss your problems with your colleagues (and instructors) on Piazza.

- **Plagiarism is strongly forbidden and will be very strongly punished. If we find that you have copied from someone else or someone else has copied from you (with or without your knowledge) both of you will be punished. You will be awarded (straight zero in the project — which can eventually result in your failure) and appropriate action as recommended by the Disciplinary Committee (DC can even award a straight F in the subject) will be taken.**

- **Try to understand and do the project yourself even if you are not able to complete the project. Note that you will be mainly awarded on your effort not on the basis whether you have completed the project or not.**

- Divide and conquer: since you have around 12 days so you are recommended to divide the complete task in manageable subtasks. We recommend to complete the drawing and design (i.e. number of classes and their relationships) phase as quickly as possible and then focus on the intelligence phase.

- Before writing even one line of code, you must design your final project. This process will require you to break down and outline your program into classes, design your data structure(s), clarify the major functionality of your program, and pseudo-code important methods. After designing your program, you will find that writing the program is a much simpler process.

- **No Marks will be given if you do not submit your class diagram and if you do not use the object oriented design principles you have learned during the course.**

- Imagination Powers: Use your imaginative powers to make this as interesting and appealing as you can think of. An excellent solution can get you bonus marks.

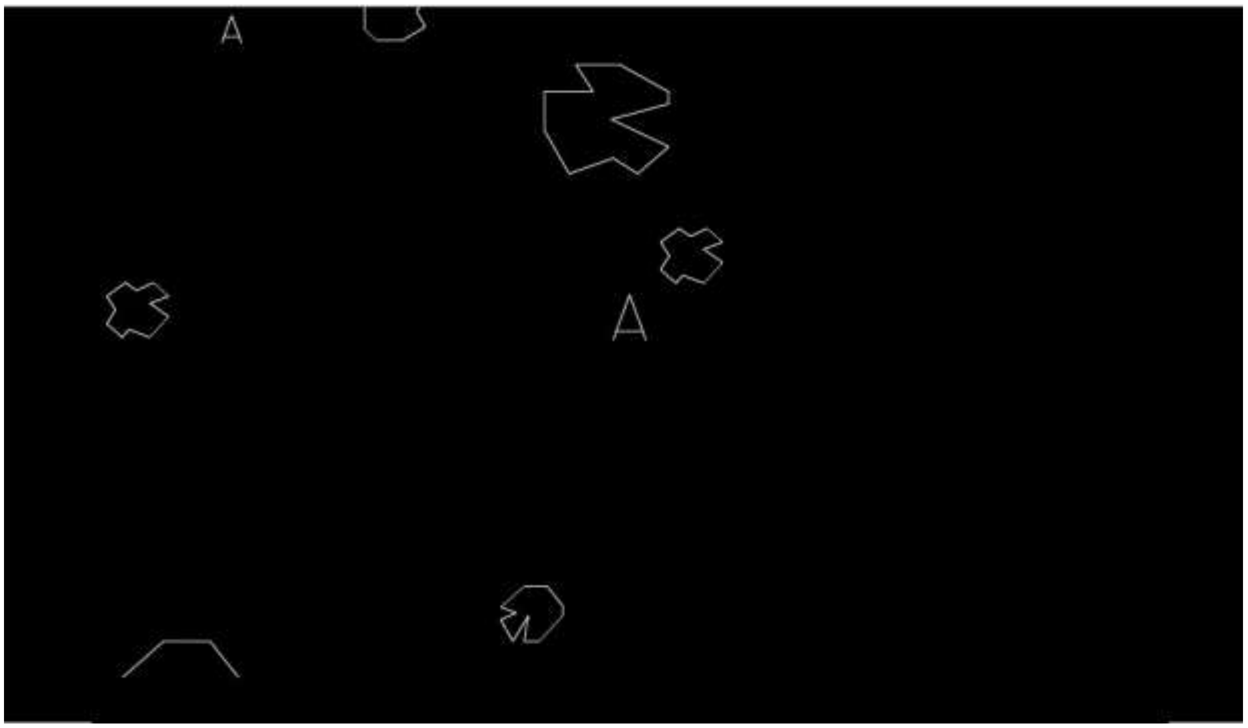
Introduction

Asteroids is an arcade game, developed by *Atari, Inc.* in 1979. The game has been a huge success at the time and has attracted users for its simple yet involving gameplay. The game was inspired by the journey in space so it follows the same physics for the environment.

Gameplay

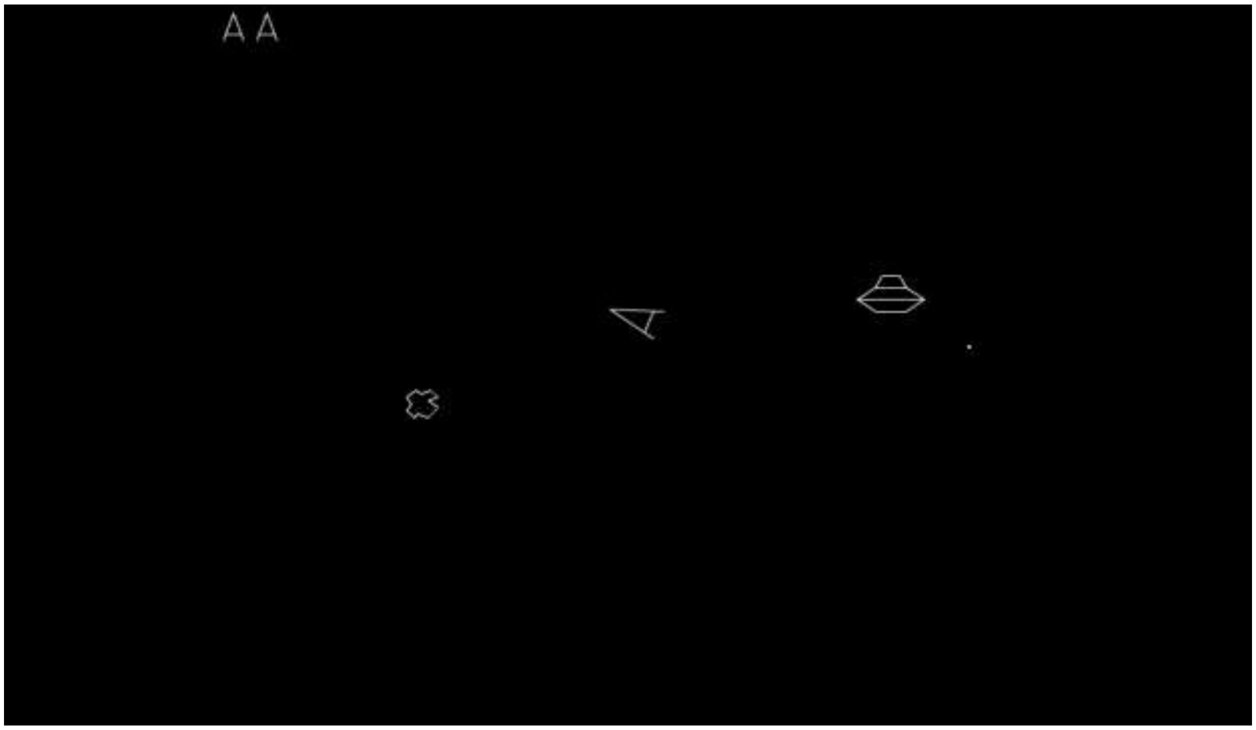
The player of the game controls a *spaceship* in an *asteroid* filled space field. The goal is to destroy the asteroids and enemy ships (*saucers*) while avoiding a collision with any of them. The spaceship can fire bombs to destroy the objects in the space field. The spaceship is destroyed in case of a collision with an object (asteroid or enemy ship) or impact with a bomb fired by the enemy ship. The player can move the spaceship by applying a thrust. The spaceship moves in the direction of the thrust and keeps moving until it stops automatically or the player applies thrust in the opposite direction. As an advanced move, a player can send the spaceship into hyperspace, i.e. a spaceship disappears for a moment and reappears into another random location on the screen. A spaceship may destroy if it reappears at a location where it overlaps an object.

An asteroid is a large random shape object that moves at a constant speed. This game has two types of asteroids (I) Simple, these are normal asteroids and move straight and (ii) Complex that changes their direction every 3 seconds in random direction and start moving in that direction. Once a Simple asteroid is hit by a bomb, it breaks into two smaller complex asteroid pieces, picks a random direction to move, and increases the speed of movement. Once a complex asteroid is hit it breaks into 4 Simple asteroid pieces and they move in 4 different directions. An asteroid is destroyed (and disappears from the field) when it is hit thrice. As an implementation hint you can predefine a set of asteroids for your game and draw them using DrawLine function.



1- Asteroids game environment: A spaceship is placed in the middle as a triangle, with asteroids of different shapes and sizes flying in the space. Remaining lives of the player are displayed on the top left side of the screen.

Enemy ships (saucers) appear on random occasions during the game. An enemy ship fires bombs in random directions and can hit the spaceship. The enemy ship is destroyed if it collides with an asteroid, or gets hit by a bomb. There are two sizes of the enemy ships, where the larger ship is slower and fires fewer bombs (an easy target) while the smaller ship is faster and fires more bombs (a difficult target). The smaller ship also changes its direction randomly wheres as the larger ship always move in the predetermined direction.



2- An enemy spaceship is firing the bombs in random direction.

The space is a wrap-around environment, i.e. an object disappears at one corner of the screen and then reappears from the exact opposite corner. At the start of the game, the number of asteroids in the space field is randomly selected. A next level is achieved once all asteroids in the current field are destroyed. The next level starts with an increased number of asteroids in the space field. The player is awarded three lives initially. A life is decreased if the spaceship is destroyed. The player gets points for hitting the objects. The largest, medium, and smaller asteroids are worth 10, 20, and 40 points respectively. The bigger enemy spaceship is worth 100 points, and smaller spaceship is worth 500 points. A new life is awarded if the player adds 40000 points in the score. The number of remaining lives, and the score is displayed on the top of the screen.

Implementation

You are required to design and implement the *Asteroids* game using Object Oriented Programming. Remember, this project is the perfect opportunity to impress your instructors with your understanding of the OOP concepts. Apparently, a better design of the concepts/classes in the game gets more marks as marks will be deducted for the incorrect design and implementation of the game.

A starter code is provided for reference of gl/glut drawing in C++.

Submission

Deadline to submit the design of the concepts/classes involved in the game is 28th April, 2018 at 17:00, but you can get the design approved earlier by your instructors to start working on the implementation quickly. You can improve the design afterwards, if needed. The submission should be made on Slate as a PDF document and should contain proper figures and explanation of the design.

Deadline to submit the implementation and updated design document is 7th May, 2018 at 10:00 AM.

Bonus

There are bonus marks for implementing any new considerable rules. You should discuss with the instructors before the implementation of any rules that can be claimed for any bonus. The instructors will decide if an implementation is worth bonus marks or not. Visit the link to get ideas for bonus implementation:

https://en.wikipedia.org/wiki/Asteroids_(video_game)

You can also add sounds to different events. Visit the link to get an idea:

http://www.freeasteroids.org/

Deliverables

1. Classes, their members and their relationships

2. Drawing Asteroids, Saucers, their translation

3. Drawing Ship

4. Rotation of Asteroids and ships

5. Collision detection

6. Firing of Missiles and Disappearance of Ships

7. Points and Lives

8. Multiple Levels

Algorithm for Drawing Random Asteroids with N Sides

1. Generate a random radius, R
2. Generate N random points on the circumference of a circle of Radius R
3. Move around the circle and draw straight lines between adjacent points on the circle.

You have got your random shaped Asteroid.

For drawing all shapes, you will need DrawLine Function which is already provided.

Instructions

We have provided complete skeleton code with all the basic drawing functions (can be located in util.h and util.cpp) needed in the project with detailed instructions and documentation. In other words, all you need to know for building the game is provided. Your main task is to understand the main design of the game and then implement it. However, before proceeding with coding, you will need to install some required libraries.

Please note that the provided skeleton already contains code for drawing the board, and asteroids.

Installing libraries on Linux (Ubuntu)

You can install libraries either from the Ubuntu software center or from command line. We recommend to use the command line method and have provided the file “install-libraries.sh” to automate the complete installation procedure. To install libraries:

1. Run the terminal
2. Using cd command, move to the directory that contains the file “install-libraries.sh”.
3. Run the command

```
bash  install-libraries.sh
```

4. provide the password and wait for the libraries to be installed. If you get an error that libglew1.6-dev cannot be found, try installing an older version, such as libglew1.5-dev by issuing following on command line

```
sudo apt-get install libglew1.5-dev
```

5. If you have any other flavour of Linux. You can follow similar procedure to install “OpenGL” libraries.

Compiling and Executing

To compile the game (skeleton) each time you will be using “g++”. However, to automate the compilation and linking process we use a program “make”. Make takes as an input a file containing the names of files to compile and libraries to link. This file is named as “Makefile” in the game folder and contains the details of all the libraries that game uses and needs to linked.

So each time you need to compile and link your program (game) you will be simply calling the “make” utility in the game directory on the terminal to perform the compilation and linking.

```
make
```

That’s it if there are no errors you will have your game executable (You can run the *game* file to see four asteroid shapes randomly placed on the game board). Otherwise try to remove the pointed syntax errors and repeat the make procedure.

Drawing Board and Shapes

Canvas

Since we will be building 2D game, our first step towards building the game will be to define a canvas (our 2D world or 2D coordinate space in number of horizontal and vertical pixels) for drawing the game objects (in our case Asteroids, Spaceship, and Enemy Ships). To define the canvas size, we will be using (calling) the function “SetCanvas” (see below) with two parameters to set the drawing-world width and height in pixels.

```
/* Function sets canvas size (drawing area) in pixels...
*   that is what dimensions (x and y) your game will have
*   Note that the bottom-left coordinate has value (0,0)
*   and top-right coordinate has value (width-1,height-1).
*   To draw any object you will need to specify its location
* */
void SetCanvasSize(int width, int height)
```

Drawing Primitives

Once we have defined the canvas our next goal will be to draw the game board and its characters using basic drawing primitives. For drawing each object, we will need to specify its elementary points’ locations (x & y coordinates) in 2D canvas space and its size. You will only need lines, circles, curves (toruses), rounded rectangles and triangles as drawing primitives to draw the complete board, asteroids, and space ships.

For this purpose, skeleton code already includes functions for drawing lines, circles, curves, rounded rectangles (see below), triangles at specified location.

Recall that a line needs two vertices (points) whereas a triangle needs three vertices so to draw these primitives we will need to provide these vertices (points) locations along with the color of the shape.

Skeleton already provides a list of ≈ 140 colors which can be used for coloring different shapes – note that each color is combinations of three individual components red, green and blue.

Drawing Board

Initially it might seem drawing and managing the board is extremely difficult however you can overcome this difficulty using a very simple trick of divide and conquer. The trick revolves around the idea of the board being split into tiles. “Tile” or “cell” in this context refers to an 8 × 8 – you can use any tile size as you wish – pixel square on the screen. Asteroids screen resolution is 224 × 288, so this gives us a total board size of 28 × 36

```
  // Drawing functions provided in the skeleton code

/* To draw a triangle we need three vertices with each
 *    vertex having 2-coordinates [x, y] and a color for the
 *    triangle.
 *    This function takes 4 arguments first three arguments
 *    (3 vertices + 1 color) to draw the triangle with the
 *    given color.
 * */
void DrawTriangle(int x1, int y1, int x2, int y2, int x3,
                    int y3,float color[]/*three
                            component color vector*/)

// Function draws a circle of given radius and color at the
// given point location (sx,sy).
void DrawCircle(float x, float y, float radius,
                            float*color);

// Function draws a circular curve of given radius
void Torus2d(int x /*Starting position x*/,
             int y /*Starting position Y*/,
             float angle, // starting angle in degrees
             float length, // length of arc in degrees, >0
             float radius, // inner radius, >0
             float width,        // width of torus, >0
             unsigned int samples=60,// number of circle samples, >=3
             float *color = NULL);

// Function draws a line between point P1(x1,y1) and P2(x2,y2)
// of given width and colour
void DrawLine(int x1, int y1, int x2, int y2,
                   int lwidth = 3, float *color =NULL);

// Function draws a rectangle with rounded corners
at given x,y coordinates
void DrawRoundRect(float x, float y, float width,
                            float height,
                            float* color = 0,
                    float radius = 0.0/*corner radius*/);
// Function draws a string at given x,y coordinates
void DrawString(int x, int y, const string& str, float * color = NULL);
```

A set of functions for drawing primitive shapes.

tiles. Since we will be working independent of pixel units, so we can define tile size in our coordinates units. For instance we can divide the board in 8 × 8 units so drawing and managing the board will require these two steps:

1. Splitting the board in tiles.
2. Finding and storing what shape to draw in each tile.

Once we have divided the boards into tiles our job reduces to finding what lies in each tile i.e. what primitive shape we need to draw in each tile. We can record this information in an offline table and then can loop over this table to draw each primitive. We can further simplify our task by defining an enum environment to assign constant names (integers) to these primitives and then build table of these primitives. Following figure shows an example to draw some part of the board using a 2D offline table. Complete code can be found in the skeleton.

Following similar lines we have drawn the complete board. **Note that your system must follow object oriented design principles.**

Remember that you can do your drawing only in the GameDisplay() function, that is only those objects will be drawn on the canvas that are mentioned inside the GameDisplay function. This GameDisplay function is automatically called by the graphics library whenever the contents of the canvas (window) will need to be drawn i.e. when the window is initially opened, and likely when the window is raised above other windows and previously obscured areas are exposed, or when glutPostRedisplay() is explicitly called.

In short, GameDisplay function is called automatically by the library and all the things inside it are drawn. However whenever you need to redraw the canvas you can explicitly call the GameDisplay() function by calling the function glutPostRedisplay(). For instance, you will calling the GameDisplay function whenever you wanted to animate (move) your objects; where first you will set the new positions of your objects and then call the glutPostRedisplay() to redraw the objects at their new positions. Also see the documentation of Timer function.

```
// A simple example of board
enum BoardParts {
        NILL, // Prohibitive Empty space
        TLC, // Left corner top
        TRC, //Right corner top
        BLC, // Left corner bottom
        BRC, //Right corner bottom
        HL, // Horizontal line
        VL, // Vertical line
        PEBB, // Pebbles
};
const int BOARD_X = 10;
const int BOARD_Y = 5;
static int board_array[BOARD_Y][BOARD_X] = {
{ PEBB, PEBB, PEBB, BRC, BLC, PEBB, VL, VL, PEBB, PEBB},
{ PEBB, PEBB, PEBB, VL, VL, PEBB, PEBB, PEBB, PEBB, PEBB },
{ PEBB, PEBB, PEBB, VL, VL, PEBB, PEBB, PEBB, PEBB, PEBB },
{ BRC, HL, HL, TLC, TRC, HL, HL, HL, HL, BLC },
{ TRC, HL, HL, HL, HL, HL, HL, HL, HL, TLC } };
```

Figure: Example code for generating some section of the board.

Interaction With Game

For the interaction with your game you will be using arrow keys on your keyboard (you can use mouse and other keys as well). Each key on your keyboard has associated ASCII code. You will be making use of these ASCII codes to check which key is pressed and will take appropriate action corresponding to the pushed key. E.g. to move the Space Ship towards left you will check for the pressed key, if the pressed key is left arrow you will move the Space Ship left (rotate its face towards left). Keyboard keys are divided in two main groups: printable characters (such as a, b, tab, etc.) and non-printable ones (such as arrow keys, ctrl, etc.). Graphics library will call your corresponding registered functions whenever any printable and non-printable key from your keyboard is pressed. In the skeleton code we have registered two different functions (see below) to graphics library. These two functions are called whenever either a printable or non-printable ASCII key is pressed (see the skeleton for complete documentation). Your main task here is to add all the necessary functionality needed to make the game work.

```
/*This function is called (automatically by library)
* whenever any non-printable key (such as up-arrow,
* down-arraw) is pressed from the keyboard
*
* You will have to add the necessary code here
* when the arrow keys are pressed or any other key
* is pressed...
* This function has three argument variable key contains
* the ASCII of the key pressed, while x and y tells the
* program coordinates of mouse pointer when key was
* pressed.
* */
void NonPrintableKeys(int key, int x, int y)

/* This function is called (automatically by library)
```

```
* whenever any printable key (such as x,b, enter, etc.)
* is pressed from the keyboard
* This function has three argument variable key contains
* the ASCII of the key pressed, while x and y tells the
* program coordinates of mouse pointer when key was
*pressed.
* */
void PrintableKeys(unsigned char key, int x, int y)
```