CMPUT MM804

Assignment 1

Herb Yang

September 23, 2015

Due date: **October 22, 2015** before 00:00 midnight
Total marks: 50

Submit your assignment electronically to eClass. Organize your files properly into folders and zip up all the folders that correspond to a particular question into one single zip file for submission.

Important notes:

- Wikipedia: "Midnight marks the beginning and ending of each day in civil time throughout the world. In 24-hour time notation, "0:00" and "0:00:00" refer to midnight at the **start** of a given date." So, if you submit your assignment at 00:01 on Oct. 22, 2015, then your submission is counted as 1 day late.
- A late penalty of 10% per day applies to all late submissions. The maximum number of days late permitted is 2. After 2 days late, your assignment will be given a mark of **ZERO**.
- You are expected to complete the assignment on your own **without** collaboration with others. Discussions at the conceptual level but not at the coding level are permitted.
- Do not wait until the last minute to work on your assignment. Start as soon as possible. Debugging programs and understanding of the materials take time.
- There will be no help from me to solve these problems. However, questions regarding clarification will be answered.
- For the programming part,
  - Your code should have sufficient comments to make it readable.
  - Your code must work with Visual Studio Community 2015.
  - Include the solution file for each submitted program.
  - Each program must be in its own folder with its own solution file.
  - Zip up all the folders and submit only one zipped file
  - No executable is required to be submitted - only source files
  - Include a README file to document features or bugs, if any, of your program.

Marking scheme:

- Full assigned mark - no observable bug
- Half of the assigned mark - observable bugs
- 0 marks - does not work at all

1. (28 marks) The goal of this problem is to get you familiar with simple OpenGL programming. You are asked to complete the implementation of a variation of the simple pong game. When your program is complete and started, you should see something similar to the screenshot shown in Figure 1. In the figure, you can see a window with the title "Pong," and within this window there is a green rectangle, which defines the walls of the pong game. The screen size is defined by variables `screenSizeX` for width and `screenSizeY` for height. Note: The green box is slightly smaller than the screen area and it is specified by the wall is created. There are a total of 20 balls of random colour initially placed at specified location (see Figure 1a). These balls are initialized with random velocities. The screenshot of the program when it is successfully implemented and running is shown in Fig. 1b. The variable `score` is to keep track of the score of the player. When a ball hits the paddle, the player earns 1 point and when the ball hits the south wall, the player losses 1 point. In the skeleton code, the position of the ball is updated every 1ms, which is stored in `updateTime`.

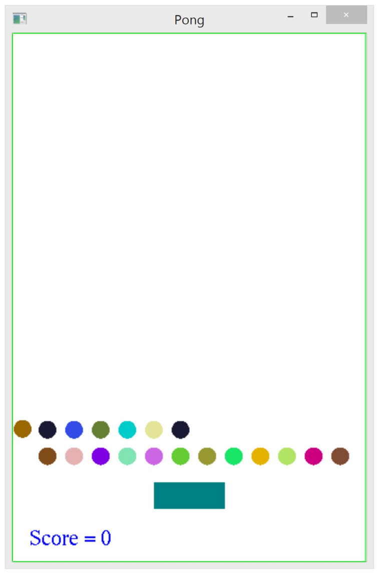
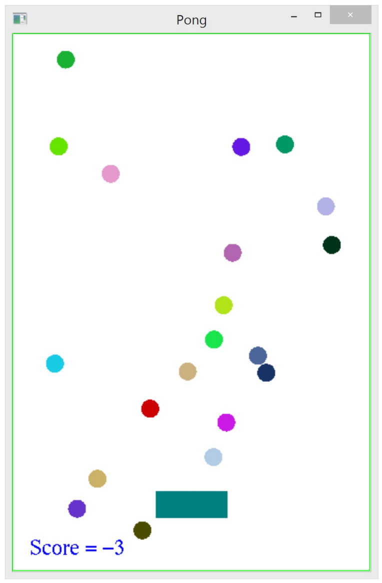


Figure 1a. Initial screenshot



1b. Screenshot when the program is running.

The location the ith ball with colour `r,g,b` is specified using

```
      bouncingBall[i].setLoc(x, y, radius, r, g, b);
```

where `x,y` are the coordinates of the centre of the ball, `radius` is the radius, the last three numbers specify the red, green and blue values of the colour of the ball. The ith ball's velocity is specified by:

```
      bouncingBall[i].setDir(incx, incy);
```

When the ball hits a wall or a side of the paddle, the direction of the ball needs to be adjusted accordingly. The paddle, which is of class `block`, is specified by:

```
    block paddle(screenSizeX / 2.0f, screenSizeY / 8.0f, 30.0f, 80.0f,
    0.0f, 0.5f, 0.5f, true);
```

The first two numbers are the coordinates of the centre of the paddle. The next two numbers are the height and width of the paddle. The red, green and blue values of the paddle are the next three numbers. The last value is a boolean value, which is either `true` or `false`. "`true`" means that the block is filled. This flag controls two things: 1) the inside of the paddle is a filled, and 2) the ball cannot get inside the paddle.

The wall, which is of class `box`, is specified by

```
        box wall(screenSizeY - 1.0f, 1.0f, 1.0f, screenSizeX - 1.0f,
        0.0f, 1.0f, 0.0f, false);
```

The first four numbers specify the top, bottom, left, and right boundaries of the wall. The next three numbers are the values of the red, green, and blue channels. The last one is of type `boolean`, which is to specify if the box is filled or not. In this case, it is specified as `false`, i.e. not filled. Similar to the case of the paddle, the value controls 1) not to fill the inside of the box, and 2) the ball can move around inside the box.

All the code that you need to fill in are in `ball.h`. The `update` routine in `main.cpp` calls all the required collision handlers. The pseudo code is given below. The above gives an

Algorithm 1 Update routine

```
for i = 0 to i< number of balls do
    update ball[i] location
    check for collision between ball[i] and each wall and update score if needed
    check for collision between ball[i] and the paddle and update score if needed
    check for collision of ball[i] with all the other balls
end for
```

overview of the whole game. At every timer event, when the balls' positions and the window are updated, then the effect generated is equivalent to moving the balls. This is how animation is created. The update routine is given below:

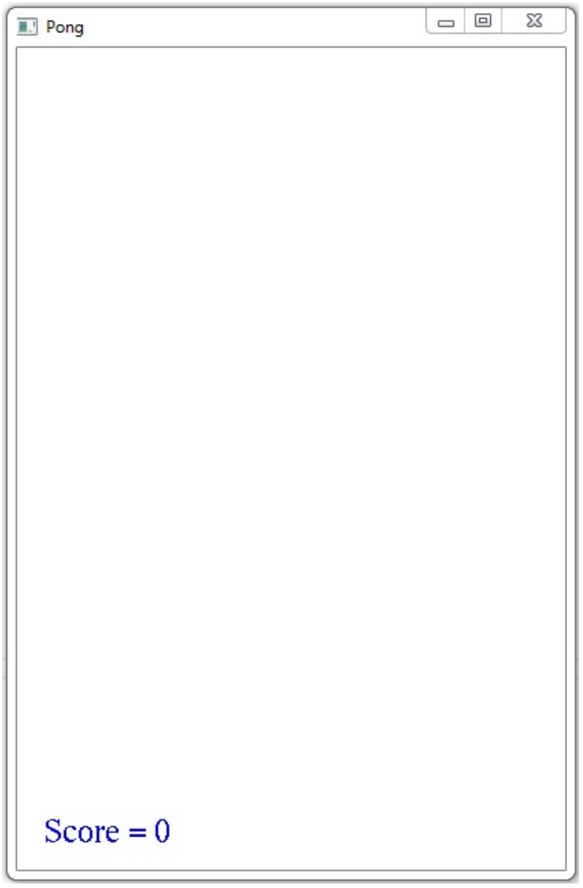
```
void update(int value)
{
        // add code below based on Algorithm 1



        // add code above
        sprintf_s(s, "%d", score); // convert to ASCII
        showText(100.0f, 20.0f, s, 0.0f, 0.0f, 1.0f);// display score
        glutPostRedisplay(); // update current window
        glutTimerFunc(updateTime, update, 0);
}
```

Note: The parameter `value` is not used. It is there to be consistent with `glutTimerFunc`.

In the following, the details of the required implementation are discussed.

(a) Download the zip file on eClass. Unzip it. In the folder demo, there is a demo program.

(b) Use `Visual Studio Community 2015` and open `bouncingBall.sln`. Note that some of files are not complete. When you build and run the executable, you will get something similar to what is shown in Figure 2. Note, if you use your home computer, check the properties of the project and set the path for the include files and the library files appropriately. Additionally, `freeglut.dll` must be in a folder that is in the `PATH` environment variable.



Score = 0

Figure 2. Screenshot when the downloaded files are compiled and run.

(c) (1 mark) In `box.h`, complete the code for `draw()`. The box is defined by its boundaries,

- ty - top boundary
- by - bottom boundary
- lx - left boundary
- rx - right boundary

and the colour of the box is specified by `cr` for red, `cg` for green, and `cb` for blue. When the program is compiled and run after completing this code, you should see a green box.

(d) The following are all in `ball.h`.

i. (2 marks) Complete the `draw()` function. Use a simple method to do this part. For example, any point on a circle can be specified by $(r \sin\theta, r\cos\theta)$, where r is the radius of the circle (see Figure 3). If you vary θ from $0°$ to $360°$, then you can compute all the points on the circle.

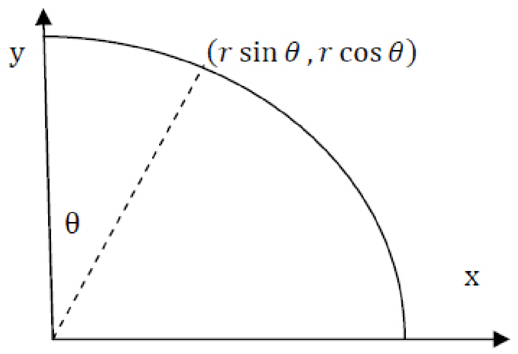


Figure 3. Generating a circle.

If your implementation is correct, you should see a red ball on the screen when you compile and run the program.

ii. (1 mark) The next function to complete is `move()`. At each timer event, the ball is moved by a distance specified by its velocity. The ball's last position should also be update. After this code is complete, the ball will when you run the program. But it will move beyond the wall.

iii. (2 marks) Complete the `update` function in `main.cpp` based on Algorithm 1.

iv. (4 marks) Complete the ball with wall collision handler. Once this is complete, the ball will be confined to move inside the wall. The collision of the ball with the wall can be simplified if you move the wall inward by the radius of the ball. Once you have done this, the collision of the ball with a wall is equivalent to the collision of point at the centre of the ball with the expanded wall (see Figure 4). This seemingly simple change will simplify subsequent implementation.

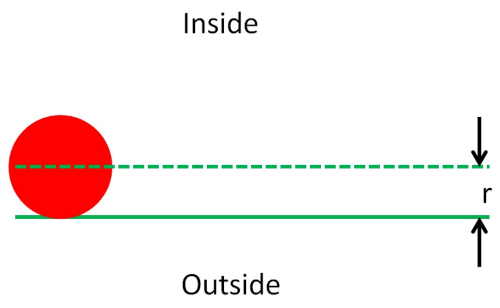


Figure 4. Moving the wall inward.

The code that you need to fill in is related to handling the response of the ball when it hits the wall. By the laws of physics, the velocity parallel to the wall is not changed, the velocity perpendicular to the wall is reversed but with the same magnitude as the one before collision (see Figure 5).

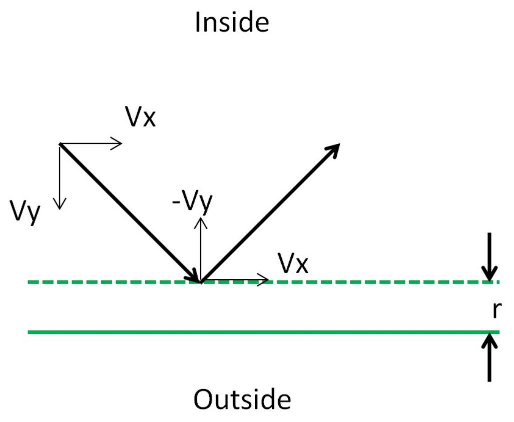


Figure 5. Collision response.

There are 4 cases to consider (2 marks for each case).

- The ball is initially above the bottom wall and is going down.
- The ball is initially below the top wall and is going up.
- The ball is initially to the left of the right wall and is going right.
- The ball is initially to the right of the left wall and is going left.

If your implementation is correct, the ball will be bounced off the wall.

v. (8 marks) The next part is to handle the response of collision of the ball with the paddle in `ball.h`. The idea is very similar to that of the wall. The difference is that the paddle is solid and the ball cannot go inside the paddle. Again, the analysis can be simplified by expanding the sides of the paddle outward and reducing the ball to a point.

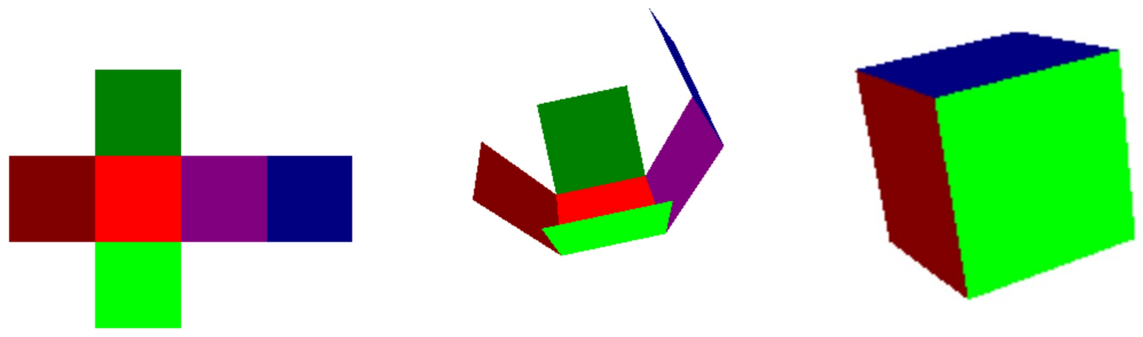
vi. (10 marks) The final part is to implement the ball-ball collision handler in `ball.h`. Denote $\mathbf{x}_1$ $\mathbf{x}_2$ as the centres of the two balls and $\mathbf{v}_1$ $\mathbf{v}_2$ their initial velocities before collision. Then their velocities after collision are given by:

$$\mathbf{v}_1' = \mathbf{v}_1 - \frac{(\mathbf{v}_1 - \mathbf{v}_2)\cdot(\mathbf{x}_1 - \mathbf{x}_2)}{(\mathbf{x}_1 - \mathbf{x}_2)\cdot(\mathbf{x}_1 - \mathbf{x}_2)}(\mathbf{x}_1 - \mathbf{x}_2)$$
$$\mathbf{v}_2' = \mathbf{v}_2 - \frac{(\mathbf{v}_1 - \mathbf{v}_2)\cdot(\mathbf{x}_1 - \mathbf{x}_2)}{(\mathbf{x}_1 - \mathbf{x}_2)\cdot(\mathbf{x}_1 - \mathbf{x}_2)}(\mathbf{x}_2 - \mathbf{x}_1)$$

where $\cdot$ denotes the dot product.

2. (22 marks) Scene graphs. In this problem, the goal is get you familiar with scene graphs and their applications.

   (a) (8 marks) Use scene graph to implement a program that shows the folding and unfolding of a cube. You may modify the code that comes with the text. When your program is started, it should show the 6 sides of the cube as shown in Fig. 6a. Fig. 6b shows a stage during folding and Fig. 6c shows the final result from a different view. You are free to use your own colours for the sides of the cube. Your program should also have the feature to unfold the cube into its original state.



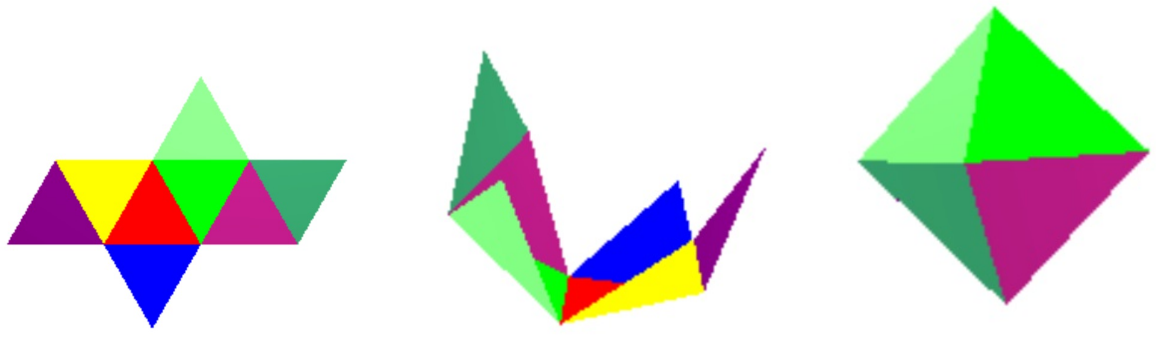
a) An unfolded cube. b) During folding from a different view. c) Final cube.

Figure 6

You are required to implement the following keyboard commands:

- ‘f’ or ‘F’ - to fold the cube
- ‘u’ or ‘U’ - to unfold the cube
- ‘a’ - to translate the cube in the negative x-axis by 1 unit
- ‘A’ - to translate the cube in the positive x-axis by 1 unit
- ‘b’ - to translate the cube in the negative y-axis by 1 unit
- ‘B’ - to translate the cube in the positive y-axis by 1 unit
- ‘c’ - to translate the cube in the negative z-axis by 1 unit
- ‘C’ - to translate the cube in the positive z-axis by 1 unit
- ‘x’ - to rotate with respect to the x-axis by -1 degree
- ‘X’ - to rotate with respect to the x-axis by +1 degree
- ‘y’ - to rotate with respect to the y-axis by -1 degree
- ‘Y’ - to rotate with respect to the y-axis by +1 degree
- ‘z’ - to rotate with respect to the z-axis by -1 degree
- ‘Z’ - to rotate with respect to the z-axis by +1 degree

(b) (14 marks) Use scene graph to implement a program that shows the folding and unfolding of a octahedron. You may modify the code that comes with the text. When your program is started, it should show the 8 sides of the octahedron as shown in Fig. 7a. Fig. 7b shows a stage during folding and Fig. 7c shows the final octahedron from a different view. You are free to use your own colours for the sides of the octahedron. Your program should also have the feature to unfold the octahedron into its original state.



a) An unfolded octahedron. b) During folding from a different view. c) Final octahedron.

Figure 7

You are required to implement the following keyboard commands:

- ‘f’ or ‘F’ - to fold the octahedron
- ‘u’ or ‘U’ - to unfold the octahedron
- ‘a’ - to translate the octahedron in the negative x-axis by 1 unit
- ‘A’ - to translate the octahedron in the positive x-axis by 1 unit
- ‘b’ - to translate the octahedron in the negative y-axis by 1 unit
- ‘B’ - to translate the octahedron in the positive y-axis by 1 unit
- ‘c’ - to translate the octahedron in the negative z-axis by 1 unit
- ‘C’ - to translate the octahedron in the positive z-axis by 1 unit
- ‘x’ - to rotate with respect to the x-axis by -1 degree
- ‘X’ - to rotate with respect to the x-axis by +1 degree
- ‘y’ - to rotate with respect to the y-axis by -1 degree
- ‘Y’ - to rotate with respect to the y-axis by +1 degree
- ‘z’ - to rotate with respect to the z-axis by -1 degree
- ‘Z’ - to rotate with respect to the z-axis by +1 degree