

# Assignment on OpenGL

---

There are three tasks in this assignment.

1. Fully Controllable Camera (Problem 1.exe)
2. Gun (Problem 1.exe)
3. Bubbles (Problem 2.exe)

## Fully Controllable Camera (Problem 1.exe)

Up arrow - move forward

Down arrow - move backward

Right arrow - move right


Left arrow - move left

PgUp - move up

PgDn - move down

1 - rotate/look left

2 - rotate/look right

3 - look up 

4 - look down

5 - tilt clockwise

6 - tilt counterclockwise

### Hint:

Maintain four global variables: one 3d point *pos* to indicate the position of the camera and three 3d **unit** vectors *u*, *r*, and *l* to indicate the up, right, and look directions respectively. *u*, *r*, and *l* must be perpendicular to each other, i.e.,

$$u \cdot r = r \cdot l = l \cdot u = 0, \quad u = r \times l, \quad l = u \times r, \quad r = l \times u.$$

You should initialize and maintain the values of  $u$ ,  $r$ , and  $l$  such that the above property holds throughout the run of the program. For example, you can initialize them as follows:

$$u = (0, 0, 1), \quad r = \left(-\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, 0\right), \quad l = \left(-\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}, 0\right), \\ pos = (100, 100, 0)$$

And while changing  $u$ ,  $r$ , and  $l$ , make sure that they remain unit vectors perpendicular to each other.

The first 6 operations listed above are move operations, where the position of the camera changes but the up, right, and look directions do not. The last 6 operations are rotate operations, where the camera position does not change, but the direction vectors do.

In case of a move operation, move  $pos$  a certain amount along the appropriate direction, but leave the direction vectors unchanged. For example, in the move right operation, move  $pos$  along  $r$  by 2 (or by any amount you find appropriate) units. In case of a rotate operation, rotate two appropriate direction vectors a certain amount around the other direction vector, but leave the position of the camera unchanged. For example, in the look up operation, rotate  $l$  and  $u$  counterclockwise with respect to  $r$  by 3 (or by any amount you find appropriate) degrees.

If you maintain  $pos$ ,  $u$ ,  $r$ , and  $l$  in this way, your `gluLookAt` statement will look as follows:

```
gluLookAt(pos.x, pos.y, pos.z, pos.x + l.x, pos.y + l.y, pos.z + l.z, u.x, u.y, u.z);
```

## Gun (Problem 1.exe)

Color pattern of the gun should be same as the one modeled in Problem 1.exe. You can't use any OpenGL library function to draw the parts of the gun.

Press the keys **q**, **w**, **e**, **r**, **a**, **s**, **d** and **f** to find out how the gun rotates. Also observe that after certain amount, each joint ceases to rotate.

Left click the mouse to fire the gun.

Draw the gunshots (red squares) slightly in front of the wall to avoid glitches. Be careful to ensure that your model is located well within the far distance (assigned in gluPerspective) from the camera. See [this](#) to find how to compute the intersection of a line with a plane.

Right click the mouse to toggle viewing the axis.

## Bubbles (Problem 2.exe)

The camera is fixed looking at the center of the green boundary square from a perpendicular position. There is red circle inside the square.

After the program starts, five bubbles will pop up **one by one** from the left bottom region of the square and start moving towards a **random direction**. None of the bubbles will go out of the boundary square. Rather, they will be reflected upon colliding with the boundary of the square. Note that, the bubbles that haven't yet gone inside the red circle will cross one another.

When a bubble goes *completely* inside the red circle, it will never go out of the circle. Rather, it will reflect upon colliding with the circle. It will also reflect with other bubbles that are *completely* inside the red circle. Note that, the bubble will not reflect with bubbles that are outside the red circle even partially.

The program will also support the following functionalities.

Up arrow – increase the speed of the bubbles

Down arrow – decrease the speed of the bubbles

Press 'p' – resume/pause the movement of the bubbles

There should be an upper and lower bound for the speed of the bubbles so that the speed can't be increased to  $\infty$  and decreased to 0.

The length of the square and the radius of the circle and bubbles are not strictly specified. Adjust the values in order to reproduce the given figure as close as possible. The bubbles have the same velocity.

## Hint:

Maintain two global variables for each bubble to indicate its position and speed.

In the idle function, update the position of the bubbles by adding to them the respective speed vectors. Check whether a bubble intersects the boundary square. If yes, reflect its direction vector on the boundary square. You may want to choose the sides of the square wisely so that reflection is easier. For example, reflection against *x axis* in a 2d plane can be done by simply reversing the *y* component of the velocity of the bubble.

Check whether a bubble comes completely inside the circle. It can be done by calculating the distance between the centers of the bubble and the red circle. The bubble is inside the circle if the distance is  $\leq$  radius of the circle – radius of the bubble.

You have to reflect bubbles with respect to one another and also with respect to the red circle. You can follow [this](#) to find the formula. Please try to understand anything before applying in the assignment.

## Submission Guideline

Create a folder named by your roll. If your roll is 1605130, then the name of the folder will be 1605130. There will be two cpp files inside the folder. Solution of Problem 1 will be named as 1605130\_1.cpp. Solution of Problem 2 will be named as 1605130\_2.cpp. Now, zip the folder 1605130 and upload the zipped folder 1605130.zip in moodle within **20<sup>th</sup> March on 11:00 PM**.