Rasmus Fredrikson

DH2323 LAB 2

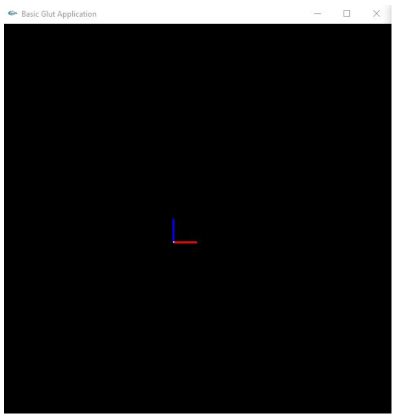
1. Vector Class and Demonstrator

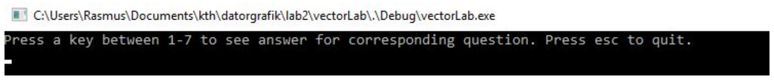
1.2 Task 1a: Complete the Vector Class

I used the standard formulas for the different calculations, found on the wikipedia page about vectors. I had to make sure I had the C++ pointers correct (which gave me some problems at first).

1.3 Task 1b: Build a Demonstrator Program

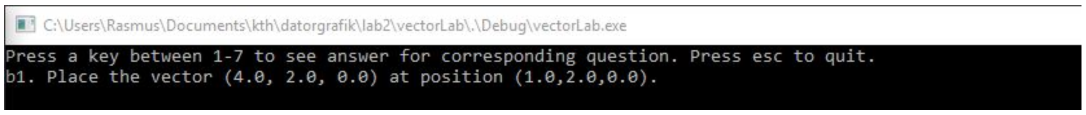
The first thing to appear on the screen is seen in the picture below to the left. The picture to the right shows what is printed in the terminal. Each answer can be seen by simply pressing corresponding number 1-7.



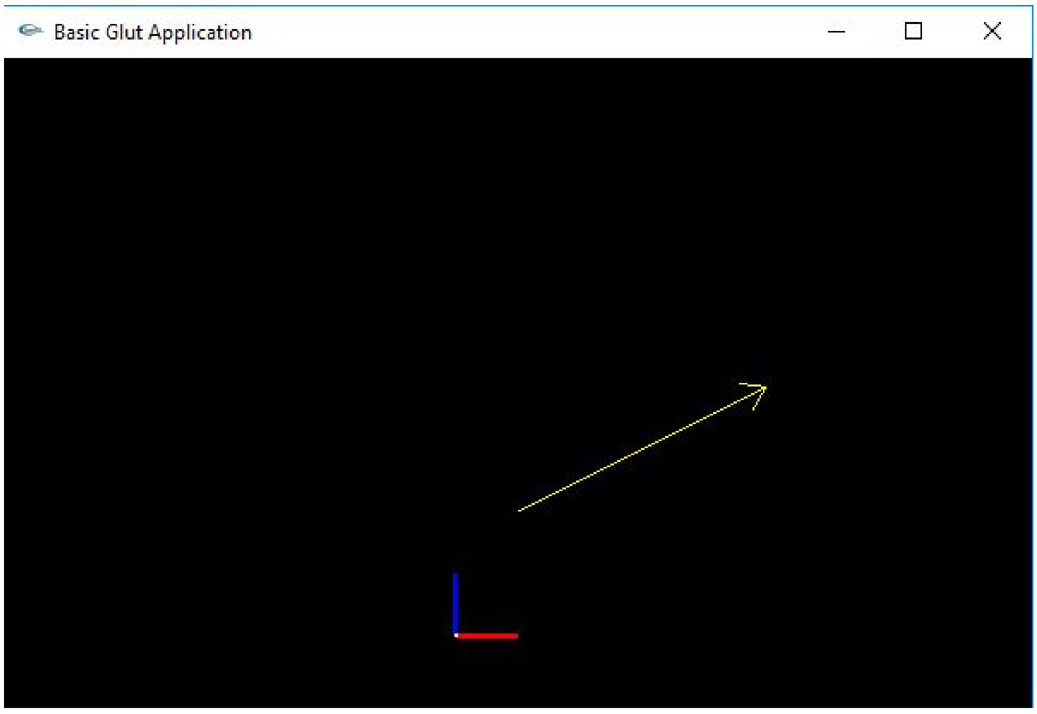


```
C:\Users\Rasmus\Documents\kth\datorgrafik\lab2\vectorLab\.\Debug\vectorLab.exe
Press a key between 1-7 to see answer for corresponding question. Press esc to quit.
```

b1.

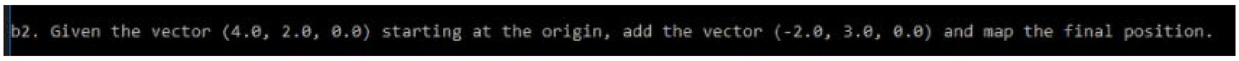


```
C:\Users\Rasmus\Documents\kth\datorgrafik\lab2\vectorLab\.\Debug\vectorLab.exe
Press a key between 1-7 to see answer for corresponding question. Press esc to quit.
b1. Place the vector (4.0, 2.0, 0.0) at position (1.0,2.0,0.0).
```

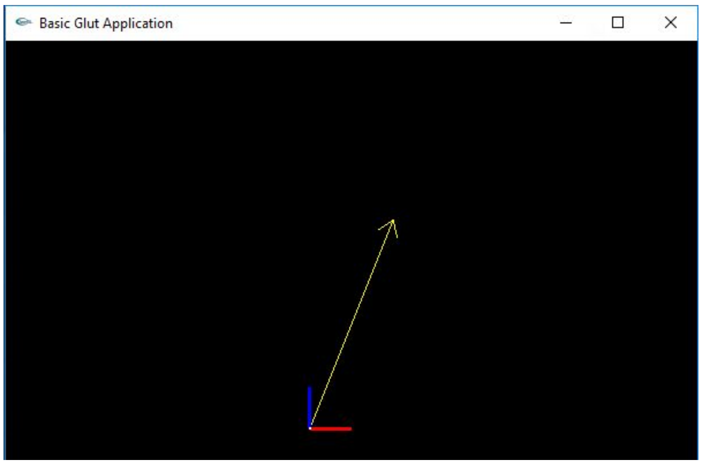


This was simply done by drawing the vector with (1.0,2.0,0.0) as starting point.

b2.

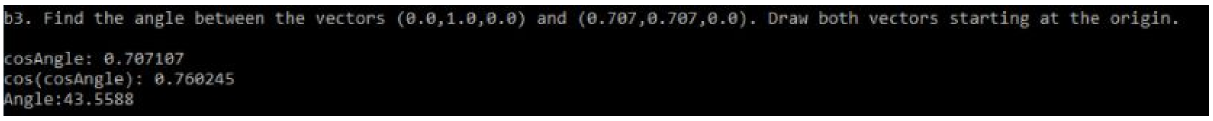


```
b2. Given the vector (4.0, 2.0, 0.0) starting at the origin, add the vector (-2.0, 3.0, 0.0) and map the final position.
```



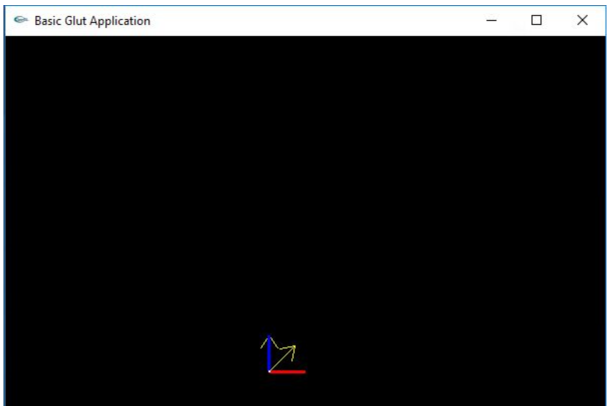
This was done by adding the two vectors with help of the addTo function.

b3.



```
b3. Find the angle between the vectors (0.0,1.0,0.0) and (0.707,0.707,0.0). Draw both vectors starting at the origin.

cosAngle: 0.707107
cos(cosAngle): 0.760245
Angle:43.5588
```

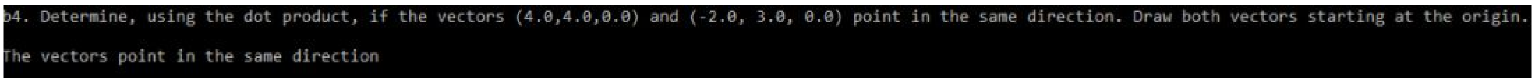


This was done by using the cosine formula.

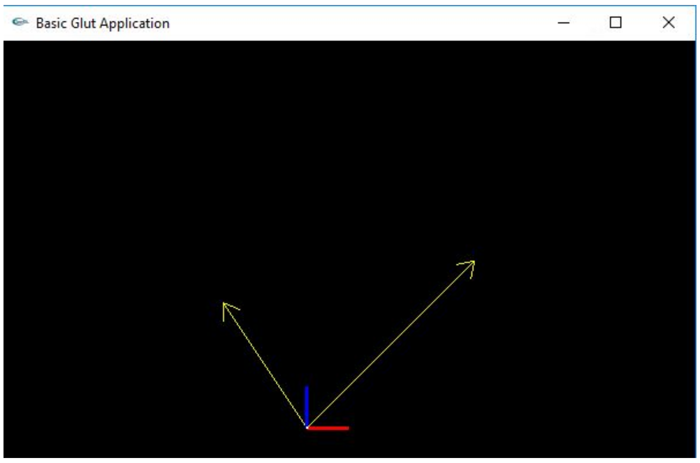
$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|_2 \|\mathbf{B}\|_2} :$$

```
v1.getDotProduct(v2)/(v1.getMagnitude()*v2.getMagnitude());
```

b4.



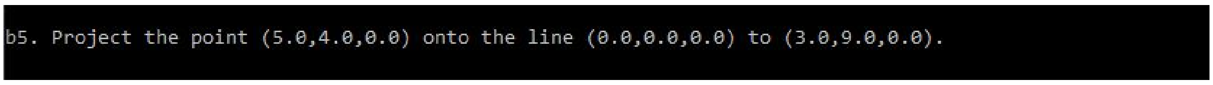
```
b4. Determine, using the dot product, if the vectors (4.0,4.0,0.0) and (-2.0, 3.0, 0.0) point in the same direction. Draw both vectors starting at the origin.
The vectors point in the same direction
```



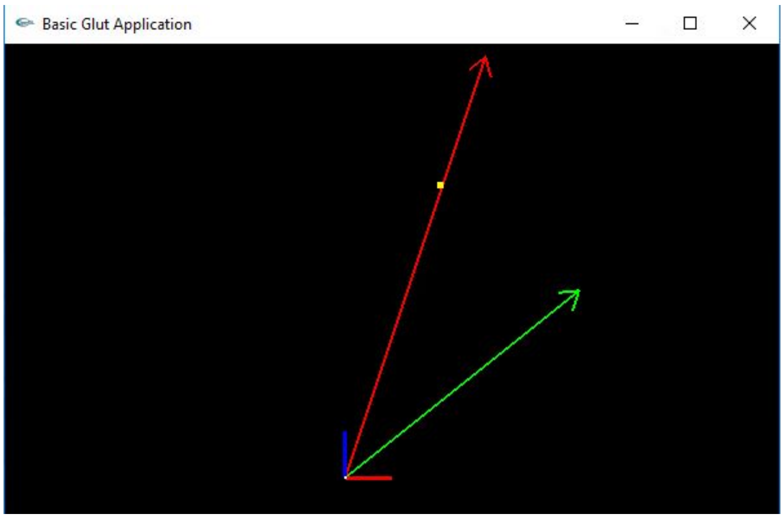
If the dot product is greater than 0, they point in same direction. If it's equal to 0 the vectors are orthogonal.

b5.

A problem I had here was that the point is drawn slightly to the left of the line and I couldn't figure out why.



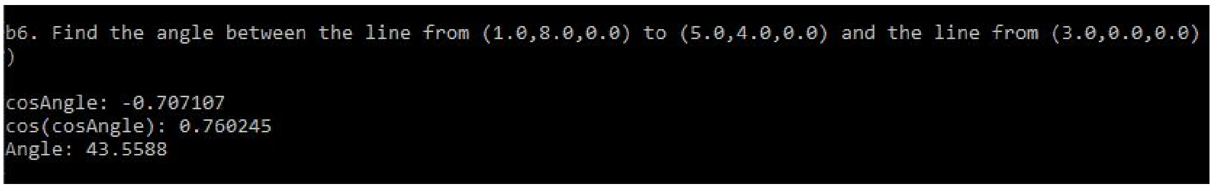
```
b5. Project the point (5.0,4.0,0.0) onto the line (0.0,0.0,0.0) to (3.0,9.0,0.0).
```



I solved this by projecting the green vector onto the red vector.

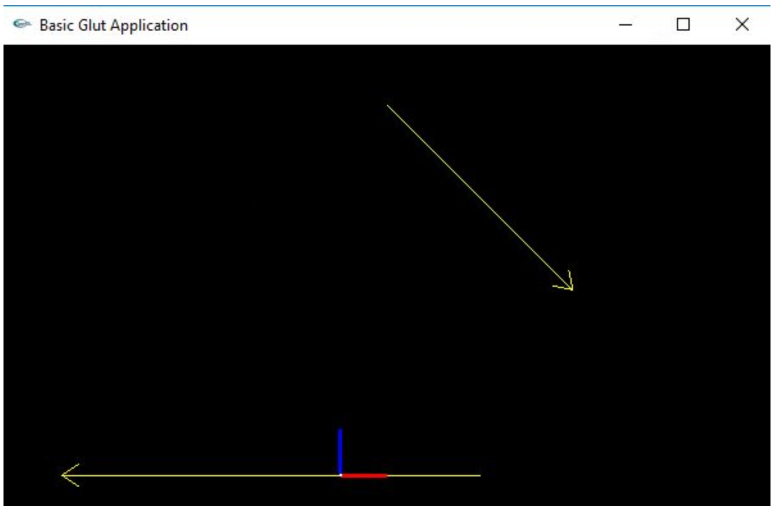
```
v1.setMagnitude(v2.getMagnitude());
```

b6.



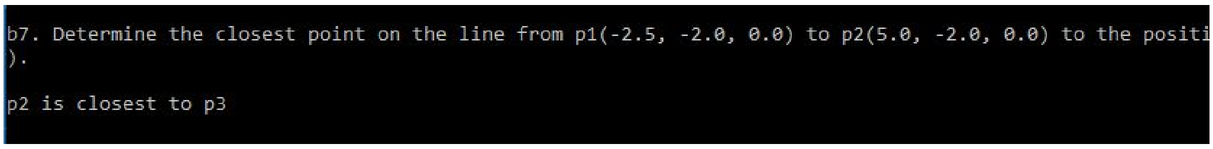
```
b6. Find the angle between the line from (1.0,8.0,0.0) to (5.0,4.0,0.0) and the line from (3.0,0.0,0.0)
)

cosAngle: -0.707107
cos(cosAngle): 0.760245
Angle: 43.5588
```



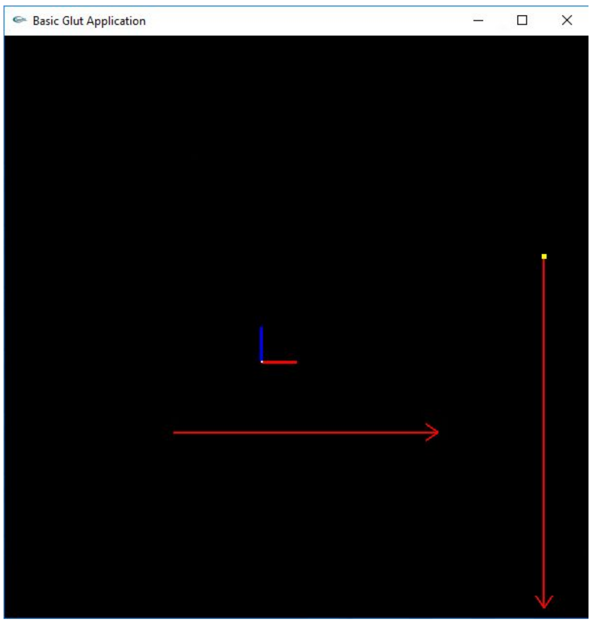
This was once again done using the cosine formula seen in b3.

b7.



```
b7. Determine the closest point on the line from p1(-2.5, -2.0, 0.0) to p2(5.0, -2.0, 0.0) to the positi
).

p2 is closest to p3
```



The closest point is the one orthogonal to the point, since the normal vector doesn't intersect the vector the closest point is one of the two endpoints, in this case the right one. This one turned out to be quite difficult to implement due to the orthogonal vector not intersecting the line, in the end I decided to just check the endpoints and see which one was the closest.

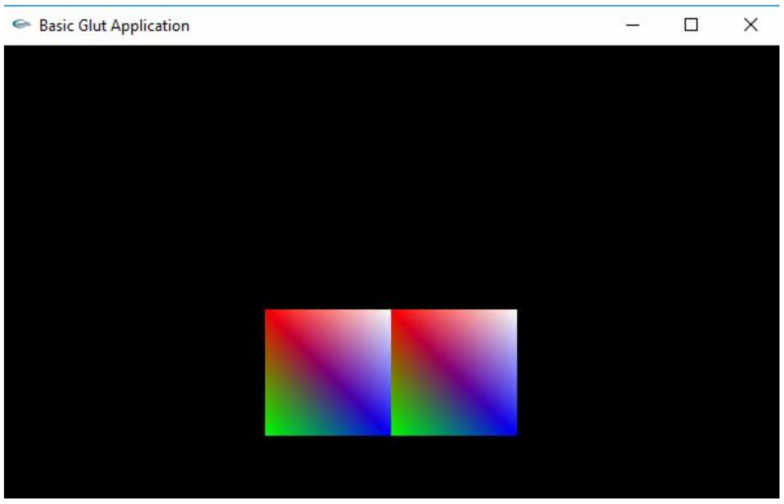
2. Transformations and Matrices

2.2 Tasks

The answer to 2.2 and 2.3 can be seen by pressing key 1-3. Pressing r will rotate the squares and pressing v will let them rotate along their axis.

1.

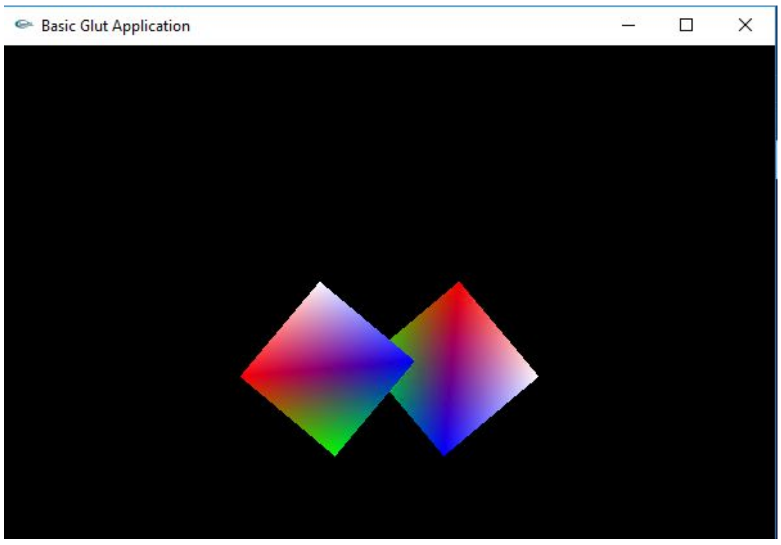
Display two squares on the screen, centered at coordinates (1.0,1.0,- 5.0) and (-1.0,1.0,-5.0).



This was simply done by translating the squares to does coordinates.

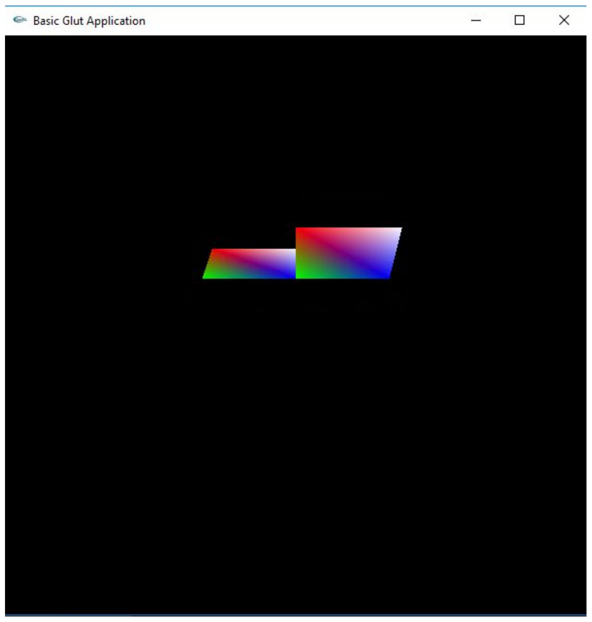
2.

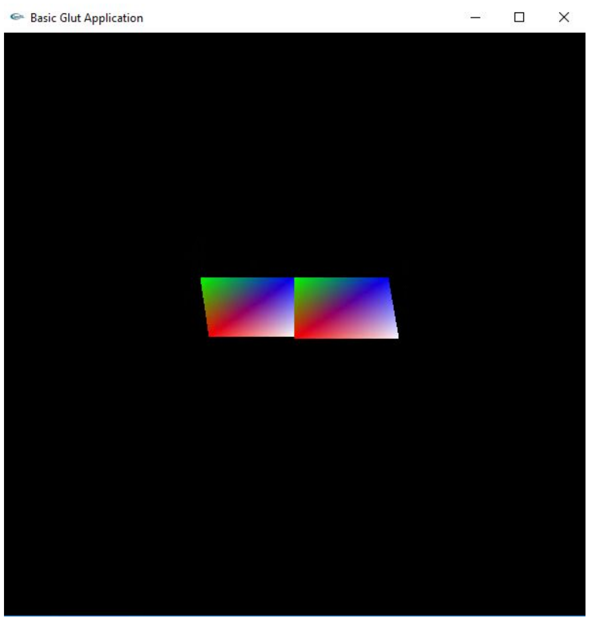
Make both squares rotate around their own z-axes by a certain amount (e.g. 5 degrees) each time the `r' key is pressed.

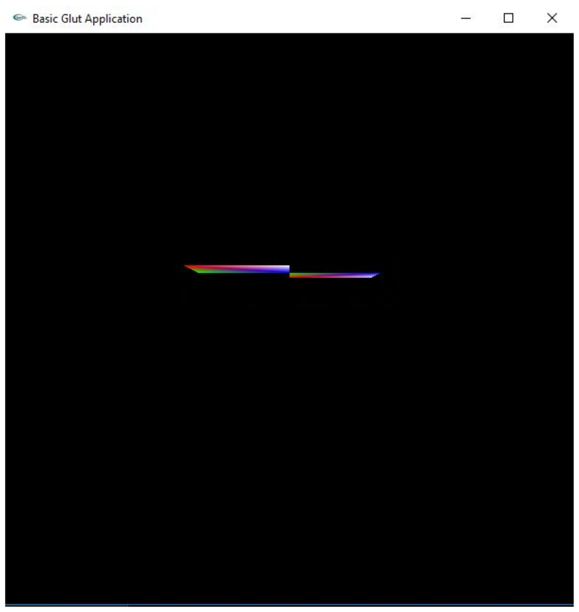


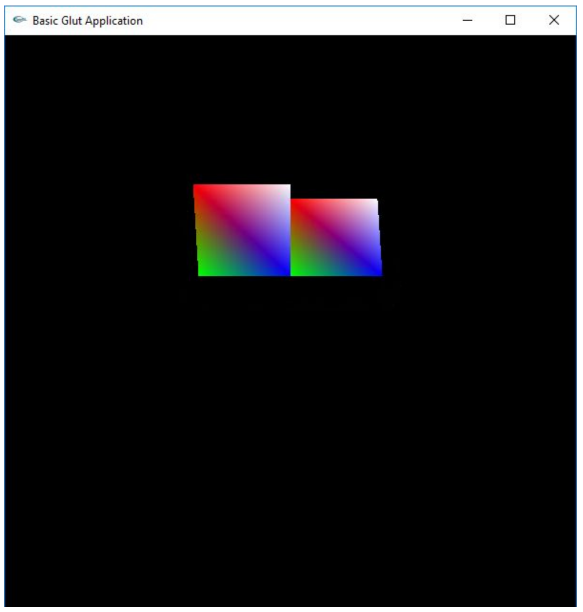
This was done by calling the rotatef function each time r was pressed

3.
Rotate a square around one of its vertices (rather than its center).









Had some trouble making them rotate around their vertices so instead let them rotate along their axis.

5.
Same image as in 1. Managed to do this by creating a rotation matrix which was then multiplied with the currently loaded matrix.

3 Quaternions

3.2 Tasks

1.

I used the standard formulas for the different calculations, found on the wikipedia page about quaternions. This was very hard to find and understand and took some time to fix, the second part of the quaternion task went really quick when all of the functions were in place.

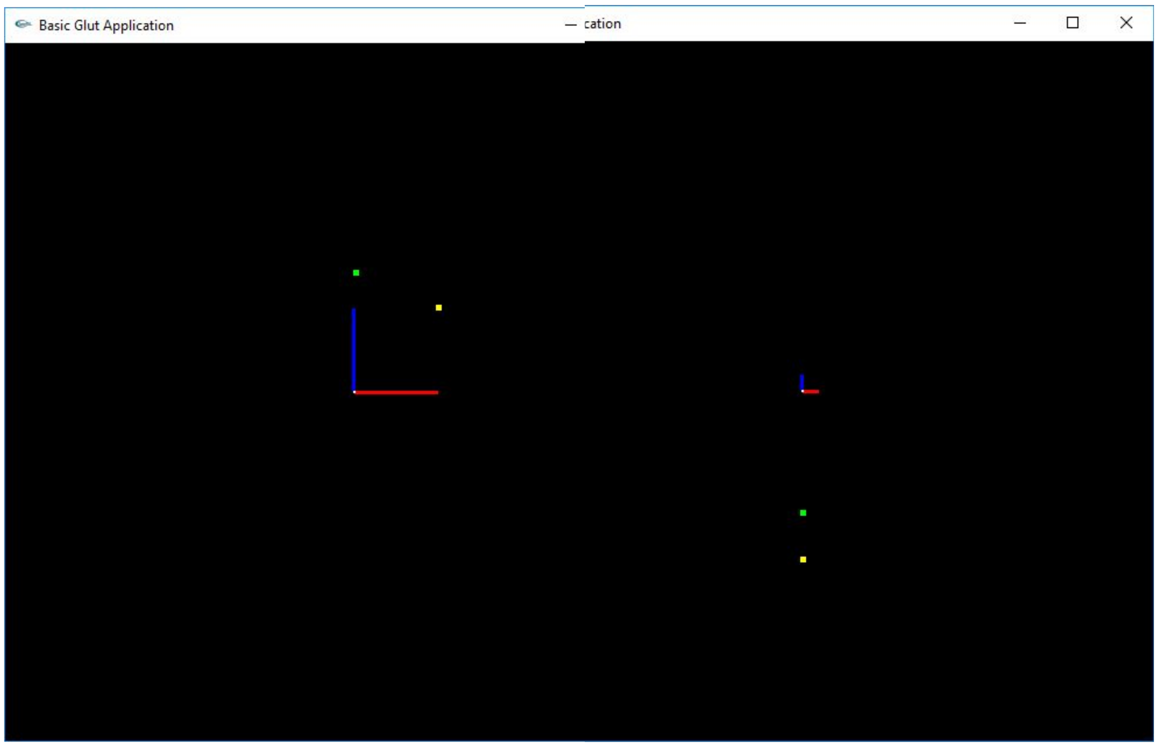
2 & 3.

2. Use the quaternion class to rotate the point (1.0,1.0,0.0) 45 degrees around the axis (0.0,0.0,1.0).

3. Use the quaternion class to rotate the point (0.0,-10.0,0.0) 45 degrees around the axis defined by the vector (10.0,0.0,0.0).

By following the formula shown in the lab spec and using the implemented functions this turned out to be quite easy. The yellow dot is the start point and the green one is the rotated Quaternion.

Task 2 to the left and task 3 to the right. Only difference between the two calculations was that the axis in 3 had to be normalized.

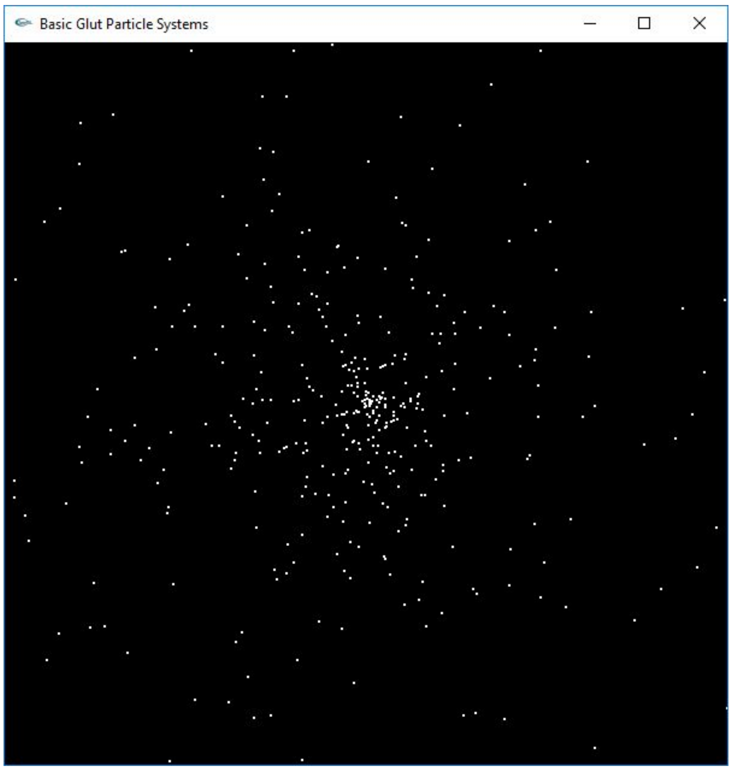


4 Particle System

4.2 Tasks

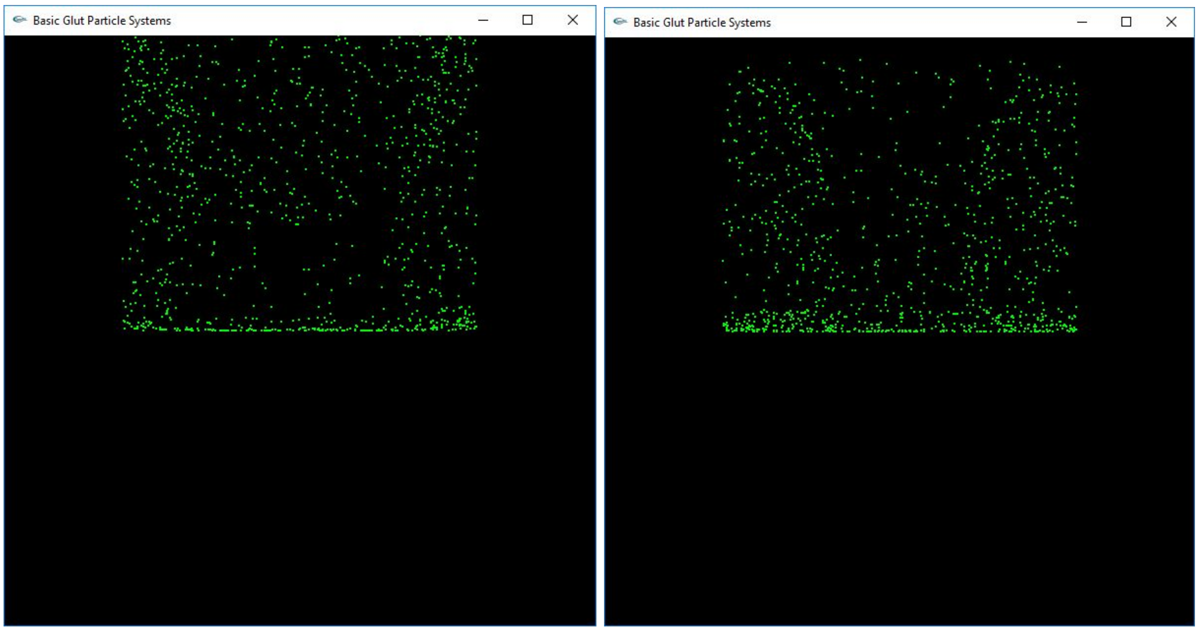
1.

To make the scene animated, the suggested code was added.



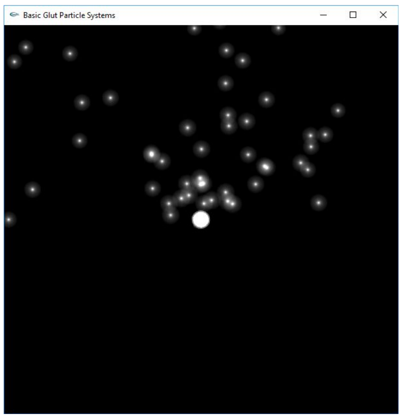
2.

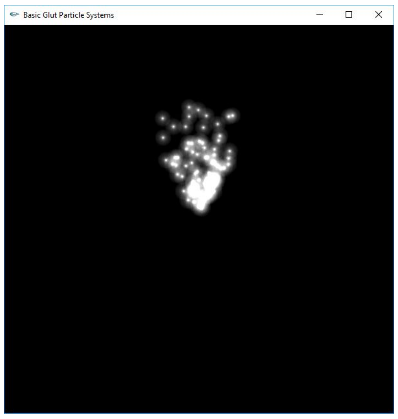
The newly defined vector was set to have green color, 1000 particles with 100 released every 0,05sec. Some gravity, velocity and wind was also added.

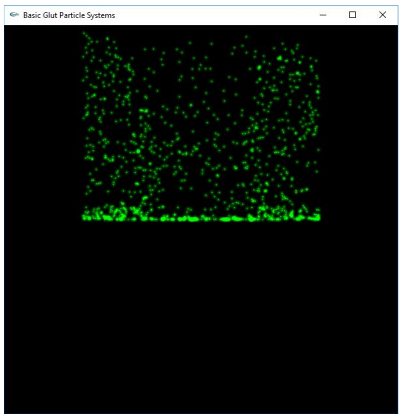


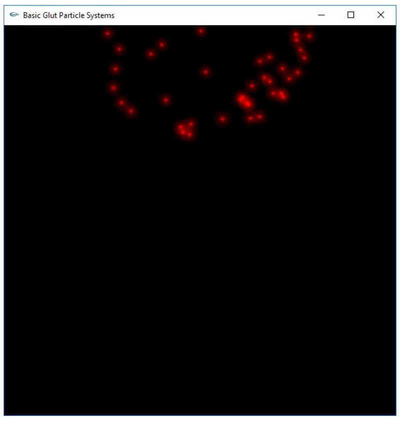
3 & 4.

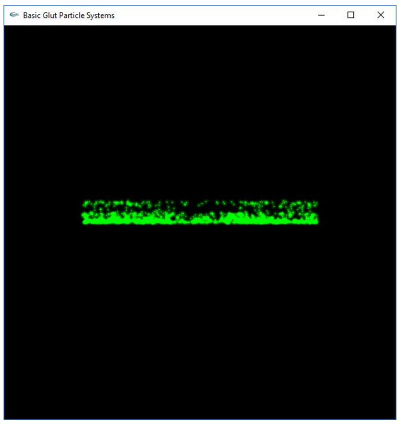
Lastly collision and blending was implemented.

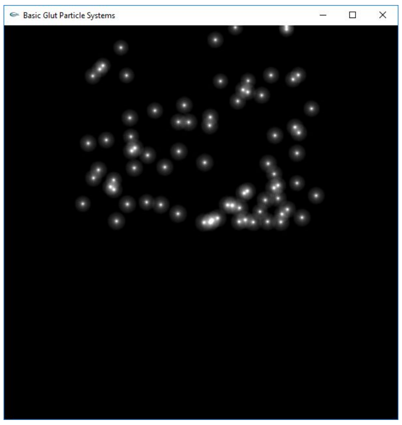

Basic Glut Particle Systems

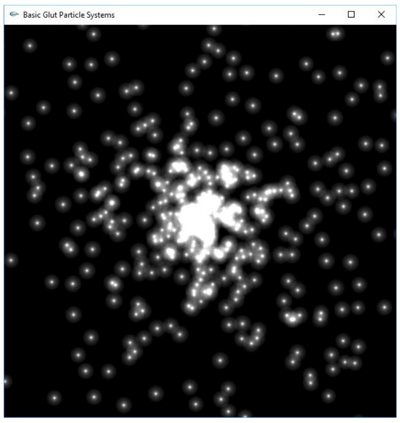

Basic Glut Particle Systems

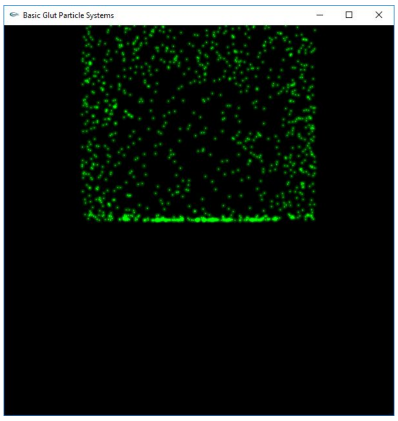

Basic Glut Particle Systems

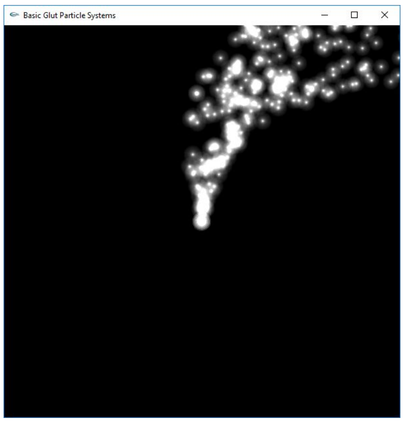

Basic Glut Particle Systems


Basic Glut Particle Systems


Basic Glut Particle Systems


Basic Glut Particle Systems


Basic Glut Particle Systems


Basic Glut Particle Systems