Tutorial 3 – Particle Class

Last week you started developing an OO framework for a simple program simulating a motion of a 2D particle inside a box. This week you are going to expand our program in two ways:

1.) Adding separate particle class
2.) Creating multiple particle simulation
3.) Experimenting with different particle trajectories

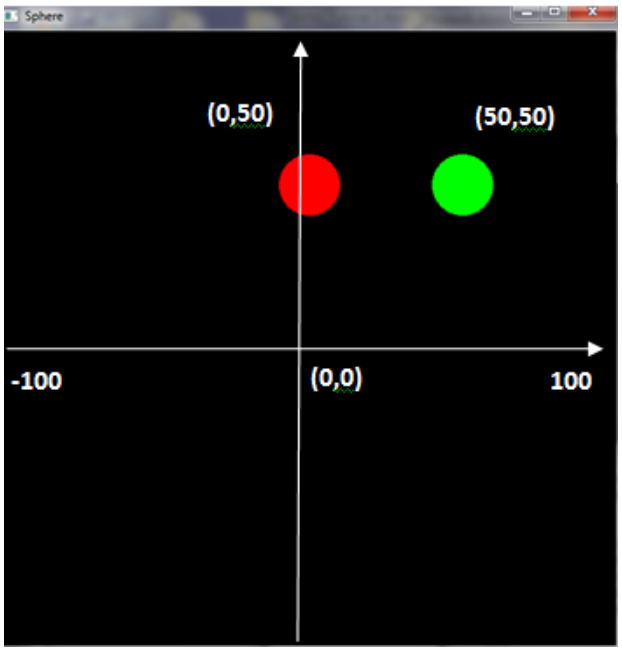
As discussed in the previous tutorial in order to move the particle in the xy-plane the modelview matrix was multiplied by a matrix that describes a translation before drawing the scene. OpenGL provides a high-level function that performs this task for you glTranslatef(GLfloat x, GLfloat y, GLfloat z); This function takes as parameters the amount to translate along the x, y, and z directions. It then constructs an appropriate matrix and multiplies it onto the current matrix stack. However, the effect of glTranslatef are cumulative. Each time you call glTranslatef, the appropriate matrix is constructed and multiplied by the current modelview matrix. The new matrix then becomes the current modelview matrix, which is then multiplied by the next transformation, and so on.

Exercise 1

Using the application framework developed in the previous tutorial, modify the display method (listing 1) Compile and execute the code. Confirm the cumulative effect of the glTranslatef.

```
void SphereDemo::display(void)
{
Application::display();

        // Go 50 units up the y-axis
        glTranslatef(0.0f, 50.0f, 0.0f);

        // Draw the first sphere
        glColor3ub(255, 0, 0);
        glutSolidSphere(10.0f,30,30);

        // Go 50 units out the x-axis
        glTranslatef(50.0f, 0.0f, 0.0f);

        // Draw the second sphere
        glColor3ub(0, 255, 0);
        glutSolidSphere(10.0f,30,30);
}
```

Listing 1



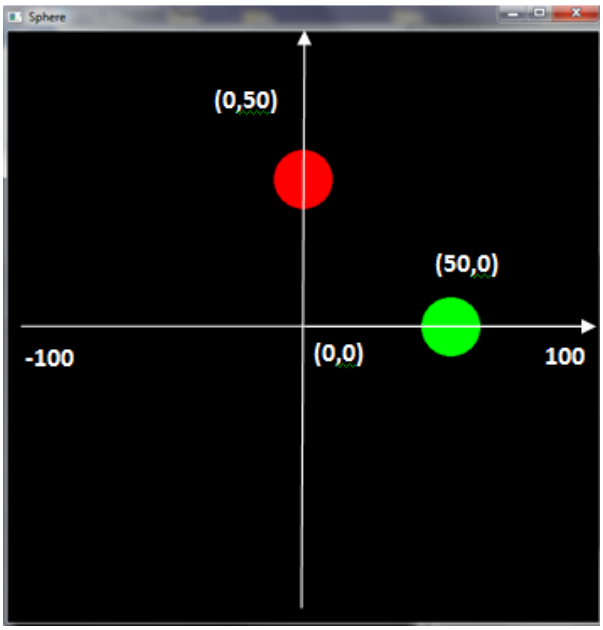
You can make an extra call to glTranslate to back down the y-axis 10 units in the negative direction, but this makes some complex scenes difficult to code and debug—not to mention that you throw extra transformation math at the CPU or GPU. A simpler method is to reset the modelview matrix to a known state—in this case, cantered at the origin of the eye coordinate system. You reset the origin by loading the modelview matrix with the identity matrix. The identity matrix specifies that no transformation is to occur, in effect saying that all the coordinates you specify when drawing are in eye coordinates. An identity matrix contains all 0s, with the exception of a diagonal row of 1s. When this matrix is multiplied by any vertex matrix, the result is that the vertex matrix is unchanged.

Exercise 2

Make further changes (shown in listing 2) to the application from the exercise 1. Compile and execute the code. Confirm the correct behaviour of the program.

```
void SphereDemo::display(void)
{
Application::display();

        glLoadIdentity();
        // Go 50 units up the y-axis
        glTranslatef(0.0f, 50.0f, 0.0f);
        // Draw the first sphere
        glColor3ub(255, 0, 0);
        glutSolidSphere(10.0f,30,30);

        glLoadIdentity();
        // Go 50 units out the x-axis
        glTranslatef(50.0f, 0.0f, 0.0f);
        // Draw the second sphere
        glColor3ub(0, 255, 0);
        glutSolidSphere(10.0f,30,30);
}
```

Listing 2



The Matrix Stacks

Resetting the modelview matrix to identity before placing every object is not always desirable. Often, you want to save the current transformation state and then restore it after some objects have been placed. This approach is most convenient when you have initially transformed the modelview matrix as your viewing transformation (and thus are no longer located at the origin). To facilitate this procedure, OpenGL maintains a matrix stack for both the modelview and projection matrices. A matrix stack works just like an ordinary program stack. You can push the current matrix onto the stack with glPushMatrix to save it and then make your changes to the current matrix. Popping the matrix off the stack with glPopMatrix then restores it. Figure 1 shows the stack principle in action.

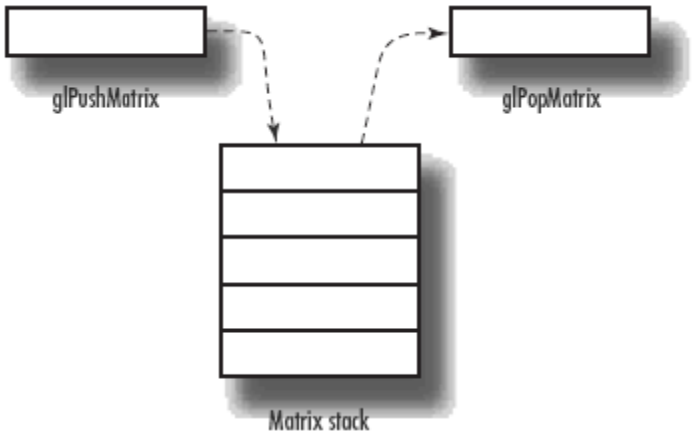


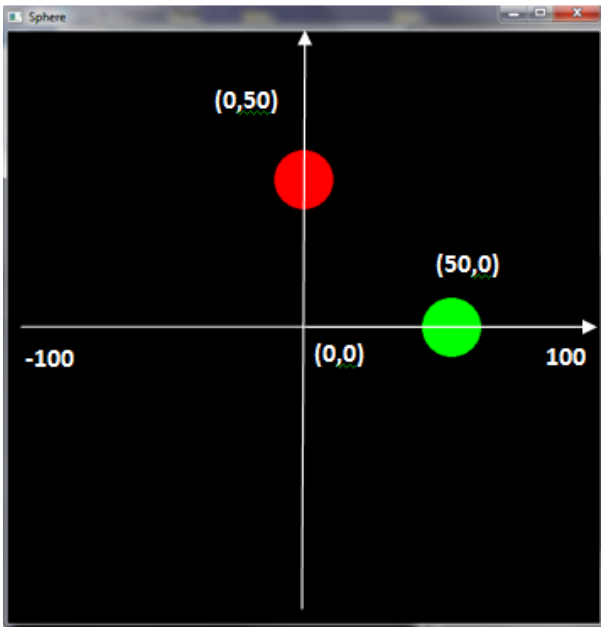
Figure 1

Exercise 3

Make further changes (highlighted in listing 3) to the application from the exercise 1 and 2. Compile and execute the code. Confirm the correct behaviour of the application.

```
void SphereDemo::display(void)
{
Application::display();

        glLoadIdentity();

        glPushMatrix();
        // Go 50 units up the y-axis
        glTranslatef(0.0f, 50.0f, 0.0f);
        // Draw the first sphere
        glColor3ub(255, 0, 0);
        glutSolidSphere(10.0f,30,30);
        glPopMatrix();

        glPushMatrix();
        // Go 50 units out the x-axis
        glTranslatef(50.0f, 0.0f, 0.0f);
        // Draw the second sphere
        glColor3ub(0, 255, 0);
        glutSolidSphere(10.0f,30,30);
        glPopMatrix();
}
```

Listing 3



Exercise 4:

Redesign your application simulating a motion of a 2D particle inside a box developed in the previous exercises.

1. Expand your class structure by adding to you application a separate particle class. Create separate files for particle class definition and implementation as shown in Figure 2. Download from the Blackboard coreMath.h containing a readymade Vector2 data structure and utility functions. For more information regarding vector algebra see:
   http://emweb.unl.edu/math/mathweb/vectors/vectors.html

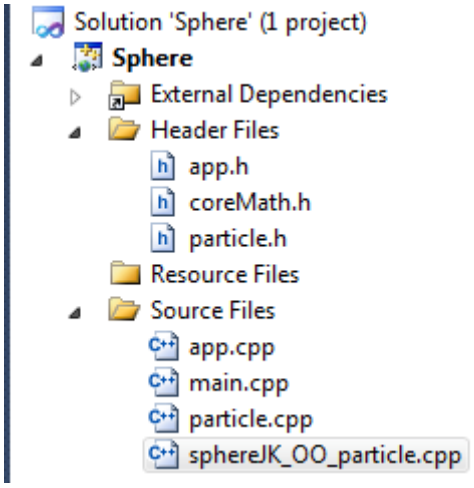


Figure 2

2. Implement all the methods and attributes of the particle class shown in the class diagram in Figure 3, and Listing 4. Use lecture slides for guidance.

Modify the initial particle velocity 1) V = [300, 300]; 2) V = [-300, 300];
Do you observe that the particle is sometimes getting stuck to one edge of the box?
What is the reason for this behaviour? Modify the code if necessary to solve this problem.

```
// calculates the next position of the particle.
// Duration denotes the time interval
//since the last update.
void integrate(float duration)

// Sets the position of the particle by component.
void setPosition(const float x, const float y)

// Gets the position of the particle.
Vector2 getPosition() const

// Sets the velocity of the particle by component.
void setVelocity(const float x, const float y) )

// Gets the velocity of the particle.
Vector2 getVelocity() const

//Sets the radius of the particle.
void setRadius(const float r);

// Gets the radius of the particle.
float getRadius() const;
```

Listing 4

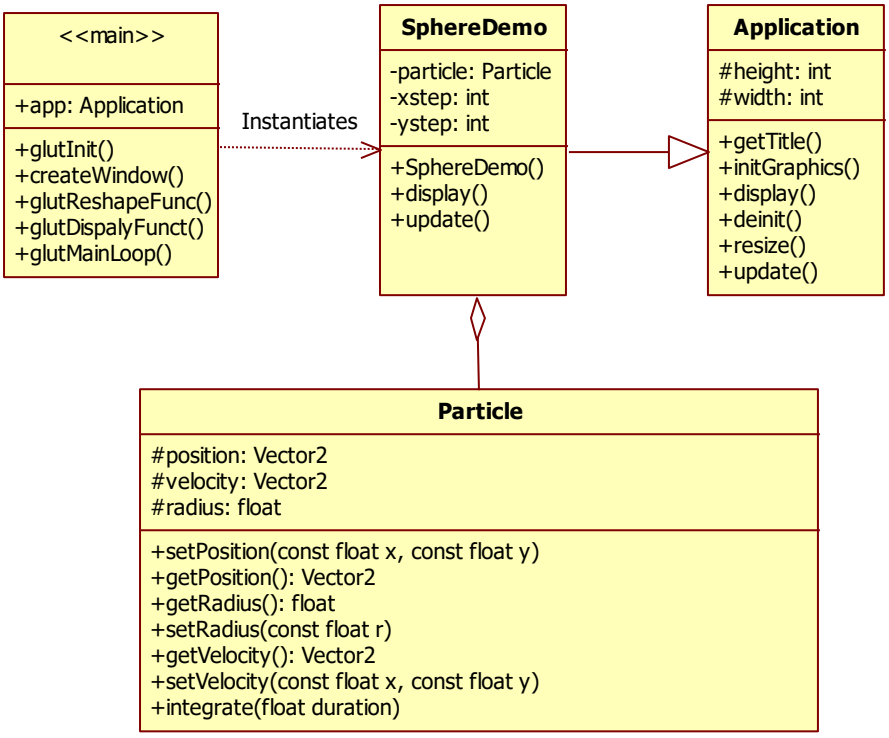
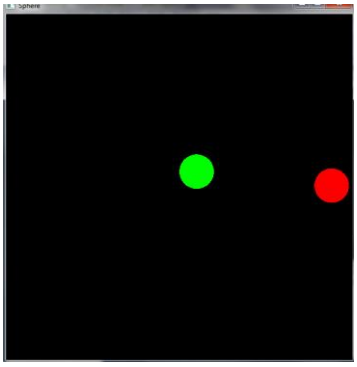


Figure 3

Compile and execute the code. Confirm the correct behaviour of the application.

Exercise 5: Two Particle Simulation

Make further changes to the application from the exercise 4 in order to simulate the motion of two spherical particles across the window. Compile and execute the code. Confirm the correct behaviour of the application.



Exercise 6: Multiple Particle Simulation

Make further changes to the application from the exercise 5 to simulate the motion of arbitrary chosen spherical particles across the window. Assign different positions, radii and velocities to the particles. Compile and execute the code. Confirm the correct behaviour of the application.

Exercise 7: Different Particle Trajectories

Experiment with different particle trajectories. E.g modify the code to make the particle gradually slow down as it travels inside the box until the full stop.

Exercise 8: Multiple Particle Collision

Implement simple form of collision detection and resolution between the particles. Compile and execute the code. Confirm the correct behaviour of the application.