DH2323 DGI16
Lab2
Transformations

In Lab1, you used SDL (Simple DirectMedia Layer) which is a cross-platform development library designed to provide low level access to audio and input devices. This lab will use another library in place of SDL: *GLUT* is the OpenGL Utility Toolkit (GLUT). Similar to SDL, GLUT is a library of utilities for OpenGL, which takes care of all the system-specific chores required for creating windows, initializing OpenGL contexts, and handling input events, to allow for truly portable OpenGL programs. If you are really interested, a more updated and free-software alternative is available here: `http://freeglut.sourceforge.net/`.

This lab will introduce you to transformations, which allow you to place objects in 3D environments and relative to each other. An understanding of transformations is fundamental to computer animation and this lab will provide you with the knowledge required to animate a 3D tank model in Lab 3 (see Figure 1).

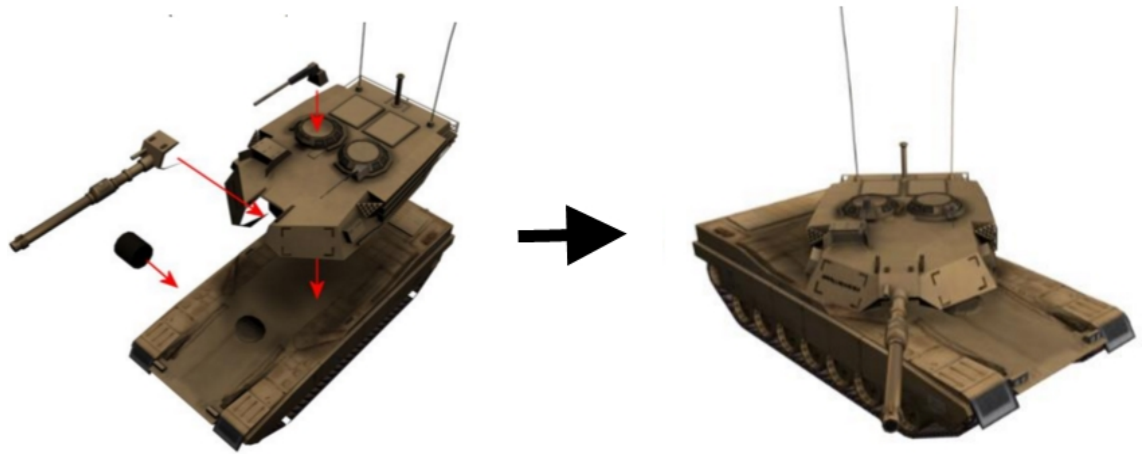


Figure 1: This lab is a basis for working toward a tank model that you will animate in real-time in Lab3 using OpenGL and hierarchical transformations.

The lab consists of four related tasks.

Tasks:

1. Create a small library for conducting math operations involving vectors and demonstrate them through a small interactive graphics program using OpenGL (Section 1).
2. Investigate translation and rotation transformations for simple primitives, including using the OpenGL *Modelview matrix* and interacting with it using your own matrix class (Section 2).
3. Implement a basic *quaternion* class and use quaternion operations with OpenGL to rotate a point around an arbitrary axis (Section 3).
4. Use a basic particle system library to generate, update and visualise large numbers of blended particles in real-time (Section 4).

1 Vector Class and Demonstrator

This section task is based on *vectorLab.zip*. Create a project for the *vectorLab* program and then build and run it. Do not forget to add the relevant GLUT files to the compilation and linking process. An example Microsoft Visual Studio 2010 project is included in the source file. The main code is contained in the vectorLab.cpp file and is heavily commented, although you will need to understand only a small subset of the code in order to complete this task. Section 1.1 contains a further description of the relevant sections of code that concern the display of vectors. The questions are specified in Sections 1.2 and 1.3.

1.1 Introduction to Drawing Primitives in OpenGL

When you build the vectorLab program, it will display the x and y axes in red and blue, respectively, and a yellow vector (see Figure 2). The dimension of the visible area is 10 units along the Y-axis and 10 units along the X-axis.

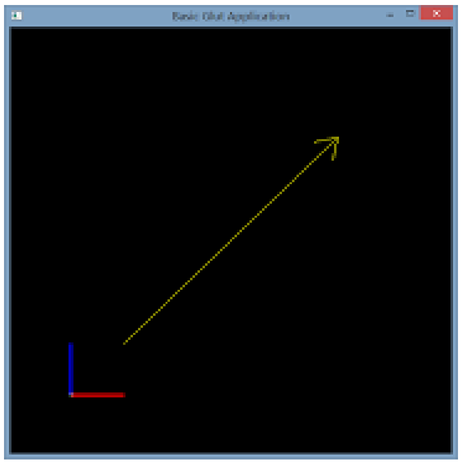


Figure 2: The output of the demo code.

Have a look at the *DisplayScene()* function, which is the central function that concerns drawing to the screen. Most of the code in this function relates to drawing the lines in OpenGL that represent vectors. Here is an example of drawing part of the yellow line in Figure 2:

```
//draw a red horizontal line, one unit long
 glLineWidth(3.0);//set the width of line
 glColor3f(1.0,1.0,0.0);//set the color to be yellow
 glPushMatrix();//store the current transformation as a matrix on a stack
    glTranslatef(0.0, 0.0, 0.0);//translate origin to (0,0,0)
    glBegin(GL_LINES);//specify the vertices defining a primitive (in this case, a line)
     glVertex2f(0.0,0.0);//2D vertex specifying start of line
     glVertex2f(1.0,1.0);//2D vertex specifying end of line
    glEnd(); //finished specifying lines to draw
 glPopMatrix();//retrieve the previously stored matrix from the stack
```

glBegin() and *glEnd()* specify the vertices that define a primitive or a group of like primitives. *glBegin()* accepts a single argument that specifies the way how the vertices are interpreted, such as *GL_LINES, GL_TRIANGLES, GL_POLYGON*. The code to draw a triangle with three vertices $a(0,0,0)$, $b(1,0,0)$, and $c(0,1,0)$ is as below:

```
glBegin(GL_TRIANGLES);
    glVertex3f(0.0f,0.0f,0.0f);
    glVertex3f(1.0f,0.0f,0.0f);
    glVertex3f(0.0f,1.0f,0.0f);
glEnd();
```

Also examine *DrawVector()*. This function draws three lines in order to represent a vector. See if you can figure out how it does so using the OpenGL line drawing code mentioned above. *glPushMatrix()* and *glPopMatrix()* will be discussed further in Section 2.1.3, but for now, you can think of them as saving and loading your position on the drawing area.

- Remember: vectors are not the same as lines. Vectors do not have a position – they are merely a direction. Vectors are drawn with arrowheads denoting their direction, lines do not contain arrowheads. You can convert a line into a vector (thus losing its positional information), or into a position and a vector.

1.2 Task 1a: Complete the Vector Class

First of all, the accompanying vector class is unfinished (see vector.h and vector.cpp). You should complete each of the unfinished functions (marked with the text *'your code here'*) by adding code in the appropriate functions of the vector class in the vector.cpp file. They relate to adding, subtracting, and normalising vectors, in addition to querying their length and doing dot product and cross product operations between them and other vectors. Specifically, you need to complete the following functions from the *vector.h*:

```
   Vector addTo(const Vector &other) const;
   Vector subtractFrom(const Vector &other) const;
   float getMagnitude(void) const;
   void setMagnitude(const float m);
   float getDotProduct(const Vector &other) const;
   Vector getCrossProduct(const Vector &other) const;
   void normalise(void);
```

Here is an example of the implementation for the vector substraction operation (in *'vector.cpp'*):

```
Vector Vector::subtractFrom(const Vector &other) const
{
   Vector result;
   result.x = other.x - this->x;
   result.y = other.y - this->y;
   result.z = other.z - this->z;
   return result;
}
```

If you need to remind yourself of some of the mathematical background of vector operators, refer to the course lecture notes or `https://en.wikipedia.org/wiki/Euclidean_vector`.

1.3 Task 1b: Build a Demonstrator Program

For each of the following, add a new function in the vectorLab.cpp file which will be called from the *DisplayScene()* function. The name of the function is up to you, but you are recommend to use clear function names, for example, *Answer1()* for the code displaying the answer to question 1 below. Inside each function, draw all vectors and positions showing the main inputs and results of the following operations:

b1. Place the vector (4.0, 2.0, 0.0) at position (1.0,2.0,0.0).

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.

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.

- Hint 1: Use dot product with two unit vectors.
- Hint 2: The cos(x) function (contained in *math.h*) returns results in radians: use the function RADTODEG(angle), defined in *vector.h* as ($angle*180.0/PI$), to convert from radians to degrees.

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.

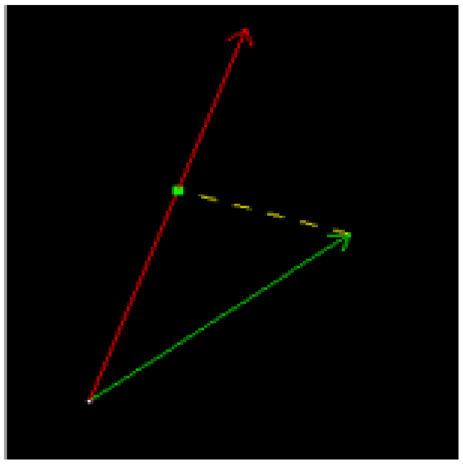


Figure 3: The displayed result of Q5.

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).

- Hint: if there are two vectors and one of them is unit length, the dot product is the projection of the other vector onto the unit length vector. The result is the distance along the vector. Result should look similar to Figure 3.

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) to (-6.0,0.0,0.0)

b7. Determine the closest point on the line from (-2.5, -2.0, 0.0) to (5.0, -2.0, 0.0) to the position (8.0, 3.0, 0.0).

Note: Each example should show the inputs to the function being demonstrated and the results of the operation. See Figure 3 for an example of the result displayed for Q5. If the result of an operation is a scalar quantity, then it should be printed out (hint: include *stdio.h* and use the *printf()* function). You should also try to keep the display within the visual area (as shown in Figure 2).

In order to nicely display the results of each of the questions separately, you might like to attach the display to a key press. Key presses can be detected in the *key()* function.

- Note: You should never call OpenGL draw functions directly from the *key()* callback. Instead, key presses should be used to set variables that are then evaluated within the *DisplayScene()* function in order to alter what is drawn to the screen.

2 Transformations and Matrices

The tasks in this Section are based on *mathLab.zip*. Once you open the archive, you will notice a more comprehensive set of source files for vectors, matrices and quaternions. You will not use the quaternion class until the next task. Your first step should be to create a project for the mathLab files and then build it into an executable. The result will be an empty black display window. You are advised to draw a simple primitive in order to ensure that the scene is properly set up (for example, see the lines denoting the principle axes in Task 1). Once you have completed this and established that everything is being displayed properly, you should copy over your vector library that you completed in Task 1. Note that some superficial changes will be required e.g. updating the name of the vector class. The main.cpp file contains the primary interface and draw functions for the lab. It is a minimal OpenGL program and could be used as a basis for other OpenGL programs that you create.

2.1 Introduction to Transformations in OpenGL

Transformations are used to position and orient objects in the 3D graphics environment. For this reason, they are fundamental to the processes of composing and animating scenes and objects. Two important transformations are translations and rotations. Each will be presented in the following sections.

2.1.1 glTranslatef(x,y,z)

glTranslate(x,y,z) produces a translation by x y z. It is notable that when using *glTranslatef(x,y,z)*, the translation is not relative to the center of the screen, but to current position. The following example program will help to understand *glTranslate(x,y,z)*:

```
glLoadIdentity();//move the current origin to the
                //center of the screen, as reset of position.
glTranslatef(-1.5f,0.0f,-6.0f);
glBegin(GL_TRIANGLES);
    glVertex3f(0.0f,0.0f,0.0f);
    glVertex3f(1.0f,0.0f,0.0f);
    glVertex3f(0.0f,1.0f,0.0f);
glEnd();

glLoadIdentity();//move the current origin to the
                //center of the screen, as reset of position.
glTranslatef(0.0f,0.0f,-6.0f);
glBegin(GL_TRIANGLES);
    glVertex3f(0.0f,0.0f,0.0f);
    glVertex3f(1.0f,0.0f,0.0f);
    glVertex3f(0.0f,1.0f,0.0f);
glEnd();
```

The left one is drawn in the first step. Start at world space origin (0,0,0) with *loadidentity()*, move to (-1.5,0.0,-6.0) with the first translate call, draw the triangle using *glBegin()* and *glEnd()* to connect

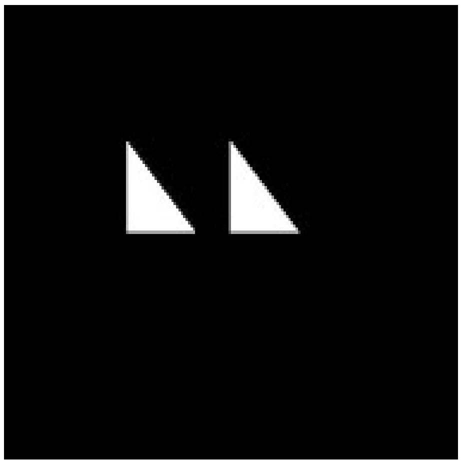


Figure 4: The graphic output of the glTranslatef example.

three vertices of the triangle. Before drawing the second triangle, *glLoadIdentity()* moves the origin back to the center of the screen. The drawing steps are similar to the drawing of the first triangle.

2.1.2 glRotatef(angle, x, y, z)

glRotatef(angle, x, y, z) rotates *angle* around the rotation axis with direction (x,y,z) by right-hand rule.

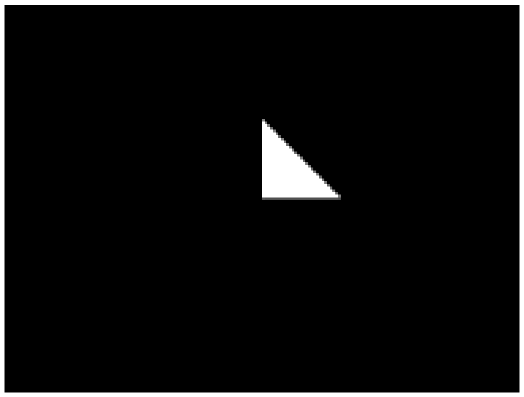
```
//drawing the first triangle without rotation.
glLoadIdentity();
glTranslatef(0.0f,0.0f,-6.0f);
glBegin(GL_TRIANGLES);
    glVertex3f(0.0f,0.0f,0.0f);
    glVertex3f(1.0f,0.0f,0.0f);
    glVertex3f(0.0f,1.0f,0.0f);
glEnd();
```

```
//drawing the second triangle rotated around z-axis with 45 degrees.
glLoadIdentity();
glRotatef(45, 0.0f, 0.0f, 1.0f);
glTranslatef(0.0f,0.0f,-6.0f);
glBegin(GL_TRIANGLES);
    glVertex3f(0.0f,0.0f,0.0f);
    glVertex3f(1.0f,0.0f,0.0f);
    glVertex3f(0.0f,1.0f,0.0f);
glEnd();
```

2.1.3 glPushMatrix() and glPopMatrix()

Before we go to the details of *glPushMatrix()* and *glPopMatrix()*, we will talk about *glMatrixMode()*. At the beginning of the *draw()*, we can see the code below:

```
//clear the current window
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```


(a) Without rotation

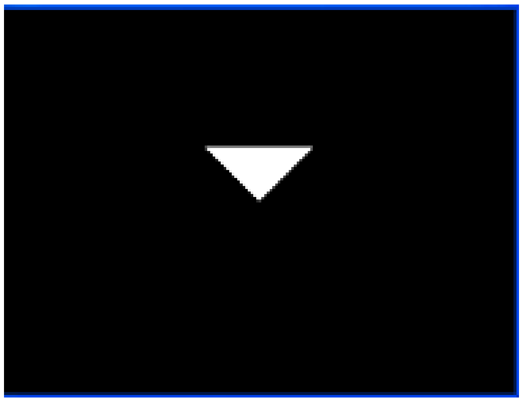

(b) With rotation

Figure 5: The graphic output of the glRotatef example.

```
//make changes to the modelview matrix
glMatrixMode(GL_MODELVIEW);
```

glMatrixMode() specifies which matrix is the current matrix. Rendering vertices depends on the current state of matrices call the *model-view matrix* and *projection matrix*. The transformation commands as: *glTranslatef()*, *glPushMatrix()*, *glLoadIdentity()*, etc. effect the current matrix. The command *glMatrixMode()* selects the matrix (model-view or projection) which is affected by the aforementioned commands.

Let's come to the function *glPushMatrix()* and *glPopMatrix()*. The main purpose is that when you do transformations like *glTranslate()* and *glRotate()*, you affect the *modelview* matrix. When you apply several transformations, this matrix will change. *glPushMatrix()* and *glPopMatrix()* essentially allow you to save and retrieve positions and orientations in the 3D scene, without having to go through lengthy operations to undo transformations that you previously made.

OpenGL's matrix stacks are especially useful for implementing hierarchical models. That is, you can define the rigging of a child object (say, a wheel) with respect to its parent body (say, a car), without regard to the location/orientation of the parent when the child is drawn. That is, it allows you to compose transforms easily and intuitively. If you want to transform two objects in a different manner, while ensuring that one is not affected by the other, the code could look like this:

```
glPushMatrix(); // Set current matrix on the stack
    glTranslatef(someX, someY, someZ); // transformation 1
    glRotatef(someangle,someaxis);// transformation 2
    DrawObject(ONE);
glPopMatrix(); // Pop the old matrix without the transformations.

glPushMatrix(); // Set current matrix on the stack
    ...
    // SomeTransformations
    ...
    DrawObject(TWO);
glPopMatrix(); // Pop the old matrix without the transformations.
```

If you want to know more about APIs, please refer to the manual (`https://www.opengl.org/sdk/docs/man2/xhtml/`).

2.2 Tasks

1. Write an OpenGL program to display two squares on the screen, centered at coordinates (1.0,1.0,-5.0) and (-1.0,1.0,-5.0). The function for drawing each square *must* be defined as:

```
void draw_square(void)
{
  //a simple function to draw a square with the current markers
  //orientation and position on screen
  glBegin(GL_POLYGON);
    glColor3f(1.0,0.0,0.0);
    glVertex3f(-1.0,1.0,0.0);
    glColor3f(0.0,1.0,0.0);
    glVertex3f(-1.0,-1.0,0.0);
    glColor3f(0.0,0.0,1.0);
    glVertex3f(1.0,-1.0,0.0);
    glColor3f(1.0,1.0,1.0);
    glVertex3f(1.0,1.0,0.0);
  glEnd();
}
```

Before the square is drawn, apply the necessary translation to place the squares in the appropriate position on the screen. You may need to do an initial translation along the z-axis in order to allow both squares to be seen. Use *glPopMatrix()* and *glPushMatrix()* to ensure that, at the end of the draw function, the OpenGL modelview matrix is set back to its original value. Be careful to ensure that there are the same number of glPushMatrix() and glPopMatrix() commands in the draw() function!

2. Add some code to make both squares rotate around their own z-axes by a certain amount (e.g. 5 degrees) each time the 'r' key is pressed, updating the display to ensure results can be seen. As before, the function for drawing each square **must** be defined as above.

- Hint1: Use *glRotatef()*.
- Hint2: Automate the rotation by adding a small increment to the rotation into the *idle()* function. When doing this, make sure to call the draw function afterwards so that the display is properly updated.

3. Write a program to rotate a square around one of its vertices (rather than its center) when a key is pressed. Use the same drawSquare() function - i.e. the function for drawing each square **must** be defined as above.

- Hint: Use an extra translation to do it – can you figure out what the translation should be and where it should go without resorting to trial and error?

While you make changes indirectly to the modelview matrix through the glTranslate and glRotate commands, it is also possible to directly interact with the matrix, by retrieving values from it and loading them into it. The *myMatrix* class defined in *myMatrix.h* and *myMatrix.cpp* provides some of these capabilities. For example, you can define an identity matrix and load it into the currently selected OpenGL matrix (essentially creating your own version of the *glLoadIdentity()* command) with the following code:

```
GLfloat myIdentityMatrix[16] =
{
    1.0,0.0,0.0,0.0,
    0.0,1.0,0.0,0.0,
    0.0,0.0,1.0,0.0,
    0.0,0.0,0.0,1.0
};
glMatrixMode(GL_MODELVIEW);
glLoadMatrixf(myIdentityMatrix);
```

You can also multiply the current OpenGL matrix by your own one with the command: *glMultMatrix(yourMatrix)*;

4. Using the functions in the supplied matrix class, load the current OpenGL matrix into your class, change the translation values, and load it back into the OpenGL matrix. Confirm that you get the same results when you use a glTranslate() function call instead.

5. Using the functions in the supplied matrix class, re-implement Question 2 to use your own rotation function instead of glRotatef(). The function only needs to be capable of handling rotations around the z-axis.

- Note: Be careful when you invoke calls that make changes to OpenGL matrices: OpenGL is a state machine, so all changes will impact the *currently selected* matrix. There are other matrices in addition to the modelview matrix. If you intend for a matrix operation to take place on the modelview matrix, make sure that matrix type has been selected by first calling the *glMatrixMode()* function.

3 Quaternions

In this task, we will continue to use the *mathLab* project that you developed as part of Task 2. You will extend it to include a Quaternion class and then use that class to enable you to rotate points.

3.1 Introduction

Quaternions are unit vectors on a 3-sphere in 4 dimensional space. Defined like complex numbers but with 4 coordinates. $q[w,(x,y,z)]$ also written $q[w,v]$, where $v=(x,y,z)$. Here, w is real part, and (x,y,z) are imaginary parts. Unit quaternions provide a convenient mathematical notation for representing orientations and rotations of objects in three dimensions.

To transform a vector P by the rotation specified by the quaternion q, we have two options: a) multiply *conj(q)* by $(0,Px,Py,Pz)$, b) convert q to matrix and use matrix transformation. In the following subsection, the first option will be explained through a worked example. Your task will then be to implement it in code using functions that you create in your quaternion class.

3.1.1 Worked example

Before we start programming, we will demonstrate a worked example of rotation with quaternions. You are recommended to follow this example with your own calculations on paper in order to understand how the end result is obtained.

Q. Using quaternions, rotate the vector (1,0,0) 90 degrees around the axis (0,0,3) by hand.

A. First of all, normalise the axis: you obtain (0,0,1)

The general process is:

- Create a quaternion *qvec* from the vector
- Create a quaternion *q1* from the angle-axis rotation
- Obtain the conjugate of *q1*, called *q1Conj*
- Multiply *q1*(qvec*q1Conj)* to obtain quaternion *qr*. This is accomplished in two substeps:
  a. Multiply *(qvec*q1Conj)* to obtain *qrA*
  b. Multiply *(q1 * qrA)* to obtain *qr*
- Extract the imaginary component of *qr* to get the resulting rotated vector

First of all, we already know that the final answer should be (0,1,0). That is very important.

Steps 1,2,3: Represent the rotation and vector as quaternions

$$\begin{aligned} qvec &= [0,(1,0,0)] \\ q1 &= [cos(90/2),((xsin(90/2),ysin(90/2),zsin(90/2))] \\ &= [cos(45),(0*sin(45),0*sin(45),1*sin(45))] \\ &= [0.7071,(0,0,0.7071)] \\ q1Conj &= [0.7071,(0,0,-0.7071)] \end{aligned}$$

Step 4a: Calculate *qvec * q1Conj*

Here, $[w1,v1]$ refers to *qvec* and $[w2,v2]$ refers to $q1Conj$:

$$\begin{aligned} &w1 : 0 \\ &v1 : (1,0,0) \\ &w2 : 0.7071 \\ &v2 : (0,0,-0.7071) \end{aligned}$$

Our quaternion multiplication equation: $= [w1*w2 - v1\cdot v2, v1\times v2 + w1v2 + w2v1]$

Here:

$$\begin{aligned} v1\cdot v2 &= \text{dot product of } v1 \text{ and } v2 \\ &= (1*0)+(0*0)+(0*-0.7071) \\ &= 0 \end{aligned}$$

$$\begin{aligned} v1\times v2 &= \text{cross product of } v1 \text{ and } v2 \\ &= ((v1.y*v2.z)-(v1.z-v2.y),(v1.z*v2.x)-(v1.x-v2.z), \end{aligned}$$

Inserting the above into our multiplication equation, we have:$(qvec*q1Conj) = [0,(0.7071,0.7071,0)]$.

We will refer to the result of $(qvec*q1Conj)$ as qrA

Step 4b: Calculate $q1*qrA$

The step is quite similar as Step 4a, I will let you do the multiplication calculation.

Step 5: Extract rotated vector from final quaternion *qr*

Extract the result from the quaternion $qr = [0,(0,1,0)]$ Just look at the imaginary part: $[0,(0,1,0)]$. We get the final vector $(0,1,0)$.

3.2 Tasks

1. Create your own quaternion class using the code provided in the *myQuat.h* and *myQuat.cpp* files. Ensure that it contains implementations of the following functions:

```
//constructor - create from point
Quaternion(Vector &point);
//constructor - create from axis angle
Quaternion(float angleDeg, Vector &axis);
Quaternion addTo(Quaternion &other);
Quaternion multiplyBy(Quaternion &other);
float getMagnitude(void);
void normalise(void);
Quaternion getConjugate(void);
Quaternion getInverse(void);
Matrix convertToRotationMatrix(void);
```

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). In the same way that you demonstrated the vector class in a previous task, use the OpenGL draw functions to display the basis vector, origin, initial point and final rotated point i.e. see Figure 2.

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). As above, use the OpenGL draw functions to display the basis vector, origin, initial point and final rotated point i.e. see Figure 2.

4 Particle System

A particle system is a technique in game physics and computer graphics that simulates the interactions between a large number of small objects in order to model various types of phenomena, ranging from snowfall, rain and sparks, to solar systems and galaxies. In this section, you will make a simple particle system using a pre-existing library that can be found in the *particleLab*.

4.1 Introduction

The project includes a number of files – *particleSystem.h* and *particleSystem.cpp* are the core files for simulating the particle system itself. *tgaLoader.h*, *tgaLoader.cpp*, *objLoader.h* and *objLoader.cpp* are extra files that you do not need to directly work with. If you are new to programming projects that have multiple files, be aware that you do not need to understand or even read the source code in many files in order to be able to use the classes. In this case, having a look at particleSystem.h will give you an idea of the interface that we will be using. The actual particle texture that will be used is based in the file *'particle.bmp'*. Have a look at that file to see the texture. This particle implementation uses a special OpenGL extension that enables what are called *Point Sprites*. Point sprites are a way of defining some geometry at a particular point in space that is oriented towards the camera – for particle systems, they are perfect for use as the quad (four vertex, two triangle) billboards that represent each particle.

This program implements a particle system using the GLUT program from Task 2 and 3 as a basis.

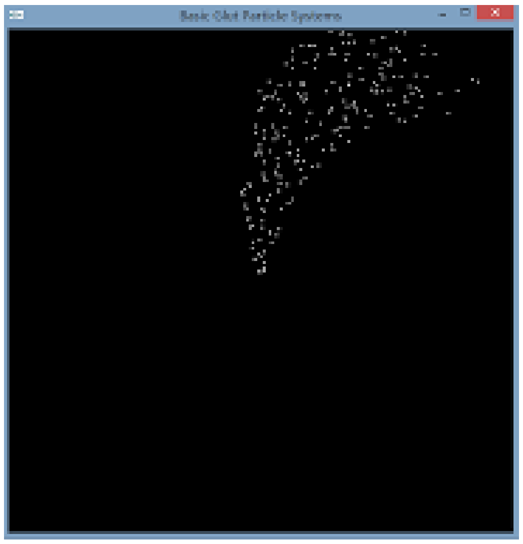
In the *main.cpp* file, we declare space for 6 particle systems, with the line of code:

```
CParticleSystem *g_pParticleSystems[6];
```

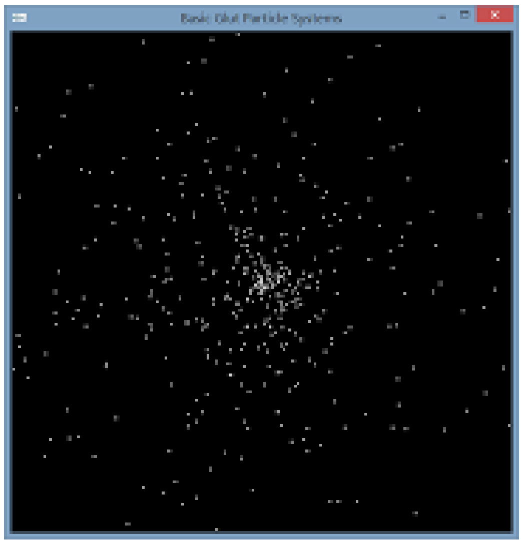
CParticleSystem is the particle system object. Each one can have different associated attributes and update in different ways. This is done when the particle system objects are being created in the function

```
void initParticles(void)
```

The initial position, velocity, size, colour and forces affecting the particle system can be defined here. Six different systems have already been defined – laster, you can change the code so that you can see the results of the other 5 particle systems. This can be done by simply changing the *g_nActiveSystem* variable, which defines the currently activated particle system. Two types of particle systems are shown in Figure 6.



(a) Type 1



(b) Type 2

Figure 6: The graphic output of the glRotatef example.

4.2 Tasks

First of all, we need to be able to see the particle system animating.

1. The particle system is currently static. Since we want the particle system to update on its own automatically, we add the particle system's *update* function to the *idle()* function as follows:

```
//idle callback function - this is called when there is nothing
//else to do
void idle(void)
{
  //this is a good place to do animation
  //since there are no animations in this test, we can leave
  //idle() empty

   g_dCurTime     = timeGetTime();
   g_fElpasedTime = (float)((g_dCurTime - g_dLastTime) * 0.001);
   g_dLastTime    = g_dCurTime;

// The particle system will need to know how much time has
// passed since the last time it was updated, so we will need to
// keep track of how much time has elapsed since the last frame
//update...
   //
   g_pParticleSystems[g_nActiveSystem]->Update((float)g_fElpasedTime );

   //redraw the OpenGL window
   glutPostRedisplay();
}
```

When you rebuild the particle system, you should now see it updating: several simple white points (particles) will be generated near the center of the screen and move outwards. This is important for updating not only the particle system, but any sort of simulation where we want the system to update on its own without having the user need to press a key.

2. Create your own new particle system and add it to the list of current systems. Define your own attributes (particle size, forces etc) for this system and display it.

3. You can also create collision planes for particles to collide with, and these are defined at the bottom of the particle initialisation function. For each of these planes, draw the plane represented by a quad. Be careful when drawing the quad – if you have textures enabled to draw the particles and the plane does not have a texture, then nothing will be drawn. You will need to disable textures before drawing the plane and then re-enable afterwards in order to make sure the particle is properly drawn. Create one new collision plane and demonstrate particles interactions with it.

4. When the particle system is rendered, it can be blended in with its background in order to produce a much nicer visual effect. This is done in the draw(void) function of main.cpp. Enabling *GL_DEPTH_TEST* and setting *glDepthMask* to *GL_FALSE* helps to remove graphical artifacts from the unsorted particle system. It does this by making the Z-Buffer read-only. By enabling *GL_BLEND* and setting *glBlendFunc* apppropriately. Read about the glBlendFunc() and see if you can find the correct alpha blending parameters that will correctly display the particle system as shown in Figure 7.

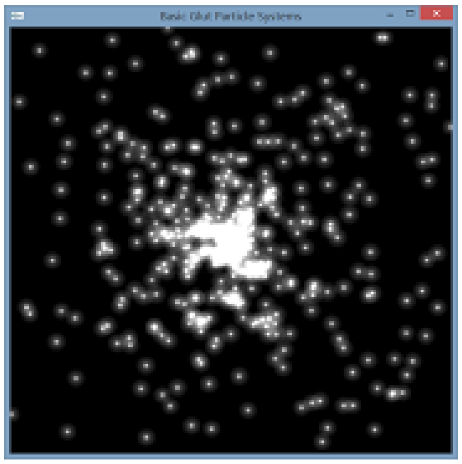


Figure 7: The output when GL_Blend is enabled