Computer Science

COMPSCI 372 S2 C - Assignment 2 Sample Solution

This assignment is worth 6.25% of your final grade.

In this assignment you will create a little game containing a plane with grass, 20 flowers, Easter eggs falling from the sky, a robot with a basket at the end of its arm and a reflecting mirror. The objective of the game is to catch the Easter eggs with the basket. If you hit an egg with the rim of the basket or if you miss it the egg gets smashed [please feel free to extend the game and add appropriate sound effects ;-)]. The game finishes after 50 eggs have been smashed.

Download the file Ass2.zip which contains a .NET solution including all necessary source files. Unzip the files and compile and run them. You should see a grass covered plane, 20 flowers, a non-reflecting mirror, and Easter eggs falling from the sky.

NOTE: In order to make debugging easier and improve the understanding of texture maps, all texture images are stored in Portable Pixel Map format (ppm). This format is very inefficient to load and your program might take a long time to load when run in DEBUG mode. You can avoid this by compiling your program in RELEASE mode.

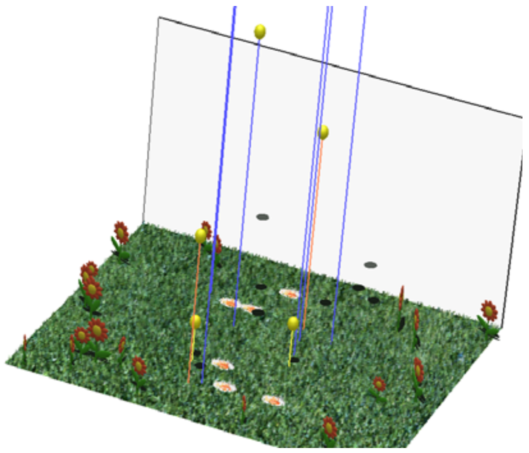
1. Peerwise (8 Marks)

The program Peerwise enables students to submit and answer multiple choice questions and to rate and comment on them. The idea is to encourage students to not only test their knowledge by answering questions, but also to gain a deeper understanding by devising new questions and plausible, but incorrect, multiple choice answers.

Solution: As long as you have submitted one appropriate course related question and answered two questions you will get full marks for part (a) and (b). For part (c) we expect honest answers with justification – we are aware that Peerwise can be cumbersome for computer graphics related questions and constructive criticism is always welcome.

2. 3D Geometry & Transformations (8 Marks)

(a) **[2 marks]** When playing the finished game you will find that it is hard to catch eggs since it is difficult to perceive the exact 3D position of an egg and where it will land. Implement the method `drawPath()` in the `CEgg` class, which draws a line from the centre of each egg to the ground. A line is coloured blue (0.3, 0.3, 1.0) if the distance to the ground is larger than 3 units, orange (1.0, 0.5, 0.3) if the distance is between 1.5 and 3 units and yellow (1.0, 1.0, 0.3) otherwise. Visibility of the lines is toggled by pressing the 'h' key (already implemented). After completing this exercise your programs should look similar to the screen shot on the right:



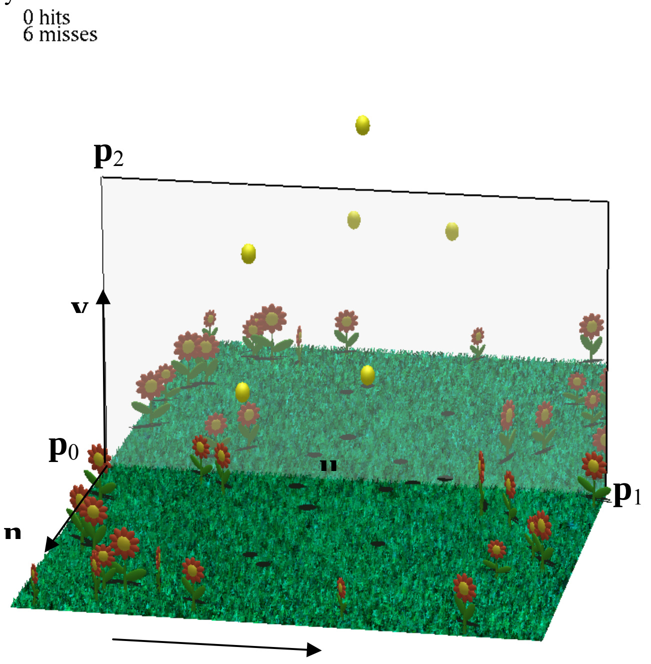
Solution: The path is defined as a simple line segment orthogonal to the ground (xz) plane and hence is given by the position of an egg and its position with the y-coordinate set to 0.

```
void CEgg::drawPath()
{
	if (position[1]>3)
		glColor3f(0.3f, 0.3f, 1.0f);
	else if (position[1]>1.5)
		glColor3f(1.0f, 0.5f, 0.3f);
	else
		glColor3f(1.0f, 1.0f, 0.3f);
	glBegin(GL_LINES);
	glVertex3fv(position);
	glVertex3f(position[0],0,position[2]);
	glEnd();
}
```

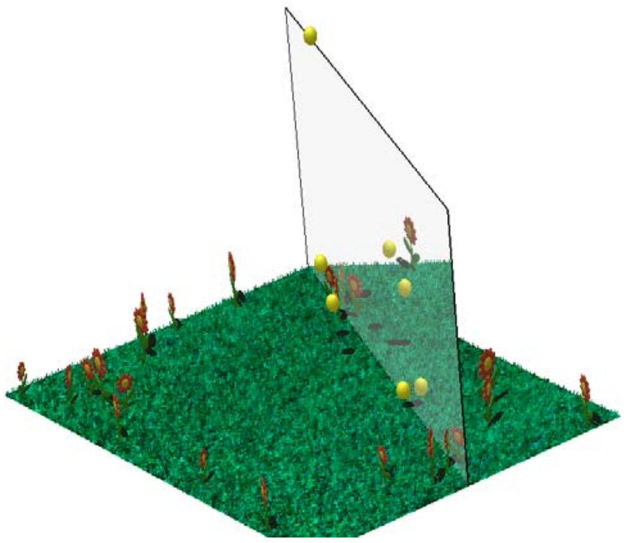
(b) [6 marks] In this question you will complete the code for implementing a mirror. The complete implementation is a bit tricky: the reflected scene is drawn such that it is only visible in the polygon representing the mirror (this is achieved using a *stencil buffer*). Furthermore the light sources are mirrored and the reflected image is blended with the grayish colour of the mirror.

Your task is to implement the functions `initReflectionMatrix()` and `initWorldToMirrorTransformation()` in the file `CMirror.cpp`. The first function initialises a reflection matrix, which reflects an object on the *xy*-plane, i.e. it inverts the *z*-coordinate of any point. The second function initialises a matrix, which transforms an object from the *uvn*-coordinate system of the mirror, defined by the points $\mathbf{p}_0$, $\mathbf{p}_1$ and $\mathbf{p}_2$, to the xyz-coordinate system in which the scene is defined. In order to do this the matrix must perform a translation by $\mathbf{p}_0$ and a rotation of the *uvn*-coordinate system into the *xyz*-coordinate system (Hint: Such a rotation is described in your lecture notes!)

The *uvn*-coordinate system of the mirror is illustrated below:



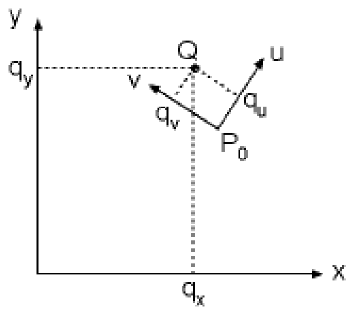
Note: Since the mirror is parallel to the xy-plane it is very easy to specify the required matrices directly. In order to get full marks you have to find the general transformation matrix for any values of **p0**, **p1** and **p2** of the mirror. For example, if you change $\mathbf{p}_1$ to (2.5, 0, 0) you should get the result shown below:



Solution: The reflection matrix inverts the n-coordinate of a point in mirror coordinates. This is achieved by scaling it with –1.

```
void CMirror::initReflectionMatrix()
{
   for(int i=0;i<4;i++)
   {
      for(int j=0;j<4;j++)
         reflectionMatrix[i][j]=0.0;
      reflectionMatrix[i][i]=1.0;
   }
   reflectionMatrix[2][2]=-1.0; // invert the third coordinate
}
```

The mirror-to-world transformation computes the xyz-coordinates of a point given in uvn-coordinates. In order to find the required transformation consider the situation in 2D as illustrated in the image below:



Given is a point Q and a uv-coordinate system with the origin P_0. The point Q has the coordinates (q_u,q_v) in the uv-coordinate system. We want to find the coordinates (q_x,q_y) of the point Q in the xy-coordinate system. In order to do this we first rotate the point Q with the matrix R that aligns the xy-axis with the uv-axis and we then translate it by $\mathbf{p}_0$. The rotation matrix is given by the uv-coordinate axes (Chapter 6, slide 38-39).

In this exercise the uvn-coordinate system is given by the points $\mathbf{p}_0$, $\mathbf{p}_1$, and $\mathbf{p}_2$ of the mirror: first compute $\mathbf{u}=\mathbf{p}_1-\mathbf{p}_0$, $\mathbf{v}'=\mathbf{p}_2-\mathbf{p}_0$ and $\mathbf{n}=\mathbf{u}\times\mathbf{v}$, and then $\mathbf{v}=\mathbf{n}\times\mathbf{u}$. All three vectors must be normalised. Note that **v'** not necessarily orthogonal to **u** so that we have to get **v** from **n** and **u**. However, because we only reflect the n-coordinate the program will work even if **v** is not orthogonal to **u**. Using these vectors and applying above idea in three dimensions gives the desired transformation matrix **M**:

$$\mathbf{M}=\mathbf{T}*\mathbf{R}=\begin{pmatrix}1&0&0&p_{0,x}\\0&1&0&p_{0,y}\\0&0&1&p_{0,z}\\0&0&0&1\end{pmatrix}\begin{pmatrix}u_x&v_x&n_x&0\\u_y&v_y&n_y&0\\u_z&v_z&n_z&0\\0&0&0&1\end{pmatrix}=\begin{pmatrix}u_x&v_x&n_x&p_{0,x}\\u_y&v_y&n_y&p_{0,y}\\u_z&v_z&n_z&p_{0,z}\\0&0&0&1\end{pmatrix}$$

The implementation is straight forward and is shown below:

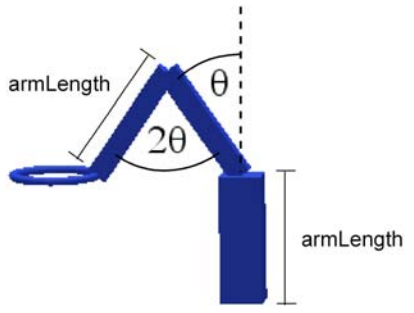
```
        // compute u,v,n coordinate system of the mirror
        CVec3df u=p1-p0;
        CVec3df v=p2-p0;
        n=cross(u,v);
        v=cross(n,u);
        n.normaliseDestructive();
        u.normaliseDestructive();
        v.normaliseDestructive();

        // create matrix m which transforms a point from u,v,n coordinates
        // into x,y,z coordinates
        CMatrix m(4,4);
        m(0,0)=(float) u[0];
        m(1,0)=(float) u[1];
        m(2,0)=(float) u[2];
        m(3,0)=0.0;
        m(0,1)=(float) v[0];
        m(1,1)=(float) v[1];
        m(2,1)=(float) v[2];
        m(3,1)=0.0;
        m(0,2)=(float) n[0];
        m(1,2)=(float) n[1];
        m(2,2)=(float) n[2];
        m(3,2)=0.0;
        m(0,3)=(float) p0[0];
        m(1,3)=(float) p0[1];
        m(2,3)=(float) p0[2];
        m(3,3)=1.0;
```

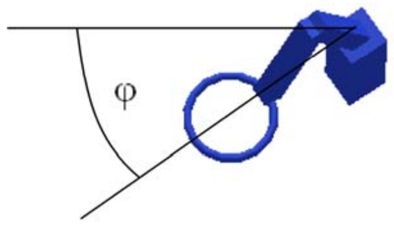
Remark: Since the program uses the inverse of **M** for the reverse transformation (computed in the last 4 lines of the method `initWorldToMirrorTransformation()`) the program works even if **u**,**v** and **n** are not normalised. Since you can't know this I expect you to use normalised vectors.
For the same reason the program will still work if **u** and **v** are not orthogonal.

Here is an explanation (only for the mathematically inclined): If **u**,**v** and **n** are not normalised or are not orthogonal then the matrix **R** is not a rotation matrix (i.e. its transpose is not the inverse) but it is still a valid coordinate transformation matrix. This means that $\mathbf{M}^{-1}$ transforms a point from xyz-coordinates into uvn-coordinates, where it is correctly reflected on the mirror (since **n** is orthogonal to **u** and **v** inverting the third coordinate is still sufficient) and from which it is transformed by **M** back into the xyz-coordinate system. However, in the lecture notes we only talked about orthonormal (orthogonal and normalized) coordinate systems in which case the coordinate transformation matrix is a rotation matrix. I therefore expect that you use this.
By the way, an alternative way to get an orthonormal mirror coordinate system is to use the Gram-Schmidt Orthonormalization method (stage 2 maths).

3. *Modelling with Polygon Meshes* *(12 Marks)*

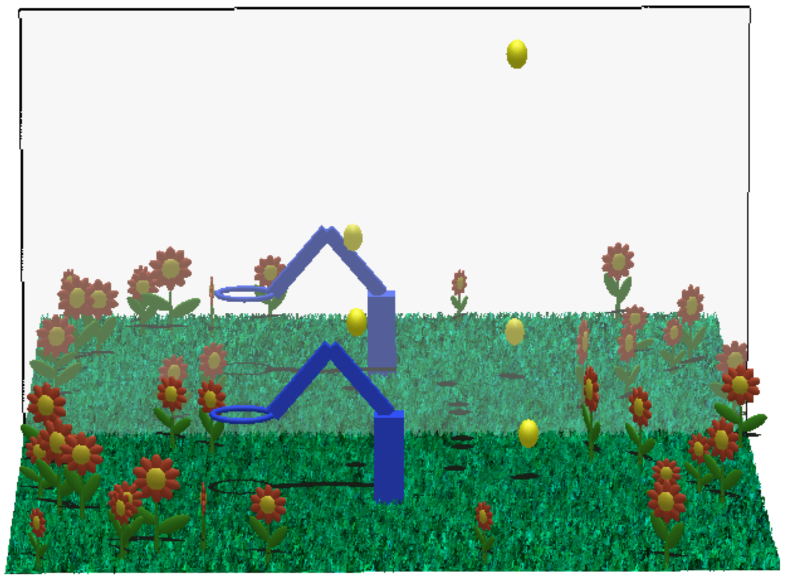
(a) [6 Marks] Define the robot shown in the image below. The robot has three arms with lengths `armLength` and at the tip of the 3rd arm it has a torus which is always parallel to the ground plane. The outer radius of the torus is `basketRadius` and its inner radius is `rimRadius`. The bottom arm is perpendicular to the ground plane and centered at the origin. The whole robot is rotated by an angle φ (phi) around its centre axis. The second arm is rotated by the angle θ (theta) with respect to the bottom arm and the third arm is rotated by 2θ with respect to the second arm as illustrated in the figure below.





At the end of this exercise your robot should look like in the picture below. Note that the keys 'a','s','w' and 'z' are used to modify the angles φ and θ, i.e. at the end of this exercise you should be able to move the robot using these keys.

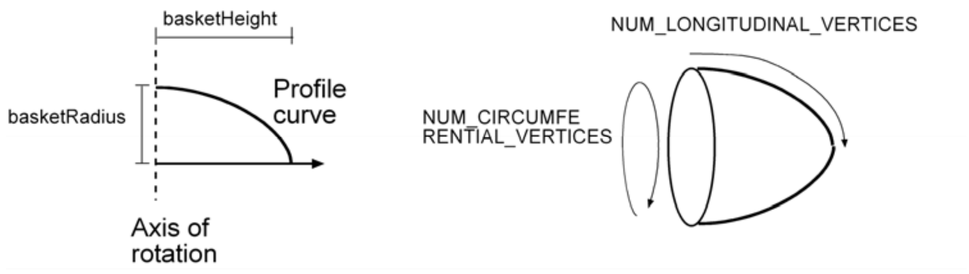
NOTE: For this exercise it is sufficient to use matrix operations (translation, rotation, scaling, push, pop) and two solid objects defined in GLUT. In order to make it easier for you to test your implementation I have defined a variable `basketCentre` which gives you the point at the centre of the torus. You can check your code by drawing this point and by verifying that it is indeed in the centre of the basket rim of the robot.



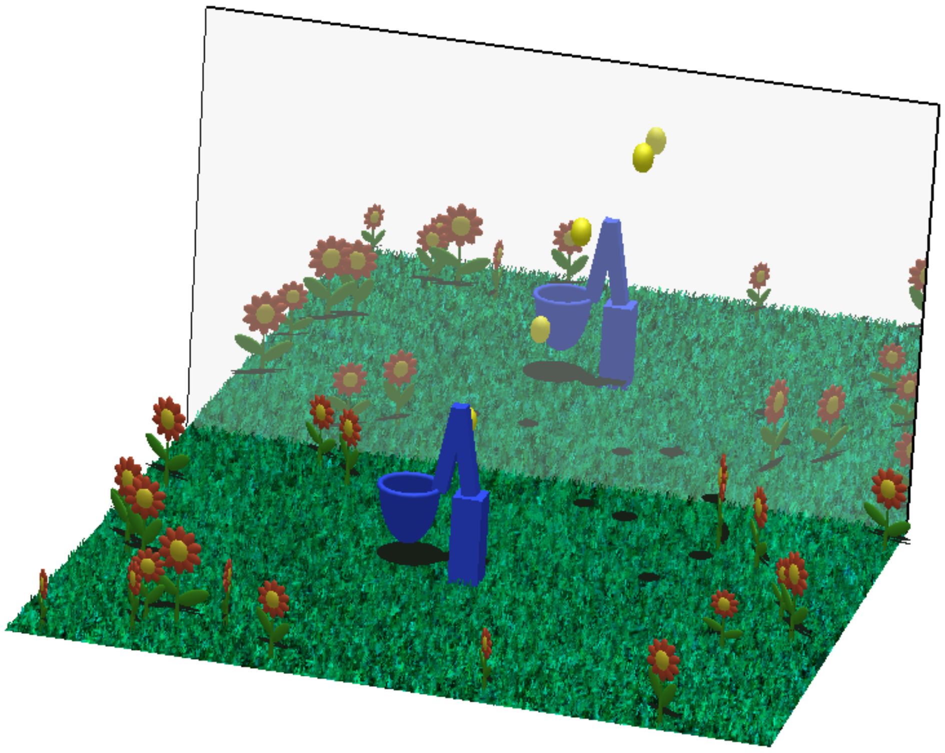
Solution: It is easiest to construct the robot using a bottom-up approach. i.e. start with the bottom arm of the robot and implement the rotation around its axis. By defining this rotation first it will be applied to all subsequent components, which is exactly what we want. Next we implement the middle arm of the robot. Note that the scaling and translation only apply to this arm (i.e. we use glPushMatrix() and gl PopMatrix()), whereas the rotation applies to this arm and all components connected to it (i.e. the top arm and the basket). Finally we define the top arm and the basket connected to it.

```
void CRobot::draw()
{
        glEnable(GL_NORMALIZE);
        glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
        glMaterialfv(GL_FRONT, GL_SHININESS, no_shininess);
        glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, mat_ambient_and_diffuse);

        // rotate the whole robot
        glPushMatrix();
        glRotatef(phi, 0, 1, 0);
        // bottom robot arm
        glPushMatrix();
        glScalef(bottomArmWidth, armLength, bottomArmWidth);
        glTranslatef(0, 0.5, 0);
        glutSolidCube(1.0);
        glPopMatrix();

        glPushMatrix();
        glTranslatef(0, armLength, 0);
        glRotatef(theta, 0, 0, 1);

        // middle robot arm
        glPushMatrix();
        glScalef(topArmWidth, armLength, topArmWidth);
        glTranslatef(0, 0.5, 0);
        glutSolidCube(1.0);
        glPopMatrix();

        // top robot arm
        glTranslatef(0, armLength, 0);
        glRotatef(180-2*theta, 0, 0, 1);
        glPushMatrix();
        glScalef(topArmWidth, armLength, topArmWidth);
        glTranslatef(0, 0.5, 0);
        glutSolidCube(1.0);
        glPopMatrix();

        // ring and basket at the top of the arm
        glTranslatef(0, armLength, 0);
        glRotatef(-90+theta, 0, 0, 1);
        glTranslatef(0, basketRadius, 0);
        glRotatef(-90,0,1,0);
        glutSolidTorus(rimRadius, basketRadius, 16, 16);
        _drawBasket();
        glPopMatrix();
        glPopMatrix();

        glDisable(GL_NORMALIZE);
        glFlush();
}
```

(b) [6 Marks] Complete the function `_drawBasket()` and add it to the draw method of the `CRobot` class so that it draws a basket defined as a **surface of revolution**. Please precompute the vertices of the basket's surface in the constructor of the `CRobot` class and draw the surface using quad strips as indicated in the figure below. Since we later on use texture mapping for the basket it's **not necessary** to define surface normals.
You must define the surface of revolution yourself – using GLUT commands or other geometry libraries gives automatically zero marks. The dimensions and number of vertices of the surface are defined in the code by constants explained in the illustration below.



At the end of this step your scene should look similar to that in the image below.



Solution: We use the formula in the lecture notes to create the basket by rotating a quarter ellipse around the z-axis, i.e. the axis of the resulting torus is aligned with the z-axis. This is practical because the torus defining the ring of the basket is also be default aligned with the z-axis. The surface of revolution is defined by creating first a profile curve **c**(*t*)=(*x*(*t*),0,z(t)) which is obtained by taking the equation of an ellipsoid with the major axis `basketHeight` and the minor axis `basketRadius`. Then

$$\mathbf{c}(t) = \begin{pmatrix} basketRadius * \sin(0.5\pi t) \\ 0 \\ basketHeight * \cos(0.5\pi t) \end{pmatrix}$$

The surface of revolution is obtained by rotating each point of the profile curve around the z-axis, i.e.

$$\mathbf{p}(s,t)=\begin{pmatrix}(basketRadius * \sin(0.5\pi t))\cos 2\pi s\\(basketRadius * \sin(0.5\pi t))\sin 2\pi s\\basketHeight * \cos(0.5\pi t)\end{pmatrix}$$

Note that we don't need the surface normals since later on we texture map this surface. The resulting code is:

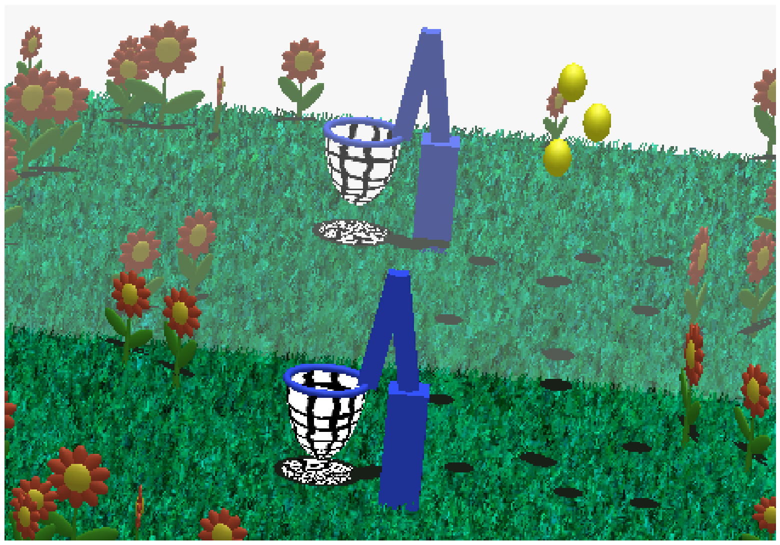
```
        // initialise vertices basket
        int i,j;
        double z,r;

        for(i=0;i<NUM_LONGITUDINAL_VERTICES;i++)
        {
                z=basketHeight*cos(Pi/2*((double) i/(NUM_LONGITUDINAL_VERTICES-1)));
                r=basketRadius*sin(Pi/2*((double) i/(NUM_LONGITUDINAL_VERTICES-1)));
                for(j=0;j<NUM_CIRCUMFERENTIALL_VERTICES;j++)
                {
                        vertices[i][j][0]=(float) (cos(2*Pi*j/(double)
                                (NUM_CIRCUMFERENTIALL_VERTICES-1))*r);
                        vertices[i][j][1]=(float) (sin(2*Pi*j/(double)
                                (NUM_CIRCUMFERENTIALL_VERTICES-1))*r);
                        vertices[i][j][2]=(float) z;
                }
        }
```

and the draw method is

```
void CRobot::_drawBasket()
{
        glShadeModel(GL_FLAT);

        int i,j;
        for(i=0;i<NUM_LONGITUDINAL_VERTICES-1;i++)
        {
                glBegin(GL_QUAD_STRIP);
                for(j=0;j<NUM_CIRCUMFERENTIALL_VERTICES;j++)
                {
                        glVertex3fv(vertices[i][j]);
                        glVertex3fv(vertices[i+1][j]);
                }
                glEnd();
        }

        glShadeModel(GL_SMOOTH);
}
```

4. Texture Mapping *(8 Marks)*

(a) [5 marks] Use texture mapping to draw the basket at the end of the robot arm. Complete the missing code so that it loads the texture map "Net.pgm" and maps it onto the basket. Note that the program contains already a routine to load a texture of type 'Portable Pixel Map' (ppm) - a 'Portable Gray Map" (pgm) is defined similarly but has only one byte per pixel (a greyscale value in the range 0 to 255).
Note that the texture image has been constructed such that the pixels on its left and ride side are identical. Consequently you should map the horizontal direction of the image onto the circumferential direction of the basket's surface in order to get a continuous texture.
At the end of this step your scene should look something like this:



Solution: Change the initialization procedures so that suitable texture coordinates are computed. The texture coordinates map the texture around the basket similarly to wrapping a paper around a cylinder. Note that the texture image will get distorted towards the apex of the basket, i.e. the black lines in it converge, which is exactly what we want to have for a realistic basket.

```
        // initialise vertices basket
        int i,j;
        double z,r;

        for(i=0;i<NUM_LONGITUDINAL_VERTICES;i++)
        {
                z=basketHeight*cos(Pi/2*((double) i/(NUM_LONGITUDINAL_VERTICES-1)));
                r=basketRadius*sin(Pi/2*((double) i/(NUM_LONGITUDINAL_VERTICES-1)));
                for(j=0;j<NUM_CIRCUMFERENTIALL_VERTICES;j++)
                {
                        vertices[i][j][0]=(float) (cos(2*Pi*j/(double)
                                (NUM_CIRCUMFERENTIALL_VERTICES-1))*r);
                        vertices[i][j][1]=(float) (sin(2*Pi*j/(double)
                                (NUM_CIRCUMFERENTIALL_VERTICES-1))*r);
                        vertices[i][j][2]=(float) z;
                        texture_coordinates[i][j][1]=(double)
                                i/(NUM_LONGITUDINAL_VERTICES-1);
                        texture_coordinates[i][j][0]=(double)
                                j/(NUM_CIRCUMFERENTIALL_VERTICES-1);
                }
        }
```

Next we complete the function for the texture initialization. It's easiest to define the texture map in the RGBA format we discussed in the lecture, i.e. the gray scale values of the input image are used as red, green and blue components of the texture image.

```
void CRobot::_initTexture()
{       // load net texture
        ifstream textureFile;
        char* fileName="Net.pgm";
        textureFile.open(fileName, ios::in);
        if (textureFile.fail())
                displayMessage(ERROR_MESSAGE, "CRobot()::CRobot():
                                                could not open file %s",fileName);

        skipLine(textureFile);
        textureFile >> textureWidth;
        textureFile >> textureHeight;
        int numRGBValues;
        textureFile >> numRGBValues;
        texture = new GLubyte[textureWidth*textureHeight*4];

        int k,l,c;
        for(k=0;k<textureHeight;k++)
                for(l=0;l<textureWidth;l++)
                {
                        textureFile >> c;
                        texture[(k*textureWidth+l)*4]=(GLubyte) c;
                        texture[(k*textureWidth+l)*4+1]=(GLubyte) c;
                        texture[(k*textureWidth+l)*4+2]=(GLubyte) c;
                        texture[(k*textureWidth+l)*4+3]=(GLubyte) 255;
                }
        textureFile.close();

        glBindTexture(GL_TEXTURE_2D, texName);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, textureWidth, textureHeight,
                                0, GL_RGBA, GL_UNSIGNED_BYTE, texture);
        delete[] texture;
}
```

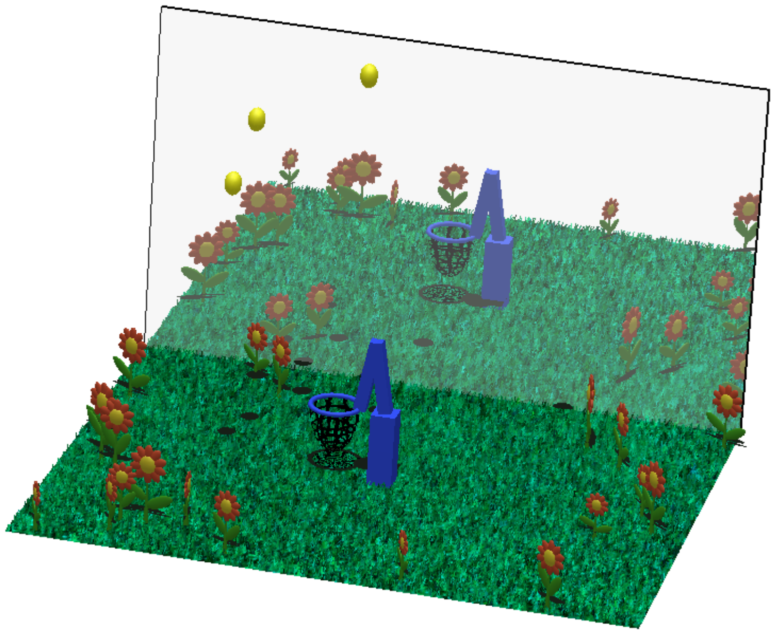
In order to draw the basket we must define the appropriate texture coordinates for each vertex in the `draw()` method:

```
        glEnable(GL_TEXTURE_2D);
        glBindTexture(GL_TEXTURE_2D, texName);
        glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);

        int i,j;
        for(i=0;i<NUM_LONGITUDINAL_VERTICES-1;i++)
        {
                glBegin(GL_QUAD_STRIP);
                for(j=0;j<NUM_CIRCUMFERENTIALL_VERTICES;j++)
                {
                        glTexCoord2fv(texture_coordinates[i][j]);
                        glVertex3fv(vertices[i][j]);
                        glTexCoord2fv(texture_coordinates[i+1][j]);
                        glVertex3fv(vertices[i+1][j]);
                }
                glEnd();
        }
```

(b) [3 marks] Modify your code such that white pixels in the texture image become transparent. In order to do this you have to set the fourth coordinate (the "alpha value") of all white pixels to 0. You also have to enable blending and alpha testing which is done by inserting the lines

```
glEnable(GL_BLEND)
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glAlphaFunc(GL_GREATER, 0.9f);
glEnable(GL_ALPHA_TEST);
```

before performing the texture mapping. You don't need to understand blending and alpha testing, but if you are interested feel free to read more about it in the OpenGL programming guide. Don't forget to disable blending and alpha testing at the end of the draw method. After implementing this step your scene should look something like this:



Solution: Change the line assigning the transparency value during texture loading to

```
        if (c==255)
                texture[(k*textureWidth+l)*4+3]=(GLubyte) 0;
        else
                texture[(k*textureWidth+l)*4+3]=(GLubyte) 255;
```

Before drawing the texture add

```
        glShadeModel(GL_FLAT);
        glEnable(GL_BLEND);
        glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
        glAlphaFunc(GL_GREATER, 0.9f);
        glEnable(GL_ALPHA_TEST);
```

and after drawing the texture add

```
        glDisable(GL_ALPHA_TEST);
        glDisable(GL_BLEND);
        glDisable(GL_TEXTURE_2D);
        glShadeModel(GL_SMOOTH);
```