Graphics and Game Technology
Assignment 2
Linear transformations

1 Part 1 - Translations, Rotations and Scaling

For this part of the assignment we have prepared a framework program[1] that renders a simple animated three-dimensional scene on the screen. Start the program and watch the teapot jump over the screen. You can modify the speed of the movement by pressing the `+` and `-` keys. The teapot is moving around in the world, slowly changes shape because it is being scaled and is also rotating.

By pressing `r` you can turn off the rotation, with `t` the transformation and with `s` the scaling. Be sure that you understand the meaning of these elementary geometric transformations. You can exit the program with the `q` key.

Your assignment is to re-program these basic transformations on your own. By pressing the spacebar you can switch between the original OpenGL routines and your own. If you do this now, the teapot will freeze because "your" transformations are not implemented yet. When you have completed the assignment correctly, you should see no difference in teapot behaviour between the original functions and your own.

[1]See directory `transformations_and_gimbal_lock`.

Figure 1: The famous "Utah teapot" used in this assignment (see `http://en.wikipedia.org/wiki/Utah_teapot`).

1.1 3D OpenGL

The framework takes care of the proper initialization of the OpenGL window with all necessary options already set up. The application is written using GLUT, which is a utility library for creating OpenGL programs. GLUT provides a so-called "idle" function. GLUT calls this function whenever there are no pending events to be dispatched. In our case we tell GLUT to call the `DrawGLScene()` function in the idle state. Whenever we draw the figure once again, we count the frames we have produced. We use this frame counter to update the model-view matrix so that a fluent animation of the teapot is generated.

Take a look at the `DrawGLScene()` function. Here you can see how the different transformations are computed and applied. Notice the use of the flag `useMyTransformations`, which chooses between applying the standard OpenGL transformation calls and the ones you are going to write.

1.2 Translations, Rotations and Scaling

All affine transformations can be represented with a 4x4 matrix and OpenGL has a special matrix to store the current transformation: the so-called "model-view" matrix. Every (world) vector passed on to the OpenGL engine is transformed using the current value of this matrix (by multiplying the homogenous vector with the matrix). The resulting coordinates are interpreted as being relative to the camera axes:

$$P_{camera} = M_{model-view} * P_{world}$$

To multiply the current model-view matrix with some other matrix M we can use the following statements:

```
GLfloat M[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};
glMultMatrixf(M);
```

Here, we have made M the identity matrix, but to actually transform objects we would need to use different values. Please note that matrix M has its rows and columns swapped compared to the usual mathematical notation of matrices.

All matrix multiplications done with `glMultMatrixf()` are applied on the right-hand side of the current matrix:

$$M_{model-view} := M_{model-view} * M$$

It is also possible to save and restore values of the model-view matrix on an internal OpenGL stack with the functions `glPushMatrix()` and `glPopMatrix()`. With these functions you can temporarily save the value of the model-view matrix, apply operations that modify the matrix, draw some objects and then restore the saved value.

Because direct element-wise matrix manipulation is not very intuitive, there are three basic functions that can be combined to produce any transformation we want and that enables us to retrace what is going on. These functions are:

- `glTranslatef(x,y,z)` to move the origin relative to the old origin in the direction of the vector (x, y, z).
- `glScalef(sx,sy,sz)` to modify the scaling with respect to the origin
- `glRotatef(alpha,x,y,z)` to produce a rotation matrix and subsequently multiply it with the current model-view matrix. The rotation matrix is constructed using the rotation axis given by (x, y, z) and a rotation angle *alpha* measured in degrees. The rotation angle follows the right-hand rule, so the rotation will be counter-clockwise when viewed with the vector (x, y, z) pointing towards you.

Each of these functions modifies the current model-view matrix, just as if `glMultMatrix()` had been called with appropriate matrix.

Be sure to have read and understood the chapter "Transformation Matrices" in the book of Shirley et al. There you will find the mathematical background you need to write these functions yourself.

Start by implementing functions `myScalef` and `myTranslatef` in `transformations.c`. We will handle rotation in the next section. Check the workings of your routines against the OpenGL ones, to make sure they perform the same operation.

1.2.1 Rotation

If you got the translation and scaling matrices working correctly then it is time to tackle rotation. We might simply look up the description of `glRotatef` in the OpenGL specification (which lists the actual matrix used), but that doesn't teach you anything. So, we are going to go a different route here.

Following the explanation on the rotation matrix of Shirley, we'll create our own orthonormal basis and use that to perform the rotation. If you have the book with you it is probably a good idea to read the relevant section. If not, we reproduce a *very* condensed version of the chapter here:

3D rotations can be represented with *orthonormal* matrices. Geometrically, this means that the three rows of the matrix are the Cartesian coordinates of three mutually-orthogonal unit vectors. The columns are three, potentially different, mutually-orthogonal unit vectors. Let's write down such a matrix:

$$R_{uvw} = \begin{bmatrix} x_u & y_u & z_u \\ x_v & y_v & z_v \\ x_w & y_w & z_w \end{bmatrix}$$

Here, $\mathbf{u} = x_u\mathbf{x} + y_u\mathbf{y} + z_u\mathbf{z}$, with $\mathbf{x} = (1, 0, 0)$, etc. The orthonormal property implies that the inproduct of each of $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{w}$ with itself is 1, while the inproduct with one of the other two vectors (e.g. $\mathbf{u}$ with $\mathbf{v}$) is 0. Note that transformation matrices in OpenGL are specified as 4x4 matrices, where R_{uvw} would be the upper-left part of such a matrix.

If we wish to rotate around an arbritary vector $\mathbf{a}$, which is what `glRotatef` does, we can form an orthonormal basis $\mathbf{uvw}$ with $\mathbf{w} = \mathbf{a}$, rotate that basis to the canonical basis $\mathbf{xyz}$, rotate around the z-axis, and then rotate the canonical basis back to the $\mathbf{uvw}$ basis. In matrix form, to rotate around the w-axis by an angle ϕ, the rotation matrix $R = A * B * C$

$$= \begin{bmatrix} x_u & x_v & x_w \\ y_u & y_v & y_w \\ z_u & z_v & z_w \end{bmatrix} * \begin{bmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x_u & y_u & z_u \\ x_v & y_v & z_v \\ x_w & y_w & z_w \end{bmatrix}.$$

Instead of computing the final rotation matrix R, we can also apply the three matrices A, B and C in sequence using `glMultMatrixf`. This saves us the trouble of having to implement a matrix product routine.

The final piece of the puzzle is how to determine the vectors making up the orthonormal basis. The axis specified in a `glRotatef` call gives us only one of the three vectors ($\mathbf{w}$), so how do we get the other two?

The answer is that we can choose them ourselves, *as long as the three vectors together form an orthonormal basis.* Shirley section 2.4.6 provides a method for coming up with $\mathbf{u}$ and $\mathbf{v}$:

Given a vector $\mathbf{a}$ we want an orthonormal bases $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{w}$ such that $\mathbf{w}$ points in the same direction as $\mathbf{a}$. A robust procedure to find any of the possible bases is the following:

$$\mathbf{w} = \frac{\mathbf{a}}{||\mathbf{a}||}$$

To get $\mathbf{u}$ and $\mathbf{v}$ we need to find a vector $\mathbf{t}$ that is not colinear with $\mathbf{w}$. To do this, simply set $\mathbf{t}$ equal to $\mathbf{w}$ and change the smallest magnitude component to 1. For example, if $\mathbf{w} = (1/\sqrt{2}, -1/\sqrt{2}, 0)$, then $\mathbf{t} = (1/\sqrt{2}, -1/\sqrt{2}, 1)$. Then, $\mathbf{u}$ and $\mathbf{v}$ follow easily:

$$\mathbf{u} = \frac{\mathbf{t} \times \mathbf{w}}{||\mathbf{t} \times \mathbf{w}||}$$

$$\mathbf{v} = \mathbf{w} \times \mathbf{u}$$

Some of the mathematical operations used in this section are:

- The *inproduct* of two vectors produces a scalar value and is defined as:

$$\mathbf{v} \cdot \mathbf{w} = v_x w_x + v_y w_y + v_z w_z$$

- The length of a vector $||\mathbf{v}||$ can be computed as $\sqrt{\mathbf{v} \cdot \mathbf{v}}$.
- The *cross-product* operation ($\times$) of two vectors produces a new vector perpendicular to the two vectors and is defined as:

$$\mathbf{v} \times \mathbf{w} = (v_y w_z - v_z w_y, v_z w_x - v_x w_z, v_x w_y - v_y w_x)$$

All of the above and Shirley's book should provide you with enough information to implement `myRotatef`. The function already contains some comments and statements to help you along.

1.3 Gimbal lock

When combining transformations, especially rotations, some interesting behaviour can result. This part of the assignment deals with a phenomenon called "gimbal lock", in which one or more rotations reduce the transformation freedom of a model.

There is a second program in the framework, consisting of the source file `gimbal.c` and the executable `gimbal`. If you run the executable you will see a 3D scene consisting of a teapot together with its local axes. This teapot is transformed by three rotations, around the X, Y and Z axes. Two of the rotations can be interactively changed by using the mouse: if you press and hold the left mouse button while dragging the mouse you change the angles of rotation around X and Z (the Y rotation is fixed at zero degrees). Moving the mouse horizontally changes the Z rotation, moving it vertically changes the X rotation. By pressing "x" you can limit the influence of mouse movements to change only the X rotation, similarly for "z" and the Z rotation. You can press the "r" to reset both X and Z rotation to zero degrees AND make the mouse influence both X and Z rotation again.

The second part of the assignment consists of the following tasks:

- Add two more teapots to the scene, by editing the function `drawTeapots`. The two new teapots should be placed next to the one already drawn at distances of 5 and 10 units along the X axis, respectively. The first teapot you add should have an Y-rotation of 45 degrees, the second an Y-rotation of 90 degrees. Hint: look at the parameters of function `drawRotatedTeapot`. If done correctly you should see that all three teapots are transformed simultaneously when using the mouse. Note that the teapots should rotate around their centers.

- (Q1) Find the relevant piece of OpenGL code in which the teapot rotations are applied and answer the following question: in what order are the teapots rotated around the three axes?

- (Q2) If you play around a bit with manipulating the teapot rotations using the mouse you should see something strange in the behaviour of the rightmost teapot. Answer the following questions: What do you observe? How can this be explained?

2 Part 2 - Positioning the camera

You have seen with the previous assignment that it is possible to do quite a lot with the use of the basic model-view transformations. You might wonder why we need additional camera positioning tools at all; we could move and rotate the world in front of the fixed camera and be perfectly happy. However, that would not be the most intuitive way to tell our camera where to look. It is far more convenient if we can place the camera at a certain position, in world

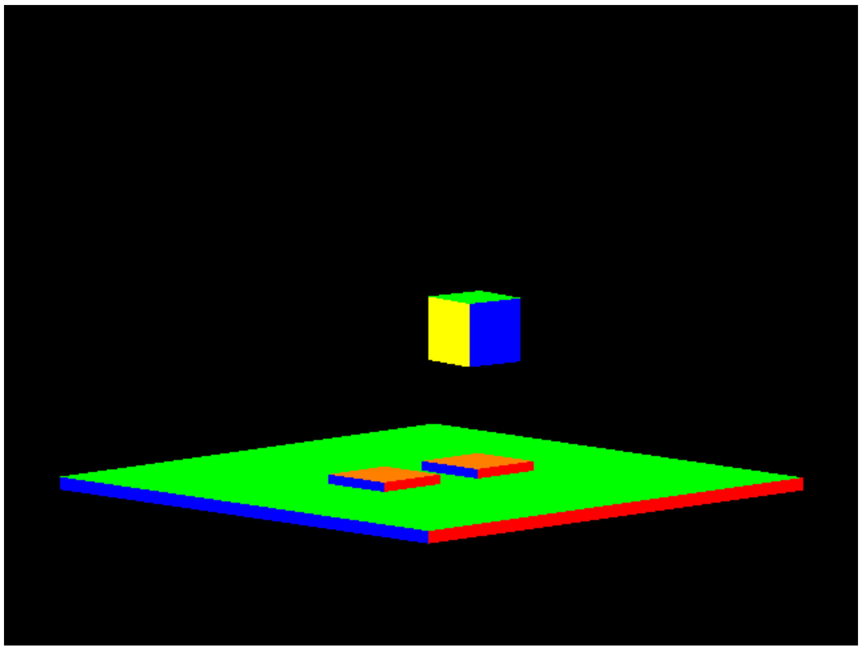

Figure 2: Screenshot of the animated scene where a camera rotates along the path of a vertical cylinder.

coordinates, so that we can look at another specific point in the world. For this, we need to be able to specify the position and orientation of the camera.

In this assignment, we will investigate the functionality of the `gluLookAt()` function and implement our own.

2.1 The framework

Build the framework[2]. When you start the program, you will see the camera rotating around a scene. The camera can be thought of to be located somewhere on the surface of a vertical cylinder around the scene. With the cursor up/down keys you can change the height of the camera, with the cursor left/right keys you can change the speed of the rotation of the camera on the cylinder. With any other key you can switch between the original `gluLookAt()` and your own.

2.2 `gluLookAt()`

Let's have a closer look on how we define the camera position.

We have to define three vectors to set up our camera: let $\overrightarrow{VRP}$[3] be the

[2]See directory `positioning_the_camera`.

[3]VRP = View Reference Point

position of the camera, $\overrightarrow{PRP}$[4] the point of interest (both specified in world coordinates) and last but not least $\overrightarrow{VUP}$[5] the vector aligning our camera upwards, also refered to as the "up-vector". Note that $\overrightarrow{PRP} - \overrightarrow{VRP} = \overrightarrow{DOP}$[6].

Suppose that we want to place the camera at a position $\overrightarrow{VRP} = (5, 5, 10)$ in the air, facing the center of our scene which we assume is at $\overrightarrow{PRP} = (0, 0, 0)$. We want the camera image to stand upright, so pass an up-vector $(0, 1, 0)$. Mathematically speaking we would assume that the up-vector would stand perpendicular on our viewing direction $\overrightarrow{DOP}$, which is not the case if we, for example, are looking down by 45 degrees. `GLU` takes care of this on its own so we don't need to bother. An up-vector like $(0, 1, 0)$ therefore is a good choice to pass to `gluLookAt()`, the function then transforms it automatically to the perpendicular $\overrightarrow{VUP}$ up-vector.

We pass these values to `gluLookAt()`, a function that modifies the current model view matrix according to our camera parameters.

```
gluLookAt( camera_x, camera_y, camera_z,
           lookat_x, lookat_y, lookat_z,
           up_x      up_y,     up_z );
```

Take a look at function `DrawGLScene()` to see whether you understand why the camera moves around the scene the way it does.

2.3 Implementing `myLookAt()`

Now you are ready to implement your own `gluLookAt()`.

First, remember what happens to the world coordinates after they enter the OpenGL pipeline:

$$\begin{pmatrix} x_{camera} \\ y_{camera} \\ z_{camera} \end{pmatrix} = M_{modelview} \begin{pmatrix} x_{world} \\ y_{world} \\ z_{world} \end{pmatrix}$$

Note that the camera transformation is actually composed of a few steps using two well known transformations:

1. A translation by $-\overrightarrow{VRP}$ (such that an object located at $(0, 0, 0)$ would appear in our example at $(-5, -5, -10)$ with respect to the camera). Use the correct OpenGL call to perform this translation.

2. A coordinate system change to camera coordinates. Construct a 4x4 matrix (see below) that performs this transformation and apply it to the model-view matrix.

We need the three vectors that represent the camera coordinate system in world coordinates, namely the x, y and z vector of the camera coordinates.

[4]PRP = Projection Reference Point
[5]VUP = View Up
[6]DOP = Direction of Projection

1. The *cz* vector is the easiest of them, it is just the vector pointing from P_{camera} to P_{lookAt}. Normalize this vector (otherwise the rotation matrix will also "stretch", which is not what we want).

2. The *up* vector points in the direction of the camera *cy* vector, but it might not be orthogonal to it. To produce a real orthogonal *cy* vector, we first compute *cx*, which is orthogonal to *up* and *cz*. Normalize *cx* into a unit vector.

3. The *cy* vector simply is orthogonal to both *cx* and *cz*.

It is very easy to construct a rotation matrix R that produces world coordinates when camera coordinates are given:

$$\begin{pmatrix} x_{world} \\ y_{world} \\ z_{world} \end{pmatrix} = R \begin{pmatrix} x_{camera} \\ y_{camera} \\ z_{camera} \end{pmatrix}$$

with

$$R = \begin{pmatrix} cx_x & cx_y & cx_z \\ cy_x & cy_y & cy_z \\ -cz_x & -cz_y & -cz_z \end{pmatrix}$$

To see how this works, suppose we want to transform the camera vector $(0, 1, 0)$ to world coordinates. After the multiplication, we obtain (cy_x, cy_y, cy_z). Unfortunately we need it exactly the other way round, because we already have world coordinates and we want to make them relative to the camera coordinate system. Therefore, we could multiply the equation above by the inverse of R. Then we get:

$$R^{-1} \begin{pmatrix} x_{world} \\ y_{world} \\ z_{world} \end{pmatrix} = R^{-1} R \begin{pmatrix} x_{camera} \\ y_{camera} \\ z_{camera} \end{pmatrix} = \begin{pmatrix} x_{camera} \\ y_{camera} \\ z_{camera} \end{pmatrix}$$

In the general case computing the inverse of a matrix is very hard. Fortunately, in the case of rotation matrices it is easy; the inverse of a rotation matrix is simply the rotation matrix transposed:

$$R^{-1} = R^T.$$

3 Part 3 - Orthogonal projection

We have now seen that all world vertices that we add to our scene are first multiplied by the modelview matrix to produce all the necessary transformations (such as translations and rotations) of the scene and/or the camera, with "camera view coordinates" of all the vertices as a result. After these transformations, we still have three-dimensional coordinates but they are now relative to the camera perspective. Next to the camera perspective, the programmer also has to define a view volume. The view volume determines what the camera sees. Everything that lies outside of the view volume will be invisible (or "clipped").

In the next processing step in the graphics pipeline, OpenGL rescales the coordinates from the view volume to fit into a unit cube where they can subsequently be clipped very easily.

In the case of the orthogonal projection, the view volume has the simple form of a three-dimensional box. The box is defined by the distance of the front clipping plane and back clipping plane, and by the distances of the left and right clipping planes as well as the top and bottom clipping planes to the orgin (that now correspond to the position of the camera).

3.1 gluOrtho

OpenGL distinguishes between two different matrices by which the world vertex coordinates are multiplied:

$$\begin{pmatrix} x_{projection} \\ y_{projection} \\ z_{projection} \end{pmatrix} = M_{projection} M_{modelview} \begin{pmatrix} x_{world} \\ y_{world} \\ z_{world} \end{pmatrix}$$

The modelview matrix stores translations and rotations, while the projection matrix is used to store the view volume transformation. To switch between the different matrix stacks, call `glMatrixMode(GL_PROJECTION)` to enable the projection matrix or `glMatrixMode(GL_MODELVIEW)` for the modelview matrix. Subsequently, you can use the functions you already know to modify the stack.

3.2 myOrtho

Create the transformation matrix that we can use to set up the orthogonal projection. Your function receives as parameters the dimensions of the view volume:

```
void myOrtho(GLdouble left,
             GLdouble right,
             GLdouble bottom,
             GLdouble top,
             GLdouble near,
             GLdouble far);
```

The resulting matrix does only some scaling and simple translation. For example, the camera coordinates $(left, top, near)$ have to be mapped to the projection coordinates $(1, 1, 1)$ while $(right, bottom, far)$ has to be mapped to a vector $(-1, -1, -1)$. This is the cube. Concerning the translation, it is easy to see that the projection origin corresponds to the camera coordinates $((left - right)/2, (top - bottom)/2, (far - near)/2)$.

4 Grading

The grading for this assignment will be performed as follows:

- Part 1 gets you max three points: one for a correctly working translation, one for a correct scaling function and one for a correct rotation function.
- For the gimbal lock part you get max two points: correctly implementing the two extra teapots as described in the text gets you one point. Answering the questions correctly is worth one point each.
- Part 2: if your function works correctly you can get up to three points.
- Part 3: if you correctly implement `myOrtho()` you will get up to two points.

Again: points may be deducted if your code is not well outlined or commented!