Programming Assignment #3

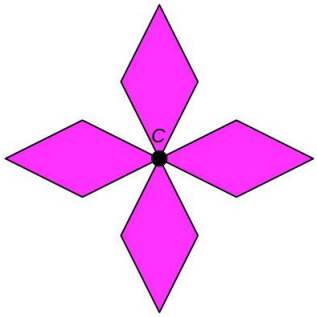
CS 200, Fall 2017

Due Wednesday, September 27

Programming Task #1

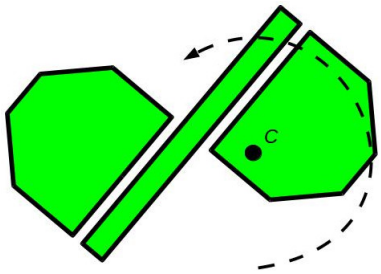
I will provide you with two files: `DPMesh.h` and `DPMesh.cpp`, which give a triangular mesh for the DigiPen logo. You are to write a program to graphically manipulate this figure, as well as the standard square mesh `SquareMesh` from the previous assignment. I have uploaded an executable of my version of the program for reference — your program should perform similar actions. Here are the requirements of this programming task.

- When the user clicks the left mouse button in the window, your program should display four copies of the `SquareMesh` figure.



  Each copy is a rotation by 90° of an adjacent copy. The point C is the location of the mouse click. You are allowed to choose the overall size of the displayed figure, but it should be small enough to fit in the window.

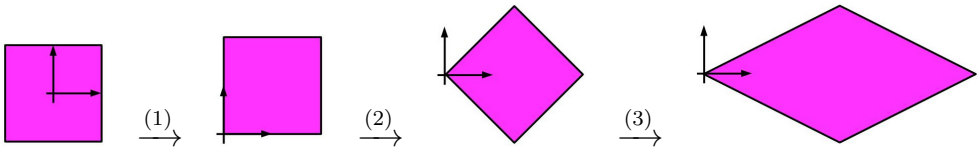
- Your program should rotate the `DPMesh` figure counterclockwise at a constant rate about the point where the user clicks the mouse; i.e., the point C described above.



Again, you may choose the size of the figure displayed on the screen, although it should fit in the window. The rotation rate should be reasonable, say taking between 1 and 10 seconds to make one rotation. The motion should be independent of the actual frame rate (time–based motion).

- The edges of the figures must all be rendered in black. However, you are free to choose the solid colors of the figures, as long as it is discernably different from black or white.

Hints. For the modeling transformation that maps the standard square to one of the four leaves of the cursor figure, you will need to modify the Scale–Rotate–Translate technique. One possible method is to perform the sequence of basic transformations indicated below



The last transformation scales the figure along the x–axis by a factor of 2.

The member function `Client::mouseclick` in the `cs200boilerplate.cpp` file gives the mouse click point (x, y) in *window coordinates*. You will need to convert this point to the corresponding point in OpenGL NDC. That is, you will need to find a transformation that maps the screen rectangle (which has an inverted y–axis) to the standard square.

Remark. Your program should behave reasonably under a window resize. Specifically, whenever the window is resized, you will need to recompute the map from the window rectangle to NDC in order to display the correct mouse click point. In this assignment, you do *not* need to be concerned about maintaining the correct aspect ratio of the displayed figures. In particular, to respond to a window resize event you should just use the OpenGL call

```
glViewport(0,0,W,H);
```

where W and H are the new window width and height. Note that this is the only place in your code where you will use an explicit OpenGL function call. All rendering should be done using the `Render` class.

For this part of the programming assignment, you should submit a single source file named `cs200_h3_1.cpp`. You may include the `Affine.h`, `DPMesh.h`, `SquareMesh.h`, `Render.h`, `SDL.h`, `glew.h`, and `gl.h` header files, as well as any *standard* C++ header file.

You are encouraged (but are not required) to use the file `cs200boilerplate.cpp` as the basis of your program. Don't forget to rename this file to `cs200_h3_1.cpp` and put your name, as well as the assignment and semester information, at the top of the file.

Task #2: inverse of an affine transformation

In the header file `Affine.h` from the previous assignment, the function `Inverse` is declared as

```
Affine inverse(const Affine& A);
```

However, you did not implement this function. For this portion of the assignment, you will implement the `Inverse` function, which returns the inverse of affine transformation A. Although A is represented by a 3×3 matrix, you may assume that this matrix is of the form

$$A = \begin{bmatrix} L_{xx} & L_{xy} & v_x \\ L_{yx} & L_{yy} & v_y \\ 0 & 0 & 1 \end{bmatrix}$$

where $\begin{pmatrix} L_{xx} & L_{xy} \\ L_{yx} & L_{yy} \end{pmatrix}$ is the matrix for the linear part of A and $\vec{v} = \langle v_x, v_y \rangle$ is the translation part; that is, $A = T_{\vec{v}} \circ L$. Therefore, the inverse of A is given by

$$A^{-1} = (T_{\vec{v}} \circ L)^{-1} = L^{-1} \circ T_{-\vec{v}}.$$

Since L is a 2×2 matrix, its inverse is given by the matrix

$$L^{-1} = \frac{1}{L_{xx}L_{yy} - L_{xy}L_{yx}} \begin{pmatrix} L_{yy} & -L_{xy} \\ -L_{yx} & L_{xx} \end{pmatrix}$$

— provided that $\det(L) = L_{xx}L_{yy} - L_{xy}L_{yx} \neq 0$. You do not have to check if the determinant of L is zero or not: this is the responsibility of the caller of the `Inverse` function.

Your submission for this part of the assignment should consist of a single file, named `Inverse.cpp`, which contains the implementation of the `Inverse` function **only**. You may only include the `Affine.h` header file, but you are free to use any of the functionality declared there.

Task #3: camera class

The header file `Camera.h` contains the following declarations.

```
class Camera {
  public:
    Camera(void);
    Camera(const Point& C, const Vector& vp, float W, float H);
    Point center(void) const;
    Vector right(void) const;
    Vector up(void) const;
    float width(void) const;
    float height(void) const;
    Camera& moveRight(float x);
    Camera& moveUp(float y);
    Camera& rotate(float t);
    Camera& zoom(float f);
  private:
    Point center_point;
    Vector right_vector, up_vector;
    float rect_width, rect_height;
};

Affine cameraToWorld(const Camera& cam);
Affine worldToCamera(const Camera& cam);
Affine cameraToNDC(const Camera& cam);
Affine NDCToCamera(const Camera& cam);
```

(the `Affine.h` header file has been included). You are to implement all of the declared functions, which are described in detail below.

`Camera()` — (default constructor) creates a camera object whose world space viewport is the standard square: the square centered at the origin with width and height of 2.

`Camera(C,vp,W,H)` — (nondefault constructor) creates a camera object whose world space viewport is the rectangle centered at C, with right vector $\vec{u}$, up vector $\vec{v}$, width W, and height H. The (unit) vector $\vec{v}$ points in the same direction as the vector `vp`, and is a 90° counterclockwise rotation of the (unit) vector $\vec{u}$.

`center()` — returns the center of the camera in world coordinates.

`right()` — returns the right vector of the camera in world coordinates.

`up()` — returns the up vector of the camera in world coordinates.

`width()` — returns the width of the camera viewport in world coordinates.

`height()` — returns the height of the camera viewport in world coordinates.

`moveRight(d)` — moves the camera d world space units in a direction parallel the the camera's right vector. A reference to the camera is returned.

`moveUp(d)` — moves the camera d world space units in a direction parallel the the camera's up vector. A reference to the camera is returned.

`rotate(t)` — rotates the camera t radians counterclockwise about the camera's center. A reference to the camera is returned.

`zoom(f)` — scales the dimensions of the camera viewport rectangle by a factor of f with respect the the camera's center. Note that the camera's aspect ratio is preserved after the call to `Zoom`. A reference to the camera is returned.

`cameraToWorld(cam)` — returns the transformation that maps the camera space coordinate system of `cam` to the world space coordinate system.

`worldToCamera(cam)` — returns the transformation that maps the world space coordinate system to the camera space coordinate system of `cam`.

`cameraToNDC(cam)` — returns the transformation that maps the camera space coordinate system of `cam` to normalized device coordinates based on the standard square. This transformation maps the viewport in camera space (the axis aligned rectangle centered at the origin with the same dimensions as the world space viewport), to the square centered at the origin with width and height of 2.

`NDCToCamera(cam)` — returns the transformation that maps the normalized device coordinates based on the standard square to the camera space coordinate system of `cam`. This transformation maps the standard square to the camera space viewport rectangle.

A programming remark. When you implement to `Camera::rotate` function, you will need to call the `rotate` function in the `Affine` package. If you write

```
Camera& Camera::rotate(float t) {
  Affine R = rotate(t);  // problem
  ...
}
```

the compiler will assume that you are attempting to make a recursive function call, and will complain that the return type of `rotate` cannot be converted to type `Affine`. To force the compiler to use the `rotate` function from the `Affine` package, you need to write

```
Affine R = ::rotate(t);  // solution
```

since this function is in the global namespace.

Your submission for this part of the assignment will consist of a single source file, which must be named `Camera.cpp`. You may include only the `Affine.h` and `Camera.h` header files.