Summary

For this project you are to write C++ code for a class for a planar, convex polygon (n-gon). You will generate a single n-gon and display it in OpenGL. You will also manipulate it in particular ways described in the project. You may use the starterkit in any way you wish, and/or use anything you have built for projects 1 and 2. You may ignore the starterkit entirely and write your own code from scratch. You may make minimal modifications to the starterkit so that it satisfies the description below. Or you may do something in between.

1 Description for All Students

You must create two files for the Ngon class, Ngon.h and Ngon.C (or Ngon.cpp, depending on your personal preference for the extension of C++ files), holding a declaration of the Ngon class (Ngon.h) and an implementation of the Ngon class (Ngon.C or Ngon.cpp). The Ngon class must have the public methods shown in Table 1. You may notice in these methods something a little unexpected. The texture coordinates in face_value and set_face_value are Vector objects, which store three components. Since texture coordinates are only two values (s, t), why use a Vector? The answer is a bit of pragmatism. We have been using the Vector class and are familiar with it. Creating a two-component object for texture coordinates can be done, but here I am allowing that we have a moment of practicality and just use a tool that we already have. So by convention, we assume that the first two values of a texture coordinate stored as a Vector are the (s, t) values.

Also in the files Ngon.h and Ngon.C (or Ngon.cpp), you must also create a class called Face that stores indices of the position, normal, and texture coordinate for each vertex. Note that Face only stores indices, NOT the actual positions, normals, and texture coordinates, which are stored in the Ngon class. Each vertex in the Face class is characterized by an index each of position, normal, and texture coordinate. The Face class must have the public methods shown in Table 2.
Table 1: Public Methods for the Ngon C++ Class

<table>
<thead>
<tr>
<th>Ngon Class Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ngon( const std::vector&lt;Vector&gt;&amp; vertices, const std::vector&lt;Vector&gt;&amp; norms,</td>
<td>Constructor that initializes the n-gon with the list of vertices, list of normals, list of texture coordinates (s,t), and the face data. Note that the size of each of these std::vector's will be different generally.</td>
</tr>
<tr>
<td>const std::vector&lt;Vector&gt;&amp; s,t, const Face&amp; f )</td>
<td></td>
</tr>
<tr>
<td>size_t ngon_size() const</td>
<td>Returns the number of vertices in the Ngon.</td>
</tr>
<tr>
<td>bool face_values( const size_t i, Vector&amp; P, Vector&amp; N, Vector&amp; tc ) const</td>
<td>For the input vertex index i, provide the vertex position P, vertex normal N, and vertex texture coordinate tc to the values in the last three items of the signature, and return true. If i is not a valid index, do not alter P, N, tc, and return false.</td>
</tr>
<tr>
<td>bool set_face_values( const size_t i, const Vector&amp; P,</td>
<td>For the input vertex index i, set the vertex position P, vertex normal N, and vertex texture coordinate tc to the values in the last three items of the signature, and return true. If i is not a valid index, do not changes the values and return false.</td>
</tr>
<tr>
<td>const Vector&amp; N, const Vector&amp; tc ) const</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Public Methods for the Face C++ Class

<table>
<thead>
<tr>
<th>Face Class Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size_t size() const</td>
<td>Returns the number of vertices in the face.</td>
</tr>
<tr>
<td>size_t vertex(const size_t i) const</td>
<td>Returns the vertex position index of the i-th vertex in the face.</td>
</tr>
<tr>
<td>size_t normal(const size_t i) const</td>
<td>Returns the vertex normal index of the i-th vertex in the face.</td>
</tr>
<tr>
<td>size_t st(const size_t i) const</td>
<td>Returns the vertex texture coordinate index of the i-th vertex in the face.</td>
</tr>
</tbody>
</table>
You are free to add public and private constructors, methods, and data as you think appropriate. In fact, you will have to add some. Note that for both the \texttt{Ngon} and \texttt{Face} classes, the number of vertices in the n-gon is arbitrary. You cannot assume that you are dealing with n-gons that are just a few vertices, for example 5 or 6 sides. As part of the grading, we will want to try to run your code for and ask for thousands of sides. It should work just as gracefully for thousands of vertices as it does for a few.

\textbf{Generating an Ngon}

Having developed the \texttt{Ngon} and \texttt{Face} classes, now you need to generate a polygon and use the classes to store it. The generation of the polygon will take place in a function \texttt{generate_symmetric_ngon}. In the discussion below, you will notice that this function appears as a private member of a \texttt{VzI ThingyDingy}-derived class, with signature

\begin{verbatim}
void generate_symmetric_ngon(int nb_sides, \ Number of vertex positions
int nb_normals, \ Number of vertex normals
int nb_texture_coordinates, \ Number of texture coordinates
const Vector& normal, \ Direction perpendicular to \ the plane of the n-gon
const Vector& center, \ Location of the center of \ the n-gon
const float radius); \ Distance from the center to \ each of the vertices
\end{verbatim}

This function (private class method) does the following tasks:

1. Generates \texttt{nb_sides} vertex positions. Each vertex position is a distance \texttt{radius} from the \texttt{center}, and the vector between each vertex position and the \texttt{center} is perpendicular to the \texttt{normal}. In the \texttt{Face} of the \texttt{Ngon}, the vertex positions must be listed in an orderly fashion so that the n-gon approximates a circular disc. The \texttt{nb_sides} will be at least 4. (Helpful Hint: There is a method \texttt{rotate} that is part of the \texttt{Vector} class).

2. Generates \texttt{nb_normals} normals at random (they must be unit vectors), and assigns them randomly to the \texttt{nb_sides} vertices.

3. Generates \texttt{nb_texture_coordinates} texture coordinates at random, and assigns them randomly to the \texttt{nb_sides} vertices. 
   SPECIAL CASE: If \texttt{nb_texture_coordinates} = 0, then it generates \texttt{nb_sides} texture coordinates and sets them to the values of the vertex positions.

4. Creates a \texttt{Face} object holding the indexing for the vertex positions and their assigned vertex normals and texture coordinates.

5. Stores all of the vertex positions, vertex normals, and texture coordinates, and the \texttt{Face} in a \texttt{Ngon}.

Make sure this function is flexible and robust. It should work whether the \texttt{nb XXX} integers are small or large, and \texttt{nb normals} and/or \texttt{nb texture coordinates} may be smaller or larger than \texttt{nb sides}. 

3
Displaying the N-gon in OpenGL - NgonThing and ngonvzl

The display of the n-gon can be done with a modification of the vzl code from Project 1. In that project a VzlThing held the triangles that you generated, and displayed them in OpenGL. For this project, the VzlThing should hold the Ngon and display it. As an example, here is the main() code for an executable called ngonvzl

```cpp
#include "VzlViewer.h"
#include "Vector.h"
#include "NgonThing.h"
using namespace vzl;

int main(int argc, char** argv)
{
    // Set up command line arguments, if any
    std::vector<std::string> args;
    for(int i=0;i<argc;i++)
    {
        args.push_back( argv[i] );
    }

    // Instantiate a viewer
    vzl::VzlViewer* viewer = vzl::CreateViewer();

    // Set up a thing
    int ngon_sides = 5;
    int ngon_normals = 4;
    int ngon_text_coord = 10;
    Vector ngon_center(0,0,0);
    Vector ngon_normal(0,1,1);
    float ngon_radius = 4.5;
    vzl::VzlThing mything = vzl::CreateNgonThing(ngon_sides,
                                               ngon_normals,
                                               ngon_text_coord,
                                               ngon_normal,
                                               ngon_center,
                                               ngon_radius);

    viewer->AddThing(mything);

    // Initialize viewer
    viewer->Init(args);
    // Run the (GLUT) main loop
    viewer->MainLoop();
}
```

The VzlThing in this case is called NgonThing. Note that the input parameter selections for generating the n-gon in `generate_symmetric_ngon` are made in the main function, passed to the VzlThing creator `vzl::CreateNgonThing`. In the grading process, we will be changing the input parameters at will, recompile and run, and want to see it still work.
Similar to other VzlThings, the NgonThing has a header file and an implementation file. The header file is shown here:

```
#include "Ngon.h"
#include "VzlThing.h"

using namespace std;
namespace vzl{

class NgonThing: public VzlThingyDingy
{
    public:

    NgonThing(int nb_sides,
              int nb_normals,
              int nb_texture_coordinates,
              const Vector& normal,
              const Vector& center,
              const float radius);

    ~NgonThing();

    /* Initialization, including GLUT initialization.
    */ Called once at the beginning. Could be used
    /* to set up things once. s
    */ It is optional to do anything with this.
    void Init( const std::vector<std::string>& args );

    // CALLBACK FUNCTIONS
    /* Cascading callback for initiating a display event
    */ This is where you code the opengl calls to display
    /* your system.
    void Display();

    // Cascading callback keyboard events
    /* This is called when you hit a key
    void Keyboard( unsigned char key, int x, int y );

    // Cascading callback for usage information
    /* If you set up actions with the Keyboard()
    /*! callback, you should include a statement
    /*! here as to what the keyboard option is.
    void Usage();

    private:

    bool display_wire;
    Ngon polygon;

    void generate_symmetric_ngon(int nb_sides,
                                  int nb_normals,
                                  int nb_texture_coordinates,
                                  const Vector& normal,
                                  const Vector& center,
                                  const float radius);
```
Color generate_vertex_color( const Vector& tc ) const;

// This function constructs the NgonThing and wraps it in a
// smart pointer called a VzlThing.
// Note that the signature of this function is the same as for the NgonThing constructor,
// and for the generate_symmetric_ngon private method in NgonThing
vzl::VzlThing CreateNgonThing(int nb_sides,
   int nb_normals,
   int nb_texture_coordinates,
   const Vector& normal,
   const Vector& center,
   const float radius);

//------------------------------------------------------------------------------

The Ngon is a private data member of NgonThing, and generate_symmetric_ngon
is a private method. Since the Ngon is needed as soon as the NgonThing exists, the best way to do that is to generate the ngon within the NgonThing constructor, like:

//--------------------- partial excerpt from NgonThing.C ------------------------
NgonThing::NgonThing(int nb_sides,
   int nb_normals,
   int nb_texture_coordinates,
   const Vector& normal,
   const Vector& center,
   const float radius) :
   VzlThingyDingy("NgonThing"), // constructor for the base class
display_wire(false) // initially display solid geometry
{
   // Build the ngon
genenerate_symmetric_ngon(nb_sides,
   nb_normals,
   nb_texture_coordinates,
   normal,
   center,
   radius);
}
//------------------------------------------------------------------------------

Recall that the Display method is called by the viewer whenever GLUT/OpenGL
decides it is time to redraw the display – in this case draw the ngon. Here we
want to use a few more capabilities of OpenGL, specifically for displaying polygons, and for variations of how color is handled. Here is the Display method for NgonThing:

//--------------------- partial excerpt from NgonThing.C ------------------------
void NgonThing::Display()
{
   if(display_wire)
   {
      // Draw polygon as wireframe
      glBegin(GL_LINE_LOOP);
      for( size_t i=0;i<polygon.ngon_size();i++ )
      {
      }
The first new tool is the type of geometry that we draw. Previously, we told OpenGL to display triangles via the function call `glBegin(GL_TRIANGLES)`, and when we wanted a wireframe of the triangle it was `glBegin(GL_LINES)`. Now, for displaying a polygon, it is `glBegin(GL_POLYGON)` and for a wireframe, `glBegin(GL_LINE_LOOP)`.

The second new tool is more control over the color that OpenGL displays. Instead of declaring a single color with `glColor3f` outside the loop that persists for every vertex added via `glVertex3f`, now we declare, inside the loop, a new color for each vertex. In this assignment, the color is generated via a private method `generate_vertex_color`, that takes in the texture coordinate for the current vertex. The particular math used in figure 1 is:
Figure 1: Example frame of a 15 sided symmetric ngon with 100 texture coordinates.

This particular set of math operations is not special, just a formula I created on the fly. You can use it, and/or explore other options. Keep in mind that the color scheme in figure 1 is based on random values of texture coordinates, so
you should not expect your colors to be the same.

The fan pattern that you see in figure 1 is due to the way OpenGL uses the colors that were assigned to the vertices sent to OpenGL. OpenGL internally tessellates the n-gon into triangles that all have a common vertex, then assigns color inside each triangle by barycenter interpolation of the colors at the vertices.

Finally, the creator for NgonThings is

```cpp
//---------------------- partial excerpt from NgonThing.C ----------------------
vzl::VzlThing vzl::CreateNgonThing(int nb_sides,
    int nb_normals,
    int nb_texture_coordinates,
    const Vector& normal,
    const Vector& center,
    const float radius)
{
    return VzlThing(new NgonThing(nb_sides, nb_normals, nb_texture_coordinates, normal, center, radius));
}
//--------------------------------------------------------------------------------
```

Modifying the N-gon While Also Displaying It

The last task in this project is to implement an ability to modify the n-gon. The modification to implement is to move each vertex position a distance $q = 0.01$ in the direction of the normal for that vertex. This should be done for all of the vertices of the n-gon when the user of ngonvzl presses the D key. Remember that the method NgonThing::Keyboard is executed by the viewer whenever a key is pressed, and the signature of NgonThing::Keyboard includes the ascii value of the key that was pressed. Similarly, if the user presses the d key, the vertex position should be displaced a distance $q = -0.01$, i.e. opposite of the effect of the D key.

Description for 6050 Students

In addition to the description above, you must also add functionality to the Ngon class that computes a determination as to whether the n-gon it holds is planar, and separately whether or not the n-gon is convex. The public methods shown in Table 3 are the signatures of the two methods you must implement. Each time displacement is applied to the n-gon (i.e. when the d or D keys are pressed), you must run these calculations and report (via cout) whether or not the n-gon is planar, and whether or not it is convex. Without displacement, the n-gon should pass both of these tests, but as displacement increases, the n-gon should lose both planarity and convexity.

Some General Guideance

- You are free to use any code I have provided in the starterkit, in the project description, and in lectures. You are free to modify it in any way. You are
Table 3: Additional Public Methods for the Ngon C++ Class, for CPSC 6050 students

<table>
<thead>
<tr>
<th>Ngon Class Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool is_planar() const</td>
<td>Returns true if the n-gon is planar, false if it is not.</td>
</tr>
<tr>
<td>bool is_convex() const</td>
<td>Returns true if the n-gon is convex, false if it is not.</td>
</tr>
</tbody>
</table>

free to completely ignore any code that has been provided or shown, or mix code as you see appropriate for the project. There is no penalty for choosing to use your own code (but there is a BIG existential penalty for using anyone else’s code - see the university guidelines for student conduct and integrity). The focus of the grading is on whether what you handin is consistent with the project description.

• There should be no “hardwired values”, i.e. numbers that are not assigned to variables. For example, in the code

```cpp
for(int i=0;i<5;i++)
{
    ...
}
```

the number 5 is a hardwired number. Don’t do this. Instead, use a descriptive variable, i.e.

```cpp
int count_of_items_in_list = 5;
for(int i=0;i<count_of_items_in_list;i++)
{
    ...
}
```

There is one exception to this: The 0 initialization of the loop variable i is very common and acceptable as long as it has the very simple interpretation of initializing a counter. If its meaning is deeper than that, then it should be defined in an appropriately named variable also.

• Remember that the code must be compilable and executable on a School of Computing Linux computer. It is not the grader’s job to ferret out the obscure and complex set of steps that you have set up to compile and run. Use a makefile to handle the process, and make it trivially easy to understand. The Makefile in the starterkit satisfies those requirements.

• Part of your grade is based on the cleanliness and clarity of your code. Think of you code as a kind of essay, and this is an essay assignment. If you turn in an essay in the English department that reads like
Times bad. Good sometimes. I don’t have pride in work so I’m just filling quickly not paying attention

You will probably be graded accordingly. Same here. Write code that is meant to be quickly understood by others.

- The starterkit gives you a lot of functionality that you can exploit, but when you change something you will generally need to make sure that you understand what the thing is doing and what your changes do.

Upload to handin

Create a folder called <username>. Put all of the following files into that folder. There should be no subfolders.

Zip compress the folder into a single zip file, named <username>.zip. Upload this file to the handin system. The course webpage has more guidance and caveats if you need them.