CCSI 3161 Project – Flight Simulator

Objectives: To develop a significant OpenGL animation application.

Due date: Dec 3rd, 11:59pm.
No late submission will be accepted since the grades need to be submitted.

Hand in: Electronic submission of entire Visual C project (source, compiled, project files, etc.) using dal.ca/brightspace. Please zip up your whole project directory and submit the zip file.

Note, before beginning your project, please read:

1. The coding style guideline.
2. The policy on plagiarism.

General Comments:

If you do decide to code on your own machine, you will need to port, re-compile and test your code on the Faculty's Windows machines prior to handing it in (if we can't compile it and run it, we can't mark it). *Marks deducted otherwise!*

Although we use platform independent libraries, please be warned that it will still require some amount of effort to port your code.

In this project you DO have to handle a window re-shape. You should also use depth buffering and double buffering for your project.

Part A: Flight Viewer [18 marks total]:

(i) Basic Scene [3]

In this part you will create a frame of reference.

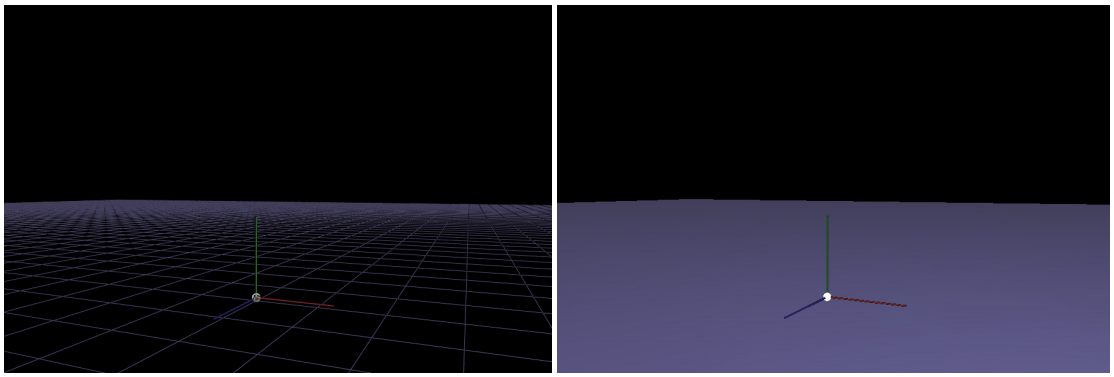
Draw 3 lines that represents the positive unit vectors in the X, Y and Z directions, with line width = 5. Make the lines colored red, green and blue, respectively. Draw a small white ball at the origin.

Construct a grid of GL_QUADS on the X-Z plane centered at the origin. The size of the grid should be set as a global C constant (which you could change later). Set the grid size to be 100x100 vertices.

Use perspective projection. Handle a window re-size properly.

The user can toggle between wire frame and solid rendering by pressing the 'w' key. The user can switch between full screen mode and windowed mode by pressing the 'f' key. Pressing 'q' quits from the application.

Your scene should look like this (wire frame and solid):



Hints:

1. Use glPolygonMode to switch between wire frame and solid rendering.
2. Use glutFullScreen and glutReshapeWindow/glutPositionWindow to toggle between full screen and windowed modes.
3. You can use gluSphere to make the sphere.

(ii) Twin Engine Plane [6]

Read in the file ‘cessna.txt’. It contains vertices that look like:

```
v 0.242636 0.170825 -0.0272018
v 0.269521 0.170825 -0.0192831
v 0.269521 0.170825 0.195895
…
```

Each ‘v’ represents a vertex’s 3D coordinates. After the vertices, it contains normals that look like:

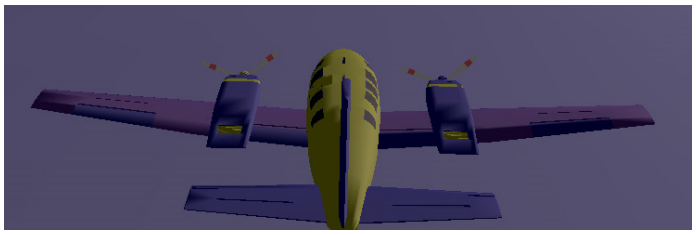
```
n 0.242636 0.170825 -0.0272018
n 0.269521 0.170825 -0.0192831
n 0.269521 0.170825 0.195895
…
```

There is one normal for each vertex. And, at the end of the file are sub-objects that look like:

```
g cargo
f 1927  1928  1929  1930
f 1931  1927  1930  1932
…
g cargo2
f 1933  1934  1935  1936
f 1937  1938  1939  1940
f 1941  1944  1935  1934
…
```

The lines that begin with ‘g’ indicate a new sub-object. The faces that form for sub-objects begin with ‘f’. The faces are polygons with a variable number of vertices, they are not just triangles. The numbers refer to the vertices of the triangle (ordered from the start of the text file).

By breaking the object up into sub-objects, it allows us to color the plane properly:



You will color the sub-objects as follows:

[0..3]	= yellow
[4..5]	= black
[6]	= light purple
[7]	= blue
[8..13]	= yellow
[14..25]	= blue
[26..32]	= yellow

Place the plane just below and in front of the camera at an appropriate scale.

There is also a separate file for the propellers called 'propeller.txt'. It contains two sub-objects which will also need separate colors. You will read in the propeller in a similar fashion to the plane. The propellers need to be continuously rotating, and positioned correctly in front of the plane.

For the model files see: www.cs.dal.ca/~sbrooks/csci3161/project/index.html

Hints:

1. Move the propeller to the origin before rotating it.

(iii) Camera Control [5]

Allow the user to control the direction of the camera/airplane movement with keyboard keys and the mouse. The page-up and page-down keys increase and decrease the speed of the camera/airplane movement in the direction that the camera is currently pointing:

GLUT_KEY_PAGE_UP	:	increase speed
GLUT_KEY_PAGE_DOWN	:	decrease speed

But, limit the speed so that it is always greater than zero. The up and down arrow keys control the vertical position of the camera/airplane:

GLUT_KEY_ UP	:	moves the camera up
GLUT_KEY_ DOWN	:	moves the camera down

Moving the camera/airplane up and down does not alter the direction that the camera/airplane is pointing. The user can hold down a key to continuously move in a direction, so you might want to use glutKeyboardUpFunc (which tells you when the user has released a key) as well as glutKeyboardFunc (which tells you when a key has been pressed).

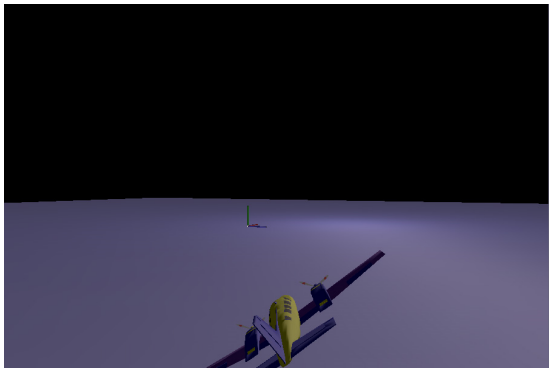
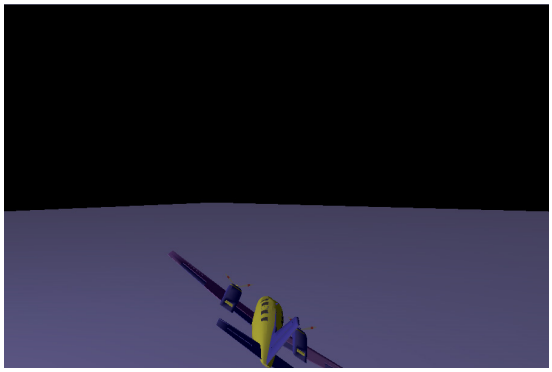
Moving left and right is significantly different. It is controlled by the mouse and actually changes the direction that camera/airplane moves (not just its position). This means that you can turn around and go in the opposite direction. When the mouse is to the right of center or the left of center:

mouse to the right : rotates the direction the camera is pointing/moving to right
mouse to the left : rotates the direction the camera is pointing/moving to left

The more the mouse is to the left or the right of center, the greater is the change of camera direction. When the mouse is in the center, there is no change of direction, but you will still be moving forward at some speed. The mouse button does not need to be down for this to work.

Use perspective projection with gluPerspective and gluLookAt to position the camera.

You DO need to tilt the airplane when the mouse is to the left or right:

Hints:

1. Use glutSpecialFunc to register a callback function for the special keys.
2. Use glutPassiveMotionFunc to handle mouse movement.

(iv) Lighting [4]

Use openGL's lighting model. Use at least one directional light source, with appropriate ambient, specular and diffuse levels. Position the light source so that it is similar to sunlight.

You need to set appropriate shininess, ambient, specular and diffuse material properties for each object. You will have set the normal for each vertex in the plane/propeller and the grid.

Hints:

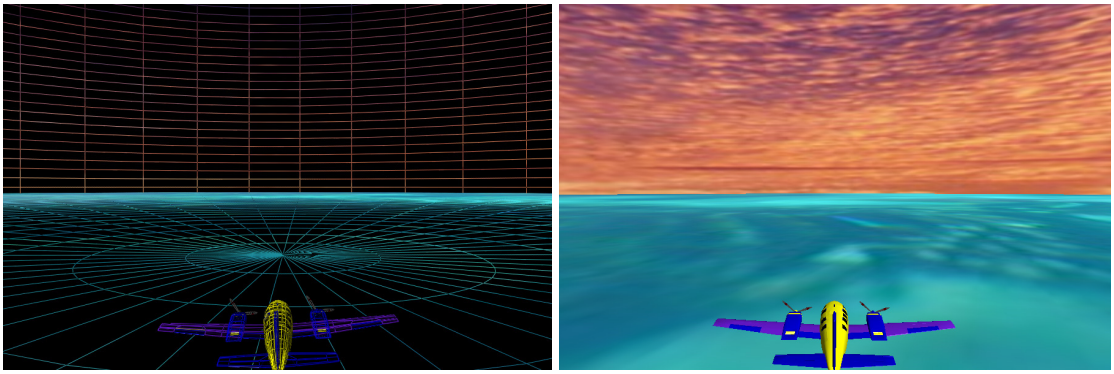
1. The normal for the grid is all the same and is really easy to 'compute'.
2. Set the light position after calling gluLookAt.

Part B: Sea, Sky and Land [16 marks total]:

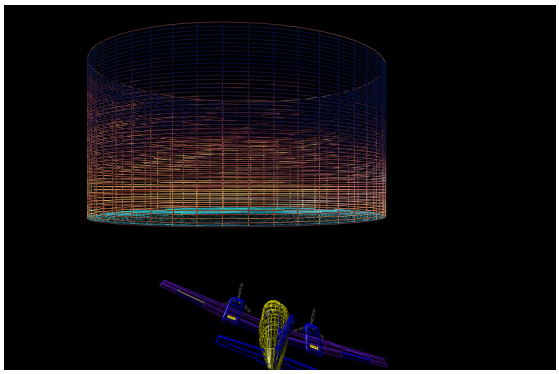
(v) Sea & Sky [6]

In this section you will replace the grid floor (and origin lines) from Part A with a texture mapped disk, and a texture mapped cylinder. The disk lies on the ground and is texture mapped with a water image. The cylinder stands up right and encloses the 'world'. The cylinder is texture mapped with a sky image.

Your scene should look like this (wire frame and solid):



This is what it looks like from outside:



When the user presses the 's' key, your program will toggle between the grid (from Part A) and the sea/sky described here.

The disk should be slightly bigger than the cylinder. The cylinder should sit slightly lower than the disk. This will prevent any gaps. Make sure the cylinder is large enough so that you cannot see the top of it but not so high that the texture is overly stretched vertically.

You can either write your own functions for constructing the cylinder and the disk, or you can use the OpenGL functions that construct them for you. If you use the OpenGL functions, you will need: gluNewQuadric, gluQuadricTexture, gluCylinder and gluDisk.

Make sure that the textures appear in a similar fashion to the example program that is given. The texture on the cylinder should wrap around like a label on a soup can. The disk is textured as if a round cookie cutter was used on the water image.

Use a high emissive term for the lighting of both the sky and the sea so that they are both evenly lit.

Turn texture mapping on only for texture mapped objects. Use texture mipmapping.

For the texture images see: www.cs.dal.ca/~sbrooks/csci3161/project/index.html .

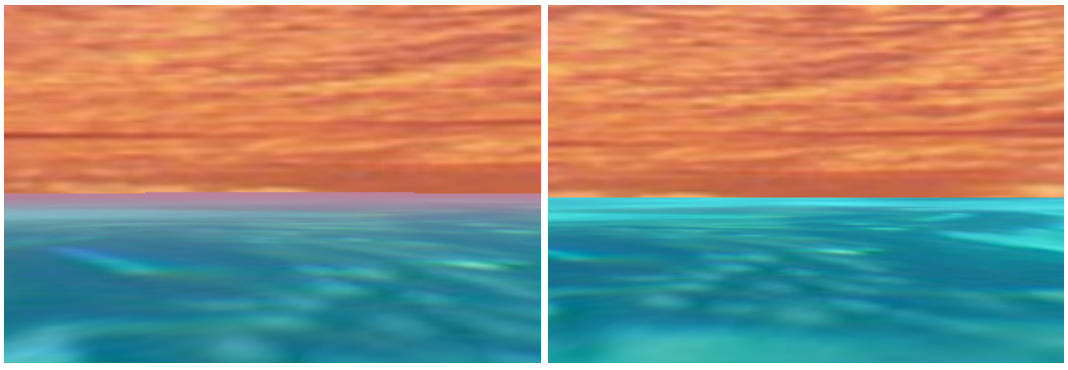
You can also (optionally) find more free textures online. But, if you are using PPM image files, you will need to convert these textures to the PPM format. You can use Paint Shop Pro to do this: www.jasc.com/products/paintshoppro/ .

(vi) Fog [2]

Add fog to the sea. Use the built-in OpenGL functions to achieve this. Make the fog a very transparent pink so that the sea blends in slightly with the sky at a distance. Use an exponential fog function. Give the fog a very low density. Make it look like the example program.

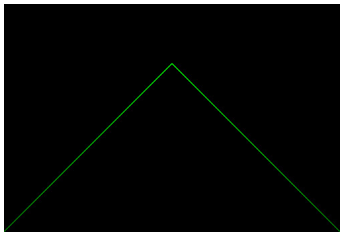
Fog should only affect the sea. Turn the fog on before drawing the sea and off afterwards. The fog is toggled on and off with the 'b' key.

Your scene should look like this (with and without fog):

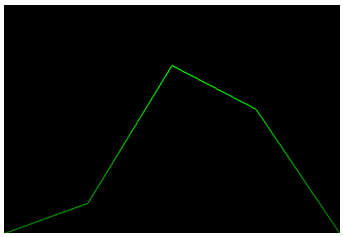


(vii) Land [8]

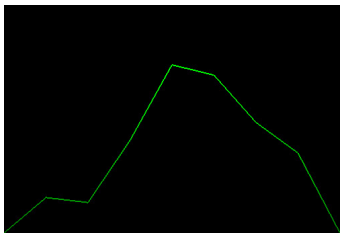
Create islands in the sea. The construction of the islands follows a simple algorithm which we will first describe in 2D. This will generate the silhouette of an island mountain. Start with two lines that form a pyramid:



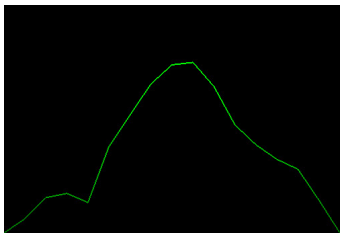
Then, at their mid-points, add a random vertical amount:

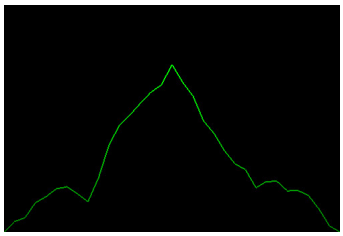


Then repeat this at the next level. At the mid-points of these four lines, add a random vertical amount (but use a smaller random amount) :

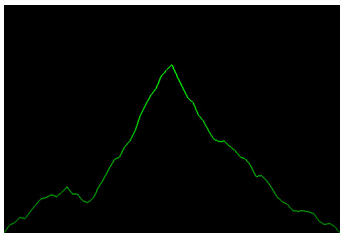


Keep repeating (with smaller random amounts each time) :





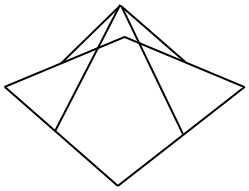
Until you get:



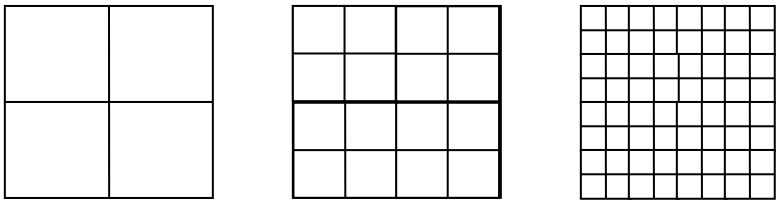
which is a nice mountain silhouette. This can then be scaled and stretched with the usual transformations.

But, how much less of a random amount do you add at each iteration? You need to divide the random amount by 2 at each level. So level 2 will have ½ the random amount added to the vertices, level 3 will have ¼ the amount, level 4 will have 1/8 the amount, etc.

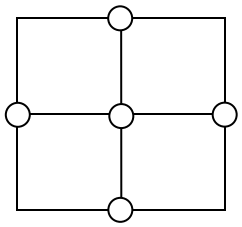
The algorithm for a 3D mountain is very similar. Start with a pyramid shape:



Then generate smaller and smaller polygons at each iteration (top view shown):



At each iteration (*except the first*), add a random vertical amount to the 'midpoint' vertices. The 'midpoints' are drawn as circles below:

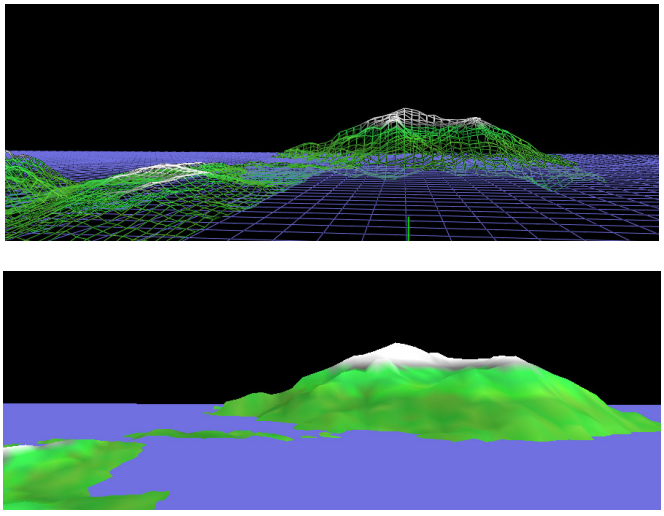


Once again, you need to divide the random amount by 2 at each level. So level 2 will have ½ the random amount added to the vertices, level 3 will have ¼ the amount, level 4 will have 1/8 the amount, etc.

You should generate the mountain only once and store it in an array, don't construct it each time you draw it. The mountain should be constructed with 5 or 6 levels, whatever looks best. The mesh resolution for the islands should be a changeable constant. Set it to a size of at least 40x40. The mountains are toggled on and off with the 'm' key.

You will need to color the mountain as well. Make the color at each vertex be a function of the height (y value) of that vertex. Make it go from grey to white near the top, and dark green to light green below the white. Also add some randomness so that it doesn't look too uniform.

Your islands should look like this (wire frame and solid):

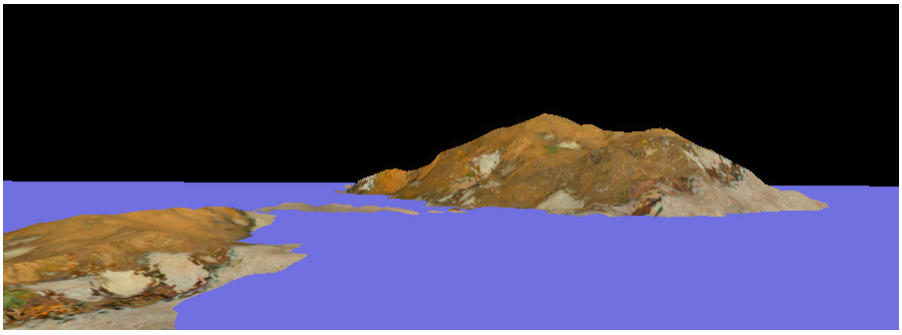


Use lighting for the mountains with some diffuse, ambient, specular highlight, but not as shinny as the plane. No emissive lighting.

Use smooth shading. For this you will need to compute the normals for each vertex (not just for each polygon).

You must also be able to switch to a texture for the mountain, instead of the coloring function. The mountain texturing is toggled on and off with the 't' key.

Your islands should look like this in textured mode:



For the texture images see: www.cs.dal.ca/~sbrooks/csci3161/project/index.html .

You must make at least 3 mountains, with a different randomness. You can try explicitly setting the value you pass to srand if you want to keep particular mountain shapes. Each should be scaled differently in X, Y, Z. This you can do with glScale. You can use the same texture and coloring function for all three mountains.

Hints:

1. If your textures look dark, make sure that glColor() is set to white, because the texture colors are multiplied by the current color.

Additional Enhancements [6]:

Add some additional feature(s) to your project. It will have to be something *very* special for full marks. **Add a readme file that states exactly what your inventions are and how to use them.**

Keyboard controls [0]

Use printf to dump out a listing of all the keyboard key functions that you are supporting. Get the keyboard controls working.