CSCI 4611: Programming Interactive Graphics and Games

Assignment 3: Earthquake

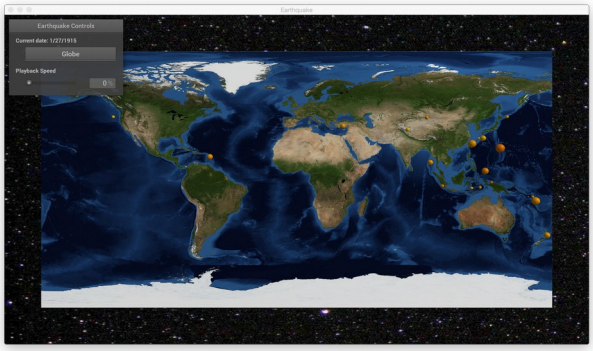
Handed out: Wed, February 28

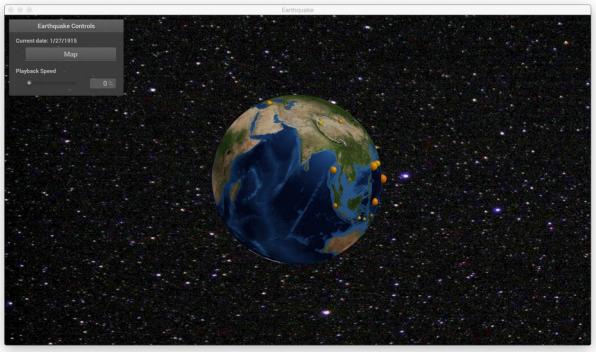
Due: Tue, March 20

Introduction

For this assignment, you'll be working with data from NASA and the USGS to visualize on a globe the locations where earthquakes happened between 1905 and 2007. Visualizations incorporating geospatial data are used and analyzed in many different contexts, including navigating a city (as seen in GPS devices), commercial development, and setting governmental policy. This area also receives a significant amount of research attention. For example, Prof. Vipin Kumar and others at the University of Minnesota are working on visualizing and understanding global warming datasets as part of an NSF project (you can find more information at http://climatechange.cs.umn.edu/).

Your application will be able to morph between two complementary views of the data, a 2D map view and a 3D globe view, as shown below.





The earthquake dataset you'll be using includes 13,540 different earthquakes. The Earth texture dataset from NASA is available in resolutions down to 500m/pixel (although getting this to display on your graphics card is well beyond the scope of this assignment).

In this assignment, you will learn to:

- Visualize real-world geographical data on a 3D textured globe.
- Apply textures to 3D objects.
- Algorithmically create a deforming 3D mesh and display it using vertex buffers.
- Define normal vectors and texture coordinates for a sphere.
- Convert from spherical coordinates (latitude and longitude) to 3D Cartesian coordinates.

Earth and Earthquake Data

We have included multiple scaled-down versions of this Earth texture with the code distributed on the website, since you need a fairly powerful computer to render even the lowest-quality image from the NASA page. In order of decreasing quality, the following images are provided:

- earth.png: Full-resolution (8 km/px, 5400×2700) image from NASA page
- earth-2k.png: 2048×1024 scaled-down version of image
- earth-1k.png: 1024×512 version of image
- earth-512.png: 512×256 version of image
- earth-256.png: 256×256 version of image

Almost any graphics card should be able to use the 256 version of the Earth texture. If your computer isn't able to use this texture, you will need to use the computers available in the various CSE Labs. The Earth textures are stored in a equirectangular projection, which simply means that the *x* coordinate corresponds directly to longitude and *y* directly to latitude.

The earthquake dataset contains information about the earthquake's magnitude (a measure of how severe the earthquake is) and its longitude and latitude. You'll be required to display this at the correct locations while animating through time. More information on the earthquakes is available in the data file, and if you are interested, you can try to figure out ways to integrate additional data variables into your visualization.

Requirements and Grading Rubric

You will be required to write the code that displays the Earth on screen and animates it between flat rectangular map and a three-dimensional globe. You will also need to write the code to display the earthquakes on the Earth. The starter code defines a single light

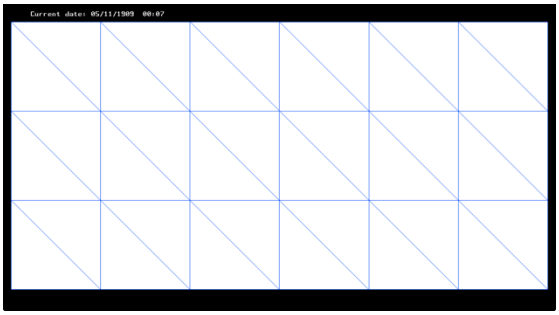
source at position (10,10,10), so you will see some brighter lighting on the top-right portion of the globe if your normals are defined correctly. (Note: The lighting will show up best on the landmasses. Our lighting model is multiplicative, so texture colors close to black (0,0,0) zero out the lighting.)

Work in the "C" range will:

Use the MinGfx::Mesh class to draw a rectangle subdivided into multiple triangles, representing a flat map of the Earth. The Flat Map Mesh should:

- Lie in the *xy* plane.
- Have x values ranging from $-\pi$ to π.
- Have y values ranging from $-\pi/2$ to $\pi/2$.
- Be divided into *nslices* divisions horizontally and *nstacks* divisions vertically. (You will find these constants in the code. Note that this will produce (*slices*+1)×(*stacks*+1) vertices and 2×*slices*×*stacks* triangles.)

The best way to go about this is to start with a small example and then build up from there. We recommend:



- On the right is an example with 6 slices and 3 stacks. Study how vertices are connected into triangles here.
- Work out the indices on paper by hand for a similar small example, say *nslices* = 4, *nstacks* = 2. Draw a picture, numbering the vertices and develop your own indices array based on your picture.
- When you have the concept down, recreate your picture in code inside the `Earth::Init()` function. Call `earth_mesh_.SetVertices()` to set your vertex array and `earth_mesh_.SetIndices()` to set your index array.
- To check your work, try clicking on the "Toggle Debug Mode" button in the app. If you have a large mesh, this will considerably slow down your rendering, but it is very useful when debugging – it draws the edges of each triangle in the mesh so that you can see your work.
- When your simple example is working, transition to a mesh that will work well for the earth. Make sure your x and y values follow the ranges mentioned above.
- At this point, we also recommend that you clean up your code a bit. At some point in your code, you will have needed to covert a latitude and longitude to a

(x,y,z) point. If you have not already, move this code to the function called Earth::LatLongToPlane(). This will make it easier for you to use the same code later to find the correct positions for earthquakes.

Work in the "B" range will:

Apply a texture of the Earth to the rectangle so that it looks like the original image.

- To do this you will need to call earth_mesh_.SetTexCoords(0, ...). Remember to use 0 for the first argument. We will only be applying a single texture to the earth mesh, so we'll only be using texture unit 0.

Display on the Earth all the earthquakes that have happened within the past one year of the current visualization time.

- Use Earth::LatLongToPlane() to obtain the earthquake positions such that they match the textures (i.e., an earthquake occurring in California must be displayed in the same location as the California of the Earth texture). One way to tell if your mapping is correct is if you see a lot of earthquakes along the "Ring of Fire" in the Pacific Ocean off the coast of Asia.
- You may use the QuickShapes functions to draw the earthquake markers or come up with your own custom geometry. Earthquake sizes and colors should be based on the earthquake data in some meaningful way. Explain and justify the mapping you choose in your README.

Work in the "A" range will:

Change the vertex positions and normals to draw the Earth as a sphere instead of a rectangle.

- If you fill in Earth::LatLongToSphere() and use this rather than the Plane version when creating your mesh, you should end up with a good sphere geometry. The mesh connectivity (i.e., the indices that define the triangles) and texture coordinates do not need to change when from moving from a plane to a sphere, but the vertex locations do.
- You will need to convert latitude and longitude into the three-dimensional Cartesian coordinates of the corresponding point on the sphere, using the formulas

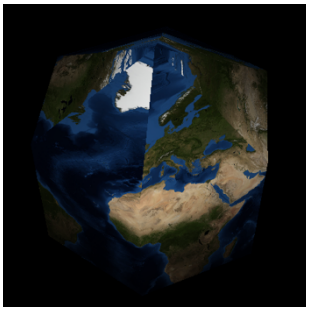
  $x = \cos(lat)\ \sin(lon)$,

  $y = \sin(lat)$,

$z = \cos(lat)\cos(lon)$.

Be careful that the input latitude and longitude are in degrees, not radians.

- Since you are now rendering a sphere, you'll need to add some normals to get the lighting on the globe to look good. The normals array should be exactly the same length as the vertices array, and you can set them using `earth_mesh_.SetNormals()`.
- The texture is permitted to look slightly "cut off" *only* at the top and bottom stack of the Earth mesh. The image to the right shows what this looks like when *slices* = 6 and *stacks* = 3. (See below for why this happens.) Increasing the number of stacks and slices will make this problem harder to see.
- The earthquakes must appear on the sphere in the correct geographical locations (though they may not lie exactly on the mesh if it has too few slices and stacks).



Smoothly transform the mesh between the flat map and the globe based on user input. The program should start with the Earth displayed as a map. When the user presses the globe button, it should morph into a sphere. If the user presses the button again, it should go back.

- To get started on this, you will want to store the parts of the mesh that change between the two representations (vertices and normals) as member variables within your Earth class rather than local variables within the Init() function. This way, you will be able to refer to them later in your program.
- Once you have these values stored, try updating the mesh when the Globe button is pressed. `QuakeApp::OnGlobeBtnPressed()` is will be called whener the user presses this button. To update the mesh, you just need to call earth_mesh_.SetVerices() and earth_mesh_.SetNormals() again with the appropriate arrays and then call earth_mesh_.UpdateGPUMemory().
- Once you have an abrupt transition working, try creating a smooth morph. Remember that UpdateSimulation() is the callback to use for any sort of physical simulation or animation. Inside that function, the strategy you should use is to interpolate between the vertices for the map and for the globe based upon the elapsed time since you start the morph. Check out the Vector3::Lerp() and Point3::Lerp() functions for help with this. (Technically, the correct normal of the interpolated shape is not simply the interpolation of the normals, but we may pretend it is for this assignment.)

If you reduce the value of *slices* and *stacks* and then watch the mesh structure as it interpolates between a rectangle and a sphere, you should be able to see why the texture appears to be cut off near the poles.

Useful Math

Here are a few mathematical operations that are very common in graphics and may be useful for this assignment:

- Linear interpolation: One way to blend smoothly between two values x and y (which could be reals, or vectors, or matrices, etc.) is to define a function whose output varies continuously from x to y as a scalar parameter a goes from 0 to 1. This function is traditionally abbreviated "lerp":
  $$\text{lerp}(x, y, a) = x + a(y - x).$$
  Thus, for example, $\text{lerp}(x, y, 0) = x$, $\text{lerp}(x, y, 1) = y$, and $\text{lerp}(x, y, ½) = (x + y)/2$. The MinGfx library provides Lerp() functions for several of the built-in types, include Vector3, Point3, and Color that could be useful to you in this assignment.
- Clamping: A concise way to constrain a value to lie in a specified interval $[a, b]$ is to define a "clamp" function $\text{clamp}(x, a, b)$ which returns a if $x \leq a$, returns b if $x \geq b$, and returns x otherwise. MinGfx similarly provides a Clamp function within the GfxMath class.
- Rescaling: Suppose you have a value x in the range $[xmin, xmax]$, and you want to find the corresponding value in $[ymin, ymax]$. Observe that $x - xmin$ lies in $[0, xmax - xmin]$, and $(x - xmin)/(xmax - xmin)$ lies in $[0, 1]$, so the desired value is
  $$y = ymin + (ymax - ymin)(x - xmin)/(xmax - xmin).$$

Above and Beyond

All the assignments in the course will include great opportunities for students to go beyond the requirements of the assignment and do cool extra work. We don't offer any extra credit for this work — if you're going beyond the assignment, then chances are you are already kicking butt in the class. However, we do offer a chance to show off… While grading the assignments the TAs will identify the best 4 or 5 examples of people doing cool stuff with computer graphics. After each assignment, the selected students will get a chance to demonstrate their programs to the class!

There are great opportunities for extra work in this assignment. For example, the source website for the Earth texture (see Data Credits below) has images for each month of the

year. You could animate between the textures based upon the current time of year. We have also included height images that contain the elevation and bathymetry data for the Earth, with sea level being 50% gray, and black and white indicating 8 km below and above sea level respectively. You could use this image to visualize the shape of the Earth's surface by loading it in as a Texture2D and using its pixel data to displace mesh vertices along their normals. You'll need to use a very high-res mesh to make this look good. You can also adjust the model matrix for the earth to make it spin around and even apply an axial tilt. Be creative!

Support Code

Before getting started, you will need to update, build, and re-install the MinGfx library inside your repo in order to get the new Lerp() functions and a few bug fixes that we have added since the last assignment. Below is a URL documenting how to update MinGfx for your repositories (under the section "Updating MinGfx"):

- https://github.umn.edu/umn-csci-4611-S18/student-repo/blob/master/README.md#updating-mingfx

Once you have your repo ready to go, you should download the support code for assignment 3, which is posted on canvas as a zip file. This will be the same process as what you did for assignment #2. Download the support code and place it in a directory named a3-earthquake within your dev directory. Then do:

```
cd dev/a3- earthquake

mkdir build

cd build

cmake-gui ..
```

"Configure" then "Generate" then "Open Project"

Handing It In

To hand in your code, check in to the master branch of your Github repository by the deadline. Any commits past the deadline will be assessed according to the late penalties described in the syllabus.

Data Credits

The earthquake data comes from http://earthquake.usgs.gov/data/centennial/. As per http://www.usgs.gov/laws/info_policies.html, this data is in the public domain.
Credit: U.S. Geological Survey,
Department of the Interior/USGS

The Earth texture comes from http://visibleearth.nasa.gov/view_cat.php?categoryID=1484. As per http://visibleearth.nasa.gov/useterms.php, this data is freely available for re-use.
Credit: NASA Earth Observatory