3D Rendering

Introduction

I have created a "readme" file, which documents how to run the application, the application usage, and the file structure in the "src" folder. Reading this is not necessary to run the application or understand the project, but it does contain useful information and clarification which may make your life easier.

Implemented Features

- Calculating face meshes from the mesh, averages, weights and offsets provided in the data
- Rendering face meshes with flat shading
- Interpolating between three faces to create a new face
- Loading additional faces from the data, and replacing a reference face with one of these faces
  - A preview for the additional face
  - Preserving previous contributions when a reference face is replaced
- Flat Phong shading (along with the default lambertian)
  - User controls for the Phong parameters
- User controls for the light direction, and light colour

Library Used

This time, I decided to use the p5 js library, instead of using raw javascript like last time. p5 js comes with some very useful features that build on top of the HTML5 canvas. It comes with a 3D rendering mode, but I only used 2D features as using that would be against the spirit of this practical.

This is what I used from the library:

- The runtime framework, consisting of `preload`, `setup` and `draw` functions, as well as the `mouseClicked` function
  - `preload` is called when the page loads, and is there to help with loading data
  - `setup` is called once all asynchronously loading in `preload` is finished
  - `draw` is called every frame, once setup is finished
  - `mouseClicked` handles mouse clicks
- `Vector` object, and the accompanying vector maths functions (including dot and cross products)
- `triangle` function, which draws a 2D triangle on the screen
- `loadTable` function to load the csv data - this is done asynchronously
- `p5.Element` object and accompanying functions, which help in constructing and placing html elements, and using properties of those objects in the code

Basic Spec

Loading & Preprocessing the Data

I wrapped the `loadTable` function with some helper functions, so I can load files using just the file name. These functions are called in `preload` to load the initial data - `mesh`, `sh_ev`, `tx_ev` and `tx_n`, `sh_n` for 0 through 3 (the average face, and the first 3 offsets). Since this is being called in `preload`, the load is guaranteed to finish before `setup` (and therefore `draw`) are called. Within `setup`, I then call the `processTables()` function, which converts all the tables loaded into the appropriate data format. Tables are replaced with arrays, the coordinates are converted into p5 Vectors, and colours are converted to arrays. The use of Vectors for coordinates should be clear. Arrays are used for colours to allow for easy manipulation of colours through the `Array.prototype.map()` and similar functions.

Rendering the Mesh

Since performing the calculations and rendering the mesh take a significant amount of time, I only redraw elements when necessary. To do so, I have an object, `updated`, which has attributes used to communicate with the draw function that it should redraw a face(s). I also switched the renderer from the default bitmap canvas to webGL, which significantly sped up the rendering process (though the calculations beforehand do stil take some time). I could have simply not used the `draw`

function, and just call the `drawFace` function from within whichever function causes it to be redrawn, but collecting all calls to the `drawFace` function within one function makes the code cleaner.

I use p5's `triangle` to render the mesh with an orthographic projection. To do this, I calculate the coordinates of the tris in the mesh by using the `map` function to combine the average, offsets and weight, and calculate the vertex colours in a similar way. Then, the `map` function is used on the mesh to create the array of tri objects for that face. A tri object contains the average z coordinate, used for z-sorting, the colour, calculated from the renderer function and the average vertex colour, and the vertex coordinates, which have been added to the centre coordinates for this face. The centre coordinates are added only now, so that each face has the same lighting.

Lambertian Shading

Thanks to the built-in Vector type and corresponding functions from p5 js, I didn't have to write my own functions for vector maths again. Thanks to this, calculating the $cos\theta$ for the faces was fairly easy to implement. I simply used the `.sub` function to create two directional vectors from the three points making up a tri, then used the `.cross` function and `.normalize` function to get the unit normal vector for the plane. I then use the `.dot` function with the light vector (which is already normalized) to get the $cos\theta$. This can then easily be used with the average colour of the vertices to implement basic flat shading.

Synthesising the new face

Checking bounds of the triangle

Once the mouse is clicked, its posision must first be checked with the bounds of the triangle so that a new face synthesis isn't triggered when interacting with other elements on the screen. To check this, I check which side of each line the mouse position is on, and make sure these are all the same or 0. One problem faced when implementing this was that the mouse coordinates are on a different system than the canvas, with the mouse coordinates centred on the top-left of the screen, and the canvas coordinates centred on the centre of the canvas, but this is solved by offsetting with half the canvas width and height.

Calculating contributions

First, p5's `dist` function is used to get the distance to each corner of the triangle. Since we want a smaller distance to mean a larger contribution, we take $\frac{1}{dist}$, and then normalise these value to sum to 1. The normalisation means the synthesised offset is never larger than the largest offset among the reference faces.

Calculating the offsets for the new face

Now that we know the contributions, we simply iterate through the three reference faces, and multiply the corresponding offsets and weights by the contributions, summing the results to get the offsets and weights for the synthesised face. The resulting offsets and weights are stored in variables, so that the same function to draw the reference faces can be used to draw the synthesised face.

Advanced Spec

User Input

I decided to have user input be controlled through a dropdown menu each for selecting the new face, and for selecting which reference to replace, and a button each for showing a preview and replacing the face. I have the preview be controlled by a button, and not automatically updated upon selecting a new face, as it takes some time to load and render the new face, which would make the user experience poor. The UI elements are HTML elements created through p5, and are positioned within a container div through some css. The container div is positioned absolutely, so that it can display on top of the canvas. This allows me to easily position UI elements without having to create multiple canvases.

Loading extra data

Loading all the data when starting the app would take a long time. Instead, data for the new face is loaded when the "show preview" or "replace" buttons are pressed, at which point the data is loaded through the wrapper function for loadTable, which is also passed a callback function. The callback function is called on the loaded table once loading is finished, and is used to convert the table into the appropriate data format, and assign it to the corresponding index in the `tx_n` or `sh_n` array. Arrays in javascript are usually implemented as sparse, or as a hashmap, so indexes in the array can be treated like keys.

Since data is now being asynchronously loaded at runtime, I created an array each for the shape and texture offsets containing whether loading has finished. This is set to false before calling the load function, and the callback function sets it

to true. Additionally, since the reference faces are now arbitrary indexes, not 1, 2 and 3, I created an array storing which index the reference face at each corner of the triangle is, and index into that to find the correct index for the data. I also created a wrapper function for `drawFace` which instead takes the index of the offsets, returning `false` if the data hasn't been loaded yet, otherwise calling `drawFace` and returning `true`. This wrapper function is used to set the corresponding attributed in the `updated` object so that `draw` will continue to attempt to draw the new face until the data is loaded, and it can draw the face successfully.

Extending Synthesis

As per the spec, contributions from the replaced face must remain. Exactly how previous contributions should be weighted with respect to new contributions is unclear, as contributions are defined in the UI as a proportion of the three reference faces, by clicking in the triangle. I decided to go with the simplest solution, and just associate the normalized weights defined by clicking in the triangle with the appropriate offset index, and use those weights to create a new face by combining all loaded offsets with their associated weights (default zero). This also gives a clear definition of what should happen if an old reference face is used again - it simple has its weight updated, which keeps the behaviour consistent with previous behaviour from when there were only three reference faces.

Additionally, since selecting the offsets used the corresponding index of a corner of the interpolation triangle, I needed to refactor this to separately keep track of the indexes of the loaded faces. Because of javascript's sparse arrays, and the way `map` works, I can do this by creating a sparse array of contributions, which behaves like a key-value map. Calling `map` on this iterates through the values, and gives the key - `map` skips empty entries in an array. Because of this, I can easily use `map` functions to combine the component offsets into the offsets for the synthesised face.

Highly Advanced Spec

Phong Shading

Flat Phong shading was fairly easy to implement. First, I added a checkbox to toggle between phong and lambertian shading, and used the state of that checkbox to choose between the exerting lambertian shading function and the new phong shading function. The shading function takes the average colour and the calculated $cos\theta$, and uses it for the Phong formula. Since the data provides no values for the ambient or specular components, these are parameterised as variables.

User-Controlled Phong Shading

Since the Phong shading values are parameterised, I added text inputs to allow the user to define new values for these, along with a button to update the rendering, and a button to reset it to the initial values. As with all text input in this project, when updating the variables the variable is set to either the input number, or the current value if the input is empty or not a number. I decided to allow arbitrary number input, despite RGB values technically being bounded from 0 to 255, as allowing values out of these bounds can create interesting results, as the resulting colour can then clip.

User-Controlled Light Source

Light Direction

Since the light source is just a normalized directional vector, the direction of the light can easily be changed by modifying that vector. To allow the user to do this, I give them text inputs for the x, y and z of the light source, and a button to update the light source with the new values and redraw the faces.

Light Colour

In my shading formulae, I have assumed a pure white light source. I can extend this to a coloured light source by multiplying the each channel by the light source's power in that colour (bound from 0 to 1). To make this more useful, I added controls for the light colour similar to the light direction. The text inputs expect input between 0 and 255, normal RGB values, and these are then converted to values between 0 and 1 by dividing by 255 so the maths will work correctly.

Conclusion

Thanks to the experience from the previous graphics practical, and from using the p5 js library for another module, I was much more familiar with the javascript, and with a library that makes rendering to the HTML canvas much easier. This time, I was much better able to make use of the single-line function, which allowed me to write very neat code, and allowed me to write correct code faster, allowing me to complete more of this project than the previous one.

An exception to this is anything involving the HTML UI elements, which pretty much require large blocks of code in order to do anything useful, as it needs multiple function calls to setup each element. To reduce some of this code, I created container divs for some of the UI, within which other UI elements are positioned automatically, rather than having to set the position of

every UI element. However, putting each UI element within this container does require the exact same number of function calls as setting the positions independently.

There were some extensions I planned on attempting, but didn't have time to. First, multiple light sources. This would be a fairly easy extension to my existing code, requiring only a small amount of refactoring to combine the effects of multiple light sources. Additionally, I would have created three new UI elements - a dropdown to choose which light source you're editing, and a button to add a new light source, or remove the chosen light source - to allow the user to control the multiple light sources. Next, I would have added perspective rendering, which would just require applying a projection matrix within the `drawFaces` function (though this is non-trivial). Finally, I would have added smooth shading. Currently, my project only has flat shading. This is partially due to my use of p5's `triangle`, which can only render as a flat colour. However, p5 also allows you to draw a shape with differently coloured vertices, between which it will then interpolate a colour gradient to fill the shape. By making use of this, I could have implemented smooth shading,