Advanced Computer Graphics

WebGL 3D visualisation of the solar system

Abstract

This paper describes the considerations and difficulties involved in constructing a 3D visualisation of the solar system using WebGL, following a modularised, object-oriented JavaScript coding style without the assistance of libraries such as Three.js.

It details the structure of the program, both from a client-side and pre-production processing perspective. It goes on to describe the caveats of development (the self-imposed requirement of passing scientifically correct parameters to the constructors), and the results of the completed work, including tests to validate the application behaviour. There follows a conclusion and a self-evaluation of how well the assignment was completed.

The program, accompanied by this paper, contributes a reusable, modularised and extendible codebase to the field of 3D visualisations for the web. The implementation includes:

- *Animated, texture-mapped spheres representing all of the planets of our solar system in circular orbits.*
- *Earth's Moon and Jupiter's Galilean moons orbiting their respective planets.*
- *Saturn's rings.*
- *Specular maps for the Earth, so that light is reflected more strongly from the ocean than the continents.*
- *Configurable Phong-shading parameters, with light appearing to emanate from the Sun.*
- *A means of navigating the solar system through keyboard and mouse controls.*

A live demo is available here: http://users.aber.ac.uk/cba1/webgl/source/

Introduction

The assignment required building a 3D visualisation of the solar system, the aim being to gain familiarity with WebGL and its implementations of transformations, texture mapping, shading and animation.

This wasn't the first reproduction of the solar system in WebGL; other visualisations already exist[1]. However, the scope for the assignment would allow us to go above and beyond what others have managed to achieve in previous visualisations, including semi-transparent rings of Saturn, specular mapping, keyboard and mouse controls and configurable lighting effects.

[1] http://mgvez.github.io/jsorrery/, accessed 17/11/2014

Methods

Assignment structure

My assignment comprises both front-end and back-end parts. The back-end uses *Grunt* (written in *Node.js*) and encapsulates formatting checking and documentation generation. The front-end uses the *Require* JavaScript library to handle dependency management and to modularise the code.

Application structure

- *README.md* – the markdown file associated with my project. I intend to open source my program on or after the submission deadline.
- *Gruntfile.js* – defines the tasks I want to automate.
- *package.json* – a file used by Node to install Grunt's dependencies and the dependencies of all of its registered tasks.
- */source* – contains the source code of the application.
    - *index.html* – runs the application. Also contains the application shaders and the CSS for the web page.
    - *textures/* - contains the planet image textures.
    - *js/* - contains the JavaScript for the application.
        - *main.js* – the file initially downloaded in index.html. Uses *Require* to handle downloading its dependencies.
        - *lib/* - contains third-party JavaScript libraries.
        - *app/* - contains my own JavaScript code for the application.

Program structure

Inside *source/js/app/*, we have these files:

- *app.js* – entry point for the application (pulled into *source/js/main.js*).
- *astronomical_object.js* – defines my class for Astronomical Objects, including the Sun, Planets, Moons, Saturn's Rings and the Galaxy.
- *buffers.js* – handles initialising the buffers and drawing the individual elements that make up the astronomical objects.
- *camera.js* – defines the projection view matrix of the 'camera' in the 'world', and defines functions allowing the user to manipulate the camera.
- *controls.js* – defines mouse and keyboard controls.
- *controls_gui.js* – dynamically creates the graphical user interface surrounding the canvas, allowing the user to adjust lighting conditions and orbital speeds.
- *gl.js* – handles getting the WebGL context.
- *lighting.js* – handles getting the lighting parameters from the GUI and preparing the shaders for drawing.
- *shaders.js* – defines the JavaScript attributes that link to the custom shader code.
- *solar_system.js* - defines all of the Astronomical Objects that compose my solar system.

Matrix methods

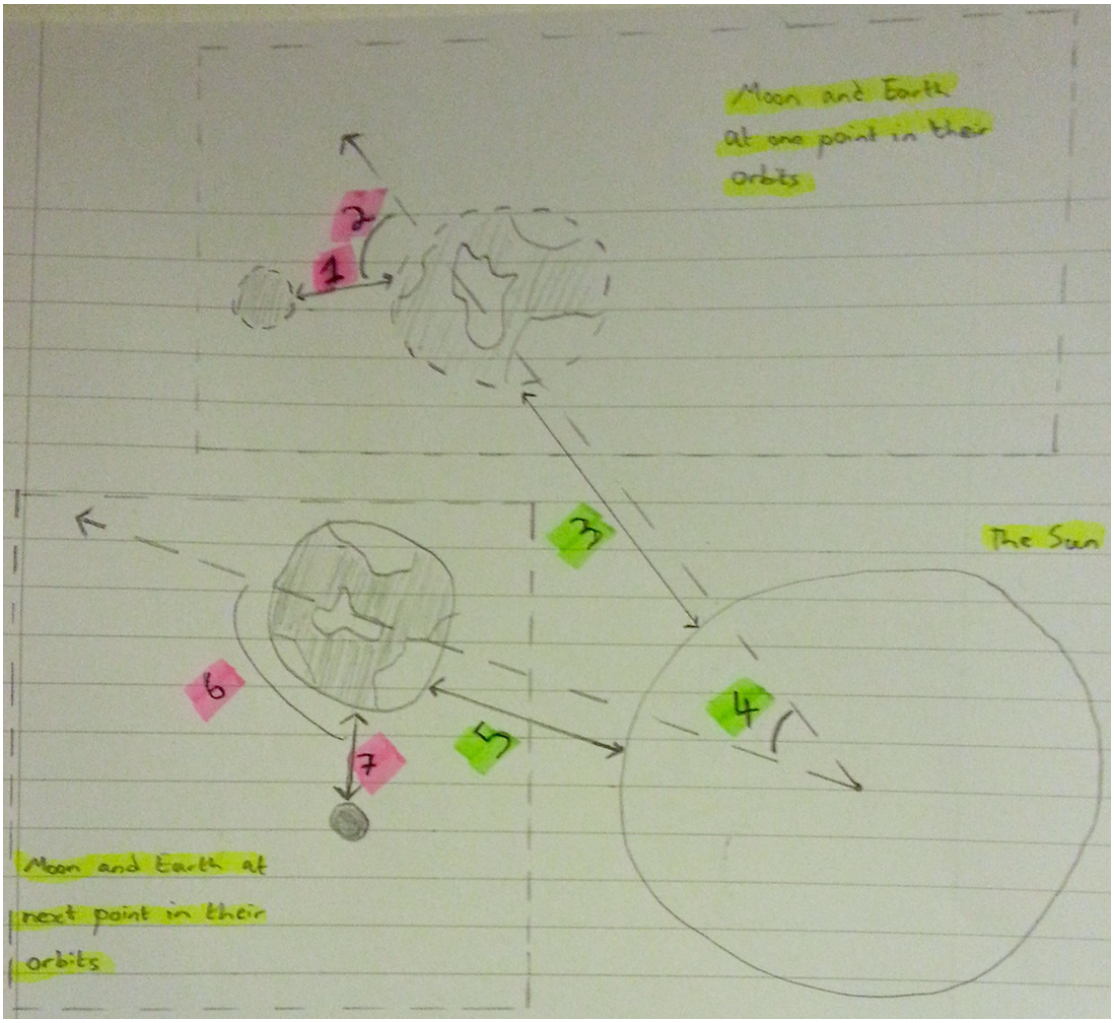

Figure 1. An early design for my orbiting algorithm

Orbiting was by far the most difficult algorithm to perfect in the assignment. Figure 1 shows the complication of the orbital steps even without the complication of bodies spinning on their axes.

For planets, the process was relatively straightforward: translate to the origin of the Sun, rotate by a small orbital angle then translate back out by the same distance.

The algorithm of the moon's orbit is more complex:

- Translate to the origin of the orbited planet.
- Rotate back to the original starting angle (to be in line with the Sun).
- Translate to the origin of the Sun.
- Rotate by the last orbit angle of the orbited planet.
- Translate back out by the planet orbital distance.
- Rotate by the new *moon* orbit angle.
- Translate back out by the moon orbital distance.

Taking into account bodies spinning on their axes adds additional complexity. Before each translation, the body must be rotated by the negative value of the cumulative rotation angle to date. After the translation, the body is rotated by the cumulate rotation angle to date (resetting its last orbit) *plus* a new small rotation angle, so that frame by frame the body slowly rotates on its axis.

Orbits and rotations required the following glMatrix functions:

- mat4.rotate() – to rotate/orbit.
- mat4.translate() – to translate to the origin of the body being orbited, allowing orbiting behaviour.
- mat4.multiply() – to multiply orbit and rotation by the local model view matrix so that the changes take effect.
- mat3.normalFromMat4() – extracts the normal from the model view matrix (required for Phong shading).

Source/js/app/camera.js uses:

- mat4.perspective() - change the projection of the world according to the height and width of the canvas, and given a certain degree of view.
- mat4.identity() – resets the camera matrix (used in the snapTo function).

Timing considerations

I wanted my system to be scientifically accurate, so *source/js/app/solar_system.js* passes scientifically correct parameters, which dictate orbital and rotational speed. A further complication is the fact that the number of milliseconds that represents a day is configurable, hence millisecondsPerDay is passed to the animate function in *source/js/app/astronomical_object.js* and is used to determine the amount by which each body should move per frame.

Data structures & coding considerations

source/js/app/astronomical_object.js describes the data structure for my Astronomical Objects, which represent everything rendered on the canvas. Although Saturn's Rings may seem very different to, say, Mars, enough code is shared that splitting into separate classes would create unmaintainable code duplication.

Almost every astronomical object needs to hook into the animate() function and use the number of milliseconds representing a day, to calculate the amount of orbiting and rotation required. Every object has a shape and must be rendered. Conceptually, Saturn's Rings are the most unique object in my solar system, but only differ from planets and moons in that they're represented as a cuboid rather than a sphere. They differ only from moons in that they have an orbital distance of zero, i.e. are not translated away from Saturn but are rendered from Saturn's origin.

Results

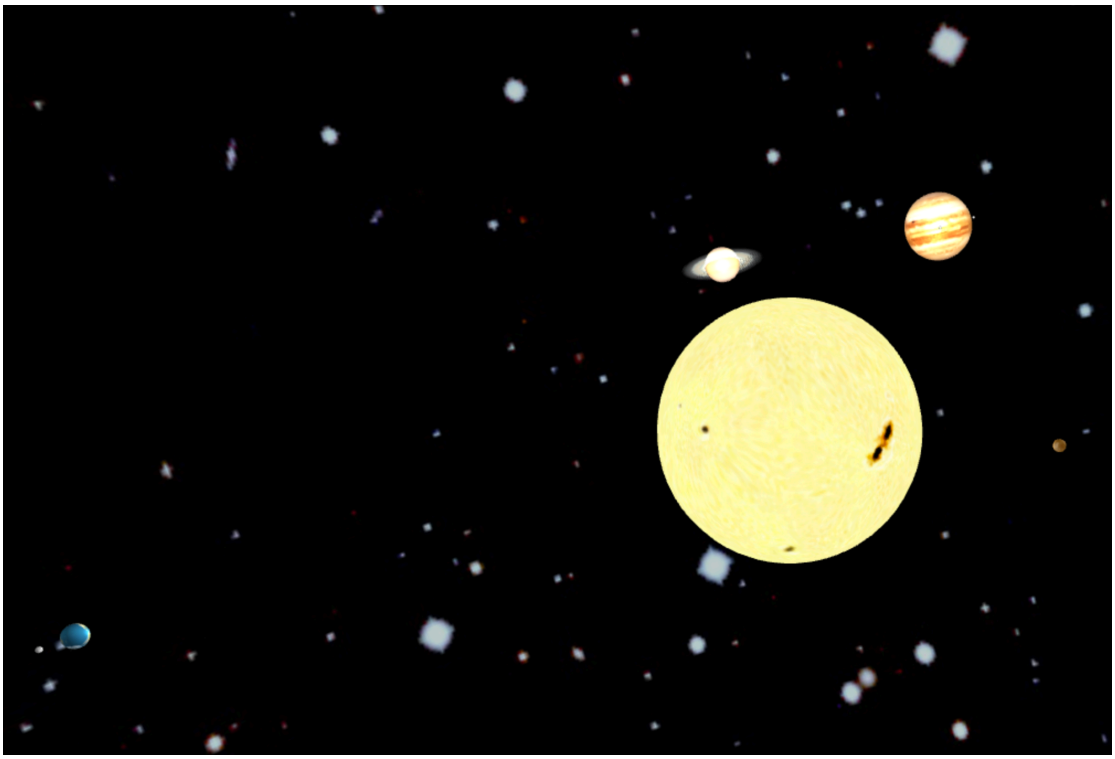

Figure 2. Screenshot of part of my solar system implementation, depicting the Sun, Jupiter, Saturn and its Rings, the Earth and the Moon, and the Galaxy

Solar system composition

- The Sun
- The following planets, orbiting the Sun:
    - Mercury
    - Venus
    - Earth
    - Mars
    - Jupiter
    - Saturn
    - Uranus
    - Neptune
- The Earth's Moon, orbiting the Earth
- Jupiter's Galilean Moons (Io, Europa, Ganymede, Callisto), orbiting Jupiter
- Saturn's Rings
- Galaxy 'skybox' encasing my solar system and providing a better perception of depth

Saturn's rings are on a scientifically accurate tilt and are semi-transparent, as can be seen in Figure 3. They're rendered as a flat cuboidal element with blending enabled, as suggested in the assignment brief.

All of the planets and moons are passed scientifically correct parameters (seen in *source/js/app/solar_system.js*) regarding orbital distance, orbital and spin period, radius and axis. These are normalised in *source/js/app/astronomical_object.js*.

By 'normalised', I mean that orbital distances of the planets and moons are accurate relative to one another, but for presentational purposes have been scaled down by a factor of 50,000. Similarly, radii are proportionally accurate to one another but are scaled down by a factor of 100, and in the case of the Sun, 1000.

Finally, orbital distances and rotation speeds are in proportion to one another, and can be sped up or slowed down using the GUI sliders available. By default, one second in my program is equivalent to one Earth day.

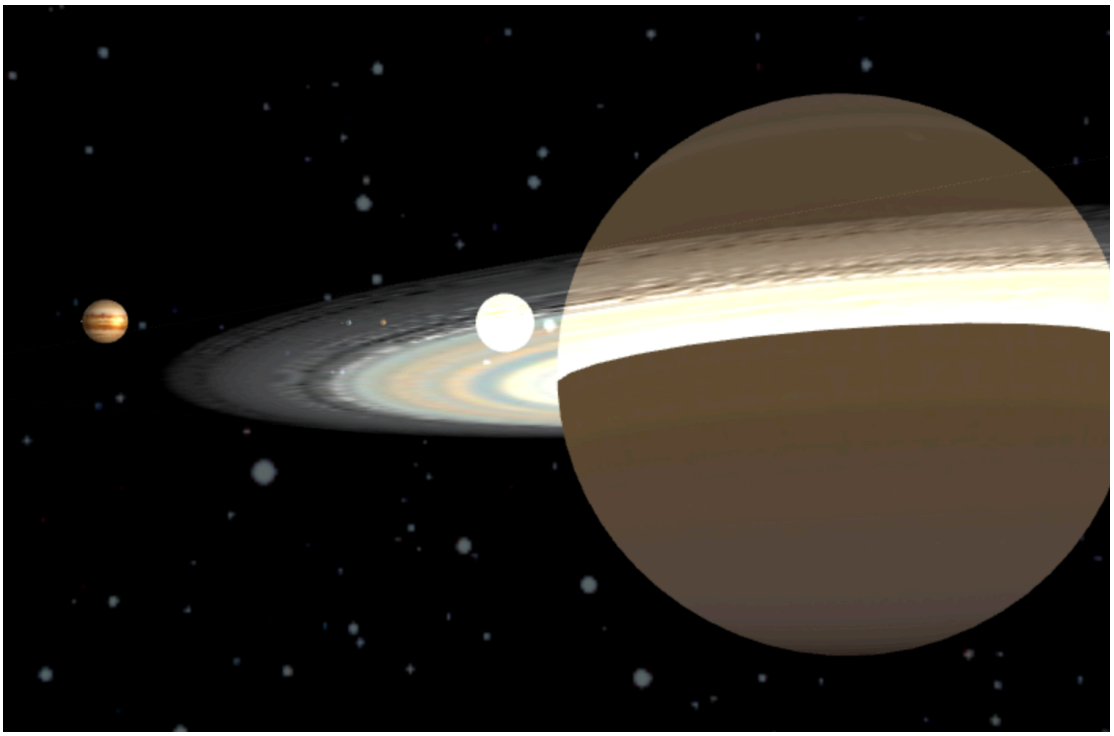

Figure 3. Screenshot showing the transparency of Saturn's Rings.

Orbital testing

Planets and moons spin and orbit counter clockwise with the exception of Venus and Uranus, who rotate on their axis in a clockwise fashion due to their axial tilt. The easiest way to confirm this is to hit keyboard shortcut '7' to view Uranus spinning clockwise.

In order to ensure the orbits are working correctly, I produced these conditions (seen in Figure 4):

- Set the Earth day to be equal to 1000 milliseconds
- Press '3' to snap the camera to Earth
- Make a mental note of the position of the Moon in relation to Earth.
- Count for 28 seconds, slowly moving the camera (or repeatedly pressing '3') to follow Earth's movement.
- The Moon should have orbited the Earth and be back at its original starting position.
- In addition, try and follow a specific continent on Earth and you'll see that the Earth takes 1 second to fully rotate.

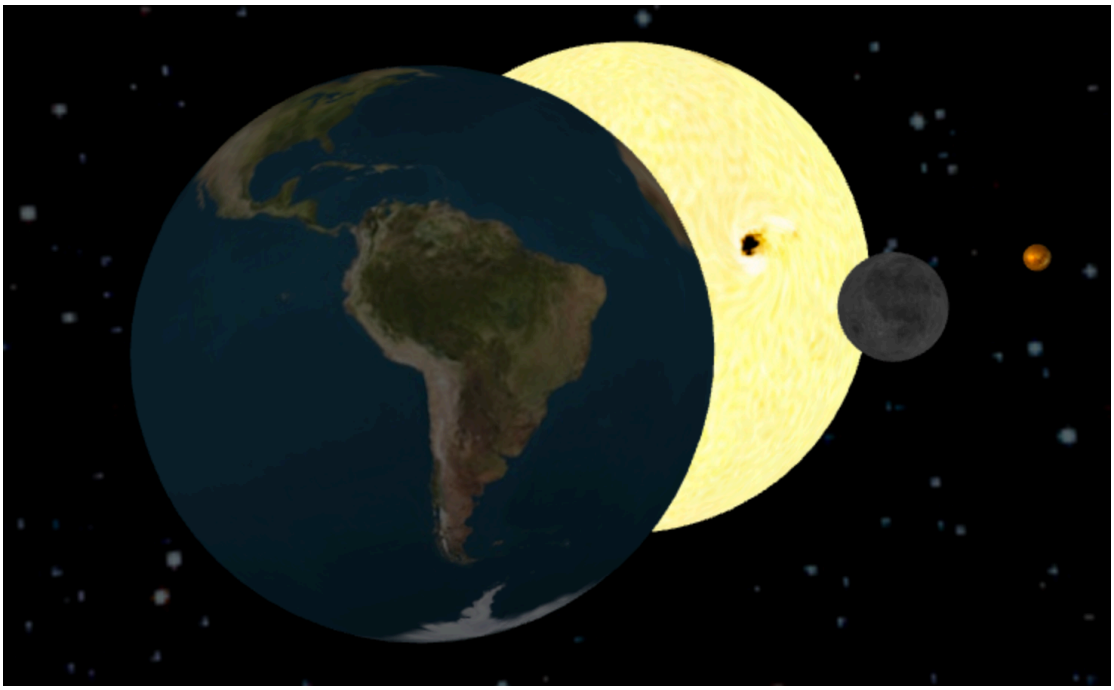

Figure 4. Checking the moon's orbit matches the number of full Earth rotations

To test the correct orbital speeds of the planets orbiting the Sun, reproduce these conditions:

- Press 'r' to reset the camera.
- Drag the canvas so that we have a bird's-eye view of the Sun and the planets orbiting it.
- Speed up time to be 100 milliseconds per Earth day.
- Watch Mercury – it takes 87.66 Earth days to orbit the Sun, so should take around 8.77 seconds to go around the Sun once.

Specular maps

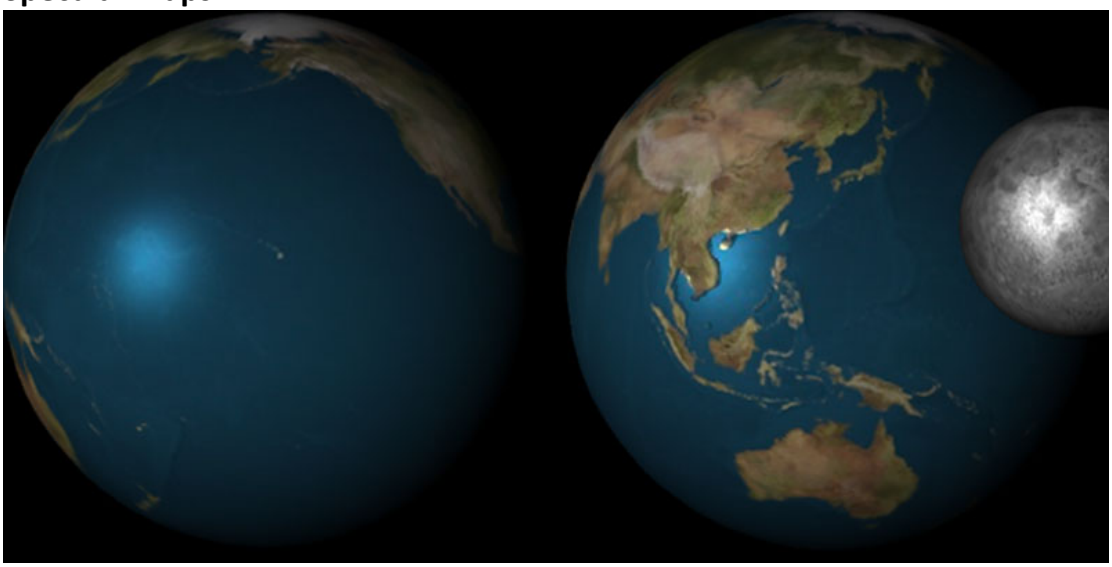

Figure 5. Phong shading of the Earth with specular maps

Using the black and white GIF of the Earth from Learning WebGL[2], I added specular mapping so that the naturally shiny parts of the Earth (i.e. the ocean) reflected light more strongly than the continents, which are more dense and opaque. This can be seen in Figure 5, which shows the specular effect on pure ocean and also on a combination of ocean and land. It is difficult to capture the effect in a screenshot. Instead, to recreate the conditions using the application:

- Press 'r' to reset the camera.
- Slow down time (1 Earth day = 5000 milliseconds).
- Use a combination of the W, A, S, D keyboard controls and the mouse to try and get the Earth into view.
- To maximise the effect, lower the Ambient Light levels to zero and increase the Specular term globally to 1.0.

Note: planets that use specular mapping are unaffected by changes to the global 'Planet shininess' input.

GUI Controls

The GUI has a slider for speeding up and slowing down time, which affects the time taken for planets and moons to complete their orbits and full rotations. There is a slider affecting the 'shininess' property of planets for specular shading. Finally, there are three 'global' sliders affecting the overall RGB values of the Ambient Light, Specular Light and Diffuse Light terms, as well as individual sliders for each colour spectrum of each term. These are immediately reflected in what is rendered on the canvas, whether or not the animation is paused.

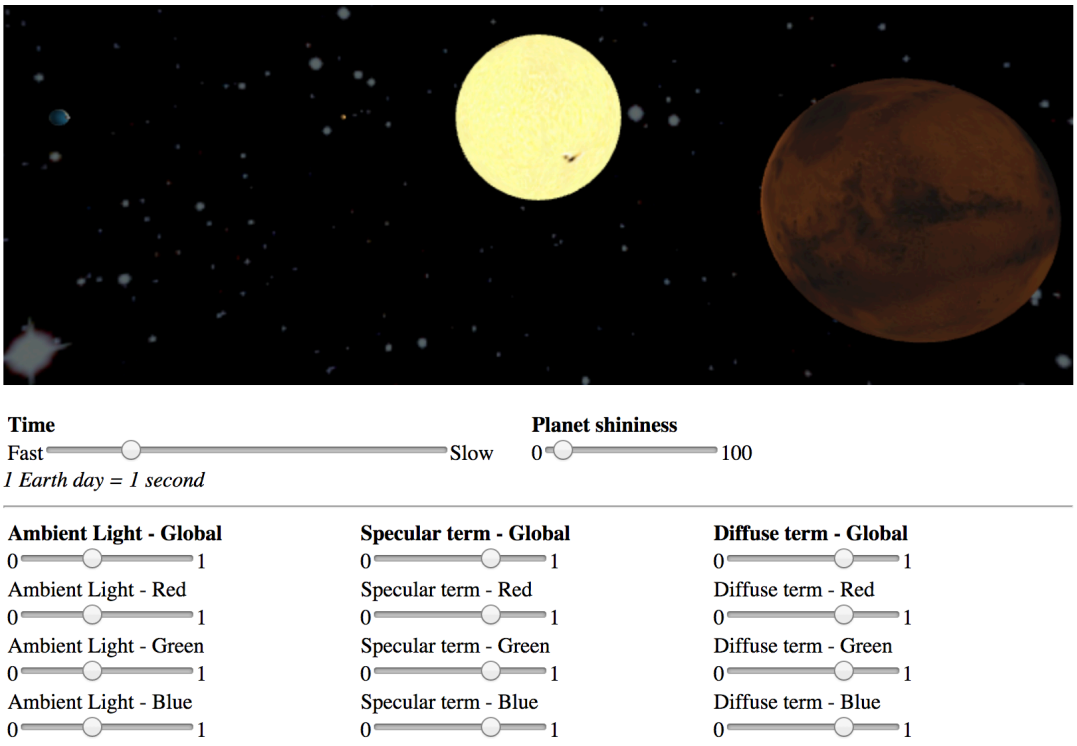

Figure 6. Screenshot of the GUI controls generated by the program

[2] http://learningwebgl.com/blog/?p=1778, accessed 19/11/2014

The GUI elements are rendered using JavaScript to allow for further configuration in future, without forcing people who may be using my code to manually update their HTML. It also gives me the freedom to add a configuration property, which would allow users/website hosts to turn the GUI on or off.

Other Controls

The user's position in the solar system can be manipulated by dragging the canvas using the mouse, which changes the user's perspective. In addition to mouse controls, my program has numerous keyboard controls:

- P – pause animation
- F – toggle Full Screen Mode
- R – reset camera to original position
- W, A, S, D – controls allowing movement around the solar system. Holding down any one of these keys for a period of time accelerates the movement speed, making movement reasonably quick.
- 1-8 – snaps the camera to an associated planet. Full instructions can be viewed by clicking the 'Toggle instructions' link at the top of *index.html*.

Discussion

I decided against using a global matrix stack for maintaining the positions and rotations of the system, preferring the more object-oriented approach of having each Astronomical Object handle its own matrix history.

I believe this was the right decision, but it did make the planets' orbits (and particularly the moons' orbits) quite a complex mathematical problem to solve. Throughout my *source/js/app/astronomical_object.js* class I check if the current object orbits something that orbits something else, conditionally performing different operations depending on the result. As it stands, my program would not be able to cope with a triple-orbit hierarchy without some significant refactoring Additionally I found elliptical orbits too difficult to integrate into my system in the remaining time.

Conclusion and self-evaluation

This was a challenging assignment with plenty of scope for creativity and additional functionality. Though difficult, it was enjoyable to program and I'm grateful to have another portfolio piece that I can display on my website.

If I had been able to put in more time than the ~60 hours already spent getting the system to this stage, I would have liked to have implemented elliptical orbits and bump mapping for Earth's moon. I understand that without these non-trivial extras my assignment is unlikely to gain the full 100%.

Overall, I believe that my work is deserving of around 90%. Below is my justification, using the marking scheme as a guide.

20% - Written report
This report is an accurate and honest description of my program and has been written in the scientific style favoured during lectures.

15% - For correctly drawing the spheres
My WebGL solar system correctly draws the spheres representing the Sun, planets and moons, at their correct scales (adjusted before rendering for aesthetic purposes).

15% - For correctly texture mapping the spheres
Good, object-oriented code allows me to pass the URL of the texture map image to each Astronomical Object as part of its individual constructor parameters. The optional specular map texture parameter shows that my application makes use of multi-texturing.

15% - For correctly lighting the spheres
My spheres are lit using Phong shading, as described in the assignment brief. A GUI allows the user to adjust the lighting and shininess parameters and immediately see the effects rendered onto the canvas. Specular maps can be passed to Astronomical Objects for more realistic lighting effects.

15% - For correctly positioning and animating the spheres
I positioned the spheres at the correct relative distances to the Sun, and each planet, moon and star spins on its scientifically correct axis. The Astronomical Objects orbit and rotate at the correct speed proportional to one another, and this speed is adjustable using the GUI.

20% - For additional functionality implemented
Here is a list of additional functionality added to my system:

- As suggested in the brief, I've added the rings of Saturn. These are correctly tilted and are semi-transparent, as seen in Figure 3.
- Specular mapping of the Earth makes use of multi-texturing to improve the realism of the Phong shading. This was built in an abstract way so that specular maps could be passed to any of the Astronomical Objects easily.
- Keyboard controls (including general navigation and planet 'snapping') and mouse controls allow easy navigation around the solar system.
- GUI elements allow the user to control the lighting conditions of the solar system as well as the orbital and rotational speed of the elements.
- An attractive web interface, decoupled from the specifics of the application, containing content and scripts that progressively enhance the user experience.
- Additional documentation for the API of the JavaScript itself. This is generated by the `yuidoc` Grunt task reading the comments in my code and can be accessed by clicking the 'Documentation' link from *index.html*.
- Finally, I've added a universe background to make the solar system look more realistic and add an element of depth.

References/Acknowledgements

Orbit distances taken from:
http://www.northern-stars.com/solar_system_distance_scal.htm

Orbit periods and rotation periods taken from:
http://www.windows2universe.org/our_solar_system/planets_table.html

Planet sizes taken from:
http://www.universetoday.com/36649/planets-in-order-of-size/

Planet axes taken from:
http://www.astronomynotes.com/tables/tablesb.htm

Jupiter's Galilean moons information taken from:
http://www.daviddarling.info/encyclopedia/J/Jupitermoons.html

Saturn's rings info taken from:
http://cseligman.com/text/planets/saturnrings.htm

Planet texture maps were taken from:
http://planetpixelemporium.com/

Lots of code taken from/inspired by various lessons at:
http://learningwebgl.com/blog/?page_id=1217

Phong shading heavily based upon:
http://learningwebgl.com/blog/?p=1658

Specular map code heavily based upon:
http://learningwebgl.com/blog/?p=1778