­CS32310 Assignment - Report

# Abstract

*This paper details how to construct a 3D visualisation of the solar system using WebGL. It details a modularised, object-oriented JavaScript coding style that allows maximum configurability of the generated planets.*

# 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]](#footnote-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, keyboard and mouse controls and configurable lighting effects.

These were in addition to the core requirements of rendering and correctly animating texture-mapped phong-shaded spheres representing the Sun, Earth and Moon.

# 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, unit testing, and other forms of pre-processing. The front-end uses the *Require* JavaScript library to handle dependency management and to modularise my code.

### Application structure

* *README.md* – the markdown file associated with my project. I intend to open source my program after submission.
* *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

## Macintosh HD:Users:ashton:Dropbox:uni:year_4:cs323_10_advanced_computer_graphics:assignment:documentation:orbit_alorithm_design.jpg

*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 axis.

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 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 rotating 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 require 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

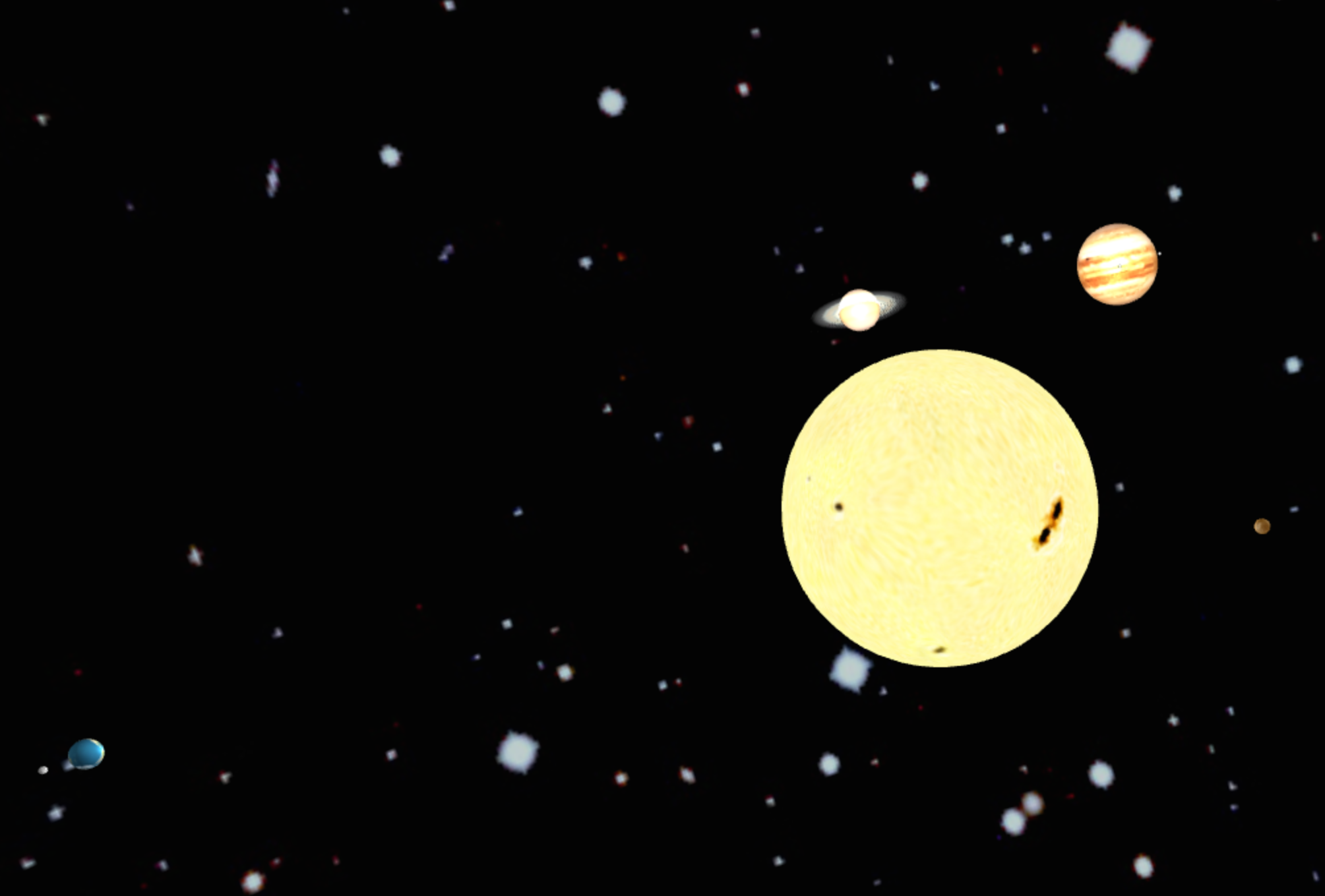
I wanted my system to be scientifically accurate, so *source/js/app/solar\_system.js* passes scientifically correct parameters, which dictate orbital and rotational speed. A further complication is the fact that the number of milliseconds that represents a day is configurable, hence millisecondsPerDay 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

*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 by splitting into separate classes there’d be unmaintainable code duplication.

Almost every astronomical object needs to hook into the animate() function and respond to the number of milliseconds representing day, be it by orbiting, rotating, and so on. 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

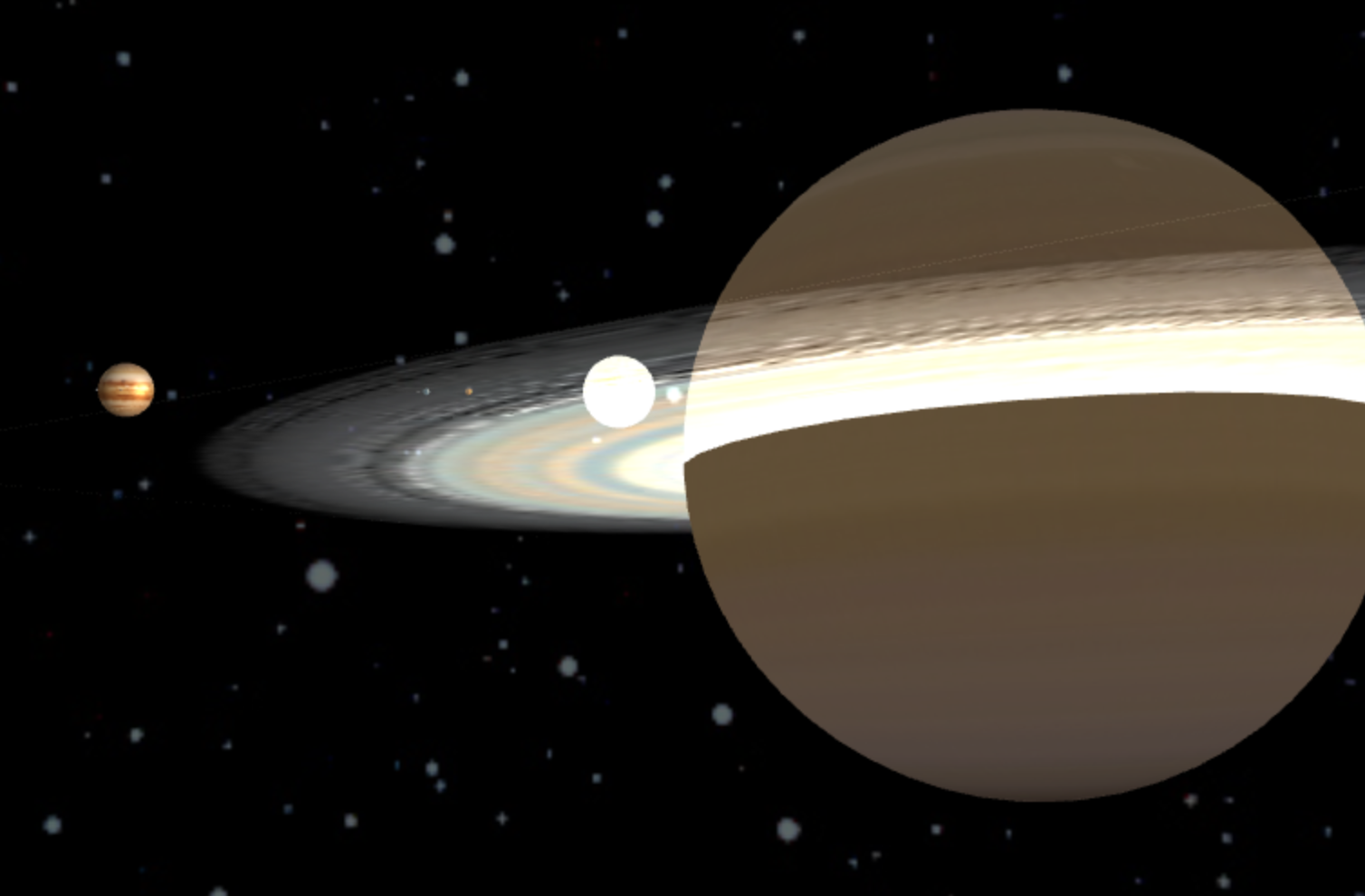
* The Sun
* The following planets, orbiting the Sun:
  + Mercury
  + Venus
  + Earth
  + Mars
  + Jupiter
  + Saturn
  + Uranus
  + Neptune
* The Earth’s Moon, orbiting the Earth
* Jupiter’s Galilean Moons (Io, Europa, Ganymede, Callisto), orbiting Jupiter
* Saturn’s Rings
* Galaxy ‘skybox’ encasing 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 2. 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.

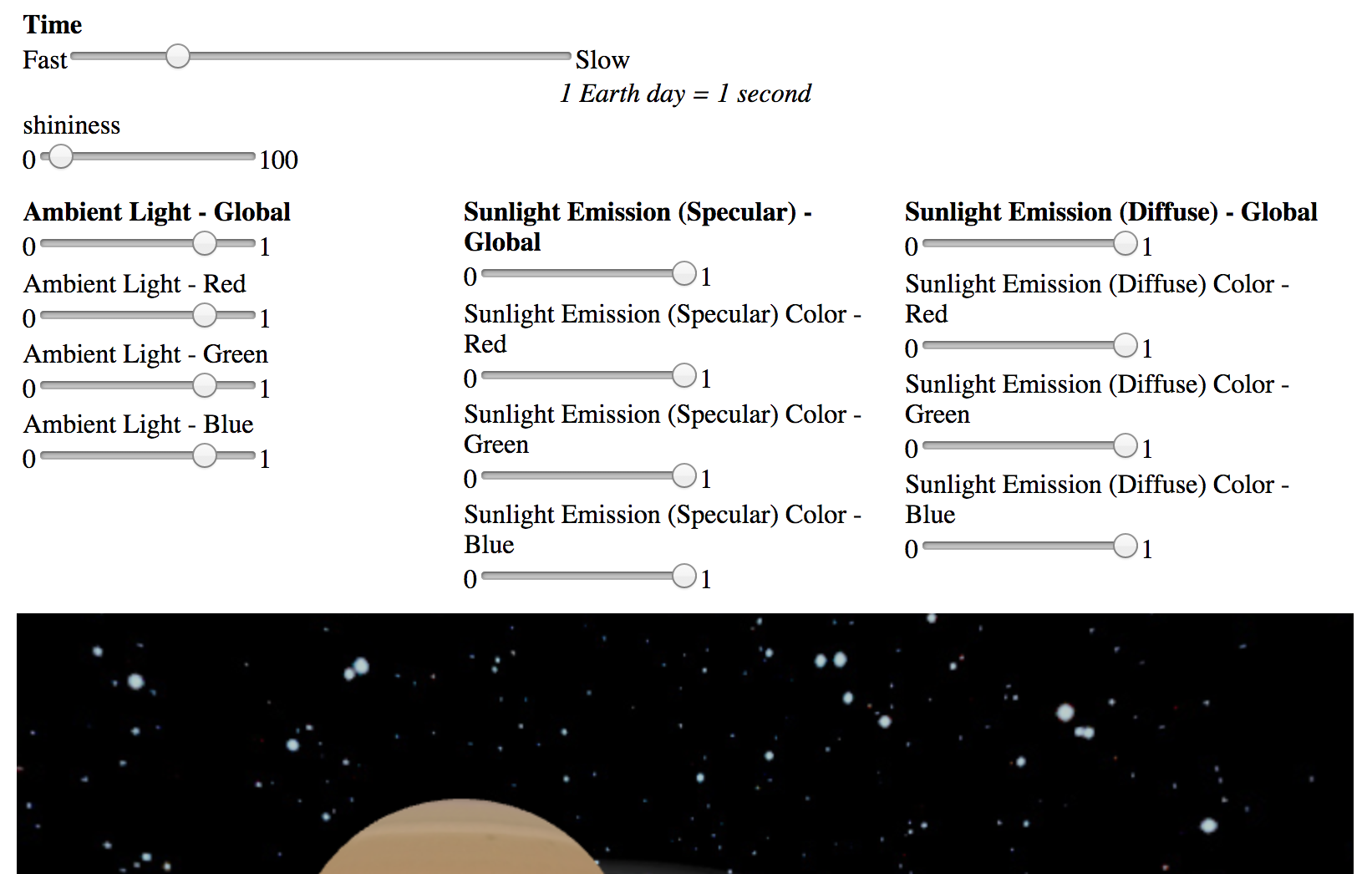


*Figure 3. Screenshot showing the transparency of Saturn’s Rings.*

## GUI Controls

The GUI elements (seen in Figure 3) are rendered using JavaScript to allow for further configuration in future. By being encapsulated in the JavaScript, I can deploy updates to the GUI elements and users who are hosting my open source solar system on their own websites can update to the latest version of the project without having to manually update their HTML to work with the latest codebase. I also have the freedom to add a configuration property in future, which would allow users/website hosts to turn the GUI on or off.

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 4. Screenshot of the GUI controls generated by the program*

## 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 lie below the canvas when the application is running.

# 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.

# 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 a portfolio piece that I can display on my website.

I believe that my work is deserving of around 85%. 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 is sufficiently detailed.

## 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. These are correctly rendered at the right scale. In addition, I have a texture mapped galaxy background, which required sourcing an image and cropping it to ‘power of two’ dimensions.

## 15% - For correctly lighting the spheres

My spheres are lit using Phong shading, as required by the assignment brief. A GUI allows the user to adjust the lighting and shininess parameters and immediately see the effects rendered onto the canvas.

## 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 Bodies orbit and rotate at the correct speed proportional to one another – this speed is adjustable using the GUI.

## 20% - For additional functionality implemented

* As suggested in the brief, I’ve added the rings of Saturn. These are correctly tilted and are semi-transparent, as seen in Figure 2.
* 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.
* Finally, I’ve added a universe background to make the solar system look more realistic and add an element of depth.

If I had been able to put in more time than the ~50 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%.

# 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

1. http://mgvez.github.io/jsorrery/ [↑](#footnote-ref-1)