CS323 Assignment
Building a 3D visualisation of the solar system

James Bowcott
jab41@aber.ac.uk

November 21, 2014

1 Abstract

This report details my implementation of an interactive 3D Solar System visualisation program for the CS323 Advanced Computer Graphics assignment. It describes the approach taken and techniques used in the implementation and discusses any issues that arose during the development process. Finally it evaluates the finished product and reflects on lessons learnt from the assignment.

2 Brief

The objective of the assignment was to write a WebGL application which visualises the solar system by using and building upon concepts and skills learnt in the module in 3D graphics and WebGL programming.

The assignment brief required that the application display a basic scene of our solar system, comprised of the Sun, the Earth and Earth's Moon, which should be represented as textured spheres, moving in 3D space in their expected orbits. Also, the "bodies" (a Body in this this report refers to the Sun, the Moon or a planet) should spin on their axes. The scene should have a single light source (the Sun) and that the light be reflected off the bodies in a realistic way.

The brief suggested several concepts and methods for achieving some of the required functionality, such as combing separate translation and rotation matrices, and employing a matrix stack or scene graph.

In addition to the requirements possible extensions were outlined for the opportunity to achieve a higher mark, including having all planets in the solar system instead of just Earth, adding the rings of Saturn, more realistic elliptical orbits instead of just circular ones and multi-texturing (e.g. adding a cloud mask to Earth).

3 Approach

From the outset I wanted to implement a full solar system with all 9 planets, the Sun and Earth's Moon following realistic orbits. I wanted the user to be able to interact with the scene by manipulating the camera to change the view of the solar system and to control the speed of the simulation to easily see how planets move over time. Also, if time permitted, I wanted to add one or more special effects to make it more visually appealing.

The challenge was not only in knowing how to properly create and manage the 3D graphics but also because it needed to be a WebGL program which required writing the application code in Javascript which I had no prior experience with, so this was going to be a stepper learning curve for me than for others who may have done web development modules in the past.

The general approach I took to the implementation was to make it highly flexible and well designed. Instead of the entire program being contained in a single HTML file the Javascript, Shaders, HTML and CSS are separated into several files, each responsible for a specific part of the application. The modules of the program are:

main.js Utility functions used by any part of the application. Initialises and "ticks" the other modules in the correct order as it requests each frame of animation from the browser.

gl.js All the graphics state and rendering code. Solely responsible for interfacing with WebGL, which it abstracts to other modules through constructs and functions.

solar.js Creates and manages the solar system using data stored in **solarsystem.json**

ui.js Interfaces between the browser and the rest of the program.

The contents and functionality of each module is described in the next section.

This separation of code into modules helped a lot during development as the codebase grew as it eliminated boilerplate code, made the code less error-prone and made navigation of source code easier. It also makes a lot of the source code reusable. For example the graphics source file **gl.js** only contains general purpose graphics code, it does not know what it is going to be rendering, that is all handled by the solar system source file which uses constructs provided by the graphics module to produce the scene. In theory the graphics code could be used to create an entirely different type of scene without modification. Although this was not a requirement of the assignment and may not contribute to the final mark it aided greatly during development to have these distinct parts of the program separate.

Normally WebGL shader source code has to be embedded in the HTML source. To have them as separate files they needed to be loaded dynamically as external .glsl files via XMLHttpRequest. A side effect of this however is that it prevents the application from being run by opening the HTML file from a local filesystem. The browser security models rightly do not allow XMLHttpRequest to load files from the file:// schema, so the application has to be run over a HTTP server.

The only library used was glMatrix which provided the matrix and vector math functions.

4 Methods

This section describes the various techniques used to produce the graphical scene and how other functionality of the application, such as user interaction, is achieved.

4.1 Solar System

The program simulates a solar system which has all 9 planets orbiting the Sun at the centre as well as the Moon orbiting Earth using real solar system data. The orbits have eccentricity and inclination and the bodies have axial tilt. Each body follows it's orbit and rotates on it's axis at the correct time scales.

The JSON file **solarsystem.json** describes everything about the solar system, including all body and orbit parameters and texture image filenames. Bodies are organised hierarchically through "satellites" attributes. All planets are satellites of the Sun body and the Moon is a satellite of the Earth body.

Real world units are used for the parameters, e.g. body radius is in km and orbit radius are in astronomical units. These units are heavily scaled to the coordinate system used by WebGL. Using real world units seemed like a good idea to describe the relative scales between planets, but it breaks down in some cases due to the heavy scaling involved. For example, when the Moon's real orbital radius was used it ended up inside the Earth model, and the Sun is many times larger relative to the planets, so these values were faked to achieve the best end result.

Time is represented in Earth days and all changes in the scene are computed relative to this. For example the default speed of the simulation is one Earth day per second, so you will see the Earth doing a full rotation in one second, and everything moving and rotating relative to that timescale.

The solar system is created and managed by **solar.js**. It reads solarsystem.json and uses that data to instantiate objects of two different types: **Body** and **Orbit**.

A **Body** object is responsible for everything to do with a 'body' (The Sun, a planet, the Moon or any other spherical object), including instantiating a sphere model (see Models) and applying appropriate textures, orientating it's axial tilt and rotating it on it's axis when told to 'advance' by a fractional number of Earth days. It is not responsible however for body model's position, as this is defined by the orbit which is in a separate object.

Each Body object has three 4x4 matrices to describe it's position, tilt and rotation:

The translation matrix is set by the body's Orbit object (if it has one) on each frame to position the body along the current point on the orbit path. This is just a translation and contains no rotation. It is also used by the orbits of any satellite bodies to orient them around the body.

The rotation matrix contains the current axial rotation of the body as a rotation of the Y axis. This is only used by the body's model and not to calculate the camera view matrix or satellite body orbits. On each frame this rotation is incremented by an angle of the number of Earth days the solar system is to advance by divided by how many Earth days it takes for the body to complete a rotation.

Finally the tilt matrix just has the axial tilt of the body described as a rotation of the X axis.

The body's model matrix is set to the multiplication of these three matrices.

The camera can be orientated to follow a body. The body pushes a matrix for the view matrix to be based on composed of the body's current position. This is described in more detail in the Camera section.

Finally, bodies can have a ring texture, which is simply a Pane model with a specified radius which intersects the sphere's equator. The model's matrix is that of the body's translation and tilt matrices. The texture's alpha channel makes the rings appear like a circular band of colours. Only Saturn uses this facility, but it can be used by any body from solarsystem.json.

Orbit objects describe the orbit that a Body follows. It is responsible for translating the body to the correct position at a point in time. It also instantiates an ellipse model which is drawn to show the path of the orbit.

The shape of the orbit is defined from several parameters as described in solarsystem.json:
Radius a.k.a. the semi-major axis. In astronomical units. The maximum radius of the orbit.
Eccentricity in terms of 0.0 to 1.0 defines how elliptical the orbit is and how close to the parent body the orbit is at the point of closest distance (the periapsis) by translating the orbit by $radius * eccentricity$.
Inclination is the angle of how tilted the orbit is along the orbital plane, which is the Z axis.
Ascending Node is the angle of the orbit along the orbital plane. So called as the angle by which the inclination of the orbit begins to ascend. This is the Y rotation of the orbit.
Days. The number of Earth days it takes for the body to complete a full orbit.

Orbits have two matrices:

The base matrix defines the orbit's origin and is set to the translation matrix of the orbit's parent body. The Moon's orbit base matrix is the Earth's body translation matrix. Every other orbit does not have a base matrix as their origin is the Sun, but for those who do the parent body should have it's translation matrix updated before any satellite orbits are updated otherwise they would be out of place.

The orbit matrix describes the inclination and angle of the ascending node by rotating the Y and Z axes respectively.

Multiplying these two matrices results in the model matrix for the orbit model and the translation matrix for the Body to which the orbit belongs to.

The Orbit is responsible for setting the Body's translation without rotating it. It does this with the following formula:
$r = R\sqrt{1 - E^2}$
$\theta = 2\pi P$
$x = r \cos \theta$
$z = R \sin \theta$
Where R is the major radius, E is the eccentricity, P is the period of the orbit (0 - 1) and r is the calculated minor radius.

The body's translation matrix is composed of the translation vector applied to the base matrix for origin and orbit matrix which will provide the Y position of the inclination.

Orbit's are 'advanced' in the same way as Bodies are by providing a fraction of Earth day's to advance the orbit period by.

4.2 Models

There are 3 different types of models used in the scene: spheres, ellipses and panes. Each model is described by vertex, normal, colour and texture coordinate arrays which are held in **Model** objects.

The **Model** object prototype is contained in **gl.js** and is used to fully encapsulate the data and code needed to represent any 3D model in the scene, such as the arrays listed above, texture objects and a model matrix. The code to set up the WebGL context and shaders to draw the model at render time is also encapsulated in the object. The idea is that all the programmer has to do to get a model in the scene is to create a new instance of **Model** by providing the vertex, normal and colour data and push the resulting object to the global model stack **glModels**. The rendering pass calls the **toShader** method on each model in turn which pushes the stored data to the shader. Each **Model** has a 4x4 matrix for storing that model's final translation, rotation and scale. It is passed to the vertex shader as the model matrix when rendering the model.

How the vertices should be interpreted and drawn by WebGL is specified by a **drawmode** variable, which could be as triangles, a triangle strip, lines or a line strip.

models.js contains factory functions to create model data for various shapes including spheres, ellipses and panes. Each function takes parameters which describe the required radius, resolution, colour etc. The objects returned by these functions can be directly passed into the **Model** constructor. This is how **solar.js** creates the spheres for bodies and ellipses for the orbit paths.

4.3 Rendering

All rendering is done by **gl.js** and the shaders which it manages and interacts with. A rendering pass basically iterates over each Model object on the global model stack and calls its **toShader** function which pushes all of the data it contains to the main shader. The matrix uniforms, which include the projection, view and the model's matrix, are also pushed and the appropriate rendering mode and shader flags are set for this particular model. The shaders then take over by transforming the vertices by the multiplied matrices and appropriately shading the fragments which is described below in Lighting.

The actual rendering pipeline takes multiple passes with different modes and shaders. This is described below in the Bloom section.

4.4 Textures

A model can have up to 3 different textures:

Standard/Day texture This is a single normal texture which is affected by lighting. The fragment shader multiplies the texture's colour by the calculated light intensity value I, where $I = 1$ is lit and the colours are unaffected, $I < 1$ is dim and the colours are darker, and $I > 1$ is used in specular spots and the colours are made brighter.

Night texture When supplied a night texture will be shown on the dark/unlit parts of an object instead of a darkened day texture. The colour of night textures are not affected by lighting, as it is assumed the texture image itself is dark. On parts of the object which are partly dim/bright the day and night textures are mixed. Basically the fragment shader, instead of multiplying the day texture's RGB values by the calculated light intensity, uses the same intensity value to mix between the day and night texture's colours. Where $I = 0$ the night texture colour is shown and where $I = 1$ the day texture colour is shown, and anything inbetween is mixed.

Specular map This is a monochrome mask texture which specifies where specular lighting can occur on a surface. In the fragment shader the calculated specular intensity value $Ispecular$ is multiplied by the sampled value of the spec map, so where the map is black $Ispecular$ is multiplied by 0 and no specular highlight occurs, and where it is white it is multiplied by 1 so the specular intensity makes it into the composite intensity value I and the specular highlight occurs.

Textures are created by calling the **newTexture** function found in **gl.js** with the URL of the image to use. This creates an uninitialised WebGL texture object with an embedded HTML DOM image attribute for which the browser will automatically load the specified image into asynchronously. The returned WebGL texture object is then assigned to one of three available texture variables in a **Model** object.

The texture object cannot be initialised and used until the browser has finished loading the image file into the image object. To know if a texture can be used yet status variables are inserted into the image and texture objects. The image object's **onLoad** event is called by the browser when the image is finished loading. It sets the image's status variable to let the renderer know the texture can be initialised and used.

When the model's **toShader** function comes to want to use a specific texture, it checks the image status and only if it is ready calls **initTexture** to finish setting up the texture by binding the image to the texture object and creating mipmaps. Finally the texture object can be bound to a texture unit and be sampled in the shader.

Earth is the only model to use all three texture types. The night texture shows city lights and the specular map is used to only show specular highlights on the oceans

4.5 Lighting

The scene has a single point light (the Sun) which does not change position, intensity or colour. Therefore the program does not need to provide the shaders with any information about lights in the scene - all light information is fixed in the shader code.

Fragment lighting is calculated to a single intensity value I which is multiplied against each RGB component of the fragment's unshaded colour. The light therefore is a white colour.

This intensity value is a product of three lighting techniques: ambient, diffuse and specular. Ambient lighting is simply a minimum intensity level which is fixed so you can see the dark side of bodies even when no light falls on those fragments. The diffuse intensity is calculated from the dot product of the vertex normal and the normalised direction of the light from the vertex. Finally a specular intensity value is simply the diffuse calculation but raised to a high power to increase the cutoff level. This is not how specular lighting is traditionally done which involves the position of the camera changing where the specular highlight falls on the surface. But for what I wanted specular highlighting to achieve, which was to make the Earth's oceans appear more reflective than the land, the technique used where the camera does not have an affect seems more realistic.

As described in the texturing section earlier, which intensity components are applied may depend on which textures the model has set. Some models may have lighting disabled all together by way of setting a shader boolean uniform.

4.6 Camera

The application allows the user to zoom and rotate around the bodies in the scene. This is achieved by having a separate view matrix (as opposed to a model-view matrix) which is composed of two other matrices: a base matrix, which is created by the Body that is currently being "viewed" describing it's position and orbital rotation, and the camera matrix, which is a translation of the camera's current zoom, and rotation of pitch and yaw variables.

The resulting view matrix is passed to the shader which gets multiplied with the other matrices against all the vertices to produce the final view.

The user manipulates the camera zoom, pitch and yaw by clicking and dragging on the canvas and using the mouse wheel. These mouse events are captured by **ui.js** and new camera variables are set ready for the next update to the view matrix. The zoom distance is exponentially calculated so that the distance changes more slowly the closer the camera is to a body than it does far away, giving the user more control to see details of the bodies.

The zoom distance variable is multiplied by the current body's radius, which means that when switching between bodies each body stays the same size on the screen until the user zooms in or out with their mouse wheel. This is to prevent an issue which occurs when the user is viewing the Moon and then switches to Saturn for example. Because Saturn has a much larger radius than the Moon if the distance from the centre stayed the same the camera would end up very close to or inside Saturn.

4.7 Bloom

This is an additional feature I am particularly proud of. To make the Sun look very bright and have it's light appear to glow around not just itself but also around any object obscuring the view of it a post-processing technique known as Bloom filtering is used.

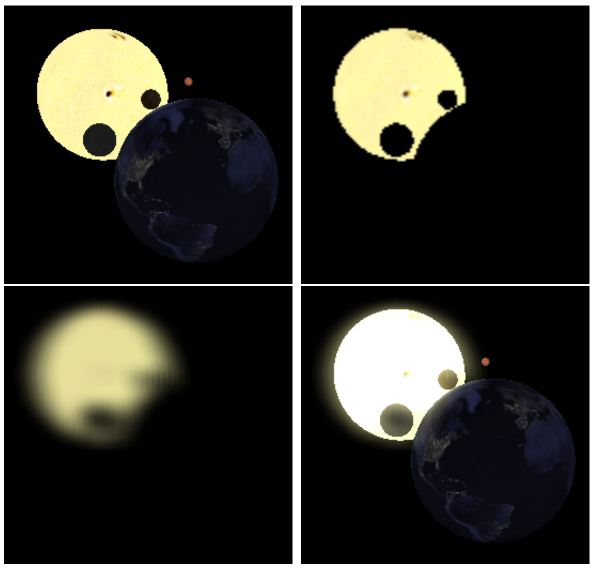
This is done in the rendering pipeline in 4 stages:

1. Render the scene as normal to an off-screen texture

2. Do the same again to another off-screen texture but only render the bodies and with lighting, texturing and colours disabled except on the Sun body which has it's texture enabled. The result is that you only see the Sun and any object in front of it is shown as a solid black circle.

3. Apply a gausian blur filter on the Sun-only texture.

4. Finally, combine the two by blitting the first texture of the normal scene to the screen and then add the colour from the blurred Sun scene texture. The result is that the blurred Sun appears on top of the final scene, making it appear to glow.



This is implemented with 4 WebGL framebuffers with textures attached for storage: one for the normal scene, one for the scene with just the Sun and two for the blurring as the Gaussian blur is a two pass process (blur vertically then horizontally). The bloom framebuffers are half the resolution of the scene framebuffer for better performance.

Also two additional shader programs exist: one to do the Guassian blur on the Sun-only texture and one to composite the original scene texture with the blurred Sun scene texture to the screen. These two shaders do not do any vertex operations as they operate solely on textures.

4.8 User Interaction

The user interacts with the program through the web browser, which creates events which Javascript can receive and handle. **ui.js** handles all user interaction and interfaces with the HTML DOM. Events from DOM entities such as mouse interaction on the WebGL canvas and clicking the various UI controls are handled and the appropriate actions are taken to control the program. It provides a clean separation of the HTML document from the program and keeps most Javascript code out of the HTML source.

5 Evaluation

I am very satisfied with the final implementation. It fulfils all of the requirements set out in the Brief and my own goals as described in the Approach. In addition I managed to implement a few more bonus features such as bloom lighting. I believe that the application could be used by anyone who wants to explore the solar system in 3D in a web browser, as it quite realistically reproduces the orbits and scales of the solar system and provides a good user experience. You can see for example the seasons of Earth change over a year as it's axial tilt remains the same as it follows it's orbit.

I have identified a few potential improvements to take it even further:

- The orbits are detailed enough to represent eccentricity and inclination, but they do not obey Kepler's second law which states that a body's velocity increases towards the point of shortest distance to the body it is orbiting and decreases towards the point of furthest distance. I researched what kind of math would be involved to implement this but decided it would be too much work to implement what would be a minor detail in a project focused on graphics rather than solar system realism.

- The camera can be fixed upon each body in the solar system by selecting it in a text list on the left of the screen. It would be cool if you could actually click on the body in the scene and have the camera jump to it. This would involve the reverse of what happens when the scene is rendered by translating the 2D screen coordinates back to 3D space and deciding if a body intersects that space whilst taking into account the depth of objects.

- It would also be nice to transition the camera gradually between different bodies so the user has a sense of where in the solar system they are moving to and how far apart the two bodies are. This may be possible by gradually interpolating the values between two view matrices defining before and after views over time.

I learnt a lot about 3D graphics and OpenGL programming during the development and found it to be a very good exercise.

The hardest part for me was writing the solar system code which went through several iterations. I had to figure out the right set and order of matrix transformations to get the bodies to follow an orbit correctly and appropriately rotate, whilst preserving the right amount of state so that it could be reused for other things like the camera view. There was a lot of trial and error but by the end I understood what effect various matrix transformations will have when applied in certain contexts and know when to break a matrix down into multiple matrices when state needs to be used for such purposes.

Shaders were also difficult to write simply because you cannot inspect their state when they are running, so you need to really think about what they are going to do and at times guess as to the data which is present. GLSL is also very restrictive compared to C for example so I learnt a lot about what is and isn't possible and where certain tasks should be done (vertex vs. fragment shader).

I put a lot of research into the theory and maths behind planetary orbits to produce the more realistic orbits, particularly on Kepler's Laws of Planetary Motion. Although not required in a graphics assignment I found it very interesting and believe it made a more interesting application to view and use.

6 Conclusion

The finished implementation is a highly polished interactive WebGL program which successfully fulfils all the requirements of the assignment brief and goes beyond it by implementing an advanced simulation of the solar system and by employing advanced graphical techniques. The design of the implementation and patterns used in it's coding makes it easy to extend and reusable for other WebGL applications. The experience taught me a lot about WebGL, Javascript programming and 3D graphics in general which will help me in future 3D programming tasks.