A Virtual Orrery

11 November, 2014

ABSTRACT – The development of a WebGL-based application simulating the solar system with a goal of realistically recreating actual dynamics in terms of lighting and planetary motion.

KEYWORDS – computer graphics, solar system, simulation, webgl.

I. Introduction

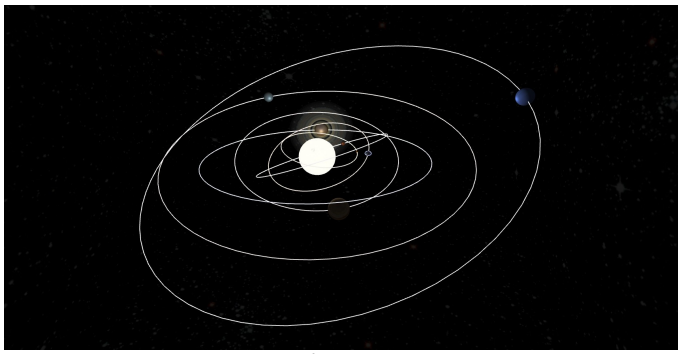

Figure 1

An orrery is a mechanical model of a solar system which demonstrates the movements of planets and moons. This 'virtual orrery' is a web application, written in JavaScript (and GSGL) using WebGL – an API for rendering 2D and 3D graphics in the web browser.

Written in an dynamic style, this application allows for users to specify their own parameters for the solar system, allowing for many different combinations of planets moving at different speeds in different ways with interesting textures (with the stipulation of textures dimensions being a power of two). The example data file given for this application is modelled on the reality of our solar system but exaggerated to make things more visible/interesting.

II. Functionality

In a bid to closely model the dynamics of a solar system, this application implements a number of features mimicking the way things are in the universe (figure 1 demonstrates some of these). This includes:

- Multiple colour maps – allowing for combining multiple texture maps.
- Spot lighting emitting from the centre of the solar system (the sun).
- Full Phong shading built up from ambient, diffuse and specular components.
- Specular map – a method of specifying where specular highlights appear (the sea on earth, for example) using texture.
- Dark map – allowing for different textures to be used for the shadowed side of planets.
- Rings on planets – utilising transparency.
- Realistic planetary motion – in accordance with Kepler's first and second laws.
- Free-roaming camera movement – allowing the user to view the scene from many different perspectives.
- Background scenery depicting deeper space (also acts as a bounding box).
- Plotted orbit paths for seeing where planet's go.
- Screenshotting for capturing those particularly scenic moments.
- 'Cubiverse mode' – a bit of an Easter egg, taking advantage of polymorphism of the inheritance chain allows planets to be converted into cubes rather than spheres.

III. Application Design

The application was designed utilising the flexibility of JavaScript's multiple paradigms. The initialisation, controls and drawing are procedural but an object orientated style is used to organise and store the items to draw.

Stars and planets come about through a chain of inheritance stemming from a basic class which defines a drawable 'item'. This item (ItemElements) consists of a series of vertices which constitute elements (faces) and corresponding element normals. There's also texture coordinates, textures for colour and specular maps, as well as animation and lighting functions which are called every time the item is drawn. It's a useful base class to have, doing lots of work in the constructor like creating many of the buffers necessary for drawing objects while maintaining a high level of flexibility.

The ItemElement class is extended by the Sphere and Cube classes which abstract it, offering simpler interfaces for their shapes and calling their super

methods with the appropriate vertices, elements, element normals and texture coordinates for their shapes. Cubes are simple and built of six elements of four vertices each, while spheres are more complex. They are constructed out of a series of twenty-sided polygons (near-circle) on top of each other, with sides of the shape connected making elements between the layers. The size of these shapes is varied by scaling the distance between the points.

A. Planets and Stars

These shapes are decorated into classes for stars and planets which add more features like dark maps and define the open functionality of lighting and animation functions using parameters passed through the constructor. The animation function is capable of modelling planetary motion using Kepler's first and second laws.

$$r = \frac{R(1+e)}{1+e\cos\theta}$$
Figure 2

$$\delta\theta = \frac{\delta t\, a R^2}{r^2}$$
Figure 3

$$\theta += \delta\theta$$
Figure 4

Where R = initial distance (distance at θ = 0), r = distance, θ = current angle of rotation, e = eccentricity, δθ = change in angle and δt = change in time (frame time) and a = velocity.

Figure 1 calculates the current distance at an angle, figure 3 provides the iteration step and figure 4 performs the iteration.

$$x = r\sin\theta$$
Figure 5

$$y = r\cos\theta$$
Figure 6

$$z = \sin\theta_z \sin\left((\theta+\theta_{offset})\, rem\left(\theta \Big/ 2\pi\, radians\right)\right)$$
Figure 7

Where θ_z = the angle tilt of the orbit and θ_{offset} = the angle at which the orbit's tilt is applied.

Figures 5, 6 and 7 are used to calculate the position of a planet.

Many of the components of these algorithms are parametrised in the constructor, allowing the user to define the distance from the sun, eccentricity of the orbit, the offset angle of rotation for the planet to begin at, the angle of the orbit (vertical movement), planet's velocity, number of days in a year, and the axis the planet rotates on. The animation available to the star class is more limited (due to the confines of realism).

Planets can have rings using a Rings class. This also inherits from ItemElements, being simply flat square plane which is textured (on both sides) using a ring texture with an alpha channel. The blend function is changed to cope with its alpha on drawing.

B. Positioning Items in Space

Manipulating the movement of objects is done with a matrix stack. This is used to store the current manipulation (pushing) or retrieve a previous one (popping). In the application the stack makes it possible to draw items relative to each other, creating a tree-like structure with different levels and different paths from those levels. This is mirrored in format of classes, where a star has an array of satellites (planets), which in turn can have array of satellites (moons) .etc.

C. View Manipulation

$$x = \sin\theta\cos\varphi$$
Figure 10

$$y = \sin\theta\sin\varphi$$
Figure 11

$$z = \cos\theta$$
Figure 12

Where θ = pitch and ϕ= yaw [1].

The view is manipulated using the mouse and keyboard controls. Movement, while smoothed in the application, is kept consistent by treating the movement as a multiple of unit vectors in the direction faced by the current perspective. The 3D space can thus be navigated using two angles (pitch and yaw) and a direction (forward/backwards).

D. Shaders

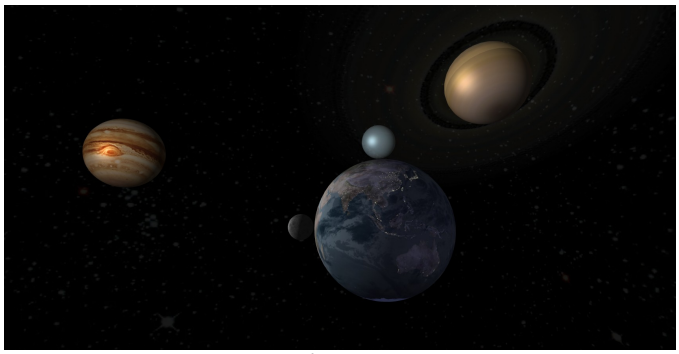

Figure 8
(From the left) Jupiter, the moon, Uranus and Saturn.

There are two shaders for this application – a vertex shader and a fragment shader. The vertex shader is fairly simple, handling only the vertices of items, while the fragment shader does the more laborious task of shading each pixel (fragment).

For each fragment, the fragment shader builds up a lighting coefficient and multiplies the sampled value (taken from the texture, colour map) at the end. This lighting coefficient is built up from the three components of the Phong shading model [2] –

ambient, diffuse and specular lighting. In cases with a specular map (a simple greyscale map which provides shininess data using intensity), this is sampled also and factored in for calculating the final light weighting.

$$m_{final} = m_{dark}((1-l)+e) + m_{colour}(l+e)$$

Figure 9

Where m_{final} = the fragment's final value, m_{dark} = the value of the dark map for that fragment, m_{colour} = the same of the colour map, l = light weighting ($0 <= l <= 1$) and e = an emissive term for overriding lighting .

If there is a dark map, the colour and dark maps are sampled and combined in a ratio relative to the lighting (figure 9, demonstrated in figure 8). In scenarios where multiple colour maps are specified, these are combined first (mean average) and then put through figure 9.

IV. Application Structure

The different components of the application are kept separated for better organisation. The HTML, JavaScript and GLSL live in different files in different folders. Anything the JavaScript needs to use is loaded via HTTP requests when needed, for example the shaders are loaded synchronously and compiled early on. Textures meanwhile are loaded asynchronously which does cause issues for the first few iterations of the drawing, but they load quickly so it's not a problem for the user.

The content of the orrery is also separated from the implementation where possible. The data for the solar system stored in a JSON file which is also loaded synchronously. This JSON file holds an array of objects describing planets with a tree-like structure. Once loaded, the file is parsed into objects for the JavaScript to use to instantiate the stars and planets of the solar system detailed in the data file.

Drawing is done using a loop which calls a function at a frequency defined by a constant variable. This function requests an animation frame, and then begins the drawing. Due to the tree structure of the stars/planets, drawing functions in those classes draw items further down their trees as well recursively, so drawing the trunk sets off drawing all the way to the leaves.

All JavaScript code is strict mode compliant and all HTML and CSS is W3C valid. It has been tested in Chromium 37 and Firefox 33 (screenshot only works on Chrome though due to browser limitations).

V. Limitations

It has been assumed that there is only one source of lighting which is at the origin (where the first star goes).

There are limitations with planets with rings. The supplied data file has Saturn last, ensuring that it is drawn last, making the blending simple. Having multiple planets with rings because of it will not work as additional calculation for which planet is nearer the view is required to get the z-buffer in the correct order.

Earth is the only planet which makes use of specular and dark maps – it isn't appropriate for other planets to have these as they tend to be either completely rocky or completely gas, and uninhabited (no lights at night).

Performance-wise, it actually does pretty well, with potential for more complex modelling, using higher fidelity lighting techniques – it runs comfortably at ~60FPS.

VI. Conclusion

The application does a fairly good job of modelling the solar system. It looks quite realistic – the lighting expressed in the shaders produces realistic looking worlds when combine with textures. The planets behave in a realistic manner also, with their motion following well defined formulae (Kepler's laws).

VII. Bibliography

[1] "Review B: Coordinate Systems", Massachusetts Institute of Technology Department of Physics, (2006-01-27). http://web.mit.edu/8.02t/www/materials/modules/ReviewB.pdf (accessed 2014-11-01).

[2] "Illumination for Computer Generated Pictures", Bui Tuong Phong, Communications of ACM 18 (1975), no. 6, 311–317.

[3] "Learning WebGL", Giles Thomas, http://www.learningwebgl.com (accessed 2014-10-15).