Solar System Explorer - An Incremental Game

Philip Prior (0920420)
Supervisor: Hayo Thielecke

Submitted in conformity with the requirements
for the degree of MSc Computer Science
School of Computer Science
University of Birmingham

Contents

Abstract

Solar System Explorer - An Incremental Game

Philip Prior (0920420)

This report details the design and implementation of a software engineering project to create a space exploration themed incremental game. The challenge was to introduce a degree of realism to the representation of space beyond that in similar products from the incremental games market.

The game was created using languages that made it suitable for delivery via a Web browser, using an incremental software development lifecycle and focusing on prototyping for testing during development.

A game named Solar System Explorer was produced, successfully incorporating several planned features including those required to meet the definition of an incremental game.

Suggestions are also provided herein as to the further development of the product beyond the timescale of this particular project.

Acknowledgments

I would like to thank my supervisor Hayo Thielecke for his advice and support in the completion of this project.

Thanks also to Uday Reddy and the rest of the teaching staff at the University of Birmingham's School of Computer science for their support and guidance.

Finally I would like to extend my appreciation to the individuals, classmates and acquaintances who assisted me in testing and providing feedback on my product.

1 Introduction

The aim of the work described in this report was to create a web based incremental game that seeks to provide a more realistic representation of the solar system than in present in other products in the genre.

It is an oft discussed issue that scientific theories and data suffer from a distortion of representation and a skew in perception in both popular media and the arts. This is common across all forms of media, from classic painting to modern forms of entertainment such as video games. Such misrepresentation hinders the public understanding of natural phenomena and is arguably more effective at (mis)educating than traditional classroom based learning. In discussing education related to the theory of evolution Brian Alters notes that, "Not only does the general public lack an understanding of evolution but so does a considerable proportion of college graduates." [4]. He goes on to explain that formal education has a very tough time combatting the informal, and often incorrect, learning

This is very evident in the discipline of astrophysics, where decades of science-fiction and science-fantasy artistic representation and storytelling have resulted in a very poor public understanding of the physical solar system, the technologies available for space exploration and the possibilities of interplanetary travel. So much so that lexicons of terms, and scales for the description of, the relative 'hardness' of such representations have sprung up in the scientific, engineering and media production communities [11]. Whilst fiction writers describe the use of 'warp drive' to navigate across interstellar distances, actual scientists are still working on the hugely chllenging problem of reaching the nearest neighbouring planets in our own solar system.

This is not to belittle the influence and inspiration to be had from speculative fiction, the number of technologies predicted or inspired by writers like Isaac Asimov or Arthur C. Clarke are well documented. However there is a great opportunity in the media representation of astrophysical phenomena to provide for a valuable learning experience at the same time as being entertained.

To this end it was considered a worthy pursuit to createl a compelling video game in a genre known as 'incremental gaming', also known as 'idle gaming' or 'clicker gaming'. This genre takes advantage of such psychological factors as accumulation desire and loss aversion placing the player in an environment that constitutes a simple Skinner box [6].

Most games of this type are delivered either as a mobile phone app or as a browser based 'web game', possibly the most famous of which is Cow Clicker, a product that parodied the repetitive gameplay of the Farmville games from publisher Zynga.

Such a product is simple enough for the core mechanics to have been achievable within the timeframe of this project whilst also providing an opportunity to address the issue of realistic representation.

2 Further background material

3 Analysis and specification

The problem was initially analysed by investigating other products in the genre, specifically focusing on a web browser game entitled 'Spaceplan' by Jake Hollands. The features and mechanical gameplay elements of these were broken down into simple lists and categorised to find conventions that span the genre. This primarily involved a card-sort exercise, derived from the technique utilised in usability design.

Given the mechanics of the incremental games model some of these features were inherent, yet some were not, for example the necessity of including some form of text based feedback at key points in progress. This text could take the form of a narrative as in a product akin to Spaceplan or as a goal/achievement flag as seen in Cookie Clicker.

As the project did not follow a standard software-development lifecycle as detailed in the project management section of this report, but instead followed a process that probably relates most closely to Agile methodologies (rapid iteration, not documentation-centric, yet eschewing test-driven-development), a simple non-exhaustive of requirements was created, including some non-functional requirements based on the heuristics provided in the GameFlow model, adapted from their approach to real-time-strategy gaming.

Very few statistics were available on the demographics of incremental game players as this information is not typically recorded as a part of play and they are usually relegated to a broader category including other 'casual games' when investigated in games research papers. With this apparent paucity of available data, very little could be done to target or analyse the requirements of any specific user group.

3.1 Example Requirements

3.2 Game specific requirements

Draw from Aleem [3]

3.3 General specification

Solar System Explorer is a desktop browser based incremental game for players of current incremental games, borrowing many conventions of the genre.

4 Design

4.1 Game design

4.2 Orbital models

4.3 Language selection

One main concern with creating Solar System Explorer was making a choice of programming language that would enable both a full implementation of the envisioned product whilst maintaining performance. The sole developer's primary skill set was in the Java language, however given the delivery platform of a web browser this essentially mandated the use of the HyperText Markup Language (HTML) and Cascading Style Sheets (CSS) in combination with JavaScript for dynamic behaviour on the client. It was possible that Java would have been useful in developing server-side functionailty, such as providing communication with a database layer but with the decision to run the code entirely on the client there was no requirement to do so.

A decision was made to conform to World Wide Web Consortium (W3C) standards as these represent the ideal model of structure in the delivery of web based content, plus compliance with standards in the manner in which web browsers parse the code should ensure a more consistent delivery across different vendor's products (e.g. Google Chrome, Mozilla Firefox, etc). No effort was to be extended to provide functionality to non-standards compliant browsers (e.g. MicroSoft Edge) through the use of proprietary code as this would add a significant overhead to development in terms of the complexity of the finished product.

HTML

As the application would require the display of graphical components with dynamic updating, the only suitable version of HTML would be 5.0 with its support for the canvas element [13]. Previous versions could display graphical information with the use of embedded applications such as Adobe Flash or Java applets, however these are considered legacy technologies, support for them is gradually being deprecated [2] and their use is discouraged in general.

CSS

With an ambition to follow standards compliant coding, it followed that the markup should attempt to be as semantically pure as possible. Which is to say that the HTML source should not contain a large number of div elements included purely as hooks for layout and styling at a later point. Due to the original focus of HTML as a documenting system with little emphasis on graphical layout it became a de facto standard practice to use the markup code as a means of arranging layout, originally by utilising tables for non-tabular data and more recently through the use of multiple sets of nested div elements with CSS positioning

applied. In many cases this is still done to maintain consistancy across non-standards compliant browsers and to provide a degree of graceful degradation in older browsers. This is very evident in some of the more popular layout libraries such as Bootstrap [10].

Given that there was no aim in this project to provide for backwards compatibility for older browsers, to follow the ideal, standards driven, approach and minimise the use of HTML elements for layout it was decided to make as much use of CSS:grid functionality as possible. This necessitated the use of Cascading Style Sheets version 3. Grid layout allows for the definition of areas as rows and columns on a developer defined grid with applicability to parent and child HTML elements to allow for a more complex graphical result. This allays the need for tablular markup or the use of complex absolute positioning, container div elements and the float attribute that was common to previous versions of CSS.

JavaScript ES6

Early exploratory development was done using a Java based sketchbook language called 'Processing' [9]. This was initially chosen with the intention of cross-compiling from Java-like source code into a JavaScript product using the p5.js library. However there are distinct differences between the capabilities of the original Processing implementation and the p5.js product. As a result other cross compilers such as JSweet [12] were investigated, but eventually this approach was abandoned in favour of starting with a baseline JavaScript solution, only utilising libraries to reduce complexity when necessary.

The most recent release of JavaScript at the point of development is JavaScript ES6, based on the ECMA International standard for ECMAScript 6 (ES6). This has enjoyed support across most desktop browsers as of March 2017 [15]. Two major benefits of using ES6 are the java-like syntax for object instantiation using classes and the availablility of variables with a block scope. Previous versions of JavaScript only used global variables whereas with the inclusion of 'let' in addition to the traditional 'var', the scope of a variable can be limited to a single code block, after which it is discarded. This also helps resolve some issues to do with JavaScript loops requiring arbitrary 'hacks' to avoid problems with enclosure.

WebGL and three.js

To provide for a complex graphical output in a web browser it was decided to use the Web Graphics Library (WebGL) capabilities of the HTML canvas element. This would allow the display of either 2-dimensional or 3-dimensional content that uses Open Graphics Library for Embedded Sysems (OpenGL ES) instructions to make the most of any available hardware acceleration through the client's Graphics Processing Unit (GPU).

The syntax for creating graphical primitives (e.g. spheres, cubes) in WebGL is relatively complex for a small return, involving the separate declaration of a number of variables before creating the object itself. It is also quite difficult to optimise for performance with a large

number of simple missteps in coding practive that could cause major issues with resultant framerates.

A number of different JavaScript libraries which provide interfaces to simplify the use of WebGL are available. One of the more potent and extensively supported of these is the three.js library [5]. It was decided to use three.js to the extent that it provides predifined objects that take constructor arguments to create elements such as spheres, lights and cameras to build a 3-dimensional output in WebGL.

4.4 Architecture

The architecture chosen was a simple client-server model, with a one off delivery of all code to the client machine at the time of the initial HyperText Transfer Protocol (HTTP) GET request to the game's Uniform Resource Locator (URL). There was no need for extended communication between the client and server beyond this point as there was no need for access to server hosted information, all game logic and data is included in the Javascript source file. In fact any such requirement would introduce needless complexity and performance restrictions. This does make the game simple to 'hack' but, given that there are no negative risks involved in someone accessing the source code, this is of no concern.

JavaScript is typically included as a single file, rather than broken down into individually localised code blocks in separate files. This is to save on request overheads and reduce server load as each HTML page, once parsed, triggers a number of asset requests for any content referred to in the file. Most usually these requests are for one or more CSS files, any embedded images and any embedded scripts. Each individual request has to travel to the server and wait in a queue to be picked up by a worker thread in the server software, thus adding more wait time until the page is ready to be fully rendered on the client. Serving the JavaScript as a single file where possible minimises this delay.

The aim was to include a single file for each of the HTML, CSS and JavaScript items, with allowance made for any libraries and image assets. Bandwidth limitations were determined to be those imposed by 'high speed' internet connections i.e. ADSL2+, 4G or faster. In practice this would mean that transfer speeds would allow for file sizes in the MegaByte range, although all effort would be made to reduce assets sizes where possible by using lossy compression algorithms for the images and minifying any large body of javascript. 'Minifying' is the term given to reducing file sizes by, at the most basic level, removing comments and whitespace characters or going as far as replacing longer, meaningful variable names with short alphanumeric sequences.

The architecture does not quite follow an Model View Controller (MVC) approach but does strive to seperate structure, presentation and logic elements into the HTML, CSS and JavaScript respectively.

5 Implementation and testing

5.1 Data types and structures

The main data types in use in Solar System Explorer are the primitives boolean, number and string and the non-primitive object.

Although these are common to most programming languages, it is worth noting that all numbers in ECMAScript are 64-bit doubles and any floating point values lack the precision of an integer format. This is important to understand as the numbers involved in calculations when it comes to the physics of the solar system can reach beyond the range at which the integer representation is exact. This was notable at times when testing functions in isolation where a trigonometric transformation would return a non-integer result when an integer was expected (e.g. a value of 1.000000000001 as opposed to 1). Whilst the project requires a good representation of reality it does not require the level of exactitude that an engineering project might so minor deviations were overlooked in this case, but calculations sometimes had to be double checked as what seemed like an inconsequential rounding error was actually indicative of an error in the order of logic.

One benefit of the use of JavaScript ES6 for a developer with a background in Java programming is the class based instantiation of objects in place of the previously standard prototype model. This allowed for the creation of 5 classes to handle the game logic and allow for the 3D graphical model: Resource, UpgradeEvent, StellarObject, Anchor and Orbit.

The Resource and UpgradeEvent classes create the Objects for the incremental game logic.

A Resource object has properties for the amount of a particular resource that is available at any given time and the rate of change for that resource, along with simple methods for increasing the amount by a value passed as a parameter or increasing the rate of change. There are no negative changes to the values as this is a core feature inherent to incremental game design.

The UpgradeEvents are a set of objects which are continuously checked for pre-requisite conditions which, when met, trigger the visibility of upgrades and store the costs of said upgrades alongside any flavour text that needs to be displayed as and when the event becomes visible.

The Anchor, Orbit and StellarObject classes are used to generate the visual display of the solar system in the 3-dimensional canvas element.

An anchor is a common feature in any system of 3D representation, sometimes referred to as null objects. They simply represent a point in the space with an x,y and z coordinate and values for rotation about the axes. This is most often used as a simple reference by which to parent other objects or as a 'control point' in kinematic animation systems.

The Orbit class takes data common to the scientific description of an orbit about a point (e.g. radius, eccentricity, period, etc.), stores those properties in a manner most

useful for further calculation and display (e.g. converting input angles given in degrees to radians, automatically calculating the aphelion and perihelion values) and provides an update method which allows for animation of an object around the orbit by incrementing a theta value. This update method includes rotational transformation in 3D space and is discussed in futher detail in the next subsection.

The StellarObject class has properties representing the metadata for any object in space. In the demonstration content at the point of delivery, this was used to generate the Sun and the planets, however it could also be used to provide data for any object in the scene. One iteration used this class to generate moons for the planets. However, given that the relative orbital rotation of the moons was so fast as to make them practically invisible, they were removed for the time being.

In future iterations these 3 classes could be used to generate objects that appear in response to user input (i.e. purchasing upgrades).

JavaScript ES6 is the first release to contain pre-defined collections in the form of maps and sets. These were investigated as potential data structures for the objects instantiated by the above classes, however it was discovered that the performance of these implementations is far slower in current web broswers than using baseline JavaScript when performing lookups [8]. For this reason and the fact that data will not be added by users from outside of the program itself it seemed most sensible to work with simple arrays of objects rather than add the complication of implementing abstract data types. The largest collection of data in the game is the array of upgrade events, but this requires constant interation over the objects to make conditional checks against the pre-requisite event values, so the performance gains from using a data structure with a different method of retrieval would likely be marginal, even if it were faster at finding any single entry (e.g. utilising a hash table).

5.2 Example method - orbital rotation in 3D space

The model used in WebGL for mapping objects in 3-dimensional space is a cartesian coordinate system with an x, y and z value describing any single point. Most data that describes the movement of astronomical bodies is given as a series of angles and distances from a point of rotation as they are usually ellipses or parabola.

To translate from the information given in the physics to that required by the program involves the use of trigonometry. At a basic level, e.g calculating the x and z values for a point on a circle on the x,z plane, this is a relatively simple application of the formulae:

$$x = radius * \sin\theta$$

$$z = radius * \cos\theta$$

In an ellipse centred at its origin the radius is subsituted for the semi-major and semi-minor axis values, in astrophysics terminology these are the equivalent of the aphelion and

perihelion, terms which describe the closest and furthest orbital distance from the parent body. (Note that this is only true for orbits where the centre and eccentricity of the ellipse are such that the aphelion and perihelion are perpendicular)

$$x = aphelion * \sin\theta$$

$$z = perihelion * \cos\theta$$

However, most orbits exist at an incline to the ecplitic, a term used for the plane which is coplanar with the orbit of the Earth around the Sun. They are also not all uniformly rotated in relation to the Sun. This necessitates the inclusion of the ability to rotate the orbital ellipse about each of the 3 axes.

Rotation of a point about an axis by a value (θ') given in radians requires application of the following formulae [1].

Rotation about the x axis:

$$z' = y * \sin\theta' + z * \cos\theta'$$

$$y' = y * \cos\theta' - z * \sin\theta'$$

Rotation about the y axis:

$$x' = z * \sin\theta' + x * \cos\theta'$$

$$z' = z * \cos\theta' - x * \sin\theta'$$

Rotation about the z axis:

$$y' = x * \sin\theta' + y * \cos\theta'$$

$$x' = x * \cos\theta' - y * \sin\theta'$$

These can be applied consecutively if there is a requirement to rotate about more than one axis.

When coded as a part of the update() method in the Orbit object care had to be taken not to overwrite previous x,y and z values until they were ready for assignment, so some temporary variables with a block scope were included for instantiation within each conditional check for rotation. Simple addition of the coordinates of any parent orbit allowed for rotation of chains of parent and child objects. If no parent was defined or there was no need for the parent to be an Orbit itself, a simple Anchor object could substitute.

```
class Orbit {

        [...]

        update() {
                // Rotate the value of theta by one measure
                this._theta += this._deltaTheta;
                // calculate the basic x,y,z cartesian coordinates
                // assuming we start by generating on a plane level with the
                    sun->earth axis
                let localX = (Math.sin(this._theta)*this._aphe);
                let localZ = (Math.cos(this._theta)*this._peri);
                let localY = 0;
                // Rotate the coordinates based on the rotation of the orbit
                    around its parent
                if(this._rotX!==0){
                        let temp_z = (Math.sin(this._rotX)*localY) + (Math.cos
                            (this._rotX)*localZ);
                        let temp_y = (Math.cos(this._rotX)*localY) - (Math.sin
                            (this._rotX)*localZ);
                        localZ = temp_z;
                        localY = temp_y;
                }
                if(this._rotY!==0){
                        let temp_x = (Math.sin(this._rotY)*localZ) + (Math.cos
                            (this._rotY)*localX);
                        let temp_z = (Math.cos(this._rotY)*localZ) - (Math.sin
                            (this._rotY)*localX);
                        localX = temp_x;
                        localZ = temp_z;
                }
                if(this._rotZ!==0){
                        let temp_y = (Math.sin(this._rotZ)*localX) + (Math.cos
                            (this._rotZ)*localY);
                        let temp_x = (Math.cos(this._rotZ)*localX) - (Math.sin
                            (this._rotZ)*localY);
                        localY = temp_y;
                        localX = temp_x;
                }
                this._x = localX + this._parent.getX();
                this._z = localZ + this._parent.getZ();
                this._y = localY + this._parent.getY();
        }
}
```

Listing 1: Implemented code for rotation in 3D space on an update() call

5.3 Unit, prototype and integration testing

There are limited options available when it comes to automated or scripted unit testing for JavaScript. With such a fast moving language, which often uses proprietary frameworks and libraries that virtually constitute domain specific languages (DSLs), it can be difficult for developers of test suites to keep up with the latest trends. The most tried and tested method appears to be maually unit testing i.e. isolating a block of code, be it a function, method or full script and create another short piece of JavaScript to test for the expected return by outputting this to the console [14].

For segments of the program where the output was not immediately verfiable as correct, e.g. when performing rotational translation as in the previous section, the method of testing was to save the function into a separate .js file, provide sample inputs and check for the correct outputs by sending them to the console in versions of both the Chrome and Firefox browsers on both machines that formed the test bed (One mid-range Core i5 laptop, one high-end Core i7 desktop).

Most of the functionality behind the game logic in this program is simple arithmetic and thus testing was usually not required. Methods that merely incremented a value or performed an addition operation were assumed to be working as expected.

The most challenging parts of the code were those that defined the behavour of on screen elements and that captured user interaction, neither of which can be fully verified using unit testing scripts. Technically mouse clicks can be simulated, but the effort required to do so and adequately measure correct outputs would mean that the main body of the project would become about writing automated tests rather than producing a game. An automated test cannot, for instance, tell you whether a texture has rendered on the surface of a sphere, wrapping in the correct direction or judge the correct opacity of a translucent background.

Most of these elements were tested by building prototypes that modelled the correct behaviour prior to integrating them into the main body of the program. For example in the first 5 weeks of development, the WebGL code for the 3D section of the UI was developed in a separate file structure, away from the game logic.

Integration tests were done using the traditional games development technique of play-throughs as a part of the quality assurance phase [7]. This is one clear way in which the development lifecycle of a game differs significantly from that of office applications or data-processing applications. At the point of integration specific non-functional behaviours were verified, such as the ability to render the animation at a consistent framerate on a mid-range PC and the legibility of text imposed over the image of a starfield.

5.4 Acceptance testing

A body of 12 users volunteered to assist with acceptance testing. Given that the product has no clearly defined audience demographic, these were left to self select into participation by clicking a link that automatically appeared in the user-interface at the end of a play-through.

The users were asked to complete a very short multiple choice questionnaire which related specifically to the aspirational goals of the product. They were subsequently invited to leave free-form comments relating to both the product and to elaborate on their multiple choice answers. This was done via a Google Form which automatically added responses to a speadsheet and generated graphical reports demonstrating the results.

6 User interface

6.1 Incremental game conventions

6.2 General UI conventions

6.3 Aesthetic

7 Project management

Tools, time management, meetings.

8 Results and evaluation

9 Discussion

10 Conclusion

References

[1] ACM SIGGRAPH Education Committee. *3D Rotation*. HyperGraph. June 6, 2005. URL: https://www.siggraph.org/education/materials/HyperGraph/modeling/mod_tran/3drota.htm (visited on 08/15/2017).

[2] Adobe Corporate Communications. *Flash & The Future of Interactive Content*. Latest company news & updates — Adobe Conversations Blog. July 25, 2017. URL: https://blogs.adobe.com/conversations/2017/07/adobe-flash-update.html (visited on 07/28/2017).

[3] Saiqa Aleem, Luiz Fernando Capretz, and Faheem Ahmed. "Game development software engineering process life cycle: a systematic review". In: *Journal of Software Engineering Research and Development* 4.1 (2016), p. 6. ISSN: 2195-1721. DOI: 10.1186/s40411-016-0032-7. URL: http://dx.doi.org/10.1186/s40411-016-0032-7.

[4] Brian J. Alters and Craig E. Nelson. "Perspective: Teaching Evolution in Higher Education". In: *Evolution* 56.10 (Oct. 2002), pp. 1891–1901. ISSN: 1558-5646. DOI: 10.1111/j.0014-3820.2002.tb00115.x. URL: http://onlinelibrary.wiley.com/doi/10.1111/j.0014-3820.2002.tb00115.x/abstract (visited on 07/12/2017).

[5] Ricardo Cabello. *three.js - Javascript 3D library*. URL: https://threejs.org/ (visited on 08/10/2017).

[6] Neil R Carlson et al. *Psychology: the Science of Behaviour, Fourth Canadian Edition with MyPsychLab*. OCLC: 820144778. Pearson Education Canada, 2009. ISBN: 978-0-205-70286-2.

[7] Heather Maxwell Chandler. *The Game Production Handbook*. Google-Books-ID: laiOw5WkdEcC. Jones & Bartlett Publishers, 2009. 510 pp. ISBN: 978-1-934015-40-7.

[8] Kevin Decker. *Six Speed*. Performance of ES6 features relative to the ES5 baseline operations per second. 2017. URL: http://incaseofstairs.com/six-speed/ (visited on 08/05/2017).

[9] Ben Fry and Casey Reas. *Processing.org*. Processing. URL: https://processing.org/ (visited on 07/01/2017).

[10] *Get started with Bootstrap*. Bootstrap. In collab. with Mark Otto and Jacob Thornton. URL: https://getbootstrap.com/docs/4.0/getting-started/introduction/ (visited on 08/14/2017).

[11] *Mohs Scale of Science Fiction Hardness - TV Tropes*. TV Tropes. URL: http://tvtropes.org/pmwiki/pmwiki.php/Main/MohsScaleOfScienceFictionHardness (visited on 08/01/2017).

[12] Renaud Pawlak. *JSweet: a transpiler to write JavaScript programs in Java*. JSweet. URL: http://www.jsweet.org/ (visited on 07/22/2017).

[13] The World Wide Web Consortium (W3C). *HTML5*. HTML5 - A vocabulary and associated APIs for HTML and XHTML. Oct. 28, 2014. URL: https://www.w3.org/TR/html5/ (visited on 07/13/2017).

[14] Jorn Zaefferer. *Introduction To JavaScript Unit*. June 27, 2012. URL: https://www.smashingmagazine.com/2012/06/introduction-to-javascript-unit-testing/ (visited on 08/14/2017).

[15] Juriy Zaytsev. *ECMAScript 6 compatibility table*. 2017. URL: https://kangax.github.io/compat-table/es6/ (visited on 08/16/2017).

Appendices

1 Image licences

ESO image licence NASA image licence

2 Framework licence

three.js licence

Declaration

The material contained within this thesis has not previously been submitted for a degree at the University of Birmingham or any other university. The research reported within this thesis has been conducted by the author unless indicated otherwise.

Signed .....