SYDENHAM AND FOREST HILL SIXTH FORM

n-body system simulator

AQA Computer Science A-Level – the computing practical project

Joe Binns

(2017 – 2018)

AQA Computer Science A-Level – the computing practical project (2017 – 2018)
n-body system simulator
Joe Binns

Contents

AQA Computer Science A-Level – the computing practical project (2017 – 2018)
n-body system simulator
Joe Binns

1. Analysis

Problem area:

The *n*-body problem is a physical problem in predicting the dynamic motions of particles in a system. *n*-body systems are fundamental to physics. The most common example of its application is astrophysical, predicting the motions of astronomical bodies under the influence of gravity. The problem is iterative in nature and may involve hundreds of thousands of considerable particles (e.g. solar systems in a galaxy OR all known bodies in our solar system). A direct approach to the simulation is where the force between each particle and every other particle is calculated, this has a complexity of $O(n^2)$, where n is the number of particles. This is problematic, especially considering that these calculations would be repeated for every 'step' forward in time. I will therefore explore reducing the complexity of my simulation whilst minimising any loss of accuracy.

Research:

My reading into the problem includes 'The Feynman Lectures on Physics - Volume I" by Richard Feynman et al, in which dynamical equations and a simplified direct particle-particle method are introduced in chapter 9 (Newton's Laws of Dynamics). I have also read into the problem in "The Fundamentals of Astrodynamics" by Roger R. Bate et al, in which a similar methodology is applied to a two-body problem. Finally, I have read a seminar by Tancred Lindholm in which possible approaches to the problem and complexity reduction are discussed.

I will be directly sourcing data from https://ssd.jpl.nasa.gov/?horizons for the initial properties of astronomical bodies for each simulation.

End user introduction:

My primary end users will be the Cambridge University and the Royal Holloway University departments of Physics. Both university departments have students undertake a substantial project in the third year of the undergraduate degree course. An example given to students of what they could attempt is the *n*-body problem. The undergraduates are limited to producing simplistic models, this is due to results most commonly being displayed on a two dimensional Cartesian coordinate system. This further limits the applications of the program since the viewpoint, orientation and zoom are fixed. The universities have an interest in improving the future outcomes of the project by overcoming the stated limits. My program will act as a well-documented example project which the universities will be able to give to future undergraduates. I will write the program in the C# language and use the Unity Engine for the Graphical User Interface (GUI) to overcome the visual limitations addressed in this paragraph.

Objectives[1]:

Lecturers from each of the Universities and myself have decided on the following objectives…
1. Produce a working *n*-body simulation (Acceptable limit: *maximum* error of 5% in any particles' position after 1 year's simulation (in any of the 3 Cartesian planes) (compared to NASA Horizons forecasted or real values)).
2. Three-dimensional display of particle positions and trails.
3. Option for user to define variables (e.g. Time between particles' position calculations).
4. NASA Horizons database interface (allowing choice of initial particles).
5. Select a body to view information regarding it and to provide the functionality of an observatory view from any position on the particle's surface.

Data modelling:

NASA Horizons database access…
Either a LINUX computer hosting the Telnet interface which is accessed through the program, Telnet interface directly used through windows via the program (must be enabled on each users' computer) or directly accessing the web-interface through the program.

For one method of reducing the complexity of the calculations (to allow for more calculations to be performed per second and thus a lesser error in position), I could use an octree in data processing to minimise the number of calculations made. Each particle will be treated as a node on a tree. The tree is divided into octants (quadrants with third dimension considered) if more than one particle is in any single cube. This process is repeated until only one particle is in any single cube. Every node is estimated to have a centre of mass in the centre of its own cube. To calculate the force on a particle, a ratio of $^{\text{quadrant length}}/_{\text{distance between particles}}$ is calculated for every other node, if the ratio is greater than a certain value (which determines the accuracy) then the other node is treated as a single body (even if it contains multiple particles), and the node is not traversed further. This reduces complexity from $O(n^2)$ to $O(n\log n)$ whilst only slightly hindering the accuracy (since only large distance calculations (with typically lesser gravitational influence) will be approximated to a greater degree.

[1] The words 'objective' and 'success criteria' are used interchangeably throughout this document.

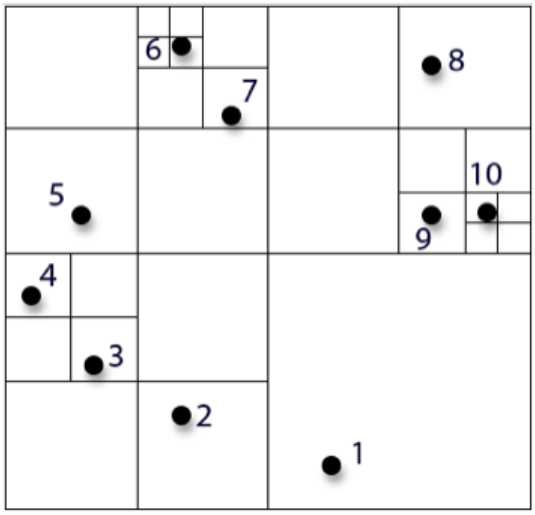

Fig 1 – Quad tree (2D)

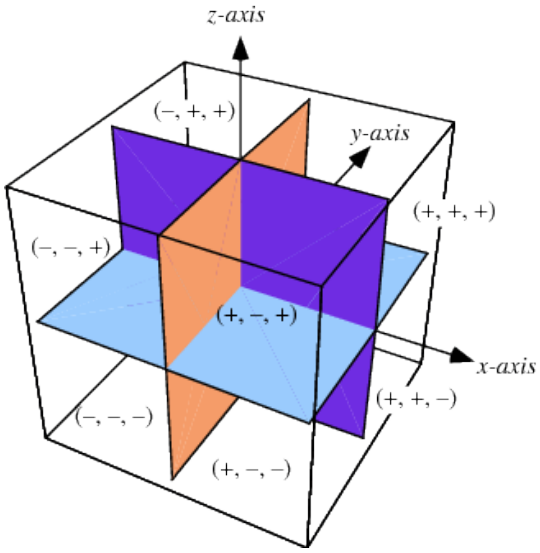

Fig 2 – Octree (3D)

2. Documented design

Each script below is for an individual class. Object Oriented Programming (OOP) is consistent for nearly all scripts used in the Unity Engine.

Each particle will act as an object, an instance of a given class with its own unique attributes. Position (in the x, y and z planes), velocity (in the x, y and z planes) and mass for each body will be required attributes for calculating the updated positions. The mean radius attribute of each body will be required for rendering the objects in approximate scale. This is all achieved by the **Body Script** class below.
The calculations will be performed by a single instance of the calculation class, **System Script**, below.
Camera movement operations will be controlled by a single instance of the camera class, **Camera Script**, below. The user interface (including inputs, processing and the GUI) will also be controlled by this script.

Algorithms:

N-BODY PROBLEM (System Script)

This class's algorithm will calculate and set the new positions of each body over a pre-set time step. It will be repeated until stopped (either due to the user or due to all calculations for a desired time period being complete).
In order to do this, the force due to gravity on each body will be calculated. The acceleration of each body will then be calculated by dividing each force by the respective mass. The positions will then be updated using the time step between sets of calculations, the previous acceleration and the previous velocity.

START System Script
INPUT time step
Gravitational Constant = 6.67408e-20
running = True
WHILE running == True
    FOR every body i
        GET i's Body Script
        GET i's mass FROM i's Body Script
        GET i's x, y and z positions FROM i's Body Script
        i's distance = $\sqrt{x^2 + y^2 + z^2}$
        force working sum of x, y and z = 0
        FOR every body j WHERE j !== i
            GET j's Body Script
            GET j's mass FROM j's Body Script
            GET j's x, y and z positions FROM j's Body Script
            j's distance = $\sqrt{x^2 + y^2 + z^2}$
            distance between i and j = $|j\text{'s distance} - i\text{'s distance}|$
            force working sum of x += $\frac{Gravitational\ Constant \cdot i\text{'s mass} \cdot j\text{'s mass} \cdot (i\text{'s } x - j\text{'s } x)}{(distance\ between\ i\ and\ j)^3}$
            force working sum of y += $\frac{Gravitational\ Constant \cdot i\text{'s mass} \cdot j\text{'s mass} \cdot (i\text{'s } y - j\text{'s } y)}{(distance\ between\ i\ and\ j)^3}$
            force working sum of z += $\frac{Gravitational\ Constant \cdot i\text{'s mass} \cdot j\text{'s mass} \cdot (i\text{'s } z - j\text{'s } z)}{(distance\ between\ i\ and\ j)^3}$
        END LOOP
        i's x acceleration = $\frac{force\ working\ sum\ of\ x}{i\text{'s mass}}$
        i's y acceleration = $\frac{force\ working\ sum\ of\ y}{i\text{'s mass}}$
        i's z acceleration = $\frac{force\ working\ sum\ of\ z}{i\text{'s mass}}$
        GET i's x, y and z velocities FROM i's Body Script
        i's x velocity += $i\text{'s } x\ acceleration \cdot time\ step$
        i's y velocity += $i\text{'s } y\ acceleration \cdot time\ step$
        i's z velocity += $i\text{'s } z\ acceleration \cdot time\ step$
        SET i's Body Script x, y and z velocities = i's x, y and z velocities
        i's x position += $i\text{'s } x\ velocity \cdot time\ step$
        i's y position += $i\text{'s } y\ velocity \cdot time\ step$
        i's z position += $i\text{'s } z\ velocity \cdot time\ step$
        SET i's Body Script x, y and z positions = i's x, y and z positions
        TRANSFORM i's POSITION TO i's x, y and z positions
    END LOOP
    INPUT running (ASSUME True IF NO INPUT)
END LOOP
END System Script

Fig 3 – System Script pseudocode

BODY (Body Script)
This class's algorithm will initiate positions and velocities as well as other variables. This will occur for every instance of the class, every particle, only once (at initialisation).

```
START Body Script
GET x, y and z positions FOR THE BODY IN THE GIVEN SYSTEM, FROM THE DATABASE
GET x, y and z velocities FOR THE BODY IN THE GIVEN SYSTEM, FROM THE DATABASE
GET mean radius FOR THE BODY IN THE GIVEN SYSTEM, FROM THE DATABASE
GET mass FOR THE BODY IN THE GIVEN SYSTEM, FROM THE DATABASE
TRANSFORM THE BODY'S POSITION TO x, y and z positions
END Body Script
```

Fig 4 – Body Script pseudocode

CAMERA MOVEMENT (Camera Script)
This class's algorithm will react to user inputs regarding camera movement and the user interface.

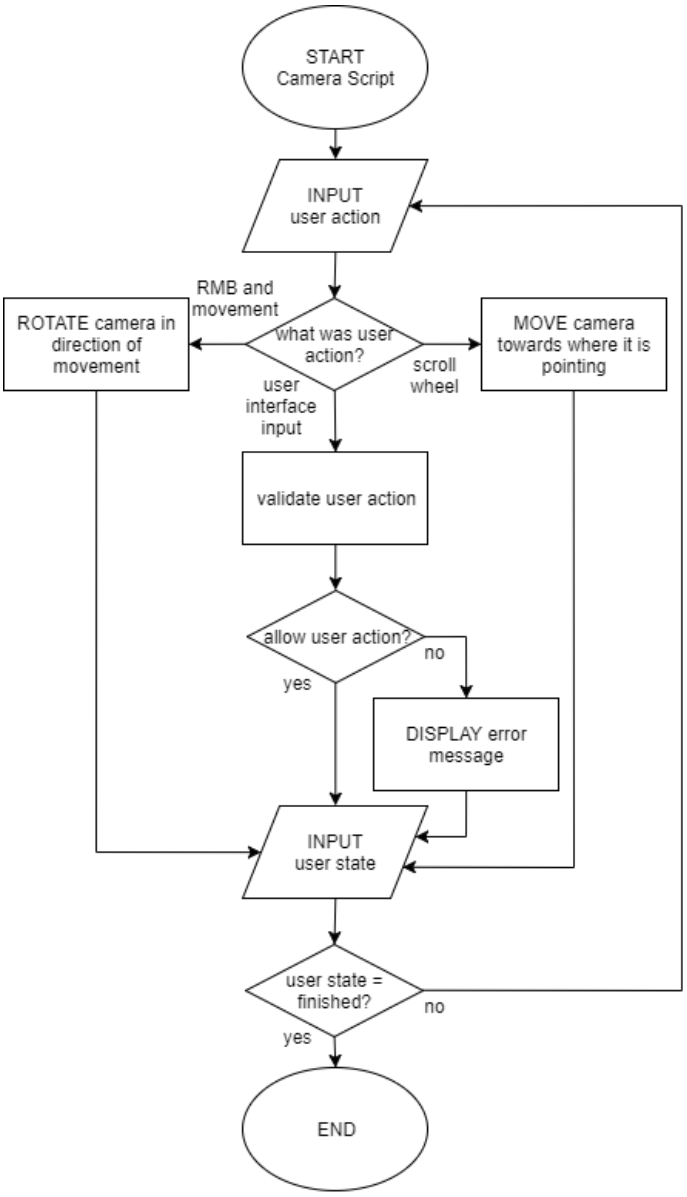

Fig 5 – Camera Script flowchart

Trace tables:
N-BODY PROBLEM and BODY

This trace table will demonstrate a 3-body system, an example of which would be the Sun-Earth-Moon system. The huge amount of processes taking place here demonstrates the great complexity of an *n*-body simulator. Recall that

this process will be squarely greater for additional bodies and that the process will be repeated as frequently as possible, this is why it is crucial for me to reduce the complexity of the algorithms used.

Body Script:

Line #	*pos*	*vel*	*mean rad*	*mass*
1	(0, 0, 0)			
2		(0, 0, 0)		
3			10000	
5				20000

Fig 6 – Body Script trace table (body 1)

Line #	*pos*	*vel*	*mean rad*	*mass*
1	(88, -42, 1)			
2		(10e-10, 0, 0)		
3			10	
5				0.06

Fig 7 – Body Script trace table (body 2)

Line #	*pos*	*vel*	*mean rad*	*mass*
1	(89, -41, 1)			
2		(11e-10, -1.0, 0)		
3			1	
5				0.0007

Fig 8 – Body Script trace table (body 3)

System Script:

Line #	τ	G	running	i	$mass_i$	pos_i	$dist_i$	$\sum Force$	j	$mass_j$	pos_j	$dist_j$	$dist_{ij}$	acc_i	vel_i
1	60														
2		7e-20													
3			True												
5				1											
7					20000										
8						(0, 0, 0)									
9							0								
10								(0, 0, 0)							
11									2						
13										0.06					
14											(88, -42, 1)				
15												97.5			
16													97.5		
17, 18, 19								(-8.0e-21, 3.8e-21, -9.1e-23)							
11									3						
13										0.0007					
14											(89, -41, 1)				
15												98.0			
16													98.0		
17, 18, 19								(-8.1e-21, 3.8e-21, -9.2e-23)							
21, 22, 23														(-4.1e-25, 1.9e-25, -4.6e-27)	
24															(0, 0, 0)

25, 26, 27															(-2.5e-23, 1.1e-23, -2.8e-25)
29, 30, 31						(-1.5e-21, 6.6e-22, -1.7e-23)									
5				2											
7					0.06										
8						(88, -42, 1)									
9							97.5								
10								(0, 0, 0)							
11									1						
13										20000					
14											(0, 0, 0)				
15												0			
16													97.5		
17, 18, 19								(8.0e-21, -3.8e-21, 9.1e-23)							
11									3						
13										0.0007					
14											(89, -41, 1)				
15												98.0			
16													1.41		
17, 18, 19								(8.0e-21, -3.8e-21, 9.1e-23)*							
21, 22, 23														(1.3e-19, -6.3e-20, 1.5e-21)	
24															(10e-10, 0.0e+0, 0.0e+0)
25, 26, 27															(10e-10, -6.3e-20, 1.5e-21)*
28						(88, -42, 1)*									
5				3											
7					0.0007										
8						(89, -41, 1)									
9							98.0								
10								(0, 0, 0)							
11									1						
13										20000					
14											(0, 0, 0)				
15												0			
16													98.0		
17, 18, 19								(9.3e-23, -4.3e-23, 1.0e-24)							
11									2						
13										0.06					
14											(88, -42,				

											1)				
15												98.0			
16													1.41		
17, 18, 19								(9.4e-23, -4.2e-23, 1.0e-24)							
21, 22, 23														(1.3e-19, -6.0e-20, 1.4e-21)	
24															(11e-10, -1.0e+0, 0.0e+0)
25, 26, 27															(1.1e-9, -1.0e+0, 8.4e-20)*
28						(89, -101, 1)*									

Fig 9 – System Script trace table

*Note that values have been rounded to two or three significant figures. Therefore, some changes in the values cannot be seen here due to the large differences in magnitude. The program will not have this issue since it will store values to a much greater accuracy.

Data structures:

I have considered my options for data structures, selecting the most appropriate and efficient data structures will be crucial to the programming stage and the performance of my program.

Data Structure	Occurrence	Purpose
Class	System Script	Encapsulates data. The inheritance offered by OOP will be crucial for the transfer of each body's attributes.
	Body Script	
	Camera Script	
List	Position	Store the respective x, y and z plane values. This will allow me to use iteration to shorten and better organise the code, making debugging easier.
	Velocity	
	Acceleration	
	Force	
Tree	Octree	For optimisation, as described in Analysis.

Fig 10 – Data structures table

Entity Relationship Diagram:

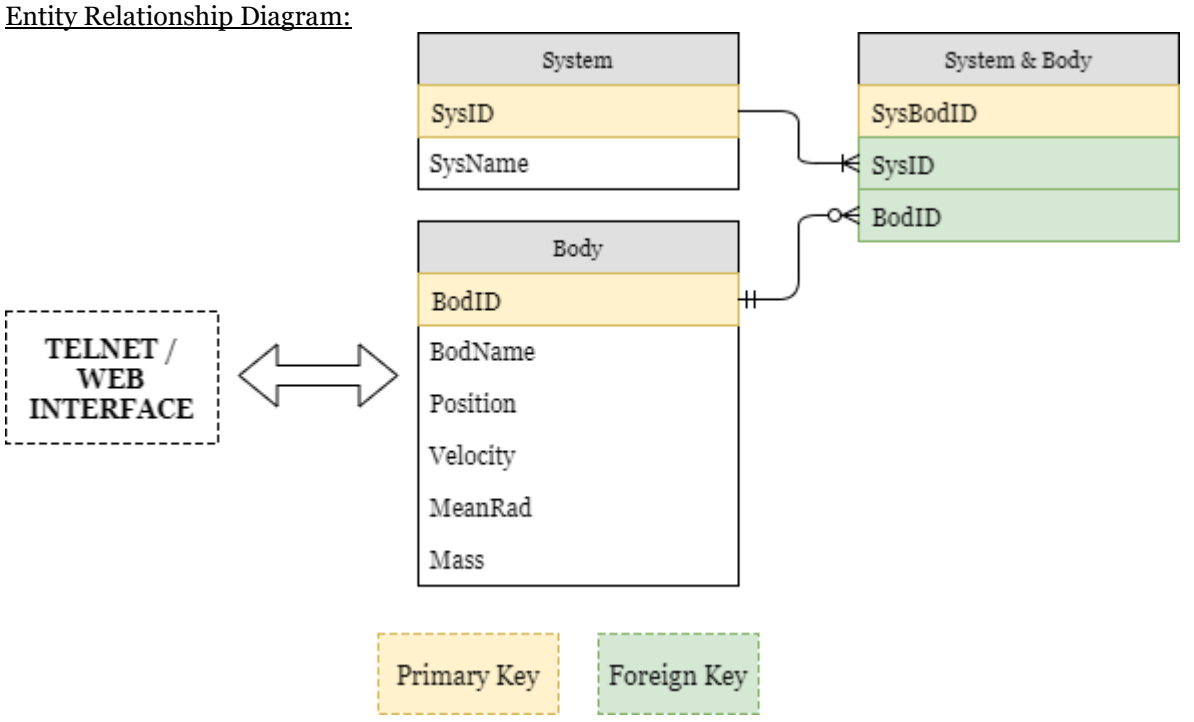

Fig 11 – Entity Relationship Diagram

Requests will be sent for a given BodName to the TELNET / WEB INTERFACE, given predetermined settings. The NASA Horizons database will then return the requested Body with its given attributes.

Graphical User Interface mock-up:

I have produced Graphical User Interface (GUI) mock-ups for a variety of different scenarios. These demonstrate the user's potential interactions with the program and have allowed me to refine my own view regarding user interaction and necessary user inputs.

Fig 12 – Concept GUI for free roam mode.

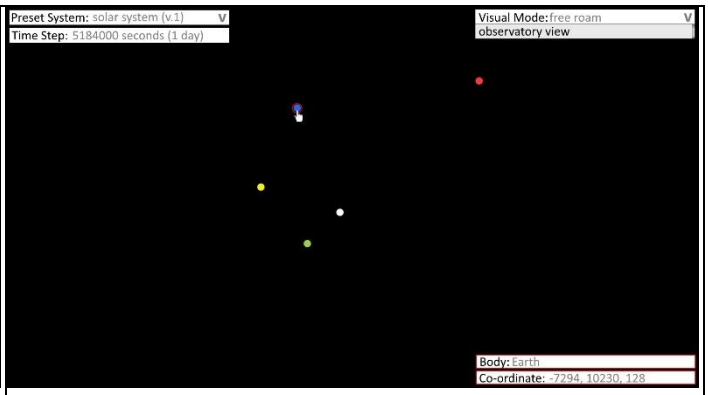

Fig 13 – Demonstrating a different orientation and distance.

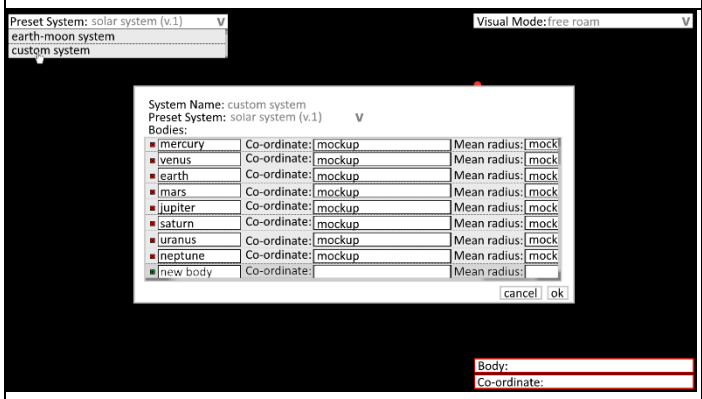

Fig 14 – Concept GUI 'pop-up' for creating a custom system of bodies.

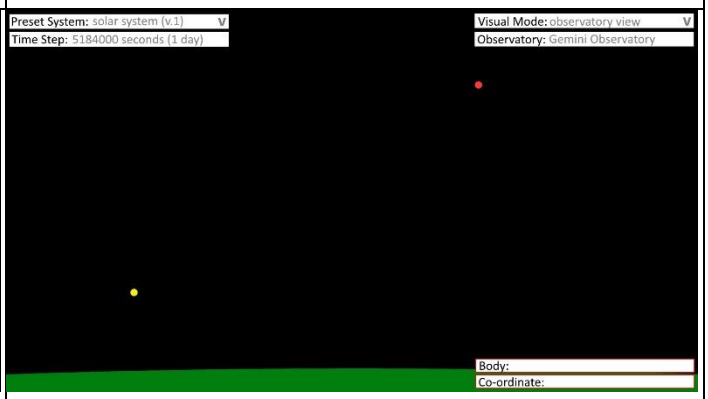

Fig 15 – Concept GUI for observatory view. The green shows the curvature of the body's surface, acting as a limiting horizon.

Problems I have encountered and amended as of completion of the Design Stage:

The trace tables have brought up some issues with the System Script. I now realise that a logical error will take place in which each body's position and velocity will be updated during each set of calculations. This is problematic since all the bodies attractions on all other bodies should be considered prior to changing any values. I will amend this by introducing 'hold' variables, which will act to hold the bodies updated position and velocity until the entire set of calculations is complete.

I have also noticed some potential repetitions of calculations in the System Script. I will amend this by exploring with altered methods of calculations. I believe this issue will be naturally overcome once the Octree and other complexity altering methods are fully implemented.

3. Technical solution

Throughout the development of my program, I will need to constantly refer to my success criteria in order ensure the appropriate functionalities are available to the user.

Determining which interface to use:

Telnet:

I was initially inclined to using the Telnet-interface. I began by attempting to implement a C# telnet script, with influences from telnet libraries such as MinimalisticTelnet. I encountered problems in doing this, since most these libraries appeared to be built for a specific use. For example, all the libraries were built on the assumption that a login is required, which is not the case for NASA JPL Horizons. Also, many of these libraries function around a terminal window, this is incompatible with the Unity engine and thus a different approach was required.

E-mail:

I then looked into the potential of using the E-mail interface. I could successfully send E-mails but found that I could not receive the respective returning E-mails through my program without creating an entire E-mail client.

Web-Interface:

NASA JPL Horizons Web-interface was my final and most complex option. I didn't look much into this after discovering how much work this could potentially require.

Given my opinions regarding my options, I decided the most appropriate approach would be to continue developing my telnet connection.

Telnet Script (a.k.a. 'TelnetTesting' Script):

I have now developed a working Telnet Script (in addition to the previous scripts described in the design stage). It sends a set of standard messages, using variables (such as the body's identifier) which are called from a list into a subroutine, where necessary. Between sending each message, the server responds by presenting values (if appropriate) and preparing for the next input.

Since the telnet server's responses are presented in dynamic forms, which depend upon the type of body (e.g. Star, gas giant, dwarf planet, moon (etc.), all of which have different output formats), I will need to create a dynamic way of scraping data from the raw outputs.

Some example telnet responses are shown below.

```
*******************************************************************************
 Revised : Jul 31, 2013              Sun                      10

 PHYSICAL PROPERTIES (revised Jan 16, 2014):
  GM (10^11 km^3/s^2)  = 1.3271244004193938  Mass (10^30 kg)   ~ 1.988544
  Radius (photosphere)  = 6.963(10^5) km  Angular diam at 1 AU  = 1919.3"
  Solar Radius (IAU)    = 6.955(10^5) km  Mean density         = 1.408 g/cm^3
  Surface gravity       =  274.0 m/s^2    Moment of inertia     = 0.059
  Escape velocity       =  617.7 km/s     Adopted sidereal per  = 25.38 d
  Pole (RA,DEC in deg.) =  286.13,63.87   Obliquity to ecliptic = 7 deg 15'
  Solar constant (1 AU) = 1367.6 W/m^2    Solar lumin.(erg/s)   =  3.846(10^33)
  Mass-energy conv rate = 4.3(10^12 gm/s) Effective temp (K)    =  5778
  Surf. temp (photosphr)= 6600 K (bottom) Surf. temp (photosphr)=  4400 K (top)
  Photospheric depth    = ~400 km         Chromospheric depth   = ~2500 km
  Sunspot cycle         = 11.4 yr         Cycle 22 sunspot min. =  1991 A.D.

  Motn. rel to nrby strs= apex : RA=271 deg; DEC=+30 deg
                         speed: 19.4 km/s = 0.0112 AU/day
  Motn. rel to 2.73K BB = apex : l=264.7+-0.8; b=48.2+-0.5
                         speed: 369 +-11 km/s
*******************************************************************************
```

Fig 16 – The Sun's body details

```
*******************************************************************************
 Revised: Jul 31, 2013          Mercury                  199 / 1

 GEOPHYSICAL DATA (updated 2008-Feb-07):
  Mean radius (km)    = 2440(+-1)   Density (g cm^-3)    = 5.427
  Mass (10^23 kg )    =   3.302     Flattening, f        =
  Volume (x10^10 km^3) =   6.085    Semi-major axis      =
  Sidereal rot. period =  58.6462 d  Rot. Rate (x10^5 s)  = 0.124001
  Mean solar day      =  175.9421 d  Polar gravity ms^-2  =
  Mom. of Inertia     =   0.33      Equ. gravity  ms^-2   = 3.701
  Core radius (km)    = ~1600       Potential Love # k2   =

  GM (km^3 s^-2)      = 22032.09    Equatorial Radius, Re =   2440 km
  GM 1-sigma (km^3 s^-2)=  +-0.91   Mass ratio (sun/plnt) = 6023600

  Atmos. pressure (bar) =           Max. angular diam.   = 11.0"
  Mean Temperature (K)  =           Visual mag. V(1,0)   = -0.42
  Geometric albedo    =  0.106      Obliquity to orbit[1] = 2.11' +/- 0.1'
  Sidereal orb. per.  =  0.2408467 y  Mean Orbit vel.  km/s = 47.362
  Sidereal orb. per.  =  87.969257  d  Escape vel. km/s    = 4.435
  Hill's sphere rad. Rp =  94.4     Planetary Solar Const = 9936.9 (Wm^2)

 [1] Margot et al., Science 316, 2007
*******************************************************************************
```

Fig 17 – Mercury's body details

Evidently, the body details are displayed in different ways depending on the type of body. For example, fetching the mass would be concerning the output lines
" GM (10^11 km^3/s^2) = 1.3271244004193938 Mass (10^30 kg) ~ 1.988544"
and
" Mass (10^23 kg) = 3.302 Flattening, f ="
For the Sun and Mercury respectively.

As you can see, these lines contain a different combination of variables, in different orders, with different prefixes. My dynamic system seems to work perfectly, tackling all of these issues – extracting the relevant data and constructing the appropriate variables successfully. I will evaluate the success of this in greater detail in my final Testing stage. The exact method used to do this is explained by comments throughout the scripts[2].

You will notice that the Telnet Script uses multiple WAIT statements, this is to allow my program to wait between sending messages to the HORIZONS database – limiting potential errors that occurred when simultaneously sending the sets of messages. Although this does cause a costly time loss, it is fairly unavoidable whilst using the telnet connection (although the time loss issue could surely be optimised and improved with a potentially higher risk of errors).

Due to me using the telnet Interface through a script, it seems unnecessary to create a database to store the values fetched from telnet – since the data can be directly extracted and used (as I have done) without this. I am therefore going to avoid using the planned database in favour of a simpler and more direct approach.

<u>System Script (a.k.a 'SolarSystemScript' Script):</u>
The method used to calculate the updated positions of the bodies has gone through multiple iterations. I first established a Euler integration system. This is a first-order method of calculations and thus the error of the result per time-step is directly proportional to the square of the size of the time-step. For an N-Body simulation, this means that the updated position will be grossly inaccurate, especially for greater time-steps.

[2] All of the completed four scripts with full commenting are provided in the appendix.

A visualisation of Euler method approximations to a curve is displayed below.

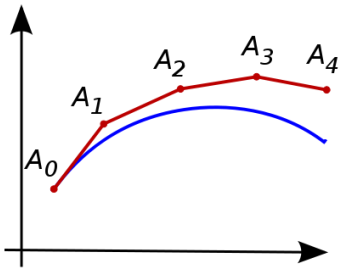

Fig 18 – Euler method approximations

This inaccuracy was visually evident in testing this code, with bodies rather quickly drifting away from their intended orbits. By using very minor time-steps, this error could be avoided, however more calculations would be necessary.

I have decided not to implement the original octree complexity reduction algorithms, favouring my time to explore the potential for complexity reduction through higher-order integrators.

In order to avoid performing vast amounts of sets of calculations, I employed the Runge-Kutta fourth-order (*RK*4) integrator. This takes multiple predicted averages of the result and calculates an approximation using the Simpson's rule. Somehow, my implementation of *RK*4 was erroneous – as the force ('xWorkingSum' in the x-plane) rapidly became negligible after consecutive sets of calculations.

A comparison of the results is shown below.

Fig 20 – Expected results

Fig 20 – Erroneous RK4 results

After struggling to find where the flaw in my implementation, I decided to attempt a slightly less accurate integrator – the 'Leapfrog' method. This is a second-order integrator and was fairly easy to implement, as it just involved using the previous calculations results with the current results to find a weighted average. After successfully implementing this, I found that the difference in results were practically negligible, whilst causing some additional strain on storage due to the extra variables required (the previous runs results).

The difference in the results of many hundreds of sets of calculations for two bodies are shown below.

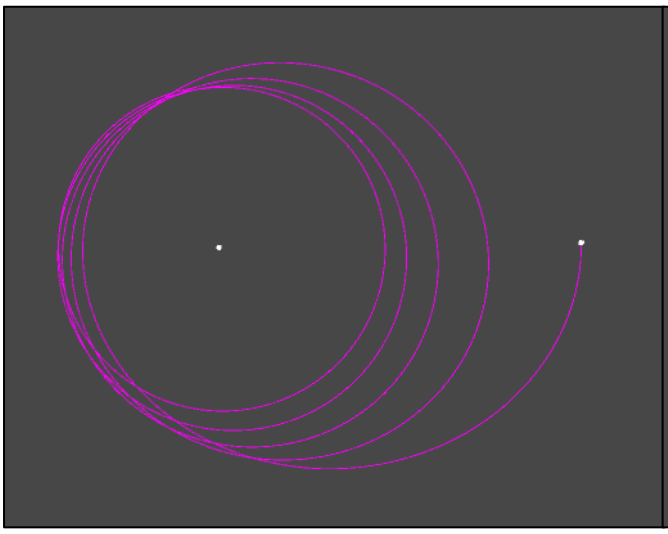

Fig 21 – Euler method results over multiple complete orbits

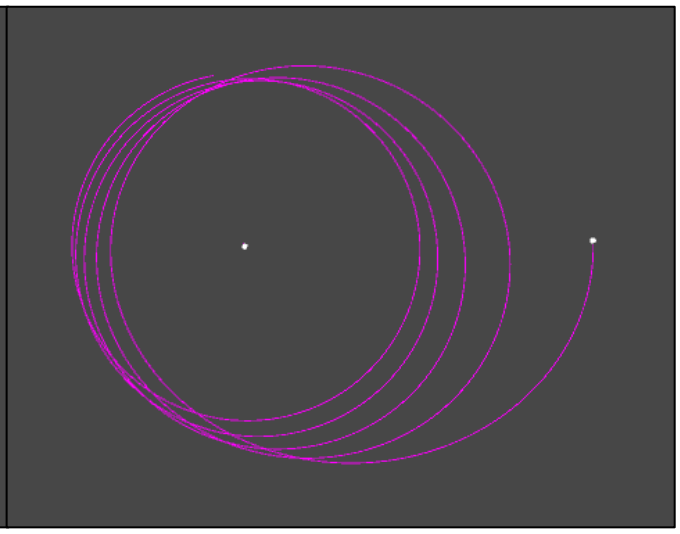

Fig 22 – Leapfrog method results over multiple complete orbits

As you can see, the difference is rather subtle.

I decided to revert back to the Euler method, reasoning that the lesser calculations to perform and lesser variables to store (compared to the Leapfrog) was outweighed the very slight improvement in accuracy. Thus potentially allowing more calculations to be performed per second with lesser time-steps, ultimately improving the accuracy.

Body Script (a.k.a. 'ObjectScript' Script):
As mentioned at the end of the design stage, I have now implemented position updates to occur after the entire set of calculations - avoiding some unnecessary error. The implementation of this was fairly easy, as I just needed to create a position setting subroutine ('SetPos'), which would use variables storing the results ('holdX' etc.) to then be used to set the position variables and translate the body once all the calculations are completed.

Camera Script (a.k.a. 'CameraMovement' Script):
Implementing the pop-up 'Custom System' window along with the rest of the UI ended up being a rather long process - using multiple long lines of code to access and validate the correct UI element's inputs.

My approach to validation was prioritised as such…
1 → Avoid ALL potential crashes.
2 → Check inputs are in the correct format – and thus immediately inform the user of ill formatted or disallowed inputs.
3 → Provide an accurate and informative error message to the user in the occurrence of an error.

I decided it would also be useful to the user if instructions on how to operate the program were provided in the place where error messages would be displayed in the absence of an error.

The complete hierarchy of Game Objects and the Unity API 'Inspectors' of the main game objects are provided in the appendix. Some of these inspectors show pre-set constant constants that I have set through the Unity API, rather than through scripts. The inspectors also show each Game Objects component and their component settings (e.g. renderers), along with their 'Transformers', which display the position, rotation and scale of the Game Object relative to its parent in the hierarchy (relative to (0, 0, 0) if no parent).

4. Testing

Methods for and reasons for testing:

Testing will be crucial for my project. It will be necessary in evaluating how modules of my code are working, both individually and collectively. This continuous evaluation will contribute to the development of my project by potentially presenting faulty features and other flaws which the project presents (e.g. lack of validation).
Both implicit and explicit testing of my final project and its development will be essential for evaluating the success of and potential further improvements to my project. It will also allow me to better reflect on my approach to development, allowing for a refined approach to design and development in future projects.

Instances of inadequate testing:

In my testing, it would not have been possible to attempt all combinations of potential inputs by hand. This will mean that some untested inputs that could cause errors or crashes will not be accounted for. I have therefore opted to limiting my range of tests to a variety of situations in which I could foresee potential errors occurring. I decided to test the code in a modular fashion, allowing for errors to be easily recognised and an ease in diagnosing the cause of errors.

Test purposes:

The purpose of each test will be in evaluating one of the following:
Proper validation is in place, crashes do not occur or whether a success criteria is met.

Each subject piece of data I have tested can be categorised as one the following…
'Valid' → Common data expected to work,
'Valid extreme' → Extreme data expected to work (e.g. values at their limits),
'Invalid' → Common data expected to fail (e.g. a date given in the wrong format (12/31/2000, where 31/12/2000 was expected),
'Invalid extreme' → Extreme data expected to fail
and 'Erroneous' → Data of the wrong data type.

Test table:

Test no.	Details	Data category	Expected outcome	Actual outcome	Action required
1	Testing Preset System. Loading dropdown option 'Solar System' or 'Sun – Inner planets', along with a typical start date and typical time interval.	Valid	Correct initial conditions for the bodies are presented.	The expected outcome.	None.
2	Testing Preset System. Loading dropdown option 'Custom Sytem' with typical body ID inputs, along with a typical start date and typical time interval.	Valid	Custom System windows pop-up, correct initial conditions for the bodies are presented.	The expected outcome.	None.
3	Testing Preset System. Loading dropdown option 'Custom System' with invalid body ID inputs, along with a typical start date and typical time interval.	Invalid	Custom System windows pop-up, no bodies will be presented.	The expected outcome.	None.
4	Testing Preset System. Loading dropdown option 'Custom System' with erroneous body ID inputs, along with a typical start date and typical time interval.	Erroneous	Custom System windows pop-up, no bodies will be presented.	The expected outcome.	None.
5	Testing Preset System. Loading dropdown option 'Custom System' with no body ID inputs, along with a typical start date and typical time interval.	Valid extreme	Custom System windows pop-up, no bodies will be presented.	The expected outcome.	None.
6	Testing Start Date. Loading a typical preset system, along with an invalid start date (wrong date format) and typical time interval.	Invalid	No bodies will be presented.	One of the bodies was presented (the sun).	Add direct validation to the date.
7	Testing Start Date. Loading a typical preset system, along with a valid extreme start date (very low value) and typical time interval.	Valid extreme	Bodies will be presented.	Only one of the bodies was presented (the sun).	The outcome could mean that the position values for all the missing bodies are not available. I should find the lower limit for all major body data and implement this as a limit on the input.
8	Testing Start Date. Loading a typical preset system, along with a valid extreme start date (very high value) and typical time interval.	Valid extreme	Bodies will be presented.	Four of the bodies were presented (one missing) and none of the bodies were moving.	The outcome could mean that the position values for the presented bodies are available but the velocity values are not. I should find the upper limit for all

					major body data and implement this as a limit on the input.
9	Testing Start Date. Loading a typical preset system, along with an erroneous start date and typical time interval.	Erroneous	No bodies will be presented.	One of the bodies was presented (the sun).	Add direct validation to the date.
10	Testing Time Interval. Loading a typical preset system, along with a typical start date and a valid extreme time interval (of 0).	Valid extreme	Bodies will be presented, the bodies will be motionless.	The expected outcome.	None
11	Testing Time Interval. Loading a typical preset system, along with a typical start date and a valid extreme time interval (of -2700).	Valid extreme	Bodies will be presented, the bodies will move in reverse (as if backwards in time)	The expected outcome.	None
12	Testing Time Interval. Loading a typical preset system, along with a typical start date and a valid extreme time interval (of 1000000).	Valid extreme	Bodies will be presented, the bodies will move very quickly.	The expected outcome.	None
13	Testing Time Interval. Loading a typical preset system, along with a typical start date and an erroneous time interval.	Erroneous	Error message will be displayed.	The expected outcome.	None
14	Testing the accuracy of the solution. Loading a typical preset system, along with a typical start date and a typical time interval.	Valid	Error in positions of less than 5% after a year's simulation.	The expected outcome. However, I have noticed the position-error of other simulations (typically involving more eccentric orbits such as those of Mercury) to be greater than the anticipated '<5%', causing the system to eventually lose the expected stability.	Successfully implement *RK4* or a similar higher-order integrator.

Fig 21 – Test Table

Evidence of testing:

Test no. 1

Fig 22 – Evidence of test no. 1

Test no. 2

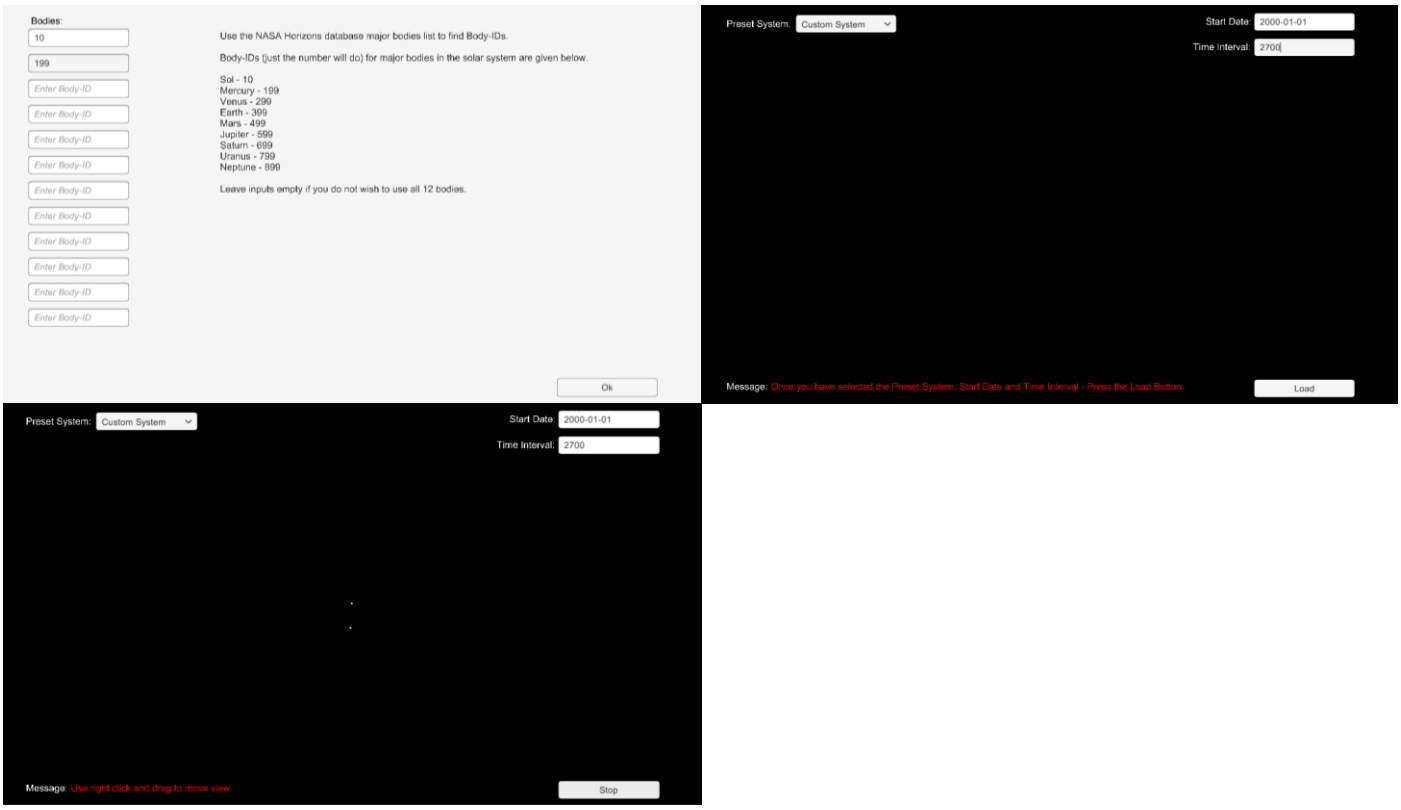

Fig 23 – Evidence of test no. 2

Test no. 3

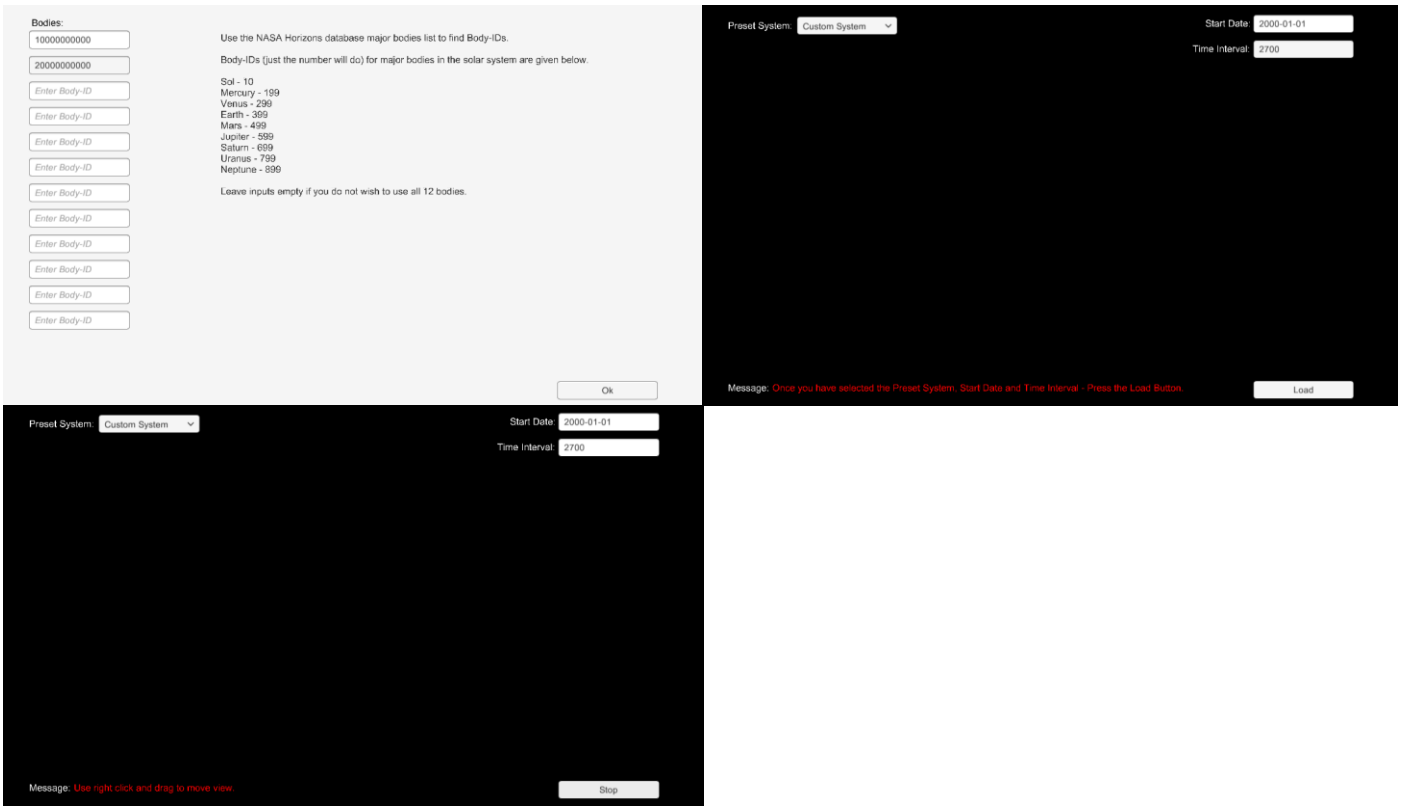

Fig 24 – Evidence of test no. 3

Test no. 4

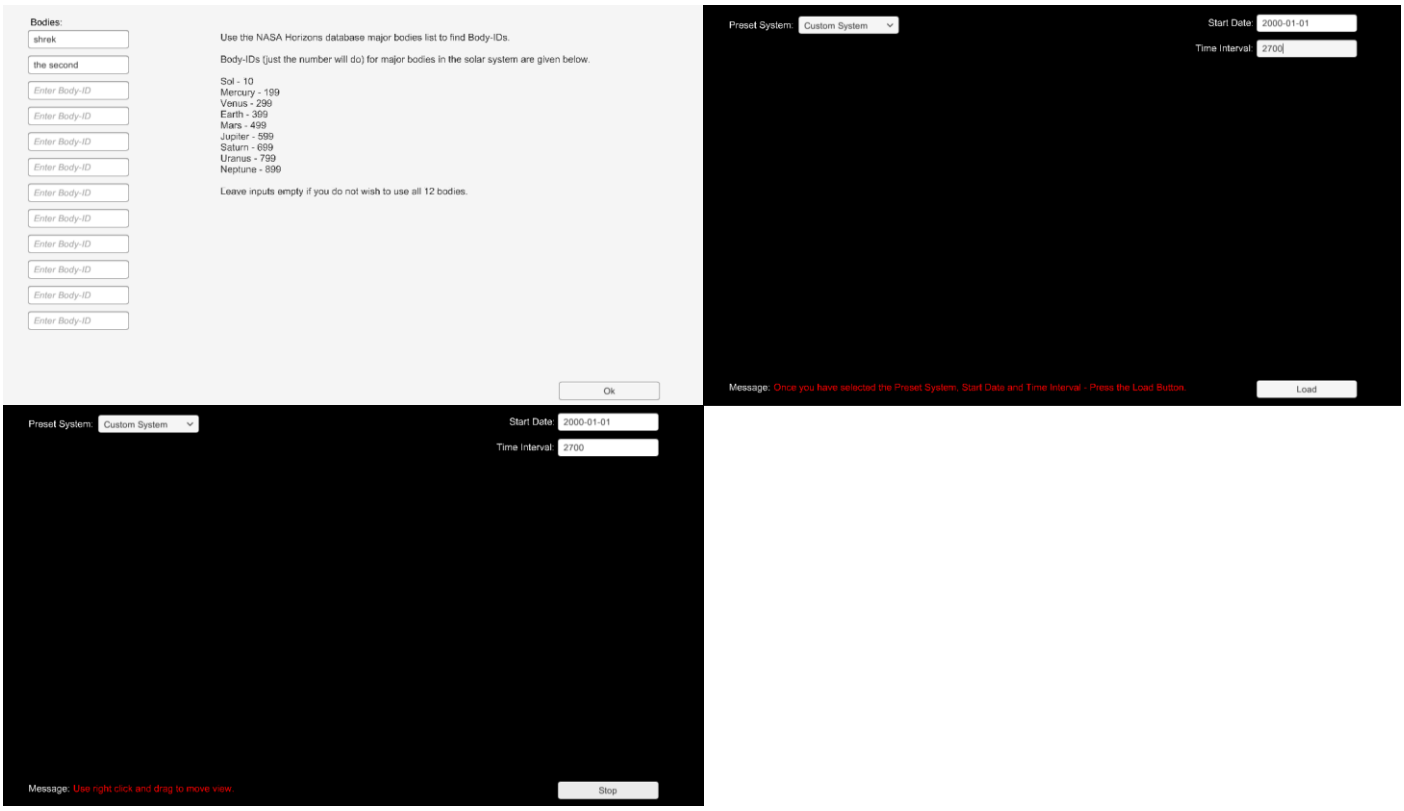

Fig 25 – Evidence of test no. 4

Test no. 5

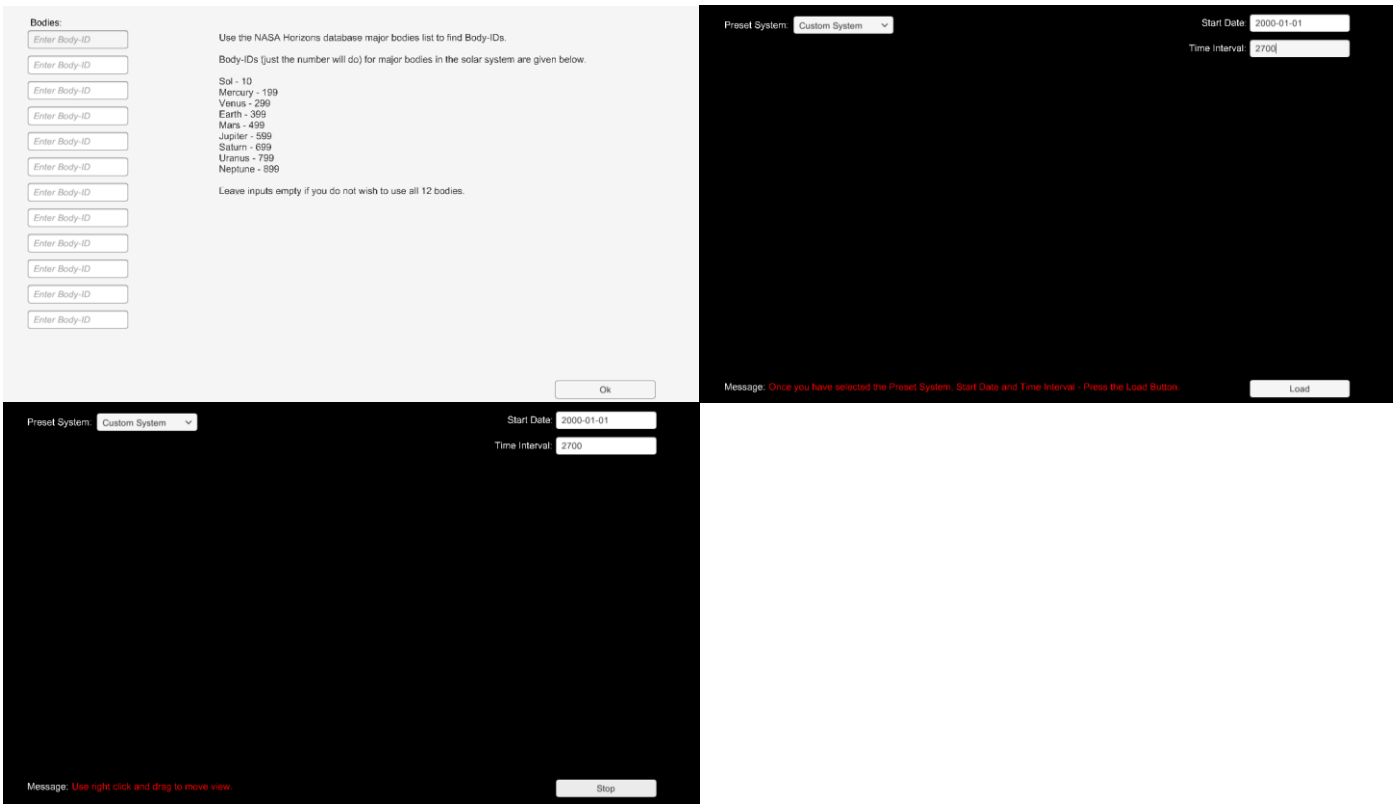

Fig 26 – Evidence of test no. 5

Test no. 6

Fig 27 – Evidence of test no. 6

Test no. 7

Fig 28 – Evidence of test no. 7

Test no. 8

Fig 29 – Evidence of test no. 8

Test no. 9

Fig 30 – Evidence of test no. 9

Test no. 10

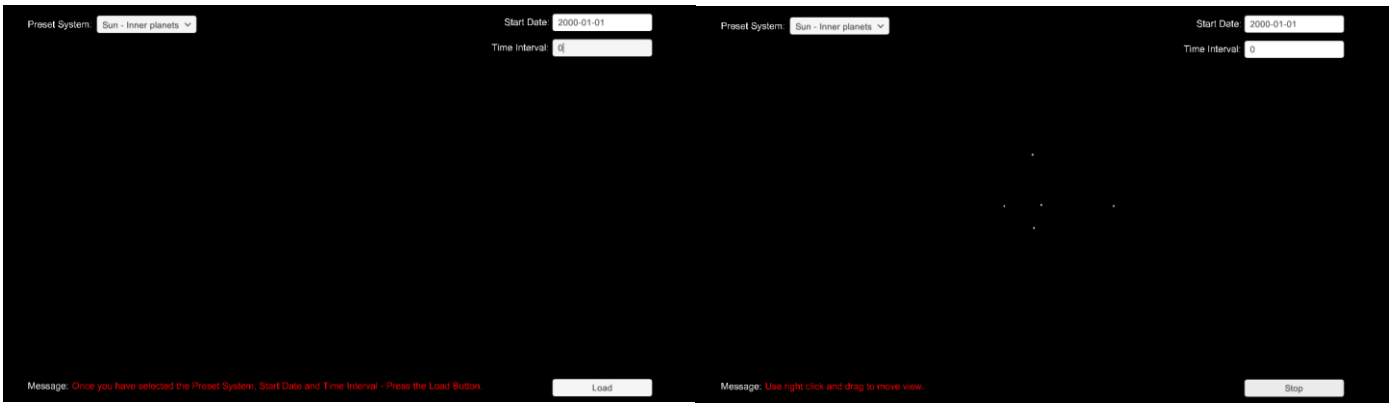

Fig 31 – Evidence of test no. 10

Test no. 11

Fig 32 – Evidence of test no. 11

Test no. 12

Fig 33 – Evidence of test no. 12

Test no. 13

Fig 34 – Evidence of test no. 13

Test no. 14

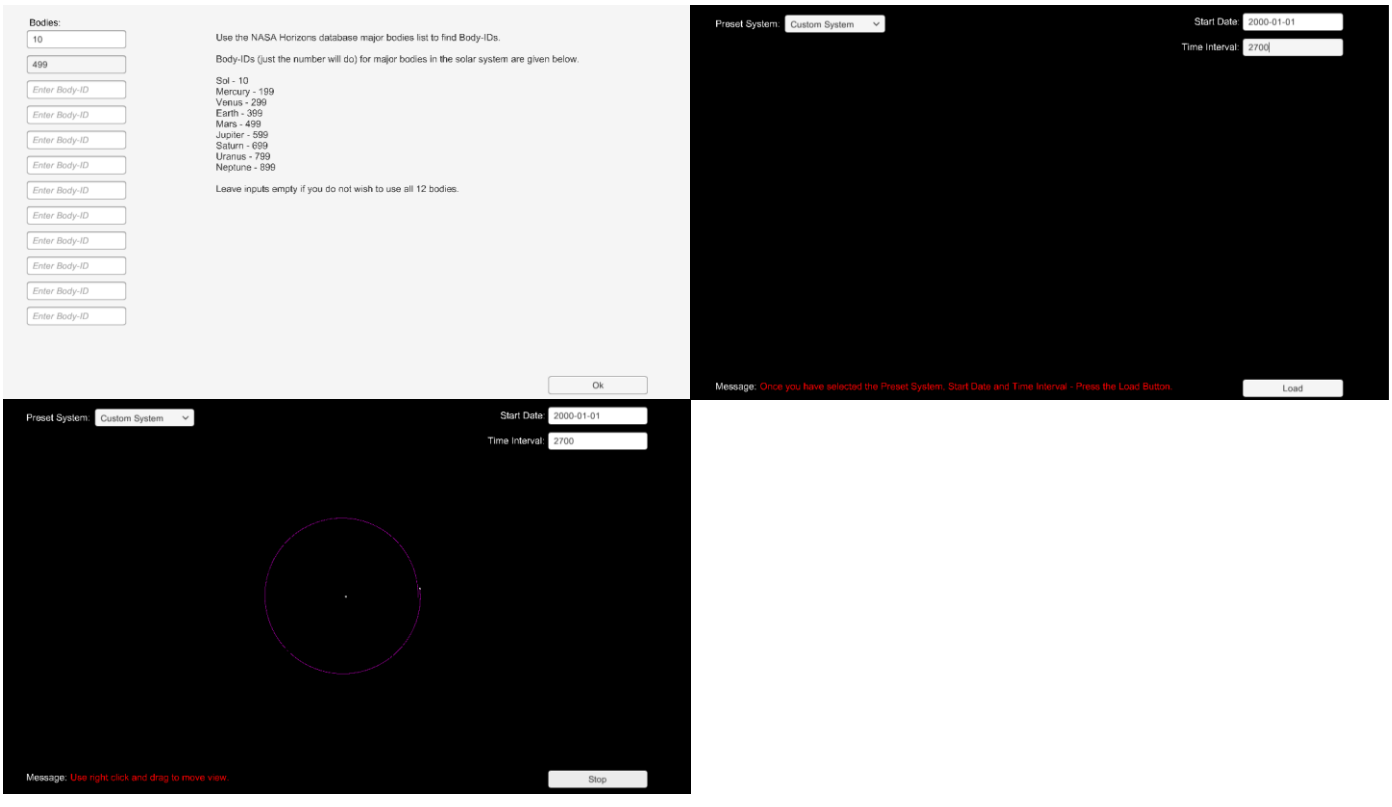

Fig 35 – Evidence of test no. 14

5. Evaluation

How the project compares to the original success criteria:

1. Produce a working *n*-body simulation (Acceptable limit: *maximum* error of 5% in any particles' position after 1 year's simulation (in any of the 3 Cartesian planes) (compared to NASA Horizons forecasted or real values)).

Partially met – Although this has been met for the majority of orbits involving low eccentricities, orbits with greater eccentricities (such as Mercury) suffer so greatly from the inaccuracies of the Euler integrator that they rapidly lose stability. This is so problematic to an *n*-body simulator that the next steps in improving the solution would be through improving the accuracy. This would most easily be done by improving the integrator to one of a much higher order.

2. Three-dimensional display of particle positions and trails.

***Met*.**

3. Option for user to define variables (e.g. Time between particles' position calculations).

***Met*.**

4. NASA Horizons database interface (allowing choice of initial particles).

***Met*.**

5. Select a body to view information regarding it and to provide the functionality of an observatory view from any position on the particle's surface.

***Not Met*.**

Regarding the end-user's evaluation and potential expansions in functionality:

Both of the end users have provided feedback throughout the development of the project regarding the approach they would prefer me to take on major decisions that affect the functionality of the final solution.

The end users are pleased with how the solution has turned out – especially the $3D$ plotting, camera controls and the NASA Horizons database access. They have also informed me that the issues regarding the first success criteria are to be expected. They support my proposition to use higher-order integrator to reduce the position-error and have suggested that I should attempt to implement this again. They would also like to see the implementation of the fifth success criteria. I expect this to be the greater challenge to overcome, but it is one I still wish to attempt. I am already fetching each body's radius data from the NASA Horizons database, which I planned to display as information on the body upon selection.

We have also conversed about potential further functionalities of the solution. We have agreed that allowing for more bodies to be used in custom-systems and calculating and providing apoapsis and periapsis of the selected body's last complete orbit would be further functionalities we might like to see.

Appendix:

Scripts:

Fig 36 – CameraMovement Script

```
using UnityEngine;
using System.Collections;
using UnityEngine.UI;
using System.Collections.Generic;
using System;
using System.Linq;

public class CameraMovement : MonoBehaviour
{
    // DECLARING appropriate variables.

    public float scrollPower;
    public List<string> temporaryBodies = new List<string>();

    // DECLARING Game Objects.
    GameObject windowPreset;

    // DECLARING Scripts.
    TelnetTesting telnetTesting;
    SolarSystemScript solarSystemScript;

    // DECLARING UI Text accessors.
    Text catchMessage;
    Text loadButton;

    // Start is CALLED at initialisation.       (called when the program is run, since an instance of this script
already exists through the Unity API.)
    void Start()
    {
        // SETTING appropriate variables.

        // SETTING Game Objects.
        windowPreset = GameObject.Find("Custom System Window Preset");

        // SETTING Scripts.
        telnetTesting = GameObject.Find("Telnet").GetComponent<TelnetTesting>();
        solarSystemScript = GameObject.Find("Solar System").GetComponent<SolarSystemScript>();

        // SETTING UI Text accessors.
        catchMessage = this.transform.FindChild("Canvas").FindChild("Message:
Text").FindChild("Text").GetComponent("Text") as Text;
        loadButton = this.transform.FindChild("Canvas").FindChild("Load
Button").FindChild("Text").GetComponent("Text") as Text;

        // DISABLING (and therefore immediately hiding) the custom system 'window'.     (which is originally
enabled since the above '.Find()' statement does not work for disabled Game Objects.)
        windowPreset.SetActive(false);
    }

    // Update is CALLED once per frame.
    void Update()
    {
        // ROTATE 'Camera Man' with appropriate angular velocity and direction if right button is pressed and the
mouse is dragged.
        if (Input.GetMouseButton(1))
        {
            transform.eulerAngles += new Vector3(Input.GetAxis("Mouse Y"), Input.GetAxis("Mouse X"), 0F);
        }

        // MOVE 'Camera Man' respectively closer or further to what it is facing if there is a respective forwards
or backwards scroll wheel action - essentially a ZOOM.
        if (Input.GetAxis("Mouse ScrollWheel") > 0)
        {
            transform.position += transform.forward * Time.deltaTime * scrollPower;
        }
        else if (Input.GetAxis("Mouse ScrollWheel") < 0)
        {
            transform.position -= transform.forward * Time.deltaTime * scrollPower;
        }
    }

    // CALLED from 'TelnetTesting' upon Preset System dropdown selecting 'Custom System'.
    public void CustomSystemWindow()
```

```
    {
        // ENABLE the custom system 'window'. This displays and activates the 'window' and all the UI elements that
are children of it.
        windowPreset.SetActive(true);

        // EMPTY the custom systems temporary list of user-inputted bodies, removing any potential previous data.
        temporaryBodies.Clear();
    }

    // CALLED by Unity API upon custom system window's 'Ok' button being pressed.
    public void OkButtonPressed()
    {
        // FOR all children of 'Custom System Window Preset' with "Body Text" in their name that do not have an
empty string in the input field, add the respective string to a list.
        for (int i = 0; i < windowPreset.transform.childCount; i++)
        {
            if (windowPreset.transform.GetChild(i).name.Contains("Body Text"))
            {
                Text inputText =
windowPreset.transform.GetChild(i).GetChild(0).FindChild("Text").GetComponent("Text") as Text;     // this looks
complicated but it's just traversing the Unity API hierarchy to access the appropriate Game Object.

                if (inputText.text != "")
                {
                    temporaryBodies.Add(inputText.text);
                }
            }
        }

        // RUN 'TelnetTesting's 'WindowClosed' sub-routine, setting the forementioned list of inputs to the
'TelnetTesting's list of bodies to later be fetched from the HORIZONS database.
        telnetTesting.WindowClosed(temporaryBodies);

        // DISABLE the custom system 'window'. This prevents the 'window' and all the UI elements that are children
of it from functioning and being displayed.
        windowPreset.SetActive(false);
    }

    // CALLED by Unity API upon the 'Load'('Stop') button being pressed.
    public void LoadButtonPressed()
    {
        // IF the simulation is not currently taking place, then TRY to set 'SolarSystemScript's interval ('E') to
the string in the time interval input field.
        // IF this doesn't fail, SET the 'Load' button's text to "Stop" and RUN 'TelnetTesting's
'PrepareInitiateTelnet' sub-routine - SETTING the parsed string from the start date input field for future HORIZONS
database access, and initiating TELNET access.
        // Otherwise IF this does fail, SET the user message to an appropriate error report.
        if (loadButton.text == "Load")
        {
            Text inputDateText = this.transform.FindChild("Canvas").FindChild("Start Date: Text").FindChild("Date
Input Field").FindChild("Text").GetComponent("Text") as Text;
            Text inputIntervalText = this.transform.FindChild("Canvas").FindChild("Time Interval:
Text").FindChild("Interval Input Field").FindChild("Text").GetComponent("Text") as Text;

            String temporaryDate = inputDateText.text;

            try
            {
                double temporaryInterval = Convert.ToDouble(inputIntervalText.text);
                solarSystemScript.E = (float)temporaryInterval;

                loadButton.text = "Stop";

                telnetTesting.PrepareInitiateTelnet(temporaryDate);
            }
            catch
            {
                catchMessage.text = "Time interval invalid.";
            }
        }

        // IF the simulation is currently taking place, then SET the 'Load' button's text to "Load", SET the user
message to it's initial string and SET 'SolarSystemScript's 'play' variable to false - DISABLING calculations (and
thus the simulation) unless set back to true.
        else if (loadButton.text == "Stop")
        {
            loadButton.text = "Load";

            catchMessage.text = "Once you have selected the Preset System, Start Date and Time Interval. Press the
```

```
Load Button.";

            solarSystemScript.play = false;
        }
    }

    // CALLED by 'SolarSystemScript' upon a simulation calculation failing.
    public void ErrorOccuredButtonSwap()
    {
        // SET the 'Load' button's text to "Load", SET the user message to an appropriate error report and DISABLE
future calculations (unless 'SolarSystemScript's 'play' is set back to true)
        loadButton.text = "Load";

        catchMessage.text = "Error occured. Inputs are not all valid or the condition were too extreme.";

        solarSystemScript.play = false;
    }

    // CALLED by 'TelnetTesting' upon intialising HORIZONS database access.
    public void DisplayETA(float telnetETA)
    {
        // SET the user message to an appropriate message, including the expected database access time.
        catchMessage.text = "Approximate time to receive this system's data is just under " + telnetETA + "
seconds.";
    }

    // CALLED by 'TelnetTesting' upon completing HORIZONS database access.
    public void InstructionsMessage()
    {
        // SET the user message to instructions on how to manipulate the camera.
        catchMessage.text = "Use right click and drag to move view.";
    }
}
```

Fig 37 – TelnetTesting Script

```
using UnityEngine;
using System.Collections;
using System;
using System.IO;
using System.Net.Sockets;
using System.Collections.Generic;
using System.Text.RegularExpressions;
using System.Threading;

public class TelnetTesting : MonoBehaviour
{
    // DECLARING appropriate variables.

    public string startDate;

    // DECLARING Game Objects.
    public GameObject solarSystemObject;
    public SolarSystemScript solarSystemScript;

    // DECLARING Scripts.
    CameraMovement cameraMovement;

    // DECLARING Lists.
    public List<bool> orderMassRadius = new List<bool>();
    public List<decimal> cartesianValues = new List<decimal>();
    public List<double> otherValues = new List<double>();
    public List<string> defaultBodyIdentifiers = new List<string>();
    public List<string> output = new List<string>();

    // DECLARING Telnet connection specific variables.
    internal Boolean socketReady = false;
    TcpClient mySocket;
    NetworkStream theStream;
    StreamWriter theWriter;
    StreamReader theReader;
    String Host = "horizons.jpl.nasa.gov";
    Int32 Port = 6775;
    public Boolean set = false;

    // Start is CALLED at initialisation.       (called when the program is run, since an instance of this script
already exists through the Unity API.)
    void Start()
    {
        // SETTING appropriate variables.

        // SETTING Game Objects.
        cameraMovement = GameObject.Find("Camera Man").GetComponent<CameraMovement>();
        solarSystemScript = GameObject.Find("Solar System").GetComponent<SolarSystemScript>();
    }

    // Update is CALLED once per frame.
    void Update()
    {
        // IF it is safe to do so, then read values from the Telnet Server and add them (line by line) to a list
('output'). The IF statement is required because unexpected and (as far as I can tell) unexplainable Unity engine
crashes occur if the values are read at certain times.
        if (set == true)
        {
            string receivedText = readSocket();

            if (receivedText != "")
            {
                output.Add(receivedText);
            }
        }
    }

    // CALLED by Unity API upon change in preset system dropdown selection.
    public void PresetSystemChange(int newOption)
    {
        // EMPTY the body ID list, removing any potential previous data.
        defaultBodyIdentifiers.Clear();

        // IF dropdown selection is 'Solar System'
        if (newOption == 1)
        {
            // ADD respective body ID's to the body ID list.
```

```
            defaultBodyIdentifiers.Add("10");
            defaultBodyIdentifiers.Add("199");
            defaultBodyIdentifiers.Add("299");
            defaultBodyIdentifiers.Add("399");
            //defaultBodyIdentifiers.Add("301");
            defaultBodyIdentifiers.Add("499");
            defaultBodyIdentifiers.Add("599");
            defaultBodyIdentifiers.Add("699");
            defaultBodyIdentifiers.Add("799");
            defaultBodyIdentifiers.Add("899");
        }

        // Otherwise IF dropdown selection is 'Sun - Inner planets'
        else if (newOption == 2)
        {
            // ADD respective body ID's to the body ID list.
            defaultBodyIdentifiers.Add("10");
            defaultBodyIdentifiers.Add("199");
            defaultBodyIdentifiers.Add("299");
            defaultBodyIdentifiers.Add("399");
            //defaultBodyIdentifiers.Add("301");
            defaultBodyIdentifiers.Add("499");
            //defaultBodyIdentifiers.Add("599");
            //defaultBodyIdentifiers.Add("699");
            //defaultBodyIdentifiers.Add("799");
            //defaultBodyIdentifiers.Add("899");
        }

        // Otherwise IF dropdown selection is 'Custom System'
        else if (newOption == 3)
        {
            // RUN 'CameraMovement's 'CustomSystemWindow' sub-routine - which deals with the UI and inputs as
described in the respective script's comments.
            cameraMovement.CustomSystemWindow();
        }
    }

    // CALLED from 'CameraMovement' upon the custom system window being closed.
    public void WindowClosed(List<String> temporaryBodies)
    {
        // SET the body ID's list to the user inputted body ID's list.
        defaultBodyIdentifiers = temporaryBodies;
    }

    // CALLED from 'CameraMovement' upon the 'Load' button being pressed when the simulation calculations are not
running and the time interval is a float.
    public void PrepareInitiateTelnet(string temporaryDate)
    {
        // SET the start date to the user inputted start date.
        startDate = temporaryDate;

        // RUN 'InitiateTelnet()' - establishing the telnet connection and corresponding with HORIZONS database.
        StartCoroutine(InitiateTelnet());
    }

    // CALLED from 'PrepareInitiateTelnet' upon completed preperation for HORIZONS databse correspondence.
    public IEnumerator InitiateTelnet()
    {
        // EMPTYING appropriate lists used to store read data and appropriate data for body creation, removing any
potential previous data.
        output.Clear();
        cartesianValues.Clear();
        otherValues.Clear();
        orderMassRadius.Clear();

        // RUN 'CameraMovement's 'DisplayETA' using the calculated expected time - setting the user message to an
appropriate message containing the expected time.
        float eta = 4f + 25f + (defaultBodyIdentifiers.Count * 13f);
        cameraMovement.DisplayETA(eta);

        // INITIALISING the telnet connection.
        SetupSocket();
        WriteSocket("serverStatus:");

        // WAITING 4 seconds for Telnet Correspondence.
        yield return new WaitForSeconds(4f);

        // RUN 'SendAll()' - sending body data to the telnet server in the appropriate form, and harvesting the
appropriate data from the list of read telnet messages.
```

```
        StartCoroutine(SendAll());
    }

    // CALLED from 'InitiateTelnet' upon completed intilisation of the telnet connection.
    public IEnumerator SendAll()
    {
        // FOR each body, RUN the appropriate sub-routine ('WaitSendAllFirst' (first body) or 'WaitSendAllOthers'
(all other bodies).) The need for two (very similar) sub-routines is because the server asks if some of the
previous bodies settings will be the same for every body after the first. - SENDING messages to the telnet server
in an accepted form.
        for (int i = 0; i < defaultBodyIdentifiers.Count; i++)
        {
            if (i == 0)
            {
                StartCoroutine(WaitSendAllFirst(defaultBodyIdentifiers[i]));
            }

            else if (i > 0)
            {
                // WAIT 13 seconds - allowing the previous bodies messages to be sent safely.
                yield return new WaitForSeconds(13f);
                StartCoroutine(WaitSendAllOthers(defaultBodyIdentifiers[i]));
            }
        }

        // WAIT 25 seconds - allowing for final body's messages to be fully sent and respones fully received.
        yield return new WaitForSeconds(25f);

        // CLOSE the telnet connection, since necessary communications are complete.
        CloseSocket();

        // RUN 'CameraMovement's 'InstructionsMessage' sub-routine - SETTING the user message to instructions on
how to manipulate the camera.
        cameraMovement.InstructionsMessage();

        // FOR each read line, take appropriate actions in harvesting the necessary body data.
        for (int i = 0; i < output.Count; i++)
        {
            // IF the line is "$$SOE" (written by telnet to suggest the next few lines will be regarding the
cartesian values). THEN for appropriate lines, harvest data for the particular format and add this to a List of
cartesian values.
            // FORMAT E.G.: " X =-7.139143380212697E-03 Y =-2.792019770161695E-03 Z = 2.061838852554664E-04"
            if (output[i] == "$$SOE")
            {
                // Since these are the appropriate lines to get data from after the "$$SOE" line.
                for (int x = 2; x < 4; x++)
                {
                    // Since each line contains three variables to harvest data from ('y' values are used in the
respective sub-routine for this purpose.)
                    for (int y = 0; y < 3; y++)
                    {
                        cartesianValues.Add(ScrapeData(i, x, y, 2, 1));
                    }
                }
            }

            // The proceeding selective statements are for the unique formats in which body data is presented by
the HORIZONS database. Note that 'orderMassRadius' values are only assigned for lines regarding mass, since every
single object potentially has it's mass and radius values on two unique lines.
            // The 'ScrapeDataDouble' sub-routine is used for the following calls since the resultant data is
wanted in double form (rather than decimal). This is because these values (e.g. mass) can exceed the range of
decimal values.
            // For the appropriate lines in which data is harvested, the results are added to a list of non-
cartesian values ('otherValues'), along with another list for the order in which these variables occured
('orderMassRadius', where true means mass occured before radius.)

            // MASS

            // FORMAT E.G.: "  GM (10^11 km^3/s^2)    = 1.3271244004193938  Mass (10^30 kg)    ~ 1.988544"
            else if (output[i].Contains("GM") && output[i].Contains("Mass") && !output[i].Contains("Mass ratio"))
            {
                otherValues.Add(ScrapeDataDouble(i, 0, 0, 7, 8));
                orderMassRadius.Add(true);
            }

            // FORMAT E.G.: "  Mass (10^23 kg )      =     3.302      Flattening, f         ="
            else if (output[i].Contains("Mass") && output[i].Contains("Flattening"))
            {
                otherValues.Add(ScrapeDataDouble(i, 0, 0, 2, 3));
```

```
                orderMassRadius.Add(false);
            }

            // FORMAT E.G.: "  Mass (10^24 kg)       =  1898.13+-.19   Density (g/cm^3)       =  1.326"
            else if (output[i].Contains("Mass") && output[i].Contains("Density"))
            {
                otherValues.Add(ScrapeDataDouble(i, 0, 0, 2, 3));
                orderMassRadius.Add(true);
            }

            // MASS & RADIUS

            // FORMAT E.G.: " Mean radius, km          = 6371.01+-0.01   Mass, 10^24 kg = 5.97219+-0.0006"
            else if (output[i].Contains("Mean radius") && output[i].Contains("Mass"))
            {
                otherValues.Add(ScrapeDataDouble(i, 0, 0, 0, 1));    // Radius
                otherValues.Add(ScrapeDataDouble(i, 0, 0, 4, 5));    // Mass
                orderMassRadius.Add(false);
            }

            // FORMAT E.G.: " Radius (IAU), km      = 1737.4          Mass, 10^20 kg        =  734.9"
            else if (output[i].Contains("Radius") && output[i].Contains("Mass"))
            {
                otherValues.Add(ScrapeDataDouble(i, 0, 0, 0, 1));   // Radius
                otherValues.Add(ScrapeDataDouble(i, 0, 0, 3, 4));   // Mass
                orderMassRadius.Add(false);
            }

            // RADIUS

            // FORMAT E.G.: "  Radius (photosphere)  = 6.963(10^5) km  Angular diam at 1 AU  = 1919.3""
            else if (output[i].Contains("Radius (photosphere)") && output[i].Contains("Angular diam"))
            {
                otherValues.Add(ScrapeDataDouble(i, 0, 0, 3, 1));
            }

            // FORMAT E.G.: "  Mean radius (km)      =  2440(+-1)     Density (g cm^-3)     =  5.427"
            else if (output[i].Contains("Mean radius") && output[i].Contains("Density"))
            {
                otherValues.Add(ScrapeDataDouble(i, 0, 0, 0, 1));                               //(nValue == 0
-->     no exponent)
            }

            // FORMAT E.G.: "  Volumetric mean radius= 69911+-6 km     Flattening             =  0.06487"
            else if (output[i].Contains("Volumetric mean radius") && output[i].Contains("Flattening"))
            {
                otherValues.Add(ScrapeDataDouble(i, 0, 0, 0, 1));
            }
        }

        // RUN 'SolarSystemScript's 'PrepareCreateBody' sub-routine using the Lists of appropriate body data -
this extracts each body's respective data from the Lists to create the Game Objects for each body.
        solarSystemScript.PrepareCreateBody(cartesianValues, otherValues, orderMassRadius);
    }

    // CALLED from 'SendAll' upon identifying lines containing cartesian values.
    public decimal ScrapeData(int i, int x, int y, int nExponent, int nValue)
    {
        // SETTING 'bodyValues' to the appropriate line
        string bodyValues = output[i + x];

        // REPLACING all non-numbers in 'bodyValues' with empty spaces and then REPLACING repeated empty spaces
with a single empty space.
        bodyValues = Regex.Replace(bodyValues, "[^0-9-.]", " ");
        bodyValues = Regex.Replace(bodyValues, @"\s+", " ");

        // DECLARING a List of body values and SETTING each number in 'bodyValues' as an element of the List. Note
that 'bodyValuesList' emulates base 1 (since each line starts with " " and so "" is treated as the first string
upon SPLITTING)
        string[] bodyValuesList = bodyValues.Split(' ');

        // RE-CONSTRUCTING the body value.

        // SETTING prefix initial conditions (the final prefix will depend on the exponent).
        decimal prefixScaler = 1.0m;

        // IF the exponent exists (0 means it doesn't exist), then SET the prefix to match the respective exponent
from the list of body values.
        if (nExponent != 0)
```

```
        {
            decimal prefixExponent = Convert.ToDecimal(bodyValuesList[nExponent + (2 * y)]);

            // IF the exponent is negative then make the prefix to negative powers of 10.
            if (prefixExponent < 0)
            {
                for (int k = 0; k < -prefixExponent; k++)       // Using '-prefixExponent' to get the modulus
(negative & negative is positive), allowing for k to be incremented appropriately.
                {
                    prefixScaler *= 1.0e-1m;
                }
            }
            // OTHERWISE the exponent is positive and so make the prefix to positive powers of 10.
            else
            {
                for (int k = 0; k < prefixExponent; k++)
                {
                    prefixScaler *= 1.0e+1m;
                }
            }
        }

        // SET the value to match the respective value from the list of body values.
        decimal value = Convert.ToDecimal(bodyValuesList[nValue + (2 * y)]);

        // SET the whole value to the product of the value and the prefix and RETURN it.
        decimal valueWhole = value * prefixScaler;
        return (valueWhole);
    }

    // CALLED from 'SendAll' upon identifying lines containing non cartesian values.
    public double ScrapeDataDouble(int i, int x, int y, int nExponent, int nValue)
    {
        //SETTING 'bodyValues' to the appropriate line
        string bodyValues = output[i + x];

        // REPLACING all non-numbers in 'bodyValues' with empty spaces and then REPLACING repeated empty spaces
with a single empty space.
        bodyValues = Regex.Replace(bodyValues, "[^0-9-.]", " ");
        bodyValues = Regex.Replace(bodyValues, @"\s+", " ");

        // DECLARING a List of body values and SETTING each number in 'bodyValues' as an element of the List. Note
that 'bodyValuesList' emulates base 1 (since each line starts with " " and so "" is treated as the first string
upon SPLITTING)
        string[] bodyValuesList = bodyValues.Split(' ');

        //  RE-CONSTRUCTING the body value.

        // SETTING the prefix initial conditions (the final prefix will depend on the exponent).
        double prefixScaler = 1.0d;

        // IF the exponent exists (0 means it doesn't exist), then SET the prefix to match the respective exponent
from the list fo body values.
        if (nExponent != 0)
        {
            double prefixExponent = Convert.ToDouble(bodyValuesList[nExponent + (2 * y)]);      // prefix exponent

            // IF the exponent is negative then make the prefix to negative powers of 10.
            if (prefixExponent < 0)
            {
                for (int k = 0; k < -prefixExponent; k++)     // Using '-prefixExponent' to get the modules
(negative & negative is positive), allowing for k to be incremented appropriately.
                {
                    prefixScaler *= 1.0e-1d;
                }
            }
            // OTHERWISE the exponent is positive and so make the prefix to positive powers of 10.
            else
            {
                for (int k = 0; k < prefixExponent; k++)
                {
                    prefixScaler *= 1.0e+1d;
                }
            }
        }

        // SET the value to match the respective value from the list of body values.
        double value = Convert.ToDouble(bodyValuesList[nValue + (2 * y)]);
```

```
        // SET the whole value to the product of the value and the prefix and RETURN it.
        double valueWhole = value * prefixScaler;
        return (valueWhole);
    }


    // CALLED by 'SendAll' upon preparing to send telnet the first body's data.
    public IEnumerator WaitSendAllFirst(string bodyIdentifier)
    {
        // WRITING the appropriate set of messages between fixed intervals, using the user defined body ID and
start date.
        WriteSocket("");
        yield return new WaitForSeconds(0.5f);
        WriteSocket("");
        yield return new WaitForSeconds(0.5f);
        WriteSocket("");
        yield return new WaitForSeconds(0.5f);
        WriteSocket(bodyIdentifier);
        yield return new WaitForSeconds(0.5f);
        WriteSocket("E");
        yield return new WaitForSeconds(0.5f);
        WriteSocket("v");
        yield return new WaitForSeconds(0.5f);
        WriteSocket("500@0");
        yield return new WaitForSeconds(0.5f);
        WriteSocket("y");
        yield return new WaitForSeconds(0.5f);
        WriteSocket("eclip");
        yield return new WaitForSeconds(0.5f);
        WriteSocket(startDate + " 00:00");
        yield return new WaitForSeconds(0.5f);
        WriteSocket(startDate + " 00:01");
        yield return new WaitForSeconds(0.5f);
        WriteSocket("1h");
        yield return new WaitForSeconds(0.5f);
        WriteSocket("n");
        yield return new WaitForSeconds(0.5f);
        WriteSocket("J2000");
        yield return new WaitForSeconds(0.5f);
        WriteSocket("1");
        yield return new WaitForSeconds(0.5f);
        WriteSocket("1");
        yield return new WaitForSeconds(0.5f);
        WriteSocket("NO");
        yield return new WaitForSeconds(0.5f);
        WriteSocket("NO");
        yield return new WaitForSeconds(0.5f);
        WriteSocket("3");
        yield return new WaitForSeconds(0.5f);

        // ENABLING (and then DISABLING) 'set' for the final few commands. READING at this point in the code
doesn't seem to cause crashes. (Whilst still receiving all necessary data).
        set = true;
        WriteSocket("YES");
        yield return new WaitForSeconds(0.5f);
        WriteSocket("N");
        yield return new WaitForSeconds(4f);
        set = false;
    }

    // CALLED by 'SendAll' upon preparing to send telnet consecutive body's data.
    public IEnumerator WaitSendAllOthers(string bodyIdentifier)
    {
        // WRITING the appropriate set of messages between fixed intervals, using the user defined body ID and
start date.
        yield return new WaitForSeconds(4f);
        WriteSocket("");
        yield return new WaitForSeconds(0.5f);
        WriteSocket("");
        yield return new WaitForSeconds(0.5f);
        WriteSocket("");
        yield return new WaitForSeconds(0.5f);
        WriteSocket(bodyIdentifier);
        yield return new WaitForSeconds(0.5f);
        WriteSocket("E");
        yield return new WaitForSeconds(0.5f);
        WriteSocket("v");
        yield return new WaitForSeconds(0.5f);
        WriteSocket("y");
```

```
        yield return new WaitForSeconds(0.5f);
        WriteSocket("eclip");
        yield return new WaitForSeconds(0.5f);
        WriteSocket(startDate + " 00:00");
        yield return new WaitForSeconds(0.5f);
        WriteSocket(startDate + " 00:01");
        yield return new WaitForSeconds(0.5f);
        WriteSocket("1h");
        yield return new WaitForSeconds(0.5f);
        WriteSocket("n");
        yield return new WaitForSeconds(0.5f);
        WriteSocket("J2000");
        yield return new WaitForSeconds(0.5f);
        WriteSocket("1");
        yield return new WaitForSeconds(0.5f);
        WriteSocket("1");
        yield return new WaitForSeconds(0.5f);
        WriteSocket("NO");
        yield return new WaitForSeconds(0.5f);
        WriteSocket("NO");
        yield return new WaitForSeconds(0.5f);
        WriteSocket("3");
        yield return new WaitForSeconds(0.5f);

        // ENABLING (and then DISABLING) 'set' for the final few commands. READING at this point in the code
doesn't seem to cause crashes. (Whilst still receiving all necessary data).
        set = true;
        WriteSocket("YES");
        yield return new WaitForSeconds(0.5f);
        WriteSocket("N");
        yield return new WaitForSeconds(4f);
        set = false;
    }

    // CALLED from Unity API upon closing the application.
    void OnApplicationQuit()
    {
        // CLOSE the telnet connection.
        CloseSocket();
    }

    // CALLED by 'InitiateTelnet' upon preparing to establish telnet connection.
    public void SetupSocket()
    {
        // TRY to establish telnet connection. If this fails (e.g. host offline), then LOG the error (potentially
avoiding crash).
        try
        {
            // SET the client and establish the respective stream for communicating.
            mySocket = new TcpClient(Host, Port);
            theStream = mySocket.GetStream();

            // SET the writer and the reader.
            theWriter = new StreamWriter(theStream);
            theReader = new StreamReader(theStream);

            // ENABLE 'socketReady' - The connection was correctly established and thus writing and reading can
occur.
            socketReady = true;
        }
        catch (Exception e)
        {
            Debug.Log("Socket error: " + e);
        }
    }

    // CALLED by 'InitiateTelnet', 'WaitSendAllFirst' and 'WaitSendAllOthers'.
    public void WriteSocket(string theLine)
    {
        // IF 'socketReady' is DISABLED then do not attempt to access the writer, RETURN immediately.       (since
connection was not correctly esatblished.)
        if (socketReady == false)
        {
            return;
        }

        // SET the message to write given the data to send, WRITE the message and SEND it via the stream.
        String message = theLine + "\r\n";
        theWriter.Write(message);
```

```
        theWriter.Flush();
    }

    // CALLED by 'Update' every second upon data being safe to read.
    public String readSocket()
    {
        // IF 'socketReady' is DISABLED then do not attempt to access the reader, RETURN "" immediately.
(since connection was not correctly esatblished.)
        if (socketReady == false)
        {
            return "";
        }

        // IF the stream has data, READ and RETURN it.
        if (theStream.DataAvailable == true)
        {
            return theReader.ReadLine();
        }

        // IF the streame has no data, RETURN "".
        return "";
    }

    // CALLED by 'SendAll' and 'OnApplicationQuit'.
    public void CloseSocket()
    {
        // IF 'socketReady' is DISABLED then do not attempt to CLOSE the connection, RETURN immediately.
(since connection was not correctly esatblished.)
        if (socketReady == false)
        {
            return;
        }

        // CLOSE the writer, reader and connection. DISABLE 'socketReady' - preventing telnet functions from still
being accessed.
        theWriter.Close();
        theReader.Close();
        mySocket.Close();
        socketReady = false;
    }
}
```

Fig 38 – ObjectScript Script

```
using UnityEngine;
using System.Collections;

public class ObjectScript : MonoBehaviour
{
    // DECLARE appropriate variables.

    public float UIscale;
    public double ObjectRad;
    public double ObjectMass;

    // DECLARE Transforms.
    Transform thisTransform;

    // DECLARE cartesian values.
    public decimal ObjectX;
    public decimal ObjectY;
    public decimal ObjectZ;

    public decimal ObjectVX;
    public decimal ObjectVY;
    public decimal ObjectVZ;

    public decimal holdX;
    public decimal holdY;
    public decimal holdZ;

    public decimal ObjectAX0 = 0m;
    public decimal ObjectAY0 = 0m;
    public decimal ObjectAZ0 = 0m;

    // Start is CALLED at initialisation.       (called when assigned to the Game Object)
    void Start()
    {
        // SET appropriate variables.

        UIscale = 1e-6F;

        // SET Transforms.
        thisTransform = gameObject.GetComponent<Transform>();

        // SET cartesian values. These are simply float versions of the (more accurate) decimal values since the
Unity API only accepts float values for Game Object transformations.
        float ObjectXf = (float)ObjectX;
        float ObjectYf = (float)ObjectY;
        float ObjectZf = (float)ObjectZ;

        // SET initial position of the body.
        transform.position = new Vector3(ObjectXf, ObjectYf, ObjectZf) * UIscale;
        transform.localScale = new Vector3(5e+0f, 5e+0f, 5e+0f);
    }

    // CALLED by 'SolarSystempScript's 'NBodyProblem' sub-routine upon completion of entire set of calculations for
all bodies.
    public void SetPos()
    {
        // SET position variables
        ObjectX = holdX;
        ObjectY = holdY;
        ObjectZ = holdZ;

        // SET float variables for the translation.
        float xif = (float)(holdX);
        float yif = (float)(holdY);
        float zif = (float)(holdZ);

        // TRANSLATE the position to the new calculated position, taking into account the appropriate UI scaler -
ensuring that the object is well within the Unity Scene's spacial limits.
        thisTransform.localPosition = new Vector3(xif, yif, zif) * UIscale;
    }

    // CALLED by 'SolarSystemScript's 'CreateBody' sub-routine upon completed preparation of a body's values.
    public void InitBody(SolarSystemScript parent, decimal objectX, decimal objectY, decimal objectZ, decimal
objectVX, decimal objectVY, decimal objectVZ, double objectMass, double objectRad)
    {
        // SET appropriate variables.
```

```
        ObjectMass = objectMass;
        ObjectRad = objectRad;

        // SET Transform (position, rotation and scale) to that of 'SolarSystemScript's body - 'Solar System'
        transform.parent = parent.transform;

        // SET cartesian values.
        ObjectX = objectX;
        ObjectY = objectY;
        ObjectZ = objectZ;

        ObjectVX = objectVX;
        ObjectVY = objectVY;
        ObjectVZ = objectVZ;

        // SET the mesh and material to those of 'Solar System' (which are disabled through the Unity API). ADD
the MeshFilter and MeshRenderer components to this Game Object.
        Mesh mesh = parent.mesh;
        Material material = parent.material;
        gameObject.AddComponent<MeshFilter>().mesh = mesh;
        gameObject.AddComponent<MeshRenderer>().material = material;

        // SET and ENABLE all rendering of this Game Object.
        Renderer rend = GetComponent<Renderer>();
        rend.enabled = true;

        // SET and ENABLE the trail for this Game Object.
        TrailRenderer trail = gameObject.AddComponent<TrailRenderer>();
        trail.time = Mathf.Infinity;
    }
}
```

Fig 39 – SolarSystemScript Script

```
using UnityEngine;
using System.Collections;
using System.Net;
using System.Net.Sockets;
using System.Security.Cryptography.X509Certificates;
using System;
using System.IO;
using System.Collections.Generic;
using System.Text.RegularExpressions;

public class SolarSystemScript : MonoBehaviour
{
    // DECLARE appropriate variables.

    public int presetSystem;
    private double G;
    public float E;
    public bool play;

    // DECLARE Mesh, Material and TrailRenderer.
    public Mesh mesh;
    public Material material;

    // DECLARE Scripts.
    CameraMovement cameraMovement;

    // Start is CALLED at initialisation.       (called when the program is run, since an instance of this script already exists through the Unity API.)
    void Start()
    {
        // SET appropriate variables.

        G = 6.6738e-20d;

        // SET Scripts.
        cameraMovement = GameObject.Find("Camera Man").GetComponent<CameraMovement>();
    }

    // Update is CALLED once per frame.
    void Update()
    {
        // IF the simulation is ENABLED, TRY RUNNING the 'NBodyProblem' sub-routine - calculating the new positions of every body given initial conditions and a given time-step.
        // If this fails, RUN 'CameraMovement's 'ErrorOccuredButtonSwap' sub-routine - displaying an appropriate error message, DISABLING the simulation and swapping the 'Load' button's text.
        if (play == true)
        {
            try
            {
                NBodyProblem();
            }
            catch
            {
                cameraMovement.ErrorOccuredButtonSwap();
            }
        }
    }

    // CALLED by 'TelnetTesting's 'SendAll' sub-routine upon sorting all the necessary data into three appropriate Lists.
    public void PrepareCreateBody(List<decimal> cartesianValues, List<double> otherValues, List<bool> orderMassRadius)
    {
        // FOR each expected body, CREATE the body using the appropriate data.
        for (int i = 0; i < orderMassRadius.Count; i++)
        {
            // IF an expected body's format had mass before radius, THEN RUN the 'CreateBody' sub-routine, getting the respective data for the body from the Lists.
            if (orderMassRadius[i] == true)
            {
                CreateBody("test body", cartesianValues[(6 * i)], cartesianValues[(6 * i) + 1], cartesianValues[(6 * i) + 2], cartesianValues[(6 * i) + 3], cartesianValues[(6 * i) + 4], cartesianValues[(6 * i) + 5], otherValues[(2 * i)], otherValues[(2 * i) + 1]);
            }
            // Otherwise IF an expected body's format had radius before mass, THEN create a child to this Game Object ('Solar System') called "test body", getting it's respective data from the Lists.
            else
```

```
            {
                CreateBody("test body", cartesianValues[(6 * i)], cartesianValues[(6 * i) + 1], cartesianValues[(6
* i) + 2], cartesianValues[(6 * i) + 3], cartesianValues[(6 * i) + 4], cartesianValues[(6 * i) + 5],
otherValues[(2 * i) + 1], otherValues[(2 * i)]);
            }
        }

        // ENABLE the simulation.
        play = true;
    }

    // CALLED by the 'PrepareCreateBody' sub-routine upon collecting and assigning the respective body's data.
    private void CreateBody(string bodyName, decimal objectX, decimal objectY, decimal objectZ, decimal objectVX,
decimal objectVY, decimal objectVZ, double objectMass, double objectRad)      //input properties here (from telnet
or from user input)
    {
        // CREATE a new Child Game Object to this Game Object ('Solar System') called "test body", using the body
variables passed from the respective call.
        new GameObject(bodyName).AddComponent<ObjectScript>().InitBody(this, objectX, objectY, objectZ, objectVX,
objectVY, objectVZ, objectMass, objectRad);
    }

    // CALLED by 'Update' every frame if the simulation is ENABLED.
    private void NBodyProblem()
    {
        // FOR each Child body, CALCULATE the new position of the respective body under the gravitational
influences of the other bodies.
        for (int i = 0; i < transform.childCount; i++)
        {
            // SET appropriate variables.

            decimal xWorkingSum = 0m;
            decimal yWorkingSum = 0m;
            decimal zWorkingSum = 0m;

            // SET Scripts.
            ObjectScript childiScript = gameObject.transform.GetChild(i).GetComponent<ObjectScript>();

            // SET variables to those from the respective Child's ('i') Script.
            double mi = childiScript.ObjectMass;

            decimal xi = childiScript.ObjectX;
            decimal yi = childiScript.ObjectY;
            decimal zi = childiScript.ObjectZ;

            // CALCULATE the distance of the repsective Child Body from the origin.
            decimal di = (decimal)Mathf.Sqrt((float)((xi * xi) + (yi * yi) + (zi * zi)));

            // FOR every other body...
            for (int j = 0; j < transform.childCount; j++)
            {
                if (j != i)
                {
                    // SET appropriate variables.

                    // SET Scripts.
                    ObjectScript childjScript = gameObject.transform.GetChild(j).GetComponent<ObjectScript>();

                    // SET variables to those from the respective Child's ('j') Script.
                    double mj = childjScript.ObjectMass;

                    decimal xj = childjScript.ObjectX;
                    decimal yj = childjScript.ObjectY;
                    decimal zj = childjScript.ObjectZ;

                    // CALCULATE the distance of the respective Child Body from the origin.
                    decimal dj = (decimal)Mathf.Sqrt((float)((xj * xj) + (yj * yj) + (zj * zj)));

                    // CALCULATE the absolute difference in distances.
                    decimal rij = System.Math.Abs(dj - di);

                    // CALCULATE the cartesian forces acting upon the respective body 'i'.
                    xWorkingSum += (decimal)-(((G) * mi * mj * (double)(xi - xj)) / ((double)rij * (double)rij *
(double)rij));
                    yWorkingSum += (decimal)-(((G) * mi * mj * (double)(yi - yj)) / ((double)rij * (double)rij *
(double)rij));
                    zWorkingSum += (decimal)-(((G) * mi * mj * (double)(zi - zj)) / ((double)rij * (double)rij *
(double)rij));
                }
```

```
            }

            // CALCULATE the cartesian accelerations acting upon the respective body 'i'.
            decimal ax = (decimal)((double)xWorkingSum / (mi));
            decimal ay = (decimal)((double)yWorkingSum / (mi));
            decimal az = (decimal)((double)zWorkingSum / (mi));

            // SET variables to those from the respective Child's ('i') Script.
            decimal vx = childiScript.ObjectVX;
            decimal vy = childiScript.ObjectVY;
            decimal vz = childiScript.ObjectVZ;

            // CALCULATE the cartesian velocity of the respective Child 'i'.
            vx += ((decimal)(E) * ax);
            vy += ((decimal)(E) * ay);
            vz += ((decimal)(E) * az);

            // SET the respective Child's ('i') velocity to the CALCULATED velocity.
            childiScript.ObjectVX = vx;
            childiScript.ObjectVY = vy;
            childiScript.ObjectVZ = vz;

            // CALCULATE the cartesian position of the respective Child 'i'.
            xi += ((decimal)(E) * vx);
            yi += ((decimal)(E) * vy);
            zi += ((decimal)(E) * vz);

            // SET the respective Child's ('i') 'hold' position (for storage, improved accuracy if updating the
positions once the entire set of calculations is complete for every child body per frame.) to the CALCULATED
position.
            childiScript.holdX = xi;
            childiScript.holdY = yi;
            childiScript.holdZ = zi;
        }

        // FOR every Child body, RUN 'ObjectScript's 'SetPos' sub-routine - using the respective body's 'hold'
cartesian position values in the Script to translate the body.
        for (int i = 0; i < transform.childCount; i++)
        {
            ObjectScript childiScript = gameObject.transform.GetChild(i).GetComponent<ObjectScript>();
            childiScript.SetPos();
        }
    }
}
```

Inspectors of the main Game Objects:

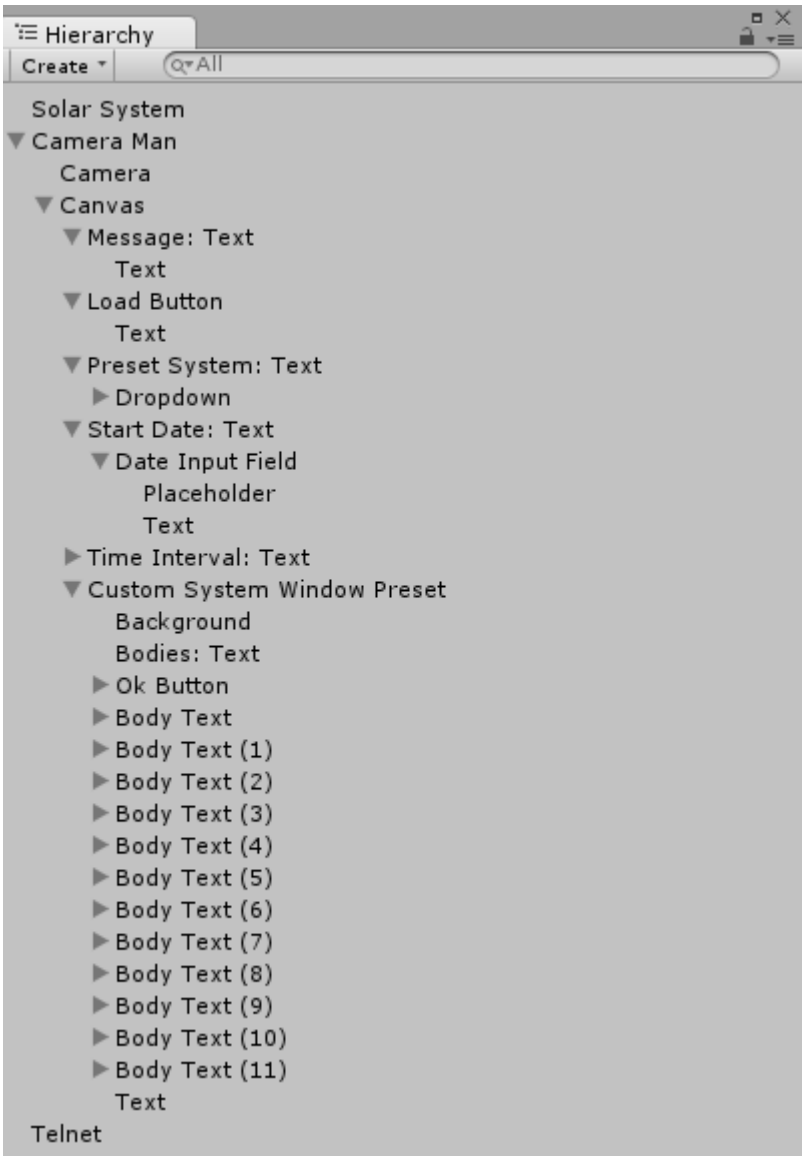

Fig 40 – Hierarchy of the Game Objects (prior to running the program)

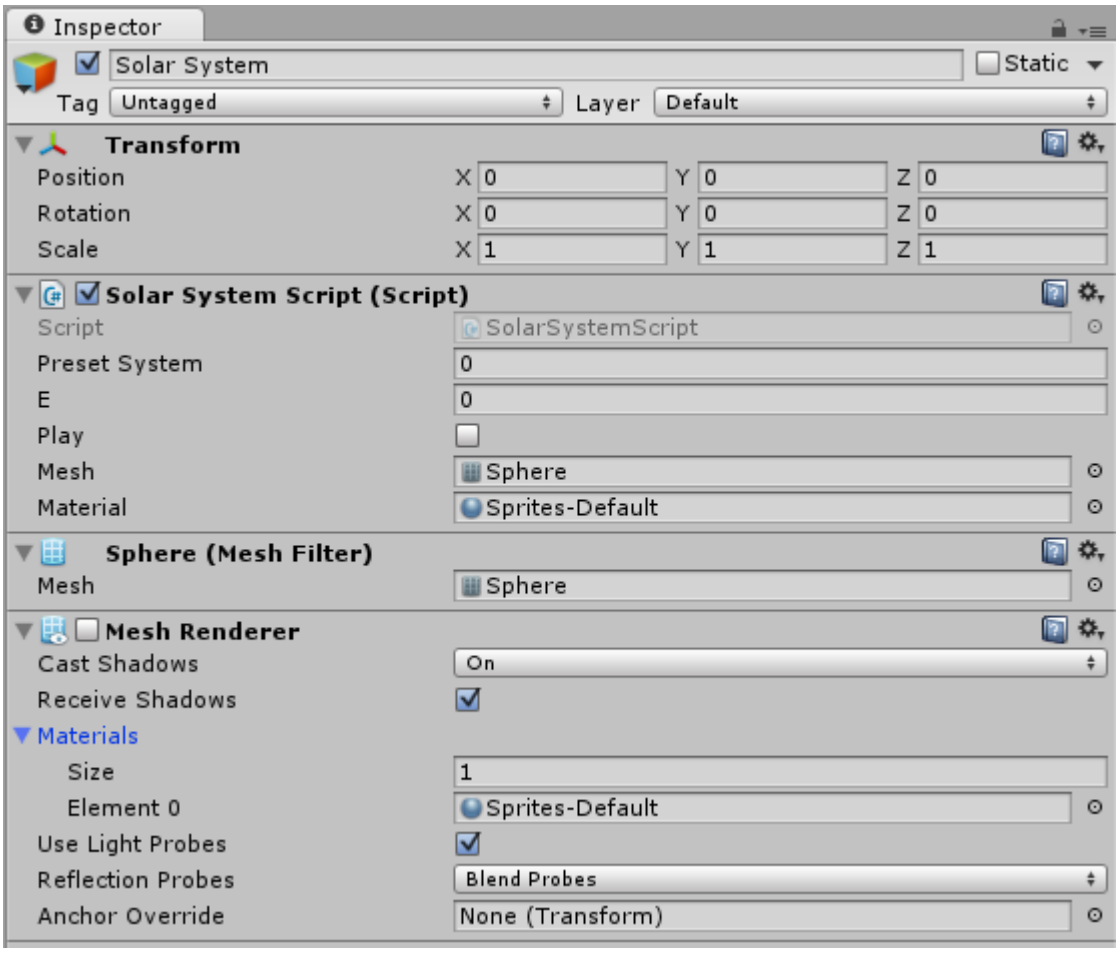

Fig 41 – Inspector of Solar System

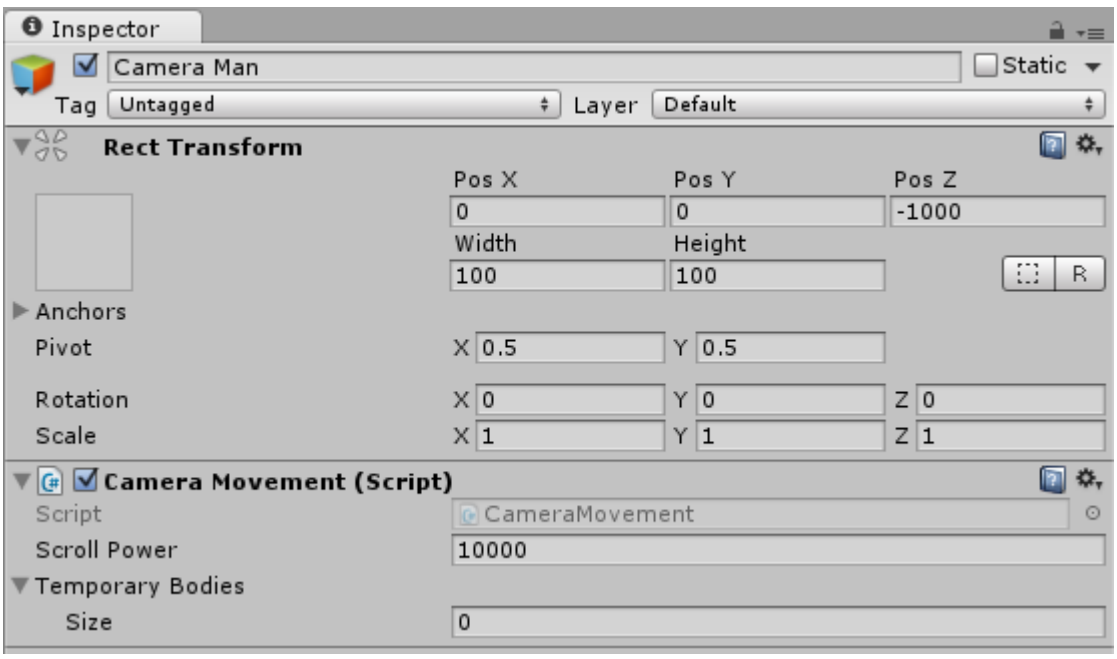

Fig 42 – Inspector of Camera Man

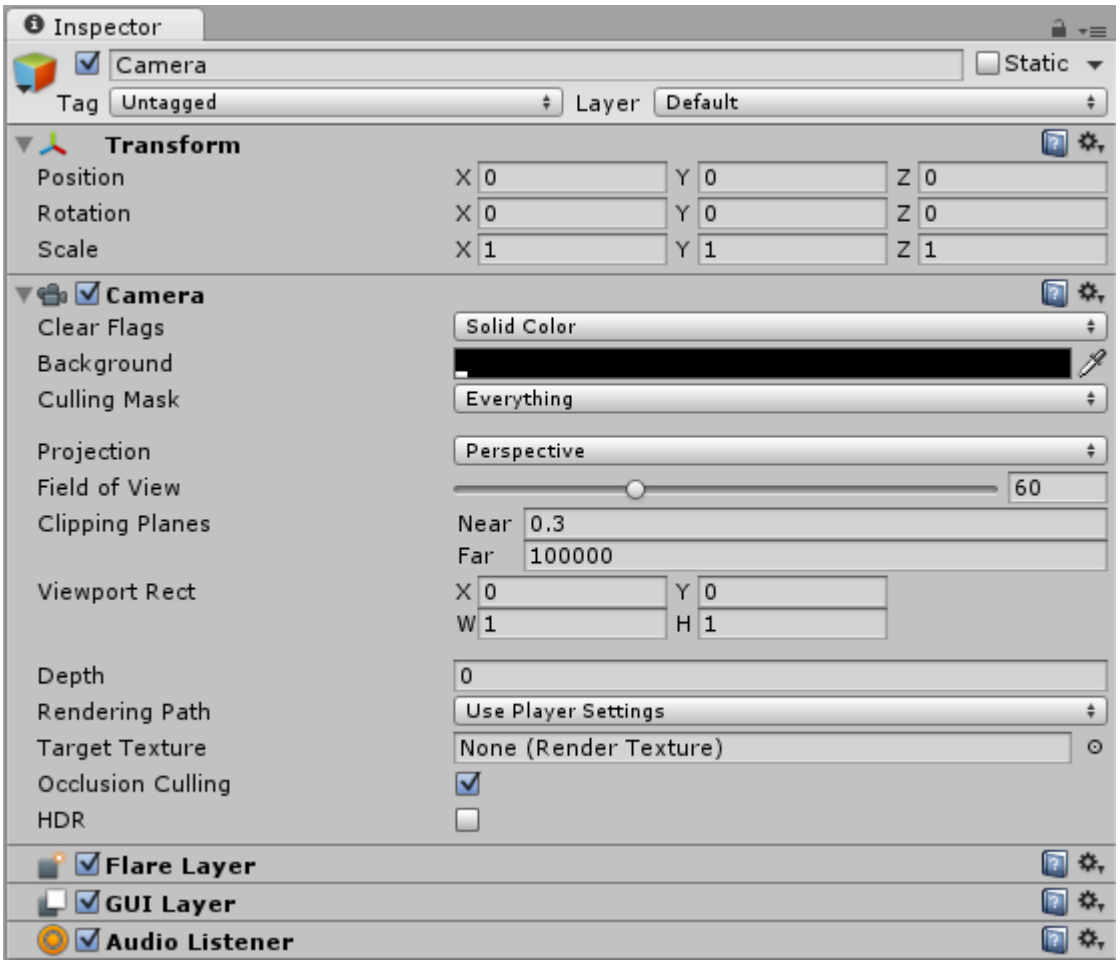

Fig 43 – Inspector of Camera

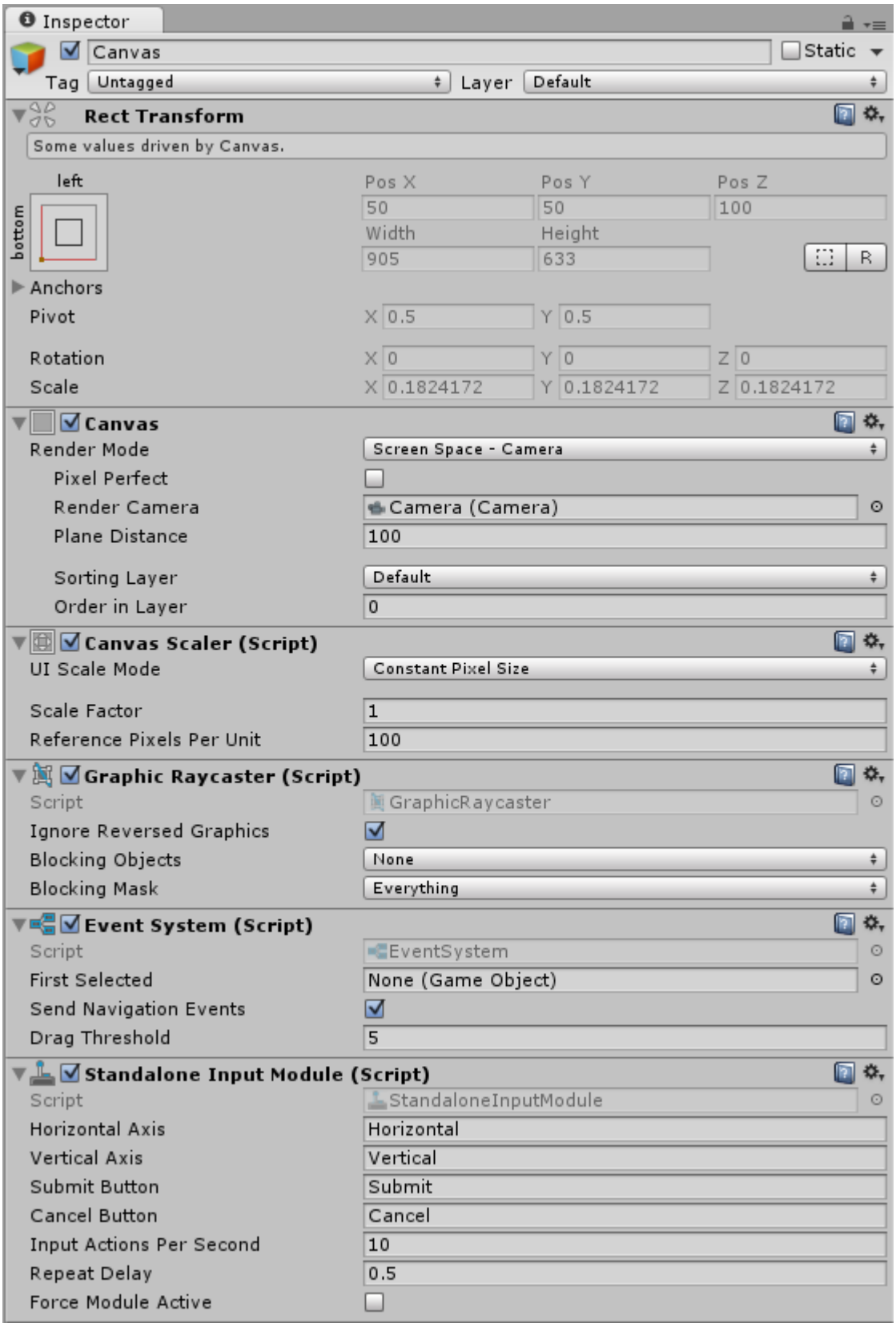

Fig 44 – Inspector of Canvas

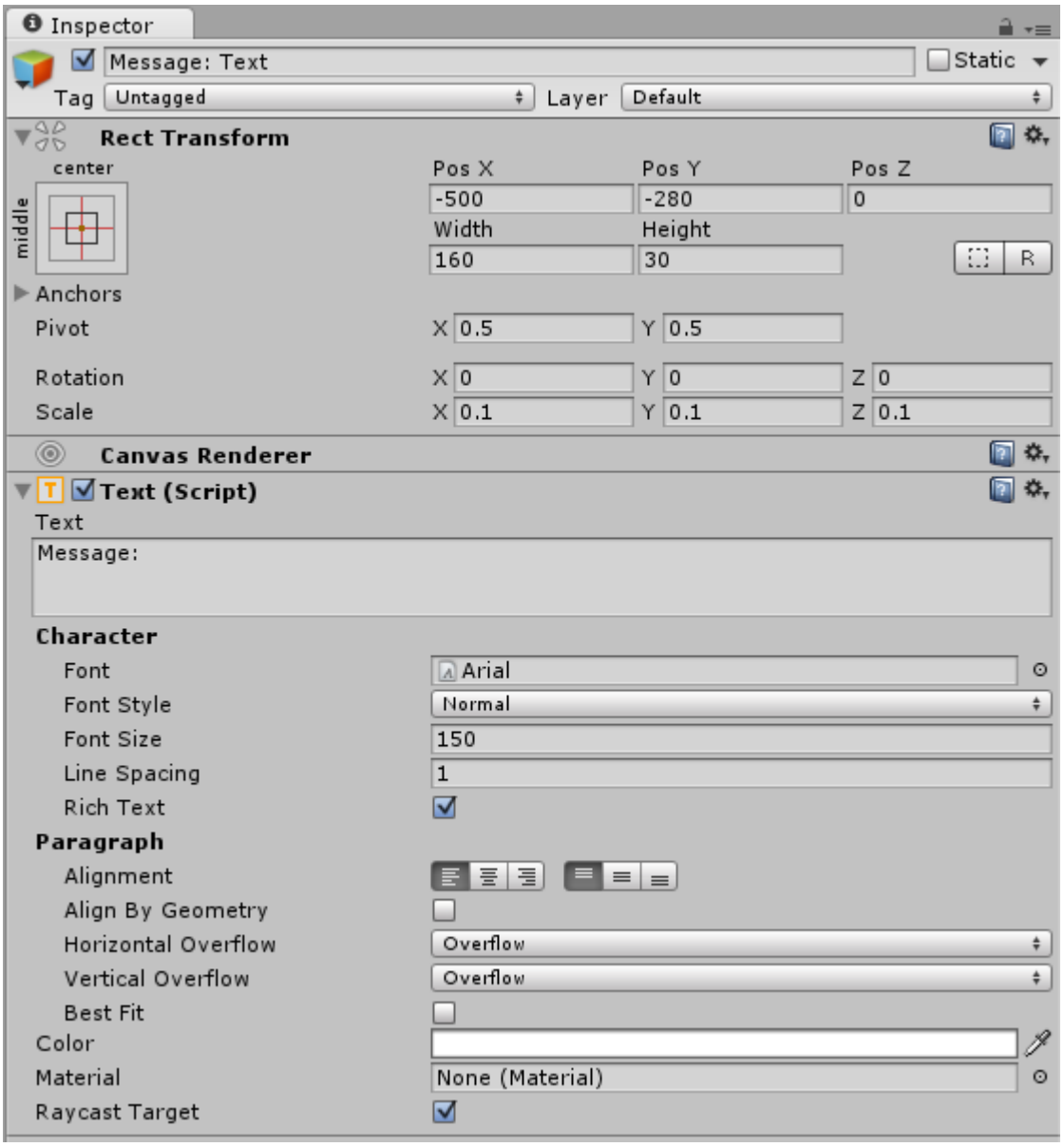

Fig 45 – Inspector of Message: Text

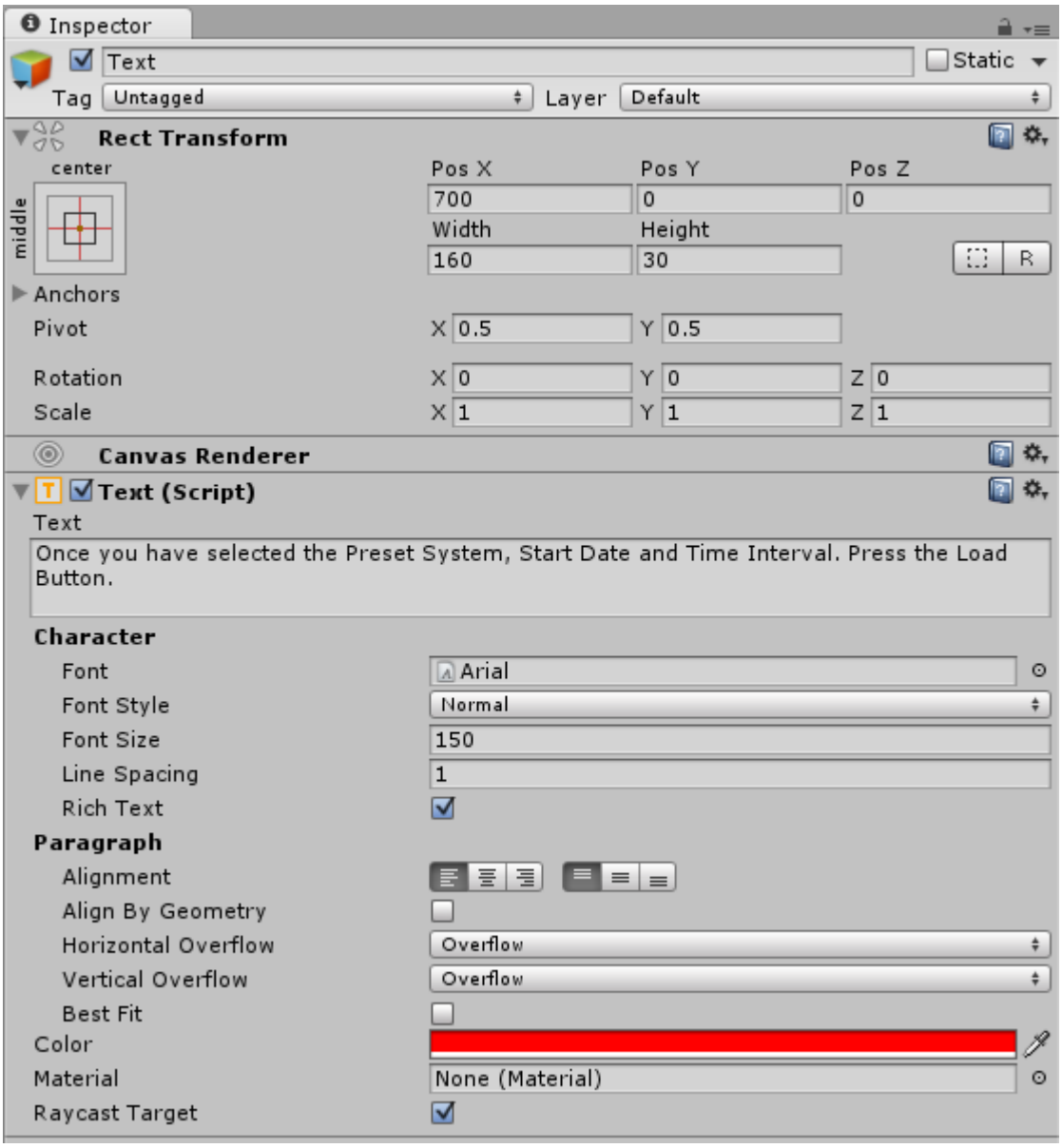

Fig 46 – Inspector of Message: Text/Text

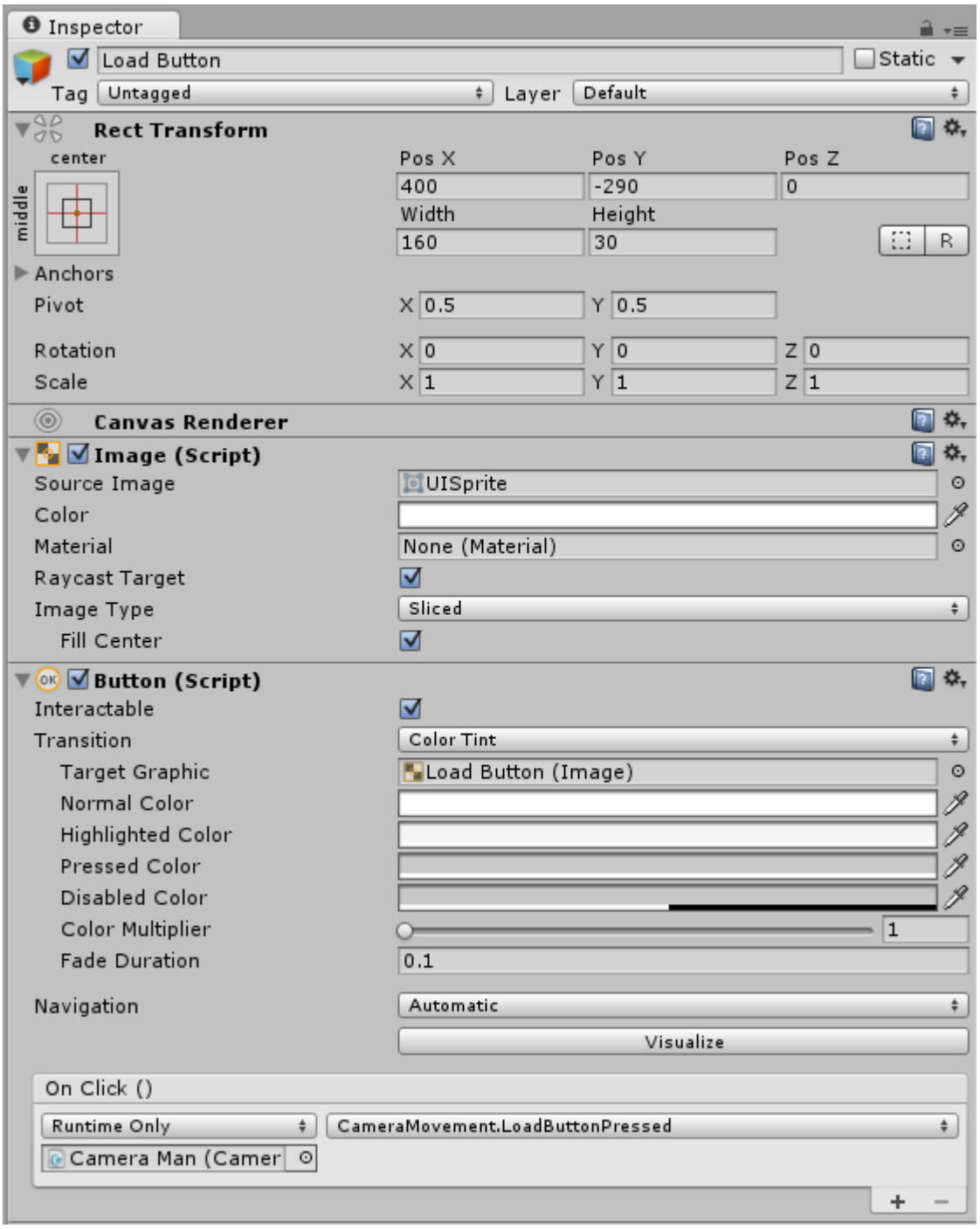

Fig 47 – Inspector of Load Button

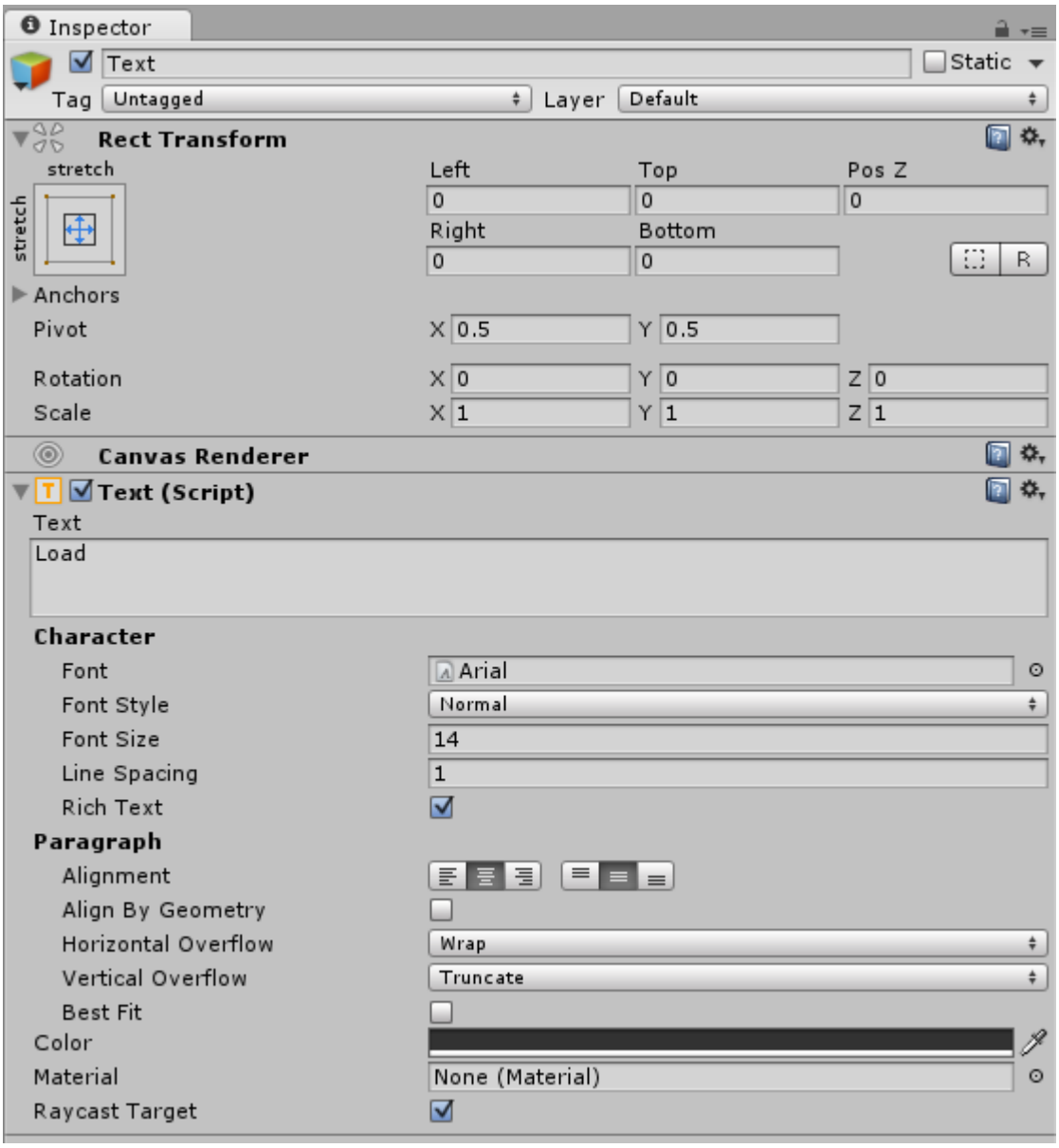

Fig 48 –Inspector of Load Button/Text

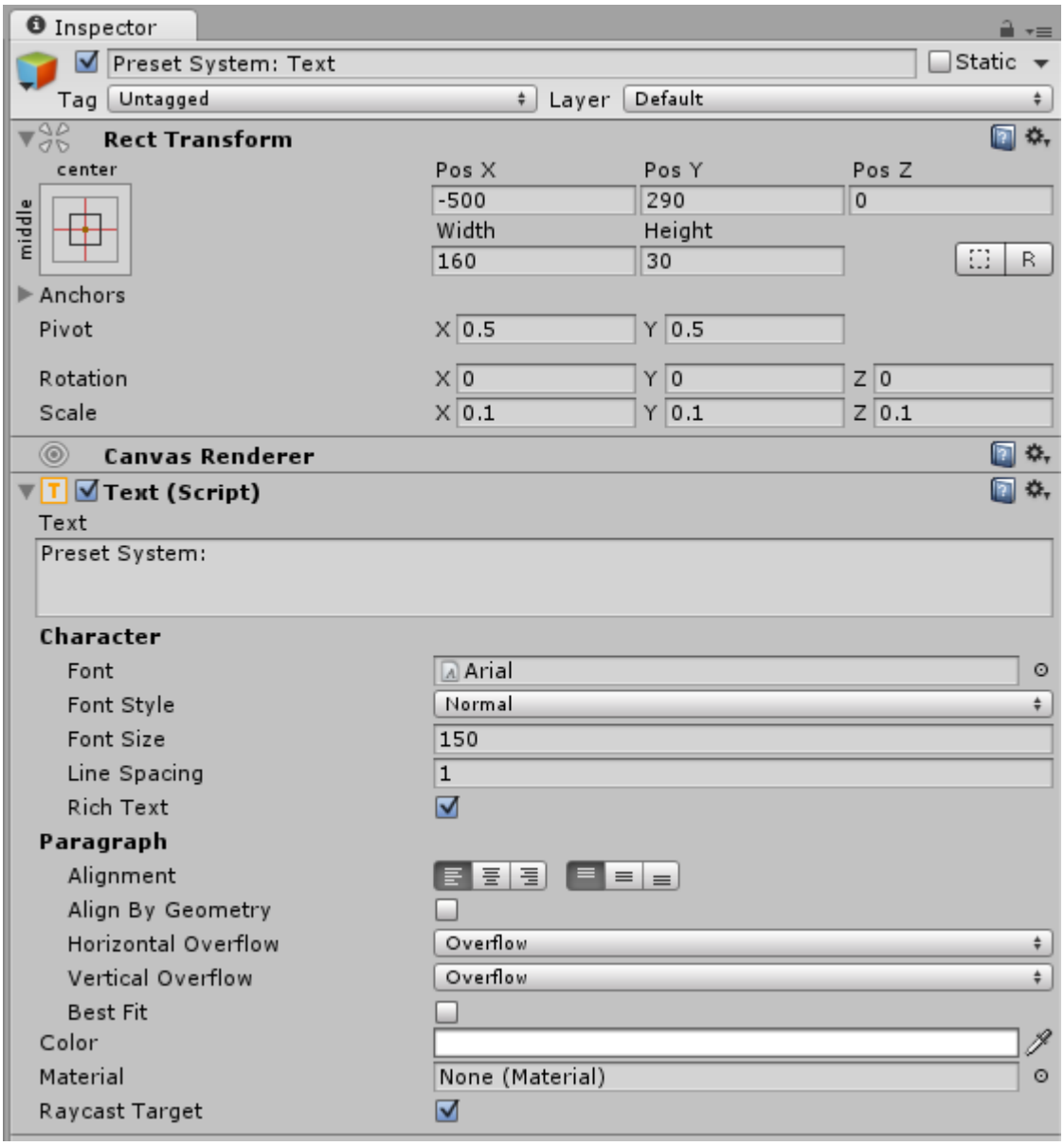

Fig 49 – Inspector of Preset System: Text

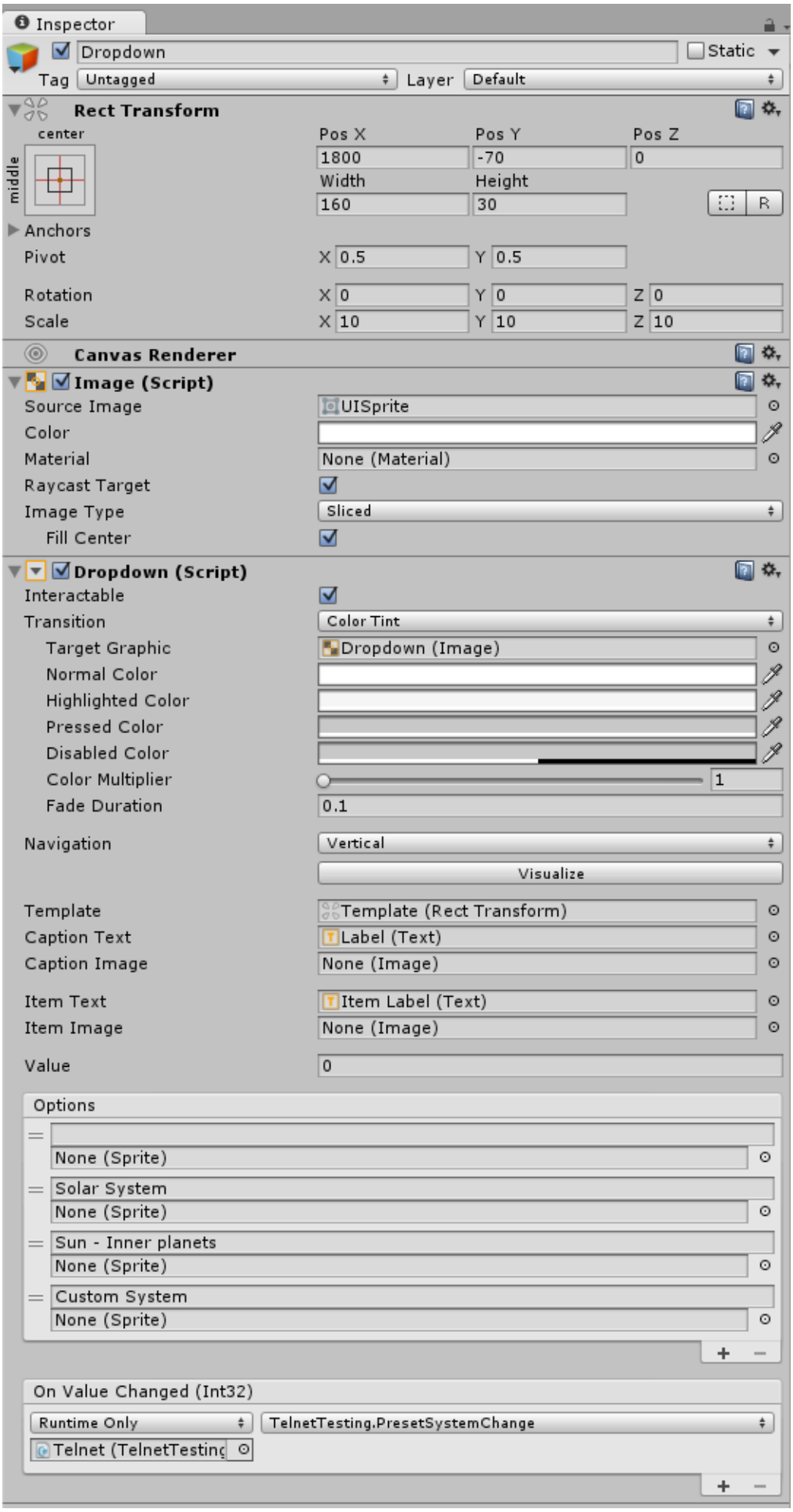

Fig 50 – Inspector of Preset System: Text/Dropdown

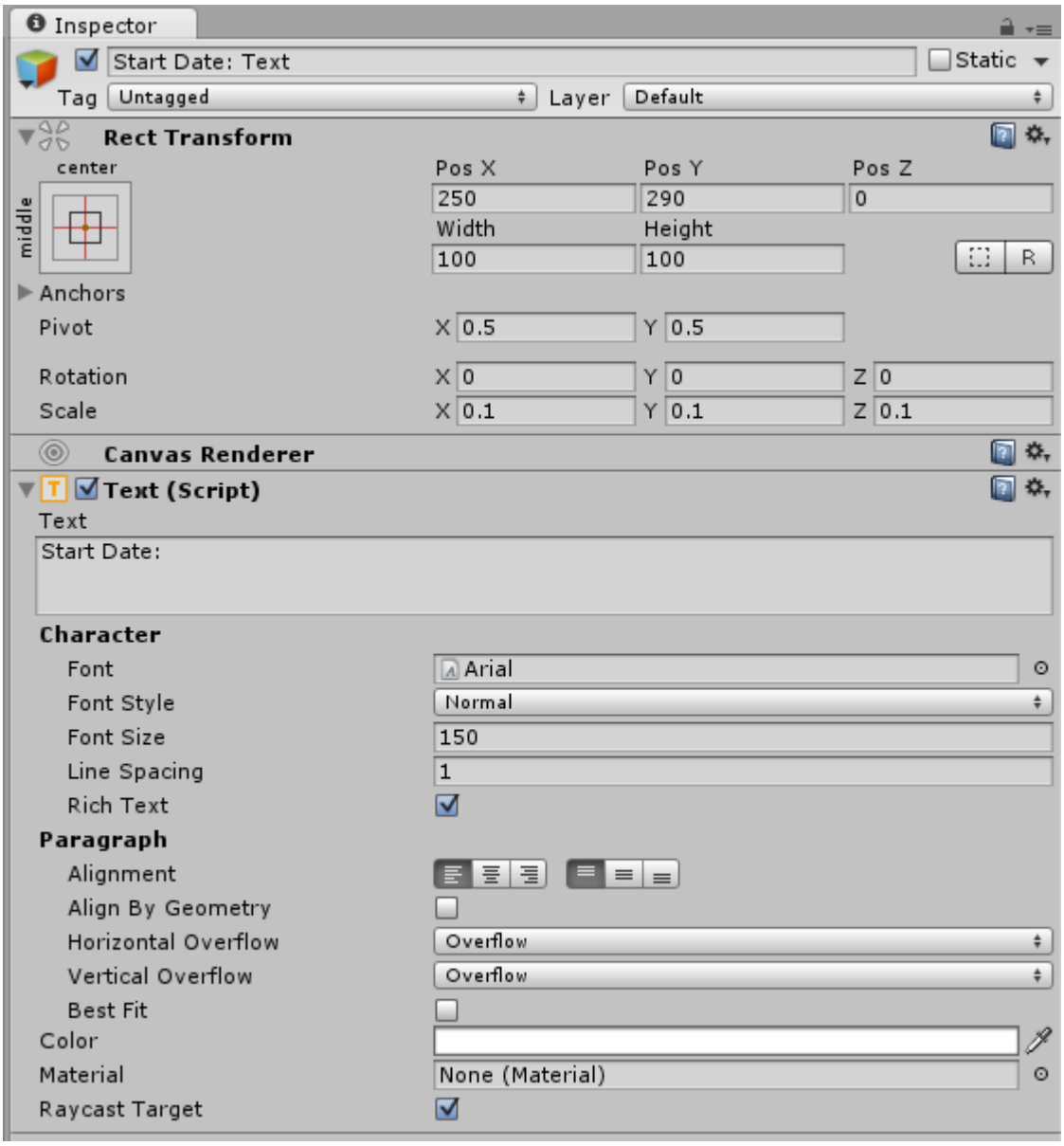

Fig 51 – Inspector of Start Date: Text

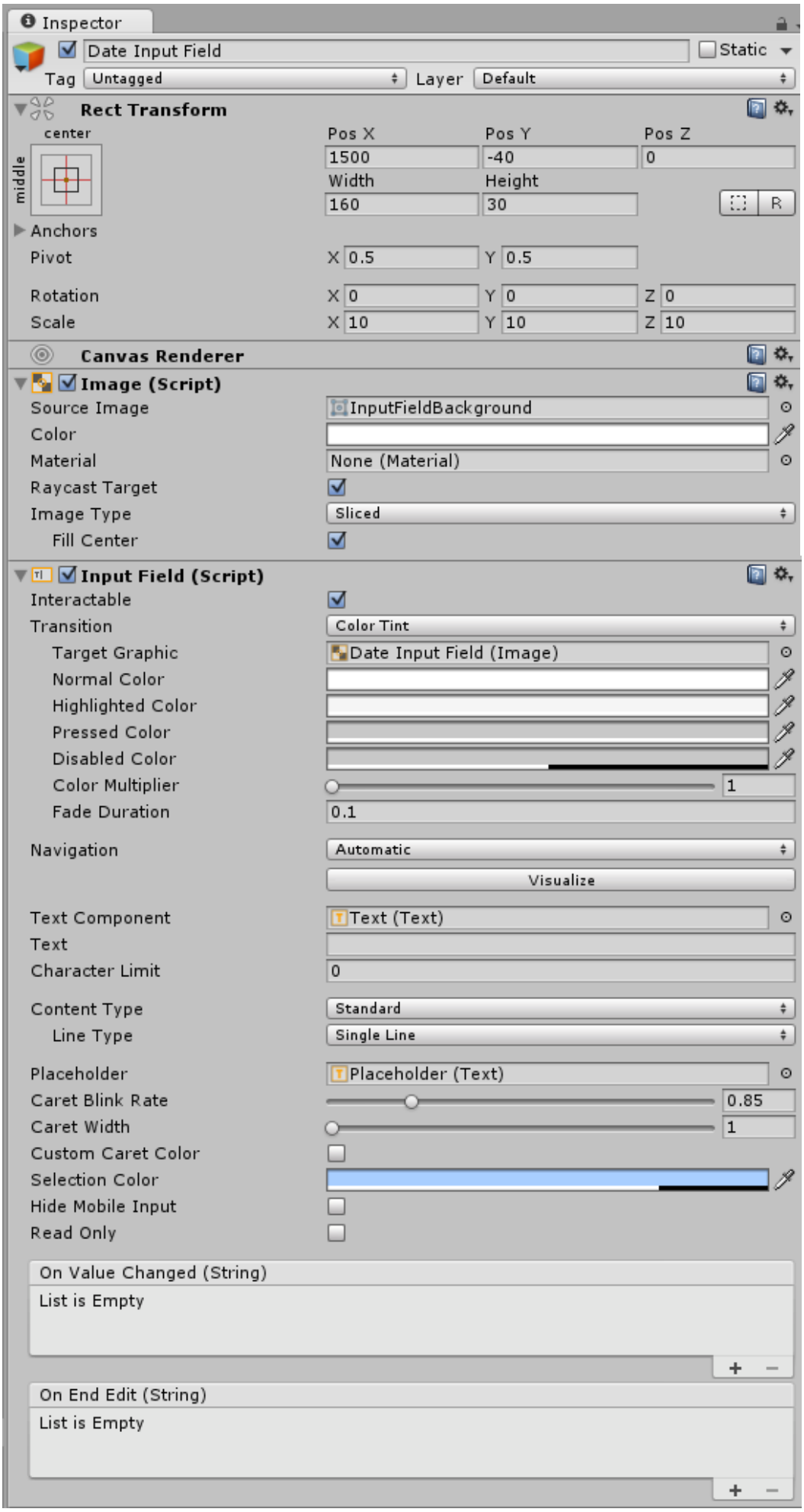

Fig 52 – Inspector of Date Input Field

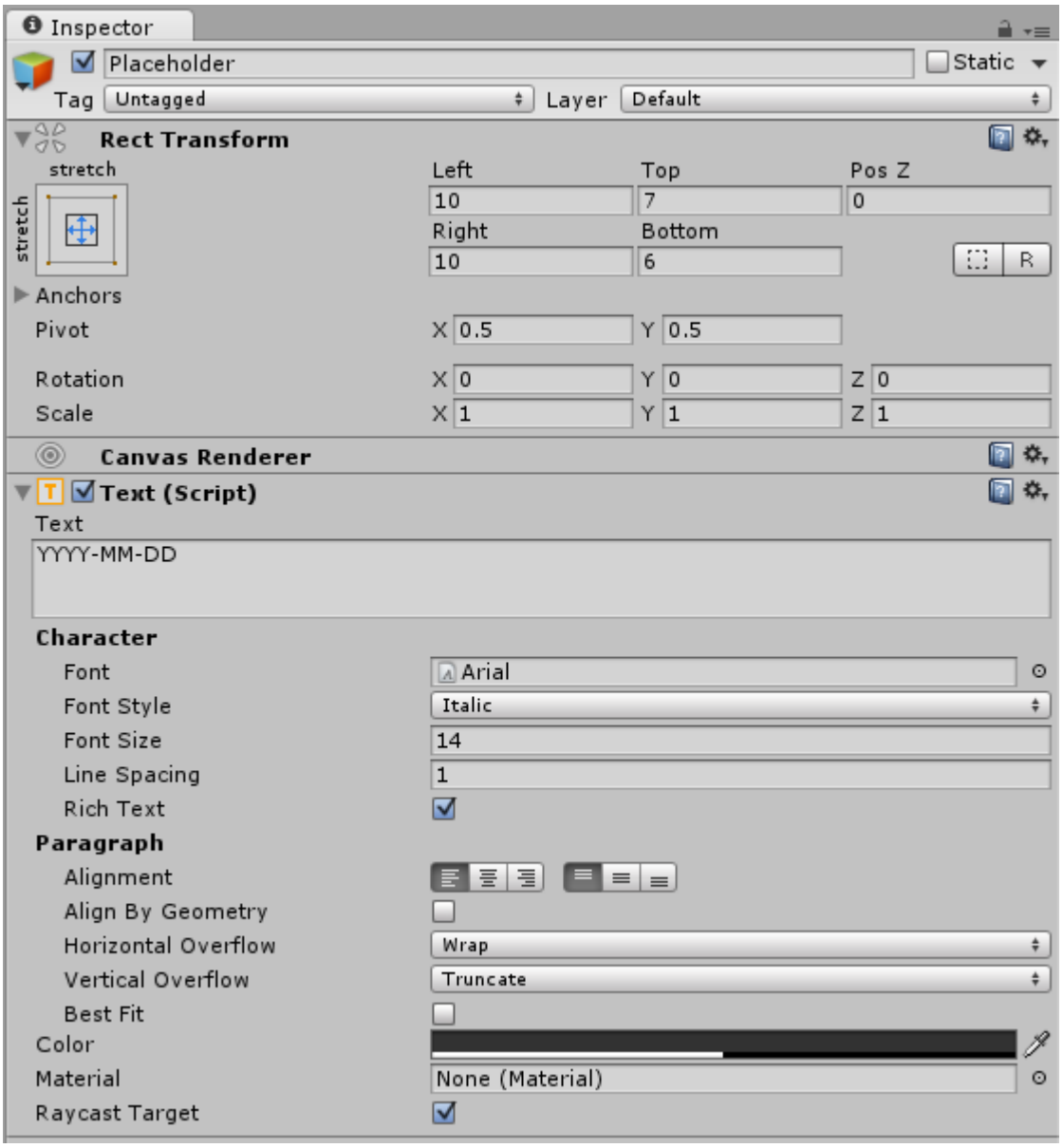

Fig 53 – Inspector of Date Input Field/Placeholder

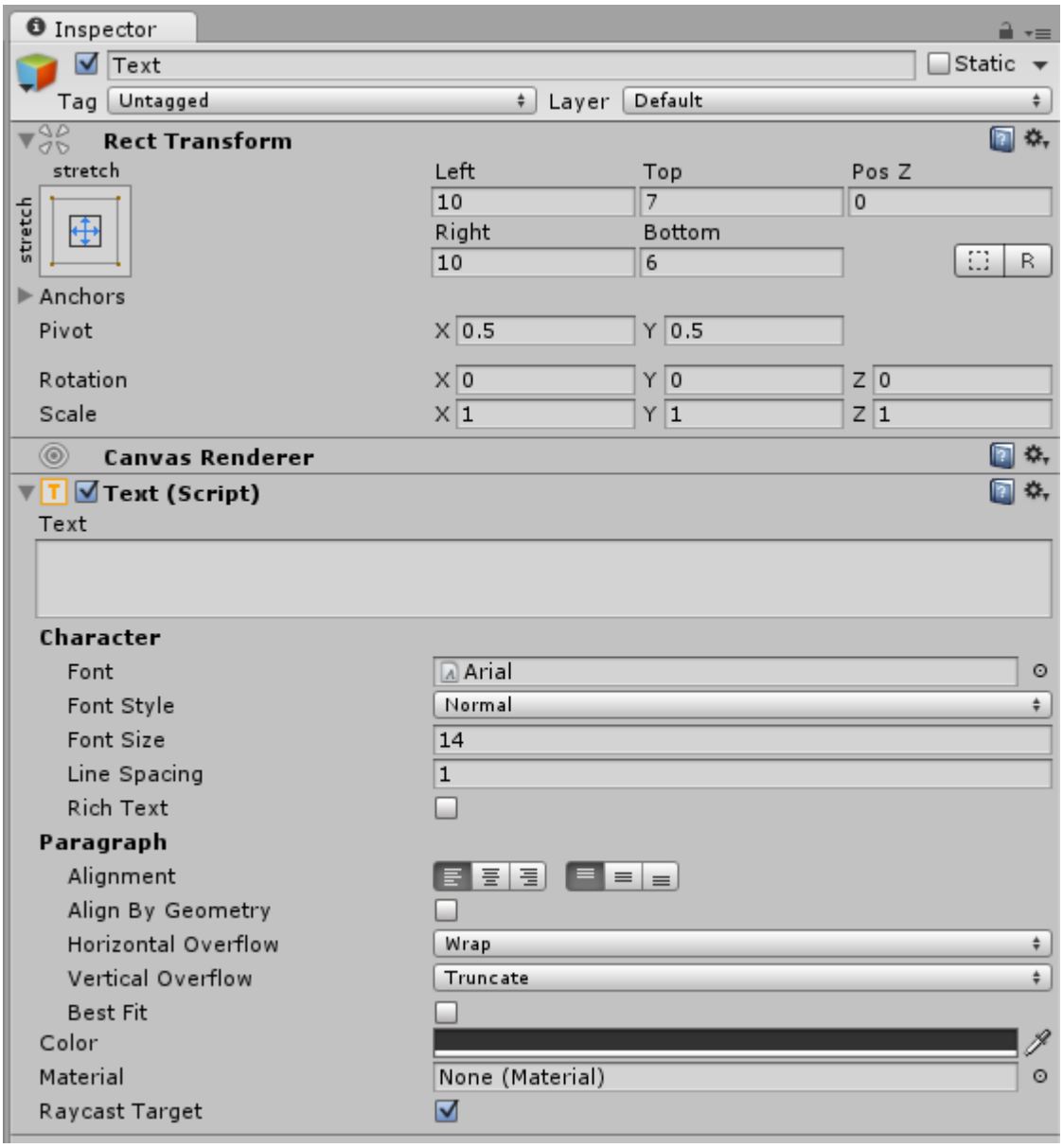

Fig 54 – Inspector of Date Input Field/Text

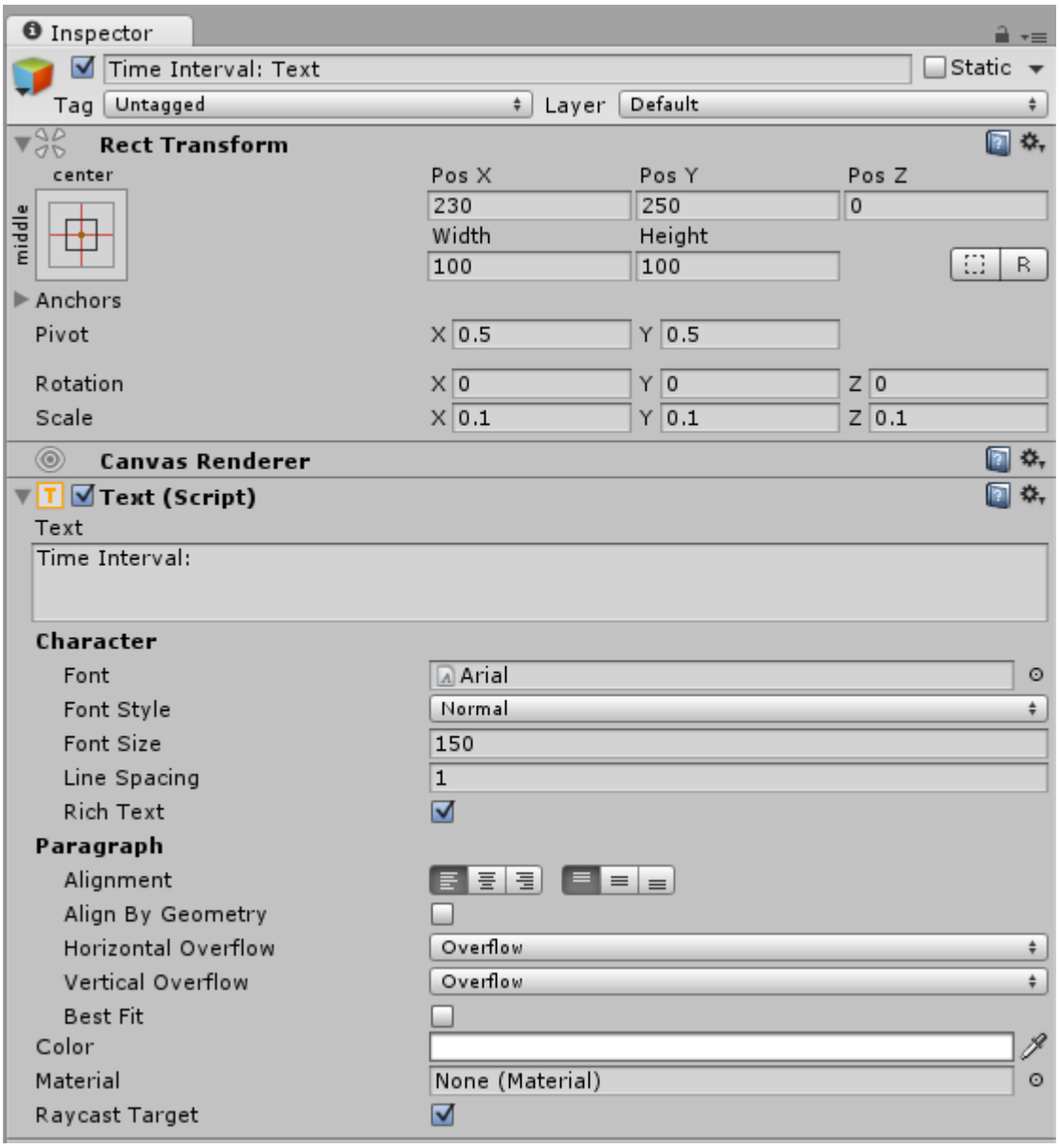

Fig 55 – Inspector of Time Interval: Text

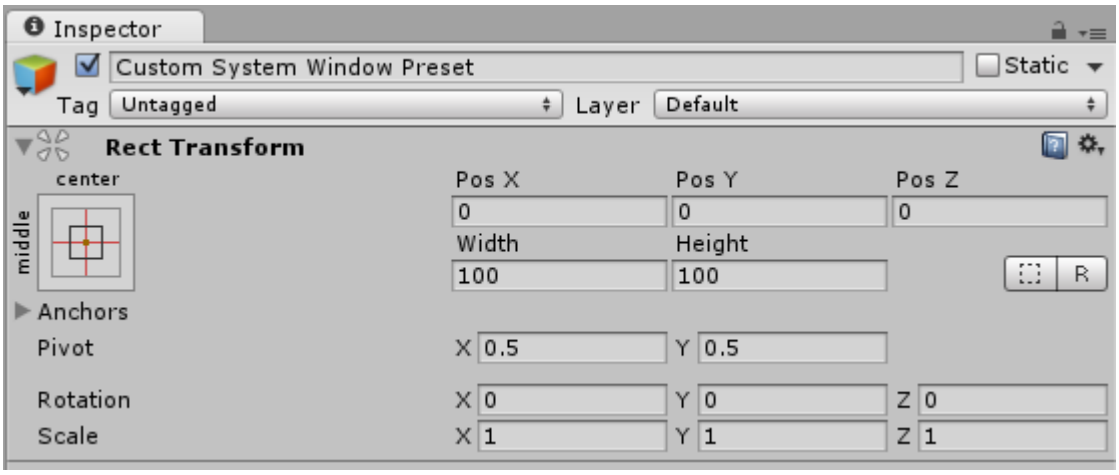

Fig 56 – Inspector of Custom System Window Preset

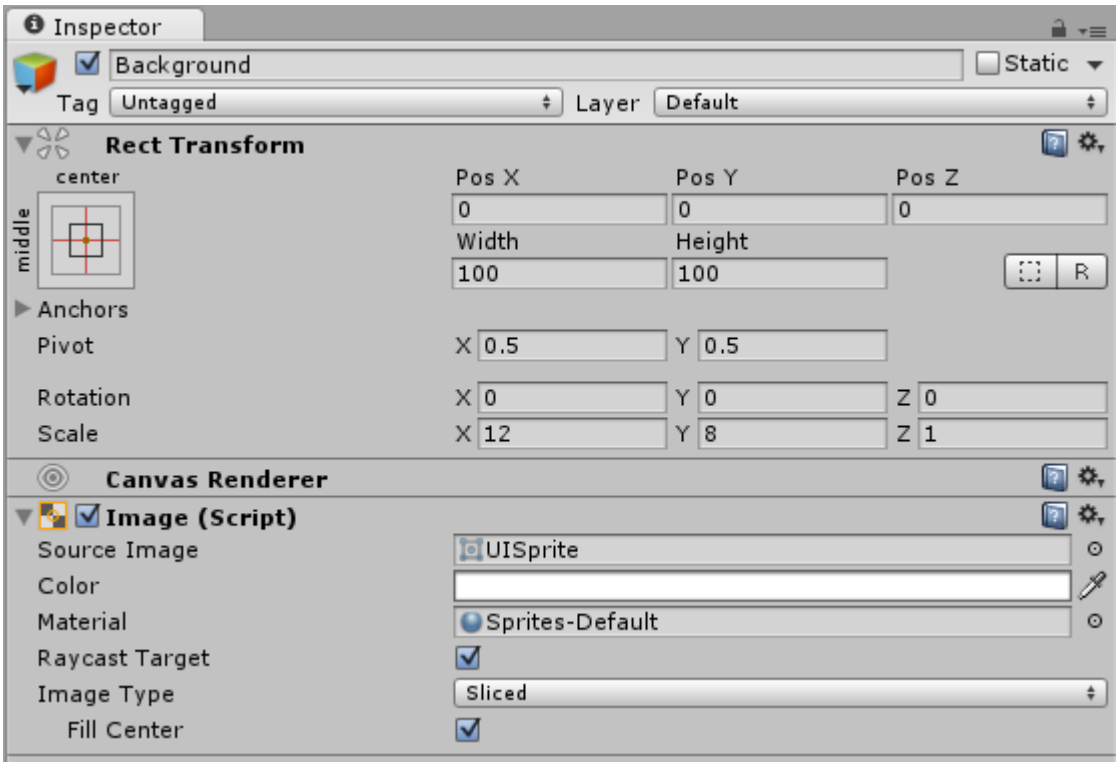

Fig 57 – Inspector of Background

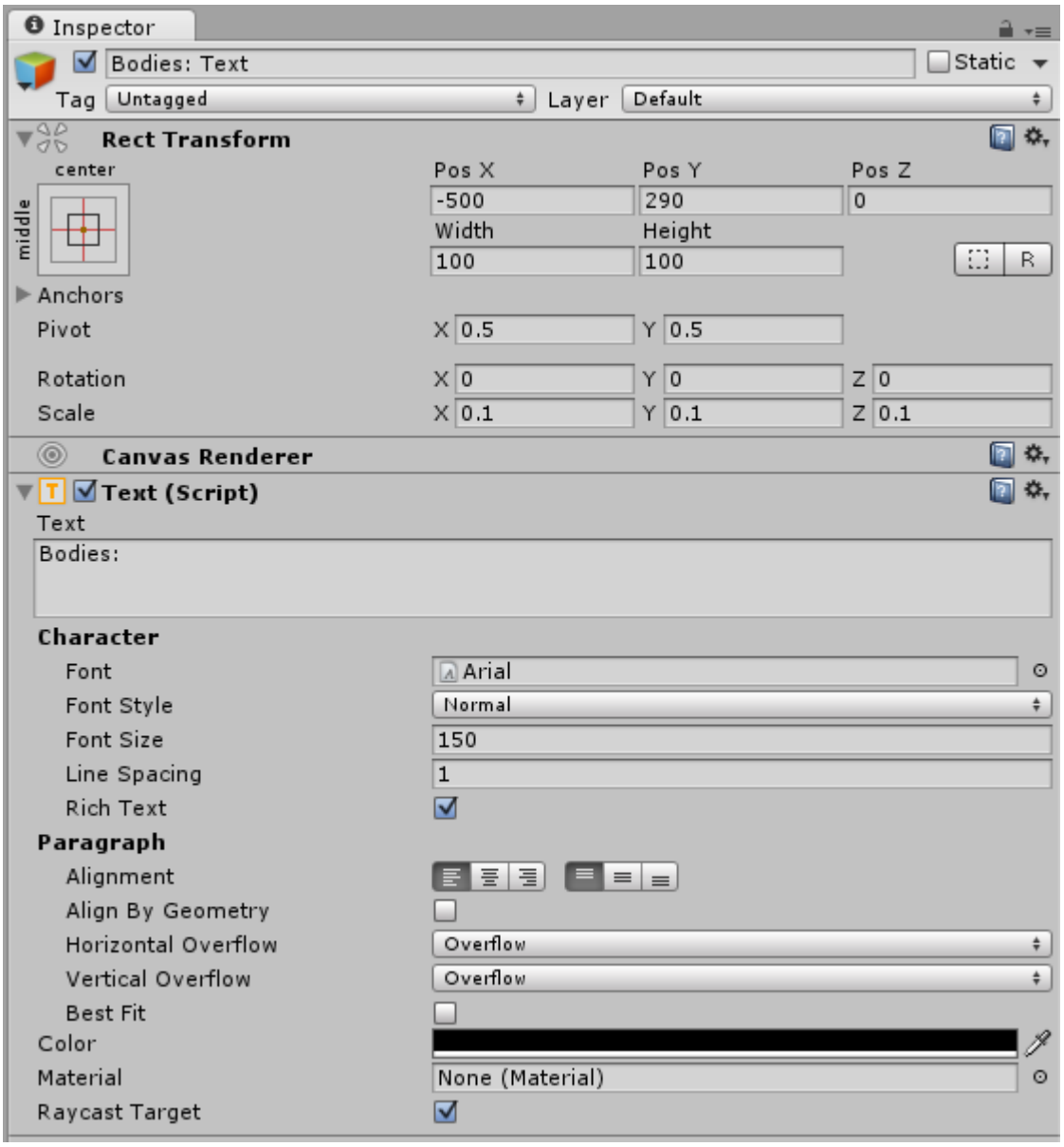

Fig 58 – Inspector of Bodies: Text

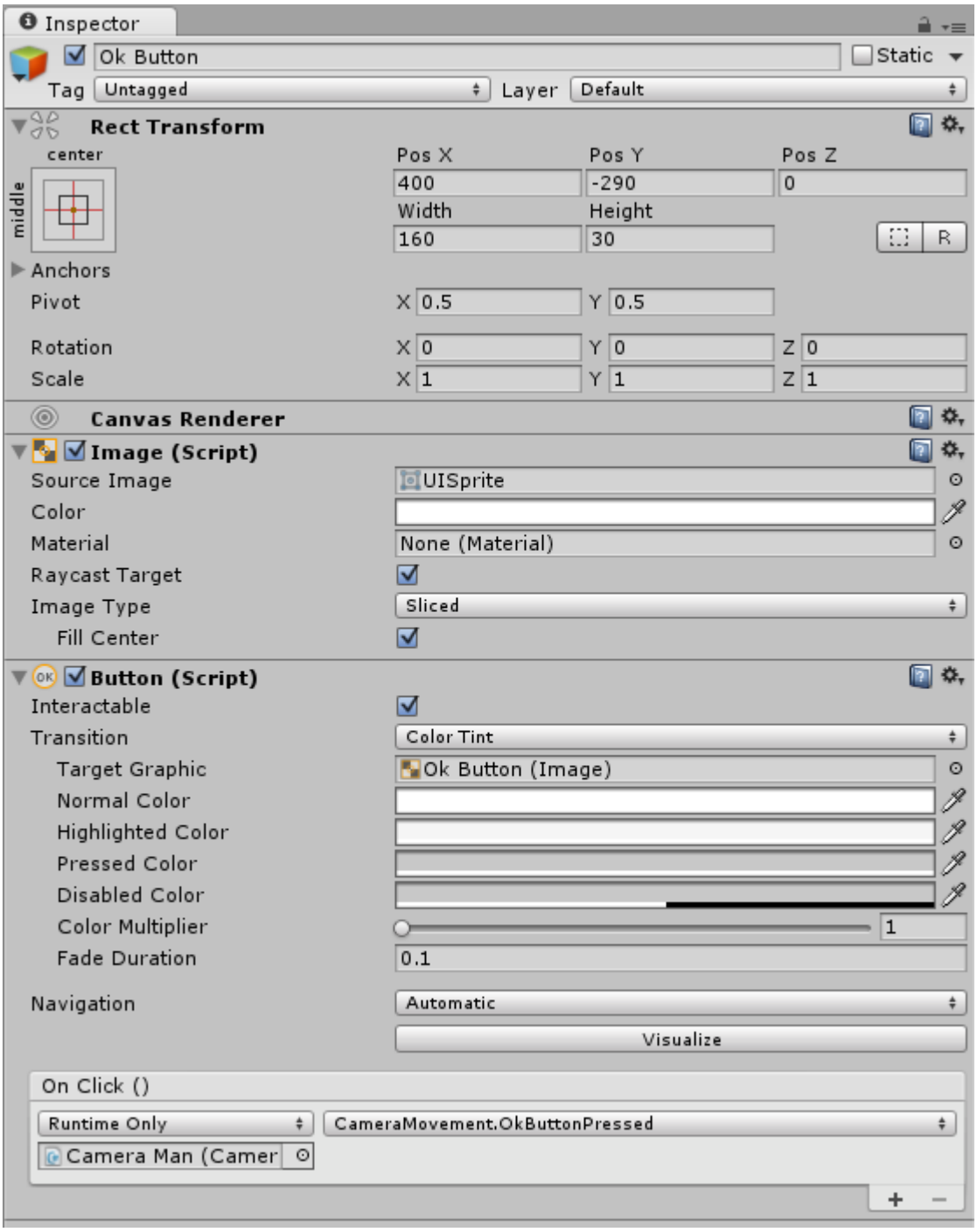

Fig 59 – Inspector of Ok Button

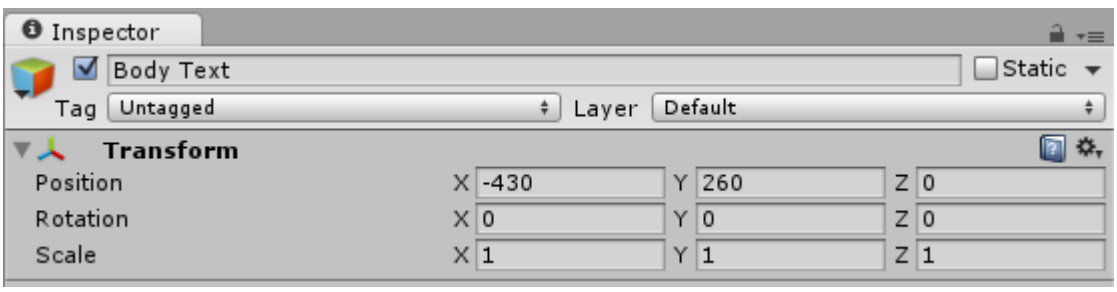

Fig 60 – Inspector of Body Text

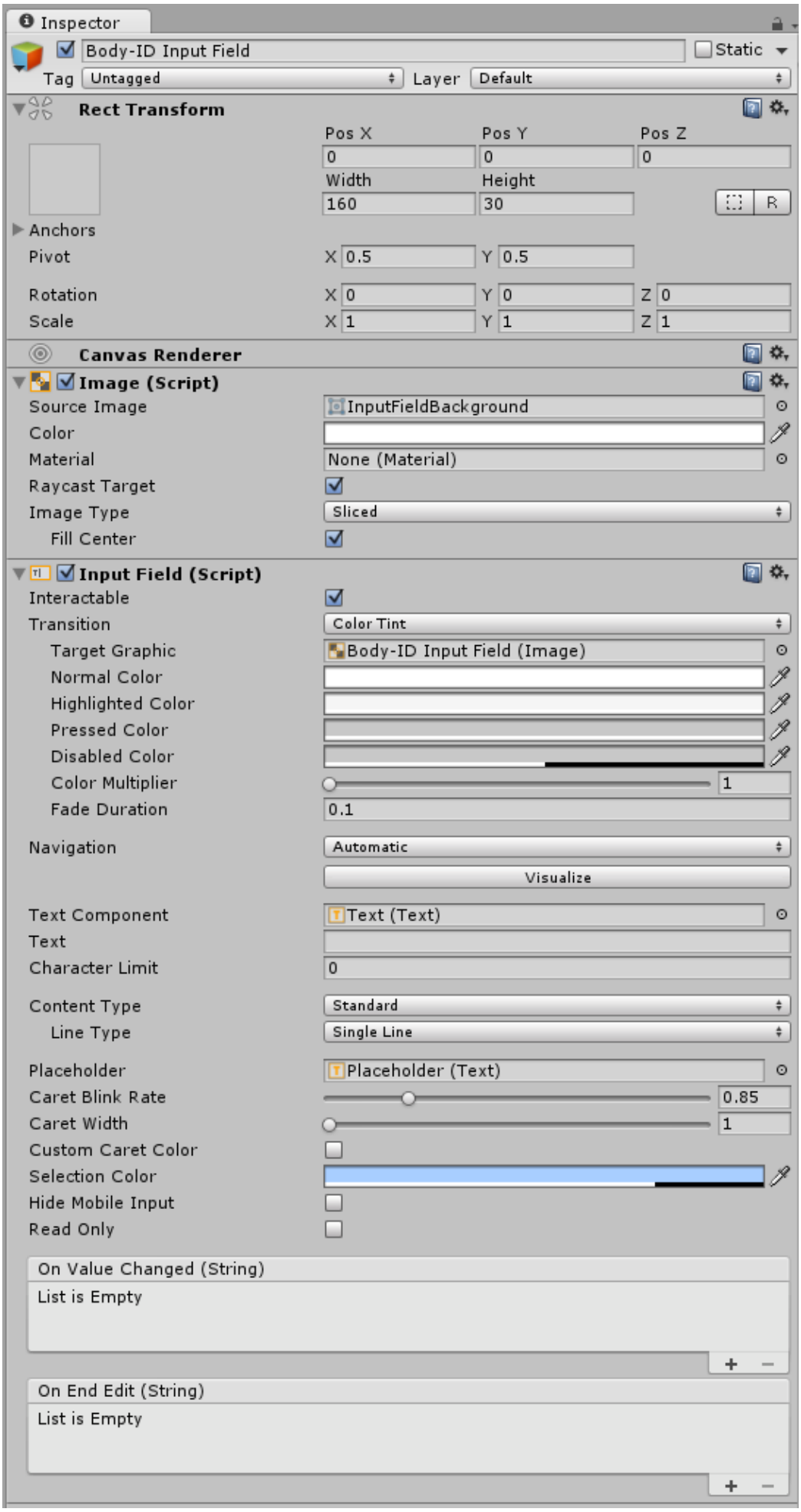

Fig 61 – Inspector of Body-ID Input Field

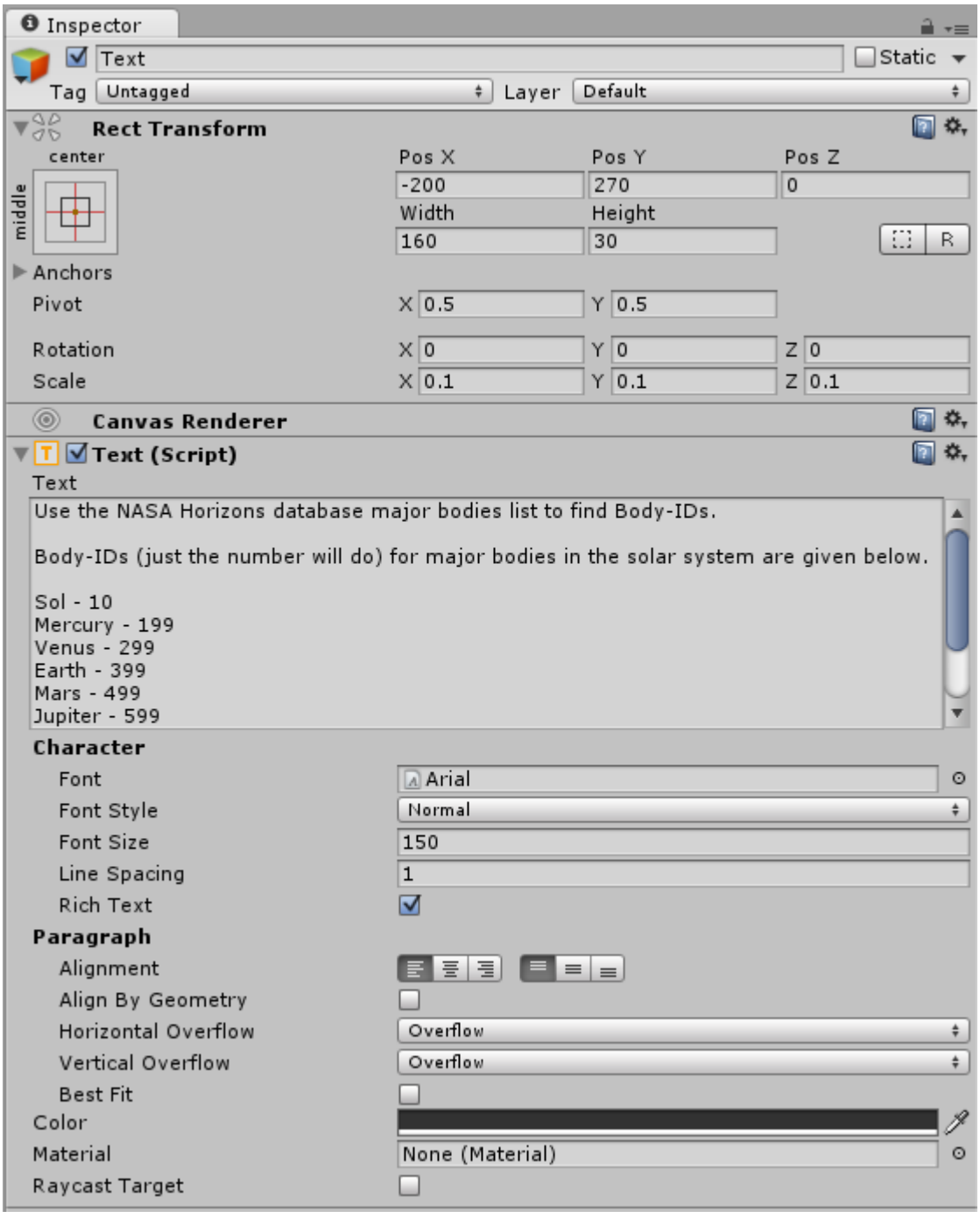

Fig 62 – Inspector of Custom System Window Preset/Text

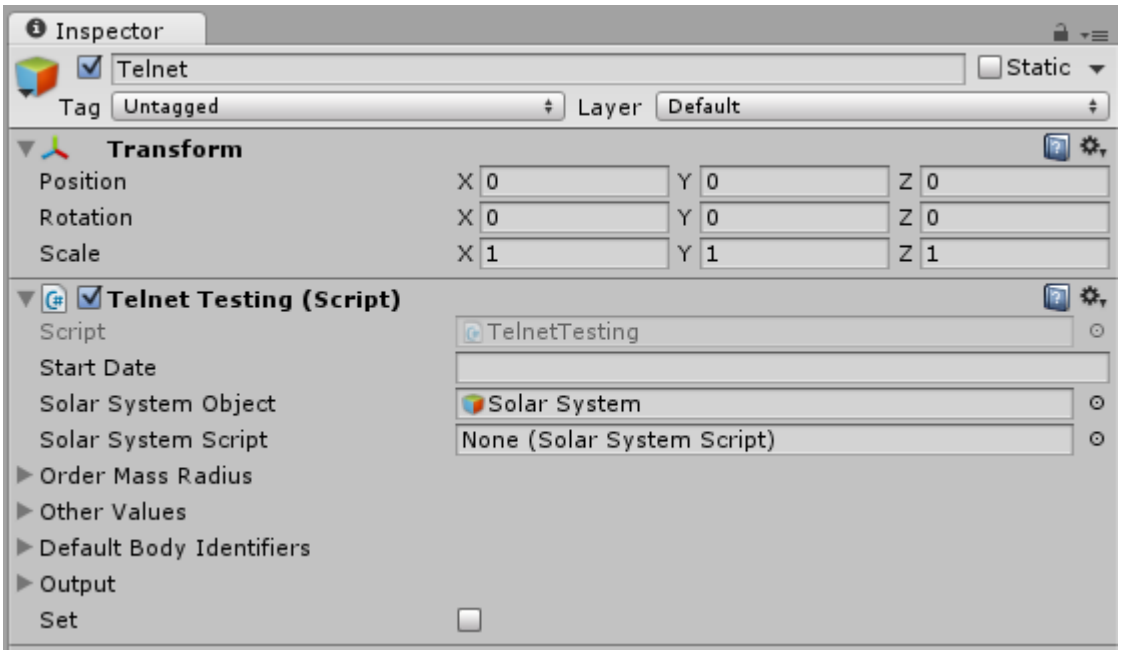

Fig 63 – Inspector of Telnet