Solar System Designer  
Documentation

[**Overview**](#_gkva01yt8abv) **2**

[Terminology](#_qsi31mupj4aw) 3

[Acronyms](#_pogtkes4zkb8) 3

[**User Manual**](#_if7p3qjevmld) **4**

[User Interface](#_2ck6kbj6hox) 4

[Menu Bar](#_3k82k7ehrfyj) 4

[Screens](#_d6bhlpqh101b) 5

[Create Body](#_le9brc249yxk) 5

[Preferences](#_s4vzj2muqix) 5

[Controls](#_3ni7w9m36sv2) 6

[Statistics](#_iesl2jr2n2r8) 6

[Time Bar](#_iapklcmnkz1z) 6

[**Instructor Manual**](#_o3mjcfwcckw0) **7**

[Smart Sparrow / AeLP](#_ouef1mzaahp7) 7

[Creating the Component](#_899fi6zzwf8) 7

[Generating a Solar System](#_hzzsv5n66jgr) 7

[Setup the Component to start with the Solar System](#_b4epryfrtg6b) 8

[Inspector Variables](#_jywdudl26lfv) 8

[**Technical**](#_2tvktmf0p2br) **9**

[The State String](#_u2dzt4u9uccx) 9

[Physics](#_dafatr5ac1hm) 10

[PhysicsBody.cs](#_8i1y4cjuhg3q) 10

[Gravity.cs](#_plq1c5rt6f3e) 10

[How New Positions are Calculated for a Body](#_cpwqzbhlfus8) 11

[How is an Initial Velocity Calculated](#_cec5qh5p68o7) 13

[Non-Implemented Features](#_rt91eql7laxw) 14

[Technical Limitations](#_605dctuebing) 14

[Inspector Value Precision](#_pu85cuh3ijwr) 14

[Running Update Rate](#_uaqhmkp5vphh) 14

[Send/Receive Race In Running Simulation](#_bkz55gxx9km6) 14

[**Unity Merging Process**](#_jf7mmqn2gh3f) **15**

[Git Branch Set-Up](#_7fn52pd7hg1r) 15

[Initial Merging](#_qhyvz26pjtid) 15

[Final Merging](#_29szx4qr67gq) 15

[Testing](#_hrhuudrinu2p) 15

[Performing Object Merge](#_uzyih9hniqk8) 15

# Overview

The solar system designer project is for ASU’s center for Education Through Exploration (ETX). The solar system designer is a web-based simulation, which allows the user to design their own solar system. The simulator is designed for educational purposes to teach students, ages 10-17, how a solar system works. The project is uploaded to ASU’s School of Earth and Space Exploration and NASA’s Science Mission Directorate website: https://infiniscope.org/ and is similar to the Small World simulator located on the website. The motivation behind the simulator was to develop a creative and immersive learning environment, which will ease the user into gaining a sound understanding on what makes a solar system work.

The solar system designer simulator is developed in HTML5 and is integrating with the Smart Sparrow API (SS\_API). The SS\_API allows for a step-guided, interactive and adaptive learning environment. This allows for the simulation to be guided through user objective completion and will adapt according to a user’s choices throughout the simulation. Utilizing HTML5, we created a graphical model of space such as planets, stars, space itself and everything else that is involved in a solar system.

Since the simulation being used for educational purposes. The solar system designer will be integrating real physics such as planetary motion and other celestial mechanics. This will allow users to see what makes a solar system works and why a solar system behaves the way it does.

## 

## Terminology

**Body-** Any object within the simulation that has mass. This includes stars, planets, moons, and asteroids.

**The Inspector-** While in AeLP Preview mode, the inspector allows you to view details of both the lesson and simulation.

**State-** The entire status of the simulation: If it’s paused, it’s running speed, and all stored data of all bodies within the solar system.

**State String-** A string storing the state.

## Acronyms

**SSD -** Solar System Designer.

# User Manual

## User Interface

The major components that contribute to the User Interface(UI) are the menu bar, the sim manipulation panels and the time bar.

### Menu Bar

The menu bar has consists of four items. Allowing you to manipulate the simulation as a user.



**File**

**Import-** Import the entire state of the solar system from a line of text.

**Export-** Save the entire state of the solar system into a line of text.

**Presets-** Select from a pre-generated solar system.

**Solar System:** Our Solar System.

**Collision:** Body collision occurrence.

**Gas Giant:** Wobbling star, Smaller planets.

**Larger Outer Body:** Tug causes elliptical loop.

**Sim**

**Add Body-** Ability to add a body to the simulation.

**Clear-** Ability to clear a simulation of all bodies.

**Options**

**Preferences-** Ability to change menu display settings as well as 

keyboard and mouse movement speeds.

**Controls-** Ability to change keyboard movement and zoom controls.

**Help**

**Help-** Ability to access listed controls, instructions on inserting a

body or removing a body.

**About-** Ability to view project version, description and team members.

**Statistics**

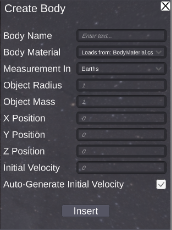
**FPS-** Frames Per Second.

**V-** View the Body Statistics screen.

### 

### Screens

#### Create Body

**Description:** Screen provides input fields to create a body with the following inputs:

**Body Name=String:** Values accepted are letters and numbers.

**Body Material=Item:** Dropdown with set values.

**Measurements In=Item:** Dropdown with set values.

**Object Radius=Number**: Must be greater than 0.

**Object Mass=Number**: Must be greater than 0.

**X Position=Number:** Any number value is acceptable.

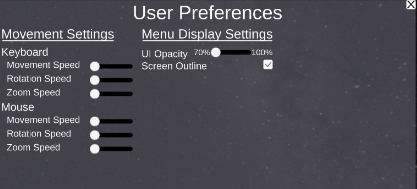
**Y Position=Number:** Any number value is acceptable.

**Z Position=Number:** Any number value is acceptable.

#### 

#### Preferences

**Description:** Screen provides slider to adjust the speed at which the peripherals move the simulation.



**Keyboard Movement Settings:**

**Movement Speed:** Change speed.

**Rotation Speed:** Change speed.

**Zoom Speed:** Change speed.

**Mouse Movement Settings:**

**Movement Speed:** Change speed.

**Rotation Speed:** Change speed.

**Zoom Speed:** Change speed.

**Menu Display Settings:**

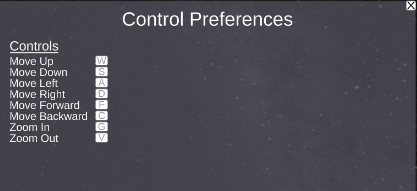
**UI Opacity:** Change UI opacity percentage.

**Screen Outline:** Adds outline or border around screens. (To Be Completed)

#### 

#### Controls

**Description:** Screen provides input fields to create a body with the following inputs.



### 

### 

### 

### 

### 

#### Statistics

**Description:** Screen provides text labels to display statistics of selected body from the dropdown.

### 

### 

### 

### 

### 

### Time Bar

The time bar allows you to control the simulation by allowing a user to reset, pause or play the simulation.

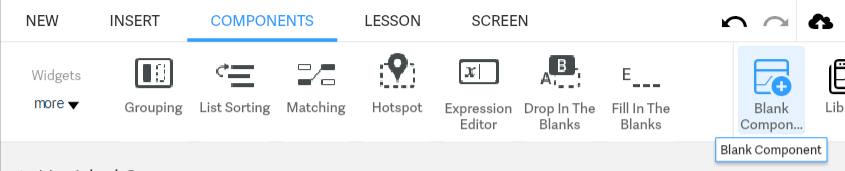
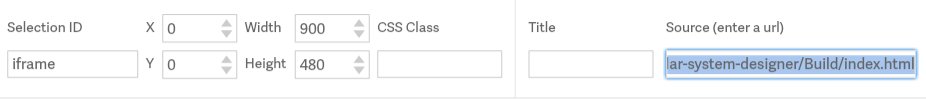


# Instructor Manual

## Smart Sparrow / AeLP

See Smart Sparrow’s guide for general components and lesson setup. The instructions provided here are to interface Smart Sparrow with the Solar System Designer.

### Creating the Component

1. After creating a Scene: Under **Components**, click on **Blank Component**.
2. Move and resize the component to your specifications. *900x480 or 480x480 (WidthxHeight) is recommended, though most others will work as well.*
3. In **Source (enter a url):**  
   <https://jbresett.github.io/solar-system-designer/Build/index.html>

### Generating a Solar System

There are a few ways to build the initial solar system:

1. A few Presets can be loaded. From the active Component, **File, Preset,** and select the preset. *The system can also be further modified by the methods below.*
2. Bodies can be created through the component itself (**Configure**, **Preview,** or running the source directly**)**, adding the bodies.
3. Bodies can be activated and modified within the Inspector window as well. See Inspector section below for further details.
4. Manually creating and/or editing a *State String*. See State String in the Technical section below for specifications .

If the state is generated via methods 1 or 2 above, you’ll want to get the *State String* next. It can be found within the running Component, via **File -> Export,** and in the Inspector, under **State.Current**.

### 

### Setup the Component to start with the Solar System

it can be loaded in two different ways:

1. Modify the **Source (enter a url).**  Adding the *State String* as the url parameters:  
   https://….html?StateString
2. Have the Inspector initialize with the **State.Current**=StateString

### Inspector Variables

All the bodies can be modified via the inspector. Note that while the simulation is running (non-Paused), body variables are constantly updating and should not be modified.

|  |  |  |  |
| --- | --- | --- | --- |
| **Variable** | **Type** | **Default** | **Description** |
| Speed.Paused | Boolean | True | True if the simulation is paused, false if it’s running. |
| Speed.Time Speed.Ratio | Number  Enum | 10  Day(s) | Determines how fast the simulation runs, e.g. “5 days” or “3 months” per second.  *Enum:* *Second, Minute, Hour, Day, Month, Year.* |
| Event.#.Type  Event.#.Details | Enum  StringArray | None  [] | Tracks collision events. Each collision will show up as a Type Collision and the Array showing which bodies collided. |
| Body.#.↓ |  |  | *# ranges from 0 to 30.* |
| .Name | String | “” | Name of the body. |
| .Active | Boolean | False | Set to true to activate. Inactive bodies will not be seen nor interact when the simulation is running. |
| .Type | Enum | Star | Body type.  *Enum: Star, Planet, Moon, Asteroid, Undefined* |
| .Material | Enum | Star\_Red | Graphical image/mesh, what the body looks most like.  *Enum: Earth, Jupiter, Mars, Mercury, Moon, Neptune, Saturn, Star\_(Red,White,Yellow,Orange,Blue), Uranus, Venus* |
| .Mass | Number | 1 | Mass in Earths. 1 = 5.972×10^24 kg. |
| .Diameter | Number | 1 | Diameter in Earths. 1 = 12,742 km. |
| .Rotation | Number | 1 | Rotation Speed in Earths. 1 = 1 Rotation per 24 hours. |
| .Position.(X/Y/Z) | Number | 0 | Initial Position in AUs (Earth’s distance to the sun). |
| .Velocity.(X/Y/Z) | Number | 0 | Velocity in AUs. |
| .Velocity.Auto | Boolean | False | If set to True, the velocity will automatically calculate for an initially stable orbit. Collisions or extreme situations may destabilize this orbit later.  *Note:* Once the simulation starts running, this will be set to false and the Velocity X/Y/Z set to the stable orbit. |
| State.Current  State.# | String |  | State of the simulation, including speed and all Body objects. The **Current** state is live (may be delayed while the simulation is running), while the **#** tracks prior states every time the Play button is clicked (starts at 0). |

# Technical

## The State String

The state string is stored as a key=value escaped string, with a '&' separating each key/value pair. This allows it to be added directly as URL parameters to the Component’s URL. While similar to Inspector, several fields have been streamlined to make it more manageable:

|  |  |  |
| --- | --- | --- |
| **Variable** | **Type** | **Description** |
| Speed | Numeric | Determines how fast the simulation run, in seconds per real second. |
| #.↓ |  | *# ranges from 0 to 30.* |
| .Name | String | Name of the body. |
| .Type | String | Body type.  *Values: Star, Planet, Moon, Asteroid, Undefined* |
| .Material | String | Graphical image/mesh, what the body looks most like.  *Values: Earth, Jupiter, Mars, Mercury, Moon, Neptune, Saturn, Star\_(Red,White,Yellow,Orange,Blue), Uranus, Venus* |
| .Mass | Numeric | Mass in Earths. 1 = 5.972×10^24 kg. |
| .Diameter | Numeric | Diameter in Earths. 1 = 12,742 km. |
| .Rotation | Numeric | Rotation Speed in Earths. 1 = 1 Rotation per 24 hours. |
| .Position | X,Y,Z (Numeric) | Initial Position in AU’s. |
| .Velocity | X,Y,Z  -or- “Auto” | Velocity. If set to “Auto”, it will be automatically calculated for a stable orbit. |

**Differences from Inspector Values:**

* Numeric values can store up to double precision.
* These can be formatted as whole #’s, fractional, or scientific notation
  + Scientific Notation: Use e for “x10^” e.g.: 5e-4 = 5x10-4
* Only the active planets are stored. Non-active planet data is not included.
* The State string will not store the previously saved states and events.
* Spaces are replaced with ‘+’.

**Example:**

Speed=864000&0.Name=Far+Star&0.Type=Star&0.Material=Sun&0.Mass=3.33e5&0.Diameter=15&0.Rotation=4.4676&0.Position=0,0.2,0&0.Velocity=Auto

## Physics

This section defines the functionality, which allows the celestial bodies to move and interact with each other within the simulation.

### PhysicsBody.cs

PhysicsBody.cs is part of the chain-based inheritance structure of the project. Each body in the simulation has an associated “Body” object, which through the inheritance chain, inherits the PhysicsBody functionality. The PhysicsBody’s job is to handle all the physics related information needed for the Gravity.cs class and also relays new information Generated by the Gravity.cs class.

**Detailed Functionality:**

**Unit Conversion:** Converts the supplied mass and position values to values that are computable for the Gravity.cs class.

* Position: Converts Astronomical Units to Meters.
* Mass: Converts Earth Mass to Kilograms.

**Stores Velocity Information:** The class stores the velocity of the object, which is computed by the Gravity.cs.

-Getter and setter for velocity.

-Getter and setter for initial velocity control variable.

**Stores Force Information:** The class stores the total force applied to the body as well as a pointer to a body that it has the most attraction to.

-Total Force: This is needed to calculate the how the body moves.

-Most Pull: This is needed to help determine an initial velocity.

### 

### Gravity.cs

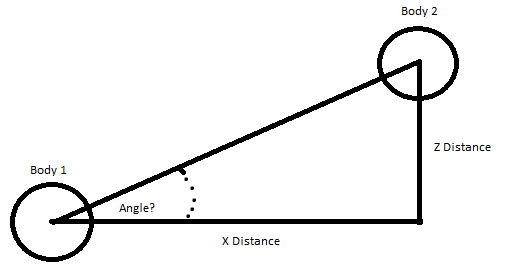
This class is the heart of the simulation. This class is used to compute the force, velocity, and position of every celestial body in the simulation each update cycle, from the information it retrieves from each body’s PhysicsBody object. The computed body updates are then passed back through the PhysicsBody objects, which later updates the Visual System.

**Flow of Physics Functions:**

1. update() is called each frame, which first calls calcPosition().
2. calcPosition() calls updateVelocity().
3. updateVelocity calls updateForce().
4. updateForce() calls calcForce() for each body to body interaction.
5. updateForce() then sums the force for each body and stores it in the body’s PhysicsBody object.
6. updateForce() completes and returns to updateVelocity().
7. updateVelocity() then calls calcInitialVelocity().
8. Initial velocity is calculated for each body if necessary and stores as velocity in the PhysicsBody object for each body.
9. calcInitialVelocity() then returns back to updateVelocity(), which utilizes the total force of each body to calculate an acceleration for each body.
10. The acceleration is then used to compute the change in velocity using kinematic equations and the new velocity for each body is updated.
11. updateVelocity() then returns to calcPosition(), which uses the calculated velocity and the change in time based on the update() function and speed of the simulation, to calculate a new position for each body.
12. Process repeats.

### How New Positions are Calculated for a Body

1. The first step is to calculate the total net force that is applied to the body.
   1. If we want the force applied on “Body 1” by “Body 2” we will use the gravitational force equation, which is: force = (g \*m1\*m2)/ r^2.
      1. m1 and m2 are the masses of the bodies, which are known before the gravity class takes over.
      2. g is a constant value of: 6.67408 × 10-11 m3 kg-1 s-2
      3. r is the radius or center to center distance between the object. Radius is also known when entering the gravity class. Since the distance is a vector, the magnitude of the vector is the radius.
   2. Some bodies might pull straight on another body and some may pull on another body’s side, which means the force applied to the body is a vector, so we now need to determine the angle of the force applied.
      1. One keynote to keep in mind, solar systems are formed spinning at extremely fast speeds, which tends to flatten them out and is the reason why you don’t see planets orbiting the sun in all different directions, so even though we are dealing with 3-dimensional space, we can calculate the physics in 2-dimensions. Our simulation places the solar systems in the (x,z) plane, so that will be how the physics are calculated. \*\*\*Recently added was the ability to calculate physics in the (y,x) plane, which also allows for a mixed orbits and tilts, but the tilted orbits may not be as accurate as a flat plane \*\*\*
      2. From my previous body example above, we need to determine the angle of which “Body 2” is pulling on “Body 1.” Since we know the x distance and z distance we can easily determine this angle by using atan2(z/x). We use atan2 instead of atan because atan does not support all quadrants, which we need for 360 degrees of angles.



* + 1. Now That we have computed our angle, we know that for force vectors, Fx = force\*sin(angle) and Fy = force \* cos(angle), and in our case Fy is Fz.
    2. Now we have our directional force.
    3. \*\*\*Newly added\*\*\* the equations have also been manipulated to switch to the (y,x) plane
  1. Now we repeat the gravitational force equation for all bodies with “Body 1” i.e. “Body 1” by “Body 2,” “Body 1” by “Body 3,” …… “Body 1” by “Body n” and we sum all those forces.

1. Now that the net force for the bodies we can move on to updating the velocity.
   1. Velocity is a constant meters per second, unless acceleration changes, so we first need to determine the acceleration.
      1. Since we know the force and the mass of the body we can use the equation: force = mass \* acceleration and with a little manipulation we get: acceleration = force/ mass.
      2. Now that we have the acceleration, we can use the kinematic equations to find our new velocity: velocity final = velocity initial + acceleration \* change in time.
         1. At this point in the simulation we will already know the previous velocity (or velocity initial), acceleration, and change in time.
      3. We complete this for both x, y, and z positions.
2. Now that we have our new velocity we can easily calculate the change in position.
   1. We know velocity is meters/ seconds, so if we just multiply by the change in time, in seconds, then seconds will cancel out and we will have a change in meters, we do this for both x, y, and z velocity.
   2. We can take this change meters and add it to our position vector and we have successfully updated the position of our body!

### How is an Initial Velocity Calculated

1. Without an initial velocity, bodies would just pull straight into each other, when we would rather see orbits form. I like to think of it as the game “tetherball.” I you just let go of the ball it will just fall straight into the pole, but if you throw it hard enough left or right it will swing all the way around.
2. Now initial velocity for our purposes does not mean the speed at which the body first started going to form an orbit, but rather an initial value of velocity for the body when the simulation first begins, based on orbital speed calculations, which are derived from Kepler’s third law.
   1. The equation used to calculate this initial velocity is: v = sqrt((g\*LargeMass)/ radius).
3. The equation is pretty straight forward, except that we need to know what body we will be orbiting to accomplish this.
   1. In our simulation, for each body, we determine which body we will most likely be orbiting by comparing all of the forces applied the specific body and determining which body has the greatest pull on that specific body.
4. Knowing which body has the strongest pull, we can apply the formula and generate an initial velocity for that body.
   1. Each body also has a control variable that determines if initial velocity has been set or not, which allows the calculation to only happen once or allows the user to determine their own velocity.

## Non-Implemented Features

**Permission System:** While Perms exist, no setup has been designed to disable/enable various features based on the permission.

**UI:**

* **Limited Planet Zoom:** Can hold “Shift” to enlarge further away planets (visual effect only, does not affect collisions).
* **“Wide Scope”:** No button to easy zoom out to see the entire system at once.
* **Presets:** Basic Sol System and Collision created. Further presets would be useful.

**Graphical Limits/Issues:**

* **No Planet Trails:** No trail system exists for planets. Small planets will flicker/glow slightly, though far enough and they won’t be visible.
* **Enlarge planet flicker:** Larger, Far away planets can flicker.

## Technical Limitations

### Inspector Value Precision

The Inspector can only store single-precision values, where the physics are calculated in double precision. When values are updated by the Simulation, the rounded value is sent to the Inspector. When updated via the Inspector, the values are limited to single-precision details.

### Running Update Rate

Due to the transfer-rate limit of Capi: while the simulation is running, changes to the body’s position and velocity do not show in the Inspector in real time. These will update to the new values every second.

### Send/Receive Race In Running Simulation

When the simulation is running (not-paused), the position and velocity of each body is constantly updating. Changing either at this time could lead to unexpected consequences. In the case of changes while running, the simulation should be paused first, values updated, and then resumed.

# Unity Merging Process

These processes were used to develop the initial version of this software.

### Git Branch Set-Up

1. Checkout Develop
2. Pull Develop
3. Create New Branch with Naming Convention *“US\_Number”*
4. Create Personal Scene in Unity with Naming Convention *“US\_Number\_Name”*
5. Push US Branch to make Accessible on Github

*\*\*\*Any Other People Working on Branch. Pull Branch and Complete Steps 4 and 5.\*\*\**

### Initial Merging

1. Create Scene with Naming Convention *“US\_Number”*
2. Preform Object Merging *(Grab hierarchy element, copy and paste into “US\_Number”)*
3. Check to Make Sure Object Links are Not Broken.
4. Delete All Personal Scenes *(Once everything merge properly)*

### Final Merging

1. Checkout Develop Branch
2. Pull All Other US Branches into Develop
3. Move US Scenes to *“Assets->Editor->Scenes”*
4. Preform Object Merge from each US Scene to Main Scene

### Testing

*\*\*\*Each Member Must Pull Develop and Test Before Demo is Provided\*\*\**

### Performing Object Merge

1. Open Base Scene to Merge Objects into *(Scene 1)*
2. Drap Scene to Hierarchy to Take the Objects from *(Scene 2)*
3. Take Items from Scene 2 and Drag into Scene 1
4. Check All the Links on the Objects and Relink if Needed