Gravity Engine Unity Asset

Version 1.2

Overview

Gravity Engine (GE) provides a powerful, accurate physics engine to simulate gravity including:

- Creation of orbits with control of shape, inclination and orientation.
- Binary star generation
- Gravitational simulation of particle systems
- Gravitational collisions with particle ejection effects
- Simple spaceship example that allows accurate movement under gravity
- A library of over 50 three body solutions with special purpose algorithms to allow simulation through near-misses
- Centralized control of overall evolution speed, distance and mass scales
- Accurate, energy conserving physics evolution

It is not necessary to understand the details of the gravitational physics. Gravity can be added to a scene without scripting.

Tutorial and Demo Videos:
https://www.youtube.com/channel/UCxH9ldb8ULCO_B7_hZwIPvw

Support Forum: http://nbodyphysics.com/blog/forums/

Script Documentation: http://nbodyphysics.com/gravityengine/html/

Support: nbodyphysics@gmail.com

Table of Contents

Package Overview

Fundamental Classes

There are three key classes involved in adding gravitational evolution of objects to your Unity project:

1. GravityEngine – the script that performs all the physics calculations and updates object positions in the scene.
2. NBody – a script attached to all bodies to be controlled by the GravityEngine. Used to specify their mass and initial velocity.
3. GravityParticles – added to a particle system to allow it to be evolved by the GravityEngine

The GravityEngine is a very flexible physics engine that provides:

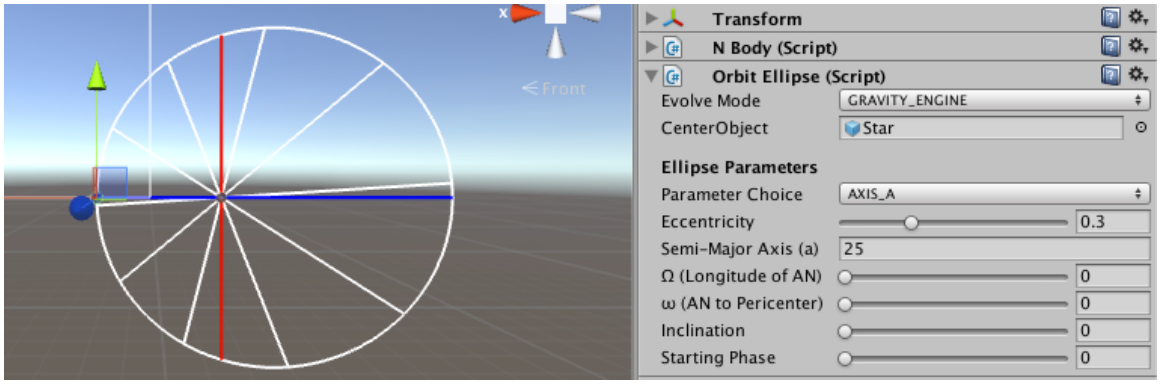
- Options to scale overall mass and timescale to control the speed of evolution

- Advance options to select:
    - Physics algorithm
    - Time steps per frame for algorithms
    - Performance optimizations for gravitational particles
    - Options to allow objects to be detected automatically or added via scripts

NBody Initial Velocity Scripts

Finding the exact initial velocity that creates a specific orbit can be challenging, requiring either physics calculations or trial and error. To eliminate the need for this there are a number of scripts that can be used in addition to the NBody script to control the initial conditions of a body to create a specific orbit.

These scripts will show the path of the orbit in the scene view as the objects are added.

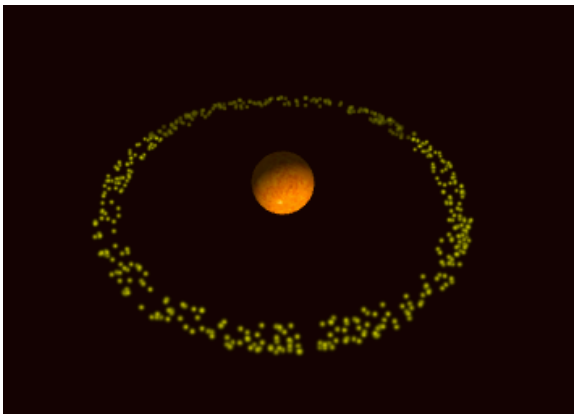


There are several options for adding initial conditions or constraining the motion of an NBody object:

- OrbitEllipse – start a body in a specified elliptical orbit around another body. The body can be moved using gravitational force, or it can move in a fixed path using Kepler's Equation.
- Binary Pair – initialize a pair of NBody objects so that they form a binary pair with specified orbits. Based on interactions with other bodies the actual orbit followed may change.
- FixedObject – object does not move, however it's mass will influence other objects in the scene

Gravitational Particles

GravityEngine allows particle systems to move based on the gravitational field. Particles are treated as massless bodies and are subject to the gravity of NBody objects in the scene. An existing particle system can be placed under the control of the GravityEngine by adding a GravityParticles script.



The GravityParticles script can also have an optional delegate that controls the initialization of the particles. Several delegates are provided:

- DustBall – a spherical shell of particles moving at a specified initial velocity
- DustBox – a 3D rectangle of dust with individual control over size in (X,Y,Z)
- DustRing – a ring of particles that orbit around a massive NBody parent. The orbit parameters can be controlled. Useful for effects like planetary rings.

Collisions

Collisions between NBody objects are detected using Unity colliders. Since these collisions are due to gravitational interactions, the NBodyCollision class handles updates to the physics. The collision can be handled in one of three ways:

1. Absorb
2. Bounce
3. Explode

In each case the physics engine is updated as required and momentum and energy are conserved.

Tutorials

Tutorial videos highlight the key elements of the package. These are available at <link>. For a quick reminder (or for those who prefer written steps) each tutorial is summarized here.

Tutorial 0: Quick Start

1. Begin with an empty scene
2. Create a GravityEngine (GameObject/3D Object/GravityEngine)
3. Create a Star:
    a. Create an empty game object and name it “Star”
    b. Add an NBody component (Add Component/Scripts/NBody). Set the mass to 1000

        i. This script ensures the gravity engine controls this object. It provides the values for mass and initial velocity.
        ii. The Transform component now changes to show only the position. Orbits in the gravity engine asset use hierarchy and this relies on a scale of 1 in the transform element.
4. Create a model for the star (Game Objects/3D Object/Star)
    a. Make this a child of the Star with local position (0,0,0)
        i. Making the model a child allows the size of the Star to be adjusted without adversely impacting the scaling of other children.
    b. Add the Solar Texture as a Material
    c. Adjust the size to (5,5,5)
5. Create a planet (Repeat 3 and 4 naming the object "Planet").
    a. Give the planet a mass of 5
    b. Add a material e.g. Ganymede from PATH_HERE
6. Add an OrbitEllipse component to the Planet.
    a. This script defines orbital motion around a specified body. It will automatically select a parent object as this body.
7. Make the planet a child of the Star.
8. Look at OrbitEllipse in the Inspector and scene view
    a. Set the "Semi-Major Axis" to 10
    b. Experiment with the orbit attributes and observe how they change in the scene view.
9. Press play to see the planet orbit the sun
    a. Can adjust the camera settings to see the orbit (z=-30)
    b. Change camera settings to a background with a solid color if you wish.
10. Create an empty game object "DustRing"
11. Add a particle system
    a. Duration=999
    b. Start lifetime=999
    c. Start Speed=0
    d. Emission Rate=0
    e. Emit as a burst of 1000 at time=0
12. Add a GravityParticles script to DustRing object.
    a. This script indicates the Gravity Engine will control the particles.
13. Add a DustRing script to DustRing object. This controls the initial position and velocity of the particles.
    a. Add a material (e.g. BlueMat at PATH)
    b. Adjust the orbit to the desired size and shape.
14. Make the DustRing object a child of the star.
15. Press play to see the dust ring and planet.

Tutorial 1: Binary Stars and Some Gravity Engine Tips

1. Create an GravityEngine (GameObject/3D Object/GravityEngine)
2. Move two copies of the Star prefab into the scene
    a. Place them at X=-10 and X =10

3. Press PLAY and observe
    a. The gravity engine moves the objects and they go through each other and fly apart. This is due to a numerical breakdown when they get very close and the forces become large. In most cases a collision handler would intervene.
    b. There is an alternative algorithm that is less vulnerable to this. It can be found under the GravityEngine/Advanced section in the inspector. Select HERMITE8 and see what happens. (Hermite adapts the time step and does a better job with large forces when objects get close. It used more CPU than LEAPFROG but does a better job of conserving energy).
4. Create an orbiting pair by setting the Y velocity of the NBody component of one star to +4 and the other to -4.
    a. PLAY now shows these objects in orbit around each other as a binary pair.
    b. Getting a specific orbit by setting the velocity is challenging
5. Create an empty game object "BinaryPair"
    a. Attach the Script "BinaryPair"
    b. Make each Star a child of the binary pair
    c. Set the size (semi-major axis) of the ellipse to 20. Now the inspector shows BOTH orbits and they can be adjusted.
    d. Make one of the stars a different mass – notice how the orbits are adjusted.
6. The game speed of the binary star evolution can be controlled in several ways:
    a. The mass of each star could be scaled. Larger mass values result in larger forces and faster motion.
    b. The mass scale of the entire scene is controlled by the Gravity Engine. In the Scaling section of the GravityEngine the overall mass scale can be adjusted with the 'mass scale' parameter. Increasing the mass scale will make everything move faster. Velocities will also be adjusted so the same motion results – just faster.
    c. GravityEngine also allows control of the timescale. This changes the motion speed but by doing more physics calculations each frame.

Tutorial 2: Orbits

This tutorial demonstrates how to setup bodies in orbits around a "star". "Star" refers to a body that has larger mass than any of the bodies orbiting it (e.g. a factor of 50 or more.)

1. Begin with an empty scene.
2. Create an GravityEngine (GameObject/3D Object/GravityEngine)
3. Add an Nbody prefab to the scene.
    a. Set the mass to 1000
    b. Set the scale of the NBodyModel child to (5,5,5).

    c. Rename the parent to “Star”
4. Add a PlanetInOrbit prefab to the scene and then drag it onto the “Star”.
5. Select the planet.
    a. The Editor panel contains a component “Elliptical Start”. The parameters here control the shape of the orbit.
6. Set the camera Z position to zoom out to see both the star and planet.
7. Press PLAY.
    a. The planet will move in a circle at a fixed radius
8. Press stop.
9. Adjust the orbit settings and get a feel for how the affect the orbit. As you change the parameters the orbit path in the Scene window will be updated.

Some Comments:

The planet motion is evolved using the gravitational force calculated as the scene evolves. The predicted orbit in the editor window assumes the planet and the star are the only objects interacting. If you have other planets and stars the resulting orbit will not be exactly as show. (Predicting the exact orbit when even three bodies are present is not solvable in general and is know as “The Three Body Problem”).

The planets gravity will move the star by a small amount. In general this will cause the star to drift in the scene. There are several solutions.

1. A FixedObject component (in Scripts/Orbits) can be added to the star. The GravityEngine will detect this and the position of the star will not be updated.
2. If you have only two bodies, instead of using a Planet with an OrbitEllipse component, create a binary system (see the Binary Tutorial below).
3. The star can be given a velocity to offset the drift. (Aside: is it worth doing this automatically? Can we for multiple planets? Could have NBE report CM position and CM velocity…)
4. The camera can be made to follow the center of mass of the system. (See X)

Compound systems can be created. For example you can add the third PlanetInOrbit and make it a child of the existing planet – as a moon. Generally the moon should have a smaller (10x) mass than the planet.

The Planet prefab makes the planet a full participant in the gravitational evolution. As more planets are added the computation requires the force between every NBody object in the scene to be evaluated. If the planets are not expected to influence each other then it can be better to move the planets in a fixed path due to the star’s gravity, ignoring the gravity from other planets. This can be done by selecting the KEPLERS_EQN mode for the OrbitEllipse component.

Tutorial 3: Collisions

1. In an empty scene, Create an GravityEngine (GameObject/3D Object/GravityEngine)

2. Create 2 NBody objects by dragging the Prefab "Star" into the scene twice. Change the names to NBody1 and NBody2
    a. Move bodies to X=-10 and X=10 respectivly.
3. To setup bodies for collisions we need to attach some components to the models of each NBody. i.e the child StarModel that has the MeshRenderer
    a. Confirm a Sphere Collider is present (it's part of the prefab)
    b. Ensure isTrigger is enabled for the collider
    c. Add a RigidBody (Component/Physics/RigidBody)
    d. Set isKinematic enabled, rigidbody mass to zero, Use gravity disabled.

The models attached to the NBodies will now report trigger events when they touch each other. The action that occurs is under script control via the usual collision handling routine OnTriggerEnter(), OnTriggerStay() and OnTriggerExit().

Space Physics provides a NBodyCollision script that provides several options for a collision:

- Bounce
- Explode: Remove the body and start a particle explosion (e.g. small body hits a larger one and is destroyed)
- Absorb: remove the body from physics evolution

Looking at these each in turn:

Try BOUNCE. When PLAY is pressed the objects move together and then bounce away from each other.

<u>Absorb Immediate</u> is a good choice when two small bodies collide and one is absorbed at the moment of contact.

1. Add a NBodyCollision to the model attached to NBody2. This is the body that will be absorbed.
    a. It MUST be attached to the model, since that's where the Rigidbody/Sphere Collider are attached!
2. In the Inspector for NbodyCollision set the type to ABSORB_IMMEDIATE
3. Press PLAY
    a. The smaller body will move towards the larger and at the moment of contact be inactivated.
    b. The momentum of the smaller body will be used to adjust the velocity of the larger body so that momentum is conserved.
    c. The mass of the remaining body will be increased by the mass of the colliding body

Now try ABSORB_ENVELOP.
The body being absorbed must first enter completely inside the other body before being. With bodies the same size they often fly apart before the envelop can be recognized.

1. Press PLAY to see this happen.

2. Change the size of the StarModel being absorbed to be 2 and PLAY again. Now the body is absorbed.

To absorb the body and initiate an explosion of debris, select EXPLODE.

1. Drag the ExplosionSmall prefab onto the Inspector entry for the NBodyCollision Explosion Prefab field. (Small refers to the size of the particles)
2. Select Customize Explosion.
3. Adjust explosion parameters
    a. Cone angle is the ejection cone for the particles generated
    b. Explosion size is the target for the explosion size. However if the body is heavy and/or has a small radius the initial velocity may need to be adjusted down using "Velocity Adjust"
    c. Velocity adjust modifies the particle velocity calculated to reach the explode size of the body being impacted. Try 0.7
    d. Velocity spread defines the variation in the velocity. The value provided is the standard deviation. (A value of 1 will give a velocity spread of about +/- 30%)
4. Press PLAY and observe the explosion.

Tutorial 4: Particles

1. In an empty game scene add a GravityEngine.
2. Drag the Star prefab into the scene.
3. Using the Object menu add a Unity particle system and position it at X=+15.
4. Attach a GravityParticles script to the particle system and press PLAY.
    a. The particles are detected by the gravity engine and evolved. They fall towards the star.
    b. The usual particle system controls can be used to control the shape, rate and velocity of the particles.

There are cases where it useful to have script control over the initial positions and velocities of the particles. One example is the DustRing script that enures all the particles are in orbit around a central body. A simpler illustrative example is DustBall.

1. Open the DustBall script in an editor.
    a. DustBall implements the IGravityParticlesInit interface. This is described in the script documentation.
2. Add a DustBall to the particle system in the scene.
3. Change the particle system to emit long-lived particles in a one-time burst.
    a. Change lifetime to 999
    b. Set Looping to OFF
    c. Change start lifetime to 999
    d. Set initial velocity to 0

    e. Under Emission, set the rate to zero and configure a one time burst of 300 particles at time 0.
4. Press PLAY.
    a. The ball of particles falls into the Star
    b. An initial velocity can be added to the ball via the DustBall inspector.

There are other init scripts for particles provided:

- DustBox creates a 3D box of particles
- DustRing (As described in the Overview) creates a ring of particles in orbit around a central object.

Demonstration 1: Spaceship

In the Demos folder is a scene "Spaceship Demo". There is a video tutorial walk-through of this scene online.

The demo introduces three classes that can be useful in developing space games:

1. Spaceship – a script that uses keyboard input to change orientation and apply thrust to the ship to move it in the gravitational field.
2. Grid2D – a script that provides a visual grid reference
3. CameraSpin – allows the player to spin the camera around the origin using keyboard input

Demonstration 2- Three Body Solutions

There are many fascinating solutions of the interaction of three bodies of equal mass. These are provided in Gravity Engine via the ThreeBodySolution class. This demonstration showcases these solutions. There is an explanatory video online.

These solutions can also be found in the mobile app ThreeBody/ThreeBody Lite for iOS and Android. This app uses an earlier version of Gravity Engine.

Documentation

Gravitational Objects

GravityEngine

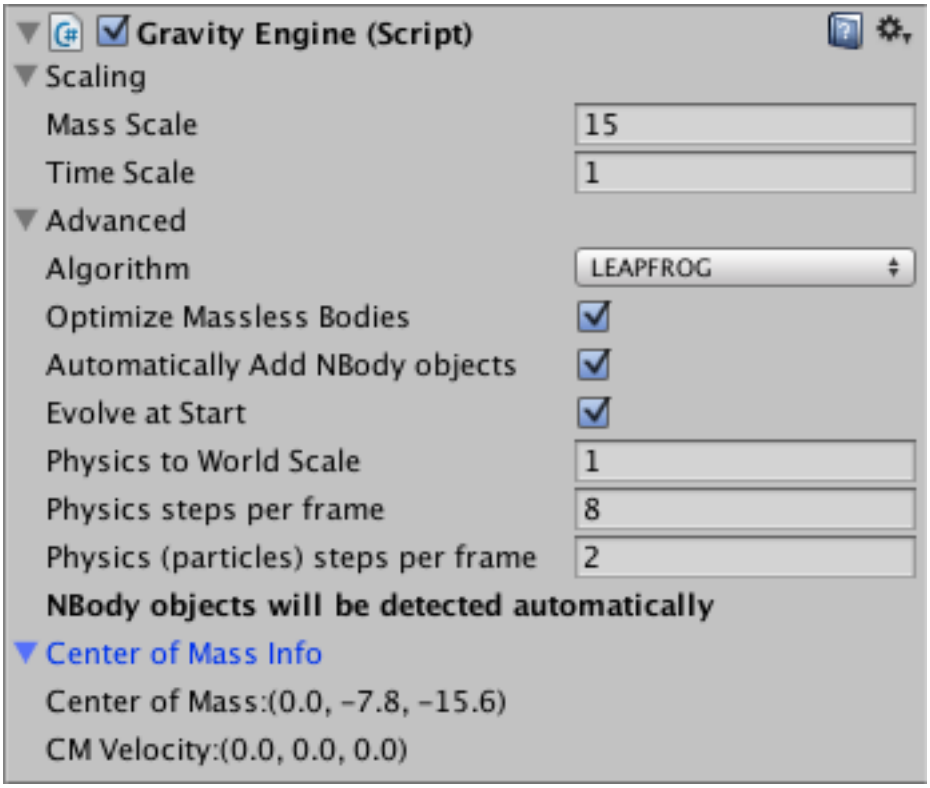
GravityEngine is the primary class in the asset. For gravitational motion the scene must contain an object with a GravityEngine component. This can be added via the Unity Menus (GameObject/3D Object) or by adding the GravityEngine script to an existing object in a scene.

GravityEngine evolves objects that have an NBody script attached. NBody defines the mass and initial velocity. The GravityEngine can automatically detect all NBody objects in a scene when it starts (the default), be given an explicit list of objects, or objects can be added by script calls to GravityEngine.AddBody().

The engine operates on the FixedUpdate() cycle to ensure that physics evolution tracks game time. In some cases the exact amount of evolution in a specific cycle will vary slightly depending on whether the evolution algorithm has run ahead or behind on previous cycles.

In general the GravityEngine attributes cannot be changed while the scene is running (with the exception of the timescale attribute).

The inspector view shows the key parameters:



Mass Scale

The GravityEngine allows for the changing of mass scale of all NBody objects in the scene with a single parameter. Initial velocities of NBody objects will also be scaled such that paths taken by the bodies will not change when mass scale is altered.

Adjusting mass scale can be a useful way to change the speed at which things happen in the scene. As the mass scale increases, forces between the bodies increase and the evolution in the scene happens faster. This is the most common technique for adjusting the overall speed of the evolution in the scene.

Mass scale adjustments do not affect the amount of CPU time used in gravitational evolution.

To delve further, see the section Scaling below.

TimeScale

The timescale controls the ratio of time in "the physics world" as compared to time in the game. It can be used to control the overall speed but this happens in a way that affects the CPU time used for gravitational evolution. A timescale value larger than 1 performs more calculations per cycle. A timescale value of less than one will result in fewer calculations per cycle. In either case the evolution calculations will have the same accuracy.

Algorithm

There are a number of possible algorithms for gravitational evolution. They vary in accuracy and computational complexity.

1. Leapfrog: The best default choice. This is a computationally cheap algorithm with a fixed time step that conserves energy well.
2. Hermite: A variable time-step algorithm that will spend more CPU cycles when objects get close to improve accuracy.
3. AZTTriple: A special case algorithm for ONLY three bodies. It is a "regularizing integrator" that allows objects to get infinitely close without numerical instability. Typically used in conjunction with the ThreeBodySolution class.

Optimize Massless Bodies

NBody objects with a mass of zero (massless) are influenced by other bodies' mass but do not affect others. The GravityEngine detects this and massless bodies are evolved separately using a Leapfrog algorithm. This reduces the CPU cost of evolving massless bodies since the interactions between them are skipped.

If there is a need to use the Hermite algorithm for massless bodies then this toggle can be cleared and no massless optimization will occur.

Automatically Add Nbodies

When selected the GravityEngine will examine all objects in the scene (during the Awake phase) and take control of the evolution of those bodies that have an NBody or GravityParticles component.

In cases where this behavior is not desired (e.g. a player spaceship is not moved while the rest of a solar system is) this option can be unselected. In this case the inspector will show a new section to list the bodies that should initially controlled. Existing scene objects with an NBody component can be dragged onto the Bodies list.

In both cases additional bodies can be added via calling GravityEngine.AddBody() and particles systems via GravityEngine.RegisterParticles()

Evolve At Start

By default the GravityEngine will begin evolving all the NBody/GravityParticles objects in the scene when the system starts. Clearing this checkbox can disable this. When cleared, evolution will start when GravityEngine.SetEvolve (true) is called.

Physics to World Scale

A scale factor used to adjust the distance scale used in the physics engine to the Unity positions in the scene. Physics positions are determined by dividing their Unity positions by this factor.

For example, if two bodies are 10 units apart and the physics to world scale is 1 the GravityEngine will use 10 as the distance between bodies. In some initial conditions from the scientific literature the distance scale defined in may be in a very small range, typically (-1..1). It is easier to use these values in the physics calculations because it avoids the need to scale masses and velocities. In this case, setting the physics to world scale to 10 and then positioning the bodies 10 units apart will result in them being scaled down by 10 in the physics calculations and the physics algorithm will see them as 1 unit apart.

Physics Steps Per Frame

This parameter defines the approximate number of times per frame the physics engine will re-calculate positions for NBody objects. A larger number will result in more accurate physics at the cost of more CPU effort. Default value is 8.

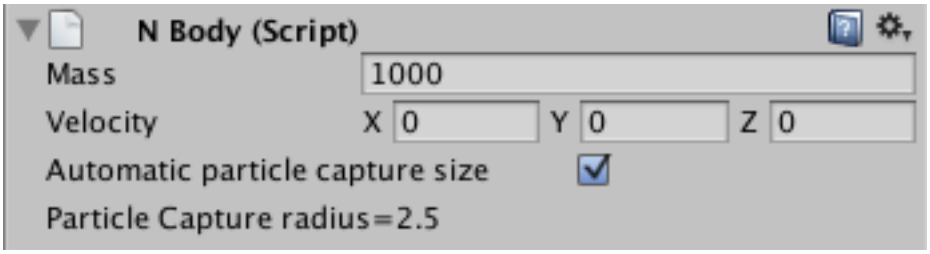
Physics (Particles) Steps Per Frame

This parameter defines the approximate number of times per frame the physics engine will re-calculate positions for particles. A larger number will result in more accurate physics at the cost of more CPU effort. Default value is 2.

Nbody

The GravityEngine evolves objects with an NBody script attached to them.

The inspector view for a general NBody object is:



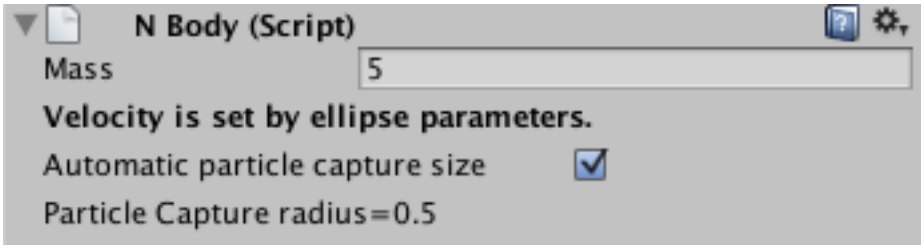
Mass

Mass is specified in dimensionless units. A value of less than zero will be set to zero.

Velocity

A Vector3D specifying the initial velocity of the Nbody. If the initial conditions are controlled by an additional component this field may not appear.

The inspector view for an object that has another component providing initial conditions for motions (i.e. an OrbitEllipse or BinaryPair component) will look like:



In this case the initial velocity is determined by another component.

Automatic Particle Capture Size/Particle Capture Radius

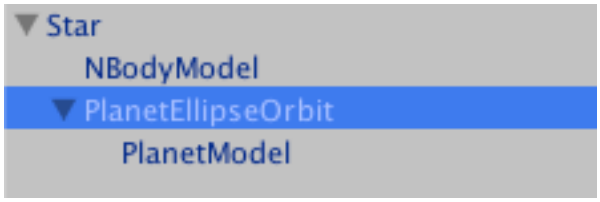
If automatic capture size is selected, the particle capture size will be taken from the scale of a child object containing a MeshFilter object. This corresponds to typical use of NBody objects in which a child is a 3D Sphere. This ensures the particle capture size matches the sphere.

Any particle that enters the capture radius of an NBody will be inactivated.

If the automatic checkbox is cleared, a value for capture radius is required.

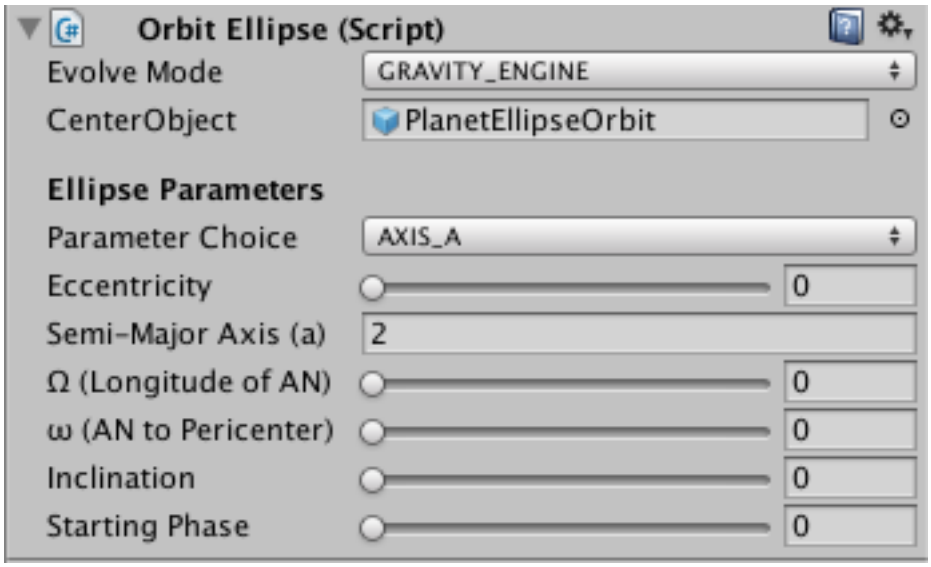
OrbitEllipse

An Orbit around another object can be initialized with the OrbitEllipse component. This script must be attached to a game object containing a NBody component. Typically, the object around which this NBody object is to orbit is the parent. For example:



Selecting an OrbitEllipse object will show the ellipse in the scene view. Adjusting these parameters and seeing the result is the best way to gain insight into their role. In cases where the inclination is zero, Ω and ω will do the same thing.

The inspector view of this component is:



Evolve Mode

There are two modes in which OrbitEllipse operates:

> GRAVITY_ENGINE mode will initialize the velocity of the NBody to launch it on the specified orbit. Forces from other bodies may affect the NBody and affect its path. This is the mode that is used when correct physics is important.
>
> KEPLERS_EQN mode will force the planet to follow the specified orbital path. The mass of the NBody will affect other bodies but they will not affect it. This mode is used when deterministic motion is required (at the cost of physical accuracy). It can also be useful in large orbital systems – since Kepler's equation uses less CPU than calculating all the forces between bodies in a large orbital system.

Center Object

Displays the name of the object around which this body will orbit. This is detected automatically if the object is the child of another NBody object. It can also be set explicitly to any object in the scene with an NBody component.

Parameter Choice

Selects whether the size is specified as semi-major axis (a) or distance of closest approach (p).

Pericenter/Semi-Major Axis/Eccentricity/Ω/i/ω

1. Size: specified by either the semi-major axis (a) or the point of closest approach, the periapse (p).
2. Shape: Controlled by eccentricity in the range 0 to 1. A circle has e=0, an infinitely thin ellipse has e=1. GravityEngine will limit e to a maximum of 0.99
3. Orientation: Celestial mechanics uses a specific choice of Euler angles defined as follows:
    a. Ω (longitude of the ascending node)
    b. Angle of inclination
    c. ω (angle from ascending node to pericenter)

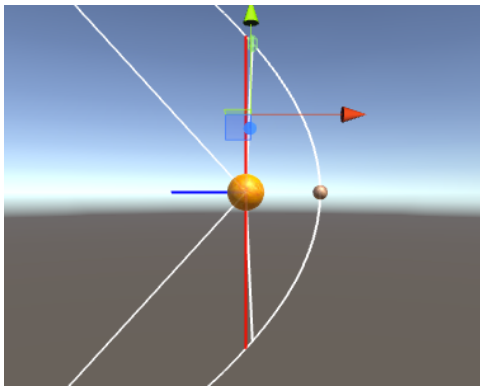
Experimenting with orientation parameters in the scene view and observing the changes in the ellipse is the best way to understand these parameters. (The ascending node is the point where the object moves through the reference (x,y) plane in the +Z direction. The "line of nodes" is the line where the ellipse crosses the reference plane, assuming the inclination is non-zero. See the Wikipedia page on Orbital Elements.)

Starting Phase

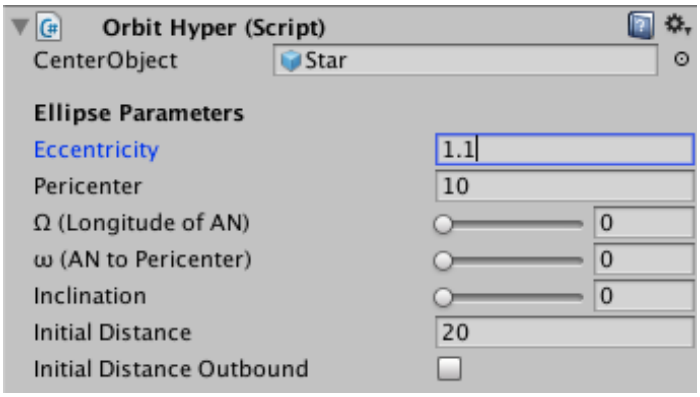
Defines the objects initial position on the ellipse in degress (0..360).

OrbitHyper

Initializes a hyperbola path around a central object. This script must be attached to a game object with an NBody component. When selected the path of the orbit will appear in the scene view.



Hyperbolic orbits by definition have an eccentricity > 1.



Pericenter/Semi-Major Axis/Eccentricity/Ω/i/ω

The same as OrbitEllipse, with the requirement that eccentricity be greater that 1.

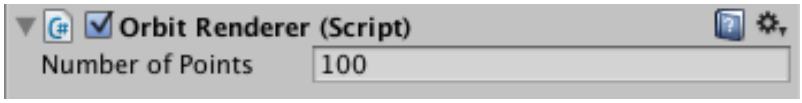
Initial Distance

Sets the initial distance from the center object where the orbit evolution will being. By default this will be on the inbound portion of the hyperbola.

Initial Distance Outbound

When set, indicates that the initial distance should be used to position the body on the outbound leg of the hyperbola.

OrbitRenderer



The OrbitRenderer component works in conjunction with a LineRenderer to show the predicted orbit path in the scene view during game play. The script must be attached to an object with an OrbitEllipse or OrbitHyper component – from which the orbit path will be determined at the time the scene starts.

A body may not exactly follow this path if there are interactions with other masses. The OrbitRenderer is attached to a game object that is a child of the NBody to allow for separate material selection.

To see the orbit, the Line Renderer component must be correctly configured (i.e. add a suitable material)

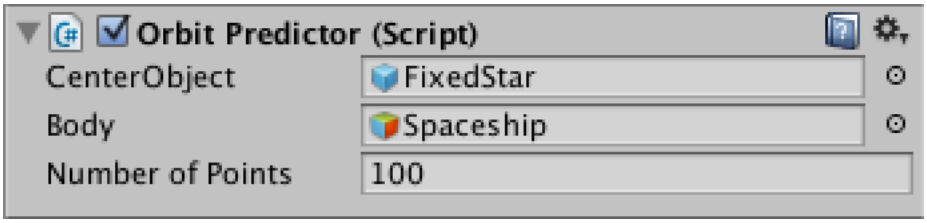
A prefab “OrbitPath” is provided.

OrbitPredictor

The OrbitPredictor script predicts the orbital path a body will take given the current velocity of the body and draws the path using a LineRenderer. The projected orbital

path updates on each FixedUpdate() cycle – so the influence of user provided velocity changes or changes due to interactions with masses other than the central body will be reflected in the updated path.

Orbit prediction is done using a two-body calculation, transforming the position and velocity with respect to the center body into orbital parameters. The predicted orbit will be accurate if the center body is the dominant source of gravity. If there are other significant sources of gravity in the scene the prediction will not be accurate. [Solving the general orbit prediction problem is not mathematically possible in most cases and would require a simulation of the future evolution of the system – which is impractical].

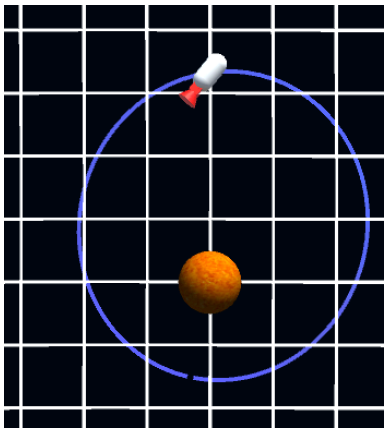


CenterBody: The NBody containing object around which the orbit should be calculated.

Body: The body for which the orbit will be calculated.

Number of Points: The number of points to use in the line renderer to represent the orbit.

Adding an OrbitPredictor component will automatically add a LineRenderer component. To see the orbit, the Line Renderer component must be correctly configured (i.e. add a suitable material)

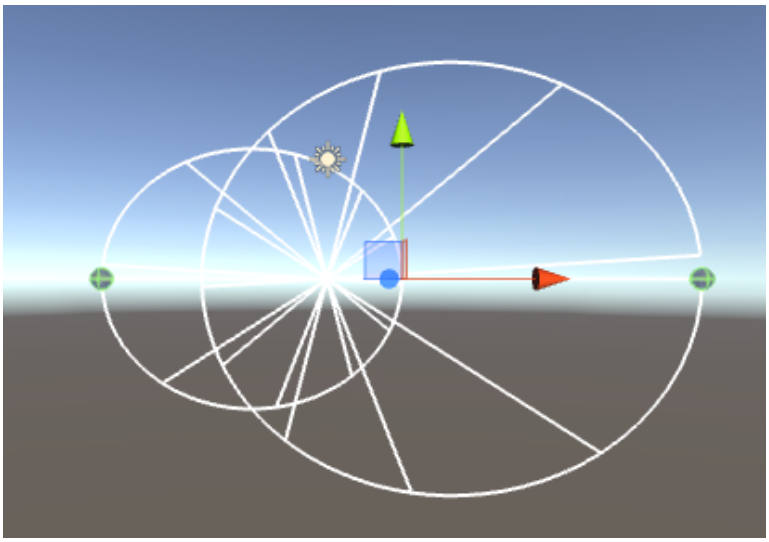


OrbitSimpleDecay

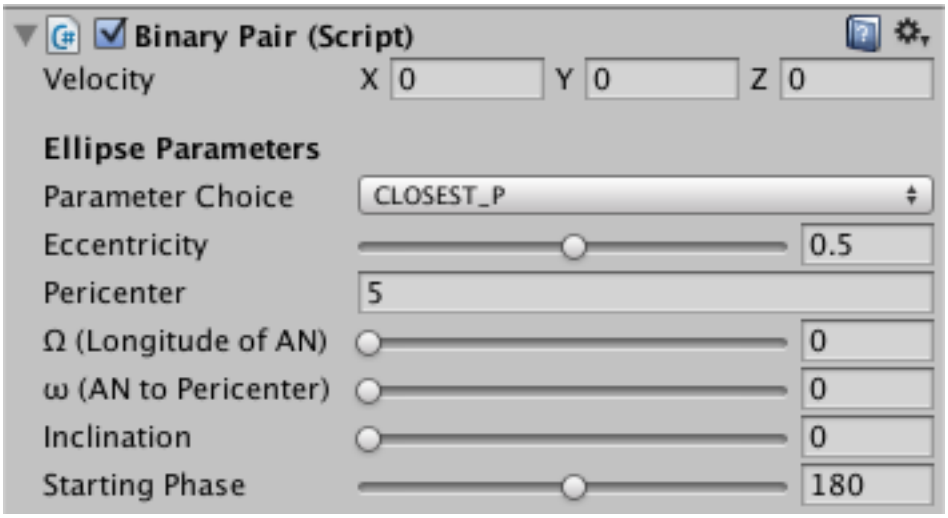
This script provides a simple per-frame reduction in velocity and results in the decay of an object in orbit.

Binary Pair

Binary star systems are very common. Configuring a pair of objects in a specific orbit around the center of mass can require careful calculation of the initial velocities. The BinaryPair component handles this task and provides a scene view of the resulting orbits.



The BinaryPair component is added to a top-level game object and two "stars" are added as children. These stars each have an NBody component that describes their mass. By selecting the BinaryPair object in the scene view the orbits of the objects will be displayed and can be adjusted. GravityEngine provides a binary pair as a prefab object.



Velocity

Specifies the velocity of the center of mass of BinaryPair.

Ellipse Parameters

See the description of fields in OrbitEllipse above.

FixedObject

Adding a FixedObject component to an NBody game object creates an NBody object that has a mass but remains at a fixed position.

Gravitational Particles

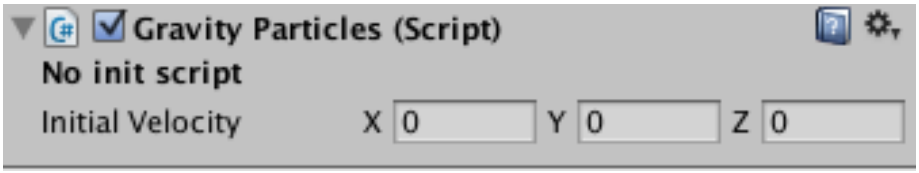
Gravity Particles

Adding a GravityParticles component to a Unity particle system will allow the GravityEngine to move the particles according to the gravity of all objects in the scene. The particles will be treated as massless bodies and evolved using a Leapfrog algorithm.

Particle creation can be done by the Unity particle system component or can be over-ridden by adding a component that implements the IGravityParticlesInit interface (scripts DustBall, DustBox and DustRing demonstrate how this is used).

The particle system must operate in World Space. This particle system attribute will be changed automatically (and a Warning issued) if it is set to local.

Scenes with large numbers of GravityParticles can require more from the CPU. Care must be taken when adding large numbers of particles to ensure game performance is still acceptable. Reducing the steps per frame for Particles in the GravityEngine may also be useful.

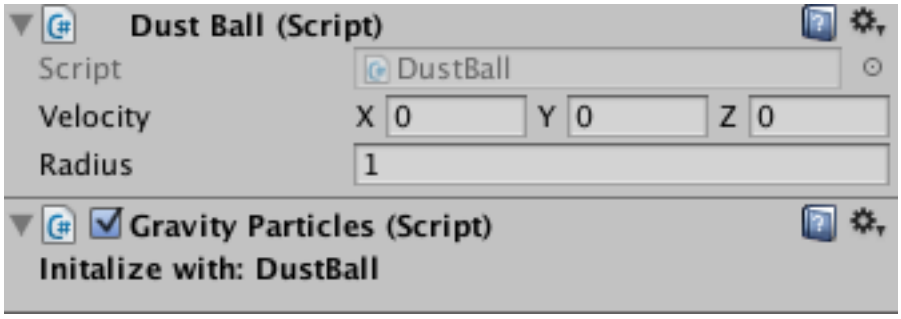


Initial Velocity

The initial velocity to be added to all particles from the particle system on creation. This option only appears if there is no init script attached to the component.

Dust Ball

DustBall implements IGravityParticlesInit and is used to initialize a sphere of particles all moving at the same velocity.



Velocity

Initial velocity of the particles in the dust ball.

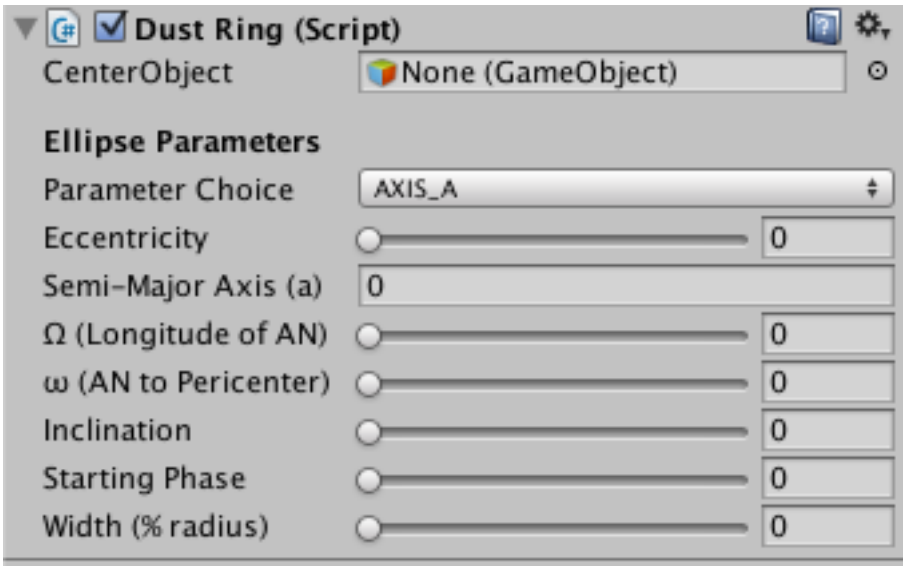
Radius

Radius of the dust ball.

Dust Ring

This script initializes a ring of particles around an NBody object.

The shape and orientation of the ring are controlled in the same way as an OrbitEllipse object.



Ellipse Parameters

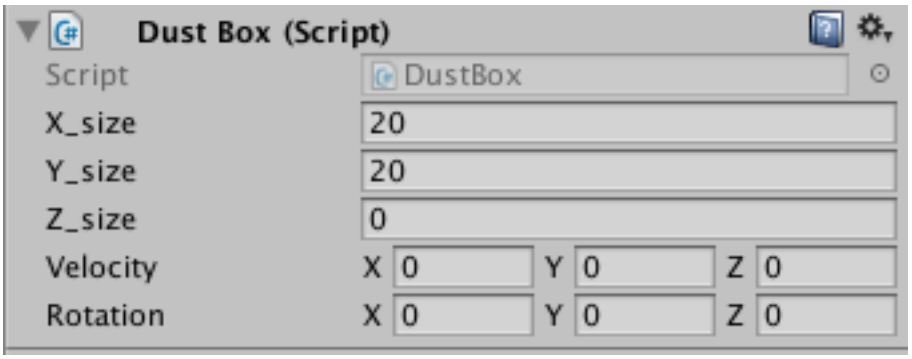
See the description under OrbitEllipse, above.

Width

Defines the width of the particle ring as a percentage of the radius.

Dust Box

Initializes particles in a 3D box.



X_size/Y_size/Z_size

Size of the X, Y and Z axes of the box. Box will be centered at the transform position.

Velocity

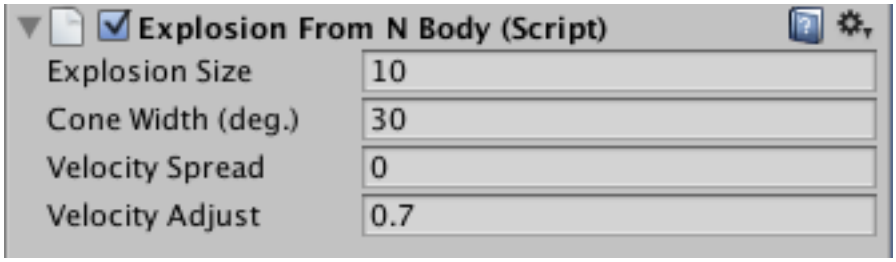
Initial velocity of dust particles

Rotation

3D rotation of the dust box. (Rotation must be controlled here and not in the game object transform to ensure correct rendering by the GravityEngine).

ExplosionFromNBody

ExplosionFromNBody creates a cone of particles that 'explode' away from an NBody object. It determines the required velocity to escape the gravitational field of the NBody such the particles reach a specified distance from the body.



Explosion Size/Velocity Adjust

The maximum distance the explosion particles should travel before falling back to the NBody on which the explosion is occurring.

The size is used to determine the initial velocity of the explosion. In cases where there is a large initial velocity due to a massive NBody or small radius, the first integration step may move the particles too far. This will result in the size being exceeded. The Velocity Adjust field can be used to "tune down" the initial velocity. Typically values in the range of 0.7-0.9 are useful.

Cone Width

Defines the width of the ejection cone in degrees (from center axis to outside).

Velocity Spread

The variation in the velocity applied to the particles. The value used in the standard deviation around the velocity required to obtain the indicated size. A value of 0 will ensure all particles start at the same speed, larger values will give a wider range of velocities resulting in a more spread out value.

This component is usually triggered by a collision between NBody objects.

Collisions

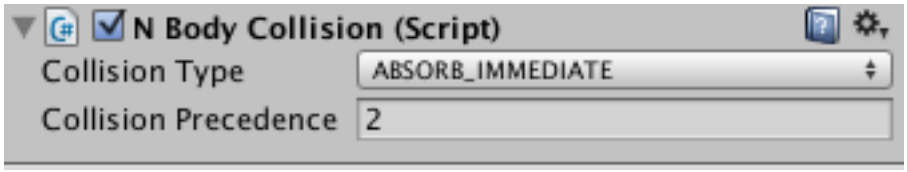
GravityEngine relies on the Unity collider components to detect collisions via trigger events. It does not make use of them to handle the physics of the collisions.

NBodyCollision

NBodyCollision is attached to the CHILD of an NBody object. It is typically attached to a MeshFilter object with a Collider and Rigid Body attached. The Collider/Rigid body configuration must be:

1. Collider **IsTrigger** Enabled
2. Rigid Body **IsKinematic** Enabled
3. Rigid Body **Mass** set to 0 (will appear as a very small number in inspector)
4. RigidBody **UseGravity** Disabled

The NBodyCollision object will handle the trigger events from the collider and interact with the GravityEngine to achieve the desired outcome.



Collision Type

Defines how the collision should be handled. One of:

ABSORB_IMMEDIATE: On trigger detection the object and its NBody parent will be removed from the scene. The parent NBody's momentum will be added to the body that it made contact with.

ABSORB_ENVELOP: Once the body with this component has completely entered the other body (based on sphere sizes) the object and NBody parent will be removed from the scene. The parent NBody's momentum will be added to the body that it made contact with. Envelop works reliably only when the sizes of the objects are different.

EXPLODE: On contact the object and NBody parent will be removed and a particle explosion will be triggered. When selected this element requires an additional parameter: Explosion Prefab

BOUNCE: On contact the parent NBody will reverse direction (as will the parent NBody of the object it made contact with). When selected a Bounce Parameter is displayed. This is value in the range 0..1 with 1 indicating a no energy loss full bounce.

Collision Precedence

In cases where two bodies with NBodyCollision scripts collide, one must take precedence. The body with the lower precedence will be run. If the precedences of the objects are equal, the Unity game object unique id is used to break the tie.

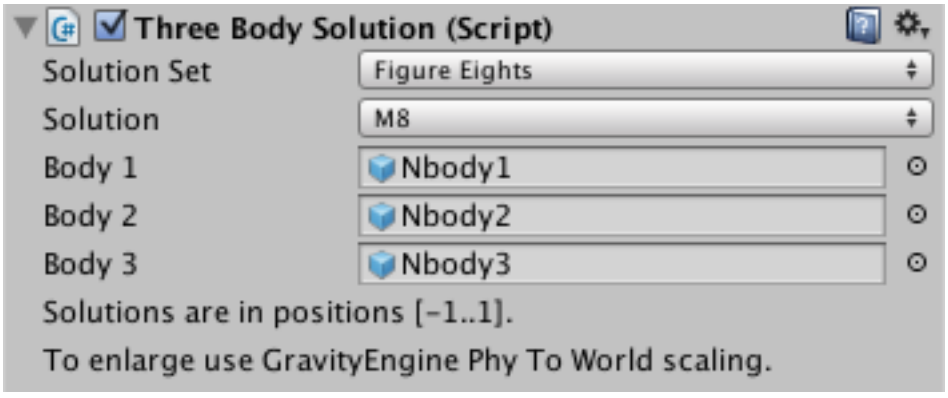
ThreeBody

The gravitational interaction between three equal-mass bodies is a long-standing area of investigation in celestial mechanics. Over the past three hundred years a variety of solutions have been found. The Gravity Engine asset provides a set of these solutions, coupled with a special algorithm to allow them to evolve through regions where the bodies come very close to each other.

These solutions can be explored using the ThreeBodySolution script. They can also be found in the app ThreeBody available on iOS and Android. (The Gravity Engine asset derives from the work done on ThreeBody).

ThreeBodySolution

This component is added to a top-level game object in the scene. (Typically it is added to the GravityEngine game object).



Solution Set

Name of the solution set to select a specific solution from.

Solution

Name of the solution from within the solution

Body1/2/3

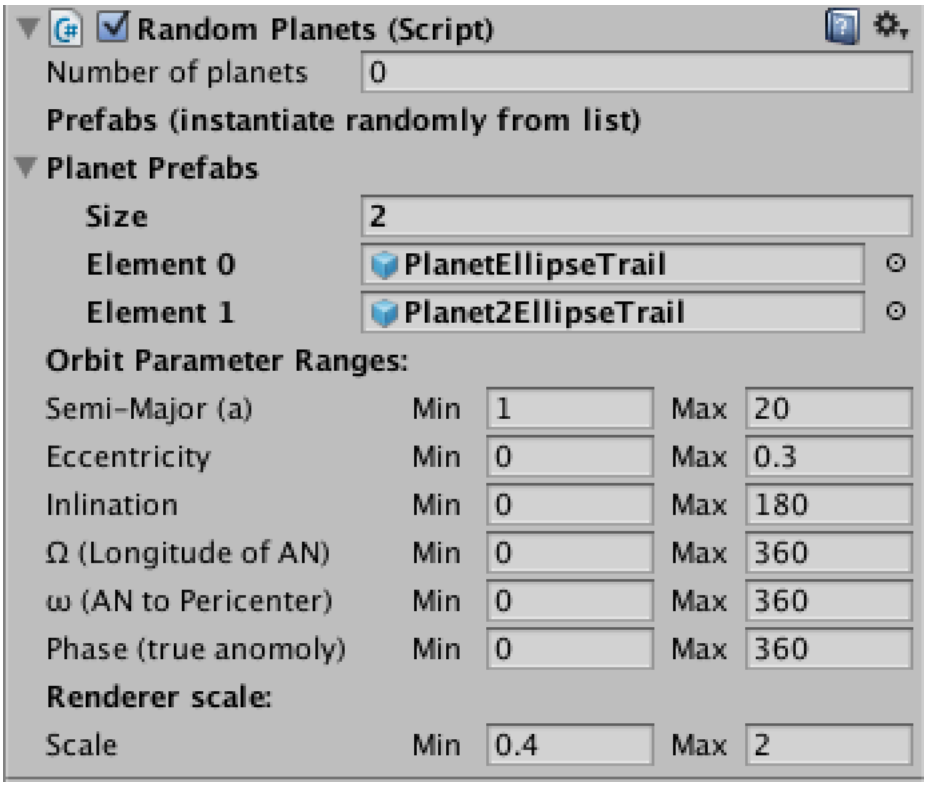
Nbody objects with mass 1 to be used in the scene. Their positions and initial velocities will be initialized based on the solution that has been selected when the scene starts.

System Builders

RandomPlanets

The RandomPlanets script is used to create a system of planets in random orbits around a central body when the scene starts. It allows control of the number of planets and a range for each of the orbital parameters. In order to allow more than one planet game object type it allows a list of planet prefabs to be provided and will pick randomly from the list for each planet creation.

This script is also a useful example demonstrating how to procedurally create and link planets.



Number of Planets: The number of planets to be generated

Planet Prefabs: A list of prefab objects that will be used to create the planets. The prefab planet structure must match the examples in the prefab folder:

1. Base game object with an NBody and OrbitEllipse component

2. A child game object with a MeshRenderer to display to model for the planet.

Orbit Parameter Ranges: Min/Max ranges to be used for each of the orbital parameters of the planets.

Renderer Scale: Min/Max range of the local scale to be applied to the models of the planets as they are created.

Scripting Interface

Scene setup can also be done under script control. Documentation for the script interaction in HTML and can be found in html/index.html.

Scaling

Gravity simulation deals with mass, distance and time. The units/scale for these elements in simulation is never in the standard units of physics (sec, m, kg). In general simulations rescale time, mass and distance to allow the computations to use numbers of "reasonable size". The range of the number depends on the range of values being modeled. For example, in a Solar system simulation it may be useful to work in terms of "Jupiter masses" and set the mass of Jupiter to 1 and the mass of the Sun to 1047.56. (Wikepedia: Jupiter Mass).

The effect of scaling can be understood via Kepler's third law:

$$G(m_1 + m_2)a^3 = T^2$$

The effect a change of the scale of mass and distance is summarized in Table 1:

Change	**Effect**	**Change by 2x**	**Change by 0.5x**
Mass	Orbital speed increases as mass value increases	Evolve 1.4 x faster	Evolve by 0.7x slower
Distance	Orbit speed slows as distance increases	Evolve 0.35 x slower	Evolve 2.8 x faster

Gravity engine sets G=1 (to avoid multiplying by G in every force calculation).