PHYSICS BEHAVIOR SCRIPT INTERFACE

View online: https://www.construct.net/en/make-games/manuals/construct-3/scripting/scripting-reference/behavior-interfaces/physics

---

The `IPhysicsBehaviorInstance` interface derives from IBehaviorInstance to add APIs specific to the Physics behavior.

The Physics behavior also provides a `IPhysicsBehavior` interface deriving from IBehavior, which specifies the global settings affecting the entire Physics world. This interface can be accessed through the `behavior` property of a Physics behavior instance.

Examples

See the Physics scripting example for a demonstration of using physics from JavaScript code.

IPhysicsBehavior APIs

The behavior script interface specifies the properties of the physics world. It is typically accessed through the `behavior` property. Below shows an example of this to change the physics world gravity.

```
const behaviorInst = spriteInst.behaviors.Physics;
const behavior = behaviorInst.behavior;
behavior.worldGravity = 0;
```

---

worldGravity

Set or get the force of gravity affecting all Physics objects. By default this is a force of 10 downwards.

---

steppingMode

Set or get a string of either `"fixed"` or `"variable"` indicating the Physics time stepping mode. Variable mode uses delta-time for framerate independent simulation, but may be non-deterministic due to variance in timer measurements. Fixed mode uses exactly the same time step every frame regardless of the framerate. This is not recommended since modern devices have a range of refresh rates, and it can cause physics to run too fast or too slow depending on the device. However it also makes the physics simulation deterministic (reproducing identical results every time). For more information see the tutorial Delta-time and framerate independence.

velocityIterations

positionIterations

Set or get the number of velocity iterations and position iterations used in the physics engine. The default is 8 and 3 respectively. Lower values run faster but are less accurate, and higher values can reduce performance but provide a more realistic simulation.

---

setCollisionsEnabled(iObjectClassA, iObjectClassB, state)

Set whether collisions are enabled between object types using the Physics behavior. The object types are specified by IObjectClass, and *state* is a boolean indicating whether collisions between these types are enabled. Note this affects all instances of the given object types.

IPhysicsBehaviorInstance APIs

---

isEnabled

A boolean indicating if the behavior is enabled. If disabled, the behavior no longer has any effect on the object, and the corresponding physics body will be destroyed.

---

applyForce(fx, fy, imgPt = 0)

applyForceTowardPosition(f, px, py, imgPt = 0)

applyForceAtAngle(f, a, imgPt = 0)

Apply a force on the object, either with custom X and Y components, towards a position (in layout co-ordinates), or at an angle (in radians). The latter two are just convenience methods that internally calculate the X and Y components. Applying a force causes the object to accelerate in the direction of the force.
Forces can be applied at an image point with the `imgPt` parameter, which normally also causes the object to rotate. Using `0` (the default) for the image point uses the object's center of mass, which does not cause rotation. Use `-1` to use the object's origin, which may be different to the center of mass and cause rotation. A string of the image point name can also be used.

---

applyImpulse(ix, iy, imgPt = 0)

applyImpulseTowardPosition(i, px, py, imgPt = 0)

applyImpulseAtAngle(i, a, imgPt = 0)

Apply an impulse on the object, either with custom X and Y components, towards a position (in layout co-ordinates), or at an angle (in radians). The latter two are just convenience methods that internally calculate the X and Y components. Applying an impulse simulates the object being struck, e.g. hit by a bat.
Impulses can be applied at an image point with the `imgPt` parameter, which normally also causes the object to rotate. Using `0` (the default) for the image point uses the object's

center of mass, which does not cause rotation. Use `-1` to use the object's origin, which may be different to the center of mass and cause rotation.

---

applyTorque(m)

applyTorqueToAngle(m, a)

applyTorqueToPosition(m, px, py)

Apply a torque (rotational acceleration) to the object, either directly, or towards an angle or position. The torque and angle are specified in radians.

---

setVelocity(vx, vy)

Set the object's current velocity, providing a speed in pixels per second for the X and Y axes.

---

getVelocityX()

getVelocityY()

getVelocity()

Get the X or Y components of the object's current velocity, in pixels per second. `getVelocity()` returns both as `[x, y]`.

---

teleport(x, y)

Set the object position preserving the Physics velocity. Normally changing the position of the object will reposition it, but alter the velocity to try to ensure the physics simulation remains realistic even though some external change was made to the object position. Using the `teleport()` method will reposition the object but not alter its Physics velocity in any way, which is sometimes desirable for purposes such as if a Physics object goes through a portal and is meant to appear somewhere else but with the same velocity.

---

angularVelocity

Set or get the angular velocity, in radians per second.

---

isImmovable

isPreventRotation

density

friction

elasticity

linearDamping

angularDamping

isBullet

These are setters and getters for the various properties of the Physics behavior. For more details, refer to the section *Physics properties* in the Physics behavior manual entry.

---

mass

Read-only number representing the mass of the physics object, as calculated by the physics engine. This is the area of the object's collision mask multiplied by its density.

---

getCenterOfMassX()

getCenterOfMassY()

getCenterOfMass()

Get the X and Y position of the center of mass of the physics object, as calculated by the physics engine. This depends on the *collision mask* property, and is not necessarily in the middle of the object. `getCenterOfMass()` returns both components as `[x, y]`.

---

isAwake

Set or get a boolean indicating whether the Physics object is awake or asleep. Physics simulations are relatively CPU intensive, requiring a lot of calculations. To save processor time, the Physics engine will make objects that have come to a complete stop go in to "sleep" mode so they no longer require processing. However sometimes changes like repositioning an adjacent object will leave the object in "sleep" mode so it will not respond properly. In this situation setting `isAwake` to `true` can be used to force a sleeping object to resume simulation. (It can also be used to force an object to go in to "sleep" mode, but note this is normally done automatically when possible.)

---

isSleeping

Deprecated Returns true when `isAwake` is false. Only provided for backwards compatibility; use `isAwake` instead.

---

createDistanceJoint(imgPt, iOtherInst, otherImgPt, damping, freq)

Fix two physics objects at a given distance apart, as if connected by a pole. The other instance must be an IWorldInstance which also uses the Physics behavior. An image point can be specified for each with `imgPt` to connect to a specific part of the object. Note that an image point of `0` specifies the center of gravity of the object - if you intend to connect to the object origin, use `-1`. `damping` is the joint damping ratio from 0 to 1, and `freq` is the mass-spring-damper frequency in Hertz.

---

createRevoluteJoint(imgPt, iOtherInst)

createLimitedRevoluteJoint(imgPt, iOtherInst, lower, upper)

Hinge two physics objects together, so they can rotate freely as if connected by a pin. Limited revolute joints only allow rotation through a certain range of angles (given in radians), like the clapper of a bell. The other instance must be an IWorldInstance which also uses the Physics behavior. An image point can also be specified to connect to a specific part of the object. Note that an image point of `0` specifies the center of gravity of the object - if you intend to connect to the object origin, use `-1`.

---

createPrismaticJoint(imgPt, iOtherInst, axisAngle, enableLimit, lowerTranslation, upperTranslation, enableMotor, motorSpeed, maxMotorForce)

Restrict the movement of two physics objects along a specific axis, given by `axisAngle` in radians. An image point can also be specified to connect to a specific part of the object. Note that an image point of `0` specifies the center of gravity of the object - if you intend to connect to the object origin, use `-1`. The other instance must be an IWorldInstance which also uses the Physics behavior. `enableLimit` is a boolean specifying whether there is a lower and upper movement limit; if enabled these are given by the lower and upper translation (in pixels), otherwise unlimited movement is allowed along the axis. A motor can also be enabled by `enableMotor` to provide a continuous force along the axis with `motorSpeed` in radians per second, and `maxMotorForce` the maximum torque.

---

removeAllJoints()

Remove all joints from the object. Any objects this object was attached to via joints is also affected. Note some joints automatically disable collisions between the objects, so you may want to manually disable collisions again after removing joints otherwise overlapping objects will "teleport" apart (as the physics engine will try to prevent them overlapping).

---

getContactCount()

Return the number of locations the physics engine has identified this object as touching other physics objects.

---

getContactX(index)

getContactY(index)

getContact(index)

Return the position of a contact with another physics object, in layout co-ordinates, given by the zero-based index of the contact. The `getContact` variant returns `[x, y]`.

---

setCollisionFilter(isInclusive, tags)

Change the current *Collision filter mode* and *Collision filter tags* properties. `isInclusive` is a boolean where true sets inclusive collision filter mode, and false sets exclusive collision filter mode. `tags` may be either a space-separated string of tags, or an iterable of strings such as an array or Set.