12.6.4 Game Engine - Physics - Physics Types

Physics Types

- Static Physics
  - Options
  - Collision Bounds
  - Create Obstacle

Static Physics

Static objects in the *Blender Game Engine* do not automatically react to physics, including gravity and collisions. Even if hit by the force of a speeding 18-wheeler truck, it will remain unresponsive in terms of location, rotation, or deformation.

It will, however, give collision reactions. Objects will bounce off of Static Objects, and rotational inertia will transfer to objects capable of rotating (that is, Rigid Body Objects will spin in response, though Dynamic Objects will not).

Note that none of this prevents you from transforming the Static Objects with *Logic Bricks* or Python code. The visual objects will correctly move and their physics representation will update in the engine as well.

Another important note is that the default Collision Bounds is a Triangle Mesh, meaning it is higher in computational requirements but also in detail. This in turn means the “Radius” option has no effect by default.

For more documentation, see the *Top BGE Physics page*.

Options

Note

bpy Access

Note that most of these properties are accessible through the non- BGE scripting API via `bpy.data.objects["ObjectName"].game`, which is of type `bpy.types.GameObjectSetting`. This is useful so you can, for example, set a range of objects to have gradated values via a for-loop.

- Actor - Enables detection by Near and Radar Sensors. - Default: On. - Python property: `obj.game.use_actor`
- Ghost - Disables collisions completely, similar to No Collision. - Default: Off. - Python property: `obj.game.use_ghost`
- Invisible - Does not display, the same as setting the object to unrendered (such as unchecking the “Camera” icon in the Outliner). - Default: Off. - Python property: `obj.use_render`

Radius

If you have the “Collision Bounds: Sphere” set explicitly (or implicitly through having the Collision Bounds subpanel unchecked), this will multiply with the Object’s (unapplied) Scale. Note that none of the other bounds types are affected. Also note that in the 3D View the display will show this for all types, even though it is only actually used with Sphere. Python property: `obj.game.radius`

Basic	**Radius= 1.5**	**Unapplied Scale**	**Applied Scale**	**Collision Bounds**
Rolls, radius of 1 BU	Rolls, radius of 1.5 BU (after “popping” upward)	Rolls, radius of 1.5 BU	Rolls, radius of 1 BU (!)	Default (which is Sphere)
Slides, extent of 1 BU	Slides, extent of 1 BU	Slides, extent of 1 BU	Slides, extent of 1 BU	Box
“”	“”	“”	“”	Convex Hull
Slides, extent of 1 BU (but with more friction than above)	Slides, extent of 1 BU (but with more friction than above)	Acts insane	Slides extent of 1.5 BU	Triangle Mesh

Anisotropic Friction

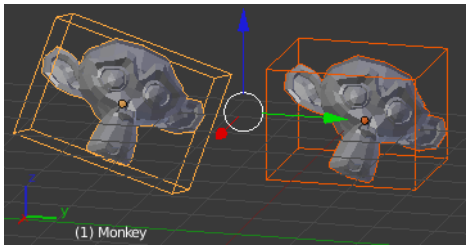
Isotropic friction is identical at all angles. Anisotropic is directionally-dependant. Here you can vary the coefficients for the three axes individually, or disable friction entirely. Python properties: `obj.game.use_anisotropic_friction` (boolean) and `obj.game.friction_coefficients` (a 3-element array).

Collision Bounds

Note

The Static type differs from the others in that it defaults to a Triangle Mesh bounds, instead of a simple sphere.

The first thing you must understand is the idea of the 3d Bounding Box. If you run through all the vertices of a mesh and record the lowest and highest x values, you have found the *x min/max* the complete boundary for all x values within the mesh. Do this again for y and z, then make a rectangular prism out of these values, and you have a *Bounding Box*. This box could be oriented relative globally to the world or locally to the object's rotation.

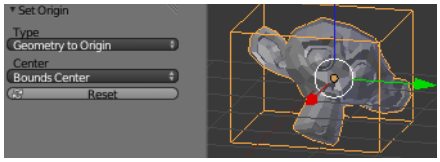


Demonstration of a Local Bounding Box (left) and a Global Bounding Box (right).

The *x extent*, then, is half of the distance between the x min/max.

Throughout all of this you must be cognizant of the Object Origin. For the Game engine, the default `Ctrl-Alt-Shift-C`, `3` (Set Origin ‣ Origin to Geometry) is unlikely to get the desired placement of the Collision Bounds that you want. Instead, you should generally set the origin by looking at the `T`-toolshelf after you do the *Set Origin*, and changing the *Center* from *Median Center* to *Bounds Center*. Blender will remember this change for future `Ctrl-Alt-Shift-C` executions.

All Collision Bounds are centered on this origin. All boxes are oriented locally, so object rotation matters.



Setting the origin to Bounds Center instead of Median Center.

A final introductory comment: When you set the Collision Bounds on an object, Blender will attempt to display a visualization of the bounds in the form of a dotted outline. Currently, there is a bug: *The 3D View* does not display this bounds preview where it actually will be during the game. To see it, go to Game ‣ Show Physics Visualization and look for the white (or green, if sleeping) geometry.

Now we can explain the various options for the *Collision Bounds* settings:

Default
: For Dynamic and Static objects, it is a Triangle Mesh (see below). For everything else, it is a Sphere (see below).

Capsule - A cylinder with hemispherical caps, like a pill.
: Radius of the hemispheres is the greater of the x or y extent. Height is the z bounds

Box

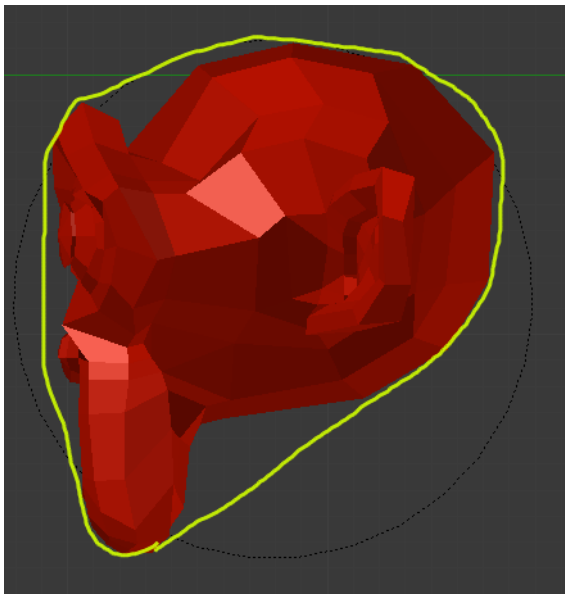
The x,y,z bounding box, as defined above.

Sphere
Radius is defined by the object's scale (visible in the N properties panel) times the physics radius (can be found in Physics » Attributes » Radius. Note: This is the only bounds that respects the Radius option.

Cylinder
Radius is the greater of the x or y extent. Height is the z bounds.

Cone
Base radius is the greater of the x or y extent. Height is the z bounds.

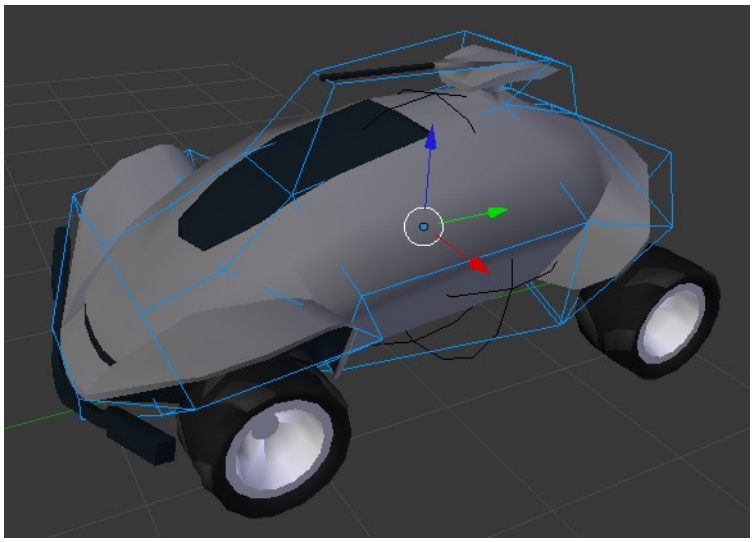
Convex Hull
Forms a shrink-wrapped, simplified geometry around the object.



A convex hull sketch

Triangle mesh
Most expensive, but most precise. Collision will happen with all of triangulated polygons, instead of using a virtual mesh to approximate that collision.

By Hand
This is not an option in the Physics tab's Collision Bounds settings, but a different approach, entirely. You create a second mesh, which is invisible, to be the physics representation. This becomes the parent for your display object. Then, your display object is set to ghost so it doesn't fight with the parent object. This method allows you to strike a balance between the accuracy of *Triangle Mesh* with the efficiency of some of the others. See the demo of this in the dune buggy to the right.



Another way to create Collision Bounds – By hand.

Options

There are only two options in the Collision Bounds subpanel.

Margin

> "Add extra margin around object for collision detection, small amount required for stability." If you find your objects are getting stuck in places they shouldn't, try increasing this to, say, 0.06.
>
> Sometimes 0.06 is the default (such as on the Default Cube), but sometimes it is not. You have to keep an eye on the setting, or else learn the symptoms so you can respond when it gives you trouble. If you're lazy/paranoid/unsure/diligent/bored, you can always run this on the Python Console to bump all 0.0 margins to 0.06: for *obj* in `bpy.data.objects`: `obj.game.collision_margin = obj.game.collision_margin` or 0.06

Compound

> "Add children to form compound collision object." Basically, if you have a child object and do not have this enabled, the child's collisions will not have an effect on that object "family" (though it will still push other objects around). If you do have it checked, the parent's physics will respond to the child's collision (thus updating the whole family). Python property: `obj.game.use_collision_compound`

Create Obstacle

Todo

No Collision Physics

"No Collision" objects in the *Game Engine* are completely unaffected by *Physics*, and do cause physics reactions. They are useful as pure display objects, such as the child of a *Custom Collision Hull* (Collision Bounds).

For more documentation, see the *Top BGE Physics page*.

Options

The only option available on No Collision types is:

Invisible
Does not display, the same as setting the object to unrendered (such as unchecking the “Camera” icon in the Outliner). Python property: `obj.use_render`

Dynamic Physics

Dynamic objects in the *Game Engine* give/receive collisions, but when they do so they themselves do not rotate in response. So, a Dynamic ball will hit a ramp and slide down, while a Rigid Body ball would begin rotating.

If you do not need the rotational response the Dynamic type can save the extra computation.

Note that these objects can still be rotated with *Logic Bricks* or Python code. Their physics meshes will update when you do these rotations - so collisions will be based on the new orientations.

For more documentation, see the *Top BGE Physics page*.

Options

Note

bpy Access

Note that most of these properties are accessible through the non- BGE scripting API via `bpy.data.objects["ObjectName"].game`, which is of type `bpy.types.GameObjectSetting`. This is useful so you can, for example, set a range of objects to have gradated values via a for-loop.

Actor
Enables detection by Near and Radar Sensors. Python property: `obj.game.use_actor`
Ghost
Disables collisions completely, similar to *No Collision*. `Python property: obj.game.use_ghost`
Invisible
Does not display, the same as setting the object to unrendered (such as unchecking the “Camera” icon in the *Outliner*. Python property: `obj.use_render`
Use Material Force Field
Materials can have physics settings on them as well: Friction, Elasticity, Force Field (positive or negative force), and also Dampening to other materials. When you turn on this checkbox, you are enabling the Material to exhibit this spring force. Python property: `obj.game.use_material_physics_fh`
Rotate From Normal
Todo Python property: `obj.game.use_rotate_from_normal`
No Sleeping
Prevents simulation meshes from sleeping. When an object has a linear velocity or angular velocity, it is in motion. It will detect collisions, receive gravity, etc. Once these thresholds are close to zero, it will

cease these calculations—until another object interacts with it wake it up. Python property: `obj.game.use_sleep`

Mass

Affects the reaction due to collision between objects – more massive objects have more inertia. Will also affect material force fields. Will also change behaviors if you are using the suspension and steering portions of Bullet physics. Python property: `obj.game.mass`

Radius

If you have the "Collision Bounds: Sphere" set explicitly (or implicitly through having the Collision Bounds subpanel unchecked), this will multiply with the Object's (unapplied) Scale. Note that none of the other bounds types are affected. Also note that in the 3D View the display will show this for all types, even though it is only actually used with Sphere. Python property: `obj.game.radius`

Basic	**Radius= 1.5**	**Unapplied Scale**	**Applied Scale**	**Collision Bounds**
Rolls, radius of 1 BU	Rolls, radius of 1.5 BU (after "popping" upward)	Rolls, radius of 1.5 BU	Rolls, radius of 1 BU (!)	Default (which is Sphere)
Slides, extent of 1 BU	Slides, extent of 1 BU	Slides, extent of 1 BU	Slides, extent of 1 BU	Box
""	""	""	""	Convex Hull
Slides, extent of 1 BU (but with more friction than above)	Slides, extent of 1 BU (but with more friction than above)	Acts insane	Slides extent of 1.5 BU	Triangle Mesh

Form Factor

For affecting the Inertia Tensor. The higher the value, the greater the rotational inertia, and thus the more resistant to torque. You might think this is strange, considering Dynamic types do not have torque in response to collisions – but you can still see this value's effects when you manually apply Torque. Python property: `obj.game.form_factor`

Anisotropic Friction

Isotropic friction is identical at all angles. Anisotropic is directionally-dependant. Here you can vary the coefficients for the three axes individually, or disable friction entirely. Python properties: `obj.game.use_anisotropic_friction` (boolean) and `obj.game.friction_coefficients` (a 3-element array).

Velocity- Limit the speed of an object 0 - 1000.

Minimum

The object is allowed to be at complete rest, but as soon as it accelerates it will immediately jump to the minimum speed. Python property: `obj.game.velocity_min`

Maximum

Top speed of the object. Python property: `obj.game.velocity_max`

Damping- Increase the "sluggishness" of the object.

Translation

Resist movement 0-1. At "1" the object is completely immobile. Python property: `obj.game.damping`

Rotation

Resist rotation, but not the kind of rotation that comes from a collision. For example, if a Motion Controller applies Torque to an object, this damping will be a factor. Python property: `obj.game.rotation_damping`

Lock Translation

Seize the object in the world along one or more axes. Note that this is global coordinates, not local or otherwise.

- X Python property: `obj.game.lock_location_x`
- Y Python property: `obj.game.lock_location_y`
- Z Python property: `obj.game.lock_location_z`

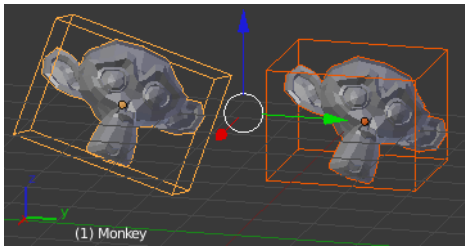
Lock Rotation

Same, but for rotation (also with respect to the global coordinates).

- X Python property: `obj.game.lock_rotation_x`
- Y Python property: `obj.game.lock_rotation_y`
- Z Python property: `obj.game.lock_rotation_z`

Collision Bounds

The first thing you must understand is the idea of the 3d Bounding Box. If you run through all the vertices of a mesh and record the lowest and highest x values, you have found the *x min/max* the complete boundary for all x values within the mesh. Do this again for y and z, then make a rectangular prism out of these values, and you have a *Bounding Box*. This box could be oriented relative globally to the world or locally to the object's rotation.

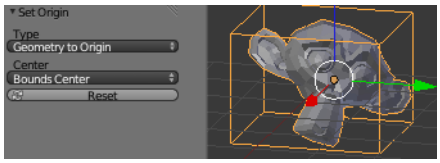


Demonstration of a Local Bounding Box (left) and a Global Bounding Box (right).

The *x extent*, then, is half of the distance between the x min/max.

Throughout all of this you must be cognizant of the Object Origin. For the Game engine, the default `Ctrl-Alt-Shift-C`, `3` (Set Origin ‣ Origin to Geometry) is unlikely to get the desired placement of the Collision Bounds that you want. Instead, you should generally set the origin by looking at the `T`-toolshelf after you do the *Set Origin*, and changing the *Center* from *Median Center* to *Bounds Center*. Blender will remember this change for future `Ctrl-Alt-Shift-C` executions.

All Collision Bounds are centered on this origin. All boxes are oriented locally, so object rotation matters.



Setting the origin to Bounds Center instead of Median Center.

A final introductory comment: When you set the Collision Bounds on an object, Blender will attempt to display a visualization of the bounds in the form of a dotted outline. Currently, there is a bug: *The 3D View* does not display this bounds preview where it actually will be during the game. To see it, go to Game ‣ Show Physics Visualization and look for the white (or green, if sleeping) geometry.

Now we can explain the various options for the *Collision Bounds* settings:

Default
: For Dynamic and Static objects, it is a Triangle Mesh (see below). For everything else, it is a Sphere (see below).

Capsule - A cylinder with hemispherical caps, like a pill.
: Radius of the hemispheres is the greater of the x or y extent. Height is the z bounds

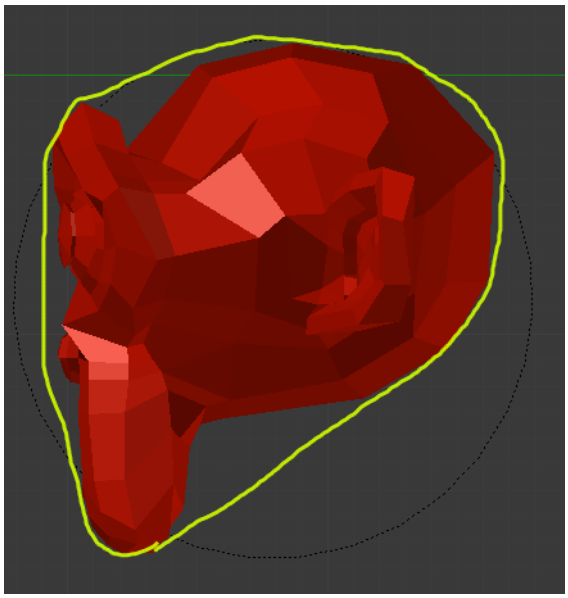
Box
: The x,y,z bounding box, as defined above.

Sphere
: Radius is defined by the object's scale (visible in the N properties panel) times the physics radius (can be found in Physics » Attributes » Radius. Note: This is the only bounds that respects the Radius option.

Cylinder
: Radius is the greater of the x or y extent. Height is the z bounds.

Cone
: Base radius is the greater of the x or y extent. Height is the z bounds.

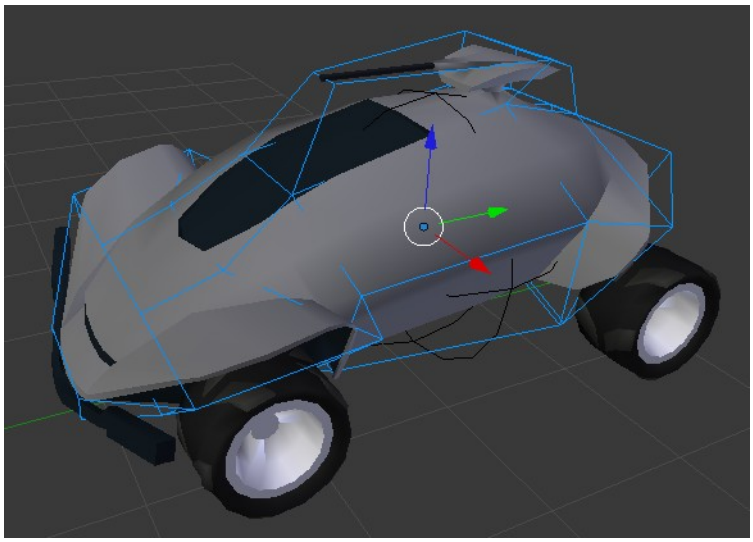
Convex Hull
: Forms a shrink-wrapped, simplified geometry around the object.



A convex hull sketch

Triangle mesh
: Most expensive, but most precise. Collision will happen with all of triangulated polygons, instead of using a virtual mesh to approximate that collision.

By Hand
: This is not an option in the Physics tab's Collision Bounds settings, but a different approach, entirely. You create a second mesh, which is invisible, to be the physics representation. This becomes the parent for your display object. Then, your display object is set to ghost so it doesn't fight with the parent object. This method allows you to strike a balance between the accuracy of *Triangle Mesh* with the efficiency of some of the others. See the demo of this in the dune buggy to the right.



Another way to create Collision Bounds – By hand.

Options

There are only two options in the Collision Bounds subpanel.

Margin

> “Add extra margin around object for collision detection, small amount required for stability.” If you find your objects are getting stuck in places they shouldn’t, try increasing this to, say, 0.06.
>
> Sometimes 0.06 is the default (such as on the Default Cube), but sometimes it is not. You have to keep an eye on the setting, or else learn the symptoms so you can respond when it gives you trouble. If you’re lazy/paranoid/unsure/diligent/bored, you can always run this on the Python Console to bump all 0.0 margins to 0.06: for *obj* in `bpy.data.objects`: `obj.game.collision_margin = obj.game.collision_margin` or 0.06

Compound

> “Add children to form compound collision object.” Basically, if you have a child object and do not have this enabled, the child’s collisions will not have an effect on that object “family” (though it will still push other objects around). If you do have it checked, the parent’s physics will respond to the child’s collision (thus updating the whole family). Python property: `obj.game.use_collision_compound`

Create Obstacle

Todo

Rigid Body Physics

Probably the most common type of object in the *Game Engine*. It will give/receive collisions and react with a change in its velocity and its rotation. A Rigid Body ball would begin rotating and roll down (where a *Dynamic* ball would only hit and slide down the ramp).

The idea behind Rigid Body dynamics is that the mesh does not deform. If you need deformation you will need to either go to *Soft Body* or else fake it with animated Actions.

For more documentation, see the *Top BGE Physics page*.

Options

Note

bpy Access

Note that most of these properties are accessible through the non- BGE scripting API via `bpy.data.objects["ObjectName"].game`, which is of type `bpy.types.GameObjectSetting`. This is useful so you can, for example, set a range of objects to have gradated values via a for-loop.

Actor
: Enables detection by Near and Radar Sensors. Python property: `obj.game.use_actor`

Ghost
: Disables collisions completely, similar to *No Collision*. `Python property: obj.game.use_ghost`

Invisible
: Does not display, the same as setting the object to unrendered (such as unchecking the “Camera” icon in the *Outliner*. Python property: `obj.use_render`

Use Material Force Field
: Materials can have physics settings on them as well: Friction, Elasticity, Force Field (positive or negative force), and also Dampening to other materials. When you turn on this checkbox, you are enabling the Material to exhibit this spring force. Python property: `obj.game.use_material_physics_fh`

Rotate From Normal
: Todo Python property: `obj.game.use_rotate_from_normal`

No Sleeping
: Prevents simulation meshes from sleeping. When an object has a linear velocity or angular velocity, it is in motion. It will detect collisions, receive gravity, etc. Once these thresholds are close to zero, it will cease these calculations—until another object interacts with it wake it up. Python property: `obj.game.use_sleep`

Mass
: Affects the reaction due to collision between objects – more massive objects have more inertia. Will also affect material force fields. Will also change behaviors if you are using the suspension and steering portions of Bullet physics. Python property: `obj.game.mass`

Radius
: If you have the “Collision Bounds: Sphere” set explicitly (or implicitly through having the Collision Bounds subpanel unchecked), this will multiply with the Object’s (unapplied) Scale. Note that none of the other bounds types are affected. Also note that in the 3D View the display will show this for all types, even though it is only actually used with Sphere. Python property: `obj.game.radius`

Basic	Radius= 1.5	Unapplied Scale	Applied Scale	Collision Bounds
Rolls, radius of 1 BU	Rolls, radius of 1.5	Rolls, radius of 1.5	Rolls, radius of 1 BU	Default (which is

	BU (after “popping” upward)	BU	(!)	Sphere)
Slides, extent of 1 BU	Slides, extent of 1 BU	Slides, extent of 1 BU	Slides, extent of 1 BU	Box
“”	“”	“”	“”	Convex Hull
Slides, extent of 1 BU (but with more friction than above)	Slides, extent of 1 BU (but with more friction than above)	Acts insane	Slides extent of 1.5 BU	Triangle Mesh

Form Factor

For affecting the Inertia Tensor. The higher the value, the greater the rotational inertia, and thus the more resistant to torque. You might think this is strange, considering Dynamic types do not have torque in response to collisions – but you can still see this value’s effects when you manually apply Torque. Python property: `obj.game.form_factor`

Anisotropic Friction

Isotropic friction is identical at all angles. Anisotropic is directionally-dependant. Here you can vary the coefficients for the three axes individually, or disable friction entirely. Python properties: `obj.game.use_anisotropic_friction` (boolean) and `obj.game.friction_coefficients` (a 3-element array).

Velocity- Limit the speed of an object 0 - 1000.

Minimum

The object is allowed to be at complete rest, but as soon as it accelerates it will immediately jump to the minimum speed. Python property: `obj.game.velocity_min`

Maximum

Top speed of the object. Python property: `obj.game.velocity_max`

Damping- Increase the “sluggishness” of the object.

Translation

Resist movement 0-1. At “1” the object is completely immobile. Python property: `obj.game.damping`

Rotation

Resist rotation, but not the kind of rotation that comes from a collision. For example, if a Motion Controller applies Torque to an object, this damping will be a factor. Python property: `obj.game.rotation_damping`

Lock Translation

Seize the object in the world along one or more axes. Note that this is global coordinates, not local or otherwise.

- X Python property: `obj.game.lock_location_x`
- Y Python property: `obj.game.lock_location_y`
- Z Python property: `obj.game.lock_location_z`

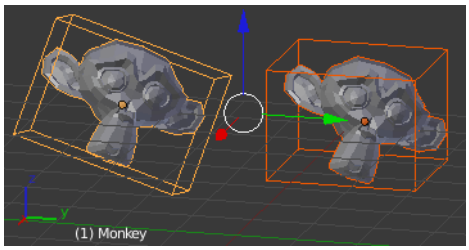
Lock Rotation

Same, but for rotation (also with respect to the global coordinates).

- X Python property: `obj.game.lock_rotation_x`
- Y Python property: `obj.game.lock_rotation_y`
- Z Python property: `obj.game.lock_rotation_z`

Collision Bounds

The first thing you must understand is the idea of the 3d Bounding Box. If you run through all the vertices of a mesh and record the lowest and highest x values, you have found the *x min/max* the complete boundary for all x values within the mesh. Do this again for y and z, then make a rectangular prism out of these values, and you have a *Bounding Box*. This box could be oriented relative globally to the world or locally to the object's rotation.

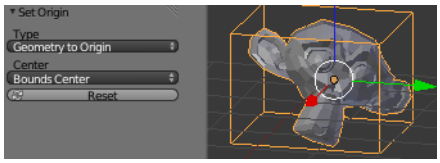


Demonstration of a Local Bounding Box (left) and a Global Bounding Box (right).

The *x extent*, then, is half of the distance between the x min/max.

Throughout all of this you must be cognizant of the Object Origin. For the Game engine, the default `Ctrl-Alt-Shift-C`, `3` (Set Origin ‣ Origin to Geometry) is unlikely to get the desired placement of the Collision Bounds that you want. Instead, you should generally set the origin by looking at the `T`-toolshelf after you do the *Set Origin*, and changing the *Center* from *Median Center* to *Bounds Center*. Blender will remember this change for future `Ctrl-Alt-Shift-C` executions.

All Collision Bounds are centered on this origin. All boxes are oriented locally, so object rotation matters.



Setting the origin to Bounds Center instead of Median Center.

A final introductory comment: When you set the Collision Bounds on an object, Blender will attempt to display a visualization of the bounds in the form of a dotted outline. Currently, there is a bug: *The 3D View* does not display this bounds preview where it actually will be during the game. To see it, go to Game ‣ Show Physics Visualization and look for the white (or green, if sleeping) geometry.

Now we can explain the various options for the *Collision Bounds* settings:

Default
: For Dynamic and Static objects, it is a Triangle Mesh (see below). For everything else, it is a Sphere (see below).

Capsule - A cylinder with hemispherical caps, like a pill.
: Radius of the hemispheres is the greater of the x or y extent. Height is the z bounds

Box
: The x,y,z bounding box, as defined above.

Sphere
: Radius is defined by the object's scale (visible in the N properties panel) times the physics radius (can be found in Physics » Attributes » Radius. Note: This is the only bounds that respects the Radius option.

Cylinder

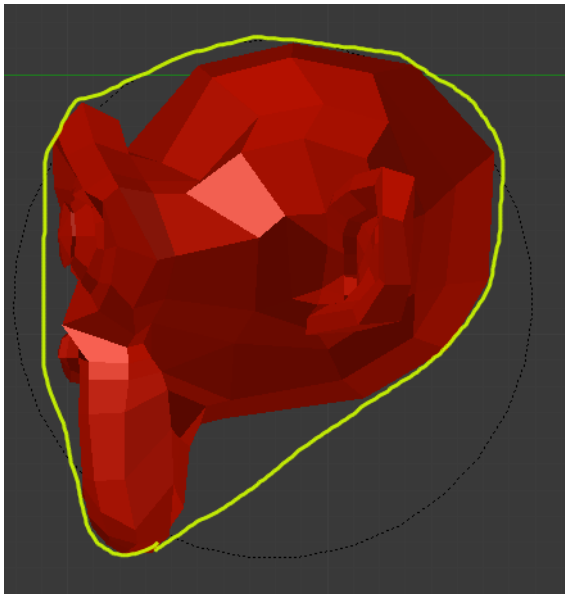
Radius is the greater of the x or y extent. Height is the z bounds.

Cone

Base radius is the greater of the x or y extent. Height is the z bounds.

Convex Hull

Forms a shrink-wrapped, simplified geometry around the object.



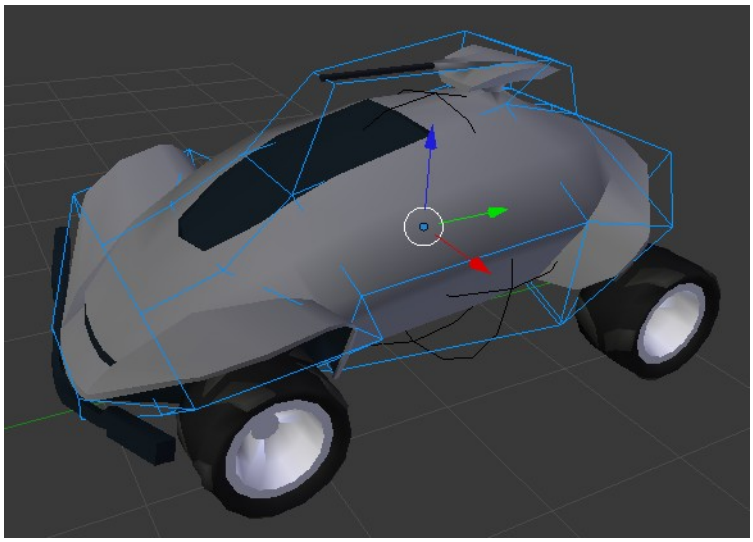
A convex hull sketch

Triangle mesh

Most expensive, but most precise. Collision will happen with all of triangulated polygons, instead of using a virtual mesh to approximate that collision.

By Hand

This is not an option in the Physics tab's Collision Bounds settings, but a different approach, entirely. You create a second mesh, which is invisible, to be the physics representation. This becomes the parent for your display object. Then, your display object is set to ghost so it doesn't fight with the parent object. This method allows you to strike a balance between the accuracy of *Triangle Mesh* with the efficiency of some of the others. See the demo of this in the dune buggy to the right.



Another way to create Collision Bounds – By hand.

Options

There are only two options in the Collision Bounds subpanel.

Margin

> "Add extra margin around object for collision detection, small amount required for stability." If you find your objects are getting stuck in places they shouldn't, try increasing this to, say, 0.06.
>
> Sometimes 0.06 is the default (such as on the Default Cube), but sometimes it is not. You have to keep an eye on the setting, or else learn the symptoms so you can respond when it gives you trouble. If you're lazy/paranoid/unsure/diligent/bored, you can always run this on the Python Console to bump all 0.0 margins to 0.06: for *obj* in `bpy.data.objects`: `obj.game.collision_margin = obj.game.collision_margin` or 0.06

Compound
> "Add children to form compound collision object." Basically, if you have a child object and do not have this enabled, the child's collisions will not have an effect on that object "family" (though it will still push other objects around). If you do have it checked, the parent's physics will respond to the child's collision (thus updating the whole family). Python property: `obj.game.use_collision_compound`

Create Obstacle

Todo

Soft Body Physics

The most advanced type of object in the *Game Engine*. Also, it is the most finicky. If you are used to the fun experimentation that comes from playing around with the non-BGE Soft Body sims (such as Cloth), you will probably find a frustrating lack of options and exciting results. Do not despair, we are here to help you get some reasonable settings.

Your setup will involve making sure you have sufficient geometry in the Soft Body's mesh to support the deformation, as well as tweaking the options.

Options

Actor

> Enables detection by Near and Radar Sensors.
>
> - Default: On.
> - Python property: `obj.game.use_actor`

Ghost

> Disables collisions completely, similar to No Collision.
>
> - Default: Off.
> - Python property: `obj.game.use_ghost`

Invisible

Does not display, the same as setting the object to unrendered (such as unchecking the “Camera” icon in the Outliner).

- Default: Off.
- Python property: `obj.use_render`

Mass

Affects the reaction due to collision between objects – more massive objects have more inertia. Will also affect material force fields. Will also change behaviors if you are using the suspension and steering portions of Bullet physics.

- Range: 0.01-10,000.
- Default: 1.
- Python property: `obj.game.mass`

Shape Match

Upon starting the Game Engine this will record the starting shape of the mesh as the “lowest energy” state. This means that the edges will have tension whenever they are flexed to some other form. This is set to on by default, and in this configuration turns the object into more of a thin sheet of metal rather than a cloth.

- Default: On.
- Python property: `obj.game.soft_body.use_shape_match`

Threshold

Linearly scales the pose match

- A threshold of 1.0 makes it behave like *Shape Match* on with a *Linear Stiffness* of 1.0.
- A threshold of 0.0 makes it behave like *Shape Match* off with a *Linear Stiffness* of 0.0.
- Range: 0-1.
- Default: 0.5.
- Python property: `obj.game.soft_body.shape_threshold`

Welding

TODO.

Position Iteration

Increase the accuracy at a linearly-increasing expense of time. The effect is visible especially with Soft Bodies that fall on sharp corners, though this can slow down even very simple scenes.

- Range: 0-10.
- Default: 2.
- Python property: `obj.game.soft_body.location_iterations`

Linear Stiffness

Linear stiffness of the soft body links. This is most evident when you have *Shape Match* off, but it is also evident with it on.

- Range: 0-1.
- Default: 0.5.
- Python property: `obj.game.soft_body.linear_stiffness`

Friction
Dynamic friction coefficient.

Margin
Small value makes the algorithm unstable.

Bending Constraint
Enable Bending Constraints

Cluster Collision

Affects Collision sensors as well as physics.

Rigid to Soft Body

Enable cluster collisions between Rigid and Soft Bodies.

- Default: Off.
- Python property: `obj.game.soft_body.use_cluster_rigid_to_softbody`

Soft to Soft Body

Enable cluster collisions among Soft Bodies.

- Default: Off.
- Python property: `obj.game.soft_body.use_cluster_soft_to_softbody`

Iterations

Number of cluster iterations.

- Range: 1-128.
- Default: 64.
- Python property: `obj.game.soft_body.cluster_iterations`

Hints

- A very important configurable in the case of Soft Body interactions is *World Properties* ‣ Physics ‣ Physics Steps ‣ Substeps.
- Surprisingly, the more vertices you have in your hit object, the less likely the Soft Body is to react with it. If you try letting it hit a Plane, it might stop, but a subdivided Grid might fail.

Note

Soft bodies do not work with the Collision, Touch, Near, and Radar logic brick sensors.

Warning

A common practice within the non-BGE Cloth simulator is to employ *Force Fields* to animate the cloth. These do not work in the BGE, so you will have to figure out a way to use Python (or perhaps plain Logic Bricks) to apply forces to the Soft Body objects.

Goal Weights

Vehicle Controller Physics

Introduction

The Vehicle Controller is a special type of physics object that the Physics Engine (bullet) recognizes.

It is composed of a **rigid body** representing the chassis and a set of wheels that are set to **no collision**. Emphasizing the distinction between a GameEngine, Logical or Render object and its representation for the Physics Engine is important.

To simulate a vehicle as a true rigid body, on top of also rigid body wheels, with a real suspension system made with joints, would be far too complicated and unstable. Cars and other vehicles are complicated mechanical devices and most often we do not want to simulate that, only that it ‘acts as expected’. The Vehicle Controller exists to provide a dedicated way of simulating a vehicle behavior without having to simulate all the physics that would actually happen in the real world. It abstracts the complexity away by providing a simple interface with tweakable parameters such as suspension force, damping and compression.

How it works

Bullet’s approach to a vehicle controller is called a “Raycast Vehicle”. Collision detection for the wheels is approximated by ray casts and the tire friction is an anisotropic friction model.

A raycast vehicle works by casting a ray for each wheel. Using the ray’s intersection point, we can calculate the suspension length and hence the suspension force that is then applied to the chassis, keeping it from hitting the ground. In effect, the vehicle chassis ‘floats’ along on the rays.

The friction force is calculated for each wheel where the ray contacts the ground. This is applied as a sideways and forwards force.

You can check Kester Maddock’s approach to vehicle simulation here. It includes some common problems, workarounds and tips and tricks.

How to use

Currently the Vehicle Controller can only be used as a constraint via Python. There are plans to add it to the interface.

Setup

You should have a body acting as the chassis, set it as a ‘Rigid Body’.

The wheels should be separate objects set to ‘No Collision’. The vehicle controller will calculate the collisions for you as rays so, if you set it to something else, it will calculate it twice in different ways and produce weird results.

Collisions

A cylinder is typically a good collision shape for the wheels. For the chassis, the shape should be rough, like a box. If the vehicle is very complicated, you should split it into simpler objects and parent those (with their collision shapes) to the vehicle controller so that they will follow it. If your vehicle even has moving bits (weapons, wrecking balls, trolleys etc) they should also be simulated separately and connected to the vehicle as a joint.

Python

Assembling the Vehicle

The overall steps are:

- create a constraint for the vehicle and save its ID for future reference
- attach the wheels
- set wheel parameters: influence, stiffness, damping, compression and friction
- init variables

You can see an example in the file below.

Controlling the Vehicle

This is done in 2 parts and it should be modeled according to the desired behavior. You should think of your gameplay and research appropriate functions for the input. For instance, can the vehicle reverse? jump? drift? does it turn slowly? How much time does it take to brake or get to full speed? The first part is **response to keys**. Whenever the player presses a key, you should set a value accordingly, such as increase acceleration. Example:

```
if key[0] == events.UPARROWKEY:
    logic.car["force"] = -15.0
elif key[0] == events.RIGHTARROWKEY:
    logic.car["steer"] -= 0.05
```

The second part is to **compute the movement** according to your functions.

```
## apply engine force ##
for i in range(0, totalWheels):
    vehicle.applyEngineForce(logic.car["force"],i)
...
## slowly ease off gas and center steering ##
logic.car["steer"] *= 0.6
logic.car["force"] *= 0.9
```

Both should be run each frame.

Example

demo_file.zip (last update 9 September 2014)

https://dl.dropboxusercontent.com/u/3226675/blender/vehicle_controller_demo.zip

Occlude Object Physics

If an Occlude type object is between the camera and another object, that other object will not be rasterized (calculated for rendering). It is culled because it is occluded.

There is a demo .blend file to examplify some concepts: BGE-Physics-Objects-Occluder.blend

- A messed-up, subdivided Cube named “Cube”.
- Another one behind a “Physics Type: Occlude” plane, named “Cube.BG”.
- Another one outside the view Frustum, named “Cube.OffCamera”.

Now observe what happens to the profiling stats for each of the following (in order):

- Hit P as the scene is. It hums along at a fairly slow rate. On my system the Rasterizer step takes 130ms. The framerate will finally jump up once the “Cube” object has completely moved out of the view frustum. It’s as if the Occluder doesn’t do anything while the Cube is behind it.
- Delete the “Cube.OffCamera” object above, and notice that there is no improvement in speed. This is the view frustum culling working for you - it does not matter if that object exists or not.
- Hit Z to view wireframe. Notice that in the 3D Viewport you can see “Cube.BG”, but once you press P, it is not there.
- Make the “Occluder” object take up the whole camera’s view with S-X-5. You will see a huge leap in framerate, since almost nothing is being Rasterized. On my system the Rasterizer step drops to 5ms.
- Try a run with World properties ▸ Physics ▸ Occlusion Culling disabled. It will be slow again.
- Reenable World properties ▸ Physics ▸ Occlusion Culling and run it one more time to prove to yourself that your speed is back.
- Change the Occluder to “Physics Type: Static”. Notice that it is back to the original slowness.
- Change it back to “Physics Type: Occlude”.
- Now make the “Occluder” invisible. The framerate is back down to its original, slow rate.

Details

As far as Physics is concerned, this type is equivalent to Rigid Object “No collision”. The reason why the Occluder mode is mutually exclusive with other physics mode is to emphasize the fact that occluders should be specifically designed for that purpose and not every mesh should be an occluder. However, you can enable the Occlusion capability on physics objects using Python and Logic bricks - see (Link- TODO)

When an occluder object enters the view frustrum, the BGE builds a ZDepth buffer from the faces of that object. Whether the faces are one-side or two-side is important: only the front faces and two-side faces are used to build the ZDepth buffer. If multiple occluders are in the view frustrum, the BGE combines them and keeps the most foreground faces.

The resolution of the ZDepth buffer is controllable in the World settings with the “Occlu Res” button:

By default the resolution is 128 pixels for the largest dimension of the viewport while the resolution of the other dimension is set proportionally. Although 128 is a very low resolution, it is sufficient for the purpose of culling. The resolution can be increased to maximum 1024 but at great CPU expense.

The BGE traverses the DBVT (Dynamic Bounding Volume Tree) and for each node checks if it is entirely hidden by the occluders and if so, culls the node (and all the objects it contains).

To further optimize the feature, the BGE builds and uses the ZDepth buffer only when at least one occluder is in

the view frustrum. Until then, there is no performance decrease compared to regular view frustrum culling.

Recommendations

Occlusion culling is most useful when the occluders are large objects (buildings, mountains...) that hide many complex objects in an unpredictable way. However, don’t be too concerned about performance: even if you use it inappropriately, the performance decrease will be limited due to the structure of the algorithm.

There are situations where occlusion culling will not bring any benefit:

- If the occluders are small and don’t hide many objects.
    - In that case, occlusion culling is just dragging your CPU down).
- If the occluders are large but hides simple objects.
    - In that case you’re better off sending the objects to the GPU).
- If the occluders are large and hides many complex objects but in a very predictable way.
    - Example: a house full of complex objects. Although occlusion culling will perform well in this case, you will get better performance by implementing a specific logic that hides/unhides the objects; for instance making the objects visible only when the camera enters the house).
- Occluders can be visible graphic objects but beware that too many faces will make the ZDepth buffer creation slow.
    - For example, a terrain is not a good candidate for occlusion: too many faces and too many overlap. Occluder can be invisible objects placed inside more complex objects (ex: “in the walls” of a building with complex architecture). Occluders can have “holes” through which you will see objects.

Sensor Physics

The object detects static and dynamic objects but not other collisions sensors objects. The Sensor is similar to the physics objects that underlie the Near and Radar sensors. Like the Near and Radar object it is:

- static and ghost
- invisible by default
- always active to ensure correct collision detection
- capable of detecting both static and dynamic objects
- ignoring collision with their parent
- capable of broadphase filtering based on: - Actor option: the collisioning object must have the Actor flag set to be detected - property/material: as specified in the collision sensors attached to it.

Broadphase filtering is important for performance reason: the collision points will be computed only for the objects that pass the broadphase filter.

- automatically removed from the simulation when no collision sensor is active on it

Unlike the Near and Radar object it can:

- take any shape, including triangle mesh
- be made visible for debugging (just use the Visible actuator)
- have multiple collision sensors using it

Other than that, the sensor objects are ordinary objects. You can move them freely or parent them. When parented to a dynamic object, they can provide advanced collision control to this object.

The type of collision capability depends on the shape:

- box, sphere, cylinder, cone, convex hull provide volume detection.
- triangle mesh provides surface detection but you can give some volume to the surface by increasing the margin in the Advanced Settings panel. The margin applies on both sides of the surface.

Performance tip:

- Sensor objects perform better than Near and Radar: they do less synchronizations because of the Scenegraph optimizations and they can have multiple collision sensors on them (with different property filtering for example).
- Always prefer simple shape (box, sphere) to complex shape whenever possible.
- Always use broadphase filtering (avoid collision sensor with empty propery/material)
- Use collision sensor only when you need them. When no collision sensor is active on the sensor object, it is removed from the simulation and consume no CPU.

Known limitations:

- When running Blender in debug mode, you will see one warning line of the console:

  ::

      warning btCollisionDispatcher::needsCollision: static-static collision!” In release mode this message is not printed.

- Collision margin has no effect on sphere, cone and cylinder shape.

Settings

Invisible
: See *Here*

Collision Bounds

See Here.

Character Physics

TODO.

Navigation Mesh Physics

TODO.