



Blender - Python API

#9



Serdar ARITAN

Department of Computer Graphics
Hacettepe University, Ankara, Turkey



Blender/Python API

Blender's Physics

Constraints (also known as joints) for rigid bodies connect two rigid bodies.

The physics constraints available in the animations are meant to be attached to an Empty object. The constraint then has fields which can be pointed at the two physics-enabled object which will be bound by the constraint. The Empty object provides a location and axis for the constraint distinct from the two constrained objects.



Blender/Python API

Blender's Physics

Rigid Body Constraint Type

FIXED, Glue rigid bodies together.

POINT, Constrain rigid bodies to move around common pivot point.

HINGE, Restrict rigid body rotation to one axis.

SLIDER, Restrict rigid body translation to one axis.

PISTON, Restrict rigid body translation and rotation to one axis.

GENERIC, Restrict translation and rotation to specified axes.

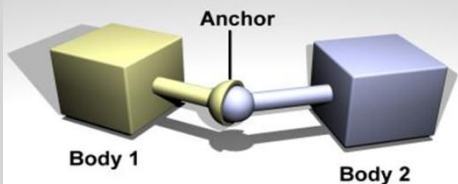
GENERIC_SPRING, Restrict translation and rotation to specified axes with springs.

MOTOR, Drive rigid body around or along an axis.

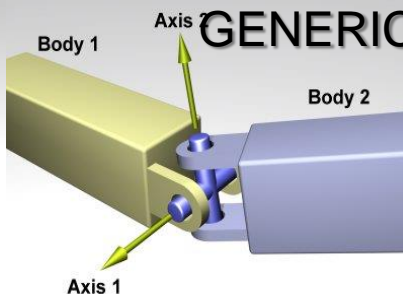


Blender/Python API Blender's Physics

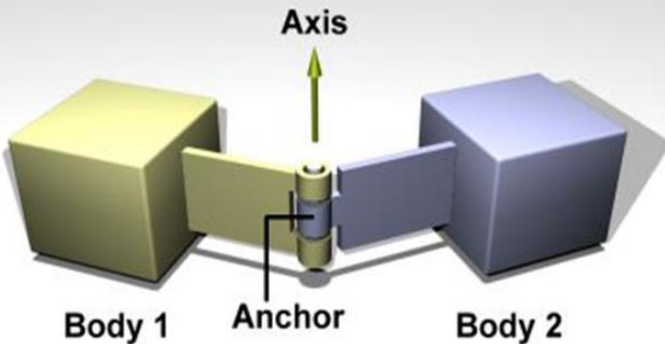
POINT



GENERIC



HINGE





Blender/Python API

Blender's Physics

Simulation Stability

The simplest way of improving simulation **stability** is to increase the **steps per second**. However, care has to be taken since making too many steps can cause problems and make the simulation even less stable (if you need more than 1000 steps, you should look at other ways to improve stability).

INCREASING the number of **solver iterations** helps making **CONSTRAINTS STRONGER** and also improves object stacking stability.

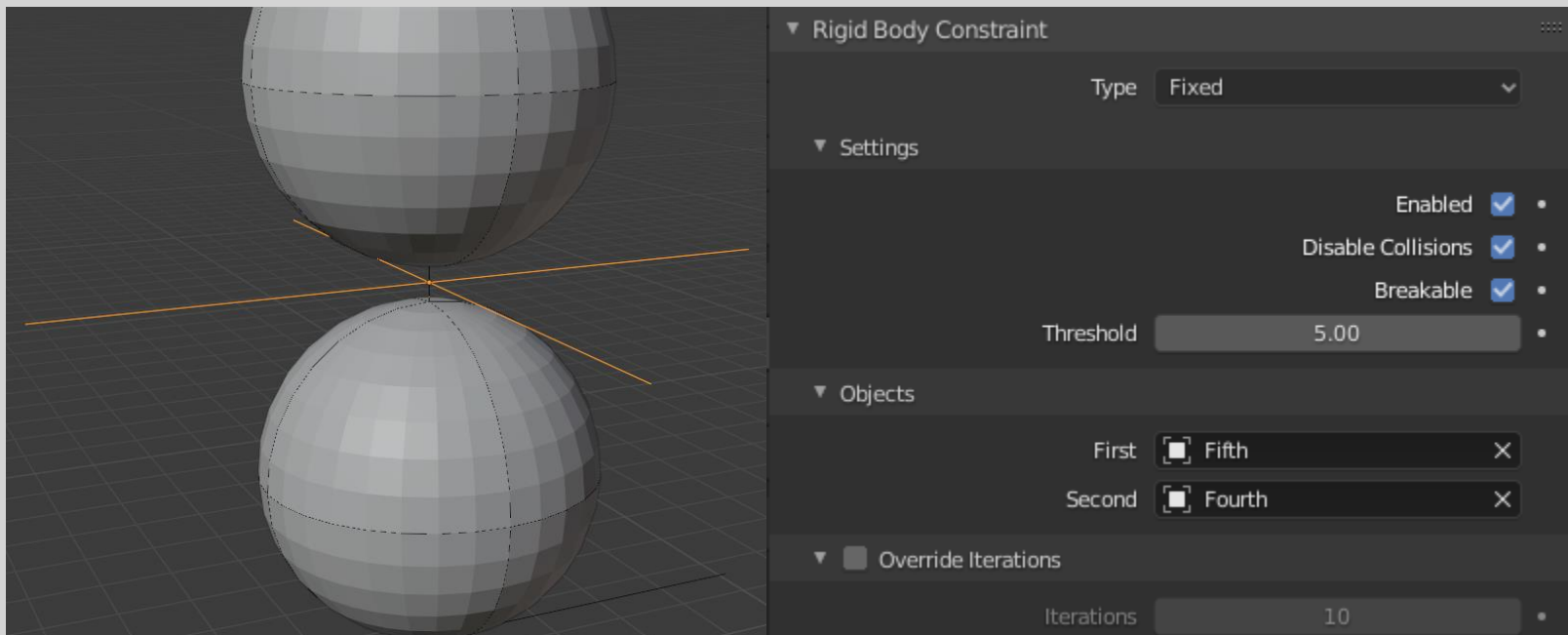
It is best to avoid small objects, as they are currently unstable. Ideally, objects should be at least 20 cm in diameter. If it is still necessary, setting the collision margin to 0, while generally not recommended, can help making small object behave more naturally.



Blender/Python API

Blender's Physics

Fixed : This constraint cause the two objects to move as one. Since the physics system does have a tiny bit of slop in it, the objects do not move as rigidly as they would if they were part of the same mesh.





Blender/Python API 5 Spheres and 1 Plane

```
import bpy
```

```
# plane
```

```
bpy.ops.mesh.primitive_plane_add(size=15, location=(0, 0, 0))
```

```
bpy.ops.rigidbody.object_add()
```

```
bpy.context.object.rigid_body.type = 'PASSIVE'
```

```
bpy.context.object.rigid_body.collision_shape = 'MESH'
```

```
for i in range(5, 11):
```

```
    bpy.ops.mesh.primitive_uv_sphere_add(radius=0.45, location=(0, 0, i))
```

```
    bpy.ops.rigidbody.object_add()
```

```
    bpy.context.object.rigid_body.type = 'ACTIVE'
```

```
    bpy.context.object.rigid_body.collision_shape = 'SPHERE'
```



SCRIPT LANGUAGES FOR ANIMATION

```
objNames = ['First', 'Second', 'Third', 'Fourth', 'Fifth', 'Sixth']

for i in range(5, 10):
    bpy.ops.object.empty_add(type='PLAIN_AXES', location=(0, 0, i+0.5))
    bpy.ops.rigidbody.constraint_add()
    bpy.context.object.rigidbody_constraint.type = 'POINT'
    bpy.context.object.rigidbody_constraint.disable_collisions = False
    bpy.context.object.rigidbody_constraint.use_breaking = False
    if bpy.context.object.rigidbody_constraint.use_breaking == True:
        bpy.context.object.rigidbody_constraint.breaking_threshold = 10

    bpy.context.object.rigidbody_constraint.object1 = bpy.data.objects[objNames[i-5]] # child
    bpy.context.object.rigidbody_constraint.object2 = bpy.data.objects[objNames[i-4]] # parent
```

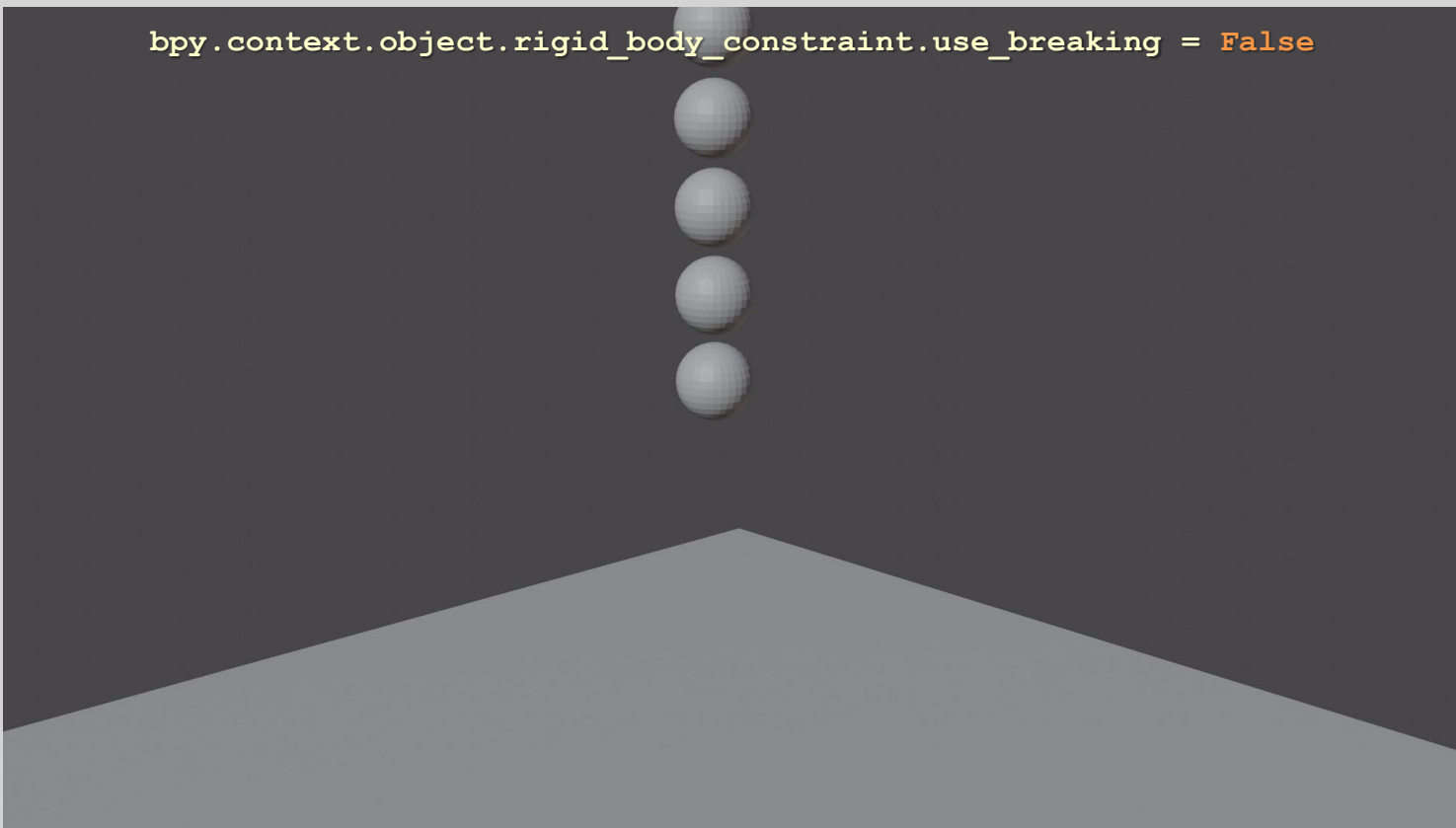



Blender/Python API

Blender's Physics

Fixed

```
bpy.context.object.rigid_body_constraint.use_breaking = False
```



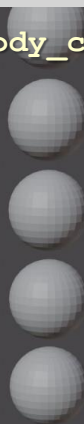


Blender/Python API

Blender's Physics

Fixed

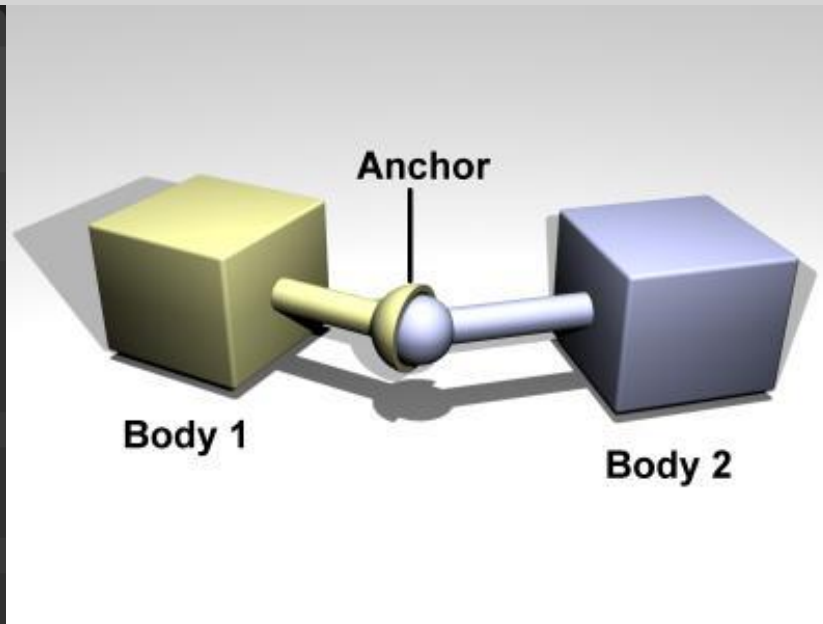
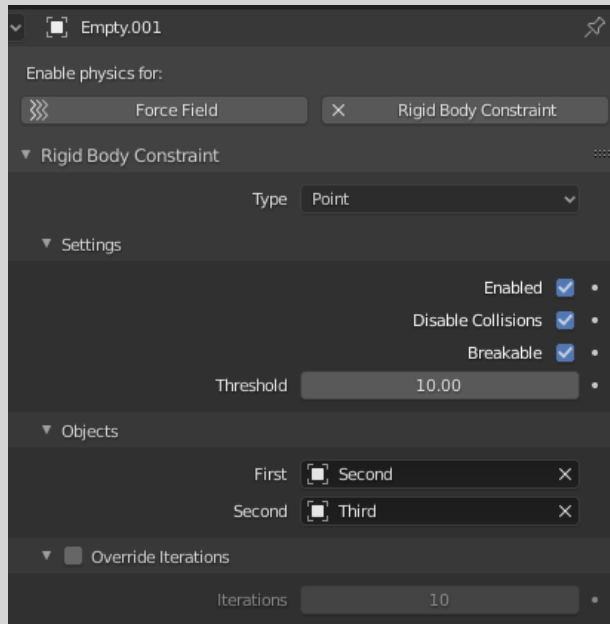
```
bpy.context.object.rigid_body_constraint.use_breaking = True
```



Blender/Python API

Blender's Physics

Point : The objects are linked by a point bearing allowing any kind of rotation around the location of the constraint object, but no relative translation is permitted.



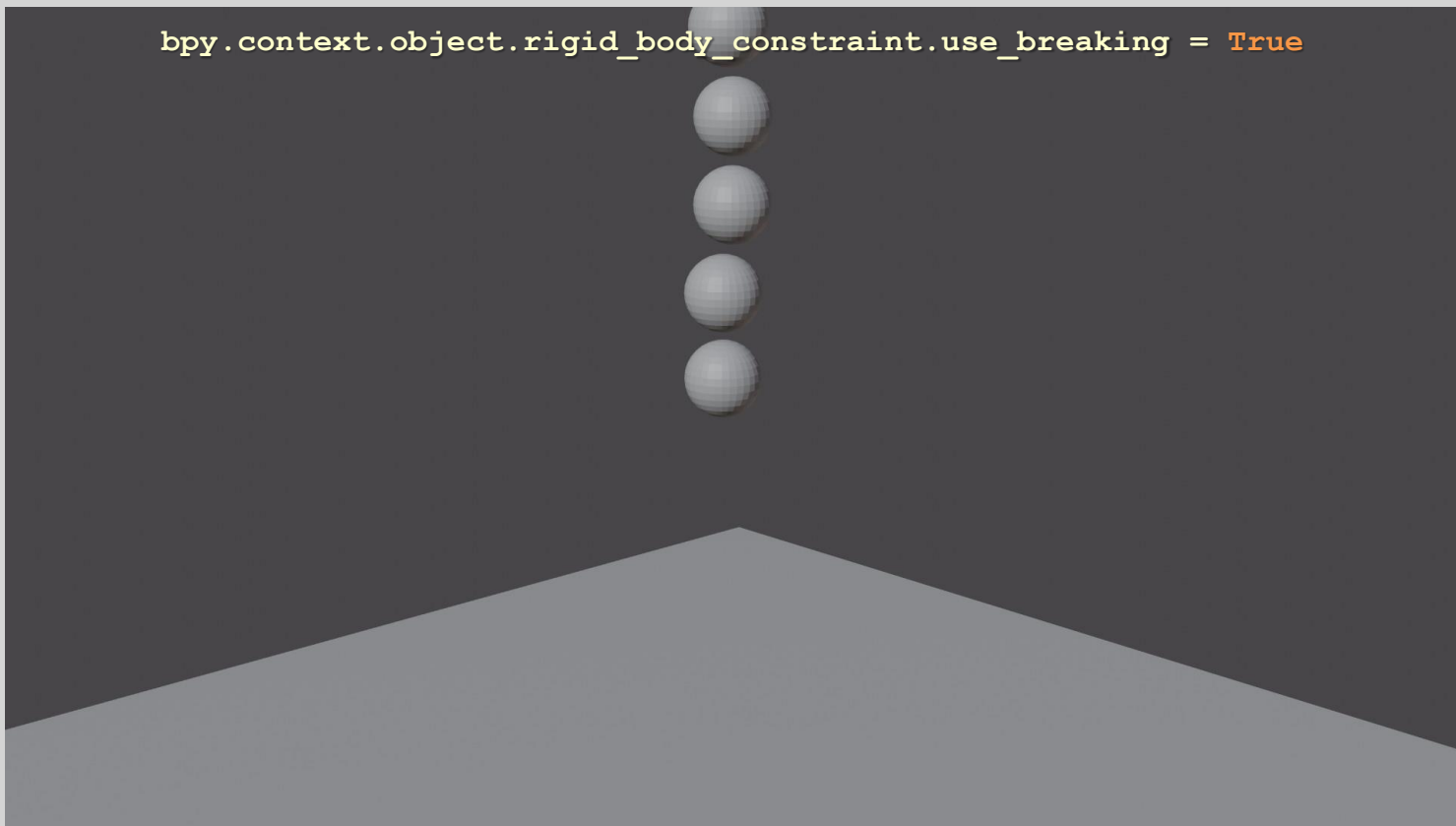


Blender/Python API

Blender's Physics

Point

```
bpy.context.object.rigid_body_constraint.use_breaking = True
```

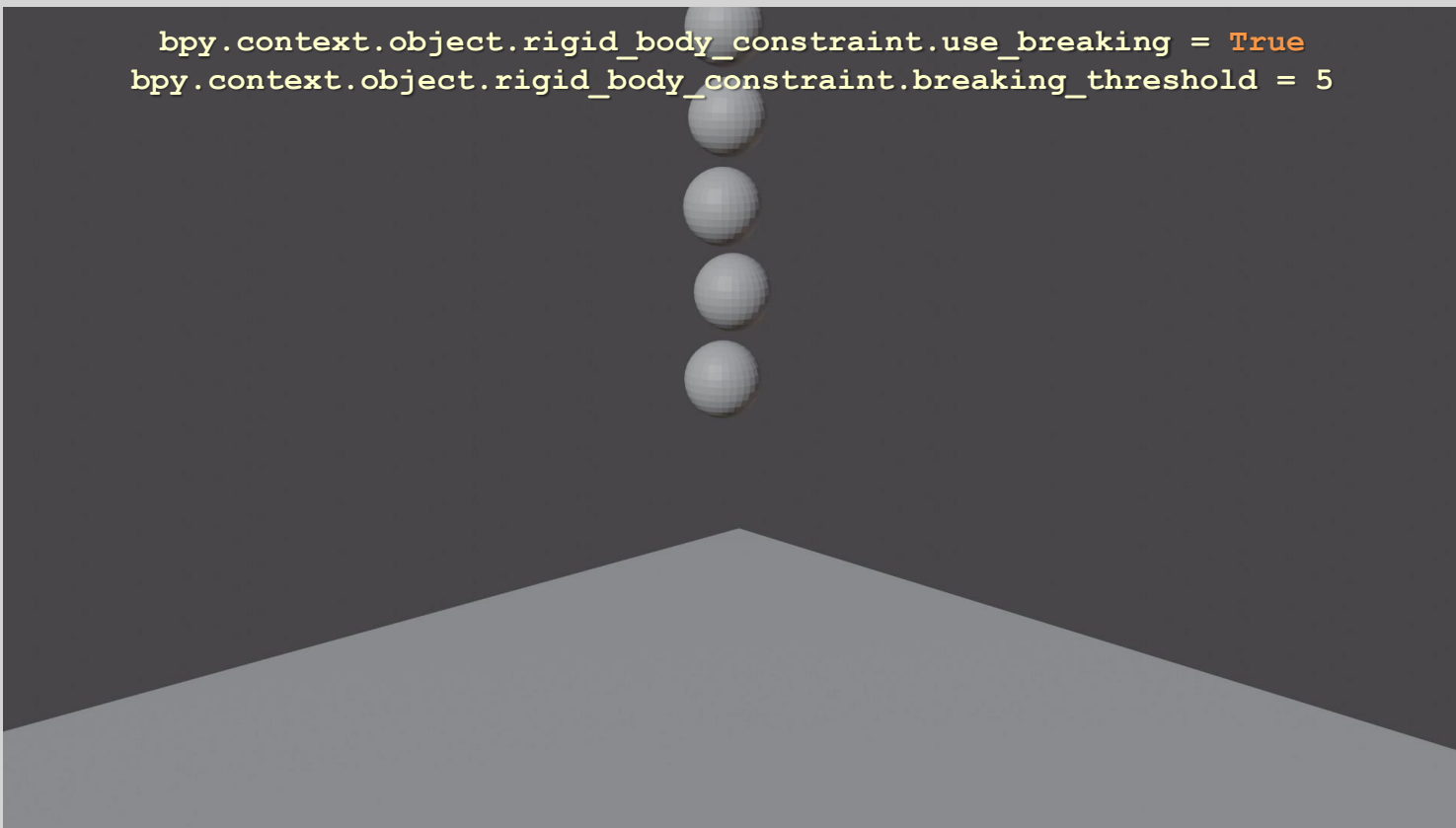




Blender/Python API Blender's Physics

Point

```
bpy.context.object.rigid_body_constraint.use_breaking = True  
bpy.context.object.rigid_body_constraint.breaking_threshold = 5
```

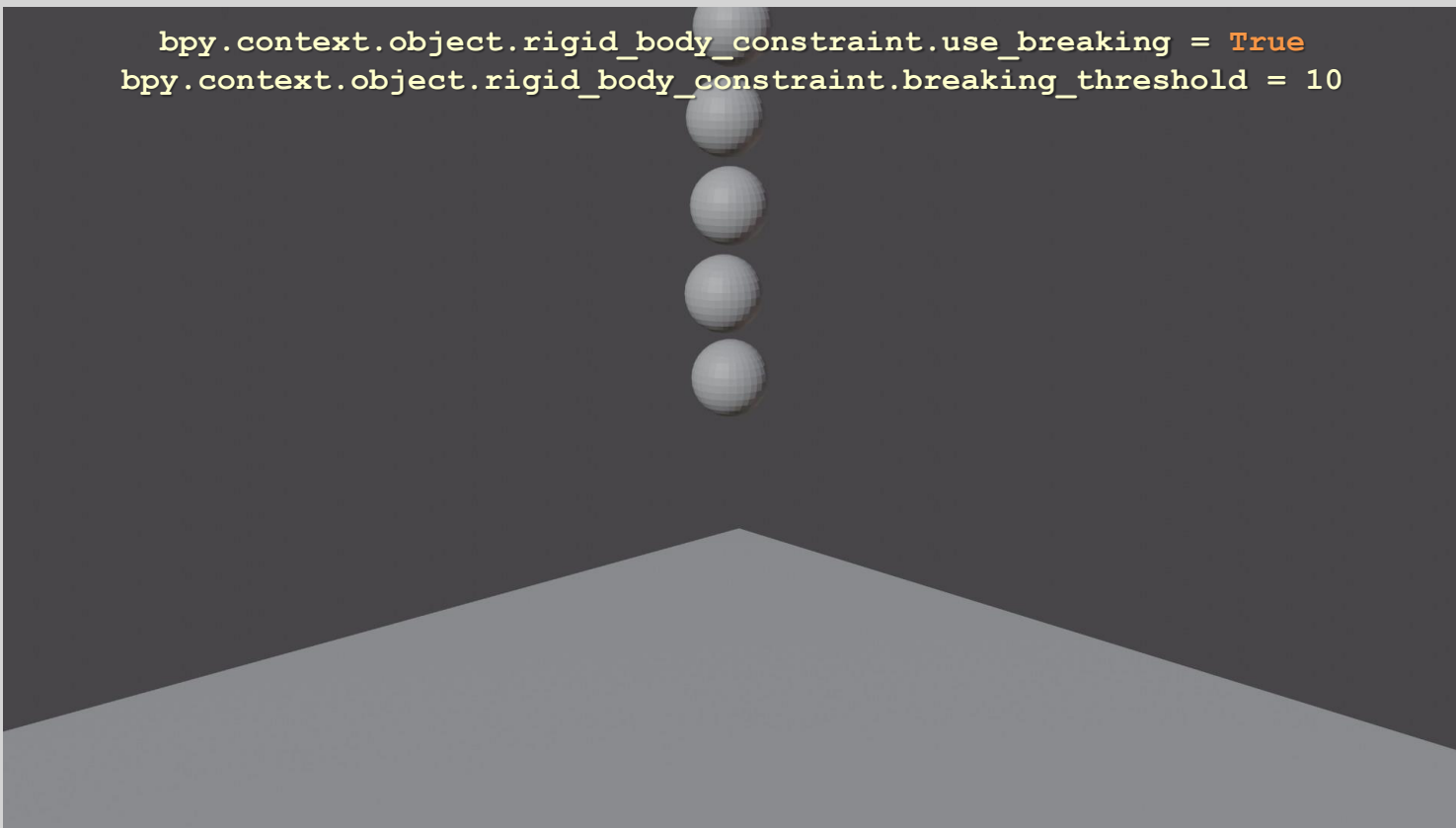




Blender/Python API Blender's Physics

Point

```
bpy.context.object.rigid_body_constraint.use_breaking = True  
bpy.context.object.rigid_body_constraint.breaking_threshold = 10
```

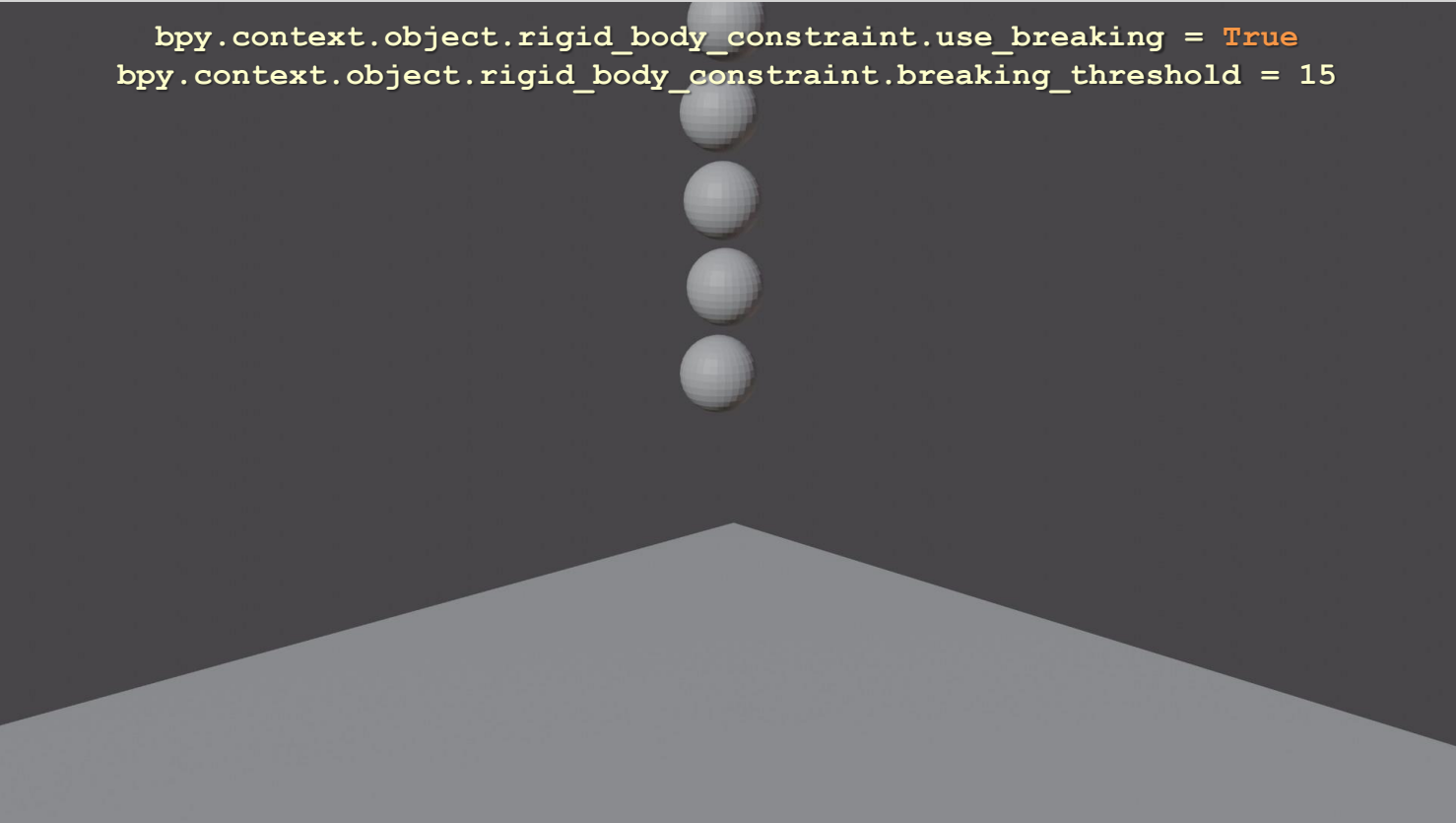




Blender/Python API Blender's Physics

Point

```
bpy.context.object.rigid_body_constraint.use_breaking = True  
bpy.context.object.rigid_body_constraint.breaking_threshold = 15
```

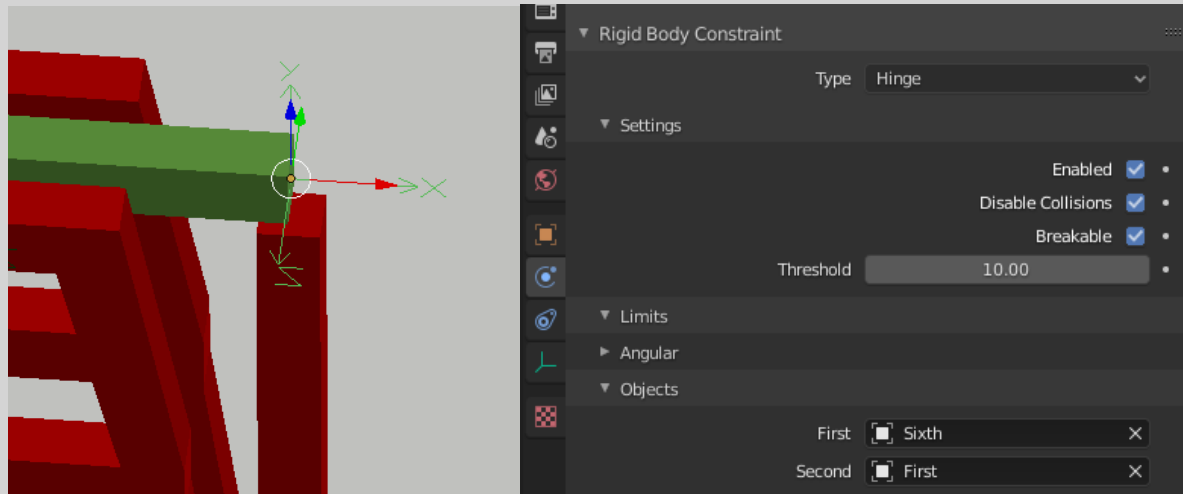
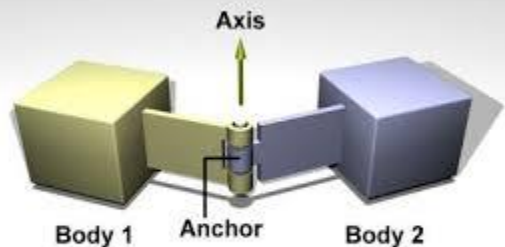


Blender/Python API

Blender's Physics

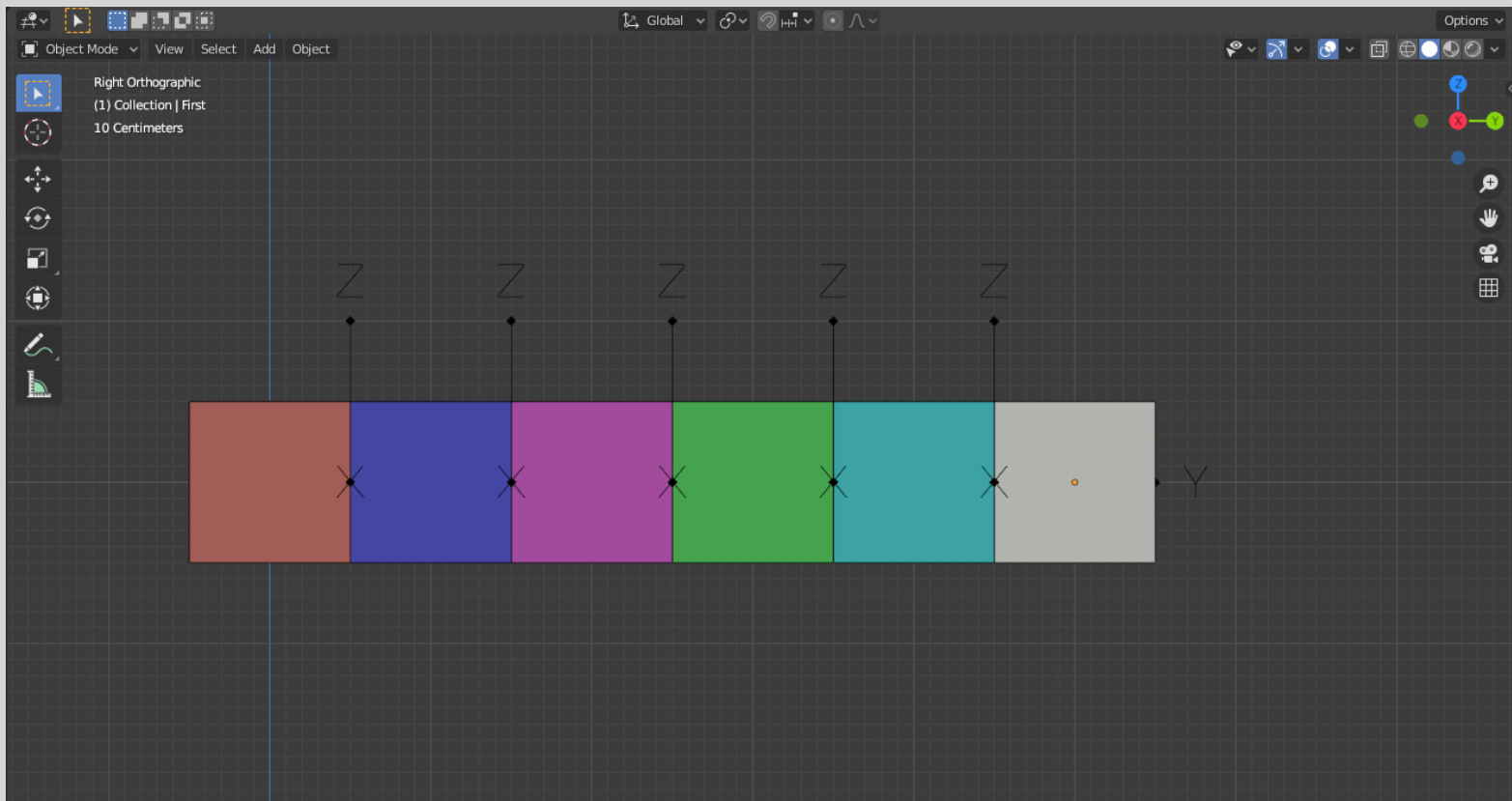
Constraints types: Fixed, Point, **Hinge**, Slider, Piston, Generic, Generic Spring and Motor

Hinge: The hinge permits 1 degree of freedom between two objects. Translation is completely constrained. Rotation is permitted about the Z axis of the object hosting the Physics constraint (usually an Empty, distinct from the two objects that are being linked).





Blender/Python API Blender's Physics





Blender/Python API

Blender's Physics

```
bpy.context.object.rigid_body_constraint.use_breaking = False
```



Blender/Python API

Blender's Physics

```
bpy.context.object.rigid_body_constraint.use_breaking = True  
bpy.context.object.rigid_body_constraint.breaking_threshold = 1
```



Blender/Python API

Blender's Physics

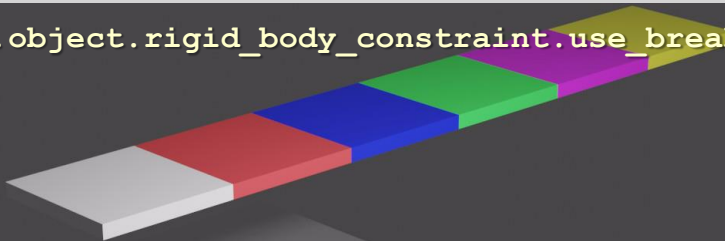
```
bpy.context.object.rigid_body_constraint.use_breaking = True  
bpy.context.object.rigid_body_constraint.breaking_threshold = 10
```



Blender/Python API

Blender's Physics

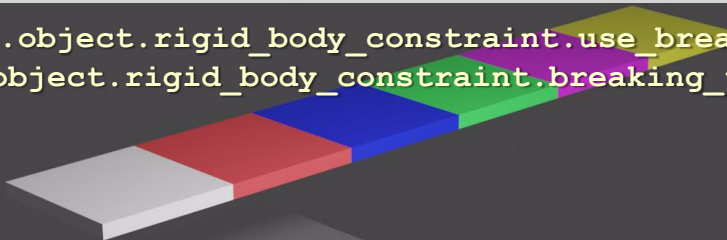
```
bpy.context.object.rigid_body_constraint.use_breaking = False
```





Blender/Python API Blender's Physics

```
bpy.context.object.rigid_body_constraint.use_breaking = True  
bpy.context.object.rigid_body_constraint.breaking_threshold = 1
```



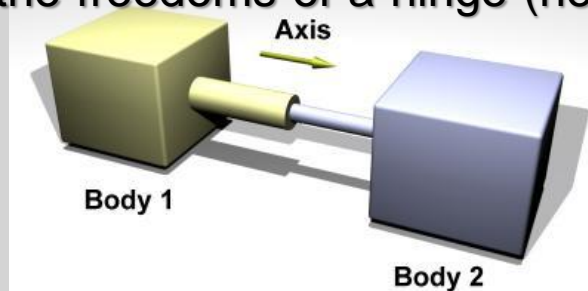
Blender/Python API

Blender's Physics

Constraints types: Fixed, Point, Hinge, **Slider**, **Piston**, Generic, Generic Spring and Motor

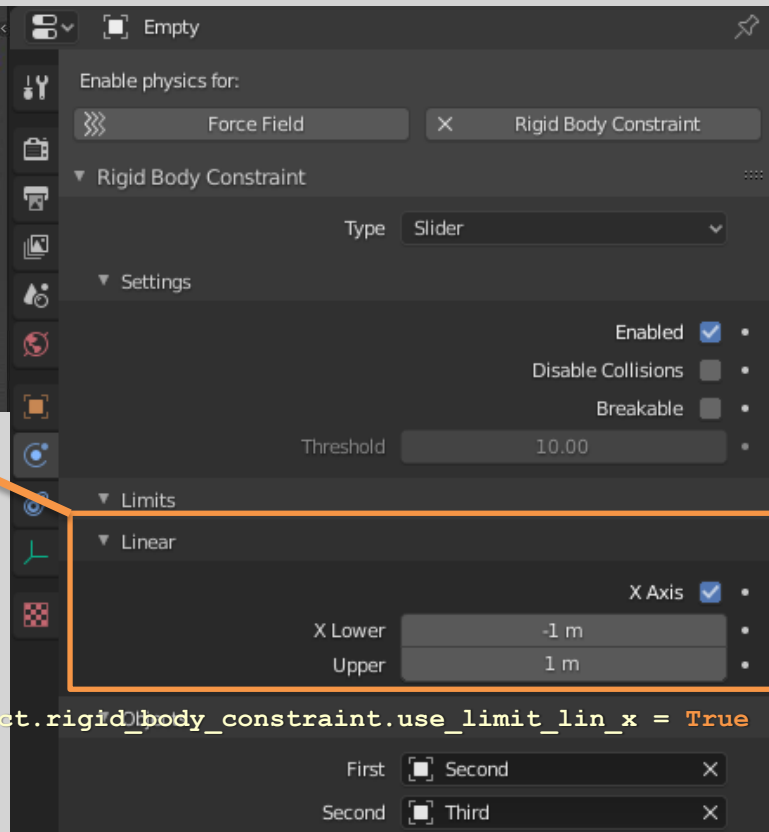
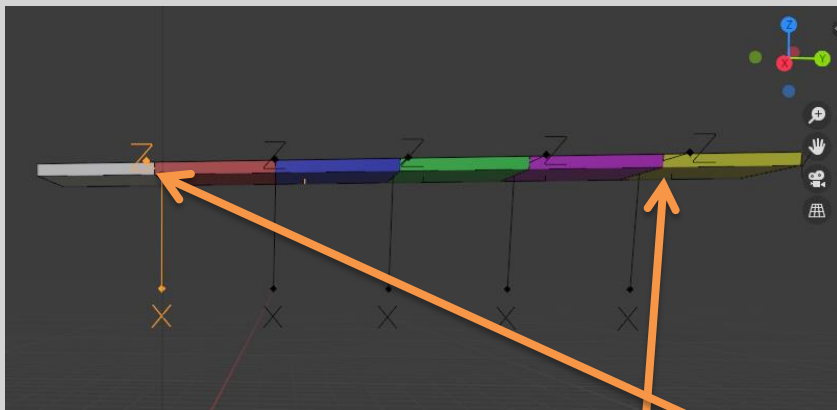
Slider: The Slider constraint allows relative translation along the X axis of the constraint object, but permits no relative rotation, or relative translation along other axes.

Piston : A piston permits translation along the X axis of the constraint object. It also allows rotation around the X axis of the constraint object. It is like a combination of the freedoms of a slider with the freedoms of a hinge (neither of which is very free alone).





Blender/Python API Blender's Physics



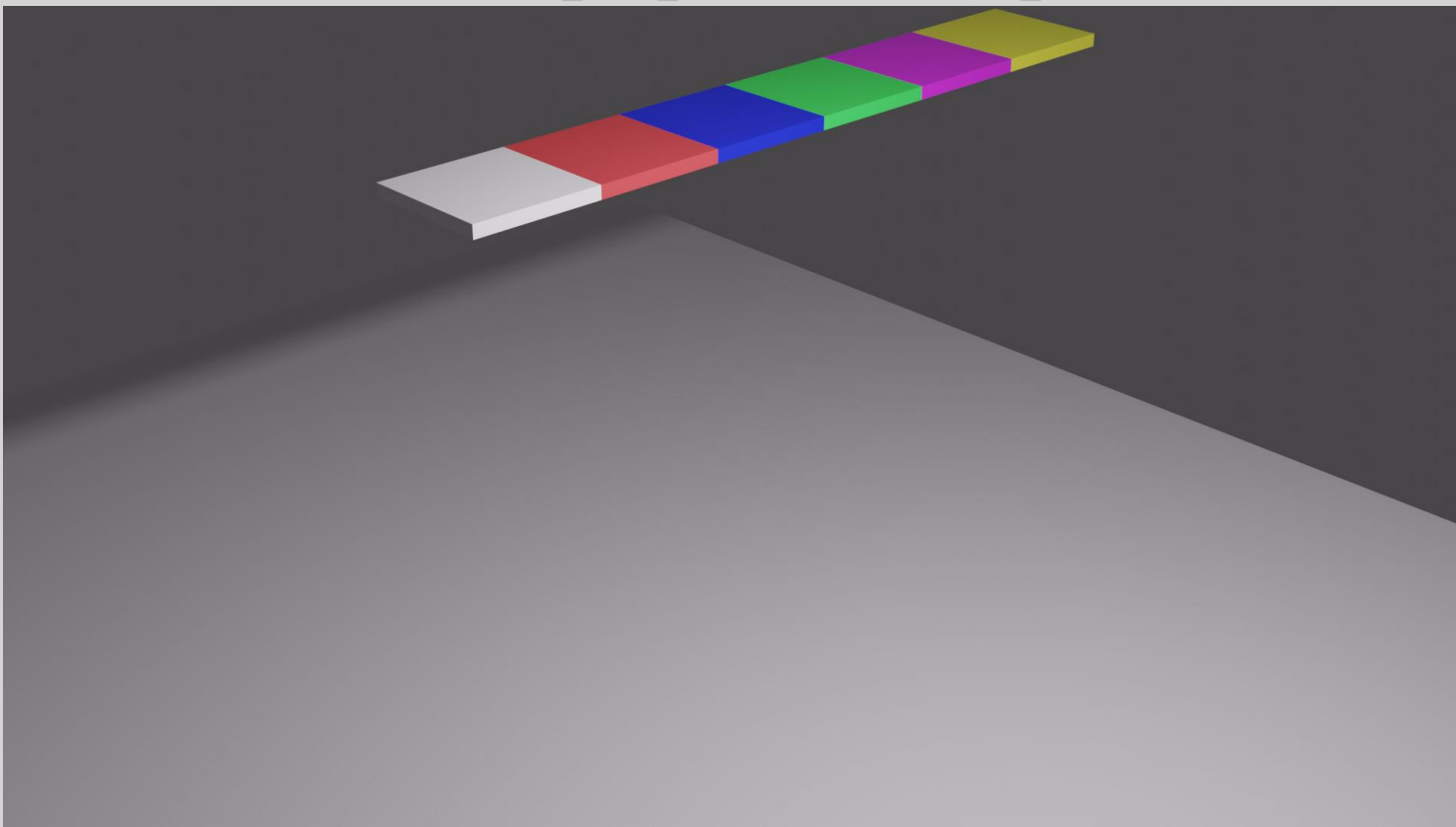
```
bpy.context.object.rigid_body_constraint.disable_collisions = True
```

```
bpy.context.object.rigid_body_constraint.use_limit_lin_x = True
```



Blender/Python API Blender's Physics

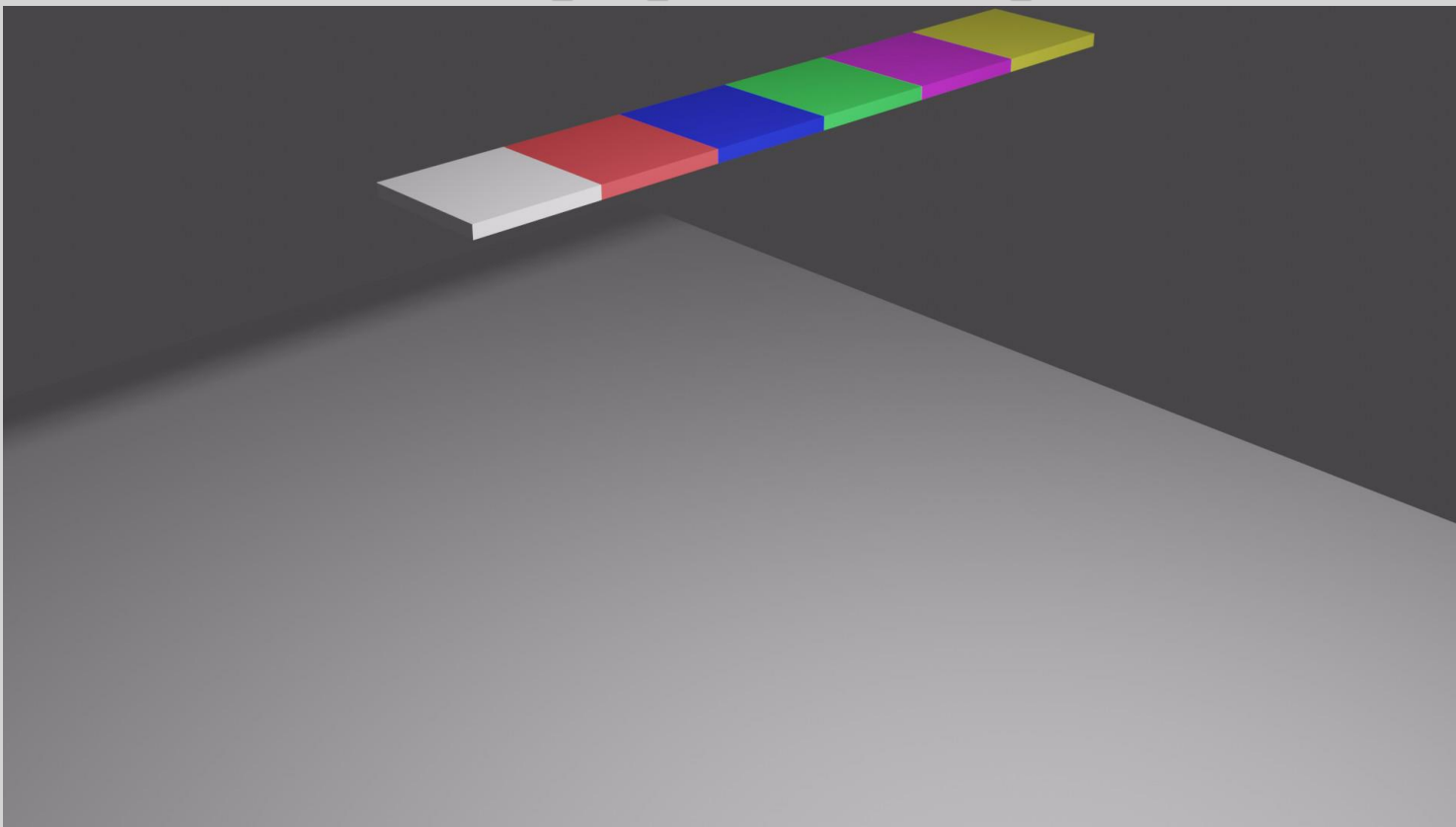
```
bpy.context.object.rigid_body_constraint.disable_collisions = True
```





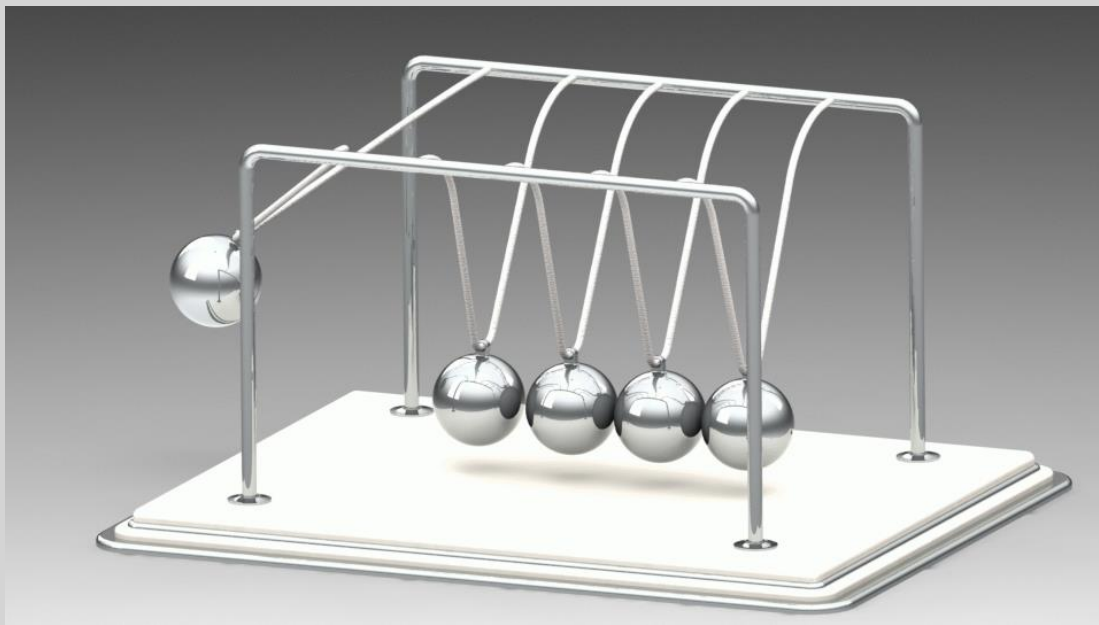
Blender/Python API Blender's Physics

```
bpy.context.object.rigid_body_constraint.disable_collisions = False
```





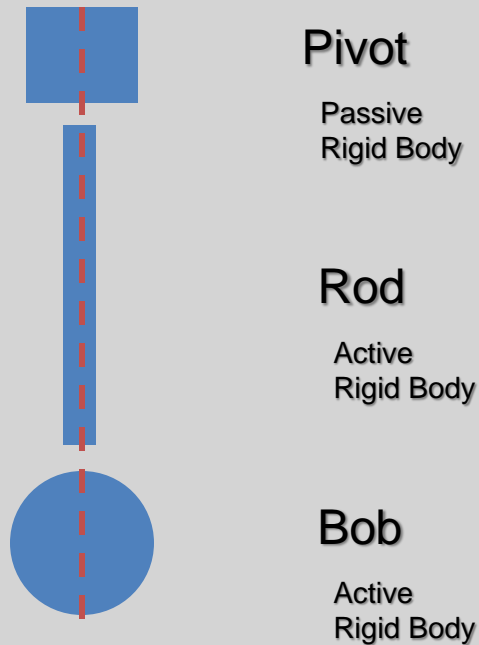
Blender/Python API Blender's Physics



A Newton's cradle is a device that demonstrates conservation of momentum and energy using a series of swinging spheres.

Blender/Python API

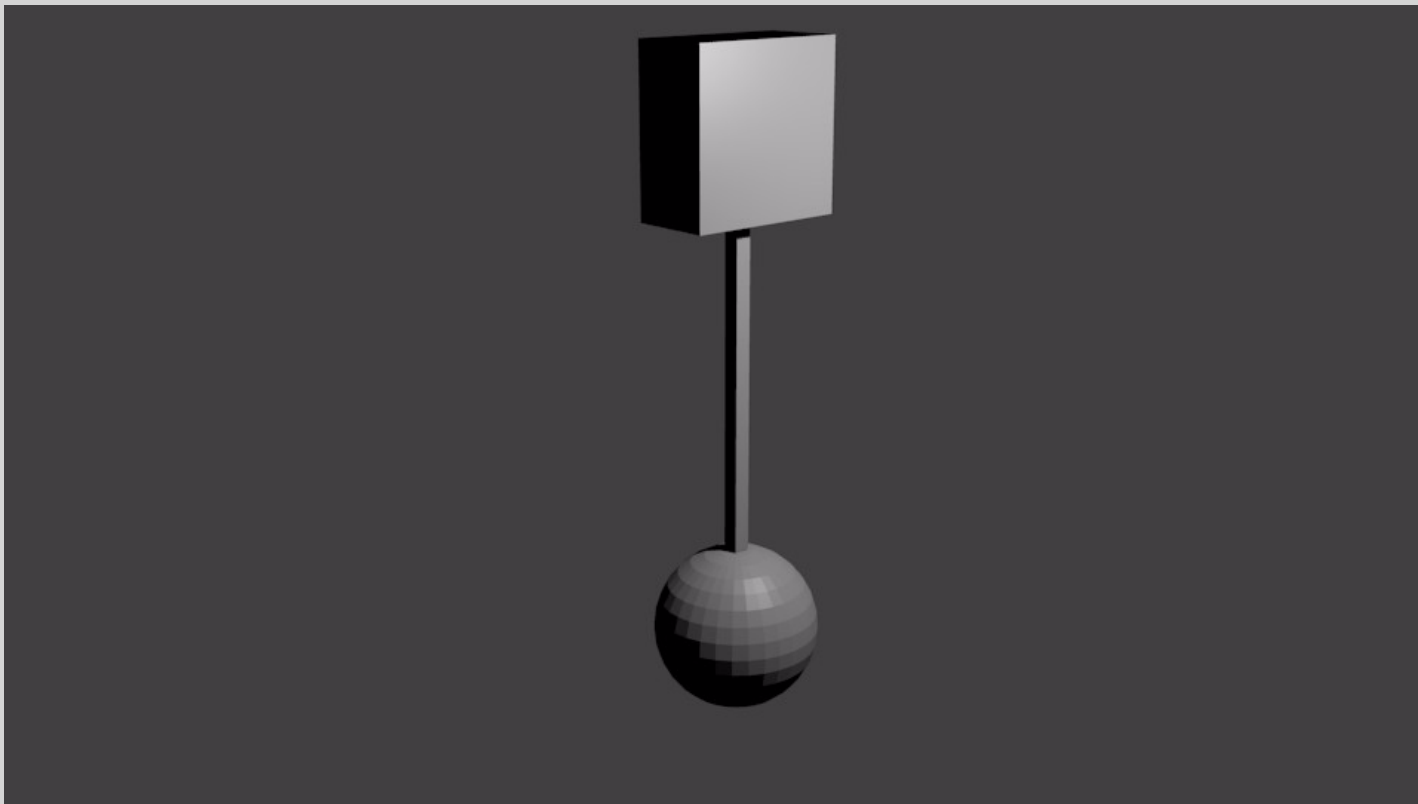
Blender's Physics



A Newton's cradle is a device that demonstrates conservation of momentum and energy using a series of swinging spheres.

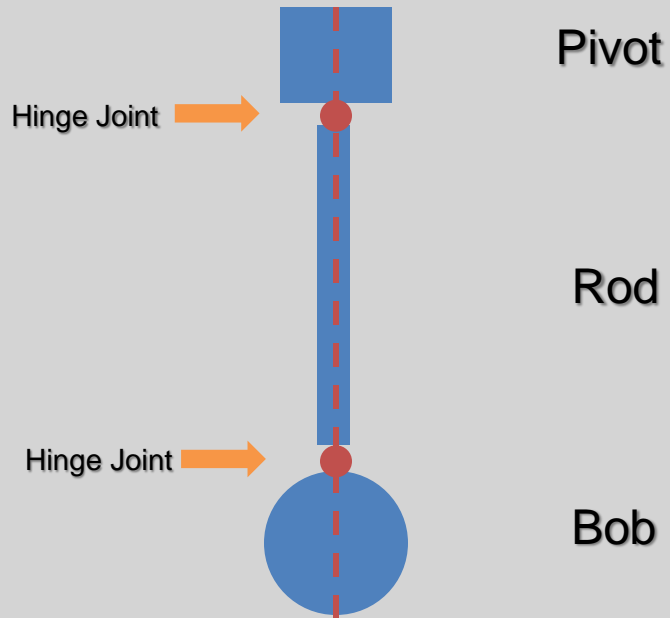
Blender/Python API

Blender's Physics



Blender/Python API

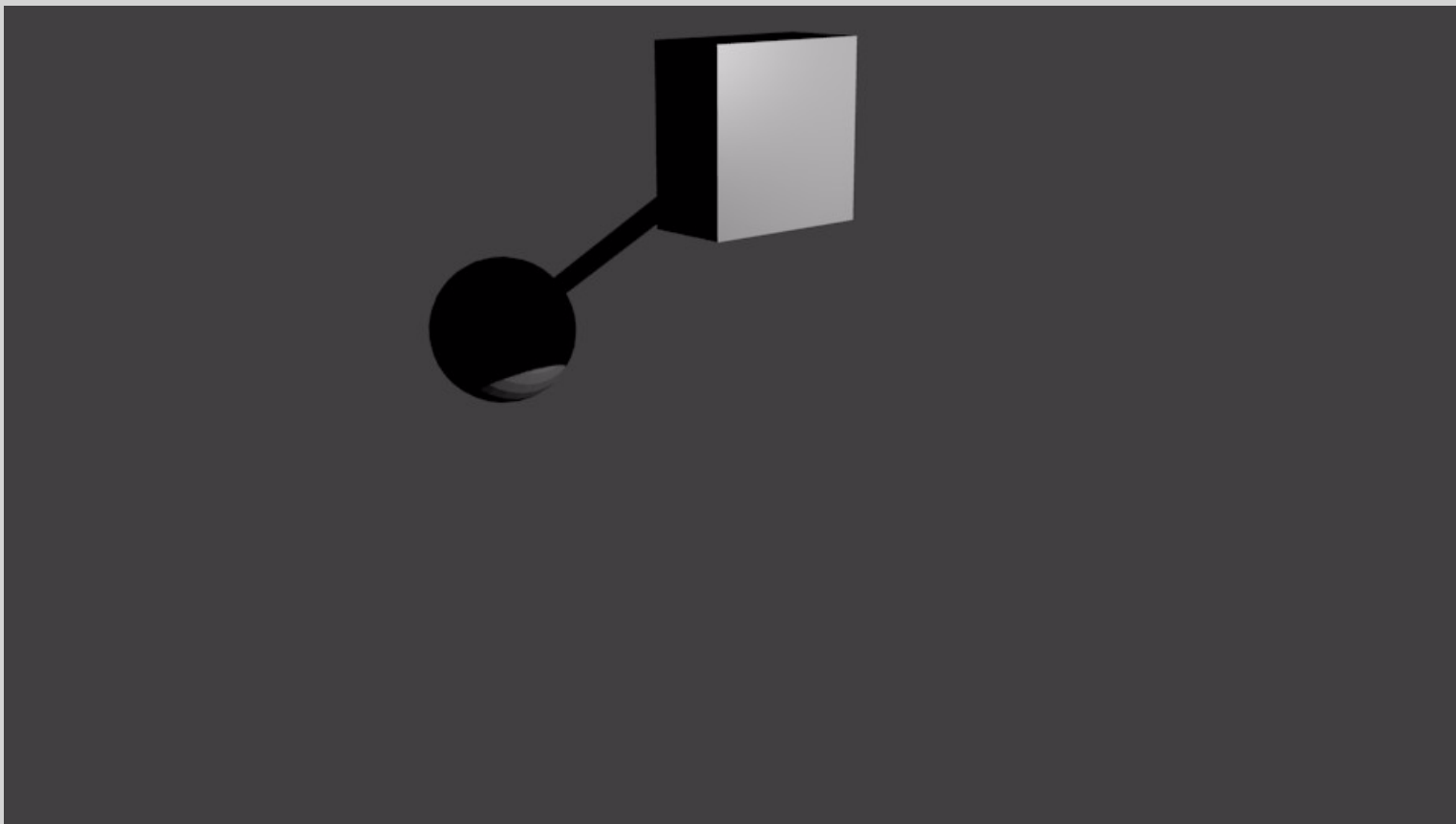
Blender's Physics





Blender/Python API

Blender's Physics



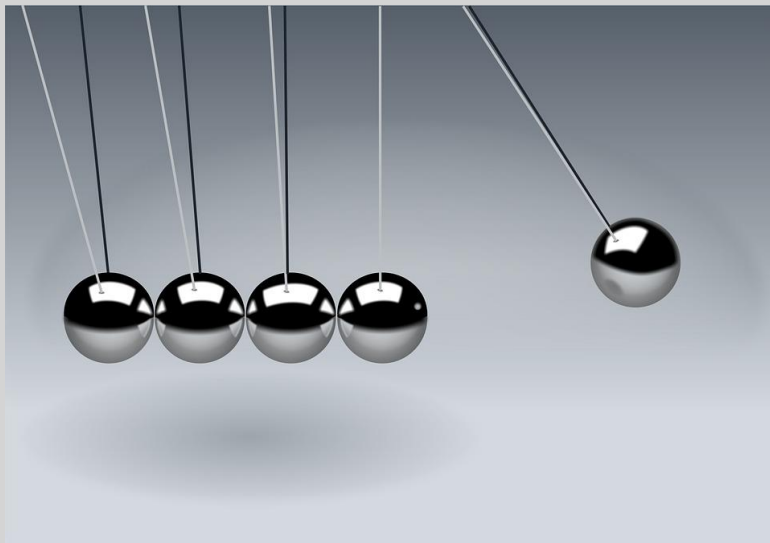


Blender/Python API Blender's Physics

Homework : Design a function that creates one piece of pendulum.

```
def createPendulum(location, rodLength, rodMass, bobSize, bobMass, ..., color..etc)
```

Then, create a Newton's Cradle animation by using the function you design.





Blender/Python API

Blender's Physics

Constraints types: Fixed, Point, Hinge, Slider, Piston, **Generic**, **Generic Spring** and Motor

Generic: The generic constraint has a lot of available parameters. The X, Y, and Z axis constraints can be used to limit the amount of translation between the objects. Clamping the min/max to zero has the same effect as the Point constraint. Clamping the relative rotation to zero keeps the objects in alignment. Combining an absolute rotation and translation clamp would behave much like the Fixed constraint.

Generic Spring : The generic spring constraint adds some spring parameters for the X/Y/Z axes to all the options available on the Generic constraint. Using the spring alone allows the objects to bounce around as if attached with a spring anchored at the constraint object.

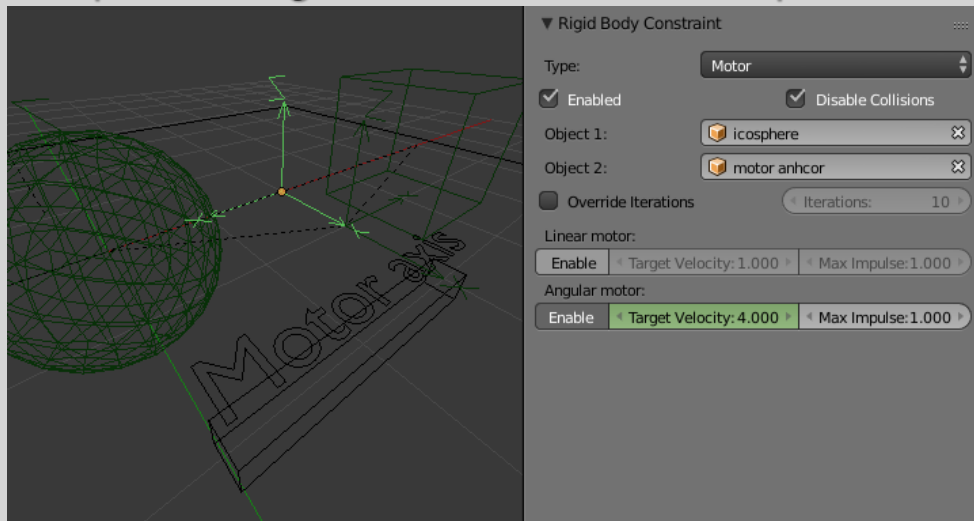
Blender/Python API

Blender's Physics

Constraints types: Fixed, Point, Hinge, Slider, Piston, Generic, Generic Spring and **Motor**

Motor: The motor constraint causes translation and/or rotation between two entities. It can drive two objects apart or together. It can drive simple rotation, or rotation and translation.

The rotation axis is the X axis of the object hosting the constraint. This is in contrast with the Hinge which uses the Z axis.





Blender/Python API Handlers Overview

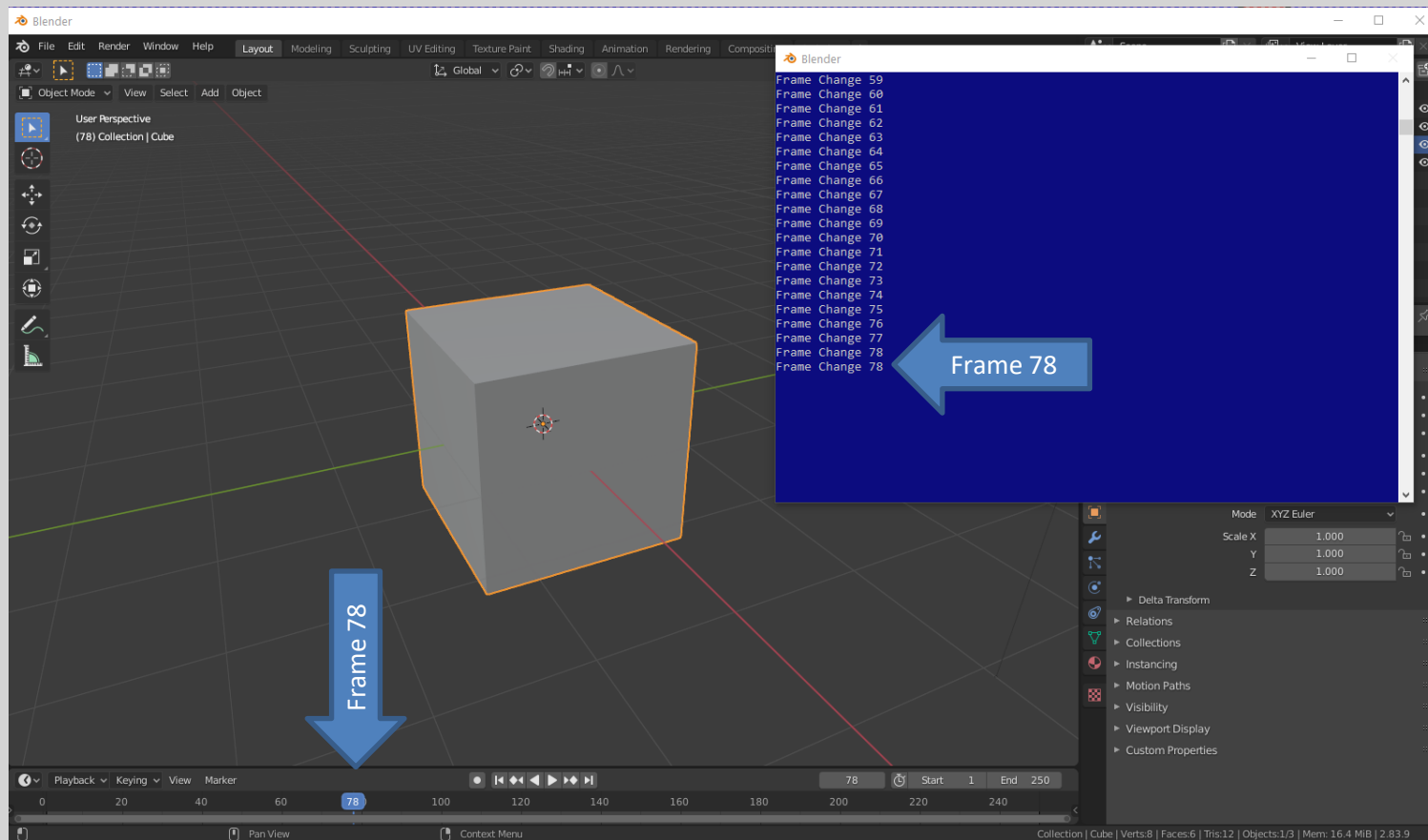
Handlers are functions that are set to run every time an event occurs. To instantiate a handler, we declare a function, then add it to one of the possible lists of handlers in Blender.

```
import bpy
```

```
def my_handler(scene):  
    print("Frame Change", scene.frame_current)
```

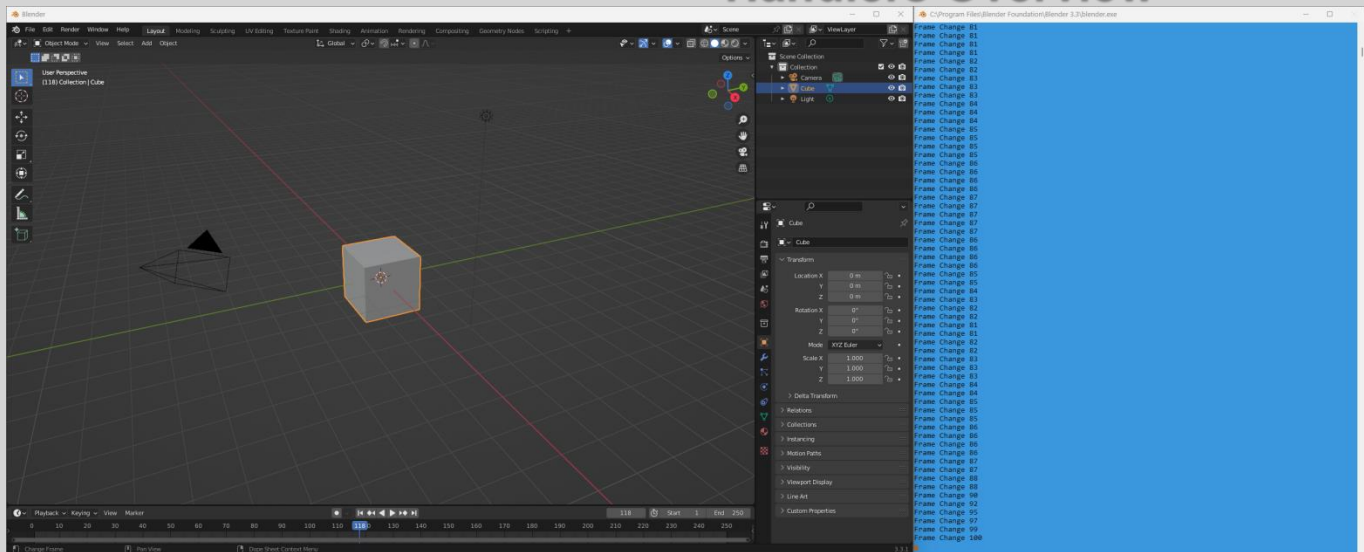
```
bpy.app.handlers.frame_change_pre.append(my_handler)
```

Blender/Python API Handlers Overview





Blender/Python API Handlers Overview



```
import bpy
```

```
def my_handler(scene):  
    print("Frame Change", scene.frame_current)  
    if scene.frame_current >= 100:  
        bpy.app.handlers.frame_change_pre.remove(my_handler)  
  
bpy.app.handlers.frame_change_pre.append(my_handler)
```



Blender/Python API Handlers Overview

By default handlers are freed when loading new files, in some cases you may want the handler stay running across multiple files (when the handler is part of an add-on for example). For this the `bpy.app.handlers.persistent` decorator needs to be used.

```
import bpy
from bpy.app.handlers import persistent

@persistent
def load_handler(dummy):
    print("Load Handler:", bpy.data.filepath)

bpy.app.handlers.load_post.append(load_handler)
```



Decorators are a way to modify the behavior of an object by wrapping it within a function.

```
def my_function():  
    print("Function called")
```

```
print("Script start")  
my_function()  
print("Script end")
```

```
----- RUN -----  
Script start  
Function called  
Script end
```



Blender/Python API Decorators Overview



A **decorator** is a callable that takes **another function** as **argument** (the decorated function).

```
def my_decorator(fn):  
    def wrapper():  
        print("entering the function...")  
        fn()  
        print("exiting the function...")  
    return wrapper
```

```
@my_decorator  
def my_function():  
    print("inside the function...")
```

my_function()

run

entering the function...
inside the function...
exiting the function...



Decorators can also be stacked

```
def decorator_1(fn):  
    def wrapper():  
        print("entering the decorator 1...")  
        fn()  
        print("exiting the decorator 1...")  
    return wrapper  
  
def decorator_2(fn):  
    def wrapper():  
        print("entering the decorator 2...")  
        fn()  
        print("exiting the decorator 2...")  
    return wrapper  
  
def decorator_3(fn):  
    def wrapper():  
        print("entering the decorator 3...")  
        fn()  
        print("exiting the decorator 3...")  
    return wrapper
```



Blender/Python API Decorators Overview



```
@decorator_1
@decorator_2
@decorator_3
def my_function():
    print("inside the function.")
```

```
>>> my_function()
entering decorator 1...
entering decorator 2...
entering decorator 3...
inside the function
exiting decorator 3...
exiting decorator 2...
exiting decorator 1...
```



Blender/Python API Decorators Overview



It's important to note that **decorators** are a **syntactic sugar**, meaning they make a particular design pattern more elegant, but they **do not add any additional functionality** to the language.

```
def dec(fn):  
    def wrapper():  
        fn()  
    return wrapper
```

syntactic sugar, because this...

```
@dec  
def fn():  
    pass
```

is functionally equivalent to this.

```
fn = dec(fn)
```



```
# adding decorator to the functions
```

```
@myLog_decorator
```

```
def twoSum(x, y):
```

```
    return x + y
```

```
@myLog_decorator
```

```
def twoMultiply(x, y):
```

```
    return x * y
```

```
a, b = 5, 3
```

```
# getting the value through return of the function
```

```
print("Sum =", twoSum(a, b))
```

```
twoMultiply(a, b)
```



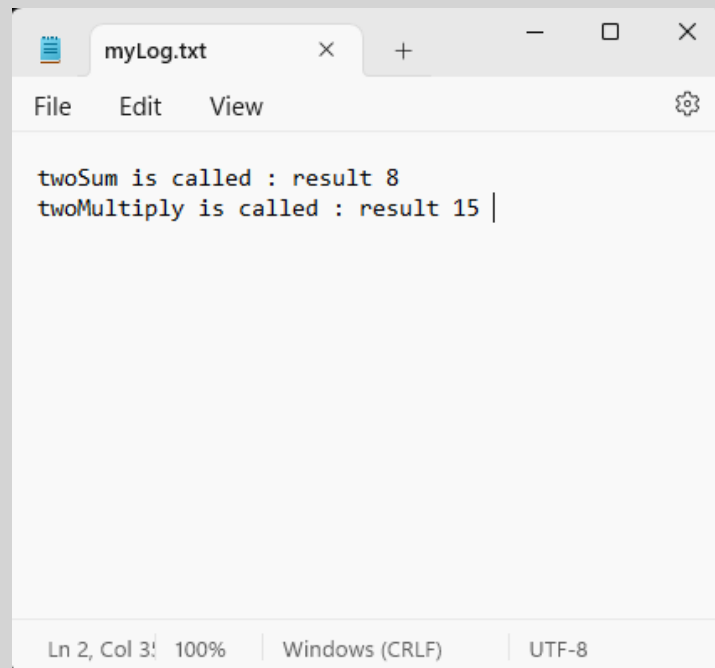

```
def myLog_decorator(func):  
    def inner(*args, **kwargs):  
  
        print("1 : before Execution")  
  
        # getting the returned value  
        returned_value = func(*args, **kwargs)  
        with open('myLog.txt', 'a') as f:  
            f.write(func.__name__ +  
                    ' is called : result ' +  
                    str(returned_value) + ' \n')  
  
        print("2 : after Execution")  
  
        # returning the value to the original frame  
        return returned_value  
  
    return inner
```



```
1 : before Execution
2 : after Execution
Sum = 8
1 : before Execution
2 : after Execution
```

Blender/Python API

Decorators Overview

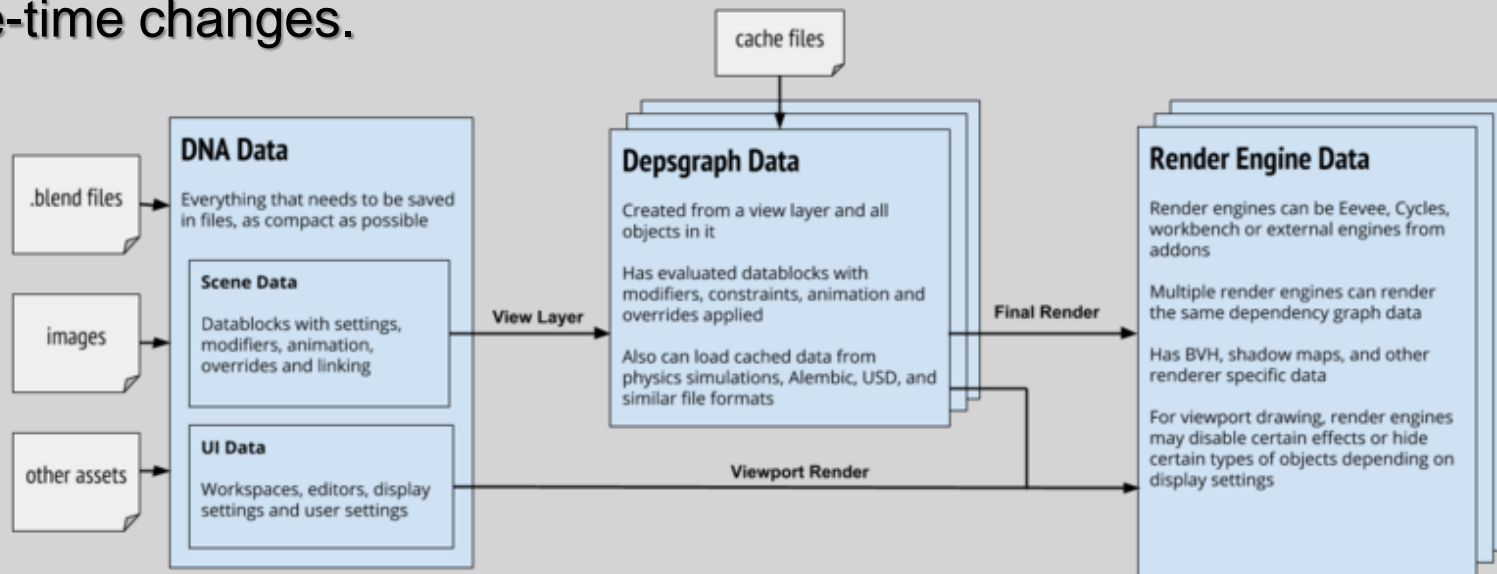


```
twoSum is called : result 8
twoMultiply is called : result 15 |
```



Blender/Python API Handlers Overview

The dependency graph is responsible for dynamic updates of the scene, where some value varies over time. It is not responsible for one-time changes.





Blender/Python API Handlers Overview

In the example, we create a function that modifies the text of a text mesh with the current time. We then add the function to `bpy.app.handlers.depsgraph_update_pre`. to indicate that we would like it to run right before the 3D Viewport is updated and displayed.

The result is what appears to be a clock in the 3D Viewport. In actuality, it is a text mesh that is updating many times per second. This example is not safe or full-proof, but as long as we keep the object in the scene and named `MyTextObj`, we can add and edit other objects with the clock running in the background.



Blender/Python API Handlers Overview

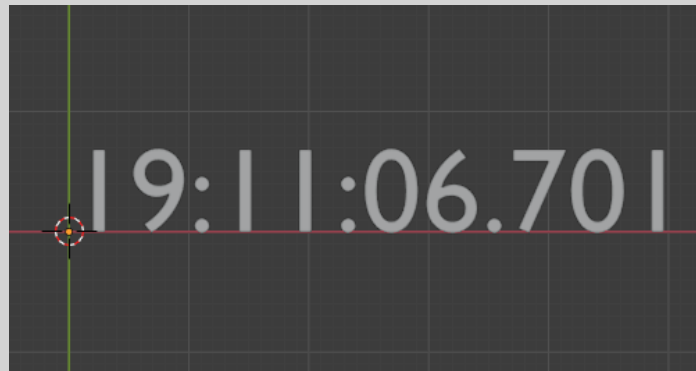
```
import bpy
import datetime

# Clear the scene
bpy.ops.object.select_all(action='SELECT')
bpy.ops.object.delete()

# Create an object for our clock
bpy.ops.object.text_add(location=(0, 0, 0))
bpy.context.object.name = 'MyTextObj'

# Create a handler function
def tell_time():
    current_time = datetime.datetime.now().strftime('%H:%M:%S.%f')[:-3]
    bpy.data.objects['MyTextObj'].data.body = current_time

# Add to the list of handler functions "depsgraph_update_pre"
bpy.app.handlers.depsgraph_update_pre.append(tell_time)
```





Blender/Python API Handlers Overview

The behavior of the clock is not a documented behavior and may change with future releases of blender. specifically, blender intends to change what they qualify as a frame change. Currently, frame changes seem to happen instantaneously and constantly.

The official blender documentation gives examples where the only parameter passed to the handler is a **dummy**. **handler functions** should be treated like traditional python **lambdas**, with the exception that a single dummy argument is required as the first parameter. We pass the function itself rather than the output of the function, and a new unnamed instance of the function is created when it is passed. We cannot easily access this unnamed function after it is created for the handler.



Blender/Python API Managing Handler Lists

In the case of `bpy.app.handlers`, we can edit various lists of functions to manage our handlers. These lists are quite literally Python classes of type `list`, and we can operate on them as such. We can use list class methods such as `append()`, `pop()`, `remove()`, and `clear()` to manage our handler functions.

```
# Will only work if 'tell_time' is in scope
bpy.app.handlers.depsgraph_update_pre.remove(tell_time)
# Useful in development for a clean state
bpy.app.handlers.depsgraph_update_pre.clear()
# Remove handler at the end of the list and return it
bpy.app.handlers.depsgraph_update_pre.pop()
```


Blender/Python API

Types of Handlers

Handler

bpy.app.handlers.depsgraph_update_post
bpy.app.handlers.depsgraph_update_pre
bpy.app.handlers.frame_change_post
bpy.app.handlers.frame_change_pre
bpy.app.handlers.load_factory_preferences_post
bpy.app.handlers.load_factory_startup_post
bpy.app.handlers.load_post
bpy.app.handlers.load_pre
bpy.app.handlers.redo_post
bpy.app.handlers.redo_pre
bpy.app.handlers.render_cancel
bpy.app.handlers.render_complete
bpy.app.handlers.render_init
bpy.app.handlers.render_post
bpy.app.handlers.render_pre
bpy.app.handlers.render_stats
bpy.app.handlers.render_write
bpy.app.handlers.save_post
bpy.app.handlers.save_pre
bpy.app.handlers.undo_post
bpy.app.handlers.undo_pre
bpy.app.handlers.version_update
bpy.app.handlers.persistent

Called On

on depsgraph update (post)
on depsgraph update (pre)
Called after frame change for playback and rendering, after the data has been evaluated for the new frame.
Called after frame change for playback and rendering, before any data is evaluated for the new frame.
on loading factory preferences (after)
on loading factory startup (after)
on loading a new blend file (after)
on loading a new blend file (before)
on loading a redo step (after)
on loading a redo step (before)
on canceling a render job
on completion of render job
on initialization of a render job
on render (after)
on render (before)
on printing render statistics
on writing a render frame (directly after the frame is written)
on saving a blend file (after)
on saving a blend file (before)
on loading an undo step (after)
on loading an undo step (before)
on ending the versioning code
Function decorator for callback functions not to be removed when loading new files



Blender/Python API

Display Current Frame

Assuming your text object is called 'MyTextObj', this will have it read the current frame:

```
import bpy
# Clear the scene
bpy.ops.object.select_all(action='SELECT')
bpy.ops.object.delete()
# Create an object for frame
bpy.ops.object.text_add(location=(0, 0, 0))
bpy.context.object.name = 'MyTextObj'
scene = bpy.context.scene
obj = scene.objects['MyTextObj']

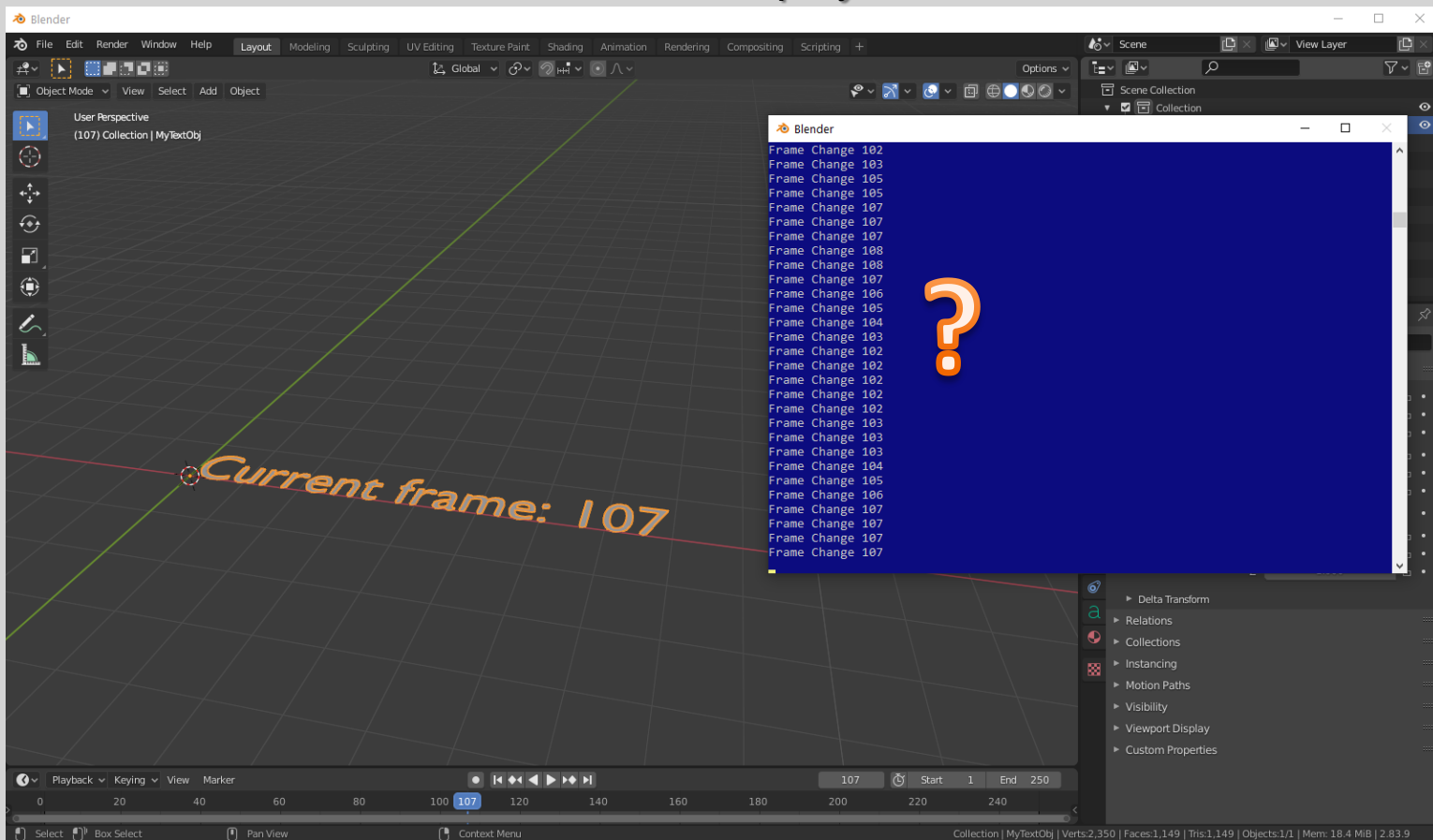
def recalculate_text(scene):
    obj.data.body = 'Current frame: ' + str(scene.frame_current)

bpy.app.handlers.frame_change_pre.append(recalculate_text)
```

The last line just causes the `recalculate_text` function to be run each time the frame is changed

Blender/Python API

Display Current Frame





Run a Function in x Seconds:

```
import bpy
```

```
def in_5_seconds():  
    print("Hello World")
```

```
bpy.app.timers.register(in_5_seconds, first_interval=5)
```

Run a Function every x Seconds

```
import bpy
```

```
def every_2_seconds():  
    print("Hello World")  
    return 2.0
```

```
bpy.app.timers.register(every_2_seconds)
```