

Blender - Python API

#12

Serdar ARITAN

Department of Computer Graphics
Hacettepe University, Ankara, Turkey





Blender/Python API Materials/Textures

Material and Textures

Materials and Textures add color and define the surfaces of Objects. The application of Materials and Textures in Blender utilises a graphical display called a **Node System** which represents the computer code generating the effect of Material and Texture. In Computer Graphics, how the surface of an Object displays is determined by three factors;

Material, Texture and Lighting.

Material: The Base Color of the surface.

Texture: The physical characteristics of the surface.

Lighting: The background illumination or light emitting from Lights.



Blender/Python API Materials/Textures

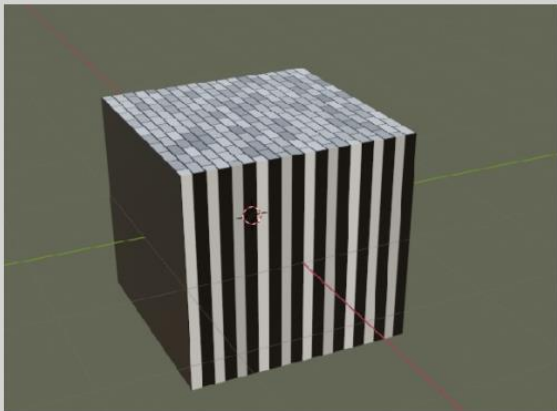
A **Material** is the color of an Object which is how the visible spectrum of light reflects from the Object's surface. A Material also defines whether the surface appears dull (matt) or shiny (metallic).

A **Texture** defines the physical appearance of a surface such as how lumpy or bumpy it appears or its pattern which visually defines the perception of the physical make up of the Object. This definition determines what you perceive the surface to be; wood, brick, water etc.

Blender/Python API

Materials/Textures

Textures are generated by computer code built into Blender (Procedural Textures) or from images saved on the computer (Image Textures)



Procedural Textures

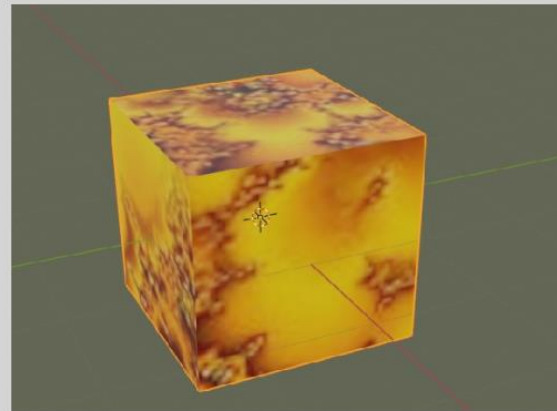
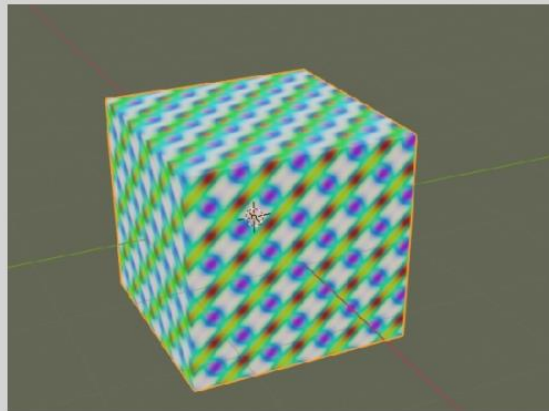
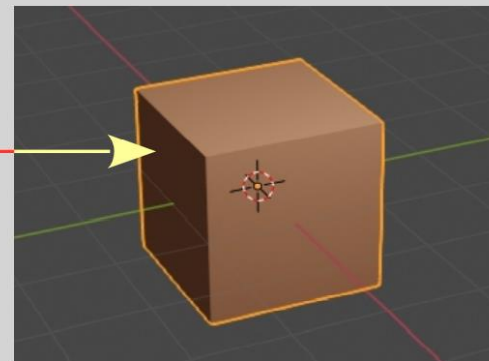
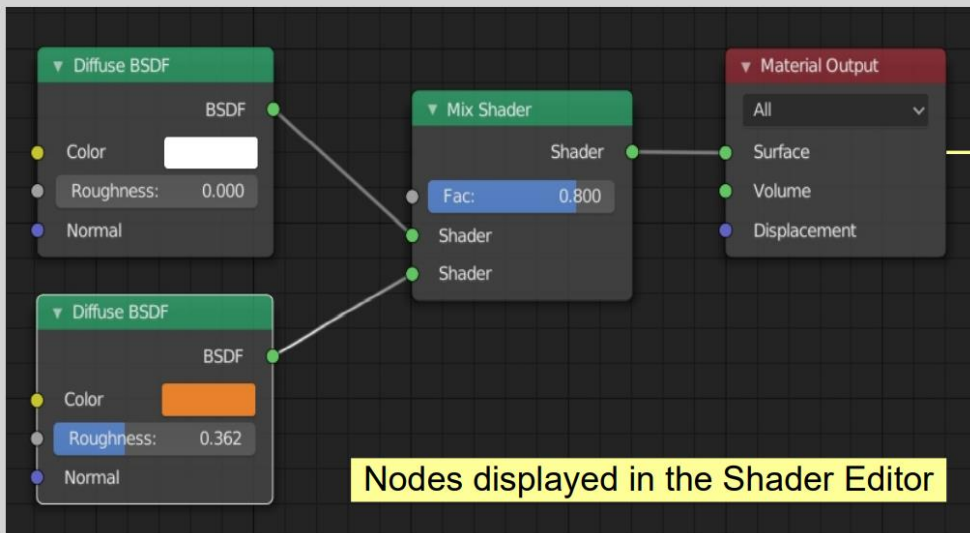


Image Texture

Blender/Python API

Materials/Textures

Nodes are graphical representations of blocks of computer data which instructs the computer how to display Materials and Texture. In Blender, Nodes are arranged in a chain or pipeline to produce effects. In the case of Materials and Texture the arrangement of Nodes create the visual affect of an Objects' surface appearance.



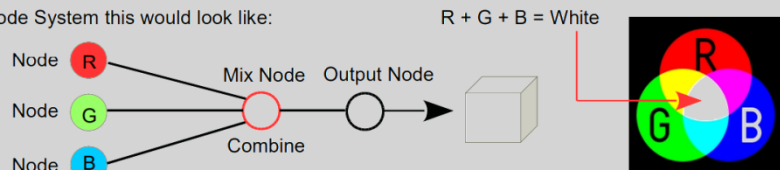
Node arrangement producing the Material (color) of the default Cube Object.

Blender/Python API

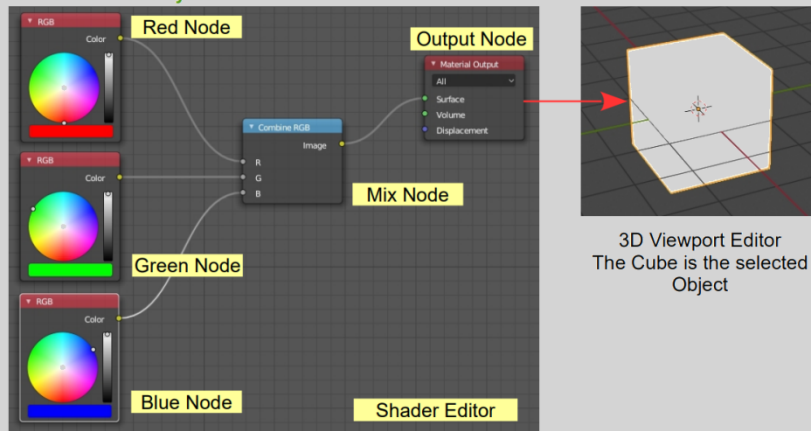
Materials Using Nodes

Material Nodes, in their basic application, allow you to add color to the surface of an Object. The Node is a graphical representation of computer data or instruction which is arranged in a pipeline. Think about mixing colors. The primary colors are Red, Green and Blue, which when mixed in equal proportions produce White

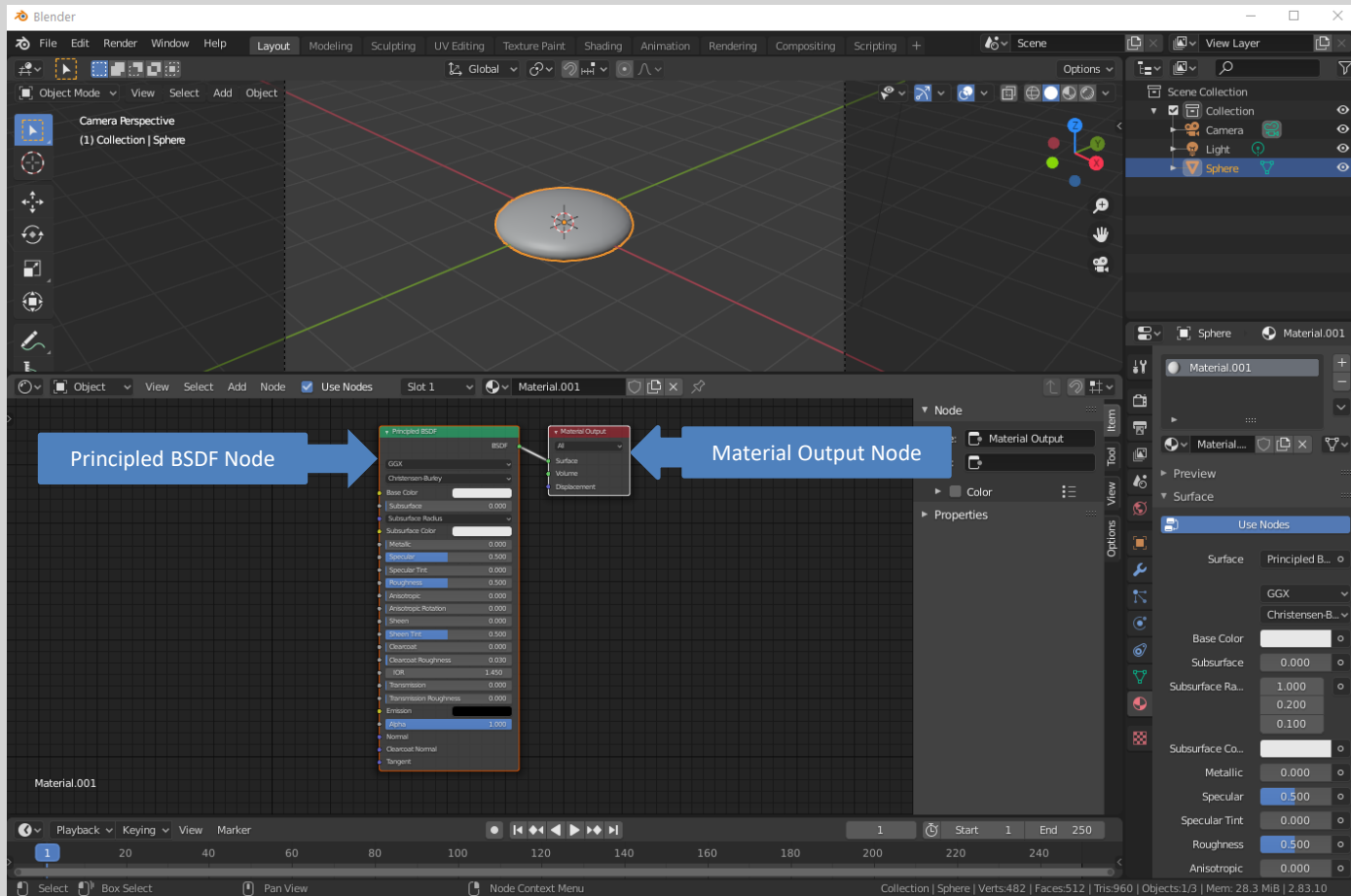
In a Node System this would look like:



The Blender Node system looks like this:

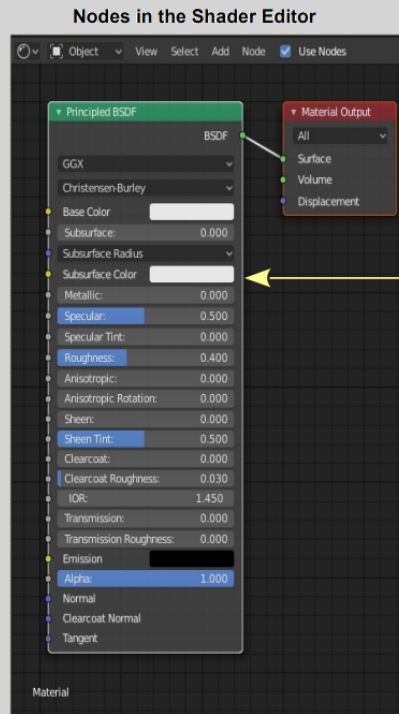


Blender/Python API Materials/Textures

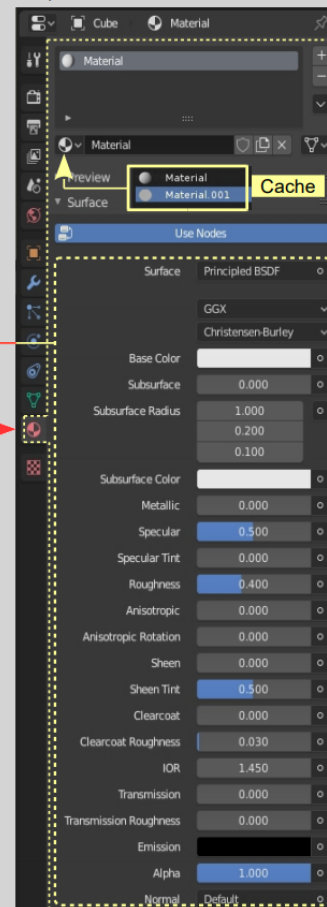


Blender/Python API Materials/Textures

The Principled BSDF Node displays **values and controls identical** to the controls in the Properties Editor, Material buttons when the Node System is active.



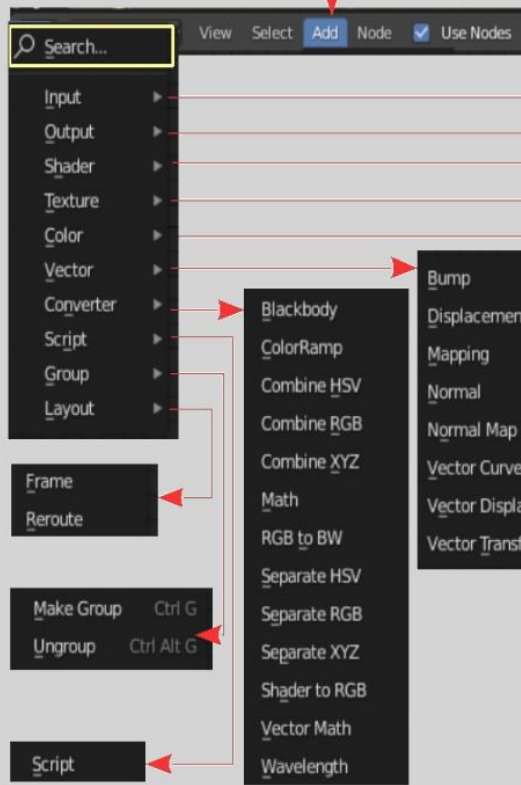
Properties Editor Material Buttons



Note: The Material Button is in the vertical column at the side of the Properties Editor.

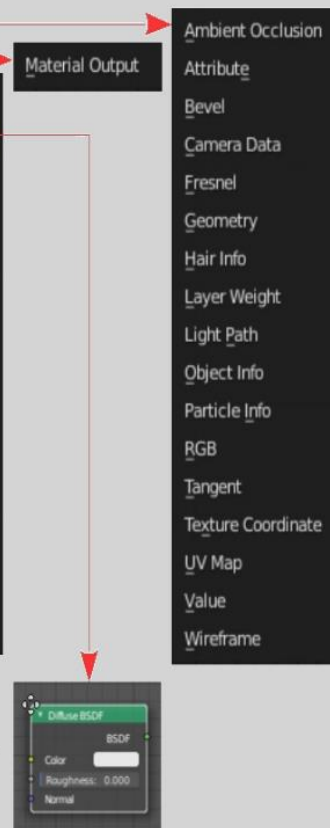
Blender/Python API Node Selection Menu

To add a Node click the **Add** button

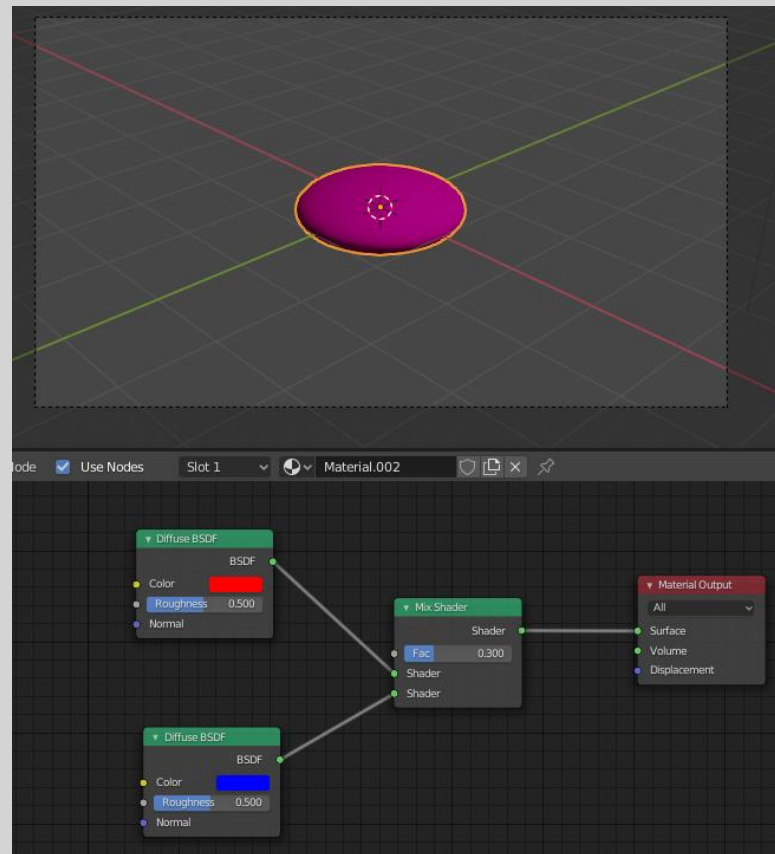
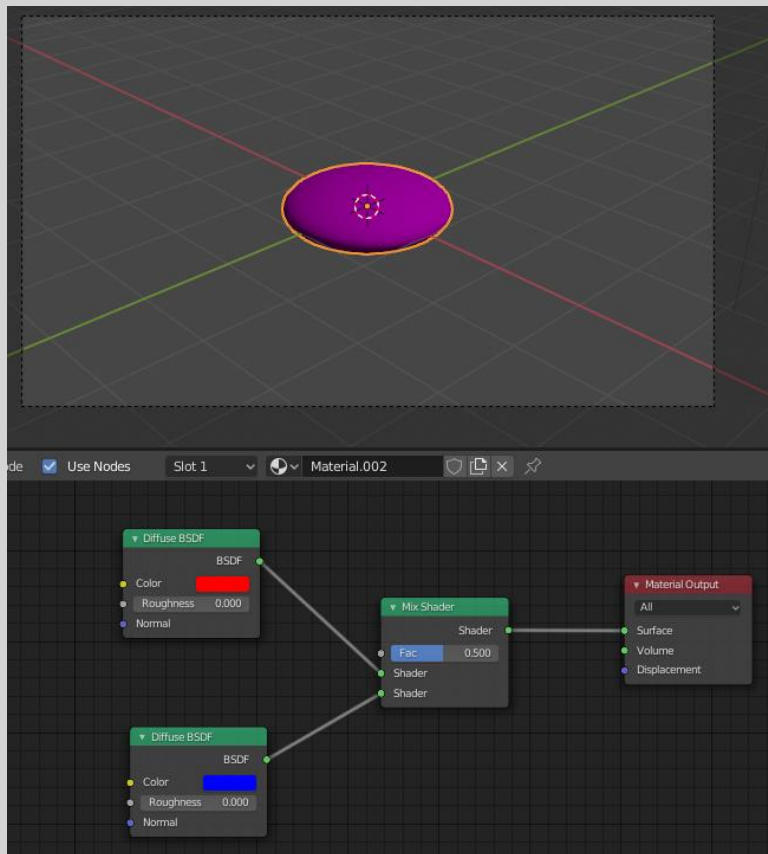


Clicking Add displays the Node Category Menu with the Search Panel at the top.
Clicking on a Category opens a Node List.

Having the Node Lists displayed assists when following tutorials and seeking to replicate Node Arrangements for particular effects.



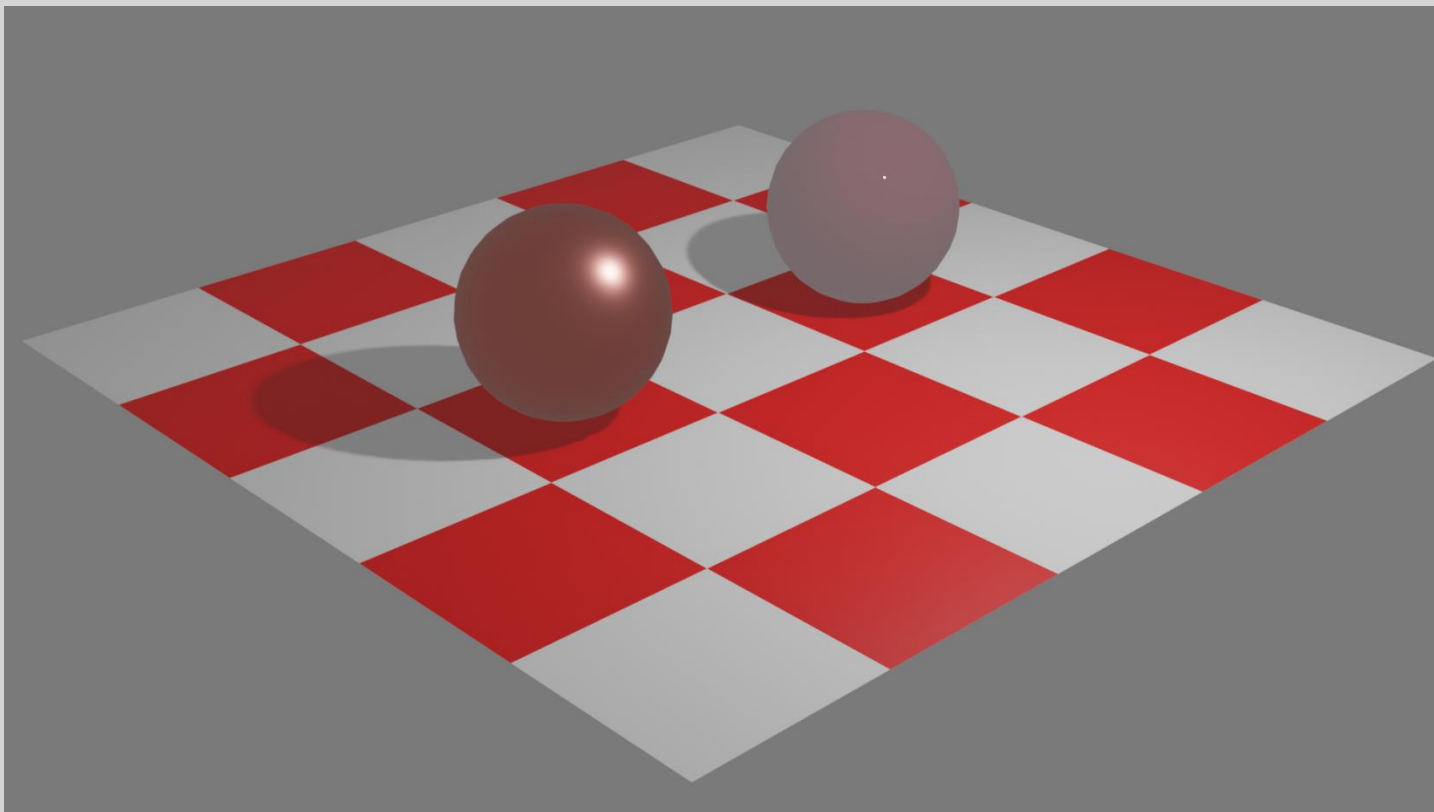
Blender/Python API Materials/Textures





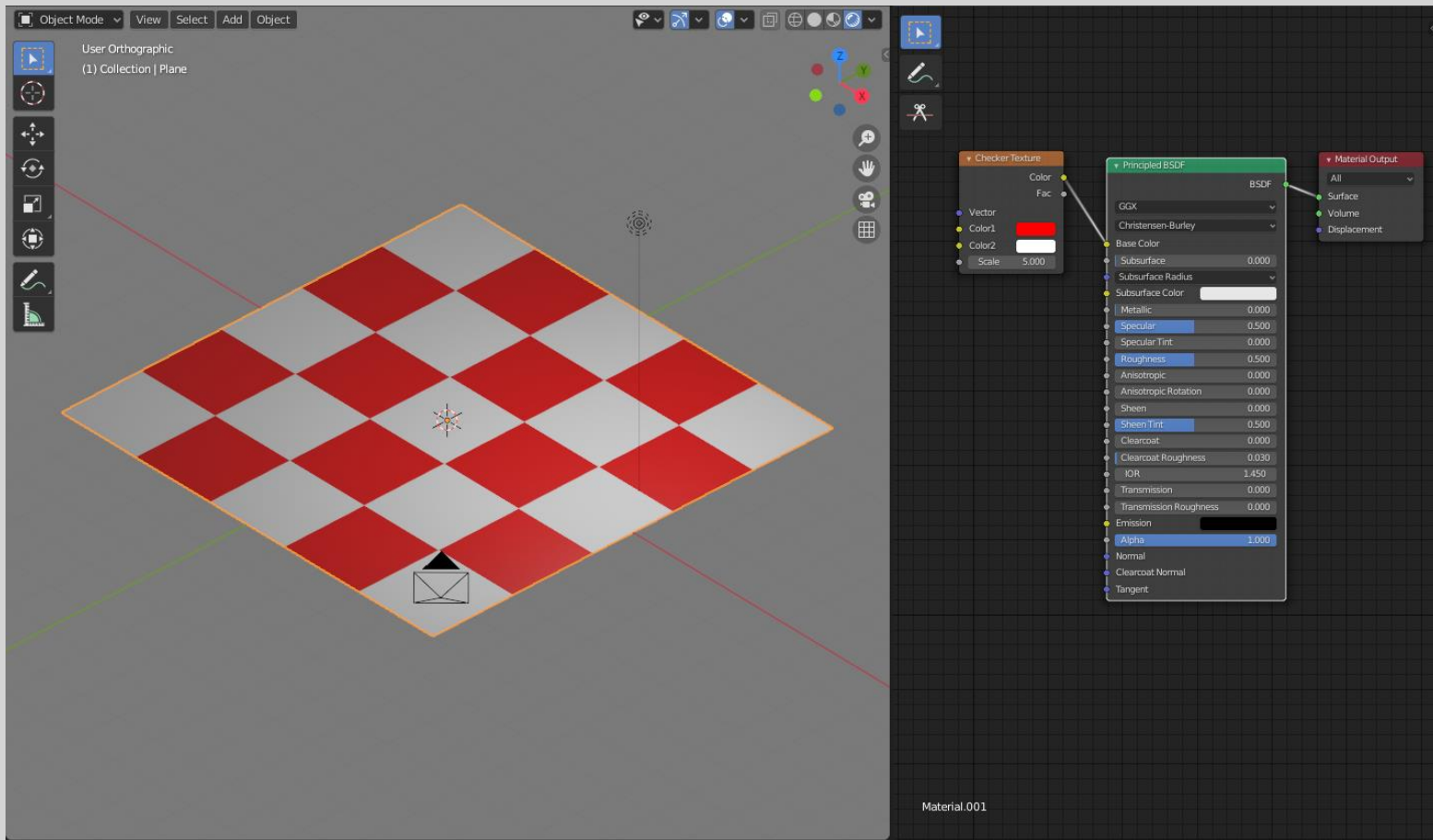
Blender/Python API

Materials/Textures



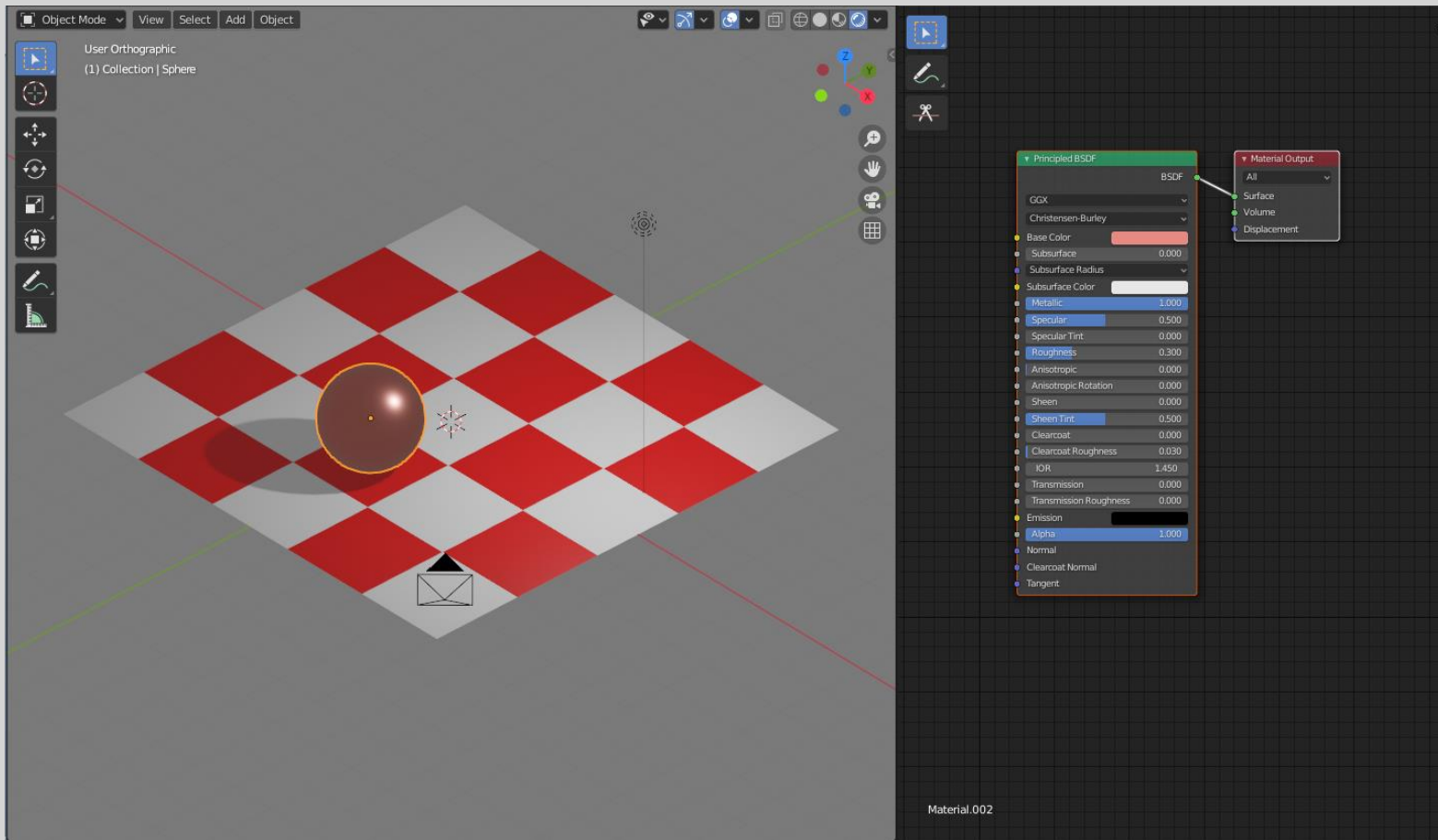


Blender/Python API Materials/Textures



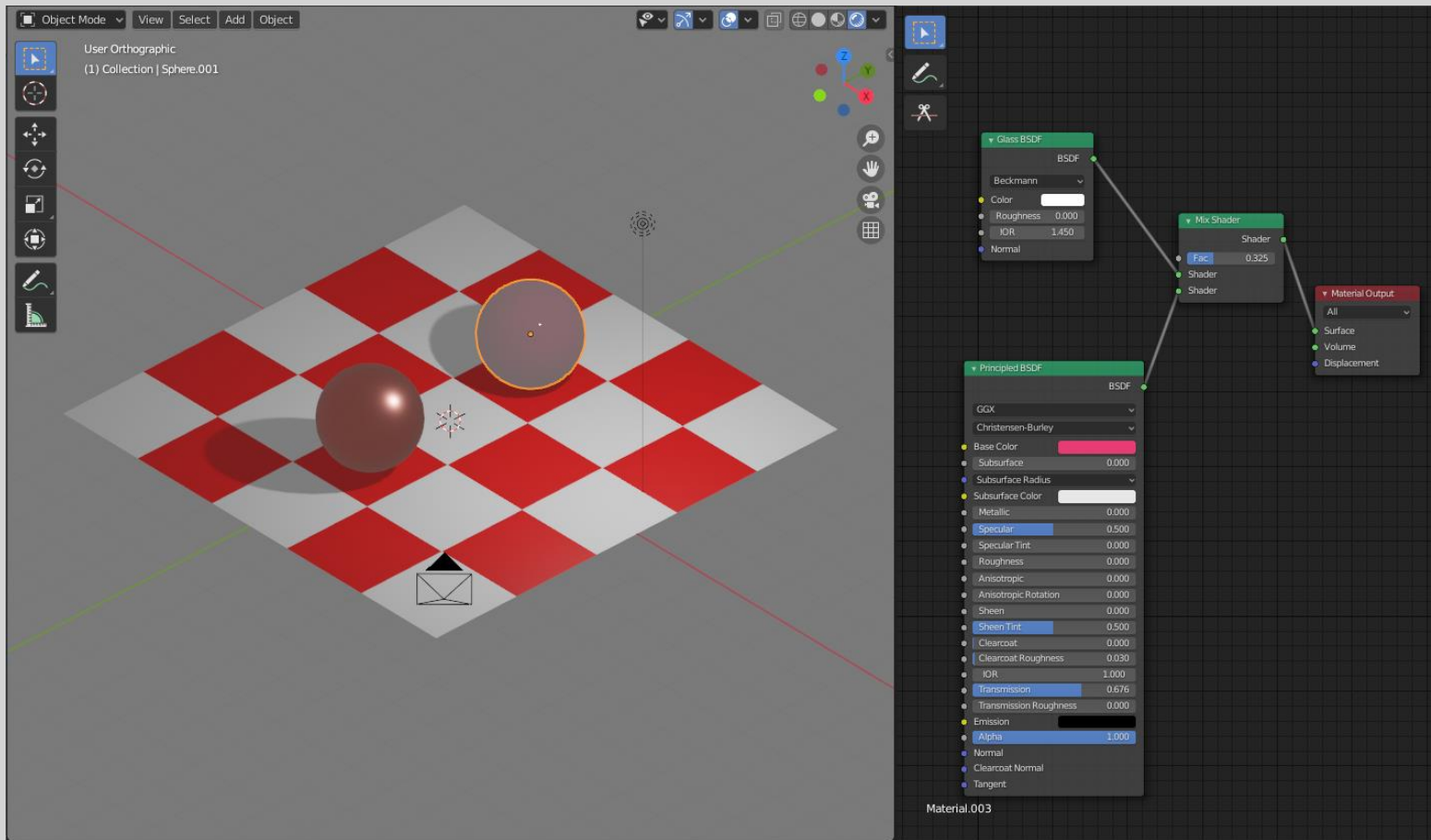


Blender/Python API Materials/Textures



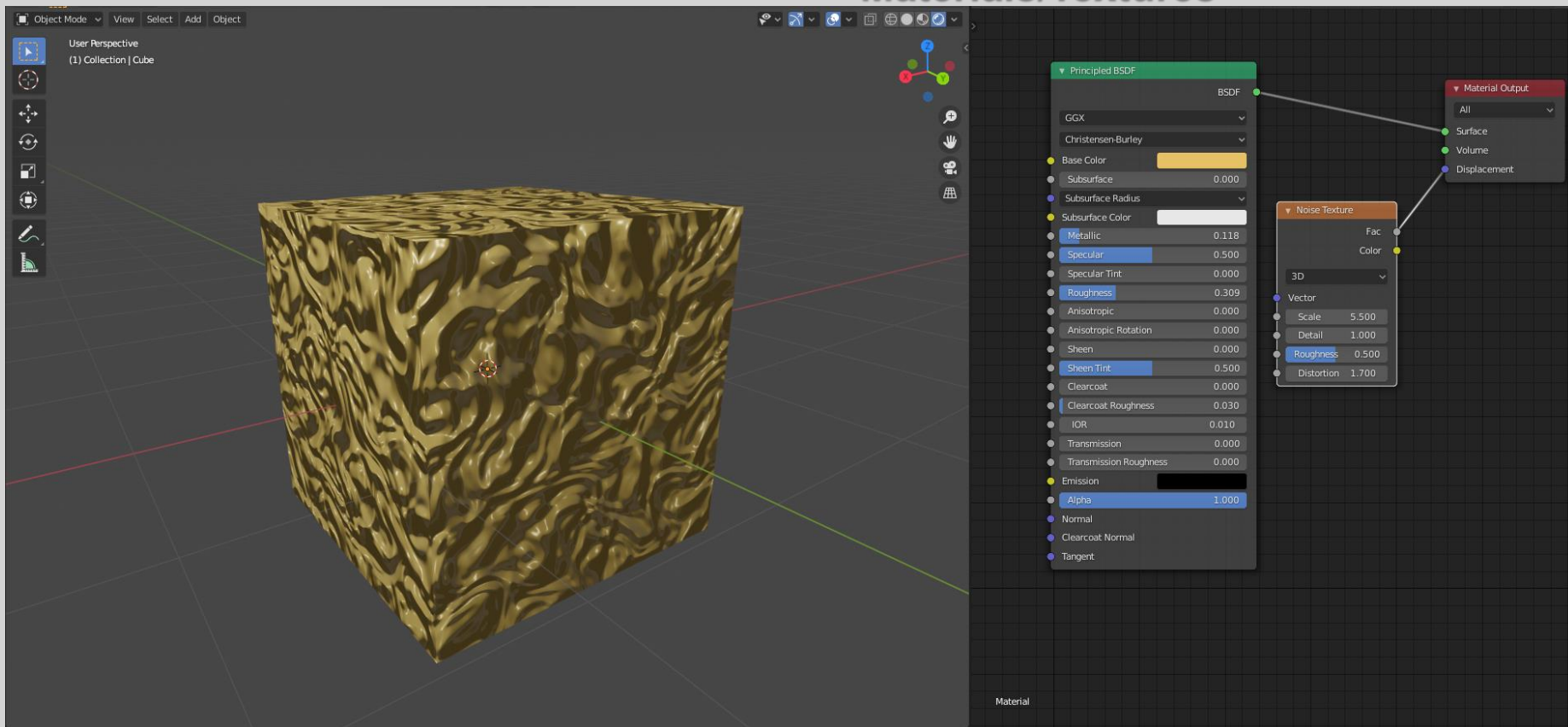


Blender/Python API Materials/Textures





Blender/Python API Materials/Textures

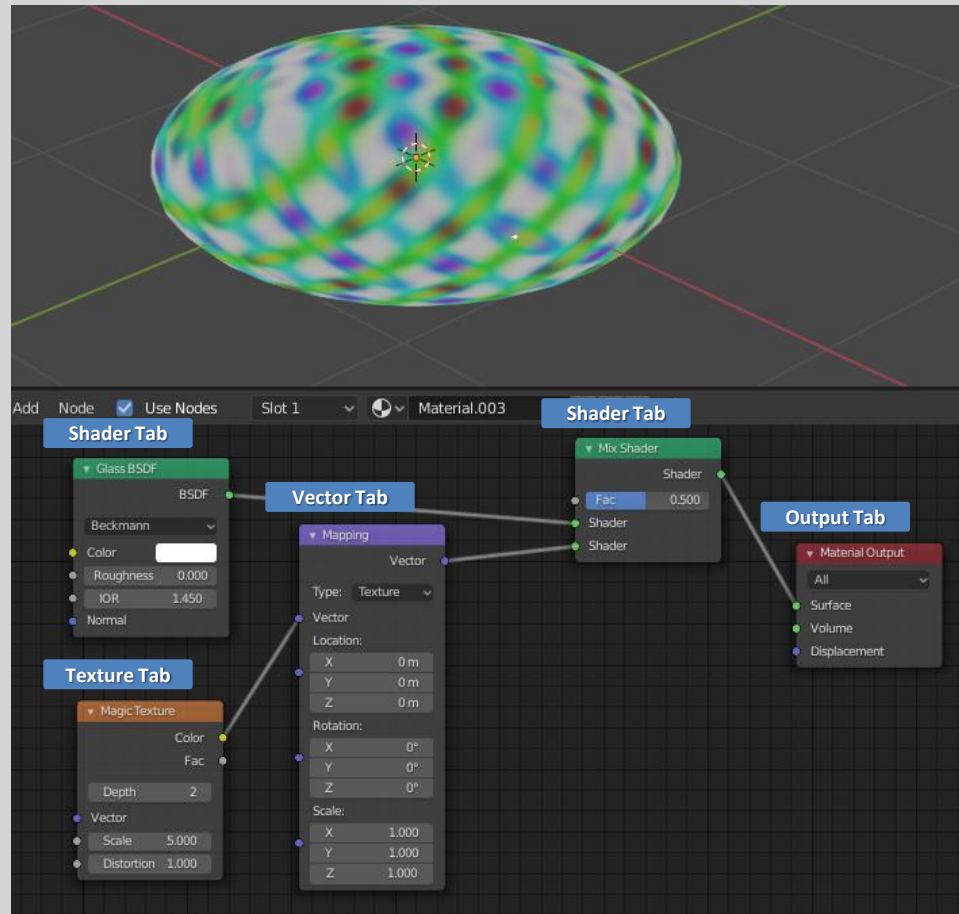


Blender/Python API

Materials/Textures

Magic Texture which is one of Blender's procedural or in-built Textures and at the same time is given the look of glass by the **Glass BSDF** Node. The Nodes are accessed from the Shader Editor, Add Menu.

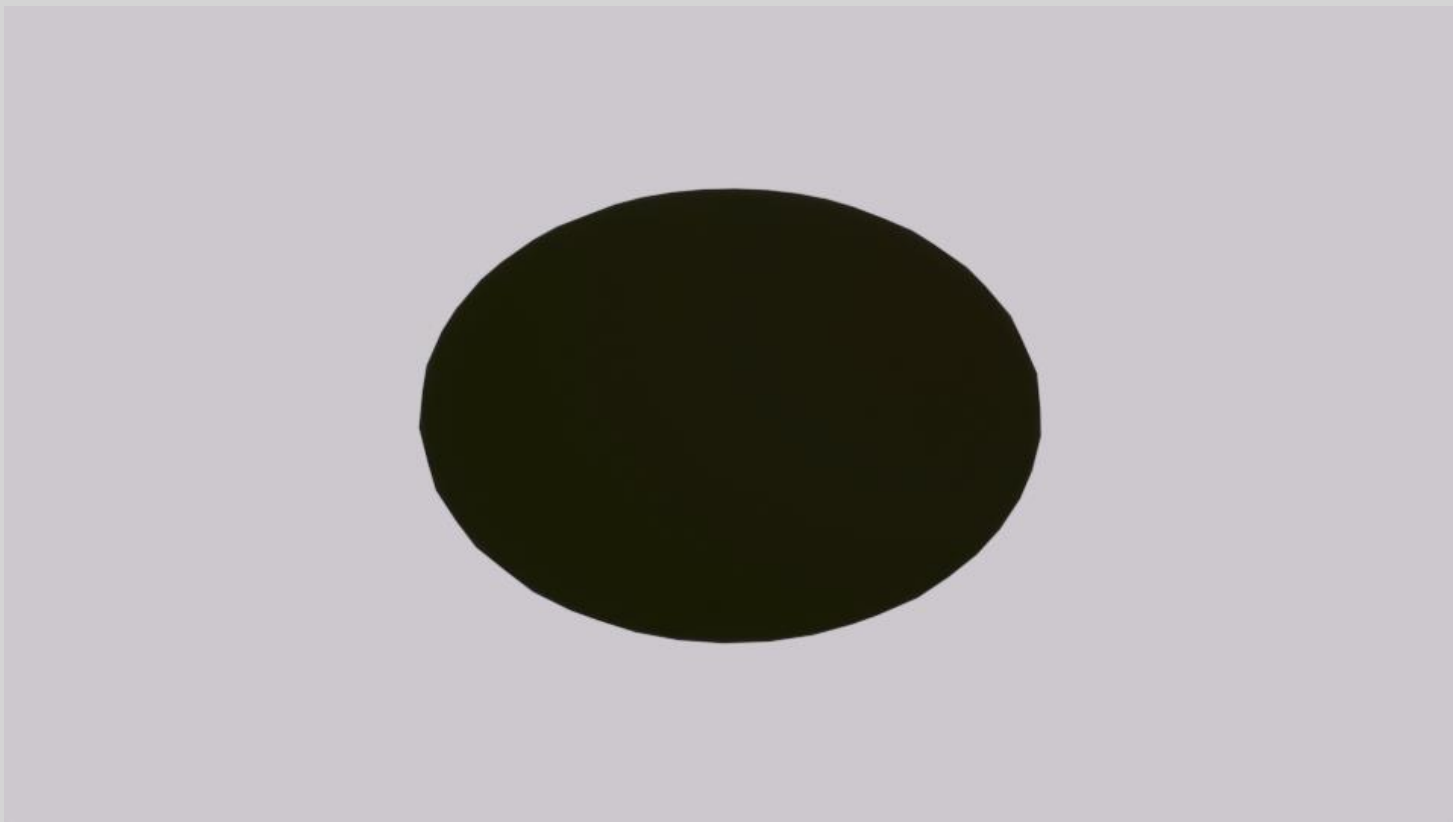
Magic Texture, which contains only two values for altering the characteristics of the texture: depth and turbulence.



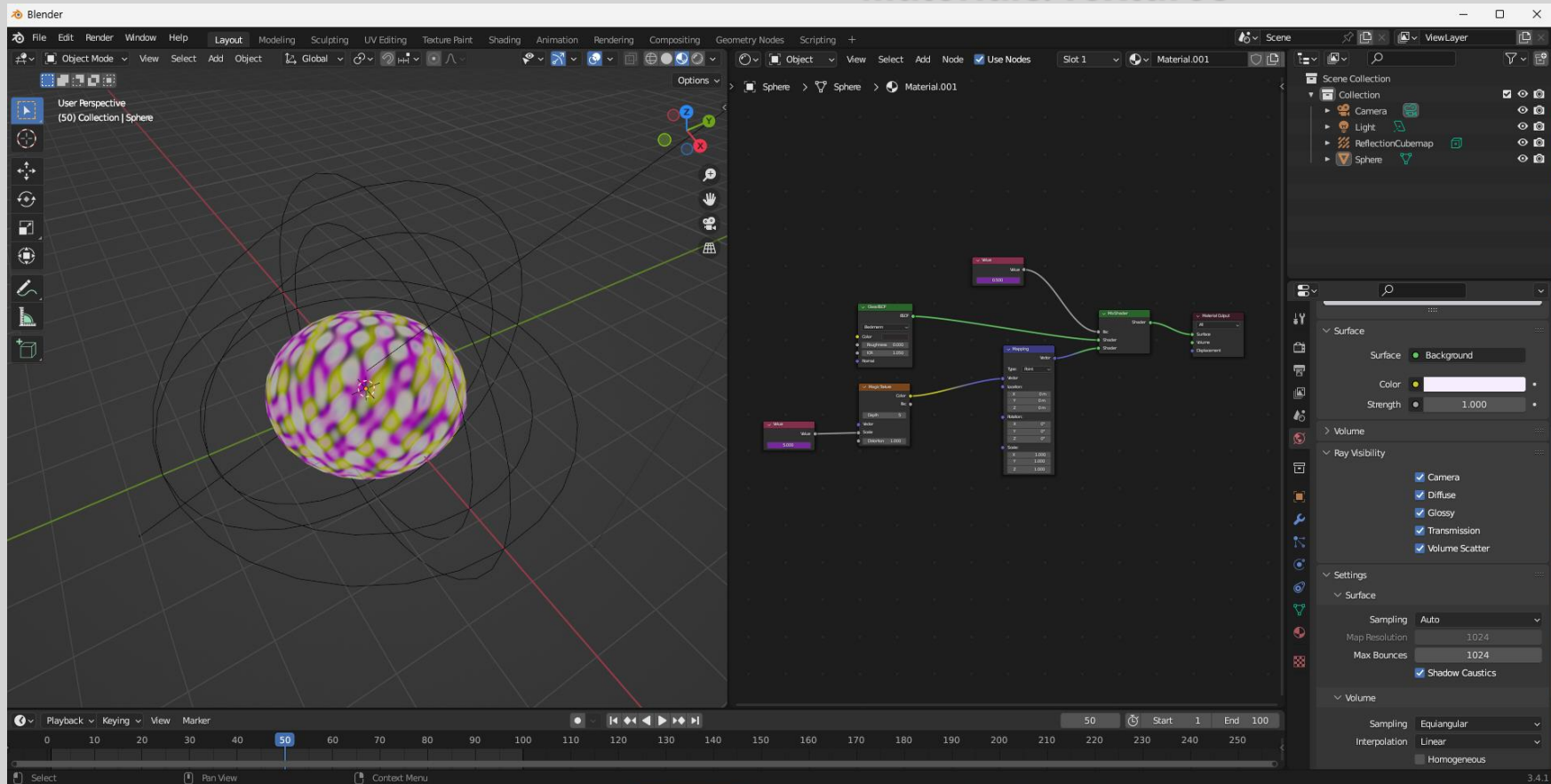


Blender/Python API

Materials/Textures

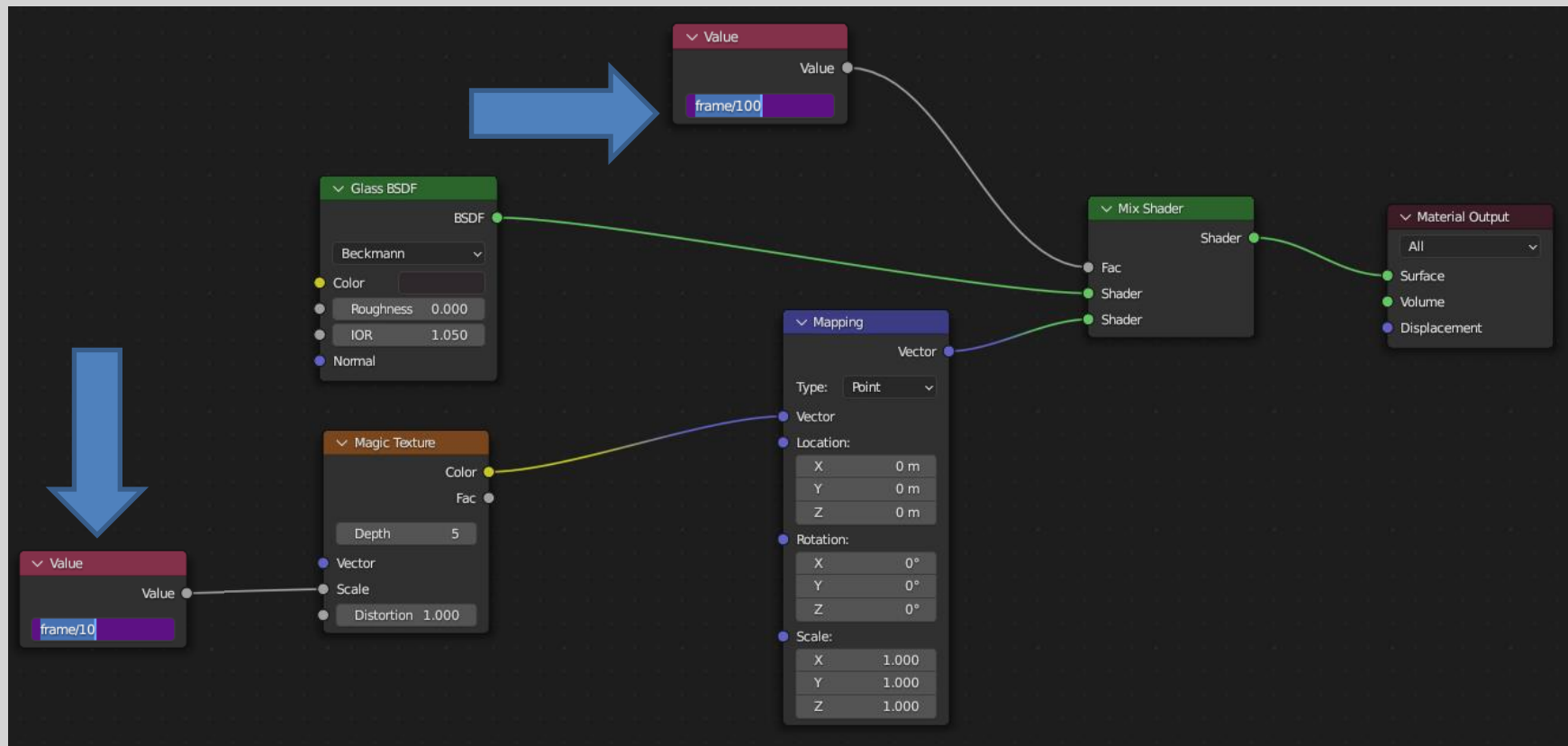


Blender/Python API Materials/Textures





Blender/Python API Materials/Textures





```
nodes = material.node_tree.nodes # shortcut for nodes
links = material.node_tree.links # shortcut for links

# Add the material output node
material_output = nodes.new("ShaderNodeOutputMaterial")
# Add the Value node for magic texture
magic_value = nodes.new("ShaderNodeValue")
# Add the Value node for mix shader (scale)
frame_value = nodes.new("ShaderNodeValue")
frame_value.outputs[0].default_value = frame_no/100
magic_text = nodes.new("ShaderNodeTexMagic")
glass_shader = nodes.new("ShaderNodeBsdfGlass")
mix_shader = nodes.new("ShaderNodeMixShader")
mapping = nodes.new("ShaderNodeMapping")

links.new(magic_value.outputs["Value"], magic_text.inputs["Scale"])
links.new(frame_value.outputs["Value"], mix_shader.inputs[0])
links.new(glass_shader.outputs["BSDF"], mix_shader.inputs[1])
links.new(magic_text.outputs["Color"], mapping.inputs["Vector"])
links.new(mapping.outputs["Vector"], mix_shader.inputs[2])
# Connect the Mix Shader node to the Material Output node
links.new(material_output.inputs["Surface"], mix_shader.outputs["Shader"])
```



SCRIPT LANGUAGES FOR ANIMATION

Blender/Python API Materials/Textures

The screenshot displays the Blender 3.0.8 interface with a Python script running in the Text Editor. The script defines a material named "My_Material" and sets up a texture mapping for a cube. The console shows the script's execution, including the creation of the material and the assignment of the texture.

```
1 import bpy
2 frame_no = 1
3 scene = bpy.context.scene
4
5 def frame_handler(scene):
6     global frame_no
7     frame_no = scene.frame_current
8     print(frame_no)
9     magic_value.outputs[0].default_value = frame_no/10
10    frame_value.outputs[0].default_value = frame_no/100
11
12 material_name = "My_Material"
13
14 # Check whether the material already exists
15 if bpy.data.materials.get(material_name):
16     material = bpy.data.materials[material_name]
17 else:
18     # Create the material
19     material = bpy.data.materials.new(material_name)
20     material.use_nodes = True
21
22 # Get the material nodes
23 nodes = material.node_tree.nodes
24 # Clear all nodes to start clean
25 for node in nodes:
26     nodes.remove(node)
27
28 links = material.node_tree.links
29
30 # Add the material output node
31 material_output = nodes.new("ShaderNodeOutputMaterial")
32 # Material output location = (480, 380)
33 # You need to add the actual path to the texture
34 magic_value = nodes.new("ShaderNodeValue")
35 magic_value.location = (-450, 5)
36 magic_value.outputs[0].default_value = frame_no/10
37 magic_value.outputs[0].value = frame_no/10
38 frame_value = nodes.new("ShaderNodeValue")
39 frame_value.location = (-250, 400)
40 frame_value.outputs[0].default_value = frame_no/100
41 frame_value.outputs[0].value = frame_no/100
42 magic_text = nodes.new("ShaderNodeText")
43 magic_text.location = (-150, 5)
44 glass_shader = nodes.new("ShaderNodeEmission")
45 # Glass shader location = (-250, 250)
46 mix_shader = nodes.new("ShaderNodeMixShader")
47 # Mix shader location = (250, 380)
48 mapping = nodes.new("ShaderNodeMapping")
49
50 links.new(magic_value.outputs[0], magic_text.inputs[0])
51 links.new(frame_value.outputs[0], mix_shader.inputs[0])
52 links.new(glass_shader.outputs[0], mix_shader.inputs[1])
53 links.new(magic_text.outputs[0], mapping.inputs[0])
54 links.new(mapping.outputs[0], mix_shader.inputs[2])
55 links.new(mix_shader.outputs[0], material_output.inputs[0])
56
57 # Connect the Mix Shader node to the Material output node
58 links.new(material_output.inputs[0], material_output.outputs[0])
59
60 # Get the active object
61 active_obj = bpy.context.view_layer.objects.active
62 # Check if the active object has a material slot, create one if it doesn't.
63 # Assign the material to the first slot for the active object
64 if active_obj.material_slots:
65     active_obj.material_slots[0].material = material
66 else:
67     active_obj.material_slots.append(material)
68
69 bpy.app.handlers.frame_change_pre.append(frame_handler)
```

The console output shows the script's execution, including the creation of the material and the assignment of the texture.

```
PYTHON INTERACTIVE CONSOLE 3.0.8 (main, Oct 18 2022, 21:01:35) [MSC v.1920 64 bit (AMD64)]
>>>
bpy.context.area.ui_type = 'TIMELINE'
bpy.ops.outliner.item_activate(deselect_all=True)
bpy.ops.text.run_script()
bpy.context.space_data.shading.type = 'SOLID'
bpy.context.space_data.shading.type = 'MATERIAL'
```

The screenshot displays the Blender 3.10.0 interface with a Python script in the left panel and a material node setup in the center-right panel.

Python Script (Left Panel):

```

import bpy

frame_no = 1
scene = bpy.context.scene

def frame_handler(scene):
    global frame_no
    frame_no = scene.frame_current
    print(frame_no)
    magic_value.outputs[0].default_value = frame_no/10
    frame_value.outputs[0].default_value = frame_no/100
    material_name = "My_Material"

    # check whether the material already exists
    if bpy.data.materials.get(material_name):
        material = bpy.data.materials[material_name]
    else:
        # create the material
        material = bpy.data.materials.new(material_name)
        material.use_nodes = True

    # get the material nodes
    nodes = material.node_tree.nodes
    # clear all nodes to start clean
    for node in nodes:
        nodes.remove(node)
    links = material.node_tree.links

    # Add the material output node
    material_output = nodes.new("ShaderNodeOutputMaterial")
    material_output.location = (400, 350)
    # Create Image Texture node and load the displacement texture.
    # You need to add the actual path to the texture
    magic_value = nodes.new("ShaderNodeMagic")
    magic_value.location = (-450, 5)
    magic_value.outputs[0].default_value = frame_no/10
    magic_value.value[0] = frame_no/10
    frame_value = nodes.new("ShaderNodeValue")
    frame_value.location = (-250, 400)
    frame_value.outputs[0].default_value = frame_no/100
    magic_text = nodes.new("ShaderNodeText")
    magic_text.location = (-250, 250)
    glass_shader = nodes.new("ShaderNodeGlass")
    glass_shader.location = (-250, 250)
    mix_shader = nodes.new("ShaderNodeMixShader")
    mix_shader.location = (-250, 350)
    mapping = nodes.new("ShaderNodeMapping")
    mapping.location = (-250, 350)

    links.new(magic_value.outputs["Value"], mix_shader.inputs["Scale"])
    links.new(frame_value.outputs["Value"], mix_shader.inputs["0"])
    links.new(glass_shader.outputs["BSDF"], mix_shader.inputs["1"])
    links.new(magic_text.outputs["Color"], mapping.inputs["Vector"])
    links.new(mapping.outputs["Vector"], mix_shader.inputs["2"])
    # Connect the mix_shader node to the material output node
    links.new(material_output.inputs["Surface"], mix_shader.outputs["Shader"])

    # Get the active object
    active_obj = bpy.context.view_layer.objects.active
    # Check if the active object has a material slot, create one if it doesn't.
    # Assign the material to the first slot for the active object.
    if active_obj.material_slots:
        active_obj.material_slots[0].material = material
    else:
        active_obj.material_slots.append(material)

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

Material Node Setup (Center-Right Panel):

- Value** node (Value: 0.400) connected to **Mix Shader** node (Factor: 0.400).
- Image Texture** node (Image: BGGF) connected to **Mix Shader** node (Texture: 0.400).
- Mix Shader** node (Factor: 0.400, Texture: 0.400) connected to **Material Output** node (Surface: 0.400).
- Value** node (Value: 0.100) connected to **Magic Text** node (Text: 0.100).
- Magic Text** node (Text: 0.100) connected to **Mapping** node (Vector: 0.100).
- Mapping** node (Vector: 0.100) connected to **Mix Shader** node (Texture: 0.400).

Right Panel (Properties):

- Scene** tab: Render Engine: Eevee, Viewport: 64, Viewport Denoising: 16.
- Render** tab: Ambient Occlusion, Bloom, Depth of Field, Subsurface Scattering, Screen Space Reflections, Motion Blur, Volumetrics, Performance, Curves, Shadows, Indirect Lighting, Film, Simplify.



```
import bpy

frame_no = 1
scene = bpy.context.scene

def frame_handler(scene):
    global frame_no
    frame_no = scene.frame_current
    print(frame_no)
    magic_value.outputs[0].default_value = frame_no/10
    frame_value.outputs[0].default_value = frame_no/100

material_name = "My_Material"
# check whether the material already exists
if bpy.data.materials.get(material_name):
    material = bpy.data.materials[material_name]
else:
    # create the material
    material = bpy.data.materials.new(material_name)
    material.use_nodes = True
```



Blender/Python API Materials/Textures

```
# get the material nodes
```

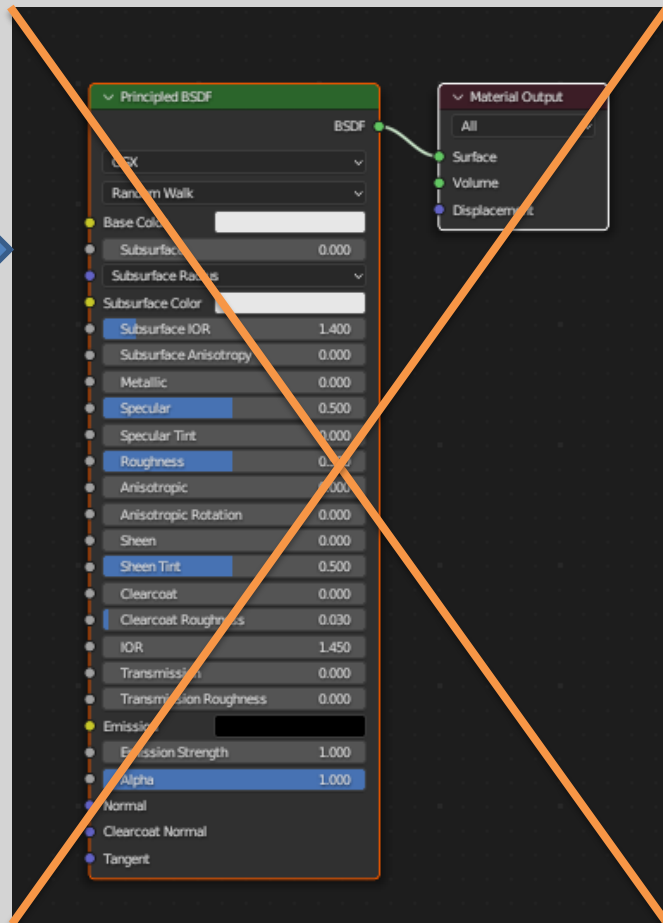
```
nodes = material.node_tree.nodes
```

```
# clear all nodes to start clean
```

```
for node in nodes:
```

```
    nodes.remove(node)
```

```
links = material.node_tree.links # links
```





```
# Add the material output node
material_output = nodes.new("ShaderNodeOutputMaterial")
material_output.location = (450, 350)
# Add the Value node for magic texture
magic_value = nodes.new("ShaderNodeValue")
magic_value.location = (-450, 5)
magic_value.outputs[0].default_value = frame_no/10
# Add the Value node for mix shader (scale)
frame_value = nodes.new("ShaderNodeValue")
frame_value.location = (-250, 400)
frame_value.outputs[0].default_value = frame_no/100
# Add the Magic Texture node
magic_text = nodes.new("ShaderNodeTexMagic")
magic_text.location = (-250, 5)
# Add the Glass Shader node
glass_shader = nodes.new("ShaderNodeBsdfGlass")
glass_shader.location = (-250, 250)
# Add the Mix Shader node
mix_shader = nodes.new("ShaderNodeMixShader")
mix_shader.location = (250, 350)
# Add the Mix Shader node
mapping = nodes.new("ShaderNodeMapping")
```



```
# Link all nodes
```

```
links.new(magic_value.outputs["Value"], magic_text.inputs["Scale"])
```

```
links.new(frame_value.outputs["Value"], mix_shader.inputs[0])
```

```
links.new(glass_shader.outputs["BSDF"], mix_shader.inputs[1])
```

```
links.new(magic_text.outputs["Color"], mapping.inputs["Vector"])
```

```
links.new(mapping.outputs["Vector"], mix_shader.inputs[2])
```

```
# Connect the Mix Shader node to the Material Output node
```

```
links.new(material_output.inputs["Surface"], mix_shader.outputs["Shader"])
```

```
# Get the active object
```

```
active_obj = bpy.context.view_layer.objects.active
```

```
# Check if the active object has a material slot, create one if it doesn't.
```

```
# Assign the material to the first slot for the active object.
```

```
if active_obj.material_slots:
```

```
    active_obj.material_slots[0].material = material
```

```
else:
```

```
    bpy.ops.material.new()
```

```
    active_obj.material_slots[0].material = material
```

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



Blender/Python API Materials/Textures

The screenshot displays the Blender 3.7.4 interface with the following components:

- Python Script (Left):** A script titled 'import bpy' that creates a material, adds a displacement texture, and assigns it to a plane object. The script includes comments and uses bpy modules for material and texture creation.
- Shader Editor (Center):** A Principled BSDF node is connected to a Material Output node. A Texture Coordinate node is connected to a Displacement node, which is then connected to the Principled BSDF node's Displacement input.
- Properties Panel (Right):** The 'Use Nodes' tab is active, showing the material's node tree. The 'Surface' node is connected to the 'Principled BSDF' node.
- 3D Viewport (Left):** A 3D view of a plane with a displacement texture applied, showing a distorted grid pattern.
- Console (Bottom Left):** The Python Interactive Console shows the execution of the script, with output messages indicating successful material creation and assignment.



Blender/Python API Materials/Textures

```
import bpy
```

```
# Create a material
```

```
material = bpy.data.materials.new(name="Example_Material")
```

```
material.use_nodes = True
```

```
nodes = material.node_tree.nodes
```

```
links = material.node_tree.links
```

```
# Reuse the material output node that is created by default
```

```
material_output = nodes.get("Material Output")
```

```
# Create Image Texture node and load the displacement texture.
```

```
# You need to add the actual path to the texture.
```

```
displacement_tex = nodes.new("ShaderNodeTexImage")
```

```
displacement_tex.image = bpy.data.images.load("blender_logo.png")
```

```
displacement_tex.image.colorspace_settings.name = "Non-Color"
```



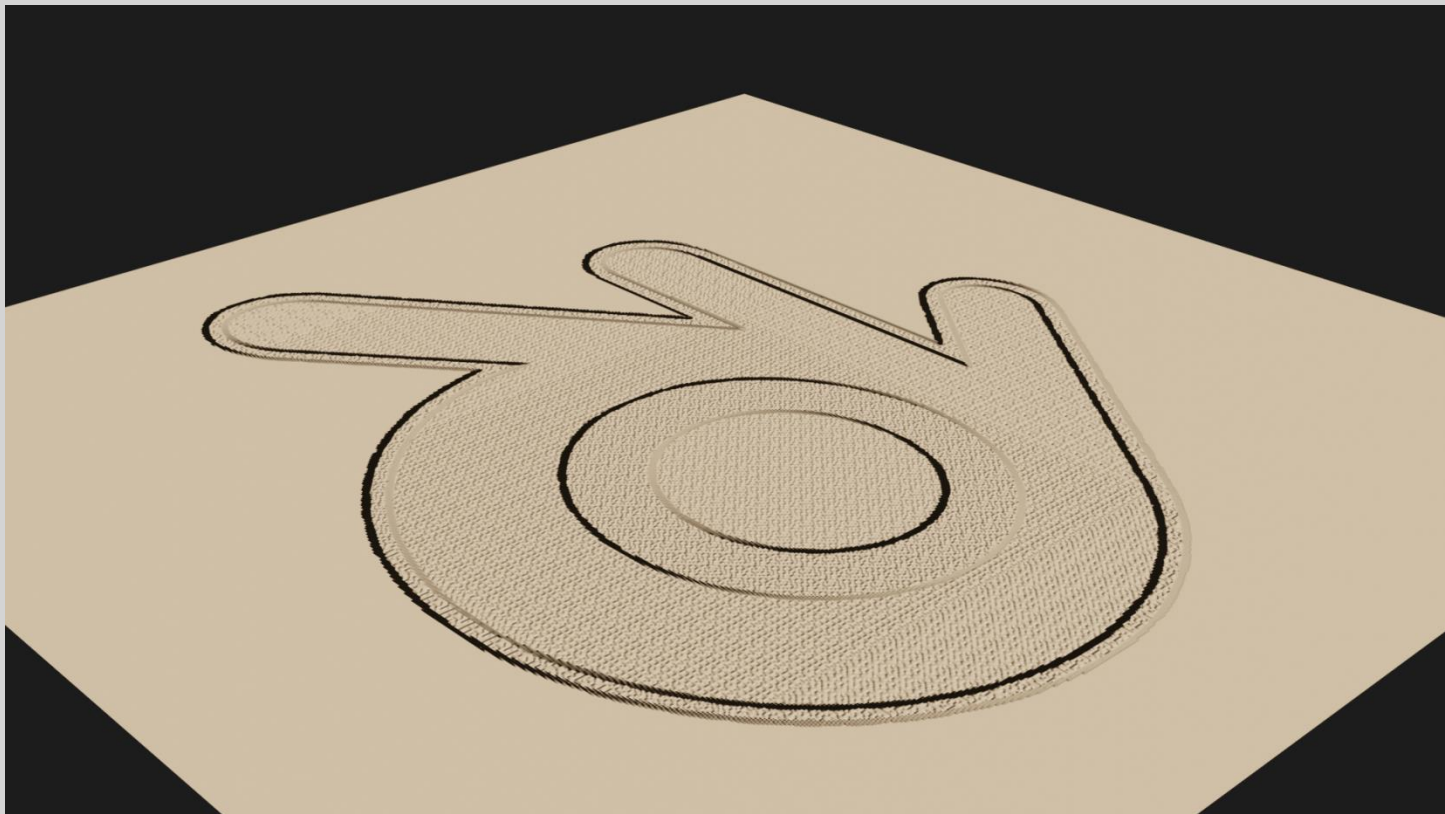

Blender/Python API Materials/Textures

```
# Create the Displacement node
displacement = nodes.new("ShaderNodeDisplacement")
# Create the Texture Coordinate node
tex_coordinate = nodes.new("ShaderNodeTexCoord")
# Connect the Texture Coordinate node to the displacement texture.
# This uses the active UV map of the object.
links.new(displacement_tex.inputs["Vector"], tex_coordinate.outputs["UV"])
# Connect the displacement texture to the Displacement node
links.new(displacement.inputs["Height"], displacement_tex.outputs["Color"])
# Connect the Displacement node to the Material Output node
links.new(material_output.inputs["Displacement"], displacement.outputs["Displacement"])
# Get the active object
active_obj = bpy.context.view_layer.objects.active
# Check if the active object has a material slot, create one if it doesn't.
# Assign the material to the first slot for the active object.
if active_obj.material_slots:
    active_obj.material_slots[0].material = material
else:
    bpy.ops.material.new()
    active_obj.material_slots[0].material = material
```



Blender/Python API

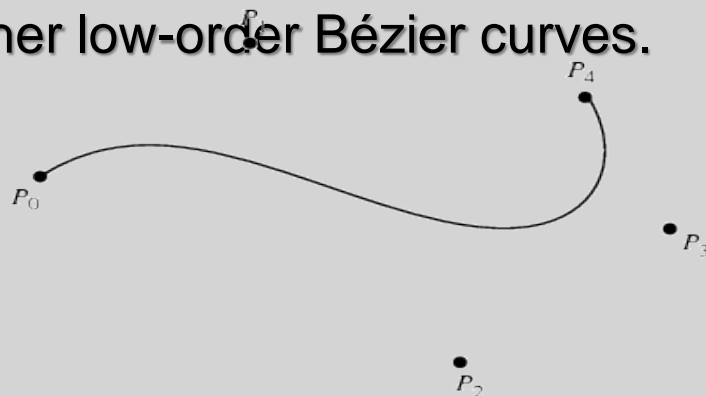
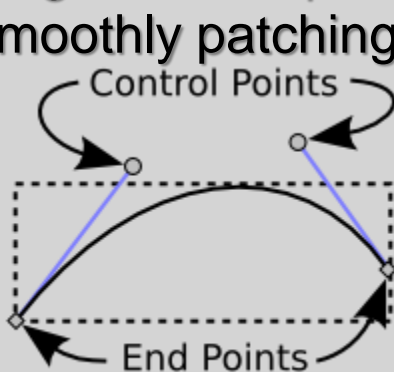
Materials/Textures





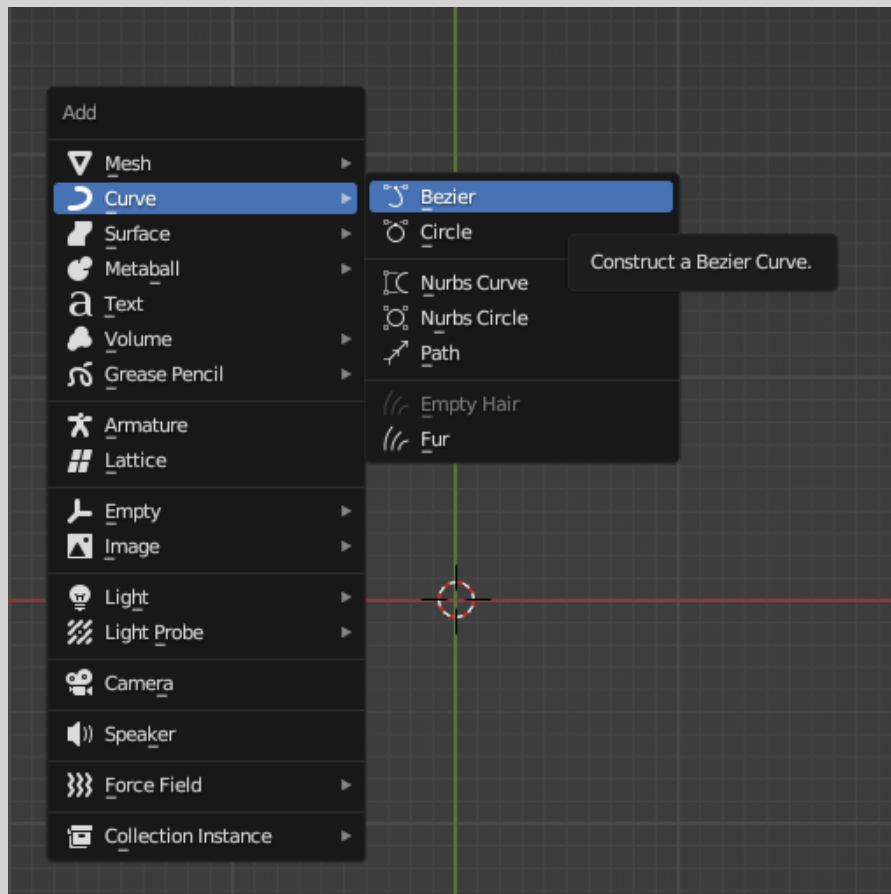
Blender/Python API Bezier Curves

The Bézier curve always **passes through** the **first** and **last** control points and lies within the convex hull of the control points. The curve is **tangent to $P_1 - P_0$** and **$P_n - P_{n-1}$** at the endpoints. A desirable property of these curves is that the curve can be translated and rotated by performing these operations on the control points. Undesirable properties of Bézier curves are their numerical instability for large numbers of control points, and the fact that moving a single control point changes the global shape of the curve. The former is sometimes avoided by smoothly patching together low-order Bézier curves.



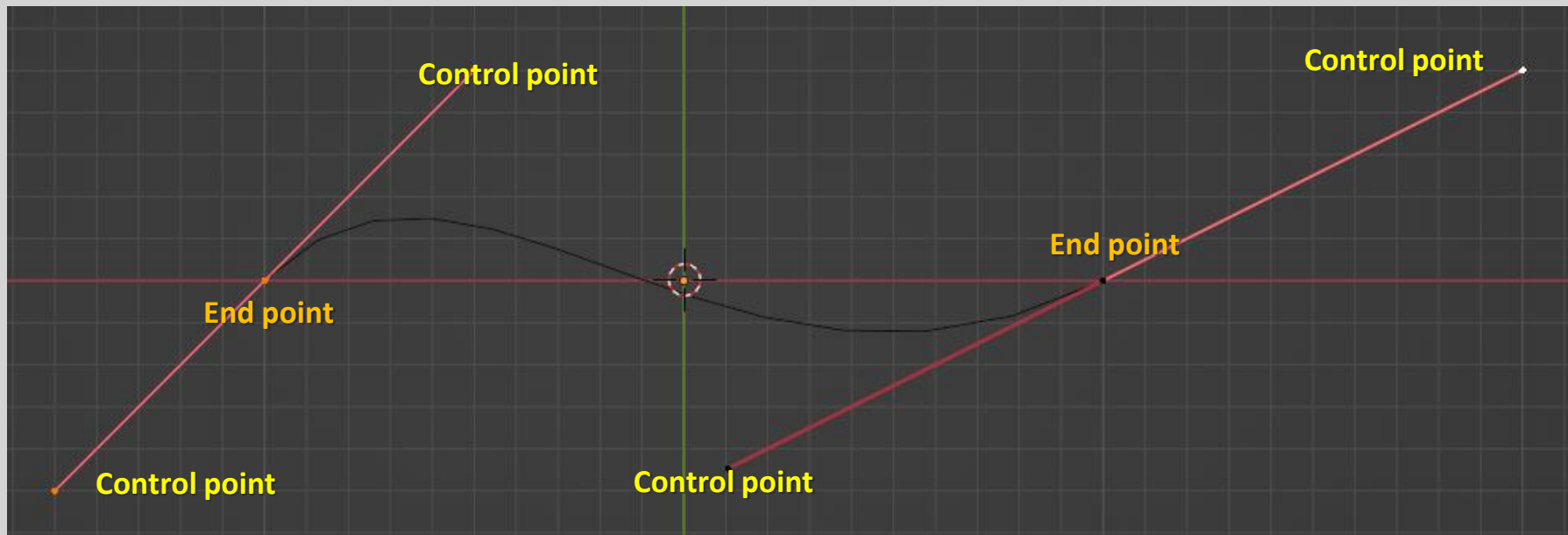
Blender/Python API

Bezier Curves



Blender/Python API

Bezier Curves





import bpy, csv

Blender/Python API Bezier Curves

```
csv_filepath= "yourPath\myControlPoints.csv" # replace with a real path
```

```
# read all end points from csv file
```

```
with open(csv_filepath) as csv_file: # work with the opened file
```

```
# first read the complete file (we need to know the amount of points)
```

```
csv_reader = csv.reader(csv_file, delimiter=',')
```

```
control_points = []
```

```
for idx, row in enumerate(csv_reader):
```

```
    if idx != 0: # ignore csv header
```

```
        control_points.append((float(row[0]), float(row[1]), float(row[2])))
```

```
# create bezier curve and add enough control points to it
```

```
bpy.ops.curve.primitive_bezier_curve_add()
```

```
curve = bpy.context.active_object
```

```
bez_points = curve.data.splines[0].bezier_points
```

```
# note: a created bezier curve has already 2 control points
```

```
bez_points.add(len(control_points) - 2)
```

```
# now copy the csv data
```

```
for i in range(len(control_points)):
```

```
    bez_points[i].co = control_points[i]
```

```
    bez_points[i].handle_left_type = 'FREE'
```

```
    bez_points[i].handle_right_type = 'FREE'
```

```
# add your end points locations here
```

```
bez_points[i].handle_left = control_points[i]
```

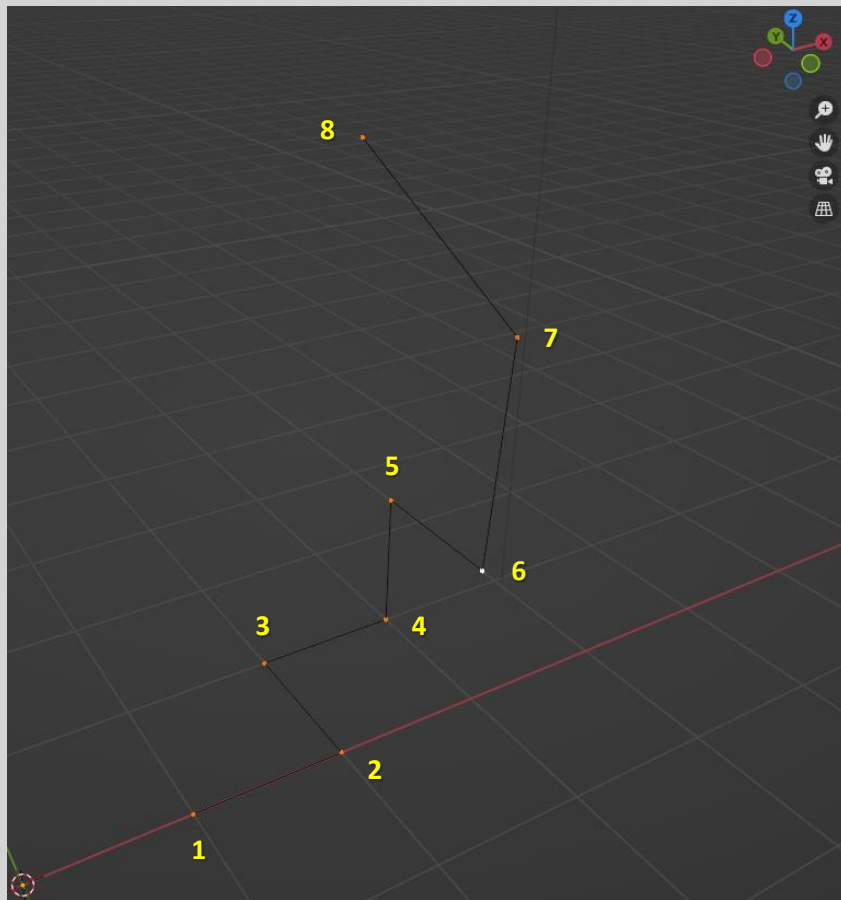
```
bez_points[i].handle_right = control_points[i]
```

myControlPoints.csv	
1	x;y;z;
2	1;0;0;
3	2;0;0;
4	2;1;0;
5	3;1;0;
6	3;1;1;
7	3;0;1;
8	4;1;2;
9	4;3;3;
10	



Blender/Python API Bezier Curves

myControlPoints.csv			
1	x	y	z
2	1	0	0
3	2	0	0
4	2	1	0
5	3	1	0
6	3	1	1
7	3	0	1
8	4	1	2
9	4	3	3
10			





Blender/Python API Bezier Curves

```
enum in ['FREE', 'VECTOR', 'ALIGNED', 'AUTO'], default 'FREE'  
  
import bpy  
  
# Type of the first handle  
print(bpy.data.curves['BezierCurve'].splines[0].bezier_points[0].handle_left_type)  
  
# Type of the second handle  
print(bpy.data.curves['BezierCurve'].splines[0].bezier_points[0].handle_right_type)  
  
# Coordinates of the first handle  
print(bpy.data.curves['BezierCurve'].splines[0].bezier_points[0].handle_left)  
  
# Coordinates of the second handle  
print(bpy.data.curves['BezierCurve'].splines[0].bezier_points[0].handle_right)
```

<https://docs.blender.org/api/current/bpy.types.BezierSplinePoint.html>

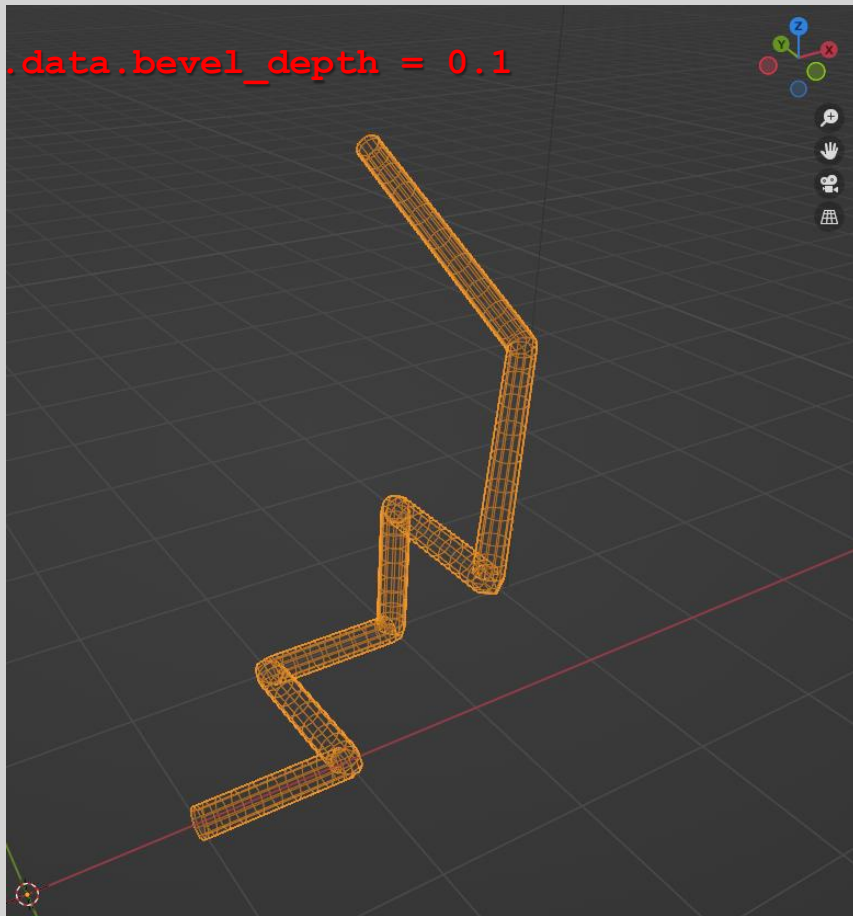


Blender/Python API Bezier Curves

```
# Both are the same
```

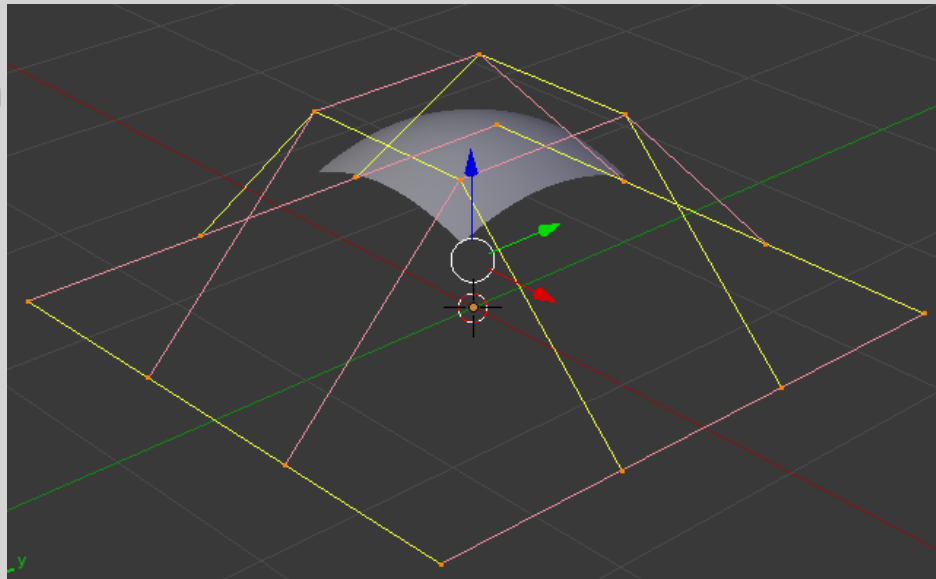
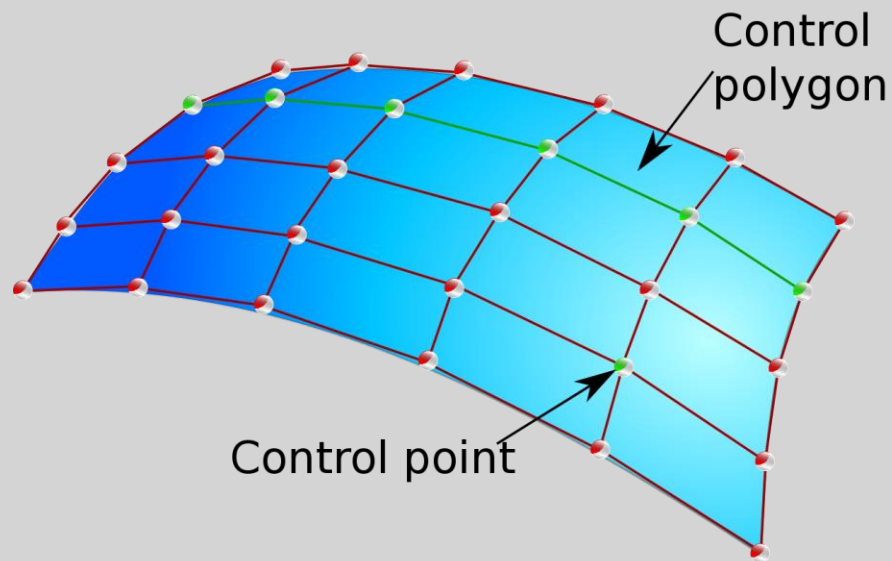
```
#bpy.data.objects['BezierCurve'].data.bevel_depth = 0.1
```

```
curve.data.bevel_depth = 0.1
```





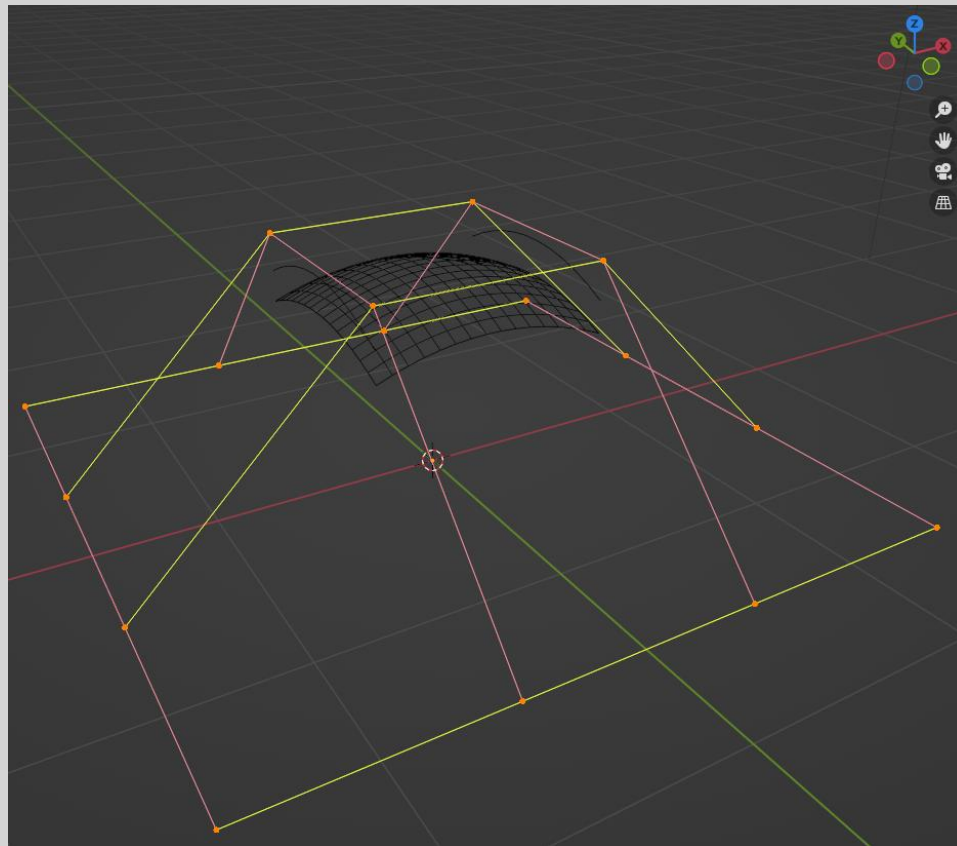
Blender/Python API Nurbs



NURBS, Non-Uniform Rational B-Splines, are mathematical representations of 3D geometry that can accurately describe any shape from a simple 2D line, circle, arc, or curve to the most complex 3D organic free-form surface or solid.

Blender/Python API

Nurbs





```
import bpy
from mathutils import Vector

surface_data = bpy.data.curves.new('BSurface', 'SURFACE')
surface_data.dimensions = '3D'

# 16 coordinates
points = [
    Vector((-1.5, -1.5, 0.0, 1.0)), Vector((-1.5, -0.5, 0.0, 1.0)),
    Vector((-1.5, 0.5, 0.0, 1.0)), Vector((-1.5, 1.5, 0.0, 1.0)),
    Vector((-0.5, -1.5, 0.0, 1.0)), Vector((-0.5, -0.5, 1.0, 1.0)),
    Vector((-0.5, 0.5, 1.0, 1.0)), Vector((-0.5, 1.5, 0.0, 1.0)),
    Vector((0.5, -1.5, 0.0, 1.0)), Vector((0.5, -0.5, 1.0, 1.0)),
    Vector((0.5, 0.5, 1.0, 1.0)), Vector((0.5, 1.5, 0.0, 1.0)),
    Vector((1.5, -1.5, 0.0, 1.0)), Vector((1.5, -0.5, 0.0, 1.0)),
    Vector((1.5, 0.5, 0.0, 1.0)), Vector((1.5, 1.5, 0.0, 1.0))
]
```




```
# set points per segments (U * V)
for i in range(0, 16, 4):
    spline = surface_data.splines.new(type='NURBS')
    spline.points.add(3) # already has a default vector

    for p, new_co in zip(spline.points, points[i:i+4]):
        p.co = new_co

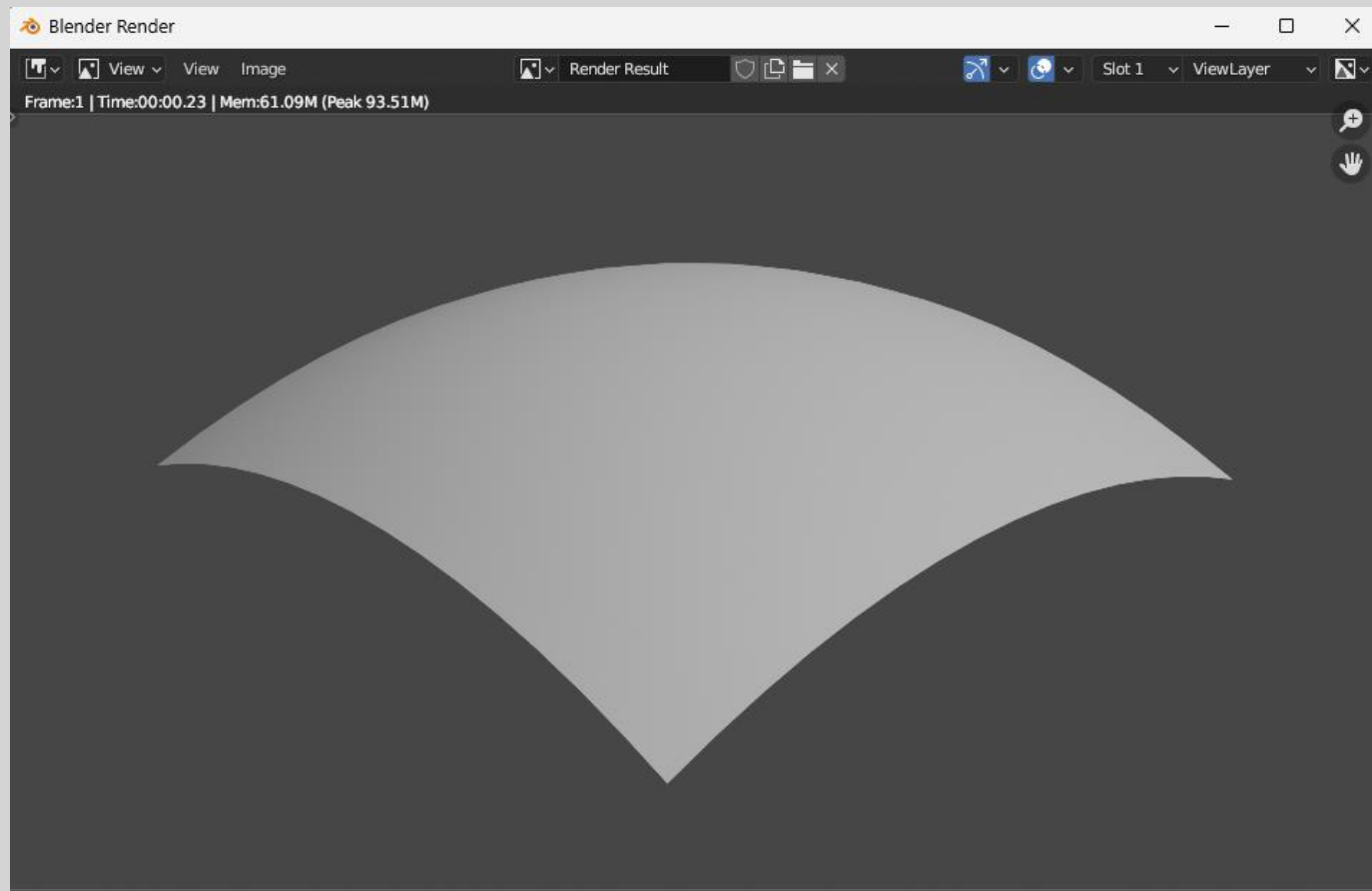
surface_object = bpy.data.objects.new('Nurbs_Obj', surface_data)
scene = bpy.context.scene
scene.collection.objects.link(surface_object)

splines = surface_object.data.splines
for s in splines:
    for p in s.points:
        p.select = True

bpy.context.view_layer.objects.active = surface_object
bpy.ops.object.mode_set(mode = 'EDIT')
bpy.ops.curve.make_segment()
```

Blender/Python API

Nurbs





Keyframe

- A **Keyframe** is simply a marker of time which stores the value of a property.
- For example, a **Keyframe** might define that the horizontal position of a cube is at 3m on frame 1.
- The purpose of a **Keyframe** is to allow for interpolated animation, meaning, for example, that the user could then add another key on frame 10, specifying the cube's horizontal position at 20m, and Blender will automatically determine the correct position of the cube for all the frames between frame 1 and 10 depending on the chosen interpolation method (e.g. Linear, Bézier, Quadratic, etc.).



Blender/Python API Keyframe

```
import bpy

positions = (0,0,2), (0,1,2), (3,2,1), (3,4,1), (1,2,1)
start_pos = (0,0,0)

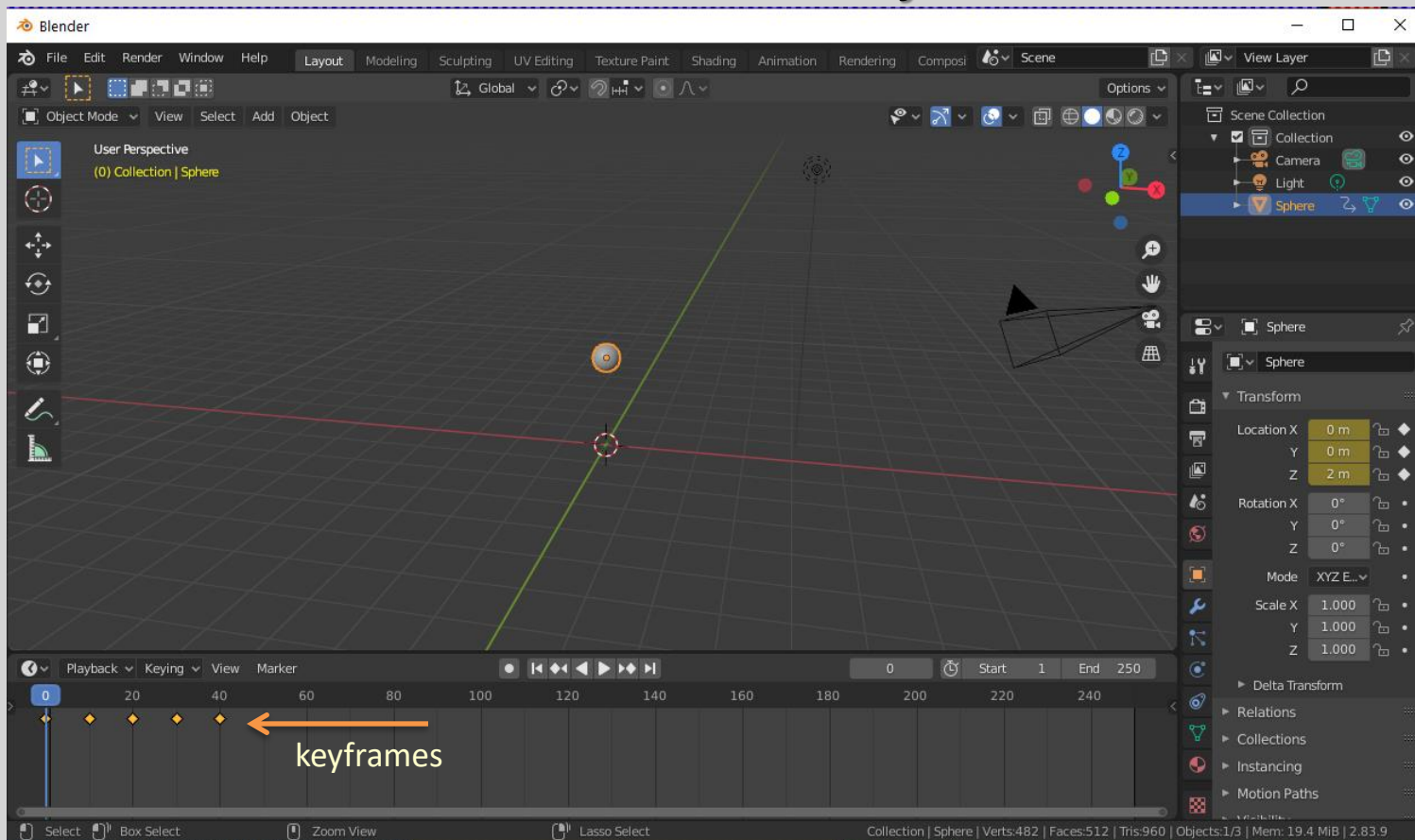
bpy.ops.mesh.primitive_uv_sphere_add(radius=0.3, location=start_pos)
bpy.ops.object.shade_smooth()
ob = bpy.context.active_object

frame_num = 0

for position in positions:
    bpy.context.scene.frame_set(frame_num)
    ob.location = position
    ob.keyframe_insert(data_path="location", index=-1)
    frame_num += 10
```

Blender/Python API

Keyframe





Blender/Python API Keyframe Rendering

```
import bpy
import os, sys

# key positions
positions = (0,0,2), (0,1,2), (3,2,1), (3,4,1), (1,2,1)
start_pos = (0,0,0)

scene = bpy.context.scene
scene.frame_end = 49
bpy.ops.mesh.primitive_uv_sphere_add(radius=0.3, location=start_pos)
bpy.ops.object.shade_smooth()
ob = bpy.context.active_object
frame_num = 0
# insert key frames
for position in positions:
    bpy.context.scene.frame_set(frame_num)
    ob.location = position
    ob.keyframe_insert(data_path="location", index=-1)
    frame_num += 10
```




Blender/Python API Keyframe Rendering

```
# Check the frame Number
print("Scene %r frames: %d..%d = %d" % (scene.name, scene.frame_start,
scene.frame_end, scene.frame_end - scene.frame_start + 1)) # frame_end is
included

fp = scene.render.filepath # get existing output path

if(bpy.context.space_data == None):
    cwd = os.path.dirname(os.path.abspath(__file__))
else:
    cwd = os.path.dirname(bpy.context.space_data.text.filepath)
    # Get folder of script and add current working directory to path
    sys.path.append(cwd)

# Specify folder to save rendering
render_folder = os.path.join(cwd, 'rendering')
if(not os.path.exists(render_folder)):
    os.mkdir(render_folder)
```



Blender/Python API Keyframe Rendering

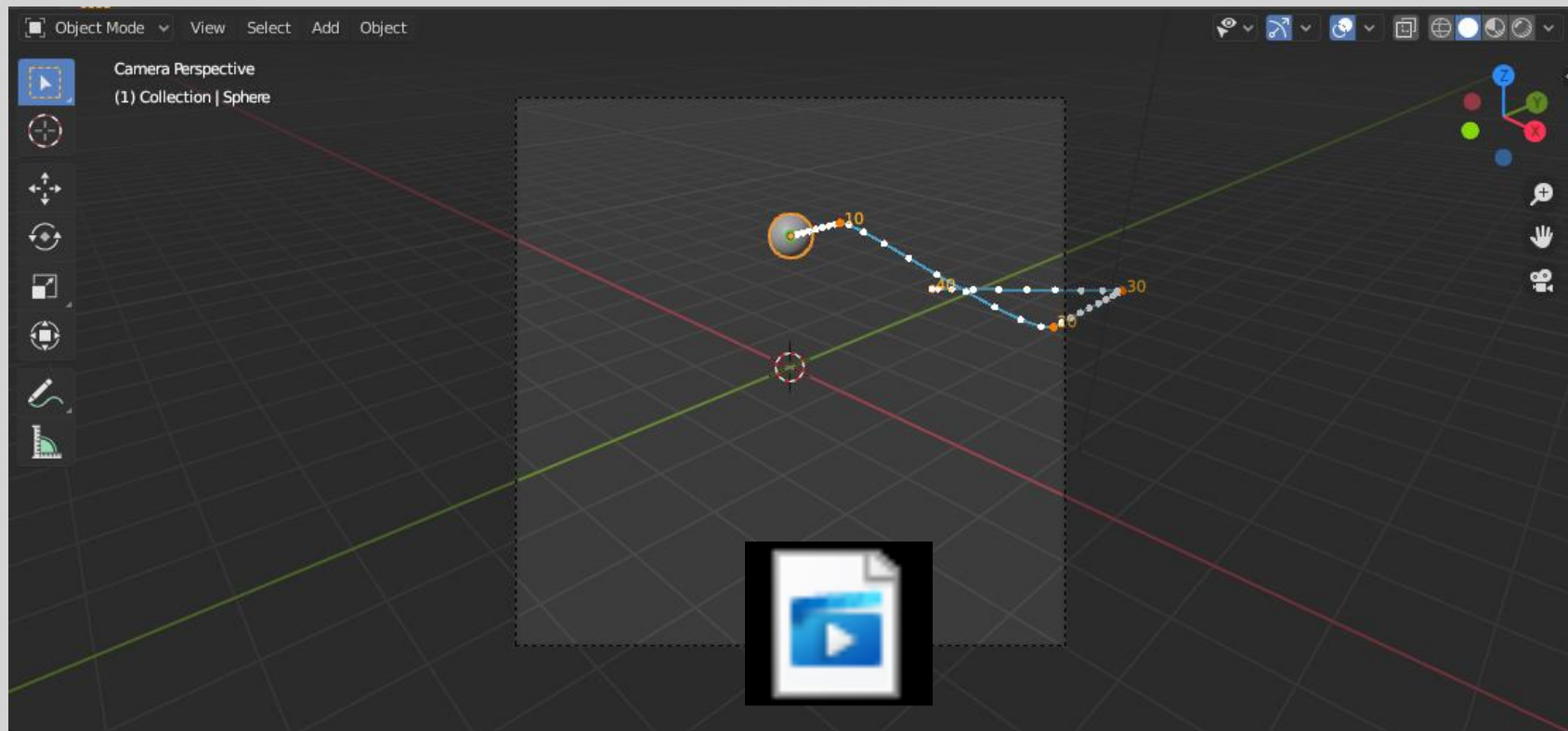
```
rnd = bpy.data.scenes['Scene'].render
rnd.resolution_x = 800
rnd.resolution_y = 800
rnd.resolution_percentage = 100
rnd.image_settings.file_format = 'PNG' # set output format to .png

for frame_nr in range(scene.frame_end - scene.frame_start + 1):
    # set current frame to frame
    scene.frame_set(frame_nr)
    # set output path so render won't get overwritten
    rnd.filepath = os.path.join(render_folder, 'moving_sphere_'+
str(frame_nr)+'.png')
    # Render image
    bpy.ops.render.render(write_still=True)

# restore the filepath
scene.render.filepath = fp
```

Blender/Python API

Keyframe Rendering

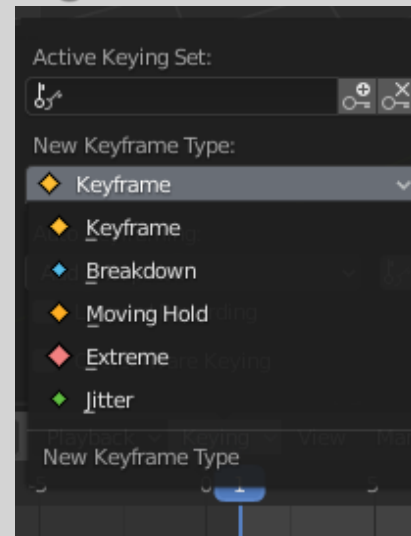


keySphereMove.mp4

Blender/Python API

Keyframe Rendering

- **Keyframe** (yellow diamond)
Normal keyframe.
- **Breakdown** (cyan small diamond)
Breakdown state. e.g. for transitions between key poses.
- **Moving Hold** (slight orange diamond)
A keyframe that adds a small amount of motion around a holding pose. In the Dope Sheet it will also draw a bar between them.
- **Extreme** (red big diamond)
An 'extreme' state, or some other purpose as needed.
- **Jitter** (green tiny diamond)
A filler or baked keyframe for keying on ones, or some other purpose as needed.





Blender/Python API

Distance Travelled by Object

```
# To have a text showing the distance travelled by 'Sphere'
import bpy

positions = (0,0,0), (0,5,0), (5,5,0), (5,0,0), (0,0,0)

scene = bpy.context.scene
bpy.ops.mesh.primitive_uv_sphere_add(radius=0.3, location = (0,0,0) )
bpy.ops.object.shade_smooth()
obj = bpy.context.active_object

def recalculate_text(scene):
    x = scene.objects['Sphere'].location[0]
    print('Distance in x-direction: {0:.1f} meters'.format(x))

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

frame_num = 0
for position in positions:
    bpy.context.scene.frame_set(frame_num)
    obj.location = position
    obj.keyframe_insert(data_path="location", index = -1)
    frame_num += 10
```



The screenshot displays the Blender 2.80 interface with the following components:

- Scripting Editor:** A Python script named `tryKey_square.py` is shown. It defines a `recalculate_text` function that updates the location of a sphere object in a scene. The script uses `bpy.ops.mesh.primitive_uv_sphere_add` to create a sphere and `bpy.ops.object.shade_smooth` to smooth its shading. The `recalculate_text` function calculates the distance in the x-direction and updates a text property on the sphere. The script is executed using `bpy.app.handlers.frame_change_pre.append(recalculate_text)`.
- Outliner:** Shows a collection named `(31) Collection | Sphere` containing a sphere object.
- Properties Panel:** Shows the sphere's location (X, Y, Z) and other properties.
- Console:** Displays the output of the script, showing the sphere's location and the execution of the `recalculate_text` function.

```
1 import bpy
2
3 positions = (0,0,0),(0,5,0),(5,5,0),(5,0,0),(0,0,0)
4 start_pos = (0,0,0)
5
6 scene = bpy.context.scene
7
8 bpy.ops.mesh.primitive_uv_sphere_add(segments=32, radius=0.3, location=start_pos)
9 bpy.ops.object.shade_smooth()
10 obj = bpy.context.active_object
11
12 def recalculate_text(scene):
13     x = scene.objects['Sphere'].location[0]
14     print('Distance in x-direction: {0:1f} meters'.format(x))
15
16 bpy.app.handlers.frame_change_pre.append(recalculate_text)
17
18 frame_num = 0
19
20 for position in positions:
21     bpy.context.scene.frame_set(frame_num)
22     obj.location = position
23     obj.keyframe_insert(data_path="location", index=-1)
24     frame_num += 10
25
26
27
```

PYTHON INTERACTIVE CONSOLE 3.7.4 (default, Feb 17 2020, 16:23:28) [MSC v. 1916 64 bit (AMD64)]

Built-in Modules: bpy, bpy.data, bpy.ops, bpy.props, bpy.types, bpy.context, bpy.utils, bgl, blf, mathutils
Convenience Imports: from mathutils import *; from math import *
Convenience Variables: C = bpy.context, D = bpy.data

```
>>>
>>> bpy.ops.mesh.primitive_uv_sphere_add(segments=32, radius=0.3, enter_editmode=False, align='WORLD', location=(0, 0, 0))
>>> bpy.ops.object.shade_smooth()
>>> bpy.ops.text.run_script()
>>> bpy.context.scene.frame_end = 40
```

File: C:\Lectures\BCO 602 Animasyon İcin Betik Dilleri\Hatta_07\tryKey_square.py

Collection | Sphere | Verts:482 | Faces:512 | Tris:960 | Objects:1/3 | Mem: 18.0 MiB | 2.83.9



```
Distance in x-direction: 0.0 meters
Distance in x-direction: 0.0 meters
Distance in x-direction: 0.0 meters
Distance in x-direction: 0.0 meters
Distance in x-direction: 0.0 meters
Distance in x-direction: 0.0 meters
Distance in x-direction: 0.0 meters
Distance in x-direction: 0.0 meters
Distance in x-direction: 0.0 meters
Distance in x-direction: 0.0 meters
Distance in x-direction: 0.1 meters
Distance in x-direction: 0.5 meters
Distance in x-direction: 1.1 meters
Distance in x-direction: 1.8 meters
Distance in x-direction: 2.5 meters
Distance in x-direction: 3.2 meters
Distance in x-direction: 3.9 meters
Distance in x-direction: 4.5 meters
Distance in x-direction: 4.9 meters
Distance in x-direction: 5.0 meters
Distance in x-direction: 5.0 meters
Distance in x-direction: 5.0 meters
Distance in x-direction: 5.0 meters
Distance in x-direction: 5.0 meters
Distance in x-direction: 5.0 meters
Distance in x-direction: 5.0 meters
Distance in x-direction: 5.0 meters
```



Blender/Python API

Midterm Exam III: Key-Frame Animation



This is the third part of mid-term exam. Write a Blender script that creates 20x20 random colored and random height (z-axis). Then you set for each cube a key frame random z position.



Blender/Python API

Midterm Exam III: Key-Frame Animation

