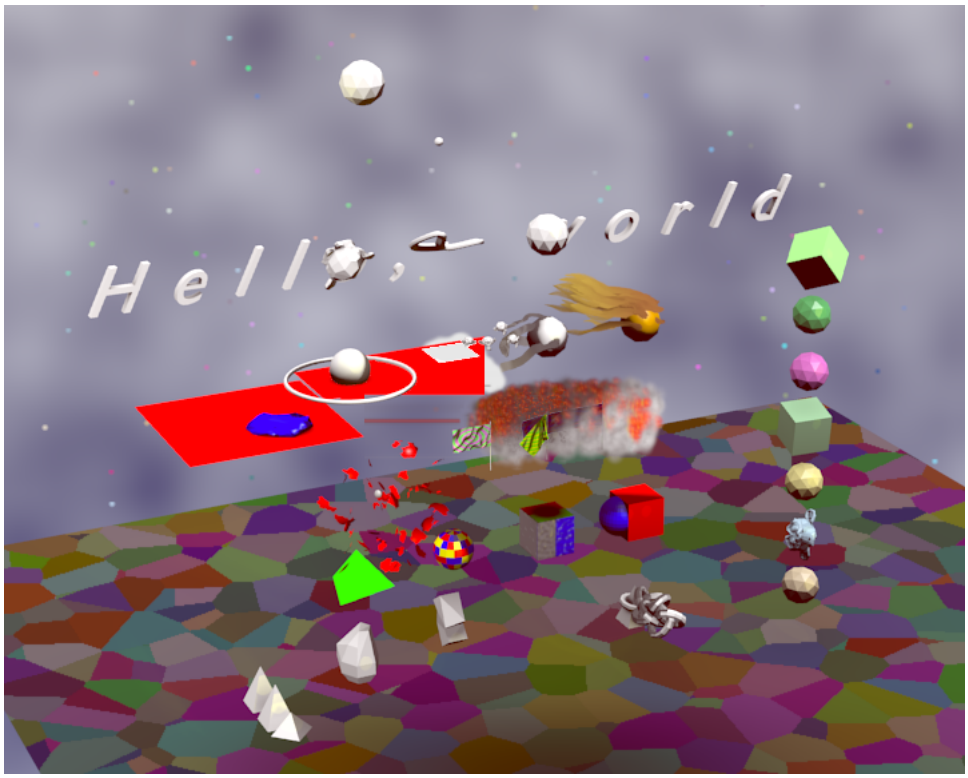
Code snippets.
Introduction to Python scripting for Blender 2.5x.
Third edition, expanded and
updated for Blender 2.58

Thomas Larsson

June 26, 2011



1 Introduction

With the arrival of the Blender 2.5x versions of Blender, Python scripting is taken to a new level. Whereas the Python API up to Blender 2.49 was quite incomplete and ad hoc, the API in Blender 2.5x promises to allow for Python access to all Blender features, in a complete and systematic way.

However, the learning curve for this amazing tool can be quite steep. The purpose of these notes is to simplify the learning process, by providing example scripts that illustrate various aspects of Python scripting in Blender.

Blender is still under heavy development, and the Python API is not yet quite stable. During the few months that passed between the two first editions of these notes, the Python API underwent a complete overhaul, breaking all old scripts. The differences between the second edition (for Blender 2.54.0) and the third edition (for Blender 2.57.0) are much less dramatic. However, also minor changes in the API can make the scripts stop working. The scripts in these notes have been tested with Blender 2.58.0 rev 37702 (this information is available on the splash screen).

Since Blender 2.57 is advertised as the first stable release, there is some hope that the API can remain stable in the future. Hence there is a fair chance that the scripts in these notes will continue to work for a long time, but there are no guarantees.

The covered subjects fall into the following categories:

- Data creation and manipulation. Most of the programs are not very useful, but only constructed to illustrate the concepts.
- Custom properties.
- User interfaces: panels, buttons and menus.
- Turning scripts into Blender add-ons, which can be loaded automatically when Blender starts.
- Scripts distributed over several files.
- Simulations of particles, hair, cloth, softbodies, smoke, fluids, etc.
- Nodes

1.1 Running the scripts

Each script example, with the exception of the multi-file packages, is a complete program. It can be copied from this pdf file and pasted into the Text editor in Blender, which is found on the Scripting screen. Execute the script by pressing the Run button, or press Alt-P on your keyboard.

The scripts are also available as separate Python files, located in the scripts folder which should have come bundled with this file. Just load the Python file into the Text editor (Alt-O), and run it. There is also a batch script which runs many of the other scripts at once. It is described in detail in the last section.

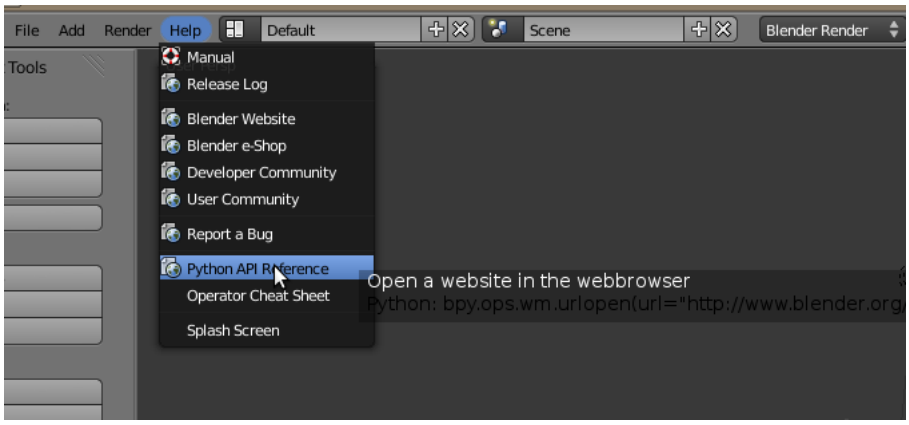
It is assumed that the scripts are located in the `~/snippets/scripts` folder, where ~ is your home directory (e.g. `/home/thomas` on Linux, `C:\Documents and Settings\users\thomas` on Windows XP, or `C:\Users\thomas` on Windows Vista. The scripts can be placed anywhere, but path names in some Python files (batch.py, texture.py and uvs.py) must be modified accordingly. Python will inform you if it fails to find the relevant files.

It is possible to make a batch run of all scripts in the object and simulation folders by loading and executing the file batch.py. We can easily confirm that all scripts work correctly (or at least that they don't generate any errors) by running the batch script. If there are problems, look in the console window for further information.

1.2 Getting more information

The example scripts only scratch on the surface of what can be done with Python scripting in Blender 2.5x. When you write your own scripts, you will certainly want to access operators and variables not mentioned here. There are several ways to obtain this information.

- The main source of information is the Blender Python documentation. It can be found on-line at `http://www.blender.org/documentation/250PythonDoc/contents.html`. It can be conveniently accessed from the help menu in Blender.

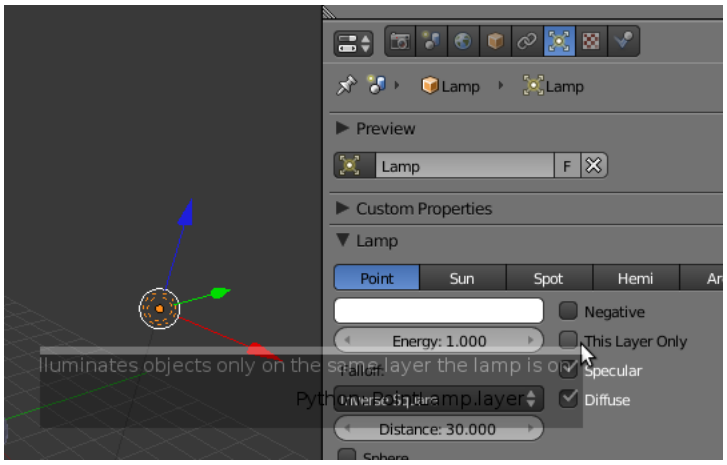


- There is also an official scripting tutorial called *Blender 3D: Noob to Pro/Advanced Tutorials/Blender 2.5x Scripting/Introduction*, located at `http://en.wikibooks.org/wiki/Blender_3D:_Noob_to_Pro/Advanced_Tutorials/Python_Scripting/Introduction_New`.

- Use tooltips. E.g., hovering over the "This Layer Only" option in the Lamp context reveals the following text[1]:

  ```
  Illuminates objects only on the same layer the lamp is on.
  Python: PointLamp.layer
  ```

  From this we conclude that this option is accessed as `lamp.layer`, where `lamp` is the data of the active object, i.e. `lamp = bpy.context.object.data`

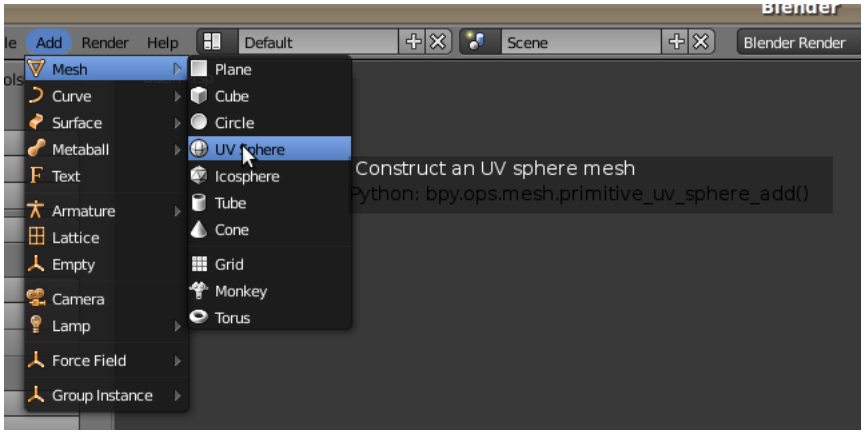


- There are tooltips also when adding

[1] Unfortunately, the help text disappears when the print-screen button is pressed. The help texts in the pictures have been added manually afterwards, and look slightly different from how they appear on screen.

```
Construct an UV sphere mesh
Python:bpy.ops.primitive_uv_sphere_add()
```

This tells us what operator call is needed to add a primitive UV sphere mesh.



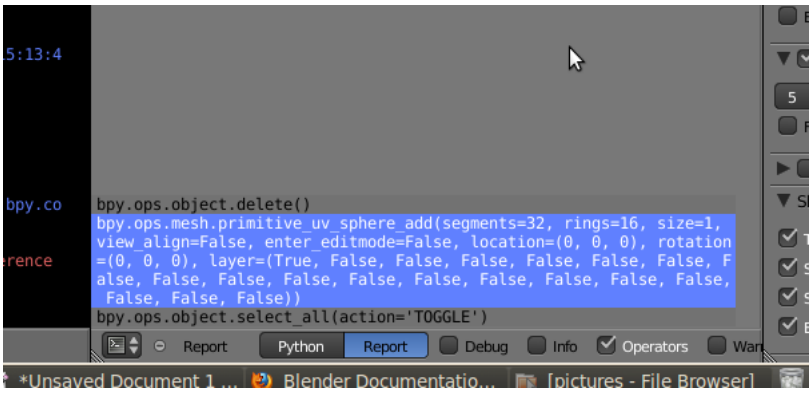
- Once an operator has been executed, it leaves a trail in the report window in the scripting screen.

```
bpy.ops.mesh.primitive_uv_sphere_add(segments=32, rings=16,
size=1, view_align=False, enter_editmode=False,
location=(0, 0, 0), rotation=(0, 0, 0), layer=(True, False, False,
False, False, False, False, False, False, False, False, False, False,
False, False, False, False, False, False, False))
```

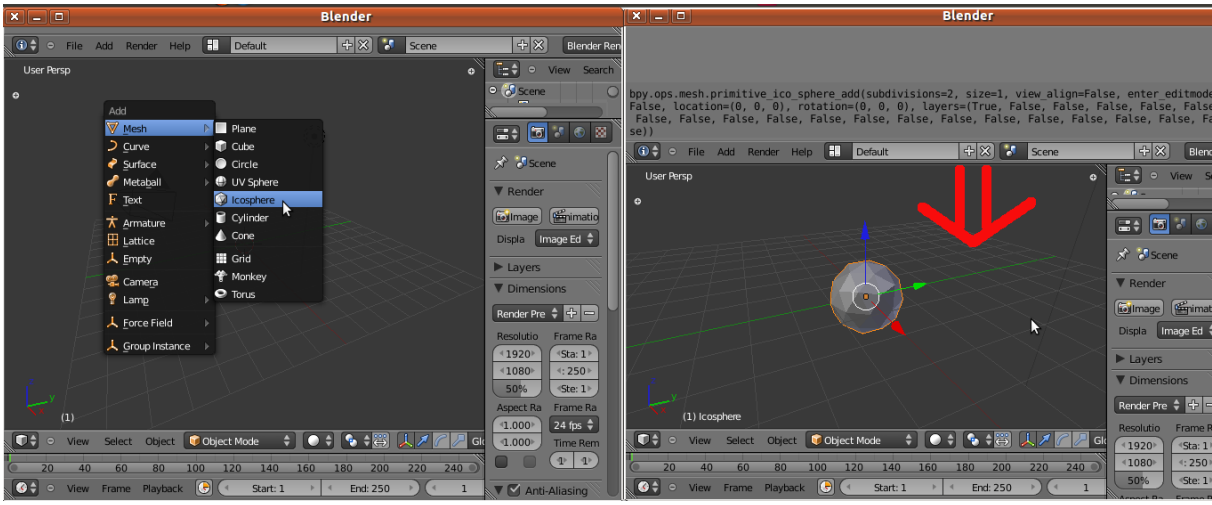
When we add an UV sphere from the menu, it always has 32 segments, 16 rings, etc. But it is straightforward to figure out which call we need to make a sphere with other data, e.g. 12 segments, 6 rings, radius 3, and centered at $(1, 1, 1)$:

```
bpy.ops.mesh.primitive_uv_sphere_add(
    segments=12,
    rings=6,
    size=3,
    enter_editmode=True,
    location=(1, 1, 1))
```

Only operator execution is recorded in the report window, and not e.g. setting a value.



In recent Blender versions the scripting trail is instead printed in the info window, which can be revealed by pulling down the top menubar.



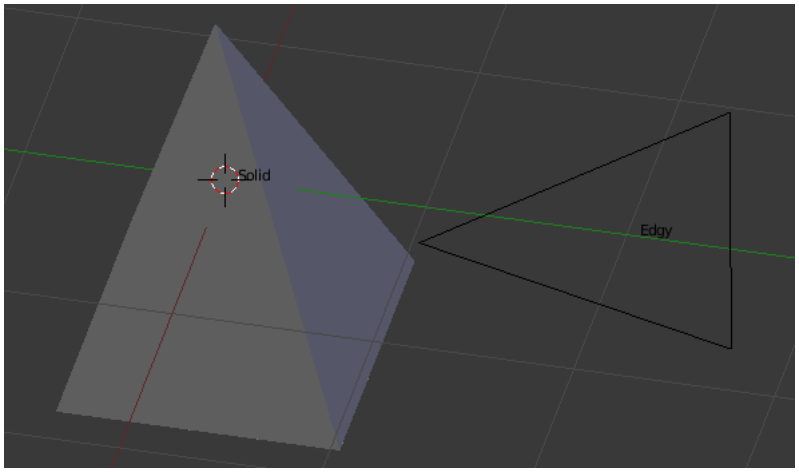
- Learn from other people's code. The scripts that come bundled with Blender are a great source of inspiration.
- There is a thriving on-line community at the Python and scripting subforum at Blenderartists.org. The URL is `http://blenderartists.org/forum/forumdisplay.php?f=11`.

2 Meshes and armatures

2.1 Mesh

This program creates two meshes. The first is a solid pyramid, with both triangular and quad faces. The second is a wire triangle. The names of both

meshes are displayed. The triangle is translated away so it can be seen beside the pyramid. This requires that it is selected.

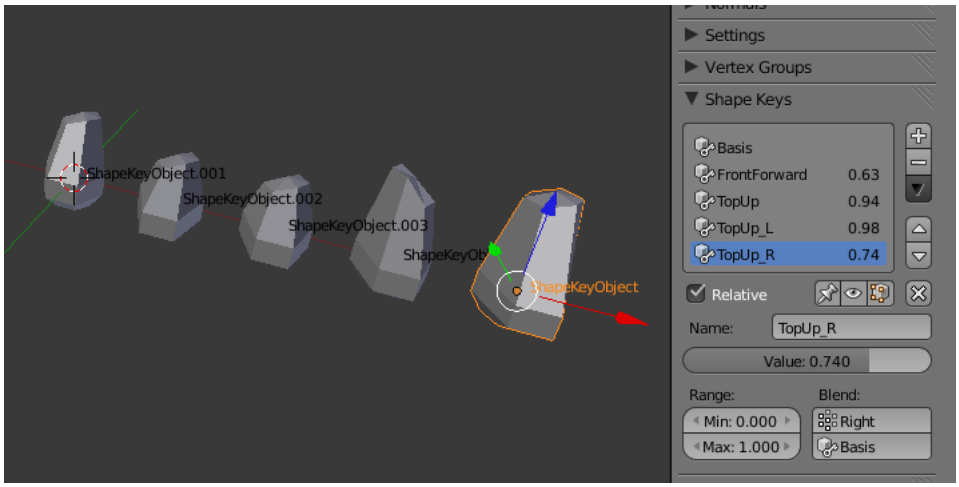



```
#----------------------------------------------------------
# File meshes.py
#----------------------------------------------------------
import bpy

def createMesh(name, origin, verts, edges, faces):
    # Create mesh and object
    me = bpy.data.meshes.new(name+'Mesh')
    ob = bpy.data.objects.new(name, me)
    ob.location = origin
    ob.show_name = True
    # Link object to scene
    bpy.context.scene.objects.link(ob)

    # Create mesh from given verts, edges, faces. Either edges or
    # faces should be [], or you ask for problems
    me.from_pydata(verts, edges, faces)

    # Update mesh with new data
    me.update(calc_edges=True)
    return ob

def run(origin):
    (x,y,z) = (0.707107, 0.258819, 0.965926)
    verts1 = ((x,x,-1), (x,-x,-1), (-x,-x,-1), (-x,x,-1), (0,0,1))
    faces1 = ((1,0,4), (4,2,1), (4,3,2), (4,0,3), (0,1,2,3))
    ob1 = createMesh('Solid', origin, verts1, [], faces1)
```

```
    verts2 = ((x,x,0), (y,-z,0), (-z,y,0))
    edges2 = ((1,0), (1,2), (2,0))
    ob2 = createMesh('Edgy', origin, verts2, edges2, [])

    # Move second object out of the way
    ob1.select = False
    ob2.select = True
    bpy.ops.transform.translate(value=(0,2,0))
    return

if __name__ == "__main__":
    run((0,0,0))
```

2.2 Vertex groups and shapekeys

This program adds an UV sphere with two vertex groups (Left and Right) and four shapekeys.



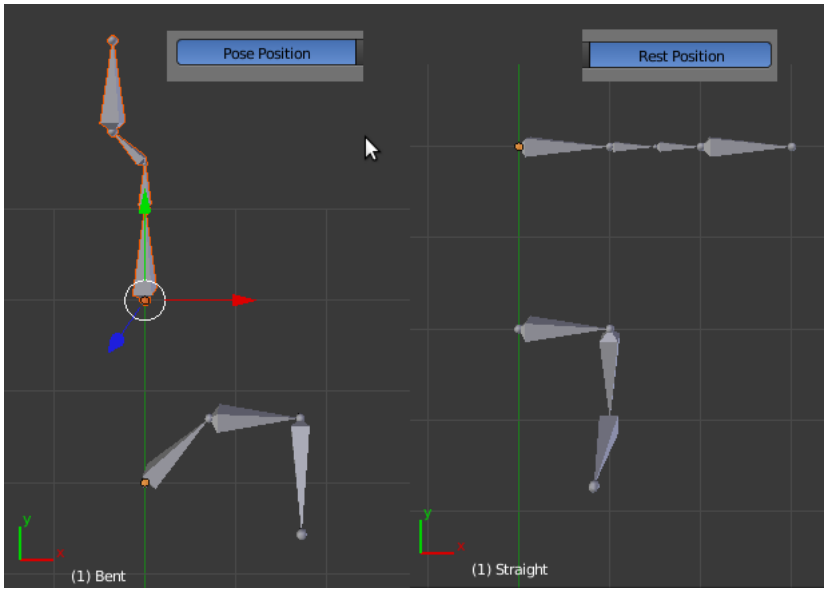

```
#----------------------------------------------------------
# File shapekey.py
#----------------------------------------------------------
import bpy, random

def run(origin):
    # Add UV sphere
    bpy.ops.mesh.primitive_uv_sphere_add(
        segments=6, ring_count=5, size=1, location=origin)
    ob = bpy.context.object
```

```
ob.name = 'ShapeKeyObject'
ob.show_name = True

# Create Left and Right vertex groups
left = ob.vertex_groups.new('Left')
right = ob.vertex_groups.new('Right')
for v in ob.data.vertices:
    if v.co[0] > 0.001:
        left.add([v.index], 1.0, 'REPLACE')
    elif v.co[0] < -0.001:
        right.add([v.index], 1.0, 'REPLACE')
    else:
        left.add([v.index], 0.5, 'REPLACE')
        right.add([v.index], 0.5, 'REPLACE')

# Add Basis key
bpy.ops.object.shape_key_add(None)
basis = ob.active_shape_key

# Add FrontForward key: front verts move one unit forward
# Slider from -1.0 to +2.0
bpy.ops.object.shape_key_add(None)
frontFwd = ob.active_shape_key
frontFwd.name = 'FrontForward'
frontFwd.slider_min = -1.0
frontFwd.slider_max = 2.0
for v in [19, 20, 23, 24]:
    pt = frontFwd.data[v].co
    pt[1] = pt[1] - 1

# Add TopUp keys: top verts move one unit up.  TopUp_L and
# TopUp_R only affect left and right halves, respectively
keylist = [(None, ''), ('Left', '_L'), ('Right', '_R')]
for (vgrp, suffix) in keylist:
    bpy.ops.object.shape_key_add(None)
    topUp = ob.active_shape_key
    topUp.name = 'TopUp' + suffix
    if vgrp:
        topUp.vertex_group = vgrp
    for v in [0, 1, 9, 10, 17, 18, 25]:
        pt = topUp.data[v].co
        pt[2] = pt[2] + 1

# Pose shape keys
for shape in ob.data.shape_keys.key_blocks:
    shape.value = random.random()
```

```
    return

if __name__ == "__main__":
    # Create five object with random shapekeys
    for j in range(5):
        run((3*j,0,0))
```

2.3 Armature

This program creates an armature.




```
#----------------------------------------------------
# File armature.py
#----------------------------------------------------
import bpy, math
from mathutils import Vector, Matrix

def createRig(name, origin, boneTable):
    # Create armature and object
    bpy.ops.object.add(
        type='ARMATURE',
        enter_editmode=True,
        location=origin)
    ob = bpy.context.object
    ob.show_x_ray = True
```

```
    ob.name = name
    amt = ob.data
    amt.name = name+'Amt'
    amt.show_axes = True

    # Create bones
    bpy.ops.object.mode_set(mode='EDIT')
    for (bname, pname, vector) in boneTable:
        bone = amt.edit_bones.new(bname)
        if pname:
            parent = amt.edit_bones[pname]
            bone.parent = parent
            bone.head = parent.tail
            bone.use_connect = False
            (trans, rot, scale) = parent.matrix.decompose()
        else:
            bone.head = (0,0,0)
            rot = Matrix.Translation((0,0,0)) # identity matrix
        bone.tail = Vector(vector) * rot + bone.head
    bpy.ops.object.mode_set(mode='OBJECT')
    return ob

def poseRig(ob, poseTable):
    bpy.context.scene.objects.active = ob
    bpy.ops.object.mode_set(mode='POSE')
    deg2rad = 2*math.pi/360

    for (bname, axis, angle) in poseTable:
        pbone = ob.pose.bones[bname]
        # Set rotation mode to Euler XYZ, easier to understand
        # than default quaternions
        pbone.rotation_mode = 'XYZ'
        # Documentation bug: Euler.rotate(angle,axis):
        # axis in ['x','y','z'] and not ['X','Y','Z']
        pbone.rotation_euler.rotate_axis(axis, angle*deg2rad)
    bpy.ops.object.mode_set(mode='OBJECT')
    return

def run(origo):
    origin = Vector(origo)
    # Table of bones in the form (bone, parent, vector)
    # The vector is given in local coordinates
    boneTable1 = [
        ('Base', None, (1,0,0)),
        ('Mid', 'Base', (1,0,0)),
        ('Tip', 'Mid', (0,0,1))
```

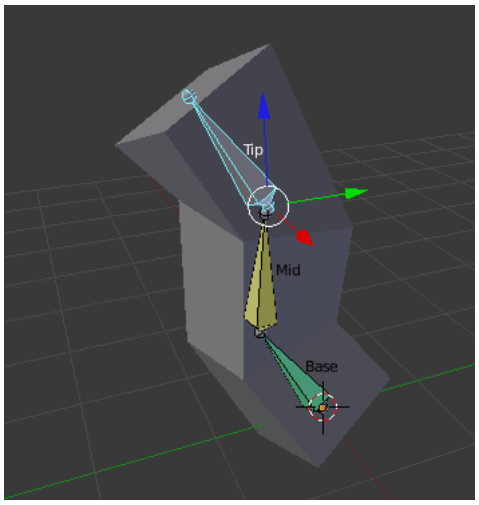
```
    ]
    bent = createRig('Bent', origin, boneTable1)

    # The second rig is a straight line, i.e. bones run along local Y axis
    boneTable2 = [
        ('Base', None, (1,0,0)),
        ('Mid', 'Base', (0,0.5,0)),
        ('Mid2', 'Mid', (0,0.5,0)),
        ('Tip', 'Mid2', (0,1,0))
    ]
    straight = createRig('Straight', origin+Vector((0,2,0)), boneTable2)

    # Pose second rig
    poseTable2 = [
        ('Base', 'X', 90),
        ('Mid2', 'Z', 45),
        ('Tip', 'Y', -45)
    ]
    poseRig(straight, poseTable2)

    # Pose first rig
    poseTable1 = [
        ('Tip', 'Y', 45),
        ('Mid', 'Y', 45),
        ('Base', 'Y', 45)
    ]
    poseRig(bent, poseTable1)
    return

if __name__ == "__main__":
    run((0,5,0))
```

2.4 Rigged mesh

This program adds an armature and a mesh. The armature has three bones (Base, Mid, Tip) and constraints:

1. An IK constraint Mid → Tip.
2. A Stretch_To constraint Mid → Tip.
3. A Copy_Rotation constraint Base → Tip.

The mesh is deformed by the armature. Hence an armature modifier and the corresponding vertex groups are created.




```
#----------------------------------------------------------
# File rigged_mesh.py
#----------------------------------------------------------
import bpy, mathutils

def createArmature(origin):
    # Create armature and object
    amt = bpy.data.armatures.new('MyRigData')
    rig = bpy.data.objects.new('MyRig', amt)
    rig.location = origin
    rig.show_x_ray = True
    amt.show_names = True
    # Link object to scene
    scn = bpy.context.scene
    scn.objects.link(rig)
    scn.objects.active = rig
    scn.update()

    # Create bones
    bpy.ops.object.mode_set(mode='EDIT')
    base = amt.edit_bones.new('Base')
    base.head = (0,0,0)
    base.tail = (0,0,1)

    mid = amt.edit_bones.new('Mid')
    mid.head = (0,0,1)
    mid.tail = (0,0,2)
    mid.parent = base
    mid.use_connect = True
```

```
    tip = amt.edit_bones.new('Tip')
    tip.head = (0,0,2)
    tip.tail = (0,0,3)

    # Bone constraints. Armature must be in pose mode.
    bpy.ops.object.mode_set(mode='POSE')

    # IK constraint Mid -> Tip
    pMid = rig.pose.bones['Mid']
    cns1 = pMid.constraints.new('IK')
    cns1.name = 'Ik'
    cns1.target = rig
    cns1.subtarget = 'Tip'
    cns1.chain_count = 1

    # StretchTo constraint Mid -> Tip with influence 0.5
    cns2 = pMid.constraints.new('STRETCH_TO')
    cns2.name = 'Stretchy'
    cns2.target = rig
    cns2.subtarget = 'Tip'
    cns2.influence = 0.5
    cns2.keep_axis = 'PLANE_X'
    cns2.volume = 'VOLUME_XZX'

    # Copy rotation constraints Base -> Tip
    pBase = rig.pose.bones['Base']
    cns3 = pBase.constraints.new('COPY_ROTATION')
    cns3.name = 'Copy_Rotation'
    cns3.target = rig
    cns3.subtarget = 'Tip'
    cns3.owner_space = 'WORLD'
    cns3.target_space = 'WORLD'

    bpy.ops.object.mode_set(mode='OBJECT')
    return rig

def createMesh(origin):
    # Create mesh and object
    me = bpy.data.meshes.new('Mesh')
    ob = bpy.data.objects.new('MeshObject', me)
    ob.location = origin
    # Link object to scene
    scn = bpy.context.scene
    scn.objects.link(ob)
    scn.objects.active = ob
```

```
    scn.update()

    # List of vertex coordinates
    verts = [
        (0.5, 0.5,0), (0.5,-0.5,0), (-0.5,-0.5,0), (-0.5,0.5,0),
        (0.5,0.5,1), (0.5,-0.5,1), (-0.5,-0.5,1), (-0.5,0.5,1),
        (-0.5,0.5,2), (-0.5,-0.5,2), (0.5,-0.5,2), (0.5,0.5,2),
        (0.5,0.5,3), (0.5,-0.5,3), (-0.5,-0.5,3), (-0.5, 0.5,3)
    ]
    # List of faces.
    faces = [
        (0, 1, 2, 3),
        (0, 4, 5, 1),
        (1, 5, 6, 2),
        (2, 6, 7, 3),
        (4, 0, 3, 7),
        (4, 7, 8, 11),
        (7, 6, 9, 8),
        (6, 5, 10, 9),
        (5, 4, 11, 10),
        (10, 11, 12, 13),
        (9, 10, 13, 14),
        (8, 9, 14, 15),
        (11, 8, 15, 12),
        (12, 15, 14, 13)
    ]

    # Create mesh from given verts, edges, faces. Either edges or
    # faces should be [], or you ask for problems
    me.from_pydata(verts, [], faces)

    # Update mesh with new data
    me.update(calc_edges=True)
    return ob

def skinMesh(ob, rig):
    # List of vertex groups, in the form (vertex, weight)
    vgroups = {}
    vgroups['Base'] = [
        (0, 1.0), (1, 1.0), (2, 1.0), (3, 1.0),
        (4, 0.5), (5, 0.5), (6, 0.5), (7, 0.5)]
    vgroups['Mid'] = [
        (4, 0.5), (5, 0.5), (6, 0.5), (7, 0.5),
        (8, 1.0), (9, 1.0), (10, 1.0), (11, 1.0)]
    vgroups['Tip'] = [(12, 1.0), (13, 1.0), (14, 1.0), (15, 1.0)]
```

```
    # Create vertex groups, and add verts and weights
    # First arg in assignment is a list, can assign several verts at once
    for name in vgroups.keys():
        grp = ob.vertex_groups.new(name)
        for (v, w) in vgroups[name]:
            grp.add([v], w, 'REPLACE')

    # Give mesh object an armature modifier, using vertex groups but
    # not envelopes
    mod = ob.modifiers.new('MyRigModif', 'ARMATURE')
    mod.object = rig
    mod.use_bone_envelopes = False
    mod.use_vertex_groups = True

    return

def run(origin):
    rig = createArmature(origin)
    ob = createMesh(origin)
    skinMesh(ob, rig)

    # Move and rotate the tip bone in pose mode
    bpy.context.scene.objects.active = rig
    bpy.ops.object.mode_set(mode='POSE')
    ptip = rig.pose.bones['Tip']
    ptip.location = (0.2,-0.5,0)
    rotMatrix = mathutils.Matrix.Rotation(0.6, 3, 'X')
    ptip.rotation_quaternion = rotMatrix.to_quaternion()

    return

if __name__ == "__main__":
    run((0,0,0))
```

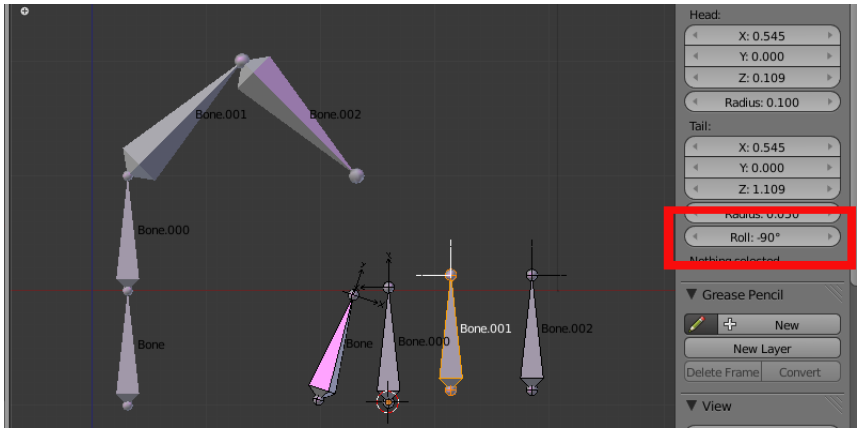
2.5 Edit mode versus pose mode

Bone attributes which affect the rest pose of an armature (head, tail, roll, parent, use_connect, etc.) are only available in edit mode (using a bone in `ob.data.edit_bones`), whereas attributes which involve posing require the the armature is in pose mode (using a bone in `ob.pose.bones`). To my knowledge, the only way to switch between edit and pose mode is with the operator calls

```
    bpy.ops.object.mode_set(mode='EDIT')
    bpy.ops.object.mode_set(mode='POSE')
```

Since operators act on the active object, we must ensure that the right object is active by setting `bpy.context.scene.objects.active`.

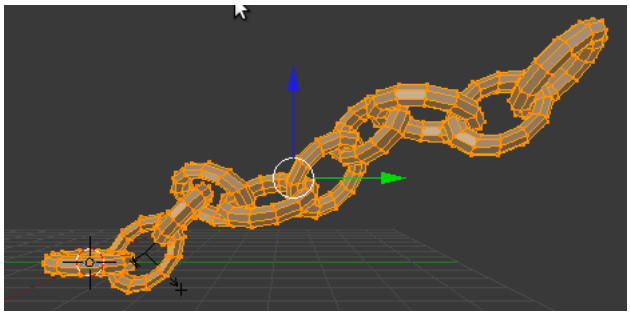
This script copies the roll angles from a source rig (object name 'SrcRig') to a target rig (object name 'TrgRig'). Both armatures must have the same number of bones with identical names.




```
#----------------------------------------------------------
# File copy_roll.py
#----------------------------------------------------------
import bpy

def copyRolls(src, trg):
    rolls = {}

    bpy.context.scene.objects.active = src
    bpy.ops.object.mode_set(mode='EDIT')
    for eb in src.data.edit_bones:
        rolls[eb.name] = eb.roll
    bpy.ops.object.mode_set(mode='POSE')

    bpy.context.scene.objects.active = trg
    bpy.ops.object.mode_set(mode='EDIT')
    for eb in trg.data.edit_bones:
        oldRoll = eb.roll
        eb.roll = rolls[eb.name]
        print(eb.name, oldRoll, eb.roll)
    bpy.ops.object.mode_set(mode='POSE')
    return

objects = bpy.context.scene.objects
copyRolls(objects['SrcRig'], objects['TrgRig'])
```

2.6 Applying an array modifier

This program creates a chain with ten links. A link is a primitive torus scaled along the x axis. We give the link an array modifier, where the offset is controlled by an empty. Finally the array modifier is applied, making the chain into a single mesh.



```
#----------------------------------------------------------
# File chain.py
# Creates an array modifier and applies it
#----------------------------------------------------------
import bpy
import math
from math import pi

def run(origin):
    # Add single chain link to the scene
    bpy.ops.mesh.primitive_torus_add(
        #major_radius=1,
        #minor_radius=0.25,
        major_segments=12,
        minor_segments=8,
        use_abso=True,
        abso_major_rad=1,
        abso_minor_rad=0.6,
        location=(0,0,0),
        rotation=(0,0,0))

    # Scale the torus along the x axis
    ob = bpy.context.object
    ob.scale = (0.7, 1, 1)
    bpy.ops.object.transform_apply(scale=True)

    # Create an empty
    bpy.ops.object.add(
```

```
        type='EMPTY',
        location=(0,1.2,0.2),
        rotation=(pi/2, pi/4, pi/2))
    empty = bpy.context.object

    # Make chain link active again
    scn = bpy.context.scene
    scn.objects.active = ob

    # Add modifier
    mod = ob.modifiers.new('Chain', 'ARRAY')
    mod.fit_type = 'FIXED_COUNT'
    mod.count = 10
    mod.use_relative_offset = 0
    mod.use_object_offset = True
    mod.offset_object = empty

    # Apply the modifier
    bpy.ops.object.visual_transform_apply()
    bpy.ops.object.modifier_apply(apply_as='DATA', modifier='Chain')

    # Move chain into place
    bpy.ops.transform.translate(value=origin)

    # Don't need empty anymore
    scn.objects.unlink(empty)
    del(empty)

    return

if __name__ == "__main__":
    run((0,3,0))
```

3 Three ways to create objects

The examples covered so far show that object can be created from Python using different paradigms.

3.1 Data method

The data method closely mimics how data are stored internally in Blender.

1. Add the data, and then the object. For a mesh:

```
me = bpy.data.meshes.new(meshName)
ob = bpy.data.objects.new(obName, me)
```

   and for an armature:

```
amt = bpy.data.armatures.new(amtname)
ob = bpy.data.objects.new(obname, amt)
```

2. Link the object to the current scene and make it active. Optionally, we can make the newly created object active or selected. This code is the same for all kinds of objects.

```
scn = bpy.context.scene
scn.objects.link(ob)
scn.objects.active = ob
ob.select = True
```

3. Fill in the data. In the mesh case, we add the lists of vertices and faces.

```
me.from_pydata(verts, [], faces)
```

   In the armature case, we switch to edit mode and add a bone.

```
bpy.ops.object.mode_set(mode='EDIT')
bone = amt.edit_bones.new('Bone')
bone.head = (0,0,0)
bone.tail = (0,0,1)
```

4. Finally, it is usually necessary to update the modified data. In the mesh case, we call an update function explicity.

```
me.update()
```

   The armature is implicitly update when we switch to object mode.

```
bpy.ops.object.mode_set(mode='OBJECT')
```

3.2 Operator method

The operator method adds an object and a data block at the same time. The data block is currently empty, and needs to be filled with actual data later.

1. Add the object with the `bpy.ops.object.add` operator. This automatically takes care of several things that we had to do manually in the data method: it creates object data (i.e. the mesh or armature), links the object to the scene, makes it active and selects the object. On the other hand, we must now retrieve the object and its data. This is straightforward because `bpy.context.data` always points to the active object.

   To add a mesh object, we do

   ```
   bpy.ops.object.add(type='MESH')
   ob = bpy.context.object
   me = ob.data
   ```

   and to add an armature:

   ```
   bpy.ops.object.add(
       type='ARMATURE',
       enter_editmode=True,
       location=origin)
   ob = bpy.context.object
   amt = ob.data
   ```

2. As in the data method, the actual data must be filled in and updated before use. For a mesh we add the verts and faces:

   ```
   me.from_pydata(verts, [], faces)
   me.update()
   ```

   and for an armature we add a bone:

   ```
   bone = amt.edit_bones.new('Bone')
   bone.head = (0,0,0)
   bone.tail = (0,0,1)
   bpy.ops.object.mode_set(mode='OBJECT')
   ```

   Note that we do not need to explicitly enter edit mode, because the armature entered edit mode already on creation.

3.3 Primitive method

If we want to make an object of a primitive type, there may exist an operator which creates the primitive with the desired properties.

1. To create a pyramid mesh

```
bpy.ops.mesh.primitive_cone_add(
    vertices=4,
    radius=1,
    depth=1,
    cap_end=True)
```

whereas the following code adds a armature with a single bone;

```
bpy.ops.object.armature_add()
bpy.ops.transform.translate(value=origin)
```

2. As in the operator method, we then retrieve the newly create object from `bpy.context.object`.

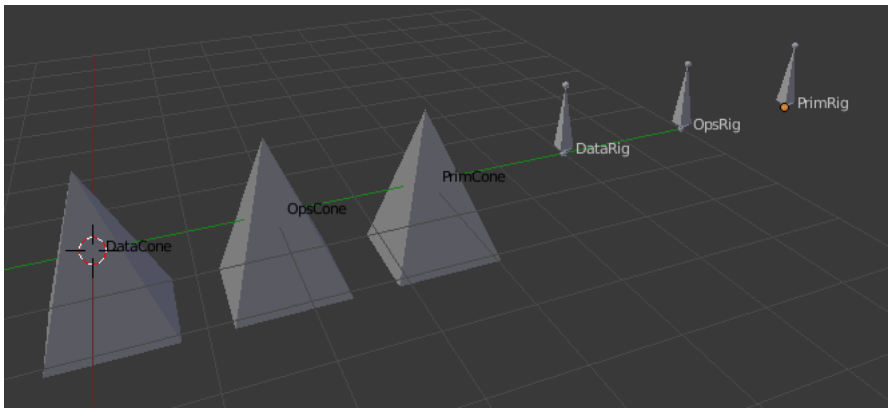
```
ob = bpy.context.object
me = ob.data
```

3.4 Comparison

The primitive method is simplest, but it only works when a suitable primitive is available. Even in the example program, it creates a pyramid mesh which is slightly different from the other two methods; the base is not a single quad, but rather consists of four triangles with a common point in the middle of the base. The other two methods are more or less equivalent.

A primitive does not need to be particularly simple; there are primitives for creating a monkey mesh or a human rig. But the primitive method is always limited to prefabricated objects.

We use all three methods in the examples in this note.



```
#----------------------------------------------------------
# File objects.py
#----------------------------------------------------------
import bpy
import mathutils
from mathutils import Vector

def createMeshFromData(name, origin, verts, faces):
    # Create mesh and object
    me = bpy.data.meshes.new(name+'Mesh')
    ob = bpy.data.objects.new(name, me)
    ob.location = origin
    ob.show_name = True

    # Link object to scene and make active
    scn = bpy.context.scene
    scn.objects.link(ob)
    scn.objects.active = ob
    ob.select = True

    # Create mesh from given verts, faces.
    me.from_pydata(verts, [], faces)
    # Update mesh with new data
    me.update()
    return ob

def createMeshFromOperator(name, origin, verts, faces):
    bpy.ops.object.add(
        type='MESH',
        enter_editmode=False,
        location=origin)
    ob = bpy.context.object
    ob.name = name
    ob.show_name = True
    me = ob.data
    me.name = name+'Mesh'

    # Create mesh from given verts, faces.
    me.from_pydata(verts, [], faces)
    # Update mesh with new data
    me.update()
    # Set object mode
    bpy.ops.object.mode_set(mode='OBJECT')
    return ob

def createMeshFromPrimitive(name, origin):
```

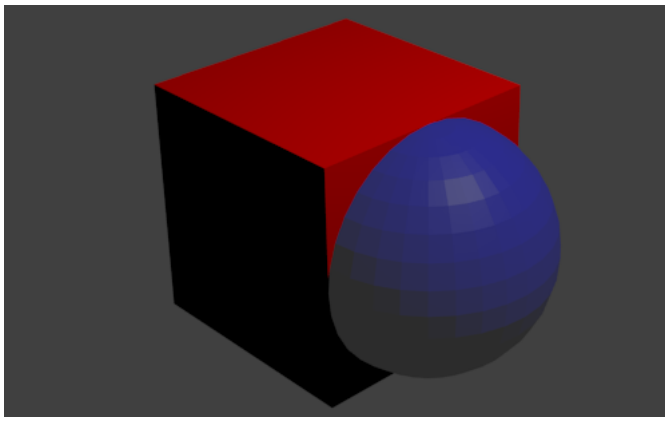
```
    bpy.ops.mesh.primitive_cone_add(
        vertices=4,
        radius=1,
        depth=1,
        cap_end=True,
        view_align=False,
        enter_editmode=False,
        location=origin,
        rotation=(0, 0, 0))

    ob = bpy.context.object
    ob.name = name
    ob.show_name = True
    me = ob.data
    me.name = name+'Mesh'
    return ob

def createArmatureFromData(name, origin):
    # Create armature and object
    amt = bpy.data.armatures.new(name+'Amt')
    ob = bpy.data.objects.new(name, amt)
    ob.location = origin
    ob.show_name = True

    # Link object to scene and make active
    scn = bpy.context.scene
    scn.objects.link(ob)
    scn.objects.active = ob
    ob.select = True

    # Create single bone
    bpy.ops.object.mode_set(mode='EDIT')
    bone = amt.edit_bones.new('Bone')
    bone.head = (0,0,0)
    bone.tail = (0,0,1)
    bpy.ops.object.mode_set(mode='OBJECT')
    return ob

def createArmatureFromOperator(name, origin):
    bpy.ops.object.add(
        type='ARMATURE',
        enter_editmode=True,
        location=origin)
    ob = bpy.context.object
    ob.name = name
    ob.show_name = True
```

```
    amt = ob.data
    amt.name = name+'Amt'

    # Create single bone
    bone = amt.edit_bones.new('Bone')
    bone.head = (0,0,0)
    bone.tail = (0,0,1)
    bpy.ops.object.mode_set(mode='OBJECT')
    return ob

def createArmatureFromPrimitive(name, origin):
    bpy.ops.object.armature_add()
    bpy.ops.transform.translate(value=origin)
    ob = bpy.context.object
    ob.name = name
    ob.show_name = True
    amt = ob.data
    amt.name = name+'Amt'
    return ob

def run(origo):
    origin = Vector(origo)
    (x,y,z) = (0.707107, 0.258819, 0.965926)
    verts = ((x,x,-1), (x,-x,-1), (-x,-x,-1), (-x,x,-1), (0,0,1))
    faces = ((1,0,4), (4,2,1), (4,3,2), (4,0,3), (0,1,2,3))

    cone1 = createMeshFromData('DataCone', origin, verts, faces)
    cone2 = createMeshFromOperator('OpsCone', origin+Vector((0,2,0)), verts, faces)
    cone3 = createMeshFromPrimitive('PrimCone', origin+Vector((0,4,0)))

    rig1 = createArmatureFromData('DataRig', origin+Vector((0,6,0)))
    rig2 = createArmatureFromOperator('OpsRig', origin+Vector((0,8,0)))
    rig3 = createArmatureFromPrimitive('PrimRig', origin+Vector((0,10,0)))
    return

if __name__ == "__main__":
    run((0,0,0))
```

4 Materials and textures

4.1 Materials

This program adds a red, opaque material and a blue, semi-transparent one, add assigns them to a cube an sphere, respectively.

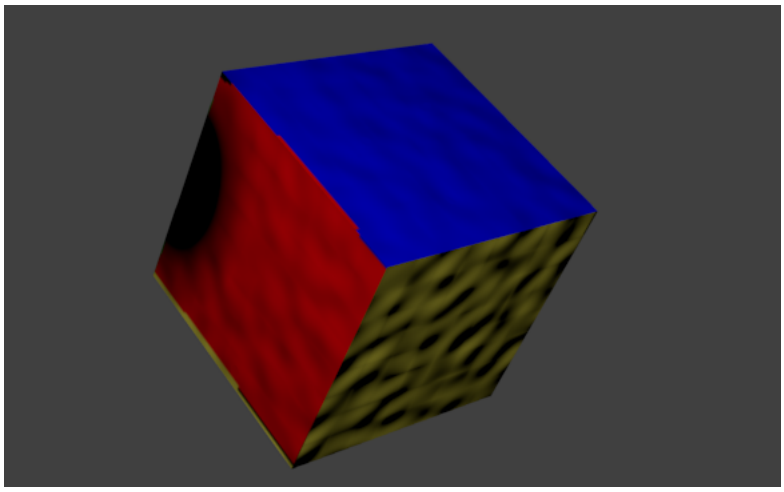


```
#----------------------------------------------------------
# File material.py
#----------------------------------------------------------
import bpy

def makeMaterial(name, diffuse, specular, alpha):
    mat = bpy.data.materials.new(name)
    mat.diffuse_color = diffuse
    mat.diffuse_shader = 'LAMBERT'
    mat.diffuse_intensity = 1.0
    mat.specular_color = specular
    mat.specular_shader = 'COOKTORR'
    mat.specular_intensity = 0.5
    mat.alpha = alpha
    mat.ambient = 1
    return mat

def setMaterial(ob, mat):
    me = ob.data
    me.materials.append(mat)

def run(origin):
    # Create two materials
    red = makeMaterial('Red', (1,0,0), (1,1,1), 1)
    blue = makeMaterial('BlueSemi', (0,0,1), (0.5,0.5,0), 0.5)

    # Create red cube
    bpy.ops.mesh.primitive_cube_add(location=origin)
    setMaterial(bpy.context.object, red)
    # and blue sphere
    bpy.ops.mesh.primitive_uv_sphere_add(location=origin)
```

```
    bpy.ops.transform.translate(value=(1,0,0))
    setMaterial(bpy.context.object, blue)

if __name__ == "__main__":
    run((0,0,0))
```

4.2 Textures

This program creates a material with two textures: an image texture mapped to color and alpha, and a procedural bump texture.



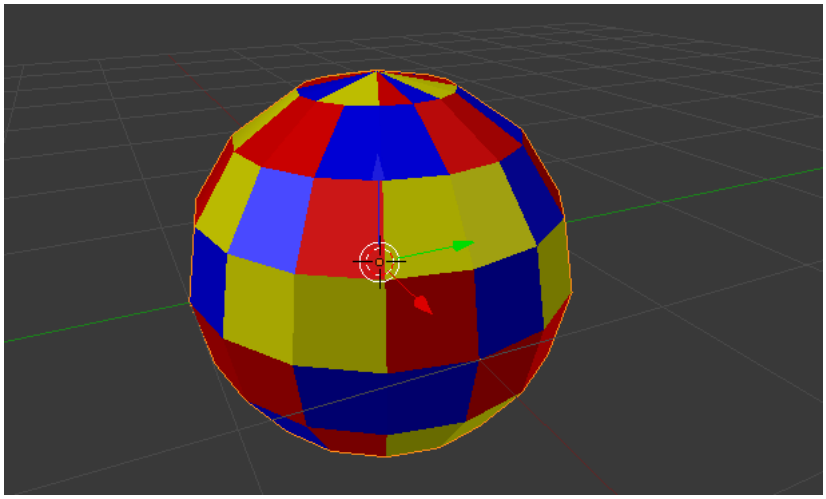
```
#----------------------------------------------------------
# File texture.py
#----------------------------------------------------------
import bpy, os

def run(origin):
    # Load image file. Change here if the snippet folder is
    # not located in you home directory.
    realpath = os.path.expanduser('~/snippets/textures/color.png')
    try:
        img = bpy.data.images.load(realpath)
    except:
        raise NameError("Cannot load image %s" % realpath)

    # Create image texture from image
    cTex = bpy.data.textures.new('ColorTex', type = 'IMAGE')
    cTex.image = img
```

```
# Create procedural texture
sTex = bpy.data.textures.new('BumpTex', type = 'STUCCI')
sTex.noise_basis = 'BLENDER_ORIGINAL'
sTex.noise_scale = 0.25
sTex.noise_type = 'SOFT_NOISE'
sTex.saturation = 1
sTex.stucci_type = 'PLASTIC'
sTex.turbulence = 5

# Create blend texture with color ramp
# Don't know how to add elements to ramp, so only two for now
bTex = bpy.data.textures.new('BlendTex', type = 'BLEND')
bTex.progression = 'SPHERICAL'
bTex.use_color_ramp = True
ramp = bTex.color_ramp
values = [(0.6, (1,1,1,1)), (0.8, (0,0,0,1))]
for n,value in enumerate(values):
    elt = ramp.elements[n]
    (pos, color) = value
    elt.position = pos
    elt.color = color

# Create material
mat = bpy.data.materials.new('TexMat')

# Add texture slot for color texture
mtex = mat.texture_slots.add()
mtex.texture = cTex
mtex.texture_coords = 'UV'
mtex.use_map_color_diffuse = True
mtex.use_map_color_emission = True
mtex.emission_color_factor = 0.5
mtex.use_map_density = True
mtex.mapping = 'FLAT'

# Add texture slot for bump texture
mtex = mat.texture_slots.add()
mtex.texture = sTex
mtex.texture_coords = 'ORCO'
mtex.use_map_color_diffuse = False
mtex.use_map_normal = True
#mtex.rgb_to_intensity = True

# Add texture slot
mtex = mat.texture_slots.add()
```

```
    mtex.texture = bTex
    mtex.texture_coords = 'UV'
    mtex.use_map_color_diffuse = True
    mtex.diffuse_color_factor = 1.0
    mtex.blend_type = 'MULTIPLY'

    # Create new cube and give it UVs
    bpy.ops.mesh.primitive_cube_add(location=origin)
    bpy.ops.object.mode_set(mode='EDIT')
    bpy.ops.uv.smart_project()
    bpy.ops.object.mode_set(mode='OBJECT')

    # Add material to current object
    ob = bpy.context.object
    me = ob.data
    me.materials.append(mat)

    return

if __name__ == "__main__":
    run((0,0,0))
```

4.3 Multiple materials

This program adds three materials to the same mesh.

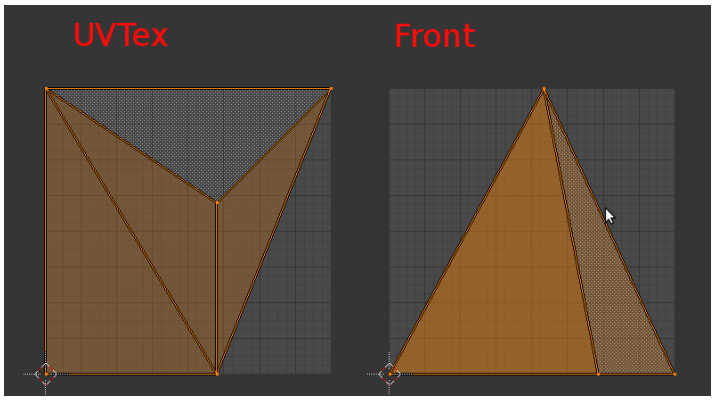


```
#----------------------------------------------------------
```

```
# File multi_material.py
#----------------------------------------------------------
import bpy

def run(origin):
    # Create three materials
    red = bpy.data.materials.new('Red')
    red.diffuse_color = (1,0,0)
    blue = bpy.data.materials.new('Blue')
    blue.diffuse_color = (0,0,1)
    yellow = bpy.data.materials.new('Yellow')
    yellow.diffuse_color = (1,1,0)

    # Create mesh and assign materials
    bpy.ops.mesh.primitive_uv_sphere_add(
        segments = 16,
        ring_count = 8,
        location=origin)
    ob = bpy.context.object
    ob.name = 'MultiMatSphere'
    me = ob.data
    me.materials.append(red)
    me.materials.append(blue)
    me.materials.append(yellow)

    # Assign materials to faces
    for f in me.faces:
        f.material_index = f.index % 3

    # Set left half of sphere smooth, right half flat shading
    for f in me.faces:
        f.use_smooth = (f.center[0] < 0)

if __name__ == "__main__":
    run((0,0,0))
```

4.4 UV layers

This program adds two UV layers to a mesh.




```
#----------------------------------------------------------
# File uvs.py
#----------------------------------------------------------
import bpy
import os

def createMesh(origin):
    # Create mesh and object
    me = bpy.data.meshes.new('TetraMesh')
    ob = bpy.data.objects.new('Tetra', me)
    ob.location = origin
    # Link object to scene
    scn = bpy.context.scene
    scn.objects.link(ob)
    scn.objects.active = ob
    scn.update()

    # List of verts and faces
    verts = [
        (1.41936, 1.41936, -1),
        (0.589378, -1.67818, -1),
        (-1.67818, 0.58938, -1),
        (0, 0, 1)
    ]
    faces = [(1,0,3), (3,2,1), (3,0,2), (0,1,2)]
    # Create mesh from given verts, edges, faces. Either edges or
    # faces should be [], or you ask for problems
    me.from_pydata(verts, [], faces)

    # Update mesh with new data
    me.update(calc_edges=True)
```

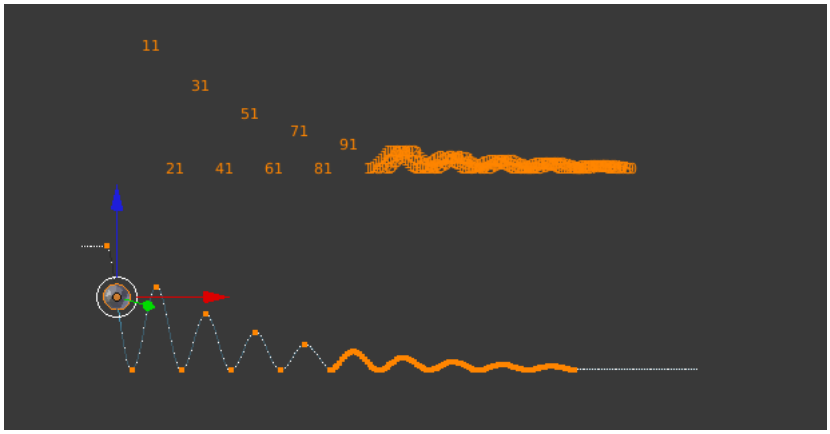
```
    # First texture layer: Main UV texture
    texFaces = [
        [(0.6,0.6), (1,1), (0,1)],
        [(0,1), (0.6,0), (0.6,0.6)],
        [(0,1), (0,0), (0.6,0)],
        [(1,1), (0.6,0.6), (0.6,0)]
    ]
    uvMain = createTextureLayer("UVMain", me, texFaces)

    # Second texture layer: Front projection
    texFaces = [
        [(0.732051,0), (1,0), (0.541778,1)],
        [(0.541778,1), (0,0), (0.732051,0)],
        [(0.541778,1), (1,0), (0,0)],
        [(1,0), (0.732051,0), (0,0)]
    ]
    uvFront = createTextureLayer("UVFront", me, texFaces)

    # Third texture layer: Smart projection
    bpy.ops.mesh.uv_texture_add()
    uvCyl = me.uv_textures.active
    uvCyl.name = 'UVCyl'
    bpy.ops.object.mode_set(mode='EDIT')
    bpy.ops.uv.cylinder_project()
    bpy.ops.object.mode_set(mode='OBJECT')

    # Want to make main layer active, but nothing seems to work - TBF
    me.uv_textures.active = uvMain
    me.uv_texture_clone = uvMain
    uvMain.active_render = True
    uvFront.active_render = False
    uvCyl.active_render = False

    return ob

def createTextureLayer(name, me, texFaces):
    uvtex = me.uv_textures.new()
    uvtex.name = name
    for n,tf in enumerate(texFaces):
        datum = uvtex.data[n]
        datum.uv1 = tf[0]
        datum.uv2 = tf[1]
        datum.uv3 = tf[2]
    return uvtex

def createMaterial():
```

```
    # Create image texture from image. Change here if the snippet
    # folder is not located in you home directory.
    realpath = os.path.expanduser('~/snippets/textures/color.png')
    tex = bpy.data.textures.new('ColorTex', type = 'IMAGE')
    tex.image = bpy.data.images.load(realpath)
    tex.use_alpha = True

    # Create shadeless material and MTex
    mat = bpy.data.materials.new('TexMat')
    mat.use_shadeless = True
    mtex = mat.texture_slots.add()
    mtex.texture = tex
    mtex.texture_coords = 'UV'
    mtex.use_map_color_diffuse = True
    return mat

def run(origin):
    ob = createMesh(origin)
    mat = createMaterial()
    ob.data.materials.append(mat)
    return

if __name__ == "__main__":
    run((0,0,0))
```

5 Actions and drivers

5.1 Object action

A bouncing ball.

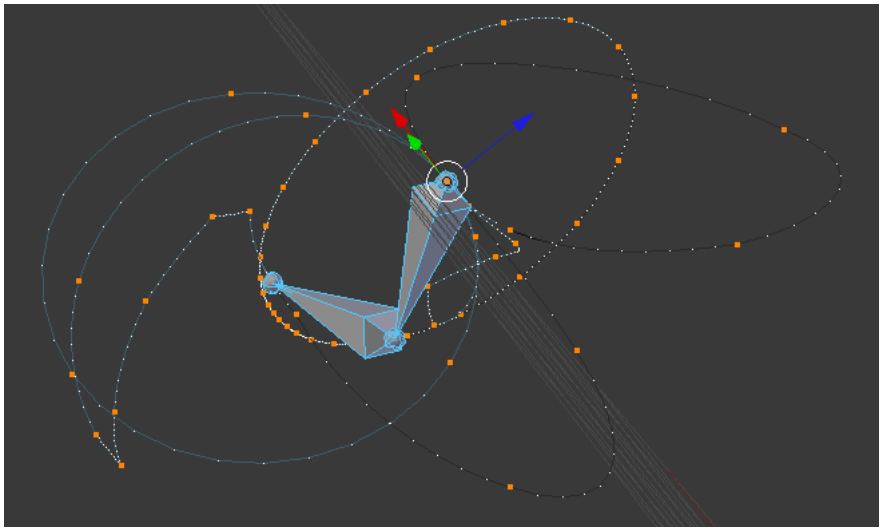


```
#----------------------------------------------
# File ob_action.py
#----------------------------------------------
import bpy
import math

def run(origin):
    # Set animation start and stop
    scn = bpy.context.scene
    scn.frame_start = 11
    scn.frame_end = 200

    # Create ico sphere
    bpy.ops.mesh.primitive_ico_sphere_add(location=origin)
    ob = bpy.context.object

    # Insert keyframes with operator code
    # Object should be automatically selected
    z = 10
    t = 1
    for n in range(5):
        t += 10
        bpy.ops.anim.change_frame(frame = t)
        bpy.ops.transform.translate(value=(2, 0, z))
        bpy.ops.anim.keyframe_insert_menu(type='Location')
        t += 10
        bpy.ops.anim.change_frame(frame = t)
        bpy.ops.transform.translate(value=(2, 0, -z))
        bpy.ops.anim.keyframe_insert_menu(type='Location')
        z *= 0.67

    action = ob.animation_data.action

    # Create dict with location FCurves
    fcus = {}
    for fcu in action.fcurves:
        if fcu.data_path == 'location':
            fcus[fcu.array_index] = fcu
    print(fcus.items())

    # Add new keypoints to x and z
    kpts_x = fcus[0].keyframe_points
    kpts_z = fcus[2].keyframe_points
    (x0,y0,z0) = origin
    omega = 2*math.pi/20
    z *= 0.67
```

```
    for t in range(101, 201):
        xt = 20 + 0.2*(t-101)
        zt = z*(1-math.cos(omega*(t - 101)))
        z *= 0.98
        kpts_z.insert(t, zt+z0, options={'FAST'})
    kpts_x.insert(t, xt+x0)

    # Change extrapolation and interpolation for
    # X curve to linear
    fcus[0].extrapolation = 'LINEAR'
    for kp in kpts_x:
        kp.interpolation = 'LINEAR'

    # Y location constant and can be removed
    action.fcurves.remove(fcus[1])
    bpy.ops.object.paths_calculate()
    return

if __name__ == "__main__":
    run((0,0,10))
    bpy.ops.screen.animation_play(reverse=False, sync=False)
```

5.2 Posebone action

This program creates an armature with two bones, which rotate in some complicated curves.

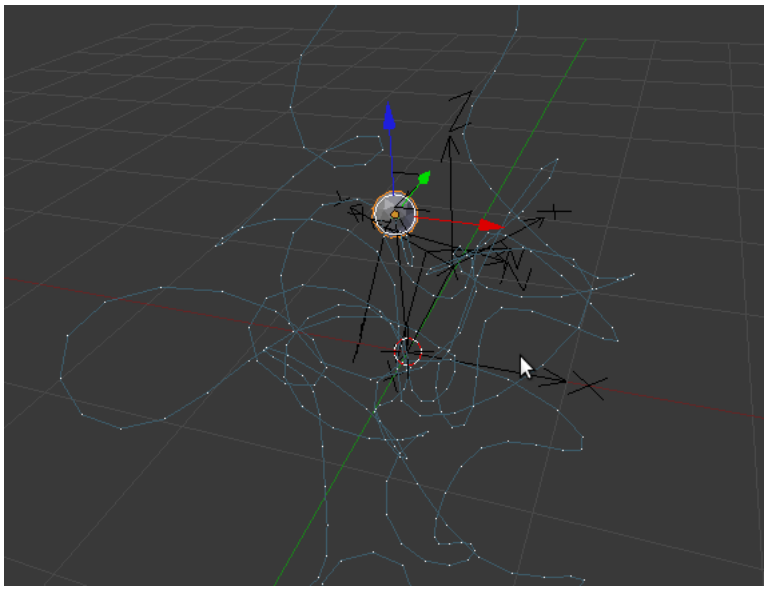


```
#----------------------------------------------
# File pose_action.py
#----------------------------------------------
import bpy
import math

def run(origin):
    # Set animation start and stop
    scn = bpy.context.scene
    scn.frame_start = 1
    scn.frame_end = 250

    # Create armature and object
    bpy.ops.object.armature_add()
    ob = bpy.context.object
    amt = ob.data

    # Rename first bone and create second bone
    bpy.ops.object.mode_set(mode='EDIT')
    base = amt.edit_bones['Bone']
    base.name = 'Base'
    tip = amt.edit_bones.new('Tip')
    tip.head = (0,0,1)
    tip.tail = (0,0,2)
    tip.parent = base
    tip.use_connect = True

    # Set object location in object mode
    bpy.ops.object.mode_set(mode='OBJECT')
    ob.location=origin

    # Set rotation mode to Euler ZYX
    bpy.ops.object.mode_set(mode='POSE')
    pbase = ob.pose.bones['Base']
    pbase.rotation_mode = 'ZYX'
    ptip = ob.pose.bones['Tip']
    ptip.rotation_mode = 'ZYX'

    # Insert 26 keyframes for two rotation FCurves
    # Last keyframe will be outside animation range

    for n in range(26):
        pbase.keyframe_insert(
            'rotation_euler',
            index=0,
            frame=n,
```

```
            group='Base')
        ptip.keyframe_insert(
            'rotation_euler',
            index=2,
            frame=n,
            group='Tip')

    # Get FCurves from newly created action
    action = ob.animation_data.action
    fcus = {}
    for fcu in action.fcurves:
        bone = fcu.data_path.split('"')[1]
        fcus[(bone, fcu.array_index)] = fcu

    # Modify the keypoints
    baseKptsRotX = fcus[('Base', 0)].keyframe_points
    tipKptsRotZ = fcus[('Tip', 2)].keyframe_points

    omega = 2*math.pi/250
    for n in range(26):
        t = 10*n
        phi = omega*t
        kp = baseKptsRotX[n]
        kp.co = (t+1,phi+0.7*math.sin(phi))
        kp.interpolation = 'LINEAR'
        kp = tipKptsRotZ[n]
        kp.co = (t+1, -3*phi+2.7*math.cos(2*phi))
        kp.interpolation = 'LINEAR'

    # Calculate paths for posebones
    bpy.ops.pose.select_all(action='SELECT')
    bpy.ops.pose.paths_calculate()
    return

if __name__ == "__main__":
    run((10,0,0))
    bpy.ops.screen.animation_play(reverse=False, sync=False)
```

5.3 Parenting

This program creates a complicated motion by consecutively parenting a few empties to each other, and assigning a simple rotation to each of them.

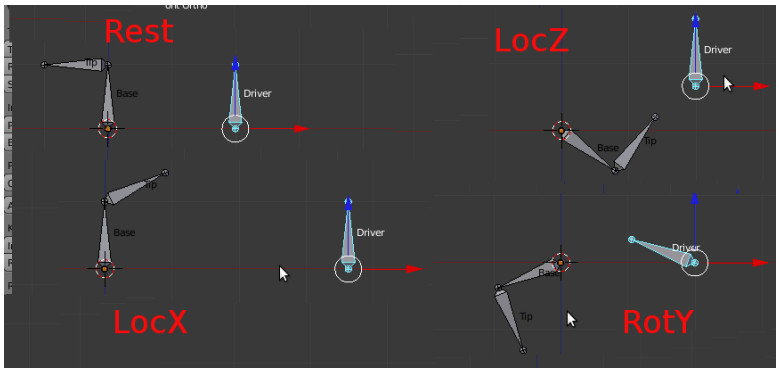


```
#-----------------------------------------------------------
# File epicycle.py
#-----------------------------------------------------------
import bpy
import math
from math import pi

def createEpiCycle(origin):
    periods = [1, 5, 8, 17]
    radii = [1.0, 0.3, 0.5, 0.1]
    axes = [0, 2, 1, 0]
    phases = [0, pi/4, pi/2, 0]

    # Add empties
    scn = bpy.context.scene
    empties = []
    nEmpties = len(periods)
    for n in range(nEmpties):
        empty = bpy.data.objects.new('Empty_%d' % n, None)
        scn.objects.link(empty)
        empties.append(empty)

    # Make each empty the parent of the consecutive one
    for n in range(1, nEmpties):
        empties[n].parent = empties[n-1]
        empties[n].location = (0, radii[n-1], 0)
```

```
    # Insert two keyframes for each empty
    for n in range(nEmpties):
        empty = empties[n]
        empty.keyframe_insert(
            'rotation_euler',
            index=axes[n],
            frame=0,
            group=empty.name)
        empty.keyframe_insert(
            'rotation_euler',
            index=axes[n],
            frame=periods[n],
            group=empty.name)
        fcu = empty.animation_data.action.fcurves[0]
        print(empty, fcu.data_path, fcu.array_index)

        kp0 = fcu.keyframe_points[0]
        kp0.co = (0, phases[n])
        kp0.interpolation = 'LINEAR'
        kp1 = fcu.keyframe_points[1]
        kp1.co = (250.0/periods[n], 2*pi + phases[n])
        kp1.interpolation = 'LINEAR'
        fcu.extrapolation = 'LINEAR'

    last = empties[nEmpties-1]
    bpy.ops.mesh.primitive_ico_sphere_add(
        size = 0.2,
        location=last.location)
    ob = bpy.context.object
    ob.parent = last

    empties[0].location = origin
    return

def run(origin):
    createEpiCycle(origin)
    bpy.ops.object.paths_calculate()
    return

if __name__ == "__main__":
    run((0,0,0))
    bpy.ops.screen.animation_play(reverse=False, sync=False)
```

5.4 Drivers

This program adds an armature with one driver bones and two driven bones. The tip's Z rotation is driven by the driver's x location. The base's Z rotation is driven both by the driver's Y location and its Z rotation.




```
#----------------------------------------------------------
# File driver.py
#----------------------------------------------------------
import bpy

def run(origin):
    # Create armature and object
    amt = bpy.data.armatures.new('MyRigData')
    rig = bpy.data.objects.new('MyRig', amt)
    rig.location = origin
    amt.show_names = True
    # Link object to scene
    scn = bpy.context.scene
    scn.objects.link(rig)
    scn.objects.active = rig
    scn.update()

    # Create bones
    bpy.ops.object.mode_set(mode='EDIT')
    base = amt.edit_bones.new('Base')
    base.head = (0,0,0)
    base.tail = (0,0,1)

    tip = amt.edit_bones.new('Tip')
    tip.head = (0,0,1)
    tip.tail = (0,0,2)
    tip.parent = base
```

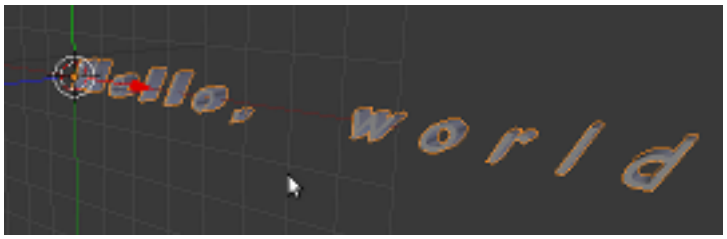
```
tip.use_connect = True

driver = amt.edit_bones.new('Driver')
driver.head = (2,0,0)
driver.tail = (2,0,1)

bpy.ops.object.mode_set(mode='POSE')

# Add driver for Tip's Z rotation
# Tip.rotz = 1.0 - 1.0*x, where x = Driver.locx
fcurve = rig.pose.bones["Tip"].driver_add('rotation_quaternion', 3)
drv = fcurve.driver
drv.type = 'AVERAGE'
drv.show_debug_info = True

var = drv.variables.new()
var.name = 'x'
var.type = 'TRANSFORMS'

targ = var.targets[0]
targ.id = rig
targ.transform_type = 'LOC_X'
targ.bone_target = 'Driver'
targ.use_local_space_transform = True

fmod = fcurve.modifiers[0]
fmod.mode = 'POLYNOMIAL'
fmod.poly_order = 1
fmod.coefficients = (1.0, -1.0)

# Add driver for Base's Z rotation
# Base.rotz = z*z - 3*y, where y = Driver.locy and z = Driver.rotz
fcurve = rig.pose.bones["Base"].driver_add('rotation_quaternion', 3)
drv = fcurve.driver
drv.type = 'SCRIPTED'
drv.expression = 'z*z - 3*y'
drv.show_debug_info = True

var1 = drv.variables.new()
var1.name = 'y'
var1.type = 'TRANSFORMS'

targ1 = var1.targets[0]
targ1.id = rig
targ1.transform_type = 'LOC_Y'
targ1.bone_target = 'Driver'
```

```
    targ1.use_local_space_transform = True

    var2 = drv.variables.new()
    var2.name = 'z'
    var2.type = 'TRANSFORMS'

    targ2 = var2.targets[0]
    targ2.id = rig
    targ2.transform_type = 'ROT_Z'
    targ2.bone_target = 'Driver'
    targ2.use_local_space_transform = True

    return

if __name__ == "__main__":
    run((0,0,0))
```

6 Other data types

6.1 Text

This program adds a piece of text to the viewport and sets some attributes. Note that the data type is TextCurve; the type Text is for text in the text editor.




```
#----------------------------------------------------------
# File text.py
#----------------------------------------------------------
import bpy
import math
from math import pi

def run(origin):
    # Create and name TextCurve object
```

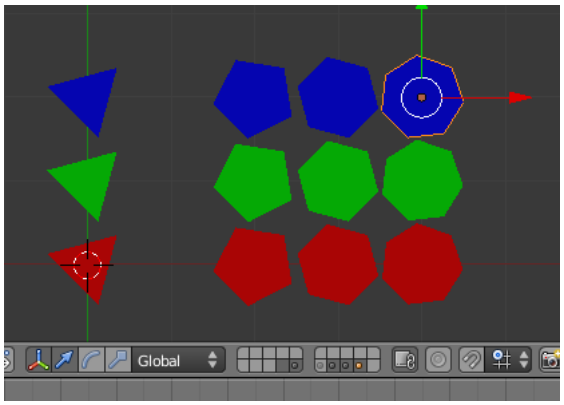
```
    bpy.ops.object.text_add(
    location=origin,
    rotation=(pi/2,0,pi))
    ob = bpy.context.object
    ob.name = 'HelloWorldText'
    tcu = ob.data
    tcu.name = 'HelloWorldData'

    # TextCurve attributes
    tcu.body = "Hello, world"
    tcu.font = bpy.data.fonts[0]
    tcu.offset_x = -9
    tcu.offset_y = -0.25
    tcu.shear = 0.5
    tcu.size = 3
    tcu.space_character = 2
    tcu.space_word = 4

    # Inherited Curve attributes
    tcu.extrude = 0.2
    tcu.use_fill_back = True
    tcu.use_fill_deform = True
    tcu.use_fill_front = True

if __name__ == "__main__":
    run((0,0,0))
```

6.2 Layers

This program illustrates three methods to place an object on a new level:

1. Create it on the right level.
2. Create it on layer 1, and change Object.layer
3. Create it on layer 1, and use an operator to move it.

It is also shown how to change the visible layers.



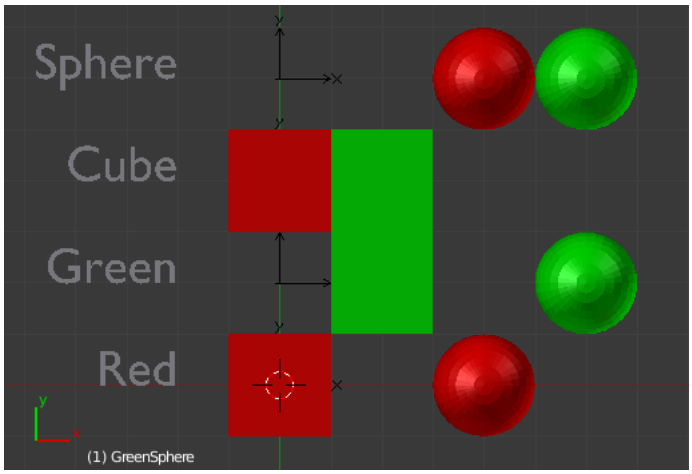

```
#------------------------------------------------------------
# File layers.py
#------------------------------------------------------------
import bpy

def createOnLayer(mat):
    for n in range(3, 8):
        # Create a n-gon on layer n+11
        layers = 20*[False]
        layers[n+11] = True

        bpy.ops.mesh.primitive_circle_add(
            vertices=n,
            radius=0.5,
            fill=True,
            view_align=True,
            layers=layers,
            location=(n-3,0,0)
        )
        bpy.context.object.data.materials.append(mat)
    return

def changeLayerData(mat):
    for n in range(3, 8):
        # Create a n-gon on layer 1
        bpy.ops.mesh.primitive_circle_add(
            vertices=n,
            radius=0.5,
            fill=True,
            view_align=True,
            location=(n-3,1,0)
        )
```

```
        bpy.context.object.data.materials.append(mat)

        # Then move it to a new layer
        ob = bpy.context.object
        ob.layers[n+11] = True

        # Remove it from other layers.
        layers = 20*[False]
        layers[n+11] = True
        for m in range(20):
            ob.layers[m] = layers[m]
    return

def moveLayerOperator(mat):
    for n in range(3, 8):
        # Create a n-gon on layer 1
        bpy.ops.mesh.primitive_circle_add(
            vertices=n,
            radius=0.5,
            fill=True,
            view_align=True,
            location=(n-3,2,0)
        )
        bpy.context.object.data.materials.append(mat)

        # Then move it to a new layer
        layers = 20*[False]
        layers[n+11] = True
        bpy.ops.object.move_to_layer(layers=layers)

    return


def run():
    # Create some materials
    red = bpy.data.materials.new('Red')
    red.diffuse_color = (1,0,0)
    green = bpy.data.materials.new('Green')
    green.diffuse_color = (0,1,0)
    blue = bpy.data.materials.new('Blue')
    blue.diffuse_color = (0,0,1)

    # Three methods to move objects to new layer
    createOnLayer(red)
    changeLayerData(green)
    moveLayerOperator(blue)
```

```
    # Select layers 14 - 20
    scn = bpy.context.scene
    bpy.ops.object.select_all(action='SELECT')
    for n in range(13,19):
        scn.layers[n] = True

    # Deselect layers 1 - 13, but only afterwards.
    # Seems like at least one layer must be selected at all times.
    for n in range(0,13):
        scn.layers[n] = False

    # Deselect layer 16
    scn.layers[15] = False
    return


if __name__ == "__main__":
    run()
```

6.3 Groups

This program shows how to create groups, add objects to groups, and empties that duplicates the groups. We add four groups, four mesh objects assigned to two groups each, and four texts assigned to a single group. Then we add four empties, which dupli-group the four groups. Finally the empties are moved so each row contains the elements in that group.



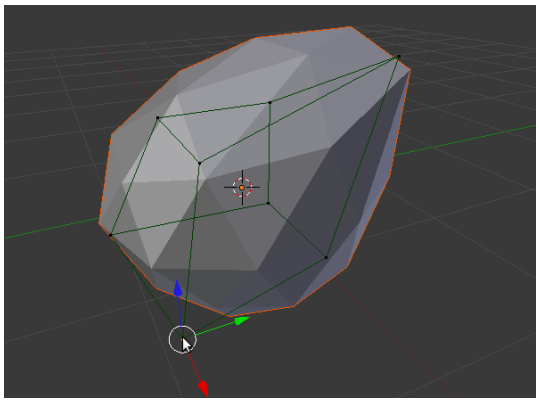
```
#----------------------------------------------------------
# File groups.py
# Create groups
#----------------------------------------------------------
import bpy
import mathutils
from mathutils import Vector

# Layers
Display = 5
Build = 6

def setObject(name, mat):
    ob = bpy.context.object
    ob.name = name
    ob.data.materials.append(mat)
    return ob

# Move object to given layer.
def moveToLayer(ob, layer):
    ob.layers[layer] = True
    for n in range(20):
        if n != layer:
            ob.layers[n] = False
    return

# Add a TextCurve object in layer 13
def addText(string, loc):
    tcu = bpy.data.curves.new(string+'Data', 'FONT')
    text = bpy.data.objects.new(string+'Text', tcu)
    tcu.body = string
    tcu.align = 'RIGHT'
    text.location = loc
    bpy.context.scene.objects.link(text)
    # Must change text.layers after text has been linked to scene,
    # otherwise the change may not stick.
    moveToLayer(text, Build)
    return text

def run():
    # Create two materials
    red = bpy.data.materials.new('RedMat')
    red.diffuse_color = (1,0,0)
    green = bpy.data.materials.new('GreenMat')
    green.diffuse_color = (0,1,0)
```

```
# Locations
origin = Vector((0,0,0))
dx = Vector((2,0,0))
dy = Vector((0,2,0))
dz = Vector((0,0,2))

# Put objects on the build layer
layers = 20*[False]
layers[Build] = True

# Create objects
bpy.ops.mesh.primitive_cube_add(location=dz, layers=layers)
redCube = setObject('RedCube', red)
bpy.ops.mesh.primitive_cube_add(location=dx+dz, layers=layers)
greenCube = setObject('GreenCube', green)
bpy.ops.mesh.primitive_uv_sphere_add(location=2*dx+dz, layers=layers)
redSphere = setObject('RedSphere', red)
bpy.ops.mesh.primitive_uv_sphere_add(location=3*dx+dz, layers=layers)
greenSphere = setObject('GreenSphere', green)

# Create texts
redText = addText('Red', -dx)
greenText = addText('Green', -dx)
cubeText = addText('Cube', -dx)
sphereText = addText('Sphere', -dx)

# Create groups
redGrp = bpy.data.groups.new('RedGroup')
greenGrp = bpy.data.groups.new('GreenGroup')
cubeGrp = bpy.data.groups.new('CubeGroup')
sphereGrp = bpy.data.groups.new('SphereGroup')

# Table of group members
members = {
    redGrp : [redCube, redSphere, redText],
    greenGrp : [greenCube, greenSphere, greenText],
    cubeGrp : [redCube, greenCube, cubeText],
    sphereGrp : [redSphere, greenSphere, sphereText]
}

# Link objects to groups
for group in members.keys():
    for ob in members[group]:
        group.objects.link(ob)

# List of empties
```

```
    empties = [
        ('RedEmpty', origin, redGrp),
        ('GreenEmpty', dy, greenGrp),
        ('CubeEmpty', 2*dy, cubeGrp),
        ('SphereEmpty', 3*dy, sphereGrp)
    ]

    # Create Empties and put them on the display layer
    scn = bpy.context.scene
    for (name, loc, group) in empties:
        empty = bpy.data.objects.new(name, None)
        empty.location = loc
        empty.name = name
        empty.dupli_type = 'GROUP'
        empty.dupli_group = group
        scn.objects.link(empty)
        moveToLayer(empty, Display)


    # Make display layer into the active layer
    scn.layers[Display] = True
    for n in range(20):
        if n != Display:
            scn.layers[n] = False

    return

if __name__ == "__main__":
    run()
```

6.4 Lattice

This program adds an icosphere deformed by a lattice. The lattice modifier only acts on the vertex group on the upper half of the sphere.

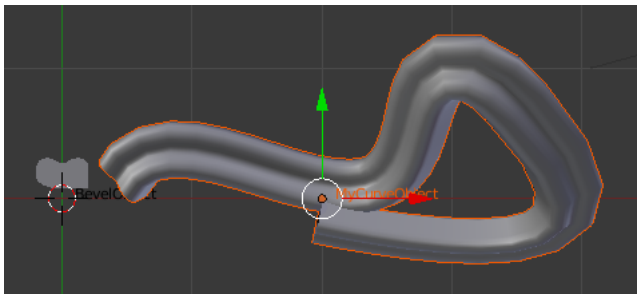


```
#------------------------------------------------------------
# File lattice.py
#------------------------------------------------------------
import bpy

def createIcoSphere(origin):
    # Create an icosphere
    bpy.ops.mesh.primitive_ico_sphere_add(location=origin)
    ob = bpy.context.object
    me = ob.data

    # Create vertex groups
    upper = ob.vertex_groups.new('Upper')
    lower = ob.vertex_groups.new('Lower')
    for v in me.vertices:
        if v.co[2] > 0.001:
            upper.add([v.index], 1.0, 'REPLACE')
        elif v.co[2] < -0.001:
            lower.add([v.index], 1.0, 'REPLACE')
        else:
            upper.add([v.index], 0.5, 'REPLACE')
            lower.add([v.index], 0.5, 'REPLACE')
    return ob

def createLattice(origin):
    # Create lattice and object
    lat = bpy.data.lattices.new('MyLattice')
    ob = bpy.data.objects.new('LatticeObject', lat)
    ob.location = origin
    ob.show_x_ray = True
    # Link object to scene
    scn = bpy.context.scene
```

```
    scn.objects.link(ob)
    scn.objects.active = ob
    scn.update()

    # Set lattice attributes
    lat.interpolation_type_u = 'KEY_LINEAR'
    lat.interpolation_type_v = 'KEY_CARDINAL'
    lat.interpolation_type_w = 'KEY_BSPLINE'
    lat.use_outside = False
    lat.points_u = 2
    lat.points_v = 2
    lat.points_w = 2

    # Set lattice points
    s = 1.0
    points = [
        (-s,-s,-s), (s,-s,-s), (-s,s,-s), (s,s,-s),
        (-s,-s,s), (s,-s,s), (-s,s,s), (s,s,s)
    ]
    for n,pt in enumerate(lat.points):
        for k in range(3):
            #pt.co[k] = points[n][k]
            pass
    return ob

def run(origin):
    sphere = createIcoSphere(origin)
    lat = createLattice(origin)
    # Create lattice modifier
    mod = sphere.modifiers.new('Lat', 'LATTICE')
    mod.object = lat
    mod.vertex_group = 'Upper'
    # Lattice in edit mode for easy deform
    bpy.context.scene.update()
    bpy.ops.object.mode_set(mode='EDIT')
    return

if __name__ == "__main__":
    run((0,0,0))
```

6.5 Curve

This program adds a Bezier curve. It also adds a Nurbs circle which is used as a bevel object.



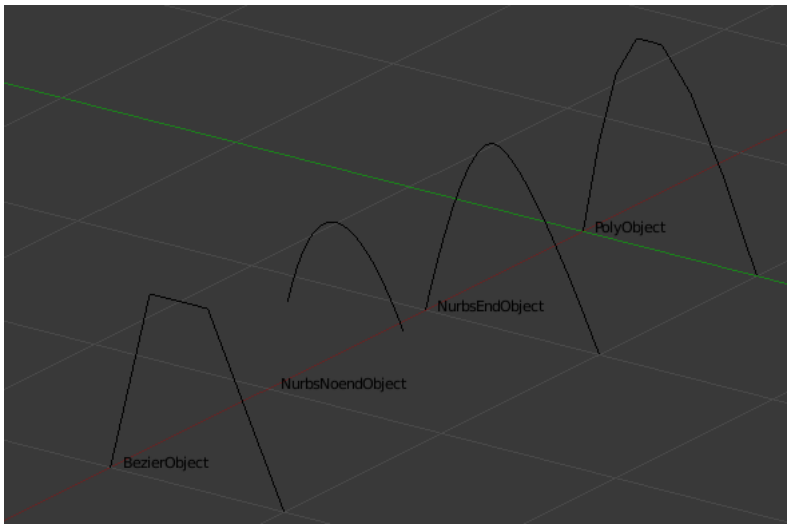
```
#-------------------------------------------------------------
# File curve.py
#-------------------------------------------------------------
import bpy

def createBevelObject():
    # Create Bevel curve and object
    cu = bpy.data.curves.new('BevelCurve', 'CURVE')
    ob = bpy.data.objects.new('BevelObject', cu)
    bpy.context.scene.objects.link(ob)

    # Set some attributes
    cu.dimensions = '2D'
    cu.resolution_u = 6
    cu.twist_mode = 'MINIMUM'
    ob.show_name = True

    # Control point coordinates
    coords = [
        (0.00,0.08,0.00,1.00),
        (-0.20,0.08,0.00,0.35),
        (-0.20,0.19,0.00,1.00),
        (-0.20,0.39,0.00,0.35),
        (0.00,0.26,0.00,1.00),
        (0.20,0.39,0.00,0.35),
        (0.20,0.19,0.00,1.00),
        (0.20,0.08,0.00,0.35)
    ]

    # Create spline and set control points
    spline = cu.splines.new('NURBS')
    nPointsU = len(coords)
    spline.points.add(nPointsU)
    for n in range(nPointsU):
        spline.points[n].co = coords[n]
```

```
    # Set spline attributes. Points probably need to exist here.
    spline.use_cyclic_u = True
    spline.resolution_u = 6
    spline.order_u = 3

    return ob


def createCurveObject(bevob):
    # Create curve and object
    cu = bpy.data.curves.new('MyCurve', 'CURVE')
    ob = bpy.data.objects.new('MyCurveObject', cu)
    bpy.context.scene.objects.link(ob)

    # Set some attributes
    cu.bevel_object = bevob
    cu.dimensions = '3D'
    cu.use_fill_back = True
    cu.use_fill_front = True
    ob.show_name = True

    # Bezier coordinates
    beziers = [
        ((-1.44,0.20,0.00), (-1.86,-0.51,-0.36), (-1.10,0.75,0.28)),
        ((0.42,0.13,-0.03), (-0.21,-0.04,-0.27), (1.05,0.29,0.21)),
        ((1.20,0.75,0.78), (0.52,1.36,1.19), (2.76,-0.63,-0.14))
    ]

    # Create spline and set Bezier control points
    spline = cu.splines.new('BEZIER')
    nPointsU = len(beziers)
    spline.bezier_points.add(nPointsU)
    for n in range(nPointsU):
        bpt = spline.bezier_points[n]
        (bpt.co, bpt.handle_left, bpt.handle_right) = beziers[n]
    return ob

def run(origin):
    bevob = createBevelObject()
    bevob.location = origin

    curveob = createCurveObject(bevob)
    curveob.location = origin
    bevob.select = False
    curveob.select = True
```

```
    bpy.ops.transform.translate(value=(2,0,0))
    return

if __name__ == "__main__":
    run((0,0,0))
```

6.6 Curve types

This program illustrates the difference between the curve types: POLY, NURBS and BEZIER.



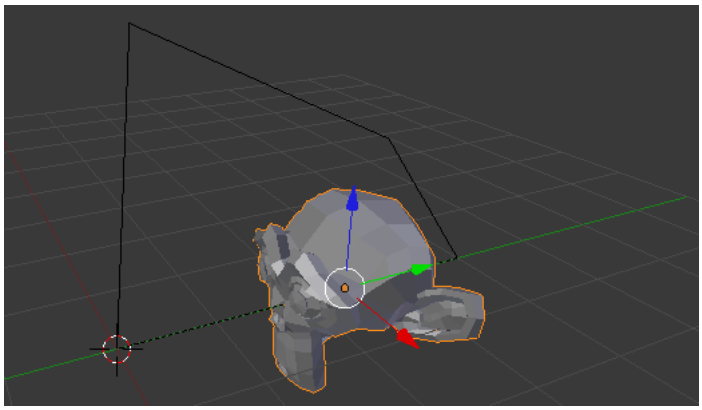

```
#----------------------------------------------------------
# File curve_types.py
#----------------------------------------------------------
import bpy
from math import sin, pi

# Poly and nurbs
def makePolySpline(cu):
    spline = cu.splines.new('POLY')
    cu.dimensions = '3D'
    addPoints(spline, 8)

def makeNurbsSpline(cu):
    spline = cu.splines.new('NURBS')
```

```
    cu.dimensions = '3D'
    addPoints(spline, 4)
    spline.order_u = 3
    return spline

def addPoints(spline, nPoints):
    spline.points.add(nPoints-1)
    delta = 1/(nPoints-1)
    for n in range(nPoints):
        spline.points[n].co = (0, n*delta, sin(n*pi*delta), 1)

# Bezier
def makeBezierSpline(cu):
    spline = cu.splines.new('BEZIER')
    cu.dimensions = '3D'
    order = 3
    addBezierPoints(spline, order+1)
    spline.order_u = order

def addBezierPoints(spline, nPoints):
    spline.bezier_points.add(nPoints-1)
    bzs = spline.bezier_points
    delta = 1/(nPoints-1)
    for n in range(nPoints):
        bzs[n].co = (0, n*delta, sin(n*pi*delta))
        print(bzs[n].co)
    for n in range(1, nPoints):
        bzs[n].handle_left = bzs[n-1].co
    for n in range(nPoints-1):
        bzs[n].handle_right = bzs[n+1].co
    return spline

# Create curve and object and link to scene
def makeCurve(name, origin, dx):
    cu = bpy.data.curves.new('%sCurve' % name, 'CURVE')
    ob = bpy.data.objects.new('%sObject' % name, cu)
    (x,y,z) = origin
    ob.location = (x+dx,y,z)
    ob.show_name = True
    bpy.context.scene.objects.link(ob)
    return cu

def run(origin):
    polyCurve = makeCurve("Poly", origin, 0)
    makePolySpline(polyCurve)
    nurbsCurve = makeCurve("NurbsEnd", origin, 1)
```

```
    spline = makeNurbsSpline(nurbsCurve)
    spline.use_endpoint_u = True
    nurbsCurve = makeCurve("NurbsNoend", origin, 2)
    spline = makeNurbsSpline(nurbsCurve)
    spline.use_endpoint_u = False
    bezierCurve = makeCurve("Bezier", origin, 3)
    makeBezierSpline(bezierCurve)
    return

if __name__ == "__main__":
    run((0,0,0))
```

6.7 Path

This program adds a path and a monkey with a follow path constraint.



```
#----------------------------------------------------------
# File path.py
#----------------------------------------------------------
import bpy

def run(origin):
    # Create path data and object
    path = bpy.data.curves.new('MyPath', 'CURVE')
    pathOb = bpy.data.objects.new('Path', path)
    pathOb.location = origin
    bpy.context.scene.objects.link(pathOb)

    # Set path data
    path.dimensions = '3D'
```

```
    path.use_path = True
    path.use_path_follow = True
    path.path_duration = 100

    # Animate path
    path.eval_time = 0
    path.keyframe_insert(data_path="eval_time", frame=0)
    path.eval_time = 100
    path.keyframe_insert(data_path="eval_time", frame=250)

    # Add a spline to path
    spline = path.splines.new('POLY')
    spline.use_cyclic_u = True
    spline.use_endpoint_u = False

    # Add points to spline
    pointTable = [(0,0,0,0), (1,0,3,0),
        (1,2,2,0), (0,4,0,0), (0,0,0,0)]
    nPoints = len(pointTable)
    spline.points.add(nPoints-1)
    for n in range(nPoints):
        spline.points[n].co = pointTable[n]

    # Add a monkey
    bpy.ops.mesh.primitive_monkey_add()
    monkey = bpy.context.object

    # Add follow path constraint to monkey
    cns = monkey.constraints.new('FOLLOW_PATH')
    cns.target = pathOb
    cns.use_curve_follow = True
    cns.use_curve_radius = True
    cns.use_fixed_location = False
    cns.forward_axis = 'FORWARD_Z'
    cns.up_axis = 'UP_Y'

    return

if __name__ == "__main__":
    run((0,0,0))
    bpy.ops.screen.animation_play(reverse=False, sync=False)
```

6.8 Camera and lights

This program adds a sun light to the scene, and a spot light for every render object in the scene. Each spot has a TrackTo constraint making to point to its object, whereas the sun tracks the center of all objects in the scene.

Then a new camera is added and made into the active camera. Finally the program switches to the Default screen, and changes the viewport to the new camera.

```
#-------------------------------------------------------------
# File camera.py
# Adds one camera and several lights
#-------------------------------------------------------------
import bpy, mathutils, math
from mathutils import Vector
from math import pi

def findMidPoint():
    sum = Vector((0,0,0))
    n = 0
    for ob in bpy.data.objects:
        if ob.type not in ['CAMERA', 'LAMP', 'EMPTY']:
            sum += ob.location
            n += 1
    if n == 0:
        return sum
    else:
        return sum/n

def addTrackToConstraint(ob, name, target):
    cns = ob.constraints.new('TRACK_TO')
    cns.name = name
    cns.target = target
    cns.track_axis = 'TRACK_NEGATIVE_Z'
    cns.up_axis = 'UP_Y'
    cns.owner_space = 'WORLD'
    cns.target_space = 'WORLD'
    return

def createLamp(name, lamptype, loc):
    bpy.ops.object.add(
        type='LAMP',
        location=loc)
    ob = bpy.context.object
```

```
    ob.name = name
    lamp = ob.data
    lamp.name = 'Lamp'+name
    lamp.type = lamptype
    return ob

def createLamps(origin, target):
    deg2rad = 2*pi/360

    sun = createLamp('sun', 'SUN', origin+Vector((0,20,50)))
    lamp = sun.data
    lamp.type = 'SUN'
    addTrackToConstraint(sun, 'TrackMiddle', target)

    for ob in bpy.context.scene.objects:
        if ob.type == 'MESH':
            spot = createLamp(ob.name+'Spot', 'SPOT', ob.location+Vector((0,2,1)))
            bpy.ops.transform.resize(value=(0.5,0.5,0.5))
            lamp = spot.data

            # Lamp
            lamp.type = 'SPOT'
            lamp.color = (0.5,0.5,0)
            lamp.energy = 0.9
            lamp.falloff_type = 'INVERSE_LINEAR'
            lamp.distance = 7.5

            # Spot shape
            lamp.spot_size = 30*deg2rad
            lamp.spot_blend = 0.3

            # Shadows
            lamp.shadow_method = 'BUFFER_SHADOW'
            lamp.use_shadow_layer = True
            lamp.shadow_buffer_type = 'REGULAR'
            lamp.shadow_color = (0,0,1)

            addTrackToConstraint(spot, 'Track'+ob.name, ob)
    return

def createCamera(origin, target):
    # Create object and camera
    bpy.ops.object.add(
        type='CAMERA',
        location=origin,
        rotation=(pi/2,0,pi))
```

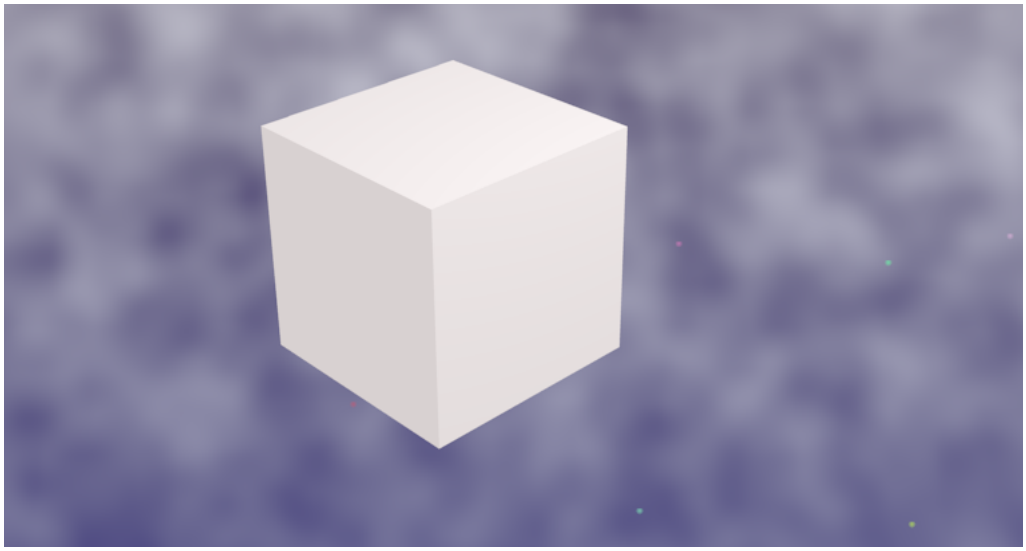
```
    ob = bpy.context.object
    ob.name = 'MyCamOb'
    cam = ob.data
    cam.name = 'MyCam'
    addTrackToConstraint(ob, 'TrackMiddle', target)


    # Lens
    cam.type = 'PERSP'
    cam.lens = 75
    cam.lens_unit = 'MILLIMETERS'
    cam.shift_x = -0.05
    cam.shift_y = 0.1
    cam.clip_start = 10.0
    cam.clip_end = 250.0

    empty = bpy.data.objects.new('DofEmpty', None)
    empty.location = origin+Vector((0,10,0))
    cam.dof_object = empty

    # Display
    cam.show_title_safe = True
    cam.show_name = True

    # Make this the current camera
    scn = bpy.context.scene
    scn.camera = ob
    return ob

def run(origin):
    # Delete all old cameras and lamps
    scn = bpy.context.scene
    for ob in scn.objects:
        if ob.type == 'CAMERA' or ob.type == 'LAMP':
            scn.objects.unlink(ob)

    # Add an empty at the middle of all render objects
    midpoint = findMidPoint()
    bpy.ops.object.add(
        type='EMPTY',
        location=midpoint),
    target = bpy.context.object
    target.name = 'Target'

    createCamera(origin+Vector((50,90,50)), target)
    createLamps(origin, target)
```

```
    return

if __name__ == "__main__":
    run(Vector((0,0,0)))
```

7 World, view and render

7.1 World

This program modifies the world settings. The picture is a rendering of the default cube with the default camera and lighting.



```
#----------------------------------------------
# File world.py
#----------------------------------------------
import bpy

def run():
    world = bpy.context.scene.world

    # World settings
    world.use_sky_blend = True
    world.ambient_color = (0.05, 0, 0)
    world.horizon_color = (0, 0, 0.2)
    world.zenith_color = (0.04, 0, 0.04)
```

```
    # Stars
    sset = world.star_settings
    sset.use_stars = True
    sset.average_separation = 17.8
    sset.color_random = 1.0
    sset.distance_min = 0.7
    sset.size = 10

    # Environment lighting
    wset = world.light_settings
    wset.use_environment_light = True
    wset.use_ambient_occlusion = True
    wset.ao_blend_type = 'MULTIPLY'
    wset.ao_factor = 0.8
    wset.gather_method = 'APPROXIMATE'

    # Clouds texture
    tex = bpy.data.textures.new('Clouds', type = 'CLOUDS')
    tex.cloud_type = 'GREYSCALE'
    tex.noise_type = 'SOFT_NOISE'
    tex.noise_basis = 'ORIGINAL_PERLIN'
    tex.noise_scale = 0.06
    tex.noise_depth = 1

    # Set texture as active world texture
    world.active_texture = tex

    # Retrieve texture slot
    wtex = world.texture_slots[world.active_texture_index]
    print(wtex, world.active_texture_index)

    # Texture slot settings
    wtex.use_map_blend = False
    wtex.use_map_horizon = False
    wtex.use_map_zenith_down = False
    wtex.use_map_zenith_up = True
    wtex.color = (1,1,1)
    wtex.texture_coords = 'VIEW'
    wtex.zenith_up_factor = 1.0
    return

if __name__ == "__main__":
    run()
```

7.2 View and render

This program modifies the render settings, switches to the Default screen, and changes to Camera viewport. Finally the animation is started, unfortunately in the old view.

```
#----------------------------------------------------------
# File view.py
# Changes the view and render settings
#----------------------------------------------------------
import bpy

def setRenderSettings():
    render = bpy.context.scene.render
    render.resolution_x = 720
    render.resolution_y = 576
    render.resolution_percentage = 100
    render.fps = 24
    render.use_raytrace = False
    render.use_color_management = True
    render.use_sss = False
    return

def setDefaultCameraView():
    for scrn in bpy.data.screens:
        if scrn.name == 'Default':
            bpy.context.window.screen = scrn
            for area in scrn.areas:
                if area.type == 'VIEW_3D':
                    for space in area.spaces:
                        if space.type == 'VIEW_3D':
                            space.viewport_shade = 'SOLID'
                            reg = space.region_3d
                            reg.view_perspective = 'CAMERA'
            break
    return

def run():
    setRenderSettings()
    setDefaultCameraView()
    # Start animation, unfortunately in the old view
    bpy.ops.screen.animation_play(reverse=False, sync=False)
    return

if __name__ == "__main__":
    run()
```

8 Properties

8.1 RNA properties versus ID properties

In Blender there are two different types of properties: ID properties and RNA properties An RNA property extends the definition given data structure. It must be declared before it is used.

```
bpy.types.Object.myRnaInt = bpy.props.IntProperty(
    name = "RNA int",
    min = -100, max = 100,
    default = 33)
```

Once declared, RNA properties are accessed with the dot syntax:

```
    cube.myRnaInt = -99
```

Since the declaration of the RNA property `myRnaInt` extends the definition of the Object data structure, every object will have this property.

An ID property is added to a single datablock, without affecting other data of the same type. It does not need any prior declaration, but is automatically defined when it is set, e.g.

```
    cube.data["MyIdInt"] = 4711
```

ID properties can only be integers, floats, and strings; other types will be automatically converted. Hence the line

```
    cube.data["MyIdBool"] = True
```

defines an integer ID property and not a boolean.

Properties are stored in the blend file, but property declarations are not.

Here is a script which creates three meshes, assigns various properties and prints their values to the console.

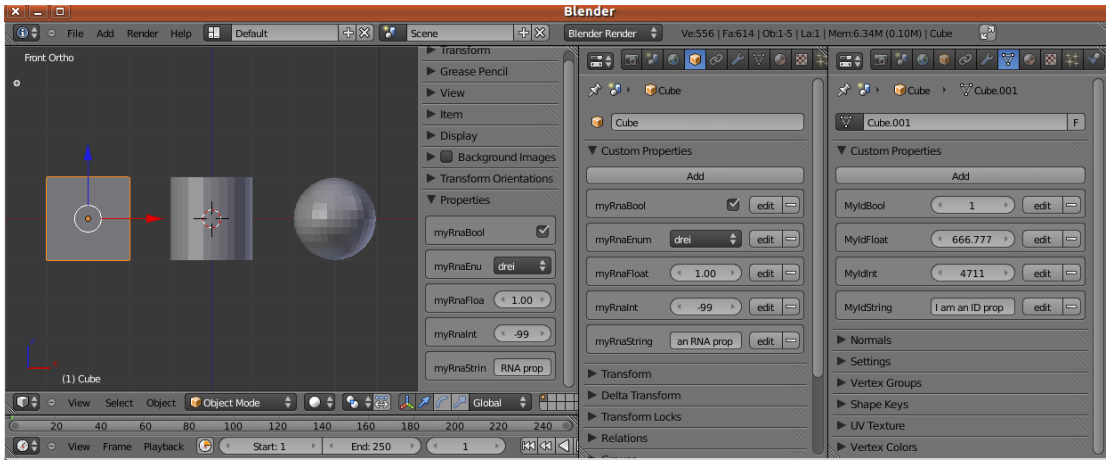
```
#----------------------------------------------------------
# File properties.py
#----------------------------------------------------------
import bpy
from bpy.props import *

# Clean the scene and create some objects
bpy.ops.object.select_by_type(type='MESH')
bpy.ops.object.delete()
bpy.ops.mesh.primitive_cube_add(location=(-3,0,0))
cube = bpy.context.object
bpy.ops.mesh.primitive_cylinder_add(location=(0,0,0))
cyl = bpy.context.object
bpy.ops.mesh.primitive_uv_sphere_add(location=(3,0,0))
sphere = bpy.context.object

# Define RNA props for every object
bpy.types.Object.myRnaInt = IntProperty(
    name = "RNA int",
    min = -100, max = 100,
    default = 33)

bpy.types.Object.myRnaFloat = FloatProperty(
    name = "RNA float",
    default = 12.345,
    min = 1, max = 20)

bpy.types.Object.myRnaString = StringProperty(
    name = "RNA string",
    default = "Ribonucleic acid")

bpy.types.Object.myRnaBool = BoolProperty(
    name = "RNA bool")

bpy.types.Object.myRnaEnum = EnumProperty(
    items = [('one', 'eins', 'un'),
             ('two', 'zwei', 'deux'),
             ('three', 'drei', 'trois')],
    name = "RNA enum")

# Set the cube's RNA props
cube.myRnaInt = -99
cube.myRnaFloat = -1
cube.myRnaString = "I am an RNA prop"
cube.myRnaBool = True
cube.myRnaEnum = 'three'
```

```
# Create ID props fore cube mesh by setting them.
cube.data["MyIdInt"] = 4711
cube.data["MyIdFloat"] = 666.777
cube.data["MyIdString"] = "I am an ID prop"
cube.data["MyIdBool"] = True

# Print all properties
def printProp(rna, path):
    try:
        print('    %s%s =' % (rna.name, path), eval("rna"+path))
    except:
        print('    %s%s does not exist' % (rna.name, path))

for ob in [cube, cyl, sphere]:
    print("%s RNA properties" % ob)
    printProp(ob, ".myRnaInt")
    printProp(ob, ".myRnaFloat")
    printProp(ob, ".myRnaString")
    printProp(ob, ".myRnaBool")
    printProp(ob, ".myRnaEnum")
    print("%s ID properties" % ob.data)
    printProp(ob.data, '["MyIdInt"]')
    printProp(ob.data, '["MyIdFloat"]')
    printProp(ob.data, '["MyIdString"]')
    printProp(ob.data, '["MyIdBool"]')
```

The script prints the following output to the console.

```
<bpy_struct, Object("Cube")> RNA properties
    Cube.myRnaInt = -99
    Cube.myRnaFloat = 1.0
    Cube.myRnaString = I am an RNA prop
    Cube.myRnaBool = True
    Cube.myRnaEnum = three
<bpy_struct, Mesh("Cube.001")> ID properties
    Cube.001["MyIdInt"] = 4711
    Cube.001["MyIdFloat"] = 666.777
    Cube.001["MyIdString"] = I am an ID prop
    Cube.001["MyIdBool"] = 1
<bpy_struct, Object("Cylinder")> RNA properties
    Cylinder.myRnaInt = 33
    Cylinder.myRnaFloat = 12.345000267028809
    Cylinder.myRnaString = Ribonucleic acid
    Cylinder.myRnaBool = False
```

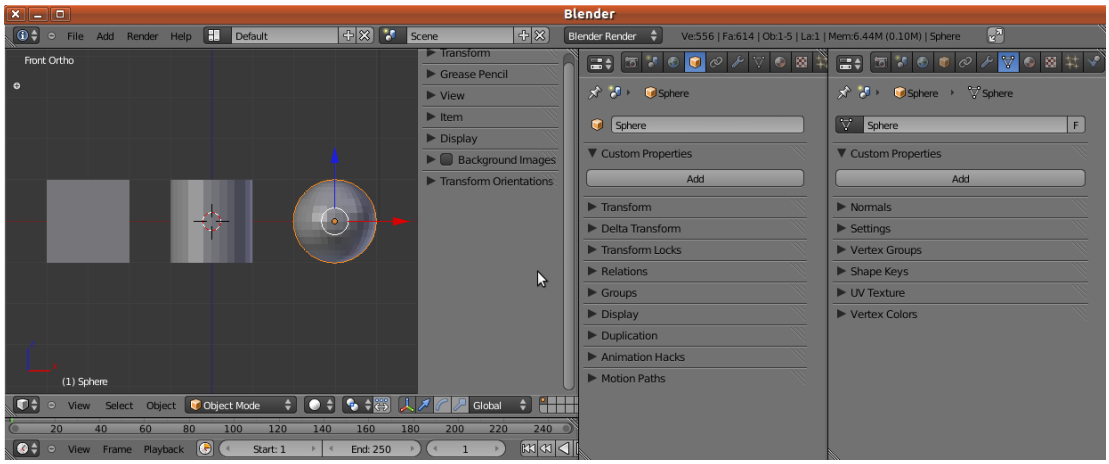
```
    Cylinder.myRnaEnum = one
<bpy_struct, Mesh("Cylinder")> ID properties
    Cylinder["MyIdInt"] does not exist
    Cylinder["MyIdFloat"] does not exist
    Cylinder["MyIdString"] does not exist
    Cylinder["MyIdBool"] does not exist
<bpy_struct, Object("Sphere")> RNA properties
    Sphere.myRnaInt = 33
    Sphere.myRnaFloat = 12.345000267028809
    Sphere.myRnaString = Ribonucleic acid
    Sphere.myRnaBool = False
    Sphere.myRnaEnum = one
<bpy_struct, Mesh("Sphere")> ID properties
    Sphere["MyIdInt"] does not exist
    Sphere["MyIdFloat"] does not exist
    Sphere["MyIdString"] does not exist
    Sphere["MyIdBool"] does not exist
```



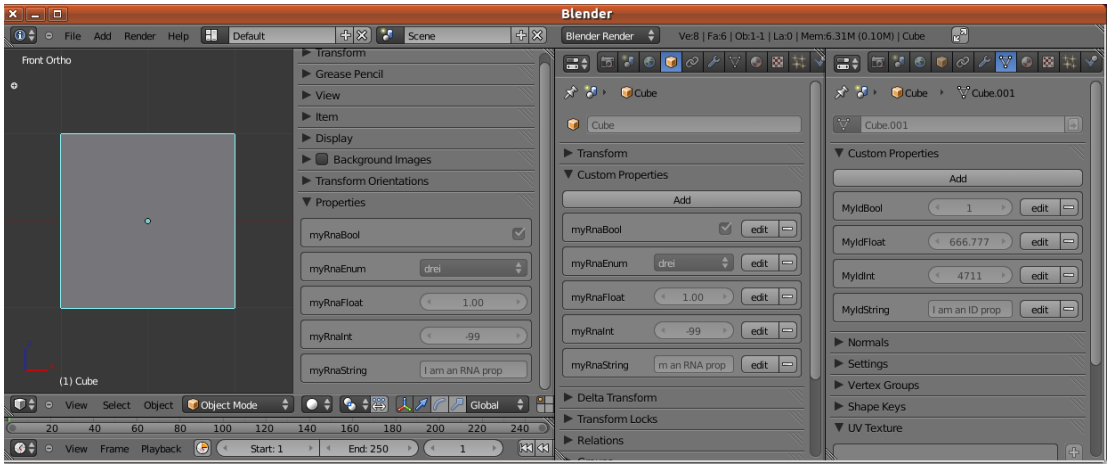
All three objects have RNA properties, because they are an extension of the Object datatype. The cube's RNA properties have the values assigned by the program, except the `myRnaFloat` value which is not allowed to be smaller than 1. No properties were set for the cylinder and the sphere, but they still have RNA properties with default values.

The cube mesh has ID properties set by the program. Note that the MyIdBool property is the integer 1 rather than the boolean True.

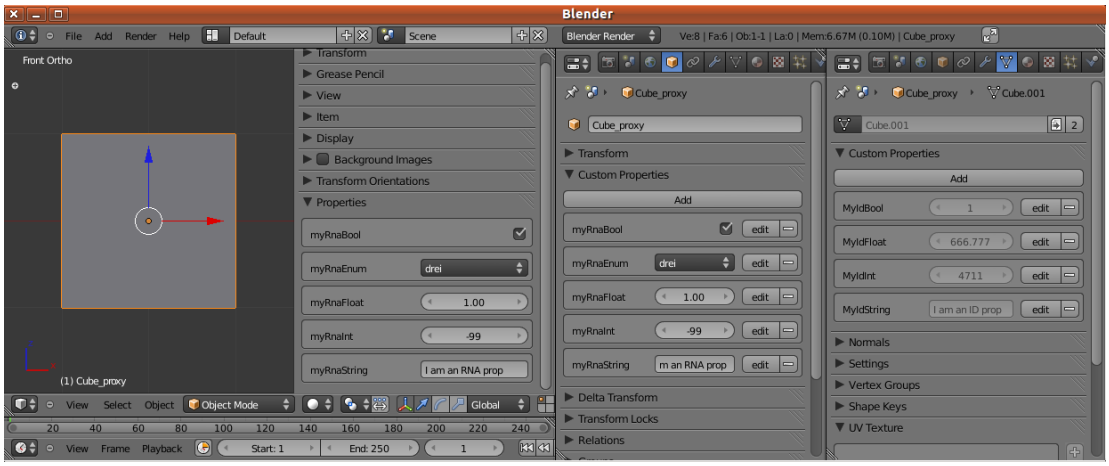
The object properties are displayed in the in the UI panel under Properties, and also in the object context. The mesh properties be found in the mesh context.



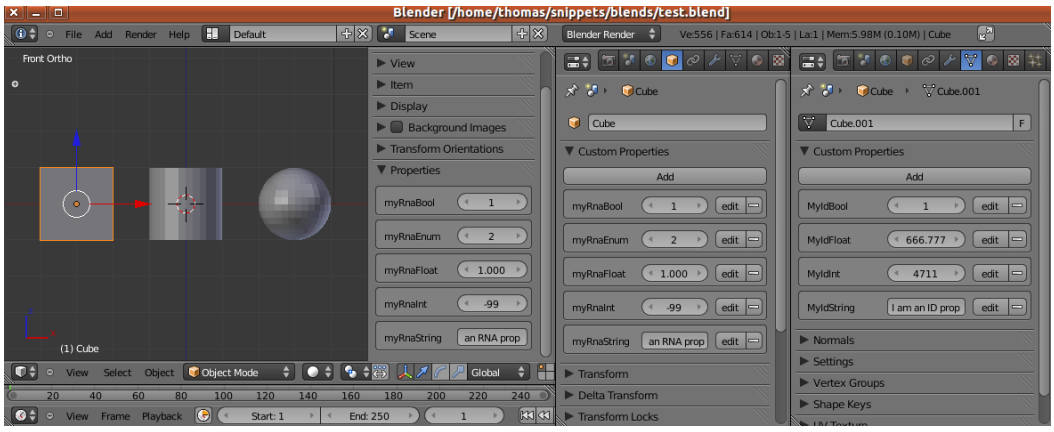
As we saw in the printout, we can access the sphere object's RNA properties. However, they do not show up in user interface. Apparently only set properties are stored in the Object data block. We can use an RNA property that has not been set in a script; it then takes the default value. In contrast, a error is raised if we try to access a non-set ID property.



Properties are compatible with file linking. Save the blend file and link the cube into a new file. Both the RNA and ID properties appear in the new file, but they are greyed out because they can not be accessed in the linking file.



If we proxify the linked cube, the object properties belong to the proxy object datablock, and can be modified in the linking file. In contrast, the mesh properties belong to the mesh datablock and can not be changed.



As mentioned above, properties are stored in the blend files but property declarations are not. Quit and restart Blender and open the file which we save above. The `myRnaBool` and `myRnaEnum` properties have been converted to integers. They were in fact stored as integers all the time, but were displayed as booleans and enums due to the property declarations stored in the Object data type.

To confirm that the RNA properties have turned into ID properties, execute the following script.

```
#-----------------------------------------------------------
# File print_props.py
```

```
#----------------------------------------------------------
import bpy

def printProp(rna, path):
    try:
        print('    %s%s =' % (rna.name, path), eval("rna"+path))
    except:
        print('    %s%s does not exist' % (rna.name, path))

ob = bpy.context.object
print("%s RNA properties" % ob)
printProp(ob, ".myRnaInt")
printProp(ob, ".myRnaFloat")
printProp(ob, ".myRnaString")
printProp(ob, ".myRnaBool")
printProp(ob, ".myRnaEnum")
print("%s ID properties" % ob)
printProp(ob, '["myRnaInt"]')
printProp(ob, '["myRnaFloat"]')
printProp(ob, '["myRnaString"]')
printProp(ob, '["myRnaBool"]')
printProp(ob, '["myRnaEnum"]')
print("%s ID properties" % ob.data)
printProp(ob.data, '["MyIdInt"]')
printProp(ob.data, '["MyIdFloat"]')
printProp(ob.data, '["MyIdString"]')
printProp(ob.data, '["MyIdBool"]')
```

This script prints the following text to the terminal.

```
<bpy_struct, Object("Cube")> RNA properties
    Cube.myRnaInt does not exist
    Cube.myRnaFloat does not exist
    Cube.myRnaString does not exist
    Cube.myRnaBool does not exist
    Cube.myRnaEnum does not exist
<bpy_struct, Object("Cube")> ID properties
    Cube["myRnaInt"] = -99
    Cube["myRnaFloat"] = 1.0
    Cube["myRnaString"] = I am an RNA prop
    Cube["myRnaBool"] = 1
    Cube["myRnaEnum"] = 2
<bpy_struct, Mesh("Cube.001")> ID properties
    Cube.001["MyIdInt"] = 4711
    Cube.001["MyIdFloat"] = 666.777
```

```
Cube.001["MyIdString"] = I am an ID prop
Cube.001["MyIdBool"] = 1
```

If we restore the property declarations, the ID properties are converted into RNA properties again.

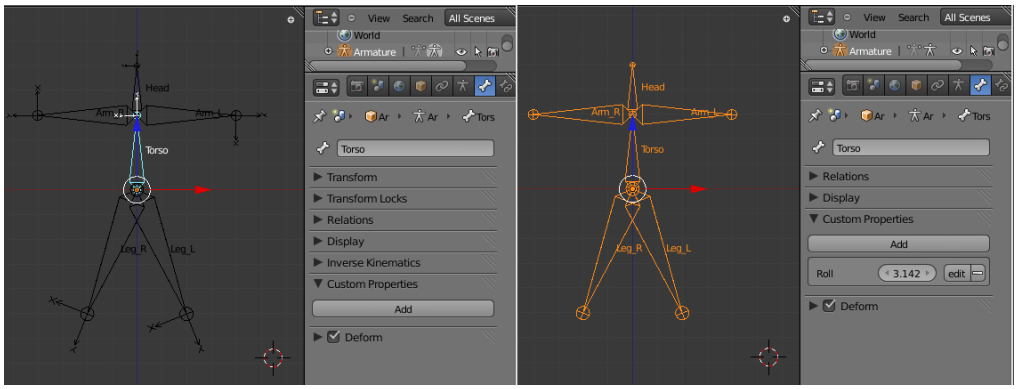
8.2 Bone roll

This program expects that the active object is an armature. It stores the roll angle of each editbone as a property of the corresponding bone, and finally prints the property values in the terminal. When executed with the armature in the picture selected, the terminal output is as follows.

```
 Head   3.1416
Arm_L   1.5708
Leg_R  -2.7646
Leg_L   2.7646
Arm_R  -1.5708
Torso   3.1416
```

Note that the property values are in radians. Angles are displayed in the viewport in degrees, but when accessed from Python they are expressed in radians. However, the Roll property is just some float property, and Blender does not know that is supposed to be an angle.

To find the property in the user interface, we need to select the bone in pose mode and then toggle into edit mode, as shown in the picture.



This code is actually somewhat useful for a script that retargets motion capture data. In order to do so properly, we need to know the roll angles. However, they

can not be found if the armature has been linked into another file and proxified. To access the roll angle `rig.data.edit_bones[name].roll`, the armature must be toggled into edit mode, which is not possible with linked assets. But if the script has been executed in the file where the armature is defined, the Roll property can be accessed from the linking file as `rig.pose.bones[name].bone["Roll"]`.

```
#-----------------------------------------------------------
# File bone_roll.py
#-----------------------------------------------------------
import bpy

def createBoneRollProps(rig):
    if rig.type != 'ARMATURE':
        raise NameError("Object not an armature")
    bpy.context.scene.objects.active = rig
    try:
        bpy.ops.object.mode_set(mode='EDIT')
        editable = (len(rig.data.edit_bones) > 0)
    except:
        editable = False

    rolls = {}
    if editable:
        for eb in rig.data.edit_bones:
            rolls[eb.name] = eb.roll
        bpy.ops.object.mode_set(mode='POSE')
        for pb in rig.pose.bones:
            pb.bone["Roll"] = rolls[pb.name]
    else:
        try:
            bpy.ops.object.mode_set(mode='POSE')
        except:
            raise NameError("Armature is not posable. Create proxy")
        for pb in rig.pose.bones:
            try:
                rolls[pb.name] = pb.bone["Roll"]
            except:
                raise NameError("Create roll props in asset file")
    return rolls

rolls = createBoneRollProps(bpy.context.object)
for (bname, roll) in rolls.items():
        print("  %16s %8.4f" % (bname, roll))
```

9 Interface

Most scripts need to communicate with the user in some way. A script may be invoked from a menu or from a button in a panel, and it may take its input from sliders, checkboxes, drop-down menus or inputs boxes. User interface elements are implemented as Python classes. Two types of interface elements are discussed in these notes:

- A panel is a class derived from bpy.types.Panel. It has properties and a draw function, which is called every time the panel is redrawn.
- An operator is a class derived from bpy.types.Operator. It has properties, an execute function, and optionally an invoke function. Operators can be registred to make them appear in menus. In particular, a button is an operator. When you press the button, its execute function is called.

Both panels and operators must be registered before they can be used. The simplest way to register everything in a file is to end it with a call to `bpy.utils.register_module(__name__)`.

The interface part of the API is probably less stable than other parts, so the code in this section may break in future releases.

9.1 Panels and buttons

This program adds five different panels to the user interface in different places. Each panel has a name and a button. The same operator is used for all buttons, but the text on it is can be changed with the text argument. When you press the button, Blender prints a greeting in the terminal.

The button operator can be invoked without arguments, as in the first panel:

```
self.layout.operator("hello.hello")
```

Blender will then search for an operator with the bl_idname `hello.hello` and place it in the panel. The text on the button defaults to its bl_label, i.e. `Say Hello`. The `OBJECT_OT_HelloButton` class also has a custom string property called `country`. It can be used for passing arguments to the button. If the operator is invoked without argument, the `country` property defaults to the empty string.

A bl_idname must be a string containing only lowercase letters, digits and underscores, plus exactly one dot; `hello.hello` satisfies these criteria. Apart from that there are apparently no restrictions in the bl_idname.

The button's default appearance and behaviour can be modified. Let us invoke the button in the following way:

```
self.layout.operator("hello.hello", text='Hej').country = "Sweden"
```

The text on this button is Hej, and the value of the country property is "Sweden". When we press this button, Blender prints the following to the terminal window.

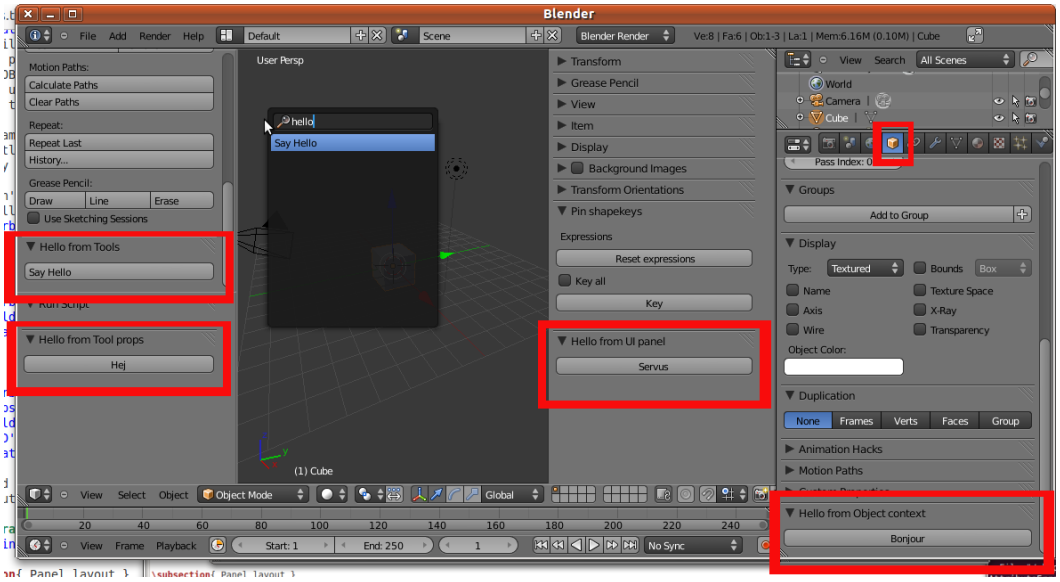
```
Hello world from Sweden!
```

At the end of the file, everything is registered with a call to `bpy.utils.register_module(__name__)`.

Our newly defined button operator can now be used as any other Blender operator. Here is a session from the Blender's python console:

```
>>> bpy.ops.hello.hello(country = "USA")
Hello world from USA!
{'FINISHED'}
```

Another way to invoke our new operator is to hit spacebar. A selector with all available operators pops up at the mouse location. Prune the selection by typing a substring of our operator's bl_label in the edit box. The operator with default parameters is executed and Hello world! is printed in the terminal window.



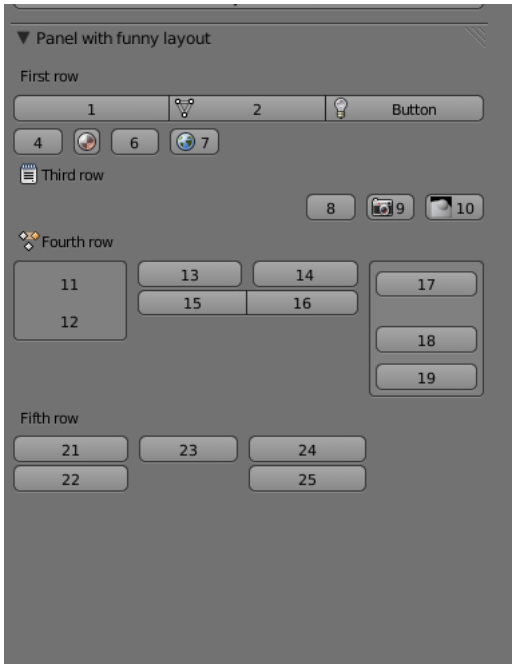
```
#-----------------------------------------------------------
# File hello.py
#-----------------------------------------------------------
import bpy

#
#    Menu in tools region
#
class ToolsPanel(bpy.types.Panel):
    bl_label = "Hello from Tools"
    bl_space_type = "VIEW_3D"
    bl_region_type = "TOOLS"

    def draw(self, context):
        self.layout.operator("hello.hello")

#
#    Menu in toolprops region
#
class ToolPropsPanel(bpy.types.Panel):
    bl_label = "Hello from Tool props"
    bl_space_type = "VIEW_3D"
    bl_region_type = "TOOL_PROPS"

    def draw(self, context):
        self.layout.operator("hello.hello", text='Hej').country = "Sweden"

#
#    Menu in UI region
#
class UIPanel(bpy.types.Panel):
    bl_label = "Hello from UI panel"
    bl_space_type = "VIEW_3D"
    bl_region_type = "UI"

    def draw(self, context):
        self.layout.operator("hello.hello", text='Servus')

#
#    Menu in window region, object context
#
class ObjectPanel(bpy.types.Panel):
    bl_label = "Hello from Object context"
    bl_space_type = "PROPERTIES"
    bl_region_type = "WINDOW"
    bl_context = "object"
```

```
    def draw(self, context):
        self.layout.operator("hello.hello", text='Bonjour').country = "France"

#
#    Menu in window region, material context
#
class MaterialPanel(bpy.types.Panel):
    bl_label = "Hello from Material context"
    bl_space_type = "PROPERTIES"
    bl_region_type = "WINDOW"
    bl_context = "material"

    def draw(self, context):
        self.layout.operator("hello.hello", text='Ciao').country = "Italy"

#
#    The Hello button prints a message in the console
#
class OBJECT_OT_HelloButton(bpy.types.Operator):
    bl_idname = "hello.hello"
    bl_label = "Say Hello"
    country = bpy.props.StringProperty()

    def execute(self, context):
        if self.country == '':
            print("Hello world!")
        else:
            print("Hello world from %s!" % self.country)
        return{'FINISHED'}

#
# Registration
#   All panels and operators must be registered with Blender; otherwise
#   they do not show up. The simplest way to register everything in the
#   file is with a call to bpy.utils.register_module(__name__).
#

bpy.utils.register_module(__name__)
```

9.2 Panel layout and several arguments

This program illustrates how to organize your panel layout. When the script is run, a panel is created in the tool props area, with buttons placed in a non-trivial

fashion.



The script also shows one method to send several arguments to an operator. The `OBJECT_OT_Button` class has two properties, `number` and `row`, and prints the values of these properties to the terminal. Being integer properties, they both default to 0 if not set. Thus, if we press buttons 7, 8 and 23, the script prints

```
Row 0 button 7
Row 3 button 0
Row 0 button 0
```

But what if we want to set both the `number` and `row` properties, i.e. invoke the operator with two arguments? This can not be done directly, but we can create a third property `loc`, which is a string that is parsed by the operator if non-zero. If we press button 13, the script prints

```
Row 4 button 13
```

This method can also be used to send more complicated data structures to an operator. Alternatively, we can use global variables for this purpose, cf subsection 9.10.

```
#----------------------------------------------------------
# File layout.py
#----------------------------------------------------------
import bpy

#   Layout panel
class LayoutPanel(bpy.types.Panel):
    bl_label = "Panel with funny layout"
    bl_space_type = "VIEW_3D"
    bl_region_type = "TOOL_PROPS"

    def draw(self, context):
        layout = self.layout

        layout.label("First row")
        row = layout.row(align=True)
        row.alignment = 'EXPAND'
        row.operator("my.button", text="1").number=1
        row.operator("my.button", text="2", icon='MESH_DATA').number=2
        row.operator("my.button", icon='LAMP_DATA').number=3

        row = layout.row(align=False)
        row.alignment = 'LEFT'
        row.operator("my.button", text="4").number=4
        row.operator("my.button", text="", icon='MATERIAL').number=5
        row.operator("my.button", text="6", icon='BLENDER').number=6
        row.operator("my.button", text="7", icon='WORLD').number=7

        layout.label("Third row", icon='TEXT')
        row = layout.row()
        row.alignment = 'RIGHT'
        row.operator("my.button", text="8").row=3
        row.operator("my.button", text="9", icon='SCENE').row=3
        row.operator("my.button", text="10", icon='BRUSH_INFLATE').row=3

        layout.label("Fourth row", icon='ACTION')
        row = layout.row()
        box = row.box()
        box.operator("my.button", text="11", emboss=False).loc="4 11"
        box.operator("my.button", text="12", emboss=False).loc="4 12"
        col = row.column()
        subrow = col.row()
        subrow.operator("my.button", text="13").loc="4 13"
        subrow.operator("my.button", text="14").loc="4 14"
        subrow = col.row(align=True)
        subrow.operator("my.button", text="15").loc="4 15"
```

```
        subrow.operator("my.button", text="16").loc="4 16"
        box = row.box()
        box.operator("my.button", text="17").number=17
        box.separator()
        box.operator("my.button", text="18")
        box.operator("my.button", text="19")

        layout.label("Fifth row")
        row = layout.row()
        split = row.split(percentage=0.25)
        col = split.column()
        col.operator("my.button", text="21").loc="5 21"
        col.operator("my.button", text="22")
        split = split.split(percentage=0.3)
        col = split.column()
        col.operator("my.button", text="23")
        split = split.split(percentage=0.5)
        col = split.column()
        col.operator("my.button", text="24")
        col.operator("my.button", text="25")

#   Button
class OBJECT_OT_Button(bpy.types.Operator):
    bl_idname = "my.button"
    bl_label = "Button"
    number = bpy.props.IntProperty()
    row = bpy.props.IntProperty()
    loc = bpy.props.StringProperty()

    def execute(self, context):
        if self.loc:
            words = self.loc.split()
            self.row = int(words[0])
            self.number = int(words[1])
        print("Row %d button %d" % (self.row, self.number))
        return{'FINISHED'}

#    Registration
bpy.utils.register_module(__name__)
```

9.3 Panel properties

Properties were discussed in section 8, but we did not explain how to display custom properties in a panel. This script does exactly that. An RNA property

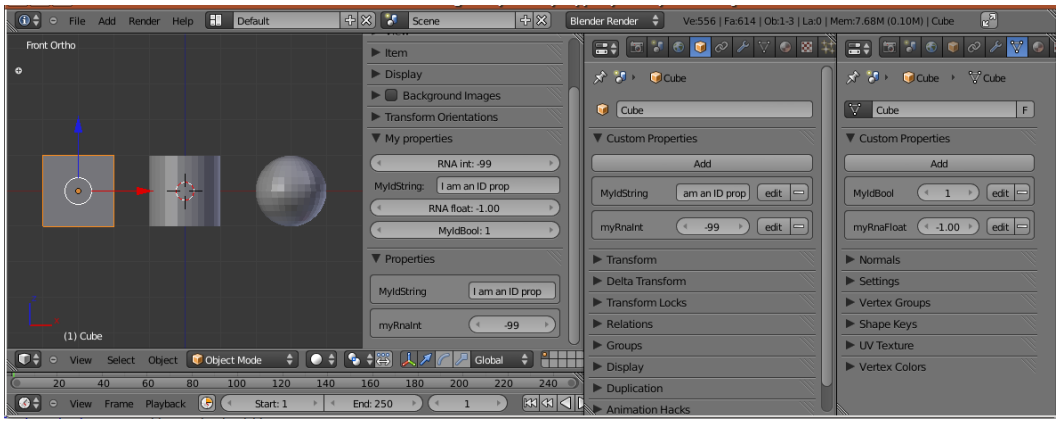
is displayed with the syntax

```
layout.prop(ob, 'myRnaInt')
```

An ID property is displayed with

```
layout.prop(ob, '["MyIdString"]')
```

Note that the panel is registered explicitly with `bpy.utils.register_class(MyPropPanel)`, instead of using register_module to register everything. Which method is used does not matter in this example, because `MyPropPanel` is the only thing that needs to be registred.

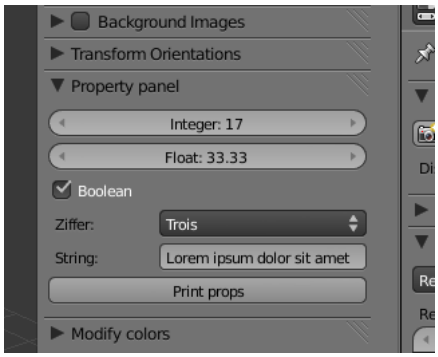



```
#----------------------------------------------------------
# File panel_props.py
#----------------------------------------------------------
import bpy
from bpy.props import *

# Clean the scene and create some objects
bpy.ops.object.select_by_type(type='MESH')
bpy.ops.object.delete()
bpy.ops.mesh.primitive_cube_add(location=(-3,0,0))
cube = bpy.context.object
bpy.ops.mesh.primitive_cylinder_add(location=(0,0,0))
cyl = bpy.context.object
bpy.ops.mesh.primitive_uv_sphere_add(location=(3,0,0))
sphere = bpy.context.object

# Define an RNA prop for every object
```

```
bpy.types.Object.myRnaInt = IntProperty(
    name="RNA int",
    min = -100, max = 100,
    default = 33)

# Define an RNA prop for every mesh
bpy.types.Mesh.myRnaFloat = FloatProperty(
    name="RNA float",
    default = 12.345)

# Set the cube's RNA props
cube.myRnaInt = -99
cube.data.myRnaFloat = -1

# Create ID props by setting them.
cube["MyIdString"] = "I am an ID prop"
cube.data["MyIdBool"] = True

#    Property panel
class MyPropPanel(bpy.types.Panel):
    bl_label = "My properties"
    bl_space_type = "VIEW_3D"
    bl_region_type = "UI"

    def draw(self, context):
        ob = context.object
        if not ob:
            return
        layout = self.layout
        layout.prop(ob, 'myRnaInt')
        try:
            ob["MyIdString"]
            layout.prop(ob, '["MyIdString"]')
        except:
            pass
        if ob.type == 'MESH':
            me = ob.data
            layout.prop(me, 'myRnaFloat')
            try:
                me["MyIdBool"]
                layout.prop(me, '["MyIdBool"]')
            except:
                pass

# Registration
bpy.utils.register_class(MyPropPanel)
```

9.4 Using scene properties to store information

This program lets the user input various kind of information, which is then sent from the panel to the buttons. The mechanism is to use user-defined RNA properties, which can be set by the panel and read by the buttons. All kind of Blender data can have properties. Global properties which are not directly associated with any specific object can conveniently be stored in the current scene. Note however that they will be lost if you switch to a new scene.




```
#----------------------------------------------------------
# File scene_props.py
#----------------------------------------------------------
import bpy
from bpy.props import *

#
#    Store properties in the active scene
#
def initSceneProperties(scn):
    bpy.types.Scene.MyInt = IntProperty(
        name = "Integer",
        description = "Enter an integer")
    scn['MyInt'] = 17

    bpy.types.Scene.MyFloat = FloatProperty(
        name = "Float",
        description = "Enter a float",
        default = 33.33,
        min = -100,
```

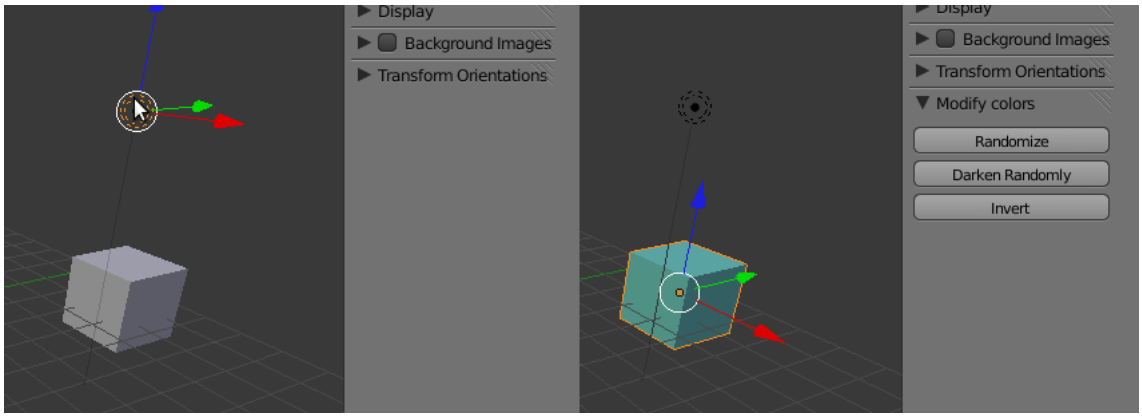
```
        max = 100)

    bpy.types.Scene.MyBool = BoolProperty(
        name = "Boolean",
        description = "True or False?")
    scn['MyBool'] = True

    bpy.types.Scene.MyEnum = EnumProperty(
        items = [('Eine', 'Un', 'One'),
                 ('Zwei', 'Deux', 'Two'),
                 ('Drei', 'Trois', 'Three')],
        name = "Ziffer")
    scn['MyEnum'] = 2

    bpy.types.Scene.MyString = StringProperty(
        name = "String")
    scn['MyString'] = "Lorem ipsum dolor sit amet"
    return

initSceneProperties(bpy.context.scene)

#
#    Menu in UI region
#
class UIPanel(bpy.types.Panel):
    bl_label = "Property panel"
    bl_space_type = "VIEW_3D"
    bl_region_type = "UI"

    def draw(self, context):
        layout = self.layout
        scn = context.scene
        layout.prop(scn, 'MyInt', icon='BLENDER', toggle=True)
        layout.prop(scn, 'MyFloat')
        layout.prop(scn, 'MyBool')
        layout.prop(scn, 'MyEnum')
        layout.prop(scn, 'MyString')
        layout.operator("idname_must.be_all_lowercase_and_contain_one_dot")

#
#    The button prints the values of the properites in the console.
#

class OBJECT_OT_PrintPropsButton(bpy.types.Operator):
    bl_idname = "idname_must.be_all_lowercase_and_contain_one_dot"
    bl_label = "Print props"
```

```
    def execute(self, context):
        scn = context.scene
        printProp("Int:    ", 'MyInt', scn)
        printProp("Float:  ", 'MyFloat', scn)
        printProp("Bool:   ", 'MyBool', scn)
        printProp("Enum:   ", 'MyEnum', scn)
        printProp("String: ", 'MyString', scn)
        return{'FINISHED'}

def printProp(label, key, scn):
    try:
        val = scn[key]
    except:
        val = 'Undefined'
    print("%s %s" % (key, val))

#    Registration
bpy.utils.register_module(__name__)
```

9.5 Polling

A script often only works in some specific context, e.g. when an object of the right kind is active. E.g., a script that manipulate mesh vertices can not do anything meaningful if the active object is an armature.

This program adds a panel which modifies the active object's material. The panel resides in the user interface section (open with N-key), but it is only visible if the active object is a mesh with at least one material. Checking how many materials the active object has is done by poll(). This is not a function but rather a class method, indicated by the command `@classmethod` above the definition. So what is the difference between a function and a class method? Don't ask me! All I know is that the code works with the `@classmethod` line in place, but not without.



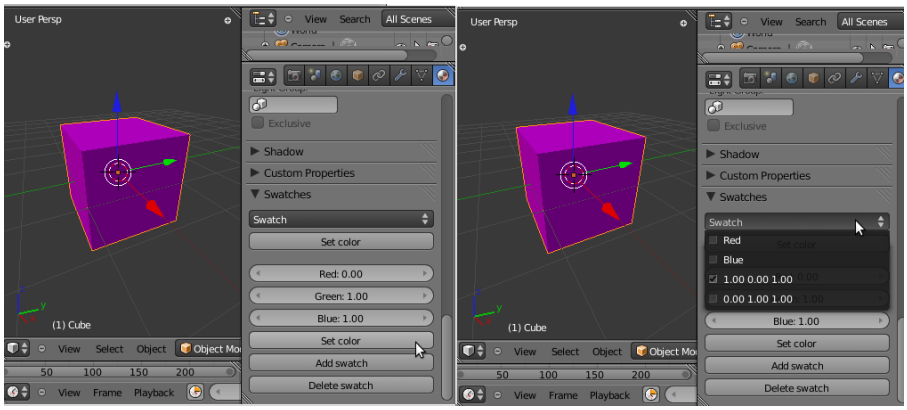

```
#----------------------------------------------------------
# File poll.py
#----------------------------------------------------------
import bpy, random

#
#    Menu in UI region
#
class ColorPanel(bpy.types.Panel):
    bl_label = "Modify colors"
    bl_space_type = "VIEW_3D"
    bl_region_type = "UI"

    @classmethod
    def poll(self, context):
        if context.object and context.object.type == 'MESH':
            return len(context.object.data.materials)

    def draw(self, context):
        layout = self.layout
        scn = context.scene
        layout.operator("random.button")
        layout.operator("darken_random.button")
        layout.operator("invert.button")

#
#    The three buttons
#

class RandomButton(bpy.types.Operator):
    bl_idname = "random.button"
    bl_label = "Randomize"
```

```
    def execute(self, context):
        mat = context.object.data.materials[0]
        for i in range(3):
            mat.diffuse_color[i] = random.random()
        return{'FINISHED'}

class DarkenRandomButton(bpy.types.Operator):
    bl_idname = "darken_random.button"
    bl_label = "Darken Randomly"

    def execute(self, context):
        mat = context.object.data.materials[0]
        for i in range(3):
            mat.diffuse_color[i] *= random.random()
        return{'FINISHED'}

class InvertButton(bpy.types.Operator):
    bl_idname = "invert.button"
    bl_label = "Invert"

    def execute(self, context):
        mat = context.object.data.materials[0]
        for i in range(3):
            mat.diffuse_color[i] = 1 - mat.diffuse_color[i]
        return{'FINISHED'}

#    Registration
bpy.utils.register_module(__name__)
```

9.6 Dynamic drop-down menus

This program adds a panel with a drop-down menu to the User interface panel. In the beginning the menu contains three items: red, green and blue. There are two buttons labelled Set color. The upper one changes the color of the active object to the color selected in the drop-down menu, and the lower one sets it to the color specified by the three sliders. Colors can be added to and deleted from the drop-down menu.

Also note that polling works for buttons as well; the Set color button is greyed out unless the active object is a mesh with at least one material.




```
#------------------------------------------------------------
# File swatches.py
#------------------------------------------------------------
import bpy
from bpy.props import *

theSwatches = [
    ("1 0 0" , "Red" , "1 0 0"),
    ("0 1 0" , "Green" , "0 1 0"),
    ("0 0 1" , "Blue" , "0 0 1")]

def setSwatches():
    global theSwatches
    bpy.types.Object.my_swatch = EnumProperty(
        items = theSwatches,
        name = "Swatch")

setSwatches()

bpy.types.Object.my_red = FloatProperty(
    name = "Red", default = 0.5,
    min = 0, max = 1)

bpy.types.Object.my_green = FloatProperty(
    name = "Green", default = 0.5,
    min = 0, max = 1)

bpy.types.Object.my_blue = FloatProperty(
    name = "Blue", default = 0.5,
    min = 0, max = 1)

def findSwatch(key):
```

```
    for n,swatch in enumerate(theSwatches):
        (key1, name, colors) = swatch
        if key == key1:
            return n
    raise NameError("Unrecognized key %s" % key)

#    Swatch Panel
class SwatchPanel(bpy.types.Panel):
    bl_label = "Swatches"
    #bl_idname = "myPanelID"
    bl_space_type = "PROPERTIES"
    bl_region_type = "WINDOW"
    bl_context = "material"

    def draw(self , context):
        layout = self.layout
        ob = context.active_object
        layout.prop_menu_enum(ob, "my_swatch")
        layout.operator("swatches.set").swatch=True
        layout.separator()
        layout.prop(ob, "my_red")
        layout.prop(ob, "my_green")
        layout.prop(ob, "my_blue")
        layout.operator("swatches.set").swatch=False
        layout.operator("swatches.add")
        layout.operator("swatches.delete")

#    Set button
class OBJECT_OT_SetButton(bpy.types.Operator):
    bl_idname = "swatches.set"
    bl_label = "Set color"
    swatch = bpy.props.BoolProperty()

    @classmethod
    def poll(self, context):
        if context.object and context.object.type == 'MESH':
            return len(context.object.data.materials)

    def execute(self, context):
        global theSwatches
        ob = context.object
        if self.swatch:
            n = findSwatch(ob.my_swatch)
            (key, name, colors) = theSwatches[n]
            words = colors.split()
            color = (float(words[0]), float(words[1]), float(words[2]))
```

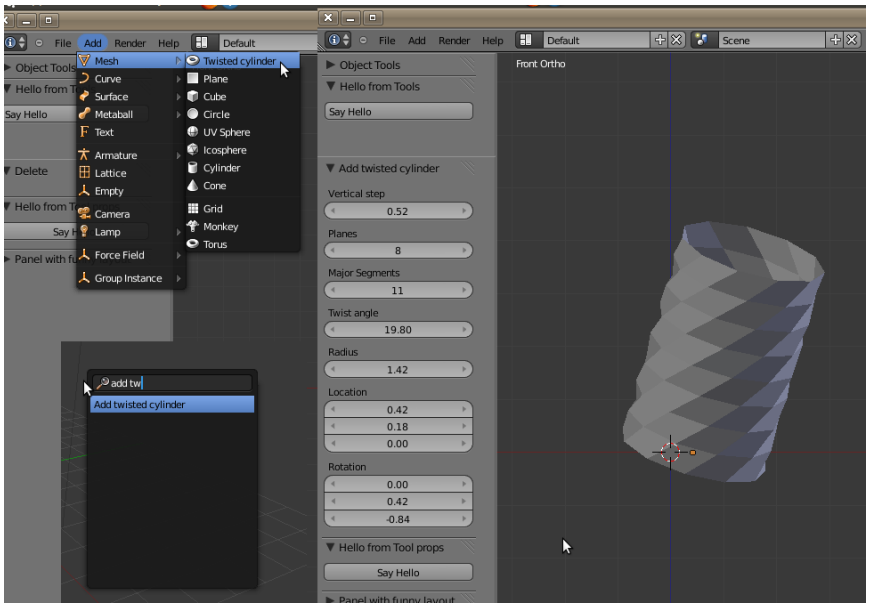
```
        else:
            color = (ob.my_red, ob.my_green, ob.my_blue)
        ob.data.materials[0].diffuse_color = color
        return{'FINISHED'}

#    Add button
class OBJECT_OT_AddButton(bpy.types.Operator):
    bl_idname = "swatches.add"
    bl_label = "Add swatch"

    def execute(self, context):
        global theSwatches
        ob = context.object
        colors = "%.2f %.2f %.2f" % (ob.my_red, ob.my_green, ob.my_blue)
        theSwatches.append((colors, colors, colors))
        setSwatches()
        return{'FINISHED'}

#    Delete button
class OBJECT_OT_DeleteButton(bpy.types.Operator):
    bl_idname = "swatches.delete"
    bl_label = "Delete swatch"

    def execute(self, context):
        global theSwatches
        n = findSwatch(context.object.my_swatch)
        theSwatches.pop(n)
        setSwatches()
        return{'FINISHED'}

#    Registration
bpy.utils.register_module(__name__)
```

9.7 Adding an operator and appending it to a menu

The only operators encountered so far were simple buttons. In this program we make a more complicated operator, which creates a twisted cylinder.

To invoke the operator, press spacebar and type in "Add twisted cylinder"; Blender suggests matching operator names while you type. The cylinder have several options, which appear in the Tool props area (below the Tools section) once the cylinder has been created. These can be modified interactively and the result is immediately displayed in the viewport.

The last part of the script registers the script. Instead of pressing spacebar, you can now invoke the script more conveniently from the Add ¿ Mesh submenu. If we had used `append` instead of `prepend` in `register()`, the entry had appeared at the bottom instead of at the top of the menu.




```
#------------------------------------------------------------
# File twisted.py
#------------------------------------------------------------
import bpy, math

def addTwistedCylinder(context, r, nseg, vstep, nplanes, twist):
    verts = []
    faces = []
    w = 2*math.pi/nseg
    a = 0
    da = twist*math.pi/180
    for j in range(nplanes+1):
        z = j*vstep
        a += da
        for i in range(nseg):
            verts.append((r*math.cos(w*i+a), r*math.sin(w*i+a), z))
            if j > 0:
                i0 = (j-1)*nseg
                i1 = j*nseg
                for i in range(1, nseg):
                    faces.append((i0+i-1, i0+i, i1+i, i1+i-1))
```

```
                faces.append((i0+nseg-1, i0, i1, i1+nseg-1))

    me = bpy.data.meshes.new("TwistedCylinder")
    me.from_pydata(verts, [], faces)
    ob = bpy.data.objects.new("TwistedCylinder", me)
    context.scene.objects.link(ob)
    context.scene.objects.active = ob
    return ob

#
#    User interface
#

from bpy.props import *

class MESH_OT_primitive_twisted_cylinder_add(bpy.types.Operator):
    '''Add a twisted cylinder'''
    bl_idname = "mesh.primitive_twisted_cylinder_add"
    bl_label = "Add twisted cylinder"
    bl_options = {'REGISTER', 'UNDO'}

    radius = FloatProperty(name="Radius",
            default=1.0, min=0.01, max=100.0)
    nseg = IntProperty(name="Major Segments",
            description="Number of segments for one layer",
            default=12, min=3, max=256)
    vstep = FloatProperty(name="Vertical step",
            description="Distance between subsequent planes",
            default=1.0, min=0.01, max=100.0)
    nplanes = IntProperty(name="Planes",
            description="Number of vertical planes",
            default=4, min=2, max=256)
    twist = FloatProperty(name="Twist angle",
            description="Angle between subsequent planes (degrees)",
            default=15, min=0, max=90)

    location = FloatVectorProperty(name="Location")
    rotation = FloatVectorProperty(name="Rotation")
    # Note: rotation in radians!

    def execute(self, context):
        ob = addTwistedCylinder(context,
            self.radius, self.nseg, self.vstep, self.nplanes, self.twist)
        ob.location = self.location
        ob.rotation_euler = self.rotation
        #context.scene.objects.link(ob)
```

```
        #context.scene.objects.active = ob
        return {'FINISHED'}

#
#    Registration
#    Makes it possible to access the script from the Add > Mesh menu
#

def menu_func(self, context):
    self.layout.operator("mesh.primitive_twisted_cylinder_add",
        text="Twisted cylinder",
        icon='MESH_TORUS')

def register():
   bpy.utils.register_module(__name__)
   bpy.types.INFO_MT_mesh_add.prepend(menu_func)

def unregister():
    bpy.utils.unregister_module(__name__)
    bpy.types.INFO_MT_mesh_add.remove(menu_func)

if __name__ == "__main__":
    register()
```

9.8 A modal operator

The following example is taken directly from the API documentation, as are the next few examples.

A modal operator defines a `Operator.modal` function which running, handling events until it returns {`'FINISHED'`} or {`'CANCELLED'`}. Grab, Rotate, Scale and Fly-Mode are examples of modal operators. They are especially useful for interactive tools, your operator can have its own state where keys toggle options as the operator runs.

When the operator in this example is invoked, it adds a modal handler to itself with the call `context.window_manager.modal_handler_add(self)`. After that, the active object keeps moving in the XY-plane as long as we move the mouse. To quit, press either mouse button or the escape key.

The modal method triggers on three kinds of events:

1. A mouse move moves the active object.

2. Press the left mouse button to confirm and exit to normal mode. The object is left at its new position.

3. Press the right moust button or the escape key to cancel and exit to normal mode. The object reverts to its original position.

It is important that there is some way to exit to normal mode. If the `modal()` function always returns {`'RUNNING_MODAL'`}, the script will be stuck in an infinite loop and you have to restart Blender.

A modal operator defines two special methods called `__init()__` and `__del()__`, which are called when modal operation starts and stops, respectively.

Run the script. The active object moves in the XY-plane when you move the mouse. The script also create a panel with a button, from which you can also invoke the modal operator.

```
#------------------------------------------------------------
# File modal.py
# from API documentation
#------------------------------------------------------------
import bpy

class MyModalOperator(bpy.types.Operator):
    bl_idname = "mine.modal_op"
    bl_label = "Move in XY plane"

    def __init__(self):
        print("Start moving")

    def __del__(self):
        print("Moved from (%d %d) to (%d %d)" %
            (self.init_x, self.init_y, self.x, self.y))

    def execute(self, context):
        context.object.location.x = self.x / 100.0
        context.object.location.y = self.y / 100.0

    def modal(self, context, event):
        if event.type == 'MOUSEMOVE':  # Apply
            self.x = event.mouse_x
            self.y = event.mouse_y
            self.execute(context)
        elif event.type == 'LEFTMOUSE':  # Confirm
            return {'FINISHED'}
        elif event.type in ('RIGHTMOUSE', 'ESC'):  # Cancel
```

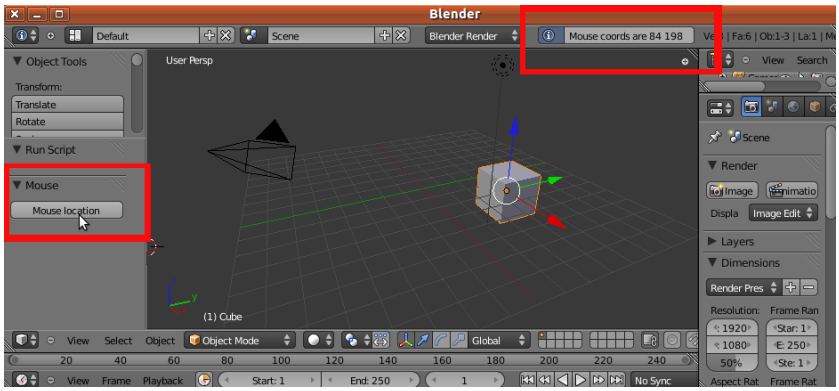
```
            return {'CANCELLED'}

        return {'RUNNING_MODAL'}

    def invoke(self, context, event):
        self.x = event.mouse_x
        self.y = event.mouse_y
        self.init_x = self.x
        self.init_y = self.y
        self.execute(context)

        print(context.window_manager.modal_handler_add(self))
        return {'RUNNING_MODAL'}

#
#    Panel in tools region
#
class MyModalPanel(bpy.types.Panel):
    bl_label = "My modal operator"
    bl_space_type = "VIEW_3D"
    bl_region_type = "TOOLS"

    def draw(self, context):
        self.layout.operator("mine.modal_op")

# Registration
bpy.utils.register_module(__name__)

# Automatically move active object on startup
bpy.ops.mine.modal_op('INVOKE_DEFAULT')
```

9.9 Invoke versus execute

This script illustrates the difference between invoke and execute. The invoking event is an argument of the `Operator.invoke` function, which sets the two integer properties `x` and `y` to the mouse location and calls the `Operator.execute` function. Alternatively, we can execture the operator and explicitly set `x` and `y` : `bpy.ops.wm.mouse_position(EXEC_DEFAULT, x=20, y=66)`

Instead of printing the mouse coordinates in the terminal window, the information is sent to the info panel in the upper right corner. This is a good place to display short notification, because the user does not have to look in another window, especially since the terminal/DOS window is not visible in all versions

of Blender. However, long messages are difficult to fit into the limited space of the info panel.



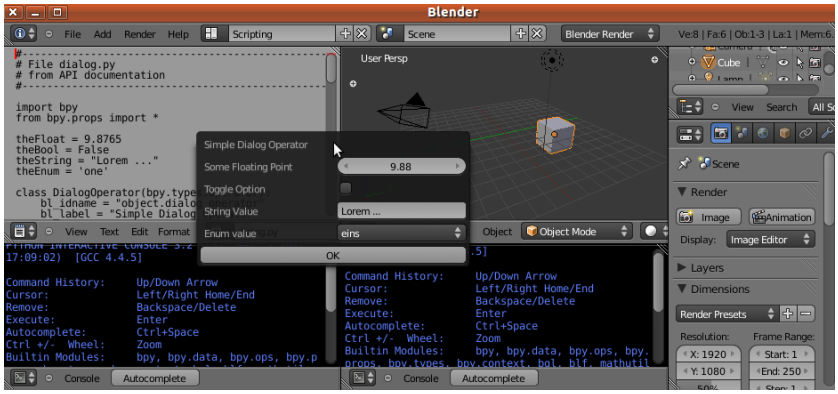

```
#------------------------------------------------------------
# File invoke.py
# from API documentation
#------------------------------------------------------------

import bpy

class SimpleMouseOperator(bpy.types.Operator):
    """ This operator shows the mouse location,
        this string is used for the tooltip and API docs
    """
    bl_idname = "wm.mouse_position"
    bl_label = "Mouse location"

    x = bpy.props.IntProperty()
    y = bpy.props.IntProperty()

    def execute(self, context):
        # rather then printing, use the report function,
        # this way the message appears in the header,
        self.report({'INFO'}, "Mouse coords are %d %d" % (self.x, self.y))
        return {'FINISHED'}

    def invoke(self, context, event):
        self.x = event.mouse_x
        self.y = event.mouse_y
        return self.execute(context)

#
```

```
#    Panel in tools region
#
class MousePanel(bpy.types.Panel):
    bl_label = "Mouse"
    bl_space_type = "VIEW_3D"
    bl_region_type = "TOOL_PROPS"

    def draw(self, context):
        self.layout.operator("wm.mouse_position")

#
# Registration
#   Not really necessary to register the class, because this happens
#   automatically when the module is registered. OTOH, it does not hurt either.
bpy.utils.register_class(SimpleMouseOperator)
bpy.utils.register_module(__name__)

# Automatically display mouse position on startup
bpy.ops.wm.mouse_position('INVOKE_DEFAULT')

# Another test call, this time call execute() directly with pre-defined settings.
#bpy.ops.wm.mouse_position('EXEC_DEFAULT', x=20, y=66)
```

9.10 A popup dialog

When this script is run a popup window appears, where you can set some properties. After you quit the popup window by moving the mouse outside of it, the properties are written both to the info window and to the console.

In subsection 9.2 we used a single string to send several arguments to an operator. Here we use global variables for the same purpose.



```
#----------------------------------------------------------
# File popup.py
# from API documentation
#----------------------------------------------------------

import bpy
from bpy.props import *

theFloat = 9.8765
theBool = False
theString = "Lorem ..."
theEnum = 'one'

class DialogOperator(bpy.types.Operator):
    bl_idname = "object.dialog_operator"
    bl_label = "Simple Dialog Operator"

    my_float = FloatProperty(name="Some Floating Point",
        min=0.0, max=100.0)
    my_bool = BoolProperty(name="Toggle Option")
    my_string = StringProperty(name="String Value")
    my_enum = EnumProperty(name="Enum value",
        items = [('one', 'eins', 'un'),
                 ('two', 'zwei', 'deux'),
                 ('three', 'drei', 'trois')])

    def execute(self, context):
        message = "%.3f, %d, '%s' %s" % (self.my_float,
            self.my_bool, self.my_string, self.my_enum)
        self.report({'INFO'}, message)
        print(message)
        return {'FINISHED'}

    def invoke(self, context, event):
        global theFloat, theBool, theString, theEnum
        self.my_float = theFloat
        self.my_bool = theBool
        self.my_string = theString
        self.my_enum = theEnum
        return context.window_manager.invoke_props_dialog(self)


bpy.utils.register_class(DialogOperator)

# Invoke the dialog when loading
bpy.ops.object.dialog_operator('INVOKE_DEFAULT')
```

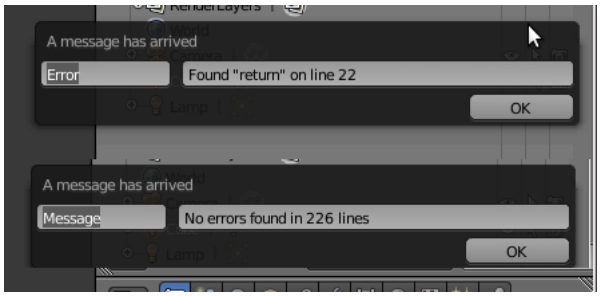
```
#
#    Panel in tools region
#
class DialogPanel(bpy.types.Panel):
    bl_label = "Dialog"
    bl_space_type = "VIEW_3D"
    bl_region_type = "UI"

    def draw(self, context):
        global theFloat, theBool, theString, theEnum
        theFloat = 12.345
        theBool = True
        theString = "Code snippets"
        theEnum = 'two'
        self.layout.operator("object.dialog_operator")

#
# Registration
bpy.utils.register_module(__name__)
```

9.11 An error dialog

As far as I know, Blender does not have any elegant means to notify the user if something has gone wrong. One can print a message in the terminal window or info panel, and then raise an exception. Most modern application would instead open a message box and display the error message. The following script uses the Blender API to create a popup dialog to notify the user.

The script scans a file. If the word `return` is found, the script opens a popup window to tell the user that an error has occurred and on which line. If there no such word in the entire file, a popup window displays the number of scanned lines.

At the time of writing, this script causes a memory leak which makes Blender unstable. This bug will hopefully be fixed soon.




```
#----------------------------------------------------------
# File error.py
# Simple error dialog
#----------------------------------------------------------

import bpy
from bpy.props import *

#
#   The error message operator. When invoked, pops up a dialog
#   window with the given message.
#
class MessageOperator(bpy.types.Operator):
    bl_idname = "error.message"
    bl_label = "Message"
    type = StringProperty()
    message = StringProperty()

    def execute(self, context):
        self.report({'INFO'}, self.message)
        print(self.message)
        return {'FINISHED'}

    def invoke(self, context, event):
        wm = context.window_manager
        return wm.invoke_popup(self, width=400, height=200)

    def draw(self, context):
        self.layout.label("A message has arrived")
        row = self.layout.split(0.25)
        row.prop(self, "type")
        row.prop(self, "message")
        row = self.layout.split(0.80)
        row.label("")
        row.operator("error.ok")
```

```
#
#   The OK button in the error dialog
#
class OkOperator(bpy.types.Operator):
    bl_idname = "error.ok"
    bl_label = "OK"
    def execute(self, context):
        return {'FINISHED'}

#
#   Opens a file select dialog and starts scanning the selected file.
#
class ScanFileOperator(bpy.types.Operator):
    bl_idname = "error.scan_file"
    bl_label = "Scan file for return"
    filepath = bpy.props.StringProperty(subtype="FILE_PATH")

    def execute(self, context):
        scanFile(self.filepath)
        return {'FINISHED'}

    def invoke(self, context, event):
        context.window_manager.fileselect_add(self)
        return {'RUNNING_MODAL'}

#
#   Scan the file. If a line contains the word "return", invoke error
#   dialog and quit. If reached end of file, display another message.
#
def scanFile(filepath):
    fp = open(filepath, "rU")
    n = 1
    for line in fp:
        words = line.split()
        if "return" in words:
            bpy.ops.error.message('INVOKE_DEFAULT',
                type = "Error",
                message = 'Found "return" on line %d' % n)
            return
        n += 1
    fp.close()
    bpy.ops.error.message('INVOKE_DEFAULT',
        type = "Message",
        message = "No errors found in %d lines" % n)
    return
```

```
# Register classes and start scan automatically
bpy.utils.register_class(OkOperator)
bpy.utils.register_class(MessageOperator)
bpy.utils.register_class(ScanFileOperator)
bpy.ops.error.scan_file('INVOKE_DEFAULT')
```

10 Blender add-ons

Until now we have only considered stand-alone scripts, which are executed from the text editor window. For an end user it is more convenient if the script is a Blender add-on, which can be enabled from the User preferences window. It is also possible to auto-load a script every time Blender starts.

In order for a script to be an add-on, it must be written in a special way. There must be a `bl_info` structure in the beginning of the file, and `register` and `unregister` functions must be defined at the end. Moreover, the script must be placed at a location where Blender looks for add-ons on upstart. This includes the `addons` and `addons-contrib` folders, which are located under the `2.57/scripts` subdirectory of the directory where Blender is located.

10.1 Shapekey pinning

This script can be executed from the text editor window as usual. However, it can also be accessed as a Blender add-on. The add-on info is specified in the `bl_info` dictonary in the beginning of the file.

```
bl\_info = {
    'name': 'Shapekey pinning',
    'author': 'Thomas Larsson',
    'version': (0, 1, 2),
    'blender': (2, 5, 7),
    'api': 35774,
    "location": "View3D > UI panel > Shapekey pinning",
    'description': 'Pin and key the shapekeys of a mesh',
    'warning': '',
    'wiki_url': 'http://blenderartists.org/forum/showthread.php?193908',
    'tracker_url': '',
    "support": 'COMMUNITY',
    "category": "3D View"}
```

The meaning of most slots in this dictionary is self-evident.

- name
  The name of the add-on.
- author
  The author's name.
- version
  The script version.
- blender
  Blender version
- api
  Revision that the script works with.
- location
  Where to find the buttons.
- description
  A description displayed as a tooltip and in documentation.
- warning
  A warning message. If not empty, a little warning sign will show up in the User preference window.
- wiki_url
  Link to the script's wiki page. Should really be on the Blender site, but here we are linking to a thread at blenderartists.org.
- tracker_url
  Link to the scripts bug tracker.
- support
  Official or community.
- category
  Script category, e.g. `"3D View"`,`"Import-Export"`, `"Add Mesh"` or `"Rigging"`. Correspond to categories in the User preferences window.

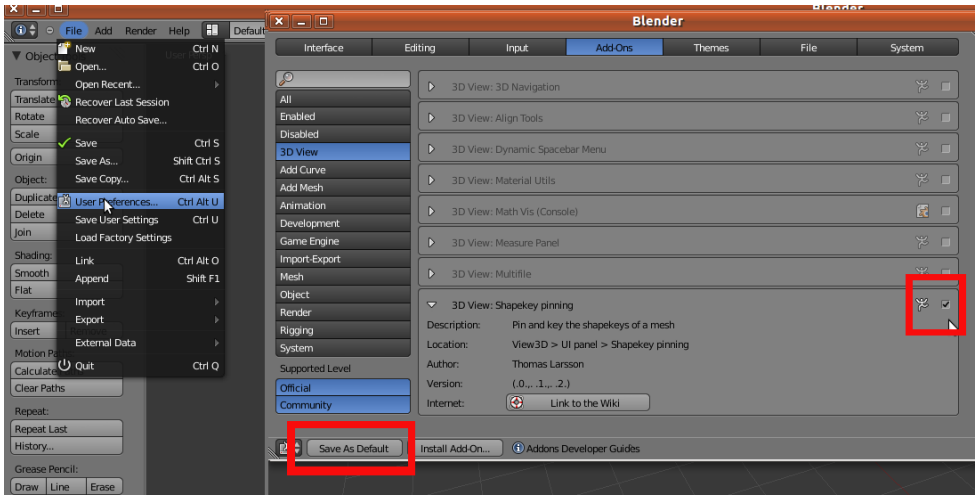
Many of the items can simply be omitted, as we will see in other examples below.

The second requirement on an add-on is that it defines `register()` and `unregister()` functions, which usually are placed at the end of the file. `register()` normally makes a call to `bpy.utils.register_module(__name__)`, which registers all classes defined in the file. It can also contain some custom initialization tasks. The present script also declares a custom RNA property. As we saw in subsection 8.1, declaration is necessary here because the bool property would appear as an integer otherwise.

```
def register():
    initialize()
    bpy.utils.register_module(__name__)

def unregister():
    bpy.utils.unregister_module(__name__)

if __name__ == "__main__":
    register()
```

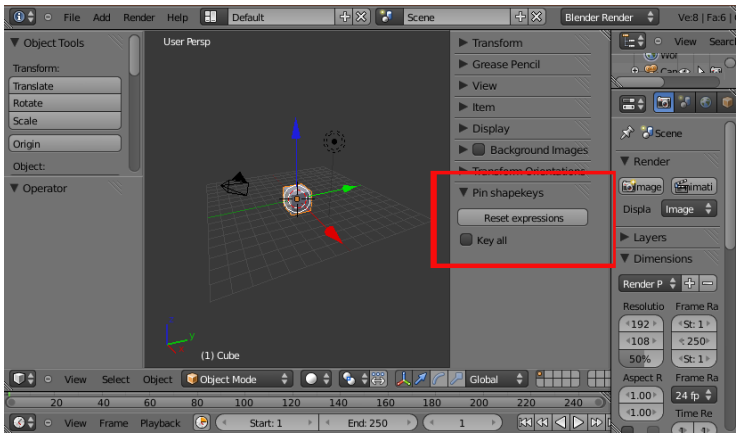
Unregistration is analogous to registration. The last lines makes it possible to run the script in stand-alone mode from the editor window. Even if the end user will never run the script from the editor, it is useful to have this possibility when debugging.

Copy the file to a location where Blender looks for add-ons. Restart Blender. Open the user preferences window from the File > User Preferences menu, and navigate to the Add-ons tab. Our script can be found at the bottom of the 3D View section.



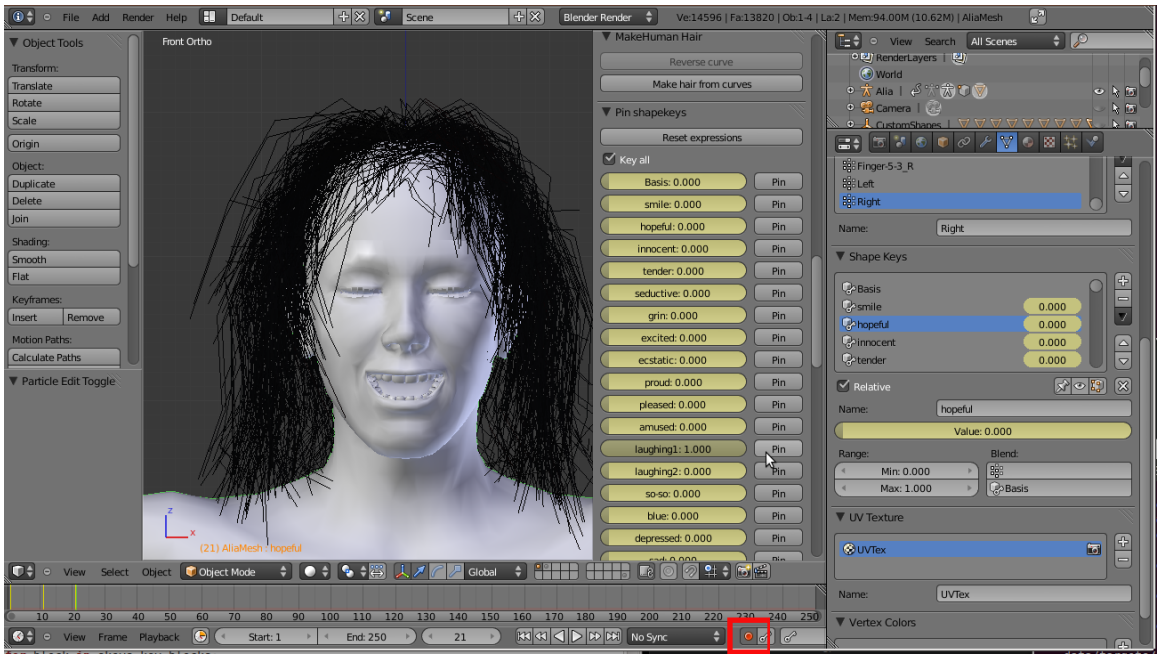
We recognize the fields in the `bl_info` dictionary. Enable the script by pressing the checkbox in the upper-right corner. If you want the add-on to load every time you start Blender, press the Save as default buttom at the bottom of the window.

After the add-on has been enabled, it appears in the UI panel.



The script itself displays the shapekeys of the active object in the UI panel. The default cube does not have any shapekeys. Instead we import a MakeHuman character which has a lot of facial expressions that are implemented as shapekeys. MakeHuman (`www.makehuman.org`) is a application which easily allows you to design a character. A fully rigged and textured character can then be exported to Blender using the MHX (MakeHuman eXchange) format. MHX files can be imported into Blender with the MHX importer, which is an add-on which is distributed with Blender.

What matters for the present example is that the MakeHuman mesh has a lot of shapekeys. If you press the Pin button right to the shapekey value, the shapekey is pinned, i.e. its value is set to one whereas the values of all other shapekeys are zero. If Autokey button in the timeline is set, a key is set for the shapekey value. If furthermore the Key all option in selected, keys are set for every shapekey in the mesh.




```
#----------------------------------------------------------
# File shapekey_pin.py
#----------------------------------------------------------

bl_info = {
    'name': 'Shapekey pinning',
    'author': 'Thomas Larsson',
    'version': '(0, 1, 2)',
    'blender': (2, 5, 7),
    "location": "View3D > UI panel > Shapekey pinning",
    'description': 'Pin and key the shapekeys of a mesh',
    'warning': '',
    'wiki_url': 'http://blenderartists.org/forum/showthread.php?193908',
    'tracker_url': '',
    "support": 'COMMUNITY',
     "category": "3D View"}

import bpy
from bpy.props import *

#
#    class VIEW3D_OT_ResetExpressionsButton(bpy.types.Operator):
#
class VIEW3D_OT_ResetExpressionsButton(bpy.types.Operator):
```

```
    bl_idname = "shapepin.reset_expressions"
    bl_label = "Reset expressions"

    def execute(self, context):
        keys = context.object.data.shape_keys
        if keys:
            for shape in keys.keys:
                shape.value = 0.0
        return{'FINISHED'}

#
#    class VIEW3D_OT_PinExpressionButton(bpy.types.Operator):
#

class VIEW3D_OT_PinExpressionButton(bpy.types.Operator):
    bl_idname = "shapepin.pin_expression"
    bl_label = "Pin"
    expression = bpy.props.StringProperty()

    def execute(self, context):
        skeys = context.object.data.shape_keys
        if skeys:
            frame = context.scene.frame_current
            for block in skeys.key_blocks:
                oldvalue = block.value
                block.value = 1.0 if block.name == self.expression else 0.0
                if (context.tool_settings.use_keyframe_insert_auto and
                    (context.scene.key_all or
                    (block.value > 0.01) or
                    (abs(block.value-oldvalue) > 0.01))):
                    block.keyframe_insert("value", index=-1, frame=frame)
        return{'FINISHED'}

#
#    class ExpressionsPanel(bpy.types.Panel):
#

class ExpressionsPanel(bpy.types.Panel):
    bl_label = "Pin shapekeys"
    bl_space_type = "VIEW_3D"
    bl_region_type = "UI"

    @classmethod
    def poll(cls, context):
        return context.object and (context.object.type == 'MESH')
```

```
    def draw(self, context):
        layout = self.layout
        layout.operator("shapepin.reset_expressions")
        layout.prop(context.scene, "key_all")
        skeys = context.object.data.shape_keys
        if skeys:
            for block in skeys.key_blocks:
                row = layout.split(0.75)
                row.prop(block, 'value', text=block.name)
                row.operator("shapepin.pin_expression",
                    text="Pin").expression = block.name
        return

#
#    initialize and register
#

def initialize():
    bpy.types.Scene.key_all = BoolProperty(
        name="Key all",
        description="Set keys for all shapes",
        default=False)

def register():
    initialize()
    bpy.utils.register_module(__name__)

def unregister():
    bpy.utils.unregister_module(__name__)

if __name__ == "__main__":
    register()
```

10.2 Simple BVH import

The BVH format is commonly used to transfer character animation, e.g. from mocap data. This program is a simple BVH importer, which builds a skeleton with the action described by the BVH file. It is implemented as a Blender add-on, which a `bl_info` dictionary at the top of the file and registration code at the bottom.

Once the script below has been executed, or enabled as an add-on, the simple BVH importer can be invoked from user interface panel (Ctrl-N). There are two options: a boolean choice to rotate the mesh 90 degrees (to make Z up), and a scale.

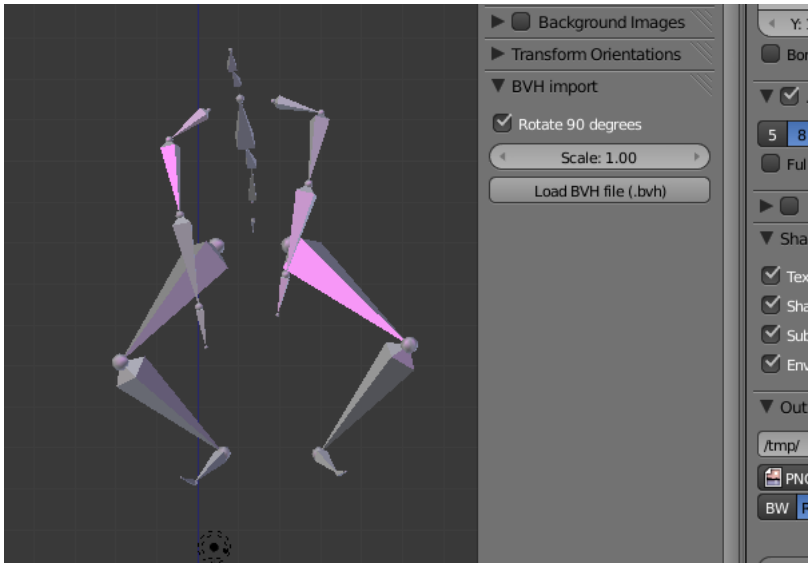
This program also illustrates how to invoke a file selector by pushing a button in a panel. The Load BVH button class inherits both from the `bpy.types.Operator` and `ImportHelper` base classes.

```
class OBJECT_OT_LoadBvhButton(bpy.types.Operator, ImportHelper):
```

The (possibly undocumented) `ImportHelper` class defines some attributes which are used to filter the files visible in the file selector.

```
    filename_ext = ".bvh"
    filter_glob = bpy.props.StringProperty(default="*.bvh", options={'HIDDEN'})

    filepath = bpy.props.StringProperty(name="File Path",
        maxlen=1024, default="")
```

There is an analogous `ExportHelper` class which limits the available choices in an export file selector.




```
#----------------------------------------------------------
# File simple_bvh_import.py
# Simple bvh importer
#----------------------------------------------------------

bl_info = {
```

```
    'name': 'Simple BVH importer (.bvh)',
    'author': 'Thomas Larsson',
    'version': (1, 0, 0),
    'blender': (2, 5, 7),
    'api': 34786,
    'location': "File > Import",
    'description': 'Simple import of Biovision bvh',
    'category': 'Import-Export'}

import bpy, os, math, mathutils, time
from mathutils import Vector, Matrix
from bpy_extras.io_utils import ImportHelper

#
#    class CNode:
#

class CNode:
    def __init__(self, words, parent):
        name = words[1]
        for word in words[2:]:
            name += ' '+word

        self.name = name
        self.parent = parent
        self.children = []
        self.head = Vector((0,0,0))
        self.offset = Vector((0,0,0))
        if parent:
            parent.children.append(self)
        self.channels = []
        self.matrix = None
        self.inverse = None
        return

    def __repr__(self):
        return "CNode %s" % (self.name)

    def display(self, pad):
        vec = self.offset
        if vec.length < Epsilon:
            c = '*'
        else:
            c = ' '
        print("%s%s%10s (%8.3f %8.3f %8.3f)" %
            (c, pad, self.name, vec[0], vec[1], vec[2]))
```

```
        for child in self.children:
            child.display(pad+"  ")
        return

    def build(self, amt, orig, parent):
        self.head = orig + self.offset
        if not self.children:
            return self.head

        zero = (self.offset.length < Epsilon)
        eb = amt.edit_bones.new(self.name)
        if parent:
            eb.parent = parent
        eb.head = self.head
        tails = Vector((0,0,0))
        for child in self.children:
            tails += child.build(amt, self.head, eb)
        n = len(self.children)
        eb.tail = tails/n
        (trans,quat,scale) = eb.matrix.decompose()
        self.matrix = quat.to_matrix()
        self.inverse = self.matrix.copy()
        self.inverse.invert()
        if zero:
            return eb.tail
        else:
            return eb.head

#
#    readBvhFile(context, filepath, rot90, scale):
#

Location = 1
Rotation = 2
Hierarchy = 1
Motion = 2
Frames = 3

Deg2Rad = math.pi/180
Epsilon = 1e-5

def readBvhFile(context, filepath, rot90, scale):
    fileName = os.path.realpath(os.path.expanduser(filepath))
    (shortName, ext) = os.path.splitext(fileName)
    if ext.lower() != ".bvh":
        raise NameError("Not a bvh file: " + fileName)
```

```
print( "Loading BVH file "+ fileName )

time1 = time.clock()
level = 0
nErrors = 0
scn = context.scene

fp = open(fileName, "rU")
print( "Reading skeleton" )
lineNo = 0
for line in fp:
    words= line.split()
    lineNo += 1
    if len(words) == 0:
        continue
    key = words[0].upper()
    if key == 'HIERARCHY':
        status = Hierarchy
    elif key == 'MOTION':
        if level != 0:
            raise NameError("Tokenizer out of kilter %d" % level)
        amt = bpy.data.armatures.new("BvhAmt")
        rig = bpy.data.objects.new("BvhRig", amt)
        scn.objects.link(rig)
        scn.objects.active = rig
        bpy.ops.object.mode_set(mode='EDIT')
        root.build(amt, Vector((0,0,0)), None)
        #root.display('')
        bpy.ops.object.mode_set(mode='OBJECT')
        status = Motion
    elif status == Hierarchy:
        if key == 'ROOT':
            node = CNode(words, None)
            root = node
            nodes = [root]
        elif key == 'JOINT':
            node = CNode(words, node)
            nodes.append(node)
        elif key == 'OFFSET':
            (x,y,z) = (float(words[1]), float(words[2]), float(words[3]))
            if rot90:
                node.offset = scale*Vector((x,-z,y))
            else:
                node.offset = scale*Vector((x,y,z))
        elif key == 'END':
            node = CNode(words, node)
```

```
            elif key == 'CHANNELS':
                oldmode = None
                for word in words[2:]:
                    if rot90:
                        (index, mode, sign) = channelZup(word)
                    else:
                        (index, mode, sign) = channelYup(word)
                    if mode != oldmode:
                        indices = []
                        node.channels.append((mode, indices))
                        oldmode = mode
                    indices.append((index, sign))
            elif key == '{':
                level += 1
            elif key == '}':
                level -= 1
                node = node.parent
            else:
                raise NameError("Did not expect %s" % words[0])
        elif status == Motion:
            if key == 'FRAMES:':
                nFrames = int(words[1])
            elif key == 'FRAME' and words[1].upper() == 'TIME:':
                frameTime = float(words[2])
                frameTime = 1
                status = Frames
                frame = 0
                t = 0
                bpy.ops.object.mode_set(mode='POSE')
                pbones = rig.pose.bones
                for pb in pbones:
                    pb.rotation_mode = 'QUATERNION'
        elif status == Frames:
            addFrame(words, frame, nodes, pbones, scale)
            t += frameTime
            frame += 1

    fp.close()
    time2 = time.clock()
    print("Bvh file loaded in %.3f s" % (time2-time1))
    return rig

#
#    channelYup(word):
#    channelZup(word):
#
```

```
def channelYup(word):
    if word == 'Xrotation':
        return ('X', Rotation, +1)
    elif word == 'Yrotation':
        return ('Y', Rotation, +1)
    elif word == 'Zrotation':
        return ('Z', Rotation, +1)
    elif word == 'Xposition':
        return (0, Location, +1)
    elif word == 'Yposition':
        return (1, Location, +1)
    elif word == 'Zposition':
        return (2, Location, +1)

def channelZup(word):
    if word == 'Xrotation':
        return ('X', Rotation, +1)
    elif word == 'Yrotation':
        return ('Z', Rotation, +1)
    elif word == 'Zrotation':
        return ('Y', Rotation, -1)
    elif word == 'Xposition':
        return (0, Location, +1)
    elif word == 'Yposition':
        return (2, Location, +1)
    elif word == 'Zposition':
        return (1, Location, -1)

#
#    addFrame(words, frame, nodes, pbones, scale):
#

def addFrame(words, frame, nodes, pbones, scale):
    m = 0
    for node in nodes:
        name = node.name
        try:
            pb = pbones[name]
        except:
            pb = None
        if pb:
            for (mode, indices) in node.channels:
                if mode == Location:
                    vec = Vector((0,0,0))
                    for (index, sign) in indices:
```

```
                        vec[index] = sign*float(words[m])
                        m += 1
                    pb.location = (scale * vec  - node.head) * node.inverse
                    for n in range(3):
                        pb.keyframe_insert('location', index=n, frame=frame, group=name)
                elif mode == Rotation:
                    mats = []
                    for (axis, sign) in indices:
                        angle = sign*float(words[m])*Deg2Rad
                        mats.append(Matrix.Rotation(angle, 3, axis))
                        m += 1
                    mat = node.inverse * mats[0] * mats[1] * mats[2] * node.matrix
                    pb.rotation_quaternion = mat.to_quaternion()
                    for n in range(4):
                        pb.keyframe_insert('rotation_quaternion',
                                           index=n, frame=frame, group=name)
    return

#
#    initSceneProperties(scn):
#

def initSceneProperties(scn):
    bpy.types.Scene.MyBvhRot90 = bpy.props.BoolProperty(
        name="Rotate 90 degrees",
        description="Rotate the armature to make Z point up")
    scn['MyBvhRot90'] = True

    bpy.types.Scene.MyBvhScale = bpy.props.FloatProperty(
        name="Scale",
        default = 1.0,
        min = 0.01,
        max = 100)
    scn['MyBvhScale'] = 1.0

initSceneProperties(bpy.context.scene)

#
#    class BvhImportPanel(bpy.types.Panel):
#

class BvhImportPanel(bpy.types.Panel):
    bl_label = "BVH import"
    bl_space_type = "VIEW_3D"
    bl_region_type = "UI"
```

```
    def draw(self, context):
        self.layout.prop(context.scene, "MyBvhRot90")
        self.layout.prop(context.scene, "MyBvhScale")
        self.layout.operator("simple_bvh.load")

#
#    class OBJECT_OT_LoadBvhButton(bpy.types.Operator, ImportHelper):
#

class OBJECT_OT_LoadBvhButton(bpy.types.Operator, ImportHelper):
    bl_idname = "simple_bvh.load"
    bl_label = "Load BVH file (.bvh)"

    # From ImportHelper. Filter filenames.
    filename_ext = ".bvh"
    filter_glob = bpy.props.StringProperty(default="*.bvh", options={'HIDDEN'})

    filepath = bpy.props.StringProperty(name="File Path",
        maxlen=1024, default="")

    def execute(self, context):
        import bpy, os
        readBvhFile(context, self.properties.filepath,
            context.scene.MyBvhRot90, context.scene.MyBvhScale)
        return{'FINISHED'}

    def invoke(self, context, event):
        context.window_manager.fileselect_add(self)
        return {'RUNNING_MODAL'}

#
#    Registration
#

def menu_func(self, context):
    self.layout.operator("simple_bvh.load", text="Simple BVH (.bvh)...")

def register():
    bpy.utils.register_module(__name__)
    bpy.types.INFO_MT_file_import.append(menu_func)

def unregister():
    bpy.utils.unregister_module(__name__)
    bpy.types.INFO_MT_file_import.remove(menu_func)

if __name__ == "__main__":
```

```
try:
    unregister()
except:
    pass
register()
```

11 Multi-file packages

Packages are a way of structuring Python's module namespace by using "dotted module names". For example, the module name `A.B` designates a submodule named `B` in a package named `A`. Just like the use of modules saves the authors of different modules from having to worry about each other's global variable names, the use of dotted module names saves the authors of multi-module packages from having to worry about each other's module names. For more information about Python packages, please refer to the Python documentation at `http://docs.python.org/tutorial/modules.html#packages`.

Every package must contain file called `__init__.py`. This file is required to make Python treat the directory as containing a package; this is done to prevent directories with a common name, such as string, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code for the package. In Blender, `__init__.py` often contains the user interface and the add-ons information, whereas the real work is done in other files.

Unlike the other scripts in this book, a multi-file package can not be executed from the text editor. It must be copied to a location which is included in Blender's search path, e.g. the `addons` or `addons-contrib`, cf. section 10. Fortunately, you don't have to restart Blender to reload the files after each modification. Hitting F8 on the keyboard reloads all activated add-ons into Blender.

11.1 A simple example

This package is spread on four files. Three of these create meshes; a cube, a cylinder, and a sphere, respectively. These files are stand-alone scripts which can be executed in the text editor window for debug purposes. The condition `(__name__ == "__main__")` is true when the file is run in stand-alone mode.

11.1.1 mycube.py

```
#-----------------------------------------------------------
# File mycube.py
#-----------------------------------------------------------
import bpy

def makeMesh(z):
    bpy.ops.mesh.primitive_cube_add(location=(0,0,z))
    return bpy.context.object

if __name__ == "__main__":
    ob = makeMesh(1)
    print(ob, "created")
```

11.1.2 mycylinder.py

```
#-----------------------------------------------------------
# File mycylinder.py
#-----------------------------------------------------------
import bpy

def makeMesh(z):
    bpy.ops.mesh.primitive_cylinder_add(location=(0,0,z))
    return bpy.context.object

if __name__ == "__main__":
    ob = makeMesh(5)
    print(ob, "created")
```

11.1.3 mysphere.py

```
#-----------------------------------------------------------
# File mysphere.py
#-----------------------------------------------------------
import bpy

def makeMesh(z):
    bpy.ops.mesh.primitive_ico_sphere_add(location=(0,0,z))
    return bpy.context.object

if __name__ == "__main__":
    ob = makeMesh(3)
    print(ob, "created")
```

11.1.4 __init__.py

The fourth file contains the `bl_info` dictionary and registration needed for add-ons, and the user interface. It also contains the following piece of code to import the other files in the package.

```
# To support reload properly, try to access a package var,
# if it's there, reload everything
if "bpy" in locals():
    import imp
    imp.reload(mycube)
    imp.reload(mysphere)
    imp.reload(mycylinder)
    print("Reloaded multifiles")
else:
    from . import mycube, mysphere, mycylinder
    print("Imported multifiles")
```

This code works as follows.

- If `__init__.py()` is run for the first time, i.e. on starting Blender with the addon enabled in your default.blend, `"bpy" in locals()` is False. The other files in the package are imported, and Imported multifiles is printed in the terminal.
- If `__init__.py()` is run for the first time after starting Blender with the addon disabled in your default.blend, and you click enable addon, `"bpy" in locals()` is False. The other files in the package are imported, and Imported multifiles is printed in the terminal.
- Once the add-on is enabled, anytime you press F8 to reload add-ons, `"bpy" in locals()` is True. The other files in the package are reloaded, and Reloaded multifiles is printed in the terminal.

```
#-----------------------------------------------------------
# File __init__.py
#-----------------------------------------------------------

#    Addon info
bl_info = {
    "name": "Multifile",
    'author': 'Thomas Larsson',
    "location": "View3D > UI panel > Add meshes",
     "category": "3D View"
```

```
    }

# To support reload properly, try to access a package var,
# if it's there, reload everything
if "bpy" in locals():
    import imp
    imp.reload(mycube)
    imp.reload(mysphere)
    imp.reload(mycylinder)
    print("Reloaded multifiles")
else:
    from . import mycube, mysphere, mycylinder
    print("Imported multifiles")

import bpy
from bpy.props import *

#
#   class AddMeshPanel(bpy.types.Panel):
#
class AddMeshPanel(bpy.types.Panel):
    bl_label = "Add meshes"
    bl_space_type = "VIEW_3D"
    bl_region_type = "UI"

    def draw(self, context):
        self.layout.operator("multifile.add",
            text="Add cube").mesh = "cube"
        self.layout.operator("multifile.add",
            text="Add cylinder").mesh = "cylinder"
        self.layout.operator("multifile.add",
            text="Add sphere").mesh = "sphere"

#
#   class OBJECT_OT_AddButton(bpy.types.Operator):
#
class OBJECT_OT_AddButton(bpy.types.Operator):
    bl_idname = "multifile.add"
    bl_label = "Add"
    mesh = bpy.props.StringProperty()

    def execute(self, context):
        if self.mesh == "cube":
            mycube.makeMesh(-8)
        elif self.mesh == "cylinder":
            mycylinder.makeMesh(-5)
```

```
        elif self.mesh == "sphere":
            mysphere.makeMesh(-2)
        return{'FINISHED'}

#
#    Registration
#

def register():
    bpy.utils.register_module(__name__)

def unregister():
    bpy.utils.unregister_module(__name__)

if __name__ == "__main__":
    register()
```

11.2 Simple obj importer and exporter

The obj format is commonly used for exchanging mesh data between different applications. Originally invented for Wavefront Maya, it has become the industry standard. It is a simple ascii format which contain lines of the following form:

- v x y z

  Vertex coordinates are (x, y, z)

- vt u v

  Texture vertex coordinates are (u, v)

- f v_1 v_2 ... v_n

  Face with n corners, at vertex $v_1, v_2, ...v_n$. For meshes without UVs.

- f v_1/vt_1 v_2/vt_2 ... v_n/vt_n

  Face with n corners. The corners are at a vertex $v_1, v_2, ...v_n$ in 3D space and at $vt_1, vt_2, ...vt_n$ in texture space.

More constructs, e.g. for material settings and face groups, exist in full-fledged obj exporter and importers.

There are two things to be aware of. First, most applications (to my knowledge, all except Blender) use a convention were the Y axis points up, whereas Blender uses the Z up convention. Second, Maya start counting vertices from 1, whereas

Blender starts counting from 0. This means that the face corners are really located at $v_1 - 1, v_2 - 1, ...v_n - 1$ in 3D space and at $vt_1 - 1, vt_2 - 1, ...vt_n - 1$ in texture space.

The simple obj exporter-importer is a Python package which consists of three files: two files that does the actual export/import work, and `__init__.py` which makes the directory into a package.

11.2.1 Simple obj exporter

This script exports the selected mesh as an obj file.

```
#----------------------------------------------------------
# File export_simple_obj.py
# Simple obj exporter which writes only verts, faces, and texture verts
#----------------------------------------------------------
import bpy, os

def export_simple_obj(filepath, ob, rot90, scale):
    name = os.path.basename(filepath)
    realpath = os.path.realpath(os.path.expanduser(filepath))
    fp = open(realpath, 'w')
    print('Exporting %s' % realpath)

    if not ob or ob.type != 'MESH':
        raise NameError('Cannot export: active object %s is not a mesh.' % ob)
    me = ob.data

    for v in me.vertices:
        x = scale*v.co
        if rot90:
            fp.write("v %.5f %.5f %.5f\n" % (x[0], x[2], -x[1]))
        else:
            fp.write("v %.5f %.5f %.5f\n" % (x[0], x[1], x[2]))

    if len(me.uv_textures) > 0:
        uvtex = me.uv_textures[0]
        for f in me.faces:
            data = uvtex.data[f.index]
            fp.write("vt %.5f %.5f\n" % (data.uv1[0], data.uv1[1]))
            fp.write("vt %.5f %.5f\n" % (data.uv2[0], data.uv2[1]))
            fp.write("vt %.5f %.5f\n" % (data.uv3[0], data.uv3[1]))
            if len(f.vertices) == 4:
                fp.write("vt %.5f %.5f\n" % (data.uv4[0], data.uv4[1]))
```

```
        vt = 1
        for f in me.faces:
            vs = f.vertices
            fp.write("f %d/%d %d/%d %d/%d" % (vs[0]+1, vt, vs[1]+1, vt+1, vs[2]+1, vt+2))
            vt += 3
            if len(f.vertices) == 4:
                fp.write(" %d/%d\n" % (vs[3]+1, vt))
                vt += 1
            else:
                fp.write("\n")
    else:
        for f in me.faces:
            vs = f.vertices
            fp.write("f %d %d %d" % (vs[0]+1, vs[1]+1, vs[2]+1))
            if len(f.vertices) == 4:
                fp.write(" %d\n" % (vs[3]+1))
            else:
                fp.write("\n")

    print('%s successfully exported' % realpath)
    fp.close()
    return
```

11.2.2 Simple obj import

This script is the import companion of the previous one. It can of course also be used to import obj files from other applications.

```
#----------------------------------------------------------
# File import_simple_obj.py
# Simple obj importer which reads only verts, faces, and texture verts
#----------------------------------------------------------
import bpy, os

def import_simple_obj(filepath, rot90, scale):
    name = os.path.basename(filepath)
    realpath = os.path.realpath(os.path.expanduser(filepath))
    fp = open(realpath, 'rU')    # Universal read
    print('Importing %s' % realpath)

    verts = []
    faces = []
    texverts = []
```

```
texfaces = []

for line in fp:
    words = line.split()
    if len(words) == 0:
        pass
    elif words[0] == 'v':
        (x,y,z) = (float(words[1]), float(words[2]), float(words[3]))
        if rot90:
            verts.append( (scale*x, -scale*z, scale*y) )
        else:
            verts.append( (scale*x, scale*y, scale*z) )
    elif words[0] == 'vt':
        texverts.append( (float(words[1]), float(words[2])) )
    elif words[0] == 'f':
        (f,tf) = parseFace(words)
        faces.append(f)
        if tf:
            texfaces.append(tf)
    else:
        pass
print('%s successfully imported' % realpath)
fp.close()

me = bpy.data.meshes.new(name)
me.from_pydata(verts, [], faces)
me.update()

if texverts:
    uvtex = me.uv_textures.new()
    uvtex.name = name
    data = uvtex.data
    for n in range(len(texfaces)):
        tf = texfaces[n]
        data[n].uv1 = texverts[tf[0]]
        data[n].uv2 = texverts[tf[1]]
        data[n].uv3 = texverts[tf[2]]
        if len(tf) == 4:
            data[n].uv4 = texverts[tf[3]]

scn = bpy.context.scene
ob = bpy.data.objects.new(name, me)
scn.objects.link(ob)
scn.objects.active = ob

return
```

```
def parseFace(words):
    face = []
    texface = []
    for n in range(1, len(words)):
        li = words[n].split('/')
        face.append( int(li[0])-1 )
        try:
            texface.append( int(li[1])-1 )
        except:
            pass
    return (face, texface)
```

11.2.3 __init__.py

This file contains the user interface, i.e. the two classes that create menu items for the exporter and importer. The simple exporter is invoked from the File > Export menu. There are two options: a boolean choice to rotate the mesh 90 degrees (to convert between Y up and Z up), and a scale. The simple importer is invoked from the File > Import menu. There are two options: a boolean choice to rotate the mesh 90 degrees (to make Z up), and a scale.

`__init__.py` also contains the `bl_info` dictionary which makes the package into a Blender add-on, the registration code, and the code to import/reload the two other files.

```
#------------------------------------------------------------
# File __init__.py
#------------------------------------------------------------

#    Addon info
bl_info = {
    "name": "Simple OBJ format",
    "author": "Thomas Larsson",
    "location": "File > Import-Export",
    "description": "Simple Wavefront obj import/export. Does meshes and UV coordinates",
    "category": "Import-Export"}

# To support reload properly, try to access a package var,
# if it's there, reload everything
if "bpy" in locals():
    import imp
    if 'simple_obj_import' in locals():
        imp.reload(simple_obj_import)
```

```
    if 'simple_obj_export' in locals():
        imp.reload(simple_obj_export)

import bpy
from bpy.props import *
from bpy_extras.io_utils import ExportHelper, ImportHelper

#
#    Import menu
#

class IMPORT_OT_simple_obj(bpy.types.Operator, ImportHelper):
    bl_idname = "io_import_scene.simple_obj"
    bl_description = 'Import from simple OBJ file format (.obj)'
    bl_label = "Import simple OBJ"
    bl_space_type = "PROPERTIES"
    bl_region_type = "WINDOW"

    filename_ext = ".obj"
    filter_glob = StringProperty(default="*.obj;*.mtl", options={'HIDDEN'})

    filepath = bpy.props.StringProperty(
        name="File Path",
        description="File path used for importing the simple OBJ file",
        maxlen= 1024, default= "")

    rot90 = bpy.props.BoolProperty(
        name = "Rotate 90 degrees",
        description="Rotate mesh to Z up",
        default = True)

    scale = bpy.props.FloatProperty(
        name = "Scale",
        description="Scale mesh",
        default = 0.1, min = 0.001, max = 1000.0)

    def execute(self, context):
        from . import simple_obj_import
        print("Load", self.properties.filepath)
        simple_obj_import.import_simple_obj(
            self.properties.filepath,
            self.rot90,
            self.scale)
        return {'FINISHED'}

    def invoke(self, context, event):
```

```
        context.window_manager.fileselect_add(self)
        return {'RUNNING_MODAL'}

#
#    Export menu
#

class EXPORT_OT_simple_obj(bpy.types.Operator, ExportHelper):
    bl_idname = "io_export_scene.simple_obj"
    bl_description = 'Export from simple OBJ file format (.obj)'
    bl_label = "Export simple OBJ"
    bl_space_type = "PROPERTIES"
    bl_region_type = "WINDOW"

    # From ExportHelper. Filter filenames.
    filename_ext = ".obj"
    filter_glob = StringProperty(default="*.obj", options={'HIDDEN'})

    filepath = bpy.props.StringProperty(
        name="File Path",
        description="File path used for exporting the simple OBJ file",
        maxlen= 1024, default= "")

    rot90 = bpy.props.BoolProperty(
        name = "Rotate 90 degrees",
        description="Rotate mesh to Y up",
        default = True)

    scale = bpy.props.FloatProperty(
        name = "Scale",
        description="Scale mesh",
        default = 0.1, min = 0.001, max = 1000.0)

    def execute(self, context):
        print("Load", self.properties.filepath)
        from . import simple_obj_export
        simple_obj_export.export_simple_obj(
            self.properties.filepath,
            context.object,
            self.rot90,
            1.0/self.scale)
        return {'FINISHED'}

    def invoke(self, context, event):
        context.window_manager.fileselect_add(self)
        return {'RUNNING_MODAL'}
```

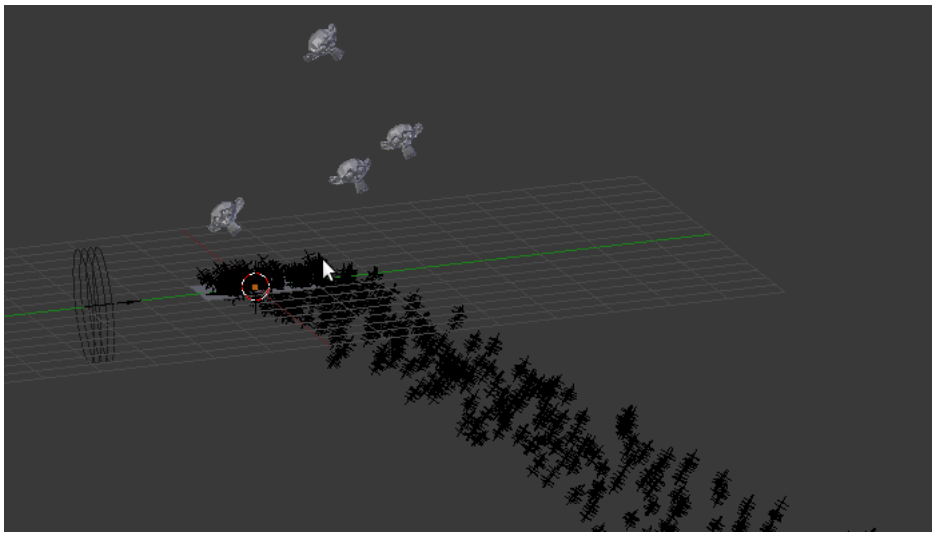
```
#
#    Registration
#

def menu_func_import(self, context):
    self.layout.operator(IMPORT_OT_simple_obj.bl_idname, text="Simple OBJ (.obj)...")

def menu_func_export(self, context):
    self.layout.operator(EXPORT_OT_simple_obj.bl_idname, text="Simple OBJ (.obj)...")

def register():
    bpy.utils.register_module(__name__)
    bpy.types.INFO_MT_file_import.append(menu_func_import)
    bpy.types.INFO_MT_file_export.append(menu_func_export)

def unregister():
    bpy.utils.unregister_module(__name__)
    bpy.types.INFO_MT_file_import.remove(menu_func_import)
    bpy.types.INFO_MT_file_export.remove(menu_func_export)

if __name__ == "__main__":
    register()
```

12 Simulations

In this section we access Blender's simulation capacity from Python. Several of the examples were inspired by the book Bounce, Tumble and Splash by Tony Mullen. Hoewver, most renders do not look as pretty as they do in Mullen's book, since the purpose of these note is not to find the optimal way to tweak parameters, but rather to illustrate how the can be tweaked from Python.

12.1 Particles

This program adds two particle systems



```
#----------------------------------------------------
# File particle.py
#----------------------------------------------------
import bpy, mathutils, math
from mathutils import Vector, Matrix
from math import pi

def run(origo):
    # Add emitter mesh
    origin = Vector(origo)
    bpy.ops.mesh.primitive_plane_add(location=origin)
    emitter = bpy.context.object

    # --- Particle system 1: Falling and blowing drops ---

    # Add first particle system
    bpy.ops.object.particle_system_add()
    psys1 = emitter.particle_systems[-1]
    psys1.name = 'Drops'

    # Emission
    pset1 = psys1.settings
    pset1.name = 'DropSettings'
    pset1.frame_start = 40
    pset1.frame_end = 200
    pset1.lifetime = 50
    pset1.lifetime_random = 0.4
    pset1.emit_from = 'FACE'
```

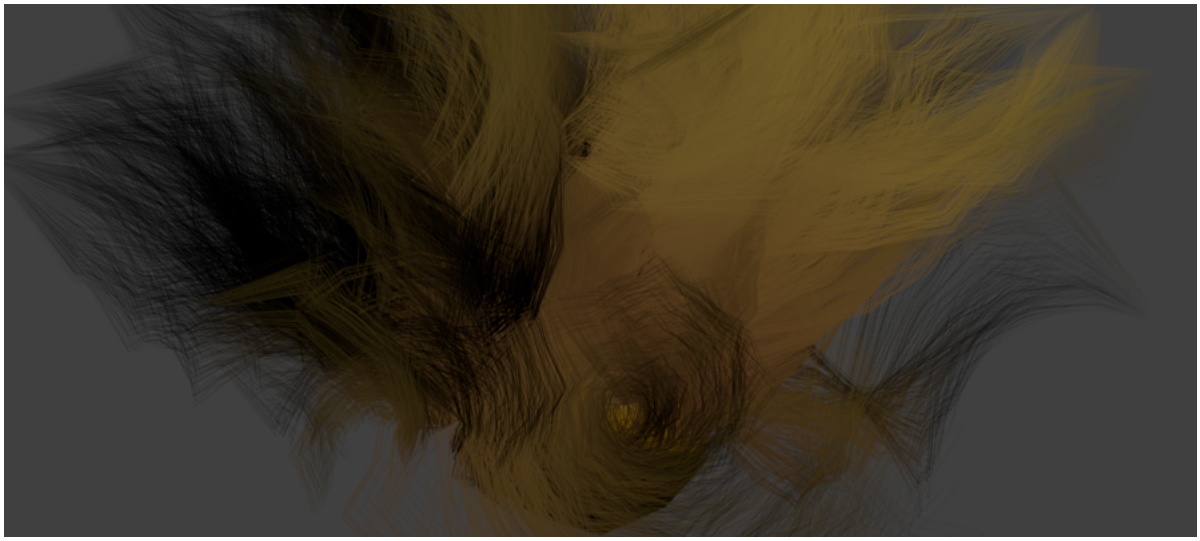
```
pset1.use_render_emitter = True
pset1.object_align_factor = (0,0,1)

# Physics
pset1.physics_type = 'NEWTON'
pset1.mass = 2.5
pset1.particle_size = 0.3
pset1.use_multiply_size_mass = True

# Effector weights
ew = pset1.effector_weights
ew.gravity = 1.0
ew.wind = 1.0

# Children
pset1.child_nbr = 10
pset1.rendered_child_count = 10
pset1.child_type = 'SIMPLE'

# Display and render
pset1.draw_percentage = 100
pset1.draw_method = 'CROSS'
pset1.material = 1
pset1.particle_size = 0.1
pset1.render_type = 'HALO'
pset1.render_step = 3

# ------------ Wind effector -----

# Add wind effector
bpy.ops.object.effector_add(
    type='WIND',
    enter_editmode=False,
    location = origin - Vector((0,3,0)),
    rotation = (-pi/2, 0, 0))
wind = bpy.context.object

# Field settings
fld = wind.field
fld.strength = 2.3
fld.noise = 3.2
fld.flow = 0.3

# --- Particle system 2: Monkeys in the wind ----

# Add monkey to be used as dupli object
```

```
# Hide the monkey on layer 2
layers = 20*[False]
layers[1] = True
bpy.ops.mesh.primitive_monkey_add(
    location=origin+Vector((0,5,0)),
    rotation = (pi/2, 0, 0),
    layers = layers)
monkey = bpy.context.object

#Add second particle system
bpy.context.scene.objects.active = emitter
bpy.ops.object.particle_system_add()
psys2 = emitter.particle_systems[-1]
psys2.name = 'Monkeys'
pset2 = psys2.settings
pset2.name = 'MonkeySettings'

# Emission
pset2.count = 4
pset2.frame_start = 1
pset2.frame_end = 50
pset2.lifetime = 250
pset2.emit_from = 'FACE'
pset2.use_render_emitter = True

# Velocity
pset2.factor_random = 0.5

# Physics
pset2.physics_type = 'NEWTON'
pset2.brownian_factor = 0.5

# Effector weights
ew = pset2.effector_weights
ew.gravity = 0
ew.wind = 0.2

# Children
pset2.child_nbr = 1
pset2.rendered_child_count = 1
pset2.child_size = 3
pset2.child_type = 'SIMPLE'

# Display and render
pset2.draw_percentage = 1
pset2.draw_method = 'RENDER'
```

```
    pset2.dupli_object = monkey
    pset2.material = 1
    pset2.particle_size = 0.1
    pset2.render_type = 'OBJECT'
    pset2.render_step = 3

    return

if __name__ == "__main__":
    bpy.ops.object.select_by_type(type='MESH')
    bpy.ops.object.delete()
    run((0,0,0))
    bpy.ops.screen.animation_play(reverse=False, sync=False)
```

12.2 Hair

This program adds a sphere with hair. A strand shader is constructed for the hair.



```
#----------------------------------------------
# File hair.py
#----------------------------------------------
import bpy

def createHead(origin):
```

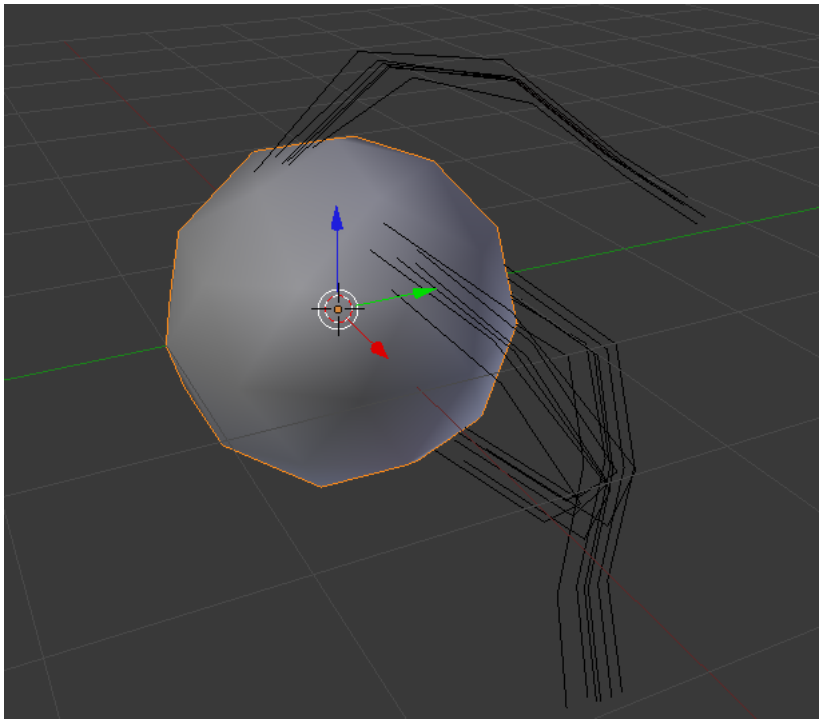
```
    # Add emitter mesh
    bpy.ops.mesh.primitive_ico_sphere_add(location=origin)
    ob = bpy.context.object
    bpy.ops.object.shade_smooth()

    # Create scalp vertex group, and add verts and weights
    scalp = ob.vertex_groups.new('Scalp')
    for v in ob.data.vertices:
        z = v.co[2]
        y = v.co[1]
        if z > 0.3 or y > 0.3:
            w = 2*(z-0.3)
            if w > 1:
                w = 1
            scalp.add([v.index], w, 'REPLACE')
    return ob

def createMaterials(ob):
    # Some material for the skin
    skinmat = bpy.data.materials.new('Skin')
    skinmat.diffuse_color = (0.6,0.3,0)

    # Strand material for hair
    hairmat = bpy.data.materials.new('Strand')
    hairmat.diffuse_color = (0.2,0.04,0.0)
    hairmat.specular_intensity = 0

    # Transparency
    hairmat.use_transparency = True
    hairmat.transparency_method = 'Z_TRANSPARENCY'
    hairmat.alpha = 0

    # Strand. Must use Blender units before sizes are pset.
    strand = hairmat.strand
    strand.use_blender_units = True
    strand.root_size = 0.01
    strand.tip_size = 0.0025
    strand.size_min = 0.001
    #strand.use_surface_diffuse = True # read-only
    strand.use_tangent_shading = True

    # Texture
    tex = bpy.data.textures.new('Blend', type = 'BLEND')
    tex.progression = 'LINEAR'
    tex.use_flip_axis = 'HORIZONTAL'
```

```
    # Create a color ramp for color and alpha
    tex.use_color_ramp = True
    tex.color_ramp.interpolation = 'B_SPLINE'
    # Points in color ramp: (pos, rgba)
    # Have not figured out how to add points to ramp
    rampTable = [
        (0.0, (0.23,0.07,0.03,0.75)),
        #(0.2, (0.4,0.4,0,0.5)),
        #(0.7, (0.6,0.6,0,0.5)),
        (1.0, (0.4,0.3,0.05,0))
    ]
    elts = tex.color_ramp.elements
    n = 0
    for (pos, rgba) in rampTable:
        elts[n].position = pos
        elts[n].color = rgba
        n += 1

    # Add blend texture to hairmat
    mtex = hairmat.texture_slots.add()
    mtex.texture = tex
    mtex.texture_coords = 'STRAND'
    mtex.use_map_color_diffuse = True
    mtex.use_map_alpha = True

    # Add materials to mesh
    ob.data.materials.append(skinmat)    # Material 1 = Skin
    ob.data.materials.append(hairmat)    # Material 2 = Strand
    return

def createHair(ob):
    # Create hair particle system
    bpy.ops.object.particle_system_add()
    psys = ob.particle_systems.active
    psys.name = 'Hair'
    # psys.global_hair = True
    psys.vertex_group_density = 'Scalp'

    pset = psys.settings
    pset.type = 'HAIR'
    pset.name = 'HairSettings'

    # Emission
    pset.count = 40
    pset.hair_step = 7
    pset.emit_from = 'FACE'
```

```
    # Render
    pset.material = 2
    pset.use_render_emitter = True
    pset.render_type = 'PATH'
    pset.use_strand_primitive = True
    pset.use_hair_bspline = True

    # Children
    pset.child_type = 'SIMPLE'
    pset.child_nbr = 10
    pset.rendered_child_count = 500
    pset.child_length = 1.0
    pset.child_length_threshold = 0.0

    pset.child_roundness = 0.4
    pset.clump_factor = 0.862
    pset.clump_shape = 0.999

    pset.roughness_endpoint = 0.0
    pset.roughness_end_shape = 1.0
    pset.roughness_1 = 0.0
    pset.roughness_1_size = 1.0
    pset.roughness_2 = 0.0
    pset.roughness_2_size = 1.0
    pset.roughness_2_threshold = 0.0

    pset.kink = 'CURL'
    pset.kink_amplitude = 0.2
    pset.kink_shape = 0.0
    pset.kink_frequency = 2.0

    return

def run(origin):
    ob = createHead(origin)
    createMaterials(ob)
    createHair(ob)
    return

if __name__ == "__main__":
    bpy.ops.object.select_by_type(type='MESH')
    bpy.ops.object.delete()
    run((0,0,0))
```

12.3 Editable hair

This program adds a sphere with editable hair from given hair guides. If we toggle to edit mode, all strands become straight, i.e. the editing is lost. This is prevented if you toggle to particle mode, select the object, and toggle back into object mode. Unfortunately, I have not found a way to do this with a script.



```
#----------------------------------------------------
# File edit_hair.py
# Has flaws, but may be of interest anyway.
#----------------------------------------------------
import bpy

def createHead():
    # Add emitter mesh
    bpy.ops.mesh.primitive_ico_sphere_add()
    ob = bpy.context.object
    ob.name = 'EditedHair'
    bpy.ops.object.shade_smooth()
    return ob

def createHair(ob, guides):
```

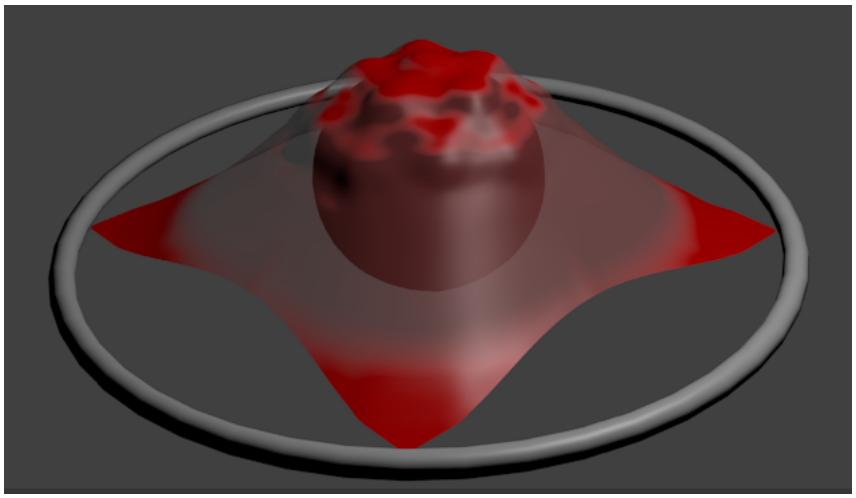
```
nGuides = len(guides)
nSteps = len(guides[0])

# Create hair particle system
bpy.ops.object.mode_set(mode='OBJECT')
bpy.ops.object.particle_system_add()
psys = ob.particle_systems.active
psys.name = 'Hair'

# Particle settings
pset = psys.settings
pset.type = 'HAIR'
pset.name = 'HairSettings'
pset.count = nGuides
pset.hair_step = nSteps-1
pset.emit_from = 'FACE'
pset.use_render_emitter = True

# Children
pset.child_type = 'SIMPLE'
pset.child_nbr = 6
pset.rendered_child_count = 300
pset.child_length = 1.0
pset.child_length_threshold = 0.0

# Disconnect hair and switch to particle edit mode
bpy.ops.particle.disconnect_hair(all=True)
bpy.ops.particle.particle_edit_toggle()

# Set all hair-keys
dt = 100.0/(nSteps-1)
dw = 1.0/(nSteps-1)
for m in range(nGuides):
    guide = guides[m]
    part = psys.particles[m]
    part.location = guide[0]
    for n in range(1, nSteps):
        point = guide[n]
        h = part.hair_keys[n-1]
        h.co_hair_space = point
        h.time = n*dt
        h.weight = 1.0 - n*dw

# Toggle particle edit mode
bpy.ops.particle.select_all(action='SELECT')
bpy.ops.particle.particle_edit_toggle()
```

```
    # Connect hair to mesh
    # Segmentation violation during render if this line is absent.
    bpy.ops.particle.connect_hair(all=True)

    # Unfortunately, here a manual step appears to be needed:
    # 1. Toggle to particle mode
    # 2. Touch object with a brush
    # 3. Toggle to object mode
    # 4. Toggle to edit mode
    # 5. Toggle to object mode
    # This should correspond to the code below, but fails due to
    # wrong context
    '''
    bpy.ops.particle.particle_edit_toggle()
    bpy.ops.particle.brush_edit()
    bpy.ops.particle.particle_edit_toggle()
    bpy.ops.object.editmode_toggle()
    bpy.ops.object.editmode_toggle()
    '''
    return

# Hair guides. Four hair with five points each.
hairGuides = [
    [(-0.334596,0.863821,0.368362),
     (-0.351643,2.33203,-0.24479),
     (0.0811583,2.76695,-0.758137),
     (0.244019,2.73683,-1.5408),
     (0.199297,2.60424,-2.32847)],
    [(0.646501,0.361173,0.662151),
     (1.33538,-0.15509,1.17099),
     (2.07275,0.296789,0.668891),
     (2.55172,0.767097,-0.0723231),
     (2.75942,1.5089,-0.709962)],
    [(-0.892345,-0.0182112,0.438324),
     (-1.5723,0.484807,0.971839),
     (-2.2393,0.116525,0.324168),
     (-2.18426,-0.00867975,-0.666435),
     (-1.99681,-0.0600535,-1.64737)],
    [(-0.0154996,0.0387489,0.995887),
     (-0.205679,-0.528201,1.79738),
     (-0.191354,0.36126,2.25417),
     (0.0876127,1.1781,1.74925),
     (0.300626,1.48545,0.821801)]
    ]
```

```
def run(origin):
    ob = createHead()
    createHair(ob, hairGuides)
    ob.location = origin
    return

if __name__ == "__main__":
    run((0,0,0))
```

12.4 Cloth

This program adds a plane with a cloth modifier. The plane is parented to a hoop which moves downward, where it meets a sphere obstacle. The influence of the cloth modifier is controlled by a vertex group, which means that the corners move with the hoop while the middle is deformed by the obstacle. The plane is given a material with stress-mapped transparency.



```
#----------------------------------------------------------
# File cloth.py
#----------------------------------------------------------
import bpy, mathutils, math
from mathutils import Vector

def run(origin):
    side = 4
    diagonal = side/math.sqrt(2)
```

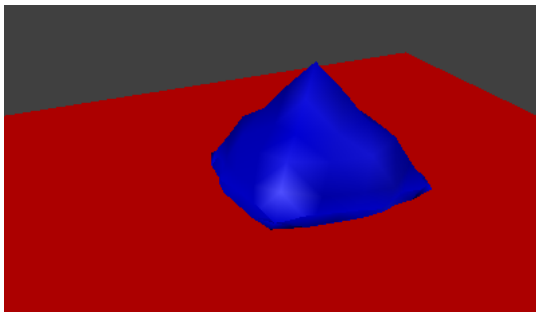
```
hoopRad = 0.1
eps = 0.75
nDivs = 40

scn = bpy.context.scene

# Add a sphere acting as a collision object
bpy.ops.mesh.primitive_ico_sphere_add(location=origin)
sphere = bpy.context.object
bpy.ops.object.shade_smooth()

# Add a collision modifier to the sphere
bpy.ops.object.modifier_add(type='COLLISION')
cset = sphere.modifiers[0].settings
cset.thickness_outer = 0.2
cset.thickness_inner = 0.5
cset.permeability = 0.2
cset.stickness = 0.2
bpy.ops.object.modifier_add(type='SUBSURF')

# Add ring
center = origin+Vector((0,0,2))
bpy.ops.mesh.primitive_torus_add(
    major_radius= diagonal + hoopRad,
    minor_radius= hoopRad,
    location=center,
    rotation=(0, 0, 0))
bpy.ops.object.shade_smooth()
ring = bpy.context.object

# Add a plane over the sphere and parent it to ring
bpy.ops.mesh.primitive_plane_add(location=(0,0,0))
bpy.ops.transform.resize(value=(side/2,side/2,1))
bpy.ops.object.mode_set(mode='EDIT')
bpy.ops.mesh.subdivide(number_cuts=nDivs)
bpy.ops.object.mode_set(mode='OBJECT')
plane = bpy.context.object
plane.parent = ring
me = plane.data

# Create vertex group. Object must not be active?
scn.objects.active = None
grp = plane.vertex_groups.new('Group')
for v in plane.data.vertices:
    r = v.co - center
    x = r.length/diagonal
```

```
    w = 3*(x-eps)/(1-eps)
    if w > 1:
        w = 1
    if w > 0:
       grp.add([v.index], w, 'REPLACE')

# Reactivate plane
scn.objects.active = plane

# Add cloth modifier
cloth = plane.modifiers.new(name='Cloth', type='CLOTH')
cset = cloth.settings
cset.use_pin_cloth = True
cset.vertex_group_mass = 'Group'
# Silk presets, copied from "scripts/presets/cloth/silk.py"
cset.quality = 5
cset.mass = 0.150
cset.structural_stiffness = 5
cset.bending_stiffness = 0.05
cset.spring_damping = 0
cset.air_damping = 1

# Smooth shading
plane.select = True
bpy.ops.object.shade_smooth()
bpy.ops.object.modifier_add(type='SUBSURF')

# Blend texture
tex = bpy.data.textures.new('Blend', type = 'BLEND')
tex.progression = 'SPHERICAL'
tex.intensity = 1.0
tex.contrast = 1.0
tex.use_color_ramp = True
elts = tex.color_ramp.elements
elts[0].color = (0, 0, 0, 1)
elts[0].position = 0.56
elts[1].color = (1, 1, 1, 0)
elts[1].position = 0.63

# Rubber material
mat = bpy.data.materials.new('Rubber')
mat.diffuse_color = (1,0,0)
mat.use_transparency = True
mat.alpha = 0.25

mtex = mat.texture_slots.add()
```

```
    mtex.texture = tex
    mtex.texture_coords = 'STRESS'
    mtex.use_map_color_diffuse = True
    mtex.diffuse_color_factor = 0.25
    mtex.use_map_alpha = True
    mtex.alpha_factor = 1.0
    mtex.blend_type = 'ADD'

    # Add material to plane
    plane.data.materials.append(mat)

    # Animate ring
    ring.location = center
    ring.keyframe_insert('location', index=2, frame=1)
    ring.location = origin - Vector((0,0,0.5))
    ring.keyframe_insert('location', index=2, frame=20)
    ring.location = center

    return

if __name__ == "__main__":
    bpy.ops.object.select_by_type(type='MESH')
    bpy.ops.object.delete()
    run(Vector((0,0,0)))
    scn = bpy.context.scene
    scn.frame_current = 1
    bpy.ops.screen.animation_play()
```

12.5 Softbodies

This program adds a sphere with a softbody modifier and a plane obstacle.



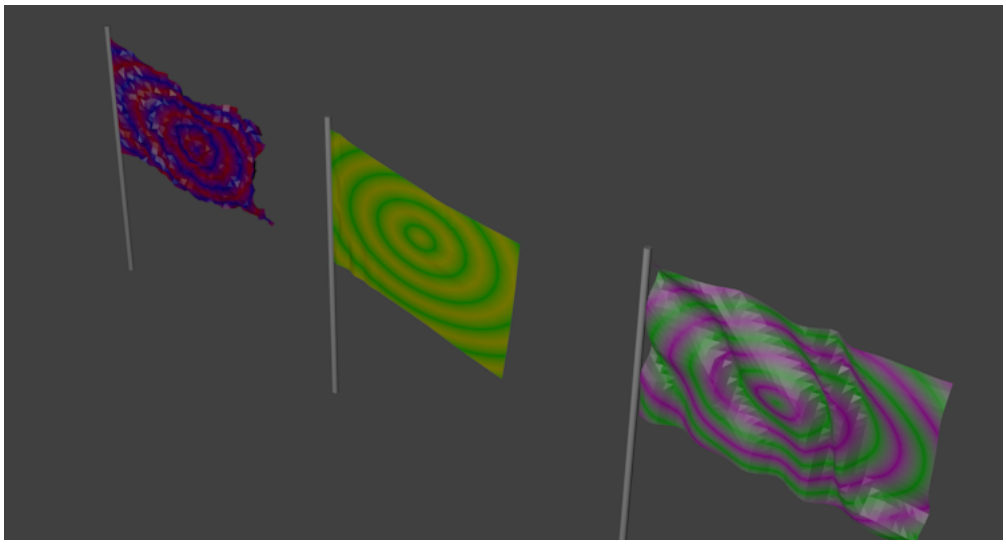
```
#----------------------------------------------------------
# File softbody.py
#----------------------------------------------------------
import bpy
import mathutils
from mathutils import Vector

def run(origin):
    # Add materials
    red = bpy.data.materials.new('Red')
    red.diffuse_color = (1,0,0)
    blue = bpy.data.materials.new('Blue')
    blue.diffuse_color = (0,0,1)

    # Add a cone
    bpy.ops.mesh.primitive_cone_add(
        vertices=4,
        radius=1.5,
        cap_end=True)
    ob1 = bpy.context.object
    me1 = ob1.data
    bpy.ops.object.mode_set(mode='EDIT')
    bpy.ops.mesh.subdivide(number_cuts=5, smoothness=1, fractal=1)
    bpy.ops.object.mode_set(mode='OBJECT')

    # Weirdly, need new mesh which is a copy of
    verts = []
    faces = []
    for v in me1.vertices:
        verts.append(v.co)
    for f in me1.faces:
        faces.append(f.vertices)
    me2 = bpy.data.meshes.new('Drop')
    me2.from_pydata(verts, [], faces)
    me2.update(calc_edges=True)

    # Set faces smooth
    for f in me2.faces:
        f.use_smooth = True

    # Add new object and make it active
    ob2 = bpy.data.objects.new('Drop', me2)
    scn = bpy.context.scene
    scn.objects.link(ob2)
    scn.objects.unlink(ob1)
    scn.objects.active = ob2
```

```
# Add vertex groups
top = ob2.vertex_groups.new('Top')
bottom = ob2.vertex_groups.new('Bottom')
for v in me2.vertices:
    w = v.co[2] - 0.2
    if w < 0:
        if w < -1:
            w = -1
        bottom.add([v.index], -w, 'REPLACE')
    elif w > 0:
        if w > 1:
            w = 1
        top.add([v.index], w, 'REPLACE')
bpy.ops.object.mode_set(mode='OBJECT')
ob2.location = origin
me2.materials.append(blue)

# Add a softbody modifier
mod = ob2.modifiers.new(name='SoftBody', type='SOFT_BODY')
sbset = mod.settings

# Soft body
sbset.friction = 0.6
sbset.speed = 0.4
sbset.mass = 8.1

# Goal
sbset.goal_default = 0.7
sbset.goal_spring = 0.3
sbset.goal_friction = 0.0
sbset.vertex_group_goal = 'Top'

# Soft body edges
sbset.pull = 0.6
sbset.push = 0.1
sbset.bend = 0.1
sbset.aerodynamics_type = 'LIFT_FORCE'
sbset.aero = 0.5

# Add a vortex
bpy.ops.object.effector_add(
    type='VORTEX',
    location=origin+Vector((0,0,-4)))
vortex = bpy.context.object
fset = vortex.field
```

```
    fset.strength = 4.5
    fset.shape = 'PLANE'
    fset.apply_to_location = False
    fset.apply_to_rotation = True
    fset.falloff_type = 'TUBE'

    # Add collision plane
    # Warning. Collision objects make simulation very slow!
    bpy.ops.mesh.primitive_plane_add(
        location=origin-Vector((0,0,1.7)))
    bpy.ops.transform.resize(value=(4, 4, 4))
    plane = bpy.context.object
    plane.data.materials.append(red)
    mod = plane.modifiers.new(name='Collision', type='COLLISION')

    return

if __name__ == "__main__":
    bpy.context.scene.frame_end = 600
    bpy.ops.object.select_all(action='SELECT')
    bpy.ops.object.delete()
    run(Vector((0,0,6)))
    bpy.ops.screen.animation_play()
    #bpy.ops.render.opengl(animation=True)
```

12.6 Cloth, softbodies and displacement textures

This program illustrates three different methods to make a waving flag: with a cloth modifier, with a softbody modifier, and with animated displacement textures.



```
#-----------------------------------------------------------
# File flags.py
# Creates a softbody flag and a cloth flag in the wind.
#-----------------------------------------------------------
import bpy, mathutils, math
from mathutils import Vector
from math import pi

# Flag size, global variables
xmax = 40
zmax = 24
ds = 2.0/xmax

def makeFlag(name, origin, invert):
    # Add new mesh that will be the flag

    me = bpy.data.meshes.new(name)
    flag = bpy.data.objects.new(name, me)
    scn = bpy.context.scene
    scn.objects.link(flag)
    scn.objects.active = flag

    # Build flag mesh
    verts = []
    faces = []
    for x in range(xmax):
        for z in range(zmax):
            verts.append(((x+0.5)*ds, 0, z*ds))
```

```
            if x > 0 and z > 0:
                faces.append(((x-1)*zmax+(z-1), (x-1)*zmax+z, x*zmax+z, x*zmax+(z-1)))
    me.from_pydata(verts, [], faces)
    me.update(calc_edges=True)
    flag.location = origin

    # Add vertex groups
    grp = flag.vertex_groups.new('Pole')
    for v in me.vertices:
        w = 1.5 - 7*v.co[0]
        if invert:
            if w > 1:
                grp.add([v.index], 0.0, 'REPLACE')
            else:
                grp.add([v.index], 1-w, 'REPLACE')
        else:
            if w > 1:
                grp.add([v.index], 1.0, 'REPLACE')
            elif w > 0:
                grp.add([v.index], w, 'REPLACE')

    bpy.ops.object.mode_set(mode='OBJECT')
    bpy.ops.object.shade_smooth()
    return flag

def makePole(origin):
    bpy.ops.mesh.primitive_cylinder_add(
        vertices=32,
        radius=ds/2,
        depth=1,
        cap_ends=True)
    bpy.ops.transform.resize(value=(1, 1, 2.5))
    pole = bpy.context.object
    pole.location = origin
    return pole

def addSoftBodyModifier(ob):
    mod = ob.modifiers.new(name='SoftBody', type='SOFT_BODY')
    sbset = mod.settings

    # Soft body
    sbset.friction = 0.3
    sbset.speed = 1.4
    sbset.mass = 0.9

    # Goal
```

```
    sbset.goal_default = 0.3
    sbset.goal_spring = 0.1
    sbset.goal_friction = 0.1
    sbset.vertex_group_goal = 'Pole'

    # Soft body edges
    sbset.pull = 0.1
    sbset.push = 0.1
    sbset.bend = 0.1
    sbset.aerodynamics_type = 'LIFT_FORCE'
    sbset.aero = 0.5

    #Effector weights
    ew = sbset.effector_weights
    ew.gravity = 0.1
    ew.wind = 0.8
    return

def addClothModifier(ob):
    cloth = ob.modifiers.new(name='Cloth', type='CLOTH')
    cset = cloth.settings

    cset.quality = 4
    cset.mass = 0.2
    cset.structural_stiffness = 0.5
    cset.bending_stiffness = 0.05
    cset.spring_damping = 0
    cset.air_damping = 0.3

    cset.use_pin_cloth = True
    cset.vertex_group_mass = 'Pole'

    #Effector weights
    ew = cset.effector_weights
    ew.gravity = 0.1
    ew.wind = 1.0
    return


def addWindEffector(origin):
    # Add wind effector
    bpy.ops.object.effector_add(
        type='WIND',
        location=origin,
        rotation=(pi/2,0,0))
    wind = bpy.context.object
```

```
    fset = wind.field

    fset.strength = -2.0
    fset.noise = 10.0
    fset.flow = 0.8
    fset.shape = 'PLANE'
    return

def addFlagMaterial(name, ob, color1, color2):
    # Flag texture
    tex = bpy.data.textures.new('Flag', type = 'WOOD')
    tex.noise_basis_2 = 'TRI'
    tex.wood_type = 'RINGS'

    # Create material
    mat = bpy.data.materials.new(name)
    mat.diffuse_color = color1

    # Add texture slot for color texture
    mtex = mat.texture_slots.add()
    mtex.texture = tex
    mtex.texture_coords = 'ORCO'
    mtex.use_map_color_diffuse = True
    mtex.color = color2

    # Add material to flags
    ob.data.materials.append(mat)
    return mat

def createDisplacementTexture(mat):
    tex = bpy.data.textures.new('Flag', type = 'WOOD')
    tex.noise_basis_2 = 'SIN'
    tex.wood_type = 'BANDNOISE'
    tex.noise_type = 'SOFT_NOISE'
    tex.noise_scale = 0.576
    tex.turbulence = 9.0
    # Store texture in material for easy access. Not really necessary
    mtex = mat.texture_slots.add()
    mtex.texture = tex
    mat.use_textures[1] = False
    return tex

def addDisplacementModifier(ob, tex, vgrp, empty):
    mod = ob.modifiers.new('Displace', 'DISPLACE')
    mod.texture = tex
    mod.vertex_group = vgrp
```

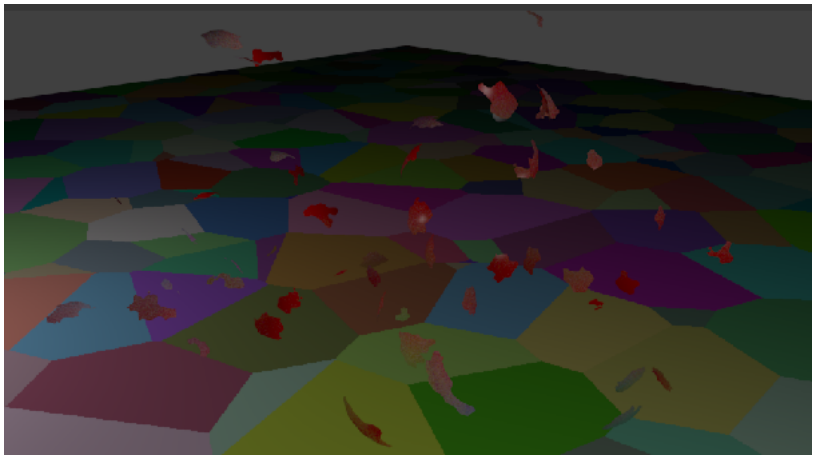
```
    mod.direction = 'NORMAL'
    mod.texture_coords = 'OBJECT'
    mod.texture_coords_object = empty
    mod.mid_level = 0.0
    mod.strength = 0.1
    print("'%s' '%s'" % (vgrp, mod.vertex_group))
    mod.vertex_group = vgrp
    print("'%s' '%s'" % (vgrp, mod.vertex_group))
    return mod

def createAndAnimateEmpty(origin):
    bpy.ops.object.add(type='EMPTY', location=origin)
    empty = bpy.context.object
    scn = bpy.context.scene
    scn.frame_current = 1
    bpy.ops.anim.keyframe_insert_menu(type='Location')
    scn.frame_current = 26
    bpy.ops.transform.translate(value=(1,0,1))
    bpy.ops.anim.keyframe_insert_menu(type='Location')
    scn.frame_current = 1
    for fcu in empty.animation_data.action.fcurves:
        fcu.extrapolation = 'LINEAR'
        for kp in fcu.keyframe_points:
            kp.interpolation = 'LINEAR'
    return empty

def run(origin):
    # Create flags and poles
    flag1 = makeFlag('SoftBodyFlag', origin+Vector((-3,0,0)), False)
    flag2 = makeFlag('ClothFlag', origin+Vector((0,0,0)), False)
    flag3 = makeFlag('DisplacementFlag', origin+Vector((3,0,0)), True)
    pole1 = makePole(origin+Vector((-3,0,0)))
    pole2 = makePole(origin+Vector((0,0,0)))
    pole3 = makePole(origin+Vector((3,0,0)))

    # Materials
    mat1 = addFlagMaterial('SoftBodyFlag', flag1, (1,0,0), (0,0,1))
    mat2 = addFlagMaterial('ClothFlag', flag2, (0,1,0), (1,1,0))
    mat3 = addFlagMaterial('DisplacementFlag', flag3, (1,0,1), (0,1,0))

    white = bpy.data.materials.new('White')
    white.diffuse_color = (1,1,1)
    pole1.data.materials.append(white)
    pole2.data.materials.append(white)
    pole3.data.materials.append(white)
```

```
    # Add modifiers and wind
    addSoftBodyModifier(flag1)
    addClothModifier(flag2)
    addWindEffector(origin+Vector((-1,-2,0)))

    # Create displacement
    tex3 = createDisplacementTexture(mat3)
    empty = createAndAnimateEmpty(origin + Vector((3,0,0)))
    mod = addDisplacementModifier(flag3, tex3, 'POLE', empty)

    return

if __name__ == "__main__":
    bpy.ops.object.select_by_type(type='MESH')
    bpy.ops.object.delete()
    run(Vector((0,0,0)))
    bpy.ops.screen.animation_play()
```

12.7 Particles and explode modifier

A bullet with an invisible particle system is fired at a crystal ball. The ball is shattered and the pieces fall to the floot.

The effect is achieved by giving the ball an explode modifier which is triggered by a particle system. The idea was to make this into a reaction particle system, which is triggered by the particle system of the bullet. However, reactor particles are apparently not yet implemented in Blender 2.5, so the balls particles are set to emit at a specific time instead.



```
#-------------------------------------------------------
# File crystal.py
#-------------------------------------------------------
import bpy, mathutils, math
from mathutils import *

def addSphere(name, size, origin):
    bpy.ops.mesh.primitive_ico_sphere_add(
        subdivisions=2,
        size=size,
        location=origin)
    bpy.ops.object.shade_smooth()
    bpy.ops.object.modifier_add(type='SUBSURF')
    ob = bpy.context.object
    ob.name = name
    return ob

def addFloor(name, origin, hidden):
    bpy.ops.mesh.primitive_plane_add(location=origin)
    bpy.ops.transform.resize(value=(30, 30, 30))
    floor = bpy.context.object
    floor.name = name
    if hidden:
        floor.hide = True
        floor.hide_render = True
        return floor

    # Floor material
    voronoi = bpy.data.textures.new('Voronoi', type = 'VORONOI')
    voronoi.color_mode = 'POSITION'
    voronoi.noise_scale = 0.1

    plastic = bpy.data.materials.new('Plastic')
    plastic.diffuse_color = (1,1,0)
    plastic.diffuse_intensity = 0.1
    mtex = plastic.texture_slots.add()
    mtex.texture = voronoi
    mtex.texture_coords = 'ORCO'
    mtex.color = (0,0,1)
    floor.data.materials.append(plastic)
    return floor

def run(origin):
    # ----------- Materials
    red = bpy.data.materials.new('Red')
    red.diffuse_color = (1,0,0)
```

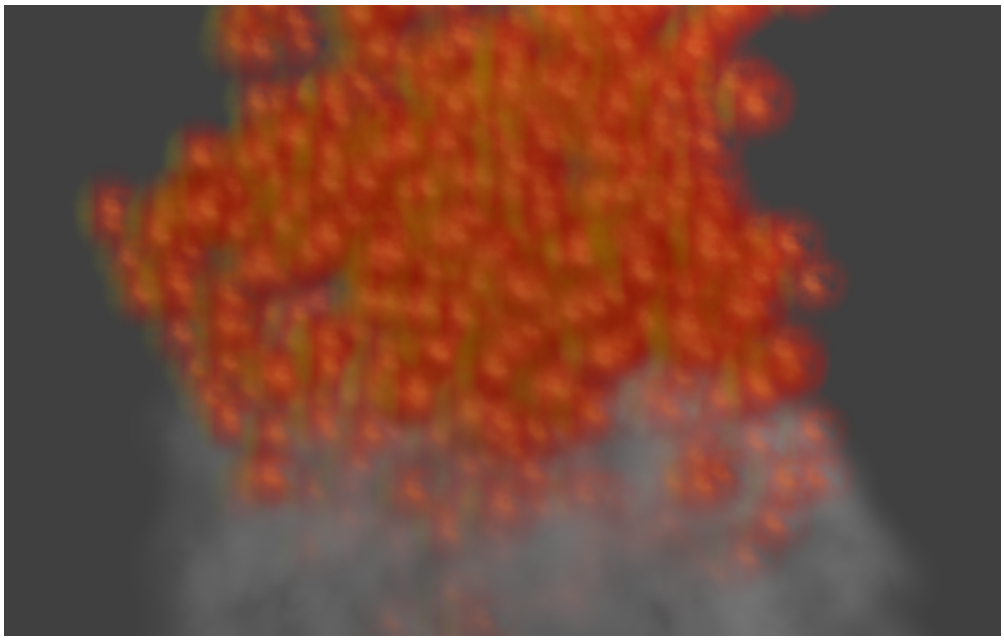
```
red.specular_hardness = 200
rmir = red.raytrace_mirror
rmir.use = True
rmir.distance = 0.001
rmir.fade_to = 'FADE_TO_MATERIAL'
rmir.distance = 0.0
rmir.reflect_factor = 0.7
rmir.gloss_factor = 0.4

grey = bpy.data.materials.new('Grey')
grey.diffuse_color = (0.5,0.5,0.5)

# ----------- Bullet - a small sphere
bullet = addSphere('Bullet', 0.2, origin)
bullet.data.materials.append(grey)

# Animate bullet
scn = bpy.context.scene
scn.frame_current = 51
bullet.location = origin
bpy.ops.anim.keyframe_insert_menu(type='Location')
bullet.location = origin+Vector((0,30,0))
scn.frame_current = 251
bpy.ops.anim.keyframe_insert_menu(type='Location')
scn.frame_current = 1
action = bullet.animation_data.action
for fcu in action.fcurves:
    fcu.extrapolation = 'LINEAR'
    for kp in fcu.keyframe_points:
        kp.interpolation = 'LINEAR'

# Trail particle system for bullet

bpy.ops.object.particle_system_add()
trail = bullet.particle_systems[0]
trail.name = 'Trail'
fset = trail.settings
# Emission
fset.name = 'TrailSettings'
fset.count = 1000
fset.frame_start = 1
fset.frame_end = 250
fset.lifetime = 25
fset.emit_from = 'FACE'
fset.use_render_emitter = True
# Velocity
```

```
fset.normal_factor = 1.0
fset.factor_random = 0.5
# Physics
fset.physics_type = 'NEWTON'
fset.mass = 0
# Set all effector weights to zero
ew = fset.effector_weights
ew.gravity = 0.0
# Don't render
fset.draw_method = 'DOT'
fset.render_type = 'NONE'

# -------------- Ball
ball = addSphere('Ball', 1.0, origin)
ball.data.materials.append(red)

# Particle system
bpy.ops.object.particle_system_add()
react = ball.particle_systems[0]
react.name = 'React'
fset = react.settings
# Emission
fset.name = 'TrailSettings'
fset.count = 50
fset.frame_start = 47
fset.frame_end = 57
fset.lifetime = 250
fset.emit_from = 'FACE'
fset.use_render_emitter = True
# Velocity
fset.normal_factor = 5.0
fset.factor_random = 2.5
# Physics
fset.physics_type = 'NEWTON'
fset.mass = 1.0
# Don't render
fset.draw_method = 'CROSS'
fset.render_type = 'NONE'

# Explode modifier
mod = ball.modifiers.new(name='Explode', type='EXPLODE')
mod.use_edge_cut = True
mod.show_unborn = True
mod.show_alive = True
mod.show_dead = True
mod.use_size = False
```

```
    # ----------- Hidden floor with collision modifier
    hidden = addFloor('Hidden', origin+Vector((0,0,-3.9)), True)
    mod = hidden.modifiers.new(name='Collision', type='COLLISION')
    mset = mod.settings
    mset.permeability = 0.01
    mset.stickness = 0.1
    mset.use_particle_kill = False
    mset.damping_factor = 0.6
    mset.damping_random = 0.2
    mset.friction_factor = 0.3
    mset.friction_random = 0.1

    addFloor('Floor', Vector((0,0,-4)), False)
    return

if __name__ == "__main__":
    bpy.ops.object.select_all(action='SELECT')
    bpy.ops.object.delete()

    # Camera, lights
    bpy.ops.object.camera_add(
        location = Vector((12,-12,4)),
        rotation = Vector((70,0,45))*math.pi/180)
    cam = bpy.context.object.data
    cam.lens = 35
    bpy.ops.object.lamp_add(type='POINT',
        location = Vector((11,-7,6)))
    bpy.ops.object.lamp_add(type='POINT',
        location =Vector((-7,-10,2)))

    run(Vector((0,0,0)))
```

12.8 Particle fire and smoke

This program adds two particle systems for fire and smoke. The particles render as billboards with procedural textures.



```
#----------------------------------------------------
# File fire.py
#----------------------------------------------------
import bpy, mathutils, math
from mathutils import Vector, Matrix
from math import pi

def createEmitter(origin):
    bpy.ops.mesh.primitive_plane_add(location=origin)
    emitter = bpy.context.object
    bpy.ops.mesh.uv_texture_add()
    return emitter

def createFire(emitter):
    # Add first particle system
    bpy.context.scene.objects.active = emitter
    bpy.ops.object.particle_system_add()
    fire = emitter.particle_systems[-1]
    fire.name = 'Fire'
    fset = fire.settings

    # Emission
    fset.name = 'FireSettings'
    fset.count = 100
    fset.frame_start = 1
```

```
    fset.frame_end = 200
    fset.lifetime = 70
    fset.lifetime_random = 0.2
    fset.emit_from = 'FACE'
    fset.use_render_emitter = False
    fset.distribution = 'RAND'
    fset.object_align_factor = (0,0,1)

    # Velocity
    fset.normal_factor = 0.55
    fset.factor_random = 0.5

    # Physics
    fset.physics_type = 'NEWTON'
    fset.mass = 1.0
    fset.particle_size = 10.0
    fset.use_multiply_size_mass = False

    # Effector weights
    ew = fset.effector_weights
    ew.gravity = 0.0
    ew.wind = 1.0

    # Display and render
    fset.draw_percentage = 100
    fset.draw_method = 'RENDER'
    fset.material = 1
    fset.particle_size = 0.3
    fset.render_type = 'BILLBOARD'
    fset.render_step = 3

    # Children
    fset.child_type = 'SIMPLE'
    fset.rendered_child_count = 50
    fset.child_radius = 1.1
    fset.child_roundness = 0.5
    return fire

def createSmoke(emitter):
    # Add second particle system
    bpy.context.scene.objects.active = emitter
    bpy.ops.object.particle_system_add()
    smoke = emitter.particle_systems[-1]
    smoke.name = 'Smoke'
    sset = smoke.settings
```

```
    # Emission
    sset.name = 'FireSettings'
    sset.count = 100
    sset.frame_start = 1
    sset.frame_end = 100
    sset.lifetime = 70
    sset.lifetime_random = 0.2
    sset.emit_from = 'FACE'
    sset.use_render_emitter = False
    sset.distribution = 'RAND'

    # Velocity
    sset.normal_factor = 0.0
    sset.factor_random = 0.5

    # Physics
    sset.physics_type = 'NEWTON'
    sset.mass = 2.5
    sset.particle_size = 0.3
    sset.use_multiply_size_mass = True

    # Effector weights
    ew = sset.effector_weights
    ew.gravity = 0.0
    ew.wind = 1.0

    # Display and render
    sset.draw_percentage = 100
    sset.draw_method = 'RENDER'
    sset.material = 2
    sset.particle_size = 0.5
    sset.render_type = 'BILLBOARD'
    sset.render_step = 3

    # Children
    sset.child_type = 'SIMPLE'
    sset.rendered_child_count = 50
    sset.child_radius = 1.6
    return smoke

def createWind(origin):
    bpy.ops.object.effector_add(
        type='WIND',
        enter_editmode=False,
        location = origin - Vector((0,3,0)),
        rotation = (-pi/2, 0, 0))
```

```
    wind = bpy.context.object

    # Field settings
    fld = wind.field
    fld.strength = 2.3
    fld.noise = 3.2
    fld.flow = 0.3
    return wind

def createColorRamp(tex, values):
    tex.use_color_ramp = True
    ramp = tex.color_ramp
    for n,value in enumerate(values):
        elt = ramp.elements[n]
        (pos, color) = value
        elt.position = pos
        elt.color = color
    return

def createFlameTexture():
    tex = bpy.data.textures.new('Flame', type = 'CLOUDS')
    createColorRamp(tex, [(0.2, (1,0.5,0.1,1)), (0.8, (0.5,0,0,0))])
    tex.noise_type = 'HARD_NOISE'
    tex.noise_scale = 0.7
    tex.noise_depth = 5
    return tex

def createStencilTexture():
    tex = bpy.data.textures.new('Stencil', type = 'BLEND')
    tex.progression = 'SPHERICAL'
    createColorRamp(tex, [(0.0, (0,0,0,0)), (0.85, (1,1,1,0.6))])
    return tex

def createEmitTexture():
    tex = bpy.data.textures.new('Emit', type = 'BLEND')
    tex.progression = 'LINEAR'
    createColorRamp(tex, [(0.1, (1,1,0,1)), (0.3, (1,0,0,1))])
    return tex

def createSmokeTexture():
    tex = bpy.data.textures.new('Smoke', type = 'CLOUDS')
    createColorRamp(tex, [(0.2, (0,0,0,1)), (0.6, (1,1,1,1))])
    tex.noise_type = 'HARD_NOISE'
    tex.noise_scale = 1.05
    tex.noise_depth = 5
    return tex
```

```
def createFireMaterial(textures, objects):
    (flame, stencil, emit) = textures
    (emitter, empty) = objects

    mat = bpy.data.materials.new('Fire')
    mat.specular_intensity = 0.0
    mat.use_transparency = True
    mat.transparency_method = 'Z_TRANSPARENCY'
    mat.alpha = 0.0
    mat.use_raytrace = False
    mat.use_face_texture = True
    mat.use_shadows = False
    mat.use_cast_buffer_shadows = True

    mtex = mat.texture_slots.add()
    mtex.texture = emit
    mtex.texture_coords = 'UV'
    mtex.use_map_color_diffuse = True

    mtex = mat.texture_slots.add()
    mtex.texture = stencil
    mtex.texture_coords = 'UV'
    mtex.use_map_color_diffuse = False
    mtex.use_map_emit = True
    mtex.use_stencil = True

    mtex = mat.texture_slots.add()
    mtex.texture = flame
    mtex.texture_coords = 'UV'
    mtex.use_map_color_diffuse = True
    mtex.use_map_alpha = True
    #mtex.object = empty
    return mat

def createSmokeMaterial(textures, objects):
    (smoke, stencil) = textures
    (emitter, empty) = objects

    mat = bpy.data.materials.new('Smoke')
    mat.specular_intensity = 0.0
    mat.use_transparency = True
    mat.transparency_method = 'Z_TRANSPARENCY'
    mat.alpha = 0.0
    mat.use_raytrace = False
    mat.use_face_texture = True
```

```
    mat.use_shadows = True
    mat.use_cast_buffer_shadows = True

    mtex = mat.texture_slots.add()
    mtex.texture = stencil
    mtex.texture_coords = 'UV'
    mtex.use_map_color_diffuse = False
    mtex.use_map_alpha = True
    mtex.use_stencil = True

    mtex = mat.texture_slots.add()
    mtex.texture = smoke
    mtex.texture_coords = 'OBJECT'
    mtex.object = empty
    return mat

def run(origin):
    emitter = createEmitter(origin)
    #wind = createWind()
    bpy.ops.object.add(type='EMPTY')
    empty = bpy.context.object

    fire = createFire(emitter)
    flameTex = createFlameTexture()
    stencilTex = createStencilTexture()
    emitTex = createEmitTexture()
    flameMat = createFireMaterial(
        (flameTex, stencilTex, emitTex),
        (emitter, empty))
    emitter.data.materials.append(flameMat)

    smoke = createSmoke(emitter)
    smokeTex = createSmokeTexture()
    smokeMat = createSmokeMaterial(
        (smokeTex, stencilTex), (emitter, empty))
    emitter.data.materials.append(smokeMat)
    return

if __name__ == "__main__":
    bpy.ops.object.select_by_type(type='MESH')
    bpy.ops.object.delete()
    run((0,0,0))
    bpy.ops.screen.animation_play(reverse=False, sync=False)
```

12.9 Smoke

This program sets up a smoke simulation and assigns a voxel material.

```
#----------------------------------------------------------
# File smoke.py
# Creates smoke and smoke material.
# Heavily inspired by Andrew Price's tutorial at
# http://www.blenderguru.com/introduction-to-smoke-simulation/
#----------------------------------------------------------
import bpy, mathutils, math
from mathutils import Vector
from math import pi

def createDomain(origin):
    # Add cube as domain
    bpy.ops.mesh.primitive_cube_add(location=origin)
    bpy.ops.transform.resize(value=(4, 4, 4))
    domain = bpy.context.object
    domain.name = 'Domain'

    # Add domain modifier
    dmod = domain.modifiers.new(name='Smoke', type='SMOKE')
    dmod.smoke_type = 'DOMAIN'
    dset = dmod.domain_settings
```

```
    # Domain settings
    dset.resolution_max = 32
    dset.alpha = -0.001
    dset.beta = 2.0
    dset.time_scale = 1.2
    dset.vorticity = 2.0
    dset.use_dissolve_smoke = True
    dset.dissolve_speed = 80
    dset.use_dissolve_smoke_log = True
    dset.use_high_resolution = True
    dset.show_high_resolution = True

    # Effector weights
    ew = dset.effector_weights
    ew.gravity = 0.4
    ew.force = 0.8
    return domain

def createFlow(origin):
    # Add plane as flow
    bpy.ops.mesh.primitive_plane_add(location = origin)
    bpy.ops.transform.resize(value=(2, 2, 2))
    flow = bpy.context.object
    flow.name = 'Flow'

    # Add smoke particle system
    pmod = flow.modifiers.new(name='SmokeParticles', type='PARTICLE_SYSTEM')
    pmod.name = 'SmokeParticles'
    psys = pmod.particle_system
    psys.seed = 4711

    # Particle settings
    pset = psys.settings
    pset.type = 'EMITTER'
    pset.lifetime = 1
    pset.emit_from = 'VOLUME'
    pset.use_render_emitter = False
    pset.render_type = 'NONE'
    pset.normal_factor = 8.0

    # Add smoke modifier
    smod = flow.modifiers.new(name='Smoke', type='SMOKE')
    smod.smoke_type = 'FLOW'
    sfset = smod.flow_settings

    # Flow settings
```

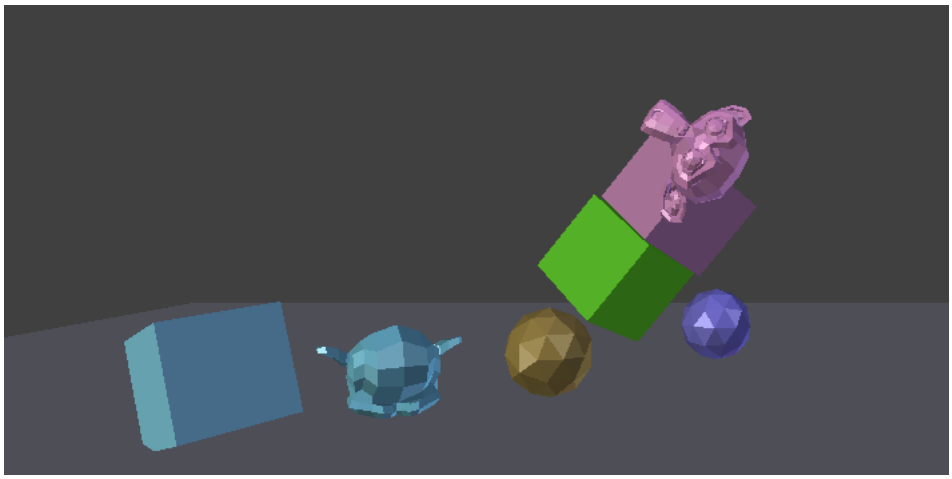
```
    sfset.use_outflow = False
    sfset.temperature = 0.7
    sfset.density = 0.8
    sfset.initial_velocity = True
    sfset.particle_system = psys
    return flow

def createVortexEffector(origin):
    bpy.ops.object.effector_add(type='VORTEX', location=origin)
    vortex = bpy.context.object
    return vortex

def createVoxelTexture(domain):
    tex = bpy.data.textures.new('VoxelTex', type = 'VOXEL_DATA')
    voxdata = tex.voxel_data
    voxdata.file_format = 'SMOKE'
    voxdata.domain_object = domain
    return tex

def createVolumeMaterial(tex):
    mat = bpy.data.materials.new('VolumeMat')
    mat.type = 'VOLUME'
    vol = mat.volume
    vol.density = 0.0
    vol.density_scale = 8.0
    vol.scattering = 6.0
    vol.asymmetry = 0.3
    vol.emission = 0.3
    vol.emission_color = (1,1,1)
    vol.transmission_color = (0.9,0.2,0)
    vol.reflection = 0.7
    vol.reflection_color = (0.8,0.9,0)
    # To remove pixelation effects
    vol.step_size = 0.05

    # Add voxdata texture
    mtex = mat.texture_slots.add()
    mtex.texture = tex
    mtex.texture_coords = 'ORCO'
    mtex.use_map_density = True
    mtex.use_map_emission = True
    mtex.use_map_scatter = False
    mtex.use_map_reflect = True
    mtex.use_map_color_emission = True
    mtex.use_map_color_transmission = True
    mtex.use_map_color_reflection = True
```

```
    mtex.density_factor = 1.0
    mtex.emission_factor = 0.2
    mtex.scattering_factor = 0.2
    mtex.reflection_factor = 0.3
    mtex.emission_color_factor = 0.9
    mtex.transmission_color_factor = 0.5
    mtex.reflection_color_factor = 0.6
    return mat

def addFloor(origin):
    # Create floor that receives transparent shadows
    bpy.ops.mesh.primitive_plane_add(
        location = origin,
        rotation = (0, 0, pi/4))
    bpy.ops.transform.resize(value=(4, 4, 4))
    bpy.ops.transform.resize(value=(2, 2, 2),
        constraint_axis=(True, False, False),
        constraint_orientation='LOCAL')
    floor = bpy.context.object
    mat = bpy.data.materials.new('Floor')
    mat.use_transparent_shadows = True
    floor.data.materials.append(mat)
    return

def setupWorld():
    scn = bpy.context.scene
    # Blue blend sky
    scn.world.use_sky_blend = True
    scn.world.horizon_color = (0.25, 0.3, 0.4)
    scn.world.zenith_color = (0, 0, 0.7)
    # PAL 4:3 render
    scn.render.resolution_x = 720
    scn.render.resolution_y = 567
    return

def run(origin):
    domain = createDomain(origin)
    flow = createFlow(origin-Vector((0,0,3.5)))
    vortex = createVortexEffector(origin)
    tex = createVoxelTexture(domain)
    mat = createVolumeMaterial(tex)
    domain.data.materials.append(mat)
    return

if __name__ == "__main__":
```

```
for ob in bpy.context.scene.objects:
    bpy.context.scene.objects.unlink(ob)

addFloor(Vector((0,0,-4)))
setupWorld()
# Lights and camera
bpy.ops.object.lamp_add( type = 'POINT', location=(4,6,1))
bpy.ops.object.lamp_add( type = 'POINT', location=(-7,-5,0))
bpy.ops.object.camera_add(location=Vector((8,-8,3)),
    rotation=(pi/3, 0, pi/6))

run(Vector((0,0,0)))
bpy.ops.screen.animation_play()
```

12.10 Rigid body simulation

This program uses the Blender game engine to simulate a pile of objects falling to the ground. The animation is recorded and can be replayed afterwards.



```
#-------------------------------------------------------
# File pile.py
#-------------------------------------------------------
import bpy, mathutils, math, random
from mathutils import Vector

NObjects = 7
```

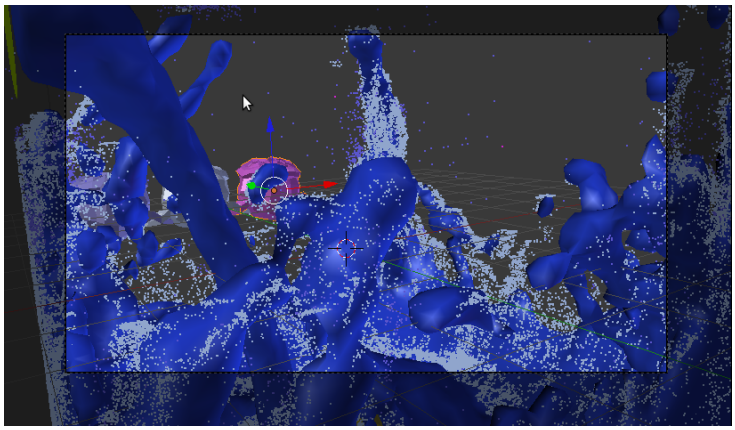
```
Seed = 444

def addSceneGameSettings(scn):
    # SceneGameData
    sgdata = scn.game_settings
    sgdata.fps = 25
    sgdata.frequency = True
    sgdata.material_mode = 'GLSL'
    sgdata.show_debug_properties = True
    sgdata.show_framerate_profile = True
    sgdata.show_fullscreen = True
    sgdata.show_physics_visualization = True
    sgdata.use_animation_record = True
    return

def addMonkeyGameSettings(ob):
    # GameObjectSettings
    goset = ob.game
    goset.physics_type = 'RIGID_BODY'
    goset.use_actor = True
    goset.use_ghost = False
    goset.mass = 7.0
    goset.damping = 0.0

    goset.use_collision_bounds = True
    goset.collision_bounds_type = 'BOX'

    goset.show_actuators = True
    goset.show_controllers = True
    goset.show_debug_state = True
    goset.show_sensors = True
    goset.show_state_panel = True

    return

def run(origin):
    # Change render engine from BLENDER_RENDER to BLENDER_GAME
    bpy.context.scene.render.engine = 'BLENDER_GAME'

    # Create floor
    bpy.ops.mesh.primitive_plane_add(location=origin)
    bpy.ops.transform.resize(value=(20, 20, 20))
    floor = bpy.context.object
    mat = bpy.data.materials.new(name = 'FloorMaterial')
    mat.diffuse_color = (0.5, 0.5, 0.5)
```

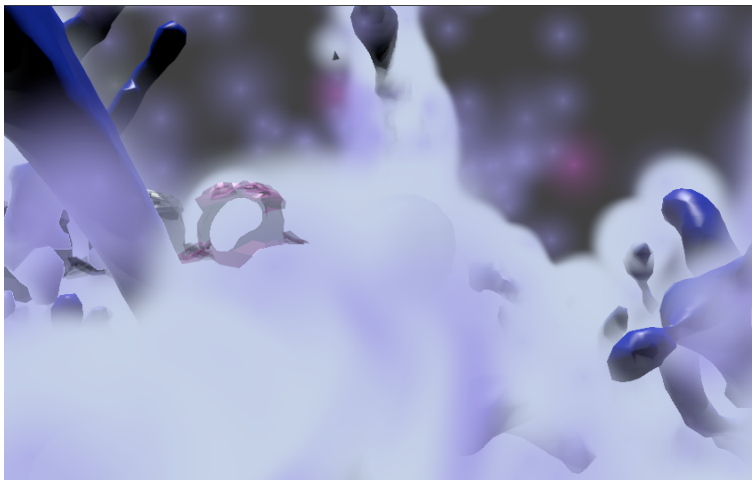
```
    # Create a pile of objects
    objectType = ["cube", "ico_sphere", "monkey"]
    objects = []
    deg2rad = math.pi/180
    random.seed(Seed)
    for n in range(NObjects):
        x = []
        for i in range(3):
            x.append( random.randrange(0, 360, 1) )
        dx = 0.5*random.random()
        dy = 0.5*random.random()
        obType = objectType[ random.randrange(0, 3, 1) ]
        fcn = eval("bpy.ops.mesh.primitive_%s_add" % obType)
        fcn(location=origin+Vector((dx, dy, 3*n+3)),
            rotation=deg2rad*Vector((x[0], x[1], x[2])))
        ob = bpy.context.object
        objects.append( ob )
        mat = bpy.data.materials.new(name='Material_%02d' % n)
        c = []
        for j in range(3):
            c.append( random.random() )
        mat.diffuse_color = c
        ob.data.materials.append(mat)

    # Set game settings for floor
    fset = floor.game
    fset.physics_type = 'STATIC'

    # Set game settings for monkeys
    for n in range(NObjects):
        addMonkeyGameSettings(objects[n])

    # Set game settings for scene
    scn = bpy.context.scene
    addSceneGameSettings(scn)
    scn.frame_start = 1
    scn.frame_end = 200
    return

if __name__ == "__main__":
    bpy.ops.object.select_by_type(type='MESH')
    bpy.ops.object.delete()
    run(Vector((0,0,0)))
    bpy.ops.view3d.game_start()
```

12.11 Fluids

This program sets up a fluid simulation with a domain, a fluid, a moving obstacle, an inflow, an outflow, and three kinds of drops. Note that we must bake the simulation first; I don't think that used to be necessary.

The pictures are from frame 57, after some materials have been added. The drop dominate the render completely unless they are given a low opacity, around alpha = 0.2.





```
#------------------------------------------------------------
# File fluid.py
#------------------------------------------------------------
import bpy, math
from mathutils import Vector
```

```
from math import pi

def createDomain(origin):
    bpy.ops.mesh.primitive_cube_add(location=origin)
    bpy.ops.transform.resize(value=(4, 4, 4))
    domain = bpy.context.object
    domain.name = 'Domain'
    bpy.ops.object.shade_smooth()
    # Add domain modifier
    mod = domain.modifiers.new(name='FluidDomain', type='FLUID_SIMULATION')
    # mod.settings is FluidSettings
    mod.settings.type = 'DOMAIN'
    # mod.settings now changed to DomainFluidSettings
    settings = mod.settings
    settings.use_speed_vectors = False
    settings.simulation_scale = 3.0
    settings.slip_type = 'FREESLIP'
    settings.tracer_particles = 10
    settings.generate_particles = 1.5
    #settings.start_time = 0.0
    #settings.end_time = 2.0
    return domain


def createFluid(origin):
    bpy.ops.mesh.primitive_ico_sphere_add(
        size=3.5,
        subdivisions=1,
        location=origin)
    fluid = bpy.context.object
    fluid.name = 'Fluid'
    fluid.hide = True
    fluid.hide_render = True
    # Add fluid modifier.
    mod = fluid.modifiers.new(name='Fluid', type='FLUID_SIMULATION')
    mod.settings.type = 'FLUID'
    return fluid

def createObstacle(origin):
    bpy.ops.mesh.primitive_cylinder_add(
        vertices=12,
        radius=0.3,
        depth=2,
        cap_ends=True,
        location=origin + Vector((0,0,-2.5)),
        rotation=(pi/2, 0, 0))
```

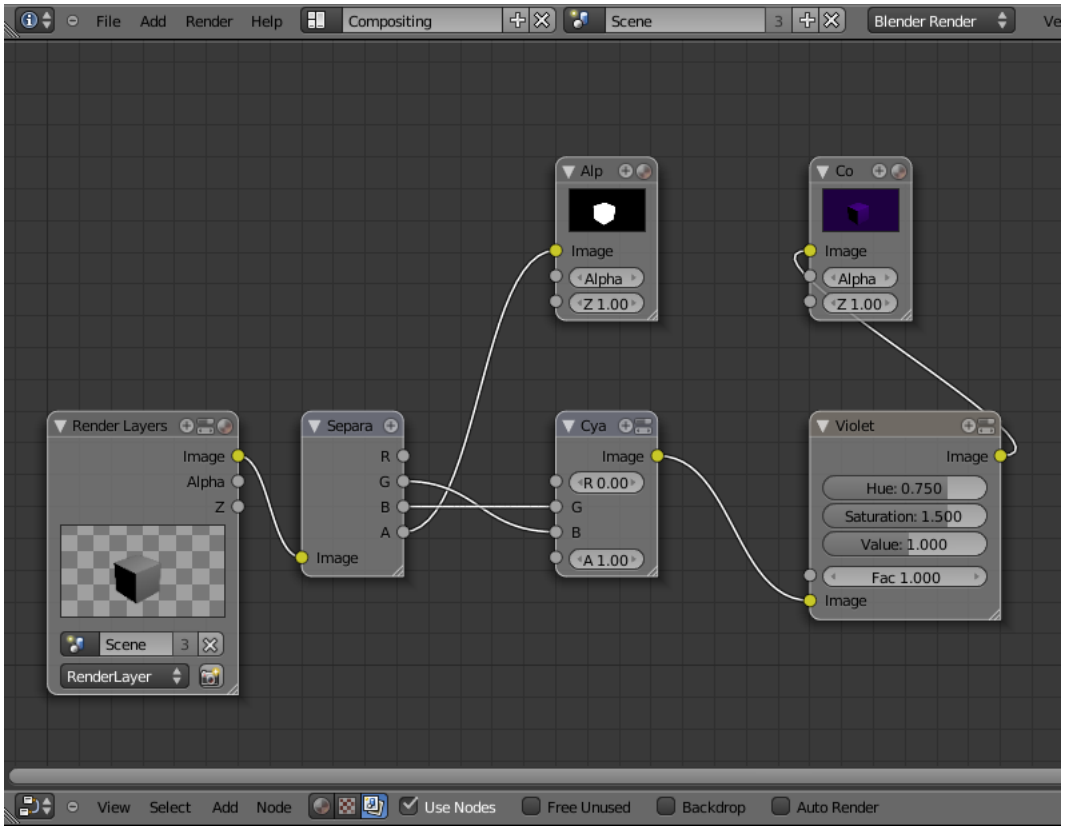
```
    bpy.ops.object.modifier_add(type='FLUID_SIMULATION')
    obst = bpy.context.object
    obst.name = 'Obstacle'
    # Add fluid modifier
    bpy.ops.object.modifier_add(type='FLUID_SIMULATION')
    mod = obst.modifiers[-1]
    mod.settings.type = 'OBSTACLE'

    # Animate obstacle
    scn = bpy.context.scene
    scn.frame_current = 1
    bpy.ops.anim.keyframe_insert_menu(type='Rotation')
    scn.frame_current = 26
    bpy.ops.transform.rotate(value=(pi/2,), axis=(-0, -0, -1))
    bpy.ops.anim.keyframe_insert_menu(type='Rotation')
    scn.frame_current = 1
    for fcu in obst.animation_data.action.fcurves:
        fcu.extrapolation = 'LINEAR'
        for kp in fcu.keyframe_points:
            kp.interpolation = 'LINEAR'
    return obst

def createInflow(origin):
    bpy.ops.mesh.primitive_circle_add(
        radius=0.75,
        fill=True,
        location=origin+Vector((-3.9,0,3)),
        rotation=(0, pi/2, 0))
    inflow = bpy.context.object
    inflow.name = 'Inflow'
    # Add fluid modifier
    bpy.ops.object.modifier_add(type='FLUID_SIMULATION')
    mod = inflow.modifiers[-1]
    mod.settings.type = 'INFLOW'
    settings = mod.settings
    settings.inflow_velocity = (1.5,0,0)
    settings.volume_initialization = 'SHELL'
    return inflow

def createOutflow(origin):
    bpy.ops.mesh.primitive_circle_add(
        radius=0.75,
        fill=True,
        location=origin+Vector((3.9,0,-3)),
        rotation=(0, -pi/2, 0))
    outflow = bpy.context.object
```

```
    outflow.name = 'Outflow'
    # Add fluid modifier
    bpy.ops.object.modifier_add(type='FLUID_SIMULATION')
    mod = outflow.modifiers[-1]
    mod.settings.type = 'OUTFLOW'
    mod.settings.volume_initialization = 'SHELL'
    return outflow

def createFluidParticle(name, origin, data):
    bpy.ops.mesh.primitive_monkey_add(location=origin)
    monkey = bpy.context.object
    monkey.name = name
    # Add fluid modifier
    bpy.ops.object.modifier_add(type='FLUID_SIMULATION')
    mod = monkey.modifiers[-1]
    mod.settings.type = 'PARTICLE'
    (drops, floats, tracer) = data
    mod.settings.use_drops = drops
    mod.settings.use_floats = floats
    mod.settings.show_tracer = tracer

    # Setting type to particle created a particle system
    psys = monkey.modifiers[-1].particle_system
    psys.name = name+'Psys'
    #psys.settings.name = name+'Pset'
    return (mod.settings, None)

def run(origin):
    domain = createDomain(origin)
    fluid = createFluid(origin)
    obst = createObstacle(origin)
    inflow = createInflow(origin)
    outflow = createOutflow(origin)

    (settings, pset) = createFluidParticle('Drops',
        origin+Vector((-2,7,0)), (True, False, False))
    settings.particle_influence = 0.7
    settings.alpha_influence = 0.3

    (settings, pset) = createFluidParticle('Floats',
        origin+Vector((0,7,0)), (False, True, False))

    (settings, pset) = createFluidParticle('Tracer',
        origin+Vector((2,7,0)), (False, False, True))
    settings.particle_influence = 1.5
    settings.alpha_influence = 1.2
```

```
    return

if __name__ == "__main__":
    bpy.ops.object.select_all(action='SELECT')
    bpy.ops.object.delete()
    run(Vector((0,0,0)))
    #bpy.ops.fluid.bake()
```

13 Nodes

This program creates a node network.




```
#----------------------------------------------
# File nodes.py
#----------------------------------------------
```

```
import bpy, math

# switch on nodes
bpy.context.scene.use_nodes = True
tree = bpy.context.scene.node_tree
links = tree.links

# clear default nodes
for n in tree.nodes:
    tree.nodes.remove(n)

# create input render layer  node
rl = tree.nodes.new('R_LAYERS')
rl.location = 0,200

# create SEP_RGBA node
sep = tree.nodes.new('SEPRGBA')
sep.name = "Split"
sep.location = 200,200
links.new(rl.outputs[0],sep.inputs[0])  # image-image

# create VIEWER node
viewer = tree.nodes.new('VIEWER')
viewer.label = "Alpha"
viewer.location = 400,400
links.new(sep.outputs[3],viewer.inputs[0])  # A-image

# create COMBRGBA node
comb = tree.nodes.new('COMBRGBA')
comb.label = "Cyan"
comb.location = 400,200
links.new(sep.outputs[1],comb.inputs[2])  # G - B
links.new(sep.outputs[2],comb.inputs[1])  # B - G

# create HUE_SAT node
hs = tree.nodes.new('HUE_SAT')
hs.label = "Violet"
hs.location = 600,200
hs.color_hue = 0.75
hs.color_saturation = 1.5
links.new(comb.outputs[0],hs.inputs[1])  # image-image

# create output node
comp = tree.nodes.new('COMPOSITE')
comp.location = 600,400
links.new(hs.outputs[0],comp.inputs[0])  # image-image
```

14 Batch run

The program runs all scripts in the object and simulation folders. The main purpose is to verify that all scripts run correctly, or at least that they can be executed without causing errors.

Most scripts do not run in earlier versions of Blender. To ensure that we are not stuck with an outdated Blender, we first check the current Blender version, which is available as `bpy.app.version`.

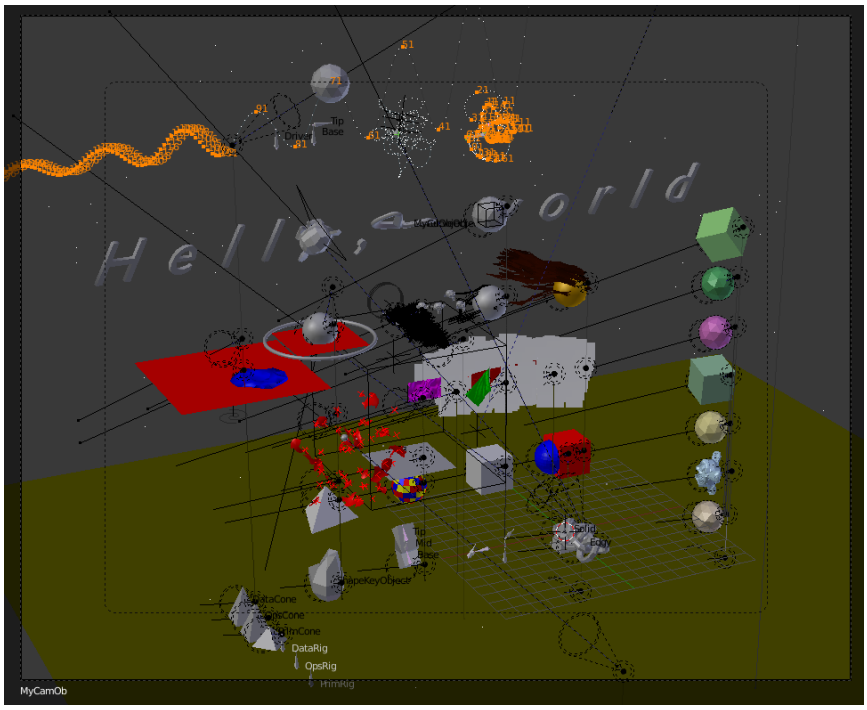
```
#------------------------------------------------------------
# File batch.py
#------------------------------------------------------------
import bpy, sys, os, mathutils
from mathutils import Vector

# Check Blender version
version = [2, 58, 0]
(a,b,c) = bpy.app.version
if b < version[1] or (b == version[1] and c < version[2]):
    msg = 'Blender too old: %s < %s' % ((a,b,c), tuple(version))
    raise NameError(msg)

# Delete all old objects, so we start with a clean slate.
scn = bpy.context.scene
for ob in scn.objects:
    scn.objects.active = ob
    print("Delete", ob, bpy.context.object)
    bpy.ops.object.mode_set(mode='OBJECT')
    scn.objects.unlink(ob)
    del ob

# Path to the code. You must change this unless you place the
# snippets folder in your home directory
scripts = os.path.expanduser('~/snippets/scripts/')
for folder in ['object', 'simulation', 'interface']:
sys.path.append(scripts+folder)
print(sys.path)

origin = Vector((0,0,0))
```

```
# Meshes and armatures
origin[2] = 0
import meshes
meshes.run(origin)
origin[0] += 5
import armature
armature.run(origin)
origin[0] += 5
import rigged_mesh
rigged_mesh.run(origin)
origin[0] += 5
import shapekey
shapekey.run(origin)
origin[0] += 5

# Three ways to construct objects
import objects
objects.run(origin)
origin[0] += 5

# Materials and textures
origin[2] = 5
origin[0] = 0
import material
material.run(origin)
origin[0] += 5
import texture
texture.run(origin)
origin[0] += 5
import multi_material
multi_material.run(origin)
origin[0] += 5
import uvs
uvs.run(origin)
origin[0] += 5
import chain
chain.run(origin)

# Actions and drivers
origin[2] = 25
origin[0] = 0
import ob_action
ob_action.run(origin)
origin[0] += 5
import pose_action
pose_action.run(origin)
```

```
origin[0] += 5
import epicycle
epicycle.run(origin)
origin[0] += 5
import driver
driver.run(origin)

# Simulations
origin[2] = 15
origin[0] = 0
import hair
hair.run(origin)
origin[0] += 5
import edit_hair
edit_hair.run(origin)
origin[0] += 5
import particle
particle.run(origin)
origin[0] += 5
import cloth
cloth.run(origin)
origin[0] += 5
import softbody
softbody.run(origin)

origin[2] = 10
origin[0] = 0
import fire
fire.run(origin)
origin[0] += 5
import flags
flags.run(origin)
origin[0] += 5
import smoke
smoke.run(origin)
origin[0] += 5
import crystal
crystal.run(origin)
origin[0] += 5

origin[2] = -4.02
origin[0] = -10
import pile
pile.run(origin)
# Restore render engine
bpy.context.scene.render.engine = 'BLENDER_RENDER'
```

```
# Other data types
origin[2] = 20
origin[0] = 0
import text
text.run(origin)
origin[0] += 5
import lattice
lattice.run(origin)
origin[0] += 5
import curve
curve.run(origin)
origin[0] += 5
import path
path.run(origin)
import camera
camera.run(Vector((0,0,0)))

# Layers and groups
import layers
layers.run()
import groups
groups.run()
# Restore layers after layers and groups
scn.layers[0] = True
for n in range(1,20):
scn.layers[n] = False

# View
import world
world.run()
import view
view.run()
```

At frame 71, your screen should look as the picture below. The rendered version appears on the front page.


Hello, world
Tip
Base
DataRig
OpsRig
PrimRig
MyCamOb