ratcave Documentation

Release 0.5

Nicholas A Del Grosso

October 10, 2016

Contents

CHAPTER 1

Table of Contents:

1.1 Introduction

ratCAVE is a Python package for displaying 3D Graphics. It was inspired by a Virtual Reality CAVE Setup for rodents in a neuroscience lab in Munich, Germany, and was meant to make the creation of 3D experiments simple and accessible.

ratCAVE has since evolved into a standalone wrapper for modern OpenGL constructs, like programmable shaders, environment mapping, and deferred rendering. Because it wraps these OpenGL features directly, it also works with all popular python OpenGL graphics engines, including Pyglet, PsychoPy, and PyGame.

Finally, ratCAVE is written to reduce boilerplate code, in order to make writing simple 3D environments easy. It does this using many python features, including dictionary-like uniform assignment and context managers to bind OpenGL objects.

1.2 Installation

ratCAVE supports both Python 2 and Python 3, and can be installed via pip!:

```
pip install ratcave
```

1.3 Features

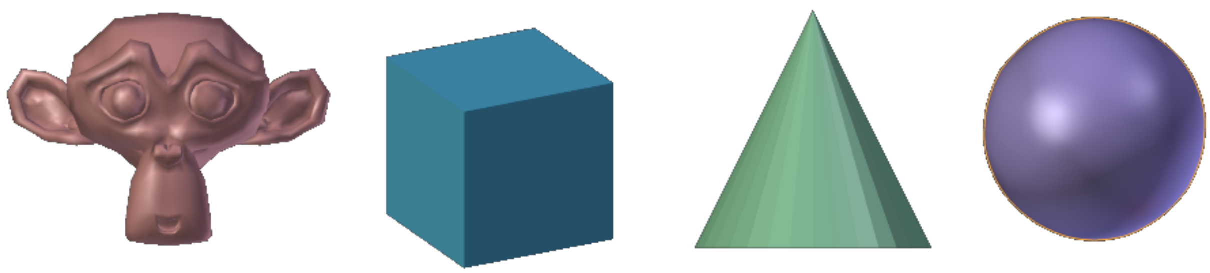
ratCAVE was created to be an grapics package for doing behavioral experiments with animals in a freely-moving virtual reality environment. The goals of this project are:

- *Less Boilerplate, more Coding*: More code means more errors. Many behavioral experiments tend to be complex sets of logic written in a single script, so I tried to make ratCAVE as low-profile as possible to keep the focus on the experiment, not on the graphics management.
- *Ease of Use*: Moving objects in a scene, displaying a window, and changing objects' colors should be intuitive.
- *high-temporal performance*: Lag is the enemy of immersive VR, and we wanted to take advantage of our 360 fps display for VR research. Advanced hardware-accelerated algorithms and modern OpenGL constructs are an essential part of doing high-performance graphics in Python.
- *Cubemapping Support* (the essential algorithmic approach for a single-projector CAVE VR system),
- *Free and Open Source*

What I've found so far is that ratCAVE makes for a succinct 3D graphics engine, even for simple 3D scenes, making it a useful candidate package for psychophysics research in general. To that end, I've made it very compatible with the PsychoPy package, as a way of extending PsychoPy experiments to 3D scenes. While we are still at an early stage of development with ratCAVE, we've already reached the requirements listed above, with a goal of continually refactoring and adding features to make ratCAVE the mature backend it has the potential to be. If you are interested in aiding the development of ratCAVE, either through contributions on GitHub, bug reporting, or even simply testing it out yourself and giving us feedback, we hope you'll get involved and help us develop this little project into something wonderful!

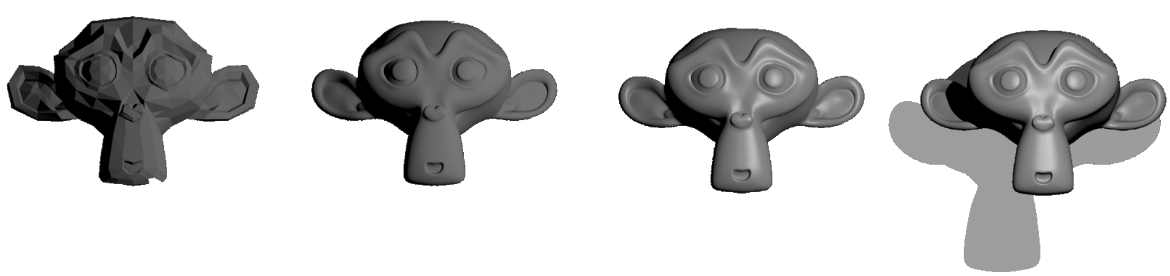
1.3.1 Supplied 3D Primitives

Blender 3D's built-in primitives (Cone, Sphere, Cube, etc) come packaged with ratCAVE, making it easier to get started and prototype your 3D application. A reader object for Blender's .obj Wavefront files is also included.



1.3.2 Supplied 3D Shaders

ratCAVE is "batteries-included": You get diffuse shading, specular reflections, shadows, and even FXAA antialiasing in the packaged shaders. These shaders are open-source and free to be edited and improved!



1.3.3 Pythonic Interface

FrameBuffer Context Managers

Normally, the OpenGL code to bind a framebuffer involves the following:

```
glGetIntegerv(GL_VIEWPORT, old_viewport_size)
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fbo_id)  # Rendering off-screen
glViewport(0, 0, texture_width, texture_height)
<< Draw Scene Here >>
```

```
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0)
glViewport(old_viewport_size)
```

In ratCAVE, this is a simple context manager:

```
with fbo:
   scene.draw()
```

Shader Uniforms

OpenGL Shader Uniform creation and setting is also wrapped in a pythonic way:

```
sphere.uniforms['diffuse_color'] = [1., 0., 0.]  # RGB values
```

1.3.4 Fast Execution

ratCAVE uses Numpy arrays, c binaries, and GLSL OpenGL to make rendering detailed scenes fast!

1.4 System Requirements

At the moment, ratCAVE's shaders require OpenGL 3.3, though this is planned to change in future releases. If you'd like to use ratCAVE and don't have a graphics driver that supports OpenGL 3.3, however, you can already load your own shaders and it will work fine.

1.5 Tutorials

These Tutorials are meant to help you get started!

1.5.1 Tutorial 1: Displaying a 3D Object

This tutorial will show the process of displaying a 3D object onscreen. This will be done in four steps:

- We'll open a file containing 3D objects–a Wavefront .obj file containing basic 3D primitives that comes with ratCAVE (although you can use any .obj file outputted by 3D modeling software), using the *WavefrontReader* class.
- We then retrieve a *Mesh* object from the file. Mesh objects contain all information about the object, including its position (inside its Local and World attributes, which are *Physical* objects), color (inside its Material attribute, which are of the *Material* class), and even the vertex data itself (inside its Data attribute, which is a *MeshData* object).
- We'll put the Mesh inside a *Scene* object, which is a container class that holds *Mesh* objects, a *Camera* object, and a `Light` object, along with an RGB background color. Multiple Scenes can be created, even ones that contain the same Meshes, and rendering one vs another one is as simple as changing which Scene is the active one inside the `Window`.
- Finally, we'll put the Scene inside a `Window` object, and render it by calling its `Window.draw()` and `Window.flip()` methods.

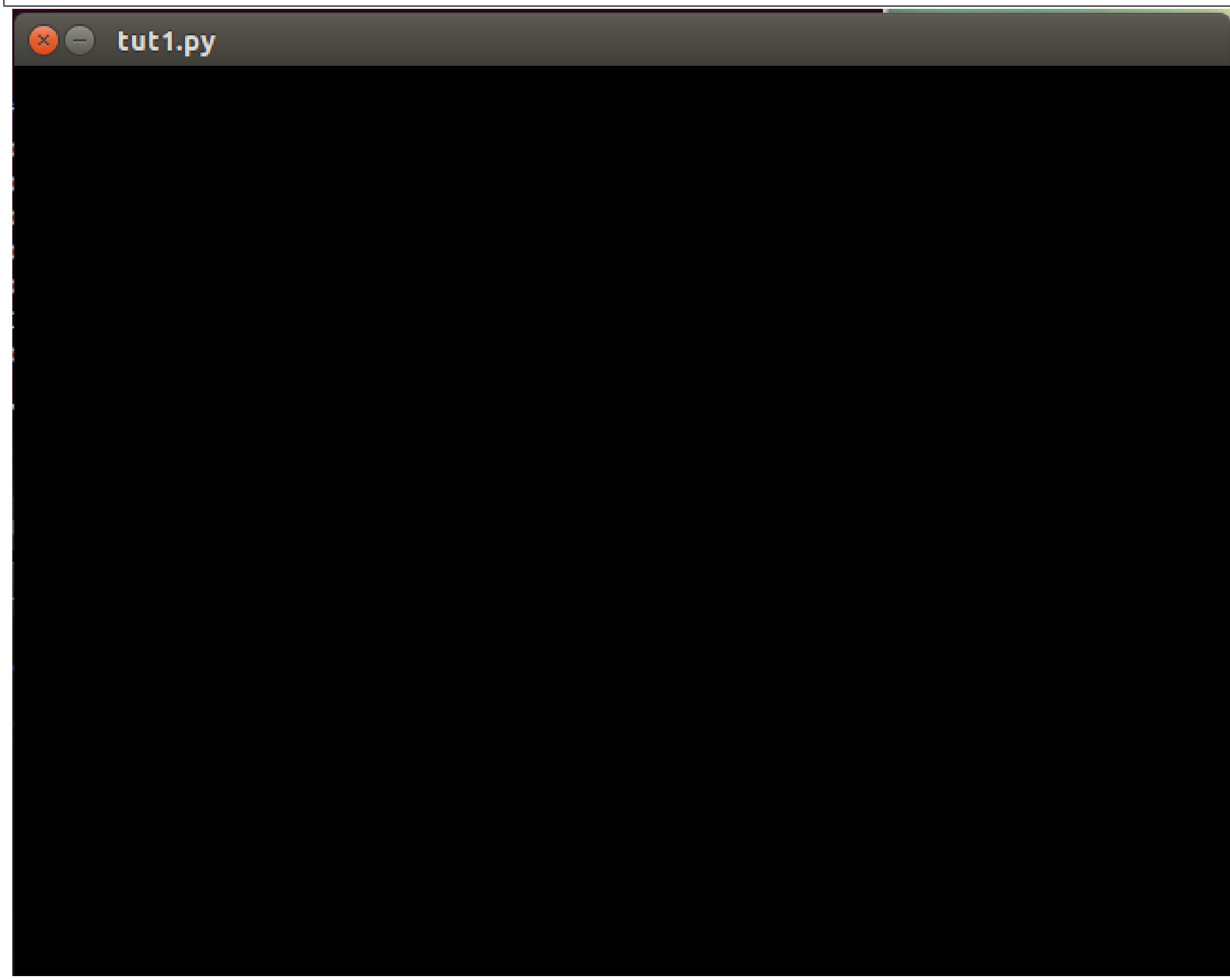
Starting an OpenGL Context and a Window

ratCAVE depends on their already being an OpenGL context set up before loading objects. This can be done by any OpenGL manager (Pyglet and PsychoPy are useful, but PyGame and Qt OpenGL windows should work fine as well). So, before doing anything in ratCAVE, a window must first be created. In these tutorials, I'll show it with Pyglet:

```
import pyglet
import ratcave as rc

window = pyglet.window.Window()
```

If you want to verify that Pyglet is working and the window gets created, just start Pyglet's event loop at the end of your script. This script will create a Pyglet window that closes when you press the escape key:

```
pyglet.app.run()
```



Getting Pyglet Actively Drawing

Pyglet's event loop won't automatically update the scene, so we'll stick in a function that does nothing, that gets called every frame, to guarantee that everything appears onscreen. Anything you want done between frames (updating positions, logging events, etc) can go in this function.:

```
def update(dt):
    pass
pyglet.clock.schedule(update)
```

Reading a Wavefront .obj file

To load a 3D object, let's read in a .obj file! The built-in `WavefrontReader` can read triangulated, uv-maped .obj files exported from Blender. RatCAVE comes with some primitive 3D objects in its `resources` module, so let's use one of those.:

```
# Insert filename into WavefrontReader.
obj_filename = rc.resources.obj_primitives
obj_reader = rc.WavefrontReader(obj_filename)

# Check which meshes can be found inside the Wavefront file, and extract it into a Mesh object for re
print(obj_reader.mesh_names)
>>> ['Torus', 'Sphere', 'Monkey', 'Cube']
```

Loading a Mesh from the WavefrontReader and Positioning it

Loading a mesh can be done through the *WavefrontReader.get_mesh()* method. By default, the mesh will have its position in the same location as in its .obj file, but this can be easily changed. Because the camera is in the -z direction by default per OpenGL convention, let's set it in front of the camera:

```
monkey = obj_reader.get_mesh("Monkey")
monkey.position = 0, 0, -2  # x, y, z
```

Creating a Scene

Scenes consist of meshes, lights, and a camera–everything we need to view and position and object in the real world! Let's put the monkey *Mesh* into a *Scene*:

```
scene = rc.Scene(meshes=[monkey])
```

Drawing the Scene

To draw the scene, simply call the Scene.draw() method in your draw loop! In Pyglet, this looks like this:

```
@window.event
def on_draw():
    scene.draw()

pyglet.app.run()
```

Summary

That's it! Here's the final script, in one place. This script wll be modified in the next tutorial to animate the scene.:

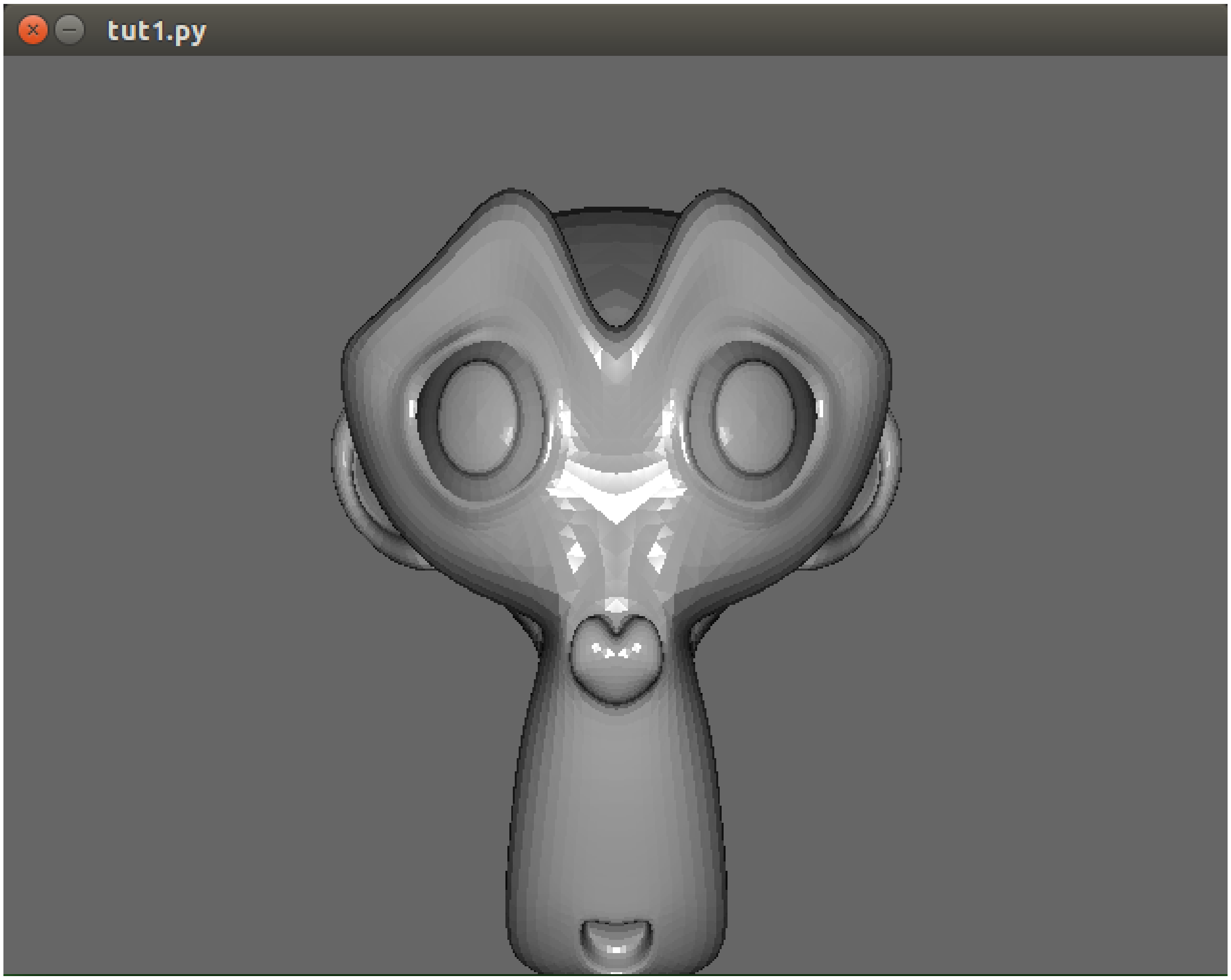
```
import pyglet
import ratcave as rc

# Create Window
```

```
window = pyglet.window.Window()

def update(dt):
    pass
pyglet.clock.schedule(update)

# Insert filename into WavefrontReader.
obj_filename = rc.resources.obj_primitives
obj_reader = rc.WavefrontReader(obj_filename)

# Create Mesh
monkey = obj_reader.get_mesh("Monkey")
monkey.position = 0, 0, -2

# Create Scene
scene = rc.Scene(meshes=[monkey])

@window.event
def on_draw():
    scene.draw()

pyglet.app.run()
```

Version using PsychoPy

Alternatively, you can see the same example using a PsychoPy window:

```
import ratcave as rc
from psychopy import visual, events

# Create Window
window = visual.Window()

# Insert filename into WavefrontReader.
obj_filename = rc.resources.obj_primitives
obj_reader = rc.WavefrontReader(obj_filename)

# Create Mesh
monkey = obj_reader.get_mesh("Monkey")
monkey.position = 0, 0, -2

# Create Scene
scene = rc.Scene(meshes=[monkey])

while 'escape' not in events.getKeys():
    scene.draw()
    window.flip()

window.close()
```



1.5.2 Tutorial 2: Animating a Scene with Multiple Meshes and Moving the Camera with the Keyboard

This tutorial will build on the previous one by adding some more interesting elements. We'll allow the user to move the scene's camera by pressing the left and right arrow keys, and have multiple meshes in the scene that move.

Scenes Hold Lists of Meshes

Let's insert a couple Meshes from our obj_reader WavefrontReader object into the scene!:

```
# Create Meshes from WavefrontReader
monkey = obj_reader.get_mesh("Monkey", position=(0, 0, -1.5))
torus = obj_reader.get_mesh("Torus", position=(-1, 0, -1.5), scale=.2)

# Create Scenes with Meshes.
scene = rc.Scene([monkey, torus])
```

Moving a Mesh

Now, we'll animate the Meshes by changing their rotation attributes in the update function:

```
def rotate_meshes(dt):
    monkey.rot_y += 5 * dt  # dt is the time between frames
    torus.rot_x += 10 * dt
pyglet.clock.schedule(rotate_meshes)
```

Modifying Scene's Background Color

Scenes also have a background color, saved as an RGB tuple in the Scene.bgColor attribute:

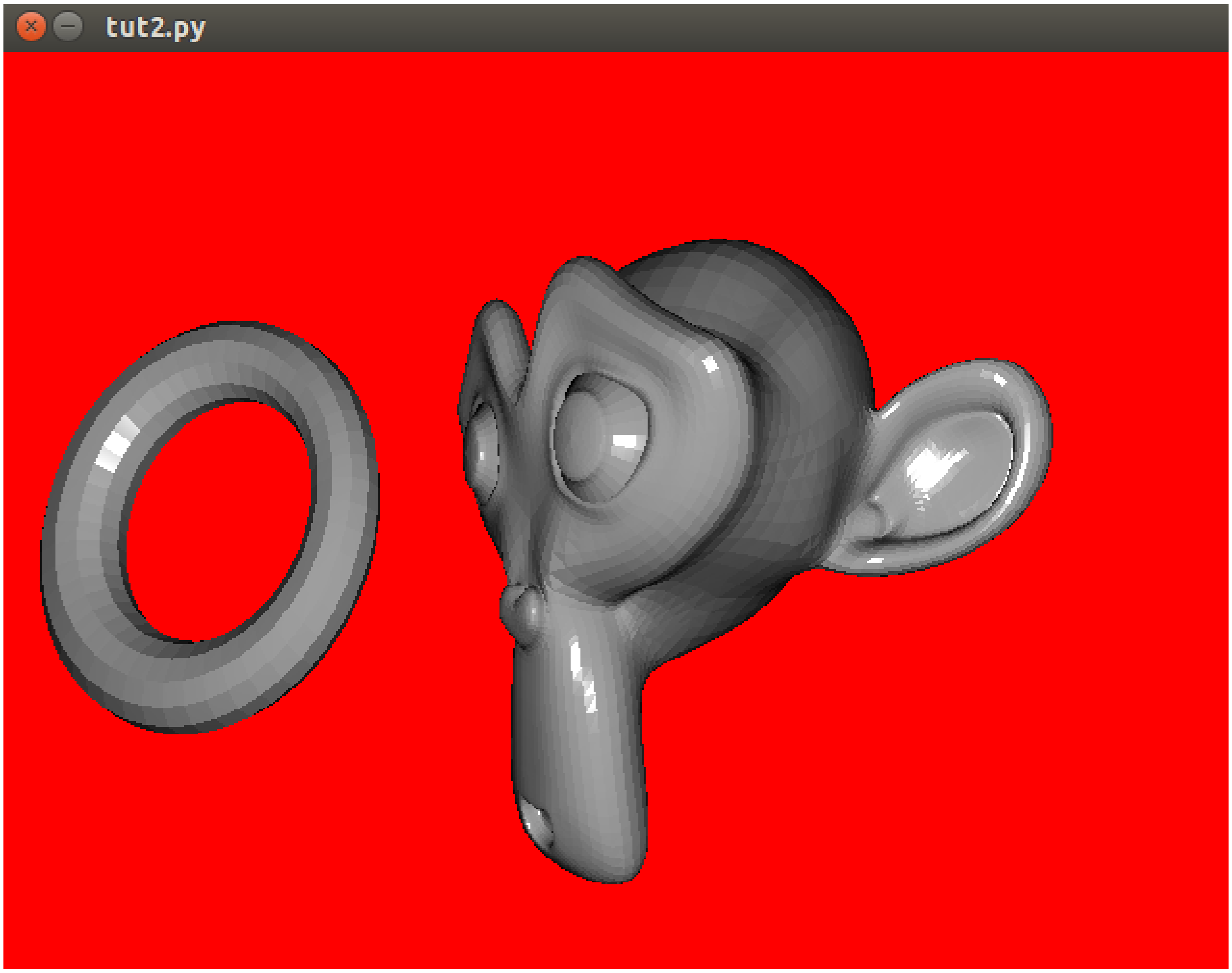
```
scene.bgColor = 1, 0, 0
```

Moving the Camera with the Keyboard

While we could easily make a new `Camera` object from scratch, we'll just grab the scene's camera and have it accept keyboard inputs for movement:

```
# This is how to get keyboard input in pyglet:
from pyglet.window import key
keys = key.KeyStateHandler()
window.push_handlers(keys)

def move_camera(dt):
  camera_speed = 3
  if keys[key.LEFT]:
      scene.camera.x -= camera_speed * dt
  if keys[key.RIGHT]:
      scene.camera.x += camera_speed * dt
pyglet.clock.schedule(move_camera)
```

Now you should have an interactive scene! Don't forget to use the arrow keys to move around!



Summary

Here's the full code for Tutorial 2:

```
import pyglet
from pyglet.window import key
import ratcave as rc

# Create Window and Add Keyboard State Handler to it's Event Loop
window = pyglet.window.Window()
keys = key.KeyStateHandler()
window.push_handlers(keys)

# Insert filename into WavefrontReader.
obj_filename = rc.resources.obj_primitives
obj_reader = rc.WavefrontReader(obj_filename)

# Create Mesh
monkey = obj_reader.get_mesh("Monkey", position=(0, 0, -1.5), scale=.6)
torus = obj_reader.get_mesh("Torus", position=(-1, 0, -1.5), scale=.4)

# Create Scene
scene = rc.Scene(meshes=[monkey, torus])
```

```
scene.bgColor = 1, 0, 0

# Functions to Run in Event Loop
def rotate_meshes(dt):
    monkey.rot_y += 15 * dt  # dt is the time between frames
    torus.rot_x += 80 * dt
pyglet.clock.schedule(rotate_meshes)

def move_camera(dt):
    camera_speed = 3
    if keys[key.LEFT]:
        scene.camera.x -= camera_speed * dt
    if keys[key.RIGHT]:
        scene.camera.x += camera_speed * dt
pyglet.clock.schedule(move_camera)

@window.event
def on_draw():
  scene.draw()

pyglet.app.run()
```

PsychoPy Version

Here's the same scenario, done in PsychoPy:

```
from psychopy import visual, event
import ratcave as rc

camera_speed = 2

# Create Window and Add Keyboard State Handler to it's Event Loop
window = visual.Window()

# Insert filename into WavefrontReader.
obj_filename = rc.resources.obj_primitives
obj_reader = rc.WavefrontReader(obj_filename)

# Create Mesh
monkey = obj_reader.get_mesh("Monkey", position=(0, 0, -1.5), scale=.6)
torus = obj_reader.get_mesh("Torus", position=(-1, 0, -1.5), scale=.4)

# Create Scene
scene = rc.Scene(meshes=[monkey, torus])
scene.bgColor = 1, 0, 0

while True:

    dt = .016

    keys_pressed = event.getKeys()
    if 'escape' in keys_pressed:
        window.close()
        break

    # Move Camera
    for key in keys_pressed:
```

```
        if key == 'left':
            scene.camera.x -= camera_speed * dt
        elif key == 'right':
            scene.camera.x += camera_speed * dt

    # Rotate Meshes
    monkey.rot_y += 15 * dt  # dt is the time between frames
    torus.rot_x += 80 * dt

    # Draw Scene and Flip to Window
    scene.draw()
    window.flip()
```

1.5.3 Tutorial 3: Custom GLSL Shaders, Sending Data to the Graphics Card

To get the most out of our graphics, many newer graphics engines use programs running on the graphics card called "shaders" to specify how objects should be shown on-screen. While teaching GLSL shaders is beyond the scope of this tutorial, and ratCAVE allows you to completely skip writing shaders at all by supplying a few useful ones, you'll likely want to use a shader of your own.

In this tutorial, you'll learn how to use ratCAVE to:

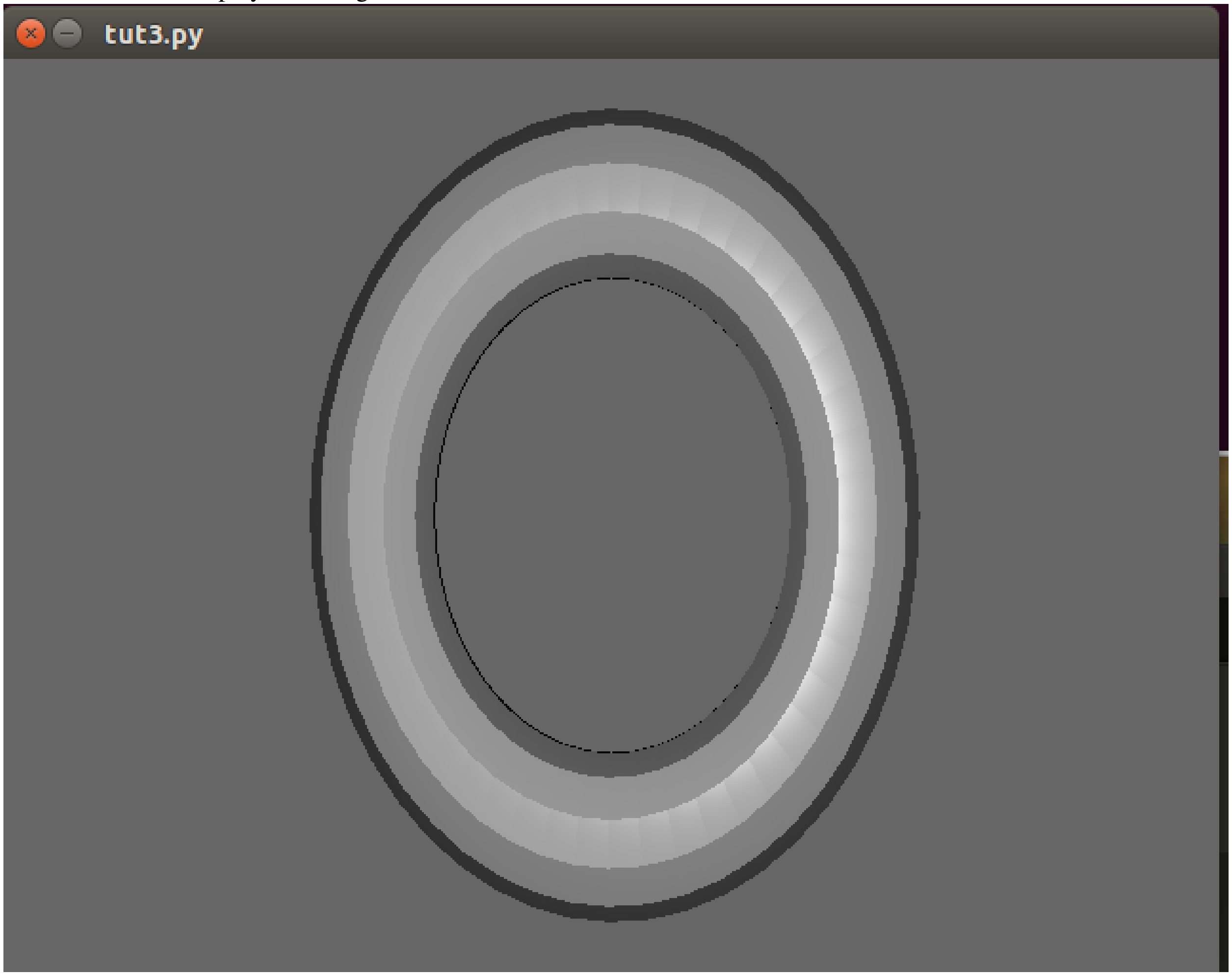
- Compile a *Shader* object and use it in the `Scene.draw()` function.
- Send data to the shader from Python as a *Uniform* variable.

Initial Script

Since the previous tutorials have already covered a lot of ratCAVE methods, let's just start with the following script:

```
import pyglet
import ratcave as rc

# Create window and OpenGL context (always must come first!)
window = pyglet.window.Window()

# Load Meshes and put into a Scene
obj_reader = rc.WavefrontReader(rc.resources.obj_primitives)
torus = obj_reader.get_mesh('Torus', position=(0, 0, -2))

scene = rc.Scene(meshes=[torus])

# Constantly-Running mesh rotation, for fun
def update(dt):
    torus.rot_y += 20. * dt
pyglet.clock.schedule(update)


# Draw Function
@window.event
def on_draw():
    scene.draw()

# Pyglet's event loop run function
pyglet.app.run()
```

This code should display a rotating torus on the window.



Creating a Custom GLSL Shader

Now, one thing ratCAVE does automatically is use it's built-in **genShader** *Shader*, if none is specified. This is to make it easier to get started. Let's replace it with our own custom shader program, which simply positions the mesh in 3D space.

Shader programs come in two types. **Vertex Shaders** tell the graphics card where a vertex will appear on your screen. Our shader here will take data from the meshes, the lights, and the camera to determine where everything goes:

```
vert_shader = """
#version 330

layout(location = 0) in vec3 vertexPosition;
uniform mat4 projection_matrix, view_matrix, model_matrix;
out vec4 vVertex;

void main()
{
    vVertex = model_matrix * vec4(vertexPosition, 1.0);
    gl_Position = projection_matrix * view_matrix * vVertex;
}
"""
```

> **Warning:** This shader requires OpenGL 3.3 drivers to be installed, along with an OpenGL 3.3-compatible graphics card on your system.

The **fragment shader** takes the vertex shader's position data determines what color a pixel on the screen will be. These can get quite complex, but we'll use a fairly simple one here, and just make everything automatically appear red:

```
frag_shader = """
#version 330
out vec4 final_color;
void main()
{
    final_color = vec4(1., 0., 0., 1.);
}
"""
```

> **Note:** Normally, you would just put these shaders in their own files, but here we'll keep everything together and use them as strings.

Now, to make the *Shader*

```
shader = rc.Shader(vert=vert_shader, frag=frag_shader)
```

Using the shader during drawing is done in a shader keyword argument in `Scene.draw()`:

```
scene.draw(shader=shader)
```

Here is what the code should look like now:

```
import pyglet
import ratcave as rc

vert_shader = """
#version 330

layout(location = 0) in vec3 vertexPosition;
uniform mat4 projection_matrix, view_matrix, model_matrix;
out vec4 vVertex;

void main()
{
    vVertex = model_matrix * vec4(vertexPosition, 1.0);
    gl_Position = projection_matrix * view_matrix * vVertex;
}
"""

frag_shader = """
#version 330
out vec4 final_color;
void main()
{
    final_color = vec4(1., 0., 0., 1.);
}
"""

# Create window and OpenGL context (always must come first!)
window = pyglet.window.Window()
```

```
# Load Meshes and put into a Scene
obj_reader = rc.WavefrontReader(rc.resources.obj_primitives)
torus = obj_reader.get_mesh('Torus', position=(0, 0, -2))

scene = rc.Scene(meshes=[torus])

# Constantly-Running mesh rotation, for fun
def update(dt):
    torus.rot_y += 20. * dt
pyglet.clock.schedule(update)

shader = rc.Shader(vert=vert_shader, frag=frag_shader)

# Draw Function
@window.event
def on_draw():
    scene.draw(shader=shader)

# Pyglet's event loop run function
pyglet.app.run()
```

If you run it, you should see a flat red torus!

Sending Data to the Shader using Uniforms

Data can be attached to each object and sent to the shaders, to customize their behavior. Here, let's let the `Mesh.uniforms['diffuse']()` uniform control what color the torus takes.

In the fragment shader, add this line to initialize the **diffuse** uniform variable before the main function:

```
uniform vec3 diffuse;
```

In the python code, modify the diffuse key in the `Mesh.uniforms()` attribute:

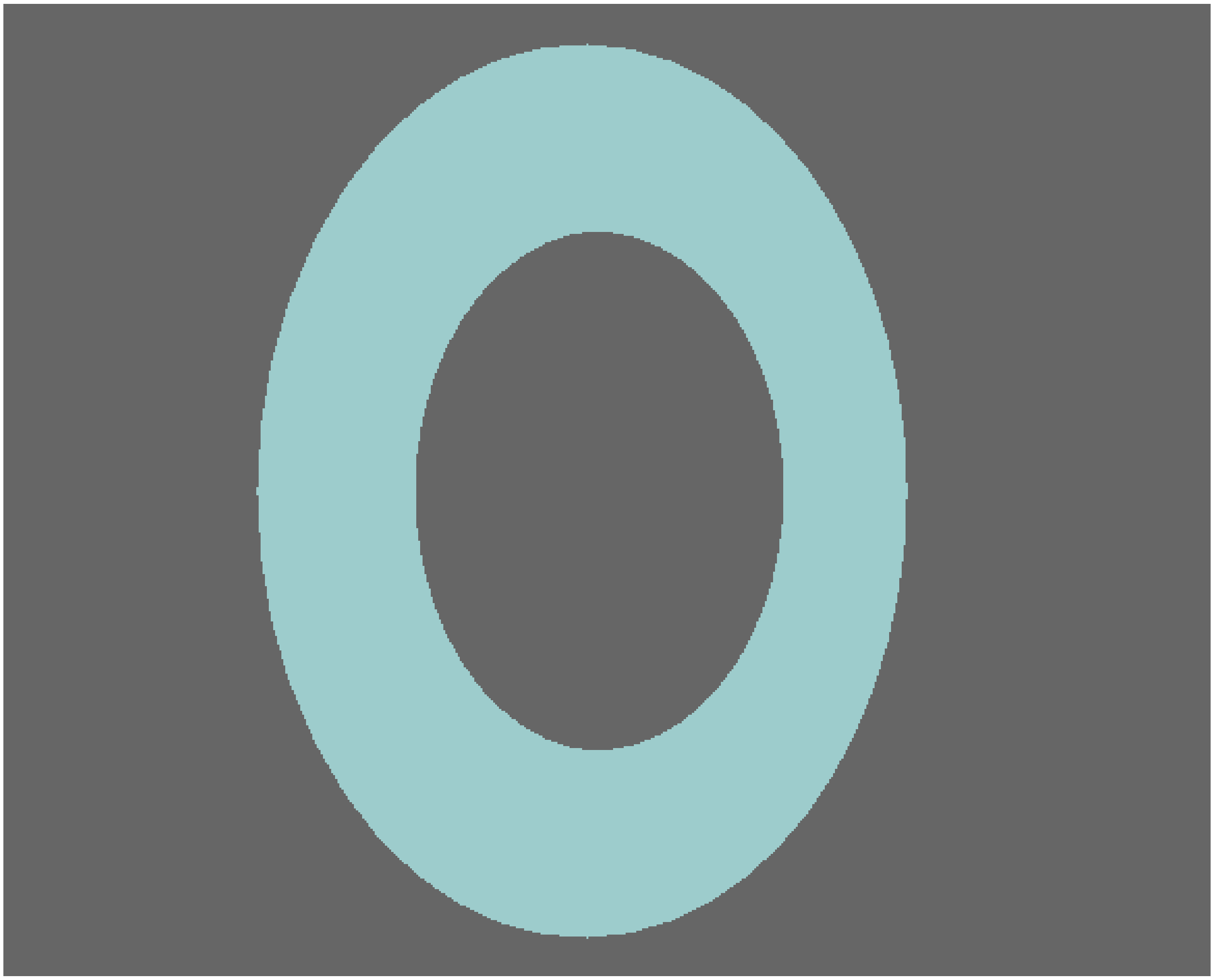
```
torus.uniforms['diffuse'] = [.2, .8, .8]
```

Note: All ratCAVE objects come with some default uniforms, to make setting up easier and to make naming schemas more consistent. This shouldn't restrict you, though–new uniforms are automatically initialized when you add them dictionary-style, like **torus.uniforms['my_uniform'] = 3.0**!

If you run the code now, you should now see a cyan rotating torus. Let's make it a little more dynamic, shall we?

```
import time
import math
def update_color(dt):
    torus.uniforms['diffuse'][0] = 0.5 * math.sin(time.clock()) + 1
pyglet.clock.schedule(update_color)
```

Now the torus will change color!



Summary

Here's the updated code:

```
import pyglet
import ratcave as rc
import time
import math

vert_shader = """
#version 330

layout(location = 0) in vec3 vertexPosition;
uniform mat4 projection_matrix, view_matrix, model_matrix;
out vec4 vVertex;

void main()
{
    vVertex = model_matrix * vec4(vertexPosition, 1.0);
    gl_Position = projection_matrix * view_matrix * vVertex;
}
"""
```

```
frag_shader = """
#version 330
out vec4 final_color;
uniform vec3 diffuse;
void main()
{
    final_color = vec4(diffuse, 1.);
}
"""

# Create window and OpenGL context (always must come first!)
window = pyglet.window.Window()

# Load Meshes and put into a Scene
obj_reader = rc.WavefrontReader(rc.resources.obj_primitives)
torus = obj_reader.get_mesh('Torus', position=(0, 0, -2))
torus.uniforms['diffuse'] = [.2, .8, .8]

scene = rc.Scene(meshes=[torus])

# Constantly-Running mesh rotation, for fun
def update(dt):
    torus.rot_y += 20. * dt
pyglet.clock.schedule(update)

shader = rc.Shader(vert=vert_shader, frag=frag_shader)

def update_color(dt):
    torus.uniforms['diffuse'][0] = 0.5 * math.sin(time.clock()) + 1
pyglet.clock.schedule(update_color)

# Draw Function
@window.event
def on_draw():
    scene.draw(shader=shader)

# Pyglet's event loop run function
pyglet.app.run()
```

In the next tutorial, we'll follow this up by drawing to an FBO dynamically!

1.5.4 Tutorial 4: Using Cubemapping to Render a CAVE VR System

CAVE VR relies on position updates from head trackers to render a virtual scene from the subject's perspective in virtual space

- Two different *Scene* objects are used: - a virtual Scene, which contains the virtual environment to be cubemapped which is rendered from the subject's perspective (meaning, the camera goes where the subject is) - a "real" Scene, which contains just the model (also a *Mesh*) of the screen on which the VR is being projected, seen from the perspective of teh video projector.

While this is difficult to show without having an actual tracking system, we'll illustrate this effect and the code needed to run it by making an animation:

Import Pyglet and Ratcave, and Start the Window and OpenGL Context

At the beginning of the script:

```
import pyglet
import ratcave as rc

window = pyglet.window.Window(resizable=True)
```

At the end of the script:

```
pyglet.app.run()
```

Create the Virtual Scene

Let's say that our virtual scene contains a red sphere and a cyan cube:

```
obj_reader = rc.WavefrontReader(rc.resources.obj_primitives)
sphere = obj_reader.get_mesh("Sphere", position=(0, 0, 2), scale=0.2)
sphere.uniforms['diffuse'] = 1, 0, 0

cube = obj_reader.get_mesh("Cube", position=(0, 0, 0), scale=0.2)
cube.uniforms['diffuse'] = 1, 1, 0

# Put inside a Scene
virtual_scene = rc.Scene(meshes=[sphere, cube])
```

Note that we have one object at the origin (0, 0, 0). Since our light is also at 0,0,0 by default, this may affect how things appear. Let's move the scene's light:

```
virtual_scene.light.position = 0, 3, -1
```

Create the Projected Scene

The Projected Scene is what is actually sent to the display. It will contain the screen (or rodent arena, if you're in a rodent neuroscience lab like us!). Here, let's just use a flat plane to be used as our screen, and use a monkey to show where the subject is looking from (note: the subject isn't necessary for actual VR, it's just used here for illustration of the cubemapping approach).

```
monkey = obj_reader.get_mesh("Monkey", position=(0, 0, -1), scale=0.8)
screen = obj_reader.get_mesh("Plane", position=(0, 0, 1), rotation=(1.5, 180, 0))

projected_scene = rc.Scene(meshes=[monkey, screen], bgColor=(1., 1., 1.))
projected_scene.light.position = virtual_scene.light.position
```

To ensure that the cubemapped texture appears on the screen, the `Mesh.cubemap()` flag needs to be set to True:

```
screen.cubemap = True
```

Setting Your Cameras

A Camera used for Cubemapping

Cubemapping involves rendering an image from six different angles: up, down, left, right, forward, and backward, and stitching each of these six images onto the faces of a cube (for more info, see http://www.nvidia.com/object/cube_map_ogl_tutorial.html). For this algorithm to work, then, two of the *Camera*'s properties must be customized:

- `Camera.aspect()`: The camera's image must be square (meaning it's width-to-height aspect ratio must be 1.0)
- `Camera.fov_y()`: The camera must be able to see 90-degrees, so that the sides all match up.

Altering the camera to be useful for cubemapping is straightforward:

```
cube_camera = rc.Camera(fov_y=90, aspect=1.)
virtual_scene.camera = cube_camera
```

The Projector Camera

In order to do CAVE VR, the camera you use to render the screen must exactly match not only the position and rotation of your video projector relative to the screen, but also the lens characteristics as well. This requires some calibration and measuring on your part, which will differ based on your setup and hardware. Since this is just a demo, let's just arbitrarily place the camera above the scene, looking down:

```
projected_scene.camera = rc.Camera(position=(0, 4, 0), rotation=(-90, 0, 0), z_far=6)
```

The aspect of the camera should, ideally, match that of the window. Let's do that here, using Pyglet's on_resize event handler so that it will happen automatically, even when the screen is resized:

```
@window.event
def on_resize(width, height):
    projected_scene.camera.aspect = width / float(height)
```

Create the OpenGL FrameBuffer and Cube Texture

So far, we've always rendered our Scenes straight to the monitor. However, we can also render to a texture! This lets us do all kinds of image postprocessing effects, but here we'll just use it to update a cube texture, so the screen always has the latest VR image:

```
cube_texture = rc.texture.TextureCube()  # this is the actual cube texture
cube_fbo = rc.FBO(cube_texture)
```

All that's left is to apply the texture the screen:

```
screen.texture = cube_texture
```

> **Warning:** The built-in shader that comes with ratCAVE requires the subject's position to be sent to it throught the **playerPos** uniform. This may be remedied in future releases, or can be changed in your own custom shaders. To do this, use: screen.uniforms['playerPos'] = virtual_scene.camera.position

Move the Subject

Let's have the Monkey move left-to-right, just to illustrate what cubemapping does:

```
import math, time
def update(dt):
    monkey.x = math.sin(.3 * time.clock())
    virtual_scene.camera.position = monkey.position
```

```
    screen.uniforms['playerPos'] = virtual_scene.camera.position
pyglet.clock.schedule(update)
```

Note: The uniforms currently don't update automatically, and should be explicitly changed.

Draw the Scenes

All that's left is for the scenes to be drawn. The virtual_scene should be drawn to the `FBO`, and the projected_scene to the window. To perform the rotations correctly and in the right order, a convenient `Scene.draw360_to_texture()` method has been supplied:

```
@window.event
def on_draw():
  with cube_fbo:
      virtual_scene.draw360_to_texture(cube_texture)
  projected_scene.draw()
```

Summary

Here's the full code:

```
import pyglet
import ratcave as rc
import math, time

window = pyglet.window.Window(resizable=True)


# Assemble the Virtual Scene
obj_reader = rc.WavefrontReader(rc.resources.obj_primitives)
sphere = obj_reader.get_mesh("Sphere", position=(0, 0, 2), scale=0.2)
sphere.uniforms['diffuse'] = 1, 0, 0

cube = obj_reader.get_mesh("Cube", position=(0, 0, 0), scale=0.2)
cube.uniforms['diffuse'] = 1, 1, 0

virtual_scene = rc.Scene(meshes=[sphere, cube])
virtual_scene.light.position = 0, 3, -1

cube_camera = rc.Camera(fov_y=90, aspect=1.)
virtual_scene.camera = cube_camera

# Assemble the Projected Scene
monkey = obj_reader.get_mesh("Monkey", position=(0, 0, -1), scale=0.8)
screen = obj_reader.get_mesh("Plane", position=(0, 0, 1), rotation=(1.5, 180, 0))
screen.cubemap = True

projected_scene = rc.Scene(meshes=[monkey, screen, sphere, cube], bgColor=(1., 1., 1.))
projected_scene.light.position = virtual_scene.light.position
projected_scene.camera = rc.Camera(position=(0, 4, 0), rotation=(-90, 0, 0), z_far=6)

# Create Framebuffer and Textures
cube_texture = rc.texture.TextureCube()  # this is the actual cube texture
cube_fbo = rc.FBO(cube_texture)
```

```
screen.texture = cube_texture


@window.event
def on_resize(width, height):
    projected_scene.camera.aspect = width / float(height)


def update(dt):
    monkey.x = math.sin(.3 * time.clock())
    virtual_scene.camera.position = monkey.position
    screen.uniforms['playerPos'] = virtual_scene.camera.position
pyglet.clock.schedule(update)


@window.event
def on_draw():
    with cube_fbo:
        virtual_scene.draw360_to_texture(cube_texture)
    projected_scene.draw()


pyglet.app.run()
```

PsychoPy Version

Here's the same scenario, done in PsychoPy:

```
from psychopy import visual, event
import ratcave as rc
import math, time


window = visual.Window()

# Assemble the Virtual Scene
obj_reader = rc.WavefrontReader(rc.resources.obj_primitives)
sphere = obj_reader.get_mesh("Sphere", position=(0, 0, 2), scale=0.2)
sphere.uniforms['diffuse'] = 1, 0, 0

cube = obj_reader.get_mesh("Cube", position=(0, 0, 0), scale=0.2)
cube.uniforms['diffuse'] = 1, 1, 0

virtual_scene = rc.Scene(meshes=[sphere, cube])
virtual_scene.light.position = 0, 3, -1

cube_camera = rc.Camera(fov_y=90, aspect=1.)
virtual_scene.camera = cube_camera

# Assemble the Projected Scene
monkey = obj_reader.get_mesh("Monkey", position=(0, 0, -1), scale=0.8)
screen = obj_reader.get_mesh("Plane", position=(0, 0, 1), rotation=(1.5, 180, 0))
screen.cubemap = True

projected_scene = rc.Scene(meshes=[monkey, screen, sphere, cube], bgColor=(1., 1., 1.))
projected_scene.light.position = virtual_scene.light.position
projected_scene.camera = rc.Camera(position=(0, 4, 0), rotation=(-90, 0, 0), z_far=6)
```

```
# Create Framebuffer and Textures
cube_texture = rc.texture.TextureCube()  # this is the actual cube texture
cube_fbo = rc.FBO(cube_texture)
screen.texture = cube_texture

# Main Loop
while True:

    if 'escape' in event.getKeys():
        window.close()
        break

    monkey.x = math.sin(.3 * time.clock())
    virtual_scene.camera.position = monkey.position
    screen.uniforms['playerPos'] = virtual_scene.camera.position

    with cube_fbo:
        virtual_scene.draw360_to_texture(cube_texture)
    projected_scene.draw()
    window.flip()
```

1.6 CAVE Virtual Reality

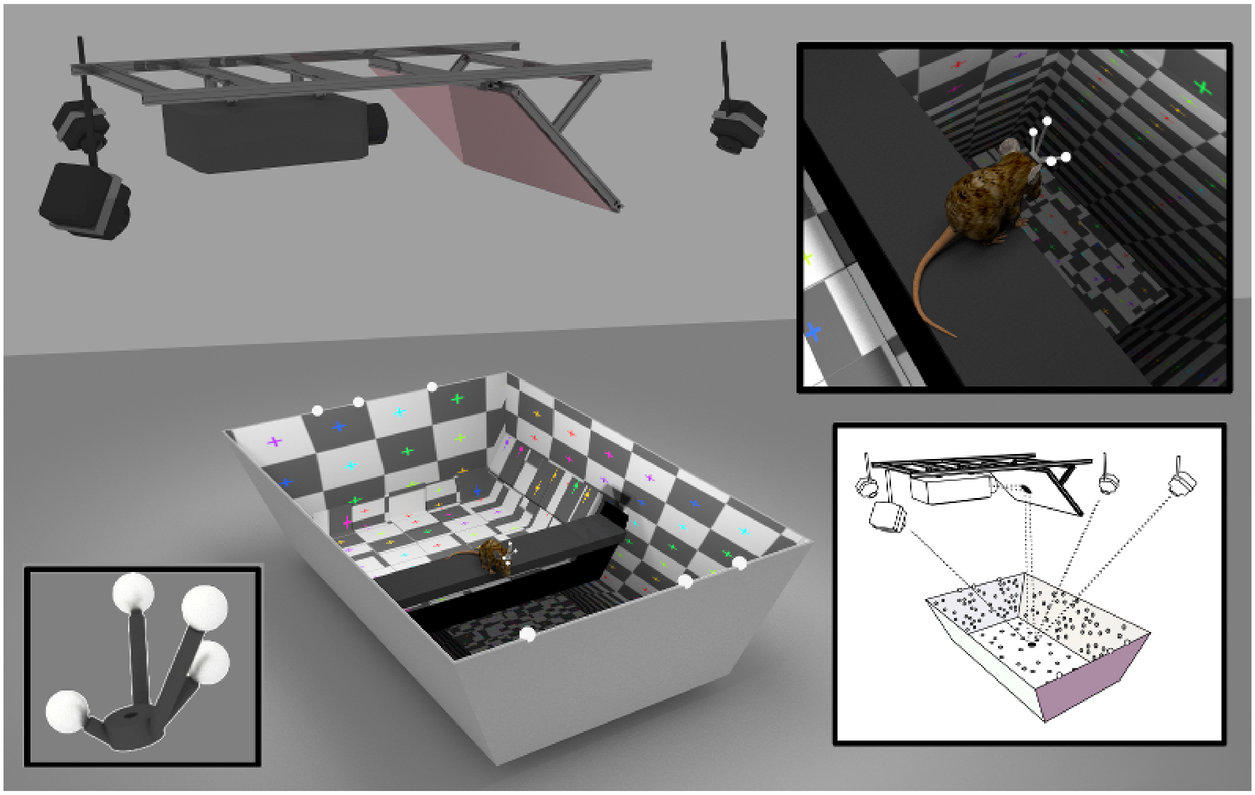
Building your own virtual reality setup requires several parts:

- A 3D Graphics Engine (to render the virtual environment)
- Video Displays (to show the environment to the user). Ideally, these should be large enough to allow the user to see a large amount of the virtual environment! This means you'll either want something that is: - Head-fixed (the display is attached ot the user's head, so that they can always see a virtual space, no matter where they turn) - Projected on surfaces all around the user. If there are screens in a 360-degree arc around the user, you get a CAVE system!
- A head tracking system, to update the virtual environment when the user moves their head. This is one of the things that gives such a strong immersive sense to virtual reality.

1.6.1 Our Setup

The VR setup we made is intended for use by rats, mice, and gerbils for cognitive neuroscience experiments, which is why we call this python package ratCAVE! In selecting our components, we were both limited by and helped by the small sizes of our users:

- We use a simplified 3D graphics engine, to make our experiment scripts small and simple to deploy (python, with pyglet or psychopy + ratCAVE)
- Head-mounting a display on mice wasn't an option because of their small size, so we use a single wide-lens video projector, which front-projects onto the walls and floor of the rodent's arena. This gives them a 360-degree view of the virtual environment, while reducing the computational demands of our setup.
- We use an optical tracking system for measuring the rodent's head space via data collected from camera arrays. We use _NaturalPoint's Optitrack System: http://www.optitrack.com/ - To control this tracking system from Python, we wrote a Python package called MotivePy, available here: https://github.com/neuroneuro15/motivepy - To access the data from the tracking system on a remote client in our experiment scripts, we wrote another Python client called NatNetClient, available here: https://github.com/neuroneuro15/natnetclient



1.6.2 Example VR Experiment Script

Writing a VR Script is relatively straightforward, and involves three main components:

1. Connect to your tracking system
2. Render your 3D environment
3. In a loop, re-render the 3D environment, setting the camera position at the new eye position of your user.

In Pyglet, RatCAVE, and NatNetClient, this would something like this:

```
import pyglet
import ratcave as rc
import natnetclient

# Connect to Tracking System
client = natnetclient.NatClient(ip='197.0.0.10', port=5023)
user = client.rigid_bodies['User']

# Create Scene and put in a draw loop
window = pyglet.window.Window()
reader = rc.WavefrontReader(rc.resources.obj_primitives)
scene = rc.scene(meshes=[reader.get_mesh('Sphere', position=(0, 1, -2))

@window.event()
def on_draw():
    scene.draw()

# Update camera position, based on user's position
def update(dt):
```

```
    scene.camera.position = user.position
pyglet.clock.schedule()

# Start App
pyglet.app.run()
```

In Psychopy, which is written in a more imperative format, it looks like this:

```
from psychopy import visual, event
import ratcave as rc
import natnetclient

# Connect to Tracking System
client = natnetclient.NatClient(ip='197.0.0.10', port=5023)
user = client.rigid_bodies['User']

window = visual.Window()

# Main Loop
while 'escape' not in event.getKeys():

    # Create Scene
    reader = rc.WavefrontReader(rc.resources.obj_primitives)
    scene = rc.scene(meshes=[reader.get_mesh('Sphere', position=(0, 1, -2))

    # Update Camera position, based on user's position
    scene.camera.position = user.position

    # Draw
    scene.draw()
    window.flip()
```

1.6.3 Modular Nature of VR

At it's core, VR does not stand for any one technology. Whether you are using head-mounted screens or projector, optical camera array tracking systems or treadmills, or any sort of graphics engine out there, the key is that you are changing the camera position on a loop, based on some user movement. Hopefully, this description has given you some ideas for how you can set up the your own VR system!

1.7 API Documentation

1.7.1 Module contents

class `Camera` (*fov_y=60.0*, *aspect=1.778*, *z_near=0.01*, *z_far=4.5*, *x_shift=0.0*, *y_shift=0.0*, *ortho_mode=False*, ***kwargs*)

Bases: `ratcave.mixins.PhysicalNode`, `ratcave.mixins.Picklable`

A convenient object for controlling the scene viewing angle.

Returns a Camera instance, which determines the world-to-screen perspective transformation.

Parameters

- **fov_y** (*float*) – vertical field of view (degrees)
- **aspect** (*float*) – screen width/height

- **z_near** (*float*) – near clipping distance
- **z_far** (*float*) – far clipping distance
- **x_shift** (*float*) – horizontal lens shift
- **y_shift** (*float*) – vertical lens shift
- **ortho_mode** (*bool*) – Whether to use orthographic projection instead of perpective projection.

Returns Camera instance

reset_aspect()
Gets the viewport size, and matches the camera aspect ratio to it.

update()

class **Mesh**(*name*, *meshdata*, *uniforms=UniformCollection: {}*, *drawstyle='fill'*, *visible=True*, *point_size=4*, *texture=None*, ***kwargs*)
Bases: *ratcave.mesh.EmptyMesh*, *ratcave.mixins.Picklable*

Returns a Mesh object, containing the position, rotation, and color info of an OpenGL Mesh.

Meshes have two coordinate system, the "local" and "world" systems, on which the transforms are performed sequentially. This allows them to be placed in the scene while maintaining a relative position to one another.

Note: Meshes are not usually instantiated directly, but from a 3D file, like the WavefrontReader .obj and .mtl files.

Parameters

- **name** (*str*) – the mesh's name.
- **vertices** – the Nx3 vertex coordinate data
- **normals** – the Nx3 normal coordinate data
- **texcoords** – the Nx2 texture coordinate data
- **uniforms** (*list*) – a list of all Uniform objects
- **drawstyle** (*str*) – 'point': only vertices, 'line': points and edges, 'fill': points, edges, and faces (full)
- **visible** (*bool*) – whether the Mesh is available to be rendered. To make hidden (invisible), set to False.
- **point_size** (*int*) – How big to draw the points, when drawstyle is 'point'

Returns Mesh instance

drawstyle = {'line': <Mock id='140562677782864'>, 'point': <Mock id='140562677782992'>, 'fill': <Mock id='140562

update()

class **MeshData**(*vertices*, *face_indices*, *normals*, *texcoords=None*)
Bases: object

Collects all vertex data for rendering in 3D graphics packages.

Parameters

- **vertices** (*list*) – Nx3 vertex array

- **face_indices** (*list*) – Nx3 Face index array (0-indexed)
- **normals** (*list*) – Nx3 normals array
- **texture_uv** (*list*) – Nx2 texture_uv array

Returns MeshData object

draw(*mode*)

load()

reindex()

class Material(*diffuse=[0.8, 0.8, 0.8], spec_weight=0.0, specular=[0.0, 0.0, 0.0], ambient=[0.0, 0.0, 0.0], opacity=1.0, flat_shading=False, texture_file=None*)
Bases: `object`

class Physical(*position=(0.0, 0.0, 0.0), rotation=(0.0, 0.0, 0.0), scale=1.0, *args, **kwargs*)
Bases: `object`

XYZ Position, Scale and XYZEuler Rotation Class.

Parameters

- **position** (*list*) – (x, y, z) translation values.
- **rotation** (*list*) – (x, y, z) rotation values
- **scale** (*float*) – uniform scale factor. 1 = no scaling.

has_changed()

on_change()
This method fires when object position or geometry changes. Can be overwritten by parent classes to add more actions.

position
xyz local position

rotation
XYZ Euler rotation, in degrees

update()
Calculate model, normal, and view matrices from position, rotation, and scale data.

update_model_and_normal_matrix()

update_model_matrix()

update_view_matrix()

class Scene(*meshes=(), camera=None, light=None, bgColor=(0.4, 0.4, 0.4)*)
Bases: `object`

Returns a Scene object. Scenes manage rendering of Meshes, Lights, and Cameras.

clear()
Clear Screen and Apply Background Color

draw(*shader=<ratcave.shader.Shader object>, clear=True, send_mesh_uniforms=True, send_camera_uniforms=True, send_light_uniforms=True, userdata={}, gl_states=(<Mock id='140562678429200'>, <Mock id='140562678429072'>, <Mock id='140562678428624'>, <Mock id='140562678427984'>)*)
Draw each visible mesh in the scene from the perspective of the scene's camera and lit by its light.

draw360_to_texture (*cubetexture, shader=<ratcave.shader.Shader object>, autoclear=True, userdata={}, gl_states=(<Mock id='140562678429008'>, <Mock id='140562678428688'>, <Mock id='140562678428560'>, <Mock id='140562677866768'>)*)
Draw each visible mesh in the scene from the perspective of the scene's camera and lit by its light, and applies it to each face of cubetexture, which should be currently bound to an FBO.

class **WavefrontReader** (*file_name*)
Bases: `object`

Reads Wavefront (.obj) files created in Blender to build ratCAVE.graphics Mesh objects.

Parameters **file_name** (`str`) – .obj file to read (assumes an accompanying .mtl file has the same base file name.)

Returns

Return type *WavefrontReader*

get_mesh (*mesh_name, **kwargs*)
Returns a Mesh object for directly rendering in a scene.

Parameters

- **mesh_name** –
- **kwargs** – All of Mesh's keword arguments will be applied to the Mesh, for convenient Mesh creation.

Returns Mesh

Return type *Mesh*

get_meshes (*mesh_names, **kwargs*)
Returns a dictionary of meshes, with kwargs applied to all meshes identically, as in get_mesh()

get_scene (*include=[], exclude=[]*)
Return a Scene object containing the Meshes in the file.

Parameters

- **include** (`list`) – mesh names to only include.
- **exclude** (`list`) – mesh names to exclude.

Returns `Scene`

1.7.2 Submodules

1.7.3 ratcave.mesh module

mesh

This module contains the Mesh, MeshData, and Material classes. This documentation was auto-generated from the mesh.py file.

class **EmptyMesh** (**args, **kwargs*)
Bases: `ratcave.mixins.PhysicalNode`

class **Material** (*diffuse=[0.8, 0.8, 0.8], spec_weight=0.0, specular=[0.0, 0.0, 0.0], ambient=[0.0, 0.0, 0.0], opacity=1.0, flat_shading=False, texture_file=None*)
Bases: `object`

class **Mesh** (*name*, *meshdata*, *uniforms=UniformCollection: {}*, *drawstyle='fill'*, *visible=True*, *point_size=4*, *texture=None*, ***kwargs*)

Bases: *ratcave.mesh.EmptyMesh*, *ratcave.mixins.Picklable*

Returns a Mesh object, containing the position, rotation, and color info of an OpenGL Mesh.

Meshes have two coordinate system, the "local" and "world" systems, on which the transforms are performed sequentially. This allows them to be placed in the scene while maintaining a relative position to one another.

Note: Meshes are not usually instantiated directly, but from a 3D file, like the WavefrontReader .obj and .mtl files.

Parameters

- **name** (*str*) – the mesh's name.
- **vertices** – the Nx3 vertex coordinate data
- **normals** – the Nx3 normal coordinate data
- **texcoords** – the Nx2 texture coordinate data
- **uniforms** (*list*) – a list of all Uniform objects
- **drawstyle** (*str*) – 'point': only vertices, 'line': points and edges, 'fill': points, edges, and faces (full)
- **visible** (*bool*) – whether the Mesh is available to be rendered. To make hidden (invisible), set to False.
- **point_size** (*int*) – How big to draw the points, when drawstyle is 'point'

Returns Mesh instance

drawstyle = {'line': <Mock id='140562677782864'>, 'point': <Mock id='140562677782992'>, 'fill': <Mock id='1405626

texture = None
Pyglet texture object for mapping an image file to the vertices (set using Mesh.load_texture())

uniforms = None
Physical, World Mesh coordinates Local Mesh coordinates (Physical type)

update ()

visible = None
Bool – if the Mesh is visible for rendering. If false, will not be rendered.

class **MeshData** (*vertices*, *face_indices*, *normals*, *texcoords=None*)

Bases: object

Collects all vertex data for rendering in 3D graphics packages.

Parameters

- **vertices** (*list*) – Nx3 vertex array
- **face_indices** (*list*) – Nx3 Face index array (0-indexed)
- **normals** (*list*) – Nx3 normals array
- **texture_uv** (*list*) – Nx2 texture_uv array

Returns MeshData object

draw (*mode*)

load()

reindex()

class **MeshLoader**(*name*, *meshdata*, *material=None*)

Bases: `object`

Creates various types of Meshes from MeshData and Material objects.

load_mesh(***kwargs*)

1.7.4 ratcave.mixins module

class **Physical**(*position=(0.0, 0.0, 0.0)*, *rotation=(0.0, 0.0, 0.0)*, *scale=1.0*, **args*, ***kwargs*)

Bases: `object`

XYZ Position, Scale and XYZEuler Rotation Class.

Parameters

- **position** (*list*) – (x, y, z) translation values.
- **rotation** (*list*) – (x, y, z) rotation values
- **scale** (*float*) – uniform scale factor. 1 = no scaling.

has_changed()

on_change()

This method fires when object position or geometry changes. Can be overwritten by parent classes to add more actions.

position

xyz local position

rotation

XYZ Euler rotation, in degrees

update()

Calculate model, normal, and view matrices from position, rotation, and scale data.

update_model_and_normal_matrix()

update_model_matrix()

update_view_matrix()

class **PhysicalNode**(**args*, ***kwargs*)

Bases: `ratcave.mixins.Physical`, `ratcave.mixins.SceneNode`

Object with xyz position and rotation properties that are relative to its parent.

position_global

update()

class **Picklable**

Bases: `object`

classmethod **load**(*filename*)

Load the object from a pickle file.

save(*filename*)

Save the object to a file. Will be Pickled in the process, but can be loaded easily with Class.load()

class SceneNode(*parent=None*, *children=None*)
Bases: `object`

A Node of the Scenegraph. Has children, but no parent.

add_children(*children=[]*)
Adds a list of objects as children in the scene graph.

children

parent
A SceneNode object that is this object's parent in the scene graph.

remove_children(*children*)

1.7.5 ratcave.resources module

gen_fullscreen_quad()

1.7.6 ratcave.scene module

class Scene(*meshes=()*, *camera=None*, *light=None*, *bgColor=(0.4, 0.4, 0.4)*)
Bases: `object`

Returns a Scene object. Scenes manage rendering of Meshes, Lights, and Cameras.

clear()
Clear Screen and Apply Background Color

draw(*shader=<ratcave.shader.Shader object>*, *clear=True*, *send_mesh_uniforms=True*, *send_camera_uniforms=True*, *send_light_uniforms=True*, *userdata={}*, *gl_states=(<Mock id='140562678429200'>, <Mock id='140562678429072'>, <Mock id='140562678428624'>, <Mock id='140562678427984'>)*)
Draw each visible mesh in the scene from the perspective of the scene's camera and lit by its light.

draw360_to_texture(*cubetexture*, *shader=<ratcave.shader.Shader object>*, *autoclear=True*, *userdata={}*, *gl_states=(<Mock id='140562678429008'>, <Mock id='140562678428688'>, <Mock id='140562678428560'>, <Mock id='140562677866768'>)*)
Draw each visible mesh in the scene from the perspective of the scene's camera and lit by its light, and applies it to each face of cubetexture, which should be currently bound to an FBO.

1.7.7 ratcave.shader module

class Shader(*vert=''*, *frag=''*, *geom=''*)
Bases: `ratcave.utils.gl.BindingContextMixin`, `ratcave.utils.gl.BindNoTargetMixin`

GLSL Shader program object for rendering in OpenGL.

Parameters

- **vert** (*–*) – The vertex shader program string
- **frag** (*–*) – The fragment shader program string
- **geom** (*–*) – The geometry shader program

bindfun

createShader(*strings*, *shadertype*)

get_uniform_location (*name*)

link ()
link the program

uniform_matrixf (*name*, *mat*, *loc=None*)
Send 4x4 NumPy matrix data as a uniform to the shader, named 'name'. Shader must be already bound.

uniformf (*name*, **vals*)
Send data as a float uniform, named 'name'. Shader must be already bound.

uniformf_funs = (<Mock id='140562678227152'>, <Mock id='140562678227280'>, <Mock id='140562678227408'>, <M

uniformi (*name*, **vals*)
Send data as an integer uniform, named 'name'. Shader must be already bound.

uniformi_funs = (<Mock id='140562678227664'>, <Mock id='140562678227792'>, <Mock id='140562678227920'>, <M

class Uniform (*name*, **vals*)
Bases: `object`

A fixed-length, fixed-type array with a pre-assigned glUniform function for quick shader data sending.

classmethod from_dict (*data_dict*)
A factory function that can build multiple uniforms from a name: val dictionary

send_to (*shader*)
Sends uniform to a currently-bound shader, returning its location (-1 means not sent)

value

class UniformCollection (*uniform_dict={}*)
Bases: `object`

send_to (*shader*)

1.7.8 ratcave.texture module

class BaseTexture
Bases: `object`

cube_name = 'CubeMap'

int_flag = 0

tex_name = 'TextureMap'

class DepthTexture (*id=None*, *width=1024*, *height=1024*, *data=None*)
Bases: `ratcave.texture.Texture`

2D Color Texture class. Width and height can be set, and will generate a new OpenGL texture if no id is given.

attachment_point

internal_fmt

pixel_fmt

class GrayscaleTexture (*id=None*, *width=1024*, *height=1024*, *data=None*)
Bases: `ratcave.texture.Texture`

2D Color Texture class. Width and height can be set, and will generate a new OpenGL texture if no id is given.

internal_fmt

pixel_fmt

class GrayscaleTextureCube (**args, **kwargs*)
Bases: `ratcave.texture.TextureCube`

the Color Cube Texture class.

internal_fmt

pixel_fmt

class RenderBuffer (*width, height*)
Bases: `ratcave.utils.gl.BindingContextMixin`, `ratcave.utils.gl.BindTargetMixin`

attach_to_fbo ()

attachment_point

bindfun

internal_fmt

target

class Texture (*id=None, width=1024, height=1024, data=None*)
Bases: `ratcave.texture.BaseTexture`, `ratcave.utils.gl.BindTargetMixin`

2D Color Texture class. Width and height can be set, and will generate a new OpenGL texture if no id is given.

attach_to_fbo ()
Attach the texture to a bound FBO object, for rendering to texture.

attachment_point

bindfun

classmethod from_image (*img_filename, **kwargs*)
Uses Pyglet's image.load function to generate a Texture from an image file.

int_flag = 1

internal_fmt

pixel_fmt

slot
The texture's ActiveTexture slot.

target

target0

class TextureCube (**args, **kwargs*)
Bases: `ratcave.texture.Texture`

the Color Cube Texture class.

attach_to_fbo (*face=0*)

classmethod from_image (*img_filename*)

int_flag = 2

target

target0

1.7.9 ratcave.wavefront module

class WavefrontReader (*file_name*)

Bases: `object`

Reads Wavefront (.obj) files created in Blender to build ratCAVE.graphics Mesh objects.

Parameters **file_name** (*str*) – .obj file to read (assumes an accompanying .mtl file has the same base file name.)

Returns

Return type *WavefrontReader*

get_mesh (*mesh_name*, ***kwargs*)

Returns a Mesh object for directly rendering in a scene.

Parameters

- **mesh_name** –
- **kwargs** – All of Mesh's keword arguments will be applied to the Mesh, for convenient Mesh creation.

Returns Mesh

Return type *Mesh*

get_meshes (*mesh_names*, ***kwargs*)

Returns a dictionary of meshes, with kwargs applied to all meshes identically, as in get_mesh()

get_scene (*include=[]*, *exclude=[]*)

Return a Scene object containing the Meshes in the file.

Parameters

- **include** (*list*) – mesh names to only include.
- **exclude** (*list*) – mesh names to exclude.

Returns *Scene*

- genindex

Python Module Index

r

Index

A

B

C

D

E

F

G

H

I

L

M

O

P

R

S

T

U

V

W