



Ratcave: A 3D graphics python package for cognitive psychology experiments

Nicholas A. Del Grosso¹ · Anton Sirota¹

Published online: 6 May 2019
© The Author(s) 2019

Abstract

We present here a free, open source Python 3D graphics library called Ratcave that extends existing Python psychology stimulus software by allowing scientists to load, display, and transform 3D stimuli created in 3D modeling software. This library makes 3D programming intuitive to new users by providing 3D graphics engine concepts (Mesh, Scene, Light, and Camera classes) that can be manipulated using an interface similar to existing 2D stimulus libraries. In addition, the use of modern OpenGL constructs by Ratcave helps scientists create fast, hardware-accelerated dynamic stimuli using the same intuitive high-level, lightweight interface. Because Ratcave supplements, rather than replaces, existing Python stimulus libraries, scientists can continue to use their preferred libraries by simply adding Ratcave graphics to their existing experiments. We hope this tool will be useful both as a stimulus library and as an example of how tightly-focused libraries can add quality to the existing scientific open-source software ecosystem.

Keywords 3D graphics · Python · Stimulus software · Vision · 3D

Cognitive psychology and neuroscience experiments use software that presents stimuli to a subject, detects subject responses, and logs events for future analysis, all with high temporal accuracy. An ever-expanding list of other features included in this software are compatibility with third-party hardware devices (e.g. button boxes, amplifiers, eye tracking systems), support for custom experimental designs, and online analysis for adaptive stimulus sequences; these tools are available both as self-enclosed software solutions (e.g. Neurobs Presentation, BC12000, SuperLab, E-Prime) and open-source libraries (e.g. Psychtoolbox by Brainard, 1997; PsychoPy by Peirce, 2007; VisionEgg by Straw, 2008; Expyriment by Krause & Lindemann, 2013; for a review of psychophysics libraries, see Kötter, 2009). However, these popular libraries are missing 3D graphics support, needed for a wide range visual psychophysics

experiments, such as 3D mental rotation or object recognition, virtual reality in spatial navigation research, to name a few. While 3D graphics libraries do exist in Python (e.g. Panda3D, PyOgre, Vizard) and other languages (e.g. Unity3D, Unreal Engine), the stimuli, logging, and hardware support of all of these libraries are designed to work with the windows and event loops they supply, making it difficult to integrate 3D graphics functionality into different psychophysics libraries without (sometimes-extensive) modification (e.g. to mix PsychoPy's DotStim and Expyriment's video support). In practice, this means that each software suite is relatively self-contained; researchers who require 3D stimuli, for example, have to, thereby, resort to use or develop different experiment control software when employing 3D visual stimuli (essentially, building interface to 3D game engines), losing out on the rich features that exist in the psychophysics software ecosystem developed for the 2D graphics. Extension libraries help reduce these feature-tradeoff decisions; for example, OpenSesame, a Python-powered GUI (Mathôt & Theeuwes, 2012), uses PsychoPy, Expyriment, and PyGame as “backends” to its experiment-building graphical interface, thereby supporting all researchers who rely on those libraries. A similar extension approach could be used for 3D stimuli—not to compete with the existing 3D frameworks on a feature-by-feature basis, but to simply add simple-to-use 3D stimulus presentation and manipulation support to the feature list of existing 2D stimulus libraries in Python.

Electronic supplementary material The online version of this article (<https://doi.org/10.3758/s13428-019-01245-x>) contains supplementary material, which is available to authorized users.

✉ Anton Sirota
sirota@biologie.uni-muenchen.de

¹ Bernstein Centre for Computational Neuroscience, Graduate School of Systemic Neurosciences, Faculty of Medicine, Ludwig-Maximilians-Universität München, Großhaderner Straße 2, 82152 Planegg, Germany

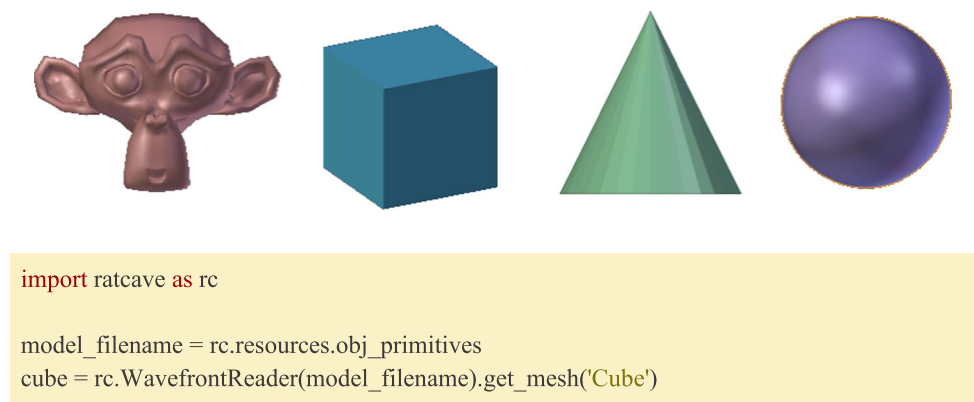


Fig. 1 (Top): Some of the 3D Mesh primitives supplied with Ratcave. (Bottom): Importing Ratcave into the Python environment and creating a cube stimulus from the Ratcave’s supplied primitive meshes

In this paper, we present an open-source, cross-platform Python library called Ratcave that adds 3D stimulus support to all OpenGL-based 2D Python stimulus libraries, including VisionEgg, Psychopy, Pyglet, and PyGame. We review the core features of Ratcave (<https://github.com/ratcave/ratcave>) and highlight key connections of its interface to underlying graphics programming strategies (a thorough manual, complete with API guide and tutorials for first-time users can be found at <https://ratcave.readthedocs.org>). This library, which derives its name from our high-speed RatcaveVR experimental setup (Del Grosso, Graboski, Chen, Hernández, & Sirota, 2017), is designed to increase accessibility of 3D graphics programming to the existing ecosystem of psychology software for Python.

Software description

Built-in primitives and graphics resources

In order to make 3D programming accessible, Ratcave comes with a collection of resources, including basic 3D object primitives (Fig. 1) and a wide range of 3D lighting effects (Fig. 2,

Supplementary Video 1). This way, a user can get started quickly, writing customized code only when needed.

Creating Meshes, a term used for any 3D object in Ratcave, is done either by supplying the vertex coordinates as an array or by importing from a 3d-formatted file (for example, the popular Wavefront file format, for which Ratcave provides a parser). All objects in a Ratcave Scene (Meshes, Lights, and Cameras) can be repositioned, rotated, and scaled using an intuitive object-oriented interface (Fig. 3).

Rendering 3D Meshes in Ratcave

Once a Mesh is loaded and positioned, it can be drawn in any active OpenGL window (e.g. a Psychopy window, Pyglet window, Vision Egg window, etc) by binding it to a Shader program using Python’s *with* keyword and calling its *draw()* method. Ratcave provides a default shader that performs many industry-standard 3D transformation and lighting steps (including diffuse and specular lighting, and shadow-mapping, Fig. 2), allowing users to create and use arbitrary 3D stimuli in syntactically the same way as they would use 2D stimuli (Fig. 4).

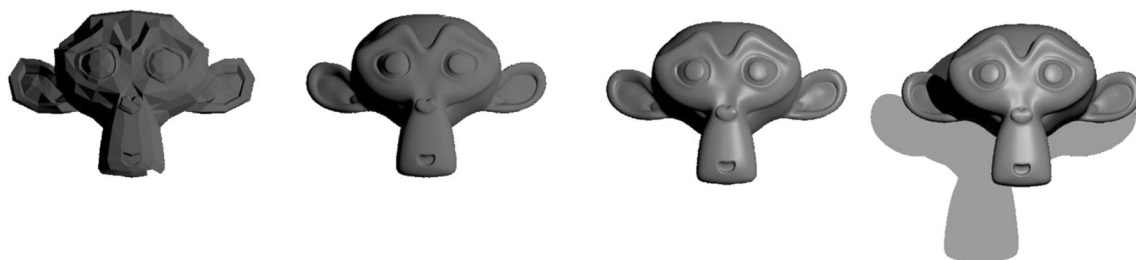


Fig. 2 Examples of some simple 3D lighting effects available in Ratcave: diffuse and “glossy” specular reflections, ambient lighting, and shadows

```

cube.position.x = 5
cube.rotation.xyz = 90, 0, 180
cube.scale.y = 0.5

```

Fig. 3 Code example: positioning, rotating, and scaling a Ratscave Mesh by assigning new values to their correspondingly-named attributes

Manipulating perspective in 3D scenes: Ratscave’s camera class

Unlike in 2D graphics, where the screen’s pixels provide a natural coordinate space for positioning objects, a 3D scene is composed of 3D objects (“Meshes”) viewed from a given perspective (the “Camera”) which is projected down onto the 2D surface of the display. Positioning objects on-screen is further made intuitive by Ratscave’s Camera class, which functions similarly to virtual cameras in 3D modeling software. Besides being positioned and rotated to face an object, properties of the Camera’s intrinsic projection model (e.g. field of view, aspect ratio, and frustum cutoff thresholds, orthographic vs perspective projection) can be manipulated as well. To draw a Mesh from the perspective of the camera, it is bound by the user using a similar method as with the shader: using Python’s *with* keyword statement (Fig. 5).

Working with groups of Meshes: scenes and scene graphs

Once all meshes are all loaded, they can be collected together in any Python iterator object and passed to a

Scene, which is drawn using a `draw()` method. Scenes contain Mesh, Camera, and Light objects, which are applied automatically within the `draw()` call. Scenes can share objects between each other, making them useful, lightweight containers for different experimental conditions (Fig. 6).

Complex relationships of object positions can be specified via Ratscave’s simplified scene graph functionality by parenting objects to each other, allowing the experimenter to move sets of objects in a single call to the top-most parent. For example, a much-simplified solar system model could be arranged as follows (Fig. 7).

Integrating multiple rotation coordinate systems

Unlike 2D objects, there are multiple ways to format rotations in three-dimensional space, three of which are: euler rotations, rotation matrices, and quaternions. Euler rotations, sequential 2D rotations about three axes stored as XYZ coordinates, have the advantage of being intuitive to use and to set; a rotation about the X axis can be written as an angle in the X rotation coordinate. However, they also come with disadvantages; for example, they must be applied in the same order every time to achieve the same ending rotation (so mixing 3D modeling

```

win = psychopy.visual.Window()
text = psychopy.visual.TextStim(win,
“ABC”)
while True:
    win.clear()
    text.draw()
    win.flip()

```

```

win = pygame.window.Window()
text = pygame.text.Label(‘ABC’)
@win.event
def on_draw():
    text.draw()
pygame.app.run()

```

```

win = psychopy.visual.Window()
reader = ratscave.WavefrontReader(‘cube.obj’)
cube = reader.get_mesh(‘Cube’)
while True:
    win.clear()
    with ratscave.default_shader:
        cube.draw()
    win.flip()

```

```

win = pygame.window.Window()
reader = ratscave.WavefrontReader(‘cube.obj’)
cube = reader.get_mesh(‘Cube’)
@win.event
def on_draw():
    with ratscave.default_shader:
        cube.draw()
pygame.app.run()

```

Fig. 4: Demonstrating the flexibility and generality of Ratscave by drawing a cube in two different libraries: Psychopy (top) and Pyglet (bottom). On the left side is the code for drawing a 2D text stimulus for that library, with the right side showing the addition of a Ratscave cube

stimulus. Although the syntax of each library differs from each other in how they create a window and begin a draw loop, the Ratscave drawing code is the same and inserts cleanly into the existing code structure for each experiment

```

camera = rc.Camera()
camera.position.xyz = 1, 2, 3 # Move the camera
camera.projection.fov_y = 70 # Expand the vertical field of view to 70 degrees
camera.projection.aspect = 1.2 # Stretch the view to a 5/4 aspect ratio

with camera:
    cube.draw()

```

Fig. 5: Positioning a Ratcave Camera is done using the same interface as with Mesh objects, and applying it to a draw operation which is done with the same interface as Shader objects. Changing the Camera’s intrinsic

projection properties (e.g. field of view, aspect ratio, frustum section, orthographic vs perspective projection) is done through its projection attributes

programs and 3D rendering programs in different order can result in unequivocal rotations), and they are vulnerable to a phenomenon known “gimbal lock”, a situation in which certain rotations can make a given axis useless. Rotation matrices, 3x3 square matrices that describe Euler rotations that apply rotation transformations with a single dot product, always apply the rotations in the same order as bound by the rules of linear algebra. However, building rotation matrices by hand is cumbersome at best. Quaternions, a single rotation about an arbitrary 3-element vector, stored as WXYZ or XYZW coordinates, have the advantage of being compact, non-sequential and invulnerable to gimbal lock; however, they can be unintuitive in practice. OpenGL also requires a fourth variant, a model matrix, that has a 4x4 matrix format, whose sequential application is order sensitive. Finally, different users may prefer setting their rotations as degrees or radians.

To allow full flexibility between different rotation coordinate systems, Ratcave allows all Physical object (Meshes, Cameras, and Lights) rotations to be set using any rotation coordinate system, as well as providing conversion methods between them: for example, `to_quaternion()`, `to_euler()`, and `to_matrix()`, with options in each for setting rotation sequence and radian or degree units. This feature is, naturally, optional; by default, all rotations are specified as Euler coordinates as degrees.

Updating the data pipeline to the graphics card: uniforms, shaders, and vertex arrays

Ratcave uses modern OpenGL constructs from the ground up, rendering by passing data to graphics card-compiled

“shader” programs, rather than sending individual commands to the OpenGL state machine from Python itself. While this creates a two-language situation (Python programs on the CPU and shader programs in the GLSL language on the graphics card) that may initially seem complex, it represents a scalable solution that allows scientists to take advantage of each language’s strengths. In addition, this approach helps with creating performant 3D graphics applications in slower dynamic languages like Python, where high numbers of C library calls (common in legacy OpenGL 3D applications) can create a significant performance bottleneck; in fact, this library is used by our lab’s virtual reality system to render full 3D scenes through a multi-pass rendering pipeline at 360 fps (data not shown). Three different types of data are passed to the graphics card, with each one wrapped by Ratcave with a Pythonic interface: Vertex Arrays, Uniforms, and Textures.

Vertex arrays Meshes in 3D applications are composed of arrays of vertex coordinates, with each defining the endpoint of an edge or the boundary of a face on that Mesh. This data could be passed to the graphics card from Python point-by-point upon drawing (OpenGL’s ‘Immediate Mode’, used by many Python 2D graphics libraries), but this process can be made more efficient by sending the data as a single array using OpenGL’s VAO (Vertex Array Object) functionality and storing it on the graphics card itself. Sets of arrays (most commonly, a Mesh’s vertex, normal, and texture coordinate arrays) can be associated together via OpenGL’s VBO (Vertex Buffer Object), and then all that is needed is a single draw call when the actual rendering is performed. Since the data is

```

cube_scene = rc.Scene(meshes=[torus, cube], camera=cam)
monkey_scene = rc.Scene(meshes=[torus, monkey], camera=cam)

with rc.default_shader:
    cube_scene.draw()

```

Fig. 6: Collecting Meshes in a scene. Because scene objects hold Meshes, cameras, and lights, every Mesh they contain can be drawn in a single `Scene.draw()` call

```

sun, venus, earth, moon = obj_reader.get_meshes(['Sun', "Venus", 'Earth', 'Moon'])
sun.add_children([venus, earth])
moon.add_parent(earth)
sun.rotation.y += 5 # Rotate everything about the sun's axis.

with rc.default_shader:
    for mesh in sun:

        mesh.draw()

```

Fig. 7: Building a scene graph. Meshes can be arranged in a tree-like parent/child collection using their `add_children()` and `add_parent()` methods, which sets their position and rotation to be relative to the parent.

This tree can also be traversed by iterating over the top-most parent can then iterated over in a for loop

already present on the graphics card, the operation is much more efficient. Ratcave pipes vertex array data using VAOs and VBOs on Meshes and uses pointers to associate NumPy arrays to the graphics card array data. The result is that users can pass NumPy arrays to Meshes and even edit them like normal NumPy arrays, while Ratcave updates the data on the graphics card as needed (Fig. 8). Using this approach, over 30,000 vertices can be streamed in real-time to the graphics card and rendered onscreen at 60 Hz, a performance level surpassing the needs of most behavioral research studies (Supplementary Video 2).

Uniform data Any data that can be associated with drawing a Mesh, whether its position, color, or even custom properties that are stimulus-specific can be received by the graphics shader as so-called “Uniform” data, meaning that it has the same value across all vertices of the Mesh. Uniform data can be single values, arrays, or even matrices. Most Ratcave

objects have a dictionary-like *uniforms* attribute that automatically links, transforms, and sends its set of uniform data whenever the object is bound (whether by calling its *bind()* method or using Python’s *with* keyword) or drawn using a *draw()* method. Ratcave builds and maintains many uniforms automatically, including the matrices associated with positioning, viewing, and projecting objects on-screen (the Model matrix, which describes an object’s position; the View matrix, which describes the camera’s position; and the Projection matrices, which describes the camera’s lens characteristics) and adds some extra uniforms for coloring and lighting a mesh (Fig. 9).

Besides enabling full customizability of all stimuli, using uniforms helps increase performance of OpenGL rendering pipelines in Python. Legacy OpenGL typically requires five library calls to position an object on-screen, even if the mesh’s position is unchanged from the last rendered frame. As the number of objects to be rendered increases, the computational

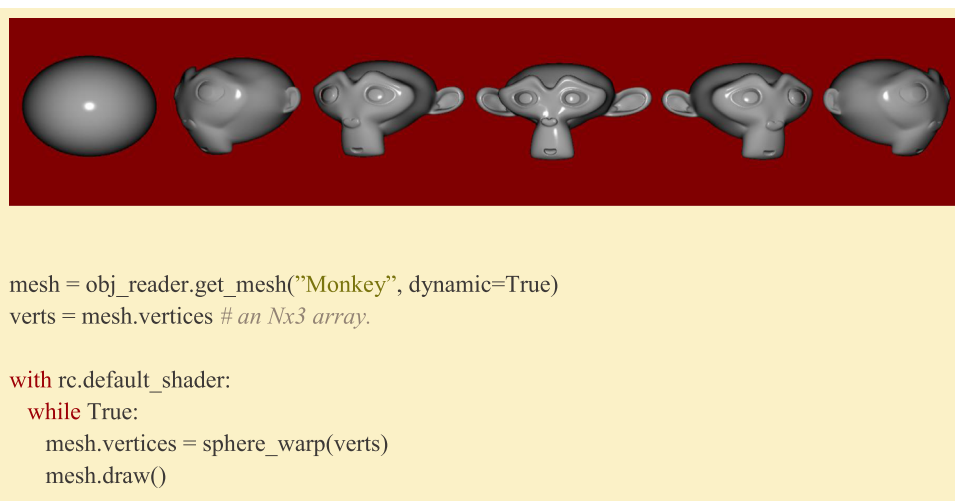


Fig. 8: Warping a Mesh in real-time. If the Mesh is *dynamic* (meaning its data can be updated between frames), their $N \times 3$ vertex array data can be accessed and manipulated like any NumPy array, and Ratcave will handle

all graphics card buffering for the user. In this example sequence, a mesh is interpolated between its original coordinates (a Monkey primitive) and a sphere. For a video example, please see Supplementary Video 2

```

torus.uniforms['diffuse'] = 1, 0, 0
monkey.uniforms['diffuse'] = 0, 0, 1

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

with rc.default_shader:
    scene.draw()

```

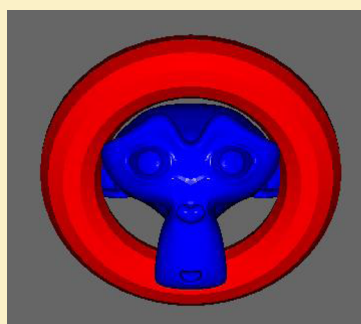


Fig. 9: Creating and updating uniform values is done via the *uniforms* dictionary. In this example, updating the ‘diffuse’ property sends a 3-element vector to the shader upon drawing. If the shader program has a

‘diffuse’ variable declared, it will then use the supplied value. Ratcave’s default shader uses ‘diffuse’ to set the diffuse lighting color

cost scales linearly. Utilizing shaders, on the other hand, requires only a single library call, which sends a single matrix (a “Model” matrix) to the shader. Ratcave makes calculating these matrices straightforward by automatically updating the model matrix whenever a stimulus’ position, rotation, or scale attributes are modified. It also saves these transformations intelligently, “lazily” updating the matrix (via an Observer software design pattern) only when needed, using the optimized Numpy array library. Sending the matrix to the shader is done when the “draw()” method is called. Similar steps are done for the Camera’s view matrix and projection matrix (Fig. 10).

Texture data Ratcave also supports mapping image data to 3D meshes using a technique called “UV mapping”, named after the coordinate system used for specifying the rows and columns of an image (“u” and “v”). With this method, each Mesh’s vertex is associated with an image and its 2D coordinate (called a “texture” and a “texture coordinate”, respectively), and the image’s pixels are interpolated between the vertices, effectively stretching the 2D image across the Mesh’s surface.

The following three steps are needed to display this data using OpenGL: the texture data must be formatted and passed to the graphics card as an OpenGL Texture, it

must be bound, and it must have an associated uniform name in order to link a given texture with a given rendering step in the shader. All of these steps are performed by Ratcave’s Texture objects, taking the OpenGL ID from an image loaded using another image processing library (making it compatible with a wide variety of image processing software), or loading it from an image file using Pyglet’s image module. If it is appended to a Mesh’s textures list attribute, it is automatically bound and its uniforms sent upon the Mesh’s *draw()* method call. Any number of textures of any OpenGL type (e.g. color vs depth textures, 2D vs 3D Textures, 2D vs Spherical vs 3D texture coordinates) can be appended to a Mesh, allowing any image algorithm to be implemented on the graphics card online, during stimulus rendering (Fig. 11, Supplementary Video 3).

Writeable textures: building deferred rendering pipelines

OpenGL’s Framebuffer objects allow users to create virtual windows that redirect a rendered image to a texture saved in memory on the graphics card instead of the display. This creates opportunities to build “deferred” rendering pipelines, in which several different image processing

# Legacy OpenGL:	# Modern OpenGL:	# Ratcave
glPushMatrix()	model_matrix = [[1, 1, 0, 7],	cube.position.xyz = 7, 8, -2
glScale(1, 1, 1)	[-1, 1, 0, 8],	cube.rotation.x = 90
glRotate(90, 0, 0)	[0, 0, 1, -2],	cube.scale.xyz = 1
glTranslate(7, 8, -2)	[0, 0, 0, 1]]	
glPopMatrix()		# Done during cube.draw()
	glUniformMatrix4fv(model_matrix)	cube.uniforms.send()

Fig. 10: Comparison of Model Matrix computation and sending to OpenGL between legacy OpenGL, modern shader-based OpenGL, and Ratcave’s interface to modern OpenGL

```

tex = rc.Texture.from_image('oak.png')
monkey.textures.append(tex)

with rc.default_shader:
    monkey.draw()

-----

tex = rc.Texture.from_image('oak.png')

with rc.default_shader, tex:
    monkey.draw()

```



Fig. 11: Images can be read in as OpenGL textures. To bind them, they can be either appended to Meshes in order to automatically activate them when the mesh is drawn (top-left), or activated explicitly with the *with*

statement before drawing (bottom-left). Each approach has advantages for different experimental paradigms

algorithms are run and saved for a final step that combines the previous images into more complex and dynamic images (Fig. 12).

Deferred rendering is an important technique for CAVE-style virtual reality systems, which project a 360-degree rendering of the virtual environment onto projection screens surrounding the subject (Del Grosso et al., 2017). This is done in two steps: first, the full virtual environment is saved to six textures, each representing a different viewpoint of the scene from the perspective of the subject. Then, these textures are drawn onto meshes representing the screen from the perspective of the video projector and rendered onscreen. When all of these renders are counted up, a total of seven rendering passes (six renders-to-texture and one render-to-screen) are performed. Deferred rendering is also useful for rendering shadows, where the scene is first rendered to texture from the perspective of a light source in order to calculate where

the shadow should appear from the camera’s perspective for the final render-to-screen pass. Any multi-pass rendering algorithm can be done in Ratcave by simply binding an FBO object and then drawing a scene (Fig. 2).

OpenGL shader programs

Besides enabling full customization of graphics rendering, OpenGL’s programmable pipeline speeds up graphics applications by allowing users to off-load calculations to the graphics card through “shader” programs written in a C-like language called GLSL. Each program is made up of two smaller programs: one that is run for each mesh vertex (the “Vertex Shader”, Fig. 13), which is most commonly used for positioning something onscreen, and one that is run for each pixel of the display (the “Fragment Shader”, Fig. 14,

```

stars = obj_reader.get_mesh('Grid3D')
stars.drawmode = rc.POINTS

monkey = obj_reader.get_mesh('Monkey')

fbo = rc.FBO(texture=rc.Texture())
with rc.default_shader:
    with fbo:
        stars.rotation.z += 10.
        stars.draw()
    with fbo.texture:
        monkey.draw()

```

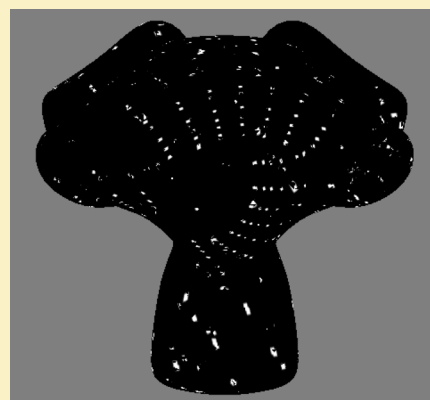


Fig. 12: An example of two-pass rendering using Framebuffer objects. First, the stars Mesh is drawn and the resulting image saved to a Framebuffer-attached texture. This texture then becomes the texture

bound to the monkey Mesh, producing an unusual effect: a rotating star field mapped on a monkey head

```

layout(location = 0) in vec4 vertex;
uniform mat4 projection_matrix, view_matrix, model_matrix;

void main(){
    gl_Position = projection_matrix * view_matrix * model_matrix * vertex;
}

```

Fig. 13: A GLSL Vertex Shader. This program takes the vertex data (the first array in a Ratcave mesh, specified as location 0) and the various matrices (received as uniforms), and calculates the onscreen position by calculating their dot product, outputting the onscreen position

Supplementary Video 3), which is most commonly used for setting the color and lighting properties of the image.

Ratcave Shader objects compile these programs when needed and run them when bound, as seen in previous examples (Fig. 15). Because shader programs can be mixed and matched, and because they run on all hardware, platforms, and graphics engines, these short programs are useful formats for a wide variety of visual stimuli.

Discussion

Ratcave is useful for Python-using scientists who want both 3D graphics support in their existing experiment scripts and a quick introduction to computer graphics in OpenGL, smoothing the learning curve toward using more advanced, feature-complete 3D graphics software. Because Ratcave provides access to the more-advanced OpenGL rendering methods (e.g. framebuffers and custom shaders), researchers can also use Ratcave to rapidly prototype complex visual stimuli. These GLSL shader programs run directly on the graphics card (e.g. the Gabor patch stimulus in Fig. 14, often used for vision studies), making them cross-compatible between Python programmers using

Ratcave, Matlab programmers using Psychtoolbox, and 3D graphics engine users (e.g. Unity3D, Unreal Engine 4, and Panda3D), a property that has interesting potential for future stimulus libraries.

Future development of Ratcave will include support for more 3D-object file formats, stereoscopic stimuli, animations, logging-event hooks, and more powerful scene graph functionality. Ratcave’s design also cleanly separates 3D object manipulation and GPU interaction, making it possible to extend support to other low-level graphics libraries (e.g. WebGL or Vulkan) through an adapter programming design pattern, should future psychology stimulus software in Python use these graphics libraries themselves.

With the extension package described in this paper, psychology researchers can add and manipulate 3D stimuli with minimal code in a familiar programming environment. Researchers can simply drop-in their 3D stimuli into experiment scripts that support their input hardware and experimental design managers. Ratcave is easy to use, and the most-used operations on 3D models (importing data from file, building a mesh, manipulating its position, rotation, and scale, change its lighting, and drawing to the screen) can be done with single lines of code. As such, it makes for a good addition to the existing Python psychology software ecosystem.

```

in vec2 pos;
out vec4 final_color;
uniform vec2 mean,
uniform float width, std, theta;

void main() {
    float level = sin(pos.x / width + theta) / 2.;
    level *= exp(-.5*pow(pos.x-mean.x, 2)/pow(std, 2));
    level *= exp(-.5*pow(pos.y-mean.y, 2)/pow(std, 2));
    final_color = vec3(level + .5);
}

```

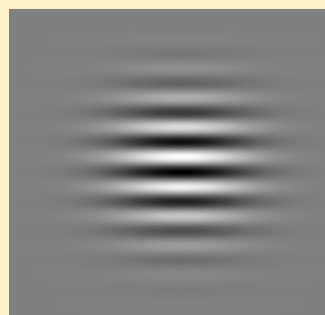


Fig. 14: A GLSL Fragment Shader. This program is run for every pixel where a mesh is present. It takes uniform data (“width”, “mean”, etc) and outputs the RGB (“final_color”). This program calculates a gabor patch

based on the screen position of a pixel (‘pos’) and the uniform parameters given by the Ratcave program


```

shader = rc.Shader.from_files(vert='position_objects.vert', frag='gabor.frag')
gabor = obj_reader.get_mesh('Plane', position=(0, 0, -1))
gabor.uniforms['theta'] = 0.5
gabor.uniforms['mean'] = 0, 0

with shader:
    gabor.draw()

```

Fig. 15: Loading custom shader files into a Ratcave Shader object and using it to draw a stimulus. Uniform values get sent to the shader when the draw() method is called, thereby connecting shader program variables to Python variables

Author note This work was supported by the Deutsche Forschungsgemeinschaft [grant numbers DFG RTG 2175, DFG Priority Program 1392, DFG Priority Program 1665], the Bundesministerium für Bildung und Forschung [grant number 01GQ0440], and the Munich Cluster for Systems Neurology [grant number SyNergy EXC 1010].

We thank Andrey Sobolev and Dustin Fetterhoff for their insights in testing and implementing the Ratcave library in their research, and for their comments and suggestions during the writing of this manuscript. We also thank Andrew Straw for organizing a graphics course using Ratcave's development, which proved invaluable for improving and refining the codebase and documentation. We thank Mohammad Bashiri and Aleksandra Teska for their programming contributions to Ratcave software, including the additions of stereoscopic rendering and improving the scene graph implementation.

Compliance with ethical standards

Conflict of interest The authors state no conflicts of interest with the findings presented in this work.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

- Brainard, D. H. (1997). The psychophysics toolbox. *Spatial Vision*, 10, 433–436.
- Del Grosso, N. A., Graboski, J. J., Chen, W., Hernández, E. B., & Sirota, A. (2017). Virtual reality system for freely-moving rodents. *bioRxiv*, 161232. <https://doi.org/10.1101/161232>
- Kötter, R. (2009). A primer of visual stimulus presentation software. *Frontiers in Neuroscience*, 3(2), 163–164. <https://doi.org/10.3389/neuro.01.021.2009>
- Krause, F., & Lindemann, O. (2013). Expyriment: A Python library for cognitive and neuroscientific experiments. *Behavior Research Methods*, (October). <https://doi.org/10.3758/s13428-013-0390-6>
- Mathôt, D., & Theeuwes, J. (2012). OpenSesame: An open-source, graphical experiment builder for the social sciences. *Behavior Research Methods*, 314–324. <https://doi.org/10.3758/s13428-011-0168-7>
- Peirce, J. W. (2007). PsychoPy-Psychophysics software in Python. *Journal of Neuroscience Methods*, 162(1-2), 8-13. <https://doi.org/10.1016/j.jneumeth.2006.11.017>
- Straw, A. D. (2008). Vision Egg: An open-source library for realtime visual stimulus generation. *Frontiers in Neuroinformatics*, 2(November), 1–10. <https://doi.org/10.3389/neuro.11.004.2008>

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.